Programming Assignment 2

DQN and Actor-Critic Methods

Gautham Govind A - EE19B022
Vishnu Vinod - CS19B048

CS6700
Reinforcement Learning

Department of Computer Science and Engineering
IIT Madras

Contents

1 CODE FOLDER

The entirety of the code written for this assignment as well as the output files containing all plots and notebooks are present in this Google Drive Link: `https://drive.google.com/drive/folders/1nAHkAv6-zYXTZmjqO0XhuOXNOOZ` `Wx?usp=sharing`

2 Deep Q-Learning

Deep Q-Learning is a variant of the popular Q-Learning algorithm used for solving an RL problem. The core idea is to make use of advancements in deep learning to enhance Q-Learning. The strategy focuses on using a neural network to parameterize the Q function and perform Q-Learning using this parameterized Q-function.

For implementing a DQN agent, the two main components are:

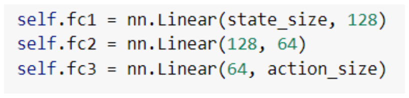
1. Q-Network
2. Replay Buffer

Q-Network

The structure of the q-learning network plays a huge role in determining the performance of the agent. For the purpose of this assignment we consider three architectures:

q1

This is the network architecture given in tutorial 4. The architecture consists of 2 hidden layers with sizes 128 and 64 as well as input and output layers with sizes equal to number of states and actions respectively. The activation function used is ReLu. The architecture is defined in the following code block:

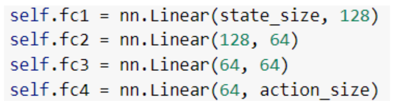


```
self.fc1 = nn.Linear(state_size, 128)
self.fc2 = nn.Linear(128, 64)
self.fc3 = nn.Linear(64, action_size)
```

Figure 1: q1 architecture

q2

The architecture consists of 3 hidden layers with sizes 128, 64 and 64 as well as input and output layers with sizes equal to number of states and actions respectively. q2 is a bigger network than q1. The activation function used is ReLu. The architecture is defined in the following code block:

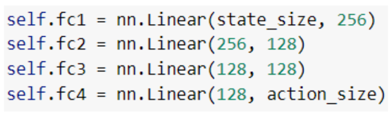


```
self.fc1 = nn.Linear(state_size, 128)
self.fc2 = nn.Linear(128, 64)
self.fc3 = nn.Linear(64, 64)
self.fc4 = nn.Linear(64, action_size)
```

Figure 2: q2 architecture

q3

The architecture consists of 3 hidden layers with sizes 256, 128 and 128 as well as input and output layers with sizes equal to number of states and actions respectively. q3 is the biggerst network among the three. The activation function used is ReLu. The architecture is defined in the following code block:

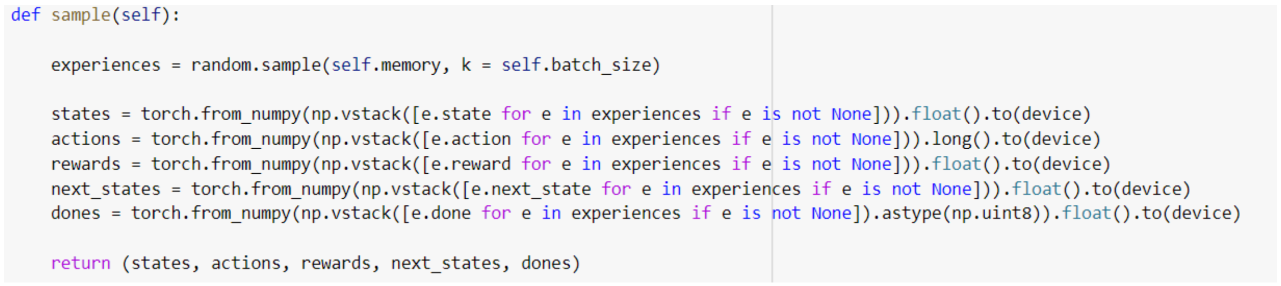


```
self.fc1 = nn.Linear(state_size, 256)
self.fc2 = nn.Linear(256, 128)
self.fc3 = nn.Linear(128, 128)
self.fc4 = nn.Linear(128, action_size)
```

Figure 3: q3 architecture

Replay buffer

The replay memory stores past experiences and samples them instead of using new experiences directly so as to break correlation. The code block corresponding to sampling the replay memory is given below:

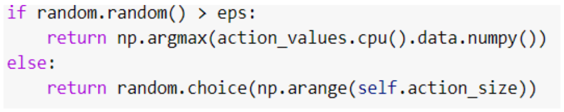


```
def sample(self):

    experiences = random.sample(self.memory, k = self.batch_size)

    states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not None])).float().to(device)
    actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None])).long().to(device)
    rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None])).float().to(device)
    next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is not None])).float().to(device)
    dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).float().to(device)

    return (states, actions, rewards, next_states, dones)
```

Figure 4: Sampling replay memory

Exploration strategy

We need to select an exploration strategy for selecting the action from the q value. We employ $\epsilon-$**greedy** strategy with **decaying** ϵ. We start with an ϵ value of 1.0 and go all the way till 0.01 with a decay of 0.995. The code block that implements this strategy is given below:



```
if random.random() > eps:
    return np.argmax(action_values.cpu().data.numpy())
else:
    return random.choice(np.arange(self.action_size))
```

Figure 5: Implementation of $\epsilon-$ greedy

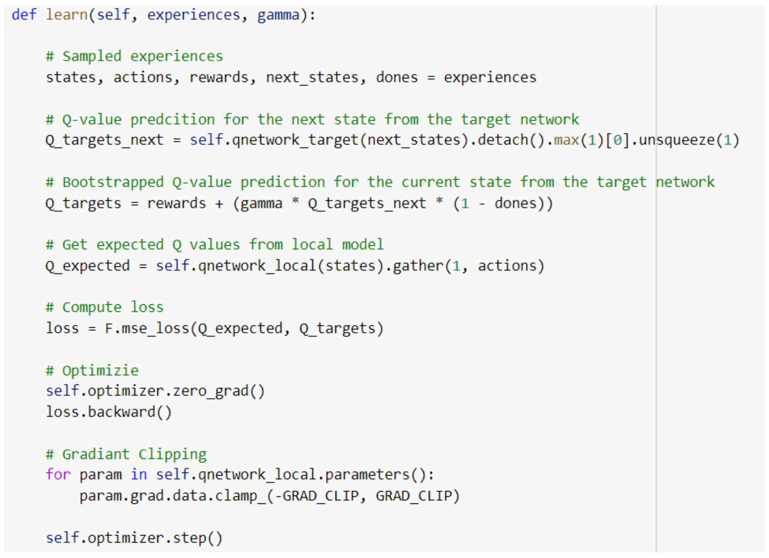
Learning the Q-function

The heart of the DQN agent lies in the learn() method which learns the q-function by changing the weights of the neural network appropriately. This is achieved using the code block given in Figure 6.

Plotting average rewards and steps

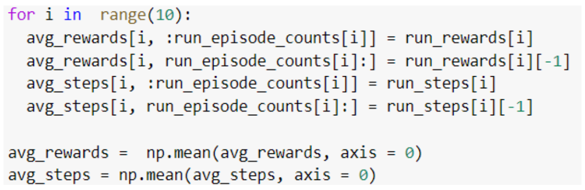
We have been asked to run each variant 10 times and plot the average rewards and steps as a function of the episode count. A key nuance here is that the number of episodes taken in each run might be different, in which case defining the average is not obvious.

We have taken the following approach for overcoming this issue: we take the run with the highest number of episodes taken as the baseline. For every other run, we duplicate the reward and number of steps in the last episode of that run till we make the number of episodes equal to the baseline. This ensures all runs have data pertaining to the same number of episodes. We then average over these values. This is implemented by the code block given in Figure 7.



```
def learn(self, experiences, gamma):

    # Sampled experiences
    states, actions, rewards, next_states, dones = experiences

    # Q-value predcition for the next state from the target network
    Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)

    # Bootstrapped Q-value prediction for the current state from the target network
    Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

    # Get expected Q values from local model
    Q_expected = self.qnetwork_local(states).gather(1, actions)

    # Compute loss
    loss = F.mse_loss(Q_expected, Q_targets)

    # Optimizie
    self.optimizer.zero_grad()
    loss.backward()

    # Gradiant Clipping
    for param in self.qnetwork_local.parameters():
        param.grad.data.clamp_(-GRAD_CLIP, GRAD_CLIP)

    self.optimizer.step()
```

Figure 6: Implementation of learn() module



```
for i in  range(10):
  avg_rewards[i, :run_episode_counts[i]] = run_rewards[i]
  avg_rewards[i, run_episode_counts[i]:] = run_rewards[i][-1]
  avg_steps[i, :run_episode_counts[i]] = run_steps[i]
  avg_steps[i, run_episode_counts[i]:] = run_steps[i][-1]

avg_rewards =  np.mean(avg_rewards, axis = 0)
avg_steps = np.mean(avg_steps, axis = 0)
```

Figure 7: Computing average rewards and steps

Hyperparameters

A brief description of the hyperparameters which are tuned is given below:

Hyperparameter	Description
Q-Network	Architecture used: q1, q2 or q3
Learning Rate	Learning rate of Q-network
Buffer size	Size of replay buffer
Batch size	Batch size for training
Update frequency	Number of steps after which target network is updated
γ	Discount factor
Gradient clip	Gradient is clipped between (-Gradient clip, Gradient clip)

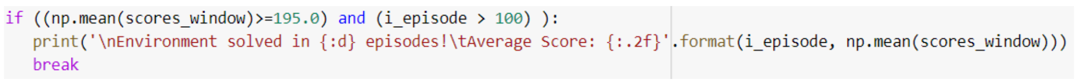
3 DQN: CartPole-v1

Environment Description

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.

Reward structure: Since the goal is to keep the pole upright for as long as possible, a reward of +1 for every step taken, including the termination step, is allotted.

Reward threshold: We take a reward threshold of +195 as given in tutorial 4. This means that the running average of the total reward over the last 100 episodes should be greater than +195. This is implemented using the following code block:



```
if ((np.mean(scores_window)>=195.0) and (i_episode > 100) ):
    print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
    break
```

Figure 8: Implementing reward threshold

Step threshold: 500 This means that an episode can run for as long as 500 steps.

We now give a description of the various configurations we tried, their performance and the reasoning behind the parameter choices made.

Variant 1 (Tutorial 4 configuration)

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	64
Update frequency	20
γ	0.99
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **940.3**

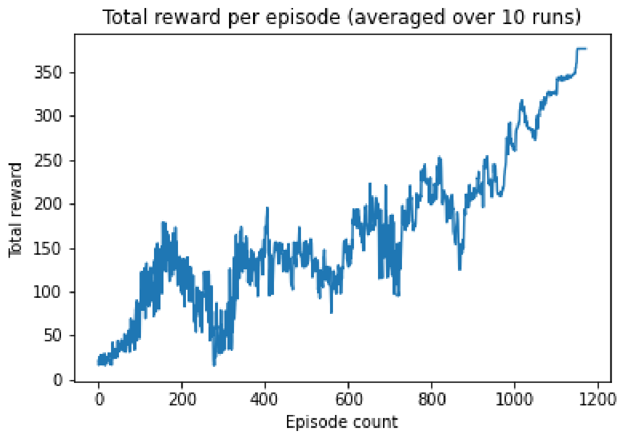


Figure 9: Averaged rewards

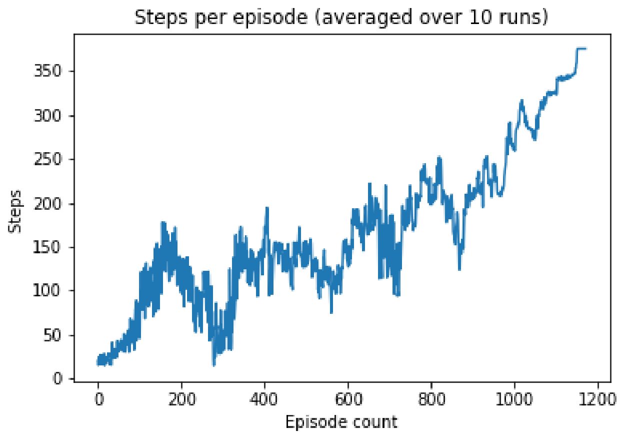


Figure 10: Averaged steps

Observations:

- The reward reaches 195. However, as we had mentioned in the description of computing average reward, since we duplicate the last reward for several cases, the plot runs well beyond 195. Essentially, since the plot has to account for the poorest performing run as well, the episode count represents the worst case scenario.
- The number of steps is initially equal to 0 and as the network learns, it increases till it reaches 195 when it solves the environment.
- The reward curve is essentially same as the step curve since each step gives a reward of +1.

Comments:

A good first step towards improving performance would be to verify whether the existing q-network architecture is sophisticated enough to represent the required q-function. To verify this, we use the bigger q-network q2 and check how the performance is affected. We keep the other parameters same.

Variant 2

Hyperparameter values:

Hyperparameter	Value
Q-Network	q2
Learning Rate	$5{\cdot}10^{-4}$
Buffer size	10^5
Batch size	64
Update frequency	20
γ	0.99
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **1561.7**

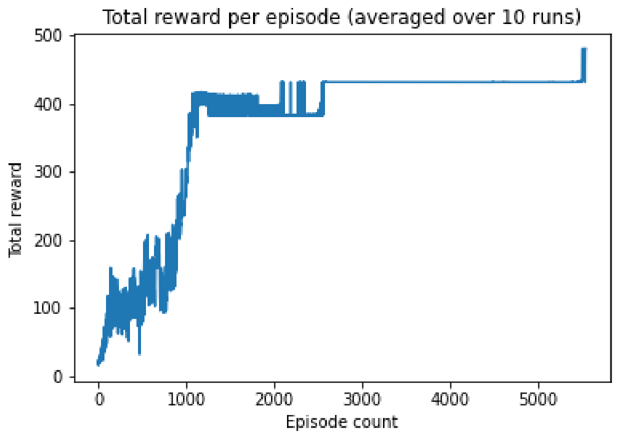


Figure 11: Averaged rewards

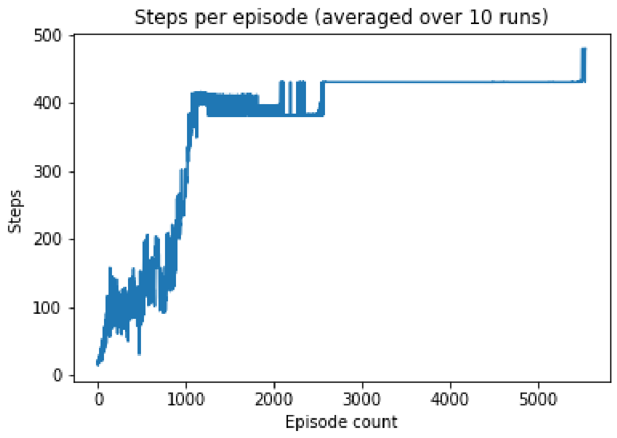


Figure 12: Averaged steps

Observations:

- The average number of episodes required to solve the environment is much higher in this case.
- Essentially, for variant 2, the learning happens slower than variant 1. In some runs, the agent takes more than 5000 episodes to solve the environment, which was not the case with variant 1.

Comments:

We see that the performance doesn't improve on using a bigger model. Rather, the performance drops. Hence, we revert back to q1 architecture.

We now investigate the impact of γ in the performance of the model. Since in this environment we keep getting a reward as long as we maintain an upright position, it makes sense to think of the problem as maximizing short term rewards as compared to the usual cases where you get a non-negative reward only at the very end. So we try decreasing the value of γ.

Variant 3

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	64
Update frequency	20
γ	0.9
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **755.7**

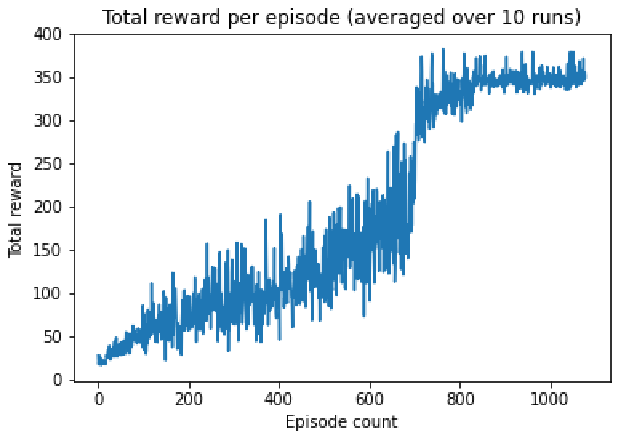


Figure 13: Averaged rewards

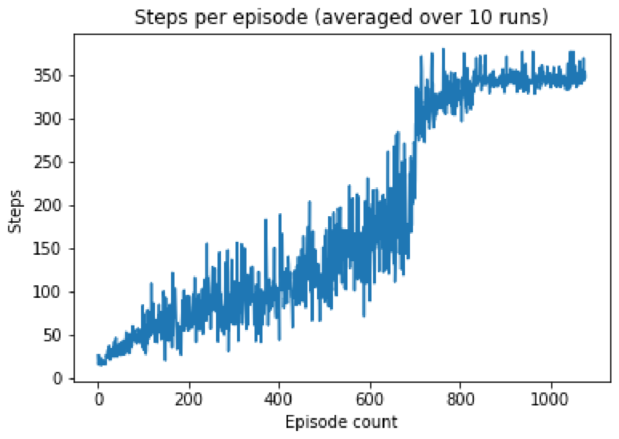


Figure 14: Averaged steps

Observations:

- The average number of episodes required to solve the environment is lower than variant 1.
- Essentially, for variant 3, the learning happens faster than variant 1.

Comments:

We see that the performance improves on decreasing γ to 0.9. Hence, we maintain this value for future variants. Note that we have now improved the performance of the model **once.**

We now investigate the impact of training batch size on the performance of the model. A higher batch size in general implies better estimates for the gradients and hence intuitively should lead to better performance. We examine this empirically.

Variant 4

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	128
Update frequency	20
γ	0.9
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **696.1**

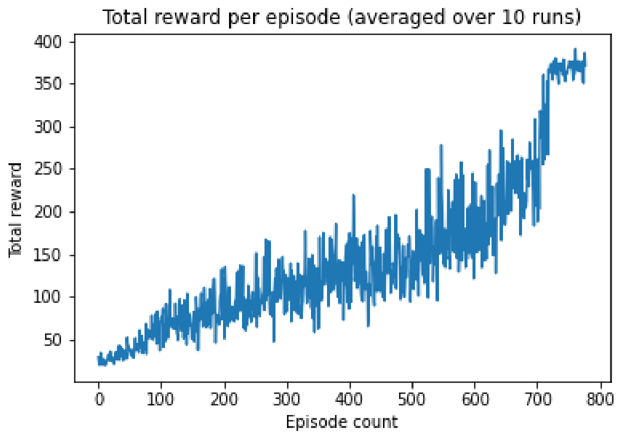


Figure 15: Averaged rewards

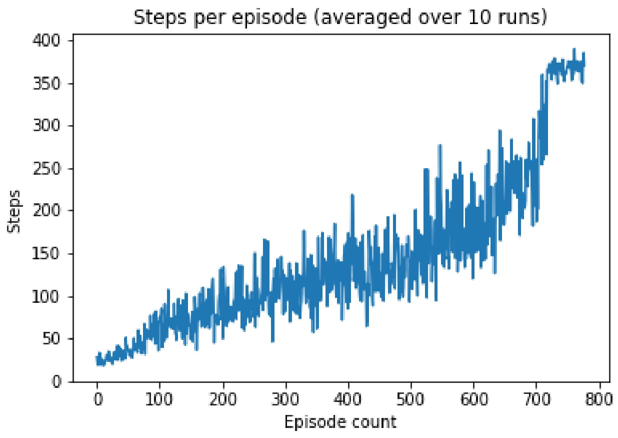


Figure 16: Averaged steps

Observations:

- The average number of episodes required to solve the environment is lower than variant 3.
- Essentially, for variant 4, the learning happens faster than variant 3.

Comments:

We see that the performance improves on increasing batch size to 128. Hence, we maintain this value for future variants. Note that we have now improved the performance of the model **twice.**

We now investigate the impact of target update frequency on the performance of the model. A lower update frequency could lead to better performance as it might help overcome the non-stationary target problem. However, this also means that the target network will be updated more slowly and could potentially slow down performance. We examine what happens empirically.

Variant 5

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	128
Update frequency	40
γ	0.9
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **755.6**

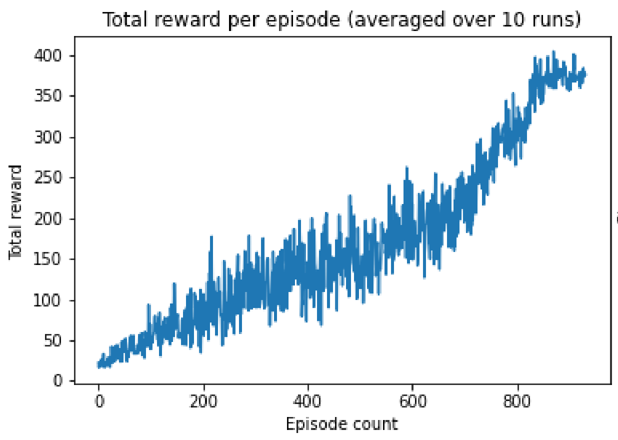


Figure 17: Averaged rewards

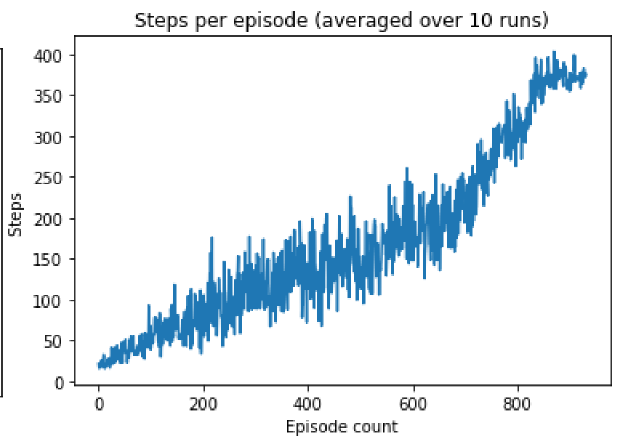


Figure 18: Averaged steps

Observations:

- The average number of episodes required to solve the environment is higher than variant 4.
- Essentially, for variant 5, the learning happens slower than variant 4.

Comments:

We see that the performance does not improve on decreasing the update frequency. Hence, we revert back to the previous value for future variants.

We now investigate the impact of gradient clipping on the performance of the model. Though clipping is done to improve convergence behaviours, exploding gradients are generally a problem only for very deep networks. Since we are dealing with relatively shallow networks, strong gradient clipping may not be necessary. We relax the threshold for clipping and examine the performance.

Variant 6

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$5{\cdot}10^{-4}$
Buffer size	10^5
Batch size	128
Update frequency	20
γ	0.9
Gradient clip	100

Results: Average number of episodes taken to solve the environment: **637.6**

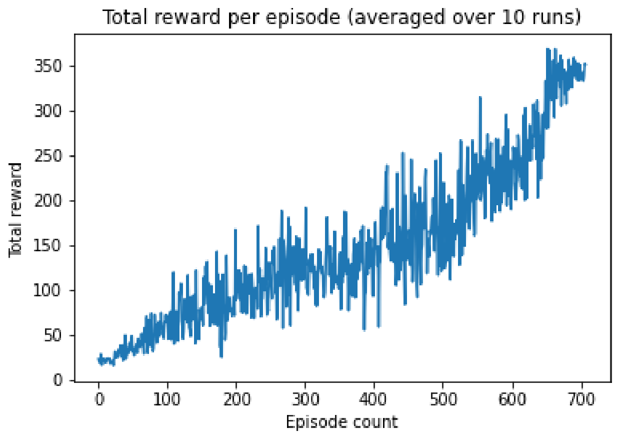


Figure 19: Averaged rewards

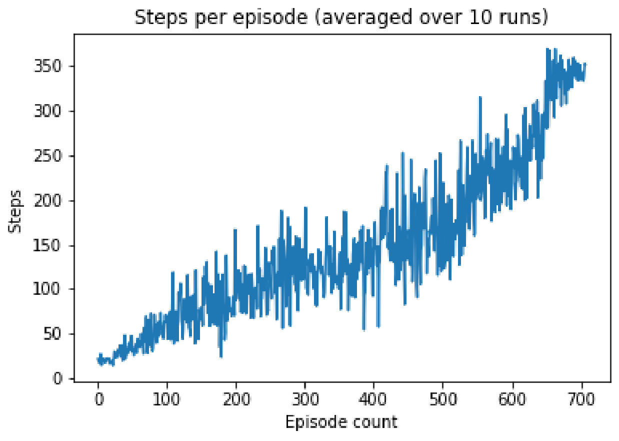


Figure 20: Averaged steps

Observations:

- The average number of episodes required to solve the environment is lower than variant 4.
- Essentially, for variant 6, the learning happens faster than variant 4.

Comments:

We see that the performance improves on increasing gradient clipping to 100. Hence, we maintain this value for future variants. Note that we have now improved the performance of the model **thrice.**

Next we try increasing the learning rate slightly with the hope of speeding up the training process.

Variant 7

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$7.5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	128
Update frequency	20
γ	0.9
Gradient clip	100

Results: Average number of episodes taken to solve the environment: **626.3**

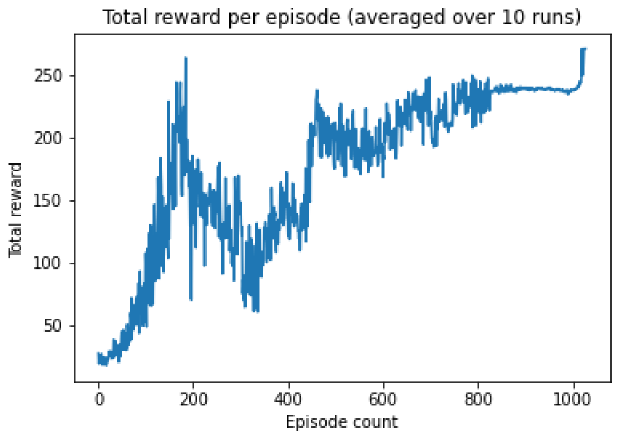


Figure 21: Averaged rewards

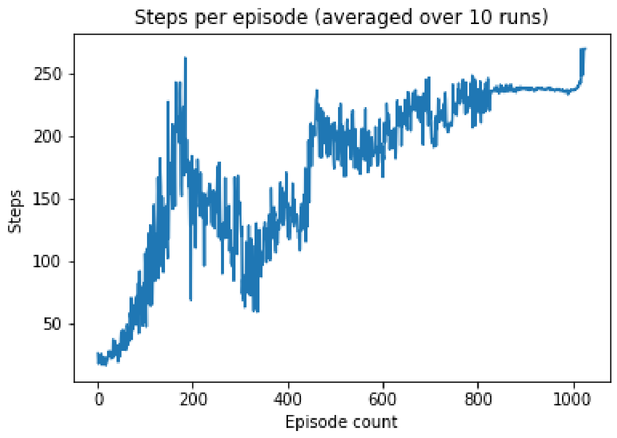


Figure 22: Averaged steps

Observations:

- The average number of episodes required to solve the environment is lower than variant 6.
- Essentially, for variant 7, the learning happens faster than variant 6.

Comments:

We see that the performance improves, though only slightly, on increasing learning rate. Hence, we maintain this value for future variants. Note that we have now improved the performance of the model **four times.**

Next we try increasing the batch size further to 256 as batch size seems to be the most promising hyperparameter for improving the performance of the model.

Variant 8

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$7.5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	256
Update frequency	20
γ	0.9
Gradient clip	100

Results: Average number of episodes taken to solve the environment: **510.2**

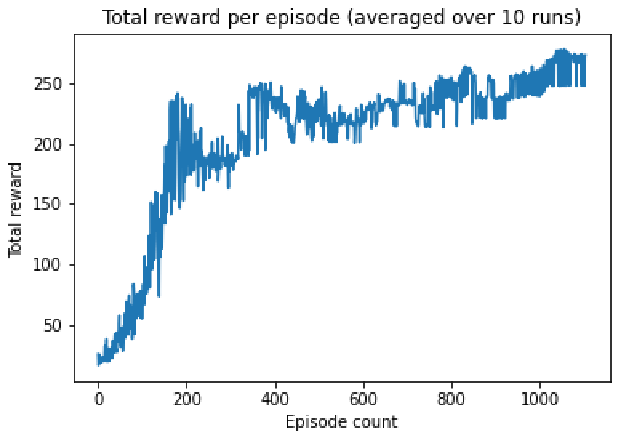


Figure 23: Averaged rewards

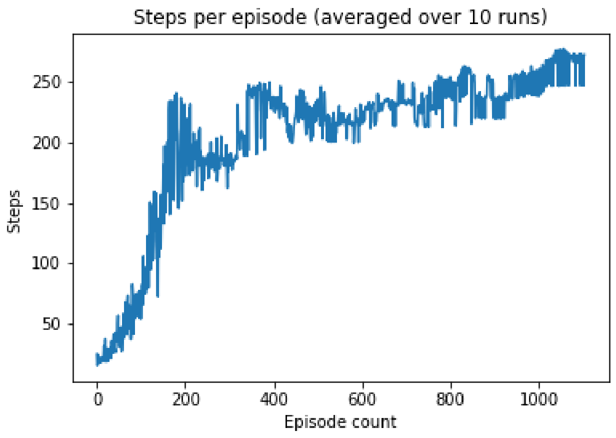


Figure 24: Averaged steps

Observations:

- The average number of episodes required to solve the environment is lower than variant 7.
- Essentially, for variant 8, the learning happens faster than variant 7.

Comments:

We see that the performance improves on increasing batch size further. Hence, we maintain this value for future variants. Note that we have now improved the performance of the model **five times.**

Since we have now improved the performance five times as requested, we stop experiments in this environment. We have improved the average number of episodes required from 940.3 in the default case to 510.2, thereby **solving the environment 45.74% faster.**

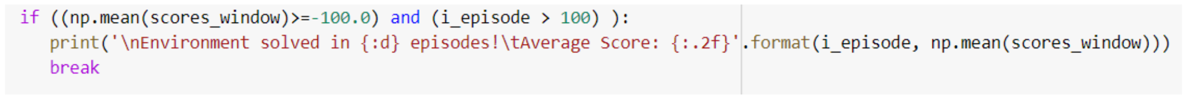
4 DQN: AcroBot-v1

4.1 Environment Description

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.

Reward structure: The goal is to have the free end reach a designated target height in as few steps as possible, and as such all steps that do not reach the goal incur a reward of -1. Achieving the target height results in termination with a reward of 0.

Reward threshold: -100. This means that once the total reward over an episode, averaged over previous 100 episodes, is more than -100, the environment is declared to be solved. This is achieved by the following block of code:



```
if ((np.mean(scores_window)>=-100.0) and (i_episode > 100) ):
    print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
    break
```

Figure 25: Condition for declaring acrobot environment as solved

Steps threshold: 500 This means that an episode can run for as long as 500 steps.

We now give a description of the various configurations we tried, their performance and the reasoning behind the parameter choices made.

4.2 Variant 1 (Tutorial 4 configuration)

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	64
Update frequency	20
γ	0.99
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **828.4**

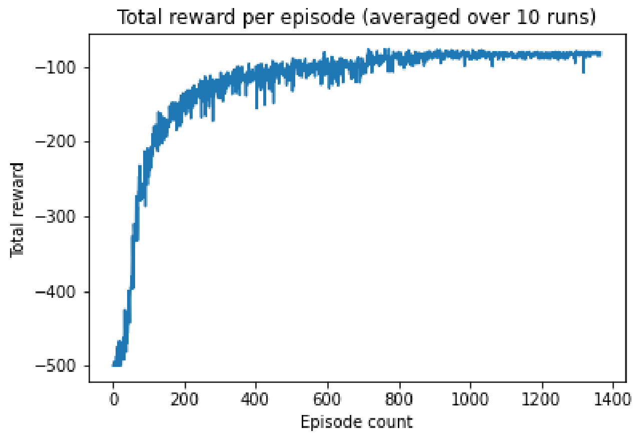


Figure 26: Averaged rewards

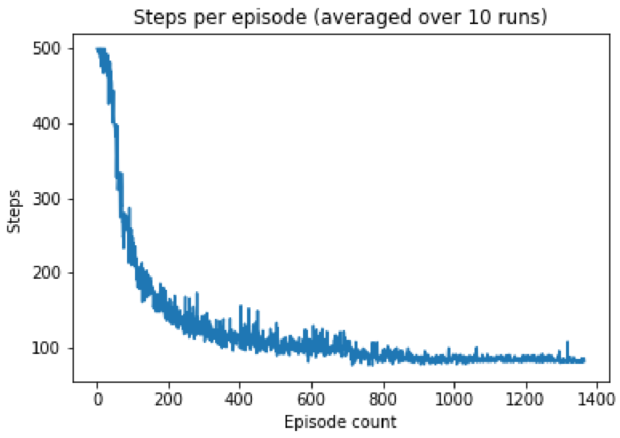


Figure 27: Averaged steps

Observations:

- The reward asymptotically reaches -100. This is expected as this is the reward threshold for solving the environment.
- The number of steps is initially equal to the maximum permissible value of 500 and as the network learns, it reduces till it reaches 100 when solves the environment.
- The reward curve is essentially ”negative” of the step curve since each step gives a reward of -1.

Comments: A good first step towards improving performance would be to verify whether the existing q-network architecture is sophisticated enough to represent the required q-function. To verify this, we use the bigger q-network q2 and check how the performance is affected. We keep the other parameters same.

4.3 Variant 2

Hyperparameter values:

Hyperparameter	Value
Q-Network	q2
Learning Rate	$5{\cdot}10^{-4}$
Buffer size	10^5
Batch size	64
Update frequency	20
γ	0.99
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **986.1**

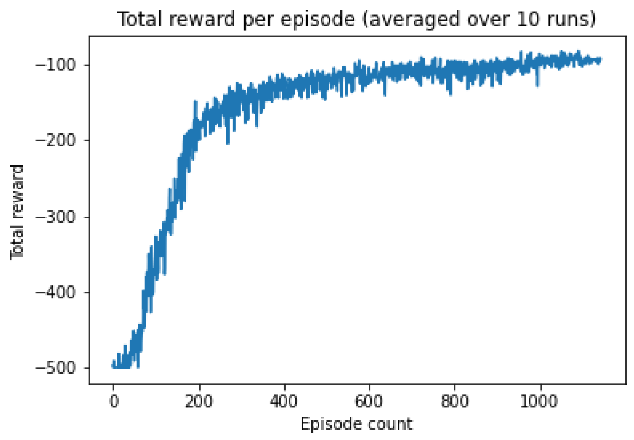


Figure 28: Averaged rewards

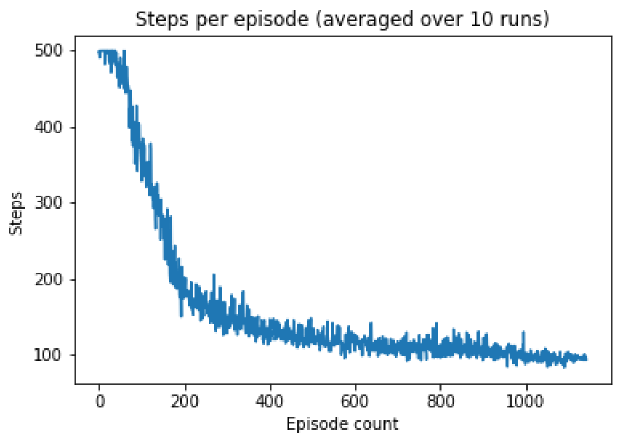


Figure 29: Averaged steps

Observations:

- We observe that for converging to the same average reward as variant 1, variant 2 takes a higher number of average steps.
- The q-network learns **slower** than variant 1.

Comments:

A bigger network does not help improve the performance for this environment. This could be because q1 itself is sufficiently expressive and a more sophisticated network simply slows down training. We revert back to q1.

We now focus on the update frequency of the target network. Decreasing the frequency of updation can help the network better handle the non-stationary target problem. We increase the number of steps between updation from 20 to 40 and observe its effect.

4.4 Variant 3

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	64
Update frequency	40
γ	0.99
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **798.5**

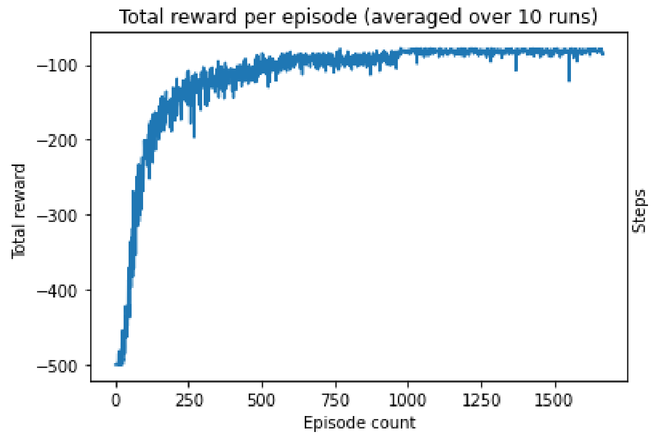


Figure 30: Averaged rewards

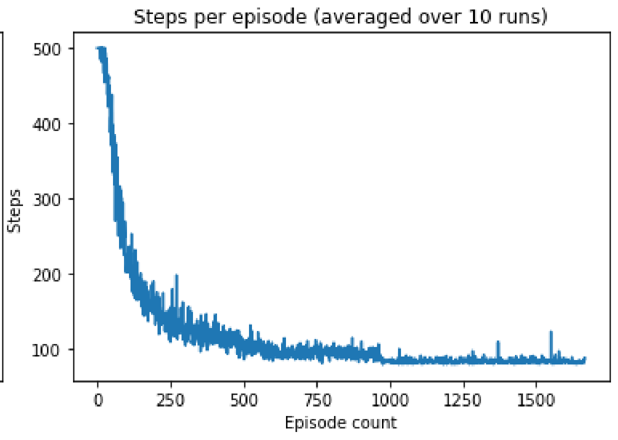


Figure 31: Averaged steps

Observations:

- We observe that for converging to the same average reward as variant 1, variant 3 takes a lower number of average steps.
- The learning is **faster** than variant 1.

Comments:

We observe better performance with lower update frequency as we anticipated. We keep this value. Note that we have now improved the existing model **once**.

We next explore the impact of learning rate. We try increasing it to see if it leads to faster convergence.

4.5 Variant 4

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	10^{-3}
Buffer size	10^5
Batch size	64
Update frequency	40
γ	0.99
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **831.0**

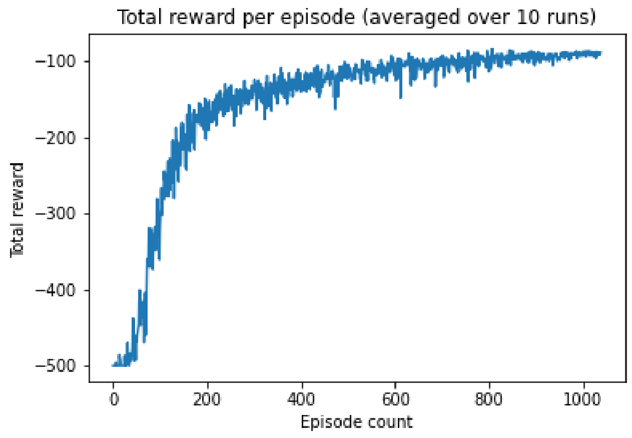


Figure 32: Averaged rewards

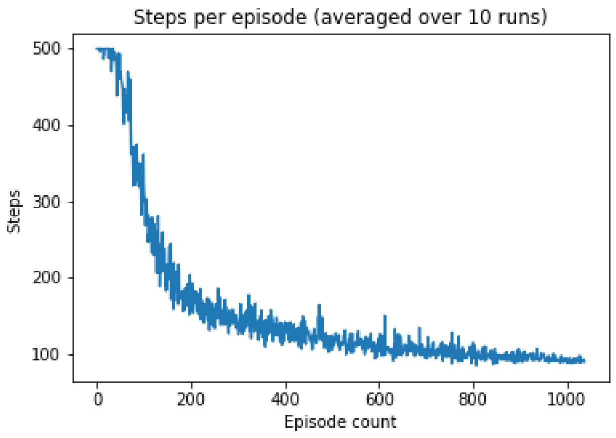


Figure 33: Averaged steps

Observations:

- We observe that for converging to the same average reward as variant 3, variant 4 takes a higher number of average steps.
- The learning is **slower** than variant 3.

Comments:

We observe poorer performance with the higher learning rate. We revert back to the previous value..

Next avenue of exploration would be the discount factor γ. Generally, it is advisable to keep the value between 0.9 and 1. We try increasing γ as we would ideally like to improve long term performance. We also try increasing the training batch size.

4.6 Variant 5

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	128
Update frequency	40
γ	0.999
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **627.2**

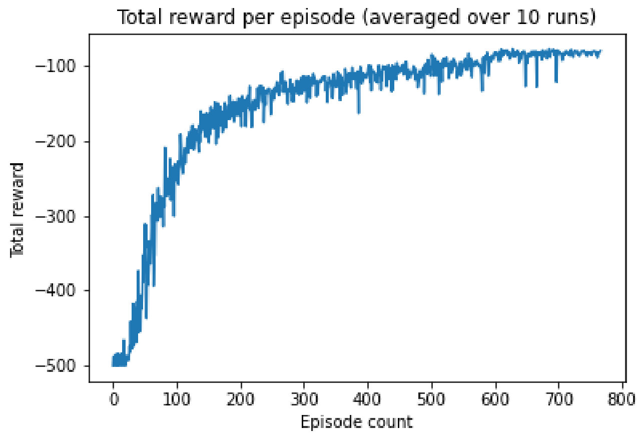


Figure 34: Averaged rewards

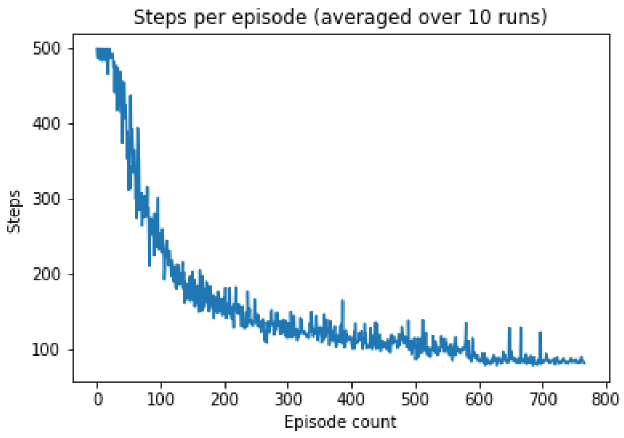


Figure 35: Averaged steps

Observations:

- We observe that for converging to the same average reward as variant 3, variant 5 takes a lower number of average steps.
- The learning is **faster** than variant 3.

Comments:

We observe better performance with higher γ and training batch size. Note that we have now improved the existing model **twice**.

To isolate out the effect of the changes we made, we now try reverting back to the previous value of γ, changing only the training batch size. Our intuition is that increasing batch size would have led to better learning due to better sampling and hence would have been the driving factor behind the improvement in performance.

4.7 Variant 6

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$5{\cdot}10^{-4}$
Buffer size	10^5
Batch size	128
Update frequency	40
γ	0.99
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **567.4**

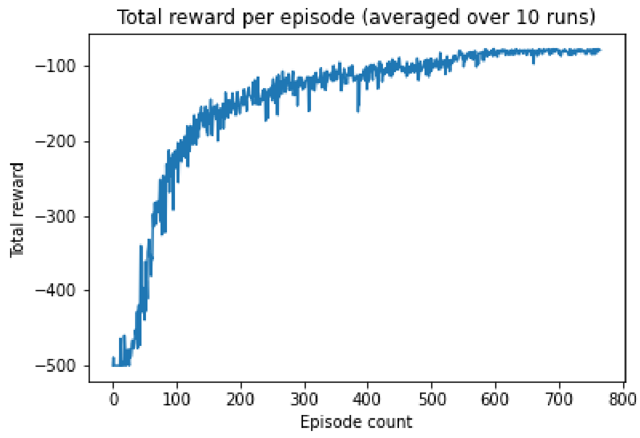


Figure 36: Averaged rewards

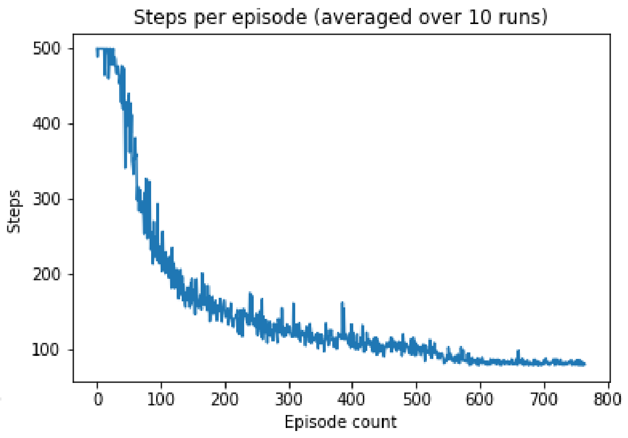


Figure 37: Averaged steps

Observations:

- We observe that for converging to the same average reward as variant 5, variant 6 takes a lower number of average steps.
- The learning is **faster** than variant 5.

Comments:

We observe better performance with lower γ and same training batch size. Note that we have now improved the existing model **thrice**.

We conclude that increasing training batch size is advisable whereas the discount factor should be retained as is. We now observe the effect of increasing the gradient clipping value. Though clipping is said to stabilize training, we believe that since the network itself is small, it may not be necessary. On the contrary, by enforcing a low clipping limit, we might actually be slowing down the training. So we set the clipping value to a much higher value of 100, which limits the gradient value between -100 and 100 instead of -1 and 1.

4.8 Variant 7

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$5{\cdot}10^{-4}$
Buffer size	10^5
Batch size	128
Update frequency	40
γ	0.99
Gradient clip	100

Results: Average number of episodes taken to solve the environment: **511.6**

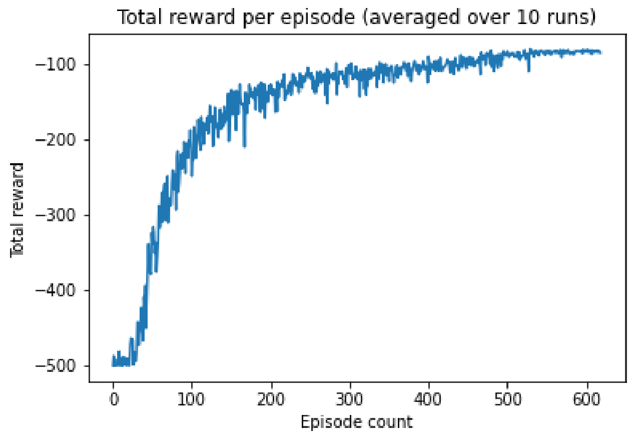


Figure 38: Averaged rewards

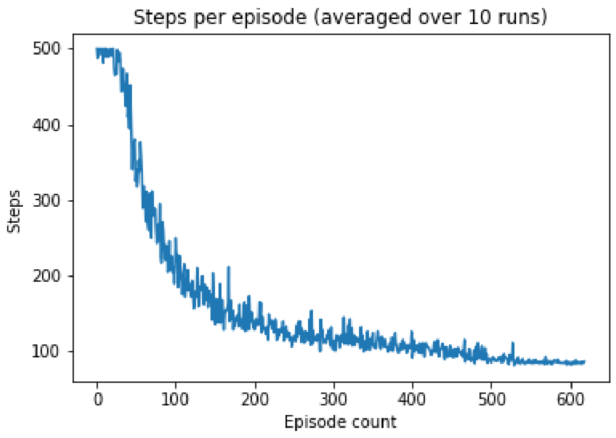


Figure 39: Averaged steps

Observations:

- We observe that for converging to the same average reward as variant 6, variant 7 takes a lower number of average steps.
- The learning is **faster** than variant 6.

Comments:

We observe better performance with a higher value for clipping the gradient. Note that we have now improved the existing model **four times**.

Since we have now explored all the hyperparameters, to further improve performance, we try increasing the batch size further. We focus on batch size as increasing batch size had the highest impact on training performance. Hence, we try doubling the batch size.

4.9 Variant 8

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	256
Update frequency	40
γ	0.99
Gradient clip	100

Results: Average number of episodes taken to solve the environment: **453.3**

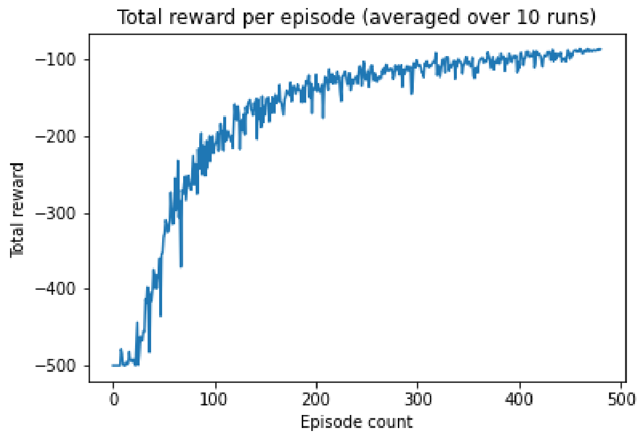


Figure 40: Averaged rewards

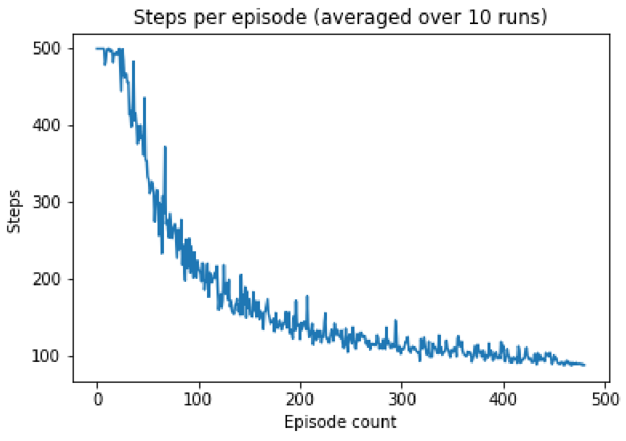


Figure 41: Averaged steps

Observations:

- We observe that for converging to the same average reward as variant 7, variant 8 takes a lower number of average steps.
- The learning is **faster** than variant 7.

Comments:

We observe better performance with a higher value for batch size as expected. Note that we have now improved the existing model **five times**.

Since we have now improved the performance five times as requested, we stop experiments in this environment. We have improved the average number of episodes required from 828.4 in the default case to 453.3, thereby **solving the environment 45.28% faster.**

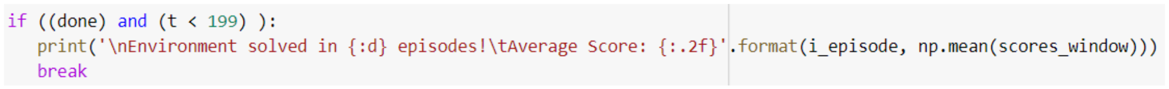
5 DQN: MountainCar-v0

5.1 Environment Description

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.

Reward structure: The goal is to reach the flag placed on top of the right hill as quickly as possible, as such the agent is penalised with a reward of -1 for each timestep.

Reward threshold: There is no particular reward threshold as such for this environment. This means that once car reaches the top of the hill (i.e., when we receive a step reward of 0), the environment is declared to be solved. This achieved by the following block of code:



```
if ((done) and (t < 199) ):
   print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
   break
```

Figure 42: Condition for declaring mountain car environment as solved

Steps threshold: 200 This means that an episode can run for as long as 200 steps.

We now give a description of the various configurations we tried, their performance and the reasoning behind the parameter choices made.

5.2 Variant 1 (Tutorial 4 configuration)

Hyperparameter values:

Hyperparameter	Value
Q-Network	q1
Learning Rate	$5{\cdot}10^{-4}$
Buffer size	10^5
Batch size	64
Update frequency	20
γ	0.99
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **1096.4**

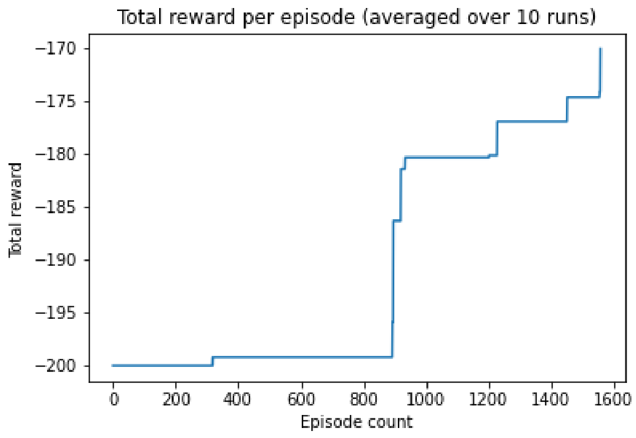


Figure 43: Averaged rewards

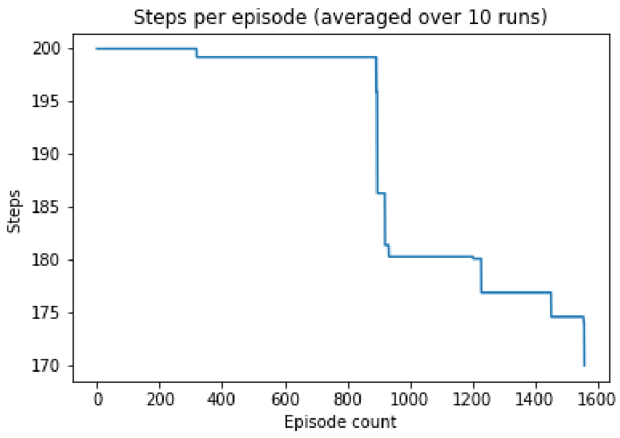


Figure 44: Averaged steps

Observations:

- We do not see the asymptotic behaviour as we saw in the previous two cases. This can be ascribed to the fact that we are only interested in reaching the solution state for each run in this case, whereas in the previous two cases, we were looking at the average rewards over previous 100 steps for each run.
- Different runs take different number of steps to solve the environment. The plots only show the average over 10 runs.
- The reward curve is essentially ”negative” of the step curve since each step gives a reward of -1.

Comments:

A good first step towards improving performance would be to verify whether the existing q-network architecture is sophisticated enough to represent the required q-function. To verify this, we use the bigger q-network q2 and check how the performance is affected. We keep the other parameters same.

5.3 Variant 2

Hyperparameter values:

Hyperparameter	Value
Q-Network	q2
Learning Rate	$5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	64
Update frequency	20
γ	0.99
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **706.0**

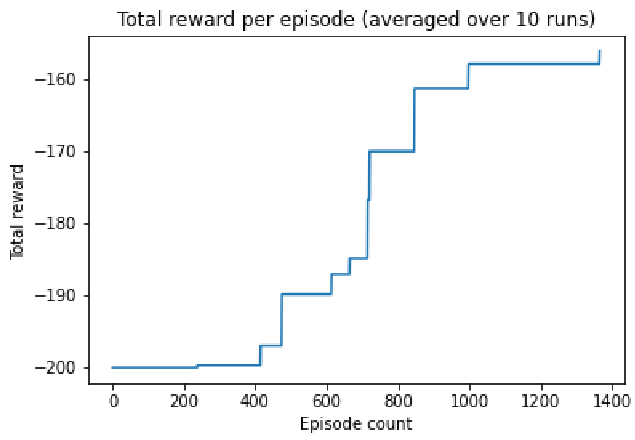


Figure 45: Averaged rewards

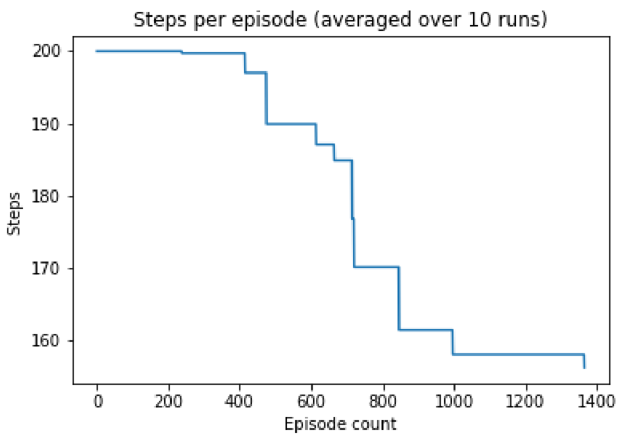


Figure 46: Averaged steps

Observations:

- The plots still have the general step-like behaviour as we saw in variant 1.
- The average number of episodes taken is lower for variant 2 as compared to variant 1.
- Essentially the agent is learning faster as compared to variant 1.

Comments:

The bigger q2 network results is better performance. This indicates that the q function in fact requires a more complex expression than q1. Note that we have now improved the performance **once**.

The logical next step would be to try out the biggest network of the three, q3.

5.4 Variant 3

Hyperparameter values:

Hyperparameter	Value
Q-Network	q3
Learning Rate	$5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	64
Update frequency	20
γ	0.99
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **489.2**

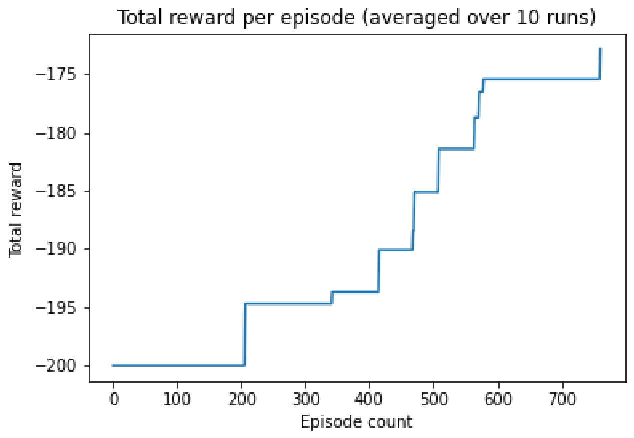


Figure 47: Averaged rewards

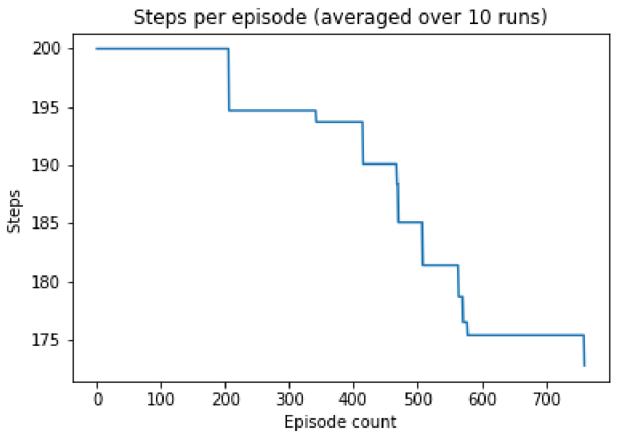


Figure 48: Averaged steps

Observations:

- The plots still have the general step-like behaviour as we saw in previous variants.
- The average number of episodes taken is lower for variant 3 as compared to variant 2.
- Essentially the agent is learning faster as compared to variant 2.

Comments:

The bigger q3 network results is better performance. We thus use q3 for all further experiments. Note that we have now improved the performance **twice**.

The next step would be to evaluate the impact of varying batch size on performance. Intuitively, a larger batch size can lead to better performance as it potentially leads to better target estimates by including more samples.

5.5 Variant 4

Hyperparameter values:

Hyperparameter	Value
Q-Network	q3
Learning Rate	$5{\cdot}10^{-4}$
Buffer size	10^5
Batch size	128
Update frequency	20
γ	0.99
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **484.7**

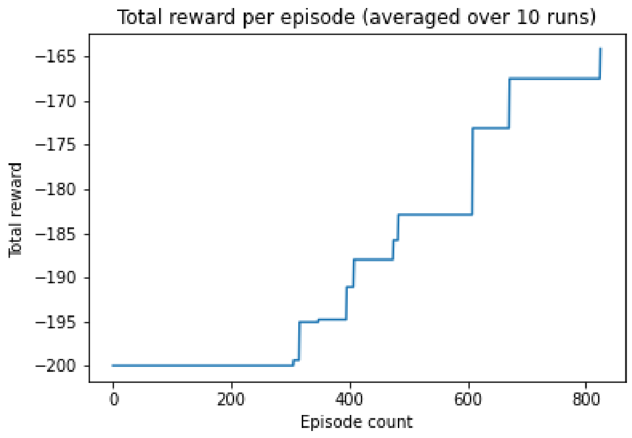


Figure 49: Averaged rewards

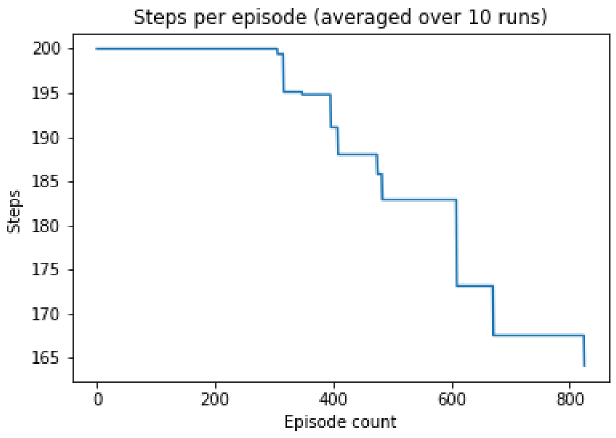


Figure 50: Averaged steps

Observations:

- The plots still have the general step-like behaviour as we saw in previous variants.
- The average number of episodes taken is lower, though only marginally, for variant 4 as compared to variant 3.
- Essentially the agent is learning faster as compared to variant 3.

Comments:

As expected, the higher batch size leads to faster convergence, though the difference is only marginal in this case. Note that we have now improved the performance **thrice**.

Next up, we try decreasing the update frequency of the target network. Previous experiments indicate that this might be a worthy avenue of exploration and hence we try decreasing the update frequency.

5.6 Variant 5

Hyperparameter values:

Hyperparameter	Value
Q-Network	q3
Learning Rate	$5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	128
Update frequency	40
γ	0.99
Gradient clip	1

Results: Average number of episodes taken to solve the environment: **529.8**

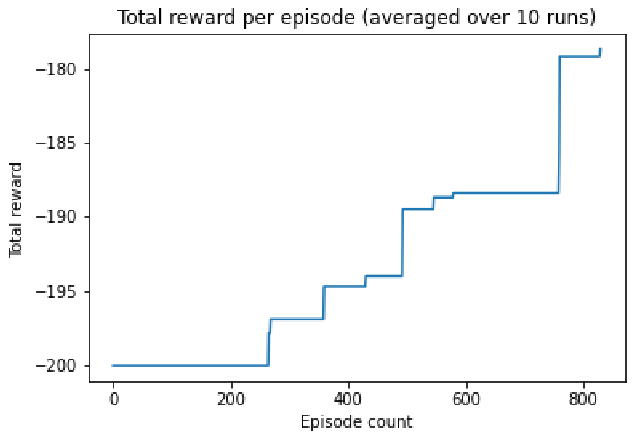


Figure 51: Averaged rewards

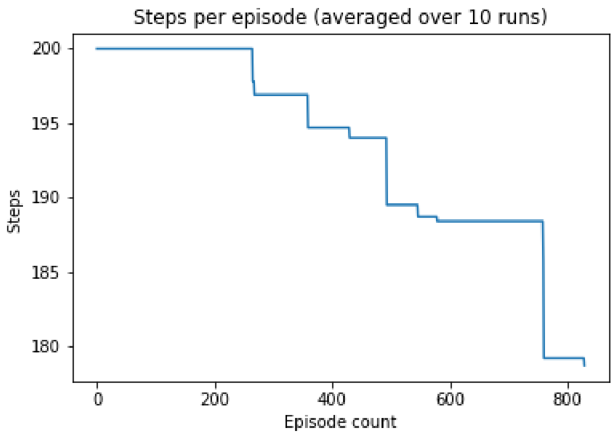


Figure 52: Averaged steps

Observations:

- The plots still have the general step-like behaviour as we saw in previous variants.
- The average number of episodes taken is higher for variant 5 as compared to variant 4.
- Essentially the agent is learning poorly as compared to variant 4.

Comments:

Contrary to what we saw in the previous environment, decreasing update frequency degrades the performance of the model.

Next up, we try setting the gradient clip value to a higher value. As we have seen before, this generally leads to faster convergence for smaller networks as the gradients are closer to their true value due to reduced clipping.

5.7 Variant 6

Hyperparameter values:

Hyperparameter	Value
Q-Network	q3
Learning Rate	$5 \cdot 10^{-4}$
Buffer size	10^5
Batch size	128
Update frequency	20
γ	0.99
Gradient clip	100

Results: Average number of episodes taken to solve the environment: **274.3**

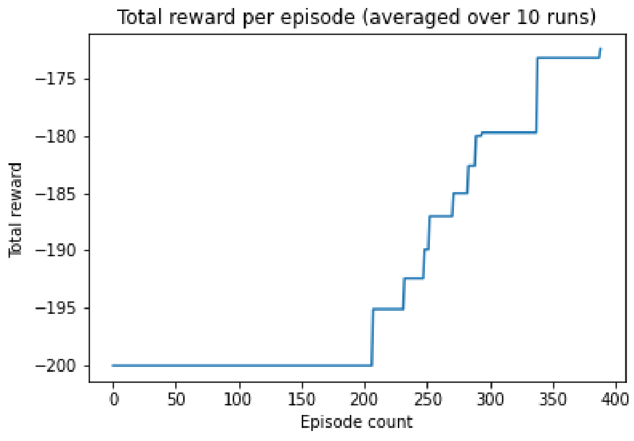


Figure 53: Averaged rewards

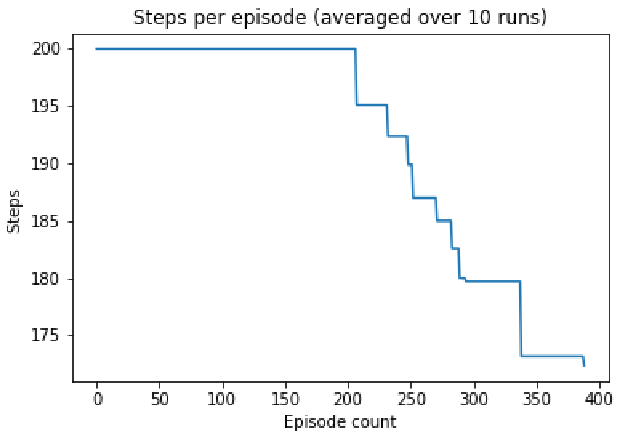


Figure 54: Averaged steps

Observations:

- The plots still have the general step-like behaviour as we saw in previous variants.
- The average number of episodes taken is much lower for variant 6 as compared to variant 4.
- Essentially the agent is learning faster as compared to variant 4.

Comments:

We see that the performance improves by a large margin. Note that we have now improved the performance of the model **four times.**

Next up, we try setting the learning rate to a higher value. Since we are training q3, which is more complex than q1, the hope is that a higher learning rate will lead to faster convergence as the parameter space itself is much larger.

5.8 Variant 7

Hyperparameter values:

Hyperparameter	Value
Q-Network	q3
Learning Rate	10^{-3}
Buffer size	10^5
Batch size	128
Update frequency	20
γ	0.99
Gradient clip	100

Results: Average number of episodes taken to solve the environment: **210.2**

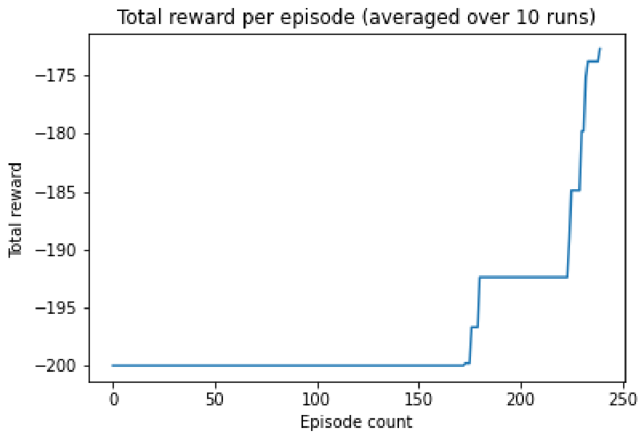


Figure 55: Averaged rewards

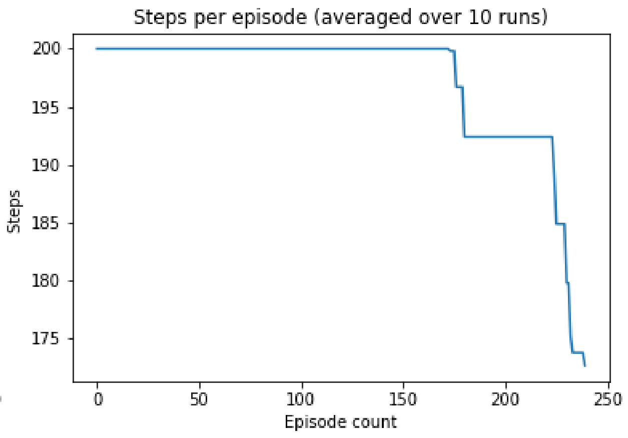


Figure 56: Averaged steps

Observations:

- The plots still have the general step-like behaviour as we saw in previous variants.
- The average number of episodes taken is much lower for variant 7 as compared to variant 6.
- Essentially the agent is learning faster as compared to variant 6.

Comments:

We see that the performance improves as compared to the previous variant. Note that we have now improved the performance of the model **five times.**

Since we have now improved the performance five times as requested, we stop experiments in this environment. We have improved the average number of episodes required from 1096.4 in the default case to 210.2, thereby **solving the environment 80.82% faster.**

6 DQN: Inferences and Conjectures

Based on extensive experimentation on the three environments, we arrive at the following inferences/ conjectures:

- All the experiments were run keeping 10000 episodes as the upper limit. It is interesting to note that none of the experiments exceeded this limit. In fact, the worst case episode count is only 5544 episodes. This attests to the fact that though these classical environments are challenging, **they are solvable in a reasonable amount of time by DQN agents even without tuning.**
- **Irrespective of the environment or the network architecture, a higher training batch size leads to a better performance.** Intuitively, this makes sense as estimating gradients with a larger number of samples is expected to yield better results as opposed to a smaller number of samples. Different variants across environments empirically validate this intuition.
- Relaxing the gradient clipping also leads to better performance in all three environments. While clipping may be necessary for very deep networks, it is safe to come to the conclusion that **for shallow neural networks, gradient clipping results in performance decay rather than performance improvement.**
- **The impact of target update frequency is difficult to predict in advance.** Experimenting different values would be necessary to come to the right value for this hyperparameter.
- **Optimal value of discount factor depends on the environment.** In environments such as cartpole where immediate rewards are positive, a relatively lower value (0.9) of γ works well. In other environments where non-negative rewards are sparse and occur only at the end of the episode, a higher value of γ (0.99) would be better.
- The buffer size by itself does not have any significant effect on the performance of the model, provided that it is sufficiently large.
- The neural network architecture should be sufficiently representative but not too sophisticated. Unnecessarily large networks can hamper performance. The level of sophistication depends on the environment. Typically, more complex environments require more complex neural network architectures along with a higher learning rate.
- **Finally, performance of the DQN itself is highly stochastic.** A particular set of hyperparameters may not work well at all for a completely different random seed. Though averaging can help decrease the impact of stochasticity, it must be kept in mind that results can drastically vary due to stochastic effects.

7 Actor-Critic Methods

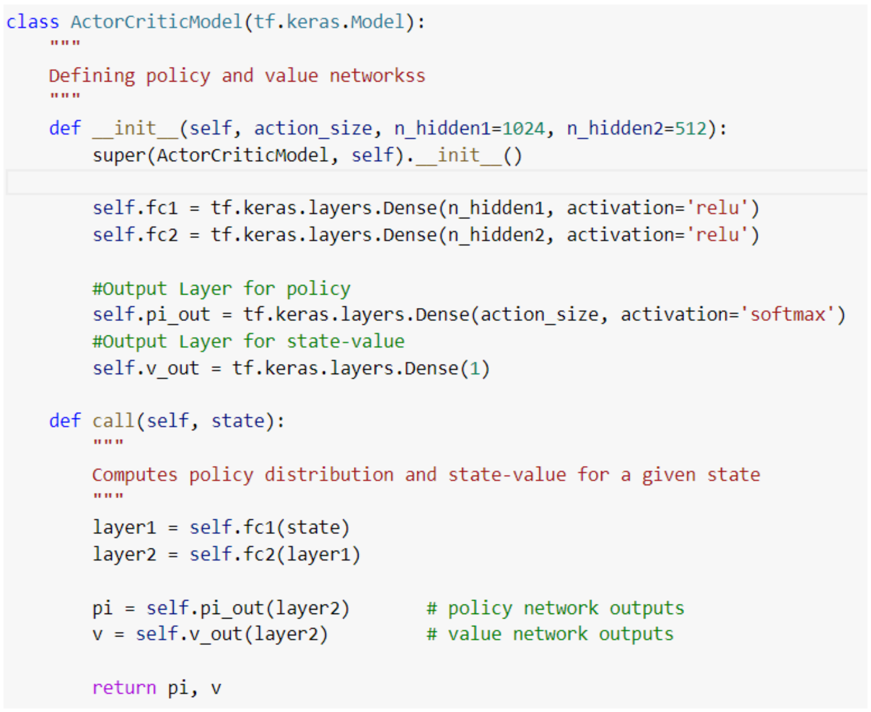
Actor-critic methods are a popular family of reinforcement learning algorithms that combine both value-based and policy-based methods. The actor-critic approach consists of two components: an **actor**, which is responsible for selecting actions given a state, and a **critic**, which estimates the value of the current state.

Actor-Critic methods learn both a **policy** $\pi(a|s;\theta)$ and a **state-value function** $v(s;w)$ simultaneously. The policy is referred to as the actor that suggests actions given a state. The estimated value function is referred to as the critic. It evaluates actions taken by the actor based on the given policy. In this exercise, both functions are approximated by feedforward neural networks.

$$\boxed{\pi(a|s;\theta) = \phi_\theta(a,s)} \tag{1}$$

- The policy network is parametrized by θ - it takes a state s as input and outputs the probabilities $\pi(a|s;\theta)\ \forall\ a$
- The value network is parametrized by w - it takes a state s as input and outputs a scalar value associated with the state, i.e., $v(s;w)$

The actor-critic model can be defined by the below codeblock:



```python
class ActorCriticModel(tf.keras.Model):
    """
    Defining policy and value networkss
    """
    def __init__(self, action_size, n_hidden1=1024, n_hidden2=512):
        super(ActorCriticModel, self).__init__()

        self.fc1 = tf.keras.layers.Dense(n_hidden1, activation='relu')
        self.fc2 = tf.keras.layers.Dense(n_hidden2, activation='relu')

        #Output Layer for policy
        self.pi_out = tf.keras.layers.Dense(action_size, activation='softmax')
        #Output Layer for state-value
        self.v_out = tf.keras.layers.Dense(1)

    def call(self, state):
        """
        Computes policy distribution and state-value for a given state
        """
        layer1 = self.fc1(state)
        layer2 = self.fc2(layer1)

        pi = self.pi_out(layer2)       # policy network outputs
        v = self.v_out(layer2)         # value network outputs

        return pi, v
```

Figure 57: Actor-Critic Model

Note that the size and number of hidden layers in the model may be changed in order to improve the performance of the model.

NOTE: Here, weights of the first two hidden layers are shared by the policy and the value network
- Default hidden layer sizes: [1024, 512]
- Output size of policy network: 2 (Softmax activation)
- Output size of value network: 1 (Linear activation)

7.1 AC: One-Step Algorithms

One-step algorithms in the context of actor-critic (AC) methods are reinforcement learning algorithms that update the policy and the value function estimates based on a **single step** of experience. The policy updates are made based on the estimated return from a single time step.

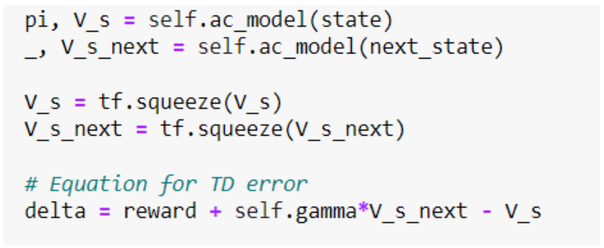
The algorithm is detailed below:

- The actor takes the current state s_t as input and outputs a probability distribution over possible actions, denoted by $\pi(a_t|s_t)$.
- The critic estimates the value function $V(s_t)$ for the current state s_t.
- The actor selects an action a according to the probability distribution $\pi(a_t|s_t)$. The environment transitions to a new state s_{t+1} and rewards the agent with a scalar reward r_{t+1}.
- The critic estimates the value function $V(s_{t+1}|w)$ for the new state s_{t+1}.

The TD error can be calculated as per the equation:

$$\boxed{\delta_t = r_{t+1} + \gamma V(s_{t+1}|w) - V(s_t|w)} \tag{2}$$

The corresponding code block is given below:



```
pi, V_s = self.ac_model(state)
_, V_s_next = self.ac_model(next_state)

V_s = tf.squeeze(V_s)
V_s_next = tf.squeeze(V_s_next)

# Equation for TD error
delta = reward + self.gamma*V_s_next - V_s
```

Figure 58: TD Error (δ)

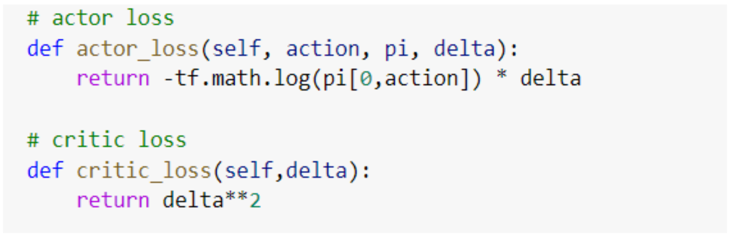
The loss function for the critic is the mean squared error between the estimated value function $V(s_t|w)$ and the target value $r + \gamma V(s')$ and the loss function for the actor is the negative log-likelihood of the selected action multiplied by the temporal difference error δ_t.

$$\boxed{L^{(t)}_{critic} = \delta_t^2} \quad \text{and} \quad \boxed{L^{(t)}_{actor} = -\log \pi(a_t|s_t;\theta)\delta_t} \tag{3}$$

The overall loss for the actor-critic algorithm is the sum of the critic and actor losses:

$$\boxed{L^{(t)}_{tot} = L^{(t)}_{actor} + L^{(t)}_{critic}} \tag{4}$$

The code for this is given below:



```
# actor loss
def actor_loss(self, action, pi, delta):
    return -tf.math.log(pi[0,action]) * delta

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

Figure 59: Actor and Critic Loss

One-step algorithms are **computationally efficient** because they only require one step of experience to update the policy and the value function estimates. They can also be used in environments where the rewards are sparse or delayed because they can estimate the value of future rewards by **bootstrapping** from the value function estimate of the next state.

7.2 AC: Full Returns Algorithm

In the context of actor-critic (AC) methods, the full returns algorithm is a reinforcement learning algorithm that uses **full episode returns** to update the policy and the value function estimates. They use the full trajectory of an episode to estimate the return and update the policy

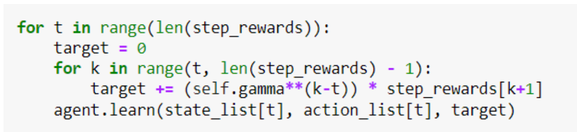
The algorithm is detailed below:

- The actor takes the current state s_t as input and outputs a probability distribution over possible actions, denoted by $\pi(a_t|s_t)$.
- The critic estimates the value function $V(s_t)$ for the current state s_t.
- The actor selects an action a according to the probability distribution $\pi(a_t|s_t)$. The environment transitions to a new state s_{t+1} and rewards the agent with a scalar reward r_{t+1}.
- The critic estimates the value function $V(s_{t+1}|w)$ for the new state s_{t+1}.

The above steps are repeated until the episode ends. At this point we begin the process of calculating loss and updating the model. First we calculate the target value as the continued weighted sum of rewards and calculate the full-return TD error δ as follows:

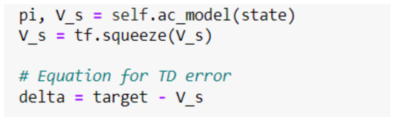
$$\boxed{\delta_t^{(T)} = \sum_{t'=t}^{T} \gamma^{t'-t} \cdot r_{t'+1} - v(s_t|w)} \tag{5}$$

The code to calculate the target and then delta value is given as:



```
for t in range(len(step_rewards)):
    target = 0
    for k in range(t, len(step_rewards) - 1):
        target += (self.gamma**(k-t)) * step_rewards[k+1]
    agent.learn(state_list[t], action_list[t], target)
```

(a) Calculating the target



```
pi, V_s = self.ac_model(state)
V_s = tf.squeeze(V_s)

# Equation for TD error
delta = target - V_s
```

(b) TD Error

Figure 60: Code for calculating target

The loss equations for this will be:

$$\boxed{L_{actor} = -\sum_{t=1}^{T} \log \pi(a_t|s_t;\theta) \cdot \delta_t^{(T)}} \quad \text{and} \quad \boxed{L_{critic} = \sum_{t=1}^{T} \delta_t^{(T)^2}} \tag{6}$$

The total loss is simply a sum of the loss for actor and critic:

$$\boxed{L_{tot} = L_{actor} + L_{critic}} \tag{7}$$

The full returns algorithm can be computationally expensive because it requires waiting for the end of each episode to update the policy and the value function estimates. It is also not suitable for environments with delayed or sparse rewards because the full returns may not provide a reliable estimate of the value function.

7.3 AC: N-Step Returns algorithm

In the context of actor-critic (AC) methods, the n-step return algorithm is a reinforcement learning method used to estimate the value of a state or action by taking into account the rewards received for the next n steps.

In this algorithm, the critic estimates the state or action value using the n-step return. The n-step return is the sum of rewards received over the next n steps, plus the estimated value of the state or action n steps in the future. The value of the state or action is then updated using the difference between the estimated n-step return and the current value estimate.

The algorithm is detailed below:

- The actor takes the current state s_t as input and outputs a probability distribution over possible actions, denoted by $\pi(a_t|s_t)$.
- The critic estimates the value function $V(s_t)$ for the current state s_t.
- The actor selects an action a according to the probability distribution $\pi(a_t|s_t)$. The environment transitions to a new state s_{t+1} and rewards the agent with a scalar reward r_{t+1}.
- The critic estimates the value function $V(s_{t+1}|w)$ for the new state s_{t+1}.

The above steps are repeated for **n-steps**. First we calculate the target value as the continued weighted sum of rewards and calculate the full-return TD error δ as follows:

$$\boxed{\delta_t^{(T)} = \sum_{t'=t}^{n+t-1} \gamma^{t'-t} \cdot r_{t'+1} + \gamma^n v(s_{t+1}|w) - v(s_t|w)} \tag{8}$$

The losses are meanwhile calculated as follows:

$$\boxed{L_{actor}^{(t)} = -\log \pi(a_t|s_t;\theta) \cdot \delta_t^{(n)}} \quad \text{and} \quad \boxed{L_{critic}^{(t)} = \sum_{t=1}^{T} {\delta_t^{(n)}}^2} \tag{9}$$

The total loss is simply a sum of the loss for actor and critic:

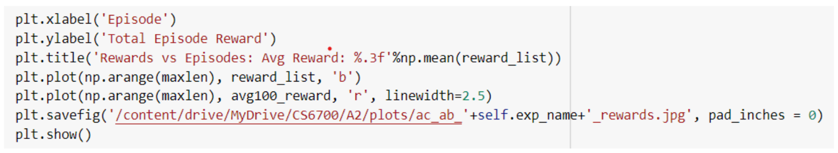
$$\boxed{L_{tot}^{(t)} = L_{actor}^{(t)} + L_{critic}^{(t)}} \tag{10}$$

The n-step return algorithm is a way to balance the trade-off between short-term and long-term rewards. By considering the rewards received over multiple steps, the algorithm can better estimate the value of a state or action and improve the overall performance of the actor-critic method.

7.4 Plotting Rewards

The rewards (averaged over 10 runs) have been plotted in two plots for each setting of hyperparameters. The first plot is that of the average rewards across the 10 runs. Note that this has been reported only until all 10 runs solve the environment. The maximum of the episode lengths is used for this. The moving average of the 10 plots is also shown for comparison.

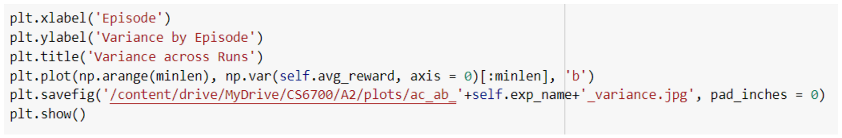
The code for this is given below:



```
plt.xlabel('Episode')
plt.ylabel('Total Episode Reward')
plt.title('Rewards vs Episodes: Avg Reward: %.3f'%np.mean(reward_list))
plt.plot(np.arange(maxlen), reward_list, 'b')
plt.plot(np.arange(maxlen), avg100_reward, 'r', linewidth=2.5)
plt.savefig('/content/drive/MyDrive/CS6700/A2/plots/ac_ab_'+self.exp_name+'_rewards.jpg', pad_inches = 0)
plt.show()
```

Figure 61: Plotting Average Reward vs Episodes across runs

The second plot plots the variance of reward by episode across the 10 runs. This can be plotted as below:



```
plt.xlabel('Episode')
plt.ylabel('Variance by Episode')
plt.title('Variance across Runs')
plt.plot(np.arange(minlen), np.var(self.avg_reward, axis = 0)[:minlen], 'b')
plt.savefig('/content/drive/MyDrive/CS6700/A2/plots/ac_ab_'+self.exp_name+'_variance.jpg', pad_inches = 0)
plt.show()
```

Figure 62: Plotting Reward Variance vs Episodes across runs

7.5 Variance Comparison

The variance of reward across runs in Actor Critic Full-returns algorithms is generally lower than in one-step algorithms.

One-step algorithms, update the policy based on the estimated return from a single time step. This can lead to high variance in the estimated returns, which can in turn lead to **high variance** in the policy updates and ultimately in the agent's behavior.

In contrast, Actor Critic **Full-returns algorithms** use the full trajectory of an episode to estimate the return and update the policy. This can lead to **more accurate estimates of the return** and lower variance in the policy updates, resulting in **more stable learning** and behavior.

The actual variance of reward across runs can depend on many factors, such as the specific implementation of the algorithms, the complexity of the environment, and the amount of exploration during training. However, in general, **Full-returns AC algorithms tend to have lower variance of reward across runs compared to one-step AC algorithms**.

7.6 Hyperparameters

The performance of actor-critic models depends on a variety of hyperparameters, some of which are more critical than others. They are listed below:

- **Learning rate**: This hyperparameter controls the rate at which the model updates its parameters based on the feedback it receives from the environment. A high learning rate can cause the model to overshoot optimal values, while a low learning rate can cause the model to converge slowly or get stuck in local optima.
- **Batch Size**: This hyperparameter controls the number of samples used to update the model parameters in each training iteration. A larger batch size can lead to faster convergence but can also increase the computational cost and memory requirements.
- **Number of hidden layers**: The number of hidden layers in the neural network used by the actor and critic can affect the model's ability to capture complex patterns in the data. A deeper network may be able to represent more complex functions, but it can also lead to overfitting if not regularized properly.
- **Discount Factor**: This hyperparameter determines the importance of future rewards relative to immediate rewards. A high discount factor puts more weight on future rewards, which can lead to long-term planning, while a low discount factor places more emphasis on immediate rewards. For the purpose of this assignment, we use a fixed discount factor of 0.99 for all experiments.

Overall, the choice of hyperparameters for actor-critic models depends on the specific problem being solved and the resources available for training. For our purposes, we shall stick to tuning the learning rate and the number of hidden layers since they are the hyperparameters with the most impact.

An important point to note is that not all experiments conducted have been included in the report for the sake of brevity and because many experiments led to bvery poor performances or failure. The range of viable hyperparameter combinations for this problem was surprisingly low.

8 AC: CartPole-v1

In this section we attempt to solve the OpenAI classic control environment known as "CartPole",

8.1 Environment Description

The environment consists of a cart and a pole that is attached to the cart with a joint. The goal of the agent is to balance the pole on top of the cart for as long as possible. The agent can control the cart by applying a force to move it left or right.

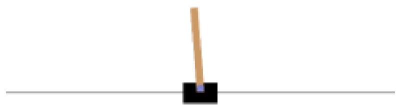


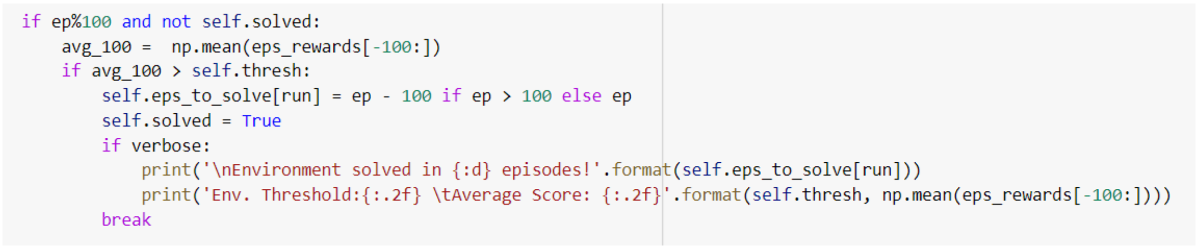
Figure 63: CartPole Visualization

8.2 Reward Structure

The environment provides the agent with feedback in the form of a reward signal, which is +1 for every time step that the pole remains upright, and 0 when the pole falls or the cart moves too far away from the center. The episode ends when the pole falls beyond a certain angle or the cart moves too far away from the center.

8.3 Reward Threshold

The task is considered "solved" when the agent is able to obtain a running average reward of at least 195.0 over 100 consecutive episodes.



```
if ep%100 and not self.solved:
    avg_100 =  np.mean(eps_rewards[-100:])
    if avg_100 > self.thresh:
        self.eps_to_solve[run] = ep - 100 if ep > 100 else ep
        self.solved = True
        if verbose:
            print('\nEnvironment solved in {:d} episodes!'.format(self.eps_to_solve[run]))
            print('Env. Threshold:{:.2f} \tAverage Score: {:.2f}'.format(self.thresh, np.mean(eps_rewards[-100:])))
        break
```

Figure 64: Condition for declaring CartPole environment as solved

Step Threshold Each episode runs for a maximum of 500 time steps.

This environment is a good starting point for RL as it is simple yet challenging.

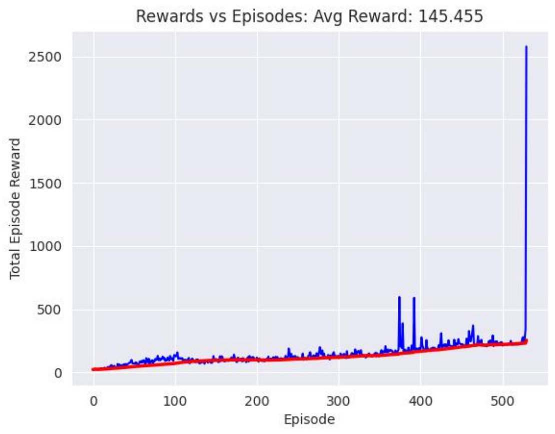
8.4 Experiment 1

Hyperparameter Values

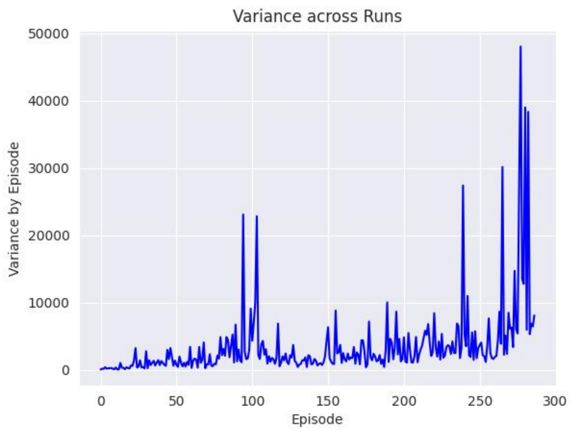
Hyperparameter	Value
Learning Rate	10^{-4}
Hidden Layer 1	512
Hidden Layer 1	256
AC Method (Batch-Size)	One-Step (1)

Results

The average number of episodes taken to solve this environment: **323.70**



(a) Average Rewards vs Episodes over 10 runs



(b) Average Rewards vs Episodes over 10 runs

Figure 65: Experiment 1

Observations

- The number of episodes to solving is sufficiently low
- The variance is also generally low but with concentrated areas of very high peaks
- The agent learns the environment very fast

Comments

As we can see the learning rate seems to be working as the agent learns the environment very well albiet taking a slightly larger number of steps than expected.

A natural avenue of exploration now is to see if using different hidden layer sizes can lead to any improvement.

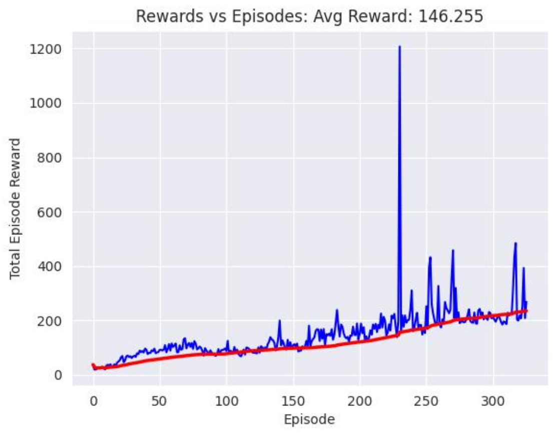
8.5 Experiment 2

Hyperparameter Values

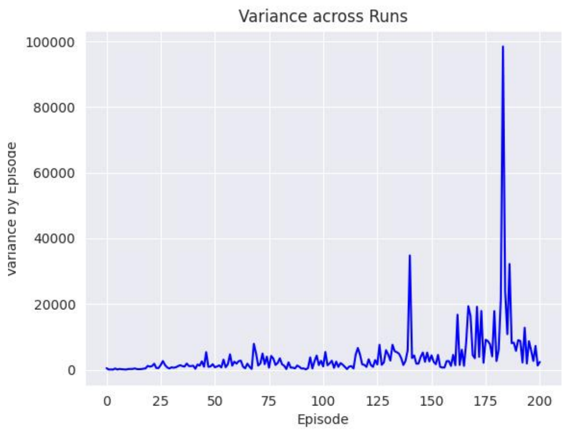
Hyperparameter	Value
Learning Rate	10^{-4}
Hidden Layer 1	1024
Hidden Layer 1	512
AC Method (Batch-Size)	One-Step (1)

Results

The average number of episodes taken to solve this environment: **169.30**



(a) Average Rewards vs Episodes over 10 runs



(b) Average Rewards vs Episodes over 10 runs

Figure 66: Experiment 2

Observations

- The number of episodes to solving is greatly reduced
- The variance is also generally low but with concentrated areas of very high peaks
- The agent learns the environment very fast.

Comments

As we can see increasing the model size causes the agent to learn the environment very well.

We thus continue our exploration to see if using larger hidden layer sizes can lead to any improvement.

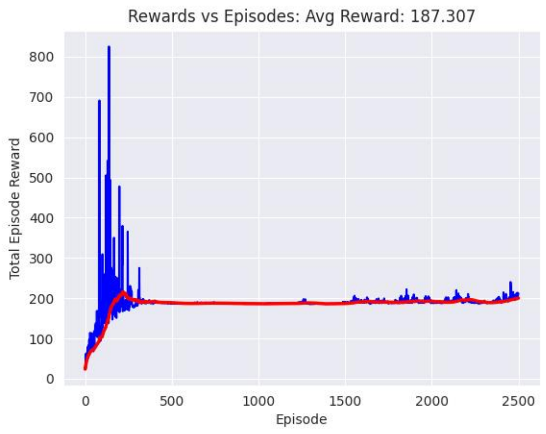
8.6 Experiment 3

Hyperparameter Values

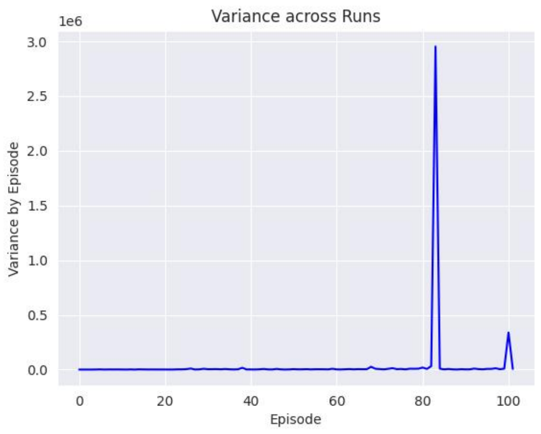
Hyperparameter	Value
Learning Rate	10^{-4}
Hidden Layer 1	2048
Hidden Layer 1	1024
AC Method (Batch-Size)	One-Step (1)

Results

The average number of episodes taken to solve this environment: **325.30**



(a) Average Rewards vs Episodes over 10 runs



(b) Average Rewards vs Episodes over 10 runs

Figure 67: Experiment 3

Observations

- The number of episodes to solving is greatly reduced **but**, in certain runs, the environment remains unsolved.
- The variance is larger for this
- The agent learns the environment very fast

Comments

As we can see the increasing the model size arbitrarily is not very useful as it leads to overfitting. Some runs show exceptional performance but some runs end with the environment remaining unsolved. This is not desirable.

We now revert back to the original hidden layer sizes and see if a change in learning rate can do any good

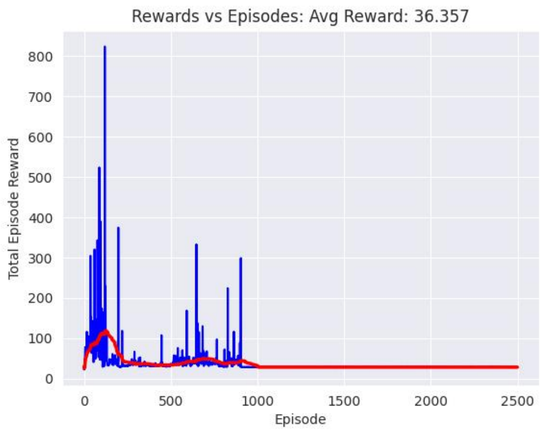
8.7 Experiment 4

Hyperparameter Values

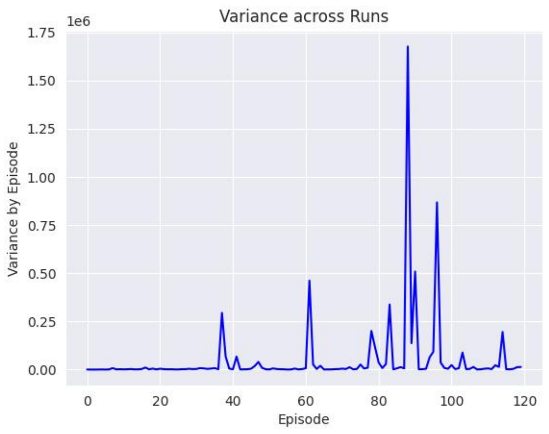
Hyperparameter	Value
Learning Rate	$5 * 10^{-4}$
Hidden Layer 1	1024
Hidden Layer 1	512
AC Method (Batch-Size)	One-Step (1)

Results

The average number of episodes taken to solve this environment: **337.60**



(a) Average Rewards vs Episodes over 10 runs



(b) Average Rewards vs Episodes over 10 runs

Figure 68: Experiment 4

Observations

- The model performs very poorly
- Multiple runs lead to a failure to solve the environment

Comments

As we can see this is the poorest performance we have encountered so far. Sine larger learning rates do not seem to work we attempt to explore in the other direction with smaller learning rates to see if the model can possibly converge better.

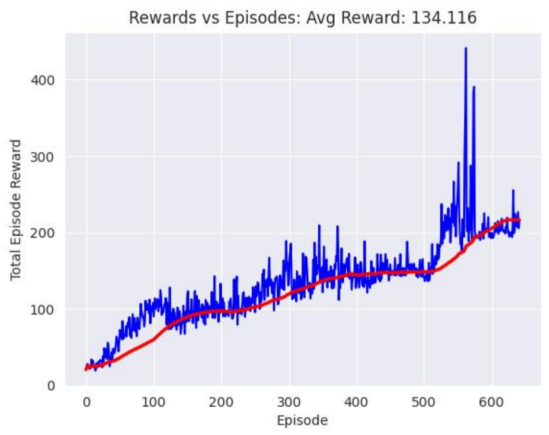
8.8 Experiment 5

Hyperparameter Values

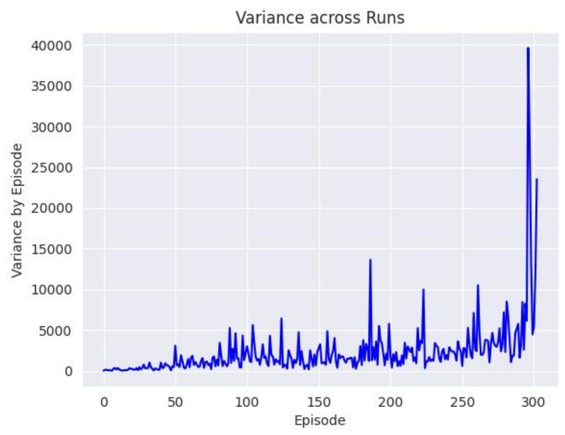
Hyperparameter	Value
Learning Rate	$5 * 10^{-5}$
Hidden Layer 1	1024
Hidden Layer 1	512
AC Method (Batch-Size)	One-Step (1)

Results

The average number of episodes taken to solve this environment: **398.50**



(a) Average Rewards vs Episodes over 10 runs



(b) Average Rewards vs Episodes over 10 runs

Figure 69: Experiment 5

Observations

- The number of episodes higher than that in Exp-2
- The variance is also generally low but with concentrated areas of very high peaks
- The agent learns the environment fast and with reliability

Comments

As we can see decreasing learning rate leads to a smooth convergence to the solution. However the lower learning rate means that the agent takes a larger number of steps (almost double) to reach the solution.

We can end our exploration here and conclude that the hyperparams in exp 2 are the best.

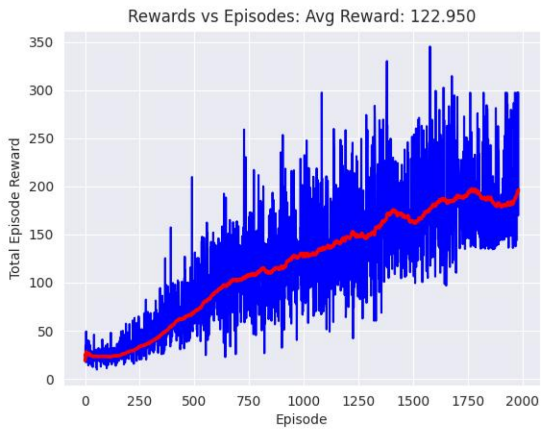
8.9 Experiment 6

Hyperparameter Values

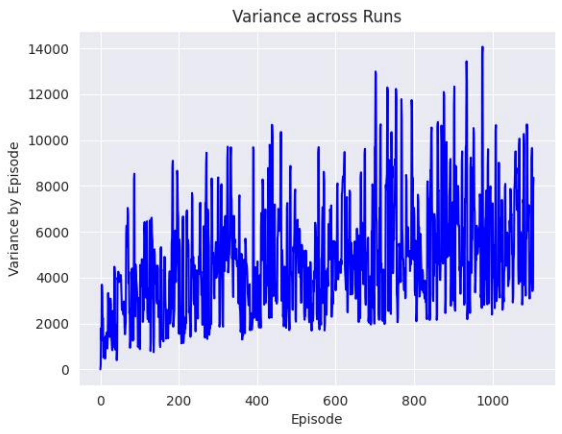
Hyperparameter	Value
Learning Rate	$1 * 10^{-5}$
Hidden Layer 1	1024
Hidden Layer 1	512
AC Method (Batch-Size)	Full-Return (upto 500)

Results

The average number of episodes taken to solve this environment: **1688.70**



(a) Average Rewards vs Episodes over 10 runs



(b) Average Rewards vs Episodes over 10 runs

Figure 70: Experiment 6

Observations

- The number of episodes significantly higher
- The variance is also generally low compared to one-step algorithms
- The agent learns the environment very slowly but reliably

Comments

This is a good initial performance. We now attempt to better the performance of the model

A natural avenue of exploration is to try to modify the model size and see how the model performs.

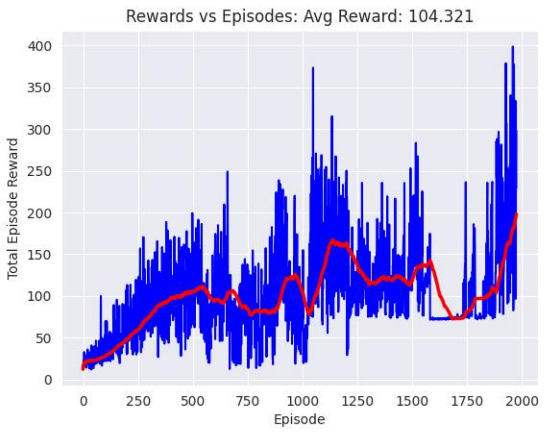
8.10 Experiment 7

Hyperparameter Values

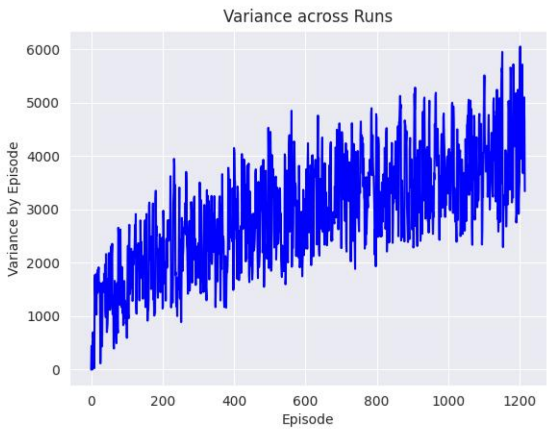
Hyperparameter	Value
Learning Rate	$1 * 10^{-5}$
Hidden Layer 1	2048
Hidden Layer 1	1024
AC Method (Batch-Size)	Full-Return (upto 500)

Results

The average number of episodes taken to solve this environment: **1357.30**



(a) Average Rewards vs Episodes over 10 runs



(b) Average Rewards vs Episodes over 10 runs

Figure 71: Experiment 7

Observations

- The number of episodes has decreased significantly
- The variance is also generally low compared to one-step algorithms, and also slightly lower than before
- The agent learns the environment slowly but reliably

Comments

This is an improvement over the previous model and so we try to see if it can be further improved upon by lowering the learning rate slightly and maintaining the same neural network model

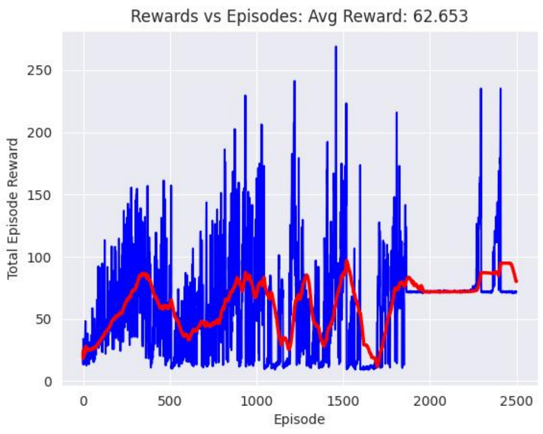
8.11 Experiment 8

Hyperparameter Values

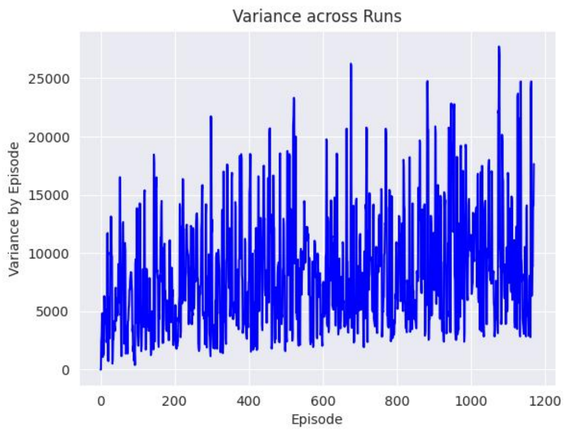
Hyperparameter	Value
Learning Rate	$5 * 10^{-6}$
Hidden Layer 1	1024
Hidden Layer 1	512
AC Method (Batch-Size)	Full-Return (upto 500)

Results

The average number of episodes taken to solve this environment: **2255.30**



(a) Average Rewards vs Episodes over 10 runs



(b) Average Rewards vs Episodes over 10 runs

Figure 72: Experiment 8

Observations

- The number of episodes significantly higher
- The agent fails to learn the environment in a number of cases and in others it fails to do so in a reasonable amount of time.

Comments

This is a very poor performance and thus we conclude that changing the learning rate or model size will not greatly improve the performance sigificantly. In all our experiments, Exp 2 has the best results.

9 AC: Acrobot-v1

In this section we attempt to solve the OpenAI classic control environment known as ”Acrobot”,

9.1 Environment Description

The Acrobot environment in the OpenAI Gym Classic Control suite is a physics-based simulation task where the goal is to swing up a two-link robot arm, known as the Acrobot, upto a certain height. The Acrobot consists of two links, which are joined by a joint, and an actuator that can apply torque to the joint.

The state of the environment is defined by the position and velocity of the two links of the Acrobot. The action space is discrete, consisting of two possible actions: applying positive torque to the joint or applying negative torque to the joint.

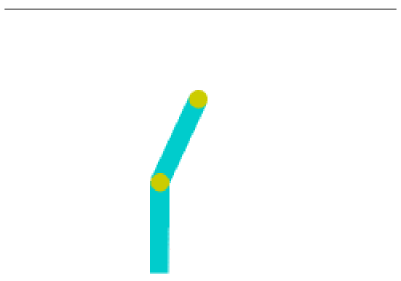


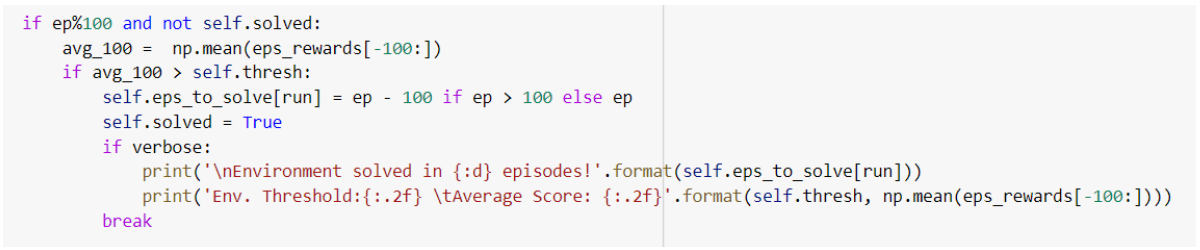
Figure 73: Acrobot Visualization

9.2 Reward Structure

The reward function is defined as follows: a reward of -1 is given for each timestep until the Acrobot reaches the upright position, at which point a reward of 0 is given.

9.3 Reward Threshold

The task is considered “solved” when the agent is able to obtain a running average reward of at least -100.0 over 100 consecutive episodes.



```
if ep%100 and not self.solved:
    avg_100 =  np.mean(eps_rewards[-100:])
    if avg_100 > self.thresh:
        self.eps_to_solve[run] = ep - 100 if ep > 100 else ep
        self.solved = True
        if verbose:
            print('\nEnvironment solved in {:d} episodes!'.format(self.eps_to_solve[run]))
            print('Env. Threshold:{:.2f} \tAverage Score: {:.2f}'.format(self.thresh, np.mean(eps_rewards[-100:])))
        break
```

Figure 74: Condition for declaring Acrobot environment as solved

Step Threshold Each episode runs for a maximum of 500 time steps.

This environment is a good starting point for RL as it is simple yet challenging.

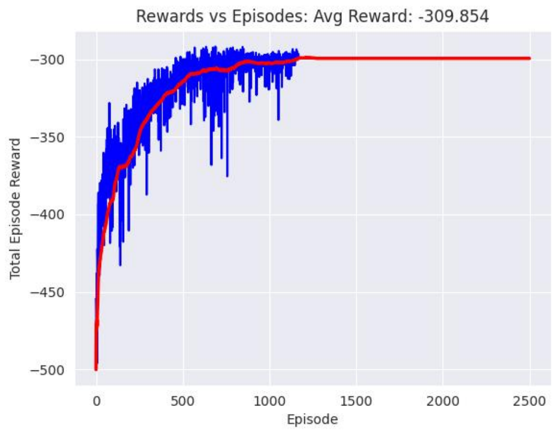
9.4 Experiment 1

Hyperparameter Values

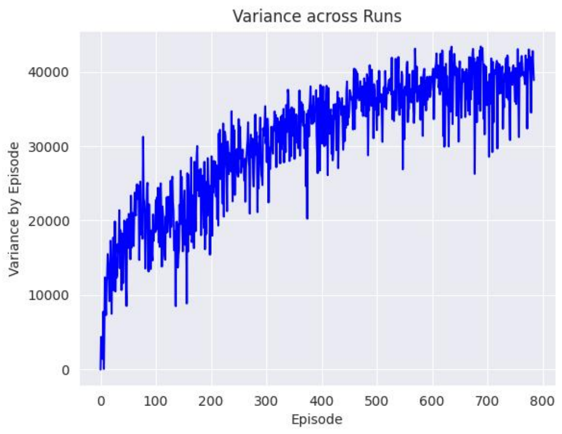
Hyperparameter	Value
Learning Rate	10^{-5}
Hidden Layer 1	512
Hidden Layer 1	256
AC Method (Batch-Size)	One-Step (1)

Results

The average number of episodes taken to solve this environment: **1502.60**



(a) Average Rewards vs Episodes over 10 runs



(b) Average Rewards vs Episodes over 10 runs

Figure 75: Experiment 1

Observations

- The number of episodes to solving is very high
- The variance is not very low and has moderate values
- The agent learns the environment but is unreliable

Comments

As we can see the learning rate seems to be non-optimal since the environment is solved very slowly and in a number of runs it is even left unsolved.

A natural avenue of exploration now is to see if using different learning rates and a bigger model can lead to any improvement.

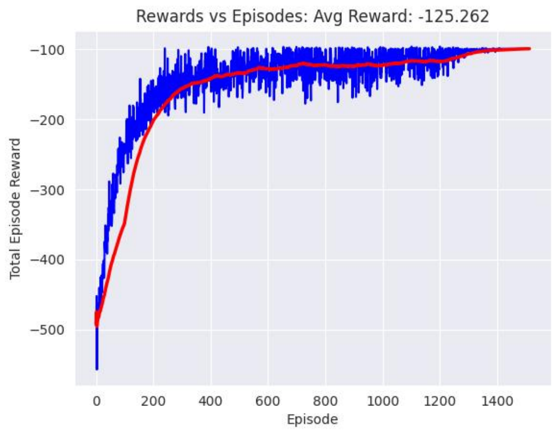
9.5 Experiment 2

Hyperparameter Values

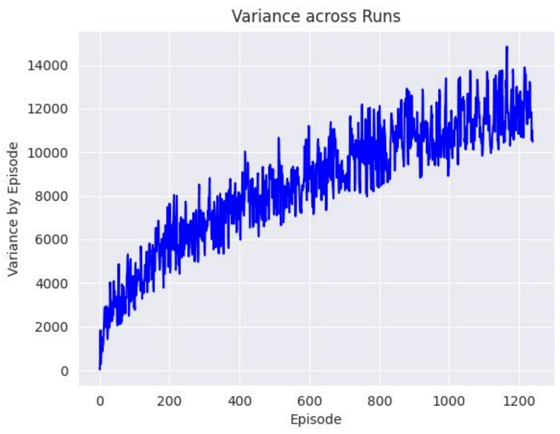
Hyperparameter	Value
Learning Rate	$5 * 10^{-6}$
Hidden Layer 1	1024
Hidden Layer 1	512
AC Method (Batch-Size)	One-Step (1)

Results

The average number of episodes taken to solve this environment: **1241.90**



(a) Average Rewards vs Episodes over 10 runs



(b) Average Rewards vs Episodes over 10 runs

Figure 76: Experiment 2

Observations

- The number of episodes is improved but still high
- The variance follows a similar pattern
- The agent learns the environment reliably but is still very slow

Comments

As we can see the learning rate seems to be non-optimal since the environment is solved very slowly but there is a significant improvement.

Next we explore the effect of increasing the learning rate for the same model.

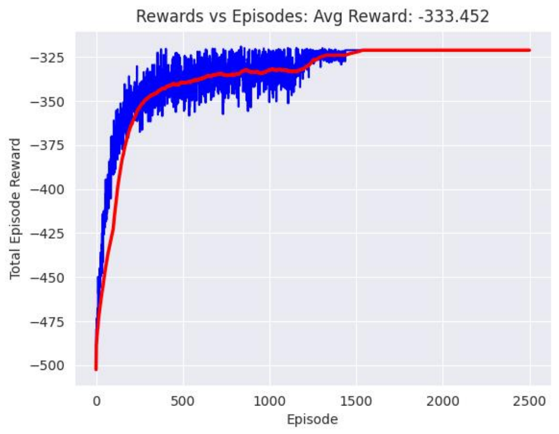
9.6 Experiment 3

Hyperparameter Values

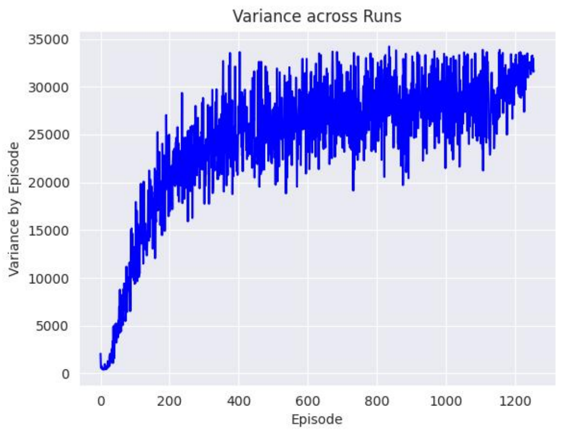
Hyperparameter	Value
Learning Rate	$5 * 10^{-5}$
Hidden Layer 1	1024
Hidden Layer 1	512
AC Method (Batch-Size)	One-Step (1)

Results

The average number of episodes taken to solve this environment: **1759.70**



(a) Average Rewards vs Episodes over 10 runs



(b) Average Rewards vs Episodes over 10 runs

Figure 77: Experiment 3

Observations

- The number of episodes is very high and in many cases the environment remains unsolved.
- The variance follows a similar pattern as previous experiment
- The agent learns the environment unreliably and is still very slow

Comments

This is one of the worst cases we have seen so far and a great deal of improvement is required to give better results. We conjecture that the higher learning rate was causing a significant loss

Next we explore the effect of increasing the model size for the previous experiment (Expt-2) which showed promise.

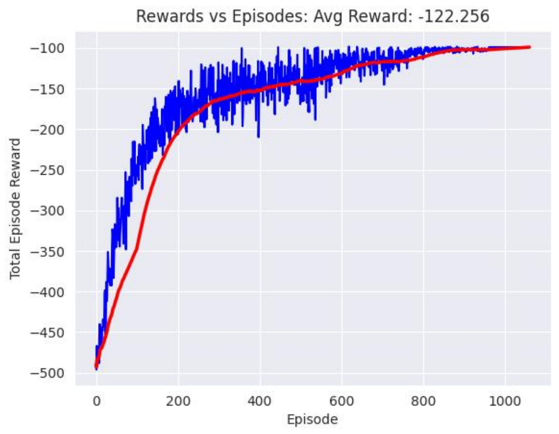
9.7 Experiment 4

Hyperparameter Values

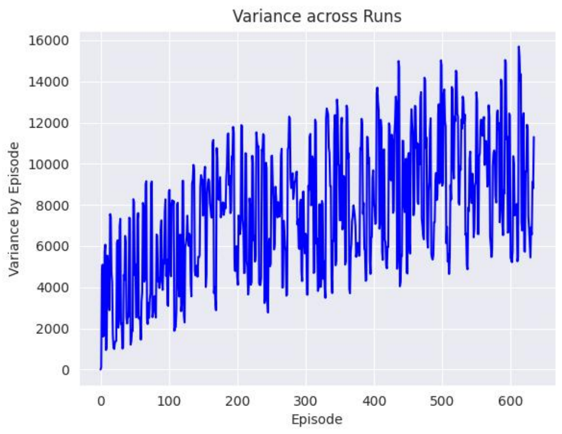
Hyperparameter	Value
Learning Rate	$5 * 10^{-6}$
Hidden Layer 1	2048
Hidden Layer 1	1024
AC Method (Batch-Size)	One-Step (1)

Results

The average number of episodes taken to solve this environment: **689.60**



(a) Average Rewards vs Episodes over 10 runs



(b) Average Rewards vs Episodes over 10 runs

Figure 78: Experiment 4

Observations

- The number of episodes is significantly decreased
- The variance follows a similar pattern as previous experiments but with lower overall values
- The agent learns the environment very well

Comments

This is one of the best cases we have seen so far and a great deal of improvement is made. We conjecture that the much bigger model was causing a significant gain in performance

Next we explore the effect of decreasing the model size to a moderate sized model.

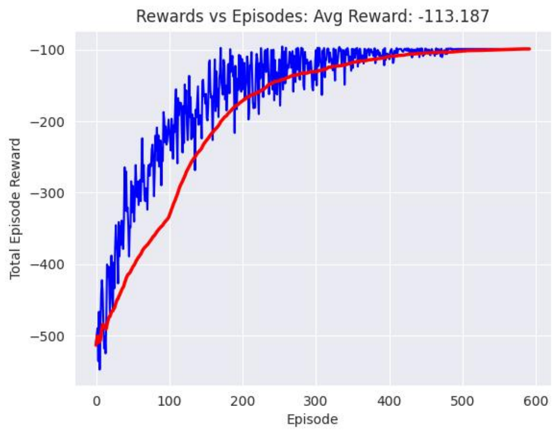
9.8 Experiment 5

Hyperparameter Values

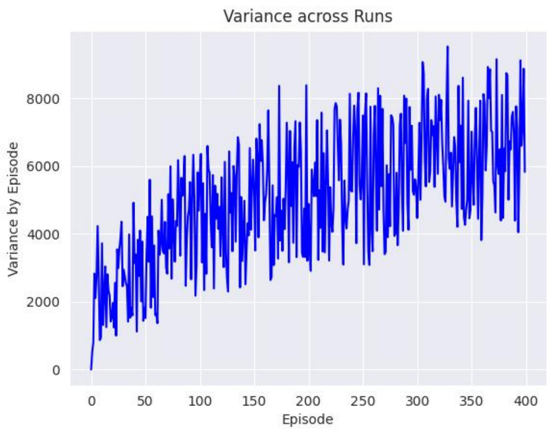
Hyperparameter	Value
Learning Rate	$5 * 10^{-6}$
Hidden Layer 1	2048
Hidden Layer 1	1024
AC Method (Batch-Size)	One-Step (1)

Results

The average number of episodes taken to solve this environment: **377.30**



(a) Average Rewards vs Episodes over 10 runs



(b) Average Rewards vs Episodes over 10 runs

Figure 79: Experiment 5

Observations

- The number of episodes is significantly decreased
- The variance follows a similar pattern as previous experiments but with lower overall values
- The agent learns the environment very well

Comments

This is one of the best cases we have seen so far and a great deal of improvement is made. We can see that the intermediate sized model has the best performance out of all the models so far.

We arrest tuning at this point and accept this as the best model.

10 AC Methods: Inferences and Conjectures

Based on extensive experimentation on the three environments, we arrive at the following inferences/ conjectures:

- All the experiments were run keeping 2500 episodes as the upper limit. It is interesting to note that many of the experiments exceeded this limit especially for poor combinations of hyperparameters. Separate experiments conducted without this upper limit showed that even hyperparaameter configurations which occasionally overshoot 2500 episodes are generally solvable in under 5000 episodes with the worst case episode count being 4688. This attests to the fact that though these classical environments are challenging, **they are solvable in a asymptotic time by AC agents even without tuning.**
- **Irrespective of the environment or the network architecture, using the full-returns algorithm instead of the one-step algorithm leads to a great decrease in the variance** Intuitively, this makes sense as estimating gradients with a larger number of samples is expected to yield low-variance results as opposed to a smaller number of samples.
- As a general rule we observed that smaller sized models perform relatively poorly when faced with any of the control tasks. On the other hand, **larger models** can capture more complex and higher-order relationships between the input features and the output. This can lead to more accurate value function estimates and **better performance** on tasks that require more sophisticated control strategies. However a tradeoff is that when the size of the models increases too much, it leads to **overfitting** and poor generalization which can cause the **control to become unstable**. This is demonstrated by large models which give unreliable performance, solving certain environments in very little time and at other times, never arriving at a solution at all. We can see that the **model with sizes of hidden layers** $(1024, 512)$ **performs exceedingly well across environments** when tuned for the learning rate.
- Optimal value of discount factor depends on the environment. In environments such as cartpole where immediate rewards are positive, a relatively lower value $\gamma = 0.90$ can also work well. In other environments where non-negative rewards are sparse and occur only at the end of the episode, a higher value of $\gamma = 0.99$ would be better.
- Finally, performance of the AC models itself is highly stochastic. A particular set of hyperparameters may not work well at all for a completely different random seed. Though averaging can help decrease the impact of stochasticity, it must be kept in mind that results can drastically vary due to stochastic effects.