MSc in Artificial Intelligence and Machine Learning

CS6482 – Deep Reinforcement Learning

Assignment 2: Sem2 AY 23/24 - DQN Classic Control

Pratik Verma – 23007575

Siddharth Prince – 23052058

Table of contents

1. Why Reinforcement Learning?

Reinforcement Learning (RL) is used to solve problems such as the one tackled in this report when the problem lacks any straightforward classification angle either due to the lack of labelled data for the task or the very nature of the task needing actions to be taken or interactions to be made with an environment. In other words, there is no precise solution to the problem and the model (agent) needs to produce approximations of an optimal strategy (policy).

2. The Gym Environment: MountainCar-v0

The Gym environment of choice for this assignment is MountainCar-v0. It is a classic control environment where the goal is to simple reach the flag at the top of the hill towards the right side of the sinusoidal slope. The detailed specifications for this environment are as follows [1].

Task Objective:

- The car starts at the bottom of a valley between two hills.
- The agent must learn to build up enough momentum by repeatedly moving left and right to reach the flag at the top of the right hill.

Action Space:

There are three discrete actions available:

- 0: Move left
- 1: Do nothing
- 2: Move right

Observation Space:

The state space is continuous and consists of two variables:

- Position: The position of the car (ranging from -1.2 to 0.6).
- Velocity: The velocity of the car (ranging from -0.07 to 0.07).

Rewards:

- The agent receives a reward of -1 for each time step until it reaches the goal.
- The goal is to reach the flag at the top of the hill (position >= 0.5) as quickly as possible.

Starting State:

The position of the car is assigned a uniform random value in [-0.6 , -0.4]. The starting velocity of the car is always assigned to 0.

Episode Termination:

The episode ends if either of the following happens:

- Termination: The position of the car is greater than or equal to 0.5 (the goal position on top of the right hill)
- Truncation: The length of the episode is 200.

The starting state of this environment can be seen in figure 2.1 below which shows the care at the bottom of the valley.

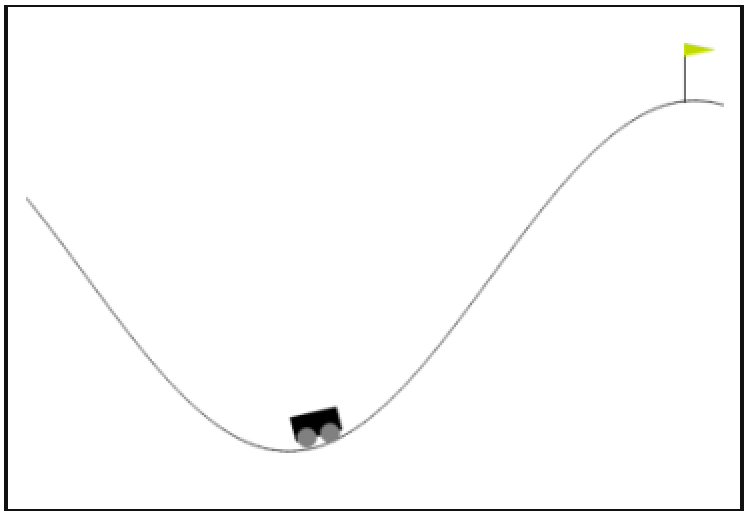


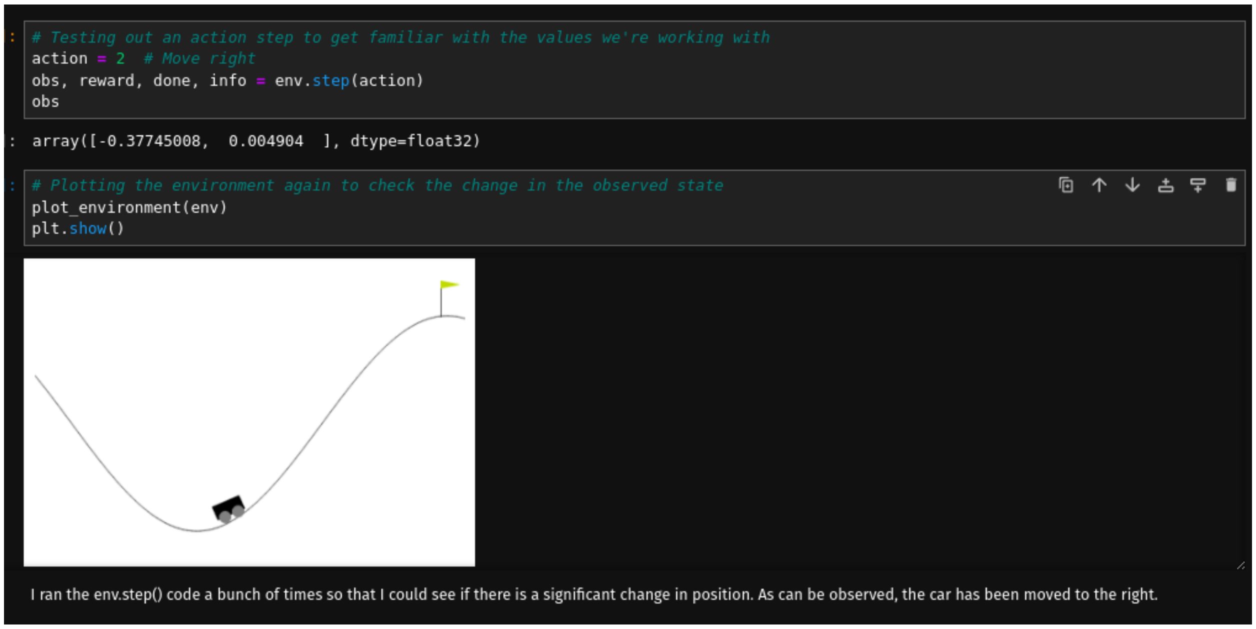
Fig 2.1: Starting state of the Mountain Car environment (seed=42)

3. Implementation

The DQN agent implementation to reliably be able to solve the problem posed for the environment is discussed in this chapter.

3.1 Capture and Sampling of Data

As discussed in the specifications above, there are two observations that can be made from a frame at a given step. They are the velocity and position along the x-axis. Reaching a position of 0.5 or higher is the end goal. Thus, the data captured for the training of the agent will consist of these two observations and their change when any of the 3 possible inputs (move left, noop, move right) are given. Figure 3.1 below shows the observed data from a frame printed out.



```
# Testing out an action step to get familiar with the values we're working with
action = 2  # Move right
obs, reward, done, info = env.step(action)
obs
```

```
array([-0.37745008,  0.004904  ], dtype=float32)
```

```
# Plotting the environment again to check the change in the observed state
plot_environment(env)
plt.show()
```

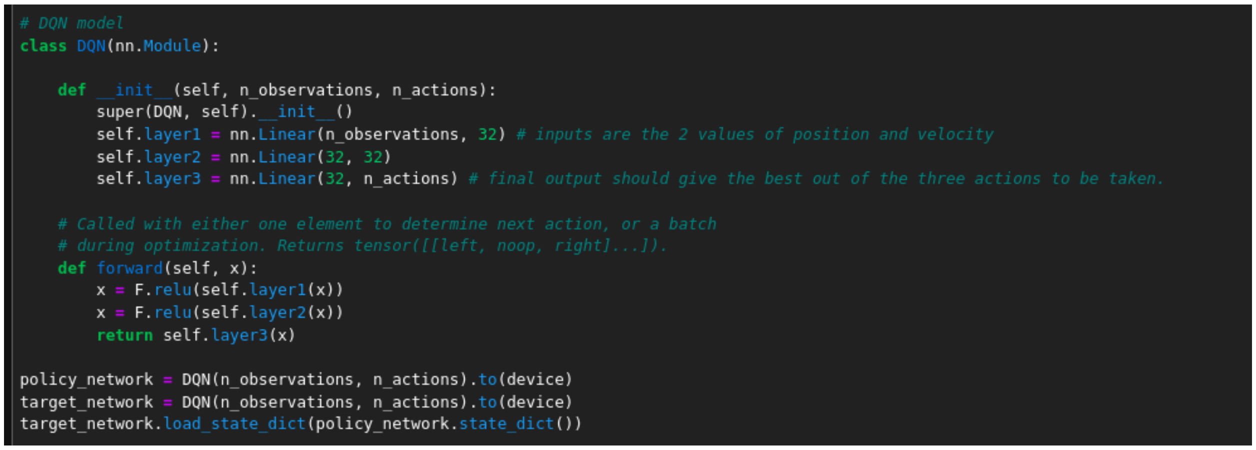
I ran the env.step() code a bunch of times so that I could see if there is a significant change in position. As can be observed, the car has been moved to the right.

Fig 3.1: Observed data and change in position of the car

Running the step multiple times of moving to the right has caused the car to move a little uphill from the initial position The *env.step(input)* line of code enables us to pass in which input to be performed and returns the corresponding observations.

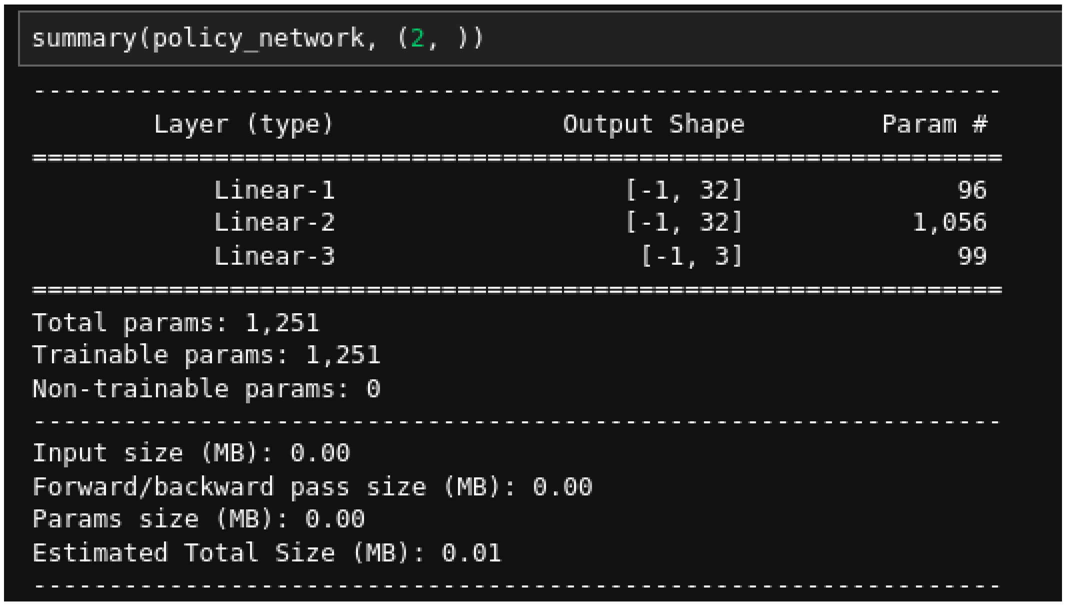
3.2 Network structure and hyperparameters

For the neural network architecture, we stuck to a simple and standard architecture that we had seen implemented in A Geron's sample notebook [2]. Figures 3.2 and 3.3 show the code and architectures summary, respectively.



```
# DQN model
class DQN(nn.Module):

    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 32) # inputs are the 2 values of position and velocity
        self.layer2 = nn.Linear(32, 32)
        self.layer3 = nn.Linear(32, n_actions) # final output should give the best out of the three actions to be taken.

    # Called with either one element to determine next action, or a batch
    # during optimization. Returns tensor([[left, noop, right]...]).
    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)

policy_network = DQN(n_observations, n_actions).to(device)
target_network = DQN(n_observations, n_actions).to(device)
target_network.load_state_dict(policy_network.state_dict())
```

Fig 3.2: DQN code implementation

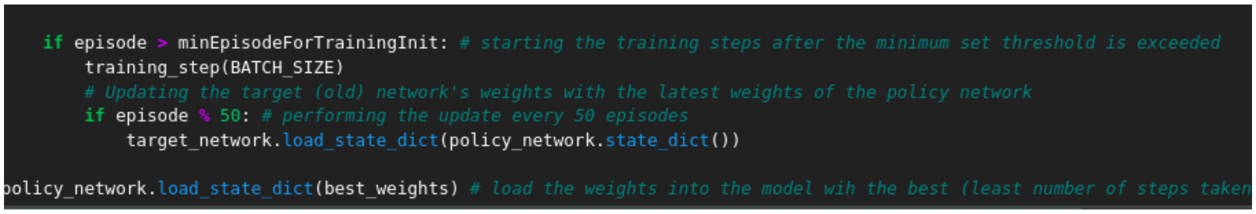


```
summary(policy_network, (2, ))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                   [-1, 32]              96
            Linear-2                   [-1, 32]           1,056
            Linear-3                    [-1, 3]              99
================================================================
Total params: 1,251
Trainable params: 1,251
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.01
----------------------------------------------------------------
```

Fig 3.3: DQN architecture summary

Also seen in fig 3.2 is the initialisation of two DQN models, the policy network and the target network. This is used to implement the double DQN concept where we use an "older" target network to compute the target value for the state while the policy network is the one with the latest weight updates which is used to predict the next action to be taken based on the current state for every step. The target network is updated with the weights of the policy network after a set number of episodes as seen in figure 3.4 below.



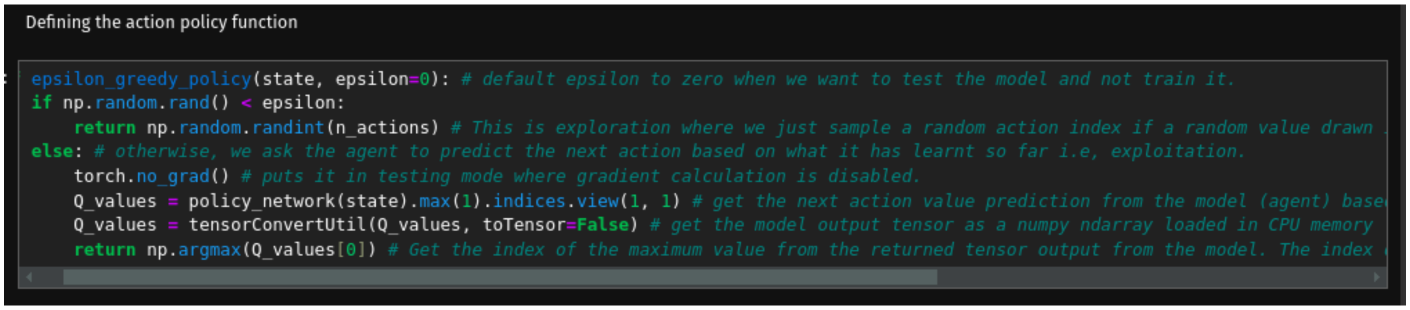
```
    if episode > minEpisodeForTrainingInit: # starting the training steps after the minimum set threshold is exceeded
        training_step(BATCH_SIZE)
        # Updating the target (old) network's weights with the latest weights of the policy network
        if episode % 50: # performing the update every 50 episodes
            target_network.load_state_dict(policy_network.state_dict())

olicy_network.load_state_dict(best_weights) # load the weights into the model wih the best (least number of steps taken
```

Fig 3.4: Code snippet of the target network's weights being updated

3.3 Q-learning updates

3.3.1 Action Policy function

The policy function determines answer to exploration vs exploitation. This condition hinges on the set Epsilon value. We are using a decaying epsilon where its value is high when the agent is in its nascent stages and does not have a minimum number of samples in the memory buffer or lacks "experience". This increases the probability of the agent sampling random actions from the action space instead of using a predicted value from the model. But as the number of episodes run increases, the epsilon value decreases till it reaches a minimum value that is set manually. At this point, the focus is more on the agent “exploiting” the learning it has done from the saved experience. The code snippet in figure 3.5 below demonstrates this.

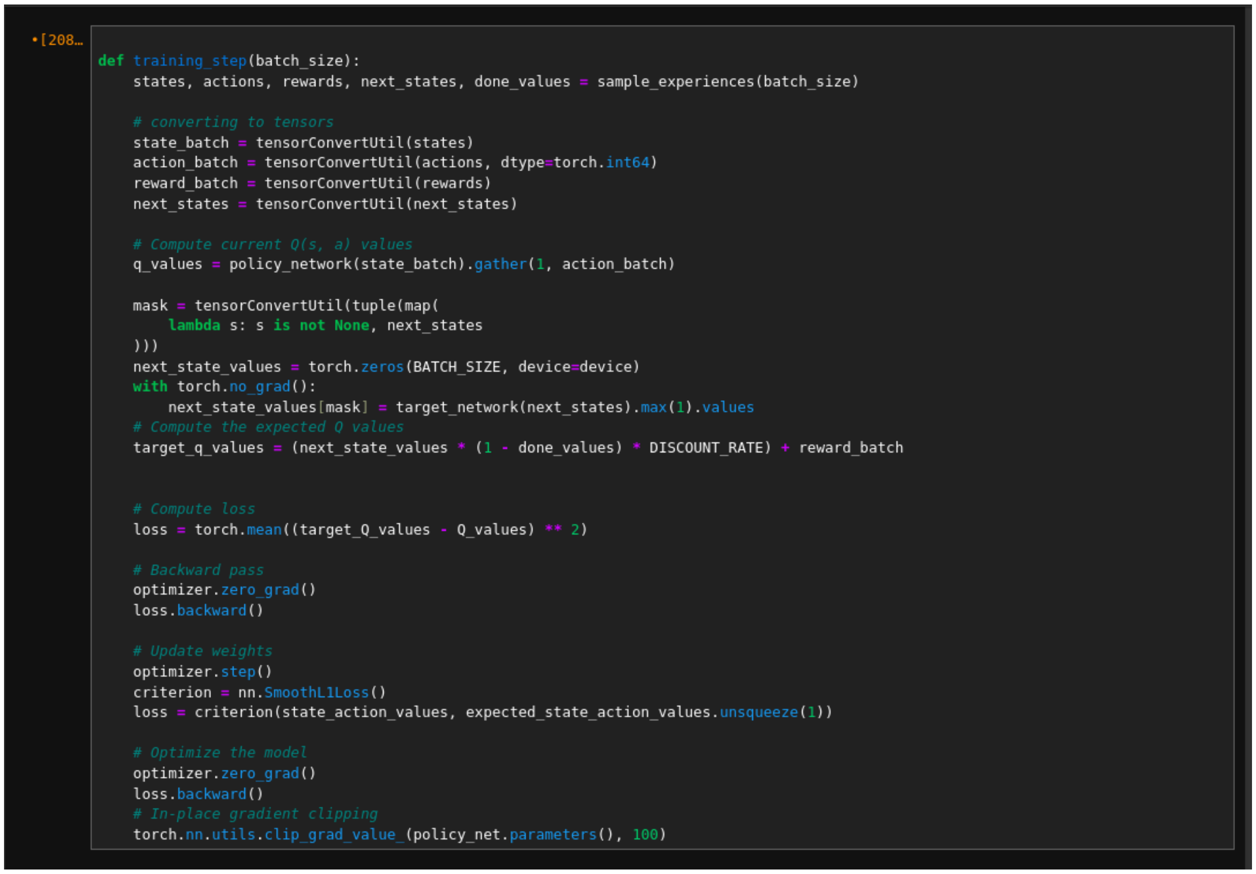

Defining the action policy function

```
epsilon_greedy_policy(state, epsilon=0): # default epsilon to zero when we want to test the model and not train it.
if np.random.rand() < epsilon:
    return np.random.randint(n_actions) # This is exploration where we just sample a random action index if a random value drawn
else: # otherwise, we ask the agent to predict the next action based on what it has learnt so far i.e, exploitation.
    torch.no_grad() # puts it in testing mode where gradient calculation is disabled.
    Q_values = policy_network(state).max(1).indices.view(1, 1) # get the next action value prediction from the model (agent) base
    Q_values = tensorConvertUtil(Q_values, toTensor=False) # get the model output tensor as a numpy ndarray loaded in CPU memory
    return np.argmax(Q_values[0]) # Get the index of the maximum value from the returned tensor output from the model. The index
```

Fig 3.5: Epsilon Greedy Policy definition

3.3.2 The Training Step

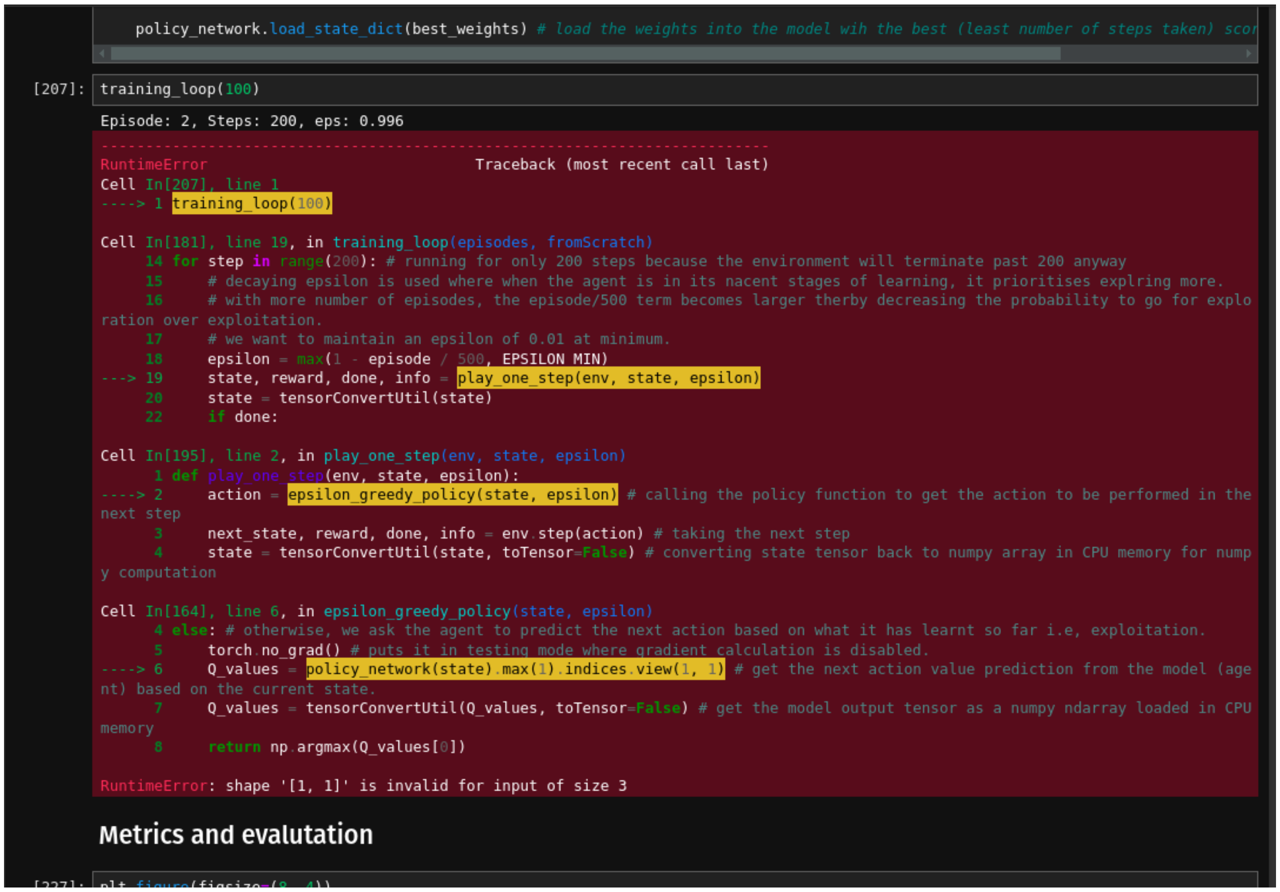
This function is responsible for getting the RL DQN agent trained by calculating the target state via the target network as discussed in section 3.2 and the loss between the predicted action from the current state and the target action computed via the target network. The code snippet in figure 3.6 shows this.


•[208…

```
def training_step(batch_size):
    states, actions, rewards, next_states, done_values = sample_experiences(batch_size)

    # converting to tensors
    state_batch = tensorConvertUtil(states)
    action_batch = tensorConvertUtil(actions, dtype=torch.int64)
    reward_batch = tensorConvertUtil(rewards)
    next_states = tensorConvertUtil(next_states)

    # Compute current Q(s, a) values
    q_values = policy_network(state_batch).gather(1, action_batch)

    mask = tensorConvertUtil(tuple(map(
        lambda s: s is not None, next_states
    )))
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
        next_state_values[mask] = target_network(next_states).max(1).values
    # Compute the expected Q values
    target_q_values = (next_state_values * (1 - done_values) * DISCOUNT_RATE) + reward_batch


    # Compute loss
    loss = torch.mean((target_Q_values - Q_values) ** 2)

    # Backward pass
    optimizer.zero_grad()
    loss.backward()

    # Update weights
    optimizer.step()
    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    # In-place gradient clipping
    torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
```

Fig 3.6: Code snippet of training_step function

However, we ran in multiple errors trying to get this to work with Pytorch. We wanted to use Tensorflow initially but ran into issues with TF not recognising the local GPU. We already had PyTorch setup and hence wanted to give it a shot.



```
    policy_network.load_state_dict(best_weights) # load the weights into the model wih the best (least number of steps taken) scor

[207]: training_loop(100)

Episode: 2, Steps: 200, eps: 0.996
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[207], line 1
----> 1 training_loop(100)

Cell In[181], line 19, in training_loop(episodes, fromScratch)
     14 for step in range(200): # running for only 200 steps because the environment will terminate past 200 anyway
     15     # decaying epsilon is used where when the agent is in its nacent stages of learning, it prioritises explring more.
     16     # with more number of episodes, the episode/500 term becomes larger therby decreasing the probability to go for explo
ration over exploitation.
     17     # we want to maintain an epsilon of 0.01 at minimum.
     18     epsilon = max(1 - episode / 500, EPSILON_MIN)
---> 19     state, reward, done, info = play_one_step(env, state, epsilon)
     20     state = tensorConvertUtil(state)
     22     if done:

Cell In[195], line 2, in play_one_step(env, state, epsilon)
      1 def play_one_step(env, state, epsilon):
----> 2     action = epsilon_greedy_policy(state, epsilon) # calling the policy function to get the action to be performed in the
next step
      3     next_state, reward, done, info = env.step(action) # taking the next step
      4     state = tensorConvertUtil(state, toTensor=False) # converting state tensor back to numpy array in CPU memory for nump
y computation

Cell In[164], line 6, in epsilon_greedy_policy(state, epsilon)
      4 else: # otherwise, we ask the agent to predict the next action based on what it has learnt so far i.e, exploitation.
      5     torch.no_grad() # puts it in testing mode where gradient calculation is disabled.
----> 6     Q_values = policy_network(state).max(1).indices.view(1, 1) # get the next action value prediction from the model (age
nt) based on the current state.
      7     Q_values = tensorConvertUtil(Q_values, toTensor=False) # get the model output tensor as a numpy ndarray loaded in CPU
memory
      8     return np.argmax(Q_values[0])

RuntimeError: shape '[1, 1]' is invalid for input of size 3
```

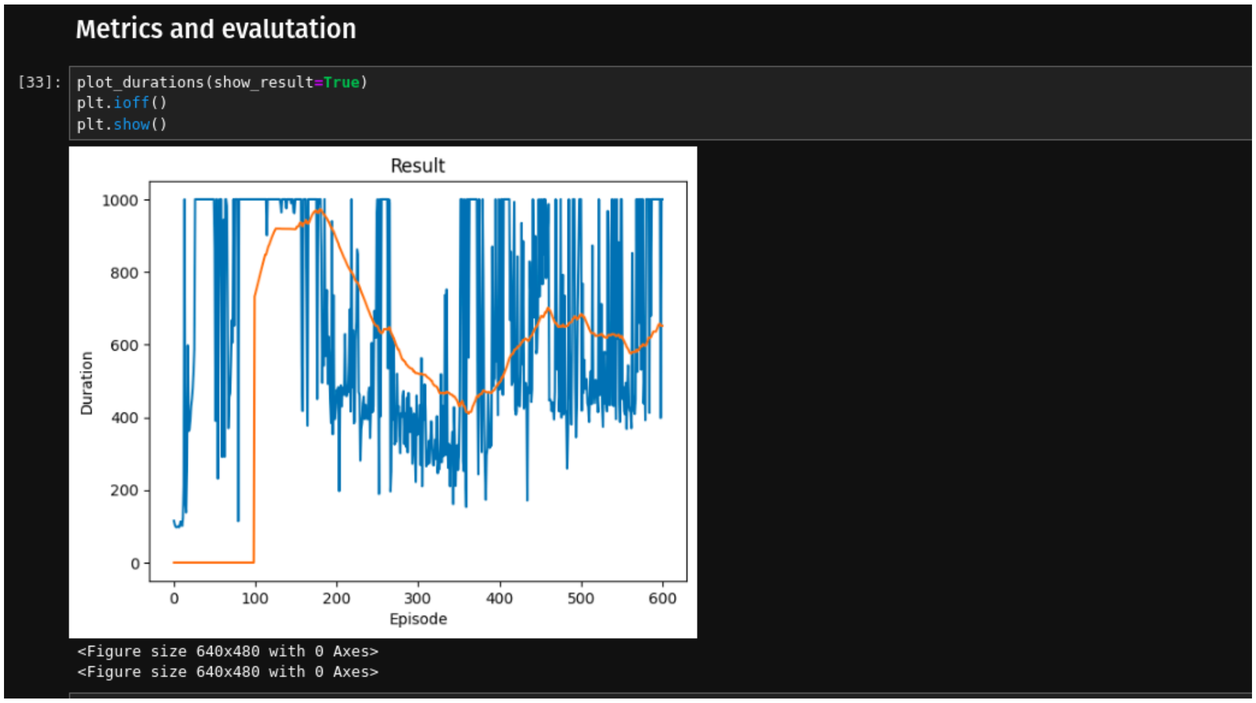
Metrics and evalutation

```
[227]: plt.figure(figsize=(8, 4))
```

Fig 3.7: Error example

4. Results

We had initially tried the Lunar Lander environment because we had not realised that it was technically not a classic control environment. We did use the code reference from PyTorch's own tutorial to implement a double DQN RL agent [3] and got results as shown below in figure 4.1.



Metrics and evalutation

```
[33]: plot_durations(show_result=True)
      plt.ioff()
      plt.show()
```

```
<Figure size 640x480 with 0 Axes>
<Figure size 640x480 with 0 Axes>
```

Fig 4.1: Result metrics for LunarLander environment

However, we had then pivoted to the Mountain Car environment when we realised our mistake. We ended up not being able to complete either to the standard that is expected. The code that is used has been understood for the most part and extensively commented in the Mountain Car implementation.

5. Experimentation

6. References

1. The Gymnasium documentation for Mountain Car environment: https://gymnasium.farama.org/environments/classic_control/mountain_car/
2. A Geron's ML notebook for Reinforcement Learning: https://github.com/ageron/handson-ml2/blob/master/18_reinforcement_learning.ipynb
3. PyTorch tutorial for DQN: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
4. GitHub repository containing RL implementation of DQN with PyTorch: https://github.com/lazavgeridis/LunarLander-v2/blob/main/train.py
5. Efficient memory-based learning for robot control, Andrew William Moore; University of Cambridge Technincal Report – November 1990: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-209.pdf