Comparison Of Reinforcement Learning Algorithms

Siddhant Garg **Andrew Teeter**

1 Introduction

In this project we implement REINFORCE with Baseline [9] and PPO [6]. The goal of this project was to implement some reinforcement learning algorithms and then evaluate their performance in various environments. All the environments used in this project were implemented by OpenAI Gym [3]. We will now give brief descriptions about each of the environments.

1.1 Cart Pole

The Cart Pole problem consists of a pole on a cart. The goal is to balance the pole for as long as possible without leaving the designated area or allowing the pole to fall more than 15 degrees from vertical. For each time step that the pole remains balanced, the agent is provided one reward. The agent can either move left or right. This is an example of an environment with a dense reward function as the agent is provided rewards at every time step. This is a finite horizon environment with maximum horizon of length 200. If the agent gets the reward of 200, then the agent wins and the game is terminated.

1.2 Lunar Lander

The Lunar Lander problem consists of a rocket and a landing pad designated by two flags. The goal is to land the rocket in between the two flags at a slow enough velocity to not crash. The agent gets a reward of -0.3 for every time step that the main engine is fired and -100 if the rocket crashes. The agent is given 100 reward of 100 for landing, 100 additional for landing on the landing pad, and another 10 for leg that touches the ground. The episode ends when the rocket either crashes or comes to rest. The agent can do four discrete actions those include using the left engine, using the right engine, firing the main downward engine, or do nothing. This reward function is more sparse than Cart Pole as the large rewards are given out at the end of the episode. When the agent gets a total reward of more than 200, then it is considered as a win.

1.3 Breakout

Breakout is a classic atari game that was implemented as an OpenAI gym [2]. The goal is move a paddle to bounce a ball against bricks on the top of the screen. Reward is gained for each brick broken. The agent has the ability to move right, left, or do nothing. The state is given in terms of the pixels of the game.

2 REINFORCE with Baseline

REINFORCE is a policy gradient algorithm first described by Williams [9] in 1992 and is further described by Sutton and Barto [8]. The foundation of the algorithm relies on the fact that the expected return of an episode is proportional to the gradient. This means that the policy can be updated based off of the Monte Carlo value estimates of a given episode. An extension to the REINFORCE algorithm is the use of

a Baseline. A baseline is any function that is independent of the action taken at a given time step. The addition of a baseline function is used to further reduce the variance of the REINFORCE algorithm and help to converge faster. Commonly, some estimate for the value function of the given MDP is used as a baseline. The psuedocode for the REINFORCE with Baseline algorithm can be seen below 1.

Algorithm 1 REINFORCE with Baseline[8]

Input: Differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: Differentiable value-function parameterization $\hat{v}(s, \boldsymbol{w})$
Parameters: Step sizes $\alpha^{\theta} > 0, \alpha^{w} > 0$
$\boldsymbol{\theta} \leftarrow \mathbf{0} \in \mathbb{R}^{d}$
$\boldsymbol{w} \leftarrow \mathbf{0} \in \mathbb{R}^{d'}$
while True **do**
  Generate episode following π
  for $S_t, A_t, R_t + 1$ in the episode with T steps **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ //Discounted Return
    $\delta \leftarrow G - \hat{v}(S_t, \boldsymbol{w})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{w} \delta \nabla \hat{v}(S_t, \boldsymbol{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\theta} \delta \nabla \log(\pi(A_t|S_t, \boldsymbol{\theta}))$
  end for
end while

Implementation Details

For this project, neural networks were used for the policy and value function parameterizations. This seemed like a logical choice because it allows for a large range in complexity that can be tuned to the specific environment that the algorithm is being evaluated on. It is also efficient in interpreting continuous states. Both the networks consist of a single hidden layer followed by a ReLU activation function. The networks were trained using an Adam optimizer. The policy is given by the softmax of the output logits of the policy network. The biggest downside of relying on Monte Carlo return to inform the updates of the policy and value function estimates is that there is a large amount of variance. Slightly deviating from the original algorithm, the implementation created for this project performed episode generation and policy and value function updates in batches. This was to help reduce the variance of the Monte Carlo return of the episodes and help to avoid local maxima in the return. All hyperparameters including hidden layer size, learning rates, discount rates, and batch size, were selected through the use of a grid search over various combinations.

Cart Pole

For Cart Pole, the model was trained using a hidden layer size of 32, learning rates of .02 for each network, discount rate of .99, and a batch size of 64. Given this relatively simple environment, REINFORCE with Baseline was able to solve it relatively quickly as can be seen in Figure 1. It can be seen that the agent solves the environment and the variation in returns starts to reduce as the number of episodes increases. Figure 2 shows the loss of both the policy network and the value function network during the training process. At the beginning of training, the value-function loss sharply increases at the same time that the policy loss sharply increases. This is indicative of the agent running into a new strategy that allows for a higher reward than before. In this case, the agent learns one method for stopping the pole from falling over.

Lunar Lander

For Lunar Lander, the model was trained using a hidden layer size of 64, learning rates of .005 for each network, discount rate of .99, and a batch size of 64. As can be seen in the charts in Figure 3 Lunar Lander is a much more difficult problem for REINFORCE to solve. These charts show that the algorithm learns very slowly in this environment. Two key points can be seen in the graph of steps per episode. It can be seen where the agent learns that preventing a crash using the main engine because the average number of steps per episode dramatically increases. The steps then dip as the agent reduces how often it fires the main

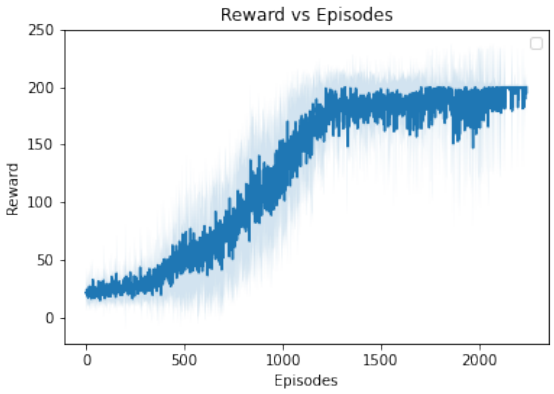


Figure 1: Results over ten training runs of REINFORCE with Baseline on the Cart Pole environment.

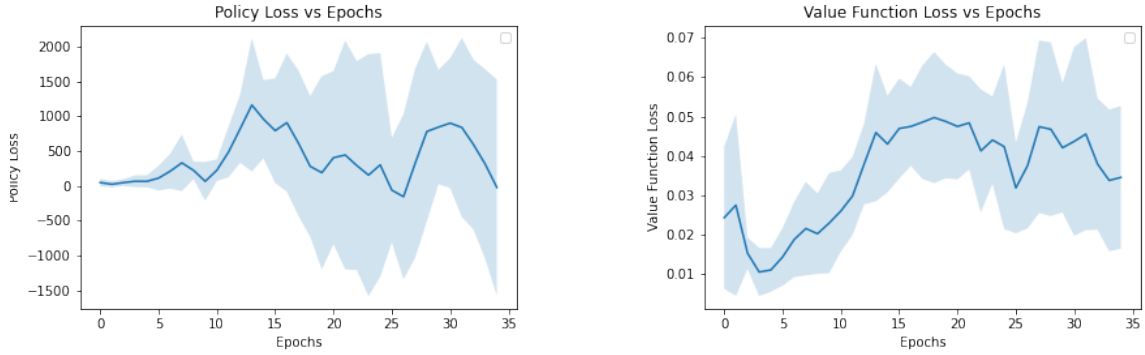


Figure 2: Training Loss over ten training runs of REINFORCE with Baseline on the Cart Pole environment.

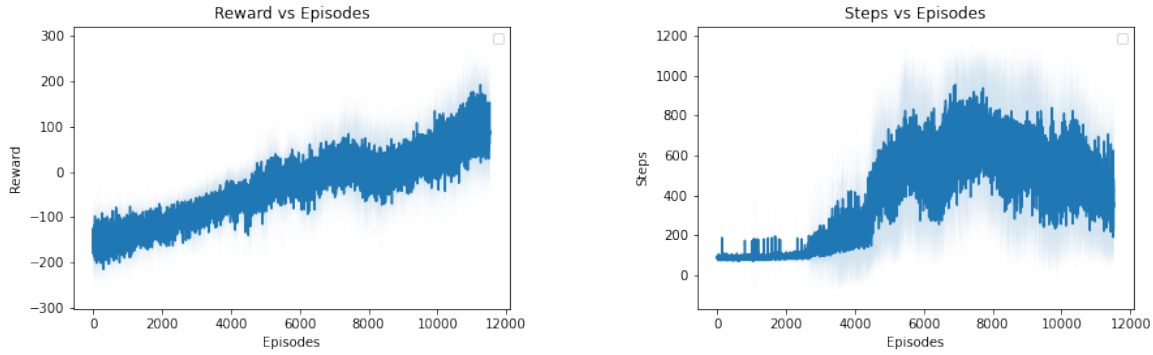


Figure 3: Results over ten training runs of REINFORCE with Baseline on the Lunar Lander environment.

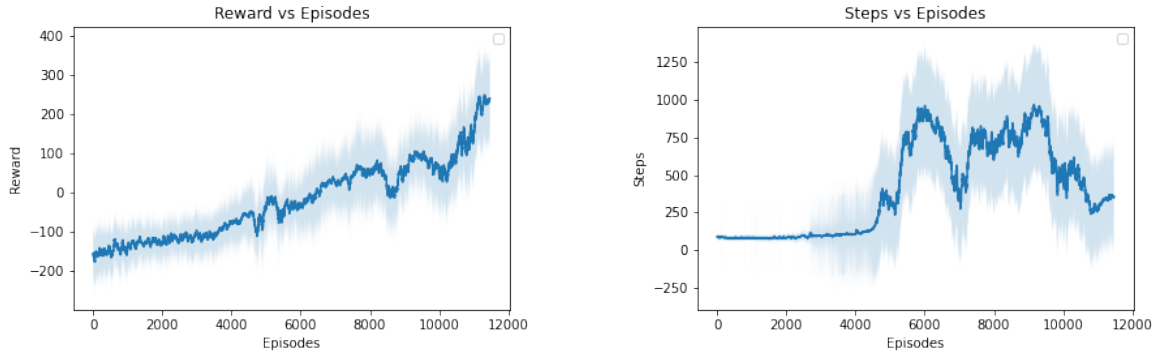


Figure 4: Results from the best training run of REINFORCE with Baseline on the Lunar Lander environment.

engine in order to reduce the amount of negative reward. The episode length increases again as the agent tries to land on the landing pad. This is also visible in the policy loss chart in Figure 5. These step can more clearly be seen in the best run shown in Figure 4. Some of the runs were not able to completely solve the environment, but this is an example of a training run where the agent was able to solve it.

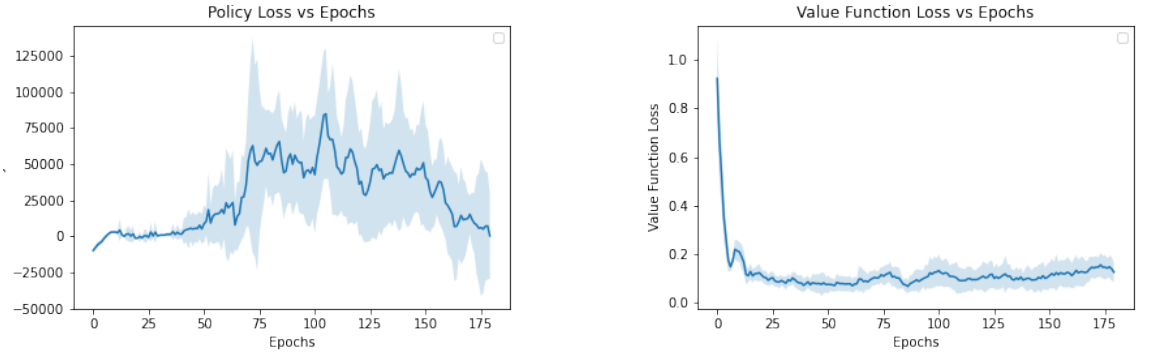


Figure 5: Training Loss over ten training runs of REINFORCE with Baseline on the Lunar Lander environment.

Breakout

Using a small convolutional neural network each for the policy and value function parameterizations, REINFORCE with Baseline was able to achieve a reward of 1 on breakout. The rewards given in breakout can be very sparse and there is a large state space to explore. The policy network would completely converge before the agent was able to score any higher than 1 reward. Getting to this reward took many episodes and exemplifies the difficulty of tuning REINFORCE on larger state spaces and sparse reward functions.

3 Proximal Policy Optimization

In this section we will describe the details of a popular policy gradient estimator called Proximal Policy Optimization(PPO) [6]. The policy gradients methods are popular in reinforcement learning because they can be used with non linear function approximators like neural networks and optimize for achieving high rewards. But the vanilla policy gradient estimates have high variance and a high sample complexity. The policy updates are often not steady and stable and they largely depends on the trajectory that is being used to estimate the policy gradient. There are several methods to improve the sample complexity and reduce the variance of the policy gradient estimators. As described in [5], the general form of policy gradient can be expressed as equation 1

$$\hat{g} = \mathbf{E}\Big[\sum_{t=0}^{\infty} \mathbf{\Psi}_t \nabla_{\theta} \log \pi_\theta(s_t, a_t)] \tag{1}$$

The function $\mathbf{\Psi}_t$ can be substituted with

1. $G_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_k$ (discounted returns)
2. $\hat{Q}^\pi(s_t, a_t)$ (Action-value function)
3. $\hat{\delta}_t = r_t + \gamma \hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s_t)$ (TD error)
4. $\hat{A}_t(s_t, a_t) = \hat{Q}^\pi(s_t, a_t) - \hat{V}^\pi(s_t, a_t)$ (advantage)

Generalized Advantage Estimator

Before diving into the PPO algorithm, we will briefly describe the Generalized Advantage Estimators[5] that are used for estimating the policy gradients for PPO and many other policy gradient algorithms. These estimators reduce the variance of policy gradients estimators but induces some degree of bias.

For the policy gradient methods, the advantage function $\hat{A}_t(s_t, a_t)$ gives the lowest variance for the policy gradient estimates for equation 1. The advantage function calculates difference between the action-value function and the state value function for a state-action pair at time t. It denotes whether the action a_t is better or worse than the current value estimate of the state s_t. The policy gradient with the $\mathbf{\Psi}_t = \hat{A}_t(s_t, a_t)$ calculates the policy gradient and takes the step that "increases the probability of the better than average actions and reduce the probability of worse than the policy's default behaviour" [5].

Now let's consider the TD error $\delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$. Note that δ_t is an estimate of the advantage function $\hat{A}_t(s_t, a_t)$. Let's call this estimate $\hat{A}_t^{(1)} = \delta_t$ and sum of k of the TD error terms as $\hat{A}_t^{(k)}$.

$$\hat{A}_t^{(1)} \coloneqq \delta_t = -\hat{V}(s_t) + r_t + \gamma \hat{V}(s_{t+1}) \tag{2}$$

$$\hat{A}_t^{(2)} \coloneqq \delta_t + \gamma \delta_{t+1} = -\hat{V}(s_t) + r_t + \gamma r_{t+1} + \gamma^2 \hat{V}(s_{t+2}) \tag{3}$$

$$\hat{A}_t^{(3)} \coloneqq \delta_t + \gamma \delta_{t+1} + \gamma^2 \delta_{t+2} = -\hat{V}(s_t) + r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 \hat{V}(s_{t+3}) \tag{4}$$

$$\tag{5}$$

$$\hat{A}_t^{(k)} \coloneqq \sum_{l=0}^{k-1} \gamma^l \delta_{t+l} = -\hat{V}(s_t) + r_t + \gamma r_{t+1} + \ldots \gamma^{k-1} r_{t+k-1} + \gamma^k \hat{V}(s_{t+k}) \tag{6}$$

Note that for $k = 1$, the advantage estimate has highest bias and as $k \to \infty$, the advantage estimate becomes $\hat{A}_t^{(\infty)} = \hat{V}(s_t) + \sum_{l=0}^{\infty} \gamma^l r_l$, which has the lowest bias. The Generalized Advantage Estimator is then defined

as the exponentially weighted average of $\hat{A}_t^{(k)}$.

$$\hat{A}_t^{GAE(\gamma,\lambda)} := (1-\lambda)\Big(\hat{A}_t^{(1)} + \lambda\hat{A}_t^{(2)} + \lambda^2\hat{A}_t^{(3)} + \dots\Big) \tag{7}$$

$$= \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l} \tag{8}$$

$$GAE(\gamma,0) = \hat{A}_t^{(1)}$$
$$GAE(\gamma,1) = \hat{A}_t^{(\infty)}$$

We can see that GAE$(\gamma,0)$ has the lowest variance but a high bias whereas GAE$(\gamma,1)$ has a high variance and low bias. The parameter λ, $(0<\lambda<1)$ controls the bias-variance trade-off in GAE(γ,λ).

Trust Region Methods and PPO

As we discussed, the policy gradient methods described in equation 1 are unstable and could lead to either very large updates for some states or very low updates some other states. Also, from the equation 1, we can see that we have to sample a complete episode before updating our policy. This procedure is very slow and leads to sample inefficiency. However, to address the issue of sample efficiency, trust region methods use a "surrogate" objective function [7, 6] that can be updated using the trajectories of an old policy and new trajectories are sampled fairly regularly after updating the old policy for some iterations. To avoid any drastic updates in policy the trust region methods apply a constraint on how much a policy can be updated in a single iteration. For example, Trust Region Policy Optimization(TRPO) [7] uses a hard constraint in terms of maximum KL-Divergence between the old policy and the new policy. PPO applies a much simpler constraint on the ratio of the new policy and old policy by clipping the surrogate objective function. Let's consider θ as the policy parameters and define $r_t(\theta) = \frac{\pi_\theta(s_t,a_t)}{\pi_{\theta_{old}}(s_t,a_t)}$ which the ratio of the current policy and some old policy at time step t. The "surrogate" objective function is defined as

$$L(\theta) = \hat{E}\Big[\frac{\pi_\theta(s_t,a_t)}{\pi_{\theta_{old}}(s_t,a_t)}\hat{A}_t\Big] = \hat{E}[r_t(\theta)\hat{A}_t] \tag{9}$$

PPO penalizes the objective function of equation 9 when $r_t(\theta)$ moves away from 1 by a large amount by clipping the value of $r_t(\theta)$. More specifically PPO optimizes the clipped surrogate function.

$$L_{PPO}(\theta) = \hat{E}\Big[\min\Big(r_t(\theta)\hat{A}_t, clip(r_t(\theta),1-\epsilon,1+\epsilon)\hat{A}_t\Big)\Big] \tag{10}$$

where ϵ is the hyperparameter that constraint the value of $r_t(\theta)$ within $[1-\epsilon,1+\epsilon]$. PPO takes the minimum value of the clipped and unclipped surrogate function for updating the policy with small steps that leads to a stable policy gradient algorithm. The policy π_θ is implemented using neural networks with weights θ and therefore, the gradients of L_{PPO} can be calculated using the autograd methods in PyTorch.

The advantages, $\hat{A}_t$ used in $L_{(PPO)}$, are the truncated Generalized Advantage Estimators(GAE). They are calculated by estimating the value function using a neural network function approximator. The truncated GAE calculate the value of $\hat{A}_t$ within T timesteps, where T is much less than the episode length.

$$\hat{A}_t = -V(s_t) + \delta_t + \gamma\lambda\delta_{t+1} + \dots + (\gamma\lambda)^{T-t+1}\delta_{T-1}, \tag{11}$$

$$\delta_t = r_t + \gamma\hat{V}_w(s_{t+1}) - \hat{V}_w(s_t) \tag{12}$$

In the above equations, $\hat{V}_w$ represents the value function neural network called critic network, with parameters w. The target values for updating the critic network are the discounted returns for T time steps.

$$V_t^{target}(s_t) = r_t + \gamma r_{t+1} + \dots + \gamma^{T-t}r_T \tag{13}$$

The critic network is updated by minimizing the mean squared error between the target and the predicted values called $L^{critic}(w)$.

$$L_t^{critic}(w) = \frac{1}{2}\Big(V_t^{target}(s_t) - V_w(s_t)\Big)^2 \tag{14}$$

Furthermore, to ensure sufficient exploration, PPO also uses an entropy objective with called $S_\theta(s_t)$. Therefore, the loss function for updating the policy and backpropagating the gradients throught the neural network can be defined as

$$\mathcal{L}_t(\theta) = -L_{(PPO)}(\theta) - cS_\theta(s_t) \tag{15}$$

In the above equation c is a small coefficient (≈ 0.01) to control the degree of exploration. Minimizing the loss function of equation 15 using automatic differentiation libraries of PyTorch with gradient descent methods will increase the surrogate objective in equation 10 and also encourage some degree of exploration by increasing the entropy of the current policy so that the agent does not get stuck in the suboptimal solution.

Algorithm

Algorithm 2 PPO-Clip

Input: γ, λ, ϵ, c π_θ, V_w
Initialize the policy parameters θ, and the value function parameters w randomly.
for iteration $= 1, 2, \ldots$ **do**
  for $n = 1, 2, \ldots, N$ **do**
    Run $\pi_{\theta_{old}}$ for T timesteps and collect $\{(s_t, a_t, r_t)\}_{t=0}^{T}$
    Compute advantage estimates $\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_T$ using equation 11
    Compute the estimated returns $\hat{R}_1, \hat{R}_2, \ldots, \hat{R}_T$ using equation 13
    Store the tuples of $(s_t, a_t, \hat{R}_t, \hat{A}_t)$ in a memory buffer.
  end for
  for $k = 1, 2, \ldots, K$ epochs **do**
    Sample $M(\leq NT)$ tuples from the buffer and calculate the following objective functions.

$$r_m(\theta) = \frac{\pi_\theta(s_m, a_m)}{\pi_{\theta_{old}}(s_m, a_m)}$$

$$\hat{L}_{PPO}(\theta) = \frac{1}{M}\sum_{m}^{M} \min\Big(r_m(\theta)\hat{A}_m, \text{clip}(r_m(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_m\Big)$$

$$\hat{S}(\theta) = -\frac{1}{M}\sum_{m=1}^{M} \pi_\theta(s_m, a_m) \log \pi_\theta(s_m, a_m)$$

$$\hat{L}^{critic}(w) = \frac{1}{M}\sum_{m=1}^{M} \frac{1}{2}\Big(V_w(s_m) - \hat{R}_m\Big)^2$$

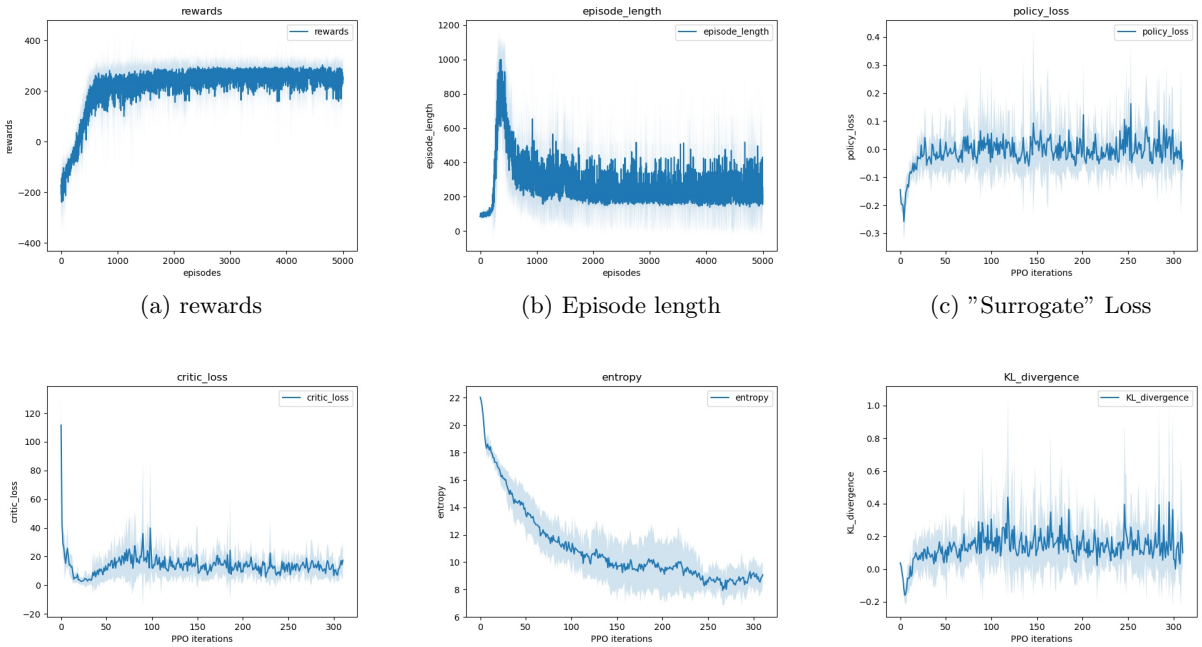
    Minimize $\hat{\mathcal{L}}(\theta) = -\hat{L}_{PPO}(\theta) - c\hat{S}(\theta)$ and $\hat{L}^{critic}(w)$ using Stochastic Gradient Descent with Adam.
  end for
end for

Experiments and Results

PPO was trained on two different domains called Lunar Lander and Cartpole as the part of the project. We also trained PPO on the Atari Breakout domain for extra credits. In this section, we will describe the details of the neural network architectures that are used to learn the policies and value functions for each of the domains separately. Lunar Lander and Cartpole domains use the same neural network design for both the policy and the value function. For Breakout domain, the neural networks for policy and value function contain convolutional layers for processing the RGB inputs. Despite the different types of feature extraction layers of the neural networks, all the domains have discrete action space. Lunar Lander and Breakout has 4 actions whereas Cartpole has 2 actions. Full code and video can be found https://github.com/gargsid/Proximal-Policy-Optimization.

Lunar Lander	Cartpole	Breakout
Input - 8	Input - 8	Input - (4,84, 84)
FC + ReLU - 128	FC + ReLU - 128	Conv, C=32, K=8, S=4
FC + ReLU - 128	FC + ReLU - 128	Conv, C=64, K=4, S=2
FC + ReLU - 128	FC + ReLU - 128	Conv, C=64, K=3, S=1
		FC-512
FC(logits)-4, FC(value)-1	FC(logits)-2, FC(value)-1	FC(logits)-4, FC(value)-1

Table 1: The above table describes the details of the neural network design for the policy and the value functions in case of PPO. Separate actor(policy) and critic(value function) networks are used and they share the same feature extraction design. The architecture design gets different in the last row only where the policy network output multidimensional logits vector whereas the value function network outputs a single value estimate of the input state. FC: Fully-connected layers, C: channels, K: kernel size, S: stride.



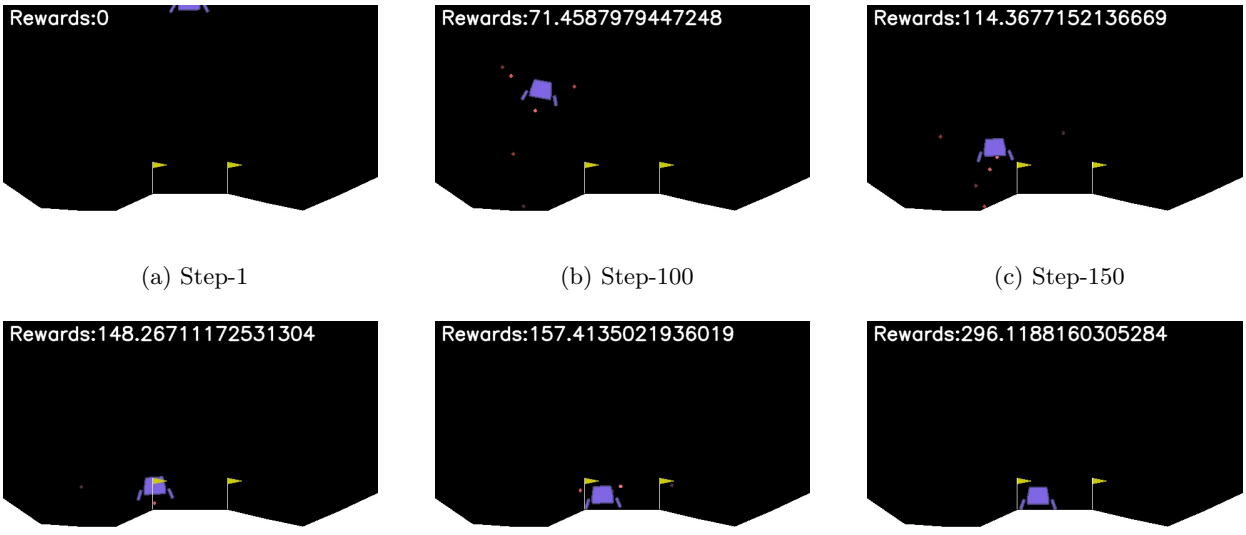
(a) rewards (b) Episode length (c) "Surrogate" Loss

(d) Critic Loss (e) Entropy (f) KL-Divergence

Figure 6: PPO results on Lunar Lander domain. Note the average rewards in fig.(a) that we achieved for this domain is around 300 with PPO. The PPO algorithm was run 10 times for 1000 episodes each.

Neural Network Architecture details for Policy and Value Functions

For the Lunar Lander and Cartpole domains, we trained two separate neural networks with three fully connected layers with ReLU [1] for modeling the policy and value function. The policy network outputs a fixed length vector of logits depending on the action space of the domain. The logits are normalized using softmax function to get the probability values between 0 and 1. The policy is then constructed using the Categorical distribution, from which the actions are sampled while running the PPO algorithm. The value function network contains the same design as the policy network for processing the input features but outputs only a single value for the value function estimate of the current state. Both the networks are optimized using Adam optimizer with learning rate of 10^{-3} and clip value, $\epsilon = 0.2$. The feature extraction layers of the Breakout consists of three convolutional layers and a fully connected layer with ReLU activation functions. The details of the feature extraction layers for all the three domains are given in the Table 1.



(a) Step-1 (b) Step-100 (c) Step-150

(d) Step-200 (e) Step-250 (f) Step-298

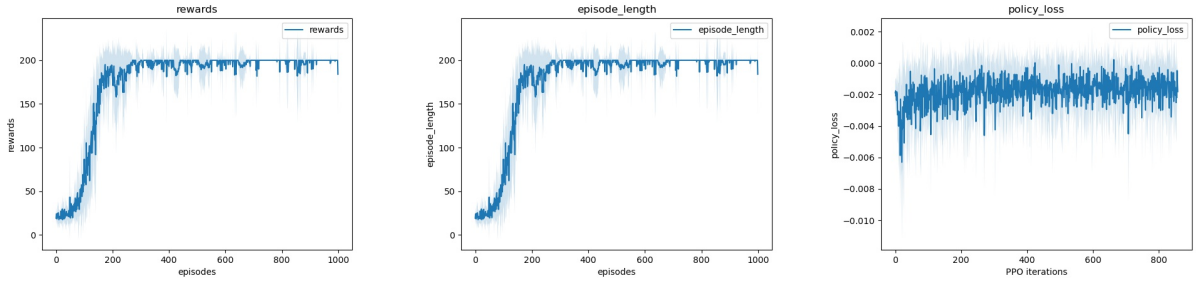
Figure 7: The figure consists of the states from a 298 step episode run with the trained policy with PPO. In figure (b) the agent fires the engine to come closer to the landing area and accumulates the reward in figure (c). In the final figure (f) we can see that the agent successfully landed between the flags and the final accumulated reward is 296.12. Full code and video can be found here.

Processing States for Breakout

The Breakout OpenAI gym environment outputs an RGB image of height 210 and width 160. The image frame represents the current state of the game as it is a snapshot of the position of the ball, paddle, and the remaining bricks. However, this state does not consist enough information to model this game as a Markov Decision Process(MDP) because based on the snapshot of the game the agent cannot decide how to move the paddle. The agent needs to know in which direction the ball is going to make the decision. Therefore, to make this as an MDP we take a same action four times to get four different snapshot of the game after every action. During the execution of the four actions the ball has moved in some direction. Now, to make this state as an MDP the idea is to stack the four frames on top of each other to create a new state and then give this state as the input to the agent. Note that this state consists of enough information for the agent to take an action which leads to a high reward. To further reduce the complexity of inputs, the RGB frame is converted to grayscale images and downsampled from (210, 160, 3) to (84, 84) image. The four frames are then stacked together to give $(4, 84, 84)$ dimensional state. To create the next state for the breakout, the sampled action is again repeated 4 times and the next state is then created by removing the first grayscale frame from the current state and appending the new grayscale frame to the current state at the end. The action space of the game consists of four discrete action - do nothing, fire(to start the game), move left, move right. The game does not start automatically so, after resetting, we execute the action fire to start the game so that the agent does not have to learn to do it.

Hyperparameters Values of PPO

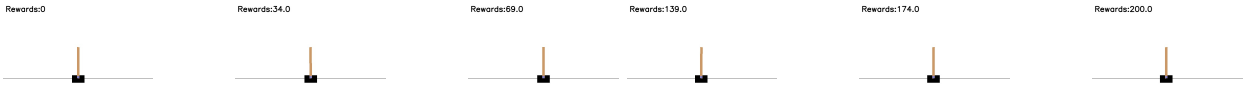
The value of hyperparameters were referenced from the PPO paper [6] and the TRPO paper [7] and then fine-tuned to fit the Lunar Lander and Cartpole domains. Specifically, the value of $\gamma = 0.99$, $\lambda = 0.95$ was kept constant for all the experiments. Also, from our experiments we saw that the large number of episodes and large values of T increases the convergence rates of the algorithm. Specifically, for Lunar Lander and Cartpole, most of the configurations were same. We sampled minimum of $N = 10$ trajectories for each iteration and we set the value of $T = 1024$. This value of T for both of the domains is more than the



(a) rewards (b) Episode length (c) "Surrogate" Loss

(d) Critic Loss (e) Entropy (f) KL-Divergence

Figure 8: PPO results on Cartpole domain. Note the average rewards that we achieved for this domain is 200 with PPO. Since the reward function in Cartpole is the number of steps in the episode the graphs (a) and (b) are same. The PPO algorithm was run 10 times on cartpole for 1000 episodes for each run. Full code and video can be found here.

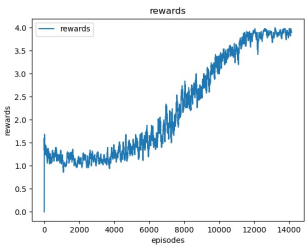


(a) Step-1, $R = 0$ (b) Step-35, $R = 34$ (c) Step-70, $R = 64$ (d) Step-140, $R = 139$ (e) Step-175, $R = 174$ (f) Step-200, $R = 200$

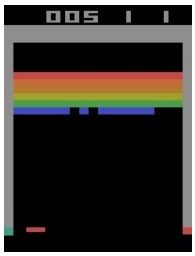
Figure 9: The figure consists of the states from a 200 step episode run with the trained policy with PPO on Cartpole domain. The agent is successfully able to maintain the balance of the pole and collects the maximum reward of 200. Full code and video can be found here.

maximum episode length and therefore the agent samples the whole episode before updating the policy. For the hidden dimensions of the policy and value function networks, we experimented with $h = 64, 128, 256$ among which the $h = 128$, resulted in optimal results. We used Adam [4] optimizer with learning rate of 10^{-3} without any decay. The value of ϵ was set to 0.2. The values of $M = 256$ and $K = 4$ were constant for all the experiments.

The Breakout domain has a separate of hyperparameters. In the PPO [6] paper, the authors have used a clip value of $\epsilon = 0.1$ and learning rate as 2.5×10^{-4} with linear decay rate applied to both the parameters. In our experiments we used the same value of learning rate and set the value of $\epsilon = 0.2$. We fixed the maximum number of iterations of PPO algorithm to 10^5 and apply linear decay rate on both the learning rate and ϵ. The episode length of a trajectory for Breakout is also considerably bigger than the Cartpole domain and we saw that high values of T like $T = 2048$, and $T = 4096$ results in learning of the agent. Unfortunately, we were only able to train the agent to get a reward of 5. The graph of which is shown in figure 10



(a) Rewards



(b) Test Rewards

Figure 10: The figure consists of the learning statistics of the PPO trained agent on Breakout domain. From fig. (a) we can see that currently the agent is able to collect a cumulative reward of 4 in every episode without losing any life. In fig (b) we can see that the agent has collected the reward of 5 after 5 lives during test time. The agent is stuck at the left side always. Full code and video can be found here.

4 Sarsa(λ)

Sarsa(λ) was implemented as an extra credit third algorithm. Sarsa(λ) is an extension of the TD(λ) algorithm. Are both reliant on the use of decaying eligibility traces called the λ return. Eligibility traces allow for algorithms to operate in between the extremes of Monte Carlo methods and one step methods. The benefit of eligibility traces over n-step methods is that it allows for decaying weights to be applied and for a computational advantage.

The main hurdle to overcome is handling a continuous state with Sarsa(λ). A common approach to achieving this is through the use of tile coding [8]. Creating a tile involves partitioning the state space into different partitions and assigning each of those partitions a different index. In order to gain more fine-tuned state representations, a tile coding consists of multiple tilings set apart by some offset. The state is then represented by the index of each partition that is active in each tiling. The use of a tile coding translates a continuous state space into a discrete one. This tile coding can be used as the function $\mathcal{F}$ that is required in the pseudocode for Sarsa(λ) 3.

Implementation Details

The state is translated to indices by 10 tilings of 10 bins each offset by a uniform distribution of 1/10 of their each respective feature's range. Actions are selected according to a softmax policy over the state action value function.

Cart Pole

For Cart Pole, the Sarsa agent was trained with a learning rate of .25, trace decay of .9, and a reward discount of .99. As can be seen in Figure 11, the algorithm converges in a very small number of episodes albeit to a non optimal score. The reason that the agent does not improve past a score of 150 is that it doesn't learn to avoid the edge of the designated area. This may be due to a non optimal tiling implementation making the edge of the designated area difficult to discern in the parameterized functions.

Other Environments

Due to the much larger state spaces of both Lunar Lander and Breakout, it was not feasible to learn in these environments using this implementation of Sarsa(λ). The tabular tiling method grows exponentially with the dimensions of the state vector and the memory requirements of this implementation were the limiting factor of testing this algorithm in more complex environments.

Algorithm 3 Sarsa(λ)[8]

Input: function $\mathcal{F}(s,a)$ that returns the indices of active features
Parameters: step size $\alpha > 0$, trace decay $\lambda \in [0,1]$
Initialize: $\boldsymbol{w} \in \mathbb{R}^d$ //State Action parameterization
Initialize: $\boldsymbol{z} \in \mathbb{R}^d$ //Trace Parameterization
while True **do**
Initialize: $S \leftarrow$ initial state
Choose A $\sim \pi(S)$
$\boldsymbol{z} \leftarrow \mathbf{0}$
for Each step of Episode **do**
Take action A, observe R, S'
$\delta \leftarrow R$
for i in $\mathcal{F}(S,A)$ **do**
$\delta \leftarrow \delta - \boldsymbol{w}_i$
$\boldsymbol{z}_i \leftarrow 1$ //Replacing Traces
end for
if S' is terminal **then**:
$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\delta\boldsymbol{z}$
Move to next episode
end if
Choose $A' \sim \pi(S')$
for $i \in \mathcal{F}(S',A')$ **do**
$\delta \leftarrow \delta - \gamma\boldsymbol{w_i}$
end for
$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\delta\boldsymbol{z}$
$\boldsymbol{z} \leftarrow \gamma\lambda\boldsymbol{z}$
$S \leftarrow S'; A \leftarrow A'$
end for
end while

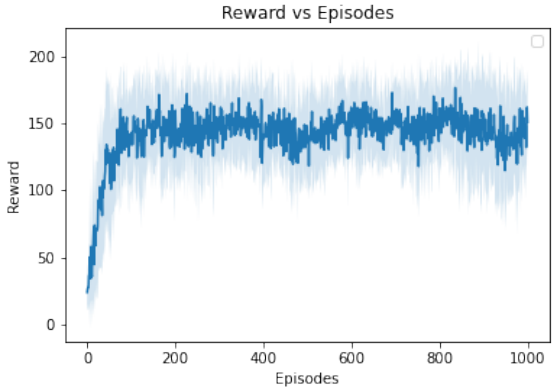
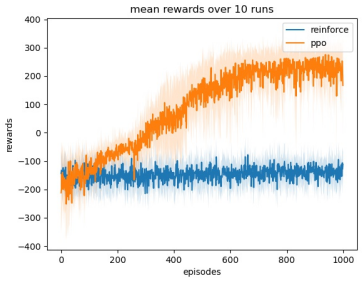
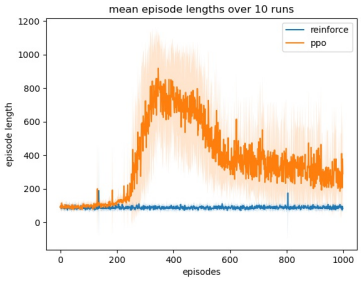


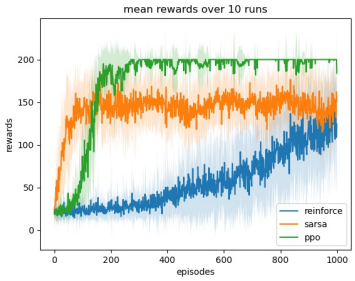
Figure 11: Results over ten training runs of Sarsa(λ) on the Cart Pole environment.



(a) Rewards collected on Lunar Lander



(b) Episode length for Lunar Lander



(c) Rewards collected on Cartpole

Figure 12: The figure shows the comparison of different RL algorithms on Lunar Lander and Cartpole domains. In figures (a), and (b) we can see that the PPO (orange) performs better than REINFORCE with baselines on Lunar Lander Environment. From figure (c) we can see PPO (green) has the best performance when compared with REINFORCE, and Sarsa (λ).

5 Discussion and Comparison

Overall, PPO offered the most robust performance out of the three algorithms by a large margin. On Cart Pole, Sarsa(λ) converged faster but it did not converge to the optimal solution. In all cases REINFORCE with Baseline converged the slowest and was never able to out perform PPO. Charts summarizing these results can be seen in Figure 12. On Lunar Lander, in the number of episodes that PPO was able to solve the environment, REINFORCE with baseline was barely able to make any progress at all.

Summary Of Contributions

Andrew implemented REINFORCE with baseline and Sarsa(λ) while Siddhant implemented Proximal Policy Optimization.

References

[1] Abien Fred Agarap. "Deep learning using rectified linear units (relu)". **in***arXiv preprint arXiv:1803.08375*: (2018).

[2] M. G. Bellemare **andothers**. "The Arcade Learning Environment: An Evaluation Platform for General Agents". **in***Journal of Artificial Intelligence Research*: 47 (**june** 2013), **pages** 253–279.

[3] Greg Brockman **andothers**. *OpenAI Gym*. 2016. arXiv: `1606.01540 [cs.LG]`.

[4] Diederik P. Kingma **and** Jimmy Ba. "Adam: A Method for Stochastic Optimization". **in***CoRR*: abs/1412.6980 (2015).

[5] John Schulman **andothers**. "High-dimensional continuous control using generalized advantage estimation". **in***arXiv preprint arXiv:1506.02438*: (2015).

[6] John Schulman **andothers**. "Proximal policy optimization algorithms". **in***arXiv preprint arXiv:1707.06347*: (2017).

[7] John Schulman **andothers**. "Trust region policy optimization". **in***International conference on machine learning*: PMLR. 2015, **pages** 1889–1897.

[8] Richard S Sutton **and** Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[9] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". **in***Machine Learning*: 8 (2004), **pages** 229–256.