CS6700: Reinforcement Learning

Assignment 2

Royyuru Sai Prasanna Gangadhar - ME19B190

Abhigyan C - EE19B146

Course Professor : Prof. Balaraman Ravindran

INDEX

March 29, 2023

Contents

List of Figures

1 Abstract

In this assignment, we explore Deep Q-Networks and Actor Critic Methods to find the best possible solution for 3 different OpenAI Gym RL environments with discrete actions - Acrobot-v1, CartPole-v1 and MountainCar-v1.

The aim is to find the best hyperparameters and configurations for the three OpenAI gym environments CartPole, Acrobot and MountainCar.

In order to keep the report concise only important plots and tables have been added in this assignment. All the plots that have been obtained during the working of assignment have been stored in a folder and zipped together along with the code files and the report. Code snippets where ever necessary have been added in this report. The performance metric that has been used to compare various models is chosen to be the number of episodes that an agent takes to solve the environment

2 Environment Description

- **Acrobot-v1:** 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.
- **CartPole-v1:** 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.
- **MountainCar-v0:** 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.

3 DQN

3.1 CartPole-v1

Now, let us implement our DQN agent on the cart pole environment.

Techniques Used: Prioritized Replay Buffer, Target network and Hyper-parameter tuning

3.2 Prioritized Sampling from Replay Buffer

One way of sampling from the replay buffer is to sample a random set of experiences of a given batch size. Instead we sample using a technique called prioritized sampling. We sample batches in proportion to the magnitude of their TD error. Thus, we preferentially learn the values of states that have been approximated poorly. This helps to speed up the training process. Priortized sampling drastically decreases the number of training episodes but increases the computational overhead considerably as well.

With prioritized sampling not only the computation increases, but also the number of hyperparameters the bias correction(a) in finding the probability and the weight decay in the importance sampling weights(b). To simplify our analysis the weight decay is considered the same as the decay in epsilon and we keep the value of a as a hyperparameter to get tuned.

3.3 Methodology followed:

Listing 1: DQN Algorithm

```
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n
env = gym.make('Acrobot-v1') #Replace this with the appropriate ...
    environment for CartPole and MountainCar

def dqn(agent,n_episodes, max_t, eps_start, eps_end, eps_decay,a):
#necessary initializations
    for i_episode in range(1, n_episodes+1):
        state = env.reset() #initalizes the environment
        score = 0
        step = 0
        for t in range(max_t):
            action = agent.act(state, eps)  #action taken in an epsilon ...
                greedy fashion
            next_state, reward, done, _ = env.step(action)
            agent.step(state, action, reward, next_state, done,a = a) ...
                #updates the replay buffer with the current experience and ...
                updates the weights of neural network
            state = next_state
            score += reward
            step += 1
            if done:
                break
        steps.append(step)
        scores_window.append(score)
        scores_window_printing.append(score)
        scores_exit_bad_paramas.append(score)

        eps = max(eps_end, eps_decay*eps)
        #necessary print statements are added, please refer to the code ...
            files submitted
        #Termination criteria
        if np.mean(scores_window)≥-100.0: #termination criteria
           print('\nEnvironment solved in {:d} episodes!\tAverage Score: ...
               {:.2f}'.format(i_episode-100, np.mean(scores_window)))
           break
    return [np.array(steps),np.array(scores),i_episode,False]
```

Listing 2: Prioritize replay buffer

```

    def modify_priorities(self,indices,TD_errors,offset=0.001):#
        #updates the value of the priorities of experiences
        np.put(self.priorities ,indices, np.abs(TD_errors).flatten()+offset)

    def return_probabilities(self, a):#a = priority scale
        #returns the probability distribution of the priorities of each ...
            experience, with a as the bias correction factor
        arr = np.array(self.priorities)**a   #modify a
        probab = arr/arr.sum()
        return probab

    def return_importance_weights(self,probabilities):
        #return importacne weights correspnding to the current probability ...
            distribution
```

```
        importance_weights = 1/(probabilities*len(self.memory))
        normalized_weights = importance_weights/max(importance_weights)
        return normalized_weights

    def sample(self,batch_size=None,a=1.0):
        """Randomly sample a batch of experiences from memory."""
        num_samples = self.batch_size if batch_size ==None else batch_size
        probab      = self.return_probabilities(a)
        get_sample_indxs = np.random.choice(np.arange(len(self.memory)), ...
            size = num_samples,replace = False,p = probab)
        importance_weights = ...
            self.return_importance_weights(probab[get_sample_indxs])
        experiences = self.memory[get_sample_indxs]
        states = torch.from_numpy(np.vstack([e[0] for e in experiences if ...
            e is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e[1] for e in experiences if ...
            e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e[2] for e in experiences if ...
            e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e[3] for e in ...
            experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e[4] for e in experiences if e ...
            is not None]).astype(np.uint8)).float().to(device)
        return (states, actions, rewards, next_states, ...
            dones),importance_weights,get_sample_indxs
```

Listing 3: Target Q network

```
class TutorialAgent():

    def __init__(self, state_size, action_size, ...
        seed,num_layers,neurons_each_layer,lr,gamma,update_every,batch_size
    ,max_tRUNCATION):

        ''' Agent Environment Interaction '''
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)
        self.batch_size = batch_size
        self.gamma = gamma
        self.MAX_TRUNCATION = max_tRUNCATION
        self.update_every = update_every
        ''' Q-Network '''
        self.qnetwork_local = QNetwork1(seed=seed, state_size= state_size, ...
            action_size=action_size ...
            ,num_layers=num_layers,neurons_each_layer=neurons_each_layer).
        to(device)
        self.qnetwork_target =QNetwork1(seed=seed, state_size= state_size, ...
            action_size=action_size ...
            ,num_layers=num_layers,neurons_each_layer=neurons_each_layer).
        to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=lr)

        ''' Replay memory '''
        self.memory = ReplayBuffer(action_size, int(1e5), self.batch_size, ...
            seed=seed)
```

Initial setting: We start our analysis with the network architecture and network hyper parameters given in the tutorial 5. We compare the performance of our model with respect to

the performance of this model given in tutorial 5. We will initially start with a random set of hyper parameters and then tune them to reach an optimal performance The following are the hyper parameters that are tuned in training a DQN Agent

- Epsilon decay
- Gamma
- Learning rate
- Number of layers of the neural network
- Number of neurons in each layer
- Update frequency of target network
- a = bias correction in priority sampling

To be concise, only the graphs corresponding to the initial set of hyper parameters and final graphs corresponding to optimal values of hyper parameters have been illustrated in this report. However, a complete summary of Hyper parameter tuning of CartPole-v1 has been displayed in the Figure 1.

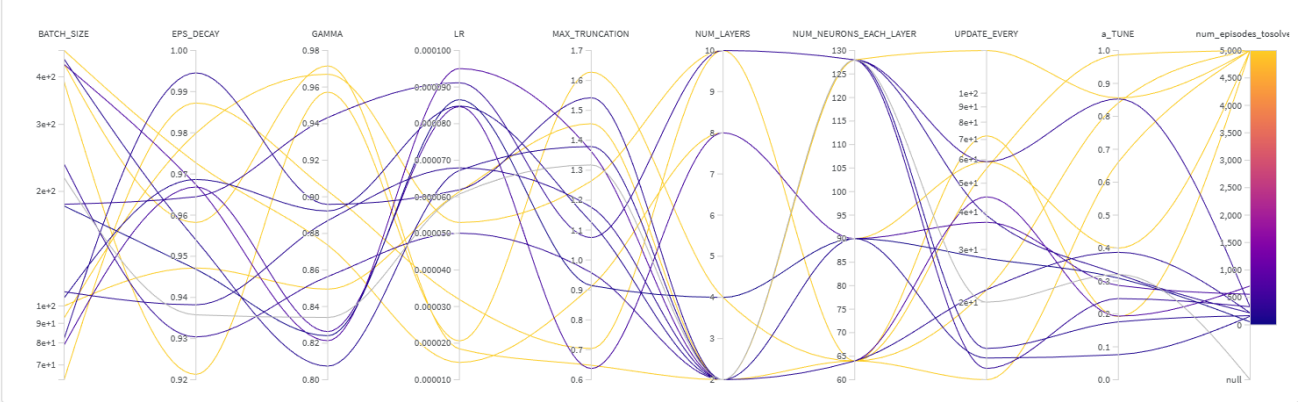


Figure 1: Summary of the results of hyperparameter tuning for CartPole-v1

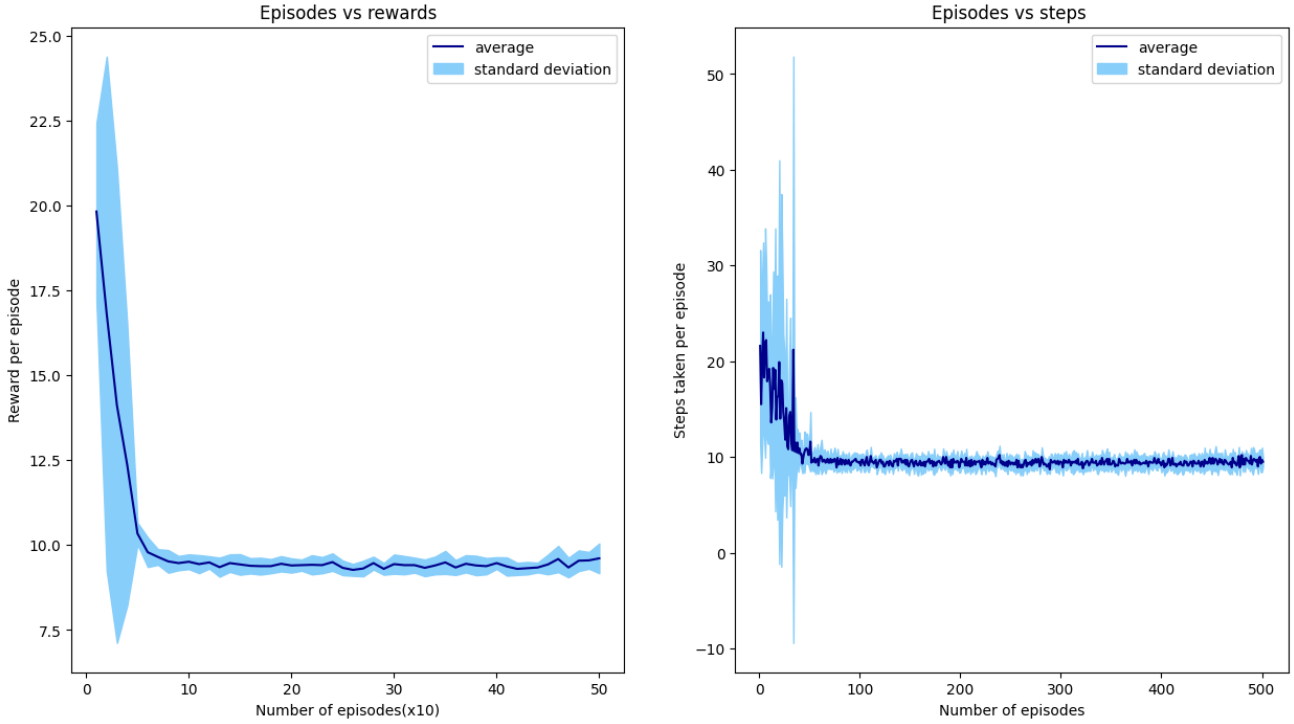


Figure 2: Initial Hyper parameter setting

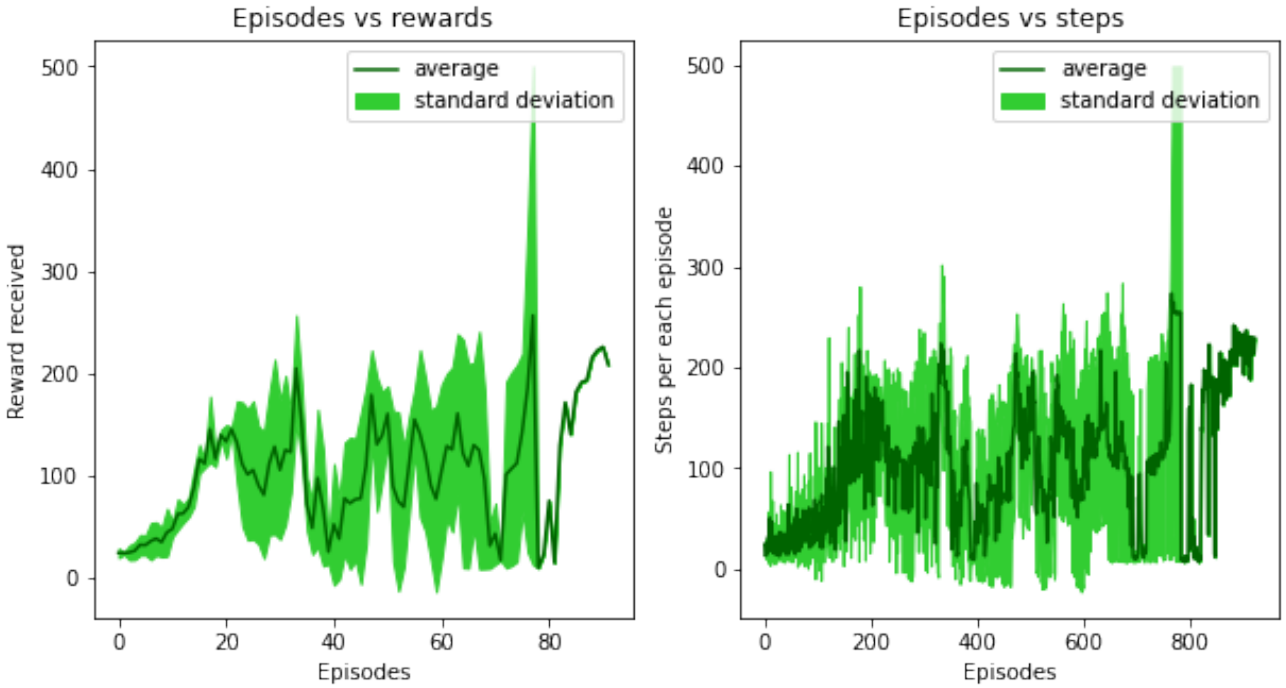


Figure 3: Performance of the tutorial DQN agent

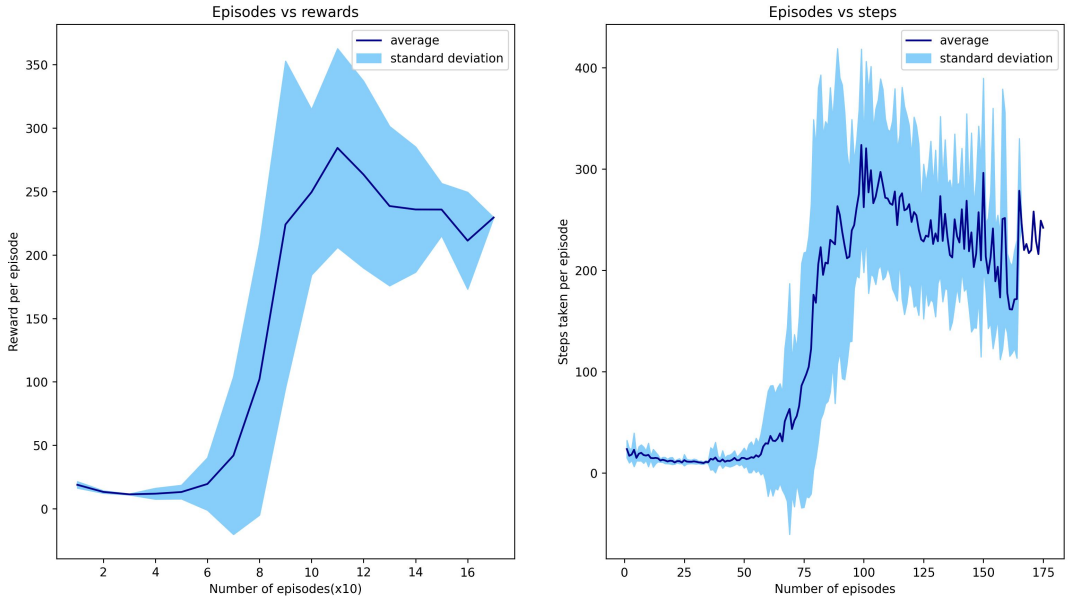


Figure 4: Best Hyper parameter setting for Cartpole DQN

State	Batch size	Epsilon decay	Gamma	Learning rate	Num network layers	Neurons each layer	Update frequency (Target network)	a(bias correction)	Priortized sampling	Number of episodes to solve the env
Initial Hyperparameter setting	432	0.958	0.971	1.84e-05	2	64	11	0.648	Yes	Did not solve!
Tutorial Agent	64	0.995	0.99	5e-4	2	64	20	None	No	685
Optimal Agent	183	0.946	0.8241	8.6e-5	4	90	28	0.308	Yes	54

3.4 Acrobot-v1

Methodology Followed:
The methodology followed here is the same as that followed in the Cartpole environment, except, we have the following initial hyperparameters:
Following is the summary of the hyperparameters over the course of their tuning:

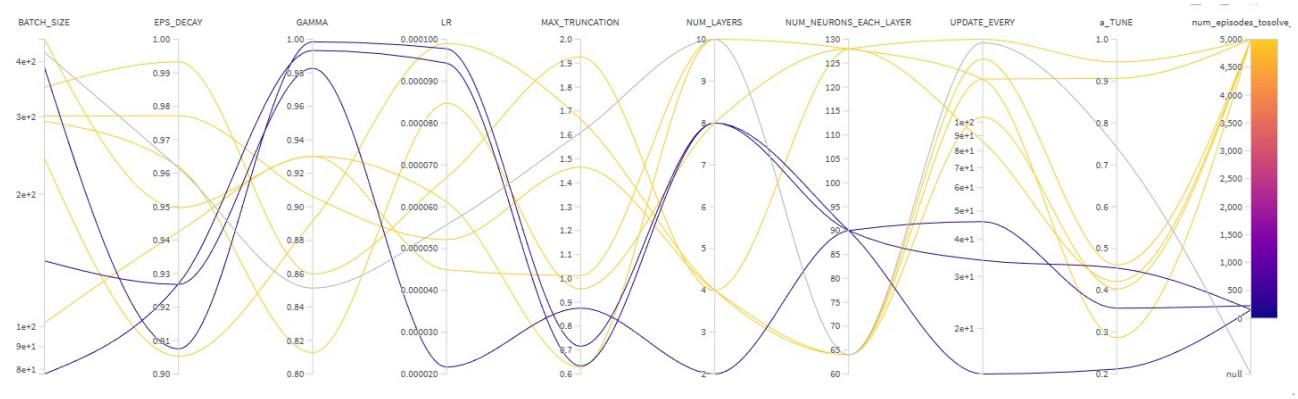


Figure 5: Summary of the results of hyperparameter tuning for Acrobot-v1

State	Batch size	Epsilon decay	Gamma	Learning rate	Num network layers	Neurons each layer	Update frequency (Target network)	a(bias correction)	Priortized sampling	Number of episodes to solve the env
Initial Hyperparameter setting	450	0.949	0.93	4.49e-05	10	64	140	None	No	Did not solve!
Tutorial Agent	64	0.995	0.99	5e-4	2	64	20	None	No	685
Optimal Agent	388	0.90	0.9985	9.774e-5	8	90	14	None	NO	150

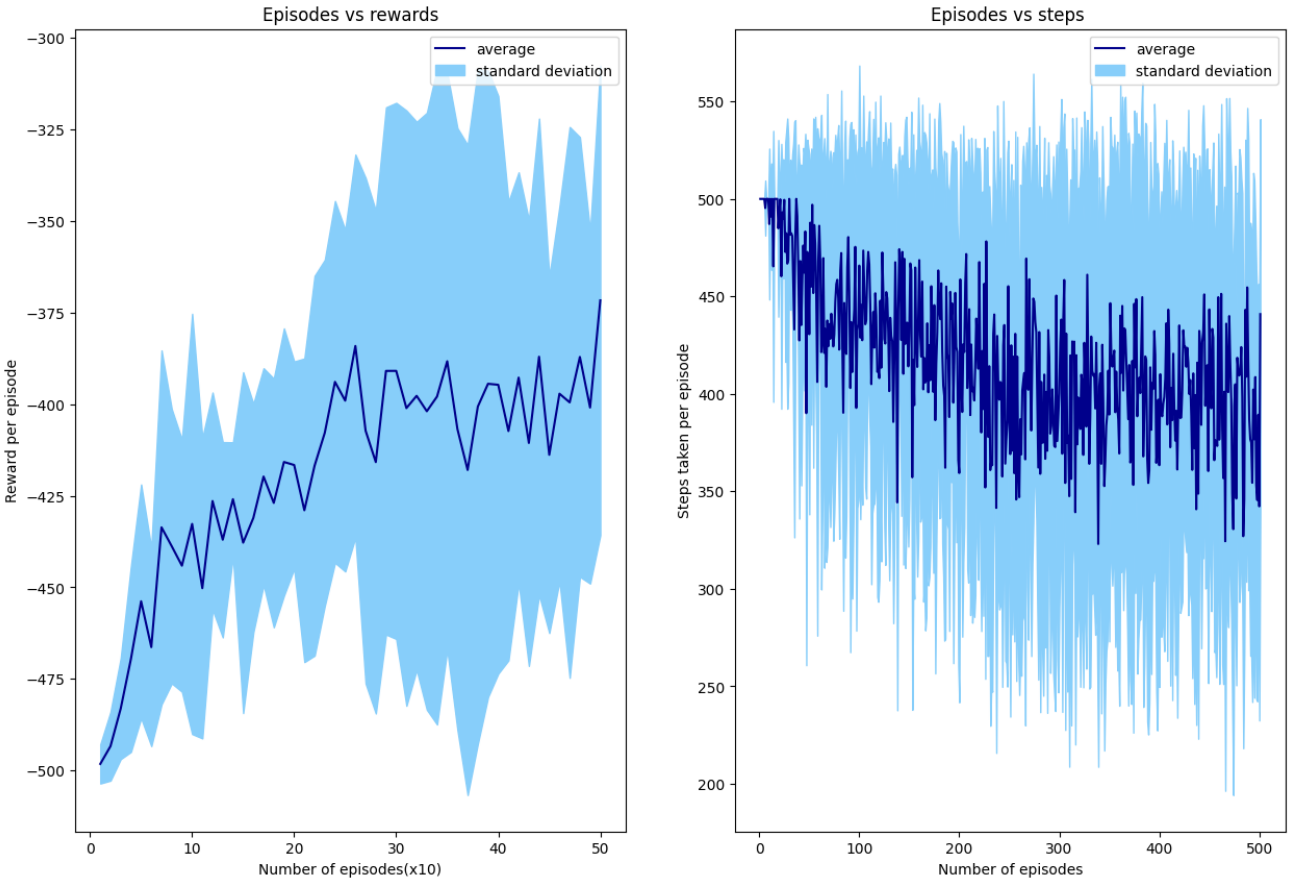


Figure 6: Initial Hyper parameter Settings for Acrobot

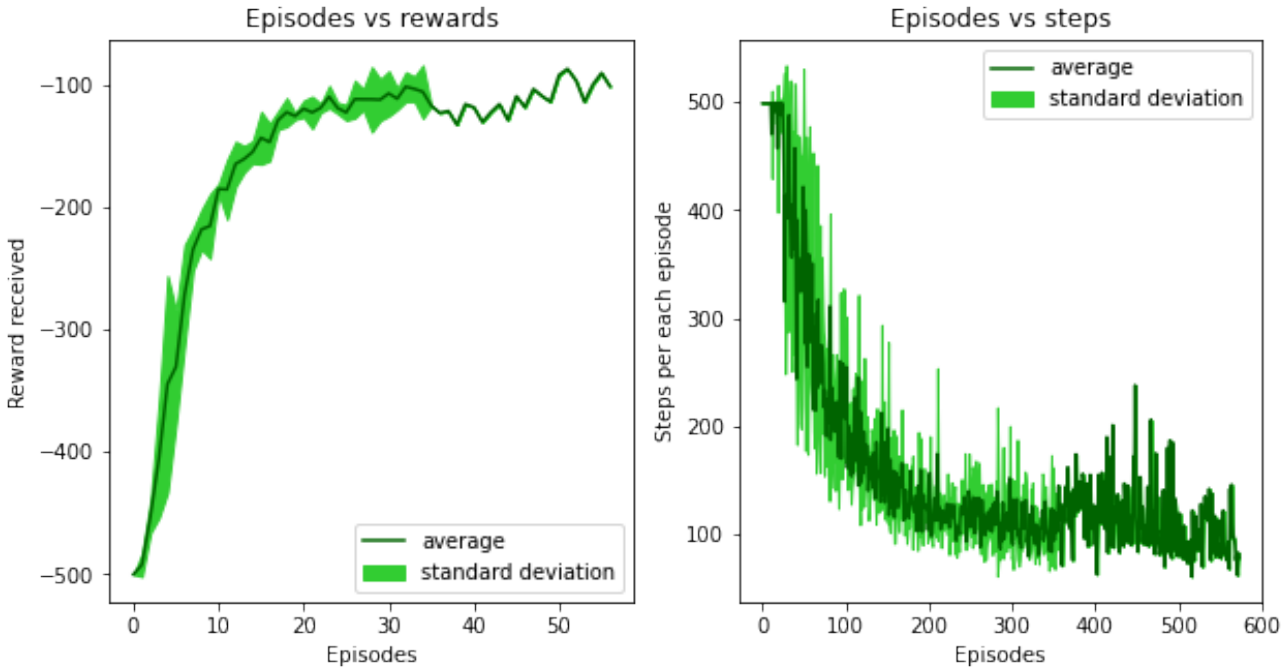


Figure 7: Performance of Acrobot with respect to tutorial performance

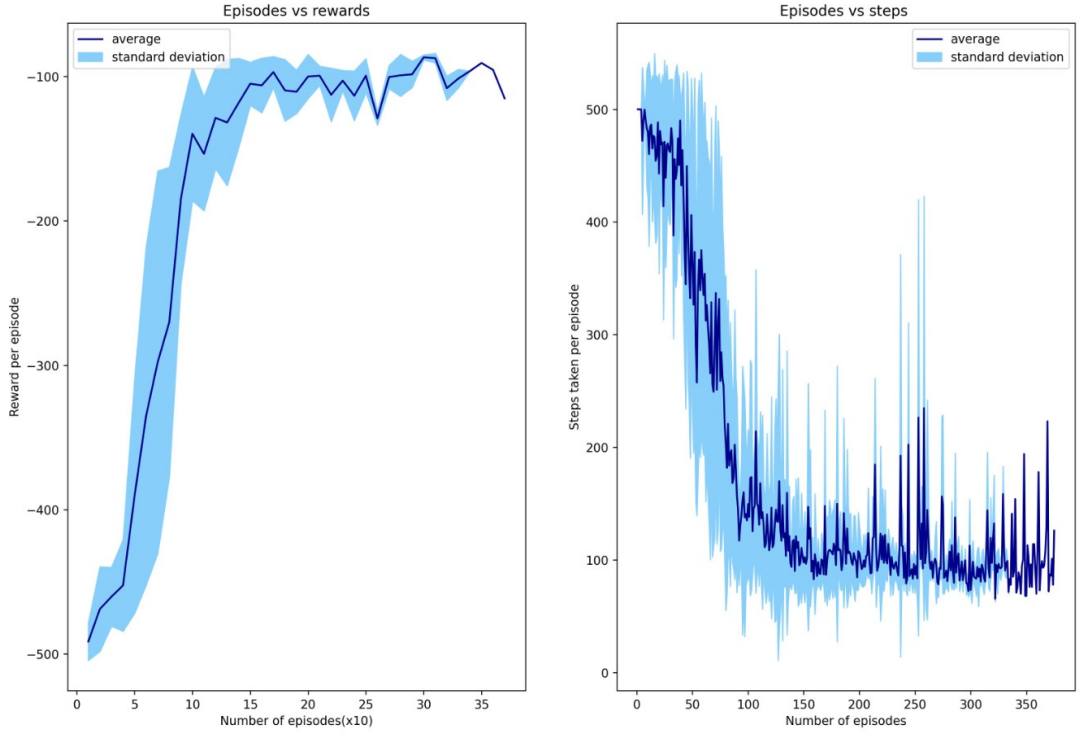


Figure 8: Best Hyper parameter setting for Acrobot using DQNs

We are able to reach a performance,where the agent in able to solve the environment in 154 episodes over a average trail of 154 runs.

3.5 MountainCar-v0

3.6 Discretization of State Space for mountain car

In the MountainCar experiment, due to the low speed of convergence, the model did not converge easily. We therefore chose to discretize the state space (varying from -1.2 to 0.6) into steps of 0.1, leading to 19 discrete states. This might help the agent to learn better in continuous state spaces, as the agent is having very limited exploration (with a maximum time steps of 200) in each episode. We also in the same way multiply the velocity by 50 and round it of to nearest integer to discretize it. We have also increased the truncation steps to 500 instead of 200.
The agent now learns a value function over these states with the help of a Deep-Q Network. Let us compare our results before and after implementation of discrete state spaces.

Listing 4: Discretization of continuous states for MountainCar-v0

```
for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        step = 0
        for t in range(max_t):
            state_adj = (state-env.observation_space.low)*np.array([10,50])
            state_adj = np.round(state_adj,0).astype(int)
            action = agent.act(state_adj, eps)
            next_state, reward, done, _ = env.step(action)
            next_state_adj = ...
                (next_state-env.observation_space.low)*np.array([10,50])
            next_state_adj = np.round(next_state_adj,0).astype(int)
            agent.step(state_adj, action, reward, next_state_adj, done,a = a)
            state = next_state
            score += reward
            step += 1
            if done:
                break
```

State	Batch size	Epsilon decay	Gamma	Learning rate	Num network layers	Neurons each layer	Update frequency (Target network)	Discrete States	Priortized sampling	Number of episodes Avg reward
Initial Hyperparameter setting	314	0.90	0.81	0.002	2	64	15	No	No	-500
Tutorial Agent	64	0.995	0.99	5e-4	2	64	20	No	No	-300
Optimal Agent	64	0.995	0.99	5e-4	2	64	20	Yes	NO	-150

Before discrete state space implementation: The following picture illustrates the learning of Mountain car before state space discretization

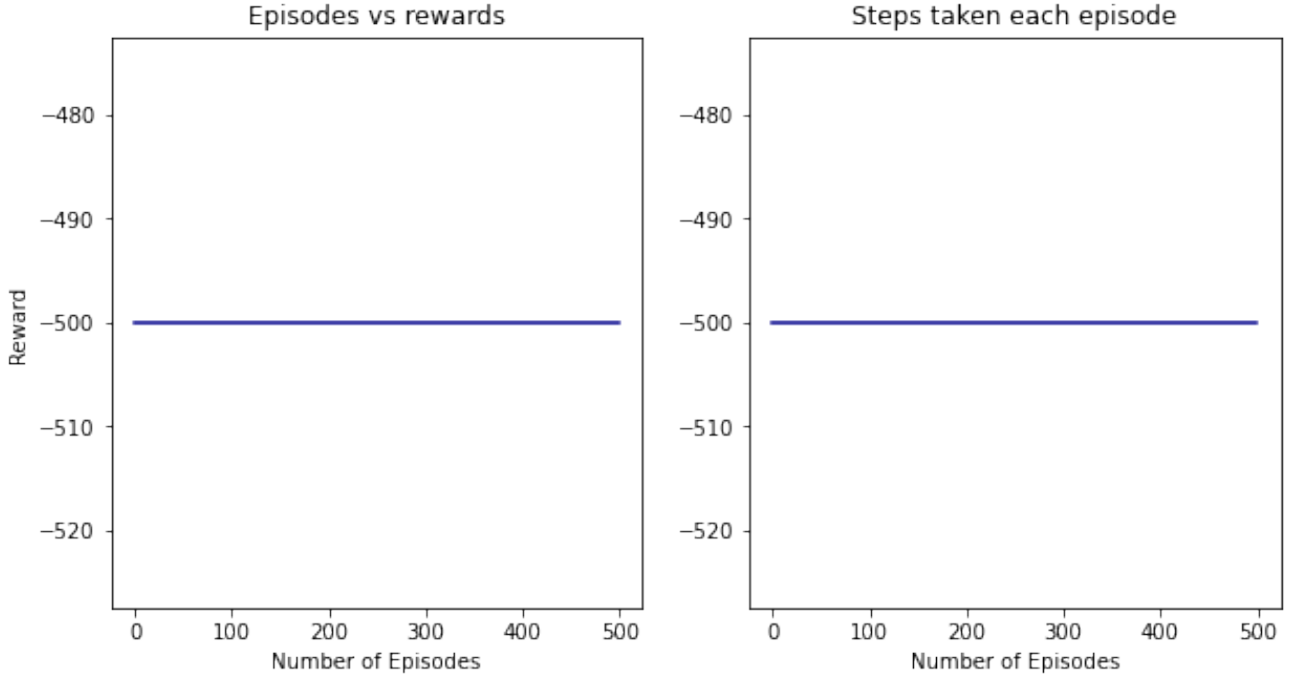


Figure 9: Initial setting of Acrobot

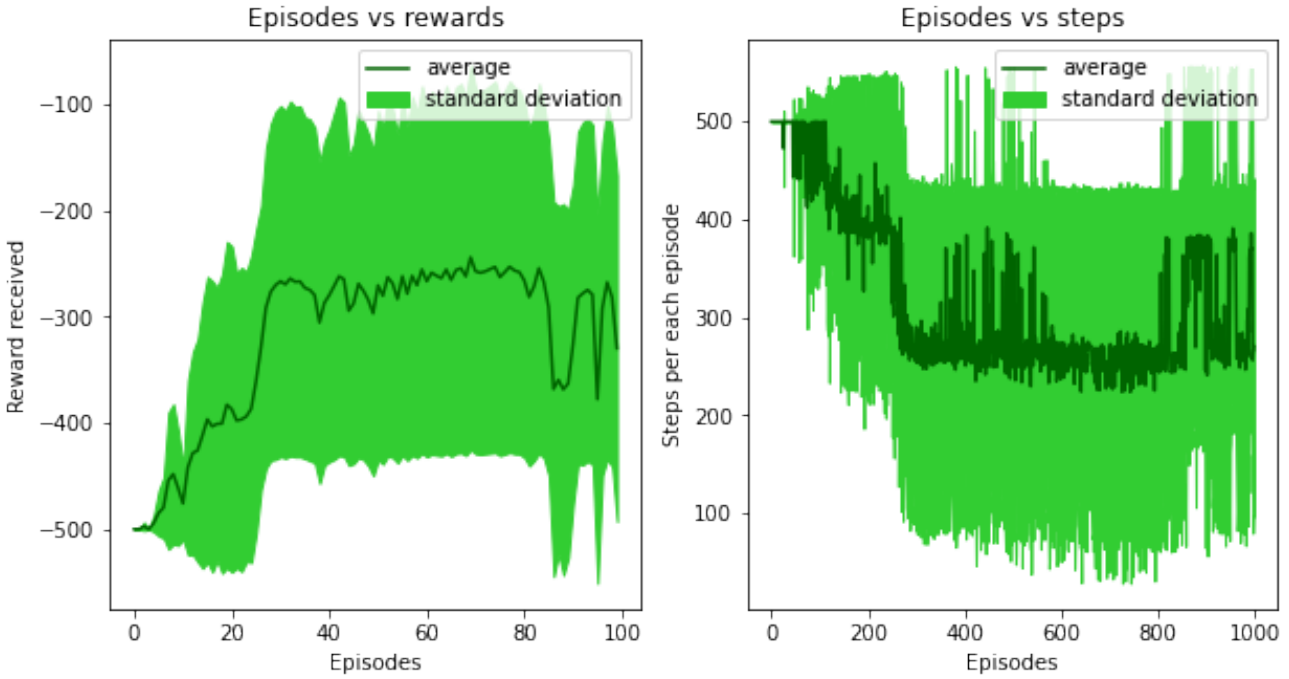


Figure 10: Performance with respect to the tutorial hyperparameters

Results for mountain car after state space discretization: The results are much better when we discretize the state space for mountain car, when the training is done for 5000 episodes. The truncation steps of mountain car has been increased to 500.

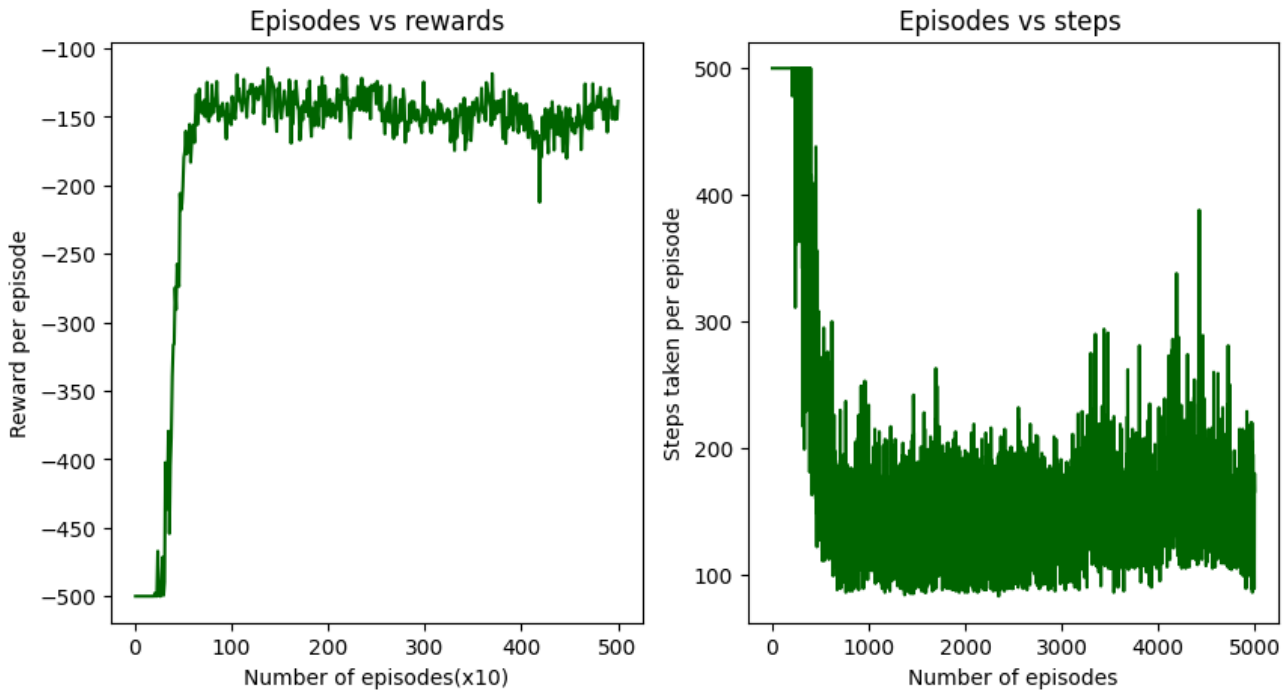


Figure 11: Success with the use of state space discretization for mountain car

Although not illustrated in this report, some sucess compared to normal training of DQN is observed when we use priortized sampling for mountain car. The learning through this approach is very very slow and not very significant (reached a average reward value of approximately -400).

4 Actor-Critic

Listing 5: Neural netework architechture that acts as "Actor Critic"

```
class ActorCriticModel(tf.keras.Model):
    def __init__(self, ...
        action_size,num_layers=2,num_neurons_each_layer=[1024,512]):
        super(ActorCriticModel, self).__init__()
        #defining the hidden layers of the model
        self.num_layers  =num_layers
        self.neurons_each_layer = num_neurons_each_layer
        self.linears = [tf.keras.layers.Dense(self.neurons_each_layer[0], ...
            activation=tf.nn.relu)]
        self.linears.extend([tf.keras.layers.Dense(self.neurons_each_layer[i], ...
            activation=tf.nn.relu) for i in range(1, self.num_layers)])
        #outputs the policy and the value of the state
        self.pi_out = tf.keras.layers.Dense(action_size, ...
            activation=tf.nn.softmax)
        self.v_out = tf.keras.layers.Dense(1)

    def call(self, state):
        h = None
        for i in range(self.num_layers+1):
          if i == 0:
             h = tf.nn.relu(self.linears[0](state))
          elif i < self.num_layers:
             h = tf.nn.relu(self.linears[i](h))
          else:
             return self.pi_out(h), self.v_out(h)
```

The one step TD loss and learning the weights from the observed TD loss is defined as below.

Listing 6: Learn function

```
    def actor_loss(self, action, pi, Δ):  #actor loss

        return -tf.math.log(pi[0,action]) * Δ

    def critic_loss(self,Δ):  #critic loss
        return Δ**2

    @tf.function #function used to update the weights of the network.
    def learn(self, state, action, reward, next_state, done):

        with tf.GradientTape(persistent=True) as tape:
            pi, V_s = self.ac_model(state)
            _, V_s_next = self.ac_model(next_state)

            V_s = tf.squeeze(V_s) #get the value of current and next ...
               states using Actorcritic neural network
            V_s_next = tf.squeeze(V_s_next)
            #TD return update
            Δ = reward+((self.gamma)*V_s_next) - V_s
            loss_a = self.actor_loss(action, pi, Δ)
            loss_c = self.critic_loss(Δ)
            loss_total = loss_a + loss_c

        gradient = tape.gradient(loss_total, ...
           self.ac_model.trainable_variables)
        self.ac_model.optimizer.apply_gradients(zip(gradient, ...
           self.ac_model.trainable_variables))
```

The full TD return algorithm for actor critic is defined as follows

Listing 7: Full TD return

```
def learn_full_return(self, states, actions, rewards, next_states, dones,T):
#get the complete trajectory and iteratively loop over each time and ...
    calculate the complete return and then accumulate it in reward.
      with tf.GradientTape(persistent=True) as tape:
          Δ_ts = []
          pis = []
          for t in range(0,T):
            Δ_t = 0
            for t_ in range(t,T):
              Δ_t += (((self.gamma)**(t_-t))*rewards[t_+1] if t_+1< ...
                 len(rewards) else 0) #calculating the total return

            pi, V_s = self.ac_model(states[t]) #get the value of current state
            V_s = tf.squeeze(V_s)
            Δ_t = Δ_t - V_s    #Full TD return
            Δ_ts.append(Δ_t)
            pis.apend(pi)
          loss_a = 0
          loss_c = 0
          for i in range(T): #loop through all time steps and accumulate ...
             the actor loss and critic loss
            loss_a += self.actor_loss(actions[i], pis[i], Δ_ts[i])
            loss_c +=self.critic_loss(Δ_ts[i])
          loss_total = loss_a + loss_c
```

```
        gradient = tape.gradient(loss_total, ...
            self.ac_model.trainable_variables, ...
            unconnected_gradients=tf.UnconnectedGradients.ZERO)
        self.ac_model.optimizer.apply_gradients(zip(gradient, ...
            self.ac_model.trainable_variables))
```

Listing 8: N step TD return

```
def learn_nstep_return(self, states, actions, rewards, next_states, dones, ...
    n, T):
#get the entire trajectory but at each time step, only accumulate the ...
    rewards till t+n and approximate the remaining return with the state ...
    s(t+n) and then calculate the n-step return
        with tf.GradientTape(persistent=True) as tape:
            Δ_ts = []
            pis = []
            for t in range(0,T):
              Δ_t = 0
              for t_ in range(t,t+n-1):
                Δ_t += (((self.gamma)**(t_-t))*rewards[t_+1] if t_+1< ...
                    len(rewards) else 0)
              pi, V_s = self.ac_model(states[t])
              _, V_tn = self.ac_model(states[t+n]) if t+n < T else 0,0
              Δ_t += (self.gamma**n*V_tn - V_s)
              V_s = tf.squeeze(V_s)
              pi, V_s = self.ac_model(states[t])
              Δ_t = Δ_t - V_s
              Δ_ts.append(Δ_t)
              pis.append(pi)
            loss_a = 0
            loss_c = 0
            for i in range(T):
              loss_a += self.actor_loss(actions[i], pis[i], Δ_ts[i])
              loss_c += self.critic_loss(Δ_ts[i])
              loss_total = loss_a + loss_c
              gradient = tape.gradient(loss_total, ...
                  self.ac_model.trainable_variables, ...
                  unconnected_gradients=tf.UnconnectedGradients.ZERO)
              self.ac_model.optimizer.apply_gradients(zip(gradient, ...
                  self.ac_model.trainable_variables))
```

Since, we are averaging over multiple runs, we need to be careful of the averaging strategy as not all runs end at the same termination value.

Listing 9: Calucating the mean of various arrays that have different lengths

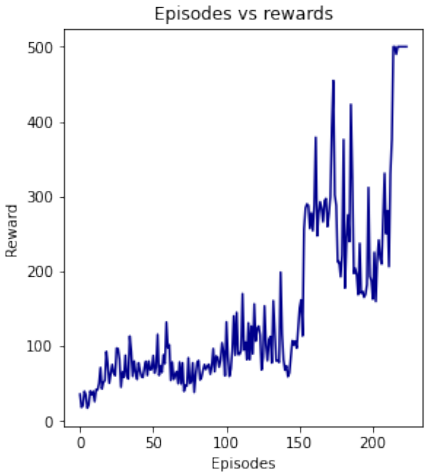
```
def calculate_mean(array):
    lens = [len(i) for i in array]
    arr = np.ma.empty((np.max(lens),len(array)))
    arr.mask = True
    for idx, l in enumerate(array):
        arr[:len(l),idx] = l
    return arr.mean(axis = -1), arr.std(axis=-1)
```

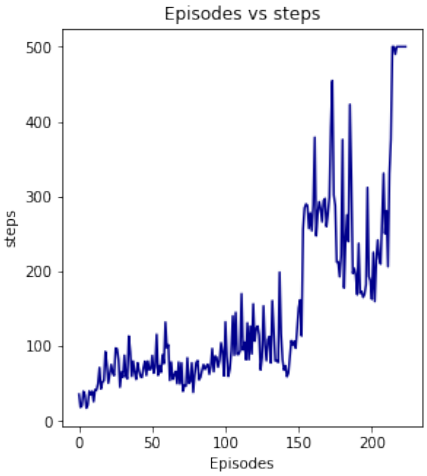
4.1 CartPole-v1

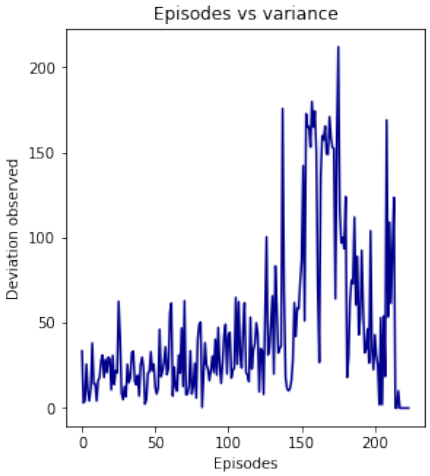
The Cartpole algorithm is run for various actor critic methods, including one step actor critic, n step actor critic and full returns actor critic. By intuition, one step actor critic has a higher

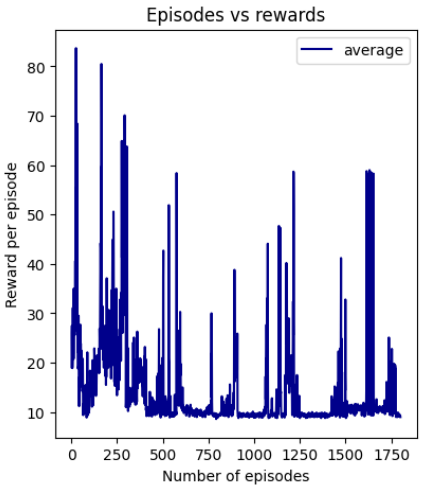
variance compared to n-step actor critic and n-step has a higher variance compared to full returns actor critic, let us compare our results.
The graphs are plotted for the following configurations:

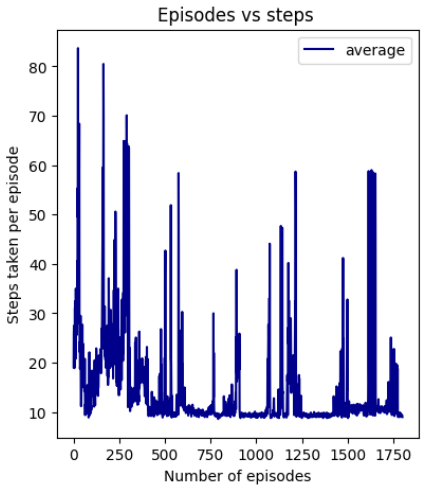
State	Learning rate	Num_episodes	Num_layers	Num_neurons_each_layer
Tutorial agent	1e-05	1800	2	1024,512
Hyperparameter setting 1	8.8e-05	1800	08	724
Hyperparameter setting 2	8.62e-05	2000	02	724
Hyperparameter setting 3	4.66e-05	1800	08	724

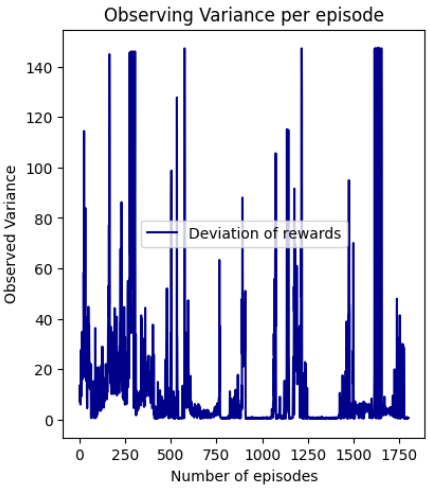


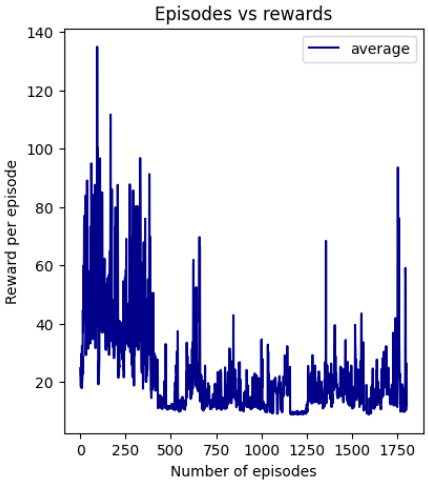


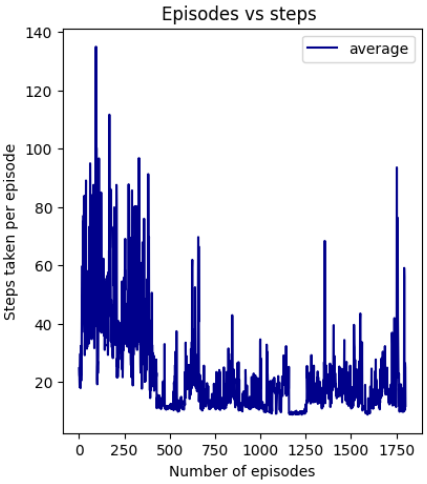


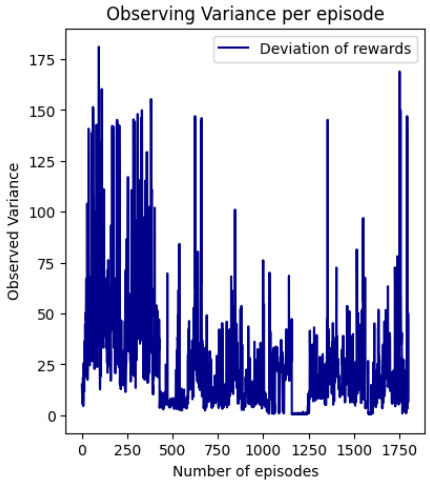












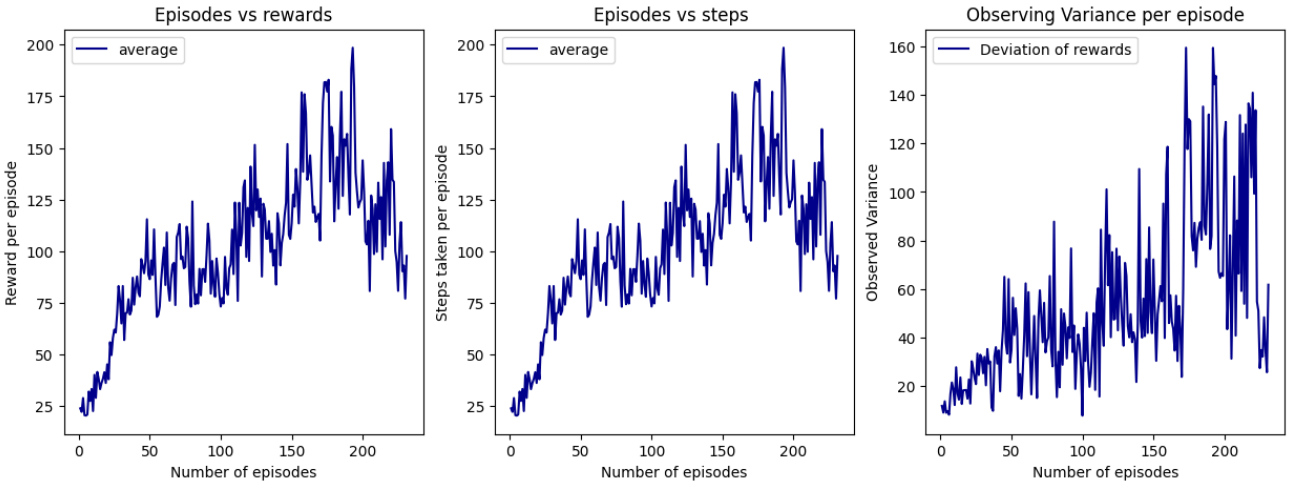


Figure 12: Actor Critic Method on the OpenAI Cartpole-v1 Environment for 3 different runs

Now we plot the best full returns for cartpole:

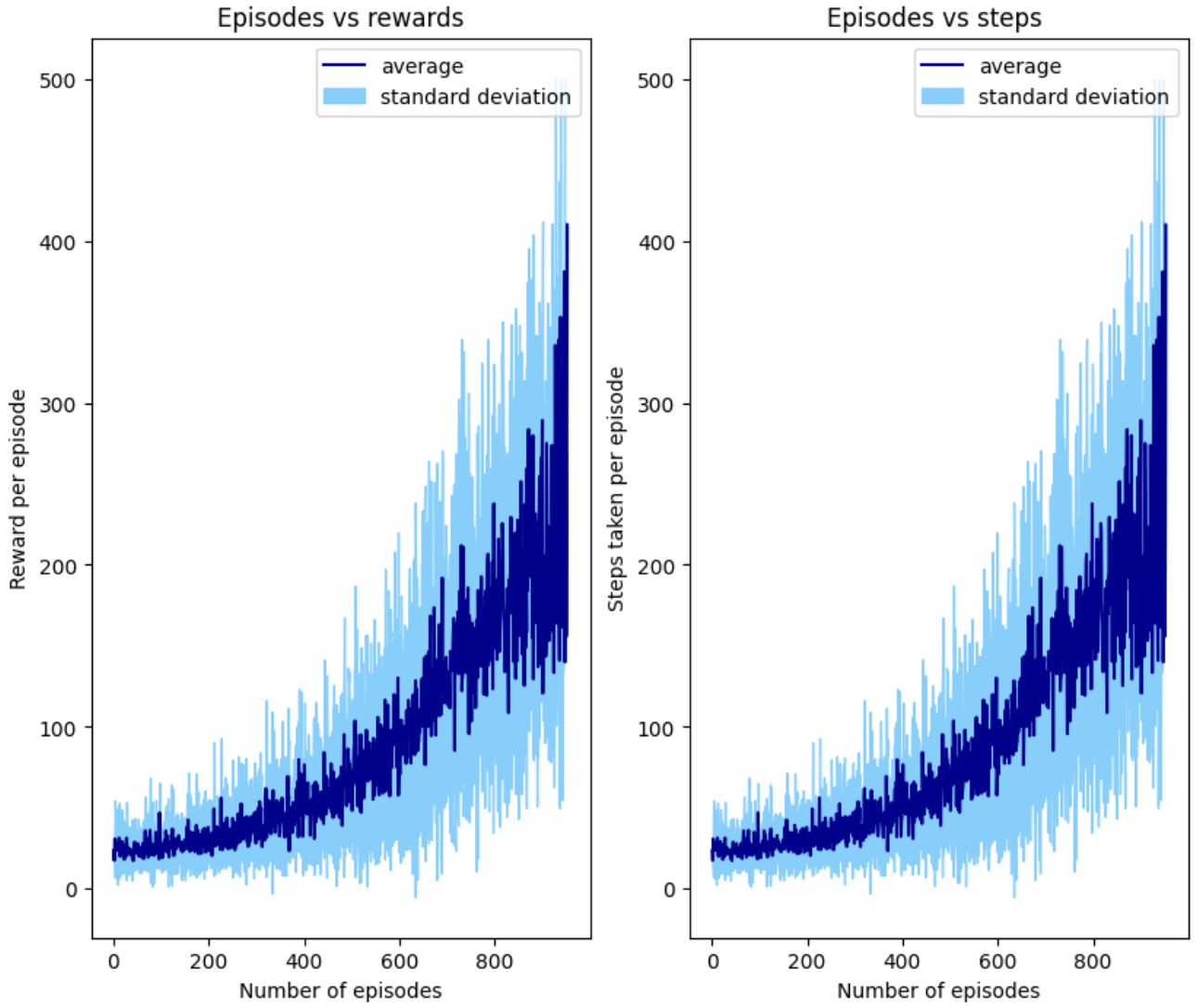


Figure 13: Best full return for Cartpole

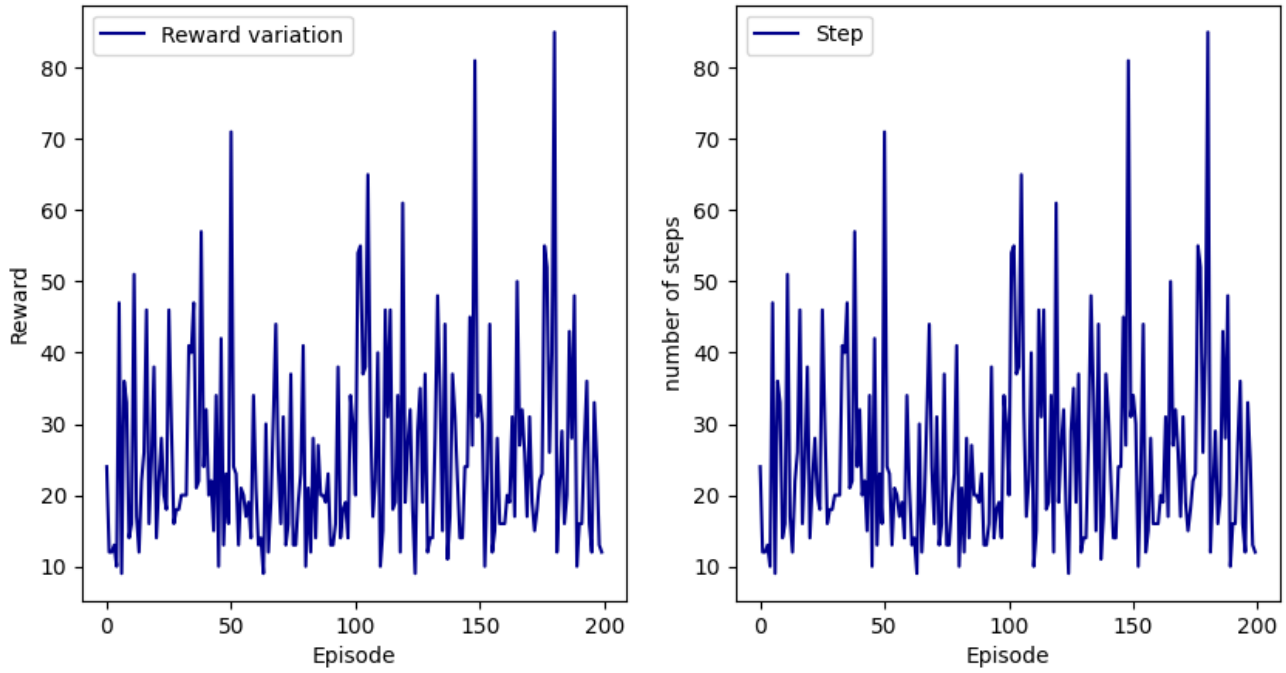
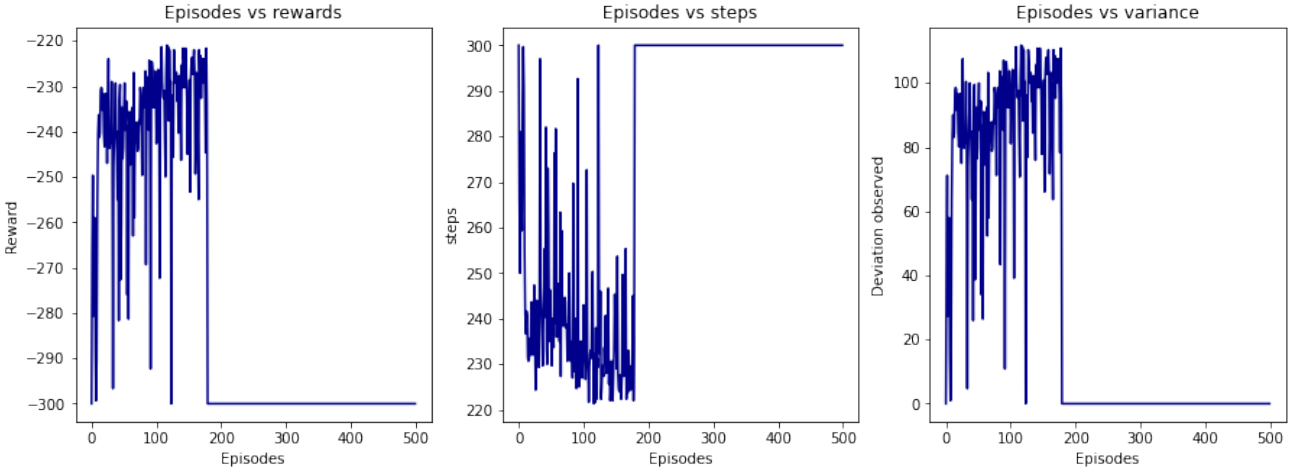


Figure 14: Actor Critic using N-step return for Cartpole-v1

4.2 Acrobot-v1

Actor-critic models have been successful on a wide range of reinforcement learning tasks, but they may not perform as well on certain tasks, such as in this case with the acrobot-v1 environment. The Acrobot-v1 environment is quite complicated and exploring to find the winning trajectory proves to be quite tough for the Actor-Critic Model with only a few runs (owing to limited computational resources and time).
Hence, we have the following trajectories, with only 1 of the 3 giving us results that result in eventual convergence:

State	Learning rate	Num_episodes	Num_layers	Num_neurons_each_layer
Tutorial agent	1e-05	1800	02	1024,512
Hyperparameter setting 1	4.8e-04	2000	08	128
Hyperparameter setting 2	1e-04	500	02	128



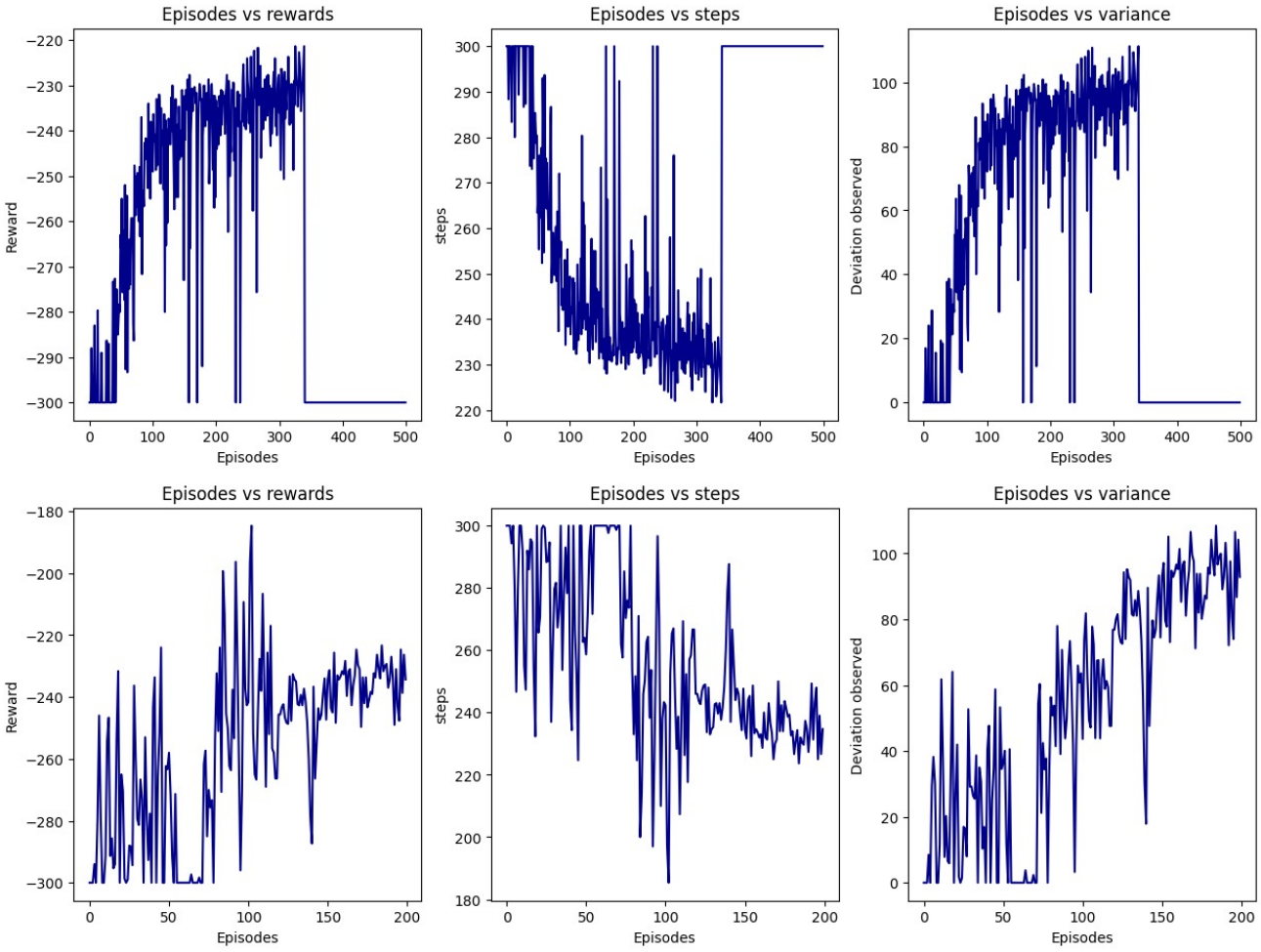


Figure 15: Actor Critic Method on the OpenAI Acrobot-v1 Environment for 3 different runs
Now we plot the best full returns for cartpole:

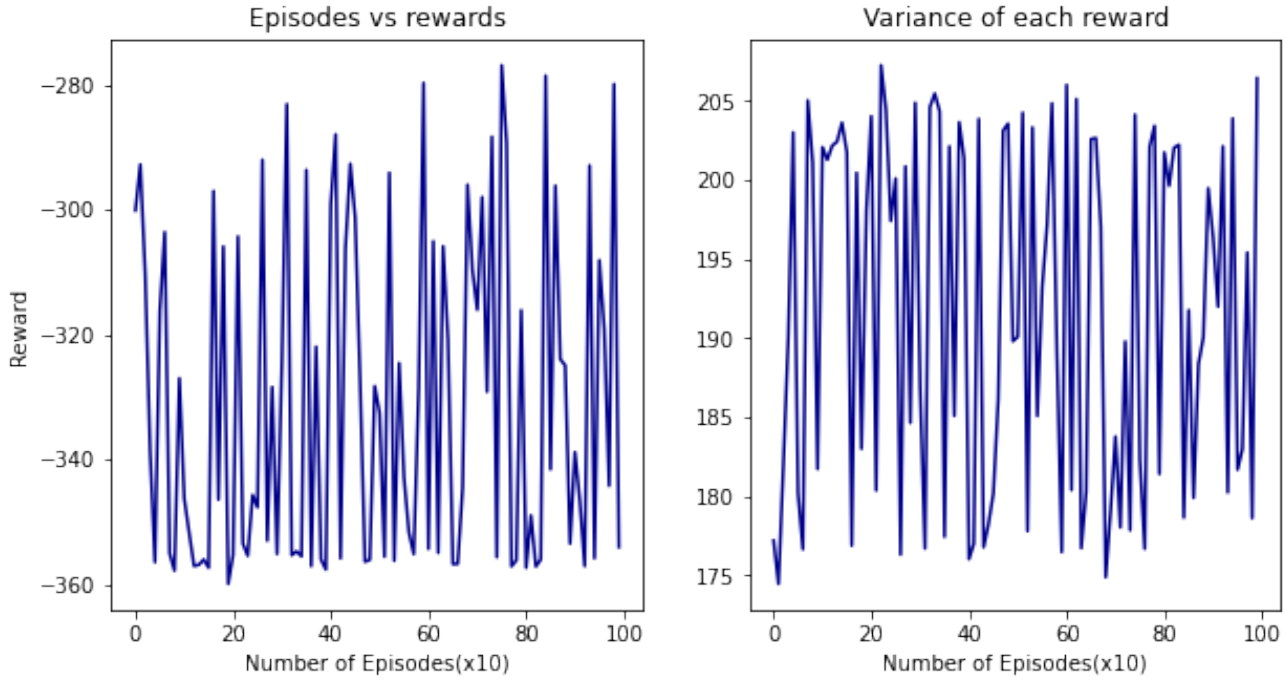


Figure 16: Actor Critic using Full-step return for Acrobot-v1

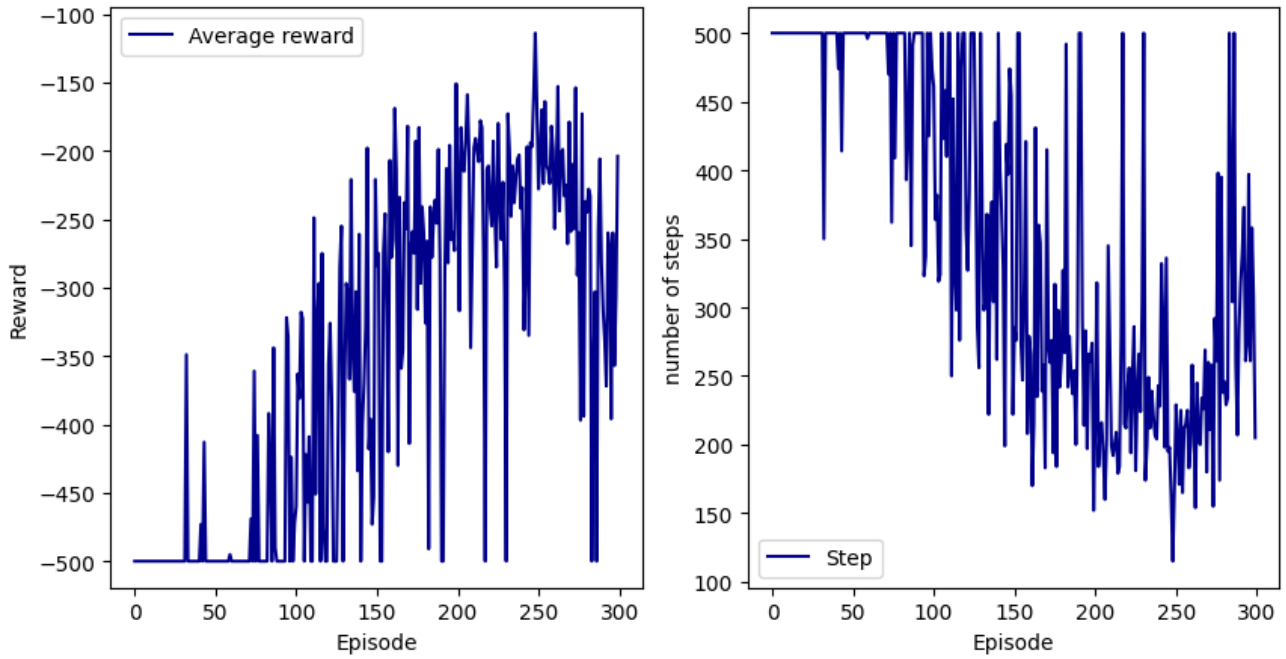


Figure 17: Actor Critic using N-step return for Acrobot-v1

4.3 MountainCar-v0

In the MountainCar-v0 environment of OpenAI Gym, the agent controls a car that must reach the top of a hill, but its engine is not powerful enough to climb the hill directly. Instead, the agent must learn to rock back and forth to gain enough momentum to get to the top. This reward function is rather tough to optimize due to the tricky shape - the actor critic model is unable to generalize to this environment as a result of this trickiness and often ends up criticizing winning algorithms.
Owing to the computational complexity, we were unable to get the Actor-Critic model to converge despite trying a large amount of hyperparameters.
Many hyperparameter settings were tried for mountain car and many of them did not even budge and had a constant reward of -200 when the system is set to truncate at 200 steps and has a constant reward of -500 when the system is set to truncate at -500.
Mountain car again did not improve its performance in one step, n-step Actor critic and full return actor critic and has been constantly maintaining a return of -200, even multiple configurations of the network and truncation criteria are used and thus has not been plotted.

5 Conclusions

We found much better results with the Deep-Q Networks than with the Actor-Critic model.
Deep-Q Learning in general is more effective in learning in discrete action environments with low-dimensional state spaces, whereas Actor-Critic would be more effective with continuous action and with more complex state spaces.
For cartpole algorithm, it has been observed that out all hyperparamter settings learning rate is the most important hyperparamter of all, followed by batch size of experiences, which is followed by the update frequency of the target network.
For Acrobot algorithm, number of neurons each layer, maximuim value of gradients to which the magnitude of gradients truncated, gamma, and update frequency are some of the important parameters for acrobot environment.
The success of Moutaincar environment depends on the discretization of state space. Results

have been significantly improved in case of mountain car if discretization of state space has been employed.
The below table gives the importance of various hyperparameters in decreasing order of their correlation with the performance metric.

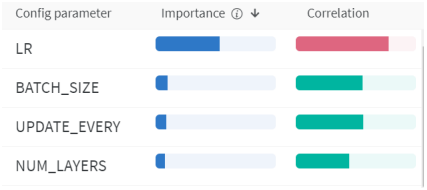


Figure 18: Relative importance of hyperparameters and its correlation with the performance metric

Bayes exploration strategy has been used to identify the best set of hyperparameters. The parameters are chosen in such a way that the distance of new parameters from the current hyperparameters is proportional to the error int current metric chosen. Mountain car has the slowest learning process and it was not able to improve its performance for any wide range of hyperparameter settings in Actor critic. From the graphs that have been, plotted for actor critic methods, it is evident that the one step actor critic methods have huge variance in all the three environments namely cartpole, acrobot and mountain car. It is evident from the graphs that have been plotted above. The huge variation in one step actor critic is because of the TD(0) update and less variance in the full return actor critic methods is due to monte-carlo return update. As n-step actor critic is in between the TD(0) update and the monte carlo update, it has variance less than one step, but greater than Full return actor critic. For, Actor critic methods, the value of learning rate is a very important hyparameter, followed by the neural network architecture. The architecture of neural network determines how well we generalized the action and state space values.
Thus DQN and Actor critic methods have been implemented on the gym environments CartPole, Acrobot and MountainCar.