International Institute of Information Technology
Bangalore

Reinforcement Learning
AI-832

Assignment 1

Monjoy Narayan Choudhury
-IMT2020502

March 27, 2024

Part 1 - The Cartpole problem

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.

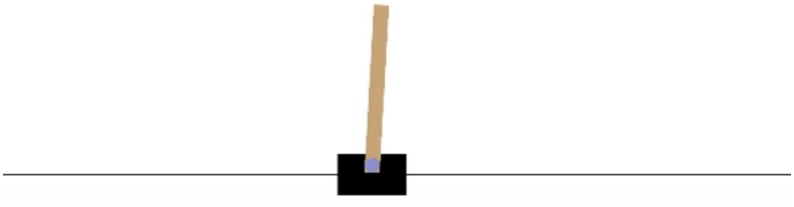

Figure 1: Cartpole Problem as shown in OpenAI gym/gymnasium

States

The OpenAI Cartpole system returns the following information as the state:

1. Cart Position
2. Cart Velocity
3. Pole Angle
4. Pole Angular Velocity

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

Figure 2: Range of values of the state variables.

The range of the values these states can be represented with are given in Fig. 2.

Termination conditions

Based on the state variables ranges defined above, the following are the conditions on the variables that cause the termination of the state:

1. The cart x-position (index 0) can take values between (-4.8, 4.8), but the episode terminates if the cart leaves the (-2.4, 2.4) range.
2. The pole angle can be observed between (-.418, .418) radians (or ±24°), but the episode terminates if the pole angle is not in the range (-.2095, .2095) (or ±12°)
3. Truncation: Episode length is greater than 500 (200 for v0)

Action Space

The action is an nd-array with shape (1,) which can take values 0, 1 indicating the direction of the fixed force the cart is pushed with.

- 0: Push the cart to the left
- 1: Push the cart to the right

Reward Space

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. The threshold for rewards is 500 for v1 and 200 for v0.

Starting state

All observations are assigned a uniformly random value in (-0.05, 0.05)

Designing the DQN

For implementing a DQN we heavily inspire our code from the Deepmind paper and have the following aspects to it.

DQN Architecture

We use a simplified architecture for our deep Q Network. This is to avoid long training times as we need to perform parameter search experiments as one of the result objectives of this work. The architecture consists of the following layers:

1. Linear Layer taking in the state vector and projecting it to a 128-dimension space, followed by a ReLU activation.
2. Linear Layer from 128 dimensions to 56 followed by ReLU activation.
3. Linear Layer from 56 dimensions to 2 (number of possible actions).

Loss Function

Soft L1 loss is a smoother version of regular L1 loss. It combines the benefits of both L1 and L2 loss (mean squared error).
For small errors, it acts like L2 loss, squaring the difference to make small mistakes less punishing. For larger errors, it behaves like L1 loss, using the absolute difference to handle outliers effectively. This combination makes it a good choice for many machine-learning tasks, especially when you want to balance handling outliers with smoother learning during training.

Off-Policy Implementation and Soft Update

Since the Deep Q Networks are implemented in an off-policy way. We also need to simulate the same in our implementation. This comes from the fact that the experience used from experience replay was generated by a different policy while the policy updated is a different one. In practice, we take the same model architecture and create 2 networks policy and target net and match their state dictionaries initially so that start from the same initialization. At this moment these nets are practically identical. We compute the next step using the policy net and perform a soft update on the target net.
The soft update is defined as the following. This ensure that there is stability in the update process in the target net.

$$\theta'_{new} = \tau\theta + (1 - \tau)\theta'$$

Experience Replay Implementation

To create a deep Q network we need uncorrelated samples. For this, we need to cache the episodes seen previously (experiences). For this as suggested in the DeepMind paper, we implement a simple Experience Replay buffer that stores an experience as a list (deque in our implementation) of state, action, next state, and reward tuple in each of its buffer entries. We take a batch of these entries in the replay buffer to perform the training step. This ensures uncorrelated samples while introducing an off-policy nature to training which can be updated using soft rules as discussed above.

Part 2 - Hyperparameter Search for policy

As part of the second deliverable, we need to analyze the impact of the various hyperparameters, we break the analysis into the following parts.

List of Hyperparameters

The possible hyper-parameters in our implementations are

1. Batch size of sampling from experience replay buffer.
2. Size of the experience buffer.
3. Discount factor (γ).
4. The ratio for smoothening contribution in the soft update of parameters (τ).
5. Learning Rate of the Optimizer.

To understand the effect for each, we use the following metrics.

1. Average Q value curves with respect to epochs.
2. Loss curves with respect to epochs.
3. Average Reward curves with respect to epochs.

The metric selections were done based on the DeepMind paper on playing Atari as discussed in class where they try to evaluate the performance using these two plots.
For our experiments, we will keep our number of episodes/epochs to 500 as we need to perform a large number of hyperparameter analyses and would become time-consuming if every epoch takes a large amount of time.

Observations

Before we start our discussion on the impact of hyperparameters in the training of the Deep RL agent. We use the following default values for the rest while we vary the others:

1. BATCH_SIZE = 512
2. GAMMA = 0.4
3. EPS_START = 0.9
4. EPS_END = 0.05
5. EPS_DECAY = 1000
6. TAU = 0.005
7. LR = 1e-4
8. REPLAY BUFFER SIZE = 100000

We run each of the experiments for 500 episodes/ epochs. With this set of hyperparameters it takes around 4-5 mins to train on a Kaggle P100 GPU.

Experiments on Replay Buffer

We take the following replay buffer sizes - [1000,10000,100000,1000000] and note the following observations.

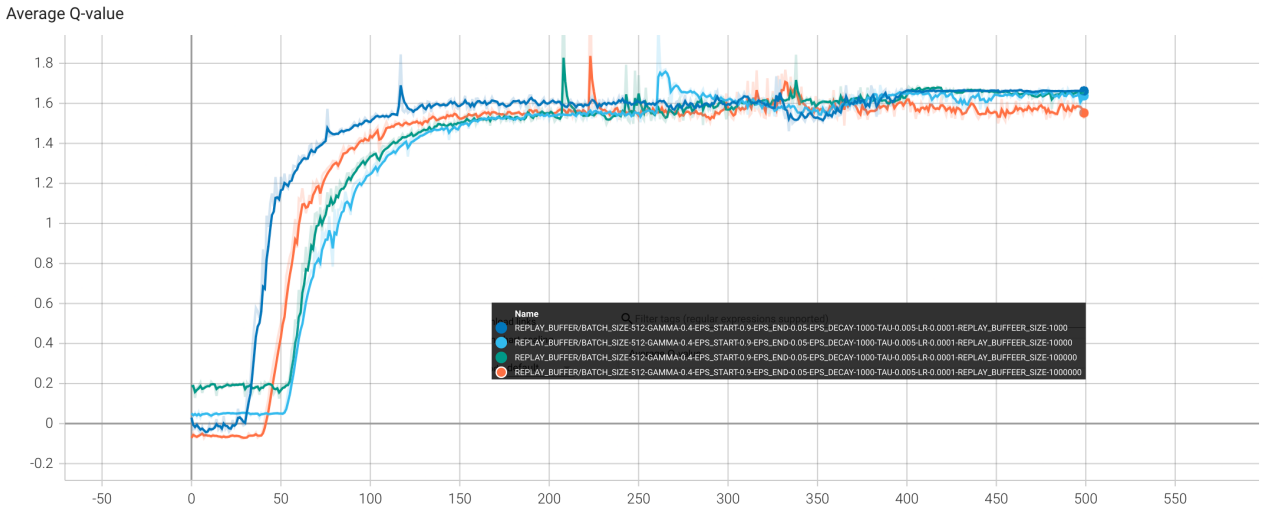

Figure 3: Replay Buffer Experiment - Average Q Value vs Number of Episodes

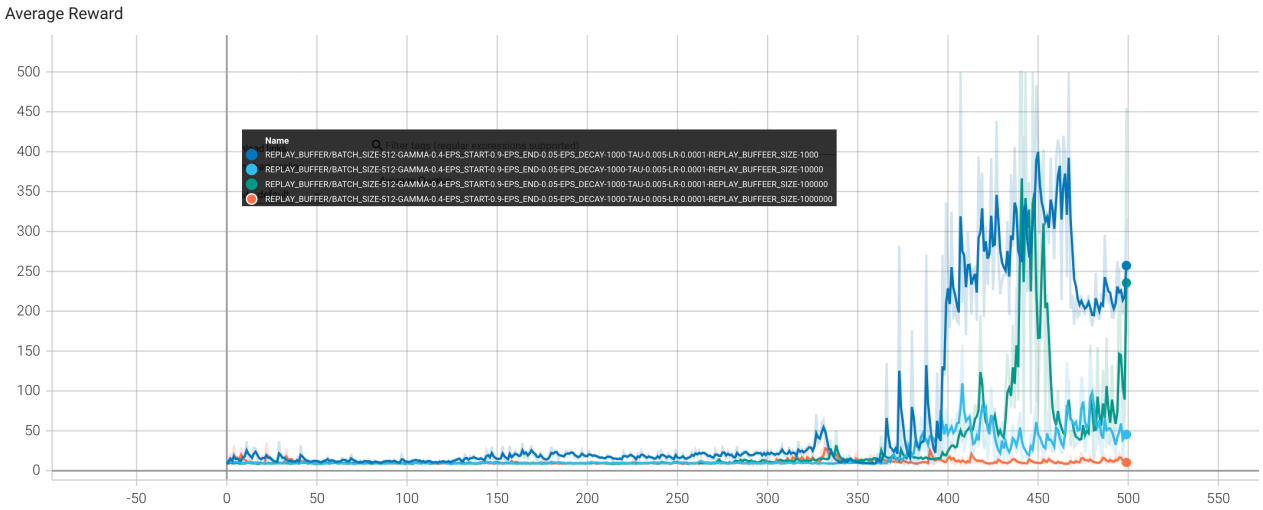

Figure 4: Replay Buffer Experiment - Average Reward vs Number of Episodes

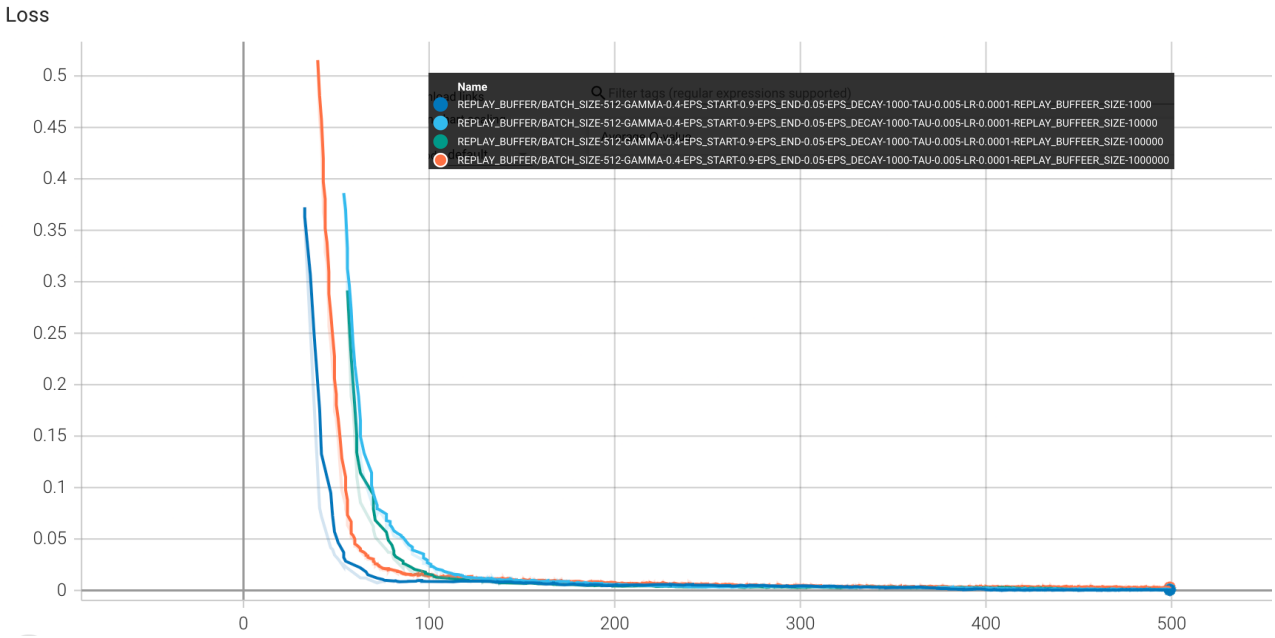

Figure 5: Replay Buffer Experiment - Loss vs Number of Episodes

We notice in Fig. 3 that the smallest Q value occurs for the largest buffer size in this case. This leads to a case of diminishing returns where there are some epoch intervals where a higher replay buffer size benefits the training process but in general in our case, it reduces. A similar trend is also observed in the case of the other two metrics - average reward and loss as seen in Fig. 4 and 5 respectively. In case of loss an intermediate size of 10000 gave the best/least loss. So there is a diminishing return trend where we believe that a better output may occur for an intermediate size.

Experiments on Batch Size

In the case of batch size we experiment with the following batch sizes - [16,32,128,256,512,1024] and observe using Fig. 6, 7, 8 that on increasing batch size, all three parameters - average q-value, average reward, loss value have a positive output i.e increases in the case of the first two parameters while loss decreases. Also, we notice that the convergence based on the plots that a larger batch size attains better values in lesser epochs. A larger batch size may provide more stable gradients and smoother updates to the neural network parameters, potentially leading to faster convergence and improved performance.

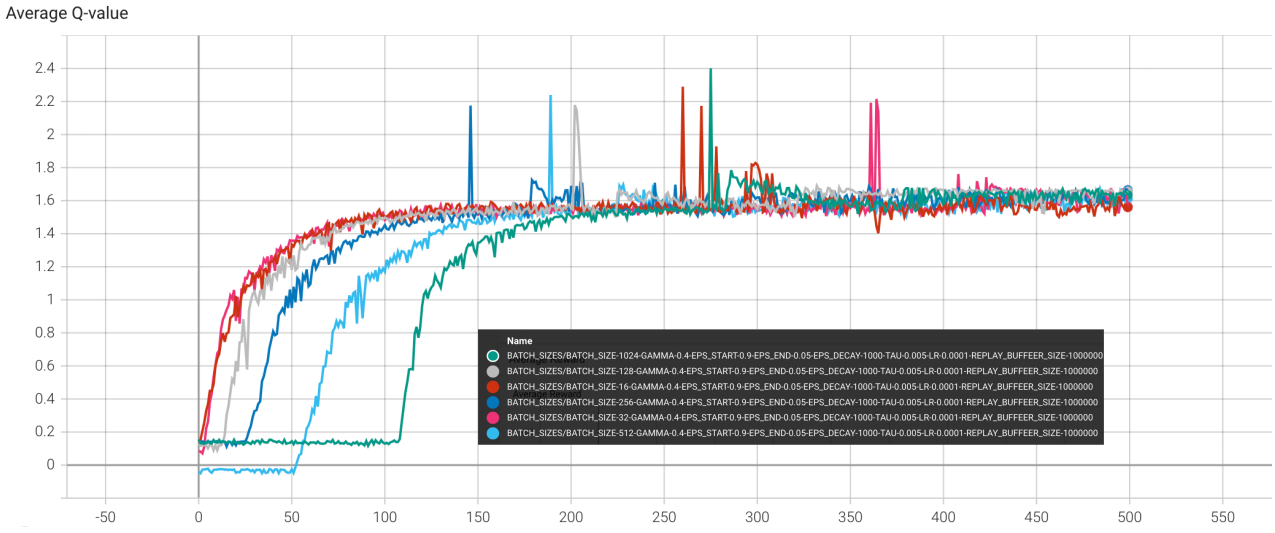

Figure 6: Batch Size Experiment - Average Q Value vs Number of Episodes

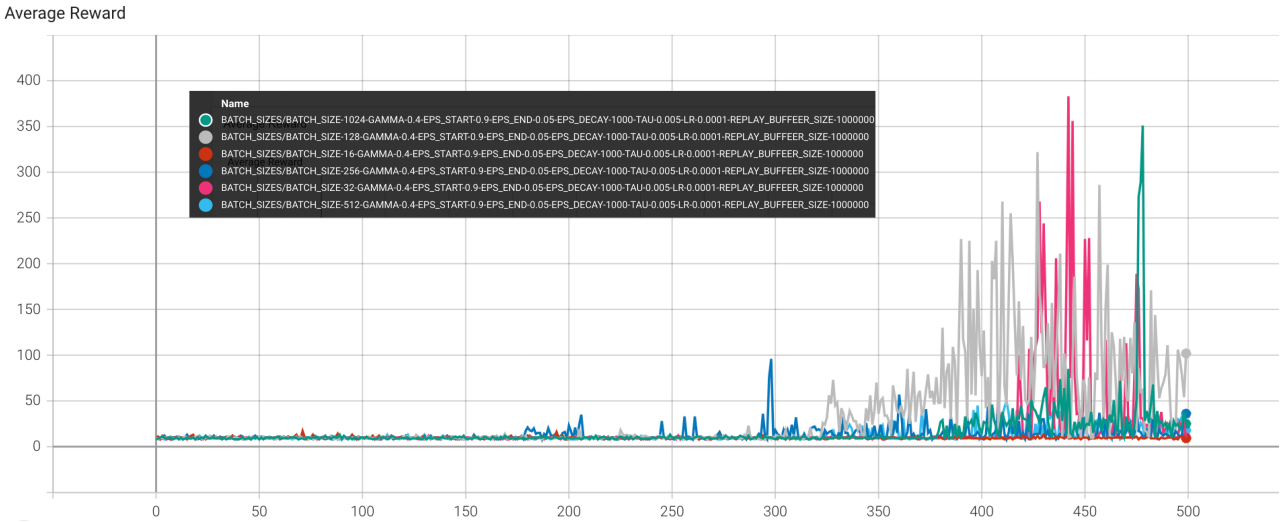

Figure 7: Batch Size Experiment - Average Reward vs Number of Episodes

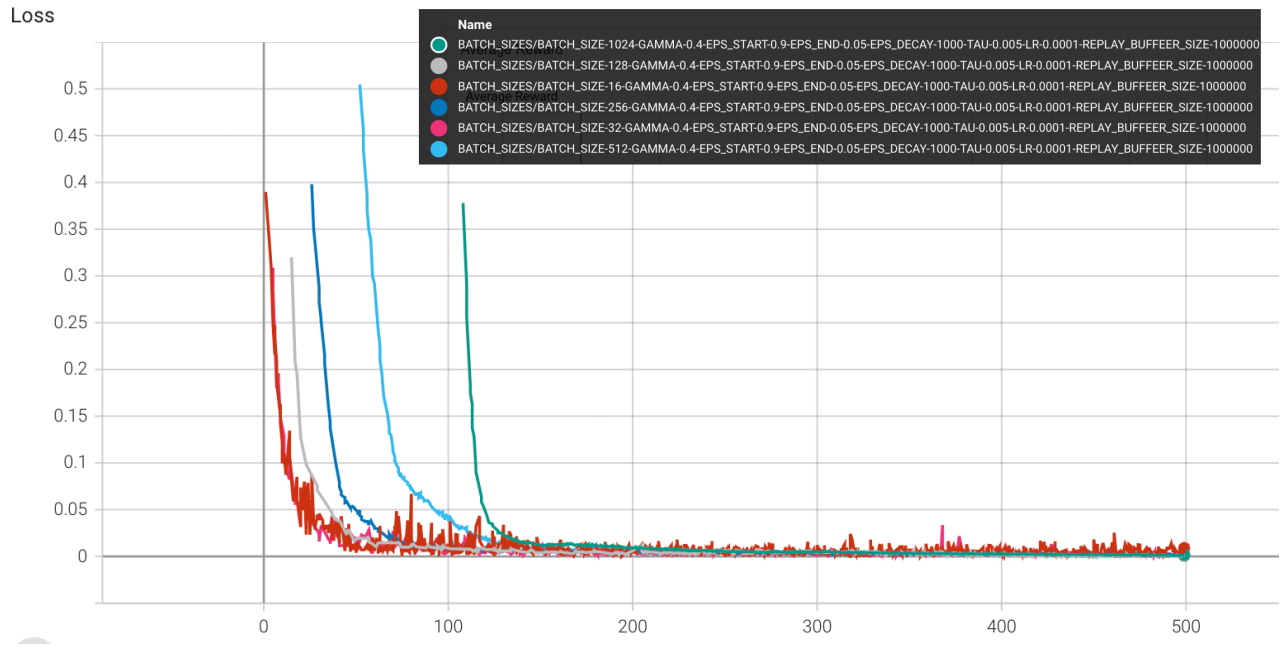

Figure 8: Batch Size Experiment - Loss vs Number of Episodes

Experiments on Gamma

In general, Gamma accounts for how much each further reward should be given importance to while computing returns. So this should generally not have a trend as such with the quantities we are computing as it should vary from problem to problem. We use the following values for gamma - [0.1,0.2,0.4,0.8,0.9,1].

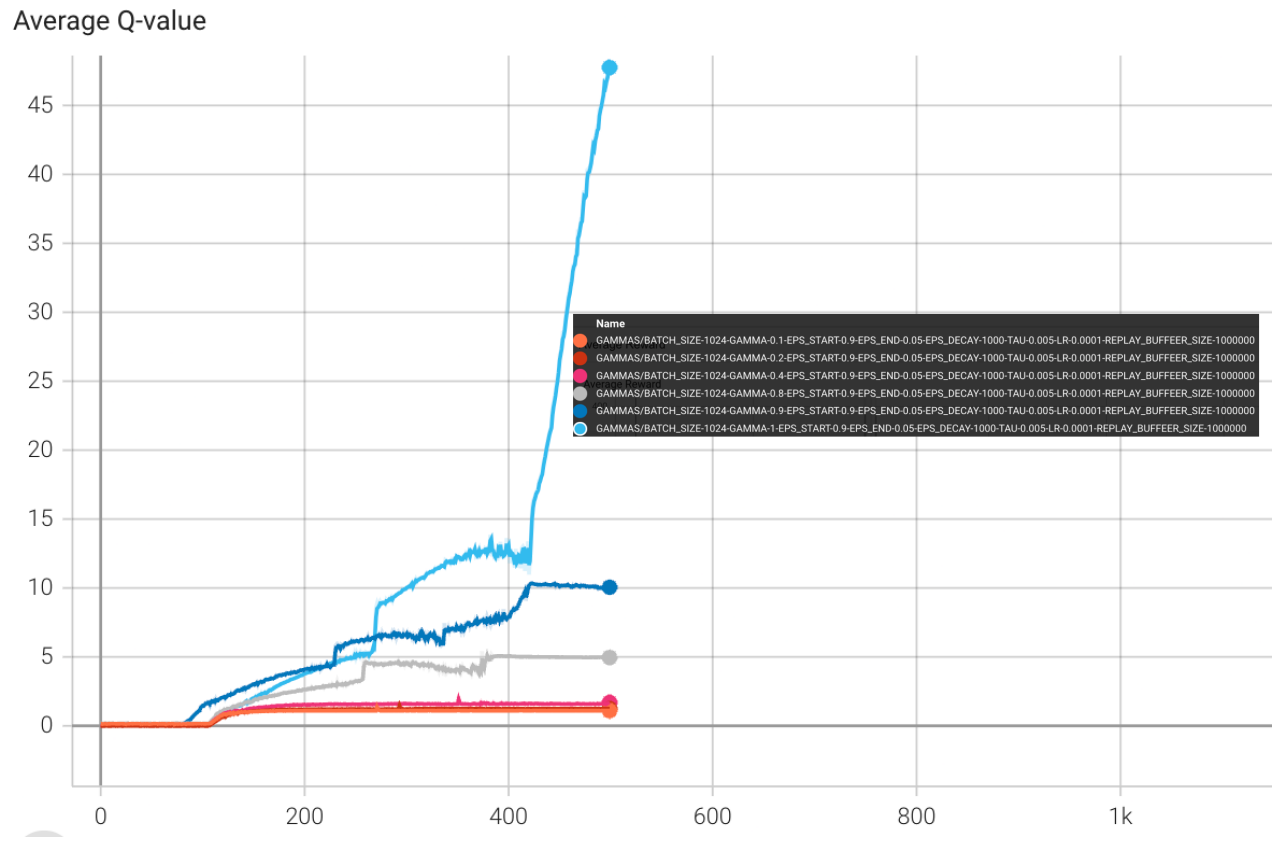

Figure 9: Gamma Experiment - Average Q Value vs Number of Episodes

We notice in Fig. 9 that the average Q value in general increases as gamma increases for the given problem however, the same is not observed for average reward where in Fig. 10 surprisingly gamma = 0.8 gives the better average reward value than all of the rest while, the loss curve as per Fig. 11 tells a different story.

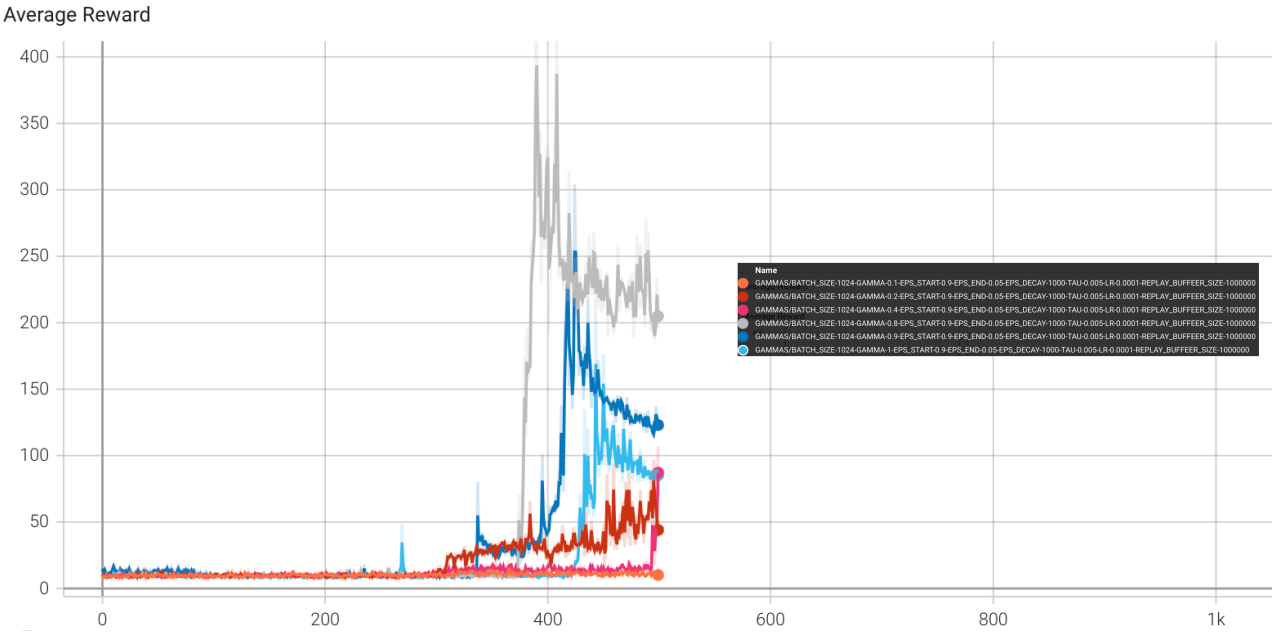

Figure 10: Gamma Experiment - Average Reward vs Number of Episodes

The loss values for gamma 0.3 and 0.4 are as expected while unstable trends can be seen for the rest of the gamma values. Gamma = 0.8 renders an expected trend but the loss values decrease noisily.

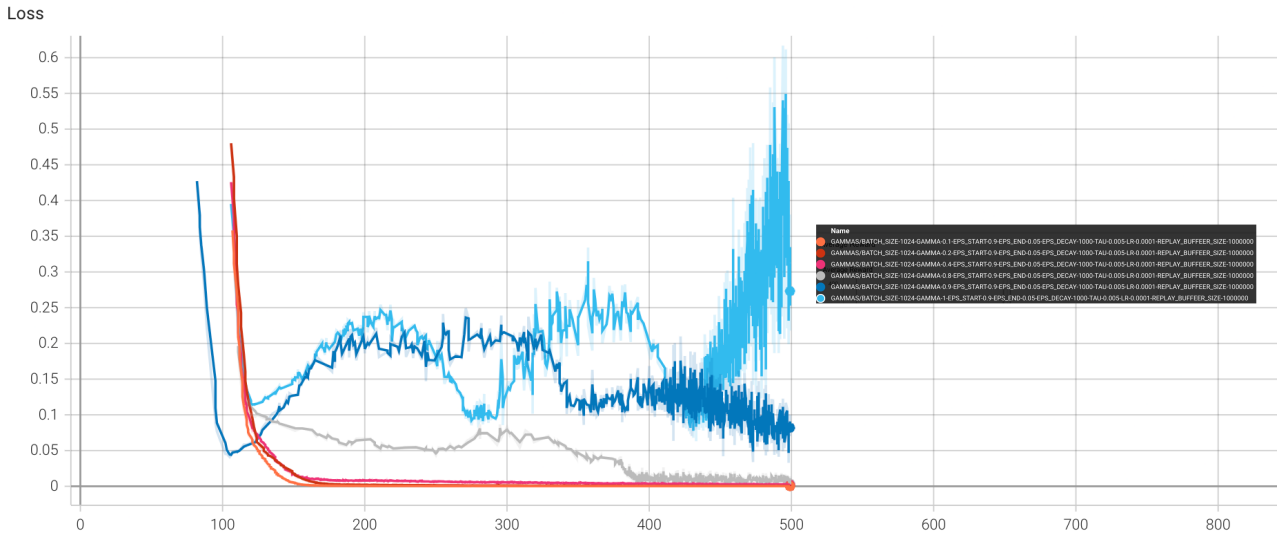

Figure 11: Gamma Experiment - Loss vs Number of Episodes

Experiments on τ - the soft weight update hyperparameter

We use the following value of τ - [0.005,0.05,0.1,0.5]. This parameter usually defines how the policy net and target net weights are updated. We notice the following trends as shown below:

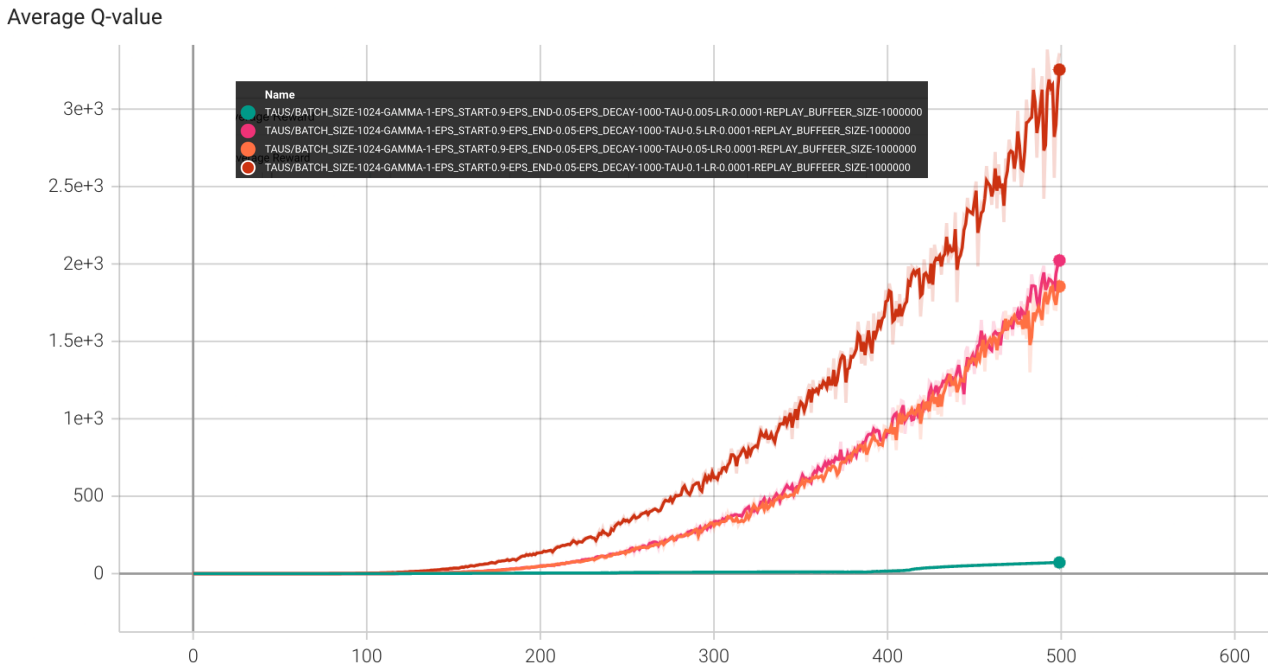

Figure 12: Tau Experiment - Average Q Value vs Number of Episodes

Figure 13: Tau Experiment - Average Reward vs Number of Episodes

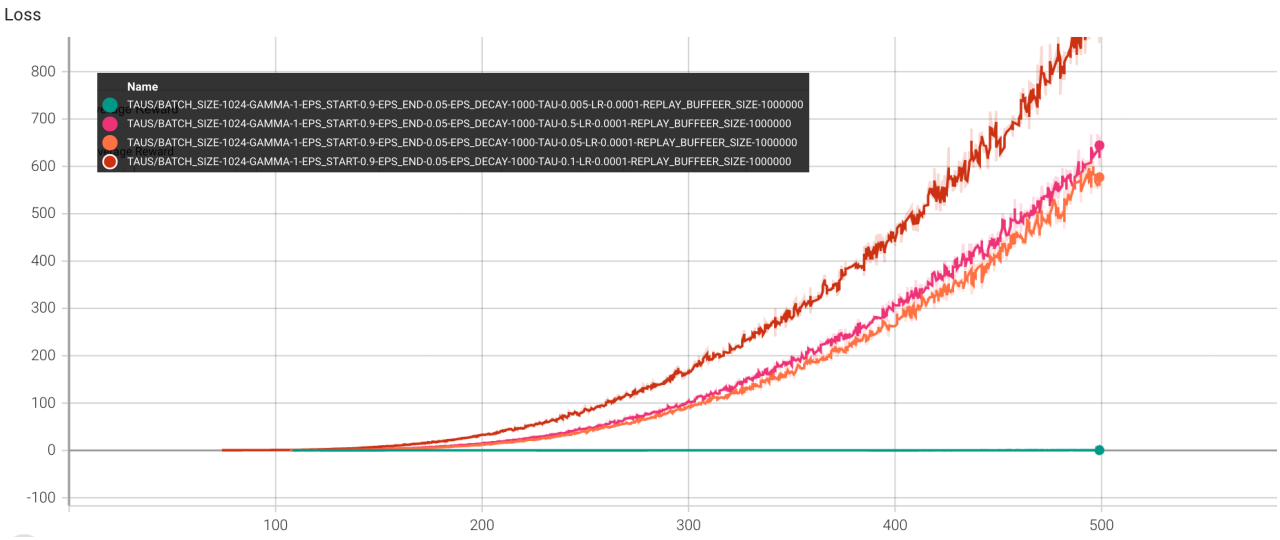

Figure 14: Tau Experiment - Loss vs Number of Episodes

The hyperparameter τ determines the rate at which the target network parameters are updated. A smaller τ implies slower updates, while a larger τ leads to faster updates. This can be also seen in Fig. 12, 13 and 14. As we grow τ the average Q value and Average Reward increases as expected (generally) but we notice that the loss suffers miserably when τ is increased from the standard value of 0.005. Therefore this confirm the hypothesis that even if convergence becomes better the loss becomes noisy and increases.

Experiments on Learning Rate

As seen in Fig. 15, 16, 17 we notice that for very low learning rate the loss remains less or is lesser for certain epochs, and increases when learning rate becomes more.

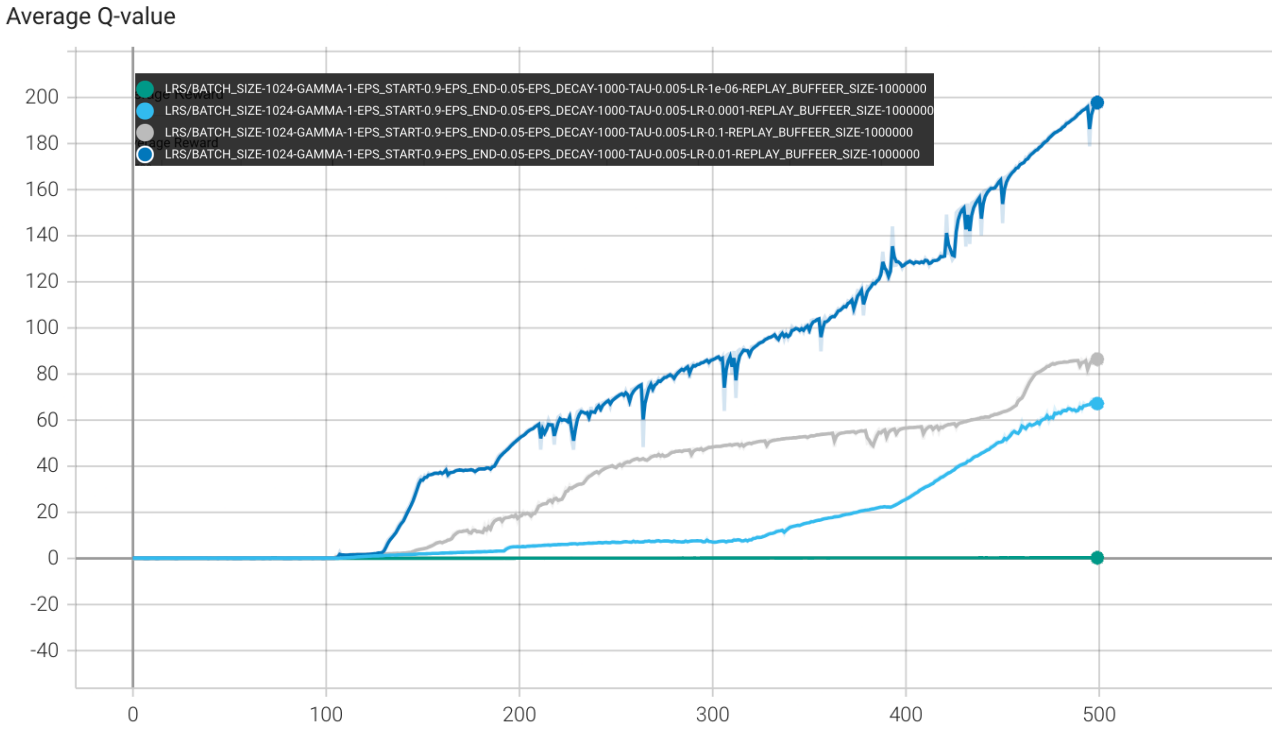

Figure 15: LR Experiment - Average Q Value vs Number of Episodes

While we notice unstable average rewards for higher learning rate values we notice a faster convergence if we decide to reduce the number of episodes.

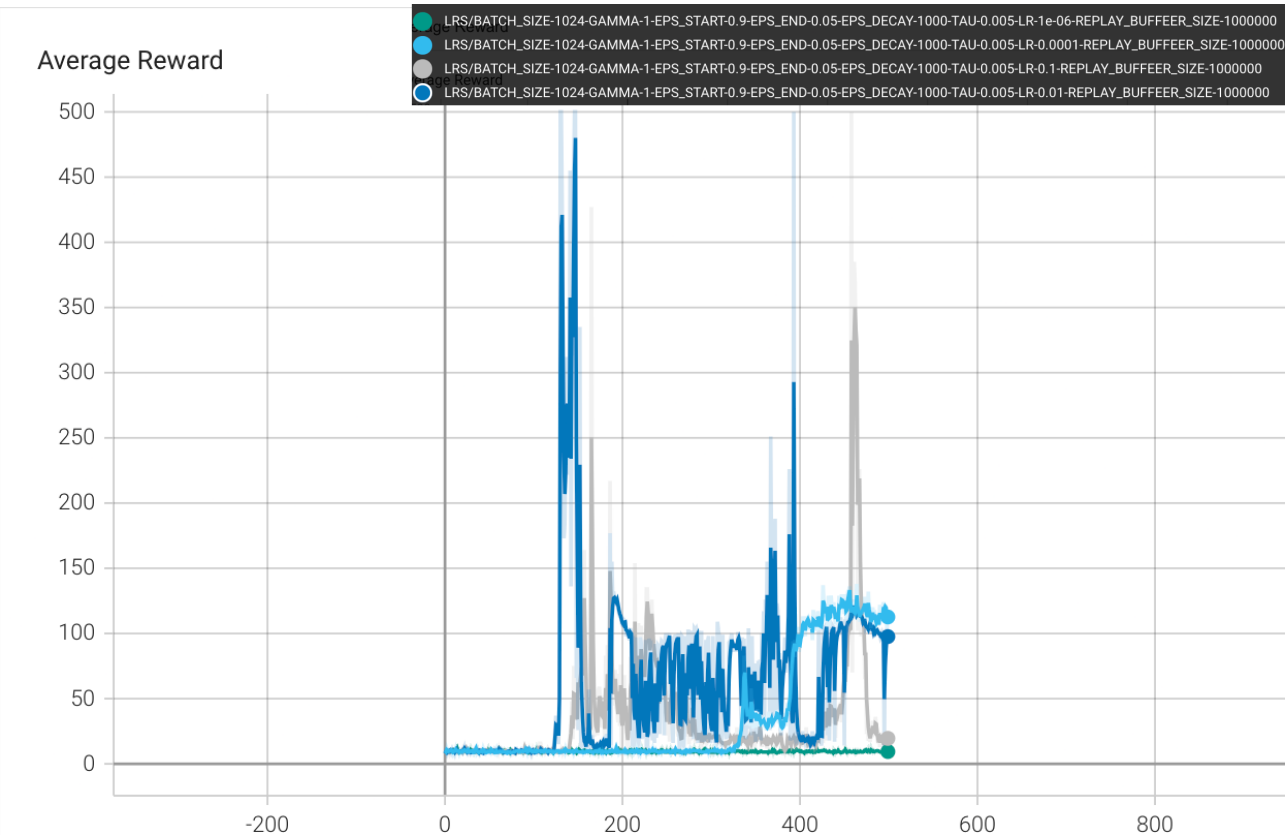

Figure 16: LR Experiment - Average Reward vs Number of Episodes

Lastly, we notice that the Q value rises in an unorderly fashion. There is no clear general trend as to what is the best trend. That is independently we cant comment what is the best learning rate and a hyperparameter search is necessary.

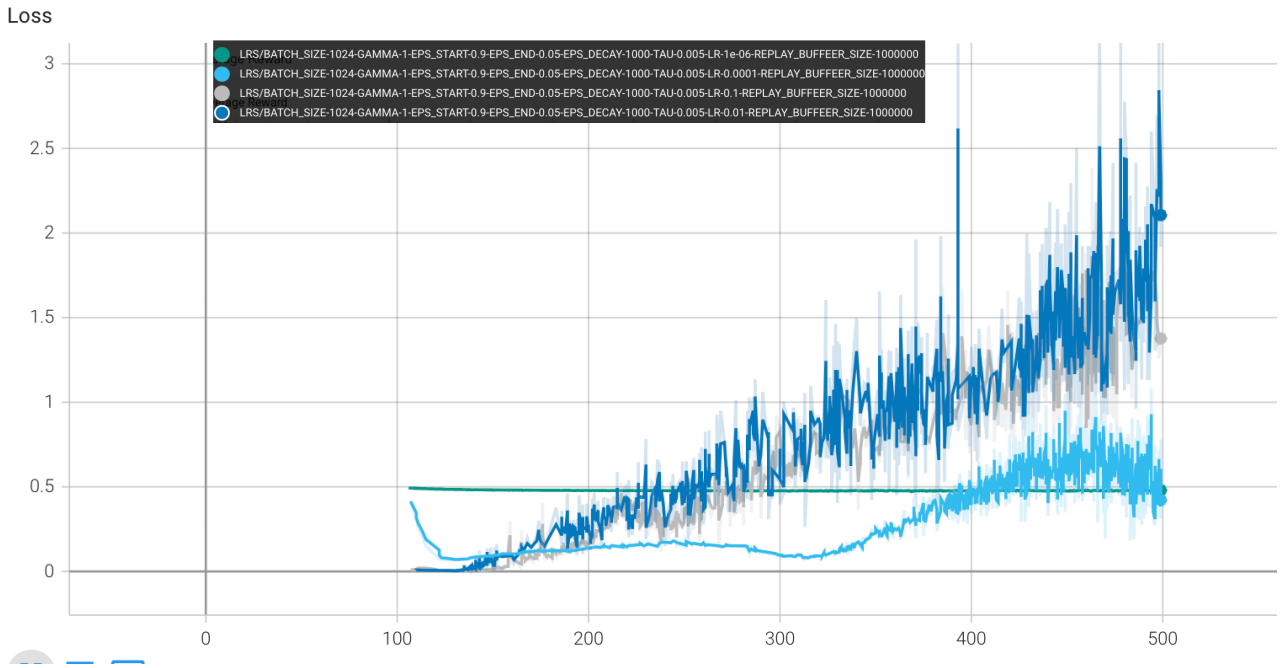

Figure 17: LR Experiment - Loss vs Number of Episodes

Part 3 - Deep Q Networks where inputs are images

We use the same cartpole definition but in place of passing a state vector of 4 entries we now have the inputs present as an RGB array image. We use the gym's render method to get the images and the use a Deep Q Net for learning an agent that takes an image and then provides an action as an output/ learns the policy based on the image shown.

Architecture for Deep Q Network

We use two variants of architecture where the major difference occurs in the type of CNN defined. The two architectures are as follows:

1. ResNet 18 as a feature extractor where the resnet architecture parameters are frozen and a similar fully connected network is defined as above is used to get the action. This architecture can be seen in Fig. 18

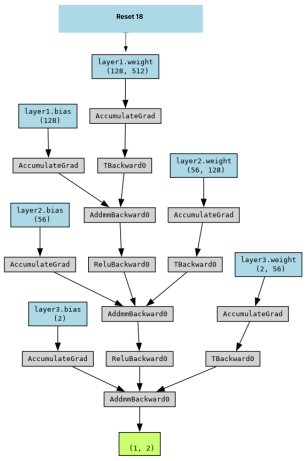

Figure 18: Resnet-based architecture for Deep RL Agent.

2. Self-defined CNN architecture where the CNN is also trained to understand frames better. This is a lighter variant and we expect that this if trained for enough epochs can perform better than resent as it understands the game scene better and is not pre-loaded with image information. This architecture can be seen in Fig. 19

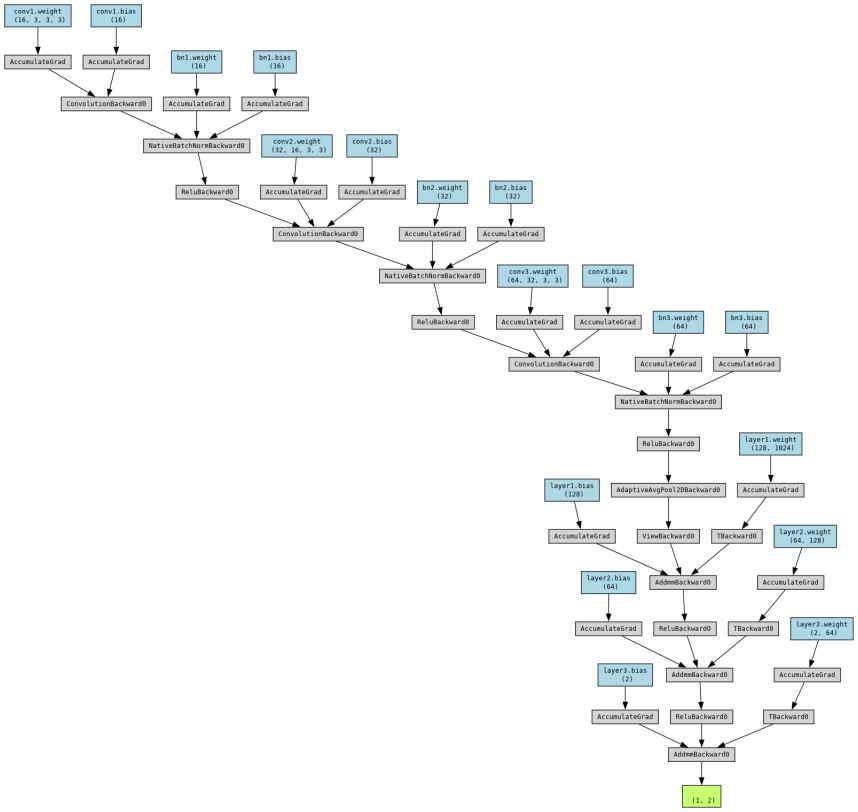

Figure 19: Our Architecture for Deep RL Agent.

Observations

We train the CNN layers as well as the fully connected layers for the agent and use similar default hyper-parameters as defined (500 episodes and batch size = 1024, memory buffer size = 100000, and the remaining parameters same as the second deliverable) in the previous problem and divide our observations for the two architectures.

Deep RL agent based on Resnet-18 feature extractor

We notice from Fig. 22 that the loss progressively reduces as we increase the number of episodes. This is in expected trends to what we have also observed in the second statement.

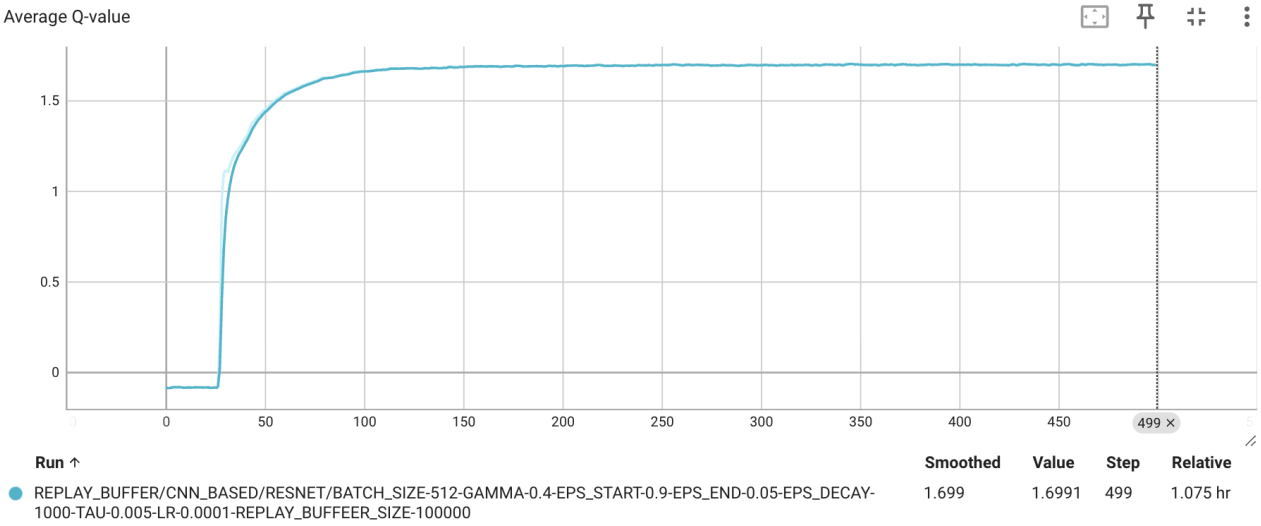

Figure 20: Resent based architecture - Average Q value vs epochs

However, in the case of Average Rewards, from Fig. 21 (and consequently Fig. 20) we notice a noisy reward output and it is not stable. Along with that we also notice that the value obtained for reward is also lesser than compared to the model performance shown for the second deliverable. We notice that the CNN based inputs are unable to provide enough state information to the network defined to do well in getting the maximum average reward/return.

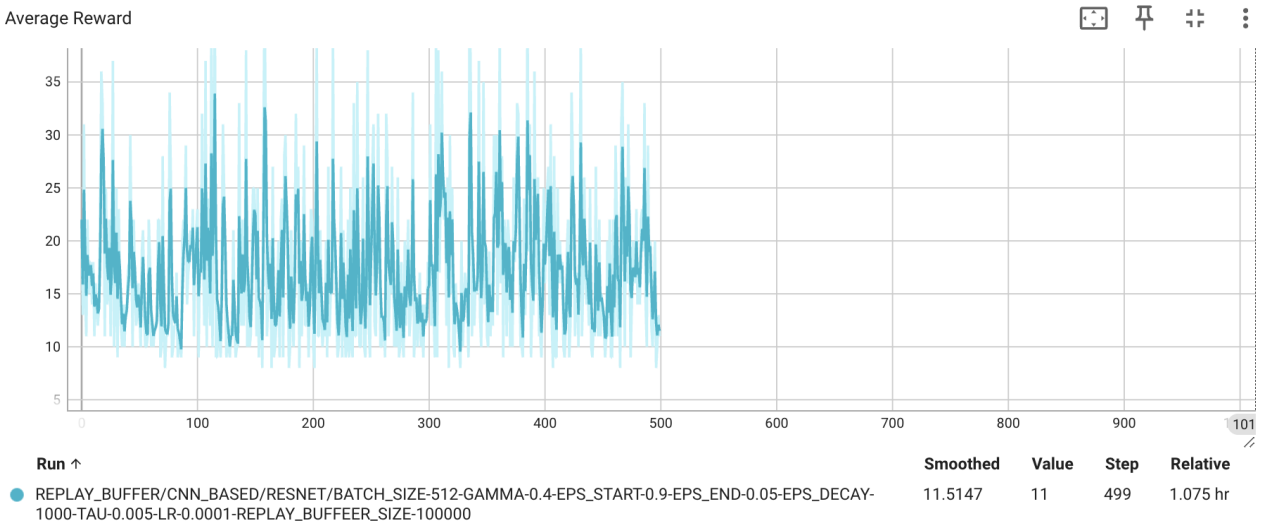

Figure 21: Resent-based architecture - Average Reward vs epochs

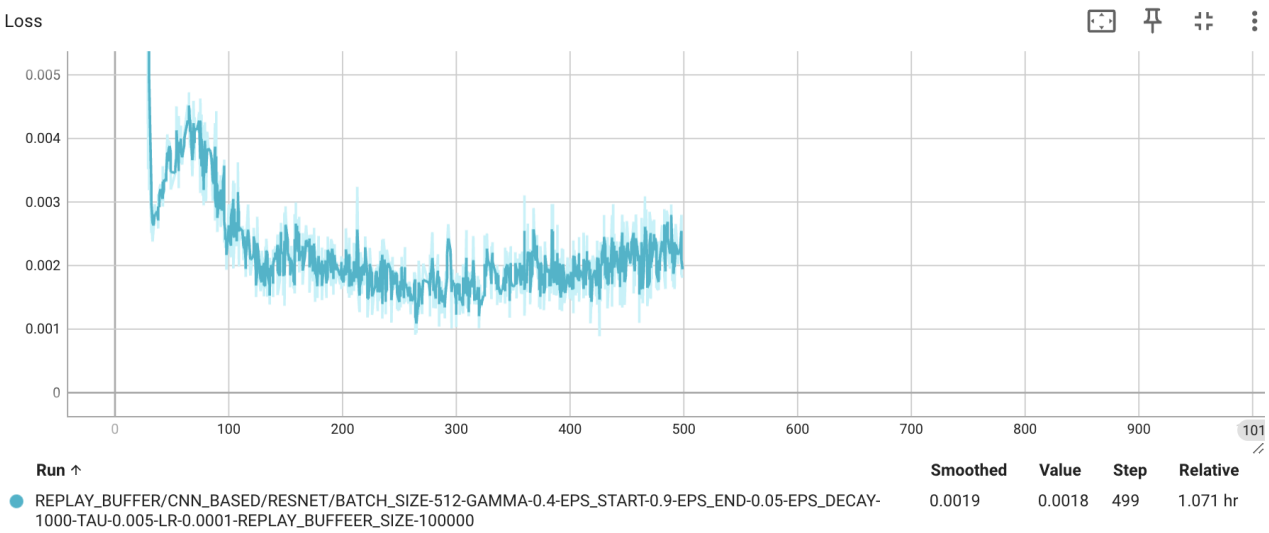

Figure 22: Resent-based architecture - Loss vs epochs

Do note that this network was trained for about 48 minutes on a Kaggle GPU (P100).

Deep RL agent based on own architecture and CNN training

We believe that training a simple CNN will firstly reduce the size of the network and also should perform better than the resnet architecture as the resnet representation is better at processing real-life objects while the simulation environment frames are mostly graphics elements drawn using a graphics engine. Therefore it is better to inherently understand the features of the out of the gym.render() method using a simpler CNN. We use the same set of hyperparameters as the one above. This network takes around 2hr 20 mins to train for 500 episodes on a Kaggle P100 GPU.

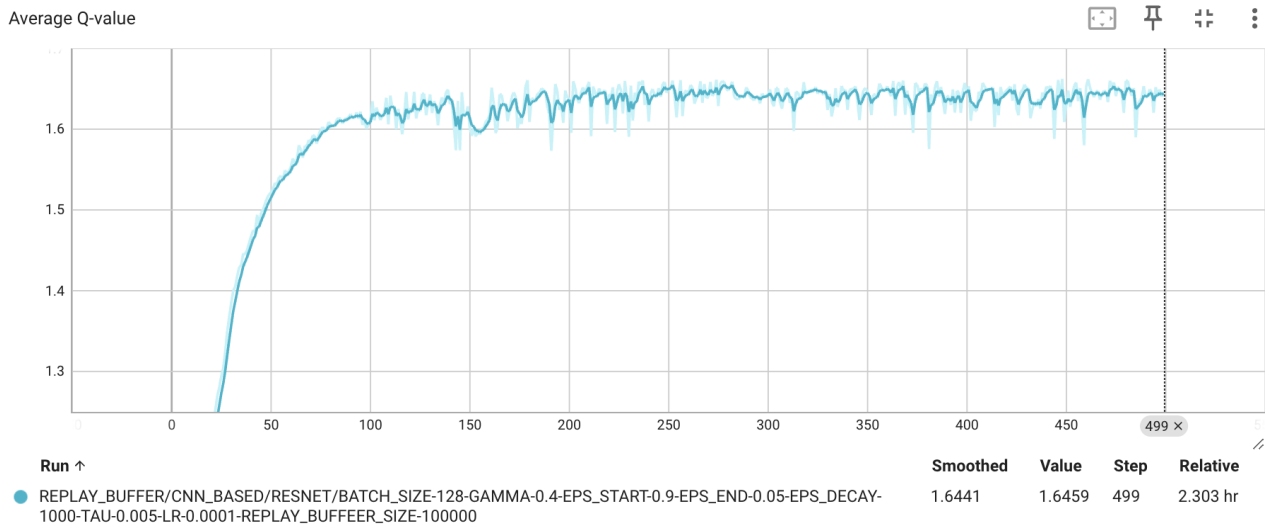

Figure 23: Our CNN-based architecture - Average Q Value vs epochs

We notice a higher average Q value as shown in Fig. 23 compared to the resnet-based architecture. However, this still doesn't compare to the previous methods tried in the second deliverable. We also notice an increase in average reward as shown in Fig. 24, however just like the average Q value we don't see it beating the simple state vectors as used in the second problem.

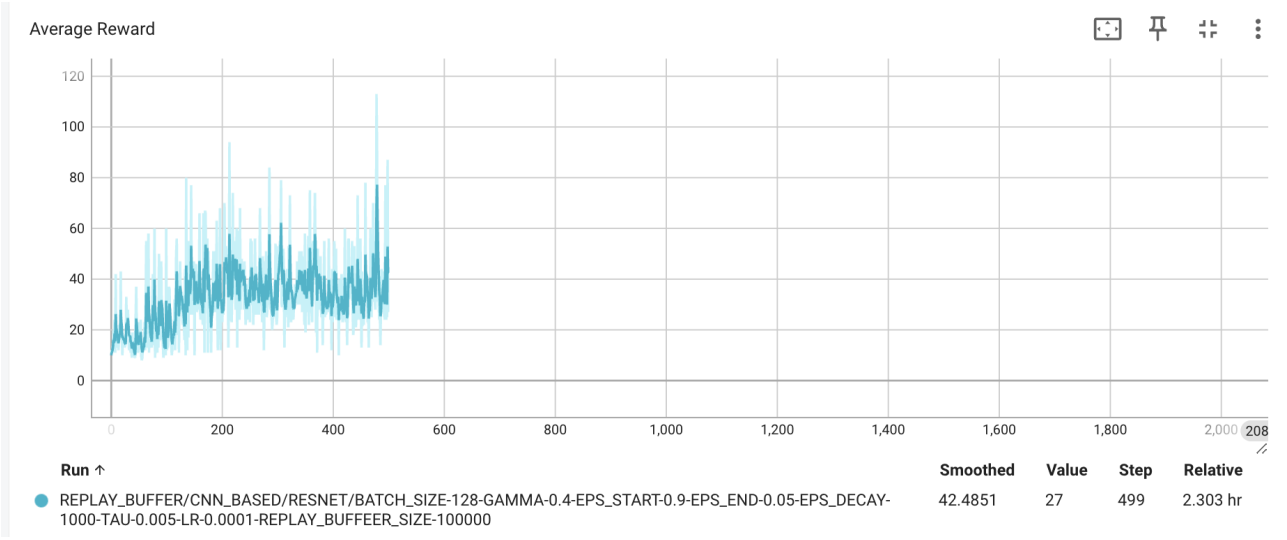

Figure 24: Our CNN-based architecture - Average Reward vs epochs

Lastly, the loss as shown in 25 looks similar in both cases and behaves similarly.

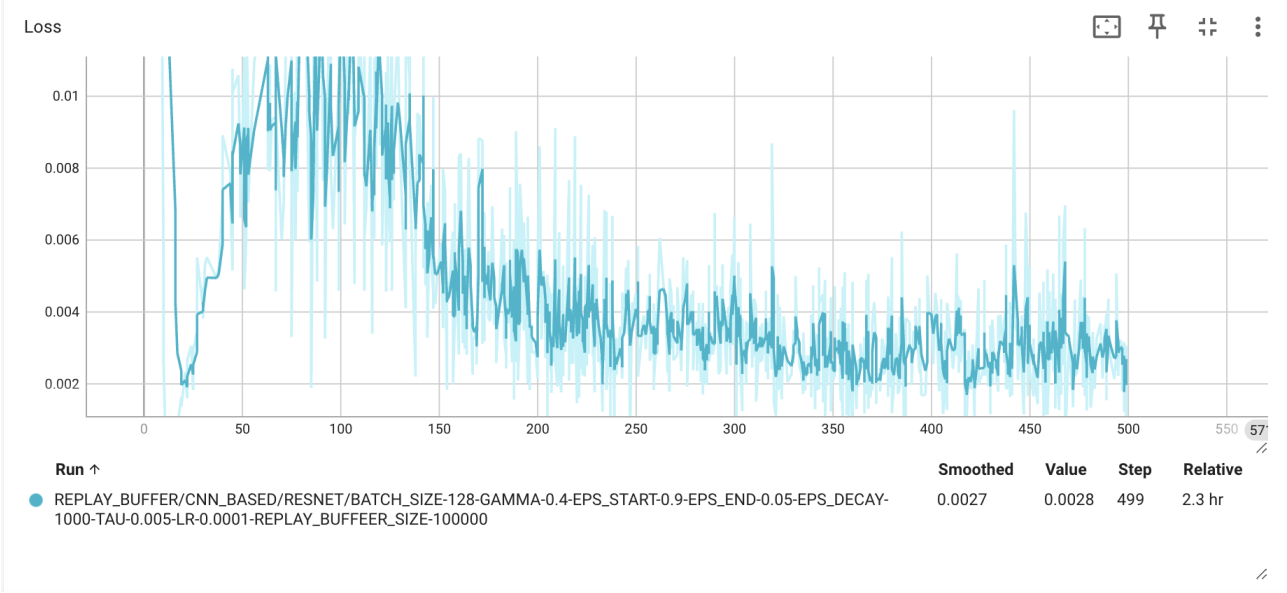

Figure 25: Our CNN-based architecture - Loss vs epochs

Extending this idea for Multiple Sequence of Images - Challenges

We notice that we don't have enough context in a single frame to denote the velocity of the cart as well as the associated angular velocity of the pole. For this we can either cache multiple frames in the replay buffer sampled at some interval. However, we noticed that even for a single image, the replay buffer memory overflowed on the free version of Kaggle and Colab and we were unable to test the intuition. To further improvise this, we could cache a single frame and compute the optical flow of the rest of the frames of the motion sample. This will reduce the memory limitations but will substantially increase training time as for each situation we need to compute the optical flow (which is mostly done as a CPU process) and is beyond the scope of our discussion. Lastly to encompass the temporal nature of the frames in the CNN itself, 3D CNNs can serve as an interesting alternative to the approaches discussed above that would cut the memory requirements.