

Using Deep Reinforcement Learning To Approach Artificial General Intelligence

Austin Derrow-Pinion

Kice Sanders

Adam Gastineau

{derrowap, sanderkd, gastinad} @ rose-hulman.edu

Rose-Hulman Institute of Technology
MA 490 - Deep Learning: Final Project Report

Executive Summary

This study provides an in-depth look into the latest reinforcement learning techniques and their applications. With DeepMind's 2013 Atari paper serving as a basis, a basic reinforcement network was created to test various methods and optimizations. From there, this network was adapted to three different environments to test its performance and to determine further areas of study.

1 Introduction

The world produces around 2.5 quintillion bytes of data every day. A very small portion of this data is labeled, in part due to the huge cost inherent in labeling data. A lot of research has been done to apply deep learning to utilize this data. Deep reinforced learning is one such method that utilizes unlabeled data to generalize learning about sequential decisions in many different environments.

In this project, we have outlined some previous work done in this field and then some initial steps we took to begin investigating the topic for ourselves. We then fully explain the algorithm and architecture that amazingly learns how to interact with environments by only receiving image pixels as input. The codebase to find our implementation and work is available at:

<https://github.com/derrowap/Deep-Reinforcement-Learning-Project>. Implementation was done using Python 3, TensorFlow, OpenAI Gym, and running on the Gauss server.

2 Background and Related Work

Below is a state-of-the-art example of related work followed by some background work we did building up to our main project.

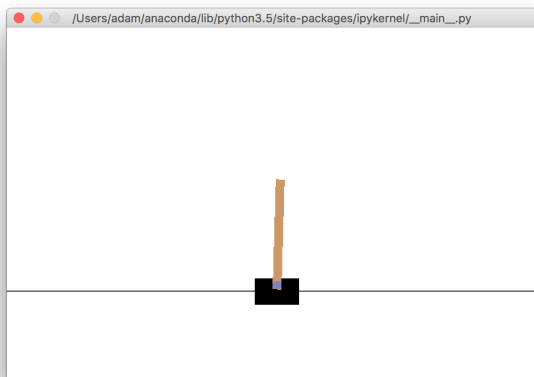
2.1 Google DeepMind's Atari 2600 Deep Q-Network

In late 2013, DeepMind surprised the world with their results on training neural networks to play Atari 2600 games. DeepMind's architecture was designed around the concept of a Q-Network, a network designed around the Q function, which determines the value of a certain action given a system state (A much more in-depth explanation of Q-Networks can be seen in *Section 3: Deep Reinforcement Learning*). DeepMind expanded upon Q-Networks, developing the Deep Q-Network algorithm, an algorithm that approximates Q using a deep neural network. This network was trained "end-to-end", meaning the network took in raw pixel data and directly output the controller inputs to the game, without any additional intervention. This allowed for the network to more closely approximate an artificial intelligence; it was required to take in raw visual input in order to play the game, just like humans do.

As shown in their paper, the Deep Q-Network (hereafter abbreviated DQN) performed admirably in interpreting the interaction models of particular Atari games. In six of the seven tested games, the DQN performed significantly better than an "expert" human player [3]. These results completely dwarf all similar attempts at playing the same titles using more traditional machine learning methods. Although the results were quite promising, the DQN was trained individually and separately on each game; the network did not generalize the interaction model across multiple games.

2.2 Q-Network Policy Agent for the Cart-Pole Task

On the backs of these giants, we entered the world of reinforcement networks with a simple policy based network as a proof of concept. Using OpenAI Gym as our main development framework, we began our experimentation with a scenario deemed "The Cart Pole" problem.



The Cart Pole problem is a very simple physics simulation consisting of a "cart", with a "pole" balancing upright on it. The goal of this scenario is to keep the pole balanced and upright for as long as possible. This acts as a very good starting reinforcement learning problem, as there are only three environment states (stable, pole falling left, and pole falling right) and two input states (move the cart left or right).

Our policy network is based off of very simple principles. A short experiment called an episode is performed many times, in this case being our physics simulation. Each time an action is taken within an episode, a "reward" value is generated, based on how the action performed. If the pole balanced for a long time, a high reward value would be provided. If the pole quickly fell over, a low reward value would be provided. This process, repeated many times, provides the data for which to train the neural network.

In order to more thoroughly train the model, reward values are discounted and back propagated, to determine what actions led to what outcomes (which is explained in more detail in section 3 of this paper). In this way, the network learns what decisions were favorable, and which should never be repeated again. Our original network was trained using the provided features (the position and velocity of the cart, and the angle and velocity of

the pole) and provided left and right input to the physics simulation. Without much fine-tuning to the network, it usually “solves” the game based on criteria from OpenAI Gym (200 actions in a row without failing) in under 5000 episodes which takes under 90 seconds in all to run on Gauss. The produced reward gets exponentially larger as the network continues to train.

3 Deep Reinforcement Learning

The concept of reinforcement learning can first be introduced by associating it with a Markov decision process. A Markov decision process is simply a state machine that jumps from state to state based on the observations an agent makes. For example, a state in the Cart Pole problem can be represented as the x-position of the cart from the center of the game, the speed in which the cart is traveling, and the angle at which the pole is tilted. The Markov decision process model then outputs what the action should be and the sequence of decision problems continue until the episode finishes.

3.1 Summary of Q-Learning

Q-learning is a method in which agents can learn how to optimize its current and future reward. Similarly to a Markov decision process, Q-learning first observes the current state of the agent in an environment. That observation then leads to a decision of what action to take. This action is determined by which one is associated with the greatest Q-value. A Q-value can be thought of as the quality of an action, or how much evidence a specific action provides, to result in a future state of greatest reward. The greater the future reward, the greater the evidence. An entire episode's rewards can be defined as all the rewards resulting from n actions where a_n is the final action which resulted in the episode to terminate:

$$R = r_1 + r_2 + \dots + r_n$$

Now when an agent in the environment is at time t , the future reward we want to optimize can be defined as:

$$R_t = r_t + r_{t+1} + \dots + r_n$$

For a sequence of decisions, we can never be 100% certain whether it will lead us to the optimal future reward. Some random events may happen that we didn't expect and therefore must act accordingly. Because of this, Q-learning takes advantage of discounted future reward. Discounted future reward is simply defining the optimal action to more strongly consider the rewards we can gain in a short amount of time rather than in a long amount of time. The future reward is now predicted by multiplying each reward by a discount rate as such:

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \dots)) = r_t + \gamma R_{t+1}$$

The discount rate value will always be between 0 and 1. If the value is 1, there is no effective discount on the reward and therefore consider future reward as strongly as we consider the possible reward for the next action. When the discount gets closer to 0, we consider future reward less and less. Each Q-value can be defined as the following equation:

$$Q(s_t, a_t) = \max R_{t+1}$$

This equation for Q-value tells us to take the action which is thought to maximize the discounted reward and continue taking action from the resulting state.

If we knew for certain all the states and resulting rewards for each action, this would be a closed problem with a certain sequence of actions. The problem with this is that few environments include a small set of states we can store and calculate. Rather, most environments have an indefinite number of states that we can't possibly keep track of. Before going into how this problem is solved, consider the case where there is a finite number of states to consider. Q-learning can be represented as a simple table lookup. A table is initialized with random values as the Q-values. We then step through the environment, taking predicted actions in a given state and observing the resulting state and reward. To then learn the Q-values to make better decisions in the future, we calculate the updated Q-value as such:

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

Where α is the learning rate of the model, s' and a' are the resulting state and action after taking action a on state s . If the learning rate is equal to 1, then the updated Q-value completely forgets about the previous Q-value and will simply update to the sum of the observed reward, r , and the predicted maximum future reward.

3.2 Deep Q-Networks

Now we need to address how to solve the problem with having an indefinite number of states to keep track of. For most problems, the table of Q-values would be too large to learn anything from because it would be unlikely to ever reach the same exact state visited previously. Luckily, deep learning neural networks are very good at learning difficult tasks. Instead of calculating all of these Q-values ourselves, the neural network will predict them for us. The input to the network will simply be the images of the observed states. Then the output will be the predicted Q-values for each possible action in our environment. The action we then take will be the max predicted Q-value from the output of the network.

Now that we have established how these Q-values will be calculated, how do we train the network to output predictive Q-values that actually result in optimal future rewards? The loss function, L , can be defined as such:

$$\begin{aligned} \text{target} &= r + \max_{a'} Q(s', a'), \quad \text{prediction} = Q(s, a) \\ L &= \frac{1}{2}(\text{target} - \text{prediction})^2 \end{aligned}$$

We can represent a single, trainable transition, or experience, as the tuple $\langle \text{state}, \text{action}, \text{reward}, \text{state}' \rangle$ where we have an initial state, then take an action, and observe the resulting reward and new state. The overall algorithm can be broken up into the following steps:

1. Get predicted Q-values for the current state s .
2. Calculate the target Q-value by finding the maximum predicted Q-value for the state s' .

3. Backpropagate the network to make the Q-value for the action we took on state s to be more similar to our target Q-value calculated from s' .

This is enough for a deep Q-network to start training on some environments. Note that if the model is trained and updated one at a time after every new state the environment enters, computations will be slow and hard to train. Also, it will tend to train on the beginning of the environment a lot more than the rest of the states since it will repeatedly be entering the beginning. This can cause problems with the network trying to converge to stable Q-value predictions.

3.3 Experience Replay

An experience replay is a buffer to store many transition experiences. For example, an agent will interact with an environment for 10,000 steps and with each step, it stores a new experience. If the buffer becomes full, we remove the oldest experiences. Every so often, we want to grab a batch of random experiences in the buffer and train the network on them. By doing this, the network is being exposed to a much more evenly distributed experience and is able to converge predicted Q-values much more quickly. Also, training on batches larger than one experience will be more efficient and result in the network training quicker. The batch size is a hyper parameter than can be tuned for different purposes.

3.4 Exploration and Randomness Within the Deep Q-Network

When a Deep Q-Network first initializes, the predicted Q-values are completely random. This means that the beginning of the training will consist of the network exploring possible options for optimizing the future reward. The problem with this is that as soon as it finds any method that provides some form of reward, it will drive itself in that direction and will likely become stuck at a local minimum. Ideally, we would want the network to continue exploring after it has found its initial rewarding function. This is so that it has the chance to find an even more rewarding method with possibly less steps.

To solve this problem, we must apply some randomness to the network itself. With probability ϵ , we ignore what the network predicts is the best action and simply take a random action in the environment. As the network learns, we want to take less and less random actions so that the network's learned methods can be carried out. The best way to do this is to set an initial value for ϵ , such as 1.0. You then set an ending value for ϵ , such as 0.1. A hyper-parameter you can set is the number of annealing steps it takes for the starting value of ϵ to reach the ending value. For example, over the first 100,000 steps, the network's probability of taking a random action will decrease at a constant rate. At the end of the annealing steps, the network will ideally be taking actions that are rewarding and can be used to learn even further. At this point, the probability of taking a random action is the final ϵ value (in our case, 0.1).

3.5 Double Deep Q-Network

Another problem with the current architecture is that researchers discovered some Q-values become overestimated and therefore that potential action becomes more likely to be favored over others [4]. This is what you would expect to happen when a potential action has shown to provide a better future reward than the others. However, this is not the case, as Q-values are very unstable in the beginning of the training process. This causes there to be an unfair number of actions taken and will result in the network having a higher chance at converging in a local minimum. To overcome this, the researchers developed Double Deep Q-Networks [4]. This simply means there will be two instances of the network, a primary and a secondary version. The primary network will always be used to predict which action to take. The secondary network will be used to predict the target Q-value for that specified action. The primary network then gets updated to be the same as the secondary network every so often. Decoupling this prediction leads to more stable action predictions to train on and reduces the overestimation problem previously mentioned. As this overestimation is reduced, the network trains quicker and is

less likely to fall in a local minimum and therefore performs better in the long run as well.

3.6 Dueling Deep Q-Network

The last main problem with Deep Q-Networks addressed by this report is how Q-values are predicted. Q-values represent the quality of an action which can be defined as a relation between the value of being in the current state and the advantage of taking a specific action:

$$Q(s, a) = V(s) + A(a)$$

Researchers found that sometimes the best action to take is optimally decided by not caring about the value of the current state and the advantage of an action equally [5]. The fix to this problem is by again decoupling the predictions for these values. By letting the neural network learn both $V(s)$ and $A(a)$ separately, it learns when to care about each one differently. This lead to better predictions for the Q-values and therefore less likely to get stuck in a local minimum.

3.7 Our Final Network Architecture

Our final network architecture consists of four convolutional layers and then a layer for predicting $V(s)$ and $A(a)$ separately. This architecture was adopted from DeepMind's architecture which performed very well on all the Atari 2600 games [3].

The input consists of an 84x84x3 image for environments where the previous states wouldn't add any extra features. For Breakout, the motion of the ball would not be able to be determined from a single image. To fix this, the input is 84x84x4 where each channel is a previous state. Each 84x84 image is calculated in grayscale and downsampled as needed. The full architecture is shown in table **3.7a**. All optimizations discussed in this report were implemented in our network.

Table 3.7a						
Layer	Input	Filter size	Stride	# Filters	Activation	Output
conv1	84x84x3	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
conv4	7x7x64	7x7	1	512	ReLU	1x1x512
fc	512			# actions		

4 Results and Analysis

As a team, we tried multiple times to train a network on the Atari game, Breakout. After long training sessions with no visible learning being done, it was obvious we needed to try something else. We needed to ensure that the model that we had could learn anything at all. Similarly to how we first ensure a model can overfit on a small enough dataset, we needed proof that our model was correct. To do this, we started with a more simple environment, Grid-World.

4.1 GridWorld

The problem we faced is that no environment on the OpenAI Gym was suitable for our capabilities on Gauss, while still being easy enough to test and debug our model. We used a modified version of GridWorld [2], which is written to be a simple game that uses almost the same structure as a game on OpenAI Gym. This allowed us to solidify



our network and easily transfer it to apply to Atari games on OpenAI Gym.

The concept behind GridWorld is very simple. The hero (a blue square) must navigate to rewards (green squares) and avoid fire (red squares), using only the actions Left, Right, Up, and Down. Once all of the rewards are completed, the game ends. The game is terminated with a negative reward if the hero encounters fire.

At first, the network that we had wasn't performing at all. Under the easiest circumstances we could provide, after fully training, our agent could complete less than 20% of episodes. Even at that, the only episodes that the agent could complete reliably would be those that had a reward (green), that spawned next to the hero (blue), as shown below.

After thinking through the game and how it should be configured to best help our agent learn, we came up with several optimizations which we believed would help training including:

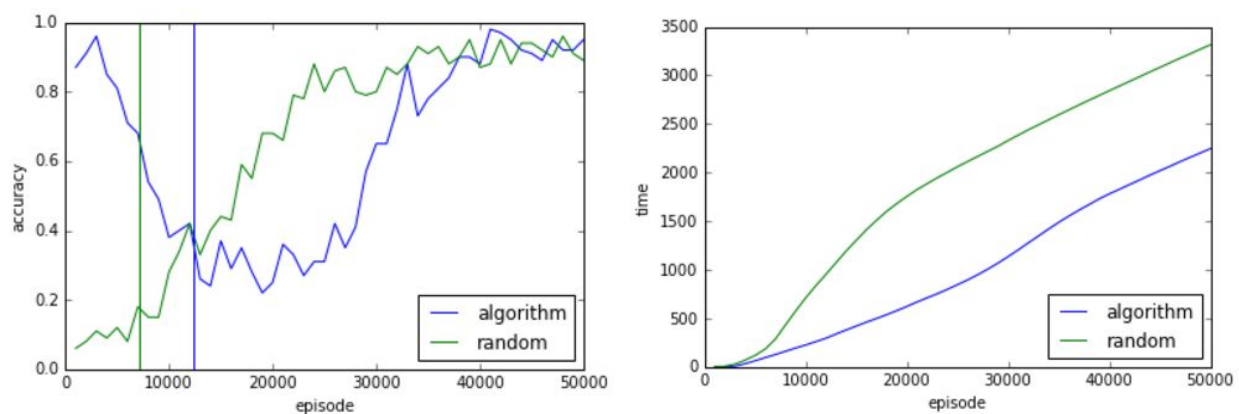
1. Ending the episode if the agent attempted to go off screen
2. Not adding a penalty to every step taken
3. Instead of starting with random movements, implement a simple algorithm to go straight to the goal (to be used to populate the experience buffer)
4. Tweaking hyper parameters to ensure a more steady learning (before we were trying to optimize for speed)

With all these considerations in mind, we were able to produce an agent that started to learn GridWorld with only one reward and one fire block in each instance.

Ending the episode early if the agent attempted to go off screen and giving that action a negative reward was another major factor in both increasing the learning rate and decreasing training time. For us to come to this conclusion, we had to take a step back and think from the perspective of playing the game. After failing to train our agent, we loaded our network and rendered the screens to get an idea of what was actually

happening in the game. We saw that the agent was going straight to the edge of the screen and staying there until the episode timed out. By ending the episode early, we were able to decrease training time by a significant amount, and also prevented reinforcing negative behavior.

The third optimization made was an effort to speed up the initial training process. Our thought was that with random steps and only one reward on the map, the hero might have a hard time ever finding that reward. We implemented a simple algorithm that moves the hero vertically until it is even with the object, and then horizontally until it hits the object. This does not take into account the position of the fire. Graph 4.1a shows the error based on the number of episodes for these two approaches. You can see that initially, as expected, the algorithm does a lot better than the random movements. However, as soon as the number of annealing steps taken increases and the probability to take an action provided by the algorithm or randomness decreases (indicated by vertical lines), the algorithm becomes worse and worse. Although the algorithm approach isn't especially useful for increasing accuracy, because the hero is always moving toward the reward, episodes take far fewer average steps per episode in the beginning. This is shown by graph 4.1b.



Graph 4.1a on the left and 4.1b on the right.

The biggest factor in being able to train our agent was to change the hyper parameters in our system. Doubling the episodes that it took to update our main network from our target network and decreasing the learning rate by 10x were two changes that,

by themselves, were absolutely necessary to be able to train our agent. This proved to us just how brittle these networks can be.

4.2 Breakout

After ensuring that our model worked on the easy GridWorld problem, we jumped to trying to train our network on the classic Atari game, Breakout. Training on Breakout proved to be very slow because of the vast difference in steps/episode. In GridWorld, there are an average of 3-4 steps that the game has to take to complete one iteration of the game. In Breakout, the game takes around 200 steps just to lose in the lowest amount of time possible. That means that even if our network became proficient at playing it, it would take thousands of steps per episode.

Another problem with training breakout compared to GridWorld is that using random movements, it is very hard to get a score over 2. Random movements will rarely ever hit the ball because of the paddle width.

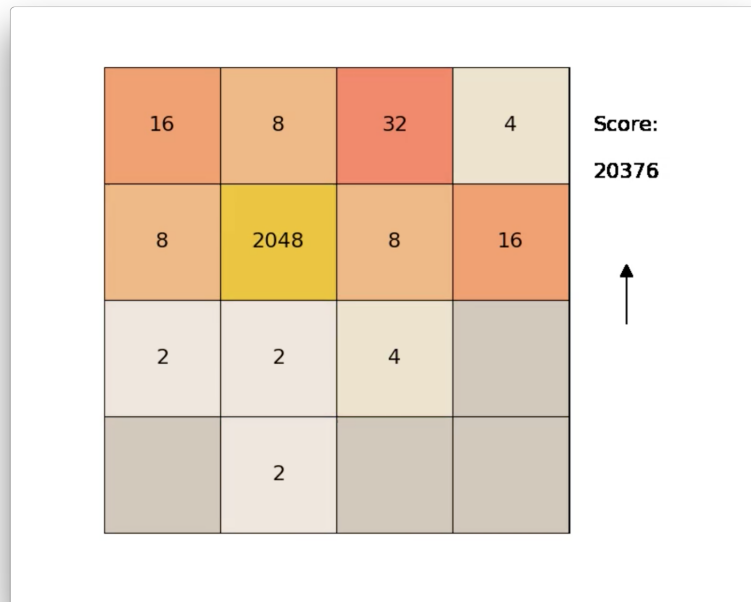
Along with having a lot more steps, the Atari environment is displayed as a 210x160x3 pixel image. This image was converted to grayscale as a 210x160x1 image. Then to match the network's architecture, each image was resized into an 84x84x1 image. Since Breakout has more complexities than the other games, the network would never be able to tell what the current motion of the ball or paddle is. In order to fix this, we stacked the last 4 states on top of each other as different channels. So the final input the neural network received had the shape 84x84x4. It turns out that this requires a lot more computational time than the other environments.

Because of the reasons listed, even after training for over eight hours, our network was not showing signs of learning Breakout. We thought that if we could train our network on ourselves playing the game, it would "jump-start" the training, and our network might learn. However, with time running low on

the project, we opted to create a new game that would be easier to train on.

4.3 2048

After optimizing our network for playing GridWorld, we then applied it to playing the popular game 2048. In this game, the number 2 will spawn in a 4x4 grid after each move, in a random location. Each move is in a given direction (Left, Right, Up, and Down). Every time a move occurs,



16	8	32	4
8	2048	8	16
2	2	4	
	2		

Score: 20376

↑

all numbers will slide in the move's direction. If a number collides with an identical number, the numbers are combined into their sum. This allows the player to work up to creating 2048.

We used the exact same network that learned how to play GridWorld and applied it to 2048. In this game, if you are doing random movements, your final score will average 7000. After less than 12 hours of training, our network was able to achieve a score of 35500. To give perspective, winning the game is considered to be when you first get the 2048 tile, and this happens at a score of around 22000.

5 Conclusions

Our goal throughout this project was to create a generic neural network that we could apply to multiple different problems and

have success with each problem. We were able to teach our network to play two games with success, GridWorld and 2048.

At all points along this project, when our network wasn't performing well, inspiration for any changes were taken from thinking about how a human mind would think about the problem. For example, we noticed initially that our network in GridWorld was trying to make the hero go off the screen, so it would get "stuck". To prevent this, we thought that we should "teach" the network that that was a bad move, so we applied a negative reward to that. Now our network will almost never predict an action that will take it off the screen. This is a very apt way to optimize any aspect of a neural network, because we are effectively trying to recreate what the human brain does, and how it thinks.

Although big, richly labeled datasets are nice to have and will commonly train quickly, they are often hard to obtain and very expensive. In order to offset this cost, deep learning algorithms need to be able to take advantage of the ability to generate their own data. End-to-end training augments this process, providing a solid basis for a network and decreasing unnecessary human involvement.

References

- [1] OpenAI Gym <https://gym.openai.com/>
- [2] GridWorld <https://github.com/awjuliani/DeepRL-Agents>
- [3] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).
Available: <https://arxiv.org/pdf/1312.5602v1.pdf>
- [4] Van Hasselt, H., Guez, A., & Silver, D. (2015). Deep reinforcement learning with double Q-learning. *CoRR*, *abs/1509.06461*.
Available: <https://arxiv.org/pdf/1509.06461v3.pdf>
- [5] Wang, Z., de Freitas, N., & Lanctot, M. (2015). Dueling network architectures for deep reinforcement learning.

arXiv preprint arXiv:1511.06581.

Available: <https://arxiv.org/pdf/1511.06581v3.pdf>