Deep Learning and its Applications

Tom Wright
2071952

The rise of powerful AI will be either the best or the worst thing ever to happen to humanity. We do not yet know which. —Stephen Hawking

1 Introduction

Deep learning is a family of statistical techniques, centering around the concept of the **artificial neural network**. Neural networks are functions of real values, and are defined as compositions of a series of simpler functions, known as **layers**. These layers are themselves compositions of many smaller **units**. With a large enough number of such compositions, neural networks can exhibit incredibly complex behaviour [1][2].

Deep learning is a type of machine learning (ML), which itself is a class of artificial intelligence (AI). ML encompasses a range of tools, techniques, and algorithms that allow a computer to perform a variety of complex tasks typically carried out by humans (e.g. driving a car) without being explicitly programmed to do so. Machine learning algorithms "train" the computer using a set of **training examples**. Each example comprises a vector of inputs $X = (X_1, \ldots, X_d)^{\mathrm{T}}$, and a **response** Y that we want the computer to output given input X. For example, given a vector of radar data from a driver-less car, we may wish to output a suitable movement of the car's steering wheel, breaks, and accelerator [3][4][2].

The response Y is influenced by an uncountable number of abstract factors of variation (e.g. the position and speed of other vehicles and pedestrians, in the case of a driver-less car, but also the weather and lighting conditions), known as **features**, which are encoded in the unprocessed observed data. Traditional ML techniques require a well-trained engineer to choose a suitable **representation** of the input data. That is, the engineer must identify suitable features to use as inputs, and must extract these manually from the raw data using pre-processing. Her exact choice of representation may significantly affect the performance of her ML model. For example, a linear model can easily partition the two-class data set shown in Figure 1a if the data is represented using polar coordinates (left), but not when it is represented in Cartesian coordinates (right).

Deep neural networks can automatically learn a useful representation from the raw data; they are examples of **representation learning**. In fact, each layer in the network encodes a representation, based on that of the previous layer. In this way, complex representations are built up hierarchically across layers (e.g. edges, then shapes, then body parts, then pedestrians). The more units and connections the network has, the more complex the representations it can learn [5].

Deep learning has developed in three distinct phases [1]. Early research began in the 1940s, and was characterised by attempts to replicate the human brain. These researchers were inspired by classical associationism[1], and by advances in psychology and neuroscience. Donald Hebb's famous principle of Hebbian learning ("cells that fire together, wire together") laid the foundations for the first primitive network: the perceptron, invented in 1943 by McCulloch and Pitts [7]. The perceptron was based on a simple linear model and was implemented in 1958 by Frank Rosenblatt

[1]The idea that experiences are composed of associations between simpler experiences which themselves are associations of sensations. This was described by Aristotle in 300BC, and developed by the likes of Thomas Hobbes and John Locke [6].

in the form of an enormous electronic machine with potentiometers to program individual weights [8]. (We study the perceptron in detail in Section 3.)

These early networks were limited in their use, in part because their complexity was constrained by the technology of the time. In particular, they could not be applied to non-linear problems such as the XOR problem (see Section 4). Research thus slowed in the late 1960s, but the field was revived by the introduction of non-linearity, distributed learning, and the backpropagation algorithm for efficient gradient descent (see Section 6). As the field developed, it moved away from replicating biology in favour of models that could be efficiently stored and optimised by digital computers. Unfortunately, in the late 1980s, attention turned away from neural networks towards other machine learning methods (including logistic regression) that were showing more promise, bringing the second wave of research to an end [1, 6].

The current surge of research into deep learning was ignited in 2006 when Geoffrey Hinton introduced Deep Belief Networks and an efficient algorithm for training them [9][10]. At the same time, technological advancements were allowing researchers to collate vast sets of data and share them around the world. Increased computational power also meant that increasingly large network models could be trained with unprecedented speed [1]. These networks yielded high performance (even when **over-parameterised**, with many more parameters than training examples), and quickly began out-performing state-of-the-art techniques in image recognition, speech recognition, and other areas of machine learning. Several subsequent breakthroughs have allowed deep neural nets to achieve human-level performance in complex tasks. In 2016, a network called AlphaGo beat a professional player 5-0 in a game of Go - a feat unthinkable a few years earlier due to the complexity of the game [11]. Deep learning is now used extensively throughout the modern world, with applications in almost every field, from natural language processing (NLP) to science and medicine [5].

In this article, we will outline some foundational mathematical ideas in deep learning. We will begin by studying **fully-connected** (the units in each layer take every unit from the previous layer as input) neural networks applied to supervised learning problems. We will see how the perceptron can be extended to shallow (and then deep) neural networks that *can* represent the XOR problem. In Sections 6 and 7, we will introduce the work-horse of modern deep learning: the stochastic gradient descent optimisation algorithm with backpropagation. We will then apply these techniques to train a shallow network to solve the XOR problem from scratch. Finally, we will cover some specialised variants of FC deep networks and their applications.

2 Supervised Learning

What exactly do we mean by learning? Humans and animals learn; we accumulate knowledge and skills through repeated experiences. From a stream of raw sensory data, our brains extract useful concepts and relate these to past experiences to produce new ideas. Computers cannot (yet) learn in this way. Instead, machine learning algorithms attempt to find a function that approximates the relationship between an input and an output. This function may be highly complex and non-linear, and the inputs and outputs are often vectors with many entries.

We will focus on **supervised learning**[2] Here, we wish to map input vector $X \in \mathbb{R}^d$ to output $Y \in \mathcal{Y}$, using labelled training data $s = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$. The entries of X are known as input units, input features, or predictors, and Y is known as the response or **target**. We consider each pair $s^{(i)} = (x^{(i)}, y^{(i)})$ in the training set to be a realisation of the random variable pair

[2]Unsupervised learning, in contrast, is a class of techniques for finding patterns in unlabelled data.

$Z = (X, Y) \in \mathcal{Z} := \mathbb{R}^d \times \mathcal{Y}$. We assume that the distribution of Z can be described by

$$Y = f^*(X) + \epsilon,$$

where ϵ is some random deviation known as the **irreducible error**. The function $f^* : \mathbb{R}^d \to \mathcal{Y}$ defines the "true" distribution in the sense that it minimises ϵ over the set of measurable functions $\mathcal{M}(\mathbb{R}^d, \mathcal{Y})$ from $\mathbb{R}^d$ to $\mathcal{Y}$; it's the best possible function for describing Y in terms of X.[3]

The task of supervised learning is to use the training data s to determine a model $f : \mathbb{R}^d \to \mathcal{Y}$ that makes an accurate **prediction** $\hat{y} = f(x)$ of Y given input $X = x$. To do this we must identify three things [12].

1. A **loss function** $\mathcal{L}(f, s)$ that measures the performance of $f : \mathbb{R}^d \to \mathcal{Y}$ on a data point $s = (x, y) \in \mathcal{Z}$.

2. A **hypothesis set** $\mathcal{F}_m \subset \mathcal{M}(\mathbb{R}^d, \mathcal{Y})$ that restricts our search to a specific family functions $f \in \mathcal{F}_m$ with parameters θ. The subscript m usually denotes the number of such parameters, and indicates how **flexible** our approach will be, i.e. the complexity of the model f.

3. A **training algorithm** $\mathcal{A} : \bigcup_{n \in \mathbb{N}} \mathcal{Z}^n \to \mathcal{F}_m$ that outputs a trained model $f_s = \mathcal{A}(s)$ given a data set s of arbitrary size n. The algorithm typically determines model parameters by minimising a **cost** function C, defined as the average loss over all training examples, with

$$C(f, s) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f, s^{(i)}).$$

   We may write $\mathcal{L}_\theta$ or $\mathcal{L}(\theta)$ to indicate the dependency of the loss on model parameters θ.

Our primary goal is generalisation: we want the trained model f_s to be similar to f^*, so that it performs well on unseen test data. That is, we want to minimise **risk**, defined $\mathcal{R}(f_s) = \mathbb{E}[\mathcal{L}(f_s, Z)]$.

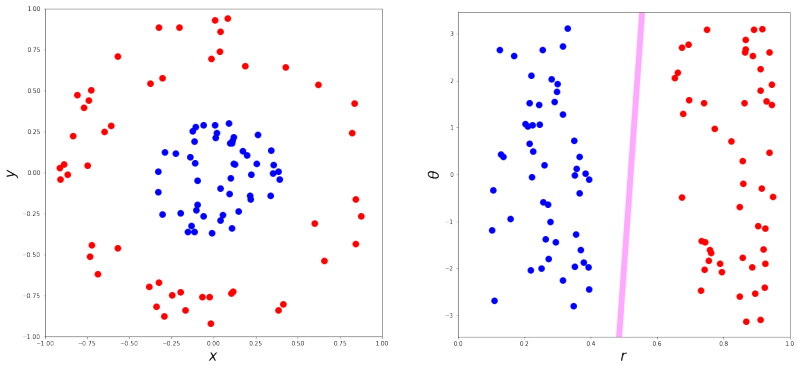
An example of a supervised learning task is simple linear regression. Here, $X, Y \in \mathbb{R}$, and we fit a linear model with two parameters with hypothesis set $\mathcal{F}_2 = \{f(x) = ax + b \mid a, b \in \mathbb{R}\}$. For the loss function we use the squared residual $\mathcal{L}(f, (x, y)) = (f(x) - y)^2$. We then determine the parameters a and b using the the least squares method;

$$f_s = \mathcal{A}(s) = \underset{f \in \mathcal{F}_2}{\arg\min} \sum_{i=1}^{n} \mathcal{L}\left(f, s^{(i)}\right) = \underset{f(x)=ax+b}{\arg\min} \sum_{i=1}^{n} \left(ax^{(i)} + b - y^{(i)}\right)^2.$$

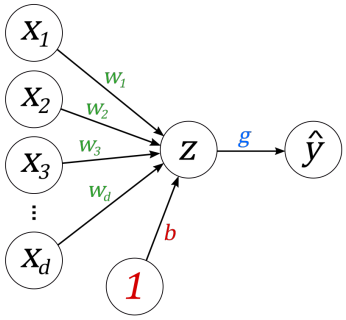
Instead of approximating a continuous output, we may wish to classify inputs into K discrete categories. For example given the pixels $X \in \mathbb{R}^{784}$ of a 28x28 greyscale image containing a single handwritten digit, we may want to predict the correct digit $Y \in \mathcal{Y} = \{1, 2, 3, \ldots, 9\}$. The MNIST data set contains 60,000 such images, and is commonly used for testing deep learning methods[13]. All learning tasks we consider will be either regression tasks ($\mathcal{Y} = \mathbb{R}^k$) or classification tasks ($\mathcal{Y} = \{1, 2, \ldots, K\}$). Classification with two classes $\mathcal{Y} = \{0, 1\}$ is known as **binary classification**.

The flexibility of our model (the size of $\mathcal{F}_m$) must be chosen carefully, due to the bias-variance trade-off. If our model is too flexible, it will fit too closely to the random noise ϵ, and will be highly dependent on the specific realisations of the data (high variance). This is known as **over-fitting**. If our model is too inflexible, it will fail to approximate the true relationship at all (high bias). This is known as **under-fitting**. Both of these issues cause an increase in the risk $\mathcal{R}(f_s)$. Through trial and error, we must fine tune our hypothesis set to find the sweet spot between these two extremes [2][14].

[3] Technically, f^* minimises *risk* over $\mathcal{M}(\mathbb{R}^d, \mathcal{Y})$ with respect to some loss $\mathcal{L}$.



(a) The same data is linearly separable in polar coords (left) but not in cartesian coords (right).



(b) A computational graph of the perceptron.

Figure 1

3 Shallow Networks

Neural networks solve the supervised learning task by restricting the hypothesis set to a composition of a large number simple functions. This composition is often represented using as a computational graph, such as that in Figure 1b. Each node represents an entry in a layer vector, and are known as units. (Each unit is analogous to a biological neuron, which is "firing" when the unit has a large numerical value.) A directed edge from node A to node B signifies that A was used as an input in the calculation of B via some function which may be indicated on the edge. We will study neural networks that can be represented by an acyclic directed graph. These are known as feed-forward, since information flows forwards from input layer to output layer[4].

The perceptron, introduced in Section 1, is a primitive neural network for binary classification; it maps input $x \in \mathbb{R}^p$ to one of two classes $y \in \{0, 1\}$. The perceptron first computes a weighted sum $z(x) = w^{\mathrm{T}}x + b$ of the inputs $x_1, \ldots, x_p$, with weights $w = (w_1, \ldots w_p)^{\mathrm{T}}$ and **bias** b. It then passes $z \in \mathbb{R}$ through a threshold function $g(z) = \mathbb{1}(z > \theta)$; this outputs class 1 if $z > \theta$, and class 0 otherwise. The overall model is thus

$$\hat{y} = f(x) = g(z(x)) = \begin{cases} 1 & \text{if } w^{\mathrm{T}}x + b > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Formally, the layers of this network are the two composed function g and z. Somewhat confusingly, we may also refer to individual vectors in the network as layers. We may refer to x as the input layer, $\hat{y}(x)$ as the output layer, and $z(x)$ as the hidden layer (since its value is not output directly).

The perceptron was a foundational step in the development of deep learning. However, its complexity was severely limited by the computational hardware of the time. As we will see in Section 4, the model is essentially linear, and can only represent linear distributions. To boost the power of the model (and expand its hypothesis set), we must make the function more complex.

To do this, we add a new layer h with N_1 units between the input layer and z, defined using a new intermediate variable $z^{(1)} = W^{(1)}x + b^{(1)}$ as

$$h(x) := \sigma(z^{(1)}) := \sigma\left(W^{(1)}x + b^{(1)}\right) \in \mathbb{R}^{N_1},$$

[4]A network which is not feed-forward is known as a recurrent neural network. Here, later units may be re-used in the computation of those in preceding layers (see Section 9).

where $W^{(1)} \in \mathbb{R}^{N_1 \times p}$ and $b^{(1)} \in \mathbb{R}^{N_1}$ are the **weight matrix** and **bias vector**, respectively, and $\sigma : \mathbb{R} \to \mathbb{R}$ is a non-linear **activation function** which is applied entry-wise to $z^{(1)}$. The vector $z^{(1)}$ is known as the vector of **pre-activations**. Each entry in $z_j^{(1)}$ is a distinct weighted sum of the input, with

$$z_j^{(1)} = b_j^{(1)} + \sum_{k=1}^{p} w_{jk}^{(1)} \cdot x_j,$$

where $w_{jk}^{(1)}$ and $b_j^{(1)}$ are the entries of $W^{(1)}$ and $b^{(1)}$. The entries $h_j = \sigma(z_j^{(1)})$ of $h(x)$ are known as **activations**.

Commonly used activation functions include the **sigmoid** function

$$\sigma(z) = \frac{e^z}{e^z + 1} = \frac{1}{1 + e^{-z}} \tag{3.1}$$

and the Rectified Linear Unit (**ReLU**) function

$$\sigma(z) = \max\{0, z\}. \tag{3.2}$$

As we will see in the next section, the non-linearity of the activation function σ is essential

We insert the new layer h into our network so that $\hat{y} = (g \circ z \circ h)(x) = \mathbb{1}(w^{\mathrm{T}} h + b > 0)$, as illustrated by Figure 4a. This can be extended further by replacing z with vector $z^{(2)} \in \mathbb{R}^{N_2}$, with

$$z^{(2)} = W^{(2)} h + b^{(2)} \quad \text{and} \quad W^{(2)} \in \mathbb{R}^{N_2 \times N_1}, b^{(2)} \in \mathbb{R}^{N_2},$$

so that each entry $z_j^{(2)}$ is a distinct weighted sum of the elements of h. We also allow g to be a general function $g : \mathbb{R}^{N_2} \to \mathcal{Y}$ defining the form of the output units. Our final network is given

$$\hat{y} = g\left(W^{(2)} \sigma\left(W^{(1)} x + b^{(1)}\right) + b^{(2)}\right)$$

Conventionally, the pre-activations $z^{(1)}$ and $z^{(2)}$ are omitted in network diagrams. Denoting $a = g \circ z^{(2)}$, we write $\hat{y} = a(h(x))$ and say that the network has two layers.

We can choose the function g to give a variety of different types of output unit. For regression, we use a linear output unit, with $\hat{y} = g(z^{(2)}) = z^{(2)} \in \mathbb{R}^{N_2}$. For binary classification, a common strategy is to output a probability $g(x) = \hat{P}(y = 1|x)$, and predict $\hat{y} = 1$ if $g > 0.5$. To do this, we use the sigmoid function $g(z) = e^z/(1 + e^z)$. Similarly, for multi-class classification with K classes, we associate a probability $g_i(x) = \hat{P}(y = i|x)$ to each class, and predict the class with the highest g_i. This is accomplished using the **softmax** function, defined

$$g_i(z) = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

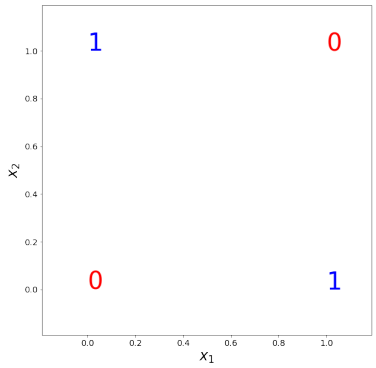
for $z \in \mathbb{R}^K$. It can be easily verified that sigmoid and softmax output units define a valid probability distribution (see Chapter 6 in [1]).

4 XOR and the need for non-linearity

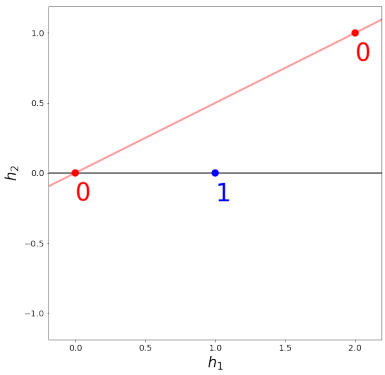
In the last section we introduced the perceptron model for binary classification. Similarly to the decision boundary in Figure 1a, it classifies $f(x) = 1$ if $w^{\mathrm{T}} x + b > 0$, and 0 otherwise, where the

(x_1, x_2)	$\text{XOR}(x_1, x_2)$
$(0, 0)$	0
$(1, 0)$	1
$(0, 1)$	1
$(1, 1)$	0

(a) The XOR data set, with $y = \text{XOR}(x_1, x_2)$.



(b) XOR is not linearly separable.



(c) But, it *is* under the transformation $h(x)$.

Figure 2

line $w^{\mathrm{T}}x+b = 0$ is called the **decision boundary**. A linear decision boundary is perfectly suitable if the two classes are **linearly separable**. Unfortunately, in practice this is quite rare.

The **exclusive or** logical operator, denoted XOR, is defined XOR(A,B) = (A and not(B)) or (B and not(A)). By treating 0 as False and 1 as True, we extract a set of data points. These are given in Figure 2.

We see from Figure 2b that the two classes (0 and 1) are not linearly separable; there is no linear decision boundary that can separate them. The perceptron therefore cannot correctly classify the data for *any* set of parameters. This fact was identified by Minsky and Papert in 1969 [15], and marked the end of the first wave of deep learning research.

Fortunately, our 2-layer neural network can correctly classify the XOR problem data, with suitable weights and biases. We use a network with two input units (x_1 and x_2), two hidden units (h_1 and h_2), and one output unit a. We use threshold function $g(z) = \mathbb{1}(z > 0)$ and activation function ReLU $\sigma(z) = \max\{0, z\}$, so that our overall network classifier is

$$\hat{y} = f(x) = (a \circ h)(x) = \begin{cases} 1 & \text{if } w^{\mathrm{T}}\sigma\left(W^{(1)}x + b^{(1)}\right) + b > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{4.1}$$

with parameters $W^{(1)} \in \mathbb{R}^{2\times 2}$, $b^{(1)} \in \mathbb{R}^2$, $w \in \mathbb{R}^2$, $b \in \mathbb{R}$. Figure 4a shows a similar model.

Note that the non-linear activation function here is critical. Without it, we would have

$$w^{\mathrm{T}}h + b = w^{\mathrm{T}}\left(W^{(1)}x + b^{(1)}\right) + b = (w^{\mathrm{T}}W^{(1)})x + (w^{\mathrm{T}}b^{(1)} + b) = \tilde{w}^{\mathrm{T}}x + \tilde{b},$$

which gives the perceptron model. Indeed, it can be shown that the layers of an arbitrarily large neural network will collapse to a single layer (a linear model) in the absence of a non-linear activation function [1].

We now select suitable network parameters, starting with $W^{(1)}$ and $b^{(1)}$. This amounts to constructing a non-linear transformation into a new "h-space". Figure 2c that with

$$W^{(1)} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad b^{(1)} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \tag{4.2}$$

our transformed data becomes linearly separable in the new space[5]. Now, we just need a linear

[5]At the address `https://www.geogebra.org/m/ygxdpj3j` is an interactive tool, custom made for this project. It demonstrates the transformation of the XOR data to h-space under different values for $W^{(1)}$ and $b^{(1)}$.

decision boundary in h-space. Let

$$w = \begin{pmatrix} 1 \\ -2 \end{pmatrix}, \quad b = 0. \tag{4.3}$$

Then, with decision boundary $w^{\mathrm{T}}h + b = 0$, we see from Figure 2c that

$$w^{\mathrm{T}}h(x^{(i)}) + b = 0 \implies f(x^{(i)}) = y^{(i)} = 0$$

for both data points $x^{(i)}$ labelled $y^{(i)} = 0$ (since both fall on the decision boundary), and

$$w^{\mathrm{T}}h(x^{(i)}) + b = \begin{pmatrix} 1 & -2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} + 0 = 1 > 0 \implies f(x^{(i)}) = y^{(i)} = 1$$

for both data points $x^{(i)}$ labelled $y^{(i)} = 1$ (since both are mapped to $(1, 0)^{\mathrm{T}}$ in h-space). Hence, our network with weights given in Equations 4.2 and 4.3 classifies all four points in the XOR training set correctly. This can be further verified evaluating 4.1 at each $x^{(i)}$. The solution to the XOR problem outlined here is given in Chapter 6.1 of [1].

In choosing the hidden layer parameters $W^{(1)}$ and $b^{(1)}$, we have defined a new representation: one in which the transformed data could be linearly separated by the second layer. In practice, we do not hand-pick parameter values. Instead, parameters are determined automatically through optimisation. In this way, a neural network automatically learns a sequence of increasingly complex hierarchical representations. As more layers are added (and more units added to those layers), increasingly complex representations can be learned [1].

5 Deep Networks

We now have everything we need to define a fully-connected feed-forward neural network with an arbitrary number of layers and units. To construct such a network, we simply add an arbitrary number of hidden layers, each with a specified number of units, and each taking input from the output of the previous layer. The following definition has been adapted from Chapter 1 of [12].

Definition 5.1 (FCFF Neural Network)**.** A fully-connected feed-foward (FCFF) neural network is defined by its architecture $\alpha = (N, \sigma, g)$, where

- $L \in \mathbb{N}$ is the number of layers in the network,
- $N = (N_0, N_1, \ldots, N_L) \in \mathbb{R}^{L+1}$ defines the number of units in each layer, where $d = N_0$ denotes the number of units in the input layer, and N_L the number of units in the output layer,
- $\sigma : \mathbb{R} \to \mathbb{R}$ is a non-linear activation function (ReLU, sigmoid, etc.), and
- $g : \mathbb{R} \to \mathbb{R}$ is a mapping that defines the units of the output layer.

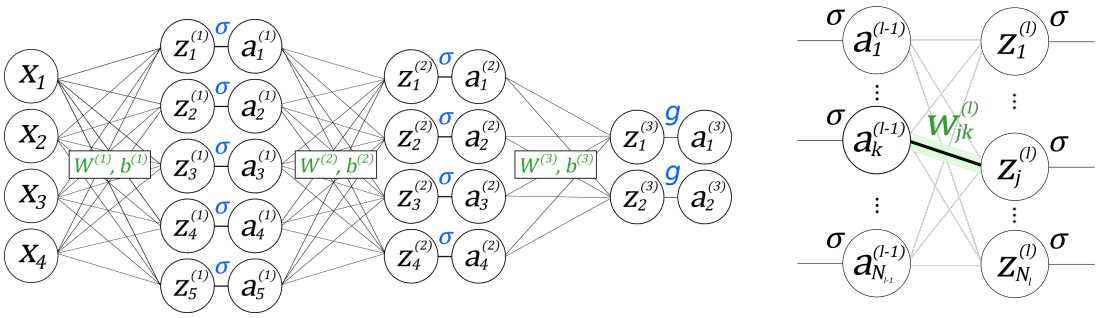
To each layer $l = 1, 2, \ldots, L$, we associate a vector of activations $a^{(l)} = (a_1^{(l)}, a_2^{(l)}, \ldots, a_{N_l}^{(l)})^{\mathrm{T}} \in \mathbb{R}^{N_l}$ and a vector of pre-activations $z^{(l)} = (z_1^{(l)}, z_2^{(l)}, \ldots, z_{N_l}^{(l)})^{\mathrm{T}} \in \mathbb{R}^{N_l}$. Given network parameters $\theta = \big(\big(W^{(l)}, b^{(l)}\big)\big)_{l=1}^{L}$, these vectors are related via the following equations:

$$z^{(1)}(x) = W^{(1)}x + b^{(1)}, \tag{NN1}$$

$$z^{(l)}(x) = W^{(l)}a^{(l-1)} + b^{(l)} \qquad \forall\ l = 2, 3, \ldots, L, \tag{NN2}$$

$$a^{(l)}(x) = \sigma(z^{(l)}) \qquad \forall\ l = 1, 2, \ldots, L-1, \tag{NN3}$$

$$\text{and} \quad a^{(L)}(x) = g(z^{(L)}). \tag{NN4}$$



(a) A network with architecture $((4, 5, 4, 2), \sigma, g)$. (b) The single weight $w_{jk}^{(l)}$.

Figure 3: The figure (a) was adapted from Figure 1.1 in [12].

The **realisation** of the network is the function $\Phi_{(N,\sigma,g)}(x;\theta) = a^{(L)}(x;\theta)$. For prediction $\hat{y}$, we may use Φ_α or some derived value thereof.

The weight matrix $W^{(l)} \in \mathbb{R}^{N_l \times N_{l-1}}$ in layer l has entries $w_{jk}^{(l)}$, where $w_{jk}^{(l)}$ gives the weight of $a_k^{(l-1)}$ in the calculation of unit $z_j^{(l)} = \mathbf{r}_j(W^{(l)})a^{(l-1)} + b_j^{(l)}$, as illustrated in Figure 3b. The entries of the bias vector $b^{(l)} \in \mathbb{R}^{N_l}$ give the bias $b_j^{(l)}$ applied to $z_j^{(l)}$. We may write $\theta \in \mathbb{R}^{P(N)}$, where $P(N) = \sum_{l=1}^{N} N_l(N_{l-1} + 1)$ is the total number of parameters[6].

Finally, we define the depth of the network to be L, and the width of the network to be $\|N\|_\infty$. We say that the network is shallow if $L = 2$, and deep if $L > 2$.

The hypothesis set for this model is the set $\mathcal{F}_{P(N)} = \left\{\Phi_{(N,\sigma,g)}(\,\cdot\,;\theta) : \theta \in \mathbb{R}^{P(N)}\right\}$ of network realisations over every possible parameter value. It may at first seem that this hypothesis set would be limited by the specific choices of architecture and activation. In fact, these networks are universal approximators under weak conditions on their architecture.

Theorem 5.1 (Universal Approximation Theorem)**.** Let $\Phi_{(N,\sigma,g)}$ be a neural network with $L \geq 2$ layers, a linear output layer $g(z) = z$, and an activation σ with "squashing" properties (e.g. the sigmoid function). Suppose that the feature $\mathbb{R}^d$ and response space $\mathcal{Y}$ are finite-dimensional. Then, with enough hidden units, the network can approximate any Borel-measurable function $f : \mathbb{R}^d \to \mathcal{Y}$ with arbitrary precision [12].

The set of Borel-measurable functions is a subset $\mathcal{M}(\mathcal{X}, \mathcal{Y})$, and contains every continuous function $f : S \subset \mathbb{R}^N \to \mathbb{R}$ defined on a compact (closed and bounded) set S. This theorem was initially proved for the sigmoid function only, but was later extended to a wider class of activation functions including ReLU [16].

In practice, the Universal Approximation Theorem implies that a large neural network will be able to represent *any* distribution of data, just as a shallow network could represent the XOR data. However, it has been shown that no single machine learning method can be superior over every data set (see *No Free Lunch Theorem* [12][2]). Indeed, there's no guarantee that our chosen training algorithm will be able to learn the distribution; it may be simply unable to find suitable parameters. Furthermore, Theorem 5.1 says nothing about exactly how large the network must be. For an extremely complex distribution, the required network size may be infeasibly large.

That said, a 2-layer neural network can in theory be fit with zero error to *any* training data, as long as the number of hidden units is greater than the number of data points and the functions

[6]Indeed, the set containing θ is isomorphic to $\mathbb{R}^{P(N)}$.

σ and g satisfy some conditions (see Theorem A.1, *Interpolation*). To prevent over-fitting, it is therefore common to apply **regularisation**. This means constraining the model in some way, to reduce its flexibility. One way of doing this is to penalize large weights by adding a penalty term to our cost function (known as weight decay), with

$$C_\theta(f, s) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_\theta(f, s^{(i)}) + \lambda \left\|\theta\right\|_2^2 .$$

Other regularisation methods including stopping training early, reducing connectivity [SEE CNNS], and drop-out learning (this involves, for each training example considered, removing a randomly-selected proportion of the units in each layer by setting their activations to zero). For more on regularisation, see Chapter 7 in [1].

6 Gradient Descent

In the previous section, we introduced our hypothesis set for neural networks. We now address the other two parts of learning tasks: the loss function $\mathcal{L}$ and training algorithm $\mathcal{A}$. The choice of loss function depends on the form of the output units. Often, these units define a distribution $\hat{Y}|X$; a common strategy is to maximise the likelihood of the parameters θ having given rise to data set s. In practice, this involves using the **negative log likelihood** of the data as the cost C to be minimised[7]. For example, if $Y \in \mathbb{R}^k$ is a vector with normally distributed noise, we recover the mean squared loss[8], with

$$C(f, (x, y)) = \frac{1}{2n} \sum_{i=1}^{n} \left\|y^{(i)} - f(x^{(i)})\right\|_2^2 ,$$

as shown in Chapter 6.2.1 of [1].

The cost C over data set s depends on the model evaluations $f(x^{(i)})$, which in turn depend on the parameters θ. Consequently, we can find the direction of steepest descent of the cost $C(\theta; s)$ with respect to θ. This is given by the negative gradient $-\nabla_\theta C(\theta; s)$. We can reduce the cost function iteratively by taking small steps in the direction of the this gradient, via a **gradient descent** step

$$\theta \leftarrow \theta - \eta \nabla_\theta C(\theta; s), \tag{6.1}$$

where $\eta > 0$ is known as the **learning rate** and determines the size of each step. If the step size is too large, we risk overshooting the minimum of cost the function, and may even increase its value. If the step size is too small, the cost will be reduced in very small increments, and the algorithm will take a long time to converge. The learning rate must be chosen empirically to balance these two considerations (see Chapter 4.3 in [1]).

If the cost function $C(\theta; s)$ is convex with respect to the parameters, gradient descent will converge to its global minimum. However, neural network cost functions are typically highly non-convex; gradient descent will therefore converge to some local minimum of $C(\theta; s)$ which may be highly sensitive to the initial parameters. In practice, initial weights are often set to random values

[7]This is equivalent to minimising the cross-entropy (a kind of dissimilarity) between the data distribution and the model distribution.

[8]Multiplying by the constant $1/2$ makes the partial derivatives of C slightly easier to work with, and does not affect the minimisation of C.

and initial biases to zero vectors. Gradient descent is typically run many times with different initial weights, to increase the likelihood of finding a good local minimum.

To execute (6.1), we must compute the loss $\mathcal{L}$ at every example in the training set s. This quickly becomes infeasible for large training data sets (which may contain millions or billions of individual observations). To rectify this, we approximate the gradient of the cost $C(\theta; s)$ using a randomly-selected **mini-batch** B of m training samples. The modified method is known as ***stochastic*** **gradient descent** (SGD). Although SGD requires more gradient descent steps, it drastically reduces the number of computations needed to evaluate the gradient.

7 Backpropagation

Stochastic gradient descent provides us with a way to train our neural networks, but we still need a method for calculating the gradient $\nabla_\theta C(\theta; B)$ over each mini-batch B. In particular, we need to compute the partial partial derivatives

$$\frac{\partial C}{\partial w_{jk}^{(l)}} \quad \text{and} \quad \frac{\partial C}{\partial b_j^{(l)}}. \tag{7.1}$$

We could attempt a direct calculation, or an approximation using finite differences, but for a large network this would involve evaluating the entire network - known as a **forward pass** - for every parameter[9].

Fortunately, research throughout the 1970s and 1980s produced an elegant method for computing such gradients: the backpropagation (backprop) algorithm [5]. This involves propagating changes in the output layer backwards through the network, using the multi-variate chain rule repeatedly. We follow the derivation presented in [4]. We assume the forward pass has already occurred so that all pre-activations $z^{(l)}$ and activations $a^{(l)}$ are known. Also, the cost C over a mini-batch B is the average of the losses $\mathcal{L}$, so by the linearity of the gradient operator, we have

$$\nabla_\theta C(\theta; B) = \nabla_\theta \frac{1}{m} \sum_{(x,y)\in B} \mathcal{L}(\theta; (x, y)) = \frac{1}{m} \sum_{(x,y)\in B} \nabla_\theta \mathcal{L}(\theta; (x, y)). \tag{7.2}$$

It therefore suffices first to find the gradient of the loss function $\mathcal{L}$ at a single example (x, y).

Firstly, we define a vector of **errors** $\delta^{(l)}$ for each layer $l = 1, \ldots, L$, with entries $\delta_j^{(l)} = \partial\mathcal{L}/\partial z_j^{(l)}$ for $j = 1, 2, \ldots, N_l$. We will use this error term to derive the four defining equations of the backpropagation algorithm. Equation (BP1) will allow us to calculate the error vector $\delta^{(L)}$ at the output layer. Equation (BP2) will give us $\delta^{(l)}$ in terms of $\delta^{(l+1)}$, allowing us to propagate the errors backwards through the network starting with $\delta^{(L)}$. Finally (BP3) and (BP4) will allow us to compute the gradient $\nabla_\theta \mathcal{L}(\theta; (x, y))$.

Firstly, observe that by definition, the entries $\delta^{(L)}$ at the output layer are

$$\delta_j^{(L)} = \frac{\partial \mathcal{L}}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \frac{\partial \mathcal{L}}{\partial a_j^{(L)}}\, g'\left(z_j^{(L)}\right) p \implies \delta^{(L)} = \nabla_{a^{(L)}} \mathcal{L}(\theta) \odot g'(z^{(L)}), \tag{BP1}$$

[9] For example, if $N = (80, 40, 20, 3)$ - a relatively small network - we would need to make at least $P(N) = 4123$ forward passes at every descent step!

where $\odot$ denotes the operation that multiplies corresponding entries of the two vectors (known as the Hadamard product). The derivative g' can be pre-programmed in any implementation, as can the partial derivatives[10] $\partial\mathcal{L}/\partial a_j^{(L)}$.

Now, for $l = 1, 2, \ldots, L-1$, we have that $z^{(l+1)} = W^{(l+1)}a^{(l)} + b^{(l+1)}$ in layer $l+1$, so

$$z_r^{(l+1)} = b_r^{(l+1)} + \sum_{j=1}^{N_l} w_{rj}^{(l+1)} a_j^{(l)} = b_r^{(l+1)} + \sum_{j=1}^{N_l} w_{rj}^{(l+1)} \sigma\left(z_j^{(l)}\right). \tag{7.3}$$

Thus, by the multi-variate chain rule,

$$\begin{aligned}
\delta_j^{(l)} = \frac{\partial\mathcal{L}}{\partial z_j^{(l)}} = \sum_{r=1}^{N_{l+1}} \left(\frac{\partial\mathcal{L}}{\partial z_r^{(l+1)}} \frac{\partial z_r^{(l+1)}}{\partial z_j^{(l)}}\right) &= \sum_{r=1}^{N_{l+1}} \delta_r^{(l+1)} \cdot w_{rj}^{(l+1)}\ \sigma'\left(z_j^{(l)}\right) \\
&= \left(\mathbf{col}_j\left(W^{(l+1)}\right)^{\mathrm{T}} \delta^{(l+1)}\right) \cdot \sigma'\left(z_j^{(l)}\right), \\
\implies \delta^{(l)} &= {W^{(l+1)}}^{\mathrm{T}} \delta^{(l+1)} \odot \sigma'\left(z^{(l)}\right),
\end{aligned} \tag{BP2}$$

where σ' is applied element-wise. The third equality here holds using (7.3) and the definition of $\delta_k^{(l+1)}$. Again, the derivative σ' can be pre-programmed. For example, if σ is the ReLU function, we have

$$\sigma(z) = \max\{0, z\} \quad \implies \quad \sigma'(z) = \begin{cases} 1: & z \geq 0; \\ 0: & z < 0, \end{cases}$$

where $\sigma'(0) = 1$ is chosen arbitarily.

Finally, we compute the gradient entries. As in (7.3), we have $z_j^{(l)} = b_j^{(l)} + \sum_{k=1}^{N_{l-1}} w_{jk}^{(l)} a_k^{(l-1)}$. Thus,

$$\frac{\partial\mathcal{L}}{\partial b_j^{(l)}} = \frac{\partial\mathcal{L}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \cdot 1 = \left[\delta^{(l)}\right]_j, \tag{BP3}$$

$$\text{and} \quad \frac{\partial\mathcal{L}}{\partial w_{jk}^{(l)}} = \frac{\partial\mathcal{L}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} \cdot a_k^{(l-1)} = \left[\delta^{(l)} {a^{(l-1)}}^{\mathrm{T}}\right]_{jk}. \tag{BP4}$$

We compute these values for each example $(x, y) \in B$, and average them over B according to (7.2) to obtain the cost gradient $\nabla_\theta C(\theta; B)$. A complete description of the SGD algorithm with backprop is given in Appendix B. This algorithm is the primary method for training modern deep networks [5].

8 Learning XOR using backprop

So far, we have defined a deep neural network and have introduced SGD with backprop for training it. We now demonstrate these concepts by training a network (from scratch) to solve the XOR problem seen in Section 4. The python code for this example can be found online at `https://github.com/tomnwright/ShallowXOR`. Inspiration was taken from [17].

[10] For example, if $\mathcal{L}(\theta; (x, y)) = \frac{1}{2}\sum_{j=1}^{N_L}\left(y_j - a_j^{(L)}(x)\right)^2$, we have $\partial\mathcal{L}/\partial a_j^{(L)} = a_j^{(L)}(x) - y_j$.

Firstly, we generate a data set of n=120 points, by applying some normally distributed random error to the XOR data from Section 4. This new data is plotted in Figure 4b. We fit a two-layer network that outputs a probability $g(x) = \hat{P}(y = 1|x)$. We classify $\hat{y} = 1$ if $g > 0.5$. This network has the same structure as the network described by Equation (4.1) and illustrated in Figure 4a, but with sigmoid output g and sigmoid activation σ (see Equation 3.1). It has parameters $W^{(1)} \in \mathbb{R}^{2\times 2},\ b^{(1)} \in \mathbb{R}^2,\ w \in \mathbb{R}^2,\ b \in \mathbb{R}$, and realisation

$$\Phi_{((2,2,1),\sigma,g)}(x) = a^{(2)}(x) = g(z^{(2)}(x)) = g(w^{\mathrm{T}}a^{(1)}(x) + b), \quad \text{with} \tag{8.1}$$

$$a^{(1)}(x) = \sigma(z^{(1)}(x)) = \sigma(W^{(1)}x + b^{(1)}). \tag{8.2}$$

For the loss function at point (x, y), we use binary cross-entropy loss[11]

$$\mathcal{L}(\theta) = \zeta\left((1-2y)z^{(2)}\right),$$

written in terms of pre-activation $z^{(2)} \in \mathbb{R}$. Here, ζ is the **softplus function** $\zeta(t) = \log(1+\exp(t))$, which has the useful property that $\zeta'(t) = \sigma(t)$.

To train the network, we first initialise the biases to zeros and the weights to have random values between –0.5 and 0.5. We train the network using SGD with a mini-batch size of $m = 1$, and a learning rate of $\eta = 0.1$ (chosen empirically to give reasonably fast convergence). At each step of gradient descent, we randomly pick a single point $(x^{(i)}, y^{(i)})$ from the data set, apply backprop to compute the gradient, and update our parameters accordingly.

To compute the gradient using backprop, we first compute the forward pass; we use Equations 8.1 and 8.2 to determine $z^{(1)}$, $a^{(1)}$, $z^{(2)}$, and $a^{(2)}$. Next, we compute the backpropagation errors $\delta^{(1)} \in \mathbb{R}^2$ and $\delta^{(2)} \in \mathbb{R}$ for the 2 layers. By the explicit definition of $\delta^{(l)}$, we have

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} = \zeta'\left((1-2y)z^{(2)}\right)(1-2y) = \sigma\left((1-2y)z^{(2)}\right)(1-2y).$$

Then, by (BP2), we have

$$\delta^{(1)} = \delta^{(2)}w \odot \sigma'\left(z^{(1)}\right).$$

We perform this step using the derivative $\sigma'(z) = \sigma(z)(1-\sigma(z))$. Finally, we update the parameters via gradient descent, using (BP3) and (BP4), and with a learning rate of $\eta = 0.1$ (chosen empirically). We perform

$$w \leftarrow w - \eta(\delta^{(2)}a^{(1)}), \qquad b \leftarrow b - \eta\delta^{(2)},$$

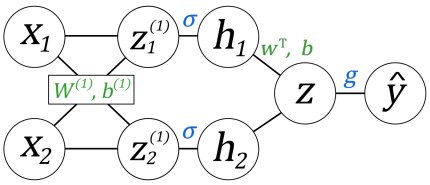
$$W^{(1)} \leftarrow W^{(1)} - \eta(\delta^{(1)}x^{\mathrm{T}}), \qquad \text{and } b^{(1)} \leftarrow b^{(1)} - \eta\delta^{(1)}.$$

We see from Figure 4b that our algorithm can indeed learn to classify the data successfully, although Figure 4c shows that sometimes (depending on the initial random weights), the gradient descent finds a bad local minimum and the cost plateaus at a bad solution.

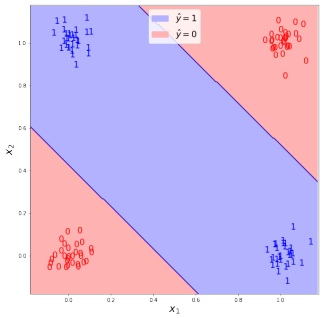
9 Specialised Networks

Many different types of specialised network architectures have been developed that perform well in certain circumstances. Convolutional neural networks (CNNs) are one such architecture, and are

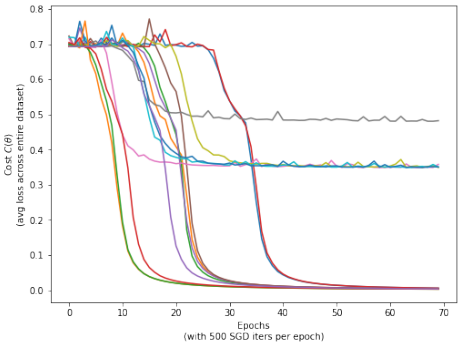
[11] The binary cross-entropy loss is derived as the negative log-likelihood for the distribution defined by the sigmoid output unit. It can also be defined in terms of output $\hat{y} = a$, but the above form is more suitable for our purposes [1].



(a) A simple network with a single hidden layer, as described in (4.1).



(b) Our trained model classifies all points correctly.



(c) Evolution of cost during many training runs (failure rate exaggerated).

Figure 4

designed classification tasks with inputs that take a grid-like form (e.g. 2D images, 1D time-series data) [2]. A CNN uses a unique convolution layer, with three phases: first, **convolution** (this detects parts of the image that resemble a parameter called a filter), then **detection** (normally a ReLU layer), and finally **pooling** (this reduces the size of the grid by computing a statistic - e.g. maximum or average - in each area of the image)[12]. A CNN normally has many convolution layers in sequence, followed by several fully-connected layers, and finally a softmax function for classification. The convolution layers differ from those we have studied in that they are sparsely connected (each unit is affected by only some of the previous layer's inputs)[1]citeislr.

Convolutional neural networks have many interesting applications, including in medical imaging analysis. For example, in [18], researchers trained a CNN to diagnose lung pathologies using a training set of 35,038 chest x-rays (inputs) and final reports (labels). They achieved a sensitivity (rate of true positives, i.e. the rate of correct diagnosis on scans of people who actually had the disease) and specificity (rate of true negatives) of 82% for pulmonary edema (build-up of fluid inside the lungs), 91% for pleural effusion (fluid build-up in the surrounding tissues of the lungs), and 81% for cardiomegaly (enlargement of the heart).

Another important type of specialised network is the recurrent neural network (RNN), which can be trained on sequential data such as a list of words or weather telemetry. An RNN essentially trains a sequence of networks, with each network taking as input an entry from the input sequence and the activations of the previous network. In this way, information is cycled through the network (i.e. the network is not feed-forward). The model learns weights for each network, as well as a shared hidden state that passes activations between layers [1][2].

10 Conclusion

In this report we've covered several fundamental concepts in deep learning. We've seen how to construct deep FC networks for supervised learning, and have demonstrated how these networks can be efficiently optimised using the stochastic gradient descent algorithm with backpropagation. However, we have only scratched the surface of the field; deep learning is vast, and ever-changing. For brevity, we have glossed over many nuances, including complications that have been discovered and addressed (see the Vanishing Gradient Problem [3]).

[12]Importantly, the output of the pooling phase is location invariant; it is insensitive to small changes in the location of structures in the image.

Still, the field of deep learning is still relatively new, with many areas being researched actively. Perhaps the area with greatest potential growth is in unsupervised learning[5]. Deep learning techniques have been developed primarily for supervised learning. Yet, human learning is mostly unsupervised. We often explore the world around us without a specific goal, driven by curiosity; we play. A similar area of research is known as active learning, where the deep network algorithm actively highlights examples on which it performs badly and requires human intervention[3]. This can speed up learning drastically, by avoiding training the network on millions of samples it already performs well on. Finally, as the field of neuroscience develops, new insights into the human brain may prove useful for developing more sophisticated AI[1].

Over the next two decades, AI systems will continue to improve exponentially. In [19], the authors predicts that we will see an AI revolution, with more significant impacts than the Industrial revolution and digital revolution combined. Such a revolution would offer many potential benefits almost universally, including AI-driven scientific discovery and the automation of dangerous professions like mining and fire-fighting. However, there are concerns that such automation could have terrible consequences. Even if new professions are created, many mid-range jobs (e.g. office workers and administrators) could be destroyed, increasing the wealth gap between the richest and poorest people and countries. Human-level AI may decrease the number of employees needed to run even large companies, causing wealth to be increasingly concentrated with the most powerful [19]. These ethical issues are currently being researched extensively.

Further Reading

For interested readers, the textbooks [2], [4], and [1] (listed in order of increasing complexity and detail) are excellent sources of information on a wide number of topics in deep learning.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[2] Gareth James et al. *An introduction to statistical learning: With applications in R*. Springer, 2022.

[3] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

[4] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015.

[5] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (May 2015), pp. 436–444. DOI: `10.1038/nature14539`. URL: `https://doi.org/10.1038/nature14539`.

[6] Haohan Wang and Bhiksha Raj. *On the Origin of Deep Learning*. 2017. DOI: `10.48550/ARXIV.1702.07800`. URL: `https://arxiv.org/abs/1702.07800`.

[7] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[8] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[9] Geoffrey E Hinton and Ruslan R Salakhutdinov. "Reducing the dimensionality of data with neural networks". In: *science* 313.5786 (2006), pp. 504–507.

[10] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. "A fast learning algorithm for deep belief nets". In: *Neural computation* 18.7 (2006), pp. 1527–1554.

[11] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[12] Philipp Grohs and Gitta Kutyniok, eds. *Mathematical Aspects of Deep Learning*. Cambridge University Press, 2022. DOI: 10.1017/9781009025096.

[13] Li Deng. "The mnist database of handwritten digit images for machine learning research [best of the web]". In: *IEEE signal processing magazine* 29.6 (2012), pp. 141–142.

[14] Andriy Burkov. *The hundred-page machine learning book*. Andriy Burkov, 2019. URL: https://themlbook.com/.

[15] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 1969.

[16] Weinan E et al. *Towards a Mathematical Understanding of Neural Network-Based Machine Learning: what we know and what we don't*. 2020. DOI: 10.48550/ARXIV.2009.10713. URL: https://arxiv.org/abs/2009.10713.

[17] Aniruddha Karajgi. *How Neural Networks Solve the XOR Problem*. Nov. 2020. URL: https://towardsdatascience.com/how-neural-networks-solve-the-xor-problem-59763136bdd7 (visited on 05/02/2023).

[18] Mark Cicero et al. "Training and validating a deep convolutional neural network for computer-aided detection and classification of abnormalities on frontal chest radiographs". In: *Investigative radiology* 52.5 (2017), pp. 281–287.

[19] Spyros Makridakis. "The forthcoming Artificial Intelligence (AI) revolution: Its impact on society and firms". In: *Futures* 90 (2017), pp. 46–60. DOI: https://doi.org/10.1016/j.futures.2017.03.006.

[20] Catherine F. Higham and Desmond J. Higham. "Deep Learning: An Introduction for Applied Mathematicians". In: *SIAM Review* 61.4 (2019), pp. 860–891. DOI: 10.1137/18M1165748. eprint: https://doi.org/10.1137/18M1165748. URL: https://doi.org/10.1137/18M1165748.

[21] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

A Interpolation

Theorem A.1 (Interpolation)**.** We have the following.

- Let $s = \left(z^{(i)}\right)_{i=1}^{m}$ be the training data set, with $z^{(i)} = (x^{(i)}, y^{(i)}) \in \mathbb{R}^d \times \mathbb{R}^k$.
- Let $\sigma \in C(\mathbb{R})$ be a continuous activation function that is not a polynomial.
- Let the output unit be linear; $g(z) = z$.

Then there exists first layer parameters $\theta^{(1)} = (W^{(1)}, b^{(1)}) \in \mathbb{R}^{m\times d} \times \mathbb{R}^m$ such that for any labels $y^{(i)}$, there exists second layer parameters $\theta^{(2)} = (W^{(2)}, b^{(2)}) \in \mathbb{R}^{k\times m} \times \mathbb{R}^k$ such that the network with architecture $\alpha = ((d, m, k), \sigma, g)$ and parameters $\theta = (\theta^{(1)}, \theta^{(2)})$ interpolates the dataset s. That is,

$$\Phi_\alpha(x^{(i)}; \theta) = y^{(i)} \quad \forall\ i = 1, 2, \ldots, m.$$

Theorem A.1 is stated with a sketch proof in [12].

B Further notes on backpropagation

The full stochastic gradient descent (SGD) algorithm with backpropagation is stated in Algorithm 1. For more information, see [4].

Algorithm 1 SGD with Backprop

for SGD iterations $i = 1, 2, \ldots, i_{\max}$ **do**

Select mini-batch B.
for $(x, y) \in B$ **do**

First, compute forwards pass for input x.
$z^{(1),x} \leftarrow W^{(1)}x + b^{(1)}$
for $l = 1, 2, \ldots, L-1$ **do**
$a^{(l),x} \leftarrow \sigma(z^{(l),x})$
$z^{(l+1),x} \leftarrow W^{(l+1)}a^{(l),x} + b^{(l+1)}$
end for
$a^{(L),x} \leftarrow g(z^{(L),x})$

Now, compute the backward pass.
$\delta^{(L),x} \leftarrow \nabla_{a^{(L),x}} \mathcal{L}(\Phi, (x, y)) \ \odot\ g'(z^{(L),x})$ ▷ (BP1)
for $r = 1, 2, \ldots, L-1$ **do**
$l = L - r$
$\delta^{(l),x} \leftarrow {W^{(l+1)}}^{\mathrm{T}} \delta^{(l+1),x} \odot \sigma'\left(z^{(l),x}\right)$ ▷ (BP2)
end for

end for

Update the weights using cost gradient.
for $l = 1, 2, \ldots, L$ **do**
$b^{(l)} \leftarrow b^{(l)} - \eta \sum_{(x,y)\in B} \delta^{(l),x}$ ▷ (BP3)
$W^{(l)} \leftarrow W^{(l)} - \eta \sum_{(x,y)\in B} \delta^{(l),x} {a^{(l-1),x}}^{\mathrm{T}}$ ▷ (BP4)
end for
end for

C Mysteries

For most of the history of deep learning, deep networks were designed intuitively based on empirical knowledge and their incredible properties were mathematically mysterious. Recently, new research has attempted to find answer to the following questions

[12, 16, 20]. *Why do large networks not over-fit?* Even when highly over-parameterised, and trained on noisy data, neural networks over-fit far less than would be otherwise expected (see *Double Descent*, [2]).

Why do neural networks not suffer from the Curse of Dimensionality? The performance of typical machine learning algorithms falls exponentially as the dimension of the feature space X increases. Neural networks perform exceedingly well on such tasks, and do not seem to suffer from increased dimension.

What is the role of depth? We have seen that shallow networks of sufficient width can be extremely powerful. However, empirically evidence suggests deeper networks exhibit superior generalisation power.

Why does stochastic gradient descent converge to a good local minimum despite the non-convexity of the parameter space? (See Section 6).