Popular Approaches to Deep Reinforcement Learning

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

1. Task

The main task of our project is to implement agents that can learn how to complete tasks such as Atari video games and robotic control simulations using deep reinforcement learning techniques. We plan to compare and contrast three popular deep reinforcement learning approaches: value function, policy gradient, and actor-critic methods.

Two key challenges that our group will face in implementing these agents are the exploration-exploitation dilemma and the credit assignment problem. The goal in reinforcement learning is to find the optimal policy, or sequence of actions, that maximizes long-term reward. The exploration-exploitation dilemma represents the difficult tradeoff that occurs when trying to find the optimal policy. The agent can choose to more thoroughly explore the state-action space to try to find a policy that yields a better reward but at the cost of slower convergence. Or the agent can exploit what it has learned so far and explore a more limited region of the state-action space, thereby converging faster to a less optimal policy. The credit assignment problem refers to the issue of determining which actions within a series of actions were most responsible for causing an outcome. This difficulty arises because rewards are often sparse and time-delayed.

2. Related Work

Mnih et al. provide one of the first successful applications of deep reinforcement learning to a complex environment. The authors show that by adapting the classical Q-Learning technique in a few important ways, agents can learn control policies at or exceeding human level in several different Atari games [1]. Two of the key changes to the Q-Learning technique include convolutional input layers to better understand the input image and a replay buffer, a "memory" device that improves learning by enabling the neural network to learn from past experience that is not limited to the most recent actions. One flaw with the DQN algorithm the authors developed is that the algorithm can only be applied to environments with discrete and not continuous action spaces. Thus, DQN could not be applied to problems like robotic control.

Several other papers provide theory and best practices for implementing policy gradient methods. Sutton et al. prove the 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 reward estimate component of the policy gradient does not involve a value function.

Lillicrap et al. present the Deep Deterministic Policy Gradient actor-critic algorithm in [4], an actor-critic method used for continuous action spaces. The authors provide a closed form solution to the deterministic policy gradient, which shows how the parameters of the actor network should be updated to maximize the value network.

3. Approach

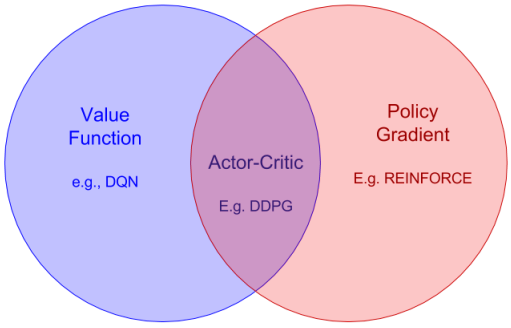

Figure 1. RL Approach Venn Diagram

The goal of our project is to compare the effectiveness of three popular approaches to deep reinforcement learning approaches: value function, policy gradient, and actor-critic methods. All of these techniques attempt to find the policy that maximizes long-term reward through trial-and-error. However, each technique accomplishes the goal in its own way.

A. Value Methods

Background

Value methods find an optimal or near optimal policy with the help of a value function. The value function, also known as the Q-function, is used to evaluate the expected long-term reward of the current policy. Mathematically, this can be expressed as [3]

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

In the above equation, the Q-value of the state-action combination (s,a) is the expected long-term reward G_t given that action *a* in state *s* is taken at time *t* and then the current policy π thereafter. Value methods are used in tasks with discrete action spaces so the optimal policy π is to take the action with the highest Q(s,a) given current state s. The long-term reward G_t is the discounted sum of rewards R_{t+k+1} given k time-steps after action *a* was taken at time *t*. γ represents the discount rate, which can be used to modify the importance of future rewards relative to current rewards.

Value function methods determine optimal policies by using a cyclical learning process known as generalized policy iteration [3]. This cyclical process has two main components: (i) policy evaluation and (ii) policy improvement.

In the policy evaluation stage, the agent spends time interacting with its environment to try to better estimate the long-term rewards of its current policy. One approach to estimating the Q-values, or long-term rewards of various state-action pairs, is to take advantage of the recursive nature of the Q-function.

$$Q(s,a) = E[\Sigma_{k=0:\infty}(\gamma^k R_{t+k+1}) \mid S_t = s, A_t = a]$$
$$= E[R_{t+1} + \gamma\Sigma_{k=0:\infty}(\gamma^k R_{t+k+2}) \mid S_t = s, A_t = a]$$
$$= E[R_{t+1} + \gamma * \max_a(Q(s_{t+1},a)] \quad (2)$$

Per the last equation above, an unbiased estimate for Q(s,a) can be determined by sampling one action from the current state and then summing the subsequent reward and the largest Q-value of the next state [R_{t+1} + $\max_a Q(s_{t+1},a)$]. During the policy evaluation stage, an agent can use this unbiased estimate as the target for a learning update of its current Q-function estimate. For instance, if the learning task was simple (i.e., finite state and actions spaces) and Q(s,a) was stored in a table, the following learning update could be used.

$$Q(s,a) \leftarrow Q(s,a) + \eta[R_{t+1} + Q(s_{t+1},a) - Q(s,a)] \quad (3)$$

Here, Q(s,a) is updated by a small amount η in the direction of the difference between the new estimate, R_{t+1} + $Q(s_{t+1},a)$, and the old estimate, Q(s,a). If the task is more complex (i.e., infinite state space, finite action space), $Q_\theta(s,a)$ can be represented by a function approximator such as neural network. θ could then be updated in the direction of the gradient that minimizes the mean-squared error of the difference between the old and new estimates. Thus, the process of learning the Q-function is similar to supervised learning. The target estimate here is equivalent to the label, and Q(s,a) is the predicted output.

To promote exploration of the state-action space, value method agents learn to use a slightly noisy policy during the policy evaluation phase. As a result, value method agents usually have two different policies: one policy that it is optimizing and another that it uses to explore. In simple discrete action environments, the policy to be optimized always follows the same strategy - simply choose the action with the largest Q-value. This policy is always used when determining the $Q(s_{t+1},a)$ value in the target estimate, which is always $\max_a Q(s_{t+1},a)$. However, a slightly different policy is used to sample the first action that produces the R_{t+1} value in the target estimate. If the optimized policy was followed here as well, the agent might only learn the Q-function for a certain region of the environment. Instead, one way to force the agent to explore the environment and develop a more robust representation of Q(s,a) is to use an ε - greedy policy. Using this approach, there is a 1 - ε probability that the agent will sample an action that is consistent with the optimal policy. However, there is an ε probability that the agent will sample a random action. An ε-greedy approach is one way for value method approaches that use deterministic optimal policies to find a balance in the exploration vs. exploitation dilemma discussed earlier.

In the policy improvement stage of the learning process, the agent modifies the policy that it is optimizing to one that achieves a larger average Q-value. In the finite action spaces within which value methods are used, this is simple - choose the action with the largest Q(s,a). There will be new Q-values after the Q-function was fine-tuned in the policy evaluation stage so there will be different actions that maximize the Q-values for every state.

After improving the policy, the cyclical generalized policy iteration learning process continues. The agent

spends time estimating the new policy and exploring new areas in the policy evaluation stage. The agent then further fine tunes the optimized policy based on what it has learned. In the value-based approaches, 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 have been visited, this generalized policy iteration approach will converge to the optimal policy. There is no such guarantee when neural network function approximation needs to be used for the Q-function in infinite state space environments. However, convergence to a local optimum that represents a near-optimal solution is achievable.

Our Algorithm

We are using a value method known as Deep Q-Networks (DQN) to train an agent to play the Atari game Breakout competently.

Due to the near infinite state space of Breakout, DQN approximates the Q-function using a convolutional neural network, $Q_\theta(s,a)$. The state space of Breakout 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. As a result, it would be impossible to implement the Q-function as a table. Instead, the Q-function for Breakout likely requires a universal function approximator such as neural networks. The structure of the CNN will look similar to that shown in figure 2. The input layer will be a series of 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.

A second defining feature of DQN is the manner in which the algorithms performs policy evaluation. Similarly to the generic value method approach described above, DQN uses temporal difference learning to estimate the Q-function. The algorithm samples actions using an ε-greedy policy and uses $[R_{t+1} + \max_a Q_\theta(s_{t+1},a)]$ to determine a target estimate for the Q-value. The algorithms then updates the parameters of $Q_\theta(s,a)$ in the direction of the gradient of the loss function, which here is the squared difference between the target estimate and the prior estimate.

$$\text{Target Estimate: } y = R_{t+1} + \max_a Q_\theta(s_{t+1},a) \quad (4)$$

$$\text{Loss: } L = (y - Q_\theta(s,a))^2 \quad (5)$$

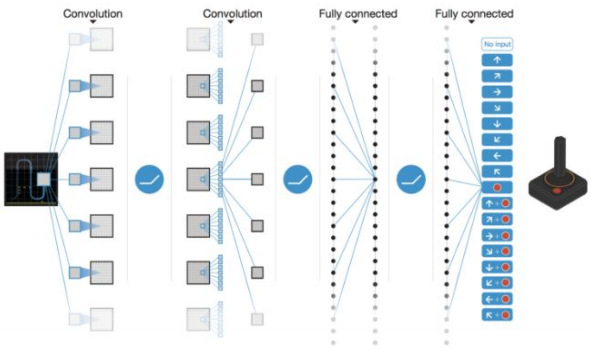

Figure 2. Deep Q-Learning Neural Network Mapping

However, the algorithm uses a unique approach to optimizing the loss function. The algorithm cannot repeatedly sample an action, calculate the target estimate, and then perform gradient descent on the loss function because the neural network would then be trained using highly correlated inputs because sequential states of a game look highly similar. Neural networks require independent and identically distributed inputs. If the inputs are highly correlated, the network will be prone to diverge and less likely to learn a generalizable function. The algorithm overcomes this issue by using the concept of a replay buffer [1]. The replay buffer is essentially a memory like device that stores thousands of game transitions. The replay buffer is then used to enable batch gradient descent. Instead of sampling one action, and then calculating an estimate of the Q-function using (2), which would be problematic due to correlated inputs, the algorithm can use the replay buffer to sample hundreds of uncorrelated actions and then calculate hundreds of estimates of the Q-values. Batch gradient descent can then be used to find a more robust parameter update.

B. Policy Gradient Methods

Background

Policy gradient algorithms use a different strategy to find the optimal policy: optimize a policy directly without the requirement of a value function. As described above, value methods find a near-optimal policy by first calculating a sense of how "good" each action in each state is by estimating Q-values. Once the agent has a sense of the best choice for each state, the agent then backs into a near-optimal policy. Policy gradient methods use a different approach of iteratively improving a parameterized policy, $\pi_\theta(a|s)$, to maximize some performance variable.

Policy gradient algorithms use a gradient ascent strategy to iteratively optimize a policy relative to some performance measure. The performance criterion that is typically maximized is the expected reward for an episode (e.g., a game of Breakout), which will consist of a series of actions. In the notation described below, the performance criterion will be denoted by $R(\tau)$, which represents an estimate for total reward from a trajectory of actions τ. The goal of policy gradient is thus to maximize $E[R(\tau)]$ with respect to the parameters θ in $\pi_\theta(a|s)$. The algorithm will use gradient ascent to maximize the performance variable because the parameters of the policy, θ, will be updated in the direction of the gradient of $E[R(\tau)]$ with respect to θ. Sutton and Barto show that the gradient, $\nabla_\theta E[R(\tau)]$, is

$$E_\tau[R(\tau)\ \nabla_\theta \Sigma_{t=0:T-1} \log(\pi_\theta(a_t|s_t)) \quad (6)$$

In other words, the gradient of the expected return of an episode, is the expected product of (i) the estimate of 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. This equation is actually an intuitive result. $\nabla_\theta \Sigma_{t=0:T-1} \log(\pi_\theta(a_t|s_t))$ is essentially the gradient of the output of the policy with respect to the parameters at every time step. In other words, 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, (6) 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. If the total reward was small or negative, make the actions used less likely to occur in the future.

There are both advantages and disadvantages to using policy gradient in regards to how the algorithm deals with the issues of exploration vs. exploitation and the credit assignment problem. A benefit of policy gradient is that the parameterized policy can inherently model a stochastic policy that is capable of naturally exploring. For tasks with discrete action spaces, softmax can be integrated into $\pi_\theta(a|s)$, so the policy represents a mapping from state to probabilities of actions. However, a disadvantage is that because policy gradient techniques do not need to use Q-values, these techniques struggle to determine which actions within a series of actions are most important. For instance, when the return from one episode is used for the estimate $R(\tau)$, as is the case for our policy gradient algorithm REINFORCE, every action is made more likely to occur if there was a good final result, regardless of how beneficial the individual actions were.

Our Algorithm

The policy gradient algorithm our agents will be using to learn Atari games is REINFORCE. The key distinguishing feature with this algorithm is that the value used for the $R(\tau)$ estimate in (6) is a sample return from one episode. This will produce as unbiased estimate of the gradient, $\nabla_\theta E[R(\tau)]$. However, this strategy also produces large variance, which can slow training times.

C. Actor-Critic Methods

Background

Algorithms that learn both policy and 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).

Our Algorithm

DDPG is essentially Deep Q-Networks for environments with continuous action spaces. Similar to other value methods, DDPG uses the cyclical generalized policy iteration approach, in which the algorithm switches between policy evaluation and policy improvement. Like DQN, DDPG maintains a neural network to approximate the Q-values of its current policy. In its policy evaluation stage, DDPG samples past transitions from a replay buffer to improve the Q-value estimates of its policy. However, DDPG differs from DQN in how it implements the policy improvement stage. DDPG must use a separate neural network to approximate the policy function because the action space is continuous. To optimize this policy network, DDPG takes a similar approach to policy gradient - modify the parameters of the policy to maximize a performance criterion. However, instead of maximizing expected total reward of an entire episode, DDPG maximizes $Q_{\theta q}(s,a)$, the long-term reward given the current state and action prescribed by the policy. Because DDPG optimizes its policy to maximize the Q-function, the algorithm tries to determine the optimal action for a given state. As a result, this technique should outperform more generic policy gradient algorithm such as REINFORCE that do not differentiate among actions. The equivalent policy

gradient for DDPG that shows the direction by which to update the parameters of the actor neural network to maximize Q(s,a) is called the deterministic policy gradient and is given below [4].

$$E[\nabla_a Q_{\theta q}(s,a)\ \nabla_{\theta p}\pi(s)] \quad (7)$$

(7) simply states that the deterministic policy gradient is the product of the gradient of the Q-function with respect to the action and the gradient of the policy with respect to θ_p, the parameters of the policy.

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 score per game and the number of training episodes required to achieve this average. We hope to achieve reward averages that meet or surpass the average human scores detailed in [1].

5. Preliminary Results

***Please note short videos of our results have been submitted separately on Blackboard**

- **Deep Q-Networks (DQN)**

Deep Q-Network (DQN) was implemented for the Atari game Breakout. Below are some notable features that have been implemented.

OpenAi outputs 210x160x3 RGB images that were converted into gray scale images and resized to 80x80 pixel frame. Throughout the game the velocity of the ball changes, so to count for this change we stacked together four consecutive frames. The 4-frames stack is used as input to a convolutional neural network that has three convolutional layers and two fully connected layers. The similarity of consecutive training samples was broken by implementing an experience replay buffer.

A separate network was used to generate target-Q values. The target network has the same architecture as the function approximator network, but every T steps the parameters from the Q network are copied to the target network.

The results of our DQN implementation trained for 2000 episodes are shown in Fig. 3 with a final reward of 36. We are planning to increase the number of training episodes and test a few other greedy exploration parameters. Implementation of dueling network architectures could also improve the actor performance by determining separate estimators for the state value function and state action function. This method can produce more robust estimates by allowing our reinforcement learning agent to learn which states are valuable without having to learn the effect of each action for each state [5].

Fig. 3: Two DQN Episodes - Trained on 2000 Episodes

- **Deep Deterministic Policy Gradient (DDPG)**

We have had successful progress implementing DDPG on basic continuous control environments. We have included a video of an agent learning an environment called Mountain-Car. In this environment, the goal is for the agent, or the car, to reach the top of a hill. The agent can apply a continuous amount of forward pressure on the gas. However, the amount of forward pressure is limited so the agent cannot drive directly up the hill. Instead, the agent needs to learn to swing up and down a hill to reach the top. This short clips of selected episodes show how the agent learns over time to accomplish this task.

- **Policy Gradient (PG)**

The team has also achieved solid success using PG to train an agent to play pong. Training will be moved to a GPU to speed up progress.

6. Detailed Timeline and Roles

Task	**Deadline**	**Lead**
DDPG Implementation	04/25/17	Andrew Levy
PG Implementation	04/25/17	*Zhe Cai*
DQN Implementation Breakout	04/25/17	*Silvia Ionescu*
DQN Implementation Pong	04/25/17	*Monil Jhaveri*
Report and presentation	05/02/17	all

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.
5) Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, N. de Freitas, Dueling Network Architectures for Deep Reinforcement Learning, arXiv.org, 2015.