Deep Q-Learning with Pong

Liam Tyler — tyler147@umn.edu

May 14th, 2019

Abstract — AI is one of the fastest growing fields, especially the topics of automation and learning. Reinforcement learning is one of the areas within AI that focuses on automating an agent's learning process by letting it explore an environment by itself. This automated learning approach has the potential to save a lot of time and effort from domain experts and programmers. Despite this, reinforcement learning has largely suffered in high-dimensional problems and often requires hand crafted state representations. In 2013, DeepMind demonstrated that this limitation can be overcome in the context of Atari games. Their approach was to use something called a Deep Q-Network. It uses only the game images and scores as inputs, and it requires no hand-crafting or knowledge of the game. The goal of this paper is to recreate DeepMind's technique for one Atari game: Pong. The results show that the agent is able to achieve super-human levels of performance on par with DeepMind's results. Since DeepMind's paper, the Deep Q-Network technique has been shown to perform well in most Atari games with no game-specific information.

1 Introduction

Two of the overarching goals of AI have always been to create agents that behave intelligently and learn. In many situations, agents have been successfully crafted to behave intelligently. However, as the problems become more complex, more effort from programmers and domain experts is needed to craft an intelligent agent. At some point, the problem becomes too complex for experts to teach an agent how to behave intelligently in all scenarios of an environment. Reinforcement learning (RL) is one of the subsets of AI that has long been a been a proposed solution to this problem. Instead of having experts spend time teaching an agent, get the agent to teach itself how to behave intelligently and react properly in new environments. This approach lets the agent explore the environment by itself and learn over time which actions it should be taking from the rewards and punishments it receives, much like humans naturally do. While it would be great if this was always successful, RL is not without its own challenges. One of the long standing challenges has been getting the agent to learn directly from high-dimensional data such as images[1], where the number of possible states an agent could observe is very large. As an example, if an image is 100x100 pixels and has standard RGB coloring, then there are $255^{100*100*3}$ possible images. This becomes extremely impractical for an agent that tries to learn which action it should take in each possible state, since there are just far too many possible states. As a result, a lot of effort needs to be put into carefully extracting the important features to come up with a more compact representation of the state, which partially defeats the goal of RL. RL is not the only field to study how to learn from images though.

Modern advances in hardware and the subsequent advances and popularity of deep learning have enabled learning from high-dimensional data such as images[2]. Specifically, neural networks with convolutional layers have had a lot of success in learning useful, compact information from images. Given this success in deep learning, this seems like the perfect solution to the high-dimensional hurdle RL has. Combining reinforcement learning with deep learning (deep reinforcement learning) is not without its own challenges though. One reason for this is that in RL, an agent can go many steps without receiving any reward, and even when it does receive one, it can be due to an action it took a long time ago. This problem of associating rewards to the actions that caused them is called the credit assignment problem[3]. This is an issue for most deep learning algorithms because they need clear examples of an association in order to learn it. Another reason deep RL can be challenging is because usually deep learning assumes that all of the examples you give it to train on are independent from each other[1]. This assumption is often broken in RL though, because time-steps are often highly-correlated. Take the game of Pong for example. The position of the paddle in one time-step is very dependent on where it was the last time-step. As a result, training on consecutive time-steps of Pong would cause many deep learning algorithms to fail.

In 2013, DeepMind overcame these challenges of combining deep learning and reinforcement learning[1]. They demonstrated this by using deep reinforcement learning to teach an AI how to play seven Atari 2600 games, achieving a human level or super-human level of play for all seven games. The deep RL technique they created is called Deep Q-Network. It is a variant of the Q-learning algorithm that uses a neural network. This means that no hand-crafted feature extraction is done from the frames, but rather the frames are used directly as input into the network, which will learn the features itself. In addition, the only things the network has access to are the image, reward, and game over signal for each frame. No game-specific information is provided, no hard-coded rules are used, and the emulator internal state is not exposed. The goal of this project was to try to recreate the results that DeepMind had just for the game of Pong, as seen in Figure 1. In order to understand Deep Q-learning however, some background in standard Q-learning is needed.

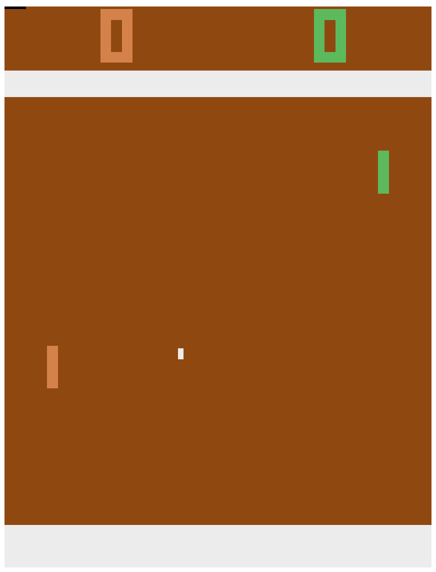


Figure 1: A schematic from the Atari 2600 version of Pong. The agent controls the green paddle on the right.

2 Background

Q-learning is one way to solve a type of problem known as a Markov Decision Process (MDP). MDPs are the standard way of modeling RL problems. Pong, and all of the Atari games are modeled as MDPs, so if we can solve an MDP with Q-learning, we can theoretically solve Pong with Q-learning. With an MDP, the world is viewed as a sequence of discrete time-steps. At each time-step t, the agent can see some state of the environment s_t. Based on this state, the agent must select an action a_t to take. Once the agent performs that action in the environment, the environment updates to the next time-step $t+1$. The agent receives a reward (or punishment) r_{t+1} and can observe the new state of the environment s_{t+1}. This process then repeats itself, as depicted in Figure 2. For Pong, the environment is the Atari Emulator. The state is the image rendered to the screen at that time-step. The actions the agent can take are moving the paddle up, down, or not at all. The reward is -1 when the agent is scored on, 1 when the agent scores, and 0 otherwise. MDPs also define two other things: transition probabilities, and the Markov property. Transition probabilities are how likely an agent is to successfully move to its intended state. For this paper however, we know that this probability is 1 for Pong, so we will omit this from the upcoming equations and discussions. The Markov property states that where an agent moves to in the next time-step only depends on where it is now. This holds true for Pong because one does not need to remember where the paddle was previously in order to play for example. This helps simplify the equations for how an agent learns to reach its goal.

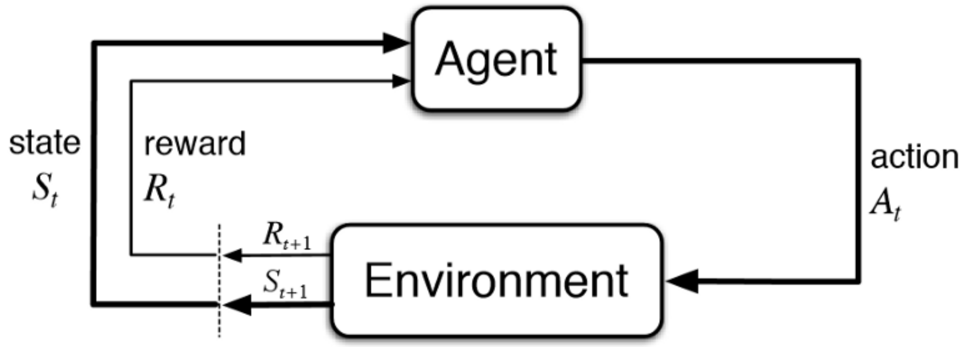


Figure 2: The repeated process of an agent acting, and observing a new reward and state [4].

The goal of an agent in an MDP is to maximize its future total reward. The standard approach is to actually use a discounted reward function. This is so that the agent prefers more immediate rewards that it is more certain about, rather than long-term rewards that are more uncertain. This is seen in Equation (1) below, where t is the time-step, and γ is the discount factor ($0 < \gamma < 1$):

$$R_t = r_{t+1} + \gamma r_{t+2} + r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{1}$$

The strategy an agent follows to select its actions is known as the policy π. This means that the goal of the agent is really to learn the policy that gives it the highest future reward. To create this policy, the agent needs some way of telling how good of a state it is in, and which action to take. This is usually done by something called the *value function*. The value function is shown in Equation (2), and it is the total reward an agent is expected to receive when starting in state s and continuing to follow its policy π. The optimal value function (3) is the expected reward when following the best strategy.

$$V^{\pi}(s) = \mathbb{E}[R_t | s_t = s] \tag{2}$$

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \tag{3}$$

This value function is the basis of the Q-learning. Standard Q-learning works by defining a function Q, which is a mapping of state-action pairs to expected total rewards of those pairs. The optimal Q function Q^* is seen in the equations below, and it is the expected total reward from taking an action a in a state s and following the optimal policy thereafter.

$$\begin{aligned} Q^*(s,a) &= V^*(s) & (4)\\ &= \max_{\pi} \mathbb{E}[R_t | s_t = s, a_t = a, \pi] & (5)\\ &= \mathbb{E}[r_t + \gamma \max_{a'} Q(s', a')] & (6) \end{aligned}$$

This way of rearranging the optimal value function is very useful to the agent because it defines which action the agent should take. The agent can simply look at the Q values for all possible actions in its current state s_t, and select the action that gives the highest Q value: $a* = \text{argmax}_a Q^*(s, a)$. The final form of Q^* in Equation (6) above is a form of the Bellman optimality equation, with a' and s' being the next action and state respectively. This optimality equation is the core of most RL algorithms because it defines an iterative update step:

$$Q_{i+1} = \mathbb{E}[r + \gamma \max_{a'} Q_i(s', a')] \tag{7}$$

This converges to Q^* as $i \to \infty$ in a process known as *value-iteration*. Value-iteration cannot be directly applied to the problem of Pong, but it still is at the core of the approach used by DeepMind, as well as for many related problems.

3 Related Work

To know why DeepMind used a Deep Q-Network, it is important to know what some of the previous attempts at this problem were. Even without machine learning, reinforcement learning has been around for a long time and there are many techniques that have been applied to solve MDPs similar to Pong. One of the first approaches in the late 1950s and early 1960s was to just apply value-iteration or policy-iteration (very similar to value-iteration) with the value function $V(s)$[5]. While both of these methods can converge and work for many problems, they both have two main issues that prevent them from being directly applied to Pong. The first is that every iteration, every action-state pair must be updated, which makes it not suitable for high-dimensional problems like Pong with image based states. This inability to handle large state spaces is true for all methods of RL that store their functions as a table, such as standard TD-learning, Q-learning, and SARSA. Another issue is that in most RL environments, the rewards and transition probabilities are not known before hand. If only this technique is used, the agent would need to explore every action-state pair before doing value-iteration, which again is infeasible. Due to these limitations, a lot of effort was towards value function approximation. This again worked for some problems, but early approaches often had discontinuities in these functions that have been a key issue in getting convergence even in simple environments[6]. It became clear that the value function alone wouldn't be enough for problems like Pong.

A method that had some success in learning video games was TD-gammon[7]. TD-gammon was a backgammon AI that learned to play at the level of world-class players only through self-play. It used temporal-difference learning, which keeps track of the rewards for each state, and whenever the same state is seen again, it will update its estimate for the reward. This is very similar to Q-learning, and the paper also modified TD-learning to use a neural network to map the states to their expected rewards, just like the approach in this project. Despite their similarities, TD-gammon did not have much success beyond the game of backgammon. In the end it was believed that this was a special case that only worked due to the nature of backgammon[8].

Another related approach was to not use the value function at all, and instead try to estimate the policy function directly using a neural network. This is known as the policy gradient method. This is setup like a typical deep learning problem by defining the gradient of the policy, and the using it to run an optimization algorithm on it. The main problem is calculating the gradient. In 2000 however, it was shown that an agent's past experiences could be used in combination with an advantage function to get a good enough estimate of the gradient to still prove convergence to a locally optimal policy[6]. Since then, this method has received attention in problems of locomotion. It was successfully used the context of automated searching of parameters for a walking robot[9]. One of the highlights of this method is that it easily handles continuous state and actions spaces. This is an advantage over Q-learning. The paper pointed out though that the starting point for the parameter search had a large impact on which local optima it would find, or if it would converge at all. Policy gradient methods in general are heavily impacted by the starting states and the large variance when estimating the gradient. Methods to reduce the variance have been published, but none were applied to the problem of Atari games until after DeepMind's paper.

As DeepMind points out in their paper, probably the closest algorithm to theirs before it came out was neural fitted Q-learning[10]. This paper was tryiing to use a multilayered perceptron network to tackle the issues that often came when doing value function approximation where convergence takes a long time, if ever, to occur. This is largely due to the fact that each update has the chance to overwrite all of the past knowledge and cause the network to forget. They take the approach of saving past experiences in memory, which is exactly what DeepMind does. The main flaw with this paper was that it does not scale up very well for high-dimensional datasets. It has been applied successfully to images, but only after the images go through an auto-encoder network to once again learn a more compact representation of the space. The gradient descent update in DeepMind's paper does not have this scaling problem, and is able to directly take the images and output Q values.

It is worth noting that since this paper came out there have been several improvements upon the technique. One of the first ones was an improvement on how to best sample past experiences, called Prioritized Experience Replay[11]. The idea behind this technique is that an agent can learn more from some experiences than others, and those should be sampled more often in order to learn more efficiently. DeepMind's approach was to sample uniformly, but this approach samples experiences based on how large the difference between the network's prediction and the actual experience is. This technique was shown to out perform DeepMind's approach in 41 of the 49 Atari games. Another improvement is to use a separate target network when predicting the loss values of the actions taken[12]. With the standard DQN approach, the same network is used for estimating the Q values and for generating the target Q values to optimize to. This is analogous to a dog trying to catch it's own tail. In some situations this can cause the network to destabilize and prevent convergence. Fixing the target value network and only updating it every once in a while has been shown to improve stability.

Perhaps the simplest modification to the original DQN with large success is the Double Deep Q Network (DDQN)[13]. The original DQN has a problem of overestimating Q values because it uses the maximum action value, as seen in Equation (6). In some situations, this can overestimate how good a suboptimal policy is, and prevent the agent from learning a better policy. Instead of using the max action, DDQN selects the action using the primary network and then uses the target network to estimate the Q value for that action. With this decoupling, the overestimation is reduced and the training becomes more stable. This was shown to have much better performance on several Atari games, and equal performance on most others.

One recent technique that made an improvement upon the architecture is the Dueling-DQN[14]. The dueling-DQN splits up the last hidden layer of the network into two components: the typical value function $V(s)$ and the advantage function $A(s, a)$, which describes how much better certain actions are compared to others in a state. These two components are then combined back together in the final layer of the network to compute the Q value. The benefit of splitting the Q value computation into these components is that it allows the agent to better learn the value of a state without needing to learn the effect of each action in each state[14]. For example in Pong, if the agent has just hit the ball back, then its next action does not matter. This architecture allows the agent to better learn this compared to a regular DQN. Since it still outputs a normal Q value, it can be used in addition to other improvements such as the DDQN and prioritized experience replay mentioned earlier. This has caused it to be be one of the best approaches for Atari games to date.

4 Approach

Standard Q-learning performs the Bellman update on a table of Q values, with one cell for every possible state-action pair. Since Pong has 3 possible actions and 210x160 colored images, there would be $3 * 255^{210*160*3}$ entries in this table. From both a memory and computational perspective, this is completely infeasible without simplifying the state representation. Since the goal is to use the raw images as state representations, a new approach needs to be used for representing the Q function.

4.1 Deep Q-learning

The approach taken by DeepMind to represent the Q function was to use a neural network. With neural networks' ability to estimate functions[7] combined with their ability to extract high-level features, one should be able to estimate Q^* well. If the network can estimate Q^* well enough, then the agent should be able to perform well in the game. To train the network, batches of past time-steps can be sampled. For each past time-step, $s_t, a_t, r_t, and s_{t+1}$ are known. This means that the Bellman optimality equation can be used to calculate the target value that s_t, a_t should have resulted in. The algorithm for this process is called *experience replay*, and it is the main contribution from the DeepMind paper. The psuedocode is shown in listing (1) below:

Listing 1: Experience replay psuedocode

```
batch = a list of multiple past experiences ($s_t$,$a_t$,$r_t$,$s_{t+1}$,$gameover_t$)
inputs, outputs = []
for i = 1 to len(batch):
    state, action, reward, newState, gameover = batch[i]
    target = reward
    if not gameover:
        target = reward + $\gamma$ * max(network.predict($s_{t+1}$))

    currentMappings = network.predict($s_t$)
    # approximately map the current Q estimates to the updated estimate
    currentMappings[action] = target

    inputs.append(state)
    outputs.append(currentMappings)

network.train(inputs, outputs)
```

This algorithm is the key insight from DeepMind. The key idea is that the agent's experiences need to be saved and then sampled from in order to train the network. The memory that the experiences are saved into is called *replay memory*. By randomly sampling from these past experiences, there will be very few consecutive frames in the batches. This solves the issue of the neural network assuming that each sample it is given is independent. In practice, the replay memory only stores the last N frames due to memory constraints. Earlier, it was also mentioned that an agent's current state s_t is the current image on the screen. In reality, a single frame is aliased and not enough to infer anything besides positions. In Pong, the agent would have no idea if the ball is moving above them, below them, or away from them, even though the agent should react differently in each of those scenarios. In order to infer more information such as velocities and accelerations, the s_t seen in the pseudocode is actually the last 4 frames leading up to that time-step, $[s_{t-3}, s_{t-2}, s_{t-1}, s_t]$, and the new state $s_{t+1} = [s_{t-2}, s_{t-1}, s_t, s_{t+1}]$.

When an agent needs to select an action, there is one issue. At the beginning of training, the Q^* has not actually been learned yet. This means that if the agent uses this to select its action, it is not going to lead to a high future reward. As a result, the agent first needs to explore the environment randomly to get new experiences to learn from, until Q^* has been well estimated. When to stop exploring randomly and start using the network's prediction is a common problem in RL called exploration vs exploitation. One common solution which is used here is the ϵ-greedy strategy. With this strategy, the agent generates a random probability r, and explores randomly if $r < \epsilon$. If $r > \epsilon$, it uses the network's greedy selection. Every time an action is chosen, ϵ is then linearly decreased, eventually to some minimum amount. This way the agent slowly transitions to mostly using the network's selection as it is trained more. The full algorithm for training the agent can be seen in the psuedocode in listing (2) below:

Listing 2: Full training algorithm

```
Fill up the replay memory with a random agent's experiences
Initialize the weights of the network with the HE initialization
for step = 1 to M:
    Restart the game and record the first state $s$ and state stack [$s$,$s$,$s$,$s$]
    while not gameover:
        Select action $a_t$ randomly with probability $\epsilon$, otherwise select $a_t = \max_a Q^*(stack, a)$
        Execute the action in the game, and get the reward $r_t$ and new state $s_{t+1}$
        Store experience ($s_t$,$a_t$,$r_t$,$s_{t+1}$, gameover) into replay memory
        Perform experience replay on one batch to train the network
        if step % 1000 == 0:
            Update the target model's weights to the primary model's weights
```

The psuedocode above reveals two differences in the approach I take and the approach taken by DeepMind. The main difference is that DeepMind did not use a target network. As mentioned in the related work, the use of a target network is to help prevent the Q function converge. An extensive comparison between these two approaches is not done in this paper, but it should help the network converge in more situations than DeepMind's would. The second difference is how the weights are initialized. In general, neural network weights are often initialized in two ways. The first is randomly sampling from a uniform distribution, which is what DeepMind did. The second is randomly sampling from a Gaussian distribution, which is known as Xavier initialization. In 2015 a paper was written on *HE-initialization*, and the results suggested that when using ReLU activation functions like in this project, more complex networks can be trained than with random or Xavier initialization[15]. As a result, this is the initialization used in this project. While this may help, comparing the exact effects of this initialization in Pong is beyond the scope of this project.

DeepMind preprocessed the frames from 210x160 RGB to 84x84 grayscale. Originally this project tried to avoid the cropping done to be more general, but due to a memory leak bug in the neural network library being used, the final images in this project were cropped and downsampled to 80x80 grayscale images. The neural network architecture itself is the same as the DQN in DeepMind's paper. The input is a stack of (4x80x80) images. This feeds into a convolutional layer of 16 8x8 filters with a stride of 4. The second convolutional layer has 32 4x4 filters with a stride of 2. This feeds into a fully connected layer with 256 units, and then to the fully connected output layer to the number of possible actions. While Pong really only has 3 useful actions, the emulator for that game actually uses 6 possible actions with the fire button, even though it does nothing in the game of Pong. Despite this, having a larger action space makes it harder for the network to converge. With this architecture, the results were gathered and are presented below.

5 Results

Results were gathered by letting an agent train on 2,000 games of Pong. This was about 3 million frames, with the primary network being trained every step, and the target network being updated every 1000 frames. In order to speed up the training time, a common technique called frame-skipping was used[16]. With this technique, the agent only sees and trains on every fourth frame. Whatever action the agent selects for that time-step will be executed four times before the agent sees the next frame. This means that about 12 million frames were actually played. To avoid confusion, the rest of the paper will talk about the number of frames the agent saw with frame skipping (3 million), not the number of frames it played (12 million). The ϵ for the ϵ-greedy strategy was linearly annealed over the first one million frames until it reached a minimum value of 0.1, and then it stayed there for the rest of the training. The rest of the hyper-parameters were related to how the networks were set up and trained.

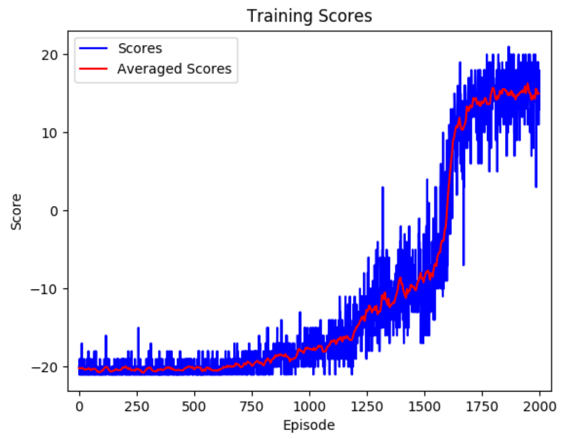
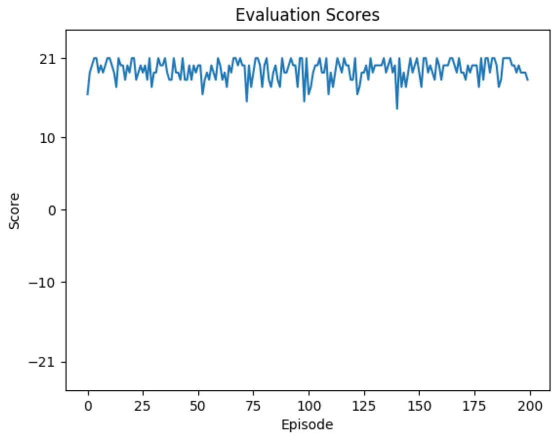
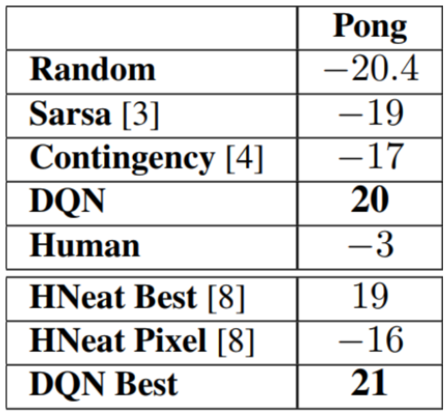


Figure 3: A plot of the scores and averaged scores during the training.

Both networks used the Adam optimizer with a learning rate of 0.00001 instead of DeepMind's RMSProp optimizer. The Huber loss function[17] was used instead of Mean Squared Error. The network was trained with a batch size of 32, with one epoch per experience replay. The replay memory itself only held 500,000 of the most recent frames, and it was sampled uniformly. The training results are shown in Figure 3.

The results look similar to a logistic function. For the first 1,000 games roughly, there is only a slight increase in the average score. Shortly after 1,000 games are trained, the score starts to improve rapidly. This is also the point when the agent reaches its minimum ϵ value of 0.1. From game 1,500 to 1,750 the average score goes from about -10 to 10, and the agent always wins after that point. These were the results during the training phase. After 2,000 games were played, there was an evaluation phase, whose results are shown in the left image of Figure 4.





	Pong
Random	-20.4
Sarsa [3]	-19
Contingency [4]	-17
DQN	**20**
Human	-3
HNeat Best [8]	19
HNeat Pixel [8]	-16
DQN Best	**21**

Figure 4: On the left is a plot showing the scores during the evaluation phase. The right is showing a table of Pong evaluation scores of different agents from DeepMind's paper.

The evaluation phase was done by setting the agent's ϵ to 0.05 because this is how DeepMind evaluated their results. 200 games were then played, and the score was recorded for each game. The plot shows that the scores ranged from 14 to 21, and the average value of all of them was **19.36**. This is slightly lower than DeepMind's score average score of 20, and higher than any other evaluated algorithm as seen in the right image of Figure 4. The average score when using $\epsilon = 0$ is **21.0**. One additional observation is that while watching a video of the agent play during the evaluation games, it seems to always perform the same actions and strategy. After it hits the ball the first time, it then seems to always go to the bottom of the screen and try to hit the ball the same way that either scores, or bounces it back to the bottom where it already is. The AI does not react fast enough to this, and only scores a point when the agent selects a random action that deviates it from the network's selection. How these results compare to DeepMind's and an analysis on them can be found in the next section.

6 Discussion

6.1 Comparison to DeepMind

The first comparison between this implementation and DeepMind's is to look at the average evaluation score. This implementation was less than a point lower than what DeepMind achieved. The most likely reason for this is because DeepMind trained all of the agents for 10 million frames, which is over three times as long as what this implementation had due to time constraints. Another consideration is that this implementation had a replay buffer size of 500,000 due to memory limitations of the training computer, while DeepMind's was 1,000,000. If these two things are taken into account, then it is likely that the performance of both algorithms would be even closer with more training and a higher memory size. Given how close the scores are and these two differences, it is safe to say that the results achieved in this implementation are on par with DeepMind's. There also were a few differences between DeepMind's implementation and this paper's, which could have an effect on the scores.

It is possible that using the target network would improve performance over an implementation without it, but it is unlikely. The use of a target model is to help convergence in an unstable training process. The original DQN already converged to an optimal solution in Pong without the help of a target network however, so there probably would not be any visible difference for this problem. This would be more visible for a harder Atari game like Breakout, where the original DQN's performance is far from optimal. This also applies to the use of Huber loss and HE initialization. Huber loss helps prevent the optimizer from changing the weights a lot in one batch that has unexpected rewards. In these situations Mean Squared Error can destabilize the whole network and prevent convergence. As with the target model, it is hard to compare this when both solutions seem to have converged to an optimal solution. One interesting note is that DeepMind actually clips the rewards to be within -1 to 1. With Huber loss this might be unnecessary, since the effect of the reward is already reduced compared to using Mean Squared Error as the loss function. Finally with HE initialization, this has been shown to help train deeper and more complex networks. A relatively simple network was used in this problem, so HE was more of a precaution than necessity, and there would likely be no performance difference unless the architecture was changed, or maybe if a harder game was tested.

6.2 Potential Limitations

One potentially concerning thing about the results is the single-minded pattern that was observed when watching the agent during evaluation. From a visual analysis of the agent's movement, it seemed to know exactly one way to score, and that it could not do well if it was not exactly in its pattern. The evaluation difference between $\epsilon = 0.0$ and $\epsilon = 0.05$ is 1.64. While this does not seem large, it is important to remember how small an epsilon of 0.05 already is. It means that on average the agent will have 1 in 20 actions be selected randomly for it, for only one time-step. If that alone is enough to cause a 1.64 difference, then there is a chance that this learning is not very robust. This would imply that we see much worse performance in harder Atari games such as Space Invaders or Seaquest. This theory is actually supported by the original DQN paper as well. Pong and Breakout (the two games with a moving ball and paddle) were by far the best performance, and most of the more complex games were below human level performance. More testing would have to be done and to try this on other games, but it seems possible that this DQN implementation has yet to find globally intelligent strategies even for simple Atari games.

7 Conclusion

This paper describes implementation of DeepMind's Deep Q-Network technique for the Atari game Pong. It explains what some of the challenges are of reinforcement learning in high-dimensional spaces, as well as some of the previous attempts to solve this problem in the field. It describes how the Deep Q-Network technique builds off of standard RL techniques and combines deep learning into it. The results show that this agent performs about as well as DeepMind's does. They also show that it has potential to learn other Atari games as well, but would likely perform not as well in more complex games.

In the future this agent would be tested on other games that are more difficult than Pong. Since the results show that the agent possibly found a single lucky strategy, it would be something to watch for in the other games. Since the DQN paper came out, there have also been many improvements upon the technique, such as prioritized experience replay and dueling double DQNs. I would like to use these techniques on all of the Atari games and compare to the more recent paper results.

8 Code

Code used in this project can be found in the GitHub Repository at `https://github.com/LiamTyler/DeepRLAtariGames`.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[2] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015.

[3] R. S. Sutton, "Temporal credit assignment in reinforcement learning.," 1985.

[4] Greg Surma, "Cartpole - introduction to reinforcement learning (dqn - deep q-learning)." `https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288`, 2018. [Online; accessed 11-May-2019].

[5] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1st ed., 1998.

[6] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, pp. 1057–1063, 2000.

[7] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[8] J. B. Pollack and A. D. Blair, "Why did td-gammon work?," in *Advances in Neural Information Processing Systems*, pp. 10–16, 1997.

[9] N. Kohl and P. Stone, "Policy gradient reinforcement learning for fast quadrupedal locomotion," in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, vol. 3, pp. 2619–2624, IEEE, 2004.

[10] M. Riedmiller, "Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method," in *European Conference on Machine Learning*, pp. 317–328, Springer, 2005.

[11] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[12] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[13] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015.

[14] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015.

[16] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, *et al.*, "Massively parallel methods for deep reinforcement learning," *arXiv preprint arXiv:1507.04296*, 2015.

[17] Y.-G. Wang, X. Lin, M. Zhu, and Z. Bai, "Robust estimation using the huber function with a data-dependent tuning constant," *Journal of Computational and Graphical Statistics*, vol. 16, no. 2, pp. 468–481, 2007.