

# THESIS

## SOLVING DOTS & BOXES USING REINFORCEMENT LEARNING

Submitted by

Apoorv Pandey

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2022

Master's Committee:

Advisor: Charles W Anderson

James Ross Beveridge

Edwin K P Chong

Copyright by Apoorv Pandey 2022

All Rights Reserved

## ABSTRACT

### SOLVING DOTS & BOXES USING REINFORCEMENT LEARNING

Reinforcement learning is being used to solve games which were previously deemed too complex to solve, the most notable example in recent years being DeepMind solving Go. Dots and boxes is a 2-person game, known by many names across the world and quite popular with children. Here, a reinforcement learning agent learns to play the game.

The goal was to develop an agent which would learn to win games, could intelligently execute complex trapping strategies present in the game, and shed new light on game-playing strategy. A 3x3-sized dots and boxes board was used.

The agent learned to defeat a random opponent with a win rate of over 80%, and the next version of the agent learned to defeat the previous agent with a win rate of over 99%. A full game analysis was performed for the agent. Unfortunately, the agent was not intelligent enough to defeat a human player.

## ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Charles Anderson, for his keen guidance and incredibly valuable suggestions while performing my research. I would also like to thank the Computer Science department at CSU for helping me providing me with knowledge and resources required to perform the experiments for this thesis. Finally, I would like to thank the CSU Graduate School for their support during the entirety of my graduate program.

## DEDICATION

*I would like to dedicate this thesis to my parents.*

## TABLE OF CONTENTS

ABSTRACT . . . . .	ii
ACKNOWLEDGEMENTS . . . . .	iii
DEDICATION . . . . .	iv
LIST OF TABLES . . . . .	vii
LIST OF FIGURES . . . . .	viii
Chapter 1      Introduction . . . . .	1
Chapter 2      Background and Related Work . . . . .	4
2.1          Reinforcement Learning . . . . .	4
2.1.1      DeepMind . . . . .	6
2.2          Dots & Boxes . . . . .	7
2.2.1      Related work . . . . .	9
Chapter 3      Methodology . . . . .	12
3.1          Game Representation . . . . .	12
3.2          Reinforcement Learning . . . . .	12
3.3          Training . . . . .	14
3.3.1      Initial missteps . . . . .	16
Chapter 4      Results . . . . .	18
4.1          Simple opponent . . . . .	18
4.1.1      Training . . . . .	18
4.1.2      Testing . . . . .	19
4.2          Random opponent . . . . .	20
4.2.1      Training . . . . .	20
4.2.2      Testing . . . . .	22
4.2.3      Predicted moves . . . . .	24
4.2.4      More training . . . . .	24
4.2.5      Trapping . . . . .	25
4.3          Against itself . . . . .	26
4.3.1      Training . . . . .	26
4.3.2      Testing . . . . .	27
4.3.3      Predicted moves . . . . .	30
4.3.4      Trapping . . . . .	31
4.3.5      Game-play analysis . . . . .	31
Chapter 5      Conclusion . . . . .	35
5.1          Comparison again related work . . . . .	35
5.2          Future work . . . . .	36
Bibliography . . . . .	37

Appendix A	License . . . . .	39
------------	-------------------	----

## LIST OF TABLES

4.1	Hyperparameter values for best network against simple opponent . . . . .	20
4.2	Hyperparameter values for best network against random opponent . . . . .	23
4.3	Hyperparameter values for best network against intelligent opponent . . . . .	28
4.4	Game moves (sequential). Play proceeds left to right across a row and continues on the next row. . . . .	32
4.5	Game moves (continued) (sequential) . . . . .	33



## LIST OF FIGURES

2.1	Neural network for learning state prediction and Q function . . . . .	5
2.2	A 2x2 dots & boxes game [1]. Red edges are moves made by Player A and gray edges are moves made by Player B. In this example, Player A wins with 3 completed boxes, while Player B only has 1 completed box. . . . .	8
2.3	Double Cross strategy [1] . . . . .	8
3.1	Game state representation: a.) edge ordering, b.) example state, represented as $[0, 1, 1, 0, 1, 0 \dots, 0]$ . . . . .	13
4.1	Win rate against simple player during training . . . . .	19
4.2	Win rate against simple player during testing . . . . .	21
4.3	Win rate against random player during training . . . . .	22
4.4	Win rate against random player during testing . . . . .	23
4.5	Intermediate state of the game . . . . .	24
4.6	a.) First action taken by agent for state in Fig. 4.5. b.) Second action taken by agent. . . . .	24
4.7	Performance of agent during further training. The line represents the average of all runs, while the shaded area shows the upper and lower bounds of performance of the agent across all runs. . . . .	25
4.8	a.) Position 1 in the double cross strategy depicted in Fig. 2.3. b.) Next action taken by agent. . . . .	26
4.9	Win rate against intelligent player during training . . . . .	28
4.10	Win rate against trained player during testing . . . . .	29
4.11	Win rate against random player during testing . . . . .	29
4.12	Intermediate state of the game . . . . .	30
4.13	a.) First action taken by agent for state in Fig. 4.12. b.) Second action taken by agent. . . . .	30
4.14	a.) Position 1 in the double cross strategy depicted in Fig. 2.3. b.) Next action taken by agent. . . . .	31

# Chapter 1

## Introduction

With increase in computational power, machine learning has become increasingly viable in solving problems previously deemed too complex to solve programmatically. Reinforcement learning is a type of approach in machine learning in which the agent receives "reinforcements" based on whether or not it performed well or poorly. Among the various sub-domains of machine learning, reinforcement learning has arguably benefitted the most from the power increase, since the approach focuses on training an agent to achieve its goal an enormous number of times using some reward policy (of reinforcements).

A major application of reinforcement learning is in solving games. Games align well with reinforcement learning due to their nature of having definitive step-oriented actions in order to achieve certain outcomes. Consequently, a lot of research has been done in the space. DeepMind has been a notable example, developing their own "AlphaGo" program to learn to play the game of Go and beating world-class human players. Furthermore, they developed the next iteration of it called AlphaGo Zero, which handily beat AlphaGo, demonstrating the potential of sheer pace of improvement of game-playing using reinforcement learning.

However, there are certain factors that reinforcement learning algorithms struggle with when learning to play games. One factor is when the myopic outcome is not necessarily the globally optimal one, such as sacrificing a chess piece in order to gain a more advantageous position. This complicates computing and providing reinforcements because the global outcomes need to be accounted for when providing intermediate move reinforcements. This slows down the rate of learning, since the same move might work well in a particular chain of events, but be a poor move in another chain of events.

Another scenario with which reinforcement learning struggles is when the number of moves for a player is variable, and not periodic. Many games have clauses where making certain moves or achieving certain intermediate states results in the player receiving another turn. This complicates

analysis considerably because the state space becomes much harder to explore. The player might not encounter certain states whatsoever, despite them being valid states, and thus not learning how to play in those states.

Keeping the above in mind, I attempted to solve the game of Dots and Boxes, a game I myself have played a lot. Dots and boxes is a 2-person game, known by many names across the world, and quite popular with children. More specifically, I attempted to develop an agent that would never lose a game of Dots and Boxes to a human player, similar to what DeepMind achieved. The goal was to develop an agent which would learn to win games, could intelligently execute complex trapping strategies present in the game, and shed new light on game-playing strategy.

However, Dots and Boxes has not one but both of the aforementioned challenges that reinforcement learning struggles with. Further details about the game and its specific rules which lead to the challenges mentioned are discussed in section 2.2.

In this work, I focused on a 3x3-sized dots and boxes board. The dots were constant, and the edges were variable. Hence, the game was represented as a list of all edges, where 0 indicated that the edge did not exist and 1 denoted that the edge exists. The action was represented as a one-hot encoded vector. Together, they constituted of the 48 inputs, with the 1 output being whether the agent won or lost (1 for player 1 win, -1 for player 1 loss). Since the board was 3x3 (i.e. 9 boxes) there was no possibility of a draw game.

A fully connected neural network was used for training the agent. The agent was initially trained against a simple opponent, which picked the first available move, and learned to handily beat this opponent. Training against this opponent helped shed light on the hyperparameters which worked well for training the agent. Next, the agent was trained against a random opponent, and it took over 400 thousand games for the agent to defeat it with a win rate of over 80%. Finally, a new agent was trained against the previously trained agent and beat the previous agent with a win rate of 100% after training for 20 thousand games.

A game-play analysis was performed of the new agent against the previous one, and the agent made optimal moves towards victory at many opportunities. However, it also attempted to com-

plete boxes by itself, often by making a 3rd edge for a box, which is not a good strategy and would lead to defeat against a human opponent. Training against an opponent without the knowledge to complete such opportunities led to the agent learning a strategy which was not suitable in a general adversarial setting.

The methodology chapter details the game representation and training process, as well as outlining initial pitfalls that were made when attempting to train the agent. The results chapter details the best results against each type of opponent, alongside observations about the hyperparameters which worked best in each scenario. Finally, the conclusion chapter summarizes the outcomes of this research, and details its comparison against a similar work. It also lists potential suggestions to improve the current outcomes achieved.

# Chapter 2

## Background and Related Work

This chapter discusses the history and concepts of reinforcement learning. It also talks about the role of reinforcement learning in solving games. Finally, it provides an introduction to the games of Dots & Boxes and previous work done to solve it.

### 2.1 Reinforcement Learning

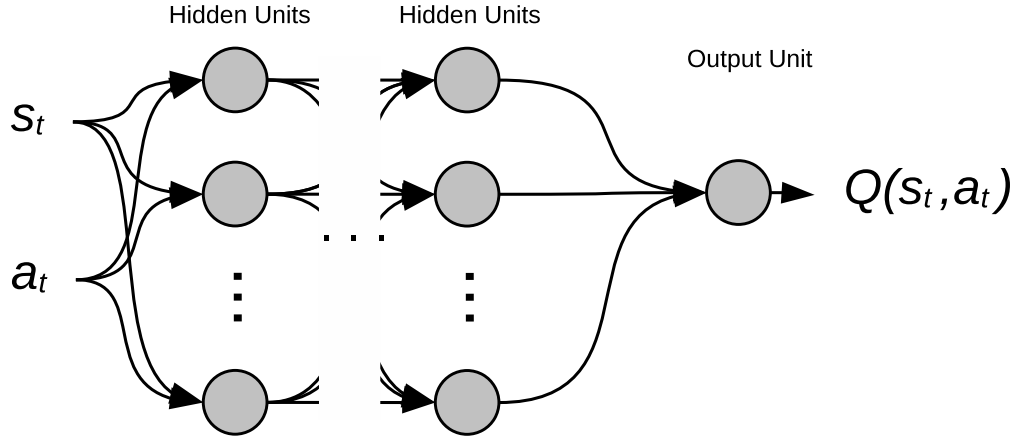
Reinforcement learning is an area of machine learning which focuses on training an agent using some reward policy. The agent learns by attempting to achieve its goal multiple times. The agent's training is a balance of exploration (of untried actions) and exploitation (of previous experience).

Reinforcement learning is one of the 3 basic machine learning paradigms, alongside supervised and unsupervised learning. It does not require labeled input/output data, and sub-optimal actions do not require explicit correction.

Reinforcement learning algorithms are generally model-free, i.e. do not use the transition probability distribution (and the reward function) associated with the Markov decision process (MDP) [2]. The transition probability distribution (or transition model) and the reward function are often collectively called the "model" of the environment (or MDP), hence the name.

SARSA is an algorithm for learning a Markov decision process policy, used in the reinforcement learning. Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy which tells an agent what action to take under what circumstances. A SARSA agent interacts with the environment and updates the policy based on actions taken, hence it is an on-policy learning algorithm.

This is done by determining a reward system, which determines a reward for each step taken by the agent. It makes use of the reward values in subsequent attempts, i.e. exploitation of knowledge. "Q" names the function that is trained to approximate the sum of future reinforcements.



**Figure 2.1:** Neural network for learning state prediction and Q function

Neural networks have been used for Q function approximation since the 1980's [3, 4], using stochastic gradient descent to optimize the Q network's approximation of the expected sum of future reinforcements.

The neural network structure used is shown in Figure 2.1. The hidden units of the neural network form a representation of the game state and action which is then combined by the output unit to approximate the Q value.

“State” refers to all distinct values for the game representation with which the reinforcement learning agent interacts. “Action” refers to the decision taken at a state, which alters the state. The function for updating the Q-value depends on the current state,  $s_t$ , the action the agent choose,  $a_t$ , the reward,  $r$ , the agent gets for choosing  $a_t$ , the subsequent state,  $s_t + 1$ , that the agent enters after taking that action, and finally the next action,  $a_t + 1$ , the agent chooses in its new state. Samples of  $(s_t, a_t, r, s_t + 1, a_t + 1)$  were collected for training the neural network.

Mini-batches of  $s_t$ ,  $a_t$ , reinforcement,  $r_{t+1}$ ,  $s_{t+1}$ , and  $a_{t+1}$  are collected. Actions are selected using the  $\epsilon$ -greedy algorithm. Stochastic Gradient Descent (SGD) was applied to train the output Q value to approximate the sum of future reinforcements. During this phase the state change outputs are ignored. The SARSA algorithm [2] is followed to form the error being minimized as summarized below.

The Q output value,  $Q(s_t, a_t)$ , that is output by the network is determined by inputs  $s_t$  and  $a_t$ .

We wish this function to form the approximation

$$Q(s_t, a_t) \approx E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\right],$$

where  $0 < \gamma < 1$  is the discount factor. Actions are chosen using an  $\epsilon$ -greedy policy  $\pi(s_t)$ ,

$$\pi(s_t) = \begin{cases} \underset{a \in A}{\operatorname{argmax}} Q(s_t, a) & \text{with probability } 1 - \epsilon, \\ z \sim \mathcal{U}(A) & \text{with probability } \epsilon \end{cases}$$

where  $z$  is a uniformly-distributed random variable drawn from the set of valid actions  $A$ .

If each mini-batch consists of  $n$  samples collected from time  $t_1$  through time  $t_n$ , then the SARSA error to be minimized for each mini-batch is

$$\sum_{t=t_1}^{t_n} (r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))^2$$

The gradient of this error for a mini-batch of  $n$  samples,  $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$  for  $t = t_1, \dots, t_n$ , with respect to the weights of the neural network drives the error minimization performed by gradient descent.

It is important to not overfit the  $Q$  function to each mini-batch of samples. Each mini-batch is a sample that is limited to the particular sequence of world states experienced. Therefore, the gradient descent algorithm is applied for a small number of iterations for each mini-batch.

### 2.1.1 DeepMind

DeepMind, a company famous for its AlphaGo program, developed their system using only raw pixels as data input. They used deep learning on a convolutional neural network, with a novel form of Q-learning. While their work was not targeted towards Dots & Boxes, it was a massive step towards developing reinforcement learning-trained agents for playing 2-player games.

In March 2016, their program AlphaGo beat Lee Sedol, a world-class player by 4-1. In 2017, an improved version, AlphaGo Zero, defeated AlphaGo 100 games to 0. Later that year, AlphaZero [5], a modified version of AlphaGo Zero but for handling any two-player game of perfect information, was able to gain a similar level of ability at chess and shogi.

AlphaGo was developed based on the deep reinforcement learning approach, playing against itself and learning from both wins and losses. AlphaGo used two deep neural networks: a value network for evaluating positions and a policy network to predict probabilities. The value network learned to predict the winners based on policy network's games, which in turn used supervised learning and was subsequently refined by policy-gradient reinforcement learning. They then used the Monte Carlo tree search with policy network predicting high-probability candidate moves and the value network evaluated positions.[59]

AlphaGo Zero trained using reinforcement learning by playing games against itself, without learning from games played by humans. It used one neural network instead of separate policy and value networks, with a simplified tree search to predict positions and sample moves.

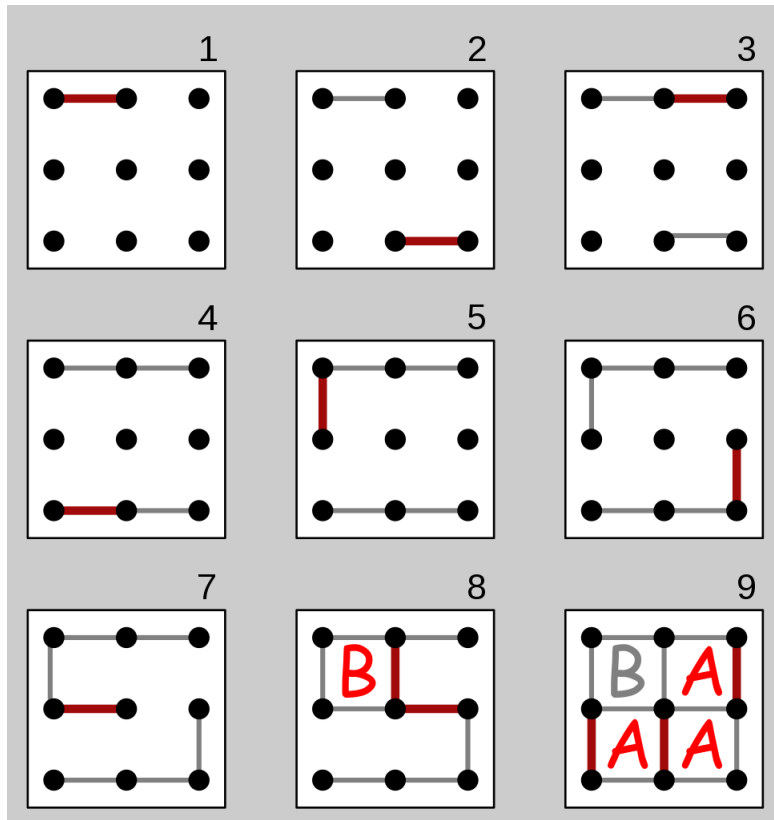
## **2.2 Dots & Boxes**

Dots and boxes is a 2-player game. Figure 2.2 illustrates a 2x2-sized game. The initial state is an empty grid of dots. Both players take turns making a move; a move consists of adding either a horizontal or vertical line between two non-joined adjacent dots. If making a move completes a 1x1 box, then the player who made that move wins that particular box (essentially, gets a point); the player also retains their turn. The game ends when there are no more available moves left to make. The player with the most completed boxes at the end is the winner of the game.

In combinatorial game theory, Dots and Boxes is an impartial game. However, Dots and Boxes lacks the normal play convention of most impartial games (where the last player to move wins), which complicates the analysis considerably [6].

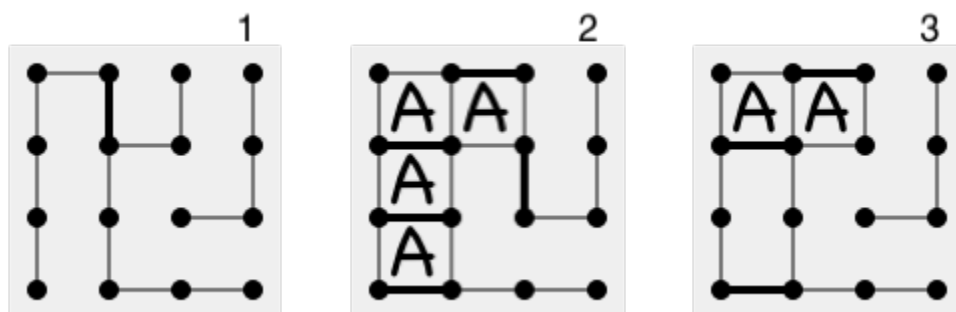
The reason for this is the quirk of the game that a player retains their turn when they complete a box. A common technique used by more experienced players of the game is the double-cross





**Figure 2.2:** A 2x2 dots & boxes game [1]. Red edges are moves made by Player A and gray edges are moves made by Player B. In this example, Player A wins with 3 completed boxes, while Player B only has 1 completed box.

strategy. Since completing a box causes a player to retain their turn, this can be used to force a player to enable a longer chain of boxes for the other player, thereby losing the game.



**Figure 2.3:** Double Cross strategy [1]

In Figure 2.3, faced with position 1 (latest move by player B in bold), a novice player would proceed to complete all boxes in the chain, one after another. At the end of the chain, having completed a box, they would have retained their turn, and are therefore forced to make a move, creating position 2. Now player B would complete the other chain, and player A would lose the game. However, an experienced player would create position 3 instead, where the last 2 boxes of the first chain are deliberately sacrificed. In this case, player B would complete the last 2 boxes, but also retain their turn, and they would be forced to enable the longer chain instead, thereby causing player A to win.

### **2.2.1 Related work**

Like many other games, dots and boxes is a combinatorial game and can be solved mathematically. However, the rules make the state space extremely large to search within for a perfect solution within a reasonable amount of time. People have made attempts to reduce this search space [7] and thus make it more feasible to solve the game. Others have attempted to solve the game of dots and boxes using reinforcement learning [8–10] with artificial neural networks to further improve the rate at which a solution is discovered.

Barker and Korf [7] attempted to solve the game of Dots and Boxes by optimizing the search technique. They used an Alpha-Beta minimax search to reduce the state space significantly. Similar to other 2-player games such as Tic Tac Toe, Dots & Boxes also has the occurrence of "chains" of certain structure during intermediate stages of the game, incomplete boxes with 2 edges filled. They made use of this observation to help their search algorithm. Using their work, they were able to solve a 4x5 sized game within a reasonable amount of time.

Zhuang et al. [10] developed an agent called QDab in 2015 to play Dots and Boxes on a 5x5 sized game. They made use of a brand new game representation, using strings and coins, in order to reduce the search space drastically and then using an artificial neural network in order to learn the optimal move using backpropagation.

The strings and coins representation works by transforming the game into a different representation. Each potential box is represented using a coin, and it has 4 strings attached to it, which represent the unmade edges. Once an edge is made/created, the "string" is cut, and once all strings to a coin are cut, the player making the final cut gets the coin. The benefit of this representation was that the game board could be represented as being constructed from a finite set of 12 basic chain structures.

They analyzed the properties of the chains to determine the ones which were preferable, and created reinforcements to optimize which moves were preferable for each chain structure. Alongside using a greedy policy, they also made use of min-max searches close to the game end state to definitively compute the best move in a reasonable amount of time. Additionally, they made use of random pruning of available moves towards the start of the game to reduce the state space, with the assumption that enough good moves would be left over (however, they did not provide any proof towards the same).

Using a combination of the above, the QDab agent performed excellently compared to other existing agents. It took nearly 20 seconds to compute the steps necessary to play the entire game.

In the work done by Miller et al. [8], they expanded on the approach made by Zhuang et al. and attempted to solve a 6x6-sized game of dots and boxes using reinforcement learning. They used the Monte-Carlo tree search approach, as well as the strings and coins representation method of the game.

They designed and trained 4 separate agents, each with a different reward system. Their best agent was able to do well enough against an opponent which was randomly selecting moves, however, it was far from being able to beat a human during testing.

In a different approach adopted by Deakos Matthew [9], they made use of convolutional neural networks for training the agent to play a game of 5x5 instead. Making use of 2 convolutional layers and 2 fully connected layers, they designed a system in which an agent would play against itself, and the "opponent" agent would be periodically updated with the learning agent's parameters, in order to develop a consistently improving opponent to play against.

Their initial training was similar to brute-forcing against a random opponent, however, progress was slow. The basic approach was modified to look at the next state, and determine if it was won by the opponent, which helped the convergence significantly. This approach was used to train with a million games in order to train the model. By utilizing this approach, the agent was able to achieve a win rate of over 95% against the random opponent after 50 thousand games, and over 90% after just 10000 games. This work was most closely comparable to the work done in this thesis, and the results will be compared and discussed in the Conclusion chapter.

# Chapter 3

## Methodology

This chapter outlines the approach taken to train an agent using reinforcement learning. It elaborates on the specifics, such as the game representation used for training, the training process and so on. Finally, it touches on the initial missteps that were made when trying to train the agent.

### 3.1 Game Representation

Determining how to store and represent the game is a bit tricky, since both the dots and their intermediate edges are valid to the game state. One can, however, observe that the dots are constant for every state. Hence, a game state can be represented solely by its edges. All edges in the game are represented as a list (of length 24, since there are 24 edges in a 3x3 size game), with 0 denoting that an edge does not exist, and 1 denoting otherwise.

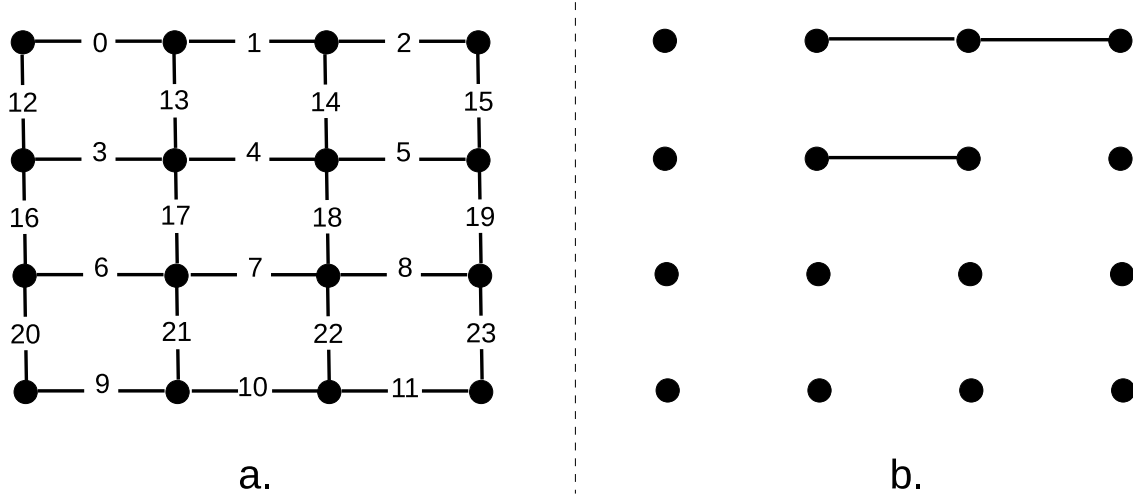
The edge ordering being considered is shown in Figure 3.1a. For example, the game state shown in Figure 3.1b is represented as  $[0, 1, 1, 0, 1, 0, \dots, 0]$ .

The action was represented as a one-hot encoding in a 24-size list, with the index of the new edge being set to 1. An action could be taken only if the corresponding edge index was 0. For example, if edge 1 was 0, and a player wanted to create that edge, the action value would be represented as  $[0, 1, 0, 0, \dots, 0]$ .

### 3.2 Reinforcement Learning

The training begins by creating a neural network. The network has 48 inputs, and 1 output. The 48 inputs are formed by concatenating the state  $s$  (represented as a 24-size vector) and one-hot encoded representation of the move taken  $a$  (also represented as a 24-size vector).

The hidden layers of the network are represented using a list, where the value at each index in the list informs the number of hidden units in the layer. For example,  $[20, 10]$  describes a network with 2 hidden layers, where the first hidden layer has 20 hidden units and the second hidden layer



**Figure 3.1:** Game state representation: a.) edge ordering, b.) example state, represented as  $[0, 1, 1, 0, 1, 0 \dots, 0]$

has 10 hidden units. As another example,  $[30, 40, 50]$  describes a network with 3 hidden layers, hidden layer 1 with 3 hidden units, hidden layer 2 with 40 hidden units and hidden layer 3 with 50 hidden units.

The probability of the agent making a random move,  $\epsilon$  is determined for every epoch. The initial value is set to 1, to allow complete randomness in the beginning, and is slowly decayed by a constant factor at each epoch during training. A minimum threshold is set for the probability of randomness, beyond which the probability does not decrease. This threshold is min epsilon.

Each repetition in an epoch constitutes of playing an entire game. The game starts with player 1, the agent's turn. The move to make is determined using the aforementioned  $\epsilon$ -greedy policy, and the next state of the game is compared to the list of existing boxes to determine if a box is created. If no boxes are created, the player turn is flipped, otherwise the player retains their turn.

Samples are aggregated for all turns of the agent for the games, including the reinforcement value  $r_t$  for each state  $s_t$ .  $r_t$  is defined as:

$$r_t = \begin{cases} 1 & \text{if game is over and agent **won**,} \\ -1 & \text{if game is over and agent **lost**,} \\ 0 & \text{if game is not over.} \end{cases}$$

Note that the reinforcement should also be 0 for games which are drawn, but during this research a 3x3 sized-board was used, with 9 boxes, hence a draw outcome was not possible.

The determination of winning and losing can only be done when the game ends, the winner being the player who created more boxes throughout the game. Hence, a myopic move might be globally sub-optimal. The combination of global goal optimization and non-alternating turns makes the game more complex.

The samples are used for training the neural network, and another parameter trains determines the size of the mini-batches to be used for training. Two other boolean parameters, SGD and ReLU are used to determine the optimizer and activation function respectively. Setting SGD to false resulted in the Adam optimizer being used and setting ReLU to false resulted in hyperbolic tangent being used as the activation function.

During testing, the trained network is used to play against a random opponent, but no samples are collected and the network is not updated.

For further training of the network, instead of training a new network from scratch, the network was instead used with the  $\epsilon$ -greedy policy.  $\epsilon$  was set to 0.01 from the beginning, and not varied during the training. Thus, the probability of making a random move was 1 in 100 from the beginning for further training.

### 3.3 Training

During training, several hyperparameters were varied in order to experimentally determine the best results:

- Epochs: Determined the number of times the samples were recollected from scratch

- Games per Epoch: Determined the number of games for each epoch; all games within an epoch had the same measure of randomness for the agent (determined by  $\epsilon$ )
- Network structure: Listed the configuration of the network
- Updates per Epoch: Number of times network was updated while training at each epoch
- Learning rate: Determined the rate at which the learning was performed
- Activation function: TanH (hyperbolic tangents) or ReLU (rectified linear units)
- Optimizer: Adam or SGD (Stochastic Gradient Descent)
- Epsilon Decay Factor: The rate at which the degree of randomness was reduced between each epoch as a multiplicative factor
- Min Epsilon: This was introduced to maintain a minimum threshold of randomness for the entirety of the training

A lot of hyperparameter tuning was done based on empirical observations. First, a base set of good hyperparameter values was determined by training against a simple opponent. A simple opponent would pick the first available move from the list of available moves. By having such an opponent, some observations were generalized, which held true across all experiments. For instance, a 2 layer network with 100-500 units worked out better than 3 layer networks, or 2 layer networks which were even wider.

After training an agent which did well against a simple opponent, and discovering a good set of hyperparameter ranges, the agent was next trained to perform against a random opponent. A random opponent would select a move at random from the list of available moves. Some of the previously determined observations for hyperparameters held true, such as network structure, while others had to be revised, such as the learning rate. After much experimentation, a good set of hyperparameters were discovered, which allowed for consistent wins against a random opponent. After this, the predictions and trapping capabilities of the random opponent were tested and the



agent was not able to perform trapping. In an attempt to solve this, the agent was trained much further, with a varying degrees of randomness.

Finally, a new agent was trained from scratch against the previously trained agent. The hope was that training against a more complex opponent would help the agent learn a more complex game strategy. To do so, the orientation of the new agent was set as the opposite of the previous agent (learning to play as player 2, if agent 1 learned to play as player 1). The new agent learned to perform optimally against the previous agent.

### **3.3.1 Initial missteps**

Initially, the network has 25 inputs only, 24 representing the state of the game and 1 value representing the action, an integer from  $[0, 23]$ , denoting the index where the action would be taken.

However, the model was becoming very strongly biased during training towards the action input, even after standardizing the action to be between the range of  $[0, 1]$ . This bias was hindering the agent from learning and the win rates would consistently hover around 50%, with no indication of improvement.

Thus, the methodology was altered to have the action represented as a 1-hot encoded vector instead. This solved the issue and the agent began to learn more from the state of the game instead of zeroing in on the action taken.

The facet of non-standard turn alteration posed a problem during training. While determining the turn could be implemented by checking if a box was being created compared to the previous state, selecting the correct samples would be a misstep.

The collected samples would have to be at the next turn of the agent, and would have the reinforcement and the Q value predicted at that stage, and compare it to the previous stage when the agent had a turn. The number of steps between the two stages when agent had a turn could be variable, depending on how many boxes the opponent player made during their turn.

To fix this, the state and action for the agent would be stored temporarily, and when the agent gets their turn next, the temporarily stored values would be used to collect the samples correctly.

# Chapter 4

## Results

In this chapter, the results at each step in the process are discussed in detail, from the agent that plays against a basic player to the one that is able to handily beat a somewhat strategic player. The predicted moves at intermediate stages are demonstrated, along with analyzing whether the agent was able to execute a complex trapping strategy. Lastly, an entire game played by the agent was observed and analyzed.

### 4.1 Simple opponent

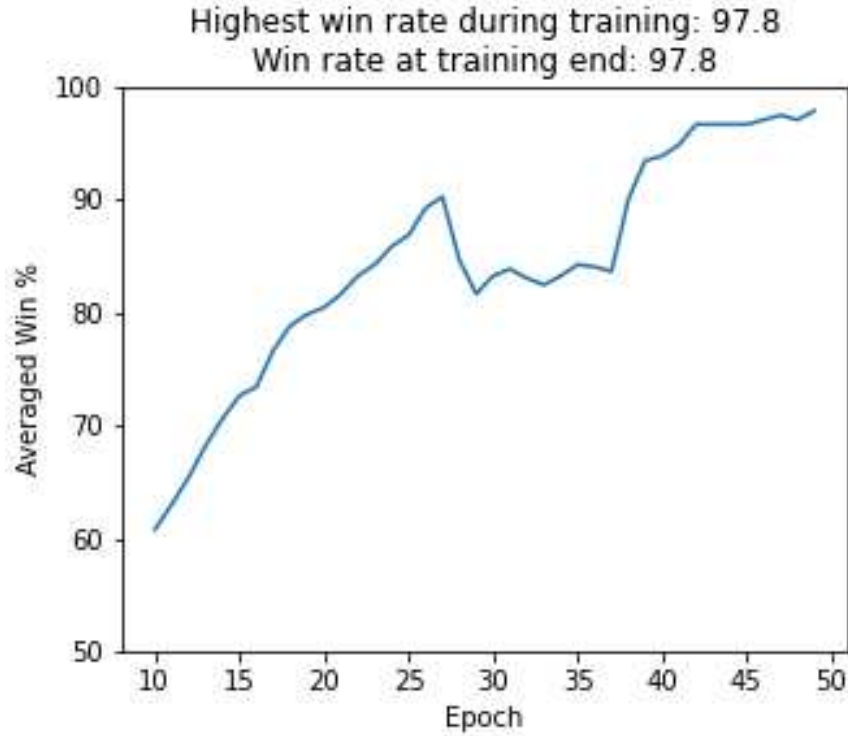
The base approach was against a simple opponent in order to help train the agent, such as determining the samples collected and tuning the agent's game-playing ability. It also helped determine a set of hyperparameters which worked well for training.

#### 4.1.1 Training

Training against a simple opponent, for different parameters, such as the network structure, the number of epochs and games per epoch, optimizers, activation functions, the most successful network had a win rate of over 95%. The curvature for the win rate was stable, and to get a better idea of sustained performance, binning for previous 10 epoch wins was done and plotted.

While the performance of the agent varied across different hyperparameters, there were a few commonalities observed:

- **Network structure:** 2 layers with 100 units in each layer worked best; using wider (more units), shallower (fewer layers) networks did not work as well, and deeper networks with more layers also overtrained on mini-batches, leading to sharp declines in performance after gradual improvement when training;



**Figure 4.1:** Win rate against simple player during training

- **Learning rate:** A learning rate of the order  $10^{-3}$  led to the most stable training performance; decreasing the learning rate to  $10^{-4}$  impacted training speed severely and increasing it to  $10^{-2}$  made the learning highly unstable;
- **Activation function:** The hyperbolic tangent *TanH* activation function worked better than Rectified Linear Unit *ReLU* nearly always;
- **Optimizer:** SGD worked better than Adam, with smoother improvement and better win rate on average; using Nesterov momentum with SGD helped even further.

Figure 4.1 shows the win rate of the agent against a simple player. The hyperparameters for the run are listed in Table 4.1.

### 4.1.2 Testing

Once the trained agent was tested against a simple player, the agent performed perfectly, achieving a win rate of 100% in all 10 instances. Figure 4.2 shows win rate of the trained agent

**Table 4.1:** Hyperparameter values for best network against simple opponent

Hyperparameter	Value
Epochs	50
Games per Epoch	50
Network structure	[100, 100]
Updates per Epoch	25
Learning Rate	0.001
Activation Function	TanH
Optimizer	SGD
Epsilon Decay Factor	0.92

against a simple player for 10 tests. Each test averaged performance for 100 games over 5 runs. The win rate of the agent was a staggering 100%.

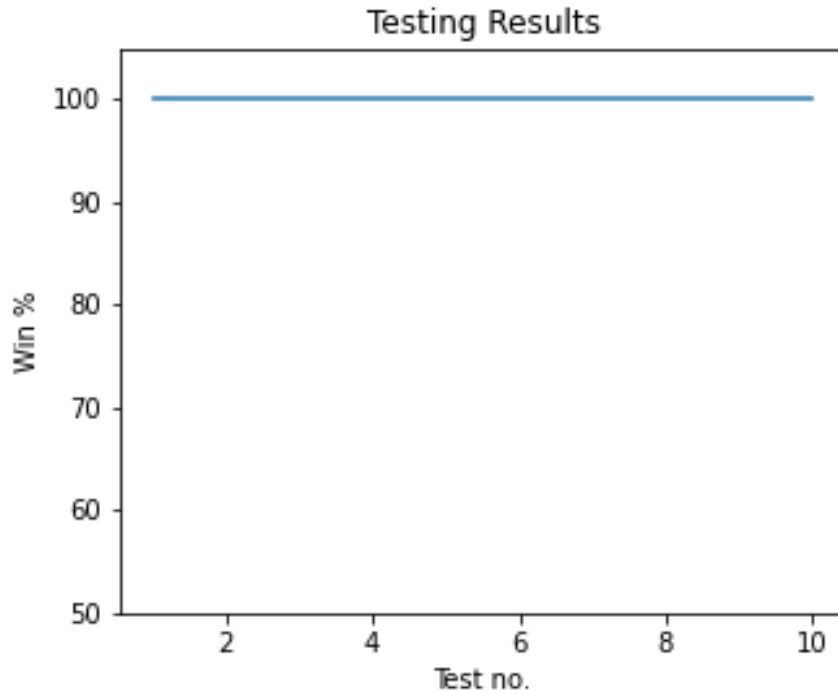
## 4.2 Random opponent

Playing against a simple opponent only helped the agent learn a very specific strategy, and was barely usable against an actual player. The hope was that training against a random opponent, the agent would learn to play much better in general.

### 4.2.1 Training

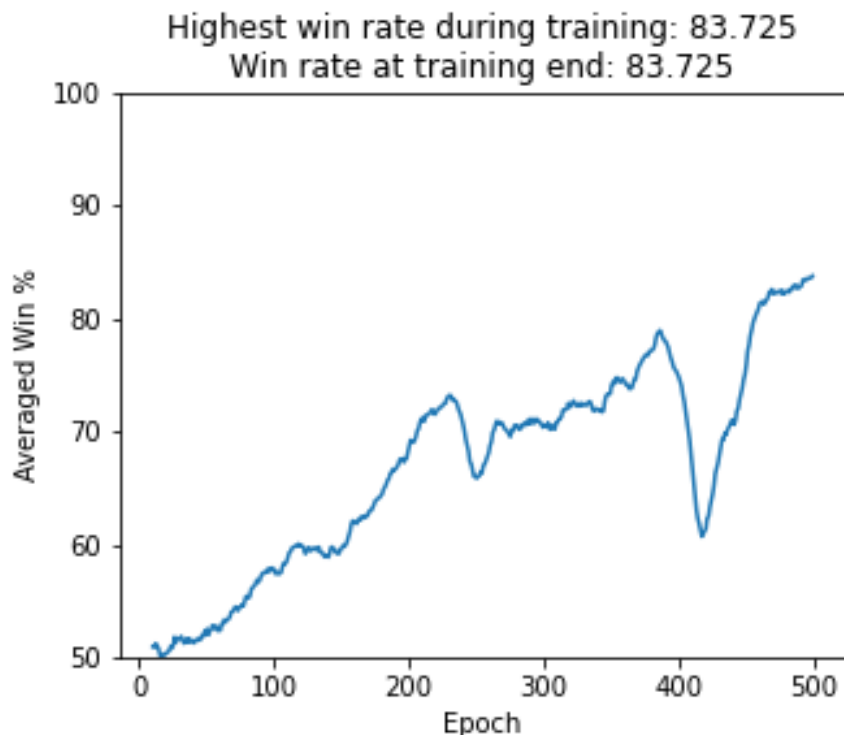
Training against a random player, for different parameters, such as the network structure, the number of epochs and games per epoch, optimizers, activation functions, the most successful network had a win rate of over 80%. The curvature for the win rate was unstable in this case, and binning for previous 10 epoch wins provided much more coherent performance insight. The number of runs required compared to train a simple player was far greater, and setting the minimum  $\epsilon$  to 0.01 (1 in 100 chance of randomness) resulted in overfitting during the latter epochs. Hence, a new hyperparameter was introduced, *Min Epsilon*, in order to vary and test the minimum randomness preventing overfitting.

While the performance of the agent varied across different hyperparameters, there were a few commonalities observed:



**Figure 4.2:** Win rate against simple player during testing

- **Network structure:** 2 layers with 100-250 units in each layer worked best; using wider (more units), shallower (fewer layers) networks did not work as well, and deeper networks with more layers also overtrained on mini-batches, leading to sharp declines in performance after gradual improvement when training;
- **Learning rate:** A learning rate of the order  $10^{-3}$  led to the most stable training performance; decreasing the learning rate to  $10^{-4}$  impacted training speed severely and increasing it to  $10^{-2}$  made the learning highly unstable;
- **Activation function:** The hyperbolic tangent *TanH* activation function worked better than Rectified Linear Unit *ReLU* nearly always;
- **Optimizer:** SGD worked better than Adam, with smoother improvement and better win rate on average; using Nesterov momentum with SGD helped even further;



**Figure 4.3:** Win rate against random player during training

- **Min Epsilon:** Setting min  $\epsilon$  to 0.01 or even 0.02 resulted in overfitting and sharp declines in performance in the later epochs and a value higher than 0.075 resulted in the win rate decreasing; a min  $\epsilon$  value of 0.05 mitigated the training overfitting issue without compromising win rate

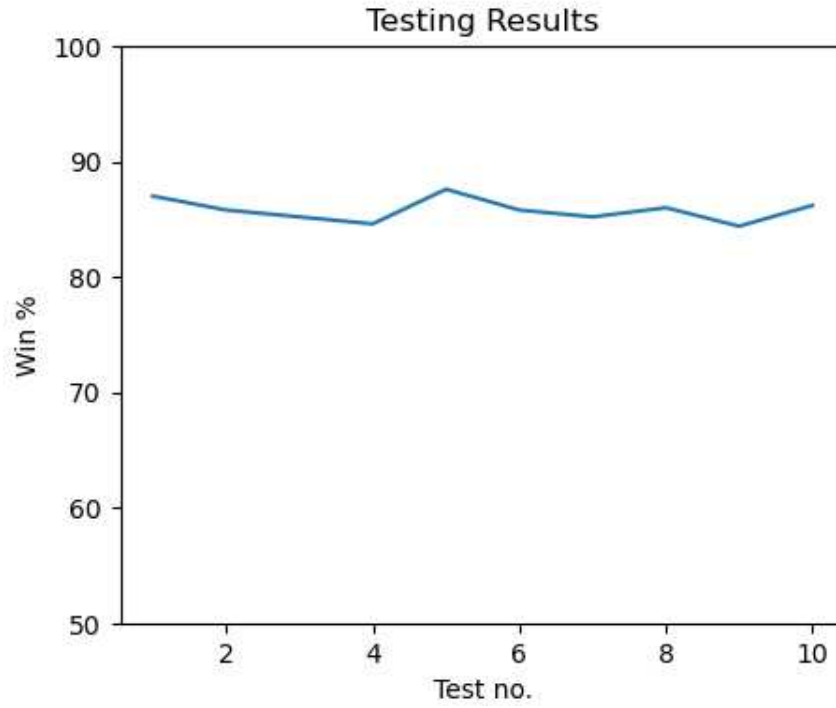
Figure 4.3 shows the win rate of the agent against a random player. The hyperparameters for the run are listed in Table 4.2.

### 4.2.2 Testing

Once the trained agent was tested against a random player, the agent performed very well, achieving a win rate consistently over 80%. Figure 4.4 shows win rate of the trained agent against a random player for 10 tests. Each test averaged performance for 100 games over 5 runs. The performance of the agent was over 80%, and all tests except 2 did better than 85%.

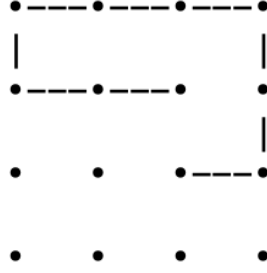
**Table 4.2:** Hyperparameter values for best network against random opponent

Hyperparameter	Value
Epochs	500
Games per Epoch	800
Network structure	[100, 100]
Updates per Epoch	50
Learning Rate	0.001
Activation Function	TanH
Optimizer	SGD
Epsilon Decay Factor	0.9925
Min Epsilon	0.05

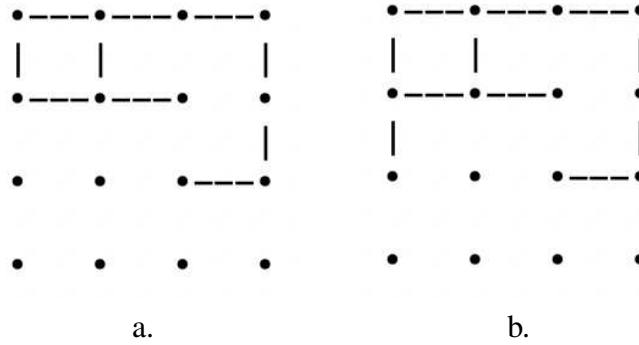


**Figure 4.4:** Win rate against random player during testing





**Figure 4.5:** Intermediate state of the game



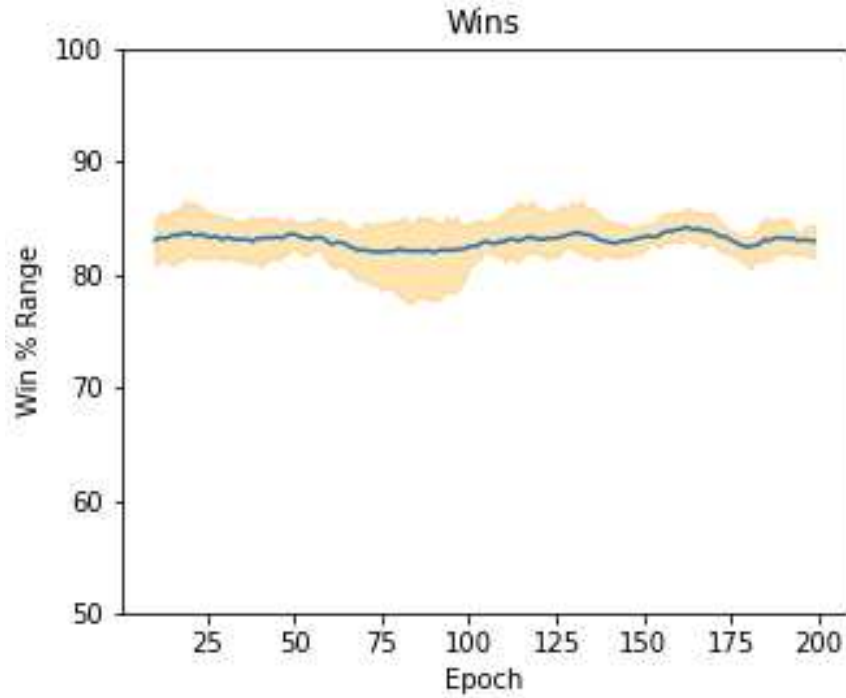
**Figure 4.6:** a.) First action taken by agent for state in Fig. 4.5. b.) Second action taken by agent.

### 4.2.3 Predicted moves

When observing an intermediate state, the agent's move can be observed from Figure 4.6a, which depicts the move made by the agent from the state in Figure 4.5. Assuming Figure 4.6a is the state at which agent has a turn instead, the agent takes an unexpected move, which can be observed from Figure 4.6b. Since the moves are non-alternating, it is possible for the agent to come across either of states in Figure 4.5 or Figure 4.6a. The agent's action choice observed in Figure 4.6b is likely a result of pursuing the globally optimal goal of winning more boxes overall.

### 4.2.4 More training

Since there was the possibility of further training improving the performance of the agent, the network was trained further in order to observe the effects. However, further training of the network did not yield an improvement in performance against a random player, as observed in Figure 4.7. The Win % is averaged into bins of 10 epochs.



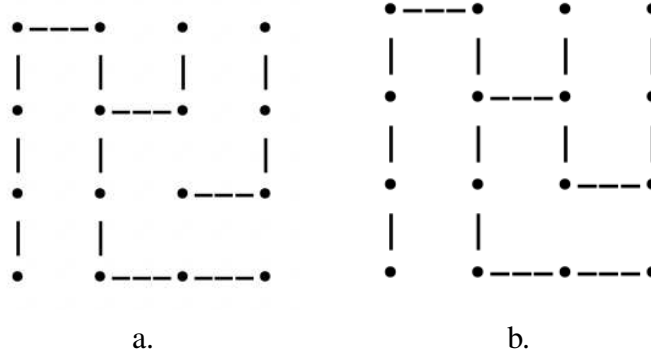
**Figure 4.7:** Performance of agent during further training. The line represents the average of all runs, while the shaded area shows the upper and lower bounds of performance of the agent across all runs.

An attempt was also made to set  $\epsilon$  to 1 again and then decay it, in order to observe the effect of additional exploration with a trained network, but the performance at the beginning plummeted and did not surpass the performance at the start of this additional training.

Figure 4.7 shows the win rate of the agent against a random player averaged over 5 runs. As can be observed, while the performance of the agent was over 80%, the win rate did not increase, but in fact decreased slightly.

### 4.2.5 Trapping

We attempted to observe if the trapping strategy mentioned in the game background section was being replicated by the agent, and found out this was not the case, as observed in Figure 4.8b. The agent did not transition to either states 2 or 3 in Figure 2.3, and lost the game instead. It is possible that this is due to being trained against a random player.



**Figure 4.8:** a.) Position 1 in the double cross strategy depicted in Fig. 2.3. b.) Next action taken by agent.

## 4.3 Against itself

While playing against a random player helped the player develop a more complex strategy, it still lacked the intelligence to play against a smart opponent. Hence, the agent was trained to play against itself, in the hope that playing against a trained agent would help the new agent develop more complex game strategy.

### 4.3.1 Training

Training against a random player, for different parameters, such as the network structure, the number of epochs and games per epoch, optimizers, activation functions, the most successful network had a win rate of over 99%. The curvature for the win rate was *extremely* unstable in this case. While previous iterations seemed to improve but suffer from brief bouts of decline due to overfitting, in case of training against itself, once performance started to decline, it either skyrocketed or crashed to zero. Binning for previous 10 epoch wins was done to standardize performance visualization against previous results.

The number of runs required compared to train a random player was fewer, but the batches were overfitting very sharply, no matter the variation in hyperparameters. To reduce this, the number of epochs was increased, and number of games per epoch was reduced significantly. Setting the minimum  $\epsilon$  to 0.05 (5 in 100 chance of randomness) resulted in the training going off kilter again and again and crashing to 0. Hence, the minimum  $\epsilon$  was set to 0.01.

While the performance of the agent varied across different hyperparameters, there were a few commonalities observed:

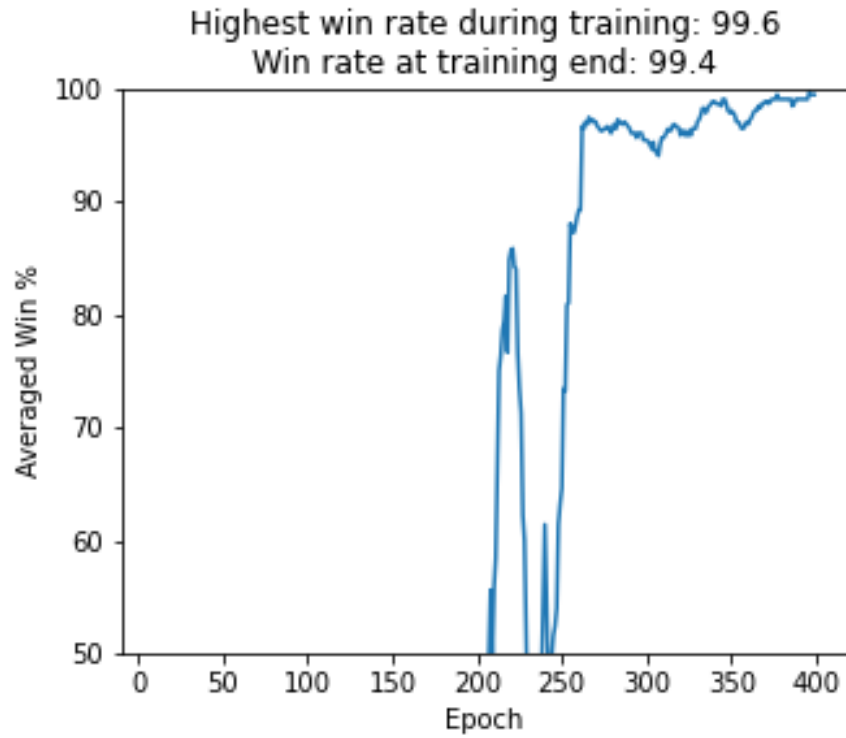
- **Network structure:** 2 layers with 100 units in each layer worked best; using wider (more units), shallower (fewer layers) networks worked extremely poorly, and deeper networks with more layers overtrained almost instantly, reducing win rates to near zero;
- **Learning rate:** A learning rate of the order  $10^{-3}$  led to the most stable training performance; decreasing the learning rate to  $10^{-4}$  impacted training speed severely and increasing it to  $10^{-2}$  made the learning highly unstable;
- **Activation function:** The hyperbolic tangent *TanH* activation function worked better than Rectified Linear Unit *ReLU* nearly always;
- **Optimizer:** SGD worked better than Adam, with smoother improvement and better win rate on average; using Nesterov momentum with SGD helped even further;
- **Min Epsilon:** Setting  $\min \epsilon$  to 0.05 resulted in the training increasing performance gradually, then going the opposite direction sharply, ending up with a win rate below 5%; setting the minimum  $\epsilon$  to 0.01 seemed to prevent this.

Figure 4.9 shows the win rate of the agent against the previously trained agent. The hyperparameters for the run are listed in Table 4.3.

### 4.3.2 Testing

Once the trained agent was tested against the previous agent, the new agent performed perfectly, achieving a win rate of 100% in all 10 instances. Figure 4.10 shows win rate of the newly trained agent against a trained player for 10 tests. Each test averaged performance for 100 games over 5 runs. The win rate of the agent was a staggering 100%.

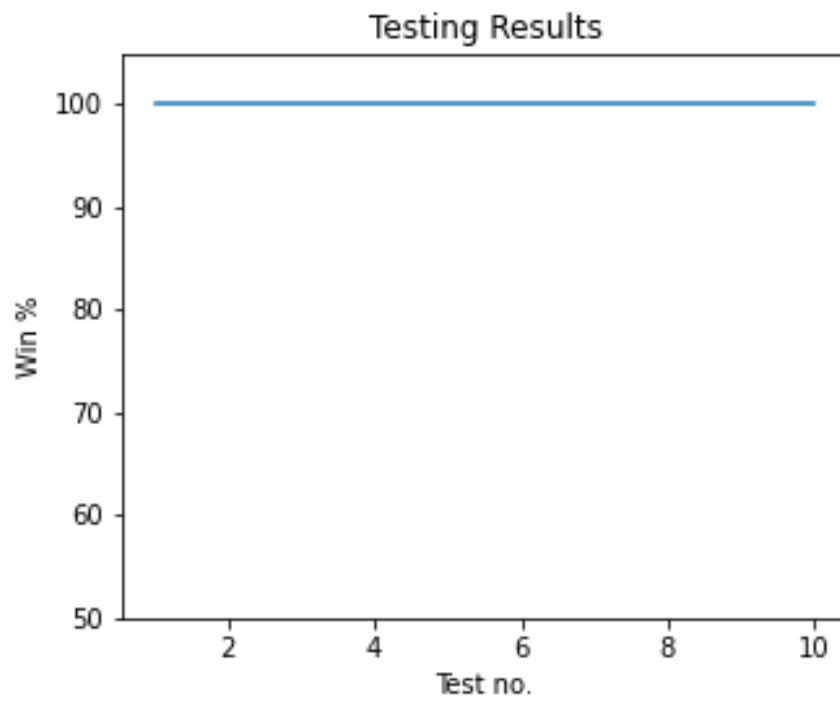
Attempting to test the new agent against a random opponent instead, the performance dropped to 55%. This is not surprising, given that once the new agent learns to play against some semblance



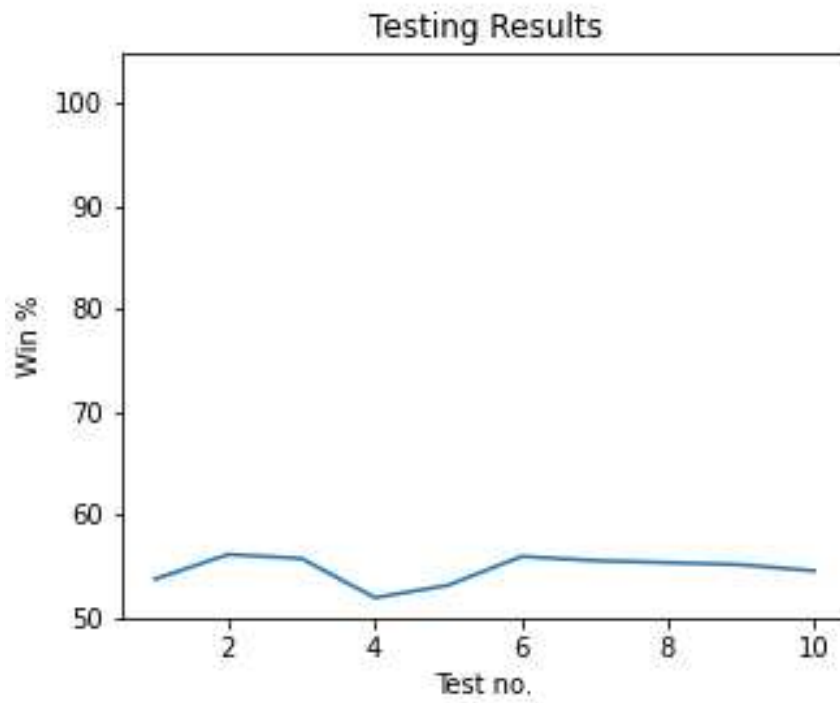
**Figure 4.9:** Win rate against intelligent player during training

**Table 4.3:** Hyperparameter values for best network against intelligent opponent

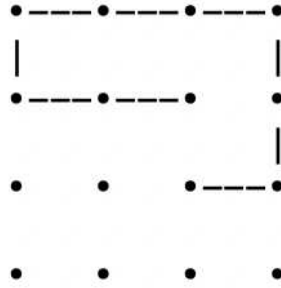
Hyperparameter	Value
Epochs	400
Games per Epoch	50
Network structure	[100, 100]
Updates per Epoch	100
Learning Rate	0.001
Activation Function	TanH
Optimizer	SGD
Epsilon Decay Factor	0.9875
Min Epsilon	0.01



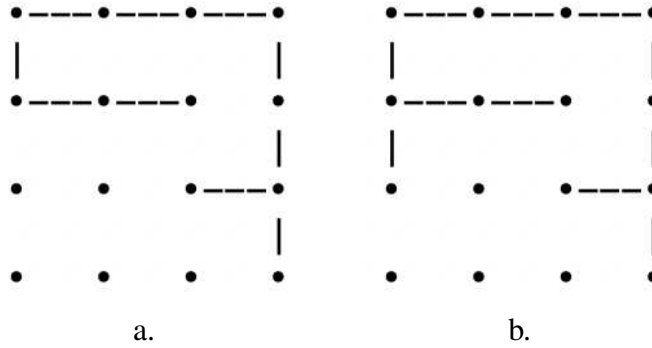
**Figure 4.10:** Win rate against trained player during testing



**Figure 4.11:** Win rate against random player during testing



**Figure 4.12:** Intermediate state of the game



**Figure 4.13:** a.) First action taken by agent for state in Fig. 4.12. b.) Second action taken by agent.

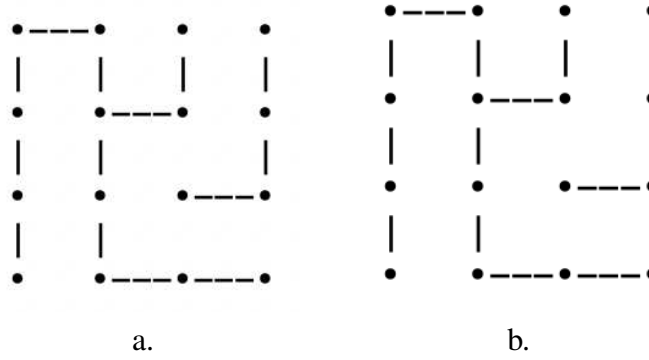
of strategy, a completely random player would throw off the effectiveness of its newly learned optimal strategy.

Figure 4.11 shows win rate of the newly trained agent against a random player for 10 tests. Each test averaged performance for 100 games over 5 runs. The win rate of the agent averaged around 55%, and did not drop below 50% in any instance.

### 4.3.3 Predicted moves

When observing an intermediate state, the agent takes a seemingly non-optimal move. This can be observed from Figure 4.13a, which depicts the move made by the agent from the state in Figure 4.12.

Assuming Figure 4.13a is the state at which agent has a turn instead, the agent takes another seemingly non-optimal move, which can be observed from Figure 4.13b. It is possible that these



**Figure 4.14:** a.) Position 1 in the double cross strategy depicted in Fig. 2.3. b.) Next action taken by agent.

locally non-optimal moves are globally optimal, since the result of dots and boxes depends on which player has more boxes at the end of the game.

#### 4.3.4 Trapping

We attempted to observe if the trapping strategy mentioned in the game background section was being replicated by the agent, and found out this was not the case, as observed in Figure 4.8b. The agent did not transition to either states 2 or 3 in Figure 2.3, and lost the game instead. It is likely that training an agent against a previously trained agent is not enough for the agent to learn such a subtle strategy, and multiple rounds of training against previous best agent might be necessary for the agent to learn it.

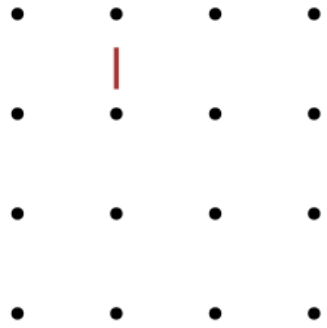
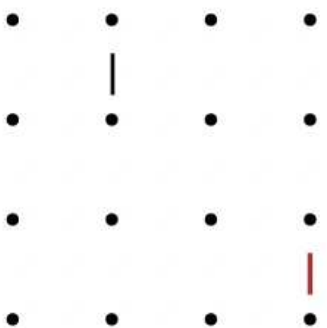
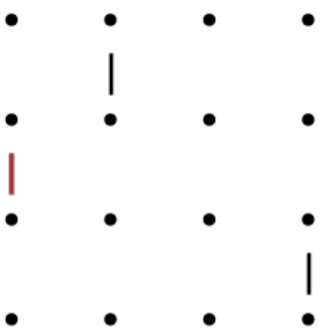
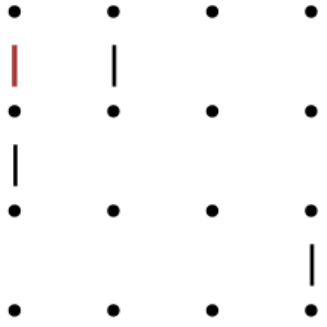
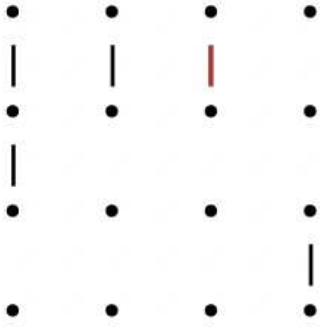
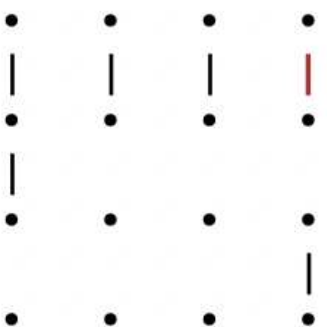
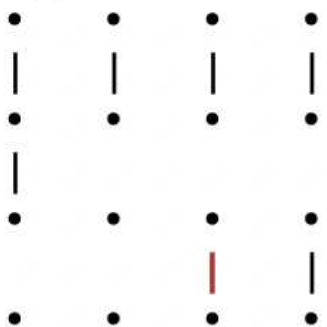
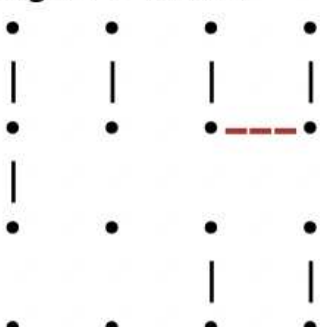
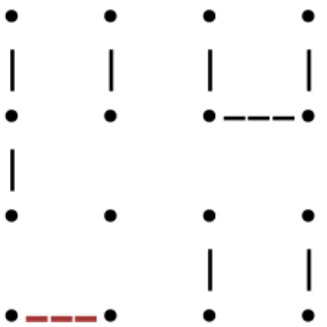
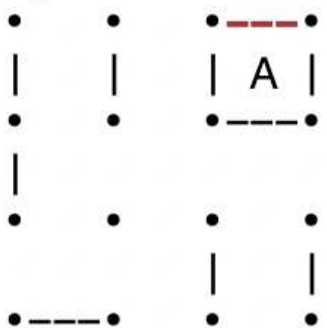
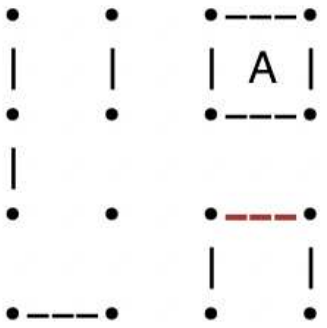
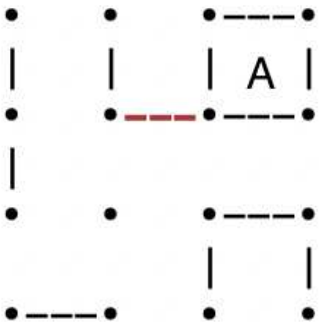
#### 4.3.5 Game-play analysis

An analysis of the performance of the agent is done for a sample game. The agent won the game, winning 6 of the 9 total boxes.

Since the agent trained as player 2, the previous-agent-as-opponent makes the first move. On step 8, the agent make a 3rd side to the box, a poor move strategically, and one that any human opponent would easily capitalize on. The agent does complete the box in the very next turn, however. In step 11, the agent makes a similar move, but this time does not complete the box in the next available turn. Furthermore, it does not complete open/available boxes in step 13. Starting from step 15, the agent quickly takes all available boxes, and then adds a 3rd line to the last open



**Table 4.4:** Game moves (sequential). Play proceeds left to right across a row and continues on the next row.

<p>Opponent move</p> 	<p>Agent move</p> 	<p>Opponent move</p> 
<p>Agent move</p> 	<p>Opponent move</p> 	<p>Agent move</p> 
<p>Opponent move</p> 	<p>Agent move</p> 	<p>Opponent move</p> 
<p>Agent move</p> 	<p>Agent move</p> 	<p>Opponent move</p> 

**Table 4.5:** Game moves (continued) (sequential)

<p>Agent move</p>	<p>Opponent move</p>	<p>Agent move</p>
<p>Agent move</p>	<p>Agent move</p>	<p>Agent move</p>
<p>Opponent move</p>	<p>Opponent move</p>	<p>Agent move</p>
<p>Agent move</p>	<p>Agent move</p>	<p>Opponent move</p>

box in the chain. The opponent capitalizes on this and takes its first box. Further opportunity arrives at step 21, when agent takes another 2 boxes. Somewhat disappointingly, on step 23, the agent chooses the incorrect move, conceding the last 2 boxes to the opponent instead of winning them itself.

Based on the behavior of the agent, it seems far more inclined towards completing a 3rd side of a box than it should be. My speculation is that this is because the agent learned against an agent which itself learned to play against a random opponent. Due to this, the agent learned to try to complete boxes by itself, including boxes with only 2 sides by making a 3rd side. Since the opponent was also random initially, it didn't capitalize on each such opportunity and make the agent lose games, which is why this strategy likely remained ingrained within the agent. I believe further training against existing agents as opponents would help the agent learn that such moves are a poor strategic choice, as the likelihood of opponents capitalizing on such a mistake will increase with each iteration of the opponent.

# Chapter 5

## Conclusion

This research demonstrates that playing a game of Dots and Boxes can be successfully implemented using reinforcement learning. The game is translated to state and action and fed as input to a neural network, and the sum of future reinforcements is predicted and improved upon.

The agent improves significantly during training against a random player, despite the complexity of the game. The testing results reinforce the same, with the agent beating a random player handily. Further training of the agent, even with varying parameters, did not lead to any gains, and the agent could not breach the 90% win rate barrier, indicating that this might be the best that an agent can do against a random player using the current approach. The agent's actions in pursuing a globally optimal goal sometimes led to actions which were not myopic.

However, the agent was unable to execute the double trapping strategy, likely due to never encountering it properly when playing against a random player. Furthermore, defeating a random player is not an accurate metric of game strategy.

To that end, an exploration was done by having the agent train against itself in the hope that training against an intelligent opponent would help it learn a deeper level of strategy. Nevertheless, the revised agent was still unable to pick up on the double cross strategy, showing that the new training was not sufficient.

It is likely that many iterations of training against a previous version will enable the agent to learn to play the game with a significant level of intelligence.

### 5.1 Comparison against related work

The work done by Deakos Matthew [9] was most closely related to the work done in this thesis. They also trained their agent against an opponent agent, albeit using a DQN approach and with opponent agent updating mid-training.

The network used by Deakos was far larger, with 2 convolutional layers of 16 filters and 32 filters respectively, each with a 3x3 kernel and then 2 fully connected layers with 256 units each. The game board they trained on, however, was also larger at 4x4.

They were able to achieve a win rate of over 90% against a random agent after 10000 games after implementing their modification, which looked at next state of the game and whether that was won by opponent, which is impressive. The agent in this thesis took nearly 400 thousand games in order to converge on a win rate of over 80%.

## **5.2 Future work**

Another aspect of exploration would be to train the agent using images of the game instead, similar to DeepMind's approach [5]. This would be a massive paradigm change, and require a large amount of training data generation, but could yield other benefits. Using a convolutional neural network for training could significantly reduce overfitting during training, which was a persistent problem while training the agent and understandably so, given the sheer size of the state space.

Further exploration could also be attempted to use a different Q-learning approach, for instance, using Double Deep Q Networks, with two different functions for training and selecting actions.

# Bibliography

- [1] Wikipedia contributors. Dots and Boxes. [https://en.wikipedia.org/wiki/Dots\\_and\\_Boxes](https://en.wikipedia.org/wiki/Dots_and_Boxes), October 2019.
- [2] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*, volume 2. The MIT Press Cambridge, Massachusetts London, England, 2018.
- [3] C. Anderson. *Learning and Problem Solving with Multilayered Connectionist Systems*. PhD thesis, University of Massachusetts, Amherst, MA, 1986.
- [4] G. Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.
- [5] 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. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.
- [6] J. West. *Games of No Chance*, volume 29, pages 79–84. MSRI Publications, 1996.
- [7] J. Barker and R. Korf. Solving 4x5 dots-and-boxes. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25(1):1756–1757, Aug. 2011.
- [8] A. Miller, N. Phung and D. Dowling. Fun with Dots and Boxes: An Approach Using Q-learning and Artificial Neural Networks. [https://github.com/phung025/Dots\\_and\\_Boxes\\_RL](https://github.com/phung025/Dots_and_Boxes_RL), June 2021.
- [9] M. Deakos. A Deep Reinforcement Learning Approach to "Dots and Boxes". <https://github.com/mattdeak/dots-boxes-RL>, June 2021.

- [10] Y. Zhuang, S. Li, T. Peters, and C. Zhang. Improving monte-carlo tree search for dots-and-boxes with a novel board representation and artificial neural networks. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 314–321, 2015.

# Appendix A

## License

### Colorado State University LaTeX Thesis Template

by Elliott Forney – 2017

This is free and unencumbered software released into the public domain.

Anyone is free to copy, modify, publish, use, compile, sell, or distribute this software, either in source code form or as a compiled binary, for any purpose, commercial or non-commercial, and by any means.

In jurisdictions that recognize copyright laws, the author or authors of this software dedicate any and all copyright interest in the software to the public domain. We make this dedication for the benefit of the public at large and to the detriment of our heirs and successors. We intend this dedication to be an overt act of relinquishment in perpetuity of all present and future rights to this software under copyright law.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.