Manchester Metropolitan University

Faculty of Science and Engineering

Alexander J. Johnson

Mathematics

Neural Networks with Python and TensorFlow

April 4, 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, and control physical/physically-based system in real-time.

`TODO`: *Abstract*.

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

Alexander Johnson

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.

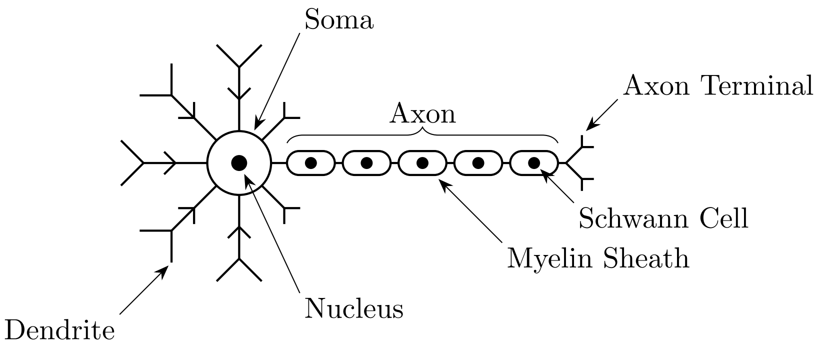


Figure 1.1: Diagram of a biological neuron.

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

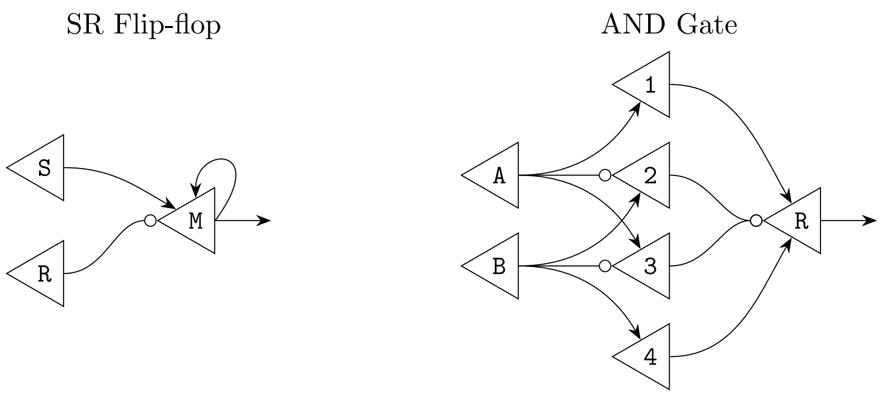
The idea of artificial beings capable of human intelligence can be traced back to mythical stories from ancient Greece. One such story was that of a mythical automaton called Talos, who circled an island's shores to protect it from pirates and other invaders.

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.



$N_M(t+1) = (N_S(t) \vee N_M(t)) \wedge \neg N_R(t)$ $\qquad$ $N_R(t+2) = N_A(t) \vee N_B(t)$

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 understanding 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.

When this 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.

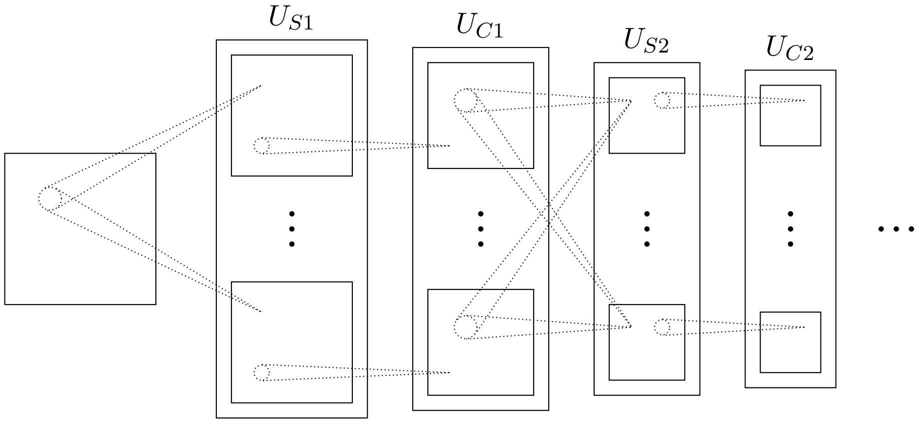


Figure 1.3: Representation of the neocognitron's connectivity.

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.

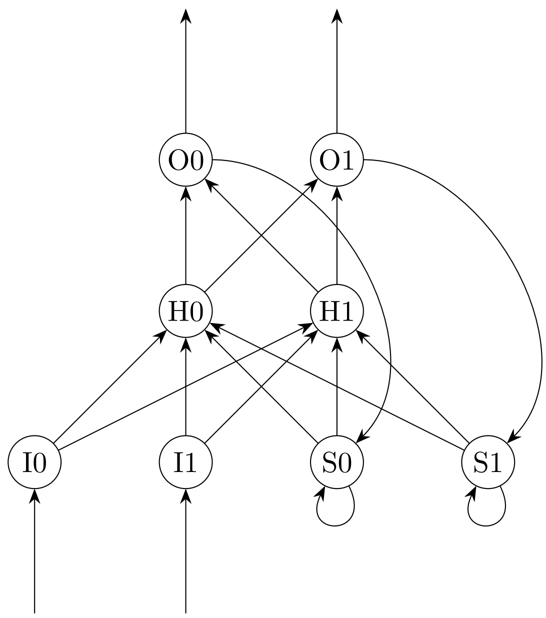


Figure 1.4: Example of Jordan's recurrent neural network.

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.

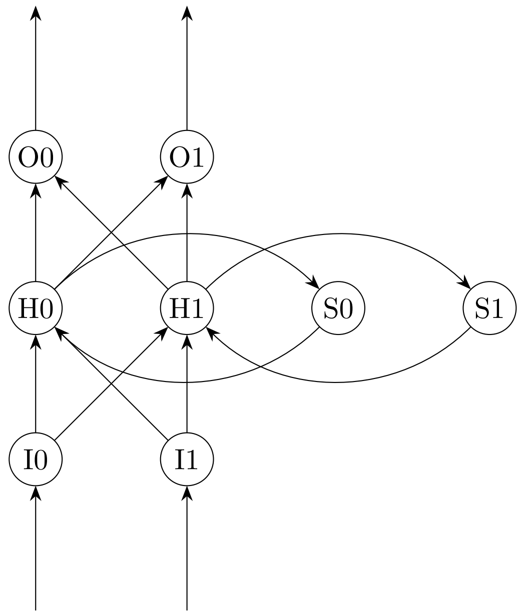


Figure 1.5: Example of Elman's recurrent neural network.

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.

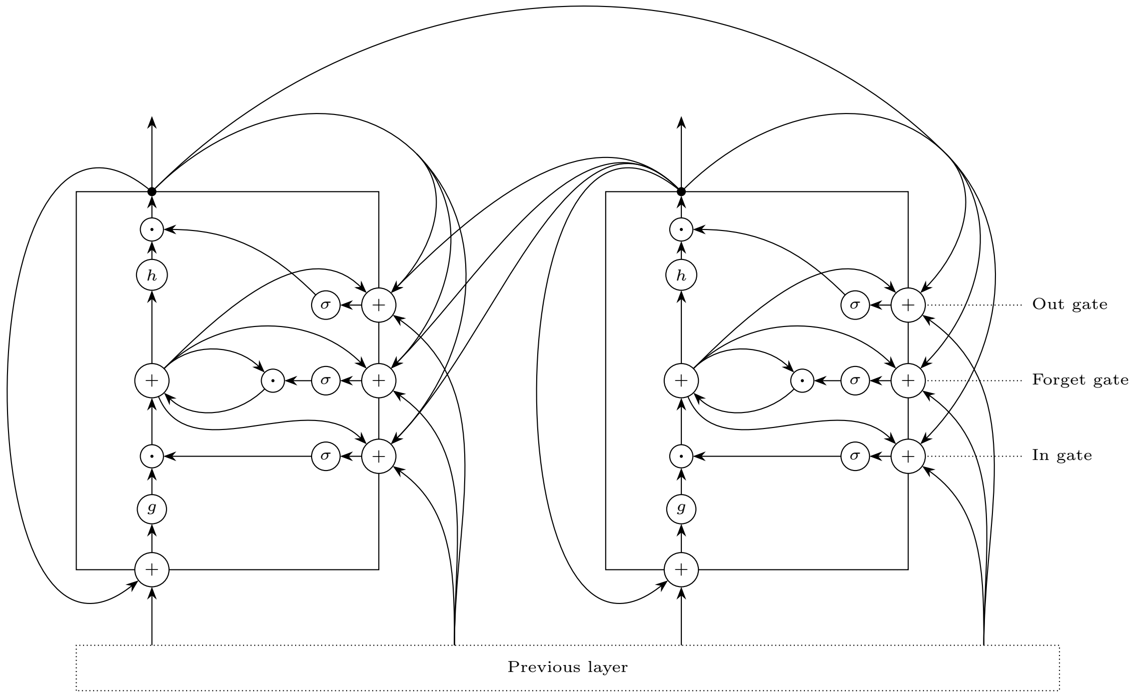


Figure 1.6: Visualisation of two LSTM units, where σ, f, and g are function units; $+$ is a weighted summation unit; and $\cdot$ is multiply unit.

Chapter 2

Introduction to Python

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

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

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

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

By chain rule,

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

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.

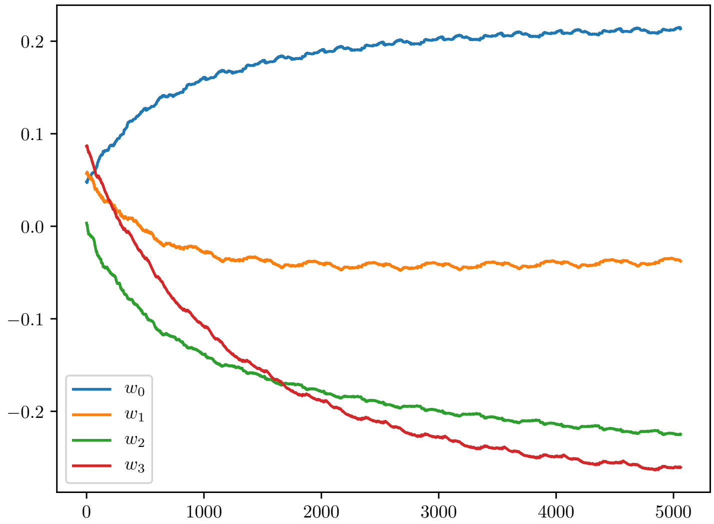


Figure 2.1: Graph of network weights against iteration number.

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.

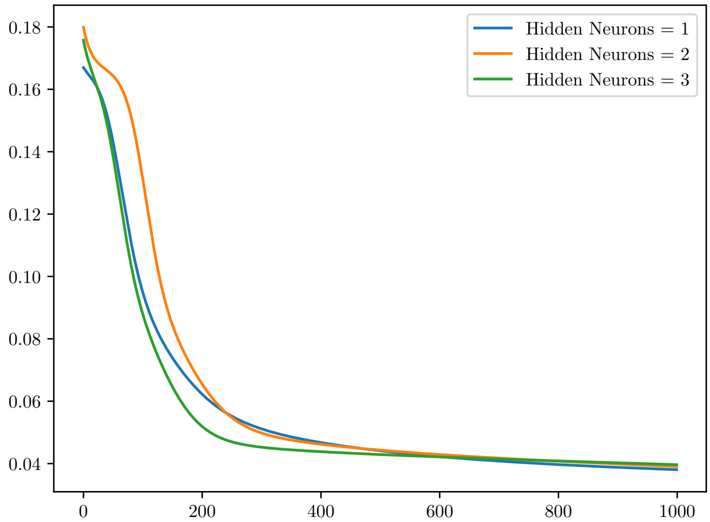


Figure 2.2: Graph of network error against iteration number for various numbers of hidden neurons.

The network was trained multiple times with varying numbers of hidden neurons. 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:

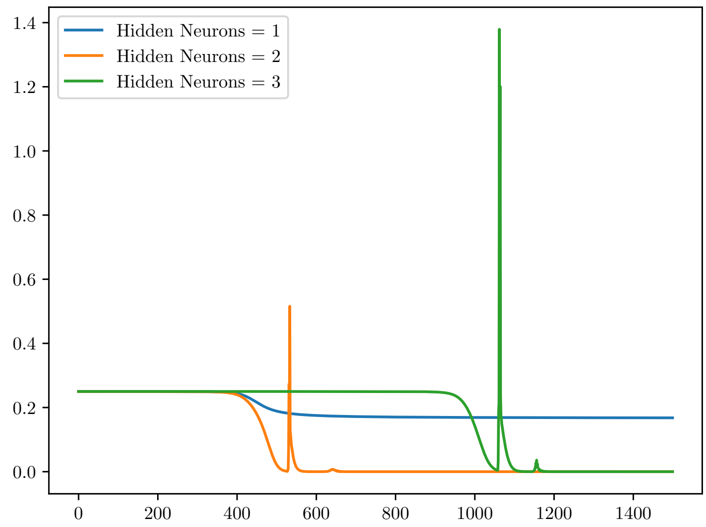
1. it had access to all 13 attributes, instead of the 3 attributes that had been manually selected; and

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

Logical XOR

Using the same, multilayer network architecture, a network was trained to perform the logical exclusive-or operation.

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



Using TensorFlow and Keras

Boston Housing Data

Logical XOR

Linear Regression

Character Recognition

Chapter 3

Introduction To TensorFlow

TensorFlow `TODO`: *Chapter: Introduction To TensorFlow*

Linear Regression

`TODO`: *Section: Linear Regression*

XOR

`TODO`: *Section: XOR*

Boston Housing with Keras

`TODO`: *Section: Boston Housing with Keras*

Chapter 4

Deep Learning

TODO: *Chapter: Deep Learning*

Recurrent Neural Networks

TODO: *Section: Recurrent Neural Networks*

Convolutional Neural Networks

TODO: *Section: Convolutional Neural Networks*

Image Processing

TODO: *Subsection: Image Processing*

Supervised Learning

TODO: *Section: Supervised Learning*

Unsupervised Learning

TODO: *Section: Unsupervised Learning*

Autoencoding

TODO: *Section: Autoencoding*

Reinforcement Learning

TODO: *Section: Reinforcement Learning*

Chapter 5

Null

TODO: *Decide parent chapter for CNN, RNN, and LSTM sections*

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:

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

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

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

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:

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

Bibliography

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:

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

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.

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

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.

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

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).

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.

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.

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