SOLVING MATCH THREE WITH ARTIFICIAL INTELLIGENCE: A COMPARISON OF EVALUATION FUNCTIONS

by

THOMAS BRERETON

A thesis submitted to
The University of Birmingham
for the degree of
MASTER OF SCIENCE

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
August 2017

Abstract

To be completed.

CONTENTS

CHAPTER 1

INTRODUCTION

The game of Go has long been considered the hardest challenge of artificial intelligence (AI) due to its intractable search space (10^{170} possible positions [5]). In this dissertation we introduce a new game, match three, to test the performance of AI. This required building a game named Gem Island and a screen-shot of it is in shown Figure 1.1.

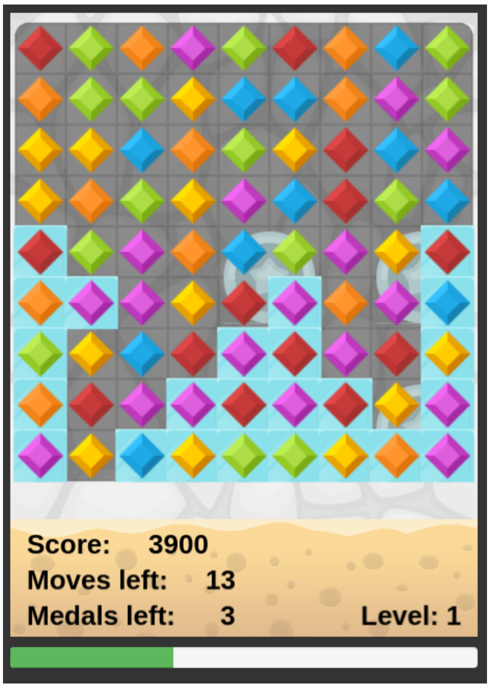

Figure 1.1: Gem Island: a match three game.

The game of Gem Island is not more challenging for AI than Go, however, the stochastic nature of the game leads to a comparable search space and a much larger branching factor. For example, the branching factor for Go is 200 on average [5] while for match three it is conservatively 1080. This branching factor can also reach a value of up to 1×10^{60} as it is theoretically possible to reach any state from one move.

We then create an AI program that successfully beats match three with a win rate of 85% and mean of 13 moves to finish (out of 20). This AI uses several variants of Monte Carlo Tree Search (MCTS) including no evaluation function, a crude evaluation function, and a neural network based function. The main contribution of this report is the comparison of MCTS with varying evaluation functions and we find that the MCTS with a crude evaluation function performs best. The second major contribution is the game of match three itself, which has an open source application programming interface (API) tailored for AI research including gathering training data, easy access to the game state, and simple method of connecting AI controllers.

CHAPTER 2

LITERATURE REVIEW

In this chapter we highlight the literature required to complete and understand this thesis.

2.1 The Agent

In this dissertation we refer to the AI program as the Agent. Where the agent senses the environment, makes some rational decision, then acts upon the environment. The environment in this case is the match three game; Gem Island.

An agent is anything that perceives the environment through sensors and acts upon it through actuators [8]. A rational agent is an agent which chooses the best action value given some reward function. In other words, it makes the right decision given some state of the environment. For example, if we have a self-driving car and it sees a person on the road through some camera sensors, the right decision would be to stop.

The agent program is how it maps the state to an action and it is this which makes the agent rational. A optimum agent program would be perfectly rational whereas a suboptimal program would select actions at random.

2.2 Decision Theory

Decision theory is the theory of making the best decisions based upon utility and probability theory. Where utility is the 'usefulness' of a state or, in other words, the happiness of the agent in a certain state. Utility is agent specific as for example two agents in chess, one white and the other black, the white agent would prefer a state where it won much more than the black agent.

2.3 Markov Decision Process

Markov Decision Processes (MDPs) are a way of modelling a sequence of decisions where there are some probabilistic and deterministic elements. The four components used to model an MDP are [8]:

- S: A set of states, with s_0 being the initial state.
- A: A set of actions.
- $T(s, a, s')$: A transition model that determines the probability of reaching state s' if action a is applied to state s.
- $R(s)$: A reward function.

For any given state s, you have a set of legal action $a \in A$, and there is some probability you move to a new state s' by the Equation:

$$P(s'|s, a) \tag{2.1}$$

Markov Decision Processes satisfy the Markov property which means transitioning to state s' is only dependent on its immediate previous state s, but independent of all other states. For every transition you receive a reward $R(s)$, according to the state and the overall goal of MDPs is to find an optimal policy (action sequence) to maximise the sum of all the rewards from each state.

2.4 Partially Observable Markov Decision Process

Words.

2.5 Game Theory

Game theory extends decision and is concerned with how multiple self-interested agents interact. (What does self-interested mean). Single player with stochastic elements can be modelled using game theory as the agent playing against the 'puzzle maker,' where the puzzle maker is, for example, the dealer in blackjack.

A game is generally represented with the following terms [4]:

- S: A set of states, with s_0 being the initial state.
- $S_T \subseteq S$: The set of terminal states.
- $n \in N$: The number of players.
- A: A set of actions.
- $T(s, a, s')$: A transition model that determines the probability of reaching state s' if action a is applied to state s.
- $R(s)$: A reward function.

2.6 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) has been ground-breaking in helping solve games with intractable search space. Previous brute force approaches could not handle the search

Figure 2.1: Initial tree

Figure 2.2: Initial Sims

space of Go as it could not fit into memory or it would take too long to compute. MCTS overcomes this by simulating games and recording the outcome W, i.e. win or lose. This process is repeated hundreds or thousands of times to record the outcome and number of plays for each move and its previous state (state-action pairs). After N simulations the move with the best win rate (denoted as $Z(s,a)$) is selected which is given by:

$$Z(s,a) = \frac{W}{N} \tag{2.2}$$

Where s is the state and a is the action taken from state s, W is the win count, and N is the number of times action a has been selected from state s.

Monte Carlo Tree Search builds a tree data structure in memory representing sequences of actions and each nodes stores the win rate (W and N). This building process can be broken into four phases; selection, expansion, roll-out and back-propagation.

To begin with we take the current state of the game as the root node (State s_0) of the tree as shown in Figure 2.1. We then simulate a game for each legal action of that root node. This means we now have statistics for each action and its subsequent state. Figure 2.2 shows the state s_0 as the root node, the legal actions as its branches, and the subsequent states of the actions as children nodes. After initialising the tree we can enter the selection phase. This involves using an algorithm (refer Subsection 2.6.1) to pick the 'best' child node as per the algorithm. This process is repeated with subsequent child nodes until we reach one without any statistics. Selecting these nodes presents a problem of exploiting actions with known good win rates and exploring alternative actions which might have better win rates. This problem is known as the multi-armed bandit problem which has been studied extensively in literature and is discussed further in Section 2.6.1.

Reaching a node with no statistics means we enter the expansion phase. The new node is added to the tree and is initialised with the statistics $N = 0$ and $W = 0$. Following on

Figure 2.3: Tree expansion

Figure 2.4: Tree Rollout

from the example shown in Figure 2.2, the expansion phase is illustrated in Figure 2.3. From there we enter the roll-out phase which means rapidly simulating the remaining moves of the game with a simple default policy (refer Subsection 2.6.2) as depicted in Figure 2.4. Once we reach the end of the game (terminal state), we can determine if we win or lose. As the name of the final phase suggest, this outcome is back-propagated up the sequence of moves carried out in the tree. For example, if the outcome was a win each node in this sequence would have its win and play count increased by one. This process is illustrated in Figure 2.5, note that we do not have any expanded nodes in the roll-out phase so no statistics are recorded in that section.

2.6.1 Tree Policy

The multi-armed problem is a problem of a gambler maximising his cumulative reward from several slot machine (one-armed bandits). The gambler plays a slot machine to get a reward from some unknown probability distribution, the more he gambler plays, the more accurate he can estimate the distribution and therefore the expected (read average) reward. The gambler must decide which machine to play, how many times to play it, and when to switch to another. He must balance this in order to maximise his cumulative reward over all the slot machines played.

The multi-armed bandit problem is a way of modelling the selection process of a child node in a Monte Carlo tree. Many solution exist but the most common and successful one is the Upper Confidence Bound for Tree (UCT) [4]. This uses the formula:

Figure 2.5: Tree Backprop

$$Z(s,a) = \frac{W(s,a)}{N(s,a)} + C\sqrt{\frac{\log(N(s))}{N(s,a)}} \tag{2.3}$$

Where $N(s,a)$ is the number of times action a has been selection from state s, $W(s,a)$ the number of times this action has resulted in a win at the terminal state, $N(s)$ is the total number of simulations played from state s, and C is a tuning constant.

UCT is used to calculate the value of the child nodes and the one with the highest value is selected. This is repeated to follow a sequence through the current tree until we come to a node which has not been explored before. UCT comprises two main components; the exploitation factor $\frac{W(s,a)}{N(s,a)}$ representing the win rate, the exploration factor $\sqrt{\frac{\log(N(s))}{N(s,a)}}$ representing how many times a node has been selected, and C which is tuned to how much the exploration factor should have an impact. The exploitation factor is high for successful nodes and the exploration factor is high for nodes that have been explored very few times. This means the UCT tree policy favours actions that result in a good win rate but will still select actions if they have not been explored very many times.

There are many other bandit-based algorithms to potentially improve upon the tree policy. Alternative upper confidence bound algorithms include UCB1-Tuned [2], Bayesian UCT [6], and EXP3 [3] [1], however the policy is not the focus of this report is therefore not covered extensively. The papers for the alternative algorithms mentioned above are included in the references for the reader to view at his leisure.

2.6.2 Roll-out

The roll-out phase is designed for rapid play-out to reduce the overall computational expense of Monte Carlo Tree Search. This is achieved by having a default policy which can be as simple as always selecting the move location at position (0,0) in subsequent states until a terminal state is reached. A more common approach is sampling over a uniform random distribution of legal moves but they can also be tailored to suit the problem domain to dramatically improve performance [4].

Figure 2.6: Neural network

2.6.3 One Look-ahead

2.6.4 One Look-ahead with Roll-out

2.6.5 Full Monte Carlo Tree Search

2.7 Neural Networks and Deep Learning

In this section we cover the background literature on neural network layers, convolutional networks, and deep learning.

2.7.1 Neural Networks

A neural network is a style of programming where programs learn from observing data. It can be thought of as a function which must first be taught how to do something so that it can give accurate answers. They are commonly used in state-of-the-art programs for image recognigtion, speech recognition, and natural language procession. In other words, they are excellent at classification and identifying patterns in complex data.

Neural networks are made up of an input, output, and one hidden layer as shown in Figure 2.6. Whereas deep neural networks are the same except with more than one hidden layer.

2.7.2 Training Neural Networks

2.7.3 Convolution Layer

In this subsection we describe the convolution operation, how it is used in the convolutional layer, and what advantages they have.

A convolution is essentially a weighted average. It can be used to detect edges in an image or create blur effects, it all depends on the weights in the convolution. Imagine we have a very small three by three image as shown in Equation 2.4, where each number is a pixel and the value represents a colour. Also shown is a filter which is typically much smaller and in our image is shown as a two by two matrix.

We then compute the first convolution by placing the filter over the four top left numbers in the image which results in the operation shown in Equation 2.5 resulting in a single value. This is repeated repeated 3 more times by sliding the filter across the image to get four values in toal as shown in Equation 2.6.

$$image = \begin{bmatrix} 0 & 1 & 3 \\ 2 & 1 & 4 \\ 1 & 5 & 6 \end{bmatrix} filter = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{2.4}$$

$$\begin{bmatrix} 0 \times 1 & 1 \times 2 & 3 \\ 2 \times 3 & 1 \times 4 & 4 \\ 1 & 5 & 6 \end{bmatrix} \Rightarrow cell(0,0) = (1 + 2 + 6 + 4)/4 = 3 \tag{2.5}$$

$$ConvolvedImage = \begin{bmatrix} 3 & 5.5 \\ 6.75 & 12 \end{bmatrix} \tag{2.6}$$

The filter can have certain weights to detect edges, dark regions, and other attributes. If we now imagine a convolutional neural network we get something like the example illustrated in Figure 2.7. We repeat the previously described process on the input layer to produce the first hidden layer. The weights, depicted as lines in the illustration, are learnt by the neural network and shared for all neurons in the first hidden layer. One shared filter is called a feature map and allows you detect certain aspects of an image such as all vertical edges in the image. For each different aspect of an image you need another feature map and therefore convolutional neural networks are generally made up of several feature mappings. Figure 2.8 shows three feature maps from the input layer to

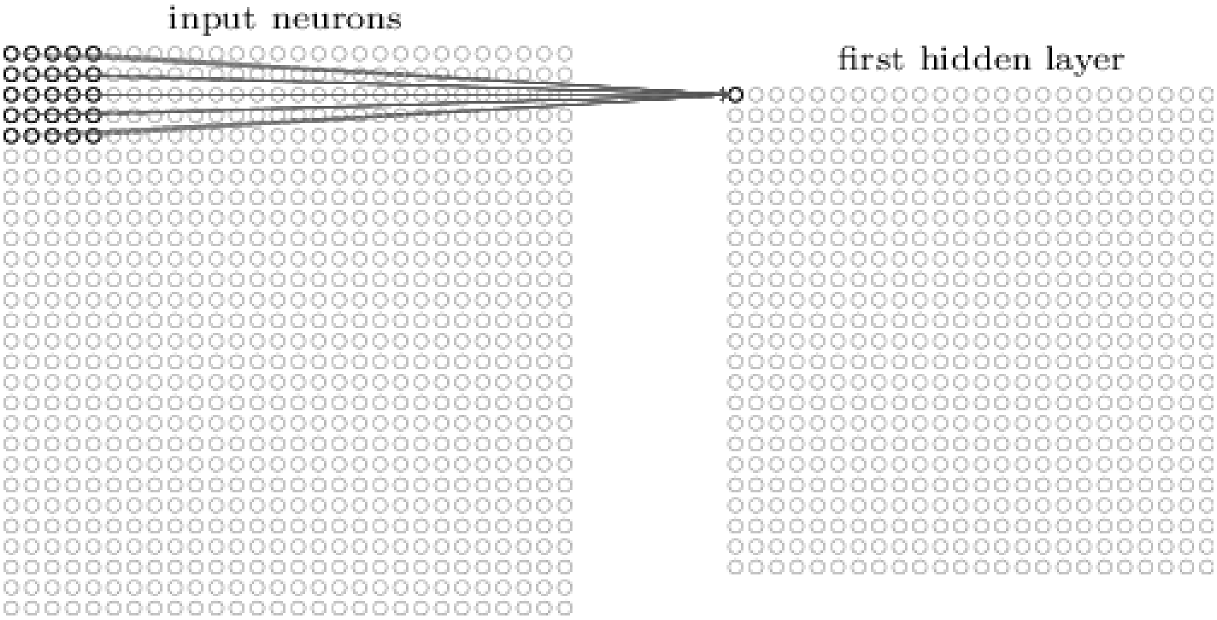

Figure 2.7: Input and First Hidden Layer of a convolutional neural network [7]

produce three different layers (channels) in the first hidden layer (3 channels of 24 by 24 neurons). Each layer would be detecting a different aspect of the image such as vertical, horizontal, and circular edges.

An advantage of a convolutional layer is that it takes into account the spatial structure of an image. This means it knows the top-left cell is next to the second top-left cell whereas in normal neural networks these two cells would be considered the same distance from each other as the top-left and bottom-right i.e. there is no distance metric.

Another advantage is the sharing of weights for each filter map. This greatly reduces the number of parameters to learn therefore significantly reducing the time to train a network. Before incorporating convolutional layers it was considered to hard to train image classifiers as they contain too much data.

2.7.4 Max Pooling Layer

Max pooling is typical pooling layer that occurs immediately after convolution layers. This layer is similar to a convolutional layer except the filter selects the maximum value rather than computing the weighted average. For example, if we have the same image

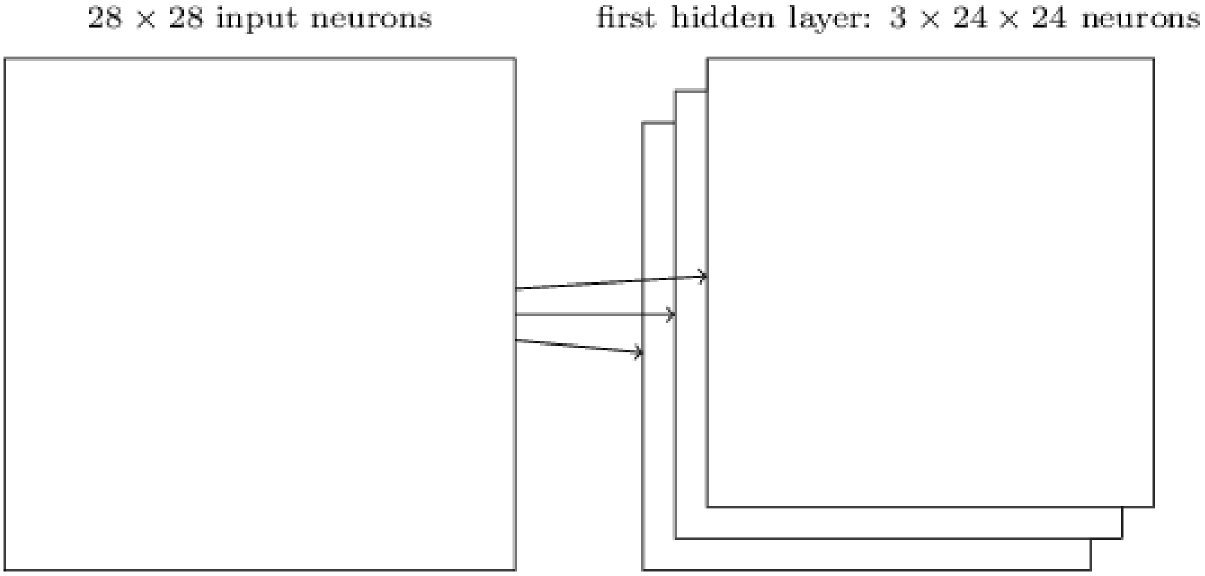

Figure 2.8: Three feature maps from the input layer [7]

as shown in Equation 2.4 and apply a two by two max-pooling filter to the four top left numbers we get a value of two as it is the maximum. This is repeated three more times to get the hidden layer as shown in Equation 2.7.

$$maxPooledImage = \begin{bmatrix} 2 & 4 \\ 5 & 6 \end{bmatrix} \tag{2.7}$$

The advantage of this to reduce the number of parameters by even more without any significant loss in data quality.

2.7.5 Dropout Layer

CHAPTER 3

PROBLEM DOMAIN

In this chapter we classify the game of match three in terms of determinism, hidden information, number of players.

The game of Gem Island (refer Section 5.1) is classified as non-deterministic due to its stochastic nature and hidden information. The non-deterministic elements are:

- Gem generation
- Hidden medals

Gems are created by randomly selecting a type between and including the range of 1 and 6. Medals are randomly placed underneath ice with the constraint that they do not overlap.

Gem Island is a single player game but can be modelled as a two player zero sum game with the player playing against the stochastic elements (commonly referred to as the puzzle maker). The player must win (1) and puzzle maker lose (-1), or vice versa, so the outcome sums to zero. Visualising the game like this helps with modelling it as a Monte Carlo Tree (refer Chapter 2.6)

Gem Island involves making a sequence of decisions to win or lose the game and it has hidden information therefore it can be classified as a Partially Observable Markov Decision Process (refer Sections 2.3 and 2.4).

For example in Gem Island, if you match three gems of the same type, the matched gems will disappear, existing ones will fall leaving empty cells at the top where three gems will randomly appear. As there are 6 gem types, this means we have $6^3 = 216$ new states. This is the minimum number of new states from any move. It is also not uncommon for an entire row to be removed (9 gems) from a cross bonus (ref game rules) which results in $6^9 = 10,077,696$ new states. Each state has equal probability of occurring and is therefore non-deterministic.

CHAPTER 4

SOLUTION DESIGN

In this chapter we discuss how to solve the non-deterministic game of Gem Island by applying current literature of Monte Carlo Tree Search (MCTS) and neural networks (refer Chapter 3 for problem domain). Moreover, we introduce a number of MCTS designs each with a different evaluation function, which are:

- No evaluation function
- Crude evaluation function
- Heuristic based evaluation function
- Evaluation network with outcome labels
- Evaluation network with utility labels

4.1 Monte Carlo Tree Search

In this section we outline the design of the modified selection and expansion process, the 'opponent' modelling to account for the stochastic medal placements, and the roll-out and back-propagation phases.

Monte Carlo Tree Search works well in deterministic games because a tree can be built to store the win rate of state-action pairs (cite). Building a tree is also useful for some non-deterministic games where the number of possible states from the previous state (branching factor) is relatively low. However, in Gem Island the branching factor is at minimum 216 (three gems) and can easily reach 10,000,000 (9 gems). In addition, the possible medal locations under the ice contribute to this branching factor. Furthermore, there is equal chance of reaching any one of those 10 million states. This means if we store a sequential state we have a very low likelihood of getting the same state in a following simulation ($1 \times 10^{-5}\%$). If we store a following sequence (level 2 in the tree) this chance drops significantly to $1 \times 10^{-10}\%$. Because of this very low chance of reaching the same state we do not expand the tree at all in our MCTS design. Rather we only store the legal actions from the root state and their corresponding win and play counts. This means we use the roll-out phase with the default policy (refer Section 2.6.2) for the remaining moves. This design is referred to as Flat UCT in literature (cite). A benefit of this is efficient memory storage as only one level of actions is stored and the encoded state (refer Section 5.2) representations are ignored. Furthermore, this reduces the back-propagation process from iterating over all expanded nodes in the sequence to simply adding the win and play count directly to the first action.

4.2 No Evaluation Function

The first design is a full Monte Carlo Tree Search without an evaluation function. This means the roll-out phase is played until the a terminal state and the action selected from the root state has its win and play count updated.

4.3 Crude Evaluation Function

This design incorporates a crude evaluation function which gives a state a rating in the range of zero to one, where one represents a certain win. This also means the roll-out phase can go to any depth, such as one look-ahead or fives moves. The function is considered crude as its only feature is to count the portion of medals remaining. The portions count is normalised by dividing it by the total portions remaining in the root state. This feature was chosen as removing all the medals is the goal of the game and is therefore the most important feature.

4.4 Heuristic Based Evaluation Function

The heuristic based evaluation function is similar to the crude version except it includes additional features. These features are designed to encourage exploration under the ice, remove as many gems as possible, and to finish the game quickly. In doing so we hope to increase the overall win rate of the AI.

4.5 Evaluation Network with Outcome Labels

The Evaluation network with outcome labels is a simple neural network with a convolutional layer with 6 feature mappings and a 50% drop-out layer. The architecture of this network is illustrated in Figure 4.1.

The network was trained by selecting one state (refer Section 5.2) at random from each of the six thousand games that were recorded from the website (refer Section 5.3). Only one state was selected from each game to avoid having correlated states. The states were changed into fourteen one-hot encoded states to make training the network simpler. This means the six gem types are now split into by six different layers where the first layer is a 9×9 array composed of ones if a type one gem existed at that position and zero if not. The next three layers represent the bonus types, one layer for ice, and four layers for medal portions. An interpretation of how this would look is illustrated in Figure 4.2.

For each state we have the outcome of that game which is what was used as the training labels. This means if state s_n belonged to a game which resulted in a win it would be assigned a label of one and zero if it was a loss. These corresponding labels were

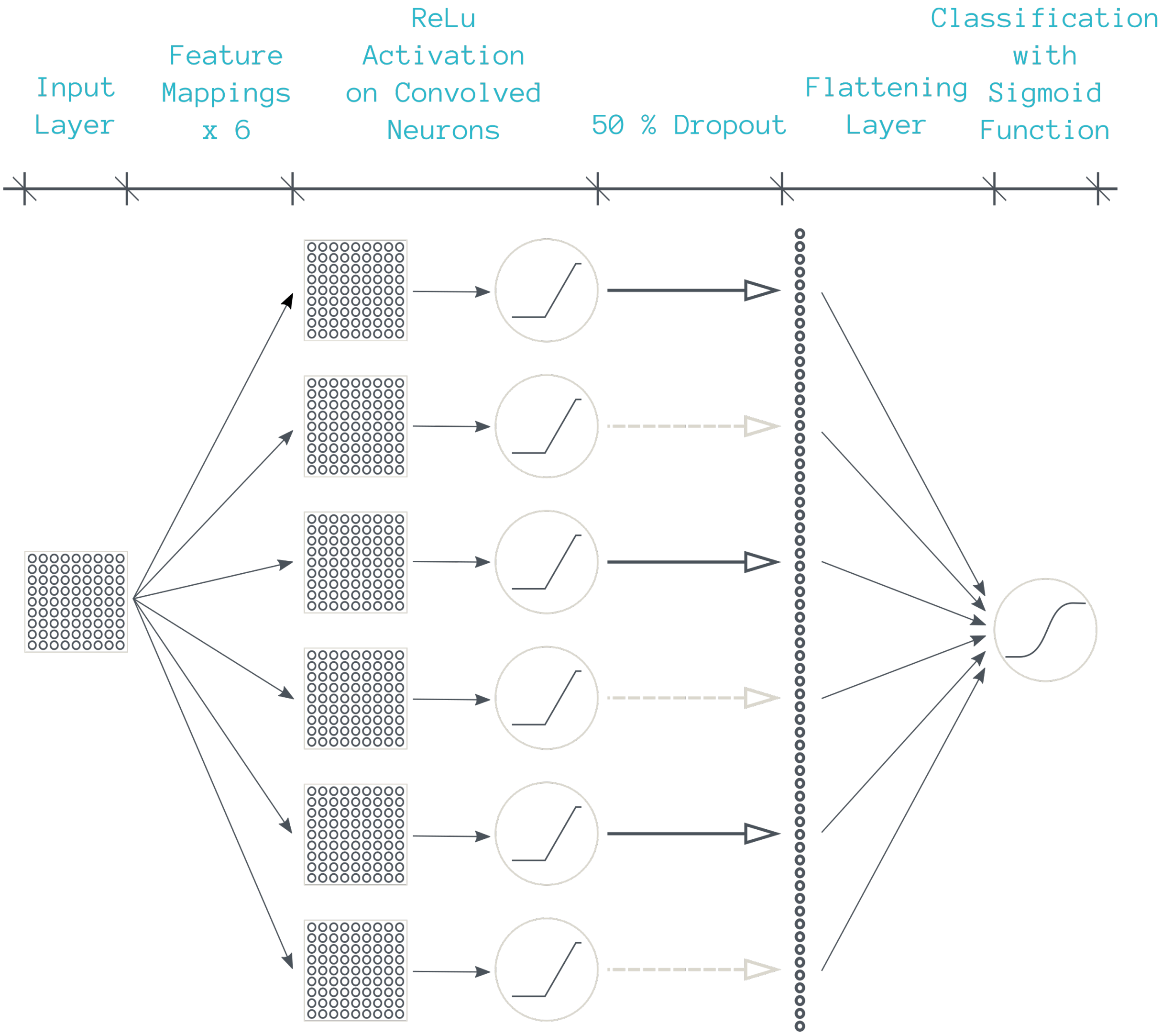
Input Layer
Feature Mappings x 6
ReLu Activation on Convolved Neurons
50 % Dropout
Flattening Layer
Classification with Sigmoid Function

Figure 4.1: The architecture of the evaluation network trained with outcome labels.

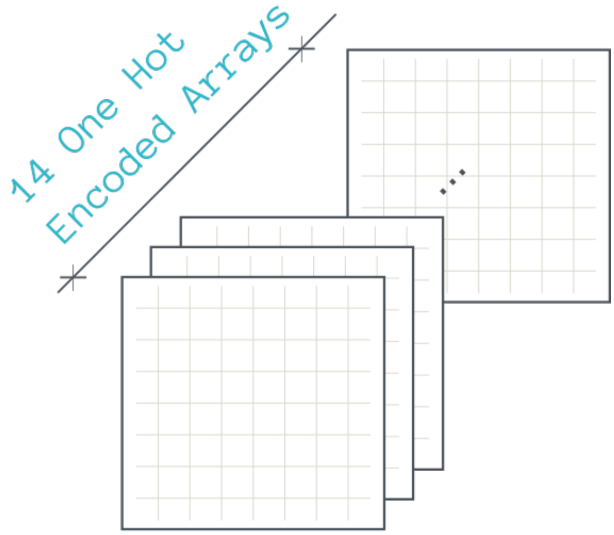

Figure 4.2: The one-hot encoded states for training the neural networks.

stored in a one-dimensional array as shown in Figure 4.3.

There is also a risk of over-fitting to the gem types rather than the pattern of gems. To avoid this we permuted the states to get every every combination and to make it gem type agnostic. Consequently this increased our training data by a factor of 720.

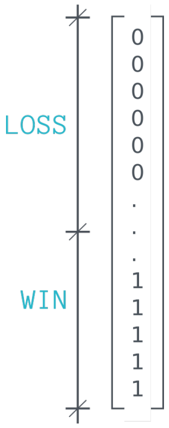

Figure 4.3: The corresponding labels for the training data.

4.6 Evaluation Network with Utility Labels

This network used a similar design to make it comparable to the outcome label network, however, instead it used what we named utility labels. These labels were computed by running Monte Carlo Tree Search on each state and selecting the maximum action win rate. In other words it works out the likelihood of winning from an arbitrary state.

CHAPTER 5

TECHNICAL REQUIREMENTS

To build an agent to win match three we also needed an open source version of the game so we could get the game state and control it with AI. Therefore we designed and implemented our own version which is outlined in Section 5.1.

To train the neural network we needed data. To gather this data we built a website that recorded users moves and logged it all in files on the server. The design of the website is outlined in Section 5.3.

5.1 Game Design: Gem Island

5.2 State Representation

The AI program must be able to read the current state of the game. This means the arrangement of the gems, ice, and medals must all be encoded so the program can determine what is shown on screen. For this we chose a simple string representation where a sequence of four numbers represented one cell in the 9×9 grid. The four numbers represented the gem type (colour), the bonus type, if ice was present, and a medal portion if it existed. This sequence of four numbers was repeated 81 times to represent each cell in the grid. The AI program could then decode this state string and simulate games via the Monte Carlo Search Tree.

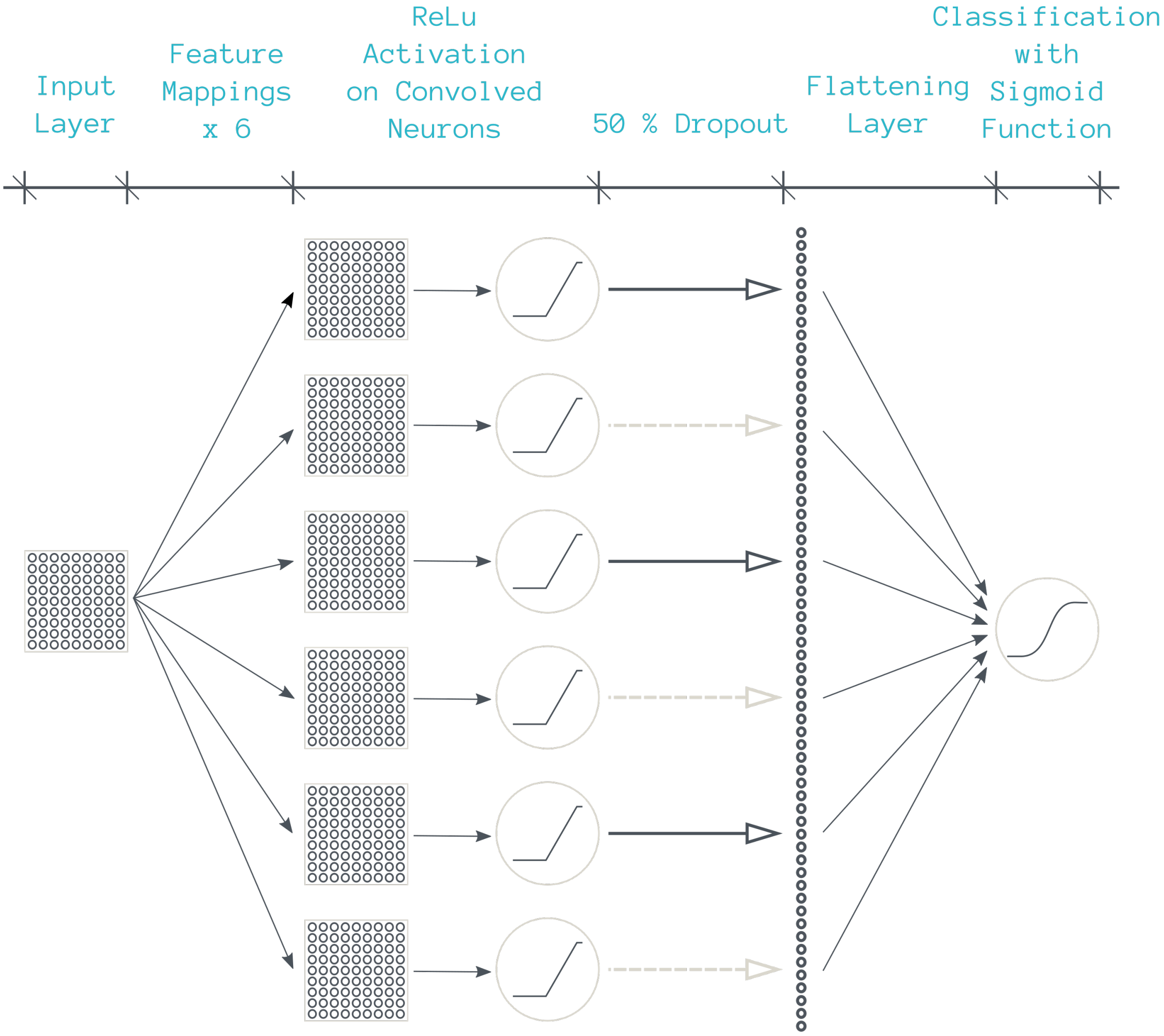

Figure 5.1: The architecture of the evaluation network trained with outcome labels.

5.3 Website Design

CHAPTER 6

ARTIFICIAL INTELLIGENCE PERFORMANCE ANALYSIS

6.1 Analysis Methodology

6.2 Results

Results from analysis. [4] Which algorithms and parameters performed better. Why did they perform better. Why did some fail.

6.3 Comparison of Evaluation Functions

How the analysis was performed (how to reproduce results).

CHAPTER 7

EVALUATION OF PROJECT

In this chapter we discuss the limitation of the author and evaluate the difficulty and usefulness of the project.

CHAPTER 8

CONCLUSION

Summary of results and findings. Recommendations of design and parameters for similar AI and game design.

CHAPTER 9

FURTHER STUDY

In this chapter we discuss possible future studies using the findings in this report and also the existing game API.

CHAPTER 10

REFERENCES

[1] Jean-Yves Audibert and Sébastien Bubeck. "Minimax policies for adversarial and stochastic bandits". In: (2009). URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/COLT09%7B%5C_%7DAB.pdf.

[2] Peter Auer and Paul Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem*". In: *Machine Learning* 47 (2002), pp. 235–256. URL: https://d2925a48-a-62cb3a1a-s-sites.googlegroups.com/site/anrexplora/bibliography/fta-2002.pdf?attachauth=ANoY7coeZCOHlDGNM8HV9yv-Ortdkr4PV0f4s4ACpG6jD5frf1k4uI4h4UO5QBDvrQgw6Q6VLAzWNLQlWmI%7B%5C_%7D%7B%5C_%7DO4CnzTwZDM9ed3nEa9h1haSb-Hyz6N1nAT7dSee3.

[3] P. Auer et al. "Gambling in a rigged casino: The adversarial multi-armed bandit problem". In: *Proceedings of IEEE 36th Annual Foundations of Computer Science.* IEEE Comput. Soc. Press, 1995, pp. 322–331. ISBN: 0-8186-7183-1. DOI: 10.1109/SFCS.1995.492488. URL: http://ieeexplore.ieee.org/document/492488/.

[4] Cameron Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *Artificial Intelligence* 4.1 (2012), p. 51. DOI: 10.1109/TCIAIG.2012.2186810. URL: http://www.cameronius.com/cv/mcts-survey-master.pdf.

[5] Sylvain Gelly et al. "The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions". In: 55.3 (2012). DOI: 10.1145/2093548.2093574. URL: http://delivery.acm.org/10.1145/2100000/2093574/p106-gelly.pdf?ip=147.

188.254.221%7B%5C&%7Did=2093574%7B%5C&%7Dacc=ACTIVE%20SERVICE%7B%5C&%7Dkey=BF07A2EE685417C5.426E5CB647D49637.4D4702B0C3E38B35.4D4702B0C3E38B35%7B%5C&%7DCFID=802598993%7B%5C&%7DCFTOKEN=66784509%7B%5C&%7D%7B%5C_%7D%7B%5C_%7Dacm%7B%5C_%7D%7B%5C_%7D=1503920618%7B%5C_%7D3dff4b75dd84e3d34976.

[6] Peter Grunwald et al. *Uncertainty in artificial intelligence : proceedings of the Twenty-sixth Conference (2010), June 8-11, 2010, Avalon, CA*. AUAI Press, 2010, p. 753. ISBN: 9780974903965. URL: http://dl.acm.org/citation.cfm?id=3023618.

[7] Michael Neilsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/.

[8] Stuart J. Russell and Peter Norvig. *Artificial intelligence a modern approach*. Pearson, 2016. ISBN: 1292153962.

CHAPTER 11

APPENDICES