Deep Q Learning with OpenAI

Yizhe Hu (Computer Sciences - Senior)
Feng Jiang (Biochemistry - Senior)
Brandon Klein (Mathematics - Junior)
Zijie(Jay) Wang (Statistics - Sophomore)

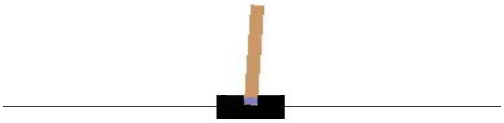
May 2017

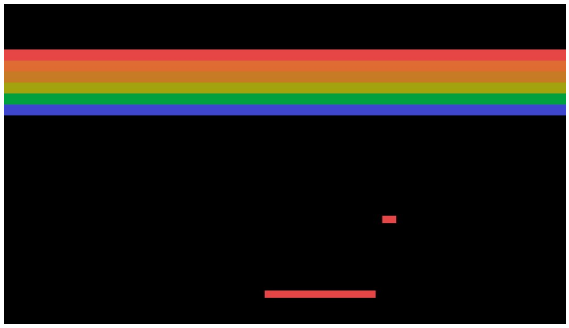
Abstract

In this paper, we aim to explore reinforcement learning by utilizing deep neural networks. The environment for this problem is OpenAI's gym, primarily focusing on the CartPole and BreakOut games. While our approach is mainly focused on learning how to use machine learning libraries such as Tensor Flow and experimenting with cloud computing frameworks such as Google Cloud, we also explore different implementations of reinforcement learning ideas.

INTRODUCTION

In our project we aim to use reinforcement learning to create a model capable of playing ‘complex’ decisions in CartPole and Breakout. These games (pictured below) rely on the agent being able to make simple instructions such as moving left and right, which then generate a new gamestate to be used in making the next decision. Both games have established rewards and end game conditions which allow them to be good environments for reinforcement learning.





The basis of reinforcement learning is to start the problem with generating either manual or randomly generated sample games that the network can use as a baseline for learning. Once this baseline is established, it is the job of the model to determine good games versus bad games and use the better games to establish its knowledge of the mechanics of what makes a game good. The step that distinguishes reinforcement learning from other learning methods, is that we then use this preliminary model from the baseline set of games to generate more games based on the model’s behavior. These examples, mixed with a variety of randomness factors to avoid memorization, allow for the model to continue learning from itself and continuously generate new data.

Our networks when implementing this technique followed closely to other projects done in class to allow more focus to be put engaging in the ideas behind reinforcement learning. For CartPole we used a series of two fully connected ReLU layers followed by a linear layer. This was due to the input from CartPole being preprocessed information on the state of the game. For Breakout this model was changed to be two convolutional layers followed by a fully connected linear activated layer due to the input being an RGB pixel array representation of the current frame. In both models we used the Adam optimizer.

Algorithm Definition

We mainly used two implementation strategies to approach this problem: traditional neural network and Q learning.

Due to the simplicity of the CartPole environment (the observation is 4 discrete values instead of the raw pixel data), we tried to treat gameplay as a classification problem -- given the four observation values, which category (left or right) should the network predict -- this setting is exactly what traditional Neural Networks are designed for. The first step of building our neural network was to generate training data (since this won't be a supervised learning). We played the game with randomly chosen actions and gather the corresponding reward. If the reward was greater than a threshold, we saved the current state (observation), the action taken, and the reward into a memory bank. We kept performing random walks until we have enough memories in the memory bank. We then proceeded to train the neural network using the generated information.

The core of our Q learning method is a "memory". It solves the issue where the traditional DNNs tend to "forget" the previously learned states when the new information is fed in. At the beginning of each frame, the algorithm makes an action by performing either exploration (take a random action from the action space) or exploitation (take the best action predicted by the neural network). We want to mostly explore early on, so the neural network is exposed to all possible actions under many different states. As the training progresses, we want to move on to exploitation more and more, so the network can actually learn. Once an action is made, we record the current state, the action taken, the reward, and the next state in the memory. The neural network then samples the memory and update accordingly.

EXPERIMENTAL EVALUATION

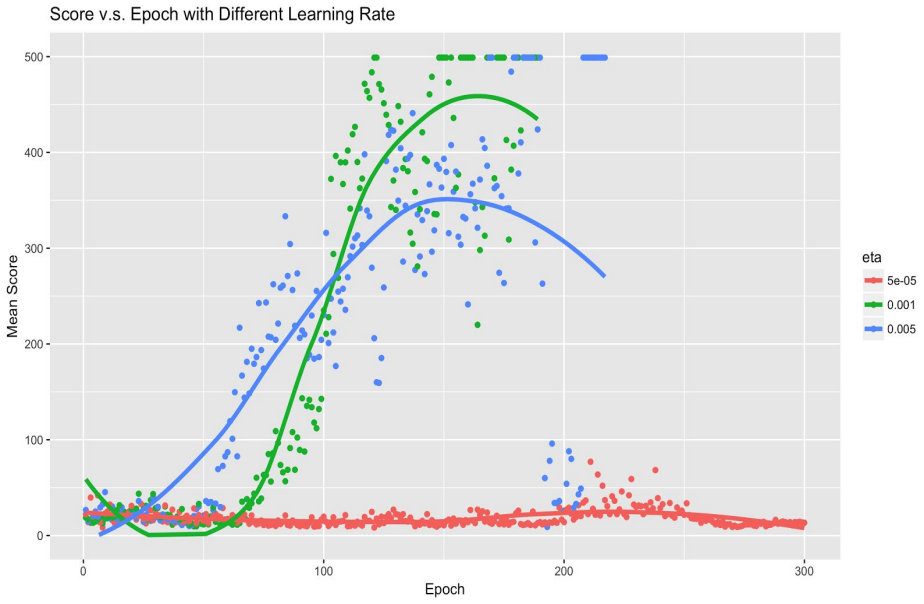
Methodology

We evaluate our methods using the average score of three independent runs over epoch (if comparing the same model), or over time (if comparing different models, or time-sensitive variables). We are interested to see how different parameters affect the performance of Q-learning, namely how learning rate, memory size, minibatch size impact on Q-learning. Our hypotheses are: 1. There should be an optimal learning rate, since large learning rate causes overshooting while small learning rate fails to effectively change the weights. 2. Smaller memory size is better, because the agent can use the newest memory to make better actions. 3. Smaller minibatch size makes the training faster while larger minibatch size converges faster. The training data are interesting because we generate our training data within the training process, the total amount of applicable training data is controlled by the memory size. For each experiment, we run the experimental group and control group for three times respectively then take average of the results, in order to decrease variance. These experiments were run on the CartPoleV1 environment, which has a maximum game score of 500, and was run on a Google Cloud VM with 8 virtual CPUs.

Results

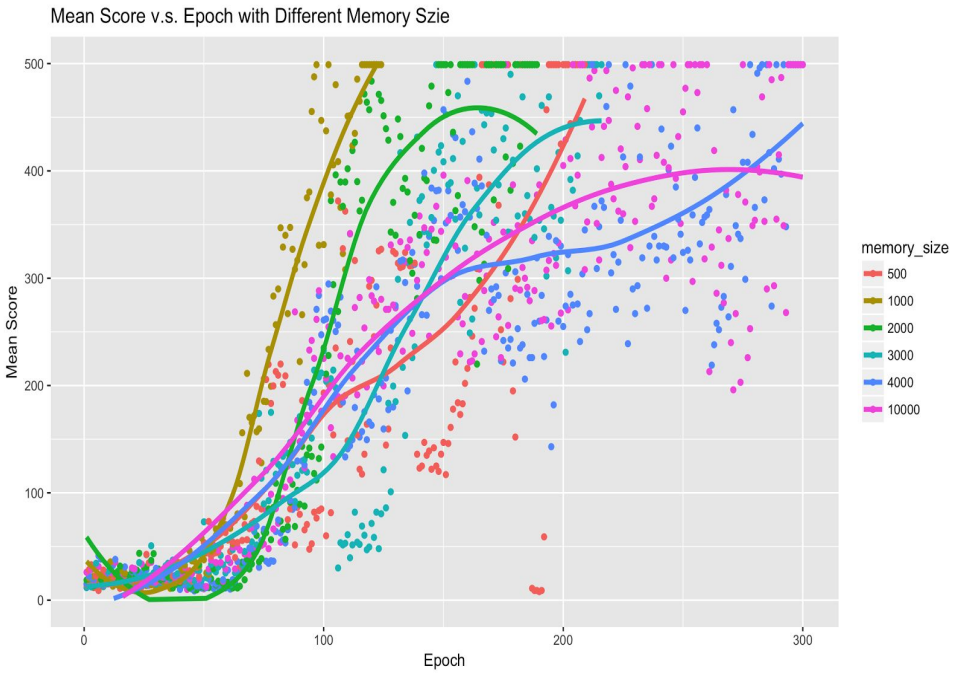
1. How learning rate influences Q-learning

We have tried to use three different learning rates to train the gaming agent. If the learning rate is too small (0.0005), the agent cannot actually learn anything. If the learning rate is too large (0.005), it takes longer time to train, since the agent may overshoot from the correct optimization path. From the plots we can see the training process has higher variance compared to other groups. Thus, it is



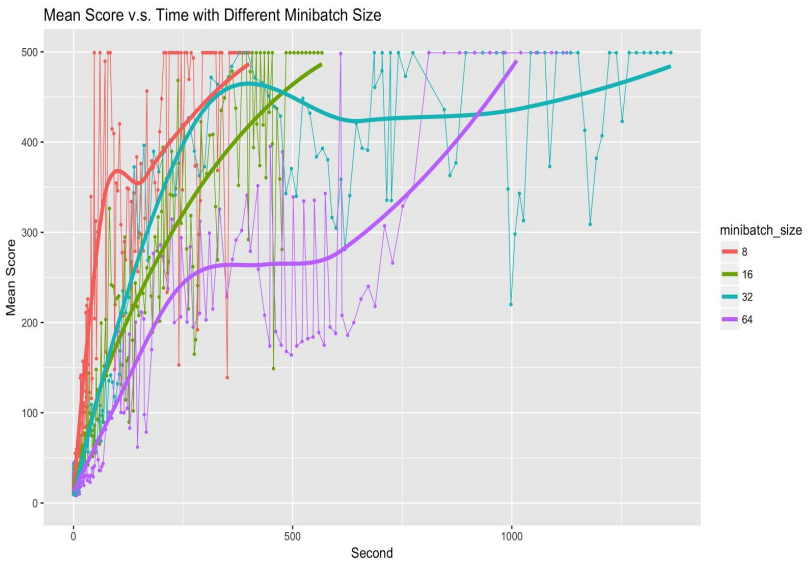
critical to use the optimal learning rate for Q-learning, slight change can cause a huge difference.

2. How memory size influences Q-learning



In the implementation, we used a queue as our memory container. While we add new memories, we also deque the old memories. Then we randomly sample memories for the "next step" learning. Therefore we are interested to know how the memory size affects the learning. From the plot we can see that the training gets unstable while increasing the memory size. We think it is due to the agent is less likely to pick the good and fresh training examples, instead it keeps picking old memories. However, using a smaller memory size (500) does not mean it can learn faster. We think it is because the agent is less likely to pick alternative "backup" memories when it realizes that it goes on a wrong path. Therefore, it worth tuning the memory size for Q-learning projects.

4. How minibatch size influences Q-learning?



We are using a minibatch approach to train the DNN. From the plot we can see that the agent learns faster while using a smaller minibatch size. When using a larger minibatch size, it takes longer for each epoch and the learning process becomes less stable. Therefore we can just stick using small

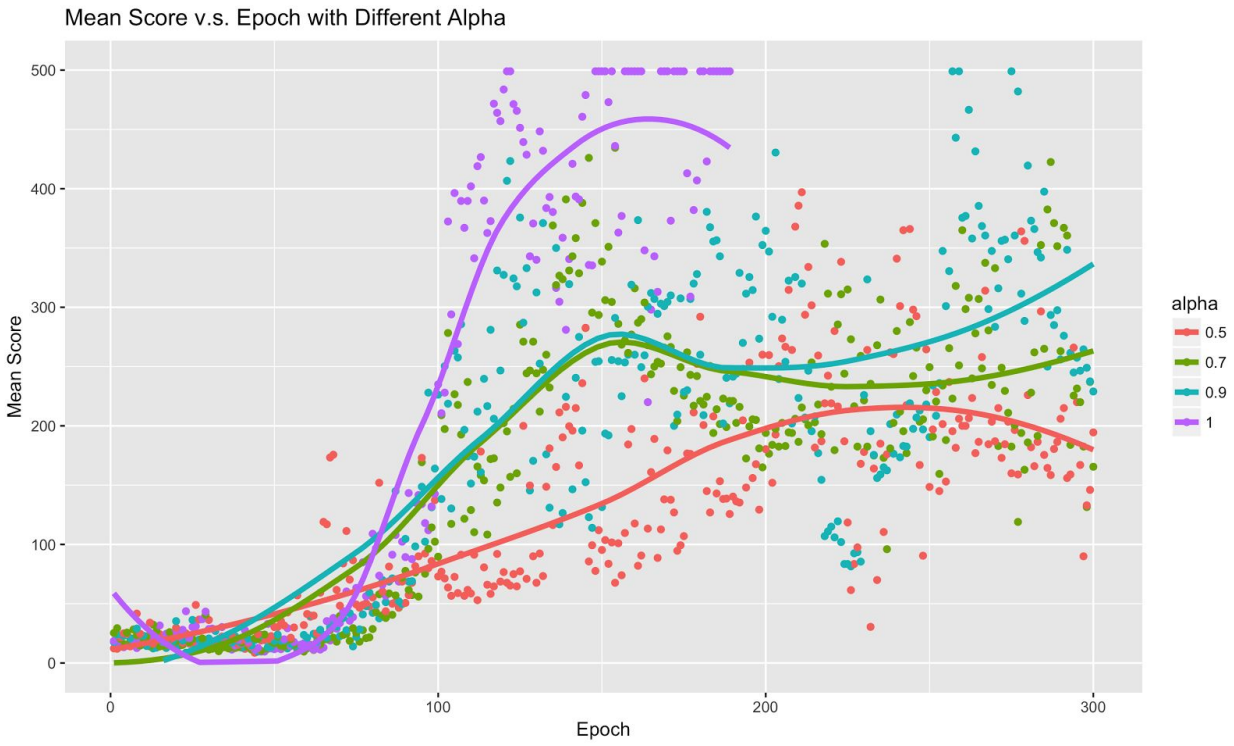
minibatch size, since large minibatch size does not really help converging.

5. Does using old Q value help?

Instead of entirely using the Q-value learned from each step, we try add the weighted Q-value. The idea is like momentum, which can avoid overshooting. Then our new updating formula is:

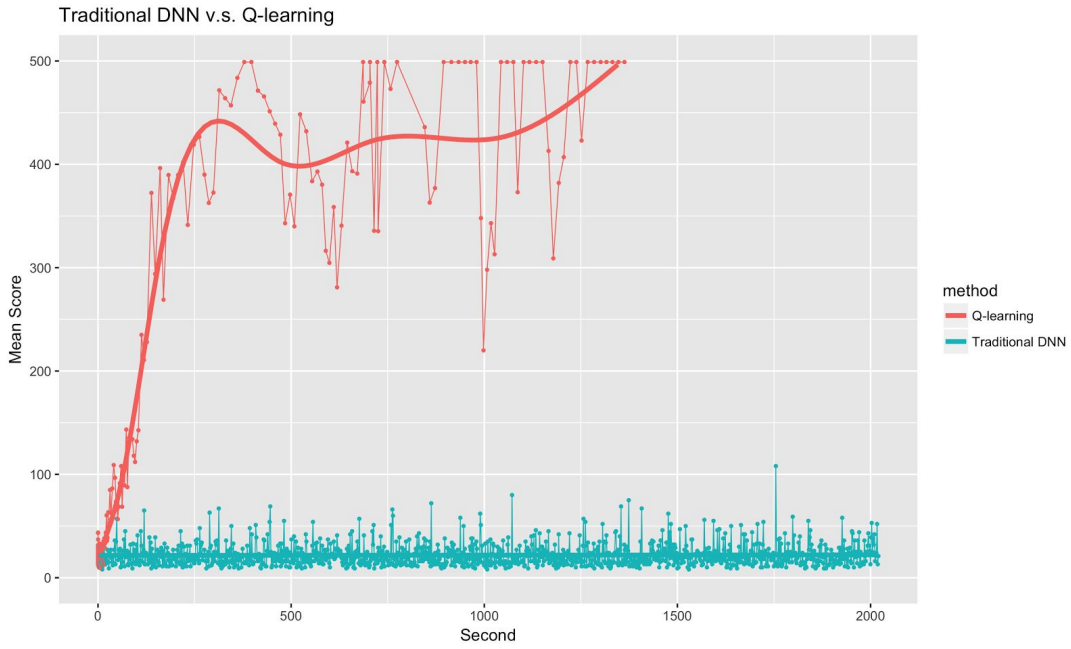
$$y = (1-\alpha)\hat{Q}(s_0, a) + \alpha\left(r + \gamma \max_{a'} \hat{Q}(s_1, a')\right)$$

Alpha is a tuning parameter, which control how much we will “trust” the new Q-value. If alpha is 1, then we don’t use the old Q-value at all. If alpha is 0, then the agent cannot learn.



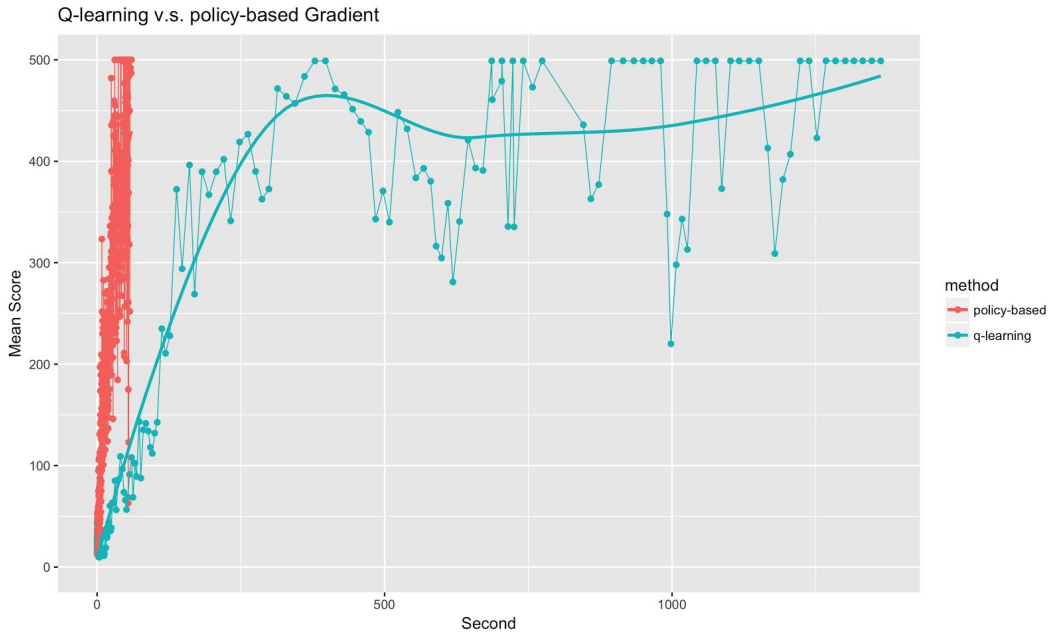
From the plot we can see adding the old Q-value is not a good idea. When counting the old Q-value, the training becomes very unstable. Even agent using alpha = 0.9 fails to learn eventually. Therefore we can see “Cartpole” is a highly deterministic game, which requires a high alpha.

6. Q-Learning v.s. Traditional DNN



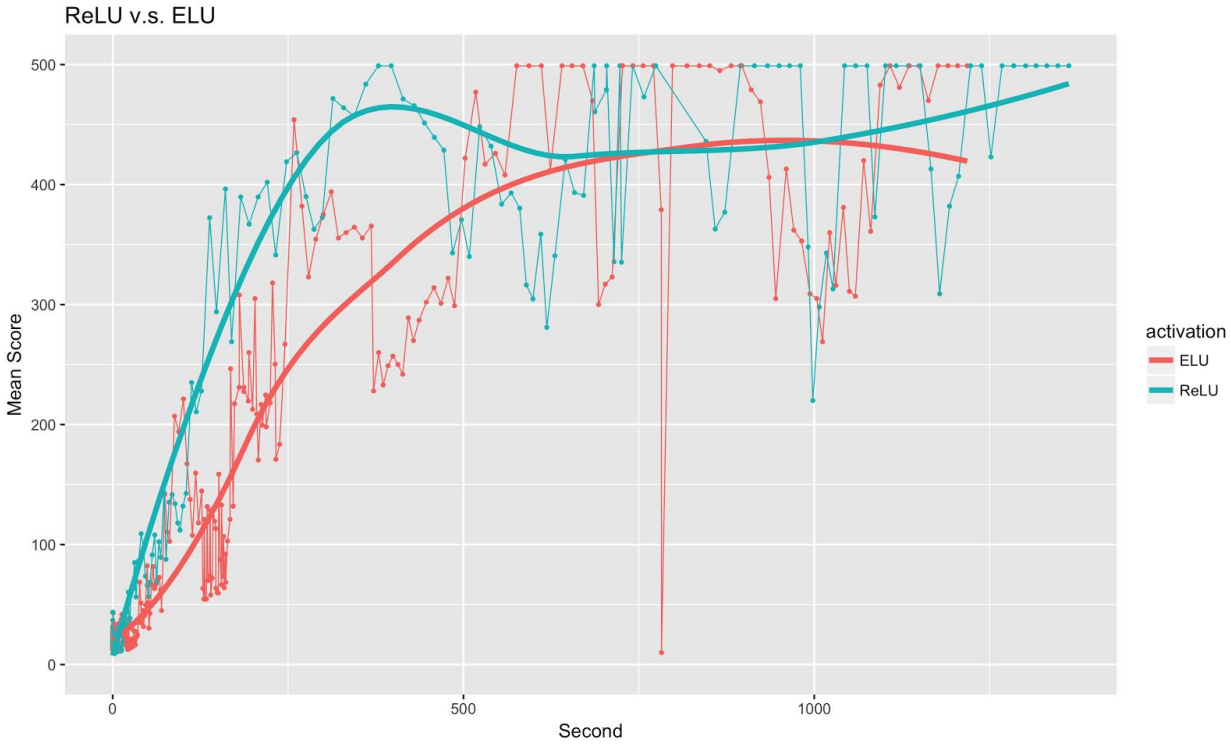
We have implemented a traditional deep neural network with 5 hidden layers to learn to play CartPole. All training examples are randomly generated for that model. By comparing two models, we can see the reinforcement learning model is much more efficient and better than the traditional DNN model.

7. Policy-Based Gradient v.s. Q-Learning



We also implemented a Monte–Carlo Policy Gradient method to play CartPole. Instead of learning the reward functions, policy-based method tries to learn the policy. From the result we can see it is much more efficient to use a policy-based method compared to use Q-learning. We think it is because CartPole has an easier policy, but a rather complicated rewarding function. From the Alpha experiment we noticed CartPole is highly deterministic, so we think it makes its policy easier to learn.

8. Can ELU Activation Function improve Q-Learning?



Other groups have tried to use exponential linear (ELU) instead of rectified linear (ReLu) as hidden layer activation function. Based on our experiment, agent using ELU does learn to play CartPole faster than using ReLU, but the difference is not significant. Also there is a much larger variance using ELU compared to ReLU.

INDIVIDUAL CONTRIBUTIONS

Our group mainly worked on the different aspects of the project collectively. The main points of emphasis Charles and Brandon worked on were setting up cloud integration of the project as well as aiding in development of the neural network used for CartPole. Feng and Jay focused on transitioning the neural network to work when presented with raw pixel data such as in Breakout, as well as fine tuning the DQN. Everyone was involved in testing and experimenting different aspects of the model.

RELATED WORK

Part of our inspiration for this project included the vast amount of projects that have sprung up in the last year, which revolve around deep reinforcement learning. One major contributor to this growing field is Google's Deepmind group. This group does a large amount of reinforcement learning, and is most known for its role in revolutionizing public knowledge of the field with its recent AlphaGo player. These ideas are then applied to both healthcare as well as aiding in Google's daily operations.

FUTURE WORK

One major shortcoming of our project came when we changed from CartPole to Breakout because this entailed transitioning our input from predetermined game details to simply a raw image of the frame. This meant transitioning from a simple DQN to one which utilized convolution neural network layers. Even with this transition we still did not yield our intended performance level for our model.

To aide in solving this issue we believe one thing that we could further experiment with is adding in LSTM methods to the final layers of the network to allow for the network to 'remember' the previous results and use those to make its decision. We feel this would be an interesting path to explore as there are some important details that can be gained such as directional movement of objects on the screen.

CONCLUSION

Overall we found the using a DQN as compared to a standard deep neural network provided sizeable improvements. We also feel we have gained good experience in using libraries such as TensorFlow and Keras. Finally we learned the value of cloud computing when using Google Cloud to help generate our models on a larger scale.

BIBLIOGRAPHY

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Ghemawat, S. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI gym. *arXiv preprint arXiv:1606.01540*.

Chollet, François. *Keras*. 2017. Print.

Mnih, Volodymyr et al. "Human-Level Control Through Deep Reinforcement Learning". *Nature* 518.7540 (2015): 529-533. Web.