Name: **Rohan Prasad**
Email: rpp5524@psu.edu
PSU ID: **980707395**

Date: 10/27/2024

CSE 584 Homework 2

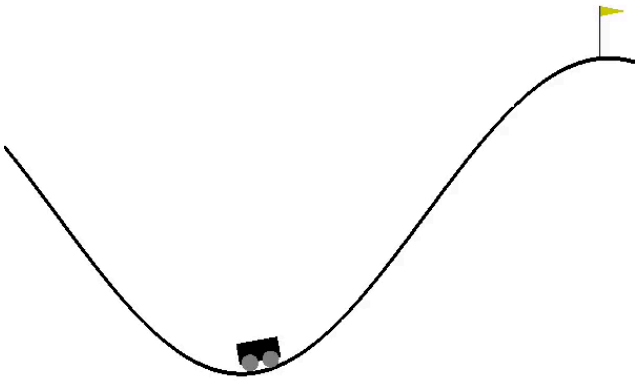
Algorithm: **Deep Q-Network (DQN) algorithm**

I have chosen the following GitHub repository:
https://github.com/Rafael1s/Deep-Reinforcement-Learning-Algorithms/blob/master/MountainCar-DQN/

This project is an implementation of a **Deep Q-Network (DQN) algorithm** applied to the **Mountain Car problem**

Problem Description

The objective is to climb a steep hill with a mountain car which is underpowered. The problem is that the car's engine isn't powerful enough to climb the hill easily, so the agent has to figure out how to use momentum by pacing back and forth until they have enough power to finish the mission. Typically, the car starts in a valley and has to travel to the top of the right slope, which is between two hills.



The repository has the following files;

1. **model.py**: It defines the neural network architecture for the DQN, consisting of:
    a. Three hidden layers
    b. A non-linear activations (to approximate the optimal action-value function.)

2. **agent.py**: the core functionalities of the DQN agent, including action selection via an epsilon-greedy strategy for exploration and exploitation balance, and learning from stored experiences in a replay buffer to update the network weights.

3. **replay_buffer.py**: Implements the replay memory. It allows the DQN agent to remember and reuse past experiences.

4. **Training Notebook (WatchAgent-MountainCar-DQN.ipynb)**: It handles the interaction between the agent and environment. It initializes the environment, the agent and has the train function over numerous epochs. The notebook also contains utilities for loading pre-trained models and visualizing the agent's performance in the environment.

Overall Process

The overall process starts with giving the agent an **initial state** and **action** space derived from the Mountain Car environment. The agent is then trained where it **learns** to navigate the environment by maximizing cumulative rewards, adjusted through a discount factor. The model's parameters are **updated** every training pass by sampling from the replay buffer to minimize the loss between predicted and **targeted Q-values**. The effectiveness of the agent is evaluated through episodic rewards and visual assessments of its ability to solve the Mountain Car problem.

Reinforcement Learning Functions

The important functions are in model.py and the agent.py.
model.py contain the neural network architecture
agent.py contains the agent class. The agents will be instantiated from this class and they will interact with the environment.

Functions:

1. model.py
    a. __init__()
    b. forward()
2. agent.py
    a. get_action()
    b. learn()
    c. soft_update()
    d. hard_update()

Code with comments

model.py

```
class QNetwork(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim) -> None:
        # Constructor to initialize QNetwork
        super(QNetwork, self).__init__()

        # This is the 1st hidden layer with PReLU activation fn
        # It takes the state input and applies a linear transformation
        # followed by a PReLU activation function
        self.layer1 = torch.nn.Sequential(
            torch.nn.Linear(input_dim, hidden_dim), # This Linear layer
transform inputs to hidden dimension
            torch.nn.PReLU() # PReLU activation function adds non-linearity
to the neuron outputs
            # model can learn more complex functions with this activation
        )

        # The 2nd hidden layer takes the output of the first layer and
applies the same transformations
        self.layer2 = torch.nn.Sequential(
            torch.nn.Linear(hidden_dim, hidden_dim), # Continues from the
output of the first layer
            torch.nn.PReLU() # PReLU allows the network to further adjust
activations during training
        )

        # The 3rd hidden layer repeats the structure of previous layers
        self.layer3 = torch.nn.Sequential(
            torch.nn.Linear(hidden_dim, hidden_dim), # Same transformation
as previous layers
            torch.nn.PReLU() # Maintains non-linearity in deeper parts of
the network
        )

        # This output layer maps the final hidden layer's output to the
action values
        # No activation function is used here because we need the raw score
values to compute the Q-value
        self.final = torch.nn.Linear(hidden_dim, output_dim)
```

```
    # This is the function for forward propagation
    # This is using the layers that the code has defined above in the
__init__() contructor
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Returns a Q_value

        Args:
            x (torch.Tensor): `State` 2-D tensor of shape (n, input_dim)

        Returns:
            torch.Tensor: Q_value, 2-D tensor of shape (n, output_dim)
        """
        # They pass the input through the first hidden layer
        x = self.layer1(x)
        # Then through the second hidden layer
        x = self.layer2(x)
        # And through the third hidden layer
        x = self.layer3(x)
        # The output layer returns the predicted Q-values
        x = self.final(x)

        return x
```

---

agent.py

```
class Agent(object):
```

The Agent class's __init__ method:

- uses Deep Q-Learning to create a deep reinforcement learning agent.
- sets up two neural networks (q_local and q_target) to operate on a designated device (CPU or GPU),
- initializes them for the current and target Q-values, and uses mean squared error for the loss function.
- incorporates a replay memory with a 10,000-item capacity to store experiences, which is essential for efficient learning, and employs the Adam optimizer for network updates.

```
    # This is the function to select actions based on current state and epsilon value.
    def get_action(self, state, eps, check_eps=True):
        """Returns an action

        Args:
            state : 2-D tensor of shape (n, input_dim)
            eps (float): eps-greedy for exploration
        Returns: int: action index
        """
        global steps_done
        sample = random.random() # They generate a random sample for epsilon comparison.
        # Now they check whether to use the model for action or select a random action based on epsilon value.
        if check_eps==False or sample > eps:
            with torch.no_grad(): # They temporarily set all the required gradient calculations to off.
                # Forward pass through the local Q-network and get the action with the maximum Q-value.
                return self.q_local(Variable(state).type(FloatTensor)).data.max(1)[1].view(1, 1)
        else:
           # now returns a random action as a tensor on the specified device.
            return torch.tensor([[random.randrange(self.n_actions)]], device=self.device)
```

```
    # This is the Function to update the Q-network from a batch of
experiences.
    def learn(self, experiences, gamma):
        """Prepare minibatch and train them

        Args:
        experiences (List[Transition]): batch of `Transition`
        gamma (float): Discount rate of Q_target
        """

        if len(self.replay_memory.memory) < BATCH_SIZE:
            return;

        # First we sample a batch of transitions from memory.
        transitions = self.replay_memory.sample(BATCH_SIZE)
        # Then we unpack batch data into a Transition named tuple.
        batch = Transition(*zip(*transitions))

        # Then we concatenate all respective items from the batch into
separate tensors.
        states = torch.cat(batch.state)
        actions = torch.cat(batch.action)
        rewards = torch.cat(batch.reward)
        next_states = torch.cat(batch.next_state)
        dones = torch.cat(batch.done)

        # Now we compute the expected Q-values from the local model for the
sampled states and actions.
        Q_expected = self.q_local(states).gather(1, actions)

        # Then we compute the next Q-values from the target model and detach
from the graph to prevent further gradient calculations.
        Q_targets_next = self.q_target(next_states).detach().max(1)[0]

        # Then we compute the target Q-values for the current states using
the Bellman equation.
        Q_targets = rewards + (gamma * Q_targets_next * (1-dones))

        self.q_local.train(mode=True) # Train forward with train mode set to
True
        # Zero the gradients to prevent accumulation from previous
iterations.
        self.optim.zero_grad()
```

```
        # Now we Calculate the loss between the expected Q-values and the
target Q-values.
        loss = self.mse_loss(Q_expected, Q_targets.unsqueeze(1))
        # then perform backpropagation to minimize the loss and update
network weights.
        loss.backward()
        self.optim.step() # Update the optimizer to move toward the minima.
```

```
    # This function helps slowly adjust the weights of the target network
towards the weights of the local network.
    # A soft update blends the parameters of the local model and the target
model, which can stabilize training
    # in Deep Q-Learning by making gradual updates to the target model
    def soft_update(self, local_model, target_model, tau):
        # for loop for each parameter (weights and biases) in both the local
and target models.
        for target_param, local_param in zip(target_model.parameters(),
local_model.parameters()):
            # They perform a weighted sum of the local model's parameter and
the target model's parameter.
            target_param.data.copy_(tau*local_param.data +
(1.0-tau)*target_param.data)
```

```
    # This function copies all of the parameters from the local model to the
target model directly.
    # I think it is usually used at the start of training or periodically
during training to ensure the target
    # network stays up-to-date.
    def hard_update(self, local, target):
        for target_param, param in zip(target.parameters(),
local.parameters()):
            # Thye directly copy the local model's parameter values into the
target model.
            target_param.data.copy_(param.data)
```

Step-by-step Understanding of the code

Step 1 - First define the Environment and Neural Network Architecture

- The code defines the Neural Network (QNetwork) in model.py, This network estimates the **Q-values** for each action given the current state. The network uses several layers of linear transformations followed by non-linear activations (**PReLU**).

Step 2 - Agent get initialized

- In agent.py, an instance of the Agent class is created with two Q-networks (**q_local** and **q_target**`).
- The agent is initialized with hyper-parameters such as the number of states, number of actions, learning rate etc.
- The **replay buffer** stores experiences that the agent encounters.

Step 3 - Epsilon-Greedy Policy for Getting Action (get_action() function in agent.py)

- The agent uses an **epsilon-greedy strategy**. It selects random actions with probability epsilon and greedy actions from the Q-network otherwise.
- Gradually, epsilon is decreased to make the agent exploit more of its learned experiences.

Step 4 - Agent interacts with the Environment

- In a single pass of the episode, the agent starts in an initial state and takes actions until an end state is reached (for example: if the car has reached the goal).
- Each step generates an **experience tuple (state, action, reward, next_state, done)**. This tuple is stored in the replay buffer.

Step 5 - Agents Learn from Experience (done in the learn() function)

- The agent periodically samples a batch of experiences from the buffer to learn from.
- For each sampled experience, the agent computes Q-target using the reward and the discounted **highest Q-value of the next state**. (by q_target)
- Then we calculate mean squared error loss between the Q-targets and the Q-values predicted by the q_local network for the actions taken.
- The parameters of the q_local network are updated using **backpropagation**.

Step 6 - Update the q_target (done in the soft_update() function)

- q_target network weights are slowly updated to q_local network weights using a **weighted sum** (based on tau). This keeps the target network stable but slowly adapting to improved policies.

This process is repeated over many episodes/epochs until the q_local network weights and q_target network weights converge.

References:

1) MountainCar DQN Implementation: The Deep Q-Network (DQN) implementation for the MountainCar problem as seen in this repository is a practical example of using reinforcement learning algorithms.
GitHub repository:
https://github.com/Rafael1s/Deep-Reinforcement-Learning-Algorithms/tree/master/MountainCar-DQN