

MLND Capstone Project

A Deep Reinforcement Learning Approach to “Dots and Boxes”

Matthew Deakos
May 10th, 2017

1. Definition

1.1 Project Overview

In this project, a reinforcement learning approach was used to tackle the game “Dots and Boxes”. Reinforcement learning within the machine learning domain is a process by which an agent learns to act by interacting with its environment. It does this by experiencing “rewards” for taking given actions. A useful analogy, albeit a naive one, could be the process training a dog. If a dog does something correct, you reward that dog with a treat. If that dog does something incorrect, it may receive a scolding. To translate that metaphor to the problem of game playing; when the agent performs actions that result in a ‘win’, it receives a reward, and actions that result in a loss are punished with a negative reward. Ultimately, the program should seek to maximize its reward by picking the best actions in the current state of the environment.

Game Description

The game “Dots and Boxes” is played on a game board as shown in figure 1. The players take turns drawing lines between two adjacent dots. When a player completes a box by drawing the fourth wall, they get a point and get to draw again. The game continues until all lines are filled, and the winner is the player who has the most points at end-game.

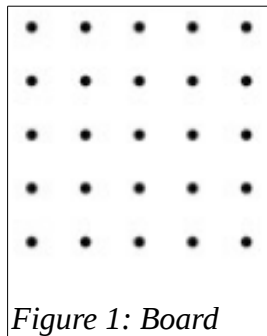


Figure 1: Board

In this paper, techniques used in attempting to accomplish this goal for the case of the two-player adversarial game “Dots and Boxes” are explored and analyzed.

1.2 Problem Statement

Unfortunately, the game “Dots and Boxes” is not a popular game, and so does not have any publicly available datasets. This means that the agent has to learn to play this game from scratch, unlike AlphaGo, the famous Go playing AI agent, which was able to use a comprehensive dataset of human moves to learn an initial policy. The agent is planned to play one million matches against previous versions of itself in order to improve its policy in the game environment.

The general learning technique proposed is using a Deep Q Network (DQN hereafter), though some important modifications are made. A DQN is in some regard an extension of Q-Learning, a very popular technique in the reinforcement learning domain. Q-Learning, in general, seeks to find evaluate state-action pairs in some environment. In this case, that means determining the value of “drawing a wall” given a certain board. A DQN uses this concept in tandem with deep neural networks, which serve as the value function for these state-action pairs. Further details of the algorithm are described in Section 3.

Ultimately, the goal of this project is to train a machine learning agent is to play ‘Dots and Boxes’ substantially better than random, and ideally at super-human levels.

1.3 Metrics

The following metrics will be used to evaluate the effectiveness of the project implementation.

Win Rate

This statistic will be the proportion of the number of games that the agent can win against a test agent. A high number of games will be played to ensure accuracy (>10 thousand).

Loss Rate

This statistic measure the proportion of games that the agent loses against a test agent. This is necessary, because there is a third outcome for this game, so losses should be explicitly measured.

Draw Rate

This statistic measures the proportion of games that the agent plays against a test agent that end in a draw.

These metrics have been selected because they are clear indicators of performance for a game environment. Typical metrics in supervised learning are not robust in this setting. Mean Squared Error, for example, is most useful when error is clearly defined, and some ground truth values are known. In this setting, however, we do not know whether a given move helps the agent win or lose a match, and so the *meaning* of error is not clear.

Conversely, an increase in the win-rate over time means we can be unambiguously sure that the agent is actually learning how to play the game. Therefor, it is clear that the metrics listed above represent the most logical choice of performance indicators .

2. Analysis

Note: Section on data exploration have been changed due to the nature of the problem. Data exploration has been replaced with a detailed explanation of the input and output space as well as an overview of the environment and agent API.

2.1 Input Space

The game of “Dots and Boxes” is played on the game board shown in figure 1. The size of the game board can be as large or small as desired, but for the purposes of this project a 5 x 5 board was used. A board this size has 40 potential actions, corresponding to 40 wall spaces. Since every wall is a class of the set {Exists, Not Exists}, this board has a state space of 2^{40} , or roughly 1.1 trillion possible states.

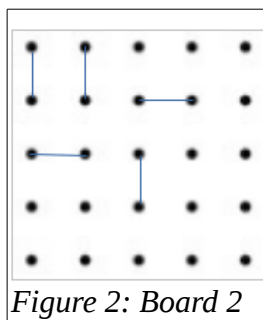


Figure 2: Board 2

The state of the game board is represented as a series of 4 x 4 cells, and each cell is represented by an array containing a one-hot encoding of wall positions corresponding to the pattern {N, S, E, W}. For example, if a cell has a 1 in the north position, it means that there is a wall on the north side of the cell. As an example, the top-left cell in figure 2 would be represented by the array {0, 0, 1, 1}. In some sense, this encoding mimics typical image input, which is an $n \times n \times 3$ matrix where the last dimension represents the colour channels. This encoding was selected because it retains local patterns among on the board. The hypothesis is that a convolutional neural net may be able to capitalize on these local patterns when evaluating the game board.

2.2 Output Space

In the 5 x 5 board, there are a total of 40 possible actions to take. These actions are encoded numerically, from 0 – 39 inclusive. Horizontal walls are represented by [0-19], where vertical walls are represented by [20-39]. A map of the action space is shown in figure 3.

2.3 Environment and Agent API

The interface for agents and the environment are planned as shown in figure 4.

●	0	●	1	●	2	●	3	●
20	21	22	23	24				
●	4	●	5	●	6	●	7	●
25	26	27	28	29				
●	8	●	9	●	10	●	11	●
30	31	32	33	34				
●	12	●	13	●	14	●	15	●
35	36	37	38	39				
●	16	●	17	●	18	●	19	●

Figure 3: Action Map

```

class Agent:
    """The basic agent class"""
    def act(self, state):
        """Act in the environment"""
    def observe(self, state, reward):
        """Observe the environment and collect a reward"""

class Environment:
    """Environment Class"""
    def step(self, action):
        """Take the action in the environment and allocate rewards"""
    def score_action(self, action):
        """Returns the score of an action (0, 1 or 2)"""
    def play(self):
        """Play an entire game and return results"""

```

Figure 4: API

2.4 Exploratory Visualization

Since there is no dataset to visualize, we will instead take a look at some potential game boards and hypothesize on what strategies the agent should learn to act on in those scenarios.

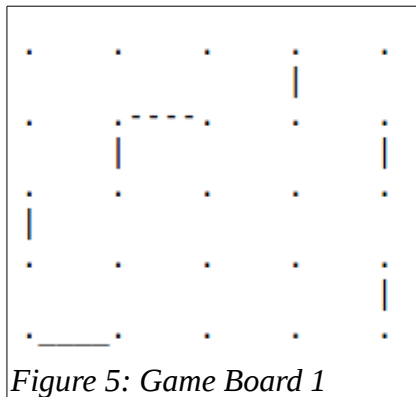


Figure 5: Game Board 1

The game board in figure 5 is an early game board. It is not clear as a human player what move is the correct choice. However, on the cell in the second row by second column, the north and west walls are present. An obvious strategy, then, is to *avoid* placing a wall in the southern or eastern walls, as that would allow the opponent to score on the next turn.

The game board in figure 6 represents a potential mid-game state.

The obvious choice from a human perspective is the to draw the northern wall on the cell in row 2, column 3. This move will allow the player to get a point. An agent in this environment should learn that completing these boxes is the correct choice.

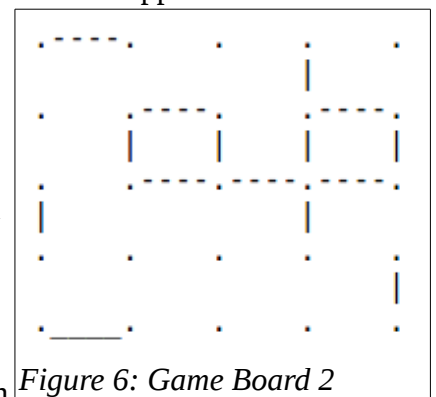


Figure 6: Game Board 2

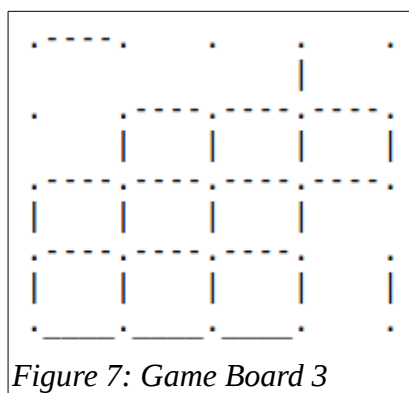


Figure 7: Game Board 3

Lastly, figure 7 represents an end-game board. In this setting, there are no clear 'right' actions. There are, however, actions that are intuitively worse than others. For example, drawing a wall along any space in the north-west end of the board will allow the opponent a long series of points. However, choosing to draw a wall in the top-right corner allows the opponent only one point. Ideally, assuming that my human intuition is correct, the learning agent will evaluate the first options as worse and the latter as better.

2.5 Background on Algorithms and Techniques

The general technique chosen for approaching this problem is a DQN. A DQN is, in part, a combination of other machine learning techniques.

2.5.1 Q-Learning

Q-Learning is a popular method of determining the value of state-action pairs in reinforcement learning. The Q-Value, represented by $Q(S, a)$ where S is the state and a is the action, is derived from the reward received for taking action a , plus the discounted sum of all future rewards. The generic formula for updating a Q-Value can be represented by the equation in figure 8.

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t}_{\text{learning rate}} \cdot \left(\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Figure 8: Q-Learning Algorithm

The ‘learned value’ in the formula above can be thought of as the target. It is comprised of the reward (the value of the state-action pair), the discount factor (discussed in 2.4.6), and the estimate of optimal future value. This means the value of choosing the right action, according to the current policy, at every state going forward. This ‘target’ is essentially what the Q-value should be as dictated by the reward from the environment and beliefs about the future. The learning rate (discussed in section 2.4.6) adjusts the current Q-Value so that it moves towards the target value by some amount. This parameter is between 0 and 1.

In this project, the reward will be received via winning (reward = 1) or losing (reward = -1) a game in the environment.

2.5.2 Convolutional Neural Network

Given that the number of states in this project is nearly 1.1 trillion, exploring every state-action pair is simply impossible to accomplish. A DQN uses Q-learning in conjunction with deep neural nets to approximate the Q-Value, so that it’s not necessary for every state to be visited, nor every action explored.

Neural nets are computational models that mimic the way brains work. They consist of a set of neurons that interact with each other to create complex functions. Mathematically, this is represented by a series of matrix multiplications. However, in order for a neural net to represent non-linear functions, ‘activation functions’ must be added to these neurons. An activation function transforms linear information into a non-linear representation, thereby allowing the development of non-linear predictive models. Typically, neural nets are trained using a dataset of labeled data, but it is possible to update a neural net according to Q-Learning by multiplying the loss $((\text{Reward} + \text{discount} * Q_{\text{next}}) - Q_{\text{current}})$ by the gradient of the loss with respect to the weights in the Q-Function.

In this project, this concept was extended to convolutional neural networks. Convolutional layers are neural networks that share parameters in *kernels*. This works by multiplying the channels (3rd dimension of the input space) by some shared weights to create different features with a different number of channels. Although the input space for this problem is much smaller than those typically used for convolutional neural nets (images), convolutional neural nets are useful for local pattern recognition, and it is intuitive that perhaps that recognizing local patterns may have some strategic

benefit in dots and boxes. It is in that spirit that a convolutional neural net was chosen as the Q-Function.

2.5.3 Experience Replay

An important aspect of DQNs is the concept of experience replay. In a DQN, the Q-Function is not updated online. Instead, records of gameplay are stored in a replay table. Typically, these records take the form of state transitions and rewards as such: (State, Action, Reward, Next State). This replay table is sampled randomly in mini-batches and these mini-batches are used to update the Q-Function. This helps reduce the correlation between actions in a game, so the true value of an action is learned more accurately. This technique is to be implemented in this project.

2.5.4 Exploration vs Exploitation

Lastly, there is an issue of exploration and exploitation. In order to determine which actions are optimal, an agent should be trying actions that it would not necessarily choose to try according to its policy. The question becomes: How much of the time should an agent explore new actions vs pursue known actions? A common way to manage this problem is with an epsilon-greedy exploration policy. This is a simple way of exploring the action space while the agent learns.

At some percent of the time, the agent will choose to take a random action rather than follow its current policy. Occasionally, the chosen action will be better than the action dictated by the policy, and the policy has the opportunity to change. Although there are some more advanced methods, for the purpose of this project this method is acceptable.

2.5.5 Acting on Policy

In order for the trained agent to act to its best ability, it simply needs to choose the action that has the maximum Q-Value, as decided by the Q-Function and the state.

2.5.6 Hyper-Parameters

Several important hyper-parameters need to be selected for these algorithms. Two very important selections are the discount rate and the learning rate. This discount rate controls how much an agent is going to care about future states when making a decision. This parameter ranged from 0-1, 0 being entirely in-the-moment, where 1 represents the notion that future states matter as much as the current state.

Learning rate is the degree to which new information is considered by the agent. A high learning rate means that new information will change the current 'belief system' of the agent very quickly, while a low learning rate means adjustments are slow. Although a high learning rate sounds better on the surface, lower learning rates tend to prevent divergence. Learning rates can be between 0 and 1.

2.5 Benchmark

Three benchmark models are designed to test the ability of the learning agent.

Random Agent

The first model is a random player. This is a player that acts entirely randomly in the game environment. It is the simplest possible model to overcome, and a failure to do so would signify that the model is not working. Due to the simplicity of this model, we set the threshold of success at a 95% win-rate.

Naive Agents

The second and third models are 'naive' agents, in that they will follow some hard-coded rules. The second model acts according to the following heuristic: If a scoring move is available, take it. Otherwise, act randomly. If the agent can reliably draw against this model, it means it has learnt one of the most obvious principles of 'Dots and Boxes': scoring. Thus, we set the threshold of success at a combined win rate and draw rate of 50%

The third model acts according to the following heuristic: If a scoring move is available, take it. If a move that completes the third line of a box is available, avoid it at all costs (because this would allow the opponent to score). Otherwise, choose randomly. This model represents a significant barrier in strategy, which is to consider the opponents next move. Since this model is a relatively difficult one to beat, we set the threshold for success at a combined win and draw rate of 25%.

Models 2 and 3 are a significant level of complexity above the first one, and on some level represent the intuitively expected behaviour of naive human players. As such, these are difficult hurdles to overcome, but will provide interesting benchmarks for performance.

3. Methodology

3.1. Preprocessing

Given the nature of the problem, very little preprocessing is required beyond setting up the environment. Since the state is already encoded as a 3-dimensional array, it can be passed directly into the Q-Function of an agent playing the game.

3.2 Implementation

3.2.1 Algorithms and Architecture

Convolutional Model Architecture

The architecture of the convolutional network is described as follows (all activations are elu (exponential linear unit) unless otherwise specified).

Input

-> Layer 1: 3x3 convolution and 1x1 convolution with 16 filters (concatenated)

-> Layer 2: 3x3 convolution and 1x1 convolution with 32 filters (concatenated)

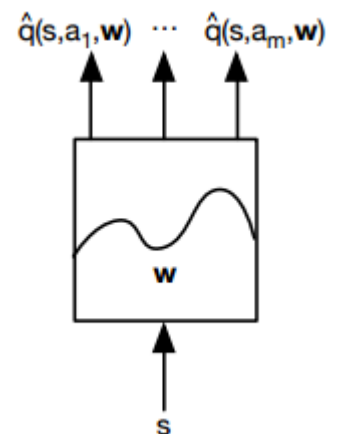


Figure 9: Q-Function Diagram

- > Layer 3: Fully Connected Layer with 256 Neurons
- > Layer 4: Fully Connected Layer with 256 Neurons
- > Output Layer: Fully Connected layer with 40 neurons and tanh activation

Rather than a Q-Function that takes a state-action pair and returns a value, this function takes a state and returns values for every state-action pair. This allows for far more efficiency, as this way only one forward pass is needed to find all Q-Values, rather than a pass for every possible action. (See figure 9).

The ELU activation was chosen to help avoid dead neurons, which frequently occurred while using RELU activations.

Following the Policy

When the agent was not exploring the environment, it made actions based on evaluating the Q-Values at the current state. However, in each state every action had a corresponding Q-Value, even invalid choices. To deal with this, the agent simply chose the action that corresponded to the highest Q-Value that was in the space of valid actions. Code is shown in figure 10.

```
# Follow Epsilon-Greedy
if random.random() < self.epsilon and self.learning:
    chosen_action = np.random.choice(self._environment.valid_actions)
else:
    q_values = self.DQN.predict(feature_vector)[0] # Get all Q-Values for the current state
    max_valid_q = q_values[self._environment.valid_actions].max() # Get the highest Q value that corresponds to a valid action
    best_actions = np.where(q_values == max_valid_q)[0] # Select all actions that have this Q-Value
    chosen_action = random.choice([action for action in best_actions if action in self._environment.valid_actions]) # Choose randomly among valid actions
return chosen_action
```

Figure 10: Action Choice Code

Q-Learning Algorithm and Replay Table

The original Q-Learning algorithm was modified to allow for faster convergence in an adversarial environment. In the initial update algorithm, shown in figure 11, the value of the current state is expressed as the reward at the current state plus the sum of the discounted expected rewards at future states. However, “Dots and Boxes” is adversarial. This means that if an action results in an excellent state for your opponent, that is a bad action to take (unless your expected reward for the action is still much greater). In Dots and Boxes, a scoring move allows the player to go again, and a non-scoring move means the opponent gets to move next. So, in addition to storing transitions and rewards in the replay table, another parameter was stored: an integer which represents whether or not the next state is ‘owned’ by the opponent or by the current player. During an update, this integer was converted into a -1 if the next state belongs to the opponent, or 1 if the next state belongs to the player (denoted in the pseudocode as phi). The Q update then followed this general algorithm:

$$Q(S, a) = Q(S, a) + \alpha(R + \gamma \cdot \phi \cdot \max_a Q(S', a))$$

The result is that the value of the current state now subtracts the value of the next state if it belongs to an opponent, and adds it if it belongs to the agent.

As shown, updates are not permitted unless the replay table is at least at half-capacity. This heuristic is mostly arbitrary, and serves only to make sure that updates are relatively uncorrelated.

The phi variable as described above is represented by the “next_turn_vector” variable in the python code.

As mentioned, the model architecture calculates all action Q-Values for the given state. The update, then, is only applied to actions that are being replayed in the replay table. The gradient for other matrix elements must be 0, so the target at those locations is set to be equal to the Q-Values predicted by the Q-Function.

```
def train(self):
    """
    Train the network based on replay table information
    """
    if self.transition_count >= self.replay_size/2:
        if self.transition_count == self.replay_size/2:
            print("Replay Table Ready")

        random_tbl = np.random.choice(self.replay_table[:min(self.transition_count, self.replay_size)],
                                      size=self.update_size)

        # Get the information from the replay table
        feature_vectors = np.vstack(random_tbl['state'])
        actions = random_tbl['action']
        next_feature_vectors = np.vstack(random_tbl['next_state'])
        rewards = random_tbl['reward']
        next_turn_vector = random_tbl['had_next_turn']

        # Get the indices of the non-terminal states
        non_terminal_ix = np.where([~np.any(np.isnan(next_feature_vectors), axis=(1, 2, 3))])[1]
        next_turn_vector[next_turn_vector == 0] = -1

        q_current = self.predict(feature_vectors)
        # Default q_next will be all zeros (this encompasses terminal states)
        q_next = np.zeros([self.update_size, self.n_outputs])
        q_next[non_terminal_ix] = self.predict(next_feature_vectors[non_terminal_ix])

        # The target should be equal to q_current in every place
        target = q_current.copy()

        # Apply hyperbolic tangent non-linearity to reward
        rewards = np.tanh(rewards)

        # Only actions that have been taken should be updated with the reward
        # This means that the target - q_current will be [0 0 0 0 0 x 0 0...]
        # so the gradient update will only be applied to the action taken
        # for a given feature vector.
        # The next turn vector controls for a conditional minimax. If the opponents turn is next,
        # The value of the next state is actually the negative maximum across all actions. If our turn is next,
        # The value is the maximum.
        target[np.arange(len(target)), actions] += (rewards + self.gamma * next_turn_vector * q_next.max(axis=1))

        #Update the model
        self.sess.run(self.update_model, feed_dict={self.input_matrix: feature_vectors, self.target_Q: target})
```

Figure 11

Hyperbolic Tangent Output Layer

During the initial training, there was a recurring problem with the learning algorithm. The Q-Values kept exploding – no matter the learning rate attempted, and eventually became too large for the computer to represent. This caused an error, and the agent could no longer learn. A number of methods were implemented in an attempt to solve this problem, including error clipping, gradient clipping, and hyper-parameter adjustment.

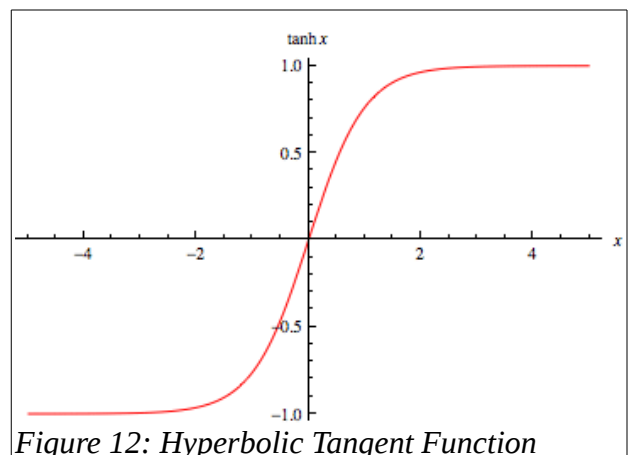


Figure 12: Hyperbolic Tangent Function

Finally, a hyperbolic tangent non-linearity was added to the output layer of the convolutional neural network (see figure 12). The impact of this feature was that the Q-Function output was explicitly capped at -1 and 1. However, this also meant that the Q-Function was no longer learning an actual *value* for the state-action pairs. The actions were still influenced by the values of the future states. Though it certainly seemed plausible that this might lead to ineffective learning, the speed of convergence to a policy was wildly more successful than without this implementation. It seems that this method still allows the agent to rank the value of actions in the current state against one another effectively, though a more rigorous mathematical investigation into this phenomenon is probably warranted. A simple comparison between learning progress of the DQN including the tanh activation and excluding the tanh activation is shown in section 3.3.

Hyper-Parameters

A number of hyper-parameters had to be selected for this project. However, due to the computationally intense nature of this problem, it was difficult to rigorously evaluate a wide selection of hyper-parameters without a great deal of time or computational power. The discount rate did not seem to have much of an effect on learning at the early stages of training, but any learning rate higher than $1e-5$ caused rapid divergence. The final model used a *discount rate* of 0.6 and a *learning rate* of $1e-6$.

3.2.2 Training Procedure

The training process for the learning agent was fairly straightforward. Two agents with a DQN ‘mind’ were initialized, which will be referred to as the ‘learning agent’ and the ‘target agent’. The target agent did not learn while it played the games. These agents played games against each other, and the learning agent would update their Q-Function as the games went on. At some interval, the learning agent would save its Q-Function and that function would be loaded into the target agent. This allows the ‘opponent’ to improve as the agent improves.

1 million games were played against the target agent from start to finish.

The interval at which the target agent was initialized at 1000 games, however this did not remain constant. This parameter is discussed more thoroughly in section 3.3

3.2.3 Testing Procedure

After 1000 of games of training against an opponent, the learning agent was then tested against the three benchmark agents outlined in section 2.5. The learning agent was set so that it wouldn’t update its Q-Function or record information for the replay table during these games. One thousand games were played against each agent, and the metrics outlined in section 1.3 were recorded in a log file. 500 of these games were played as the first player, and the other 500 as second player, to ensure that the tests were robust.

3.3 Refinement

Although the computational intensity of the problem made refinement a difficult process, several important modifications to the planned algorithms were made.

Modified Q-Learning Algorithm

The basic Q-Learning algorithm was modified to fit an adversarial game environment, as described in section 3.2.1. Before implementing this modification, the algorithm was very slow to converge. Results over the first 10 thousand training iterations are shown in figure 13.

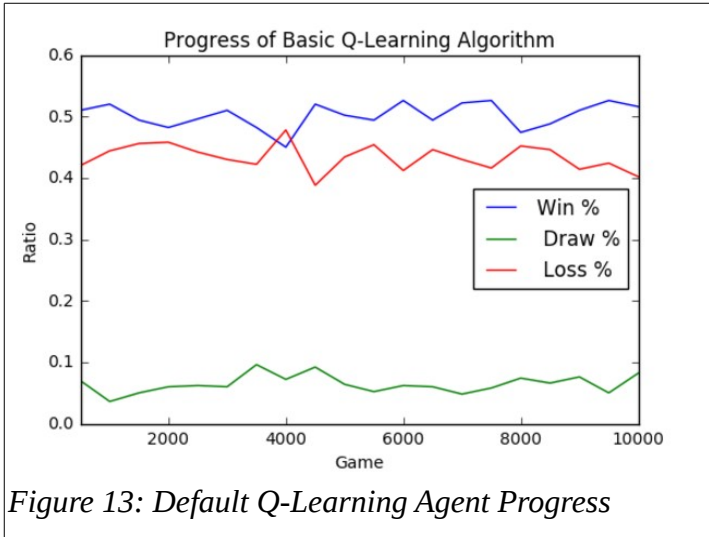


Figure 13: Default Q-Learning Agent Progress

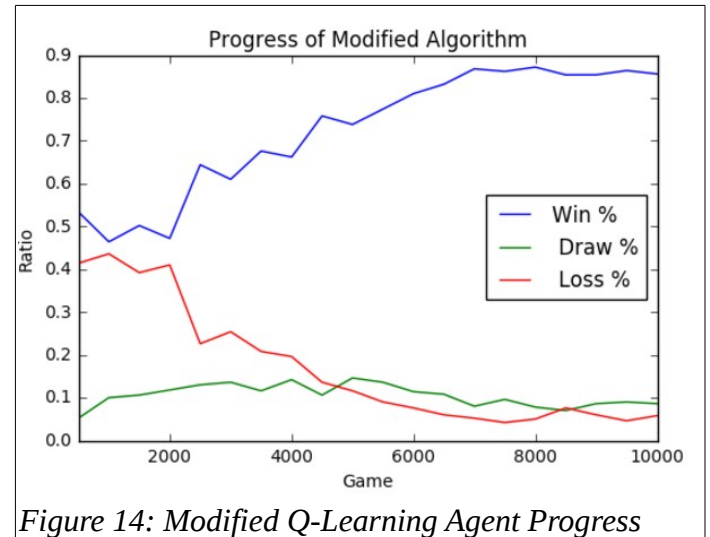


Figure 14: Modified Q-Learning Agent Progress

Although it does appear to be performing slightly better than random, it's clear that very little progress was made over the first ten thousand games. This is likely due to the problems posed in an adversarial setting, which were outlined in Section 3.1.2.

Conversely, the modified algorithm proved quite effective over a short period of time. Progress of this algorithm is shown in figure 14. By the ten thousandth game, this algorithm is already outperforming the random benchmark by a significant margin.

Analysis of tanh output layer

To support the implementation of a hyperbolic tangent non-linearity on the output layer of the Q-Function, some performance metrics have been analyzed over the first 10 thousand games with an agent using a hyperbolic tangent layer and another using a more traditional linear output layer. Results are summarized in figures 15 and 16.

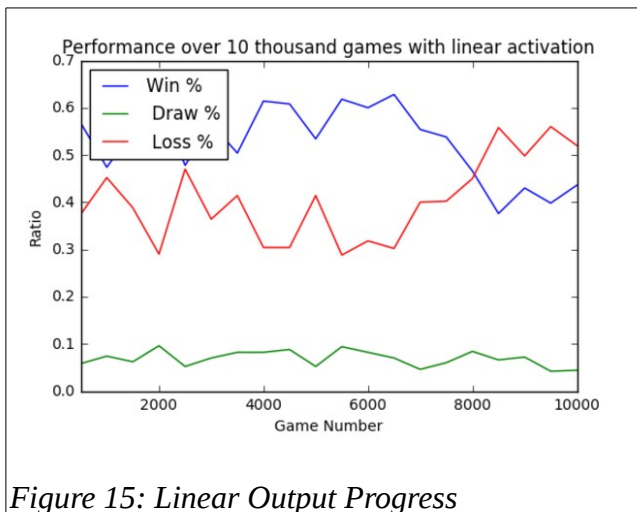


Figure 15: Linear Output Progress

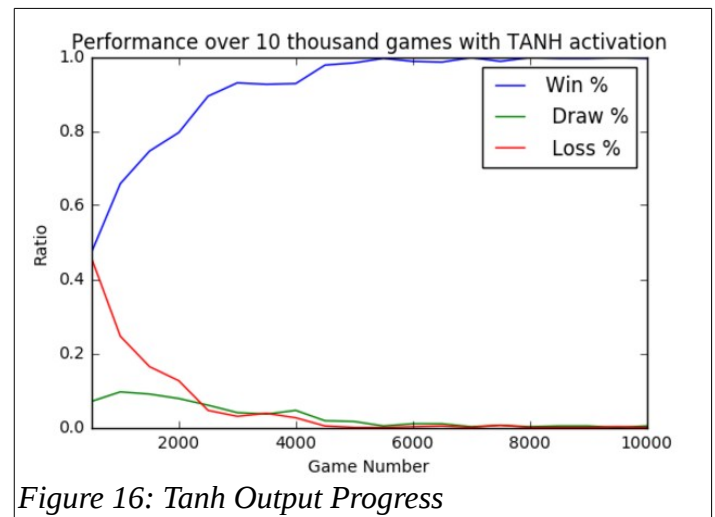


Figure 16: Tanh Output Progress

Clearly, the agent using this non-linearity has a much more rapid performance increase than the agent using a typical linear output, despite the potential pitfalls outlined in Section 3.2.1. In fact, no discernable performance increase appears in the first ten thousand games without the tanh activation. Other activations should be explored in the future to view the impact on performance in a DQN system.

4. Results

4.1 Model Evaluation and Validation

4.1.1 Test Results

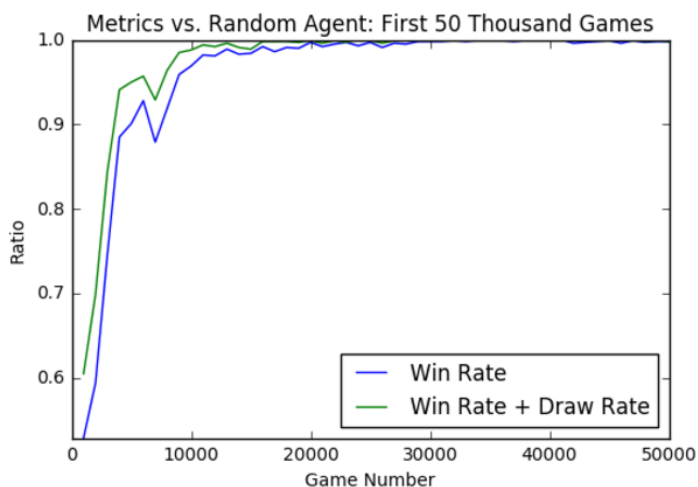


Figure 17: Agents Vs. Random

Benchmark: Random Agent

As shown in figure 17, which visualizes the win rates and the win-draw rates over the first 50 thousand games, the learning agent outperforms the random benchmark extremely quickly. By game ten thousand, the learning agent was beating the random agent over 90% of the time consistently. This signifies that implementation was certainly effective to some degree. Loss rate was not included as it can be inferred as $1 - (\text{win rate} + \text{draw rate})$.

Clearly, the threshold of success set for the first benchmark (win-rate greater than 95%) has been met.

Benchmark: Moderate Agent

The second benchmark, here referred too as the ‘moderate agent’, used the naive heuristic of ‘score when possible, act randomly otherwise’. Truncated and non-truncated summaries of the test logs are shown in figures 18 and 19.

Overall, the agent was performing worse than the moderate agent by the end of the millionth game, winning approximately 24% of games on average. Still, the fact that this benchmark was reliably losing a significant portion of games to the learning model shows that progress clearly occurred. It also appears that performance was still improving by the end of the training iterations. The combined win and draw rate reached around 40% by the millionth iteration, which did not pass the 50% threshold.

Clearly, success rates against the moderate agent did not begin to improve noticeably until near game 400 000. One explanation for this rapid improvement in performance could

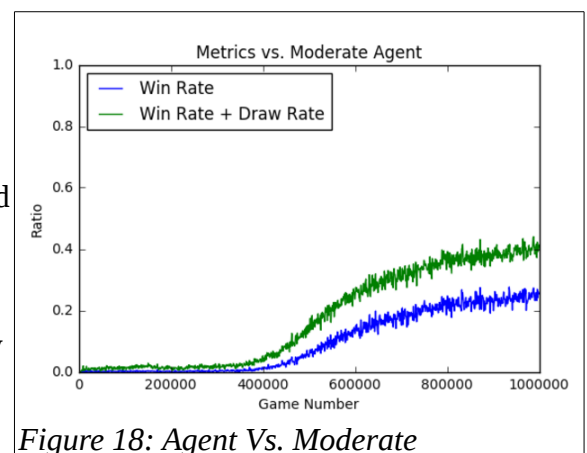


Figure 18: Agent Vs. Moderate

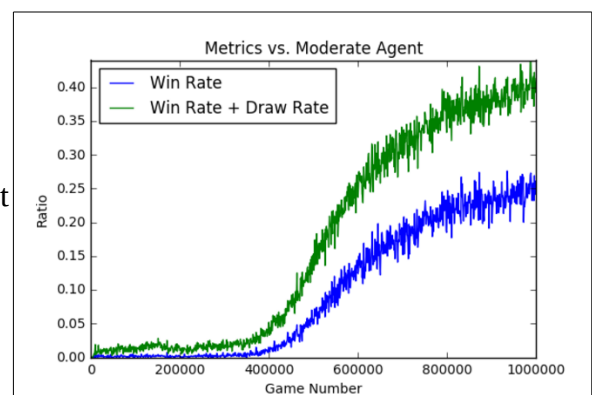


Figure 19: Agent Vs. Moderate (Truncated)

be that winning and drawing against this benchmark requires entire strategies to be learned. For example, it's possible that the agent learned how to score end-game, but was giving up points early-on. If this is indeed the case, using a higher discount rate may improve early performance by ensuring that the early-game is effectively influenced by the late game. This, and other potential improvements are analyzed in more depth in section 5.3.

Another possible explanation is an update step parameter. At game 400 thousand, the number of training iterations required before the opponent model updated was increased from 5 000 to 10 000. This switch may have helped the learning agent to converge on a strategy against the opponent with more success. However, it appears that the rise in performance actually began slightly before the 400 thousand game mark, so it's certainly possible that the performance gain was coincidental.

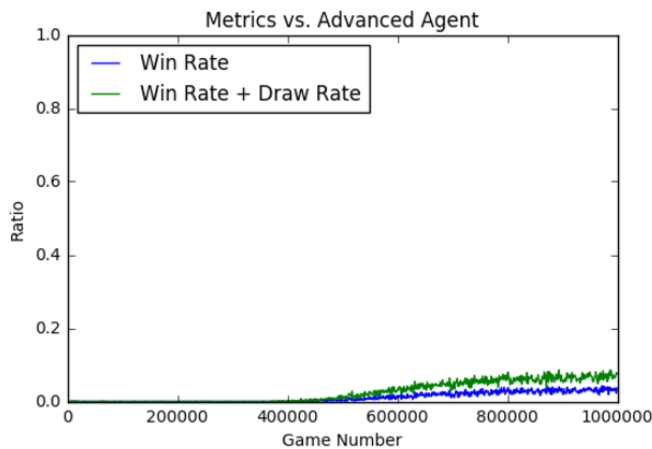


Figure 20: Agent Vs. Advanced

Benchmark: Advanced Agent

The final benchmark, here referred too as the 'advanced agent', used a similar strategy to that of the previous benchmark. Not only would this agent score when it had the opportunity, but it would avoid moves that provided the opponent an opportunity to immediately score. As shown in figure 13, the success rate against this agent was quite low. However, some progress was certainly apparent. Given the rapid increase in the performance against the 'moderate' benchmark, it seems possible that given more training time, the agent could have made a similar improvement here. The combined win and draw rate reached 9.7% at its highest. As such, the agent did not

overcome the threshold for success for the third benchmark.

Overall

Figure 14 shows consolidated win-rate progress across benchmark models. Figure 15 displays a moving average of performance at $n = 20$ thousand games.

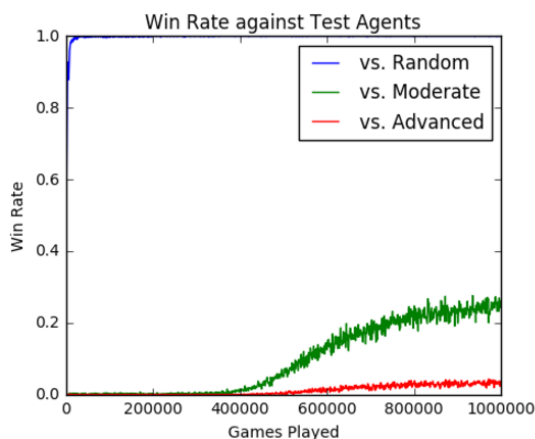


Figure 21: Consolidated Win Rate

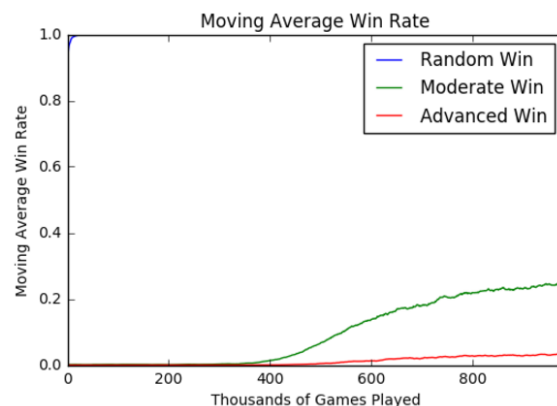


Figure 22: Smoothed Win Rate

Overall, the moving average win-rates ($n = 20\,000$) at the end of the training period were as follows:

Benchmark 1 (Random Agent) = 99.995%

Benchmark 2 (Moderate Agent) = 25.175%

Benchmark 3 (Advanced Agent) = 3.240%

4.2 Justification

Given the performance gains outlined in section 4.1, it seems clear that the method described in this paper is at least a viable option for an agent learning Dots and Boxes. The model was a decisive winner against the first benchmark and a contender in the second, despite not overcoming the threshold that was set for it. Overall performance was low for the last benchmark, but that heuristic was a fairly robust strategy to overcome.

The trained agent overcame the threshold that was set for the first benchmark, but not for the other benchmarks. There was a far higher disparity between the first and second benchmark than was predicted at the outset of the project; it took the agent only 10 thousand games to measure as successful against the random player, but 400 thousand just to *win* against the simple heuristic model.

However, there also seems to be room for growth given more training games. As mentioned, it seems that performance was still on the rise by the millionth iteration, and it seems likely that the agent would have been a competitive player against the second benchmark given time. Unfortunately that was beyond my capacity at this time. It certainly seems worth pursuing in the future, though, so that it can be seen how far this model can be pushed.

5. Conclusion

5.1 Additional Analysis

By taking a look at the ‘mind’ of the agent during certain board positions, we can try to gain insight into how the model is working. In fact, we will evaluate the board positions that we explored hypothetically in section 2.4. We can have the network evaluate individual states and analyze the Q-value output. For easy reference, I’ve copied the output map (figure 3) below:

●	0	●	1	●	2	●	3	●
20		21		22		23		24
●	4	●	5	●	6	●	7	●
25		26		27		28		29
●	8	●	9	●	10	●	11	●
30		31		32		33		34
●	12	●	13	●	14	●	15	●
35		36		37		38		39
●	16	●	17	●	18	●	19	●

Figure 3: Action Map

First let's take a look at an early game board.

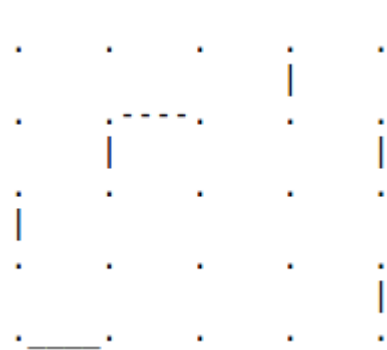


Figure 23: Game Board 1

taking either action would allow the opponent to score a point on the next round. This might simply indicate that more training is required, although may be a weakness in using a convolutional neural net. Intuitively, it might be presumed that a CNN would value 'box-like' positions as higher than non-box-like positions in general.

A simple representation of the dots and boxes game board is shown in figure 20, and the corresponding Q-Values in figure 21. It's immediately clear that the Q-Value of all possible outputs have converged to either 1 or -1, representing a 'good action' or a 'bad action'. Since only valid actions are reinforced in this implementation, (as discussed in Section X) we don't care about moves that are impossible (I. E there is already a wall in that position). The only *valid* moves that the agent considers 'good' on this board are actions 9 and 27. We can see by referencing figure 4 that these actions represent the east and south wall the second cell of the second row. It is surprising, at first glance, that both of these actions are considered 'good', as

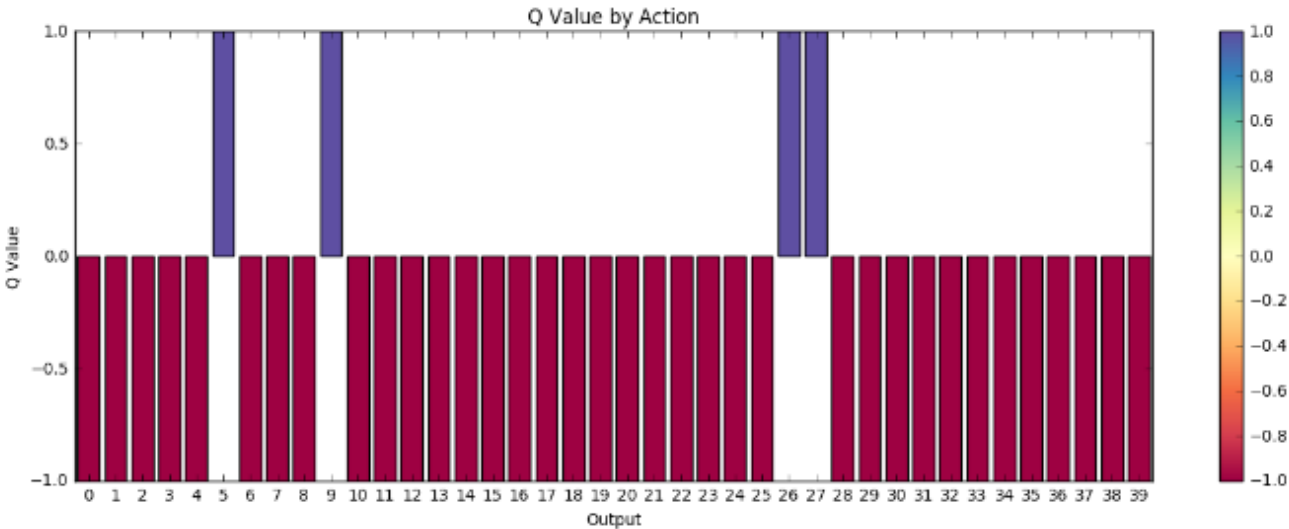


Figure 24

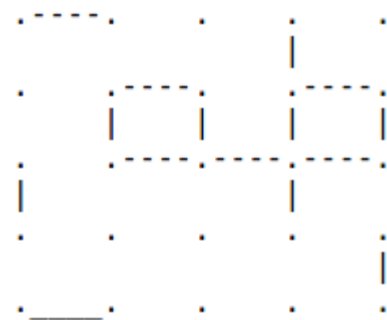


Figure 25: Game Board 2

Figure 25 shows a board that is further into the mid-game. In this case, the only valid action that the agent considers 'good' is action 6 (see figure 26), which is the action that completes the square on the second row. It appears that in this scenario, the Q-Function did not mistake moves that would complete only a third wall as beneficial, which might indicate that the previous result simply indicated that more training time may have been necessary.

Interestingly, the invalid moves that are ranked as 'good' are geographically close to the valid move. This might represent a strategic bias towards acting in the 'busy' part of the board.

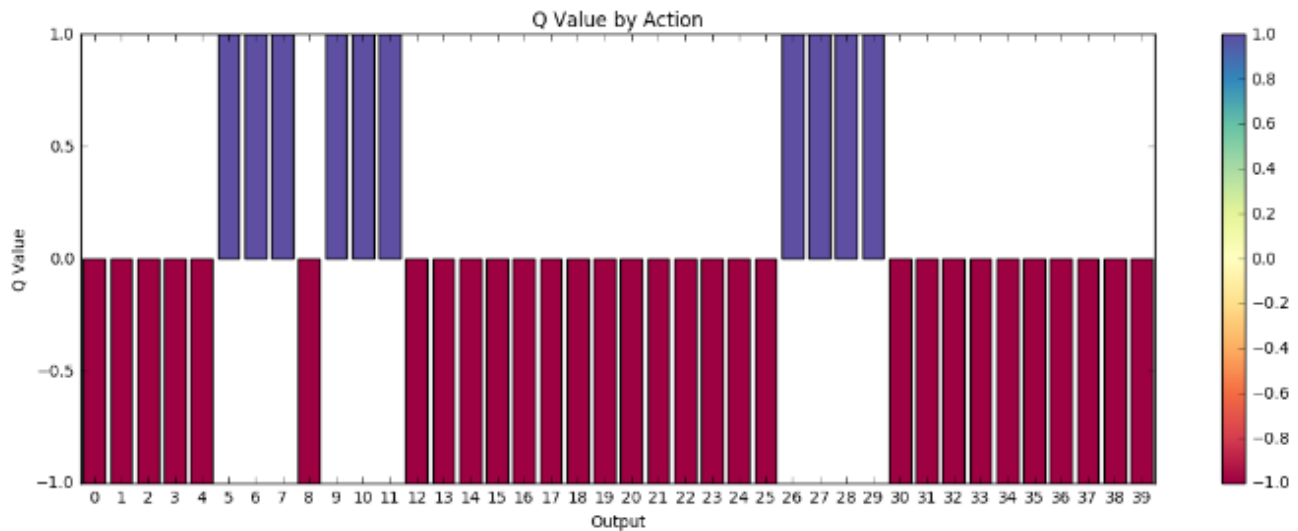


Figure 26: Q-Values 2

Lastly, we take a look at a near end-game board, where the game state and the Q-values are represented by figures 27 and 28.

In this situation, every single legal move is considered ‘bad’. It is in this situation that a negative aspect of the non-linear layer becomes apparent. Although it could be argued that there are no obvious ‘good’ choices, some choices (like position 1) seem far less bad than others (like position 19). It’s still possible that the agent could eventually learn a policy that represents that sort of logic, but it seems that this is exactly the sort of scenario that a linear model should perform better in. As a result of every legal action being ‘bad’, the agent will randomly take an action, which might have negative consequences on the game.

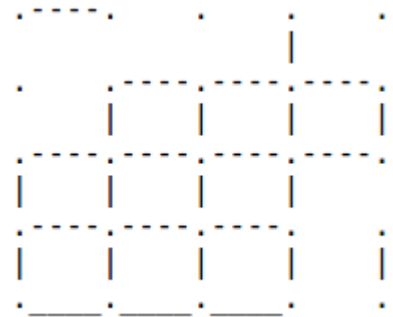


Figure 27: Game Board 3

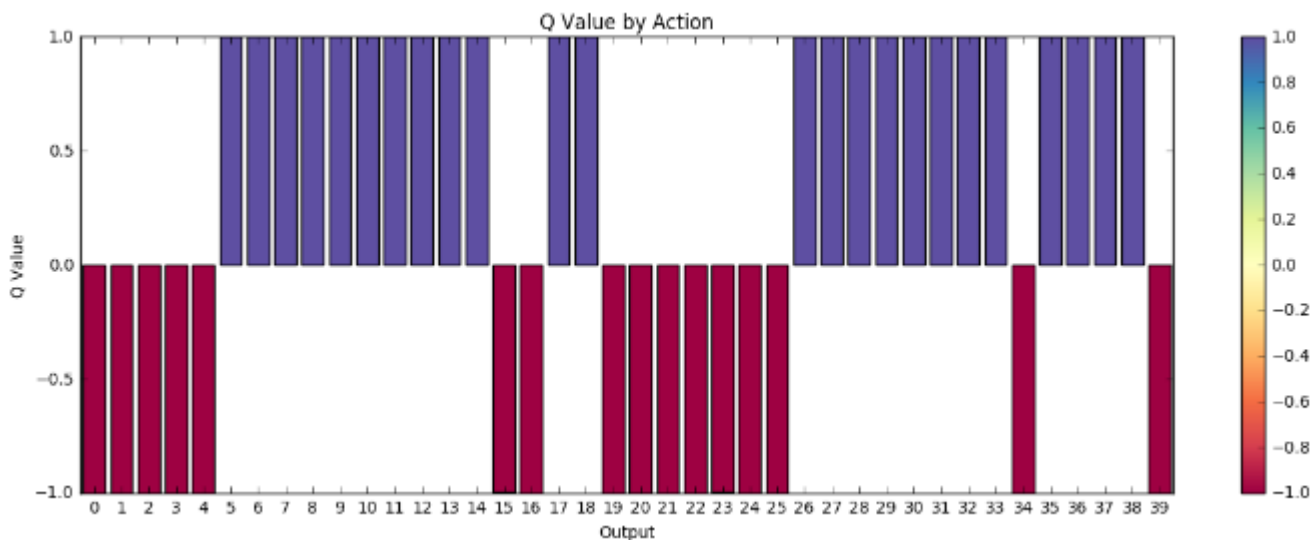


Figure 28: Q-Values 3

Implications

By examining the Q-Values at individual states, we have gleaned some insight into how the agent is operating. Upon inspection, it is unsurprising that actions have been reduced to a binary ‘good, bad’ option given our implementation; since the Q-Function output is a bounded-non-linearity, good actions will have a value that increase towards 1 until they can no longer be represented by their decimal places. Similar behaviour would occur in ‘bad’ actions.

It’s possible that some algorithmic changes could still mitigate this. If the reward, for instance, could be modified depending on the current value of the state, we may be able to drastically slow this convergence. Other non-linear activations might behave in a more nuance-friendly manner, which could be explored.

However, it may not even be a problem that the Q-value choice becomes this simple, and given the time perhaps the agents would still behave optimally. Simply putting an agent through more training sequences may provide an answer.

5.2 Reflection

Several decisions had to be made while designing this end-to-end project. A state space encoding was selected to be a 3-D array, the purpose of which was to be fed into a CNN. The output space was a simple integer choice. A DQN was implemented with several modifications, including a modified Q-Learning algorithm and a hyperbolic tangent on the output layer of the Q-Function. Designing these modifications required significant investigation and understanding, and were incredibly challenging and interesting aspects of this project. After the learning model was complete, the agent ran through one million games and actually managed to start performing significantly better than a random agent, and was at least a contender against the second benchmark.

In terms of difficulties, the first and foremost was that this project ended up being far more taxing on my resources (in both time and compute power) than I had assumed it would be, so it made evaluating different choices very difficult. For example, the agent didn’t even start to beat the second benchmark until around 400 thousand games in, which translates to roughly three days of continuously simulating games on my NVIDIA Geforce 940M graphics card. This large stretch of time meant that choices in hyperparameters, the model architecture, as well as other variables were very difficult to evaluate, so the final model is almost certainly sub-optimal in at least some of these areas. In order to optimize many of these choices, a significant investigation into alternative evaluation methods will be necessary.

5.3 Improvement

There are plenty of other techniques and extensions that could be applied to this project in the future, but were not implemented for time or resource purposes. Double DQN’s for example, use two different Q-Functions when training and selecting actions. Asynchronous methods exist to parallelize the training sequence, which can improve speed and performance. Policy Gradient is a different technique which tries to optimize for the policy (which action to choose) rather than learning a value function. Actor-critic networks combine Q-Learning and Policy Gradient. Many more choices exist for a reinforcement learning problem, and all-in-all the implemented model is relatively simplistic.

In addition to all of these techniques, and more, there are many changes that could be made to the current model that might improve performance. The CNN architecture, for example, was not changed largely from the first implementation due to the time it takes to adequately test it. Perhaps a different net would allow for better performance. Hyper-parameters, as well, were largely unchanged from the first implementation for a similar reason. It seems very unlikely that these choices were optimal for the problem, and given time and effort a better solution might be found.

References

- <https://en.wikipedia.org/wiki/Q-learning>
- <https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>
- <http://cs231n.github.io/convolutional-networks/>
- Udacity Forum and Lectures