Gunjan K Khut
UID–115813083

Homework 4

Introduction:

This project is about making use of a Q-learning reinforcement algorithm to implement a game-playing agent which learns how to play a game of 3x3 Dots and Boxes optimally for both players.

3x3 Dots & Boxes:

Dots & boxes is a 2-player game.

The starting state is an empty grid of dots (16 dots in case of a 3x3 size board). Both players take turns making a move; a move consists of adding either a horizontal or vertical line between two unjoined 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 greatest points number of points is the winner of the game.

Representing the game:

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 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 one denoting otherwise.

The edge ordering being considered is:

```
·  0  ·  1  ·  2  ·
12    13    14    15
·  3  ·  4  ·  5  ·
16    17    18    19
·  6  ·  7  ·  8  ·
20    21    22    23
·  9  · 10  · 11  ·
```

The index of a value in this list represents the corresponding edge, e.g. if edge 0 exists in a state, then index 0 in the list has value 1, and vice versa.

Problem Statement:

1. Program a game of Dots and Boxes:

- Players take turns drawing lines to complete boxes. A player is allowed to take another turn after completing a box. The game concludes when no more lines can be drawn and all boxes are claimed.

- Begin with a 2x2 grid. Assign a reward of +1 for completion of a box, +5 for a win, and 0 otherwise. Have it train itself to play through self-play and Q-learning. Use a table to store the Q-values. Compare performance against an automated player making random (legal) moves. Train for 100, 1000, and 10,000 games. Document improvement in performance against the random player as a function of the number of games of self-play.

2. Change the grid size to 3x3 and repeat training. Evaluate the potential of using the Q-value table from (1) to seed learning (to initialize the Q-table).

3. Change Replace the Q-value table from (1) with a functional approximation, such as a neural network. Use Q-learning to self-train (refine the Q-value functional mapping) and document the improvement in performance (against the random player) as a function of the number of games of self-play. Compare results obtained with performance from (1).

4. Using a 3x3 grid and a functional approximation to represent the Q-values, train the system for 100, 1000, and 10,000 games. Document performance against a random player as a function of the number of games played. Compare with results from (2) and determine which Q-value representation is more effective: table or functional approximation.

Metrics:

The following metrics are used to evaluate the efficiency 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.
Loss Rate
This statistic measures 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.

Conversely, a rise in the win-rate over time means we can be explicitly sure that the agent is learning how to play the game. Therefore, the metrics listed above represent the most logical choice of performance indicators.

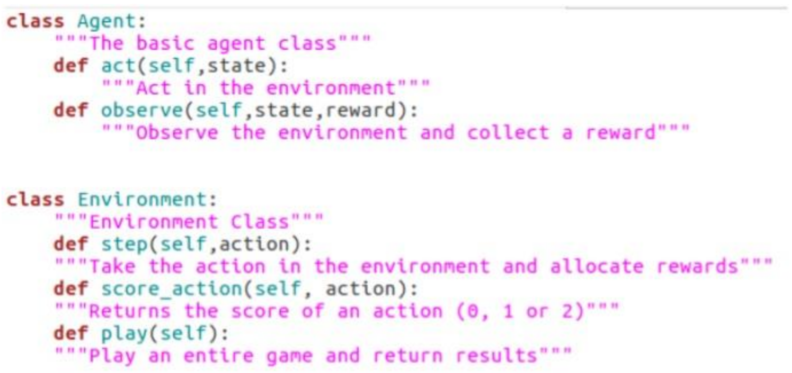
Analysis

Input Space

The size of the game board can be as large or small as desired, but here the game of "Dots and Boxes" is played on the game board of size 3 X 3 cells. A board this size has 24 potential actions, corresponding to 24 wall spaces. Since every wall is a class of the set {Exists, Not Exists}, this board has a state space of 224 possible states.
The state of the game board is represented as a series of 3 x 3 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. In some sense, this encoding mimic typical image input, which is an n x n x 3 matrix where the last dimension represents the color 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.

Environment and Agent API

The interface for agents and the environment is planned as shown in figure below



```
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"""
```

Description of Algorithm and Methods:

The method chosen to tackle this problem is a DQN. A DQN is, in part, a combination of other machine learning techniques.

Q - Learning

Q-Learning is a prevalent 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 below.

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

The 'learned value' in the formula above can be understood of as the target. It is comprised of the reward (the value of the state-action pair), the discount factor, and the estimate of optimal future value. This implies 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 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 given to the agent for completing the box is +1 via winning (reward = +5) or losing (reward = -1) or draw (reward = 0) a game in the environment.

Convolutional Neural Network

Given the number of states in this Homework is 224, exploring every state-action pair is 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, 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 ((Reward + discount * Qnext) – Qcurrent) by the gradient of the loss with respect to the weights in the Q-Function.

In this homework, this concept was adapted 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.

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 choose 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 decrease the correlation between actions in a game, so the true value of an action is learned more accurately. This technique is implemented in this homework.

Exploration Vs Exploitation

In order to determine which actions are best, 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 can change. Even though there are some more advanced methods, for the purpose of this homework this method is acceptable.

Acting on Policy

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.

Hyper – Parameters

The discount rate controls how much an agent is going to care about future states when deciding. 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.

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 a simple 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 giving to the following heuristic: If a scoring move is available, take it. Otherwise, act randomly. If the agent can consistently 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 important block in strategy, which is to consider the opponents next move. Since this model is a relatively tough 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 difficulty above the first one, and on some level signify the instinctively predictable behavior of naive human players. As such, these are difficult hurdles to overcome, but will provide interesting benchmarks for performance.

Methodology

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.

Implementation of Algorithms and Architectures

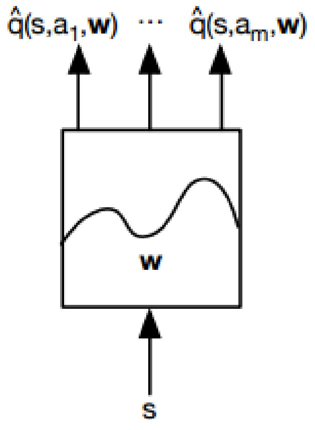
Convolutional Model Architecture

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)
- 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



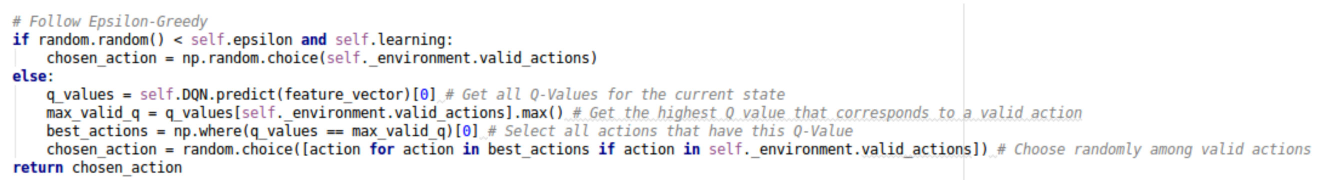
Q-Function Diagram

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.

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 below.



```
# 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
```

Action Choice Code

Q-Learning Algorithm and Replay Table

The original Q-Learning algorithm was altered to allow for faster convergence in an adversarial environment. In the initial update algorithm, shown in figure below, 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. In Dots and Boxes, a scoring moves 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 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:

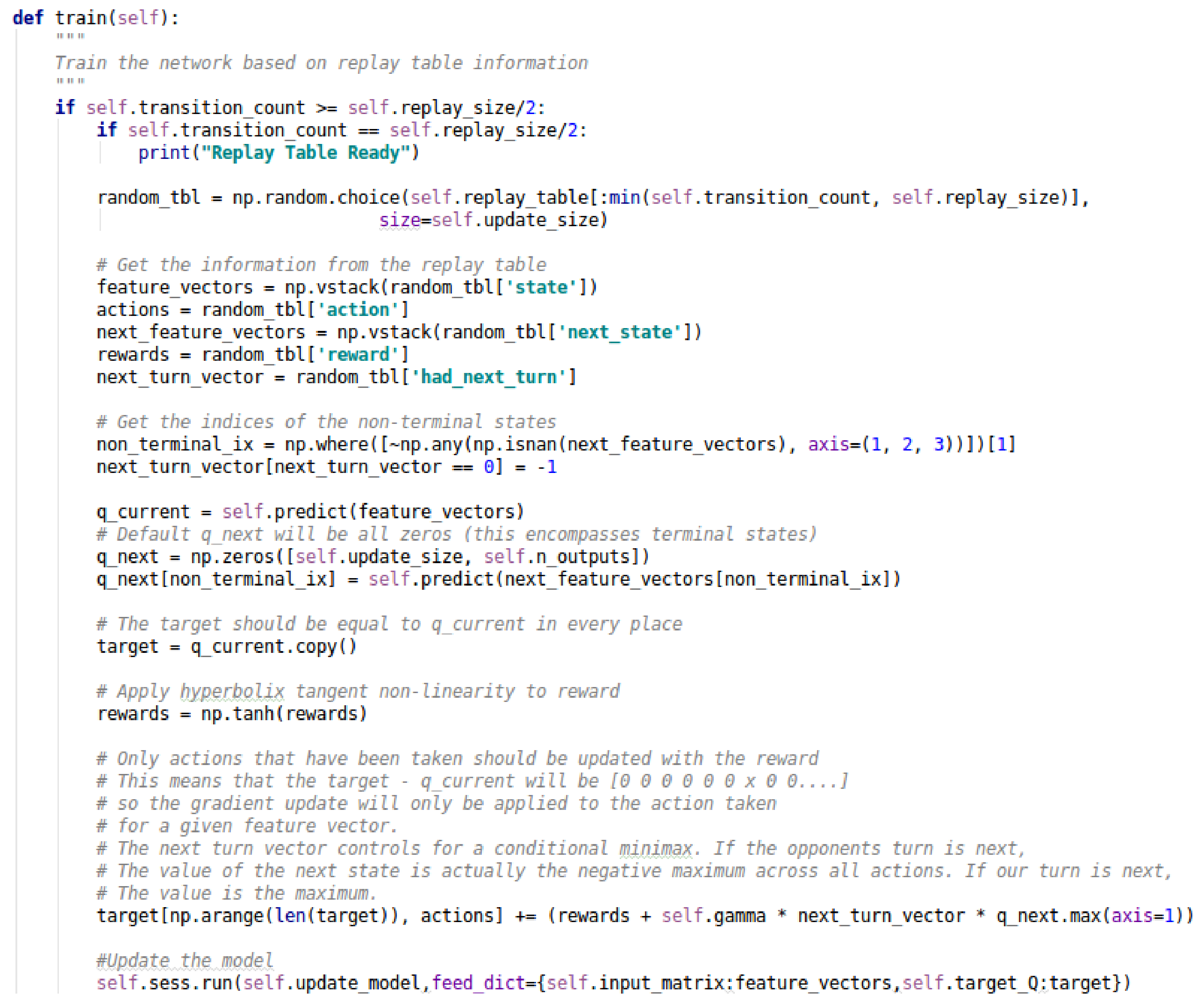
$$Q(S,a) = Q(S,a) + \alpha(R + \gamma * \varphi * Max_a\, Q(S',a)\,)$$

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 a half-capacity. This heuristic is mostly arbitrary and serves only to make sure that updates are relatively uncorrelated.

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

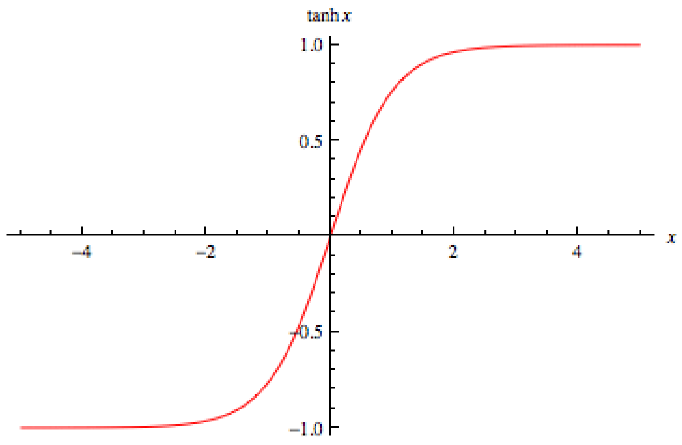
As mentioned, the model architecture calculates all action Q-Values for the given state. The update, then, is only applied to actions that are being replayed in the replay table. The gradient for other matrix elements must be 0, so the target at those locations is set to be equal to the Q-Values predicted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

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. Several methods were implemented to solve this problem, including error clipping, gradient clipping, and hyper-parameter adjustment.



Finally, a hyperbolic tangent non-linearity was added to the output layer of the convolutional neural network. The impact of this feature was that the Q-Function output was explicitly capped at -1 and 1. Yet, 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.

Hyper-Parameters

Several hyper-parameters had to be selected for this Homework. Still, 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 $1e^{-6}$.

Training Procedure

The training process for the learning agent was 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.

Conclusion:

2 X 2 Cells at 100, 1000 and 10000 games

It can be observed as the 2 X 2 cells game is small and has 212 possible states which are covered in the number of games used in Training. The learning agent quickly learned the game and can beat the Random, Moderate and Advanced Test Players. The learning agent high winning rate against Test players.

Learning Agent Against Random Agent = 99.99%

Learning Agent against Moderate Agent = 80.23%

Learning Agent against Advanced Agent = 78.54%

3 X 3 Cells at 100, 1000 and 10000 games

Given the performance gains in earlier sections, it is visibly clear that the method described here is at least a viable option for an agent learning Dots and Boxes. The model was a decisive winner against the Random Agent and a contender against Moderate despite not overcoming the threshold that was set for it. Overall performance was low for the last benchmark, but that heuristic was a robust strategy to overcome.

The trained agent overcame the threshold that was set for the Random agent, but not for the other two agents. There was a far higher discrepancy between the first and second agents than was predicted at the outset of the homework; it took the agent only 100 games to measure as successful against the random player but requires more than 10 thousand just to win against the simple heuristic model.

Nevertheless, there is room for growth given more training games. As mentioned, it seems that performance was still on the rise by the 224 iteration, and it's seems likely that the agent would have been a competitive player against the Moderate and Advanced agents given time.

Learning Agent Against Random Agent = 98.45%

Learning Agent against Moderate Agent = 20.24%

Learning Agent against Advanced Agent = 2.35%

References:

[1] https://en.wikipedia.org/wiki/Q-learning
[2] https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
[3] https://github.com/mattdeak/dots-boxes-RL
[4] Udacity Forum and Lectures
[5] http://cs231n.github.io/convolutional-networks/