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

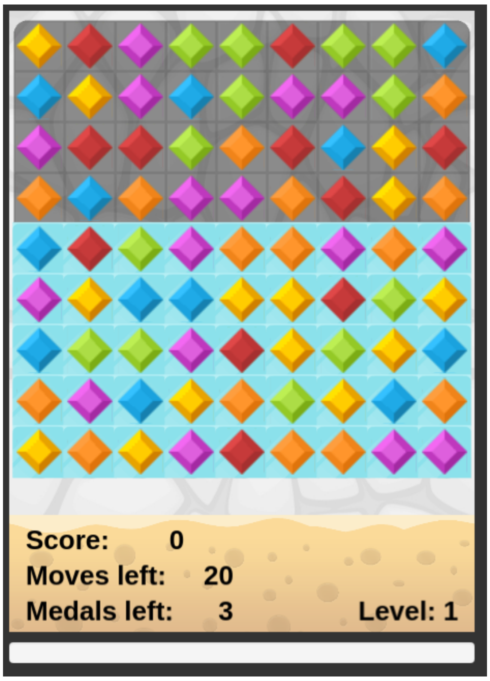
CHAPTER 1

INTRODUCTION

The game of Go has long been considered the hardest challenge of artificial intelligence (AI) due to its intractable search space (approx. 10^{170} possible states [8]). However, the game of Go is still deterministic in the sense that if we are in state s_0, and select a legal action a_0, we must transition to s_1, given the opponent is rational. This is far from real-life scenarios where there is significant factors of hidden information and randomness. This motivated us to build an AI program which would perform successfully in partially observable Markov Decision Processes (POMDPs) with stochastic elements. In this report we use Monte Carlo Tree Search (MCTS) and neural networks (NN) to beat a POMDP game (with stochastic elements) of our own making, Gem Island.

The aim of Gem Island (refer Section 5.1) is to match three or more gems of the same type over the ice to free the medals underneath. Each game begins by generating gems in a nine by nine grid each of which is randomly one of six different types. Additionally, opaque ice covers the bottom five row of the game to hide the medals, which are randomly placed underneath. Figure 1.1 illustrates a beginning state of the game. A legal move is made by swapping two gems which results in at least one match.

To account for the hidden information of the medals, we used a simulator to predict state s given observations o. To handle the stochastic elements of the gem generation we used a flat UCT MCTS [4], which stores the actions only from the root state rather than building a tree.



Score: 0
Moves left: 20
Medals left: 3 Level: 1

Figure 1.1: Gem Island: a match three game.

The main contribution of this report is the comparison of different MCTS programs with varying evaluation functions. We find that the Heuristic Based (HB) evaluation function (refer Section 4.4) performs best with a win rate of 87% and mean moves made of 12.5. in comparison to human ability, this is better than average (43%) and comparable to elite human play (89%).

The second contribution is the game of Gem Island. It provides a standard test bed for POMDPs with stochastic elements and it is designed with an application programming interface (API) suited for AI research including a method for gathering data, easy access to the game state, and a simple method of connecting AI controllers. Gem Island has a much larger branching factor than full sized Go and its search space is larger than nine by nine Go. For a direct comparison, the branching factor of Go is 200 on average [7] whilst for Gem Island it is conservatively 1080. This branching factor can also reach a value of approximately 10^{60} as it is theoretically possible to reach any state from one move. Given these factors, it clearly has an intractable search space and proves to be a non-trivial game for AI to solve.

We begin this report by introducing Markov Decision Processes (MDPs), POMDPs, Monte Carlo Tree Search (MCTS), and neural networks (NN). We then classify the game in Chapter 3: Problem Domain so that we can design a solution as outlined in Chapter 4. We complete the report with a comparison of the evaluation functions, concluding remarks, and recommendation for future research.

CHAPTER 2

LITERATURE REVIEW

In this chapter we highlight the literature and background information relevant to the project. We begin by defining an Agent, which is a rational AI program, followed by Decision Theory which is about making the best decisions given the ‘usefulness’ of each state. Markov Decision Process (MDPs) is outlined in the next section which builds on decision theory. It is a method of selecting a sequence of decisions which maximises the sum of the rewards from each state. We also build upon MDPs with Partially Observable Markov Decision Processes (POMDPs), which is similar except the agent can only make partial observations of the state and must guess the rest. In the next section we cover how Game Theory extends Decision Theory and that it is a way of modelling the interaction of rational agents. The penultimate section covers Monte Carlo Tree Search (MCTS) which is a method of finding the optimal move from the current state by simulating many games. We conclude the literature review with neural networks (NN), which is a style of programming used to ‘teach’ functions how to classify data.

2.1 The Agent

In this dissertation we refer to the AI program as the Agent. An agent is anything that perceives the environment through sensors and acts upon it through actuators [13]. 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.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 [12]. 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 [13]:

- 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 actions A, and when picking action $a \in A$, there is some probability the agent moves to a new state s' given by $P(s'|s, a)$.

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 the agent transitioned to. The overall goal of MDPs is to find an optimal policy (sequence of actions) to maximise the cumulative sum of all the rewards from each state.

2.4 Partially Observable Markov Decision Process

Partially Observable Markov Decision Processes (POMDPs) is a Markov Decision Process (MDP) except there is hidden information. To account for the hidden information, states are predicted from observations, for instance if the agent makes observation o_0, what is the probability it is in state s_0, or s_1, or any other state where $s_t \in S$. POMDPs are governed by the same elements as MDPs with the addition of [4]:

- $O(s, o)$: An observation model that gives the probability of observing o in state s.

Interestingly POMPDs, if possible, can acquire more information to perform better. As such, to achieve optimal performance Kaelbing et al [10] said that, "the agent chooses between actions based on the amount of information they provide, the amount of reward they produce, and how they change the state of the world."

2.5 Game Theory

Game Theory extends Decision Theory and is concerned with how multiple rational agents interact. It is applicable to this project as single player with stochastic elements can be modelled as the agent playing against the 'puzzle maker,' where the puzzle maker is the random element. For example, Solitaire could be modelled as a user playing against a dealer.

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

$$Z(s, a) = \frac{W}{N} \tag{2.1}$$

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 states and actions where each nodes stores the win rate. 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 part a. We then simulate a game for each legal action of that

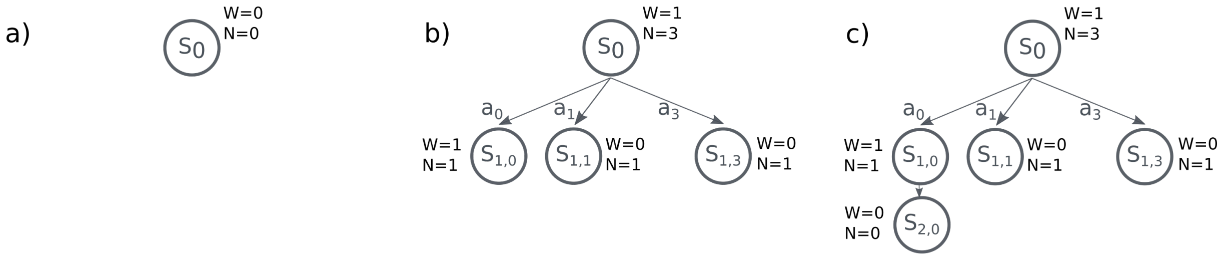


Figure 2.1: The first three phases of building a Monte Carlo Search Tree: (a) Set current state as root node (b) Initialise all legal moves of root state we can perform UCT (c) Selection of best action and expansion at first node not included in the tree.

root node. This means we now have statistics for each action and its subsequent state. Figure 2.1 part b 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 and it 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 extpansion phase. The new node is added to the tree and is initialised with the statistics $N = 0$ and $W = 0$. Following on from the example shown in Figure 2.1 part a, the expansion phase is illustrated in Figure 2.1 part c. 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 2.2. 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 2.3, note that we do not have any expanded nodes in the roll-out phase so no statistics are recorded in that section.

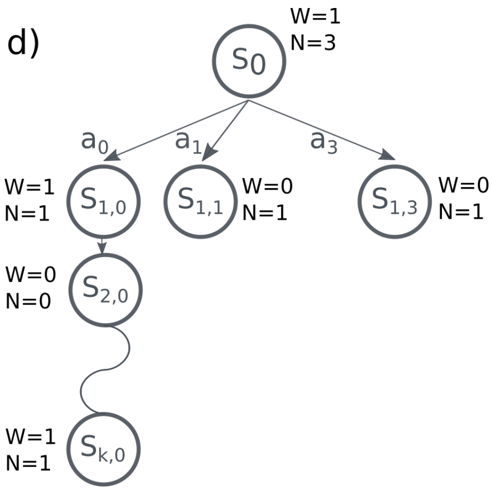


Figure 2.2: The roll-out phase of a building a Monte Carlo Tree.

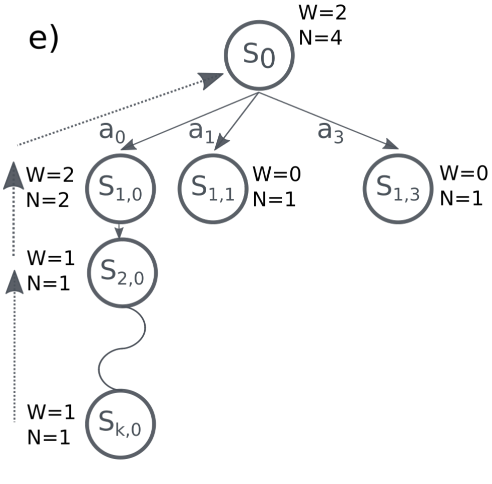


Figure 2.3: The back-propagation phase of a building a Monte Carlo Tree.

2.6.1 Tree Policy

The multi-armed problem is when a gambler tries to maximise 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 child nodes in a Monte Carlo tree. Many solution exist but the most common and successful one is the Upper Confidence Bound for Trees (UCT) [4]. This uses the formula:

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

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 three 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 the exploration constant C which is tuned to the desired level of exploration. The exploitation factor is high for successful nodes and the exploration factor is high if they have been explored very few times. This means the UCT tree policy favours actions that result in wins 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 and alternaitve upper confidence bound algorithms include UCB1-Tuned [2], Bayesian

UCT [9], 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 their 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].

2.6.3 Other Enhancements

There are further modification that can be done to Monte Carlo Tree Search to enhance it. One popular algorithm is rapid action value estimate (RAVE) and was introduced by MoGo, a 9×9 Go program [6]. RAVE solves the problems of initialising the tree (refer Section 2.6) by treating moves selected in subsequent states as if they selected from the root state (if they also exist from the root state). This means that any moves from the root state that match those in the simulation can be given the same statistics. RAVE also introduces a decay factor so that this initialisation feature disappears as more simulations are carried out.

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 recognition, 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 2.4 and deep neural networks have more than one hidden layer. A neuron fires when the value of the output function goes above some threshold. This function is composed of weights, input values, and a bias and is given in Equation 2.3.

$$o = w_i x_i - b_i \tag{2.3}$$

Where w_i is the weight, x_i is the input, and b_i is the bias. The output fires when it is above 0 and the bias is the threshold value. Typically w_i is tweaked in the training stage so the input data has an appropriate influence.

2.7.2 Training Neural Networks

Training a neural network (NN) can take days or even weeks and different methods of doing this are common areas of current research. Training a network involves supervised learning which means that it will try to classify something and we tell how close it is to being correct. If it is not correct, the program is reversed and the weights connecting the neurons are changed by a small amount to make it more accurate. This is repeated millions of times until the NN is able to produce accurate results.

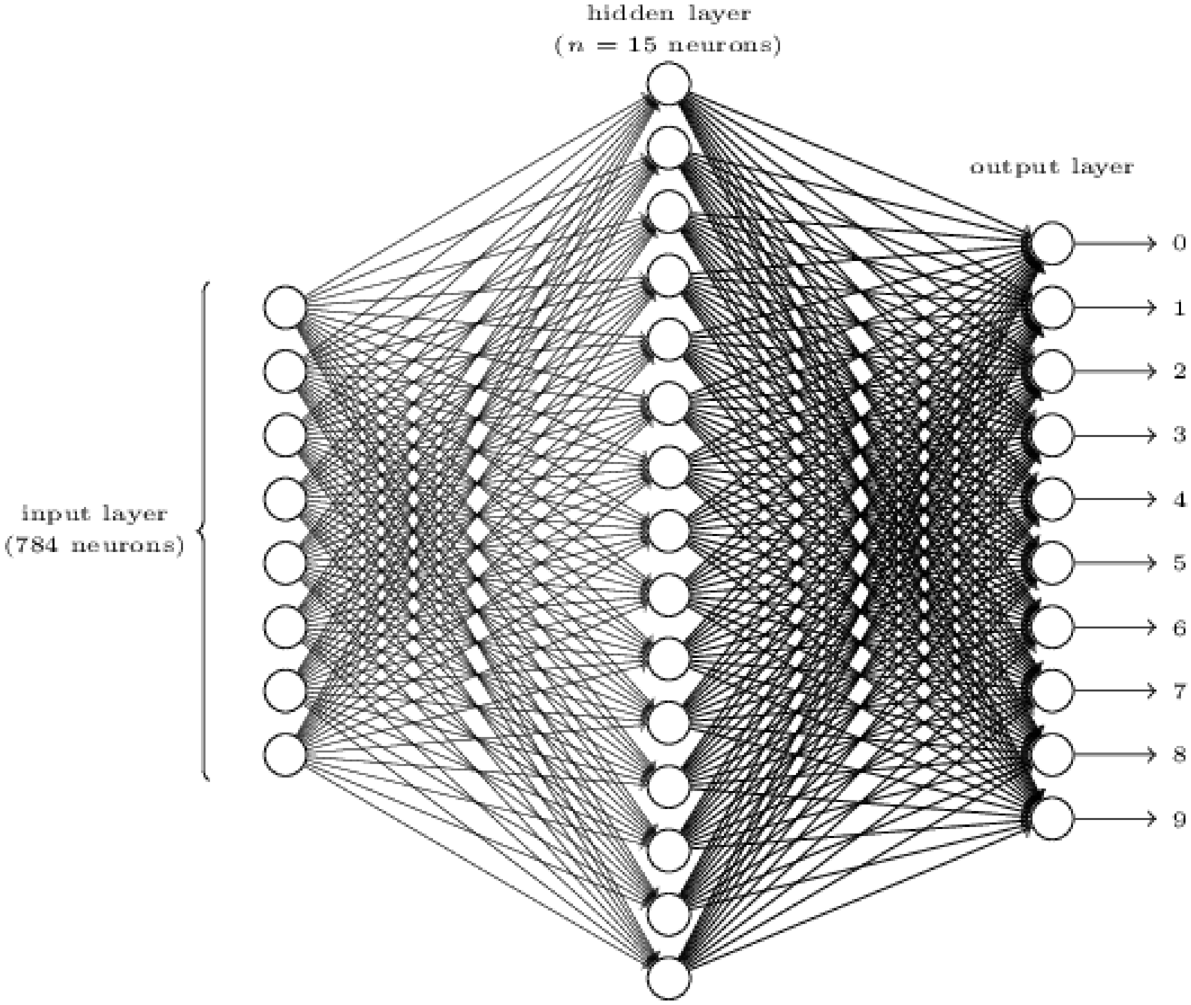


Figure 2.4: A neural network with 784 input values (only 8 shown), one hidden layer, and 10 output values for classifying numbers 0 to 9. [11]

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 vertical or horizontal edges in an image depending on the weights in the convolution. For example, imagine we have a very small 3 × 3 image as shown in Equation 2.4, where each number is a pixel and the value represents a colour. Also shown is a filter, typically much smaller than the image but here is it as a 2 × 2 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 and a single value of three. This is repeated three more times by sliding the filter across the image (the top-right four values, then bottom-left, then bottom-right) to get four values in total 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 2.5. 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

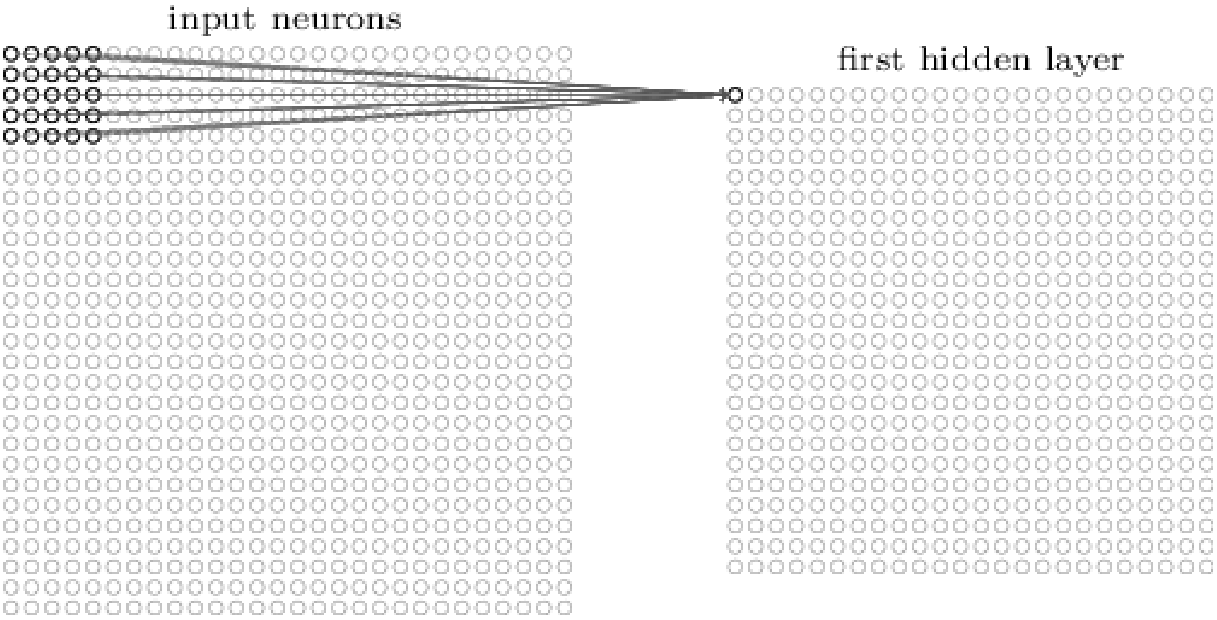


Figure 2.5: Input and First Hidden Layer of a convolutional neural network [11]

shared filter is called a feature map and allows you detect certain aspects of an image such as all the vertical edges. 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 2.6 shows three feature maps from the input layer to 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. regular layers have 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. Now, simple convolutional neural networks can be trained in minutes.

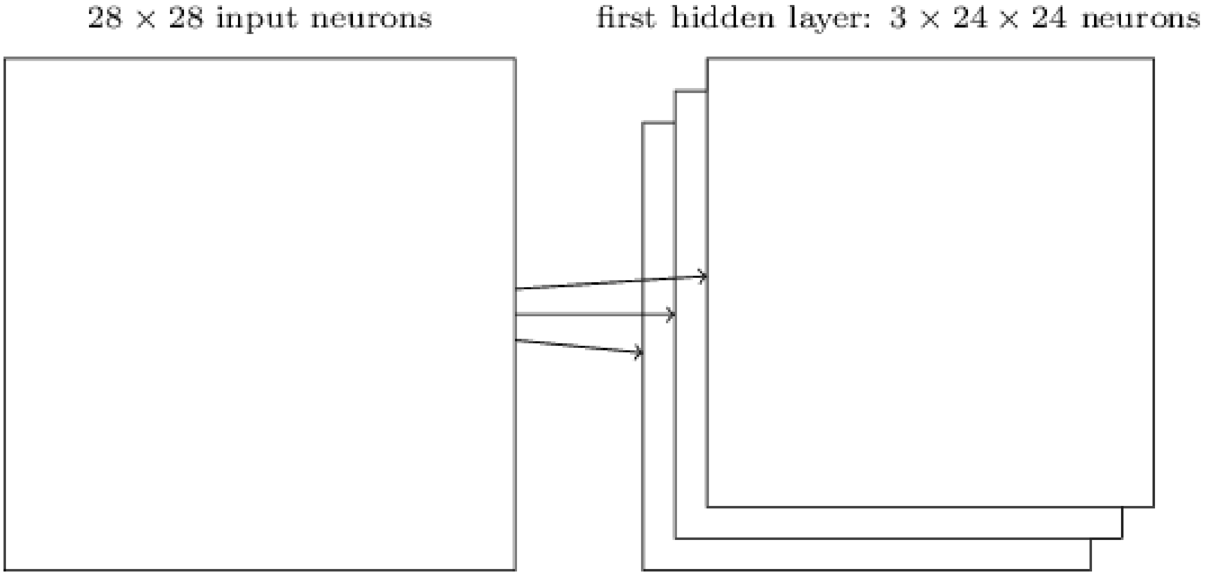


Figure 2.6: Three feature maps from the input layer [11]

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 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 Drop-out Layer

A drop-out layer is simply a training method to prevent over fitting. Over fitting is a term used to describe when a neural network predicts accurately on the training data but

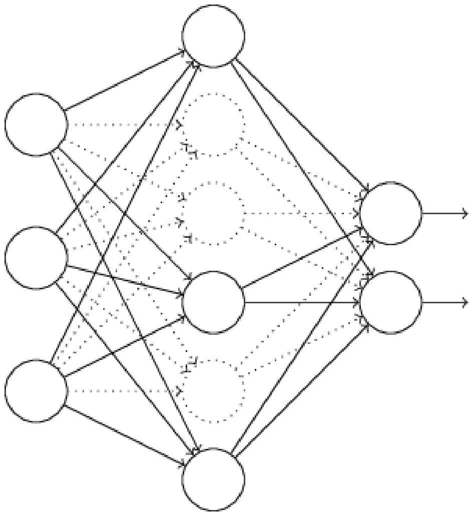


Figure 2.7: Drop-out in a neural network where the dotted lines indicate the temporarily deleted connections [11].

does not generalise well to unseen data. Drop-out prevents this by randomly selecting and temporarily deleting different connections between neurons of different layers as shown in Figure 2.7.

For example, a 50% drop-out essentially means each half of the NN will trained and able to classify the data independently. Then when it is used in practice, we no longer drop any connections and it is like having two NNs voting on the classification and we take the average as the result.

CHAPTER 3

PROBLEM DOMAIN

In this chapter we explain why Gem Island (refer Section 5.1) is classified as a POMDP (refer Section 2.4) with stochastic elements.

Gem Island is a MDP (refer Section 2.3) as it requires a sequence of decision to reach a terminal state and each state transition involves a reward. The agent must find a sequence of actions to maximise the sum of these rewards.

It is partially observable as each game is generated with five rows of ice with three medals hidden underneath. The medals are also randomly located as long as they do not overlap. An agent cannot see the medals until it removes the ice therefore, this hidden information indicates it is a POMDP.

Gem Island has stochastic elements due to the random gem generation. Each gem is generated by randomly assigning it a type with value between (and including) one to six. Both the gem generation and medal placement are sampled from a uniform distribution which means these elements are non-deterministic. This element makes it even more challenging for an agent to solve.

CHAPTER 4

SOLUTION DESIGN

In this chapter we discuss our agent design which will solve Gem Island. We apply current literature of Monte Carlo Tree Search (MCTS) and neural networks (NN) to reach a working solution. The problem domain can be found in Chapter 3 and the MCTS and NN literature can be found in Sections 2.6 and 2.7. In our solution we introduce a number of MCTS designs each with different evaluation functions and they 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 and it is possible to reach the same subsequent states in following simulations. 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 (nine 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 in the tree we have a very low likelihood of reaching the same state in a following simulation ($10^{-5}\%$). If we make a move from this state and store it in the tree (level 2 in the tree) the chance of reaching it drops significantly to $10^{-10}\%$. We can see that after twenty levels it is extremely unlikely to reach the same state again. Because of this very low chance of reaching the same state we skip the expansion phase 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 UCB in literature [5] and is illustrated in Figure 4.1. A benefit of this is efficient memory storage as only one level of actions is stored and we ignore the encoded state (refer Section 5.2). 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.

The algorithm used for the Flat UCB design is outlined in Figure 4.1 which is based upon Algorithm two in Browne et al's paper [4]

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 according to the outcome. We did not expect this to perform well as it does not realise partially uncovering medals is helpful.

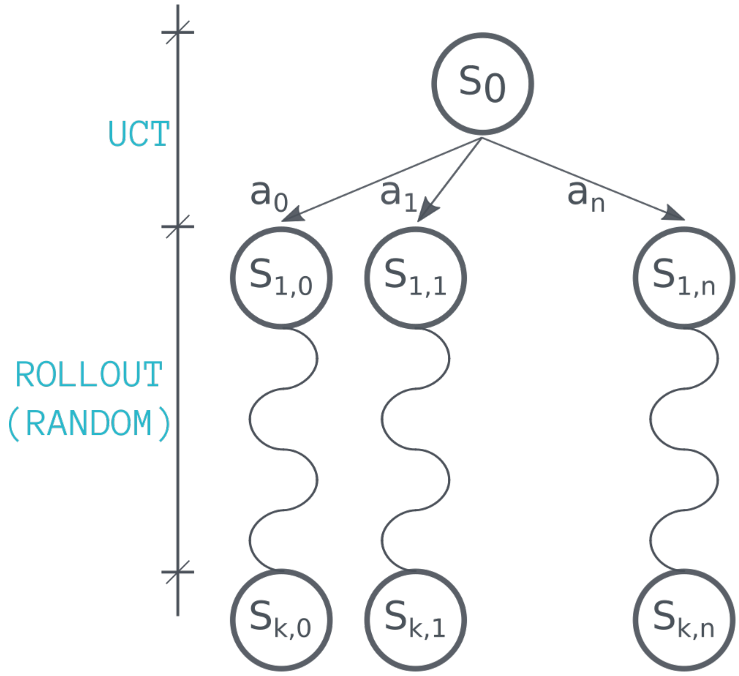


Figure 4.1: Monte Carlo Tree Search with a Flat UCB design.

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, rather than until a terminal state. 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 and this function is shown in Equation 4.1. This feature was chosen as removing all the medals is the goal of the game and is therefore the most important feature. We expected this to perform well as it can now determine that partially uncovering medals leads to success.

$$crude(mp) = \frac{mp}{mp_{remaining}} \tag{4.1}$$

Where mp represents medal portions.

```
function FlatUCB(s0)
  create root node v0 with state s0
  while within computational budget do
    vl = TreePolicy(v0)
    update = DefaultPolicy(s(vl))
    Backup(vl, update)
  return a(BestChiled(v0,0))

function TreePolicy(v)
  bestChild = argmax(v’ children of v) W(v’)/N(v’) + c.SQRT(2lnN(v)/N(v’))
  return bestChild

function DefaultPolicy(s)
  moveLimit = n
  count = 0
  while s is non-terminal or count < moveLimit do
    chose a legal action unformly at random, a
    s = state reached from selecting action a from state s
    count = count + 1
  return reward for state s

function Backup(v, update)
  N(v) = N(v) + 1
  W(v) = W(v) + update
```

Figure 4.2: The Flat UCB algorithm used in the author’s MCTS.

4.4 Heuristic Based Evaluation Function

The heuristic based (HB) evaluation function is similar to the crude version except it includes additional features. We decided upon the features of medal portions uncovered, ice removed compared to the root state, and the number of different gems compared to the root state. 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 hoped to increase the overall win rate of the AI.

Finding the correct weights of these features involved much trial and error. We began with coarse grained tuning of the weights which involved running a set of values for 20 to 50 simulations. Once we determined a rough range of values that resulted in a performance comparable to the crude function we switched to fine grained tuning. This involved tweaking the weights by small amounts and each set of values was run 100 to 300 times. The final HB function is shown in Equation 4.2. Where ir is the amount of ice removed and gr is the number of gems changed from the root state.

$$HB(mp, ir, gr) = \frac{10}{16} \times mp + \frac{5}{16} \times ir + \frac{1}{16} \times gr \tag{4.2}$$

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 4.3.

The network was trained by randomly selecting one state (refer Section 5.2) per game from the six thousand games that were recorded from the website (refer Section 5.5). Only one state was selected from each game to avoid any correlation between subsequent 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. An example of this is shown in Figure 4.4. 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 4.5.

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 stored in a one-dimensional array as shown in Figure 4.6.

There is also a risk of over-fitting (refer Section 2.7.2) 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

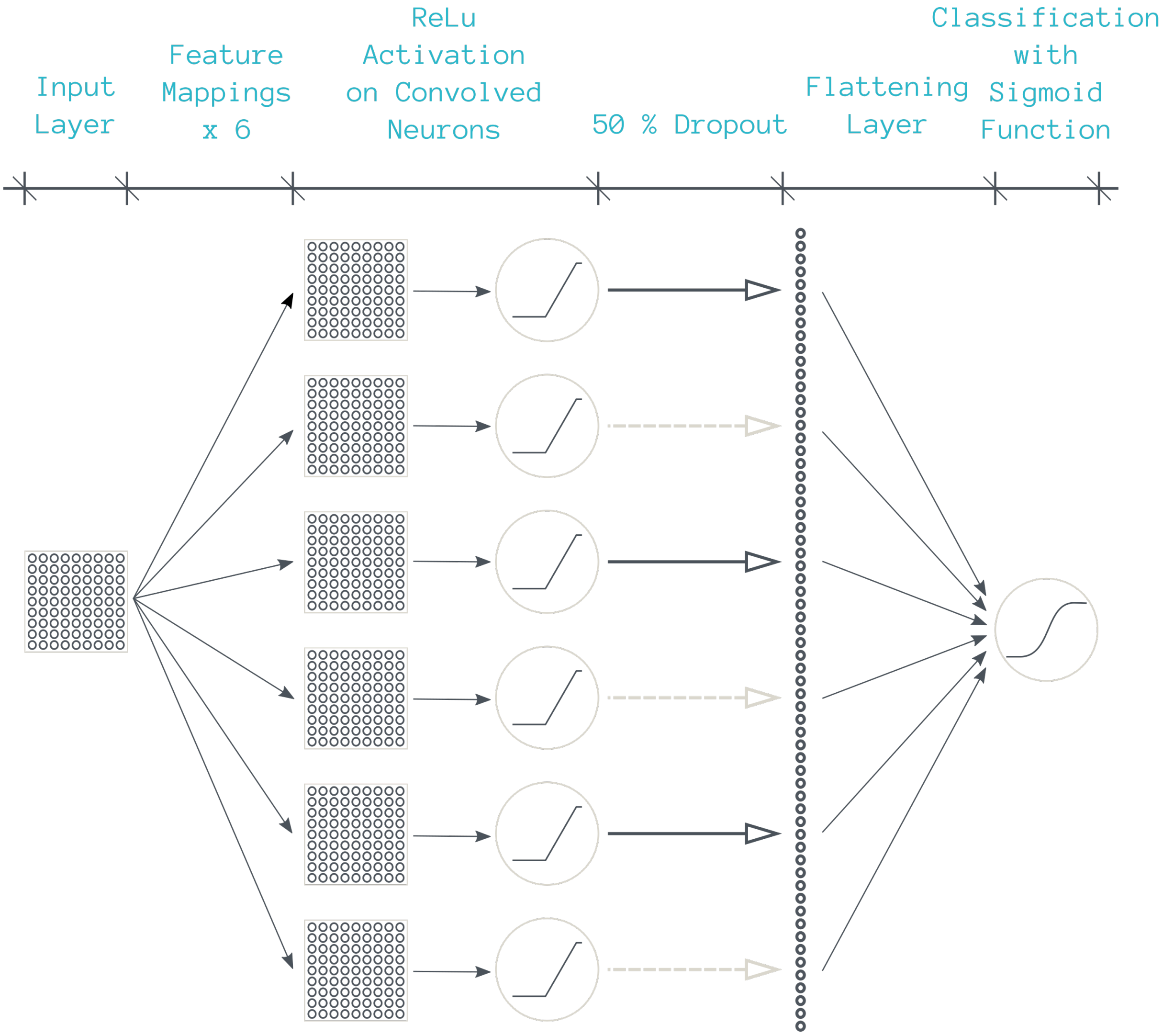

Input Layer
Feature Mappings x 6
ReLu Activation on Convolved Neurons
50 % Dropout
Flattening Layer
Classification with Sigmoid Function

Figure 4.3: The architecture of the evaluation network trained with outcome labels.

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{4.3}$$

Figure 4.4: An example of a 9 × 9 one-hot encoded array.

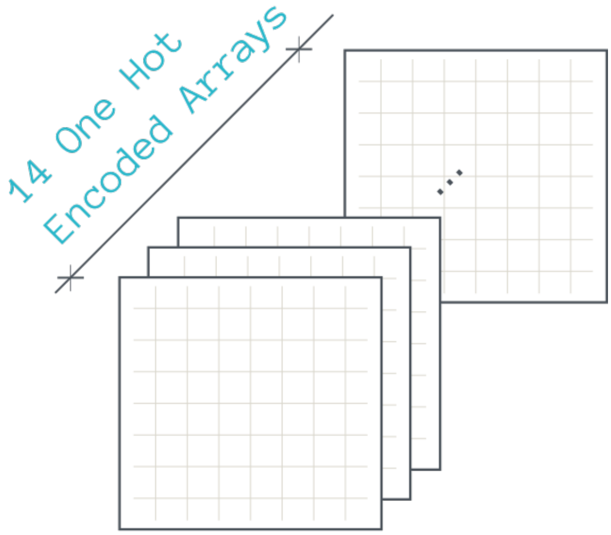


Figure 4.5: The one-hot encoded states for training the neural networks.

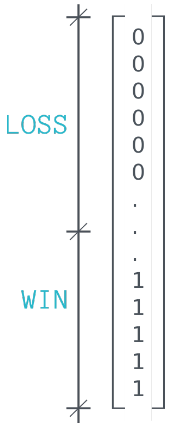


Figure 4.6: The corresponding labels for the training data.

factor of 720. We expected this network to perform reasonably well as it should learn what local gem patterns lead to a win. However, it did not and this is discussed in the results in Section 6.2.

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. We expected this evaluation network to perform very well however due to time constraints it was not carried out but it remains a possibility for future work.

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 5.5.

5.1 Game Design: Gem Island

In this section we outline the game design, design patterns used, the game logic, and sprite animation.

The board in Gem Island is a 9×9 grid of cells. Each cell can contain a gem, ice, or a medal portion. The ice is always generated in the bottom five rows with one layer (only one match to remove ice) as shown in Figure 1.1. Three medals are randomly placed under the ice at the beginning of the game and must not overlap each other. To win a game all the medals must be freed. To free a medal all the ice covering it must be removed and to remove ice a match must be made on top of it. A match is three or more gems of the same type in a succession. To make a match a user (or agent) must swap two adjacent gems. Only legal moves are allowed where a legal move means it must result in

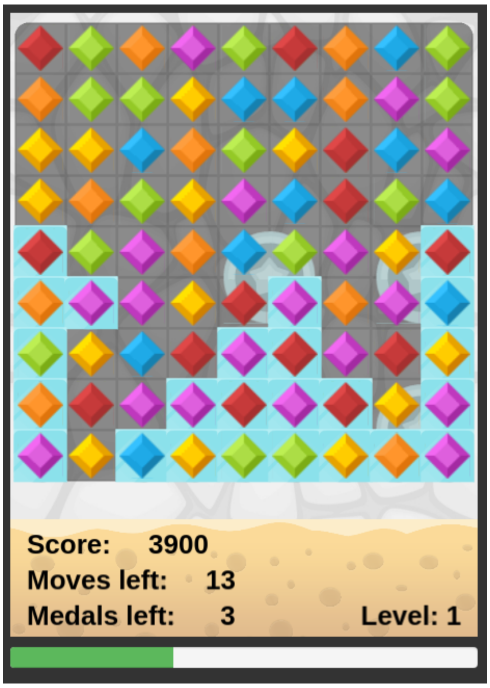


Figure 5.1: A game of Gem Island with some silver medals partially uncovered.

at least one match. Each medal always has four portions numbered zero to three from the top-left to the bottom-right (square). Figure 5.1 shows the state of a game with some portions uncovered. There are three possible bonuses and the way they are earned are as outlined in the rules below as well as their effects.

5.1.1 Game Rules

Goal: To remove all the medals.

Gem Types

- 6 gem types.
- 3 bonus types (star, cross, diamond).

Earning Bonuses

- If a bonus gem removes another bonus gem, it also performs its bonus action. This is done recursively.
- 3 or more gems in a succession of the same type is a match.

- 5 gems in a succession earns you a star bonus.
- An intersection of a vertical and horizontal match earns you a diamond bonus.
- If a match generates multiple bonuses only one is generated following the hierarchy: star, cross, diamond.

Bonus Actions

- Star bonus removes all gems of the star gem's type.
- The cross bonus removes all gems in the row/column. If the match is horizontal, the row is removed. Vertical removes the column.
- The diamond bonus removes the 9 surrounding gems of the diamond gem.

5.1.2 Game Code Design

The overall design pattern used was Model View Controller (MVC), where the model contained the game logic and sent the state and movement information to the view. This information was wrapped in a container class named `UpdateBag`. The view was designed to be naive, in the sense that it only arranged the graphics on the screens as per the information sent in the `UpdateBag`. The view also handled loading in the sprites (gem, ice, and medal graphics) and the animations of exploding and moving gems.

Several different controllers were used but they all controlled the game by sending `SwapGemsRequest` objects to the model. The `SwapGemsRequest` class is designed to contain two coordinates to indicate the gems to be swapped. The `MouseController` class listens for mouse events, such as the user clicking and dragging, and sends them via a `SwapGemsRequest`. The `NaiveAIController` reads in the encoded states, selects a legal move at random, and sends a swap request. Finally, the `MCTSController` class is designed to read in the encoded states, run one of the various MCTS implementations, and send a swap request.

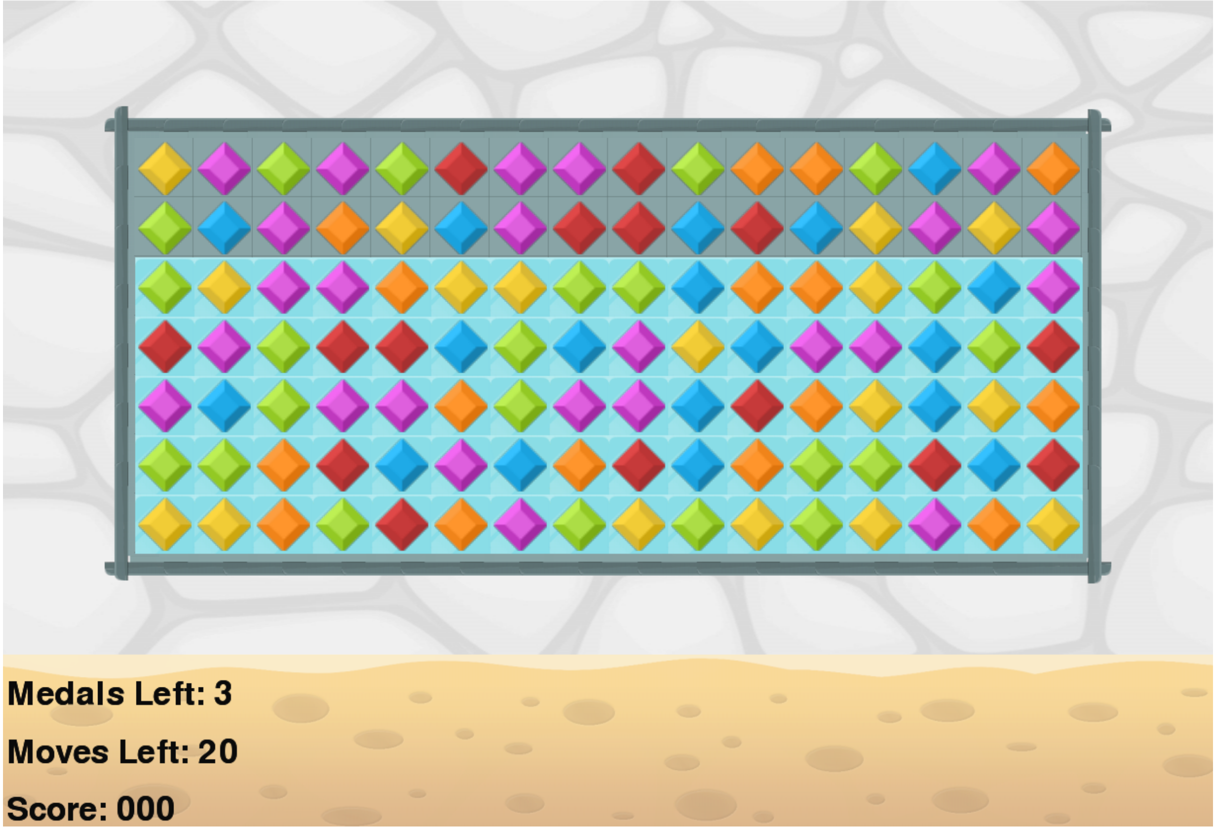
The communication between the model, view, and controller classes was handled by an event manager class: `EventManager`. The event manager class was passed into each of the model, view, and controllers so they could 'register' themselves. Every registered class could now received events and requests via their own notify method, which `EventManger` called by looping over the registered objects. Additionally, they could all send events and requests by calling the `post` method on the passed in `EventManager` object.

The model class was split into a parent and child class. The parent class contained the logic and could be used by the AI to simulate games quickly without any graphics. The child class contained additional methods for communicating with the view, such as to compute movement coordinates for the animations.

The model and view are designed to be dynamically sized allowing the gem grid to be any $m \times n$ size. Furthermore, any number of ice rows, layers of ice, and medals could be added. Figure 5.2 shows a 7×16 grid to illustrate the dynamic properties. A benefit of this dynamic design is the game can be changed to any difficulty therefore it can be made more challenging for the AI or user.

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 is present, and a medal portion if it exists. 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.



Medals Left: 3

Moves Left: 20

Score: 000

Figure 5.2: A Game of Gem Island with 7 rows and 16 columns.

Table 5.1: Gem Island Levels

Level	Ice Rows	Total Medals	Total Moves
1	5	3	20
2	7	4	25
3	9	5	30

5.3 Testing and Validation

Todo.

5.4 Analysis of Product Suitability

Todo.

5.5 Website Design

The website was designed and constructed for users to play Gem Island on either there computer or phone. It was required so we could record the games and gather training data for the neural networks (refer Section 2.7.2). We recorded the files by sending an encoded state and the actions back to the server, which wrote it in a log file.

The overall design was a distributed design with the game running on the client (user's browser) and only the database and file manipulations would be handled on the server. The game was therefore re-written in JavaScript so it could run in the browser. This design was made as it was deemed easier than setting up a full python based server application and also the current Python implementation sends `UpdateBags` for each movement of one cell, which would cause delays and stuttering. This would degrade from the game quality and discourage users to play.

To encourage people to play we created three levels with the variations as outlined in Table 5.1. Another incentive we designed was a high score table for the top fifty users of each level. This turned out to be very successful in encouraging users to play.

Writing the state-action pairs to file and saving the users high scores required server side programs. These were written in PHP and they handled the database queries and the writing of the state-action pairs to file. PHP was used as it works well with web programming languages such as JavaScript and made building the website simple.

Sending client side data to the server always involves risk and all data must be considered malicious. Therefore, security measures were designed. This included sanitising SQL queries before inserting into the database. This prevented cross site scripting (XSS) and SQL injection. Furthermore, we ran regex (regular expression) pattern matching on the state-action pairs before writing to disk. This ensured we only received data in the format we desired and prevented users from potentially writing malicious scripts to corrupt the server. Additionally, we minimised the impact of denial of service (DoS) attacks by only allowing requests to be sent to the server every two seconds. This was deemed appropriate as a user is unlikely to make a move in less than two seconds.

CHAPTER 6

ARTIFICIAL INTELLIGENCE PERFORMANCE ANALYSIS

In this chapter we outline the methodology used to perform the analysis. We then comment on the results and compare the performance of the various evaluation functions.

6.1 Analysis Methodology

Each experiment began with a game generated as per Section 5.1 and a variation of the Monte Carlo Tree Search (MCTS) program was executed until it won or lost a game. Each variation was repeated a certain number of times to gather an adequate sample size as shown in Table 6.1. Note that some of the variations failed consistently therefore they were repeated fewer times. The MCTS parameters are also outlined in Table 6.1 where Move Depth is how many moves are made per simulation, Game Limit is the number of simulations before selecting a move, C is the exploration constant.

The configuration for Gem Island is shown in Table 6.2. As per Section 5.1, the gem

Table 6.1: The MCTS parameters and the number of times each variation was completed

Evaluation Function	Times Completed	Move Depth	Game Limit	C
None	100	20	100	1.4
Crude	300	5	100	1.4
Heuristic Based	300	5	100	1.4
NN with Outcome Labels	100	5	100	1.4

Table 6.2: The Gem Island Configuration for the Analysis

Parameter	Value
Rows	9
Columns	9
Ice Rows	5
Ice Layers	1
Total Moves	20
Total Medals	3

Table 6.3: MCTS variations and their win rates

Evaluation Function	Win Rate	Mean Moves Made	Mean Score
Random	0 %	20	33,809
None	33 %	17.9	65,441
Crude	85 %	12.8	142,855
Heuristic Based	87 %	12.5	167,783
NN with Outcome Labels	21 %	18.8	51,415

types and medal locations are generated randomly at the beginning of each game.

6.2 Results and Comparison

The results from the experiments performed as per Section 6.1 are shown in Table 6.3. For a baseline, one hundred games were played by randomly selecting a legal move and the results have been included as the first row in the table under the evaluation function name of Random. The win rate of 0% for selecting random moves also provides evidence that it is a non-trivial game and requires intelligence to solve it.

From Table 6.3 we can see the Heuristic Based (HB) evaluation function (refer Section 4.4) performed best with a win rate of 87% and a mean moves made (MMM) of 12.5. This is only slightly better than the Crude function (refer Section 4.3) which is unsurprising as medal portions uncovered is the most important feature to win the game. The HB function also included two additional features, ice and gems removed. This slight increased win rate and slight decreased MMM importantly indicates that these features do not subtract from the success and given more time to tune the parameters it may have performed better. As per the results, we can conclude that removing ice and gems assists in success of the function. This is likely due to the agent acquiring more information when removing ice and the removal of more gems gives a greater chance of random combinations which lead to medal removals. This logic follows along the lines of the quote by Kaelbing et al in Section 2.4 on POMDPs, where optimum moves must balance between maximising reward and acquiring more information.

No evaluation function and the NN with outcome labels (NNOL) both perform badly but are still better than randomly selected moves. It shows that an MCTS with a full roll-out and no evaluation function can only complete the game with a win rate of 33% and MMM of 17.9. This is likely due to the lack of domain knowledge and indicates some of this knowledge is required to achieve a high win rate. Interestingly, the NNOL performs worse than no function with a win rate of 21 %. This is likely due to the method in which it was trained. It was trained so that a states s_0 (start) and s_{20} (terminal) of the same game which results in a loss will have labels of 0. However, s_0 will have zero medal portions uncovered and s_{20} will likely have eight medal portions uncovered. s_{20} clearly has a higher value than the initial state due to the amount of portions uncovered. This means the neural network is not learning to uncover medals as the labels give no indication that uncovering them is important.

If we compare the performance of the HB function with humans we see that it is approximately equal to the human players, if not better on average. The win rate of the top players that have played more than fifty games are shown in Table 6.4. ‘Astrophysicist’ achieved a much higher win rate compared to others with a rate of 89%, which is only 2% better than the MCTS with HB function. Considering all top players, the program achieved significantly better than average human playing ability and comparable to the human elite.

Table 6.4: The top human players for level 1 of the website

Nick name	Win Rate	Game Played
Astrophysicist	89 %	118
InputNameHere	69 %	65
Elliott	68 %	82
Gillyb	37 %	931
default	35 %	147

CHAPTER 7

EVALUATION OF PROJECT

In this chapter we discuss the limitation of the author and evaluate the difficulty and usefulness of the project. To evaluate the project we must first mention the authors limitations and background. The author did not undertake the modules for Introduction to Artificial Intelligence and Neural Networks which covered both Monte Carlo Tree Search (MCTS) and convolutional neural networks (CNN). This meant the author learned this knowledge independently and external to the university courses. Moreover, prior to this project the author had not used the language of Python, JavaScript, or PHP. All of these skills were learn during the project. To appreciate what was completed in this project we provide an outline as follows.

- Build the Gem Island Game in Python
- Build the game in JavaScript
- Build website and server application in PHP
- Build, tune and evaluate MCTS
- Build, tune and evaluate CNN

It is also worth mentioning the authors motivation in choosing this project. He desired to learn how artificial intelligence (AI) works and how it is implemented. More specifically, how MCTS and CNN work. This required two objectives to be completed, which are (1) build a non-trivial game for AI to solve and (2) build an AI agent to solve it. Both of these objective were met. A game was produced that many users played as shown by over 6000 games recorded in the database. Also, an AI was built successfully with a 87% win rate over 300 games. The software engineering of the game was also successful. We employed the design pattern of model-view-controller (MVC) and it separated the code into logical classes. Furthermore, it was simple to connect additional controllers such as the AI agent.

CHAPTER 8

CONCLUSION

In conclusion the Heuristic Based (HB) evaluation function (refer Section 4.4) performed best with a win rate of 87% and a mean moves made (MMM) of 12.5. This performance was only slightly better than the crude function (refer Section 4.3), which had a win rate of 85% and MMM of 12.8 . Interestingly, the evaluation network with outcome labels (refer Section 4.5) performed the worst even when considering the full Monte Carlo Tree Search (refer Section 4.2). The crude evaluation function only considered medal portions uncovered whilst the HB version also measured ice removed and total gems changed. While these additional features did not dramatically improve the performance it proves that there is still room for improvement upon the crude version. Given more time the win rate of the HB function may have been higher. In conclusion, we show that a POMDP (refer Section 2.4) with stochastic elements can be solved effectively with our Flat UCT MCTS (refer Section 4.1) and HB evaluation function (refer Section 4.4). The recommended weighting of the function are given in Equation 4.2. We also conclude that training an evaluation network cannot simply be done with outcome labels. this is evidenced by the performance of the network as shown in the analysis section (refer Section 6.2). To achieve better neural network performance we recommend that each state from a game must be given a rating so that it learns uncovering medal portions is critical to winning the game.

CHAPTER 9

FURTHER STUDY

In this chapter we discuss possible future studies using the findings in this report. For further study we recommend training the evaluation network with utility labels as per Section 4.6. The code repository listed in Appendix A already has a script to calculate these labels and some have been produced. Time must now be spend on designing a successful network.

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://link.springer.com/article/10.1023/A:1013689704352.

[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] Pierre-Arnaud Coquelin and Rémi Munos. "Bandit Algorithms for Tree Search". In: (2007). URL: https://hal.inria.fr/inria-00150207.

[6] Sylvain Gelly and David Silver. "Combining online and offline knowledge in UCT". In: *Proceedings of the 24th international conference on Machine learning - ICML '07*. New York, New York, USA: ACM Press, 2007, pp. 273–280. ISBN: 9781595937933.

DOI: 10.1145/1273496.1273531. URL: http://portal.acm.org/citation.cfm?doid=1273496.1273531.

[7] Sylvain Gelly et al. "Modification of UCT with Patterns in Monte-Carlo Go". In: (2006). URL: https://hal.inria.fr/inria-00117266v3.

[8] 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://dl.acm.org/citation.cfm?id=2093574.

[9] 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.

[10] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. "Planning and acting in partially observable stochastic domains". In: *Artificial Intelligence* 101 (1998), pp. 99–134. URL: http://people.csail.mit.edu/lpk/papers/aij98-pomdp.pdf.

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

[12] Martin J. Osborne and Ariel. Rubinstein. *A course in game theory*. MIT Press, 1994, p. 352. ISBN: 0262650401.

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

CHAPTER 11

APPENDICES