Assessing Deep Q-Learning Models Through OpenAI Gym

James Bocinsky
College of Engineering
University of Florida
Gainesville, USA
jbocinsky@ufl.edu

Lorenzo Tabares
College of Engineering
University of Florida
Gainesville, USA
tabareslorenzo@ufl.edu

***Abstract*—In this document, The Mean []'s team (T-1), is proposing to train a deep neural network coded in TensorFlow through Keras, using reinforcement learning, to play a game. The premise of the project will be to create a few generic reinforcement learning neural networks of varying architectures and experiment their performance across different openAI games. This will allow the team to be able to compare how the different models' performance compares across different games. Ideally, the models will be generic enough to perform well in many different categories of games and through experimentation the results were determined. Through this project, the team learned how to interface a deep neural network with an external game. While doing so, reinforcement learning was used to allow the neural network to learn its best solution, rather than guiding it to a human guided solution with labeled data.**

Keywords—deep learning, machine learning, neural network, q-learning, reinforcement learning, tensorflow, keras.

I. INTRODUCTION

In this paper, we examine the performance of a deep neural network trained using reinforcement learning, specifically, deep q-learning, to beat various OpenAI Gym video games. We will analyze and determine whether the models provide an optimal solution to beat the games and analyze the results of our models to determine what types of games it can perform well on.

In general, an artificial neural network is a set of interconnected processing units that receive input from external sources. The connected units compute an output that may be propagated to other units, [1]. Artificial neural networks are an application of the field of deep learning which has gained much popularity in recent years. Artificial neural networks were created to model the behavior of the brain. A deep neural network is typically identified as an artificial neural network that has at least five hidden layers. [12] Each hidden layer is a set of interconnected processing units within the network which receives, processes, and feeds information to the next hidden layer or output layer. [13] Figure 1 shows an example of a single-hidden layer neural network.

In this paper, we train a few neural networks to beat two different OpenAI Gym control games using a variation of reinforcement learning called deep q-learning. A reinforcement learning model will use an environment, the state of the game, an agent, the player of the game, and a set of actions and rewards in the states to update the model selected. In q-learning, a set of possible states and actions are updated using rewards to find an optimal model for the game.

An investigation and analysis of the performance of our neural networks is detailed in this paper. We will train and test our neural networks using the aforementioned methods and compare the performance to see which game or type of game has the best results. Here we use TensorFlow as the framework for our network with Keras as our high-level design software.

A. Deep Neural Networks

Deep neural networks are artificial neural networks that have at least five hidden layers where each hidden layer is a set of interconnected processing units within the network [19]. These processing units receive and transmit information through the network's layers in a fashion that mimics the functionality of a human brain. For example, neurons in the brain receive signals from other neurons along connections called synapses. Once the receiving neuron is activated, it sends a signal to other neurons through these synapses [2]. The processing units of an artificial neural network function in the same way with neurons being processing units and synapses being weights.

The root of an artificial neural network is built around the famous Rosenblatt perceptron. The Rosenblatt perceptron was introduced in 1958 but is infamous for not being able to solve the XOR classification problem because it was one single perceptron. It consists of one unit that has an input with a weight and one output. Every input unit in the input layer is connected to each output unit in the output layer with each connection carrying a weight [1].

Returning to the brain analogy, input units can be thought of as signals and the output units as processing units. A multi-layer perceptron serves as the skeleton of the few deep neural networks we analyzed [20]. The deep neural network is multi-layered, having hidden layers in between the input layers and output layers [14]. One of our neural networks is longer than 5 hidden layers, so we will compare how a deep neural network compares to the other two more refined simple models.

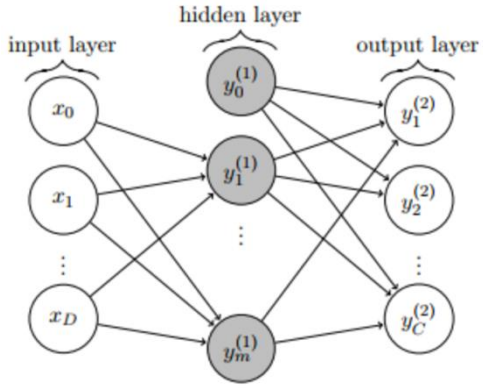

Figure 1: Single-Hidden Layer Perceptron

B. Reinforcement Learning

Reinforcement learning methods model a player as an agent, which is trained using rewards given from every action in an environment, unlike most machine learning models which train on a labeled dataset. [17] This means a model can be trained on an environment it has not seen before. Some methods that fall under reinforcement learning are q-learning, genetic algorithms, and deep q-learning. [18]

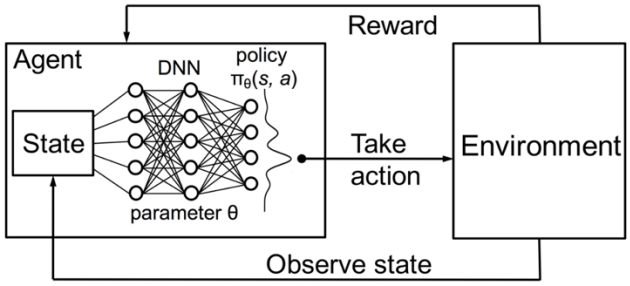

Figure 2: Single-Hidden Layer Perceptron

Our team will focus on implementing a deep q-learning method. Using deep q-learning, the q-table of a typical q-learning algorithm can be estimated with a neural network, possibly even using a convolutional neural network (CNN) with a group of game frames to give the model a sense of movement and location. Due to time restraints and resources, our team was not able to complete a CNN based deep q learning network. Then the model will make a choice, an action, and get rewarded or not for that action [7]. This method will be used by inputting the state of the current environment of the game into the deep neural network, and then based on the reward that was received or not, will update the weights of the network. In doing so, a model can learn through reinforcement without having any prior knowledge about the environment.

C. Deep Q-Learning

For large scale challenges, like to play a game, deep q-learning currently is regarded as one of the best solutions for finding an approximation of the global optimal policy. Before, genetic algorithms dominated, they were useful for finding the local optimal quickly but not the global optima policy. The iterative implementation of q-learning also is not efficient for large scale problems because every state action pair must be computed. The way around the scalability issue is to use a function approximator to reach close to the optimal value function [5]. Implementing a neural network, a close approximation to the optimal value function can be found, thus removing any scalability issues.

Now that it is established why deep q-learning is the optimal solution let us discusses more in-depth about the components. Q-learning is composed of a two by two matrix known as a q-table. The q-table has row vectors as actions and column vectors as states. The q-table is updated by using the Bellman Ford equation [5] (1).

$$Q(s,a) = r + \gamma(\max(Q(s',a')) \qquad (1)$$

The function (1) calculates the optimal action-value policy by taking the maximum future Q value reward times the discounted value, gamma, plus the current reward (2). [4]

$$Q^*(s,a) = E_{s'\sim\mathcal{E}}[r + \gamma(\max(Q(s',a'))] \quad (2)$$

Deep q-learning is applying this algorithm (1) while using a deep neural network instead of a table to approximate the decision-making process. Therefore, since games often have many states, too large for the memory of a computer, we can model and estimate the optimal policy by using a network instead of a table, where the network's inputs are the states and the outputs are the possible actions.

The process of approximating the optimal value function is achieved by minimizing the sequence of loss functions (3).

$$L_i(\theta_i) = E_{s.a\sim\rho(.)}\left[\left(y_i - Q(s,a;\theta_i)\right)^2\right] \quad (3)$$

$$y_i = E_{s'\sim\mathcal{E}}\left[r + \gamma\left(\max\left(Q(s',a';\theta_{i-1}\right)\right)|s,a\right] \quad (4)$$

The target value (4) for the current iteration (i), p(s,a) is the behavior distribution, the probability distribution over the sequence of states and actions and θ are the parameters of the network and $_E$ is the environment in which the artificial intelligence is trained on [4]. The process of optimizing is back propagation, calculating the error at each neuron, through gradient descent.

D. OpenAI Gym Environment

OpenAI Gym is a framework that allows machine learning, algorithm developers to have access to a variety of games to test their models [3]. Our team will use this to train and test our deep q-learning models across a number of games. This will allow us to analyze the models' results in different games, while keeping the majority of the framework the same. This will make the model and game interface easier to use and reduce the

amount of development needed for our training and testing framework.

II. Related Work

A. Deepmind Alpha Go

Go is a game originating from China and popularized over twelve hundred years in Japan. Go is known to be a difficult and complex game for humans to master, and even more so the most strenuous game to master for artificial intelligence. This is due to the overwhelmingly large spatial search and complication of determining which moves and board positions are optimal.

The enormous hurdle to master Go was finally accomplished by Deepmind's Alpha Go. Alpha Go used deep neural networks trained by a duet of reinforcement learning and supervised learning from observing real players that already mastered the game. On top of using neural networks, the Monte Carlo tree search algorithm was used to sample the search space to determine the most advantageous move set. This was used to evaluate each game state [8].

The Monte Carlo tree search implementation in Alpha Go can be broken down into a four-stage selection. First, determine which edges have the highest action value (as shown in figure 3). Second, expansion, when a node is expanded, supervised learning is used to predict the optimal move. Third, evaluation, the value network and Fast Rollout policy (speeds process) assesses each node. Finally, backup, the values returned from evaluation are used to update the action value pairs (as shown in figure 4).

Through these methods Alpha Go was able to reach a win rate of 99.8% over the most recent, best Go playing programs and went 5 - 0 against a professional Go player.

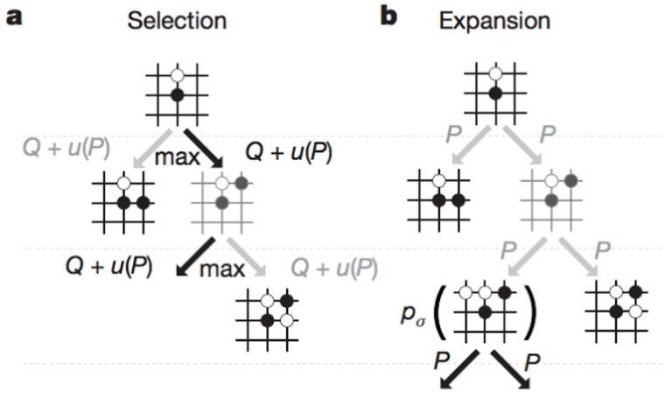

Figure 3: Selection and Expansion phases of Alpha Go's Monte Carlo search tree

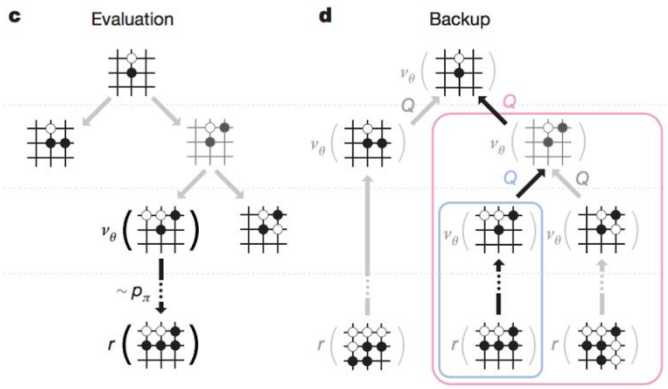

Figure 4: Evaluation and Backup phases of Alpha Go's Monte Carlo search tree

III. System Architecture

Our overall system architecture, consists of a few different tools. We are using OpenAI's gym environment to run the game environment that the agent will learn from. OpenAI's gym environment provides the state and reward that the agent will learn from. For the midterm we focused on the pole cart problem within OpenAI's library of games. This consists of 4 inputs defining the state and 2 possible actions for the output. The agent's actions are produced by the neural network and feed back into the environment. As a result, OpenAI's gym will produce another state and the game will run until an end condition is met. For the pole cart game we ran, the possible end conditions are if the pole angle is +- 12 degrees, if the center of the cart reaches the edge of the screen, or if the episode has reached a length of 500 steps. For the mountain car game we ran, the end condition is if the car made it to the top of the mountain or if it took 200 steps. This process is done iteratively to train the network, where a reward is given for each time step taken. For the pole cart game, the longer the agent is in the environment, the larger the reward for that network architecture. We used 3 different networks to provide analysis of what worked best. The deepest network is a 6 layer, fully connected neural network coded in python. Then we have a 4 layer and 3 layer network which is very similar. Our architecture is designed using keras and uses tensorflow for the backend support. This allowed the team to rapid prototype a model that can be quickly and easily edited in the future to create different architectures for testing.

IV. System Design

A. Network Desgin

The design of our largest neural network is made up of one input layer, five dense/fully connected layers, and one output layer. This brings the overall network to consist of 7 layers. The input layer provides the environment information. For the sake of the classic control problem of balancing a pole on a cart, the

four inputs are the cart position, the cart velocity, the pole angle, and the pole velocity at the tip. The 2 discrete outputs in the output layer of the network are the possible actions that the agent can take. For the sake of this control problem, the possible actions are move cart left, an output of 0 and move cart right, an output of 1. For the sake of the mountain car game, the inputs are the car x position and velocity while the outputs are push left, push right, or no push.

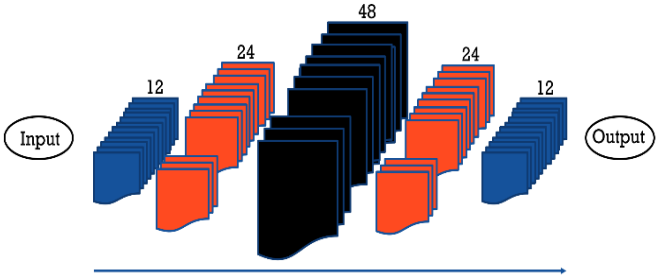

Figure 5: The largest network design we trained

The first layer, the input layer, takes a four-dimensional input vector and is fully connected to the first hidden layer which consists of 12 nodes as shown in figure 5. The second hidden layer has 24 nodes, twice as the previous, and uses the rectified linear unit activation function. The third hidden layer has 48 nodes, twice as the previous, and uses the rectified linear unit activation function. The fourth hidden layer has 24 nodes, half as the previous, and uses the rectified linear unit activation function. The last hidden layer has 12 nodes, again half as the previous layer. Finally, the output layer, has two nodes and uses a linear activation function. After each dense/fully connected layer we used twenty percent dropout (shown in figure 6) to reduce overfitting and encourage the model to learn more robust features.

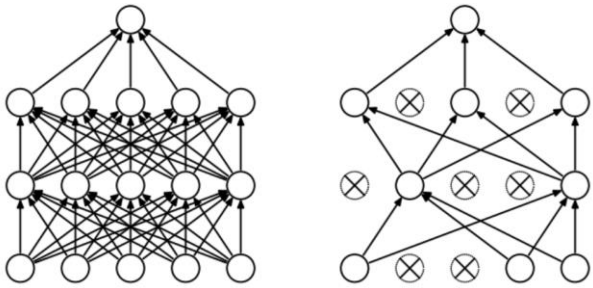

Figure 6: Example of using drop out in a neural network

The other two models are very similar in overall design but just shorter in length. We provided multiple models to be able to analyze the differences in performance for these two classic control problems. The smallest network consisted of 3 layers, the input layer, a fully connected hidden layer of 24 nodes, and finally the output layer. The network that we found to have the best performance was not too short nor too long for the complexity of these games. This network consisted of 4 layers, the input layer, followed by two fully connected layers of 24 nodes and then the output layer. These networks did not consist of any drop out layers in the network as they were not as deep and theoretically should not require it.

B. Memory implementation

To keep track of what our artificial intelligence is experiencing through its actions at each state, we used a deque data structure, imported from the collections package. In the deque each action, reward, state, and next state is stored so it can be referenced in the future. The data stored in the deque is referenced through random sampling before each iteration of training. The amount of random sampling depends on the size of the batch that we set and the current length of the deque. The size of the batch used is determined by calculating the minimum of the batch size and deque length.

An immediate reward is preferred over a reward in the distant future. To account for this a discount rate of 0.93 was used to reduce future rewards.

C. Randomization

At the beginning of training, our agent will primarily randomly choose its actions. This is accomplished by comparing a value named epsilon to a random value between 0 and 1. When the random value is less than epsilon, the model acts randomly. Epsilon is originally initialized to 1 but decreases across time. This is done by dividing epsilon by 1.0005 across every minibatch. Thus, randomization of the model decreases over time. By doing this, the model will explore much of the environment space in the beginning and then continue to learn from its own network further into training. This is important to allow the network to have a vast understanding of the environment, so it can form a correct policy. Afterwards we use Bellman's equation and neural network predictions (via Keras) to figure out the future discount reward. Finally, with this new information we input it into our neural network for training.

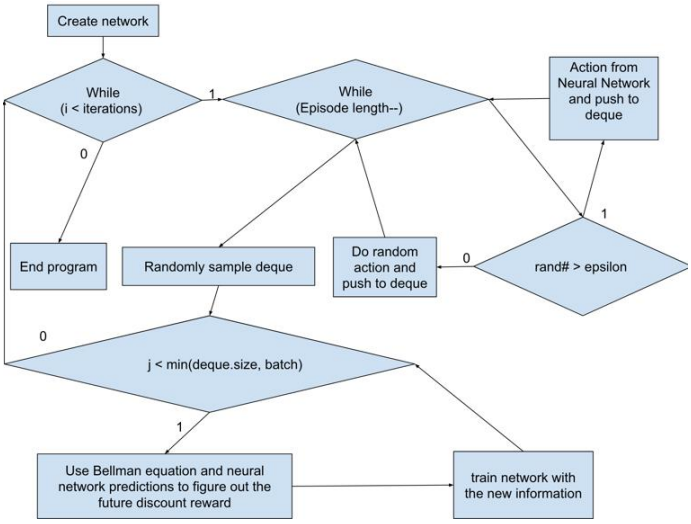

Figure 7: Flow chart of our Design

D. Online and Target Networks

To further develop our design, we implemented the use of an online network as well as a target network. The target network is design to allow us to compute a true target Q-values during each update for evaluating the online network. The target network, network architecture is exactly the same as the online network architecture. The only difference between the target network and the online network is that the weights of the target network are only updated every Tau steps and is kept fixed on all other steps. The updated weights are just the current copy of weights from the online network. As shown in figure 9, the weights are copied over to the target network from the online network. The online network is used to form memory of previous actions and received rewards. The target network then uses this memory from the online network to calculate the loss. As shown in figure 8, the online network stores its observed reward into memory. Later the target network randomly samples the same memory and uses it to calculate the loss. The loss is then back propagated through the network.

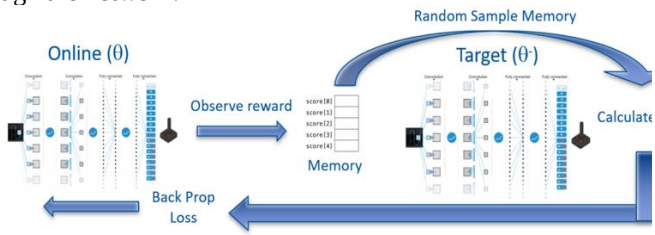

Figure 8: High level algorithm

Figure 9: Shows interaction between target and online networks

V. Performance Evaluation

A. Experimental Setting

The team's objective for this project was to sample different architectures performance and determine a suitable architecture for two classic control problems. These control problems are hosted in OpenAI's gym environment. The control problems tested were the pole cart game and the mountain car game. The team choose to use OpenAI's gym environment for our experimental setting to be able to rapid prototype our network architecture without having to worry about making a custom framework to interface with the game environment. Especially for a reinforcement learning architecture, it is important to be able to easily interact with the environment and receive well defined rewards and penalties. With OpenAI's gym environment, this is possible.

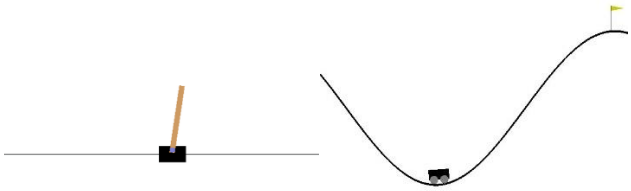

Figure 10: Pole cart game (left), mountain car game (right)

During training, the team experimented with all hyper parameters and came to conclude that with these, the networks performed the best. A batch size of 10, a gamma rate of 0.93, epsilon of .9997, start training after 1000 iterations, maximum of 1000 games played (epochs), a learning rate of 0.001, and a maximum deque memory length of 2000. These parameters were chosen for the following reasons.

A batch size of 10 was used to encourage the network to follow the general gradient of the network but not be too large where it would be suspect to fall in local minima. The gamma rate of 0.93 was used so that the network can look far into the future and still try to reach a large reward but still be small enough so that it can still explore new spaces instead of memorizing a certain pattern to beat the game. We started training after 1000 iterations to make sure that the first SGD optimizations were not weighted too heavily by memorizing what the network did in the very beginning. If this was not prevented, we found that the network would often start the games the same way which may not always be the most optimal. A maximum of 1000 games (epochs) were set to be played for a couple reasons. First, we noticed that our networks were not too large and thus should not require too much training. Secondly, if the network were to converge to an optimal policy solution, it would well before 1000 games. A learning rate of 0.001 was selected as a middle ground between extreme values. [16] This ensured we would not lose our gradient from the vanishing gradient problem. As well, it was small enough that the networks could still learn an exact policy instead of hopping around local minima. Lastly, a maximum deque memory length of 2000 was chosen so that the network would learn from the most previous 2000 instances of the network. We noticed if this was too small, it may not learn from enough of a sample space, while if this was too big, it may be learning from outdated versions of the network. Somewhere in between these extremes is adequate for training.

To ensure our network could end training at an optimal solution, we required the network to beat the game 20 times in a row. Once this happened, we would stop training and use those weights for our final network. This was needed as the results would often jump around, sometimes winning while in the next game completely failing because it would get into a state it had never seen before. This would help the network make sure it has learned enough from the environment while still not overtraining. This was needed to ensure reasonable performance on the networks that would converge.

B. *Performance Metrics*

The team's architecture performance metrics are a few things. The team designed multiple architectures and compared the following. These are loose definitions because depending on the game, succeeding and time rates will be defined differently. Can the network succeed the control challenge? If so, what was the average rate for succeeding? Meaning how long does it take to succeed at the task. We will visualize the scores across training. We should see a gradual climb of score with some jumping around showing that it is exploring different areas of the environment. We will also analyze the training time vs the length of the game. With this we hope to see little training time to reach our defined succession rate/time. Finally, we will observe the average game time rate of our final trained network. We should see it perform well and be stable so that it does not fail or have inconsistent results across runs.

C. *Experimental Results and Analysis*

Overall, our training design was fairly robust to differing network sizes, as you can see in table 1, the networks could beat the games very consistently for 4 of the 6 scenarios.

Table 1: Final Network Scores

Game:	**Network A (24)**	**Network B (24-24)**	**Network C (12-24-48-24-12)**
Pole Cart	38.1	500	500
Mountain Car	13.79	13.29	1.78
Acrobot	-94	-161	-385

Table 1: Average score over 10 game runs using learned network

For the pole cart game, a score of 500 was considered beating the game, where a score of 500 means that the cart could hold the pole up for 500 steps. According to the OpenAI gym, beating the game is considered holding it up for 200 steps. To ensure a well-built policy we defined beating the game to be 500 steps. It was seen that if the network took random steps, that it would typically fail within 10-15 steps. So even though network A did not converge to an optimal policy, it still was better than taking random steps.

For the mountain car game, we modified the reward given to the car to encourage speed. This makes sense for this game as the goal is to build speed to get to the top of the mountain as the car does not have enough power to make to the top without building speed first. If this change was not done, the car would only acquire a reward if it reached the top of the mountain which is more of an exploration problem. Deep q learning networks are limited to their exploration based on their ability to take random steps while getting rewards along the way. This was experimented to be not possible for our network and to converge as well, because the network would have to take random steps for a very long time in training. The score of a game was the average overall speed of the car plus if it made it to the top of the mountain before the end time of 200 steps. The maximum average speed the car could acquire through a game was around 6, so we defined the reward to get to the top of the hill to be 10. This value was important, because if this was too large, the network would only try to go to the top of the hill, to the right, and not be concerned about gaining speed and thus not fail as it could not build enough speed to get to the top. But if this was too small, it would not be encouraged to make it over the smaller slope of the mountain and make it to the top. The higher the score above ten shows that the network not only beat the game but did it with considerable speed as well denoting better performance.

The following images were recorded to allow analysis of the networks and which one we can say is able to generalize an optimal policy the best.

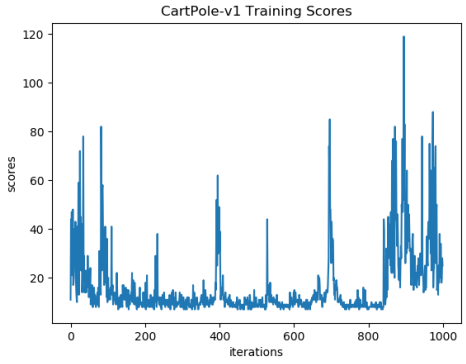

Figure 11: Cartpole, network A training scores across epochs

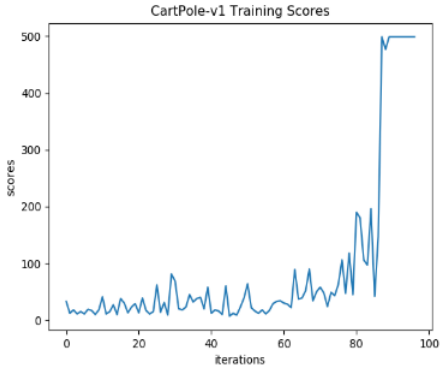

Figure 12: Cartpole, network B training scores across epochs

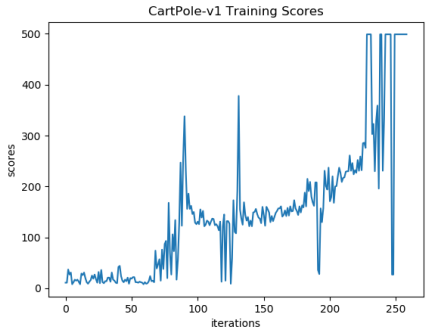

Figure 13: Cartpole, network C training scores across epochs

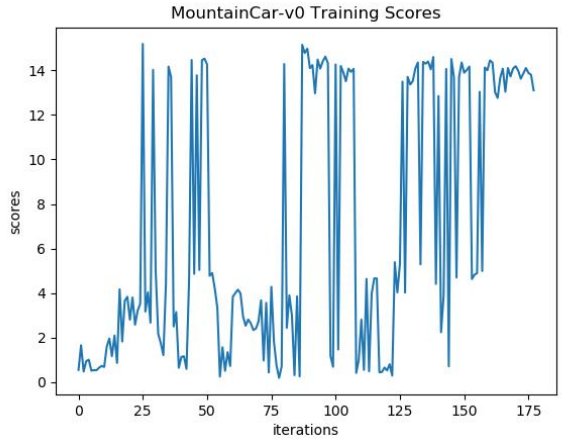

Figure 14: Mountain car, network A training scores across epochs

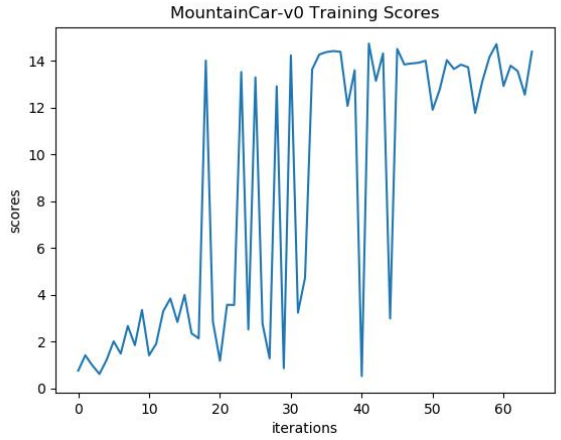

Figure 15: Mountain car, network B training scores across epochs

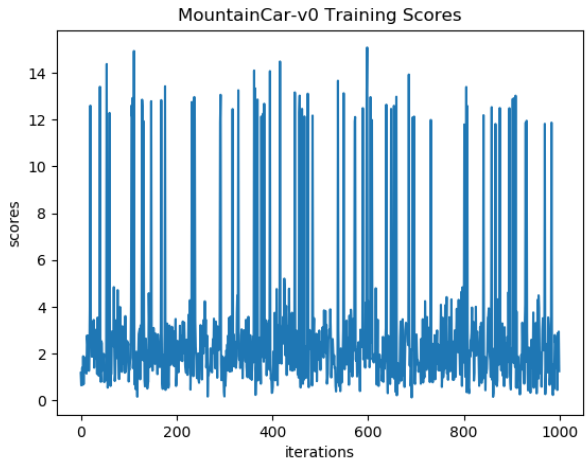

Figure 16: Mountain car, network C training scores across epochs

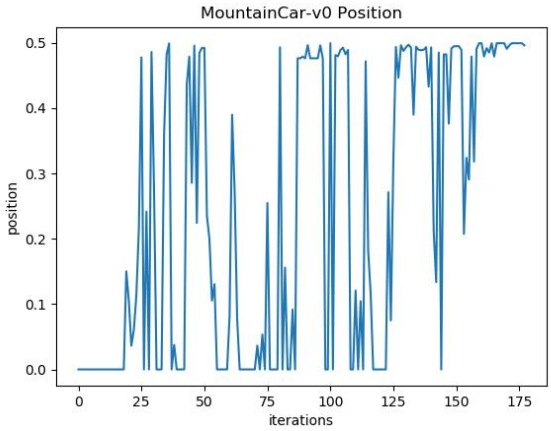

Figure 17: Mountain car, network A maximum position during run. Flag at .5 and start position is 0.

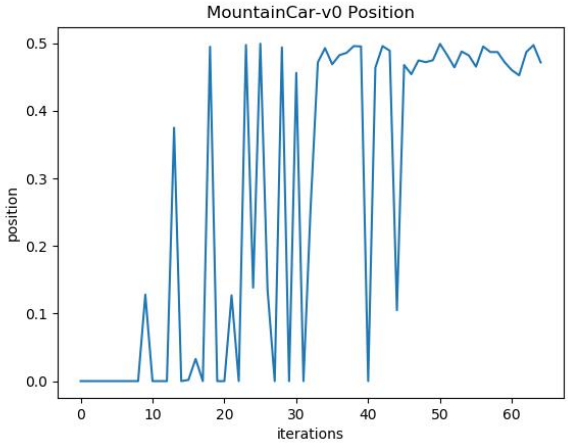

Figure 18: Mountain car, network B maximum position during run. Flag at .5 and start position is 0.

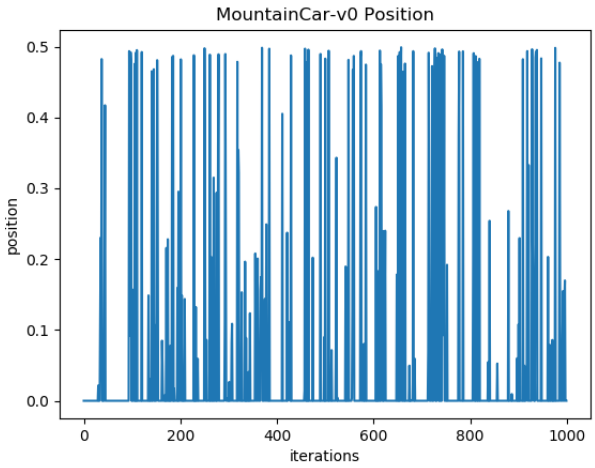

Figure 19: Mountain car, network C maximum position during run. Flag at .5 and start position is 0.

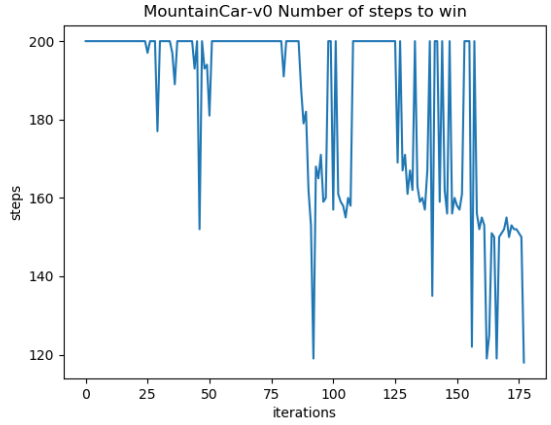

Figure 20: Mountain car, network A number of steps it takes till end game. Game ends at 200. Anything smaller than 200 means it won.

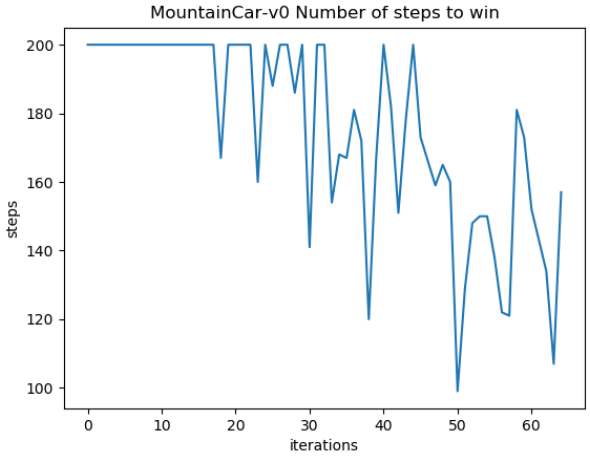

Figure 21: Mountain car, network B number of steps it takes till end game. Game ends at 200. Anything smaller than 200 means it won.

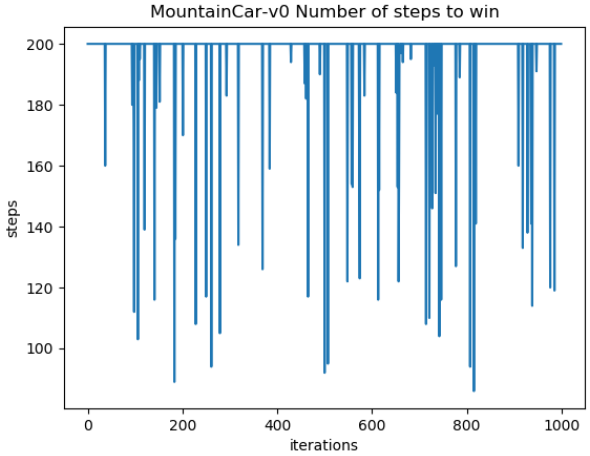

Figure 22: Mountain car, network C number of steps it takes till end game. Game ends at 200. Anything smaller than 200 means it won.

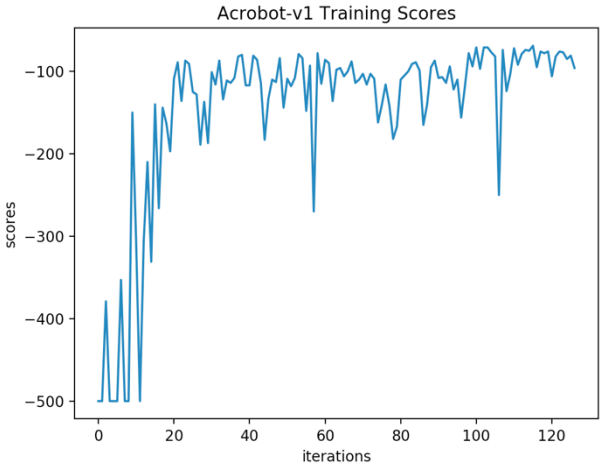

Figure 23: Acrobot, network A training scores across epochs

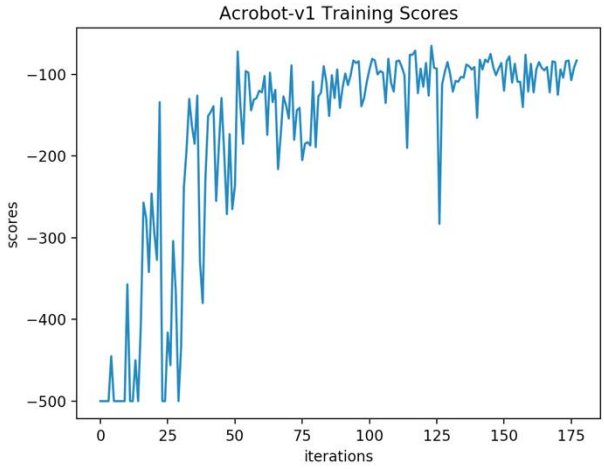

Figure 24: Acrobot, network B training scores across epochs.

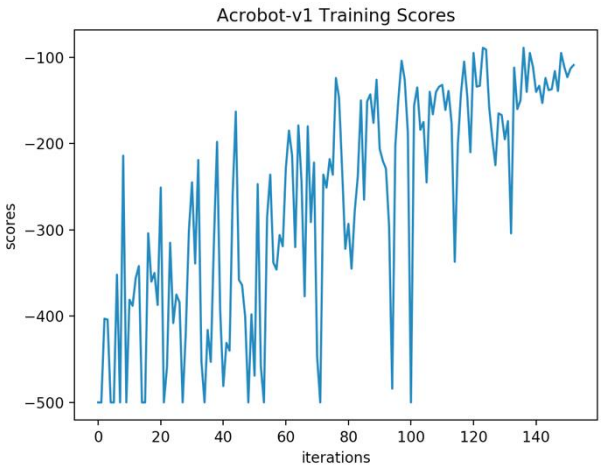

Figure 25: Acrobot, network C training scores across epochs.

Starting with the pole cart game, we can see in figures 11 through 13 that network B and C are able to converge to a final solution after approximately 100 and 250 steps respectively. What's important to see is a gradual general increase in score over time. This can be seen for network B and C but not network A. This means that network A was not large enough to acquire the features it needed from the environment. You can see that it used all 1000 epochs of training but there was no amount of further training that would've allowed it to converge. Since both network B and C converged, the time to converge can be analyzed to pick a better solution. Since B converges more quickly we can conclude that network B is a better architecture for this problem. We expected the deepest network, network C, to perform the best because the amount of states is quite large in this game, even though it is simple. With this we see that it still performs well and even provides a better gradual progress towards a solution, which is what you would want to see in a reinforcement learning network.

The mountain car game needed to be analyzed with more visualized to really see what's going on with the networks, therefore each network was given three visualizations. Starting with figures 14 through 16 we can see how the networks scores changed over each game. What you can see at first glance is that in this case the deepest network, network C was unable to converge to a solution. This is probably because the network is too complex for the environment space which just consisted of the position and speed of the cart. With figures 14 and 15 we can see the two networks that succeeded were networks A and B. With network A, we can see a lot of oscillation along the performance showing that there may not be enough nodes to learn a general enough solution. It also took much more time to find a solution, approximately 175 games, which was essentially a memorization of what to do. In fact while watching network A's training, it would often try to do the same approach once it found one that worked, but when it came across a state it would not have seen before or one that was no longer recorded in the network it would fail. Network B looks to be the best network of these three, it converged quite quickly at about 65 games.

With figures 17 through 19 you can see that the position of the cart would often correspond to the score it received. This validates the assumption that it would be okay to train the cart to learn to increase speed. The score of the cart was the sum of the average speed and if it made it to the top of the mountain or not. This encouraged exploration of the environment space. A good network should show that the position of the cart should increase over time showing that it is exploring more of the environment. This can be seen with network B the most, following it is network A, and then C. Once again, we can see with network C it has difficulty forming a generalized solution to the problem since it has so many layers and nodes.

Figures 20 through 21 show the time it took the network to beat the game. If 200 steps were taken, this means that the network did not beat the game. What a well-trained network should show is that over time it not only beats the game but beats the game in less steps. This would show that it has learned what states it would like to be in, even in the early stages of the game. Thus, it would have learned what states to be in to reach the top of the mountain. We can see that network A and network B perform very similarly in this regard, except the fact that network B requires much less training to learn an optimal policy. Network C on the other hand does not tend to show a pattern when it does beat the game. This shows that the network is unable to learn a consistent approach to beat the game and is just beating the game by luck at this point and is not learning to make it to the top of the hill.

Unlike the Mountain-cart and Cart-pole games, the Arcobot game is unsolvable due to the nature of the game. There is no possible way to get the poles to stand exactly still at a tall position because of the chaos of the system. This means there is not an exact reward that demonstrates winning the game. Despite this added complication network A, B, and C were still able to learn on the environment. Network A performed the best and was able to reach the artificial goal we set in the least amount of epochs. The agent reached the goal when it was able to maintain a score better than -130 for 20 epochs. Network A was able to do this in 126 epochs, the fastest of the three networks, while achieving the best final average score of -94 on those trained weights. Network B was able to reach the conditions set in 177 epochs with an average final score of -161 on those trained weights. Network C was able to reach the set condition in epochs with a final average score of -385. Over 1000 epochs due to strengthen conditions, the agent must maintain a score of at least -100 for 20 epochs, network C performed the best achieving a final score of -97 followed by network B with a final score of -100.2, and finally network A with a final score of -108.7. We think the reason network C performed the worst on the Arcobot game in the first test because the network was under trained. Since network C had the most parameters and did not have as much training, this is believed to be why it did not perform as well.

VI. Conclusions and Summaries

In conclusion we can see that choosing a different architecture based on the problem at hand is very important when designing neural networks. For complex problems, a more complex network is desired but for simple problems, it may not be necessary. We can see through our analysis that of the three networks, network B performed the best. It was able to successfully beat both games 10 times in a row with its final weights. Network A was able to learn a policy for the simpler game of mountain car, which had a smaller environment space, but was not able to learn an optimal policy for the more complex game of pole cart. Similarly, network C was able to learn an optimal policy for the pole cart game, as this was more complex, but it was not able to learn an optimal policy for the simpler environment space of mountain car.

VII. Future Work

If the team had a GPU and more time, the team would be encouraged to implement a convolutional neural network for some of the Atari games within openAI's gym environment. Especially to provide some analysis to what architectures work

best for these games. Many papers that have implemented deep q-learning models brush over the architecture they use, but it has not been seen what kind of affects different architectures have on performance. Even though deep q-learning is an effective design for implementing reinforcement learning, it has its issues. The main problem of deep q-learning is that it tends to be overoptimistic in the sense that during training, the deep q learning model overestimates the reward given for a specific action state. [11] As the agent is trained, the issue of overestimating worsens. Since the selection and evaluation of actions are taken by taking the max across all possible action values, the chances of the deep q learning model overestimating action values increases. So, with more time, the team would like to implement a double deep q-learning network. [15] This was a modification to the deep q-learning network to allow for the loss function to be broken in to two separate weights. One of the weights is used for evaluation, the target network, while the other weights are the real time weights used, the online network.

VIII. Team Coordination

A. Final Status

Below is a table of our final status according to our original proposal. With this you can see that we completed all our goals. We had a stretch goal to apply our networks to an Atari game but due to a lack of resources, with one member dropping the course and a lack of a GPU, this was deemed not possible given the time within the semester.

Table 2: Team Milestones

Proposed Milestones	Week	Percent Done
Research Deep Q learning papers	1	100%
Determine scoring metric for model evaluation	2	100%
Proof of concept tensorFlow NN	3	100%
Develop deep Q learning NN model	4	100%
...	5	100%
Import model into OpenAI Gym	6	100%
...	7	100%
Train and Test on different games	8	100%
...	9	100%
Assess results	10	100%
Compare different model's results	11	100%
Write final paper and create demo PowerPoint	12	100%
Project Due	Final	100%

Table 1: All original milestones proposed were met

B. Team Coordination

As a team, we worked in parallel on the code and project deliverables. Lorenzo provided the initial code backbone of the project in early development while all future and further development, debugging, and visualizations were completed by James. Please reference our github project link: https://github.com/jbocinsky/Big_Data_Project. Both James and Lorenzo worked on the PowerPoints and final paper together. Princess was our other team member but did not provide much support for the project and she dropped the course.

C. Milestones, Weekly Plans, and Deliverables

Our milestones, weekly plans, and deliverables can be seen in the table used in section 6.1. Our final deliverable consists of an assessment on the three different neural network architectures for reinforcement learning across the two control OpenAI Gym games. After further development in implementing our design, we realized that the only games we could plausibly train on were the control games as other games would require a computer with a GPU which we did not have access to. After completion of this project, all of our goals were met to full completion while only the stretch goal efforts on the Atari games were not completed.

References

[1] D. Stutz, introduction to neural networks, 2014

[2] "Basic concepts for neural networks", cheshireeng.com, 2003. [online]. Available: https://www.cheshireeng.com/neuralyst/nnbg.htm. [Accessed: 03 Feb. 2018].

[3] "OpenAI Gym", *Gym.openai.com*, 2018. [Online]. Available: https://gym.openai.com/docs/. [Accessed: 04- Feb- 2018].

[4] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. (2018). Cite a Website - Cite This For Me. [online] Cs.toronto.edu. Available at: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf [Accessed 4 Feb. 2018].

[5] Juliani, A. (2018). Simple Reinforcement Learning with Tensorflow Part 0: Q-Learning with Tables and Neural Networks. [online] Medium. Available at: https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0 [Accessed 5 Feb. 2018].

[6] Hu, J. (2018). Reinforcement learning explained. [online] O'Reilly Media. Available at: https://www.oreilly.com/ideas/reinforcement-learning-explained [Accessed 4 Feb. 2018

[7] S. implementation, S. implementation and F. Shaikh, "Beginner's guide to Reinforcement Learning & its implementation in Python", *Analytics Vidhya*, 2018. [Online]. Available: https://www.analyticsvidhya.com/blog/2017/01/introduction-to-reinforcement-learning-implementation/. [Accessed: 04- Feb- 2018].

[8] Machine Learnings. (2018). Understanding AlphaGo – Machine Learnings. [online] Available at: https://machinelearnings.co/understanding-alphago-948607845bb1 [Accessed 10 Mar. 2018].

[9] Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), pp.484-489.

[10] Budhiraja, A. (2018). Learning Less to Learn Better — Dropout in (Deep) Machine learning. [online] Medium. Available at: https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5 [Accessed 11 Mar. 2018]

[11] Hasselt, V., H., G., A., & D. (2015, December 08). Deep Reinforcement Learning with Double Q-learning. Retrieved April 18, 2018, from https://arxiv.org/abs/1509.06461

[12] Hansen, L., & Salamon, P. (1990). Neural network ensembles. IEEE Transactions on Pattern Analysis and Machine Intelligence, 12(10), 993-1001. doi:10.1109/34.58871

[13] Zaknich, A. (2003). Neural networks for intelligent signal processing. River Edge, NJ: World Scientific.

[14] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. Communications of the ACM, 60(6), 84-90. doi:10.1145/3065386

[15] Hasselt, H. V. (n.d.). Double Q-learning. Multi-agent and Adaptive Computation Group Centrum Wiskunde & Informatica.

[16] Dar, E. E., & Mansour, Y. (n.d.). Learning Rates for Q-learning (Tech. No. 5). School of Computer Science Tel-Aviv University.

[17] L. Baird. Residual algorithms: Reinforcement learning with function approximation. In Machine Learning: Proceedings of the Twelfth International Conference, 1995

[18] R. S. Sutton and A. G. Barto. Introduction to reinforcement learning. MIT Press, 1998.

[19] Lecun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444. doi:10.1038/nature14539

[20] Schmidhuber, J. (n.d.). Deep Learning in Neural Networks: An Overview. The Swiss AI Lab IDSIA. Retrieved from https://arxiv.org/pdf/1404.7828.pdf.