CS6700: Programming Assignment 2
DQN & Actor Critic

Aaditya Kumar (EE21D411) and Arunesh J B (CS20B009)

March 28, 2023

1 Introduction

In this assignment we will be implementing the **DQN** and **Actor-Critic** methods. We will run the above two algorithms on three environments from gym **Acrobot-v1** , **CartPole-v1** and **MountainCar-v0**. We will be tuning hyperparameters such as QNetwork Structure, gamma, update frequency, learning rate, batch size, replay buffer size etc for each of these three environments using both the above algorithms. To account for stochasticity, we will be using the average of 10 runs for each hyperparameter configuration.

2 DQN

1. Code to Q-Network which takes in an arbitrary number of hidden layers

```
class QNetwork2(nn.Module):

    def __init__(self,state_size,action_size,seed,no_hidden_layers=2,
   hidden_layer_dims=[128,64]):
        super(QNetwork2, self).__init__()
        self.fc = nn.ModuleList()
        prev = state_size
        for i in range(no_hidden_layers):
          self.fc.append(nn.Linear(prev,hidden_layer_dims[i]))
          prev = hidden_layer_dims[i]
        self.fc.append(nn.Linear(prev,action_size))
        self.seed =torch.manual_seed(seed)
        self.no_hidden_layers = no_hidden_layers

    def forward(self, state):
        x = state
        for i in range(self.no_hidden_layers):
          x = F.relu(self.fc[i](x))
        return F.relu(self.fc[self.no_hidden_layers](x))
```

Listing 1: Funtion QNetwork2

2. action selection using softmax

```
def act_sm(self, state, tau=0.2):

  state = torch.from_numpy(state).float().unsqueeze(0).to(device)
  self.qnetwork_local.eval()
  with torch.no_grad():
      action_values = self.qnetwork_local(state)
  self.qnetwork_local.train()

  ''' Softmax action selection '''
  max_Q = np.max(action_values.cpu().data.numpy()[0])
  return np.random.choice(np.arange(self.action_size), p = np.exp((
   action_values.cpu().data.numpy()[0] - max_Q)/tau)/(np.sum(np.exp((
   action_values.cpu().data.numpy()[0] - max_Q)/tau))))
```

Listing 2: Function to select actions based on the softmax policy

3. dqn using softmax

```
def dqn_sm(n_episodes=1000, max_t=200, tau_start=1.0, tau_end=1e-8,
    tau_decay=0.995):

    episode_ = [] #
    all_scores = [] #
    all_episodes = [] #
    scores = []
    ''' list containing scores from each episode '''

    scores_window_printing = deque(maxlen=10)
    ''' For printing in the graph '''

    scores_window= deque(maxlen=100)
    ''' last 100 scores for checking if the avg is more than 195 '''

    eps = tau_start
    ''' initialize epsilon '''

    for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        for t in range(max_t):
            action = agent.act_sm(state, eps)
            next_state, reward, done, _ = env.step(action)
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break
        all_scores.append(score) #
        all_episodes.append(i_episode) #
        scores_window.append(score)
        scores_window_printing.append(score)
        ''' save most recent score '''
```

```
34
35          eps = max(tau_end, tau_decay*eps)
36          ''' decrease epsilon '''
37
38          print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode,
     np.mean(scores_window)), end="")
39          if i_episode % 10 == 0:
40              scores.append(np.mean(scores_window_printing))
41              episode_.append(i_episode) #
42          if i_episode % 100 == 0:
43              print('\rEpisode {}\tAverage Score: {:.2f}'.format(
    i_episode, np.mean(scores_window)))
44          if np.mean(scores_window)>=195.0:
45              print('\nEnvironment solved in {:d} episodes!\tAverage
    Score: {:.2f}'.format(i_episode-100, np.mean(scores_window)))
46              break
47      return [np.array(scores),i_episode-100,episode_,all_scores,
    all_episodes] #
```

Listing 3: dqn using softmax

4. To create a QNetwork

```
1 self.qnetwork = QNetwork2(state_size, action_size, seed,
    no_hidden_layers = NUM_HIDDEN_LAYERS, hidden_layer_dims =
    HIDDEN_LAYER_DIMS).to(device)
```

Listing 4: Create an object to class QNetwork

2.1 CartPole-v1

1. Trial Runs on different hyperparameters

	buffer size	batch size	gamma	learning rate	update every	hidden layer dims	no of hidden layers	Avg episodes to solve env
1	1e5	64	0.99	5e-4	20	[128,64]	2	613.7
2	1e5	128	0.99	5e-4	20	[128,64]	2	285.3
3	1e6	128	0.99	5e-4	20	[128,64]	2	521.7
4	1e7	128	0.99	5e-4	25	[128,64]	2	468.4
5	1e6	128	0.99	1e-4	25	[128,64]	2	160.2
6	1e7	128	0.99	5e-4	25	[128,128]	2	352.2
7	1e7	128	0.99	1e-4	25	[128,64]	2	147.5
8	1e5	64	0.99	5e-4	30	[128,64]	2	674.0
9	1e5	64	0.99	1e-4	20	[128,64]	2	371.9
10	1e6	128	0.99	1e-4	25	[128,64]	2	143.2
11	1e6	128	0.999	1e-4	25	[128,64]	2	176.5
12	1e6	128	0.99	1e-4	25	[128,64,64]	3	300.2
13	1e7	128	0.99	1e-4	25	[128,64,64]	3	135.0
14	1e7	128	0.99	5e-4	25	[128,64,64]	3	3528.4

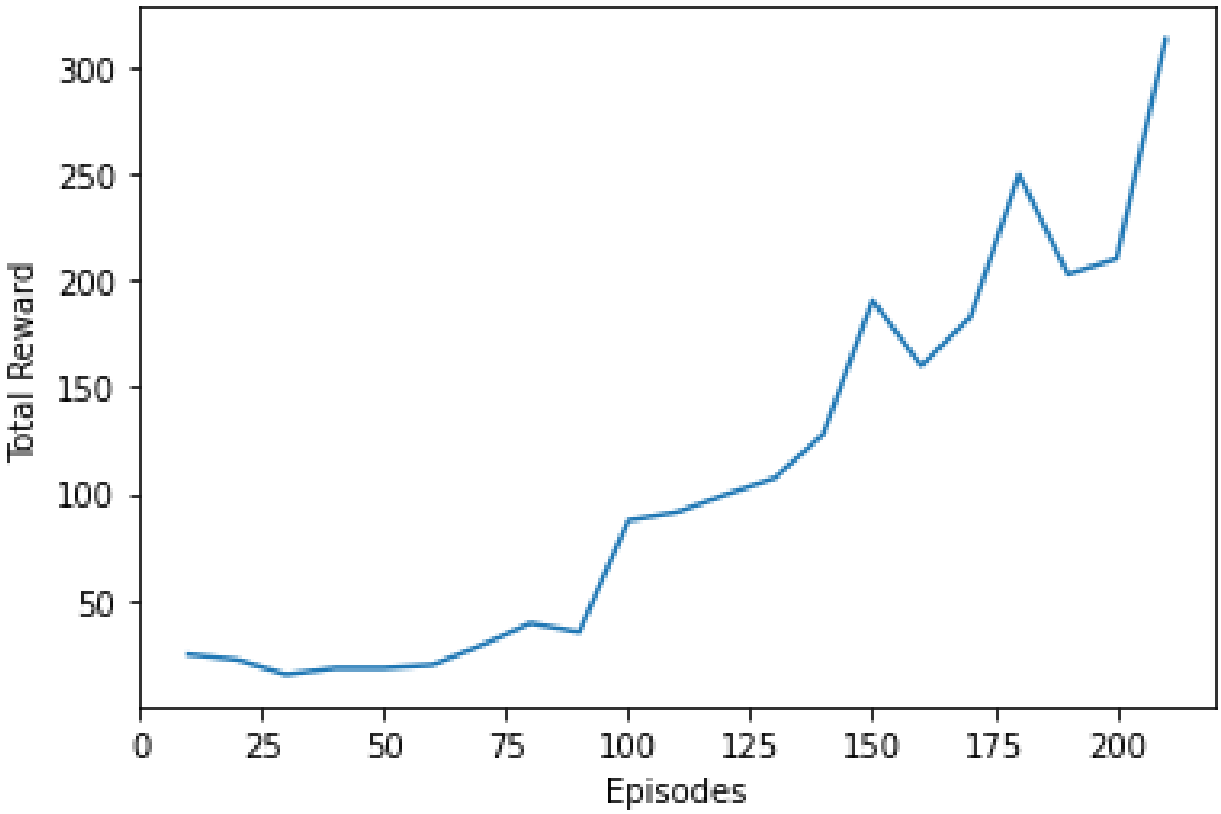
2. Hyperparameter Tuning

- **Trial 1** : Started of with the set of hyperparameters given in Tutorial-5. Got average episodes to solve to be 613.7.
- **Trial 2** : Now started the tuning procedure by increasing the batch size to 128 from 64. This gave a better result compared to the previous configuration. The reason might be due to better better training because a larger batch size might give us a **better representation of data** (also reducing the impact of noise if present).
- **Trial 3** : Now increased the buffer size, expected better performance because increasing the replay buffer size is basically increasing the size of the set of experiences which might lead to better training but this didn't lead to better performance instead the performance decreased. The reason might be due to **over-fitting** on the experiences stored in the replay buffer.
- **Trial 4** : I decreased the update frequency to have a **better generalization** so that we can avoid over-fitting to some extent since updates are now slower. This lead to slightly better performance than the previous case but not enough.
- **Trial 5** : Now experimented by decreasing the learning rate. Decreasing the learning rate can make the updates of the parameters happen more gradually this might lead to better convergence because if learning rate is high the loss function might **oscillate** near the minima. This added a lot to increase performance.
- **Trial 6**: This time tried to experiment on the structure of the neural network. Didnt change the number of hidden layers, kept the number of hidden layers 2 but changed the number of neurons. Increased the number of neurons in the second from 64 to 128, we increase thew number of neurons in a neural net will give a better approximation of the function we want to fit, hence it might lead to a **more robust Q-function**. This gave a value of 352.2 .
- **Trial 7**: Tried increasing the size of replay buffer to 1e7. This is better because **increasing the replay buffer size** is basically increasing the size of the set of experiences which might lead to better training. It can also help overcome the effects of noisy environments. This gave the best value uptill now(147.5).
- **Trial 8**: Now tried to increase update every to 30. This turned to be a bad idea since decreased the avg episodes to solve. The reason might be due to slower learning process.
- **Trial 9**: Now since in case of trial - 5 decreasing learning rate worked pretty well, I tried to revert to the initial configuration and decrease the learning rate alone. This turned out to be better compared to the initial configuration but this is not the best.
- **Trial 10**: Tried another buffer size of 1e6 worked well for trial 5. I tried it with learning rate 1e-4 which we saw clearly worked well in case of trial 8 and update every parameter to 25. This turned out to be a pretty good model.

- **Trial 11**: All this time experimented on every parameter except gamma. So lets try something with gamma this time. Lets increase gamma i.e increase it's focus towards more long term rewards. This weaken our model. Maybe due to overestimation of Q-values.
- **Trial 12**: Now we tried increasing the depth of the neural net with the same configuration as trial 10. Didn't optimize, might be due to over-fitting/overestimation.
- **Trial 13**: We tried increasing the depth of the neural net with the same configuration as trial 7 (which turned out well). This turned out to be the optimization so far and gave the avg episode to solve the env to be 135.0 . The reason might be due to **better estimation of the Q-Function** and also **larger replay buffer** leading to learning from more experiences.
- **Trial 14**: Now tried increasing learning rate for the previous model, this turned to be way worse. Might be due to bad convergence.

3. Reward curve for best model (Configuration 13 in the table)

   Episode 100 Average Score: 30.77
   Episode 200 Average Score: 162.19
   Episode 214 Average Score: 196.
   Environment solved in **114 episodes**! Average Score: **196.77**
   0:01:56.775629



2.2 Acrobot-v1

1. Trial Runs on different hyperparameters

	buffer size	batch size	gamma	learning rate	update every	hidden layer dims	no of hidden layers	Avg episodes to solve env
1	1e6	256	0.99	5e-4	20	[512,256,128]	3	-200
2	1e6	256	0.99	5e-4	20	[128,64]	2	-200
3	1e6	256	0.99	5e-4	20	[1024,512]	2	-200
4	1e6	256	0.99	1e-4	20	[512,256,128]	3	-200
5	1e7	128	0.99	5e-4	25	[512,256,128]	3	-200
6	1e7	128	0.99	1e-4	25	[512,256,128]	3	-200
7	1e5	256	0.99	5e-4	25	[512,256,128]	3	-200
8	1e5	256	0.99	5e-4	20	[256,128,64,32]	4	-200

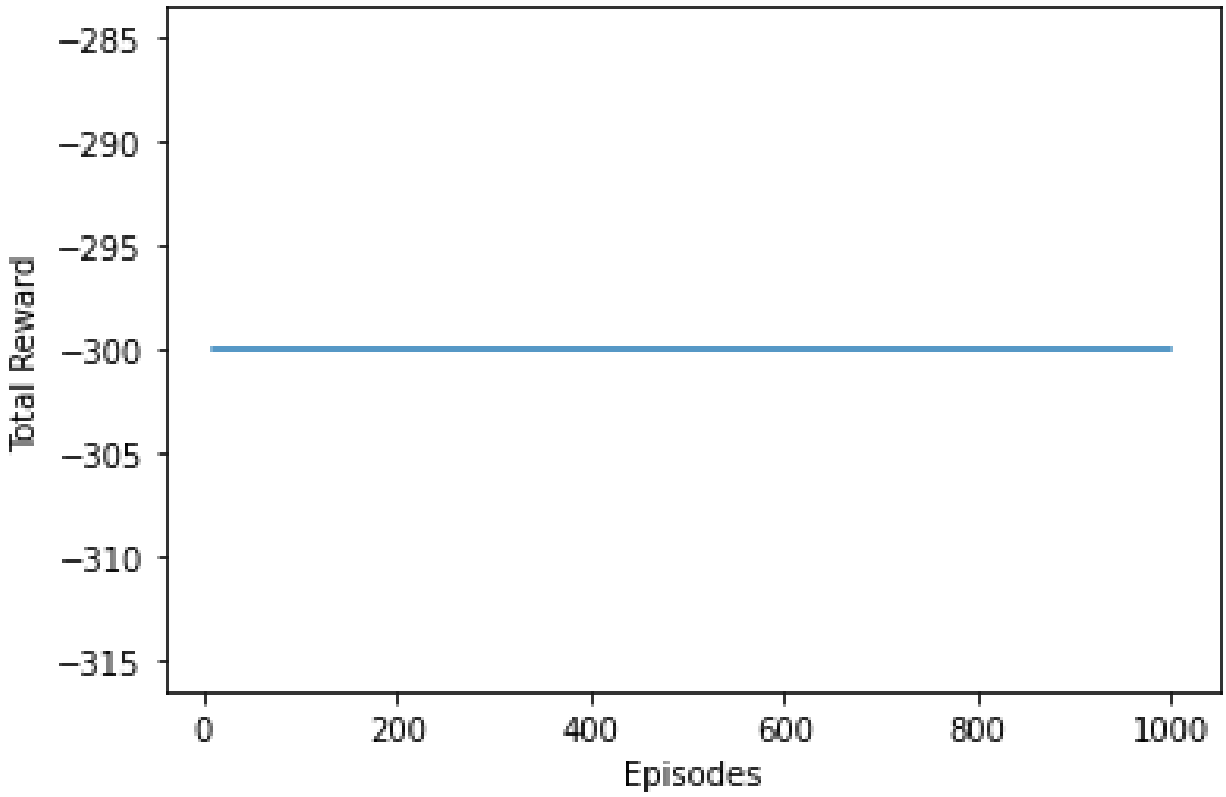
2. Hyperparameter Tuning

   - Started of with a deep neural network with 512,256,128 neurons. Environment didnt solve.
   - To avoid overfitting made the network architecture less complex by reducing the number of hidden layers and number of neurons.
   - Did you few more changes like decreasing the learning rate to not converge to a local minima in the loss function and tried playing around with the network architecture.
   - also tried a very deep network in the last trial. No change in the avg reward.

3. Reasons for the above behaviour:

   - One reason could be that the epsilon is decaying fast hence after a certain point the agent is not able to explore or learn.
   - Also as we increase the depth or number of neurons in the neural network the computation becomes much slower, hence it's computationally expensive to check more deeper neural nets and also since we are taking 10 runs for each trial and take average the process becomes computationally heavy.
   - Also in this case I didn't take into account other possible hyper parameters like dropout, optimizers and activation functions. These might give some better results if tried.
   - We could have also tried replay buffer with priorities.
   - Also due to the time constraint didnt explore more hyperparameter configurations

4. Reward Curve :



2.3 MountainCar-v0

1. Trial Runs on different hyperparameters

	buffer size	batch size	gamma	learning rate	update every	hidden layer dims	no of hidden layers	Avg episodes to solve env
1	1e6	256	0.99	5e-4	20	[512,256,128]	3	-200
2	1e6	256	0.99	5e-4	20	[1024,512]	2	-200
3	1e6	256	0.99	5e-4	20	[128,64]	2	-200
4	1e6	256	0.99	1e-4	20	[512,256,128]	3	-200
5	1e7	128	0.99	5e-4	25	[512,256,128]	3	-200
6	1e7	128	0.99	1e-4	25	[512,256,128]	3	-200
7	1e5	256	0.99	1e-4	25	[512,256,128]	3	-200
8	1e5	64	0.99	1e-4	20	[256,128,64,32]	4	-200

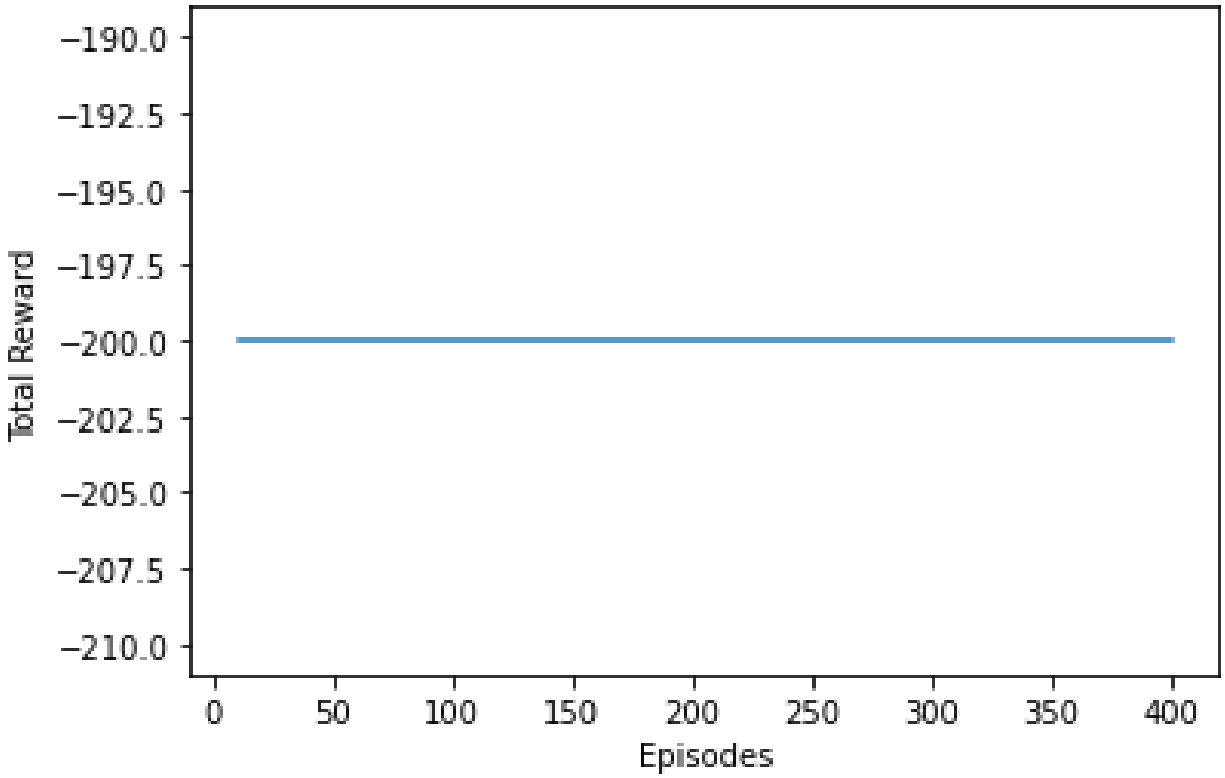
2. Hyperparameter Tuning
   Before starting the trials tried to find a set of hyperparameters for which environment is solved, but wasn't able to find a configuration for which average reward changes.

- Trial 1: Tried to use a deep network so that it captures a broader Q function. Got a average reward of -200 and environments was not solved
- Trial 2: Tried to decrease the depth increase the number of neurons, but still got the same result.
- Trial 3: Now tried the initial configuration which was given in tut-5.
- Trial 4: Thought decreasing the learning rate might help the loss function to converge to a minima.
- Trial 5: Increased the replay buffer so that the agent can learn from its history but didn't make any difference.
- Trial 6: With the configurations of the above trial tried to decrease the learning rate.
- Trial 7: Decreased the replay buffer capacity just in case its over-fitting form ites experiences. But this didn't work either.
- Trial 8: Now took a very deep network, and decreased the batch size to reduce the computational overhead but this didn't make any difference either.

3. Reasons for the above behaviour:

   - One reason could be that the epsilon is decaying fast hence after a certain point the agent is not able to explore or learn.
   - Also as we increase the depth or number of neurons in the neural network the computation becomes much slower, hence it's computationally expensive to check more deeper neural nets and also since we are taking 10 runs for each trial and take average the process becomes computationally heavy.
   - Also in this case I didn't take into account other possible hyper parameters like dropout, optimizers and activation functions. These might give some better results if tried.
   - We could have also tried replay buffer with priorities.

4. Reward Curve



3 Actor Critic

3.1 Experiments

- **CartPole:**
  'CartPole-v1' is considered to be solved if the average reward is >= 195 for over 100 consecutive trials.

  - **One Step Return:**
    The hyperparameters selected are as follows:

    1. Actor-Critic Network: Hidden layers: [512,256]
    2. learning rate: 0.001
    3. Number of episodes: 500

    Number of steps to reach the goal: 645
    Mean Rewards for each episode (for 10 runs): 250.6658

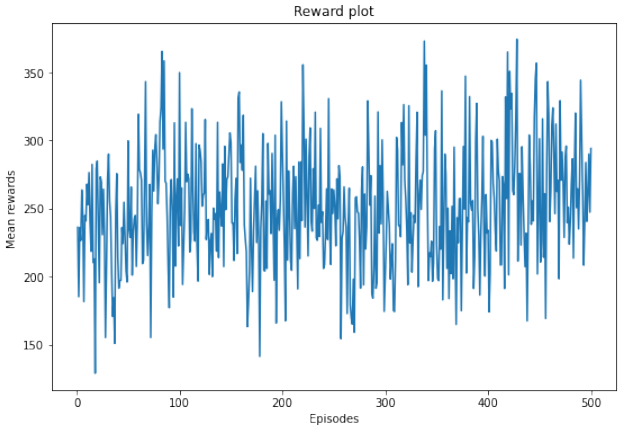


Figure 1: Mean Rewards for each episode (for 10 runs)

Variance of Rewards for each episode (for 10 runs): 46.4841

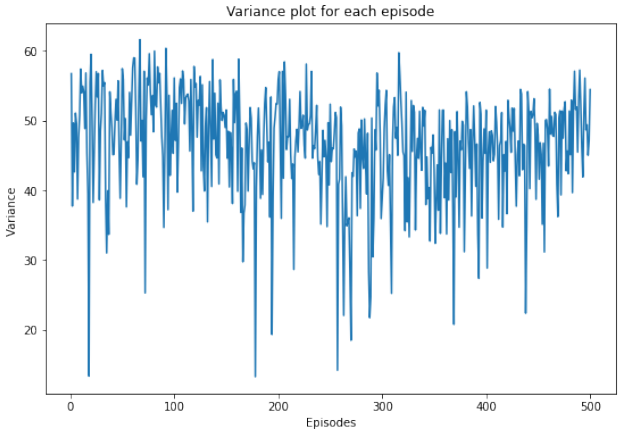


Figure 2: Variance of Rewards for each episode (for 10 runs)

– **Full Return:**
The hyperparameters selected are as follows:

1. Actor-Critic Network: Hidden Layer [128]
2. learning rate: 0.01
3. Number of episodes: 500

Number of steps to reach the goal: 131
Mean Rewards for each episode (for 10 runs): 222.7534

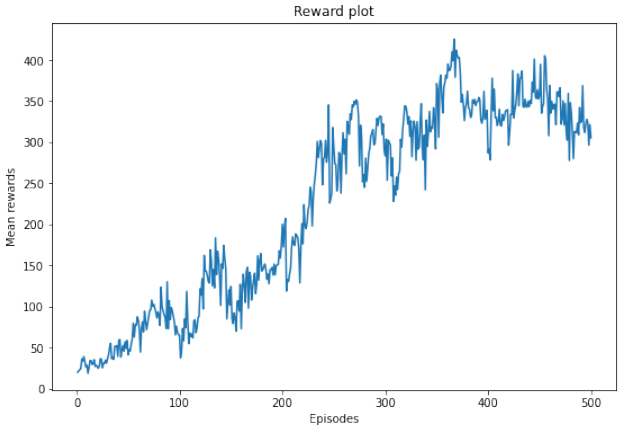


Figure 3: Mean Rewards for each episode (for 10 runs)

Variance of Rewards for each episode (for 10 runs): 131.4025

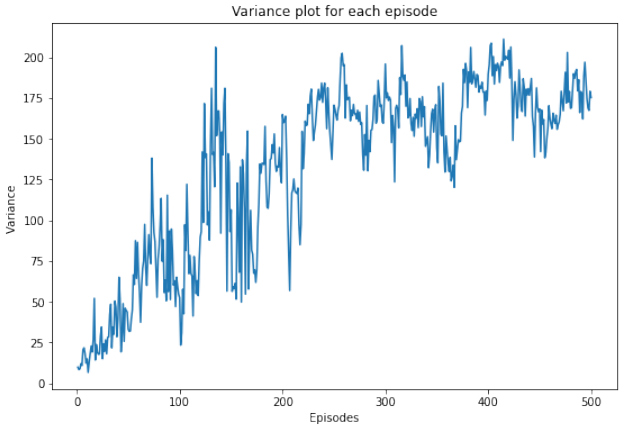


Figure 4: Variance of Rewards for each episode (for 10 runs)

– **n-Step Returns:**
The hyperparameters selected are as follows:

1. Actor-Critic Network: [1024]
2. learning rate: 0.001
3. Number of episodes: 500

Number of steps to reach the goal: 175
Mean Rewards for each episode (for 10 runs): 125.6114

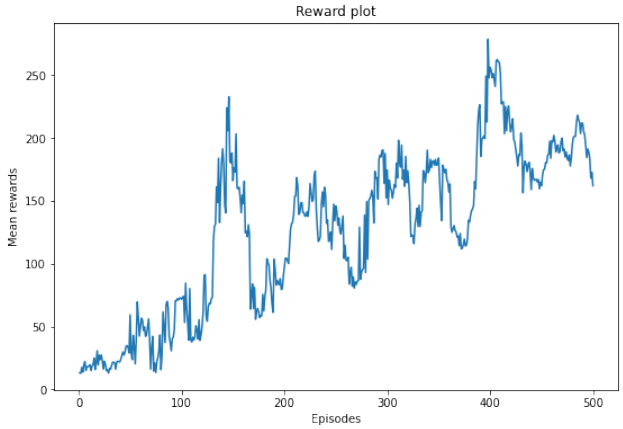


Figure 5: Mean Rewards for each episode (for 10 runs)

Variance of Rewards for each episode (for 10 runs): 101.024

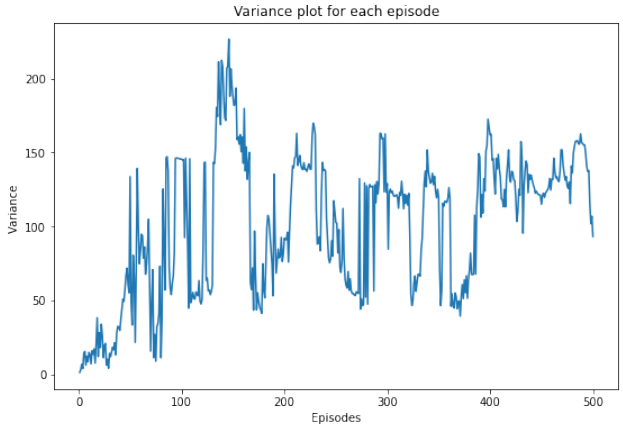


Figure 6: Variance of Rewards for each episode (for 10 runs)

- **Acrobot:**
  'Acrobot-v1' is considered to be solved if the average reward is >= -100 for over 100 consecutive trials.

  - **One Step Return:**
    The hyperparameters selected are as follows:

    1. Actor-Critic Network: Hidden Layers [512,256]
    2. learning rate: 0.0001
    3. Number of episodes: 300

    Number of steps to reach the goal: 250
    Mean Rewards for each episode (for 10 runs): -98.47

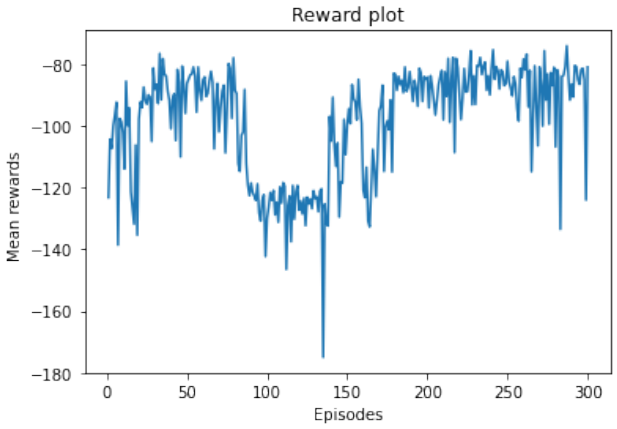


Figure 7: Mean Rewards for each episode (for 10 runs)

Variance of Rewards for each episode (for 10 runs): 15.60

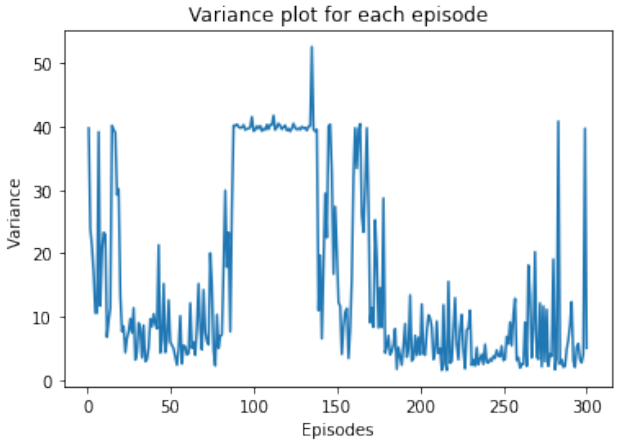


Figure 8: Variance of Rewards for each episode (for 10 runs)

– **Full Return:**
The hyperparameters selected are as follows:

1. Actor-Critic Network: Hidden Layer: [512]
2. learning rate: 0.001
3. Number of episodes: 750

Number of steps to reach the goal: 750
Mean Rewards for each episode (for 10 runs): -169.9086

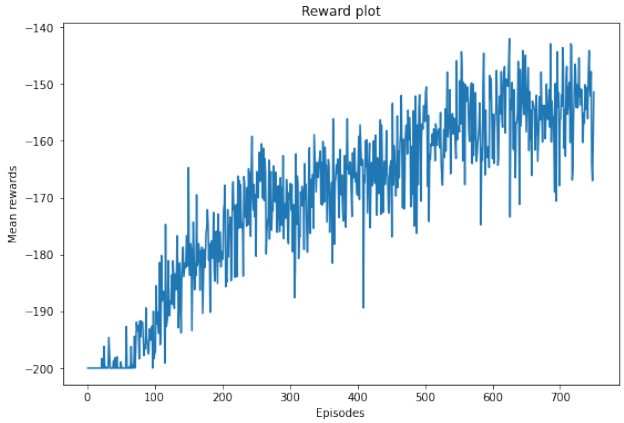


Figure 9: Mean Rewards for each episode (for 10 runs)

Variance of Rewards for each episode (for 10 runs): 34.6364

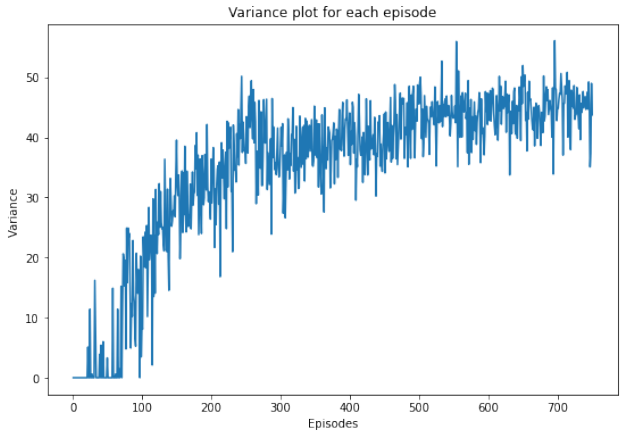


Figure 10: Variance of Rewards for each episode (for 10 runs)

– **n-Step Returns:**
The hyperparameters selected are as follows:

1. Actor-Critic Network: Hidden Layer: [256]
2. learning rate: 0.001
3. Number of episodes: 600

Number of steps to reach the goal: 363
Mean Rewards for each episode (for 10 runs): -174.375

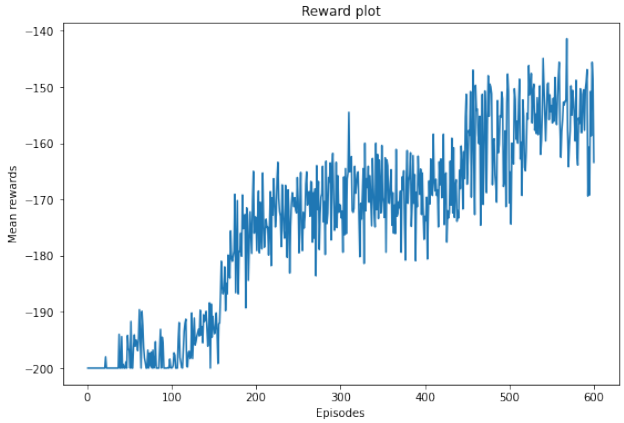


Figure 11: Mean Rewards for each episode (for 10 runs)

Variance of Rewards for each episode (for 10 runs): 31.9349

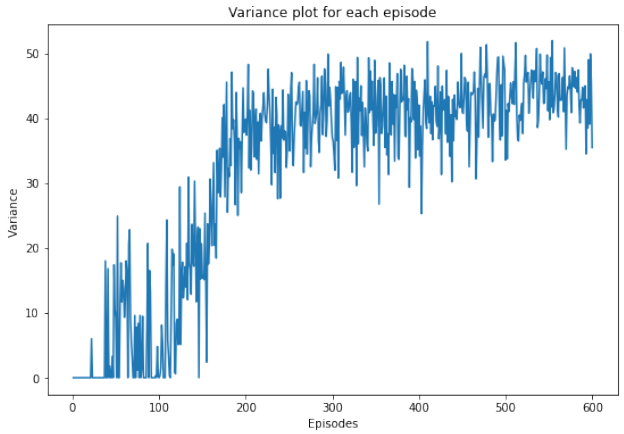


Figure 12: Variance of Rewards for each episode (for 10 runs)

- **MountainCar:**
  'MountainCar-v0' is considered to be solved if the average reward is >= -200 for over 100 consecutive trials.

  - **One Step Return:**
    The hyperparameters selected are as follows:

    1. Actor-Critic Network: Hidden Layers [1024,512]
    2. learning rate: 0.1
    3. Number of episodes: 250

    Number of steps to reach the goal: (Did not converge for value more than -200).
    A reward of -200 was received for all 1000 episodes.
    Mean Rewards for each episode (for 10 runs): -200

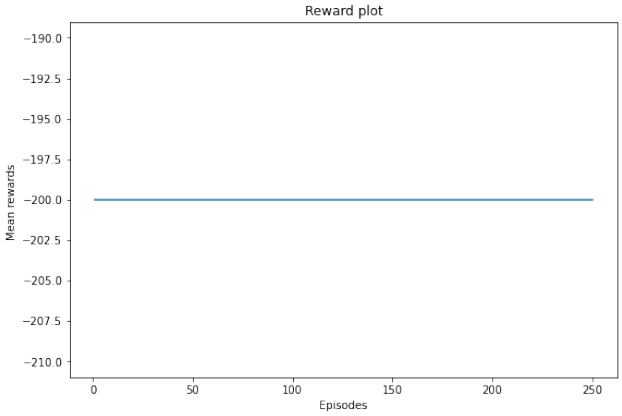


Figure 13: Mean of Rewards for each episode (for 10 runs)

Variance of Rewards for each episode (for 10 runs): 0

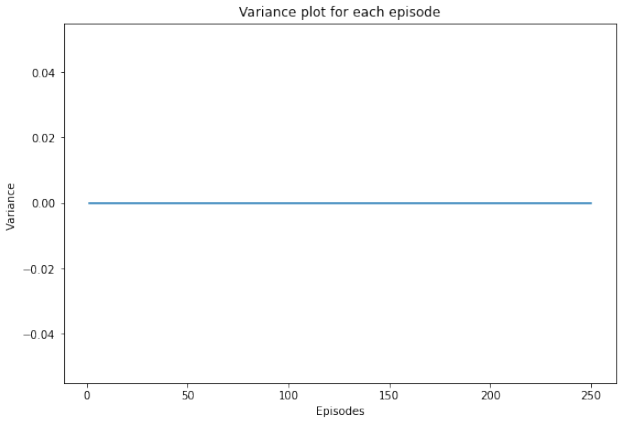


Figure 14: Variance of Rewards for each episode (for 10 runs)

– **Full Return:**
The hyperparameters selected are as follows:

1. Actor-Critic Network: Hidden Layer: [128]
2. learning rate: 0.01
3. Number of episodes: 200

Number of steps to reach the goal: (Did not converge for value more than -200).
A reward of -200 was received for all 1000 episodes.
Mean Rewards for each episode (for 10 runs): -199.085

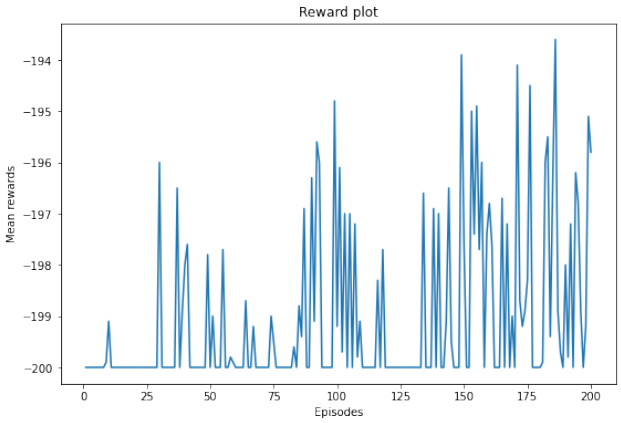


Figure 15: Mean of Rewards for each episode (for 10 runs)

Variance of Rewards for each episode (for 10 runs): 2.7130

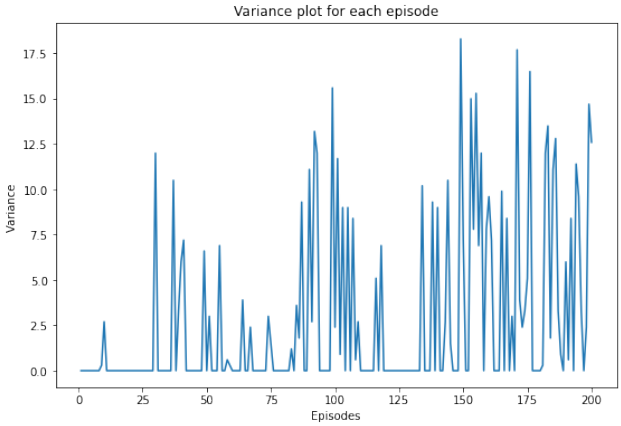


Figure 16: Variance of Rewards for each episode (for 10 runs)

– **n-Step Returns:**
The hyperparameters selected are as follows:

1. Actor-Critic Network: Hidden Layer: [256]
2. learning rate: 0.001
3. Number of episodes: 200

Number of steps to reach the goal: (Did not converge for value more than -200).
A reward of -200 was received for all 1000 episodes.
Mean Rewards for each episode (for 10 runs): -200

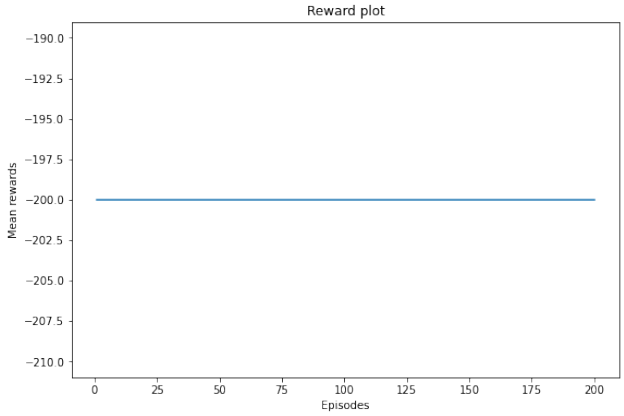


Figure 17: Mean of Rewards for each episode (for 10 runs)

Variance of Rewards for each episode (for 10 runs): 0

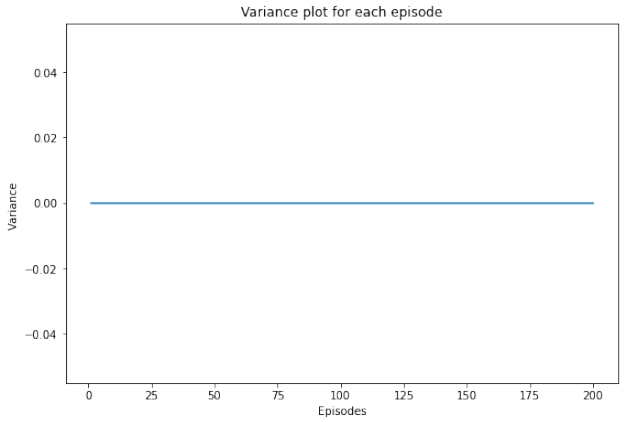


Figure 18: Variance of Rewards for each episode (for 10 runs)

3.2 Code Snippets

- To Compute the Expected Returns for Full Return Actor-Critic:

```
def get_expected_return(
    rewards: tf.Tensor,
    gamma: float,
    standardize: bool = True) -> tf.Tensor:
  """
  Compute expected returns per timestep.

    Args:
    rewards: tensor of shape (n,), containing rewards at each
   timestep.
    gamma: discount factor.
    standardize: whether to standardize the returns or not.

  Returns:
```

```
    tensor of shape (n,), containing expected returns at each
    timestep.
  """

  n = tf.shape(rewards)[0]
  returns = tf.TensorArray(dtype=tf.float32, size=n)

  # Start from the end of `rewards` and accumulate reward sums
  # into the `returns` array
  rewards = tf.cast(rewards[::-1], dtype=tf.float32)
  discounted_sum = tf.constant(0.0)
  discounted_sum_shape = discounted_sum.shape
  for i in tf.range(n):
    reward = rewards[i]
    discounted_sum = reward + gamma * discounted_sum
    discounted_sum.set_shape(discounted_sum_shape)
    returns = returns.write(i, discounted_sum)
  returns = returns.stack()[::-1]

  if standardize:
    returns = ((returns - tf.math.reduce_mean(returns)) /
               (tf.math.reduce_std(returns) + eps))

  return returns

```

Listing 5: Funtion Expected Returns for Full Return AC

- To Compute the Expected Returns for n-Step Returns Actor-Critic:

```
def get_expected_return(
    rewards: tf.Tensor,
    values: tf.Tensor,
    step: int,
    gamma: float,
    standardize: bool = True) -> tf.Tensor:
  """
  Compute expected returns per timestep.

  Args:
    rewards: tensor of shape (n,), containing rewards at each
    timestep.
    values: tensor of shape (n,), containing value estimates at each
    timestep.
    step: number of steps to look ahead for n-step returns.
    gamma: discount factor.
    standardize: whether to standardize the returns or not.

  Returns:
    tensor of shape (n,), containing expected returns at each
    timestep.
  """

  n = tf.shape(rewards)[0]
```

```
  returns = tf.TensorArray(dtype=tf.float32, size=n)

  # Start from the end of `rewards` and accumulate reward sums
  # into the `returns` array

  rewards = tf.cast(rewards[::-1], dtype=tf.float32)
  discounted_sum = tf.constant(0.0)
  discounted_sum_shape = discounted_sum.shape

  for i in tf.range(n):
      if i+step >= n:
        n_step_return = tf.constant(0.0)
        n_step_return_shape = n_step_return.shape
        for j in tf.range(i,n):
            n_step_return = tf.math.pow(gamma,float(j-i))*rewards[j]
   + n_step_return
            n_step_return.set_shape(n_step_return_shape)
        discounted_sum = n_step_return
      else:
        n_step_return = tf.constant(0.0)
        n_step_return_shape = n_step_return.shape
        for j in tf.range(i,i+step):
            n_step_return = tf.math.pow(gamma,float(j-i))*rewards[j]
   + n_step_return
            n_step_return.set_shape(n_step_return_shape)
        discounted_sum = n_step_return + tf.math.pow(gamma,float(step
   ))*values[i+step]


      discounted_sum.set_shape(discounted_sum_shape)
      returns = returns.write(i, discounted_sum)
  returns = returns.stack()

  if standardize:
    returns = ((returns - tf.math.reduce_mean(returns)) /
               (tf.math.reduce_std(returns) + eps))

  return returns

```

Listing 6: Funtion Expected Returns for n-Step Returns AC

- Expression for delta (TD error) for one-Step AC:

```
            # equation for delta (TD error):

            delta = delta = reward + self.gamma * V_s_next * (1 -
   done) - V_s
            loss_a = self.actor_loss(action, pi, delta)
            loss_c =self.critic_loss(delta)
            loss_total = loss_a + loss_c

```

Listing 7: Equation for delta (TD error) for one-Step AC

- loss function for full-step returns AC

```
huber_loss = tf.keras.losses.Huber(reduction=tf.keras.losses.
    Reduction.SUM)

def compute_loss(
    action_probs: tf.Tensor,
    values: tf.Tensor,
    returns: tf.Tensor) -> tf.Tensor:
  """Computes the combined Actor-Critic loss."""

  #delta or TD error
  advantage = returns - values

  action_log_probs = tf.math.log(action_probs)
  actor_loss = -tf.math.reduce_sum(action_log_probs * advantage)

  critic_loss = huber_loss(values, returns)

  return actor_loss + critic_loss

```

Listing 8: loss function for full-step returns AC

4 Conclusions

DQN:

- We observed that in case of Cart-Pole updating less frequently can optimize performance sometimes. Batch size of 128 works best.
- Replay buffer of size 1 million or 10 million works well
- In case of Acrobat wasn't able to find a good hyperparameter configuration and could'nt be solved. But selecting actions with soft-max policy would have worked. Ma
- The MountainCar-v0 also couldnt be solved with the hyperparameters I used to explore.

AC:

- Full returns variation of the Actor-Critic Method had the highest variance of episode reward for each episode.
- The average episode reward for each episode was the highest for the one-step Actor-Critic for almost all the different environments.
- MountainCar-v0 environment couldn't be solved for a varied choice of hyperparameters. A reward of -200 was received for all 1000 episodes almost always with zero variance.

References

[1] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction* , The MIT Press (1 January 1998).

[2] *DQN*, Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529{533 (2015). https://doi.org/10.1038/nature14236.

[3] *Actor-Critic*, Tutorial: Playing CartPole with the Actor-Critic method, https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic.

[4] *Tutorial 5* CS6700 Tutorial-5 DQN and AC Code