CS6700 - Reinforcement Learning
Programming Assignment 2

Argha Boksi(CS21D407), Jashaswimalya Acharjee(CS22E005)

March 27, 2023

1 DQN

1.1 Acrobot-v1

- We tried solving the environment using the set of hyperparameters **(Network architecture : [64, 64], replay buffer size :** 10^5**, batch size = 64,** $\gamma = 0.99$**, learning rate = 0.0005, update target = 20, explore end = 0.01, decay = 0.995, truncation limit = 1)** originally given in the tutorial. For performance of the agent, please check the second hyperparameter configuration.
- For every hyperparameter configuration we ran 10 experiments. We averaged the performance of the agent across these 10 runs and presented the results.
- We improved the agent's performance **(considering number of episodes taken to solve the environment as performance metric)** 5 times by changing the hyperparameter configurations appropriately. The results are shown here.

1.1.1 Network: [32,32,32], $\gamma = 0.99$, learning rate = 0.001, replay buffer size = 10^5, batch size = 64, update target = 20, truncation limit = 3, explore end = 0.3, decay = 0.8

- Number of episodes taken(averaged over 10 runs) to solve the environment = 436.
- For this hyperparameter configuration learning is very slow.
- Some of the changes that could improve performance are the following -
  - Changing the neural network architecture
  - Changing the learning rate
  - Modifying the truncation limit

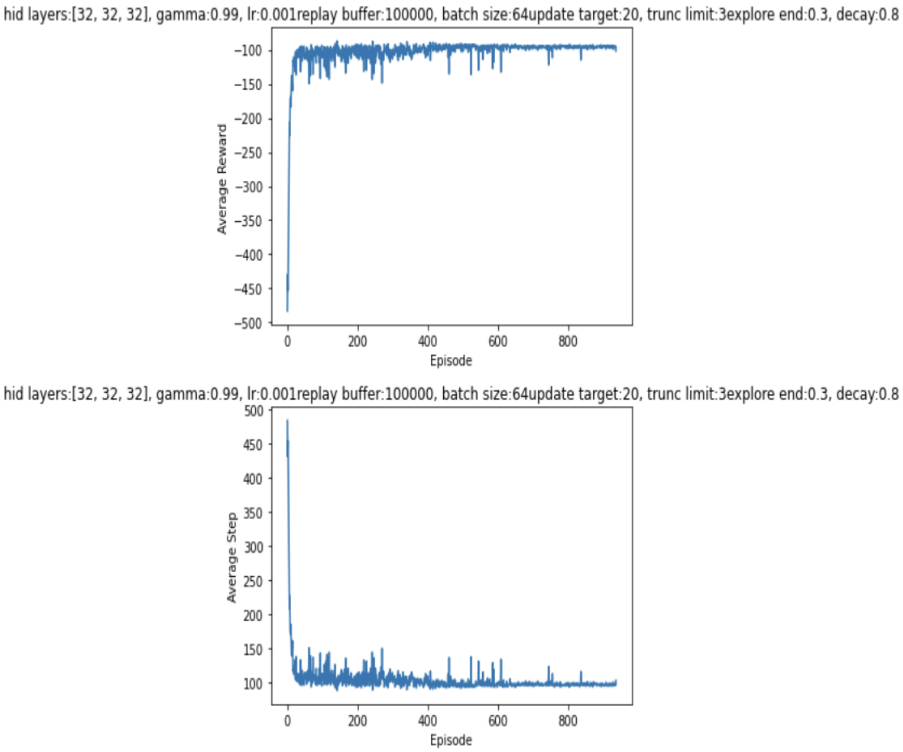


Figure 1: Average Rewards and Average Number of steps

1.1.2 Network: [64, 64], $\gamma = 0.99$, learning rate = 0.0005, replay buffer size = 10^5, batch size = 64, update target = 20, truncation limit = 1, explore end = 0.01, decay = 0.995

- Number of episodes taken(averaged over 10 runs) to solve the environment = 305.9.
- We made the following changes from the previous configuration-
  - Changed the network architecture from [32, 32, 32] to [64, 64]
  - Reduced the learning rate from 0.001 to 0.0005
  - Changed the truncation limit to 1
  - explore end = 0.01, decay = 0.995

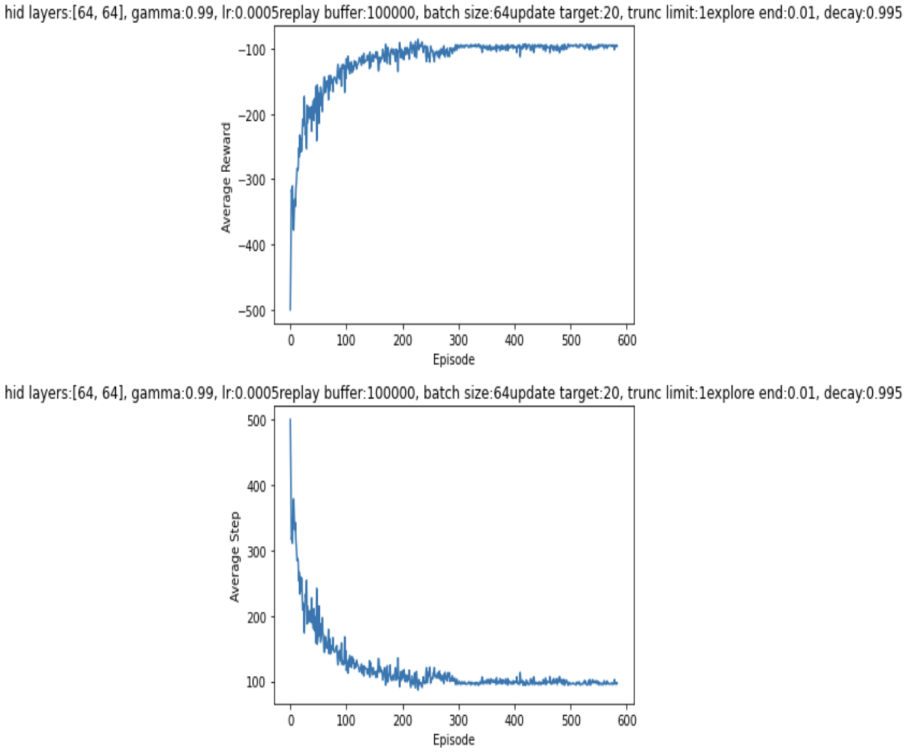


Figure 2: Average Rewards and Average Number of steps

1.1.3 Network: [32, 32], $\gamma = 0.99$, learning rate = 0.001, replay buffer size = 10^5, batch size = 64, update target = 20, truncation limit = 3, explore end = 0.3, decay = 0.8

- Number of episodes taken(averaged over 10 runs) to solve the environment = 264.8.
- We made the following changes from the previous configuration-
  - Changed the network architecture from [64, 64] to [32, 32]
  - Changed the learning rate from 0.0005 to 0.001
  - Changed the truncation limit to 3
  - Explore end = 0.3, Decay = 0.8

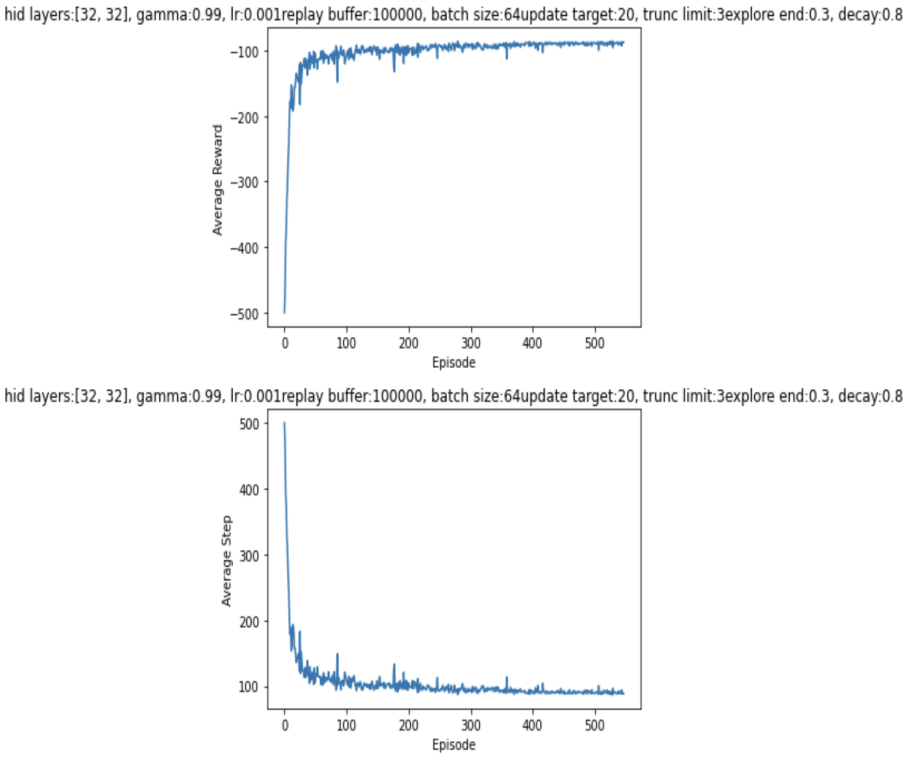


Figure 3: Average Rewards and Average Number of steps

1.1.4 Network: [64, 32], $\gamma = 0.99$, learning rate = 0.0005, replay buffer size = 10^5, batch size = 32, update target = 20, truncation limit = 1, explore end = 0.1, decay = 0.9

- Number of episodes taken(averaged over 10 runs) to solve the environment = 145.1.
- We made the following changes from the previous configuration-
  - Changed the network architecture to [64, 32]
  - Changed the learning rate to 0.0005
  - Changed batch size to 32
  - Changed the truncation limit to 1
  - Explore end = 0.1, Decay = 0.9

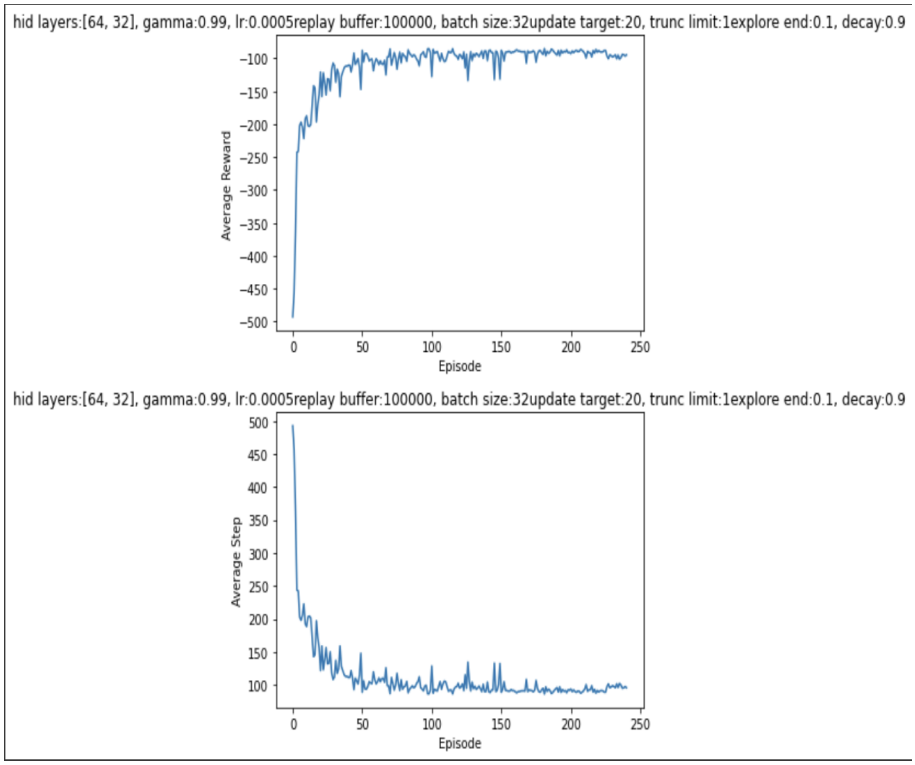


Figure 4: Average Rewards and Average Number of steps

1.1.5 Network: [32, 32], $\gamma = 0.99$, learning rate = 0.0005, replay buffer size = 10^5, batch size = 64, update target = 20, truncation limit = 2, explore end = 0.1, decay = 0.9

- Number of episodes taken(averaged over 10 runs) to solve the environment = 132.4.
- We made the following changes from the previous configuration-
  - Changed the network architecture to [32, 32]
  - Changed batch size to 64
  - Changed the truncation limit to 2

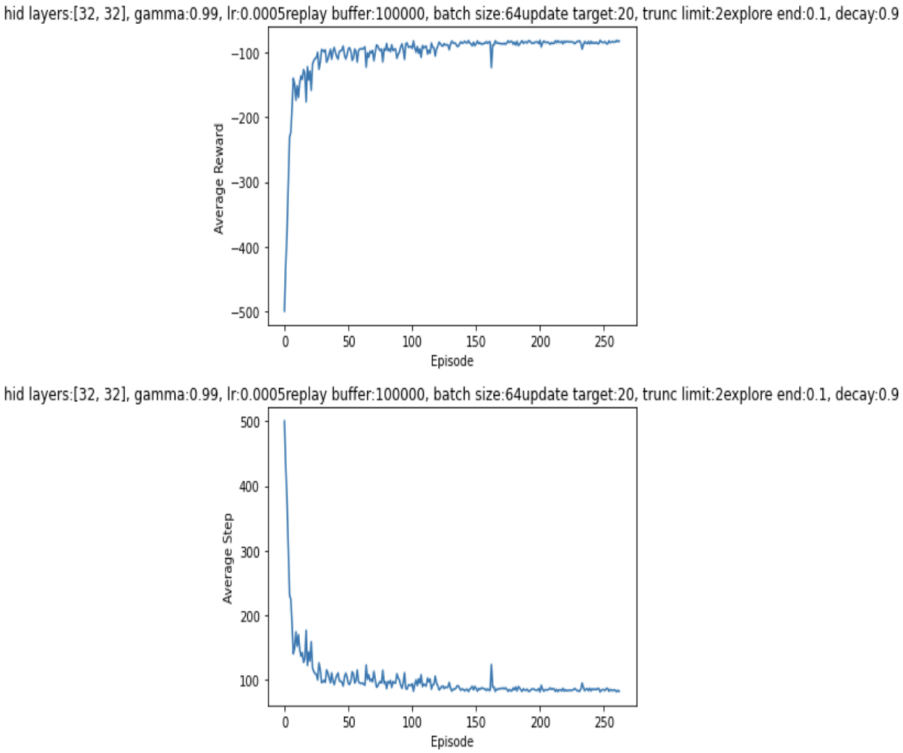


Figure 5: Average Rewards and Average Number of steps

1.1.6 Inferences

- Increasing the model complexity does not necessarily improve the agent's performance always. From our experiments we found that [32, 32] neural network architecture gave good results.
- Increasing the learning rate for faster convergence may not be a good idea always. We found learning rate = 0.0005 gave good performance.
- We noticed that increasing the truncation limit facilitates learning. Truncation limit of 2 performed better than 1 or 1.5.
- Increasing the batch size facilitates agent learning. From our experiments we found that batch size of 64 gave good results compared to batch size of 32.

1.2 CartPole-v1

- We tried solving the environment using the set of hyperparameters**(Network architecture : [64, 64], replay buffer size :** 10^5**, batch size = 64,** $\gamma = 0.99$**, learning rate = 0.0005, update target = 20, explore end = 0.01, decay = 0.995, truncation limit = 1)** originally given in the tutorial. For this configuration we tried multiple runs. Most of the time the agent was unable to solve the environment within 4000 episodes.

- For every hyperparameter configuration we ran 10 experiments. We averaged the performance of the agent across these 10 runs and presented the results.

- We improved the agent's performance**(considering number of episodes taken to solve the environment as performance metric)** 5 times by changing the hyperparameter configurations appropriately. The results are shown here.

1.2.1 Network: [128, 128], $\gamma = 0.99$, learning rate = 0.0001, replay buffer size = 10^5, batch size = 128, update target = 20, truncation limit = 2, explore end = 0.1, decay = 0.9

- Number of episodes taken(averaged over 10 runs) to solve the environment = 790.
- We made the following changes to the hyperparameter configuration given in the tutorial -
  - Changed the network architecture to [128, 128]
  - Reduced the learning rate to 0.0001
  - Increased the batch size to 128
  - Changed the truncation limit to 2
  - explore end = 0.1 and decay = 0.9

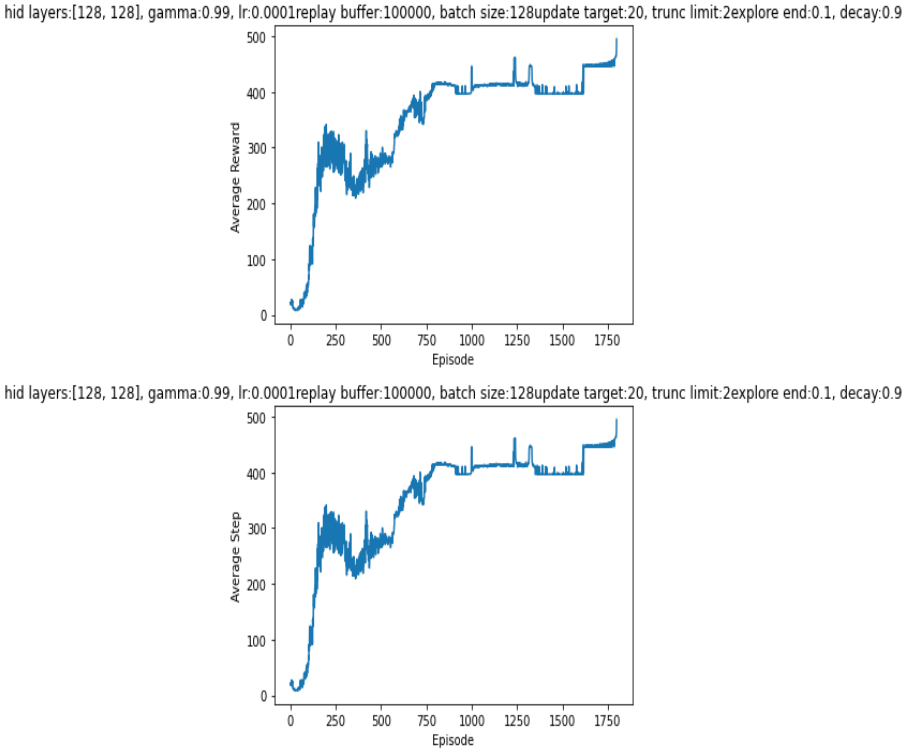


Figure 6: Average Rewards and Average Steps

1.2.2 Network: [64, 128], $\gamma = 1$, learning rate = 0.0001, replay buffer size = 10^5, batch size = 256, update target = 30, truncation limit = 2, explore end = 0.1, decay = 0.9

- Number of episodes taken(averaged over 10 runs) to solve the environment = 745.
- We made the following changes to the previous hyperparameter configuration -
  - Changed the network architecture to [64, 128]
  - Increased the value if γ to 1
  - Increased the batch size to 256
  - update target = 30

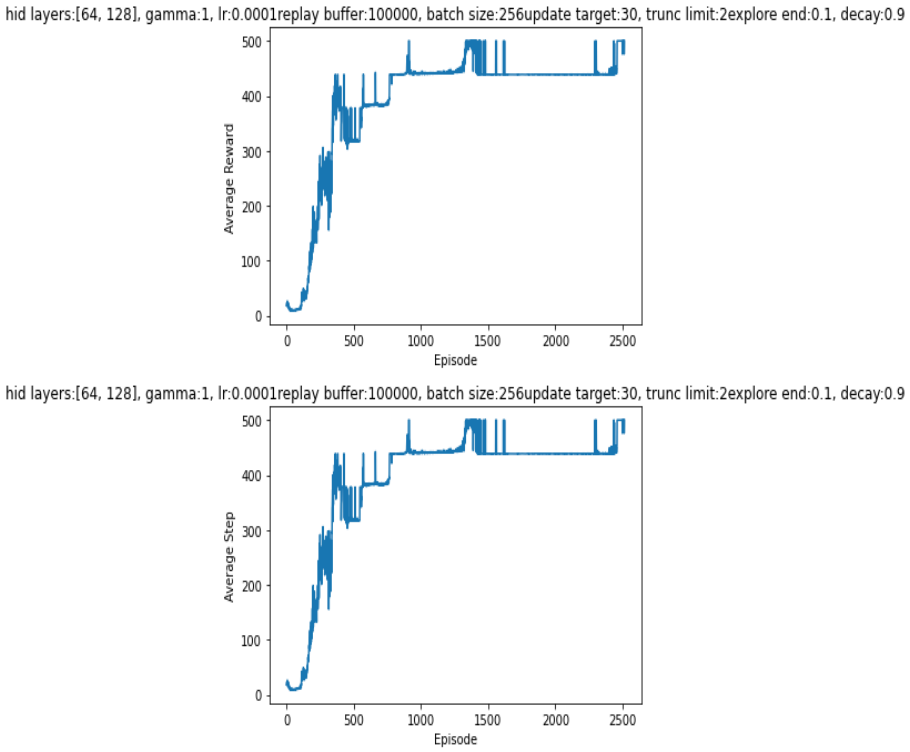


Figure 7: Average Rewards and Average Steps

1.2.3 Network: [64, 128], $\gamma = 0.99$, learning rate = 0.0001, replay buffer size = 10^5, batch size = 64, update target = 20, truncation limit = 2, explore end = 0.1, decay = 0.9

- Number of episodes taken(averaged over 10 runs) to solve the environment = 649.
- We made the following changes to the previous hyperparameter configuration -
  - Reduced the value of γ to 0.99
  - Reduced the batch size to 64
  - update target = 20

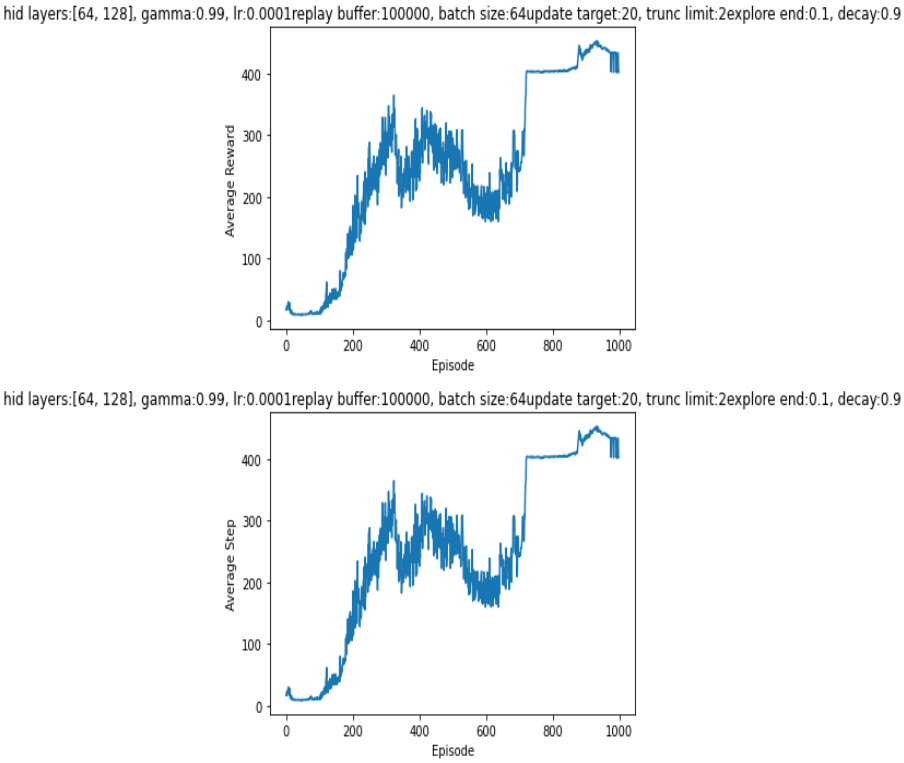


Figure 8: Average Rewards and Average Steps

1.2.4 Network: [128, 128], $\gamma = 1$, learning rate = 0.001, replay buffer size = 10^5, batch size = 256, update target = 20, truncation limit = 2, explore end = 0.1, decay = 0.9

- Number of episodes taken(averaged over 10 runs) to solve the environment = 622.
- We made the following changes to the previous hyperparameter configuration -
  - Increased the value of γ to 1
  - Increased the learning rate to 0.001
  - Increased the batch size to 256

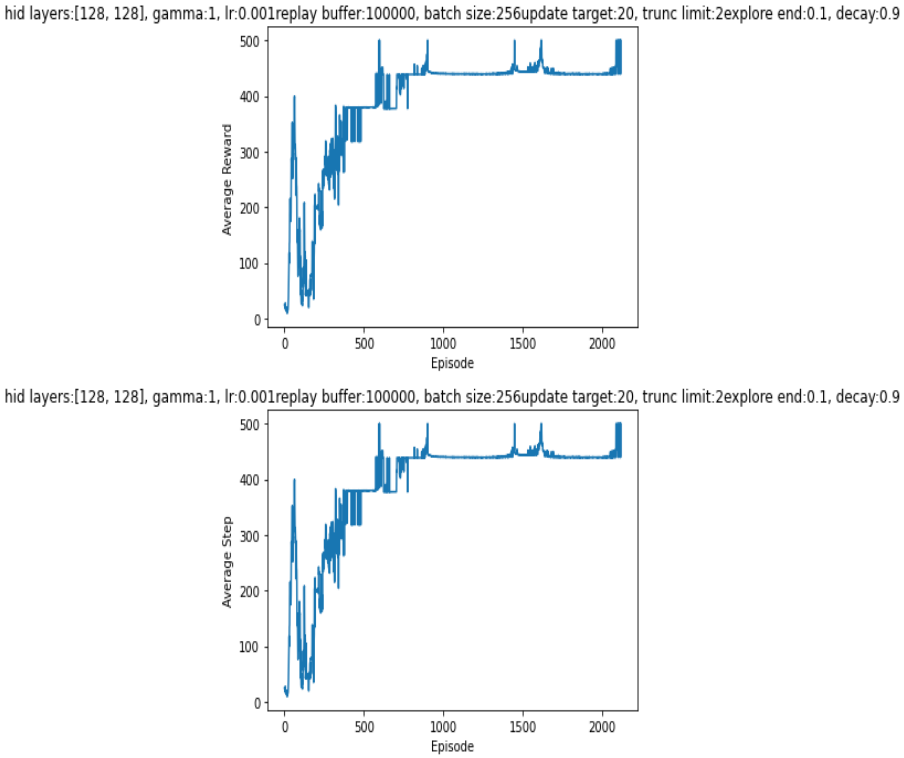


Figure 9: Average Rewards and Average Steps

1.2.5 Network: [32, 32, 32], $\gamma = 1$, learning rate = 0.001, replay buffer size = 10^5, batch size = 512, update target = 30, truncation limit = 1.5, explore end = 0.2, decay = 0.8

- Number of episodes taken(averaged over 10 runs) to solve the environment = 399.
- We made the following changes to the previous hyperparameter configuration -
  - Changed the network architecture to [32, 32, 32]
  - update target = 30
  - truncation limit = 1.5
  - Increased the batch size to 512
  - explore end = 0.2, decay = 0.8

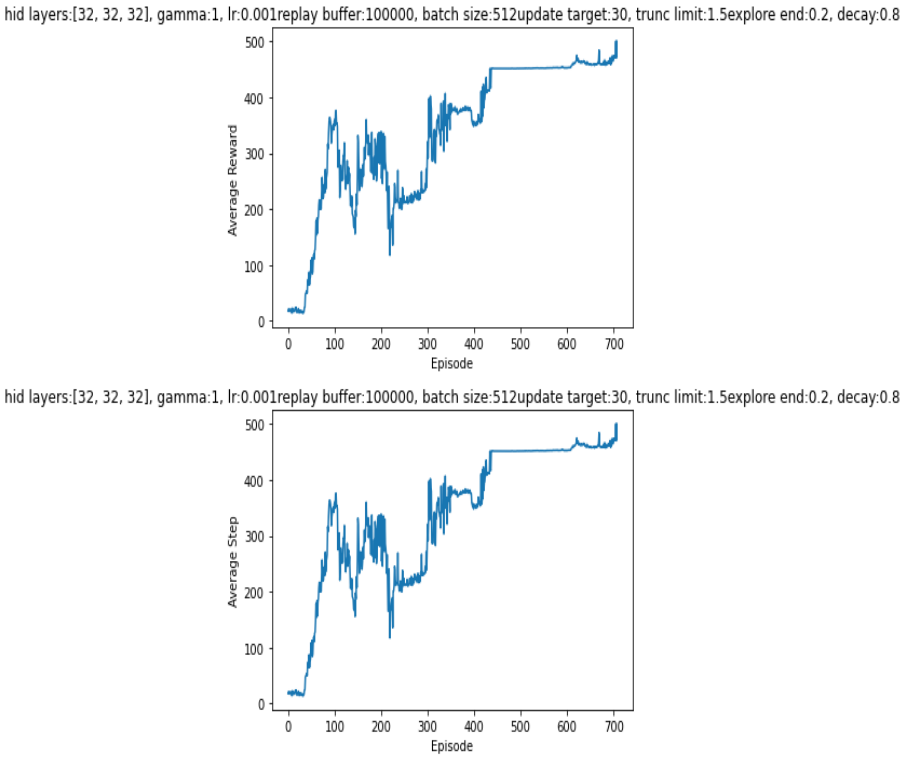


Figure 10: Average Rewards and Average Steps

1.2.6 Inferences

- Neural networks where the number of neurons across hidden layers are same gave good results. [32, 32, 32] and [128, 128] architecture performed better than the other architectures we tried.
- We found that high value of γ is more suitable for solving the CartPole-v1 environment. We experimented with $\gamma = 0.99$ and $\gamma = 1$.
- We found that increasing the learning rate significantly improved the performance of the agent. Learning rate of 0.001 gave best results.
- Increasing the batch size facilitates learning. We noticed that batch size of 512 and 256 improved performance significantly.

1.3 MountainCar-v0

- We tried solving the environment using the set of hyperparameters**(Network architecture : [64, 64], replay buffer size :** 10^5**, batch size = 64,** $\gamma = 0.99$**, learning rate = 0.0005, update target = 20, explore end = 0.01, decay = 0.995, truncation limit = 1)** originally given in the tutorial. For this configuration we tried multiple runs. The agent was unable to solve the environment within 3000 episodes.

- We tried with some other hyperparameter configurations but the agent was unable to solve the environment within 3000 episodes.

- For the agent to figure out the optimal strategy on its own, it needs to be provided with an appropriate reward function. Simply giving a positive reward once the car reaches the destination by trial and error in some episode and a negative reward for all other time steps will not work and it will take a pretty long time before the network learns the optimal strategy. So we used a customised reward function for training our agent.

- For every hyperparameter configuration we ran 10 experiments. We averaged the performance of the agent across these 10 runs and presented the results.

- We improved the agent's performance**(considering number of episodes taken to solve the environment as performance metric)** 5 times by changing the hyperparameter configurations appropriately. The results are shown here.

1.3.1 Network: [64, 64], $\gamma = 0.99$, learning rate = 0.001, replay buffer size = 20000, batch size = 32, update target = 30, truncation limit = 2, explore end = 0.1, decay = 0.9

- For this configuration we generated 10 runs, but even with customised reward function the agent was unable to solve the environment within 3000 episodes.
- For this configuration agent was learning very slowly.
- It is clear from the plots that the performance of the agent is very poor.

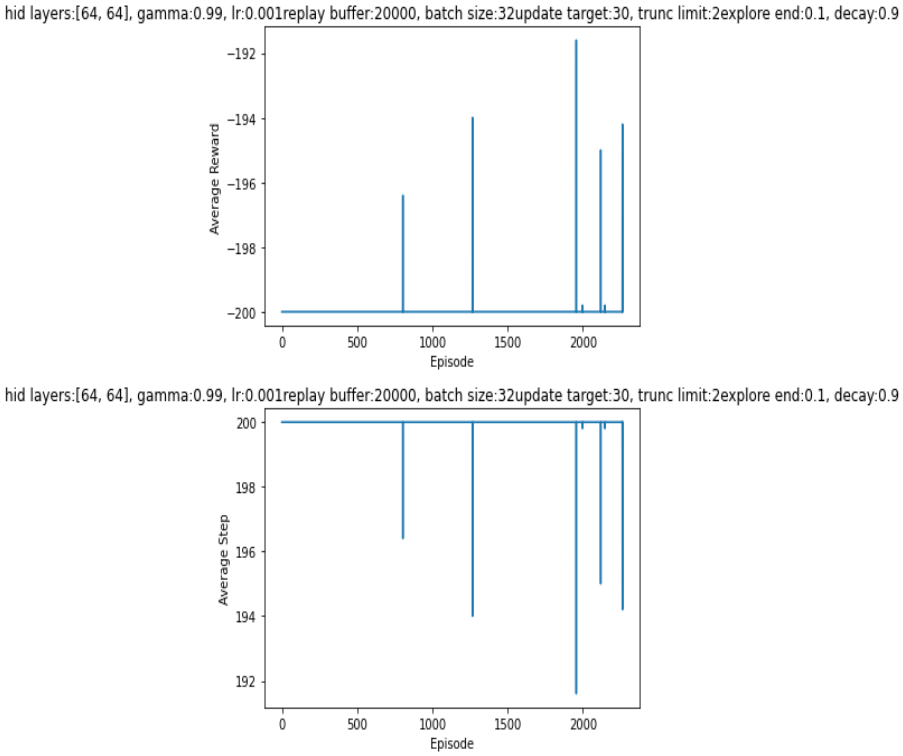


Figure 11: Average Rewards and Average Steps

1.3.2 Network: [24, 48], $\gamma = 0.99$, learning rate = 0.001, replay buffer size = 20000, batch size = 32, update target = 30, truncation limit = 2, explore end = 0.1, decay = 0.9

- For this configuration we ran 10 experiments. The agent was able to solve the environment 3 times.
- Number of episodes taken to solve the environment for the 3 runs were - 1521, 946 and 2989.
- It is clear from the plots that the performance of the agent improved significantly.
- We made the following change to the previous hyperparameter configuration -
  - Changed the network architecture to [24, 48]

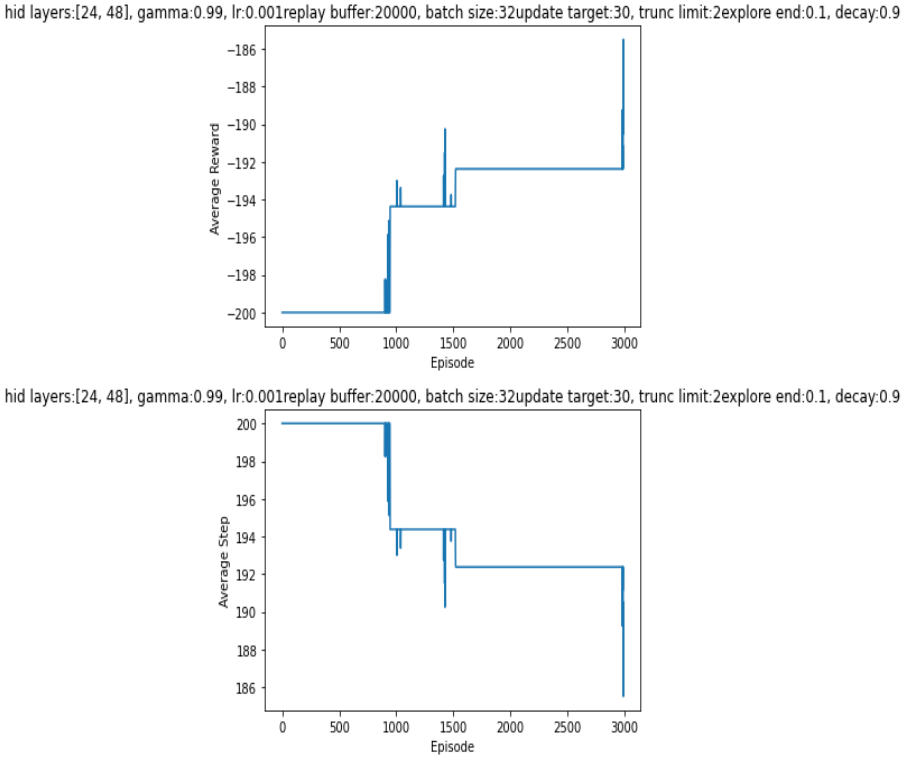


Figure 12: Average Rewards and Average Steps

1.3.3 Network: [64, 64], $\gamma = 0.99$, learning rate = 0.0005, replay buffer size = 10^5, batch size = 64, update target = 20, truncation limit = 1, explore end = 0.01, decay = 0.995

- For this configuration we ran 10 experiments. The agent was able to solve the environment 3 times.
- Number of episodes taken to solve the environment for the 3 runs were - 981, 1092 and 2875.
- We made the following changes to the previous hyperparameter configuration -
  - Changed the network architecture to [64, 64]
  - Reduced the learning rate to 0.0005
  - Increased the batch size to 64
  - truncation limit = 1, update target = 20
  - Increased the replay buffer size to 10^5
  - explore end = 0.01, decay = 0.995

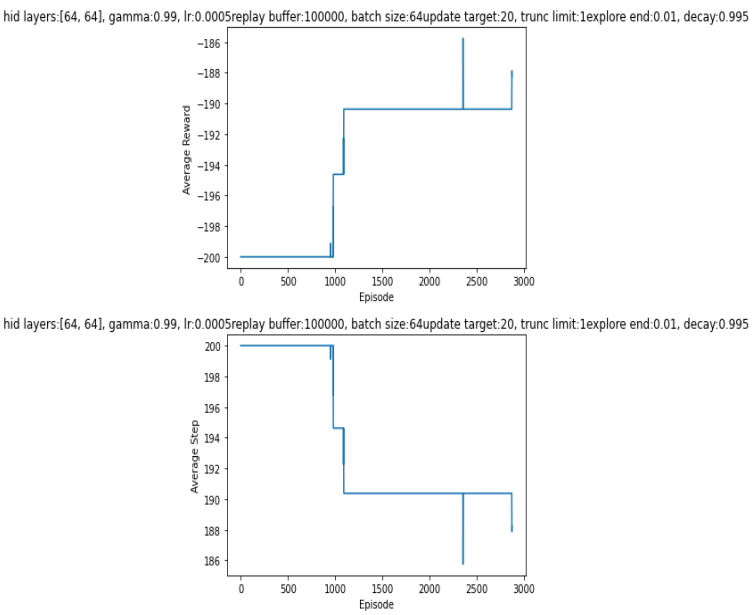


Figure 13: Average Rewards and Average Steps

1.3.4 Network: [128, 128], $\gamma = 0.99$, learning rate = 0.0005, replay buffer size = 10^5, batch size = 64, update target = 20, truncation limit = 1, explore end = 0.01, decay = 0.995

- For this configuration we ran 10 experiments. The agent was able to solve the environment 3 times.
- Number of episodes taken to solve the environment for the 3 runs were - 2622, 1127 and 708.
- We made the following change to the previous hyperparameter configuration -
  - Changed the network architecture to [128, 128]

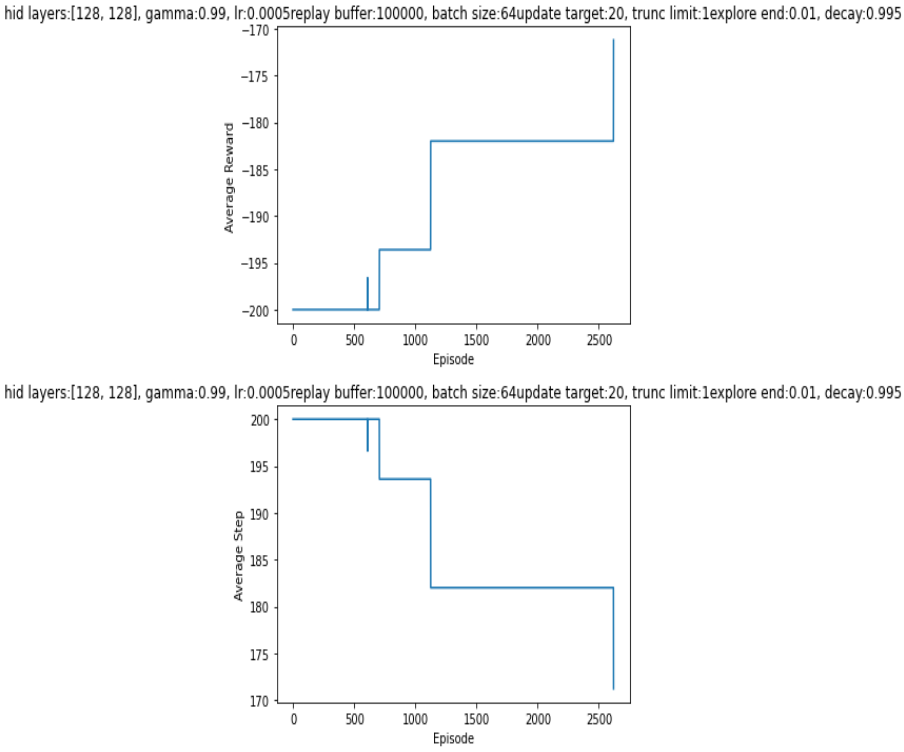


Figure 14: Average Rewards and Average Steps

1.3.5 Network: [32, 32, 32], $\gamma = 0.99$, learning rate = 0.0005, replay buffer size = 10^5, batch size = 128, update target = 20, truncation limit = 1, explore end = 0.01, decay = 0.995

- For this configuration we ran 10 experiments. The agent was able to solve the environment 3 times.
- Number of episodes taken to solve the environment for the 3 runs were - 1030, 1475 and 961.
- We made the following changes to the previous hyperparameter configuration -
  - Changed the network architecture to [32, 32, 32]
  - Increased the batch size to 128

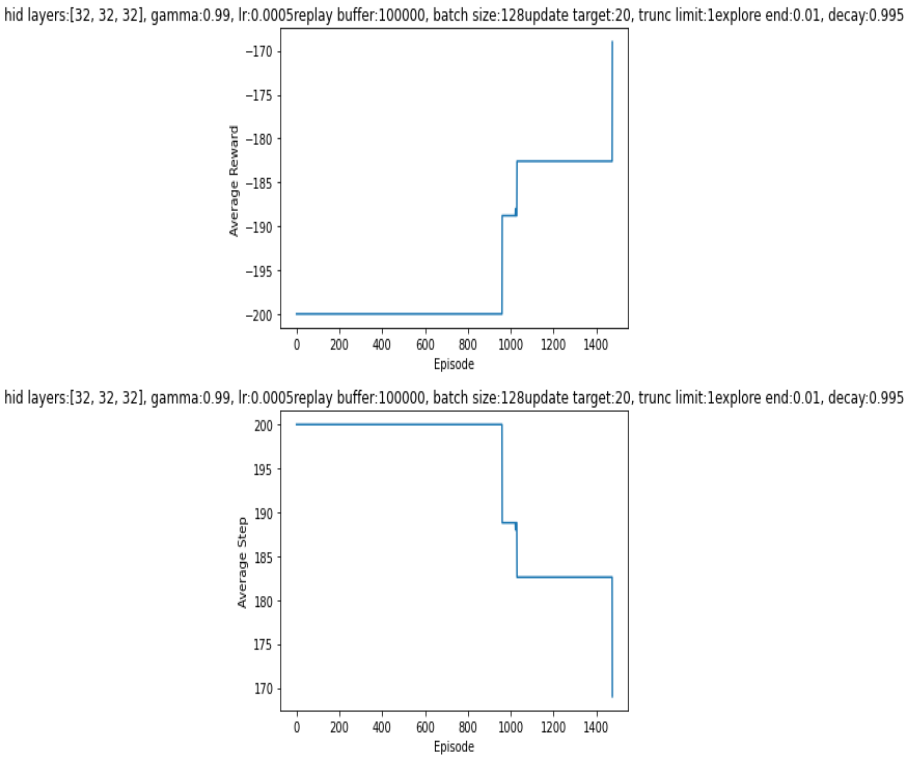


Figure 15: Average Rewards and Average Steps

1.3.6 Inferences

- Neural networks where the number of neurons across hidden layers are same gave good results. [32, 32, 32] and [128, 128] architecture performed better than the other architectures we tried.
- We found that high value of γ is more suiatble for solving the MountainCar-v0 environment. We experimented with $\gamma = 0.99$.
- Increasing the learning rate for faster convergence may not be a good idea always. We found learning rate = 0.0005 gave good results.
- Increasing the batch size facilitates learning. We noticed that batch size of 64 and 128 improved performance significantly compared to batch size of 32.
- Due to computational overhead we trained our agent for 3000 episodes. Increasing the number of episodes might help the agent to converge to better policies.

2.2 Algorithm: Actor Critic (A2C)

As per given directions, the Actor-Critic network or A2C was implemented in pytorch by taking queues from the network provided in the tutorial.

The variations to Actor-Critic methods, for:

- One-step: $\delta_t^{(1)} = R_{t+1} + \gamma v(s_{t+1}) - v(s_t)$
- n-step: $\delta_t^{(n)} = \sum_{t'=t}^{n+t-1} \gamma^{t'-t} R_{t'+1} + \gamma^n v(s_{n+t}) - v(s_t)$
- Full return: $\delta_t^{(T)} = \sum_{t'=t}^{T} \gamma^{t'-t} R_{t'+1} - v(s_t)$; T being the maximum number of timesteps considered.

have been implemented, along with tune-able number and sizes of hidden layers, learning rate α, and number of episodes.

Hyperparameters:

- $\gamma = 0.99$, #Fixed
- $\beta_1 = 0.99$, #Fixed
- $\beta_2 = 0.999$, #Fixed
- **Number of hidden layers**
- **Size of each hidden layer**
- **n-step:**
  - -1 $\rightarrow$ Full Return
  - 1 $\rightarrow$ One-step Return
  - n (> 1) $\rightarrow$ n-step Return

Code Snippet: `calculateLoss()` function

```
# Full return || Case with (n = -1)
if n == -1:
    rewards = []

    # Discounting the Rewards:
    dis_reward = 0
    for reward in self.rewards[::-1]:
        dis_reward = reward + gamma * dis_reward
        rewards.insert(0, dis_reward)

    # Normalizing the rewards:
    rewards = torch.tensor(rewards)
    rewards = (rewards - rewards.mean()) / (rewards.std())

    # Implementing the Algorithm for Full Returns (Monte Carlo)
    loss = 0
    for log_prob, value, reward in zip(self.log_probs, self.state_values, rewards):
        d_t = reward  - value.item()
        action_loss = -log_prob * d_t
        value_loss = (d_t ** 2)
        loss += (action_loss + value_loss)

    return loss

else:
    loss = 0

    for t, state_value in enumerate(zip(self.log_probs, self.state_values)):
        log_prob, value = state_value
```

```
        d_t = 0

        # n-Step TD Return
        # Summation t'=t to t'=n+t-1
        for tp in range(t, min(n+t, len(self.rewards)-1)):
            d_t += ((gamma**(tp - t)) * self.rewards[tp]) / n

        if min(n+t, len(self.rewards)-1) == n+t:
            d_t += ((gamma**n) * self.state_values[n+t])

        d_t = d_t - value

        action_loss = -log_prob * d_t
        value_loss = (d_t ** 2)
        loss += (action_loss + value_loss)

    return loss
```

The above is the desired source code snippet in which the different TD Return variations have been implemented.

2.2.1 Acrobot-v1

Observations:

- In general, it has been observed that there is a significant correlation between `runtime`, `n_step` and `learning rate`. The parametric importance has been inferred with respect to episodic rewards.
- The number and size of hidden layers do impact the training time, but given best set of hyperparameters and enough number of episodes to train, the network would eventually learn to solve the environment.
- Almost all of the 20 best runs started learning the optimal policy after about 450 episodes, with few exceptional configurations performing really well and learning the optimal policy after about 200 episodes.
- With **TD(1)**, the `Best Run`:
  - **Hidden Layers:** [32, 64, 64],[64, 64],[64, 64],[64, 64],[64, 64]
  - **Learning Rate:** 0.002,0.0005,0.003,0.002,0.001
  - **Maximum Episodic Reward:** -72,-65,-68,-80,-71
- With **TD(3)**, the `Best Run`:
  - **Hidden Layers:** [64,64], [32,64,64], [64,64], [32,64,64], [64,64]
  - **Learning Rate:** 0.003,0.003,0.002,0.002,0.002
  - **Maximum Episodic Reward:** -77, -64, -75, -62, -72, .
- With **TD(5)**, the `Best Run`:
  - **Hidden Layers:** [64, 64],[32, 64, 64],[64, 64],[32, 64, 64],[64, 64]
  - **Learning Rate:** 0.001, 0.003, 0.002, 0.002, 0.003
  - **Maximum Episodic Reward:** -64, -77, -69, -76, -79
- With **TD(20)**, the `Best Run`:
  - **Hidden Layers:** [64, 64], [64, 64], [128, 64],[64, 64], [64, 64]
  - **Learning Rate:** 0.002, 0.003, 0.0005, 0.0005, 0.001
  - **Maximum Episodic Reward:** -80, -67, -90, -91, -87
- With **TD(25)**, the `Best Run`:
  - **Hidden Layers:** [64, 64], [64, 64], [64,64]

  - **Learning Rate:** 0.002, 0.001, 0.003
  - **Maximum Episodic Reward:** -99, -77, -78

- With **TD(MC)**, the `Best Run`:
  - **Hidden Layers:** [64, 64], [32, 64, 64], [64, 64]
  - **Learning Rate:** 0.001, 0.001, 0.003
  - **Maximum Episodic Reward:** -79, -84, -70

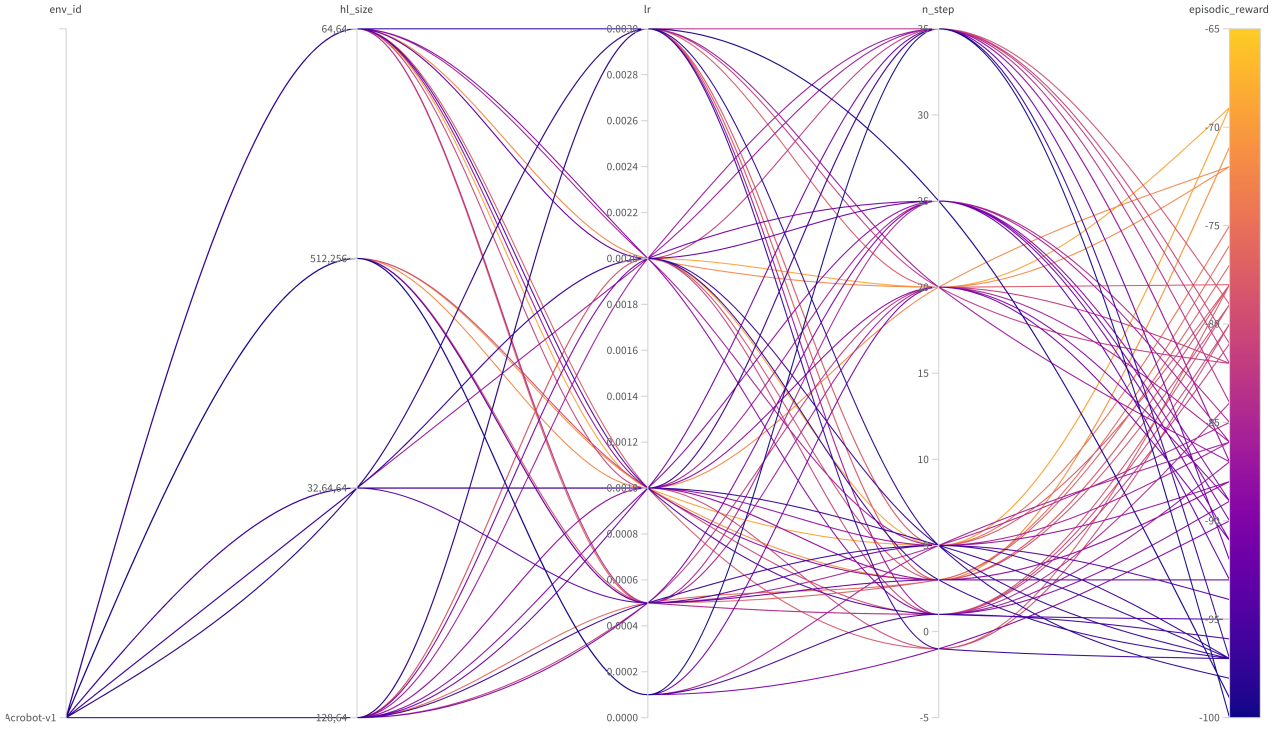
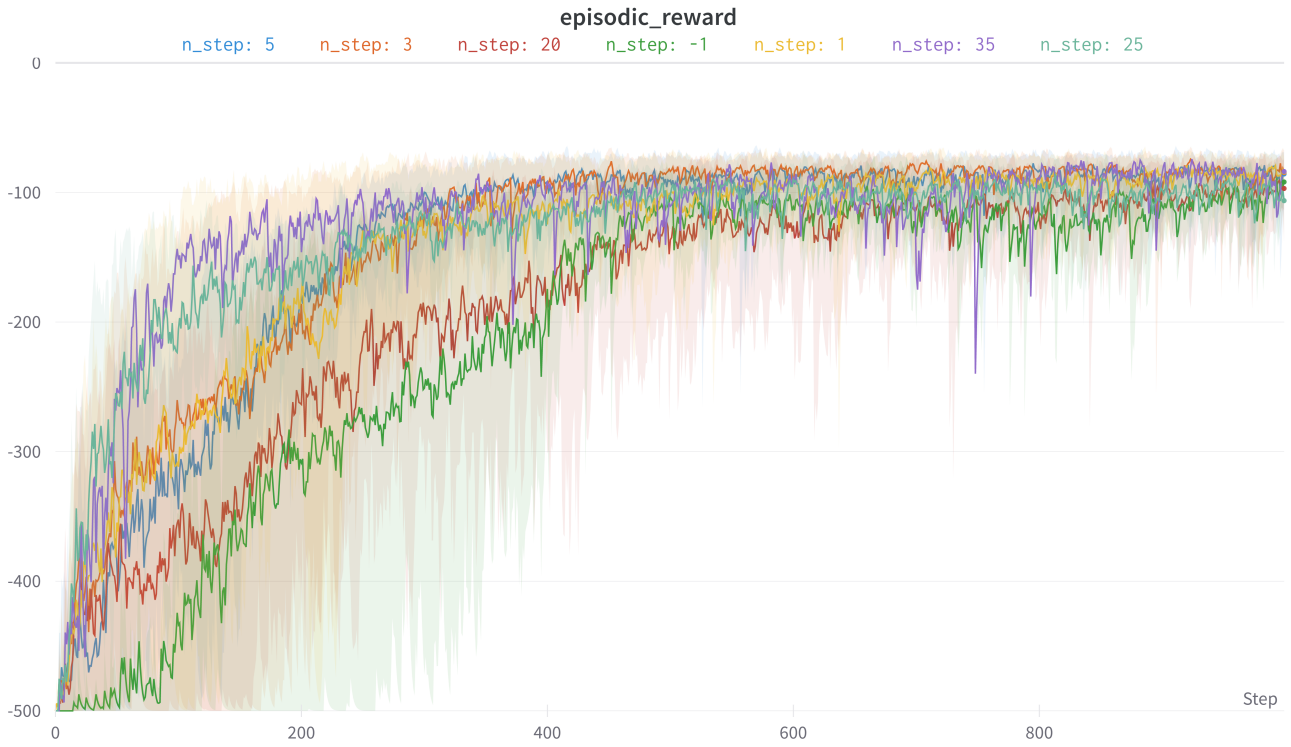
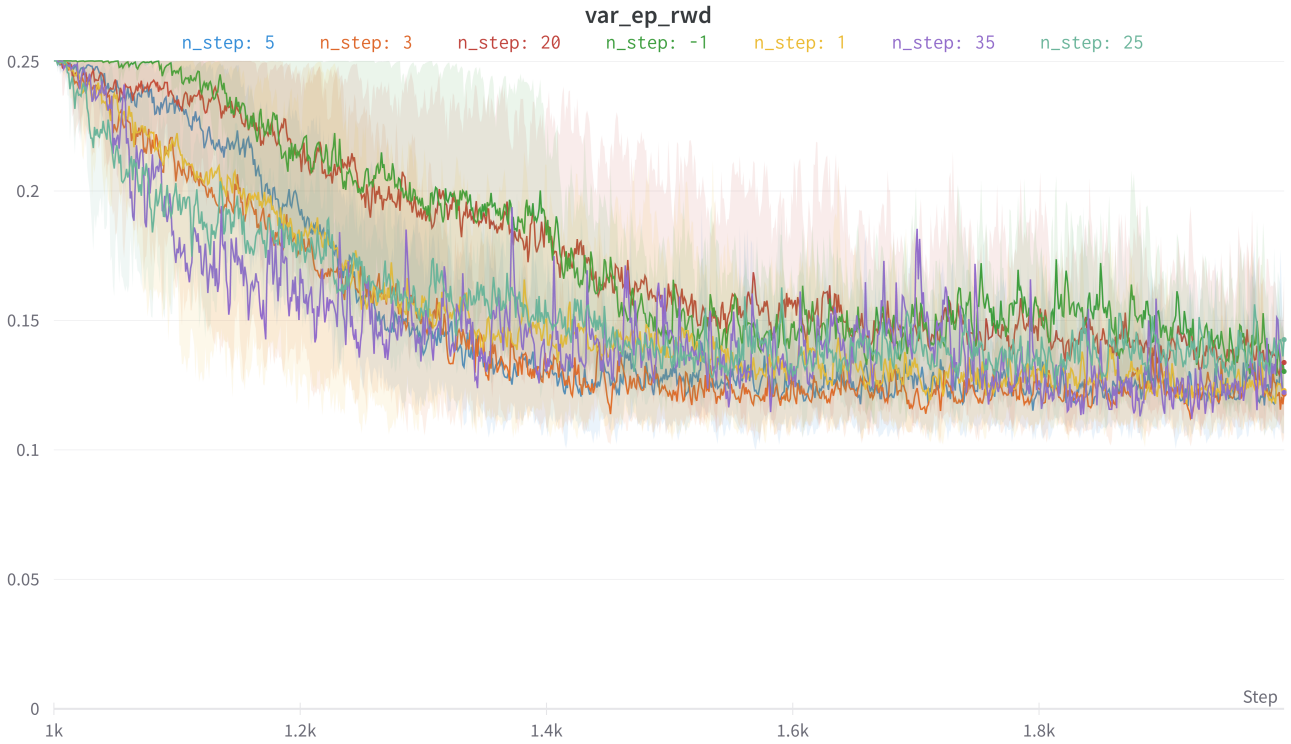


Figure 1: Parallel Plot for Winning Combination of Hyperparameters



(a) Average Episodic Reward for Actor-Critic for various configurations of TD(n) for Acrobot-v1 Gym Environment.



(b) Average Variance for Actor-Critic for various configurations of TD(n) for Acrobot-v1 Gym Environment.

Figure 2: Comparative Plots for Different TD(n) Values for Acrobot-v1 Gym Environment.

2.2.2 Cartpole-v1

Observations:

- In general, it has been observed that there is a significant correlation between `runtime`, `n_step` and `learning rate`. The parametric importance has been inferred with respect to episodic rewards.
- The number and size of hidden layers do impact the training time, but given best set of hyperparameters and enough number of episodes to train, the network would eventually learn to solve the environment.
- After approximately 350-400 episodes, the environment is solved by the best configurations across 10 different random seeds.
- With n-step returns, TD(n=5) and TD(n=3) (or n=10,15), significantly outperforms TD(1) and TD(MC) methods in terms of number of steps required to reach goal state.
- With **TD(1)**, the `Best Run`:
  - **Hidden Layers:** [15, 15]
  - **Learning Rate:** 0.003
  - **Maximum Episodic Reward:** 210
- With **TD(3)**, the `Best Run`:
  - **Hidden Layers:** [32, 32], [128]
  - **Learning Rate:** 0.0025, 0.003
  - **Maximum Episodic Reward:** 500, 448.
  - There's very high variance noticed with [32, 32] hidden layer configuration. Given enough number of episodes, the run with Hidden Layer [128] would've also converged to yield better results.
- With **TD(5)**, the `Best Run`:
  - **Hidden Layers:** [128], [20,15]
  - **Learning Rate:** 0.0025, 0.003
  - **Maximum Episodic Reward:** 500, 483
- With **TD(10)**, the `Best Run`:
  - **Hidden Layers:** [32, 32], [128]
  - **Learning Rate:** 0.003
  - **Maximum Episodic Reward:** 500
  - The network with Hidden Layer [128], took longer to converge to an optimal policy, than the network with Hidden Layer [32, 32].
- With **TD(15)**, the `Best Run`:
  - **Hidden Layers:** [15, 15], [128], [20,25]
  - **Learning Rate:** 0.003, 0.0025, 0.003
  - **Maximum Episodic Reward:** 500
  - Here, the network with Hidden layer [128], converged faster to an optimal policy than the other configurations.
- With **TD(MC)**, the `Best Run`:
  - **Hidden Layers:** Pretty Much all Hidden Layer Combinations, which we have tried, [32, 32], [20,25], [20,15], [20,10], [15,15], [128], converge to optimal policies.

- **Learning Rate:** 0.001, 0.0025, 0.003
- **Maximum Episodic Reward:** 500
- The network with Hidden Layer [128], took longer to converge to an optimal policy, than the network with Hidden Layer [32, 32].

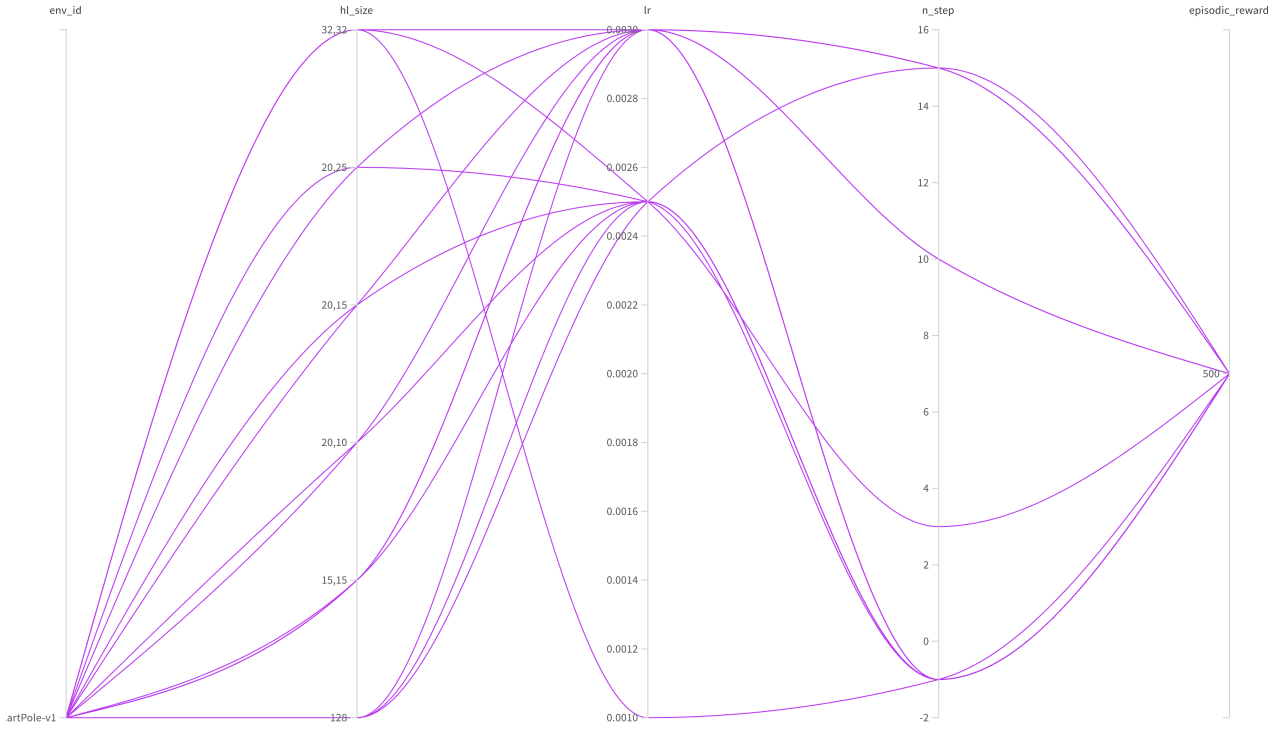
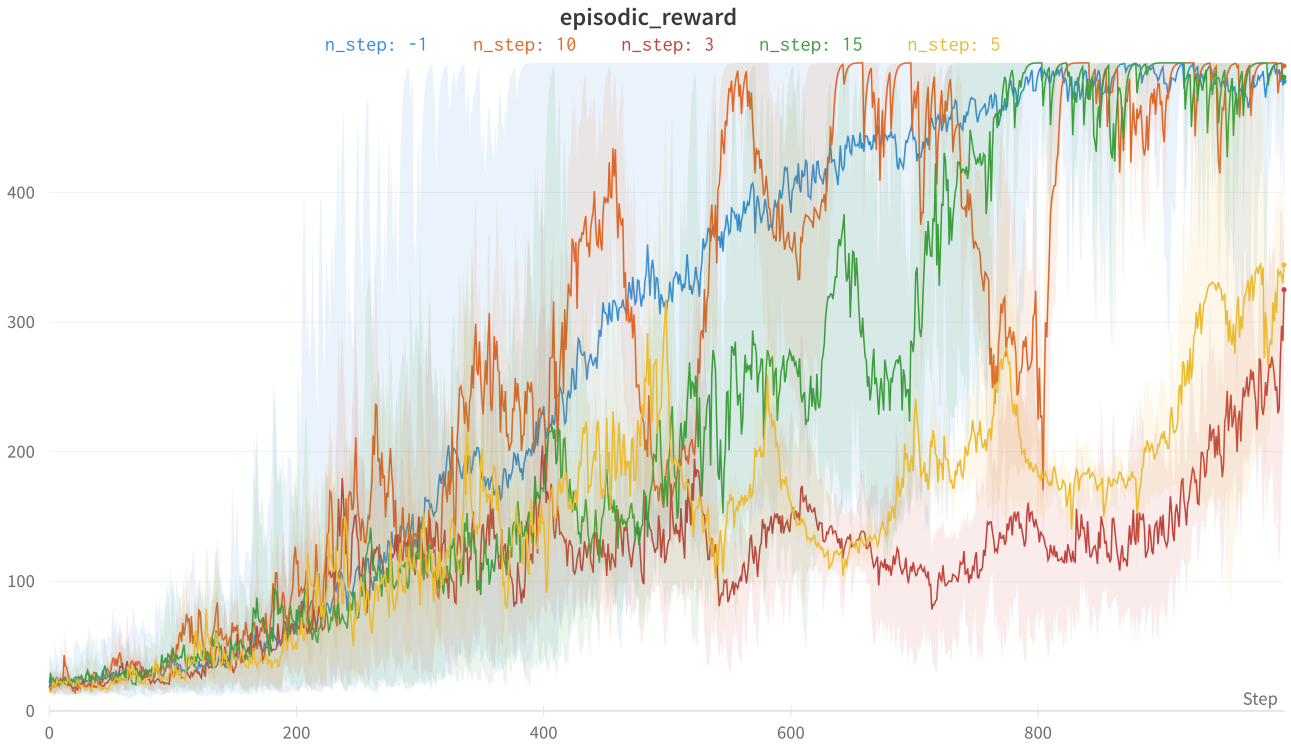
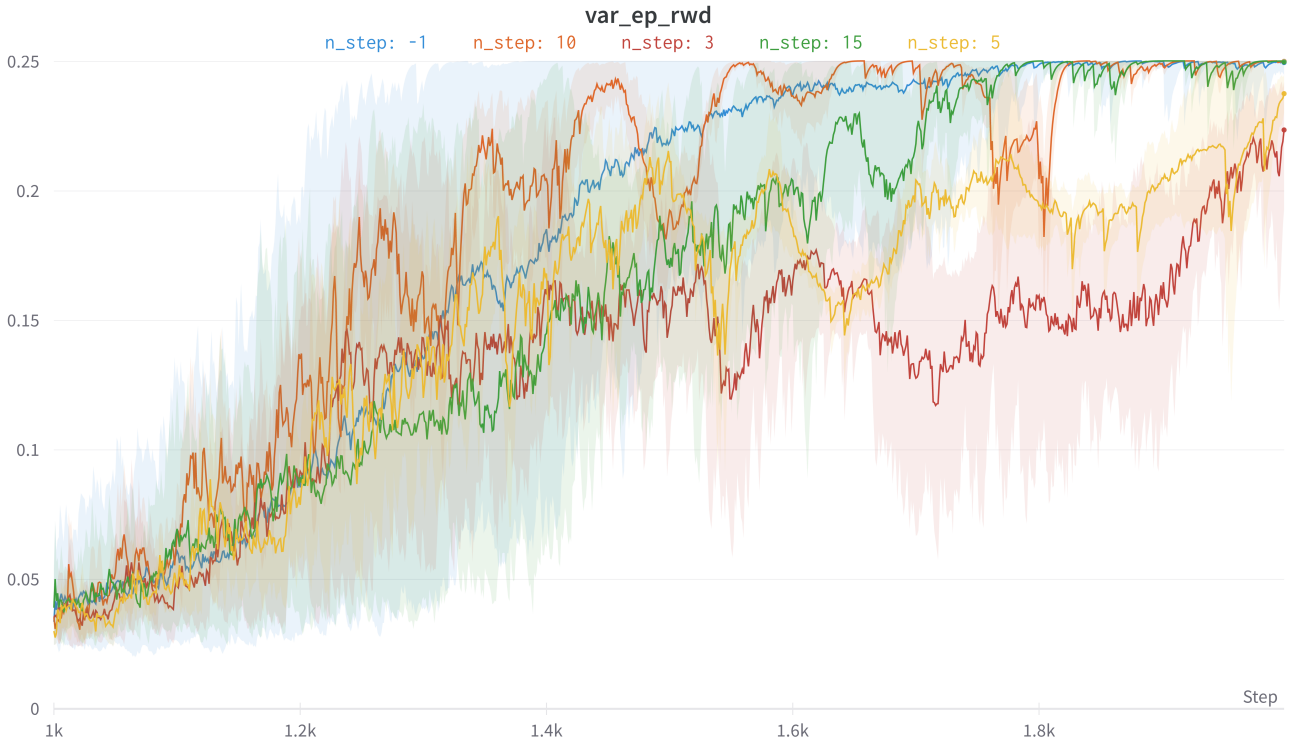


Figure 3: Parallel Plot for best set of Hyperparameters for CartPole-v1 Gym Environment using A2C Algorithm.



(a) Average Episodic Reward for Actor-Critic for various configurations of TD(n) for CartPole-v1 Gym Environment.



(b) Average Variance for Actor-Critic for various configurations of TD(n) for CartPole-v1 Gym Environment.

Figure 4: Comparative Plots for Different TD(n) Values for CartPole-v1 Gym Environment.

2.2.3 MountainCar-v0

Observations:

- For the `MountainCar-v0` Gym Environment, we have tried a bunch of hyperparameters and none of them yielded improvements over the other.
- The environment truncates after 200 timesteps, with -1 reward for each timestep.
- All of the hyperparameter combinations in our set resulted in a -200 reward which means the environment's task has not been solved.
- We tried implementing reward shaping by using the concept of Mechanical Energy(Potential Energy + Kinetic Energy). Which in theory should allow the Mountain Car to reach its goal, but in practice within 1000 epsiodes[1] it never learnt an optimal policy.
- Attaching below a parallel plot fortifying my claims.

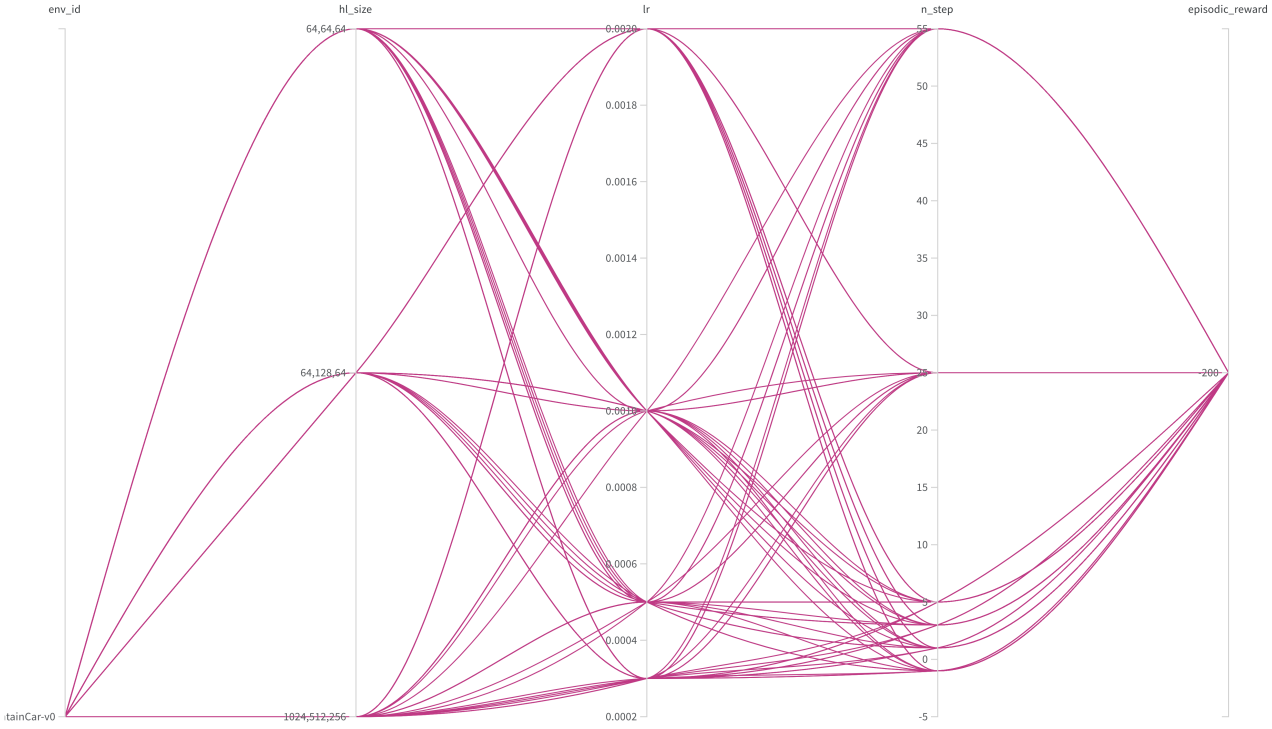


Figure 5: Parallel Plot for the set of Hyperparameters for MountainCar-v0 Gym Environment using A2C Algorithm.

[1] In one of the untracked runs, the mountain car learned to reach its goal after 9000 episodes.