CSE-584 Homework #2

Name: Naga Aravind Kundeti
PSU id: 970551363
Email: nfk5351@psu.edu

Abstract for Reinforcement Learning Code

The reinforcement learning code for the game environment of snake which uses a Deep Q- Network (DQN) is intended to produce an agent capable of developing good strategies from experience. It aims at achieving the greatest overall reward in decision making over a given period through the improvement of the efficiency of decision making with time. Here is an elaboration on the key components and processes involved:

Key Components

Q-Network Architecture: The neural network model that is constructed here will incorporate an input layer and three hidden layers and an output layer and is trained using PyTorch. They all consists of ReLU activation functions to enable the introduction of non-linearity in each layer. This architecture is important for using the Q-value function that predicts the amount of the future utility for each possible action, when the current state of the game is known.

Replay Memory: A replay buffer is incorporated to allow the storing of past experiences as tuples of (state, action, reward, next state, and done flag). As we mentioned earlier this storage mechanism enables the model to sample random batches of experiences during training. In doing so, it disrupts the relation of consecutive experiences and helps to balance through the learning curve.

Training Process

The training procedure is obtained through how the Q-network is trained from time to time using the mini-batches that are drawn randomly from the replay buffers.

The key steps include:

Sampling: Just choose a random batch of experiences from the replay memory, with no need to remember which of them were chosen before and which of them should be chosen next.

Prediction: By applying the Q-network we define the Q-values of each action of the sampled states.

Target Calculation: With respect to the immediate reward that is associated with a state-action pair as well as the estimated future reward from another state-action pair identified by a policy, use the Bellman equation to compute the target Q-values.

Loss Computation: In order to prevent the action with the highest Q-value from being chosen regardless of the actual situation on the game, calculate the loss as the difference of the target and predicted Q-values.

Optimization: Update the network weights as applicable from above equation devised for using gradient descent method to minimize this loss.

Other parameters include the learning rate, which other hyperparameters are adjusted during experimentation for the most efficient learning. The main advantage of this approach is that, in this way, the agent is able to enhance its performance gradually on the basis of results which can be both positive and negative.

In general, this form of reinforcement learning uses deep learning approaches to improve choice-making mechanisms and suitability of a game such as Snake. The agent then improves over time after multiple rounds, though with high dimensions for evaluating the success probabilities, which represents an example of reinforcement learning for working on difficult tasks for which there is an expansive number of states.

Below are the core sections of the reinforcement learning implementation extracted from the code files provided for the paper, followed by the description of steps for better understanding of each line of the code.

QNetwork Class (from model.py)

Code:

```
import torch
import torch.nn as nn

# Specify a class of the neural network for the Q-Network
class QNetwork(nn.Module):

    # Set the network's input, hidden, and output dimensions during
initialisation.
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(QNetwork, self).__init__()

        # First connected layer
        self.fc1 = nn.Linear(input_dim, hidden_dim)

        # Second connected layer
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)

        # Third connected layer
        self.fc3 = nn.Linear(hidden_dim, hidden_dim)

        # Output layer
        self.fc4 = nn.Linear(hidden_dim, output_dim)

        # Activation function (ReLU)
        self.relu = nn.ReLU()

    # Forward traffic via the network
    def forward(self, x):
        # Apply ReLU activation after passing input via the first layer.
        l1 = self.relu(self.fc1(x.float()))

        # Apply ReLU activation after passing the result through the second
layer.
        l2 = self.relu(self.fc2(l1))

        # Apply ReLU activation after passing the result via the third
layer.
        l3 = self.relu(self.fc3(l2))

        # Send the output layer the result (no activation function here).
        l4 = self.fc4(l3)

        return l4

    # Get the network input from the locations of the player and the apple.
    def get_network_input(player, apple):
        # Obtain the player's proximity information, such as the distance
to walls or an apple.
        proximity = player.getproximity()

        # Create a single tensor by concatenating the player's location,
closeness, direction, and apple's position.
        x = torch.cat([
```

```
            torch.from_numpy(player.pos).double(),
            torch.from_numpy(apple.pos).double(),
            torch.from_numpy(player.dir).double(),
            torch.tensor(proximity).double()
        ])

        return x
```

Screenshots of the code (model.py)

```
import torch
import torch.nn as nn

# Specify a class of the neural network for the Q-Network
class QNetwork(nn.Module):

    # Set the network's input, hidden, and output dimensions during initialisation.
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(QNetwork, self).__init__()

        # First connected layer
        self.fc1 = nn.Linear(input_dim, hidden_dim)

        # Second connected layer
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)

        # Third connected layer
        self.fc3 = nn.Linear(hidden_dim, hidden_dim)

        # Output layer
        self.fc4 = nn.Linear(hidden_dim, output_dim)

        # Activation function (ReLU)
        self.relu = nn.ReLU()

    # Forward traffic via the network
    def forward(self, x):
        # Apply ReLU activation after passing input via the first layer.
        l1 = self.relu(self.fc1(x.float()))

        # Apply ReLU activation after passing the result through the second layer.
        l2 = self.relu(self.fc2(l1))

        # Apply ReLU activation after passing the result via the third layer.
        l3 = self.relu(self.fc3(l2))

        # Send the output layer the result (no activation function here).
        l4 = self.fc4(l3)

        return l4

    # Get the network input from the locations of the player and the apple.
    def get_network_input(player, apple):
        # Obtain the player's proximity information, such as the distance to walls or an apple.
        proximity = player.getproximity()

        # Create a single tensor by concatenating the player's location, closeness, direction, and apple's position.
        x = torch.cat([
            torch.from_numpy(player.pos).double(),
            torch.from_numpy(apple.pos).double(),
            torch.from_numpy(player.dir).double(),
            torch.tensor(proximity).double()
        ])

        return x
```

Explanation of model.py:

QNetwork class: It also defines the neural network used in the approximation of the Q-values which this class defines.

Input structure: The input includes position of the player and the apple, movement schemata and distance to the obstacles.

Forward propagation: These proposed the sequential linear layers, with ReLU non-linearities as the basic building block followed by the final layer predicting Q-values of all possible actions.

ReplayMemory Class (from replay_buffer.py)

Code:

```
import random

# Establish a class to store and sample training sessions.
class ReplayMemory(object):

    #  Set the maximum size for the replay memory at initialisation.
    def __init__(self, max_size):
        self.max_size = max_size  #  The most experiences that can be
stored
        self.buffer = []          # Create a list to save experiences

    # Fill the memory buffer with a fresh experience.
    def push(self, state, action, reward, next_state, done):
        experience = (state, action, reward, next_state, done)  # Make a
tuple of experiences
        self.buffer.append(experience)                          # Increase
the buffer's experience

    # Take a sample of the memory buffer's experiences.
    def sample(self, batch_size):
        state_batch = []       # List for batch state storage
        action_batch = []      # List for batch action storage
        reward_batch = []      # List for batch reward storage
        next_state_batch = []  # List for batch storage of the upcoming
states
        done_batch = []        # List for batch storage of completed flags

        batch = random.sample(self.buffer, batch_size)  # Choose a group of
experiences at random.

        for experience in batch:
            state, action, reward, next_state, done = experience  # Unpack
the tuple of experiences

            state_batch.append(state)         # Include the state in the
state batch
            action_batch.append(action)       # Include an action in the
action batch.
            reward_batch.append(reward)       # Include the reward in the
reward batch.
            next_state_batch.append(next_state)  # Add the following state
to the batch of states.
```

```
            done_batch.append(done)            # Include the completed
batch's flag.

        return (state_batch, action_batch, reward_batch, next_state_batch,
done_batch)

    # Cut memory buffer if it currently occupies more space than permitted
    def truncate(self):
        self.buffer = self.buffer[-self.max_size:]  # Filter out the most
current events

    # Get present size of memory buffer
    def __len__(self):
        return len(self.buffer)
```

Screenshot of the code (replay_buffer.py)

```python
import random

# Establish a class to store and sample training sessions.
class ReplayMemory(object):

    #  Set the maximum size for the replay memory at initialisation.
    def __init__(self, max_size):
        self.max_size = max_size  #  The most experiences that can be stored
        self.buffer = []          # Create a list to save experiences

    # Fill the memory buffer with a fresh experience.
    def push(self, state, action, reward, next_state, done):
        experience = (state, action, reward, next_state, done)  # Make a tuple of experiences
        self.buffer.append(experience)                          # Increase the buffer's experience

    # Take a sample of the memory buffer's experiences.
    def sample(self, batch_size):
        state_batch = []       # List for batch state storage
        action_batch = []      # List for batch action storage
        reward_batch = []      # List for batch reward storage
        next_state_batch = []  # List for batch storage of the upcoming states
        done_batch = []        # List for batch storage of completed flags

        batch = random.sample(self.buffer, batch_size)  # Choose a group of experiences at random.

        for experience in batch:
            state, action, reward, next_state, done = experience  # Unpack the tuple of experiences

            state_batch.append(state)         # Include the state in the state batch
            action_batch.append(action)       # Include an action in the action batch.
            reward_batch.append(reward)       # Include the reward in the reward batch.
            next_state_batch.append(next_state)  # Add the following state to the batch of states.
            done_batch.append(done)           # Include the completed batch's flag.

        return (state_batch, action_batch, reward_batch, next_state_batch, done_batch)

    # Cut memory buffer if it currently occupies more space than permitted
    def truncate(self):
        self.buffer = self.buffer[-self.max_size:]  # Filter out the most current events

    # Get present size of memory buffer
    def __len__(self):
        return len(self.buffer)

```

Explanation of replay_buffer.py:

1. **ReplayMemory class:** This class implements experience replay as stated in the paper by causing the environment to randomly select from the experiences that have been created.
2. **Buffer management**: Record transitions into a format of (state, action, learning reward, next_state, done) beneficial to the learning process of the agent.
3. **Sampling**: This is done on the premise that experiences are also randomly selected to reduce the association between successive observations and to make the learning process more stable.
4. **Truncation**: If the buffer goes beyond the size defined as max_size then only some of the most recent experiences are saved.

These two files form a crucial part of another algorithm known as the Deep Q-Network or just DQN for short. The model.py contains architecture of Q-network that we have used to predict Q-values; the replay_buffer.py contain logic to organize the experience replay buffer, which allows efficient learning as it consists of sampling from our past experience.

GitHub Link: https://github.com/Rafael1s/Deep-Reinforcement-Learning-Algorithms/tree/master/Snake-Pygame-DQN