NEURAL NETWORKS FOR CLASSIFICATION

Owen Jones
[olj23@bath.ac.uk]
University of Bath

May 2019

ABSTRACT

Classification tasks continue to represent a large proportion of modern applications of machine learning, and the excitement generated by "deep learning" and similar topics in recent years has led to the resurgence of the artificial neural network as an adaptable and powerful classification model.

In this work we introduce the concept of binary and multiclass classification tasks and how they might be formulated mathematically. Two popular classification models, logistic regression and neural networks, are then discussed in some detail and the theory behind their use is developed and explored. An implementation of these models is demonstrated on a variety of example tasks, and we conclude by highlighting some possible topics for further exploration.

1 Introduction to Classification Tasks

Most real-world problems have an answer which falls into one of two categories: an *amount*, or a *choice* between two or more options.

The first type of question, where the answer is generally a number on some continuous scale, are known as *regression* problems. The second type, where the answer is some discrete label or category, are known as *classification* problems; and this second type of question will be the focus of this paper.

Expanding slightly on the definition, a classification task consists of observing some set of inputs (e.g. "Do I want to be on the other side of the road? How fast is that car approaching? How tasty is the chocolate I am eating?"). The influence and relative importance of each input is assessed, and the combination of these assessments is used to assign a label to that observation (e.g. "I *shouldn't* cross the road right now."). For every task which comes with such a set of inputs, in order to consistently make good decisions the influences and relative importances of each input must be either previously known, prescribed by principle, or learned.

The learning process usually falls under one of two approaches. *Unsupervised learning* results in decisions being made based on commonalities in the inputs of multiple observations (e.g. "There are lots of other people on that side of the road, so I'll cross too and see what's going on."). On the other hand, *supervised learning* uses the decisions which were made based on previous observations to inform the making of the current decision (e.g. "In the past, whenever a car was approaching as quickly as this, I didn't cross; and it didn't seem to matter whether or not the chocolate I was eating was tasty; so I *shouldn't* cross right now.").

We will focus on the latter approach in this paper. Mathematically, such problems can be formulated as follows: we wish to define a model f characterised by some set of parameters $\boldsymbol{\theta}$, which maps a given set of inputs $\mathbf{x_i} \in \mathbb{R}^n$ as closely as possible to an associated output $y_i \in \mathcal{C} \subset \mathbb{Z}$:

$$f \colon \mathbb{R}^n \to \mathcal{C}$$

$$f(\mathbf{x_i} \mid \boldsymbol{\theta}) \approx y_i$$

The following sections provide an overview of two popular supervised classification models: *logistic regression*, and the *artificial neural network*.

2 Logistic Regression

Consider any question where the answer is either "yes" or "no". Will the next train arrive on time? Does this image contain a face? Should we prescribe this patient Drug B instead of Drug A? In this way, many problems in the real world can be reduced in essence to a *binary classification* task, where the "correct" answer belongs to one of two distinct categories.

Therefore if we wish to produce a model capable of performing classification tasks, a useful starting point would be a binary classifier: that is, we would like to produce a model f which, given some observation $\mathbf{x}$ with an associated label $y \in \{0,\ 1\}$, produces a result such that

$$f(\mathbf{x} \mid \boldsymbol{\theta}) \approx y\,.$$

This section outlines the motivation behind and the definition of one of the best-known binary classifiers, the *logistic regression* model.

2.1 Linear Regression

A reasonable assumption when beginning to define a model is that the class of the observation is somehow related to its features - if that is not the case, then we will struggle to extract any meaningful information from the data available to us!

Therefore suppose we have an observation $\mathbf{x}$ consisting of $n \geq 1$ inputs:

$$\mathbf{x} = (x_1,\ x_2,\ \ldots,\ x_{n-1},\ x_n) \in \mathbb{R}^n\,.$$

Let us define a set of weights $\boldsymbol{\theta} = (\theta_0, \ldots, \theta_n) \in \mathbb{R}^{n+1}$ which in combination with $\mathbf{x}$ produce a scalar output

$$z = \theta_0 + \sum_{k=1}^{n} \theta_k x_k\,.$$

Notice the additional *bias* weight θ_0, which corresponds to a shift in the output independent of any of the inputs. To simplify notation later on, we can redefine $\mathbf{x} \in \mathbb{R}^{n+1}$ by augmenting with $x_0 = 1$, which allows us to write z as a single summation:

$$z = \sum_{k=0}^{n} \theta_k x_k = \boldsymbol{\theta}^\top \mathbf{x}\,.$$

By adjusting the weights $\theta_0, \ldots, \theta_n$, the same inputs $\mathbf{x}$ can give us a completely different output. Note that if we do not restrict $\boldsymbol{\theta}$ then z is unbounded, i.e. $z \in (-\infty, \infty)$. This is exactly the formulation of a *linear regression* model (a *linear model* in statistical terms), and this is what lends the name "regression" to the logistic regression model, as we shall see shortly.

2.2 Introducing Non-Linearity

Recall that the goal of a binary classification task is to assign a categorical label to each observation; so if we wish to use linear regression as the basis for our algorithm, ultimately we will need to somehow interpret the continuous output $z \in \mathbb{R}$ as corresponding to one of those discrete labels.

This can be achieved by means of a non-linear *activation function*:

$$\sigma\colon (-\infty, \infty) \to (0, 1)\,.$$

In later sections we will consider other activation functions, but for logistic regression we typically use the *sigmoid* function,

$$\sigma_2(z) = \frac{1}{1 + e^{-z}}\,.$$

Note how large positive values of z are approximately mapped to 1, large negative values are approximately mapped to 0, and the transition between the two asymptotic limits is relatively sudden, yet smooth, with the inflection point at $z = 0$.

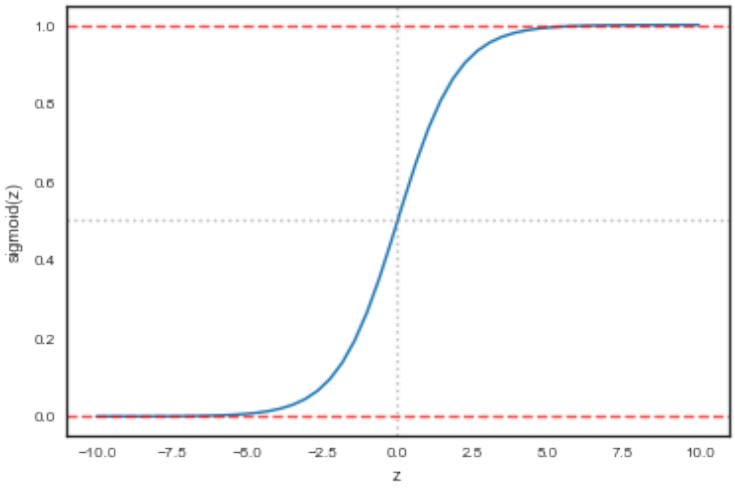

Figure 1: The sigmoid function over the domain $-10 \leq z \leq 10$.

In order to extract quantitative predictions from the model, we can simply set some threshold - typically 0.5 - and assign an integer label to the observation according to which side of the threshold $\sigma_2(z)$ falls. This *decision rule* is what transforms logistic regression from a regression to a classification model.

Hence our input has now been mapped to a linear output, and then transformed via the non-linear sigmoid function. This is the basis of the *logistic regression* model: in statistical terms, this is a *generalised linear model* with a *logit* (inverse sigmoid) link function.

In the remainder of this paper, the activated output $\sigma(z)$ is sometimes denoted as a for the sake of notational simplicity.

2.3 Quantifying Incorrectness

Now that we have a means of mapping an input $\mathbf{x}$ to an activated output $a \in (0, 1)$, we can begin to think about how to adjust the weights $\boldsymbol{\theta}$ so that a is as close to the target output y as possible.

In order to assess how well the model is performing, we shall require some sort of "penalty value" based on the dissimilarity between a and y. For example, the *binary cross-entropy* is defined as follows:

$$CE(a, y \mid \boldsymbol{\theta}) = \begin{cases} -\log(a) & y = 1 \\ -\log(1-a) & y = 0 \end{cases} .$$

Note that CE is a function of $\boldsymbol{\theta}$ (since a is a function of z, which is dependent on $\boldsymbol{\theta}$), that it takes a value of 0 if $a = y$, and that it increases exponentially as a becomes further from y.

So far we have considered a single observation $\mathbf{x}$ with a label y, which in combination with $\boldsymbol{\theta}$ produces a single activated output a. However in reality, we have multiple observations (say $m \geq 1$) each with an associated label and each producing an activated output in combination with $\boldsymbol{\theta}$. Therefore when assessing how well $\boldsymbol{\theta}$ parameterises our entire set of observations, we need to consider the cost of each of those observations.

For that purpose, we generally consider some *cost function* $J(\boldsymbol{\theta})$ which combines the individual costs from each observation. Such a function effectively quantifies how "wrong" the model is, so when adjusting $\boldsymbol{\theta}$ we will be aiming to minimise $J(\boldsymbol{\theta})$. A typical choice for binary classification problems is the *binary cross-entropy cost function*:

$$J_2(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} CE(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^{m} \left(y_i \log(a_i) + (1 - y_i) \log(1 - a_i) \right) .$$

Since $y_i \in \{0, 1\}$, at most one of the terms in the body of the summation is non-zero for each i; and so $J_2(\boldsymbol{\theta})$ is simply the average cross-entropy across all m observations in our dataset.

2.4 Training

As described at the end of Section 2.3, the goal of the *training* process is to minimise the cost function $J(\boldsymbol{\theta})$ by making changes to the parameters $\boldsymbol{\theta}$: in other words, we essentially have an optimisation problem, which we can tackle in countless different ways.

Here we will focus on *gradient descent*, an iterative method which takes successive steps converging towards a minimum optimal value. Although many other more sophisticated optimisation algorithms exist, gradient descent is one of the most commonly used optimisation methods due to its simplicity yet relative effectiveness: see [2] for further details.

2.4.1 Descent Direction

Since we wish to reduce $J(\boldsymbol{\theta})$ by taking some step in parameter space, before doing anything else we must determine the direction in which to take that step.

Consider a perturbation $\Delta\boldsymbol{\theta}$ to our current parameters $\boldsymbol{\theta} \in \mathbb{R}^{n+1}$, and notice that a Taylor expansion yields

$$J(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \sum_{k=0}^{n+1} \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_k} (\Delta\boldsymbol{\theta})_k + \mathcal{O}((\Delta\boldsymbol{\theta})^2) \,.$$

Then ignoring higher-order terms, we have

$$J(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}) + \nabla J(\boldsymbol{\theta})^\top \Delta\boldsymbol{\theta} \,.$$

Now we greedily choose $\Delta\boldsymbol{\theta}$ such that we reduce $J(\boldsymbol{\theta})$ by as much as possible - i.e. by making $\nabla J(\boldsymbol{\theta})^\top \Delta\boldsymbol{\theta}$ as negative as possible. By the Cauchy-Schwarz inequality, the most negative value of $\mathbf{f}^\top \mathbf{g}$ for any $\mathbf{f}, \mathbf{g} \in \mathbb{R}^{n+1}$ is $-\|\mathbf{f}\|_2 \|\mathbf{g}\|_2$, when $\mathbf{f} = -\mathbf{g}$. Therefore we shall choose $\Delta\boldsymbol{\theta} = -\nabla J(\boldsymbol{\theta})$ as our *descent direction*. Note that $\nabla J(\boldsymbol{\theta})^\top (-\nabla J(\boldsymbol{\theta})) = -\|\nabla J(\boldsymbol{\theta})\|_2^2 \leq 0$ with equality if and only if $\nabla J(\boldsymbol{\theta}) = 0$, and consequently $J(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) < J(\boldsymbol{\theta})$ so long as $\nabla J(\boldsymbol{\theta}) \neq 0$.

2.4.2 Learning rate

While it is guaranteed that the "steepest descent" direction we have just chosen, $\Delta\boldsymbol{\theta} = -\nabla J(\boldsymbol{\theta})$, is indeed a descent direction, we must recall at this point that we are using a Taylor series approximation for $J(\boldsymbol{\theta} + \Delta\boldsymbol{\theta})$. Therefore this guarantee only applies in some (potentially very small) radius of the parameter space, centred on $\boldsymbol{\theta}$.

Consequently in order to ensure that we obtain a reduction in J, we must scale $\Delta\boldsymbol{\theta}$ by some value η. This value is called the *learning rate*.

Selecting a learning rate is often a delicate task. Too small a value results in undersized steps in the descent direction, meaning convergence might be slow; too large a value may result in "overshooting" the region where descent is guaranteed, leading to oscillation around the optimum or even to complete divergence.

As such, it is often favourable to use an *adaptive learning rate*: rather than a fixed constant η, a new value η_k is used at each iteration of the algorithm. Many methods exist for determining such adaptive rates, often based on line search methods - see [3] and [4].

In this paper we will use a relatively simple method based on backtracking line search, where the learning rate is steadily grown by some constant factor at each iteration, and then more dramatically decreased when it is too large (see Algorithm 1).

2.5 Assessing Performance

When developing a model, it is essential to know how well the model is performing - both in terms of relating the output back to the original input data, and for the purpose of comparing the performance different models. This implies that we will require at least one *performance metric*, which somehow quantifies how well the model is completing the task at hand.

Algorithm 1 "Grow and slash" learning rate

Choose $\eta_1 > 0$, $\alpha > 1$, $\beta > 1$ (in practice, $\alpha = 1.1$, $\beta = 2$ work well).
for iterations $k = 1, ...$ of gradient descent **do**
  while $J(\boldsymbol{\theta} + \eta_k(\Delta\boldsymbol{\theta})) > J(\boldsymbol{\theta})$ **do**
    $\eta_k \leftarrow \eta_k/\beta$
  end while
  $\eta_{k+1} \leftarrow \alpha\eta_k$
end for

Arguably the simplest such metric is *accuracy*, i.e. the proportion of observations which the network predicts correctly as belonging to their associated class. Accuracy is measured on a scale of 0 to 1, where 0 corresponds to a model which predicts everything incorrectly and 1 corresponds to every observation being classified correctly.

The problems we will consider in Section 4 of this paper will use data with *balanced classes*, where an identical (or very similar) proportion of the data belongs to each class. This means it is reasonable for us to use accuracy as our performance metric, since poor performance on any class will affect the overall accuracy evenly.

Consider, however, a case of imbalanced classes - say 90% of observations belonging to one, and 10% to another. A model which simply predicts that every observation belongs to the first class would achieve 90% accuracy without serving any useful purpose at all! Therefore many other performance metrics have been developed (perhaps most famously, *precision*, *recall* and F_1 *score*) which penalise mistakes differently, depending on their perceived significance.

3 Neural Networks

The logistic regression model is, by definition, a *linear classifier*: the output is directly linked (via the sigmoid function) to the weighted sum of the inputs. Since we are using such a model for binary classification, we set some threshold (generally 0.5 for sigmoid) such that all outputs above the threshold are mapped to one class, and all outputs below are mapped to the other. We can thus think of the decision boundary as a hyperplane which divides the problem's feature space into two halves.

However, such a linear boundary may not be particularly suitable - the classes may not be linearly separable, meaning we could never achieve perfect accuracy with our model. Moreover we are only able to distinguish between binary-labelled data, which is not particularly useful if our observations can belong to more than two classes.

Hence we would like to extend our logistic regression model, to allow for an arbitrary number of outputs to be predicted and for the construction of potentially non-linear decision boundaries.

3.1 Multiple Outputs

So far we have considered passing the input $\mathbf{x}$ into a single logistic regression model, yielding a single scalar output a.

By passing the same input $\mathbf{x}$ to n separate logistic regression models, we can obtain n separate outputs $a_1, \ldots, a_n$. If each model is initialised with a different set of weights $\boldsymbol{\theta_n}$, each would produce a slightly different output.

Now suppose that we have a classification problem where each observation can belong to one of $n \geq 2$ classes. We can characterise the target output as a *one-hot encoded* vector $\mathbf{y} \in \mathbb{R}^n$: for $\mathbf{x}$ belonging to class k, we define

$$y_k = \begin{cases} 1 & \text{observation belongs to class } k \\ 0 & \text{otherwise} \end{cases}.$$

Thus we can train the kth model to produce an output of 1, and the other $(n-1)$ models to produce an output of 0; and consequently we can obtain a prediction for the class of a new observation by seeing which model produces the output closest to 1.

Moreover, often in such settings it is useful to have not only a prediction of the class, but also some estimation of the *certainty* of that prediction.

This motivates the introduction of a new activation function, the *softmax* function: if $\mathbf{z} \in \mathbb{R}^n$, then for $j = 1, \ldots, n$,

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{n} e^{z_k}} .$$

Since this is a normalisation of the exponentiated raw output $\mathbf{z}$, the elements of the resulting vector $\mathbf{a}$ sum to 1; and therefore each of these n values can be interpreted as the probability of x belonging to that class. The prediction of the model will simply be the class corresponding to the highest probability.

Note the following useful property of the softmax function:

Proposition 1. *The softmax function is invariant to constant shifts.*

Proof. Let $c \in \mathbb{R}$, and let $\bar{\mathbf{z}} \in \mathbb{R}^n$ be defined by $\bar{z}_j = z_j - c$ for all $j = 1, \ldots, n$. Observe that

$$\begin{aligned}
\sigma(\bar{z}_j) &= \frac{e^{\bar{z}_j}}{\sum_{k=1}^{n} e^{\bar{z}_k}} \\
&= \frac{e^{z_j - c}}{\sum_{k=1}^{n} e^{z_k - c}} \\
&= \frac{e^{-c} e^{z_j}}{\sum_{k=1}^{n} e^{-c} e^{z_k}} \\
&= \frac{e^{z_j}}{\sum_{k=1}^{n} e^{z_k}} \\
&= \sigma(z_j) .
\end{aligned}$$

□

In practical applications, a shift of $c = \max_{j=1,\ldots,n} z_j$ is often applied since this eliminates the slight risk of computational inaccuracies being introduced by very large values of e^{z_j}.

The *multiclass cross-entropy cost function* is typically used with multiclass softmax outputs:

$$J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(-\sum_{k=1}^{n} (\mathbf{y_i})_k \log((\mathbf{a_i})_k) \right) .$$

Note that this is in fact a generalisation of the binary cross-entropy cost function: since $\mathbf{y_i}$ is a one-hot vector, only one term of the inner summation is non-zero, just as only one of the two terms in the binary cross-entropy is non-zero.

3.2 Layers and Forward Propagation

Recall that the motivation behind logistic regression was to find some underlying pattern in the input which characterised the class of that input.

With the "multiple output" model described in Section 3.1, we produced n separate models, and initialised each of them with a different set of weights $\boldsymbol{\theta_n}$ in the hope of learning different patterns and consequently producing different outputs (see [2] for more information on selecting initial weights).

But now notice that these n outputs, when considered together, form a new vector $\mathbf{a} \in \mathbb{R}^n$ - and there is no reason why we shouldn't use this new vector as the input into a *different* set of logistic regression models, which learn to identify patterns in *those* values (i.e. patterns of patterns in the original input $\mathbf{x}$). And subsequently, there is no reason why we shouldn't use the vector produced by *that* set of models as the input to *another* set of models, and so on.

This is the underlying principle of an *artificial neural network*. Each set of logistic regression models is referred to as a *layer*, and each model is referred to as a *neuron*. This terminology was inspired by the parallels between such networks of connected models and the activity of neurons in the brain, and there continues to be much ongoing work in the development of biologically-inspired models [5].

Since we now have a set of outputs $\mathbf{z}$ (and activated outputs $\mathbf{a}$) from each layer of our network, some extensions to the notation we have been using so far are necessary:

- A superscript index will be used to indicate the layer which produces each output, i.e. in an L-layer network we will have $\mathbf{z}^{(l)}$ and $\mathbf{a}^{(l)}$ for $l = 1, \ldots, L$.
- We shall denote the dimension of these vectors as n_l for each l.
- A bias unit $a_0 = 1$ is prepended as the first element of each output $\mathbf{a}^{(l)}$ before it is used as the input to the next layer.
- Each neuron in layer l will consequently contain $(n_{l-1} + 1)$ weights; the weights for the n_l models in each layer shall be stored as the rows of a matrix $\theta^{(l)} \in \mathbb{R}^{n_l,(n_{l-1}+1)}$, so the ith weight from the jth neuron in the layer which produces $\mathbf{a}^{(l)}$ is $\theta^{(l)}_{i,j}$.

The propagation of the original input through the layers of the network is called *forward propagation*. Note that the activated output of the final layer, $\mathbf{a}^{(L)}$, is considered to be the overall output of the network, and it is this output which we wish to produce results similar to our targets $\mathbf{y_i}$. Note also that it is this output which is used in the cost function. The layers of the network between the input and output layers are often called *hidden layers*, and a network containing one or more hidden layers is sometimes referred to as a *deep learning* model.

3.3 Backpropagation

Just as in the logistic regression model described in Section 2, when training our model we wish to minimise the cost function J by adjusting the weights in the model. The difference now is that we have multiple layers of neurons, only the last of which is directly connected to the network's output $\mathbf{a}^{(L)}$; and therefore if we still wish to use some gradient-based optimisation method to update the weights, we will need to determine way of calculating $\frac{\partial J}{\partial \theta^{(l)}_{i,j}}$, i.e. the derivative of the cost function with respect to each individual weight in the network.

We now prove a series of results culminating in Theorem 2, which allows us to do exactly that. The general structure of this section is similar to the corresponding section in [1] but with some additional results relating to different cost and activation functions.

Whenever relevant, we consider the cost for a dataset of size $m = 1$ where the observation has a one-hot target label $\mathbf{y} \in \mathbb{R}^{n_L}$; but note that everything immediately generalises to larger values of m, since in all cases differentiation distributes over the outer summation (over m).

Before starting, we define a useful intermediate quantity: the *error* of layer l, $\boldsymbol{\delta}^{(l)} \in \mathbb{R}^{n_l}$, with components defined for $j = 1, \ldots, n_l$ by

$$\delta^{(l)}_j = \frac{\partial J}{\partial z^{(l)}_j} \, .$$

The jth component of this error vector is a measure of the sensitivity of the cost function J to the (weighted) output from neuron j in layer l. Although it is not exactly a direct measure of how much a particular neuron is at fault for mistakes in the network's eventual output, such an explanation is at least valid intuitively, since all partial derivatives must be zero if the cost function is at a minimum.

Initially we calculate the error term $\delta^{(L)}_j$ for each neuron in the final layer of the network, for each of the binary and multiclass settings.

Lemma 1. *If J_2 is the binomial cross-entropy cost function, and σ_2 is the sigmoid activation function, then for all $j = 1, \ldots, n_L$,*

$$\delta^{(L)}_j = a^{(L)}_j - y_j \, .$$

Proof. Observe that the partial derivative of J_2 with respect to $a^{(L)}_i$ (the ith component of the activated output $\mathbf{a}^{(L)}$) can be represented as

$$\frac{\partial J_2}{\partial a_i^{(L)}} = \frac{\partial}{\partial a_i^{(L)}} \left(-y_i \log(a_i^{(L)}) - (1-y_i)\log(1-a_i^{(L)}) \right)$$

$$= -\frac{y_i}{a_i^{(L)}} + \frac{1-y_i}{1-a_i^{(L)}} \,.$$

Consider now the partial derivative of the activated output of layer L, with respect to the non-activated output; that is,

$$\frac{\partial a_i^{(L)}}{\partial z_j^{(L)}} = \frac{\partial}{\partial z_j^{(L)}} \left(\sigma_2(z_j^{(L)}) \right) \quad = \frac{\partial}{\partial z_j^{(L)}} \left(\frac{1}{1+e^{-z_i^{(L)}}} \right) .$$

Note that if $i \neq j$ then this is clearly zero. If $i = j$,

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \left(-e^{-z_j^{(L)}} \right) \left(- \left(1+e^{-z_j^{(L)}} \right)^{-2} \right)$$

$$= \frac{e^{-z_j^{(L)}}}{\left(1+e^{-z_j^{(L)}} \right) \left(1+e^{-z_j^{(L)}} \right)}$$

$$= \left(\frac{1}{1+e^{-z_j^{(L)}}} \right) \left(\frac{\left(1+e^{-z_j^{(L)}} \right) - 1}{1+e^{-z_j^{(L)}}} \right)$$

$$= \sigma_2(z_j^{(L)}) \left(1 - \sigma_2(z_j^{(L)}) \right)$$

$$= a_j^{(L)} \left(1 - a_j^{(L)} \right) .$$

Now using the definition of $\boldsymbol{\delta}^{(L)}$ and the chain rule,

$$
\begin{aligned}
\delta_j^{(L)} &= \frac{\partial J_2}{\partial z_j^{(L)}} \\
&= \sum_{i=1}^{n_L} \left(\frac{\partial J_2}{\partial a_i^{(L)}} \, \frac{\partial a_i^{(L)}}{\partial z_j^{(L)}} \right) \\
&= \frac{\partial J_2}{\partial a_j^{(L)}} \, \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} + \sum_{\substack{i=1 \\ i \neq j}}^{n_L} \left(\frac{\partial J_2}{\partial a_i^{(L)}} \, \frac{\partial a_i^{(L)}}{\partial z_j^{(L)}} \right) \\
&= \frac{\partial J_2}{\partial a_j^{(L)}} \, \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} + 0 \\
&= \left(-\frac{y_j}{a_j^{(L)}} + \frac{1 - y_j}{1 - a_j^{(L)}} \right) a_j^{(L)} \left(1 - a_j^{(L)} \right) \\
&= a_j^{(L)}(1 - y_j) - y_j(1 - a_j^{(L)}) \\
&= a_j^{(L)} - y_j \, .
\end{aligned}
$$

□

Lemma 2. *If J is the multiclass cross-entropy cost function, and σ is the softmax activation function, then for all $j = 1, \ldots, n_L$,*

$$\delta_j^{(L)} = a_j^{(L)} - y_j \, .$$

Proof. Firstly note that

$$
\begin{aligned}
\frac{\partial J}{\partial a_i^{(L)}} &= \frac{\partial}{\partial a_i^{(L)}} \left(-\sum_{k=1}^{n_L} y_k \log(a_k^{(L)}) \right) \\
&= -\frac{y_i}{a_i^{(L)}} \, .
\end{aligned}
$$

Then using the quotient rule,

$$
\begin{aligned}
\frac{\partial a_i^{(L)}}{\partial z_j^{(L)}} &= \frac{\partial}{\partial z_j^{(L)}} \left(\frac{e^{z_i^{(L)}}}{\sum_{k=1}^{n_L} e^{z_k^{(L)}}} \right) \\
&= \frac{\left(\frac{\partial \left(e^{z_i^{(L)}} \right)}{\partial z_j^{(L)}} \sum_{k=1}^{n_L} e^{z_k^{(L)}} \right) - e^{z_i^{(L)}} e^{z_j^{(L)}}}{\left(\sum_{k=1}^{n_L} e^{z_k^{(L)}} \right)^2} \, .
\end{aligned}
$$

- If $i \neq j$,

$$\begin{aligned}\frac{\partial a_i^{(L)}}{\partial z_j^{(L)}} &= \frac{0 - e^{z_i} e^{z_j}}{\left(\sum_{k=1}^{n_L} e^{z_k^{(L)}}\right)^2} \\ &= -\sigma(z_i)\sigma(z_j) \\ &= -a_i^{(L)} a_j^{(L)} .\end{aligned}$$

- On the other hand, if $i = j$,

$$\begin{aligned}\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} &= \frac{\left(e^{z_j^{(L)}} \sum_{k=1}^{n_L} e^{z_k^{(L)}}\right) - \left(e^{z_j^{(L)}}\right)^2}{\left(\sum_{k=1}^{n_L} e^{z_k^{(L)}}\right)^2} \\ &= \frac{e^{z_j^{(L)}}}{\sum_{k=1}^{n_L} e^{z_k^{(L)}}} \left(1 - \frac{e^{z_j^{(L)}}}{\sum_{k=1}^{n_L} e^{z_k^{(L)}}}\right) \\ &= \sigma(z_j^{(L)}) \left(1 - \sigma(z_j^{(L)})\right) \\ &= a_j^{(L)} \left(1 - a_j^{(L)}\right) .\end{aligned}$$

Now once again using the definition of $\boldsymbol{\delta}^{(L)}$ and the chain rule, observe that

$$\begin{aligned}\delta_j^{(L)} &= \frac{\partial J(\theta)}{\partial z_j^{(L)}} \\ &= \sum_{i=1}^{n_L} \left(\frac{\partial J(\theta)}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_j^{(L)}}\right) \\ &= \frac{\partial J(\theta)}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} + \sum_{\substack{i=1 \\ i \neq j}}^{n_L} \left(\frac{\partial J(\theta)}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_j^{(L)}}\right) \\ &= -\frac{y_j}{a_j^{(L)}} \left(a_j^{(L)}(1 - a_j^{(L)})\right) + \sum_{\substack{i=1 \\ i \neq j}}^{n_L} \left(-\frac{y_i}{a_i^{(L)}} \left(-a_i^{(L)} a_j^{(L)}\right)\right) \\ &= -y_j(1 - a_j^{(L)}) + a_j^{(L)} \left(\sum_{i=1}^{n_L} (y_i) - y_j\right) .\end{aligned}$$

This looks slightly complicated, but recall that $\mathbf{y}$ is a one-hot encoded vector, and therefore $\sum_{=1}^{n_L} y_i = 1$. Hence we have

$$\begin{aligned}\delta_j^{(L)} &= -y_j(1 - a_j^{(L)}) + a_j^{(L)}(1 - y_j) \\ &= a_j^{(L)} - y_j .\end{aligned}$$

□

We now show some auxiliary results which will come in handy shortly.

Lemma 3. *For all $i = 1, \ldots, n_l$ and $j = 0, \ldots, n_{l-1}$, and all $l = 2, \ldots, L$,*

$$\frac{\partial z_i^{(l)}}{\partial a_j^{(l-1)}} = \theta_{i,j}^{(l)} .$$

Proof. By the definition of $z_i^{(l)}$, we have

$$\begin{aligned} \frac{\partial z_i^{(l)}}{\partial a_j^{(l-1)}} &= \frac{\partial}{\partial a_j^{(l-1)}} \left(\sum_{k=0}^{n_{l-1}} a_k^{(l-1)} \theta_{i,k}^{(l)} \right) \\ &= \theta_{i,j}^{(l)} . \end{aligned}$$

□

Lemma 4. *For all $i = 1, \ldots, n_l$ and $j = 0, \ldots, n_{l-1}$, and all $l = 2, \ldots, L$,*

$$\frac{\partial a_i^{(l)}}{\partial z_j^{(l)}} = \begin{cases} \sigma'(z_j^{(l)}) & i = j \\ 0 & i \neq j . \end{cases}$$

Proof. Immediate, recalling that $a_i^{(l)} = \sigma(z_i^{(l)})$.

□

Using the results so far, we can find an expression for $\boldsymbol{\delta}^{(l)}$ in terms of $\boldsymbol{\delta}^{(l+1)}$.

Theorem 1. *For $j = 1, \ldots, n_l$, and for $l = 2, \ldots, L-1$,*

$$\delta_j^{(l)} = \sigma'(z_j^{(l)}) \sum_{i=1}^{n_l} \left(\delta_i^{(l+1)} \theta_{i,j}^{(l+1)} \right) .$$

Proof. By the chain rule and the definition of $\boldsymbol{\delta^{(l+1)}}$, notice

$$\begin{aligned} \delta_j^{(l)} &= \frac{\partial J}{\partial z_j^{(l)}} \\ &= \sum_{i=1}^{n_{l+1}} \left(\frac{\partial J}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}} \right) \\ &= \sum_{i=1}^{n_{l+1}} \left(\delta_i^{(l+1)} \frac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}} \right) . \end{aligned}$$

Then considering the remaining differential term, and using the chain rule once again, observe that

$$\begin{aligned}\frac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}} &= \sum_{k=0}^{n_l}\left(\frac{\partial z_i^{(l+1)}}{\partial a_k^{(l)}}\frac{\partial a_k^{(l)}}{\partial z_j^{(l)}}\right)\\ &= \sum_{k=0}^{n_l}\left(\theta_{i,k}^{(l+1)}\frac{\partial a_k^{(l)}}{\partial z_j^{(l)}}\right) \qquad \text{by Lemma 3}\\ &= \theta_{i,j}^{(l+1)}\sigma'(z_j^{(l)}) \qquad \text{by Lemma 4}\,.\end{aligned}$$

Therefore combining these results, we have that

$$\begin{aligned}\delta_j^{(l)} &= \sum_{i=1}^{n_{l+1}}\left(\delta_i^{(l+1)}\theta_{i,j}^{(l+1)}\sigma'(z_j^{(l)})\right)\\ &= \sigma'(z_j^{(l)})\sum_{i=1}^{n_l}\left(\delta_i^{(l+1)}\theta_{i,j}^{(l+1)}\right)\,.\end{aligned}$$

□

This result is the core of the infamous *backpropagation* algorithm, widely used since its popularisation in the 1980s [6] and so called because it allows the error of each layer to be efficiently calculated using the following layer's error.

We require one more auxiliary result before we can fully derive the derivative of J with respect to each parameter in the network.

Lemma 5. *For all $i = 1, \ldots, n_l$ and $j = 0, \ldots, n_{l-1}$, and all $l = 2, \ldots, L$,*

$$\frac{\partial z_p^{(l)}}{\partial \theta_{i,j}^{(l)}} = \begin{cases} 0 & p \neq i \\ a_j^{(l-1)} & p = i\,. \end{cases}$$

Proof. Using the definition of $z_p^{(l)}$,

$$\begin{aligned}\frac{\partial z_p^{(l)}}{\partial \theta_{i,j}^{(l)}} &= \frac{\partial}{\partial \theta_{i,j}^{(l)}}\left(\sum_{k=0}^{n_{l-1}}\theta_{p,k}^{(l)}a_k^{(l-1)}\right)\\ &= \sum_{k=0}^{n_{l-1}}\frac{\partial}{\partial \theta_{i,j}^{(l)}}\left(\theta_{p,k}^{(l)}a_k^{(l-1)}\right)\,.\end{aligned}$$

The result follows by noting that all terms in the summation, or all-but-one terms if $p = i$, are zero.

□

We now have all the results we need to achieve our goal.

Theorem 2. *For all $i = 1, \ldots, n_l$ and $j = 0, \ldots, n_{l-1}$, and all $l = 2, \ldots, L$,*

$$\frac{\partial J}{\partial \theta_{i,j}^{(l)}} = \delta_i^{(l)}a_j^{(l-1)}\,.$$

Proof. Using the chain rule, the definition of $\boldsymbol{\delta}^{(l)}$ and the result from Lemma 5,

$$\begin{aligned}\frac{\partial J}{\partial \theta_{i,j}^{(l)}} &= \sum_{k=1}^{n_l} \left(\frac{\partial J}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial \theta_{i,j}^{(l)}} \right) \\ &= \sum_{k=1}^{n_l} \left(\delta_k^{(l)} \frac{\partial z_k^{(l)}}{\partial \theta_{i,j}^{(l)}} \right) \\ &= \delta_i^{(l)} a_j^{(l-1)} \, .\end{aligned}$$

□

Thus with the combination of Theorems 1 and 2, and the starting value provided by either Lemma 1 or 2, we now have an efficient means of calculating the derivative of the cost function J with respect to every weight in the network; and these derivatives will allow us to minimise J using any first-order gradient method, such as gradient descent.

3.4 Improvements to Training

In Section 2.4 we discussed the update to be made to the weights in the logistic regression model - concretely, a step in the direction of steepest descent $-\nabla J(\boldsymbol{\theta})$. If we consider $\boldsymbol{\theta}$ to be a concatenated vector of the unrolled weight matrices, i.e.

$$\boldsymbol{\theta} = \left(\theta_{1,1}^{(2)},\ \theta_{1,2}^{(2)},\ \ldots,\ \theta_{n_L,(n_{L-1}+1)}^{(L)} \right) ,$$

then we can use gradient descent as the basis for the training process of the neural network too.

However, since there are vastly more weights in even a small neural network than in the logistic regression model, it will take a significantly longer amount of time to calculate the descent direction $\Delta\boldsymbol{\theta}$ at each iteration.

Moreover, the potential for the model to *overfit* the training data by learning overly specific patterns (such as noise in the training dataset) is also greatly increased, due to the larger number of degrees of freedom in the model. Usually, we will want to use our model to make predictions based on new data, rather than predicting the already-known labels of our training dataset; and therefore we would like to produce a model which will *generalise*, performing as well on unseen data as possible.

Thankfully, measures can be taken in order to mitigate the issues described above: several are outlined in the following subsections.

3.4.1 Mini-Batch Gradient Descent

Given that we only have a certain amount of information available to us for training our model, we would of course like to take full advantage of it; therefore the training process generally involves making multiple passes over the dataset, using it to determine the updates we will make to the weights in the model. Each pass is known as an *epoch*: one training epoch has passed after each observation in the dataset has influenced an update to the model. This definition allows for several different interpretations of gradient descent.

The simplest approach is *(full-)batch gradient descent*: the costs of all m observations in the training set are calculated and averaged, and then the descent direction is defined as $-\nabla J(\boldsymbol{\theta})$.

However there are certain serious downsides to this approach:

- The cost of every single observation in the dataset must be calculated before any update is made to the model's weights. This is potentially computationally expensive.
- The update which *is* eventually made *always* results in a smaller value of J. However, since J is as a rule not a convex function, there are multiple minima; and by always moving in a strict descent direction, we risk converging to a "sharp minimum" where the weights we learn will define a model which does not generalise well to unseen data [7].

Stochastic gradient descent instead considers a subset of the training data: that is, taking some subset of indices $\mathcal{S}_i \subset \{1,\ \ldots,\ m\}$, we define our descent direction as

$$\Delta\boldsymbol{\theta} = -\nabla J(\boldsymbol{\theta})\big|_{\mathcal{S}_i} = -\left(\frac{1}{|\mathcal{S}_i|}\sum_{k\in\mathcal{S}_i} CE(a_k^{(L)}, y_k \mid \boldsymbol{\theta})\right).$$

This stochastic gradient is essentially a "noisy" version of the true gradient $-\nabla J(\boldsymbol{\theta})$. Therefore if after a certain iteration we are sitting in a sharp minimum of the cost function, then by making an update to the weights based on the stochastic gradient we are likely to jump out of this minimum (since it is quite possible that it doesn't even exist in the stochastic version of the cost function surface). Therefore when using stochastic gradient descent we tend to converge to "flat" minima, where the performance of the model is usually close to its performance at the global minimum and where the model generalises well to unseen data [8].

The number of observations belonging to $\mathcal{S}_i$ becomes a *hyperparameter* of the model (so called because the weights of the model are the standard parameters). This value, and the values of other hyperparameters which we will meet shortly, is chosen when setting up the training process.

In *mini-batch gradient descent*, in each training epoch the full training dataset is randomly separated into equally-sized sets (typically $32 \leq |\mathcal{S}_i| \leq 512$), and the model's weights are updated based on each set in turn. In this way multiple updates are made to the model during each training epoch, and we improve the chances of converging to a flat minimum of the cost function.

3.4.2 Validation and Testing

To reduce the likelihood that we overfit our model to the dataset, it is good practice to split the available data into separate *training and validation sets*. The model is trained and the weights updated based on the training set; and then its performance on the validation set is assessed. Hyperparameters are adjusted to give optimal results on the validation set. Since the validation data is not used to update the weights in the network, this process should hopefully produce a model which generalises better to unseen data.

The simplest implementation is *holdout cross-validation*: some proportion of the available dataset is "held out" as the validation set for the duration of the training process. A more robust extension is *k-fold cross-validation*: the available dataset is split into k approximately equally-sized sections, and the training process is repeated k times, where each section of the data in turn is considered as the validation set while the remaining sections form the training set. The performance metric from each of the k repetitions can be somehow combined (usually averaged) to produce an overall value.

As a final measure, it is usually sensible to split the available data still further - into training, validation sets and a *test* set, which remains isolated and unused until all training and hyperparameter tuning is completed. The model's performance on this test set is as reliable an indicator as we can achieve of how well the model will perform on unseen data.

3.4.3 Regularisation

Another typical symptom of overfitting is the interpretation of noise in the training dataset as patterns, and consequent updates being made to the model's weights such that this noise becomes significant in how observations are classified. In that situation, small variation in a few specific inputs can lead to a massive change in the model's output, while the remaining inputs are largely unconsidered.

We can encourage better "spreading" of the weights by adding some *regularisation term* $R(\bar{\boldsymbol{\theta}})$, which penalises large weight values, to the cost function. Here $\bar{\boldsymbol{\theta}}$ is the same as $\boldsymbol{\theta}$, except the elements corresponding to bias units ($\theta_{i,0}^{(l)}$) are set to zero, since we do not wish to penalise changes to the weights which simply shift the solution away from the origin.

One of the most widely-used regularisation methods is *L_2-regularisation* (sometimes called *ridge regression*), where the regularisation term is the L_2 norm of the weight vector, scaled by some constant λ. Then our cost function becomes

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(-\sum_{k=1}^{n}(\mathbf{y_i})_k \log((\mathbf{a_i})_k)\right) + \frac{\lambda}{2}\sum_j \bar{\theta}_j^2,$$

and notice that the differential of the new term with respect to each θ_j is simple to calculate - so during the backpropagation calculations we will now have

$$\frac{\partial J}{\partial \theta_{i,j}^{(l)}} = \delta_i^{(l)} a_j^{(l-1)} + \lambda \bar{\theta}_{i,j}^{(l)} .$$

3.4.4 Early Stopping

One major potential cause of overfitting and wasted training effort is *overtraining*, where the network is forced to complete a fixed number of training epochs despite the fact that additional training may have no benefits after a certain point. *Early stopping methods* try to prevent this by terminating the training process as soon as certain "stopping criteria" are met.

The most widely used such method is *validation-based early stopping*. While there are no hard-and-fast rules, many approaches have proven to be useful in practice [9].

In the implementation used in Section 4, early stopping is controlled by a criterion based on the third class of criteria introduced in Section 2.2.1 of [9]. A window of s iterations is considered: if the average validation cost from the most recent s epochs is higher than the average validation cost from the preceding s epochs, then training terminates and the weights are set to the values they held when validation cost was at its lowest. The width of the window s is a hyperparameter of the network which usually requires some tuning for the best results: too small a value might cause premature termination due to variation caused by stochastic weight updates, whereas too large a value might result in many redundant iterations taking place after the optimal weights were found.

4 Demonstration of Methods

In the previous sections of this paper, the theory and intuition behind logistic regression and neural network models has been explored and explained in some detail. However in order to actually solve problems in the real world, theory alone is of limited use; and therefore in order to provide a demonstration that methods based on the theory do indeed perform as expected, we provide accompanying implementations of these models (written in Python). [1]

A highly structured, object-oriented approach has been taken when writing the code, which results in the end product being relatively simple to use, to experiment with and hopefully to extend if desired.

A model is created as an instance of the `Network` class, and contains one or more layers (all of which are individual classes built on top of the `Layer` superclass). The first layer in a network must always be an `Input` layer which defines the shape of the input; thereafter, layers can be added at will as instances of their respective class (e.g. `Dense`, so called because every input is passed into every neuron and therefore influences every output), the only requirement being that the output size of each layer matches the input size of the next. An activation function can additionally be specified for each layer individually. Hence a typical neural network model with two hidden layers designed to classify observations of size 10 into 4 groups, might be constructed as follows:

```
model = Network(layers = [
    Input(output_size=10),
    Dense(input_size=10, output_size=20, activation="sigmoid"),
    Dense(input_size=20, output_size=12, activation="sigmoid"),
    Dense(input_size=12, output_size=4, activation="softmax")
])
```

Such a model has an associated `train()` method where data, labels and hyperparameters are passed in. A typical training run might look like:

```
success = model.train(
    X_train, y_train, X_val, y_val, epochs=1000, batch_size=128,
    learning_rate=10, penalty=0.1, early_stopping=200
)
```

Useful vectors such as the history of the training and validation costs after each epoch are stored, and an assortment of other methods are available to assist with training, inspection and refinement of the model.

[1] Code available at `https://github.com/owenjonesuob/neural-nets-for-classification`.

The following subsections demonstrate the results of the implementation of these methods to solve some small example problems, concluding with an application of a neural network model to a larger, more challenging dataset.

4.1 Binary Classification

To show the difference in approach between logistic regression and neural network models, we shall attempt to carry out classification on a two-class dataset where the classes are not linearly separable in 2-dimensional space.

First we shall implement a logistic regression model. In neural network terminology, a logistic regression model is a single sigmoid-activated neuron, so we can use the object-oriented implementation to define a logistic regression model as follows:

```
model = Network(layers = [
    Input(2),
    Dense(2, 2, activation="sigmoid")
])
```

There is one slight quirk here, in that we must provide one-hot encoded labels to the `train()` method; and therefore we must actually train two output neurons, one corresponding to a class of 0 and the other to a class of 1. However, note that these two neurons are in fact "mirror images": if an observation does not belong to one class, then it *must* belong to the other, and therefore by the symmetry of the binary cross-entropy cost function the weights of one neuron are updated in precisely the opposite direction to those of the other. Consequently after training, the weights of one neuron are simply the negatives of the weights of the other.

In the left-hand plot of Figure 2 we see the result of 200 epochs of training: a linear decision boundary giving a training set accuracy of approximately 88%.

In order to do better, we will need to increase the size of our model - although not by a very large amount:

```
model = Network(layers = [
    Input(2),
    Dense(2, 3, activation="sigmoid"),
    Dense(3, 2, activation="sigmoid")
])
```

Adding a very small hidden layer to the model is enough to capture the non-linear decision boundary almost perfectly, as can be seen in the right-hand plot of Figure 2. This model achieves over 99% accuracy after 200 epochs of training.

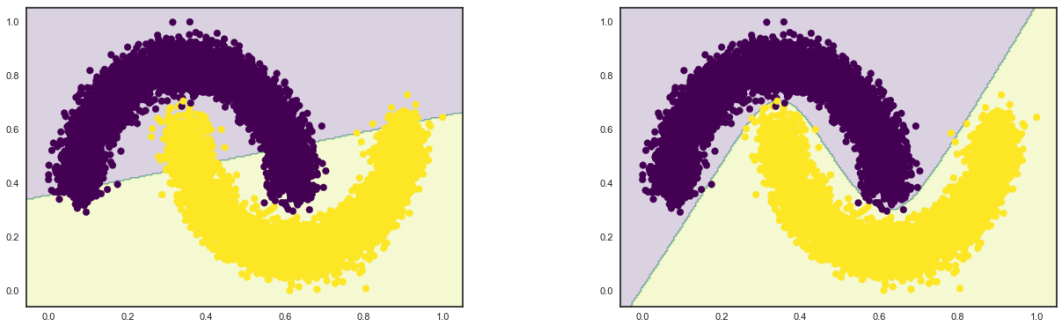

Figure 2: The *left-hand* plot shows the decision boundary found by logistic regression after 200 training epochs. The *right-hand* plot shows the decision boundary found by a small neural network after 200 training epochs. All hyperparameters are the same in both models.

We should take care, however, not to define a model which is unnecessarily complex for the task at hand; and not to overtrain our model once an adequate level of performance has been achieved, since otherwise we are likely to

encounter overfitting as described in Section 3.4. Figure 3 shows the consequences of overtraining a model and how early stopping can help to prevent this from happening.

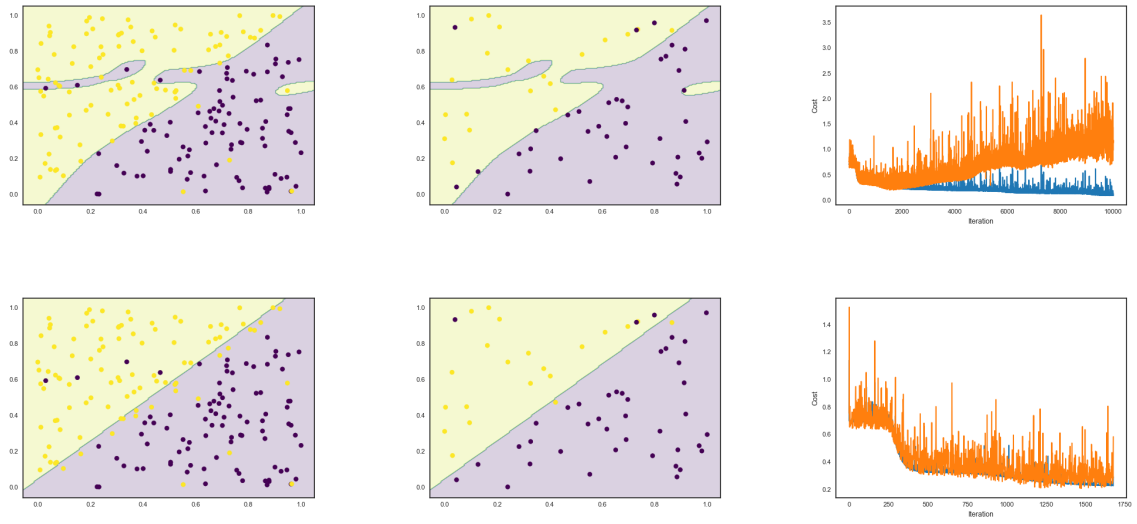

Figure 3: The *top* row corresponds to a model which was overtrained: noise in the training dataset has been learned (first column), and the model does not generalise well to unseen test data (second column). The validation cost curve, orange, veers noisily away from the training cost curve, blue (final column). The *bottom* row corresponds to exactly the same model, but with early stopping implemented as described in Section 3.4.4 ($s = 200$). Note how training has been terminated once the validation cost curve starts to rise, which has resulted in a much better decision boundary for generalisation of the model.

4.2 Multiclass Classification

In order to demonstrate the behaviour of a multiclass neural network classifier, we first consider a small two-dimensional problem featuring four fully separable classes. The initialisation of a model is much the same as was demonstrated in the previous subsection, although we will add an extra hidden layer:

```
model = Network(layers = [
    Input(2),
    Dense(2, 20, "sigmoid"),
    Dense(20, 20, "sigmoid"),
    Dense(20, 4, "softmax")
])
```

Notice also that since we are no longer predicting a binary outcome, we have switched to using the softmax activation function in the final layer of the network.

Figure 4 provides an insight into how the decision boundaries change during the training process. The left-hand plot shows the boundaries after approximately 400 training epochs, when the outer groups of observations have already been largely classified correctly (but training was prematurely terminated by early stopping); then the central plot shows how after approximately 1400 training epochs, the boundaries have shifted to enclose the inner groups.

For the remainder of this section we will consider the Kuzushiji-MNIST (KMNIST) dataset [2] of handwritten Japanese *hiragana* characters. More information and background on the dataset can be found in [10].

KMNIST is designed as a drop-in replacement for the infamous MNIST dataset of handwritten numeric digits, and therefore is identical in structure to MNIST: each observation is a single 28x28 matrix representing a 28x28-pixel

[2] Dataset available at `https://www.kaggle.com/anokas/kuzushiji`.

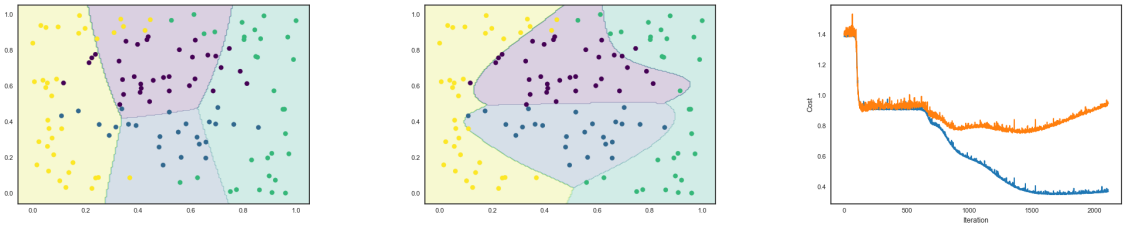

Figure 4: Varying window width s in early stopping, implemented as described in Section 3.4.4. When $s = 80$, training terminated after 434 epochs and weights were reset to the values they held after epoch 402 (left). When $s = 800$, training terminated after 2110 epochs and weights were reset to the values they held after epoch 1409 (centre). These saved epochs correspond to local minima of the validation cost function, in orange (right).

greyscale image. Each element of the matrix represents a pixel, and its value represents the "intensity" of that pixel on a scale of 0 (white) to 1 (black). Some examples from the dataset are shown in Figure 5.

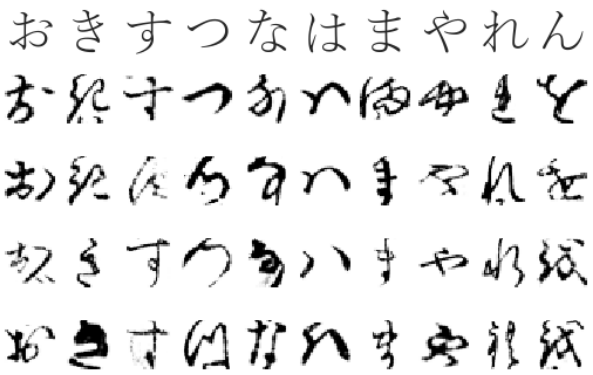

Figure 5: Examples of images belonging to each of the 10 classes of the KMNIST dataset. The top row contains the modern-day Japanese *hiragana* character, typeset, which each class represents.

The training dataset is provided as a 3-dimensional array of size 60000x28x28, and therefore a small amount of data preprocessing was required to "unroll" each of the 60000 observations into a one-dimensional vector of length 784. The dataset was then split by assigning each observation at random into one of three groups making up 60%, 20% and 20% of the original dataset: the largest set was used for training, and the other two for hyperparameter tuning via holdout cross-validation and for final performance testing respectively.

Generally when deciding on a network architecture, it is useful to follow the principle of Occam's razor: in general we ought to choose the simplest model which explains the data sufficiently well. Therefore a relatively small model was defined at first, with a single hidden layer containing a few tens of neurons.

However, it soon became apparent that this model was plateauing at a performance of approximately 87% validation accuracy, even when the size of the hidden layer was increased to several hundred neurons in size.

Therefore a second, smaller, hidden layer was added in an attempt to take advantage of higher-level patterns which might be present in the dataset - in practical terms, a single line of code was added defining the new layer. After some experimentation and adjustment of hyperparameters, a network with two hidden layers containing 200 and 30 neurons respectively was capable of consistently achieving approximately 93% validation set accuracy (and 92% test set accuracy). Figure 6 shows examples from the test dataset which were correctly and incorrectly classified by this model, and gives some idea of the complexity of the task at hand.

Figure 6: "Confusion plot" showing examples from the test dataset which were correctly and incorrectly classified by the trained two-hidden-layer model.

We hope that each neuron in our model has "learned to recognise" some pattern in the dataset, and that the weights for that neuron have been adjusted such that the presence of that pattern will trigger a strong response. It therefore is an interesting exercise to visualise the weights of the network and see the patterns that they have learned. It is only really possible for us to interpret with any meaning the weights of the neurons in the first layer, which receive the original images as input, since neurons in subsequent layers are learning "patterns of patterns" as described in Section 3.2. The weights from a selection of neurons in the first layer of the KMNIST-trained model are visualised in Figure 7: in some cases, hints of patterns resembling brushstrokes in the training images are evident.

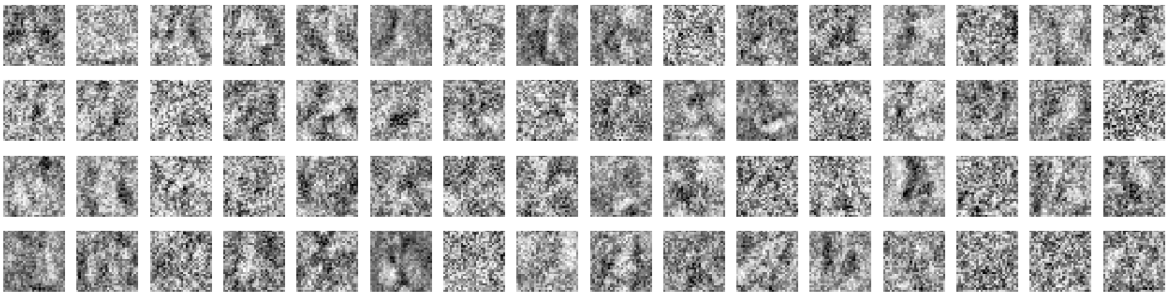

Figure 7: Visualisations of the weights from several neurons in the first layer of the KMNIST-trained network. The black pixels in each image correspond to the most positive weight values, and white pixels to the most negative weight values.

5 Areas for Further Work

The theory and methods introduced in this paper are, by modern standards, considered rather limited and simplistic; and therefore they are not often used alone in modern machine learning applications. However, many larger and more powerful models can be constructed using these simpler models as fundamental building blocks.

The vast majority of modern image classification methods make use of *convolutional neural networks* (CNNs). A "window" of weights is convolved over each training image, and thus only a subsection of the image is used to update the weights at any one time. This preserves much more of the spatial information inherent in the training images (since we avoid the "flattening" process described in Section 4.2) and allows for localised patterns to be learned.

Another family of models which has gained popularity in recent times is the *recurrent neural network* (RNN), which feeds the output from a layer repeatedly back into the same layer. Such networks have been found to perform extremely well when trained on (potentially limitless) sequential data, such as time series or natural language; and many modern applications of natural language processing and machine translation are underpinned by RNNs [11]. A detailed overview of RNNs can be found at [12].

The code implementation demonstrated in Section 4 has intentionally been written with an extendable structure: additional layer classes such `Convolutional` or `Recurrent` could be built on top of the `Layer` superclass.

Additionally, recent interest in machine learning has rekindled research efforts in the field of numerical optimisation. The method used in this paper, stochastic gradient descent, is one of the oldest and least complex optimisation methods; however, countless variants and improvements exist, as well as many other diverse methods which minimise the objective function by other means. A survey of current techniques can be found in [13] and the majority of these could be added to the implementation from Section 4 without too much additional work.

References

[1] C. Higham and D. Higham. Deep Learning: An Introduction for Applied Mathematicians. *arXiv:1801.05894 [math.HO]*, 2018.

[2] Y. LeCun, L. Bottou, G. Orr and K. Muller. Efficient BackProp. In *Neural Networks: Tricks of the trade*, Springer, 1998.

[3] P. Stanimirovic and M. Miladinovic. Accelerated gradient descent methods with line search. In *Numerical Algorithms* 54:503-520, Springer, 2010.

[4] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980 [cs.LG]*, 2014.

[5] A. Thomas and C. Kaltschmidt. Bio-inspired Neural Networks. In *Memristor Networks*, Springer, 2014.

[6] D. Rumelhart, G. Hinton and R. Williams. Learning representations by back-propagating errors. In *Nature* 323:533-536, 1986.

[7] N. S. Keskar et al. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *arXiv:1609.04836 [cs.LG]*, 2017.

[8] A. Choromanska et al. The Loss Surfaces of Multilayer Networks. *arXiv:1412.0233 [cs.LG]*, 2015.

[9] L. Prechelt. Early Stopping — But When?. In *Neural Networks: Tricks of the Trade*. Lecture Notes in Computer Science, vol 7700. Springer, 2012.

[10] T. Clanuwat et al. Deep Learning for Classical Japanese Literature. *arXiv:1812.01718 [cs.CV]*, 2018.

[11] Y. Wu et al. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv:1609.08144 [cs.CL]*, 2016.

[12] A. Kaparthy. The Unreasonable Effectiveness of Recurrent Neural Networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`, 2015.

[13] P. Oliveira et al. Machine-Learning: An overview of optimization techniques. In *Recent Advances in Computer Science*, INASE, Zakynthos, 2015.