MACHINE LEARNING WITH APPLICATIONS IN AGRICULTURE

BY

Abiodun Uthman ALLISON

B.Sc. Mathematics

(Federal University of Agriculture, Abeokuta)

Matric. Number: 204603

A Project submitted to the Department of Mathematics

Faculty of Science

University of Ibadan, Ibadan

Nigeria

In Partial Fulfilment of the Award of Master of Science (M.Sc.) in Mathematics

In The Department of Mathematics, Faculty of Science

University of Ibadan, Ibadan

Nigeria

FEBRUARY, 2021

Certification

I certify that this project was carried out by **ABIODUN UTHMAN ALLISON (Matric No. 204603)**, in the Department of Mathematics, Faculty of Science, University of Ibadan, under my supervision.

(Supervisor)

Professor G.O.S. Ekhaguere

Ph.D (London)

Department of Mathematics,

University of Ibadan, Nigeria

Dedication

Dedicated to Mulikat Allison.

Acknowledgement

I give all glory, honour and adoration to Almighty Allah who brought me to this stage of my life. I owe a million debt of gratitude to my ever-loving parents, Mr Jamiu Allison and Mrs Mulikat Allison for their relentless supports toward the success of my education. I also appreciate my ever loving siblings for their continuous support.

Also my profound appreciation goes to my supervisor, Professor G.O.S. Ekhaguere, for his patience, guidance and helpful suggestions, thank you sir. In the same vein, my unmeasured gratitude goes to all the distinguished Professors and members of the Department of Mathematics for their unrelenting effort. I also acknowledge my friends and my course mates for their love and support, I say thank you all.

Abstract

Current rapid development in Artificial Intelligence (AI) provides a vast selection of high quality tools to solve complex problems in more efficient ways than before. As a consequence, many fields of science and engineering are starting to explore AI tools, especially Deep Learning (DL) models for computer vision, audio and video understanding, speech recognition and decision making. In this project, we studied a type of deep learning model - Convolutional Neural Networks (CNNs), starting with a basic Machine Learning algorithm - Logistic Regression, moving on to building blocks of Neural Networks and method of training and optimising neural network models. We then trained three different CNN models; a base model, VGG16 and ResNext-50, to classify different plant diseases using healthy and diseased leaf images. The best performing model was ResNext-50 with 98% accuracy.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1

Introduction

1.0 Introduction

Agriculture, for decades, had been associated with the production of essential food crops. At present, agriculture also includes forestry, dairy, fruit cultivation, poultry, bee keeping, mushroom etc. Today, processing, marketing, and distribution of crops and livestock products are all acknowledged as part of current agriculture. Thus, agriculture could be referred to as the production, processing, promotion and distribution agricultural products. Agriculture plays a critical role in the entire life of a given economy. In addition to providing food and raw materials, agriculture also provides employment opportunities, government revenue, food security, significance in transport among other things.

Machine learning is the scientific study of algorithms and statistical models that computer systems use to perform a specific task without using explicit instructions, relying on patterns and inference instead. Machine learning algorithms build a mathematical model based on training data (a sample data), in order to make predictions or decisions without being explicitly programmed to perform the task. Machine learning algorithms are used in a wide variety of applications, such as email filtering, computer vision, and stock market prediction, where it is difficult to develop a conventional algorithm for effectively performing the task.

Plant health and food safety are closely linked. The Food and Agriculture Organization of the United Nations (FAO) estimates that pests and diseases lead to the loss of $20 - 40$ % of global food production, constituting a threat to food security (Food and Agriculture Or-

ganization of the United Nation, International Plant Protection Convention, 2017). Use of pesticides has been a way of protecting crops from these infestations and has been one of the factors behind the increase in food production since the 1950s, enabling it to meet the needs of a growing population **Cooper and Dobson (2007)**. However, the use of such substances is not environmentally harmless. Applying these substances negatively impacts biodiversity, including insect, bird, and fish populations, as well as soil, air, and water quality **Kaur and Garg (2014)**; **Sanchez-Bayo and Goka (2014)**; **Knillmann and Liess (2019)**. Their use also constitutes a risk to human health, with both acute and chronic effects **Sanborn et al. (2007)**; **Kim et al. (2017)**. Plant diseases are not only a threat to food security at the global scale, but can also have disastrous consequences for smallholder farmers whose livelihoods depend on healthy crops.

Computer vision, and object recognition in particular, has made tremendous advances in the past few years. The PASCAL Visual Object Classes (VOC) challenge **Everingham et al. (2010)**, established in 2005, which set the precedent for standardized evaluation of recognition algorithms in the form of yearly competitions, and more recently the Large Scale Visual Recognition Challenge (ILSVRC) **Russakovsky et al. (2014)** based on the ImageNet dataset **Deng et al. (2009)** have been widely used as benchmarks for numerous visualization-related problems in computer vision,including object classification. In 2012, a large, deep convolutional neural network achieved a top-5 test error (the fraction of test images for which the correct label is not among the five labels considered most probable by the model) rate of 15.3 %, almost halved the error rate compared to 26.2 % achieved by the second-best entry, at the ILSVRC 2012 image classification competition for the classification of 15 million images into 22 thousand possible categories **Krizhevsky et al. (2012)**. In the following years, various advances in deep convolutional neural networks lowered the top-5 error rate to 3.57 % **He et al. (2015a)**; **Szegedy et al. (2016)**, 3.03 % **Xie et al. (2017)**, 2.251 % **Hu et al. (2017)**.

In image classification problems with lots of classes, a considerable amount of sample size is required to train deep learning models. But with increase in the amount of data available, the development of Graphical Processing Unit (GPU) parallel computing ability, and cloud computing, there have a seen a significant improvement in training deep learning models.

This project work deals with the application of machine learning (deep learning in particular) algorithms in the areas of plant disease recognition and in agriculture.

1.1 Basic Definitions and Results

In what follows in the section, we give some preliminary definitions and results which will aid in the understanding of machine leaning and deep learning.

1.1.1 Tensor and Vectorisation

In this project, we use a symbol shown in boldface to represent a vector, for example, $\boldsymbol{x} \in \mathbb{R}^D$ is a column vector with D elements; a capital letter to denote a matrix, for example, $X \in \mathbb{R}^{H \times W}$ is a matrix with H rows and W columns. The vector $\boldsymbol{x}$ can also be viewed as a matrix with 1 column and D rows.

These concepts can be generalised to higher order matrices, i.e., tensors. For example, $\boldsymbol{x} \in \mathbb{R}^{H \times W \times D}$ is an order 3 tensor. It contains HWD elements, and each of them can be indexed by an index triplet (i, j, d), with $0 \leq i < H$, $0 \leq j < W$, and $0 \leq d < D$. Another way to view an order 3 tensor is to treat it as containing D channels of matrices. Every channel is a matrix with size $H \times W$. The first channel contains all the numbers in the tensor that are indexed by $(i, j, 0)$. When $D = 1$, an order 3 tensor reduces to a matrix.

A scalar value is an order 0 tensor; a vector is an order 1 tensor; and a matrix is a second order tensor. A color image is in fact an order 3 tensor, if a color image is stored in the RGB format, it has 3 channels (for R, G and B, respectively), and each channel is a $H \times W$ matrix that contains R (or G, or B) values of all pixels. It is beneficial to represent images (or other types of raw data) as a tensor, this helps keep the color information of the images as color is very important in various image based learning and recognition problem.

Given a tensor, we can arrange all the numbers inside it into a long vector, following a pre-specified order. In order to vectorise an order 3 tensor, we would vectorise its first channel (which is a matrix), then the second channel, ..., till all channels are vectorised. The vectorisation of the order 3 is then the concatenation of the vectorisation of all the channels in this order. This method can be applied to vectorise tensors of order greater than 3. For

example, in Python 3.7, `numpy.reshape(A.shape[0]*A.shape[1]*A.shape[2],1)` converts tensor $\boldsymbol{A}$ into a column vector in the row-first order.

1.1.2 Vector Calculus and the Chain Rule

Suppose $z \in \mathbb{R}$ is a scalar and $\boldsymbol{y} \in \mathbb{R}^H$ is a vector. IF z is a function of $\boldsymbol{y}$, the partial derivative of z with respect $\boldsymbol{y}$ is a vector, defined as

$$\left[\frac{\partial z}{\partial \boldsymbol{y}}\right]_i = \frac{\partial z}{\partial y_i} \tag{1.1.1}$$

In other words, $\frac{\partial z}{\partial \boldsymbol{y}}$ is a vector having the same size as $\boldsymbol{y}$, and its i-th element is $\frac{\partial z}{\partial y}$. We also note that $\frac{\partial z}{\partial \boldsymbol{y}^T} = (\frac{\partial z}{\partial \boldsymbol{y}})^T$.

Furthermore, suppose $\boldsymbol{x} \in \mathbb{R}^W$ is another vector, and $\boldsymbol{y}$ is a function of $\boldsymbol{x}$. Then, the partial derivative of $\boldsymbol{y}$ with respect $\boldsymbol{x}$ is defined as

$$\left[\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}^T}\right]_{ij} = \frac{\partial y_i}{\partial x_j} \tag{1.1.2}$$

This partial derivative is $H \times W$ matrix, whose entry at the intersection of the i-th row and j-th column is $\frac{\partial y_i}{\partial x_j}$.

In a chain-like argument, we see that z is a function of $\boldsymbol{x}$. The chain rule can be used to compute $\frac{\partial z}{\partial \boldsymbol{x}^T}$ as

$$\frac{\partial z}{\partial \boldsymbol{x}^T} = \frac{\partial z}{\partial \boldsymbol{y}^T}\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}^T} \tag{1.1.3}$$

Since $\frac{\partial z}{\partial \boldsymbol{x}^T}$ is a $1 \times H$ matrix and $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}^T}$ is an $H \times W$ matrix, the matrix multiplication in (1.1.3) is valid, and the result would be a row vector with W elements, which matches the dimensionality of $\frac{\partial z}{\partial \boldsymbol{x}^T}$.

1.1.3 Binomial Distribution

Consider a random experiment which consists of n independent trials, such that for each trial they are two possible outcomes; success and failure. Let $P(\text{success}) = p$ and $P(\text{failure}) = 1-p$, and X the random variable with x number of successes, then the binomial distribution is defined as

$$B(n,p) = \binom{n}{x} p^x (1-p)^{n-x}, \quad 0 \leq x \leq n \tag{1.1.4}$$

p^x computes the probability of achieving the the demanded x success over all trials, and $(1-p)^{n-x}$ computes the probability of the rest of the trials being failure.

1.1.4 Norms

The norm function is used to measure the size of vectors in machine learning. Formally, the L^p norm is given by

$$\|\boldsymbol{x}\|_p = \left(\sum_i |\boldsymbol{x}_i|^p\right)^{\frac{1}{p}} \tag{1.1.5}$$

for $p \in \mathbb{R}$, $p \geq 1$.

Norms, including the L^p norm, are functions mapping vectors to non-negative values. On an intuitive level, the norm of a vector $\boldsymbol{x}$ measures the distance from the origin to the point $\boldsymbol{x}$. A norm is a map $\|.\| : \boldsymbol{X} \to \mathbb{R}^+$ that satisfies the following properties:

- $\|x\| \geq 0, \quad \forall\, \boldsymbol{x} \in \boldsymbol{X}$ (non-negativity)
- $\|\boldsymbol{x}\| = 0$ if and only if $\boldsymbol{x} = 0$
- $\forall\, \alpha \in \mathbb{R}$, $\|\alpha\boldsymbol{x}\| = |\alpha|\|\boldsymbol{x}\|$ (positive homogeneous)
- $\|\boldsymbol{x} + \boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\| \quad \forall\, \boldsymbol{x}, \boldsymbol{y} \in \boldsymbol{X}$ (triangle inequality)

The L^2 norm, with $p = 2$, is known as the *Euclidean norm*, which is simply the Euclidean distance from the origin to the point identified by $\boldsymbol{x}$. The L^2 norm is used to keep a model less more complex by keeping the coefficients of the model small. It is given by:

$$\|\boldsymbol{x}\|_2 = \sqrt{\sum_i x_i^2} \tag{1.1.6}$$

In several machine learning applications, it is important to discriminate between elements that are exactly zero and elements that are small but nonzero. In these cases, we use the L^1 norm, which is given by:

$$\|\boldsymbol{x}\| = \sum_i |x_i| \tag{1.1.7}$$

One other norm that commonly arises in machine learning is the L^∞ norm, also known as the *max norm*. This norm simplifies to the absolute value of the element with the largest

magnitude in the vector, it is given by:

$$\|\boldsymbol{x}\|_\infty = \max_i |x_i| \tag{1.1.8}$$

Sometimes we may also wish to measure the size of a matrix. In the context of deep learning, the most common way to do this is with the *Frobenius norm*:

$$\|\boldsymbol{A}\|_F = \sqrt{\sum_{i,j} A_{i,j}^2} \tag{1.1.9}$$

which is analogous to the L^2 norm of a vector.

1.1.5 Convex Optimisation

Consider the general *constrained* optimisation problem

$$\begin{aligned} &\text{minimize} \quad f(x) \\ &\text{subject to} \quad \boldsymbol{x} \in D \subset \mathbb{R}^n \end{aligned} \tag{1.1.10}$$

The function $f : \mathbb{R}^n \to \mathbb{R}$ that we wish to minimise is a real valued function called *objective function*. The vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ is an n-vector of independent variables $x_1, x_2, \ldots, x_n$ which are called *decision variables*, the set $D \subset \mathbb{R}$ is called the *constrained or feasible set*. The optimisation problem involves obtaining the best vector of decision variables over all possible vectors in D. The vector here is the minimizer of f over D. If $D = \mathbb{R}^n$, the problem (1.1.10) is called an *unconstrained* optimisation problem.

Definition 1.1.1. (Global minimum). Let $\boldsymbol{x}_0 \in D \subseteq \mathbb{R}^n$ and $f : D \to \mathbb{R}$ such that $D \neq \emptyset$. If $f(\boldsymbol{x}) \geq f(\boldsymbol{x}_0) \; \forall \; \boldsymbol{x} \in D$, then point $\boldsymbol{x}_0$ is called a global minimum point.

Definition 1.1.2. (Local minimum). Let $\boldsymbol{x}_0 \in D \subseteq \mathbb{R}^n$ and $f : D \to \mathbb{R}$ with $D \neq \emptyset$. If $f(\boldsymbol{x}) \geq f(\boldsymbol{x}_0) \; \forall \; \boldsymbol{x} \in D \cap \mathcal{B}_{\boldsymbol{x}_0}(\varepsilon), \; \varepsilon > 0$, then point $\boldsymbol{x}_0$ is called a local minimum point. ($\mathcal{B}_{\boldsymbol{x}_0}(\varepsilon) = \|\boldsymbol{x} - \boldsymbol{x}_0 < \varepsilon\|$)

Definition 1.1.3. (Argument of the minimum). Given an arbitrary set X, and a function, $f\colon X \to Y$, the argument of the minimum (written $\arg\min$) over some subset, S, of X is defined by

$$\arg\min_{x \in S} f(x) := \{x \in S | \; \forall y \in S : f(y) \geq f(x)\}$$

They are points x for which $f(x)$ attains its smallest value.

Definition 1.1.4. (Local and global minimizer). Let $f : D \to \mathbb{R}$. If $\exists\ \varepsilon > 0$ such that $f(x) \geq f(x_0)\ \forall\ x \in D\backslash\{x_0\}$, and $\forall\ \|x - x_0\| < \varepsilon$. Then the point $x_0 \in D$ is a local minimizer of f over D. If $f(x) \geq f(x_0)\ \forall\ x \in D\backslash\{x_0\}$, then $x_0 \in D$ is a global minimizer of f over D

Definition 1.1.5. (Convex set). Let $\boldsymbol{x}_1, \boldsymbol{x}_2 \in D \neq \emptyset$ and $0 \leq \varphi \leq 1$. Then D is convex provided that

$$\varphi \boldsymbol{x}_1 + (1 - \varphi)\boldsymbol{x}_2 \in D$$

Definition 1.1.6. (Convex functions). Let $f : \mathbb{R}^n \to \mathbb{R}$ be defined on a non empty set $D \subseteq \mathbb{R}^n$. Then f is convex on D if $\forall\ \boldsymbol{x}_1, \boldsymbol{x}_2 \in D$ and $\varphi \in [0, 1]$

$$f(\varphi \boldsymbol{x}_1 + (1 - \varphi)\boldsymbol{x}_2) \leq \varphi f(\boldsymbol{x}_1) + (1 - \varphi) f(\boldsymbol{x}_2) \tag{1.1.11}$$

f is strictly convex if the inequality is always strict, i.e. if $\boldsymbol{x}_1 \neq \boldsymbol{x}_2$ implies that

$$f(\varphi \boldsymbol{x}_1 + (1 - \varphi)\boldsymbol{x}_2) \leq \varphi f(\boldsymbol{x}_1) + (1 - \varphi) f(\boldsymbol{x}_2) \tag{1.1.12}$$

f is concave if $-f$ is convex. If the graph of a convex function is drawn, any line joining any two points in the graph would be above the graph.

Definition 1.1.7. (Strong convexity). A differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be μ-strongly convex if:

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T(\boldsymbol{y} - \boldsymbol{x}) + \frac{\mu}{2}\|\boldsymbol{y} - \boldsymbol{x}\|^2 \tag{1.1.13}$$

for some $\mu > 0$ and $\forall\ \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$. Strong convexity does not necessarily require the function to be differentiable, and the gradient is replaced by the subgradient when the function is non-smooth. The set of subgradients is the subdifferential $\partial f(\boldsymbol{x})$, given by:

$$\partial f(\boldsymbol{x}) = \{\boldsymbol{g} : f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \boldsymbol{g}^T(\boldsymbol{y} - \boldsymbol{x}),\ \forall \boldsymbol{y}\} \tag{1.1.14}$$

where $\boldsymbol{g} \in \mathbb{R}^n$ is the subgradient at point $\boldsymbol{x}$.

Proposition 1.1.1. The following conditions are all equivalent to the condition that a differentiable function f is strongly-convex with constant $\mu > 0$.

(a) $f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T(\boldsymbol{y} - \boldsymbol{x}) + \frac{\mu}{2}\|\boldsymbol{y} - \boldsymbol{x}\|^2,\ \forall \boldsymbol{x}, \boldsymbol{y}$

(b) $g(\boldsymbol{x}) = f(\boldsymbol{x}) - \frac{\mu}{2}\|\boldsymbol{x}\|^2$ is convex $\forall\ \boldsymbol{x}$

(c) $(\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y}))^T(\boldsymbol{x} - \boldsymbol{y}) \geq \mu\|\boldsymbol{x} - \boldsymbol{y}\|^2,\ \forall\ \boldsymbol{x}, \boldsymbol{y}$

(d) $f(\varphi\boldsymbol{x} + (1-\varphi)\boldsymbol{y}) \leq \varphi f(\boldsymbol{x}) + (1-\varphi)f(\boldsymbol{x}) - \frac{\varphi(1-\varphi)\mu}{2}\|\boldsymbol{x} - \boldsymbol{y}\|^2,\ \varphi \in [0,1]$

Lemma 1.1.1. If f is twice continuously differentiable, then it is strongly convex with parameter $\mu > 0$ if and only if $\nabla^2 f(\boldsymbol{x}) \succeq \mu I,\ \forall\ \boldsymbol{x}$, where I is the identity matrix. This means that the matrix $\nabla^2 f(\boldsymbol{x}) - \mu I$ is positive semi definite.

Proposition 1.1.2. Let $h = f + g$ where f is strongly convex function and g is a convex function, then h is a strongly convex function.

Proof. For $\varphi \in [0,1]$, $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$, and $\mu > 0$, we have

$$\begin{aligned}
h(\varphi\boldsymbol{x} + (1-\varphi)\boldsymbol{y}) &= f(\varphi\boldsymbol{x} + (1-\varphi)\boldsymbol{y}) + g(\varphi\boldsymbol{x} + (1-\varphi)\boldsymbol{y}) \\
&\leq \varphi f(\boldsymbol{x}) + (1-\varphi)f(\boldsymbol{y}) - \frac{\mu\varphi(1-\varphi)}{2}\|\boldsymbol{x} - \boldsymbol{y}\|^2 + \varphi g(\boldsymbol{x}) + (1-\varphi)g(\boldsymbol{y}) \\
&= \varphi[f(\boldsymbol{x}) + g(\boldsymbol{x})] + (1-\varphi)[f(\boldsymbol{y}) + g(\boldsymbol{y})] - \frac{\mu\varphi(1-\varphi)}{2}\|\boldsymbol{x} - \boldsymbol{y}\|^2 \\
&= \varphi h(\boldsymbol{x}) + (1-\varphi)h(\boldsymbol{y}) - \frac{\mu\varphi(1-\varphi)}{2}\|\boldsymbol{x} - \boldsymbol{y}\|^2
\end{aligned}$$

□

Theorem 1.1.1. Any local minimum of a convex function $f : \mathbb{R}^n \to \mathbb{R}$ is also a global minimum.

Lemma 1.1.2. The weighted sums of m convex functions $f_1, f_2, \ldots, f_m$

$$f := \sum_{i=1}^{m} \alpha_i f_i$$

is convex as long as the weights $\alpha_1, \alpha_2, \ldots, \alpha_m$ are non-negative.

Proof. By convexity of $f_1, f_2, \ldots, f_m$, for any $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{R}^m$ and $\varphi \in [0,1]$

$$\begin{aligned}
f(\varphi\boldsymbol{x}_1 + (1-\varphi)\boldsymbol{x}_2) &= \sum_{i=1}^{m} \alpha_i f_i(\varphi\boldsymbol{x}_1 + (1-\varphi)\boldsymbol{x}_2) \\
&\leq \sum_{i=1}^{m} \alpha_i \left(\varphi f_i(\boldsymbol{x}_1) + (1-\varphi)f_i(\boldsymbol{x}_2)\right) \\
&= \varphi f(\boldsymbol{x}_1) + (1-\varphi)f(\boldsymbol{x}_2)
\end{aligned}$$

which shows that f is convex. □

Definition 1.1.8. (Quadratic from of a symmetric matrix). The quadratic form of an $n \times n$ symmetric matrix $\boldsymbol{A}$, denoted by $Q_{\boldsymbol{A}}(\boldsymbol{x})$ is a function $f : \mathbb{R}^n \to \mathbb{R}$ given by $f(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{Q}\boldsymbol{x}^{\boldsymbol{T}}$, $\boldsymbol{x} \in \mathbb{R}^n$. $\boldsymbol{A}$ is

- positive definite, if $Q_{\boldsymbol{A}}(\boldsymbol{x}) > 0 \; \forall \; \boldsymbol{x} \in \mathbb{R}^n$
- positive semi-definite, if $Q_{\boldsymbol{A}}(\boldsymbol{x}) \geq 0 \; \forall \; \boldsymbol{x} \in \mathbb{R}^n$
- negative definite, if $Q_{\boldsymbol{A}}(\boldsymbol{x}) < 0 \; \forall \; \boldsymbol{x} \in \mathbb{R}^n$
- negative semi-definite, if $Q_{\boldsymbol{A}}(\boldsymbol{x}) \leq 0 \; \forall \; \boldsymbol{x} \in \mathbb{R}^n$

Definition 1.1.9. (Hessian matrix). Suppose $f : \mathbb{R}^n \to \mathbb{R}$, $\boldsymbol{x} \in \mathbb{R}^n$. If all second partial derivatives of f exist and are continuous over the domain of the function (i.e. $f \in C^2(dom\ f)$), then the Hessian matrix $\nabla^2(f(\boldsymbol{x}))$ of f is a symmetric $n \times n$ matrix, defined as:

$$\nabla^2 f(\boldsymbol{x}) = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1 \partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2 \partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n \partial x_1} & \dfrac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{1.1.15}$$

or, by stating an equation for the coefficients using indices i and j:

$$\nabla^2 f(\boldsymbol{x}) = \left(\frac{\partial^2 f}{\partial x_i \partial x_j} \right)_{1 \leq i,j \leq n} \tag{1.1.16}$$

Theorem 1.1.2. (First order condition). Let $f : \mathbb{R}^n \to \mathbb{R}$, $D \subset \mathbb{R}^n$ a convex set. If $f \in C^1(D)$, then f is convex if and only if for every $\boldsymbol{x}, \boldsymbol{y} \in D$

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \nabla^T f(\boldsymbol{x})(\boldsymbol{y} - \boldsymbol{x}) \tag{1.1.17}$$

Theorem 1.1.3. (Second order condition). Let $f : \mathbb{R}^n \to \mathbb{R}$. If $f \in C^2(dom\ f)$, then f is convex if and only if for every $\boldsymbol{x} \in \mathbb{R}^n$, the Hessian matrix $\nabla^2 f(\boldsymbol{x})$ positive semi-definite.

1.1.6 Rate of Convergence

The speed at which a convergent sequence approaches its limit is called the rate of convergence of the sequence. Rate of convergence is of practical importance when working with a sequence of successive approximations for an iterative method, as then fewer iterations are needed to yield a useful approximation if the rate of convergence is higher. One of the key measures of performance of an algorithm is its rate of convergence.

Definition 1.1.10. Let $\{\boldsymbol{x}_k\}$ be a sequence in $\mathbb{R}^n$ that converges to x^*. Then,

i) $\{\boldsymbol{x}_k\}$ is said to converge *Q-linearly* to x^* if there exists $\beta \in (0,1)$, such that

$$\frac{\|\boldsymbol{x}_{k+1} - x^*\|}{\|\boldsymbol{x}_k - x^*\|} \leq \beta$$

This means that the distance to the solution x^* decreases at each iteration by at least a constant factor.

ii) $\{\boldsymbol{x}_k\}$ is said to converge *Q-superlinearly* (i.e. faster than linearly) to x^* if

$$\lim_{k\to\infty} \frac{\|\boldsymbol{x}_{k+1} - x^*\|}{\|\boldsymbol{x}_k - x^*\|} = 0$$

iii) $\{\boldsymbol{x}_k\}$ is said to converge *Q-sublinearly* (i.e. slower than linearly) to x^* if

$$\lim_{k\to\infty} \frac{\|\boldsymbol{x}_{k+1} - x^*\|}{\|\boldsymbol{x}_k - x^*\|} = 1$$

iv) $\{\boldsymbol{x}_k\}$ converges with p-order of convergence for $p > 1$ if

$$\frac{\|\boldsymbol{x}_{k+1} - x^*\|}{\|\boldsymbol{x}_k - x^*\|^p} \leq M \quad \forall\ k \text{ sufficiently large}$$

for some positive constant M (not necessarily less than 1). When $q = 2$, the convergence is quadratic, cubic when $q = 3$ and so on.

The Q in the definition stands for quotient, because the definition uses the quotient between two successive errors.

1.1.7 Logistic Sigmoid Function

A sigmoid function is a bounded, differentiable, real function that is defined for all real input values and has a non-negative derivative at each point. It takes in a real number and outputs a number in range $(0, 1)$, having a characteristic "S"-shaped curve or sigmoid curve. A standard choice for a sigmoid function is the standard logistic function shown in Figure (1.1) and defined by the formula

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.1.18}$$

with the following properties

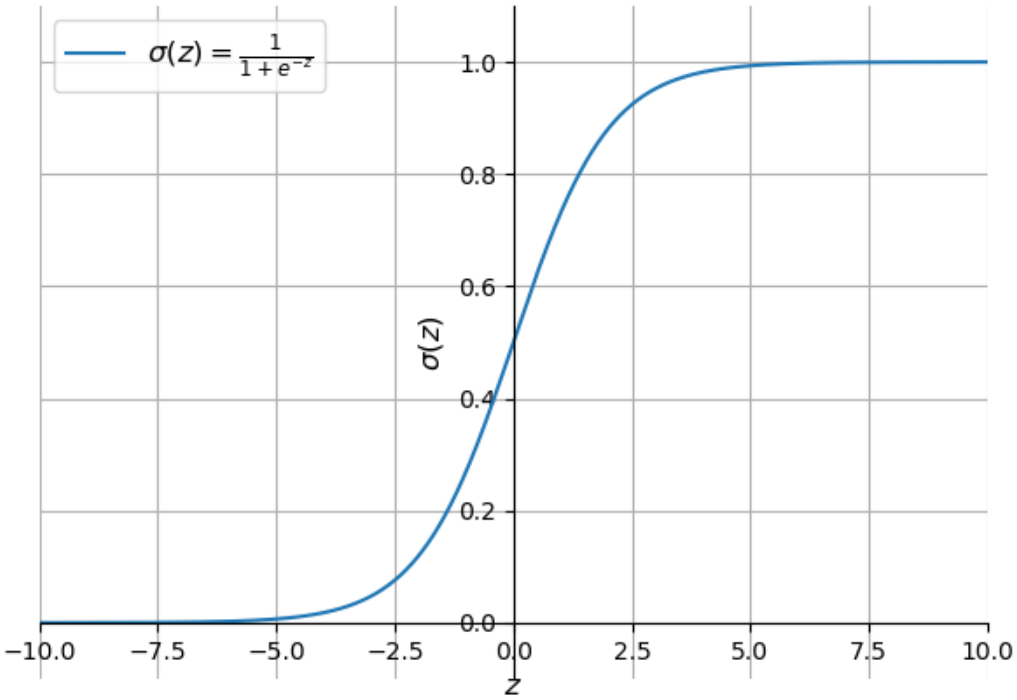


Figure 1.1: Logistic sigmoid function

- Horizontal asymptotes exists at $\sigma(x) = 0$ and $\sigma(x) = 1$, and y-intercept at $(0, 0.5)$

$$\lim_{x\to\infty} \frac{1}{1 + e^{-x}} = \frac{1}{1 + \lim_{x\to\infty} e^{-x}} = \frac{1}{1 + 0} = 1$$

$$\lim_{x\to-\infty} \frac{1}{1 + e^{-x}} = \frac{1}{1 + \lim_{x\to-\infty} e^{-x}} = \frac{1}{1 + \lim_{x\to\infty} e^{x}} = 0$$

- Symmetry property

$$\sigma(x) + \sigma(-x) = 1 \tag{1.1.19}$$

$$\frac{1}{1+e^{-x}} + \frac{1}{1+e^{-(-x)}} = \frac{e^x}{e^x+1} + \frac{1}{e^x+1} = 1$$

- Its derivative can be derived thus

$$\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] \\
&= \frac{d}{dx}\left(1+\mathrm{e}^{-x}\right)^{-1} \\
&= -(1+e^{-x})^{-2}(-e^{-x}) \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{(1+e^{-x})-1}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right) \\
&= \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \\
\sigma'(x) &= \sigma(x) \cdot (1-\sigma(x))
\end{aligned} \tag{1.1.20}$$

- Its integral is calculated thus

$$\int \frac{1}{1+e^{-x}}\, dx = \int \frac{e^x}{e^x+1}\, dx = \ln|e^x+1| + C \tag{1.1.21}$$

1.2 Machine Learning

In the early days of "intelligent" applications, many systems used hand-coded rules of "if" and "else" decisions to process data or adjust to user input. An example is a spam filter whose job is to move the appropriate incoming email messages to a spam folder, a blacklist of words could be made up that would result in an email being marked as spam. This would be an example of using an expert-designed rule system to design an "intelligent" application. Manually crafting decision rules is feasible for some applications, particularly those in which humans have a good understanding of the process to model. However, using hand-coded rules to make decisions has two major disadvantages:

- The logic required to make a decision is specific to a single domain and task, changing the task even slightly might require a rewrite of the whole system.
- Designing rules requires a deep understanding of how a decision should be made by a human expert.

One example of where this hand-coded approach will fail is in detecting faces in images. The main problem is that the way in which pixels (which make up an image in a computer) are perceived by the computer is very different from how humans perceive a face. This difference in representation makes it basically impossible for a human to come up with a good set of rules to describe what constitutes a face in a digital image. Using machine learning, however, simply presenting a program with a large collection of labelled images of faces is enough for an algorithm to determine what characteristics are needed to identify a face.

Far better results can be obtained by adopting a machine learning approach in which a large set of D images $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_D\}$ called a *training set* is used to tune the parameters of an adaptive model. The categories of the images in the training set are known in advance, typically by inspecting them individually and hand-labelling them. We can express the category of an image using *target vector* $\boldsymbol{y}$, which represents the identity of the corresponding image. Note that there is one such target vector $\boldsymbol{y}$ for each image $\boldsymbol{x}$.

The result of running the machine learning algorithm can be expressed as a function $\boldsymbol{f}(\boldsymbol{x})$ which takes a new image $\boldsymbol{x}$ as input and that generates an output vector $\boldsymbol{f}$, encoded in the same way as the target vectors. The precise form of the function $\boldsymbol{f}(\boldsymbol{x})$ is determined during the *training* (or *learning*) phase, on the basis of the training data. Once the model is trained it can then determine the identity of new images, which are said to comprise a *test set*. The ability to categorize correctly new examples that differ from those used for training is known as *generalisation*. In practical applications, the variability of the input vectors will be such that the training data can comprise only a tiny fraction of all possible input vectors, and so generalisation is a central goal in machine learning.

For most practical applications, the original input variables are typically preprocessed to transform them into some new space of variables. For instance, in the face recognition problem, the images are typically translated and scaled so that each image is contained within a box of

a fixed size. This greatly reduces the variability within each image class, because the location and scale of all the images are now the same, which makes it much easier for a subsequent face recognition algorithm to distinguish between the different classes. This pre-processing stage is sometimes also called *feature extraction*. The new test data must be pre-processed using the same steps as the training data. Care must be taken during pre-processing because often information is discarded, and if this information is important to the solution of the problem then the overall accuracy of the system can suffer.

Applications in which the training data comprises examples of the input vectors $\boldsymbol{x}$ along with their corresponding target vectors $\boldsymbol{y}$ are known as *supervised learning* problems. Cases such as the face recognition example, in which the aim is to assign each input vector to one of a finite number of discrete categories, are called *classification problems*. If the desired output consists of one or more continuous variables, then the task is called a *regression problem*. An example of a regression problem would be the prediction of the stock market prices.

In other pattern recognition problems, the training data consists of a set of input vectors $\boldsymbol{x}$ without any corresponding target values. The goal in such *unsupervised learning* problems may be to discover groups of similar examples within the data, where it is called *clustering*, or to determine the distribution of data within the input space, known as *density estimation*, or to project the data from a high-dimensional space down to two or three dimensions for the purpose of *visualization*.

The technique of *reinforcement learning*, **Sutton and Barto (1998)** is concerned with the problem of finding suitable actions to take in a given situation in order to maximize a reward. Here the learning algorithm is not given examples of optimal outputs, in contrast to supervised learning, but must instead discover them by a process of trial and error. Typically there is a sequence of states and actions in which the learning algorithm is interacting with its environment. In many cases, the current action not only affects the immediate reward but also has an impact on the reward at all subsequent time steps. Our focus in this project would be on classification problems.

1.3 Logistic Regression

Logistic regression is an algorithm for for binary classification. In binary classification, the goal is to learn a classifier that can input data (e.g. an image) represented by feature vector $\boldsymbol{x}$ and present a label $y \in \{0, 1\}$. A single training example is represented by the pair $(\boldsymbol{x}, y)$, $\boldsymbol{x} \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$, where n_x is the dimension of the vector $\boldsymbol{x}$, and m training set is represented as $\{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$. To put all the training examples in more compact notation, we define matrix $\boldsymbol{X} \in \mathbb{R}^{n_x \times m}$ by stacking the training examples column wise as:

$$\boldsymbol{X} = \begin{bmatrix} | & | & & | \\ \boldsymbol{x}^{(1)} & \boldsymbol{x}^{(2)} & \ldots & \boldsymbol{x}^{(m)} \\ | & | & & | \end{bmatrix} \tag{1.3.1}$$

and vector $\boldsymbol{y} \in \mathbb{R}^{1 \times m}$ as

$$\boldsymbol{y} = \begin{bmatrix} y^{(1)} & y^{(2)} & \ldots & y^{(m)} \end{bmatrix} \tag{1.3.2}$$

To transform the feature vector $\boldsymbol{x}$ into a scalar we can input to the probabilistic function for prediction, the linear predictive model with parameters, weight $\boldsymbol{w} \in \mathbb{R}^{n_x}$, and bias $b \in \mathbb{R}$

$$z = \boldsymbol{w}^T \boldsymbol{x} + b \tag{1.3.3}$$

is used. The probabilistic sigmoid function $a = \sigma(z)$ as defined in equation (1.1.18) is then used to estimate the class of a new input. For a given point $\boldsymbol{x}$, the prediction is estimated thus

$$\text{Prediction} = \begin{cases} 0, & \text{if } a < 0.5 \\ 1, & \text{otherwise} \end{cases} \tag{1.3.4}$$

Given feature vector $\boldsymbol{x}$, the aim is to output prediction $\hat{y}$ which is an estimate of t, i.e

$$\hat{y} = P(y = 1|\boldsymbol{x}) = a = \frac{1}{1 + e^{-z}} \tag{1.3.5}$$

1.4 Maximum Likelihood Estimation

To learn the parameters of the model, given the training set $\{(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$, we want to find parameters $\boldsymbol{w}$ and b such that the predictions on the training set $a^{(i)}$ which

would be close to the labels $y^{(i)}$. We devise an error function, which describes the error created by any arbitrary values of $\boldsymbol{w}$ and b, and then we minimise the output of the function. The optimal value of $\boldsymbol{w}$ and b gotten can be used for future predictions. This function is derived using the Bernoulli probability distribution.

The Bernoulli trials which have two possible, mutually exclusive outcomes; success where $y = 1$ and failure where $y = 0$. In a Bernoulli trial, the probability of success $P(y = 1|\boldsymbol{x})$ and failure $P(y = 0|\boldsymbol{x})$ sum to 1, i.e

$$P(y = 0|\boldsymbol{x}) = 1 - P(y = 1|\boldsymbol{x}) \tag{1.4.1}$$

The output thus belong to a Bernoulli distribution which is a special case of binomial distribution where $n = 1$ in equation (1.1.4). Thus

$$P(Y = y) = a^y(1 - a)^{1-y} = \begin{cases} a & \text{if} \quad y = 1 \\ 1 - a & \text{if} \quad y = 0 \end{cases} \tag{1.4.2}$$

Equation (1.4.2) gives a good classifier for our task since it produces very high a when $y = 1$, and conversely very low a when $y = 0$. To make a very close to y for each sample, we would want to maximize

$$a^y.(1 - a)^{1-y}$$

over parameters $\boldsymbol{w}$ and b, i.e.

$$\arg\max_{\mathbf{w},b} a^y.(1 - a)^{1-y}$$

which is equivalent to

$$\arg\max_{\mathbf{w},b} \log(a^y.(1 - a)^{1-y}) = \arg\max_{\mathbf{w},b} [y \log(a) + (1 - y) \log(1 - a)] \tag{1.4.3}$$

since the log function is a monotonically increasing function. We can rewrite equation (1.4.3) as minimizing the function L over $\boldsymbol{w}$ and b

$$\mathrm{L}(a, y) = -[y \log(a) + (1 - y) \log(1 - a)] \tag{1.4.4}$$

equation (1.4.4) is called the *loss function* L which measures how good the output a is when given the true label y on a single training example $(\boldsymbol{x}, y)$.

$$\mathrm{L}(a, y) = \begin{cases} -\log(a) & \text{if} \quad y = 1 \\ -\log(1 - a) & \text{if} \quad y = 0 \end{cases} \tag{1.4.5}$$

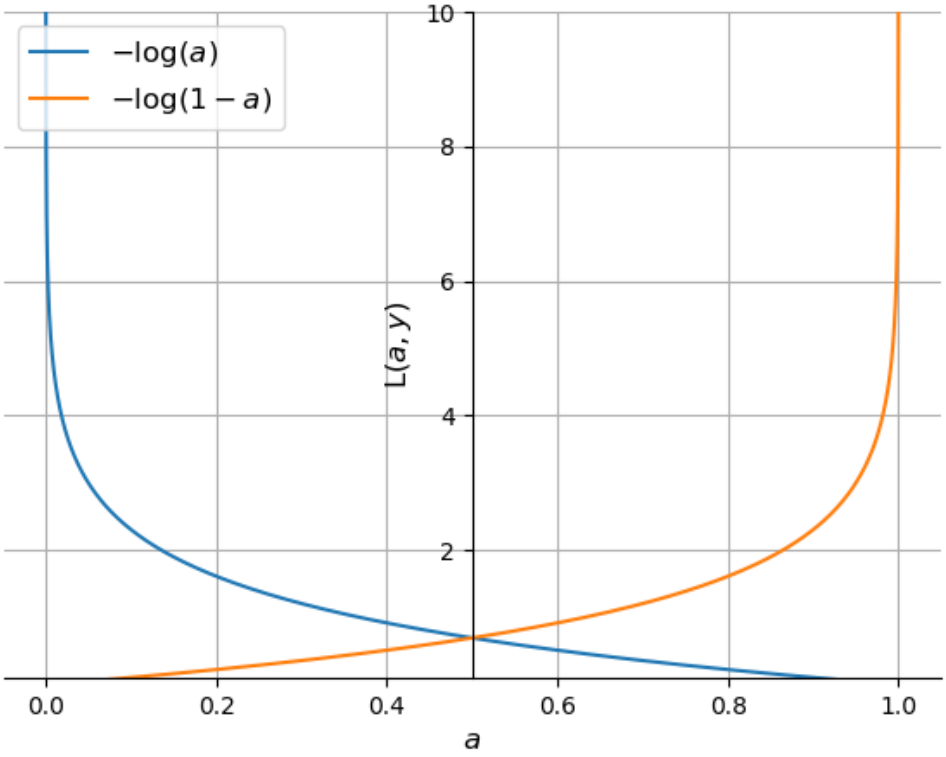


Figure 1.2: Loss function for $y = 0$ and $y = 1$

Next we define the overall loss function on the entire training set, i.e. on m multiple samples. Assuming that the training examples are independent and identically distributed, then

$$P(Y = y) = \prod_{i=1}^{m} P(y^{(i)}|\boldsymbol{x}^{(i)})$$

From equation (1.4.2), we would want to maximize

$$\prod_{i=1}^{m} a^{(i)^{y^{(i)}}} \cdot \left(1 - a^{(i)}\right)^{\left(1-y^{(i)}\right)}$$

over $\boldsymbol{w}$ and b for multiple samples, which is equal to

$$\begin{aligned}\arg\max_{\mathbf{w},b} \log\left(\prod_{i=1}^{m} a^{(i)^{y^{(i)}}} \cdot \left(1 - a^{(i)}\right)^{\left(1-y^{(i)}\right)}\right) &= \arg\max_{\mathbf{w},b} \sum_{i=1}^{m} \log\left(a^{(i)^{y^{(i)}}} \cdot \left(1 - a^{(i)}\right)^{\left(1-y^{(i)}\right)}\right) \\ &= \arg\max_{\mathbf{w},b} \sum_{i=1}^{m} y^{(i)} \log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)\end{aligned}$$

or, equivalently, we want to minimize

$$\begin{aligned} \mathrm{J}(\boldsymbol{w}, b) &= -\frac{1}{m}\sum_{i=1}^{m}\left[\left(y^{(i)}\log\left(a^{(i)}\right) + \left(1-y^{(i)}\right)\log\left(1-a^{(i)}\right)\right] \\ &= \frac{1}{m}\sum_{i=1}^{m}\mathrm{L}\left(a^{(i)}, y^{(i)}\right) \end{aligned} \tag{1.4.6}$$

Equation (1.4.6) is called the *cost function* which measures how well parameters $\boldsymbol{w}$ and b are doing on the entire training set $(\boldsymbol{X}, \boldsymbol{y})$. It would be required to show that $\mathrm{J}(\boldsymbol{w}, b)$ given in equation (1.4.6) is a convex function which guarantees a local minimum, which is a global minimum (from Theorem 1.1.1) for the function.

$$\begin{aligned} \frac{\partial \mathrm{L}}{\partial a} &= \frac{\partial}{\partial a}\left(-\left(y\log(a) + (1-y)\log(1-a)\right)\right) \\ &= \frac{\partial}{\partial a}\left(-y\log(a)\right) + \frac{\partial}{\partial a}\left(-(1-y)\log(1-a)\right) \\ &= \frac{-y}{a} + \frac{1-y}{1-a} = \frac{a-y}{a(1-a)} \end{aligned} \tag{1.4.7}$$

From equation (1.1.20),

$$\frac{da}{dz} = a(1-a)$$

Using chain rule;

$$\frac{\partial \mathrm{L}}{\partial z} = \frac{\partial \mathrm{L}}{\partial a}\frac{da}{dz} = \frac{a-y}{a(1-a)}a(1-a) = a-y \tag{1.4.8}$$

$$\frac{\partial \mathrm{L}}{\partial \boldsymbol{w}} = \frac{\partial \mathrm{L}}{dz}\frac{\partial z}{\partial \boldsymbol{w}} = (a-y)\boldsymbol{x}, \quad \frac{\partial \mathrm{L}}{\partial b} = \frac{\partial \mathrm{L}}{\partial z}\frac{\partial z}{\partial b} = (a-y)$$

Hence

$$\frac{\partial \mathrm{J}}{\partial \boldsymbol{w}} = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)})\boldsymbol{x}^{(i)}, \quad \frac{\partial \mathrm{J}}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)}) \tag{1.4.9}$$

If it can be proved that the functions $-\log a$ and $-\log(1-a)$ are convex functions of $\boldsymbol{w}$ and b, then the objective function

$$\mathrm{J}(\boldsymbol{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left[\left(y^{(i)}\log\left(a^{(i)}\right) + \left(1-y^{(i)}\right)\log\left(1-a^{(i)}\right)\right]$$

must also be convex since any linear combination of two or more convex functions is also convex (Lemma 1.1.2). Using the second order condition from Theorem 1.1.3. Deriving the

Hessian of $-\log(a)$ with respect to $\boldsymbol{w}$

$$\begin{aligned}
\nabla^2_{\boldsymbol{w}}[-\log(a)] &= \nabla_{\boldsymbol{w}}\{\nabla_{\boldsymbol{w}}[-\log(a)]\} = \nabla_{\boldsymbol{w}}\left\{\frac{d(-\log(a))}{da}.\frac{da}{dz}.\frac{\partial z}{\partial \boldsymbol{w}}\right\} \\
&= \nabla_{\boldsymbol{w}}\left\{-\frac{1}{a}.[a(1-a)].\boldsymbol{x}\right\} = \nabla_{\boldsymbol{w}}\left\{(a-1).\boldsymbol{x}\right\} \\
&= \frac{d((a-1).\boldsymbol{x})}{da}.\frac{da}{dz}.\frac{\partial z}{\partial \boldsymbol{w}} = \boldsymbol{x}.[a(1-a)].\boldsymbol{x} \\
\nabla^2_{\boldsymbol{w}}[-\log(a)] &= a(1-a)\boldsymbol{x}\boldsymbol{x}^T
\end{aligned} \tag{1.4.10}$$

Similarly, for the Hessian of $-\log(a)$ with respect to b

$$\begin{aligned}
\nabla^2_{b}[-\log(a)] &= \nabla_{b}\{\nabla_{b}[-\log(a)]\} = \nabla_{b}\left\{\frac{d(-\log(a))}{da}.\frac{da}{dz}.\frac{\partial z}{\partial b}\right\} \\
&= \nabla_{b}\left\{-\frac{1}{a}.[a(1-a)]\right\} = \nabla_{b}\left\{(a-1)\right\} \\
&= \frac{d((a-1))}{da}.\frac{da}{dz}.\frac{\partial z}{\partial \boldsymbol{w}} = a(1-a) \\
\nabla^2_{b}[-\log(a)] &= a(1-a)
\end{aligned} \tag{1.4.11}$$

To show that the Hessian matrices in equation 1.4.10 and 1.4.11 are positive semi-definite

$$\begin{aligned}
\forall\ \boldsymbol{q} \in \mathbb{R}^n :\quad \boldsymbol{q}^T\nabla^2_{\boldsymbol{w}}[-\log(a)]\boldsymbol{q} &= \boldsymbol{q}^T\left[a(1-a)\boldsymbol{x}\boldsymbol{x}^T\right]\boldsymbol{q} \\
&= a(1-a)(\boldsymbol{x}^T\boldsymbol{q})^2 \geq 0
\end{aligned} \tag{1.4.12}$$

$$\begin{aligned}
\forall\ \boldsymbol{q} \in \mathbb{R}^n :\quad \boldsymbol{q}^T\nabla^2_{\boldsymbol{w}}[-\log(a)]\boldsymbol{q} &= \boldsymbol{q}^T\left[a(1-a)\boldsymbol{x}\boldsymbol{x}^T\right]\boldsymbol{q} \\
&= a(1-a)(\boldsymbol{x}^T\boldsymbol{q})^2 \geq 0
\end{aligned} \tag{1.4.13}$$

Deriving the Hessian of $-\log(1-a)$ with respect to $\boldsymbol{w}$

$$\begin{aligned}
\nabla^2_{\boldsymbol{w}}[-\log(1-a)] &= \nabla_{\boldsymbol{w}}\{\nabla_{\boldsymbol{w}}[-\log(1-a)]\} = \nabla_{\boldsymbol{w}}\left\{\frac{d(-\log(1-a))}{da}.\frac{da}{dz}.\frac{\partial z}{\partial \boldsymbol{w}}\right\} \\
&= \nabla_{\boldsymbol{w}}\left\{-\frac{1}{1-a}.[a(1-a)].\boldsymbol{x}\right\} = \nabla_{\boldsymbol{w}}\left\{-a\boldsymbol{x}\right\} \\
&= \frac{d(-a\boldsymbol{x})}{da}.\frac{da}{dz}.\frac{\partial z}{\partial \boldsymbol{w}} = \boldsymbol{x}.[a(1-a)].\boldsymbol{x} \\
\nabla^2_{\boldsymbol{w}}[-\log(a)] &= a(1-a)\boldsymbol{x}\boldsymbol{x}^T
\end{aligned} \tag{1.4.14}$$

Similarly, for the Hessian of $-\log(1-a)$ with respect to b

$$\begin{aligned}\nabla_b^2[-\log(1-a)] &= \nabla_b\{\nabla_b[-\log(1-a)]\} = \nabla_b\left\{\frac{d(-\log(1-a))}{da}.\frac{da}{dz}.\frac{\partial z}{\partial b}\right\} \\ &= \nabla_b\left\{-\frac{1}{1-a}.[a(1-a)]\right\} = \nabla_b\{-a\} \\ &= \frac{d(-a)}{da}.\frac{da}{dz}.\frac{\partial z}{\partial \boldsymbol{w}} = a(1-a) \\ \nabla_b^2[-\log(a)] &= a(1-a) \end{aligned} \tag{1.4.15}$$

It has been shown that equations (1.4.14) and (1.4.15) are positive semi-definite in equations (1.4.12) and (1.4.13) respectively. Hence the cost function $\mathrm{J}(\boldsymbol{w}, b)$ given in (1.4.6) is convex which guarantees a global minimum for the function.

1.5 Gradient Descent

Gradient-based learning procedures have been used since the late 1950's but they were mostly limited to linear systems **Duda and Hart (1973)**. The surprising usefulness of such simple gradient descent techniques for complex machine learning tasks was not widely realized until it was realised that:

- The presence of a local minima for the loss function was not a major problem in practice. In fact, the local minima was not an hindrance to the success of early non-linear gradient-based learning techniques such as Boltzmann machines **Ackley et al. (1985)**.

- The efficiency of the back-propagation algorithm to compute the gradient in a non-linear system composed of several layers of processing.

- The demonstration that the back-propagation procedure applied to multi layer neural networks with sigmoidal units can solve complicated learning tasks.

To find the parameters $\boldsymbol{w},\ b$ that minimizes the cost function $\mathrm{J}(\boldsymbol{w},\ b)$, we use the gradient descent algorithm; which is an optimisation algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. It involves the following steps:

1. Initialize parameters $\boldsymbol{w},\ b$ to some initial value, and calculate $\mathrm{J}(\boldsymbol{w},\ b)$. We can initialize to zeros or any random number, initializing to zeros is preferable for logistic regression. The same global minimum point would be reached regardless of initial point since the cost function is convex.

2. Take a step in steepest downhill direction using the negative gradient with respect to $\boldsymbol{w},\ b$. The size of these steps is called the learning rate, α. With a high learning rate we can cover more ground each step, but we risk overshooting the lowest point since the slope of the function is constantly changing. With a very low learning rate, we can confidently move in the direction of the negative gradient since we are recalculating it so frequently, a low learning rate is more precise.

3. Update the parameters with the gradients to reach the optimal values where $\mathrm{J}(\boldsymbol{w},\ b)$ is minimized.

4. Repeat steps 2 and 3 till further adjustments to parameters does not significantly reduce $\mathrm{J}(\boldsymbol{w},\ b)$.

The steps above can be summarised as;

$$\text{Repeat until convergence} \quad \{$$

$$\boldsymbol{w} := \boldsymbol{w} - \alpha \frac{\partial \mathrm{J}(\boldsymbol{w},\ b)}{\partial \boldsymbol{w}}$$

$$b := b - \alpha \frac{\partial \mathrm{J}(\boldsymbol{w},\ b)}{\partial b}$$

$$\}$$

Theoretically, considering the problem:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x} \in \mathbb{R}^m} f(\boldsymbol{x})$$

choosing a suitable learning rate $\alpha > 0$, gradient descent is an iterative method;

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \alpha \nabla f(\boldsymbol{x}^k) \tag{1.5.1}$$

where $f : \mathbb{R}^m \to \mathbb{R}$ is the objective function to be minimised, and $\boldsymbol{x} \in \mathbb{R}^m$. Next, we analyse the convergence rate of the gradient descent algorithm.

Theorem 1.5.1. Bottou et al. (2018) Suppose the function $f : \mathbb{R}^m \to \mathbb{R}$ is convex and twice continuously differentiable, and it is L-smooth (i.e. its gradient is Lipschitz continuous with constant $L > 0$):

$$\|\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})\|_2 \leq L\|\boldsymbol{x} - \boldsymbol{y}\|_2 \tag{1.5.2}$$

for any $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^m$. Gradient descent for k iterations with fixed learning rate $\alpha \leq 1/L$ satisfies

$$f\left(\boldsymbol{x}^{(k)}\right) - f\left(\boldsymbol{x}^*\right) \leq \frac{\left\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^*\right\|_2^2}{2\alpha k} \tag{1.5.3}$$

where $f\left(\boldsymbol{x}^*\right)$ is the optimal value of $f(\boldsymbol{x})$.
Intuitively, this means that gradient descent converges sublinearly with rate $\mathcal{O}(1/k)$.

Proof. If f is twice differentiable, then we have by Taylor first order expansion

$$\nabla f(\boldsymbol{x} + \alpha\boldsymbol{d}) = \nabla f(\boldsymbol{x}) + \int_0^\alpha \nabla^2 f(\boldsymbol{x} + t\boldsymbol{d})\boldsymbol{d}\, \mathrm{d}t$$

$$\implies \nabla f(\boldsymbol{x} + \alpha\boldsymbol{d}) - \nabla f(\boldsymbol{x}) = \int_0^\alpha \nabla^2 f(\boldsymbol{x} + t\boldsymbol{d})\boldsymbol{d}\, \mathrm{d}t$$

where $\boldsymbol{d} \in \mathbb{R}^m$, $t \in (0, \alpha)$, taking norm of both sides we get

$$\left\| \int_0^\alpha \nabla^2 f(\boldsymbol{x} + t\boldsymbol{d})\boldsymbol{d}\, \mathrm{d}t \right\|_2 \leq L\alpha\|\boldsymbol{d}\|_2$$

dividing through by $\alpha\|\boldsymbol{d}\|$, $\boldsymbol{d} \neq 0$

$$\frac{\left\|\int_0^\alpha \nabla^2 f(\boldsymbol{x} + t\boldsymbol{d})\boldsymbol{d}\, \mathrm{d}t\right\|_2}{\alpha\|\boldsymbol{d}\|_2} \leq L \implies \frac{\|\alpha\nabla^2 f(\boldsymbol{x})\boldsymbol{d}\|_2}{\alpha\|\boldsymbol{d}\|_2} + O(\alpha) \leq L$$

taking limit as $\alpha \to 0$, we get

$$= \frac{\|\alpha\nabla^2 f(\boldsymbol{x})\boldsymbol{d}\|_2}{\boldsymbol{d}} \leq L, \quad \forall\, \boldsymbol{d} \neq 0 \in \mathbb{R}^m$$

Taking the supremum over $\boldsymbol{d} \neq 0 \in \mathbb{R}^m$ in the above gives

$$\nabla^2 f(\boldsymbol{x}) \preceq LI \tag{1.5.4}$$

or equivalently that $\nabla^2 f(\boldsymbol{x}) - LI$ is a negative semi-definite matrix. Hence,

$$(\boldsymbol{x} - \boldsymbol{y})^T(\nabla^2 f(\boldsymbol{x}) - LI)(\boldsymbol{x} - \boldsymbol{y}) \leq 0 \tag{1.5.5}$$

where the LHS of (1.5.5) is the quadratic form of $\nabla^2 f(\boldsymbol{x}) - LI$.

Since $L > 0$, and using the non-negativity property of norm, we have that

$$L\|\boldsymbol{x} - \boldsymbol{y}\|_2^2 \geq 0 \tag{1.5.6}$$

Combining equations (1.5.5) and (1.5.6), we get

$$L\|\boldsymbol{x} - \boldsymbol{y}\|_2^2 \geq (\boldsymbol{x} - \boldsymbol{y})^T \nabla^2 f(\boldsymbol{x})(\boldsymbol{x} - \boldsymbol{y}) \tag{1.5.7}$$

Performing a quadratic expansion of f around $f(x)$;

$$\begin{aligned} f(\boldsymbol{y}) &= f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T(\boldsymbol{y} - \boldsymbol{x}) + \frac{1}{2}(\boldsymbol{y} - \boldsymbol{x})^T \nabla^2 f(\boldsymbol{x})(\boldsymbol{y} - \boldsymbol{x}) \\ &\leq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T(\boldsymbol{y} - \boldsymbol{x}) + \frac{1}{2}L\|\boldsymbol{y} - \boldsymbol{x}\|_2^2 \qquad \text{from (1.5.7)} \end{aligned}$$

Plugging in gradient descent update by letting $y = \boldsymbol{x}^+ = \boldsymbol{x} - \alpha \nabla f(\boldsymbol{x})$, we get:

$$\begin{aligned} f(\boldsymbol{x}^+) &\leq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T \left(\boldsymbol{x}^+ - \boldsymbol{x}\right) + \frac{1}{2}L \left\|\boldsymbol{x}^+ - \boldsymbol{x}\right\|_2^2 \\ &= f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T(\boldsymbol{x} - \alpha \nabla f(\boldsymbol{x}) - \boldsymbol{x}) + \frac{1}{2}L\|\boldsymbol{x} - \alpha \nabla f(\boldsymbol{x}) - \boldsymbol{x}\|_2^2 \\ &= f(\boldsymbol{x}) - \nabla f(\boldsymbol{x})^T \alpha \nabla f(\boldsymbol{x}) + \frac{1}{2}L\|\alpha \nabla f(\boldsymbol{x})\|_2^2 \\ &= f(\boldsymbol{x}) - \alpha\|\nabla f(\boldsymbol{x})\|_2^2 + \frac{1}{2}L^2\|\nabla f(\boldsymbol{x})\|_2^2 \\ f(\boldsymbol{x}^+) &= f(\boldsymbol{x}) - \left(1 - \frac{1}{2}L\alpha\right)\alpha\|\nabla f(\boldsymbol{x})\|_2^2 \end{aligned} \tag{1.5.8}$$

Using $\alpha \leq 1/L$, $-\left(1 - \frac{1}{2}L\alpha\right) = \frac{1}{2}L\alpha - 1 \leq \frac{1}{2}L\left(\frac{1}{L}\right) - 1 = -\frac{1}{2}$. Substituting this result into (1.5.8) we get:

$$f\left(\boldsymbol{x}^+\right) \leq f(\boldsymbol{x}) - \frac{1}{2}\alpha\|\nabla f(\boldsymbol{x})\|_2^2 \tag{1.5.9}$$

since $\frac{1}{2}\alpha\|\nabla f(\boldsymbol{x})\|_2^2$ will always be positive unless $\nabla f(\boldsymbol{x}) = 0$, this inequality implies that the objective function value strictly decreases with each iteration of gradient descent until it reaches the optimal value $f(\boldsymbol{x}) = f\left(\boldsymbol{x}x^*\right)$. But this convergence result only holds when we choose the learning rate α to be small enough, i.e. $\alpha \leq \frac{1}{L}$.

Next, we bound $f\left(\boldsymbol{x}^+\right)$, the objective value at the next iteration, in terms of $f\left(\boldsymbol{x}^*\right)$, the optimal objective value. Since f is convex,

$$\begin{aligned} f(\boldsymbol{x}^*) &\geq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T(\boldsymbol{x}^* - \boldsymbol{x}) \\ \implies f(\boldsymbol{x}) &\leq f(\boldsymbol{x}^*) + \nabla f(\boldsymbol{x})^T(\boldsymbol{x} - \boldsymbol{x}^*) \end{aligned} \tag{1.5.10}$$

Plugging this into (1.5.9) we obtain:

$$\begin{aligned}
f(\boldsymbol{x}^+) &\leq f(\boldsymbol{x}^*) + \nabla f(\boldsymbol{x})^T(\boldsymbol{x} - \boldsymbol{x}^*) - \frac{\alpha}{2}\|\nabla f(\boldsymbol{x})\|_2^2 \\
\implies f(\boldsymbol{x}^+) - f(\boldsymbol{x}^*) &\leq \frac{1}{2\alpha}\left[\, 2\alpha\nabla f(\boldsymbol{x})^T(\boldsymbol{x} - \boldsymbol{x}^*) - \alpha^2\|\nabla f(\boldsymbol{x})\|_2^2 \,\right] \\
\implies f(\boldsymbol{x}^+) - f(\boldsymbol{x}^*) &\leq \frac{1}{2\alpha}\left[\, 2\alpha\nabla f(\boldsymbol{x})^T(\boldsymbol{x} - \boldsymbol{x}^*) - \alpha^2\|\nabla f(\boldsymbol{x})\|_2^2 - \|\boldsymbol{x} - \boldsymbol{x}^*\|_2^2 + \|\boldsymbol{x} - \boldsymbol{x}^*\|_2^2 \,\right] \\
\implies f(\boldsymbol{x}^+) - f(\boldsymbol{x}^*) &\leq \frac{1}{2\alpha}\left[\, \|\boldsymbol{x} - \boldsymbol{x}^*\|_2^2 - \|\boldsymbol{x} - \alpha\nabla f(\boldsymbol{x}) - \boldsymbol{x}^*\|_2^2 \,\right]
\end{aligned} \tag{1.5.11}$$

Substituting a gradient descent update $\boldsymbol{x}^+ = \boldsymbol{x} - \alpha\nabla f(\boldsymbol{x})$ in (1.5.11), we get:

$$f\left(\boldsymbol{x}^+\right) - f\left(\boldsymbol{x}^*\right) \leq \frac{1}{2\alpha}\left(\|\boldsymbol{x} - \boldsymbol{x}^*\|_2^2 - \left\|\boldsymbol{x}^+ - \boldsymbol{x}^*\right\|_2^2\right) \tag{1.5.12}$$

This inequality holds for $\boldsymbol{x}^+$ on every iteration of gradient descent. Summing over iterations, we get:

$$\begin{aligned}
\sum_{i=1}^{k} f(\boldsymbol{x}^{(i)}) - f(\boldsymbol{x}^*) &\leq \sum_{i=1}^{k} \frac{1}{2\alpha}\left(\left\|\boldsymbol{x}^{(i-1)} - \boldsymbol{x}^*\right\|_2^2 - \left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^*\right\|_2^2\right) \\
&= \frac{1}{2\alpha}\left(\left\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^*\right\|_2^2 - \left\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^*\right\|_2^2\right) \\
&\leq \frac{1}{2\alpha}\left(\left\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^*\right\|_2^2\right)
\end{aligned} \tag{1.5.13}$$

where the summation on the right-hand side disappears because each subsequent terms cancel each other, leaving only the initial and final terms. Finally, using the fact that f decreases on every iteration, we can conclude that

$$\begin{aligned}
f\left(\boldsymbol{x}^{(k)}\right) - f\left(\boldsymbol{x}^*\right) &\leq \frac{1}{k}\sum_{i=1}^{k} f\left(\boldsymbol{x}^{(i)}\right) - f\left(\boldsymbol{x}^*\right) \\
&\leq \frac{\left\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^*\right\|_2^2}{2\alpha k}
\end{aligned} \tag{1.5.14}$$

which gives the inequality (1.5.3). □

Theorem 1.5.2. Bottou et al. (2018) Suppose the function $f : \mathbb{R}^m \to \mathbb{R}$ is μ-strongly convex and L-smooth. Gradient descent for k iterations with fixed learning rate $\alpha \leq 2/(\mu + L)$ satisfies

$$f\left(\boldsymbol{x}^{(k)}\right) - f\left(\boldsymbol{x}^*\right) \leq c^k \frac{L}{2}\left\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^*\right\|_2^2 \tag{1.5.15}$$

where $0 < c < 1$ and $\mu > 0$.

This means that gradient descent on a strong convex function converges linearly with rate $\mathcal{O}(c^k)$ which is a faster convergence rate than that of convex functions.

One of the key problems in gradient descent is that we might overshoot the optimal value or make insufficient progress. A simple fix for the problem is to use line search in conjunction with gradient descent. That is, we use the direction given by $\nabla f(\boldsymbol{x})$ and then perform binary search as to which learning rate α minimizes $f(\boldsymbol{x} - \alpha \nabla f(\boldsymbol{x}))$. This algorithm converges rapidly (**Boyd et al. (2004)**). However, for the purpose of deep learning this is not quite so feasible, since each step of the line search would require us to evaluate the objective function on the entire dataset. This is way too costly to accomplish.

1.6 Stochastic Gradient Descent

In deep learning, the objective (cost) function is usually the average of the loss functions for each example in the training dataset. Assuming that $f_i(\boldsymbol{x})$ is the loss function of the training data instance with m examples, an index of i, and parameter vector of $\boldsymbol{x}$, then we have the objective function:

$$f(\boldsymbol{x}) = \frac{1}{m} \sum_{i=1}^{m} f_i(\boldsymbol{x}) \tag{1.6.1}$$

The gradient of the objective function at $\boldsymbol{x}$ is computed as

$$\nabla f(\boldsymbol{x}) = \frac{1}{m} \sum_{i=1}^{m} \nabla f_i(\boldsymbol{x})$$

gradient descent would repeat:

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \alpha \nabla f(\boldsymbol{x}^k)$$

if gradient descent is used, the computing cost for each independent variable iteration is $\mathcal{O}(n)$ which grows linearly with n. Therefore, when the model training data instance is large, the cost of gradient descent for each iteration will be very high.

Stochastic gradient descent (SGD) **Robbins and Monro (1951)** is ideal in the context of large scale machine learning, it reduces computational cost at each iteration by uniform

sampling an index $i_k \in \{1, \ldots, m\}$ for data instances at random, and compute the gradient $\nabla f_{i_k}(\boldsymbol{x})$ to update $\boldsymbol{x}$:

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \alpha_k \cdot \nabla f_{i_k}\left(\boldsymbol{x}^k\right), \quad k = 1, 2, 3, \ldots \tag{1.6.2}$$

where $i_k \in \{1, \ldots, m\}$ is some chosen index at iteration k and α_k is the learning rate at iteration k. The computing cost for each iteration drops from $\mathcal{O}(n)$ of the gradient descent to the constant $\mathcal{O}(1)$. Stochastic gradient descent uses an unbiased estimate of the gradient $\nabla f(\boldsymbol{x})$ at each step, i.e,

$$\mathbb{E}[\nabla f_{i_k}(\boldsymbol{x})] = \frac{1}{m} \sum_{i=1}^{m} \nabla f_i(\boldsymbol{x}) = \nabla f(\boldsymbol{x}) \tag{1.6.3}$$

The iterate sequence is not determined uniquely by the function f, the starting point $\boldsymbol{x}^1$, and the sequence of step sizes $\{\alpha_k\}$, as it would in gradient descent. Rather, $\{\boldsymbol{x}_k\}$ is a stochastic process whose behaviour is determined by the random sequence $\{i_k\}$.

Theorem 1.6.1. Nemirovski et al. (2009) Suppose $f : \mathbb{R}^m \to \mathbb{R}$ is convex, stochastic gradient descent with diminishing learning rate $\alpha_k = 1/k$ satisfies

$$\mathbb{E}[f(\boldsymbol{x}^{(k)})] - f(\boldsymbol{x}^*) = \mathcal{O}(1/\sqrt{k})$$

When f is μ-strongly convex and L-smooth, stochastic gradient descent with diminishing learning rate $\alpha_k = \theta/k$, with constant $\theta > 1/(2\mu)$ satisfies

$$\mathbb{E}[f(\boldsymbol{x}^{(k)})] - f(\boldsymbol{x}^*) = \mathcal{O}(1/k)$$

This shows that stochastic gradient descent does not enjoy the linear convergence rate of gradient descent under strong convexity.

A popular method used in deep learning is to compute the gradient against more than one training example, called a minibatch, at each step. This method improves computational efficiency compared to stochastic gradient descent, because the code can make use of vectorization libraries rather than computing each step separately. It may also result in smoother convergence, as the gradient computed at each step is averaged over more training examples.

In minibatch stochastic gradient descent, we choose a random subset $I_k \subseteq \{1, \ldots, m\}$, $|I_k| = b \ll m$, repeat:

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \alpha_k \cdot \frac{1}{b} \sum_{i \in I_k} \nabla f_i\left(\boldsymbol{x}^k\right), \quad k = 1, 2, 3, \ldots \tag{1.6.4}$$

Again, we are approximating full gradient by an unbiased estimate

$$\mathbb{E}\left[\frac{1}{b}\sum_{i\in I_k}\nabla f_i(\boldsymbol{x})\right] = \nabla f(\boldsymbol{x})$$

Theorem 1.6.2. Dekel et al. (2012) Let $f : \mathbb{R}^m \to \mathbb{R}$ be an L-smooth convex function and assume that the stochastic gradient $\nabla f_i(\boldsymbol{x}^k)$ has σ^2-bounded variance for all $\boldsymbol{x} \in \mathbb{R}^n$, i.e.

$$\mathbb{E}\left[\ \|\nabla f(\boldsymbol{x}) - \nabla\mathbb{E}\{f(\boldsymbol{x})\}\|_2\ \right] \leq \sigma^2$$

then the convergence rate of minibatch stochastic gradient descent, with minibatch size b after k iterations is of order $\mathcal{O}(1/\sqrt{bk} + 1/k)$.

Since the total number of examples examined is bk though there is only an increase of $\sqrt{b}$ times, the convergence speed degrades with increasing minibatch size b.

1.7 Regularisation

The fundamental problem of machine learning is the conflict between optimisation and generalisation. Optimisation refers to the process of modifying a model parameters to produce the best possible outcome on the training data, whereas generalisation refers to how well the trained model performs on data it has never seen before. The goal is to achieve good generalisation, but generalisation can not be monitored; the model can only changed based on its training data.

Optimisation and generalisation are associated at the beginning of the training: the lower the loss on the training data, the lower the loss on test results. The model is said to be *underfit* as this is happening, there is still progress to be made because the network has not yet modelled all the related trends in the training data. But after a number of iterations on the training data, generalisation will stop improving and validation metrics will stop and then start degrading; the model will start to *overfit*. That is, trends which are common to training data but which are deceptive or meaningless when it comes to new data are beginning to be learned.

To prevent a model from overfitting, the best solution is to get more training data; a model trained on more data will naturally generalise better. If that is not feasible, the next best

solution is by regularisation, i.e., to modulate the amount of information the model can store or impose limits on what information the model can store. If only a small number of patterns can be memorized by a network, the optimisation process would cause it to concentrate on the most prominent patterns, which have a better chance of generalising well.

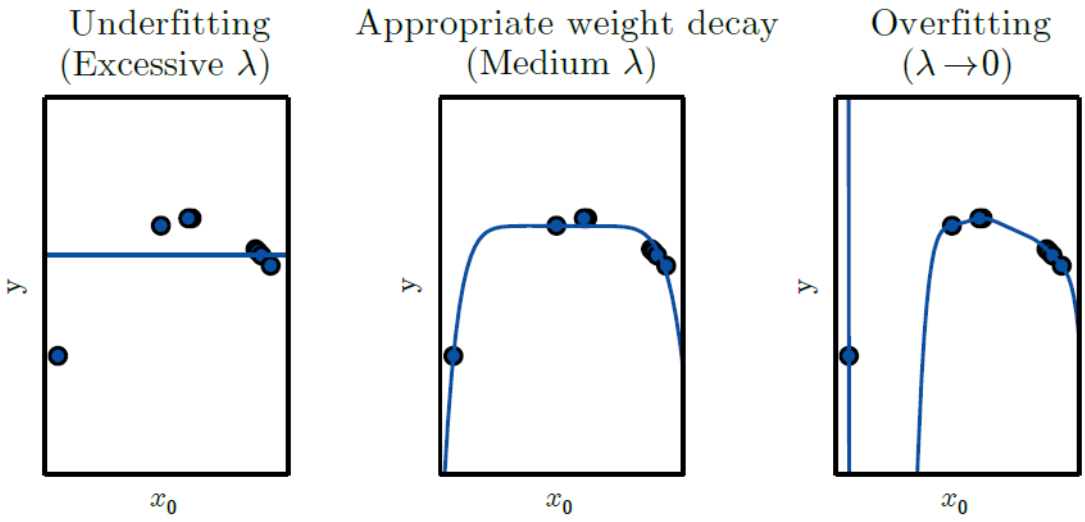


Figure 1.3: Regularisation constant effect on underfitting and overfitting in a model, **Goodfellow et al. (2016)**

Modulating the amount of information the model can store is done by reducing the number of learnable parameters in the model. A model with more parameters has more memorizing ability and can therefore easily learn a perfect dictionary-like mapping between the training samples and their targets — a mapping without generalizing strength. Imposing limits on what information the model can store is achieved by *weight regularisation*; in which complexity of a model is constrained by forcing its weights to take only small values, which makes the distribution of weight values more *regular*. Weight regularization is done by adding to the cost function of the model a cost associated with having large weights. The cost comes in two forms

- L^1 regularisation: The cost added is the L^1 norm of the weights.
- L^2 regularisation: The cost added is the square L^2 norm of the weights, it is the most widely used technique for regularising machine learning models. Hence, the new objec-

tive function for a logistic regression model with L^2 regularisation would be;

$$\mathrm{J}(\boldsymbol{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left[\left(y^{(i)}\log\left(a^{(i)}\right) + \left(1-y^{(i)}\right)\log\left(1-a^{(i)}\right)\right] + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2 \tag{1.7.1}$$

where $\lambda \geq 0$ is an hyperparameter called the *regularisation constant*, and it governs the amount of regularisation. For $\lambda = 0$, the original cost function is recovered, whereas for $\lambda > 0$ we ensure that $\boldsymbol{w}$ cannot grow too large. L^2 regularisation is also called weight decay in the context of neural networks.

Since the square L^2 norm of weights $\frac{\lambda}{2}\|\boldsymbol{w}\|_2^2$ is a strongly convex function (it has a quadratic lower bound at every chosen point), the cost function $\mathrm{J}(\boldsymbol{w}, b)$ in (1.7.1) becomes a strongly convex function (Proposition 1.1.2), hence gradient descent applied on a L^2 regularised cost function converges faster.

1.8 Decision Boundary

The decision boundary is a function that separates the two classes, $y = 0$ and $y = 1$.

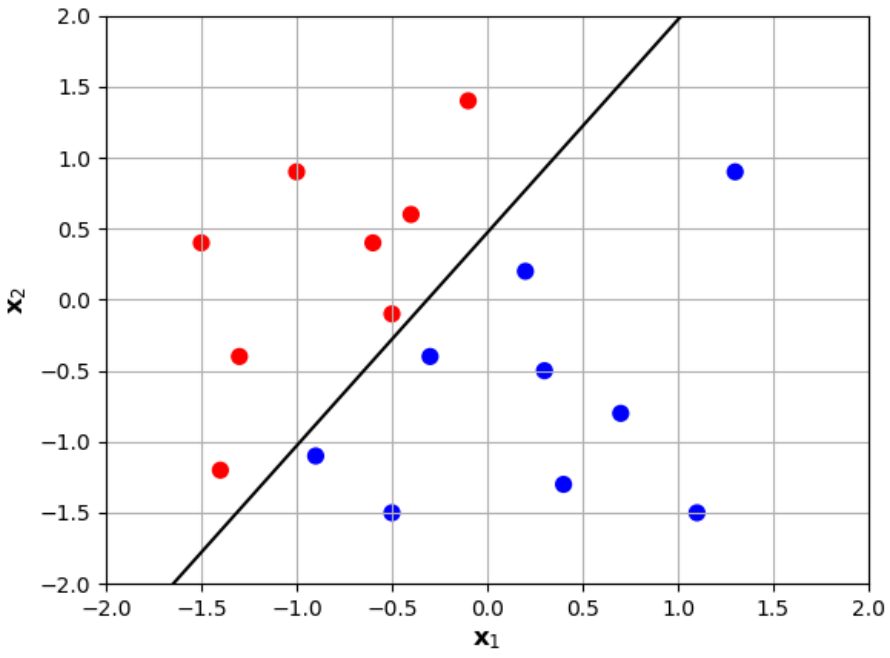


Figure 1.4: Decision boundary for two inputs $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$

When $z = 0$, then $\hat{y} = 0.5$ due to the sigmoid function's y-intercept. The decision boundary is a hyperplane of the equation $\boldsymbol{w}^T\boldsymbol{x} = 0$ as it discriminates the two different regions of classification. On the left hand side of the decision boundary where $z < 0$, the output is $a < 0.5$, thus class 0 is predicted, while class 1 is predicted on the right side of the line. If the parameters $\boldsymbol{w}$ and b are not optimal, then the separation would be inaccurate, this provides intuition into optimising the parameters for the classification task.

Chapter 2

Literature Review

2.0 Introduction

Machine learning is a powerful tool in applying modern science area for achieving the incredible success in taking place of tedious and complex manual processing. In different science area, deep learning models play different roles in significant applications. Our focus in this project are on the problems involving computer vision area, which takes most advantages from Convolutional Neural Networks (CNNs). In this chapter, we discuss briefly the history of deep learning and provide a literature survey of past works in the application of machine learning to some agricultural problems, particularly in the areas of weed management, pest control, crop disease detection, plant phenotyping, plant growth and yield prediction.

2.1 Deep Learning

2.1.1 Perceptrons

The idea of connecting units to local receptive fields on the input goes back to the perceptron in 1958 **Rosenblatt (1958)**. The perceptron was intended to be a machine, rather than a program. This machine was designed for image recognition: it had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors **Bishop (2006)**. Although the

Input Pattern	Output
00	0
10	1
101	1
111	0
010	1

Table 2.1: XOR table

perceptron initially seemed promising, the constraint that similar input patterns lead to similar outputs lead to an inability of the system to learn certain mappings from input to output, which proved that perceptrons could not be trained to recognise many classes of patterns.

Single layer perceptrons are only capable of learning linearly separable patterns. **Minsky and Papert (1969)** showed that it was impossible for these classes of network to learn an exclusive or (XOR) problem illustrated in Table (2.1) (a logical operation that outputs true only when inputs differ). They also pointed out that, if there is a layer of simple perceptron like hidden units, with which the original input pattern can be augmented, there is always a recoding (i.e., an internal representation) of the input patterns in the hidden units in which the similarity of the patterns among the hidden units can support any required mapping from the input to the output units. Thus, if we have the right connections from the input units to a large enough set of hidden units, we can always find a representation that will perform any mapping from input to output through these hidden units.

The limitations of the perceptron caused the field of neural network research to stagnate for many years, before it was recognised that a feedforward neural network with two or more layers (also called a multilayer perceptron) had greater processing power than perceptrons with one layer. Also, the Minsky and Papert text was often miscited to note that the multi-layer perceptron could not solve the XOR problem. This caused a significant decline in interest and funding of neural network research. It took ten more years until neural network research experienced a resurgence in the 1980s.

2.1.2 Back Propagation

Connectionist architectures; where large numbers of weighted connections of "neuron-like" processing elements, with weighted connections (which encodes the knowledge of the system) between elements, and with emphasis on learning internal representations rather than having them hand programmed, which were developed to result in systems that are: fast and resistant to damage, able to generalise from its inputs, and to learn efficiently for large-scale masses drew considerable attention in the early 80's because of their interesting learning abilities. Among the numerous learning algorithms that have been proposed for complex connectionist networks, Back-Propagation (BP) is probably the most widespread. Back propagation was proposed in **Rumelhart et al. (1986)** but had been developed before by several independent groups in different contexts and for different purposes, such as; **Kelley (1960)** in the context of control theory, **Bryson and Ho (1969)** in the framework of optimal control and system identification, **Werbos (1974)** proposed that it could be used for neural networks, **LeCun (1985)** in the learning of the threshold for asymmetrical network. The error back-propagation presented by **Rumelhart et al. (1986)** showed experimentally that the method can generate useful internal representations of incoming data in hidden layers of neural networks which was crucial in the learning phase of deep neural networks.

2.1.3 Convolutional Neural Networks

Hubel and Wiesel (1962) found that cells in a cat and monkey visual cortex are responsible for detecting light in receptive fields, these visual cortexes contain neurons that individually respond to small regions of the visual field. The "neocognitron" was introduced by **Fukushima (1980)**. It was inspired by the above-mentioned work of Hubel and Wiesel. The neocognitron introduced the two basic types of layers in Convolutional Neural Networks (CNNs): convolutional layers, and downsampling layers. A convolutional layer contains units whose receptive fields cover a patch of the previous layer. The weight vector (the set of adaptive parameters) of such a unit is often called a filter. Units can share filters. Downsampling layers contain units whose receptive fields cover patches of previous convolutional layers. Such a unit typically computes the average of the activations of the units in its patch. This down-

sampling helps to correctly classify objects in visual scenes even when the objects are shifted. The neocognitron is the first CNN which requires units located at multiple network positions to have shared weights.

The Time Delay Neural Network (TDNN) was introduced by **Waibel et al. (1990)** and was the first convolutional network, as it achieved shift invariance. It did so by utilizing weight sharing in combination with backpropagation training. Thus, while also using a pyramidal structure as in the neocognitron, it performed a global optimisation of the weights, instead of a local one. TDNNs are convolutional networks that share weights along the temporal dimension, they allow speech signals to be processed time-invariantly.

A system to recognize hand-written ZIP Code numbers **Denker et al. (1989)** involved convolutions in which the kernel coefficients had been laboriously hand designed. **LeCun et al. (1989)** used back-propagation to learn the convolution kernel coefficients directly from images of hand-written numbers. Learning was thus fully automatic, performed better than manual coefficient design, and was suited to a broader range of image recognition problems and image types. This approach became a foundation of modern computer vision.

LeNet-5, a pioneering 7-level convolutional network **LeCun et al. (1998)** that classifies digits, was applied by several banks to recognize hand-written numbers on cheques digitized in 32×32 pixel images. The ability to process higher resolution images requires larger and more layers of convolutional neural networks, so this technique was constrained by the availability of computing resources. This constraint lead to mostly bad performance of neural network models applied in the real world.

2.1.4 Long Short Term Memory

With conventional back propagation through time **Werbos (1988)**, error signals flowing backwards in time tend to either blow up or vanish; the temporal updates of the backpropagated error exponentially depends on the size of the weight **Hochreiter and Schmidhuber (1997)**. Long Short Term Memory (LSTM) **Hochreiter and Schmidhuber (1997)**, overcame the error back-flow problems by learning to bridge time intervals in excess of 1000 steps even in noisy input data, without loss of short time lag capabilities. This is achieved by an efficient, gradient-based algorithm. LSTMs are effective at capturing long term tempo-

ral dependencies. LSTM achieved record results in natural language text compression, and unsegmented connected handwriting recognition **Graves et al. (2009)**. As of 2016, major technology companies including Google, Apple, and Microsoft were using LSTMs as fundamental components in new products.

2.1.5 Advancements in Computing Power

Although CNNs were invented in the 1980s, their breakthrough in the 2000s required fast implementations on Graphics Processing Units (GPUs). GPUs are extremely efficient at matrix multiplication, which basically forms the core of machine learning. The strength of GPU lies in data parallelization, which means that instead of relying on a single core, as CPUs did before, a GPU can have many small cores. A single GPU can have thousands of Arithmetic Logic Units (ALUs), each performing a parallel computation to return a higher throughput faster, thus running complex computations in a short span of time. This makes GPUs highly suitable for performing complex matrix multiplications in a short span of time.

Oh and Jung (2004) showed that standard neural networks can be greatly accelerated on GPUs, their implementation was 20 times faster than an equivalent implementation on CPU. **Chellapilla et al. (2006)** described GPU-implementation of a CNN on character recognition problems, their implementation was 4 times faster than an equivalent implementation on CPU. **Ciresan et al. (2010)** showed that deep standard neural networks with many layers can be quickly trained on GPU by supervised learning through backpropagation, their network outperformed previous machine learning methods on the Modified National Institute of Standards and Technology (MNIST) database (a large database of handwritten digits) handwritten digits benchmark. **Ciregan et al. (2012)** significantly improved on the best performance in the literature for multiple image databases, including the MNIST database **LeCun et al. (1989)**, the NORB (New York University Object Recognition Benchmark) dataset (contains stereo image pairs of 50 uniform-coloured toys under 36 azimuths, 9 elevations, and 6 lighting conditions, for a total of 194,400 individual images) **LeCun et al. (2004)**, the HWDB1.0 dataset (containing $3,866$ hand-written Chinese characters) **Liu et al. (2011)**, the CIFAR-10 (Canadian Institute For Advanced Research) dataset (containing 60000 32×32 labelled RGB images) **Krizhevsky et al. (2009)**, and traffic signs (dataset of more than $50,000$

traffic sign images) **Stallkamp et al. (2011)**, using backpropagated trained deep CNNs implemented on GPUs.

The Tensor Processing Unit (TPU), an application-specific integrated circuit (ASIC) designed to accelerate artificial intelligence applications was announced by Google in 2016, it was built specifically for neural network machine learning on the TensorFlow framework. TPUs were used in the AlphaGo versus Lee Sedol series of man-machine Go games, where AlphaGo defeated human champions in Go **Silver et al. (2016)**. Third generation cloud TPUs pods can deliver up to 100 petaFLOPS (floating point operations per second) per pod. This makes TPUs perfect for both training and inferencing of deep learning models, since training the models requires millions of floating points calculations. In addition to accuracy and speed, the TPUs also offer a higher energy efficient than conventional processing chips, consuming much less watts and also reducing the heat generated, TPUs are also a lot cheaper than GPUs and they are easily available on the cloud.

2.2 Deep Learning in Agriculture

Deep learning constitutes a modern technique for highly performing image processing and data analysis with credible results. As deep learning has been applied in various domains, it has also found its place in the applied agriculture-analysis and phenotyping realm. Majority of those applications are based on typical CNN architectures to perform supervised classification to solve complex problems.

2.2.1 Plant Phenotyping and Yield Prediction

Baweja et al. (2018) developed a model by coupling deep convolutional neural networks (fast region-based convolutional network, **Girshick (2015)** and fully convolutional network, **Long et al. (2014)**) trained on a GTX 970 GPU together to measure stalk count and stalk width of Sorghum plants. The pipeline developed accurately extracts both detected object regions and dense semantic segmentation for extracting both stalk counts and stalk width. A ground robot was used to deploy a high-resolution stereo camera to capture dense image data of experimental plots of Sorghum plants in South Carolina and Mexico. Their method yielded

an R-squared correlation of 0.88 for stalk count and a mean absolute error of 2.77mm where average stalk width is 14.354mm.

Taghavi et al. (2018) proposed a CNN-LSTM framework for plant classification of various genotypes, LSTMs were also used to study the growth of the plants and their dynamic behaviours as important discriminative phenotypes for accession classification. A dataset of time series image sequences of four accessions of *Arabidopsis*, captured in similar imaging conditions, were used to train the models. The CNN-LSTM framework yielded an average accuracy of 93%.

Oliveira et al. (2018) applied recurrent LSTM layers to forecast pre-season soybean and maize yields for Brazil and USA yield. Field sensors, satellites, and unmanned aerial vehicles (UAVs) were used to derive precipitation, soil properties and seasonal climate forecasting datasets. The model had a root mean square percentage error of 14.31%, 12.85% and 15.28% for Brazil-soybean, US-soybean, and US-maize yields respectively.

Khaki et al. (2019) designed a two-step approach using unsupervised deep CNNs and regression analysis to classify corn hybrids as either tolerant or susceptible to drought stress, heat stress, and combined drought and heat stress. The dataset used was from the 2019 Syngenta crop challenge **Syngenta (2019)**, several large datasets that recorded the yield performances of $2,452$ corn hybrids planted in $1,560$ locations between 2008 and 2017. The results labelled 121 hybrids as drought tolerant, 193 as heat tolerant, and 29 as tolerant to both stresses. The proposed approach and its classification results were recognized as one of the winners of the Syngenta Crop Challenge.

Rasmussen and Moeslund (2019) adopted two CNN based deep learning-based methods (Multi-Task Network Cascades (MNC) and Region-Based Fully Convolutional Networks (R-FCN)) trained on NVIDIA Titan XP GPU for corn silage kernel processing, which is an important step in determining the quality of silage harvested from a forage harvester. The dataset contained $2,500$ annotated RGB colour images of harvested silage which were taken over three years. Average precision scores for the model where 66.9% and 71.8% for R-FCN and MNC respectively.

Zhang (2019) applied the RetinaNet deep learning model **Lin et al. (2017)** with different backbones to the problem of pod-counting and flower detection in soybean crop

production. Seven videos of soybean flower crops were captured by a ground robot under canopy level conditions in the field. 4,233 annotated images extracted from those videos was used as the dataset. The model performed with a high degree of accuracy, which the highest accuracy reaching up to 95%.

2.2.2 Weed Management and Pest Control

Chen et al. (2018) proposed a method for segmentation and counting of aphid nymphs using CNNs (U-Net). Digital images of *pakchoi* (Chinese cabbage) leaves at different stages of aphid infestation were obtained and a binary mask at the corresponding pixel level was annotated for each image manually, identifying each pixel as a aphid (white) or background (black). After segmentation, they simply counted the number of connected white components as the amount of aphid nymphs for each image. The result of automatic counting showed high accuracy (0.9563) and recall (0.9650), the correlation between aphid nymph count by the proposed method and manual counting was also high ($R^2 = 0.99$).

Sa et al. (2018) developed a crop/weed segmentation and mapping framework that processes multispectral images obtained from an unmanned aerial vehicle (UAV) using a deep neural network. They collected datasets from sugar beet fields in Eschikon, Switzerland, and Rheinbach, Germany, covering a 16,550 square meter sugar beet field, with a time interval of five months using two commercial quadrotor UAV platforms carrying multispectral cameras. Two deep neural network segmentation tools were compared; SegNet **Garcia-Garcia et al. (2017)**, and a modified version of SegNet **Badrinarayanan et al. (2017)**. The modified SegNet performed better than the baseline SegNet with the area under the curve of a precision-recall curve value of $[0.839, 0.863, 0.782]$.

Tetila et al. (2019) evaluated deep learning for the tasks of classifying and counting insect pests in soybeans. The approach consisted of segmenting an image from the plantation with the simple linear iterative clustering (SLIC) method and classifying each superpixel segment into a pest insect class using CNN-trained classification models. Three models of CNNs; Inception-Resnet-v2 **Szegedy et al. (2016)**, ResNet-50 **He et al. (2015a)**, and DenseNet-201 **Huang et al. (2017)**, with three different training strategies: 100% fine-tuning with the weights obtained from ImageNet, a complete network with the weights initialized randomly

and transfer learning with the weights obtained from ImageNet. Plantation images were collected from a soybean agricultural area of 100 ha located in the city of Dourados-MS, Brazil. A total of $1,000$ images were collected in different days and climatic conditions during the reproductive phenological stages of the soybean reproductive phase in the 2018/19 crop. DenseNet-201 trained with 100% fine-tuning of weights obtained from ImageNet gave the best result with 98.9% accuracy.

Yu et al. (2019) reported several deep CNN-based models that are exceptionally accurate at detecting weeds in bermudagrass [*Cynodon dactylon* (L.) Pers.]. Images of *Hydrocotyle* spp., *Hedyotis cormybosa*, and *Richardia scabra* in actively growing bermudagrass were taken from April to September 2018 using a digital camera at a ratio of $16:9$, with a resolution of 1920×1080 pixels. The training images were taken at multiple golf courses in different Florida cities. Images of *Poa annua* growing with various broadleaf weeds in dormant bermudagrass were taken in early February 2018 using a digital camera in different Georgia cities. The three deep CNN architectures investigated were; DetectNet **Tao et al. (2016)**, GoogLeNet, and VGGNet **Simonyan and Zisserman (2014)**. DetectNet exhibited the best performance for detection of weeds while growing in dormant bermudagrass, with F1 scores of 0.99.

2.2.3 Crop Diseases

Mohanty et al. (2016) deployed an automated image recognition system in which widespread smartphone penetration, HD cameras, and high performance processors were used for plant disease detection. This model based on an automated image recognition system and CNN based architectures (AlexNet **Krizhevsky et al. (2012)**, and GoogLeNet **Szegedy et al. (2015)**) achieved an overall accuracy of 99.35% on a held-out test data. They used the CNN to detect 26 diseases over 14 crop species. A total of $54,306$ colour images was tested. **Sladojevic et al. (2016)** also developed a plant disease recognition model based on leaf image classification using deep CNN. The images of 13 crop diseases collected for the dataset were downloaded from the Internet, searched by disease and plant name on various sources in different languages, the crops included powdery mildew, rust (apple), leaf spot (pear), and wilt, mites, downey mildew (grapevine). This model achieved an overall detection accuracy of 96.3%.

Ferentinos (2018) used five CNN-based architectures (AlexNet, AlexNetOWTBn **Krizhevsky (2014)**, GoogLeNet, Overfeat **Sermanet et al. (2013)**, and VGG **Simonyan and Zisserman (2014)**) to perform plant disease detection and diagnosis using simple leaves images of healthy and diseased plants. Training of the models was performed with the use of an open database of $87,848$ images, containing 25 different plants in a set of 58 distinct classes of (plant, disease) combinations, including healthy plants. The most successful model architecture was the VGG, it achieved a success rate of 99.53% (top-1 error of 0.47%) in the classification of $17,548$ previously unseen plant leaves images.

Chen et al. (2019) implemented LeafNet **Barré et al. (2017)**, a CNN-based architecture with different sized feature extractor filters that automatically extract the features of tea plant diseases from images. The performance of LeafNet was compared with support vector machine (SVM) and multi-layer perceptron (MLP) classifiers in the disease recognition task. The LeafNet algorithm identified tea leaf diseases most accurately, with an average classification accuracy of 90.16%. Images showing tea leaf diseases were captured in the natural environments within the Hubei province of China. A total of $3,810$ tea leaf images were used that showed symptoms for seven different diseases.

Chapter 3

Neural Networks

3.0 Introduction

Logistic regression model discussed in Chapter 1 only works effectively in cases where the lineation between class 0 and class 1 can be separated by a hyperplane. But there are situations in which the data are not well separated by a linear classifier. In this chapter, we consider a model that is capable of building decision boundaries between multiple classes that are more sophisticated than what a linear classifier can do.

The term 'neural networks' has its origins in attempts to find mathematical representations of information processing in biological systems **McCulloch and Pitts (1943)**; **Rosenblatt (1958)** - the center nervous system in particular. Smaller processing units (neurons or nodes) are connected together to form a complex network that is capable of learning and adapting.

In neural networks, we train with supervised learning, where the training set contains of inputs $\boldsymbol{x}$ and target output y. A basic neural network model can be described a series of functional transformations. First we construct linear combinations of the input variables $[\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_D]$ in the form

$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]} \tag{3.0.1}$$

the superscript [1] indicates that the corresponding parameters $\boldsymbol{W}$ (weights) and $\boldsymbol{b}$ (biases) are in the first 'layer' of the network. Each of the linear combinations is then transformed

using a differentiable, non-linear activation function $g(.)$ to give

$$\boldsymbol{a}^{[1]} = g^{[1]}(\boldsymbol{z}^{[1]}) \tag{3.0.2}$$

where $\boldsymbol{a}^{[l]}$ represents the activations at layer l. These quantities in the context of neural networks, are called *hidden units* which forms the hidden layer, The term hidden layer refers to the fact that in the training set, the true values for the nodes in the middle are not observed. The non-linear functions $g(.)$ are generally chosen to be sigmoidal functions such as the logistic sigmoid or the 'tanh' function. These values are again linearly combined to give *output unit*

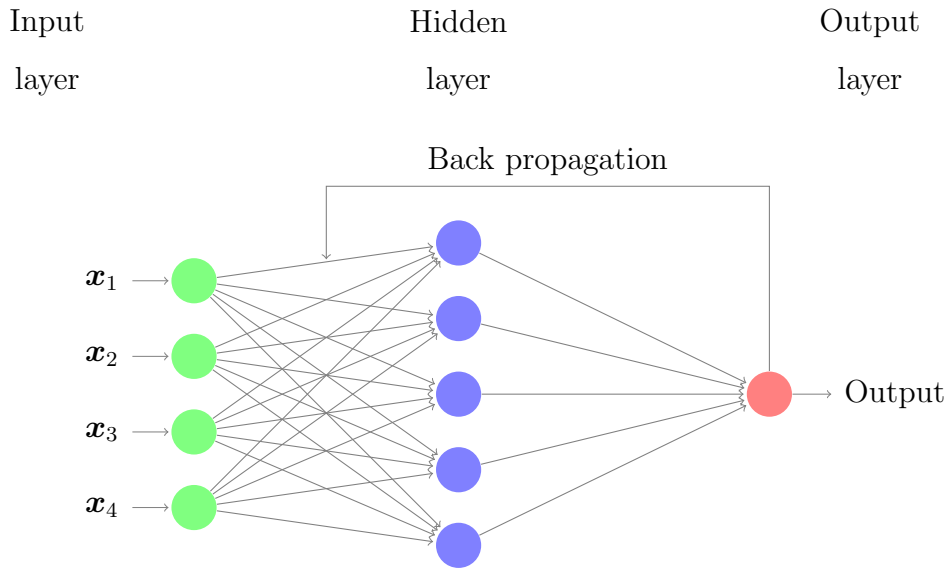


Figure 3.1: Network diagram for one hidden layer neural network

activations

$$\boldsymbol{z}^{[2]} = \boldsymbol{W}^{[2]}\boldsymbol{a}^{[1]} + \boldsymbol{b}^{[2]} \tag{3.0.3}$$

This transformation corresponds to the second layer of the network. Finally, the output unit activations are transformed using an appropriate activation function to give a set of network output $\hat{y}$. The choice of activation function is determined by the nature of the data and the assumed distribution of target variables.

Activations refers to the values that different layers of the neural network are passing to subsequent layers. So the input layer passes on the values $\boldsymbol{X}$ to the hidden layer with activation $\boldsymbol{a}^{[0]}$, hidden layer would in turn generate activation $\boldsymbol{a}^{[1]}$. First node generates $a_1^{[1]}$,

second node generates $a_2^{[1]}$, and so on, i.e.,

$$\boldsymbol{a}^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$

with parameters $\boldsymbol{w}^{[1]}$ and $\boldsymbol{b}^{[1]}$. The output layer would generate $a^{[2]} \in \mathbb{R} = \hat{y}$ with parameters $\boldsymbol{w}^{[2]}$ and $b^{[2]}$.

Let $n^{[i]}$ be the number of nodes at layer i. The dimension of the parameters $\boldsymbol{W}^{[1]}$, $\boldsymbol{b}^{[1]}$, $\boldsymbol{W}^{[2]}$ and $\boldsymbol{b}^{[2]}$ depends on $n^{[i]}$ at different layers. The dimensions are

- $\boldsymbol{W}^{[1]}$ - $(n^{[1]}, n^{[0]})$
- $\boldsymbol{b}^{[1]}$ - $(n^{[1]}, 1)$
- $\boldsymbol{W}^{[2]}$ - $(n^{[2]}, n^{[1]})$
- $\boldsymbol{b}^{[2]}$ - $(n^{[2]}, 1)$

For example, in Figure (3.1), the dimensions of the parameters would be

- $\boldsymbol{W}^{[1]}$ - $(4, 3)$
- $\boldsymbol{b}^{[1]}$ - $(4, 1)$
- $\boldsymbol{W}^{[2]}$ - $(1, 4)$
- $\boldsymbol{b}^{[2]}$ - $(1, 1)$

Remark 1.

1. This process of moving from the input to the output layer is known as *forward propagation.*

2. The number of nodes in the input layer equals to the number of input variables in the data being processed. The number of nodes in the output layer equals the number of outputs associated with each inputs.

3. If linear activation function is used in the layers of a neural network, the neural network outputs would be a linear function of the input, no matter how many layers the neural network has, since the composition of linear functions is also a linear function, hence, there would be no hidden layer in the neural network.

4. If a linear function is used in the hidden layers, and logistic sigmoid function is used in outer layer, then the model would just be a standard logistic regression.

5. In practise, the optimal number of hidden layers and nodes depends on the nature of the problem. Some problem would require using trial and error method.

6. Although multiple hidden layers allows the network to build more and more abstract representation of the input variables, it comes at a price of needing more data, otherwise the network overfits, i.e., the network will 'memorise' the data.

7. Some machine learning problem requires multiple nodes in the output layer, but our focus in this work would be problems that requires a single output in the output layer.

3.1 Activation Functions

When building a neural network, one of the choices to be made is what activation function to use in the hidden layer and output layer of the neural network. Different activation function can be used in different layer. In this section, we discuss some of the options for an activation function.

3.1.1 Logistic Sigmoid Function

The logistic sigmoid function discussed in section 1.1.7 is one choice, the function is smooth and differentiable, this means that during back-propagation, the error can be back-propagated and the weights can be accordingly updated. The sigmoid is preferred in the outer layer for a binary classification, where $y \in \{0, 1\}$, and we want $0 \leq \hat{y} \leq 1$. The function is widely used but we still have some problems that needs to addressed. The gradient of the function is pretty flat beyond the $+3$ and -3 region (as seen in Figure (3.2)), this means that once the function falls in that region the gradients become very small, which implies that network is not really learning in the region. Another problem that the sigmoid function suffers is that the values only range from 0 to 1. This means that the sigmoid function is not symmetric around the origin and the values received are all positive,and not all times would we desire the values going to the next node to be all of the same sign. This can be addressed by scaling the logistic sigmoid function to get the tanh function.

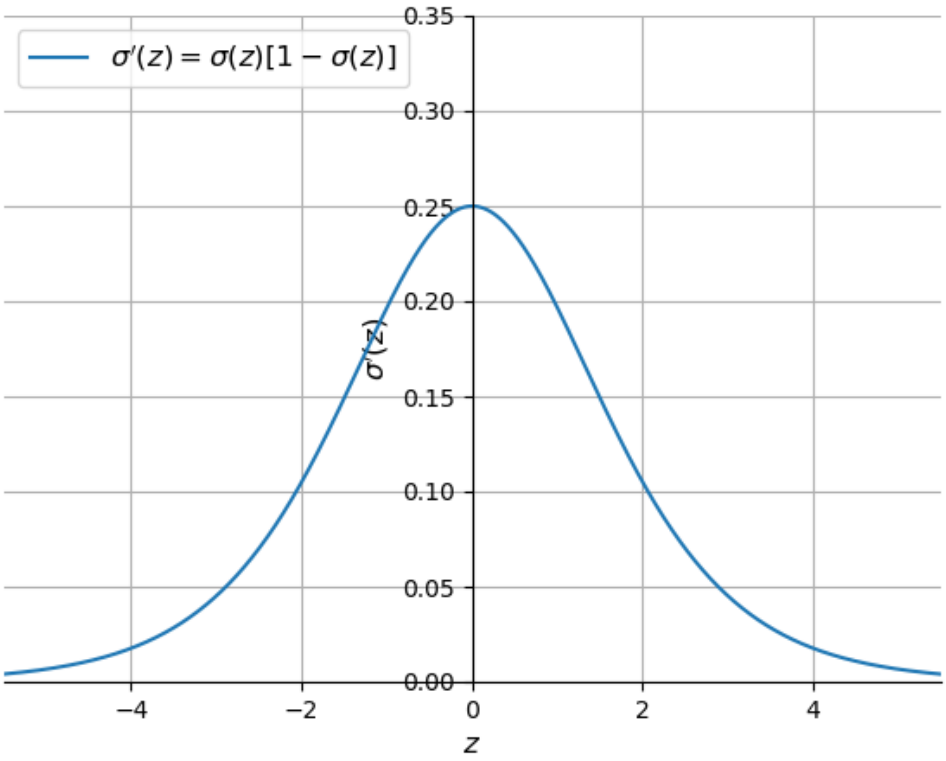


Figure 3.2: Gradient of logistic sigmoid function

3.1.2 Hyperpoblic Tangent Function

The hyperbolic tangent (tanh) function is also a sigmoid function similar to the logistic sigmoid function. It is actually just a scaled version of the logistic sigmoid function. It is given by

$$\tanh(z) = 2\sigma(2z) - 1 = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{3.1.1}$$

Its derivative is calculated thus;

$$\begin{aligned} \frac{d}{dz}\tanh(z) &= \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2} \\ &= 1 - \frac{(e^z - e^{-z})^2}{(e^z + e^{-z})^2} \\ \tanh'(z) &= 1 - \tanh^2(z) \end{aligned} \tag{3.1.2}$$

For hidden units, tanh function almost always work better than the logistic sigmoid function

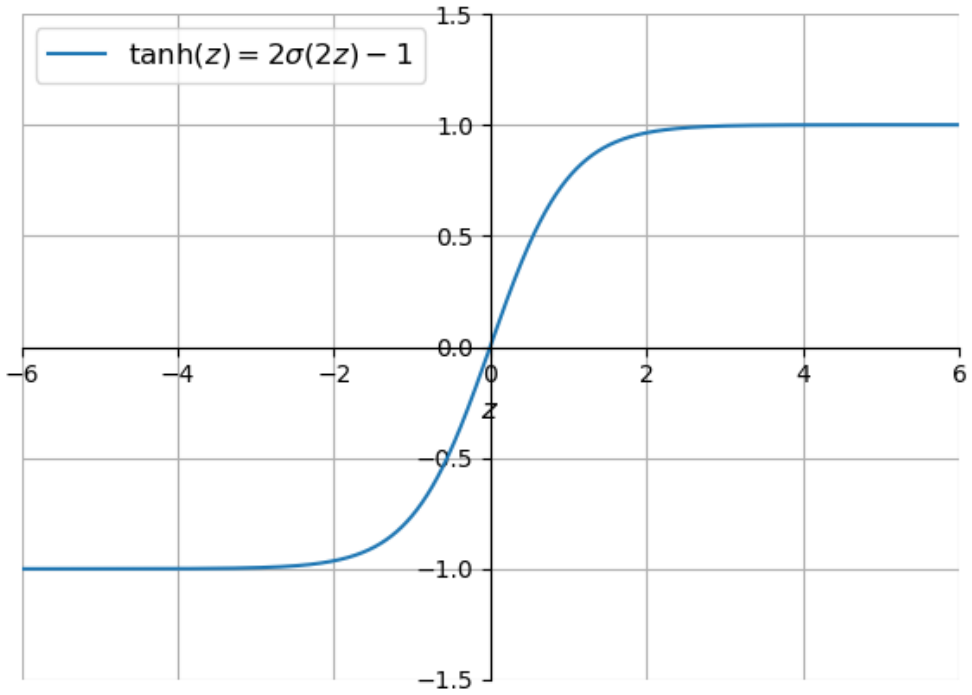


Figure 3.3: tanh function

because the values of the tanh function lies between +1 and −1 (as seen in Figure (3.3)), hence the mean of the activations that come out from the hidden layer are closer to having mean 0, rather than 0.5 (for logistic sigmoid), which makes learning the outer layer a little bit easier. It is also continuous, differentiable and non linear, so errors can be easily backpropagated.

Sigmoidal units saturate across most of their domain; they saturate to a high value when z is very positive, saturate to a low value when z is very negative, and are only strongly sensitive to their input when z is near 0 **(Goodfellow et al., 2016, p. 195)**. This feature is a general problem for the sigmoidal functions, since large values snap to 1.0 and small values snap to -1 or 0 for tanh and logistic sigmoid respectively. Further, the functions are only really sensitive to changes around their mid-point of their input, such as 0.5 for sigmoid and 0.0 for tanh. The limited sensitivity and saturation of the function happen regardless of whether the summed activation from the node provided as input contains useful information or not. Once saturated, it becomes challenging for the learning algorithm to continue to adapt the weights

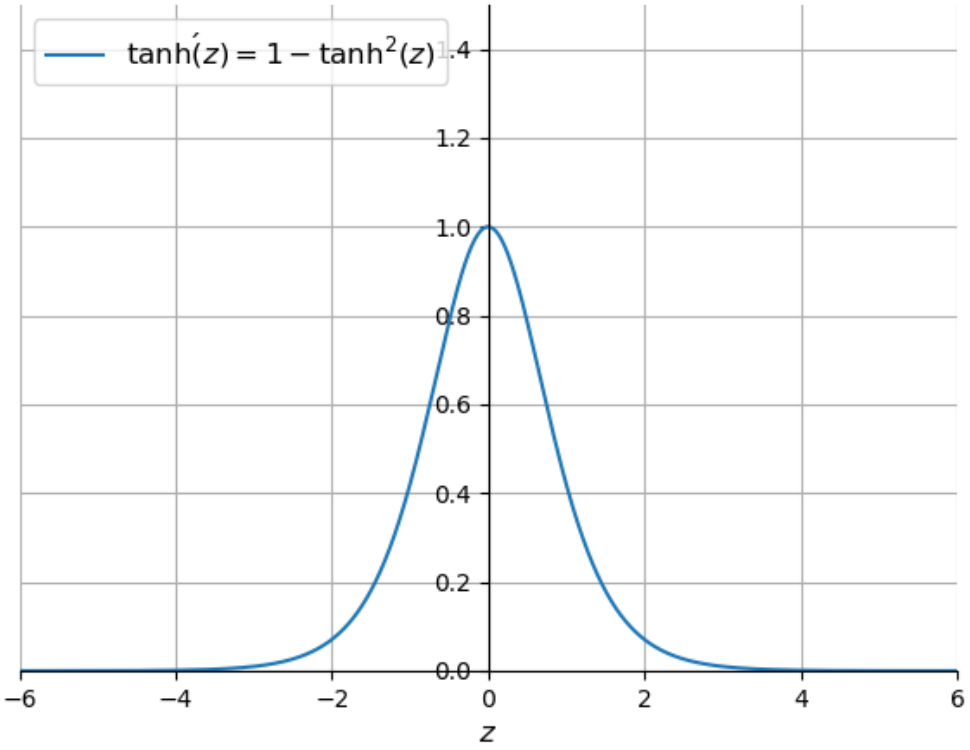


Figure 3.4: Gradient of tanh function

to improve the performance of the model.

Finally, as the capability of hardware increased through GPUs very deep neural networks using logistic sigmoid and tanh activation functions could not easily be trained. Layers deep in large networks using these non-linear activation functions fail to receive useful gradient information. Error is back propagated through the network and used to update the weights. The amount of error decreases dramatically with each additional layer through which it is propagated, given the derivative of the chosen activation function. This is called the *vanishing gradient problem* and prevents deep (multi-layered) neural networks from learning effectively. Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function **(Goodfellow et al., 2016, p. 290)**.

3.1.3 Rectified Linear Activation Function

Although the use of non-linear activation functions allows neural networks to learn complex mapping functions, they effectively prevent the learning algorithm from working with deep networks. In order to use stochastic gradient descent with backpropagation of errors to train deep neural networks, an activation function is needed that looks and acts like a linear function, but is, in fact, a non-linear function allowing complex relationships in the data to be learned. The function must also provide more sensitivity to the activation sum input and avoid easy saturation. The solution is to use the rectified linear activation function. A node or neuron that implements this activation function is referred to as a *rectified linear activation unit* (ReLU) **Hahnloser et al. (2000)**; **Jarrett et al. (2009)**; **Nair and Hinton (2010)**. Adoption of ReLU may easily be considered one of the few milestones in the deep learning revolution, e.g. the techniques that now permit the routine development of state-of-the-art deep neural networks **Krizhevsky et al. (2012)**. Currently, ReLU is the most successful and widely-used activation function. The rectified linear activation function is a simple calculation that returns the value provided as input directly, or the value 0 if the input is 0 or negative. The function is given by

$$h(z) = \max(0, z) \tag{3.1.3}$$

The function is linear for values greater than zero, meaning it has a lot of the desirable properties of a linear activation function when training a neural network using backpropagation. Yet, it is a non-linear function as negative values are always output as zero. Some advantages of using the rectified linear activation function are:

(a) The rectifier function is trivial to implement, requiring a max() function. This is unlike the sigmoidal activation functions that require the use of an exponential calculation.

(b) It does not activate all the nodes at the same time. Looking at the ReLU function (Figure (3.5)) if the input is negative it will convert it to zero and the node does not get activated. This means that at a time only a few nodes are activated making the network sparse which makes it efficient and easy for computation.

(c) An important benefit of the rectifier function is that it is capable of outputting a true

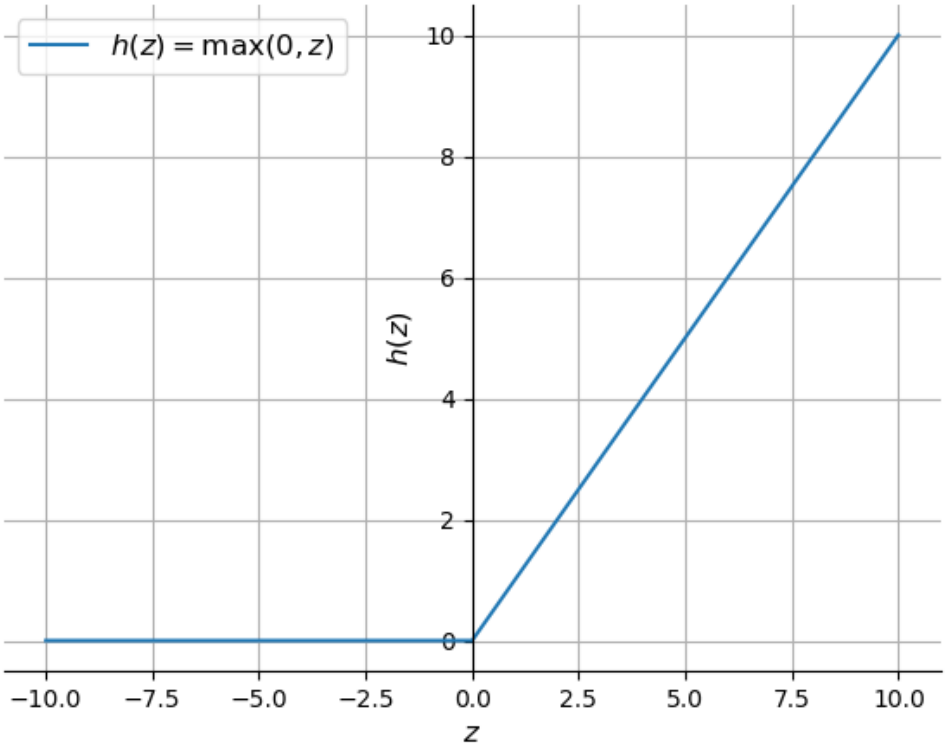


Figure 3.5: ReLU function

zero value. This is unlike the sigmoidal activation functions that learn to approximate a zero output, i.e., a value very close to zero, but not a true zero value. This means that negative inputs can output true zero values allowing the activation of hidden layers in neural networks to contain one or more true zero values. This is called a *sparse representation* and is a desirable property in representational learning as it can accelerate learning and simplify the model.

(d) The rectified linear activation function mostly looks and acts like a linear activation function. Because rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimise with gradient-based method, they also preserve many of the properties that make linear models generalise well **(Goodfellow et al., 2016, p. 175)**.

(e) The discovery and adoption of the rectified linear activation function meant that it

became possible to exploit improvements in hardware and successfully train deep multi-layered networks with a non-linear activation function using backpropagation.

Its derivative is given by:

$$h'(z) = \begin{cases} 1 & \text{if} \quad z \geq 0 \\ 0 & \text{if} \quad z < 0 \end{cases} \tag{3.1.4}$$

Because of the horizontal line in ReLU, for negative values (Figure (3.5)), the gradient can

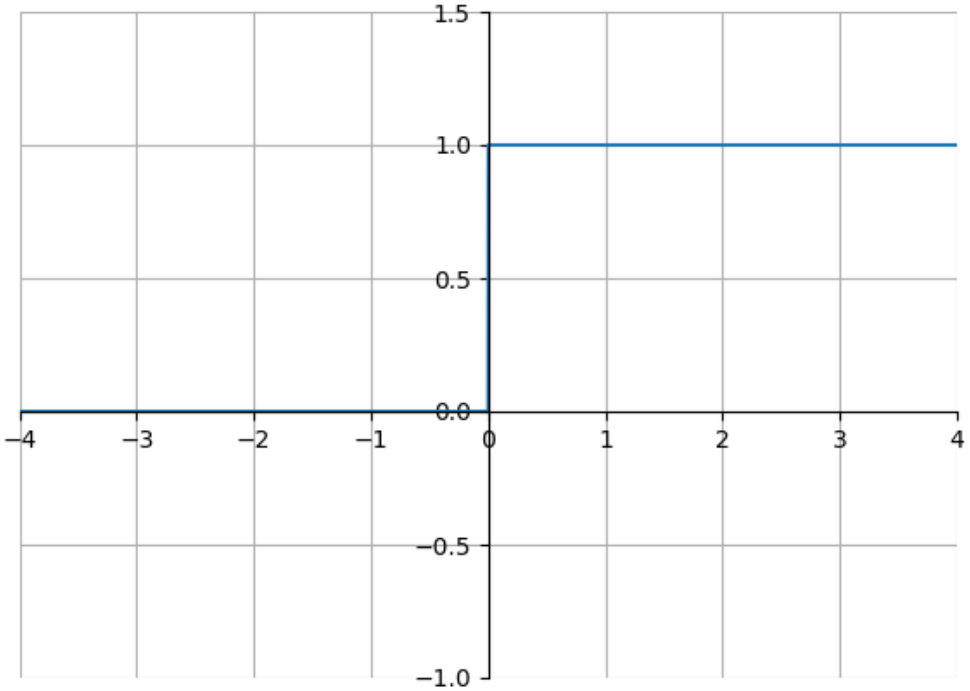


Figure 3.6: Gradient of ReLU

go towards 0. For activations in that region of ReLU, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those nodes which go into that state will stop responding to variations in error or input (simply because gradient is 0, nothing changes). This is called *dying ReLU problem*. This problem can cause several nodes to just die and not respond making a substantial part of the network passive. There

are variations in ReLU to mitigate this issue by simply making the horizontal line into non-horizontal component. We investigate some of these variations;

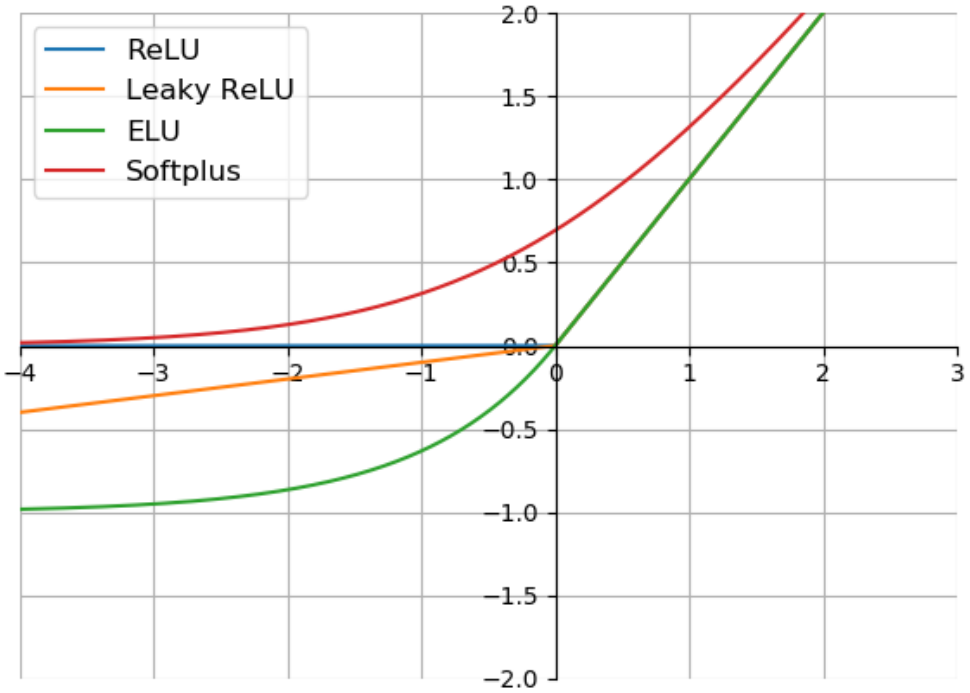


Figure 3.7: ReLU and some of its common variations

- Leaky ReLU (LReLU) **Maas et al. (2013)**:

$$f(z) = \begin{cases} z & \text{if} \quad z \geq 0 \\ \alpha z & \text{if} \quad z < 0 \end{cases} \tag{3.1.5}$$

  where $\alpha = 0.01$. LReLU enables a small amount of information to flow to the next layer when $x < 0$.

- Parametric ReLU (PReLU) **He et al. (2015b)**: The same form as LReLU but α is a learnable parameter. The network also learns the value of α for faster and more optimum convergence. Each channel has a shared α which is initialized to 0.25. PReLU function is used when the LReLU function still fails to solve the problem of dead nodes and the relevant information is not successfully passed to the next layer.

- Softplus **Nair and Hinton (2010)**:

$$f(z) = \log(1 + \exp(z)) \tag{3.1.6}$$

Softplus can be viewed as a smooth version of ReLU.

- Exponential Linear Unit (ELU) **Clevert et al. (2015)**:

$$f(z) = \begin{cases} z & \text{if} \quad z \geq 0 \\ \alpha(\exp(z) - 1) & \text{if} \quad z < 0 \end{cases} \tag{3.1.7}$$

where $\alpha = 0.1$.

- Scaled Exponential Linear Unit (SELU) **Klambauer et al. (2017)**:

$$f(z) = \lambda \begin{cases} z & \text{if} \quad z \geq 0 \\ \alpha(\exp(z) - 1) & \text{if} \quad z < 0 \end{cases} \tag{3.1.8}$$

where $\alpha \approx 1.6733$ and $\lambda \approx 1.0507$.

3.1.4 Swish Activation Function

Swish is a new, self-gated activation function discovered by researchers at Google **Ramachandran et al. (2017)**. It was shown that Swish consistently matches or outperforms ReLU on deep networks applied to a variety of challenging domains such as image classification and machine translation. On ImageNet, replacing ReLUs with Swish units improves top-1 classification accuracy by 0.9%. The Swish function is given by:

$$\varphi(z) = z \cdot \sigma(z) \tag{3.1.9}$$

where $\sigma(z)$ is the sigmoid function as defined in (1.1.18).

Like ReLU, Swish is unbounded above and bounded below. Unlike ReLU, Swish is smooth and non-monotonic. The non-monotonicity property of Swish distinguishes itself from most common activation functions. The derivative of Swish is

$$\begin{aligned} \varphi'(z) &= \sigma(z) + z \cdot \sigma(z)(1 - \sigma(z)) \\ &= \sigma(z) + z \cdot \sigma(z) - z \cdot \sigma(z)^2 \\ &= z \cdot \sigma(z) + \sigma(z)(1 - z \cdot \sigma(z)) \\ \varphi'(z) &= \varphi(z) + \sigma(z)(1 - \varphi(z)) \end{aligned} \tag{3.1.10}$$

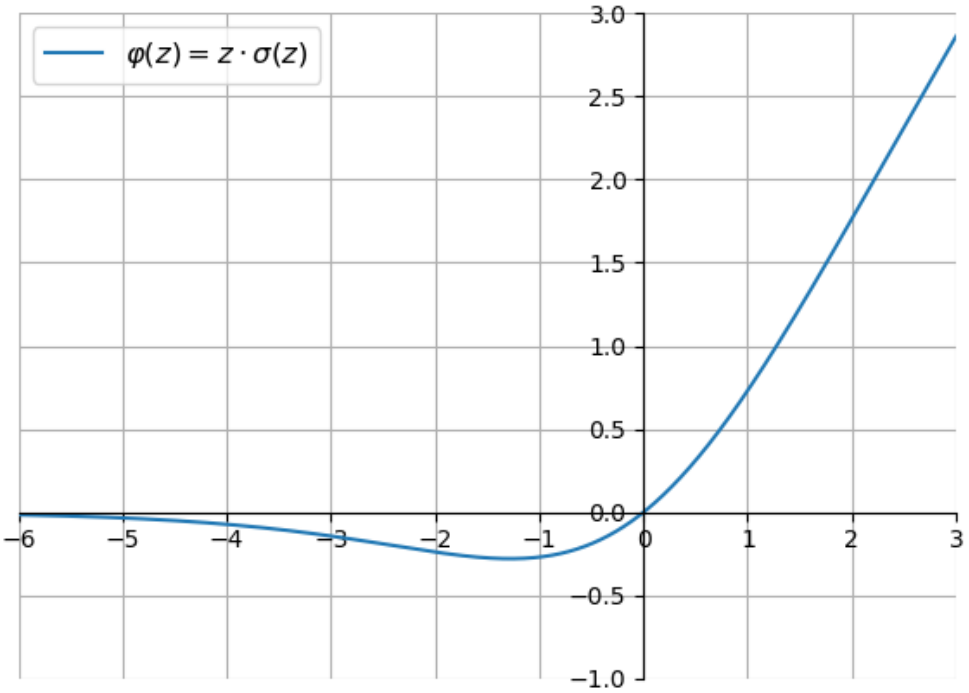


Figure 3.8: Swish function

An additional connection with ReLU can be seen if Swish is slightly reparameterized as follows:

$$\varphi(z;\beta) = 2z \cdot \sigma(\beta z)$$

If $\beta = 0$, Swish becomes the linear function $f(z) = z$. As $\beta \to \infty$, the sigmoid approaches a $0 - 1$ function, so Swish becomes like the ReLU function. This suggests that Swish can be loosely viewed as a smooth function, which non-linearly interpolates between the linear function and the ReLU function. The degree of interpolation can be controlled by the model if β is set as a trainable parameter. This variant, without the factor of 2, is called *Swish-β*.

The paper by **Ramachandran et al. (2017)** outlines the benefits of Swish;

- It is bounded below. Swish therefore benefits from sparsity similar to ReLU. Very negative weights are simply zeroed out.

- It is unbounded above. This means that for very large values, the outputs do not saturate to the maximum value (i.e., to 1 for all the nodes). The ReLU function was such a large improvement over tanh because it is unbounded above, which avoids saturation whenever $z > 0$. This property is so important that almost every recently-proposed successful activation function is unbounded above. Swish shares this property, with its positive side approaching the linear function as the input becomes more positive.

- The fact that Swish is a smooth curve means that its output landscape will be smooth. This provides benefits when optimizing the model in terms of convergence towards the minimum loss.

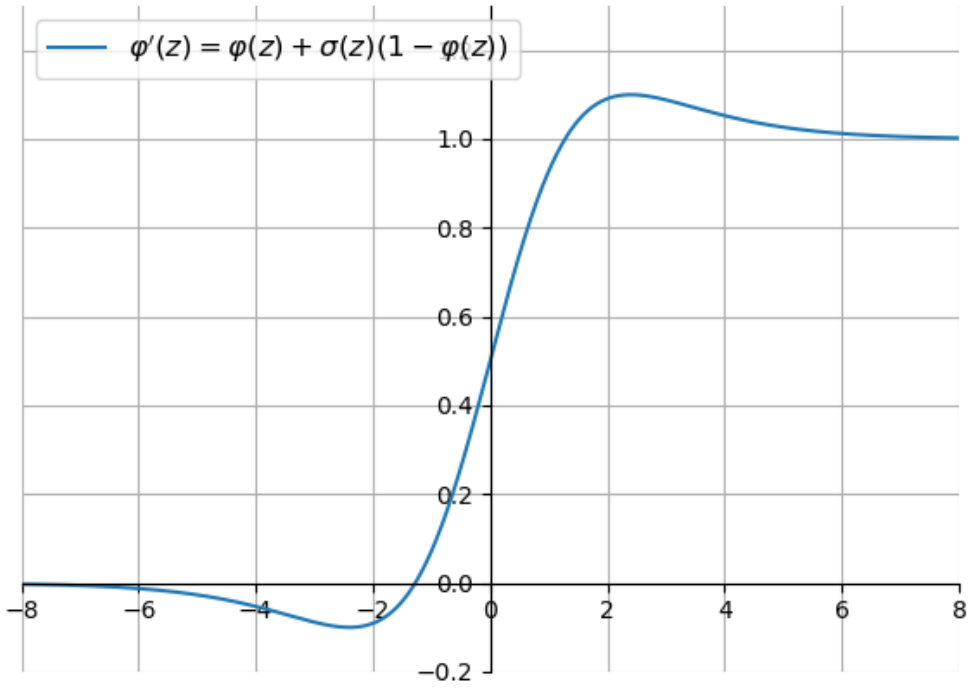


Figure 3.9: Gradient of swish function

- Small negative values are zeroed out in ReLU (since $h(z) = 0$ for $z < 0$). However, those negative values may still be relevant for capturing patterns underlying the data. Swish nonetheless differs from ReLU because it produces negative outputs for small negative

inputs due to its non-monotonicity. The non-monotonicity of Swish improves gradient flow, which is important since it avoids dead nodes.

3.1.5 Softmax Function

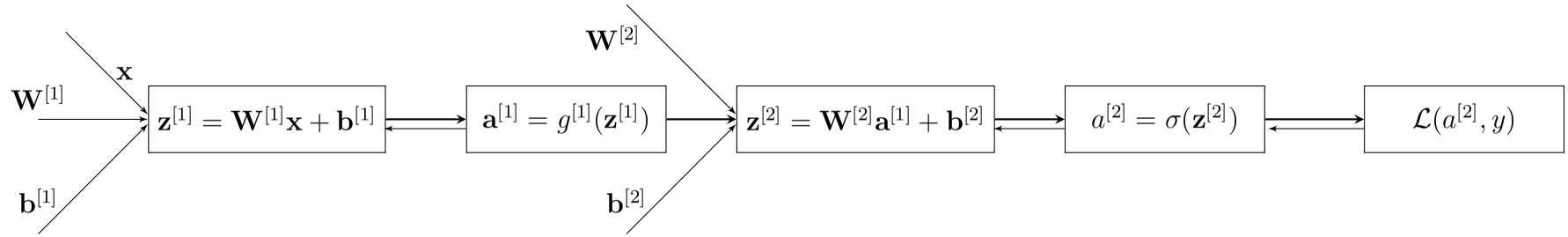
The softmax function is also a type of sigmoid function but is handy when we are trying to handle classification problems. The sigmoid function as we saw earlier was able to handle just two classes. The softmax function can handle multiple classes, it would squeeze the outputs for each class between 0 and 1 and would also divide by the sum of the outputs. This essentially gives the probability of the input being in a particular class. The standard (unit) softmax function $\sigma : \mathbb{R}^K \to \mathbb{R}^K$ is defined by the formula:

$$\sigma(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, \ldots, K \text{ and } \boldsymbol{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K \tag{3.1.11}$$

Prior to applying softmax, some vector components could be negative, or greater than one; and might not sum to 1; but after applying softmax, each component will be in the interval $(0, 1)$, and the components will add up to 1, so that they can be interpreted as probabilities. Furthermore, the larger input components will correspond to larger probabilities. Typically, Softmax is used only for the output layer, for neural networks that need to classify inputs into multiple categories. It is frequently appended to the last layer of an image classification network such as VGG16 used in ImageNet competitions.

3.2 Backpropagation

In order to apply the stochastic gradient descent algorithm to optimise the parameters of a neural network, we need to calculate the gradient of an objective function with respect to the parameters of the network. Backpropagation refers to the method of calculating the gradient of neural network parameters using the calculus chain rule. The core insight is that the objective function derivative with respect to a module's input can be computed by working backwards from the gradient with respect to that module's output (or the corresponding module's input) (Figure 3.10). The backpropagation equation can be repeatedly applied to propagate gradients through all modules, starting from the output at the top (where the network produces its



$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}} = \frac{\partial \mathbf{z}^{[2]}}{\partial \mathbf{a}^{[1]}} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[2]}} \cdot \frac{\partial \mathbf{a}^{[1]}}{\partial \mathbf{z}^{[1]}}$$
$$= \mathbf{W}^{[2]^T} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[2]}} * g^{[1]'}(\mathbf{z}^{[1]})$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[1]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}} \cdot \frac{\partial \mathbf{z}^{[1]}}{\partial \mathbf{W}^{[1]}}$$
$$= \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}} \cdot \mathbf{x}^T$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[1]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}} \cdot \frac{\partial \mathbf{z}^{[1]}}{\partial \mathbf{b}^{[1]}}$$
$$= \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}}$$

.

$$\frac{d\mathbf{a}^{[1]}}{d\mathbf{z}^{[1]}} = g^{[1]'}(\mathbf{z}^{[1]})$$

.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[2]}} = \frac{\partial \mathcal{L}}{\partial a^{[2]}} \cdot \frac{da^{[2]}}{d\mathbf{z}^{[2]}}$$
$$= a^{[2]} - y$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[2]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[2]}} \cdot \frac{\partial \mathbf{z}^{[2]}}{\partial \mathbf{W}^{[2]}}$$
$$= \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[2]}} \cdot \mathbf{a}^{[1]}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[2]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[2]}} \cdot \frac{\partial \mathbf{z}^{[2]}}{\partial \mathbf{b}^{[2]}}$$
$$= \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[2]}}$$

Figure 3.10: Forward and backward propagation in a neural network layer

prediction) all the way down (where the external input is fed). Once these gradients are determined, the computation of the gradients with respect to the weights of each element is straightforward. For a one hidden layer neural network to solve a binary classification problem, with parameters

- $\boldsymbol{W}^{[1]}$ with shape $(n^{[1]}, n^{[0]})$
- $\boldsymbol{b}^{[1]}$ with shape $(n^{[1]}, 1)$
- $\boldsymbol{W}^{[2]}$ with shape $(n^{[2]}, n^{[1]})$
- $\boldsymbol{b}^{[2]}$ with shape $(n^{[2]}, 1)$

with cost function

$$J(\boldsymbol{W}^{[1]}, \boldsymbol{b}^{[1]}, \boldsymbol{W}^{[2]}, \boldsymbol{b}^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$$

where $\mathcal{L}(\hat{y}, y)$ is the loss function defined in equation (1.4.4).

Figure (3.10) shows the network gradients for a single training example. To train multiple training examples, we vectorise across different training examples;

$$\boldsymbol{Z}^{[1]} = \begin{bmatrix} | & | & & | \\ \boldsymbol{z}^{[1](1)} & \boldsymbol{z}^{[1](2)} & \dots & \boldsymbol{z}^{[1](m)} \\ | & | & & | \end{bmatrix} \tag{3.2.1}$$

Similarly,

$$\boldsymbol{Z}^{[2]} = \begin{bmatrix} | & | & & | \\ \boldsymbol{z}^{[2](1)} & \boldsymbol{z}^{[2](2)} & \dots & \boldsymbol{z}^{[2](m)} \\ | & | & & | \end{bmatrix} \tag{3.2.2}$$

Hence,

$$\boldsymbol{Z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{X} + \boldsymbol{b}^{[1]}; \qquad \boldsymbol{A}^{[1]} = g^{[1]}(\boldsymbol{Z}^{[1]}); \qquad \boldsymbol{Z}^{[2]} = \boldsymbol{W}^{[2]}\boldsymbol{A}^{[1]} + \boldsymbol{b}^{[2]}; \qquad \boldsymbol{A}^{[2]} = g^{[2]}(\boldsymbol{Z}^{[2]})$$

The gradients would be;

$$\frac{\partial J}{\partial \boldsymbol{Z}^{[2]}} = \boldsymbol{A}^{[2]} - \boldsymbol{y} \qquad \frac{\partial J}{\partial \boldsymbol{W}^{[2]}} = \frac{1}{m} \frac{\partial J}{\partial \boldsymbol{Z}^{[2]}} \boldsymbol{A}^{[1]^T}$$

$$\frac{\partial J}{\partial \boldsymbol{b}^{[2]}} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{A}^{[2](i)} - \boldsymbol{y}^{[i]} \frac{\partial J}{\partial \boldsymbol{Z}^{[2](i)}} \qquad \frac{\partial J}{\partial \boldsymbol{Z}^{[1]}} = \boldsymbol{W}^{[2]^T} * g^{[1]'}(\boldsymbol{Z}^{[1]})$$

$$\frac{\partial J}{\partial \boldsymbol{W}^{[1]}} = \frac{1}{m} \frac{\partial J}{\partial \boldsymbol{Z}^{[1]}} \boldsymbol{X}^T \qquad \frac{\partial J}{\partial \boldsymbol{b}^{[1]}} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial J}{\partial \boldsymbol{Z}^{[1](i)}}$$

where $*$ is the element-wise product of two matrices. The above operations matches the dimensions of the matrices.

In general, for an L layer neural network, $L > 0$, which has $L-1$ hidden layers and 1 input and output layer each. The parameters (weights and biases) for layer $l = 1, 2, \ldots, L$ are represented as

- $\boldsymbol{W}^{[l]}$ with shape $(n^{[l]}, n^{[l-1]})$ - $\boldsymbol{b}^{[l]}$ with shape $(n^{[l]}, 1)$

where $n^{[l]}$ is the number of nodes in layer l, and $n^{[0]}$ is the number of nodes in the input layer. Hence, forward propagation would be;

$$\boldsymbol{Z}^{[l]} = \boldsymbol{W}^{[1]}\boldsymbol{A}^{[l-1]} + \boldsymbol{b}^{[l]}$$
$$\boldsymbol{A}^{[l]} = g^{[l]}(\boldsymbol{Z}^{[l]})$$

with $\boldsymbol{A}^{[0]} = \boldsymbol{X}$, and the backpropagation gradients are calculated thus;

$$\frac{\partial J}{\partial \boldsymbol{Z}^{[l]}} = \frac{\partial J}{\partial \boldsymbol{A}^{[l]}} * g^{[l]'}(\boldsymbol{Z}^{[l]})$$
$$\frac{\partial J}{\partial \boldsymbol{W}^{[l]}} = \frac{1}{m}\frac{\partial J}{\partial \boldsymbol{Z}^{[l]}}\boldsymbol{A}^{[l-1]^T}$$
$$\frac{\partial J}{\partial \boldsymbol{b}^{[l]}} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial J}{\partial \boldsymbol{Z}^{[l](i)}}$$
$$\frac{\partial J}{\partial \boldsymbol{A}^{[l-1]}} = \boldsymbol{W}^{[l]^T}\frac{\partial J}{\partial \boldsymbol{Z}^{[1]}}$$

3.2.1 Random Initialisation

In neural networks, the weights are initialised with random number and the biases with zeros. If all the weights are initialized with 0, the derivative with respect to cost function is the same for every $\boldsymbol{W}$ in the first layer of the network, thus all weights have the same value in subsequent iterations. This makes hidden units symmetric and continues for all the k iterations i.e. setting weights to 0 makes the model linear. In Python 3.7, `numpy.random.randn(n,m)` generates a $n \times m$ array of samples from the standard normal distribution, while `numpy.zeros((n,m))` generates a $n \times m$ array of zeros. To prevent the vanishing gradient problem, which makes learning takes a lot time, the initialised weight are multiplied by a small number (say 0.01), so as to make the values going to the activation activation not too highly positive or highly

negative, i.e.,

$$\boldsymbol{W}^{[l]} = \texttt{numpy.random.randn}(n^{[l]}, n^{[l-1]}) * 0.01$$

Sometimes there can be better constants depending on the activation function used. **He et al. (2015b)** proposed activation aware initialization of weights (called He initialisation) used for ReLU activation that improved accuracy results on ImageNet classification task, the weights are initialised thus;

$$\boldsymbol{W}^{[l]} = \texttt{numpy.random.randn}(n^{[l]}, n^{[l-1]}) * \sqrt{\frac{2}{n^{[l-1]}}}$$

Xavier initialisation is similar to He initialisation, it is used for tanh activation, and the weights are initialised as;

$$\boldsymbol{W}^{[l]} = \texttt{numpy.random.randn}(n^{[l]}, n^{[l-1]}) * \sqrt{\frac{1}{n^{[l-1]}}}$$

3.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are similar to neural networks in that they consist of neurons which optimize themselves through learning, each neuron will still receive an input and perform an operation such as a scalar product followed by a non-linear function. They are designed to process data that are in form of multiple arrays, such as a color image consisting of three 2D arrays containing pixel intensities in three color channels; red, green and blue (RGB). CNN-based network architectures now dominate the field of computer vision and are pivotal in the rapid advancement of computer vision applications.

One of the challenges of computer vision problems is that inputs can get really big. For example, an input image of $1,000 \times 1,000$ pixels, the dimension of the input features would be $1,000 \times 1,000 \times 3$ for its RGB channels. Hence, for a fully connected neural network with $1,000$ hidden units in the first layer, dimension of $\boldsymbol{W}^{[1]}$ is $(1,000, 3 \text{ million})$. Getting enough data to train 3 billion parameters without overfitting the network would be difficult, and also the computation and memory requirements needed to train the network is infeasible. Also, images with large pixel density would generally give more accurate results in computer vision applications. Our focus in this project would be on the applicability of CNNs in image data.

3.3.1 Convolution Layer

In a convolutional layer, an input array and a convolution kernel array are combined to produce an array through a cross-correlation operation, a scalar bias is added to the array to produce an output, followed by a non-linear activation function (mostly ReLU) being applied to the output array.

The cross-correlation operation is similar to the convolution between two functions $f, g : \mathbb{R}^n \to \mathbb{R}$, which is given by:

$$[f \circledast g](x) = \begin{cases} \displaystyle\int_{\mathbb{R}^n} f(z)g(x-z)\,\mathrm{d}z & \text{for continous variables} \\ \\ \displaystyle\sum_a f(a)g(x-a) & \text{for discrete variables} \end{cases} \tag{3.3.1}$$

That is, convolution is the measure of the overlap between f and g when one of the functions is shifted by x and flipped. For a two-dimensional input image $\boldsymbol{I}$ and a two-dimensional kernel $\boldsymbol{K}$, convolution of $\boldsymbol{K}$ over $\boldsymbol{I}$ is given by:

$$(\boldsymbol{I} \circledast \boldsymbol{K})(i,j) = \sum_m \sum_n \boldsymbol{I}(m,n)\boldsymbol{K}(i-m, j-n) \tag{3.3.2}$$

Cross-correlation operation $\star$, is the same as convolution but without flipping the kernel:

$$(\boldsymbol{I} \star \boldsymbol{K})(i,j) = \sum_m \sum_n \boldsymbol{I}(i+m, j+n)\boldsymbol{K}(m,n) \tag{3.3.3}$$

In a two-dimensional cross-correlation operation, a convolutional kernel (or filter) is positioned at the at the top-left corner of the input array and we slide it across the input array, both from left to right and top to bottom. When the kernel slides to a certain position, the input subarray contained in that position and the kernel array are multiplied elementwise and the resulting array is summed up yielding a single scalar value. This result if precisely the value of the output array at the corresponding location. Since the kernel has a width greater than one, and we can only compute the cross-correlation for locations where the kernel fits wholly within the image, the output size is given by the input size $H \times W$ minus the size of the convolutional kernel $h \times w$ via $(H - h + 1) \times (W - w + 1)$, since we need enough space to shift the convolutional kernel across the image. In practice, the height and width of the input

image and convolution kernel are usually the same, hence, the height and width of the output would be of the same size $(H - h + 1)$.

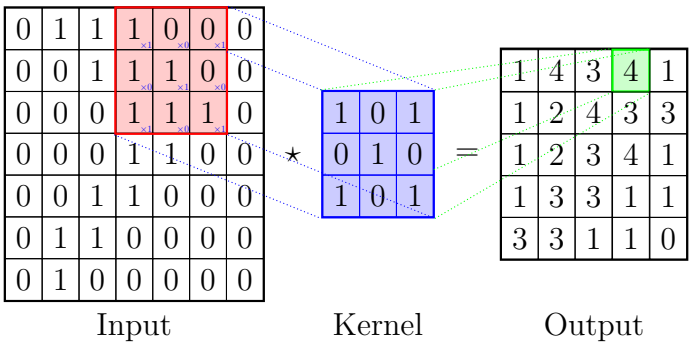
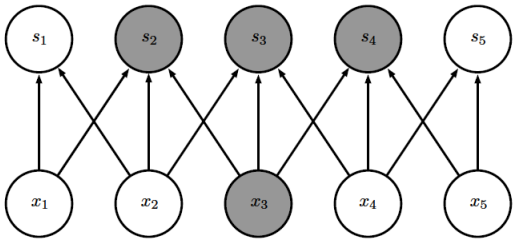
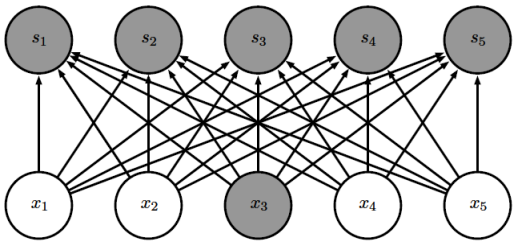


Figure 3.11: Two-dimensional cross-correlation operation. The shaded portions are the input, kernel array and fourth output elements, used in its computation: $1 \times 0 + 0 \times 0 + 0 \times 1 + 1 \times 0 + 1 \times 1 + 0 \times 0 + 1 \times 1 + 1 \times 0 + 1 \times 1 = 4$.

With matrix multiplications in traditional neural networks, every output unit depends on every input unit, convolutional networks, however, typically have sparse interactions which is



(a) When $\boldsymbol{s}$ is formed by convolution with a kernel of width 3, only three outputs are affected by $\boldsymbol{x}$



(b) When $\boldsymbol{s}$ is formed by matrix multiplication, connectivity is no longer sparse, so all the outputs are affected by $\boldsymbol{x}_3$

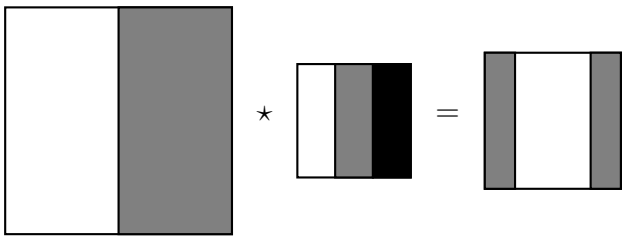
Figure 3.12: Representation of sparse and full connectivity, **Goodfellow et al. (2016)**

accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but small, meaningful features such as edges can be detected with kernels that occupy only tens or hundreds of pixels. This means that fewer parameters are stored, which both reduces the memory requirements of the model and improves its efficiency. These improvements in efficiency are usually quite large. Matrix multiplication used in traditional neural networks (Figure 3.12b) requires $m \times n$ parameters for m inputs and n outputs, which corresponds to $\mathcal{O}(m \times n)$ runtime (per example). Convolution networks limits the number of connections each output may have to k (Figure 3.12a), then the sparsely connected approach requires only $k \times n$ parameters and $\mathcal{O}(k \times n)$ runtime, where k is several orders of magnitude smaller than m.

Convolution layers are also useful for local feature detection (such as edge detection by finding the locations of pixel change in the input), since a feature that is useful in one part of the image is probably useful in another part of the image. The lighter region in the middle of

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} \star \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix}$$

(a) Cross-correlating a vertical edge filter on an input image



(b) Picture plots where bright halves gives brighter pixel intensity values, and darker halves gives darker pixel intensity values

Figure 3.13: Vertical edge detection

the output from Figure 3.13b corresponds to the vertical edge filter (which consists of a 3×3 region where pixel intensity values are relatively bright on the left part and relatively dark on

the right part) having detected a strong vertical edge down the middle of the input image, the light region shows a light to dark transition in the input image, a dark region would show a dark to light transition in the input image. Larger input images, such as $1,000 \times 1,000$ pixel image would give better edge detection results. Different filter values have been used for edge detection in computer vision literature, such as the Sobel filter **Sobel and Feldman (1968)**, and the Scharr filter **Jähne et al. (1999)**; with deep learning, the values of the edge filter $w_1, w_2, \ldots, w_9$ are treated as parameters which are learnt using back propagation, this enables the convolution layer to detect edges in different orientations. When training CNN models based on convolutional layers, the kernels are randomly initialized, just as we would with a fully-connected layer.

3.3.2 Padding and Stride

In the previous example of Figure 3.11, our input had both a height and width of 7 and our convolution kernel had both a height and width of 3, yielding an output representation with dimension 5×5, which follows the generalisation from Section 3.3.1. Hence, the output shape of the convolutional layer is determined by the shape of the input and the shape of the convolution kernel.

Since kernels generally have width and height greater than 1, after applying many successive convolutions, we tend to wind up with outputs that are considerably smaller than our input. If we start with a 240×240 pixel image, 10 layers of 5×5 convolutions reduce the image to 200×200 pixels, slicing 30% off the image and with it obliterating all useful details about the borders of the original image. Padding is the most used tool for handling this issue. In other examples, if we find the original input resolution to be unhandy, we may want to significantly decrease the dimensionality, strided convolutions in these instances are a common technique that can aid.

One tricky issue when applying convolutional layers, as mentioned above, is that we tend to lose pixels on the boundary of our image. Since we usually use small kernels, we might only lose a few pixels for every given convolution, but this can add up as we apply several

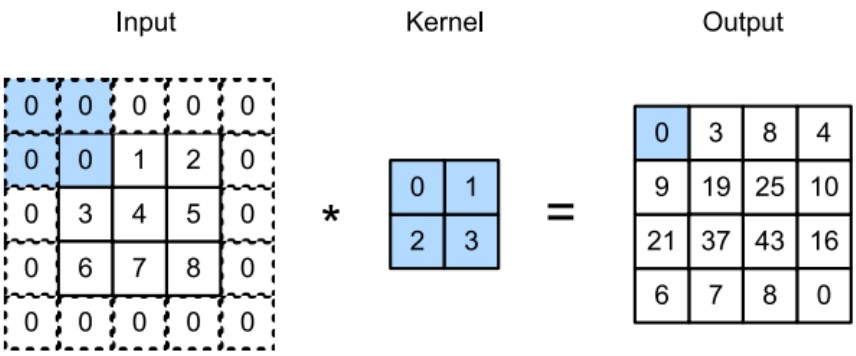


Figure 3.14: Two-dimensional cross-correlation with padding, **Zhang et al. (2020)**

successive convolutional layers. One easy solution to this problem is to add additional filler pixels around the boundary of our input image, thereby increasing the image's effective size. Typically, we set the values of the extra pixels to zero. In Figure 3.14, we pad a 3×3 input, increasing its size to 5×5. The corresponding output then increases to a 4×4 matrix. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$. In general, if we add a total of p number of padding around the input, the height and width of the output will be

$$(H - w + 2p + 1) \tag{3.3.4}$$

In many cases, we will want to set $p = h - 1$ to give the input and output the same height and width. This will make it easier to predict each layer's output shape while building the network. CNNs commonly use convolution kernels with odd height and width values, choosing odd kernel sizes has the benefit that we can preserve the spatial dimensionality while padding with the same number of rows on top and bottom, and the same number of columns on left and right.

When computing the cross-correlation, we begin with the convolution window at the top-left corner of the input tensor, and then slide it both down and to the right over all locations, defaulting to sliding one element at a time in previous examples. However, often we shift our window more than one element at a time, skipping the intermediate positions, either for computational effectiveness or because we want to downsample. We refer to the number of rows and columns traversed per slide as the stride. So far, we have used strides of 1, both

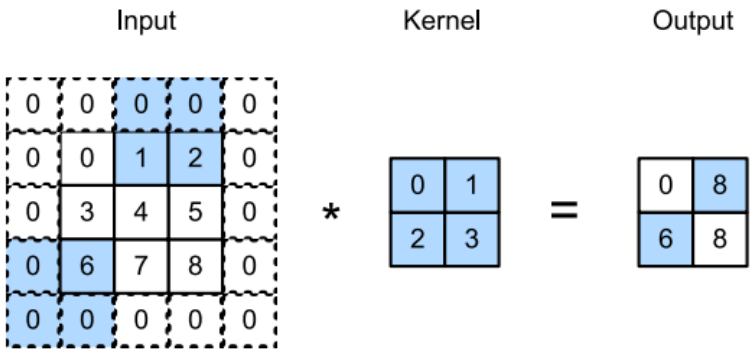


Figure 3.15: Cross-correlation with strides of 3 and 2 for height and width respectively, **Zhang et al. (2020)**

for height and width. Sometimes, we may want to use a larger stride, Figure 3.15 shows a two-dimensional cross-correlation operation with a stride of 3 vertically and 2 horizontally. The shaded portions are the output elements as well as the input and kernel tensor elements used for the output computation: $0\times0+0\times1+1\times2+2\times3=8, 0\times0+6\times1+0\times2+0\times3=6$. The convolution window slides two columns to the right when the second element of the first row is outputted. There is no output when the convolution window continues to slide two columns to the right on the input, since the input variable does not fill the window (unless we add another column of padding). In practice, we apply the same amount of stride vertically and horizontally. Hence, given an input image with size $H \times H$, a convolution kernel with size $h \times h$, applied at stride s with pad p, the output would be of height and width

$$\left\lfloor \frac{H + 2p - h}{s} + 1 \right\rfloor \tag{3.3.5}$$

3.3.3 Multiple Input and Multiple Output Channels

Before now, we have thought of the inputs, convolution kernels, and outputs each as two-dimensional tensors. But, color images have the standard RGB channels to indicate the amount of red, green and blue. Hence our our inputs and hidden representations both become three dimensional tensors when we consider the channels. For example, each RGB input image has shape $H \times H \times 3$, where the last axis with size 3 is known as the channel dimension. In this section, we will take a deeper look at convolution kernels with multiple input and multiple

output channels.

For input data containing multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-correlation with the input data. Assuming that the number of channels for the input data is c_i, the number of input channels of the convolution kernel also needs to be c_i. When $c_i = 1$, the convolution kernel is just a two-dimensional tensor of shape $h \times h$. However, when $c_i > 1$, we need a kernel that contains a tensor of shape $h \times h$ for every input channel. Thus, we concatenate these c_i tensors together to yields a convolution kernel of shape $h \times h \times c_i$. Since the input and convolution kernel each have c_i channels, we can perform a cross-correlation operation on the two-dimensional tensor of the input and the two-dimensional tensor of the convolution kernel for each channel by summing over the channels to yield a two-dimensional tensor.

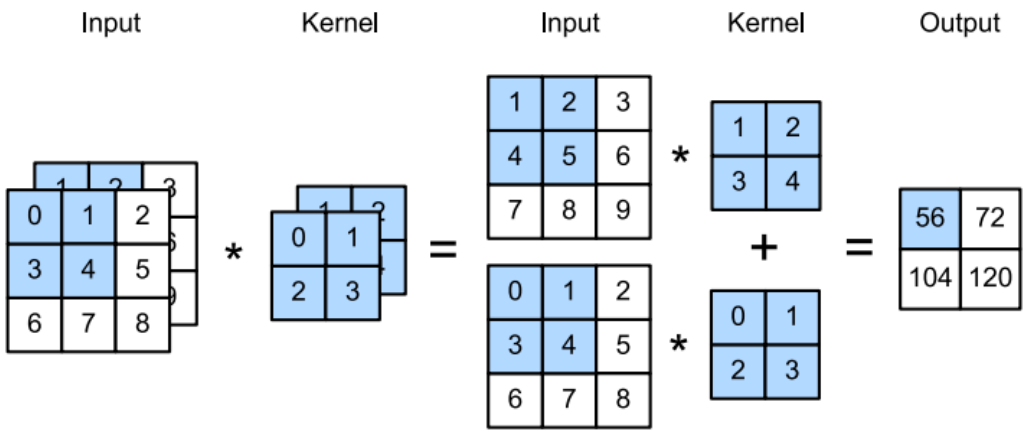


Figure 3.16: Cross-correlation computation with 2 input channels, **Zhang et al. (2020)**

Figure (3.16) shows an example of a two-dimensional cross-correlation with two input channels. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $(1\times1+2\times2+4\times3+5\times4)+(0\times0+1\times1+3\times2+4\times3) = 56$.

We have always ended up with one output channel regardless of the number of input channels. However, it is essential to have multiple channels, so that each channel provides a set of learned features to the subsequent layer. Some channels could become skilled in identifying edges at lower layers that are closer to inputs, while others could identify textures. Also, in most popular neural network architectures, channel dimension is usually increased

as the network gets deeper typically downsampling to trade off spatial resolution for greater channel depth. Let by c_i and c_0 be the number of input and output channels, respectively, with kernel width of shape $h \times h$. To get an output with multiple channels, we can create a kernel tensor of shape $h \times h \times c_i$ for every output channel, we then concatenate them on the output channel dimension, so that the shape of the convolution kernel is $h \times h \times c_i \times c_0$. In cross-correlation operations, the result on each output channel is calculated from the convolution kernel corresponding to that output channel and takes input from all channels in the input tensor. The outcome on each output channel is determined from the convolution kernel corresponding to that output channel in cross-correlation operations and takes input from all channels in the input tensor.

3.3.4 Pooling Layer

Sometimes, as we process images, we want to progressively reduce the spatial resolution of our hidden representations, aggregating information so that the higher we go up in the network, the greater the input receptive field to which each hidden node is sensitive. By gradually aggregating information, creating larger and larger maps, we achieve the goal of eventually learning global representation, while at the same time retaining all the advantages of the convolutional layers in the intermediate layers of processing. In this section, pooling layers are studied, which serve the dual purpose of merging semantically similar features into one and of spatially downsampling representations.

Like convolutional layers, pooling operators consist of a fixed-shaped window that slides across all regions of the input according to its stride, computing a single output for each

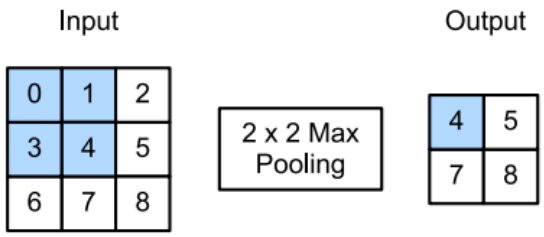


Figure 3.17: Maximum pooling with a pooling window shape of 2×2, **Zhang et al. (2020)**

position crossed by the window. However, unlike the cross-correlation operation of inputs and kernels in the convolutional layer, the pooling layer does not include any parameters, i.e., there is no kernel. Instead, pooling operators are deterministic, usually measuring either the maximum (max pooling) or the average value (average pooling) of the elements in the pooling window. In both cases, as with the cross-correlation operator, the pooling window starts from the top left of the input tensor and slides through the input tensor from left to right and from top to bottom. The maximum or average value of the input sub-tensor in the window is determined at each position that the pooling window reaches, depending on whether the maximum or average pooling value is used. Figure 3.17 shows a maximum pooling operation, the four elements are derived from the maximum value in each pooling window:

$$\begin{aligned} &\max(0,1,3,4) = 4, \quad \max(1,2,4,5) = 5, \\ &\max(3,4,6,7) = 7, \quad \max(4,5,7,8) = 8. \end{aligned} \tag{3.3.6}$$

As with convolutional layers, we can alter the polling operation to achieve a desired output shape by padding the input and adjusting the stride. When processing multi-channel input data, the pooling layer pools each input channel separately, rather than summing up inputs over channels as in a convolutional layer. This means that the number of input channels is the same as the number output channels of the pooling layer.

3.4 Dropout

Dropout **Srivastava et al. (2014)** is a regularisation technique which involves injecting noise while computing each internal layer during forward propagation. In each forward pass, we randomly set some nodes to zero in each layer before calculating the subsequent layer., i.e., we dropout some neurons during training. The injected noise is added in an unbiased manner so that the expected value of each layer, while fixing the others, is equal to the value it would have taken absent noise. During dropout, each layer normalised using the fraction of nodes that have been retained. In general, with dropout probability p, each intermediate activation

h is replaced by a random variable h' using the equation

$$h' = \begin{cases} 0 & \text{with probability } p \\ \dfrac{h}{1-p} & \text{otherwise} \end{cases} \tag{3.4.1}$$

with $\mathbb{E}[\ h'\ ] = h$, probability of dropping p is a hyperparameter. Dropout helps reduce the number of learnable parameters in fully connected layers since some nodes are dropped, this helps speed up training of neural networks. Figure 3.18 shows a fully connected layer before and after dropout using $p = 0.6$.

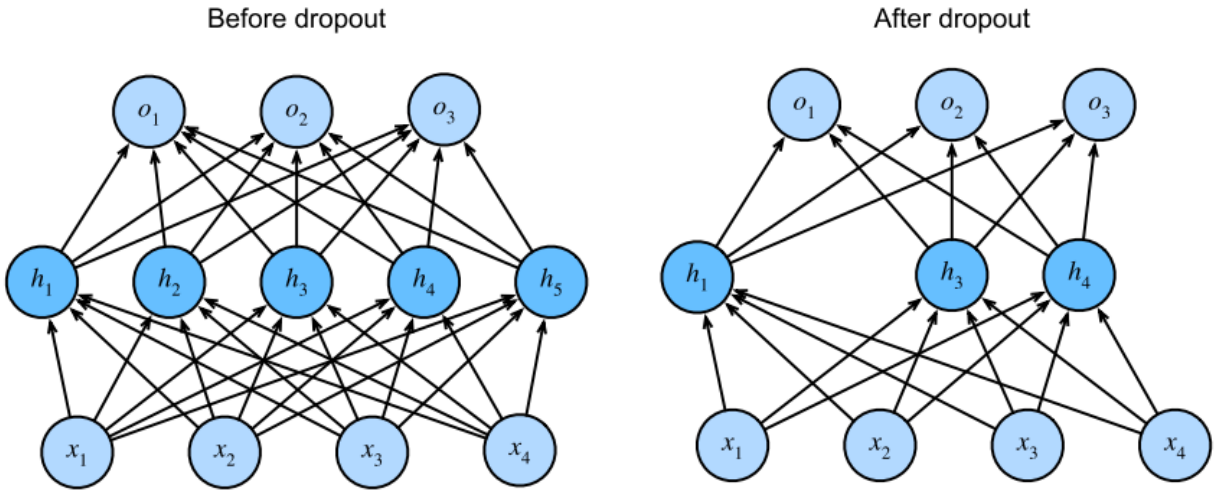


Figure 3.18: Fully connected layer before and after dropout, **Zhang et al. (2020)**

3.5 Batch Normalisation

A number of practical challenges arises when training deep learning models. First, choices in pre-processing data often make a huge difference in the final results. Hence, we normalise the input layer, hidden layers should also be normalised. Second, if a model learns mapping $X \to Y$, if the distribution of X changes, we might need to retrain the model by trying to align distribution of X with distribution of Y. This drift in the distribution of the variables could hamper the convergence of the network, we might also need to make compensatory adjustments in the learning rates. Third, regularisation becomes more critical since deeper networks are complex and easily capable of overfitting.

Batch Normalisation **Ioffe and Szegedy (2015)**, tackles these challenges (and more) in two steps as follows: In each training iteration, normalise the inputs of preceding layers by subtracting the mean and dividing by their standard deviation, where both are estimated based on the statistics of the current minibatch. Next, we apply scale coefficient and shift offset, these operations maintains the expressive power of the network to learn whether to keep the original (or slightly different normalised) input or the normalised input when moving to the next layer. In general, if $\boldsymbol{x} \in \mathcal{B}$ is an input to batch normalisation (BN) that is from a minibatch $\mathcal{B}$, batch normalisation transforms $\boldsymbol{x}$ according to the equation:

$$\mathrm{BN}(\boldsymbol{x}) = \boldsymbol{\gamma} \odot \frac{\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta} \tag{3.5.1}$$

where $\odot$ is the elementwise product operator, $\hat{\boldsymbol{\mu}}$ and $\hat{\boldsymbol{\sigma}}$ are the sample mean and sample standard deviation of minibatch $\mathcal{B}$ respectively, $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are the scale parameter and shift parameter respectively. These parameters are learnable parameters that have the same shape as $\boldsymbol{x}$. As a result, variable magnitudes for intermediate layers cannot diverge during training because the batch normalisation centers and rescales back to the specified mean and size. It also allows high learning rate since it makes sure there is no activation that gets really high or really low. $\hat{\boldsymbol{\mu}}$ and $\hat{\boldsymbol{\sigma}}$ are calculated using the equation

$$\begin{aligned} \hat{\boldsymbol{\mu}}_{\mathcal{B}} &= \frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{x} \in \mathcal{B}} \boldsymbol{x} \\ \hat{\boldsymbol{\sigma}}_{\mathcal{B}}^2 &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}})^2 + \epsilon \end{aligned} \tag{3.5.2}$$

A small $\epsilon > 0$ is added to the variance estimate to ensure that we never attempt to divide by zero, this extra constant adds noise estimates of the mean and variance to each hidden layer activations, similar to dropout which has helps in reducing overfitting.

Batch normalisation layers are often used for convolutional layers, added after convolutional layers and before the non-linear activation functions. When the convolution has several output channels, we need to perform batch normalisation for each output of these channels, and each channel has its own scale and shift parameters, all of which are scalars. Assume that our minibatches contain m examples and that for each channel, the output of the convolution is of size $q \times q$. For convolutional layers, each batch normalisation is carried out over the $m \times q \times q$ elements per output channel simultaneously. Thus, when calculating mean and

variance, we collect values across all spatial locations and then apply the same mean and variance within the given channel to normalise the value at each spatial location.

3.6 Gradient Descent Optimisation Algorithms

We introduced minibatch stochastic gradient descent (SGD) in Section 1.6, which improved on the convergence rate of gradient descent, where we made parameter updates using a minibatch with size b, instead of the whole dataset at each iteration. We showed that this optimiser has good convergence rate for smooth convex objective (loss) functions, using sufficiently small minibatch size b. In practice, common loss functions for neural networks are highly non-convex, minimising the loss becomes a challenge using SGD algorithms due to saddle points in the function **Dauphin et al. (2014)**, saddle points are usually surrounded by a plateau of the same error, which makes it notoriously difficult for SGD to escape, as the gradient is close to zero in all dimensions. In this section SGD implies stochastic gradient descent with minibatches, also we leave out hyperparameter b and the summation over k from Equation 1.6.4 for simplicity.

In the case mentioned above, SGD oscillates around the slopes of the areas where the surface curves more steeply in one direction than in another, while only slowing down towards the local optimum. Momentum **Qian (1999)** is a method that helps speed SGD in the relevant direction and dampens oscillations, by adding a fraction ρ of the update vector of the previous time step to the current update vector:

$$\begin{aligned} \boldsymbol{v}^{k+1} &= \rho \boldsymbol{v}^k + \alpha \nabla f(\boldsymbol{x}^k) \\ \boldsymbol{x}^{k+1} &= \boldsymbol{x}^k - \boldsymbol{v}^{k+1} \end{aligned} \tag{3.6.1}$$

The momentum term ρ is usually set to 0.9 or a similar value. The momentum term increases for dimensions whose gradients point in the same direction and decreases updates for dimensions whose gradients change directions. As a consequence, we are achieving faster convergence and reduced oscillation.

Nesterov accelerated gradient **Nesterov (1983)** is a way to give the momentum the notion of where it is going so that it knows when to slow down depending on the slope of the hill. In Equation 3.6.1, the momentum term $\rho \boldsymbol{v}^t$ is used to move the parameters $\boldsymbol{x}$. Computing

$\boldsymbol{x} - \rho\boldsymbol{v}^t$ thus gives us an approximation of the next position of the parameters. We can now effectively look ahead by calculating the gradient not with respect to our current parameters $\boldsymbol{x}$ but with respect to the approximate future position of our parameters:

$$\begin{aligned} \boldsymbol{v}^{k+1} &= \rho\boldsymbol{v}^k + \alpha\nabla f(\boldsymbol{x} - \rho\boldsymbol{v}^t) \\ \boldsymbol{x}^{k+1} &= \boldsymbol{x}^k - \boldsymbol{v}^{k+1} \end{aligned} \tag{3.6.2}$$

For easy implementation, we would want update in terms of $\boldsymbol{x},\ \nabla f(\boldsymbol{x})$. To achieve this, we set $\hat{\boldsymbol{x}}^k = \boldsymbol{x} - \rho\boldsymbol{v}^t$ and rearrange, Equation 3.6.2 then becomes

$$\begin{aligned} \boldsymbol{v}^{k+1} &= \rho\boldsymbol{v}^k + \alpha\nabla f(\hat{\boldsymbol{x}}^k) \\ \hat{\boldsymbol{x}}^{k+1} &= \hat{\boldsymbol{x}}^k - \rho\boldsymbol{v}^{k+1} - \rho\boldsymbol{v}^k + \boldsymbol{v}^{k+1} \\ &= \hat{\boldsymbol{x}}^k + \boldsymbol{v}^{k+1} + \rho(\boldsymbol{v}^{k+1} - \boldsymbol{v}^k) \end{aligned} \tag{3.6.3}$$

While momentum first computes the current gradient and then takes a big jump in the direction of the updated accumulated gradient, Nesterov accelerated gradient first makes a big jump in the direction of the previous accumulated gradient, measures the gradient and then makes a correction, which results in the complete Nesterov accelerated gradient update. This anticipatory update prevents us from going too far, resulting in improved responsiveness. We have been able to change updates to the slope of the loss function and to speed up SGD in turn. Next, we would adapt updates to each individual parameter in order to perform larger or smaller updates depending on their significance.

Adagrad **Duchi et al. (2011)** adapts the learning rate to the parameters, allowing slight changes (i.e. low learning rates) for parameters associated with regularly occurring features, and higher updates (i.e. high learning rates) for parameters associated with uncommon features. Previously, we performed an update for all parameters $\boldsymbol{x}$ at once as every parameter $\boldsymbol{x}_i$ used the same learning rate α. As Adagrad uses a different learning rate for every parameter $\boldsymbol{x}_i$ at every time step k, we first show Adagrad's per-parameter update, then vectorise them. Using $\boldsymbol{g}_k$ to denote the gradient at time step k, and $\boldsymbol{g}_i^k$ to denote the partial derivative of the objective function with respect to the parameter $\boldsymbol{x}_i$ at time step k:

$$\boldsymbol{x}_i^{k+1} = \nabla f(\boldsymbol{x}_i^k) \tag{3.6.4}$$

SGD update for every parameter $\boldsymbol{x}_i$ at each time step k becomes

$$\boldsymbol{x}_i^{k+1} = \boldsymbol{x}_i^k - \alpha\boldsymbol{g}_i^k \tag{3.6.5}$$

In its update rule, Adagrad modifies the general learning rate α at each time step k for every parameter $\boldsymbol{x}_i$ based on the past gradients that have been computed for $\boldsymbol{x}_i$:

$$\boldsymbol{x}_i^{k+1} = \boldsymbol{x}_i^k - \frac{\alpha}{\sqrt{\boldsymbol{G}_{ii}^k + \epsilon}} \cdot \boldsymbol{g}_i^k \tag{3.6.6}$$

$\boldsymbol{G}^k \in \mathbb{R}^{d,d}$ is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients with respect to $\boldsymbol{x}_i$ up to time step k, while ϵ is a smoothing term that avoids division by zero, vectorising Equation 3.6.6, we get:

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \frac{\alpha}{\sqrt{\boldsymbol{G}^k + \epsilon}} \odot \boldsymbol{g}^k \tag{3.6.7}$$

Adagrad's eliminates the need to manually tune the learning rate, most implementations use a default value of 0.01.

Adaptive Moment Estimation (Adam) **Kingma and Ba (2014)** computes individual adaptive learning rates for different parameters from estimate of the first and second moments of the gradients. Adam stores an exponentially decaying average of past squared gradients $\boldsymbol{v}^k$, and an exponentially decaying average of past gradients $\boldsymbol{m}^k$, similar to momentum. The decaying averages of past and past squared gradients $\boldsymbol{m}^k$ and $\boldsymbol{v}^k$ respectively are calculated as follows:

$$\begin{aligned} \boldsymbol{m}^k &= \beta_1 \boldsymbol{m}^{k-1} + (1 - \beta_1)\boldsymbol{g}^k \\ \boldsymbol{v}^k &= \beta_2 \boldsymbol{v}^{k-1} + (1 - \beta_2)(\boldsymbol{g}^k)^2 \end{aligned} \tag{3.6.8}$$

$\boldsymbol{m}^k$ and $\boldsymbol{v}^k$ are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively. Since $\boldsymbol{m}^k$ and $\boldsymbol{v}^k$ are initialized as vectors of 0's, they become biased towards zero, especially during the initial time steps, and especially when the decay rates are small, i.e., β_1 and $\beta_2$are close to 1). Bias-corrected first and second moment estimates are calculated to tackle this problem:

$$\begin{aligned} \hat{\boldsymbol{m}}^k &= \frac{\boldsymbol{m}^k}{1 - \beta_1^k} \\ \hat{\boldsymbol{v}}^k &= \frac{\boldsymbol{v}^k}{1 - \beta_2^k} \end{aligned} \tag{3.6.9}$$

The Adam update rule is given as:

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \frac{\alpha}{\sqrt{\hat{\boldsymbol{v}}^k} + \epsilon}\, \hat{\boldsymbol{m}}^k \tag{3.6.10}$$

Default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ were proposed by the authors. Adam works well in practice and compares favourably to other adaptive learning-method algorithms. We used the Adam optimiser for our CNN in this project.

Chapter 4

Plant Disease Detection

4.0 Problem Statement and Dataset

In this chapter, we adopted three different trained CNN models to detect plant disease using the PlantVillage dataset (`https://github.com/spMohanty/PlantVillage-Dataset`)

Figure 4.1: The PlantVillage image dataset. This dataset contains 38 categories of diseased or healthy leaf images

(Figure 4.1) and compared results of each model. This dataset comprises healthy or diseased leaf images classified into 38 labels, 42, 754 images, 14 different crop species and 20 different diseases including healthy ones. The dataset was split with 3, 847 images set out as validation set and 3, 102 images as test set, all images were reshaped into size $224 \times 224 \times 3$ since images usually comes in different sizes and a batch size of 64 was used. Different variations of images in the dataset was created using different data augmentation techniques (flipping, zooming, rotating, warping, lightening), data augmentation is used to artificially create new data from the existing train dataset by creating modified version of the images with different amount of variances, this technique improves the training accuracy and helps the model generalize better from different variance added to the image **Stephen et al. (2019)**.

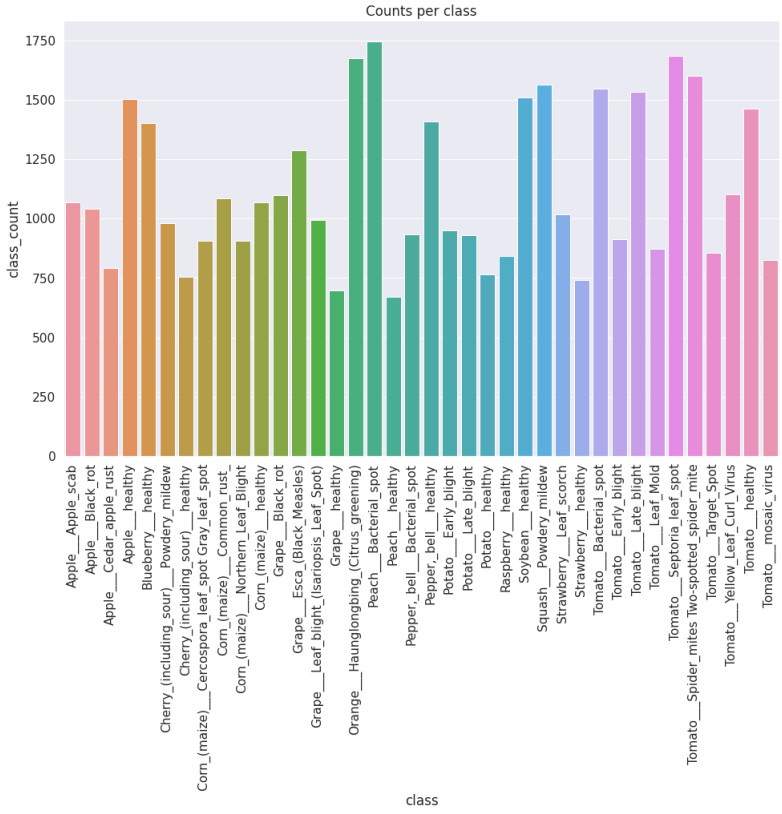


Figure 4.2: Bar chart showing the number of images in each class

4.1 Model Training

Training the models was performed using Fastai deep learning library **Howard and Gugger (2020)** and Pytorch deep leaning library **Paszke et al. (2019)** on Python 3.7, running on Google colab (`https://colab.research.google.com`) with the provided free GPU from colab, it took about 3 to 6 minutes to train the models per epoch. The network was initialized with random weights, using a categorical cross-entropy loss metric, network weights were optimized using the Adam optimization algorithm with a learning rate of 0.01, a set of 64 images with a size of 224×224 were fed to the network as a batch per iteration. A pseudo-random number was seeded to make sure the same weight values are initialised for each training instance, which makes sure results are reproducible for each trained model.

4.2 Base Model Architecture

A convolution neural network was built to learn the class where each image belongs to. The network consists of the input image, six convolution layers, two fully connected layers and the output layer (Figure 4.3). Each convolution layer was of kernel size 3×3, applied at stride 1 and padding 1, the number of kernels were 16, 32, 64 , 128 , 256 , 512 respectively for each

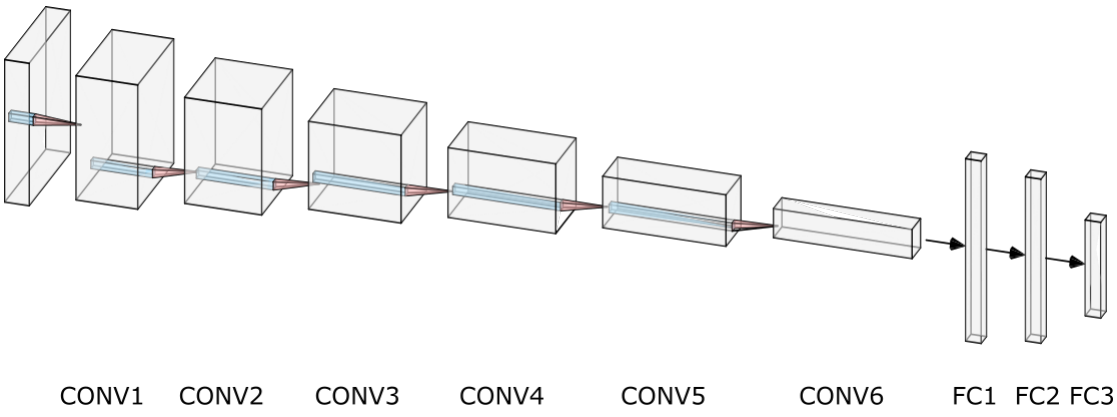


Figure 4.3: Simplified base model architecture

preceding convolution layer, batch normalisation was also added after each convolution layer to speed up training. To reduce dimensions of output features after each convolution layer, max pooling layers of kernel size 2×2, applied at stride 2 and zero padding was added

after each convolution layer (Table 4.2). Output image from the final max pool layer was flattened to a 1D array and fed into fully connected layers with size 1024, , 512, and , 38 respectively, batch norm and dropout with rate 0.5 was added to the first two fully connected layers. ReLU activation was used for all layers except the output layer which uses softmax to compute probabilities of an image belonging to a particular class. The model contained $6,841,766$ trainable parameters, batch size 64 and learning rate 0.01 was used during training.

	Type / Stride(s) / Padding(p)	Filter Shape	Input Size
CONV1	Conv / $s = 1$ / $p = 1$	$3 \times 3 \times 3 \times 16$	$224 \times 224 \times 3$
	Max Pool / $s = 2$ / $p = 0$	2×2	$112 \times 112 \times 16$
CONV2	Conv / $s = 1$ / $p = 1$	$3 \times 3 \times 16 \times 32$	$112 \times 112 \times 16$
	Max Pool / $s = 2$ / $p = 0$	2×2	$56 \times 56 \times 32$
CONV3	Conv / $s = 1$ / $p = 1$	$3 \times 3 \times 32 \times 64$	$56 \times 56 \times 32$
	Max Pool / $s = 2$ / $p = 0$	2×2	$28 \times 28 \times 64$
CONV4	Conv / $s = 1$ / $p = 1$	$3 \times 3 \times 64 \times 128$	$28 \times 28 \times 64$
	Max Pool / $s = 2$ / $p = 0$	2×2	$14 \times 14 \times 128$
CONV5	Conv / $s = 1$ / $p = 1$	$3 \times 3 \times 128 \times 256$	$14 \times 14 \times 128$
	Max Pool / $s = 2$ / $p = 0$	2×2	$7 \times 7 \times 256$
CONV6	Conv / $s = 1$ / $p = 1$	$3 \times 3 \times 256 \times 512$	$7 \times 7 \times 256$
	Max Pool / $s = 2$ / $p = 0$	2×2	$3 \times 3 \times 512$
FC1	Fully Connected	4068×1024	$1 \times 1 \times 4068$
FC2	Fully Connected	1024×512	$1 \times 1 \times 1024$
FC3	Fully Connected	512×38	$1 \times 1 \times 512$
	Softmax	Classifier	$1 \times 1 \times 38$

Table 4.2: Base model architecture

4.3 Transfer Learning

To improve training accuracy and speed up training time we adopt transfer learning. Transfer learning can be described as the improvement of learning of a new task by the transition of information from a previously learned similar task, model weights from pre-trained models created for large-scale image-classification datasets, such as ImageNet are reused to solve similar image-classification tasks. The following steps were taken to implement transfer learning:

1. Pre-train the source model on a source dataset. The pre-trained model and its parameters were downloaded and the source dataset used is the ImageNet dataset containing $1,000$ classes in our case.

2. Create a new neural network model, i.e., the target model. This replicates all model designs and their parameters on the source model, except the output layer. We assume that these model parameters contain the knowledge learned from the source dataset and this knowledge will be equally applicable to the target dataset. We also assume that the output layer of the source model is closely related to the labels of the source dataset and is therefore not used in the target model.

3. Add an output layer whose output size is the number of target dataset classes to the target model.

4. Train the target model on a target dataset. We would freeze the pre-trained layers and train the output layer from scratch, then unfreeze the pre-trained layers and jointly train both the pre-trained layers and the output layer of the target model with a much more smaller learning rate.

VGG16 and ResNext-50 were used as the pre-trained models in this project.

4.3.1 VGG16

VGG16 **Simonyan and Zisserman (2015)** model achieved 7.1% top-5 error rate in the ImageNet dataset and came second for classification in the ILSVRC 2014 challenge. It makes the improvement over AlexNet **Krizhevsky et al. (2012)** by replacing large kernel-sized

filters (11 and 5 in the first and second convolutional layer respectively) with stacks of 3×3 kernel-sized filters. Each stack of 3×3 convolutional layer with stride 1, pad 1 and different depths is followed by 2×2 max pooling layers applied at stride 2, three fully-connected (FC) layers follow the final max pool layer; the first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class), the final layer is the softmax layer and ReLU is applied to all hidden layers (Figure 4.4). It was shown that the added depth and smaller filters is beneficial for classification accuracy.

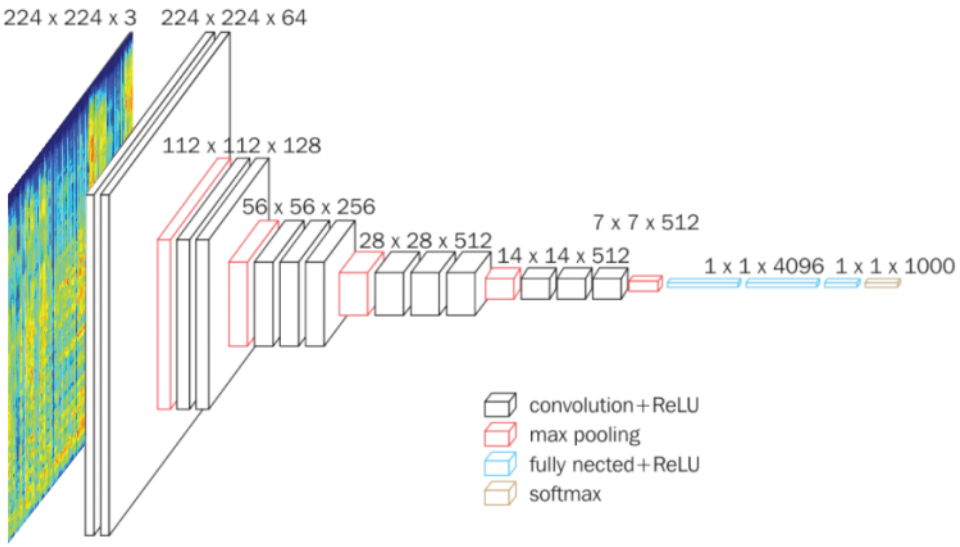


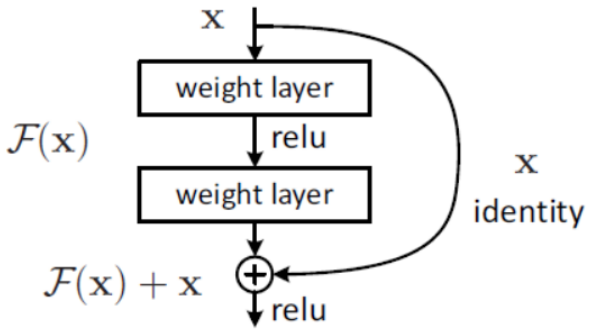
Figure 4.4: VGG16 model architecture, **Das et al. (2019)**

4.3.2 ResNext-50

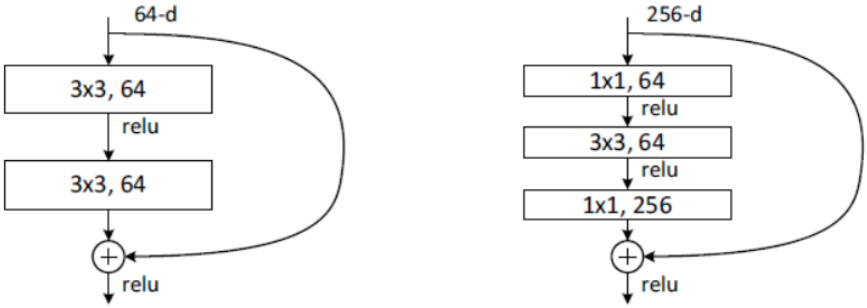
With the success of plain deep networks such as VGG16, the knee-jerk reaction would be to build even more deeper networks (simply stacking more convolutional and pooling layers) in order to improve prediction results, but a 20-layer plain network got lower training error and test error than 56-layer plain network when trained with the CIFAR-10 dataset **He et al. (2015a)**. Deep networks are in fact susceptible to vanishing/exploding gradients, making it harder to optimise compared to shallow networks. During backpropagation, deriving partial derivative of the error function with respect to the current weight in each iteration of training, has the effect of multiplying n of these small/large numbers to compute gradients of intermediate layers in an n-layer network. For deeper networks, multiplying n of these small

numbers will become zero (vanished), while multiplying n of these large numbers will become too large (exploded).

Residual learning introduced in ResNet architecture **He et al. (2015a)** tackles the problem stated above by using network layers to fit a residual mapping using skip connections instead of directly trying to fit a desired underlying mapping, i.e., for the output $H(\boldsymbol{x}) = F(\boldsymbol{x}) + \boldsymbol{x}$, use weight layers to learn residual mapping: $F(\boldsymbol{x}) = H(\boldsymbol{x}) - \boldsymbol{x}$ (Figure 4.5a) as we move to the next layer, instead of desired function $H(\boldsymbol{x})$ directly.



(a) A residual block



(b) A bottleneck building block for ResNet-50/101/152

Figure 4.5: Residual learning blocks, **He et al. (2015a)**

ResNet architecture consists of a 7×7 convolutional layer, 2×2 pooling layer, stacks of residual blocks with two 3×3 convolutional layers, number of filters are double after each block, and the layers are down-sampled spatially using stride 2 and a global average pooling layer after the last convolutional layer. The final output is the only fully connected layer in the network, batch norm was added after every convolutional layer. For deep residual

networks, 1×1 convolutional layers (bottlenecks) are added to the start and end of residual blocks (Figure 4.5b) to improve efficiency. Adding bottlenecks is a technique suggested in GoogLeNet **Szegedy et al. (2015)** (Inception-v1). The 1×1 convolutional layers reduces the number of connections (parameters) since it's only computation occurs on the channel dimension, while not degrading the performance of the network. ResNet won the ILSVRC 2015 in image classification, detection, and localization with 3.57% top-5 error rate (with ensembles).

Zagoruyko and Komodakis (2017) argued that residual blocks are more important than depth, hence, wide residual networks were introduced, where filter size for residual blocks where increased by factors of k ($F \times k$ filters instead of F filters in each layer). It was also showed that increasing width instead of depth is more computationally efficient. Wide 50-layer residual networks outperforms 152-layer ResNet when trained on the ImageNet dataset 6.03% versus 6.16% top-5 error rate.

ResNext **Xie et al. (2017)** tried to improve wide ResNet by increasing width of the residual blocks through multiple parallel pathways. Multiple residual blocks with bottlenecks are stacked in parallel with d internal dimension for each path and c total number of pathways, called cardinality (Figure 4.6). The architecture for ResNext-50 is the same as ResNet-50 but with the extra parallel pathways in the residual blocks, it achieved a 6.6% top-5 error rate when trained on the ImageNet dataset with $1,000$ classes compared to a 8.2% error rate for ResNet-50 on the same dataset.

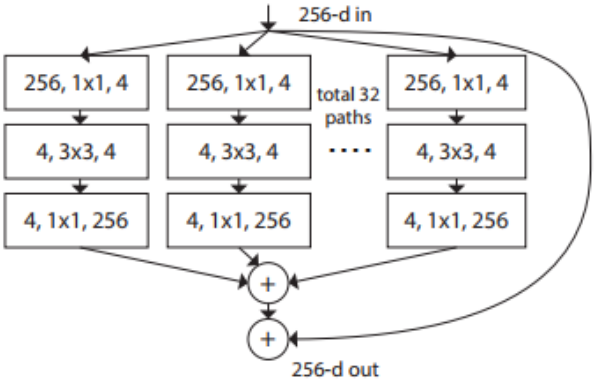


Figure 4.6: A block of ResNeXt with cardinality = 32, **Xie et al. (2017)**

4.4 Results

In this section, results gotten on validation data are compared for the three different models studied in this research; a base CNN model, a VGG16 model and a ResNext-50 $32 \times 4d$ model, evaluation metrics used are also defined. Performance measures such as accuracy, precision, recall, and f1 score are defined by using four features; true positive (tp_i) - correctly identified prediction for each class, true negative (tn_i) - correctly rejected prediction for certain class,

		Predicted class			
		A	B	C	D
Actual class	A	9	1	0	0
	B	1	15	3	1
	C	5	0	24	1
	D	0	4	1	15

Table 4.3: Confusion matrix for 4-class classification. The blue, red, and brown cells shows tn, fp, and fn values respectively for class A, matrix diagonal shows tn values for each class.

false positive (fp_i) - incorrectly identified predictions for certain class, and false negative (fn_i) - incorrectly rejected data for certain class, of class C_i, these four features can be gotten from the confusion matrix (Table 4.3). Weighted average was used for all performance measure, except accuracy, because it factors in weight that depends on the number of true labels of each class which gives a better picture for imbalanced datasets. The performance measures are defined as follows:

- Accuracy: Accuracy is the ratio of correctly predicted observation to the total observations. It is not a good measure for imbalanced dataset i.e., dataset with uneven class distribution, since it does not distinguish between the numbers of correctly classified examples of different classes, it is given by:

$$\text{Accuracy} = \frac{\sum_{i=1}^{m} \frac{tp_i}{tp_i + fp_i + fn_i + tn_i}}{m} \tag{4.4.1}$$

- Top-3 Accuracy: Top-3 accuracy takes the 3 model predictions with higher probabilities, if one of them is a true label, it classifies the prediction as correct. Accuracy is a particular situation where only the highest probability prediction is taken into consideration.

- Precision: Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. This metric answers the question: of all image labelled as being to a class, how many are actually in that class, it is given by:

$$\text{Precision}_{\text{Weighted}} = \frac{\sum_{i=1}^{m} \left(|y_i| \, \frac{tp_i}{tp_i + fp_i} \right)}{\sum_{i=1}^{m} |y_i|} \tag{4.4.2}$$

- Recall or sensitivity: Recall is the ratio of correctly predicted positive observations to the all observations in actual class. The metric answers the question: of all the images belonging to a class, how many were labelled, it is given by:

$$\text{Recall}_{\text{Weighted}} = \frac{\sum_{i=1}^{m} \left(|y_i| \, \frac{tp_i}{tp_i + fn_i} \right)}{\sum_{i=1}^{m} |y_i|} \tag{4.4.3}$$

- F1 score: F1 Score is the weighted average of precision and recall, it is usually more useful than accuracy, especially for uneven class distribution, it is given by:

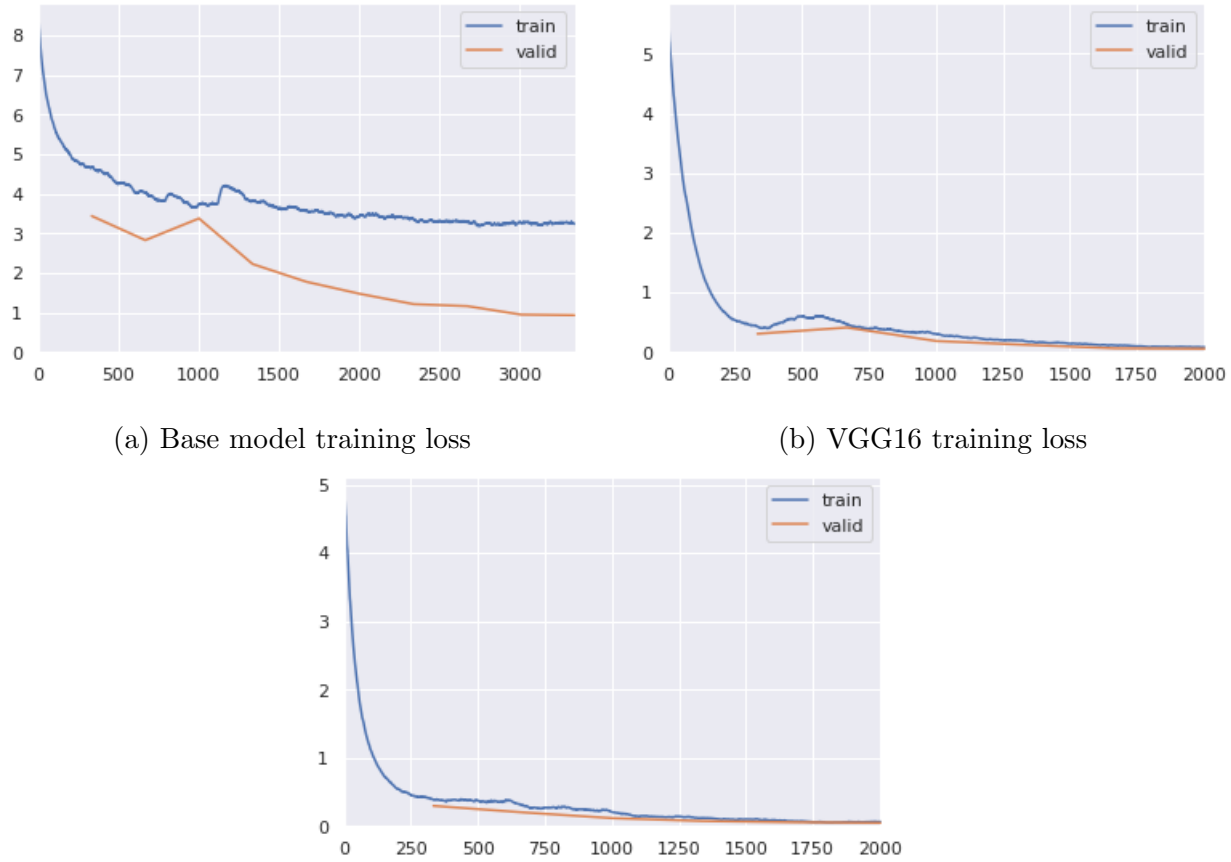
$$\text{F1 score}_{\text{Weighted}} = \frac{\sum_{i=1}^{m} \left(|y_i| \, \frac{2tp_i}{2tp_i + fp_i + fn_i} \right)}{\sum_{i=1}^{m} |y_i|} \tag{4.4.4}$$

where y_i is the number of instances for class i. Table 4.4 shows the results gotten for each model.

	Accuracy	Top-3 Accuracy	Precision	Recall	F1 score
Base Model	0.9277	0.9923	0.9329	0.9277	0.9277
VGG16	0.9821	0.9995	0.9853	0.9821	0.9819
ResNext-50	0.9823	1.00	0.9850	0.9823	0.9822

Table 4.4: Comparing different evaluation metrics for the models studied on validation data

Visualisation of the model was carried out using the cross-entropy (or logistic) loss defined in Equation 1.4.6, loss of the model decreases as training is done (Figure 4.7). ResNext-50 has the best performance since training loss and validation loss converges, with minimal space in between both curves as the model trains, the base model performs worst.



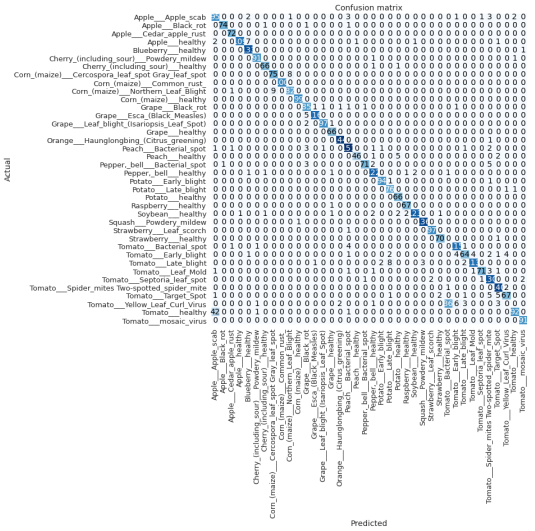
(a) Base model training loss

(b) VGG16 training loss

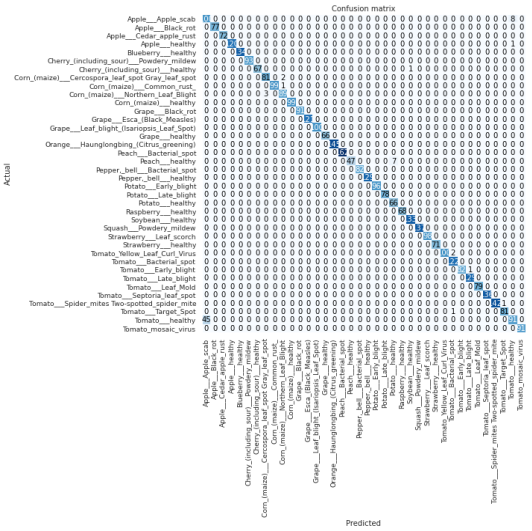
(c) ResNext-50 training loss

Figure 4.7: Training loss for different models, the horizontal axes denotes number of epochs, while the vertical axes denotes loss

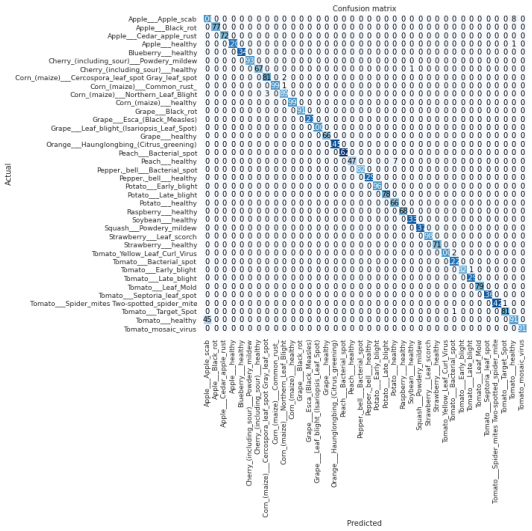
Confusion matrices are used to visualise more outputs (Figure 4.8). Most confused prediction pair for VGG16 and ResNext50 are the same; tomato healthy and apple scab with 45 incorrectly identified predictions, while for the base model, tomato yellow leaf and tomato bacterial spot with 86 incorrectly identified predictions is the most confused prediction pair.



(a) Base model confusion matrix



(b) VGG16 confusion matrix



(c) ResNext-50 confusion matrix

Figure 4.8: Confusion matrix for different models, thicker colours signifies better accuracy

Chapter 5

Conclusion

In this project, we implemented convolutional neural networks to classify plant leaf diseases using images of healthy and diseased leaves reaching an accuracy of 98%. In Chapter 1, we defined basic terms and gave corresponding results, then a binary classification problem was introduced and how to solve the problem using logistic regression. It is difficult to solve binary classification problem where the two classes can not be separated by a hyperplane using standard machine learning algorithms, so we introduced multi layer perceptrons (neural networks) to solve this problem. Basic optimisation and regularisation algorithms for neural networks was also introduced in this chapter.

In Chapter 2, we reviewed advancements in neural network architectures and how better computer power has helped accelerate research into the field due to high computational demands of training neural networks. We also reviewed areas of agriculture where deep learning has been applied with good accuracy and generalisation, from plant phenotyping and yield prediction, to weed management and pest control, and crop diseases. These high quality models has helped improved agricultural produce yield and improved efficiency of solving different problems faced in the agricultural process.

In Chapter 3, we took a deep dive into training neural networks (convolutional neural networks in particular), first looking at some different activation functions which introduces non-linearity to the network. Then moving on to explain the backpropagation algorithm which forms the basis of training and optimising neural networks. Building blocks of convolutional neural networks were then investigated, and finally modern methods for optimising

and regularising these networks were studied.

Chapter 4 deals with the main problem statement, which consists of identifying healthy and diseased crop plants using convolution neural networks. Three different models were trained on the dataset - a custom built base model and two popular models (VGG16 and ResNext-50). The popular models were studied in detail, considering mainly it's architectures and what makes them achieve state of the art result. Finally, we compare and visualise results from the three models, with ResNext-50 having the best all round performance.

The results gotten from this project could be improved upon by making ensembles of the different models, we aim to study this problem further. Also, more applications of deep learning in agriculture can be studied, for example, fine mapping of key soil nutrients using high resolution remote sensing image with convolutional neural networks.

References

Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169.

Badrinarayanan, V., Kendall, A., and Cipolla, R. (2017). Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 39(12):2481–2495.

Barré, P., Stöver, B. C., Müller, K. F., and Steinhage, V. (2017). Leafnet: A computer vision system for automatic plant species identification. *Ecological Informatics*, 40:50–56.

Baweja, H. S., Parhar, T., Mirbod, O., and Nuske, S. (2018). Stalknet: A deep learning pipeline for high-throughput measurement of plant stalk count and stalk width. In Hutter, M. and Siegwart, R., editors, *Field and Service Robotics*, pages 271–284, Cham. Springer International Publishing.

Behera, B., Kumaravelan, G., et al. (2019). Performance evaluation of deep learning algorithms in biomedical document classification. In *2019 11th International Conference on Advanced Computing (ICoAC)*, pages 220–224. IEEE.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

Bottou, L., Curtis, F. E., and Nocedal, J. (2018). Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311.

Boyd, S., Boyd, S. P., and Vandenberghe, L. (2004). *Convex optimization*. Cambridge university press.

Bryson, A. E. and Ho, Y. C. (1969). *Applied Optimal Control*. Blaisdell, New York.

Chellapilla, K., Puri, S., and Simard, P. (2006). High Performance Convolutional Neural Networks for Document Processing. In Lorette, G., editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France). Université de Rennes 1, Suvisoft. http://www.suvisoft.com.

Chen, J., Fan, Y., Wang, T., Zhang, C., Qiu, Z., and He, Y. (2018). Automatic segmentation and counting of aphid nymphs on leaves using convolutional neural networks. *Agronomy*, 8(8):129.

Chen, J., Liu, Q., and Gao, L. (2019). Visual tea leaf disease recognition using a convolutional neural network model. *Symmetry*, 11(3):343.

Ciregan, D., Meier, U., and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3642–3649. IEEE.

Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2010). Deep, big, simple neural nets for handwritten digit recognition. *Neural Comput.*, 22(12):3207–3220.

Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289.

Cooper, J. and Dobson, H. (2007). The benefits of pesticides to mankind and the environment. *Crop Protection*, 26:1337–1348.

Das, P., Acharjee, A., and Marium-E-Jannat (2019). Double coated vgg16 architecture: An enhanced approach for genre classification of spectrographic representation of musical pieces. *2019 22nd International Conference on Computer and Information Technology (ICCIT)*, pages 1–5.

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., and Weinberger, K. Q.,

editors, *Advances in Neural Information Processing Systems*, volume 27, pages 2933–2941. Curran Associates, Inc.

Dekel, O., Gilad-Bachrach, R., Shamir, O., and Xiao, L. (2012). Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13(Jan):165–202.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Li, F. F. (2009). Imagenet: a large-scale hierarchical image database. pages 248–255.

Denker, J. S., Gardner, W. R., Graf, H. P., Henderson, D., Howard, R. E., Hubbard, W., Jackel, L. D., Baird, H. S., and Guyon, I. (1989). Neural network recognizer for hand-written zip code digits. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 1*, pages 323–331. Morgan-Kaufmann.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7).

Duda, R. O. and Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. John Willey & Sons, New Yotk.

Everingham, M., Van Gool, L., Williams, C., Winn, J., and Zisserman, A. (2010). The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88:303–338.

Ferentinos, K. P. (2018). Deep learning models for plant disease detection and diagnosis. *Computers and Electronics in Agriculture*, 145:311–318.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202.

Garcia-Garcia, A., Orts-Escolano, S., Oprea, S., Villena-Martinez, V., and Garcia-Rodriguez, J. (2017). A review on deep learning techniques applied to semantic segmentation. *arXiv preprint arXiv:1704.06857*.

Gaurav Belani (2020). Should you choose a GPU or a TPU to train your machine learning models. `https://www.predictiveanalyticsworld.com/machinelearningtimes/`

should-you-choose-a-gpu-or-a-tpu-to-train-your-machine-learning-models/10460/. [Online; accessed 16-January-2020].

Girshick, R. B. (2015). Fast R-CNN. *CoRR*, abs/1504.08083.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In Gordon, G., Dunson, D., and Dudík, M., editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA. PMLR.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. The MIT Press.

Graves, A., Liwicki, M., Fernández, S., Bertolami, R., Bunke, H., and Schmidhuber, J. (2009). A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence*, 31:855–68.

Hahnloser, R., Sarpeshkar, R., Mahowald, M., Douglas, R., and Seung, H. (2000). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405:947–51.

He, K., Zhang, X., Ren, S., and Sun, J. (2015a). Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

He, K., Zhang, X., Ren, S., and Sun, J. (2015b). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, page 1026–1034, USA. IEEE Computer Society.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

Howard, J. and Gugger, S. (2020). Fastai: A layered api for deep learning. *Information*, 11(2):108.

Hu, J., Shen, L., and Sun, G. (2017). Squeeze-and-excitation networks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7132–7141.

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708.

Hubel, D. H. and Wiesel, T. (1962). Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. *Journal of Physiology (London)*, 160:106–154.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France. PMLR.

Jähne, B., Scharr, H., Körkel, S., Jähne, B., Haußecker, H., and Geißler, P. (1999). Principles of filter design. volume 2, pages 125–151. Academic Press.

Jarrett, K., Kavukcuoglu, K., Ranzato, M., and Lecun, Y. (2009). What is the best multi-stage architecture for object recognition? volume 12.

Kaur, H. and Garg, H. (2014). *Pesticides: Environmental Impacts and Management Strategies*.

Kelley, H. J. (1960). Gradient theory of optimal flight paths. *ARS Journal*, 30(10):947–954.

Khaki, S., Khalilzadeh, Z., and Wang, L. (2019). Classification of crop tolerance to heat and drought—a deep convolutional neural networks approach. *Agronomy*, 9(12).

Kim, K.-H., Kabir, E., and Jahan, S. A. (2017). Exposure to pesticides and the associated human health effects. *Science of The Total Environment*, 575:525 – 535.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Klambauer, G., Unterthiner, T., Mayr, A., and Hochreiter, S. (2017). Self-normalizing neural networks. *CoRR*, abs/1706.02515.

Knillmann, S. and Liess, M. (2019). *Pesticide Effects on Stream Ecosystems: Drivers, Risks, and Societal Responses*, pages 211–214.

Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*.

Krizhevsky, A., Hinton, G., et al. (2009). Learning multiple layers of features from tiny images.

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25.

LeCun, Y. (1985). Une procedure d'apprentissage pour reseau a seuil asymmetrique (a learning scheme for asymmetric threshold networks). In *Proceedings of Cognitiva 85, Paris, France*, pages 599–604.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Back-propagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324.

LeCun, Y., Huang, F. J., and Bottou, L. (2004). Learning methods for generic object recognition with invariance to pose and lighting. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR'04, page 97–104, USA. IEEE Computer Society.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., and Dollár, P. (2017). Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988.

Liu, C., Yin, F., Wang, Q., and Wang, D. (2011). ICDAR 2011 Chinese handwriting recognition competition. In *11th International Conference on Document Analysis and Recognition*, pages 1464–1469.

Long, J., Shelhamer, E., and Darrell, T. (2014). Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *International Conference on Machine Learning (ICML)*.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

Minsky, M. and Papert, S. (1969). *Perceptrons*. Cambridge, MA: MIT Press.

Mohanty, S., Hughes, D., and Salathe, M. (2016). Using deep learning for image-based plant disease detection. *arXiv preprint arXiv:1604.03169*.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA. Omnipress.

Nemirovski, A., Juditsky, A., Lan, G., and Shapiro, A. (2009). Robust stochastic approximation approach to stochastic programming. *SIAM Journal on optimization*, 19(4):1574–1609.

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). In *Doklady ANSSSR*, volume 269, pages 543–547.

Oh, K.-S. and Jung, K. (2004). GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314.

Oliveira, I., Cunha, R. L., Silva, B., and Netto, M. A. (2018). A scalable machine learning system for pre-season agriculture yield forecast. *arXiv preprint arXiv:1806.09244*.

Parr, T. and Howard, J. (2018). The matrix calculus you need for deep learning. *CoRR*, abs/1802.01528.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019).

Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *The Official Journal of the International Neural Network Society*, 12(1):145–151.

Ramachandran, P., Zoph, B., and Le, Q. V. (2017). Swish: a self-gated activation function. *CoRR*, abs/1710.05941.

Rasmussen, C. B. and Moeslund, T. B. (2019). Maize silage kernel fragment estimation using deep learning-based object recognition in non-separated kernel/stover rgb images. *Sensors*, 19(16):3506.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E., McClelland, J. L., and Group, P. R., editors, *Parallel distributed processing: explorations in the microstructure of cognition*, volume 1: foundations, pages 318–362. MIT Press, Cambridge, MA, USA.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A., and Li, F. F. (2014). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115.

Sa, I., Popović, M., Khanna, R., Chen, Z., Lottes, P., Liebisch, F., Nieto, J., Stachniss, C., Walter, A., and Siegwart, R. (2018). Weedmap: a large-scale semantic weed mapping framework using aerial multispectral imaging and deep neural network for precision farming. *Remote Sensing*, 10(9):1423.

Sanborn, M., Kerr, K., Sanin, L., Cole, D., Bassil, K., and Vakil, C. (2007). Non-cancer health effects of pesticides systematic review and implications for family doctors. *Canadian family physician Médecin de famille canadien*, 53:1712–20.

Sanchez-Bayo, F. and Goka, K. (2014). Pesticide residues and bees - a risk assessment. *PloS one*, 9:e94482.

Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y. (2013). Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition.

Sladojevic, S., Arsenovic, M., Anderla, A., Culibrk, D., and Stefanovic, D. (2016). Deep neural networks based recognition of plant diseases by leaf image classification. *Computational intelligence and neuroscience*, 2016.

Sobel, I. and Feldman, G. (1968). A 3x3 isotropic gradient operator for image processing. *Stanford Artificial Intelligence Project*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

Stallkamp, J., Schlipsing, M., Salmen, J., and Igel, C. (2011). The german traffic sign recognition benchmark: A multi-class classification competition. pages 1453 – 1460.

Stephen, O., Sain, M., Maduh, U. J., and Jeong, D.-U. (2019). An efficient deep learning approach to pneumonia classification in healthcare. *Journal of healthcare engineering*, 2019.

Sutton, R. S. and Barto, A. G. (1998). *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition.

Syngenta (2019). Syngenta Crop Challenge In Analytics. `https://www.ideaconnection.com/syngenta-crop-challenge/challenge.php`. [Online; accessed 16-January-2020].

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826.

Taghavi, S., Esmaeilzadeh, M., Najafi, M., Brown, T., and Borevitz, J. (2018). Deep phenotyping: Deep learning for temporal phenotype/genotype classification. *Plant Methods*, 14.

Tao, A., Barker, J., and Sarathy, S. (2016). Detectnet: Deep neural network for object detection in digits. *Parallel Forall*, 4.

Tetila, E. C., Machado, B. B., Menezes, G. V., de Souza Belete, N. A., Astolfi, G., and Pistori, H. (2019). A deep-learning approach for automatic counting of soybean insect pests. *IEEE Geoscience and Remote Sensing Letters*.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., and Lang, K. J. (1990). *Phoneme Recognition Using Time-Delay Neural Networks*, page 393–404. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Werbos, P. J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University.

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1.

Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K. (2017). Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500.

Yu, J., Sharpe, S. M., Schumann, A. W., and Boyd, N. S. (2019). Deep learning for image-based weed detection in turfgrass. *European journal of agronomy*, 104:78–84.

Zagoruyko, S. and Komodakis, N. (2017). Wide residual networks.

Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. (2020). *Dive into Deep Learning*. `https://d2l.ai`.

Zhang, Z. (2019). Deep learning for field-based automated high-throughput plant phenotyping. *Graduate Theses and Dissertations*, 17628.