CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

MACHINE LEARNING WITH FOX AND HOUNDS

A graduate project submitted in partial fulfillment of the requirements

For the degree of Master of Science in Computer Science

By

Dino Biel

December 2018

The graduate project of Dino Biel is approved:

_____	_____
John Noga, Ph.D.	DATE
_____	_____
Robert D. McIlhenny, Ph.D.	DATE
_____	_____
Richard Lorentz, Ph.D., Chair	DATE

California State University, Northridge

DEDICATION

I dedicate my thesis to my mother. I would never have made it this far without all your sacrifices. Your love and support through the years has always been instrumental in my journey of knowledge, and I owe you the world. I would also like to dedicate my thesis to you, the reader! Thank you for taking the time to read my work and I am truly humbled if I was helpful to you in any way.

Table of Contents

List of Figures

ABSTRACT

MACHINE LEARNING WITH FOX AND HOUNDS

By

Dino Biel

Master of Science in Computer Science

Deep Q Learning is a topic that has become very popular of late in the field of Machine Learning (ML), and especially when it comes to games. AlphaGo Zero and AlphaZero are two ML agents that have been created by Google utilizing Deep Q Networks (DQN) that have had massive success. AlphaGo Zero became a master of the game go, while AlphaZero was able to master the games of: go, chess, and shogi. The games of go and chess are played between two players, each player has the same number of pieces to move, and the move sets of each player are equivalent.

Instead of working with a game where both sides have equal pieces and move sets, I decided to work with a game that does not share those qualities. The game of fox and hounds is a simple game to both understand and play, but it deviates from chess and go by giving each player different pieces, move sets, and goals to accomplish. This makes the problem of creating ML agents more complex, as you now must create two different ML agents to play the same game instead of just one. I created a DQN agent for both the fox and hounds' players to play the game. The results taught me how impactful different inputs, objectives, and outputs could be on the agents involved.

Chapter 1: Introduction

1.1 Motivation

I have always been interested in learning new things throughout my scholastic career. Pushing myself to explore new topics outside of my comfort zone is a hobby of mine. With these things in mind, I chose my thesis topic to be my ultimate test. I wanted to throw myself into all sorts of discomfort for this final act of my master's degree.

I have never explored the topic of ML prior to my thesis, nor have I used the language of Python extensively. Hearing about AlphaGo Zero and AlphaZero quickly drew my interest into the field of ML and naturally that is how my thesis topic was born. The game of fox and hounds was an intriguing game to dive into because despite its simplicity, the game itself does have a unique quality with respect to ML. That quality is both players require separate model types since they do not share the same pieces and move sets. I became curious to learn how this quality would impact both agents from a learning aspect with everything else being equal.

1.2 The Game of Fox and Hounds

Fox and hounds is a board game that is played between 2 players on an 8x8 chessboard. The player who is controlling the fox only has one piece to move, and the player controlling the hounds has 4 pieces to move. The goal for the fox player is to successfully get to the other side of the board where the hounds start, and the goal for the hound's player is to trap the fox in such a way that the fox has no valid moves.

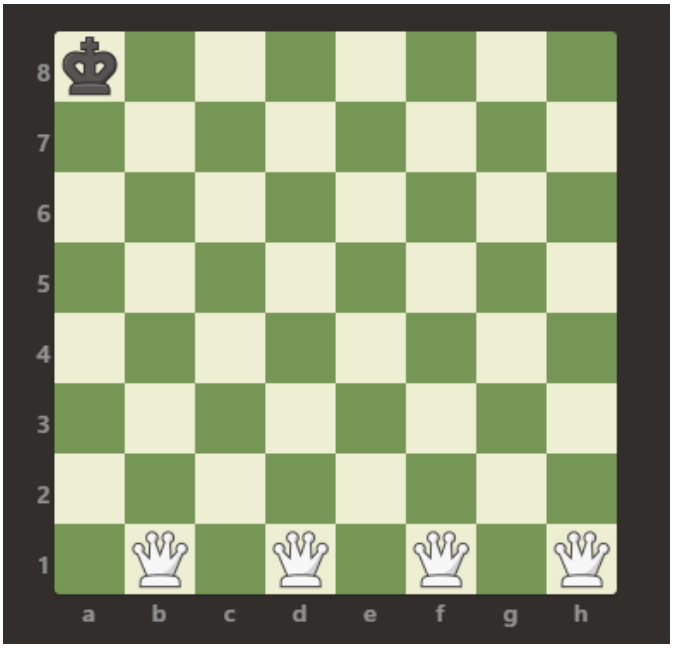

Figure 1.1: Initial game state with 1 fox (black) and 4 hounds (white)

The initial setup of the board is shown above in figure 1.1. The board is structured using standard chess notation. The black piece for the fox above would be in location “a8” to denote column “a” row “8”. If the fox can reach: b1, d1, f1, or h1 the fox will have won the game. As stated previously, the hound’s goal is a little less straight-forward. The goal for the hounds is to trap the fox on any square where the fox cannot make a valid move.

This end game scenario can take place on any valid square except for the winning squares for the fox: b1, d1, f1, and h1. The winning conditions are not the only thing that the fox and hounds differ on. The movement for the fox piece also differs from the movement of the hounds’ pieces.

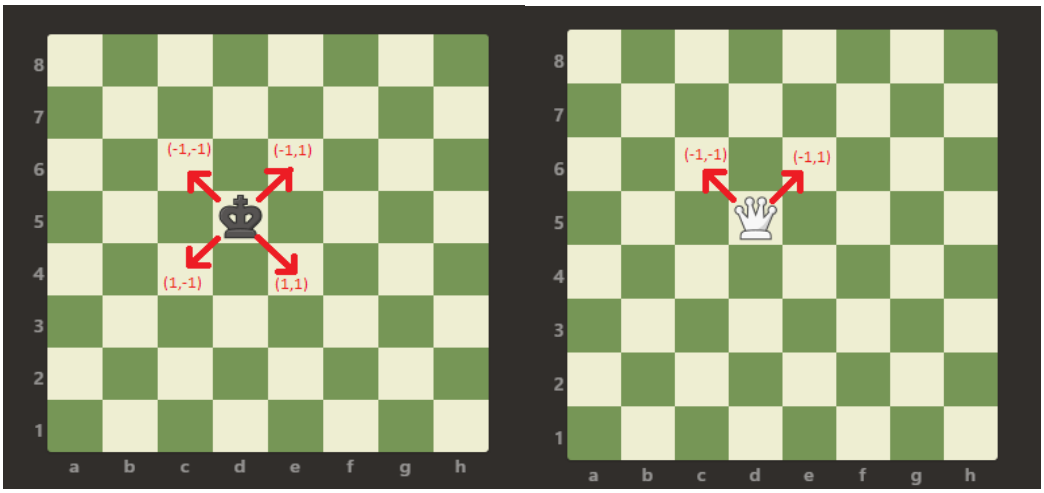

Figure 1.2: (Left) valid moves for fox. (Right) valid moves for hounds.

As seen above in figure 1.2 the fox can move diagonally 1 step in any direction; however, the hounds can only move diagonally forward from their current position. Once the hounds move forward there is no way for them to back track. Below are some possible end game scenarios for the hounds in figure 1.3.

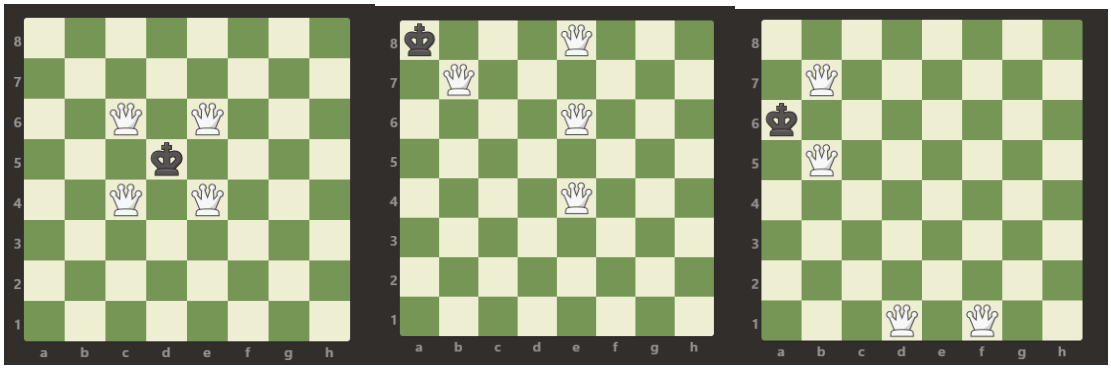

Figure 1.3: Example winning scenarios for hounds

The hounds can also enter no-win scenarios for themselves which exist when all the hounds have moved past the row the fox is on. For example, if the hounds have reached row 8 before the fox has reached one of their winnings squares on row 1 the hounds have lost. Human players who play this game may realize that once the hounds

have all passed the fox the game is over, but for an ML agent this may not be apparent. I will have to account for this scenario which I will discuss in a later chapter. Below in figure 1.4 you can see a sample no win scenario for the hounds. Because the hounds can no longer move forward and they cannot backtrack they have no chance of victory.

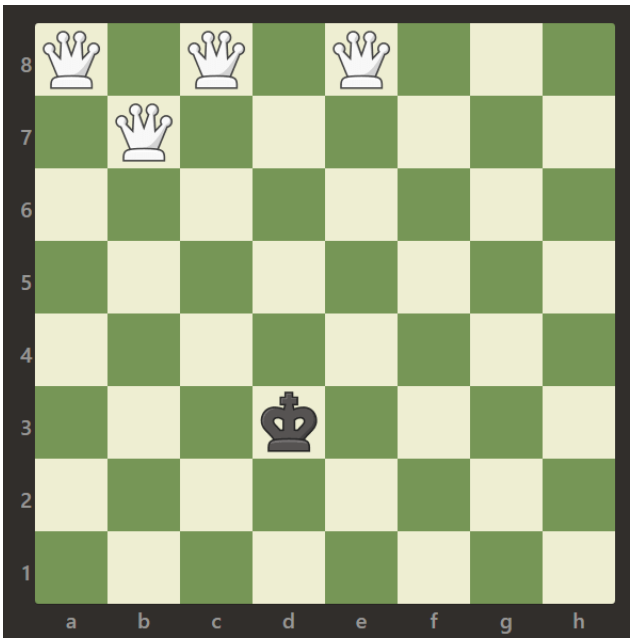

Figure 1.4: Example no-win scenario the hounds can enter

1.3 Objectives

My overall objective was to create two agents, one for the fox and one for the hounds, utilizing a DQN structure and perform different types of analysis on both agents. After I had created the agents, I tested them under several conditions to see how well they would perform. My primary goal was to pit the agent controlling the fox and the agent controlling the hounds against each other to see which one has better performance.

I also performed tests to see how each of the agents learned against varying levels of AI opponents. It became clear over extensive testing that one agent was more

dominant than the other, so I decided to explore other methods of improvement outside the structure of a DQN. I also have added the ability for humans to play against both agents to gauge their abilities.

Chapter 2: Literature Review

2.1 Reinforcement Learning

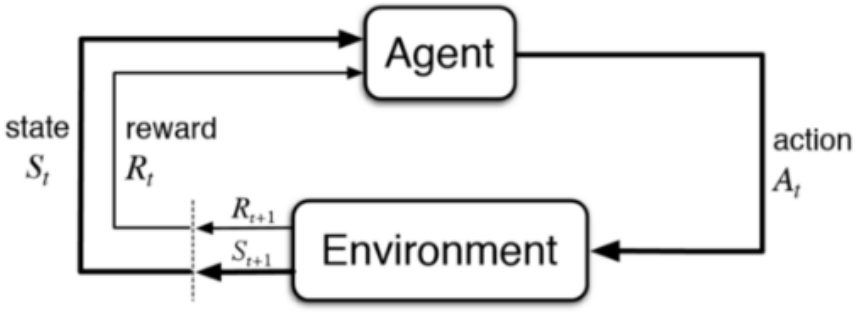

Figure 2.1: The agent acts upon the environment to obtain new information

In terms of reinforcement learning (RL), an agent is a model which learns to act within an environment. RL is a computational approach to learning whereby an agent tries to maximize the total amount of reward it receives when interacting with a complex, uncertain environment [1]. The goal of RL is to learn the optimal policy, or the optimal approach for an agent to act within an environment.

Think of the agent as an actor in a play, and the environment the stage in which the actor acts upon. At any given moment the actor can perform several actions inside of the environment. If the actor performs well, they are rewarded with cheers from the audience. If the actor performs poorly they are punished with boos from the audience. Every action that the actor performs always takes us to a new state in which the actor must decide on what to do next.

The RL loop in figure 2.1 can be summarized as a Markov Decision Process (MDP). An MDP Contains:

- A set of possible states ‘S’.

- A set of possible actions ‘A’.
- A real valued reward function R(s,a).
- A description T of each action’s effects in state.

I also assume the Markov Property which states, “the effects of an action taken in a state depend only on that state and not on the prior history.” [2]. Looking back to figure 2.1 you can see that an MDP helps define the variables needed for our RL loop.

2.2 Deep Learning

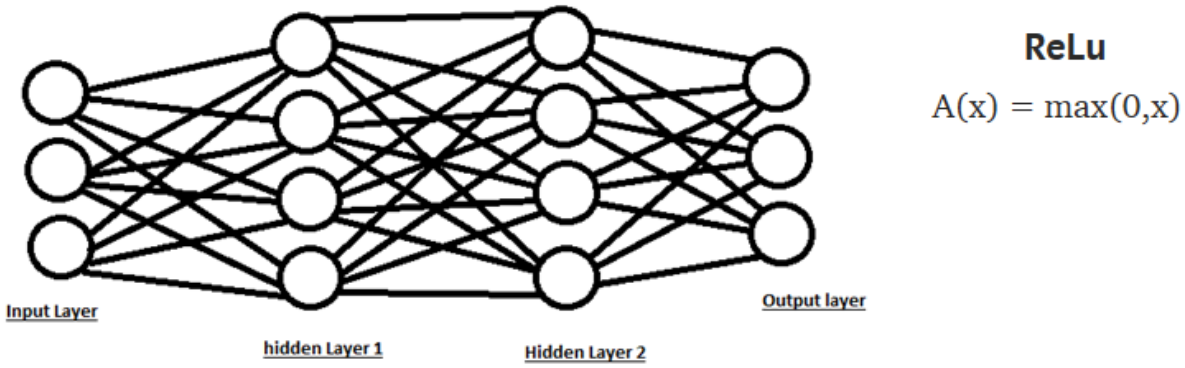

Figure 2.2: A deep, densely connected NN and the activation function between layers

Deep learning (DL) is a subfield of machine learning concerned with algorithms inspired by the structure and function of the brain called artificial neural networks (ANN) [3]. An ANN is a network of connected nodes, and each node is called a neuron. Each connection in the network has a weight, a bias, and an activation function. Weights are values that are housed in a neuron based on the input data fed to the ANN. This data could be the current state of a game board for example. Biases are numbers that are designed to make sure the expected outcome occurs, and they are fine-tuned over time as a ML model learns. Both the weight and the bias are fed to the activation function for the neuron, and these values will decide if the neuron will fire.

The activation function's output will vary depending on the function used, and the one I elected to use for my project was a rectified linear unit (ReLu). In figure 2.2 you can see the formula that ReLu utilizes. The variable x is the sum of the weight and the bias for the current neuron, and if x is greater than 0, then the neuron will fire sending the value of x to the next hidden layer. A negative value for x will cause the neuron to send 0 instead. When the weight alone is not enough for the neuron to fire as necessary, the bias will act as a correction to make sure the intended outcome occurs. Weights alone are not enough to ensure the intended outcome occurs. Since weights are multiplied against the input value passed in to the neuron, if the input value is 0 it will wipe out the weight leaving the bias alone to determine if the neuron should fire. The importance of the bias builds over time as a ML model learns to act upon its environment with the data it receives.

In figure 2.2 you can see an example of what an ANN can look like. An ANN can take many forms, but for it to be considered deep it must contain at least two hidden layers in its architecture [4]. The core of DL consists of an ANN that contains an input layer, at least two hidden layers, and an output layer. This is also referred to as a deep neural network (DNN). If the network has fewer than two hidden layers it is simply referred to as a neural network (NN).

There are multiple ways to connect the layers of an ANN. Figure 2.2 refers to the fact that each layer in its network is densely connected. This simply means that given two layers of nodes in a network, all the nodes in the first layer are connected to the nodes in the second layer.

Another type of connection is called a convolutional layer. Convolutional layers apply a convolution operation to the input, passing the result to the next layer. The convolution emulates the response of an individual neuron to visual stimuli. If an ANN utilizes convolutional layers between hidden layers it is typically referred to as a convolutional neural network (CNN). CNN's have become very popular in the field of ML because of their ability to take raw pixels as inputs. This means that an ANN can utilize image data as an input and convert the image into a 2-dimensional array to be fed as input to the ANN. A CNN can still work with non-image related data if needed.

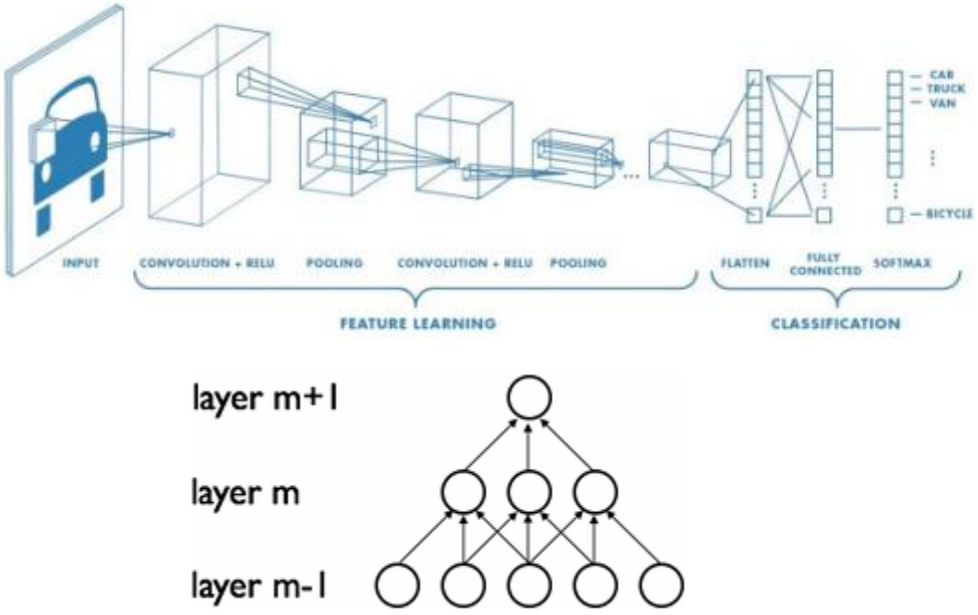

Figure 2.3: The process of converting an image to data in a CNN

Source: https://goo.gl/v6nrDn

This allows us to create a network that can generalize problems more quickly and efficiently than networks with only densely connected layers. CNN's are not fully connected between layers. This means there are fewer calculations that are needed as data

passes through the network, and there are fewer neurons in the network waiting for calculations to complete as they would be in a densely connected neural network. In figure 2.3 you can see how a CNN converts an image to data, and how the layers are interconnected.

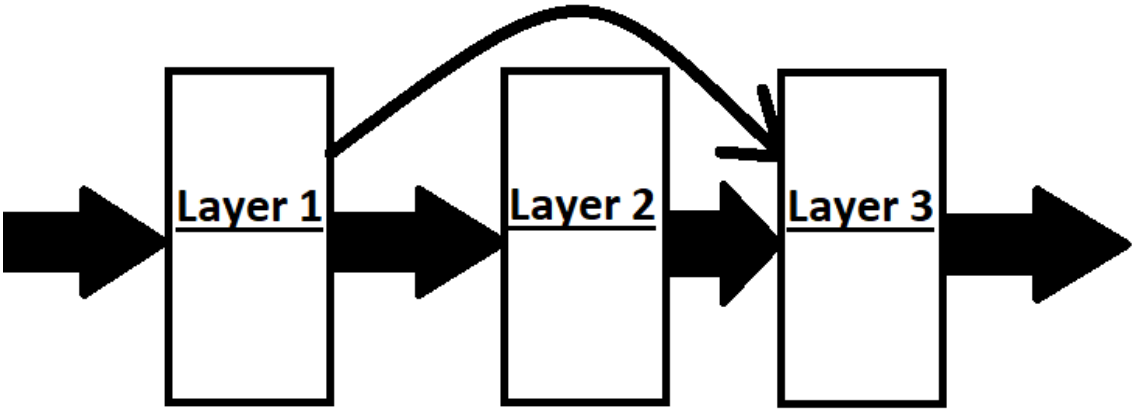

Figure 2.4: Example of a residual connection

The final type of connection that I will discuss is a residual connection. Networks that contain these types of connections are referred to as residual neural networks (RNN). RNN's utilize skip connections or short-cuts to jump over some layers [5]. In figure 2.4 you can see an example of a skip connection represented by the smaller arrow going from layer 1 to layer 3. This type of structure has shown to be largely beneficial to ANN's due to the fact it protects against the vanishing gradient problem suffered by other type of ANN's.

When you have a DNN with several layers and neurons this can cause instability in training the network. The network learns through back-propagation, which means that the values of the weights and biases are updated in a process that feeds backwards through the DNN. This value that is updating the DNN is called the gradient. As the

gradient works backwards through the network it is calculated as the product of multiplication between the layers.

During this process the initial layers are the last to be updated and this is where the vanishing gradient occurs. The initial layers rely on all the layers after them to update their gradient. However, if layers towards the end of the network compute a small gradient, when this is passed to the beginning of the ANN the gradient will no longer be able to update the weights and biases effectively. The updates will become disproportionally small compared to the weights and biases of the early layers prohibiting the network from learning at all, and this is what is referred to as the vanishing gradient problem. A RNN manages to protect against this problem by allowing layers to communicate with more than just the layers immediately before or after it. This allows the gradient to propagate backwards successfully and prohibit the vanishing gradient problem from occurring.

2.3 Q- Learning

Q-Learning (QL) is a subset of RL. Chapter 2.1 showed that RL is focused on states, actions, and rewards at given time steps to teach an agent an optimal policy. QL shares the same overall goal as RL, but the added twist is that QL does not require a model and is a model-free algorithm in the realm of ML [6].

QL will converge on an optimal policy for any finite MDP. It accomplishes this task by maximizing the expected value of the total reward over all successive steps starting from the current state [7]. The “Q” in QL comes from the name given to the

function that provides the reward. People typically associate the “Q” to stand for quality of an action taken in each state.

$$Q = \begin{array}{c|cccccc} \text{State} & \multicolumn{6}{c}{\text{Action}} \\ & 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 0 & 0 & 0 & 80 & 0 \\ 1 & 0 & 0 & 0 & 64 & 0 & 100 \\ 2 & 0 & 0 & 0 & 64 & 0 & 0 \\ 3 & 0 & 80 & 51 & 0 & 80 & 0 \\ 4 & 64 & 0 & 0 & 64 & 0 & 100 \\ 5 & 0 & 80 & 0 & 0 & 80 & 100 \end{array} \qquad R = \begin{array}{c|cccccc} \text{State} & \multicolumn{6}{c}{\text{Action}} \\ & 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & -1 & -1 & -1 & -1 & 0 & -1 \\ 1 & -1 & -1 & -1 & 0 & -1 & 100 \\ 2 & -1 & -1 & -1 & 0 & -1 & -1 \\ 3 & -1 & 0 & 0 & -1 & 0 & -1 \\ 4 & 0 & -1 & -1 & 0 & -1 & 100 \\ 5 & -1 & 0 & -1 & -1 & 0 & 100 \end{array}$$

Figure 2.5: An example Q-matrix, and reward matrix

In figure 2.5 you can see an example of a Q-matrix. The rows represent the current state and the columns represent the action to take. More formally it is defined as Q(s,a) which represent the given action at a current state. In QL the Q-matrix is essentially the brain where the decisions on actions will be made. The reward matrix contains the reward for the action taken in a given state.

When you observe the Q-matrix in figure 2.5, if you look at the states to be rooms and the actions to be entering a different room you can determine the best path to our goal from any room in the house. The reward matrix identifies room 5 to be our goal room by assigning the value of 100 as the reward for entering that room. The value -1 in the reward matrix symbolizes that those rooms are not connected. The value 0 in the reward matrix symbolizes that 2 rooms are connected, but the action does not lead to the reward room.

For the Q-matrix, the values in the table are the confidence that the action taken will lead to a reward. As you may have noticed these numbers vary, but simply following

the path of greatest confidence in a fully trained network will lead you to the reward as fast as possible. For an untrained network the values will update during a process of trial and error until it converges on an optimal policy.

The Q-matrix confidence values are set arbitrarily when it is initialized, and it will take the agent time to discover the optimal policy. The agent learns this policy from experience, but it takes an extremely large amount of attempts at finding the goal to get a firm understanding of what the best policy truly is. The following formula is the heart of how the Q-matrix ultimately gets updated.

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

Figure 2.6: Updating the Q-matrix

In figure 2.6 you can see the formula for how to update the Q-matrix and the terminology in the equation will now be defined.

- $\boldsymbol{Q(s,a)}$ –This contains the current policy for a given state and action pair.
- **r** – This is the reward for a given state and action. Rewards can be defined in several ways including in matrix form such as by R(s,a) similarly to the Q-matrix.
- $\boldsymbol{maxQ(s_{t+1}, a)}$– This represents all future states and actions in the formula and returns the max Q value from them.
- $\boldsymbol{\alpha}$ – This is the learning rate which is a hyper-parameter. This controls how the model reacts to new information. When this variable is close to 1 it treats

new information with the highest priority, and when this variable is close to 0 it treats new information with extremely low priority.

- $\boldsymbol{\gamma}$ $-$ This is the discount rate which controls how far into the future the agent will look when it comes to rewards. This variable is typically a number between 1 and 0. If this variable is set to 1, then all rewards no matter how far into the future will be considered with equal weight. If this variable is set to 0, then the algorithm will act greedily only considering the reward for the next step and no other. This is another variable where finding the right balance is important for the given task. This is also a hyper-parameter.
- $\boldsymbol{t}$ $-$ This is simply the current time step of our iteration.

There is one more thing to be considered before evaluating the final algorithm. The topic that I will discuss briefly is exploration vs. exploitation. Let's return to the example of our agent trying to locate the goal room of room 5 from figure 2.5. If the agent on the first run found the goal room in twenty steps, after the Q-matrix is updated it now knows a path to room 5. What is to stop the agent from taking the route again?

Once the agent has a policy that can tell it how to get to the end goal, the agent will repeat that policy without fail. This means that if the agent does not receive a push of some kind then it will be stuck in repeating the same actions forever and will never learn the true optimal policy. This is the exploration vs. exploitation dilemma.

There is a rather elegant simple strategy called the epsilon-greedy strategy. The epsilon-greedy strategy employs the use of two variables, epsilon and a randomly generated number. Both epsilon and the randomly generated number are values between

1 and 0. If the random number is larger than epsilon then the agent will take a random action instead of an action based on its policy. If the random number is smaller than epsilon, then the agent will continue to follow its policy acting deterministically. If epsilon is set to 1 then the agent will always act deterministically, and if epsilon is set to 0 then the agent will always act randomly.

The last piece of information to introduce before seeing the final formula is the definition of an episode. An episode is all actions taken from a starting state to reach a goal state.

```
Initialize Q(s, a) arbitrarily
Repeat (for each episode):
   Initialize s
   Repeat (for each step of episode):
      Choose a from s using policy derived from Q
         (e.g., ε-greedy)
      Take action a, observe r, s'
      Q(s, a) <-- Q(s, a) + α [r + γ max_α,Q(s', a') - Q(s, a)]
      s <-- s';
   until s is terminal
```

Figure 2.7: Pseudocode for QL

In the pseudocode in figure 2.7 you can see all the pieces discussed have come together. This is QL and it is a completely model-free approach to ML.

2.4 Deep Q-Learning

To this point I have already covered three major concepts in ML; however, Deep Q-Learning (DQL) is where they all come together. As the name implies, DQL is a marriage of the concepts of DL and QL. Looking back at figure 2.5 you can see that our Q-matrix is relatively small for the problem presented and it is easy to picture all the possible states.

Now try to imagine creating a Q-matrix for all the possible states for the game of chess. This task quickly becomes impossible as chess has more states than there are atoms in the universe. Even relating it to the game of fox and hounds will net you a difficult venture in trying to create the correct sized Q-matrix.

DQL keeps all the concepts discussed in section 2.3, but it replaces the Q-matrix with ANN from section 2.2. The core concept here is that the ANN is going to assume the role of the Q-matrix and simulate it within its structure. Since the ANN is going to simulate the Q-matrix, I must determine how the ANN will receive the information from the environment.

The two layers of an ANN I did not discuss in detail were the input layer and the output layer which I will do now. For the remainder of this section I will use the game of fox and hounds for our example. The input layer is where you pass the current state of the game to the agent. This is where the ANN can effectively determine what is going on in the game at the current moment.

There are several ways to pass this information to the ANN, and these methods depend on what type of structure you elected to utilize such as a CNN or a RNN. I elected to utilize a densely connected neural network. I also elected to represent the board as a two-dimensional array of integers. This array is what I ultimately pass to the ANN in the input layer. This way the ANN can see the entire current state of the board. There are more specifics to the parameters which I will cover in chapter 6.

Past the hidden layers is the output layer which is the final layer I did not discuss in the previous section. The output layer ultimately is where the ANN decides to provide

an action; however, I need to define how the ANN will provide the action it would like to select. All the moves that can be done ultimately get mapped to an integer value and the agent selects from one of the available integers and this is the method I elected to utilize. The key pieces of information are all in place for our core understanding of how an ANN can successfully augment a QL agent.

The input layer provides the states of the game and the output layer will provide the actions needed. Recall from 2.3 that the Q-matrix is comprised of state and action pairs or Q(s,a). This is how our DQL agent can successfully replicate the elegance of QL with the power of DL. The states are provided to the ANN in the input layer, and the output layer will provide the action. Once that action is taken, the loop will continue in the same form as it did previously in section 2.3 until the final state is reached. There is only one more wrinkle which we will throw in to complete the DQL agent and that wrinkle is called experience replay [8].

Think of experience replay as the memories of the ANN. As the model plays the game it continues to collect states, actions, rewards, and checks if the game has ended. Once the game has ended, experience replay takes the game that was played and plays it back for the agent in reverse. In the game of fox and hounds this allows it to see who has won and the steps the game took to get to that final state. Experience replay is a major part of the algorithm that helped the researchers over at DeepMind Technologies create an AI agent that could beat most Atari games at a super-human level [8].

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = argmax_a' Q(s,a')
      carry out action a
      observe reward r and new state s'
      store experience <s, a, r, s'> in replay memory D

      sample random transitions <ss, aa, rr, ss'> from replay memory D
      calculate target for each minibatch transition
            if ss' is terminal state then tt = rr
            otherwise tt = rr + γmax_a' Q(ss', aa')
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

Figure 2.8: The complete DQN algorithm with experience replay

The pseudocode in figure 2.8 shows that the overall structure of the QL algorithm remains largely intact. The changes of adding an ANN and experience replay where the two major additions utilized to create a DQN agent. The two new pieces of terminology that are important to understanding the code are the terms batch and mini-batch.

A batch is a sample from an episode or episodes of variable length. For my project I elected to use the batch size of 32. This means that when experience replay occurs my agent will train on 32 samples of: state, actions, next state, and reward tuples. The batch size has heavy implications on how fast your network will train, because once it gathers the required size of 32 samples the network will train on that batch. The bigger the batch the longer it will take the network to train. This can cause the network to learn more with bigger batches, but it significantly reduces the performance during training as the training loops are larger.

A mini-batch is a single sample from a batch that contains: a state, an action, a reward, and the next state. The mini-batch is used in the main loop of experience replay on each iteration to analyze the events that unfolded on that turn in the game. The main loop in experience replay runs for the size of the batch. With a batch size of 32 the loop would occur 32 times and there would 32 separate mini-batches analyzed every time experience replay occurs.

2.5 Application of Literature to Project

For my project I utilized an ANN with five densely connected layers. I chose to do only densely connected layers because it is the quickest of the three types of connections discussed to implement and understand. Because this is my first venture into ML, a proof of concept was more beneficial than trying to create something completely optimized. I also utilize a DQN structure with experience replay as described in section 2.4. I elected to use a batch size of 32 because this seemed to the average length of a single game in most cases. This ensures that my agent will see enough states to learn patterns, but not have a batch size so large that the training runs slowly.

Chapter 3: Technology Stack

3.1 Operating System and Hardware

I elected to use Linux 16.04 LTS as my operating system as Linux seems to have more widely available support amongst the ML community. For training the agents I had the choice of utilizing a cloud service or my local machine and I chose the latter. My local machine is equipped with a NVIDIA 1080TI GPU which has proven to be capable of handling training for my project.

3.2 Programming Languages and Libraries

The code for my project is written purely in Python. As of this writing many of the libraries written for ML have heavy support for Python. The following is a list of libraries I used within my project with an explanation of its use:

- NumPy – Assists with matrix mathematics
- TensorFlow – Primary ML library for computations
- Keras – Used to create the actual ML Model within the code
- Matplotlib – Visualization of data provided from model
- CUDA + cuDNN – GPU libraries created by NVIDIA to communicate with the TensorFlow and Keras libraries
- Anaconda – used for installing CUDA and cuDNN
- Pip – used to install NumPy and Matplotlib as well as TensorFlow (GPU version)
- Python 3.6 – Coding language used for development
- Pygame – Library used to create GUI chess board environment

Chapter 4: Applied Methods

4.1 State Representation

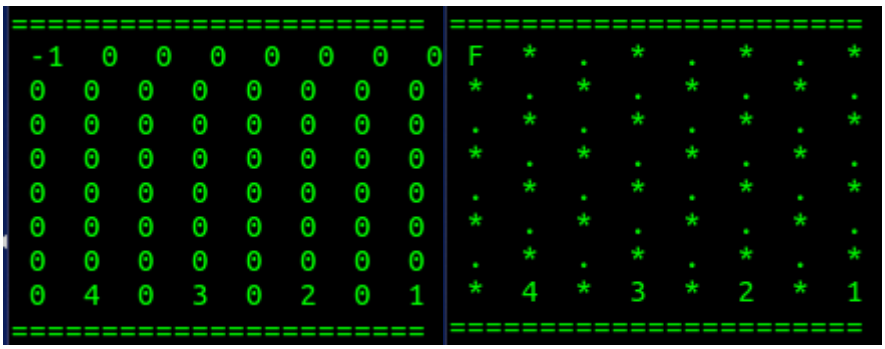

Figure 4.1: (Left) How the AI see's the board. (Right) How humans see the board.

As I discussed earlier in chapter 3, the state is what the ANN will use to see the board. In figure 1.1 I used a king to represent the fox and queens to represent the hounds. This was to provide a visual differentiation between the two players. For the ANN I pass the game board as the state with all the pieces on the board. In figure 4.1 you can see the two-dimensional array that the ANN uses to see the current state of the board on the left panel, and on the right panel you can see the output I give to the human, so they can also assess the current game state.

Piece	Description
-1	The Fox
0	An empty position on the board
1	Hound 1
2	Hound 2
3	Hound 3
4	Hound 4

Figure 4.2: Description of pieces shown in state for figure 4.1

4.2 Action Representation for Fox and Hounds

Value	Fox Action
0	Move the Fox (-1,-1) or back-left
1	Move the Fox (-1,1) or back-right
2	Move the Fox (1,-1) or forward-left
3	Move the Fox (1,1) or forward-right

Figure 4.3: Fox action commands to be given by agent

As I discussed briefly in chapter 3, I cannot just hand the game to the agent and tell it to start learning. I had to define a way for the agent to provide me the desired - actions that it can perform. Above in figure 4.3 are the actions for the fox and the integer value I have assigned each action. The fox can perform 4 possible moves, and I give this information to the output layer of our ANN so it knows how many moves it can make. The output layer of a given ANN seeks to know the total number of actions it can take at a given time step.

Value	Hound Action
0	Move Hound 1 (-1,-1) or forward-left
1	Move Hound 1 (-1,1) or forward-right
2	Move Hound 2 (-1,-1) or forward-left
3	Move Hound 2 (-1,1) or forward-right
4	Move Hound 3 (-1,-1) or forward-left
5	Move Hound 3 (-1,1) or forward-right
6	Move Hound 4 (-1,-1) or forward-left
7	Move Hound 4 (-1,1) or forward-right

Figure 4.4: Hound action commands to be given by agent

The difference between the fox ANN and hound ANN breaks down to the action size that will be fed to the output layer. The fox simply has 1 piece with 4 possible moves as discussed earlier which means the action size is 4 for the fox. Figure 4.4 displays all the possible actions for the hounds. The hounds have a total of 4 pieces, but these pieces can only move forward in two directions giving them a total action size of 8.

4.3 Rewards and Handling Illegal Moves

Fox Action	Reward
Win	+10
Loss	-10
Forward move	+0.25
Back Move	-0.50

Figure 4.5: Rewards and penalties for fox

For the agents to learn how to play, they need rewards and penalties to know if they are performing well or poorly. In figure 4.5 you can see the rewards and penalties provided to the fox agent as it plays the game. The largest reward for the fox is to win the game which will earn the agent 10 points. I wanted to reinforce to the fox that the only goal it should have is to win the game any way possible. There are 2 possible penalties which we will discuss further.

A loss is the worst possible outcome for the agent which is why losing is worth -10 points. I want the fox agent to associate being trapped by the hounds as a negative, and ultimately the fox wants to avoid being blocked on all four possible movements. Another goal for the fox is to continue moving forward when there is an opening to do so which is why I reward it when it moves forward with 0.25 points, but punish it for moving backward with -0.5 points. This way it encourages the fox agent to reach the goal when it has an opening to do so and not back track needlessly.

Finally, I had to find a way to handle illegal moves. My first method for handling this was to punish the agent when it made an illegal move. When the agent has the controls, it is not explained what it can and cannot do. For example, I did not tell the agent that when it is at position (0, 0) on the board that position (-1, -1) is illegal. I only

gave the agent the controls to move, and it was up to the agent to learn what it could and could not do by the rewards and punishments which is why I originally punished the agent for taking an illegal move. Punishing the agent was effective over time, but it still took valuable training time for the agent to learn some moves are invalid.

I then discovered that you can directly alter the predictions that the ANN will make by changing the values in its prediction. The agent's prediction will be an array of the size of how many actions it can perform, and the values will be a confidence percentage of what moves it wants to take. If you sum the values of the array the total will equal 1 representing 100%. The indices of the array represent the action values the fox can take in figure 4.2. By setting invalid moves to 0 or below, you are telling the agent that the action will not be taken. When the agent begins its learning process it will also back-propagate your adjustment, meaning future predictions will know a move is invalid based off the corrected value provided. I elected to utilize this method in the end, because it saves the agent a large amount of time with trial and error making invalid moves. This configuration also eliminated the need to punish the agent for making an invalid move as it can no longer make invalid moves.

Hound Action	Reward
Win	+100
Lose	-100

Figure 4.6: Rewards and penalties for hounds

Figure 4.6 shows the potential rewards and penalties for the hounds' agent. You may notice that the hounds have heavier rewards and punishments than the fox. Since the hounds receive no rewards while the game is being played, it becomes important for the rewards and punishments to become more potent. Also recall that hounds have a goal that is not as simple as the fox's goal. The fox simply must reach the end of the board to score points, where the hounds must trap the fox in such a way where the fox has no valid moves to make.

The hounds can win the game from any position on the board except the squares where the fox can win which are: b1, d1, f1, h1. Because of this ambiguity, I originally rewarded the hounds with 5 points for forcing the fox to move backwards.

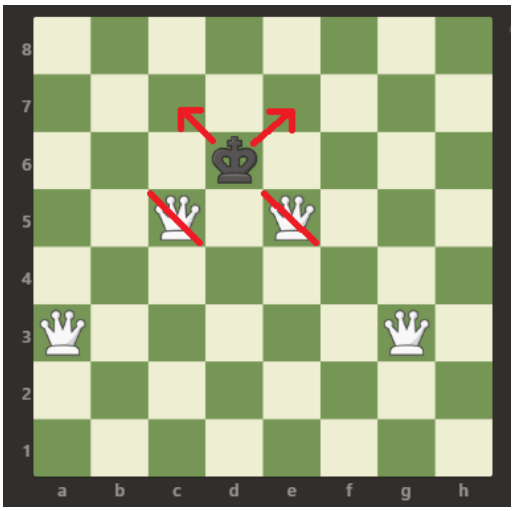

Figure 4.7: Hounds block forward moves for fox causing it to retreat

In figure 4.7 you can see an example of the hounds causing the fox to retreat. The logic behind this reward is that it takes at least two hounds working together for them to cause the fox to retreat. This reward encourages teamwork amongst the hounds which I had hoped would lead the hounds to the optimal strategy. I also created a punishment of -5 points if the hounds let the fox advance from its current position. This punishment's

purpose is to provide the hounds with an incentive to impede the progress of the fox. Despite the hounds showing promising results with this reward structure, I did not see enough progress to continue forward utilizing these rewards and elected to use the rewards in figure 4.6.

I elected to handle illegal moves exactly the same as I handled it with the fox. If a move is illegal, I set the value for that move to 0 which guarantees that the move will never be selected. This allows the agent to focus on simply improving at the game rather than learning what it can and cannot do legally on the board.

4.4 Neural Network Hyper-parameters

Fox agent hyper-parameters		
Variable	Value	Description
self.state_size	(8,8)	Represents 8x8 board
self.action_size	4	4 total actions the fox can take
self.memory	30000	Size of the agents memory (how many states it retains)
self.gamma	0.95	Discount on future rewards
self.epsilon	0.9	Controls if the fox acts randomly or deterministically
self.epsilon_min	0.6	Minimum threshold epsilon can reach
self.epsilon_decay	0.9995	How fast epsilon decreases over time
self.learning_rate	0.001	Control how much the agent retains after an epsiode
self.batch_size	32	Number of states used in experience replay
episodes	10000	Total number of games to train for

Figure 4.8: Hyper-parameters for fox agent

In figure 4.8 you can see the full list of hyper-parameters for the fox agent. I deploy the epsilon-greedy strategy in order to give the agent a chance to explore moves it normally would not when acting deterministically. This strategy is designed to prevent the agent from locking in on 1 strategy it may try to repeatedly exploit that may be sub optimal. I augment this strategy by allowing epsilon to decay over time and become smaller. This will ultimately prompt more exploration as the agent plays through more

and more episodes. I do not let epsilon fall below 60% because I want the agent to act deterministically more often than not.

The learning rate seen in figure 4.8 is designed to let the agent retain its information at a slow pace. This way the agents don't allow anomalies to override its policy. When the agent is learning from experience replay, the batch size controls how many states will be played back to the agent so it can learn from them. This variable has a heavy impact on how fast episodes are completed because experience replay occurs frequently. A single element in a batch, called a mini-batch, contains: a state, action, reward, and the next state attained after the action was taken.

Hound agent hyper-parameters		
Variable	Value	Description
self.state_size	(8,8)	Represents 8x8 board
self.action_size	8	8 total actions the hounds can take
self.memory	30000	Size of the agents memory (how many states it retains)
self.gamma	0.95	Discount on future rewards
self.epsilon	0.9	Controls if the fox acts randomly or deterministically
self.epsilon_min	0.6	Minimum threshold epsilon can reach
self.epsilon_decay	0.9995	How fast epsilon decreases over time
self.learning_rate	0.001	Control how much the agent retains after an epsiode
self.batch_size	32	Number of states used in experience replay
episodes	10000	Total number of games to train for

Figure 4.9: Hyper-parameters for hound agent

Above in figure 4.9 you can see the full list of hyper-parameters for the hound agent. They remain largely unchanged from the fox agent as I would like to keep things as even as possible to keep a level playing field for both agents. The one notable change here is that the action size for the hounds was raised from 4 to 8 to account for the different actions the hounds have. Both agents share 10,000 episodes during their training which is set arbitrarily. During training each agent saves its model after 10 games played to make sure it can continue to build off what it has already learned.

4.5 Network Architecture

```
def build_model(self):
    #builds the NN for Deep Q-Network
    model = Sequential() #establishes a feed forward NN
    model.add(Dense(64,input_shape = (LENGTH,), activation='relu')) #input Layer
    model.add(Dense(64, activation='relu')) #Hidden Layer 1
    model.add(Dense(64, activation='relu')) #Hidden Layer 2
    model.add(Dense(64, activation='relu')) #hidden Layer 3
    model.add(Dense(self.action_size, activation = 'softmax')) #Output Layer
    model.compile(loss='mse', optimizer='Adam')

    return model
```

Figure 4.10: Construction of the fox ANN

In figure 4.10 you can see the construction of the ANN for the fox. Thanks to the TensorFlow and Keras libraries, the creation of the model is relatively easy. In each line of code you may notice that the model starts off with the word dense, signifying a densely connected layer. 64 represents the total number of neurons in that layer, and this was decided largely due to the board size.

Between each layer is an activation function which I have chosen to use ReLu as shown in figure 4.10. The activation function oversees what data gets to move on to the next layer of the ANN. The ReLu activation function is the most commonly used activation function for deep learning models [9].

In the output layer you can see where the total number of moves is fed to the agent. Also, you can see that the activation function is different from all other layers where a ‘softmax’ is utilized. Softmax will provide a distributed prediction amongst all the possible actions provided. In the case of the fox with four possible actions, the softmax activation function will give a prediction of which move will most likely return success amongst the four possible moves.

For the loss function I utilize the mean squared error (MSE) method. This method measures the predicted outcome from the ANN vs. the actual outcome. It takes the difference of the two results, the actual outcome and predicted outcome, and squares it. This result is back-propagated through the network in order to update the values in hopes of making the predictions better in the future.

The optimizer I elected to use is the Adam optimizer. The Adam optimizer is utilized to assist with optimizing the neural networks and decrease the loss between the actual outcomes vs. the expected outcomes. It employs a strategy which utilizes momentum to get out of local minima and maxima. Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient [3].

```
def build_model(self):
    #builds the NN for Deep Q-Network
    model = Sequential() #establishes a feed forward NN
    model.add(Dense(64,input_shape = (LENGTH,), activation='relu')) #Input Layer
    model.add(Dense(64, activation='relu')) #Hidden Layer 1
    model.add(Dense(64, activation='relu')) #Hidden Layer 2
    model.add(Dense(64, activation='relu')) #Hidden Layer 3
    model.add(Dense(self.action_size, activation = 'softmax')) #Output Layer
    model.compile(loss='mse', optimizer='Adam')

    return model ##passes model to hound agent
```

Figure 4.11: Construction of the hounds ANN

In figure 4.11 you can see the construction of the ANN for the hounds. Code wise, it is structurally equivalent to the fox ANN. The difference lies in the action size as I have discussed previously where the hounds have a total of 8 possible actions. The softmax activation function will distribute its prediction amongst the eight possible outcomes.

4.6 Double Deep Q-Network for the Hounds

A Double Deep Q-Network (DDQN) is simply a DQN that employs the use of a 2nd neural network to keep the training balanced. Think of a DDQN as a DQN with a supervisor neural network monitoring the DQN. The DQN is updated every iteration, but the supervisor ANN is updated periodically to keep things in balance.

This method is utilized to prevent a single ANN from overestimating or underestimating values for moves. When utilizing a DQN structure the ANN is updated on every single iteration, and if the play out on certain iterations was not favorable this could cause the ANN to destabilize and give less reliable predictions in the future. In my DDQN I update the supervisor ANN every 50 games in order to give the supervisor a larger sample size of data to absorb. This method is designed to keep a few bad games from ruining optimal strategies the ANN may have learned. In chapter 6 I will discuss the results of this implementation including comparing it to the normal DQN implementation.

Chapter 5: Results

5.1 AI Fox vs. Random Hounds

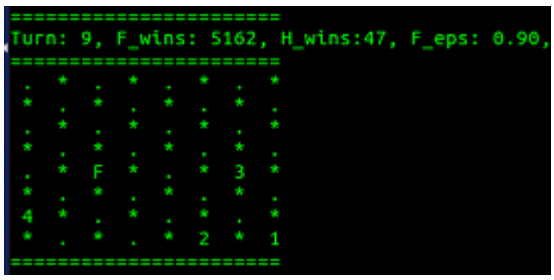

Figure 5.1: A snapshot from a game (results above)

One of my first tests for the fox was to face off against random hounds. I did not expect the random hounds to win very often because the design of this test was to see if the fox truly was learning. In figure 5.1 you can see the fox agent has already claimed 5162 games successfully in this current run. The average score was 11.34 points, and this is significant because the maximum score the fox agent can achieve is 11.75. The fox agent gets 10 points for winning, and since it earns 0.25 points per move forward, if it moves forward 7 consecutive times it will get 1.75 points totaling 11.75 points. When training the model, the first few games typically saw the fox wandering around aimlessly until fox lost or the fox stumbled upon a victory square.

In this stage, typically the number of turns taken was very high because the fox had not learned what rewards it positively to this point. Once the fox reaches its goal state and earns a positive reward, this information back-propagates through the ANN updating the weights and biases. The next few episodes usually see the fox trying to reach the same goal state as it has before.

This is where epsilon plays a large role in learning. By allowing the fox to perform a random move by some epsilon factor, the fox will eventually land on a new

winning square causing the neural network to update. After a lot time has passed and the fox has discovered all of its possible positive reward squares the problem shifts to navigating the hounds.

Fortunately for the fox agent, the hounds are random which means they typically do not work together. In this training exercise more often than not the fox was simply impeded by one or two hounds which it typically avoids with ease. The results in this training where very encouraging as the fox not only showed improvement in win percentage, but also in the number of turns it took to win against random hounds.

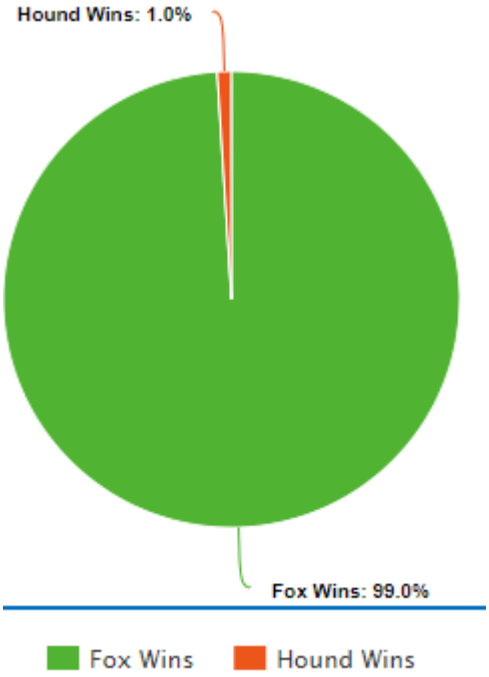

Figure 5.2: Graph of Average win percentage of AI Fox vs. Random Hounds

In figure 5.2 you can see the average win percentage which is rounded down for the fox. Once the agent had trained for over 100,000 games, I noticed the fox agent performed well enough that a random hound agent could not really compete with the fox agent.

5.2 Random Fox vs. AI Hounds

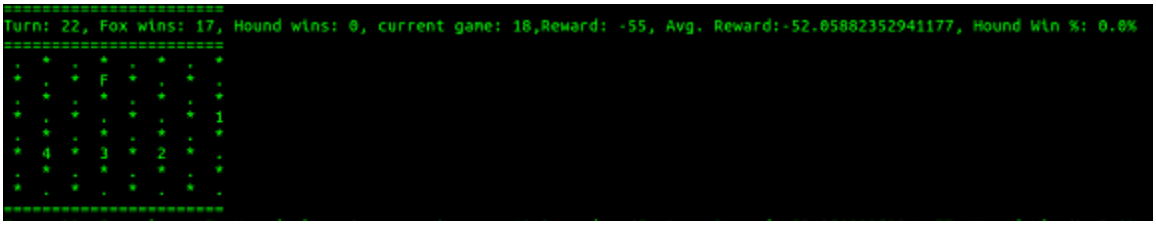

Figure: 5.3: Hound agent vs. random fox (results above)

Due to the complex nature of the end-game for the hounds, the agent requires an extremely-large number of episodes before it begins to converge on an optimal policy. I would also like to point out that the hounds do have an optimal strategy that will always win the game, but as of this writing my best performing agent has yet to achieve this strategy and it has trained for well over 1,000,000 episodes.

I assumed that the end-game for the hounds would require a large amount of time to learn an optimal policy. Because of this factor I believed it was best to start the hounds off against a completely random fox agent in hopes of having it learn its goal. The hounds struggled to capture the fox over a very long stretch of training. This poor performance I attribute to several factors which I will outline in this section.

Figure 4.6 in chapter 4 outlines the rewards for the hounds that have achieved the best results. I did reward the hounds at one point for forcing the fox to move backwards, but ultimately, I felt this reward structure became more of a hindrance than achieving my desired results. I instead opted for a more generic reward structure, and my hope was for the hounds to generalize the problem to be able to handle multiple different gameplay scenarios. I also at this time uncovered a logic error I had created in my code and the combination of fixing the error and applying the rewards had promising results.

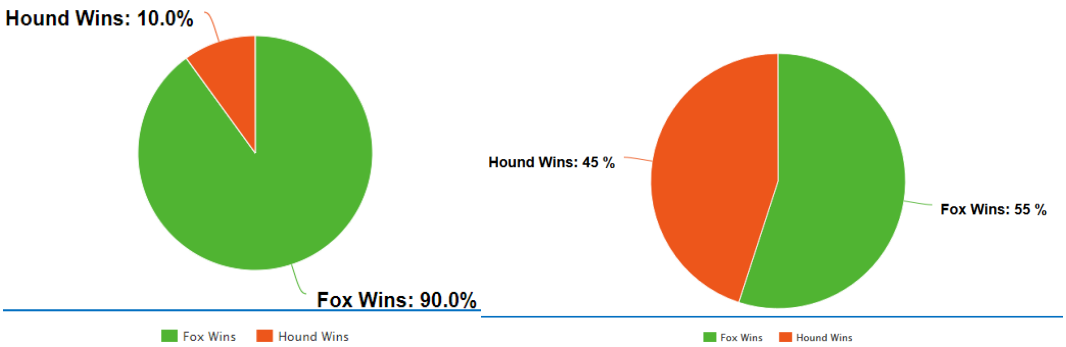

Figure 5.4: (left) Hound win % before change (right) Hound win % after change

After making this adjustment and over 100,000 episodes later, the hounds win percentage did increase by almost 35%. In figure 5.4 you can see the results from before and after the change. Although these results are encouraging, the increase seemingly plateaued and did not grow beyond 45%.

When testing the performance of the ANN I set epsilon to 1, which means that the agent will always act deterministically. This way I can accurately gauge its performance without a random move interfering with the agent's decisions. Even with everything seemingly stacked in the hound's favor, the agent still wins less than 1/2 of the time against a vastly inferior opponent. I do believe that it can still perform better, but the issue here might be with the DQN structure as I will discuss in a later section.

5.3 Fox AI vs. Hound AI

In this section I will discuss the performance of both agents as they play against each other. The results of section 5.2 show that the hounds' agent has a lot of ground to make up. My first primary test was to see how the agents fared against each other with no training.

In section 5.1 I outlined that the fox agent quickly was able to identify a winning strategy and then began to exploit that strategy until a new one was discovered. The fox agent now playing against an opponent not acting randomly managed to repeat the same

results as it had in its first test. Once the fox learned its goal over the first few episodes of its life, the agent for the hounds did not stand a chance.

This is largely attributed to the fact that the fox usually learns its goal first, and then begins to optimize its policy right away. The hounds may not win a game for over several hundred episodes which can cause the agent to fall behind the fox drastically. The secondary point which we have already discussed is the difficulty of the goal for the hounds to achieve. Even though the hounds may end up trapping the fox in a corner that does not guarantee that the same events will occur in the next episode. It became clear during the training that the hounds would not be able to compete with the fox if both agents started evenly.

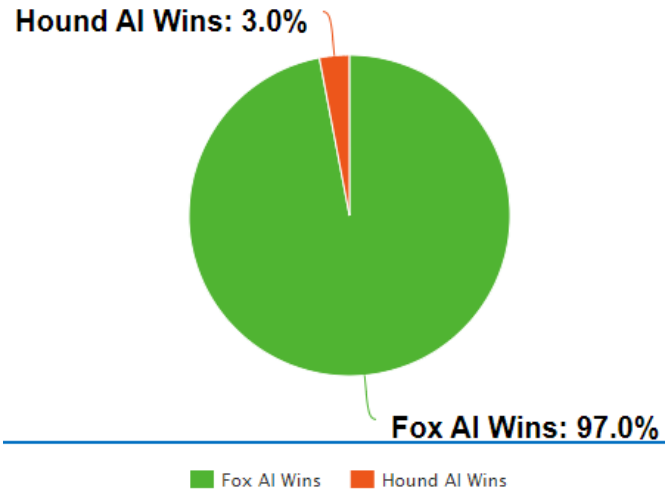

Figure 5.5: Average win percentage of both AI's vs. each other

Figure 5.5 unfortunately tells a gruesome tail off loss for the hounds' agent. There are still important takeaways from the graph in figure 5.5. The first is that the agent for the hounds managed to perform better against the fox agent than a random hound agent. Unfortunately, the hounds' agent only manages to out-perform the random hounds by

only 2%. Even as the training progressed the hounds would only close the gap by tenths of a percent over the period of several days.

During this time, I wanted to test if a trained hounds' agent could have any success against a fox that was not trained. Despite very early promising returns, as the fox agent begins to receive more feedback it eventually will continue its dominance over the hounds' agent. The results over 100,000 games ultimately register a slight uptick in win-percentage for the hounds but not enough to break 4%.

5.4 Fox AI vs. Hound AI with DDQN

As discussed in chapter 4, the DDQN was implemented for the hounds in order to prevent the agent from destabilizing during training. The hounds had proven to this point that they needed every possible advantage that they could afford against the fox agent which is what lead me to implement this structure. Despite some promising early returns, even a DDQN proved to not be enough for the hounds to overcome the agent for the fox. Training with the DDQN for over 100,000 episodes, the results for the DDQN unfortunately fell in line with the results from figure 5.5.

5.5 Human vs. Trained Fox AI

After I trained the fox agent sufficiently, I created a method for humans to play against each agent. The fox agent continued to impress, but this time with a human controlling the hounds the fox did not fare nearly as well as before. The fox agent does show solid awareness for gaps it can attack if given the chance, but the agent also displayed a lack of situational awareness which will be discussed in chapter 6. My goal for these tests was not necessarily to win, but to test how the fox responds in several situations.

In each situation the agent seems to greedily attack openings when it can. The major downfall in its performance is it does not seem to grasp the bigger picture of the game and how it loses. I found that the agent is aware that it wants to avoid position a8 when threatened as it is very easy for it to lose in this position. Other positions that it tries to avoid are positions on the edges of the board, but I feel this leads to some indecision when it finds itself on the edge of the board. Overall despite some of the abnormalities the agent performs capably against a human opponent.

5.6 Human vs. Trained Hounds AI

The agent for the hounds had not progressed well enough to test in a game of wins and losses. The tests in this section were purely situational, and mainly designed to see if the hounds' agent could detect possible chances to win. Unfortunately, despite the agent showing some awareness of general scoring positions such as a8 on the board, it does not seem to realize fully that the fox should be its target.

The agent will blindly rush to a8 sometimes completely ignoring if the fox piece is in that location. Other times the agent does show some willingness to utilize multiple pieces and block the fox. But unfortunately, the agent for the hounds prefers to rush individual points more often than not and shows a complete lack of situational awareness.

Chapter 6: Further Discussion of Results

6.1 Overall Evaluation of Fox AI

With over 1,000,000 games played for the fox AI, I am thoroughly impressed with the speed of which it recognizes how to accomplish its primary task. However, this does not mean this agent performs perfectly in every given scenario. When the fox agent has an unimpeded path to one of the winning squares the agent performs as you would hope directly moving toward a target. When the hounds provide the fox agent some resistance the fox sometimes acts irrationally.

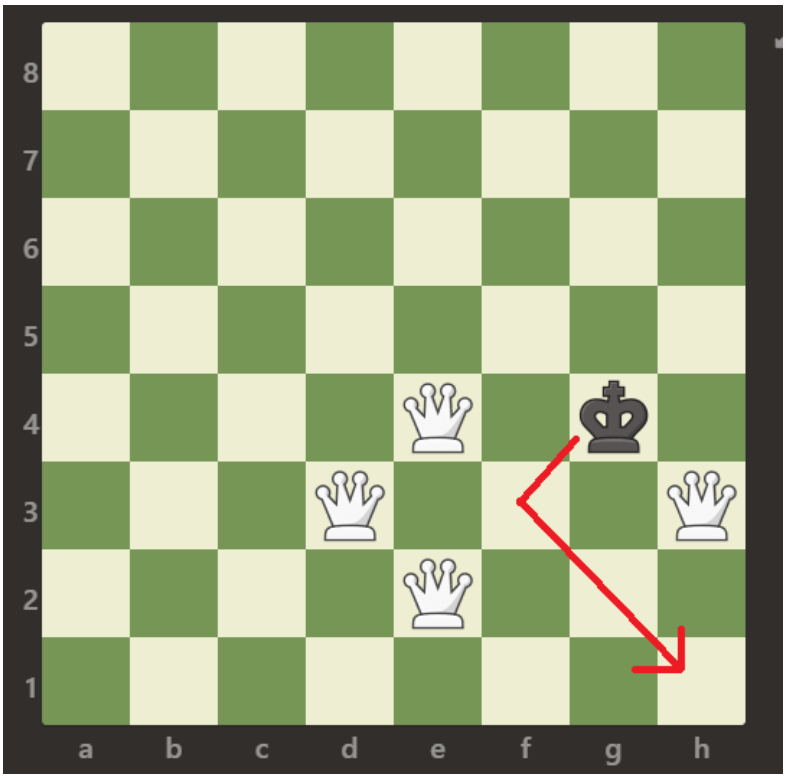

Figure 6.1: Example of the fox agent failed to make

In figure 6.1 you can see a move the fox agent can make where it has certain victory if the moves are taken as outlined. Inexplicably, even though this move is available the agent has had odd occasions where it retreats to either square behind it.

These odd events typically occur on the edges of the board where there is some resistance by the hounds. This situation doesn't occur often, but it does pop enough to be noticed. I believe with more training in these situations they should eventually smooth themselves out.

When there is pressure in the middle of board the agent seems to be more confident in its decision making. It seemingly retreats when appropriate and attacks when there is an opening. I attribute this to most of the games being played in the middle of the board prompting the agent to have more confidence in these situations. Overall with some flaws the agent for the fox seems to perform admirably.

6.2 Overall Evaluation of Hounds AI

1,000,000 plus games played for the hounds' agent and it still feels that it has not trained enough. There are times where the hounds will offer a glimmer of teamwork forcing the fox to retreat. There are also times where the hounds will have the fox in a virtual no-win scenario, and somehow fail to recognize the situation. This occurs far more frequently than one would hope which is why they boast a 3%-win rate.

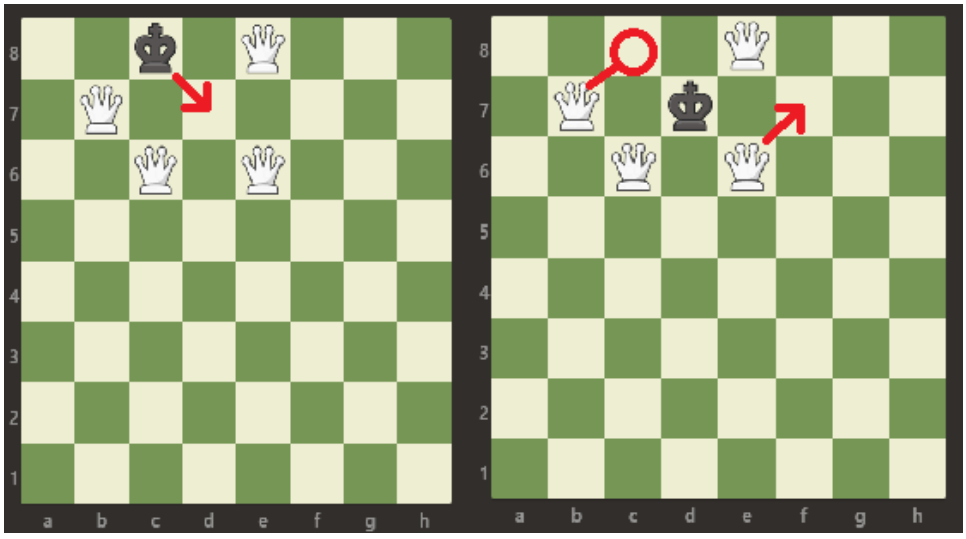

Figure 6.2: An example of a winning scenario lost by the hounds' agent

In figure 6.2 you can see an example where the hounds failed to detect the winning strategy. The hounds had two moves left to make to secure victory. After the fox makes a move forward to d7, the hound on b7 could move to c8 and secure the victory. The agent elected to move the hound on e6 to f7 instead which ended up costing the hounds the win.

Unfortunately, I cannot say at this point that the hounds perform even near human level. The agent continuously makes mistakes that humans playing at a novice level would not make. More training may ultimately make the agent converge to the optimal policy, but progress only continues forward at a snail's pace. DQN's have shown they can handle games as complex as chess, but ample time to train with other speedups typically accomplish this task. Either more time, or other methods augmenting the DQN may be the way to go when attempting to have it converge on the optimal policy.

Chapter 7: Conclusion and Future Work

7.1 Conclusion

My objective when I started on this journey was to create 2 agents to play the game of fox and hounds utilizing a DQN structure for both agents. I was able to accomplish this task and apply all the tests I sought out to perform to see which agent ultimately would become the best. Even though it became easy to see the fox agent was superior to the agent for the hounds', I still believe it is possible for the hounds to converge on the optimal strategy to win every game. I believe with a better reward strategy and augmenting the DQN for the hounds' agent with different techniques can cause it to ultimately learn the optimal strategy. I also managed to implement a DDQN for the hounds, but ultimately failed in seeing the hounds converge on an optimal strategy.

7.2 Future Work

Future work for my project will mainly be focused on improving the hounds' performance. I believe that an attempt to use different methods to augment the DQN such as Monte Carlo tree search (MCTS) could have major benefits to the hounds' agent performance. I feel a CNN will speed up training and may be useful in the search for an optimal strategy. A final addition I would like to see is expanding the game to 2 foxes and 7 hounds. Since the current game has a scenario where the hounds can always win, it would be interesting to see how the game would play out with more pieces on each side.

References

[1] Richard Sutton and Andrew Barto. *Reinforcment Learning: An Introduction.* The MIT Press, 1998.

[2] Bob Givan and Ron Parr. *An Introduction to Markov Decision Processes.* https://www.cs.rice.edu/~vardi/dag01/givan1.pdf

[3] Jason Brownlee. *What is Deep Learning?.* https://machinelearningmastery.com/what-is-deep-learning/. 2016.

[4] Harrison Kinsley. *Introduction to Deep Learning*. https://bit.ly/2NozLEQ. 2018.

[5] He Kaiming, Zhang Xiangyu, Ren Shaoqing, and Sun Jian. *Deep Residual Learning for Image Recognition.* https://arxiv.org/abs/1512.03385. 2015.

[6] Leslie Kaelbling. *Learning an Optimal Policy: Model-Free Methods.* https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/node23.html. 1996.

[7] Francisco Melo. *Convergence of Q-Learning: A Simple Proof.* http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wiestra, and Martin Riedmiller. *Playing Atari With Deep Reinforcement Learning.* https://arxiv.org/pdf/1312.5602v1.pdf. 2013.

[9] Daniel B. *Rectified Linear Units (ReLU) in Deep Learning.* https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning. 2018.