Faculty of Science and Engineering

Alexander J. Johnson

Mathematics

Neural Networks with Python and TensorFlow

May 25, 2020

School of Computing, Mathematics and Digital Technology

Abstract

Artificial Intelligence (AI) still remains as one of the greatest challenges in scientific research to this date, but much progress in the field has been made using artificial neural networks. The design of artificial neural networks is loosely inspired by that of biological brains, and serves as an expansion of an earlier concept called the perceptron (Rosenblatt, 1958). By using multiple layers of these artificial neurons, we can form a highly connected system that is referred to as a neural network, these networks can then be trained on a large data set to predict the output with high accuracy.

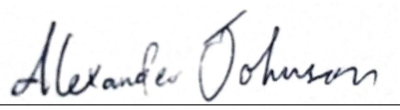
The range applications for neural networks is wide: they can be used to classify data, predict future states of chaotic systems, apply stylisations to images, or control physical/physically-based systems in real-time.

This project aims to cover the history of a wide range of artificial neural networks and their applications.

Declaration

With the exception of any statement to the contrary, all the material presented in this report is the result of my own efforts. In addition, no parts of this report are copied from other sources. I understand that any evidence of plagiarism and/or the use of unacknowledged third party materials will be dealt with as a serious matter.

Signed



Contents

Chapter 1

Introduction to Neural Networks

Biological Neurons

Biological neurons are electrically excitable cells that are found in almost all animals. These neurons can transmit and receive electrical signals to one another via synaptic connections, which maybe either excitatory or inhibitory. Any given neuron will be either active or inactive depending on whether or not its input exceeds a threshold.

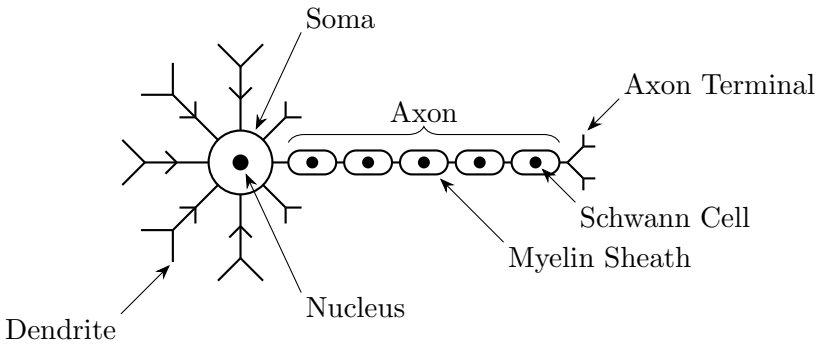


Figure 1.1: Diagram of a biological neuron.

Signals are received by the neuron via connections to dendrites and soma. If the threshold is met, electrical signals are sent along the axon to the terminal, where it is connected to other neurons, or to a controllable cell such as a neuromuscular junction.

Artificial Intelligence

The idea of artificial beings capable of human intelligence can be traced back to mythical stories from ancient Greece. One such story was that of a mythical automaton called Talos, who circled

an island's shores to protect it from pirates and other invaders.

In the 19th century, other notions of artificial intelligence were explored by fiction in stories such as Mary Shelley's "Frankenstein", and Karel Čapek's "R.U.R.". Some of the fictional writings of the 20th century further continued exploring the concept in novels such as Isaac Asimov's "I, Robot".

Academic research into artificial intelligence began around the 1940's, primarily due to findings in neurological research at the time. The first explorations of artificial neural networks was done by McCulloch and Pitts, 1943, who investigated how simple logic functions might be performed by idealised networks. The neurons within these networks operated using some basic logic rules, applied to a discrete time system which, can be summarised using the expression

$$N(t) = (E_1(t-1) \vee E_2(t-1) \vee \dots) \wedge \neg(I_1(t-1) \vee I_2(t-1) \vee \dots),$$

where $N(t)$ is the state of a neuron at time t, and $E_i(t-1)$ and $I_i(t-1)$ are the states of the excitatory and inhibitory connections from the previous time step respectively. The result is such that the neuron will only be active if at least one excitatory connection is active and all inhibitory connections are inactive. The versatility of this definition is demonstrated in the following examples.

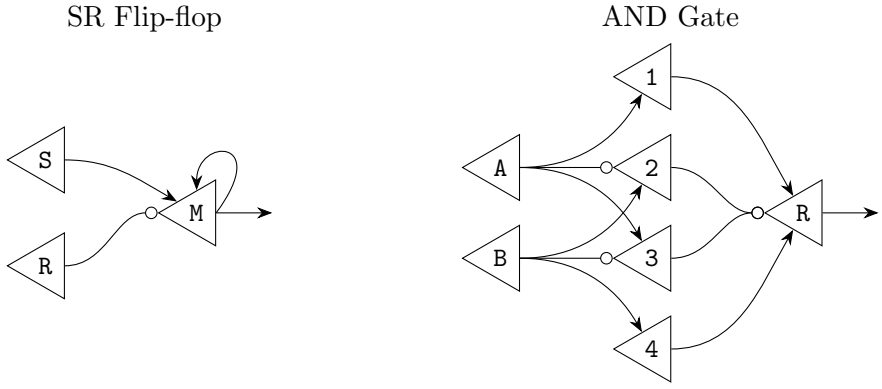


Figure 1.2: Two common logic circuits using McCulloch's neurons where the arrows and circles indicate excitatory and inhibitory connections respectively.

While this model provided insight into the mechanisms by which neurons operate, the structure was static, and incapable of learning.

McCulloch's work was later cited by psychologist Hebb, 1949, as important for understand-

ing how logical operations are performed; but proposed that the structure of biological neurons was dynamic, not static, and that frequently repeated stimuli caused gradual development. At the scale of neuron, it was theorised that if one neuron successfully excited another, the connection between them would strengthen, hence increasing the likelihood that the former would be able to excite the latter again in the future. His theory was supported by research conducted by himself and others that showed that intelligence-test performance in mature patients was often unaffected by brain operations that would have prevented development in younger patients, which suggested that learnt stimuli are processed differently to unknown stimuli. This hypothesis became known as Hebbian learning.

Computer simulations applying this theory to a small network were done by Farley and Clark, 1954. The actions of the network were compared to that of a servo system which must counteract any displacements so as to maintain a steady position. The network was trained using a set of input patterns, which were subject to non-linear transformations. Similar to the Hebbian theory, when the network produces the correct responses, the active connections are strengthened. Although the results were of little neurophysiological significance, the results were of great use for demonstrating computational simulations, which were considerably slower at the time.

Perceptrons

The idea of the perceptron was originally conceived by Rosenblatt, 1958, to represent a simplified model of intelligent systems free from particularities of biological organisms, whilst maintaining some of their fundamental properties.

The perceptron was built as a dedicated machine that consisted of a number of photovoltaic cells, analogous to a retina, that feed into an"association area". This association area contains a number of cells that each calculate a weighted sum of the receptor values and output a signal if it exceeds a threshold. Expressed mathematically, the output of a given association cell is given by

$$A_i = \begin{cases} 1, & \sum_j w_{i,j} x_j > \theta \\ 0, & \text{otherwise} \end{cases},$$

where x_j is the value from the j^{th} photovoltaic cell, $w_{i,j}$ is the weight of the connection between

association cell i and photovoltaic cell j, and θ is the threshold. These value weights were implemented using variable resistance wires that the perceptron could adjust automatically. The outputs from the association area are then connected to response cells, which operate in a similar fashion to the association cells. The activation of these response cells are the outputs of the perceptron, and indicated the classification of the input.

Similar to Farley and Clark, 1954, the method by which the perceptron adjusted it weights was also based on which cells were active, and whether the correct output was produced; except that the perceptron was also able to "penalise" weights when an incorrect result was outputted.

This machine was initially trained to reliably identify three different shapes: a square, a circle, and a triangle; and did so with a better than chance probability. When attempting to use the perceptron for more complicated tasks, such as character recognition, it failed to produce better than chance results.

After a decade of unsuccessful real world application attempts, a book titled "*Perceptrons: An introduction to computational geometry*" by Minsky and Papert, 1969, was released. The book provided a rigorous mathematical analysis of the model, the results showed that single layered, simple linear perceptron networks could not calculate XOR predicates. A 2017 reissue of the book contained a foreword by Léon Bottou, who wrote "Their rigorous work and brilliant technique does not make the perceptron look very good..."

Following the book's release, perceptron research effectively halted for 15 years until the first successful uses of multilayer networks by McClelland, Rumelhart, and Group, 1986, which also served as a departure from the neuron outputs being boolean values. The multilayered structure of this new model allowed it to calculate the XOR predicates that the single layer perceptrons could not.

The output of the units within these networks were defined by

$$\mathbf{a}(t+1) = \mathbf{F}(\mathbf{a}(t), \mathbf{net}_1(t), \mathbf{net}_2(t), ...),$$

where $\mathbf{net}_i$ is the i^{th} propagation rule applied to the inputs, $\mathbf{F}$ is the activation function, and $\mathbf{a}(t)$ is the activation of the units at time step t. The model usually uses a simplified version which can be summarised as

$$a_i = F\left(\sum_j w_{i,j} o_j\right),$$

where o_j is the output of unit j. Hebbian learning could be performed the network by using iterative methods, the most simple of which was given by

$$\Delta w_{i,j} = \eta\, a_i o_j,$$

where η is the learning rate, which is a constant.

Backpropagation

In order for a neural network to learn, it must undergo some form of optimisation process. For the perceptron, this process was one of positive and negative reinforcement.

In the field of control theory, an optimisation method known as gradient descent was developed by Kelley, 1960, in which a given function of the system is either maximised or minimised. This is achieved by taking partial derivatives of the function with respect to each parameter, which gives an approximation of how the function value will change as the parameter changes. By evaluating the partial derivatives, multiplying them by a constant, and adding them to their respective parameters, the parameter values can be updated. Using these new parameter values, one can expect to improve the function value. This can be written as

$$w_i' = w_i + \eta \frac{\partial f}{\partial w_i}(\mathbf{x}),$$

where $f(\mathbf{x})$ is the function to be optimised, w_i is a parameter of f, w_i' is the updated parameter, and η is ascent/descent parameter. Positive η values will maximise the function value, where as negative values will minimise it. The magnitude of η determines the rate at which the method will attempt change the parameters: if the value is too large, the method will overshoot the optimal values; if the value is too small, the method will be too slow to converge. This method is known as stochastic gradient descent.

When the method was applied to neural networks, researchers sometimes encountered an issue now known as the vanishing gradient problem. A computer program will typically calculate the gradient via repeat applications of chain rule; if there are many small terms, the gradient will tend to zero, and the learning rate of the network will be minimal.

One of the methods that overcame this problem was developed by Schmidhuber, 1992, where each layer of the network was pre-trained to predict the next input from previous inputs. Once each layer had been pre-trained, the network was then fine tuned using backpropagation. The

method also provided a way of calculating which inputs were least expected, so that more training time could be devoted to learning them.

Since then, computational power has significantly increased, and the slow convergence caused by the vanishing gradient problem is less significant. Further more, backpropagation and a simple variant the model outlined by McClelland, Rumelhart, and Group, 1986, have become the standard for neural networks. Namely

$$x_i = \phi\left(b_i + \sum_j w_{i,j} x_i\right),$$

where x_i is the output of neuron i, $w_{i,j}$ is the weight of connection from j to j, b_i is the input bias of i, and ϕ is the activation function.

Other Types of Artificial Neural Networks

The preceding discussion has been focused on densely connected neural networks, where each neuron in a layer is connected to every neuron in the previous, but it is important to note, that many other neural network architectures are often used together, and maybe more suitable under certain contexts.

Convolutional Neural Networks

Many of the neural networks that had been employed for image recognition, such as the preceptron, suffered from two major issues:

1. processing high resolution images required each neuron in the first layer to be connected to every input, which caused the number of connections to become too large to process; and,

2. most networks could not correctly identify an input if it was shifted.

Similar to how biology inspired neural networks, findings in neurophysiology inspired the architectures that would overcome these issues. Hubel and Wiesel, 1959, discovered that certain cells within a cat's visual cortex would only respond to stimuli from specific regions of the retina. Another important observation was that neighboring cells had overlapping response regions.

Later research by Hubel and Wiesel, 1962, also distinguished two categories of cells termed: "simple", which had distinct excitatory and inhibitory connections, where firing was maximised

by slits at specific angles that passed through the centre; and "complex", which could not be mapped out as trivial inhibitory/excitatory regions, but were maximised slits at specific angles, regardless of position.

These findings inspired Fukushima, 1980, to design the neocognitron. The neocognitron featured alternating layers of "S-Planes" and "C-Planes", which were representations of the simple and complex cells respectively. Each plane contains a number of feature maps, each unit within a feature map is a function of a small region of the previous layer, a process now commonly referred to as convolution. S-Plane feature maps connect to all feature maps of the previous layer, but C-Plane feature maps only connect to the corresponding feature map.

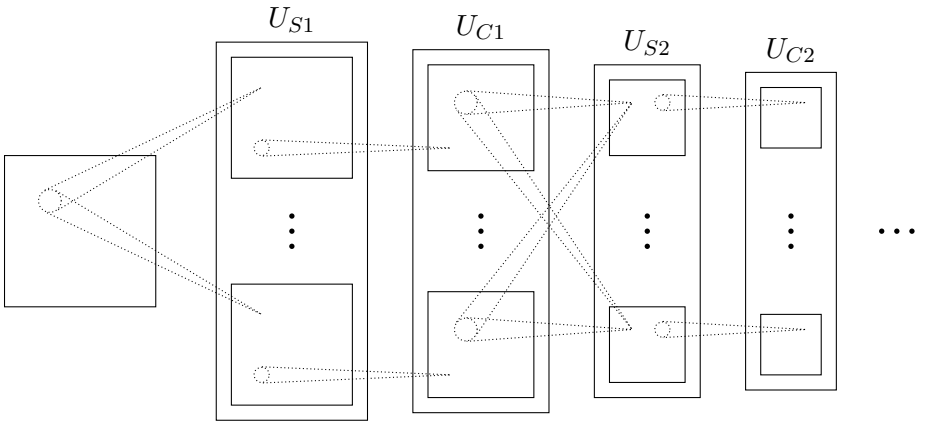


Figure 1.3: Representation of the neocognitron's connectivity.

Each layer of the network reduces the size of input image until the final layer consists of single unit feature maps. The network was originally trained to distinguish 5 digits, numbers 0 to 4, using an unsupervised learning method, and was the first to reliably handle shifted inputs.

Waibel et al., 1989, used concepts from the neocognitron to design the time delay neural network, which was originally proposed for phoneme recognition. The networks were initially trained using backpropagation to detect and distinguish between three acoustically similar phonemes (/b/, /d/, and /g/). The model consisted of units similar to the neocognitron's S-Planes, where the output of a unit is a function of a region from the previous layer. The input for the model was a 2D spectrogram of an audio sample, with each column representing a set of 16 spectral coefficients for a given time frame.

The first hidden layer contained columns of 8 units, each of which convolved the spectral coefficients across 3 time frames, requiring a total of 384 weights. Similarly, the second hidden layer contained columns of 3 units, each convolving the previous layer across 5 time frames,

requiring a total of 120 weights. Finally, the output layer contains three units, each of which is a function of the previous layer's corresponding row sum. The most active unit is the phoneme present in the audio.

Once trained, the network was able to detect the correct phoneme, in real time, under a variety of contexts, with an error of 1.5%, a significant improvement over the 6.3% error from the most popular method at the time.

Similar techniques were used by LeCun et al., 1989, to classify handwritten digits. A large number of 16 by 16 pixel images were used to train the network using backpropagation. The network consisted of two convolutional layers, and two dense layers. Although the all of the units in a feature map shared the same weights, each had a unique, adjustable bias. Once trained, the network correctly classified 99.86% of the training data, and 95% of the test data.

This technique of combining convolutional and dense dense layers was also used by Yamaguchi et al., 1990, for speech recognition. Similar to Waibel et al., the network takes a 2D spectrogram as it's input and predicts which word was spoken.

The first layer, referred to as the event-net, convolves the input spectrogram using a two layer subnetwork, which consists of a hidden partially connected layer, and a fully connected single unit output layer. Backpropagation is used to train each subnetwork of the event-net to only fire when a word of the corresponding category is inputted.

The second layer, referred to as the time-alignment procedure, performs an operation now known as max pooling, where each unit of the layer is the largest value from the response region. If all the values are similar, use the one from the middle of the search range; if all of the values are very low, the response region size is increased

The third layer, referred to as the word-net, is another convolutional layer using subnetwork, consisting of a fully connected layer, and a full connected single unit output layer. Backpropagation is used again in the same manner as in the event-net.

The forth layer, referred to as the super-net, is another convolutional subnetwork that takes N inputs and densely connects to $N + 1$ outputs. Each output corresponds to a word, except the last which denotes a rejected result.

Finally, a decision algorithm compares the two highest outputs and rejects the result if the difference does not exceed a preset threshold.

Recurrent Neural Networks

So far, the progression of time has been implemented as being another spacial dimension. Although this methodology proved effective for phoneme recognition tasks, it did not work for other, more advanced temporal tasks.

Jordan, 1986, noted that representing time as a spacial dimension required the network inputs to be stored in a buffer. This buffer storage method was susceptible to a number of problems, including:

- inability to account for input errors,
- lack of distinction between relative positions,
- difficulty with repeated actions, and
- difficulty processing different orderings of the same actions.

He proposed an alternative architecture, where the network takes the temporal input in serial, whilst modifying its internal state. This internal state is implemented by introducing cycles and delays into the network. Any network containing one or more cycles is described as being recurrent.

Such a network was initially trained to measure 8 phonetic features across time. The input layer contained two groups of units: input units, providing data from the current time frame; and state units, providing compressed data from all previous time frames. A hidden layer connects fully to all units from both groups in the input layer. The output layer connects fully to the hidden layer. The state units are connected to the output units and to themselves.

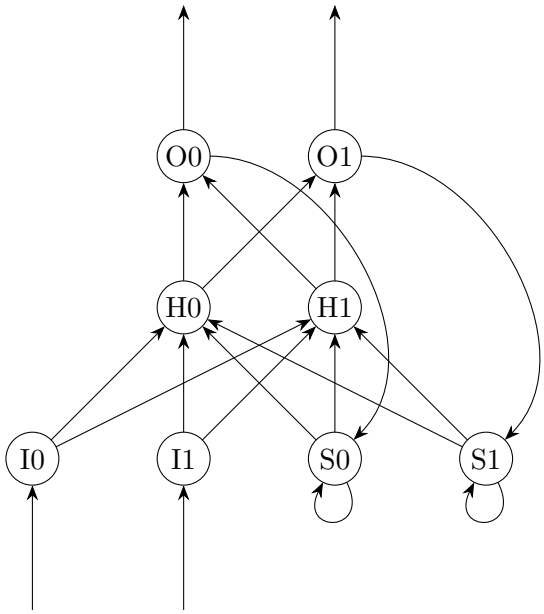


Figure 1.4: Example of Jordan's recurrent neural network.

During training, the network was exposed to one utterance at a time, and trained to output the corresponding measures for each time frame. For some combinations phonemes and features, any value is acceptable, these were marked as "don't-care", and did not affect the error. Due to the recurrent nature of the network, weight values were only updated every 4 time steps. The network learnt to process utterances and produce feature graphs that could be used to identify the correct phonemes.

Elman, 1990, expanded Jordan's critiques of buffer-based techniques by noting that there was no evidence of any biological equivalent. He proposed that the state units should correspond to the hidden layer instead. For each unit in the hidden layer, there is a corresponding state unit, which holds the previous value of the hidden unit. Units in the hidden layer is connected to the corresponding state unit, and fully to the previous layer.

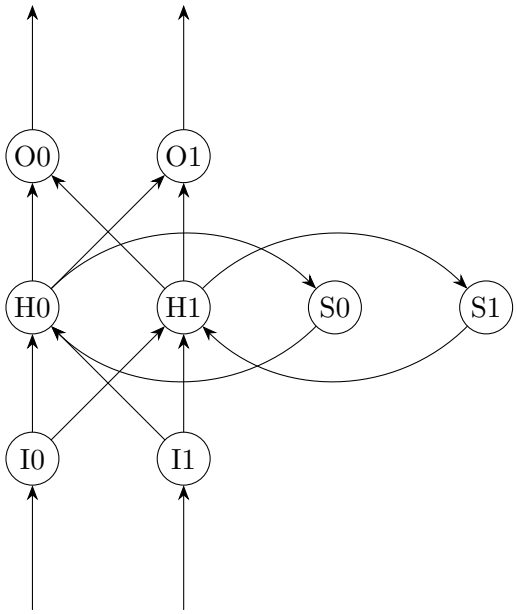


Figure 1.5: Example of Elman's recurrent neural network.

The network structure was initially used to predict sequential bit patterns. In one such experiment, a number of random sentences were generated using a small lexicon, and were presented to a network one character at a time without breaks. Each character was presented to the network as a 5-bit number, via 5 input units. The network processes this input using 20 hidden units each with a respective state unit, and outputs a prediction of the next letter via 5 output units.

Once trained, the network struggled to predict the first letter of each randomly selected word, but was able to accurately predict the letters that followed.

The recurrent neural networks previously described performed well for problems with short time delays, but failed to perform tasks involving more than 10 discrete time steps. The reason for this is the vanishing/exploding gradient problem. Because errors backpropagate through the recurrent connections across multiple time steps, the errors either tend towards zero or infinity.

A solution to this problem was proposed by Hochreiter and Schmidhuber, 1997, where the error term of the memory unit could be guaranteed to be a fixed constant. This was achieved by using multiplicative gates to control the input and output of the memory cell, resulting in a "constant error carousel".

Each cell took a weighted sum of the inputs and own previous state; and applied a nonlinear function g to obtain the net input signal, which connects to a single, self-connected state unit. A nonlinear function h is applied to the value of the state unit to obtain the output signal. Gate

units were controlled by taking a weighted sum of the layers inputs and outputs, applying a nonlinear function, and multiplying by the relevant signal. These gates prevented the network from perturbing its state, and prevents the cell state from perturbing the reset of the network, allowing it to learn long-lag-time tasks, This new neural architecture was termed "Long Short-Term Memory" (LSTM).

Although LSTM networks could learn long-lag-time tasks, they could not learn certain, very long inputs that contain multiple sub-sequences. This was because continual input streams would cause the cell state to grow without bound, even in cases where the inputs suggest that the state should be occasionally reset. This problem was solved by Gers, Schmidhuber, and Cummins, 1999, by introducing a third gate to the LSTM, which controlled the cell's internal recurrent connection.

It should be noted that an LSTM cell can "learn to never forget", hence recovering the previous model.

A further expansion of the model by Gers and Schmidhuber, 2000, added "peephole" connections between the internal state unit and the gates, allowing the gates to access the state, even when the output gate was closed. Additionally, the output function, h, was removed, as there was no empirical evidence to suggest it was required.

These peephole connections allowed the model to perform count and timing tasks, such as producing nonlinear, precisely timed spikes. A full diagram is given in Figure 1.6

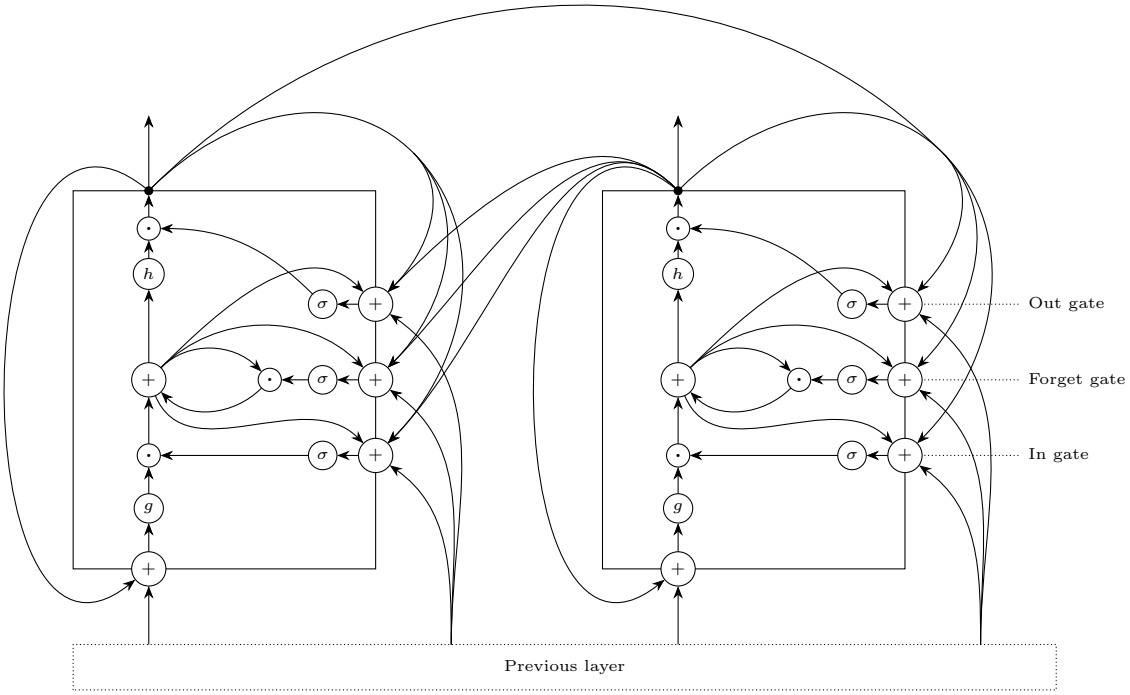


Figure 1.6: Visualisation of two LSTM units, where σ, g, and h are function units; $+$ is a weighted summation unit; and $\cdot$ is multiply unit.

Chapter 2

Introduction to Python and TensorFlow

Python is a general-purpose programming language designed by Guido van Rossum, with an emphasis on readability and reusability (Rossum, 1996). It comes with an extensive standard library and is one of the most popular programming languages.

There are multiple options for interacting with Python, these include:

- typing commands into an interpreter,
- writing files and running them with an interpreter,
- using an online service such as Google Colab.

A small snippet of trivial python code is given below to display the syntax.

```
# simple loop (i = 0, 1, ..., 4)
for i in range (3+2):
    print (i)

# function definition
# returns (n!, sum_{i=1}^{n} i!)
def fact_and_sumf (n):
    # documentation string
    """Returns n! and sum(n!)"""
    fact = 1
    sumf = 0
    for i in range (1, n+1):
        fact *= i
        sumf += fact
    return fact, sumf

```

```
print (fact_and_sumf (4))
# (24, 33)

a = [2**i for i in [3,2,4]]
print (a)
# [8, 4, 16]

print (sum (a))
# 28
```

Neural Networks in Python

Before using any neural network packages, a few small examples networks were produced in python. The `numpy` package was used to perform matrix operations, and the `matplotlib.pyplot` package was used for plotting, as these features were non-trivial. For some examples, the `keras` module from the `tensorflow` package was also used to access specific datasets, but their broader purpose will be explored in the next section.

Single Layer Boston Housing Data

The first example network consisted of 3 input neurons connected to a single output neuron, which is given by the equation

$$y = \tanh\left(b + \sum_{i=1}^{3} w_i x_i\right).$$

The bias term was implemented by adding a forth input node with a constant value of one, giving

$$y = \tanh\left(\mathbf{w} \cdot \mathbf{x}\right),$$

where $w_4 = b$, and $x_4 = 1$.

The network was trained using the Boston housing dataset from `keras`, which provided a number of attributes about houses from late 1970's Boston suburbs. The network took in normalised data from three of these attributes (number of rooms, highway accessibility index, percentage of lower status population), and used them to predict the value of the house.

Training was performed using backpropagation, defined by the equation

$$\Delta w_i = \eta \frac{\partial e}{\partial w_i},$$
$$d = y - y_t,$$
$$e = \frac{1}{2} d^2,$$

where e is the network error, y is the network prediction, and y_t is the target value. By definition,

$$y = \tanh(\text{net}),$$
$$\text{net} = \sum w_i x_i.$$

By chain rule,

$$\frac{\partial e}{\partial w_i} = \frac{\partial e}{\partial y} \cdot \frac{\partial y}{\partial \text{net}} \cdot \frac{\partial \text{net}}{\partial w_i}.$$
$$\frac{\partial net}{\partial w_i} = x_i,$$
$$\frac{\partial y}{\partial net} = 1 - \tanh^2(net) = 1 - y^2,$$
$$\frac{\partial e}{\partial y} = \frac{1}{2} \frac{\partial (y - y_t)^2}{\partial y} = y - y_t = d,$$
$$\therefore \frac{\partial e}{\partial w_i} = x_i \cdot (1 - y^2) \cdot d.$$

Δw_i can be written in vector notation, giving

$$\Delta \mathbf{w} = \eta \cdot \mathbf{x} \cdot (1 - y^2) \cdot d.$$

Weights were updated after each sample using

$$\mathbf{w}' = \mathbf{w} - \Delta \mathbf{w},$$

where $\mathbf{w}'$ is the new set of weights.

The network was initialised using random weights, and was trained using the full dataset.

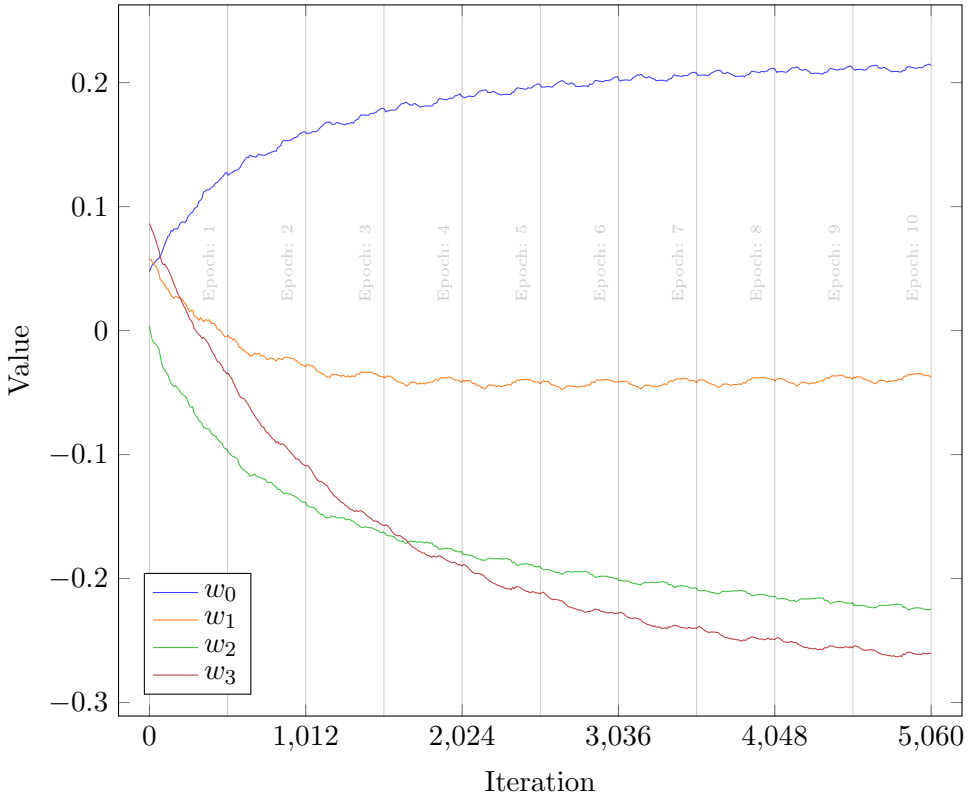


Figure 2.1: Graph of network weights against iteration number.

After 10 epochs, the mean square error had reduced from 0.1637 to 0.0507. Note that the weight value lines appear to be jagged, this was a side effect of updating the value after each individual presentation; a problem which can be mitigated by batching $\Delta\mathbf{w}$ terms from multiple presentations.

Multilayer Boston Housing Data

The same task was repeated using the full set of attributes. To accommodate this, a larger network with 13 input neurons, 1 output neuron, and n hidden neurons; which was expressed by the equation

$$h_i = \tanh\left(b_i + \sum_{j=0}^{13} w_{i,j}x_j\right),$$
$$y = b + \sum_{i=1}^{n} w_i h_i.$$

Similar to the single layer example, a constant neuron was added to the input and hidden layer to implement the bias, giving

$$\begin{aligned}\mathbf{h} &= \tanh(W\mathbf{x}),\\ \mathbf{h}' &= \begin{pmatrix}\mathbf{h}\\ 1\end{pmatrix},\\ y &= \mathbf{w}\cdot\mathbf{h}',\end{aligned}$$

where tanh acts component-wise on the input.

All of the inputs were batched together into a single matrix X, where each column was a data point, giving

$$\begin{aligned}\Phi &= \tanh(W\cdot X),\\ \Psi &= \begin{pmatrix}\Phi\\ \mathbf{1}\end{pmatrix},\\ \mathbf{y} &= \mathbf{w}\cdot\Psi,\end{aligned}$$

where $\mathbf{y}$ is a row vector of results. The error gradients for the output neurons were given by

$$\begin{aligned}\mathbf{d} &= \mathbf{y}-\mathbf{y}_t,\\ e &= \frac{1}{2}|\mathbf{d}|^2,\\ \mathbf{g}_O &= \frac{\partial e}{\partial \mathbf{w}} = \mathbf{d}\cdot\Psi^T;\end{aligned}$$

and for the hidden neurons by

$$\begin{aligned}D &= \hat{\mathbf{w}}^T\cdot\mathbf{d},\\ G_H &= ((1-\Phi\odot\Phi)\odot D)\cdot X^T,\end{aligned}$$

where $\odot$ is the component-wise product, and $\hat{\mathbf{w}}$ is the weight vector without the bias term. Note that $\hat{\mathbf{w}}^T$ and $\mathbf{e}$ are column and row vectors respectively, and that their product is a matrix. See Appendix A for full derivation.

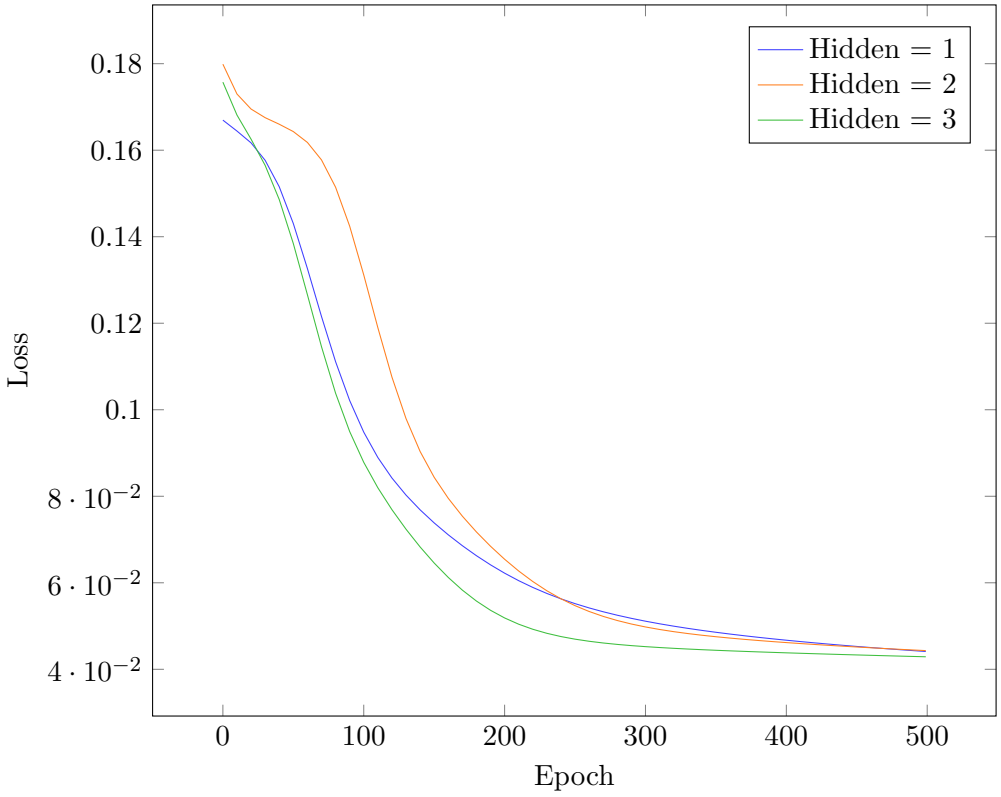


Figure 2.2: Graph of network error against iteration number for various numbers of hidden neurons, using the Boston housing data.

The network was trained multiple times with varying numbers of hidden neurons. The same random seed value was used for all trials. In each case, the network successfully reduced it's error to roughly the same level: for 1 neuron, from 0.1670 to 0.0380; for 2 neurons, from 0.1799 to 0.0390; and for 3 neurons, from 0.1757 to 0.0396. The difference between the results was negligible, which suggested that one hidden neuron was sufficient for learning the data.

When compared with the single layer network from the previous section, the single neuron, multilayer network performed better (0.0507 vs 0.0380) for two reasons:

1. it had access to all 13 attributes, instead of the 3 attributes that had been manually selected; and

2. the additional layer enabled it to apply a linear transformation to the activation.

Additionally, the use of batched inputs resulted in a smoother descent, as the total gradient is descended; instead of multiple, often opposing gradients.

Logical XOR

Using the same, multilayer network architecture, a network was trained to perform the logical exclusive-or (XOR) operation. The input matrix X contained all four combinations of binary inputs, and the target outputs in $\mathbf{y}_t$.

$$X = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix},$$
$$\mathbf{y}_t = \begin{pmatrix} 0 & 1 & 1 & 0 \end{pmatrix}.$$

Given that the problem is mathematically well defined, a successfully trained network should reduce the error to near zero. Training the network for varying numbers of hidden neurons gave the following results.

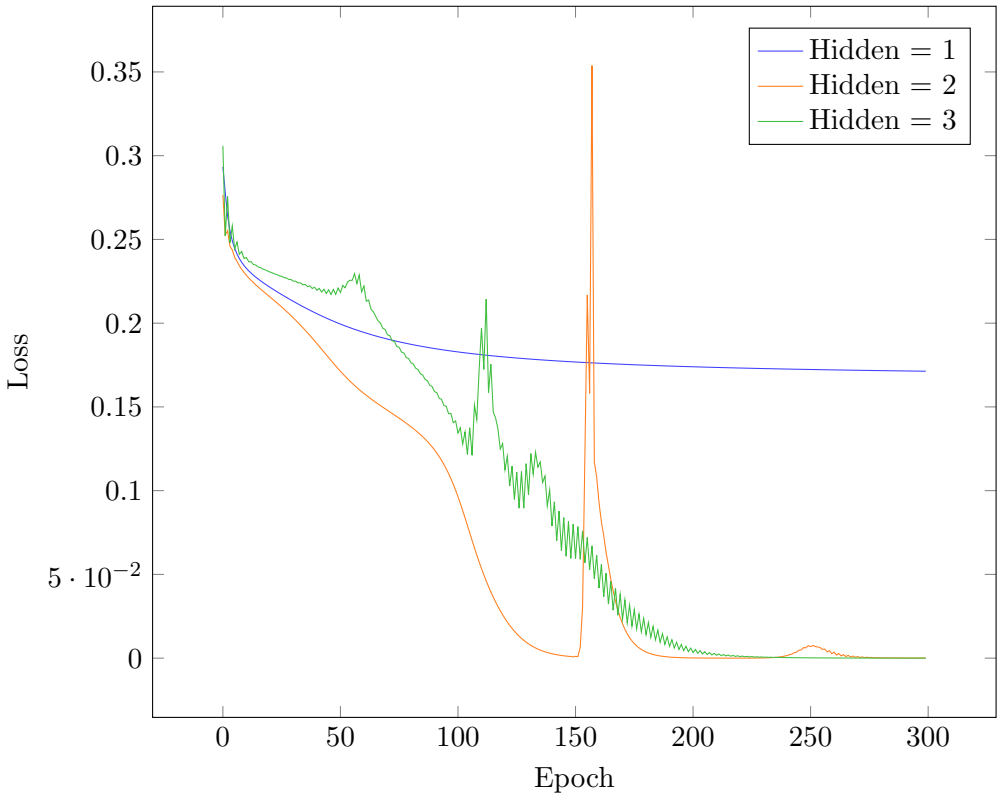


Figure 2.3: Graph of network error against iteration number for various numbers of hidden neurons, using the XOR data.

The final mean square errors for one, two, and three hidden neurons were 0.1713, 2.04×10^{-5}, and 9.65×10^{-6} respectively. These results showed that a single, nonlinear neuron is not sufficient

for learning the XOR problem, as proven by Minsky and Papert, 1969, and is only capable of solving three of the four inputs at a time. Two nonlinear neurons is sufficient for solving the problem.

It is important to note that the learning rates were much more susceptible to the initial weights than that of the Boston housing data. Under certain initial conditions, the network displayed long periods of negligible change before significant learning occurred, the longest of which that had been observed lasted over 800 iterations. Error spiking was observed across the majority of initial conditions for both two and three hidden neurons.

Using TensorFlow and Keras

Creating efficient neural networks by hand was difficult, repetitive, and prone to mistakes; and making simple modifications, such as changing the activation function, could prove tricky for larger networks. Thankfully, python packages that automate large portions of the process are available, namely TensorFlow (Abadi et al., 2016).

Once installed, TensorFlow can be imported into python. TensorFlow contains a module called Keras, which provides a number of objects and constructors that make the process much simpler. The XOR network, for instance, can be constructed with a few lines of code.

```
1  import tensorflow as tf
2  layers = tf.keras.layers
3  # Create a model
4  model = tf.keras.models.Sequential()
5  # Add a dense hidden layer with the built in tanh activation function
6  model.add(layers.Dense(3, input_dim=2, activation='tanh')
7  # Add the output layer with linear activation
8  model.add(layers.Dense(1))
9  # Finalise the model and specify the loss function
10 model.compile(loss='mean_squared_error')
```

TensorFlow provides multiple ways of constructing models, and a wide variety of options to configure layers, models, and optimisers, as well as custom definitions.

Once constructed, the model can be trained using the `fit` method, which provides a history of network properties from each epoch; and used to predict inputs using the `predict` method. Models can be saved to a file using the `save` method, which includes the network structure and weights, and loaded using `tensorflow.keras.models.load_model`.

TensorFlow also supports multithreading and GPU acceleration, making it especially suitable for large networks and training sets.

Optimisers and the Boston Housing Data

The Boston housing example was repeated using TensorFlow, with two hidden layers, each with five neurons using the tanh activation function, and a linear output layer, to see if adding additional layers and neurons would improve the results. The data was also split into two groups, training and validation, the latter of which was not used for training the network. For a well-fitting network, the loss values for both groups should be similar, and the ratio between them provided a rough measure of over-fitting.

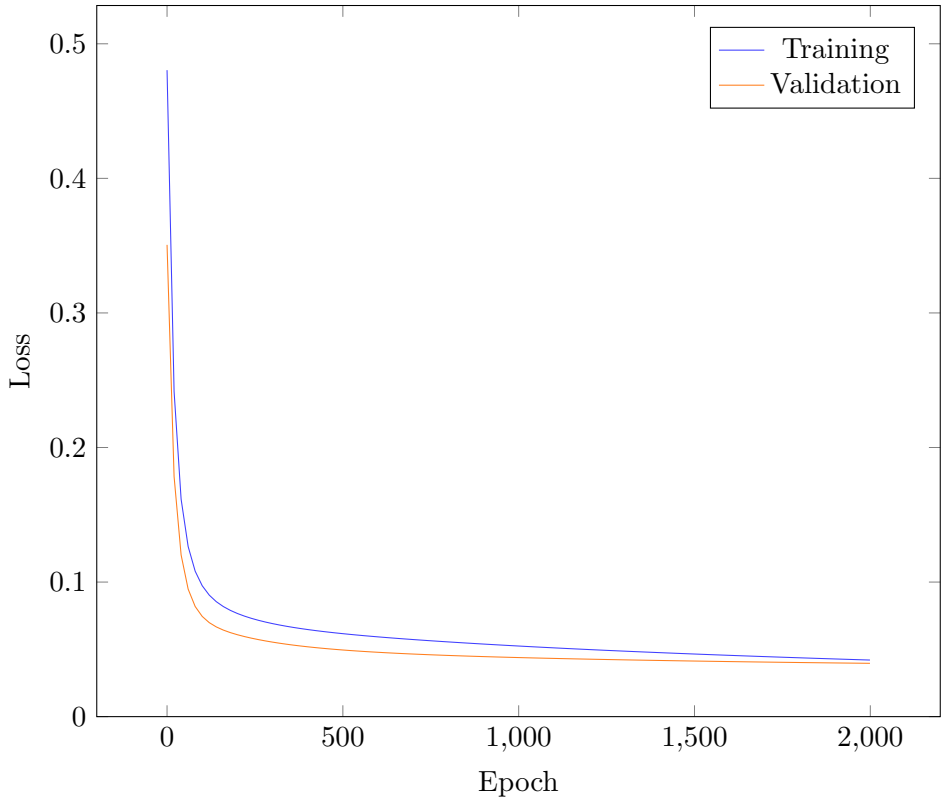


Figure 2.4: Graph of training and validation loss against iteration number for the Boston housing data using SGD.

The network was trained over 2000 epochs using the stochastic gradient descent optimiser,

obtaining an final training loss of 0.0421, and validation loss of 0.0396. The ratio between the loss values is ~ 0.94, this value is close to 1, suggesting that the network was not over-fitting.

The network was trained again using the "Adam" optimiser (Kingma and Ba, 2014), which uses estimations of both first and second-order moments, and consistently outperforms the stochastic gradient descent method.

With the Adam optimiser, the final training and validation loss values were 0.0147 and 0.0443 respectively. Although the loss value is smaller than with stochastic gradient descent, the loss ratio was ~ 3.01, which suggests that the network was over-fitting. This was further evidenced the loss graph, which show that beyond a certain point, the training loss and validation loss diverge, with the validation loss increasing. As such, the network is expected to be less generalised.

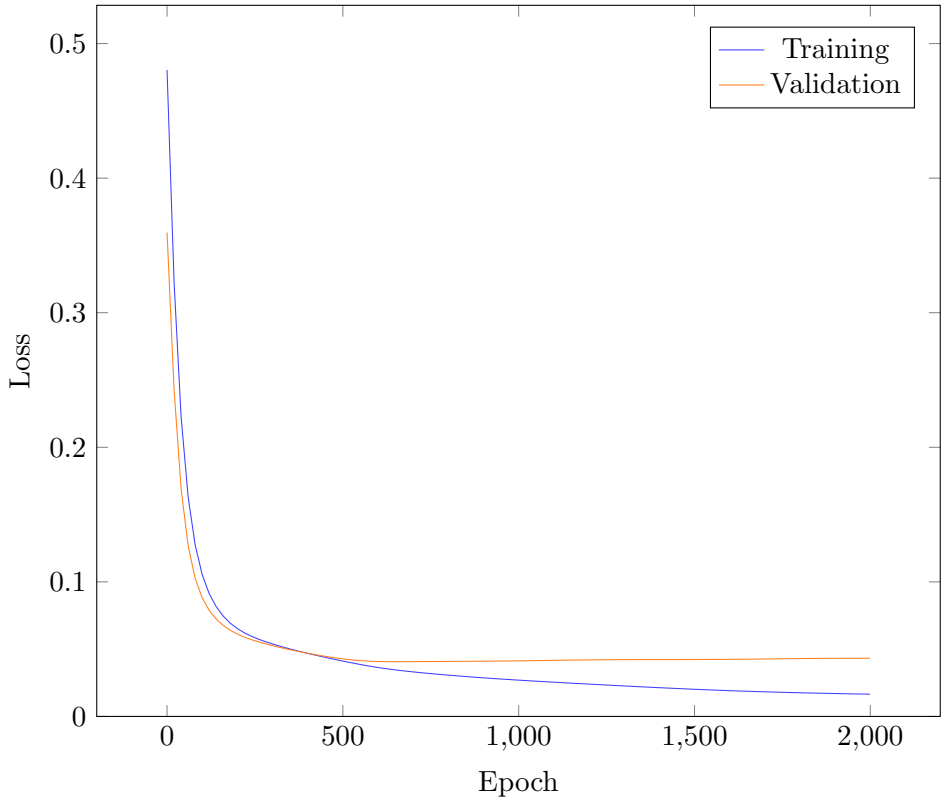


Figure 2.5: Graph of training and validation loss against iteration number for the Boston housing data using Adam.

When comparing the two optimisers, Adam trains the network significantly faster than SGD. Given a sufficient number of epochs, stochastic gradient descent will over-fit the data, just as it did with Adam.

TensorFlow provided eight optimisers at the time writing:

1. SGD, stochastic gradient descent, uses a first-order error approximation;
2. Adam, uses a first and second-order error approximation (Kingma and Ba, 2014);
3. AdaMax, Adam variant based on the infinity norm (Kingma and Ba, 2014);
4. AdaGrad, uses parameter-specific learning rates based on update frequency (Duchi, Hazan, and Singer, 2011);
5. AdaDelta, stochastic gradient descent with adaptive learning rates (Zeiler, 2012);
6. RMSProp, uses a root-mean-squared approach to adjust the learning rate (G. Hinton, Srivastava, and Swersky, 2014).
7. NAdam, Adam variant using Nesterov momentum (Dozat, 2016).
8. FTRL, follow the regularized leader, online learning algorithm (McMahan et al., 2013).

The optimal choice of optimiser varies with the network and data set size. Adaptive rate optimisers, such as Adagrad, perform best with sparse input data; but Adam, Adamax, NAdam, and RMSProp are typically the best options. See next page for figure.

The overhead of each optimiser is minimal, and is unlikely to affect the time taken to train the network.

The choice of optimiser is provided when compiling the model, via the `optimizer` argument. If no optimiser if provided, the default option, which was RMSProp, will be used. If a string is provided, TensorFlow will use the corresponding optimiser with default parameters. If an optimiser instance is provided, that instance will be used.

One may also define their own optimiser class.

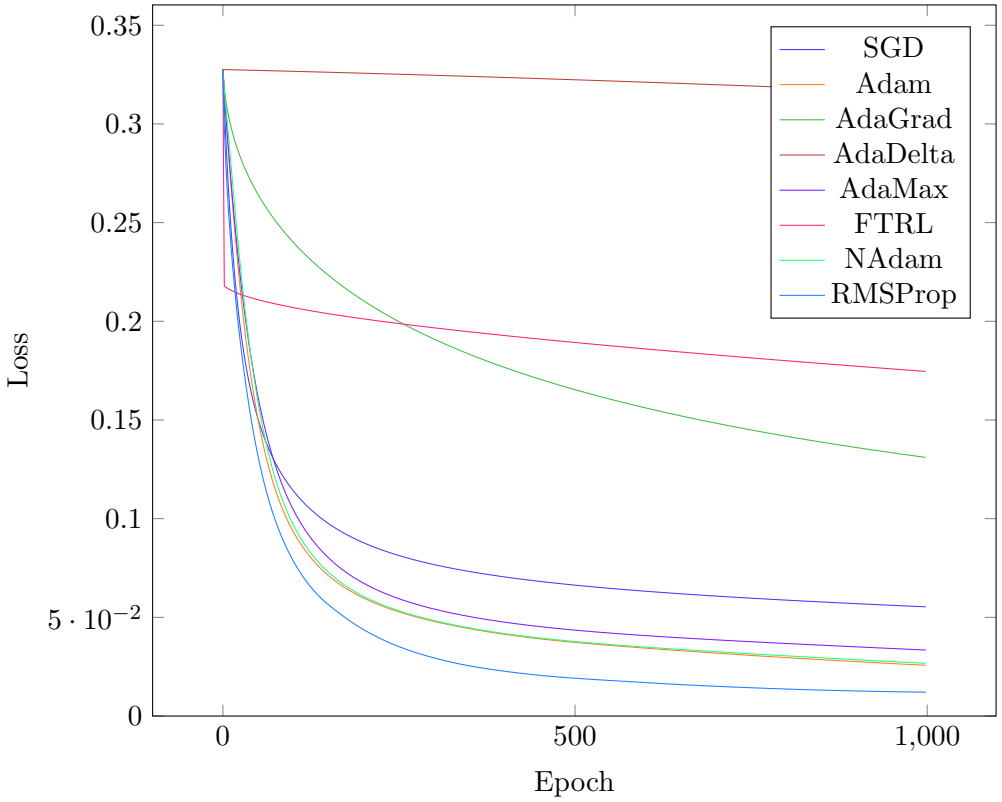


Figure 2.6: Graph of training loss against time for the Boston housing data using various optimisers.

Activation Functions

So far, only tanh and linear activation functions have been considered. TensorFlow provides a total of eleven activation functions:

- `linear`, no activation function applied,

$$\text{linear}(x) = x;$$

- `relu`, rectified linear unit,

$$\text{relu}(x) = \begin{cases} m, & x > m \\ \alpha x, & x < 0 \\ x, & \text{otherwise} \end{cases},$$

  where $\alpha = 0$ and $m = \infty$ by default;

- `exponential`,

$$\text{exponential}(x) = e^x;$$

- `elu`, exponential linear unit,

$$\text{elu}(x) = \begin{cases} \alpha(e^x - 1) & x < 0 \\ x, & \text{otherwise} \end{cases},$$

  where $\alpha = 1$ by default;

- `selu`, special case of elu with an additional scaling factor and fixed parameters;

- `sigmoid`,

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

- `hard_sigmoid`, approximation of sigmoid using three linear segments,

$$\text{hard-sigmoid}(x) = \begin{cases} 0, & x < -2.5; \\ 1, & x > 2.5; \\ 0.2x + 0.5, & \text{otherwise}; \end{cases}$$

- `tanh`,

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

- `softsign`, smoothed sign function,

$$\text{softsign}(x) = \frac{x}{|x| + 1};$$

- `softplus`, smoothed $\max(x, 0)$ function,

$$\text{softplus}(x) = \ln(e^x + 1);$$

- `softmax`, normalised exponentials,

$$y_i = \frac{e^{x_i}}{\sum_j e^{x_j}},$$

  where x_i is the net value into softmax neuron i.

With the exception of linear, all of these functions are discontinuous and/or bounded. These properties are useful for activation functions as they allow groups of neurons to approximate heavyside functions, which is necessary for decision making. Note that the tanh and sigmoid functions are essentially equivalent, as

$$\tanh(x) = 2\text{sigmoid}(2x) - 1,$$

and that such a transformation is possible on any, non-output layer.

The softmax function is typically used on the output layer of classification models, with the results representing a probability distribution.

One may also use a custom activation function by using the TensorFlow math functions.

Layer Types and Character Recognition

TensorFlow provides a large number of layer types that can easily be added to a model, as well as tools for defining custom layers. The most commonly use ones are:

- AveragePooling,
- MaxPooling,
- and Convolution, which are available in 1D, 2D, and 3D variants;
- Dense, each neuron connects to all neurons from the previous layer;
- Dropout, one to one mapping of the previous layer, setting a random selection of a fix proportion, to zero during training;
- Flatten, takes the previous layer and makes it into a 1D array of neurons;
- SimpleRNN, fully connected recurrent neural network layer;
- GRU, layer of gated recurrent units;

- LSTM, layer of long short-term memory units.

These layers can be roughly categorised into three groups: spacial layers, which use neuron locality to encode position, namely pooling and convolutional layers; simple layers, which act irrespective of input shape, such as dense, dropout, and flatten; and recurent layers, which have recurrent connections, such as SimpleRNN, GRU, and LSTM. There is also ConvLSTM2D layer, which has convolutional connections on both input and recurrent transformations, which may be classed as spacial and recurrent.

It should be noted that the N dimensional pooling and convolutional layers expect $N + 1$ dimensional tensors, where the last dimension is the channel of the input. For example: to use $w \times h$ greyscale images as the network input, the inputs must be reshaped into a $w \times h \times 1$ tensor.

To test the spacial layers, an image classification problem was considered. The MNIST dataset provides 70000, 28×28 images of handwritten numbers, and a set of corresponding labels.

The dataset was split into training and validation sets of 60000 and 10000 samples respectively.

For all of the following networks, the `sparse_categorical_crossentropy` loss function was used, which uses the data label as the index of the neuron that should be most active.

First, a network using: a flatten layer; three dense relu layers of 128, 64, and 32 neurons respectively; and a dense softmax output; was considered. From preliminary testing, the relu function was the most optimal choice for the hidden layers.

With a total of 111,146 parameters, the network was trained for 350 epochs. The final training and validation loss values were 0.0264 and 0.0956 respectively. The trained network correctly identified 58674/60000 training samples, and 8923/10000 validation samples, giving an error of 2.21% and 10.77% respectively. With a loss ratio of ~ 3.6, the network is very likely to be over-fitting, which is further supported by difference in error rates.

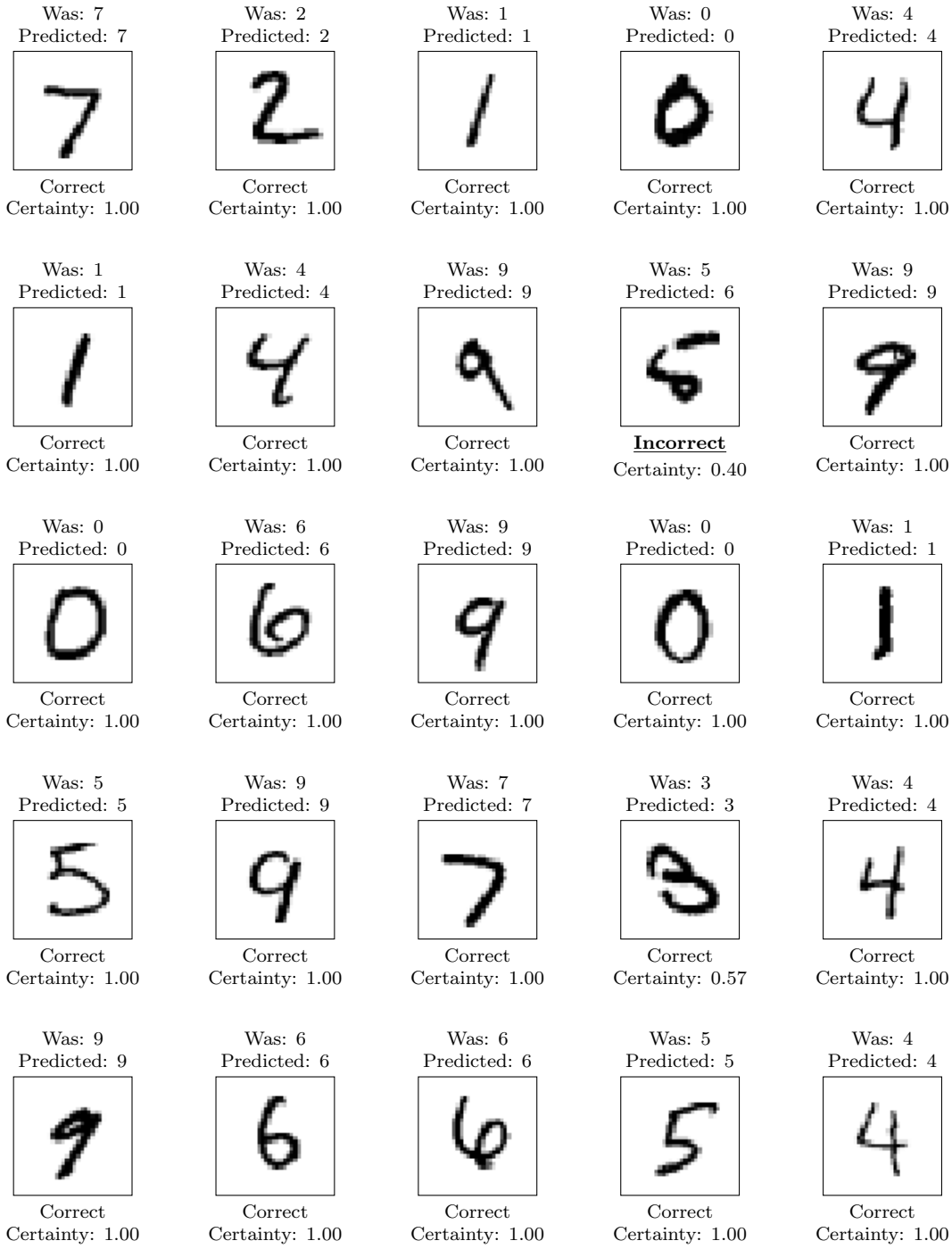


Figure 2.7: Hand written numbers from the MNIST validation set, with the prediction and certainty values, for the dense, non-dropout network.

In order to prevent over-fitting, dropout layers were added after each relu layer. By setting random neurons to 0, the dropout layers prevent neurons from forming codependence (G. E. Hinton et al., 2012). All three dropout layers used a rate of 0.2, meaning that a fifth of the neurons were set to zero during training.

After training the new network for 350 epochs, the training and validation loss values were 0.1055 and 0.0941 respectively. It should be noted that the training loss is based of the network with dropout enabled, whereas the validation loss is with dropout disabled. The trained network correctly identified 57157/60000 training samples, and 9024/10000 validation samples, with dropout disabled in both cases, giving an error of 4.74% and 9.76% respectively.

Although the new network has clearly underperformed on the training data in comparison to the previous model, the amount of over-fitting has been reduced, and the network performance on the training data is much more representative of the actual network performance. The difference in validation performance measures suggest an improvement, as both loss and error values are reduced by the new model.

As mentioned in Section 1.3.1, densely connected networks struggle to correctly identify shifted inputs, which makes them less suited to image recognition tasks. As such a network using: three, 7×7 convolutional relu layers with 4, 6, and 10 filters respectively; a flatten layer; and a dense softmax output; was also considered. Due to the number of neurons in the network and memory constraints, full-batch processing was not possible; for all of the following the networks a batch size of 600 was used.

With a total of 14,342 parameters, the network was trained 20 epochs. The final training and validation loss values were 0.0557 and 0.0566 respectively. The trained network correctly identified 56166/60000 training samples, and 9313/10000 validation samples, giving an error of 6.39% and 6.87% respectively.

Comparing the convolutional network against both dense networks, the convolutional network may have reduced performance on the training samples, but is significantly better on the validation samples. Further more, with a loss ratio of ~ 1.02, the network does not appear to be over-fitting.

As with the dense network, dropout layers were then added after each convolutional layer. After training the new network for 20 epochs, the training and validation loss values were 0.0593 and 0.0408 respectively. The trained network correctly identified 57493/60000 training samples and 9499/10000 validation samples, giving an error of 4.18% and 5.01% respectively.

Even though over-fitting was not a problem in the non-dropout convolutional network, the use of dropout layers has improved the performance of the network

	Dense		Convolutional	
Dropout?	No	Yes	No	Yes
Training Loss	0.0264	0.1055	0.0557	0.0593
Training Error	2.21%	4.74%	6.39%	4.18%
Validation Loss	0.0956	0.0941	0.0566	0.0408
Validation Error	10.77%	9.76%	6.87%	5.01%

Figure 2.8: Table summarising the network loss and error values for all of the character recognition networks.

Chapter 3

Reinforcement Learning

The neural network structures previously discussed for the Boston Housing data are essentially curve fitting models, with continuous inputs and output. The character recognition networks for the MNIST dataset can also be though of as an abstract curve fitting model with multiple outputs, where the result is a probability value.

In both cases, training is performed on a set of inputs with known target outputs.

Not all problems can be constructed in this manner as some problems require discrete actions to be taken. Similarly, problems may contain reward systems. Such systems may have delayed rewards, were one action provides an immediate reward, but prevents reaching a state of higher reward.

Consider the "Chain" problem from Strens, 2000:

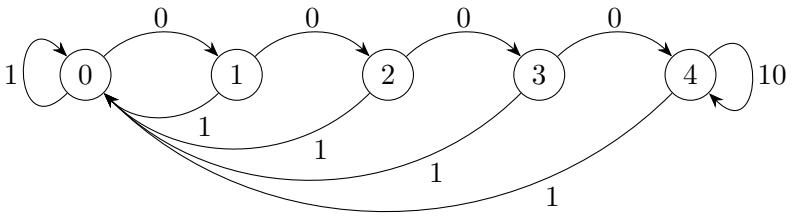


Figure 3.1: Five-state chain problem. Arrows from the top of the states denote action 0, and from the bottom denote action 1. The labels on each arrow denote the reward associated with that action. The system is initially at state 0.

The optimal, long term strategy for this system is to always perform action 0, which will cause the system to reach state 4 and obtain a reward of 10 on each step. A greedy algorithm will discover the reward from always performing action 1, which will cause the system to stay at

state 0 and obtain a reward of 1 on each step. The greedy algorithm is only optimal for four steps or less.

The chain problem was recreated in python with the following code:

```
class Chain:
    def __init__ (self):
        self.state = 0
        self.dings = 0
    def reset (self):
        self.state = 0
        self.dings = 0
        return 0
    def step (self, action):
        if action == 1:
            self.state = 0
            return 1
        if self.state == 4:
            self.dings += 1
            return 10
        self.state += 1
        return 0
```

Note that `dings` is a performance metric used to count the number of times the agent performed action 1 from state 4, and is not made available to the network at any point.

Q-Learning

One way to find a solution to this problem without neural networks is "Q-Learning". Proposed by Watkins, 1989, Q-Learning is an algorithm for finding optimal solutions to Markovian domain problems, such as the chain.

The algorithm starts with a table, referred to as the Q-table, which has a row for each system state, and a column for each action, initialised as zero. Once trained, each element of the Q-table will represent the maximum expected reward for each action in each state.

Let $Q(s, a)$ denote the Q-table value for state s and action a.

Initially, actions are taken at random, which allows the agent to explore the network; but as training progresses, the agent gradually shifts towards making decisions based on the Q-table values.

Each time an action a is taken from state s, the reward r and new state s' are observed, and the Q-table is updated using the Bellman equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a)), \tag{3.1}$$

where α is the learning rate, and γ is the discount rate, which is needed to prevent the values from growing exponentially. Note that $\max_{a'} Q(s', a')$ is the maximum expected future reward of state s', according to the current Q-table values.

The algorithm was applied to the chain problem with learning rate $\alpha = 0.8$, and discount rate $\gamma = 0.7$, for 200 episodes, each consisting of 50 steps. During training, the probability of the agent performing a random action was given by $0.01^{t/E}$, where t is the episode number, and E in the total number of episodes.

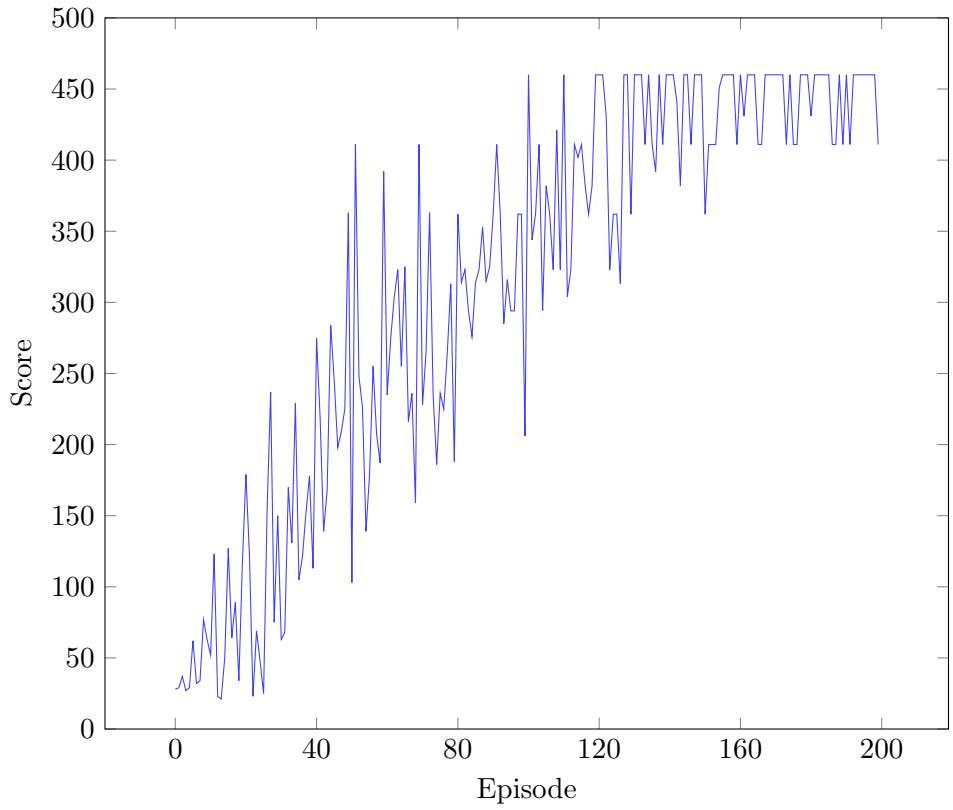


Figure 3.2: Graph of score against episode number for the Q-Learning chain problem.

Once trained, the agent was able to obtain the highest possible score of 460. The final values of the Q-table are provided below.

State	Action 0	Action 1
0	8.00	6.60
1	11.43	6.60
2	16.33	6.60
3	23.33	6.60
4	33.33	6.60

Figure 3.3: Q-table for the full trained Q-Learning chain problem, to two decimal places.

Note that all of the values in the Action 0 column are larger than the corresponding values in the Action 1 column, which denotes the larger expected future reward.

Learning Condition for Generalised Chain Problem

In order for Q-Learning to converge on the optimal policy for the chain problem, at each state, the Q-value for action 0 must be greater than the value of action 1.

Consider a generalised Q-Learning chain problem, consisting of N states ($[0, N-1]$), where: action 0 gives a reward of $R > 1$ when at state $N-1$, and 0 otherwise; and action 1 always gives a reward of 1. As before, action 1 always reverts the state to 0, and action 0 advances the state until it reaches the end of the chain.

By rearranging Equation 3.1, the true Q-value for any state is given by

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(s',a').$$

Applying the properties of the chain problem gives

$$Q(N-1,0) = R + \gamma \max_{a'} Q(N-1,a'),$$
$$\forall s,\, Q(s,1) = 1 + \gamma \max_{a'} Q(0,a').$$

Note that the Q-value for action 0 is the same constant for all states, and is dependent on the Q-value of the initial state; from this, it follows that there are only two possible local maxima

for the chain problem starting at state 0: the greedy policy, where action 1 is always taken; and the truly optimal policy, where action 0 is always taken.

Consider the Q-values: $Q(0,1)$ the greedy policy, given by

$$Q^{\mathrm{G}}(0,1) = 1 + \gamma Q^{\mathrm{G}}(0,1) = \frac{1}{1-\gamma};$$

and $Q(0,0)$ for the truly optimal policy, given by

$$Q^{\mathrm{O}}(0,0) = \sum_{t=N-1}^{\infty} \gamma^t R = \frac{\gamma^{N-1}}{1-\gamma} R.$$

In order for the truly optimal policy to be chosen, it's Q-value must be the largest, which gives the constrain

$$Q^{O}(0,0) > Q^{G}(0,1),$$
$$\gamma^{N-1} R > 1.$$

If this condition is met, all subsequent states will have a higher q-value for action 0.

In practice, most problems are not as simple to reason about; but it does provide a useful insight into how the choice of γ can affect the learning process.

Deep Q-Learning

The Q-Learning algorithm works well for problems with well defined states and actions, but cannot accommodate problems where the state is either to too large, not directly visible, or non-Markovian. To solve this problem, a neural network may be trained to take in a set of observations and predict the corresponding Q-values for each action (Lin, 1990).

To prevent the system from forgetting past experiences, a replay memory was created, which stores a limited number of experiences (Lin, 1992). Each experience contains: the initial observations, the action performed, the reward received, and the new observations.

The use of replay memory also allows training to be performed with fewer total experiences.

After each action, the experience is added to the replay memory. If the memory is full, the oldest element is removed to accommodate the newest. In python, this behaviour is handled automatically by using the `deque` class, from the inbuilt `collections` package.

Initially, a number of random trails are performed to fill the memory.

Training is then performed in a similar manner as Q-Learning. As the Q-values cannot be modified directly, a batch of random experiences from the replay memory are selected for training the network. The network target is calculated using a modified version of Equation 3.1, where $Q(s, a)$ is the network's Q-value prediction.

Deep Q-Learning was applied to the chain problem using a single-layer network with: five input neurons, one for each state; and two densely connected, ReLU output neuron. Training was performed over 40 episodes, each consisting of 50 steps, with the same parameters as before. After each step, the network trained using a random sample of 100 experiences from a replay memory with a capacity of 500.

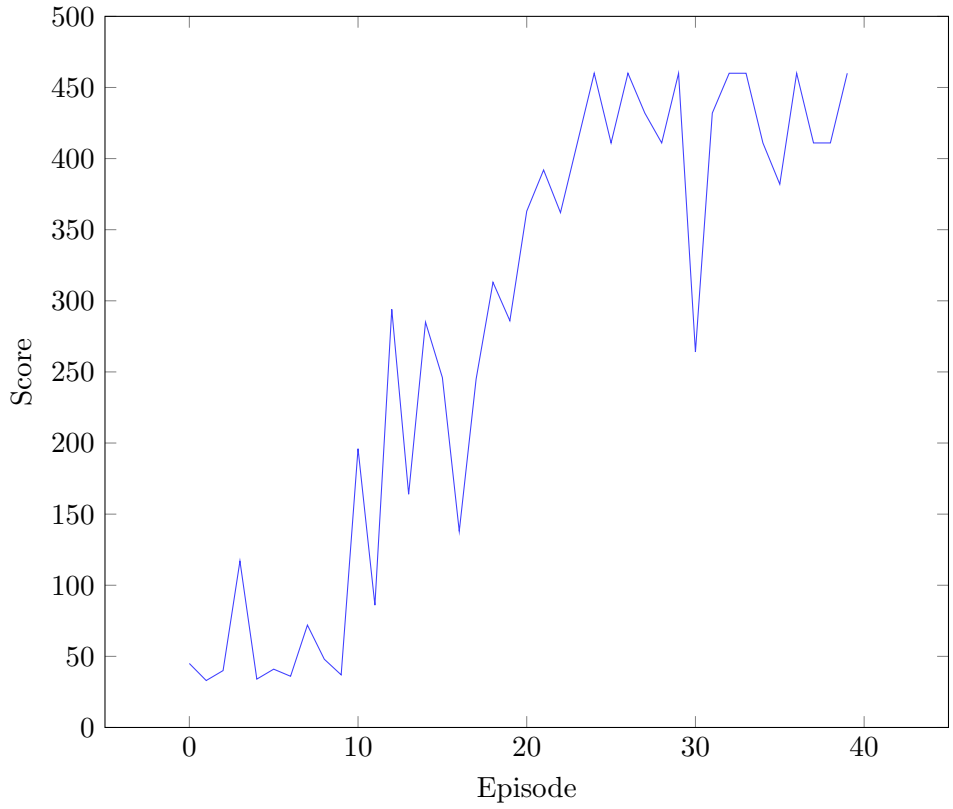


Figure 3.4: Graph of score against episode number for the Deep Q-Learning chain problem.

Prioritised Experience Replay

The replay memory provided a simple way for the network to remember and reuse past experiences. In the first instance, experiences were randomly sampled from the memory with a uniform distribution.

Schaul et al., 2015, notes that improved training performance can be obtained by prioritising some experiences over others. This is achieved by creating a weighted distribution that favours elements with higher priority values. The probability of picking the i^{th} memory is given by

$$P(i) = \frac{p_i^a}{\sum_k p_k^a},$$

where p_i is the priority value of the i^{th} memory, and a is a parameter that controls the distribution. Using $a = 0$ will make the distribution uniform.

The priority value is defined as

$$p_t = |\delta_t| + e,$$

where e is some constant to prevent zeros, and δ_t is the temporal difference,

$$\delta_t = r + \gamma \max_{a'} Q(s', a') - Q(s, a).$$

However, a problem arises from using this distribution, which is that the sample distribution no longer matches the source distribution, which in turn causes bias. To compensate, a weighting is introduced to the network training loss:

$$\left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\beta},$$

where β is a parameter to be increased during training.

This technique is known as Prioritised Experience Replay (PER).

Efficient sampling of the replay memory according to the priority values was achieved by implementing a "sum tree", which replaces the `deque` structure, see Appendix B for details.

Deep Q-Learning with PER was applied to the chain problem using the same network layout as before. Using PER, the agent could be reliably trained using fewer episodes. Training was performed over 30 episodes, each consisting of 50 steps, with the same memory capacity, batch size, and parameters as before.

Fixed Q-Targets

When training, the expected future reward from state s' is calculated using the same network as the expected future reward from state s, which is being updated each step. This causes both $Q(s', a')$ and $Q(s, a)$ to change on each step, which can cause instabilities.

To reduce the instability, a separate target network may be introduced. This gives

$$\delta_t = r + \gamma \max_{a'} Q^T(s', a') - Q^P(s, a),$$

where Q^T is the target network, and Q^P is the policy network. Both networks are initialised with the same weights, but only the policy network is trained, hence providing a fixed approximation of the future Q-values.

The weights of the target network are periodically updated to match those of the policy network, this is done every τ steps, where τ is some chosen parameter.

Deep Q-Learning with Fixed Q-Targets was applied to the chain problem using the same network layout as before. Training was reliably performed over 35 episodes, each consisting of 50 steps, with $\tau = 10$, and the same memory capacity, batch size, and parameters as before.

Double Deep Q-Networks

When dealing with larger systems, the random agent may not have fully explored the system. In such cases, there may be insufficient information about which actions should be taken during the initial stages of training. Taking the action with the maximum Q-value can lead to false positives, which can cause over predictions for frequently taken actions.

The solution proposed by Hasselt, 2010, is to use two Q-Networks, Q^A and Q^B to predict the Q-Values, which reduces overestimation within the policy. At each step, one network is randomly chosen for training, and the other is used to predict the maximum future value, giving either:

$$\delta_t^A = r + \gamma Q^B(s', a^*) - Q^A(s, a),$$
$$a^* = \operatorname*{argmax}_a Q^A(s', a);$$

or

$$\delta_t^B = r + \gamma Q^A(s', a^*) - Q^B(s, a),$$
$$a^* = \underset{a}{\text{argmax}}\ Q^B(s', a).$$

Doing this reduces the overestimation of the Q-values and improves training stability.

For the trivial case of the chain problem, applying Double Deep Q-Networks negatively impacts performance, and often prevented the network from learning the optimal policy. A possible reason for the reduced training performance is that the network does not suffer from overestimation when learning the chain problem, and so applying Double Deep Q-Networks to the problem causes underestimation, which in turn prevents the policy from progressing through the chain.

Dueling Deep Q-Network

When recalling the interpretation of the Q-value, Wang et al., 2015 noted that the Q-value represents a combination of both the state values and the state dependent action advantages, and proposed an architecture that explicitly separates the two. This might be naively implemented as $Q(s, a) = V(s) + A(s, a)$, but Wang et al. noted that this does not produce unique values for $V(s)$ and $A(s, a)$. To counteract this, the $A(s, a)$ value are offset by a property thereof, such as the maximum or mean value, the latter giving

$$Q(s, a) = V(s) + \left(A(s, a) - \underset{a'}{\text{mean}}\ A(s, a') \right). \tag{3.2}$$

Implementing this into the neural network is achieved as follows:

1. a number of layers input the observations and produce a set of state representations;
2. the state representation is inputted into two sets of layer streams, the former predicts the single state value, and the latter predicts the advantage values for each action;
3. the two streams are merged according to Equation 3.2, producing a Q-value for each action.

Dueling Deep Q-Learning was applied to the chain problem using a network with five input neurons, one for each state of the environment; which is connected to two streams: a single

densely connected ReLU for the state value, and two densely connected ReLUs for the advantage values which feed into a "lambda" layer, which calculates the offset values; and an add layer, which provides the output Q-value.

Training was reliably performed over 30 episodes, each consisting of 50 steps, with the same memory capacity, batch size, and parameters as before.

Combined

All of the previously discussed improvements can be used simultaneously, in any combination, to enhance the training performance of a policy.

A Dueling Deep Q-Network with PER and Fixed Q-Targets was applied to the chain problem. Training was reliably performed over 25 episodes, each consisting of 50 steps, with the same memory capacity, batch size, and parameters as before.

Policy Gradient

Although Q-Learning provided an efficient method to produce deterministic polices, many problems require stochastic policy, where actions are selected according to probabilities. Another issue with Q-Learning is that it uses an implicit greedy algorithm, where the action with the highest q-value is always selected; small changes in the q-value can radically change the policy.

A method for producing stochastic polices was introduced by Sutton et al., 2000, which defined probability of taking action a at state s as $\pi_\theta(s, a)$, where π is a general function approximator, such as a neural network, and θ a vector of parameters for π.

It is necessary to know the state distribution, which may be given by

$$d^\pi(s) = \frac{N(s)}{\sum_{s'} N(s')},$$

where $N(s)$ is the number of occurrences of state s. When summing across an experience replay memory, the distribution is implicit.

One of two possible objectives may be chosen, either: maximise the cumulative discounted reward from a designated initial state, s_0, or; maximise the long-term expected reward per step. In either case, two functions are defined: $\rho(\pi)$, which is the long-term performance metric; and $Q^\pi(s, a)$, which is the value of a given state-action pair.

In the case of cumulative discounted reward, the long-term performance is given by

$$\rho(\pi) = E\left\{\sum_{i=0}^{\infty} \gamma^i r_i \;\middle|\; s_0,\, \pi\right\}$$

and the state-action pair value is given by

$$Q^\pi(s,a) = E\left\{\sum_{i=0}^{\infty} \gamma^i r_{t+i} \;\middle|\; s_t = s,\, a_t = a,\, \pi\right\},$$

where γ is the discount rate, and r_i is the reward at step i.

In the case of expected reward per step, the long-term performance is given by

$$\rho(\pi) = \sum_s d^\pi(s) \sum_a \pi(s,a) R_s^a,$$

where R_s^a is the reward for taking action a at state s; and the state-action pair value is given by

$$Q^\pi(s,a) = \sum_{t=1}^{\infty} E\left\{r_t - \rho(\pi)\right\}.$$

In either case, the policy gradient is given by

$$\frac{\partial \rho}{\partial \theta} = \sum_s d^\pi(s) \sum_a \frac{\partial \pi(s,a)}{\partial \theta} Q^\pi(s,a),$$

which is used to form a discretised iterative formula. As with neural networks, a step size is included as a factor of the gradient, which controls the learning rate of the policy.

For neural networks, the policy gradient problem is often reformulated as minimising the loss function,

$$L^{\mathrm{PG}}(\theta') = E_t\left\{\log(\pi_\theta(s_t,a_t))\hat{A}_t\right\},$$

where $\hat{A}_t$ is the discounted advantage from time step t.

It should be noted that, unlike Q-Learning, policy gradient training is performed at the end of an episode, not during.

Whilst implementing policy gradients in TensorFlow is possible, it requires the use of lower level functions and classes, which are lacking the thorough resources that are available for the

higher level functions. As such, an implementation of policy gradients, or any of it's subsequent variations, was not possible in the available time.

Trust Region Policy Optimisation

The convergence of a policy gradient method is highly dependent on the chosen learning rate: if the learning rate is too small, the training process will be slow to converge; if the learning rate is too large, training will be unstable, due too high parameter variance. Further more, the optimal choice of learning rate can often change during training.

In order to address instabilities, Schulman, Levine, et al., 2015, introduced a method called Trust Region Policy Optimisation (TRPO). The method considers the optimisation problem

$$\underset{\theta'}{\text{maximize}} = E_t \left\{ r_t(\theta') \hat{A}_t \right\}, r_t(\theta') = \frac{\pi_{\theta'}(s_t, a_t)}{\pi_\theta(s_t, a_t)},$$

where θ' is the new set of policy parameters; which is subject to the constraint,

$$E_t \left\{ \text{KL} \left[\pi_\theta, \pi_{\theta'} \right] \mid s_t \right\} \leq \delta,$$

where KL is the Kullback–Leibler divergence, and δ is some value.

This constrains the amount by which the parameters can change between each iteration.

Proximal Policy Optimisation

Although trust region policy optimisation has better convergence than normal policy gradient methods, solving the optimisation constraint is computationally expensive.

An alternate method for restricting the maximum change between steps was introduced by Schulman, Wolski, et al., 2017, called Proximal Policy Optimisation (PPO). Instead of introducing a constraint, PPO clips values to a fixed interval.

$$L^{\text{CLIP}}(\theta') = E_t \left\{ \min \left(r_t(\theta') \hat{A}_t, \text{clip}(r_t(\theta'), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right\},$$

where ϵ is some value, and clip is the function given by

$$\text{clip}(x, l, u) = \begin{cases} l, & x < l; \\ u, & x > u; \\ x, & \text{otherwise}; \end{cases}$$

which limits the input x to the region $[l, u]$.

Actor-Critic

So far, two kinds of model have considered: actor-only methods, such as policy gradients, where the parameters of the model are directly estimated by simulation; and critic-only methods, such as Q-Learning, which attempt to approximate the value function of simulation.

A form of hybrid methods, known as Actor-Critic methods, were introduced by Konda and Tsitsiklis, 2000, to combine the advantages of actor and critic-only methods. In these methods, a critic model learns an approximate value function, which is then used to update the actor model policy.

Two networks are defined: the actor network, $\pi_\theta(s, a)$, with learning rate α; and the critic network, $Q_w(s, a)$, with learning rate η. The parameter updates for the actor and critic networks are given by

$$\Delta\theta = \alpha Q_w(s, a) \nabla_\theta (\log \pi_\theta(s, a))$$

and

$$\Delta w = \eta(R(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \nabla_w Q(s_t, a_t)$$

respectively.

As with Deep Q-Learning, improved performance can be achieved by splitting the critic network's Q-value calculation into separate state and advantage values (Section 3.2.4).

As example applications have not been provided throughout this section, consider reading "DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning" (Peng et al., 2017), as it provides an excellent example of how actor-critic models can be used

to solve complex problems.

Chapter 4

Conclusions

Neural networks have a wide range of applications, and TensorFlow provides an easy-to-use framework for defining and training models for image recognition and curve fitting within python, via the Keras sub module.

One major difficulty with TensorFlow is that, when attempting to search for informational resources regarding custom TensorFlow models (as would have been required for implementing policy gradients), a large number of the results were for TensorFlow 1, not TensorFlow 2, and the difference between the versions is substantial.

Had more time been available, a more thorough understanding of TensorFlow's lower level functions and class may have been possible.

A point of interest for further research maybe a comparison between TensorFlow 2 and other neural network packages, such as PyTorch, with respect to performance and ease of use.

Appendices

Appendix A

Derivation of Multi-layer Neural Network Equation

I is the number of inputs per sample, N is the number of samples, H is the number of hidden neurons.

$$
\begin{aligned}
h_i &= \tanh(s_i), \\
s_i &= b_i + \sum_{j=0}^{I} w_{i,j} x_j \\
y &= b + \sum_{i=0}^{H} w_i h_i.
\end{aligned}
$$

Vectorise:

$$
s_i = \begin{pmatrix} w_{i,1} & \cdots & w_{i,I} & b_i \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_I \\ 1 \end{pmatrix}, \qquad y = \begin{pmatrix} w_1 & \cdots & w_H & b \end{pmatrix} \cdot \begin{pmatrix} h_1 \\ \vdots \\ h_H \\ 1 \end{pmatrix}.
$$

$$
\mathbf{s} = \begin{pmatrix} s_1 \\ \vdots \\ s_H \end{pmatrix} = \begin{pmatrix} w_{1,1} & \cdots & w_{1,I} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ w_{H,1} & \cdots & w_{H,I} & b_H \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_I \\ 1 \end{pmatrix} = W \cdot \mathbf{x},
$$

$$
\mathbf{h} = \begin{pmatrix} \tanh(\mathbf{s}) \\ 1 \end{pmatrix}, \qquad y = \mathbf{w} \cdot \mathbf{h}.
$$

Batch:

$$
S = \begin{pmatrix} \mathbf{s}^{(1)} & \cdots & \mathbf{s}^{(N)} \end{pmatrix} = W \cdot \begin{pmatrix} \mathbf{x}^{(1)} & \cdots & \mathbf{x}^{(N)} \end{pmatrix},
$$

$$
\Phi = \tanh(S), \qquad \Psi = \begin{pmatrix} \Phi \\ \mathbf{1} \end{pmatrix},
$$

$$\mathbf{y} = \begin{pmatrix} y^{(1)} & \cdots & y^{(N)} \end{pmatrix} = \mathbf{w} \cdot \Psi.$$

Backpropagation:

$$\begin{aligned} d &= y - y_t, \\ e &= \frac{1}{2} d^2, \end{aligned}$$

$$\begin{aligned} \frac{\partial s_i}{\partial w_{i,j}} &= x_j, & \frac{\partial h_i}{\partial s_i} &= 1 - h_i^2, \\ \frac{\partial y}{\partial w_i} &= h_i, & \frac{\partial y}{\partial h_i} &= w_i, \\ \frac{\partial e}{\partial y} &= d. \end{aligned}$$

$$\begin{aligned} \frac{\partial e}{\partial w_i} &= \frac{\partial e}{\partial y}\frac{\partial y}{\partial w_i} = dh_i, \\ \frac{\partial e}{\partial w_{i,j}} &= \frac{\partial e}{\partial y}\frac{\partial y}{\partial h_i}\frac{\partial h_i}{\partial s_i}\frac{\partial s_i}{\partial w_{i,j}} = dw_i(1 - h_i^2)x_j. \end{aligned}$$

Batch:

$$\begin{aligned} \mathbf{d} &= \mathbf{y} - \mathbf{y}_t, \\ \frac{\partial e}{\partial w_i} &= \sum_k^N d^{(k)} h_i^{(k)} \\ &= \begin{pmatrix} d^{(1)} & \cdots & d^{(N)} \end{pmatrix} \cdot \begin{pmatrix} h_i^{(1)} & \cdots & h_i^{(N)} \end{pmatrix} \\ &= \mathbf{d} \cdot \mathbf{h}_i. \\ \frac{\partial e}{\partial w_{i,j}} &= \sum_k^N d^{(k)} w_i \left(1 - \left(h_i^{(k)}\right)^2\right) x_j^{(k)} \\ &= (\mathbf{1} - \mathbf{h}_i \odot \mathbf{h}_i) \odot (w_i \mathbf{d}) \cdot \mathbf{x}_j. \end{aligned}$$

Vectorise:

$$\begin{aligned} \frac{\partial e}{\partial \mathbf{w}} &= \mathbf{d} \cdot \begin{pmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_{H+1} \end{pmatrix} = \mathbf{d} \cdot \Psi^T. \\ \frac{\partial e}{\partial W} &= \begin{pmatrix} \mathbf{a}_1 \\ \vdots \\ \mathbf{a}_{I+1} \end{pmatrix} = \left(1 - \begin{pmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_H \end{pmatrix} \odot \begin{pmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_H \end{pmatrix}\right) \odot \begin{pmatrix} w_1 \mathbf{d} \\ \vdots \\ w_H \mathbf{d} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x}_1^T & \cdots & \mathbf{x}_{I+1}^T \end{pmatrix} \\ &= (1 - \Phi \odot \Phi) \odot (\hat{\mathbf{w}} \cdot \mathbf{d}) \cdot X^T, \end{aligned}$$

Appendix B

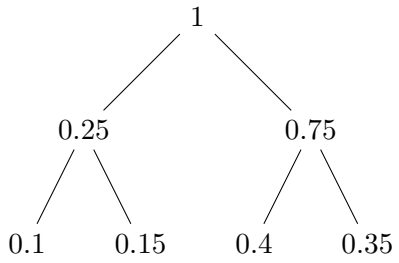
Sum Tree

A sum tree is a binary search tree where each non-leaf node is the sum of it's child nodes. This was useful in the case of Prioritised Experience Replay as it reduces the search time from order $O(n)$ to order $O(\log n)$. Another useful property of the sum tree is that the root node is the sum of all the leaf nodes, which is required to normalise the priority values into probabilities.

Randomly sampling experiences with the correct distribution is achieved by first sampling some value r from uniform distribution, then navigating the tree until a leaf node is reached, the index of the node corresponds to the index of experience to be sampled.

Navigating the tree starts at the root node, the value r is compared against the left child node: if r is less than the child node, then that node becomes the current node; if not, the value of the child node is subtracted from r and the right child node becomes the current node. If no left child node is present, the search is complete.

Example:
Consider the following tree:



Suppose the value $r = 0.45$ is randomly generated from a uniform distribution: r is greater than the left child node, 0.25, so subtract the left node value and move to the right child node, $r \leftarrow r - 0.25 = 0.2$; r is less than the left child node, 0.4, so move to the left child node; no left child node is present, search complete; search ended on the 3$^{\text{rd}}$ leaf node, select the 3$^{\text{rd}}$ experience from the replay memory.

Appendix C

Python Script for Dueling Double Deep Q-Network

Full source code, including unused files, can be found at `https://github.com/RanfordS/2020-Masters-Project`.

```
# Imports
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import random
from collections import deque

# Environment
class Chain:

    def __init__ (self):
        self.state = 0
        self.dings = 0

    def reset (self):
        self.state = 0
        self.dings = 0
        return 0

    def step (self, action):

        if action == 1:
            self.state = 0
            return 1

        if self.state == 4:
            self.dings += 1
            return 10

        self.state += 1
```

```
33          return 0
34
35  env = Chain ()
36
37  def state_to_array (index):
38      z = np.zeros (5)
39      z[index] = 1.0
40      return z
41
42  # Hyperparameters
43  num_episodes = 25
44  num_testepis = 1
45  num_steps = 50
46  tau = 10
47
48  size_batch = 100
49  size_memory = 500
50
51  lr = 0.8
52  gamma = 0.7
53
54  max_eps = 1.0
55  min_eps = 0.01
56
57  use_double = False
58  data_filename = "DataAllDeepQLearning.csv"
59
60  # Memory
61  class SumTree:
62
63      def __init__ (self, capacity):
64          self.capacity = capacity
65          self.index = 0
66          self.data = np.zeros (capacity, dtype=object)
67          self.tree = np.zeros (2*capacity - 1)
68          #       0
69          #   ,---'---,
70          #   1       2
71          # ,-'-,   ,-'-,
72          # 3   4   5   6
73
74      def add (self, priority, data):
75          tree_index = self.index + self.capacity - 1
76          self.data[self.index] = data
77          self.update (tree_index, priority)
78          self.index = (self.index + 1) % self.capacity
79
80      def update (self, index, priority):
81          delta = priority - self.tree[index]
82          self.tree[index] = priority
83          # cascade up the tree
```

```
84          while index != 0:
85              index = (index - 1)//2
86              self.tree[index] += delta
87
88      def get_leaf (self, val):
89          parent = 0
90          while True:
91              left  = 2*parent + 1
92              right = 2*parent + 2
93
94              if left >= len (self.tree):
95                  # if the left child node would exceed the bounds of the tree
96                  # then the current node is a leaf
97                  break
98
99              if val <= self.tree[left]:
100                 parent = left
101             else:
102                 val -= self.tree[left]
103                 parent = right
104
105         data = parent - self.capacity + 1
106         return parent, self.tree[parent], self.data[data]
107
108     @property
109     def max_priority (self):
110         return np.max (self.tree[-self.capacity:])
111
112     @property
113     def min_priority (self):
114         return np.min (self.tree[-self.capacity:])
115
116     @property
117     def total_priority (self):
118         return self.tree[0]
119
120 class Memory:
121
122     def __init__ (self, capacity):
123         self.e = 0.01
124         self.a = 0.6
125         self.b = 0.4
126         self.clip = 1.0
127         self.tree = SumTree (capacity)
128
129     def append (self, experience):
130         max_priority = self.tree.max_priority
131         if max_priority == 0:
132             max_priority = self.clip
133         self.tree.add (max_priority, experience)
134
```

```
135     def sample (self, num):
136         batch = []
137         b_index  = np.empty ((num,),   dtype = np.int32)
138         b_weight = np.empty ((num,), dtype = np.float32)
139 
140         priority_segment = self.tree.total_priority / num
141         max_weight = num*self.tree.min_priority / self.tree.total_priority
142         max_weight = max_weight**(-self.b)
143 
144         for i in range (num):
145             a, b = priority_segment*i, priority_segment*(i + 1)
146             value = np.random.uniform (a, b)
147             index, priority, data = self.tree.get_leaf (value)
148             sampling_probability = priority / self.tree.total_priority
149 
150             b_weight[i] = (num*sampling_probability)**(-self.b) / max_weight
151             b_index[i] = index
152             batch.append (data)
153         b_weight = np.array (b_weight)
154 
155         return b_index, batch, b_weight
156 
157     def batch_update (self, index, err):
158         err += self.e
159         err = np.minimum (err, self.clip)
160         err = np.power (err, self.a)
161         for i, e in zip (index, err):
162             self.tree.update (i, e)
163 
164 class Model (tf.keras.Model):
165 
166     def __init__ (self):
167         super (Model, self).__init__ ()
168         l = tf.keras.layers
169         self.action_layer = l.Dense (2, input_dim = 5, activation = 'relu')
170         self.value_layer = l.Dense (1, input_dim = 5, activation = 'relu')
171         self.average_layer = l.Lambda (lambda x: x - tf.reduce_mean (x))
172         self.q_layer = l.Add ()
173 
174         self.compile (optimizer = 'adam',
175                       loss = 'mean_squared_error',
176                       metrics = [])
177 
178     def call (self, x_in):
179         x_a = self.action_layer (x_in)
180         x_a = self.average_layer (x_a)
181         x_v = self.value_layer (x_in)
182         return self.q_layer ([x_v, x_a])
183 
184 policy_a = Model ()
185 target_a = Model ()
```

```
186 target_a.set_weights (policy_a.get_weights ())
187 policy_b = Model ()
188 target_b = Model ()
189 target_b.set_weights (policy_b.get_weights ())
190
191 memory = Memory (size_memory)
192
193 # Populate Memory
194
195 for i in range (size_memory):
196     state0 = state_to_array (env.state)
197     action = random.randrange (2)
198     reward = env.step (action)
199     state1 = state_to_array (env.state)
200     memory.append ((state0, action, reward, state1))
201
202 # Training
203
204 scores = []
205 for episode in range (num_episodes):
206     eps = max_eps*(min_eps/max_eps)**(episode/num_episodes)
207
208     state0 = state_to_array (env.reset ())
209     score = 0
210     for step in range (num_steps):
211
212         # choose action
213         action = None
214         if random.random () > eps:
215             q_vals = policy_a.predict (np.array ([state0]))
216             if use_double:
217                 q_vals = q_vals + policy_b.predict (np.array ([state0]))
218             action = np.argmax (q_vals)
219         else:
220             action = random.randrange (2)
221
222         # perform
223         reward = env.step (action)
224         score += reward
225         state1 = state_to_array (env.state)
226         memory.append ((state0, action, reward, state1))
227         state0 = state1
228
229         # create training batch
230         b_index, batch, b_weight = memory.sample (size_batch)
231         batch = list (zip (*batch))
232         b_state0 = np.array (batch[0])
233         b_action = np.array (batch[1])
234         b_reward = np.array (batch[2])
235         b_state1 = np.array (batch[3])
236
```

```
237         # predict Q-values
238         policy = None
239         qf_val = None
240         a_star = np.empty (size_batch)
241 
242         if use_double:
243             target = None
244             othert = None
245             if random.random () < 0.5:
246                 policy = policy_a
247                 target = target_a
248                 othert = target_b
249             else:
250                 policy = policy_b
251                 target = target_b
252                 othert = target_a
253 
254             qp_val = target.predict (b_state1)
255             qf_val = othert.predict (b_state1)
256             a_star = np.argmax (qp_val, axis = 1)
257         else:
258             policy = policy_a
259             target = target_a
260 
261             qf_val = target.predict (b_state1)
262             a_star = np.argmax (qf_val, axis = 1)
263 
264         b_target = policy.predict (b_state0)
265         for i in range (size_batch):
266             a = b_action[i]
267             a_s = a_star[i]
268             r = b_reward[i]
269             b_target[i,a] += lr*(r + gamma*qf_val[i,a_s] - b_target[i,a])
270 
271         # fit values
272         policy.fit (b_state0,
273                     b_target,
274                     epochs = 10,
275                     batch_size = size_batch,
276                     verbose = 0)
277 
278         # update fixed q-values
279         if (step + 1) % tau == 0:
280             target_a.set_weights (policy_a.get_weights ())
281             target_b.set_weights (policy_b.get_weights ())
282 
283     # display episode results
284     print ("episode {0:3d}, score {1:3d}, dings {2:2d}, eps {3:6f}"
285             .format (episode, score, env.dings, eps))
286     scores.append (score)
287 
```

```
288 # write results to file
289 if data_filename:
290     with open (data_filename, 'w') as f:
291         for i in range (num_episodes):
292             f.write ("{0:d},{1:d}\n".format (i, scores[i]))
293 
294 plt.plot (scores)
295 plt.show ()
296 
297 # Play
298 for episode in range (num_testepis):
299     state0 = state_to_array (env.reset ())
300     score = 0
301     for step in range (num_steps):
302         q_vals = policy_a.predict (np.array ([state0]))
303         if use_double:
304             q_vals = q_vals + policy_b.predict (np.array ([state0]))
305         action = np.argmax (q_vals)
306         reward = env.step (action)
307         score += reward
308         state = state_to_array (env.state)
309 
310     print ("score", score)
```

Bibliography

Abadi, Martín et al. (2016). "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283.

Dozat, Timothy (2016). "Incorporating nesterov momentum into adam". In:

Duchi, John, Elad Hazan, and Yoram Singer (2011). "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of machine learning research* 12.Jul, pp. 2121–2159.

Elman, Jeffrey L (1990). "Finding structure in time". In: *Cognitive science* 14.2, pp. 179–211.

Farley, B. and W. Clark (1954). "Simulation of self-organizing systems by digital computer". In: *Transactions of the IRE Professional Group on Information Theory* 4.4, pp. 76–84. ISSN: 2168-2704. DOI: 10.1109/TIT.1954.1057468.

Fukushima, Kunihiko (1980). "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological cybernetics* 36.4, pp. 193–202.

Gers, Felix A and Jürgen Schmidhuber (2000). "Recurrent nets that time and count". In: *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium.* Vol. 3. IEEE, pp. 189–194.

Gers, Felix A, Jürgen Schmidhuber, and Fred Cummins (1999). "Learning to forget: Continual prediction with LSTM". In:

Hasselt, Hado V. (2010). "Double Q-learning". In: *Advances in Neural Information Processing Systems 23*. Ed. by J. D. Lafferty et al. Curran Associates, Inc., pp. 2613–2621. URL: http://papers.nips.cc/paper/3964-double-q-learning.pdf.

Hebb, Donald O (1949). *The organization of behavior*. John Wiley & Sons, Inc.

Hinton, Geoffrey E et al. (2012). "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580*.

Hinton, Geoffrey, N Srivastava, and K Swersky (2014). "Lecture 6e: Neural Networks for Machine Learning". In: *Coursera: Neural Networks for Machine Learning* 4.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural computation* 9.8, pp. 1735–1780.

Hubel, David H and Torsten N Wiesel (1959). "Receptive fields of single neurones in the cat's striate cortex". In: *The Journal of physiology* 148.3, pp. 574–591.

— (1962). "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". In: *The Journal of physiology* 160.1, pp. 106–154.

Jordan, MI (1986). *Serial order: a parallel distributed processing approach. Technical report, June 1985-March 1986*. Tech. rep. California Univ., San Diego, La Jolla (USA). Inst. for Cognitive Science.

Kelley, Henry J (1960). "Gradient theory of optimal flight paths". In: *Ars Journal* 30.10, pp. 947–954.

Kingma, Diederik P. and Jimmy Ba (2014). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980 [cs.LG].

Konda, Vijay R and John N Tsitsiklis (2000). "Actor-critic algorithms". In: *Advances in neural information processing systems*, pp. 1008–1014.

LeCun, Yann et al. (1989). "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4, pp. 541–551.

Lin, Long-Ji (1990). *Self-improving reactive agents: Case studies of reinforcement learning frameworks*. Carnegie-Mellon University. Department of Computer Science.

— (1992). "Self-improving reactive agents based on reinforcement learning, planning and teaching". In: *Machine learning* 8.3-4, pp. 293–321.

McClelland, James L, David E Rumelhart, PDP Research Group, et al. (1986). "Parallel distributed processing". In: *Explorations in the Microstructure of Cognition* 2, pp. 216–271.

McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133. DOI: 10.1007/BF02478259.

McMahan, H Brendan et al. (2013). "Ad click prediction: a view from the trenches". In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1222–1230.

Minsky, Marvin and Seymour A Papert (1969). *Perceptrons: An introduction to computational geometry*. MIT press.

Peng, Xue Bin et al. (2017). "Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning". In: *ACM Transactions on Graphics (TOG)* 36.4, pp. 1–13.

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6, p. 386. DOI: 10.1037/h0042519.

Rossum, Guido van (1996). *Foreword for "Programming Python" (1st ed.)* URL: https://www.python.org/doc/essays/foreword/ (visited on 02/07/2020).

Schaul, Tom et al. (2015). *Prioritized Experience Replay*. arXiv: 1511.05952 [cs.LG].

Schmidhuber, Jürgen (1992). "Learning Complex, Extended Sequences Using the Principle of History Compression". In: *Neural Computation* 4.2, pp. 234–242. ISSN: 0899-7667. DOI: 10.1162/neco.1992.4.2.234.

Schulman, John, Sergey Levine, et al. (2015). "Trust region policy optimization". In: *International conference on machine learning*, pp. 1889–1897.

Schulman, John, Filip Wolski, et al. (2017). "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347*.

Strens, Malcolm J. A. (2000). "A Bayesian Framework for Reinforcement Learning". In: *ICML*.

Sutton, Richard S et al. (2000). "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems*, pp. 1057–1063.

Waibel, Alex et al. (1989). "Phoneme recognition using time-delay neural networks". In: *IEEE transactions on acoustics, speech, and signal processing* 37.3, pp. 328–339.

Wang, Ziyu et al. (2015). "Dueling network architectures for deep reinforcement learning". In: *arXiv preprint arXiv:1511.06581*.

Watkins, Christopher John Cornish Hellaby (1989). "Learning from delayed rewards". In:

Yamaguchi, Kouichi et al. (1990). "A neural network for speaker-independent isolated word recognition". In: *First International Conference on Spoken Language Processing*.

Zeiler, Matthew D. (2012). *ADADELTA: An Adaptive Learning Rate Method*. arXiv: 1212.5701 [cs.LG].