Popular Approaches to Deep Reinforcement Learning

Zhe Cai, Silvia Ionescu, Monil Jhaveri, Andrew Levy
{s20525xx,mjhaveri,sionescu,levya}@bu.edu

1. Introduction

A. Task

The main task of our project was to compare and contrast three popular approaches to deep reinforcement learning: value iteration methods, policy gradient algorithms, and actor-critic techniques. Our goal was to both understand the theory behind these methods and assess their performance on various learning tasks. The learning tasks in which we evaluated the performance of our agents included Atari video games and robotic control simulations.

B. Reinforcement Learning Background

Reinforcement learning is a subfield within machine learning focused on the idea of learning by interaction and by trial and error. Learning by interaction is an important concept to both understand and be able to mathematically model because a significant part of human learning is done by interacting with the environment.

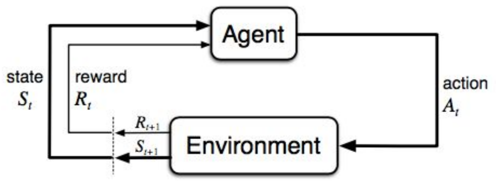


Figure 1: Reinforcement Learning Setup

There are three main components to a reinforcement learning problem: states, actions, and rewards. The state represents the current condition of the environment. If the agent were some sort of robot, the state may include the joint angles and joint velocities. Actions represent how the agent acts in a moment in time. In the case of a robot agent, the action at time *t*, A_t, would be the torque applied to each joint. Rewards are a numerical quantity the agent receives after taking action A_t in state S_t. The rewards given are assumed to be consistent with the ultimate behavior that is desired. Given these definitions, a reinforcement learning problem proceeds according to the loop shown in Figure 1. The agent observes a state S_t, takes action A_t, and then receives R_{t+1} and S_{t+1}. After observing S_{t+1}, the agent takes A_{t+1} and receives R_{t+2} and S_{t+2}. This loop then continues until a some terminating condition is reached. For the agents we trained, this occurs when an episode of training is complete, which may occur when an Atari game is won or lost or a simulated robot falls or crashes.

From this basic setup, we can derive two key functions, the policy and action value functions. These will be important to understanding the three reinforcement learning approaches we examine. The policy function, denoted $\pi(s)$ or $\pi_{\theta p}(s)$, if it is parameterized by parameter vector θ_p, represents how the agent will act in given state. The policy function is thus a mapping from states to actions ($\pi_{\theta p}$: S → A). The action value function, also known as the Q-value function and denoted $Q_{\theta q,\pi}(s,a)$, tries to measure how the long-term effectiveness of an action in a certain state. The Q-function represents the expected long-term discounted reward given that the agent takes action *a* in state *s* and thereafter follows policy π. The function is a mapping from states and actions to a number that represents the expected long-term reward ($Q_{\theta q,\pi}$: S x A → $\mathbb{R}$), and can be mathematically written as

(1) $Q_{\theta q,\pi}(s,a) = E[\Sigma_{k=0:\infty}(\gamma^k R_{t+k+1}) \mid S_t = s, A_t = a]$,

in which γ represents a discount rate. A discount rate is used to give the agent a preference for shorter-term rewards over longer-term rewards. The Q-function is important because it provides the agent with a way to compare the long-term effectiveness of different actions in the same or similar states. In our implementation, the parameterized policy and action value functions, $\pi_{\theta p}$ and $Q_{\theta q,\pi}$, will be represented with neural networks.

2. Related Work

We used several papers to help us understand, implement, and compare the various methods we used.

Mnih et al. present deep Q-learning as a method to enable agents to learn control policies at or exceeding human level in several different Atari games in [1]. Two of the key changes to the Q-Learning technique include (i) convolutional input layers to better understand the input image and (ii) a replay buffer, which is essentially a “memory”-like device that improves learning by enabling the neural network to learn from past experience. One downside with the DQN algorithm is that the algorithm is applied to environments with discrete and not continuous action spaces. Thus, DQN could not be applied to continuous problems like robotic control.

Several other papers provide theory and best practices for implementing policy gradient methods. Sutton et al. prove the policy convergence properties of the policy gradient method [2]. The authors show that if the parameters of the neural network are updated in the direction of the policy gradient, a local optimum in expected reward can be attained. Sutton and Barto, however, discuss one of the main drawbacks to generic policy gradient methods [3]. Convergence to a local optimum can be difficult if the policy gradient is calculated with actual episode rewards versus a Q-function.

Lillicrap et al. present the Deep Deterministic Policy Gradient actor-critic algorithm in [4], a method that shows how deep Q-learning can be modified to work for continuous action spaces.

3. Approach

Most reinforcement learning algorithms use the same high-level strategy for learning by trial and error. Reinforcement learning agents learn by following a two-step cycle that alternates between (i) Experimentation and Evaluation and (ii) Learning phases, as shown below.

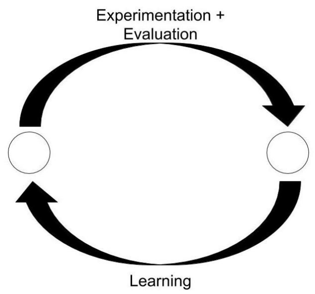


Figure 2: RL Algorithm Strategy

In the "Experimentation and Evaluation" phase, the agent tries different actions and then evaluates their effectiveness. In the “Learning” phase, the agent modifies its behavior depending on whether the experimental actions were successful or not. The reinforcement learning algorithm will then repeat this loop numerous times, and upon completion the agent should be able to perform a task in a near-optimal manner. We next describe the manner by which three popular RL approaches complete the “Evaluation” and “Learning” phases. We show that the key differences are mostly due to (i) the reward metric to be maximized and (ii) whether the action space is discrete or continuous.

A. Value Iteration Methods

Background

Agents that use value iteration methods in discrete environments find the policy that maximizes reward by finding the action that maximizes Q-value in each given state. If a policy selects the action with a near-optimal Q-value in every state, this policy will produce a near-optimal amount of total reward. A consequence of using Q-values is that the agent’s policy becomes simple -- choose the action with the largest Q-value ($a_t = \text{argmax}_a\ Q(s_t,a_t)$).

In the “Experimentation and Evaluation” phase, agents will try out different actions and then assess these experimental actions by estimating their Q-values. Value method agents typically test different actions within a state by using an ε - greedy policy. Using this approach, there is an ε probability that the agent will select an action that does not have the highest Q-value in the state, thus, encouraging exploration.

To complete the “Experimentation and Evaluation” phase, the agent needs to assess the long-term value of the previous action. One common approach is to take advantage of the recursive nature of the Q-function. Equation (1) above that defines the Q-function can be simplified as follows:

$$\begin{aligned}
(2)\quad Q(s_t,a_t) &= E[\Sigma_{k=0:\infty}(\gamma^k R_{t+k+1}) \mid S_t = s, A_t = a] \\
&= E[R_{t+1} + \gamma\Sigma_{k=0:\infty}(\gamma^k R_{t+k+2}) \mid S_t = s, A_t = a] \\
&= E[R_{t+1} + \gamma * Q(s_{t+1},a_{t+1})] \\
(3)\quad &= E[R_{t+1} + \gamma * Q(s_{t+1},\pi(s_{t+1}))]
\end{aligned}$$

According to equation (2), an unbiased estimate for $Q(s_t,a_t)$ can be found by calculating the sum of the next reward R_{t+1} and the product of the discount rate and the Q-value of the next state s_{t+1}, $Q(s_{t+1},\pi(s_{t+1}))$. The $\pi(s_{t+1})$ term is used to derive the action in the next state because the Q-function assumes the policy $\pi(s)$ is followed after the first action is taken. Equation (2) provides the agent a target of what $Q(s,a)$ should be. The agent can then use this target to update its current estimate of $Q(s,a)$. If the learning task was simple (i.e., finite state and actions spaces) the following learning update could be used.

(4) $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \eta[R_{t+1} + Q(s_{t+1},\pi(s_{t+1})) - Q(s_t,a_t)]$

Here, $Q(s_t,a_t)$ is updated by a small amount in the direction of the difference between the target estimate, $R_{t+1} + Q(s_{t+1},\pi(s_{t+1}))$, and the old estimate, Q(s,a). If the task is more complex (i.e., infinite state space, finite action space), $Q_{\Theta q}(s_t,a_t)$ can be represented by a function approximator such as neural network. Θ_q could then be updated in the direction of the gradient that minimizes the mean-squared error of the difference between the old and target estimates. To summarize, the goal in the "Experimentation and Evaluation" stage of value methods algorithm is try out different actions and "internalize" their effectiveness by finding a target Q estimate and then updating the Q-function.

In the "Learning" stage of the learning cycle, agents improve their policy based on the results from the previous "Experimentation and Evaluation" phase. The agent updates its policy in state s_t to take into account this new Q-value. In value methods, this policy update again is simple -- find the action that now has the largest Q-value.

After the "Learning" phase is complete, the learning loop continues. The agent will select an action, which may at the moment be non-optimal. The agent will then evaluate the Q-value of that action and update its policy in that state, taking into account the new Q-value. This cyclical process can continue for thousands to millions of iterations, which each loop yielding a policy that produces larger total rewards. Indeed, Sutton and Barto note in [3] that in finite state, action space environments, assuming that all state and action combinations are repeatedly visited, this cyclical learning process will produce an optimal policy. A more detailed view of the value method learning loop is presented below.

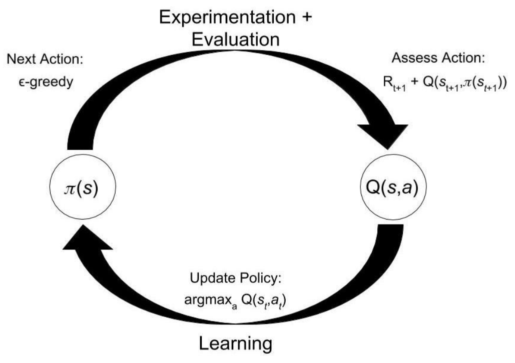

Experimentation + Evaluation
Next Action:
ϵ-greedy
Assess Action:
$R_{t+1} + Q(s_{t+1},\pi(s_{t+1}))$
$\pi(s)$
$Q(s,a)$
Update Policy:
$\text{argmax}_a\, Q(s_t,a_t)$
Learning

Figure 3: Value Iteration Method Learning Loop

Our Algorithm

We are using a value iteration method known as Deep Q-Networks (DQN) to train an agent to play various Atari games.

Due to the near infinite state space of many of these games, DQN approximates the Q-function using a convolutional neural network. The states in Atari games are images and so the state space is very large. There are hundreds of pixels on each frame. Each of those pixels is represented by a 3-dimensional RGB value, in which each dimension is a real number between 0 and 255. Here, the Q-function is represented by a convolutional neural network as shown in Figure 4. The input layer will be 4 consecutive game images. More than one is required to give the agent a sense of the velocity of the components. The output will be the Q-values for every possible action.

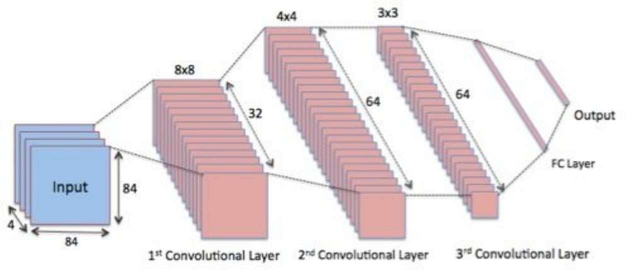


Figure 4. Deep Q-Learning Neural Network

A second defining feature of DQN is the manner in which the algorithm updates the Q-function neural network. If the algorithm were to repeatedly sample an action, calculate the target estimate, and then perform gradient descent on the loss function, which is the MSE of the difference between the target and old Q-estimate, learning may be unstable because the neural network would then be trained using highly correlated inputs. Neural networks require independent and identically distributed inputs. The algorithm overcomes this issue by using the concept of a replay buffer [1]. The replay buffer is essentially a "memory"-like device that may store up to millions of game transitions. The replay buffer is then used to enable batch gradient descent. Instead of sampling one action, and then performing gradient descent on the loss function using one data point, the algorithm can use the replay buffer to sample uncorrelated actions and then perform gradient descent using numerous data points. These extra data points should produce a more accurate gradient descent update and thus a more accurate Q-function. The entire process is similar to mini-batch stochastic gradient descent, with dynamically changing training data.

B. Policy Iteration Methods

Background

Policy gradient (PG) algorithms use a different strategy to finding a near-optimal policy in an environment with a discrete action space. Policy gradient methods do not use an action value function as the reward metric to optimize. Instead, PG algorithms look to find a policy that maximizes expected reward of an episode. Policy gradient techniques also use a parameterized policy network to sample actions as these methods cannot rely on the Q-function to generate the policy. This policy network has a softmax (multiple actions) or sigmoid (two actions) final layer and consequently will output the probability of taking each available action. Thus, the PG policy will be denoted $\pi_{\theta p}(a_t|s_t)$.

Like other reinforcement learning algorithms, in the "Experimentation and Evaluation" phase, agents that learn with PG will try out new actions and assess their effectiveness. Because the policy function in PG agents is inherently probabilistic, there is no need to force ε-greedy exploration like with the value method. Instead, agents can experiment by simply sampling actions from the probabilities of $\pi_{\theta p}(a_t|s_t)$. Next, because the goal is to maximize the expected reward of an episode, E[R], a series of actions will be evaluated by simply recording the reward from one entire episode. The reward from one episode is an unbiased estimate of E[R], so this is a valid estimation technique. Thus, the "Experimentation and Evaluation" phase lasts at least one episode in contrast to value methods in which the phase lasts one action step.

In the "Learning" phase of the learning loop, agents will use the evaluation made in the previous phase to improve the policy. Agents will update the parameters of the policy in the direction of the "policy gradient", which is the gradient of E[R] with respect to θ_p, $\nabla_{\theta p}$E[R]. Sutton and Barto in [3] show that the gradient, $\nabla_{\theta p}$ E[R] can be simplified to:

$$(5)\quad \nabla_{\theta p} E[R] = E\Big[R * \left(\nabla_{\theta p} \textstyle\sum_{t=0:T-1} \log(\pi_{\theta p}(a_t|s_t)\right) \Big]$$

(5) shows that the gradient of the expected return of an episode with respect to the policy parameters is the expected product of (i) the return of an episode and (ii) the gradient of the log of the policy at each time step of the episode. Here the subscript t refers to each time step. $\nabla_{\theta p} \sum_{t=0:T-1} \log(\pi_{\theta p}(a_t|s_t))$, the unscaled policy gradient, is the gradient of the output of the policy with respect to the parameters at every time step. This shows how to tweak the parameters of the policy to make the action that was selected more likely to occur in the future. Given this insight, (5) basically says that in order to push the average episode reward higher, modify the current policy used at every time step in proportion to the total reward received. In other words, if the trajectory yielded a strong total reward, make each action that was used in the trajectory more likely to occur in the future. Given the definition of the policy gradient in (5), we can find an unbiased estimate of the policy gradient by multiplying the actual reward from the latest episode by the gradient of the sum of the log probabilities of each action that was taken, which is shown in (6) below.

$$(6)\quad \nabla_{\theta p}E[R] \cong R \;*\; \nabla_{\theta p} \textstyle\sum_{t=0:T-1} \log(\pi_{\theta p}(a_t|s_t))$$

In the "Learning" phase, PG agents will perform gradient ascent and move the parameters of the policy θ_p in the direction of (6), the direction that should cause the greatest increase in E[R]. The gradient ascent step is shown in (7).

$$(7)\quad \theta_p \leftarrow \theta_p + R * \nabla_{\theta p} \textstyle\sum_{t=0:T-1} \log(\pi_{\theta p}(a_t|s_t))$$

Here R is the value calculated in the first phase of learning. To summarize, in the "Learning" phase, the agent learns from experience by updating its policy parameters in the direction of the policy gradient, which requires the reward estimate from the previous "Experimentation and Evaluation" stage.

A summary of the policy gradient learning loop is shown below.

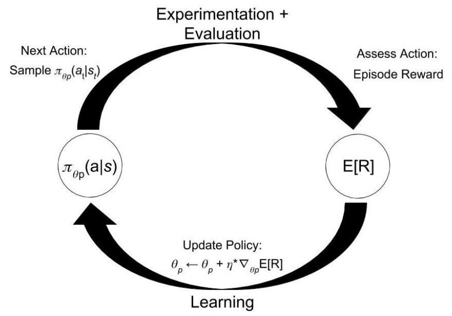

Experimentation + Evaluation
Next Action: Sample $\pi_{\theta p}(a_t|s_t)$
Assess Action: Episode Reward
$\pi_{\theta p}(a|s)$
E[R]
Update Policy: $\theta_p \leftarrow \theta_p + \eta * \nabla_{\theta p}E[R]$
Learning

Figure 5: Policy iteration method learning loop

Our Algorithm

The algorithm we used is called REINFORCE and is very similar to the generic policy gradient method described above. REINFORCE also evaluates the reward estimate after one episode and then moves onto the "Learning" phase.

C. Actor-Critic Methods

<u>Background</u>
Algorithms that learn both policy and action value functions are known as actor-critic algorithms. There are various types of actor-critic methods. We used a method known as Deep Deterministic Policy Gradient (DDPG).

<u>Our Algorithm</u>
DDPG is essentially deep Q-learning for environments with continuous action spaces. Similar to value methods, DDPG seeks to find the policy that maximizes the Q-value in every state. However, no longer can the agent simply derive its policy by selecting the action with the largest Q-value. Because the action space is continuous, there would be an infinite number of actions for which to find the maximum. Instead, the Deep Deterministic uses a second neural network to approximate the policy.

In the "Experimentation and Evaluation" phase, the agent tests new actions by adding noise to the action prescribed by the policy neural network in the current state, $\pi_{\theta p}(s) + \varepsilon$. The agent then evaluates the effectiveness by taking the same unbiased estimate of $Q_{\theta q,\pi}(s,a)$ that was used for value methods, $R_{t+1} + Q_{\theta p}(s_{t+1},\pi_{\theta p}(s_{t+1}))$, which is the sum of the next reward and the Q-value of the next state-action pair. The agent then updates its Q-neural network network. The updating of the Q-network is a way for the agent to "internalize" the long-term effectiveness of not only the latest action but of previous actions taken in the same or similar states.

The use of a policy neural network changes how DDPG agents learn from their experimentation in the first phase. Because the policy is no longer to select the action with the maximum value, the agent cannot simply complete another argmax to account for the updated Q-values from the first phase. Instead, DDPG agents follow a process similar to agents that learn with the policy gradient method. DDPG agents improve their policy by moving the policy parameters θ_p in the direction of the gradient of $Q_{\theta p}$ with respect to θ_p. That is, the parameters of the policy are moved in the direction that should produce the greatest increase in Q-values as shown in (8).

(8) $\Theta_p \leftarrow \Theta_p + \eta * \nabla_{\theta p} Q_{\theta q,\pi}(s_t,\pi_{\theta p}(s_t))$

This gradient is called the deterministic policy gradient. Using the chain rule, this gradient can be simplified to

$\nabla_a Q_{\theta q,\pi}(s_t,a) * \nabla_{\theta p} a$

(9) $= \nabla_a Q_{\theta q,\pi}(s_t,a) * \nabla_{\theta p}\pi_{\theta p}(s_t)$

In the above equation, $a = \pi_{\theta p}(s_t)$, the action prescribed by the policy in the current state. The deterministic policy gradient, is the product of the gradient of the Q-function with respect to the action prescribed by the policy and the gradient of this action with respect to its parameters Θ_p [4].

The accuracy of the of the deterministic policy gradient and the effectiveness of the learning update relies heavily on the "Experimentation and Evaluation" phase. The first gradient in (9) is $\nabla_a Q_{\theta q,\pi}(s_t,a)$ is essentially the slope of the Q-function at the action prescribed by the policy. If the agent has not experimented sufficiently with different actions in the current state or similar states and thereby not developed a robust sense of the shape Q-function, this gradient or slope will not be accurate. An inaccurate gradient would result in a poor parameter update and slower learning.

The full learning loop for the DDPG agent is shown below:

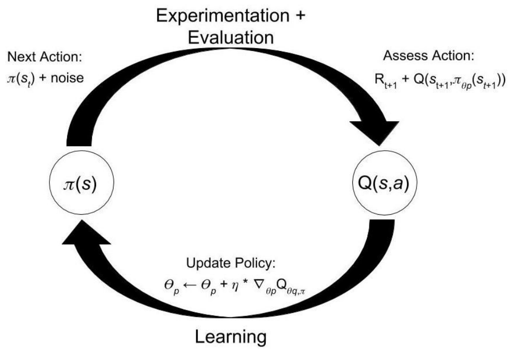

Experimentation + Evaluation
Next Action: $\pi(s_t)$ + noise
Assess Action: $R_{t+1} + Q(s_{t+1},\pi_{\theta p}(s_{t+1}))$
$\pi(s)$
$Q(s,a)$
Update Policy: $\Theta_p \leftarrow \Theta_p + \eta * \nabla_{\theta p} Q_{\theta q,\pi}$
Learning

Figure 6: DDPG learning loop

4. Dataset and Metric

We will use the OpenAI Gym environments to train the agents. These game environments provide the state, action, and reward data that are necessary to implement the Q-Learning, policy gradient, and deep deterministic policy gradient algorithms.

The key metrics that we will use to assess the performance of the different algorithms are the average reward per episode and the number of training episodes required to achieve this average.

5. Results

Please see attachments for demonstration videos

A. Deep Q-Networks (DQN)

Deep Q-Network (DQN) was implemented for the Atari games Breakout, Pong, and Enduro. We achieved solid results on all three games.

We performed the following preprocessing steps. OpenAI environment outputs 210x160x3 RGB images that are preprocessed by first taking the maximum value for each pixel colour value over the current frame being encoded and the previous frame. This step removes flickering issues that are present in Atari games, where some objects are present only in either even or odd frames. The second preprocessing step was to resize the input images to 84x84 pixel frames and convert them to grayscale.

Our final CNN had the following structure. The input layer had four consecutive frames stacked together to account for the velocity of the ball changing. The remaining convolutional and fully connected layers are described in the table below.

Layer	Input	Filter Size	Filters	Stride	Output
Conv1	84x84x4	8x8	32	4	20x20x32
Conv2	20x20x32	4x4	64	2	9x9x64
Conv3	9x9x64	3x3	63	1	7x7x64
FC1	7x7x64	-	-	-	512
FC2	512	-	-	-	3

Table1: DQN Convolutional Neural Network.

The correlation of consecutive training samples was overcome by implementing an experience replay buffer of the 1 million most recent frames. The replay memory was randomly sampled with size 32 mini batches that were used to update the network parameters by using an RMSProp optimizer. During training, the exploration-exploitation balance was achieved by applying a ε-greedy approach with ε annealed linearly from 1.0 to 0.1 over the first million frames.

To make the algorithm more stable, a separate network was used to generate target-Q values. The target network has the same architecture as shown in Fig. 4 and described in Table 1, but every T = 10000 steps the parameters from the Q network were copied to the target network.

Following the DQN implementation of Mnih et. al.[1], the error term from the update $r + \gamma \max_a Q_\theta(s_{t+1},a) - Q_\theta(s,a)$ was clipped between -1 and 1 in order to improve the stability of the algorithm.

The results of the Pong DQN implementation is shown in Fig. 7, where rewards above 0 means that we are winning the game. After training for 20000 episodes the mean episode reward for Pong is about 7 with a learning rate = 0.00025.

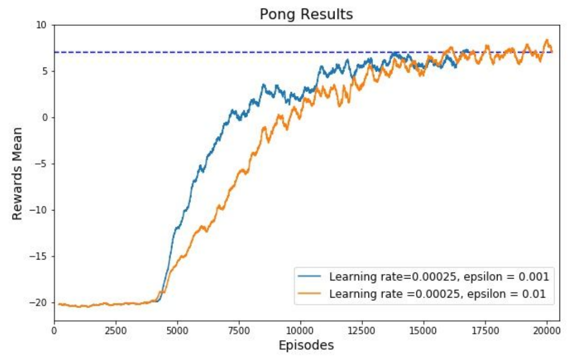


Figure 7: Pong DQN final results.

Fig. 8 shows our DQN results for the Breakout Atari game. A few different learning rates were tested and a learning rate=0.001 achieved a mean episode reward of 35.

Finally, our DQN implementation was tested with the Enduro Atari game as shown in Fig. 9. A reward mean of 160 was obtained by applying a learning rate= 0.00025.

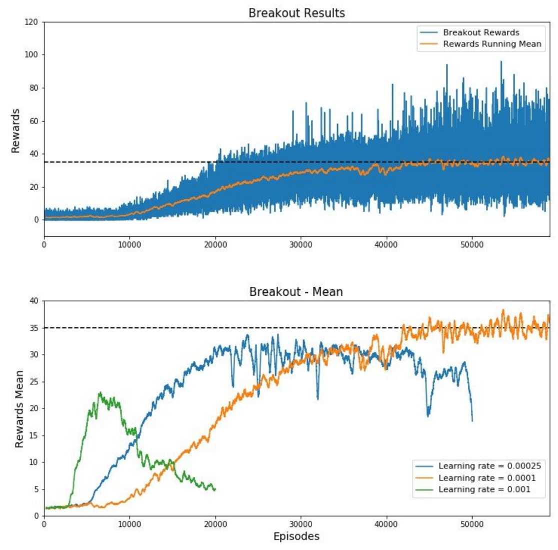


Figure 8: Breakout DQN results with different learning rates.

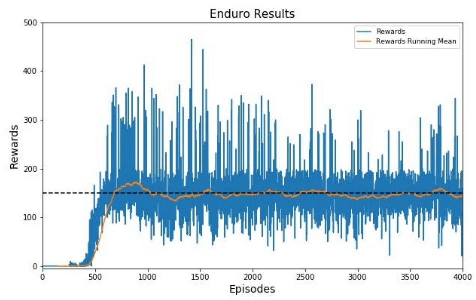


Figure 9: Enduro DQN final results.

B. Policy Gradient (PG)

We applied the policy gradient method to both Pong and Breakout, and evaluated the performance of various architectures, including CNN and ANN (i.e., feedforward NN). We observed that using a CNN did not help to improve the agent performance in this case, which can be seen in the figures below. A wider ANN and dropout were also not helpful. The reason for this poor performance may be the simpleness of the two games and the fact that the agent is only trained once per episode, compared to once per step in DQN. Increasing model complexity may have actually resulted in agent becoming stuck at a local maximum or taking too long to learn. This repetitive behavior can be seen in the videos submitted.

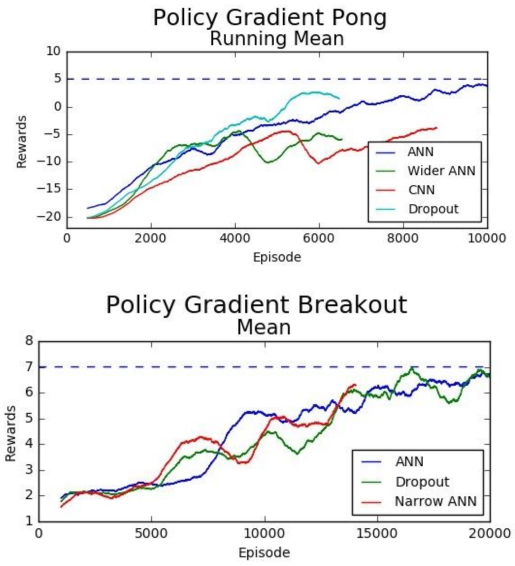


Figure 10: Performance of PG under different parameters

The best parameters of the architectures we implemented is shown below, a detailed version of the overall flow diagram is in Appendix 1:

Layer	Input	Output
FCL	84x84x1	200
FCL	200	3

Table 2: PG ANN parameters

Our final results in Figure 11 show that the PG agent was able to achieve solid results in Pong but struggled to learn Breakout. Figure 12, which shows the actual gradient updates and rewards achieved during the last episode of training, provides some additional insights into the performance of both the Pong and Breakout agents. The small magnitudes of the Pong gradients throughout indicate that the agent finds a local maximum behavior after a period of learning. The relatively quick leveling of the running mean curve in Figure 11 also confirms this. On the other hand, the sparse rewards that the Breakout agent earned may be one of the causes of the poor performance. Without a consistent reward signal, the PG agent will likely struggle to determine which actions to make more and less likely.

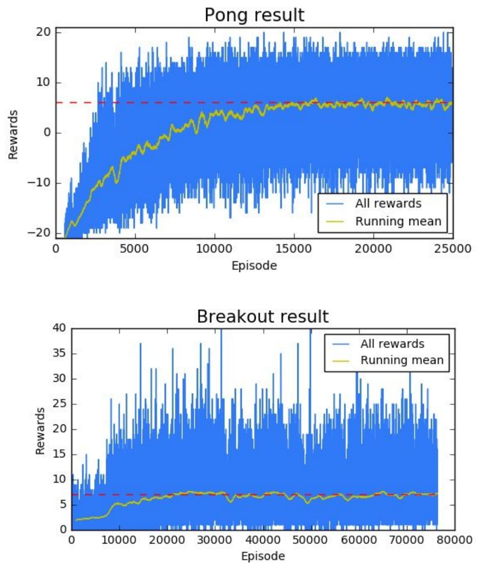


Figure 11: PG final result

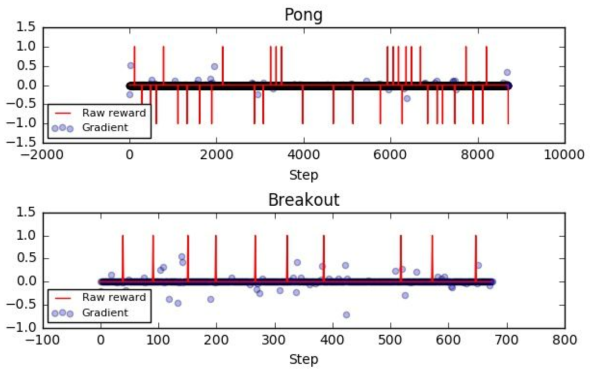
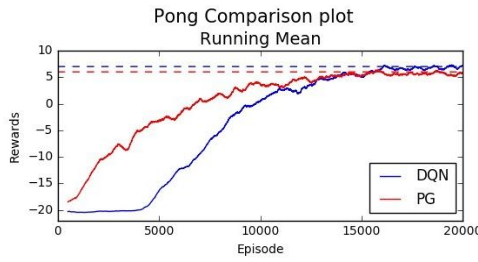


Figure 12:Policy gradients and rewards in final Episode, Pong (above) and Breakout (below)

C. DQN and PG comparison



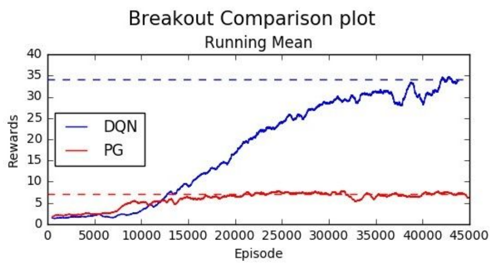


Figure 13: DQN and PG comparison

Our main conclusion is that DQN generally outperformed policy gradient in Atari Games. We believe DQN outperforms PG because DQN does a superior job of identifying the actions in a state that are both beneficial and harmful. This should be expected because DQN explicitly tries to determine the long-term effectiveness of each action in a state by calculating Q(*s*,*a*) values. On the other hand, policy gradient methods do not try to determine the long-term value of individual actions within a state. Rather, the algorithm makes actions more or less likely depending on the ultimate reward of the episode. As a result, policy gradient may be making poor actions more likely in a state because the agent ultimately receives a positive reward for the episode. One general advantage PG may have over DQN is speed of learning, as in clear in the Pong comparison plot. For many tasks, learning the Q(*s*,*a*) for a sufficient number of state-action pairs may be a lengthier process than optimizing a policy.

D. Deep Deterministic Policy Gradient (DDPG)

Our goal in implementing DDPG was to have agents learn both simple and complex robotic control tasks, and we successfully met this goal. Below are some details on the 3 simulations we submitted.

1. Lunar Lander
   a. Action Dim: $\mathbb{R}^2$ (2 engines)
   b. State Dim: $\mathbb{R}^8$ (position, velocity)
2. Bipedal Walker
   a. Action Dim: $\mathbb{R}^4$ (4 joints)
   b. State Dim: $\mathbb{R}^8$ (position, LIDAR)
3. OpenSim Musculoskeletal Simulation
   a. Action Dim: $\mathbb{R}^{18}$ (Muscle Activations)
   b. State Dim: $\mathbb{R}^{31}$ (Bone positions)

In each video, the agent clearly learns from trial and error. This is also clear in the Q-values output from the agent. Below is a graph of the Q-values from one of the more simple agents we trained, a car learning to swing itself up a hill.

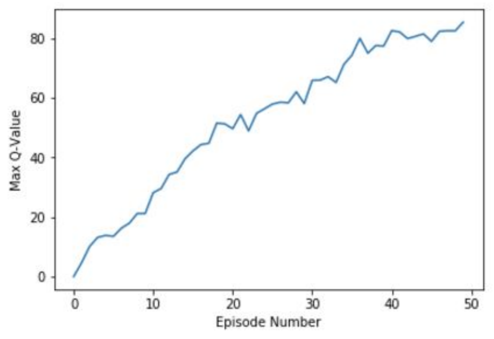


Figure 14: Mountain Car Max Q-values

The graph above shows the max Q-values output by the agent during each episode. As expected, the Q-values increase over time as the agent modifies its policy parameters to produce higher Q-values.

6. Detailed Roles

Task	**File names**	**Who**
Implemented DDPG	DDPG Folder	Andrew Levy
Implemented DQN	DQN Folder	Silvia Ionescu
Implemented DQN	DQN Folder	Monil Jhaveri
Implemented PG	PG Folder	Zhe Cai
Wrote Approach Section		Andrew Levy
Wrote Results Section		Silvia Ionescu, Zhe Cai

7. Code Repository

https://github.com/moniljhaveri/DeepLearningProject
The demo videos are also included in github in addition to Blackboard

References

1) Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." *Nature* 518.7540 (2015): 529-533.
2) Sutton, McAllester, et al. “Policy Gradient Methods for Reinforcement Learning with Function Approximation.” *Advances in Neural Information Processing Systems* (2000)
3) Sutton and Barto. *Reinforcement Learning (Draft).* MIT Press (2017)
4) T. Lillicrap et al. *Continuous Control with Deep Reinforcement Learning*. ICLR 2016.