Programming Assignment 2

Purvam Jain EE20B101, Martin Reich ME23F201

November 13, 2023

Contents

1 Abstract

This report is structured into two parts. The first section contains the approaches taken for DQN. The second part discusses the approaches and observations for Actor-Critic.

2 DQN

The first training of a DQN is always done with the hyperparameters from the tutorial (table 1) and a single run to get a sense of how fast a convergence is reached. When changing the hyperparameters, this is always done by averaging over 5 runs to get a representative score while being efficient with computation time. The seed value is kept fix at 0, as this gives a better way of reproducing the results. However, for a general notion the seed value could be changed and an average could be calculated over all runs over all seed values. Each datapoint in the plot is an average of the last ten values of the episode.

2.1 Hyperparameter settings

Training a DQN, multiple hyperparameters can be set. The general implications of value ranges for the hyperparameters are discussed in the following.

1. **Learning rate:** If the learning rate is increased, the learning speed might increase at the beginning since the emphasize on the values of new samples is higher. However, the variance might increase as well. The learning rate should be decreased towards the end of the learning to avoid strong fluctuations around the optimum.

2. **Batch size:** Decreasing the batch size leads to faster convergence, however higher variance. A bigger batch size might slow down the learning but will lead to results closer to the global optimum due to lower variance. Increasing the batch size too far will lead to poor generalization due to overfitting on the training data.

3. **Update frequency:** If the update frequency is too high, the learning will be slow because the temporary target is not the true target. However, if the update frequency is too low, the model cannot converge the the target since the target changes too often, hence, the variance will be high.

4. **Discount factor:** The higher the discount factor, the stronger the emphasize on the overall goal which implicates a more farsighted model.

5. **Replay buffer size:** The larger the replay buffer size, the less likely to train on correlated elements (i.e. consecutive elements from the same observation). Additionally, it is less likely to forget about "bad" trajectories, as the updated model does not take the according actions anymore and they are deleted from the FIFO replay buffer. If the buffer size is too big,

it might slow down the learning, as the learning is slower to adapt to good trajectories because the likelihood of picking them is slower.

6. **Network architecture:** Whole books can be filled with the network architecture. In general, linear approximators are stacked over each other with non linear activation functions. The more linear approximators, the more successful the approximation of the ideal functions will be but the longer the training will take as more weights need to be updated. Different loss functions are available as for example MSE loss. However, Huber loss is suppose to be more robust to outliers since it does not square the difference above a threshold δ but accounts for the difference linearly, implying a usage in noisy, outlier prone environments.

There can be different metrics to evaluate an agent's performance. The number of episodes taken to solve the environment is chosen as a metric for the evaluation of an agent's performances. The number of input and output nodes corresponds to the number of input states and actions respectively in each environment.

Parameter	**value**
Replay buffer size	10000
Batch size	64
Learning rate	0.0005
Update frequency	20
Hidden Layer 1	128
Hidden Layer 2	64
Loss function	MSE

Table 1: List of hyperparameters from the tutorial

2.2 Acrobot-v1

The DQN network was trained once with the parameters given in table 1. The training was done over 10000 episodes to get a notion of how good the model is. The reward increased starting from -500 (max episode length), while fluctuating around an average reward of -75, as can be seen in figure 1. A positive reward is not going to be expected, since the reward is -1 for each step and 0 for reaching the goal line (swinging up), thus reaching the goal as fast as possible. The process of swinging up apparently needs around 75 steps. This is reached in less than 1000 episodes, thus setting the benchmark for increasing the performance with different hyperparameters. Figure 2 shows the average over 5 runs. The changes in hyperparameters are depicted in table 2. Note: Parameter sets 3 and 4 were stopped early, as the results forseeably were not better.

Inferences: The acrobot was trained with different hyperparameter combinations. The hyperparameters from the tutorial could not be significantly beaten

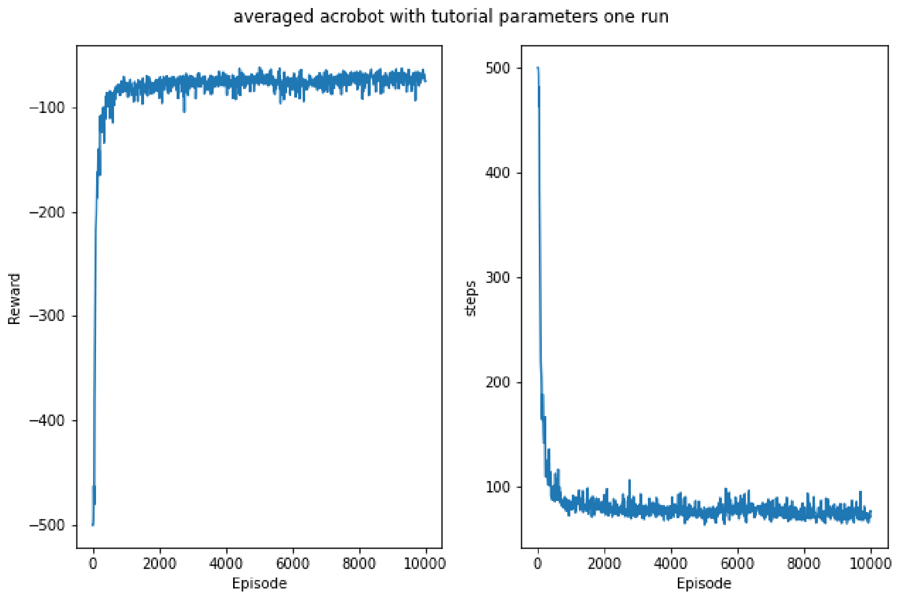

Figure 1: Averaged acrobot with the tutorial parameters and a single run

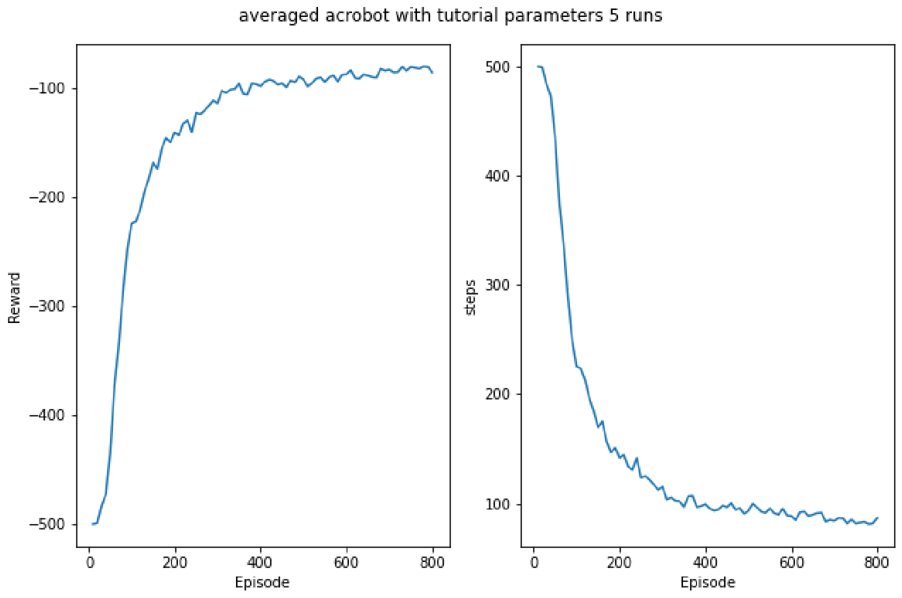

Figure 2: Averaged acrobot with the tutorial parameters and an average over five runs

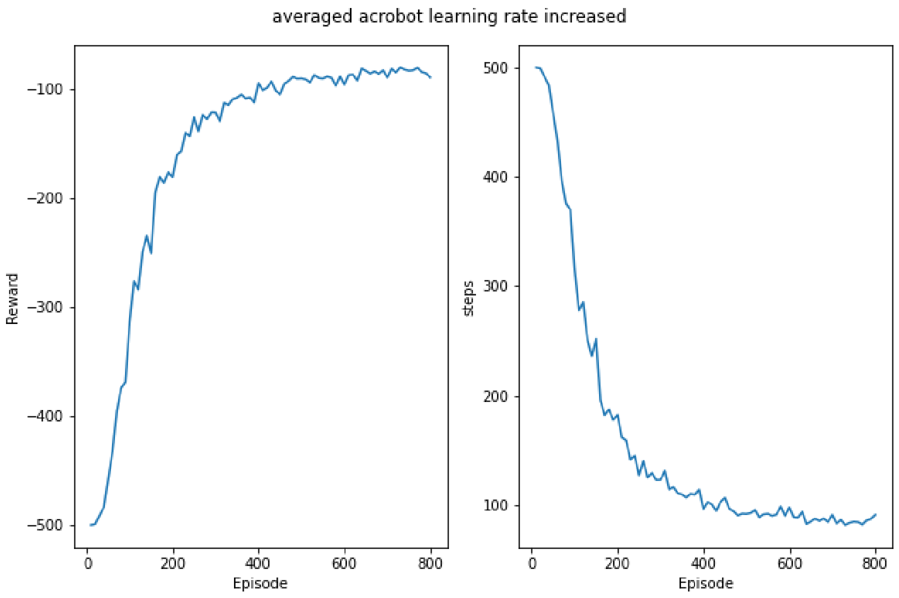

Figure 3: Averaged acrobot. Learning rate set to 0.0009

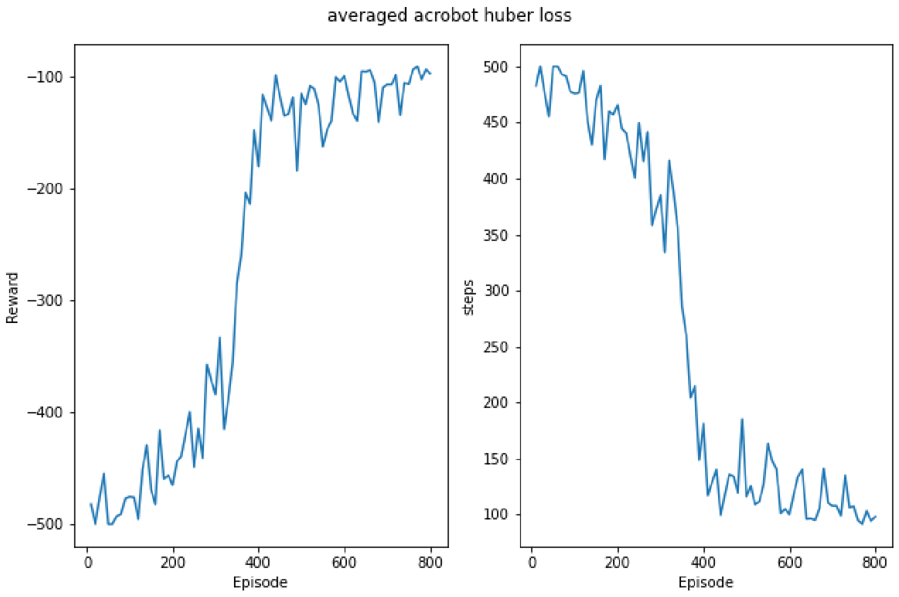

Figure 4: Averaged acrobot. Huber loss

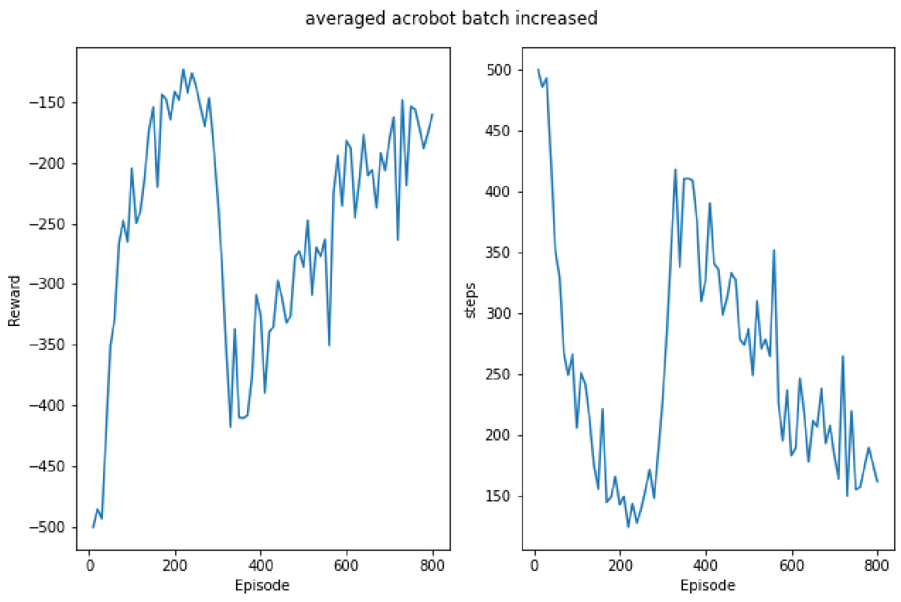

Figure 5: Averaged acrobot. Batch size set to 128

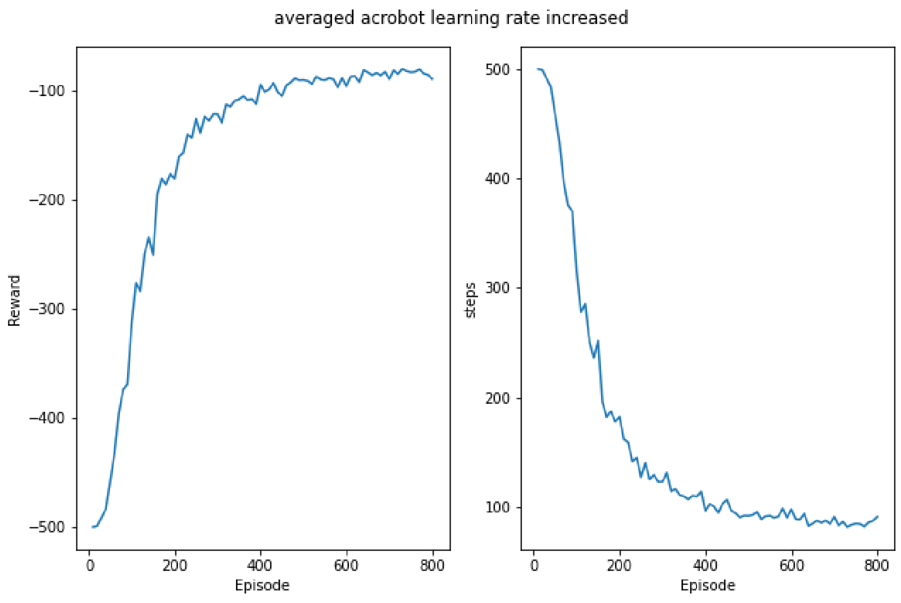

Figure 6: Averaged acrobot. Update frequency set to 12

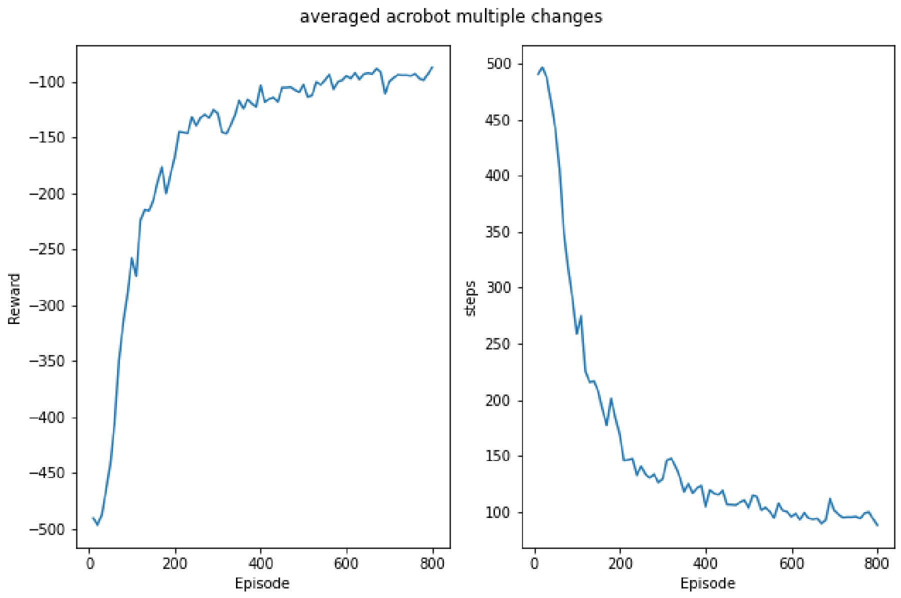

Figure 7: Averaged acrobot. Batch size and learning rate increased, update frequency and hidden layer decreased

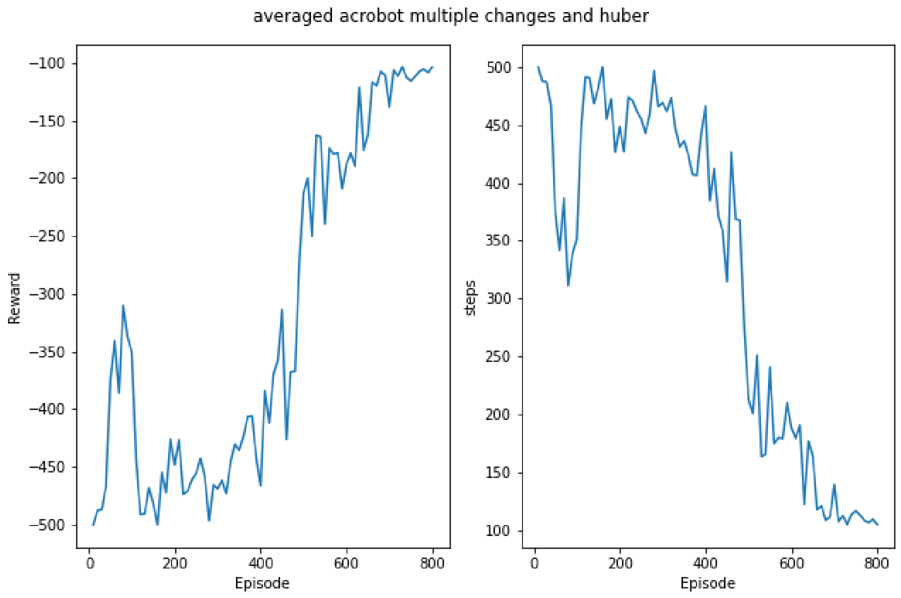

Figure 8: Averaged acrobot. Batch size and learning rate increased, update frequency and hidden layer decreased, huber loss

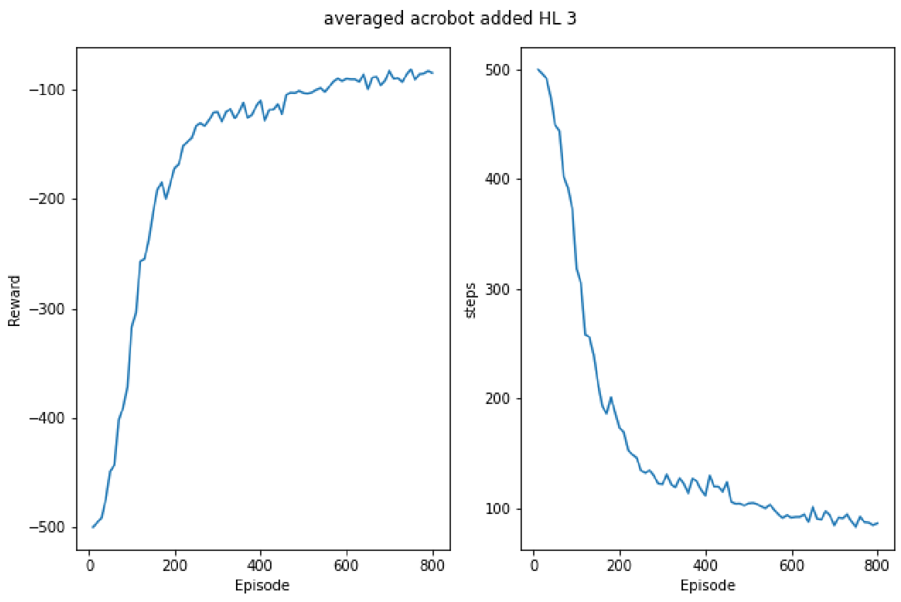

Figure 9: Averaged acrobot. Added a third hidden layer

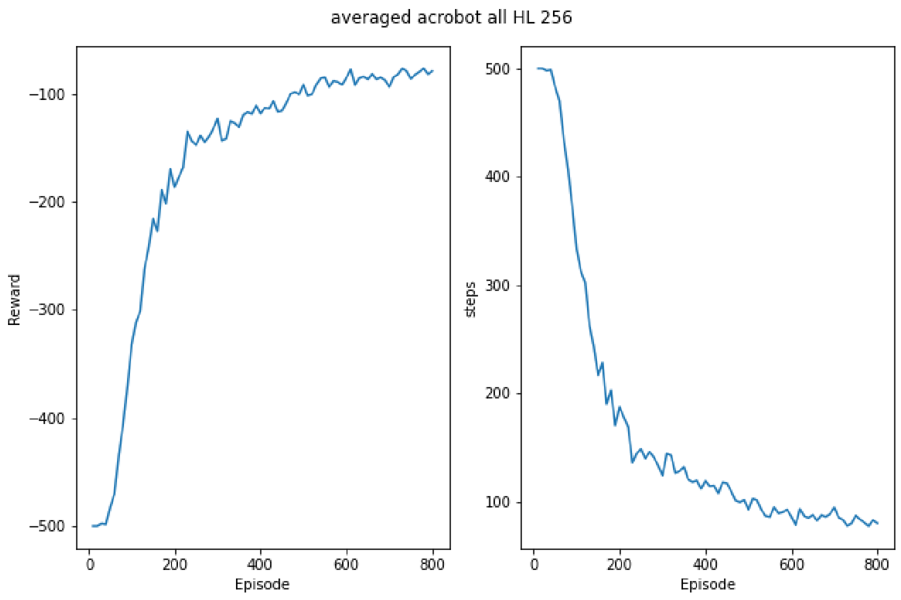

Figure 10: Averaged acrobot. All three hidden layers with size 256

Parameter	**1**	**2**	**3**	**4**	**5**	**6**
Replay buffer size	10000	10000	10000	10000	10000	10000
Batch size	64	64	64	128	128	256
Learning rate	0.0005	0.0009	0.0009	0.0005	0.0005	0.005
Update frequency	20	20	20	20	12	10
Hidden Layer 1	128	128	128	128	128	64
Hidden Layer 2	64	64	64	64	64	64
Hidden Layer 3	-	-	-	-	-	-
Loss function	MSE	MSE	Huber	MSE	MSE	MSE
Figure	2	3	4	5	6	7

Table 2: List of hyperparameters for the acrobot

Parameter	**7**	**8**	**9**
Replay buffer size	10000	10000	10000
Batch size	256	64	64
Learning rate	0.005	0.0005	0.0005
Update frequency	10	20	20
Hidden Layer 1	64	64	256
Hidden Layer 2	64	64	256
Hidden Layer 3	-	64	256
Loss function	Huber	MSE	MSE
Figure	8	9	10

Table 3: List of hyperparameters for the acrobot continued

with the tried combinations. Even more complex networks did not lead to an increased performance in faster convergence to an higher average reward than -75. With all parameter combinations tried, the performance of huber loss was less than using MSE as a loss function.

2.3 CartPole-v1

The plot of a single run with the hyperparameters from the tutorial is shown in figure 11. An average score of 195 over the last 100 episodes was used as a stopping criterion, which was reached after 231 episodes for one single training run. Hence, newly trained agents should reach the same reward in less episodes. New agents should not need more than 1000 episodes to reach the reward, fixing this number to avoid long computation times and adding the ability to average easily. Furthermore, 1000 was chosen as the previously mentioned run was just a single run and might not be representative. The steps and average reward values are actually the same for this environment, as a reward of 1 is returned for each step where the pole was in the air.

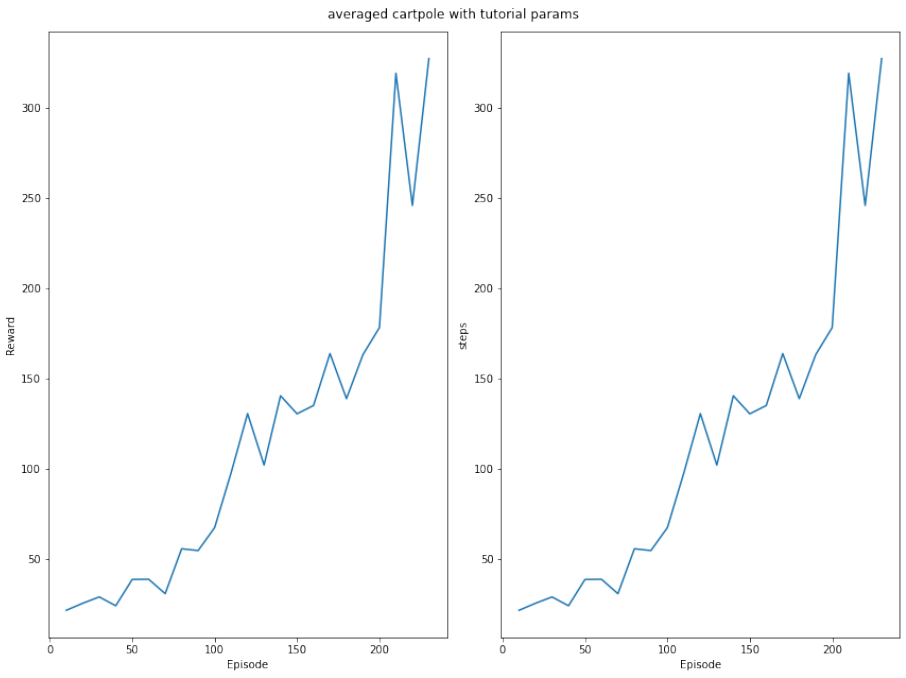

Figure 11: averaged cartpole with the tutorial parameters and a single run

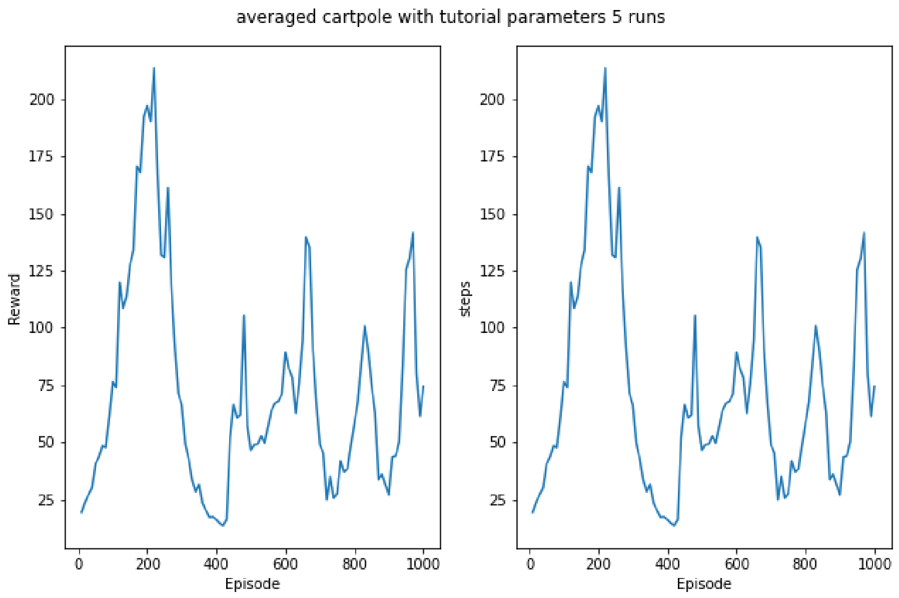

Figure 12: averaged cartpole with the tutorial parameters averaged over 5 runs

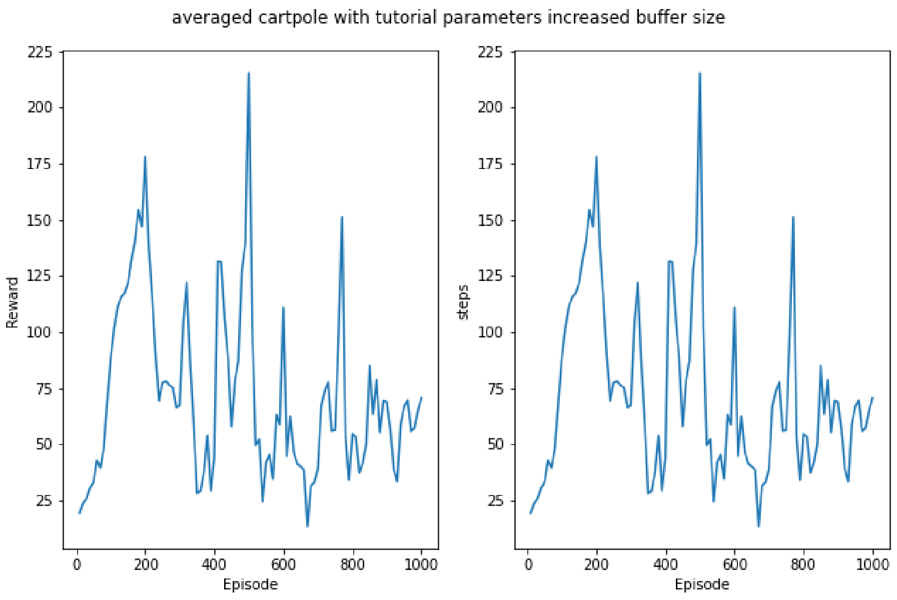

Figure 13: averaged cartpole with the tutorial parameters. replay buffer set to 100000.

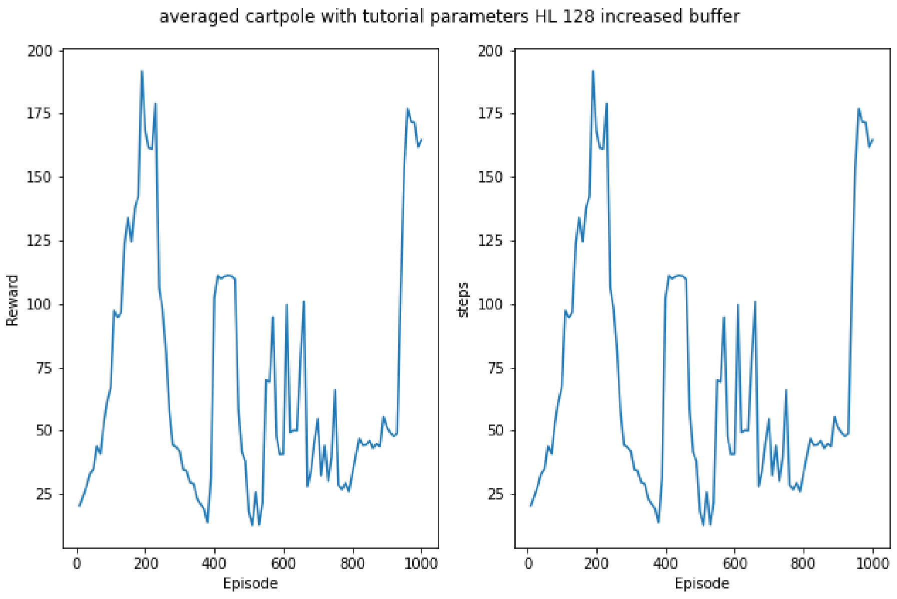

Figure 14: averaged cartpole with the tutorial parameters. replay buffer set to 100000. Hidden layer all 128

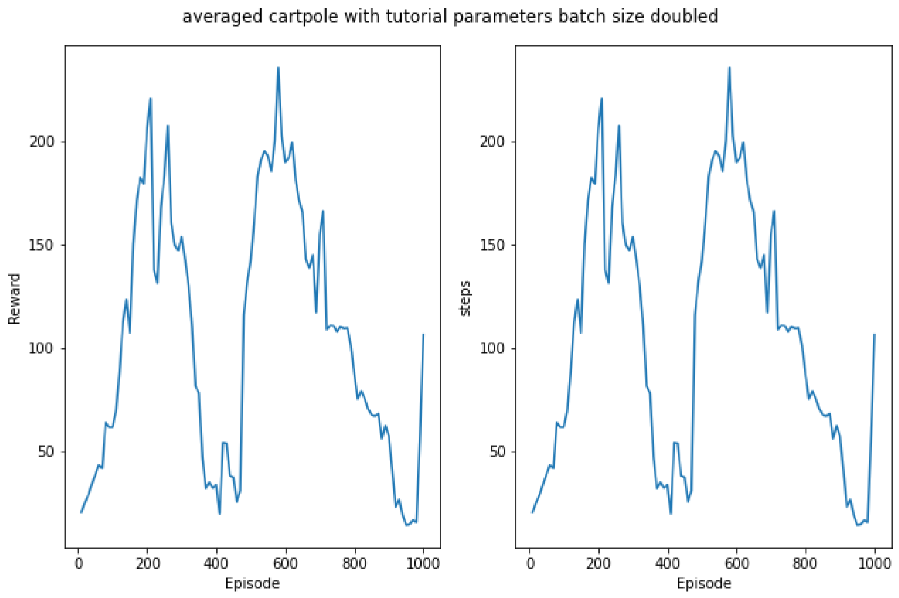

Figure 15: averaged cartpole with the tutorial parameters. batch size set to 128.

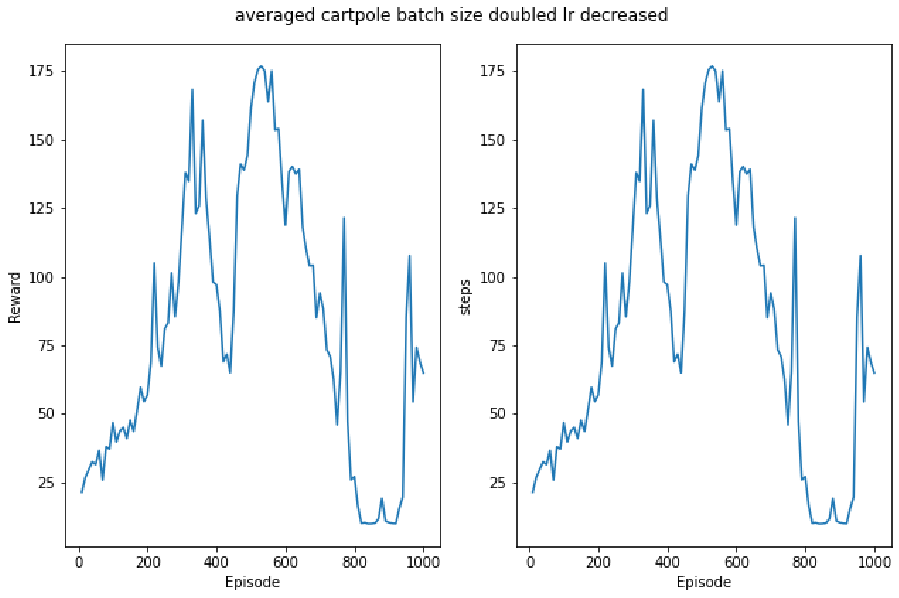

Figure 16: averaged cartpole. batch size set to 128, lr to 0.005.

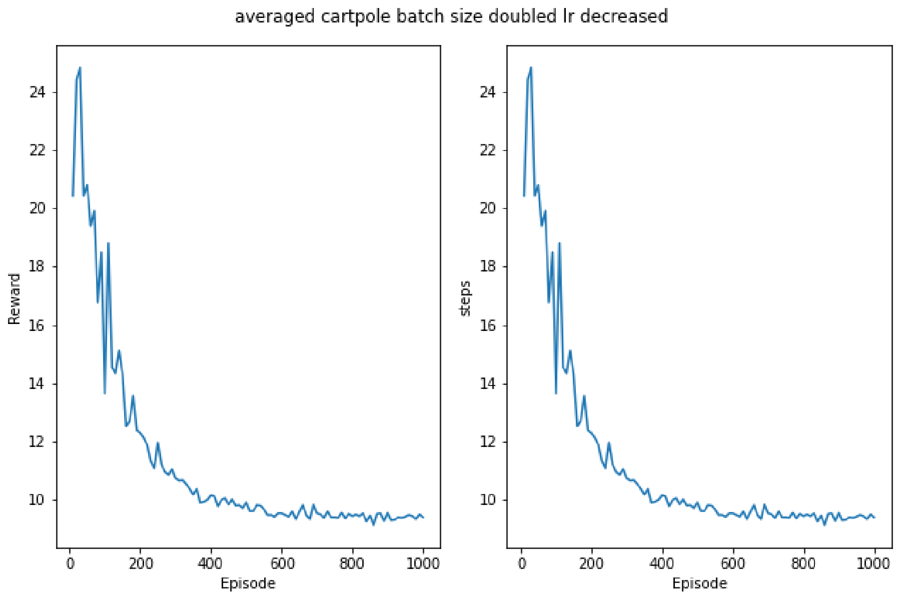

Figure 17: averaged cartpole. batch size set to 128, lr to 0.00005.

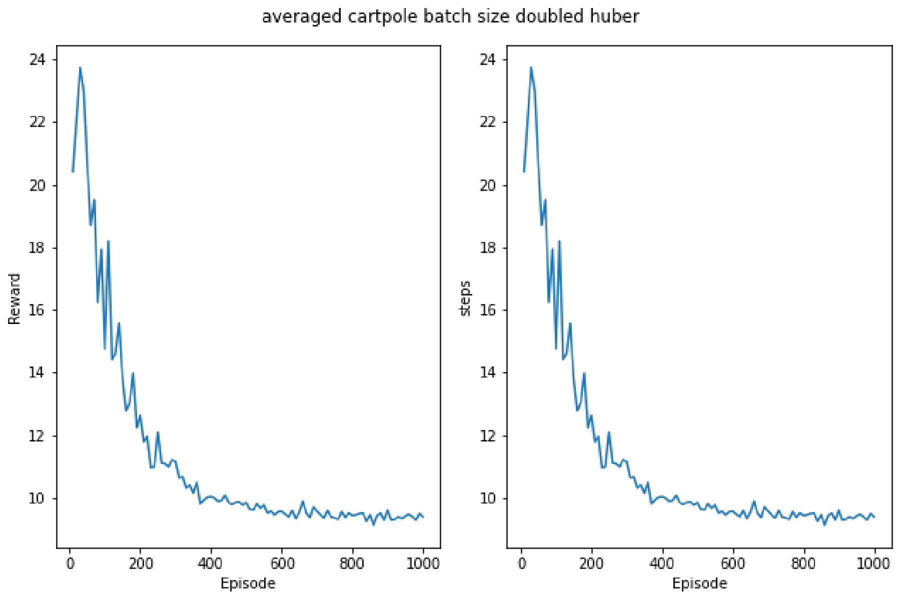

Figure 18: averaged cartpole. batch size set to 128, huber loss.

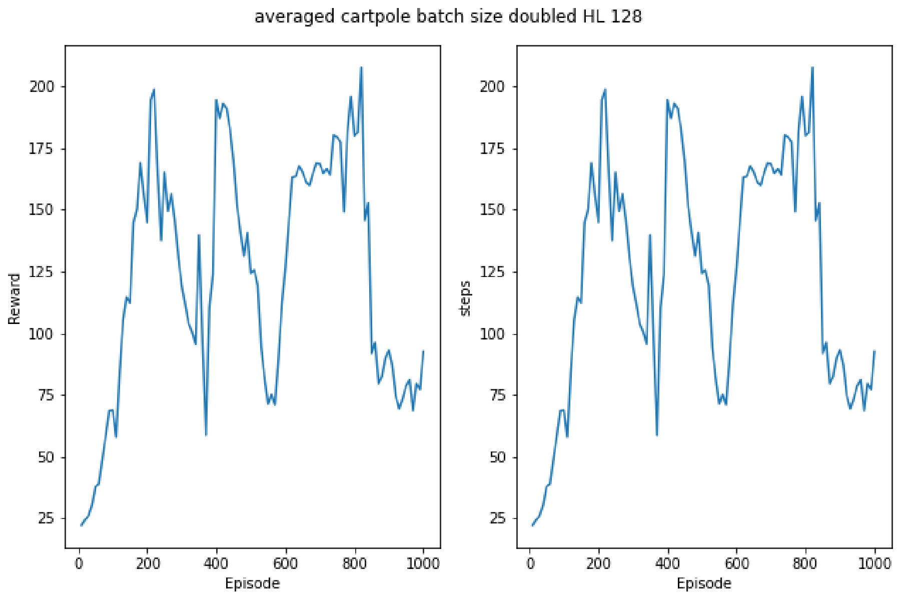

Figure 19: averaged cartpole. batch size set to 128, HL2 128.

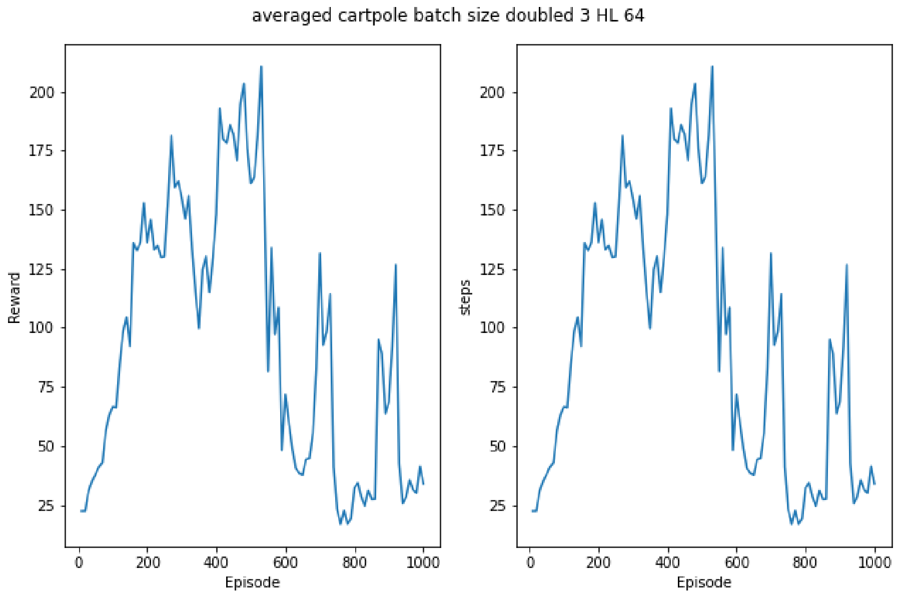

Figure 20: averaged cartpole. batch size set to 128, 3 HL 64.

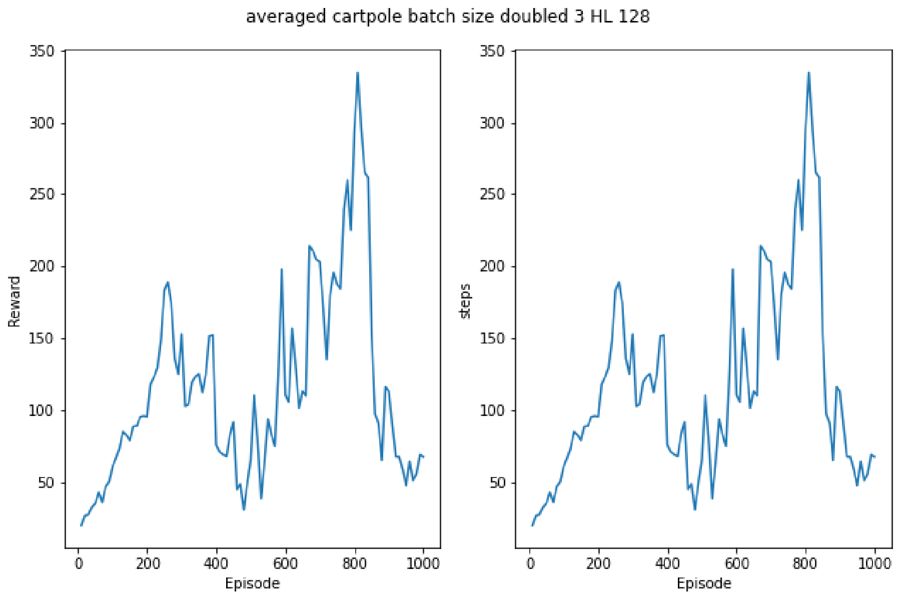

Figure 21: averaged cartpole. batch size set to 128, 3 HL 128.

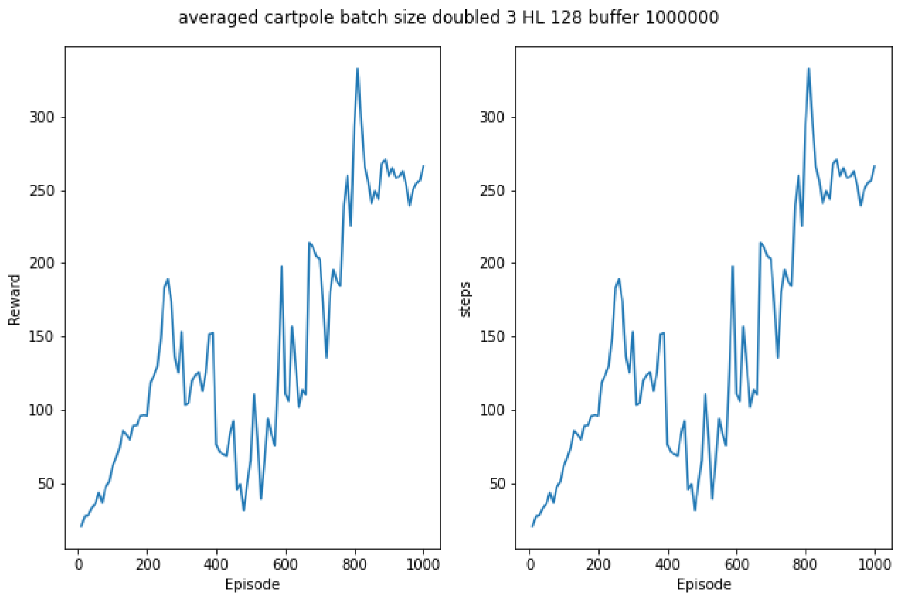

Figure 22: averaged cartpole. batch size set to 128, 3 HL 128, buffer 100000.

Parameter	**1**	**2**	**3**	**4**	**5**	**6**
Replay buffer size	10000	10000	10000	10000	10000	10000
Batch size	64	64	64	128	128	128
Learning rate	0.0005	0.0005	0.0005	0.0005	0.005	0.00005
Update frequency	20	20	20	20	20	20
Hidden Layer 1	128	128	128	128	128	128
Hidden Layer 2	64	64	128	128	128	128
Hidden Layer 3	-	-	-	-	-	-
Loss function	MSE	MSE	MSE	MSE	MSE	MSE
Figure	12	13	14	15	16	17

Table 4: List of hyperparameters for the cartpole

Parameter	**7**	**8**	**9**	**10**	**11**	**12**
Replay buffer size	10000	10000	10000	10000	100000	10000
Batch size	128	128	128	128	128	128
Learning rate	0.0005	0.0005	0.0005	0.0005	0.0005	
Update frequency	20	20	20	20	20	15
Hidden Layer 1	128	128	64	128	128	128
Hidden Layer 2	64	128	64	128	128	128
Hidden Layer 3	-	-	64	128	128	128
Loss function	huber	MSE	MSE	MSE	MSE	MSE
Figure	18	19	20	21	22	23

Table 5: List of hyperparameters for the cartpole continued

Inferences: Overall, the hyperparameters given in the tutorial worked quite well. An increase in performance could be reached by adding more complexity to the network architecture and increasing the batch size, however, adding to the computation time as well. Huber loss and increasing or decreasing the learning rate did not lead to better results. A decrease in performance after a first peak in the average reward could not be avoided. This could either be due to catastrophic forgetting or overfitting, thus not generalizing well. Catastrophic forgetting can be avoided by increasing the buffer size. This did not have an effect here though. Therefore, training should be stopped when reaching a certain reward, using the model afterwards as it is. Furthermore, the changes in hyperparameters were rather coarse. A finetuned hyperparameter search might lead to better performing models.

2.4 Mountaincar-v1

The first run for 10000 episodes showed some peaks but somehow the agent "unlearnt" the way to solve the environment (figure 24). However, it showed that the agent is indeed able to reach the flag and by that exiting the environ-

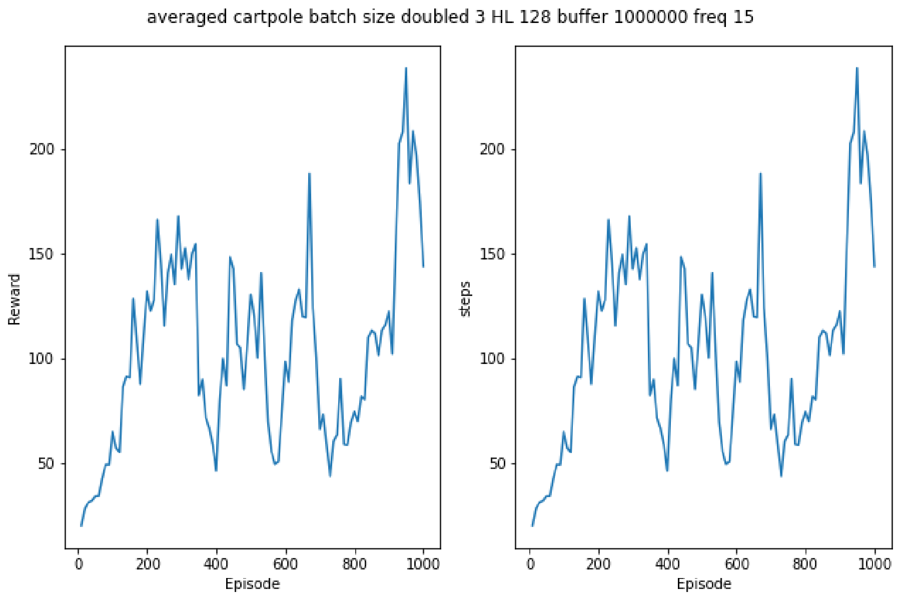

Figure 23: averaged cartpole. batch size set to 128, 3 HL 128, buffer increased, update freq 15.

ment. The number of steps was close to 200, which is the maximum of steps as well. The termination criterion of the environment was relaxed and set 500 as the probability of taking the exact steps under an exploratory policy to exit the environment is low. The following changes in parameters are shown in table 6. Note: Run 3 and run 4 was stopped early.

Parameter	**1**	**2**	**3**	**4**	**5**
Replay buffer size	10000	10000	100000	100000	100000
Batch size	64	64	128	64	64
Learning rate	0.0005	0.0005	0.005	0.0005	0.0005
Update frequency	20	20	20	40	20
Hidden Layer 1	128	128	128	128	128
Hidden Layer 2	64	64	128	128	128
Hidden Layer 3	-	-	-	-	128
Loss function	MSE	MSE	MSE	MSE	MSE
Termination steps	200	500	500	500	500
Figure	24	25	26	27	28

Table 6: List of hyperparameters for the mountaincar

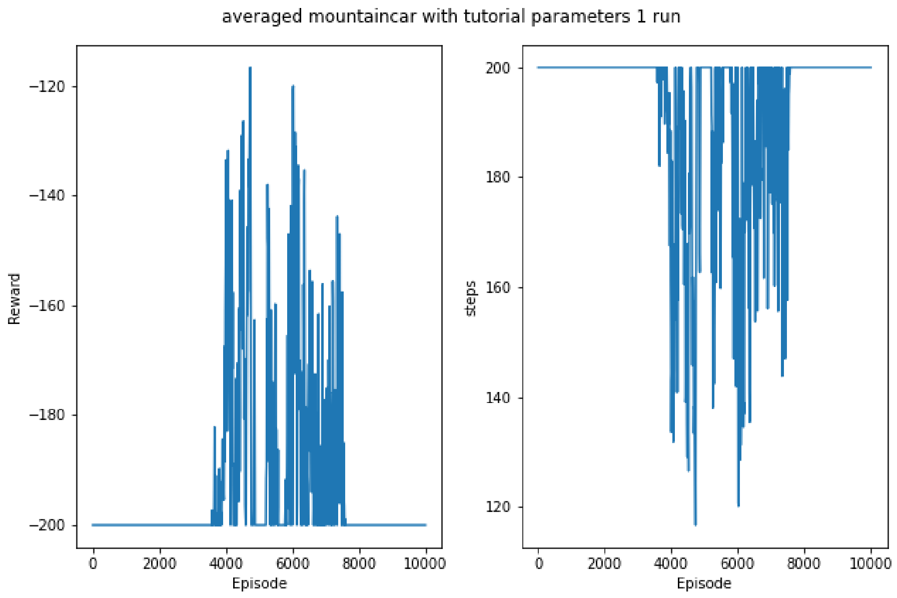

Figure 24: averaged mountaincar with parameters as in tutorial. one run.

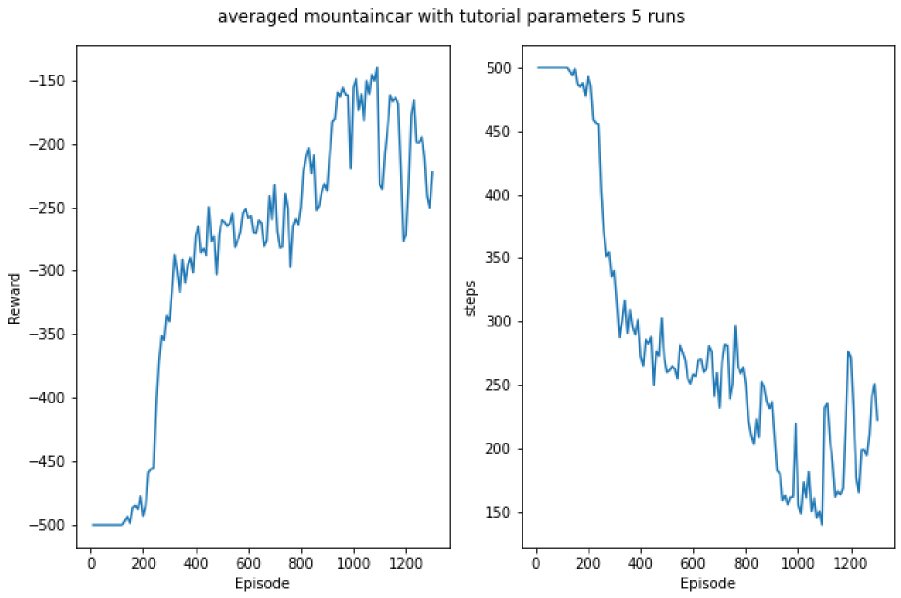

Figure 25: averaged mountaincar with parameters as in tutorial. five runs.

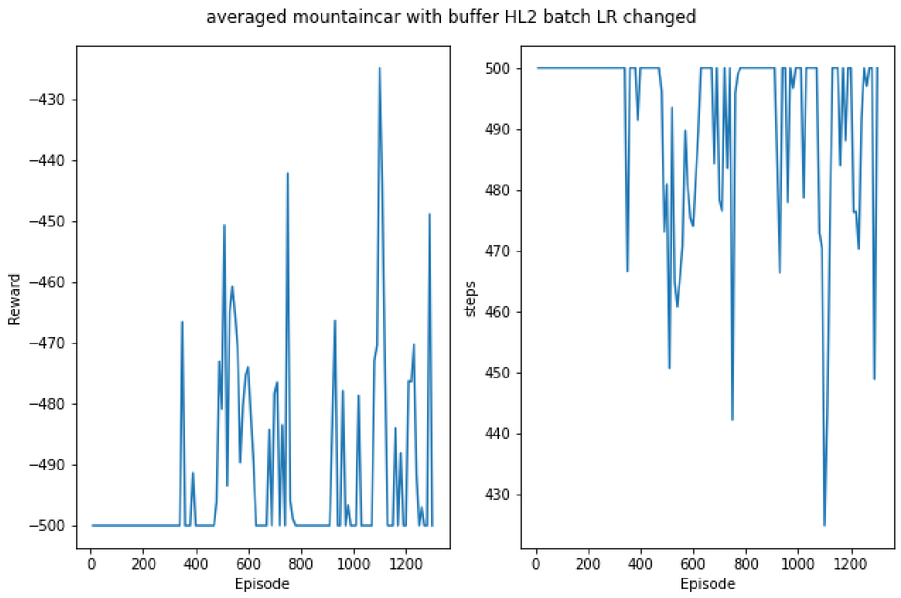

Figure 26: averaged mountaincar with buffer, batch size and hidden layer 2 increased, learning rate decreased. Stopped early.

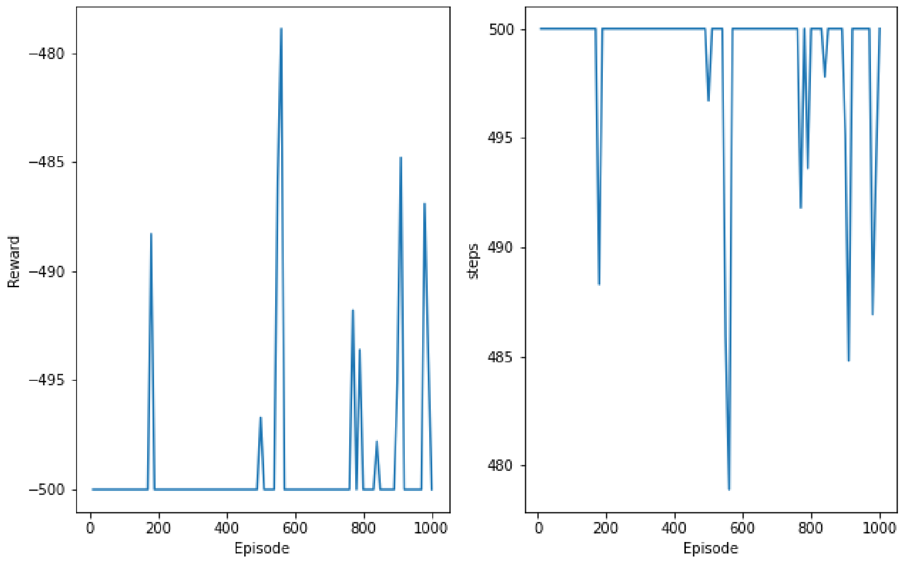

Figure 27: averaged mountaincar with a higher update frequency. Stopped early.

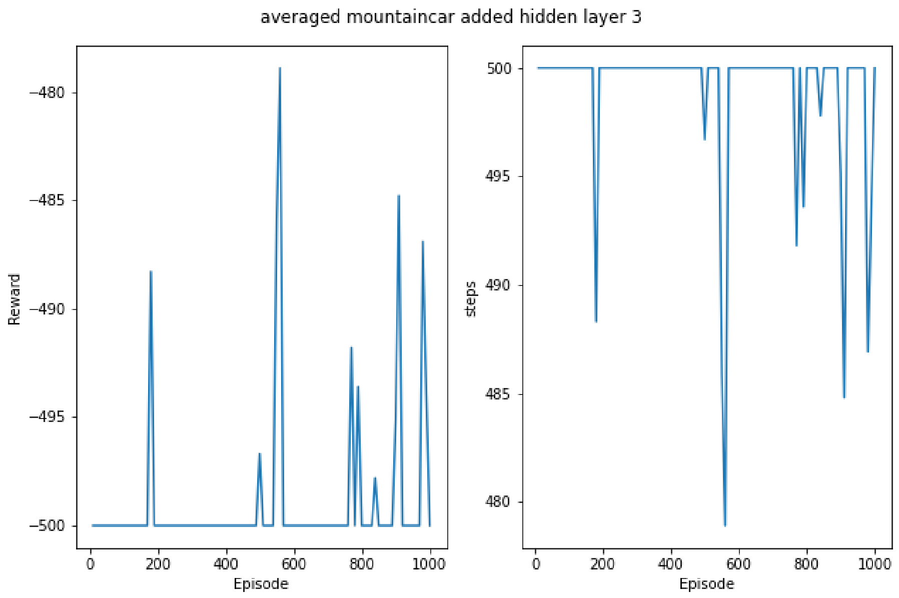

Figure 28: averaged mountaincar with a third hidden layer.

Inferences: The given hyperparameter of the tutorial did not work well on the environment as it is. However, this is not due to bad hyperparameters but rather due to a small margin between necessary steps needed to exit the environment and the termination of an episode. After setting the episode length to 500 in the environment, the learning took place from early episodes on and once again the hyperparameters from the tutorial worked quite well. More complex network structures did not increase the performance, however the computation time was significantly higher. An increase in learning speed by increasing the learning rate did not lead to better results. In fact, they were worse. The computation time took 3 hours with the hyperparameters from the tutorial, while the more complex network took even longer.

3 Actor-Critic

3.1 HyperParameter Tuning

1. **Learning Rate:** Learning rate governs the step size when updating gradients. A high value may oscillate and fail to converge; on the other hand, a really low value can take a lot of time to converge. Also, the choice of learning rate depends on model complexity since a network with more layers will have higher complexity and require a lower learning rate. For our task, we have found that decreasing the learning rate leads to a better policy based on the average reward received.

2. **Network Architecture:** This consists of the number of hidden layers and the number of neurons in hidden layers. These are dependent on the complexity of the environment, which include state space and action space and the approximate order of equations governing the task. We can increase the number of layers and neurons in the actor and critic networks for complex environments to learn better representations of complex equations and better generalization. But larger networks considerably increase the execution time and require more episodes to converge. For our task, we have found that increasing model complexity leads to a policy with better returns based on reward curves.

3. **Number of Episodes:** This parameter is dependent on both the learning rate and network architecture, as with a more complex architecture or low learning rate, we will require more episodes to converge. Moreover, if we were to use larger values of n in n-step return variation, that also leads to increased requirement of episodes to converge.

3.2 Code Structure

We follow the forward view, and as such, we update the parameters after episode termination. The function below is defined to get the cumulative return in full return case defined as: $G_t = \sum_{t'=t}^{T} \gamma^{t'-t} * R_{t'+1}$

```
def get_cumulative_discounted_rewards(rewards, done_list, gamma):
"""
This calculates basically the full reinforce return
:param rewards:
:param done_list:
:param gamma:
:return expected return G_t:
"""
cumulative_reward, _r = [], 0
for i in np.arange(len(rewards))[::-1]:
    _r = done_list[i] * _r * gamma + rewards[i]
    cumulative_reward.insert(0, _r)
return torch.stack(cumulative_reward).to(device)
```

Next we define the function to get n-step returns defined as:
$G_t = \sum_{t'=t}^{n+t-1} \gamma^{t'-t} * R_{t'+1}$

```
def compute_n_step_returns(rewards, values, gamma,
    n_step, actions=None, done=None):
'''
Should return n_step target return
:param rewards:
:param values:
:param gamma:
:param n_step:
:return n-step return:
'''
values.to(device)
T = len(rewards)
returns = []
for i in np.arange(len(rewards)):
    returns.append(torch.sum(
        torch.stack(rewards[i: np.min((i + n_step, T)
            )]) * torch.FloatTensor(
            np.power(gamma, np.arange(len(rewards[i:
                np.min((i + n_step, T))])))))[:, None])
                )
returns = torch.stack(returns)[:, None].to('cuda') #
    Get Rewards for n-window

# If n=1 we just add next reward
if n_step == 1:
    x = torch.FloatTensor(np.power(gamma, np.ones(T))
        )[:, None].to(device)
    returns += x * values[1:]
# Otherwise iteratively multiply by gamma and update
    the expected return
else:

    x = torch.FloatTensor(np.power(gamma, np.ones(T -
         (n_step - 1)) * n_step))[:,None].to(device)
    returns[:T - (n_step - 1)] += x * values[n_step:]

    returns[T - (n_step - 1):] += torch.FloatTensor(
        np.power(gamma, np.ones(n_step - 1) * n_step))
        [:,None].to(device) * values[T]

return returns
```

3.3 CartPole-v1

3.3.1 One-Step-Return

Averaged over 10 runs with different seed values.

Parameter	**value**
Number of hidden layers and size	2(1024, 512)
Learning Rate	1e-4
Episodes	500

Table 7: List of hyperparameters

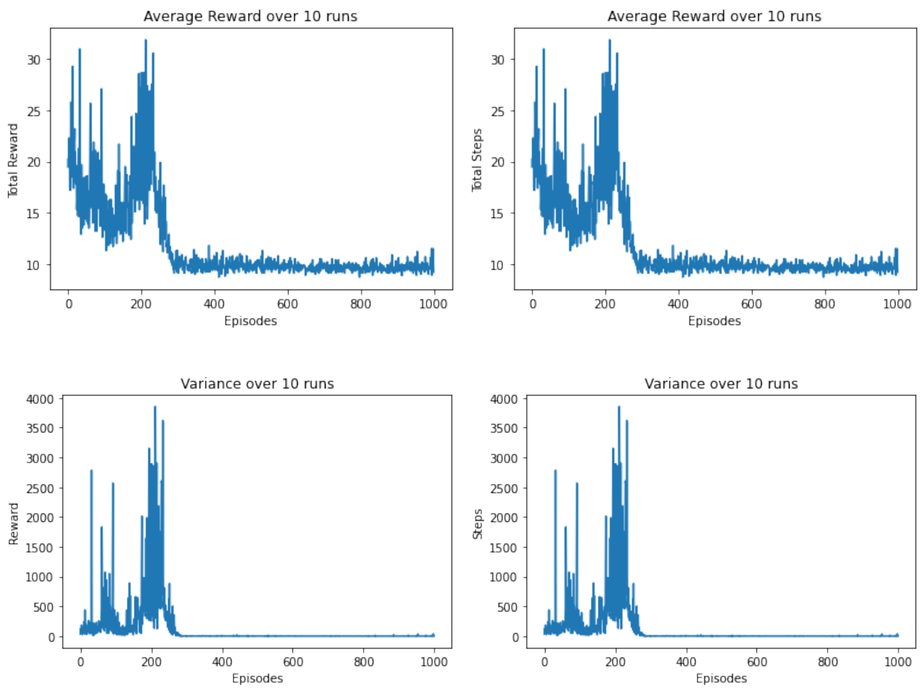

Figure 29: CartPole with Tutorial Parameters

Parameter	**value**
Number of hidden layers and size	2(1024, 512)
Learning Rate	1e-5
Episodes	1000

Table 8: List of hyperparameters

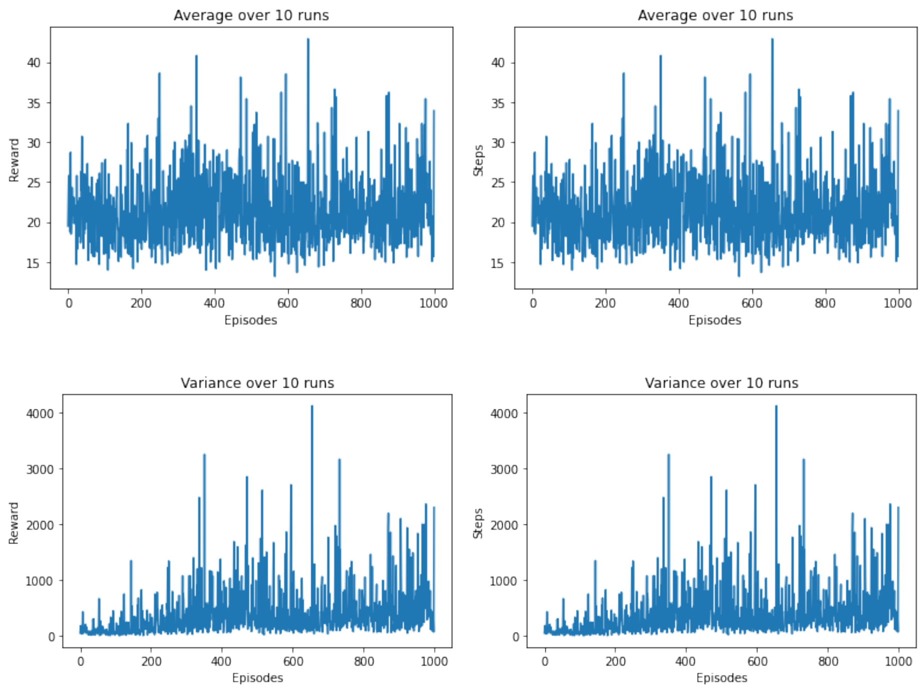

Figure 30: CartPole with Above Parameters

Note , the slight improvement in performance, as the average reward has increased. While in the previous run it was oscillating around 10.

Parameter	**value**
Number of hidden layers and size	3(512, 256, 128)
Learning Rate	1e-5
Episodes	1000

Table 9: List of hyperparameters

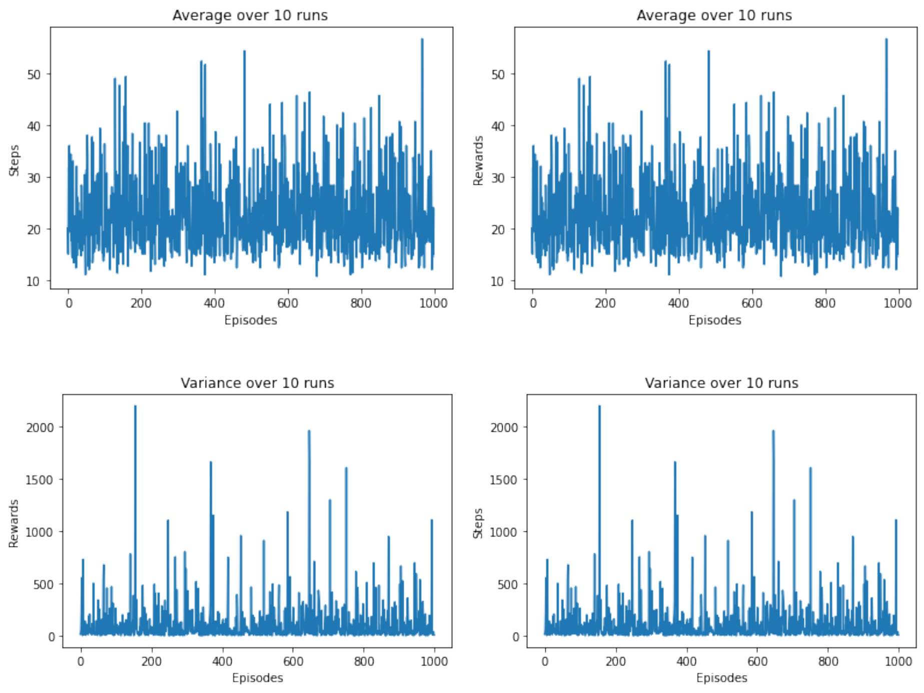

Figure 31: CartPole with Above Parameters

Again, we can see the slight increase in the average reward received per episode.
We note that graphs for rewards and steps are identical. This is because in CartPole env, we get a reward of +1 for every time step we remain upright, so the number of time steps = rewards.
Since instead of stopping training when the average reward of 195 is reached, we continue training for a max number of episodes(1000), it helps us see that for tutorial parameters, the agent learns and peaks for some episode values between 300-400 episodes, after reaching this threshold, it stagnates at a low value of around 10 steps per episode for the rest of the run.

3.3.2 N-Step Returns and Full Return

Averaged over 3 runs with different seed values. For this part, we use a running average of 10 episodes to better visualise the trends.

Parameter	value
Number of hidden layers and size	2(1024, 512)
Learning Rate	1e-4
Episodes	1000
N-step-values	2, 4, 8, 32, 64, Full-Return

Table 10: List of hyperparameters

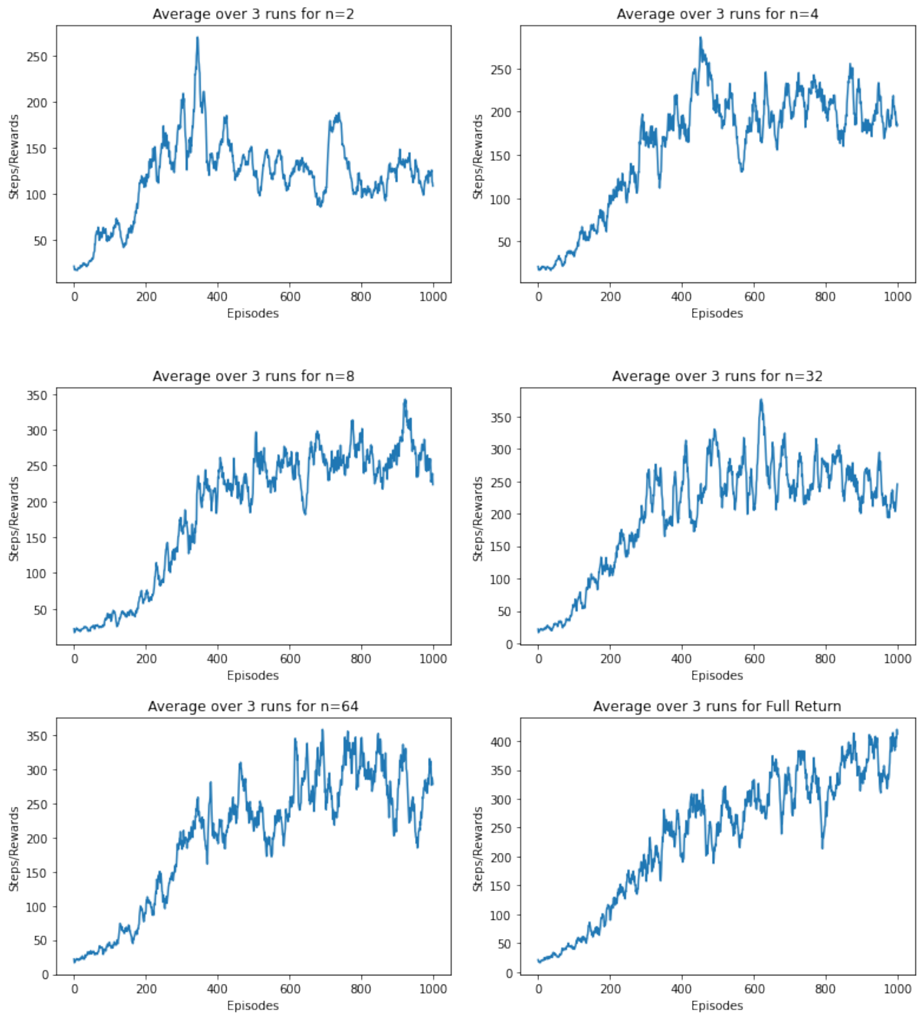

Figure 32: Average Rewards with Above Parameters for different n-values

Notice the gradual increase in the reward for higher values of n, suggesting a more stable policy than for lower values of n, where the average reward keeps peaking and tripping.

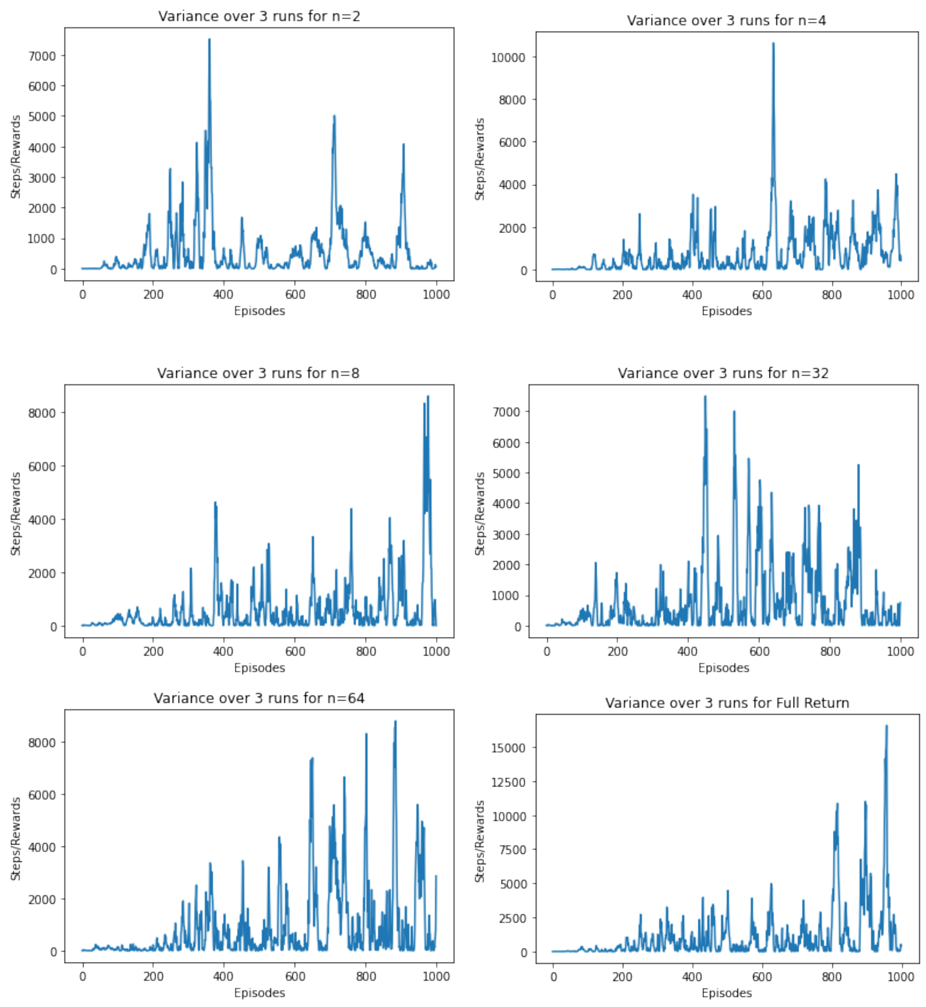

Figure 33: Variance in Rewards with Above Parameters for different n-values

3.4 Acrobot-v1

3.4.1 One-Step-Return

Averaged over 5 runs for 1000 episodes with different seed values. We plot a running average over 10 episodes.

Parameter	**value**
Number of hidden layers and size	2(1024, 512)
Learning Rate	1e-4
Episodes	1000

Table 11: List of hyperparameters

The model didn't seem to converge at all for the values from the tutorial, with only occasional peaks and hence the graphs for this run are omitted. The reward and steps were almost constant at -500 and 500, respectively.

Parameter	**value**
Number of hidden layers and size	2(1024, 512)
Learning Rate	1e-5
Episodes	500

Table 12: List of hyperparameters

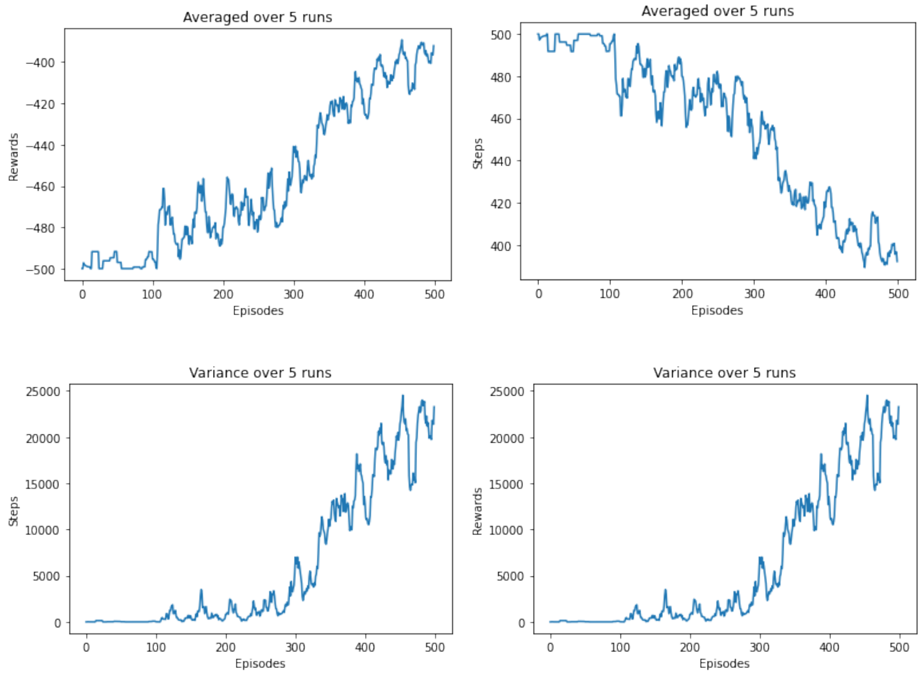

Figure 34: Acrobot with Above Parameters

As seen in CartPole, decreasing the learning rate did improve the model rewards and steps to some extent. As for the network architecture, unlike

CartPole, the same architecture didn't seem to work here, but we believe a larger and better architecture can be found such that it improves the agent's performance.

3.4.2 N-Step Returns and Full Return

Averaged over 5 runs with different seed values. For this part, we use a running average of 10 episodes to better visualise the trends.

Parameter	**value**
Number of hidden layers and size	2(1024, 512)
Learning Rate	1e-4
Episodes	Variable($\propto n$)
N-step-values	4, 8, 32, Full-Return

Table 13: List of hyperparameters

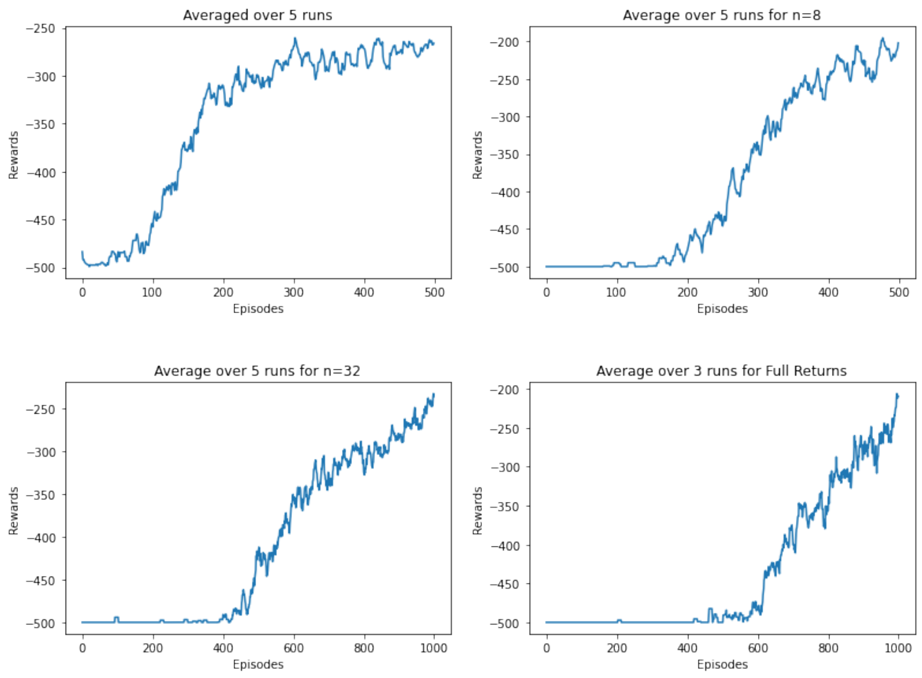

Figure 35: Average Rewards with Above Parameters for different n-values

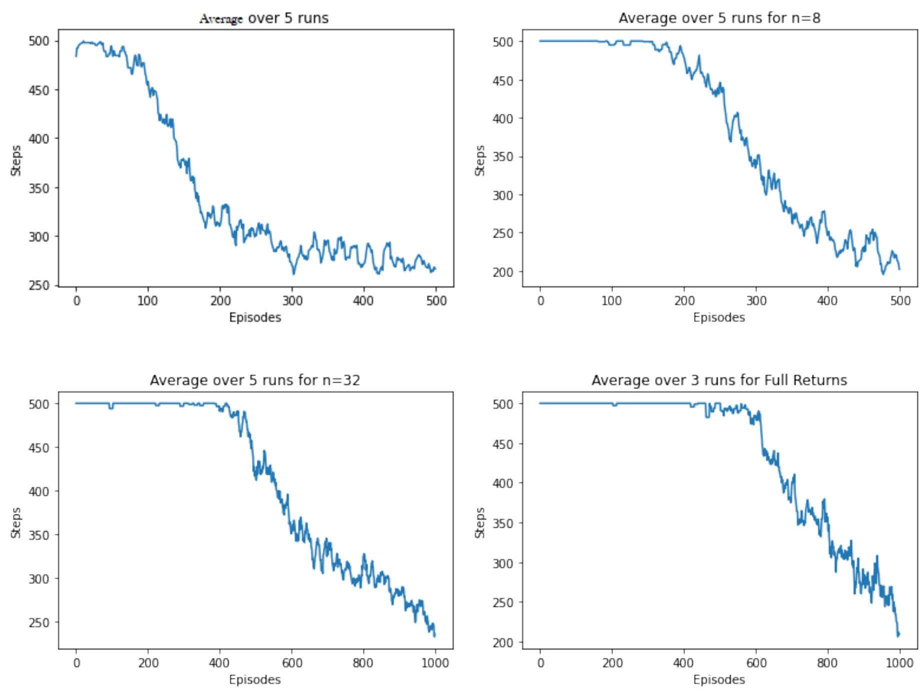

Figure 35: Average Steps with Above Parameters for different n-values

Observe that for larger values of n, the agent takes more time to take good actions. We can attribute this to increase in exploration behaviour.

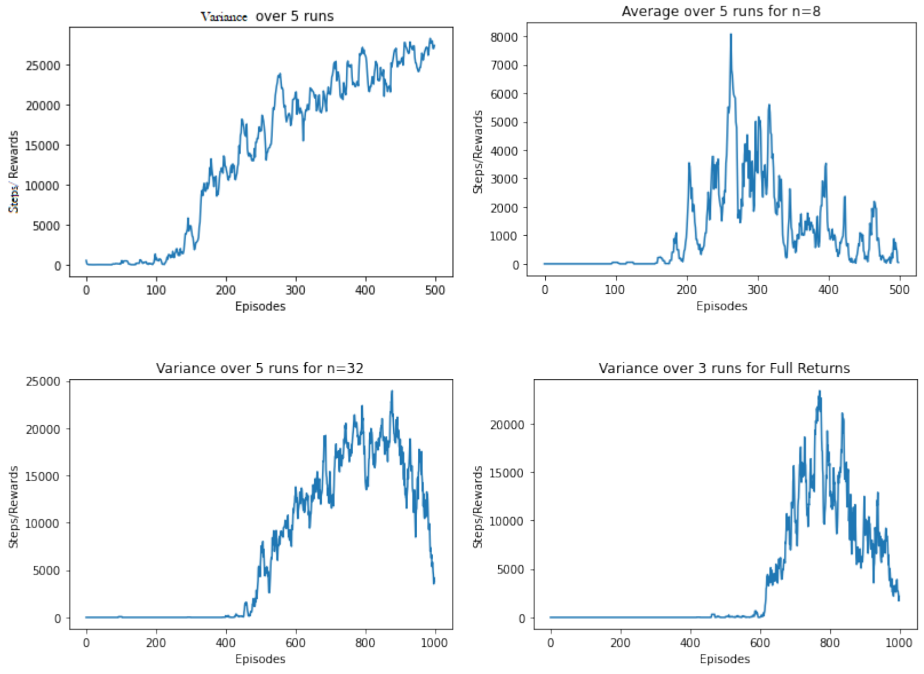

Figure 35: Variance with Above Parameters for different n-values

Notice that the variance for models with higher n decreases over time, suggesting that it is converging to an optimal policy.

3.5 MountainCar-v0

3.5.1 One-Step-Return

Averaged over 5 runs for 1000 episodes with different seed values. We plot a running average over 10 episodes.

Parameter	**value**
Number of hidden layers and size	2(1024, 512)
Learning Rate	1e-4
Episodes	1000

Table 14: List of hyperparameters

The model didn't seem to converge at all for the values from the tutorial, with only occasional peaks and hence the graphs for this run are omitted. We increased the time step limit for the environment from 200 to 500 in hopes that the agent would have more opportunity to learn, but the reward and steps remained almost constant at -500 and 500, respectively.

Parameter	**value**
Number of hidden layers and size	2(1024, 512)
Learning Rate	1e-5
Episodes	1000

Table 15: List of hyperparameters

This experiment also failed to improve the agent.

3.5.2 N-Step Returns and Full Return

Parameter	**value**
Number of hidden layers and size	2(1024, 512)
Learning Rate	1e-4
Episodes	Variable($\propto n$)
N-step-values	4, 8, 32, Full-Return

Table 16: List of hyperparameters

After running all of the described models above, in the Mountain Car environment, our implementation could not solve the Mountain Car problem. For reasons we can only speculate about, the agent did not manage to explore enough to reach the top a single time. We assume that because policy gradient methods are on-policy and the exploration is determined by the distribution over actions, the evenly distributed reward of -1 is not enough to enforce exploration.

3.6 Inferences and Observations:

1. We observe that on decreasing model complexity(eg: two layers of size 64), we hardly see any improvement and fail to learn anything.
2. As expected, increasing model complexity and/or decreasing learning rate leads to slower convergence but better rewards on average over the episodes. Also, we see a gradual trend of improvement in these models compared to the baseline model.
3. We can conclude that increasing model complexity along with a decrease in learning rate may lead to a better policy with less noise if run for a sufficient number of episodes.
4. With an increase in Model complexity, we see a decrease in variance which is to be expected from the bias-variance trade-off in neural networks since a more complex model is better suited to learn all the possible state-action pairs and environment dynamics.
5. With increasing n-value, we observe that there is an increase in variance as well as an improvement in performance based on average reward curves and variance curves above.

6. Full-return/Monte Carlo variation is the same as REINFORCE with baseline, our baseline being the state values and exhibits the highest variance. Conversely, there is a higher bias in the one-step return or TD(0) variation and a lower bias in the Monte Carlo variation.
7. These bias-variance trends are directly related to the length of cumulative reward in respective variations. In Monte Carlo or full-return variation, we sum over rewards with a single-sample estimate which introduces variance(due to multiple possible trajectories). On the other hand, in one-step actor-critic variance is controlled by the value function from the critic model, but we have a higher bias since our critic might not be perfect.
8. From the tutorial, we follow a shared network design, where the actor and critic share the same hidden parameters, but on experimentation with split network design(i.e., different networks for actor and critic), we found that it tends to perform much better than shared network design. It requires lesser computation time and resources and tends to converge much faster. Even for smaller networks(two layers with 64 neurons each), it converges pretty quickly. It tends to be more stable with Asynchronous updates as well.
9. n = 8 seems to be the ideal value for n-step returns for CartPole and AcroBot environments as it achieves performance close to full returns while keeping the variance low as well as in terms of compute times. Also, for larger values of n, we tend to require more episodes to see convergence.
10. Also, we find that the models are really sensitive to seed values. That implies how dependent these methods are on good weight initialization. For the same identical agent with different seeds, one agent was able to solve the environment while the other failed horribly. We hypothesise that in the AC models, if the agent behaves sub-optimally after some wrong updates in the policy space, the collected data will also be sub-optimal. In some cases, this can lead to the agent not being able to recover from that.