How effective are machine learning algorithms compared with traditional analytical techniques, with respect to playing abstract games?

Edmund Goodman – L6 REL
May 2019

Rouse Research Essay Award on Computer Science, supervised by Mr Gwilt
Word count: 3997

Figure 1: A piece of abstract art, generated from an input image of a chess set [1], using a convolutional neural network called DeepDream [2] to "find and enhance patterns in images via algorithmic pareidolia"

1 Abstract

Machine learning algorithms are increasingly commonly used to solve many problems, and in some cases are replacing other analytical techniques which were the previous status quo.

This essay investigates the effectiveness of both machine learning algorithms and analytical techniques, with respect to playing abstract games. I first describe neural networks from first principles as a tool for problem solving, considering their biological origins, structure, and processes for training them. I achieve this through a combination of research, mathematics and programmatic representations. Next, I consider analytical techniques, and provide the examples of mathematical analysis, genetic algorithms and tree search approaches, again by researching techniques, then implementing models of them myself. During this phase, I observed different results to those reported by popular mathematician Martin Gardner in his column in The Scientific American, which highlighted a subtle difference in training routines. Following this, I consider the cases where these techniques have been applied to games, taking the examples of AlphaGo, Stockfish and Chinook, seeing the development of these algorithms from early predictions to the cutting edge.

I found that machine learning techniques can be very effective when applied to playing abstract games, and have been shown to be able to beat traditional analytical techniques in some cases. However, they are not a silver bullet, and should not be applied to many problems. For example, applied to simple games, analytical techniques require less processing time, but they can be much less effective when applied to complex games.

This result has far-reaching consequences in many fields of research, since abstract games are appropriate analogues for many problems, and a very effective new technique to solve them could allow for breakthroughs in many different domains.

2 Table of Contents

3 Introduction

"Human beings are never more ingenious than in the invention of games"
- Gottfried Leibniz, in a letter to Blaise Pascal

The Merriam Webster dictionary defines games as "a physical or mental competition conducted according to rules with the participants in direct opposition to each other" [1]. They have been a staple of human society since the earliest civilisations, with the earliest complete board game "The Royal Game of Ur", dating back to 2500 BC [2], and the net worth of the gaming industry in 2018 being $43.8 billion [3].

Figure 2: One of the five surviving gameboards of the Royal Game of Ur [4]

Abstract games are a specific type of game, in which the theme is unimportant to the experience of gameplay [5], and, hence, can be usefully analysed by computer algorithms. They tend to be combinatorial, and played by two parties alternating a finite number of turns.

Abstract games are a useful place to develop techniques for solving problems in many fields, as these techniques can often be transferred to "real life" [6]. For example, game theory is prevalent in economics, and neural networks are used to perform microtransactions generating huge amounts of revenue [7].

Considering algorithms to play abstract games is useful, since many techniques developed can be applied to real world problems. Furthermore, they are more accessible to people without a grounding in STEM, as it is easier considering a game of tic-tac-toe than classifying a dataset of 60,000 handwritten digits [8], despite the fact both problems might have similar solutions.

Machine learning is a popular technique [9], and is being used in many areas of research [10] [11] [12]. However, it is important to contrast the current surge in development of machine learning to other techniques that might be more suited to specific problems.

In summary, abstract games are a useful problem-set to discover and develop algorithms which are useful in the real world, and both machine learning and traditional analytical techniques can be used to solve them.

4 Neural networks from first principles

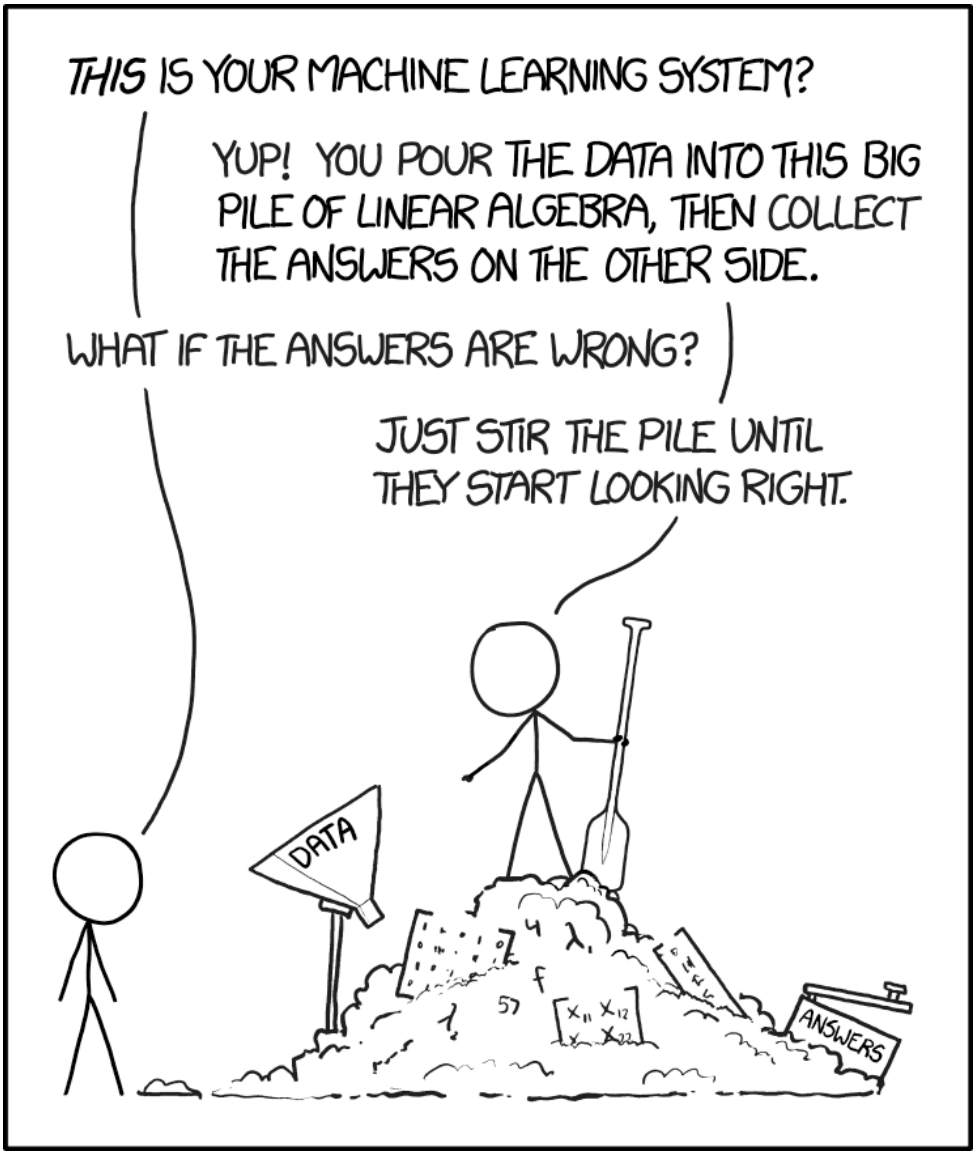


Figure 3: An XKCD comic by Randall Munroe addressing modern machine learning [13]

4.1 Introduction

Neural networks allow machines to learn from data without human intelligence, and in recent years have gained prominence as a means of solving many problems where other techniques have failed, or can only be applied to specific subsets of the problem space. They are normally used to classify large sets of data, and can often identify features other techniques and humans wouldn't find, for example in a large or multidimensional search space, or features which are difficult to perceive manually. However, they can also be applied to many other tasks, albeit with varying effectiveness [14].

Neural networks can be loosely considered as a mathematical model for processes that occur in biological brains, and their underlying structure was based on human neural structures.

4.2 From biological to mathematical systems:

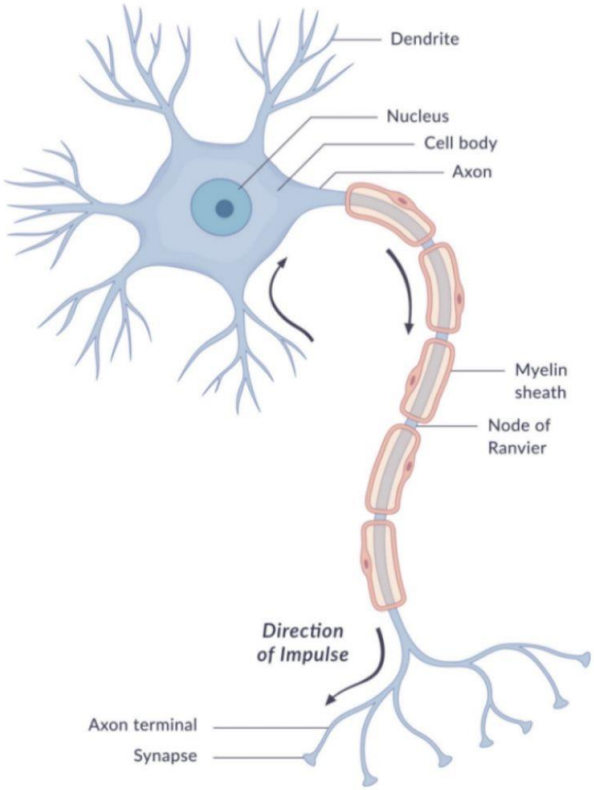


Figure 4: The structure of a human neuron [15]

As shown in Figure 4 (above), dendrites collect input signals from previous neurons' synapses, then the combined effect of all these signals passes into the axon. This signal then passes into multiple axon terminals, and then into synapses, which pass signals into other dendrites, and the process repeats itself [16]. This system appears simple, but the cumulative effect of hundreds of billions [17] of interconnected neurons is the definition of human intelligence.

This system can be mathematically modelled. A neuron is often called a node, which has inputs, like the dendrite, an activation function based on the cumulative effect of all the inputs, like the synapse, and outputs, like the axon terminal.

This mathematical model for the biological system was first proposed by Frank Rosenblatt as "Perceptrons" [18] [19], and has developed since.

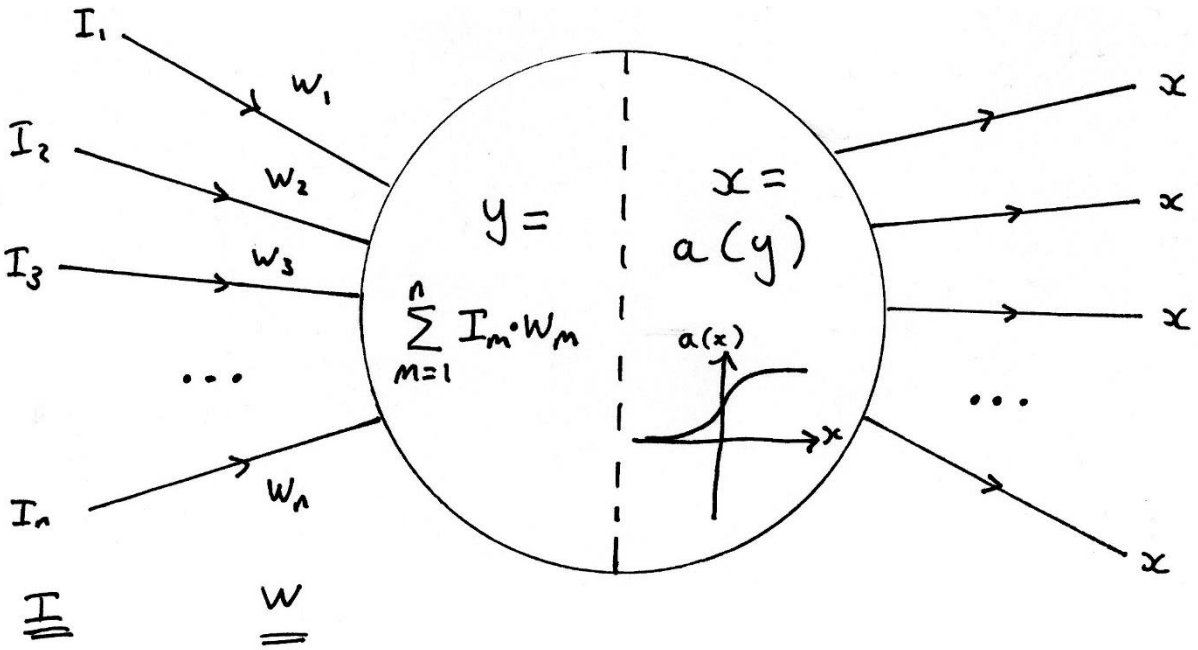


Figure 5: A diagram of a single node in a network

The output value of each node is evaluated as the sum of the products of its input values, I, and their weights, W, passed through an activation function, $a(x)$:

$$x = a\left(\sum_{m=1}^{n} I_m W_m\right)$$

A commonly-used activation function is the logistic sigmoid activation function [20], shown in Figure 6 (below), as it is non-linear, monotonic, and differentiable, so it can be used in multi-layer networks [21], and is easier to train.

$$a(x) = \frac{1}{1 + e^{-x}}$$

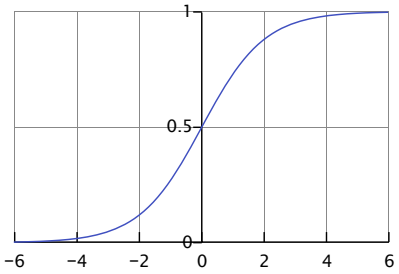


Figure 6: A sigmoid curve [74]

Other functions are also used, as detailed in appendix 8.6.8 (page 40)

4.3 Building a network

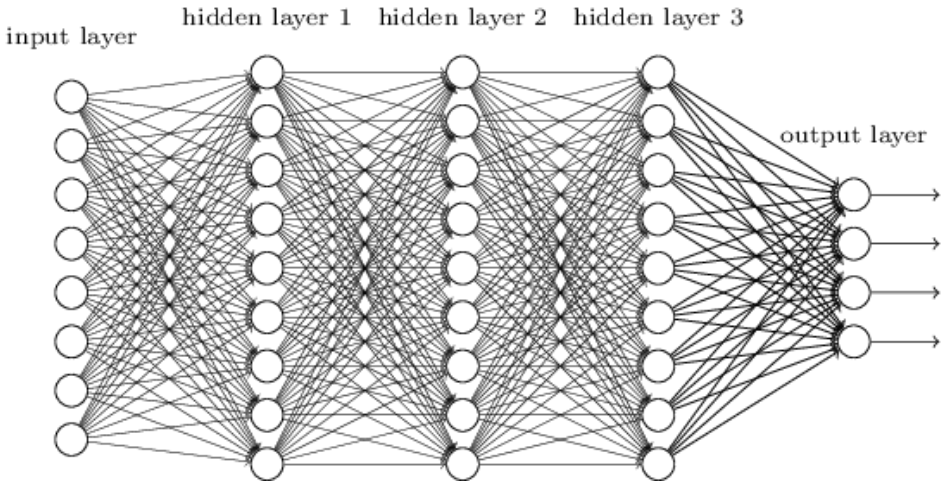


Figure 7: A deep neural network [22]

It is clear to see that individual nodes are not especially useful owing to their simplicity. However, they can be linked together to form larger networks, which allows more complex data to be modelled.

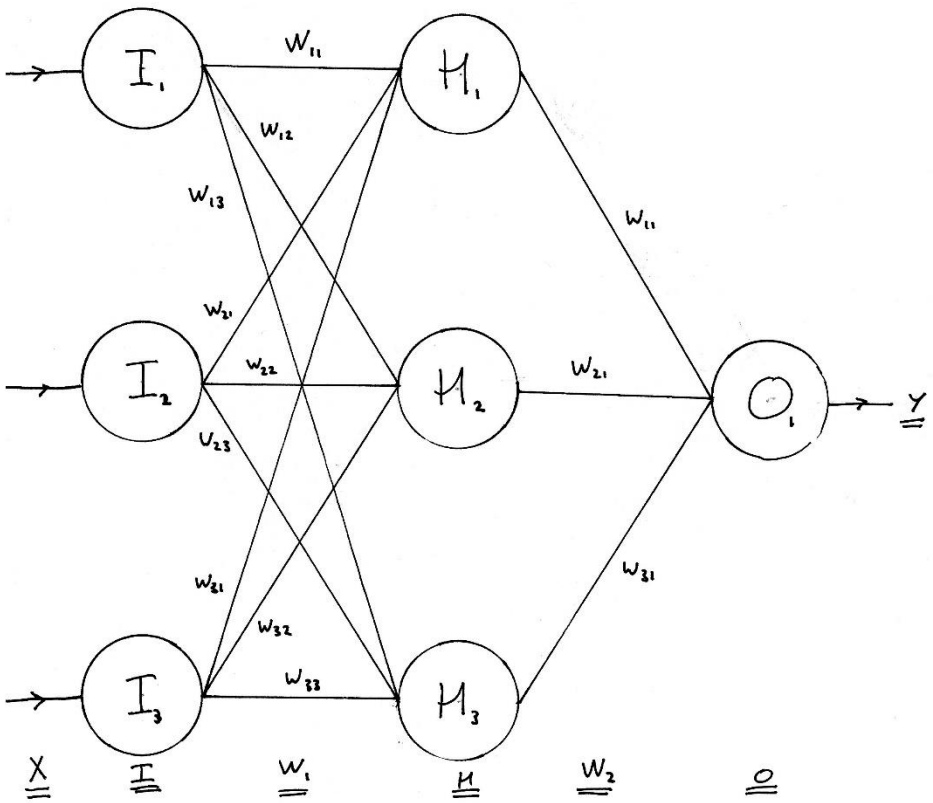


Figure 8: A small neural network, labelled to denote nodes and weights

Traditionally, a network is modelled as having input nodes, where each input data point maps directly to one node. Next, there are hidden layers, which for simplicity can be considered as shallow and fully connected, as shown in the Figure 8 (above). Each node in the hidden layer is connected to every node in the previous layer, and each of those connections have a weight. Finally, there are output nodes, which have many inputs, but only produce one output. This

model means data can be put into the network via its input nodes, and collected from the output nodes. The process of putting data though the network is known as forward propagation, and is performed by iterating over each layer, and for each node in that layer, evaluating it as previously described, and yielding its outputs.

$$H_n = a\left(\sum_{m=1}^{3} I_m W_{nm}\right)$$

$$O = a\left(\sum_{m=1}^{3} H_m W_{nm}\right)$$

This can also be expressed as a matrix multiplication, which is preferable during implementation, as there exist heavily optimised libraries to perform matrix operations, e.g. NumPy for Python

$$H = a(I \cdot W)$$

$$O = a(H \cdot W)$$

To ensure that I fully understood this, I implemented forward propagation in Python. A screenshot of the basic code is shown in Figure 9 (below):

```
def forwardpropagate(self, X):
    """Forward propagate an input vector X through the network composed of
    the weights self.W1 and self.W2, storing intermediary variables self.z
    and self.a to allow back propagation, and return the output vector
    of the process
    """

    #Forward propagate data though the network
    self.z, self.a = [], []
    #From input to hidden layer
    self.z.append( np.dot(X, self.W1) )
    self.a.append( self.activation(self.z[-1]) )
    #Through the hidden layers
    for i in range(len(self.W2) - 1):
        self.z.append( np.dot(self.a[-1], self.W2[i] )
        self.a.append( self.activation(self.z[-1]) )
    #From hidden to output layer
    self.z.append( np.dot(self.a[-1], self.W2[-1]) )
    self.a.append( self.activation(self.z[-1]) )
    #Return the final value of the output layer
    return self.a[-1]
```

Figure 9: A Python implementation of forward propagation

Having described the network, a protocol to teach it to produce correct output data from input data is required.

In the early days of development, networks were massive circuits, with each weight a potentiometer that was manually adjusted [23] [24]. Later they became computationally modelled, which allowed for more effective training algorithms.

In order to train a network, the input data is forward propagated through the network, and then compared to the expected output data. This comparison of the expected against the calculated yields the network's error, which is minimised to train the network. A naïve way to approach this would be to randomly try different weights, until reaching a set that is good enough for a given use case. However, this is very inefficient.

4.4 Training and the backpropagation algorithm

Networks can be trained using the backpropagation algorithm [25] [26], by considering how much each node contributed to the error in the output and adjusting the weights accordingly.

The mathematical derivation of backpropagation [27] is far out of scope for this project. However, the results required for an implementation of the algorithm are summarised below.

There are many online tutorials detailing single layer backpropagation [28] [29]. I extended the mathematics to multilayer backpropagation, and introduced suitable new nomenclature:

4.4.1 The mathematics of backpropagation

First, defining the variables:

$$y = the\ expected\ output\ of\ the\ network$$
$$\hat{y} = the\ output\ of\ forward\ propagating\ the\ network$$
$$X = the\ input\ to\ the\ network$$
$$H_n = the\ value\ during\ forward\ propagtion\ in\ the\ n^{th} hidden\ layer$$

$$W_{\overrightarrow{H_n O}} = the\ weight\ matrix\ between\ the\ final\ hidden\ layer\ and\ the\ output\ layer$$
$$W_{\overrightarrow{H_n H_{n+1}}} = the\ weight\ matrix\ between\ the\ hidden\ layers$$
$$W_{\overrightarrow{I H_1}} = the\ weight\ matrix\ between\ the\ input\ layer\ and\ first\ hidden\ layer$$

$$E_n = the\ matrix\ containing\ the\ error\ in\ the\ n^{th}\ layer\ weights$$
$$\Delta_n = the\ matrix\ the\ change\ to\ be\ applied\ to\ the\ n^{th}\ layer\ weights$$

$$a'(x) = a(x)\big(1 - a(x)\big) = the\ derivative\ of\ the\ sigmoid\ activation\ function$$

$$\eta = the\ learning\ rate\ of\ the\ network\ (a\ scale\ factor\ for\ the\ weight\ updates)$$

Next, specifying ambiguous operands:

$X^T = the\ matrix\ transpose\ of\ X$
$A \circ B = the\ entrywise\ product\ of\ matrices\ A\ and\ B\ (hadamard\ product)$
$A \cdot B = the\ dot\ product\ of\ matrices\ A\ \&\ B$

In essence, backpropagation is the process of tracing a route back through the network, considering how each weight contributed to the output error (the difference between the expected and calculated output values), then calculating a gradient to find which direction, and by how much, each weight should be changed, and finally adjusting that based on the amount it contributed to the error. More formally, backpropagation consists of:

First, backpropagating from the output layer to the final hidden layer:

$$E_1 = y - \hat{y}$$
$$\Delta_1 = E_1 \circ a'(W_{\overrightarrow{H_2O}})$$

Next, backpropagating within hidden layers (this step can be repeated for more hidden layers):

$$E_2 = \Delta_1 \cdot (W_{\overrightarrow{H_2O}})^T$$
$$\Delta_2 = E_2 \circ a'(W_{\overrightarrow{H_1H_2}})$$

Finally, backpropagating from the first hidden layer to the input layer:

$$E_3 = \Delta_2 \cdot (W_{\overrightarrow{H_1H_2}})^T$$
$$\Delta_2 = E_3 \circ a'(W_{\overrightarrow{IH_1}})$$

The weights can then all be updated, scaled by a learning rate η

$$W_{\overrightarrow{H_2O}} \mathrel{+}= \eta\left({H_2}^T \cdot \Delta_1\right)$$
$$W_{\overrightarrow{H_1H_2}} \mathrel{+}= \eta\left({H_1}^T \cdot \Delta_2\right)$$
$$W_{\overrightarrow{IH_1}} \mathrel{+}= \eta(X^T \cdot \Delta_3)$$

In order to train a network, this cycle, known as an epoch, of forward propagation, back propagation and updating weights, is repeated many times, until the network models the input data effectively.

As before, I implemented the processes of backpropagation: Figure 10 (below), and updating weights: Figure 11 (below), to ensure I fully understood them.

```
def backwardpropagate(self, X, y, yHat):
    """Back propagate the error of the most recent forward propagation
    through the network. producing the arrays self.error and self.delta
    to allow weight updating
    """

    #Back propagate the error through the network
    self.error, self.delta  = [], []
    #Back propagate from output to the last hidden layer
    self.error.append( y - yHat )
    self.delta.append( self.error[0]*self.activation(yHat, True) )
    #Back propagate through the hidden layers
    for i in reversed(range(0, len(self.W2))):
        self.error.append( np.dot(self.delta[-1], self.W2[i].T) )
        self.error[-1]
        self.delta.append( self.error[-1]*self.activation(self.a[i], True) )
```

Figure 10: A Python implementation of the backpropagation of error through a network

```
def updateWeights(self, X, learnRate):
    """Update the weights of the network based on the previous pass of back
    propagation in order to improve the network model
    """

    #Update the hidden layer weights
    j = 0
    for i in reversed(range(len(self.W2))):
        self.W2[i] += np.dot(self.a[i].T, self.delta[j]) * learnRate
        j += 1

    #Update the input layer weights
    self.W1[i] += np.dot(X.T, self.delta[j]) * learnRate
```

Figure 11: A Python implementation of the updating of weights following a pass of backpropagation

This training process has subtle modes of failure. Firstly, if the training data set is too small, or the network architecture is too simple, it doesn't produce a complete model of the system, which is called underfitting. Secondly, if the network is trained for too long on the data, it can produce a model that only fits the training data, not the general system, which is called overfitting. Finally, irrespective of good design and data, training networks requires time and

processing power, and a major flaw of large networks modelling complicated systems is that they require many resources to train effectively.

There are various techniques which can be employed to address these issues, which I detail in appendix 8.6 (page 38), and an example training cycle on a simple data set in appendix 8.1 (page 24).

4.5 Summary

In order to fully understand how neural networks function, it is useful to implement them yourself, as the process of development and debugging ensures a full understanding and exposes subtle errors that might otherwise go unnoticed. There are also resources that provide a taste of what neural networks do, such as TensorFlow playground [30], shown in Figure 12 (below).

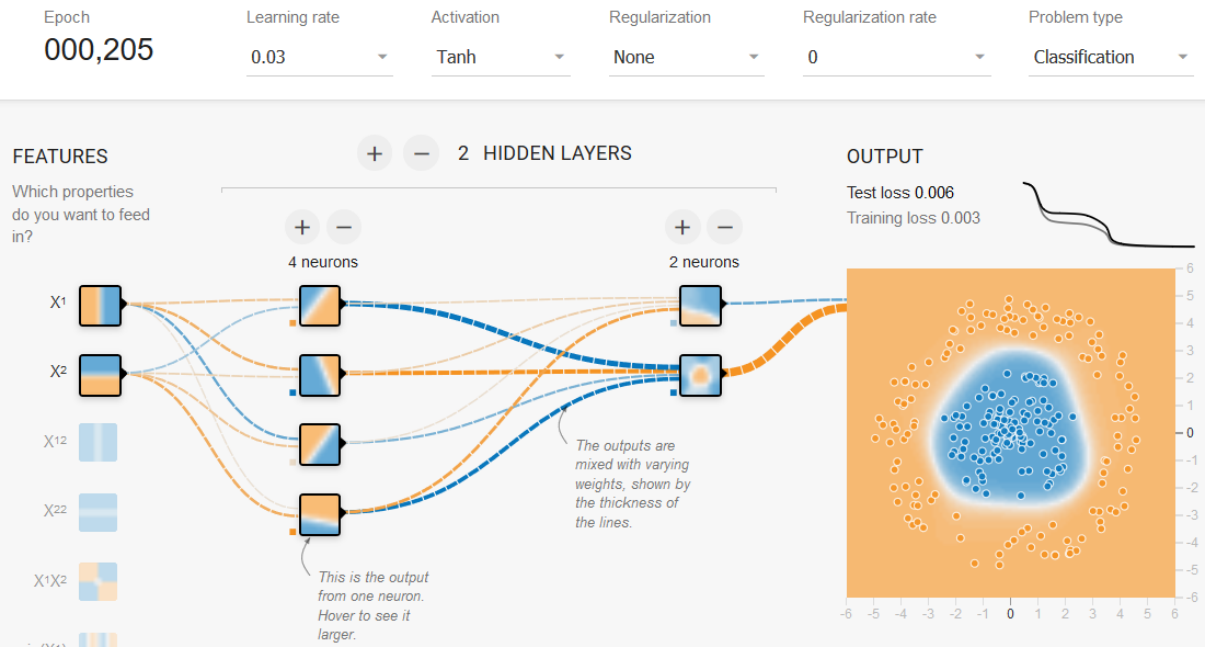
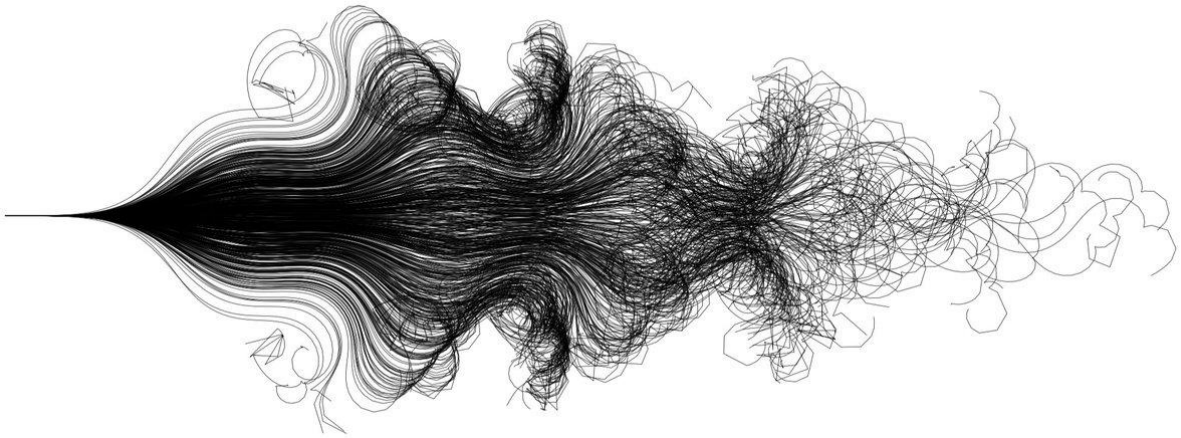


Figure 12: A screenshot of the TensorFlow playground

Neural networks are increasingly ubiquitous in many fields of research, and many researchers are trying to, often naïvely, use them as a silver bullet for difficult problems. Generally, they only work on problems with a large, comprehensive training set, and are only useful if traditional analysis of that set is not viable. Nonetheless, they can be incredibly effective for some problem types, and will no doubt grow more important in the future.

5 Analytical techniques



Instability of an unsteered bicycle. This shows 800 runs of a bicycle being pushed to the right. For each run, the path of the front wheel on the ground is shown until the bicycle has fallen over. The unstable oscillatory nature is due to the subcritical speed of the bicycle, which loses further speed with each oscillation.

Figure 13: An example of a dataset, which has been traditionally analysed [31]

5.1 Introduction

"Analytical techniques" is an ill-defined term. For the purposes of this essay I will tighten its meaning to that of techniques which are problem-specific, following analysis of the problem, as opposed to machine learning techniques, which can be applied to many different types of problem without much modification.

Since the scope of analytical techniques is so broad, I will only consider a few specific examples, including mathematical analysis, genetic algorithms, and tree search algorithms - all of which can be used to play games.

5.2 Example of mathematical analysis - Notakto

Not all games can be mathematically analysed, with only a minority that can be analysed yielding helpful results. When it is possible, analysis produces provably optimal algorithms which use little processing power, which is preferable over many other computational techniques, at the cost of human time spent doing the analysis.

A commonly used example is the game Nim [32] [33], which can be played optimally using a technique known as "nimbers". However, it is interesting to consider less well-known analyses for the sake of variety. Instead, consider the game Notakto, which is like traditional tic-tac-toe, except with both players placing the same counter (impartial play), and the player who makes three in a row loses (misère play). It is attributed to Bob Koca [34], then was mentioned in a Math Overflow thread [35], and finally solved by Plambeck and Whitehead [36].

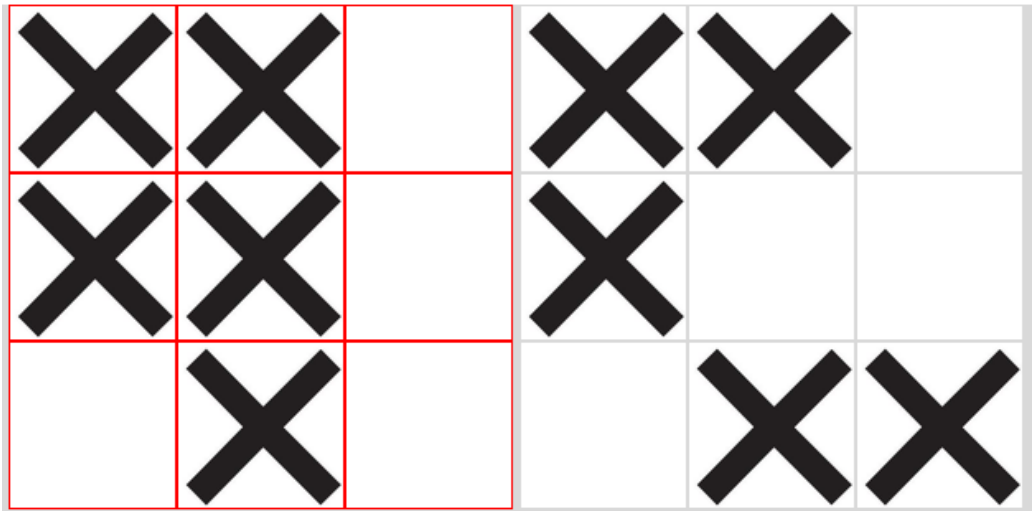


Figure 14: Two Notakto boards, with the left having been won, and the right still in play

A mathematical technique has been developed to play this game, from a more general approach to other misère games. In short, Plambeck and Whitehead suggest that a certain commutative monoid Q exists that is the misère quotient of the game.

$$Q = \langle a, b, c, d \mid a^2 = 1,\ b^3 = b,\ b^2c = c,\ c^3 = ac^2,\ b^2d = d, cd = ad,\ d^2 = c^2 \rangle$$
$$Q = \{1, a, b, ab, ab^2, c, ac, bc, abc, c^2, ac^2, bc^2, abc^2, d, ad, bd, abd\}$$

They then provide a lookup table for all 102 possible non-isomorphic board positions with elements in the monoid Q.

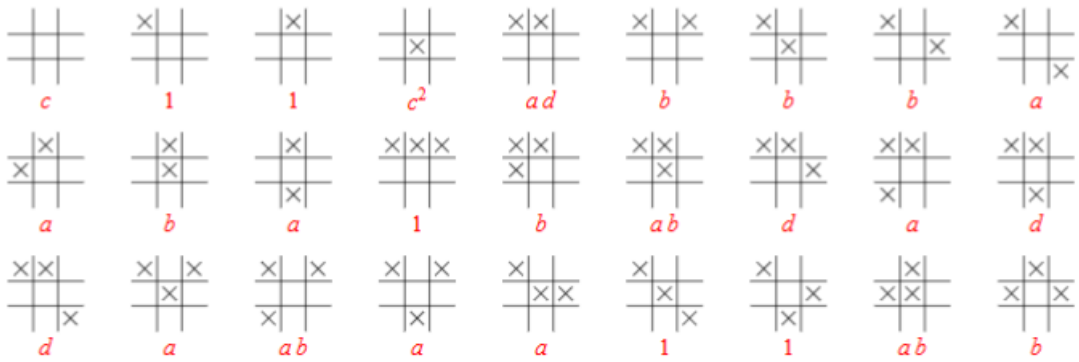


Figure 15: The first 27 items in the lookup table

They then posit that the outcome of the game can be determined by checking if the product of the representations of each board is equivalent to an element in the set P, in the commutative definition of Q.

$$P = \{a, b^2, bc, c^2\}$$

This can be used to determine the best move by considering all possible next moves, and selecting one which is predicted to win.

I implemented a computer system to model this, which is included in appendix 8.2 (page 25), and shows how effective and fast this technique is.

This approach allows a player to always win. However, such techniques are time-consuming to derive, and only applicable to individual games. Furthermore, it requires the memorisation of 102 values for game boards, which is impractical for most players. However, it does show that abstract games can be approached and solved mathematically, and whilst the solution may be unwieldy, it is guaranteed to be correct.

5.3 Example of genetic algorithms - Hexapawn

Another approach to play games optimally is the use of genetic algorithms. Genetic algorithms are designed to optimise a model, based on the way that biological systems operate, namely genetics and evolution.

These techniques are closer to neural networks than mathematical techniques, in that they are trained, rather than being the solution to the problem immediately. As a result, they can be quite processor and memory intensive, especially on games with a large search space.

A simple example was proposed by the popular mathematician Martin Gardner in his column in the Scientific American [37], as applied to a game which he called “Hexapawn”:

I have designed hexapawn, a much simpler game that requires only twenty-four boxes. The game is easily analyzed—indeed, it is trivial—but the reader is urged *not* to analyze it.

Hexapawn is played on a 3 × 3 board, with three chess pawns on each side as shown in Figure 43. Dimes and pennies can be used instead of actual chess pieces. Only two types of move are allowed: (1) A pawn may advance straight forward one square to an empty square; (2) a pawn may capture an enemy pawn by moving one square diagonally, left or right, to a square occupied by the enemy. The captured piece is removed from the board. These are the same as pawn moves in chess, except that no double move, *en passant* capture or promotion of pawns is permitted.

The game is won in any of three ways:

1. By advancing a pawn to the third row.
2. By capturing all enemy pieces.
3. By achieving a position in which the enemy cannot move.

Players alternate moves, moving one piece at a time. A draw clearly is impossible, but it is not immediately apparent whether the first or second player has the advantage.

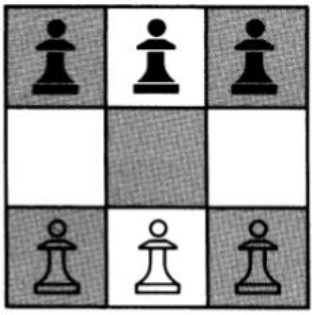


Figure 43
The game of hexapawn

Figure 16: Gardner's summary of hexapawn [38]

The purpose of this game is to have a small search space, in order to make it easier to physically manufacture the genetic algorithm. Gardner then describes the procedure to make and protocol to operate a “Hexapawn educatable robot”, using only matchboxes and beads, as contemporary readers didn’t have personal computers.

Figure 17: A physical "matchbox computer" to play Hexapawn [39]

In this protocol, each move is randomly selected, and then weighted negatively if it directly leads to losing the game. After many iterations, all the moves that could lead to a loss are removed, and hence the system can only win. This algorithm works well on games with small search spaces, such as hexapawn, especially when they can be cut down, further by removing isomorphic duplicates. However, on complicated games with large search spaces, they don't work well, as training essentially involves enumerating the entire game tree, which may not be possible.

5.3.1 Differing results

I made an implementation of the genetic algorithm following Gardner's algorithm, and found that it could train itself to play optimally. The description of the algorithm is available in appendix 8.3 (page 28). However, it takes many more games than Gardner suggested.

Gardner suggested that "HER" trained to optimal play with 11 losses. I found that my simulation trained in a minimum of 59 losses.

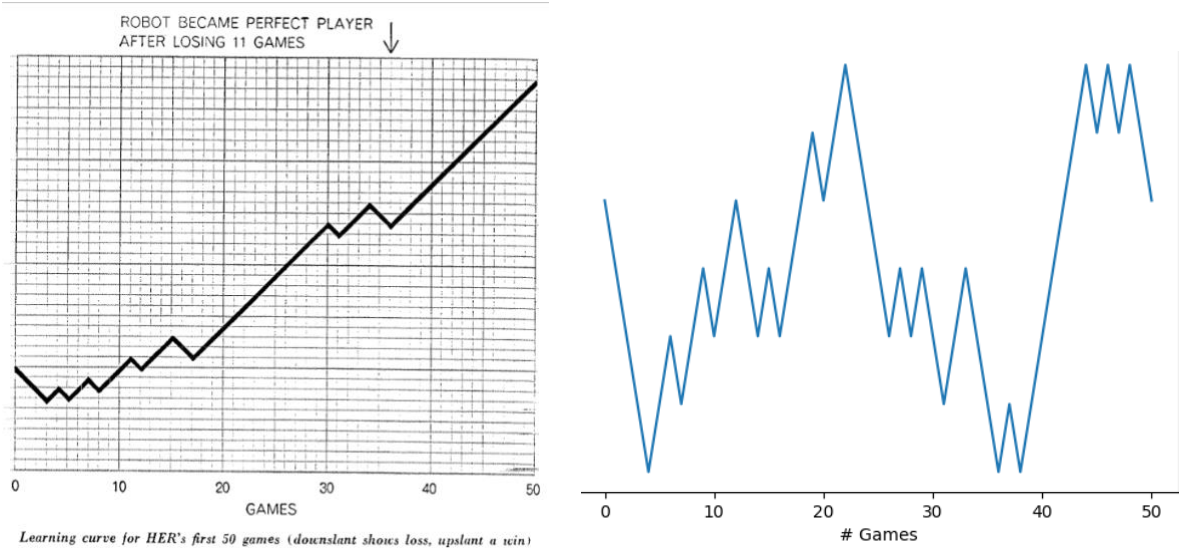


Figure 18: Plots of Gardner's and my results for the first 50 iterations of training, which look very different

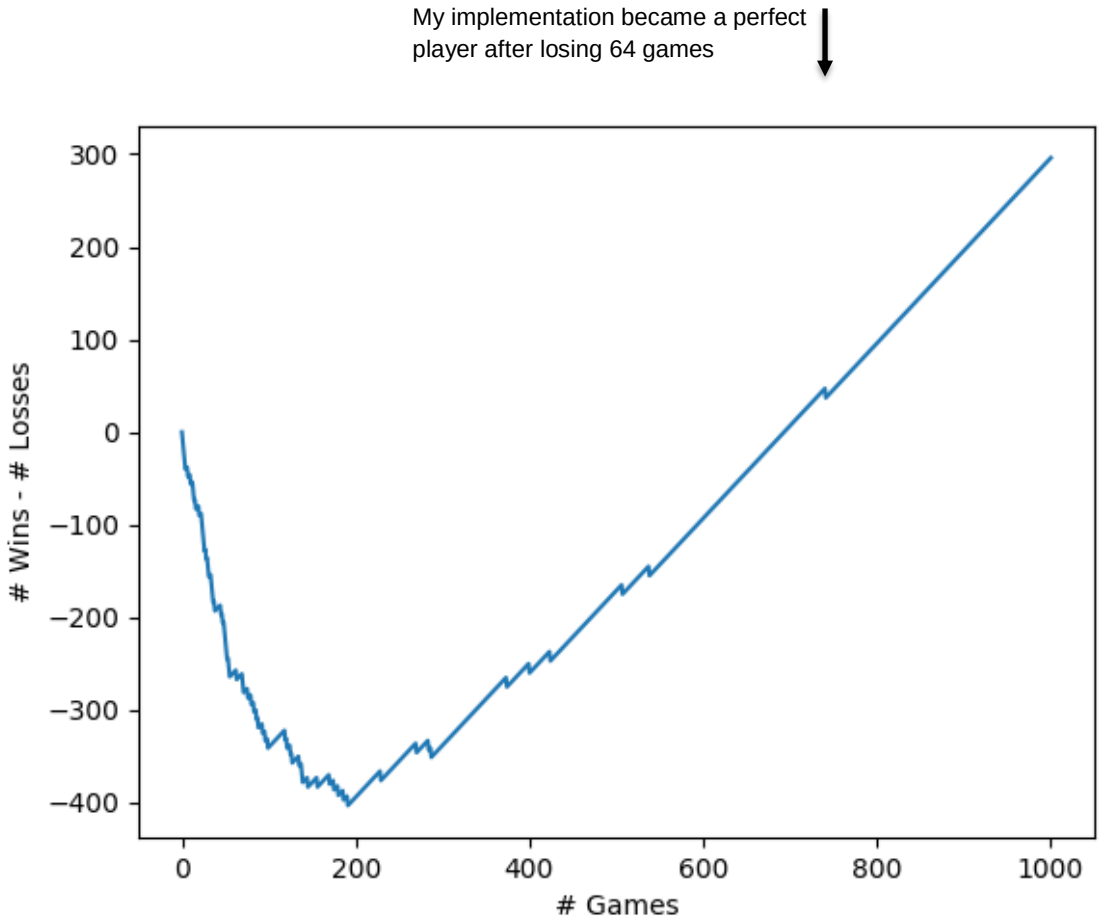


Figure 19: The training curve, with losses being scaled to be more visible (1:20 ratio). Notably, this is roughly the same shape as Gardner's curve, merely stretched along the x-axis

In order to train his system, Gardner used a tournament against a human. For ease and completeness of training I used a random algorithm to pick moves. As a result of this, I will use the metric of losses rather than games, since random moves are likely to repeat games, inflating the figure. A random training algorithm will eventually explore every possible game state, whereas the tournament training might only explore a few, since a human opponent would tend to play "sensible" moves, and not hence not test edge-cases.

An important issue to note is Gardner labelled his matchboxes using only non-isomorphic boards, as before with Notakto. Initially I did not model this, and it took about 123 losses to train. I then changed my data structure, reaching about 64 losses. I repeated the training routine with 20 different random seeds, and found the number of total losses varied slightly, between 59 and 71. I posit this occurs since, although all game states are explored, the random algorithm means some are explored first, and hence game states can be "cut off" with their parent state being removed before the algorithm encounters them, resulting in fewer losses occurring during training, since fewer states need to be removed. This is compounded by the fact that at the most losses, 24 states are explored, as Gardner suggested, however at fewer losses, fewer are, at minimum 22, states are explored. Since I trained with at most 50,000 games, I suggest with some certainty that the entire game tree has been explored.

It is possible that Gardner's statement "the system became a perfect player" [37] is erroneous, as he trained by manually playing games, which is time consuming, and he only played 14 games after the supposed turning point. Given the significant difference in the losses required to train our models I concluded that either my implementation is different to his description, or Gardner drew an erroneous conclusion, in that some moves can still result in a loss, so it is not a perfect player.

5.4 Example of tree search algorithms - Tic-tac-toe

One of the most commonly used techniques for solving games is by treating them as a tree of game states, where each state is a node, with nodes that are possible next game states after a move:

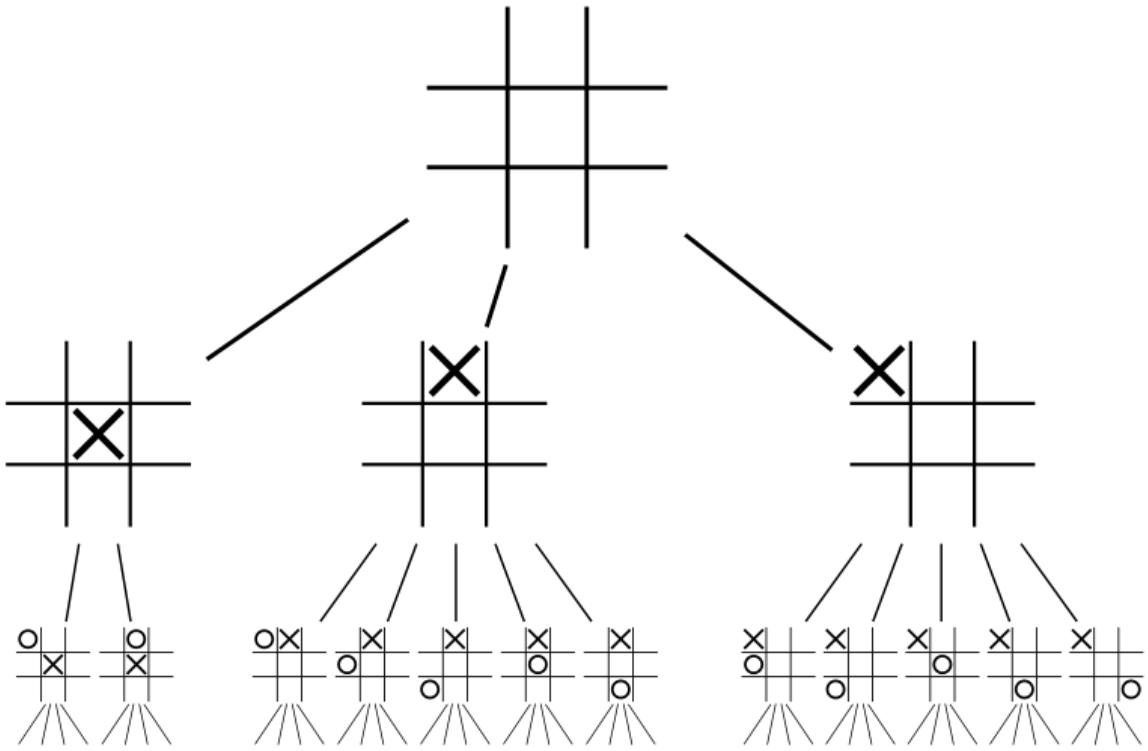


Figure 20: The first three layers of a tic-tac-toe game tree [40]

Since tree algorithms are fundamental and well understood [41], trees are a convenient data structure to use to find optimal moves.

The simplest way to do this is called the minimax algorithm. The minimax algorithm seeks to pick a decision that **mini**mises the number of ways the opposing player can win, and **max**imises the number of ways the current player can win.

It does this by recursively traversing the game tree. For odd-numbered layers, it tries to minimise the ways to win, and for even-numbered layers, it tries to maximise. It does this by assigning nodes values based on the sum of the values leaf nodes they lead to, with a positive value for a win, and a negative value for a loss. This summation propagates upwards, and the

node with the highest value is the node which is most likely to lead to the current player winning.

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

```
(* Initial call *)
minimax(origin, depth, TRUE)
```

Figure 21: The formal pseudocode describing the depth limited minimax algorithm [42]

I implemented the minimax algorithm to play Tic-tac-toe, which I detail in appendix 8.4 (page 30), and found it was effective, ensuring a draw, and even beating some of my peers. The algorithm explores every branch of the game tree, and hence is guaranteed to yield the optimal move. However, for more complex games, it can take an extremely long time as the game tree gets large. There are various optimisation techniques to make it faster (appendix 8.4.1 (page 32)).

In summary, tree search algorithms work very well applied to abstract games, and are generally the most commonly used in high performance systems, as discussed in the next section.

6 Instances of the application of algorithms to games

Alongside the technical aspects of machine learning algorithms and analytical techniques, I researched cases in which they have already been applied to playing abstract games.

6.1 Applications of analytical techniques

Chinook is a computer system designed to play checkers, developed between 1989 and 2007, and is the first computer program to win a human world championship [43]. It was entirely based on human knowledge, rather than using any machine learning. Chinook first uses an "opening book", with moves taken from games played by grandmasters. Next, it uses a deep tree search algorithm, which has a board evaluation function using heuristics chosen by humans. Finally, there is a similarly constructed end-game database, for all boards with fewer than eight pieces. This results in a system that is a mix of mathematical analysis and tree search

algorithms – making it a useful example as it is simple, and all the heuristics are hand-written, and hence understandable.

Initially, the aim for the project was to beat the best human players. It drew against "the best checkers player of all time" Marion Tinsley [44], and following his withdrawal due to pancreatic cancer, it beat Don Lafferty 1-0. Following this, the aim was shifted to solving checkers, which was finally completed in September 2007 [45]

There are many other systems which apply analytical techniques to games. One of the most notable is Stockfish, which competed against AlphaZero, a machine learning approach, and was comprehensively beaten, losing every game in the tournament, as discussed in the next sub-sections.

6.2 Applications of machine learning techniques

AlphaGo was a project to produce a system that could beat the best human Go players in the world, by Google's DeepMind [46] [47]. Go was chosen, as it is viewed as one of the most complex board games played by humans, and no other computer system had beaten a human professional on a full-sized board without handicaps.

In March 2016, it played a well-publicised game against Lee Sedol, a 9th dan (the highest rank) Go player, where it won 4-1. In the 37th move of game two, it made a move that "no human ever would". One commentator exclaimed, "That's a very strange move", but the system proceeded to win the match. It was later calculated there was a "one in ten thousand" chance a human would have made that move. This highlights one of the ways neural networks can be incredibly effective, they can produce creative outputs, contrasting analytical techniques which will not. However, in game four, Sedol also played an equally surprising move, which AlphaGo did not expect any human to make, and lost the game. In summary, in the highest-level games both human and computer can produce creative "beautiful" moves. However, modern computers win almost all the time [48].

Following AlphaGo, there were three successor projects, AlphaGo Master, AlphaGo Zero, and AlphaZero. The later versions both took "Zero knowledge" approaches, where there was no human interaction in the training, and they both only used neural networks, as opposed to earlier systems, which used analytical techniques in tandem. These later systems show that neural networks on their own can be incredibly powerful tools, which can sometimes exceed even the best analytical techniques.

6.3 Competing machine learning with analytical techniques

AlphaZero was shown to be immensely powerful after beating the foremost chess playing computer system Stockfish in a 100 round face off, winning 25 games as white, 3 as black and drawing the remaining 72.

In that tournament Stockfish used a Monte Carlo heuristic tree search algorithm to evaluate 70 million board states, whereas AlphaZero only evaluated 80 thousand [49]. AlphaZero could do this since it used machine learning techniques to filter down the set of boards to explore.

This idea of filtering board states, and only exploring promising ones is discussed on page 286 of *Gödel Escher Bach*, where Hofstadter states that "chess masters perceive the distribution of chunks […] a higher level description of the board than [the position of individual pieces]", following research that when remembering board states of a chess game "masters' mistakes involved groups of pieces […] which left the game strategically the same", and "masters were […] no better than novices in reconstructing random boards." He suggests that only considering high-level strategically valid moves, rather than just legal ones by master players in a process of "implicit pruning" is how masters play so effectively. [50] This book from 1979 seems to describe the process AlphaZero uses to select moves with remarkable foresight, and shows the progress of machine learning from formulation to realisation.

Many commentators and chess authorities were impressed by the achievement, with Garry Kasparov, who was previously defeated by Deep Blue (the first computer to win at chess against a reigning world champion) saying "it's a remarkable achievement, even if we should have expected it after AlphaGo" [51] and the Danish grandmaster Peter Heine Nielsen likened its play to that of a superior alien species [52].

However, there are criticisms of the game between Stockfish and AlphaZero, as AlphaZero trained for 9 hours before the match on specialised hardware called TPUs [53], whereas Stockfish only ran in the match, and it ran on hardware that was stated to be sub-optimal for the task [54]. Finally, others in the computer chess community stated that a newer version of Stockfish was likely to beat AlphaZero [55].

7 Conclusion

In my research I sought to consider: "How effective are machine learning algorithms compared with traditional analytical techniques, with respect to playing abstract games?"

Both machine learning algorithms and traditional analytical techniques can be very effective at playing abstract games. Machine learning approaches have recently proven more effective, however both techniques exhibit superhuman levels of play. This could change in the future, following advances in processing power from Moore's law and quantum computation, so the comparison remains somewhat unresolved. Traditional analytical techniques tend to be faster applied to simple games, whereas for complex games the more heuristic nature of machine learning tends to work better. My implementation of Gardner's genetic algorithm yielded different results to those reported in his article, from which I concluded that either my implementation didn't accurately model the physical system, or Gardner drew an erroneous conclusion.

In summary, machine learning techniques are being used increasingly to play abstract games over or in combination with traditional analytical techniques, with promising results. This might be in part as they are popular for researchers, as they are able to secure funding, however they have also proven to be very effective in their own right, and have far exceeded human level of play.

8 Appendices

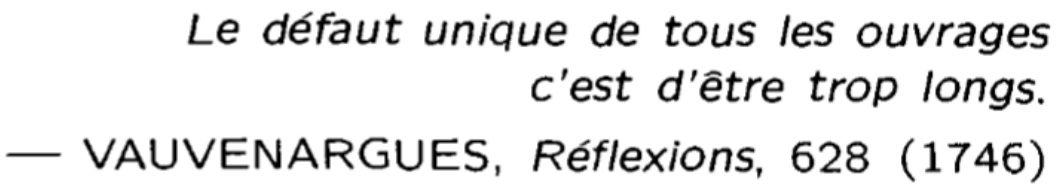


Figure 22: Donald Knuth: The Art of Programming Volume 1: Fundamental Algorithms, page xiv [33]
Translation: "The only defect of all works is to be too long"

The following appendices are quite extensive, much too long to include in the main essay, however, they are crucially important, since they outline the technical aspects of software, which I wrote to better understand my topic of research, and provide further information about the various techniques.

All the code I have referenced throughout this essay is written in Python, predominantly conforms to PEP8 [56], and is available on a public GIT server: https://github.com/EdmundGoodman/rouse-research

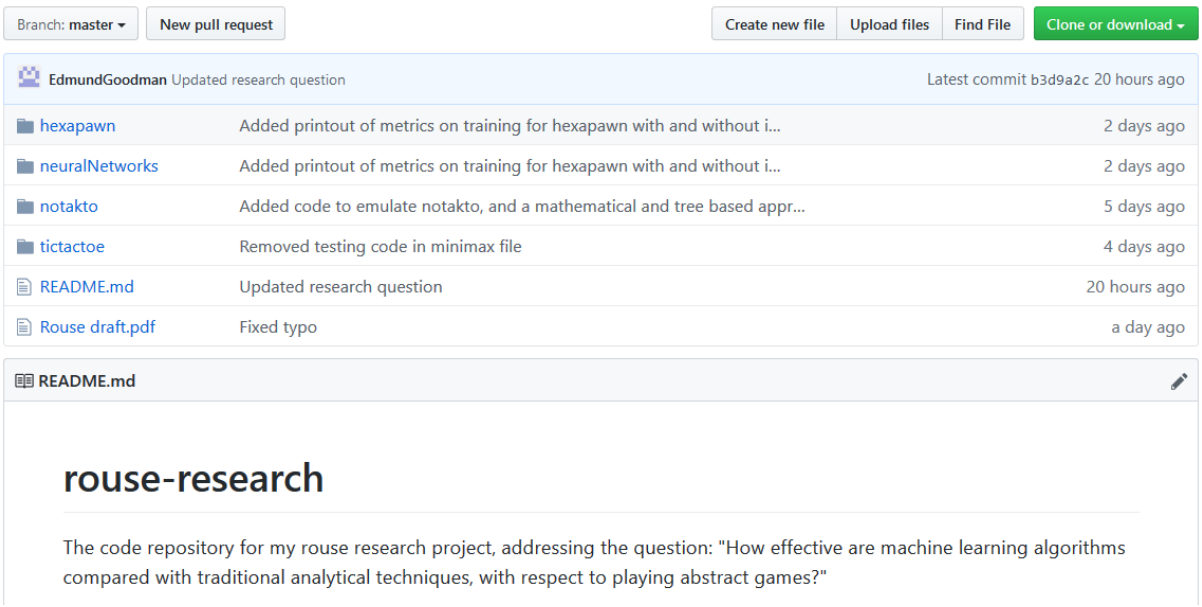


Figure 23: A screenshot of the git server webpage

8.1 Training first principles neural networks

In order to ensure that my implementation of the neural network functioned as expected, and to get a better understanding of the process of training before attempting to play Tic-tac-toe, I trained the network on a common benchmarking dataset: Iris [57]

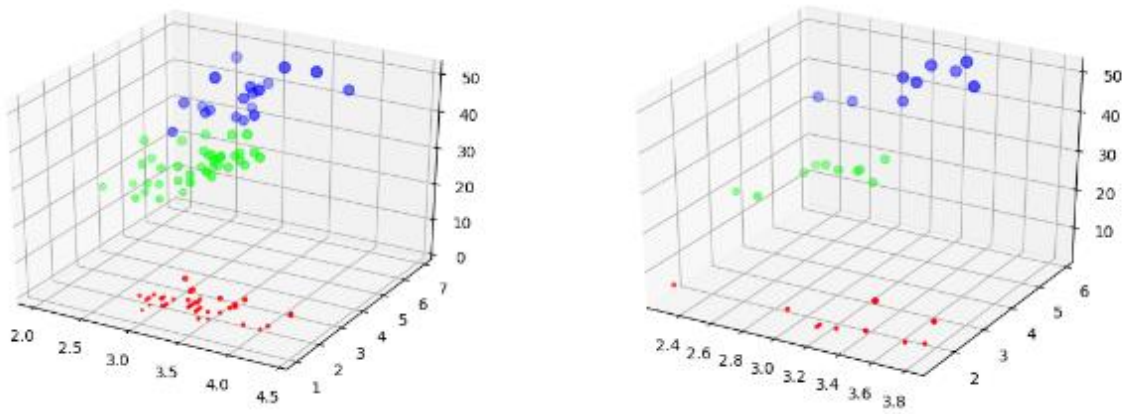


Figure 24: A graph of the training (left) and testing (right) data in the iris data set, using Matplotlib in Python

The Iris data set is a multivariate set introduced by Ronald Fisher in 1936, which is now used as a simple classification problem for testing neural networks. The expected categories are given by colour in the above graph, and the data encoded in the three axis and size.

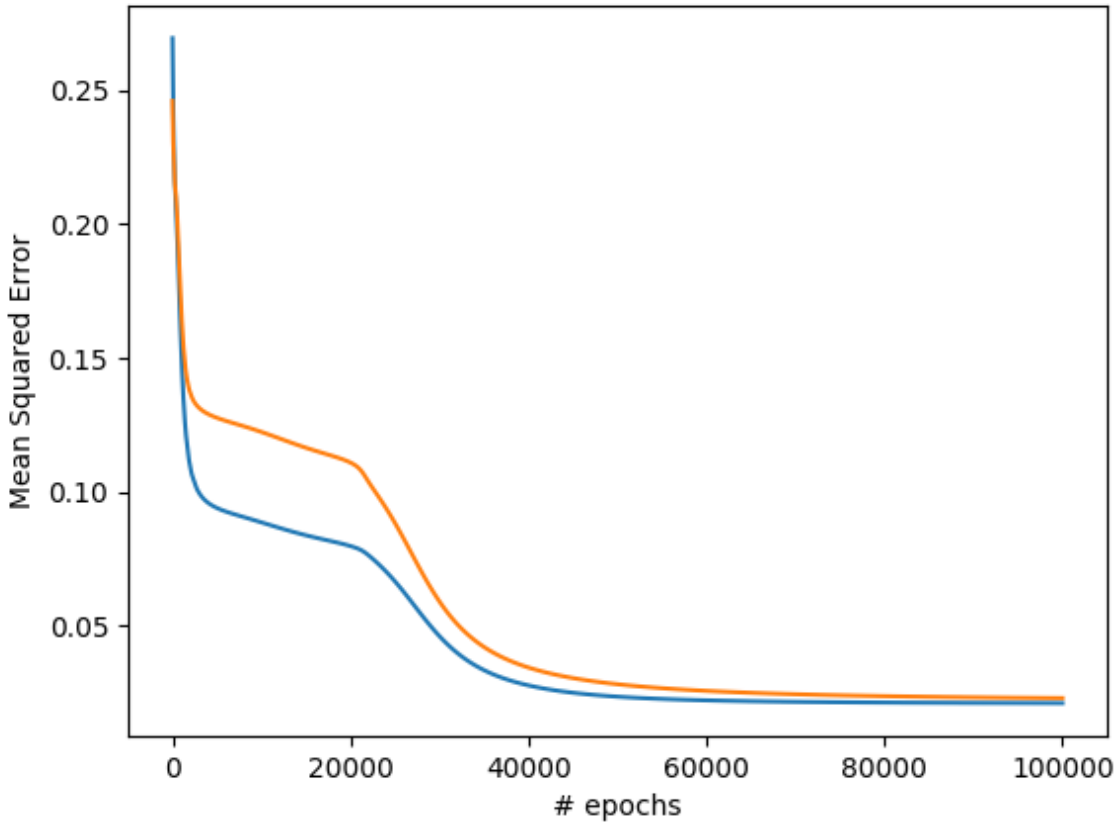


Figure 25: A graph of the mean squared error during training, with the blue line being testing, and the orange training data, using Matplotlib in Python

I calculated mean squared error during training, and plotted it on a graph, which can be seen to converge to a very small value, indicating successful training. I yielded a 98.0% training accuracy and a 100.0% testing accuracy – confirming my initial success.

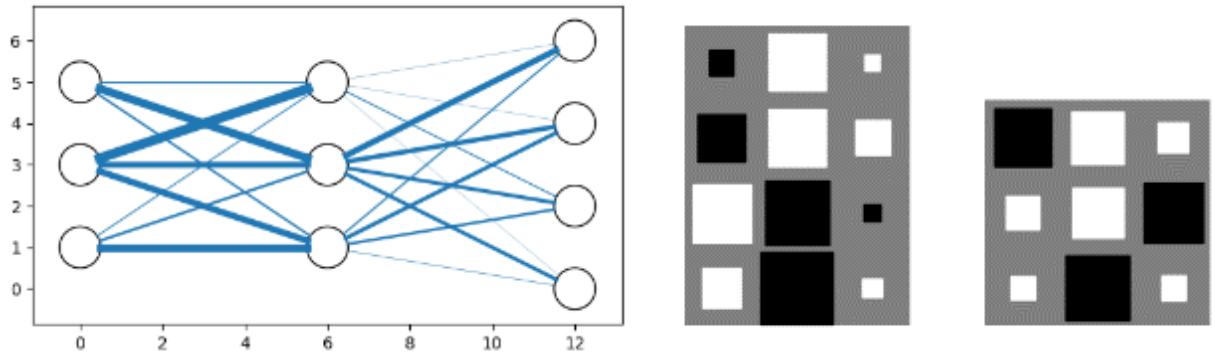


Figure 26: A representation of the trained network as a network diagram (left) and Hinton diagram (right), using Matplotlib in Python

I then wrote software to visualise the network as network and Hinton diagrams, so I could see the internal workings of the system, and verify it works.

8.2 Implementing mathematical analysis to solve Notakto

It is possible to find the optimum strategy for a game of Notakto involving only one 3x3 board after playing a couple of games by inspection of two starting cases:

1) If the opponent player plays first with any starting move other than the centre, the player can play the move mirrored through a centre line, and continue doing so until they win.

2) If the player plays first, it should always play in the centre position of the board, then a "knight's move" away from the opponent's piece, and continue doing so until they win.

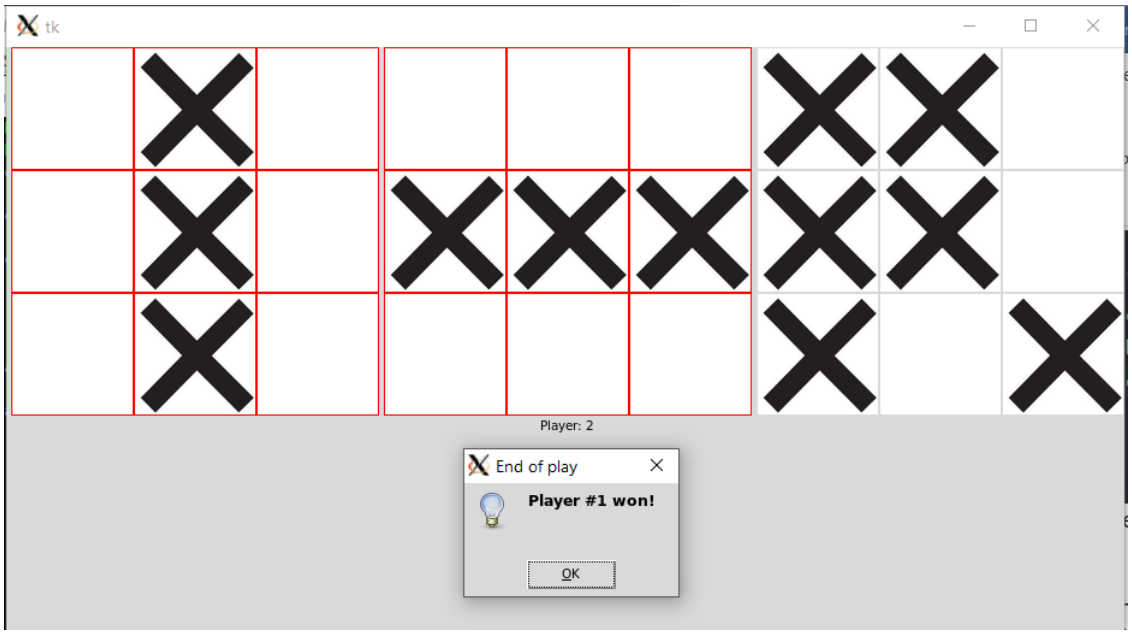
However, if play is extended to 3 concurrent boards, the problem gets much more complicated.

I implemented a computer program to graphically model 3 board, 3x3 Notakto, which allowed both human vs human, and more interestingly human vs AI play. Since the approach relied on the paper involves memorising a vast set of board states, and their corresponding polynomial terms I thought that it would be a good fit for a computer implementation, as the main problem humans face with the memorising this data is easy for a computer. I manually translated all the board states to a Python hash table of each distinct board state and its complementary polynomial, along with the transformations in the definition of the monoid Q, and the target safe states as a list.

```
data = {
    ((0,0,0),(0,0,0),(0,0,0)):"c",
    ((0,0,0),(0,1,0),(0,0,0)):"cc",
    ((1,1,0),(0,0,0),(0,0,0)):"ad",
    ((1,0,1),(0,0,0),(0,0,0)):"b",
    ((1,0,0),(0,1,0),(0,0,0)):"b",
    ((1,0,0),(0,0,1),(0,0,0)):"b",
    ((1,0,0),(0,0,0),(0,0,1)):"a",
```

```
reductions = {"aa":"","bbb":"b","bbc":"c",
              "ccc":"acc","bbd":"d","cd":"ad",
              "dd":"cc","":"aa", "b":"bbb",
              "c":"bbc","acc":"ccc","d":"bbd",
              "ad":"cd","cc":"dd"
}

targets = "a bb bc cc".split()
```

First, I produced a GUI for the game and tested it in human play, as this feature would speed up debugging later, then moved onto the AI component.



I realised that the process of mutating the polynomials to see if they were in the set of safe states was as non-trivial for a computer as a human.

In order to check if the board is a safe state, I concatenated the polynomial states of each board, then implemented a depth limited tree search of the possible transformations, until either a target state or the recursion depth is hit, indicating whether or not the string is able to be reduced to a target state.

```
def getValidChildren(self, node):
    children = []
    #Apply one substitution at a time, and defer nested over the tree
    for string,substitution in reductions.items():
        start = node
        for char in node:
            oldString = string
            string = string.replace(char,"",1)
            if oldString != string:
                start = start.replace(char,"",1)
        if string == "":
            children.append("".join(sorted(start+substitution)))
        else:
            pass
    return list(set(children))
```

```
def reducePolynomial(self, polynomial):
    polynomial = "".join(sorted(polynomial))
    count, tree, visited = 0, [polynomial], [polynomial]
    while True:
        #print(count, tree)
        next = []
        for node in tree:
            if node in targets:
                return node
            next.extend(self.getValidChildren(node))
        next = [n for n in next if n not in visited]
        visited.extend(next)
        tree = next[:]
        #If the recursion depth is exceeded
        if count > 3 or len(tree) == 0:
            return sorted(visited, key=lambda x: len(x))[0]
        count += 1
```

Using the function to check if the board state was safe, the AI then randomly selects moves from the safe board states, along with a heuristic suggested by the author for human play of “killing” boards if possible.

```
Human ccc ['a', 'bb', 'bc', 'cc']
AI cc ['a', 'bb', 'bc', 'cc']
Human bcc ['a', 'bb', 'bc', 'cc']
AI cc ['a', 'bb', 'bc', 'cc']
Human ccc ['a', 'bb', 'bc', 'cc']
AI cc ['a', 'bb', 'bc', 'cc']
Human b ['a', 'bb', 'bc', 'cc']
AI bb ['a', 'bb', 'bc', 'cc']
Human b ['a', 'bb', 'bc', 'cc']
AI a ['a', 'bb', 'bc', 'cc']
Human b ['a', 'bb', 'bc', 'cc']
AI a ['a', 'bb', 'bc', 'cc']
```

Figure 27: The AI player maintaining a safe board state throughout a game (printed in format: player, current state, safe states)

I found that the finished product exceeded my expectations in terms of how effectively it played. However, I experienced various bugs during testing, for example errors in the data file, and conceptual errors in the algorithm. I have tested the finished product to an extent, but I cannot easily verify if it is perfect, due to the massive search space, and as I previously found bugs it is important to consider the possibility there may be more lying undiscovered.

8.3 Implementing genetic algorithms to solve Hexapawn

Gardner initially suggested making a physical system to act as a genetic algorithm. However, making the physical system is time consuming, and playing many games against it is more so. As a result of this, I implemented a software version of the system, with an object modelling the matchbox computer.

```
def getComputerMove(self, board, playerNum, count):
    boardTuple = tuple([tuple(elem) for elem in board.getBoard()])
    mirrorTuple = self.getMirroredBoard(boardTuple)

    mirrored = mirrorTuple in self.boardStates

    if (boardTuple not in self.boardStates) and (mirrorTuple not in self.boardStates):
        #If we haven't encountered this move before
        self._openBoxes.append(boardTuple)
        moves = board.getValidMoves(playerNum)
        defaultWeight = {1:4, 3:3, 5:2, 7:1}[count]
        movesWeights = {move:defaultWeight for move in moves}
        self.boardStates[boardTuple] = movesWeights
```

```
    else:
        #If it is a normal board
        if mirrored:
            boardTuple = mirrorTuple

        self._openBoxes.append(boardTuple)
        movesWeights = self.boardStates[boardTuple]
        moves = []
        for key,value in movesWeights.items():
            moves.extend([key]*value)
```

```
    if len(moves) == 0:
        self._resigned = True
        return "Resign", "Resign"

    move = random.choice(moves)
    self._movesMade.append(move)
    if mirrored:
        return self.getMirroredMove(move)
    else:
        return move
```

As shown above, I modelled the matchboxes as a hash table with keys containing the tuple representation of the board, and the value being another hash table with keys containing the tuple representation of the move, and the value its weight.

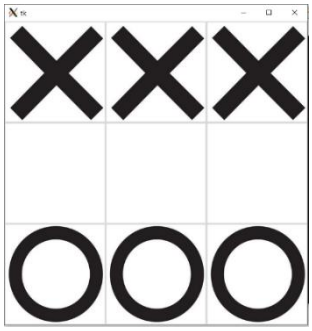
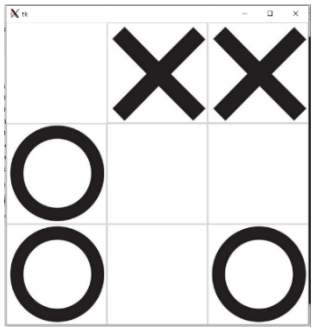
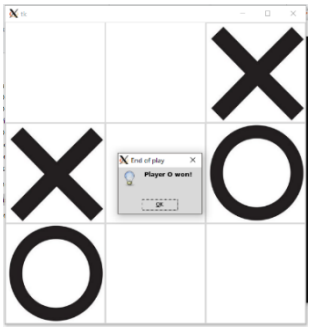
I also checked and removed isomorphically duplicate boards, as discussed before, to more closely fit the physical system, suggested by Gardner, using the below helper functions in the above code.

```
def getMirroredBoard(self, board):
    newBoard = []
    for row in board:
        newBoard.append(tuple(reversed(row)))
    return tuple(newBoard)

def getMirroredMove(self, move):
    return ((2-move[0][0], move[0][1]),(2-move[1][0], move[1][1]))
```

During training, I maintained the matchbox computer by changing weights as Gardner described following a loss. I was very careful to follow the algorithm specified in the column exactly, with the same number of starting beads, and same removal rules, to ensure the model was as accurate as possible, and hence reliable to draw conclusions from.

```
def updateWeights(self, won):
    if not won:
        index = -2 if self._resigned else -1
        self.boardStates[self._openBoxes[index]][self._movesMade[-1]] += -1
    self.resetGame()
    self.cleanUpBoardStates()

def cleanUpBoardStates(self):
    newBoardStates = {}
    for Mkey in self.boardStates.keys():
        newMovesWeights = {}
        for key, value in self.boardStates[Mkey].items():
            if value > 0:
                newMovesWeights[key] = value
        newBoardStates[Mkey] = newMovesWeights
    self.boardStates = newBoardStates
```

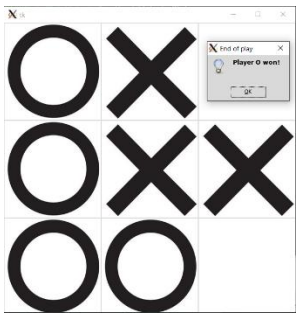
Again, I implemented a GUI to play test the AI against, as shown below, with the AI playing second and winning.

8.4 Implementing tree search algorithms to play Tic-tac-toe

I attempted to fully understand the simplest version of the minimax algorithm by implementing it to play tic-tac-toe, and hence it is a good direct comparison to similar neural network solution discussed later to the same problem.

Firstly, implemented a simple GUI for tic-tac-toe, in order to facilitate testing the algorithms:



Following this, I implemented the minimax algorithm passing the board state up a recursive call stack as a method of my game class:

```
def minimax(self, board, players, utilities=[1,-1,0,0], leafDepth=9, depth=0):
    if depth>leafDepth: #Simulate leaf nodes beyond recursion depth
        return utilities[3]

    winner = board.checkIfWon()
    if winner is not False:
        if winner == players[0]: #Win
            return utilities[0]
        elif winner == players[1]: #Loss
            return utilities[1]
        else: #Draw
            return utilities[2]

    moveWeights = {}
    for move in board.getEmptySquarePositions():
        nextBoard = TictactoeBoard()
        nextBoard.setBoard(board.getBoard())
        nextBoard.setPlayerNum(board.getPlayerNum())
        nextBoard.makeMove(move)
        nextBoard.togglePlayer()

        moveWeights[move] = self.minimax(nextBoard, players, utilities, leafDepth, depth+1)
```

```
if depth%2==0: #If it is the ai playing, play best move for the ai
    bestMove = max(moveWeights, key=lambda key: moveWeights[key])
else: #Otherwise, play the best move the other player can make (worst move for the ai)
    bestMove = min(moveWeights, key=lambda key: moveWeights[key])
bestWeight = moveWeights[bestMove]

if depth == 0:
    return bestMove
else:
    if True:
        return bestWeight
    else: #If playing against non-optimal opponent, consider:
        return sum(moveWeights.values())
```

Furthermore, I implemented a helper function to yield the optimal move given the current board state

```
def getBestMove(self):
    duplicateBoard = TictactoeBoard()
    duplicateBoard.setBoard(self.board.getBoard())
    duplicateBoard.setPlayerNum(self.board.getPlayerNum())
    x = time.time()
    val = self.minimax(duplicateBoard, duplicateBoard.getPlayers())
    print(val, round(time.time()-x, 5))
    return val
```

I then tested the algorithm thoroughly, and found that it worked incredibly effectively, and never lost against me or any other tester. However, it takes a long time (about 5 seconds) to evaluate the first move, which is a major drawback, especially since it is already only playing a simple game.

8.4.1 Further tree search algorithm techniques

The minimax algorithm can be improved by employing alpha-beta pruning, which is a heuristic which reduces search time by searching more promising paths first. The algorithm stops evaluating a possible move when a possible next move is provably worse than a previously considered move, as it cannot be a good move. When applied to a minimax search tree, it yields the same result, but much faster, since it didn't need to explore as many branches due to pruning.

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if α ≥ β then
                break (* α cut-off *)
        return value
```

```
(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)
```

Figure 28: The pseudocode for minimax with alpha-beta pruning [58]

I also updated my minimax code to increase speed using alpha-beta pruning:

```
def minimax(self, board, players, utilities=[1,-1,0,0], alpha=-math.inf, beta=math.inf, leafDepth=9, depth=0):
    if depth>leafDepth: #Simulate leaf nodes beyond recursion depth
        return utilities[3]

    if depth==0:
        maxValue, bestMove = -math.inf, None
        for move in board.getEmptySquarePositions():
            nextBoard = TictactoeBoard()
            nextBoard.setBoard(board.getBoard())
            nextBoard.setPlayerNum(board.getPlayerNum())
            nextBoard.makeMove(move)
            nextBoard.togglePlayer()

            value = self.minimax(nextBoard, players, utilities, alpha, beta, leafDepth, depth+1)
            if value > maxValue:
                bestMove, maxValue = move, value

        return bestMove
```

```
    winner = board.checkIfWon()
    if winner is not False:
        if winner == players[0]: #Win
            return utilities[0]
        elif winner == players[1]: #Loss
            return utilities[1]
        else: #Draw
            return utilities[2]

    if board.getCurrentPlayer() == players[0]: #Maximising layer
        value = -math.inf
        for move in board.getEmptySquarePositions():
            nextBoard = TictactoeBoard()
            nextBoard.setBoard(board.getBoard())
            nextBoard.setPlayerNum(board.getPlayerNum())
            nextBoard.makeMove(move)
            nextBoard.togglePlayer()

            func = self.minimax(nextBoard, players, utilities, alpha, beta, leafDepth, depth+1)
```

```
            value = max(value, func)
            alpha = max(alpha, value)
            if depth==0: print(move, alpha, beta, 1)
            if alpha >= beta:
                if depth==0: print("Hit 1")
                break

    else:
        value = math.inf
        for move in board.getEmptySquarePositions():
            nextBoard = TictactoeBoard()
            nextBoard.setBoard(board.getBoard())
            nextBoard.setPlayerNum(board.getPlayerNum())
            nextBoard.makeMove(move)
            nextBoard.togglePlayer()

            func = self.minimax(nextBoard, players, utilities, alpha, beta, leafDepth, depth+1)

            value = min(value, func)
            beta = min(beta, value)
            #if depth==1: print(move, alpha, beta, 2)
            if alpha >= beta:
                #if depth==1: print("Hit 2")
                break
    if depth == 0:
        return move
    else:
        return value
```

And again added a helper function to yield the best move for a given board position:

```
def getBestMove(self):
    duplicateBoard = TictactoeBoard()
    duplicateBoard.setBoard(self.board.getBoard())
    duplicateBoard.setPlayerNum(self.board.getPlayerNum())
    x = time.time()
    val = self.minimax(duplicateBoard, duplicateBoard.getPlayers())
    print(val, round(time.time()-x, 5))
    #print(val)
    return val
```

It is notable that whilst alpha-beta pruning greatly increases the speed of traversal, it also requires much more source code to be implemented.

Throughout my essay, I mention the Monte Carlo Heuristic Search Tree algorithm (MCTS). This is a more advanced approach, which employs a Monte Carlo style random algorithm (a randomized algorithm whose output may be incorrect, however, typically with a low probability, and significant speed benefits) to search the tree, often in combination with

uncertainty bounds. This means that it doesn't search every board state, but it searches more promising branches earlier, making it effective for complex games, such as Chess and Go.

8.5 Implementing first principles neural networks to play Tic-tac-toe

I attempted to fully understand neural networks as applied to games by implementing a program to play tic-tac-toe, and hence it is a good direct comparison to similar analytical solution discussed later to the same problem. Throughout the process, I exclusively used the first principles network I developed, as detailed above.

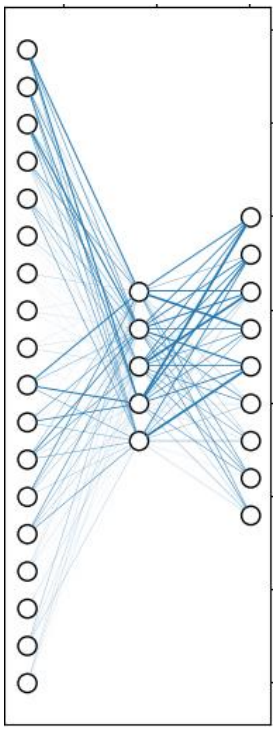
I eventually tried two approaches which both worked well: A genetic based algorithm, and a back-propagation algorithm which trained on a set of good moves generated by an optimal player.

The genetic algorithm produced started with a large pool of initial weights, and played them against a random opponent. If the weights lost the round, they were removed from the pool. At a threshold, I randomly "bred" weights together, selecting a small number, then taking the average of them, then added them to the pool.

```
def breedNetworks(nns, percent=100):
    #Breed the networks together, by randomly taking data from each of their parents
    #Somewhat analogous to "crossing over" in biological genetic systems
    def select(a, b, c, d):
        for i in range(0, len(a)):
            if type(a[i]) is list:
                z = select(a[i], b[i], [], d+1)
                c.append(z)
            else:
                c.append( random.choice([a[i]+b[i]]) )
        return c

    nnWeights = [[nn.W1, nn.W2] for nn in nns]
    breedingPairs = list(itertools.combinations(nnWeights, 2))
    random.shuffle(breedingPairs)
    breedingPairs = breedingPairs[:int( (percent/100)*len(breedingPairs) )]

    bredWeights = [select(b[0], b[1], [], 1) for b in breedingPairs]
    nns = [NeuralNetwork(*sizes, weights=bredWeights[i]) for i in range(len(bredWeights))]
    return nns
```

I then repeated these steps until the pool didn't change size 10 consecutive times, i.e. no games had been lost.

The back-propagation algorithm trained on an input and target output set generated from an alpha-beta pruned tree search, which always gave the correct move given the board state. I trained this until it made the correct move close to every time, then that set of weights was my trained network.



It is important to consider what is a good input and output schema to train the network on.

Initially I used a 9 node input layer, with empty squares taking the value zero, "X"s one, and "O"s minus one. However, I found that the sigmoid activation function I was using stripped out the sign of the value, so it couldn't train.

I then moved to an 18-node input layer, with the first 9 nodes taking the value of one if the square held the AI player, otherwise zero, and another nine nodes doing the same thing the opposing player.

Then, I used a 9-node output layer, with the highest node value indicating where to place the cross worked well.

This larger input data set helped the network avoid underfitting.

```
def aiNeuralMove(self, nn):
    #Use the neural network to generate move
    iL = np.array(self.board, dtype=str).flatten()
    X = np.array([0 if x=="X" else 1 for x in iL]+[0 if x=="O" else 1 for x in iL])
    yHat = nn.forwardpropagate(X)
    print(yHat)
    while True:
        indexMax = np.argmax(yHat)
        move = [indexMax%3, indexMax//3]
        if self.isMoveValid(move):
            return move, X, yHat, indexMax
        else:
            yHat[indexMax] = 0
```

I then trained the networks, changing the weights millions of times, in order to find a functional combination.

I also experienced issues in network architecture, and found that small networks could not describe the input set appropriately, and so I used a large 18 x 30 x 30 x 9 network in my final testing.

The full code can be found on GitHub, and both attempts have many optimisations and techniques employed to make them work, but there are too many to explicitly enumerate her.

Both take a very long time to train (several orders of magnitude higher than analytical techniques playing the same game), and owing to the random nature of their training,

sometimes end up with flaws, occasionally losing games. However, for more complex games, where enumerating the entire game tree is not viable, the more heuristic approach of neural networks becomes more beneficial.

Finally, since Tic-tac-toe is such a simple game with such a small search space, it is possible to visually represent every winning move for the first player in a lookup table, which is significantly simple than both neural networks and tree-search algorithms as discussed above, but not applicable to more complex games:

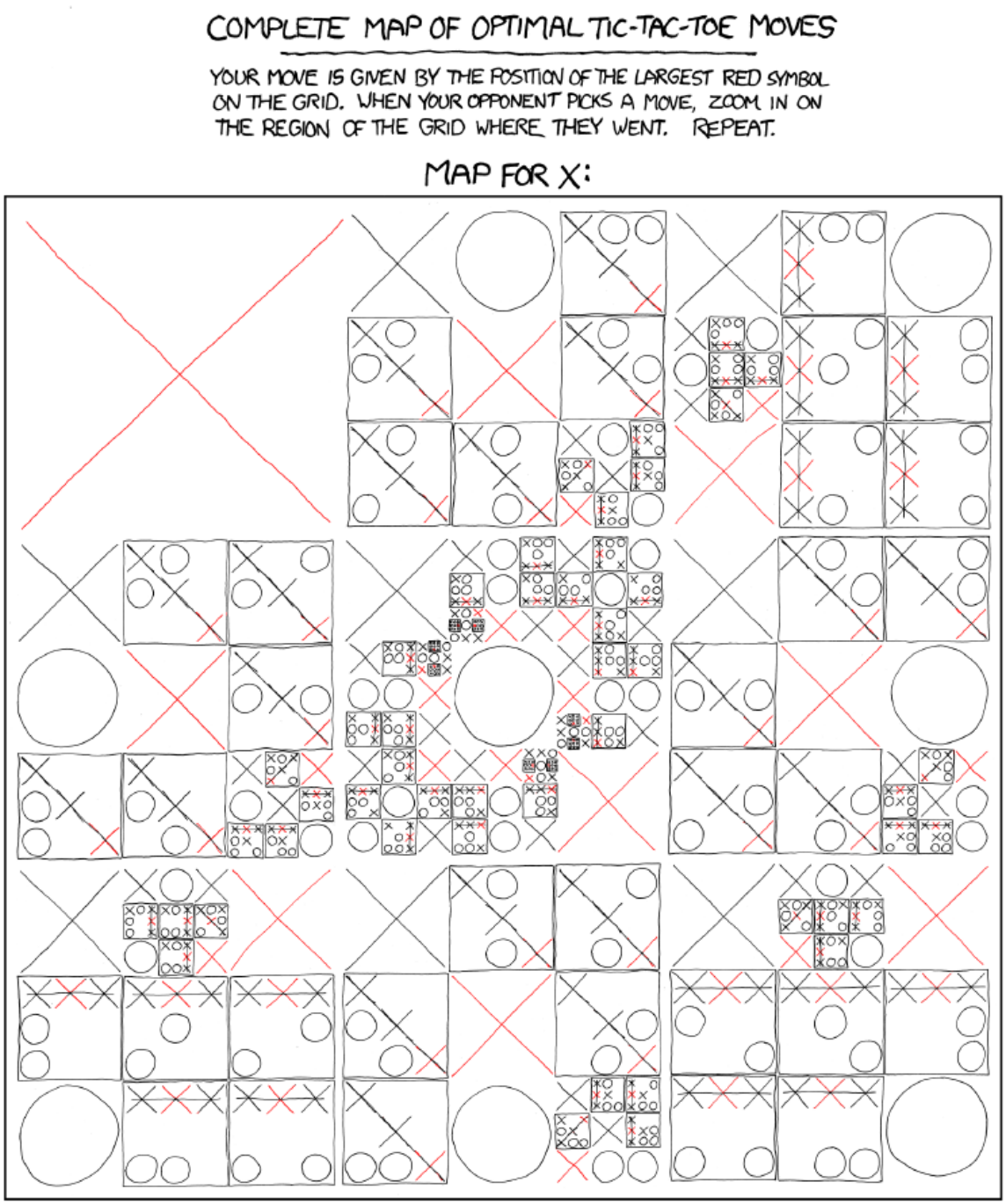


Figure 29: A lookup table for every winning move for the first player in Tic-tac-toe, an XKCD comic by Randall Munroe [59]

8.6 Further neural network techniques

I have described the basic principles of neural networks, and how they are trained. It is tempting to leave it there, and merely throw more data and processing power at a problem, which it should be noted many researchers have been known to do. However, there are a wide array of techniques which can be used to increase the efficacy of networks, and make them more applicable to different problem sets. It is almost always better to consider these techniques, many of which are not especially complicated.

8.6.1 Overfitting and underfitting

The solution to underfitting is relatively trivial, a different, larger network architecture. However, it can be difficult to identify if a network is underfitting, as the only sign is it not getting better, which could be symptomatic of many different issues, and sometimes it is not feasible to increase the network size, as it means more computation is required per epoch.
The problem of overfitting is that an overfitted network will only be correct for its training data, and hence will not yield a general solution to the problem, as required. It is also more difficult to address, and is already somewhat ill-defined, as there is not a clear line denoting when overfitting starts, and all networks will overfit to some extent, by virtue of the mechanism by which they are trained.

There are an immense number of techniques that can be used to combat overfitting and underfitting, along with other techniques to improve the speed and efficiency of training, and different architectures that have proven to be useful for specific classes of problems. Some of them are enumerated below:

8.6.2 Regularisation

Regularisation is a technique used to mitigate overfitting. It does this by taking note of the fact that if a network overfits, it will end up with large differences in value between individual weights. This means that overfitting can be mitigated by also penalising a network for having very different weights. This is often implemented with a “Lambda” hyperparameter, which denotes how much the network is penalised for weight disparity:

```
def updateWeights(self, X, learnRate):
    """Update the weights of the network based on the previous pass of back
    propagation in order to improve the network model
    """

    #Update the hidden layer weights
    j = 0
    for i in reversed(range(len(self.W2))):
        currDelta = np.dot(self.a[i].T, self.delta[j])
        currDelta -= (self.Lambda*self.W2[i]) #Regularisation - penalise large weight values
        currDelta *= learnRate #Learning rate

        self.W2[i] += currDelta
        j += 1
```

```
    #Update the input layer weights
    currDelta = np.dot(X.T, self.delta[j])
    currDelta -= (self.Lambda*self.W1) #Regularisation - penalise large weight values
    currDelta *= learnRate #Learning rate  - possible implement momentum

    self.W1 += currDelta
```

8.6.3 Hinton's dropout

A second solution to this is called Hinton's dropout [60], and is remarkably simple in its function, and is a relatively new discovery. During the forward propagation step of the training process, a small proportion of hidden nodes are randomly turned off. This effectively introduces random noise into the dataset, so irrespective of the number of training passes, the network will not overfit as much. Again, this can be implemented with a metaparameter giving the fraction of hidden nodes that are dropped.

```
def doDropout(self, layer, weight):
    #https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf
    if weight!=0:
        layer *= np.random.binomial([np.ones((len(layer),len(layer[0])))],1-weight)[0] * (1.0/(1-weight))
    return layer
```

8.6.4 Pruning

Another technique that can be used to improve the efficiency of training a network is pruning [61]. Pruning is the process of removing the least significant node in a network, in order to increase training speeds, without reducing the effectiveness of a network. If a smaller network can achieve the same accuracy of output of a larger network, it should be preferred, and pruning provides a way to reduce the size of a network architecture, without compromising the model's accuracy.

8.6.5 Stochastic & Batch processing

There are two main modes of learning for neural networks, stochastic and batch processing. In stochastic learning, the weights are updated after each training example, based on only that error, whereas in batch learning the errors are accumulated over a set of training examples, and the weights are updated at the end. Stochastic learning introduces "noise" into the descent, with each individual update going in slightly different directions, whereas batch learning tends to result in a less "noisy" descent, as the change in weights are averaged over the entire batch, however, repeatedly training on the same batch can lead to negative effects. A common technique is to use mini-batches, randomly selected sets of input data from the training set, back-propagated together, which has been shown to provide good training characteristics.

```
#Generate minibatches to train the network with
combinedData = [(X[i].tolist(), y[i].tolist()) for i in range(len(X)-1)]
random.shuffle(combinedData)
batchX = np.array([d[0] for d in combinedData[:self.batchSize]])
batchY = np.array([d[1] for d in combinedData[:self.batchSize]])

yHat = self.nn.forwardpropagate(batchX)
self.nn.backwardpropagate(batchX, batchY, yHat)
self.nn.updateWeights(batchX, self.learnRate)

trainError = self.getMSError(batchY, yHat)
testError = self.getMSError(testY, testYHat)
```

8.6.6 Learning rates

Learning rates are hyperparameters that linearly scale the rate of change of the weights in a network. If the learning rate is too high, the weights change very quickly, and so can "jump" over a minimum, however, if the learning rate is too low, the network will train very slowly. There are techniques which dynamically change the learning rate throughout training to improve the rate of training, such as momentum.

8.6.7 Weight picking

Initialising networks with good weights can influence the training of the network vastly. Weights are generally picked to be small and centred about zero. A common technique for initialisation, called Xavier's initialisation is to generate a normally distributed random sample of weights with mean zero and standard deviation one, then multiply the sample by the square root of the reciprocal of the number of inputs to a given layer [62] [63].

```
return np.round(np.random.randn(xDim,yDim),r)*(1/(numIn+numOut))**0.5
```

8.6.8 Activation functions

The logistic sigmoid function discussed in the neural networks section is a very commonly used activation function. However, it does have some drawbacks, including the "vanishing gradients" problem, where in the near horizontal sections of the function, i.e. very large magnitude input values, the gradient in backpropagation is very small, so it trains slowly [64].

There are many activation functions other than the logistic sigmoid curve used in neural networks. Some of the more common ones, the majority of which are detailed in "Information theory, inference, and learning algorithms", by David MacKay [20] include:

Linear activation

$$a(x) = x$$

This function initially seems useful, however doesn't function effectively at all. This can be explained in two ways: One is that any sequence of nodes can be expressed as a single node, so deep networks will not be more effective [21]. The second is that the derivative is constant, as $\frac{d}{dx}\big(a(x)\big) = 1$, so backpropagation, which relies on differentiation, cannot be employed.

Tanh Sigmoid

$$a(x) = \tanh(x) = 2\,sigmoid(x) - 1$$

This is a similar activation function to logistic sigmoid, however it is stretched and translated, in order to allow negative values to input nodes, and is hence preferable in some use cases.

Threshold/Step/Heaviside function

$$a(x) = \Theta(x) = \begin{cases} 1, x > 0 \\ -1, x > 0 \end{cases} = \frac{d}{dx}\left(\begin{cases} x, x > 0 \\ 0, x > 0 \end{cases}\right)$$

Often used in final layers of binary classifiers, as it will yield an output of the correct (boolean) data type, as opposed to other functions which yield analogue (floating point) values, so are less suited for use in binary classifiers, as a rounding or a ceiling function would be required.

ReLu (Rectified linear unit)

$$a(x) = \begin{cases} x, x > 0 \\ 0, x > 0 \end{cases}$$

The ReLu is non-linear, so it doesn't have some of the problems of a linear activation function, e.g. it can be used with back-propagation. However, it is not bounded, so it can still explode in size. Furthermore, it increases the efficiency of a network being improving a property called sparsity. In other functions, such as sigmoid, every input value yields an analogue output value, whereas ReLu yields a binary 0 about 50% of the time, so it is much less computationally expensive to evaluate it.

However, this binary portion means that the gradient becomes zero, so neurons stop responding to backpropagation. This is called the dying ReLu problem, where some nodes stop responding, and hence reduce the efficiency of the network. To fix this, a "Leaky" ReLu can be employed:

$$a(x) = \begin{cases} x, x > 0 \\ 0.01x, x > 0 \end{cases}$$

Which means neurons don't die. Finally, ReLu can be faster than sigmoid or tanh, since it involves simpler functions – which makes it preferable in the context of some deep networks.

8.6.9 Convolution, deep and recurrent

There are many modern neural network architectures which have been developed to be applied to specific problem sets.

Deep neural networks are an example of this, when many hidden layers are used, which allows for the network to model very complex systems. They are the most common approach to general problems, as they tend to work well, but don't have overly specific optimisations, such as those of CNNs and RNNs. However, they often take a long time to train.

There are also more drastic deviations from the initial basis of neural networks, for example convolutional neural networks, CNNs, and recurrent neural networks, RNNs.

Simple neural networks tend not to work very effectively on large data sets of images. For example, a common test for the effectiveness of a neural network is the MNIST data set, which contains 60,000 28x28 grayscale images of handwritten digits. However, in order to input this into a neural network, a 784-node input layer is required, along with several hidden layers, which results in a large network that is very hard to train. Convolutional neural networks help mitigate this problem by having an additional special type of layer bridging between the input layer and the hidden layers, which reduces the amount of processing which needs to be done. It does this by pre-processing the image, and identifying its features, reducing the amount of information going into the main hidden layers. This process has the additional benefit of making the network more resistant to features displaced in the image, which neural networks alone don't handle very well.

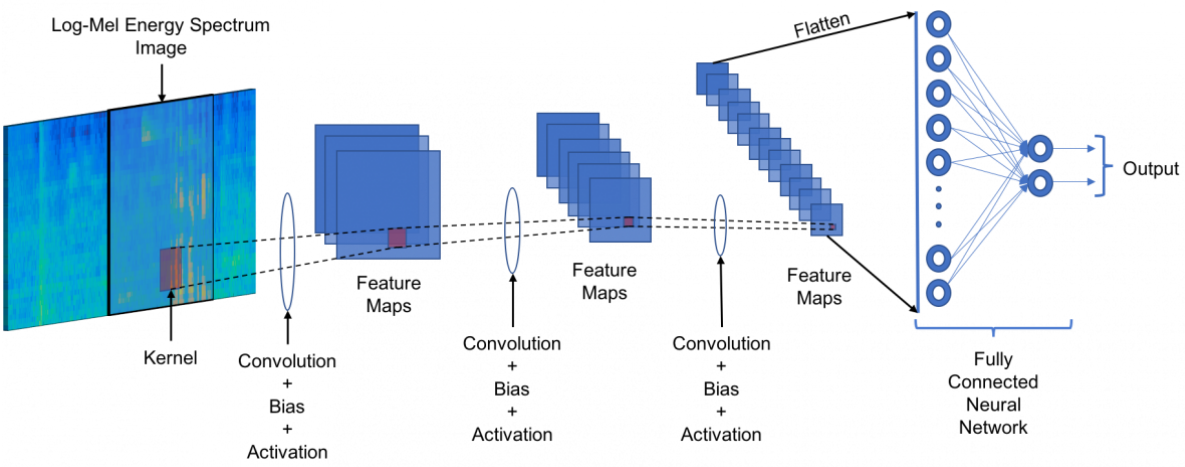


Figure 30: A diagram of the layers comprising a convolutional neural network [65]

Kernel convolution is the process of passing a matrix, known as a kernel over an input signal, in order to transform it into a feature map, dependent on the kernel. The output signal of an individual pixel is the sum of the products of the kernel weight and its corresponding input square, when the kernel is centred on the target pixel.

$$C[x,y] = \sum_i \sum_j A[i,j] \times B[x-i, y-j]$$

Where A is the input signal, B is the kernel, and C is the output signal.

This idea of employing feature maps as a pre-processing layer in deep learning was first published in 1990 by Yann LeCunn "a founding father of convolutional nets" et al. [66], and was then popularised as a viable technique by Krizhevsky, Sutskever and Hinton following the resounding success of the AlexNet entry into the 2010 ImageNet contest [67].

Traditional neural networks cannot handle real time data, as they would not be able to train using backpropagation. However, being able to process real time data would be very useful. The main approach to this problem is to employ recurrent networks, where later layers feed back into earlier ones. This allows the processing of a real time stream of data, and can also simulate memory. This is a complicated issue, which was initially considered intractable, and is best addressed in Ilya Sutskever's seminal thesis [68].

9 Glossary

This area of research includes many pieces of subject specific technical vocabulary, that is either not in normal use, or has a subtly different meaning to its traditional use in the vernacular, as a result of this, below is a glossary of terms to allow improved specificity of language, and hence reduce unnecessary additional wording.

Term	**Definition**
Games	A physical or mental competition conducted according to rules with the participants in direct opposition to each other
Abstract games	A specific type of game, in which the theme is unimportant to the experience of gameplay, and hence can be usefully analysed by computer algorithms.
Activation function	A non-linear function used in a node in a neural network
Algorithm	A process or set of rules to be followed in calculations or other problem-solving operations
Alpha-beta pruning	A heuristic improvement to minimax that doesn't explore fewer promising nodes, hence increasing speed
Analytical technique	A technique produced by human analysis of a problem
Backpropagation	The process of finding the amount to change the weights by to improve a network
Classification	A task involving dividing a dataset into classes based on properties of the data in a supervised manner
Clustering	A task involving partitioning a dataset into groups in an unsupervised manner
Convolutional neural networks	(CNNs) neural networks with specialised convolutional layers to improve image processing
Deep neural networks	(DNNs) neural networks with many hidden layers, to allow modelling of complex data sets
Epoch	A full cycle of forward propagation followed by back propagation
Forward propagation	The process passing data through a network to yield an output

Game	A physical or mental competition conducted according to rules with the participants in direct opposition to each other
Game theory	The study of mathematical models of strategic interaction between rational decision-makers
Game tree	A directed graph, whose nodes are game states, and whose edges are moves
Genetic algorithms	Algorithms designed to optimise a model, based on the way that biological systems do, namely genetics and evolution.
Impartial play	A type of game where valid moves depend only on position, not which player is moving
Machine learning	An application of artificial intelligence that allows systems to automatically learn and improve from experience without being explicitly programmed
Meta-parameters	Properties of a neural network which are fixed during runtime, e.g. network architecture
Minimax	An algorithm that finds optimal moves for a player in a two-player game by traversing the game tree
Misère play	A type of a game where the last person to play a move loses
Misère quotient	Mathematical structures (commutative monoids) that encode the additive structure of specific misère-play games
Monotonic	A function that is either entirely increasing or entirely decreasing across its entire domain
Monte Carlo Heuristic Tree Search	(MCTS) an advanced game tree search algorithm that employs a weighted random traversal to increase speed in finding optimal moves
Network architecture	The size and structure of a neural network, including number hidden nodes, types of connections, types of activation functions, etc.
Neural network	A prominent machine learning technique modelled on the human brain
Non-isomorphic	Objects that are not structurally equivalent, i.e. are not isomers of each other, for example an impartial game board mirrored in a line of symmetry. (A is isomorphic to B is denoted by $A \cong B$)
Non-linear	A function in which a change in the input isn't proportional to a change in the output
NumPy	A Python library for scientific computing, specifically with support for efficient matrix operations
Overfitting	The problem of a neural network fitting to a dataset so closely it loses applicability to the general problem
Perceptron/Node/Neuron	A single unit used to compose a neural network
Recurrent neural networks	(RNNs) neural networks which link back into themselves to allow simulated memory, and real time processing
Regression	A problem involving predicting numerical data given input data, based on its properties in a supervised manner
Search space	The set of all possible points within a set of constraints
Tensor processing units	(TPUs), a specialised piece of computer hardware, often used to train neural networks
TensorFlow	A symbolic mathematics library that is most commonly used for machine learning, such as neural networks
Underfitting	The problem of a neural network not fitting to a data set closely enough
Weights	Properties of a neural network that change during training, encoding the model

10 References

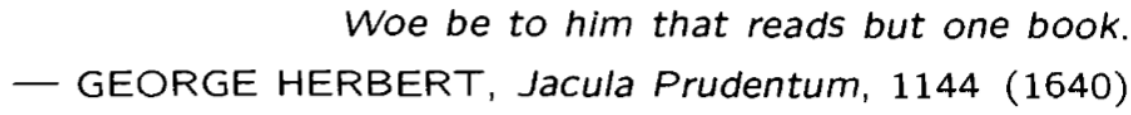


Figure 31: Donald Knuth: The Art of Programming Volume 1: Fundamental Algorithms, page xiv [41]

10.1 Citations

[1] Merriam-Webster, "Game | Definition of Game by Merriam-Webster," [Online]. Available: https://www.merriam-webster.com/dictionary/game. [Accessed 30th April 2019].

[2] Astral Castle, "A History of Board Games," 5th December 2003. [Online]. Available: https://web.archive.org/web/20031205225710/http://www.astralcastle.com/games/index.htm. [Accessed 30th April 2019].

[3] TechCrunch, "Video game revenue tops $43 billion in 2018, and 18% jump from 2017 | TechCrunch," [Online]. Available: https://techcrunch.com/2019/01/22/video-game-revenue-tops-43-billion-in-2018-an-18-jump-from-2017/. [Accessed 30th April 2019].

[4] Wikipedia Contributors, "British Museum Royal Game of Ur - Royal Game of Ur - Wikipedia," 24th June 2010. [Online]. Available: https://en.wikipedia.org/wiki/Royal_Game_of_Ur#/media/File:British_Museum_Royal_Game_of_Ur.jpg. [Accessed 20th May 2019].

[5] M. J. Thompson, "Defining the Abstract," The Games Journal, July 2000. [Online]. Available: http://www.thegamesjournal.com/articles/DefiningtheAbstract.shtml. [Accessed May 25th 2019].

[6] K. Jameson, "Game Theory and its Applications," in *2014 Sr. Seraphim Gibbons Undergraduate Symposium*, Minneapolis, 2013.

[7] A. Rachnog, "Neural networks for algorithmic trading. Correct time series forecasting + backtesting," A Medium Corporation, 11th May 2017. [Online]. Available: https://medium.com/@alexrachnog/neural-networks-for-algorithmic-trading-1-2-correct-time-series-forecasting-backtesting-9776bfd9e589. [Accessed 30th April 2019].

[8] Y. LeCun, C. Cortes and C. Burges, "MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges," [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed 10th May 2019].

[9] J. Brownlee, "Machine Learning is Popular Right Now," machinelearningmastery, [Online]. Available: https://machinelearningmastery.com/machine-learning-is-popular/. [Accessed 30th April 2019].

[10] Daffodil Software, "9 Applications of Machine Learning from Day-to-Day Life," medium.com, 31st July 2017. [Online]. Available: https://medium.com/app-affairs/9-applications-of-machine-learning-from-day-to-day-life-112a47a429d0. [Accessed 2nd May 2019].

[11] Nature, "Machine learning - Latest research and news | Nature," [Online]. Available: https://www.nature.com/subjects/machine-learning. [Accessed 2nd May 2019].

[12] Nuffield Council on Bioethics, "Artificial intelligence (AI) in healthcare and research," May 2018. [Online]. Available: http://nuffieldbioethics.org/wp-content/uploads/Artificial-Intelligence-AI-in-healthcare-and-research.pdf. [Accessed 2nd May 2019].

[13] R. Munroe, "xkcd: Machine Learning," 17th May 2017. [Online]. Available: https://imgs.xkcd.com/comics/machine_learning.png. [Accessed 10th May 2019].

[14] M. Cilimkovic, "Neural Networks and Back Propagation Algorithm," dataminingmasters, [Online]. Available: http://dataminingmasters.com/uploads/studentProjects/NeuralNetworks.pdf. [Accessed 1st May 2019].

[15] ThoughtCo, "neuron-anatomy," 10th September 2018. [Online]. Available: https://www.thoughtco.com/thmb/wBA889I01ThIHWIjCHAeimx7Q7Y=/1500x0/filters:no_upscale():max_bytes(150000):strip_icc()/neuron-anatomy-58530ffe3df78ce2c34a7350.jpg. [Accessed 2nd May 2019].

[16] R. Bailey, "Human Biology: Neurons and Nerve Impulses," ThoughtCo., 10th September 2019. [Online]. Available: https://www.thoughtco.com/neurons-373486. [Accessed 2nd May 2019].

[17] K. Cherry, "How Many Neurons Are in the Brain?," verywellmind.com, 7th May 2019. [Online]. Available: https://www.verywellmind.com/how-many-neurons-are-in-the-brain-2794889. [Accessed 9th May 2019].

[18] F. Rosenblatt, "The perceptron a perveiving and recognizing automaton," Cornell Aeronautical Laboratory, 1957.

[19] F. Rosenblatt, "The Perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review,* vol. 65, no. 6, pp. 386-408, 1958.

[20] D. J. C. MacKay, "Activity rule (a) ii. Sigmoid (logistic function)," in *Information Theory, Inference, and Learning Algorithms,* Cambridge, Cambridge University Press, 2003, p. 471.

[21] G. Cybenko, "Approximations by Superpositions of a Sigmoidal Function," *Mathematics of Control, Signals, and Systems,* vol. 2, pp. 303-314, 1989.

[22] M. A. Nielsen, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/images/tikz41.png. [Accessed 2nd May 2019].

[23] R. Rojas, "Neural Networks - A Systematic Introduction," in *Neural Networks - A Systematic Introduction*, Springer-Verlag, Berlin, New-York, 1996, p. 476.

[24] M. Minsky, Neural Nets and the Brain: Model Problem, Princeton: Unpublished Dissertation, Princeton University, 1954.

[25] Y. LeCun, "A Theoretical Framework for Back-Propagation," University of Toronto, Toronto.

[26] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning representations by back-propagating errors," *Nature,* vol. 323, pp. 533-536, 1986.

[27] M. A. Nielsen, "Chapter 2: How the backpropagation algorithm works," in *Neural Networks and Deep Learning,* Determination Press, 2015.

[28] S. Shamdasani, "Build a flexible Neural Network with Backpropagation in Python - DEV Community," dev.to, 7th August 2017. [Online]. Available: https://dev.to/shamdasani/build-a-flexible-neural-network-with-backpropagation-in-python. [Accessed 20th May 2019].

[29] B. Klein, "Machine Learning with Python: Backpropagation in Neural Network," python-course.eu, [Online]. Available: https://www.python-course.eu/neural_networks_backpropagation.php. [Accessed 20th May 2019].

[30] Tensorflow, “A Neural Network Playground,” tensorflow.org, [Online]. Available: https://playground.tensorflow.org/. [Accessed 10th May 2019].

[31] M. Cook, “It Takes Two Neurons To Ride a Bicycle,” http://citeseerx.ist.psu.edu, Pasadena, CA, 2004.

[32] C. L. Bouton, “Nim, A Game with a Complete Mathematical Theory.,” *Annals of Mathematics; Second Series,* vol. 3, no. 1/4, pp. 35-39, 1901-1902.

[33] Nrich, “Nim-like Games : nrich.maths.org,” nrich.maths.org, [Online]. Available: https://nrich.maths.org/1209. [Accessed 2nd May 2019].

[34] Grey Matters: Mental Gym, “Grey Matters: Blog: Secrets of Nim (Notakto),” http://gmmentalgym.blogspot.com/, 12th April 2012. [Online]. Available: http://headinside.blogspot.com/2012/04/secrets-of-nim-notakto.html. [Accessed 9th June 2019].

[35] “combinatorial game theory - Neutral tic tac toe - MathOverflow,” MathOverflow, [Online]. Available: https://mathoverflow.net/questions/24693/neutral-tic-tac-toe. [Accessed 1st May 2019].

[36] T. E. Plambeck and G. Whitehead, “The Secrets of Notakto: Winning at X-only Tic-Tac-Toe,” *arXiv,* 8th January 2013.

[37] M. Gardner, “How to build a game-learning machine and then teach it to play and to win,” *The Scientific American,* March 1962.

[38] M. Gardner, “A Matchbox Game-Learning Machine,” gwern.net, [Online]. Available: https://www.gwern.net/docs/rl/1991-gardner-ch8amatchboxgamelearningmachine.pdf. [Accessed 10th May 2019].

[39] Erik973, “Matchbox Mini Chess Learning Machine: 3 Steps (with Pictures),” Instructables.com, [Online]. Available: https://www.instructables.com/id/Matchbox-Mini-Chess-Learning-Machine/. [Accessed 7th June 2019].

[40] Wikipedia Contributors;, “Tic-tac-toe-game-tree - Game Tree - Wikipedia,” 1st April 2007. [Online]. Available: https://en.wikipedia.org/wiki/Game_tree#/media/File:Tic-tac-toe-game-tree.svg. [Accessed 1st May 2019].

[41] D. Knuth, “Trees,” in *The art of computer programming : fundamental algorithms,* Addison-Wesley, 1997, p. 308.

[42] Wikipedia Contributors;, “Minimax,” Wikipedia, The Free Encyclopedia., 1st May 2019. [Online]. Available: https://en.wikipedia.org/wiki/Minimax#Pseudocode. [Accessed 1st May 2019].

[43] J. Schaeffer, “Chinook,” 2nd August 2004. [Online]. Available: https://web.archive.org/web/20040930164251/http://www.cs.ualberta.ca/~chinook/. [Accessed 6th May 2019].

[44] J. Schaeffer, “Marion Tinsley: Human Perfection at Checkers?,” wylliedraughts.com, [Online]. Available: http://www.wylliedraughts.com/Tinsley.htm. [Accessed 6th May 2019].

[45] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu and S. Sutphen, “Checkers Is Solved,” *Science,* vol. 317, pp. 1518-1522, 2007.

[46] “AlphaGo Zero: Learning from scratch,” DeepMind, 19th October 2017. [Online]. Available: https://deepmind.com/blog/AlphaGo-zero-learning-scratch/. [Accessed 30th April 2019].

[47] D. Silver, J. Schrittwieser, I. Antonoglou, K. Simonyan, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre and G. van den Driessche, “Mastering the game of Go without human knowledge,” *Nature,* vol. 550, pp. 354-359, 2017.

[48] C. Metz, “In Two Moves, AlphaGo and Lee Sedol Redefined the Future | Wired,” Wired, 16th March 2016. [Online]. Available: https://www.wired.com/2016/03/two-moves-AlphaGo-lee-sedol-redefined-future/. [Accessed 30th April 2019].

[49] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan and D. Hassabis, “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm,” *arXiv,* 5th December 2017.

[50] D. R. Hofstadter, “Chunking and Chess Skill,” in *Gödel, Escher, Bach: an Eternal Golden Braid*, The Harvester Press, 1979, p. 286.

[51] S. Gibbs, “AlphaZero AI beats champion chess program after teaching itself in four hours | The Guardian,” The Guardian, 7th December 2017. [Online]. Available: https://www.theguardian.com/technology/2017/dec/07/alphazero-google-deepmind-ai-beats-champion-program-teaching-itself-to-play-four-hours. [Accessed 1st May 2019].

[52] BBC News, “Google's 'superhuman' DeepMind AI claims chess crown - BBC News,” 6th December 2017. [Online]. Available: https://www.bbc.co.uk/news/technology-42251535. [Accessed 1st May 2019].

[53] J. Osborne, “Google's Tensor Processing Unit explained: this is what the future of computing looks like | TechRadar,” TechRadar, 22nd August 2016. [Online]. Available: https://www.techradar.com/news/computing-components/processors/google-s-tensor-processing-unit-explained-this-is-what-the-future-of-computing-looks-like-1326915. [Accessed 1st May 2019].

[54] PeterDoggers, “AlphaZero Chess: Reactions From Top GMs, Stockfish Author,” chess.com, 5th October 2018. [Online]. Available: https://www.chess.com/news/view/alphazero-reactions-from-top-gms-stockfish-author. [Accessed 1st May 2019].

[55] Chessdom, “Komodo MCTS (Monte Carlo Tree Search) is the new star of TCEC | Chessdom,” chessdom.com, 18th December 2018. [Online]. Available: http://www.chessdom.com/komodo-mcts-monte-carlo-tree-search-is-the-new-star-of-tcec/. [Accessed 1st May 2019].

[56] G. Van Rossum, B. Warsaw and N. Coghlan, “PEP 8 -- Style Guide for Python Code,” The Python Software Foundation, 5th July 2001. [Online]. Available: https://www.python.org/dev/peps/pep-0008/. [Accessed 30th April 2019].

[57] D. Dua, C. Graff and R. Fisher, “UCI Machine Learning Repository,” 2019.

[58] Wikipedia Contributors, “Alpha–beta pruning,” Wikipedia, The Free Encyclopedia., 26th April 2019. [Online]. Available: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning#Pseudocode. [Accessed 10th May 2019].

[59] R. Munroe, “xkcd: Tic-Tac-Toe,” 10th December 2010. [Online]. Available: https://imgs.xkcd.com/comics/tic_tac_toe.png. [Accessed 7th June 2019].

[60] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks fromOverfitting,” *Journal of Machine Learning Research,* vol. 15, pp. 1929-1958, 2014.

[61] E. J. Crowley, J. Turner, A. Storkey and M. O'Boyle, “Pruning neural networks: is it time to nip it in the bud?,” *arXiv,* 19th January 2019.

[62] R. Vasudev, “How to Initialize weights in a neural net so it performs well?,” Hackernoon, 29th May 2018. [Online]. Available: https://hackernoon.com/how-to-initialize-weights-in-a-neural-net-so-it-performs-well-3e9302d4490f. [Accessed 10th May 2019].

[63] M. Thoma, *Analysis and Optimization of Convolutional Neural Network Architectures,* 2017.

[64] S. V. Avinash, "Understanding Activation Functions in Neural Networks," medium.com, 30th March 2017. [Online]. Available: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0. [Accessed 6th May 2019].

[65] IEEE Access, "A Convolutional Neural Network Smartphone App for Real-Time Voice Activity Detection - IEEE AccessIEEE Access," [Online]. Available: https://ieeeaccess.ieee.org/wp-content/uploads/2018/04/Sehgal-GraphicalAbstract-2-1200x480.png. [Accessed 16th June 2019].

[66] Y. LeCun, O. Matan, B. Boser, J. S. Denker, D. Henderson, H. R. E, W. Hubbard, L. D. Jackel and H. S. Baird, "Handwritten Zip Code Recognition with Multilayer Networks," AT&T Bell Laboratories, Holmdel, New Jersey, 1990.

[67] A. Krizhevsky, I. Sutskever and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *NIPS 2012: Neural Information Processing Systems*, Lake Tahoe, Nevada, 2012.

[68] I. Sutskever, *Training Recurrent Neural Networks,* Toronto, 2013.

[69] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25 (NIPS 2012)*, 2012.

[70] E. R. Berlekamp, J. H. Conway and R. K. Guy, "Green Hackenbush, The Game of Nim, and Nimbers & Getting Nimble with Nimbers," in *Winning Ways for Your Mathematical Plays, Volume 1*, Wellesley, MA, AK Petes Ltd, 2001, pp. 40-43.

[71] ChessBazaar, [Online]. Available: http://www.chessbazaar.com/blog/wp-content/uploads/2016/10/IMG_9980.jpg. [Accessed 12th May 2019].

[72] Deep Dream Generator, "Deep Dream Generator," [Online]. Available: https://deepdreamgenerator.com/. [Accessed 12th May 2019].

[73] J. Han and C. Moraga, "The influence of the sigmoid function parameters on the speed of backpropagation learning," *In: Mira J., Sandoval F. (eds) From Natural to Artificial Neural Computation. IWANN 1995. Lecture Notes in Computer Science, vol 930,* vol. 930, 1995.

[74] Wikipedia Contributors;, "Logistic-Curve - Logistic Function - Wikipedia," 2nd July 2008. [Online]. Available: https://en.wikipedia.org/wiki/Logistic_function#/media/File:Logistic-curve.svg. [Accessed 1st May 2019].