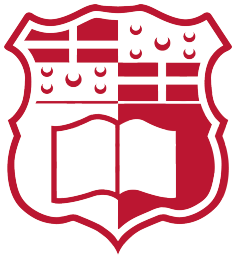


A Study on Human Pose Classification Using Convolutional Neural Networks and Tensor Regression

Andrew Spiteri

Supervised by Dr Maria Kontorinaki

Co-supervised by Dr Mark Anthony Caruana

Department of Statistics & Operations Research

Faculty of Science

University of Malta

March, 2025

A dissertation submitted in partial fulfilment of the requirements for the degree of M.Sc. in Statistics.

L-Università ta' Malta

Declaration by Postgraduate Students

(a) Authenticity of Dissertation

I hereby declare that I am the legitimate author of this Dissertation and that it is my original work.
No portion of this work has been submitted in support of an application for another degree or qualification of this or any other university or institution of higher education.
I hold the University of Malta harmless against any third party claims with regard to copyright violation, breach of confidentiality, defamation and any other third party right infringement.

(b) Research Code of Practice and Ethics Review Procedures

I declare that I have abided by the University's Research Ethics Review Procedures.
As a Master's student, as per Regulation 77 of the General Regulations for University Postgraduate Awards, I accept that should my dissertation be awarded a Grade A, it will be made publicly available on the University of Malta Institutional Repository.

Faculty/Institute/Centre/School	Faculty of Science
Degree	M.Sc. in Statistics
Title	A Study on Human Pose Classification Using Convolutional Neural Networks and Tensor Regression
Candidate (Id.)	Andrew Spiteri (49497M)
Signature of Student	_____
Date	September 10, 2025

Acknowledgements

I would like to express my deep gratitude and appreciation to Dr Maria Kontorinaki and Dr Mark Anthony Caruana for their valuable and constructive suggestions during the planning and development of this dissertation. I learned a lot during the course of this degree over the last few years from their extensive knowledge on many topics, and I am very grateful for their patience. I also would like to thank all staff members in the Department of Statistics and Operations Research and the Faculty Board for their support and patience throughout this degree.

Finally, I want to thank my family and friends for their invaluable support and encouragement which motivated me to work on this dissertation.

Abstract

Folk dances are a source of a country's history and traditions, and their documentation and analysis are important for their preservation. Human Pose Classification (HPC) covers the classification of human poses through body part detection from image, video, or measurement data. Folk dances are defined as a repeated sequence of main choreographic steps. The classification of the main choreographic steps of a dance falls under choreographic modeling, which is an application of HPC in dance. In this paper, we explore appropriate methods for choreographic modeling using image data, where we cover Convolutional Neural Networks (CNNs) and Tensor Regression (TR). CNNs are well-known in image classification since they are constructed more efficiently than Artificial Neural Networks (ANNs) for working with image data. TR is an extension of the regression problem using tensor representations, which would be more appropriate than classical regression for use with image data. We do a comparison study between CNNs and TR on a dance dataset, aiming to predict all poses over two trials. The first trial uses a standard training-test split across all dancers, while the second follows a leave-one-out approach, training on all but one dancer and testing on the excluded dancer. Both models correctly predicted all poses in the first trial, whereas the second trial proved more complicated, with fewer poses classified. CNNs yielded higher performance metrics compared to TR, where TR generally had worse results. However, CNNs contained significantly more parameters than TR, leading to signs of overfitting in the second trial.

Contents

List of Figures

List of Tables

List of Abbreviations

ADHD Attention Deficit Hyperactivity Disorder
ALS Alternating Least Squares
AdamW Adam with decoupled Weight decay
ANN Artificial Neural Network
ARMA Autoregressive Moving Average
ASHA Asynchronous Successive Halving Algorithm
bpp bits per pixel
CE Cross-Entropy error
CP Candecomp/Parafac
CNN Convolutional Neural Network
CORCONDIA Core Consistency Diagnostic
EU European Union
GLM Generalized Linear Model
GPU Graphics Processing Unit
HPC Human Pose Classification
HPE Human Pose Estimation
ICH Intangible Cultural Heritage
ILSVRC ImageNet Large Scale Visual Recognition Challenge
LSTM Long Short-Term Memory
ML Machine Learning
MLP Multilayer Perceptron
MLE Maximum Likelihood Estimation
MRI Magnetic Resonance Imaging

MSE Mean Squared Error
OGD Online Gradient Descent
PCA Principal Component Analysis
QMD Quadratic Mean Differentiable
ReLu Rectified Linear unit
RMSE Root Mean Squared Error
RGB Red-Green-Blue
SGC Strong Growth Condition
SGD Stochastic Gradient Descent
SGDW SGD with momentum using decoupled Weight decay
SKE Syrtos Kalamatianos Efthia
SME Syrtos Makedonikos Efthia
SSE Sum of Squared Errors
SVD Singular Value Decomposition
TE Trexatos Efthia
TR Tensor Regression
TRL Tensor Regression Layer
UAT Universal Approximation Theorem
VC Vapnik-Cervonenkis

List of Notations

Symbol	Meaning
$\langle \mathbb{X}, \mathbb{Y} \rangle$	inner product of tensors $\mathbb{X}$ and $\mathbb{Y}$
$\|\mathbb{X}\|$	Frobenius norm of tensor $\mathbb{X}$
$\mathbb{B}\mathbb{G}$	matrix product of two matrices $\mathbb{B}$ and $\mathbb{G}$
$\mathbb{D} \otimes \mathbb{E}$	Kronecker product of two matrices $\mathbb{D}$ and $\mathbb{E}$
$\mathbb{F} \odot \mathbb{G}$	Khatri-Rao product of two matrices $\mathbb{F}$ and $\mathbb{G}$
$\mathbb{X}_{(N)}$	mode-N matricization of $\mathbb{X}$
$\text{vec}(\mathbb{X})$	vectorization of $\mathbb{X}$
$\mathbf{a}_1 \circ \mathbf{a}_2$	outer product of two vectors $\mathbf{a}_1, \mathbf{a}_2$
$\text{rank}(\mathbb{X})$	rank of a tensor $\mathbb{X}$
$\text{rank}_N(\mathbb{X})$	column rank of $\mathbb{X}_{(N)}$
$\mathbb{X} \times_N \mathbf{J}$	N-mode product of tensor $\mathbb{X}$ with a matrix $\mathbf{J}$
$[\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$	CP decomposition of a tensor into factor matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$
$f_{ex}(\cdot), g_{ex}(\cdot)$	functions f and g for an example
$(\mathbb{F} * \mathbb{G})(x, y)$	convolution operation between tensors $\mathbb{F}$ and $\mathbb{G}$
B	batch size
K	total number of hidden layers
L	total number of labels
O	number of dimensions of tensor
P	regularization term
Q	number of iterations
g	number of hidden layers
h	number of neurons in a hidden layer
G	padding

s	stride length
b	bias/intercept
n, p	sample of n vectors of observations and p variables in data matrix $\mathbf{X}$
Z	CP decomposition rank Z
$a(.), b(.), c(.)$	functions determined by the chosen exponential-family member
$g(.)$	invertible link function
E^q	error function at iteration q
l_i	observed label for sample i
o_i	predicted label for sample i
m_k	number of feature maps / filters of k^{th} layer
R_k, C_k, D_k	number of rows, columns & channels of tensor, respectively, of k^{th} layer
P_k, Q_k, S_k	number of rows, columns & channels of tensor, respectively, of k^{th} layer
b_{vk}	bias of the v^{th} neuron in the k^{th} hidden layer
b_{jk}	bias of the j^{th} feature map in the k^{th} layer
z_{jk}	output of the j^{th} neuron in the k^{th} hidden layer
w^q_{uvk}	weight connecting the u^{th} neuron in layer $k-1$ to the v^{th} neuron in layer k for the q^{th} iteration
g^q_{ajk}	gradient of the error function w.r.t. w^{q-1}_{ajk} at iteration q
h^q_{ajk}	sum of the squares of the gradients w.r.t. w^q_{ajk} up to iteration q
$w^{p,q,s}_{ajk}$	weight assigned at position (p, q, s) in the a^{th} feature map of the $k-1^{th}$ layer for the j^{th} output feature map of the k^{th} layer
$v^{x,y,z}_{jk}$	value of neuron at position (x, y, z) in the j^{th} feature map of the k^{th} layer
$x_{r_1 \ldots r_O}$	element $(r_1, \ldots, r_O)$ of an order O tensor $\mathbb{X}$
$p^{(1)}_i, \ldots p^{(L)}_i$	probabilities of the L labels to be assigned, add up to 1
$R_1, \ldots, R_O$	general dimensions of order O tensor
$\mathbf{l}$	column vector of observed labels
$\mathbf{o}$	column vector of predicted labels
$\mathbf{x}_i$	column vector of observations for sample i with p variables
$\mathbf{x}_{i,A}$	column vector of observations for sample i with p variables with bias absorption
$\mathbf{w}_{vk}$	column vector of weights for v^{th} neuron in the k^{th} hidden layer

$\mathbf{z}_k$	column vector of outputs for layer k
$\mathbf{z}_{k,A}$	column vector of outputs for layer k with bias absorption
$\mathbf{b}_k$	bias vector for layer k
$\mathbf{x}_{:r_2r_3}$, $\mathbf{x}_{r_1:r_3}$, $\mathbf{x}_{r_1r_2:}$	mode-1, mode-2 and mode-3 fibers of order 3 tensor $\mathbb{X}$
$\mathbf{w}_{1,z}, \ldots, \mathbf{w}_{O,z}$	rank-one components of CP decomposition
$\mathbf{z}$	vector of covariates
$\mathbf{X}$	data matrix
$\mathbf{W}_k$	weight matrix for layer k
$\mathbf{W}_{ajk}$	weight matrix connecting the a^{th} feature map of the $k-1^{th}$ layer with the j^{th} output feature map of the k^{th} layer
$\mathbf{W}_{k,A}$	weight matrix for layer k with bias absorption
$\mathbf{V}_{jk}$	matrix representing the j^{th} feature map of the k^{th} layer
$\mathbf{B}_{jk}$	bias matrix for the j^{th} feature map of the k^{th} layer
$\mathbf{W}_1, \ldots, \mathbf{W}_O$	factor matrices of order O tensor
$\mathbf{L}$	1-of-K encoded categorical variable
$\mathbf{J}_N$	Jacobian matrix for N
$\mathbf{I}$	identity matrix
$H(\mathbf{W}_1, ..., \mathbf{W}_O)$	Hessian matrix
$\mathbb{X}$	data tensor
$\mathbb{F}, \mathbb{G}$	tensors $\mathbb{F}$ (representing input image) and $\mathbb{G}$ (representing filter)
$\mathbb{B}_k$	bias tensor of the k^{th} layer
$\mathbb{G}^q$	matrix/tensor with all respective g^q_{ajk} entries at the q^{th} iteration, depending on shape of $\mathbb{W}^q_k$
$\mathbb{H}^q$	diagonal matrix/tensor containing the diagonal entries h^q_{ajk}, depending on shape of $\mathbb{W}^q_k$
$\mathbb{M}^q_1$	matrix/tensor of exponentially decaying average of past gradients, depending on shape of $\mathbb{W}^q_k$
$\widetilde{\mathbb{M}}^q_1$	bias corrected estimate of $\mathbb{M}^q_1$
$\mathbb{M}^q_2$	matrix/tensor of exponentially decaying average of past squared gradients, depending on shape of $\mathbb{W}^q_k$
$\widetilde{\mathbb{M}}^q_2$	bias corrected estimate of $\mathbb{M}^q_2$
$\mathbb{W}^q_k$	parameter block of the k^{th} layer for the q^{th} iteration, matrix/tensor depending on layer type and number of filters/batch sizes
$\mathbb{V}_k$	output feature maps tensor of the k^{th} layer
$\mathbb{X}_{r_1::}$, $\mathbb{X}_{:r_2:}$, $\mathbb{X}_{::r_3}$	horizontal, lateral and frontal slices of order 3 tensor $\mathbb{X}$

$\mathbb{F}(\mathbf{W}_1, ..., \mathbf{W}_O)$	Fisher information matrix
$\mathbb{W}$	parameter tensor of order O tensor
$\mathbb{H}$	core tensor of order O tensor
β	constant that controls decay rate, includes β_1 and β_2
ρ	fixed probability value
σ	activation function
ξ_1, ξ_2	vectors $\xi_1, \xi_2 \in \mathbb{R}^m$ for any m
ω	epoch number
γ	learning rate
Δ	bounded set
ε	scalar/tensor of arbitrarily small constants
τ	scale parameter
$\theta(.), \phi$	natural and dispersion parameters
$\mu(.)$	conditional mean
$\eta(.)$	linear predictor
$\boldsymbol{\beta}$	weight vector, includes $\boldsymbol{\beta}_1$ and $\boldsymbol{\beta}_2$
ζ	weight vector for additional covariates $\mathbf{z}$
Π	permutation matrix
γ^q	step size, i.e., decayed learning rate at iteration q
λ_{LR}	regularization parameter
λ_{WD}	rate of weight decay
$\mu_{j(k-1)}, \sigma^2_{j(k-1)}$	mean and variance value of the mini-batch for the j^{th} feature map inputted to the k^{th} layer
$\boldsymbol{\mu}_{k-1}, \boldsymbol{\sigma}^2_{k-1}$	mean and variance m_{k-1}-vectors containing values of $\mu_{j(k-1)}$ & $\sigma^2_{j(k-1)}$, respectively
$\boldsymbol{\mathbb{\mu}}_{k-1}, \boldsymbol{\mathbb{\sigma}}^2_{k-1}$	mean and variance tensors containing values of $\mu_{j(k-1)}$ & $\sigma^2_{j(k-1)}$, respectively
δ_{jk}, τ_{jk}	tensors of scale and shift parameters for batch normalization that will be tuned during training
$\boldsymbol{\theta}_i$	vector of regression parameters conditioned on $\mathbb{X}_i$ and l_i
$\nabla \eta(\mathbf{W}_1, ..., \mathbf{W}_O)$	gradient of systematic part
$\nabla l(\mathbf{W}_1, ..., \mathbf{W}_O)$	score function/vector
$\mathcal{R}, \mathcal{C}$	ordered sets
$\mathcal{D}$	observed dataset
$\mathcal{I}$	index set
$\mathcal{T}$	translation group

$\mathcal{O}$	Bachmann-Landau notation
$\mathcal{Y}$	dictionary
$B_1(\mathcal{Y})$	closure of the convex, symmetric hull of $\mathcal{Y}$
$\|f\|_{\mathcal{K}(\mathcal{Y})}$	norm defined by the gauge of $B_1(\mathcal{Y})$
$\mathcal{K}(\mathcal{Y})$	subspace of Δ known as the variation norm
$\mathcal{Z}$	Banach space

1

Introduction

The challenge of understanding human behaviour in images and videos has gained considerable attention over the last couple of decades, specifically in regard to the identification of human poses through images or videos. Recognising human actions is an essential task in real-world situations and has been applied in various fields. This includes video surveillance, sports motion analysis, choreographic modeling, and real-time monitoring of patients. *Human Pose Estimation* (HPE) and *Human Pose Classification* (HPC) represent two of the most fundamental and challenging problems for understanding human behaviour in Computer Vision. HPE is defined as the problem of identifying the location of human joints in images or videos. In contrast, HPC is defined as the problem of identifying and classifying a human body posture from various possible poses through images, videos or measurement data. This dissertation will mainly focus on HPC using image data.

1.1: Literature Review on HPC

In general, much work has been done on HPC research. Elnaggar et al. (2020) proposed a method for improved monitoring of in-bed body postures using wearable sensors for HPC to check for premature signs of musculoskeletal disorders. Komarudin et al. (2021) proposed a paper discussing the ability of drones to detect human pose using a Machine Learning (ML) algorithm for HPC. Li et al. (2024) introduced a deep learning model that reconstructs 3D poses from 2D images and then uses a Long Short-Term Memory (LSTM) network to capture temporal dependencies to classify 3D poses. Yan et al. (2025) applied a novel subject-dependent HPC approach based on Tensor Regression (TR) to classify human postures. Some applications utilise a combination of HPE and HPC, such as in Ardiyanto et al. (2014), where in the case of a fallen person monitor-

ing and rescue scenario using a mobile service robot, HPE would be used to estimate the position of the limbs of people who have fallen, and HPC would be used to recognise the human state, including their pose, using onboard sensors. In addition, Srinivasan et al. (2010) described an intelligent visual surveillance system for identifying parts and poses of the human body using visual information from a list of possible poses.

With regards to HPC for dance data, there has also been much work in this area of research. Raptis et al. (2011) developed a system for the live classification of dance postures from skeleton animation data by extracting and processing joint movement information. Rallis et al. (2019) used a modified LSTM to estimate dancer poses using 3D skeleton data processing. Priya (2019) utilised an ML algorithm on different poses from several action sequences for Karate martial arts and Bharanthanatyam dance poses. Bakalos et al. (2019) proposed a method to identify and classify dance poses using Convolutional Neural Networks (CNNs). Rallis et al. (2020) integrated an Autoregressive Moving Average (ARMA) filter into a conventional CNN architecture for HPC and experimented on real-life dance sequences, which outperformed traditional deep learning models. Makantasis et al. (2021) proposed a tensor-based neural network that processes spatiotemporal data for HPC, which uses TR, where they experimented using Greek folk dance data. Tragiannis et al. (2023) studied the performance of many different classifiers to identify dance steps in a dataset containing videos of traditional Greek dances performed by several dancers.

1.2: Problem Description

The dataset used in this dissertation originates from a European Union (EU) project related to the digitalization of Intangible Cultural Heritage (ICH) for the documentation of folk dances. Folk dances are traditional dances created by people, reflecting the life of the people of a particular country or region. The protection of folk dances is a major task for ICH, as they are representative of the customs and uniqueness of the country.

The folk dances of the dataset are of Greek origin and differ in style and interpretation based on their region of origin. A popular Greek folk dance known throughout Greece and Cyprus that varies region by region is the *Syrtos* folk dance. The Syrtos folk dance originates from Crete and got its name from the ancient Greek word "syro", which means 'drag' in Greek. As such, it was named Syrtos to describe how it is danced, that is, from dragging, from the contact of the feet on the ground (Filippidou and Gialiti, 2022). Examples of regional variants of the Syrtos folk dance include Syrtos-Kalamatianos, which originates from Kalamata, and Syrtos-Makedonikos from Macedonia. Other examples of Greek folk dances include *Trehatos*, which originates from the village Neo-

chorouda in Thessaloniki and means "running" in Greek. Most Greek folk dances can be danced either in a line or in a circle. When a folk dance is danced in a line, we would say that it is the "*Efthia*" variation of that dance, where "Efthia" means 'straight'. On the other hand, if a folk dance is danced in a circle, then it would be the "*Kikliko*" variation of that dance, where "Kikliko" means 'round' or 'circular'.

The whole dataset consists of the *xyz*-coordinates and Red-Green-Blue (RGB) images of three professional dancers, where each dancer has different corresponding folk dances. Both Rallis et al. (2019) and Makantasis et al. (2021) used this dataset. The *xyz*-coordinates and an RGB image of 25 skeletal joints are identified and monitored by a Kinect-II sensor at a rate of 30 measurements per second. The recordings of the dances for each dancer range from anywhere around 10 to 15 seconds. In this dissertation, we will use the data related to three folk dances, namely the *Syrtos Kalamatianos Efthia* (SKE), *Syrtos Makedonikos Efthia* (SME), and *Trexatos Efthia* (TE) dances, and we will only use the images of this dataset.

There are many uncontrollable factors that can add complexity to the identification of poses between different humans. These include the differences in body dimensions, clothing, and appearance. Also, there would be problems that come with trying to identify human poses through images or videos. Such issues include having cluttered image backgrounds, occlusions, light variations, and viewpoint variations in regard to the sensor. The Kinect-II sensor is a cheap and effective way of achieving real-time 3D skeleton tracking. Although the Kinect-II sensor is very effective for working in radiant and dark environments (hence eliminating the problem of light variation), its capabilities remain confined since it is specifically designed to monitor only the users front, thereby making it difficult to differentiate between the front and back views. Also, the movement range of the sensor is limited to approximately 0.7 – 6m.

During the last decade, significant progress has been made in the multimodal recording of spatiotemporal features of the human body using motion capture systems such as the Wii remote, the Leap, and the Kinect sensor. The use of multimodal data is essential to observe a phenomenons progress over time continuously. A clear example of this is in dancing since any particular dance can be categorised as a set of main choreographic steps that are performed in sequence with one another. At each measurement taken, the corresponding label of the depicted human pose of the dancer is recorded along. Dancing experts manually annotated these pose labels. In total, there are seven unique poses among the three dances, where each pose would be repeated multiple times throughout the dance. The aim of our application will be a classification problem, where we will try to predict the dancers poses in the images. We will go into more detail on these seven poses in Chapter 5.

1.3: Aims and Objectives

For the preservation of folk dances, choreographic modeling, which is used to recognize the main choreographic steps (or poses) in a dance, is an important task for folk dance modeling for ICH (Rallis et al., 2020). Choreographic modeling can be interpreted as an application of HPC in dance. Our aim in this dissertation is to explore appropriate statistical and ML techniques for choreographic modeling using our dance data; specifically, we will discuss CNNs and TR.

Multidimensional arrays, which are known as *tensors*, are regarded as natural representations of image data and preserve their structural information compared to traditional vector forms (Lee et al., 2024). The advance in sensing technology and the continued research towards ML techniques has enabled the development of methods that can help circumvent the problems related to identifying the main choreographic steps of a dance. Compared to Artificial Neural Networks (ANNs), CNNs are more appropriate for use with image data (represented as tensors) and build the network architecture in a more sensible way, which helps to reduce the complexity of the model. CNNs are broadly known for their great success in solving problems in HPC and choreographic modeling (Bakalos et al. (2019), Rallis et al. (2020), Tragiannis et al. (2023)).

A drawback of ANNs and CNNs is the large amount of data needed to train them to accurately predict unseen data (Gril et al., 2022). Traditional regression models use vectors as covariates, which would not be suitable with tensor data (Guo et al., 2012). TR is an extension of the regression problem using tensorial representations and has seen use as an alternative to neural networks in situations with a smaller amount of data (Gril et al., 2022). Although TR is not as well-known as CNNs for HPC, there have been applications of TR on HPC and choreographic modeling (Makantasis et al. (2021), Yan et al. (2025)).

1.4: Structure of Dissertation

This dissertation is comprised of six chapters, covering ANNs, CNNs and TR as well as an application of CNNs and TR to the considered dataset.

Chapter 2 presents a summary of the theoretical framework of ANNs, including ANN architecture and optimization algorithms. We will review the notation used for ANNs and CNNs and cover ANNs in a fully supervised framework. We also cover several regularization methods that are used in other chapters.

Chapter 3 presents the theoretical framework of CNNs. First, we will cover tensors and how images can be represented as them, where we will then cover the convolution

operation in terms of tensors. Then, we will translate most of the material covered in Chapter 2 for CNNs, i.e., covering the general architecture of a CNN, optimization algorithms, and the Universal Approximation Theorem (UAT) for CNNs. Finally, we will go into a literature review covering some popular CNN architectures and some performance metrics that will be used in Chapter 5.

Chapter 4 presents the theoretical framework of TR. We will initially discuss tensor notation and algebra, which will be needed later on in the chapter. From there, we move on to tensor decomposition algorithms and TR as seen under a Generalized Linear Model (GLM) framework, where we briefly define GLMs in vector space and extend them to tensor space. We define the multinomial TR model originally proposed by Cao et al. (2022) that we will use in Chapter 5, and go into the models parameter estimation by MLE.

Chapter 5 will explore the CNN and TR applications using the theory described in previous chapters. We will first describe the dance dataset in more detail and then describe the objective we will aim to achieve in the applications. This objective will be composed of two trials that we will aim to achieve. We will then explore the details of the CNN and TR applications, comparing and discussing our results.

Chapter 6 provides the concluding remarks and suggestions for future research on this topic.

2

Artificial Neural Networks

As mentioned in Chapter 1, we aim to explore CNNs, and we will introduce the material on ANNs in this chapter without going into too much detail. CNNs are a specialized type of ANNs, so understanding the basic architecture of ANNs and how they are trained helps us understand how CNNs function. Note that we will only be going through the material on ANNs, assuming a classification task at hand.

Section 2.1 goes into an overview of terminology and notation for ANNs, while Section 2.2 goes into some detail on the backpropagation algorithm and optimization algorithms, specifically gradient descent. Section 2.3 briefly mentions the UAT for ANNs. Finally, Section 2.4 describes the case of overfitting for ANNs and how to check for it while also presenting several regularization methods that are used to help reduce the chance of overfitting.

2.1: Terminology and Notation used for ANNs

ANNs are nonlinear mapping structures capable of solving pattern recognition tasks such as regression and classification. They generally consist of many simple computing elements known as *neurons* that are organized into *layers* and interconnected in often complex ways, enabling them to approximate any complex nonlinear function (see Figure 2.1). Their emergence was brought about by researchers initial interest in mimicking the behaviour of biological nervous systems. ANNs are popular in many industries, including marketing, medical diagnosis, engineering, and security. We will now go on to cover the different types of layers in ANNs and the typical ANN architecture in the coming sections.

2.1.1: Overview of the ANN Architecture

The layers of an ANN can be separated into three categories, namely the initial layer, known as the *input* layer; the last layer, known as the *output* layer; and the middle layers, known as the *hidden* layers (see Figure 2.1). Hidden layers are not necessarily always present, and there can be multiple hidden layers. Figure 2.1 represents a generic ANN, where all neurons in a given layer are connected to all neurons in the subsequent layer, and there are no connections between neurons within the same layer. Also, not all layers would necessarily have the same number of neurons. An ANN with two or three layers is known as a *shallow*, and ANNs with over three layers are *deep* ANNs. The *depth* of an ANN represents the total number of its layers.

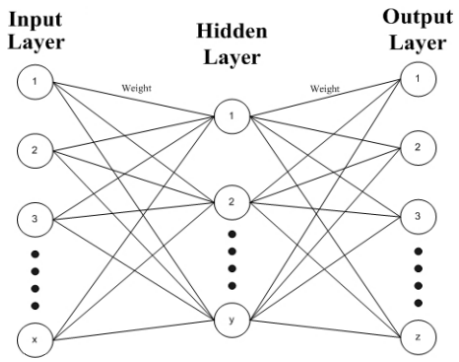


Figure 2.1: A typical ANN with one hidden layer (Tanty and Desmukh (2015)).

For an ANN, information flows initially from the neurons of the input layer to the output layer (if there are no hidden layers) or to the first hidden layer. From the first hidden layer, the information flows to the second hidden layer, and this process repeats depending on the number of hidden layers, ultimately reaching the output layer. This is referred to as the *forward pass*, and such a network is said to be *feed-forward*. The amount of information that flows from one neuron to the next is affected by weight. Each connection in the ANN has an assigned *weight* w_{uvk} that represents the quantity of information transmitted from the neuron u of the $k-1^{th}$ layer through the connection to the neuron v in the k^{th} layer (see Figure 2.2). These weights are parameters that need to be estimated by the training procedure. The neurons in the hidden and output layers apply a non-linear function called the *activation function* on the information inputted. Different hidden layers could be assigned to use different activation functions.

Given that we have a sample of n vectors of observations and p variables, let $\mathbb{X} \in \mathbb{R}^{n \times p}$ be the data matrix defined by:

$$\mathbb{X} = \begin{bmatrix} \mathbf{x}_1^{\mathrm{T}} \\ \mathbf{x}_2^{\mathrm{T}} \\ \vdots \\ \mathbf{x}_n^{\mathrm{T}} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1p} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2p} \\ x_{31} & x_{32} & x_{33} & \dots & x_{3p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & x_{n3} & \dots & x_{np} \end{bmatrix}, \tag{2.1}$$

where each $\mathbf{x}_i \in \mathbb{R}^p$, for $i = 1, ..., n$, contains the values to be inputted into the ANN. The number of neurons in the input layer would be equal to the number of variables in the dataset, which would be p in this case. For a classification task, the selected sample would already have existing labels, i.e., each observation can be in either one of L distinct categories. The class label of each observation can be stored in a single vector

$$\mathbf{l} = \begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ l_n \end{bmatrix}, \tag{2.2}$$

where $l_r \in \{1, 2, \dots, L\}$ represents the observed category of the r^{th} element in the sample. In the case of binary classification, then $L = 2$.

For the following results, let us assume that we have an ANN with p neurons in the input layer, g hidden layers with h neurons each and an output layer, and L neurons in the output layer due to L unique labels. We assume that there are h neurons for each hidden layer here for simplicity, but they can be any size. For each layer beyond the input layer, the output of each neuron can be represented by a linear combination of the weighted inputs of the previous layer, which means that the inputs of the neurons of the k^{th} layer would be the outputs of the neurons of the $(k-1)^{th}$ layer. Added to this linear combination is a value attributed to each neuron called *bias*, and is denoted by b_{vk}. The bias corresponds to the intercept in a regression model and is added to the linear combination defined as the output z_{vk}. We can represent the output z_{vk} of the v^{th} neuron in the k^{th} hidden layer as follows

$$z_{vk} = b_{vk} + w_{1vk}z_{1,k-1} + w_{2vk}z_{2,k-1} + \cdots + w_{hvk}z_{h,k-1} = b_{vk} + \mathbf{w}_{vk}^{\mathrm{T}}\mathbf{z}_{k-1}, \tag{2.3}$$

where $z_{u,k-1}$ represents the output coming from the u^{th} neuron in the $k-1^{th}$ layer and the input for z_{vk}, the weight vector $\mathbf{w}_{vk} = [w_{1vk}, w_{2vk}, \dots, w_{hvk}]^{\mathrm{T}}$ represents the weights connecting the inputs $z_{1,k-1}, z_{2,k-1}, \dots, z_{h,k-1}$ to z_{vk}, and $\mathbf{z}_{k-1}$ represents the output vector

of the h neurons of the $k-1^{th}$ layer that are the inputs for the k^{th} layer, i.e.,

$$\mathbf{z}_{k-1} = \begin{bmatrix} z_{1,k-1} \\ z_{2,k-1} \\ \vdots \\ z_{h,k-1} \end{bmatrix}. \tag{2.4}$$

Equation (2.3) shows the output of a single neuron. The outputs of the h neurons in the k^{th} layer can be represented as a h-output vector

$$\mathbf{z}_k = \mathbf{b}_k + \mathbf{W}_k^{\mathrm{T}} \mathbf{z}_{k-1}, \tag{2.5}$$

which depends on the corresponding (h x h) weight matrix

$$\mathbf{W}_k = \begin{bmatrix} w_{11k} & w_{12k} & w_{13k} & \dots & w_{1hk} \\ w_{21k} & w_{22k} & w_{23k} & \dots & w_{2hk} \\ w_{31k} & w_{32k} & w_{33k} & \dots & w_{3hk} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{h1k} & w_{h2k} & w_{h3k} & \dots & w_{hhk} \end{bmatrix} = \begin{bmatrix} \mathbf{w}_{1k} & \mathbf{w}_{2k} & \dots & \mathbf{w}_{hk} \end{bmatrix}, \tag{2.6}$$

where each column $\mathbf{w}_{vk} \in \mathbb{R}^h$ of the weight matrix represents the v^{th} neuron in the k^{th} layer of the ANN for $v = 1, ..., h$, and the h-bias vector is given as

$$\mathbf{b}_k = \begin{bmatrix} b_{1k} \\ b_{2k} \\ \vdots \\ b_{hk} \end{bmatrix}. \tag{2.7}$$

A graphical representation of a deep ANN with what we have seen so far and an output layer $\mathbf{o}$ can be seen in Figure 2.2.

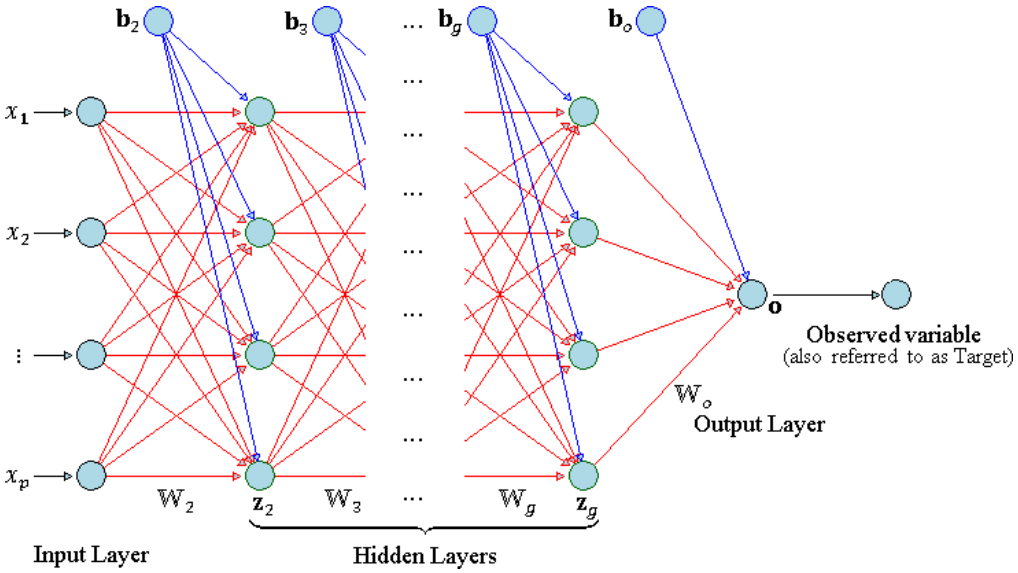


Figure 2.2: A typical layout of a deep ANN.

The bias is generally written off as another weight so that we can generalize the output z_{vk} as a weight update rule, i.e., by 'absorbing' the bias as another weight in equation (2.3), we get

$$\begin{aligned} z_{vk} &= b_{vk} + w_{1vk}z_{1,k-1} + w_{2vk}z_{2,k-1} + \cdots + w_{hvk}z_{h,k-1} \\ &= w_{0vk}z_{0,k-1} + w_{1vk}z_{1,k-1} + w_{2vk}z_{2,k-1} + \cdots + w_{hvk}z_{h,k-1} \\ &= \sum_{i=0}^{h} w_{uvk}z_{i,k-1}, \end{aligned}$$

where $b_{vk} = w_{0vk}z_{0,k-1}$ implies that b_{vk} is equal to either w_{0vk} or $z_{0,k-1}$, and the other is set to value 1. However, since the bias will be continuously updated by the learning process, and the inputs are fixed and do not change, the bias is being treated as a weight so $b_{vk} = w_{0vk}$. With this bias absorption, we would represent the updated weight matrix from equation (2.6) as

$$\mathbf{W}_{k,A} = \begin{bmatrix} b_{1k} & b_{2k} & b_{3k} & \dots & b_{hk} \\ w_{11k} & w_{12k} & w_{13k} & \dots & w_{1hk} \\ w_{21k} & w_{22k} & w_{23k} & \dots & w_{2hk} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{h1k} & w_{h2k} & w_{h3k} & \dots & w_{hhk} \end{bmatrix} = \begin{bmatrix} w_{01k} & w_{02k} & w_{03k} & \dots & w_{0hk} \\ w_{11k} & w_{12k} & w_{13k} & \dots & w_{1hk} \\ w_{21k} & w_{22k} & w_{23k} & \dots & w_{2hk} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{h1k} & w_{h2k} & w_{h3k} & \dots & w_{hhk} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_k^{\mathrm{T}} \\ \mathbf{W}_k \end{bmatrix}, \quad (2.8)$$

where the transpose of the bias vector $\mathbf{b}_k^{\mathrm{T}}$ is added as a row at the top of the previous weight matrix from equation (2.6). When absorbing the bias, we would need to update

the outputs of the h neurons in the k^{th} layer equation of before. Since the dimension of $\mathbf{W}_k$ has increased to ($h+1$ x h) we will need to increase the dimensions of an input case $\mathbf{z}_{k-1}$ from h to $h+1$ by adding a row vector of 1 which corresponds to the bias input, i.e.,

$$\mathbf{z}_{k-1,A} = \begin{bmatrix} 1 \\ z_{1,k-1} \\ \vdots \\ z_{h,k-1} \end{bmatrix}, \tag{2.9}$$

and the outputs equation (2.5) is updated to

$$\mathbf{z}_k = \mathbf{b}_k + \mathbf{W}_k^{\mathrm{T}} \mathbf{z}_{k-1} = \mathbf{W}_{k,A}^{\mathrm{T}} \mathbf{z}_{k-1,A}. \tag{2.10}$$

Similarly, we would need to increase the dimensions of $\mathbf{x}_i^{\mathrm{T}}$ to $p+1$ by adding a row of 1, i.e.,

$$\mathbf{x}_{i,A}^{\mathrm{T}} = \begin{bmatrix} 1 & x_{i1} & \dots & x_{ip} \end{bmatrix}. \tag{2.11}$$

The next section covers the notation for active functions and examples of activation functions used in hidden/output layers.

2.1.2: Activation Functions

Neurons in the hidden and output layers apply an activation function σ on the outputs z_{vk} and display $\sigma(z_{v,k+1})$ as the output for that neuron instead of simply $z_{v,k+1}$. Nonlinear activation functions introduce nonlinearity in the model. Popular activation functions used in practice include the *logistic* function given by

$$\sigma(z_{vk}) = \frac{1}{1 + e^{-\tau z_{vk}}}, \tag{2.12}$$

which gives a value between 0 and 1, where τ is a scale parameter, the *hyperbolic tangent* function

$$\sigma(z_{vk}) = \frac{e^{z_{vk}} - e^{-z_{vk}}}{e^{z_{vk}} + e^{-z_{vk}}}, \tag{2.13}$$

which is symmetric (like the logistic) about the origin with range [-1,1], the *Rectified Linear unit* (ReLu) function

$$\sigma(z_{vk}) = \max\{0, z_{vk}\}, \tag{2.14}$$

which returns a 0 for all nonpositive values, and the *softmax* function

$$o_i = \sigma(z_{vk}) = \frac{e^{z_{vk}}}{\sum_{v=1}^{h} e^{z_{vk}}}, \tag{2.15}$$

which is generally used as the activation function of the output layer to predict the probabilities $\mathbf{o} = [o_1, o_2, \ldots, o_L]^{\mathrm{T}}$ corresponding to the L labels which would add up to 1. Based on $\mathbf{o}$, whose representation we will cover in the next section, we can then get the predicted labels from the output layer.

2.1.3: Output Layer

The predicted values $\mathbf{o}$ in the output layer can be expressed as a function of the previous layers' inputs and all activation functions at every other layer. Assuming that f_g represents the activation function of the g^{th} hidden layer, and f_o represents the activation function of the output layer, then the output vector $\mathbf{o}$ can be represented in the following manner:

$$\begin{aligned}\mathbf{o} &= f_o\left(\mathbf{W}_{o,A}^{\mathrm{T}}\mathbf{z}_{g,A}\right) = f_o \circ f_g\left(\mathbf{W}_{g,A}^{\mathrm{T}}\mathbf{z}_{g-1,A}\right)\\ &= f_o \circ f_g \circ f_{g-1}\left(\mathbf{W}_{g-1,A}^{\mathrm{T}}\mathbf{z}_{g-2,A}\right)\\ &= \ldots\\ &= f_o \circ f_g \circ \cdots f_2 \circ f_1\left(\mathbf{W}_{2,A}^{\mathrm{T}}\mathbf{x}_A\right).\end{aligned} \tag{2.16}$$

Having established the general layout of ANN architectures, the next section will consider optimization algorithms that enable these architectures to learn from data and improve their performance.

2.2: Optimization Algorithms

At the start of training, the ANN makes the forward pass of information utilizing initial values of weights and biases. These initial weights and biases are generally taken to be random values. An ANN is trained using an optimization algorithm, where it would incrementally adjust the weights and biases of an ANN with the aim of improving performance iteration by iteration until an optimal set of weights and biases is found, assuming it exists.

ANN optimization algorithms fall into two broad groups, i.e., *supervised* and *unsupervised* learning. Supervised learning employs labelled data as input to the ANN for a classification problem, where the output of the ANN would be the predicted labels of the data which would be compared with the actual labels. The difference between the output and the actual serves as an error which the ANN would try to minimize. On the other hand, unsupervised learning uses unlabeled data as input with the goal of grouping it together based on hidden patterns found in the data. Unsupervised learning seeks to optimize some criterion that is defined in terms of the activity of the outputs of

the neurons in the ANN. We will focus on supervised learning in the coming sections, notably *gradient descent* and how the *backpropagation algorithm* helps gradient descent.

2.2.1: Gradient Descent and the Backpropagation Algorithm

The backpropagation algorithm computes the gradient of the *error function* with respect to the weights of the ANN using the chain rule. From there, optimization algorithms such as gradient descent are used to minimize an error function at each iteration, where the weights and biases are then updated. The error function, also referred to as the *loss* or *cost* function, behaves similarly to a goodness of fit statistic for the ANN model, i.e., it measures the discrepancy between the observed and predicted labels.

At an iteration q, the gradient of the error function with respect to the weights and biases is found, where we would then update the next set of weights of iteration $q+1$ by moving the weights of iteration q towards the direction of steepest descent. This process is repeated until we find a set of weights that minimizes the error function, which would be considered a solution to the optimization algorithm.

Assume that we have a feed-forward ANN with p input neurons for the p variables and L output neurons in the output layer. We can have any number of hidden neurons and layers present. To train our ANN, we will randomly choose t of the n observations $\mathbf{x}_{1,A}^{\mathrm{T}}, \mathbf{x}_{2,A}^{\mathrm{T}}, \ldots, \mathbf{x}_{n,A}^{\mathrm{T}}$ to serve as a training dataset, and we will start with a random initial set of weights and biases. Since we will need to find the gradient of the error function, it is important that we choose a differentiable and continuous error function. For this, we need to choose a continuous activation function such as the logistic, so that the error function will be continuous as well. In both regression and classification, the *Sum of Squared Errors* (SSE) is a commonly used error function. The SSE function for the t training observations can be defined as follows

$$SSE = \sum_{i=1}^{t} \sum_{r=1}^{L} (l_r^{(i)} - o_r^{(i)})^2, \tag{2.17}$$

where $l_r^{(i)}$ and $o_r^{(i)}$ represent the observed and predicted labels of the i^{th} training case, respectively. Other examples of available error functions include the *Mean Squared Error* (MSE)

$$MSE = \frac{1}{t} \sum_{i=1}^{t} \sum_{r=1}^{L} (l_r^{(i)} - o_r^{(i)})^2, \tag{2.18}$$

and *Cross-Entropy error* (CE) functions

$$CE = -\sum_{i=1}^{t} \sum_{r=1}^{L} l_r^{(i)} \ln o_r^{(i)}. \tag{2.19}$$

For a classification task with two labels, it is standard to use an ANN architecture with a logistic or softmax output layer and the CE error function. A softmax output layer would be more suitable for a classification task with more than two labels (Sadowski, 2018). We will be denoting the error function chosen as E.

We will be updating the weights using the following increment for the q^{th} iteration:

$$w_{uvk}^{q} = w_{uvk}^{q-1} - \gamma \frac{\partial E^{q}}{\partial w_{uvk}^{q-1}}, \tag{2.20}$$

where γ is the learning rate which is a value normally taken between 0 and 1 and $\frac{\partial E^{q}}{\partial w_{uvk}^{q-1}}$ represents the q^{th} batch gradient. Equation (2.20) is known as the *weight update rule*, where $-\gamma \frac{\partial E^{q}}{\partial w_{uvk}^{q-1}}$ represents the parameter update from w_{uvk}^{q-1} to w_{uvk}^{q}. The learning rate is a *hyperparameter* that controls how much the weights are updated at each iteration. Hyperparameters are parameters that are not adjusted by the ANN itself and need to be implemented and adjusted manually. Other examples of hyperparameters include the number of hidden layers, number of neurons in the hidden and output layers, initial weight and bias values, and the choice of optimization algorithm and error function.

The partial derivatives $\frac{\partial E^{q}}{\partial w_{uvk}}$ of the weight update rule are derived using the chain rule and represent the influence of the weight w_{uvk} on the error function E^{q}. The chain rule is necessary since the error function is an indirect function of the weights. The influence of the weights on the error will not be the same layer to layer, where changing any weight will generally affect the influence of other weights on the error.

We will not be going into the derivation of the backpropagation algorithm in this dissertation, where we refer the reader to its full derivation in Goodfellow et al. (2016).

When performing standard gradient descent, the network would use all of the data as input to calculate the loss and perform a single update to the weights. *Minibatch gradient descent* and *Stochastic Gradient Descent* (SGD) represent other variants of gradient descent, which utilize random samples of the data as input instead. We will cover the effects of this approach, as well as comparisons to standard gradient descent, in the next section, since we will be referring to these variants in later chapters.

2.2.2: Minibatch and Stochastic Gradient Descent

Minibatch gradient descent uses a group of randomly selected samples of the training dataset as input in each iteration, where these groups of samples are known as *minibatches* of *batch size B*. The batch size is a hyperparameter indicating the number of samples in each minibatch. Standard gradient descent is akin to having 1 minibatch with a batch size equal to the size of the training dataset. We define an *epoch* to repre-

sent a complete forward and backward pass over the entire training dataset whereas for comparison, an iteration represents a forward and backward pass over a single minibatch which would calculate the loss and update the weights. When training the ANN using minibatches of batch size $B = 1$, we would call this SGD.

For minibatch gradient descent, at the ω^{th} epoch and the q^{th} iteration, we would represent the weight update rule as follows:

$$w_{uvk}^{\omega,q} = w_{uvk}^{\omega,q-1} - \gamma \frac{\partial E^{\omega,q}}{\partial w_{uvk}^{\omega,q-1}}, \tag{2.21}$$

where $\frac{\partial E^{\omega,q}}{\partial w_{uvk}^{\omega,q-1}}$ represents the q^{th} minibatch gradient at the ω^{th} epoch.

Standard gradient descent would compute gradients accurately since it considers all the data at once, although the computation time will be high. SGD and minibatch gradient descent use multiple minibatches to estimate the gradient which would introduce noise into the parameter updates; however, it would have faster computation time with smaller batch sizes (Smith et al., 2020). Depending on the size of the dataset, either minibatch gradient descent or SGD could be more practical to use over standard gradient descent (Li et al., 2014).

When applying any of the three gradient methods to non-convex optimization processes, instead of arriving at a global minimum, the algorithm could get stuck on a saddle point or a local minimum. *Momentum* is a method that helps gradient descent algorithms escape such a fate. Gradient descent with momentum adds a percentage of the past updated gradient to the present gradient, which helps accelerate training and speed up convergence (Shi et al., 2022). This percentage is a hyperparameter that would need to be manually set.

Having examined gradient descent for training ANNs, the next section focuses on the UAT, which formalizes the representational capacity of these models.

2.3: Universal Approximation Theorem

ANNs featuring multiple hidden layers and nonlinear activation functions are adaptable, which helps them to closely approximate any continuous nonlinear function with arbitrary precision. Thus, we can conclude that these kinds of ANNs are *universal approximators* (White, 1992). As such, an ANN can model the linear/nonlinear relationship between a dependent variable and p independent variables (where $p \geq 1$) by a continuous function on compact subsets of $\mathbb{R}^p$.

Numerous variants of the UAT exist in the literature, differing in their assumptions about activation functions, network depth, and function spaces. For example, Arora

et al. (2018) prove that a fully connected feed-forward network using the ReLu activation function can represent any continuous function. Also, Cybenko (1989) prove that even an ANN with a single hidden layer of finite size using a logistic activation function can represent any continuous function well enough.

Because of the usefulness of ANNs as universal approximators and the huge number of parameters in the network, ANNs generally suffer from *overfitting*, which we will cover in the next section.

2.4: Overfitting

Overfitting occurs when the model yields good results with data that was used to train the model, but fails to perform well on new data. In such a scenario, the overfit model perfectly adapts to the data that was used for training but does not generalize well to new data. Conversely, *underfitting* happens when the model lacks sufficient complexity, where it would fail to capture the patterns found in both the data used to train the model and new data, thus producing poor results. Both the overfit and underfit models are not expected to generalize well to new data, so we would want to avoid both situations.

Ideally, we would want to evaluate the trained model using new data; however, new data is not always available to be gathered. Data re-sampling methods such as *cross-validation* methods can be used to assess the generalization of a model and to check for overfitting. A popular cross-validation method used in ANNs is *holdout* cross-validation. In the holdout method, the data would be randomly split into two sets, namely the training and validation sets. The training set represents the majority of our dataset which will be used to train the ANN. The trained ANN would then classify the validation sample, where we would then compare the classification results with the observed labels of the validation sample to evaluate the performance of our model. Generally, around 80% of the data is randomly taken to be the training set and the other 20% would represent the validation set, but this depends on the amount of data available.

In Section 2.4.1, we will cover several regularization methods which are commonly used in ANNs to reduce overfitting. Section 2.4.2 describes *early stopping* to avoid overtraining, which would lead to overfitting. Both methods will be used in the application.

2.4.1: Regularization Methods

Regularization methods represent a set of methods used to lower the complexity of a model during training, i.e., to discourage overly large model parameters. Regularization can be anything in the form of constraints and penalties applied to the hypothesis

search space, which help guide the learning process to simpler models (Moradi et al., 2020). In this section, we will be mentioning L_1, L_2 and *weight decay* regularization since they are commonly utilized in practice and have given excellent results in both deep ANNs (Krizhevsky et al. (2012), Srivastava et al. (2014)) and CNNs (Yu et al., 2009).

Both L_1 regularization and L_2 regularization constrain the models capacity by incorporating a penalty term to the error function E. Weight decay regularization adds a penalty to the weight update rule instead. In general, an overfit model would consider all parameters, including those which have little influence on the final classification. L_1 regularization frequently guides parameters to zero, thus guiding the optimization process to more sparse solutions. L_2 regularization would reduce the magnitude of weights toward zero, which makes the model less prone to overfitting. The main difference between both L_1 and L_2 regularization is that the latter tries to get as much information as possible by encouraging smaller weights for each parameter, whereas the former is more inclined to keep parameters which have the most influence and set the weights of irrelevant parameters to zero (Ying, 2019).

For L_1 and L_2 regularization, we will denote the overall error function E^* as

$$E^* = E + \lambda_{LR} P, \tag{2.22}$$

where P is the regularization term and λ_{LR} is a non-negative tuning parameter. P would consist of the norm of the trainable parameters of the model. For L_2 regularization, we have $P = \sqrt{\sum_{uvk} w_{uvk}^2}$ and for L_1 regularization we have $P = \sum_{uvk} \|w_{uvk}\|$. Larger values of λ_{LR} induce greater regularization and will shrink the weights closer to zero.

In the case of weight decay as defined in Hanson and Pratt (1988), the weights in the weight update rule would decay exponentially as

$$w_{uvk}^q = (1 - \lambda_{WD}) w_{uvk}^{q-1} - \gamma \frac{\partial E^q}{\partial w_{uvk}^{q-1}}, \tag{2.23}$$

where λ_{WD} is the rate of weight decay per update.

2.4.2: Early Stopping

Early stopping halts training once performance on a validation set ceases to improve after a specified number of epochs. As training progresses, model complexity increases, which can lead to overfitting if training continues unchecked.

By monitoring validation performance, early stopping prevents excessive fitting to the training data. Caruana et al. (2000) showed its effectiveness by combining early stopping with backpropagation to train large networks without significant overfitting.

3

Convolutional Neural Networks

Having established the general architecture of fully connected ANNs and their training in Chapter 2, Chapter 3 turns to Convolutional Neural Networks (CNNs), which utilize several different layers that are more suitable for image data. The details of how a CNN architecture is constructed and the optimization algorithms used to train them will be defined with some comparisons made to ANNs, where we will also describe several known CNN architectures from the literature. We will discuss how the set of images from our dance dataset will be represented as tensors for input to a CNN with the goal of pose classification.

Section 3.1 gives a brief introduction to deep ANNs and CNNs. Section 3.2 goes into an overview of tensors and images, and more specifically on how an image can be represented as a tensor. Section 3.3 goes into detail on the general CNN architecture and its layers, including some other common layers in practice. Section 3.4 goes into a literature review of several popular CNN architectures. Section 3.5 presents several adaptive optimization algorithms, including *Adam*. Section 3.6 describes the proof of the UAT for a CNN by He et al. (2022). Section 3.7 describes hyperparameter optimization and the methods used in practice for it. Finally, Section 3.8 explores several performance metrics used to evaluate the performance of a model after training.

3.1: Introduction

Deep ANNs attempt to imitate in a rather simple manner how the human brain operates using multiple layers of neurons connected with each other via a series of nodes. Given that the input data $\mathbb{X}$ and corresponding labels $\mathbf{l}$ are available, the outputs of a deep learning model are expressed by a layered sequence of nonlinear transformations over minibatches of the input data (see equation (2.16)), which would be effective in estimat-

ing the nonlinear relationship between the input data and the output data. Compared to shallow learning, deep learning would have deeper architectures available which are able to extract more abstract information. A popular deep learning model that we will be going through in this chapter is CNNs.

CNNs are deep feed-forward networks that utilize several key architectural ideas, notably the usage of several different layers in a series of stages. Their architecture makes them better suited than ANNs for processing structured arrays of data such as signals or 2D/3D images. An image can be thought of as a collection of many features present that together make up the image itself. Examples of low-level features in images include vertical/horizontal edges and curves that together form highly complex features such as faces and buildings. CNNs aim to extract the said features from the image using 'convolutions' to fulfil a task such as classification. Deeper CNNs would be more sophisticated in their ability to detect highly complex features when compared to shallower CNNs, although a deep CNN would be at a higher risk of overfitting and be more computationally demanding.

Nowadays, CNNs have come to represent the leading architecture for applications concerning Computer Vision, which is a field that looks to solve problems concerning image classification and HPE. They have also found competitive results in natural language processing (Bhandare et al., 2016), i.e., dealing with issues surrounding speech recognition and text classification. Several important factors that have led to the increase in popularity of deep CNNs are the use of deep ANNs, improvements in computer hardware like Graphics Processing Units (GPUs) and the availability of large labelled datasets.

For the duration of this chapter, we will be focusing on the problem of image classification using CNNs. It follows that images can be naturally represented as tensors, as we shall see in the next section.

3.2: Tensors and Images

In CNNs, a tensor is a multi-dimensional array that serves as the basic data structure for representing inputs, outputs, and intermediate data as it flows through the network. Tensors can be defined as multi-dimensional arrays of order O, where O represents the number of dimensions of the tensor. Scalars can be represented as tensors of order 0, whereas vectors and matrices can be represented as order 1 tensors and order 2 tensors, respectively. Given that R and C represent rows and columns of values, respectively, this means that tensors of order 1 represent vectors of size R and tensors of order 2 represent matrices of size $(R \times C)$. Tensors of higher order have additional dimensions.

For example, tensors of order 3 have an additional layer of detail D, thus, they are of size $(R \times C \times D)$.

An image is naturally a grid of *image pixels*; an image pixel represents a single grid point of an image. It is the smallest unit of the image that is defined and is valued according to the intensity of the colour it represents (Padmavathi and Thangadurai, 2016). Pixel intensity varies corresponding to the number of bits per pixel (bpp). To avoid ambiguity, it is useful to distinguish bits per channel and bits per pixel: typical consumer images use 8 bits per channel, so a grayscale pixel often uses 8 bpp, whereas an RGB image pixel commonly uses $3 \times 8 = 24$ bpp total. With 8 bits per channel, channel intensity values range from 0 to 255.

Because images are grids of pixel intensities, they are naturally represented as tensors. A 2D grayscale image of resolution $(I_1 \times I_2)$ is an order 2 tensor with shape $(R, ; C)$ whose values are in $[0, 255]$. A 2D RGB image extends that representation by adding a colour channel axis: it can be represented as an order 3 tensor of shape $(I_1 \times I_2 \times 3)$ where each pixel location stores a 3-vector (R, G, B) of channel intensities (see Figure 3.1).

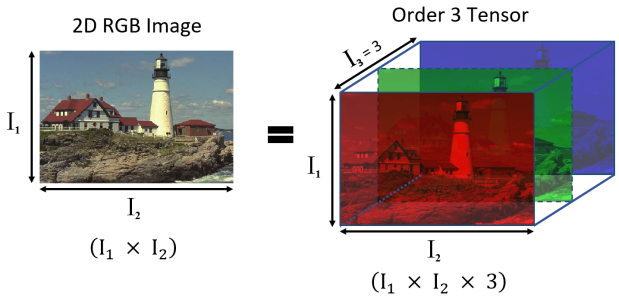


Figure 3.1: A 2D RGB image as an order 3 tensor (Ponomarenko et al. (2015)).

High-order (order 3+) tensors are widely applied in CNNs. The input, hidden layer representations, and parameters in a CNN are all represented by tensors, where the parameters of the CNN model, similarly to ANNs, represent the weights and biases of the model (Wu, 2017). For the duration of this chapter, we will be following the notation that we had established in Chapter 2, where we will be representing tensors following the notation that we used to represent matrices in the form of blackboard bold uppercase letters $\mathbb{X}$. Also, when we define an input image using blackboard bold uppercase letters, we would be referring to it in tensor form. For the k^{th} layer of a CNN, order 2 tensors are adequate for representing 2D grayscale images of size $(R_k \times C_k)$, although, order 3 tensors are needed to represent 2D RGB images of size $(R_k \times C_k \times 3)$. Similarly, order 3 tensors could represent 3D grayscale images of size $(R_k \times C_k \times D_k)$ and order 4 tensors

are needed to represent 3D RGB images of size ($R_k \times C_k \times D_k \times 3$). As for the image pixel, while referring to the input or output image of a layer: an *input pixel* refers to a pixel of the underlying input image, and similarly for an *output pixel* of an output image.

With the foundation of tensors and their role in representing images established, the discussion proceeds to the typical CNN architectures and their layers.

3.3: CNN Architecture

The input image to the CNN would go through several steps of processing via the hidden layers, including *convolution layers* and fully connected layers. We will discuss the different types of hidden layers in Section 3.3.3. However, we will first cover the convolution operation in Section 3.3.1 and stride and padding in Section 3.3.2, which will be necessary for Section 3.3.3.

3.3.1: Convolution Operation

3.3.1.1: General Definition

Convolutions are mathematical operations represented by the operator *, where a convolution operation fundamentally combines two functions defined over a real-valued domain (Goodfellow et al., 2016). In the case of two discrete functions $f(x)$ and $g(x)$, a simple way to understand how the convolution $(f * g)(x)$ is calculated can be understood as follows: reflect $f(x)$ or $g(x)$ to get $f(-x)$ or $g(-x)$ and shift it to the far left of the x-axis so that the two functions do not overlap, where the inverted function is then slid one step at a time to the right, and for each step, the sum of the overlapping products of the two functions is calculated (Healy, 1969). The convolution between two continuous functions f and g would follow similarly, albeit there would be an infinite number of sliding steps and the integral of the overlapping products of the two functions is calculated instead of the sum. It does not matter if $f(x)$ or $g(x)$ is inverted since the convolution operation is *commutative* (refer to Theorem A1 in Appendix A for proof).

To show an example shared by Healy (1969), let us say that we have two continuous functions $f_{ex}(x)$ and $g_{ex}(x)$ where $f_{ex}(x) = 1$ from $0 \leq x \leq 2$ and $g_{ex}(x) = e^{-x}$ from $0 \leq x \leq \infty$ as defined in Figure 3.2(a). The process of convolution is shown in Figure 3.2(b), where we can see that $g_{ex}(x)$ was inverted and being shifted to the right with the highlighted area denoting the product of $f_{ex}(\lambda)$ and $g_{ex}(x - \lambda)$.

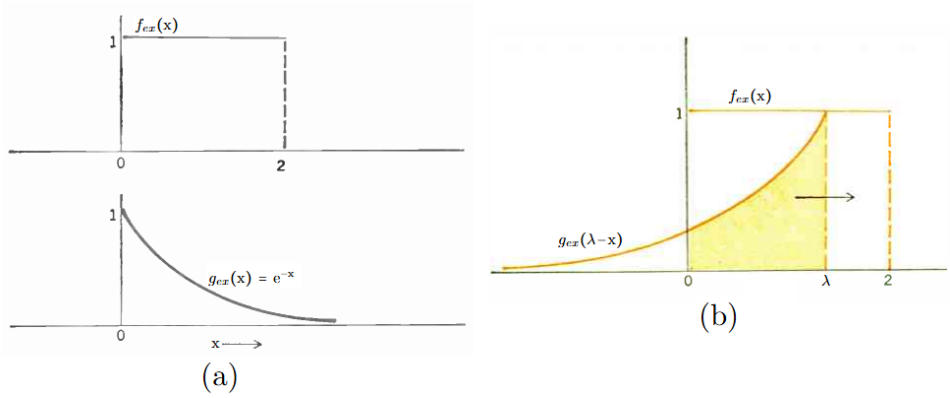


Figure 3.2: (a) Shows functions f_{ex} and g_{ex}. (b) Shows the process of convolution (Healy (1969)).

Assuming that we have two functions $f(x)$ and $g(x)$, then the convolution between these two functions at a point $x \in \mathbb{R}$ is given as

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x-\lambda)g(\lambda)d\lambda, \tag{3.1}$$

in the continuous case and

$$(f * g)(x) = \sum_{\lambda=-\infty}^{\infty} f(x-\lambda)g(\lambda), \tag{3.2}$$

in the discrete case. In the case of 2D, the convolution between f and g at a point $(x,y) \in \mathbb{R}^2$ is given as

$$(f * g)(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x-\lambda_1, y-\lambda_2)g(\lambda_1,\lambda_2)d\lambda_1 d\lambda_2, \tag{3.3}$$

in the continuous case and

$$(f * g)(x,y) = \sum_{\lambda_1=-\infty}^{\infty}\sum_{\lambda_2=-\infty}^{\infty} f(x-\lambda_1, y-\lambda_2)g(\lambda_1,\lambda_2), \tag{3.4}$$

in the discrete case.

Going back to the example from Figure 3.2 to show the calculation of the convolution operation for these two functions, the integration would be defined between 0 to x, although the upper integral would be different from $0 \leq x \leq 2$ and $x > 2$. As such, the convolution operation can be calculated as follows

$$(f_{ex} * g_{ex})(x) = \int_0^{\infty} f_{ex}(x-\lambda)g_{ex}(\lambda)d\lambda = \begin{cases} \int_0^x e^{-(x-\lambda)}d\lambda, & \text{for } 0 \leq x \leq 2, \\ \int_0^2 e^{-(x-\lambda)}d\lambda, & \text{for } x > 2, \end{cases} \tag{3.5}$$

which simplifies to

$$(f_{ex} * g_{ex})(x) = \begin{cases} 1 - e^{-x}, & \text{for } 0 \leq x \leq 2, \\ e^{-x}(e^2 - 1), & \text{for } x > 2. \end{cases} \tag{3.6}$$

With the convolution operation formally defined, the following section examines how this operation is employed as a foundational mechanism in CNNs.

3.3.1.2: Convolution Operation in CNN

The convolution layer represents a fundamental component of a CNN, where the convolution operation is performed on each input image to the convolutional layer, and the features extracted are given out as the output pixels of the output image. Depending on if the input image to the convolutional layer is 2D or 3D, then a respective tensor $\mathbb{F}$ would be used to represent it. Also, let us define a *filter* $\mathbb{G}$ which represents a tensor of weights.

We have defined the convolution operation in terms of functions, but how does it work in the case of tensors? Given that we have a 2D input image with size $(R \times C)$ represented as a tensor $\mathbb{F}$ of order 2 and R and C are integers, then we can represent $\mathbb{F}$ as a function using:

$$\mathbb{F} : [1, R] \times [1, C] \to [0, 255]. \tag{3.7}$$

Given that x is an integer in the range $[1, R]$, y is an integer in the range $[1, C]$ and z is an integer in the range $[0, 255]$, then $\mathbb{F}(x, y) = z$. Here, the coordinates (x, y) represent an input pixel in the input image and z represents the colour intensity of the input pixel at coordinates (x, y). A filter $\mathbb{G}$ with size $(P \times Q)$ can be defined similarly in this case. In the case of RGB images, given that we have a 2D input RGB image with size $(R \times C \times 3)$ represented as a tensor $\mathbb{F}$ of order 3, then we can represent $\mathbb{F}$ as a function as follows:

$$\mathbb{F} : [1, R] \times [1, C] \to [0, 255]^3. \tag{3.8}$$

Given that x is an integer in the range $[1, R]$, y is an integer in the range $[1, C]$ and $(r, g, b) \in [0, 255]^3$, then $\mathbb{F}(x, y) = (r, g, b)$ represents the colour intensity induced by the RGB colors.

In the case of tensors of order 2, given that an input image $\mathbb{F}$ is of size $(R \times C)$ and a filter $\mathbb{G}$ is of size $(P \times Q)$, then the convolution of $\mathbb{F}$ and a filter $\mathbb{G}$ at a pixel (x, y) of the output $\mathbb{F} * \mathbb{G}$ is calculated as follows:

$$(\mathbb{F} * \mathbb{G})(x, y) = \sum_{p=1}^{P} \sum_{q=1}^{Q} \mathbb{F}(x - p, y - q)\mathbb{G}(p, q). \tag{3.9}$$

The resulting image $\mathbb{F} * \mathbb{G}$ is also referred to as a *feature map*. Note that the operation performed in equation (3.9) is linear since it represents each pixel of $\mathbb{F} * \mathbb{G}$ as a linear combination of input pixels from the input image (Jacobs, 2005) (see Figure 3.4). For order 3+ tensors, the convolution operation would be defined similarly.

Figure 3.3 represents the output image at the top and the input image at the bottom, along with the filter shown as a light blue outline on the input image. It gives a visual illustration of the convolution operation at four different output pixels, where each output pixel in the output image has a distinct local region of input pixels. A convolution operation performed at each pixel (x, y) of $\mathbb{F} * \mathbb{G}$ uses a distinct region of input pixels from $\mathbb{F}$ whose size represents the width and height of the filter $\mathbb{G}$ (see Figure 3.3). This region of input pixels is known as the *local receptive field* of the pixel (x, y) of $\mathbb{F} * \mathbb{G}$. Figure 3.4 adds on Figure 3.3 since it gives a visual illustration of the convolution operation in 2D between $\mathbb{F}$ and $\mathbb{G}$ for some local receptive field.

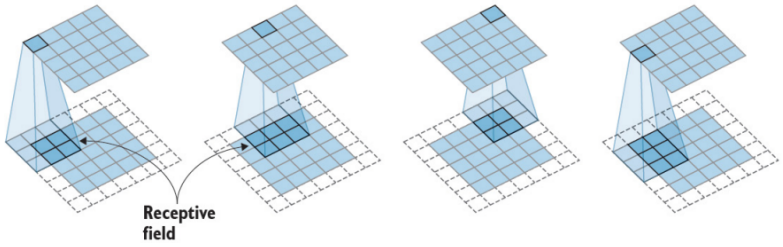


Figure 3.3: A visual illustration of the convolution operation in 2D with the input/output images both outlined in blue and displayed at the bottom/top, respectively (Elgendy (2020)).

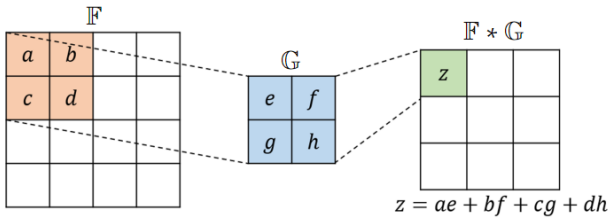


Figure 3.4: Convolution operation between an input image $\mathbb{F}$ and filter $\mathbb{G}$ (Yuan et al. (2020)).

The dimensions of the filter are generally square and of smaller size relative to the input image. Here are some examples of input image and filter sizes in the case of tensors of order 2: given input images of size (42 × 42), Kim et al. (2015) had used filters of size (4 × 4) and (5 × 5), whereas Zeiler and Fergus (2014) had used filters of size (7 × 7) and (11 × 11) for input images of size (224 × 224).

Depending on the size of the filter, it is possible that the filter would go outside of the dimensions of the input image when centering itself on the border pixels of the input image (Hashemi, 2019) (see Figure 3.3 where there are receptive fields containing outlined border pixels). In these situations, the local receptive fields centered on these border pixels would not be fully defined. In practice, the general approach is to apply

the convolution operation only on the input pixels where the filter does not go outside the dimensions of the image, although this would result in the feature map being smaller than the input image if $\mathbb{G}$ is not of size (1×1) (see Figure 3.4). However, there is an alternative approach where we add extra rows of pixels to the borders of the input image, thereby increasing its dimension so that the local receptive fields centering the border pixels of the input image (those which were not fully defined before) would be fully defined (see Figure 3.3) (Islam et al., 2021). This alternative approach is known as *padding* and we will be going into more detail on this in Section 3.3.2.

When applying the convolution operation between the input image and the filter, the order of movement of the filter starts from the top-leftmost position of the input image, where from there it would move according to some value to a neighbouring local receptive field on the right for that same row (see Figure 3.3). We will be referring to that value as the *stride length*. We will be going into more detail on the stride length in Section 3.3.2.

Figure 3.5 represents a comparison between the connections of a convolutional layer (Top) and fully connected layer (Bottom), where common among both are the neurons in the lower layer $x_1, x_2, \ldots, x_5$ and the neurons in the upper layer $s_1, s_2, \ldots, s_5$ with a grey highlighted upper neuron s_3. Figure 3.5 (Top) shows the inputs to a convolutional layer, where the weights are shared across local receptive fields, whereas Figure 3.5 (Bottom) shows the inputs to a fully connected layer, where all input/output neurons are connected and all weights are unique.

To be able to compare the connections between the input and output neurons shown in a fully connected layer to those shown in a convolutional layer as in Figure 3.5, we will be flattening the tensors representing the input image and the filter into vectors of neurons. We are doing this so that it will be easier to show visually the difference in the number of connections between the input and output images in a fully connected layer and a convolution layer using Figure 3.5. Traditional ANNs use fully connected layers, where each output neuron can be expressed as a linear combination of all input neurons and their respective weights. On the other hand, for a convolution layer, not all input neurons would be connected to an output neuron since the weights in a filter are only interacting with a selected number of input neurons corresponding to the local receptive field of the output neuron (see Figure 3.3). As such, each output neuron can be expressed as a linear combination of the input neurons from its respective local receptive field and the weights in the filter. This shows that convolutional layers utilize *sparse connectivity*, i.e., they connect only a selected number of input neurons to each output neuron (see Figure 3.5 (Top)), which is far fewer connections compared to the dense connectivity of the fully connected layer (see Figure 3.5 (Bottom)).

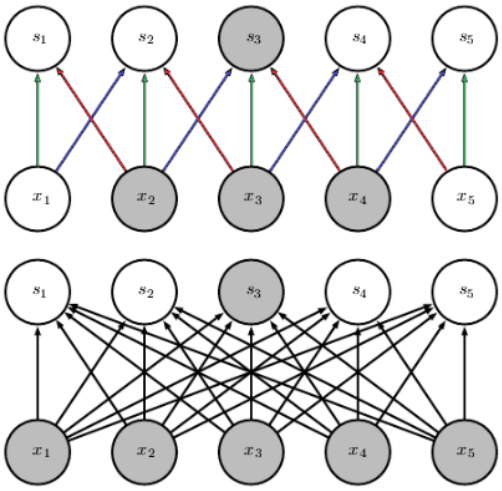


Figure 3.5: (Top) The inputs to a convolutional layer, where the weights shared across local receptive fields are symbolized by the red, green, and blue connections. (Bottom) The inputs to a fully connected layer. Taken from Goodfellow et al. (2016).

The property of *weight sharing* from using a filter is useful in that it restricts the number of parameters in a convolutional layer to only be the number of weights in the filters. Given the use of sparse connectivity and weight sharing in a convolutional layer, when compared to the number of parameters in a fully connected layer, using a convolutional layer over a fully connected layer is a drastic decrease in the number of parameters in the model which effectively reduces the chance of overfitting (Hinton et al., 2012). Furthermore, weight sharing enables the network to detect specific features, such as hands, in any region of an image once learned at a single location (Kondor and Trivedi, 2018).

Given an input image and a function that is *translation equivariant*, the result that we would acquire when the input image is shifted and then applied with the function would be the same result as if we had taken the output of the function on the input image and shifted it then (Mouton et al., 2021). Given an input image $\mathbb{F}$ and the set of all translations referred to as the translation group $\mathcal{T}$, a function f is said to be equivariant to the translations of $\mathcal{T}$ if for $t \in \mathcal{T}$, we have

$$f(t(\mathbb{F})) = t(f(\mathbb{F})). \tag{3.10}$$

It can be proven that the convolution operation is translation equivariant (refer to Theorem A2 in Appendix A for proof), i.e., translating the input image produces the same corresponding translation in the feature map (see Figure 3.6). This is a consequence of the convolution layer having weight sharing. This property is important because otherwise, a CNN would not extract the same features from the original input image

compared to the features from the shifted input image, potentially leading to different predictions made. It is worth noting that the property of translation equivariance does not necessarily hold depending on the value of the stride length (Li et al. (2018b), Azulay and Weiss (2019)). We will be going into some more detail on this in Section 3.3.2.

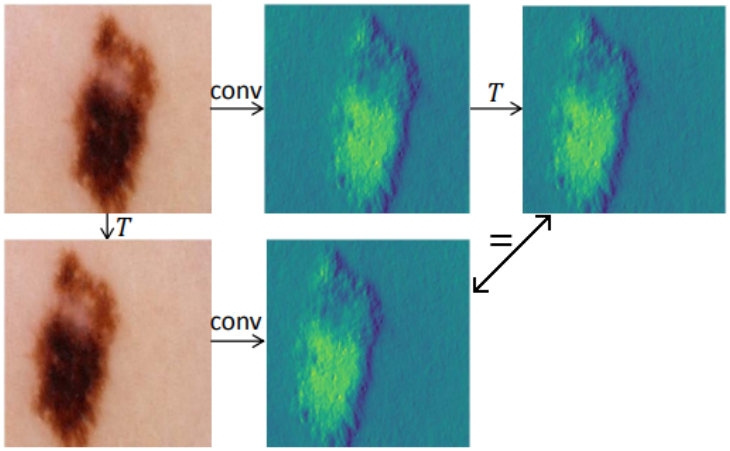


Figure 3.6: Two cases here: one where the input image undergoes convolution and then is shifted and the other showing the translation of the input image which undergoes convolution. Both result in a similar image. Taken from Li et al. (2018b).

Since the convolution operation is commutative, we can instead write equation (3.9) as

$$(\mathbb{G} * \mathbb{F}))(x,y) = \sum_{p=1}^{P}\sum_{q=1}^{Q} \mathbb{F}(p,q)\mathbb{G}(x-p,y-q), \tag{3.11}$$

meaning that the order to which we convolve $\mathbb{F}$ and $\mathbb{G}$ does not change the result.

We briefly mention the cross-correlation operation since some software and researchers (e.g. He et al. (2022)) use the operations of convolution and cross-correlation interchangeably. Compared to equation (3.9), the cross-correlation operation is given as follows

$$(\mathbb{F} * \mathbb{G})(x,y) = \sum_{p=1}^{P}\sum_{q=1}^{Q} \mathbb{F}(x+p,y+q)\mathbb{G}(p,q). \tag{3.12}$$

The cross-correlation operation operates similarly to the convolution operation but it does not reflect $\mathbb{F}$ due to the change from negative to positive indices in $(x+p,y+q)$. However, unlike the convolution operation, the cross-correlation operation does not have the commutative property (refer to Theorem A3 in Appendix A for proof).

The convolution operation can be constructed in the form of a matrix-vector multiplication operation, and we refer to Appendix A for more details on this. Having

established the role of convolution within CNNs, we will now move to examine the mechanisms that govern its application in the layers, i.e., stride and padding.

3.3.2: Stride and Padding

How far the filter moves every time during the convolution operation depends on the stride length s. A stride length above 1 means that the filter will end up skipping certain local receptive fields, i.e., a stride of length s will skip $s - 1$ local receptive fields in every move. Filters with stride values greater than 1 are commonly used in practice, where popular values for the stride length generally go from 1 to 3. Figure 3.7 depicts two separate convolution operations with stride lengths of 1 and 2.

After the filter reaches the last local receptive field on a row, the filter would then move according to some value (which is generally the same as s) to the left-most local receptive field below the current row. This process is repeated until all local receptive fields of the input image that can be reached with s have been seen. The order of movement of the filter is important since the filter can be configured to skip local receptive fields corresponding to the stride length. Figure 3.3 presents the filter started at the top-leftmost local receptive field and moved with a stride length of 2 to the next local receptive field on that same row, where it then moved again and reached the end of the row. As such, this order of movement made the filter skip two local receptive fields on that row. After it reached the end of the first row, we can see that the filter moved only one row below the current row and to the left-most local receptive field on that row.

The advantage to increasing the stride length is that it drastically reduces the size of the feature map, which has the benefit of reducing the number of parameters in the model and hence the training time (Mouton et al., 2021).

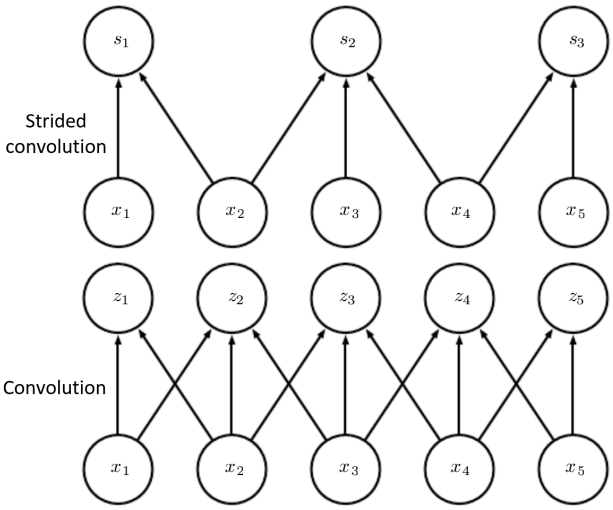


Figure 3.7: (Top) Convolution operation with $s = 2$. (Bottom) Convolution operation with $s = 1$. Taken from Goodfellow et al. (2016).

The disadvantage of using filters with stride greater than one is that the translation equivariance property of the convolution layer is broken since there is information in the input image that is being ignored. Due to this loss of information, the shift in the input image will not necessarily result in an equivalent response as in Figure 3.6 (Azulay and Weiss (2019), Mouton et al. (2021)), i.e,

$$f(t(\mathbb{F})) \neq t(f(\mathbb{F})). \tag{3.13}$$

Figure 3.8 represents the convolution of an input image with three separate filters of sizes (1 × 1), (2 × 2) and (3 × 3) respectively (from left to right) and also illustrates the number of times each input pixel appears in a local receptive field. Note that as the filter size increases, the input pixels near the image border are represented in a fewer number of local receptive fields compared to the input pixels located near the centre. The concept of padding in CNNs is used to add additional rows of image pixels at the borders of an image so that the input pixels near the image border would be represented more. It is also useful in that when we use sufficient padding on the input image when applying convolution, we will be able to maintain similar dimensions between the feature map and the input image (Islam et al., 2021).

When adding padding to an image, it is worth noting that the actual intensity of the pixels (as defined in Section 3.1) that need to be added would not be known a priori (Alguacil et al., 2021). There are several padding methods that are used in practice to try to estimate the intensity of the pixels added from padding. One such method is *replication padding*, which adds padding where the intensity of the pixels that is added

would be the same as that of the border pixels, i.e., the intensity of the border pixels would be replicated multiple times into the padding area (Alguacil et al., 2021). Another well-known padding method is *periodic* padding, wherein the horizontal axis, we pad the left of the image by the rightmost part of the image, and vice versa; similarly in the vertical axis, we pad the upper left of the image by the upper right, and vice versa (Wang et al. (2018)). The most well-known of the padding methods is *zero-padding*, where the intensity of the pixels added from padding would be zero.

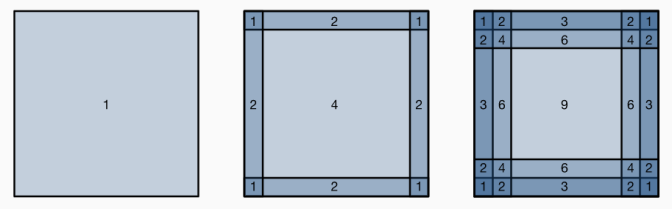


Figure 3.8: The convolution of an input image with three filters of different sizes while displaying the number of times each input pixel is in a local receptive field (Zhang et al. (2024)).

Figure 3.9 represents two scenarios involving a CNN with multiple convolutional layers in which padding is used in one and not in the other. In the case when padding is not added (see Figure 3.9 (Top)), the size of the feature maps are gradually reducing with every convolution layer. This means that for a stride length s, given an input image of size ($R \times C$) and a filter of size ($P \times Q$), then the output feature map will have size ($[\frac{R-P}{s} + 1] \times [\frac{C-Q}{s} + 1]$). The other case includes adding enough padding in every convolution layer to keep the size of the input image the same as the output feature map (see Figure 3.9 (Bottom)). As such, the output feature map would be of size ($[\frac{R-P+2G}{s} + 1] \times [\frac{C-Q+2G}{s} + 1]$) instead, where G is the amount of padding added depending on the size of the filter. The next section will examine the different layers that define CNNs.

3.3.3: Layers of a CNN

The hidden layers of a deep CNN carry out feature extraction on the input image using convolution layers, *pooling layers*, *nonlinearity layers*, and fully connected layers, although pooling layers would not always be present. Convolutional, nonlinearity, pooling and fully connected layers represent the basic building blocks of a CNN architecture, and we will cover them in Sections 3.3.3.1 to 3.3.3.6. Other types of layers are generally used in CNN architectures, two of which are known as *dropout* and *batch normalization* layers, which we will discuss in Section 3.3.3.7.

3.3.3.1: Convolution Layer

In this section, we will be going into a mathematical formulation of what happens in the convolution layer given that we have 2D or 3D images as input.

Recall that we can express a 2D or 3D image as an order 3 tensor of size $(R \times C \times m)$ or order 4 tensor of size $(R \times C \times D \times m)$, respectively, where for grayscale images $m = 1$ and for RGB images $m = 3$. Thus, we can say that the input image can be represented as a group of m feature maps, where m represents the number of feature maps. We will refer to this group of feature maps as the *input maps volume*. In a convolution layer, given an input image, there will generally be more than one filter defined, where the convolution operation would be applied between each filter and the input image. The output of the convolution layer will consist of as many feature maps as there are filters. We will refer to this group of feature maps as the *feature maps volume*. In general, we will be referring to the group of feature maps that are inputted into a layer as the input maps volume and the group of feature maps that are outputted from a layer as the feature maps volume.

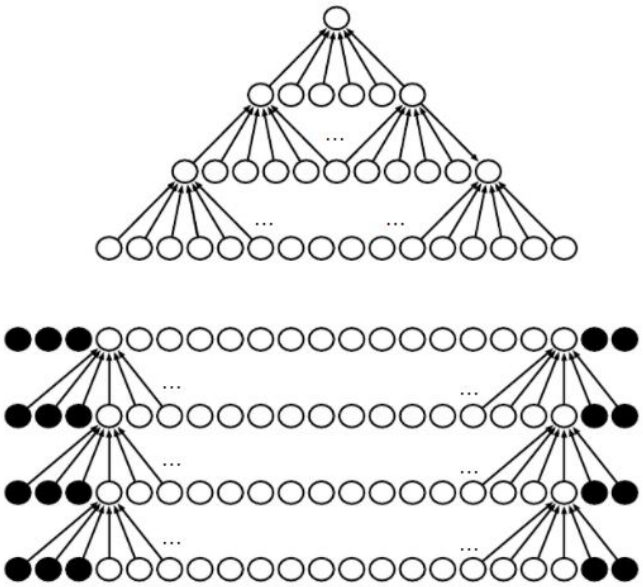


Figure 3.9: (Top) No padding added in the convolutional layers. (Bottom) CNN with multiple convolution layers with padding added (shown as black neurons). (Goodfellow et al., 2016).

Throughout this section, we will be assuming that a stride of length 1 is used, i.e., $s = 1$. Also, if padding has been added in a layer, then the size of the image that we would be referring to would be the padded image instead of the original image.

Given that the k^{th} layer represents the convolution layer, let us assume that the input maps volume at the k^{th} layer comprises of m_{k-1} feature maps of size $(R_{k-1} \times C_{k-1})$ from the $k - 1^{th}$ layer. We will be assuming that we have m_k filters, and that the size of each

filter for this convolution layer is $(P_k \times Q_k \times m_{k-1})$, where $P_k < R_{k-1}$ and $Q_k < C_{k-1}$. The value of the output pixel at position (x, y) in the j^{th} output feature map of the k^{th} layer, denoted as $v_{jk}^{(x,y)}$, is given as

$$v_{jk}^{(x,y)} = b_{jk} + \sum_{a=1}^{m_{k-1}} \sum_{p=0}^{P_k-1} \sum_{q=0}^{Q_k-1} v_{a(k-1)}^{(x+p,y+q)} w_{ajk}^{(p,q)}, \tag{3.14}$$

where $1 \leq j \leq m_k$, $0 \leq x < R_k$, $0 \leq y < C_k$, b_{jk} represents the bias value for the j^{th} output feature map of the k^{th} layer, $w_{ajk}^{(p,q)}$ represents the weight assigned at position (p, q) in the a^{th} feature map of the $k-1^{th}$ layer for the j^{th} output feature map of the k^{th} layer, and $v_{a(k-1)}^{(x+p,y+q)}$ represents the value of the neuron at position $(x + p, y + q)$ in the a^{th} feature map of the $k-1^{th}$ layer.

This means that the j^{th} output feature map of the k^{th} layer, denoted as $\mathbf{V}_{jk}$, will be of size $(R_k \times C_k)$ and is given by

$$\mathbf{V}_{jk} = \mathbf{B}_{jk} + \sum_{a=1}^{m_{k-1}} \mathbf{V}_{a(k-1)} * \mathbf{W}_{ajk}, \tag{3.15}$$

where $\mathbf{B}_{jk}$ is the bias matrix with size $(R_k \times C_k)$ filled with the b_{jk} entries, $\mathbf{W}_{ajk}$ represents the weight matrix of size $(P_k \times Q_k)$ connecting the a^{th} feature map of the $k-1^{th}$ layer with the j^{th} output feature map of the k^{th} layer and $\mathbf{V}_{a(k-1)}$ represents the a^{th} feature map of the $k-1^{th}$ layer with size $(R_{k-1} \times C_{k-1})$. The convolution operation can be represented instead as a matrix-vector multiplication operation, where in Appendix A, we show how equation (3.15) can be represented as this.

The output feature maps of the k^{th} layer, denoted as $\mathbb{V}_k$, can be represented as follows:

$$\mathbb{V}_k = \mathbb{B}_k + \mathbb{V}_{k-1} * \mathbb{W}_k, \tag{3.16}$$

where $\mathbb{W}_k$ is an order 4 tensor of size $(P_k \times Q_k \times m_{k-1} \times m_k)$, $\mathbb{V}_{k-1}$ is an order 3 tensor of size $(R_{k-1} \times C_{k-1} \times m_{k-1})$, and $\mathbb{B}_k$ and $\mathbb{V}_k$ are order 3 tensors of size $(R_k \times C_k \times m_k)$, where

$$R_k = R_{k-1} - P_k + 1, \tag{3.17}$$

and

$$C_k = C_{k-1} - Q_k + 1. \tag{3.18}$$

The convolution operation follows similarly in terms of 3D input images. We assume that there is an input maps volume of m_{k-1} feature maps from the $k-1^{th}$ layer of size $(R_{k-1} \times C_{k-1} \times D_{k-1})$ and that m_k filters of size $(P_k \times Q_k \times S_k)$ are used in the k^{th} layer,

where $P_k < R_{k-1}$, $Q_k < C_{k-1}$ and $S_k < D_{k-1}$. Then the value of the output pixel at position (x, y, z) in the j^{th} output feature map of the k^{th} layer, denoted as $v_{jk}^{(x,y,z)}$, is given by

$$v_{jk}^{(x,y,z)} = b_{jk} + \sum_{a=1}^{m_{k-1}} \sum_{p=0}^{P_k-1} \sum_{q=0}^{Q_k-1} \sum_{s=0}^{S_k-1} v_{a(k-1)}^{(x+p,y+q,z+s)} w_{ajk}^{(p,q,s)}. \tag{3.19}$$

Similarly for 3D input images, the output feature maps volume of the k^{th} layer, denoted as $\mathbb{V}_k$, can be represented as follows:

$$\mathbb{V}_k = \mathbb{B}_k + \mathbb{V}_{k-1} * \mathbb{W}_k, \tag{3.20}$$

where $\mathbb{W}_k$ is an order 5 tensor of size $(P_k \times Q_k \times S_k \times m_{k-1} \times m_k)$, $\mathbb{V}_{k-1}$ is an order 4 tensor of size $(R_{k-1} \times C_{k-1} \times D_{k-1} \times m_{k-1})$, and $\mathbb{B}_k$ and $\mathbb{V}_k$ are order 4 tensors of size $(R_k \times C_k \times D_k \times m_k)$, where

$$R_k = R_{k-1} - P_k + 1, \tag{3.21}$$

$$C_k = C_{k-1} - Q_k + 1, \tag{3.22}$$

and

$$D_k = D_{k-1} - S_k + 1. \tag{3.23}$$

The initial values for weights and biases in the convolutional layer are generally taken to be randomly drawn from a Gaussian distribution with a mean of 0 and a standard deviation of 0.01 (Krizhevsky et al., 2012). The number of filters m_k in the convolutional, the width and height of the filters, and the padding/stride at each convolutional layer are all hyperparameters that would need to be manually accounted for when modelling the CNN.

3.3.3.2: Nonlinearity Layer

Nonlinearity layers consist of a linear/nonlinear activation function which transform the feature maps generated by convolutional layers. The output of the CNN would end up being a linear function without nonlinear activation functions. As such, nonlinear activation functions are necessary to allow the CNN to be able to model nonlinear functions as is often the case in practice with images and videos (Sharma et al., 2020).

The nonlinear activation functions which are typically used in CNNs are the logistic, hyperbolic tangent and ReLU functions that we had defined in Section 2.1.2. The ReLU function and its' alternatives (leaky ReLU, etc.) are generally preferred over the other two functions due to possible complications that can happen when applied in deep networks. One such complication is the *vanishing gradient problem*. To summarize the

vanishing gradient problem, when using the logistic and hyperbolic tangent functions, the gradients calculated in the weight update rule during backpropagation will become very small which would make the weight updates close to negligible.

Following from the notation used in the convolution layer section; assuming that the input maps volume of the k^{th} layer comprises of m_{k-1} feature maps from the previous layer and given that the k^{th} layer represents the nonlinearity layer, then the j^{th} output feature map for the k^{th} layer will be given as

$$\mathbf{V}_{jk} = \sigma(\mathbf{V}_{j(k-1)}), \tag{3.24}$$

where σ represents the activation function. Note that the input maps volume and output feature maps volume of the k^{th} nonlinearity layer will have unchanged feature map sizes and the same number of feature maps.

Generally in literature, the nonlinearity layer is 'merged' with the convolutional layer, where given that the k^{th} layer is the convolutional layer, the output of the convolution operation would pass through the activation function without the $k+1^{th}$ layer necessarily being defined as a nonlinearity layer. As such, the output of the k^{th} layer would be equal to the activation function of equation (3.15), i.e.,

$$\mathbf{V}_{jk} = \sigma(\mathbf{B}_{jk} + \sum_{a=1}^{m_{k-1}} \mathbf{V}_{a(k-1)} * \mathbf{W}_{ajk}). \tag{3.25}$$

There are no new parameters added in the model from a nonlinearity layer, other than the choice of activation function which is a hyperparameter. Generally, the same activation function is applied over all nonlinearity layers in the CNN, but they can be adjusted to use a different activation function at any layer.

3.3.3.3: Pooling Layer

Convolution layers are effective in spotting the exact position of an object in the underlying image. However, this makes it vulnerable to any small changes to the position of the features in the image. A pooling layer, also known as a *downsampling* layer, performs the *pooling operation* on each feature map of the input maps volume. The pooling operation is used to aggregate information in each local receptive field for a feature map. This effectively reduces the size of a feature map and thus reduces the number of parameters needed for subsequent layers in the model as well. Since the output feature maps volume of a pooling layer will be summarized features of prior convolution layers, the model will be more robust to variations in the positioning of the features located in the input image to the CNN. Ideally, the pooling operation extracts the information that is needed for classification while weeding out unnecessary details in the feature map

and effectively reducing its dimensions (Zafar et al., 2022). There are multiple pooling operations available, the most popular of which being *max pooling* and *average pooling*.

Similarly, as in convolutional layers with the convolution operation, pooling layers make use of filters to compute the pooling operation for each feature map. However, the filters used for pooling do not have trainable weights since they are used only to compute the pooling operator. These filters are usually of small size with a small stride length. In the case of tensors of order 2, filters of size (2×2) and (3×3) are commonly used in pooling along with a stride length of 2. This is so that the size of the feature maps in the feature maps volume is not lowered by too much.

Max Pooling: The max pooling operator simply finds the maximum value of each local receptive field and displays that as the value of the output pixel. We will be referring to a pooling layer with a max pooling operator as a *max pooling layer* in short. Following the notation used in the convolution layer section; assume that the input maps volume at the k^{th} layer comprises of m_{k-1} feature maps of size $(R_{k-1} \times C_{k-1} \times D_{k-1})$ from the $k-1^{th}$ layer, where the k^{th} layer is the max pooling layer. The pooling operation will be done by a filter of size $(P_k \times Q_k \times S_k \times m_{k-1})$, where $P_k < R_{k-1}$, $Q_k < C_{k-1}$ and $S_k < D_{k-1}$. The value of the output pixel at position (x, y, z) in the j^{th} output feature map of the k^{th} layer, denoted as $v_{jk}^{(x,y,z)}$, is given by

$$v_{jk}^{(x,y,z)} = \max_{x,y,z} v_{j(k-1)}^{(xP_k+p, yQ_k+q, zS_k+s)}, \tag{3.26}$$

where $0 \leq j < F$, $0 \leq x < R_k$, $0 \leq y < C_k$, $0 \leq z < D_k$, $0 \leq p < P_k$, $0 \leq q < Q_k$, $0 \leq s < S_k$, and $\max_{x,y,z} v_{j(k-1)}^{(xP_k+p, yQ_k+q, zS_k+s)}$ represents the value of the neuron at position $(xP_k + p, yQ_k + q, zS_k + s)$ in the j^{th} output feature map of the $k-1^{th}$ layer.

Average Pooling: We will be following the previous notation of the max pooling operator but this time, the k^{th} layer is a pooling layer with an average pooling operator which we will refer to it as the *average pooling layer* in short. The average pooling operator computes the average value of each local receptive field and displays that as the value of the output pixel. The value of the output pixel at position (x, y, z) in the j^{th} output feature map of the k^{th} layer, denoted as $v_{jk}^{(x,y,z)}$, is given by

$$v_{jk}^{(x,y,z)} = \frac{1}{P_k Q_k S_k} \sum_{x,y,z} v_{j(k-1)}^{(xP_k+p, yQ_k+q, zS_k+s)}, \tag{3.27}$$

where $0 \leq x < R_k$, $0 \leq y < C_k$, $0 \leq z < D_k$, $0 \leq p < P_k$, $0 \leq q < Q_k$, $0 \leq s < S_k$, and $v_{j(k-1)}^{(xP_k+p, yQ_k+q, zS_k+s)}$ represents the value of the neuron at position $(xP_k + p, yQ_k + q, zS_k + s)$

in the j^{th} output feature map of the $k-1^{th}$ layer.

Note that the output feature maps of the k^{th} pooling layer, regardless of the chosen pooling operator, will be of size $(R_k \times C_k \times D_k)$, where

$$R_k = \frac{R_{k-1}}{P_k}, \quad C_k = \frac{C_{k-1}}{Q_k}, \quad \text{and} \quad D_k = \frac{D_{k-1}}{S_k}. \tag{3.28}$$

For tensors of other order sizes, the mathematical formulations would follow similarly. Note that the input maps volume and output feature maps volume to the k^{th} layer will have a similar amount of feature maps, albeit different sizes.

Figure 3.10 represents an example showing both the max and average pooling operators with a size (4×4) input image and a size (2×2) filter with no padding and a stride of 2. The colored regions in each image, which represent where the filter is moving, are affected by max/average pooling, resulting in an output image of size (2×2).

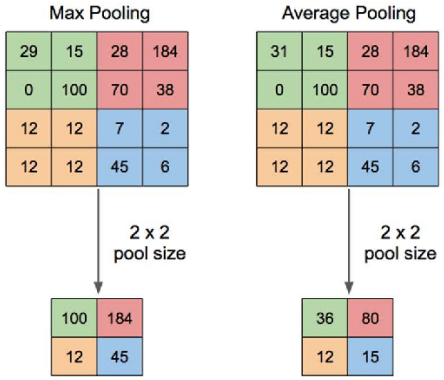


Figure 3.10: An example of both max and average pooling. Taken from Yani et al. (2019).

Note that no trainable parameters are added in the model from a pooling layer. Although, we still need to consider the hyperparameters corresponding to the pooling operator used, filter size and stride length. Padding can be used in a pooling layer (another hyperparameter to account for if used) but is generally not used.

3.3.3.4: Fully Connected Layer

Fully connected layers, as defined in Section 2.1.1, are generally included at the end of a CNN model to utilize every feature extracted from the prior layers to predict the label of the input image. To do this, the input maps volume to the fully connected layer would be first flattened giving a very high-dimensional vector. Given that $R_k = C_k$ and that $\mathbb{V}_k$ is the input maps volume to the fully connected layer with size $(R_k \times R_k \times m_k)$, then

the vectorization of $\mathbb{V}_k$ can be represented as $\text{vec}(\mathbb{V}_k)$, which will be a $R_k^2 m_k$-vector. The value of the neuron at position (r, c, j) of $\mathbb{V}_k$ and its position in $\text{vec}(\mathbb{V}_k)$ can be represented as follows:

$$[\mathbb{V}_k]_{(r,c,j)} = [\text{vec}(\mathbb{V}_k)]_{(j-1)R_k^2+(r-1)R_k+c}, \tag{3.29}$$

where $r = 1, ..., R_k$, $c = 1, ..., R_k$, and $j = 1, ..., m_k$.

In the case when the $k-1^{th}$ and k^{th} layers are fully connected layers, then the procedure follows similarly as in equation 2.3, i.e., for n_{k-1} and n_k neurons in the $k-1^{th}$ and k^{th} layers, respectively, the output of the v^{th} neuron in the k^{th} layer, denoted as v_{vk}, is given by

$$v_{vk} = \sigma(b_{vvk} + \sum_{i=1}^{n_{k-1}} w_{uvk} v_{u(k-1)}), \tag{3.30}$$

where σ represents the activation function, b_{vk} represents the bias of the v^{th} neuron in the k^{th} layer, w_{uvk} represents the weighted connection between the u^{th} neuron in the $k-1^{th}$ layer to the v^{th} neuron in the k^{th} layer, and $v_{u(k-1)}$ represents the output of the u^{th} neuron in the $k-1^{th}$ layer.

In the case when the $k-1^{th}$ layer is not a fully connected layer and the k^{th} layer is a fully connected layer: assuming an input maps volume with m_{k-1} feature maps from the $k-1^{th}$ layer of size $(R_{k-1} \times C_{k-1} \times D_{k-1} \times m_{k-1})$, then the value of the j^{th} neuron at the k^{th} layer, denoted as v_{jk}, is given by

$$v_{jk} = \sigma(b_{jk} + \sum_{a=1}^{m_{k-1}} \sum_{r=1}^{R_{k-1}} \sum_{c=1}^{C_{k-1}} \sum_{d=1}^{D_{k-1}} w_{a,j,k}^{(r,c,d)} v_{a(k-1)}^{(r,c,d)}), \tag{3.31}$$

where σ represents the activation function, b_{jk} represents the bias of the j^{th} neuron in the k^{th} layer, $w_{a,j,k}^{(r,c,d)}$ represents the weight assigned at position (r, c, d) in the a^{th} feature map of the $k-1^{th}$ layer for the j^{th} neuron of the k^{th} layer, and $v_{a(k-1)}^{(r,c,d)}$ represents the value of the neuron at position (r, c, d) in the a^{th} output feature map of the $k-1^{th}$ layer.

In either case, the output of the fully connected layer will be a vector of size corresponding to the number of neurons of the k^{th} layer. The choice for activation functions between fully connected layers is similar to those of the nonlinearity layer. If the fully connected layer is the last layer of the CNN model, the softmax activation function is generally used for classification to predict the probabilities of the image labels.

The initial values for weights and biases in a fully connected layer are randomly sampled similarly as in convolutional layers.

3.3.3.5: Combination of Several Layers

Of the mentioned layers so far from the previous sections, the first hidden layer of a

typical CNN would be a convolution layer. In general, convolution layers are followed by nonlinearity layers but not necessarily always followed by pooling layers. Multiple combinations of convolution, nonlinearity and pooling layers are necessary for finding the complex features present in an image. In general, the output feature maps volume at the k^{th} layer, denoted as $\mathbb{V}_k$, can be expressed as a composition of the previous layers' outputs $\mathcal{A}_{k-1}$, $\mathcal{A}_{k-2}$, ..., $\mathcal{A}_1$, i.e.,

$$\mathbb{V}_k(\mathbb{F}) = (\mathcal{A}_{k-1} \circ \mathcal{A}_{k-2} \circ \cdots \circ \mathcal{A}_1)(\mathbb{F}), \tag{3.32}$$

where $\mathbb{V}_k$ is an order 3 tensor given a 2D input image and an order 4 tensor given a 3D input image. The resulting representation after several convolution and pooling layers would be small enough and rich enough in information on the features of the input image that then fully connected layers are used. These layers will then utilize all the information present to predict the input image to the CNN.

Figure 3.11 represents an example of a typical CNN architecture consisting of a convolutional and pooling layer at the start and fully connected layers at the end. An image is inputted into the convolutional layer where the resulting feature maps are then passed through a nonlinearity layer. From there, the feature maps would be passed as input to a pooling layer and the pooled feature maps would then be flattened into a vector. Finally, this vector would be inputted to the fully connected layers where then classification happens.

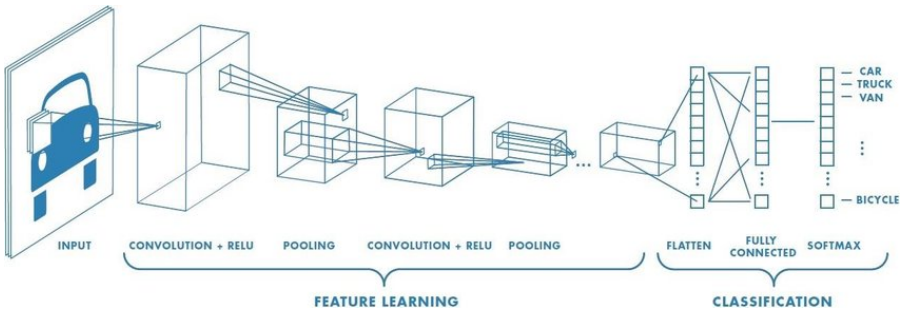


Figure 3.11: Figure representing an illustration of a typical CNN architecture. Taken from Hussain et al. (2018).

The values in the feature maps represent the patterns found by filters on the input image during the convolution operation. To better understand the operation of a CNN and its capability to extract meaningful features in an image requires interpreting the feature map activity in its hidden layers (Zeiler and Fergus, 2014). A *Deconvolutional Network* (deconvnet) can be described briefly as a multi-layered network which uses the same components of CNNs like convolutional and pooling layers but the other way

around, where deconvnets would map features to pixels (Zeiler et al., 2011). So instead of using convolutional, nonlinearity and pooling layers to respectively filter, rectify, and pool the input image into features, a deconvnet would instead unpool, rectify and filter the feature maps to reconstruct the feature maps into an image similar to the input pixel space of the input image. We refer to Zeiler et al. (2011) for more information on deconvnets. Zeiler and Fergus (2014) proposed the use of a deconvnet to visualize the feature maps outputted at every convolutional layer of a CNN, where the deconvnet would project the feature maps to the input pixel space. The deconvnet is attached to each of the layers of a CNN, where the feature maps at a layer would be processed by the deconvnet and would return an image from the input pixel space. From this, we would be able to visualize and compare the derived features of the input image at a layer with the original image that was inputted to the CNN.

Figure 3.12 shows a visual illustration of the output of several convolutional layers along with the original image to the leftmost. The first few layers would find some general features, such as the object boundaries. In the later layers, particularly at 21st layer, the convolutional layers become more focused on specific parts of the cat like the head of the cat and its eyes. These results could indicate that the classifier is trying to find a cat-like face to determine if the object in the image is a cat or not.

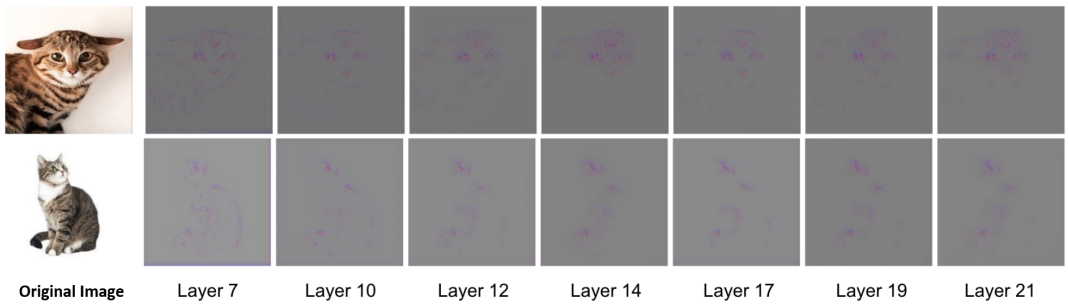


Figure 3.12: An illustration of features found from a deconvnet that was attached to a CNN. Taken from Yao (2020).

A CNN can utilize fixed filter sizes for all convolutional layers throughout the network but it is generally not necessarily optimal for all applications and all image resolutions (Han et al., 2018). A general approach is to use filters of larger sizes in convolutional layers around the start of the network and move to using filters of smaller sizes in subsequent convolutional layers (Chatfield et al., 2014). For an input image at a predefined fixed size, the optimal filter size for each convolutional layer would be either chosen via extensive experimentation (Kim et al., 2015) or by visualization using a deconvnet (Zeiler and Fergus, 2014). Another approach was considered by Han

et al. (2018), where the filter sizes of each convolutional layer are taken to be trainable parameters that are estimated by the model.

3.3.3.6: Translation Invariance in CNNs

Given an input image to the CNN and a function that is *translation invariant*, the classification result that we would acquire when the input image is shifted and then applied with the function would be the same result as if we had taken the function on the input image instead. As such, the shift has no effect on the output of the function on the input image (Mouton et al., 2021). Given an input image $\mathbb{F}$ and the translation group $\mathcal{T}$, a function f is said to be invariant to the translations of $\mathcal{T}$ if for $t \in \mathcal{T}$, we have

$$f(t(\mathbb{F})) = f(\mathbb{F}). \tag{3.33}$$

CNNs are generally praised for successfully classifying an object in the image regardless of where the object is located, i.e., that it is translation invariant. However, CNNs are not necessarily fully translation invariant since several authors found that, when compared to the classification accuracy acquired with input images, shifts of the input images could heavily affect classification accuracy (Azulay and Weiss (2019), Zhang (2019)). Such shifts of the input images could even be a single pixel shift which does not change the classification of the images by the human eye; however, it would lead to the model classifying them differently. They proved this for several popular CNN architectures and attributed this lack of translation invariance to stride values greater than 1 used in layers of the CNN.

A CNN consisting of only convolutional layers with no fully connected layers and only a stride of 1 throughout is considered to be translation invariant (Azulay and Weiss, 2019). No fully connected layers are utilized since a fully connected layer would expect input features to be located in the same spot for each image and would make the CNN not translation invariant. Instead, this CNN would make use of a *global average pooling* operator at the end of the network before classification is done. Using the notation that we have defined before in Section 3.3.3.3 for the k^{th} layer, a global average pooling operator behaves similarly as an average pooling operator but instead of using a filter of small size such as that of size $(P_k \times Q_k \times S_k \times m_{k-1})$ from before, the size of the filter would be similar as the image itself, i.e., of size $(R_{k-1} \times C_{k-1} \times D_{k-1} \times m_{k-1})$. However, such a network comes with a heavy computational price and is only possible in practice when considering input images of small sizes (Azulay and Weiss, 2019).

Given that stride values greater than 1 are used, there are several options to help improve translation invariance in a CNN. It has been observed that using pooling before a layer with stride could help *shiftability*, which is a weaker form of translation invariance

(Azulay and Weiss, 2019). Another option is to apply an 'anti-aliasing' filter introduced by Zhang (2019), i.e., for example, instead of applying a pooling operation with a stride of 2, we would instead apply a max pooling layer with stride 1 and a convolutional layer with stride 2. By changing the default max pooling layer with stride 2 of several popular CNN architectures with the anti-aliasing filter, Zhang (2019) improved the translation invariance of such CNNs.

Generalizability in a model captures the models ability to be able to predict unseen data given what it has seen during training. Improving translation invariance is important to help generalizability, i.e., a model that is generalizable should be invariant to transformations that would not affect the predicted outcome (Zhou et al., 2022). However, too much invariance can negatively affect generalization (Singhal et al., 2023). For example, to distinguish between images containing the numbers 6 and 9, where rotating either one gets the other. As such, improving translation invariance is a more instance-dependent task (Singhal et al., 2023). Mouton et al. (2021) have shown that stride values greater than 1 used in pooling layers could result in greater translation invariance over not using stride when they are used with large enough filter sizes; however, too large filter sizes could worsen test set performance metrics due to worse generalization from more information being disregarded due to the larger filters.

3.3.3.7: Other Notable Layers

For an ANN or a CNN, the gradient of each parameter is found in order to calculate how much the parameter needs to be updated to minimize an error function. However, this update is calculated given what all other neurons are doing too. As such, if there are neurons that are not contributing, this can result in the parameters of certain neurons being updated in a way where they compensate for the mistakes of other neurons (Srivastava et al., 2014). This can lead to complex *co-adaptations* which can cause overfitting, since this would weaken the generalizability of the model. Dropout is used in practice to address the problem of overfitting. Given that a dropout layer is added before a convolution or fully connected layer, it would prevent co-adaptation by randomly 'dropping' or temporarily removing neurons (including their connections) of the convolution or fully connected layer during training, and as such, makes it unreliable for neurons to depend on other neurons to correct their mistakes (Srivastava et al., 2014). Figure 3.13 represents two separate ANNs with fully connected layers, where one ANN uses dropout and the other ANN does not. When a neuron is dropped (shown as crossed neurons), all connections and their parameters are temporarily removed meaning that they would not get updated (see Figure 3.13). Note that dropout only occurs during training and not when making predictions during test time.

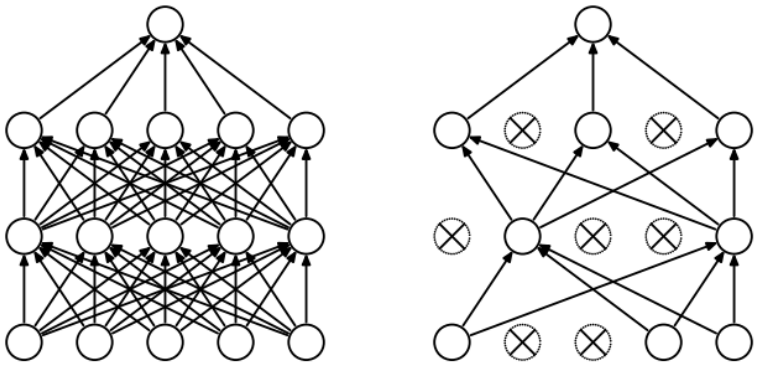


Figure 3.13: (Left) An ANN consisting of three fully connected and an output layer. (Right) Same ANN but with dropout added for each fully connected layer. Taken from Srivastava et al. (2014).

Each neuron has a fixed probability ρ to get dropped, where the value for ρ is a hyperparameter that is commonly taken to be equal to 0.5. At a probability of 1, no neurons are dropped, whereas at a probability of 0 every neuron would be dropped. Dropout may be implemented on any hidden or fully connected layer other than the final output layer with common probabilities taken to be around 0.5 to 0.8 (Srivastava et al., 2014). Dropout can also be implemented to the input layer, where a high probability of around 0.9 is recommended.

Dropout as a regularization technique is very effective and can replace or be combined with other forms of regularization (Srivastava et al., 2014). This is because, randomly dropping neurons during training injects noise, where, when optimizing the expected loss under that noise, this would be similar to adding a complexity penalty to the loss (Wager et al., 2013). In the case of CNNs, Srivastava et al. (2014) applied dropout to all convolutional layers and fully connected layers in a CNN architecture with great success. They compared the performance of the best performing CNNs using dropout and with CNNs not using dropout, where they found that CNNs using dropout in both convolutional and fully connected layers yielded better performance. Dropout is not commonly applied to convolutional layers in CNNs and is more generally applied to fully connected layers. Since the number of trainable parameters in convolutional layers is low to begin with, the risk of overfitting is lower. However, adding dropout to convolutional layers can reduce the chance of overfitting even more by improving generalization to test data (Srivastava et al. (2014), Wu and Gu (2015)).

Batch normalization layers were initially proposed in research to address a problem encountered during training known as *internal covariate shift*. During training, the distribution of the inputs to each layer is subject to change due to the randomness present

in parameter initialization as well as the update of parameters of the previous layers (Santurkar et al., 2018). This problem is known as internal covariate shift, and it hinders the training process by necessitating the use of reduced learning rates to maintain stability while training the model (Ioffe and Szegedy, 2015). To counter this problem, Ioffe and Szegedy (2015) proposed the implementation of batch normalization layers after convolution and fully connected layers in CNNs to re-center and re-scale the inputs for each layer to zero mean and a unit variance, which would normalize each minibatch during training. With batch normalization layers, the model would be more robust to randomness from parameter initialization and could use larger learning rates without worry of vanishing or exploding gradients. It also acts as a regularizing effect due to better generalization, and could replace dropout layers (Ioffe and Szegedy, 2015).

However, the subject of what batch normalization layers actually do is still in debate. Santurkar et al. (2018) had challenged the idea of batch normalization reducing internal covariate shift by comparing the performance of three models: the first where no batch normalization was used, the second where batch normalization was added after each layer, and the last where batch normalization was used at each layer but with noise adding during training. The noise added in the third model directly introduces covariate shift. The results show that the accuracy of both the second and third models being similar to one another and both outperforming the first model, thus possibly implying that the reason for batch normalization layers improving performance is not due to reducing internal covariate shift. Also, a deep network using batch norm layers is still subject to gradient explosion at initialization time (Yang et al., 2019). Regardless of what they do, they are popularly used in practice for speeding up model training and as a regularizing effect.

Given that the k^{th} layer represents the batch normalization layer, let us assume that the input maps volume at the k^{th} layer comprises of m_{k-1} feature maps of size ($R_{k-1} \times C_{k-1}$) from the $k-1^{th}$ layer. Let B represent the batch size of each minibatch, such that $\mathbb{V}_{k-1}$ is the input to the batch normalization layer with size ($R_{k-1} \times C_{k-1} \times m_{k-1} \times B$). Given that the j^{th} output feature map inputted to the k^{th} layer, denoted as $\mathbb{V}_{j(k-1)}$, will be of size ($R_{k-1} \times C_{k-1} \times B$), we find the mean $\mu_{j(k-1)}$ and variance $\sigma^2_{j(k-1)}$ as follows

$$\mu_{j(k-1)} = \frac{1}{R_{k-1}C_{k-1}B} \sum_{r=1}^{R_{k-1}} \sum_{c=1}^{C_{k-1}} \sum_{b=1}^{B} v_{rcb},$$

and

$$\sigma^2_{j(k-1)} = \frac{1}{R_{k-1}C_{k-1}B} \sum_{r=1}^{R_{k-1}} \sum_{c=1}^{C_{k-1}} \sum_{b=1}^{B} (v_{rcb} - \mu_{j(k-1)})^2,$$

where $\mu_{j(k-1)}$ and $\sigma^2_{j(k-1)}$ represent the mean and variance of the minibatch for the j^{th} feature map being inputted to the k^{th} layer. Given that $\boldsymbol{\mu}_{k-1}$ and $\boldsymbol{\sigma}^2_{k-1}$ is an m_{k-1}-vector containing the m_{k-1} values of $\mu_{j(k-1)}$ and $\sigma^2_{j(k-1)}$, respectively, then we define $\boldsymbol{\mu}_{k-1}$ and $\boldsymbol{\sigma}^2_{k-1}$ to be of size $(R_{k-1} \times C_{k-1} \times m_{k-1} \times B)$ by element-wise taking

$$\mu_{rcjb} = \mu_{j(k-1)} \quad \text{and} \quad \sigma^2_{rcjb} = \sigma j(k-1)^2,$$

for all $r \in R_{k-1}$, $c \in C_{k-1}$, and $b \in B$. The output of the batch normalization layer would be

$$\mathbb{V}_k = \frac{(\mathbb{V}_{k-1} - \boldsymbol{\mu}_{k-1})}{\sqrt{\boldsymbol{\sigma}^2_{k-1} + \varepsilon}}, \tag{3.34}$$

where ε represents a tensor of size $(R_{k-1} \times C_{k-1} \times m_{k-1} \times B)$ filled with fixed arbitrarily small constants added for numerical stability typically valued at $1\text{x}10^{-6}$ or even $1\text{x}10^{-8}$ (Shi et al., 2020). $\mathbb{V}_k$ would then be transformed using a linear transformation (Ioffe and Szegedy, 2015). This is done since this normalization would not necessarily work well with the activation function from the nonlinearity layer, and the linear transformation represents a way for the model to adjust the normalization (Ioffe and Szegedy, 2015). As such, this is followed by the following linear transformation:

$$\delta_{jk} \odot \mathbb{V}_k + \tau_{jk}, \tag{3.35}$$

where δ_{jk} and τ_{jk} are tensors of size $(R_{k-1} \times C_{k-1} \times m_{k-1} \times B)$ with scale and shift parameters which will be tuned during training.

With the fundamental building blocks of CNNs established, the discussion proceeds to a literature review of notable architectures that demonstrate how these components are combined in practice.

3.4: Literature Review of CNNs

In this section, we will review several popular architectures in the history of CNNs, including ResNet, which we will utilize in our application.

3.4.1: LeNet-5 and AlexNet Architectures

Traditionally, a few decades ago, a two-stage approach was employed to solve image classification problems. The first stage would first extract handcrafted features from images using feature descriptors, whereas in the second stage, this information would then serve as input to a trainable classifier. However, the accuracy of the classification heavily depends on the design employed at the feature extraction stage, which was not

an easy task (LeCun et al., 1998). Over the past decade, deep learning models have been shown to overcome the challenges encountered in the traditional approach.

The work by Fukushima (1980) is universally considered as a predecessor to CNNs. Fukushima's work was itself inspired by other works regarding the structures of the visual system, most notably the work by Hubel and Wiesel (1962). From their work on the visual system of cats, Hubel and Wiesel (1962) discovered the receptive field which forms the biological basis of a CNN. LeNet-5 developed by LeCun et al. (1990) and LeCun et al. (1998) was a multi-layer CNN used to classify size (32 × 32) handwritten digits and it established the framework of CNNs. LeNet-5 was a rather simple network by todays standards where in its first five layers it alternated between convolutional and average pooling layers and it ended with a fully connected layer and an output layer using a softmax classifier. Figure 3.14 represents the architecture of LeNet-5, which has three convolution layers with an average pooling layer in between and two fully connected layers at the end. However, constrained by the limited computational resources and data scarcity, LeNet-5 did not perform so well on complex problems.

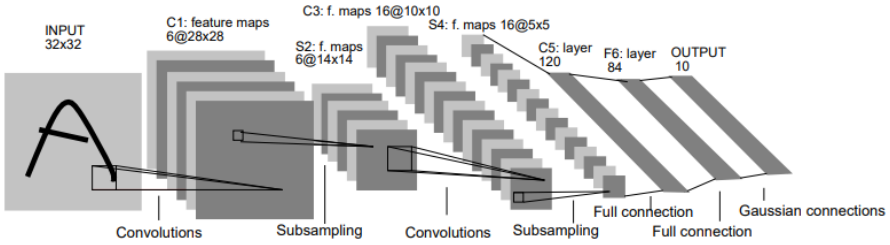


Figure 3.14: The architecture of LeNet-5. Above each layer, there is also listed the number of feature maps and the size of the layers outputs. Taken from LeCun et al. (1998).

Due to the sharp increase in computational power that came from technology advancements from the 2000s onward and increased data availability, the domain of ML flourished and has dramatically risen in popularity. This is due to the need for complex models to solve complex problems often found in practice, and due to this, deep CNNs have found fertile ground for success. Naturally, CNNs have grown in popularity due to the better efficiency found when training using GPU computing (Chellapilla et al., 2006).

However, we can say that the popularity of deep CNNs skyrocketed when their power was shown during the publication of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012 (Russakovsky et al., 2015). Krizhevsky et al. (2012), who won ILSVRC in 2012, used a deep CNN known as AlexNet to classify approximately 1.2 million images into the 1000 labels of the ImageNet challenge with record-

breaking results at the time. From there, subsequent competitions had the majority of their entries being CNN-based networks with results improving year by year (Simonyan and Zisserman (2014), Zeiler and Fergus (2014)). The architecture of AlexNet was built upon that of LeNet-5, but AlexNet expands on this by increasing the number of layers from 5 to 8 and introducing the concept of stacking convolutional layers. Three convolutional layers are stacked together after the initial two convolutional-pooling layers, which are followed by two fully connected layers and an output layer with a softmax classifier (see Figure 3.15).

The increase in depth is to make the CNN more applicable to more diverse categories of images, albeit at the risk of overfitting. To counter the risk of overfitting, dropout was applied to the fully connected layers. ReLu was used as the main activation function throughout the network over the logistic function. Krizhevsky et al. (2012) and all other architecture that we will mention in Section 3.4.2 utilized the ReLu activation function over the logistic or hyperbolic tangent functions since deep networks train faster using ReLu by alleviating the problem of vanishing gradients to some extent. Other notable differences between AlexNet and LeNet-5 are that AlexNet used max pooling over average pooling and utilized larger size filters at the initial convolution layers. AlexNet initially used filters of size (11 × 11) and (5 × 5) in its convolutional layers, before settling on (3 × 3) size filters in the remaining convolution layers.

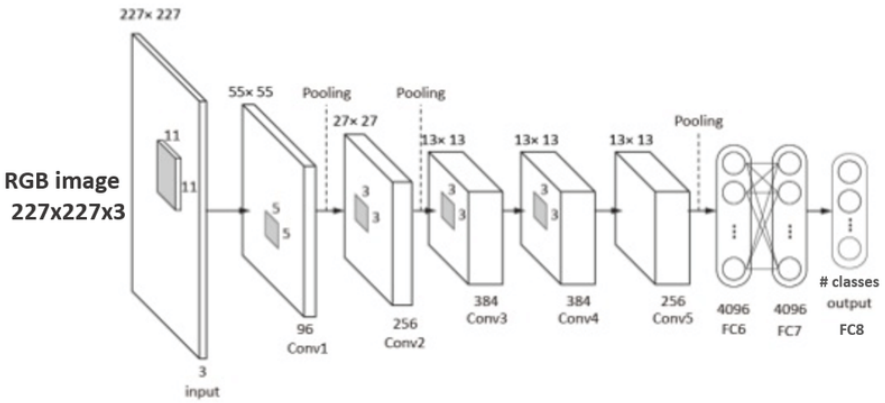


Figure 3.15: The architecture of AlexNet. The number of feature maps at each layer can be seen below the layer, while the output image size is above the layer (Khvostikov et al. (2018)).

3.4.2: ResNet Architecture

Goodfellow et al. (2013) found that deeper networks with more layers generally outperform wider networks with the same number of parameters. They argue that wider networks would need to be significantly more complex to match the performance of

deeper ones. This insight has influenced modern practice, favoring deeper architectures (He et al., 2015; Simonyan and Zisserman, 2014; Szegedy et al., 2016).

As CNNs grew deeper, they became harder to train due to the degradation problem, where model performance worsens with increasing depth. This issue is not due to overfitting or vanishing/exploding gradients, especially when using normalized inputs and batch normalization (He et al., 2015). Both training and test errors increase with depth, indicating an inherent limitation. Nichani et al. (2020) also found a critical depth beyond which performance declines.

To address this, He et al. (2015) introduced ResNet, a deep CNN architecture that uses skip connections with identity mappings (see Figure 3.16). Given an input $\mathbf{x}$ and nonlinear output $F(\mathbf{x})$, the skip connection adds $\mathbf{x}$ directly to $F(\mathbf{x})$. This helps bypass the degradation problem by enabling the model to approximate identity mappings when necessary, effectively allowing it to skip layers (He et al., 2015).

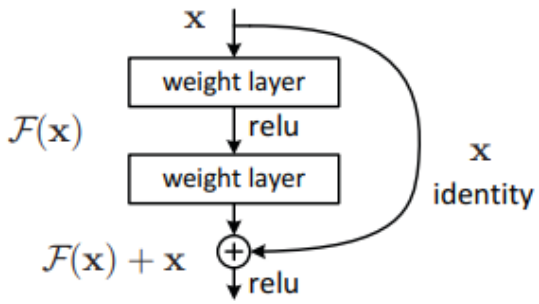


Figure 3.16: An example of a skip connection in ResNet. Taken from He et al. (2015).

Skip connections were initially introduced by Ripley (1996), where they were added in Multilayer Perceptrons (MLPs) by a linear layer connecting the network input to the output. HighwayNet utilized skip connections similar to ResNet and was the first proposed very deep ANN using skip connections with gating functions (Srivastava et al., 2015). This gating mechanism originated from the LSTM network framework, which is used for regulating information flow (Hochreiter and Schmidhuber, 1997). The difference in the skip connections between HighwayNet and ResNet is that the former are data-dependent in that they are trainable parameters whereas the latter are parameter-free since they use identity shortcuts (He et al., 2015).

Following the review of these notable CNN architectures, our attention now shifts to the optimization methods that enable their effective training, with a focus on adaptive algorithms.

3.5: Adaptive Optimization Algorithms

The learning rate in Stochastic Gradient Descent (SGD) is a challenging hyperparameter to tune and often requires adjustment during training (Zhang, 2018). To address this, adaptive optimization algorithms such as *AdaGrad* (Duchi et al., 2011), *AdaDelta* (Zeiler, 2012), *RMSProp* (Tieleman and Hinton, 2012), and *Adam* (Kingma and Ba, 2017) were developed. Instead of multiplying every parameter update by a single global number as in SGD, these adaptive optimization algorithms compute a separate scaling factor for each parameter from that parameter's own past gradients. This is referred to as *element-wise scaling*, and these methods dynamically adjust learning rates using it to accelerate training. Adam, in particular, combines the strengths of AdaGrad and RMSProp, and has become a popular choice in practice due to its computational efficiency and ease of implementation.

However, studies have shown that sometimes deep neural networks trained with adaptive methods may have a worse generalization than those trained using SGD with momentum (Wilson et al., 2017), and may suffer from convergence issues due to unstable learning rates (Luo et al., 2019). To address these issues, variants such as *AMSGrad* and *Adam with decoupled Weight decay (AdamW)* (Loshchilov and Hutter, 2017) have been proposed. AdamW replaces standard L_2 regularisation with weight decay, leading to improved generalisation (Landro et al., 2021).

We have chosen to cover adaptive optimization algorithms since they provide rapid convergence and require less hyperparameter tuning compared to SGD with momentum. This will be useful in Chapter 5 to reduce overall training time given our limited computational resources, since we will be dividing our runs using either SGD with momentum or AdamW as our optimization algorithms. The coming sections will first review AdaGrad, AdaDelta, and RMSProp, followed by Adam and AdamW. We will be covering these other adaptive optimization algorithms since we believe it will be easier to explain AdamW itself, given that AdamW came about as a combination of their ideas.

3.5.1: AdaGrad

AdaGrad is an optimization algorithm particularly effective for sparse data, i.e., datasets in which most feature values are zero, as it adjusts the learning rate for each parameter individually (Duchi et al., 2011). It reduces the learning rate for frequent features (the zero values) and increases it for infrequent ones (the non-zero values) (Ruder, 2017), allowing the model to learn more effectively from rare but potentially informative features. For example, if most images suggest it's raining, an image without rain becomes

unusual and thus valuable. However, infrequent features are seen less often, limiting learning opportunities. AdaGrad addresses this by assigning higher learning rates to such features, promoting better parameter updates based on relevance rather than frequency. An example of a domain where AdaGrad has been used is the natural language processing domain, where Pennington et al. (2014) had used it to train glove word embeddings, since rarely used words would require higher learning rates compared to words that are frequently used.

We recall from equation (2.20) that the SGD weight update rule at iteration q is represented as follows:

$$w_{ajk}^{q} = w_{ajk}^{q-1} - \gamma \frac{\partial E^{q}}{\partial w_{ajk}^{q-1}}, \tag{3.36}$$

where γ is a fixed learning rate. It is generally recommended that the initial learning rate be 0.01.

At an iteration q, AdaGrad alters the learning rate γ for a parameter w_{ajk}^{q} based on past gradients calculated for that parameter. To do this, it calculates h_{ajk}^{q} which is the sum of the squares of the gradients w.r.t. w_{ajk} up to iteration q (Ruder, 2017), i.e.,

$$h_{ajk}^{q} = \sum_{q^*=1}^{q} \left(\frac{\partial E^{q^*}}{\partial w_{ajk}^{q^*}}\right)^2. \tag{3.37}$$

As such, from the SGD weight update rule given in equation (3.36), the AdaGrad weight update rule is updated as follows

$$w_{ajk}^{q} = w_{ajk}^{q-1} - \frac{\gamma}{\sqrt{h_{ajk}^{q-1} + \varepsilon}} \cdot \frac{\partial E^{q}}{\partial w_{ajk}^{q-1}}, \tag{3.38}$$

where ε is a very small negligible value generally taken to be around $1{\times}10^{-8}$ to avoid dividing by 0.

Since the parameter block for layer k can be a matrix or tensor depending on layer type and filters/batch size amount, we will be representing it in tensor form as $\mathbb{W}_k^q$, where we will be following the same notation logic for matrices/tensors dependent on $\mathbb{W}_k^q$. We denote $\mathbb{H}^q$ to be a diagonal matrix/tensor containing the diagonal entries h_{ajk}^{q}. The diagonal of $\mathbb{H}^q$ has the sum of the squares of the previous gradients w.r.t. all parameters h_{ajk}^{q}. This means that the vectorization of equation (3.38) can be performed by doing an element-wise matrix-vector multiplication $\odot$ between $\mathbb{H}^{q-1}$ and $\frac{\partial E^q}{\partial \mathbb{W}_k^{q-1}}$, i.e.,

$$\mathbb{W}_k^{q} = \mathbb{W}_k^{q-1} - \frac{\gamma}{\sqrt{\mathbb{H}^{q-1}} + \varepsilon \mathbb{I}} \odot \frac{\partial E^{q}}{\partial \mathbb{W}_k^{q-1}}, \tag{3.39}$$

where $\mathbb{I}$ is an identity matrix/tensor of similar size as $\sqrt{\mathbb{H}^{q-1}}$.

Note that a known disadvantage of using AdaGrad is that it shrinks the learning rate to exceedingly small values which effectively nulls the ability of the algorithm to continue learning. This happens because the number of squared gradients in the denominator will accumulate as the number of iterations increase, and as such, it will eventually become too large leading to the $\frac{\gamma}{\sqrt{h_{ajk}^{q-1}+\varepsilon}}$ term to be incredibly small. Also, the starting learning rate for AdaGrad is delicate, since the learning rate will continue to decrease throughout training, and too small of a value would block any training when the initial gradients are large (Zeiler, 2012). Choosing a larger initial learning rate would help against this issue.

3.5.2: AdaDelta and RMSprop

Later, an extension to AdaGrad was proposed by Zeiler (2012), known as AdaDelta, to counteract the issue of AdaGrads learning rate becoming infinitesimally small. He restricted the window of accumulated past gradients to only take a certain number q^* of previous iterations instead of considering all past iterations. Using this window of accumulated past gradients, the algorithm would be able to avoid the denominator of AdaGrad growing exceedingly large and instead, it would behave as a local estimate of past gradients (Zeiler, 2012).

Instead of using $\mathbb{H}^{q-1}$, an exponentially decaying average of the q^* past squared gradients is used since it is considered more efficient than storing the q squared gradients themselves. This allows the model to forget earlier gradients and focus on more recent ones. For ease of notation, we will be using the notation g_{ajk}^q to refer to the gradient of the error function E with respect to the parameter w_{ajk}^q at the q^{th} iteration and the notation $\mathbb{G}^q$ to represent a matrix/tensor with all respective g_{ajk}^q entries at the q^{th} iteration, i.e.,

$$g_{ajk}^q = \frac{\partial E^q}{\partial w_{ajk}^{q-1}} \quad \text{and} \quad \mathbb{G}^q = \frac{\partial E^q}{\partial \mathbb{W}_k^{q-1}}. \tag{3.40}$$

At iteration q, the running average $\mathbb{E}[(\mathbb{G}^q)^2]$ is computed using the running average of the previous iteration $\mathbb{E}[(\mathbb{G}^{q-1})^2]$ and the current gradient $\mathbb{G}^q$ as follows:

$$\mathbb{M}_2^q = \mathbb{E}[(\mathbb{G}^q)^2] = \beta\mathbb{E}[(\mathbb{G}^{q-1})^2] + (1-\beta)(\mathbb{G}^q)^2, \tag{3.41}$$

where $(\mathbb{G}^q)^2$ indicates the element-wise square multiplication $\mathbb{G}^q \odot \mathbb{G}^q$ and β is a constant that controls the decay rate of $\mathbb{M}_2^q$ with typical value at around 0.9 (Zeiler, 2012). At an iteration q, the learning rate γ is sometimes replaced with the *step size* γ^q which

represents the learning rate at iteration q decayed at a rate of $\frac{1}{\sqrt{q}}$, i.e., $\gamma^q = \frac{\gamma}{\sqrt{q}}$. Given equation (3.39), we instead change $\mathbb{G}^{q-1}$ with $\mathbb{M}_2^{q-1}$, i.e.,

$$\mathbb{W}_k^q = \mathbb{W}_k^{q-1} - \frac{\gamma^q}{\sqrt{\mathbb{M}_2^{q-1} + \varepsilon}} \frac{\partial E^q}{\partial \mathbb{W}_k^{q-1}}. \tag{3.42}$$

The denominator $\sqrt{\mathbb{M}_2^{q-1}} + \varepsilon$ of equation (3.42) represents the *Root Mean Squared Error (RMSE)* criterion of previous squared gradients up to iteration q. Zeiler (2012) goes on to show in Section 3.2 of his paper that AdaDelta does not require the learning rate under certain assumptions and that it can be eliminated from the weight update rule by replacing it with the RMSE of parameter updates.

Around the same time, RMSprop was also proposed as an extension to AdaGrad by Tieleman and Hinton (2012) in order to counter its limitation of radically diminishing learning rates. Similar to AdaDelta, RMSprop utilizes a decaying average of partial squared gradients in the calculation of the learning rate for each parameter. As such, RMSprop utilizes a similar weight update rule as in equation (3.42) with the running average $\mathbb{M}_2^q$ given in equation (3.41). In fact, this change in the weight update rule from using the sum of squared gradients to an exponentially decaying average of squared gradients is the only difference between AdaGrad and RMSprop (Ruder, 2017).

The main difference between AdaDelta and RMSprop is that RMSprop still uses the learning rate as in AdaGrad whereas AdaDelta gets rid of it. As such, AdaDelta does not need to set an initial learning rate whereas RMSprop does like AdaGrad.

3.5.3: Adam

The individual adapted learning rates for different parameters of Adam are computed by using estimates of the first and second moments of the past gradients. This allows Adam to provide stable and parameter-specific updates even when the objective function changes over time due to noisy minibatch sampling or shifting data distributions. For example, in natural language processing, sentiment analysis is the process of analyzing text to determine whether it conveys a positive, neutral, or negative sentiment. The objective function for a sentiment analysis of a social media platform represents an environment in which user opinions/language used change day-to-day, i.e., it would be non-stationary and need to continuously adapt as new data comes in. In such a scenario, Adam would be efficient, since the exponentially decaying averages of past gradients smooth out the noise of past days. Also, Adam retains AdaGrads effectiveness for use with sparse data, while also incorporating RMSprops robustness to noisy gradient estimates through exponential averaging.

Similar to both AdaDelta and RMSprop, Adam calculates an exponentially decaying average of past squared gradients $\mathbb{M}_2^q$; however, it differs from both by also calculating an exponentially decaying average of past gradients $\mathbb{M}_1^q$, i.e.,

$$\mathbb{M}_1^q = \beta_1 \mathbb{M}_1^{q-1} + (1-\beta_1)\mathbb{G}^q \quad \text{and} \quad \mathbb{M}_2^q = \beta_2 \mathbb{M}_2^{q-1} + (1-\beta_2)(\mathbb{G}^q)^2, \tag{3.43}$$

where $\mathbb{M}_1^q$ and $\mathbb{M}_2^q$ are estimates of the first moment and the second moment, respectively, at the q^{th} iteration. β_1 and β_2 are hyperparameters defined as the exponential decay rates of the first and second moment estimates, respectively. They are generally referred to as momentum parameters (Polyak, 1964). Moreover, β_1 and β_2 are recommended to be valued at 0.9 and 0.999 respectively, although they can both range anywhere from [0, 1) (Kingma and Ba, 2017). RMSprop represents a special case of Adam where $\beta_1 = 0$.

At the start of training, $\mathbb{M}_1^q$ and $\mathbb{M}_2^q$ are initialized as matrices of zeroes at first. In situations where β_1 and β_2 are close to 1 and during the initial iterations of training, Kingma and Ba (2017) had observed that $\mathbb{M}_1^q$ and $\mathbb{M}_2^q$ are biased towards zero. To fight against this bias, bias-corrected estimates of the first and second moments of the q^{th} iteration from equation (3.43) are found as follows:

$$\widetilde{\mathbb{M}}_1^q = \frac{\mathbb{M}_1^q}{1-(\beta_1)^q} \quad \text{and} \quad \widetilde{\mathbb{M}}_2^q = \frac{\mathbb{M}_2^q}{1-(\beta_2)^q}, \tag{3.44}$$

where $(\beta_1)^q$ and $(\beta_2)^q$ denote the values β_1 and β_2, respectively, to the power of q.

Using both $\widetilde{\mathbb{M}}_1^q$ and $\widetilde{\mathbb{M}}_2^q$ in equation (3.44), the weight update rule for Adam is given as follows:

$$\mathbb{W}_k^q = \mathbb{W}_k^{q-1} - \gamma^q \frac{\widetilde{\mathbb{M}}_1^q}{\sqrt{\widetilde{\mathbb{M}}_2^q} + \varepsilon}. \tag{3.45}$$

$\mathbb{M}_2^q$ stores smoothed values of gradients $(\mathbb{G}^q)^2$ which are utilized to manage the learning rates by normalizing the gradients $\mathbb{M}_1^q$ by $\sqrt{\widetilde{\mathbb{M}}_2^q} + \varepsilon$ as in equation (3.45) (Loshchilov and Hutter, 2017).

3.5.4: AdamW

Wilson et al. (2017) had proven that generalization performance for Adam is generally worse than SGD, even when they would have better training performance. Regularization in Adam is used both to penalize large parameters and to stabilize noisy updates, where L_2 regularization, as in equation (2.22), is commonly applied in Adam. AdamW is a variant of the Adam optimization algorithm that replaces the typical implementation of using L_2 regularization in Adam with weight decay regularization instead. AdamW

has similar definitions for g^q_{ajk}, $\mathbb{G}^q$ and $\widetilde{\mathbb{M}}^q_1$ and $\widetilde{\mathbb{M}}^q_2$ as in Adam; however, it introduces weight decay in the weight update rule as follows:

$$\mathbb{W}^q_k = \mathbb{W}^{q-1}_k - \gamma \left(\frac{\widetilde{\mathbb{M}}^q_1}{\sqrt{\widetilde{\mathbb{M}}^q_2 + \varepsilon}} + \lambda_{WD} \mathbb{W}^q_k \right). \tag{3.46}$$

Loshchilov and Hutter (2017) investigated whether it was better to use L_2 regularization or weight decay regularization for the training of very deep networks using SGD or adaptive optimization algorithms. This was brought about due to the interest in improving the generalization performance of adaptive optimization algorithms to allow it to be able to compete with the better generalization performance of SGD with momentum.

Given that we have a SGD optimizer using weight decay regularization with regularization parameter λ_{WD} and a SGD optimizer using L_2 regularization with regularization parameter λ_{LR} with both optimizers having the same learning rate of γ, then it can be found that there is an equivalence relationship between L_2 regularization and weight decay regularization when we have $\lambda_{LR} = \frac{\lambda}{\gamma}$. This shows that L_2 regularization or weight decay regularization are analogous for SGD, although this equivalence would not hold otherwise for other values of λ_{LR}. Note that the regularization parameter is 'coupled' with the learning rate. Loshchilov and Hutter (2017) go on to propose a variant of SGD known as *SGD with momentum using decoupled weight decay* (SGDW) which 'decouples' this equivalence between λ_{WD} and γ. The impact of decoupling hyperparameters from each other makes them more independent, and as such, eases hyperparameter optimization since their optimal value can be found much easier.

We now go on to consider the use of L_2 regularization or weight decay regularization in adaptive optimization algorithms. Both regularization methods are analogous for SGD; however, this is not the case for adaptive optimization algorithms. This is because using L_2 regularization in Adam leads to weights being regularized less compared to how they would have been if using weight decay instead (Loshchilov and Hutter, 2017). Proposition 3.1 goes on to show how decoupled weight decay regularization and L_2 regularization are not analogous for adaptive optimization algorithms.

Proposition 3.1: *At the q^{th} iteration, given that there is an optimizer that has iterates $\mathbb{W}^q_k = \mathbb{W}^{q-1}_k - \gamma \mathbb{N}^{q-1} \frac{\partial E^q}{\partial \mathbb{W}^{q-1}_k}$ when run on the error function E without weight decay, and iterates $\mathbb{W}^q_k = (1 - \lambda_{WD}) \mathbb{W}^{q-1}_k - \gamma \mathbb{N}^{q-1} \frac{\partial E^q}{\partial \mathbb{W}^{q-1}_k}$ when run on E with weight decay, where $\mathbb{N}^{q-1} \neq k\mathbb{I}$ and $k \in \mathbb{R}$. Then, for this optimizer there exists no L_2 regularization parameter λ_{LR} such that running the optimizer on the error function with L_2 regularization $E_{L_2 reg} = E + \frac{\lambda_{LR}}{2} \|\mathbb{W}\|^2_2$ without weight decay is equivalent the optimizer on E with decay $\lambda_{WD} \in \mathbb{R}^+$.*

Proof: The iterates of the optimizer without weight decay on the L_2 regulated error E_{L_2reg} are

$$\mathbb{W}_k^q = \mathbb{W}_k^{q-1} - \gamma\lambda_{LR}\mathbb{N}^{q-1}\mathbb{W}_k^{q-1} - \gamma\mathbb{N}^{q-1}\frac{\partial E^q}{\partial \mathbb{W}_k^{q-1}}, \tag{3.47}$$

while the iterates of the optimizer with weight decay having parameter λ_{WD} on the error function E are

$$\mathbb{W}_k^q = (1-\lambda_{WD})\mathbb{W}_k^{q-1} - \gamma\mathbb{N}^{q-1}\frac{\partial E^q}{\partial \mathbb{W}_k^{q-1}} = \mathbb{W}_k^{q-1} - \lambda_{WD}\mathbb{W}_k^{q-1} - \gamma\mathbb{N}^{q-1}\frac{\partial E^q}{\partial \mathbb{W}_k^{q-1}}. \tag{3.48}$$

For these iterates to be equal for all iterations, it must be that at the q^{th} iteration, $\lambda_{WD}\mathbb{W}_k^{q-1} = \gamma\lambda_{LR}\mathbb{N}^{q-1}\mathbb{W}_k^{q-1}$. However, this holds only if $\mathbb{N}^{q-1} = k\mathbb{I}$, which is not possible. As such, there exists no λ_{LR} that makes the iterates equal and hence, L_2 regularization and weight decay regularization are not analogous for adaptive optimization algorithms. ■

Loshchilov and Hutter (2017) mention that Adam has a poor generalization performance since, when compared to SGD, L_2 regularization is not as efficient on Adam as it is for SGD. When using L_2 regularization in Adam, the notation of g_{ajk}^q and $\mathbb{G}^q$ will be updated to the following:

$$g_{ajk}^q = \frac{\partial E^q}{\partial w_{ajk}^{q-1}} + \lambda_{WD}w_{ajk}^q \quad \text{and} \quad \mathbb{G}^q = \frac{\partial E^q}{\partial \mathbb{W}_k^{q-1}} + \lambda_{WD}\mathbb{W}_k^q. \tag{3.49}$$

In the case when we will have large values of iteration q, $(\beta_1)^q$ and $(\beta_2)^q$ will go to 0 meaning that $\widetilde{\mathbb{M}}_1^q$ and $\widetilde{\mathbb{M}}_2^q$ will be equal to $\mathbb{M}_1^q$ and $\mathbb{M}_2^q$ respectively. This means that we will have the weight update rule of Adam be as follows:

$$\mathbb{W}_k^q = \mathbb{W}_k^{q-1} - \gamma\frac{\mathbb{M}_1^{q-1} + (1-\beta_1)\mathbb{G}^q}{\sqrt{\beta_2\mathbb{M}_2^{q-1} + (1-\beta_2)(\mathbb{G}^q)^2 + \varepsilon}}. \tag{3.50}$$

Since $\mathbb{G}^q$ is defined as in equation (3.49), $\frac{\partial E^q}{\partial w_{ajk}^q}$ is normalized as usual; however, the weight decay $\lambda_{WD}\mathbb{W}_k^q$ will also be normalized. Since the weights are not valued proportionally to their original value in equation (3.50), this leads 'to the relative decay being weaker for weights with large gradients' (Loshchilov and Hutter, 2017). As such, using Adam with L_2 regularization leads to weights that have large gradients in the error function not being regularized as much as they would be if we were using weight decay regularization instead, where weight decay regularization would regularize all weights by the same factor.

The difference between using L_2 regularization and weight decay regularization is that the latter decouples the weight decay from the weight update rule similar to what

is done in SGDW. From their analysis, they had compared Adam with L_2 regularization against AdamW and found that the latter achieved improved generalization performance for the test dataset over the former.

Algorithm 3.1 represents a full algorithmic formulation for AdamW, mentioning steps that are common with the adaptive optimization algorithms we have mentioned in the previous sections.

Algorithm 3.1 AdamW

Require: learning rate γ, decay rates $\beta_1, \beta_2 \in [0, 1)$, stability constant ϵ, weight decay λ_{WD}, assuming k^{th} layer
Initialize: $\mathbb{W}_k^0$ (parameters), $\mathbb{M}_1^0 \leftarrow 0$, $\mathbb{M}_2^0 \leftarrow 0$, $q \leftarrow 0$
for Iteration $q = 1$ to Q **do**
  Compute gradient: $\mathbb{G}^q = \partial E^q / \partial \mathbb{W}_k^{q-1}$
  First-moment update: $\mathbb{M}_1^q \leftarrow \beta_1 \mathbb{M}_1^{q-1} + (1 - \beta_1)\mathbb{G}^q$ {*not in Adagrad, appears in Adam/RMSprop/AdaDelta*}
  Second-moment update: $\mathbb{M}_2^q \leftarrow \beta_2 \mathbb{M}_2^{q-1} + (1 - \beta_2)(\mathbb{G}^q)^2$ {*same idea as RMSprop/AdaDelta; Adagrad uses cumulative sum instead*}
  Bias correction: $\mathbb{M}_1^q \leftarrow \mathbb{M}_1^q / (1 - (\beta_1)^q)$, $\mathbb{M}_2^q \leftarrow \mathbb{M}_2^q / (1 - (\beta_2)^q)$ {*unique to Adam/AdamW*}
  Parameter update: $\mathbb{W}_k^q \leftarrow \mathbb{W}_{k-1}^{q-1} - \gamma\, \mathbb{M}_1^q / (\sqrt{\mathbb{M}_2^q} + \epsilon)$ {*adaptive step; same structure as Adagrad/RMSprop/AdaDelta/Adam/AdamW*}
  Apply decoupled weight decay: $\mathbb{W}_k^q \leftarrow \mathbb{W}_k^q - \gamma \lambda_{WD} \mathbb{W}_k^q$ {*AdamW extra step compared to Adam*}
end for
Return: $\mathbb{W}_k^Q$

Adam remains an exceptionally popular algorithm in practice; however, its behaviour is not well understood due to the well-known criticism by researchers regarding its convergence (Reddi et al. (2019), Zhang et al. (2022)). In our application for CNN, we took note of this during our hyperparameter optimization for Adam outside the default values of β_1, β_2. However, despite this criticism, Adam generally works well in practice with proper hyperparameter tuning or with the default hyperparameters recommended by Kingma and Ba (2017). In Appendix A, we covered the work by these researchers on this topic.

With architectural design and optimization methods established, the discussion now shifts to the next section, where we will discuss a fundamental theoretical guarantee for CNNs, i.e., the UAT.

3.6: Universal Approximation Theorem for CNNs

The UAT formalizes the ability for CNNs to approximate broad classes of functions given sufficient data and appropriate architecture, as this justifies their widespread application in various fields, including HPC. The UAT discussed in Section 2.3 focuses on ANNs but has also been extended to CNNs by several researchers (Bao et al., 2023; Yarotsky, 2018; Zhou, 2018, 2020). However, the approximation properties of CNNs are less well-explained in the literature, likely due to limited mathematical analysis (Kohler and Langer, 2020; Lin et al., 2021). Yarotsky (2018) showed that CNNs can act as universal approximators in infinite-dimensional settings, capable of approximating any translation-equivariant function with sufficient width. In finite-dimensional settings, Zhou (2020) demonstrated universality for CNNs using zero-padding and small filters, building on earlier work (Zhou, 2018). However, Zhou's results were for 1D convolutions with fixed-length filters intended for signal data, not 2D image data, and required unbounded depth. Petersen and Voigtlaender (2021) extended Zhou's findings to 2D ReLU CNNs with periodic padding for specific function classes (He et al., 2022).

Most of these studies exclude pooling layers. While some, like Cohen and Shashua (2016), incorporate pooling, they impose strict constraints such as using only 1×1 filters and no fully connected layers before classification. In this section, we follow He et al. (2022), who use an algebraic decomposition theorem to prove the universal approximation of deep ReLU CNNs with 2D filters and padding suitable for RGB images. They also relate single-hidden-layer ReLU ANNs to deep CNNs without pooling, showing the latter can match the former in approximation power. For simplicity, we present their proof using periodic padding only.

We refer to Appendix A for all proofs in this section.

3.6.1: Preliminaries

We will be following the notation of He et al. (2022), which we will define in this section. Instead of presenting filters with sizes $(P \times P)$, we use filters with sizes $((2P+1) \times (2P+1))$, where P denotes a positive integer. Given that we have a filter $\mathbf{G}$ of size $((2P+1) \times (2P+1))$, we will refer to its index going from $-P$ to P instead of the standard 1 to $2P+1$, i.e.,

$$\mathbf{G} = \begin{bmatrix} \mathbf{G}_{-P,-P} & \cdots & \mathbf{G}_{-P,P} \\ \vdots & \ddots & \vdots \\ \mathbf{G}_{P,-P} & \cdots & \mathbf{G}_{P,P} \end{bmatrix}.$$

However, the input data or resulting tensor from convolution will still have the standard indexing style. We also update our notation for the k^{th} layer from equation (3.14) to

where we replace the number of feature maps m_k to c_k.

The convolution operation defined in He et al. (2022) is the cross-correlation operation mentioned in equation (3.12), which does not adhere to the principles of commutativity or associativity, so it is assumed that

$$\mathbb{G}_1 * \mathbb{G}_2 * \mathbb{F} := \mathbb{G}_1 * (\mathbb{G}_2 * \mathbb{F}).$$

For images with size $\mathbb{R}^{R\text{x}R}$, where R denotes a positive integer such that $R > P$, He et al. (2022) considered $\mathbb{R}^{R\text{x}R}$ as a R^2-dimensional vector space with Frobenius norm.

3.6.2: Decomposition Theorem of Large Convolutional Filters

Given a filter of size (5×5), it can instead be represented as the combination of two size (3×3) filters:

Lemma 3.1: *Given a 2D input image* $\mathbf{F} \in \mathbb{R}^{R\text{x}R}$ *and* $\mathbf{G} \in \mathbb{R}^{5\text{x}5}$ *where* $R > 2$, *then there exists* $\mathbf{G}^1_{i,j}, \mathbf{G}^2_{i,j} \in \mathbb{R}^{3\text{x}3}$ *for* $i, j = -1, 0, 1$ *such that*

$$\mathbf{G} * \mathbf{F} = \sum_{i,j=-1,0,1} \mathbf{G}^1_{i,j} * \mathbf{G}^2_{i,j} * \mathbf{F}, \tag{3.51}$$

where $*$ *means the convolution operation with one filter and periodic padding.*

Lemma 3.1 was shown in the case of a convolution operation with a single filter; however, it can be extended to the case of a convolution operation with 9 filters of size (3×3), i.e.,

$$\mathbb{G}^1 = (\mathbf{G}^1_{-1,-1}, \mathbf{G}^1_{-1,0}, ..., \mathbf{G}^1_{1,1}) \in \mathbb{R}^{3\text{x}3\text{x}9}, \tag{3.52}$$

and

$$\mathbb{G}^2 = (\mathbf{G}^2_{-1,-1}, \mathbf{G}^2_{-1,0}, ..., \mathbf{G}^2_{1,1}) \in \mathbb{R}^{3\text{x}3\text{x}9}, \tag{3.53}$$

then the convolution operation written in equation (3.51) can be defined as

$$\mathbf{G} * \mathbf{F} = \mathbb{G}^1 * \mathbb{G}^2 * \mathbf{F}. \tag{3.54}$$

Lemma 3.1 is limited only to filters of size (5×5), so ideally the lemma would be extended to filters of different sizes. Theorem 3.4 does this, where for a filter $\mathbf{G}$ of size $((2P + 1) \times (2P + 1))$, it decomposes it into a combination of two filters $\mathbb{G}^1$ and $\mathbb{G}^2$.

Theorem 3.4: *Given a 2D input image* $\mathbf{F} \in \mathbb{R}^{R\text{x}R}$ *and* $\mathbf{G} \in \mathbb{R}^{(2P+1)\text{x}(2P+1)}$, *where* $R > P$, *then there exists* $\mathbf{G}^1_{i,j} \in \mathbb{R}^{3\text{x}3}$ *and* $\mathbf{G}^2_{i,j} \in \mathbb{R}^{(2P-1)\text{x}(2P-1)}$ *for* $i, j = -1, 0, 1$ *such that*

$$\mathbf{G} * \mathbf{F} = \sum_{i,j=-1,0,1} \mathbf{G}^1_{i,j} * \mathbf{G}^2_{i,j} * \mathbf{F}, \tag{3.55}$$

where $*$ *means the convolution operation with one filter and periodic padding.*

By applying Theorem 3.4 to decompose $\mathbf{G}^2_{i,j} \in \mathbb{R}^{(2P-1)\text{x}(2P-1)}$ repeatedly until the resulting matrix is size (3×3), we get Corollary 3.2.

Corollary 3.2: *Given a 2D input image* $\mathbf{F} \in \mathbb{R}^{R\text{x}R}$ *and* $\mathbf{G} \in \mathbb{R}^{(2P+1)\text{x}(2P+1)}$, *where* $R > P$, *then there exists* $\mathbf{G}^1_{i_v,j_v} \in \mathbb{R}^{3\text{x}3}$ *and* $\mathbf{G}^2_{(i_1,j_1),\ldots,(i_{P-1},j_{P-1})} \in \mathbb{R}^{3\text{x}3}$ *for* $i_v, j_v = -1, 0, 1$ *and layer number* $v = 1, \ldots, P-1$ *such that*

$$\mathbf{G} * \mathbf{F} = \sum_{i_{P-1},j_{P-1}} \cdots \sum_{i_1,j_1} \mathbf{G}^2_{(i_1,j_1),\ldots,(i_{P-1},j_{P-1})} * \mathbf{G}^1_{i_{P-1},j_{P-1}} * \cdots * \mathbf{G}^1_{i_1,j_1} * \mathbf{F}, \tag{3.56}$$

where $*$ *means the convolution operation with one filter and periodic padding.*

Corollary 3.2 can be extended similarly as we had done before from equations (3.52), (3.53) and (3.54) by grouping all $\mathbf{G}^1_{i_v,j_v}$ and $\mathbf{G}^2_{(i_1,j_1),\ldots,(i_{P-1},j_{P-1})}$ into $\mathbb{G}^1$ and $\mathbb{G}^2$, respectively, i.e.,

$$\mathbb{G}^1 = \{\mathbf{G}^1_{i_v,j_v} |\ i_v, j_v = -1, 0, 1 \text{ and } v = 1, \ldots, P-1\} \in \mathbb{R}^{3\text{x}3\text{x}9^{P-1}}, \tag{3.57}$$

and

$$\mathbb{G}^2 = \{\mathbf{G}^2_{(i_1,j_1),\ldots,(i_{P-1},j_{P-1})} |\ i_v, j_v = -1, 0, 1 \text{ and } v = 1, \ldots, P-1\} \in \mathbb{R}^{3\text{x}3\text{x}9^{P-1}}. \tag{3.58}$$

In this case, the dimensions of $\mathbb{G}^1$ and $\mathbb{G}^2$ are of size $(3 \times 3 \times 9^{P-1})$, which could be very large depending on the value of P. Notice that from equations (A.33) and (A.36), there are a lot of zero entries. He et al. (2022) go on to show that the number of non-zero entries in the 9^{P-1} filters of $\mathbb{G}^2$ is equal to $(2P-1)^2$. This gives birth to the following lemma:

Lemma 3.2: *Given a 2D input image* $\mathbf{F} \in \mathbb{R}^{R\text{x}R}$ *and* $\mathbf{G} \in \mathbb{R}^{(2P+1)\text{x}(2P+1)}$ *where* $R > P$ *and that we have defined* $\mathbb{G}^2$ *as in equation (3.58), then there is an index set*

$$\mathcal{I}_{P-1} \subset \{((i_1, j_1), \ldots, (i_{P-1}, j_{P-1}))|\ i_v, j_v = -1, 0, 1 \text{ and } v = 1, \ldots, P-1\} \tag{3.59}$$

with a cardinality of $(2P-1)^2$ *such that*

$$\mathbf{G} * \mathbf{F} = \sum_{((i_1,j_1),\ldots,(i_{P-1},j_{P-1}))\in\mathcal{I}_{P-1}} \mathbf{G}^2_{(i_1,j_1),\ldots,(i_{P-1},j_{P-1})} * \mathbf{G}^1_{i_{P-1},j_{P-1}} * \cdots * \mathbb{G}^1_{i_1,j_1} * \mathbf{F}, \tag{3.60}$$

where $*$ *means the convolution operation with one filter and periodic padding.*

Using Lemma 3.2, the number of filters of $\mathbb{G}^2$ can be reduced to only $(2P-1)^2$ instead of 9^{P-1}. Theorem 3.5 represents the context of Theorem 3.4 extended to convolutions with multiple filters:

Theorem 3.5: *Given a 2D input image* $\mathbf{F} \in \mathbb{R}^{R\text{x}R}$ *and* $\mathbb{G} \in \mathbb{R}^{(2P+1)\text{x}(2P+1)\text{x}m}$, *where* $R > P$ *and* m *represents the number of feature maps, then there exists a series of filters* $\mathbb{G}^1_v \in \mathbb{R}^{3x3xc_{v-1}xc_v}$ *and* $\mathbb{G}^2 \in \mathbb{R}^{3x3x(2v-1)^2xm}$ *for number of feature maps per layer given as* $c_v = (2v+1)^2$ *and layer number* $v = 1, \ldots, P-1$ *such that*

$$\mathbb{G} * \mathbf{F} = \mathbb{G}^2 * \mathbb{G}^1_{P-1} * \cdots * \mathbb{G}^1_1 * \mathbf{F}, \tag{3.61}$$

where $*$ means the convolution operation with multiple filters and periodic padding.

So far, we have proved that large convolutional filters of size $(2P+1 \times 2P+1)$ can be decomposed into a series of filters of size (3×3) for a convolution operation with multiple filters and periodic padding. We will now go on to show the relation between deep ReLu CNNs and ReLu ANNs with one hidden layer, which we will use to prove the UAT for CNNs presented by He et al. (2022). Lemma 3.3 goes on to show that a ReLu ANN with one hidden layer can be represented as a ReLu CNN with one convolutional layer with a large filter:

Lemma 3.3: *For every $\mathbf{F} \in \mathbb{R}^{RxR}$, $\mathbf{A} \in \mathbb{R}^{R^2xm}$ and $\xi_1, \xi_2 \in \mathbb{R}^m$ for any m, there exists a convolutional filter $\mathbb{G}^{(2\lfloor \frac{R}{2} \rfloor+1)x(2\lfloor \frac{R}{2} \rfloor+1)xm}$, bias $\mathbb{B} \in \mathbb{R}^{RxRxm}$, and weight vector $\mathbf{w} \in \mathbb{R}^{mR^2}$ such that*

$$\xi_1 \cdot \sigma(\mathbf{A}\text{vec}(\mathbf{F}) + \xi_2) = \mathbf{w} \cdot \text{vec}(\sigma(\mathbb{G} * \mathbf{F} + \xi_2)). \tag{3.62}$$

Now we will go on to present Lemma 3.4, which shows that a deep ReLu CNN using multiple size (3×3) filters can represent a ReLu CNN with one convolutional layer using a large filter.

Lemma 3.4: *Given that Δ is a bounded set in $\mathbb{R}^{RxR}$ with $\mathbf{F} \in \Delta$ and that we have a filter $\mathbb{G} \in \mathbb{R}^{(2\lfloor \frac{R}{2} \rfloor+1)x(2\lfloor \frac{R}{2} \rfloor+1)xm}$ and bias $\mathbb{B}^{RxRxm}$, then there is a series of filters $\mathbb{G}_v \in \mathbb{R}^{3x3xc_{v-1}xc_v}$ and biases $\mathbb{B}_v \in \mathbb{R}^{RxRxc_v}$ with number of feature maps per layer given as $c_v = (2v+1)^2$, $v = 1, ..., \lfloor \frac{R}{2} \rfloor - 1$ and $c_{\lfloor \frac{R}{2} \rfloor} = m$ such that*

$$[\mathbb{G} * \mathbf{F} + \mathbb{B}]_{\lceil \frac{R}{2} \rceil, \lceil \frac{R}{2} \rceil, m} = [\mathbb{G}_{\lfloor \frac{R}{2} \rfloor} * f_{\lfloor \frac{R}{2} \rfloor - 1}(\mathbf{F}) + \mathbb{B}_{\lfloor \frac{R}{2} \rfloor}]_{\lceil \frac{R}{2} \rceil, \lceil \frac{R}{2} \rceil, m}, \tag{3.63}$$

for any m where $f_0(\mathbf{F}) = \mathbf{F}$ and

$$f_v(\mathbf{F}) = \sigma(\mathbb{G}_v * f_{v-1}(\mathbf{F}) + \mathbb{B}_v). \tag{3.64}$$

To finalize their proof, He et al. (2022) make use of Theorem 3.6 which is an approximation result presented by Bach (2014) and Siegel and Xu (2021) for ReLu ANNs with one hidden layer. They consider the class of shallow ANNs with a respective activation function and its approximation using nonlinear *dictionary* approximation (Siegel and Xu, 2021). Given that there is a Banach space $\mathcal{Z}$, then a dictionary $\mathcal{Y}$ is said to be a bounded subset of this Banach space. Nonlinear dictionary approximation involves the approximation of a function $f \in \mathcal{Z}$ using 'm elements of $\mathcal{Y}$ with some algorithm $f \mapsto f_m$ at some rate' (Gribonval and Nielsen, 2004) like for example

$$\|f - f_m\|_{\mathcal{Z}} = \mathcal{O}(m^{-1}). \tag{3.65}$$

We briefly define some terms concerned with nonlinear dictionary approximation in Theorem 3.6; however, we will not be going into more detail on them and refer to Siegel and Xu (2022) for more detail.

Theorem 3.6: *Given that $f : \Delta \subset \mathbb{R}^R \mapsto \mathbb{R}$ and that Δ is a bounded set, then there exists a ReLu ANN with one hidden layer:*

$$f_m(x) = \xi_1 \cdot \sigma(\mathbf{A}x + \xi_2), \tag{3.66}$$

where $\mathbf{A} \in \mathbb{R}^{m \times R}$ and $\xi_1, \xi_2 \in \mathbb{R}^m$ for any m such that

$$\|f - f_m\|_{L^2(\Delta)} \lessapprox m^{-\frac{1}{2}-\frac{3}{2R}} \|f\|_{\mathcal{K}(\mathbb{D})}. \tag{3.67}$$

Here, the symbol $\lessapprox$ means that for $a \lessapprox b$, there is a C that depends only on dimension R and domain Δ such that $a \leq Cb$. The dictionary created by the ReLu activation function σ as defined in He et al. (2022) is given as

$$\mathcal{Y} = \{\sigma(\mathbf{w} \cdot x + b) \ : \ \mathbf{w} \in \mathbb{R}^R, b \in \mathbb{R}\}. \tag{3.68}$$

Consider the (nonlinear) set of all linear combinations of at most m elements from $\mathcal{Y}$ (Gribonval and Nielsen, 2004):

$$\sum_m \mathcal{Y} = \{\sum_{j=1}^{n} a_j h_j \ : \ m \in \mathbb{N}, h_j \in \mathcal{Y}\}. \tag{3.69}$$

We define $B_1(\mathcal{Y})$ to be the closure of the convex, symmetric hull of $\mathcal{Y}$ (Siegel and Xu, 2022) and is given by

$$B_1(\mathcal{Y}) = \overline{\{\sum_{j=1}^{n} a_j h_j \ : \ n \in \mathbb{N}, h_j \in \mathcal{Y}, \sum_{j=1}^{n} |a_j| \leq 1\}}. \tag{3.70}$$

Finally, we define $\|f\|_{\mathcal{K}(\mathcal{Y})}$ to be the norm defined by the *gauge* of $B_1(\mathcal{Y})$ as mentioned in He et al. (2022), i.e.,

$$\|f\|_{\mathcal{K}(\mathcal{Y})} = \inf\{c > 0 \ : \ f \in cB_1(\mathcal{Y})\}, \tag{3.71}$$

and $\mathcal{K}(\mathcal{Y})$ is defined to be a subspace of Δ known as the *variation norm* denoted as

$$\mathcal{K}(\mathcal{Y}) := \{f \in \Delta \ : \ \|f\|_{\mathcal{K}(\mathcal{Y})} < \infty\}. \tag{3.72}$$

Combining Lemma 3.3, Lemma 3.4 and Theorem 3.6, we get the UAT for deep CNNs having convolutional layers using multiple filters of size (3×3):

Theorem 3.7: *Given that $f : \Delta \subset \mathbb{R}^R \mapsto \mathbb{R}$ and that Δ is a bounded set with $\|f\|_{\mathcal{K}(\mathcal{Y})} < \infty$, then there exists a CNN function $f_m : \mathbb{R}^R \mapsto \mathbb{R}$ with size (3×3) filters for number of feature maps per layer $c_v = (2v+1)^2$ and layer number $v = 1, ..., \lfloor \frac{R}{2} \rfloor - 1$, where $c_{\lfloor \frac{R}{2} \rfloor} = m$, such that*

$$\|f - f_m\|_{L^2(\Delta)} \lessapprox m^{-\frac{1}{2}-\frac{3}{2R^2}} \|f\|_{\mathcal{K}(\mathbb{D})}. \tag{3.73}$$

Theorem 3.7 indicates that deep CNNs under certain assumptions are at least as good in terms of approximation ability as ANNs with one hidden layer for image classification (He et al., 2022). For Theorem 3.7 to hold, it is necessary for the CNN depth to be at least as large as $\frac{R}{2}$ and the feature maps for the v^{th} layer to be at least $(2v+1)^2$. Both requirements are not necessarily commonly present in practice; however, several well-known deep CNN models such as ResNet would satisfy these requirements due to the large number of layers in the network and the gradual increase of feature maps outputted layer by layer throughout.

Having reviewed the UAT in the context of CNNs, the discussion now shifts from theoretical capacity to the practical challenge of hyperparameter optimization.

3.7: Hyperparameter Optimization

Setting an optimal set of hyperparameters in deep networks remains something of a black art in practice, often requiring prior experience and extensive trial and error to design an effective architecture. The objective of hyperparameter optimization is to identify sets of values that optimize the learning algorithm. Common methods include manual search, grid search, and random search.

Manual search is the most basic method, where values are initially chosen based on intuition and improved with trial and error. This becomes problematic when tuning multiple hyperparameters simultaneously, as the mutual influence between them can be complex (Kim and Kang, 2020).

Grid search is widely used due to its simplicity. It requires defining a set of values for each hyperparameter, then performs an exhaustive search across all possible combinations. However, the number of combinations grows exponentially with more hyperparameters, making it computationally expensive (Bellman, 1961).

Random search provides an alternative when grid search becomes infeasible. Instead of checking all combinations, it randomly samples hyperparameter sets until a satisfactory result is reached. Bergstra and Bengi (2012) note that random search is efficient while maintaining the reproducibility of grid search. It often performs better when some hyperparameters are more influential (Kim and Kang, 2020). However, it does not guarantee the best combination, whereas grid search might—albeit at a higher computational cost.

Both grid and random search serve as solid baselines, but a hybrid approach combining manual tuning with either method is commonly used and effective in practice (Larochelle et al. (2007), Hinton (2012)). Having examined approaches to hyperparame-

ter optimization, we now turn to the performance metrics that provide a basis for evaluating CNN models.

3.8: Performance Metrics

In this section, we will be going over several performance metrics, namely the *accuracy*, *recall*, *precision*, and *F1-measure*.

The accuracy metric is one of the simplest metrics used in practice, where it calculates the ratio of correct predictions over the total number of predictions by the model. The accuracy can be calculated as follows

$$\text{Acc} = \frac{\sum_{r=1}^{L} TP_r}{\sum_{r=1}^{L} TP_r + \sum_{r=1}^{L} FP_r + \sum_{r=1}^{L} FN_r}, \tag{3.74}$$

where TP_r represents the number of elements that were correctly predicted for the r^{th} label, FP_r represents the number of elements that were predicted for the r^{th} label but were not actually from the r^{th} label and FN_r represents the number of elements that were predicted for labels other than the r^{th} label but were actually from the r^{th} label. Since the sum of TP_r is taken in the numerator, there is no label-specific information being preserved. This means that, in the case of an imbalanced distribution of elements among the labels, accuracy could produce inflated values due to the majority labels outnumbering the minority labels (Ranawana and Palade, 2006).

The precision metric computes the number of correctly predicted elements for a label over the total number of predicted elements for a label, meaning that it measures the ability of the model to predict elements for that label. For the r^{th} label, the precision Pre_r can be calculated as follows

$$\text{Pre}_r = \frac{TP_r}{TP_r + FP_r}. \tag{3.75}$$

The recall metric computes the number of correctly predicted elements for a label over the total number of actual elements for a label, meaning that it measures the number of elements from a label that were predicted correctly by the model. For the r^{th} label, the recall Rec_r can be calculated as follows

$$\text{Rec}_r = \frac{TP_r}{TP_r + FN_r}. \tag{3.76}$$

Both the precision and recall metrics are useful in different situations. For example, if the goal is to avoid false predictions for a specific label, then optimizing precision would be more important. On the other hand, if the goal is to avoid misclassifying elements of a specific label, then optimizing recall would be key.

In situations where we need to compare two models with differing precision and recall values, we can utilize the F1-measure, which is defined to be the harmonic mean of both precision and recall. The F1-measure is seen as a special case of $F\beta - measure$, which uses a positive parameter β to weigh more importance to recall over precision or vice-versa. For the r^{th} label, the Fβ-measure can be calculated as follows

$$\mathrm{F}\beta_r = (1 + \beta^2)\frac{\mathrm{Pre}_r * \mathrm{Rec}_r}{\beta^2\mathrm{Pre}_r + \mathrm{Rec}_r}. \tag{3.77}$$

When $\beta < 1$, precision is emphasized over recall, whereas when $\beta > 1$, recall is prioritized over precision. Given that $\beta = 1$, then we would have the F1-measure.

In the case of multi-label classification, the recall, precision and F1-measure of each label can be aggregated to get a more general result by taking the *macro-average*. A macro-average will compute the performance metric independently for each r^{th} label and then take the average, which means that each label is treated equally. In the case of macro-averaging for L labels, the recall, precision and F1-measure would be first computed for each of the r^{th} labels, where the macro-averaged recall, precision and F1-measure are taken to be the average of the L groups. For example, the macro-averaged F1-measure would be calculated as follows:

$$\mathrm{F1}_{Macro} = \frac{\sum_{r=1}^{L}\mathrm{F1}_r}{L}. \tag{3.78}$$

4

Tensor Regression

In Chapter 3, we've seen how the use of filters in convolutional and pooling layers helps to capture local features in images, where, through the property of weight sharing, the CNN can detect features in any region of the image. After many stacked layers in a CNN, the result would generally be flattened and passed through fully connected layers or go through a global average pooling operator. Either way, this results in a vast number of parameters raising challenges related to model parsimony and overfitting, especially with limited data. On the other hand, Tensor Regression (TR) allows for more interpretable models, where, with the help of tensor decomposition algorithms, the reduced dimensionality makes it an option for use with limited data, unlike CNNs. In this chapter, we will cover TR and tensor decomposition, where we will then go into multinomial TR, which we will use for our application.

First, we will briefly introduce the material that we will cover in Section 4.1. Section 4.2 will go into general tensor algebra. From there, Sections 4.3 and 4.4 will go into tensor decomposition algorithms and TR models as well as an algorithm for Maximum Likelihood Estimation (MLE) of said models. Finally, Section 4.5 discusses regularization in TR.

4.1: Introduction

In Chapter 3, we showed that tensors are a natural representation for image data and that CNNs exploit this structure effectively. However, most classical statistical models, such as ordinary linear or logistic regression, are formulated for vector inputs and therefore require high-dimensional tensors to be vectorized before application. Vectorization concatenates tensor entries into a single long vector in an essentially arbitrary order (Guo et al., 2012), which creates two major problems. The first issue is that the underly-

ing structural information that there would be between different dimensions would be disregarded, effectively making it harder to exploit the neighbourhood relationship between entries of the data and losing interpretability (Tan et al., 2013). The second issue is that the vectorization of the tensor could create a very high-dimensional vector which, due to the curse of dimensionality, could lead to overfitting. In this context, the term high-dimensionality means that each element of a tensor is considered as a predictor.

To mitigate these issues, researchers have long used matrix factorization and decomposition algorithms for dimensionality reduction and numerically efficient representations (Rabanser et al. (2017), Lu (2022)). A popular matrix decomposition algorithm is *Singular Value Decomposition* (SVD), which expresses a matrix as the product of two orthogonal matrices and a diagonal matrix of singular values (Guruswami and Kannan, 2012). Tensor decomposition algorithms generalize these ideas to tensors, i.e., projecting raw tensor data to a lower-dimensional space while seeking to preserve essential information (Lu et al. (2008) Kolda and Bader (2009), Makantasis et al. (2018)). Common tensor decompositions include *Tucker* and *Candecomp/Parafac* (CP), both of which can be viewed as higher-order generalizations of SVD (Harshman, 1970; Kolda, 2006; Tucker, 1966).

These decompositions are central to TR approaches, which model outcomes directly from tensor-valued predictors without full vectorization. TR methods have been developed for linear and logistic settings (Guo et al. (2012), Zhou et al. (2013), Yu and Liu (2016), Tan et al. (2013), Makantasis et al. (2017)) and are used specifically to reduce dimensionality and preserve structure in predictive tasks. For classification, Tan et al. (2013) introduced a logistic TR model that extends classical logistic regression to tensor inputs. Others have extended this framework to multinomial outcomes for practical applications (Cao et al., 2022).

In this chapter, we develop the tensor algebra and notation needed to present these models and then describe TR in detail, including the multinomial TR model used for our application. Before this, however, we will need to go into some detail on tensor algebra in general and our notation for it in the coming section.

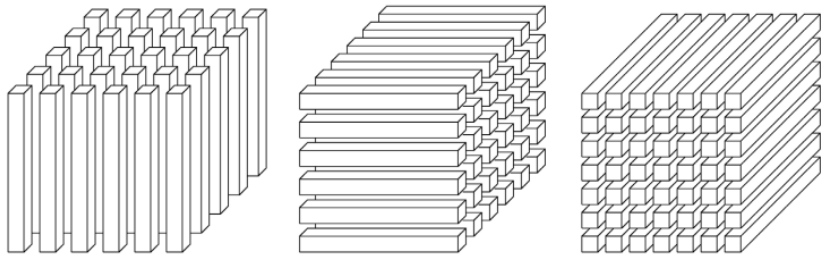
4.2: Tensor Notation and Algebra

In this section, we will be covering some fundamental material on tensor algebra, including several matrix and tensor operations, which will be important in other sections of this chapter, particularly Sections 4.3 and 4.4.

4.2.1: General Concepts and Notions related to Tensors

We will continue to use the definition of a tensor as a multidimensional array of order O from Chapter 3. However, the element-wise notation for tensors in this chapter will be slightly different from that of Chapter 3, where we were using indices for the respective feature map and layer as well. Also, when we mention that a tensor $\mathbb{X}$ is of order O, we are assuming that $\mathbb{X} \in \mathbb{R}^{R_1 \times \cdots \times R_O}$, where $R_1, R_2, ..., R_O \in \mathbb{N}$. We will be denoting the entry r_1 of a vector $\mathbf{x} \in \mathbb{R}^{R_1}$ as x_{r_1} and the element (r_1, r_2) of a matrix $\mathbf{X} \in \mathbb{R}^{R_1 \times R_2}$ as $x_{r_1 r_2}$. Similarly, the element (r_1, r_2, r_3) of a tensor $\mathbb{X} \in \mathbb{R}^{R_1 \times R_2 \times R_3}$ is denoted by $x_{r_1 r_2 r_3}$, where $r_1 = 1, ..., R_1$, $r_2 = 1, ..., R_2$, and $r_3 = 1, ..., R_3$.

A *fiber* is defined to be a higher-order portrayal of matrix rows and columns for a tensor, where every index of the tensor but one would be fixed. An order O tensor has up to *mode-O* fibers. For example, the columns of a matrix would be mode-1 fibers and the rows as mode-2 fibers. Similarly, an order 3 tensor would have mode-1, mode-2 and mode-3 fibers, which will be denoted as $\mathbf{x}_{:r_2 r_3}$, $\mathbf{x}_{r_1 : r_3}$ and $\mathbf{x}_{r_1 r_2 :}$. Fibers would always be represented as column vectors. Figure 4.1 gives a visual illustration of the mode-1, mode-2 and mode-3 fibers of an order 3 tensor.



(a) Mode-1 (column) fibers (b) Mode-2 (row) fibers (c) Mode-3 (tube) fibers

Figure 4.1: Mode-1, mode-2 and mode-3 fibers of an order 3 tensor (Kolda and Bader (2009)).

A *slice* can be described as a 2D cut of a tensor taken by fixing all but two of the indices of the tensor returning a matrix. A matrix would have only itself as a slice. An order 3 tensor would have three possible types of slices, where these are referred to as *horizontal*, *lateral* and *frontal* slices which fix the first, second and third indices of each entry, respectively. These will be denoted as $\mathbb{X}_{r_1 ::}$, $\mathbb{X}_{:r_2 :}$ and $\mathbb{X}_{::r_3}$, respectively. Figure 4.2 gives a visual illustration of horizontal, lateral and frontal slices.

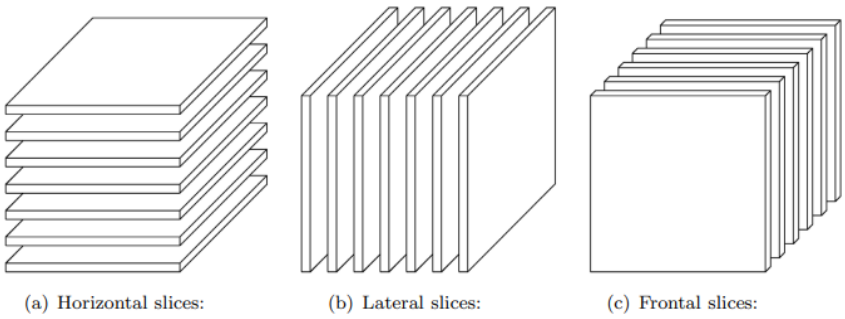


Figure 4.2: The slices of an order 3 tensor. Taken from Kolda and Bader (2009).

For example, let us say that we have the following order 3 tensor $\mathbb{X} \in \mathbb{R}^{4\times3\times2}$

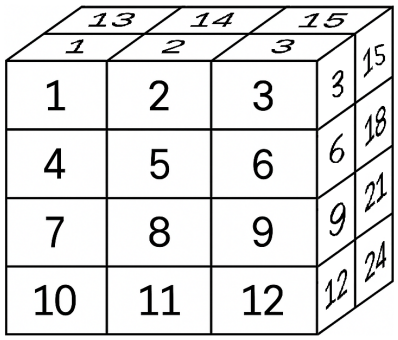


Figure 4.3: Example tensor $\mathbb{X}$.

Its frontal slices are given by

$$\mathbb{X}_{::1} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}, \quad \mathbb{X}_{::2} = \begin{bmatrix} 13 & 14 & 15 \\ 16 & 17 & 18 \\ 19 & 20 & 21 \\ 22 & 23 & 24 \end{bmatrix},$$

where $\mathbb{X}_{::1}$ and $\mathbb{X}_{::2}$ represent the front and back frontal slices of $\mathbb{X}$, respectively. In terms of horizontal and lateral slices, $\mathbb{X}$ would respectively be written as

$$\mathbb{X}_{1::} = \begin{bmatrix} 1 & 2 & 3 \\ 13 & 14 & 15 \end{bmatrix}, \quad \mathbb{X}_{2::} = \begin{bmatrix} 4 & 5 & 6 \\ 16 & 17 & 18 \end{bmatrix},$$

$$\mathbb{X}_{3::} = \begin{bmatrix} 7 & 8 & 9 \\ 19 & 20 & 21 \end{bmatrix}, \quad \mathbb{X}_{4::} = \begin{bmatrix} 10 & 11 & 12 \\ 22 & 23 & 24 \end{bmatrix},$$

where $\mathbb{X}_{1::}, \mathbb{X}_{2::}, \mathbb{X}_{3::}$ and $\mathbb{X}_{4::}$ represent the four horizontal slices of $\mathbb{X}$ going from top to bottom, and

$$\mathbb{X}_{:1:} = \begin{bmatrix} 1 & 13 \\ 4 & 16 \\ 7 & 19 \\ 10 & 22 \end{bmatrix}, \quad \mathbb{X}_{:2:} = \begin{bmatrix} 2 & 14 \\ 5 & 17 \\ 8 & 20 \\ 11 & 23 \end{bmatrix}, \quad \mathbb{X}_{:3:} = \begin{bmatrix} 3 & 15 \\ 6 & 18 \\ 9 & 21 \\ 12 & 24 \end{bmatrix},$$

where $\mathbb{X}_{:1:}, \mathbb{X}_{:2:}$ and $\mathbb{X}_{:3:}$ represent the three lateral slices of $\mathbb{X}$ going from left to right.

In terms of fibers for our example, the mode-1 fibers of $\mathbb{X}$ are

$$\mathbf{x}_{:11} = \begin{bmatrix} 1 \\ 4 \\ 7 \\ 10 \end{bmatrix}, \quad \mathbf{x}_{:21} = \begin{bmatrix} 2 \\ 5 \\ 8 \\ 11 \end{bmatrix}, \quad \mathbf{x}_{:31} = \begin{bmatrix} 3 \\ 6 \\ 9 \\ 12 \end{bmatrix},$$

$$\mathbf{x}_{:12} = \begin{bmatrix} 13 \\ 16 \\ 19 \\ 22 \end{bmatrix}, \quad \mathbf{x}_{:22} = \begin{bmatrix} 14 \\ 17 \\ 20 \\ 23 \end{bmatrix}, \quad \mathbf{x}_{:32} = \begin{bmatrix} 15 \\ 18 \\ 21 \\ 24 \end{bmatrix}.$$

The mode-2 fibers of $\mathbb{X}$ are

$$\mathbf{x}_{1:1} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \quad \mathbf{x}_{2:1} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}, \quad \mathbf{x}_{3:1} = \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix}, \quad \mathbf{x}_{4:1} = \begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix},$$

$$\mathbf{x}_{1:2} = \begin{bmatrix} 13 \\ 14 \\ 15 \end{bmatrix}, \quad \mathbf{x}_{2:2} = \begin{bmatrix} 16 \\ 17 \\ 18 \end{bmatrix}, \quad \mathbf{x}_{3:2} = \begin{bmatrix} 19 \\ 20 \\ 21 \end{bmatrix}, \quad \mathbf{x}_{4:2} = \begin{bmatrix} 22 \\ 23 \\ 24 \end{bmatrix}.$$

Finally, the mode-3 fibers of $\mathbb{X}$ are

$$\mathbf{x}_{11:} = \begin{bmatrix} 1 \\ 13 \end{bmatrix}, \quad \mathbf{x}_{21:} = \begin{bmatrix} 4 \\ 16 \end{bmatrix}, \quad \mathbf{x}_{31:} = \begin{bmatrix} 7 \\ 19 \end{bmatrix}, \quad \mathbf{x}_{41:} = \begin{bmatrix} 10 \\ 22 \end{bmatrix},$$

$$\mathbf{x}_{12:} = \begin{bmatrix} 2 \\ 14 \end{bmatrix}, \quad \mathbf{x}_{22:} = \begin{bmatrix} 5 \\ 17 \end{bmatrix}, \quad \mathbf{x}_{32:} = \begin{bmatrix} 8 \\ 20 \end{bmatrix}, \quad \mathbf{x}_{42:} = \begin{bmatrix} 11 \\ 23 \end{bmatrix},$$

$$\mathbf{x}_{13:} = \begin{bmatrix} 3 \\ 15 \end{bmatrix}, \quad \mathbf{x}_{23:} = \begin{bmatrix} 6 \\ 18 \end{bmatrix}, \quad \mathbf{x}_{33:} = \begin{bmatrix} 9 \\ 21 \end{bmatrix}, \quad \mathbf{x}_{43:} = \begin{bmatrix} 12 \\ 24 \end{bmatrix}.$$

Let $\mathbb{X}$ and $\mathbb{Y}$ represent order O tensors of similar size. The *inner product* of two tensors $\mathbb{X}$ and $\mathbb{Y}$ is calculated by the sum of the products of their entries. Vectorization

transforms a tensor into a column vector by vertically stacking the columns of the tensor in a specific way to form a tall vector. We will be going into more detail on tensor vectorization in Section 4.2.3. The inner product of $\mathbb{X}$ and $\mathbb{Y}$ is given by

$$\langle \mathbb{X}, \mathbb{Y} \rangle = \text{vec}(\mathbb{X})^{\mathsf{T}} \text{vec}(\mathbb{Y}) = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \cdots \sum_{r_O=1}^{R_O} x_{r_1 r_2 \ldots r_O} y_{r_1 r_2 \ldots r_O}. \tag{4.1}$$

Additionally, the *Frobenius norm* of a tensor $\mathbb{X}$ is given by

$$\|\mathbb{X}\| = \sqrt{\langle \mathbb{X}, \mathbb{X} \rangle} = \sqrt{\sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \cdots \sum_{r_O=1}^{R_O} |x_{r_1 r_2 \ldots r_O}|^2}. \tag{4.2}$$

4.2.2: Matrix Products

We will be briefly going into the definitions of a few products between matrices, notably the *matrix product*, *Kronecker product*, and the *Khatri-Rao product*. Let $\mathbf{B} \in \mathbb{R}^{C_1 \times C_2}$, $\mathbf{D} \in \mathbb{R}^{C_1 \times C_2}$, $\mathbf{E} \in \mathbb{R}^{L \times K}$, $\mathbf{F} \in \mathbb{R}^{C_1 \times L}$ and $\mathbf{G} \in \mathbb{R}^{C_2 \times L}$ represent five matrices, where $C_1, C_2, L, K \in \mathbb{N}$. The matrix product will be generally useful throughout, like in Section 4.2.5, where we discuss the N-mode product and mode-N matricization. The Kronecker product will be useful later on in Section 4.4.3 for likelihood-based inference and is also used in Tucker decomposition in Appendix B. It also relates to the Khatri-Rao product, which will be used in CP decomposition.

The matrix product of two matrices $\mathbf{B}$ and $\mathbf{G}$ returns a matrix $\mathbf{BG} \in \mathbb{R}^{C_1 \times L}$, where the element (r_i, r_j) of $\mathbf{BG}$ is the result of the scalar product of the i^{th} row of $\mathbf{B}$ and the j^{th} column of $\mathbf{G}$.

The Kronecker product of two matrices $\mathbf{D}$ and $\mathbf{E}$, which is represented by the $\otimes$ operator, is calculated as follows

$$\mathbf{D} \otimes \mathbf{E} = \begin{bmatrix} d_{11}\mathbf{E} & d_{12}\mathbf{E} & \cdots & d_{1C_2}\mathbf{E} \\ d_{21}\mathbf{E} & d_{22}\mathbf{E} & \cdots & d_{2C_2}\mathbf{E} \\ \vdots & \vdots & \ddots & \vdots \\ d_{C_1 1}\mathbf{E} & d_{C_1 2}\mathbf{E} & \cdots & d_{C_1 C_2}\mathbf{E} \end{bmatrix}, \tag{4.3}$$

where $(\mathbf{D} \otimes \mathbf{E}) \in \mathbb{R}^{C_1 L \times C_2 K}$.

The Khatri-Rao product of two matrices $\mathbf{F}$ and $\mathbf{G}$, which is represented by the $\odot$ operator, is defined as the "matching columnwise" Kronecker product (Kolda and Bader,

2009), and is calculated as follows

$$\mathbf{F} \odot \mathbf{G} = \begin{bmatrix} f_{11}\mathbf{g}_1 & f_{12}\mathbf{g}_2 & \cdots & f_{1L}\mathbf{g}_L \\ f_{21}\mathbf{g}_1 & f_{22}\mathbf{g}_2 & \cdots & f_{2L}\mathbf{g}_L \\ \vdots & \vdots & \ddots & \vdots \\ f_{C_1 1}\mathbf{g}_1 & f_{C_1 2}\mathbf{g}_2 & \cdots & f_{C_1 L}\mathbf{g}_L \end{bmatrix}, \tag{4.4}$$

where $\mathbf{F} \odot \mathbf{G} \in \mathbb{R}^{C_1 C_2 \times L}$ and $\mathbf{g}_1$, ..., $\mathbf{g}_L$ are the columns of $\mathbf{G}$. In the case that both $\mathbf{F}$ and $\mathbf{G}$ are vectors, then their Khatri-Rao product would be identical to their Kronecker product.

We have defined some properties of the Kronecker and Khatri-Rao products in Appendix B, which are used later on in the text and in the appendix. For a more general overview of these products and their properties, we refer to Loan (2000) and Smilde et al. (2004).

4.2.3: Matricization

Matricization refers to converting a tensor into a matrix by systematically rearranging its elements. In general, for an order O tensor $\mathbb{X}$, there would be two ordered sets $\mathcal{R} := \{\alpha_1, ..., \alpha_l\}$ and $\mathcal{C} := \{\gamma_1, ..., \gamma_m\}$ that partition the modes $1, ..., O$ of $\mathbb{X}$, where $l + m = O$. Matricization would have the indices of these two sets $\mathcal{R}$ and $\mathcal{C}$ mapped to the rows and columns of the matricized tensor $\mathbb{X}_{(\mathcal{R}\times\mathcal{C})}$, respectively (Kolda, 2006). This can be specified as

$$\mathbb{X}_{(\mathcal{R}\times\mathcal{C})} \in \mathbb{R}^{J\times K} \text{ with } J = \prod_{n\in\mathcal{R}} R_n \text{ and } K = \prod_{n\in\mathcal{C}} R_n. \tag{4.5}$$

Alternatively, $\mathbb{X}_{(\mathcal{R}\times\mathcal{C})}$ can be expressed differently. To give an example, if $\mathcal{R} := \{1, 2, 3\}$ and $\mathcal{C} := \{4, ..., O\}$, then we can express $\mathbb{X}_{(\mathcal{R}\times\mathcal{C})}$ as $\mathbb{X}_{(R_1 R_2 R_3 \times R_4 \cdots R_O)} \in \mathbb{R}^{R_1 R_2 R_3 \times R_4 \cdots R_O}$.

A special case of matricization is *mode-N matricization*, where, given an order O tensor $\mathbb{X}$ such that $1 \leq N \leq O$, we would have the set $\mathcal{R}$ consist of only the N^{th} mode (or dimension) with the set $\mathcal{C}$ containing the remaining $O - 1$ modes. This is analogous to the mode-N fibers of a tensor being arranged as the columns of the resultant matrix. The mode-N matricization of $\mathbb{X}$, which is denoted as $\mathbb{X}_{(N)} \in \mathbb{R}^{R_N \times (R_1 R_2 \cdots R_{N-1} R_{N+1} \cdots R_O)}$, maps the element $(r_1, r_2, ..., r_O)$ of $\mathbb{X}$ to element (r_N, y) of $\mathbb{X}_{(N)}$ given that

$$y = 1 + \sum_{k=1, k\neq N}^{O} (r_k - 1) J_k, \tag{4.6}$$

and

$$J_k = \prod_{m=1, m\neq N}^{k-1} R_m, \tag{4.7}$$

where $J_1 = 1$. Figure 4.4 represents a visual illustration of the mode-1 matricization of an order 3 tensor $\mathbb{X}$.

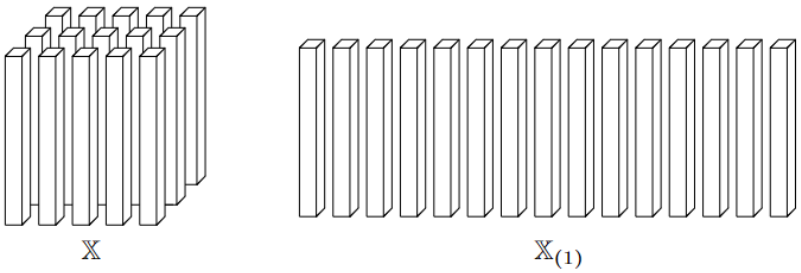


Figure 4.4: Figure depicting the mode-1 fibers of an order 3 tensor $\mathbb{X}$ and its mode-1 matricization. Taken from Kolda (2006).

Given the order 3 tensor $\mathbb{X} \in \mathbb{R}^{4\times3\times2}$ from before, its mode-N matricization at $N = 1, 2,$ and 3 are

$$\mathbb{X}_{(1)} = \begin{bmatrix} 1 & 2 & 3 & 13 & 14 & 15 \\ 4 & 5 & 6 & 16 & 17 & 18 \\ 7 & 8 & 9 & 19 & 20 & 21 \\ 10 & 11 & 12 & 22 & 23 & 24 \end{bmatrix} \in \mathbb{R}^{4\times6},$$

$$\mathbb{X}_{(2)} = \begin{bmatrix} 1 & 4 & 7 & 10 & 13 & 16 & 19 & 22 \\ 2 & 5 & 8 & 11 & 14 & 17 & 20 & 23 \\ 3 & 6 & 9 & 12 & 15 & 18 & 21 & 24 \end{bmatrix} \in \mathbb{R}^{3\times8}, \tag{4.8}$$

$$\mathbb{X}_{(3)} = \begin{bmatrix} 1 & 4 & 7 & 10 & 2 & \cdots & 9 & 12 \\ 13 & 16 & 19 & 22 & 14 & \cdots & 21 & 24 \end{bmatrix} \in \mathbb{R}^{2\times12},$$

respectively.

Vectorization is a special case of matricization, where a tensor is transformed into a vector. In this case, given an order O tensor $\mathbb{X}$, the set $\mathcal{R}$ would contain all O modes whereas the set $\mathcal{C}$ would be empty. The element $(r_1, ..., r_O)$ of $\mathbb{X}$ is mapped to the y^{th} element of vec($\mathbb{X}$) given that

$$y = 1 + \sum_{N=1}^{O} (r_N - 1) J_N, \tag{4.9}$$

and

$$J_N = \prod_{m=1}^{N-1} R_m, \tag{4.10}$$

where $J_1 = 1$. Following our earlier example, vectorizing the order 3 tensor $\mathbb{X} \in \mathbb{R}^{4\times3\times2}$ using equations (4.6) and (4.10) would yield

$$\text{vec}(\mathbb{X}) = \begin{bmatrix} 1 \\ 4 \\ 7 \\ 10 \\ 2 \\ \vdots \\ 21 \\ 24 \end{bmatrix} \in \mathbb{R}^{24}. \tag{4.11}$$

The vectorization of $\mathbb{X}_{(1)}$ would be identical to the vectorization of $\mathbb{X}$. Using our earlier example, vectorizing $\mathbb{X}_{(1)}$ given in equation (4.8) would give us the same result as in equation (4.11) meaning that

$$\text{vec}(\mathbb{X}_{(1)}) = \text{vec}(\mathbb{X}). \tag{4.12}$$

4.2.4: Rank-one tensors and N-rank tensors

We recall that the outer product of two vectors $\mathbf{a}_1, \mathbf{a}_2 \in \mathbb{R}^{R_1}$, denoted by $(\mathbf{a}_1 \circ \mathbf{a}_2) \in \mathbb{R}^{R_1\times R_1}$, is calculated as follows

$$\mathbf{a}_1 \circ \mathbf{a}_2 = \mathbf{a}_1\mathbf{a}_2^{\mathrm{T}}. \tag{4.13}$$

Given a tensor, we will be going through a few definitions which have similar names, albeit different meanings. An order O tensor $\mathbb{X}$ is defined to be *rank-one* given that it can be written as the outer product of O vectors, i.e.,

$$\mathbb{X} = \mathbf{a}_1 \circ \mathbf{a}_2 \circ \cdots \circ \mathbf{a}_O, \tag{4.14}$$

where $\mathbf{a}_1 \in \mathbb{R}^{R_1}$, $\mathbf{a}_2 \in \mathbb{R}^{R_2}$, ..., $\mathbf{a}_O \in \mathbb{R}^{R_O}$. On an element-by-element basis, this involves multiplying each pair of corresponding vector entries. So given element $(r_1, r_2, ..., r_O)$ of $\mathbb{X}$, then we are taking the product of the r_1^{th} entry of $\mathbf{a}_1$ denoted as $a_{r_1,1}$, r_2^{th} entry of $\mathbf{a}_2$ denoted as $a_{r_2,2}$, etc, i.e.,

$$x_{r_1\cdots r_O} = a_{r_1,1}a_{r_2,2}\cdots a_{r_O,O}. \tag{4.15}$$

Figure 4.5 gives a visual illustration of a rank-one order 3 tensor $\mathbb{X}$, i.e., $\mathbb{X} = \mathbf{a} \circ \mathbf{b} \circ \mathbf{c}$.

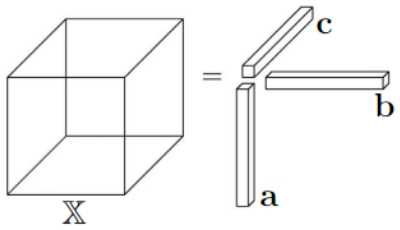


Figure 4.5: An order 3 tensor $\mathbb{X}$ that is rank-one. Taken from Kolda and Bader (2009).

The *rank* of a tensor $\mathbb{X}$, denoted by rank($\mathbb{X}$), is defined to be the minimum number of rank-one tensors that, when added together, reconstruct $\mathbb{X}$ (Tao et al., 2007). The *N-rank* of a tensor $\mathbb{X}$ is a generalization of the rank of a matrix to tensor format. It is denoted by $\text{rank}_N(\mathbb{X})$ and is defined to be the column rank of $\mathbb{X}_{(N)}$, i.e., the dimension of the vector space generated by the mode-N fibers. Given that $F_N = \text{rank}_N(\mathbb{X})$ for $N = 1, ..., O$, then we can say that $\mathbb{X}$ is a rank-$(F_1, F_2, ..., F_O)$ tensor. Trivially by definition, we have that $F_N \leq R_N$ for $N = 1, ..., O$.

4.2.5: N-mode Product

The *N-mode product* of an order O tensor $\mathbb{X}$ with a matrix $\mathbf{K} \in \mathbb{R}^{M \times R_N}$ is defined as the multiplication of $\mathbb{X}$ by $\mathbf{K}$ in mode-N using the notation $\times_N$, where $M \in \mathbb{N}$ and $1 \leq N \leq O$ (Kolda and Bader, 2009). This can be represented as follows

$$\mathbb{U} = \mathbb{X} \times_N \mathbf{K}, \tag{4.16}$$

where $\mathbb{U} \in \mathbb{R}^{R_1 \times \cdots \times R_{N-1} \times M \times R_{N+1} \times \cdots \times R_O}$ is the resulting tensor from the N-mode product. We can see that the N^{th} dimension of $\mathbb{X}$ changes to M in $\mathbb{U}$. Each of the mode-N fibers of $\mathbb{U}$ would be the result of multiplying the corresponding mode-N fiber of $\mathbb{X}$ by matrix $\mathbf{K}$ (Kolda, 2006). A useful property between the N-mode product and mode-N matricization is the following

$$\mathbb{U}_{(N)} = \mathbf{K}\mathbb{X}_{(N)}, \tag{4.17}$$

where $\mathbb{U}_{(N)}$ is the mode-N matricization of $\mathbb{U}$. This means we can first find the mode-N matricization of $\mathbb{X}$ where we would then multiply it by $\mathbf{K}$ giving us $\mathbb{U}_{(N)}$, at which point we can then change $\mathbb{U}_{(N)}$ back to $\mathbb{U}$.

Element-wise for equation (4.16), given element $(r_1, ..., r_{N-1}, m, r_{N+1}, ..., r_O)$ of $\mathbb{U}$ and element (m, r_N) of $\mathbf{K}$ denoted as j_{mr_N}, then we have

$$u_{r_1 \cdots r_{N-1} m r_{N+1} \cdots r_O} = \sum_{r_N=1}^{R_N} x_{r_1 \cdots r_O} j_{mr_N}. \tag{4.18}$$

Given the order 3 tensor $\mathbb{X} \in \mathbb{R}^{4\times3\times2}$ from before as well as the matrix

$$\mathbf{K} = \begin{bmatrix} 3 & 3 & 3 & 3 \\ 6 & 6 & 6 & 6 \\ 9 & 9 & 9 & 9 \end{bmatrix} \in \mathbb{R}^{3\times4},$$

then, as an example, we want to try to find the 1-mode product of $\mathbb{X}$ with $\mathbf{K}$. Inputting the mode-1 matricization of $\mathbb{X}$ from equation (4.8) and $\mathbf{K}$ into equation (4.17), we have

$$\mathbb{U}_{(1)} = \begin{bmatrix} 3 & 3 & 3 & 3 \\ 6 & 6 & 6 & 6 \\ 9 & 9 & 9 & 9 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 13 & 14 & 15 \\ 4 & 5 & 6 & 16 & 17 & 18 \\ 7 & 8 & 9 & 19 & 20 & 21 \\ 10 & 11 & 12 & 22 & 23 & 24 \end{bmatrix},$$

which simplifies to

$$\mathbb{U}_{(1)} = \begin{bmatrix} 66 & 78 & 90 & 210 & 222 & 234 \\ 132 & 156 & 180 & 420 & 444 & 468 \\ 198 & 234 & 270 & 630 & 666 & 702 \end{bmatrix}.$$

We know that $\mathbb{U} \in \mathbb{R}^{3\times3\times2}$ since this is a 1-mode product, where the other dimensions of $\mathbb{X}$ will remain the same with only the first dimension changing. Thus, the frontal slices of $\mathbb{U}$ will be as follows:

$$\mathbb{U}_{::1} = \begin{bmatrix} 66 & 78 & 90 \\ 132 & 156 & 180 \\ 198 & 234 & 270 \end{bmatrix}, \quad \mathbb{U}_{::2} = \begin{bmatrix} 210 & 222 & 234 \\ 420 & 444 & 468 \\ 630 & 666 & 702 \end{bmatrix}.$$

Note that when $\mathbf{K}$ is a vector, say $\mathbf{K}$ is a vector of size R_N, then equations (4.16) and (4.17) can be defined similarly but with the resulting tensor from the n-mode product $\mathbb{U}$ becoming an order $O-1$ tensor, i.e., $\mathbb{U} \in \mathbb{R}^{R_1\times\cdots\times R_{N-1}\times R_{N+1}\times\cdots\times R_O}$.

The product of an order O tensor $\mathbb{X}$ in every mode with $\mathbf{K}_N \in \mathbb{R}^{M_N\times R_N}$ for $N = 1, ..., O$ is defined as follows

$$\mathbb{U} = \mathbb{X} \times_1 \mathbf{K}_1 \times_2 \mathbf{K}_2 \times_3 \cdots \times_O \mathbf{K}_O, \tag{4.19}$$

where $\mathbb{U} \in \mathbb{R}^{M_1\times\cdots\times M_O}$.

There are some additional properties that show relationships between the N-mode product, matricization, and Kronecker products, as well as relationships between the Frobenius norm, inner product and mode-n product, which we presented in Propositions B3 and B5-B7 of Appendix B. They are used mainly for appendix proofs for content that was cut from this dissertation (Section 4.3 mentions this). We refer to Kolda (2006) for more information regarding these properties.

Having established the fundamentals of tensor notation and algebra, we now turn to tensor decomposition in the next section, notably CP decomposition.

4.3: Tensor Decomposition Algorithms

Tensor decomposition algorithms are used to approximate the representation of a high-order tensor using low-rank tensors. There are many tensor decomposition algorithms available, such as *tensor-train* decomposition (Oseledets, 2011) and *tubal rank* decomposition (Kilmer et al., 2013). However, we will focus on CP decomposition in Section 4.3.1, which we will apply in our application. We also covered Tucker decomposition in this dissertation, but due to computational reasons, we could not explore Tucker decomposition in our TR application in Chapter 5. We opted to use CP over Tucker decomposition, since the former has less computational cost and fewer hyperparameters compared to the latter. As such, we did not include the Tucker decomposition material in the dissertation and have presented it in Appendix B.

4.3.1: CP Decomposition

The CP decomposition algorithm seeks to approximate a tensor as a finite sum of rank-one tensors (Kiers (2000), Kolda and Bader (2009)). In the case of an order 3 tensor $\mathbb{X}$, the CP decomposition of $\mathbb{X}$ into Z rank-one tensors is given by

$$\mathbb{X} \approx \sum_{z=1}^{Z} \mathbf{a}_z \circ \mathbf{b}_z \circ \mathbf{c}_z, \tag{4.20}$$

where Z is a positive integer that represents the number of rank-one tensors of the CP decomposition, and the vectors $\mathbf{a}_z \in \mathbb{R}^{R_1}$, $\mathbf{b}_z \in \mathbb{R}^{R_2}$, $\mathbf{c}_z \in \mathbb{R}^{R_3}$ for $z = 1, 2, ..., Z$ are the rank-one components. Element-wise, for element (r_1, r_2, r_3) of $\mathbb{X}$, equation (4.20) is written as

$$x_{r_1 r_2 r_3} \approx \sum_{z=1}^{Z} a_{r_1 z} b_{r_2 z} c_{r_3 z}. \tag{4.21}$$

Figure 4.6 represents a visual illustration of equation (4.20).

Factor matrices are defined to be matrices whose columns consist of the vectors from the rank-one components (Kolda and Bader, 2009). From equation (4.20), we would have the factor matrices $\mathbf{A} = [\mathbf{a}_1, ..., \mathbf{a}_Z] \in \mathbb{R}^{R_1 \times Z}$, $\mathbf{B} = [\mathbf{b}_1, ..., \mathbf{b}_Z] \in \mathbb{R}^{R_2 \times Z}$ and $\mathbf{C} = [\mathbf{c}_1, ..., \mathbf{c}_Z] \in \mathbb{R}^{R_3 \times Z}$, and equation (4.20) can be instead written in matricized form according to the mode-1, mode-2, or mode-3 matricizations of $\mathbb{X}$, respectively, i.e.,

$$\begin{aligned} \mathbb{X}_{(1)} &\approx \mathbf{A}(\mathbf{C} \odot \mathbf{B})^{\mathrm{T}}, \\ \mathbb{X}_{(2)} &\approx \mathbf{B}(\mathbf{C} \odot \mathbf{A})^{\mathrm{T}}, \\ \mathbb{X}_{(3)} &\approx \mathbf{C}(\mathbf{B} \odot \mathbf{A})^{\mathrm{T}}, \end{aligned} \tag{4.22}$$

where $\odot$ is the Khatri-Rao product.

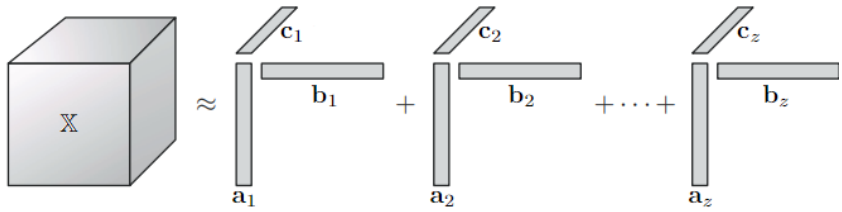


Figure 4.6: Visualizing the CP decomposition of an order 3 tensor $\mathbb{X}$ (Kolda and Bader (2009)).

Equation (4.20) can be alternatively expressed as

$$\mathbb{X} \approx [\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!] = \sum_{z=1}^{Z} \mathbf{a}_z \circ \mathbf{b}_z \circ \mathbf{c}_z, \tag{4.23}$$

which is a representation used by Kruskal (1977).

In general, for an order O tensor $\mathbb{X}$, the CP decomposition of $\mathbb{X}$ into Z rank-one tensors is given by

$$\mathbb{X} \approx [\![\mathbf{A}_1, ..., \mathbf{A}_O]\!] = \sum_{z=1}^{Z} \mathbf{a}_{1,z} \circ \mathbf{a}_{2,z} \circ \cdots \circ \mathbf{a}_{O,z}, \tag{4.24}$$

where element-wise for $\mathbf{a}_{n,z}$, following our notation from equation (4.15), we expand on it by denoting the r_1^{th} entry of $\mathbf{a}_{1,z}$ as a_{1,z,r_1}, r_2^{th} entry of $\mathbf{a}_{1,z}$ as a_{1,z,r_2}, etc. The mode-N matricization for the general case is given by

$$\mathbb{X}_{(N)} \approx \mathbf{A}_N(\mathbf{A}_O \odot \cdots \odot \mathbf{A}_{N+1} \odot \mathbf{A}_{N-1} \odot \cdots \odot \mathbf{A}_1)^{\mathrm{T}}. \tag{4.25}$$

Ideally, we are able to compute a CP decomposition with Z rank-one tensors which best approximates an order O tensor $\mathbb{X}$. We can formalize this decomposition as the following optimization problem:

$$\min_{\hat{\mathbb{X}}} \|\mathbb{X} - \hat{\mathbb{X}}\|, \text{ subject to } \hat{\mathbb{X}} = \sum_{z=1}^{Z} \mathbf{a}_{1,z} \circ \cdots \circ \mathbf{a}_{O,z}, \tag{4.26}$$

with factor matrices $\mathbf{A}_1, ..., \mathbf{A}_O$.

To perform this CP decomposition, there are many different methods available; however, one we will be mentioning is the *Alternating Least Squares* (ALS) method (Faber et al., 2003). The ALS method is a general method for parameter estimation, where it subdivides the global problem into several sub-problems and then solves these sub-problems using least squares (Bro, 1998). As such, the parameters of the global problem would be divided to several subsets of parameters (Takane et al., 1977). At each iteration, ALS updates one subset of parameters by computing its least squares estimate while treating the remaining parameters as fixed, and it repeats this process for every

subset of parameters until all are estimated. The whole process is repeated until convergence or a maximum number of iterations have passed (Takane et al., 1977). The ALS method is user-friendly and often considered the go-to algorithm for CP; however, due to the optimization problem being non-convex, it does not always converge to a global optimum (Kolda and Bader, 2009). CP decomposition of tensors via ALS has also been implemented in software and is easily accessible in Python via the TensorLy package (Kossaifi et al., 2019).

As we shall see in Section 4.4.2, we impose CP decomposition on the parameter tensor in the TR model. In the application, we will not estimate the parameter tensor itself but rather an already decomposed tensor representing it as defined in Section 4.4.2.2. Because of this, we will not be going into detail on how the CP decomposition of a tensor using ALS is computed in this section and refer to Appendix B for more details.

The CP decomposition is said to be *exact* when we have $\text{rank}(\mathbb{X}) = Z$, where this means that there will be equality in equation (4.20). An exact CP decomposition is also referred to as the *rank-Z decomposition*. Ideally, we are able to find $\text{rank}(\mathbb{X}) = Z$; however, the rank of a tensor cannot be easily determined by an algorithm, where the problem is not trivial (Hastad, 1990). In general, the rank of a tensor would not be known in advance and, in noise-free data, it is found using trial and error, where multiple CP decompositions with differing ranks would be done and the one that is 'good' would be chosen. However, in situations where the data is noisy which is frequently the case in practice, the *Core Consistency Diagnostic* (CORCONDIA) metric is used to decide on an appropriate rank (Kolda and Bader, 2009). We refer to Bro and Kiers (2003) for more information on CORCONDIA.

Given the order 3 tensor $\mathbb{X} \in \mathbb{R}^{4\times3\times2}$ from before, we will factorize $\mathbb{X}$ as a sum of rank-one tensors using CP decomposition to show an example, although we will not be looking for the 'best' possible rank in this example. If we were to approximate $\mathbb{X}$ as a sum of $Z = 2$ rank-one tensors using the CP decomposition algorithm, then we would have

$$\mathbb{X} \approx \sum_{z=1}^{2} \mathbf{a}_z \circ \mathbf{b}_z \circ \mathbf{c}_z = \begin{bmatrix} 0.394 \\ 0.466 \\ 0.538 \\ 0.61 \end{bmatrix} \circ \begin{bmatrix} 0.57 \\ 0.599 \\ 0.627 \end{bmatrix} \circ \begin{bmatrix} 36.404 \\ 64.059 \end{bmatrix} + \begin{bmatrix} 1.15 \\ 0.9 \\ 0.651 \\ 0.401 \end{bmatrix} \circ \begin{bmatrix} 2.115 \\ 1.919 \\ 1.723 \end{bmatrix} \circ \begin{bmatrix} -2.987 \\ -0.51 \end{bmatrix}, \quad (4.27)$$

where the factor matrices $\mathbf{A} \in \mathbb{R}^{4\times2}$, $\mathbf{B} \in \mathbb{R}^{3\times2}$ and $\mathbf{C} \in \mathbb{R}^{2\times2}$ are

$$\mathbf{A} = \begin{bmatrix} 0.394 & 1.15 \\ 0.466 & 0.9 \\ 0.538 & 0.651 \\ 0.61 & 0.401 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0.57 & 2.115 \\ 0.599 & 1.919 \\ 0.627 & 1.723 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 36.404 & -2.987 \\ 64.059 & -0.51 \end{bmatrix}.$$

For $Z = 3$, we would instead have

$$\mathbb{X} \approx \sum_{z=1}^{3} \mathbf{a}_z \circ \mathbf{b}_z \circ \mathbf{c}_z, \tag{4.28}$$

where the factor matrices $\mathbf{A} \in \mathbb{R}^{4\times3}$, $\mathbf{B} \in \mathbb{R}^{3\times3}$ and $\mathbf{C} \in \mathbb{R}^{2\times3}$ are

$$\mathbf{A} = \begin{bmatrix} 0.349 & 1.234 & -2.665 \\ 0.442 & 0.847 & -1.575 \\ 0.536 & 0.459 & -0.485 \\ 0.629 & 0.071 & 0.605 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0.564 & 2.444 & 71363.7 \\ 0.597 & 1.687 & 752.67 \\ 0.629 & 0.93 & -69858.8 \end{bmatrix},$$

$$\mathbf{C} = \begin{bmatrix} 29.949 & -2.041 & -0.000007 \\ 61.03 & 0.627 & 0.000005 \end{bmatrix}.$$

Having outlined the fundamentals of CP decomposition, in the next section, we turn to the question of its uniqueness, a property of theoretical importance.

4.3.2: Uniqueness of CP Decomposition

Uniqueness of tensor decompositions refers to the property that a given tensor can be represented in only one distinct manner, up to certain transformations. It is important for multiple reasons, including identifiability and interpretability of results. Identifiability in CP decomposition is necessary to do accurate inference, since otherwise, it can yield multiple sets of factor matrices that represent the same tensor. Similarly, a unique tensor decomposition is interpretable since it allows for a clear interpretation of the components derived from the decomposition.

Given an order 3 tensor $\mathbb{X}$, factor matrices $\mathbf{A} \in \mathbb{R}^{R_1\times Z}$, $\mathbf{B} \in \mathbb{R}^{R_2\times Z}$ and $\mathbf{C} \in \mathbb{R}^{R_3\times Z}$ and that the CP decomposition in equation (4.20) is exact (so we have equality), then we would say that the CP decomposition with Z rank-one tensors given in equation (4.20) is *unique* if it is the only combination of rank-one tensors possible that when summed gives $\mathbb{X}$. This is excluding any scaling and permutation of the factor matrices that generate $\mathbb{X}$, since if we include these, then the original decomposition would not be unique (Kolda

and Bader, 2009). To show this with an example: given that we have any three non-zero scalars α, β, and γ such that $\alpha\beta\gamma = 1$, we would have

$$[\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!] = [\![\alpha\mathbf{A}, \beta\mathbf{B}, \gamma\mathbf{C}]\!], \tag{4.29}$$

where, since the CP decomposition is unchanged by scaling, then the CP decomposition in equation (4.20) would not be unique. Also, given that we have a permutation matrix $\Pi \in \mathbb{R}^{Z\times Z}$, we can use it to reorder the factor matrices as follows

$$[\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!] = [\![\mathbf{A}\Pi, \mathbf{B}\Pi, \mathbf{C}\Pi]\!], \tag{4.30}$$

which shows that the CP decomposition of equation (4.20) is not unique.

A known result for uniqueness is concerned with the n-rank of a tensor, where under some assumptions of the n-rank for the factor matrices, it is a sufficient condition that the CP decomposition is unique (Kruskal (1977), Kruskal (1989)). Kruskals result assumes that we have an order 3 tensor $\mathbb{X}$ and that the n-rank of the factor matrices $\mathbf{A} \in \mathbb{R}^{R_1\times Z}$, $\mathbf{B} \in \mathbb{R}^{R_2\times Z}$ and $\mathbf{C} \in \mathbb{R}^{R_3\times Z}$ are n_A, n_B and n_C. Note that in this case, the n-rank is being applied on the matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ and not the matricization of $\mathbb{X}$ (as was defined in Section 4.2.4). If we have that

$$n_A + n_B + n_C \geq 2Z + 2, \tag{4.31}$$

then we have that $\text{rank}(\mathbb{X}) = Z$ and that it is sufficient that the CP decomposition of $\mathbb{X}$ into Z rank-one tensors is unique. This result was proven by several authors (ten Berge and Sidropoulos (2002), Stegeman and Sidiropoulos (2007)), and it was generalized from an order 3 tensor to an order O tensor by Sidiropoulos and Bro (2000). The generalized result assumes an order O tensor $\mathbb{X}$ and that the n-rank of the factor matrices $\mathbf{A}_1 \in \mathbb{R}^{R_1\times Z}$, $\mathbf{A}_2 \in \mathbb{R}^{R_2\times Z}$, ..., $\mathbf{A}_O \in \mathbb{R}^{R_O\times Z}$ are n_{A_1}, n_{A_2}, ..., n_{A_O}. If we have that

$$\sum_{N=1}^{O} n_{A_N} \geq 2Z + O - 1, \tag{4.32}$$

then it follows similarly as before that $\text{rank}(\mathbb{X}) = Z$ and that it is sufficient that the CP decomposition of $\mathbb{X}$ into Z rank-one tensors is unique. In the case of tensors of exact ranks $Z = 2$ and $Z = 3$, the sufficient condition is also necessary but this does not follow for $Z > 3$ (ten Berge and Sidropoulos, 2002).

Having established the foundations of CP decomposition, we now turn to its application in a TR employed with a GLM framework.

4.4: Generalized Tensor Regression Model

Defining TR under the GLM framework allows one to naturally handle binary, count, and other exponential-family responses through an appropriate link function and distribution. Section 4.4.1 covers TR imposed with CP decomposition under the GLM framework. Parameter estimation then follows using standard likelihood-based estimation, which we will cover in Section 4.4.2. Finally, Section 4.4.3 will then cover the statistical properties of MLE for the TR model defined in Section 4.4.1.

4.4.1: TR Models

In Section 4.4.1, we will first briefly go into GLMs in vector space, where we will then extend them to tensor space and impose CP decomposition on the parameter tensor. From there, we will then go into detail on the multinomial TR model by Cao et al. (2022).

4.4.1.1: CP TR Model

Nelder and Wedderburn (1972) introduced the concept of GLMs and how they extend conventional regression models. In the classical GLM setting by McCullagh and Nelder (1983), for each observation, we would have a random vector of predictors $\mathbf{x} \in \mathbb{R}^{R_1}$ and a random response variable l. The conditional distribution of l given $\mathbf{x}$ is assumed to belong to an exponential family with conditional density or mass function

$$\mathbb{P}\big(l \mid \mathbf{x}; \theta(\mathbf{x}), \phi\big) = \exp\left\{ \frac{l\,\theta(\mathbf{x}) - b\big(\theta(\mathbf{x})\big)}{a(\phi)} + c(l, \phi) \right\}, \tag{4.33}$$

where $\theta(\mathbf{x})$ and ϕ are the natural (canonical) and dispersion parameters, respectively, and $a(\cdot)$, $b(\cdot)$ and $c(\cdot)$ are functions determined by the chosen exponential-family member. The conditional first two moments are

$$\mathbb{E}\big[l \mid \mathbf{x}\big] = \mu(\mathbf{x}) = b'\big(\theta(\mathbf{x})\big), \qquad \mathrm{Var}\big(l \mid \mathbf{x}\big) = b''\big(\theta(\mathbf{x})\big)\, a(\phi).$$

The GLM relates the predictors to the mean $\mu(\mathbf{x}) = \mathbb{E}[l|\mathbf{x}]$ with the linear function of regressors

$$g(\mu(\mathbf{x})) = \eta(\mathbf{x}) = b + \boldsymbol{\beta}^{\mathrm{T}}\mathbf{x}, \tag{4.34}$$

where $g(\cdot)$ is an invertible link function, b is the intercept and $\boldsymbol{\beta} \in \mathbb{R}^{R_1}$ is a weight vector. When we have the canonical parameter being equal to the linear predictor, i.e., $\theta(\mathbf{x}) = \eta(\mathbf{x})$, then the link function is referred to as the canonical link function.

Given a matrix predictor $\mathbf{X} \in \mathbb{R}^{R_1 \times R_2}$ instead of $\mathbf{x}$ and by letting the weight be a rank-one matrix $\mathbf{W} = \boldsymbol{\beta}_1 \circ \boldsymbol{\beta}_2 = \boldsymbol{\beta}_1 \boldsymbol{\beta}_2^{\mathrm{T}}$ with vectorization $\widetilde{\boldsymbol{\beta}} = \mathrm{vec}(\mathbf{W})$, where $\boldsymbol{\beta}_1 \in \mathbb{R}^{R_1}$ and

$\boldsymbol{\beta} \in \mathbb{R}^{R_2}$, we can extend equation (4.34) to matrix format by considering equation (4.34) with the vectorization of $\mathbf{X}$ and $\mathbf{W}$, i.e.,

$$\begin{aligned} g(\mu(\mathbf{X})) &= b + \widetilde{\boldsymbol{\beta}}^{\mathrm{T}} \operatorname{vec}(\mathbf{X}), \\ &= b + \operatorname{vec}(\boldsymbol{\beta}_1 \boldsymbol{\beta}_2^{\mathrm{T}})^{\mathrm{T}} \operatorname{vec}(\mathbf{X}). \end{aligned}$$

By taking the Kronecker identity defined in Proposition B3, we have

$$\begin{aligned} g(\mu(\mathbf{X})) &= b + \operatorname{vec}(\boldsymbol{\beta}_1 \boldsymbol{\beta}_2^{\mathrm{T}})^{\mathrm{T}} \operatorname{vec}(\mathbf{X}), \\ &= b + (\boldsymbol{\beta}_2 \otimes \boldsymbol{\beta}_1)^{\mathrm{T}} \operatorname{vec}(\mathbf{X}), \end{aligned}$$

which is equivalent to

$$g(\mu(\mathbf{X})) = b + \boldsymbol{\beta}_1^{\mathrm{T}} \mathbf{X} \boldsymbol{\beta}_2. \tag{4.35}$$

The bilinear form $\boldsymbol{\beta}_1^{\mathrm{T}} \mathbf{X} \boldsymbol{\beta}_2$ from equation (4.35) is an extension of the linear term $\boldsymbol{\beta}^{\mathrm{T}} \mathbf{x}$ from equation (4.34) (Zhou et al., 2013). Instead of vectorizing $\mathbf{X}$, its matrix form is being preserved in equation (4.35). This extension was proposed by Li et al. (2010) with *dimension folding* for preserving the structure of an array.

In general, given that we have an order O tensor $\mathbb{X}$ as the predictors, then we would extend equation (4.34) from vector to tensor format as follows

$$g(\mu(\mathbb{X})) = b + \langle \mathbb{X}, \mathbb{W} \rangle = b + \operatorname{vec}(\mathbb{X})^{\mathrm{T}} \operatorname{vec}(\mathbb{W}) = b + \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \cdots \sum_{r_O=1}^{R_O} x_{r_1 r_2 \ldots r_O} w_{r_1 r_2 \ldots r_O}, \tag{4.36}$$

where $\mathbb{W}$ is of similar dimensions as $\mathbb{X}$. $\mathbb{W}$ would have a number of parameters $\prod_{N=1}^{O} R_N$ equal to the amount of entries of $\mathbb{X}$ which could be very large. To reduce the number of parameters, $\mathbb{W}$ is usually approximated by representing it to be a sum of Z rank-one tensors using CP decomposition (Guo et al., 2012). From the principle of CP decomposition, it follows that

$$\mathbb{W} = [\![\mathbf{W}_1, \ldots, \mathbf{W}_O]\!] = \sum_{z=1}^{Z} \mathbf{w}_{1,z} \circ \cdots \circ \mathbf{w}_{O,z}, \tag{4.37}$$

where $\mathbf{w}_{1,z} = [w_{1,z,1}, w_{1,z,2}, \ldots, w_{1,z,R_1}]^{\mathrm{T}} \in \mathbb{R}^{R_1}$, ..., $\mathbf{w}_{O,z} \in \mathbb{R}^{R_O}$ for $z = 1, \ldots, Z$ and factor matrices $\mathbf{W}_1 = [\mathbf{w}_{1,1}, \ldots, \mathbf{w}_{1,Z}] \in \mathbb{R}^{R_1 \times Z}$, ..., $\mathbf{W}_O = [\mathbf{w}_{O,1}, \ldots, \mathbf{w}_{O,Z}] \in \mathbb{R}^{R_O \times Z}$. By substituting equation (4.37) to equation (4.36), we will get

$$\begin{aligned} g(\mu(\mathbb{X})) &= b + \langle \sum_{z=1}^{Z} \mathbf{w}_{1,z} \circ \cdots \circ \mathbf{w}_{O,z}, \mathbb{X} \rangle \\ &= b + \sum_{z=1}^{Z} \langle \mathbf{w}_{1,z} \circ \cdots \circ \mathbf{w}_{O,z}, \mathbb{X} \rangle \\ &= b + \sum_{z=1}^{Z} \mathbb{X} \times_1 \mathbf{w}_{1,z} \times_2 \mathbf{w}_{2,z} \times_3 \cdots \times_O \mathbf{w}_{O,z}. \end{aligned} \tag{4.38}$$

CP decomposition reduces the number of parameters to $Z\sum_{N=1}^{O} R_N$. As an example, in the case of regression or classification, vectorizing an RGB image of size (224 x 224 x 3) would return a $150,528$-dimensional vector, which would require a similar-sized weight vector. On the other hand, if we had left the image data in tensor format and used CP decomposition as in equation (4.38), it would be $451Z$-dimensional instead. As such, keeping the image data in tensor format would greatly reduce the number of parameters, which enhances the accuracy of the estimated predictor (Li et al., 2010).

Given a vector of variables $\mathbf{z} \in \mathbb{R}^{R_0}$ as additional covariates, then equation (4.38) can be updated to

$$g(\mu(\mathbb{X}, \mathbf{z})) = b + \zeta^{\mathrm{T}}\mathbf{z} + \sum_{z=1}^{Z} \mathbb{X} \times_1 \mathbf{w}_{1,z} \times_2 \mathbf{w}_{2,z} \times_3 \cdots \times_O \mathbf{w}_{O,z}, \tag{4.39}$$

where $\zeta \in \mathbb{R}^{R_0}$ is a weight vector and R_0 is equal to the number of additional variables included. In addition to the array-valued predictor $\mathbb{X}$, another covariate $\mathbf{z}$ is useful in instances where we would have additional information available, such as when working with image data like imaging modalities (for example Magnetic Resonance Imaging (MRI)) (Zhou et al., 2013). For example, when checking for Attention Deficit Hyperactivity Disorder (ADHD) in brain imaging analysis, along with the fMRI image of the brain for a subject we could include additional information about the image like the age and gender of the subject. From equation (4.39), the number of parameters that need to be estimated would be updated to $R_0 + Z\sum_{N=1}^{O} R_N$. We will be referring to this model as a *CP TR* model.

h provides a principled statistical framework for parameter inference.

4.4.1.2: Multinomial CP TR Model

Cao et al. (2022) adapted the work by Zhou et al. (2013) and proposed a multinomial TR model to discriminate between healthy patients and patients with Parkinsons disease who have depression and those who do not. We will be going into detail on their model in this section since we will use it in our application.

Let n denote the sample size of the dataset consisting of tensors of order O $\mathbb{X}_i$ for $i = 1, ..., n$. Also given are the labels $l_i \in \{1, ..., L\}$ of each sample $\mathbb{X}_i$ in the vector $\mathbf{l}$, where L represents the number of unique labels. We will represent the categorical variable $\mathbf{l}$ using a 1-of-K encoding as in Cao et al. (2022). For $i = 1, ..., n$ and $r = 1, ..., L$, let l_{ir} be a binary variable indicating whether the i^{th} sample has the r^{th} label or not. The resulting 1-of-K encoded categorical variable will be a matrix $\mathbf{L}_i$ of size $(n \times L)$ and is represented

as follows

$$\mathbf{L}_i = \begin{bmatrix} l_{11} & l_{12} & \dots & l_{1L} \\ l_{21} & l_{22} & \dots & l_{2L} \\ l_{31} & l_{32} & \dots & l_{3L} \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nL} \end{bmatrix}. \tag{4.40}$$

Each of the $\mathbb{X}_i$ can be assigned any of the L labels with probabilities $p_i^{(1)}, p_i^{(2)}, ..., p_i^{(L)}$, respectively, where $p_i^{(1)} + p_i^{(2)} + ... + p_i^{(L)} = 1$. Given an order O tensor $\mathbb{X}_i$ and a covariate vector $\mathbf{z}_i \in \mathbb{R}^{R_0}$, the general form of a multinomial TR model with L categories can be expressed as:

$$l_{i(r+1)} \sim \text{Multinomial}(p_i^{(1)}, p_i^{(2)}, ..., p_i^{(L)}), \tag{4.41}$$

$$\ln\left(\frac{p_i^{(r)}}{p_i^{(L)}}\right) = b^{(r)} + (\zeta^{(r)})^{\mathrm{T}}\mathbf{z}_i + \langle \mathbb{X}_i, \mathbb{W}^{(r)} \rangle, \tag{4.42}$$

for $r = 1, ..., L-1$ and $i = 1, ..., n$. Since we have L labels, the systematic part of the GLM model consists of $L-1$ equations as seen in equation (4.42). Each of the weight tensors $\mathbb{W}^{(r)}$ can undergo a tensor decomposition. In the case of CP decomposition, we assume that $\mathbb{W}^{(r)}$ can be decomposed into Z rank-one tensors, i.e.,

$$\mathbb{W}^{(r)} = [\![\mathbf{W}_1^{(r)}, ..., \mathbf{W}_O^{(r)}]\!] = \sum_{z=1}^{Z} \mathbf{w}_{1,z}^{(r)} \circ \cdots \circ \mathbf{w}_{O,z}^{(r)}, \tag{4.43}$$

where $\mathbf{w}_{1,z}^{(r)} = [w_{1,z,1}^{(r)}, w_{1,z,2}^{(r)}, ..., w_{1,z,R_1}^{(r)}]^{\mathrm{T}} \in \mathbb{R}^{R_1}$, ..., and $\mathbf{w}_{O,z}^{(r)} \in \mathbb{R}^{R_O}$ for $z = 1, ..., Z$ and factor matrices $\mathbf{W}_1^{(r)} = [\mathbf{w}_{1,1}^{(r)}, ..., \mathbf{w}_{1,Z}^{(r)}] \in \mathbb{R}^{R_1 \times Z}$, ..., and $\mathbf{W}_O^{(r)} = [\mathbf{w}_{O,1}^{(r)}, ..., \mathbf{w}_{O,Z}^{(r)}] \in \mathbb{R}^{R_O \times Z}$. A tensor decomposition is done for each $\mathbb{W}^{(r)}$ for $r = 1, ..., L-1$, where the multinomial model given in equations (4.41) and (4.42) have a total of $(L-1)(R_0 + Z\sum_{N=1}^{O} R_N)$ parameters in CP decomposition.

4.4.2: Maximum Likelihood Estimation

Given the CP TR model defined in Section 4.4.1, we now go on to cover an efficient algorithm for the MLE of such a model.

4.4.2.1: MLE for CP TR Model

Given that we have a sample of n order O tensors $\mathbb{X}_i$ and a response variable $\mathbf{l}$, i.e.,

the observed dataset $\mathcal{D} = \{(\mathbb{X}_i, l_i)\}_{i=1}^n$, the conditional log-likelihood of the parameters given the data is

$$l(b, \mathbf{W}_1, \ldots, \mathbf{W}_O \mid \mathcal{D}) = \sum_{i=1}^{n} \log \mathbb{P}\big(l_i \mid \mathbb{X}_i; b, \mathbf{W}_1, \ldots, \mathbf{W}_O, \phi\big). \tag{4.44}$$

Using the exponential-family form in (4.33), this becomes

$$l(b, \mathbf{W}_1, ..., \mathbf{W}_O|\mathcal{D}) = \sum_{i=1}^{n} \Big(\frac{l_i \theta_i - b(\theta_i)}{a(\phi)} + c(l_i, \phi) \Big), \tag{4.45}$$

where θ_i is affiliated to the parameters $(b, \mathbf{W}_1, ..., \mathbf{W}_O)$ via equation (4.36).

To estimate the parameters of equation (4.45), the aim is to maximize the log-likelihood. Note that, although equation (4.36) is not linear in $(\mathbf{W}_1, ..., \mathbf{W}_O)$ jointly, it remains linear in each individual $\mathbf{W}_N$ (Zhou et al., 2013). When fixing all other parameters except for $\mathbf{W}_N \in \mathbb{R}^{R_N \times Z}$ and using CP decomposition as in equation (4.37), we can see that equation (4.36) can be simplified to

$$\begin{aligned} g(\mu(\mathbb{X})) &= b + \langle \mathbb{X}, \mathbb{W} \rangle \\ &= b + \langle \mathbb{X}_{(N)}, \mathbb{W}_{(N)} \rangle \\ &= b + \langle \mathbb{X}_{(N)}, \mathbf{W}_N (\mathbf{W}_O \odot \cdots \odot \mathbf{W}_{N+1} \odot \mathbf{W}_{N-1} \odot \cdots \odot \mathbf{W}_1)^{\mathrm{T}} \rangle \\ &= b + \langle \mathbb{X}_{(N)} (\mathbf{W}_O \odot \cdots \odot \mathbf{W}_{N+1} \odot \mathbf{W}_{N-1} \odot \cdots \odot \mathbf{W}_1), \mathbf{W}_N \rangle, \end{aligned} \tag{4.46}$$

since we can substitute the mode-N matricization $\mathbb{W}_{(N)}$ as in equation (4.25).

The *block relaxation* algorithm is an iterative method that partitions variables into disjoint blocks and updates one block at a time while holding the others fixed. At each iteration, the algorithm would optimize the objective function with respect to that one block. This procedure is then cycled across all blocks until convergence criteria are met. Using the block relaxation algorithm in our case, we would, at each iteration, fix all other parameters $(\mathbf{W}_1, ..., \mathbf{W}_{N-1}, \mathbf{W}_{N+1}, ..., \mathbf{W}_O)$ and alternately update only $(b, \mathbf{W}_N)$ in equation (4.45), where this is done for all $N = 1, ..., O$ (de Leeuw (1994), Lange (2010)). This means that, at an iteration $q + 1$ and for $N = 1, ..., O$, we would be updating $\mathbf{W}_N$ (denoted as $\mathbf{W}_N^{(q+1)}$) using

$$\mathbf{W}_N^{(q+1)} = \text{argmax}_{\mathbf{W}_N} \, l\left(b^{(q)}, \mathbf{W}_1^{(q+1)}, \ldots, \mathbf{W}_{N-1}^{(q+1)}, \mathbf{W}_N, \mathbf{W}_{N+1}^{(q)}, ..., \mathbf{W}_O^{(q)} \;\middle|\; \mathcal{D}\right), \tag{4.47}$$

where after all of $N = 1, ..., O$ is covered then we would update b (denoted as $b^{(q+1)}$) using

$$b^{(q+1)} = \text{argmax}_b \, l\left(b, \mathbf{W}_1^{(q+1)}, \ldots, \mathbf{W}_{N-1}^{(q+1)}, \mathbf{W}_N^{(q+1)}, \mathbf{W}_{N+1}^{(q+1)}, ..., \mathbf{W}_O^{(q+1)} \;\middle|\; \mathcal{D}\right). \tag{4.48}$$

The sub-problem considered when only updating $\mathbf{W}_N$ is a classical GLM regression problem with parameter term $\mathbf{W}_N$ and $\mathbb{X}_{(N)}(\mathbf{W}_O \odot \cdots \odot \mathbf{W}_{N+1} \odot \mathbf{W}_{N-1} \odot \cdots \odot \mathbf{W}_1)$ as the predictor term, where this consists of only ZR_N parameters. If including the covariates $\mathbf{z}$ as in equation (4.39), then this would be included in the parameters ($\mathcal{D} = \{(\mathbb{X}_i, z_i, l_i)\}_{i=1}^n$) and updated along with b as in equation (4.48), i.e.,

$$\left(b^{(q+1)}, \zeta^{(q+1)}\right) = \text{argmax}_{b,\zeta}\, l\left(b, \zeta, \mathbf{W}_1^{(q+1)}, \ldots, \mathbf{W}_{N-1}^{(q+1)}, \mathbf{W}_N^{(q+1)}, \mathbf{W}_{N+1}^{(q+1)}, ..., \mathbf{W}_O^{(q+1)} \mid \mathcal{D}\right). \tag{4.49}$$

The matrices of regression parameters $\mathbf{W}_1, ..., \mathbf{W}_O$ would be randomly initialized at the start before the first iteration (denoted as $\mathbf{W}_1^{(0)}, ..., \mathbf{W}_O^{(0)}$) and the parameters b and ζ would be initialized using $(b^{(0)}, \zeta^{(0)}) = \text{argmax}_{b,\zeta}\, l(b, \zeta, \mathbf{0}, ..., \mathbf{0})$ (Zhou et al., 2013). Note that the rank Z used in CP decomposition from equation (4.37) is assumed to be known, so estimating an adequate rank beforehand is important.

The overall estimation procedure for equation (4.49) boils down to a sequence of classical GLM problems which continues until the log-likelihood of the previous iteration is larger than the log-likelihood of the current iteration, i.e.,

$$l\left(b^{(q+1)}, \mathbf{W}_1^{(q+1)}, ..., \mathbf{W}_O^{(q+1)} \mid \mathcal{D}\right) < l\left(b^{(q)}, \mathbf{W}_1^{(q)}, ..., \mathbf{W}_O^{(q)} \mid \mathcal{D}\right),$$

or until a specified maximum number of iterations passed. When using a Gaussian model, the entire procedure would be similar to the ALS method (de Leeuw et al., 1976).

Regarding the convergence of the block relaxation algorithm, its objective function is increased monotonically, which ensures that its stopping rule is well-defined (Zhou et al., 2013). This is due to the algorithm being numerically stable and the convergence of $l\left(b^{(q)}, \zeta^{(q)}, \mathbf{W}_1^{(q)}, ..., \mathbf{W}_O^{(q)} \mid \mathcal{D}\right)$ being guaranteed over iterations given that $l\left(b, \zeta, \mathbf{W}_1, ..., \mathbf{W}_O \mid \mathcal{D}\right)$ is bounded above. Due to the random initialization of parameters, the algorithm almost invariably converges to a local maximum. As a result, running the algorithm multiple times is recommended to identify a suitably optimal local maximum.

Given the following notation for the sequence of parameters $\theta = (b, \zeta, \mathbf{W}_1, \ldots, \mathbf{W}_O)$ and that the algorithm map is defined by M, i.e., $M(\theta^{(q)}) = \theta^{(q+1)} = (b^{(q+1)}, \zeta^{(q+1)}, \mathbf{W}_1^{(q+1)}, ..., \mathbf{W}_O^{(q+1)})$, then Proposition 4.1 summarizes the convergence properties of the block relaxation algorithm as defined in Zhou et al. (2013).

Proposition 4.1: *Assume that (i) the log-likelihood function $l(\theta)$ is continuous, coercive, i.e., the set $\{\theta : l(\theta) \geq l(\theta^{(0)})\}$ is compact, and bounded above, (ii) the objective function in each block update of the block relaxation algorithm is strictly concave, and (iii) the set of stationary points (modulo scaling and permutation indeterminacy) of $l(\theta)$ is isolated. With these assumptions, these results follow:*

- *(Global convergence) The sequence* $\theta^{(q)}$ *generated by the block relaxation algorithm at iteration* q *converges to a stationary point of* $l(\theta)$.
- *(Local convergence) Let* $\theta^{(\infty)} = (b^{(\infty)}, \zeta^{(\infty)}, \boldsymbol{W}_1^{(\infty)}, ..., \boldsymbol{W}_O^{(\infty)})$ *be a strict local maximum of* $l(\theta)$. *The iterates generated by the block relaxation algorithm are locally attracted to* $\theta^{(\infty)}$ *for* $\theta^{(0)}$ *sufficiently close to* $\theta^{(\infty)}$.

Proof: Given assumptions (i), (ii) and (iii), we will first prove global convergence and then move on to prove local convergence.

From (ii), the block update is well-defined and differentiable, which means that the algorithm map M is a composition of $O+1$ differentiable maps. By the implicit function theorem, this means that M is continuous. Given that $\theta^{(q)}$ is the sequence generated by M in iteration q and θ is any accumulation point of $\theta^{(q)}$, we know that $l(\theta^{(q+1)}) = l(M(\theta^{(q)}) \geq l(\theta^{(q)})$ since otherwise the estimation procedure is stopped. By continuity of l and M, taking limit yields $l(M(\theta)) = l(\theta)$, which means that the accumulation point θ is a stationary point of $l(\theta)$. The set of all accumulation points is contained in the set $\{\theta : l(\theta) \geq l(\theta^{(0)})\}$ and it follows by (i) that it is compact as a result, which means that it is connected (Lange, 2010). By (iii), the set of stationary points of $l(\theta)$ is isolated, meaning that the amount of stationary points is finite. The set of accumulation points would be a connected subset of the set of stationary points, meaning that it is a single point and proves that $\theta^{(q)}$ converges to a stationary point of $l(\theta)$.

The Hessian of the objective function l at $\theta^{(\infty)}$ is defined to be

$$d^2 l(\theta^{(\infty)}) = \begin{bmatrix} d_{00}^2 l & & \mathbf{0} & \\ & d_{11}^2 l & \cdots & d_{1O}^2 l \\ \mathbf{0} & \vdots & \ddots & \vdots \\ & d_{O1}^2 l & \cdots & d_{OO}^2 l \end{bmatrix},$$

which can be represented in parts as

$$d^2 l(\theta^{(\infty)}) = \mathbf{D} + \mathbf{L} + \mathbf{L}^{\mathrm{T}},$$

where $\mathbf{D}$ is the block diagonal part composed of diagonal blocks $d_{NN}^2 l$ for $N = 0, 1, ..., O$ and $\mathbf{L}$ is the strictly lower block triangular part. Since $\theta^{(\infty)}$ is a strict local maximum of $l(\theta)$, then $d^2 l(\theta^{(\infty)})$ is strictly negative definite implying that the diagonal blocks $d_{NN}^2 l$ for $N = 0, 1, ..., O$ are also strictly negative definite. This means that $\mathbf{D} + \mathbf{L}$, which represents the lower block triangular part of $d^2 l(\theta^{(\infty)})$, is invertible since it shares the same eigenvalues as the diagonal blocks. The differential of the algorithm map M can be represented as

$$dM(\theta^{(\infty)}) = -(\mathbf{D} + \mathbf{L})^{-1} \mathbf{L}^{\mathrm{T}},$$

and, since $\mathbf{D} + \mathbf{L}$ is invertible, $dM(\theta^{(\infty)})$ has a spectral radius strictly less than 1. Ostrowski's theorem states that, if the spectral radius of $dM(\theta^{(\infty)})$ is strictly less than 1, then the sequence $M(\theta^{(q)}) = \theta^{(q+1)}$ is locally attracted to $\theta^{(\infty)}$ (Ostrowski, 1960). As such, by Ostrowski's theorem, the iterates generated by the block relaxation algorithm are locally attracted to $\theta^{(\infty)}$. ■

Having established the MLE for the CP TR model, we now move on to cover MLE for the multinomial CP TR model in the next section.

4.4.2.2: MLE for Multinomial CP TR Model

In the case of CP decomposition with rank Z, given n order O tensors $\mathbb{X}_i$ and the binary indicators of the L classes $l_{i1}, ..., l_{iL}$ for $i = 1, ..., n$, i.e., the observed dataset $\mathcal{D} = \{(\mathbb{X}_i, l_{i1}, \ldots, l_{iL})\}_{i=1}^{n}$, then the parameter vector $\theta = (b^{(1)}, \mathbf{W}_1^{(1)}, ..., \mathbf{W}_O^{(1)}, ..., b^{(L-1)}, \mathbf{W}_1^{(L-1)}, ..., \mathbf{W}_O^{(L-1)})$ will be estimated by maximizing the following log-likelihood:

$$l(\theta \mid \mathcal{D}) = \sum_{i=1}^{n} \sum_{r=1}^{L} l_{ir} \ln(p_i^{(r)}), \tag{4.50}$$

where

$$p_i^{(r)} = \frac{\exp(b^{(r)} + \langle [\![\mathbf{W}_1^{(r)}, ..., \mathbf{W}_O^{(r)}]\!], \mathbb{X}_i \rangle)}{1 + \exp(b^{(1)} + \langle [\![\mathbf{W}_1^{(1)}, ..., \mathbf{W}_O^{(1)}]\!], \mathbb{X}_i \rangle) + ... + \exp(b^{(L-1)} + \langle [\![\mathbf{W}_1^{(L-1)}, ..., \mathbf{W}_O^{(L-1)}]\!], \mathbb{X}_i \rangle)}, \tag{4.51}$$

for $r = 1, ..., L-1$ and

$$p_i^{(L)} = \frac{1}{1 + \exp(b^{(1)} + \langle [\![\mathbf{W}_1^{(1)}, ..., \mathbf{W}_O^{(1)}]\!], \mathbb{X}_i \rangle) + ... + \exp(b^{(L-1)} + \langle [\![\mathbf{W}_1^{(L-1)}, ..., \mathbf{W}_O^{(L-1)}]\!], \mathbb{X}_i \rangle)}, \tag{4.52}$$

for $r = L$.

As we saw, when using the block relaxation algorithm, each sub-problem can be reduced to a classical GLM problem. However, the sub-problems can be solved using other techniques as well. In the case of multinomial TR, the likelihood becomes very complicated and to bypass this, Cao et al. (2022) utilized gradient descent methods like the Adam optimizer to solve the sub-problems. To estimate the parameters that maximize the log-likelihood in our application, we will be using a block relaxation algorithm and gradient ascent to solve its sub-problems. We will briefly go into the partial derivatives needed in the gradient ascent method to update the weights. For ease of notation, we will be representing the following two calculations in short as follows

$$F_{i,1}^{(r)} = \exp(b^{(r)} + \langle [\![\mathbf{W}_1^{(r)}, ..., \mathbf{W}_O^{(r)}]\!], \mathbb{X}_i \rangle), \tag{4.53}$$

for classes $r = 1, ..., L-1$ and

$$F_{i,2} = 1 + \exp(b^{(1)} + \langle [\![\mathbf{W}_1^{(1)}, ..., \mathbf{W}_O^{(1)}]\!], \mathbb{X}_i \rangle) + \cdots + \exp(b^{(L-1)} + \langle [\![\mathbf{W}_1^{(L-1)}, ..., \mathbf{W}_O^{(L-1)}]\!], \mathbb{X}_i \rangle). \tag{4.54}$$

In the case of CP decomposition, we will be finding the partial derivative of the log-likelihood function from equation (4.50) with respect to the bias vector and each factor matrix, i.e., for each class $r = 1, ..., L-1$ and $N = 1, ..., O$, we want to find

$$\frac{\partial l}{\partial \mathbf{b}} = \begin{bmatrix} \frac{\partial l}{\partial b^{(1)}} \\ \frac{\partial l}{\partial b^{(2)}} \\ \vdots \\ \frac{\partial l}{\partial b^{(6)}} \end{bmatrix}, \tag{4.55}$$

and

$$\frac{\partial l}{\partial \mathbf{W}_N^{(r)}} = \begin{bmatrix} \frac{\partial l}{\partial w_{N,1,1}^{(r)}} & \frac{\partial l}{\partial w_{N,2,1}^{(r)}} & \cdots & \frac{\partial l}{\partial w_{N,Z,1}^{(r)}} \\ \frac{\partial l}{\partial w_{N,1,2}^{(r)}} & \frac{\partial l}{\partial w_{N,2,2}^{(r)}} & \cdots & \frac{\partial l}{\partial w_{N,Z,2}^{(r)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial w_{N,1,R_N}^{(r)}} & \frac{\partial l}{\partial w_{N,2,R_N}^{(r)}} & \cdots & \frac{\partial l}{\partial w_{N,Z,R_N}^{(r)}} \end{bmatrix}. \tag{4.56}$$

Also, we will be representing the element (x, y, z) of $\mathbb{X}_i$ as $x_{i,xyz}$ in short.

The partial derivatives of the log-likelihood with respect to $w_{N,z,r_N}^{(r)}$ found in equation (4.56) for $N = 1, ..., O$ and $r_N = 1, ..., R_N$ can be found as follows:

$$\begin{aligned} \frac{\partial l}{\partial w_{N,z,r_N}^{(r)}} = \sum_{i=1}^{n} \Bigg[& l_{ir} \{ \sum_{r_1=1}^{R_1} \cdots \sum_{r_{N-1}=1}^{R_{N-1}} \sum_{r_{N+1}=1}^{R_{N+1}} \cdots \sum_{r_O=1}^{R_O} x_{i,r_1 \cdots r_O} w_{1,z,r_1}^{(r)} \cdots w_{N-1,z,r_{N-1}}^{(r)} w_{N+1,z,r_{N+1}}^{(r)} \cdots w_{O,z,r_O}^{(r)} \\ & - \frac{\sum_{r_1=1}^{R_1} \cdots \sum_{r_{N-1}=1}^{R_{N-1}} \sum_{r_{N+1}=1}^{R_{N+1}} \cdots \sum_{r_O=1}^{R_O} x_{i,r_1 \cdots r_O} w_{1,z,r_1}^{(r)} \cdots w_{N-1,z,r_{N-1}}^{(r)} w_{N+1,z,r_{N+1}}^{(r)} \cdots w_{O,z,r_O}^{(r)} F_{i,1}^{(r)}}{F_{i,2}} \} \\ & + \sum_{r^*=1, r^* \neq r}^{L-1} l_{ir^*} \{ - \frac{\sum_{r_1=1}^{R_1} \cdots \sum_{r_{N-1}=1}^{R_{N-1}} \sum_{r_{N+1}=1}^{R_{N+1}} \cdots \sum_{r_O=1}^{R_O} x_{i,r_1 \cdots r_O} w_{1,z,r_1}^{(r)} \cdots w_{N-1,z,r_{N-1}}^{(r)} w_{N+1,z,r_{N+1}}^{(r)} \cdots w_{O,z,r_O}^{(r)} F_{i,1}^{(r)}}{F_{i,2}} \} \Bigg]. \end{aligned} \tag{4.57}$$

The partial derivative of the log-likelihood with respect to $b^{(r)}$ found in equation (4.55) for $r = 1, ..., L-1$ can be found using

$$\frac{\partial l}{\partial b^{(r)}} = \sum_{i=1}^{n} \left[l_{ir} \{ 1 - \frac{F_{i,1}^{(r)}}{F_{i,2}} \} + \sum_{r^*=1, r^* \neq r}^{L-1} l_{ir^*} \{ - \frac{F_{i,1}^{(r)}}{F_{i,2}} \} \right]. \tag{4.58}$$

Having introduced the MLE procedure for the CP TR model, we now turn to the inferential properties of the estimator in the next section.

4.4.3: Inference

This section presents the statistical theory of MLE for the CP TR model proposed in Section 4.4.1. We discuss the score function, Fisher information matrix, and key properties including identifiability, consistency, and asymptotic normality. The score and

information matrix are central to estimation and inference, and are particularly useful in likelihood-ratio tests. Identifiability refers to the ability to uniquely estimate model parameters with infinite data. Due to the uniqueness issues outlined in Section 4.3.2, the models parametrization is nonidentifiable. We address this issue in Section 4.4.3.2. As the sample size increases, MLE exhibits consistency—where estimates converge to the true parameters—and asymptotic normality—where estimates approximate a normal distribution. These properties are explored in Section 4.4.3.3.

It is worth noting that MLE for the Multinomial CP TR Model in Section 4.4.2.2 does not yield an explicit, closed-form solution. However, the score function, Fisher information matrix, and key properties can still be inferred in the context of an iterative estimation process by estimating them over multiple initializations (other than identifiability, which is model-based). As mentioned in Section 4.4.2.1 regarding the block relaxation algorithm, it may converge to a local maximum, which would affect these theoretical properties since they would not necessarily capture the properties of the global maximum. For example, if the algorithm converges to a local maximum, the score function estimated at that point may not reflect the true parameter values that maximize the likelihood globally. As such, running the algorithm multiple times with different initializations, then estimating and comparing the log-likelihood value and the theoretical properties for each initialization is important (including performance metrics comparisons too), although finding a global maximum is not guaranteed.

To simplify this section, the intercept b and covariates $\mathbf{z}$ are omitted, although the results shown can be extended to include them.

4.4.3.1: Score and Fisher Information

First, we will quickly go over some standard notation used in this section which was used by Zhou et al. (2013). Given a scalar function f, then the gradient ∇f is a column vector, $df = [\nabla f]^{\mathrm{T}}$ is its differential, and $d^2 f$ is its Hessian matrix. Given a multivariate function $g : \mathbb{R}^p \mapsto \mathbb{R}^q$, then the Jacobian matrix $Dg \in \mathbb{R}^{p \times q}$ is comprised of the partial derivatives $\frac{\partial g_j}{\partial x_i}$.

We will be going through the Jacobian and Hessian matrices of the systematic part $\eta = g(\mu)$ as in equation (4.34) for CP decomposition. The gradient $\nabla \eta(\mathbf{W}_1, ..., \mathbf{W}_O) \in \mathbb{R}^{Z \sum_{N=1}^{O} R_N}$ is given by

$$\nabla \eta(\mathbf{W}_1, ..., \mathbf{W}_O) = [\mathbf{J}_1 \, \mathbf{J}_2 \cdots \mathbf{J}_O]^{\mathrm{T}} \mathrm{vec}(\mathbb{X}), \tag{4.59}$$

where $\mathbf{J}_N \in \mathbb{R}^{\prod_{N=1}^{O} R_N \times R_N Z}$ is the Jacobian matrix given by

$$\mathbf{J}_N = D\mathbf{W}(\mathbf{W}_N) = \mathbf{\Pi}_N [(\mathbf{W}_O \odot \cdots \odot \mathbf{W}_{N+1} \odot \mathbf{W}_{N-1} \odot \cdots \odot \mathbf{W}_1) \otimes \mathbf{I}_{R_N}], \tag{4.60}$$

$\mathbf{\Pi}_N \in \mathbb{R}^{\prod_{N=1}^{O} R_N \times \prod_{N=1}^{O} R_N}$ is the permutation matrix which rearranges $\text{vec}(\mathbb{W}_{(N)})$ to $\text{vec}(\mathbb{W})$, i.e., $\text{vec}(\mathbb{W}) = \mathbf{\Pi}_N \text{vec}(\mathbb{W}_{(N)})$, and $\mathbf{I}_{R_N}$ is an identity matrix. The Hessian matrix defined as $d^2\eta(\mathbf{W}_1, ..., \mathbf{W}_O) \in \mathbb{R}^{Z \sum_{N=1}^{O} R_N \times Z \sum_{N=1}^{O} R_N}$ has entries

$$h_{(i_N,z),(i_{N'},z')} = 1_{\{z=z', N \neq N'\}} \sum_{j_N = i_N, j_{N'} = i_{N'}} x_{j_1,...,j_O} \prod_{N'' \neq N, N'} w_{j_{N''}z}, \tag{4.61}$$

and can be partitioned into O^2 sub-blocks as follows

$$\begin{bmatrix} \mathbf{0} & * & \cdots & * \\ \mathbf{H}_{21} & \mathbf{0} & * & \vdots \\ \vdots & \vdots & \ddots & * \\ \mathbf{H}_{O1} & \mathbf{H}_{O2} & \cdots & \mathbf{0} \end{bmatrix}, \tag{4.62}$$

where $\mathbf{H}_{NN'} \in \mathbb{R}^{R_N Z \times R_{N'} Z}$ has $R_N R_{N'} Z$ non-zero elements that can be found from the matrix $\mathbb{X}_{(NN')}(\mathbf{W}_O \odot \cdots \odot \mathbf{W}_{N+1} \odot \mathbf{W}_{N-1} \odot \cdots \odot \mathbf{W}_{N'+1} \odot \mathbf{W}_{N'-1} \odot \cdots \odot \mathbf{W}_1)$.

Given the log-density $l(\mathbf{W}_1, ..., \mathbf{W}_O | l) = \ln(\mathbb{P}[l | \mathbf{W}_1, ..., \mathbf{W}_O])$ of the GLM, we will be finding the score function, Hessian matrix and Fisher information matrix of the CP TR model. Considering the CP TR model given in equations (4.33) and (4.38), then we have that the score function/vector is given by

$$\nabla l(\mathbf{W}_1, ..., \mathbf{W}_O) = \frac{(l - \mu)\mu'(\eta)}{\sigma^2} \nabla \eta(\mathbf{W}_1, ..., \mathbf{W}_O). \tag{4.63}$$

The Hessian matrix of the log-density is given by

$$\begin{aligned} H(\mathbf{W}_1, ..., \mathbf{W}_O) = &- \frac{[\mu'(\eta)]^2}{\sigma^2} \nabla \eta(\mathbf{W}_1, ..., \mathbf{W}_O) d\eta(\mathbf{W}_1, ..., \mathbf{W}_O) \\ &+ \frac{(l - \mu)\theta''(\eta)}{\sigma^2} \nabla \eta(\mathbf{W}_1, ..., \mathbf{W}_O) d\eta(\mathbf{W}_1, ..., \mathbf{W}_O) \\ &+ \frac{(l - \mu)\theta'(\eta)}{\sigma^2} d^2\eta(\mathbf{W}_1, ..., \mathbf{W}_O). \end{aligned} \tag{4.64}$$

The Fisher information matrix can be found as follows

$$\begin{aligned} \mathbb{F}(\mathbf{W}_1, ..., \mathbf{W}_O) &= \mathbb{E}[-H(\mathbf{W}_1, ..., \mathbf{W}_O)] \\ &= \text{Var}[\nabla l(\mathbf{W}_1, ..., \mathbf{W}_O) dl(\mathbf{W}_1, ..., \mathbb{W}_O)] \\ &= \frac{[\mu'(\eta)]^2}{\sigma^2} \nabla \eta(\mathbf{W}_1, ..., \mathbf{W}_O) \text{vec}(\mathbb{X})^{\mathrm{T}} [\mathbf{J}_1 \, \mathbf{J}_2 \cdots \mathbf{J}_O]. \end{aligned} \tag{4.65}$$

GLMs using a canonical link function would have that $\theta = \eta$, meaning that $\theta'(\eta) = 1$ and $\theta''(\eta) = 0$.

4.4.3.2: Identifiability

As shown in Section 4.3.2, there are situations where, when using CP decomposition, the parameterization of the TR model will not be unique. As we shall see, under certain restrictions, local identifiability is possible, and under stricter conditions, global identifiability can be found (Zhou et al., 2013).

For the CP TR model, a specific constrained parameterization is necessary for dealing with the nonidentifiability issue due to scaling and permutation indeterminacies. There is a vast number of ways in which the parameter space can be restricted to make it identifiable, where the restrictions made can vary depending on the application at hand. To fix the scaling indeterminacy, given the factor matrices $\mathbf{W}_1, ..., \mathbf{W}_O$, Zhou et al. (2013) adopted an approach for CP decomposition which scales $\mathbf{W}_1, ..., \mathbf{W}_{O-1}$ by fixing the entries of their first Z rows as ones. This helps to fix the scaling indeterminacy since it determines what the entries of the first row of $\mathbf{W}_O$ need to be.

To fix the permutation indeterminacy, the entries of the first row of $\mathbf{W}_O$ are assumed to be distinct and arranged in order of size in descending order, i.e., we would have $w_{O,1} > ... > w_{O,Z}$. With both these constraints in mind, the restricted parameter space is open and convex (Zhou et al., 2013). Given that we do these restrictions, the formulas for the score function, Hessian and Fisher information matrices would need to be updated appropriately. Since the entries of the first rows for factor matrices $\mathbf{W}_1, ..., \mathbf{W}_{O-1}$ are fixed as ones, their respective entries, rows and columns in the score function, Hessian and Fisher information matrices would have to be removed (Zhou et al., 2013).

In cases where $O > 2$, decompositions are still not necessarily unique even when fixing the scaling and permutation indeterminacies. Several authors in the literature proposed conditions for checking uniqueness in such cases, and Zhou et al. (2013) go into a detailed review of them. Since local identifiability is related to the Fisher information matrix (Rothenberg, 1971), a sufficient and necessary condition for local identifiability proven by Zhou et al. (2013) is presented in Proposition 4.2.

Proposition 4.2: *Given that we have an independent and identically distributed sample of n order O tensors $\mathbb{X}_i$ and a response variable $\boldsymbol{l} = [l_1, ..., l_n]^T$ from the CP TR model, assume that $\mathbb{W}_0 = [[\mathbb{W}_{01}, \mathbb{W}_{02}, ..., \mathbb{W}_{0O}]]$ is a parameter point from the restricted parameter space defined earlier and that there exists an open neighborhood of $\mathbb{W}_0$ in which the Fisher information matrix has a constant rank. Then for the CP TR model, we would say that $\mathbb{W}_0$ is locally identifiable if and only if*

$$\mathbb{F}(\mathbb{W}_0) = [\mathbf{J}_1 \, \mathbf{J}_2 \cdots \mathbf{J}_O]^{\mathrm{T}} \left[\sum_{i=1}^{n} \frac{\mu'(\eta_i)^2}{\sigma_i^2} \mathrm{vec}(\mathbb{X}_i)\mathrm{vec}(\mathbb{X}_i)^{\mathrm{T}} \right] [\mathbf{J}_1 \, \mathbf{J}_2 \cdots \mathbf{J}_O]$$

is nonsingular.

From Proposition 4.2, we can see that linear independence of the 'collapsed vectors'

$$[\mathbf{J}_1 \, \mathbf{J}_2 \cdots \mathbf{J}_O]^{\mathrm{T}} \mathrm{vec}(\mathbb{X}_i) \in \mathbb{R}^{Z(\sum_{N=1}^{O} R_N - O + 1)},$$

for CP TR is required for $i = 1, ..., n$. For comparison, identifiability for classical linear regression needs $\mathrm{vec}(\mathbb{X}_i) \in \mathbb{R}^{\prod_{N=1}^{O} R_N}$ to be linearly independent to estimate all parameters, which would require a sample of $n \geq \prod_{N=1}^{O} R_N$ (Zhou et al., 2013).

Global identifiability is possible but, for a finite sample, it is generally hard to get excluding in the linear case, i.e., when $O = 1$. It is said that, given a parameter point $\mathbb{W}_0$ from the restricted parameter space, $\mathbb{W}_0$ is globally identifiable if it can be uniquely decomposed, assuming that scaling and permutation indeterminacies were adjusted, and $\sum_{i=1}^{n} \mathrm{vec}(\mathbb{X}_i)\mathrm{vec}(\mathbb{X}_i)^{\mathrm{T}}$ attains full rank for a specific sample size n (Zhou et al., 2013).

4.4.3.3: Asymptotics

As mentioned in Zhou et al. (2013), asymptotic properties for TR will follow similarly as in MLE or M-estimation. By leveraging the insight that the systematic component $g(\mu)$ of the TR model in equation (4.36) is parameterized as a polynomial of parameters of degree O and that the family of polynomials $\{\langle \mathbb{X}, \mathbb{W}_0 \rangle\}$ constitutes a Vapnik-Cervonenkis (VC) class, where $\mathbb{W}_0$ is a point of the parameter space, then consistency of TR will be proved by applying the standard uniform convergence theorem for M-estimation from van der Vaart (1998). In general, it will be rare to find that the true parameters $\mathbb{W}_{true} \in \mathbb{R}^{R_1 \times \cdots \times R_O}$ can be decomposed to an exact low-rank tensor. Although, the MLE will consistently estimate the best approximation of $\mathbb{W}_{true}$ from all possible parameter points in the parameter space with respect to the Kullback-Leibler distance (Zhou et al., 2013).

By showing that the log-likelihood function of the TR model is Quadratic Mean Differentiable (QMD), Zhou et al. (2013) show that asymptotic normality follows by using a result which relates asymptotic normality to densities that satisfy QMD. We refer the reader to Zhou et al. (2013) for the proof of Theorem 4.1.

Theorem 4.1: *Given that $\mathbb{W}_0$ is a parameter point from a parameter space which is (globally) identifiable up to permutation and the covariates $\mathbb{X}_i$ are independent and identically distributed from a bounded underlying distribution, then it follows that:*

- *(Consistency) The MLE is consistent, i.e., $\hat{\mathbb{W}}_N$ converges to $\mathbb{W}_0$ in probability, in the following models: (1) Normal TR with $\mathbb{W}_0$ contained in a compact parameter space. (2) Binary TR. (3) Poisson TR with $\mathbb{W}_0$ contained in a compact parameter space.*

- *(Asymptotic Normality) For an interior point* $\mathbb{W}_0$ *with nonsingular Fisher information matrix* $\mathbb{F}(\mathbb{W}_0)$ *and* $\hat{\mathbb{W}}_N$ *is consistent, then* $\sqrt{N}[vec(\hat{\mathbb{W}}_N) - vec(\mathbb{W}_0)]$ *converges in distribution to a normal with mean zero and covariance matrix* $\mathbb{F}^{-1}(\mathbb{W}_0)$*.*

With the inferential properties of MLE established, in the next section, we will cover regularization in TR, which is often essential in addressing the high-dimensional nature of TR.

4.5: Regularization

Even when using CP decomposition to reduce the number of parameters in the CP TR model, the sample size in a study can be limited, which would lead to situations where the number of parameters exceeds it. In such situations, regularization is often implemented to alleviate the curse of dimensionality, which is the case for our TR application. Regularization in TR is generally done on the factor matrices $\mathbf{W}_1, ..., \mathbf{W}_O$ after applying CP decomposition. Using regularization with the original parameter tensor $\mathbb{W}$ is possible, but the number of parameters considered in the regularized TR model would still generally be too high in practice (Zhou et al., 2013).

Understanding the differences between *scalar penalty functions* and *tensor-specific regularization* is important when applying regularization in TR. Scalar penalty functions are regularization techniques that apply penalties to individual parameters in a model, regardless of the structure of the data. On the other hand, tensor-specific regularization techniques take into consideration the multi-dimensional structure of tensors, i.e., they consider the relationships and interactions between different dimensions of the tensor data. Examples of scalar penalty functions include L_1 and L_2 regularization (Zhou et al., 2013), whereas for tensor-specific regularization techniques, this includes *group* L_1 regularization (Raskutti et al., 2019). With image data, tensor-specific regularization techniques would have an advantage over scalar penalty functions since this type of data generally has intricate interdependencies; however, the latter is easier to implement and less computationally demanding to handle. Because of this, we will be mainly focusing on scalar penalty functions, particularly L_1 regularization. We have chosen L_1 regularization since it will be useful to induce sparsity in the images and identify sub-regions associated with the response attributes (Zhou et al., 2013).

From equation (4.45), the following regularized log-likelihood function will be maximized:

$$l(b, \mathbf{W}_1, ..., \mathbf{W}_O|\mathcal{D}) - \sum_{N=1}^{O}\sum_{z=1}^{Z}\sum_{r=1}^{R_N} P_{\lambda_{LR}}(|w_{N,z,r}|), \tag{4.66}$$

where P is a scalar penalty function, and λ_{LR} designates an index for a specific member within the penalty family. For L_1 regularization, we have $\lambda_{LR} = 1$ and $P_1(|w_{N,z,r}|) = |w_{N,z,r}|$.

The sub-problem considered in the block relaxation algorithm when updating $\mathbf{W}_N$ from equation (4.47) would be updated as follows

$$\mathbf{W}_N^{(q+1)} = \text{argmax}_{\mathbf{W}_N} \Bigg(l\left(b^{(q)}, \mathbf{W}_1^{(q+1)}, \ldots, \mathbf{W}_{N-1}^{(q+1)}, \mathbf{W}_N, \mathbf{W}_{N+1}^{(q)}, ..., \mathbf{W}_O^{(q)} | \mathcal{D}\right) - \sum_{N=1}^{O} \sum_{z=1}^{Z} \sum_{r=1}^{R_N} P_\lambda(|w_{N,z,r}|) \Bigg),$$

which is a penalized GLM regression problem, where we are subtracting the penalty since it is a maximization problem. Overall, the estimation procedure follows similarly as defined in Section 4.4.2.

5

Applications

In this chapter, we will apply the methods we presented in the previous chapters to the dance dataset mentioned in Chapter 1. We will first present a detailed description of the dance dataset in Section 5.1. Section 5.2 will describe the objectives we aim to achieve in this application. After that, we will go into the CNN and TR applications in Sections 5.3 and 5.4, respectively, where we will discuss the results in Section 5.5.

5.1: Dataset Details

As mentioned in Chapter 1, while the dancers were dancing the SKE, SME and TE dances, the Kinect-II sensor was monitoring at a rate of 30 measurements per second, where RGB images were captured at each measurement. The main choreographic steps of the three folk dances SKE, SME, and TE consist of seven unique poses. Figure 5.1 presents a visual illustration of the seven unique poses. For easier reference to each pose, from Figure 5.1, we will be referencing the poses from the order of left to right simply as poses 1 to 7, where pose 1 is the 'Initial posture' pose, pose 2 is the 'Cross legs' pose, etc. An easy way to distinguish between poses 2 and 7 is that for the former, the dancers left leg is at the front, whereas for the latter, the dancers left leg is at the back. We will refer to the three dancers as 'Dancer 1', 'Dancer 2', and 'Dancer 3', respectively.

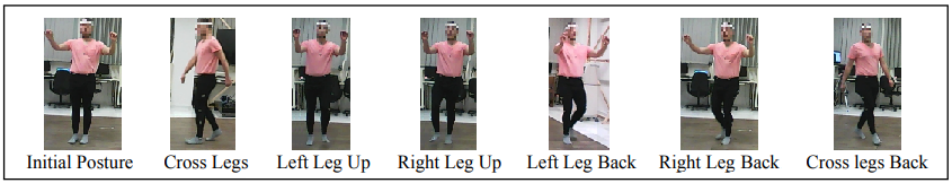


Figure 5.1: The labels of each pose by Dancer 1.

Table 5.1 presents the main choreographic steps in order from start to finish of each folk dance, while Figures 5.2, 5.3 and 5.4 illustrate the main choreographic steps of SKE, SME, and TE which were done by Dancers 2, 3, and 1, respectively.

Dance	**Main Choreographic Steps**
SKE	$1 \longrightarrow 2 \longrightarrow 2 \longrightarrow 2 \longrightarrow 2 \longrightarrow 1 \longrightarrow 7$
SME	$1 \longrightarrow 5 \longrightarrow 2 \longrightarrow 2 \longrightarrow 2 \longrightarrow 1 \longrightarrow 6$
TE	$1 \longrightarrow 2 \longrightarrow 2 \longrightarrow 2 \longrightarrow 1 \longrightarrow 3 \longrightarrow 4 \longrightarrow 3 \longrightarrow 7$

Table 5.1: Table depicting the main choreographic steps of each dance.

Note that each dance consists of different unique poses compared to each other. The SKE dance utilizes only three of the seven poses, which are poses 1, 2, and 7. The SME dance utilizes four poses which are poses 1, 2, 5, and 6. Finally, the TE dance has the greatest number of poses of the three dances with five poses, which are poses 1, 2, 3, 4, and 7. Table 5.2 shows the distribution of poses per dance for each dancer.

Figure 5.2: The main choreographic steps of SKE (from left to right and top to bottom).

Figure 5.3: The main choreographic steps of SME (from left to right and top to bottom).

Figure 5.4: The main choreographic steps of TE (from left to right and top to bottom).

	Dancer 1			Dancer 2			Dancer 3		
Pose	**SKE**	**SME**	**TE**	**SKE**	**SME**	**TE**	**SKE**	**SME**	**TE**
1	125	158	36	50	88	92	128	161	88
2	130	96	102	268	155	112	246	201	95
3	0	0	38	0	0	16	0	0	42
4	0	0	25	0	0	32	0	0	22
5	0	13	0	0	40	0	0	44	0
6	0	16	0	0	85	0	0	13	0
7	49	0	80	61	0	44	82	0	47
Total	304	283	281	379	368	296	456	419	294

Table 5.2: The number of measurements of each pose for each dancer and dance.

For a dance, when comparing the number of measurements between poses, we can see that the data is quite imbalanced, where some poses are less frequently seen when compared to poses 1 and 2. This makes sense given that poses 1 and 2 are used multiple times in each dance as in Table 5.1.

Note the discrepancy in the frequency of each pose for each dance between the three dancers in Table 5.2, where Dancer 1 has the smallest number of measurements of the

three dancers. This makes sense since, in the data, Dancer 1 only goes through one full rotation of the main choreographic movements from Table 5.1 for each dance, whereas the other two dancers have done two full rotations for each dance. Since the number of measurements of each dance for Dancer 1 is somewhat comparable to the other two dancers, this means that Dancer 1 is staying longer to finish going through some of the main choreographic movements when compared to the other two dancers.

In Chapter 1, we mentioned that dancing experts manually annotated the poses. However, the labels presented by the dancing experts can be interpreted differently from person to person. An example of this is the following: the main choreographic steps of the TE dance follow as in Table 5.1, where Dancers 1 and 3 follow the pattern as described. However, Dancer 2 follows a different pattern. Indeed, the labels of Dancer 2 when the Left Leg Up pose initially comes up are labelled as Cross Legs instead, whereas Dancer 2 then moves on as usual with the Right Leg Up pose. There could be other measurements in the data whose labels can be interpreted differently, such as where the dancers would be following the main choreographic steps as usual, but the labels in between steps could have been labelled more appropriately. This happens given that the poses have some similarities, which could cause difficulties in pose recognition.

5.2: Objective Details

For this application, we will combine the three dances of a dancer as one single dataset, which means there are three datasets in total, one for each of the dancers. Our main objective is to train a CNN and TR model that can correctly predict all seven poses of the underlying dataset. In particular, we will work on this objective in two different trials. The first trial will train a CNN and TR model using the combined dataset of all three dancers as one training and test split and will try to predict the poses of the test set. The second trial will follow similarly but instead use the combined dataset of two dancers as the training set and the dataset of the other dancers as the test set. The result of these two trials will show whether the models can accurately train and predict the poses of the dancers, while the second trial will try to see whether the models are generalizable and able to predict the poses of a different dancer.

Our preprocessing of RGB images consists of normalizing pixel values from $[0, 255]$ to $[0, 1]$, followed by standardization.. For both trials, we will conduct hyperparameter searches in which preprocessing remains fixed, while hyperparameters, model weight initializations, and train–test splits vary. After selecting the optimal models, we will run each one multiple times to assess generalization, keeping preprocessing and hyperparameters fixed but varying weight initializations and train–test splits.

For the first trial, which we will be referring to as 'Trial A', the dataset will consist of a total of 3080 images, which we will be dividing into training and test sets. Table 5.3 shows the frequency of pose labels for this dataset.

For the second trial, which we will be referring to as 'Trial B', the training set will be composed of the combined dance dataset of two dancers, and the test set will be the other dancer. Here, there are three possible combinations of training-test splits, where we will be referring to each combination as simply 'Trial B1', 'Trial B2' and 'Trial B3'. Trial B1 is with the training set consisting of the dances of Dancers 1 and 2, while the test set consists of the dances of Dancer 3. In this case, the training set will consist of a total of 1911 images, whereas the test set consists of a total of 1169 images. Table 5.4 shows the frequency of pose labels for both training and test sets.

Pose Label	**Frequency**
1	926
2	1405
3	96
4	79
5	97
6	114
7	363
Total	**3080**

Table 5.3: Frequencies of each pose label corresponding to Trial A.

Training Set	
Pose Label	**Frequency**
1	549
2	863
3	54
4	57
5	53
6	101
7	234
Total	**1911**

Test Set	
Pose Label	**Frequency**
1	377
2	542
3	42
4	22
5	44
6	13
7	129
Total	**1169**

Table 5.4: Frequencies of each pose label corresponding to Trial B1.

Trial B2 is with the training set consisting of the dances of Dancers 1 and 3 while the test set consists of the dances of Dancer 2. The training set will consist of a total of 2037 images, whereas the test set consists of a total of 1043 images. Table 5.5 shows the frequency of pose labels for both training and test sets.

Trial B3 is with the training set consisting of the dances of Dancers 2 and 3 while the test set consists of the dances of Dancer 1. The training set will consist of a total of 2212 images, whereas the test set consists of a total of 868 images. Table 5.6 shows the frequency of pose labels for both training and test sets.

Training Set	
Pose Label	**Frequency**
1	696
2	870
3	80
4	47
5	57
6	29
7	258
Total	**2037**

Test Set	
Pose Label	**Frequency**
1	230
2	535
3	16
4	32
5	40
6	85
7	105
Total	**1043**

Table 5.5: Frequencies of each pose label corresponding to Trial B2.

Training Set	
Pose Label	**Frequency**
1	607
2	1077
3	58
4	54
5	84
6	98
7	234
Total	**2212**

Test Set	
Pose Label	**Frequency**
1	319
2	328
3	38
4	25
5	13
6	16
7	129
Total	**868**

Table 5.6: Frequencies of each pose label corresponding to Trial B3.

Both the CNN and TR applications were run in Python utilizing popular packages, including NumPy, pandas, and sklearn (scikit-learn). The CNN application code uses the majority of the CNN tools from PyTorch (torch and torchvision packages), where we also used Ray Tune for hyperparameter optimization, since PyTorch and Ray Tune can be integrated together efficiently. The TR application employs CPU multiprocessing through the concurrent.futures and itertools packages. The code used for both applications can be found at the following GitHub link: https://github.com/andrewfspiteri15-cloud/MScdissertation-code.git

In the next section, we will state the performance metrics that we will use and describe a *dummy classifier* that we will use to compare the performance of our models

against, whose performance will be akin to using a purely random decision rule (Martino et al., 2019). The dummy classifier will serve as a baseline for us to compare the significance of our results, while also raising awareness of the class imbalance issue with the performance metrics it gives.

5.2.1: Application Results - Dummy Classifier

A dummy classifier represents a classifier which makes predictions that ignore the features of the inputs, i.e., predictions are made based on a strategy having no insight into the data at all. The strategy of the dummy classifier is based on predicting the most frequent label, which will be the second pose since it is the most popular pose of each dancer, as seen in Section 5.1.

To check the validity of our results, we will be computing multiple performance metrics, notably the test accuracy and the macro-averaged F1-measure at the end of each epoch. We will primarily focus on optimizing the macro-averaged Fβ-measure with $\beta = 1$, as both recall and precision are equally important in our objective. We have chosen these metrics because they weigh all labels equally, i.e., common labels are weighed similarly to rarer labels, and we view them as equally important in our objective. These metrics have also seen successful use in HPC applications in the past (Guerra et al. (2020), Raj et al. (2021)). Note that we will shorten the notation for $F1_{Macro}$ as F1 to save space. Table 5.7 represents the test accuracy and F1-measure of the dummy classifier for Trials A and B.

Training-Test Split	Test Accuracy	Test F1-measure
Trial A	45.617%	0.09
Trial B1	46.364%	0.0905
Trial B2	51.294%	0.0969
Trial B3	37.788%	0.0784

Table 5.7: Results of the dummy classifier for Trials A and B.

When we compare the performance metrics, we will see that the test accuracy is generally an unreliable performance metric compared to the F1-measure due to the imbalanced number of measurements per label in the dataset.

5.3: CNN Application

In Sections 5.3.1 and 5.3.2, we will be going over the details of the CNN architectures that we used for this application including details regarding the hyperparameter search

that we made. Section 5.3.3 will go into how we will prepare the data before processing. Finally, we will then go into our results and discuss them in the remaining sections.

5.3.1: CNN Architectures

We will be making use of several CNN architectures in our application, namely those that have been described in Section 3.4. We will use these CNN architectures since they are commonly utilized in practice and due to the large time consumption in designing a custom CNN architecture from scratch (Albelwi and Mahmood, 2017). As such, we will be using these architectures as they were initially designed and applying them to our problem without altering their structure or hyperparameters, where their parameters will be randomly initialized and trained over time.

In Section 3.4.2, we discuss the depth of the network and the critical depth threshold. In our application, we are interested in seeing how much we can increase the model depth to improve our results, whilst looking to stay below this critical depth threshold. For this application, we have chosen to use the architectures of AlexNet, ResNet18 and ResNet34, where both ResNet18 and ResNet34 are similar in network design but have different depths (Fagbohungbe and Qian, 2021). We have chosen these architectures to have a range of architectures with different depths, where we will compare their performance on the dance dataset. Of these CNNs, AlexNet is the architecture with the most trainable parameters (see Table 5.14) and the least depth at a depth of 8 layers, excluding pooling and nonlinearity layers. We expect its lack of depth means that it will be hard for it to grasp more complex image features compared to the other deeper networks. Although LeNet-5 has the least depth of all CNNs we mentioned in Section 3.4, we did not choose it since, from some testing we had done, we noticed that its feature learning is too limited and also that LeNet-5 was originally meant for small images of size (32 x 32). ResNet was chosen due to its use of skip connections, which are important and efficient for the training of very deep networks (Szegedy et al., 2016). Also, Shwartz-Ziv et al. (2023) utilized several depths of ResNet (8, 32, 50, and 152) and mentioned that larger depths can overfit and underperform on imbalanced data.

5.3.2: Training and Hyperparameter Optimization

To improve the generalization of each model, we will optimize the hyperparameters present in each model as well as utilize batch normalization. First, we will apply the data to the original architectures of AlexNet, ResNet18 and ResNet34. Since AlexNet does not have batch normalization layers unlike the other networks, we will also use a modified version of the AlexNet architecture by adding batch normalization layers after

the convolutional layers and checking for performance improvements. We will refer to this modified network as the 'AlexNet + BN' architecture in short.

We considered SGD and AdamW as our optimization algorithms, where we ran separate hyperparameter optimization searches for each optimization algorithm. To counter the data imbalance problem as mentioned in Section 5.2, we utilized *inverse frequency cross-entropy* loss. Standard cross-entropy loss would be more biased towards majority labels, whereas inverse frequency cross-entropy loss would weight the cross-entropy loss of each label by its inverse frequency, which would assign more weight to the loss of minority labels (Tian et al., 2021). An implementation choice that can be done in PyTorch for minibatch training is to take the mean instead of the sum of the loss of each data point per iteration to calculate the expected loss. This has advantages in that it makes the scale of the loss independent from the batch size, which helps to ensure that all batches contribute equally. Also, since the overall loss per iteration is lower, it reduces the effect of batch size on the learning rate. We experimented using both the mean and the sum and found that models using the former were easier to train, so the results shown use the mean.

For each model, we used a random search approach using Ray Tune in PyTorch to search for optimal sets of hyperparameters (Liaw et al., 2018). We utilized holdout cross-validation, where the training set was divided into training and validation subsets in an 80-20 split. The training subset is necessary for training the parameters of our CNN model using a learning rule to allow the parameters of the model to adjust each iteration, whereas the validation subset is useful for testing the prediction ability.

The hyperparameters we checked were the batch size, learning rate, weight decay parameter, and the hyperparameters unique to the optimization algorithm, i.e., momentum for SGD and β_1 and β_2 for AdamW. With regards to the batch size, we have opted to check for smaller batch sizes since, in cases of class imbalance, smaller batch sizes generally give better results (Shwartz-Ziv et al., 2023). Smaller batch sizes might not necessarily contain samples of the minority classes each time; however, it has been shown that smaller batch sizes improve generalization performance over using larger batch sizes (given the same number of epochs) due to the increased number of parameter updates (Hoffer et al., 2017). For hyperparameter optimization regarding the AdamW optimization algorithm, we ran separate checks where β_1 and β_2 were fixed at 0.9 and 0.999, respectively, and checks where both hyperparameters were unfixed. This was done since the default values of β_1 and β_2 in Adam generally work well in practice; however, we wanted to check if other possible values would be more viable. Table 5.8 displays the ranges considered for each hyperparameter for SGD and AdamW, respectively.

SGD	
Hyperparameter	**Range**
Batch Size	$\{x : 2^x \text{ for } x = 1, 2, ..., 8\}$
Learning rate	$\{x : \frac{1}{x \ln(b-a)} \text{ for } 1\text{x}10^{-6} \leq x \leq 1\}$
Weight Decay	$\{x : \frac{1}{x \ln(b-a)} \text{ for } 1\text{x}10^{-6} \leq x \leq 0.1\}$
Momentum	$\{x : \frac{1}{b-a} \text{ for } 0.1 \leq x \leq 0.999\}$

AdamW	
Hyperparameter	**Range**
Batch Size	$\{x : 2^x \text{ for } x = 1, 2, ..., 8\}$
Learning rate	$\{x : \frac{1}{x \ln(b-a)} \text{ for } 1\text{x}10^{-6} \leq x \leq 1\}$
Weight Decay	$\{x : \frac{1}{x \ln(b-a)} \text{ for } 1\text{x}10^{-6} \leq x \leq 0.1\}$
β_1	$\{x : \frac{1}{b-a} \text{ for } 0 \leq x \leq 0.999\}$
β_2	$\{x : \frac{1}{b-a} \text{ for } 0 \leq x \leq 0.999\}$

Table 5.8: The range of hyperparameter values tested for each hyperparameter.

At each run, for each architecture, a holdout cross-validation using an 80-20 split of the training set is done and several sets of hyperparameters from the hyperparameter ranges listed in Table 5.8 are randomly sampled. The model would then be trained on each set of hyperparameters separately, where the models performance is evaluated on the validation subset according to the validation loss. We then chose the top 3 sets of hyperparameters filtered by the lowest validation loss to present in a table. The model was trained for 50 epochs and implemented early stopping using an *Asynchronous Successive Halving Algorithm* (ASHA) scheduler (Li et al., 2018a). ASHA combines 'random search with principled early stopping in an asynchronous way' (Li et al., 2018a). We refer to Karnin et al. (2013) and Li et al. (2018a) for more information on ASHA. We capped training to 50 epochs due to time restraints, and since we had found through experimentation that it was generally more than enough epochs for the model to converge, although it did limit some models which were slowly converging.

For each model, after all hyperparameter searches are completed, results are then compiled, where the set of hyperparameters with the lowest validation loss is chosen. This is done for SGD and AdamW for a total of two sets of hyperparameters. With each set of hyperparameters, the entire training set is used for training and predictions are made on the test set. The model is trained for 100 epochs with no early stopping. To generalize our results, we ran 5 total runs of each model and will present the highest test accuracy and F1-measure seen in the runs. Also, we will average the result metrics per epoch and present the highest average test accuracy and F1-measure among them.

5.3.3: Data Preprocessing

We have normalized the training and test set RGB images from the range of [0, 255] to the range [0, 1], where they would then be standardized. To standardize the RGB images, the mean and standard deviation of each RGB colour for the training set were found.

The CNN architectures we will use were generally intended for images of specific sizes relative to the architecture itself. AlexNet and ResNet originally utilized RGB images of size (224 x 224 x 3). It is worth noting that the authors of AlexNet had written the size (224 x 224 x 3) in their original paper, but the image sizes should be (227 x 227 x 3) instead (Krizhevsky et al., 2012). Re-scaling images to fit the intended size for these CNN architectures is common in practice; however, this could lead to issues such as reducing image quality if the image size is reduced too much (Luke et al. (2019), Thambawita et al. (2021)). Reducing the quality of an image is a serious issue, where the possible loss of low-level image features could impact the prediction of the image label resulting in noisy labels. However, for CNN architectures with fully connected layers, resizing images to smaller image sizes will reduce the chance of the model to overfit, since the flattened inputs map volume to the fully connected layer will be smaller and hence reduce the number of parameters in the model (Battiti, 1994). Ideally, we would need to resize the dataset images to different sizes for each architecture to determine the optimal image resolution in terms of performance; however, due to time constraints, we have limited ourselves to using the intended image sizes for each architecture.

We will first use the original sizes of our images and present our results in Section 5.3.4. From there, we will then resize the images in the dataset to the intended image sizes for each architecture and present our results in Section 5.3.5.

5.3.4: CNN Application Results - Original Images

We will first review the results of Trial A in Section 5.3.4.1, where we will then move on to the results of Trial B in Section 5.3.4.2. Appendix C1 presents all the results that we had for each trial in this section, including the hyperparameters results and graphs. In the coming subsections, we will be going over some of the results mentioned.

5.3.4.1: Trial A Results

Table 5.9 presents our results for each CNN architecture considered for Trial A and Figure 5.5 presents the average training and test loss per epoch of all 5 runs that we had done for each CNN architecture for both SGD and AdamW. From Table 5.9, we can see that results are somewhat similar between CNN architectures, where ResNet34 using

AdamW had the highest average test accuracy and F1-measure results. The highest test accuracy and F1-measure were returned by ResNet34 using SGD.

	SGD: Trial A			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	97.89	97.565	98.539	**98.864**
H. AVR Acc ± SD	96.672 ± 0.54	96.391 ± 1.06	97.565 ± 0.82	97.338 ± 0.98
H. Test F1	0.974	0.97	0.981	**0.988**
H. AVR F1 ± SD	0.958 ± 0.005	0.954 ± 0.001	0.964 ± 0.006	0.96 ± 0.017

	AdamW: Trial A			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	98.052	98.377	98.214	98.539
H. AVR Acc ± SD	96.916 ± 0.36	96.851 ± 0.48	97.395 ± 0.49	**97.76** ± 0.37
H. Test F1	0.973	0.978	0.985	0.985
H. AVR F1 ± SD	0.955 ± 0.006	0.96 ± 0.001	0.96 ± 0.001	**0.967** ± 0.001

Table 5.9: Results from using each CNN architecture with SGD/Adam for Trial A.

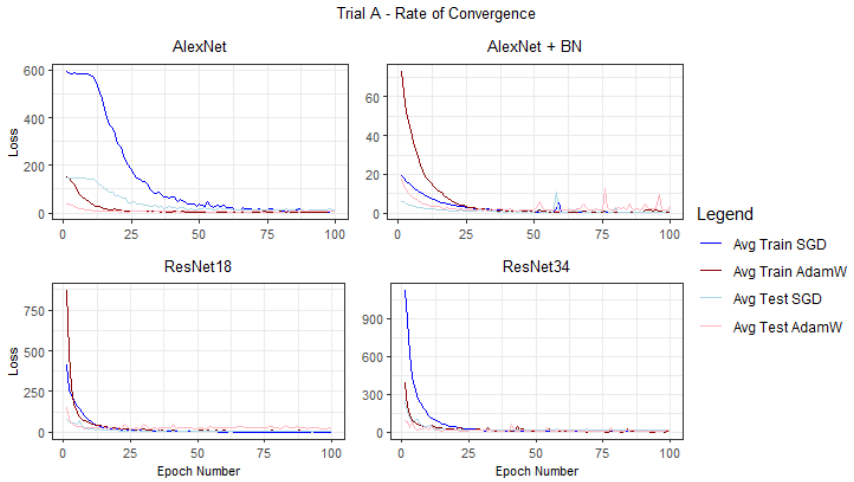


Figure 5.5: The average training and test loss per epoch of all 5 runs of each model per CNN architecture.

For each model, from the hyperparameter tuning done using the training set, we found multiple sets of hyperparameters which had very good performance metrics on both the training and validation sets. From the multiple runs that we had run when using the set of hyperparameters with the lowest validation loss on the test set per CNN architecture, the results that we received were very good and the model was able to consistently classify all 7 poses accurately. Figure 5.5 also shows that all the models converged with no issues.

5.3.4.2: Trial B Results

Tables 5.10 (Top), (Center) and (Bottom) present our results for Trials B1, B2, and B3, respectively, whereas Figure 5.6 presents the average test loss per epoch of all runs per CNN architecture per trial.

	SGD: Trial B1			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	65.526	66.125	65.612	**67.151**
H. AVR Acc ± SD	58.366 ± 3.06	60.642 ± 1.91	56.424 ± 4.67	**61.078** ± 1.72
H. Test F1	0.346	0.379	0.371	**0.462**
H. AVR F1 ± SD	0.268 ± 0.033	0.272 ± 0.391	0.296 ± 0.045	**0.372** ± 0.051

	AdamW: Trial B1			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	57.827	60.479	65.526	63.216
H. AVR Acc ± SD	49.547 ± 4.14	56.45 ± 2.37	59.196 ± 4.2	56.219 ± 3.84
H. Test F1	0.36	0.337	0.41	0.309
H. AVR F1 ± SD	0.199 ± 0.084	0.236 ± 0.03	0.33 ± 0.048	0.267 ± 0.043

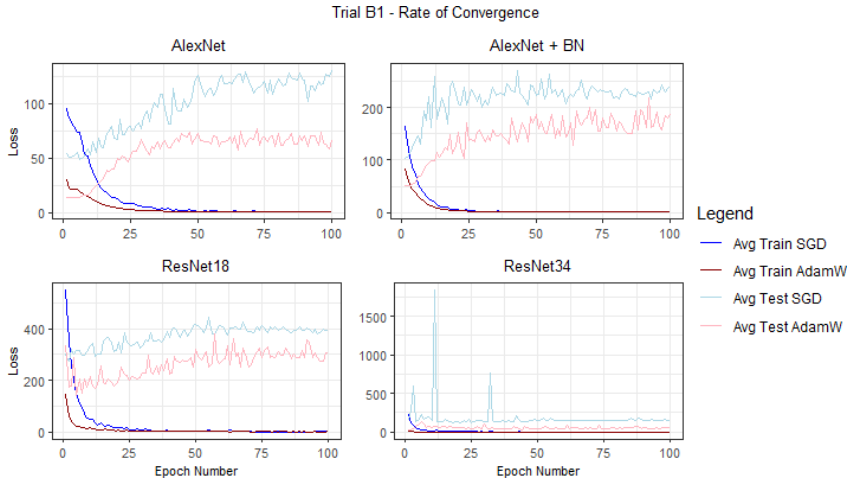
	SGD: Trial B2			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	**61.649**	60.019	56.472	53.883
H. AVR Acc ± SD	49.536 ± 2.92	**53.183** ± 1.45	52.445 ± 2.52	51.141 ± 1.38
H. Test F1	0.327	0.274	0.32	0.21
H. AVR F1 ± SD	0.226 ± 0.021	0.22 ± 0.022	0.206 ± 0.064	0.165 ± 0.02

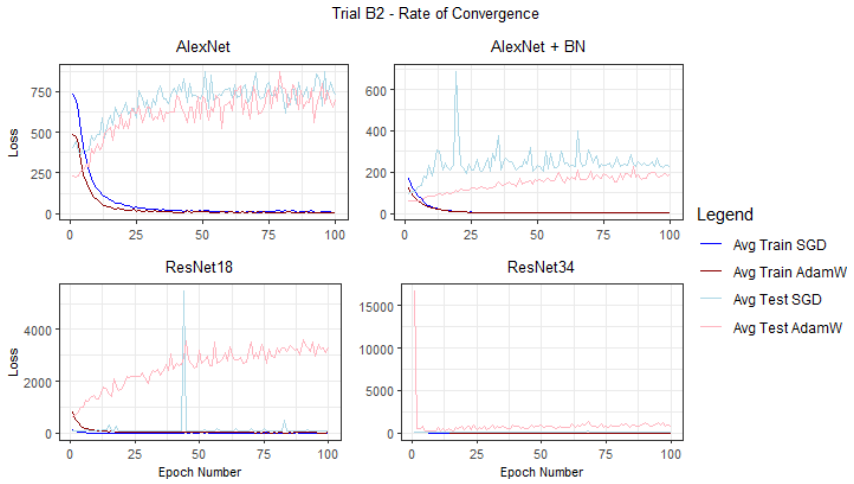
	AdamW: Trial B2			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	60.211	57.239	56.088	55.034
H. AVR Acc ± SD	52.387 ± 3.84	52.253 ± 0.76	51.697 ± 3.41	47.862 ± 5.14
H. Test F1	0.298	0.242	**0.35**	0.302
H. AVR F1 ± SD	**0.228** ± 0.019	0.206 ± 0.019	0.219 ± 0.08	0.181 ± 0.036

	SGD: Trial B3			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	57.373	56.221	57.719	57.028
H. AVR Acc ± SD	47.165 ± 10.83	48.456 ± 9.81	44.124 ± 7.44	43.456 ± 9.63
H. Test F1	0.321	0.353	0.249	0.352
H. AVR F1 ± SD	0.268 ± 0.036	0.269 ± 0.07	0.188 ± 0.032	0.201 ± 0.098

	AdamW: Trial B3			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	58.41	59.793	54.493	**61.29**
H. AVR Acc ± SD	49.7 ± 7.26	**54.147** ± 3.71	41.083 ± 7.5	46.336 ± 6.95
H. Test F1	**0.387**	0.232	0.285	0.265
H. AVR F1 ± SD	**0.281** ± 0.079	0.203 ± 0.021	0.157 ± 0.062	0.201 ± 0.043

Table 5.10: Results from using each CNN architecture with SGD/Adam for each trial. (Top) Trial B1, (Center) Trial B2, and (Bottom) Trial B3.


Trial B1 - Rate of Convergence
AlexNet
AlexNet + BN
ResNet18
ResNet34
Loss
Epoch Number
Legend
Avg Train SGD
Avg Train AdamW
Avg Test SGD
Avg Test AdamW


Trial B2 - Rate of Convergence
AlexNet
AlexNet + BN
ResNet18
ResNet34
Loss
Epoch Number
Legend
Avg Train SGD
Avg Train AdamW
Avg Test SGD
Avg Test AdamW

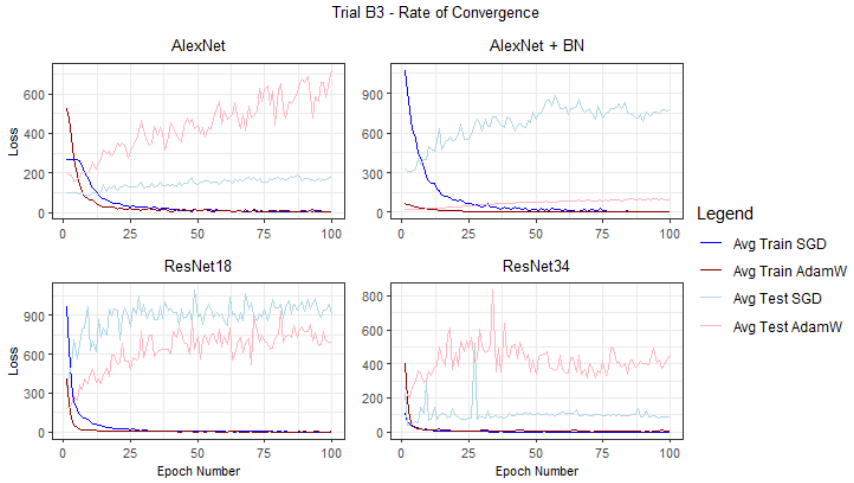


Figure 5.6: The average training and test loss per epoch of all 5 runs of each model per CNN architecture for each trial. (Top) Trial B1, (Center) Trial B2, and (Bottom) Trial B3.

From Table 5.10 (Top), we can see that results seemed to improve between CNN architectures, where ResNet34 using SGD had the highest test accuracy and F1-measure and average test accuracy and F1-measure results. For Table 5.10 (Center), on the other hand, results were not that different between CNN architectures, where ResNet34 had the worst results of the four. Also, Original AlexNet using AdamW had the highest average test F1-measure and ResNet18 using AdamW had the highest test F1-measure. For Table 5.10 (Bottom), results seemed to indicate that Original AlexNet and AlexNet + BN gave better results over ResNet, where Original AlexNet using Adam had the highest test F1-measure and average test F1-measure results.

Figure 5.6 highlights a common occurrence in all runs that there was always good performance for the training set but varying performance for the test set, where it would generally highlight an increasing average loss for the test set over time. This seems to be generally true for the AlexNet and AlexNet + BN architectures, where ResNet18 also shows occurrences of this; however, there were also runs where the average test loss only had a slight upward trend like for ResNet34. In most instances, the model failed to accurately classify images that had poses 3, 4, 5, and 6 of the test set, whereas poses 1, 2, and 7 would generally be the only poses correctly classified.

5.3.5: CNN Application Results - Resized Images

We will first review the results of Trial A in Section 5.3.5.1, where we will then go over the results from Trial B in Section 5.3.5.2. Appendix C2 presents all the hyperparameter results that we had for each trial in this section. The benchmarks we have set for Trials

A and B will be unchanged for this section.

5.3.5.1: Trial A Results

Table 5.11 presents the results for each CNN architecture considered for Trial A and Figure 5.7 presents the test loss per epoch of all 5 runs that we had done for each CNN architecture for both SGD and AdamW.

	SGD: Trial A			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	98.052	98.377	98.539	98.539
H. AVR Acc ± SD	97.013 ± 0.68	97.565 ± 0.64	97.695 ± 0.39	97.338 ± 1.1
H. Test F1	0.985	0.973	0.977	**0.988**
H. AVR F1 ± SD	0.963 ± 0.008	0.962 ± 0.006	0.964 ± 0.005	**0.967** ± 0.016

	AdamW: Trial A			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	**98.864**	97.565	98.539	98.701
H. AVR Acc ± SD	97.5 ± 0.66	96.429 ± 0.94	97.403 ± 0.28	**97.76** ± 0.35
H. Test F1	0.983	0.97	0.982	0.98
H. AVR F1 ± SD	0.965 ± 0.007	0.945 ± 0.017	0.964 ± 0.009	0.966 ± 0.015

Table 5.11: Results from using each CNN architecture with SGD/Adam for Trial A with resized images.

The results that we received were very similar to the results that we had obtained before from Section 5.3.4.1.2, and hence, much better than the benchmark method.

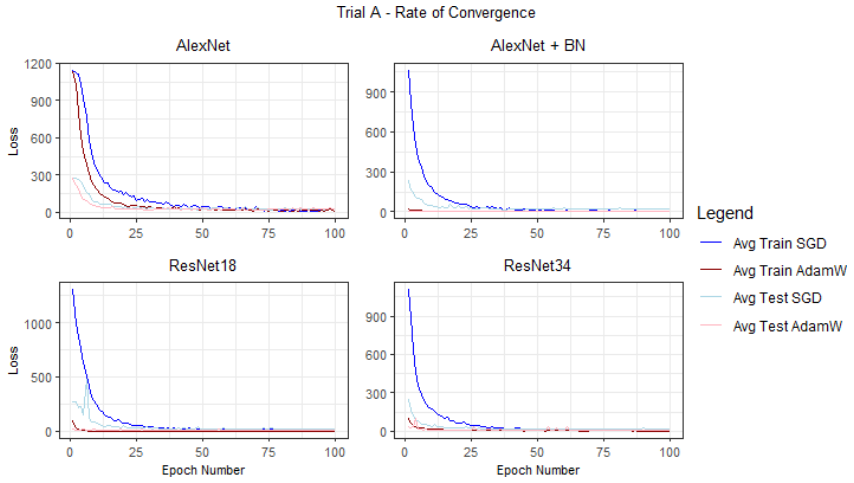


Figure 5.7: The average training and test loss per epoch of all 5 runs of each model per CNN architecture.

5.3.5.2: Trial B Results

Tables 5.12 (Top), (Center) and (Bottom) present our results for Trials B, whereas Figure 5.8 presents the average test loss per epoch of all runs per CNN architecture per trial.

	SGD: Trial B1			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	59.025	62.703	**63.73**	60.394
H. AVR Acc ± SD	52.318 ± 4.55	**60.462** ± 0.73	57.365 ± 1.77	54.867 ± 3.99
H. Test F1	0.303	0.28	0.358	0.27
H. AVR F1 ± SD	0.247 ± 0.01	0.268 ± 0.01	0.285 ± 0.043	0.232 ± 0.012

	AdamW: Trial B1			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	59.624	58.939	58.426	**63.73**
H. AVR Acc ± SD	56.527 ± 0.621	55.569 ± 0.71	53.567 ± 4.55	55.928 ± 1.78
H. Test F1	0.306	0.323	0.283	**0.425**
H. AVR F1 ± SD	0.257 ± 0.03	0.248 ± 0.055	0.227 ± 0.035	**0.296** ± 0.081

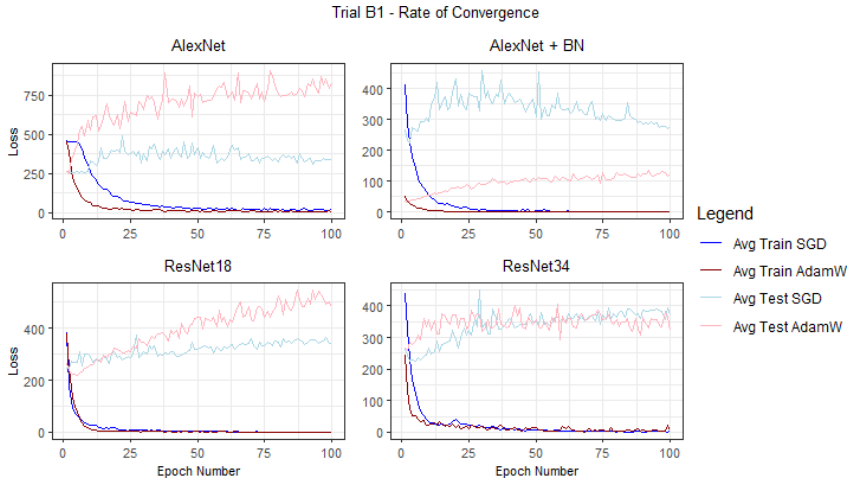
	SGD: Trial B2			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	58.869	58.389	57.526	56.088
H. AVR Acc ± SD	53.058 ± 3.41	53.058 ± 1.77	53.5 ± 2.61	53.212 ± 1.41
H. Test F1	0.292	0.302	**0.329**	0.251
H. AVR F1 ± SD	0.224 ± 0.042	0.215 ± 0.012	**0.231** ± 0.055	0.201 ± 0.035

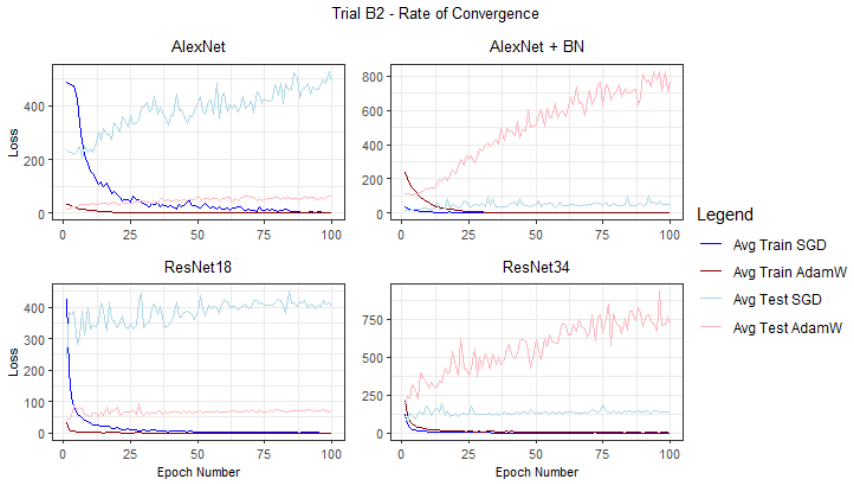
	AdamW: Trial B2			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	55.129	56.95	**59.444**	59.156
H. AVR Acc ± SD	47.344 ± 3.42	53.327 ± 1.8	**54.017** ± 3.16	52.215 ± 3.05
H. Test F1	0.234	0.237	0.245	0.297
H. AVR F1 ± SD	0.2 ± 0.029	0.199 ± 0.033	0.196 ± 0.03	0.203 ± 0.065

	SGD: Trial B3			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	50.806	**68.664**	63.479	58.41
H. AVR Acc ± SD	39.332 ± 5.65	53.433 ± 5.14	**56.037** ± 3.73	41.982 ± 8.46
H. Test F1	0.321	**0.476**	0.357	0.295
H. AVR F1 ± SD	0.245 ± 0.027	**0.3** ± 0.118	0.278 ± 0.052	0.197 ± 0.034

	AdamW: Trial B3			
	AlexNet	**AlexNet + BN**	**ResNet18**	**ResNet34**
H. Test Acc	65.438	62.788	60.484	63.594
H. AVR Acc ± SD	55.23 ± 4.38	53.71 ± 5.27	49.977 ± 7.12	51.267 ± 7.83
H. Test F1	0.384	0.276	0.355	0.321
H. AVR F1 ± SD	0.265 ± 0.061	0.194 ± 0.049	0.219 ± 0.017	0.24 ± 0.022

Table 5.12: Results from using each CNN architecture with SGD/Adam for each trial. (Top) Trial B1, (Center) Trial B2, and (Bottom) Trial B3.


Trial B1 - Rate of Convergence
AlexNet
AlexNet + BN
ResNet18
ResNet34
Loss
Epoch Number
Legend
Avg Train SGD
Avg Train AdamW
Avg Test SGD
Avg Test AdamW


Trial B2 - Rate of Convergence
AlexNet
AlexNet + BN
ResNet18
ResNet34
Loss
Epoch Number
Legend
Avg Train SGD
Avg Train AdamW
Avg Test SGD
Avg Test AdamW

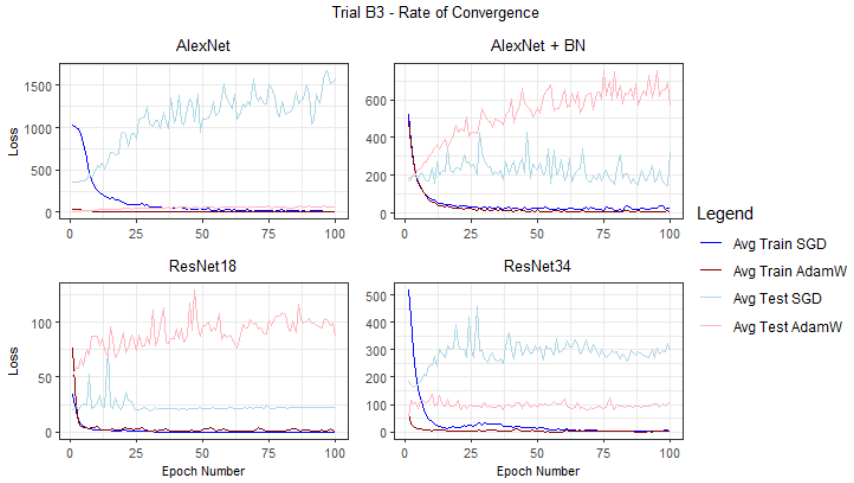


Figure 5.8: The average training and test loss per epoch of all 5 runs of each model per CNN architecture for each trial. (Top) Trial B1, (Center) Trial B2, and (Bottom) Trial B3.

From Trial B1, we can see that ResNet34 gave the best results of the four architectures again similar to before excluding the highest average test accuracy; however, there were no improvements in the result metrics when compared to the results in Section 5.3.4.2.2. This is similar to Trial B2, where results did not improve compared to the results in Section 5.3.4.2.2 excluding an increase in the average accuracy and F1-measure. For Trial B3, results seemed to be overall better, where AlexNet + BN using SGD gave the highest test F1-measure and average test F1-measure results as well as the highest test accuracy and ResNet18 using SGD had the highest average test accuracy.

5.4: TR Application

Section 5.4.1 will go over the details of the TR application and how the data was preprocessed. Section 5.4.2 will then go into the application results for each trial.

5.4.1: Application Detail

We use a multinomial TR model with tensor decomposition as defined in Chapter 4 and estimate parameters via block relaxation with minibatch gradient ascent and momentum. For tensor decomposition, we adopted CP decomposition with a fixed low rank of $Z = 3$. In practice, it will be rare that the true parameter tensor will follow a low-rank decomposition; however, a low-rank estimate will generally still be a fair enough approximation to the true parameter tensor (Zhou et al., 2013). We did not tune Z further to limit computational overhead, as the model is already demanding.

We used the log-likelihood as defined in the multinomial TR model, but also, to counter the data imbalance problem, we conducted separate tests where we added inverse frequency class weights in the log-likelihood. In some cases, we noticed that the training metrics were much higher than the testing metrics, possibly pointing towards the model overfitting to the training set. Thus, we also tested using L_1 regularization.

The hyperparameters we checked were the batch size, learning rate, L_1 regularization parameter, and momentum. To find optimal sets of hyperparameters, we used hold-out cross-validation with stratification with a similar procedure as was described in Section 5.3.2, where we test random hyperparameter values in the ranges given in Table 5.8 under SGD.

With these optimal sets of hyperparameters, we will generalize our results by running multiple runs, where each run is capped at 60 epochs of training due to time restraints. To check the validity of our results, we will similarly report the highest test accuracy and F1-measure as well as the highest average test accuracy and F1-measure among the epochs between models.

In terms of data preprocessing, similar to what was done in the CNN application, we have normalized the RGB images in the data from the range of [0, 255] to the range [0, 1], where they were then standardized. The initial factor matrices and bias values in the tensor decompositions at the start of training were initialized by taking values from a normal distribution with a mean of 0 and standard deviation of $\frac{1}{\sqrt{n}}$, where n is the sample size of the training set. We had initially tried sampling from a uniform distribution from -1 to 1 and a standard normal distribution; however, the larger initialized values led to larger log-likelihood values, which more easily led to exploding gradients early during training. As such, we used the above aiming to make the initialized values smaller, where we experienced fewer exploding gradients when using this. We took inspiration from initialization done on deep networks, although other weight initialization schemes could be more optimal (Glorot and Bengio, 2010).

5.4.2: TR Application Results

Appendix C3 presents all the hyperparameter results that we had for each trial in this section.

For Trials A and B, Table 5.13 presents the highest test accuracy and F1-measure and the highest average test accuracy and F1-measure between epochs including their standard deviation, whereas Figure 5.9 presents the average test loss per epoch of all runs per trial.

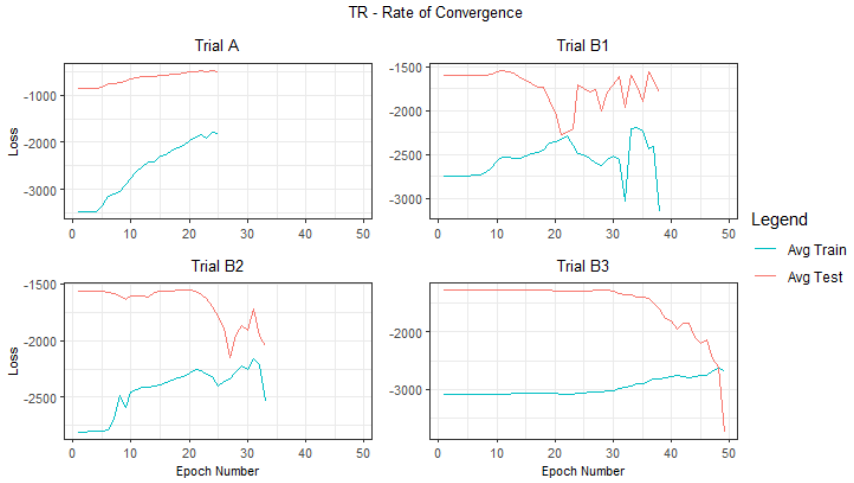


Figure 5.9: The average test loss per epoch of all 5 runs of each model.

	Trial A	**Trial B1**	**Trial B2**	**Trial B3**
H. Test Acc	79.058	57.998	55.609	54.954
H. AVR Acc ± SD	75.284 ± 5.288	55.432 ± 3.729	52.842 ± 2.144	44.124 ± 7.48
H. Test F1	0.695	0.215	0.217	0.228
H. AVR F1 ± SD	0.605 ± 0.064	0.185 ± 0.011	0.164 ± 0.044	0.175 ± 0.066

Table 5.13: TR results for Trials A and B.

In Figure 5.9, we ran each run up to 60 epochs; however, all runs did not reach the 60 epoch mark, where they diverged before this happened. The figures were capped to show the last epoch until the models diverged. The average test loss per epoch of the training and test sets generally mirror any decreases in the loss in an epoch, excluding Trial B3 although both training and test diverge eventually for them as well. From our results for Trial A and choosing the set of hyperparameters yielding the highest F1-measure, the model was able to correctly classify all 7 poses and this is reflected in the high F1 value. Similarly, for Trial B, at most only three poses were generally correctly classified, those being poses 1, 2 and 7 which is similar to what we saw in the CNN application.

5.5: Results Discussion

We will compare the results between the CNN and TR applications in this section. Note that the training and test sets used in the CNN models were not the same splits tested in the TR models. This means that it is not a fair comparison between the methods, and differences in performance could be due to variations in data rather than true differences in model effectiveness.

When comparing the performance metrics of the CNN application to the TR application, we can see that the results for Trial B from the multinomial TR model have yielded better results than some CNN architectures; however, the CNN application had the highest performance metrics overall including for the F1-measure. Since both applications were able to predict all poses in the case of Trial A, we can conclude that the objective for Trial A was achieved. Also, both the highest observed and the average performance metrics were better than the results of the dummy classifier given in Table 5.7. This confirms that each model was at least better than random guessing.

In the case of the CNN application for Trials A and B, from the hyperparameter tuning done using the training set, finding sets of hyperparameters which had very good performance metrics on both the training and validation sets was relatively common. This was for all architectures, showing that the depth of the layers we considered had not breached this critical depth threshold since we had not seen decreasing performance during training and validation. When using the sets of hyperparameters with the lowest validation loss on the test set, the results that we received all showed displays of overfitting, where we have very good performance metrics for the training set but poor performance metrics for the test set. However, this was not the case for the TR application, where poor performance metrics for the training set generally mirrored those of the validation and test sets. Thus, considering the objective for Trial B, the CNN application does not appear to have been as generalizable as the TR application. Table 5.14 highlights an approximate number of trainable parameters of the models considered in the CNN and TR applications, which shows the massive parameter difference between the models.

Model	**Number of Parameters**
Multinomial TR	10,134
ResNet18	11.4 million
ResNet34	21.5 million
AlexNet	62.3 million

Table 5.14: An approximate number of trainable parameters of each model from the CNN and TR applications.

We conducted separate tests for both the CNN and TR application where we added inverse frequency class weights in the log-likelihood. In general, this helped the model classify multiple labels including labels 4, 5 and 6 at the cost of lower accuracy due to fewer correct predictions. However, the F1-measure results were not that much different to the results that we had where we had not added these class weights in terms of improvements.

From the results of the CNN application, it is interesting to note that Trial B2 generally had worse results than Trials B1 and B3. This could be because there were two male dancers for the training set versus a female dancer for the test set, potentially pointing towards the models from Trial B2 being less generalizable against gender.

The results of the AlexNet and AlexNet + BN architectures do not indicate that adding batch normalization layers always improved performance. There were situations where one outperformed the other but both had given good results in certain trials. Although we mentioned that AlexNets lack of depth could affect its ability to grasp complex features, it still gave very good results comparable (and sometimes even better) to the results from ResNet.

In our experiments with AdamW using fixed values of β_1 and β_2 and unfixed ones, we had a similar observation to what was mentioned in Appendix A3, where smaller values of β_2 tended to lead to poorer results although not always. In Appendix C, we share the top hyperparameter search results that we had, and there were situations where small values of β_2 outperformed other results such as in Section C1.3.2.

6

Conclusion

In this dissertation, we aimed to explore material on CNNs and TR for image classification, particularly for choreographic modeling where we proceeded to do an application using both. This chapter briefly summarizes our work, limitations, and suggests potential suggestions for future work.

6.1: Summary

The primary aim of this dissertation was to explore and evaluate methods for choreographic modeling applied to a dance dataset, particularly CNNs and TR. After covering these methods in detail, we proceeded to use them in an application on the dance dataset, where we then compared their performance together and against a dummy classifier.

The methodological choices were informed by the strengths and limitations of each approach in relation to the research goals. In Chapter 3, we covered CNNs because they directly model spatial structure in image inputs via tensors and have a strong empirical track record for image-based classification tasks. We utilized adaptive optimizers and regularization techniques to reduce training time and the risk of overfitting. In Chapter 4, we covered TR and how it can be developed within a GLM framework to provide an alternative, which, when combined with tensor decomposition, requires far fewer parameters and yields estimates with interpretable statistical properties. We also discussed its extension to the multinomial setting by Cao et al. (2022).

Chapter 5 presented the dance dataset (three dancers, three dances with a total of seven poses) and two trials to build with the goal of building CNN and TR models to classify all poses in the dances. Trial A trained and tested using all dancer data mixed randomly, while Trial B trained on two dancers and tested on the held-out third dancer

to probe model generalization. We then moved on to describe the application setup and the presentation of results for each trial. There were several key outcomes, notably that the CNN models achieved the highest overall classification metrics in Trials A and B, showcasing that deep CNNs yield superior raw accuracy in these settings. However, the TR model outperformed some CNN variants on specific metrics in Trial B, suggesting that a simpler model can be more generalizable than a huge neural network. All models had better results than the dummy classifier, meaning that they were at least better than random guessing. The findings underscore that evaluating models by reporting both performance and model complexity provides better guidance for model selection.

The motivation behind this research concerns the preservation of folk dances to safeguard ICH, where choreographic modeling serves a vital role. The CNN and TR discussion developed in this dissertation has practical implications for any domain that needs pose-aware image understanding. For example, in cultural-heritage archiving systems, both methods can help detection and indexing of human poses in photographs, paintings and performance recordings. This would effectively help streamline content-based retrieval and guided restoration for that domain.

6.2: Limitations of Study

For the dance dataset, both the CNN and TR applications were computationally intensive given our hardware and time constraints. To maintain feasibility, we reduced the scope of experimentation by limiting runs from 10 to 5, capping training epochs, and prioritizing more time-efficient configurations. Certain methods, such as Tucker decomposition in the TR application, which we had initially planned to work on as well, were excluded for these reasons. These adjustments reflect a necessary trade-off between thoroughness and practical feasibility.

PyTorch provides inherent GPU acceleration for deep learning models, including CNNs. However, at the time, it lacked support for our AMD GPU on Windows. While we initially experimented with designing a custom CNN architecture in PyTorch, the extensive hyperparameter search required proved computationally prohibitive, leading us to adopt established architectures like AlexNet and ResNet from the literature instead.

Although for the CNN application we made use of efficient tools from PyTorch, to our knowledge, there was no known package which implements multinomial TR with CP decomposition. As such, the code that we used in the TR application was created manually, which inherently will be limited in its computational efficiency, even with our attempts to optimize it. For example, in the TR application, we generally ran

runs with higher batch sizes over smaller ones, because the latter took a lot more time than expected to run, when it should be the opposite. Consequently, these limitations restricted the scope of our exploration in TR; with more efficient implementations, it is plausible that we could have achieved more results surpassing those of the CNN application.

6.3: Suggestions for Future Study

Data augmentation involves techniques like random cropping or rotating images to expand the training set. It is particularly useful when data is limited, as it can improve generalization (Wang et al., 2024). While we could have explored this in Trial B, Engstrom et al. (2017) note that augmentation enhances translation invariance only for inputs similar to training images. Azulay and Weiss (2019) highlight that it does not generalize well to all inputs. Moreover, given the variability in dancer body types and our imbalanced label distribution, augmentation may have had limited benefit. Zhou et al. (2022) suggest that simple transformations like rotations often help majority classes more than minority ones, which is relevant to our dataset.

The CNNs used in our application were originally trained on large online datasets. Using pre-trained weights from these models could improve training speed and performance, especially with limited data (Bressem et al., 2021). Pre-trained weights capture general low-level features in early layers, while deeper layers are more task-specific (Rehman et al., 2021). Therefore, in our application, using pre-trained CNNs with only the final layers retrained could be a promising approach.

Image noise, such as irrelevant backgrounds, can reduce classification accuracy (Rajnoha et al., 2018). CNNs may incorporate background features instead of focusing on the subject. Background modeling addresses this by classifying pixels as foreground or background (Babaee et al., 2018). Rallis et al. (2019) applied this in a dance recognition task and saw performance gains, suggesting it could be valuable for future work.

Recent research explores integrating TR into CNNs (Cao and Rabusseau, 2017; Kossaifi et al., 2020). Traditional CNNs use fully connected layers, which flatten data and lose structural information. Tensor Regression Layers (TRLs) preserve this structure by applying tensor decomposition directly to layer inputs. TRLs reduce parameter counts with minimal accuracy loss and can be particularly useful in overfitting scenarios. Cao and Rabusseau (2017) provide guidance for selecting decomposition hyperparameters when TRLs are used before the softmax layer. Incorporating TRLs into our CNNs may help mitigate overfitting in Trial B.

References

Ahmet Alacaoglu, Yura Malitsky, Panayotis Mertikopoulos, and Volkan Cevher. A new regret analysis for adam-type algorithms. *arXiv (Cornell University)*, Mar 21 2020. doi: 10.48550/arxiv.2003.09729.

Saleh Albelwi and Ausif Mahmood. A framework for designing the architectures of deep convolutional neural networks. *Entropy (Basel, Switzerland)*, 19(6):242, Jun 1 2017. doi: 10.3390/e19060242.

Antonio Alguacil, Wagner Gonçalves Pinto, Michael Bauerheim, Marc C. Jacob, and Stéphane Moreau. *Effects of Boundary Conditions in Fully Convolutional Networks for Learning Spatio-Temporal Dynamics*, volume 12979 of *Machine Learning and Knowledge Discovery in Databases. Applied Data Science Track*, pages 102–117. Springer International Publishing AG, Switzerland, 2021. ISBN 9783030865160. doi: 10.1007/978-3-030-86517-7_7. URL `http://ebookcentral.proquest.com/lib/SITE_ID/reader.action?docID=6724616&ppg=134`.

Igi Ardiyanto, Junji Satake, and Jun Miura. Autonomous monitoring framework with fallen person pose estimation and vital sign detection. pages 1–6. IEEE, 2014. doi: 10.1109/ICITEED.2014.7007897. URL `https://ieeexplore.ieee.org/document/7007897`.

Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. Technical report, Cornell University Library, arXiv.org, Feb 28 2018. URL `https://search.proquest.com/docview/2071665270`. ID: RefID:440-arora2018understanding; M1: Report.

Woody Austin, Grey Ballard, and Tamara G. Kolda. Parallel tensor compression for large-scale scientific data. pages 912–922. IEEE, 2016. ISBN 1530-2075. doi: 10.1109/IPDPS.2016.67. URL `https://ieeexplore.ieee.org/document/7516088`.

Aharon Azulay and Yair Weiss. Why do deep convolutional networks generalize so poorly to small image transformations? Technical report, Cornell University Library, arXiv.org, Dec 31 2019. URL `https://www.proquest.com/docview/2073569695`.

Mohammadreza Babaee, Duc Tung Dinh, and Gerhard Rigoll. A deep convolutional neural network for video sequence background subtraction. *Pattern recognition*, 76:635–649, Apr 2018. doi: 10.1016/j.patcog.2017.09.040.

Francis Bach. Breaking the curse of dimensionality with convex neural networks. *Journal of machine learning research*, 18(19):1–53, Dec 26 2014.

Paul Bachmann. *Analytische Zahlentheorie*. 1894.

Nikolaos Bakalos, Ioannis Rallis, Nikolaos Doulamis, Anastasios Doulamis, Eftychios Protopapadakis, and Athanasios Voulodimos. Choreographic pose identification using convolutional neural networks. pages 1–7, Piscataway, 2019. IEEE. doi: 10.1109/VS-Games.2019.8864522. URL `https://ieeexplore.ieee.org/document/8864522`.

Chenglong Bao, Qianxiao Li, Zuowei Shen, Cheng Tai, Lei Wu, and Xueshuang Xiang. Approximation analysis of convolutional neural networks. *East Asian journal on applied mathematics*, 13(3):524–549, Aug 1 2023. doi: 10.4208/eajam.2022-270.070123.

R. Battiti. Using mutual information for selecting features in supervised neural net learning. *IEEE Transactions on Neural Networks*, 5(4):537–550, Jul 1 1994. doi: 10.1109/72.298224.

Richard E. Bellman. *Adaptive Control Processes*, volume 2045. Princeton University Press, United States, 1 edition, 1961. ISBN 1400874661. doi: 10.1515/9781400874668. URL `http://digital.casalini.it/9781400874668`.

James Bergstra and Yoshua Bengi. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(10):281–305, 2012.

Ashwin Bhandare, Maithili Bhide, Pranav Gokhale, and Rohan Chandavarkar. Applications of convolutional neural networks. *International Journal of Computer Science and Information Technologies*, 7(5):2206–2215, 2016.

Keno K. Bressem, Lisa C. Adams, Robert A. Gaudin, Daniel Tröltzsch, Bernd Hamm, Marcus R. Makowski, Chan-Yong Schüle, Janis L. Vahldiek, and Stefan M. Niehues. Highly accurate classification of chest radiographic reports using a deep learning natural language model pre-trained on 3.8 million text reports. *Bioinformatics*, 36(21):5255–5261, Jan 29 2021. doi: 10.1093/bioinformatics/btaa668.

R Bro. Multi-way analysis in the food industry. models, algorithms and applications. *PhD thesis, Department of Analytical Chemistry, University of Amsterdam*, 1998.

Rasmus Bro and Henk A. L. Kiers. A new efficient method for determining the number of components in parafac models. *Journal of Chemometrics*, 17(5):274–286, May 2003. doi: 10.1002/cem.801.

Xingwei Cao and Guillaume Rabusseau. Tensor regression networks with various low-rank tensor approximations. *arXiv (Cornell University)*, Dec 27 2017. doi: 10.48550/arxiv.1712.09520.

Xuan Cao, Fang Yang, Jingyi Zheng, Xiao Wang, and Qingling Huang. Aberrant structure mri in parkinson's disease and comorbidity with depression based on multinomial tensor regression analysis. *Journal of personalized medicine*, 12(1):89, Jan 11 2022. doi: 10.3390/jpm12010089.

R. Caruana, S. Lawrence, and L. Giles. Overfitting in neural nets: Backpropagation conjugate gradient, and early stopping. *International Conference on Neural Information Processing Systems*, 2000.

Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the devil in the details: Delving deep into convolutional nets. *Proceedings of the British Machine Vision Conference 2014*, pages 6.1–612, 2014. doi: 10.5244/c.28.6.

Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. Suvisoft, 2006. URL `https://inria.hal.science/inria-00112631`.

Nadav Cohen and Amnon Shashua. Convolutional rectifier networks as generalized tensor decompositions. *arXiv (Cornell University)*, Mar 1 2016. doi: 10.48550/arxiv.1603.00162.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, Dec 1989. doi: 10.1007/bf02551274.

Jan de Leeuw. Block-relaxation algorithms in statistics. *Studies in Classification, Data Analysis, and Knowledge Organization*, pages 308–324, 1994. doi: 10.1007/978-3-642-46808-7_28.

Jan de Leeuw, Forrest W. Young, and Yoshio Takane. Additive structure in qualitative data: An alternating least squares method with optimal scaling features. *Psychometrika*, 41(4):471–503, Dec 1976. doi: 10.1007/BF02296971.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

Mohamed Elgendy. *Deep learning for vision systems*. Manning, Shelter Island, 2020. ISBN 1617296198.

Omar Elnaggar, Frans Coenen, and Paolo Paoletti. *In-Bed Human Pose Classification Using Sparse Inertial Signals*, volume 12498 of *Artificial Intelligence XXXVII*, pages 331–344. Springer International Publishing AG, Switzerland, 2020. ISBN 9783030637989. doi: 10.1007/978-3-030-63799-6_25. URL `http://ebookcentral.proquest.com/lib/SITE_ID/reader.action?docID=6421885&ppg=333`.

Logan Engstrom, Dimitris Tsipras, Ludwig Schmidt, and Aleksander Madry. A rotation and a translation suffice: Fooling cnns with simple transformations. 2017.

Nicolaas (Klaas) M. Faber, Rasmus Bro, and Philip K. Hopke. Recent developments in candecomp/parafac algorithms: a critical review. *Chemometrics and intelligent laboratory systems*, 65(1):119–137, 2003. doi: 10.1016/S0169-7439(02)00089-8.

Omobayode Fagbohungbe and Lijun Qian. Benchmarking inference performance of deep learning models on analog devices. pages 1–9, Piscataway, 2021. IEEE. doi: 10.1109/IJCNN52387.2021.9534143. URL `https://ieeexplore.ieee.org/document/9534143`.

Eleni Filippidou and Irini Gialiti. Differences between the "first" and the "second" existence of dance in the greek island of crete. the example of the syrtos dance. *International Journal of Education and Social Science Research*, 5(3):215–229, 2022. doi: 10.37500/IJESSR.2022.5315.

K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, Apr 1980. doi: 10.1007/bf00344251.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *International Conference on Artificial Intelligence and Statistics.*, 2010.

Gene H. Golub and Charles F. Van Loan. *Matrix computations*. North Oxford Acad. Publ, Oxford, repr. edition, 1986. ISBN 9780946536009.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks. *arXiv (Cornell University)*, Dec 20 2013. doi: 10.48550/arXiv.1312.6082.

Remi Gribonval and Morten Nielsen. Nonlinear approximation with dictionaries i. direct estimates. *The Journal of fourier analysis and applications*, 10(1):51–71, Jan 1 2004. doi: 10.1007/s00041-004-8003-5.

Lorena Gril, Philipp Wedenig, Chris Torkar, and Ulrike Kleb. A tensor based regression approach for human motion prediction. *arXiv (Cornell University)*, Feb 4 2022. doi: 10.48550/arxiv.2202.03179.

Bruna Maria Vittoria Guerra, Stefano Ramat, Giorgio Beltrami, and Micaela Schmid. Automatic pose recognition for monitoring dangerous situations in ambient-assisted living. *Frontiers in bioengineering and biotechnology*, 8:415, May 14 2020. doi: 10.3389/fbioe.2020.00415.

Weiwei Guo, I. Kotsia, and I. Patras. Tensor learning for regression. *IEEE transactions on image processing*, 21(2):816–827, Feb 1 2012. doi: 10.1109/TIP.2011.2165291.

Venkatesan Guruswami and Ravi Kannan. Computer science theory for the information age. 2012.

Shizhong Han, Zibo Meng, Zhiyuan Li, James O'Reilly, Jie Cai, Xiaofeng Wang, and Yan Tong. Optimizing filter size in convolutional neural networks for facial action unit recognition. pages 5070–5078. IEEE, 2018. doi: 10.1109/CVPR.2018.00532. URL `https://ieeexplore.ieee.org/document/8578630`.

S. Hanson and L. Pratt. Comparing biases for minimal network construction with back-propagation. *In Proceedings of the 1st International Conference on Neural Information Processing Systems*, pages 177–185, 1988.

R. A Harshman. Foundations of the parafac procedure: Models and conditions for an "explanatory" multimode factor analysis. *UCLA Working Papers in Phonetics*, 16:1–84, 1970.

Mahdi Hashemi. Enlarging smaller images before inputting into convolutional neural network: zero-padding vs. interpolation. *Journal of Big Data*, 6(1):1–13, Nov 14 2019. doi: 10.1186/s40537-019-0263-7.

Johan Hastad. Tensor rank is np-complete. *Journal of algorithms*, 11(4):644–654, 1990. doi: 10.1016/0196-6774(90)90014-6.

Elad Hazan. *Introduction to Online Convex Optimization*, volume 2. now, Boston - Delft, 2016. ISBN 9781680831702. doi: 10.1561/2400000013. URL `https://ieeexplore.ieee.org/servlet/opac?bknumber=8186825`.

Juncai He, Lin Li, and Jinchao Xu. Approximation properties of deep relu cnns. *Research in the mathematical sciences*, 9(3), Sep 1 2022. doi: 10.1007/s40687-022-00336-0.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv (Cornell University)*, Dec 10 2015. doi: 10.48550/ARXIV.1512.03385.

Timothy J. Healy. Convolution revisited. *IEEE Spectrum*, 6:87–93, 1969.

Geoffrey E. Hinton. *A Practical Guide to Training Restricted Boltzmann Machines*, pages 599–619. Neural Networks: Tricks of the Trade. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 9783642352881. doi: 10.1007/978-3-642-35289-8_32. URL `http://link.springer.com/10.1007/978-3-642-35289-8_32`.

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv (Cornell University)*, Jul 3 2012. doi: 10.48550/arXiv.1207.0580.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. *arXiv (Cornell University)*, May 24 2017. doi: 10.48550/arXiv.1705.08741.

D. H. Hubel and T. N. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of Physiology*, 160(1):106–154, Jan 1962. doi: 10.1113/jphysiol.1962.sp006837.

Mahbub Hussain, Jordan J. Bird, and Diego R. Faria. *A Study on CNN Transfer Learning for Image Classification*, volume 840 of *Advances in Computational Intelligence Systems*, pages 191–202. Springer International Publishing AG, Switzerland, 2018. ISBN 9783319979816. doi: 10.1007/978-3-319-97982-3_16. URL `http://ebookcentral.proquest.com/lib/SITE_ID/reader.action?docID=5491599&ppg=199`.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv (Cornell University)*, Feb 10 2015. doi: 10.48550/arXiv.1502.03167.

Md Amirul Islam, Matthew Kowal, Sen Jia, Konstantinos G. Derpanis, and Neil D. B. Bruce. Position, padding and predictions: A deeper look at position information in cnns. *arXiv (Cornell University)*, Jan 28 2021. doi: 10.48550/arxiv.2101.12322.

David Jacobs. Correlation and convolution. *Class Notes for CMSC 426*, 2005.

A. Kapteyn, H. NEUDECKER, and T. WANSBEEK. An approach to n-mode components analysis. *Psychometrika*, 51(2):269–275, Jun 1 1986. doi: 10.1007/BF02293984.

Z. Karnin, T. Koren, and O. Somekh. Almost optimal exploration in multi-armed bandits. *ICML*, 2013.

Alexander Khvostikov, Karim Aderghal, Jenny Benois-Pineau, Andrey Krylov, and Gwenaelle Catheline. 3d cnn-based classification using smri and md-dti images for alzheimer disease studies. *arXiv (Cornell University)*, Jan 18 2018. doi: 10.48550/ARXIV.1801.05968.

Henk A. L. Kiers. Towards a standardized notation and terminology in multiway analysis. *Journal of Chemometrics*, 14(3):105–122, May 2000. doi: 10.1002/1099-128X(200005/06)14:3<105::AID-CEM582>3.0.CO;2-I.

Henk A. L. Kiers and Albert der Kinderen. A fast method for choosing the numbers of components in tucker3 analysis. *British journal of mathematical statistical psychology*, 56(1):119–125, May 2003. doi: 10.1348/000711003321645386.

Misha E. Kilmer, Karen Braman, Ning Hao, and Randy C. Hoover. Third-order tensors as operators on matrices: A theoretical and computational framework with applications in imaging. *SIAM journal on matrix analysis and applications*, 34(1):148–172, Jan 1 2013. doi: 10.1137/110837711.

Bo-Kyeong Kim, Hwaran Lee, Jihyeon Roh, and Soo-Young Lee. Hierarchical committee of deep cnns with exponentially-weighted decision fusion for static facial expression recognition. ACM Conferences, pages 427–434, New York, NY, USA, 2015. ACM. doi: 10.1145/2818346.2830590.

Ho-Chan Kim and Min-Jae Kang. Comparison of hyper-parameter optimization methods for deep neural networks. , 24(4):969–974, 2020.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. Technical report, Cornell University Library, arXiv.org, Jan 30 2017. URL `https://www.proquest.com/docview/2075396516`.

M. Kohler and S. Langer. Statistical theory for image classification using deep convolutional neural networks with cross-entropy loss, 2020. URL `https://arxiv.org/abs/2011.13602`.

Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, Sep 1 2009. doi: 10.1137/07070111X.

Tamara Gibson Kolda. Multilinear operators for higher-order decompositions. Technical report, Apr 1 2006. URL `https://www.osti.gov/servlets/purl/923081`.

Udin Komarudin, Azuan Ahmad, Widyatama, Iman Hafifi Zainal Abiddin, and Madihah Mohd Saudi. Drone monitoring system to detect human posture using deep learning algorithm. *Turkish journal of computer and mathematics education*, 12(8):1824–1833, Apr 24 2021.

Risi Kondor and Shubhendu Trivedi. On the generalization of equivariance and convolution in neural networks to the action of compact groups. *arXiv (Cornell University)*, Feb 10 2018. doi: 10.48550/arxiv.1802.03690.

J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic. Tensorly: tensor learning in python. Feb 1 2019.

Jean Kossaifi, Zachary C. Lipton, Arinbjorn Kolbeinsson, Tommaso Furlanello Aran Khanna, and Anima Anandkumar. Tensor regression networks. *Journal of Machine Learning Research*, 21(123):121, 2020.

A. Krizhevsky, Sutskever I., and Hinton G. *ImageNet Classification with Deep Convolutional Neural Networks*. 2012.

J. B Kruskal. Rank, decomposition, and uniqueness for 3-way and n-way arrays. *Multiway Data Analysis*, pages 7–18, 1989.

Joseph B. Kruskal. Three-way arrays: rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics. *Linear algebra and its applications*, 18(2):95–138, 1977. doi: 10.1016/0024-3795(77)90069-6.

Nicola Landro, Ignazio Gallo, and Riccardo La Grassa. Combining optimization methods using an adaptive meta optimizer. *Algorithms*, 14(6):186, Jun 1 2021. doi: 10.3390/a14060186.

Kenneth Lange. *Numerical Analysis for Statisticians*. Springer New York, New York, NY, 2nd ed edition, 2010. ISBN 1441959459. doi: 10.1007/978-1-4419-5945-4. URL `https://library.biblioboard.com/viewer/03f6c1ff-bee2-11ea-9068-0a28bb48d135`.

Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. ACM Other Conferences, pages 473–480, New York, NY, USA, 2007. ACM. doi: 10.1145/1273496.1273556. URL `https://search.proquest.com/docview/31368407`.

Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A multilinear singular value decomposition. *SIAM journal on matrix analysis and applications*, 21(4):1253–1278, a 2000a. doi: 10.1137/S0895479896305696.

Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. On the best rank-1 and rank-(r1 ,r2 ,. . .,rn) approximation of higher-order tensors. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1324–1342, b 2000b. doi: 10.1137/s0895479898346995.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1 1998. doi: 10.1109/5.726791.

Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Handwritten digit recognition with a back-propagation network. *Neural Information Processing Systems*, 1990.

Hung Yi Lee, Mostafa Reisi Gahrooei, Hongcheng Liu, and Massimo Pacella. Robust tensor-on-tensor regression for multidimensional data modeling. *IIE transactions*, ahead-of-print(ahead-of-print):1–11, Jan 2 2024. doi: 10.1080/24725854.2023.2183440.

Bing Li, Min Kyung Kim, and Naomi Altman. On dimension folding of matrix- or array-valued statistical objects. *The Annals of statistics*, 38(2):1094–1121, Apr 1 2010. doi: 10.1214/09-AOS737.

Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. *arXiv (Cornell University)*, Oct 13 2018a. doi: 10.48550/arxiv.1810.05934.

Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J. Smola. Efficient mini-batch training for stochastic optimization. ACM Conferences, pages 661–670, New York, NY, USA, 2014. ACM. doi: 10.1145/2623330.2623612.

Wanyi Li, Jie Tan, and Yingyin Fan. Human pose classification based on pose long short-term memory. *Modern Intelligent Times*, 2:3, Oct 15 2024. doi: 10.53964/mit.2024003.

Xiaomeng Li, Lequan Yu, Chi-Wing Fu, and Pheng-Ann Heng. *Deeply Supervised Rotation Equivariant Network for Lesion Segmentation in Dermoscopy Images*, volume 11041 of *OR 2.0 Context-Aware Operating Theaters, Computer Assisted Robotic Endoscopy, Clinical Image-Based Procedures, and Skin Image Analysis*, pages 235–243. Springer International Publishing AG, Switzerland, 2018b. ISBN 9783030012007.

doi: 10.1007/978-3-030-01201-4_25. URL http://ebookcentral.proquest.com/lib/SITE_ID/reader.action?docID=6284371&ppg=251.

Xiaoshan Li. *Tensor Based Statistical Models with Applications in Neuroimaging Data Analysis*. PhD thesis, Jan 1 2014. URL https://www.proquest.com/docview/1642719309.

Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv (Cornell University)*, Jul 13 2018. doi: 10.48550/arXiv.1807.05118.

Shao-Bo Lin, Kaidong Wang, Yao Wang, and Ding-Xuan Zhou. Universal consistency of deep convolutional neural networks. *arXiv (Cornell University)*, Jun 23 2021. doi: 10.48550/arxiv.2106.12498.

Siqi Liu, Xiaoyu Shi, and Qifeng Liao. Rank-adaptive tensor completion based on tucker decomposition. *Entropy (Basel, Switzerland)*, 25(2):225, Jan 24 2023. doi: 10.3390/e25020225.

Charles F. Van Loan. The ubiquitous kronecker product. *Journal of computational and applied mathematics*, 123(1):85–100, Nov 1 2000. doi: 10.1016/S0377-0427(00)00393-9.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv (Cornell University)*, Nov 14 2017. doi: 10.48550/ARXIV.1711.05101.

H. Lu, K. N. Plataniotis, and A. N. Venetsanopoulos. Mpca: Multilinear principal component analysis of tensor objects. *IEEE transaction on neural networks and learning systems*, 19(1):18–39, Jan 2008. doi: 10.1109/TNN.2007.901277.

Jun Lu. Matrix decomposition and applications. *arXiv (Cornell University)*, Jan 1 2022. doi: 10.48550/arxiv.2201.00145.

Jerubbaal Luke, Rajkumar Joseph, and Mahesh Balaji. Impact of image size on accuracy and generalization of convolutional neural networks. *International Journal of Research and Analytical Reviews (IJRAR)*, 6(1):70–80, 2019.

Liangchen Luo, Yuanhao Xiong, Yan Liu, and Xu Sun. Adaptive gradient methods with dynamic bound of learning rate. *arXiv (Cornell University)*, Feb 26 2019. doi: 10.48550/arxiv.1902.09843.

Konstantinos Makantasis, Anastasios Doulamis, Nikolaos Doulamis, and Antonis Nikitakis. Tensor-based classifiers for hyperspectral data analysis. Sep 24 2017. doi: 10.48550/arxiv.1709.08164.

Konstantinos Makantasis, Anastasios Doulamis, Nikolaos Doulamis, Antonis Nikitakis, and Athanasios Voulodimos. Tensor-based nonlinear classifier for high-order data analysis. pages 2221–2225. IEEE, 2018. doi: 10.1109/ICASSP.2018.8461418. URL https://ieeexplore.ieee.org/document/8461418.

Konstantinos Makantasis, Athanasios Voulodimos, Anastasios Doulamis, Nikolaos Bakalos, and Nikolaos Doulamis. Space-time domain tensor neural networks: An application on human pose classification. pages 4688–4695, Piscataway, 2021. IEEE. doi: 10.1109/ICPR48806.2021.9412482. URL https://ieeexplore.ieee.org/document/9412482.

Alessio Martino, Alessandro Giuliani, and Antonello Rizzi. (hyper)graph embedding and classification via simplicial complexes. *Algorithms*, 12(11):223, Nov 1 2019. doi: 10.3390/a12110223.

P. McCullagh and J. A. Nelder. Generalized linear models. *Monographs on Statistics and Applied Probability.*, 1983.

Reza Moradi, Reza Berangi, and Behrouz Minaei. A survey of regularization strategies for deep models. *The Artificial intelligence review*, 53(6):3947–3986, Aug 1 2020. doi: 10.1007/s10462-019-09784-7.

Coenraad Mouton, Johannes C. Myburgh, and Marelie H. Davel. Stride and translation invariance in cnns. Technical report, Cornell University Library, arXiv.org, Mar 18 2021. URL `https://www.proquest.com/docview/2503044595`.

J. A. Nelder and R. W. M. Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society: Series A (General)*, 135(3):370–384, 1972.

Eshaan Nichani, Adityanarayanan Radhakrishnan, and Caroline Uhler. Do deeper convolutional networks perform better? 2020. doi: 10.48550/arXiv.2010.09610.

I. V. Oseledets. Tensor-train decomposition. *SIAM journal on scientific computing*, 33(5):2295–2317, Jan 1 2011. doi: 10.1137/090752286.

Alexander M. Ostrowski. *Solution of equations and systems of equations*, volume 9. Acad.Pr, New York, 1960.

K. Padmavathi and K. Thangadurai. Implementation of rgb and grayscale images in plant leaves disease detection – comparative study. *Indian journal of science and technology*, 9(6), Feb 5 2016. doi: 10.17485/ijst/2016/v9i6/77739.

Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, page 1532, 2014. doi: 10.3115/v1/d14-1162.

Philipp Petersen and Felix Voigtlaender. Equivalence of approximation by convolutional neural networks and fully-connected networks. Technical report, Cornell University Library, arXiv.org, Jan 28 2021. URL `https://www.proquest.com/docview/2099748270`.

B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, Jan 1964. doi: 10.1016/0041-5553(64)90137-5.

Nikolay Ponomarenko, Lina Jin, Oleg Ieremeiev, Vladimir Lukin, Karen Egiazarian, Jaakko Astola, Benoit Vozel, Kacem Chehdi, Marco Carli, Federica Battisti, and C. C Jay Kuo. Image database tid2013: Peculiarities, results and perspectives. *Signal processing. Image communication*, 30:57–77, Jan 1 2015. doi: 10.1016/j.image.2014.10.009.

M. Arulselvi B. Gnana Priya. Deep learning for human pose classification using multi view dataset. *International Journal of Recent Technology and Engineering (IJRTE)*, 2019.

Stephan Rabanser, Oleksandr Shchur, and Stephan Günnemann. Introduction to tensor decompositions and their applications in machine learning. *arXiv (Cornell University)*, Nov 29 2017. doi: 10.48550/arxiv.1711.10781.

Sheikh Md Razibul Hasan Raj, Sultana Jahan Mukta, Tapan Kumar Godder, and Md Zahidul Islam. *An Approach for Multi-human Pose Recognition and Classification Using Multiclass SVM*, volume 1324 of *Intelligent Computing and Optimization*, pages 922–937. Springer International Publishing AG, Switzerland, 2021. ISBN 9783030681531. doi: 10.1007/978-3-030-68154-8_78. URL http://ebookcentral.proquest.com/lib/SITE_ID/reader.action?docID=6474267&ppg=933.

Martin Rajnoha, Radim Burget, and Lukas Povoda. Image background noise impact on convolutional neural network training. volume 1, pages 1–4. IEEE, 2018. doi: 10.1109/ICUMT.2018.8631242. URL https://ieeexplore.ieee.org/document/8631242.

I. Rallis, N. Bakalos, N. Doulamis, and A. Doulamis. Adaptable autoregressive moving average filter triggering convolutional neural networks for choreographic modeling. *ISPRS annals of the photogrammetry, remote sensing and spatial information sciences*, V-2-2020:467–474, Aug 3 2020. doi: 10.5194/isprs-annals-V-2-2020-467-2020.

Ioannis Rallis, Eftychios Protopapadakis, Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Georgios Bardis. Choreographic pattern analysis from heterogeneous motion capture systems using dynamic time warping. *Technologies (Basel)*, 7(3):56, Sep 1 2019. doi: 10.3390/technologies7030056.

R. Ranawana and V. Palade. Optimized precision - a new measure for classifier performance evaluation. pages 2254–2261. IEEE, 2006. ISBN 1089-778X. doi: 10.1109/CEC.2006.1688586. URL https://ieeexplore.ieee.org/document/1688586.

Michalis Raptis, Darko Kirovski, and Hugues Hoppe. Real-time classification of dance gestures from skeleton animation. ACM Conferences, pages 147–156, New York, NY, USA, 2011. ACM. doi: 10.1145/2019406.2019426.

Garvesh Raskutti, Ming Yuan, and Han Chen. Convex regularization for high-dimensional multiresponse tensor regression. *The Annals of Statistics*, 47(3):1554–1584, Jun 1 2019. doi: 10.1214/18-AOS1725.

Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *arXiv (Cornell University)*, Apr 19 2019. doi: 10.48550/arXiv.1904.09237.

Amjad Rehman, Muhammad Attique Khan, Tanzila Saba, Zahid Mehmood, Usman Tariq, and Noor Ayesha. Microscopic brain tumor detection and classification using 3d cnn and feature selection architecture. *Microscopy Research and Technique*, 84(1):133–149, Jan 2021. doi: 10.1002/jemt.23597.

Brian D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, Jan 18 1996. ISBN 0521460867. doi: 10.1017/CBO9780511812651. URL http://dx.doi.org/10.1017/CBO9780511812651.

Thomas J. Rothenberg. Identification in parametric models. *Econometrica*, 39(3):577–591, May 1 1971. doi: 10.2307/1913267.

Sebastian Ruder. An overview of gradient descent optimization algorithms. Technical report, Cornell University Library, arXiv.org, Jun 15 2017. URL https://www.proquest.com/docview/2076187906.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale

visual recognition challenge. *International journal of computer vision*, 115(3):211–252, Dec 1 2015. doi: 10.1007/s11263-015-0816-y.

P. Sadowski. Note on backpropagation in neural networks. Technical report, 2018.

Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization. *arXiv (Cornell University)*, May 29 2018. doi: 10.48550/arXiv.1805.11604.

S. Sharma, S. Sharma, and A. Athaiya. Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology*, 2020.

Haoze Shi, Naisen Yang, Hong Tang, and Xin Yang. asgd: Stochastic gradient descent with adaptive batch size for every parameter. *Mathematics (Basel)*, 10(6):863, Mar 1 2022. doi: 10.3390/math10060863.

N. Shi, D. Li, M. Hong, and R. Sun. Rmsprop converges with proper hyperparameter. *International Conference on Learning Representations*, 2020.

Ravid Shwartz-Ziv, Micah Goldblum, Yucen Lily Li, C. Bayan Bruss, and Andrew Gordon Wilson. Simplifying neural network training under class imbalance. *arXiv (Cornell University)*, Dec 5 2023. doi: 10.48550/arxiv.2312.02517.

Nicholas D. Sidiropoulos and Rasmus Bro. On the uniqueness of multilinear decomposition of n-way arrays. *Journal of chemometrics*, 14(3):229–239, May 2000. doi: 10.1002/1099-128X(200005/06)14:3<229::AID-CEM587>3.0.CO;2-N.

Jonathan W. Siegel and Jinchao Xu. Sharp bounds on the approximation rates, metric entropy, and n-widths of shallow neural networks. *arXiv (Cornell University)*, Jan 28 2021. doi: 10.48550/arxiv.2101.12365.

Jonathan W. Siegel and Jinchao Xu. High-order approximation rates for shallow neural networks with cosine and reluk activation functions. *Applied and computational harmonic analysis*, 58:1–26, May 2022. doi: 10.1016/j.acha.2021.12.005.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv (Cornell University)*, Sep 4 2014. doi: 10.48550/ARXIV.1409.1556.

Utkarsh Singhal, Carlos Esteves, Ameesh Makadia, and Stella X. Yu. Learning to transform for generalizable instance-wise invariance. Piscataway, 2023. The Institute of Electrical and Electronics Engineers, Inc. (IEEE). doi: 10.1109/ICCV51070.2023.00571. URL `https://www.proquest.com/docview/2915730694`.

Age K. Smilde, Rasmus Bro, and Paul Geladi. Multi-way analysis with applications in the chemical sciences, 2004. URL `https://onlinelibrary.wiley.com/doi/book/10.1002/0470012110`.

Samuel L. Smith, Erich Elsen, and Soham De. On the generalization benefit of noise in stochastic gradient descent. *arXiv (Cornell University)*, Jun 26 2020. doi: 10.48550/arxiv.2006.15081.

K. Srinivasan, K. Porkumaran, and G. Sainarayanan. Intelligent human body modeling and human pose classification using neural networks, 2010.

N. Srivastava, Hinton G., Krizhevsky A., Sutskever I., and Salakhutdinov R. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. *ArXiv, abs/1505.00387.*, 2015.

Alwin Stegeman and Nicholas D. Sidiropoulos. On kruskal's uniqueness condition for the cande-comp/parafac decomposition. *Linear algebra and its applications*, 420(2):540–552, Jan 15 2007. doi: 10.1016/j.laa.2006.08.010.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. *Proceedings of the 30th International Conference on Machine Learning*, 28(3): 1139–1147, 2013.

Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. *arXiv (Cornell University)*, Feb 23 2016. doi: 10.48550/arXiv.1602.07261.

Yoshio Takane, Forrest W. Young, and Jan de Leeuw. Nonmetric individual differences multidimensional scaling: An alternating least squares method with optimal scaling features. *Psychometrika*, 42(1):7–67, Mar 1977. doi: 10.1007/BF02293745.

Xu Tan, Yin Zhang, Siliang Tang, Jian Shao, Fei Wu, and Yueting Zhuang. *Logistic Tensor Regression for Classification*, volume 7751 of *Intelligent Science and Intelligent Data Engineering*, pages 573–581. Springer Berlin / Heidelberg, Germany, 2013. ISBN 3642366686. doi: 10.1007/978-3-642-36669-7_70. URL `http://ebookcentral.proquest.com/lib/SITE_ID/reader.action?docID=3093034&ppg=589`.

Rakesh Tanty and Tanweer S. Desmukh. Application of artificial neural network in hydrology- a review. *International journal of engineering research technology (Ahmedabad)*, 4(6), Jun 10 2015. doi: 10.17577/IJERTV4IS060247. ID: RefID:227-rakesh2015application; M1: Journal Article.

Dacheng Tao, Xuelong Li, Xindong Wu, Weiming Hu, and Stephen J. Maybank. Supervised tensor learning. *Knowledge and information systems*, 13(1):1–42, Sep 1 2007. doi: 10.1007/s10115-006-0050-6.

Jos M. F. ten Berge and Nikolaos D. Sidropoulos. On uniqueness in candecomp/parafac. *Psychometrika*, 67 (3):399–409, Sep 1 2002. doi: 10.1007/BF02294992.

Vajira Thambawita, Inga Strümke, Steven A. Hicks, Pål Halvorsen, Sravanthi Parasa, and Michael A. Riegler. Impact of image resolution on deep learning performance in endoscopy image classification: An experimental study using a large dataset of endoscopic images. *Diagnostics*, 11(12):2183, Nov 24 2021. doi: 10.3390/diagnostics11122183.

Junjiao Tian, Niluthpol Mithun, Zach Seymour, Han-Pang Chiu, and Zsolt Kira. Striking the right balance: Recall loss for semantic segmentation. *arXiv (Cornell University)*, Jun 28 2021. doi: 10.48550/arxiv.2106.14917.

T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4:26–30, 2012.

Konstantinos Tragiannis, Thanasis Balafoutis, Vasiliki Balaska, and Antonios Gasteratos. Skeleton-based recognition of traditional greek dance steps using machine learning algorithms. pages 1–6, Piscataway, 2023. IEEE. doi: 10.1109/IST59124.2023.10355702. URL `https://ieeexplore.ieee.org/document/10355702`.

L. R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, Sep 1966. doi: 10.1007/bf02289464.

Aad W. van der Vaart. *Asymptotic statistics*, volume 3. Cambridge Univ. Press, Cambridge [u.a.], 1. paperback ed. edition, 1998. ISBN 0521784506. doi: 10.1017/cbo9780511802256. URL `http://bvbr.bib-bvb.de:8991/F?func=service&doc_library=BVB01&local_base=BVB01&doc_number=017003721&sequence=000004&line_number=0001&func_code=DB_RECORDS&service_type=MEDIA`.

Sharan Vaswani, Francis Bach, and Mark Schmidt. Fast and faster convergence of sgd for over-parameterized models and an accelerated perceptron. *arXiv (Cornell University)*, Oct 16 2018. doi: 10.48550/arxiv.1810.07288.

Stefan Wager, Sida Wang, and Percy Liang. Dropout training as adaptive regularization. *arXiv (Cornell University)*, July 4 2013. doi: 10.48550/arxiv.1307.1493.

Guanghui Wang, Shiyin Lu, Weiwei Tu, and Lijun Zhang. Sadam: A variant of adam for strongly convex functions. *arXiv (Cornell University)*, May 8 2019. doi: 10.48550/arxiv.1905.02957.

Zaitian Wang, Pengfei Wang, Kunpeng Liu, Pengyang Wang, Yanjie Fu, Chang-Tien Lu, Charu C. Aggarwal, Jian Pei, and Yuanchun Zhou. A comprehensive survey on data augmentation. *arXiv (Cornell University)*, May 15 2024. doi: 10.48550/arxiv.2405.09591.

Halbert White. *Artificial neural networks : approximation and learning theory*. Blackwell, 1992. ISBN 9781557863294.

Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. *arXiv (Cornell University)*, May 23 2017. doi: 10.48550/arXiv.1705.08292.

Haibing Wu and Xiaodong Gu. Towards dropout training for convolutional neural networks. *Neural Networks*, 71:1–10, Nov 2015. doi: 10.1016/j.neunet.2015.07.007.

Jianxin Wu. introduction to convolutional neural networks. *National Key Lab for Novel Software Technology. Nanjing University. China*, 2017.

Kun Yan, Guannan Liu, Rende Xie, Shih-Hau Fang, Hsiao-Chun Wu, Shih Yu Chang, and Li Ma. Novel subject-dependent human-posture recognition approach using tensor regression. *IEEE sensors journal*, 25(1):1041–1053, Jan 1 2025. doi: 10.1109/JSEN.2024.3493893.

Greg Yang, Jeffrey Pennington, Vinay Rao, Jascha Sohl-Dickstein, and Samuel S. Schoenholz. A mean field theory of batch normalization. *arXiv (Cornell University)*, Feb 21 2019. doi: 10.48550/arxiv.1902.08129.

Muhamad Yani, Budhi Irawan S Si. M.T., and S. T. Casi Setiningsih M.T. Application of transfer learning using convolutional neural network method for early detection of terry's nail. *Journal of Physics: Conference Series*, 1201(1):12052, May 1 2019. doi: 10.1088/1742-6596/1201/1/012052.

Xiaozhe Yao. Implementing deconvolution to visualize and understand convolutional neural networks. 2020.

Dmitry Yarotsky. Universal approximations of invariant maps by neural networks. *arXiv (Cornell University)*, Apr 26 2018. doi: 10.48550/arxiv.1804.10306.

Xue Ying. An overview of overfitting and its solutions. *Journal of physics. Conference series*, 1168(2):22022, Feb 1 2019. doi: 10.1088/1742-6596/1168/2/022022.

K. Yu, W. Xu, and Y. Gong. Deep learning with kernel regularization for visual recognition. *Advances in neural information processing systems*, 21, 2009.

Rose Yu and Yan Liu. Learning from multiway data: Simple and efficient tensor regression. *arXiv (Cornell University)*, Jul 8 2016. doi: 10.48550/arxiv.1607.02535.

Yubai Yuan, Yujia Deng, Yanqing Zhang, and Annie Qu. Deep learning from a statistical perspective. *Stat (International Statistical Institute)*, 9(1):n/a, 2020. doi: 10.1002/sta4.294.

Afia Zafar, Muhammad Aamir, Nazri Mohd Nawi, Ali Arshad, Saman Riaz, Abdulrahman Alruban, Ashit Kumar Dutta, and Sultan Almotairi. A comparison of pooling methods for convolutional neural networks. *Applied Sciences*, 12(17):8643, Sep 1 2022. doi: 10.3390/app12178643.

M. D. Zeiler, G. W. Taylor, and R. Fergus. Adaptive deconvolutional networks for mid and high level feature learning. pages 2018–2025. IEEE, 2011. ISBN 1550-5499. doi: 10.1109/ICCV.2011.6126474. URL `https://ieeexplore.ieee.org/document/6126474`.

Matthew D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv (Cornell University)*, Dec 22 2012. doi: 10.48550/arXiv.1212.5701.

Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *Computer Vision – ECCV 2014*, pages 818–833, 2014. doi: 10.1007/978-3-319-10590-1_53.

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into deep learning*. Cambridge University Press, 2024. ISBN 1009389432.

Richard Zhang. Making convolutional networks shift-invariant again. *arXiv (Cornell University)*, Apr 25 2019. doi: 10.48550/arxiv.1904.11486.

Yushun Zhang, Congliang Chen, Naichen Shi, Ruoyu Sun, and Zhi-Quan Luo. Adam can converge without any modification on update rules. *arXiv (Cornell University)*, Aug 20 2022. doi: 10.48550/arxiv.2208.09632.

Zijun Zhang. Improved adam optimizer for deep neural networks. pages 1–2. IEEE, 2018. doi: 10.1109/IWQoS.2018.8624183. URL `https://ieeexplore.ieee.org/document/8624183`.

Allan Zhou, Fahim Tajwar, Alexander Robey, Tom Knowles, George J. Pappas, Hamed Hassani, and Chelsea Finn. Do deep networks transfer invariances across classes? *arXiv (Cornell University)*, Mar 18 2022. doi: 10.48550/arxiv.2203.09739.

Ding-Xuan Zhou. Universality of deep convolutional neural networks. *arXiv (Cornell University)*, May 28 2018. doi: 10.48550/arxiv.1805.10769.

Ding-Xuan Zhou. Theory of deep convolutional neural networks: Downsampling. *Neural Networks*, 124:319–327, Apr 2020. doi: 10.1016/j.neunet.2020.01.018.

Hua Zhou, Lexin Li, and Hongtu Zhu. Tensor regression with applications in neuroimaging data analysis. *Journal of the American Statistical Association*, 108(502):540–552, Jun 1 2013. doi: 10.1080/01621459.2013.776499.

Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. *Proceedings of the Twentieth International Conference on Machine Learning (ICML '03)*, pages 928–936, 2003.

F. Zou, L. Shen, Z. Jie, J. Sun, and W. Liu. Weighted adagrad with unified momentum. 2018.

A

Appendix A

A1: CNN Definitions and Theorems

Definition A1: *In general, given that* $\mathbf{x}, \mathbf{y} \in \mathbb{R}^p$ *and a function f:* $\mathbb{R}^p \to \mathbb{R}^s$*, then a differentiable function f is said to be Lipschitz gradient continuous if there exists a Lipschitz constant* $L > 0$ *such that*

$$\|\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} - \frac{\partial f(\mathbf{y})}{\partial \mathbf{y}}\| \leq L\|\mathbf{x} - \mathbf{y}\|. \tag{A.1}$$

Theorem A1: *The convolution operation given in equation (3.1) is commutative, i.e., given that*

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x - \alpha)g(\alpha)d\alpha, \tag{A.2}$$

and

$$(g * f)(x) = \int_{-\infty}^{\infty} g(x - \alpha)f(\alpha)d\alpha, \tag{A.3}$$

then

$$(f * g)(x) = (g * f)(x). \tag{A.4}$$

Proof: From equation (A.2), we take $u = x - \alpha$, which means that $\alpha = x - u$ and $du = d(-\alpha)$. The limits would also update from $\alpha = \inf$ to $u = x - \inf = -\inf$ and similarly for $\alpha = -\inf$. Therefore, we can update equation (A.2) to

$$(f * g)(x) = \int_{\infty}^{-\infty} f(u)g(x - u)d(-u), \tag{A.5}$$

which is identical to

$$(f * g)(x) = -\int_{\infty}^{-\infty} g(x - u)f(u)du, \tag{A.6}$$

since multiplication is commutative. Since we have

$$(f * g)(x) = \int_{-\infty}^{\infty} g(x-u)f(u)du, \tag{A.7}$$

it means that

$$(f * g)(x) = (g * f)(x), \tag{A.8}$$

which proves that the convolution operation is commutative. ■

Theorem A2: *The convolution operation given in equation (3.1) is translation equivariant.*

Proof: The convolution between functions f and g at a point $x \in \mathbb{R}$ is given as

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x-\lambda)g(\lambda)d\lambda. \tag{A.9}$$

Let us translate g by some constant t and define $g^{'}$ to be the translated function, i.e.,

$$g^{'}(x) = g(x-t). \tag{A.10}$$

The convolution operation between $g^{'}$ and f at point x is given as

$$(f * g^{'})(x) = \int_{-\infty}^{\infty} f(x-\lambda)g^{'}(\lambda)d\lambda. \tag{A.11}$$

Substituting equation (A.10) in equation (A.11) yields

$$(f * g^{'})(x) = \int_{-\infty}^{\infty} f(x-\lambda)g(\lambda-t)d\lambda. \tag{A.12}$$

Taking $\lambda^{'} = \lambda - t$ and substituting in equation (A.12) gives

$$(f * g^{'})(x) = \int_{-\infty}^{\infty} f(x-t-\lambda^{'})g(\lambda^{'})d\lambda^{'}, \tag{A.13}$$

since $d\lambda^{'} = d\lambda$. This shows that $(f * g^{'})(x) = (f * g)(x-t)$, i.e., equation (A.13) is the same as equation (A.9). ■

Theorem A3: *The cross-correlation operation given in equation (3.12) is not commutative, i.e., given that*

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x+\alpha)g(\alpha)d\alpha, \tag{A.14}$$

and

$$(g * f)(x) = \int_{-\infty}^{\infty} g(x+\alpha)f(\alpha)d\alpha, \tag{A.15}$$

then

$$(f * g)(x) \neq (g * f)(x). \tag{A.16}$$

Proof: From equation (A.14), we take $u = x + \alpha$, which means that $\alpha = u - x$ and $du = d\alpha$. Therefore, we can update equation (A.14) to

$$(f * g)(x) = \int_{-\infty}^{\infty} f(u)g(u - x)du. \tag{A.17}$$

Note that when we take $u = \alpha$ again in equation (A.17), we have

$$(f * g)(x) = \int_{-\infty}^{\infty} f(\alpha)g(\alpha - x)d\alpha = (g * f)(-x), \tag{A.18}$$

which shows that the cross-correlation operation is not commutative unless we have that $(g * f)(x) = (g * f)(-x)$. ■

A2: Convolution Operation - Matrix-Vector Multiplication

The convolution operation can be constructed in the form of a matrix-vector multiplication operation, where the filter **G** would be converted into a *Toeplitz* matrix and **F** would be flattened to a vector. The resulting vector that would be obtained from the matrix-vector multiplication operation would then be reshaped into a square matrix which would then be identical to $\mathbf{F} * \mathbf{G}$. Given that we have an input image **F** of size ($R \times C$) and a filter **G** of size ($P \times Q$), then if **F** is flattened to a RC-vector defined as $\mathbf{F}^{'}$ and **G** is replaced by a Toeplitz matrix $\mathbf{G}^{'}$ of size $((R - P + 1)(C - Q + 1) \times RC)$, then

$$\mathbf{F} * \mathbf{G} = reshape(\mathbf{G}^{'} * \mathbf{F}^{'}), \tag{A.19}$$

where the *reshape* function is to signify that $\mathbf{G}^{'} * \mathbf{F}^{'}$ will be reshaped from a $(R - P + 1)(C - Q + 1)$-vector to a $((R - P + 1) \times (C - Q + 1))$ matrix. The number of rows of $\mathbf{G}^{'}$ would be equal to the total number of filter passes over **F** and the number of columns of $\mathbf{G}^{'}$ would be equal to the total number of input pixels in **F**. Due to the reasonable size difference that there could be between the input matrix and the filter, the filter matrix can generally be very sparse. Figure A.1 is a simple example of a convolution operation that we have defined to use to compliment Figure A.2. Using the input and filter matrices from Figure A.1, Figure A.2 represents a virtual representation of the convolution operation as a matrix-vector multiplication operation. Note that when the vector output of Figure A.2 is reshaped to (3×3), then the result will be similar to the output as in Figure A.1.

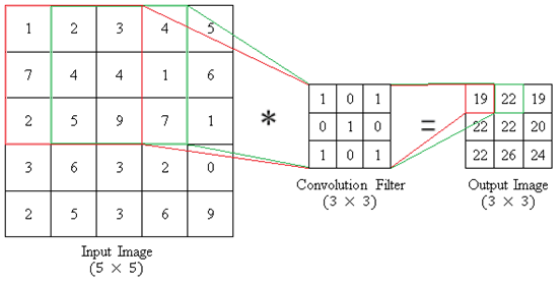


Figure A.1: Generic example of a convolution operation between an image and filter of sizes (5 × 5) and (3 × 3) respectively.

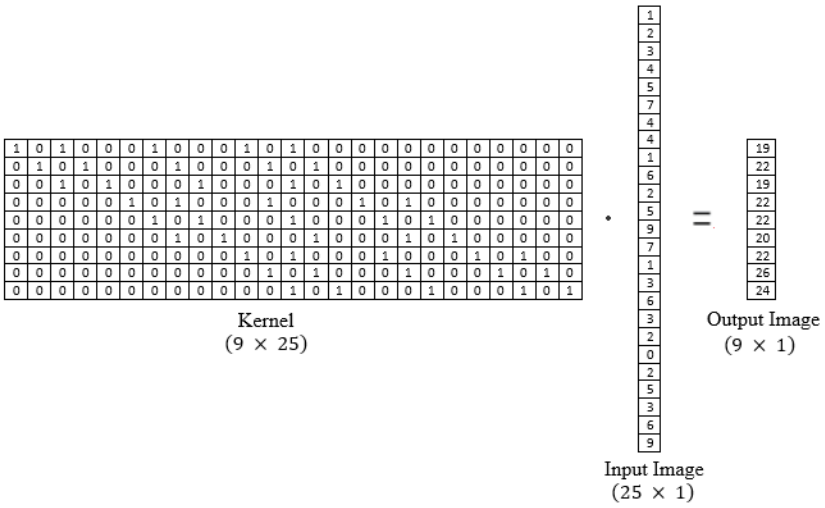


Figure A.2: A virtual representation of the convolution operation as a matrix-vector multiplication operation. The output image would then be reshaped to a (3 × 3) feature map.

The convolution operation can be represented instead as a matrix-vector multiplication operation, i.e., as in equation (3.15) for $\mathbf{V}_{jk}$, the j^{th} output feature map of the k^{th} layer, denoted as $\mathbf{V}_{jk}$, will be of size $(R_k \times C_k)$ and is given by

$$\mathbf{V}_{jk} = \mathbf{B}_{jk} + reshape(\sum_{a=1}^{m_{k-1}} \mathbf{W}_{ajk}\mathbf{V}_{a(k-1)}), \tag{A.20}$$

where $\mathbf{B}_{jk}$ is the bias matrix with size $(R_k \times C_k)$ filled with the b_{jk} entries, $\mathbf{W}_{ajk}$ is a Toeplitz matrix representing the filter matrix of size $(R_kC_k \times R_{k-1}C_{k-1})$ connecting the a^{th} feature map of the $k-1^{th}$ layer with the j^{th} output feature map of the k^th layer, $\mathbf{V}_{a(k-1)}$ represents the flattened a^{th} feature map of the $k-1^{th}$ layer with size $(R_{k-1}C_{k-1} \times 1)$, and the *reshape* function is to signify that $\sum_{a=1}^{m_{k-1}} \mathbf{W}_{ajk}\mathbf{V}_{a(k-1)}$ will be reshaped from a $(R_kC_k \times 1)$ vector to a $(R_k \times C_k)$ matrix.

A3: Convergence of Adam

In this section, we will be following the work by Reddi et al. (2019) and Zhang et al. (2022) to discover if there is any gap between theory and practice.

A3.1: Preliminaries

In this section, we will be analyzing the convergence of Adam and its variants by discussing the research made by Kingma and Ba (2017), Reddi et al. (2019), Shi et al. (2020) and Zhang et al. (2022). We are doing this since we will use Adam in the application for this chapter. We will be briefly going into some notation on the optimization setup used by Kingma and Ba (2017), which was the online learning framework proposed as in Zinkevich (2003).

Given that we consider Q iterations and a learning algorithm with error function E, we will have a sequence of Q convex error functions denoted by $E^1, E^2, ..., E^Q$, where E^q represents the error contributed by the q^{th} sample batch for $q \in [1, Q]$. Here, we are expecting E and each E^q for $q \in [1, Q]$ to represent the same type of error function. In general, the results presented in this section rely on the function $E(\mathbb{W})$ having a finite-sum structure as in equation (A.21) as presented in Vaswani et al. (2018), where our aim is to find the tensor of weights $\mathbb{W}$ that yields minimal error $E(\mathbb{W})$, i.e.,

$$\min_{\mathbb{W}} E(\mathbb{W}) = \min_{\mathbb{W}} \sum_{q=1}^{Q} E^q(\mathbb{W}). \tag{A.21}$$

At each iteration q, a tensor of parameters $\mathbb{W}^q$ is chosen and we denote its error as $E^q(\mathbb{W}^q)$, which represents the error of the learning algorithm with respect to parameters $\mathbb{W}^q$. Initially, we would expect $E^q(\mathbb{W}^q)$ to be large but it is expected that it would gradually decrease as the iterations go on due to the updated weights $\mathbb{W}^q$. Given that we have tensors of parameters $\mathbb{W}^1$, $\mathbb{W}^2$, ..., $\mathbb{W}^Q$, we represent the overall error of the algorithm until iteration Q as

$$C_{on}^Q = \sum_{q=1}^{Q} E^q(\mathbb{W}^q). \tag{A.22}$$

To analyze the performance of an algorithm given C^Q, we would compare our performance against an 'offline' learning framework that would have all of the information known to it before needing to make a decision. Given the error functions $E^1, E^2, ..., E^Q$, this framework would choose a tensor of parameters $\mathbb{W}$ that would best minimize equation (A.22) (Zinkevich, 2003), i.e.,

$$C_{off}^Q = \sum_{q=1}^{Q} E^q(\mathbb{W}). \tag{A.23}$$

By choosing only one tensor of parameters $\mathbb{W}$, equation (A.23) is assuming that $\mathbb{W}$ would remain the optimal choice over all iterations. Given equation (A.22) and the minimum of equation (A.23), the difference between these two equations would give us the *regret*. Ideally, we would try to minimize the regret since it would mean that the performance of our algorithm in the defined online learning framework would be comparable to that of an offline learning framework. The regret of the learning algorithm after Q iterations is given by

$$R^Q = \sum_{q=1}^{Q} E^q(\mathbb{W}^q) - \min_{\mathbb{W}} \sum_{q=1}^{Q} E^q(\mathbb{W}). \tag{A.24}$$

An algorithm is said to perform well if its 'regret is sublinear as a function of Q' meaning that $R^Q = \mathcal{O}(Q)$, where the notation $\mathcal{O}$ is known as the *Bachmann–Landau notation* and is used to describe how closely a finite series approximates a given function (Bachmann (1894), Hazan (2016)). This implies that, on average, the algorithm performs on par with the most effective fixed strategy determined in hindsight.

A3.2: The Non-Convergence of Adam

Zinkevich (2003) introduced an online learning framework for Online Gradient Descent (OGD) and proved that it attains a regret bound of $\mathcal{O}(\sqrt{Q})$ for the algorithm. OGD uses a fixed step size in its update rule, meaning that the algorithm is not taking into consideration the characteristics of the data being observed (Wang et al., 2019). As such, the regret bound of $\mathcal{O}(\sqrt{Q})$ is independent of the data in question and does not benefit from its structure (Wang et al., 2019). The various adaptive optimization algorithms that we have seen so far in previous sections try to address this limitation and make use of data to dynamically adjust the step size at each iteration.

Kingma and Ba (2017) evaluated Adam's theoretical convergence behaviour, deriving a regret bound for its convergence rate that matches state-of-the-art benchmarks under the online convex optimization framework. Under the assumption of bounded gradients of $E^q(\mathbb{W})$ and a linearly decreasing β_1 over time to 0, i.e., $\beta_1 \to 0$, the regret bound of Adam is guaranteed to be $\mathcal{O}(\sqrt{Q})$. The suggestion of decreasing β_1 over time towards zero was introduced by Sutskever et al. (2013), where they had shown that the reduction of momentum near the end of training can improve convergence.

Adaptive optimization algorithms have been observed to diverge in some settings. Kingma and Ba (2017) had proved the convergence of Adam for online convex optimization problems; however, Reddi et al. (2019) had discussed possible flaws in the proof made by Kingma and Ba (2017) which could lead to non-convergence under certain hyperparameter configurations with β_1 and β_2. Both Shi et al. (2020) and Zhang

et al. (2022) expanded on the work done by Reddi et al. (2019) where they proved that RMSprop and Adam can converge under certain hyperparameter assumptions on β_1 and β_2.

One thing worth noting between the works of Reddi et al. (2019), Shi et al. (2020) and Zhang et al. (2022) is with their interpretation of divergence. Both Shi et al. (2020) and Zhang et al. (2022) had interpreted divergence as 'diverging to infinity' instead of 'not converging to an optimal solution' which was the notion used in Reddi et al. (2019). We will be following the notion used by Shi et al. (2020) and Zhang et al. (2022) since 'not converging to an optimal solution' can include the case of converging to a bounded region near critical points.

Reddi et al. (2019) had shown an example which proved that Adam and other adaptive optimization algorithms like RMSprop and AdaDelta diverge for a wide range of hyperparameters. In this example, they had presented a constrained problem meaning that divergence to infinity cannot theoretically happen; however, Shi et al. (2020) had made another example with it being an unconstrained problem where the iterates and the gradients can diverge to infinity given small values of β_2. We will be presenting the example by Reddi et al. (2019) here. Given $x \in [-1, 1]$, $C \geq 2$ and iteration q, the counter-example presented by Reddi et al. (2019) considers the linear functions f^q with the following function sequence:

$$f^q(x) = \begin{cases} Cx, & \text{for } q \bmod 3 = 1, \\ -x, & \text{otherwise.} \end{cases} \tag{A.25}$$

For this example, $x = -1$ yields the minimum regret. When using Adam with equation (A.25), Reddi et al. (2019) had chosen a small β_2 value of $\beta_2 = \frac{1}{1+C^2}$ along with other hyperparameter values of $\beta_1 = 0$ and $\gamma < \sqrt{1-\beta_2}$. They go on to prove that for this convex problem, Adam does not converge to an optimal solution when $\beta_2 \leq \min\{C^{-\frac{4}{C-2}}, 1 - (\frac{9}{2C})^2\}$, where Adam converges to $x = 1$ (Zhang et al., 2022). For this example, SGD and AdaGrad would not have a problem converging to an optimal solution.

Reddi et al. (2019) present a general result where if β_1 and β_2 satisfy certain criteria, then for a convex optimization problem, the Adam algorithm will diverge, i.e.,

Theorem 3.1: *Given any constants $\beta_1, \beta_2 \in [0, 1)$ such that $\beta_1 < \sqrt{\beta_2}$, there is an online convex optimization problem where Adam has non-zero average regret, i.e., $R_Q/Q \not\to 0$ as $Q \to \infty$.*

They then go on to strengthen this result via Theorem 3.2 which considers the stochastic optimization setting, i.e.,

Theorem 3.2: *Given any constant $\beta_1, \beta_2 \in [0, 1)$ such that $\beta_1 < \sqrt{\beta_2}$, there is a stochastic convex optimization problem for which Adam diverges.*

We refer to Reddi et al. (2019) for the proof of Theorems 3.1 and 3.2. Theorem 3.1 shows that in situations where there is a small β_2, adaptive algorithms could diverge. As such, the convergence of adaptive algorithms like RMSprop and Adam are in general dependent on the value of β_2 (Shi et al., 2020). The theory by Shi et al. (2020) and Zhang et al. (2022) also confirm that β_2 needs to be large enough since otherwise it could lead to divergence. This suggests that having a large β_2 as Kingma and Ba (2017) had recommended is very important. The choice of β_1 is also restricted to $\beta_1 < \sqrt{\beta_2}$ as mentioned in Theorems 3.1 and 3.2, where with large enough β_2, β_1 can be chosen anywhere in the range $\beta_1 < \sqrt{\beta_2}$ without being at risk of divergence as in Theorem 3.1. Such results also highlight that the hyperparameters β_1 and β_2 are problem-dependent meaning that they vary problem to problem and as such need to be manually tuned to avoid bad convergence behaviour (Reddi et al. (2019), Shi et al. (2020)).

Given the step size at two adjacent iterations γ^q and γ^{q-1}, the following quantity Γ^q helps to quantify the difference in the inverse step size of the adaptive optimization algorithm iteration by iteration:

$$\Gamma^q = \frac{1}{\gamma^q} - \frac{1}{\gamma^{q-1}}, \tag{A.26}$$

where in the case of Adam it would be defined as

$$\Gamma^q = \frac{\sqrt{\widetilde{\mathbb{M}}_2^q} + \varepsilon}{\gamma} - \frac{\sqrt{\widetilde{\mathbb{M}}_2^{q-1}} + \varepsilon}{\gamma}. \tag{A.27}$$

where $\widetilde{\mathbb{M}}_2^q$ are defined as in equation (3.44) and ε is a very small negligible value.

According to Reddi et al. (2019), the main cause of divergences of such adaptive optimization algorithms is mainly controlled by the difference there would be between the inverses of the step sizes of two adjacent iterations as in equation (A.26) (Zou et al., 2018). In the case of SGD and AdaGrad, we observe that for the example given in equation (A.25), we would have $\Gamma^q \geq 0$ for all iterations due to 'non-increasing' step sizes in their update rules (Reddi et al., 2019). In settings where adaptive optimization algorithms do not converge to an optimal solution, it has been observed that it is rare for there to be minibatches with large gradients, where such minibatches would provide valuable information to the network. However, due to the use of an exponential moving average, the influence of these large gradients would eventually wane out which would lead to poor convergence where in such situations for iteration q, the positive-definiteness of Γ^q could be violated (Reddi et al., 2019).

Reddi et al. (2019) had concluded that the failure of Adam to converge to an optimal solution is brought about due to the exponential moving average used in the algorithm. The convergence issues are brought about by the updates reliance on solely using the gradients of the past few iterations as reference, where if the optimization algorithm had a long-term memory of past gradients instead of a fixed window of recent gradients then convergence would be ensured. To counter these convergence issues, Reddi et al. (2019) had proposed *AMSGrad* which is similar to Adam but relies on long-term memory of past gradients instead of an exponential average of past squared gradients and is guaranteed to converge under certain conditions (Shi et al., 2020). At a particular iteration q, the aim of AMSGrad is to maintain positive-semi definiteness of Γ^q, i.e., $\Gamma^q \succeq 0$ for all iterations and it does this by making $\widetilde{\mathbb{M}}_2^q$ non-decreasing. AMSGrad is equipped with data-dependant regret bounds, which are $\mathcal{O}(\sqrt{Q})$ in the worst case and become tighter when gradients are sparse (Wang et al., 2019). Note that AMSGrad still assumes bounded gradients of $E^q(\mathbb{W})$ and a linearly decreasing β_1 to 0 similar to Adam, although Alacaoglu et al. (2020) had relaxed the assumption of linearly decreasing β_1 to 0 to constant β_1 while maintaining data-dependent regret bounds of $\mathcal{O}(\sqrt{Q})$.

Reddi et al. (2019) addressed the convergence question of adaptive algorithms like Adam in the case of small β_2 where this would lead to divergence; however, does Adam provably converge in the case of large β_2? Shi et al. (2020) remark that with proper hyperparameter tuning and a sufficiently large β_2 value, RMSprop does not have convergence issues that often in practice. They also noted that from the example given in equation (A.25), Reddi et al. (2019) had not tested if large values of β_2 leads to convergence, where Shi et al. (2020) noted that larger values of C would require a β_2 value closer to 1 for convergence since otherwise the algorithm would diverge.

A3.3: Adam Converges with Appropriate Hyperparameters

It is worth noting that, regardless of the convergence issues raised in the previous section, both RMSprop and Adam are still very popular algorithms used in practice. Both algorithms work well in practice even though hyperparameters for β_1 and β_2 would often be defined as in Theorem 3.2 which would satisfy the divergence condition (Zhang et al., 2022). As such, most researchers had investigated whether there could be a large theory-practice gap, where the theoretical analysis of RMSprop and Adam does not match how it is used in practice (Shi et al. (2020), Zhang et al. (2022)). Indeed, the assumption of linearly decreasing β_1 to 0 by Kingma and Ba (2017) and Reddi et al. (2019) is unlike what is done in practice, where generally a constant value of β_1 and β_2 are chosen.

In practice, we would generally have a problem at hand where we would then proceed to find suitable values for β_1 and β_2 which is unlike what was done in Reddi et al. (2019), where the values of β_1 and β_2 were chosen first and then the problem was found as in equation (A.25). For example, many applications in practice use the default values of the hyperparameters recommended by Kingma and Ba (2017) for β_1 and β_2 at 0.9 and 0.999, respectively, which would fall in the divergence region $\beta_1 < \sqrt{\beta_2}$. There are other possible examples which would have different β_1 and β_2 values from the default but would still fall in this divergence region and they would not encounter any divergence issues while achieving good performance (Zhang et al., 2022).

Zhang et al. (2022) presented several experiments where values of β_1 and β_2 falling in the region of $[0,1)^2$ were chosen and the convergence of Adam was tested. In such experiments, they found that in the case of a small value of β_2 and any value of β_1 from a wide range of values, there were divergence regions of (β_1, β_2) where Adam always performs poorly with relatively large error. However, there were regions of (β_1, β_2) where Adam performs very well, specifically when there is a large value of β_2. Due to this, it is possible that Adam can converge for any problem as long as both parameters are chosen properly, particularly the value of β_2 (Zhang et al., 2022).

In their analysis, both Shi et al. (2020) and Zhang et al. (2022) make two assumptions, first they assume that at any iteration q, E^q satisfies the *Lipschitz gradient continuous* property with *Lipschitz constant* L and that E is lower bounded by some finite constant E^* (for the definition of a Lipschitz gradient continuous function, refer to Definition A1 in Appendix A for detail). The other assumption they define is that

$$\sum_{q=1}^{Q} \| \frac{\partial E^q}{\partial \mathbb{W}} \|_2 \leq D_1 \| \frac{\partial E}{\partial \mathbb{W}} \|_2 + D_0, \tag{A.28}$$

where Q represents the total number of iterations/minibatches, both D_0 and D_1 are non-negative constants and $\|.\|_2$ represents the l_2-norm. In the case when $D_0 = 0$, the assumption from equation (A.28) would be known as the *Strong Growth Condition* (SGC). The SGC relates the rates at which the l_2-norm of the stochastic gradients decrease relative to the batch gradient (Vaswani et al., 2018). For this to hold when $\frac{\partial E}{\partial \mathbb{W}} = \mathbb{O}$, where $\mathbb{O}$ represents a tensor of zeros of same size as $\mathbb{W}$, then we would have the $\frac{\partial E^q}{\partial \mathbb{W}} = \mathbb{O}$ well for all q. For over-parameterized deep neural networks, the SGC property is reasonable since such models are expressive enough to fit or interpolate the training dataset completely (Vaswani et al., 2018).

Shi et al. (2020) pointed out three different algorithm behaviours, notably the situation where the algorithm diverges, the algorithm converges to a bounded region near critical points or the algorithm converges to critical points. For equation (A.28), Shi

et al. (2020) had divided optimization problems into two cases, one where $D_0 > 0$ and the other when $D_0 = 0$. When $D_0 = 0$, we say that the optimization problem is *realizable* and it is *non-realizable* when $D_0 > 0$ (Shi et al., 2020).

Under the necessary requirement of large β_2 that is above some threshold α, Zhang et al. (2022) had presented a convergence result with Theorem 3.3 that would ensure the convergence of Adam under certain assumptions.

Theorem 3.3: *Assume that we have an error function E which is lower bounded by E^* and that E^q satisfies the Lipschitz gradient continuous property with constant L given as in equation (A.1). Also assume a number of iterations Q and that equation (A.28) holds for fixed D_0 and D_1. Given that $\beta_1 < \sqrt{\beta_2} < 1$, with β_2 greater than or equal to some threshold α, and a learning rate γ with step sizes $\gamma^q = \frac{\gamma}{\sqrt{q}}$ at iteration q, then for $\omega^* \in \mathbb{N}$ satisfying $\omega^* \geq 4$ and $\beta_1^{(\omega^*-1)Q} \leq \frac{\beta_1^Q}{\sqrt{\omega^*-1}}$ we have the following result for any total number of epochs Ω such that $\Omega > \omega^*$:*

$$\min_{\omega \in [\omega^*, \Omega]} \mathbb{E}[\min\{\sqrt{\frac{2D_1}{D_0}} \| \frac{\partial E^{\omega,1}}{\partial \mathbb{W}^{\omega,0}} \|_2^2, \| \frac{\partial E^{\omega,1}}{\partial \mathbb{W}^{\omega,0}} \|_2\}] = \mathcal{O}(\frac{\log \Omega}{\sqrt{\Omega}}) + \mathcal{O}(\sqrt{D_0}), \tag{A.29}$$

where $\mathbb{W}^{\omega,0}$ represents the starting weights of the ω^{th} epoch from the previous epoch. In the case where ω is the first epoch, then $\mathbb{W}^{\omega,0}$ represents the randomly initialized weights at the start of training. When the optimization problem is realizable, then the result changes to the following:

$$\min_{\omega \in [\omega^*, \Omega]} \mathbb{E}[\| \frac{\partial E^{\omega,1}}{\partial \mathbb{W}^{\omega,0}} \|_2\}] = \mathcal{O}(\frac{\log \Omega}{\sqrt{\Omega}}), \tag{A.30}$$

since $D_0 = 0$.

We refer to Zhang et al. (2022) for the proof of Theorem 3.3. In the case of non-realizable problems with similar conditions as in Theorem 3.3 with a large enough β_2, Adam converges to a bounded neighbourhood near critical points (Shi et al. (2020), Zhang et al. (2022)). This is because even for convex optimization problems, the effective step size $\frac{\gamma^q}{\sqrt{\widetilde{\mathbb{M}}_2^q + \varepsilon}}$ does not necessarily decay even with the decreasing step size γ^q and hence Adam does not reach exactly zero gradient (Vaswani et al. (2018), Zhang et al. (2022)). The size of this bounded neighbourhood near critical points depends on the variance of the stochastic gradients (Zhang et al., 2022). On the other hand, in the case of realizable problems, then Adam can converge to critical points (Zhang et al., 2022).

When β_2 would fall below the threshold α, then Adam would be liable to divergence. Zhang et al. (2022) had generalized Theorem 3.1 by relaxing the condition of $\beta_1 < \sqrt{\beta_2}$ and considering it in the case of fixed Q instead of $Q \to \infty$, i.e.,

Corollary 3.1: *Given any constant β_1, $\beta_2 \in [0,1)$, there exists a convex function that satisfies the two assumptions we defined earlier for which the Adam's iterates and function values diverge to infinity.*

We refer to Zhang et al. (2022) for the proof of Corollary 3.1. In the general setting where the default values of β_1 and β_2 are taken to be 0.9 and 0.999 respectively, Theorem 3.3 would cover the convergence of Adam since $\beta_1 = 0.9$ falls under the minimum possible β_1 value of $\beta_1 = \sqrt{\beta_2} = 0.9995$. Corollary 3.1, similarly as in Theorem 3.1, highlights that there is a region of (β_1, β_2) in which Adam diverges. This divergence region would fall below the threshold α meaning that the result is consistent with Theorem 3.3 (Zhang et al., 2022). To be above the threshold of α, a value of $\beta_2 \geq 1 - \mathcal{O}(\frac{1-\beta_1}{Q^2\zeta})$ is required, where ζ represents a constant observed to be valued at around $\zeta = \mathcal{O}(Q)$ (Zhang et al., 2022). Note that larger values of β_1 increase the value of the threshold α. Similarly as Q increases, the threshold α gets larger. The main difference between Theorems 3.1 and 3.4 is that Reddi et al. (2019) had considered the case where they allow Q to change, whereas Zhang et al. (2022) considers the case where Q is fixed. As Q gets larger, the divergence region of (β_1, β_2) grows and converges to the whole region of $(\beta_1, \beta_2) = [0,1)^2$, which recovers the divergence result raised in Theorem 3.1 by Reddi et al. (2019) that required $\beta_1 < \sqrt{\beta_2}$ (Zhang et al., 2022).

A4: Universal Approximation Theorem for CNNs - Proofs

Lemma 3.1: *Given a 2D input image $\mathbf{F} \in \mathbb{R}^{R \times R}$ and $\mathbf{G} \in \mathbb{R}^{5\times5}$ where $R > 2$, then there exists $\mathbf{G}^1_{i,j}, \mathbf{G}^2_{i,j} \in \mathbb{R}^{3\times3}$ for $i, j = -1, 0, 1$ such that*

$$\mathbf{G} * \mathbf{F} = \sum_{i,j=-1,0,1} \mathbf{G}^1_{i,j} * \mathbf{G}^2_{i,j} * \mathbf{F}, \tag{A.31}$$

where $*$ means the convolution operation with one filter and periodic padding.

Proof: Considering the following example where

$$\begin{aligned}
\mathbf{G}^1_{-1,-1} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \mathbf{G}^1_{-1,0} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \mathbf{G}^1_{-1,1} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \\
\mathbf{G}^1_{0,-1} &= \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \mathbf{G}^1_{0,0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \mathbf{G}^1_{0,1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \\
\mathbf{G}^1_{1,-1} &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \mathbf{G}^1_{1,0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \mathbf{G}^1_{1,1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix},
\end{aligned} \tag{A.32}$$

and

$$\mathbf{G}^2_{-1,-1} = \begin{bmatrix} \mathbf{G}_{-2,-2} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \mathbf{G}^2_{-1,0} = \begin{bmatrix} \mathbf{G}_{-2,-1} & \mathbf{G}_{-2,0} & \mathbf{G}_{-2,1} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \mathbf{G}^2_{-1,1} = \begin{bmatrix} 0 & 0 & \mathbf{G}_{-2,2} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

$$\mathbf{G}^2_{0,-1} = \begin{bmatrix} \mathbf{G}_{-1,-2} & 0 & 0 \\ \mathbf{G}_{0,-2} & 0 & 0 \\ \mathbf{G}_{1,-2} & 0 & 0 \end{bmatrix}, \mathbf{G}^2_{0,0} = \begin{bmatrix} \mathbf{G}_{-1,-1} & \mathbf{G}_{-1,0} & \mathbf{G}_{-1,1} \\ \mathbf{G}_{0,-1} & \mathbf{G}_{0,0} & \mathbf{G}_{0,1} \\ \mathbf{G}_{1,-1} & \mathbf{G}_{1,0} & \mathbf{G}_{1,1} \end{bmatrix}, \mathbf{G}^2_{0,1} = \begin{bmatrix} 0 & 0 & \mathbf{G}_{-1,2} \\ 0 & 0 & \mathbf{G}_{0,2} \\ 0 & 0 & \mathbf{G}_{1,2} \end{bmatrix}, \tag{A.33}$$

$$\mathbf{G}^2_{1,-1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \mathbf{G}_{2,-2} & 0 & 0 \end{bmatrix}, \mathbf{G}^2_{1,0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \mathbf{G}_{2,-1} & \mathbf{G}_{2,0} & \mathbf{G}_{2,1} \end{bmatrix}, \mathbf{G}^2_{1,1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathbf{G}_{2,2} \end{bmatrix},$$

then the result from equation (A.31) would follow. ■

Theorem 3.4: *Given a 2D input image* $\mathbf{F} \in \mathbb{R}^{R \times R}$ *and* $\mathbf{G} \in \mathbb{R}^{(2P+1)\times(2P+1)}$*, where* $R > P$*, then there exists* $\mathbf{G}^1_{i,j} \in \mathbb{R}^{3\times 3}$ *and* $\mathbf{G}^2_{i,j} \in \mathbb{R}^{(2P-1)\times(2P-1)}$ *for* $i, j = -1, 0, 1$ *such that*

$$\mathbf{G} * \mathbf{F} = \sum_{i,j=-1,0,1} \mathbf{G}^1_{i,j} * \mathbf{G}^2_{i,j} * \mathbf{F}, \tag{A.34}$$

where $*$ *means the convolution operation with one filter and periodic padding.*

Proof: For any $\mathbf{G} \in \mathbb{R}^{(2P+1)\times(2P+1)}$, consider its decomposition as

$$\mathbf{G} = \sum_{i,j=-1,0,1} \hat{\mathbf{G}}_{i,j}, \tag{A.35}$$

where $\hat{\mathbf{G}}_{i,j} \in \mathbb{R}^{(2P+1)\times(2P+1)}$, for $i, j = -1, 0, 1$, are given as

$$\hat{\mathbf{G}}_{-1,-1} = \begin{bmatrix} \mathbf{G}_{-P,-P} & 0 & \cdots & 0 \\ 0 & 0 & \vdots & \vdots \\ \vdots & \cdots & \ddots & 0 \\ 0 & \cdots & 0 & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{G}^2_{-1,-1} & & 0 & 0 \\ & & \vdots & \vdots \\ 0 & \cdots & 0 & 0 \\ 0 & \cdots & 0 & 0 \end{bmatrix},$$

$$\hat{\mathbf{G}}_{-1,0} = \begin{bmatrix} 0 & \mathbf{G}_{-P,-P+1} & \cdots & \mathbf{G}_{-P,P-1} & 0 \\ 0 & 0 & \cdots & 0 & \vdots \\ \vdots & \ddots & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & 0 \end{bmatrix} = \begin{bmatrix} 0 & \mathbf{G}^2_{-1,0} & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{bmatrix},$$

$$\vdots$$

$$\hat{\mathbb{G}}_{1,1} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & \ddots & \vdots & \vdots \\ \vdots & \ddots & 0 & 0 \\ 0 & \cdots & 0 & \mathbf{G}_{P,P} \end{bmatrix} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \\ 0 & 0 & \mathbf{G}^2_{-1,-1} & \end{bmatrix},$$

and $\mathbf{G}^2_{i,j} \in \mathbb{R}^{(2P-1)\text{x}(2P-1)}$

$$\mathbf{G}^2_{-1,-1} = \begin{bmatrix} \mathbf{G}_{-P,-P} & 0 & \cdots & 0 \\ 0 & 0 & \vdots & \vdots \\ \vdots & \cdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 \end{bmatrix}, \mathbf{G}^2_{-1,0} = \begin{bmatrix} \mathbf{G}_{-P,-P+1} & \cdots & \mathbf{G}_{-P,P-1} \\ 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}, \mathbf{G}^2_{-1,1} = \begin{bmatrix} 0 & \cdots & 0 & \mathbf{G}_{-P,P} \\ \vdots & \ddots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 \end{bmatrix},$$

$$\mathbf{G}^2_{0,-1} = \begin{bmatrix} \mathbf{G}_{-P+1,-P} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & 0 & \vdots & 0 \\ \mathbf{G}_{P-1,-P} & 0 & \cdots & 0 \end{bmatrix}, \mathbf{G}^2_{0,0} = \begin{bmatrix} \mathbf{G}_{-P+1,-P+1} & \cdots & \mathbf{G}_{-P+1,P-1} \\ \vdots & \ddots & \vdots \\ \mathbf{G}_{P-1,-P+1} & \cdots & \mathbf{G}_{P-1,P-1} \end{bmatrix}, \mathbf{G}^2_{0,1} = \begin{bmatrix} 0 & \cdots & 0 & \mathbf{G}_{-P+1,P} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & \vdots \\ 0 & \cdots & 0 & \mathbf{G}_{P-1,P} \end{bmatrix},$$

$$\mathbf{G}^2_{1,-1} = \begin{bmatrix} 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ddots & 0 \\ \mathbf{G}_{P,-P} & 0 & \cdots & 0 \end{bmatrix}, \mathbf{G}^2_{1,0} = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \\ \mathbf{G}_{P,-P+1} & \cdots & \mathbf{G}_{P,P-1} \end{bmatrix}, \mathbf{G}^2_{1,1} = \begin{bmatrix} 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots \\ \vdots & \cdots & 0 & 0 \\ 0 & \cdots & 0 & \mathbf{G}_{P,P} \end{bmatrix}. \tag{A.36}$$

We need to verify that

$$\hat{\mathbf{G}}_{i,j} * \mathbf{F} = \mathbf{G}^1_{i,j} * \mathbf{G}^2_{i,j} * \mathbf{F}, \tag{A.37}$$

for $i, j = -1, 0, 1$.

In the case of periodic padding, we notice that for $1 \le r, c \le R$:

$$[\mathbf{G}^2_{i,j} * \mathbf{F}]_{r,c} = \mathbf{F}_{r+i,c+j} \tag{A.38}$$

for $i, j = -1, 0, 1$. Since for $1 \le r, c \le R$ and $i, j = -1, 0, 1$, we have that

$$\begin{aligned} [\mathbf{G}^1_{i,j} * \mathbf{G}^2_{i,j} * \mathbf{F}]_{r,c} &= \sum_{p,q=-P+1,\ldots,P-1} [\mathbf{G}^1_{i,j}]_{p,q} [\mathbf{G}^2_{i,j} * \mathbf{F}]_{r+p,c+q} \\ &= \sum_{p,q=-P+1,\ldots,P-1} [\mathbf{G}^1_{i,j}]_{p,q} [\mathbf{F}]_{r+p+i,c+q+j} \\ &= \sum_{\substack{p'=-P+1+i,\ldots,P-1+j, \\ q'=-P+1+j,\ldots,P-1+j}} [\mathbf{G}^1_{i,j}]_{p'-i,q'-j} [\mathbf{F}]_{r+p',c+q'} \\ &= \sum_{p',q'=-P,\ldots,P} [\hat{\mathbf{G}}_{i,j}]_{p',q'} [\mathbf{F}]_{r+p',c+q'}, \end{aligned} \tag{A.39}$$

which when referring to equation (3.12), we will get

$$[\mathbf{G}^1_{i,j} * \mathbf{G}^2_{i,j} * \mathbf{F}]_{r,c} = [\hat{\mathbf{G}}_{i,j} * \mathbf{F}]_{r,c}, \tag{A.40}$$

then the result follows. ■

Theorem 3.5: *Given a 2D input image* $\mathbf{F} \in \mathbb{R}^{RxR}$ *and* $\mathbb{G} \in \mathbb{R}^{(2P+1)x(2P+1)xm}$*, where* $R > P$ *and* m *represents the number of feature maps, then there exists a series of filters* $\mathbb{G}^1_v \in \mathbb{R}^{3x3xc_{v-1}xc_v}$ *and* $\mathbb{G}^2 \in \mathbb{R}^{3x3x(2v-1)^2xm}$ *for number of feature maps per layer given as* $c_v = (2v+1)^2$ *and layer number* $v = 1, ..., P-1$ *such that*

$$\mathbb{G} * \mathbf{F} = \mathbb{G}^2 * \mathbb{G}^1_{P-1} * \cdots * \mathbb{G}^1_1 * \mathbf{F}, \tag{A.41}$$

where $*$ *means the convolution operation with multiple filters and periodic padding.*

Proof: Given that $v = 1, ..., P-1$ and $v' \leq v$, the index set $\mathcal{I}_v$ has the following feature

$$((i_1, j_1), ..., (i_v, j_v)) \in \mathcal{I}_v \Rightarrow ((i_1, j_1), ..., (i_{v'}, j_{v'})) \in \mathcal{I}_{v'}, \tag{A.42}$$

which means that we can define an operator $\pi_v : \mathcal{I}_v \mapsto \mathcal{I}_{v-1}$ such that

$$\pi_v(((i_1, j_1), ..., (i_v, j_v))) = ((i_1, j_1), ..., (i_{v-1}, j_{v-1})). \tag{A.43}$$

By fixing the following bijection $\Pi_v : \{1, 2, ..., (2v+1)^v\} \mapsto \mathcal{I}_v$ for each $\mathcal{I}_1, ..., \mathcal{I}_{v-1}$, each element in $\mathcal{I}_v$ can be fixed to a unique position. Using this for $a = 1, ..., c_{v-1}$ and $f = 1, ..., c_v$, we can create $\mathbb{G}^1_v \in \mathbb{R}^{3x3xc_{v-1}xc_v}$ using

$$[\mathbb{G}^1_v]_{a,f} = \begin{cases} \mathbb{G}^1_{i_v,j_v}, & \text{if } \Pi_v(f) = ((i_1, j_1), ..., (i_v, j_v)) \text{ and } \Pi_{v-1}(a) = \pi_v(\Pi_v(f)) \\ 0, & \text{otherwise,} \end{cases} \tag{A.44}$$

for $v = 1, ..., P-1$.

Given that $\Pi_v(f) = ((i_1, j_1), ..., (i_v, j_v)) \in \mathcal{I}_v$ for $1 \leq f \leq (2v+1)^2$, we have

$$\begin{aligned} [\mathbb{G}^1_v * \cdots * \mathbb{G}^1_1 * \mathbf{F}]_f &= \sum_{a=1}^{c_{v-1}} [\mathbb{G}^1_v]_{a,f} * [\mathbb{G}^1_{v-1} * \cdots * \mathbb{G}^1_1 * \mathbf{F}]_a \\ &= \mathbf{G}^1_{i_v,j_v} * [\mathbb{G}^1_{v-1} * \cdots * \mathbb{G}^1_1 * \mathbf{F}]_{\Pi^{-1}_{v-1}(\pi_v(\Pi_v(f)))} \\ &= \mathbf{G}^1_{i_v,j_v} * \mathbb{G}^1_{i_{v-1},j_{v-1}} * [\mathbb{G}^1_{v-2} * \cdots * \mathbb{G}^1_1 * \mathbf{F}]_{\Pi^{-1}_{v-2}(\pi_{v-1}(\pi_v(\Pi_v(f))))} \\ &= \cdots \\ &= \mathbf{G}^1_{i_v,j_v} * \mathbb{G}^1_{i_{v-1},j_{v-1}} * \mathbb{G}^1_{i_{v-2},j_{v-2}} * \cdots * \mathbf{G}^1_{i_1,j_1} * \mathbf{F}. \end{aligned} \tag{A.45}$$

Using Theorem 3.4 on each filter $[\mathbb{G}]_m$ of $\mathbb{G}$, we have

$$[\mathbb{G}]_m * \mathbf{F} = \sum_{((i_1,j_1),...,(i_{P-1},j_{P-1}))\in\mathcal{I}_{P-1}} [\mathbb{G}^2_{(i_1,j_1),...,(i_{P-1},j_{P-1})}]_m * \mathbf{G}^1_{i_{P-1},j_{P-1}} * \cdots * \mathbb{G}^1_{i_1,j_1} * \mathbf{F}. \tag{A.46}$$

Since $(i_1, j_1), ..., (i_{P-1}, j_{P-1}) = \Pi_{P-1}^{-1}(a)$, we have that

$$[\mathbb{G}^2_{\Pi_{P-1}^{-1}(a)}]_m = [\mathbb{G}^2]_{a,m}, \tag{A.47}$$

which completes the proof. ■

Lemma 3.3: *For every* $\mathbf{F} \in \mathbb{R}^{RxR}$, $\mathbf{A} \in \mathbb{R}^{R^2xm}$ *and* $\xi_1, \xi_2 \in \mathbb{R}^m$ *for any* m, *there exists a convolutional filter* $\mathbb{G}^{(2\lfloor \frac{R}{2} \rfloor+1)x(2\lfloor \frac{R}{2} \rfloor+1)xm}$, *bias* $\mathbb{B} \in \mathbb{R}^{RxRxm}$, *and weight vector* $\mathbf{w} \in \mathbb{R}^{mR^2}$ *such that*

$$\xi_1 \cdot \sigma(\mathbf{A}\text{vec}(\mathbf{F}) + \xi_2) = \mathbf{w} \cdot \text{vec}(\sigma(\mathbb{G} * \mathbf{F} + \xi_2)). \tag{A.48}$$

Proof: Given that R is odd valued so that we have $R = 2\lfloor \frac{R}{2} \rfloor + 1$ and any m, then we have

$$[\mathbb{G} * \mathbf{F}]_{\lceil \frac{R}{2} \rceil, \lceil \frac{R}{2} \rceil, m} = \sum_{r,c=-\lfloor \frac{R}{2} \rfloor}^{\lfloor \frac{R}{2} \rfloor} [\mathbb{G}]_{r,c,m} [\mathbf{F}]_{\lceil \frac{R}{2} \rceil + r, \lceil \frac{R}{2} \rceil + c} = \text{vec}([\mathbb{G}]_{:,:,m}) \cdot \text{vec}(\mathbf{F}). \tag{A.49}$$

When we have the vector of unique biases $\mathbb{B}^1$ for the convolutional layer equal to ξ_2, $\text{vec}([\mathbb{G}]_{:,:,m}] = [\mathbf{A}]_{:,m}$, and

$$[\mathbf{w}]_{d'} = \begin{cases} [\xi_1]_m, & \text{if } d' = (m-1)R^2 + \lceil \frac{R}{2} \rceil^2 \\ 0, & \text{otherwise,} \end{cases} \tag{A.50}$$

for any m, then the proof follows.

Given that R is even valued so that we have $R = 2\lfloor \frac{R}{2} \rfloor + 1 = R + 1$ and any m, then we have

$$[\mathbb{G} * \mathbf{F}]_{\frac{R}{2}+1, \frac{R}{2}+1, m} = \sum_{r,c=-\frac{R}{2}}^{\frac{R}{2}} [\mathbb{G}]_{r,c,m} [\mathbf{F}]_{\frac{R}{2}+1+r, \frac{R}{2}+1+c} = \text{vec}([\mathbb{G}]_{-\frac{R}{2}:\frac{R}{2}-1, -\frac{R}{2}:\frac{R}{2}-1, m}) \cdot \text{vec}(\mathbf{F}). \tag{A.51}$$

When we have similar conditions for ξ_1 and ξ_2 as before from the odd valued case and additionally taking $\text{vec}([\mathbb{G}]_{-\frac{R}{2}:\frac{R}{2}-1, -\frac{R}{2}:\frac{R}{2}-1, m}) = [\mathbf{A}]_{:,m}$ and $[\mathbb{G}]_{-\frac{R}{2}:\frac{R}{2}, \frac{R}{2}, m} = [\mathbb{G}]_{\frac{R}{2}, -\frac{R}{2}:\frac{R}{2}, m} = 0$, then the proof follows as well for the even valued case of R. ■

Lemma 3.4: *Given that* Δ *is a bounded set in* $\mathbb{R}^{RxR}$ *with* $\mathbf{F} \in \Delta$ *and that we have a filter* $\mathbb{G} \in \mathbb{R}^{(2\lfloor \frac{R}{2} \rfloor+1)x(2\lfloor \frac{R}{2} \rfloor+1)xm}$ *and bias* $\mathbb{B}^{RxRxm}$, *then there is a series of filters* $\mathbb{G}_v \in \mathbb{R}^{3x3xc_{v-1}xc_v}$ *and biases* $\mathbb{B}_v \in \mathbb{R}^{RxRxc_v}$ *with number of feature maps per layer given as* $c_v = (2v+1)^2$, $v = 1, ..., \lfloor \frac{R}{2} \rfloor - 1$ *and* $c_{\lfloor \frac{R}{2} \rfloor} = m$ *such that*

$$[\mathbb{G} * \mathbf{F} + \mathbb{B}]_{\lceil \frac{R}{2} \rceil, \lceil \frac{R}{2} \rceil, m} = [\mathbb{G}_{\lfloor \frac{R}{2} \rfloor} * f_{\lfloor \frac{R}{2} \rfloor - 1}(\mathbf{F}) + \mathbb{B}_{\lfloor \frac{R}{2} \rfloor}]_{\lceil \frac{R}{2} \rceil, \lceil \frac{R}{2} \rceil, m}, \tag{A.52}$$

for any m where $f_0(\mathbf{F}) = \mathbf{F}$ *and*

$$f_v(\mathbf{F}) = \sigma(\mathbb{G}_v * f_{v-1}(\mathbf{F}) + \mathbb{B}_v). \tag{A.53}$$

Proof: From Theorem 3.5 for $c_v = (2v+1)^2$ and $v = 1, ..., \lfloor \frac{R}{2} \rfloor - 1$, there exists $\mathbb{G}_v^1 \in \mathbb{R}^{3x3xc_{v-1}xc_v}$ and $\mathbb{G}^2 \in \mathbb{R}^{3x3x(2\lfloor \frac{R}{2} \rfloor - 1)^2)xm}$ such that

$$\mathbb{G} * \mathbf{F} = \mathbb{G}^2 * \mathbb{G}^1_{\lfloor \frac{R}{2} \rfloor - 1} * \cdots * \mathbb{G}^1_1 * \mathbf{F}, \tag{A.54}$$

for $\mathbf{F} \in \Delta$. From equation (A.54), we can say that the series of filters $\mathbb{G}_v = \mathbb{G}_v^1$, where $\mathbb{G}_{\lfloor \frac{R}{2} \rfloor} = \mathbb{G}^2$.

The vectors of unique biases B_v^1 can be defined as

$$[B_v^1]_f = \max_{1 \le r,c \le R} \sup_{\mathbf{F} \in \Delta} |[\mathbb{G}_v * f_{v-1}(\mathbf{F})]_{r,c,f}|, \tag{A.55}$$

where $f = 1, ..., c_v$.

Since Δ is bounded and the ReLu activation function σ is continuous, then $[B_v^1]_f < \infty$ for all v and f. This means that from equation (A.53), it follows that

$$f_v(\mathbf{F}) = \sigma(\mathbb{G}_v * f_{v-1}(\mathbf{F}) + \mathbb{B}_v) = \mathbb{G}_v * f_{v-1}(\mathbf{F}) + \mathbb{B}_v. \tag{A.56}$$

Also, note that we have

$$\mathbb{G}_{\frac{R}{2}} * f_{\frac{R}{2}-1}(\mathbf{F}) = \mathbb{G}^2 * \mathbb{G}^1_{\lfloor \frac{R}{2} \rfloor - 1} * \cdots * \mathbb{G}^1_1 * \mathbf{F} + \mathbb{B} = \mathbb{G} * \mathbf{F} + \mathbb{B}, \tag{A.57}$$

where

$$\mathbb{B} = \sum_{v=2}^{\frac{R}{2}-1} \mathbb{G}^2 * \mathbb{G}^1_{\lfloor \frac{R}{2} \rfloor - 1} * \cdots * \mathbb{G}^1_v * \mathbb{B}_{v-1} + \mathbb{G}^2 * \mathbb{B}_{\frac{R}{2}-1} \in \mathbb{R}^{RxRxm}. \tag{A.58}$$

Taking $[\mathbb{B}^1_{\frac{R}{2}}]_m = [\mathbb{B}^1]_m - [\mathbb{B}]_{\lfloor \frac{R}{2} \rfloor, \lfloor \frac{R}{2} \rfloor, m}$ for any m concludes the proof. ∎

Theorem 3.7: *Given that* $f : \Delta \subset \mathbb{R}^R \mapsto \mathbb{R}$ *and that* Δ *is a bounded set with* $\|f\|_{\mathcal{K}(\mathcal{Y})} < \infty$, *then there exists a CNN function* $f_m : \mathbb{R}^R \mapsto \mathbb{R}$ *with size (3 × 3) filters for number of feature maps per layer* $c_v = (2v+1)^2$ *and layer number* $v = 1, ..., \lfloor \frac{R}{2} \rfloor - 1$, *where* $c_{\lfloor \frac{R}{2} \rfloor} = m$, *such that*

$$\|f - f_m\|_{L^2(\Delta)} \lessapprox m^{-\frac{1}{2} - \frac{3}{2R^2}} \|f\|_{\mathcal{K}(\mathbb{D})}. \tag{A.59}$$

Proof: Given that $\mathbb{F} \in \Delta$, we assume that $f_m^{NN}(\text{vec}(\mathbb{F}))$ is the approximation of $f(\mathbb{F})$ using an ANN with one hidden layer as in Theorem 3.6. Using Lemmas 3.3 and 3.4, we can say that there is a CNN function f_m^{CNN} with multiple filters of size (3 × 3) such that $f_m^{NN}(\text{vec}(\mathbb{F})) = f_m^{CNN}(\mathbb{F})$. ∎

Appendix B

B1: Tensor Properties and More Definitions

Definition B1: *The Hadamard product of two matrices* $\mathbf{B}$ and $\mathbf{D}$ *of the same size, which is represented by the* $*$ *operator, is defined as the element-wise matrix product and is given as follows*

$$\mathbf{B} * \mathbf{D} = \begin{bmatrix} b_{11}d_{11} & b_{12}d_{12} & \cdots & b_{1C_2}d_{1C_2} \\ b_{21}d_{21} & b_{22}d_{22} & \cdots & b_{2C_2}d_{2C_2} \\ \vdots & \vdots & \ddots & \vdots \\ b_{C_11}d_{C_11} & b_{C_12}d_{C_12} & \cdots & b_{C_1C_2}d_{C_1C_2} \end{bmatrix}, \tag{B.1}$$

where $\mathbf{B} * \mathbf{D} \in \mathbb{R}^{C_1 \times C_2}$ is of the same size as $\mathbf{B}$ and $\mathbf{D}$.

Proposition B1: *For the Kronecker product, given* $\mathbf{F}$ and $\mathbf{G}$ *defined as in Section 4.2.2, then*

$$(\mathbf{F} \otimes \mathbf{G})^{\dagger} = \mathbf{F}^{\dagger} \otimes \mathbf{G}^{\dagger}, \tag{B.2}$$

where the 'obelus' $\dagger$ *symbol is a commonly used notation in theory to represent the Moore–Penrose pseudoinverse or just pseudoinverse in short (Golub and Loan, 1986). Given matrices* $\mathbf{B}, \mathbf{G}$ *and* $\mathbf{E}$ *defined as in Section 4.2.2, a property which connects vectorization with the Kronecker product is the following:*

$$\text{vec}(\mathbf{BGE}) = (\mathbf{E}^{\mathrm{T}} \otimes \mathbf{B})\text{vec}(\mathbf{G}). \tag{B.3}$$

Proposition B2: *For the Khatri-Rao product, the following property makes use of the Hadamard product: given* $\mathbf{F}$ *and* $\mathbf{G}$ *defined as in Section 4.2.1, then*

$$(\mathbf{F} \odot \mathbf{G})^{\dagger} = ((\mathbf{F}^{\mathrm{T}}\mathbf{F}) * (\mathbf{G}^{\mathrm{T}}\mathbf{G}))^{\dagger}(\mathbf{F} \odot \mathbf{G})^{\mathrm{T}}. \tag{B.4}$$

Proposition B3: *For the Kronecker product, given two vectors* $w_1 \in \mathbb{R}^{R_1}$ *and* $w_2 \in \mathbb{R}^{R_2}$*, then it follows that*

$$\text{vec}(\mathbf{w}_1 \mathbf{w}_2^{\mathrm{T}}) = \mathbf{w}_2 \otimes \mathbf{w}_1. \tag{B.5}$$

Proposition B4: *Given an order O tensor* $\mathbb{X}$*, tensor* $\mathbb{H} \in \mathbb{R}^{F_1 \times \cdots \times F_O}$*, matrices* $\mathbf{A}_n \in \mathbb{R}^{R_n \times F_n}$ *for* $n = 1, ..., O$*, and two ordered sets* $\boldsymbol{r} = \{r_1, ..., r_l\}$ *and* $\boldsymbol{c} = \{c_1, ..., c_m\}$ *that partition the modes* $1, ..., O$ *of* $\mathbb{X}$*, then we have*

$$\mathbb{X} = [\![\mathbb{H}; \mathbf{A}_1, ..., \mathbf{A}_O]\!] \iff \mathbb{X}_{(\mathbf{r} \times \mathbf{c})} = (\mathbf{A}_{r_l} \cdot \cdots \cdot \mathbf{A}_{r_1}) \mathbb{H}_{(\mathbf{r} \times \mathbf{c})} (\mathbf{A}_{c_m} \cdot \cdots \cdot \mathbf{A}_{c_1})^{\mathrm{T}}. \tag{B.6}$$

Also, given that we have the mode-n matricizations of $\mathbb{X}$ *and* $\mathbb{H}$*, then for any* $n \in \mathbb{N}$ *we have*

$$\mathbb{X} = [\![\mathbb{H}; \mathbf{A}_1, ..., \mathbf{A}_O]\!] \iff \mathbb{X}_{(n)} = \mathbf{A}_n \mathbb{H}_{(n)} (\mathbf{A}_O \cdot \cdots \cdot \mathbf{A}_{n+1} \cdot \mathbf{A}_{n-1} \cdot \cdots \cdot \mathbf{A}_1)^{\mathrm{T}}. \tag{B.7}$$

Proposition B5: *Given two order O tensors* $\mathbb{X}$ *and* $\mathbb{Y}$*, then we have*

$$\|\mathbb{X} - \mathbb{Y}\|^2 = \|\mathbb{X}\|^2 - 2\langle \mathbb{X}, \mathbb{Y} \rangle - \|\mathbb{Y}\|^2. \tag{B.8}$$

Proposition B6: *Given two order O tensors* $\mathbb{X} \in \mathbb{R}^{R_1 \times \cdots \times R_{n-1} \times J \times R_{n+1} \times \cdots \times R_O}$*,* $\mathbb{Y} \in \mathbb{R}^{R_1 \times \cdots \times R_{n-1} \times K \times R_{n+1} \times \cdots \times R_O}$ *and a matrix* $\mathbb{A} \in \mathbb{R}^{J \times K}$*, then we have*

$$\langle \mathbb{X}, \mathbb{Y} \times_n \mathbb{A} \rangle = \langle \mathbb{X} \times_n \mathbb{A}^{\mathrm{T}}, \mathbb{Y} \rangle. \tag{B.9}$$

Proposition B7: *Given an order O tensor* $\mathbb{X} \in \mathbb{R}^{R_1 \times \cdots \times R_O}$ *and an orthonormal matrix* $\mathbf{A} \in \mathbb{R}^{J \times R_n}$*, then we have*

$$\|\mathbb{X}\| = \|\mathbb{X} \times_n \mathbf{A}\|. \tag{B.10}$$

Proposition B8 (CP Decomposition of a Tensor using ALS Proof): Using the ALS approach, at each iteration, all but one of the factor matrices will be fixed. Given that the

rank of an order O tensor $\mathbb{X}$ is $Z > 0$ and all factor matrices $\mathbf{A}_m$, where $m \neq N$ and $m, N \leq O$, are fixed where we are solving for $\mathbf{A}_N$, this means that the subproblem from equation (4.26) can be defined as follows at each iteration

$$\min_{\mathbf{A}_N} \|\mathbb{X} - [\![\mathbf{A}_1, ..., \mathbf{A}_N, ..., \mathbf{A}_O]\!]\|. \tag{B.11}$$

Using equation (4.25), equation (B.11) can be expressed in matrix form as

$$\min_{\mathbf{A}_N} \|\mathbb{X}_{(N)} - \mathbf{A}_N(\mathbf{A}_O \odot \cdots \odot \mathbf{A}_{N+1} \odot \mathbf{A}_{N-1} \odot \cdots \odot \mathbf{A}_1)^{\mathrm{T}}\|, \tag{B.12}$$

which is a classic least squares problem. We can derive the optimal solution to be

$$\mathbf{A}_N = \mathbb{X}_{(N)}[(\mathbf{A}_O \odot \cdots \odot \mathbf{A}_{N+1} \odot \mathbf{A}_{N-1} \odot \cdots \odot \mathbf{A}_1)^{\mathrm{T}}]^{\dagger}, \tag{B.13}$$

which, when using Proposition B2, we could rewrite to

$$\mathbf{A}_N = \mathbb{X}_{(N)}(\mathbf{A}_O \odot \cdots \odot \mathbf{A}_{N+1} \odot \mathbf{A}_{N-1} \odot \cdots \odot \mathbf{A}_1)^{\mathrm{T}}\mathbf{V}^{\dagger}, \tag{B.14}$$

where

$$\mathbf{V} = (\mathbf{A}_O^{\mathrm{T}}\mathbf{A}_O) * \cdots * (\mathbf{A}_{N+1}^{\mathrm{T}}\mathbf{A}_{N+1}) * (\mathbf{A}_{N-1}^{\mathrm{T}}\mathbf{A}_{N-1}) * \cdots * (\mathbf{A}_1^{\mathrm{T}}\mathbf{A}_1) \tag{B.15}$$

is a symmetric matrix of size $(Z \times Z)$ (Kolda (2006)). The advantage of expressing the pseudoinverse from equation (B.13) to (B.14) means that we will only require calculating the pseudoinverse of a matrix of size $(Z \times Z)$. At each iteration and for $N = 1, ..., O$, the pseudoinverse of $\mathbf{V}$ would be calculated first before the calculation of $\mathbf{A}_N$. Note that the factor matrices would need to be initialized in some way, where one way is to start with random values or by using a higher-order generalization of SVD (Rasmus (2001)).

Definition B2: An order O tensor $\mathbb{X}$ is said to be *superdiagonal* if, given element $(r_1, r_2, ..., r_O)$ of $\mathbb{X}$, then

$$x_{r_1 r_2 ... r_O} = \begin{cases} 1, & \text{when } r_1 = r_2 = \cdots = r_O \\ 0, & \text{otherwise.} \end{cases} \tag{B.16}$$

B2: Tucker Decomposition

Tucker decomposition is seen as a form of higher-order Principal Component Analysis (PCA), where it decomposes a tensor into a *core tensor* and multiple factor matrices (Rabanser et al., 2017). When using Tucker decomposition, the core tensor would be multiplied by a factor matrix along each mode, where the factor matrices here are a

similar idea to the principal components in each respective mode. The entries of a core tensor represent the level of interaction between the different components (or columns) of the factor matrices (Kolda and Bader, 2009). Given an order 3 tensor $\mathbb{X}$ and that we have pre-specified values P_N such that $1 \leq P_N \leq R_N$ for $N = 1, 2,$ and 3, then it can be approximated by

$$\mathbb{X} \approx \mathbb{H} \times_1 \mathbf{A} \times_2 \mathbf{B} \times_3 \mathbf{C} = [[\mathbb{H}; \mathbf{A}, \mathbf{B}, \mathbf{C}]] = \sum_{p_1=1}^{P_1} \sum_{p_2=1}^{P_2} \sum_{p_3=1}^{P_3} h_{p_1 p_2 p_3} \mathbf{a}_{p_1} \circ \mathbf{b}_{p_2} \circ \mathbf{c}_{p_3}, \tag{B.17}$$

where $\mathbb{H} \in \mathbb{R}^{P_1 \times P_2 \times P_3}$ is the core tensor, and $\mathbf{A} = [\mathbf{a}_1, ..., \mathbf{a}_{P_1}] \in \mathbb{R}^{R_1 \times P_1}$, $\mathbf{B} = [\mathbf{b}_1, ..., \mathbf{b}_{P_2}] \in \mathbb{R}^{R_2 \times P_2}$, $\mathbf{C} = [\mathbf{c}_1, ..., \mathbf{c}_{P_3}] \in \mathbb{R}^{R_3 \times P_3}$ are the factor matrices with P_1, P_2 and P_3 components, respectively. Element-wise, for element (r_1, r_2, r_3) of $\mathbb{X}$, equation (B.17) can be written as

$$x_{r_1 r_2 r_3} \approx \sum_{p_1=1}^{P_1} \sum_{p_2=1}^{P_2} \sum_{p_3=1}^{P_3} h_{p_1 p_2 p_3} a_{r_1 p_1} b_{r_2 p_2} c_{r_3 p_3}. \tag{B.18}$$

Figure B.1 represents a visual illustration of equation (B.17).

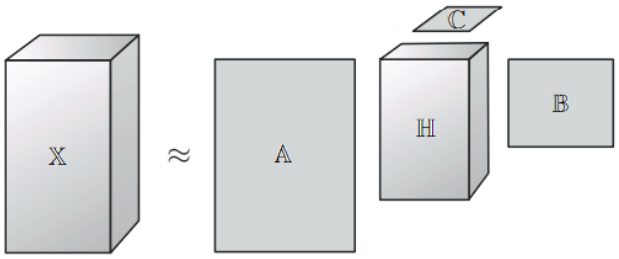


Figure B.1: Figure visualizing the Tucker decomposition of an order 3 tensor $\mathbb{X}$. Taken from Kolda and Bader (2009).

In the case that we have $P_1 < R_1$, $P_2 < R_2$ and $P_3 < R_3$, then the Tucker decomposition of $\mathbb{X}$ will result in a 'compression' of $\mathbb{X}$, where most of the variance of the tensor is preserved (Austin et al., 2016). The compression of large-scale tensor data using Tucker decomposition is a potent tool that is used in practice for more efficient data storage, where the storage of the decomposed version of the tensor would be much smaller than the original tensor (Kolda and Bader, 2009).

As in CP decomposition, equation (B.17) can be written in matricized form according to the mode-1, mode-2, or mode-3 matricizations of $\mathbb{X}$, respectively, i.e.,

$$\begin{aligned} \mathbb{X}_{(1)} &\approx \mathbf{A}\mathbb{H}_{(1)}(\mathbf{C} \otimes \mathbf{B})^{\mathrm{T}}, \\ \mathbb{X}_{(2)} &\approx \mathbf{B}\mathbb{H}_{(2)}(\mathbf{C} \otimes \mathbf{A})^{\mathrm{T}}, \\ \mathbb{X}_{(3)} &\approx \mathbf{C}\mathbb{H}_{(3)}(\mathbf{B} \otimes \mathbf{A})^{\mathrm{T}}, \end{aligned} \tag{B.19}$$

where $\mathbb{H}_{(1)}$, $\mathbb{H}_{(2)}$ and $\mathbb{H}_{(3)}$ are the mode-1, mode-2, and mode-3 matricizations of $\mathbb{H}$, respectively.

When generalizing to an order O tensor $\mathbb{X}$ with pre-specified values of P_N such that $1 \leq P_N \leq R_N$ for $N = 1, ..., O$ (Kapteyn et al., 1986), then the Tucker decomposition of $\mathbb{X}$ is given by

$$\begin{aligned} \mathbb{X} &\approx \mathbb{H} \times_1 \mathbf{A}_1 \times_2 \mathbf{A}_2 \times_3 \cdots \times_O \mathbf{A}_O \\ &= [\![\mathbb{H}; \mathbf{A}_1, ..., \mathbf{A}_O]\!] \\ &= \sum_{p_1=1}^{P_1} \cdots \sum_{p_O=1}^{P_O} h_{p_1 \cdots p_O} \mathbf{a}_{1,p_1} \circ \cdots \circ \mathbf{a}_{O,p_O}, \end{aligned} \tag{B.20}$$

where $\mathbb{H} \in \mathbb{R}^{P_1 \times \cdots \times P_O}$ and the factor matrices $\mathbf{A}_1 = [\mathbf{a}_{1,1}, ..., \mathbf{a}_{1,P_1}] \in \mathbb{R}^{R_1 \times P_1}, ..., \mathbf{A}_O = [\mathbf{a}_{O,1}, ..., \mathbf{a}_{O,P_O}] \in \mathbb{R}^{R_O \times P_O}$. The mode-$N$ matricization for equation (B.20) is given by

$$\mathbb{X}_{(N)} \approx \mathbf{A}_N \mathbb{H}_{(N)} (\mathbf{A}_O \otimes \cdots \otimes \mathbf{A}_{N+1} \otimes \mathbf{A}_{N-1} \otimes \cdots \otimes \mathbf{A}_1)^{\mathrm{T}}. \tag{B.21}$$

Given an order O tensor $\mathbb{X}$ and $P_1, ..., P_O$, the goal is to find a set of factor matrices $\mathbf{A}_1, ..., \mathbf{A}_O$ and core tensor $\mathbb{H}$ which satisfy the following optimization problem

$$\min_{\mathbb{H}, \mathbf{A}_1, ..., \mathbf{A}_O} \|\mathbb{X} - [\![\mathbb{H}; \mathbf{A}_1, ..., \mathbf{A}_O]\!]\|, \text{ subject to } \mathbb{H} \in \mathbb{R}^{P_1 \times \cdots \times P_O}, \tag{B.22}$$

where the factor matrices are columnwise orthogonal. Given that the N-rank of $\mathbb{X}$ is $P_N = F_N$ such that $F_N = \text{rank}_N(\mathbb{X})$ for $N = 1, ..., O$, then $\mathbb{X}$ is a rank-$(F_1, ..., F_O)$ tensor and its Tucker decomposition will be exact and trivial (Kolda, 2006). However, in the case when $F_N < \text{rank}_N(\mathbb{X})$ for one or more N, then the Tucker decomposition will be harder to compute, where this version of Tucker decomposition is referred to as the *truncated* version. Since the values F_N need to be pre-specified, Kiers and der Kinderen (2003) defined a procedure for choosing them based on *Higher-Order SVD* (HOSVD) and we refer to their paper for more information regarding this.

There are several methods to compute the Tucker decomposition. One method is based on a generalization of matrix SVD to tensors which is known as the HOSVD method by (Lathauwer et al., 2000a). The idea of HOSVD is to find the components in mode N that best capture the variation among all components in that mode, where this is done independent of the other modes (Tucker (1966), Lathauwer et al. (2000a)). Another popular method that is used is the *Higher-Order Orthogonal Iteration* (HOOI) method, which is simply the ALS algorithm for Tucker decomposition (Liu et al., 2023). It has a similar approach as we saw with CP decomposition, where all but one of the factor matrices are fixed each time and the objective function in equation (B.22) is minimized. Also, it is not guaranteed to converge to the global optimum or even a stationary point (Lathauwer et al., 2000b).

The Tucker operator can be expressed in terms of matricized tensors and the Kronecker product. We refer to Proposition B4 in Appendix B for more information on these expressions. Using Proposition B4, the objective function presented in equation (B.22) can be rewritten in vectorized form as follows

$$\|\mathbb{X} - [[\mathbb{H}; \mathbf{A}_1, ..., \mathbf{A}_O]]\| = \|\text{vec}(\mathbb{X}) - (\mathbf{A}_O \otimes \cdots \otimes \mathbf{A}_1)\text{vec}(\mathbb{H})\|, \tag{B.23}$$

whose solution is given by

$$\text{vec}(\mathbb{H}) = (\mathbf{A}_O \otimes \cdots \otimes \mathbf{A}_1)^{\dagger}\text{vec}(\mathbb{X}). \tag{B.24}$$

Since the matrices are orthonormal, then using equation (B.2) we have

$$\text{vec}(\mathbb{H}) = (\mathbf{A}_O^{\mathrm{T}} \otimes \cdots \otimes \mathbf{A}_1^{\mathrm{T}})\text{vec}(\mathbb{X}). \tag{B.25}$$

Using Proposition B4 again, then it follows that we have

$$\mathbb{H} = \mathbb{X} \times_1 \mathbf{A}_1^{\mathrm{T}} \times_2 \mathbf{A}_2^{\mathrm{T}} \times_3 \cdots \times_O \mathbf{A}_O^{\mathrm{T}} = [[\mathbb{X}; \mathbf{A}_1^{\mathrm{T}}, ..., \mathbf{A}_O^{\mathrm{T}}]]. \tag{B.26}$$

The objective function in equation (B.22) can be rewritten as a maximization problem. For this, we consider the following squared objective function:

$$\|\mathbb{X} - [[\mathbb{H}; \mathbf{A}_1, ..., \mathbf{A}_O]]\|^2. \tag{B.27}$$

Using Proposition B5, we have that

$$\|\mathbb{X} - [[\mathbb{H}; \mathbf{A}_1, ..., \mathbf{A}_O]]\|^2 = \|\mathbb{X}\|^2 - 2\langle \mathbb{X}, [[\mathbb{H}; \mathbf{A}_1, ..., \mathbf{A}_O]]\rangle + \|[[\mathbb{H}; \mathbf{A}_1, ..., \mathbf{A}_O]]\|^2. \tag{B.28}$$

The middle term in equation (B.28) can be simplified using Proposition B6 as follows

$$\langle \mathbb{X}, [[\mathbb{H}; \mathbf{A}_1, ..., \mathbf{A}_O]]\rangle = \langle [[\mathbb{X}; \mathbf{A}_1^{\mathrm{T}}, ..., \mathbf{A}_O^{\mathrm{T}}]], \mathbb{H}\rangle = \|\mathbb{H}\|^2. \tag{B.29}$$

The last term in equation (B.28) can be simplified using Proposition B7 as follows

$$\|[[\mathbb{H}; \mathbf{A}_1, ..., \mathbf{A}_O]]\| = \|\mathbb{H}\|. \tag{B.30}$$

Hence, inputting equations (B.29) and (B.30) into (B.28) gives us the following

$$\|\mathbb{X} - [[\mathbb{H}; \mathbf{A}_1, ..., \mathbf{A}_O]]\|^2 = \|\mathbb{X}\|^2 - \|\mathbb{H}\|^2, \tag{B.31}$$

where $\|\mathbb{X}\|^2$ is constant. As such, the optimization problem from equation (B.22) can be reformulated to a maximization problem using equation (B.31), i.e.,

$$\max_{\mathbf{A}_1, ..., \mathbf{A}_O} \|[[\mathbb{X}; \mathbf{A}_1^{\mathrm{T}}, ..., \mathbf{A}_O^{\mathrm{T}}]]\|, \tag{B.32}$$

where similarly the factor matrices are columnwise orthogonal. Using an ALS approach, all factor matrices $\mathbf{A}_m$, where $m \neq N$ and $m, N \leq O$, are fixed excluding $\mathbf{A}_N$ and we look to solve for $\mathbf{A}_N$ from equation (B.32). From here, the objective function in equation (B.32) can be simplified further. Defining $\mathbb{Z} \equiv [\![\mathbb{X}; \mathbf{A}_1^{\mathrm{T}}, ..., \mathbf{A}_{N-1}^{\mathrm{T}}, \mathbb{I}, \mathbf{A}_{N+1}^{\mathrm{T}}, .., \mathbf{A}_O^{\mathrm{T}}]\!]$, then equation (B.32) becomes

$$\max_{\mathbf{A}_N} \|\mathbb{Z} \times_N \mathbf{A}_N^{\mathrm{T}}\| \equiv \|\mathbf{A}_N^{\mathrm{T}} \mathbb{Z}_{(N)}\|, \tag{B.33}$$

where $\mathbb{Z}_{(N)} = \mathbb{X}_{(N)}(\mathbf{A}_O \otimes \cdots \otimes \mathbf{A}_{N+1} \otimes \mathbf{A}_{N-1} \otimes \cdots \otimes \mathbf{A}_1)$. The solution for the subproblem in equation (B.33) can be found by using the matrix SVD of $\mathbb{Z}_{(N)}$. The factor matrices $\mathbf{A}_1, ..., \mathbf{A}_O$ need to be initialized, where generally they are initialized using HOSVD (Kolda and Bader (2009)).

Note that CP decomposition can be seen as a special case of Tucker decomposition when the core tensor is superdiagonal and we have $P_1 = P_2 = ... = P_O$, where this implies that the factor matrices $\mathbf{A}_1, ..., \mathbf{A}_O$ would all have the same number of columns. Given that we have both of these constraints as assumptions and working element-wise for element $(r_1, ..., r_O)$ of $\mathbb{X}$ and element $(p_1, ..., p_O)$ of $\mathbb{H}$, then when opening the summations of equation (B.20) there would be non-zero $h_{p_1 \cdots p_O}$ only in the diagonal values of the core tensor since $P_1 = P_2 = ... = P_O$ and $\mathbb{H}$ is superdiagonal. This would simplify to as follows

$$\begin{aligned} x_{r_1 \cdots r_O} &\approx \sum_{p_1=1}^{P_1} \cdots \sum_{p_O=1}^{P_O} h_{p_1 \cdots p_O} a_{r_1 p_1} \cdots a_{r_O p_O} \\ &= \sum_{p_1=1}^{P_1} h_{p_1 \cdots p_1} a_{r_1 p_1} \cdots a_{r_O p_1}, \end{aligned} \tag{B.34}$$

where $h_{p_1 \cdots p_1} = 1$ which means that $[\![\mathbf{A}_1, ..., \mathbf{A}_O]\!]$ is a CP decomposition of $\mathbb{X}$. Since Tucker decomposition does not assume that $P_1 = P_2 = ... = P_O$, this allows the value P_N to differ along the different modes with the factor matrices having differing number of columns, which allows more flexibility compared to CP decomposition. This flexibility is useful in certain situations, such as when the tensor is skewed in dimensions (for example, we have two dimensions but one dimension is far larger than the other) (Li, 2014).

Given the order 3 tensor $\mathbb{X} \in \mathbb{R}^{4 \times 3 \times 2}$ defined in Section 4.2.1, we will show an example using Tucker decomposition. Given that we have $P_1 = 3$, $P_2 = 2$ and $P_3 = 2$, then the Tucker decomposition of $\mathbb{X}$ is given by

$$\mathbb{X} \approx \sum_{p_1=1}^{3} \sum_{p_2=1}^{2} \sum_{p_3=1}^{2} h_{p_1 p_2 p_3} \mathbf{a}_z \circ \mathbf{b}_z \circ \mathbf{c}_z, \tag{B.35}$$

where the core matrix $\mathbb{H} \in \mathbb{R}^{3\times2\times2}$ has front and back frontal slices

$$\mathbb{H}_{::1} = \begin{bmatrix} 69.63 & -0.07 \\ -0.018 & -0.784 \\ -0.000001 & 0.0000006 \end{bmatrix}, \quad \mathbb{H}_{::2} = \begin{bmatrix} -0.011 & -1.61 \\ -6.919 & -0.701 \\ -0.0000002 & -0.0000003 \end{bmatrix},$$

respectively, and the factor matrices $\mathbf{A} \in \mathbb{R}^{4\times3}$, $\mathbf{B} \in \mathbb{R}^{3\times2}$ and $\mathbf{C} \in \mathbb{R}^{2\times2}$ are

$$\mathbf{A} = \begin{bmatrix} 0.344 & 0.762 & -0.235 \\ 0.44 & 0.326 & -0.055 \\ 0.536 & -0.111 & 0.816 \\ 0.632 & -0.548 & -0.525 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0.541 & 0.735 \\ 0.577 & 0.029 \\ 0.612 & -0.677 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 0.353 & 0.935 \\ 0.935 & -0.353 \end{bmatrix}.$$

B3: Uniqueness of Tucker Decomposition

We saw that CP decomposition can be unique under certain conditions; however, Tucker decompositions are not unique (Kolda and Bader, 2009). For example, assume that we have an order 3 tensor $\mathbb{X}$, pre-specified values P_N such that $1 \leq P_N \leq R_N$ for $N = 1$, 2, and 3, factor matrices $\mathbf{A} \in \mathbb{R}^{R_1\times P_1}, \mathbf{B} \in \mathbb{R}^{R_2\times P_2}, \mathbf{C} \in \mathbb{R}^{R_3\times P_3}$ and core tensor $\mathbb{H} \in \mathbb{R}^{P_1\times P_2\times P_3}$ as defined in equation (B.17). Given nonsingular matrices $\mathbf{X} \in \mathbb{R}^{P_1\times P_1}, \mathbf{Y} \in \mathbb{R}^{P_2\times P_2}, \mathbf{Z} \in \mathbb{R}^{P_3\times P_3}$, then

$$\mathbb{X} \approx \mathbb{H} \times_1 \mathbf{A} \times_2 \mathbf{B} \times_3 \mathbf{C} = [\![\mathbb{H}; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!] = [\![\mathbb{H} \times_1 \mathbf{X} \times_2 \mathbf{Y} \times_3 \mathbf{Z}; \mathbf{A}\mathbf{X}^{-1}, \mathbf{B}\mathbf{Y}^{-1}, \mathbf{C}\mathbf{Z}^{-1}]\!], \quad \text{(B.36)}$$

showing that the Tucker decomposition in equation (B.17) is not unique. It follows that the core tensor can be modified without changing the overall fit by also applying the inverse modification to the factor matrices; hence, nonuniqueness follows (Kolda and Bader, 2009). Thus, Tucker decomposition is not identifiable due to this *nonsingular transformation indeterminancy* (Li, 2014). To improve uniqueness, additional constraints on the core tensor can be applied. This includes transformations to simplify the core tensor such that most of its elements are zero, which helps to eliminate interactions of the core tensor with the corresponding components (Tucker, 1966).

B4: Tucker TR and Multinomial TR Models

Alternatively, we can use Tucker decomposition instead of CP decomposition, where we would have a *Tucker TR* model. For pre-specified values of P_N such that $1 \leq P_N \leq R_N$ for $N = 1, ..., O$, we assume that $\mathbb{W}$ follows a Tucker decomposition of the form

$$\mathbb{W} = [\![\mathbb{H}; \mathbf{W}_1, ..., \mathbf{W}_O]\!] = \sum_{p_1=1}^{P_1} \cdots \sum_{p_O=1}^{P_O} h_{p_1\cdots p_O} \mathbf{w}_{1,p_1} \circ \cdots \circ \mathbf{w}_{O,p_O}, \quad \text{(B.37)}$$

where $\mathbb{H} \in \mathbb{R}^{P_1 \times \cdots \times P_O}$ is the core tensor and $\mathbf{W}_1 = [\mathbf{w}_{1,1}, ..., \mathbf{w}_{1,P_1}] \in \mathbb{R}^{R_1 \times P_1}$, ..., $\mathbf{W}_O = [\mathbf{w}_{O,1}, ..., \mathbf{w}_{O,P_O}] \in \mathbb{R}^{R_O \times P_O}$ are the factor matrices with P_1, ..., P_O components, respectively. By substituting equation (B.37) to equation (4.36), we will get

$$g(\mu(\mathbb{X})) = b + \langle \sum_{p_1=1}^{P_1} \cdots \sum_{p_O=1}^{P_O} h_{p_1 \cdots p_O} \mathbf{w}_{1,p_1} \circ \cdots \circ \mathbf{w}_{O,p_O}, \mathbb{X} \rangle. \tag{B.38}$$

This decomposition reduces the number of parameters that need to be estimated to $\sum_{N=1}^{O} P_N R_N$. Similarly, an additional vector of variables $\mathbf{z} \in \mathbb{R}^{R_0}$ can be added in equation (B.38) as in equation (4.39), meaning that equation (B.38) can be updated to

$$g(\mu(\mathbb{X})) = b + \zeta^{\mathrm{T}} \mathbf{z} + \langle \sum_{p_1=1}^{P_1} \cdots \sum_{p_O=1}^{P_O} h_{p_1 \cdots p_O} \mathbf{w}_{1,p_1} \circ \cdots \circ \mathbf{w}_{O,p_O}, \mathbb{X} \rangle. \tag{B.39}$$

In the case of Tucker decomposition for the multinomial TR model defined in equations (4.41) and (4.42), for pre-specified values of P_N such that $1 \leq P_N \leq R_N$ for $N = 1, ..., O$, we assume that $\mathbb{W}^{(r)}$ follows a Tucker decomposition of the form

$$\mathbb{W}^{(r)} = [\![\mathbb{H}^{(r)}; \mathbf{W}_1^{(r)}, ..., \mathbf{W}_O^{(r)}]\!] = \sum_{p_1=1}^{P_1} \cdots \sum_{p_O=1}^{P_O} h_{p_1 \cdots p_O}^{(r)} \mathbf{w}_{1,p_1}^{(r)} \circ \cdots \circ \mathbf{w}_{O,p_O}^{(r)}, \tag{B.40}$$

where $\mathbb{H}^{(r)} \in \mathbb{R}^{P_1 \times \cdots \times P_O}$ is the core tensor and $\mathbf{W}_1^{(r)} \in \mathbb{R}^{R_1 \times P_1}$, ..., and $\mathbf{W}_O^{(r)} \in \mathbb{R}^{R_O \times P_O}$ are the factor matrices with P_1, ..., P_O components, respectively. A tensor decomposition is done for each $\mathbb{W}^{(r)}$ for $r = 1, ..., L-1$, where the model will have a total of $(L-1) * (R_0 + \sum_{N=1}^{O} P_N R_N)$ parameters.

B5: MLE for Tucker TR Model

The overall process follows similarly as in Section 4.2.2.1. The log-likelihood to be maximized will be as follows

$$l(b, \mathbb{H}, \mathbf{W}_1, ..., \mathbf{W}_O | \mathcal{D}) = \sum_{i=1}^{n} \frac{l_i \theta_i - b(\theta_i)}{a(\phi)} + \sum_{i=1}^{n} c(l_i, \phi), \tag{B.41}$$

where similarly as before, θ_i is related to the regression parameters $(b, \mathbb{H}, \mathbf{W}_1, ..., \mathbf{W}_O)$ through equation (4.36).

It follows that equation (4.36) is not linear in $(\mathbb{H}, \mathbf{W}_1, ..., \mathbf{W}_O)$ jointly, but linear in them individually. Using the block relaxation algorithm, we look to alternately update each factor matrix and the core tensor whilst fixing all other parameters. At an iteration $q+1$ and for $N = 1, ..., O$, we would update $\mathbb{W}_N$ using

$$\mathbf{W}_N^{(q+1)} = \mathrm{argmax}_{\mathbf{W}_N} \, l \left(b^{(q)}, \mathbb{H}^{(q)}, \mathbf{W}_1^{(q+1)}, \ldots, \mathbf{W}_{N-1}^{(q+1)}, \mathbf{W}_N, \mathbf{W}_{N+1}^{(q)}, ..., \mathbf{W}_O^{(q)} | \mathcal{D} \right), \tag{B.42}$$

where after all of $N = 1, ..., O$ is covered, then we would update $\mathbb{H}$ and b using

$$\mathbb{H}^{(q+1)} = \text{argmax}_{\mathbb{H}}\, l\left(b^{(q)}, \mathbb{H}, \mathbf{W}_1^{(q+1)}, \ldots, \mathbf{W}_{N-1}^{(q+1)}, \mathbf{W}_N^{(q+1)}, \mathbf{W}_{N+1}^{(q+1)}, ..., \mathbf{W}_O^{(q+1)} | \mathcal{D}\right), \quad \text{(B.43)}$$

and

$$b^{(q+1)} = \text{argmax}_{b}\, l\left(b, \mathbb{H}^{(q+1)}, \mathbf{W}_1^{(q+1)}, \ldots, \mathbf{W}_{N-1}^{(q+1)}, \mathbf{W}_N^{(q+1)}, \mathbf{W}_{N+1}^{(q+1)}, ..., \mathbf{W}_O^{(q+1)} | \mathcal{D}\right), \quad \text{(B.44)}$$

respectively.

When fixing all other parameters except for $\mathbf{W}_N$ and using Tucker decomposition as in equation (B.37), we can see that equation (4.36) can be simplified to

$$\begin{aligned} g(\mu(\mathbb{X})) &= b + \langle \mathbb{X}, \mathbb{W} \rangle \\ &= b + \langle \mathbb{X}_{(N)}, \mathbb{W}_{(N)} \rangle \\ &= b + \langle \mathbb{X}_{(N)}, \mathbf{W}_N \mathbb{H}_{(N)} (\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_{N+1} \otimes \mathbf{W}_{N-1} \otimes \cdots \otimes \mathbf{W}_1)^{\mathrm{T}} \rangle \\ &= b + \langle \mathbb{X}_{(N)} (\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_{N+1} \otimes \mathbf{W}_{N-1} \otimes \cdots \otimes \mathbf{W}_1) \mathbb{H}_{(N)}^{\mathrm{T}}, \mathbf{W}_N \rangle, \end{aligned} \quad \text{(B.45)}$$

since we can substitute the mode-N matricization $\mathbb{W}_{(N)}$ as in equation (B.21). Similarly, when updating $\mathbb{H}$ whilst fixing all other parameters, we would have

$$\begin{aligned} g(\mu(\mathbb{X})) &= b + \langle \mathbb{X}, \mathbb{W} \rangle \\ &= b + \langle \text{vec}(\mathbb{X}), \text{vec}(\mathbb{W}) \rangle \\ &= b + \langle \text{vec}(\mathbb{X}), \text{vec}(\mathbb{W}_{(1)}) \rangle \\ &= b + \langle \text{vec}(\mathbb{X}), \text{vec}(\mathbf{W}_1 \mathbb{H}_{(1)} (\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_2)^{\mathrm{T}}) \rangle, \end{aligned} \quad \text{(B.46)}$$

since we know that $\text{vec}(\mathbb{W}) = \text{vec}(\mathbb{W}_{(1)})$ from equation (4.12). Using Proposition B1 and $\text{vec}(\mathbb{H}) = \text{vec}(\mathbb{H}_{(1)})$, this means that equation (B.46) simplifies to

$$\begin{aligned} g(\mu(\mathbb{X})) &= b + \langle \text{vec}(\mathbb{X}), (\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_1) \text{vec}(\mathbb{H}) \rangle \\ &= b + \langle (\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_1)^{\mathrm{T}} \text{vec}(\mathbb{X}), \text{vec}(\mathbb{H}) \rangle. \end{aligned} \quad \text{(B.47)}$$

The sub-problem considered when only updating $\mathbf{W}_N$ is a classical GLM problem with parameter term $\mathbf{W}_N$ and $\mathbb{X}_{(N)} (\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_{N+1} \otimes \mathbf{W}_{N-1} \otimes \cdots \otimes \mathbf{W}_1) \mathbb{H}_{(N)}^{\mathrm{T}}$ as the predictor term, where this consists of $P_N R_N$ parameters. Similarly, as in equation (B.47), we will have a classical GLM problem with parameter term $\text{vec}(\mathbb{H})$ and $(\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_1)^{\mathrm{T}} \text{vec}(\mathbb{X})$ as the predictor term with a total of $\prod_N P_N$ parameters. The covariates $\mathbf{z}$ can be included here as well and would update similarly as in the CP TR model. The regression parameters are initialized similarly as in Section 4.4.2.1 with the core tensor also being randomly initialized. The values P_N used in Tucker decomposition need to be pre-specified and estimated beforehand.

The convergence properties of the block relaxation algorithm follow similarly as in Proposition 4.1 from Section 4.4.2.1, although the sequence of parameters is updated to be $\theta = (b, \zeta, \mathbb{H}, \mathbf{W}_1, ..., \mathbf{W}_O)$ (Li, 2014).

B6: MLE for Multinomial Tucker TR Model

In the case of Tucker decomposition with pre-specified values of P_N such that $1 \leq P_N \leq R_N$ for $N = 1, ..., O$, given order O tensors $\mathbb{X}_i$ and the binary indicators of the the L classes $l_{i1}, ..., l_{iL}$ for $i = 1, ..., n$, then the parameters

$$\theta = (b^{(1)}, \mathbb{H}^{(1)}, \mathbf{W}_1^{(1)}, ..., \mathbf{W}_O^{(1)}, ..., b^{(L-1)}, \mathbb{H}^{(L-1)}, \mathbf{W}_1^{(L-1)}, ..., \mathbf{W}_O^{(L-1)})$$

are estimated by maximizing the following log-likelihood:

$$l(\theta) = \sum_{i=1}^{n} \sum_{r=1}^{L} l_{ir} \ln(p_i^{(r)}), \tag{B.48}$$

where

$$p_i^{(r)} = \frac{\exp(b^{(r)} + \langle [\![\mathbb{H}^{(r)}; \mathbf{W}_1^{(r)}, ..., \mathbf{W}_O^{(r)}]\!], \mathbb{X}_i \rangle)}{1 + \exp(b^{(1)} + \langle [\![\mathbb{H}^{(1)}; \mathbf{W}_1^{(1)}, ..., \mathbf{W}_O^{(1)}]\!], \mathbb{X}_i \rangle) + ... + \exp(b^{(L-1)} + \langle [\![\mathbb{H}^{(L-1)}; \mathbf{W}_1^{(L-1)}, ..., \mathbf{W}_O^{(L-1)}]\!], \mathbb{X}_i \rangle)}, \tag{B.49}$$

for $r = 1, ..., L-1$ and

$$p_i^{(L)} = \frac{1}{1 + \exp(b^{(1)} + \langle [\![\mathbb{H}^{(1)}; \mathbf{W}_1^{(1)}, ..., \mathbf{W}_O^{(1)}]\!], \mathbb{X}_i \rangle) + ... + \exp(b^{(L-1)} + \langle [\![\mathbb{H}^{(L-1)}; \mathbf{W}_1^{(L-1)}, ..., \mathbf{W}_O^{(L-1)}]\!], \mathbb{X}_i \rangle)}, \tag{B.50}$$

for $r = L$.

Finding the partial derivative of the log-likelihood function from equation (B.48) with respect to the bias vector and each factor matrix proceeds similarly as in CP decomposition but with P_1, ..., P_O instead of Z, and $F_{i,1}^{(r)}$ and $F_{i,2}$ given as

$$F_{i,1}^{(r)} = \exp(b^{(r)} + \langle [\![\mathbb{H}^{(r)}; \mathbf{W}_1^{(r)}, ..., \mathbf{W}_O^{(r)}]\!], \mathbb{X}_i \rangle), \tag{B.51}$$

for classes $r = 1, ..., L-1$ and

$$F_{i,2} = 1 + \exp(b^{(1)} + \langle [\![\mathbb{H}^{(1)}; \mathbf{W}_1^{(1)}, ..., \mathbf{W}_O^{(1)}]\!], \mathbb{X}_i \rangle) + ... + \exp(b^{(L-1)} + \langle [\![\mathbb{H}^{(L-1)}; \mathbf{W}_1^{(L-1)}, ..., \mathbf{W}_O^{(L-1)}]\!], \mathbb{X}_i \rangle). \tag{B.52}$$

By representing the following calculation in short as

$$F_{i,[p_N,r_N]} = \sum_{p_1=1}^{P_1} \cdots \sum_{p_{N-1}=1}^{P_{N-1}} \sum_{p_{N+1}=1}^{P_{N+1}} \cdots \sum_{p_O=1}^{P_O} \sum_{r_1=1}^{R_1} \cdots \sum_{r_{N-1}=1}^{R_{N-1}} \sum_{r_{N+1}=1}^{R_{N+1}} \cdots \sum_{r_O=1}^{R_O} x_{i,r_1 \cdots r_O} h_{p_1 \cdots p_O}^{(r)} w_{1,p_1,r_1}^{(r)} \cdots w_{N-1,p_{N-1},r_{N-1}}^{(r)} w_{N+1,p_{N+1},r_{N+1}}^{(r)} \cdots w_{O,p_O,r_O}^{(r)}, \tag{B.53}$$

the partial derivatives of the log-likelihood with respect to $w_{N,p_N,r_N}^{(r)}$ for $N = 1, ..., O$, $r_N = 1, ..., R_N$ and $p_N = 1, ..., P_N$ can be found as follows:

$$\frac{\partial l}{\partial w_{N,p_N,r_N}^{(r)}} = \sum_{i=1}^{n} \left[l_{ir} \{ F_{i,[p_N,r_N]} - \frac{F_{i,[p_N,r_N]} F_{i,1}^{(r)}}{F_{i,2}} \} + \sum_{r^*=1, r^* \neq r}^{L-1} l_{ir^*} \{ - \frac{F_{i,[p_N,r_N]} F_{i,1}^{(r)}}{F_{i,2}} \} \right]. \tag{B.54}$$

We would also find the partial derivative of the log-likelihood function from equation (B.48) with respect to the core matrix for each class $r = 1, ..., L-1$. The partial derivative of the log-likelihood with respect to $h^{(r)}_{p_1,...,p_O}$ for $N = 1, ...O$ and $p_N = 1, ..., P_N$ can be found using

$$\begin{aligned}\frac{\partial l}{\partial h^{(r)}_{p_1,...,p_O}} = \sum_{i=1}^{n}\Bigg[l_{ir}\{\sum_{r_1=1}^{R_1}\cdots\sum_{r_O=1}^{R_O} x_{i,r_1\cdots r_O} w^{(r)}_{1,p_1,r_1}\cdots w^{(r)}_{O,p_O,r_O} \\ - \frac{\sum_{r_1=1}^{R_1}\cdots\sum_{r_O=1}^{R_O} x_{i,r_1\cdots r_O} w^{(r)}_{1,p_1,r_1}\cdots w^{(r)}_{O,p_O,r_O} F^{(r)}_{i,1}}{F_{i,2}}\} \\ + \sum_{r^*=1,r^*\neq r}^{6} l_{ir^*}\{-\frac{\sum_{r_1=1}^{R_1}\cdots\sum_{r_O=1}^{R_O} x_{i,r_1\cdots r_O} w^{(r)}_{1,p_1,r_1}\cdots w^{(r)}_{O,p_O,r_O} F^{(r)}_{i,1}}{F_{i,2}}\}\Bigg].\end{aligned} \tag{B.55}$$

B7: Score and Fisher Information

B7.1: Score and Fisher Information

We will be using the same notation mentioned in Section 4.5.1. Similarly, we will go through the Jacobian and Hessian matrices of the systematic part $\eta = g(\mu)$ as in equation (4.34) for Tucker decomposition as well as the score function, Hessian matrix and Fisher information matrix for the Tucker TR model.

The gradient $\triangledown\eta(\mathbb{H}, \mathbf{W}_1, ..., \mathbf{W}_O) \in \mathbb{R}^{\prod_{N=1}^{O} P_N + \sum_{N=1}^{O} P_N R_N}$ is given by

$$\triangledown\eta(\mathbb{H}, \mathbf{W}_1, ..., \mathbf{W}_O) = [\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_1 \, \mathbf{J}_1 \, \mathbf{J}_2 \cdots \mathbf{J}_O]^{\mathrm{T}} \mathrm{vec}(\mathbb{X}), \tag{B.56}$$

where $\mathbf{J}_N \in \mathbb{R}^{\prod_{N=1}^{O} R_N \times R_N P_N}$ is the Jacobian matrix given by

$$\mathbf{J}_N = D\mathbf{W}(\mathbf{W}_N) = \mathbf{\Pi}_N\{[(\mathbf{W}_O \odot \cdots \odot \mathbf{W}_{N+1} \odot \mathbf{W}_{N-1} \odot \cdots \odot \mathbf{W}_1)\mathbb{H}^{\mathrm{T}}_{(N)}] \otimes \mathbb{I}_{R_N}\}, \tag{B.57}$$

$\mathbf{\Pi}_N \in \mathbb{R}^{\prod_{N=1}^{O} R_N \times \prod_{N=1}^{O} R_N}$ is the permutation matrix which reorders $\mathrm{vec}(\mathbb{W}_{(N)})$ to $\mathrm{vec}(\mathbb{W})$, i.e., $\mathrm{vec}(\mathbb{W}) = \mathbf{\Pi}_N \mathrm{vec}(\mathbb{W}_{(N)})$, and $\mathbb{I}_{R_N}$ is an identity matrix.

Let the Hessian matrix defined as

$$d^2\eta(\mathbb{H}, \mathbf{W}_1, ..., \mathbf{W}_O) \in \mathbb{R}^{(\prod_{N=1}^{O} P_N + \sum_{N=1}^{O} R_N P_N) \times (\prod_{N=1}^{O} P_N + \sum_{N=1}^{O} R_N P_N)}$$

be partitioned into four blocks which we will define as $\mathbb{C}_{\mathbb{H},\mathbb{H}} \in \mathbb{R}^{\prod_{N=1}^{O} P_N \times \prod_{N=1}^{O} P_N}$, $\mathbb{C}_{\mathbb{H},\mathbb{W}} = \mathbb{C}^{\mathrm{T}}_{\mathbb{W},\mathbb{H}} \in \mathbb{R}^{\prod_{N=1}^{O} P_N \times \sum_{N=1}^{O} R_N P_N}$, and $\mathbb{C}_{\mathbb{W},\mathbb{W}} \in \mathbb{R}^{\sum_{N=1}^{O} R_N P_N \times \sum_{N=1}^{O} R_N P_N}$. Then $\mathbb{C}_{\mathbb{H},\mathbb{H}} = \mathbf{0}$, $\mathbb{C}_{\mathbb{H},\mathbb{W}}$ has entries

$$h_{(p_1,...,p_N),(i_N,q_N)} = 1_{\{p_N = s_N\}} \sum_{j_N = i_N} x_{j_1,...,j_O} \prod_{N' \neq N} w_{j_{N'} p_{N'}}, \tag{B.58}$$

and $\mathbb{C}_{\mathbb{W},\mathbb{W}}$ has entries

$$h_{(i_N,p_N),(i_{N'},p_{N'})} = 1_{\{N\neq N'\}} \sum_{j_N=i_N, j_{N'}=i_{N'}} x_{j_1,\ldots,j_O} \sum_{s_N=r_N, s_{N'}=r_{N'}} h_{s_1,\ldots,s_O} \prod_{N''\neq N,N'} w_{j_{N''}s_{N''}}. \quad \text{(B.59)}$$

$\mathbb{C}_{\mathbb{W},\mathbb{W}}$ can be partitioned into O^2 sub-blocks as follows

$$\begin{bmatrix} \mathbf{0} & * & \cdots & * \\ \mathbf{H}_{21} & \mathbf{0} & * & \vdots \\ \vdots & \vdots & \ddots & * \\ \mathbf{H}_{O1} & \mathbf{H}_{O2} & \cdots & \mathbf{0} \end{bmatrix}, \quad \text{(B.60)}$$

where $\mathbf{H}_{NN'} \in \mathbb{R}^{R_N P_N \times R_{N'} P_{N'}}$ has elements that can be found from the matrix $\mathbb{X}_{(NN')}(\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_{N+1} \otimes \mathbf{W}_{N-1} \otimes \cdots \otimes \mathbf{W}_{N'+1} \otimes \mathbf{W}_{N'-1} \otimes \cdots \otimes \mathbf{W}_1)\mathbb{H}^{\mathrm{T}}_{(NN')}$. Also, $\mathbb{C}_{\mathbb{H},\mathbb{W}}$ can be partitioned into O sub-blocks as $(\mathbf{H}_1, \ldots, \mathbf{H}_O)$, where $\mathbf{H}_N \in \mathbb{R}^{\prod_{N=1}^{O} P_N \times R_N P_N}$ has up to $R_N \prod_{N=1}^{O} P_N$ non-zero elements that can be found from the matrix $\mathbb{X}_{(N)}(\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_{N+1} \otimes \mathbf{W}_{N-1} \otimes \cdots \otimes \mathbf{W}_1)$.

Consider the Tucker TR model given in equations (4.33) and (B.38), then we have that the score function/vector is given by

$$\triangledown l(\mathbb{H}, \mathbf{W}_1, \ldots, \mathbf{W}_O) = \frac{(l-\mu)\mu'(\eta)}{\sigma^2} \triangledown \eta(\mathbb{H}, \mathbf{W}_1, \ldots, \mathbf{W}_O), \quad \text{(B.61)}$$

where $\triangledown \eta(\mathbb{H}, \mathbf{W}_1, \ldots, \mathbf{W}_O)$. The Hessian matrix of the log-density is given by

$$\begin{aligned} H(\mathbb{H}, \mathbf{W}_1, \ldots, \mathbf{W}_O) = &-\left[\frac{[\mu'(\eta)]^2}{\sigma^2} - \frac{(l-\mu)\theta''(\eta)}{\sigma^2}\right] \triangledown \eta(\mathbb{H}, \mathbf{W}_1, \ldots, \mathbf{W}_O) d\eta(\mathbb{H}, \mathbf{W}_1, \ldots, \mathbf{W}_O) \\ &+ \frac{(l-\mu)\theta'(\eta)}{\sigma^2} d^2\eta(\mathbb{H}, \mathbf{W}_1, \ldots, \mathbf{W}_O), \end{aligned}$$

where $d^2\eta(\mathbb{H}, \mathbf{W}_1, \ldots, \mathbf{W}_O)$. The Fisher information matrix can be found as follows

$$\begin{aligned} \mathbb{F}(\mathbb{H}, \mathbf{W}_1, \ldots, \mathbf{W}_O) &= \mathbb{E}[-H(\mathbb{H}, \mathbf{W}_1, \ldots, \mathbf{W}_O)] \\ &= \mathrm{Var}[\triangledown l(\mathbb{H}, \mathbf{W}_1, \ldots, \mathbf{W}_O) dl(\mathbb{H}, \mathbf{W}_1, \ldots, \mathbf{W}_O)] \\ &= \frac{[\mu'(\eta)]^2}{\sigma^2} \triangledown \eta(\mathbb{H}, \mathbf{W}_1, \ldots, \mathbf{W}_O) \mathrm{vec}(\mathbb{X})^{\mathrm{T}} [\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_1 \, \mathbf{J}_1 \, \mathbf{J}_2 \cdots \mathbf{J}_O]. \end{aligned} \quad \text{(B.62)}$$

B7.2: Identifiability

Similar to what we saw for CP decomposition, parameterization of the Tucker TR model will not be unique, where local and global identifiability is possible under certain restrictions on the parameter space. The permutation indeterminacy can be fixed similar

to what was mentioned for CP decomposition. To fix the scaling indeterminacy, the factor matrices are scaled such that the entries of the first P_N rows of $\mathbb{W}_N$ are fixed as ones (so instead of a fixed Z rows for all factor matrices). However, as mentioned in Section 4.5.2, decompositions are still not necessarily unique in cases where $O > 2$ even after fixing these indeterminacies.

The sufficient and necessary condition for local identifiability defined in Proposition 4.3 follows for a Tucker TR model as well, where we would say that $\mathbb{W}_0$ is locally identifiable if and only if

$$\mathbb{F}(\mathbb{W}_0) = [\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_1 \, \mathbf{J}_1 \cdots \mathbf{J}_O]^{\mathrm{T}} \left[\sum_{i=1}^{n} \frac{\mu'(\eta_i)^2}{\sigma_i^2} \mathrm{vec}(\mathbb{X}_i)\mathrm{vec}(\mathbb{X}_i)^{\mathrm{T}} \right] [\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_1 \, \mathbf{J}_1 \cdots \mathbf{J}_O]$$

is nonsingular. From Proposition 4.3, the linear independence of the 'collapsed vectors'

$$[\mathbf{W}_O \otimes \cdots \otimes \mathbf{W}_1 \, \mathbf{J}_1 \cdots \mathbf{J}_O]^{\mathrm{T}} \mathrm{vec}(\mathbb{X}_i) \in \mathbb{R}^{\Sigma_{N=1}^{O} R_N P_N + \Pi_{N=1}^{O} P_N - \Sigma_{N=1}^{O} P_N^2},$$

is required for $i = 1, ..., n$.

B8: Regularization for Tucker TR

Regularization follows similarly for Tucker TR as in CP TR, although it can be done for the core tensor $\mathbb{H}$ only or both the core and factor matrices at the same time (Li, 2014). The following regularized log-likelihood function will be maximized:

$$l(b, \mathbf{W}_1, ..., \mathbf{W}_O | \mathcal{D}) - P_\lambda([\mathbb{H}, \mathbf{W}_1, ..., \mathbf{W}_O]), \tag{B.63}$$

where the sub-problem considered in the block relaxation algorithm when updating $\mathbf{W}_N$ from equation (B.42) would be updated as follows

$$\begin{aligned} \mathbf{W}_N^{(q+1)} = \mathrm{argmax}_{\mathbf{W}_N} \, & l(b^{(q)}, \mathbb{H}^{(q)}, \mathbf{W}_1^{(q+1)}, \ldots, \mathbf{W}_{N-1}^{(q+1)}, \mathbf{W}_N, \mathbf{W}_{N+1}^{(q)}, ..., \mathbf{W}_O^{(q)} \\ & - P_\lambda([\mathbb{H}, \mathbf{W}_1, ..., \mathbf{W}_O])). \end{aligned}$$

C

Appendix C

C1: CNN Application Results - Original Images

In this section, we will be presenting our results for the CNN application using the original images of our dance dataset. Section C1.1 considers the results of Trial B1, Section C1.2 considers the results of Trial B2, Section C1.3 considers the results of Trial B3, and Section C1.4 considers the results of Trial A. Each respective section will be further divided into subsections going into the results for each model.

C1.1: Results - Trial B1

C1.1.1: Results - Original AlexNet

Tables C.1 and C.2 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss, respectively.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	32	0.00016	0.04539	0.49976	98.433%	19.528
2	4	0.002085	0.025809	0.224387	98.433%	26.51
3	8	0.002271	0.004625	0.789483	97.389%	37.675

Table C.1: Top-5 sets of hyperparameters using SGD under the AlexNet architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	128	0.053032	0.00067	0.9	0.999	97.389%	35.575
2	64	0.001265	0.000074	0.609264	0.332917	97.65%	38.955
3	16	0.000034	0.000302	0.9	0.999	97.65%	46.836

Table C.2: Top-5 sets of hyperparameters using AdamW under the AlexNet architecture filtered by lowest validation loss.

C1.1.2: Results - AlexNet + BN

Tables C.3 and C.4 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	16	0.000008	0.000554	0.84165	98.17%	26.285
2	4	0.034492	0.000896	0.55038	97.65%	26.906
3	16	0.001542	0.067323	0.57997	96.867%	29.769

Table C.3: Top-5 sets of hyperparameters using SGD under the AlexNet + BN architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	32	0.000047	0.000016	0.9	0.999	98.433%	18.644
2	64	0.000025	0.000019	0.9	0.999	98.172%	22.439
3	16	0.0119	0.000895	0.9	0.999	96.867%	33.917

Table C.4: Top-5 sets of hyperparameters using AdamW under the AlexNet + BN architecture filtered by lowest validation loss.

C1.1.3: Results - ResNet18 and ResNet34

For ResNet18, Tables C.5 and C.6 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	8	0.000057	0.086117	0.261278	98.695%	7.693
2	4	0.000009	0.022549	0.819009	98.172%	13.625
3	8	0.000917	0.007358	0.690958	98.433%	14.642

Table C.5: Top-5 sets of hyperparameters using SGD under the ResNet18 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	16	0.021243	0.000556	0.277825	0.579039	99.217%	8.959
2	4	0.00001	0.000027	0.9	0.999	97.65%	12.062
3	8	0.000046	0.000207	0.336204	0.144413	98.172%	14.413

Table C.6: Top-5 sets of hyperparameters using AdamW under the ResNet18 architecture filtered by lowest validation loss.

For ResNet34, Tables C.7 and C.8 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	16	0.000025	0.011673	0.329518	99.739%	5.159
2	8	0.000004	0.004762	0.458226	98.172%	10.371
3	4	0.000001	0.091265	0.21843	98.433%	20.714

Table C.7: Top-5 sets of hyperparameters using SGD under the ResNet34 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	128	0.000007	0.000235	0.66969	0.71326	97.128%	42.414
2	4	0.00518	0.000032	0.37913	0.34981	97.389%	43.751
3	64	0.000022	0.003901	0.17685	0.57369	91.123%	48.684

Table C.8: Top-5 sets of hyperparameters using AdamW under the ResNet34 architecture filtered by lowest validation loss.

C1.2: Results - Trial B2

C1.2.1: Results - Original AlexNet

Tables C.9 and C.10 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	4	0.002	0.003533	0.74912	99.02%	17.095
2	8	0.000001	0.003976	0.73908	98.775%	25.954
3	4	0.003074	0.022373	0.15086	97.059%	33.116

Table C.9: Top-5 sets of hyperparameters using SGD under the AlexNet architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	8	0.000096	0.000031	0.9	0.999	97.549%	26.141
2	32	0.000002	0.000194	0.9	0.999	96.569%	26.782
3	4	0.073338	0.000159	0.9	0.999	97.549%	27.984

Table C.10: Top-5 sets of hyperparameters using AdamW under the AlexNet architecture filtered by lowest validation loss.

C1.2.2: Results - AlexNet + BN

Tables C.11 and C.12 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	16	0.000001	0.002675	0.21414	98.039%	21.58
2	32	0.007951	0.011491	0.13692	97.549%	34.382
3	16	0.055805	0.001582	0.54124	97.794%	36.596

Table C.11: Top-5 sets of hyperparameters using SGD under the AlexNet + BN architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	32	0.000047	0.000016	0.9	0.999	98.433%	19.861
2	64	0.000025	0.000019	0.9	0.999	98.172%	23.904
3	16	0.0119	0.000895	0.9	0.999	96.867%	36.13

Table C.12: Top-5 sets of hyperparameters using AdamW under the AlexNet + BN architecture filtered by lowest validation loss.

C1.2.3: Results - ResNet18 and ResNet34

For ResNet18, Tables C.13 and C.14 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	32	0.010704	0.013075	0.44376	99.265%	12.034
2	64	0.004415	0.018546	0.71618	98.284%	22.176
3	64	0.000631	0.10945	0.4293	97.304%	36.923

Table C.13: Top-5 sets of hyperparameters using SGD under the ResNet18 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	4	0.00795	0.000746	0.50105	0.84379	98.775%	24.009
2	8	0.066469	0.000225	0.72298	0.55349	97.059%	31.717
3	16	0.028007	0.00005	0.57287	0.23745	94.853%	32.132

Table C.14: Top-5 sets of hyperparameters using AdamW under the ResNet18 architecture filtered by lowest validation loss.

For ResNet34, Tables C.15 and C.16 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	32	0.0002651	0.003544	0.53048	96.814%	25.125
2	16	0.000038	0.007949	0.41607	98.039%	25.385
3	8	0.003196	0.000816	0.71444	96.568%	31.936

Table C.15: Top-5 sets of hyperparameters using SGD under the ResNet34 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	32	0.000203	0.002729	0.2798	0.64166	98.284%	22.615
2	16	0.00019	0.000031	0.9	0.999	97.794%	32.877
3	16	0.000006	0.000631	0.9	0.999	96.569%	35.085

Table C.16: Top-5 sets of hyperparameters using AdamW under the ResNet34 architecture filtered by lowest validation loss.

C1.3: Results - Trial B3

C1.3.1: Results - Original AlexNet

Tables C.17 and C.18 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	16	0.001994	0.001994	0.61152	93.228%	59.115
2	64	0.000604	0.020994	0.43355	93.228%	59.538
3	16	0.000002	0.00114	0.8476	93.68%	64.561

Table C.17: Top-5 sets of hyperparameters using SGD under the AlexNet architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	8	0.000067	0.000043	0.9	0.999	97.517%	27.211
2	16	0.018958	0.000169	0.9	0.999	95.937%	27.225
3	4	0.001017	0.000014	0.9	0.999	96.84%	34.241

Table C.18: Top-5 sets of hyperparameters using AdamW under the AlexNet architecture filtered by lowest validation loss.

C1.3.2: Results - AlexNet + BN

Tables C.19 and C.20 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	4	0.000452	0.001158	0.93997	97.743%	27.084
2	64	0.018929	0.001224	0.81732	97.743%	27.089
3	64	0.003666	0.000543	0.81917	94.131%	31.227

Table C.19: Top-5 sets of hyperparameters using SGD under the AlexNet + BN architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	64	0.000619	0.000012	0.45675	0.079152	95.485%	36.318
2	128	0.083878	0.000023	0.38117	0.51865	94.131%	39.139
3	64	0.001008	0.000009	0.9	0.999	94.582%	41.313

Table C.20: Top-5 sets of hyperparameters using AdamW under the AlexNet + BN architecture filtered by lowest validation loss.

C1.3.3: Results - ResNet18 and ResNet34

For ResNet18, Tables C.21 and C.22 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	4	0.000259	0.001211	0.70886	99.097%	8.971
2	8	0.000158	0.001797	0.583	97.968%	21
3	4	0.000108	0.000256	0.88142	97.517%	23.312

Table C.21: Top-5 sets of hyperparameters using SGD under the ResNet18 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	8	0.000588	0.000018	0.9	0.999	97.968%	19.388
2	64	0.000003	0.00003	0.24295	0.31185	97.968%	23.463
3	64	0.000021	0.000061	0.0642	0.24532	97.517%	23.821

Table C.22: Top-5 sets of hyperparameters using AdamW under the ResNet18 architecture filtered by lowest validation loss.

For ResNet34, Tables C.23 and C.24 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	64	0.000741	0.049312	0.49733	96.614%	25.935
2	32	0.001495	0.005454	0.63128	97.066%	36.798
3	64	0.000002	0.003106	0.5136	93.68%	61.422

Table C.23: Top-5 sets of hyperparameters using SGD under the ResNet34 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	8	0.000485	0.000015	0.9	0.999	96.614%	30.752
2	4	0.000009	0.000011	0.9	0.999	97.743%	32.193
3	4	0.000021	0.000032	0.9	0.999	96.614%	46.086

Table C.24: Top-5 sets of hyperparameters using AdamW under the ResNet34 architecture filtered by lowest validation loss.

C1.4: Results - Trial A

C1.4.1: Results - Original AlexNet

Tables C.25 and C.26 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	8	0.000001	0.000154	0.95165	93.507%	29.156
2	16	0.000102	0.004575	0.70619	96.104%	32.167
3	4	0.000007	0005189	0.69753	96.104%	44.862

Table C.25: Top-5 sets of hyperparameters using SGD under the AlexNet architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	32	0.007044	0.000128	0.9	0.999	98.016%	12.857
2	4	0.001328	0.000031	0.9	0.999	93.831%	45.893
3	32	0.000297	0.000026	0.52883	0.1193	96.429%	61.708

Table C.26: Top-5 sets of hyperparameters using AdamW under the AlexNet architecture filtered by lowest validation loss.

C1.4.2: Results - AlexNet + BN

Tables C.27 and C.28 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	256	0.001652	0.005999	0.41822	94.805%	27.589
2	4	0.000005	0.027074	0.043949	96.104%	33.846
3	32	0.000028	0.00572	0.87246	94.481%	42.246

Table C.27: Top-5 sets of hyperparameters using SGD under the AlexNet + BN architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	64	0.015837	0.000018	0.9	0.999	98.052%	15.976
2	4	0.055906	0.000103	0.9	0.999	97.078%	24.959
3	16	0.000011	0.000091	0.9	0.999	96.429%	33.964

Table C.28: Top-5 sets of hyperparameters using AdamW under the AlexNet + BN architecture filtered by lowest validation loss.

C1.4.3: Results - ResNet18 and ResNet34

For ResNet18, Tables C.29 and C.30 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	16	0.000006	0.048694	0.88006	98.701%	3.999
2	8	0.00034	0.008821	0.17236	98.052%	12.036
3	128	0.000522	0.004512	0.91722	97.078%	19.625

Table C.29: Top-5 sets of hyperparameters using SGD under the ResNet18 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	4	0.000055	0.000019	0.9	0.999	98.377%	7.188
2	4	0.000259	0.000006	0.23417	0.79061	95.779%	29.688
3	64	0.000009	0.000131	0.9	0.999	96.104%	31.166

Table C.30: Top-5 sets of hyperparameters using AdamW under the ResNet18 architecture filtered by lowest validation loss.

For ResNet34, Tables C.31 and C.32 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	4	0.002545	0.00068	0.71047	95.779%	23.363
2	16	0.006989	0.001778	0.74068	97.078%	29.408
3	4	0.001117	0.007361	0.24234	97.078%	34.196

Table C.31: Top-5 sets of hyperparameters using SGD under the ResNet34 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	8	0.001029	0.000082	0.9	0.999	96.753%	17.044
2	8	0.000979	0.000015	0.9	0.999	97.727%	17.389
3	4	0.000789	0.000036	0.9	0.999	96.429%	32.216

Table C.32: Top-5 sets of hyperparameters using AdamW under the ResNet34 architecture filtered by lowest validation loss.

C2: CNN Application Results - Resized Images

In this section, we will be presenting our results for the CNN application using the resized images of our dance dataset. Section C2.1 considers the results of Trial B1, Section C2.2 considers the results of Trial B2, and Section C2.3 considers the results of Trial B3. Each respective section will be further divided into subsections going into the results for each model.

C2.1: Results - Trial B1

C2.1.1: Results - Original AlexNet

Tables C.33 and C.34 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss, respectively.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	8	0.009087	0.00576	0.4554	94.517%	37.448
2	4	0.00438	0.002855	0.16013	96.084%	40.877
3	8	0.000026	0.004229	0.015124	93.99%	44.932

Table C.33: Top-5 sets of hyperparameters using SGD under the AlexNet architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	8	0.000002	0.000059	0.9	0.999	97.65%	19.351
2	8	0.000277	0.000034	0.38846	0.92627	98.172%	21.31
3	16	0.009101	0.000207	0.9	0.999	96.867%	27.66

Table C.34: Top-5 sets of hyperparameters using AdamW under the AlexNet architecture filtered by lowest validation loss.

C2.1.2: Results - AlexNet + BN

Tables C.35 and C.36 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	8	0.008347	0.001817	0.39756	98.695%	7.584
2	4	0.001689	0.000604	0.80989	98.695%	16.02
3	4	0.000019	0.000279	0.22927	97.911%	17.185

Table C.35: Top-5 sets of hyperparameters using SGD under the AlexNet + BN architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	64	0.01198	0.00003	0.9	0.999	97.65%	12.968
2	64	0.000396	0.000027	0.9	0.999	98.172%	14.9
3	64	0.042114	0.000011	0.9	0.999	97.389%	17.712

Table C.36: Top-5 sets of hyperparameters using AdamW under the AlexNet + BN architecture filtered by lowest validation loss.

C2.1.3: Results - ResNet18 and ResNet34

For ResNet18, Tables C.37 and C.38 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	8	0.000026	0.011565	0.29009	98.695%	7.352
2	4	0.000006	0.015145	0.15977	98.956%	12.582
3	8	0.010428	0.009022	0.12235	97.65%	18.526

Table C.37: Top-5 sets of hyperparameters using SGD under the ResNet18 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	8	0.000935	0.000009	0.82413	0.79807	97.911%	11.315
2	64	0.000068	0.000042	0.79875	0.14979	97.911%	12.804
3	4	0.000079	0.000165	0.9	0.999	97.911%	17.257

Table C.38: Top-5 sets of hyperparameters using AdamW under the ResNet18 architecture filtered by lowest validation loss.

For ResNet34, Tables C.39 and C.40 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	8	0.00048	0.000828	0.57775	98.433%	15.136
2	4	0.000004	0.000652	0.4464	97.389%	21.978
3	4	0.049858	0.000149	0.69694	97.389%	23.88

Table C.39: Top-5 sets of hyperparameters using SGD under the ResNet34 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	8	0.000002	0.000074	0.9	0.999	98.172%	11.427
2	32	0.000021	0.000373	0.9	0.999	97.128%	31.605
3	64	0.000032	0.000064	0.9	0.999	93.995%	49.034

Table C.40: Top-5 sets of hyperparameters using AdamW under the ResNet34 architecture filtered by lowest validation loss.

C2.2: Results - Trial B2

C2.2.1: Results - Original AlexNet

Tables C.41 and C.42 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	8	0.000004	0.009598	0.53701	97.304%	24.166
2	4	0.000009	0.002036	0.38521	96.078%	43.782
3	4	0.001823	0.0016	0.18563	94.118%	61.413

Table C.41: Top-5 sets of hyperparameters using SGD under the AlexNet architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	128	0.00016	0.000102	0.9	0.999	94.608%	26.834
2	16	0.000062	0.000046	0.9	0.999	96.324%	26.873
3	8	0.000001	0.000195	0.9	0.999	96.324%	32.667

Table C.42: Top-5 sets of hyperparameters using AdamW under the AlexNet architecture filtered by lowest validation loss.

C2.2.2: Results - AlexNet + BN

Tables C.43 and C.44 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	128	0.000231	0.030226	0.19716	94.853%	23.115
2	4	0.000015	0.000296	0.36939	98.039%	25.626
3	32	0.000067	0.000309	0.92937	94.608%	25.976

Table C.43: Top-5 sets of hyperparameters using SGD under the AlexNet + BN architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	16	0.00021	0.000009	0.68697	0.84077	97.059%	12.93
2	32	0.008404	0.000025	0.9	0.999	96.814%	18.301
3	8	0.000527	0.000017	0.9	0.999	97.794%	21.014

Table C.44: Top-5 sets of hyperparameters using AdamW under the AlexNet + BN architecture filtered by lowest validation loss.

C2.2.3: Results - ResNet18 and ResNet34

For ResNet18, Tables C.45 and C.46 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	8	0.000004	0.01185	0.32082	98.775%	8.884
2	4	0.00017	0.000623	0.65181	98.529%	15.794
3	8	0.000199	0.003034	0.72801	98.039%	16.655

Table C.45: Top-5 sets of hyperparameters using SGD under the ResNet18 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	64	0.022353	0.000221	0.9	0.999	98.284%	17.983
2	8	0.018846	0.000073	0.9	0.999	98.284%	23.666
3	32	0.00021	0.000548	0.1004	0.21774	97.549%	27.662

Table C.46: Top-5 sets of hyperparameters using AdamW under the ResNet18 architecture filtered by lowest validation loss.

For ResNet34, Tables C.47 and C.48 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	32	0.000003	0.016037	0.20325	97.549%	18.376
2	16	0.004089	0.002541	0.84498	97.549%	26.19
3	8	0.000245	0.00106	0.94324	96.569%	42.003

Table C.47: Top-5 sets of hyperparameters using SGD under the ResNet34 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	16	0.000002	0.000411	0.26215	0.74412	98.284%	11.205
2	32	0.000885	0.000338	0.46527	0.82726	98.529%	12.424
3	4	0.001918	0.000036	0.9	0.999	97.794%	14.704

Table C.48: Top-5 sets of hyperparameters using AdamW under the ResNet34 architecture filtered by lowest validation loss.

C2.3: Results - Trial B3

C2.3.1: Results - Original AlexNet

Tables C.49 and C.50 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	4	0.000008	0.00802	0.076799	97.743%	26.177
2	32	0.001482	0.023629	0.19672	95.26%	42.097
3	64	0.000027	0.051758	0.11963	90.294%	80.163

Table C.49: Top-5 sets of hyperparameters using SGD under the AlexNet architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	128	0.01112	0.000422	0.9	0.999	94.357%	30.182
2	16	0.00013	0.000027	0.9	0.999	97.066%	30.562
3	8	0.001518	0.000067	0.14434	0.98713	97.743%	37.437

Table C.50: Top-5 sets of hyperparameters using AdamW under the AlexNet architecture filtered by lowest validation loss.

C2.3.2: Results - AlexNet + BN

Tables C.51 and C.52 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	8	0.022115	0.002276	0.6244	98.194%	20.55
2	16	0.000725	0.006556	0.35721	97.066%	37.252
3	64	0.001225	0.000182	0.9464	95.937%	40.199

Table C.51: Top-5 sets of hyperparameters using SGD under the AlexNet + BN architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	8	0.054103	0.000012	0.9	0.999	98.42%	28.233
2	128	0.000374	0.000136	0.9	0.999	97.517%	32.45
3	128	0.000037	0.000012	0.30667	0.80839	95.26%	37.871

Table C.52: Top-5 sets of hyperparameters using AdamW under the AlexNet + BN architecture filtered by lowest validation loss.

C2.3.3: Results - ResNet18 and ResNet34

For ResNet18, Tables C.53 and C.54 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	128	0.000002	0.014923	0.83827	97.517%	13.1
2	4	0.000042	0.035753	0.69131	97.291%	19.316
3	16	0.00001	0.014479	0.29424	97.066%	29.894

Table C.53: Top-5 sets of hyperparameters using SGD under the ResNet18 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	32	0.000003	0.000064	0.9	0.999	98.42%	19.296
2	16	0.000003	0.000058	0.9	0.999	97.968%	19.612
3	32	0.000027	0.000052	0.9	0.999	97.517%	22.646

Table C.54: Top-5 sets of hyperparameters using AdamW under the ResNet18 architecture filtered by lowest validation loss.

For ResNet34, Tables C.55 and C.56 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	8	0.000088	0.000862	0.35885	96.84%	27.371
2	16	0.00538	0.001333	0.44909	96.84%	38.215
3	32	0.002627	0.002432	0.52648	95.937%	42.904

Table C.55: Top-5 sets of hyperparameters using SGD under the ResNet34 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	32	0.013869	0.000113	0.9	0.999	97.517%	32.521
2	4	0.000002	0.000063	0.9	0.999	96.388%	33.464
3	16	0.000362	0.000049	0.9285	0.94245	98.194%	34.925

Table C.56: Top-5 sets of hyperparameters using AdamW under the ResNet34 architecture filtered by lowest validation loss.

C2.4: Results - Trial A

C2.4.1: Results - Original AlexNet

Tables C.57 and C.58 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	4	0.000179	0.000638	0.92313	96.104%	30.988
2	8	0.000047	0.000938	0.61966	93.506%	40.32
3	8	0.008978	0.000879	0.48616	77.273%	162.159

Table C.57: Top-5 sets of hyperparameters using SGD under the AlexNet architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	4	0.000313	0.000019	0.9	0.999	96.753%	15.65
2	128	0.022094	0.000401	0.9	0.999	95.455%	31.534
3	64	0.055326	0.000079	0.45538	0.018347	92.857%	42.255

Table C.58: Top-5 sets of hyperparameters using AdamW under the AlexNet architecture filtered by lowest validation loss.

C2.4.2: Results - AlexNet + BN

Tables C.59 and C.60 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	4	0.000009	0.000402	0.78415	98.377%	11.138
2	4	0.000939	0.00233	0.26651	97.727%	15.012
3	32	0.068817	0.000948	0.60011	97.727%	25.531

Table C.59: Top-5 sets of hyperparameters using SGD under the AlexNet + BN architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	256	0.000029	0.000037	0.32329	0.77007	94.805%	17.425
2	256	0.008507	0.000024	0.9	0.999	94.481%	17.985
3	16	0.00023	0.000018	0.9	0.999	96.104%	26.008

Table C.60: Top-5 sets of hyperparameters using AdamW under the AlexNet + BN architecture filtered by lowest validation loss.

C2.4.3: Results - ResNet18 and ResNet34

For ResNet18, Tables C.61 and C.62 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	4	0.000022	0.030051	0.89934	98.377%	7.239
2	16	0.000004	0.020438	0.63155	96.429%	18.807
3	64	0.000361	0.005055	0.38966	96.429%	26.008

Table C.61: Top-5 sets of hyperparameters using SGD under the ResNet18 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	32	0.000059	0.000077	0.69951	0.70532	97.403%	11.541
2	128	0.003766	0.000054	0.9	0.999	96.753%	12.414
3	32	0.000011	0.000035	0.9	0.999	97.08%	19.557

Table C.62: Top-5 sets of hyperparameters using AdamW under the ResNet18 architecture filtered by lowest validation loss.

For ResNet34, Tables C.63 and C.64 represent the top hyperparameter search results for SGD and AdamW on the training set filtered by validation loss.

SGD	Batch Size	Weight Decay	LR	Momentum	Validation Accuracy	Validation Loss
1	4	0.001071	0.000925	0.39908	99.026%	5.466
2	8	0.011525	0.00254	0.61858	99.026%	10.866
3	16	0.000027	0.003132	0.47157	98.052%	22.727

Table C.63: Top-5 sets of hyperparameters using SGD under the ResNet34 architecture filtered by lowest validation loss.

AdamW	Batch Size	Weight Decay	LR	Beta1	Beta2	Validation Accuracy	Validation Loss
1	32	0.000019	0.000588	0.9	0.999	98.377%	8.943
2	4	0.000009	0.000012	0.9	0.999	98.052%	15.853
3	16	0.000843	0.000013	0.63977	0.10049	97.727%	17.609

Table C.64: Top-5 sets of hyperparameters using AdamW under the ResNet34 architecture filtered by lowest validation loss.

C3: TR Application Results

In this section, we will be presenting our results for the TR application. Section C3.1 considers the results of Trial B1, Section C3.2 considers the results of Trial B2, Section C3.3 considers the results of Trial B3, and Section C3.4 considers the results of Trial A.

C3.1: Results - Trial B1

Table C.65 represents the top hyperparameter search results on the training set filtered by validation loss, respectively.

	Batch Size	L_1 parameter	LR	Momentum	Validation Accuracy	Validation Loss
1	32	0	0.000188	0.32127	59.288%	-436.96
2	32	0	8.347e-05	0.746	58.713%	-438.539
3	32	0	0.000266	0.49345	58.608%	-472.391

Table C.65: Top-5 sets of hyperparameters filtered by highest validation loss.

C3.2: Results - Trial B2

Table C.66 represents the top hyperparameter search results on the training set filtered by validation loss, respectively.

	Batch Size	L_1 parameter	LR	Momentum	Validation Accuracy	Validation Loss
1	32	0	0.000204	0.50491	64.801%	-433.347
2	16	0	0.000204	0.50491	60.628%	-451.763
3	32	0	0.000229	0.29167	60.236%	-481.161

Table C.66: Top-5 sets of hyperparameters filtered by highest validation loss.

C3.3: Results - Trial B3

Table C.67 represents the top hyperparameter search results on the training set filtered by validation loss, respectively.

	Batch Size	L_1 parameter	LR	Momentum	Validation Accuracy	Validation Loss
1	32	0	8.198e-05	0.84111	59.494%	-451.71
2	64	0	9.2e-05	0.78919	58.816%	-524.045
3	64	0	0.000438	0.03933	57.052%	-580.653

Table C.67: Top-5 sets of hyperparameters filtered by highest validation loss.

C3.4: Results - Trial A

Table C.68 represents the top hyperparameter search results on the training set filtered by validation loss, respectively.

	Batch Size	L_1 parameter	LR	Momentum	Validation Accuracy	Validation Loss
1	32	0	0.00025	0.6	82.1%	-154.071
2	64	0	0.00025	0.6	79.708%	-189.772
3	32	0	0.00005	0.95	77.597%	-198.602

Table C.68: Top-5 sets of hyperparameters filtered by highest validation loss.