Deep Q-Learning Networks vs. Policy Gradient Learning in an OpenAI Pong Environment

Yash Bhutwala, Matt McNally, Kenny Rader, John Simmons

Bucknell University, Computer Science Department

1. Introduction

Games and toy problems often provide interesting environments to explore different aspects of artificial intelligence and machine learning. Through the years as this field developed, more and more approaches to learning have been created. Naturally, some work better than others. We can use video games as a great testing ground for these different approaches in order to determine definitively which is the best and most efficient way for a machine to learn.

Our experiment compares two reinforcement learning methods; Deep Q-Learning Networks (DQN) and Policy Gradient (PG) learning. We tested both of these approaches by creating an agent that utilizes each algorithm, gave them equal training against a control agent, and then measured their effectiveness against said control agent.

1.1 Goal of the Project

The overall objective of our project was to find out which method of reinforcement learning performs better in the Atari game Pong. In order to do this, we implemented a DQN agent and a PG agent. Both of these agents competed against the hard-coded AI that comes with the game of Pong. Our agents trained against this bot for 5000 games, and then the results of the next 100 games were kept for data analysis. This data was then used to compare our two agents and obtain a definitive answer to which method performs better in this context.

2. Existing Solutions

2.1 Policy Gradient

Policy gradient (PG) is one of the reinforcement learning techniques that is widely used in problems with continuous action spaces. PG relies upon optimizing parameterized policies with respect to the expected return by gradient descent. The basic idea is to represent the policy by a parametric probability distribution $\pi_\theta(a|s) = P\ [a|s;\ \theta]$ that stochastically selects action a in state s

according to parameter vector θ. Policy gradient algorithms typically proceed by sampling this stochastic policy and adjusting the policy parameters in the direction of greater cumulative reward.

In order to understand the mechanisms of policy gradients, it is easier to first review how supervised learning works and explain how PG differs from it. With supervised learning, the network would be fed an image and it would ouput some probabilities, in the case of Pong, for two classes UP and DOWN. As an example, the figure below shows the log probabilities (-1.2, -0.36) for UP and DOWN respectively. In supervised learning the network would have access to a label. For example, it might be told that the correct thing to do right now is to go UP (label 0). With this provided labeled data, the network would enter gradient of 1.0 on the log probability of UP and run back propagation to compute the gradient vector. This gradient would then tell how the network should update its weights to make it slightly more likely to predict UP. As a result, the network would now be slightly more likely to predict UP when it sees a very similar image in the future.

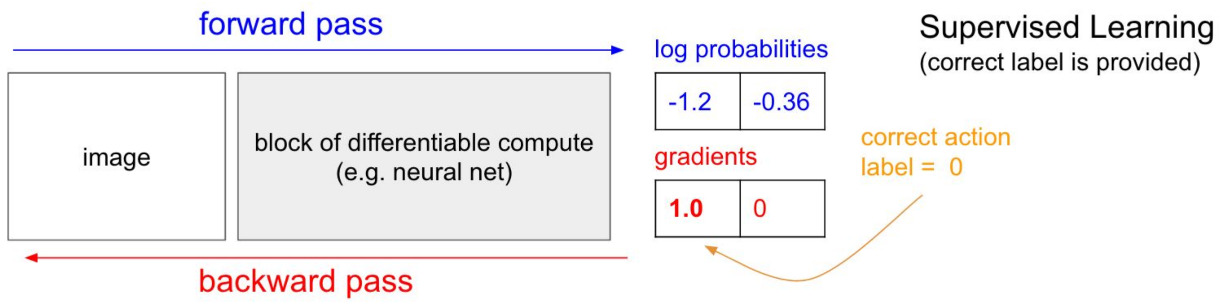

Figure 1: Supervised Learning Example

However, the challenge comes when there is no labeled data to tell the network what the right move is. This is where PG comes in. In our case of Pong, after one action (moving the paddle up or down), we do not have an idea of whether or not this was the right action. Still, the policy network calculated probability of going UP as 30% (logprob -1.2) and DOWN as 70% (logprob -0.36). The network can then sample an action from this distribution. The trick with policy gradient is that we can cheat and treat the action we end up sampling from the probability distribution as the correct action. At the end of the game, the agent will discover whether it won or not. At this point, it can fill in +/-1 reward for probability for the action and similar to supervised learning, do backpropagation to find a gradient that either encourages/discourages the network to take that action for that input in the future.

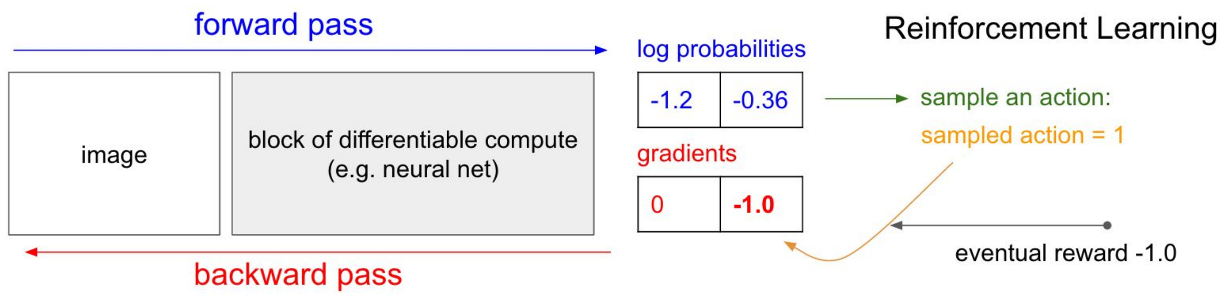

Figure 2: Reinforcement Learning Example

One of the major setbacks for policy gradient methods is that PG agents have to actually experience a positive reward, and experience it often in order to eventually and slowly shift the policy parameters towards repeating moves that give high rewards. Firstly, PG learns differently than how a human does. For example, a normal human doesn't have to experience running into a wall a few hundred times before they slowly start avoiding to do so. Secondly, in a game such as Montezuma's Revenge, in which a player must jump down, climb up, get the key, and open the door, a human understands that acquiring a key is useful whereas a PG agent would sample billions of random moves and 99% of the time fall to its death or get killed by the monster.

(Karpathy 2016). In other words, it's hard for a PG agent to “stumble into” the rewarding situation.

2.2 Deep Q-Learning Network

A Deep Q-network (DQN) is an algorithm built to combine reinforcement learning with deep neural networks to create an adaptive and intelligent agent. DQN is based on the reinforcement algorithm Q-learning and convolutional neural networks.

Q-learning is a reinforcement learning algorithm designed to be model-free. The algorithm incrementally calculates which action will be most beneficial for an agent by calculating a Q-value. This Q-value is calculated by discounting future rewards to determine the highest possible return on an action. Once an action has been performed the strength of the action is evaluated and altered accordingly. Q-learning is very simple in its techniques but because of this fact “it can operate as the basis of far more sophisticated devices.”(Watkins, 1992)

To leverage the simplicity of Q-learning, DQN replaces the primitive function for calculating the Q-value with an artificial neural network; however, we cannot use a standard neural network. If the agent is to learn in a manner that is similar to a human, we would expect the agent to have the same information as a human player, namely the pixels on the screen. By shoving thousands of pixel values into a neural network we would be creating droves of unnecessary nodes where only a fraction would suffice. To solve the issue of finding and evaluating the relevant portions of the screen, DQN employs a convolutional neural network that is “used to build up progressively more abstract representations of the data”(Mnih, 2015) throughout its layers. This is illustrated in Figure 3.

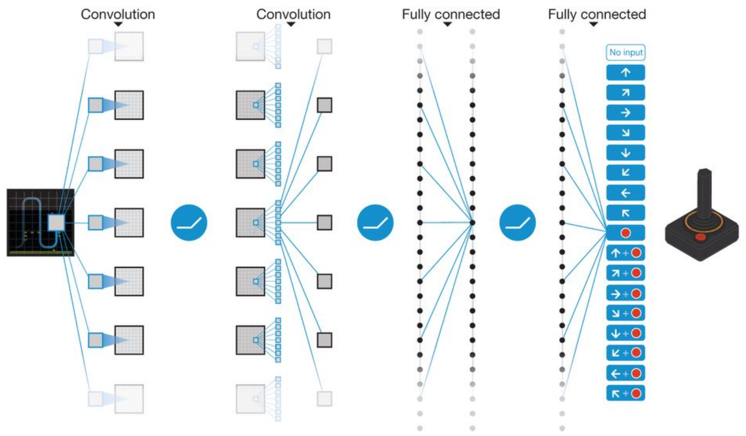

Figure 3: A convolutional neural network architecture (devsisters, 2016)

The trouble is that Q-learning "is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value"(Mnih, 2015). This stability issue is believed to be caused by the correlation between adjacent actions. To account for the correlation between actions DQN uses experience replay. This biologically inspired technique stores experiences, namely state, action, reward, and new state, at each step. Then, when propagating the error back through the convolutional neural network during training, the agent chooses an experience "uniformly at random from the pool of stored samples."(Mnih, 2015)

DQN has been shown to be a promising algorithm when used in the context of Atari games. It has even outperformed human level game play in many Atari environments, as shown in Figure 4 on the following page.

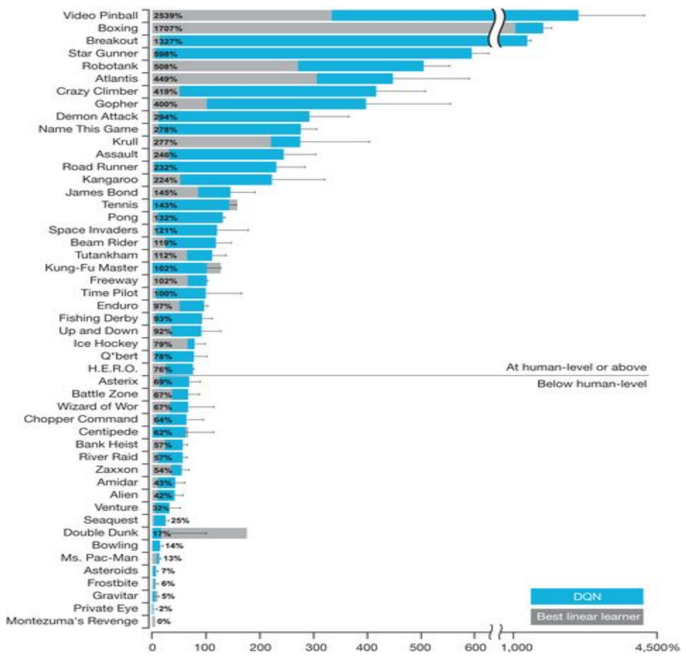

Figure 4: DQN Performance in Atari (Mnih, 2015)

3. Implementation

3.1 Policy Gradient

We adopted our implementation from Dr. Andrej Karpathy (Karpathy 2016). Dr. Karpathy uses a policy gradient architecture in combination with a neural network to compute a probability of moving up. The inputs to this neural network are the images from the game which are processed by the code before entering into the neural network. The processing involves looking at all of the pixels in the image, where each pixel is represented by three integers. The code filters out all of the background pixels so that only the valuable information, the paddles and the ball, are left. It then computes the change in the current image from the previous image. This difference representation is the final input into the neural network. The difference between the two images allows the network to determine how fast the ball is moving and in what direction. The neural network then outputs a probability of moving up. Next, the implementation samples from that probability distribution and tells the agent to move up or down. These calculations are carried out until the round is over. Then, the code determines if it won or lost that round. The agent wins the round if it scores and loses if the other agent scores. However, the backpropagation does not occur until the episode has finished (one of the agents gets to 21 points). Once either agent scores 21 points, our agent passes the result of the episode through the backpropagation algorithm to compute the gradient for the weights. After ten episodes have completed, the gradient from each episode is summed, and the weights are moved in the direction of the gradient. This training process is repeated until the weights are fine tuned enough to consistently beat the Pong agent.

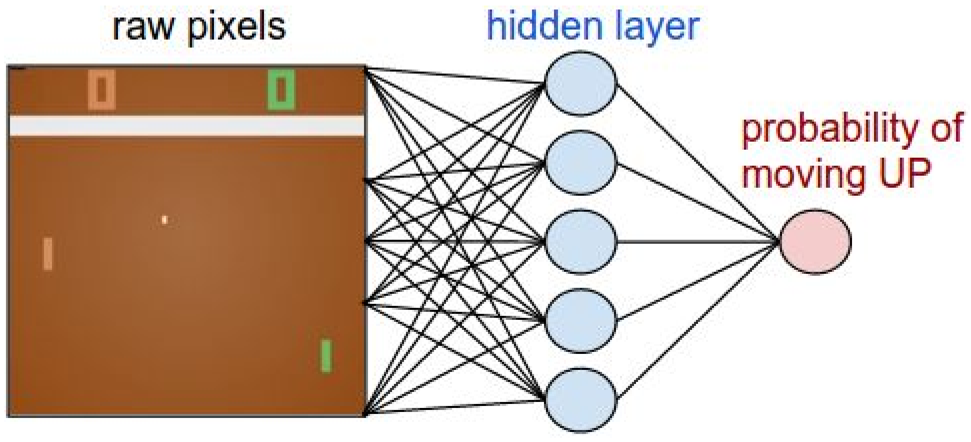

Figure 5: Policy Gradient Network Architecture (Karpathy 2016)

One of the design choices that we had to make was which type of policy gradient we wanted to implement. Vanilla and Natural both offer benefits, but ultimately we went with vanilla for a few reasons. First, Vanilla is a much simpler policy to implement compared to Natural. Natural is a modification of Vanilla policy, and since we didn't have much experience implementing policy gradients before, we wanted to ensure our agent would function before trying to improve how quickly it will become a good agent. The second reason we chose Vanilla is that given the right parameters, Vanilla can sometimes converge to usable values faster than Natural (Jan 2010).

Another design choice that we made was not using tensorflow, unlike our DQN partners. The main reason was that we didn't need to. A lot of our computation was doable using numpy, and since our neural network only had 1 hidden layer, it was a relatively basic network that didn't require too much setup.

3.2 Deep Q-Learning Network

The implementation for DQN was adapted from the solution by devsisters on github (devsisters, 2016). This implementation creates an agent utilizing an artificial neural network with three convolutional layers and a hidden layer to compute the action value (or Q-value).

The algorithm is simply an extension of Q-learning built on a neural network. The network predicts the best action based on the Q-values at its disposal (with a degree of randomness), acts upon said action, and observes the screen, reward, action, and terminal status that result. The reward is kept to backpropagate through the network and adjust the Q-values accordingly. At several predetermined points throughout this process, the agent may stop and save the weights of its network in order to pick up where it left off, should something go awry during the training process.

Since devsisters' code is geared towards implementing Breakout, some minor alterations had to be made to make it work with Pong and the experimental environment that we were going for. Changing the environment was as trivial as changing the gym environment from "Breakout.v0" to "Pong.v0" in all of the appropriate places. Since our experiment claimed that we needed to train the DQN agent for 5000 games before pitting it against the PG agent, the training method of agent.py was modified to save the weights every 500 games (among other times). Once "Agent-5000" was saved, training was manually stopped to move into the experimental phase.

The play() method of agent.py was heavily modified to run exactly 100 games and output the highest scoring game and the average reward across all games. These would be our heuristics for measuring DQN against PG.

4. Results

4.1 Policy Gradient

The result of a policy gradient agent trained on 5000 episodes are captured in the Results folder of the pg directory and are as follows:

```
Best Reward: -17

Average Reward: -20.19
```

The policy gradient agent was run an additional three times to verify the consistency of these results. Here are the results of these additional runs:

```
Run 2:   Best Reward: -17

         Average Reward: -20.21

Run 3:   Best Reward: -18

         Average Reward: -20.2

Run 4:   Best Reward: -15

         Average Reward: -20.32
```

The standard deviation of the best reward is 1.25831, while the standard deviation of the average reward is 0.060553.

The PG agent did not win a single game in any of these runs due to not having nearly enough training experience. Similar to DQN as we will see, we expect the PG agent to not learn until about a couple of hundred thousand trials, which is something that is just not feasible given this project's timeframe. However, one thing to notice between the performance of DQN vs. PG is that the standard deviation for the PG agent is much lower than that of DQN. This indicates that the PG algorithm is much more stable at learning and converges quickly towards a good agent than DQN. There has also been several papers noting the superiority of PG's performance over DQN (Volodymyr 2016).

4.2 Deep Q-Learning Network

Our results of running the play() method of agent.py on Agent-5000 are as follows (Note negative scores represent a lost game, with the absolute value of the reward being how much the agent lost by):

```
Best Reward: -10.0

Average Reward: -15.51
```

In the interest of scientific reproducibility, Agent-5000 was run an additional three times to verify the consistency of these results. Here are the results of these additional runs:

```
Run 2:   Best Reward: -5.0

         Average Reward: -15.83

Run 3:   Best Reward: -5.0

         Average Reward: -15.66

Run 4:   Best Reward: -8.0

         Average Reward: -15.3
```

The standard deviation of the best reward is 2.44949, while the standard deviation of the average reward is 0.22517.

A discussion of these results is in order. One should note that Agent-5000 did not win a single game in any of these runs. This could be for a variety of reasons, but the most obvious is that it doesn't have nearly enough training experience. The devsisters github (devsisters, 2016) provides a few graphs showing how their agent performed. According to Figure 6, their agent didn't start to exhibit any marked increase in learning until around 4.5 million trials, which is something that is just not feasible on our own humble machines given the timeframe that we have. Another thing to note is the high deviation in the best reward heuristic. This is likely due to flukes. Occasionally, the agent will perform noticeably better than usual before returning to its average reward for several more games. One can chalk this up to lucky outcomes of random actions. Given more time, these lucky outcomes would propagate through to make the agent actually perform better, but unfortunately we will not see these results.

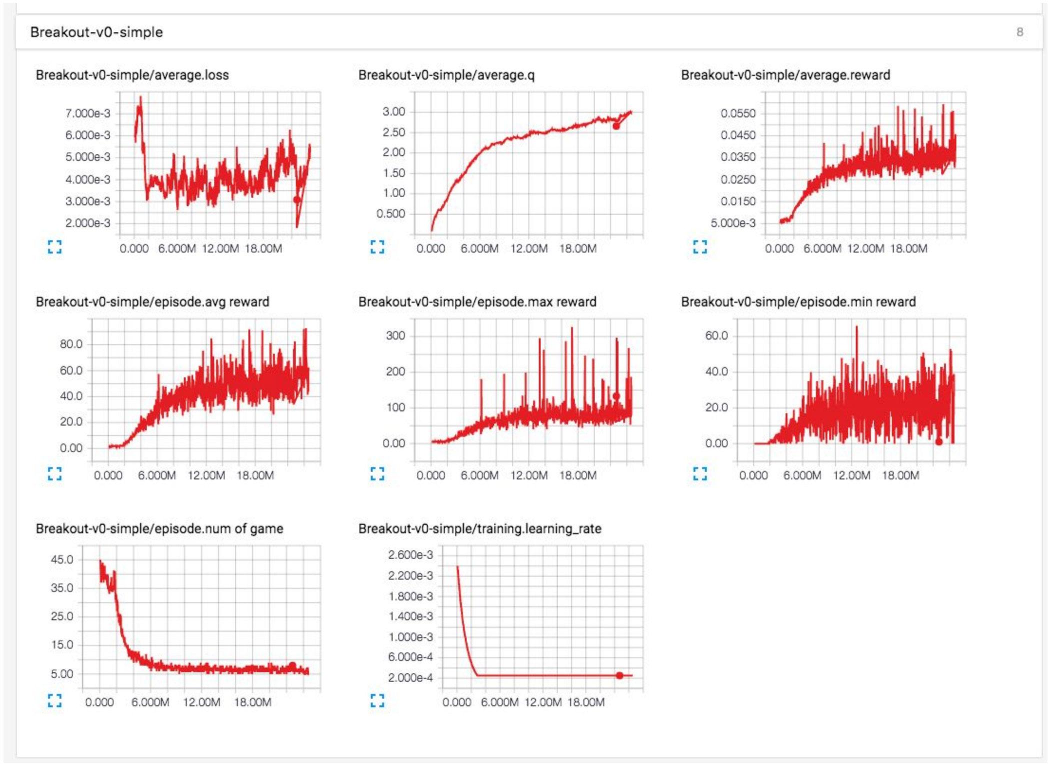

Figure 6: Analysis of devsisters Breakout DQN agent (devsisters, 2016)

5. Conclusion

Overall, even though we did not prove it conclusively, there is strong evidence that policy gradient method works better than the Deep Q-learning algorithm for the game of Pong. In addition to not having enough time for the project to sufficiently train the two agents, it was challenging to be able to train the agents for a long period of time because we discovered a few bugs after a few trainings, which made our dataset useless as we had to start over again. Regardless, our team worked well in dividing up the work, and we all learned about both the algorithms.

6. References

Mnih, V., Kavukcuoglu, K., & Silver, D. (2015). Human-level control through deep reinforcement learning. *Nature,* 529-533. doi:10.1038/nature14236

Watkins, C.J.C.H. & Dayan, P. Mach Learn (1992) 8: 279. doi:10.1007/BF00992698

D. (2017, March 09). Devsisters/DQN-tensorflow. Retrieved May 01, 2017, from https://github.com/devsisters/DQN-tensorflow

Karpathy, A. (2016, May 31). Deep Reinforcement Learning: Pong from Pixels. Retrieved May 01, 2017, from http://karpathy.github.io/2016/05/31/rl/

Jan Peters (2010) Policy gradient methods. Scholarpedia, 5(11):3698.

Volodymyr Mnih and (2016). Asynchronous Methods for Deep Reinforcement Learning. CoRR, abs/1602.01783.