Analysis of Actor-Critic Algorithms And it's Variants *

Abhishek R
Computer Science Department
Indian Institute of Technology Dharwad
160020036@iitdh.ac.in

Mentor: Prabuchandran K J
Assistant Professor
Indian Institute of Technology Dharwad
prabukj@iitdh.ac.in

June 11, 2020

Abstract

Our aim was to build a method which combines value based and policy based methods, also known as Actor-Critic methods and implement it on OpenAI gym environments like CartPole, Acrobot, Lunar Lander and Mountain Car, and atari games. We went through policy based methods and saw how the actor-critic comes in. Then we implemented different variants of it. Lastly, We combined the idea from konda's paper for Actor-Critic . We gave the results for few of the algorithms mentioned at the end and compared the results. Our main goal was to make the agent learn even if it means to run the code for longer iterations.

Keywords: *Reinforcement Learning - RL,Actor-Critic, Value Function, Q-Value;*

1 Introduction

Reinforcement learning is one of the three basic area of machine learning, alongside supervised learning and unsupervised learning. Reinforcement learning is concerned with how the agent in an unknown environment obtains some reward by interacting with it. The agent ought to take actions so as to maximize cumulative rewards.

There are two main types of RL methods:

- **Value Based** Here one tries to approximate the optimal Value function. Each state is associated with a value function predicting the expected amount of future rewards we are able to receive in this state by acting the corresponding policy. Higher the value, the better the action. The most famous algorithm is Q learning and all its enhancements like Deep Q Networks, Double Dueling Q Networks, etc
- **Policy Based** Policy Based algorithms like Policy Gradients and REINFORCE try to find the optimal policy directly.

Each method has their advantages. While Policy based are:

- Effective in high-dimensional or continuous action spaces. When the space is large, the usage of memory and computation consumption grows rapidly. The policy based RL avoids this because the objective is to learn a set of parameters that is far less than the space count.

*B.Tech Project Report

- Can learn stochastic policies
- Better convergence properties

Policy based typically converge to local rather then global optimum. Evaluating a policy is typically inefficient and high variance. Policy based RL has high variance, but there are techniques to reduce this variance. They are sensitive to the choice of step size - too small, and progress is hopelessly slow; too large and the signal is overwhelmed by the noise, or one might see catastrophic drops in performance.
While Value based are:

- more sample efficient than policy optimization(usually on-policy refer 2.1) methods as they explore more and are usually off-policy (refer 2.2).
- But they are less stable and suffer from poor convergence, because they learn an state-action value(or just state value) function approximation (refer 2.4 and 2.5), which is then used to find a corresponding policy.
- Function approximation, combined with bootstrapping(using last predicted Q value to update the present) and off-policy data makes these methods less reliable.

To reap the benefit of both the algorithms we use Actor-Critic methods at there intersection, where our goal is to optimize both the policy and the Value/Q function.

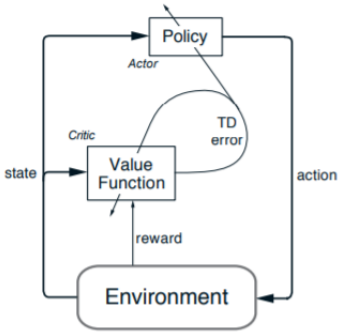


Figure 1: (Image source: Sutton and Barto (2017).)

Actor: decides which action to take
Critic: tells the actor how good its action was and how to it should adjust

2 Background

Let's go through some of the key concepts that we need to understand before going through the algorithms.

2.1 On-Policy

We use a policy to determine which actions to take. In on-policy learning, we optimize the current policy and use it to determine what actions to explore next. Since the current policy is not optimized or sub-optimized, it still allows some form of exploration.

2.2 Off-Policy

Off-policy learning allows a second policy, like the ϵ-greedy policy, to explicitly define the exploration it wants. Off-policy learning optimizes the current policy but use another policy to explore actions. Off-policy provides greater choice but slower to converge and have higher complexity.

2.3 Markov Decision Process

Almost all the RL problems can be framed as Markov Decision Processes (MDPs). All states in MDP has "Markov" property, referring to the fact that the future only depends on the current state, not the history.

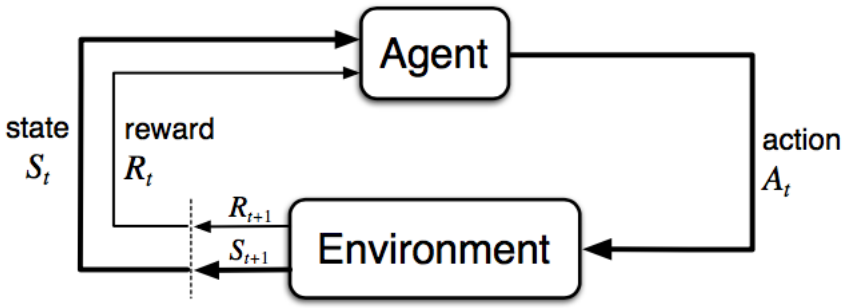


Figure 2: The agent-environment interaction in a Markov decision process. (Image source: Sutton and Barto (2017).)

MDP consists of a tuple of 5 elements:

- **S**, Set of states: At each time step the state of the environment the agent is in.
- **A** Set of actions: At each time step the agent chooses an action $\mathbf{a} \in \mathbf{A}$ to perform
- $\mathbf{p}(\mathbf{s_{t+1}}|\mathbf{s_t}, \mathbf{a_t})$, Transition probability function: State transition model that describes how the environment state changes when the user performs an action **a** depending on the action **a** and the current state **s**.
- $\mathbf{R}(\mathbf{r_{t+1}}|\mathbf{s_t}, \mathbf{a_t})$, Reward function that describes the real-valued reward value that the agent receives from the environment after performing an action. In MDP the the reward value depends on the current state and the action performed.
- Discount factor that controls the importance of future rewards.

2.4 State-action Value (Q Value)

State-action Value is a measure of the overall expected reward assuming the Agent is in state **s** and performs action **a**, and then continues playing until the end of the episode

following some policy π.

$$Q_\pi(s,a) = E_\pi[\sum_{t=0}^{t=T-1} \gamma^t r_t | S = s, A = a] \tag{1}$$

Where T is the number of states from state s till the terminal state or goal state. γ is discount factor. r_0 is the immediate reward received after performing action a_0 in state s_0. The discounting factor penalize the rewards in the future, it helps because:

- The future rewards may have higher uncertainty
- The future rewards do not provide immediate benefits

2.5 State Value Function

The Value function V(s), represent how good is a state for an agent to be in. It is equal to expected reward for an agent starting from state s. The value function depends on the policy by which the agent picks actions to perform. So, if the agent uses a given policy π to select actions, the corresponding value function is given by:

$$V_\pi(s) = E_\pi[\sum_{t=0}^{t=T-1} \gamma^t r_t | S = s] \tag{2}$$

Where T is the number of states from state s till the terminal state or goal state. γ is discount factor. r_0 is the expected reward from just being in state s, before any action was played, while in Q Value r_0 is the expected reward after a certain action was played.

2.6 Monte-Carlo Methods

In Monte Carlo (MC) we play an episode of the game starting by some random state (not necessarily the beginning) till the end and we record the states, actions and rewards that we encountered and then compute the V(s) and Q(s,a) for each state we passed through.
In Monte Carlo there is no guarantee that we will visit all the possible states and another weakness of this method is that we need to wait until the game ends to be able to update our V(s) and Q(s,a), this is problematic in games that never ends.

3 Actor Critic Algorithms

First we go through some of the well known policy based method such as REINFORCE and work our way through to understand how we can extent this to get Actor Critic Algorithm and it's variants. We have used different implementation techniques to solve some of the few variants and tried new set of variants.

3.1 Policy Gradient

In Policy Gradient we directly estimate the optimal policy using neural networks.
The main objective we need to achieve is to maximize expectation of Reward for a policy.
The main intuition is to collect a bunch of trajectories

- Making the good trajectories more probable
- Making the good actions more probable - which we will see in actor-critic methods

In Policy Gradient based reinforcement learning, the objective function which we are trying to maximise is the following:

$$J(\theta) = E_{\pi_\theta}[R] \tag{3}$$

Where R is the total reward in a trajectory.
We are attempting to find a policy(π) with parameters(θ) which maximises the expected Value to the total rewards of an agent in an environment. So we want to learn the optimal θ. We use gradient based search for θ by updating θ in following way:

$$\theta \leftarrow \theta + \alpha \nabla J(\theta) \tag{4}$$

To calculate Gradient of J(θ). Let's first look at the score function gradient estimator.

3.1.1 Score Function Gradient Estimator

The score function is the partial derivative of the log-likelihood function.
Consider an expectation $E_{x\sim p(x|\theta)}[f(x)]$ and we want to compute the gradient with respect to θ.

$$\begin{aligned}
\nabla_\theta E_x[f(x)] &= \nabla_\theta \int p(x|\theta) f(x) dx \\
&= \int \nabla_\theta p(x|\theta) f(x) dx \\
&= \int p(x|\theta) \frac{\nabla_\theta p(x|\theta)}{p(x|\theta)} f(x) dx \\
&= \int p(x|\theta) \nabla_\theta \log(p(x|\theta)) f(x) dx \\
&= E_x[f(x) \nabla_\theta \log p(x|\theta)]
\end{aligned} \tag{5}$$

So we just need to sample $x \sim p(x|\theta)$ and compute $\hat{g_i} = f(x_i)\nabla_\theta \log p(x_i|\theta)$.Now we can compute and differentiate density $p(x|\theta)$ wrt θ.
Intuition behind this gradient estimator

$$\hat{g_i} = f(x_i)\nabla_\theta \log p(x_i|\theta) \tag{6}$$

- Here f(x) measures how well the sample x is.

- Moving in the direction $\hat{g}_i$ pushes up the logprob of the sample, in proportion to how good it is.
- This is valid even if f(x) is discontinuous or sample space (containing x) is a discrete set.

3.1.2 Score Function Gradient Estimator for Policies

Now random variable x in the above equation is a whole trajectory of the agent in an environment.
$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, ..., s_{T-1}, a_{T-1}, r_{T-1}, s_T)$

$$\nabla_\theta E_\tau[R(\tau)] = E_\tau[\nabla_\theta \log p(\tau|\theta) R(\tau)] \tag{7}$$

We can write $p(\tau|\theta)$ as:

$$p(\tau|\theta) = \mu(s_0) \prod_{t=0}^{T-1} [\pi(a_t|s_t, \theta) P(s_{t+1}, r_t|s_t, a_t)]$$
$$\log p(\tau|\theta) = \log \mu(s_0) + \sum_{t=0}^{T-1} [\log \pi(a_t|s_t, \theta) + \log P(s_{s+t}, r_t|s_t, a_t)]$$
$$\nabla_\theta \log p(\tau|\theta) = \nabla_\theta \sum_{t=0}^{T-1} \log \pi(a_t|s_t, \theta)$$

So, the final expression becomes as:

$$\nabla_\theta E_\tau[R] = E_\tau[R \nabla_\theta \sum_{t=0}^{T-1} \log \pi(a_t|s_t, \theta)]$$

So final by inputting the expression for total reward we get

$$\boxed{\nabla_\theta E_\tau[R] = E_\tau[(\sum_{t=0}^{T-1} r_t) \nabla_\theta \sum_{t=0}^{T-1} \log \pi(a_t|s_t, \theta)]} \tag{8}$$

We can still reduce the expression, and also reduce the variance of this estimator by noting that for t′ < t

$$E_\tau[r_t \nabla_\theta \log \pi_\theta(a_t|s_t)] = 0$$

Because the gradients of logprob of future states don't affect the present state. So by taking the gradient estimator for one reward term, we get:

$$\nabla_\theta E_\tau[R] = E_\tau[r_{t'} \sum_{t=0}^{t'} \nabla_\theta \log \pi(a_t|s_t, \theta)]$$

Taking sum over all reward terms we get

$$\nabla_\theta E_\tau[R] = E_\tau[(\sum_{t=0}^{T-1} r_{t'}) \sum_{t=0}^{t'} \nabla_\theta \log \pi(a_t|s_t, \theta)]$$

By rearranging terms, we get each logprob multiplied by sum of future reward term

$$\boxed{\nabla_\theta E_\tau[R] = E_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t, \theta) \sum_{t'=t}^{T-1} r_{t'}]} \tag{9}$$

We can replace the future reward sum with discounted reward sum:

$$\boxed{\nabla_\theta E_\tau[R] = E_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t, \theta) \sum_{t'=t}^{T-1} \gamma^{t-t'} r_{t'}]} \tag{10}$$

Where γ is a discount factor, that tells how important future rewards are to the current state. Discount factor is a value between 0 and 1. A reward R that occurs t steps in the future from the current state, is multiplied by γ^t to describe its importance to the current state.

These two equations 9 and 10 can be called as **vanilla policy gradient**, generalized as:

$$\boxed{\nabla_\theta E_\tau[R] = E_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t, \theta) G_t]} \tag{11}$$

3.2 REINFORCE

REINFORCE, also known as Monte-Carlo policy gradient, relies on an estimated return by Monte-Carlo methods using episode samples to update the policy parameter. REINFORCE works because the expectation of the sample gradient is equal to the actual gradient.

Therefore we are able to measure from real sample trajectories and use that to update our policy gradient. It relies on a full trajectory and that's why it is a Monte-Carlo method.

Algorithm 1 REINFORCE Algorithm

Algorithm parameters: learning rate α, discount factor γ

Initialize θ arbitrarily

for *each episode* $\{s_0, a_0, r_0, ...s_{T-1}, a_{T-1}, r_{T-1}, s_T\} \sim \pi_\theta$ **do**

  for *t=1 to T-1* **do**

    $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) G_t$;

  end

end

return θ

3.2.1 Calculating the policy Gradient

For getting the gradient we need to calculate two parts, let's say we initialise the agent and let it play a trajectory τ through the environment. The actions of the agent will be selected by performing sampling from the softmax output of the neural network. So at each step we can calculate $\log \pi_\theta(s_t, a_t)$ by simply taking the log of the softmax output values from the neural network. For the second part I have implemented two variants one is by calculating the future rewards from time step t till T-1:

$$G_t = \sum_{t'=t}^{T-1} r_{t'}$$

other variant is calculating future Discounted reward:

$$G_t = \sum_{t'=t}^{T-1} \gamma^{t-t'} r_{t'}$$

One thing that we should note that we can only calculate the gradient at first step in the episode, only if we know the reward value of all.Therefore, in order to execute this method of learning, we can only take gradient learning steps after the full episode has been played to completion. Only after the episode is complete can we perform the training step. For implementing in keras/TensorFlow 2, it turns out we can just use the cross entropy loss function given by:

$$CE = -\sum p(x) log(q(x))$$

Which is just the summation between one function multiplied by the log of another function over the possible values of the argument x. Then we use Adam or RMSProp gradient descent methods. But we need to find the proper learning rate and number of hidden layers and their sizes to get the desired learning, which we should find by trial and error, there is no shorter way.

Note: The policy gradient is actually maximization problem but the negative sign in cross entropy makes it minimization problem so we can directly use the gradient decent opti-

mizers.

3.2.2 Adding a baseline

Monte Carlo plays out the whole trajectory and records the exact rewards of a trajectory. However, the stochastic policy may take different actions in different episodes. One small turn can completely alter the result. So Monte Carlo has no bias but high variance. Variance hurts deep learning optimization. The variance provides conflicting descent direction for the model to learn. One sampled rewards may want to increase the log likelihood and another may want to decrease it. This hurts the convergence. To reduce the variance caused by actions, we want to reduce the variance for the sampled rewards.
One way we can achieve this by subtracting the G_t from a baseline $b(s_t)$ as below given below:

$$\boxed{\nabla_\theta E[R] = E[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta)(\sum_{t'=t}^{T-1} \gamma^{t-t'} r_{t'} - E_{\tau_1,\tau_2..} G(\tau)]} \tag{12}$$

Here the baseline we took is the average of total discounted reward for all the episode till now.
Total discounted reward for an episode τ is:

$$b(s_t) = G(\tau) = \sum_{t=0}^{t=T-1} \gamma^t r_t \tag{13}$$

So we have to take average of all the G(τ) for all the episodes till the present episode. We can update the **REINFORCE** algorithm by replacing G_t with $(G_t - b(s_t))$ to implement the baseline version.

3.2.3 implementation

We have implemented all the the three variants of REINFORCE algorithm as given by equations 9, 10 and 12. This algorithm runs faster as we are updating after each episode rather than updating after each step. The results for there three variants are in section 4.2.1, 4.2.2 and 4.2.3

3.3 Q Actor Critic

Let's go back to vanilla policy gradient again to see how the Actor Critic architecture comes in:

$$\nabla_\theta E_\tau[R] = E_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta) G_t]$$

We can then decompose the expectation into:

$$\nabla_\theta E_\tau[R] = E_{s_0,a_0..,s_t,a_t}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta) E_{r_{t+1},s_{t+1},...,r_{T-1},s_T}[G_t]]$$

If we look at the second expectation term, it is the Q value.

$$E_{r_{t+1},s_{t+1},...,r_{T-1},s_T}[G_t] = Q(s_t,a_t)$$

Plugging that in, we can rewrite the update equation as such:

$$\nabla_\theta E_\tau[R] = E_{s_0,a_0..,s_t,a_t}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta) Q_w(s_t,a_t)]$$

$$\boxed{\nabla_\theta E_\tau[R] = E_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta) Q_w(s_t,a_t)]} \tag{14}$$

We can learn the Q value by parameterizing the Q function with a neural network (denoted by subscript w above). This leads us to **Actor Critic Methods**, where:

- **Critic** updates the value function parameters w and depending on the algorithm it could be action-value(Q Value) or state-value(V value).
- **Actor** updates the policy distribution in the direction suggested by the Critic.

Here both the Critic and Actor functions are parameterized with neural networks. Below is the pseudocode for Q-Actor-Critic:

Algorithm 2 Q Actor Critic

Initialize parameters s,θ,w and learning rate α_θ, α_w; sample a$\sim \pi_\theta(a|s)$.

for *t=0...T-1* **do**

Sample reward $r_t \sim R(s,a)$ and next state $s' \sim P(s'|s,a)$;

Then sample the nest action a$' \sim \pi_\theta(a'|s')$;

Update the policy parameters:

$\theta \leftarrow \theta + \alpha_\theta Q_w(s,a) \nabla_\theta \log \pi_\theta(a|s)$;

Compute the correction (TD error) for action-value at time t:

$\delta_t = r_t + \gamma Q_w(s',a') - Q_w(s,a)$

and use it to update the parameters of Q function:

$w \leftarrow w + \alpha_w \delta_t \nabla Q_w(s,a)$

Move to $a \leftarrow a'$ and $s \leftarrow s'$

end

The above algorithm is adapted from Weng (n.d.)

Here we update both the critic network and the value network at each step rather than

after each episode has we seen in REINFORCE algorithm.
In REINFORCE we directly use future discounted reward to guide and update of actor. Such a method does not involve any learning, because the observations during one episode do not future reward of another episode. In this approach,the observations from all past episodes affect current critic parameter w, and in this sense critic is "learning". This can be advantageous because, as long as θ is slowly changing, the observations from recent episodes carry useful information on the q-function under the current policy. So actor parameters should be updated at a time scale slower than that of critic.

3.3.1 Implementation

We implemented this as given in the algorithm, and we are using mean square error as the loss function to update our critic network. The results obtained weren't that promising, may be we have to tweak the parameters even more. Here in this algorithm we use the following equation as the target for the $Q_w(s, a)$:

$$target = r_t + \gamma Q_w(\acute{s}, \acute{a})$$

Which is also known as Bellman Equation. We thought rather than using this as target for Q, we can just take the future discounted reward from that state to act as a target, which is indeed acceptable. Look at the definition of Q function in section 2.4.
Here we can only update the critic function after an episode, because we need know the full trajectory to calculate the future discounted reward. We also update the actor as well at the end of episode by storing Q values at each step and then calculating the total loss and the gradient before updating at the end of the episode. Experimental we observed that rather than just using Q directly to update actor we got better results by normalizing the Q values before using for update, by normalizing I mean subtracting each Q values stored by the mean of all Q values stored for that episode and divide by the standard deviation. This changed version runs faster than the one given in algorithm 2 because here we are updating after each episode unlike each step in the algorithm mentioned.
We also introduced baseline which we used for REINFORCE, i.e the estimated total reward of the episodes given in equation 13, by subtracting with the all the Q values obtained at the each step of the episode and then using the resultant to update the actor network.
Some of the common problem you face here is choosing the proper network, I mean the number of hidden layers and its size, and as well as the learning rate for both actor and critic. For better results and convergence rate critic should have bigger learning rate than actor. We are using Adam optimizer for updating the weights of the network.
Results for this algorithm is in section 4.2.4.

3.4 Advantage Actor Critic (A2C)

We improve upon Q Actor Critic by subtracting with the well known baseline, the State Value function.

So using V function as baseline we are subtracting Q value. Intuitively, this means how much better it is to take a specific action compared to the average, general action at the given state. We will call this value the **Advantage value/Advantage function**:

$$A(s_t, a_t) = Q_w(s_t, a_t) - v_v(s_t)$$

Rather than having two networks for calculating advantage function, we can use the relationship between the Q and V from the Bellman optimality equation:

$$Q(s_t, a_t) = E[r_{t+1} + \gamma V(s_{t+1})]$$

So, we can rewrite the advantage as:

$$A(s_t, a_t) = E[r_{t+1} + \gamma V_v(s_{t+1})] - v_v(s_t)$$

Then, we only have to use one neural network for the V function (parameterized by v above). So we can rewrite the update equation as:

$$\boxed{\nabla_\theta E_\tau[R] = E_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t, \theta) A(s_t, a_t)]} \tag{15}$$

This is the **Advantage Actor Critic(A2C)**.
The algorithm is similar to Q Actor Critic as given in Algorithm 2. Rather than Q function in the actor/policy parameter we use Advantage function to update the parameters and for critic update we use V function parameterized by v.

3.4.1 Implementation

The network configuration for finding Q is that it takes states have inputs and the output will be Q value for all the action, so total the network will give separate values for each action.
In case for V function network there will be only one output as there is one state value for each state unlike Q function which as separate value for each action that particular state.
One thing that we conventionally follow is that that the actor parameters be updated at a time scale slower than that of critic for convergence of actor-critic.

3.5 TD(Lambda) Actor Critic

Temporal difference is an agent learning from an environment through episodes with no prior knowledge of the environment. This means temporal difference takes a model-free or unsupervised learning approach.
These allow us to learn online at the same time we interact with an environment and are based on the notion of bootstrapping. This means that we use our current approximation

for the value of a state (which might be wrong) to update our estimated value for another state. Everything goes well as long as all of the approximations get better with time. This method is called TD(0), and is biased, while having reduced variance. A Monte-Carlo estimation method is not biased, but has a lot of variance since it uses the outcome of a complete episode to perform an update. The variance comes from the fact that, at each interaction, there's randomness involved in picking an action.
One problem with TD(0) is that it uses information from only one step to perform an update. Typically, the action that caused a reward to be seen might have happened several time steps in the past. This means that using only the most recent information can lead to slow convergence.
To solve this we have to use more than one step to perform an update. But how many? Instead of doing that, we'll instead employ a mathematical trick to use all relevant timesteps, weighted by a factor that reflects the chance that said timestep caused us to see the reward we're seeing. These methods are called TD(λ) methods.
Here we employ backward view of TD(lambda). Instead of waiting for what is going to happen next to be able to perform an update, we will remember what happened in the past and use current information to update the state-values/Q values for every state we've seen so far.
To do that, we'll employ eligibility traces, a nifty idea that allows us to do just that. We first initialise with zero's (it has some dimension as the weights). The eligibility vectors can be updated according to

$$z_{t+1} = \gamma\lambda z_t + \nabla Q(s_{t+1}, a_{t+1})$$

initialize z_{-1} to zeros.
Before updating weights we also need to calculate TD error given as:

$$d_t = r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

Then we update the weights as:

$$w_{t+1} = w_t + \alpha d_t z_t$$

We used the idea similar to as given in Tsitsiklis (n.d.)

3.5.1 Implementation

The critic is updated on each step of the episode by following above update rule. But unlike Q Actor Critic here we didn't update the actor on every step rather we accumulated the gradient for each mini batch of episodes, and updated the weight using this accumulated gradient. This makes sures that critic is updated more often then actor.

3.6 Konda Actor Critic

This algorithm will differ only as far as the critic updates are concerned. The critic is a TD algorithm with a linearly parameterized approximation architecture for the Q-function. The idea is similar to as given in Konda and Tsitsiklis (n.d.).

Given a policy parameterization, can be used to derive an appropriate form for the Q-function parameterization, in other words We are using the same features as the policy for finding the Q-function linearly. The following idea is taken from Richard S. Sutton (n.d.) . Consider ϕ as the output of the layer previous to output layer of actor network. Then we can define the Q function as:

$$Q_w(s,a) = w^T[\phi_{sa} - \sum_b \pi(s,b)\phi_{sb}]$$

Here b is all the actions. Does the mean the right hand side is zero!!? There is a catch here, lets say there are 2 actions the what is the difference between ϕ_{s1} and ϕ_{s2}, and let the size of ϕ be 4(output of the layer previous to output layer of actor network). The ϕ_{s1} will be array with 4 zeros followed by ϕ, ϕ_{s2} will be array with 4 zeros followed by ϕ as it is for second action.

Now Let's consider the temporal difference d_t corresponding to the transition from t to t+1 step:

$$d_t = r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

Let z_t be the eligibility vector of dimension K. By following Tsitsiklis (n.d.), TD(λ) update are given by:

$$w_{t+1} = w_t + \alpha d_t z_t$$

and the eligibility vectors can be updated according to:

$$z_{t+1} = \gamma\lambda z_t + \phi(s_{t+1}, a_{t+1})$$

initialize z_{-1} to zeros.

3.6.1 Intuition

The reason why we are using the features from actor is mainly to reduce the error while projecting the Q value into the space of gradient of policy. As we know we are using $Q(s,a) * \nabla log(\pi(s,a))$ as the gradient. Here Konda says that we are projecting the Q function into gradient log and Q will be in different space (in case of Q Actor Critic). So there will be error while projecting into the gradient space. So to reduce this he tells us to use the feature space of actor to reduce this error or loss as it will be feature space of gradient

of log too.

3.6.2 implementation

Here the critic is updated in each step of the episode and we are updating actor similar to what we have implemented for TD(lambda) Actor-Critic, by updating after accumulating the gradient over mini batch of episodes.
We also tried without using TD(lambda), similar to Q Actor critic variant we used the total future reward from the state as target for Q function and performed mean square error loss to update the critic network after an episode. Which is nothing but TD(1)!!! Here we update actor after each episode.
We also tried one more variant on this that is rather than calculating Q, we calculated V and used similar advantage function as A2C, here we need not subtract and append zeros to ϕ we directly input the phi to critic network to obtain the V value. For this we haven't got the results due to time constraint, but others can do work on this.
One more idea is along with the features from actor, we can also add states as features and give input to critic network to V. This is purely experimental, further study is required.

4 Implementing and Experimenting on Actor Critic Algorithm and it's Variant

In this section first we will go through the environments that we are going to use. Then we are going through results obtained for the algorithms explained above on the environment we are going to through next and finally compare all the algorithms for each environment.

4.1 Environments used for testing

For testing our analysis of these RL algorithms we use OpenAI Gym's environments. There are two sets of environments we are testing - Small state environments and Atari game - Ping Pong

4.1.1 Small State Environments

• CartPole

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

Num	Observation	min	max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-∞	∞
2	Pole Angle	$-41.8°$	$41.8°$
3	Pole Velocity At Tip	-∞	∞

Table 1: Observation

Num	Action
0	Push cart to left
1	Push cart to the right

Table 2: Actions

Reward setup: +1 is provided for every time-step that the pole remains upright, including the termination step.
Termination: After reaching goal state or 200 iteration.

• **Acrobot**

The acrobot system includes two joints and two links, where the joint between the two links is actuated. Initially, the links are hanging downwards, and the goal is to swing the end of the lower link up to a given height.

Num	Observation	min	max
0	cos(theta1)	-1	1
1	sin(theta1)	-1	1
2	cos(theta2)	-1	1
3	sin(theta2)	-1	1
4	thetadot1(Angular Velocity)	-4π	4π
5	thetadot2(Angular Velocity)	-4π	4π

Table 3: Observation

Num	Action
0	+1 torque on the joint
1	0 torque on the joint
2	-1 torque on the joint

Table 4: Actions

Reward setup: -1 is provided every time-step that the goal is not achieved and 0 otherwise
Termination: After reaching goal state or 500 iteration.

• **LunarLander**

Here the main goal if to land the Lunar Lander to Landing pad which is always at coordinates(0,0). Coordinates are the first two numbers in state vector.If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest.Landing outside landing pad is possible. There are 3 engines left, right and main. Fuel is infinite, so an agent can learn to fly and then land on its first attempt.

Num	Observation	min	max
0	Horizontal Position	-∞	∞
1	Vertical Position	-∞	∞
2	Horizontal Velocity	-∞	∞
3	Vertical Velocity	-∞	∞
4	Angle	-∞	∞
5	Angular Velocity	-∞	∞
5	Left Leg Contact	-∞	∞
5	Right Leg Contact	-∞	∞

Table 5: Observation

Num	Action
0	Main Engine
1	Right Engine
2	Left Engine
3	Do Nothing

Table 6: Actions

Reward setup: Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If the lander crashes or comes to rest, receiving additional -100 or +100 points.Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points.
Termination: After reaching goal state or 1000 iteration.

• Mountain Car

A car is on a one-dimensional track, positioned between two mountains. The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum.

Num	Observation	min	max
0	Position	-1.2	0.6
1	Velocity	-0.07	0.07

Table 7: Observation

Num	Action
0	Push left
1	No Push
2	Push right

Table 8: Actions

Reward setup: -1 is provided every time-step that the goal is not achieved and 0 otherwise.
Termination: After reaching goal state or 200 iteration.

4.1.2 Atari Games

• Ping Pong

In this environment, the observation is an RGB image of the screen, which is an array of shape (210, 160, 3) Each action is repeatedly performed for a duration of k frames, where k is uniformly sampled from {2,3,4}.
In Pong the reward is simply a +1 for every round the Agent wins, and a -1 for every round the opponent CPU wins. The game terminates when one of the player reaches 21 points.

We use two actions UP_ACTION(2) and DOWN_ACTION(3) other atari actions are not needed.
Before we give the frames as input we preprocess the frame remove the score and other unwanted parts, our main focus should be on bat and the ball position so all background will be converted to black except those two.

4.2 Observation and Results

Here we go through individual algorithm on each environment and show training and testing results. Here there is no direct way to find the parameters (such as learning rate and number of hidden layers), all need to be found only through experiments and patience. There will be plots on how the total reward varies from episode, table containing the total rewards of the first episode, the final episodes, and the best episode during the training. We also fix the weights and run for many episodes and check whether they are performing well by calculating the mean and standard deviation for these episodes.
Some of the algorithms are highly sensitive to parameter and difficult to get results. I have build all my neural network using keras, also note that tenserflow.keras package is slow compare to keras library by almost 10 milliseconds per each step in the episode. So it is better to use plain keras package. If you are comfortable with keras.

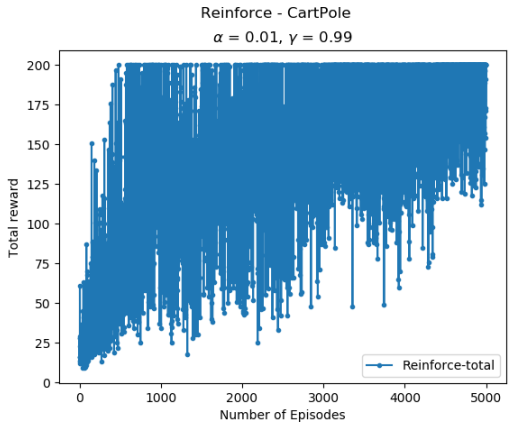
4.2.1 REINFORCE - Total Reward

Refer section 3.2 for more details
The update rule used here is:

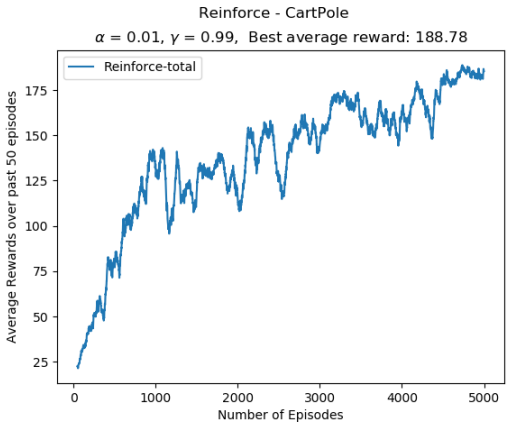
$$\boxed{\nabla_\theta E_\tau[R] = E_\tau[(\sum_{t=0}^{T-1} \gamma^t r_t) \nabla_\theta \sum_{t=0}^{T-1} \log \pi(a_t|s_t, \theta)]}$$

Parameters to look at: Learning Rate(α), Discount factor(γ), Number of hidden layers and it's size;

- **CartPole**



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 3: These are learning results for CartPole, used 0 hidden layers

No. of parameters (Weights and bias)	$\alpha = 0.01, \gamma = 0.99$	initial	final	best
10	Total Reward	29.0	200.0	200.0

Table 9: Values obtained while training

We are using the learnt weights and using it over two policies, namely **Max Policy** and the other one in **Sampling Policy**.

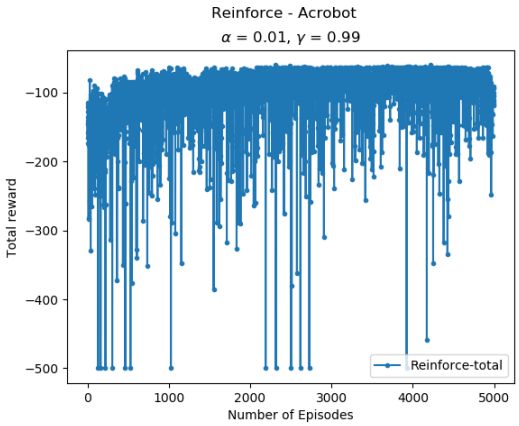
- **Max Policy** Here we choose the action which has the maximum probability given from the policy/actor network.
- **Sampling Policy** Here we sample the action value based on the probabilities provided by the policy/actor network

Weights	Max Policy		Sampling Policy	
$\alpha = 0.01, \gamma = 0.99$	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	200.000	0.000000	189.491	20.050185
Optimal Weights	200.000	0.000000	188.282	20.070039

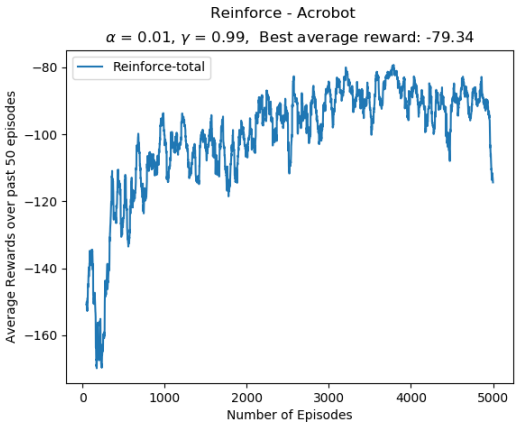
Table 10: Results obtained after freezing weights and used 1000 episodes

---

- **Acrobot**



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 4: These are learning results for Acrobot, used 0 hidden layers

No. of parameters (Weights and bias)	$\alpha = 0.01, \gamma = 0.99$	initial	final	best
21	Total Reward	-144.0	-94.0	-60.0

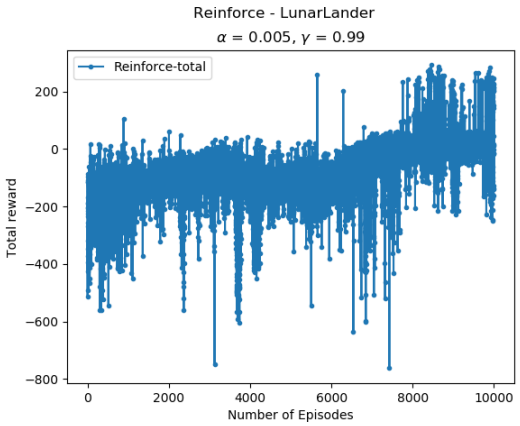
Table 11: Values obtained while training

Weights	Max Policy		Sampling Policy	
$\alpha = 0.01, \gamma = 0.99$	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	-102.676	27.230370	-111.943	40.238237
Optimal Weights	-82.359	22.035565	-87.699	25.231615

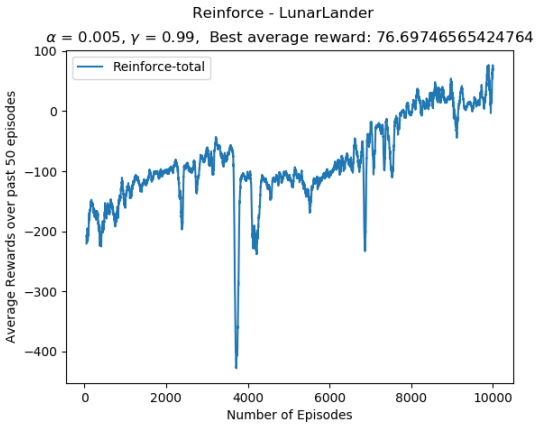
Table 12: Results obtained after freezing weights and used 1000 episodes

---

- **LunarLander**



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 5: These are learning results for LunarLander, used 2 hidden layers with sizes 16 and 16

No. of parameters (Weights and bias)	$\alpha = 0.005, \gamma = 0.99$	initial	final	best
484	Total Reward	-318.44	213.76	291.80

Table 13: Values obtained while training

Weights	Max Policy		Sampling Policy	
$\alpha = 0.005, \gamma = 0.99$	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	62.259851	118.352095	72.302301	134.676106
Optimal Weights	-82.359	22.035565	-87.699	25.231615

Table 14: Results obtained after freezing weights and used 1000 episodes

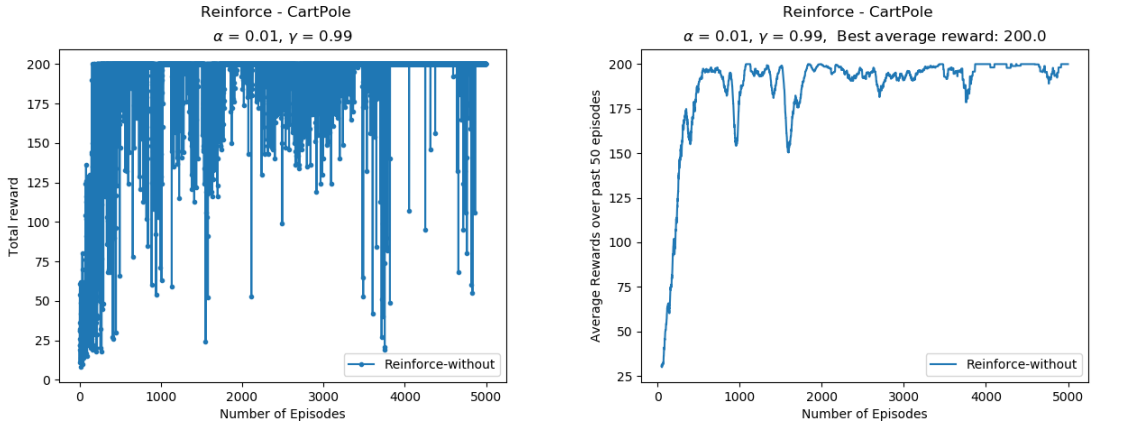
4.2.2 REINFORCE - Future Discounted Reward

Refer section 3.2 for more details
The update rule used here is:

$$\nabla_\theta E_\tau[R] = E_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta) \sum_{t'=t}^{T-1} \gamma^{t-t'} r_{t'}]$$

Parameters to look at: Learning Rate(α), Discount factor(γ), Number of hidden layers and it's size;

- **CartPole**



(a) Total Reward **vs** Episodes

(b) Total average reward over past 50 episode **vs** Episodes

Figure 6: These are learning results for CartPole, used 0 hidden layers

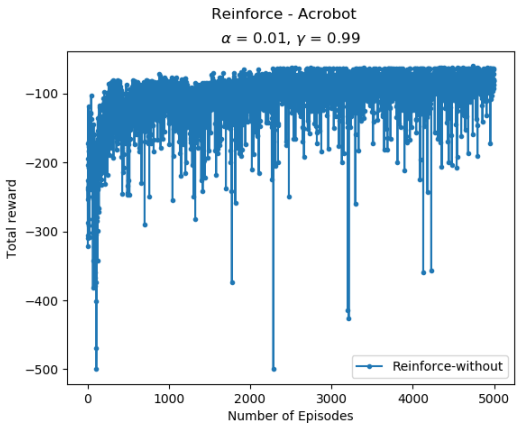
No. of parameters (Weights and bias)	$\alpha = 0.01, \gamma = 0.99$	initial	final	best
10	Total Reward	19.00	200.00	200.00

Table 15: Values obtained while training

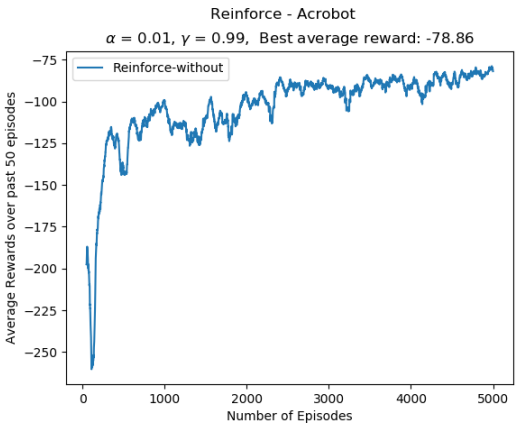
Weights	Max Policy		Sampling Policy	
$\alpha = 0.01, \gamma = 0.99$	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	200.000	0.000000	199.227	8.203260
Optimal Weights	200.000	0.000000	199.412	8.015626

Table 16: Results obtained after freezing weights and used 1000 episodes

- **Acrobot**



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 7: These are learning results for Acrobot, used 0 hidden layers

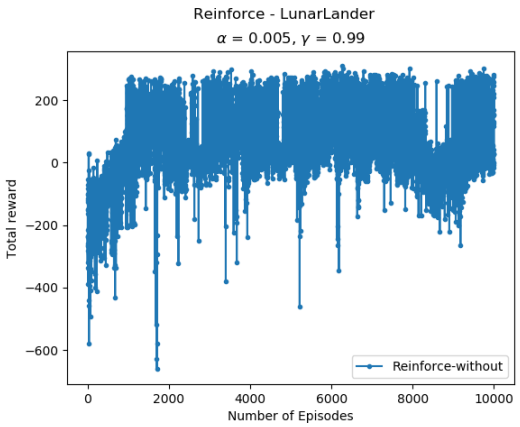
No. of parameters (Weights and bias)	$\alpha = 0.01, \gamma = 0.99$	initial	final	best
21	Total Reward	-236.00	-80.00	-60.00

Table 17: Values obtained while training

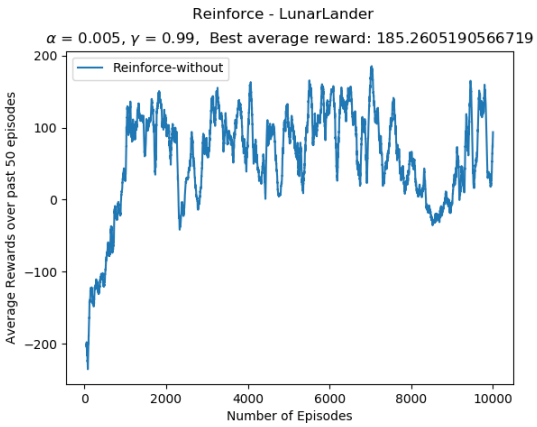
Weights	Max Policy		Sampling Policy	
$\alpha = 0.01, \gamma = 0.99$	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	-86.736	26.480376	-85.108	24.327358
Optimal Weights	-85.685	24.796205	-84.510	21.758582

Table 18: Results obtained after freezing weights and used 1000 episodes

- **LunarLander**



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 8: These are learning results for LunarLander, used 2 hidden layers with sizes 16 and 16

No. of parameters (Weights and bias)	$\alpha = 0.005, \gamma = 0.99$	initial	final	best
484	Total Reward	-238.03	152.41	308.80

Table 19: Values obtained while training

Weights	Max Policy		Sampling Policy	
$\alpha = 0.005, \gamma = 0.99$	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	131.681804	101.101556	107.564797	93.331925
Optimal Weights	145.940888	56.955596	142.753033	54.689188

Table 20: Results obtained after freezing weights and used 1000 episodes

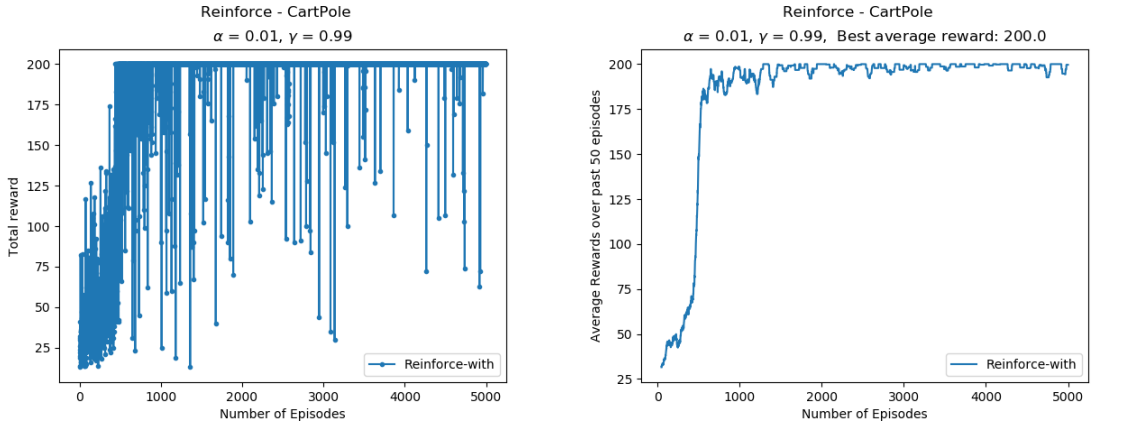
4.2.3 REINFORCE - with baseline

Refer section3.2.2 for more details
The update rule used here is:

$$\nabla_\theta E[R] = E[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta)(\sum_{t'=t}^{T-1} \gamma^{t-t'} r_{t'} - E_{\tau_1,\tau_2..} G(\tau)]$$

Parameters to look at: Learning Rate(α), Discount factor(γ), Number of hidden layers and it's size;

- **CartPole**



(a) Total Reward **vs** Episodes

(b) Total average reward over past 50 episode **vs** Episodes

Figure 9: These are learning results for CartPole, used 0 hidden layers

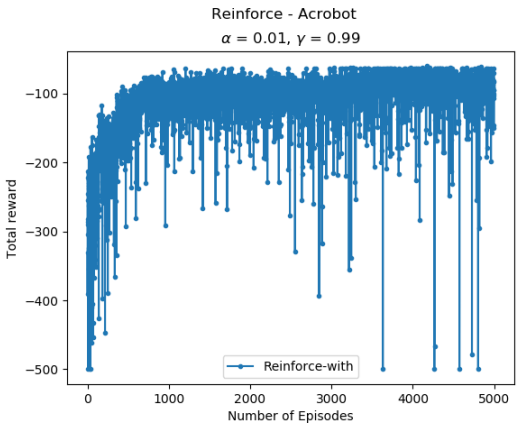
No. of parameters (Weights and bias)	$\alpha = 0.01, \gamma = 0.99$	initial	final	best
10	Total Reward	20.00	200.00	200.00

Table 21: Values obtained while training

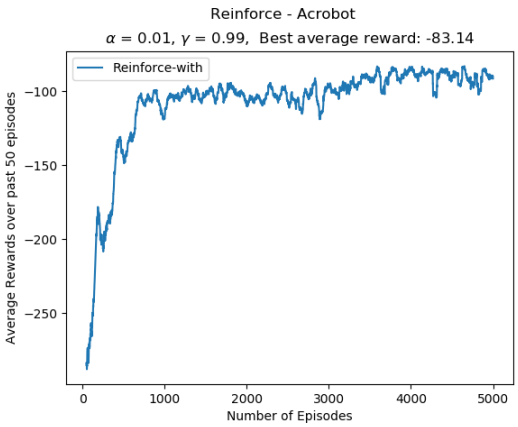
Weights	Max Policy		Sampling Policy	
$\alpha = 0.01, \gamma = 0.99$	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	200.000	0.000000	198.710	11.706831
Optimal Weights	200.000	0.000000	198.468	12.594641

Table 22: Results obtained after freezing weights and used 1000 episodes

- **Acrobot**



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 10: These are learning results for Acrobot, used 0 hidden layers

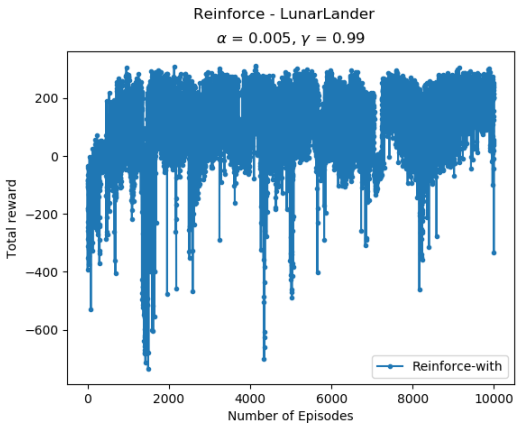
No. of parameters (Weights and bias)	$\alpha = 0.01, \gamma = 0.99$	initial	final	best
21	Total Reward	-331.00	-103.00	-60.00

Table 23: Values obtained while training

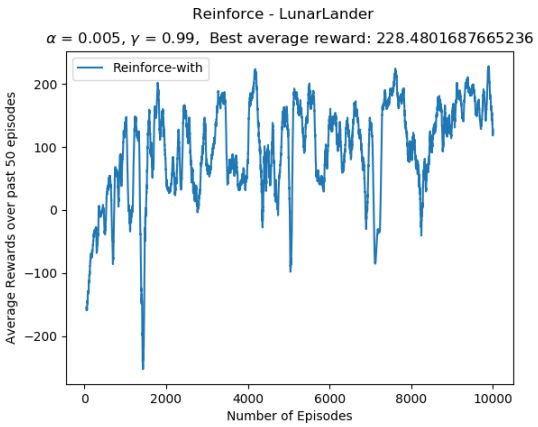
Weights	Max Policy		Sampling Policy	
$\alpha = 0.01, \gamma = 0.99$	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	-85.969	28.910310	-90.243	35.010798
Optimal Weights	-86.855	27.618001	-87.389	24.808944

Table 24: Results obtained after freezing weights and used 1000 episodes

- **LunarLander**



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 11: These are learning results for LunarLander, used 2 hidden layers with sizes 16 and 16

No. of parameters (Weights and bias)	$\alpha = 0.005, \gamma = 0.99$	initial	final	best
484	Total Reward	-92.76	202.72	310.45

Table 25: Values obtained while training

Weights	Max Policy		Sampling Policy	
$\alpha = 0.005, \gamma = 0.99$	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	127.237536	167.540327	129.196510	158.341886
Optimal Weights	193.393874	65.018306	193.450839	72.577900

Table 26: Results obtained after freezing weights and used 1000 episodes

4.2.4 Q Actor Critic

Refer section 3.3 for more details

The update rule used here is:

$$\nabla_\theta E_\tau[R] = E_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta) Q(s_t,a_t)]$$

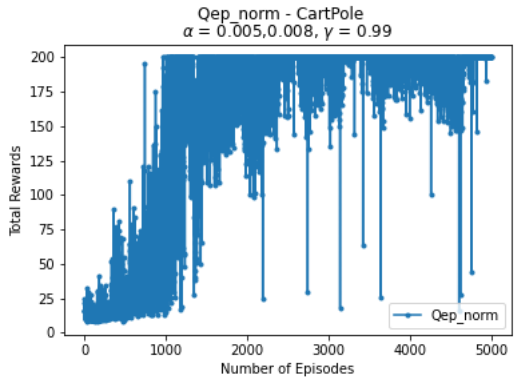
Parameters to look at: Learning Rates($\alpha 1 - actor, \alpha 2 - critic$), Discount factor($\gamma$), Number of hidden layers and it's size for both actor and critic;

Here while implementing we modified the algorithm 2, because we couldn't get the required results as expected may be due to not finding the right parameters, as these algorithms are very sensitive to above mentioned parameters, the modified version is explained in this section 3.3.1.

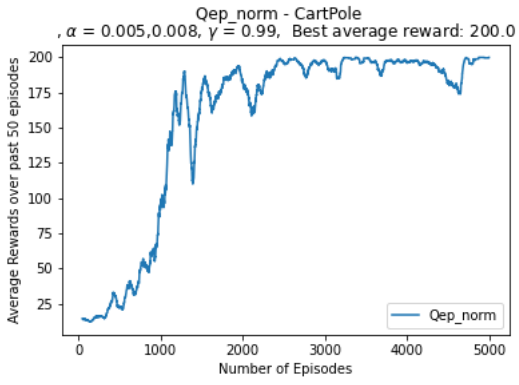
1. Using the total future reward from that particular state till terminal state as target for Q for that particular state and updating the Q network at the end of episode while doing so.
2. One more extension to this method is adding the baseline, similar to what we saw in REINFORCE algorithm, taking average total reward of all past episode and using it has the baseline, and subtracting it from Q value obtained to act as advantage function to actor network

• **CartPole**

Method 1:



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 12: These are learning results for CartPole, used 0 hidden layers for actor and 1 hidden layer with size 20 for critic

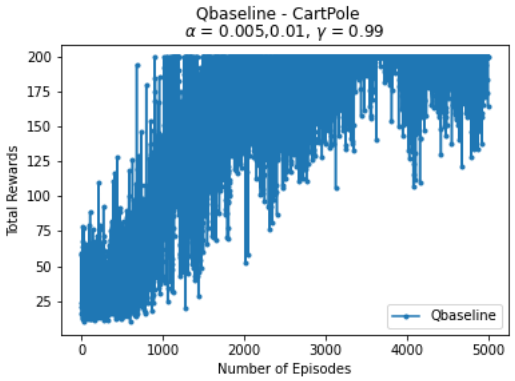
No. of parameters (Weights and bias)	$\alpha1 = 0.005, \alpha2 = 0.008, \gamma = 0.99$	initial	final	best
10+142=152	Total Reward	16.00	200.00	200.00

Table 27: Values obtained while training

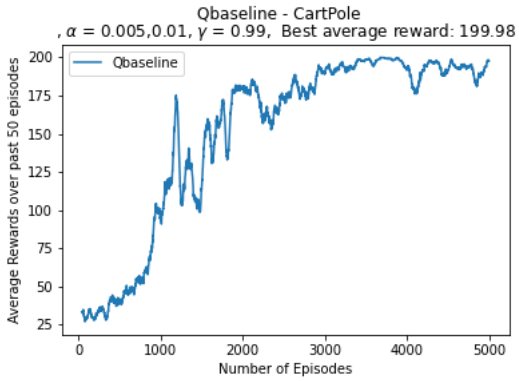
Weights $\alpha1 = 0.005, \alpha2 = 0.008, \gamma = 0.99$	Max Policy		Sampling Policy	
	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	200.0	0.0	200.000	0.000000
Optimal Weights	200.0	0.0	199.195	11.355922

Table 28: Results obtained after freezing weights and used 200 episodes

Method 2:



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 13: These are learning results for CartPole, used 0 hidden layers for actor and 1 hidden layer with size 16 for critic

No. of parameters (Weights and bias)	$\alpha1 = 0.005, \alpha2 = 0.01, \gamma = 0.99$	initial	final	best
10+114=124	Total Reward	59.00	200.00	200.00

Table 29: Values obtained while training

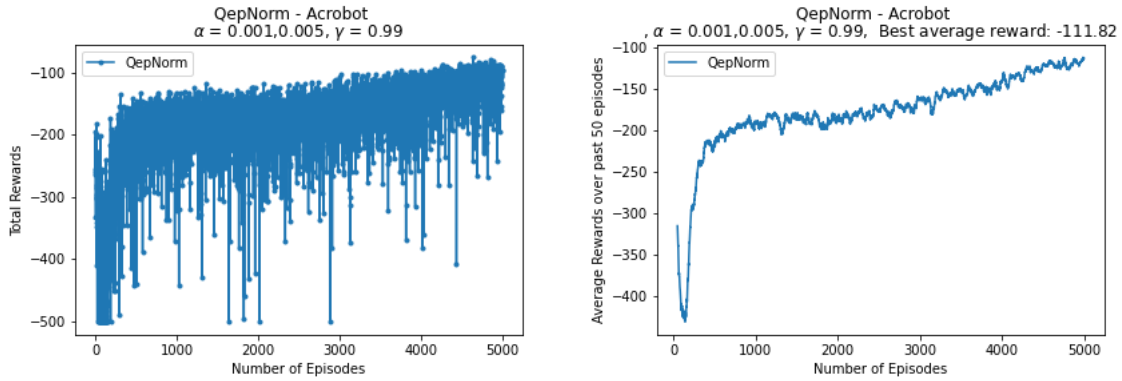
Weights $\alpha1 = 0.005, \alpha2 = 0.01, \gamma = 0.99$	Max Policy		Sampling Policy	
	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	200.0	0.0	198.690	5.645697
Optimal Weights	200.0	0.0	198.940	4.531710

Table 30: Results obtained after freezing weights and used 200 episodes

---

- **Acrobot**

Method 1:



(a) Total Reward **vs** Episodes

(b) Total average reward over past 50 episode **vs** Episodes

Figure 14: These are learning results for Acrobot, used 0 hidden layers for actor and 1 hidden layer with size 36 for critic

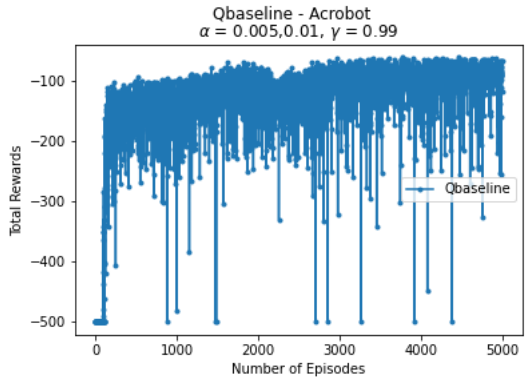
No. of parameters (Weights and bias)	$\alpha1 = 0.001, \alpha2 = 0.005, \gamma = 0.99$	initial	final	best
21+363=384	Total Reward	-263.00	-96.00	-76.00

Table 31: Values obtained while training

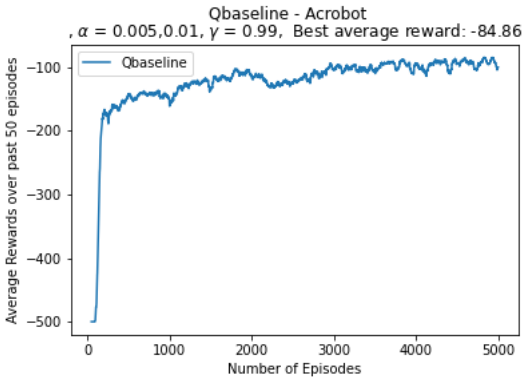
Weights $\alpha1 = 0.001, \alpha2 = 0.005, \gamma = 0.99$	Max Policy		Sampling Policy	
	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	-109.530	43.498840	-115.945	28.131334
Optimal Weights	-109.160	25.781086	-118.705	25.907875

Table 32: Results obtained after freezing weights and used 1000 episodes

Method 2:



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 15: These are learning results for Acrobot, used 0 hidden layers for actor and 1 hidden layer with size 32 for critic

No. of parameters (Weights and bias)	$\alpha 1 = 0.005, \alpha 2 = 0.01, \gamma = 0.99$	initial	final	best
21+323=344	Total Reward	-500.00	-66.00	-61.00

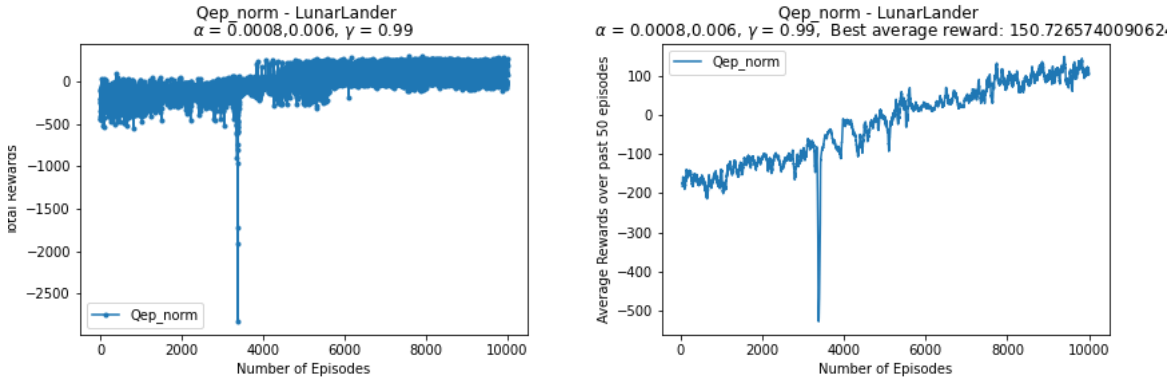
Table 33: Values obtained while training

Weights $\alpha 1 = 0.005, \alpha 2 = 0.01, \gamma = 0.99$	Max Policy		Sampling Policy	
	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	-88.930	32.567547	-94.430	36.496371
Optimal Weights	-85.740	22.990268	-93.700	36.558173

Table 34: Results obtained after freezing weights and used 1000 episodes

- **LunarLander**

Method 1:



(a) Total Reward **vs** Episodes

(b) Total average reward over past 50 episode **vs** Episodes

Figure 16: These are learning results for LunarLander, used 2 hidden layers with sizes 16 and 16 for actor, and 2 hidden layers with sizes 64 and 64

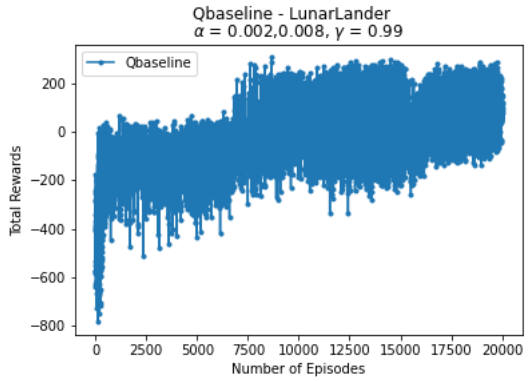
No. of parameters (Weights and bias)	$\alpha 1 = 0.0008, \alpha 2 = 0.006, \gamma = 0.99$	initial	final	best
484+4996=5480	Total Reward	-113.70	74.09	291.51

Table 35: Values obtained while training

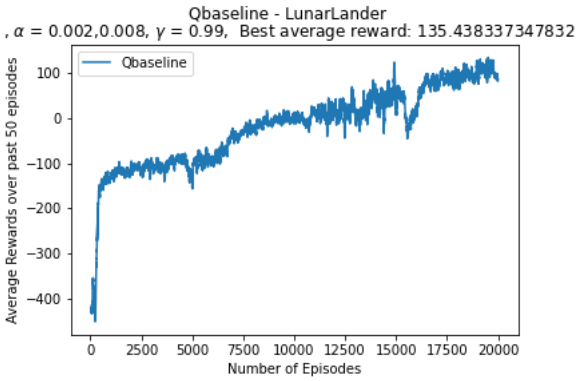
Weights $\alpha 1 = 0.0008, \alpha 2 = 0.006, \gamma = 0.99$	Max Policy		Sampling Policy	
	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	106.597253	105.569002	109.501334	100.708720
Optimal Weights	14.546658	126.567448	13.648181	116.330477

Table 36: Results obtained after freezing weights and used 1000 episodes

Method 2:



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 17: These are learning results for LunarLander, used 0 hidden layers and 1 hidden layers with size 128

No. of parameters (Weights and bias)	$\alpha 1 = 0.002, \alpha 2 = 0.008, \gamma = 0.99$	initial	final	best
36+1668=1704	Total Reward	-581.75	116.11	310.14

Table 37: Values obtained while training

Weights $\alpha 1 = 0.002, \alpha 2 = 0.008, \gamma = 0.99$	Max Policy		Sampling Policy	
	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	-48.728562	23.089275	96.855012	48.652633
Optimal Weights	-331.381458	105.206857	-9.279020	63.724983

Table 38: Results obtained after freezing weights and used 1000 episodes

4.2.5 Advantage Actor Critic

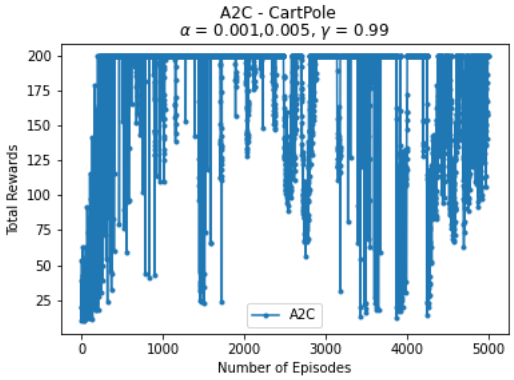
Refer section 3.4 for more details
The update rule used here is:

$$A(s_t, a_t) = E[r_{t+1} + \gamma V_v(s_{t+1})] - v_v(s_t)$$

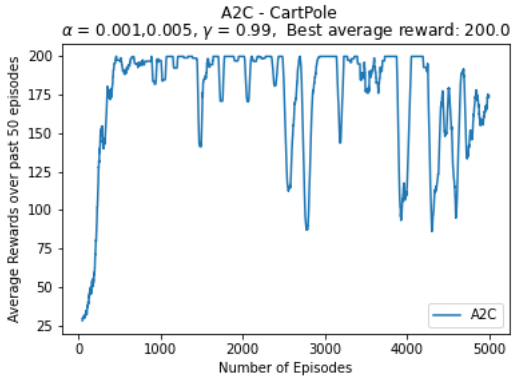
$$\boxed{\nabla_\theta E_\tau[R] = E_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t, \theta) A(s_t, a_t)]}$$

Parameters to look at: Learning Rates($\alpha 1 - actor, \alpha 2 - critic$), Discount factor($\gamma$), Number of hidden layers and it's size for both actor and critic;

- **CartPole**



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 18: These are learning results for CartPole, used 0 hidden layers for actor and 1 hidden layers with size 32

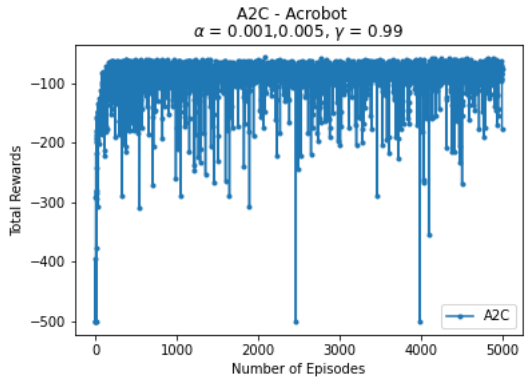
No. of parameters (Weights and bias)	$\alpha 1 = 0.001, \alpha 2 = 0.005, \gamma = 0.99$	initial	final	best
10+226=236	Total Reward	20.00	200.00	200.00

Table 39: Values obtained while training

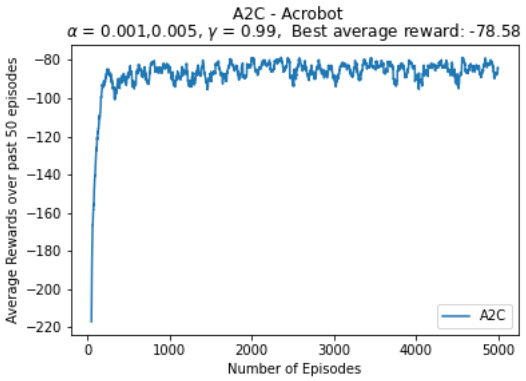
Weights $\alpha 1 = 0.001, \alpha 2 = 0.005, \gamma = 0.99$	Max Policy		Sampling Policy	
	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	200.0	0.0	192.740	14.873547
Optimal Weights	200.0	0.0	192.005	15.244506

Table 40: Results obtained after freezing weights and used 200 episodes

- **Acrobot**



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 19: These are learning results for Acrobot, used 0 hidden layers and 1 hidden layers with size 32

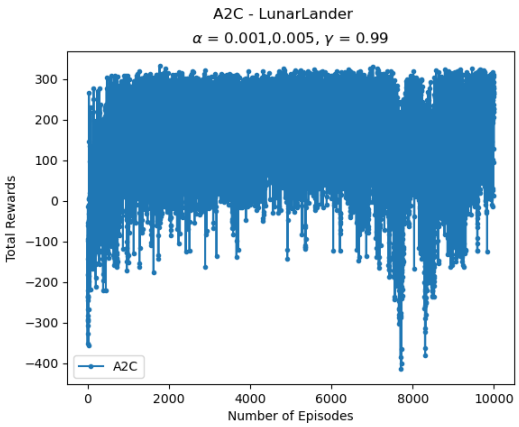
No. of parameters (Weights and bias)	$\alpha1 = 0.001, \alpha2 = 0.005, \gamma = 0.99$	initial	final	best
21+323=344	Total Reward	-500.00	-178.00	-57.00

Table 41: Values obtained while training

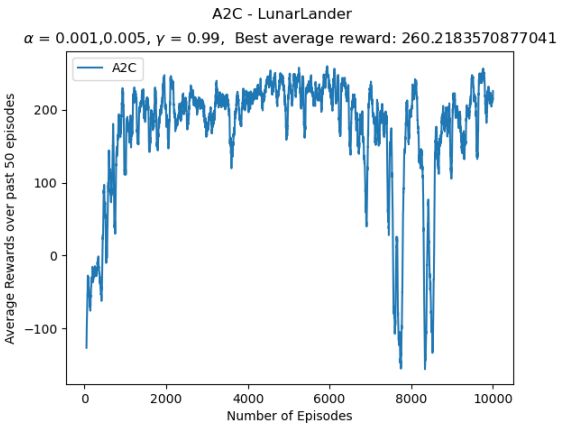
Weights $\alpha1 = 0.001, \alpha2 = 0.005, \gamma = 0.99$	Max Policy		Sampling Policy	
	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	-85.940	21.630451	-82.570	18.657843
Optimal Weights	-91.965	49.807869	-87.270	25.483075

Table 42: Results obtained after freezing weights and used 200 episodes

- **LunarLander**



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 20: These are learning results for LunarLander, used 1 hidden layer with size 64 for actor and 1 hidden layer with size 128

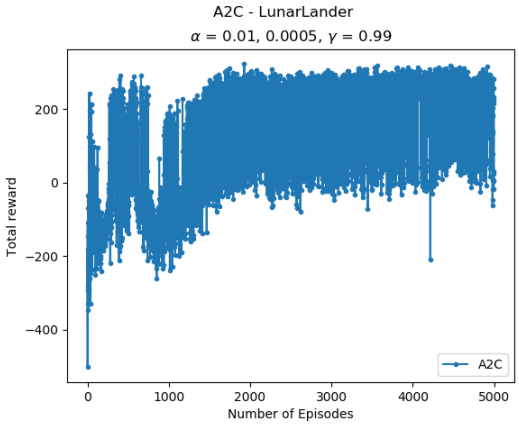
No. of parameters (Weights and bias)	$\alpha1 = 0.001, \alpha2 = 0.005, \gamma = 0.99$	initial	final	best
+4996=5032	Total Reward	-95.22	262.35	331.40

Table 43: Values obtained while training

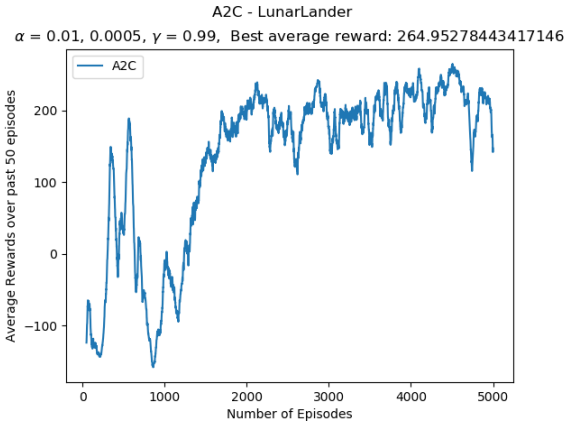
Weights $\alpha1 = 0.001, \alpha2 = 0.005, \gamma = 0.99$	Max Policy		Sampling Policy	
	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	231.60	68.50	210.15	75.55
Optimal Weights	193.78	90.90	215.25	73.56

Table 44: Results obtained after freezing weights and used 200 episodes

Experimenting by keeping actor learning rate higher then critic network



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 21: These are learning results for LunarLander, used 0 hidden layers for actor and 2 hidden layers with sizes 64 and 64

No. of parameters (Weights and bias)	$\alpha1 = 0.01, \alpha2 = 0.0005, \gamma = 0.99$	initial	final	best
36+4996=5032	Total Reward	-108.94	280.69	322.51

Table 45: Values obtained while training

Weights $\alpha1 = 0.01, \alpha2 = 0.0005, \gamma = 0.99$	Max Policy		Sampling Policy	
	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	191.887689	104.325451	193.015698	101.437626
Optimal Weights	201.008947	100.521997	192.470480	101.654280

Table 46: Results obtained after freezing weights and used 200 episodes

As we can see letting critic learn faster than actor is much more stable, we can see that in the variance/standard deviation obtained in both the case.

4.2.6 Konda Actor Critic

Refer section 3.6 for more details
The update rule used here is:

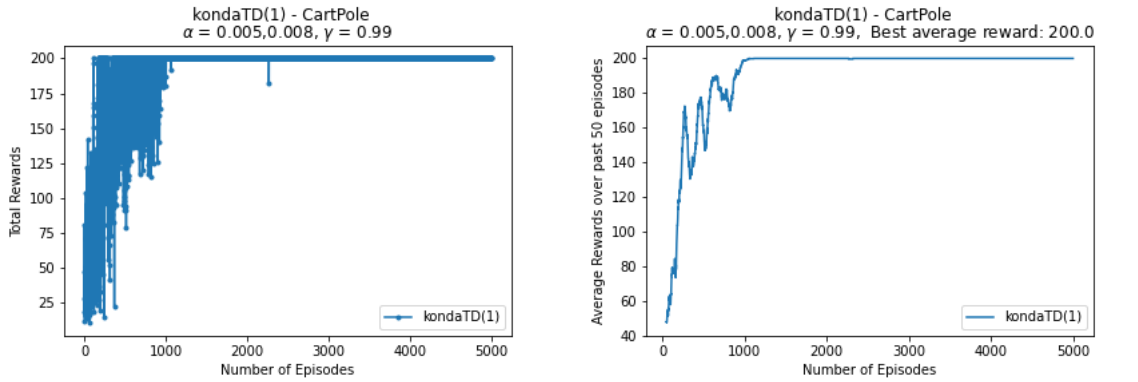
$$Q_w(s,a) = w^T[\phi_{sa} - \sum_b \pi(s,b)\phi_{sb}]$$

$$\boxed{\nabla_\theta E_\tau[R] = E_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta) Q_w(s_t,a_t)]}$$

Parameters to look at: Learning Rates($\alpha 1 - actor, \alpha 2 - critic$), Discount factor($\gamma$), Number of hidden layers and it's size for both actor and critic, Lambda also(but we are directly implementing TD(1));

We tried TD lambda for updating critic network, but we couldn't find the right parameters in this given time.
We implemented TD(1) directly to update the critic network, and used the resultant Q as advantage to actor network refer section 3.6.2 and the following are the results.

- **CartPole**



(a) Total Reward **vs** Episodes

(b) Total average reward over past 50 episode **vs** Episodes

Figure 22: These are learning results for CartPole, used 0 hidden layers for actor and critic

No. of parameters (Weights and bias)	$\alpha 1 = 0.005, \alpha 2 = 0.008, \gamma = 0.99$	initial	final	best
10+8=18	Total Reward	12.00	200.00	200.00

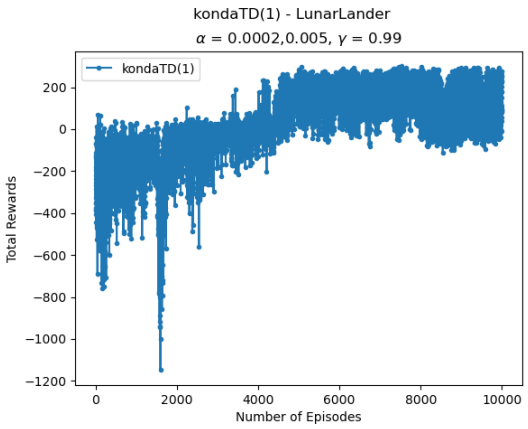
Table 47: Values obtained while training

Weights $\alpha1 = 0.005, \alpha2 = 0.008, \gamma = 0.99$	Max Policy		Sampling Policy	
	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	200.0	0.0	200.0	0.0
Optimal Weights	200.0	0.0	185.94	33.39

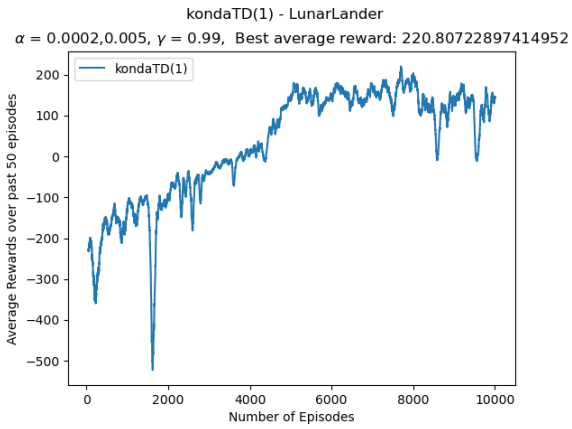
Table 48: Results obtained after freezing weights and used 200 episodes

---

- **LunarLander**



(a) Total Reward **vs** Episodes



(b) Total average reward over past 50 episode **vs** Episodes

Figure 23: These are learning results for LunarLander, used 1 hidden layers with sizes 64 for actor network

No. of parameters (Weights and bias)	$\alpha1 = 0.0002, \alpha2 = 0.005, \gamma = 0.99$	initial	final	best
836+256=1092	Total Reward	-195.42	200.80	299.86

Table 49: Values obtained while training

Weights $\alpha1 = 0.0002, \alpha2 = 0.005, \gamma = 0.99$	Max Policy		Sampling Policy	
	Mean Reward	Standard Deviation	Mean Reward	Standard Deviation
Final Weights	53.765841	95.915143	163.957316	79.741961
Optimal Weights	156.607492	72.560110	117.455399	100.555148

Table 50: Results obtained after freezing weights and used 200 episodes

---

4.3 Comparison of all variants

Things to note
max policy - Choosing the action with maximum probability
sampling policy - Sampling the action based on the probability of the actions.
Optimal Weights - is the weight of the network when the reward was the highest.
par - No. of parameters
max-μ - mean of total reward of 200 episode following max policy
max-σ - Standard deviation of total reward of 200 episode following max policy
samp-μ - mean of total reward of 200 episode following sampling policy
samp-σ - mean of total reward of 200 episode following sampling policy
Variants
total - Reinforce with total discounted reward refer section 3.2
without - Reinforce with total discounted future reward refer section 3.2
with - Reinforce with total discounted future reward with baseline refer section 3.2
Qac - Q-Actor critic given in algorithm 2
Qep - Change to Q-Actor Critic mentioned in section 3.3.1
QepNorm - similar to Qep only change is that we normalize Q before using it has advantage function to update Actor network, refer section 3.3.1
Qbaseline - Adding baseline to QepNorm method refer section 3.3.1
TD - TD Actor critic refer section 3.5
kondaTD(1) - Change to konda actor critic method mentioned in section 3.6.2

• **CartPole**

Variant	$\alpha 1$	par	γ	$\alpha 2$	par	λ	max-μ	max-σ	samp-μ	samp-σ
total	0.01	10	0.99	-	-	-	200.0	0.0	189.49	20.05
without	0.01	10	0.99	-	-	-	200.0	0.0	199.23	8.20
with	0.01	10	0.99	-	-	-	200.0	0.0	198.71	11.71
Qac	0.001	10	1	0.005	114	-	200.0	0.0	162.65	39.58
QepNorm	0.005	10	0.99	0.008	142	-	200.0	0.0	200.00	0.00
Qbaseline	0.005	10	0.99	0.01	114	-	200.0	0.0	198.69	5.65
A2C	0.001	10	0.99	0.005	226	-	200.0	0.0	192.74	14.87
TD(batch 20)	0.01	10	0.99	0.001	386	0.5	200.0	0.0	185.08	34.24
kondaTD(1)	0.005	10	0.90	0.008	8	1	200.0	0.0	200.00	0.00

Table 51: CartPole Final weights table

Variant	$\alpha 1$	par	γ	$\alpha 2$	par	λ	max-μ	max-σ	samp-μ	samp-σ
total	0.01	10	0.99	-	-	-	200.0	0.0	188.28	20.07
without	0.01	10	0.99	-	-	-	200.0	0.0	199.41	8.02
with	0.01	10	0.99	-	-	-	200.0	0.0	198.47	12.59
Qac	0.001	10	1	0.005	114	-	200.0	0.0	158.20	41.47
QepNorm	0.005	10	0.99	0.008	142	-	200.0	0.0	199.20	11.36
Qbaseline	0.005	10	0.99	0.01	114	-	200.0	0.0	198.94	4.53
A2C	0.001	10	0.99	0.005	226	-	200.0	0.0	192.00	15.24
TD(batch 20)	0.01	10	0.99	0.001	386	0.5	200.0	0.0	185.08	34.24
kondaTD(1)	0.005	10	0.90	0.008	8	1	200.0	0.0	185.94	33.39

Table 52: CartPole Optimal Weights table

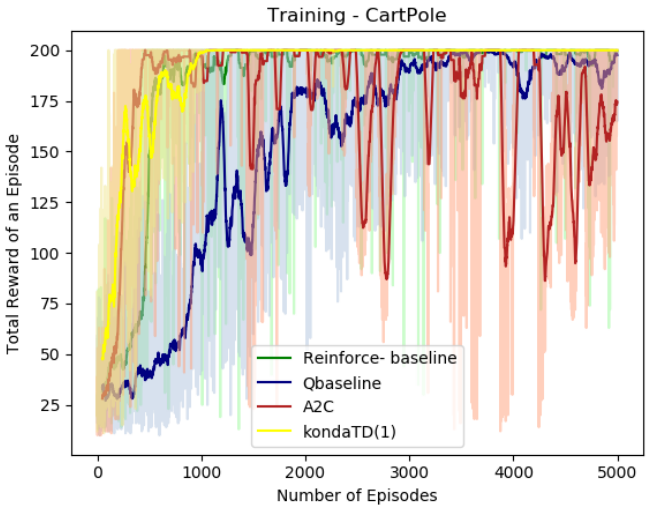


Figure 24: Comparison of training of some the main variants

- **Acrobot**

Variant	$\alpha 1$	par	γ	$\alpha 2$	par	λ	max-μ	max-σ	samp-μ	samp-σ
total	0.01	21	0.99	-	-	-	-102.68	27.23	-111.94	40.24
without	0.01	21	0.99	-	-	-	-86.74	26.48	-85.11	24.33
with	0.01	21	0.99	-	-	-	-85.97	28.91	-90.24	35.01
Qac	0.005	21	1	0.05	163	-	-89.12	25.14	-89.14	27.41
Qep	0.001	21	0.99	0.005	363	-	-95.25	21.92	-112.94	21.73
QepNorm	0.001	21	0.99	0.005	363	-	-109.53	43.50	-115.94	28.13
Qbaseline	0.005	21	0.99	0.01	323	-	-88.93	32.57	-94.43	36.50
A2C	0.001	21	0.99	0.005	323	-	-85.94	21.63	-82.57	18.66

Table 53: Acrobot Final Weights table

Variant	$\alpha 1$	par	γ	$\alpha 2$	par	λ	max-μ	max-σ	samp-μ	samp-σ
total	0.01	21	0.99	-	-	-	-82.36	22.04	-87.70	25.23
without	0.01	21	0.99	-	-	-	-85.68	24.80	-84.51	21.76
with	0.01	21	0.99	-	-	-	-86.86	27.62	-87.39	24.81
Qac	0.005	21	1	0.05	163	-	-91.16	29.16	-87.74	17.29
Qep	0.001	21	0.99	0.005	363	-	-93.79	20.69	-111.85	25.54
QepNorm	0.001	21	0.99	0.005	363	-	-109.16	25.78	-118.70	25.91
Qbaseline	0.005	21	0.99	0.01	323	-	-85.74	22.99	-93.70	36.56
A2C	0.001	21	0.99	0.005	323	-	-91.96	49.81	-87.27	25.48

Table 54: Acrobot Optimal Weights table

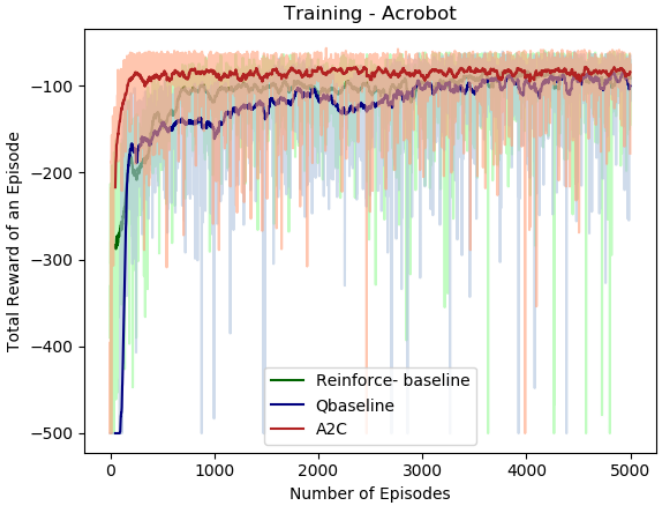


Figure 25: Comparison of training of some the main variants

• LunarLander

Variant	$\alpha 1$	par	γ	$\alpha 2$	par	λ	max-μ	max-σ	samp-μ	samp-σ
total	0.005	484	0.99	-	-	-	62.26	118.35	72.30	134.68
without	0.005	484	0.99	-	-	-	131.68	101.10	107.56	93.33
with	0.005	484	0.99	-	-	-	127.24	167.54	129.20	158.34
QepNorm	0.0008	484	0.99	0.006	4996	-	106.60	105.57	109.50	100.71
Qbaseline	0.002	36	0.99	0.008	1668	-	-48.73	23.09	96.86	48.65
A2C	0.01	36	0.99	0.0005	4996	-	191.89	104.33	193.02	101.44
A2C	0.001	836	0.99	0.005	1668	-	231.60	68.50	210.15	75.55
kondaTD(1)	0.0002	836	0.99	0.005	256	1	53.77	95.92	163.96	79.74

Table 55: LunarLander Final Weights Table

Variant	$\alpha 1$	par	γ	$\alpha 2$	par	λ	max-μ	max-σ	samp-μ	samp-σ
total	0.005	484	0.99	-	-	-	-93.98	72.65	25.85	76.26
without	0.005	484	0.99	-	-	-	145.94	56.96	142.75	54.69
with	0.005	484	0.99	-	-	-	193.39	65.02	193.45	72.58
QepNorm	0.0008	484	0.99	0.006	4996	-	14.55	126.57	13.65	116.33
Qbaseline	0.002	36	0.99	0.008	1668	-	-331.38	105.21	-9.28	63.72
A2C	0.01	36	0.99	0.0005	4996	-	201.01	100.52	192.47	101.65
A2C	0.001	836	0.99	0.005	1668	-	193.78	90.90	215.25	73.56
kondaTD(1)	0.0002	836	0.99	0.005	256	1	156.61	72.56	117.46	100.56

Table 56: LunarLander Optimal Weights Table

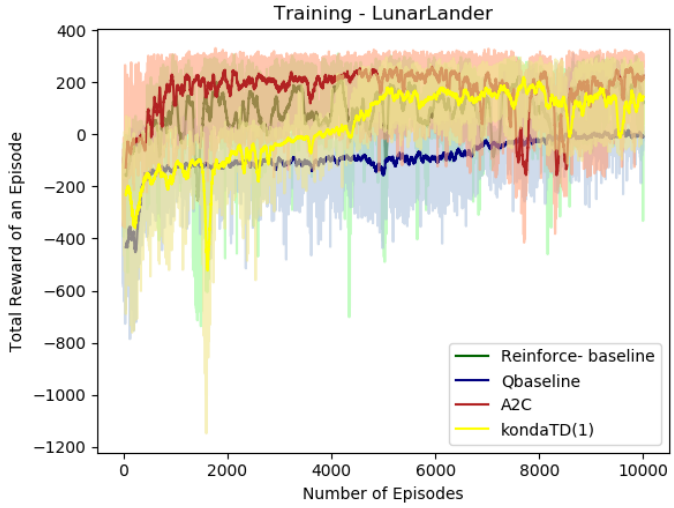


Figure 26: Comparison of training of some the main variants

Observations

- Important think to notice is that it takes lot of episode to learn for policy based and actor-critic methods. Even for simple environment like LunarLander is required as high has 10000 episode. So atari games will take days to run. Which I didn't have resources to run.
- As you can observe in REINFORCE with baseline it works better than without it, it will reduce the variance and try to put us in right direction, and taking less improper decision which would have hurt the learning.
- We can observe A2C is out performing others in terms of scoring high and learning faster but has high variance, which is problematic.

- REINFORCE with baseline and konda gives us much more stable learning and decent variance which is much appreciated.
- As you see konda for CartPole after learning is stays at 200(max it can achieve in CartPole), unlike others which deviates while learning. And konda is more promising.
- We can also see the number of parameters used for critic in konda has drastically decreased.
- Introducing a good baseline helps a lot, like in case of A2C value function as baseline works like a charm.
- Even Qbaseline is learning well, and has a stable learning as we can see in the graphs, but it still needs more number iterations to learn.
- One thing I observed is that the variant performing equal well in both max and sampling policy has stable learning, and stable learning can only happen after choosing right step size and network and a large number of episodes.
- Without proper step size and network the model will not learn, choosing the right parameters is the hardest and time consuming part of these methods, they are very sensitive to step size and network.
- Most of the time it used to converge to local optimum, as we are using gradient decent, there is no guarantee we will reach global optimum. Which I mainly faced during runing TD(λ) actor-critic and konda TD(λ) actor-critic. The only way is too to choose right step size and network through experimenting. There are no shortcut.
- Also note that critic should have larger step size than actor for better stable learning. As critic acts like a guide, without a proper guide you may not reach the goal. You can see in one of the experiment in LunarLander for A2C variant

References

Konda, V. R., & Tsitsiklis, J. N. (n.d.). Actor-critic algorithms. Retrieved from https://papers.nips.cc/paper/1786-actor-critic-algorithms.pdf

Richard S. Sutton, S. S. Y. M., David McAllester. (n.d.). Policy gradient methods for reinforcement learning with function approximation.

Tsitsiklis, J. N. (n.d.). An analysis of temporal-difference learning with function approximation. Retrieved from https://www.mit.edu/~jnt/Papers/J063-97-bvr-td.pdf

Weng, L. (n.d.). Policy gradient algorithms. Retrieved from https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html#actor-critic