ROBOT LEARNING AND VISION FOR NAVIGATION (Homework 3)
By Kathakoli Sengupta(U17696524)

3.1. Open AI Gym Installation:

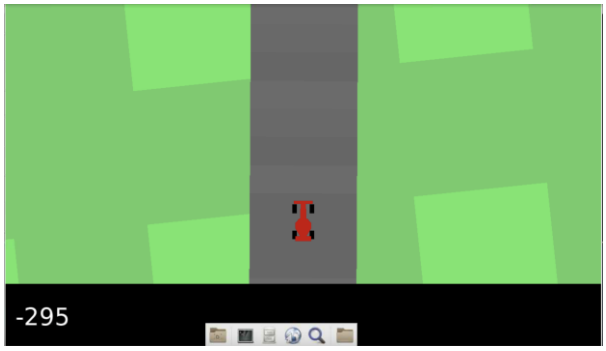

Figure 1

Fig. 1. Shows a typical game window for Car_racing.py.

3.2. Base Implementation:

a. Deep Q Learning:

```
class DQN(nn.Module):
    def __init__(self, action_size, device):
        """ Create Q-network
        Parameters
        ----------
        action_size: int
            number of actions
        device: torch.device
            device on which to the model will be allocated
        """
        super().__init__()

        self.device = device
        self.action_size = action_size

        # TODO: Create network
        #Convolution Layers
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=8, stride=4)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1)

        # input size of the first fully connected layer
        conv_output_size = self._get_conv_output_size()

        #Linear Layers
        self.fc1 = nn.Linear(in_features=conv_output_size, out_features=512)
        self.fc2 = nn.Linear(in_features=512, out_features=action_size)

    def _get_conv_output_size(self):
        """Compute the output size of the convolutional layers"""
        with torch.no_grad():
            dummy_input = torch.zeros(3*96*96).reshape(1,3,96,96)# Dummy input for a frame
            dummy_output = self.conv3(self.conv2(self.conv1(dummy_input)))
            return int(np.prod(dummy_output.size()))#Dummy output size
```

Figure 2

```
def forward(self, observation):
    """ Forward pass to compute Q-values
    Parameters
    ----------
    observation: np.array
        array of state(s)
    Returns
    ----------
    torch.Tensor
        Q-values
    """
    x = F.relu(self.conv1(observation.to(self.device)))
    x = F.relu(self.conv2(x))
    x = F.relu(self.conv3(x))
    x = x.view(x.size(0), -1)
    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    return x
```

Figure 3

Fig. 2. shows the model architecture and Fig. 3. shows the forward pass for the same. The architecture has 3 convolutional layers with 8, 4, 3 kernel sizes and 4, 2, 1 strides respectively, followed by 2 linear layers with 4096*512 weights and 512*action size weights respectively. The action size is decided by the defined action space.

i. **Will it be a problem if we only trained our network on a crop of the game pixel:** The Markov assumption states that "The state captures all the relevant information from the history." Cropping the images can result in removing some important visual cues and removing the status bar and sensor values will result in missing information related to speed, steering, lap time, gas, brake, gear etc. Hence the agent will not have knowledge of the history of agents interaction with the environment, the relevant information to predict the future states and can fail to make optimal decisions in that case.

ii. **Stack of frames not required for Car Racing:** In case of Atari games, the only information available is the position of the player and hence we need temporal data by stacking frames to give the agent sense of how the game dynamics works. However, in case of car racing we already have current speed, angle of orientation, acceleration, brake etc values other than the position image to provide the game dynamics to the agent. Hence, we don't need temporal data in case of Car Racing.

b. Fig 4. Shows the perform_qlearning_step function performing a single Q-learning update step and Fig. 5. Shows the update_target_net function performing the target net update.

```
# TODO: Update target network
target_net.load_state_dict(policy_net.state_dict())
```

Figure 5

i. **Why do we utilize fixed targets with a separate policy and target network:** In case of fixed targets, the target network generates target Q values for the loss function during training, while the policy network is updated to minimize the difference between estimated and target Q values. The use of fixed targets helps to avoid instability that might occur during training due to correlation between target and estimated Q values. The less frequent update of the target network than policy network leads to a more stable target for Q value estimates to maintain a more stable target during training. This makes the network to converge to an optimal policy more quickly.

```
# Step 1: Sample transitions from replay_buffer
state, action, reward,next_state, done = replay_buffer.sample(batch_size)


# Convert batch to tensors
state_batch = torch.tensor(state, device=device, dtype=torch.float32)
action_batch = torch.tensor(action, device=device, dtype=torch.long)
reward_batch = torch.tensor(reward, device=device, dtype=torch.float32)
next_state_batch = torch.tensor(next_state, device=device, dtype=torch.float32)
done_batch = torch.tensor(done, device=device, dtype=torch.uint8)
#print(state_batch.shape)
#print(next_state_batch.shape)

# Step 2: Compute Q(s_t, a)
q_values = policy_net(state_batch.reshape(batch_size,3,96,96)).gather(1, action_batch.unsqueeze(1))
#print(q_values.shape,"Q")

# Step 3: Compute max_a Q(s_{t+1}, a) for all next states.
with torch.no_grad():
    #print(next_state_batch[0].shape)
    next_q_values = target_net(next_state_batch.reshape(batch_size,3,96,96)).max(1)[0].unsqueeze(1)
    # Step 4: Mask next state values where episodes have terminated
    #print(next_q_values.shape,"Next")
    masked_next_q_values = (1 - done_batch.reshape(32,1)) * next_q_values
    #print(done_batch.shape)
    #print(masked_next_q_values.shape)
# Step 5: Compute the target
targets = reward_batch.reshape(32,1) + gamma * masked_next_q_values
#print(targets.shape)

# Step 6: Compute the loss
loss = F.mse_loss(q_values.to(device), targets.to(device))

# Step 7: Calculate the gradients
optimizer.zero_grad()
loss.backward()
```

Figure 4

ii.Why do we sample training data from a replay memory instead of using a set of consecutive frames? The main reason to use replay buffer over consecutive frames is to avoid overfitting by breaking the correlation in a sequence of frames that stabilises the training. The data points from replay buffer are randomly sampled from the past experiences stored in the buffer and hence the random choice breaks the correlation and also provides a diverse set of frames for generalization of the training giving an efficient way to use past experiences to train a robust agent.

c. Action Selection:

```
# TODO: Select greedy action
state = torch.from_numpy(state).float().unsqueeze(0)

# Get the Q-values for all actions in the current state
q_values = policy_net(state.reshape(1,3,96,96))

# Select the action with the highest Q-value (i.e., the greedy action)
_, action = q_values.max(1)

# Convert the action tensor to an integer
action = action[0].item()

return action
```

Figure 6

```
# TODO: Select exploratory action
exploration_prob = exploration.value(t)
if random.random() < exploration_prob:
    # Select a random action
    action_id = random.randrange(action_size)
else:
    # Select the greedy action
   action_id=select_greedy_action(state,policy_net,action_size)
return action_id
```

Figure 7

Fig. 6. Shows the implementation of select_greedy_action algorithm where the action corresponding to the maximum Q values from the policy net is selected. Fig.7. shows the implementation of select_exploratory_action algorithm where exploration probability at the t time step is taken from exploration. A random action is selected with probability exploration_prob, otherwise greedy action is selected.

i. **Why do we need to balance exploration and exploitation in RL and how does ε-greedy accomplish this:** The agent's main aim is to maximise long-term reward hence to do that it is important for the agent to keep exploring new actions in the environment and take into account their outcomes rather than only following its known policy(exploitation). That way it does not miss out on some policies that could yield better results.

   The main concept of ε-greedy algorithm is to select the best action from the current policy with 1-ε probability and to choose any random action with ε probability where ε is the exploration probability. Initially the agent has less knowledge so epsilon value is large and the agent explores more while when the agent learns with time the epsilon value is reduced so that it exploits its current best learned policy more. In this way, ε-greedy algorithm accomplish balance between exploration and exploitation .

d. Training: The Deep Q learning was trained using train_racing_cluster.py. Fig 8. and 9. Shows the loss and reward curves generated. It is evident that over the course of time both loss and reward increases and becomes kind of monotonous after a considerable period of time.

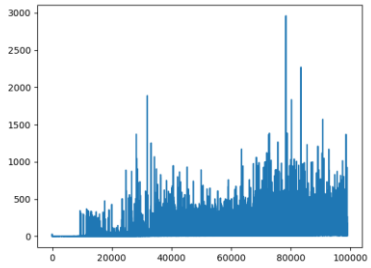

Figure 8

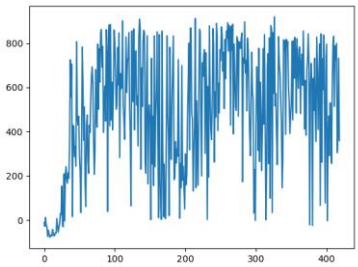

Figure 9

Agent achieving positive reward: The Q learning network was able to achieve positive reward without 4000 timesteps and further throughout the 100000 timesteps it continuously gave positive reward with fluctuations at some places.

ε-greedy and cumulative reward: The ε-greedy scheduler used an ε=0.1 and initial value of 1 and reduced till 0.02. The exploration probability was slowly reduced over time and was made very less towards the end of the training. The exploration probability was not kept high for a long time since otherwise the cumulative reward will become less as the agent will start exploring more and not taking learned action into consideration. At the end, due to less value of exploration probability, the agent chose the best learned action and hence gave large cumulative reward.

Loss curve difference from supervised learning: The loss usually decreases throughout the training in case of supervised learning. However, in Reinforcement learning with DQN the loss was increasing throughout the training.

e. Evaluation: The algorithm was evaluated with evaluate_racing_cluster.py and a positive reward of 373.302 was achieved by the agent. The agent almost performs well at every stage only at few critical turns it goes off-road sometimes.

3.3. Further investigations and Extensions:

a. Discount Factor:

a.1. Increase in discount factor: γ=2(Future rewards given more importance)

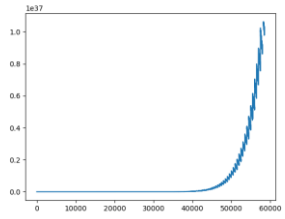

Figure 10

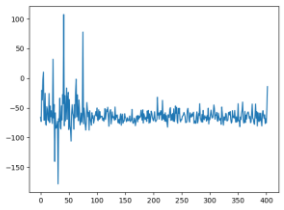

Figure 11

Fig. 10 and 11 shows the loss and reward curve for the following where the loss is steeply increasing and reward is continuously negative. Final evaluation reward is -31.48 since future rewards are dependent on the current state and cannot have more weightage than immediate rewards.

a.2. Decrease in discount factor: γ=0.5(Immediate rewards given more importance)

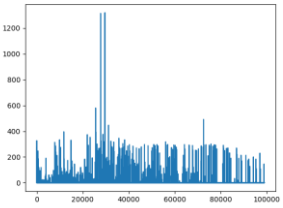

Figure 12

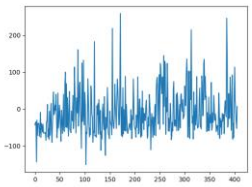

Figure 13

Fig. 12 and 13 shows the loss and reward curve for the following reward is continuously very less Final evaluation reward is 47.14 since the immediate rewards are given much higher preference and the agent fails to take into account the future rewards.

i. Why do we use discount factor and in which cases it will be a problem not to use the same?

The discount factor decides the amount of contribution future rewards should have compared to the immediate reward while calculating cumulative reward. It is the fraction of future rewards that is added to the current reward in order to get the final reward at a time step. The discount factor allows the agent to consider long term consequences into account and make decisions to maximise cumulative reward over time. If the discount factor is less than 1, the immediate reward gets more importance than the future rewards. If the discount factor is much more than 1 then future rewards are taken into account more than immediate ones. If there is no discount factor all rewards are considered equally important and hence the agent can sometimes settle for short time rewards. Hence, it is important to choose the correct discount factor.

In case of game playing or continuous control where the time horizon is not well defined not using a discount factor will make the cumulative reward sum up to infinity after considerable time steps and make the training unstable.

b. Action Repeat Parameter:

b.1. Increase in action repeat: Action Repeat=6

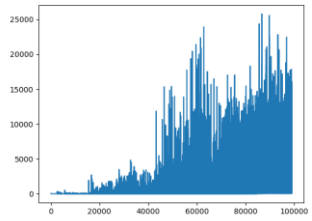

Figure 14

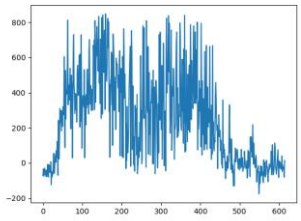

Figure 15

Fig. 14 and 15 shows the loss and reward curve for the following where the loss is increasing and reward is initially positive but slowly becomes negative. Final evaluation reward is -17.93. The probable reason is too much exploration towards to end with same action giving different rewards made the prediction ambiguous for the car.

b.2. Decrease in action repeat: Action Repeat=1(Every action is repeated only once)

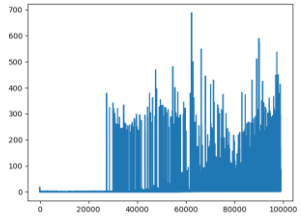

Figure 16

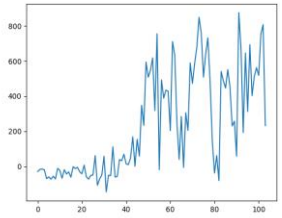

Figure 17

Fig. 16 and 17 shows the loss and reward curve for the following where the loss is increasing and reward is fluctuating a lot. Final evaluation reward is 91.529. Here, every action is attempted only once and hence only one outcome is considered giving the agent less opportunity to explore possibilities.

The training progress becomes significantly slower when repeat action parameter is increased and faster when it is reduced. The evaluation score however reduces for both.

i. Why repeating action is important?

Repeating action can reduce the variance in the outcome as one action can lead to several outcomes in this case. If the actions are repeated the agent every time learns more about the probable outcomes and hence can become a robust one since it has seen more than one scenario resulting from one action. Repeating the action several times can also give an idea whether it constantly gives good outcome of exploration or not.

c. Action Space:

c.1. Action Space 1:

Extra actions: Stop[0,0,0]

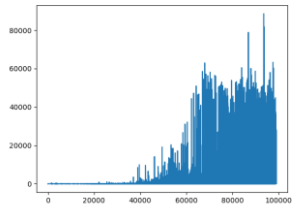

Figure 18

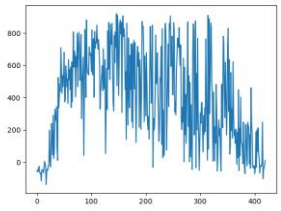

Figure 19

Fig. 18 and Fig. 19 shows the loss and reward curves for the new action space. The loss constantly increases and the reward is constant for sometime but it eventually reduces. The evaluation cumulative reward is 13.432. Addition of null action in the space results in reduction of the final reward.

c.2. Action Space 2:

Extra actions: Stop[0,0,0], Slight turn left[-0.7,0.1,0], Slight turn right[0.7,0.1,0]

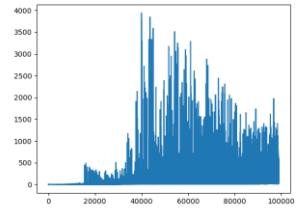

Figure 20

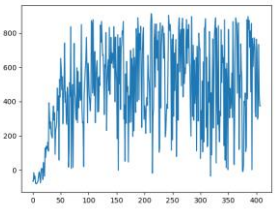

Figure 21

Fig. 20 and Fig. 21 shows the loss and reward curves for the new action space. The loss increases and eventually decreases and the reward is constant and considerably high throughout. The evaluation cumulative reward is 475.074. Addition of this action in the space shows around 100 reward value increase.

c.3. Action Space 3:

Extra actions: Stop[0,0,0], Turn left and stop[-0.7,0,1], Turn right and stop[0.7,0,1]

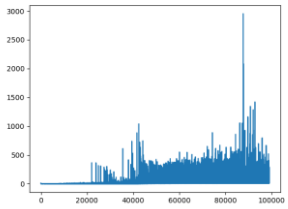

Figure 22

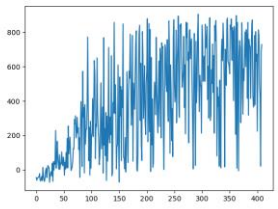

Figure 23

Fig. 22 and Fig. 23 shows the loss and reward curves for the new action space. The loss is low and considerably constant and the reward increases and eventually becomes constant towards the end. The evaluation cumulative reward is 531.156. Addition of this action in the space shows around 200 reward value increase than the baseline model.

The agent's driving style deteriorated for action space 1 however for action space 2 and 3 the agent's driving style improved considerably. The evaluation parameter also showed a similar result as stated above. The actions like turn slightly right and left and turn left or right and stop improved the agent's performance while the action stop did not.

i. Why it might not always be useful to increase the agent's action space?

Several problems right arise in this case such as,

a. Good actions might occur very less during the random choice due to large action space and hence the reward would be difficult to maximise. It hence becomes difficult for the agent to learn good policy.

b. With large action space while exploration and exploitation is carried out the agent might have to try to unnecessary many actions and corresponding states that slows down the learning process.

c. The increase in actions and states might make it difficult to define the value and policy functions as well.

ii. Why does DQN use a discrete action space and how can it be solved?

DQN uses a discrete action space since it executes a neural network for each action to predict the corresponding Q values.

Some ways of using continuous action space for by discretizing the action space or clustering similar actions together from a continuous action space. The other way to solve this is to use Deep Deterministic Policy Gradient that uses a separate neural network to approximate the policy and don't calculate the Q values. This allows the agent to choose actions from continuous space.

d. Double Q-learning:

```
# Step 1: Sample transitions from replay_buffer
state, action, reward,next_state, done = replay_buffer.sample(batch_size)


# Convert batch to tensors
state_batch = torch.tensor(state, device=device, dtype=torch.float32)
action_batch = torch.tensor(action, device=device, dtype=torch.long)
reward_batch = torch.tensor(reward, device=device, dtype=torch.float32)
next_state_batch = torch.tensor(next_state, device=device, dtype=torch.float32)
done_batch = torch.tensor(done, device=device, dtype=torch.uint8)
#print(state_batch.shape)
#print(next_state_batch.shape)

# Step 2: Compute Q(s_t, a)
q_values = policy_net(state_batch.reshape(batch_size,3,96,96)).gather(1, action_batch.unsqueeze(1))
#print(q_values.shape,"Q")

# Step 3: Compute max_a Q(s_{t+1}, a) for all next states.
with torch.no_grad():
    # Use policy network to select the action
    next_actions = policy_net(next_state_batch.reshape(batch_size, 3, 96, 96)).max(1)[1]
    # Use target network to evaluate the selected action
    next_q_values = target_net(next_state_batch.reshape(batch_size, 3, 96, 96)).gather(1, next_actions.unsqueeze(1))
    # Mask next state values where episodes have terminated
    masked_next_q_values = (1 - done_batch.reshape(batch_size, 1)) * next_q_values
    # Step 5: Compute the target
    targets = reward_batch.reshape(batch_size,1) + gamma * masked_next_q_values
#print(targets.shape)
# Step 6: Compute the loss
loss = F.mse_loss(q_values.to(device), targets.to(device))

# Step 7: Calculate the gradients
optimizer.zero_grad()
loss.backward()

# Step 8: Clip the gradients
for param in policy_net.parameters():
    param.grad.data.clamp_(-1, 1)

# Step 9: Optimize the model
optimizer.step()

return loss.item()
```

Figure 24

Figure 24 shows the updated learning for Double Q Learning. Here the actions are taken from the policy net and the final Q values are evaluated using these actions and target net. Hence the targets and final Q values are calculated using actions from different values. MSE Loss is then calculated between the inputs and the targets.

The training and evaluation performance increased considerably. The training loss and reward curves are shown in Fig. 25 and Fig. 26. The final evaluation score was 582.477.

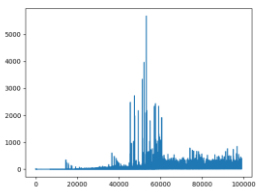

Figure 25

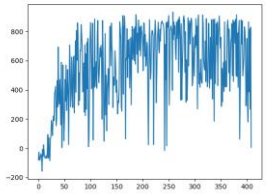

Figure 26

i. **Reason for overestimation of values in Q-learning:** In case of DQN, same values are used to select and evaluate an action and hence they can end up overestimating the values.
ii. **Double Q learning solving this issue:** Double Q learning detaches this connection between selection and evaluation. In double Q learning two value functions are separately learned to obtain two sets of weights, one for estimating the greedy policy and another for estimating the final Q values. This way the correlation between the evaluation and selection is broken.

Best Solution:

AgentComp1:

The changes in the best solution are

1. Final action space:
   Turn Left: [-1.0, 0.05, 0]
   Turn Right: [1.0, 0.05, 0]
   Acceleration: [0, 0.5, 0]
   Brake: [0, 0, 1]
   Stop: [0, 0, 0]
   Turn slightly left: [-0.7, 0.1, 0]
   Turn slightly right: [0.7, 0.1, 0]
   Turn left and brake: [-0.7, 0, 1]
   Turn right and brake: [0.7, 0, 1]
2. Double Q Learning approach is used in the best model formed.
3. The same baseline model is used for the model.
4. The agent is trained with discount factor γ=0.99, action_repeat=4, exploration factor=0.1.

Final evaluation score: 609.148.(Please use the new action space for evaluating this)

AgentComp2:

The changes in the best solution are

1. The action space is same as baseline.
2. Double Q learning approach is used in the best model.
3. The same baseline model is used for the model.
4. The agent is trained with discount factor γ=0.99, action_repeat=4, exploration factor=0.1.

Final Evaluation score: 582.477

Appendix

The zip contains

1. BaseLine:
   a. Model.py
   b. Action.py
   c. Learning.py
   d. Replay_buffer.py
   e. Schedule.py
   f. Utils.py
   g. Deepq.py
   h. Train_racing_cluster.py
   i. Evaluate_racing_cluster.py
   j. Agent.pt
   k. Episode_rewards_agent.png
   l. Training_losses_agent.png
2. Discount factor:
   a. Episode_rewards_agentgamma2.png
   b. Episode_rewards_agentgamma0.5.png
   c. Training_losses_agentgamma2.png
   d. Training_losses_agentgamma0.5.png
   e. Agent2.pt
   f. Agent0.5.pt

3. Action Repeat:
    a. Episode_rewards_agentrepeat6.png
    b. Episode_rewards_agentrepeat1.png
    c. Training_losses_agentrepeat6.png
    d. Training_losses_agentrepeat1.png
    e. Agentrepeat6.pt
    f. Agentrepeat1.pt
4. Action Space:
    a. Action_new.py
    b. Episode_rewards_action1.png
    c. Episode_rewards_action2.png
    d. Episode_rewards_action3.png
    e. Training_losses_action1.png
    f. Training_losses_action2.png
    g. Training_losses_action3.png
    h. Agentaction1.pt
    i. Agentaction2.pt
    j. Agentaction3.pt
5. Double Q learning:
    a. Leaning_double.py
    b. Episode_rewards_double.png
    c. Training_losses_double.png
    d. Agent_double.pt
6. Final Models(Competition):
    a. Agentcomp1.pt
    b. Agentcomp2.pt