Small Neural Networks with Interesting Architectures

Reg. 210138711

School of Mathematics and Statistics
University of Sheffield

The University Of Sheffield.

Dissertation submitted as part of the requirements for the award of
MSc in Statistics, University of Sheffield, 2023–2024

Contents

Acknowledgements

I would like to thank my supervisor for the hours spent guiding and reviewing my work, and for providing code examples which helped me complete this project despite having minimal prior experience with Python.

Lay Summary of the Dissertation

Machine learning is a technique where a mathematical model is built using the help of computers. With the ability to perform billions of calculations per second, the resulting models they can build are of a complexity that no human is able to reproduce. Neural networks are one of the most popular branches of machine learning, and are behind numerous examples of extremely powerful and useful artificial intelligence (AI) tools. A contributing factor to the rise in popularity of neural networks is the ability to train them on large sets of data very quickly using computational optimization and particularly using graphics processing units (GPUs) to perform the operations. This leads to models with very high performance, often being able to mimic or surpass humans.

The term "neural networks" is short for "artificial neural networks", and is what we call machine learning models that are based on biological neural networks, which are found in the brains of animals. A neural network can be represented by a network consisting of "nodes", which represent the neurons in a biological neural network, and "edges", which represent the connections between the neurons. The idea of a neural network is to supply input data and pass it through the network of edges and nodes. At each stage, a transformation is applied to the data and by the end of the network, the model provides us with some output data. The architecture of a neural network describes the structure of the nodes and edges of the underlying network; in a typical neural network these are arranged into layers, with one layer representing the input data, one layer representing the output data, and one or more layers in between known as the "hidden" layers.

One drawback of typical neural networks is that they create "black box" models once they have been trained, where we cannot understand how the model arrives at the output data from a given input. This is partly due to us not having complete control over exactly how the model learns what it does from the data it is given; there is a possibility that the same training process can result in different models based on randomly assigned starting conditions. It is also partly due to the overall complexity of any neural network beyond a relatively small total size, and the fact that we do not usually have control over which parts of the network are extracting particular patterns or information from the data we use to train it.

The aim of this project is to investigate the properties of small neural networks, giving us the chance to take away the black box and gain insight into how particular parts of a given model works. The project contains a mixture of handcrafted models and models trained through standard procedures. We then build on these ideas to create models containing a mixture of handcrafted and trained elements, with the aim to improve on performance, the training time required, or both.

Chapter 1

Introduction

1.1 Neural Networks

1.1.1 Literature Coverage

A search for relevant literature found few publications in recent decades specifically focused on small neural networks. Search terms including "small neural networks" yielded publications focused on neural networks applied to small datasets. This included specifying particular types of neural network through adding keywords including "convolution", "recurrent", "LSTM", and "transformer". Similarly, search terms containing "handbuilt", "hand-built", "handmade", or "hand-made" did not yield any publications that involved constructing neural networks in a similar manner to how we have done so throughout this project. Therefore, the introduction to this project will focus on explaining the ideas of neural networks, including the layer types used in this project.

1.1.2 Introduction to Neural Networks

Artificial Neural Networks (ANNs), or simply "Neural Networks" for short, are perhaps one of the most popular and well-known classes of machine learning models in the world today, able to outperform humans in a number of tasks with applications such as AlphaGo (Silver et al., 2016), a computer player for the game Go, and ChatGPT, a chat-bot with exceptional language capabilities that was the focus of a huge number of articles and news reports in 2023 (Ray, 2023). Neural networks have been widely applied (Sarker, 2021) in image processing, including medical image diagnostics (Jiang et al., 2010) and finance (Fadlalla and Lin, 2001), where they are often able to outperform traditional statistical models.

ANNs are inspired by biological neural networks found in animal brains, which are composed of neurons connected to each other via synapses. Organized into large networks, such a system is able to process information by sending electrical

signals throughout the network.

An ANN similarly simulates neurons and connections, using a graph of vertices (or nodes) and edges respectively. Typically, these networks are arranged into layers of nodes, where the edges connect the nodes from one layer to the nodes in an adjacent layer. The layer at the start of the network is the input layer, where the nodes take the input data, and the layer at the end of the network is the output layer, which produces the output data. The layers in between are called hidden layers. Each edge is assigned a weight, which is a real value that represents the strength of a connection between two nodes. The weights in a network determine the relationship between the input data and the output data, and so adjusting these weights determines how well a neural network will perform at a given task. An activation function is attached to the output nodes of each layer, which is applied to the scaled input it receives to create the final scaled output values. Figure 1.1 shows how a small ANN could be represented.

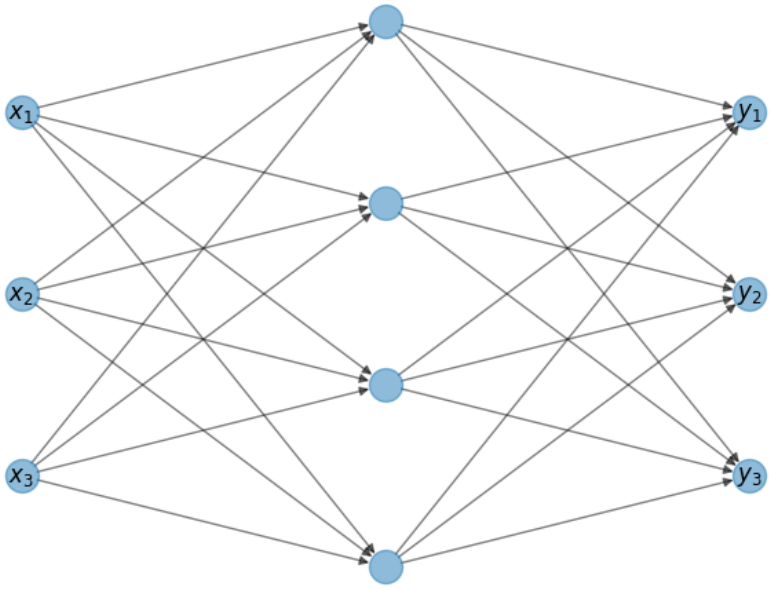

Figure 1.1: A diagram of a simple ANN. There is an input layer consisting of 3 inputs $\boldsymbol{x} = (x_1, x_2, x_3)$, a hidden layer with 4 nodes, and an output layer consisting of 3 outputs $\boldsymbol{y} = (y_1, y_2, y_3)$.

1.1.3 Layer Architectures

Here we introduce different types of layers and the notation used to describe how they function. In general, a layer $\mathcal{L}$ is a function that takes an input vector $\boldsymbol{x}$ from a previous layer, and produces an output $\mathcal{L}(\boldsymbol{x})$. We will use $\mathcal{L}^{[i]}$ to denote the i-th layer of a neural network. The input layer is unique in that it only stores the input values, and throughout this project will be $\mathcal{L}^{[0]}$, meaning the first layer

is specifically the first layer hidden layer after the input layer and so on. Neural network layers take real input values and produce real output values, although we will see examples where the networks are specially designed to restrict the inputs and/or outputs to integers or natural numbers.

A *dense layer* or *fully-connected layer* is a layer that is fully connected to the previous layer: every node in the previous layer is connected by an edge to every node in the dense layer. The output of a dense layer is calculated according to a *weights matrix* $\mathbf{W}$, a *bias vector* $\boldsymbol{b}$, and an *activation function* f. The output of a dense layer is calculated as

$$\mathcal{L}(\boldsymbol{x}) = f(\mathbf{W}\boldsymbol{x} + \boldsymbol{b}).$$

The dimensions of $\mathbf{W}$ and $\boldsymbol{b}$ are such that the result is a vector with length equal to the number of nodes in the dense layer: the matrix $\mathbf{W}$ has one row per node in the dense layer, and one column per node in the previous layer, and the vector $\boldsymbol{b}$ is the same length as the number of nodes in the dense layer. Note that when writing exact weights for our hand-built models, the dimensions often make it more convenient to transpose the weights and bias, writing $\mathbf{W}^T$ and/or $\boldsymbol{b}^T$ instead.

A *convolution layer* is a layer where the traditional connections of a dense layer are replaced by one or more *kernels*, often also called *filters*. We will consider convolution layers in 2 dimensions, meaning that instead of an input vector $\boldsymbol{x}$, a 2-dimensional convolution layer $\mathcal{L}$ receives an input matrix $\mathbf{X}$ (typically representing an image) and returns an output matrix $\mathcal{L}(\mathbf{X})$. A kernel is a collection of weights also represented by a matrix $\mathbf{W}$ with dimensions smaller than or equal to the input size, which are determined by the user. Instead of matrix multiplication, the weights are combined with the input through *convolution*. A matrix $\mathbf{W}$ of chosen size $p \times q$ can be convoluted with an input matrix $\mathbf{X}$ of size $n \times m$ to produce an output matrix $\mathbf{Y}$ of size $(n - p + 1), (m - q + 1)$ through

$$\mathbf{Y}_{i,j} = \sum_{r=0}^{p-1} \sum_{s=0}^{q-1} \mathbf{X}_{i+r,j+s} \mathbf{W}_{r,s}.$$

The layer then adds a bias b and applies an activation function f, so the output of the layer is

$$\mathcal{L}(\mathbf{X})_{i,j} = f\left(\sum_{r=0}^{p-1} \sum_{s=0}^{q-1} \mathbf{X}_{i+r,j+s} \mathbf{W}_{r,s} + b \right).$$

For image processing, convolution layers have some notable benefits over dense layers:

- convolution applies to entries in $\mathbf{X}$ that are spatially close to each other, which are typically closely related in the context of an image. The output of a convolution layer is often called a *feature map*, with elements corresponding to specific image *features* identified by the kernel.

- The dimensions p, q of the weights matrix $\mathbf{W}$ are generally much smaller than the dimensions n, m of the image $\mathbf{X}$, which we expect to be large. Therefore, a convolution layer adds far fewer parameters to the model than would be required to perform matrix multiplication on the image $\mathbf{X}$, reshaped into a vector of size nm.

- As a single kernel adds relatively few parameters to the model, convolution layers can easily be expanded by adding additional kernels to explore and learn additional features. A convolution layer with multiple kernels evaluated simultaneously can be referred to as having multiple *channels*.

Often used in conjunction with convolution layers, a *max pooling layer* is a layer that takes an image or feature map $\mathbf{X} \in \mathbb{R}^{n \times m}$ and returns an output matrix $\mathcal{L}(\mathbf{X}) \in \mathbb{R}^{u \times v}$, where $u < n$ and $v < m$. The max pooling layer takes a sub-matrix of size $p \times q$ of $\mathbf{X}$ and returns the maximum element. The purpose of a max pooling layer is to reduce the size of the output from a convolution layer, as this is usually similar in size to that of the original image, whilst retaining the most relevant information. We will consider max pooling where each sub-matrix that pooling is applied to does not overlap, so the expected output will be of size $n/p \times m/q$. If u or v does not divide n or m respectively, the input may be *padded* with zeros so that the output is of size $\lceil n/p \rceil \times \lceil m/q \rceil$. Max pooling is applied to each channel from the convolution layer, so the output will still retain the information from each channel separately.

A *recurrent layer* is a layer that takes a sequence of inputs $\{\boldsymbol{x}_t\}_{t=1}^{T}$ and returns a sequence of outputs $\{\boldsymbol{y}_t\}_{t=1}^{T}$. At time t, the output of the recurrent layer depends on both the input $\boldsymbol{x}_t$, and the previous output $\boldsymbol{y}_{t-1}$. The layer uses two weights matrices: $\mathbf{W}$ to transform the input $\boldsymbol{x}_t$, and $\mathbf{U}$ to transform the previous output $\boldsymbol{y}_{t-1}$. the recurrent layer also has a bias $\boldsymbol{b}$ and activation function f. The output at time t is given as

$$\mathcal{L}(\mathbf{X})_t = \boldsymbol{y}_t = f(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{y}_{t-1} + \boldsymbol{b}),$$

where $\boldsymbol{y}_0$ is initialized as $\mathbf{0}$. Recurrent layers are designed to process sequences of inputs, illustrated in figure 1.2, and have been applied to problems such as time series forecasting and speech recognition (Graves et al., 2013). Note that when using recurrent layers, we have the option to return the whole sequence $\{\boldsymbol{y}_t\}_{t=1}^{T}$, or just the final output $\boldsymbol{y}_T$.

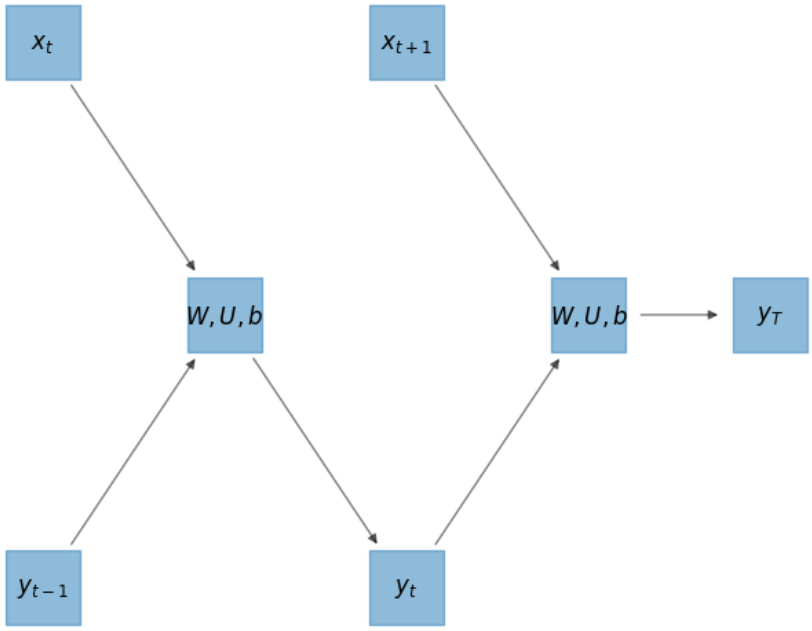

Figure 1.2: A diagram showing the last two calculations of a recurrent layer, from an input $\boldsymbol{x}_t$, to an output value $\boldsymbol{y}_T$ where $T = t + 2$. The same weights $\mathbf{W}$, $\mathbf{U}$, and bias $\boldsymbol{b}$ are used for every input in the sequence.

A *gated recurrent unit* (GRU), proposed by Cho et al. (2014), is a variation of the recurrent layer, that also takes a sequence of inputs $\{\boldsymbol{x}_t\}_{t=1}^{T}$ and returns a sequence of outputs $\{\boldsymbol{y}_t\}_{t=1}^{T}$. The GRU contains two gates: a *reset gate* which adjusts how much of the previous output is carried forward to propose a new output state, and a *update gate* which adjusts the input to be a combination of the previous output state and a new proposed output state. The output of the GRU is calculated through a series of steps outlined below.

- Calculate the reset amount,

$$\boldsymbol{r}_t = f(\mathbf{W}_r \boldsymbol{x}_t + \mathbf{U}_r \boldsymbol{y}_{t-1} + \boldsymbol{b}_r).$$

- Calculate the update amount,

$$\boldsymbol{z}_t = f(\mathbf{W}_z \boldsymbol{x}_t + \mathbf{U}_z \boldsymbol{y}_{t-1} + \boldsymbol{b}_z).$$

- Calculate the proposed output,

$$\hat{\boldsymbol{y}}_t = g(\mathbf{W}_y \boldsymbol{x}_t + \mathbf{U}_y (\boldsymbol{r}_t \odot \boldsymbol{y}_{t-1}) + \boldsymbol{b}_y),$$

  where $\odot$ denotes element-wise multiplication.

- Calculate the final output,

$$\mathcal{L}(\mathbf{X})_t = \boldsymbol{y}_t = \boldsymbol{z}_t \odot \boldsymbol{y}_{t-1} + (\mathbf{1} - \boldsymbol{z}_t) \odot \hat{\boldsymbol{y}}_t.$$

Note that we use two activation functions f and g. In this case we call f the *recurrent activation*. It is common for the output $\boldsymbol{y}_t$ to be referred to as the *hidden state* and notated as h_t or $\boldsymbol{h}_t$ because, when we consider only the final $\boldsymbol{y}_T$ as the output of the layer, we do not see the hidden states.

1.1.4 Training and Optimization

Suppose we have chosen a particular combination of layers to build a neural network with the aim of performing a given task. We now need a method of assigning the weights and biases associated with each layer in order for the network to perform the task well. In *supervised learning*, this uses a set of *training data* where inputs are paired with the corresponding desired output.

Suppose that we have training data $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^n$, where each $\boldsymbol{x}_i$ is a set input and each $\boldsymbol{y}_i$ is the true output. Given a network $\mathcal{N}$ with any valid initial weights, we can produce a predicted value for each input in the training data

$$\mathcal{N}(\boldsymbol{x}_i) = \hat{\boldsymbol{y}}_i.$$

We can evaluate how well the network is performing by introducing a *loss function* which gives us a real value that broadly indicates how well the predictions $\hat{\boldsymbol{y}} = \{\hat{\boldsymbol{y}}_i\}_{i=1}^n$ match the true outputs $\boldsymbol{y} = \{\boldsymbol{y}_i\}_{i=1}^n$. A loss function l is typically defined through a calculation involving $\hat{\boldsymbol{y}}$ and $\boldsymbol{y}$. However, given $\boldsymbol{y}$, we note that the predicted values $\hat{\boldsymbol{y}}$ are a function of the network weights and biases, which we will notate as $\mathbf{W}_\mathcal{N}$. Therefore, we treat the loss as a function of the weights, writing $l(\mathbf{W}_\mathcal{N}|\boldsymbol{y})$ instead of $l(\boldsymbol{y}, \hat{\boldsymbol{y}})$. The process of training then attempts to minimize the value of loss function l by adjusting the weights and biases $\mathbf{W}_\mathcal{N}$.

We have now set up a minimization problem for the function l with respect to its parameters $\mathbf{W}_\mathcal{N}$. A common way to solve minimization problems is to use some form of gradient descent, and we are able to do this with neural networks consisting of the layers we have introduced so far, provided the following conditions are also true:

- The loss function l is differentiable with respect to the parameters $\mathbf{W}_\mathcal{N}$.
- The activation functions within the network are all differentiable with respect to their inputs.

This ensures that, for each input $\boldsymbol{x}_i$ and target output $\boldsymbol{y}_i$, we can estimate the gradient of l at $\boldsymbol{x}_i$ by repeatedly applying the chain rule in a process known as *backpropagation*. Particular activation functions that satisfy this criteria which we will need notation for are the following:

- The *rectified linear unit* (ReLU) function, denoted by $(x)_+ = \max(0, x)$.
- The *softmax* function, denoted by $\sigma(\boldsymbol{x})$, is a vector function where $\sigma(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{j=1}^k e^{x_j}}$. This can be thought of as turning the values in $\boldsymbol{x}$ into "probabilities" that sum to 1. For example, in multinomial logistic regression,

the softmax function applied to the linear predictors does actually give us estimated probabilities.

- The *linear* activation function, which is equivalent to applying no activation function i.e. $f(x) = x$. If the activation function of a layer is not specifically mentioned, it can be assumed that it has the linear activation function.

There are several variants of gradient descent algorithms that are used to train neural networks. In this project we used Adam, which has generally been observed to perform well in comparison to other algorithms (Kingma and Ba, 2014). With Adam, the training process can be controlled through a variety of parameters:

- *Epochs*: The number of times the full set of training data is passed through during training.
- *Batch size*: The number of training samples between updating the gradient (direction of the adjustment applied to the weights).
- *Learning rate*: The step size for the adjustment to the weights in each iteration. This can be constant or can be scheduled to decay from a starting value with each epoch.

1.2 Dissertation Overview

This dissertation contains six further chapters. Each chapter will introduce a problem scenario in which we have implemented neural networks, the methods used to build and/or train the networks, and a discussion covering interesting points from each example. Each chapter has an accompanying Jupyter notebook to demonstrate the models that have been implemented working for a variety of input parameters. The Jupyter notebooks, and the underlying functions that constitute the main implementations of the neural networks, can be accessed through the Github at `https://github.com/HectorLeitch/mas6041`.

The problem scenarios covered in this dissertation are as follows:

- Chapter 2: Boolean functions. We investigate evaluating Boolean functions through neural networks. We extend a basic example with only two input variables, up to a generalization for $n \geq 2$ arguments.
- Chapter 3: Piece-wise linear functions. We investigate evaluating piecewise linear functions. Similarly to Chapter 2, we start with a basic example in one dimension and generalize it for $n \geq 1$ dimensions.
- Chapter 4: Searching for "zero" and "one". We demonstrate a method of searching for particular words in a text using a model that can cycle through a finite number of states.

- Chapter 5: Letter and word detection in images: We introduce detection of letters and words within a small image. We build models from the ground up with increased functionality at each stage.
- Chapter 6: Noughts and crosses: We investigate models that are trained to play noughts and crosses. The aim is to train a baseline model and then improve its architecture to more effectively and efficiently learn how to play.
- Chapter 7: Digit recognition in images: We tackle a popular problem of recognizing digits 0 to 9 within a small image. We investigate how elements of a hand-crafted model can be used as a guide to improving the effectiveness and efficiency of a trained model.

Each chapter includes discussion of the work done including the limitations, possible improvements, and ideas for further work. The project finishes with a short conclusion summarizing the common themes and ideas, in addition to some more general further applications that could be considered.

Across the selection of problem scenarios, this exploratory study examines:

- The extent to which small neural networks can be designed and mathematically deduced by hand to perform specific tasks, and what can be learned from the experience of applying this approach.
- The performance and efficiency of such networks when compared to networks that are using training data.
- The extent to which the insights from designing networks can be used to improve the performance and training time of neural networks that are using training data.

Ethics approval was not required because all the data was generated within the programs, as explained in appendix A.

1.3 Tensorflow and Keras

The models described in this project have been implemented using Tensorflow (Abadi et al., 2015), an open-source machine learning platform, and Keras (Chollet et al., 2015), a module for neural networks that brings Tensorflow's capabilities into Python. This was both extremely convenient and a very challenging aspect of the project.

Keras provides very high level functions and classes for implementing neural networks. This provides the convenience as layers can be added to a model using a single function and a model can be fully defined using a short block of code. However, this means that problems and errors are very hard to debug, as the messages tend to trace back through several layers of functions, often ending up returning something that cannot be understood by a relatively new

user. A common problem encountered, especially when coding the hand-built models, was setting the correct shape of objects defined in the code. Matrix multiplication of weights with an input could often be confused because the transpose of what was required had been mistakenly created. While the full code can be viewed through Github, a walk-through example is included in appendix B.

The core modules used and their versions can be found below.

- Tensorflow v2.9.1 (Abadi et al., 2015)
- Keras v2.9.0 (Chollet et al., 2015)
- Numpy v1.23.1 (Harris et al., 2020)
- Matplotlib v3.5.2 (Hunter, 2007)
- Pandas v2.0.3 (Wes McKinney, 2010)
- Itertools v3.12.3 (Van Rossum, 2020)

The version of Python used was 3.10.5.

Chapter 2

Boolean Functions

2.1 Introduction

In many settings such as computing, 1 and 0 are used to represent "True" and "False" respectively; in this case the value 1 or 0 is known as a *Boolean* value. In this section we will consider *Boolean functions*, defined as functions from $\{0,1\}^n$ to $\{0,1\}$ for some positive integer n.

Logical operators are familiar examples of Boolean functions used in programming. These include:

- The identity: $f : \{0,1\} \to \{0,1\}$ defined by $f(x) = x$.
- "Not": $f : \{0,1\} \to \{0,1\}$ defined by $f(x) = 1 - x$.
- "And": $f : \{0,1\}^2 \to \{0,1\}$ defined by $f(x,y) = xy$.
- "Or": $f : \{0,1\}^2 \to \{0,1\}$ defined by $f(x,y) = \max(x,y)$.

Boolean functions are commonly defined through *truth tables*. A truth table for a Boolean function f is a tabulation of all the possible inputs $x \in \{0,1\}^n$ and the corresponding outputs $f(x)$. For the models in this chapter, the Python function `itertools.product()` to generate the possible set of inputs $\{0,1\}^n$. The nested loops cycle like an odometer with the rightmost element advancing on every iteration. Therefore, we can assume a list of outputs $(u_0, \ldots, u_{2^n-1})$ corresponds to the list of inputs given by this ordering, uniquely defining f.

This section will start by demonstrating hand-built neural networks to calculate any Boolean function $f : \{0,1\}^n \to \{0,1\}$ given the outputs $(u_0, \ldots, u_{2^n-1})$. We then build neural networks to calculate *all* Boolean functions $f : \{0,1\}^n \to \{0,1\}$ simultaneously, given n. We then consider how the architecture of particular networks may be adapted to increase their efficiency. Note, that we consider the case $n = 1$ to be trivial: the possible Boolean functions are the constant functions $f(x) = 0$ and $f(x) = 1$, plus the logical operators "not" and the identity, which

can both clearly be calculated in a neural network with one input, one output, and no activation function.

2.2 Methods

The methods described in this section are implemented in the file `https://github.com/HectorLeitch/mas6041/blob/main/code/bool.py`.

2.2.1 A 2-input Model

We start by building a neural network with mathematically deduced weights for the case where $n = 2$. Therefore, we specify a function using a list of required outputs (u_0, u_1, u_2, u_3) and the goal is to build a neural network to calculate $f(x, y)$ given $(x, y) \in \{0, 1\}^2$, by calculating and manually assigning the weights of the network.

As we have set $n = 2$, we have $2^n = 4$ elements in a given output list (u_0, u_1, u_2, u_3). According to the ordering from `itertools.product()`, this implies that $f(0, 0) = u_0, f(0, 1) = u_1, f(1, 0) = u_2$ and $f(1, 1) = u_3$.

We find [1] that

$$f(x, y) = u_0 + (u_2 - u_0)\, x + (u_1 - u_0)\, y + (u_3 - u_2 - u_1 + u_0)\, xy. \tag{2.1}$$

This can be computed in a dense layer $\mathcal{L}^{[2]}$, with (x, y, xy) as inputs, as

$$f(x, y) = \begin{pmatrix} u_2 - u_0 \\ u_1 - u_0 \\ u_3 - u_2 - u_1 + u_0 \end{pmatrix}^T \begin{pmatrix} x \\ y \\ xy \end{pmatrix} + u_0. \tag{2.2}$$

To map the inputs (x, y) to (x, y, xy), we note that

$$x, y \in \{0, 1\} \implies (x + y - 1)_+ = xy. \tag{2.3}$$

This implies a dense layer $\mathcal{L}^{[1]}$ with ReLU activation function, which is computed using the weights and bias

$$\begin{pmatrix} x \\ y \\ xy \end{pmatrix} = \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \right)_+ . \tag{2.4}$$

[1] This result was kindly provided by my supervisor in a set of notes introducing me to this problem.

The full model is then given by

$$f(x,y) = \mathcal{L}^{[2]}\left(\mathcal{L}^{[1]}(x,y)\right).$$

2.2.2 Generalization to Higher Input Dimensions

To generalize this model to any positive integer n, we generalize equations 2.3 and 2.1.

First, we generalize equation 2.3 corresponding to the first layer. Suppose that $x_1, ..., x_n \in \{0,1\}^n$ for any $n \geq 2$. Then

$$\prod_{i=1}^{n} x_i = \left(\sum_{i=1}^{n} x_i - (n-1)\right)_+. \tag{2.5}$$

Therefore, we can map $(x_1, \ldots, x_n)$ to the product $\prod_{j\in J} x_j$ for any subset $J \subseteq \{1, \ldots, n\}$ in a dense layer with ReLU activation function. In the first layer of the generalized model, we will map $(x_1, \ldots, x_n)$ to all such products to give us $(x_1, \ldots, x_n, x_1x_2, x_1x_3, \ldots, \prod_{i=1}^{n} x_i)$ as input to the second layer.

To generalize 2.1, first note that we can rewrite it with $(x,y) = (x_1, x_2)$ as

$$f(x_1, x_2) = a_0 + a_1x_1 + a_2x_2 + a_3x_1x_2,$$

where

$$\begin{aligned} a_0 &= u_0, \\ a_1 &= u_2 - u_0, \\ a_2 &= u_1 - u_0, \\ a_3 &= u_3 - u_2 - u_1 + u_0. \end{aligned}$$

Notice that

$$\begin{aligned} a_0 &= f(0,0) \\ a_1 &= f(1,0) - f(0,0), \\ a_2 &= f(0,1) - f(0,0), \\ a_3 &= f(1,1) - f(1,0) - f(0,1) - f(0,0). \end{aligned}$$

This pattern suggests that we can write $f(x_1, ..., x_n)$ as a polynomial where the coefficients $a_0, ..., a_{2^n-1}$ will follow from applying the Inclusion-Exclusion Principle, described in appendix C. Adjusting the notation, suppose we have the coefficients u_I for subsets $I \subseteq N = \{0, \ldots, n-1\}$ and we wish to derive the corresponding coefficients a_I. From the Inclusion-Exclusion Principle, we set

$$a_I = \sum_{J\subseteq I} (-1)^{|I\setminus J|} u_J.$$

Algorithm 1 Algorithm for determining coefficients

Require: $(u_0, \ldots, u_{2^n-1})$
  for $j = 1, \ldots, 2^n$ **do**
    Set $\hat{a_j} = u_j$
    Let $X_j = (x_1, \ldots, x_n)$ s.t. $f(x_1, \ldots, x_n) = u_j$
    for $k = 1, \ldots, j-1$ **do**
      Let $X_k = (x_1, \ldots, x_n)$ s.t. $f(x_1, \ldots, x_n) = u_k$
      if $(X_k)_i \leq (X_j)_i \forall i \in \{1, \ldots, n\}$ **then**
        Set $\hat{a_j} = \hat{a_j} - a_k$
      end if
    end for
    Set $a_j = \hat{a_j}$
  end for
  return bias u_0, weights $(a_1, \ldots, a_{2^n-1})$

For example, when $n = 2$ we have $a_{\{0,1\}} = u_{\{0,1\}} - u_{\{0\}} - u_{\{1\}} + u_{\emptyset}$. This yields the coefficients $a_0, ..., a_{2^n-1}$ through mapping the power set $\mathcal{P}(N)$ to $\{0, \ldots, 2^n - 1\}$. In the code, we have reached the same coefficients through Algorithm 1.

With the output from the first layer, we can compute this polynomial in a single dense layer with no activation function, similar to equation 2.2. The overall model expression in relation to the layers is therefore unchanged.

2.2.3 Simultaneous Calculation

The above model produces a network to build f given a list of 2^n output values $(u_0, ..., u_{2^n-1})$. With a relatively small modification to the network, we can adapt the process of building it to instead build all possible Boolean functions simultaneously given n. As each Boolean function is specified with a set of 2^n values in $\{0, 1\}$, there are 2^{2^n} possible Boolean functions given n. This can be thought of as a function $F : \{0, 1\}^n \to \{0, 1\}^m$ where $m = 2^{2^n}$.

Therefore, we need to modify the network architecture so that the second layer has m outputs. As we have shown that the first layer produces all the values we need for any Boolean function, we do not need to modify the first layer.

Recall that in equation 2.2, we wrote the second layer calculation in the form

$$f(\boldsymbol{x}) = \mathbf{W}\boldsymbol{x}^* + \boldsymbol{b},$$

where $\boldsymbol{x}^*$ is the output of the first layer. The weights matrix in the 2-input example is a row vector

$$\begin{pmatrix} u_0 \\ u_2 - u_0 \\ u_1 - u_0 \\ u_3 - u_2 - u_1 + u_0 \\ \cdots \end{pmatrix}^T,$$

which we could consider as a matrix with 2^n columns and 1 row. If we replace each entry in the list $(u_0, ..., u_{2^n-1})$ with a column vector in $\mathbb{R}^m$, we can calculate the rows of the weights matrix as in algorithm 1 using vector addition/subtraction. Similarly, the bias is set to $\boldsymbol{u}_0$.

2.3 Results

The models described above are demonstrated in the Jupyter notebook `https://github.com/HectorLeitch/mas6041/blob/main/try_bool.ipynb`. The notebook is split into examples to demonstrate the code, which are described in the following paragraphs.

Example 2.1 The function `make_u_model(u=)` builds a 2-input model from a list $\boldsymbol{u} \in \{0,1\}^4$. The model evaluates $f(x)$ for a given $x \in \{0,1\}^2$.

We also verify that the model weights for each layer are as described in section 2.2.1. We note that the weights shown are the transpose of the weights that we have described, as the matrix multiplication for an input and weights matrix tends to be calculated as $\boldsymbol{x}^T\mathbf{W}$ rather than $\mathbf{W}\boldsymbol{x}$.

Example 2.2 The function `make_u_model(u=)` can also build n-input models as described in section 2.2.2 from a list $\boldsymbol{u} \in \{0,1\}^{2^n}$. The code is set up to randomly generate $\boldsymbol{u}$ where each entry is 0 or 1 with probability 1/2. We note that for a "practical" run time (i.e. < 1 minute), n should be somewhere between 2 and 8.

The model summary is shown to verify the number of parameters in each layer, which should be $(n+1)*(2^n-1)$ for the first layer, and 2^n for the second layer.

We verify that this model correctly evaluates $f(\boldsymbol{x})$ for *all* 2^n possible values of $\boldsymbol{x}$, using the model's `evaluate()` method, yielding a total loss of 0 and an accuracy of 1.

Example 2.3 The function `make_n_model(n=)` builds a full model to evaluate all possible Boolean functions given n, as described in section 2.2.3. We note that due to the number of parameters in the model, this can only be done in Python for $n = 2, 3, 4$.

The model summary is shown to verify that the number of parameters in each layer. In this example, there should be $(n+1)*(2^n-1)$ for the first layer, and 2^{2^n+n} for the second layer.

We verify that this model correctly evaluates *all* $f(\boldsymbol{x})$ for all 2^n possible values of $\boldsymbol{x}$, using the model's `evaluate()` method, again yielding a total loss of 0 and an accuracy of 1.

2.4 Discussion

2.4.1 Model Architecture

In this chapter we only used model architectures with two layers, which we found to be sufficient for all the models implemented due to results 2.3 and its generalized form 2.5. Equation 2.3 is of particular interest as for two Boolean values x and y, the product xy is equal to $(x \wedge y)$, and this can be computed in just a single ReLU layer. We will see this result used often in later hand-built models.

Within the first layer, the weights were simple to assign and were independent of the Boolean function to be computed. On the other hand, the second layer had a much higher computational and memory cost, both in assigning the weights and evaluating the network. The algorithm we used to assign the weights may not have been very efficient, and grows in cost at a rate of at least $\mathcal{O}(2^n)$. For the model described in section 2.2.3, the number of weights in the second layer also grew quickly with n, which limited the Python implementation to small values of n.

2.4.2 Possible Improvements

For a given f, it may be possible to adapt the network architecture to be more efficient using the values in $\boldsymbol{u}$. In the first layer, we only considered products of the input variables. For a Boolean function in 2 variables, it may be possible to find representations of f that use other combinations of the input variables and ReLU activation function in the first layer. For example, the "exclusive or" function can be written as

$$f(x, y) = (x - y)_+ - (y - x)_+ .$$

This slightly reduces the number of weights required as the first layer only needs 2 outputs.

Another possible avenue for building these kinds of functions is to use combinations of logic gates. A logic gate is a device that performs a Boolean function, with the most commonly used example being electrical transistor gates in computer chips. The "Not And" (NAND) gate can be written as $\neg(x \wedge y) = 1 - xy = 1 - (x + y - 1)_+$. This is a *universal* logic gate as it can be used to create all other logical gates.

For example, the function $f(\boldsymbol{x}) = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$ can be summarized as

$$f(\boldsymbol{x}) = 1 - \big(1 - \big((x_1 + x_2 - 1)_+\big) + \big(1 - (x_3 + x_4 - 1)_+\big) - 1\big)_+ .$$

This would require three total NAND gates, which could be spread over four layers. It is not immediately clear whether this would be more efficient, but one benefit of allowing the network structure to change based on $\boldsymbol{u}$ seems to be that

we would not allocate memory for parameters that end up being 0 in the models we have implemented. This would especially be true for functions that can very easily be reduced into smaller logic gates.

Chapter 3

Piece-wise Linear Functions

3.1 Introduction

Piece-wise linear functions are a common tool for approximation, and can arise in applications such as linear interpolation of a data set or local regression models such as locally estimated scatterplot smoothing (LOESS) (Cleveland, 1979). This chapter explores to what extent neural networks are able to calculate piece-wise linear functions.

We consider a function $f : \mathbb{R}^d \to \mathbb{R}$ to be a *piece-wise linear function* if it is a function with finitely many corners, and between each of the corners the function is linear. For $d = 1$, that is to say that there are real numbers $a_1, \ldots, a_r$, gradients $m_0, \ldots, m_r$, and an intercept c such that:

- $f(x) = c + m_0 x$ on $(-\infty, a_1]$.
- $f(x)$ has gradient m_i on $[a_i, a_{i+1}]$ for $i = 1, \ldots, r-1$.
- $f(x)$ has gradient m_r on $[a_r, \infty)$.

From these criteria we can also define a unique piece-wise linear function by giving the values $a_1, \ldots, a_r$, $m_0, \ldots, m_r$, and c.

For general $d \geq 1$, we can at first restrict ourselves to cases where the corners form products of d intervals, or "boxes" (for example rectangles in $d = 2$), and so we require corners $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_r$, and vector gradients $\boldsymbol{m}_0, \ldots, \boldsymbol{m}_{r^*}$ in $\mathbb{R}^d$, and an intercept $c \in \mathbb{R}$ such that:

- $f(\boldsymbol{x}) = c + \boldsymbol{m}_0^T \boldsymbol{x}$ where $x_i \leq a_{1,i}$ for $i = 1, \ldots, d$.
- $f(\boldsymbol{x})$ has gradient $\boldsymbol{m}_i$ on the box with the "bottom" corner $\boldsymbol{a}_i$.
- $f(\boldsymbol{x})$ has gradient $\boldsymbol{m}_{r^*}$ where $x_i \geq a_{r,i}$ for $i = 1, \ldots, d$.
- The gradients $\boldsymbol{m}_0, \ldots, \boldsymbol{m}_{r^*}$ are such that f is continuous. We will ensure this through constructing the corners and gradients as the Cartesian

product of vectors of real numbers $(a_{1,1}, \ldots, a_{1,r_1}), \ldots, (a_{d,1}, \ldots, a_{d,r_d})$ and $(m_{1,0}, \ldots, m_{1,r_1}), \ldots, (m_{d,0}, \ldots, m_{d,r_d})$ respectively.

This section will start by implementing hand-crafted neural networks to evaluate linear piece-wise functions. We then experiment with training neural networks to determine how well they are able to approximate f and whether they arrive at the same model as the hand-crafted ones when using the same architecture.

3.2 Methods

The methods described in this section are implemented in the file `https://github.com/HectorLeitch/mas6041/blob/main/code/piecewise.py`.

3.2.1 A 1-input Model

We start by building a neural network with mathematically deduced weights for the case where $d = 1$. Therefore, we specify a function $f : \mathbb{R} \to \mathbb{R}$ given two lists of real numbers $(a_1, \ldots, a_r)$ and $(m_0, \ldots, m_r)$, and a real number c, as described above. The goal is to build a neural network to calculate $f(x)$ given $x \in \mathbb{R}$.

To determine a suitable network architecture, we can think of evaluating $f(x)$ by starting at $x = -\infty$, and "following" the gradient m_0 until we reach $x = a_1$. We then follow the gradient m_1 until we reach $x = a_2$, and so on, until we reach the value of x we wish to evaluate $f(x)$ at (stopping earlier if $x < a_2$ or $x < a_1$), and then adding the constant c. With this in mind, we can write

$$\begin{aligned} f(x) = c &+ m_0 \max(x, a_1) \\ &+ m_1 \max(x - a_1, a_2 - a_1)\, I(x \geq a_1) + \\ &\ldots \\ &+ m_{r-1} \max(x - a_{r-1}, a_r - a_{r-1})\, I(x \geq a_{r-1}) \\ &+ m_r (x - a_r)\, I(x \geq a_r), \end{aligned} \tag{3.1}$$

where I is the indicator function. In this expression we have a mixture of linear multiplication, $\max()$ functions, and indicator functions; to be able to calculate this using a neural network we need to rewrite the $\max()$ functions and indicator functions. Recalling the definition of the ReLU activation function, we can write

$$(x)_+ = \max(0, x) = xI(x \geq 0),$$

this points us towards a strategy for rewriting equation 3.1. We find that

$$\begin{aligned} f(x) &= c + m_0 x + m_1 (x - a_1)\, I(x \geq a_1) - m_0 (x - a_1)\, I(x \geq a_1) + \\ &\qquad \ldots + m_r (x - a_r)\, I(x \geq a_r) - m_{r-1} (x - a_r)\, I(x \geq a_r) \\ &= c + m_0 x + \sum_{i=1}^{r} (m_i - m_{i-1})(x - a_i)\, I(x - a_i \geq 0) \\ &= c + m_0 x + \sum_{i=1}^{r} (m_i - m_{i-1})(x - a_i)_+ . \end{aligned} \tag{3.2}$$

We can now construct a neural network with two dense layers to evaluate this formula. The first layer $\mathcal{L}^{[1]}$ will be a dense layer to map x to the vector

$$\begin{pmatrix} x \\ -x \\ x - a_1 \\ \dots \\ x - a_r \end{pmatrix}_+ .$$

The first two elements $(x)_+$ and $(-x)_+$ will be combined in the second layer as

$$\begin{aligned} (x)_+ - (-x)_+ &= x \\ \implies m_0 (x)_+ - m_0 (-x)_+ &= m_0 x, \end{aligned} \tag{3.3}$$

allowing us to calculate the $m_0 x$ term. Therefore, we can fully define the first layer $\mathcal{L}^{[1]} : \mathbb{R} \to \mathbb{R}^{r+2}$ as

$$\mathcal{L}^{[1]}(x) = (\mathbf{W}_1 x + \boldsymbol{b}_1)_+ ,$$

where

$$\mathbf{W}_1 = \begin{pmatrix} 1 \\ -1 \\ 1 \\ \dots \\ 1 \end{pmatrix} \in \mathbb{R}^{r+2},$$

and

$$\boldsymbol{b}_1 = \begin{pmatrix} 0 \\ 0 \\ -a_1 \\ \dots \\ -a_r \end{pmatrix} \in \mathbb{R}^{r+2}.$$

The second layer carries out the simple linear multiplication and addition of the constant, and so we use a dense layer with the linear activation function to achieve this. To assign the weights we simply combine equations 3.3 and 3.2, adding the constant c as bias. Therefore, the second layer $\mathcal{L}^{[2]} : \mathbb{R}^{r+2} \to \mathbb{R}$ is

$$\mathcal{L}^{[2]}(\boldsymbol{x}) = \mathbf{W}_2 \boldsymbol{x} + \boldsymbol{b}_2,$$

where

$$\mathbf{W}_2^T = \begin{pmatrix} m_0 \\ -m_0 \\ m_1 - m_0 \\ \dots \\ m_r - m_{r-1} \end{pmatrix} \in \mathbb{R}^{r+2},$$

and

$$\boldsymbol{b}_2 = c,$$

noting that the bias vector $\boldsymbol{b}_2$ is a single real value as there is only one output, which will be the case throughout this chapter.

3.2.2 Generalization to Higher Input Dimensions

Now suppose that we want our neural network to evaluate $f : \mathbb{R}^2 \to \mathbb{R}$, where f is defined by the points $(a_{1,i}, a_{2,j})$ where $i = 1, \ldots, r_1$ and $j = 1, \ldots, r_2$, gradients $(m_{1,i}, m_{2,j})$ where $i = 0, \ldots, r_1$ and $j = 0, \ldots, r_2$, and a constant c. Now f has gradients $(m_{1,i}, m_{2,j})$ on rectangles $[a_{1,i}, a_{1,i+1}] \times [a_{2,j}, a_{2,j+1}]$ for $i = 1, \ldots, r_1 - 1$ and $j = 1, \ldots, r_2 - 1$. Similar to the 1-dimensional case when the arguments tend to $\pm\infty$, it has gradients:

- $(m_{1,0}, m_{2,j})$ for regions $(-\infty, a_{1,1}] \times [a_{2,j}, a_{2,j+1}]$ where $j = 1, \ldots, r_2 - 1$
- $(m_{1,r_1}, m_{2,j})$ for regions $[a_{1,r_1}, \infty) \times [a_{2,j}, a_{2,j+1}]$ where $j = 1, \ldots, r_2 - 1$
- $(m_{1,i}, m_{2,0})$ for regions $[a_{1,i}, a_{1,i+1}] \times (-\infty, a_{2,1}]$ where $i = 1, \ldots, r_1 - 1$
- $(m_{1,i}, m_{2,r_2})$ for regions $[a_{1,i}, a_{1,i+1}] \times [a_{2,r_2}, \infty)$ where $i = 1, \ldots, r_1 - 1$

Constructing the corner points and gradients as cross products of 2 ($= d$) sets of points and gradients in $\mathbb{R}$ means that we can break down the evaluation of f in the same way as the 1-dimensional case. We find that

$$\begin{aligned} f(x_1, x_2) = c + m_{1,0}x_1 + \sum_{i=1}^{r_1} (m_{1,i} - m_{1,i-1}) (x_1 - a_{1,i})_+ \\ + m_{2,0}x_2 + \sum_{j=1}^{r_2} (m_{2,i} - m_{2,i-1}) (x_2 - a_{2,j})_+ \end{aligned} \tag{3.4}$$

We can therefore construct our layers with the same architecture as the 1-dimensional case, but expand the dimensions of the weights and biases so $\mathcal{L}^{[1]} : \mathbb{R}^2 \to \mathbb{R}^{r_1+r_2+4}$ and $\mathcal{L}^{[2]} : \mathbb{R}^{r_1+r_2+4} \to \mathbb{R}$. The weights and biases $\mathbf{W}_1, \boldsymbol{b}_1, \mathbf{W}_2$, and $\boldsymbol{b}_2$ are as follows:

$$\mathbf{W}_1^T = \begin{pmatrix} 1 & -1 & 1 & \ldots & 1 & 0 & 0 & 0 & \ldots & 0 \\ 0 & 0 & 0 & \ldots & 0 & 1 & -1 & 1 & \ldots & 1 \end{pmatrix}$$

$$\boldsymbol{b}_1^T = \begin{pmatrix} 0 & 0 & -a_{1,1} & \ldots & -a_{1,r_1} & 0 & 0 & -a_{2,1} & \ldots & -a_{2,r_2} \end{pmatrix}$$

$$\mathbf{W}_2^T = \begin{pmatrix} m_{1,0} \\ -m_{1,0} \\ m_{1,1} - m_{1,0} \\ \ldots \\ m_{1,r_1} - m_{1,r_1-1} \\ m_{2,0} \\ -m_{2,0} \\ m_{2,1} - m_{2,0} \\ \ldots \\ m_{2,r_2} - m_{2,r_2-1} \end{pmatrix}$$

$$\boldsymbol{b}_2 = c$$

We can generalize the extension from $d = 1$ to $d = 2$ to a function $f : \mathbb{R}^d \to \mathbb{R}$ for any $d \geq 1$, which is evaluated in the same way through the sum

$$f(\boldsymbol{x}) = c + \sum_{j=1}^{d} \left\{ m_{j,0} x_j + \sum_{i=1}^{r_j} (m_{j,i} - m_{j,i-1}) (x_j - a_{j,i})_+ \right\}.$$

This gives us layers $\mathcal{L}^{[1]} : \mathbb{R}^d \to \mathbb{R}^{r^*}$ and $\mathcal{L}^{[2]} : \mathbb{R}^{r^*} \to \mathbb{R}$, where $r^* = 2d + \sum_{i=1}^{d} r_i$. The weights are

$$\mathbf{W}_1^T = \begin{pmatrix} 1 & -1 & 1 & \dots & 1 & \dots & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 0 & \dots & 1 & -1 & \dots & 1 \end{pmatrix} \in \mathbb{R}^{d \times r^*}$$

$$\boldsymbol{b}_1^T = \begin{pmatrix} 0 & 0 & -a_{1,1} & \dots & -a_{1,r_1} & \dots & 0 & 0 & -a_{d,1} & \dots & -a_{d,r_2} \end{pmatrix} \in \mathbb{R}^{1 \times r^*}$$

$$\mathbf{W}_2^T = \begin{pmatrix} m_{1,0} \\ -m_{1,0} \\ m_{1,1} - m_{1,0} \\ \dots \\ m_{1,r_1} - m_{1,r_1-1} \\ \dots \\ m_{d,0} \\ -m_{d,0} \\ m_{d,1} - m_{d,0} \\ \dots \\ m_{d,r_d} - m_{d,r_d-1} \end{pmatrix} \in \mathbb{R}^{r^* \times 1}$$

$$\boldsymbol{b}_2 = c \in \mathbb{R}.$$

3.2.3 Training an approximate model

For an experiment to test how well a model with this layer architecture is able to learn these weights through training, we fix the function $f : \mathbb{R} \to \mathbb{R}$ using the values

$$\begin{aligned} (a_1, a_2, a_3, a_4) &= (-5, -2, 2, 5) \\ (m_0, m_1, m_2, m_3, m_4, m_5) &= (3, -1, 2, -1, -3) \\ c &= 1. \end{aligned}$$

A graph of this function produced by the exact 1-dimensional model is shown in figure 3.1.

The exact model is constructed according to section 3.2.1. The approximate model is initialized by cloning the exact model, just to replicate the layer architecture, and compiling the model. Compiling the model does three things: Firstly, it initializes random weights and biases from which to begin the training.

Secondly, it allows us to assign the loss function for training; this is set to the mean squared error (MSE) defined as

$$MSE\left(\boldsymbol{y}, \hat{\boldsymbol{y}}\right) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2.$$

Thirdly, it allows us to assign the optimizer for the training procedure, which is Adam as discussed in section 1.1.4. This assignment includes setting the learning rate. The experiment was repeated with various learning rates until one which seemed to work reasonably well was found.

Training data for the model was created by evaluating the exact model on 100 points evenly distributed over the interval $[-10, 10]$, ensuring there were no parts of the domain where data was unavailable.

The model was essentially allowed to train for as many epochs as was required by setting the number of epochs to 10000, and using an early stopping criterion which monitored the loss, stopping training if no improvement had been observed for 20 epochs. The batch size was left as the default value of 32.

3.3 Results

The models described above are demonstrated in the Jupyter notebook `https://github.com/HectorLeitch/mas6041/blob/main/try_piecewise.ipynb`. The notebook is split into examples to demonstrate the code, which are described in the following paragraphs.

Example 3.1 The function `make_oned_model(a=,m=,c=)` builds a 1-dimensional model from two lists of real numbers $(a_1, \ldots, a_r)$ and $(m_0, \ldots, m_r)$, and a real number c.

Figure 3.1 shows an example of the plot produced by the exact model with the values

$$\begin{aligned}(a_1, a_2, a_3, a_4) &= (-5, -2, 2, 5)\\ (m_0, m_1, m_2, m_3, m_4) &= (3, -1, 2, -1, -3)\\ c &= 1.\end{aligned}$$

This is the same function that is used as the basis for training the approximate model.

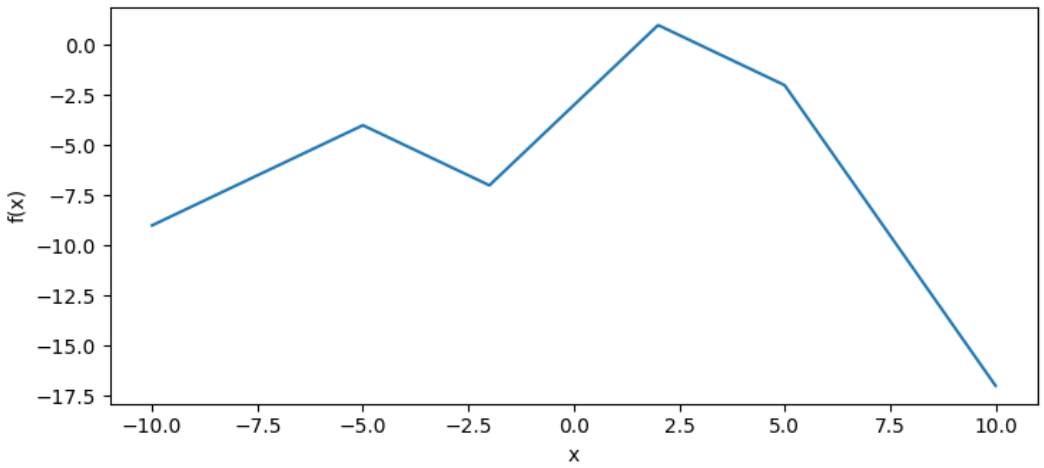

Figure 3.1: A plot produced using the exact model described in section 3.2.1. The plot was produced by evaluating the network on a set of 1000 points evenly spaced between the limits, as is commonly done so in programming to produce a line plot of a function.

Example 3.2 The function `make_nd_model(a=,m=,c=)` builds a d-dimensional model from two lists and a constant. Each of the two lists are separated into d lists representing the locations of the corners and the slopes in each dimension.

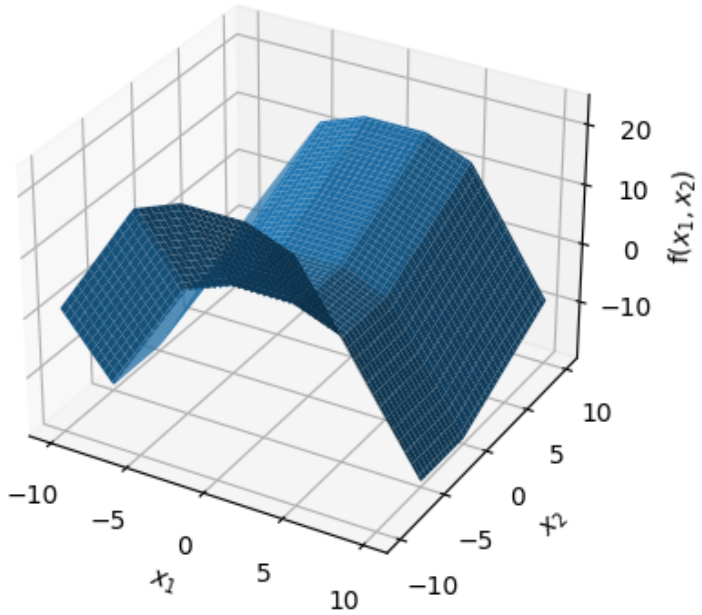

Figure 3.2: A plot produced using the exact model described in section 3.2.2. In this case, $d = 2$. The network was again evaluated at each point in a grid, formed by crossing two sets of 1000 points between the limits for each axis.

Figure 3.2 shows an example of a plot produced by the exact model with the values

$$\begin{aligned}(a_{1,1}, a_{1,2}, a_{1,3}, a_{1,4}) &= (-5, -2, 2, 5)\\ (a_{2,1}, a_{2,2}, a_{2,3}) &= (-5, 0, -5)\\ (m_{1,0}, m_{1,1}, m_{1,2}, m_{1,3}, m_{1,4}) &= (4, 1, 0, -1, -4)\\ (m_{2,0}, m_{2,1}, m_{2,2}, m_{2,3}) &= (4, 1, 0, -1)\\ c &= 1.\end{aligned}$$

Example 3.3 The function `make_approx_model(N=,r=)` initializes an approximate 1-dimensional model using an existing exact model and a learning rate. In this example we use the values from section 3.2.3 to create the exact model. It was found that a suitable value for the learning rate was 0.05.

The function `train_approx_model(M=,N=,samples=,epochs=)` was used to train the approximate model using the initialized model, exact model, and values for the number of samples to generate in the training data and the maximum number of epochs to train for.

It was found that the model was able to replicate the function f to various degrees of accuracy over different runs. Figure 3.3 shows a collection of plots produced by the trained approximate model versus the given exact model.

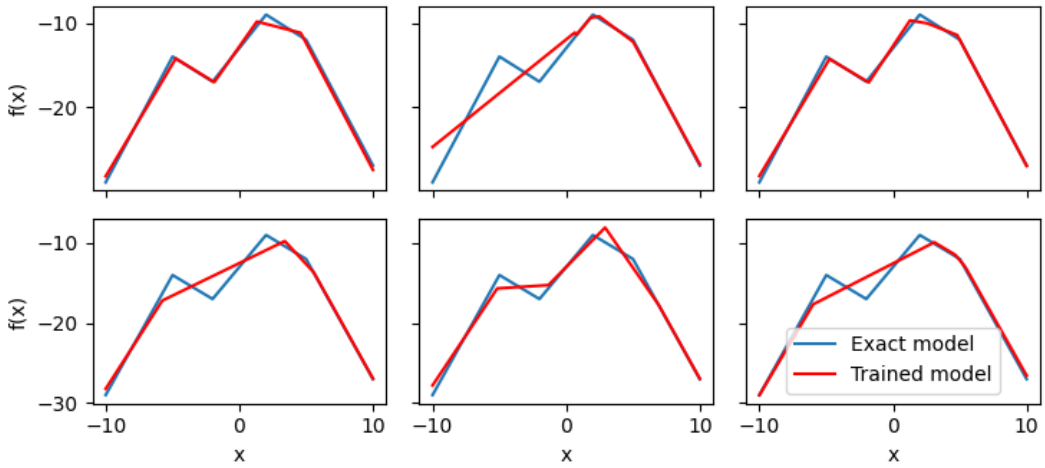

Figure 3.3: A collection of models trained as described in section 3.2.3. We can see that some of them are nearly identical to one another, suggesting that almost the same local minimum was reached during training.

Examining the weights of the trained models, it can be seen that where it is able to match the exact model, the trained model replicated the same weights up to multiplication and division by constants that cancel out between layers to arrive at the same result. This was only true for the intervals where a match was achieved. In some cases it can be seen that the model missed some of the

corners and replaced multiple intervals with a single slope. When this happened, there were weights that were set close to 0 across the layers, which caused corner points to disappear from the resulting plots.

3.4 Discussion

3.4.1 Exact Models

For the generalization we applied in this chapter, it was required that we divided the domain into (hyper-)rectangles so that we could write f in the form of equation 3.4. We could specify other continuous functions such as $f(x, y) := (x + y)_+$. This function can clearly be computed by a neural network with just one dense layer with a ReLU activation function, and could be considered to be piece-wise linear as the slopes are constant on two regions separated by the boundary along $y = -x$. In this case, we could still use equation 3.4 by inserting a dense layer to rotate the input 45 degrees clockwise from (x, y) to $\left(\frac{\sqrt{2}(x+y)}{2}, \frac{\sqrt{2}(y-x)}{2}\right)$, and then evaluating $f(x, y)$ as

$$f(x, y) = \left(\sqrt{2}\left(\frac{\sqrt{2}\,(x+y)}{2}\right) + 0\left(\frac{\sqrt{2}\,(y-x)}{2}\right)\right)_+ .$$

It is possible to use this technique in any case where the domain is divided into (hyper-)rectangles with some linear transformation applied.

Another case of interest to consider would be to divide $\mathbb{R}^2$ into triangles, as this is a technique that is used to draw surfaces in computer graphics. The technique we have used would not work in this case, as the three boundaries of a triangle would mean that at least one of them is dependent on both variables, for any linear transformation that could be applied.

Overall, the cases in which we are able to build a model are very restricted and it did not seem that we would easily be able to implement more flexible models without spending time researching the properties of piece-wise functions, moving away from the focus of the project.

3.4.2 Approximate Models

The aim of the training experiment was to see how well a model with the same layer architecture could replicate the exact model weights and biases. Visually, it was clear that the model was exactly or nearly-exactly matching the exact model in some cases. As the weights also matched the exact model in these cases, we can be satisfied that the exact structure could be found through training.

As this result was confirmed with little effort put into optimizing the training, it could be possible that there were factors that we did not consider that would

have improved the reliability of the training in order to match the exact model more often, such as adding a learning rate schedule or modifying the batch size. It appears that the random initialization may have caused the model to converge to local minima in some cases; further experimentation would be needed to identify these and determine whether they could be avoided in favour of the global minimum (exact match).

Optimization of the training process may have been more relevant if this experiment was continued to 2- or 3-input models to see if increasing the number of parameters prevented the model from replicating the exact structure.

Chapter 4

Searching for "zero" and "one"

4.1 Introduction

This chapter will consider the task of searching for particular words ("zero" and "one") in a string of text. A typical way to process text using neural networks is to use a word tokenizer, where each word is assigned an integer v from 1 to V, where V is the total size of the *vocabulary* we have used during training. This integer is then translated into a *one-hot encoded* vector, where the vector is a zero vector of length V, apart from the v-th entry which is set to 1. Applying a word tokenizer to the input would make searching for particular words in a given text string a trivial task, so the aim of the models implemented in this chapter is to take the input one character at a time.

In general, the neural networks in this chapter will be set up to take a sequence of one-hot encoded inputs $\{\boldsymbol{x}_t\}_{t=1}^T$ representing a string of characters, and produce a sequence of one-hot encoded outputs $\{\boldsymbol{y}_t\}_{t=1}^T$, each of length 2, corresponding to the completion of a detected word at input $\boldsymbol{x}_t$. For any positive integer m, let $o_m : \{1, \ldots, m\} \to \{0,1\}^m$ denote one-hot encoding, where the one-hot encoded vector $o_m(n)$ in $\{0,1\}^m$ is 1 in the n-th position and 0 everywhere else. Also if $n = 0$ or no "detection" has been made, we set the output will be a zero vector of length m.

The overall strategy is to replicate a finite state machine, which reads each character one by one and changes between a finite number of *states* according to the inputs received, with some of these states triggering an output when a word is detected.

4.2 Methods

The methods described in this section are implemented in the file `https://github.com/HectorLeitch/mas6041/blob/main/code/zeroone.py`.

4.2.1 Small Input Model

We start by building a network to detect two words: "zero" and "one". For simplicity, we assume that all punctuation has been removed from the text. To reduce the size of the one-hot encoded inputs in order to help build an initial model, each character w_t is mapped to $\boldsymbol{x}_t$ using the function

$$\mathbf{x}(w) = \begin{cases} o_7(1) & w = \text{" "} \text{ (space)} \\ o_7(2) & w = \text{"z"} \\ o_7(3) & w = \text{"e"} \\ o_7(4) & w = \text{"r"} \\ o_7(5) & w = \text{"o"} \\ o_7(6) & w = \text{"n"} \\ o_7(7) & \text{otherwise.} \end{cases}$$

This reduction in size allows us to derive the weights in the first layer without dealing with the large weights matrix required to process one-hot encoded inputs of size 128. The columns can then be translated into the full model.

This problem could be tackled with a finite state machine with the states

$$\textit{base},\ \textit{z},\ \textit{ze},\ \textit{zer},\ \textit{zero},\ \textit{o},\ \textit{on},\ \textit{one},\ \textit{zero\_},\ \textit{one\_}.$$

The model would then follow a set of transition rules. Starting in the *base* state, the model can move to either *z* or *o* if the next input is "z" or "o" respectively, or stay in the base state for any other input. From the *z* state, the states *ze*, *zer*, and *zero*, are reached if the following three inputs are "ero"; otherwise, the model returns to the *base* state. Similarly, from the *o* state, the states *on* and *one*, are reached if the following two inputs are "ne"; otherwise, the model returns to the *base* state. The states *zero_* and *one_* are then reached if a space is added to confirm the end of the word, and these would trigger the output to confirm the word has been detected. From these states the model can move straight into either *z* or *o*, or return to the *base* state.

As we are dealing with a sequence of inputs, the first layer $\mathcal{L}^{[1]}$ of the neural network will be a recurrent layer. In order to replicate the transition rules of the finite state machine, we need to account for the fact that the state of the recurrent layer is initialized without any input, and then each subsequent output is dependent on both an input and the previous output. We have constructed the output vector representing the states as

$$\mathcal{L}^{[1]}(\boldsymbol{x}_t) = \begin{pmatrix} \boldsymbol{p}_t \in \{0,1\}^7 \\ \boldsymbol{q}_t \in \{0,1\}^2 \\ r_t \in \{0,1\} \end{pmatrix}$$

where

- The states *z*, *ze*, *zer*, *zero*, *o*, *on*, and *one* are represented by $\boldsymbol{p}_t$, which is a one-hot encoding using o_7. If nothing is detected, it can also be $\mathbf{0}$.

- The states *zero_* and *one_* are represented by $\boldsymbol{q}_t$ which is a one-hot encoding using o_2. If neither is detected, it can also be $\mathbf{0}$.
- An additional indicator r_t is used to "block" $\boldsymbol{p}_t$ from moving into a non-zero state if w_{t-1} is not a space. This prevents the detection of words within other words, such as "one" within "gone".

The overall construction of the weights is designed to consider the possible states that can be reached from the given input and the previous state separately, and combine them through a sum to reach the next output state. The weights matrix applied to the input $\boldsymbol{x}_t$ is

$$\mathbf{W}_1 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

In $\mathbf{W}_1$, each column corresponds to a possible value in the input and each row corresponds to a possible value in the resulting output state. The entries determine adjustments to each value of the output state depending on which input is received. The first column indicates that when w_t is a space, both one-hot states of $\boldsymbol{q}_t$ are possible (rows 8 and 9), and furthermore that the blocking indicator r_t should be reduced from 1 to 0 as a new sequence may start following the space. The next five columns correspond to possible states of $\boldsymbol{p}_t$ where letter sequences have been detected, and r_t is set to 1 to prevent a new sequence starting. The final column resets $\boldsymbol{p}_t$ if w_t is any other character, and also sets r_t to 1.

The weights matrix applied to the previous state $\mathcal{L}^{[1]}(\boldsymbol{x}_{t-1})$ is

$$\mathbf{U}_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}. \tag{4.1}$$

In $\mathbf{U}_1$, the rows again correspond to possible values in the resulting output state, but the entries determine adjustments to each value of the output state

depending on which previous output state is received. The columns correspond to possible values in the previous output state.

- The first column indicates that if the previous state contains $\boldsymbol{p}_{t-1} = o_7(1)$ i.e. the *z* state, then $\boldsymbol{p}_t$ may equal $o_7(1)$ i.e. the *ze* state. The second, fifth, and sixth columns are similar.
- The third column indicates that if the previous state contains $\boldsymbol{p}_{t-1} = o_7(3)$ i.e. the *zer* state, then $\boldsymbol{p}_t$ may equal $o_7(4)$ i.e. the *zero* state, and also should not equal $o_7(5)$ i.e. the *o* state.
- The fourth column indicates that if the previous state contains $\boldsymbol{p}_{t-1} = o_7(4)$ i.e. the *zero* state, then $\boldsymbol{q}_t$ may equal $o_2(1)$ i.e. the *zero_* state. The seventh column is similar.
- The tenth column indicates that if r_{t-1} is 1, then $\boldsymbol{p}_t$ should not equal $o_7(1)$ i.e. the *z* state, or $o_7(5)$ i.e. the *o* state. This "blocks" the next state from starting the detection of a new word.

The bias

$$\boldsymbol{b}_1^T = \begin{pmatrix} 0 & -1 & -1 & -1 & 0 & -1 & -1 & -1 & -1 & 0 \end{pmatrix}$$

adjusts the entries where necessary so that, combined with a ReLU activation on the layer, the output values are either 1 or 0.

We now observe that $\boldsymbol{q}_t$ constitutes a suitable output for the network as a whole. We can either take the output from the first layer and discard the rest, or attach a dense layer $\mathcal{L}^{[2]}$ to the recurrent layer output with

$$\mathcal{L}^{[2]}(\boldsymbol{x}_t) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \mathcal{L}^{[1]}(\boldsymbol{x}_t),$$

where the bias is equal to $\mathbf{0}$ and the activation function is the linear activation function.

4.2.2 Full Model

We can expand this model to a "full" model, where each character has a value assigned using the Python `ord()` function. This translates characters into their Unicode values, which we assume are limited to ASCII values of 0 to 127.

The only change required in the model is to $\mathbf{W}_1$, which will have 128 columns. We simply copy the columns specified in the previous model to handle each character appropriately:

- The ASCII value for "Z" and "z" are 90 and 122 respectively. Therefore, column 91 and 123 of $\mathbf{W}_1$ are set to $\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}^T$.
- The ASCII value for "E" and "e" are 69 and 101 respectively. Therefore, column 70 and 102 of $\mathbf{W}_1$ are set to $\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}^T$.

- The ASCII value for "R" and "r" are 82 and 114 respectively. Therefore, column 83 and 115 of $\mathbf{W}_1$ are set to $\begin{pmatrix}0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1\end{pmatrix}^T$.
- The ASCII value for "O" and "o" are 79 and 111 respectively. Therefore, column 80 and 112 of $\mathbf{W}_1$ are set to $\begin{pmatrix}0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1\end{pmatrix}^T$.
- The ASCII value for "N" and "n" are 78 and 110 respectively. Therefore, column 79 and 111 of $\mathbf{W}_1$ are set to $\begin{pmatrix}0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1\end{pmatrix}^T$.
- The ASCII values for other alpha-numeric characters are 48 to 57 for numbers 0 to 9, 65 to 90 for uppercase letters, and 97 to 122 for lowercase letters. Therefore, columns 49 to 58, 66 to 91, and 98 to 123 (excluding those assigned above) of $\mathbf{W}_1$ are set to $\begin{pmatrix}0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1\end{pmatrix}^T$.
- The remaining ASCII values include space (ASCII value 32) and other punctuation. Therefore, the remaining columns of $\mathbf{W}_1$ are set to $\begin{pmatrix}0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1\end{pmatrix}^T$.

As we are still only detecting "zero" and "one" with this model, we do not need any more states. The other parts are therefore unchanged.

4.2.3 Input Padding

In the models above, we have not added anything to the end of the input but we can see that a word is only detected when it is followed by a space (or other punctuation). In both models, we therefore "pad" the input with a space at the end. This ensures that the strings "zero" and "one" given on their own are detected.

4.3 Results

The models described above are demonstrated in the Jupyter notebook `https://github.com/HectorLeitch/mas6041/blob/main/try_zeroone.ipynb`. The notebook is split into examples to demonstrate the code, which are described in the following paragraphs.

Example 4.1 The function `make_rnn_model()` builds the model described in section 4.2.1. The input is a text string that is then padded and converted into a sequence of one-hot encoded vectors, each of length 7.

The output is a sequence of vectors, each of length 2. A 1 in the first entry corresponds to detecting "zero" and a 1 in the second entry corresponds to detecting "one". If neither are detected, both entries will be 0.

Example 4.2 The function `make_full_rnn_model()` builds the model described in section 4.2.2. The input is a text string that is then padded and converted into a sequence of one-hot encoded vectors, each of length 128. The output is the same as the example above.

4.4 Discussion

4.4.1 Alternative Solutions

The design in this chapter was only one possible solution to the given problem. It may be the case that others exist. In particular, in the next chapter we use a technique where the state of the recurrent layer is initialized prior to any input; it was not immediately obvious how this could provide a solution in this chapter, and so more time was dedicated to the other chapters.

Other designs may have also been possible using a GRU layer, where z_t could be used to accept or reject the proposed state and govern some part of the transition rules, such as whether to allow the model to move away from the base state.

4.4.2 Possible Extensions

Given that the underlying model is a finite state machine, we could add any finite number of words to our list of words to detect. For example, if we wanted to detect the words for all digits from "zero" to "nine", we could build a network where $\boldsymbol{p}_t$ adds states for *t*, *tw*, *two*, *th*, *thr*, etc., up to *n*, *ni*, *nin*, and *nine*, and $\boldsymbol{q}_t \in \{0, 1\}^{10}$ has an output for each digit.

In the first model, the third and fifth column of $\mathbf{W}_1$, and the third column of $\mathbf{U}_1$ give the strategies required to handle letters that are shared between the words. For example, the column in $\mathbf{U}_1$ corresponding to "seven" detected should have a -1 in the row corresponding to the *n* state to prevent the next state moving to *ni*.

Chapter 5

Letter and Word Detection

5.1 Introduction

.

This chapter will introduce Convolution Neural Networks (CNNs) for processing images. We will build a simple neural network to find letters in an image and, with some additional assumptions, extend it to a neural network that can identify words.

The assumptions we will make throughout this chapter are that an input image $\mathbf{X}$ is a matrix with w columns and h rows. Each entry in $\mathbf{X}$ takes a value 0 or 1, where 0 represents a white pixel and 1 represents a black pixel. Furthermore, the input $\mathbf{X}$ will contain letters from the set $\{I, Y, J, C, O, L, H, T, U, X\}$, written in uppercase. These letters can be represented by matrices in $M_{3,3}\{0, 1\}$. Denote the matrix representing the letter $a \in \{I, Y, J, C, O, L, H, T, U, X\}$ as $A_a \in M_{3,3}\{0, 1\}$.

After building the convolution layer required to detect these letters, we will attach further layers to build models that can keep track of counts of each letter, and recognize words from a given list containing words using only the letters in $\{I, Y, J, C, O, L, H, T, U, X\}$.

Explicitly, the 3×3 matrices are as follows:

$$\mathbf{A}_I = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad \mathbf{A}_Y = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$\mathbf{A}_J = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix} \quad \mathbf{A}_C = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

$$\mathbf{A}_O = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix} \quad \mathbf{A}_L = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

$$\mathbf{A}_H = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \quad \mathbf{A}_T = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$
$$\mathbf{A}_U = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix} \quad \mathbf{A}_X = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

5.2 Methods

The methods described in this section are implemented in the file `https://github.com/HectorLeitch/mas6041/blob/main/code/letters.py`.

5.2.1 Letter Recognition

We start by building convolution kernels to recognize each letter, in a neural network with just a single convolution layer. For this step we restrict an input $\mathbf{X}$ to be 3×3, so that with a 3×3 kernel a single output value is given.

For a given $a \in \{I, Y, J, C, O, L, H, T, U, X\}$, let $S_a = \Big\{(i,j) \,|\, [\mathbf{A}_a]_{i,j} = 1\Big\}$. Then, for any input $\mathbf{X} \in M_{3,3}\{0,1\}$, we have

$$I(\mathbf{X} = \mathbf{A}_a) = \left(1 + \sum_{i,j \in S_a} [\mathbf{X}]_{i,j} - \sum_{i,j \notin S_a} [\mathbf{X}]_{i,j} - \sum_{i,j} [\mathbf{A}_a]_{i,j}\right)_+ . \tag{5.1}$$

It follows that we can create a kernel $\mathbf{W}_a$ by setting $[\mathbf{W}_a]_{i,j}$ to 1 for $(i,j) \in S_a$ and -1 for $(i,j) \notin S_a$, i.e.

$$\mathbf{W}_a = 2\mathbf{A}_a - \mathbf{1}_{3,3}.$$

We combine this with a bias term $\mathbf{b}_a = 1 - \sum_{i,j} [\mathbf{A}_a]_{i,j}$, and the RelU activation function so that the output values are only 0 or 1.

To create the first neural network, we combine the kernels $\mathbf{W}_a$ and biases $\boldsymbol{b}_a$ for each $a \in \{I, Y, J, C, O, L, H, T, U, X\}$, resulting in a layer $\mathcal{L}^{[1]}$ with 10 channels.

5.2.2 Expanding the Grid

The previous model was restricted to producing a single output in $\{0,1\}$ for each channel; the convolution kernels did not move across the spatial dimensions of the input $\mathbf{X}$ because we restricted the size of the input to match the size of the kernels. We now consider inputs $\mathbf{X} \in M_{h,w}\{0,1\}$, where $w, h > 3$.

For this model, some further assumptions will be made about $\mathbf{X}$.

- $\mathbf{X}$ may contain multiple sub-matrices $\mathbf{A}_a$, or none at all. And the sub-matrix for each letter a may appear more than once.

- The letters are all written from left to right on a single "line", although the vertical positions may be anywhere such that they fit within the height h.
- Each letter occupies the whole of its 3×3 sub-matrix, and the sub-matrices of any two letters cannot overlap.

The first layer of the model is a convolution layer with 10 channels, where the weights, biases, and activation are copied from the first model. Now the kernels move across the dimensions of the image $\mathbf{X}$, the result for each channel will be a feature map of size $(h-2) \times (w-2)$.

As we have assumed that the letters will not overlap, we can then pass the output $\mathcal{L}^{[1]}(\mathbf{X})$ to a max pooling layer. We know each letter can only be present once within each vertical $h \times 3$ slice, so we can set the height of the pool size to h and the width of the pool size to 3. We apply zero padding to $\mathcal{L}^{[1]}(\mathbf{X})$, so the output $\mathcal{L}^{[2]}(\mathbf{X})$ will be of size $1 \times \lfloor w/3 \rfloor$ in each channel. Therefore, as the pool moves across $\mathcal{L}^{[1]}(\mathbf{X})$, the corresponding output will be 1 each time a letter is detected within each corresponding channel.

This output is sufficient to show detection of each letter, but for the following models we will consider reshaping the output into a sequence of $w_g = \lfloor w/3 \rfloor$ vectors, where each vector is in $\{0,1\}^{w_g}$ and is either $\mathbf{0}$ for no letter detected, or a one-hot encoding of a detected letter. For example, the sequence

$$\begin{pmatrix}0&0&0&1&0&0&0&0&0&0\end{pmatrix}^T,$$
$$\begin{pmatrix}1&0&0&0&0&0&0&0&0&0\end{pmatrix}^T,$$
$$\begin{pmatrix}0&0&0&0&0&0&0&1&0&0\end{pmatrix}^T,$$
$$\begin{pmatrix}0&1&0&0&0&0&0&0&0&0\end{pmatrix}^T$$

represents the detection of the letters ("C", "I", "T", "Y").

5.2.3 Counting Occurrences

Suppose now that we wish to count the number of occurrences of each letter within the input grid. This can be done by attaching a dense layer $\mathcal{L}^{[3]}$ with input $\mathcal{L}^{[2]}(\mathbf{X})$ from the model above.

The input to this layer $\mathcal{L}^{[2]}(\mathbf{X})$ should be reshaped so that it has 10 columns, one for each letter, and w_g rows. It is then passed to the dense layer as a single input in $\{0,1\}^{w_g}$. We can then extract the sums of each of the letters by simply multiplying this input with the vector of ones $\mathbf{1}_{10}$, adding no bias or activation function.

5.2.4 Recognizing Words

To recognize words, we wish to take the output from $\mathcal{L}^{[2]}(\mathbf{X})$ and convert this into a unique integer sequence to represent a detected word. For example, the

detection of the word ("C", "I", "T", "Y") would be recognized from the sequence $\begin{pmatrix}4 & 1 & 8 & 2 & 0 & \dots & 0\end{pmatrix} \in \{0, \dots, 10\}^{w_g}$.

The problem to solve is that we may encounter gaps in the letters and so after converting the one-hot outputs to integers, they will not necessarily be in the left-most positions and there may be zeros between them. We therefore need a way to process the sequence to move the non-zero entries to the start whilst retaining their order.

The first layer following $\mathcal{L}^{[2]}$ will convert each one-hot encoded output into an integer $x_t \in \{0, \dots, 10\}$, where a zero output is mapped to 0. We also define

$$s_t = \sum_{u<t} I(x_u > 0),$$

so that if $x_t > 0$, it should appear in position s_t.

Noting that this is the largest hand-crafted model we will see, the steps taken following $\mathcal{L}^{[2]}$ in the first model are tabulated below.

Layer	Type	Output shape	Output at t
$\mathcal{L}^{[2]}$	max pooling	$(w_g, 10)$	$o_{10}(x_t)$
$\mathcal{L}^{[3]}$	dense	$(w_g, 2)$	$(x_t, I(x_t > 0))$
$\mathcal{L}^{[4]}$	GRU	(w_g, w_g)	s_t
$\mathcal{L}^{[5]}, \mathcal{L}^{[6]}$	dense	(w_g, w_g)	$o_{w_g}(s_t)$
$\mathcal{L}^{[7]}$	merge	$(w_g, w_g + 2)$	$\big(x_t, I(x_t > 0), o_{w_g}(s_t)\big)$
$\mathcal{L}^{[8]}$	dense	(w_g, w_g)	$I(x_t > 0) o_{w_g}(s_t)$
$\mathcal{L}^{[9]}$	merge	$(w_g, w_g + 2)$	$\big(x_t, I(x_t > 0), I(x_t > 0) o_{w_g}(s_t)\big)$
$\mathcal{L}^{[10]}$	GRU	(w_g)	$\sum_{u<t} x_u o_{w_g}(s_u)$

Following the max pooling layer $\mathcal{L}^{[2]}$, the output is converted to an integer sequence of x_t values, plus a corresponding indicator $I(x_t > 0)$, through a single dense layer. Treating $\mathcal{L}^{[2]}(\mathbf{X})$ as a $10 \times w_g$ matrix, we can set

$$\mathcal{L}^{[3]}(\mathbf{X}) = \begin{pmatrix} 1 & 2 & \dots & 10 \\ 1 & 1 & \dots & 1 \end{pmatrix} \mathcal{L}^{[2]}(\mathbf{X}).$$

The first row of the matrix creates the integers x_t and the second row creates $I(x_t > 0)$ for all $t \in \{1, \dots, w_g\}$ simultaneously. No bias is needed and the activation function is set to the linear function.

We now wish to create the sequence $\{s_t\}_{t=1}^{w_g}$. This will utilize a simple gated recurrent unit (GRU) layer, $\mathcal{L}^{[4]}$, using the sequence of values $\{I(x_t > 0)\}_{t=1}^{w_g}$ as input. Starting with $y_0 = 0$, at time t the GRU will propose a new state $\hat{y}_t = y_{t-1} + 1$ and then set $z_t = 1 - I(x_t > 0)$. Therefore,

$$\begin{aligned} x_t > 0 &\implies z_t = 0 \\ &\implies y_t = \hat{y}_t = y_{t-1} + 1, \end{aligned}$$

and conversely

$$\begin{aligned} x_t = 0 &\implies z_t = 1 \\ &\implies y_t = y_{t-1}. \end{aligned}$$

As we also have $\{x_t\}_{t=1}^{w_g}$ from the previous layer, any weights and biases corresponding to these inputs are set to 0. The remaining weights and biases corresponding to $\{I(x_t > 0)\}_{t=1}^{w_g}$ are set to

- $\mathbf{W}_r^{[4]} = \mathbf{U}_r^{[4]} = \boldsymbol{b}_r^{[4]} = 0 \implies r_t = 0$
- $\mathbf{W}_z^{[4]} = -1, \mathbf{U}_z^{[4]} = 0, \boldsymbol{b}_z^{[4]} = 1 \implies z_t = 1 - I(x_t > 0).$
- $\mathbf{W}_y^{[4]} = 0, \mathbf{U}_y^{[4]} = 1, \boldsymbol{b}_y^{[4]} = 1 \implies \hat{y}_t = y_{t-1} + 1.$

The output $\mathcal{L}^{[4]}(x_t) = s_t$ is then set to y_t.

We now create a sequence of vectors $\left\{o_{w_g}(s_t)\right\}_{t=1}^{w_g}$. Observe that for $x, n \in \mathbb{Z}$,

$$\begin{aligned} & I(x > n) = (x-n)_+ - (x-n-1)_+ \\ \implies & I(x = n) = (1+x-n)_+ - 2(x-n)_+ + (x-n-1)_+ . \end{aligned} \tag{5.2}$$

Therefore, we add dense layers $\mathcal{L}^{[5]}$ and $\mathcal{L}^{[6]}$, where

$$\mathcal{L}^{[5]}(s_t) = \left(\begin{pmatrix} 1 \\ 1 \\ 1 \\ \dots \\ 1 \\ 1 \\ 1 \end{pmatrix} s_t + \begin{pmatrix} 1 \\ 0 \\ -1 \\ \dots \\ w_g + 1 \\ w_g \\ w_g - 1 \end{pmatrix} \right)_+ ,$$

and

$$\mathcal{L}^{[6]}(s_t) = \begin{pmatrix} 1 & -2 & 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 1 & -2 & 1 \end{pmatrix} \mathcal{L}^{[5]}(s_t) .$$

As this output drops the original sequence $\{x_t\}_{t=1}^{w_g}$, we add a *merge* layer $\mathcal{L}^{[7]}$ which simply concatenates the output of $\mathcal{L}^{[3]}$ and $\mathcal{L}^{[6]}$

The last part before the GRU layer is combining the indicators we have produced so far into a sequence of vectors $\{\boldsymbol{q}_t\}_{t=1}^{w_g}$, where each $\boldsymbol{q}_t$ is $I(x_t > 0) o_{w_g}(s_t)$. This is the overall indicator that the non-zero output x_t should go into position s_t. This can be done in a single dense layer $\mathcal{L}^{[8]}$ where the input is the concatenation $\mathcal{L}^{[7]}(x_t)$, which is a vector of length $w_g + 2$. The calculation for $\mathcal{L}^{[8]}$ is

$$\mathcal{L}^{[8]}(\mathbf{X}) = \left(\begin{pmatrix} \mathbf{0}_{w_g} & \mathbf{1}_{w_g} & \mathbf{I}_{w_g} \end{pmatrix} \mathcal{L}^{[7]}(\mathbf{X}) - \mathbf{1}_{w_g} \right)_+ ,$$

where the weights matrix is of size $w_g \times (w_g + 2)$, consisting of two column vectors followed by the identity matrix of size w_g. This again yields a sequence where the original inputs have been dropped so we add another merge layer $\mathcal{L}^{[9]}$ to concatenate the outputs of $\mathcal{L}^{[3]}$ and $\mathcal{L}^{[8]}$.

The final layer yielding the desired sequence is a GRU layer $\mathcal{L}^{[10]}$, which takes $(x_t, I(x_t > 0), \boldsymbol{q}_t)$ as input. The components are set as follows:

$$\begin{aligned} \boldsymbol{r}_t &= \mathbf{1}_{w_g}, \\ \boldsymbol{z}_t &= \left(\mathbf{1}_{w_g} - \boldsymbol{q}_t\right)_+, \\ \hat{\boldsymbol{y}}_t &= \mathbf{1}_{w_g} x_t. \end{aligned}$$

which implies

$$\left[\boldsymbol{y}_t\right]_i = \begin{cases} \left[\boldsymbol{y}_{t-1}\right]_i & x_t = 0 \text{ or } s_t \neq i \\ x_t & x_t > 0 \text{ and } s_t = i. \end{cases}$$

So $\boldsymbol{y}_0$ is initialized as $\mathbf{0}$, and the i-th entry is updated with the i-th non-zero value in $\{x_t\}_{t=1}^{w_g}$. We then take only the final state $\boldsymbol{y}_{w_g}$ as the output of this layer, which yields the sequence with all non-zeros shifted to the start.

Explicitly, for the GRU layer $\mathcal{L}^{[10]}$, we have non-zero weights and biases

- $\boldsymbol{b}_r^{[10]} = \mathbf{1}_{w_g} \implies \boldsymbol{r}_t = 0$,
- $\mathbf{W}_z^{[10]} = \begin{pmatrix} \mathbf{0}_{w_g,2} & -\mathbf{I}_{w_g} \end{pmatrix}, \boldsymbol{b}_z = \mathbf{1}_{w_g} \implies \boldsymbol{z}_t = \left(\mathbf{1}_{w_g} - \boldsymbol{q}_t\right)_+$,
- $\mathbf{W}_y^{[10]} = \begin{pmatrix} \mathbf{1}_{w_g} & \mathbf{0}_{w_g,w_g+1} \end{pmatrix} \implies \hat{\boldsymbol{y}}_t = \mathbf{1}_{w_g} x_t$.

5.2.5 Alternative Shift Method

The model in the previous section contains many layers, spread over multiple stages. We now present an alternative architecture which slightly reduces the number of layers.

The aim is to reduce the number of layers required to take the indicator $I(x_t > 0)$ for $t = 1, \ldots, w_g$, and produce the sequence $\{\boldsymbol{q}_t\}_{t=1}^{w_g}$ directly in one layer. This eliminates the need to build the running counter s_t and then feed that into another two layers to produce indicators.

The modification replaces the layers $\mathcal{L}^{[4]}, \mathcal{L}^{[5]}$, and $\mathcal{L}^{[6]}$ with a single GRU layer $\mathcal{L}^{[4*]}$. We create a hidden state $\boldsymbol{y}_t \in \{0, 1\}^{w_g+1}$, which we initialize as

$$\boldsymbol{y}_0 = \begin{pmatrix} 1 & 0 & \ldots & 0 \end{pmatrix}^T.$$

The idea is to propose $\hat{\boldsymbol{y}}_t$ to be the right-shifted $\boldsymbol{y}_{t-1}$, and use the input $I(x_t > 0)$ to determine whether or not to update $\boldsymbol{y}_t$. We can then drop the first element of $\boldsymbol{y}_t$ to give us $\boldsymbol{p}_t$, by adjusting the weights in the following layer to ignore the first entry of $\boldsymbol{y}_t$.

Explicitly, we have:

$$\left.\begin{aligned}\mathbf{W}_r^{[4*]} &= \mathbf{0}_{w_g+1} \\ \mathbf{U}_r^{[4*]} &= \mathbf{0}_{w_g+1\times w_g+1} \\ \boldsymbol{b}_r^{[4*]} &= \mathbf{1}_{w_g+1}\end{aligned}\right\} \implies \boldsymbol{r}_t = \mathbf{1}_{w_g+1}\forall t$$

$$\left.\begin{aligned}\mathbf{W}_z^{[4*]} &= -\mathbf{1}_{w_g+1} \\ \mathbf{U}_z^{[4*]} &= \mathbf{0}_{w_g+1\times w_g+1} \\ \boldsymbol{b}_z^{[4*]} &= \mathbf{1}_{w_g+1}\end{aligned}\right\} \boldsymbol{z}_t = \mathbf{1}_{w_g+1} - I(x_t > 0) \text{ (element-wise)}$$

$$\left.\begin{aligned}\mathbf{W}_y^{[4*]} &= \mathbf{0}_{w_g+1} \\ \mathbf{U}_y^{[4*]} &= \mathbf{R}_{w_g+1} \\ \boldsymbol{b}_y^{[4*]} &= \mathbf{0}_{w_g+1}\end{aligned}\right\} \implies \hat{\boldsymbol{y}}_t = \mathbf{R}_{w_g+1}\boldsymbol{y}_{t-1}$$

where $\mathbf{R}_{w_g+1}$ is the "right-shift" matrix (although $\boldsymbol{y}_t$ is a column vector in our notation)

$$\mathbf{R}_{w_g+1} = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & 0 & 1 \end{pmatrix}.$$

We now replace $\mathcal{L}^{[8]}$ with a slightly modified version $\mathcal{L}^{[8*]}$, to account for the extra input which we do not wish to use. For this layer we have

$$\mathcal{L}^{[8*]}(\mathbf{X}) = \left(\begin{pmatrix}\mathbf{0}_{w_g} & \mathbf{1}_{w_g} & \mathbf{0}_{w_g} & \mathbf{I}_{w_g}\end{pmatrix}\mathcal{L}^{[7]}(\mathbf{X}) - \mathbf{1}_{w_g}\right)_+.$$

The output of $\mathcal{L}^{[8*]}$ and $\mathcal{L}^{[8]}$ will be identical so the final layers are unchanged.

5.2.6 Full Model

The full model takes the integer sequence with non-zero entries shifted to the end, and creates an output string by first adding a dense layer $\mathcal{L}^{[11]}$ where

$$\mathcal{L}^{[11]}(\mathbf{X}) = \begin{pmatrix}11^0 & 11^1 & \dots & 11^{w_g-1}\end{pmatrix}\mathcal{L}^{[10]}(\mathbf{X}),$$

so that each possible output sequence is converted into a unique integer. We then create two dictionaries from a file `words_3x3.txt`, available in the Github repository, which contains a list of words we can make using the letters we have available. This allows us to convert the unique integer into a string containing the detected word, or a special "UNK" token if the word is not in the file.

5.3 Results

The models described above are demonstrated in the Jupyter notebook `https://github.com/HectorLeitch/mas6041/blob/main/try_letters.ipynb`. The notebook is split into examples to demonstrate the code, which are described in the following paragraphs.

Example 5.1 The function `make_explicit_model()` builds a model to recognize letters given as a 3×3 image, as described in section 5.2.1.

We verify that it correctly identifies a single letter input and, furthermore, rejects an input (i.e. does not claim to detect any letter) given a randomly generated input that does not happen to be a letter.

The function `test_explicit_model()` verifies the model is working correctly on all possible 3×3 images.

Example 5.2 The function `make_conv_model(h=,w=)` builds a model to recognize letters within a $h \times w$ grid, as described in section 5.2.2.

We supply an input grid using the function `make_ragged_grid(word=)`, which generates a grid based on an input word, spacing the letters randomly but according to the assumptions specified in section 5.1.

Example 5.3 The function `make_sum_model(h=,w=)` builds a model to recognize letters within a $h \times w$ grid, and count the total number of occurrences of each letter, as described in section 5.2.3. We supply an input grid using the function `make_ragged_grid(word=)`. One can test this example with arbitrary strings with lots of repeating letters for more interesting output results.

Example 5.4 The function `make_gru_model(h=,w=)` builds a model to recognize words within a $h \times w$ grid, and count as described in section 5.2.4. We supply an input grid using the function `make_ragged_grid(word=)`. We also verify that particular elements of the network are producing the correct output.

Example 5.5 The function `make_alt_gru_model(h=,w=)` builds a model to recognize letters within a $h \times w$ grid, with the modification to the architecture described in section 5.2.5. We supply an input grid using the function `make_ragged_grid(word=)`. The modifications to the previous model are highlighted through showing the outputs of the new layers.

Example 5.6 The function `make_full_model(h=,w=,file=)` builds a model to recognize letters within a $h \times w$ grid, with the modification to the architecture described in section 5.2.5 and the additional layers described in section 5.2.6.

We supply an input grid using the function `make_ragged_grid(word=)`. An example grid with the input and output words is shown in figure 5.1.

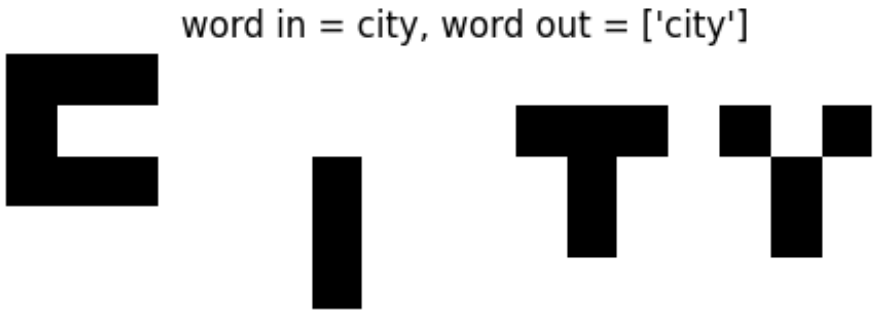

Figure 5.1: Example of the full model recognizing the word "city". The dictionary layer provides the recognized word in a tensor object which is printed with the square brackets containing the string inside.

5.4 Discussion

5.4.1 Model Architecture

The models in sections 5.2.1, 5.2.2, and 5.2.3 followed reasonably intuitive architecture, where the weights for the convolution kernels ended up looking like the letters they were trying to detect. This meant that the output node for each letter was activated when the kernels lined up with the corresponding letter in the grid.

The models in sections 5.2.4 and 5.2.5 were far more challenging to develop. Difficulties with the Tensorflow shapes meant that initial working models contained further unnecessary layers, which were ultimately replaced by a single dense layer to yield both x_t and $I(x_t > 0)$.

The GRU layers were particularly difficult to work with when deriving and assigning weights. The original paper published by Cho et al. matches the Tensorflow implementation, where

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \hat{h}_t.$$

However, variations such as the minimal gated unit, proposed by Heck and Salem (2017), set

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t.$$

The latter is slightly more intuitive when working with $z_t \in \{0, 1\}$, because z_t represents the condition for which we want to update the hidden state h_t, which would correspond to the outputs $x_t > 0$ which we are interested in. Overall, the network may have been slightly more efficient if we had replaced $I(x_t > 0)$ with $I(x_t = 0)$. Similarly, when implementing equation 2.3 to create values to be used for z_t, it would have been possible to do this within the GRU layer

using ReLU as the recurrent activation. This would slightly reduce the number of layers and overall weights in the model, although the GRU layers themselves would have been larger due to the increased number of inputs, therefore making this a difficult efficiency to implement.

That being said, using the GRU layer in section 5.2.5 provided a much more intuitive way to produce the sequence $\left\{o_{w_g}(s_t)\right\}_{t=1}^{w_g}$ and was a worthwhile improvement over the first model.

5.4.2 Possible Improvements

It is possible to further reduce the architecture of the model significantly by using a larger GRU layer taking pairs $(x_t, I(x_t > 0))$ as inputs. This solution would involve creating a hidden state of size $3w_g$, containing the output of length w_g plus two copies of a similar vector to $o_{w_g}(s_t)$ to keep track of the position. The position would be incremented using $I(x_t > 0)$ and the right-shift matrix as we have done in section 5.2.5. Two copies are stored in the hidden vector so that one of them can be multiplied by x_t in the term $(\boldsymbol{r_t} \odot \boldsymbol{y}_{t-1})$, allowing us to then propose $\hat{\boldsymbol{y}}_t$ which contains x_t right-shifted into the next available non-zero space in the output part of the hidden state, as well as the next position $o_{w_g}(s_t)$. Overall, this would eliminate the need to create a "branch" in network architecture of the model to handle the positioning, as it is done within the GRU layer structure.

The functionality of these models could be extended by allowing the images to contain multiple words, with reasonable assumptions to govern how they are split up. For example, defining a line height where each line in the image contains one word and allowing the convolution kernels to produce a feature map with more than one row would allow the model to produce outputs with multiple detected words. This would require careful reshaping of the output at each stage, but little overall modification to any of the weights or biases.

Chapter 6

Noughts and Crosses

6.1 Introduction

This chapter will explore models that are designed to play noughts and crosses (or tic-tac-toe). We will represent the board using a 3×3 matrix for each of the two players. For each player, an entry of 0 represents a space where the player has not played a move, and an entry of 1 represents a space where the player has played a move. Denote a board B as $B = \{\mathbf{X}_o, \mathbf{X}_x\}$, where $\mathbf{X}_o, \mathbf{X}_x \in M_{3,3}\{0,1\}$ represent the positions currently occupied by player o and player x respectively. Denote the starting board where all entries are 0 as B_0. Furthermore, for a player $p \in \{o, x\}$, let $\boldsymbol{x}_p \in \{0,1\}^9$ denote the vector with the same entries as $\mathbf{X}_p$, in the following order:

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}.$$

A game is played as follows. The board starts as $B = B_0$ and the value p, denoting the current player, is set to o to begin the game. Player p selects an integer $i \in \{0, \ldots, 8\}$ and updates the board B by updating the i-th entry of $\boldsymbol{x}_p$ from 0 to 1. The player p then changes from o to x or vice versa.

A player p wins, if after the board B is updated on their turn, they have a column, row, or diagonal consisting of three 1s in $\mathbf{X}_p$. In addition, if player p selects i where the i-th entry in $\boldsymbol{x}_o$ or $\boldsymbol{x}_x$ is 1 prior to updating $\boldsymbol{x}_p$, then player p loses due to making an *illegal move*. After 9 turns, if neither player has won or lost, the game is a draw.

This section will start with an experiment to see how effectively an ANN with dense layers can be trained to play noughts and crosses, and what seem to be the minimum requirements for a network to replicate perfect play. We then repeat the experiment with an intuitive modification to the network architecture to help the network quickly reduce its rate of suggesting illegal moves.

6.2 Methods

The methods described in this section are implemented in the file `https://github.com/HectorLeitch/mas6041/blob/main/code/ox.py`.

6.2.1 Creating a Perfect Player

In noughts and crosses it is possible (for both players) to play a perfect game. The player who goes first playing perfectly is guaranteed not to lose. If the player going second also plays perfectly, the game will result in a draw; otherwise, the player going first may have the opportunity to win.

We can create a perfect player, against which we can test the neural network players we create. The perfect player follows the algorithm proposed by Crowley and Siegler (1993) at each turn:

1. Win if possible: If a player has two in a row and the third position is available, play this position to win the game.
2. Block opponent: If the opponent has two in a row and the third position is available, play this position to prevent the opponent winning the game.
3. Fork: If there is a position available that creates two possibilities to play a win on the next turn, play this position.
4. Block opponent's fork: If there is an available position such that the opponent could create a fork, play this position. If there are multiple positions, play a move that creates a possibility to play a win on the next turn, that is not blocked by one of the available forks to the opponent.
5. Corner start: If it is the first move of the game, play in any corner position. This start can be ignored against a perfect player, but against an imperfect player it gives more chances for the opponent to make a mistake.
6. Center: If the center position is available, play this position.
7. Opposite corner: If the opponent has played in any corner and the opposite corner is available, play this position.
8. Empty corner: If an empty corner position is available, play this position.
9. Empty side: If an empty side position is available, play this position.

Furthermore, with procedural Python code representing this perfect player, we can generate training data by supplying it with board states and recording the suggested move as the response. We build a data set using the following assumptions:

- The board state is one where it is the turn of the player who went first. As such we are teaching our networks exclusively to play first and draw or win only; they do not need to worry about reading the board and figuring out which player's turn it is.

- The board state is one which can only be reached through legal moves.
- The board state is not a finished game so there is at least one move that can be suggested.

This data set consists of 4519 separate board states. Denote this set of board states as $\mathbf{B}$. We can think of our perfect player as a function $f_{\text{perfect}} : \mathbf{B} \to \{0, \ldots, 8\}$, where $f_{perfect}(B)$ is the position from 0 to 8 suggested by the perfect player for any $B \in \mathbf{B}$. A training dataset is constructed as $\{B_i, \boldsymbol{y}_i\}_{i=1}^{4519}$, where $\boldsymbol{y}_i = f_{perfect}(B_i)$.

Note that we have created the perfect player function to randomly select one of the four corners to start from so that more board states arise when testing the approximate models against the perfect player. This means that there are four possible entries for $f_{\text{perfect}}(B_0)$ in the training data which will depend on the random seed when the code is run, but the choice should not affect how the models train and ideally they should be able to handle board states that arise from any possible start.

6.2.2 Approximate Artificial Neural Networks

Our first goal is to build a model to approximate f_{perfect} using two hidden dense layers, and an output layer (which is also a dense layer) with 9 outputs representing the choices from 0 to 8. Later on we will experiment with the number of nodes in each layer, so let the function representing the network approximating f_{perfect} with p and q nodes in the first and second layers respectively be denoted as $f_{[p,q]}$.

We start by training a model with 20 nodes in each dense layer in order to establish suitable values for the training parameters: learning rate, epochs, and batch size. These values are then reused while training models with varying numbers of nodes in each layer to establish roughly how well the model is able to perform with p and q nodes in the first and second layers respectively, and how small p and q can be reduced before the model starts to perform significantly worse than the perfect model.

The activation functions for all the approximate models are the ReLU function on the two dense layers, and the softmax function on the output layer. As such, $f_{[20,20]}$ will actually be a function from $\mathbf{B}$ to $(0, 1)^9$, where each output can be thought of as a probability or confidence in the corresponding position. We create a player by adding a classifier to the network that selects a position i from $\{0, \ldots, 8\}$ with probability $f_{[20,20]}(B)_i$, but this does not form part of the training.

In order to train the network weights we need a suitable loss function. The positions in the training data are one-hot encoded so each $\boldsymbol{y}_i$ is a vector of length 9. The loss function is called categorical cross-entropy, and the total loss

is calculated as

$$L_{\text{cce}}\left(\boldsymbol{y}, \hat{\boldsymbol{y}}\right) = -\sum_{i=1}^{n}\sum_{j=0}^{9} y_{i,j}\log\left(\hat{y}_{i,j}\right).$$

Within each $\boldsymbol{y}_i$, only one entry will be 1 and the rest will be 0, so this loss can be thought of as trying to maximize the corresponding probability estimate in $\hat{\boldsymbol{y}}_i$.

6.2.3 Training Procedure

We will use Adam as the training optimizer, with a fixed learning rate that was manually adjusted following training runs. The number of epochs and the batch size was also manually adjusted. An early stopping criterion monitoring the loss would stop training if no improvement was observed for 20 epochs.

As the goal was to find suitable values and not necessarily optimized values, a plot of the loss over time was created after each training run and the parameters were adjusted accordingly. This included whether or not the early stopping criterion had been met. Once it appeared that the values were suitable, the model was tested against a perfect player and a random player to verify that the approximation $f_{[20,20]}$ was close to f_{perfect}. A model was tested by tabulating the results from 100 games, also recording whether any losses were due to illegal moves being suggested to understand whether this was a particular weakness.

For a given set of training values, the shape of the plot provided guidance on how to adjust the parameters. Where the loss did not consistently decrease over time, showing a jagged shape on the graph with lots of increase and decreases, this indicated that the training process was struggling to find steps that resulted in decreasing the loss. This could suggest that the learning rate was too large, resulting in "over-shooting" when adjusting the weights and biases. Alternatively, this could also suggest that the batch size was not large enough, resulting in inaccurate estimates of the gradient descent direction. Conversely, where the shape of the graph was an extremely smooth decrease that continued all the way to the maximum number of epochs without triggering the stopping criteria, this meant that the model still required more epochs to finish training. This could suggest that the learning rate was too small, and a more appropriate learning rate would reduce the loss more efficiently.

In addition to choosing a move from the values with probability proportional to the softmax outputs, we also can also test the models by changing the activation function on the final layer to the argmax function after training has been completed, so that the network chooses the position with the greatest output every time. This eliminates a lot of randomness in how the models actually play and tends to yield much better approximations of f_{perfect}.

6.2.4 Network Size Experiment

Following the determination of the performance reached by the model with 20 nodes in each hidden layer, an experiment was run to reduce the size of the network and measure the impact to performance.

Models containing two hidden dense layers of size $p, q \in \{10, 11, \ldots, 20\}$, where $p = q$, were trained using the parameters determined in the first part of the experiment. The models were tested by playing matches consisting of 100 games against the perfect player and the random player. Against the perfect player, the models were tested using the softmax and argmax position selectors. The intention behind this is to monitor the difference in softmax outputs if there are cases where the performance in the argmax models suddenly shifts after increasing or decreasing the network size by just one node per layer, to verify that there is a good reason behind a sudden shift.

6.2.5 Residual Connections

We now introduce models with a new idea: *residual connections*. A residual connection adds weights and biases that connect layers that are not adjacent to each other. The intuition behind adding these connections in this specific example is that it allows us to connect input positions directly to output positions, adding the option to include a penalty weight from positions that are occupied directly to the corresponding outputs. This should allow for models that are much better at avoiding illegal moves.

To show explicitly how this affects how the models are evaluated, note that the previous models would be evaluated as

$$f_{[p,q]}(B) = \mathcal{L}^{[3]}\left(\mathcal{L}^{[2]}\left(\mathcal{L}^{[1]}(B)\right)\right).$$

The model with residual connections is now evaluated as

$$g_{[p,q]}(B) = \mathcal{L}^{[3]}\left(\mathcal{L}^{[2]}\left(\mathcal{L}^{[1]}(B)\right) + \mathbf{W}B + \boldsymbol{b}\right)$$

where $\mathbf{W}$ and $\boldsymbol{b}$ are an additional weights matrix and bias vector respectively.

As we have an idea of how the model with residual connections might use them, we will start by testing models with added penalty weights which the model is not allowed to train, and see if this helps the model learn values for the weights in the dense layers. If we consider the board B to be $(\boldsymbol{x}_o, \boldsymbol{x}_x)^T$ then the weights matrix applying these penalties is

$$\mathbf{W} = -20\begin{pmatrix}\mathbf{I}_9 & \mathbf{I}_9\end{pmatrix},$$

where $\mathbf{I}_9$ is the 9×9 identity matrix. We leave the bias as $\boldsymbol{b} = \mathbf{0}$. With this addition, we repeat the experiment in section 6.2.4 starting by finding suitable training parameters, and training and testing models over varying sizes of dense layers. The range of layer sizes was kept the same for this experiment.

The final experiment in this chapter then repeats the process but allowing the model to also train the weights and biases of the residual connections. Based on the previous results we adjusted the range of sizes for this experiment, so that $p, q \in \{5, 6, \ldots, 15\}$.

6.3 Results

The models described above are demonstrated in the Jupyter notebook `https://github.com/HectorLeitch/mas6041/blob/main/try_ox.ipynb`. The notebook is split into examples to demonstrate the code, which are described in the following paragraphs.

Example 6.1 The function `suggest_move(pq=, player=)` is the perfect player. This example simply shows the perfect player against itself, and against a random player. The function `play_match(player_o=, player_x=, bestof=)` plays a match of `bestof` games and records the results. This will be the basis of evaluating model performance in the following examples.

The code verifies that two perfect players draw against each other every time, and that the perfect player beats the random player overwhelmingly often - around 98% of the time. These facts give us a quick way to evaluate how well an approximate model plays.

Example 6.2 The function `make_simple_model(p=,q=,r=)` builds a model with `p` and `q` nodes in the first and second dense layers respectively, and sets the learning rate `r` in the optimizer.

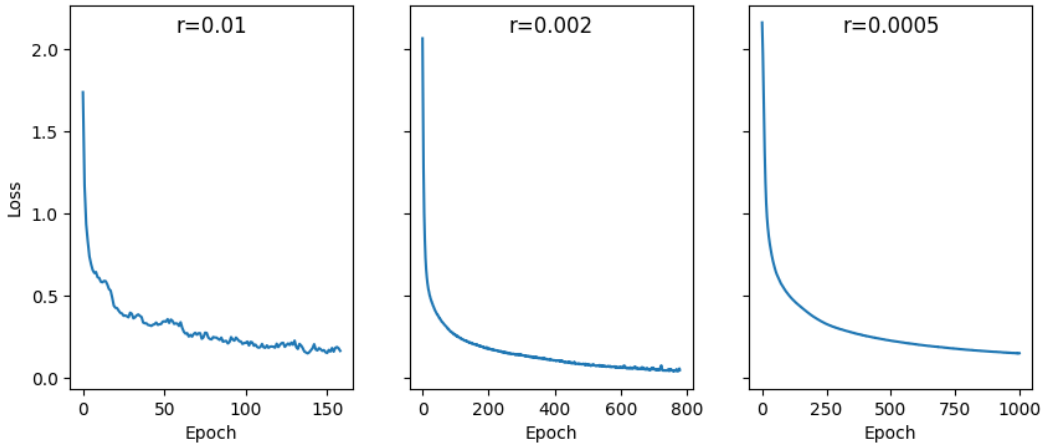

Figure 6.1: Loss over time for learning rates of 0.01, 0.002, and 0.0005, for models where $p = q = 20$. At 0.01, the steps struggle to consistently reduce the loss and training stops too early. At 0.0005, the loss is still decreasing by 1000 epochs. A learning rate of 0.02 is the "Goldilocks" value, reaching the stopping criteria at the lowest loss of the three models.

The model is trained with the function `train_model_suggestions(M=,epochs=,batch_size=)`, which trains the model with the given parameters and returns the history of the loss over time, which can be plotted. Figure 6.1 shows a collection of plots where models with 20 nodes in each hidden layer were trained for up to 1000 epochs with a batch size of 64.

The training parameters were adjusted as described in section 6.2.3 and a learning rate of 0.02 was chosen with a batch size of 64 and maximum number of epochs of 1000 to be the values for which to carry forward to training further models.

Example 6.3 Table 6.1 shows the total number of draws, wins, and losses for each size of model in the three matches specified in section 6.2.4, and the number of losses due to illegal moves. In some cases where the model using the softmax activation function on the output layer performs quite well, using the argmax activation function on the output layer instead leads to significantly better approximations of f_{perfect} to the extent that they are able to draw every time against the perfect player.

vs. Player: Activation:	Perfect player Softmax D/W/L/I	vs. Perfect player Argmax D/W/L/I	vs. Random player Argmax D/W/L/I
10	37/0/63/43	100/0/0/0	30/54/16/8
11	38/0/62/43	0/0/100/100	11/73/16/9
12	54/0/46/26	0/0/100/0	9/80/11/8
13	70/0/30/9	100/0/0/0	23/77/0/0
14	36/0/64/63	0/0/100/100	16/84/0/0
15	43/0/57/49	0/0/100/100	6/93/1/1
16	88/0/12/10	100/0/0/0	30/66/4/0
17	36/0/64/6	0/0/100/0	22/70/8/4
18	69/0/31/27	100/0/0/0	7/93/0/0
19	70/0/30/30	100/0/0/0	18/82/0/0
20	96/0/4/4	100/0/0/0	1/99/0/0
D = Draws, W = Wins, L = Losses, I = Losses due to illegal moves			

Table 6.1: Full results table for models with two dense layers, where the "vs. Player" row indicates the opponent that the model was tested against. The most difficult to explain behaviour is the losses using the argmax activation function, which are either all due to illegal moves or never due to illegal moves.

The number of draws against the perfect player and wins against the random player is shown in figure 6.2. Against both the perfect player and the random player, it appears that the larger models may be performing slightly better.

Figure 6.3 shows the number of losses due to illegal moves in each match. It appears that in most cases, a loss is due to an illegal move. In the argmax

models, it is not clear how some models lose all 100 games without any illegal moves.

The code in the notebook is able to repeat this experiment and the following two experiments, with the user able to adjust the range of layer sizes tested.

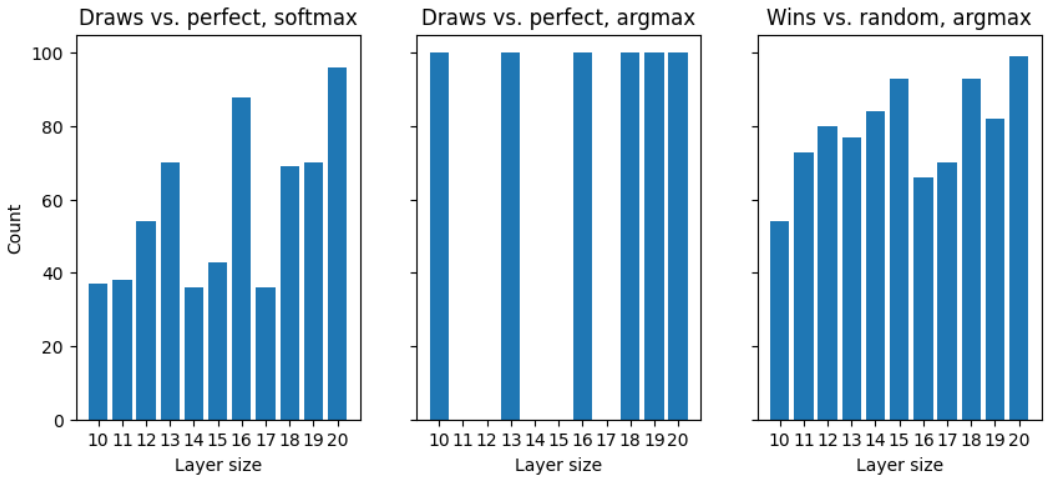

Figure 6.2: Larger models in general seem to be performing slightly better. There appear to be some potential outliers in the softmax models at sizes 13, 16, and 20.

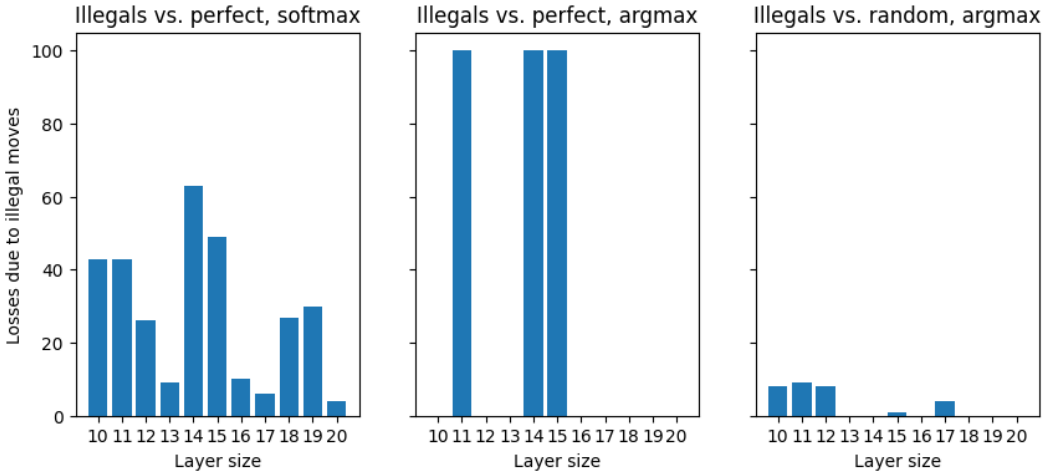

Figure 6.3: In most cases, models are losing due to illegal moves. This seems to be the default behaviour where the loss function does not treat an illegal move differently to a legal move that is not correct.

Example 6.4 Table 6.2 shows the total number of draws, wins, and losses for each size of model using residual connections with the fixed weights, and the number of losses due to illegal moves. In comparison to table 6.1, we can see that the performance is much less affected by the size and in general the

models perform very well. In particular, the performance of the models using the softmax activation function on the output layer is greatly increased with the weakest model still drawing 79% of the time against the perfect player. This translates into all of the argmax models able to draw every time vs. the perfect player over the 100 games per match.

vs. Player: Activation:	Perfect player Softmax D/W/L/I	vs. Perfect player Argmax D/W/L/I	vs. Random player Argmax D/W/L/I
10	96/0/4/3	100/0/0/0	20/80/0/0
11	82/0/18/18	100/0/0/0	16/84/0/0
12	91/0/9/9	100/0/0/0	6/94/0/0
13	84/0/16/16	100/0/0/0	10/80/0/0
14	85/0/15/15	100/0/0/0	5/95/0/0
15	79/0/21/17	100/0/0/0	13/87/0/0
16	100/0/0/0	100/0/0/0	12/88/0/0
17	80/0/20/20	100/0/0/0	13/87/0/0
18	97/0/3/3	100/0/0/0	18/82/0/0
19	94/0/6/6	100/0/0/0	21/79/0/0
20	90/0/10/15	100/0/0/0	9/91/0/0
D = Draws, W = Wins, L = Losses, I = Losses due to illegal moves			

Table 6.2: Full results table for models with two dense layers and untrained residual connections. Here we see good performances from all the models and no sign that any of the models were too small. This informed the decision to alter the ranges for the final experiment in this chapter.

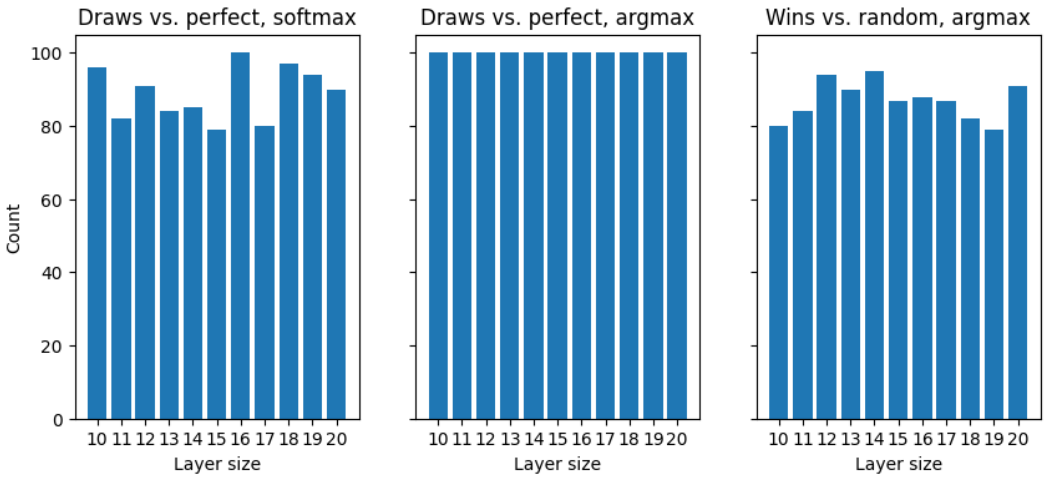

Figure 6.4: The difference in the overall level of performance from the models using residual connections is especially evident in this figure when compared to figure 6.2.

The number of draws against the perfect player and wins against the random

player is shown in figure 6.4, where we can see the significant improvement in the models using the softmax activation function on the output layer compared to the models with only dense layers. From the graph we can also see that the performance appears to be more consistent across the range of layer sizes.

The code in the notebook is able to repeat this experiment, with the user able to adjust the size of the penalty applied in the residual weights. If the results of the next example suggested it was necessary, this could have been re-adjusted from -20.

Example 6.5 Table 6.3 shows the total number of draws, wins, and losses for each size of model using residual connections with the trainable weights, and the number of losses due to illegal moves. These models performed particularly well, and we adjusted the range of sizes to see if even smaller dense layers would affect the model performance. The results show that even with the model using just 5 nodes in each hidden layer and the softmax activation function on the output layer, the approximation to f_{perfect} was very good. In particular, almost all cases of losing due to an illegal move have been eliminated.

vs. Player: Activation:	Perfect player Softmax D/W/L/I	vs. Perfect player Argmax D/W/L/I	vs. Random player Argmax D/W/L/I
5	76/0/24/0	100/0/0/0	16/84/0/0
6	78/0/22/0	100/0/0/0	15/84/1/0
7	91/0/9/0	100/0/0/0	18/82/0/0
8	99/0/1/0	100/0/0/0	25/73/2/0
9	98/0/2/0	100/0/0/0	21/77/2/0
10	87/0/13/4	100/0/0/0	15/82/3/0
11	98/0/2/1	100/0/0/0	22/77/1/0
12	95/0/5/0	100/0/0/0	11/88/1/0
13	95/0/5/2	100/0/0/0	11/88/1/0
14	93/0/7/6	100/0/0/0	16/84/0/0
15	96/0/4/1	100/0/0/0	18/76/6/4
D = Draws, W = Wins, L = Losses, I = Losses due to illegal moves			

Table 6.3: Full results table for models with two dense layers and trained residual connections. Overall the models perform extremely well, although they tended to draw slightly more often against the random player than the models with untrained residual connections.

The number of draws against the perfect player and wins against the random player is shown in figure 6.5. The notable features are that the models using the softmax activation function are performing even better than those shown in figure 6.4, but the models using the argmax activation function are not winning as often against the random player.

The code also shows that for the smallest of these models, the weights matrix did indeed learn to apply strong negative weights as our intuition suggested;

these weights also confirm that the fixed penalty of -20 was a suitable value to use for the previous experiment. While most of the other weights were close to 0, many were of a reasonably large magnitude.

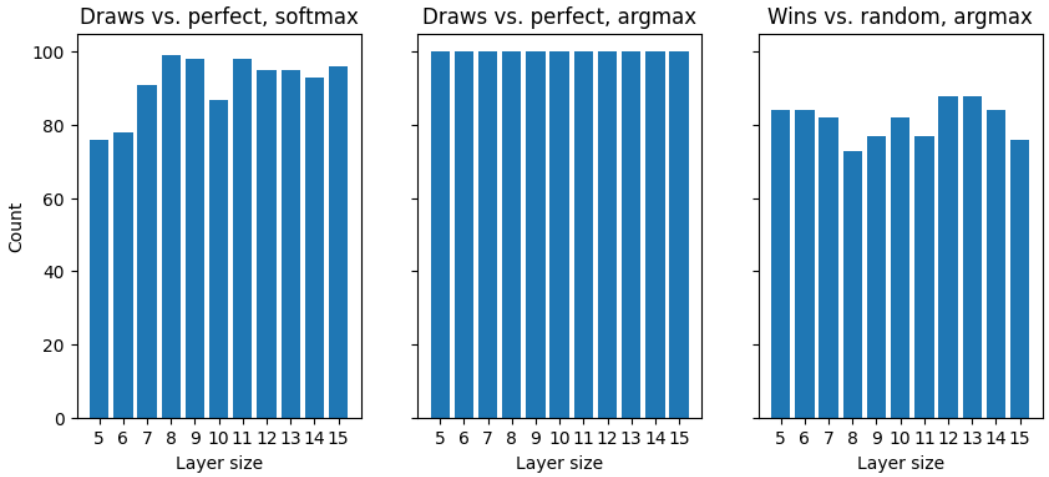

Figure 6.5: The stronger performance of the models using the softmax activation but weaker performance of the models using the argmax activation function suggests these models may have some particular bias towards moves that lead to draws.

6.4 Discussion

6.4.1 Interpretation of Results

The final part of the experiment where the models were allowed to train the weights and biases of the residual connections lead to models that performed very well despite a dramatic reduction in the number of available parameters compared to the models with no residual connections.

The model with $p = q = 20$ and no residual connections had a total of $(18 + 1) * 20 + (20 + 1) * 20 + (20 + 1) * 9 = 989$ parameters, whereas as the model with $p = q = 5$ and trainable residual connections had a total of $(18 + 1) * 5 + (5 + 1) * 5 + (18 + 1) * 9 + (5 + 1) * 9 = 350$ parameters.

It is worth noting that as we only trained one model at each size per variation, the numerical results may be quite unreliable. Particular models may have performed exceptionally well or poorly due to the initialization of the weights in the network. With more time, we could have trained multiple models at each size and measured their average performance. However, the magnitude of the difference supports the qualitative assessment that the residual connections lead to a great increase in performance and the ability to reduce the overall size of the network significantly. This is a satisfactory result given the aim of the experiment.

6.4.2 Training Limitations

This chapter consisted of experiments which involved training a collection of models. This was performed on a Windows PC using only the CPU, and was therefore many times slower than it would have been with a dedicated GPU to accelerate the training. This motivated the decision to restrict the experiments to networks with two dense layers of equal size, as relaxing this constraint would have resulted in too many models to train in a reasonable time frame. Accelerated training could have also allowed us to obtain more reliable data for the performance of models at each size, giving us a chance to compare the performances quantitatively.

6.4.3 Further Variations

With more computing resources and time, it would have been interesting to see whether creating an imbalance in the layer sizes could improve the performance. Where the models were allowed to train the weights and biases of the residual connections, the performance appeared to be mainly driven by these weights and so it would have been interesting to see if dropping one of the hidden layers entirely would have affected the performance at all, and if near-perfect performance could still be reached, whether that would have happened in a significantly shorter time.

6.4.4 Reinforcement Learning

Reinforcement learning is a different approach to training models. Broadly speaking, this involves performing the task (i.e. playing a game of noughts and crosses) with an untrained model, and scoring each performance using a *reward* or *value* function. The model parameters are then adjusted and the process it iterated, with the goal being to maximize the reward. This allows us to train models without training data prepared, monitor how the model is learning to play the game as it trains, and adjust the reward to encourage certain behaviour such as prioritizing legal moves first. This was deemed out of the scope of this project as reinforcement learning can be difficult to implement effectively and the training process tends to require a lot of computational resources.

Chapter 7

Digit Recognition

7.1 Introduction

The task of recognizing hand written digits using neural networks, or other machine learning techniques, is a popular exercise. In particular the MNIST dataset of images of handwritten digits is a well-known dataset that has been used to develop convolution neural networks (CNNs) (Ciresan et al., 2011) and support vector machines (Platt, 1998), among other machine learning techniques. In this chapter we will create both hand-crafted and trained neural networks to recognize digits, using a simplified dataset created by drawing pixels into a grid of 6×6 pixels.

The hand-crafted model uses a convolution layer to identify certain types of pixels as features, which are then counted by another layer and passed to an output layer. Ideas seen in previous chapters are used again in this model. The trained model uses a smaller version of this architecture, and the goal is to try to identify parts of the hand-built model that can be used to enhance the performance of the trained model and increase the efficiency of the training process.

An image is represented by a matrix $\mathbf{X} \in M_{6,6}\{0, 1\}$, where an entry of 1 represents a black pixel and a 0 represents a white pixel. A valid image containing a digit will have a corresponding label $y \in \{0, \ldots, 9\}$. Each digit will be drawn using horizontal and vertical lines only, and will only be drawn in one "variation". Examples can be seen in figure 7.1. There are some notable rules for certain digits, with the rest of the digits restricted from only using horizontal and vertical lines.

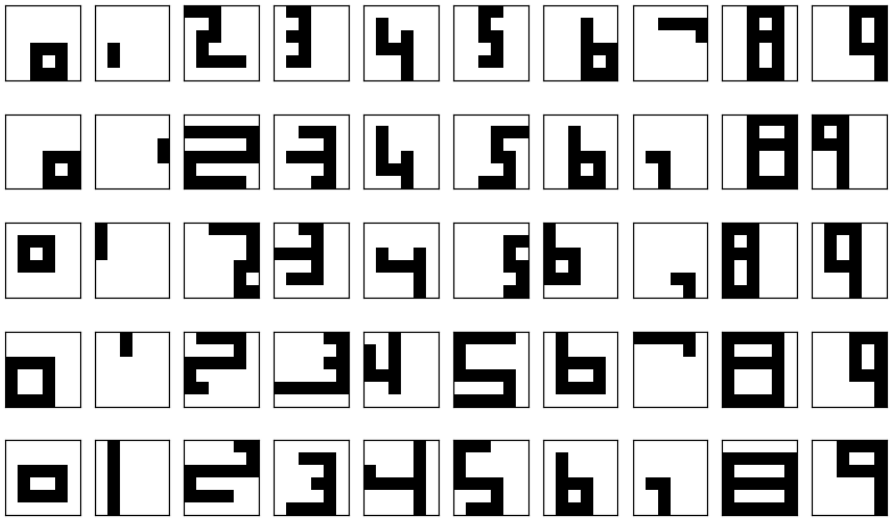

Figure 7.1: Examples of each digit. Notable rules are: 4s do not create a cross with the right vertical line; similarly 7s do not have a horizontal cross; 1s are a single vertical line with no bottom or top lines; 6s and 9s do not have a corner at the top and bottom respectively.

7.2 Methods

The methods described in this section are implemented in the file `https://github.com/HectorLeitch/mas6041/blob/main/code/digits.py`.

7.2.1 Exact Model

Our first model is a hand-crafted exact model. This model uses similar ideas to the model created to recognize letters and words. In this case, each digit is broken down into a collection of pixel types that are recognized by convolution kernels applied to the input image, and then the collection of detected pixels of each type are combined to create the correct output.

The first layer of the neural network is a convolution layer. There are 10 pixel types, shown in figure 7.2, and so the layer has 10 channels, each with its own convolution kernel. There are three "groups" of pixel types: pixel types 0 to 3 are line ends, pixel types 4 to 7 are corners, and pixel types 8 and 9 are the "T junction" pixels. The assignment of a certain pixel to a particular type applies to the pixel in the middle of the 3×3 kernel as it passes over the image; as such, we need to design the kernels slightly differently to those in section 5.2.1, to account for possible variations in the surrounding pixels.

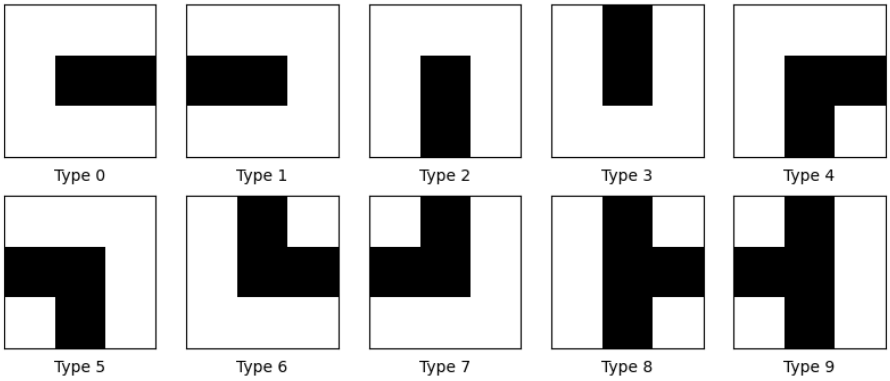

Figure 7.2: All 10 pixel types. Detection of these pixels will differ from section 5.2.1, so these should not be thought of as representations of the convolution kernels. Note that the pixel type is assigned to only the pixel at the centre of each image, but is determined by the surrounding pixels.

The convolution kernels are matrices in $M_{3,3}\{-1, 0, 1\}$. A 1 is placed in the black positions in figure 7.2, corresponding to the pixels we expect to find a value of 1 in for each type. A 0 is placed in positions where we may or may not find a value of 1 in the image. For example, the top and bottom- right values are set to 0 in the kernel for pixel type 0, as such a pixel could be found in the image as any member of the set

$$\left\{ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \right\}.$$

Similarly, a -1 is placed in positions where a value of 1 in the image would mean that the pixel is of a different type. So even though the matrix

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

contains the pixels for a type 0 pixel, the value of 1 in the top middle position means it is a type 6 pixel. Letting $\mathbf{K}_i$ denote the convolution kernel that detects pixel type i, the full list of convolution kernels in the first layer is as follows:

$$\mathbf{K}_0 = \begin{pmatrix} -1 & -1 & 0 \\ -1 & 1 & 1 \\ -1 & -1 & 0 \end{pmatrix} \quad \mathbf{K}_1 = \begin{pmatrix} 0 & -1 & -1 \\ 1 & 1 & -1 \\ 0 & -1 & -1 \end{pmatrix}$$
$$\mathbf{K}_2 = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 1 & -1 \\ 0 & 1 & 0 \end{pmatrix} \quad \mathbf{K}_3 = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 1 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$
$$\mathbf{K}_4 = \begin{pmatrix} -1 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & -1 \end{pmatrix} \quad \mathbf{K}_5 = \begin{pmatrix} 0 & -1 & -1 \\ 1 & 1 & -1 \\ -1 & 1 & 0 \end{pmatrix}$$

$$\mathbf{K}_6 = \begin{pmatrix} 0 & 1 & -1 \\ -1 & 1 & 1 \\ -1 & -1 & 0 \end{pmatrix} \quad \mathbf{K}_7 = \begin{pmatrix} -1 & 1 & 0 \\ 1 & 1 & -1 \\ 0 & -1 & -1 \end{pmatrix}$$
$$\mathbf{K}_8 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad \mathbf{K}_9 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Similar to section 5.2.1, and specifically equation 5.1, the convolution kernel weights are combined with a bias and ReLU activation function, so that the output in channel i for a particular pixel is 1 if the pixel is of type i and 0 otherwise. The biases are determined in the same way as the number of 1s in the kernel, minus one. This gives the bias vector

$$\boldsymbol{b}_1 = \begin{pmatrix} -1 & -1 & -1 & -1 & -2 & -2 & -2 & -2 & -3 & -3 \end{pmatrix}.$$

Finally, there is zero-padding applied to the input in the first convolution layer, so that the output for each channel is a feature map that is also size 6×6.

The second layer of the model counts the number of occurrences of each pixel type. This is enough to distinguish every digit apart from 2 and 5, which both have one pixel of type 0, 1, 4, 5, 6, and 7. To distinguish 2 and 5, the model must evaluate the relative positions of the pixels of type 4, 5, 6, and 7. While the feature map could be flattened into a 10 vectors of length 36, and provided to a dense layer to perform this step, it is more convenient to use another convolution layer with a 6×6 kernel for each channel. With no zero-padding, this results in a single output per channel, 10 of which are the counts for the pixel types.

In total, the second layer has 17 output channels and the output of the first layer has 10 channels, so we need to assign the weights for 170 kernels of size 6×6. For the channel corresponding to the detection of pixel type $i \in \{0, \ldots, 9\}$, the i-th kernel will be set to a 6×6 matrix of ones $\mathbf{1}_{6\times6}$ and the other channels will be set to a 6×6 matrix of zeros $\mathbf{0}_{6\times6}$. This counts the number of detected pixels in each channel from the first layer.

The kernels that determine the positions of pixels of type 4, 5, 6, and 7, are based on the matrix

$$\mathbf{H} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 & 5 \end{pmatrix}.$$

For digits 2 and 5, the first layer of the model detects one pixel of types 0, 1, 4, 5, 6, and 7. For the feature channels that contain the detected type 4, 5, 6, and 7 pixels, multiplying the feature output by $\mathbf{H}$ element-wise returns the number of pixels from the top that the pixel is on, and similarly multiplying the feature

output by $\mathbf{H}^T$ element-wise returns the number of pixels from the left that the pixel is on.

A 2 will be indicated by the pixels of type 4 and 6 appearing below the pixels of type 5 and 7, and conversely a 5 will be indicated by the pixels of type 4 and 6 appearing above the pixels of type 5 and 7. We also need to check that the pixels of type 4 and 6 are vertically aligned, and the pixels of type 5 and 7 are vertically aligned. Creating indicators for these conditions ensures that digits 2 and 5 will be detected, and the model is also able to reject configurations such as the one shown in figure 7.3.

Figure 7.3: This configuration has the heights of the pixels of type 4 and 6 equal to the heights of pixels of type 5 and 7 respectively. The pixels of type 4 and 5 could be raised or lowered, so we need to check this shape is not present by confirming the vertical alignment of the pixels instead.

Let h_p and w_p be the number of pixels that the detected pixel of type p is from the top and left respectively (noting that an increase in h_p indicates the pixel is lower in the image), obtained by multiplying the corresponding feature output by $\mathbf{H}$ and $\mathbf{H}^T$. We now re-purpose equation 5.2 with these values to create the required indicators. We have

$$\begin{aligned} I(h_4 + h_6 > h_5 + h_7) &= (h_4 + h_6 - h_5 - h_7)_+ - (h_4 + h_6 - h_5 - h_7 - 1)_+ , \\ I(h_5 + h_7 > h_4 + h_6) &= (h_5 + h_7 - h_4 - h_6)_+ - (h_5 + h_7 - h_4 - h_6 - 1)_+ , \\ I(w_4 = w_6) &= (w_4 - w_6 + 1)_+ - 2\,(w_4 - w_6)_+ + (w_4 - w_6 - 1)_+ . \end{aligned}$$

Evaluating these in the output layer requires 7 extra channels in the second layer. The kernels corresponding to the feature output for the pixel of type p are set to $\pm\mathbf{H}$ or $\pm\mathbf{H}^T$ depending on whether the coefficient for each h_p or w_p is 1 or -1 in the corresponding component. A bias of 1 or -1 is added to complete these terms. All other weights and biases are set to 0. The ReLU activation function is then applied to the layer as a whole to create the components of the indicators above; the counts in the first 10 channels are unaffected by this.

The output layer is a dense layer which functions using a variation of equation 5.1. There are 10 outputs corresponding to each digit from 0 to 9; the weights connected to the output corresponding to each digit are 1 where the connection comes from pixel types are expected to be detected within the digit, and -1 where they are not expected to be detected. From indicator component channels of the second layer, the weights are 0 apart from the weights connected to the outputs corresponding to digits 2 and 5, where the weights are assigned to construct the indicators that should be true for each output. For a digit 2, we should have

$$\begin{aligned} I(h_4 + h_6 > h_5 + h_7) &= 1, \\ I(h_5 + h_7 > h_4 + h_6) &= 0, \\ I(w_4 = w_6) &= 1, \end{aligned}$$

and for a digit 5, we should have

$$\begin{aligned} I(h_4 + h_6 > h_5 + h_7) &= 0, \\ I(h_5 + h_7 > h_4 + h_6) &= 1, \\ I(w_4 = w_6) &= 1, \end{aligned}$$

The bias for each output is the negative of the number of ones in the weights corresponding to each output, minus one. The bias vector is therefore

$$\boldsymbol{b_3} = \begin{pmatrix} -3 & -1 & -7 & -5 & -4 & -6 & -4 & -2 & -5 & -4 \end{pmatrix}.$$

The ReLU activation function is applied to the result, so that the output of this layer when a digit is present is the one-hot encoding representing the detected digit, and if no digit is present the output is a vector of zeros $\mathbf{0}_{10}$.

7.2.2 Approximate Convolution Neural Networks

We now create an approximate model that can be trained using the dataset containing all the possible images containing digits. The model uses the same architecture in terms of the layer types and their activation functions, but the convolution layers only have 3 channels each. If a model can achieve perfect performance on the training data, we are able to investigate the final weights in visualizations using colours, or 3-channel red/green/blue (RGB) values.

The process for finding suitable training values closely followed the process previously described in section 6.2.3. However, it was found that experimenting with the *patience* (the number of epochs over which no improvement would cause training to stop early) was necessary due some of the common behaviour found with this network architecture and training data. The parameters were adjusted by examining plots of the loss over time, and values were chosen once we were able to reliably train a model with satisfactory performance.

The model performance for the approximate models was assessed by counting the number or percentage of correct classifications made on the dataset containing

all the possible images containing digits, ignoring any further images that did not contain digits.

7.2.3 Model Improvements

The goal of the models created in this section was to improve the stability of the training and the overall improvement of the approximate CNNs. The strategy implemented was to repeat the training process with the established parameters, but initializing the weights and biases of the first convolution layer using the weights and biases from the first layer of the hand-crafted exact model, corresponding to a subset of the 10 channels used to identify pixel types.

As there are 10^3 subsets we can choose from to initialize the weights in this manner, it was impractical to run an experiment that tested training at least one model using each possibility. Therefore, we chose to experiment with a smaller subset. Five models were trained where the weights and biases for the first layer were initialized using three copies of the kernel and bias used to detect each pixel type from the hand-crafted exact model. This resulted in 50 total models trained, which did not take an excessive amount of time to run.

If it made sense to do so, we could have extended this experiment by repeating the experiment, but only using two copies of each kernel while fixing the third kernel to be the one that achieved the best performance in the first round of tests. A final round of tests using the best two-kernel selection from the second round would give us a selection of three starting kernels that we believe to be among the best possible using this idea. However, the results proved that only the first round of tests was necessary to achieve a satisfactory improvement in performance.

7.3 Results

The models described above are demonstrated in the Jupyter notebook `https://github.com/HectorLeitch/mas6041/blob/main/try_digits.ipynb`. The notebook is split into examples to demonstrate the code, which are described in the following paragraphs.

Example 7.1 The function `make_exact_model()` creates the hand-crafted model which takes an input $\mathbf{X} \in M_{6,6}\{0,1\}$ and returns the one-hot representation of a recognized digit, or a vector of 0s if no digit is detected. The function `model_classify(M=, img=)` shows the output for the model `M` evaluated on the image `img` as an integer, or shows "reject" if no digit is found.

The example code shows the model working on individual random images, and a selection of images without digits in them to show that it properly rejects certain cases. The function `check_model()` evaluates the model on the full set of possible inputs containing digits, showing that it is correct in every case.

Example 7.2 The function `make_approx_model(p=3, q=3, r=)` creates an approximate model sharing the same layer architecture as the exact model, but with $p = q = 3$ channels in the convolution layers. The function `train_approx_model(M=, epochs=, batch_size=, patience=)` implements the training of the model, with the training parameters free to be modified.

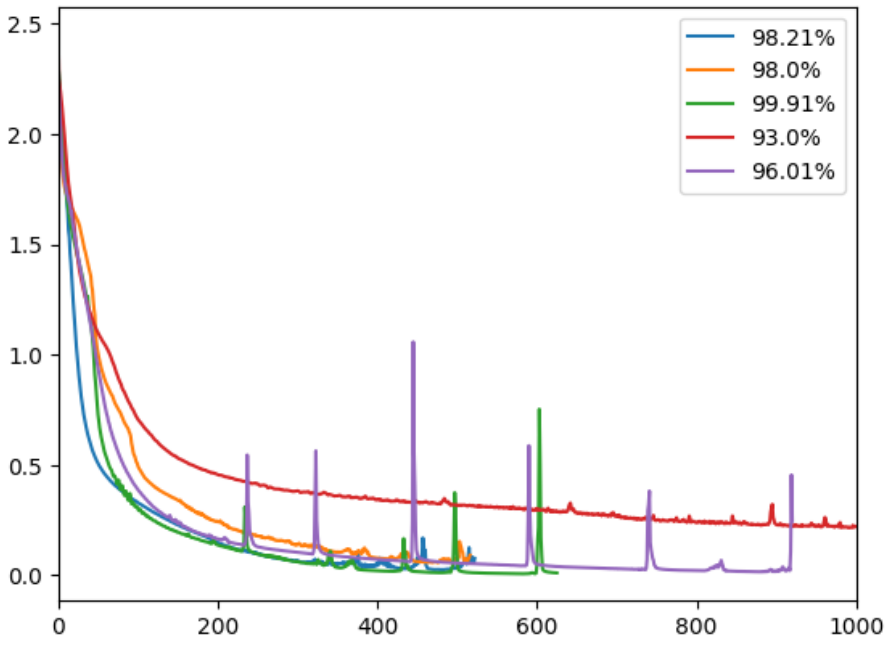

Figure 7.4: Loss over time and final percentage accuracy for five models using the same values for the training parameters; the differences are therefore due to the random initialization. There is a reasonable variety in how many epochs were needed, with one run not meeting the early stopping criteria and performing noticeably worse.

It was found that the training of this model was quite sensitive to the random initialization of the weights. Figure 7.4 shows the loss over time for five different runs using the same values for the training parameters: the learning rate was set to 0.003, and the values for epochs, batch size, and patience were 1000, 128, and 40 respectively. Furthermore, figure 7.5 shows the kernels for the first layer after training. It can be seen that there is no clear pattern in how the first layer is creating features.

However, with the training parameters set to these values, quite strong performance could be reliably achieved with the models tending to classify >96% of the inputs correctly. This was deemed satisfactory performance to use these values for the training parameters going forward.

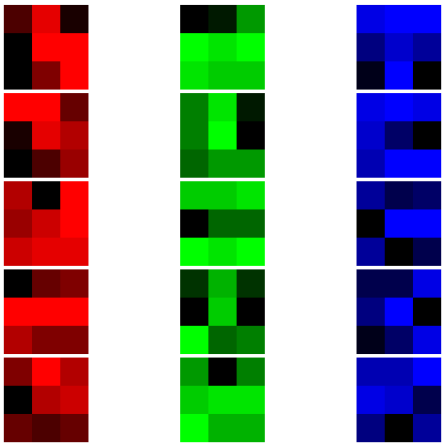

Figure 7.5: Kernels for the first layer in each trained model. Each of the five models occupies one row. While some of the kernels resemble the ones we used to detect certain pixel types, there doesn't seem to be a clear pattern to how the first layer is creating features.

Example 7.3 The function `test_starting_kernels(sk=, r=, epochs=, batch_size=, patience=)` creates an approximate model sharing the same layer architecture as the exact model, but with $p = q = 3$ channels in the convolution layers. It then initializes the kernel weights in the first layer using the kernels used to detect pixels types corresponding to the types listed in the `sk` parameter. The model is then trained using the values for the training parameters chosen from the previous example.

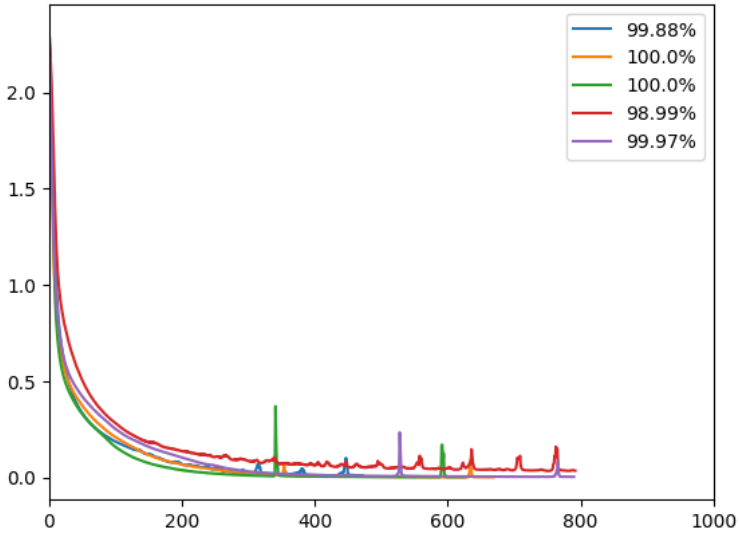

Figure 7.6: Loss over time and percentage accuracy for five models, starting with kernels $\mathbf{K}_0$, $\mathbf{K}_6$, and $\mathbf{K}_9$. The stability of the training procedure has been vastly improved and the performance of the models is very strong, sometimes resulting in a 100% score. Note that the code can be used to demonstrate that these results are representative of most other combinations.

It was found that using virtually any combination of starting kernels resulted in a much more stable training procedure, with stronger end results than the models trained from randomly initialized weights. Figure 7.6 shows the loss over time for five models, starting with kernels $\mathbf{K}_0$, $\mathbf{K}_6$, and $\mathbf{K}_9$.

We can also see that there is a clear pattern in the final kernels in the first layer after training in figure 7.7.

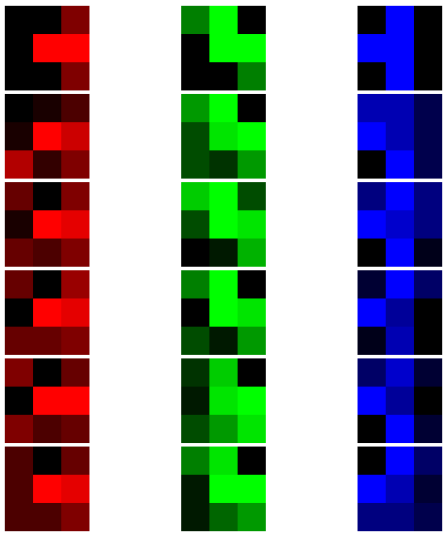

Figure 7.7: Kernels for the first layer in each trained model, starting with kernels $\mathbf{K}_0$, $\mathbf{K}_6$, and $\mathbf{K}_9$. Here we can see that the starting kernel is often clearly visible in its corresponding final kernel. The first row shows the starting kernels for this selection of models.

The fact that the kernels in the first layer still resemble the starting kernel for each channel, and consistently do so, is in correlation with the improved stability of the training procedure and performance of the resulting models. We do not necessarily know that this is the cause of the improvement but the design of the experiment suggests it is so.

Example 7.4 The function `test_all()`, performs the experiment of comparing the training of models using three copies of the kernel used to detect each pixel type as initialized weights for the first layer. Table 7.1 shows the tabulated results of the choice of starting kernel against the mean percentage of the inputs that were correctly classified across the five models.

Figure 7.8 shows the kernels for the first layer of each model after training for the models using $\mathbf{K}_9$ as the starting kernel. Interestingly, there appears to be more deviation from the starting kernel in this case, which could be due to using three copies in the initialization. While the choice of kernel was sensitive to the random seed, this behaviour was observed consistently when rerunning the experiment using the notebook.

Starting kernel	Correct classifications (%)
$\mathbf{K}_0$	97.79
$\mathbf{K}_1$	98.00
$\mathbf{K}_2$	99.85
$\mathbf{K}_3$	99.69
$\mathbf{K}_4$	99.92
$\mathbf{K}_5$	98.72
$\mathbf{K}_6$	99.72
$\mathbf{K}_7$	99.40
$\mathbf{K}_8$	99.48
$\mathbf{K}_9$	99.99

Table 7.1: Results for each choice of starting kernel. Many of the choices averaged close to 100% correctness and while $\mathbf{K}_9$ was the best performer, variation in the initialization of the weights and biases in the other layers account for the difference. Rerunning this test with a different seed can produce a different result, but with a similar interpretation.

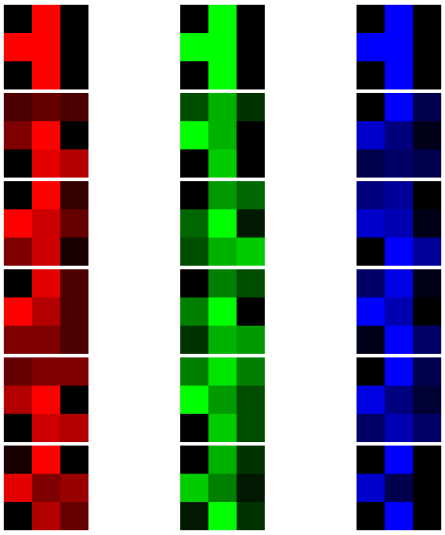

Figure 7.8: Kernels for the first layer in each trained model, starting with kernels $\mathbf{K}_0$, $\mathbf{K}_6$, and $\mathbf{K}_9$. Here we can see that the starting kernel is often clearly visible in its corresponding final kernel, but less so than in figure 7.2.

7.4 Discussion

7.4.1 Model Architecture

The exact model in this chapter followed a similar design philosophy to the letter and word detection exact model, where the first convolution layer is designed with kernels matching the structure of the pixels they are intended to detect, with the biases and ReLU activation function restricting the outputs to $\{0, 1\}$.

The designs in this chapter required a little more thought as it was possible that pixels of certain types could be next to each other within a 3×3 space.

7.4.2 Training Behaviour

The approximate models in this chapter proved slightly more challenging to train than those in previous chapters. The trained models seemed to be highly sensitive to the initialization of the weights and biases. Adjusting the learning rate and batch size was quite difficult as large spikes where the loss increased were frequently observed. This may have affected whether or not the stopping criterion was met, so this motivated the decision to add the patience to the stopping criterion as a parameter in an attempt to minimize the impact of a sudden increase in the loss.

The models using fixed kernels suffered much less from these issues. It is possible that in choosing weights and biases that were much "better" than randomly chosen ones, the model was able to train as if it had fewer parameters. A possible extension to the experiments performed would have been to not let the model train the weights and biases in the first layer and see if this limited the performance, in a similar fashion to what was done with the residual connections in the previous chapter. However, as the final models chosen in the final part of the experiment were often able to reach 100% accuracy, it was decided to spend more time on the other chapters in this project.

Chapter 8

Conclusion

8.1 Summary of Completed Work

The overall goal of this project was to use small neural networks to tackle a variety of problem scenarios, with a focus on using the design of the network architecture to yield models that worked in a way that could be understood by the user, and were less of a "black box" compared to traditional deep neural networks trained on large datasets.

Chapters 2 to 5 were mainly focused on building neural networks by hand to achieve a specific task. Some of the problems could be solved applying key results through a specific layer architecture, leading to exact solutions where this was more difficult to achieve with a model using training data. In some cases, the initial designs could be improved upon or generalized. It was found that this became increasingly difficult as the models grew larger.

Chapters 6 and 7 mainly focused on training neural networks for slightly more complex tasks, and adapting the network architecture in an intuitive fashion to try to improve the model performance. It was found in chapter 7 that elements from the hand-crafted model were effective in improving the performance and efficiency of these models. The models were demonstrated using Jupyter notebooks, each supported by custom functions written in a source file, that can be downloaded and modified to verify the robustness of the models and explore the networks in detail.

8.2 Discussion

Each chapter contains its own discussion of particular items of interest that arose working on the corresponding problems. However, there were a few common themes which are worth highlighting.

From the chapters involving hand-crafted models, it became clear that where the input domain was restricted to matrices or vectors of integers, a good strat-

egy was to use weights and biases that were also integers. This restricted the domain at each layer to vectors or matrices of integers only, allowing key results such as equations 2.5 and 5.1 to be applied. The ReLU activation function was the obvious choice to pair with these designs to increase the capabilities of certain layers, as it introduced a further restriction to non-negative integers. The main challenge in developing these models was implementing them in Tensorflow/Keras, where it became difficult to ensure the size and dimensions of each tensor object were correct.

From the chapters involving trained models, the adaptations to the model architectures were generally successful in achieving their goal of improving the model performance. The models were often able to achieve 100% accuracy with relatively little training after improvements, depending on the random seed set in the program. This was certainly a success according to the original aims of the project, as it was demonstrated that the improvements came from changing the network architectures rather than modifying the training process. However, the ability to accurately and reliably quantify the improvements in each scenario was restricted by the available computing resources and time.

8.3 Further Work

The work done in each chapter tackled each problem from scratch and, therefore, we have discussed possible extensions or improvements to the work that directly follow the work done in each case. It is also possible that some of the findings in this project could be more generally applied to other problems. There seems to be little research published regarding small neural networks in particular, so it is possible that continuing to expand these examples would provide useful findings for designing and optimizing smaller networks.

The design of a neural network architecture is a process itself that can be automated and tackled using a machine learning-based approach. Luo et al. (2018) and Pham et al. (2018) propose algorithms for efficiently optimizing the architecture of a network through exploring an *architecture space*. The results of chapters 6 and 7 suggest that for specific problems, particular human choices could be made to steer the optimization of a neural network architecture through reducing this architecture space, based on a human's understanding of the specific problem.

Appendix A

Research Ethics Approval

Ethics approval was not required for this project; the following letter was signed prior to submission of the project to confirm this.

Form 1: Student declaration (for research that does not involve human participation or analysis of secondary data)

School of Mathematics and Statistics

Research Ethics Review for Postgraduate Taught Students

Dissertation title: Small Neural Networks with Interesting Architectures

In signing this student declaration I am confirming that:

My project does **not** involve people participating in research either directly (e.g. interviews, questionnaires) and/or indirectly (e.g. people permitting access to data).

My project does not therefore require an ethics review and I have not submitted a Research Ethics Application Form.

Name of student: Hector Leitch

Signature of student: Hector Leitch Date: 01-May-2024

Name of supervisor: Kevin Walters

Signature of Supervisor: Kevin Walters Date: 01/05/2024

Figure A.1: The student declaration form provided for the work outlined in this dissertation, following compliance with the University of Sheffield offical research ethics processes.

Appendix B

Tensorflow/Keras Example

The following code is included to show some of the challenges that made progress difficult at the start of this project. Naturally, as more time was spent on the project, these challenges started to become easier to handle and avoid altogether on a regular basis. Within the `digits.py` code, the exact model is built using the following layer calls from `tf.keras.layers`.

```
%run "..\code\digits.py"

"""
inputs = tf.keras.Input(shape=(N, N))
reshape = tf.keras.layers.Reshape((N, N, 1))(inputs)
features = tf.keras.layers.Conv2D(10, [3, 3], [1, 1], 'same',
                                  activation='relu')(reshape)
counts = tf.keras.layers.Conv2D(17, [N, N], [1, 1], 'valid',
                                activation='relu')(features)
outputs = tf.keras.layers.Dense(10, activation='relu')(counts)
exact_model = tf.keras.Model(inputs=inputs, outputs=outputs,
                             name="exact_model")
"""
```

The block below creates a random image containing a digit, which can be shown to have a shape of `(6,6)` as expected.

```
rand_img = random_image()
print(rand_img)
print(rand_img.shape)

>> [[1. 1. 1. 1. 0. 0.]
>>  [0. 0. 0. 1. 0. 0.]
>>  [0. 1. 1. 1. 0. 0.]
>>  [0. 0. 0. 1. 0. 0.]
>>  [0. 0. 0. 1. 0. 0.]
```

```
>>  [0. 0. 1. 1. 0. 0.]]
>> (6, 6)
```

The model `M` is the exact model, which should take our image as input. However, calling the model `predict()` method produces errors which are not as helpful as they could be as they state that the input shape received by the first reshaping layer is `[6]`. The problem is that the image input has not been supplied with a *batch dimension*. To get the model to work, the input must be reshaped to a `(1,6,6)` shape.

```
M = make_exact_model()
M.predict(rand_img, verbose=0)
```

```
>>  ValueError: Exception encountered when calling layer "reshap ...
>>
>>  total size of new array must be unchanged, input_shape = [6] ...
>>
>>  Call arguments received by layer "reshape_6" (type Reshape):
>>    • inputs=tf.Tensor(shape=(None, 6), dtype=float32)
```

The use of `predict()` in the following code shows how this is done in the `model_classify()` function.

```
print(M.predict(rand_img.reshape(1, N, N), verbose=0))
```

```
>> [[[[0., 0., 0., 1., 0., 0., 0., 0., 0., 0.]]]]
```

Furthermore, this input is immediately reshaped to `(1,6,6,1)`, where the last dimension is an additional *channel dimension*. This makes sense in context where it is common for multiple convolution layers of multiple channels each to be used in sequence with each other, but is easy to miss for a new user starting with a small example where the inputs do not have separate channels.

```
M1 = tf.keras.Model(inputs=M.input, outputs=M.layers[1].output)
M1.predict(rand_img.reshape(1, N, N), verbose=0).shape
```

```
>> (1, 6, 6, 1)
```

Finally, the model `M2` containing the layers up to the first convolution layer shows that the challenges extend to assigning weights. The convolution layer contains 12 channels, each with a `(3,3)` matrix as a weights kernel and a single value as the bias. In the function definition we specify the weights in the intuitive manner. However, the weights shape of `(10,3,3)` in the code below is not correct and inside the `make_exact_model()` function it is reshaped to `(3,3,1,10)`. Again, the extra dimension with value 1 is the input channel dimension.

```
M2 = tf.keras.Model(inputs=M.input, outputs=M.layers[2].output)
weights2 = [[[-1, -1,  0], [-1,  1,  1], [-1, -1,  0]],
            [[ 0, -1, -1], [ 1,  1, -1], [ 0, -1, -1]],
```

```
            [[-1, -1, -1], [-1,  1, -1], [ 0,  1,  0]],
            [[ 0,  1,  0], [-1,  1, -1], [-1, -1, -1]],
            [[-1, -1,  0], [-1,  1,  1], [ 0,  1, -1]],
            [[ 0, -1, -1], [ 1,  1, -1], [-1,  1,  0]],
            [[ 0,  1, -1], [-1,  1,  1], [-1, -1,  0]],
            [[-1,  1,  0], [ 1,  1, -1], [ 0, -1, -1]],
            [[-1,  1, -1], [-1,  1,  1], [-1,  1, -1]],
            [[-1,  1, -1], [ 1,  1, -1], [-1,  1, -1]]]
bias2 = [-1, -1, -1, -1, -2, -2, -2, -2, -3, -3]
weights2 = np.array(weights2)
bias2 = np.array(bias2)
print(weights2.shape)
print(bias2.shape)
M2.layers[2].set_weights([weights2, bias2])

>> (10, 3, 3)
>> (10,)
>> ValueError: Layer conv2d_12 weight shape (3, 3, 1, 10) is not
>>             compatible with provided weight shape (10, 3, 3).
```

Rebuilding `M2` from `M`, we can see that setting the shape to `(3,3,1,10)` has the effect that in order to view the first `(3,3)` kernel, we need to first take particular slices of the weights object. Furthermore, this must be reshaped again to display in a readable format that we can use to check for correctness.

```
M2 = tf.keras.Model(inputs=M.input, outputs=M.layers[2].output)
layer2_W = M2.layers[2].get_weights()[0][:,:,:,0]
print(layer2_W.reshape((3,3)))

>> [[-1. -1.  0.]
>>  [-1.  1.  1.]
>>  [-1. -1.  0.]]
```

Appendix C

Inclusion-Exclusion Principle

The Inclusion-Exclusion Principle is the generalization of the intuitive method for counting the number of elements in the union of two sets. For two sets A and B, we have

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

This result comes from the idea that the elements in the intersection $|A \cap B|$ are "double counted" by the sum $|A|+|B|$. This can be generalized to any collection of $n \geq 2$ finite sets.

This applies to the coefficients of section 2.2.2 as follows. Let $\boldsymbol{x}_I$ have $x_i = 1$ for $i \in I$ and $x_i = 0$ for $i \notin I$. If we set $f(\boldsymbol{x}_I) = \sum_{i \in I} a_i x_i$, this would "double count" coefficients a_j for $j \in J$, for every $J \subset I$. This applies to every coefficient down to $a_\emptyset$, which we set to $f(\boldsymbol{x}_\emptyset) = f(\mathbf{0}) = u_0$, which introduces the $(-1)^{|I \setminus J|}$ term seen in the formula.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078.

Chollet, F. et al. (2015). Keras. `https://keras.io`.

Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2011). Convolutional neural network committees for handwritten character classification. In *2011 International conference on document analysis and recognition*, pages 1135–1139. IEEE.

Cleveland, W. S. (1979). Robust locally weighted regression and smoothing scatterplots. *Journal of the American statistical association*, 74(368):829–836.

Crowley, K. and Siegler, R. S. (1993). Flexible strategy use in young children's tic-tac-toe. *Cognitive Science*, 17(4):531–561.

Fadlalla, A. and Lin, C.-H. (2001). An analysis of the applications of neural networks in finance. *Interfaces*, 31(4):112–122.

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. Ieee.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

Heck, J. C. and Salem, F. M. (2017). Simplified minimal gated unit variations for recurrent neural networks. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1593–1596. IEEE.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

Jiang, J., Trundle, P., and Ren, J. (2010). Medical image analysis with artificial neural networks. *Computerized Medical Imaging and Graphics*, 34(8):617–631.

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

Luo, R., Tian, F., Qin, T., Chen, E., and Liu, T.-Y. (2018). Neural architecture optimization. *Advances in neural information processing systems*, 31.

Pham, H., Guan, M., Zoph, B., Le, Q., and Dean, J. (2018). Efficient neural architecture search via parameters sharing. In *International conference on machine learning*, pages 4095–4104. PMLR.

Platt, J. (1998). Using analytic qp and sparseness to speed training of support vector machines. *Advances in neural information processing systems*, 11.

Ray, P. P. (2023). Chatgpt: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope. *Internet of Things and Cyber-Physical Systems*.

Sarker, I. H. (2021). Deep learning: a comprehensive overview on techniques, taxonomy, applications and research directions. *SN Computer Science*, 2(6):420.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Van Rossum, G. (2020). *The Python Library Reference, release 3.8.2*. Python Software Foundation.

Wes McKinney (2010). Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61.