Artificial Intelligence

UCS411

Car Obstacle Avoidance via Reinforced Learning

PROJECT REPORT

Submitted to Dr. Swati Kumari

On 17 April 2023

Submitted by: Sannidhya Jain (102103348)

Shrey Pachauri (102103354)

Yash Sharma (102103363)

Vayansh Garg (102153039)

Problem statement:

To train a car within a simulated environment to avoid obstacles after a number of attempts.

Description of problem:

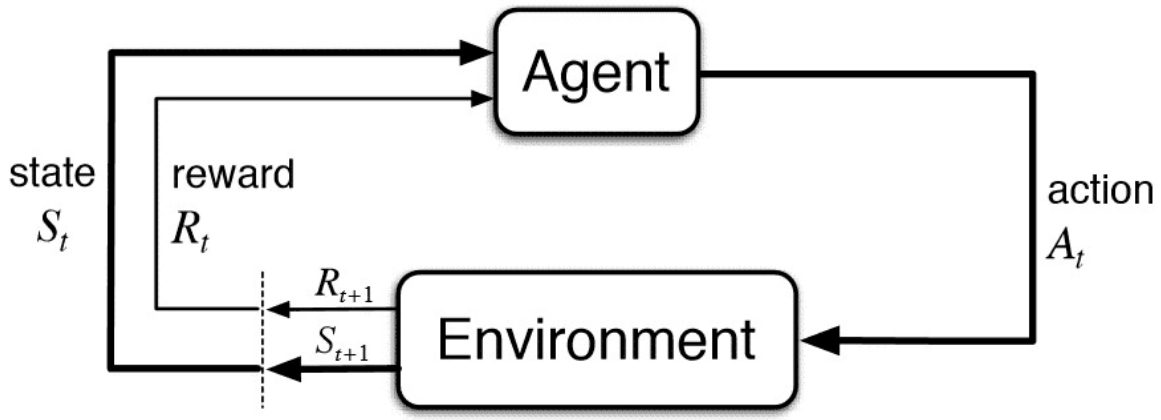
We aim to create from scratch a car racing game, where the computer shall learn what an obstacle is, and should be able to figure out how to avoid it keeping in mind the restraints set within the game by the user. The problem chosen highlights the challenges faced while creating a practical application of reinforced learning that can be used in autonomous vehicles.

Car obstacle avoidance game involves steering/lane changing decisions that must be taken in a short amount of time. This complexity makes it an interesting challenge for a reinforced learning agent.

Code and Explanation

Q-learning is a popular model-free reinforcement learning algorithm used for learning optimal policies in an agent-environment interaction setting. The goal of Q-learning is to learn a Q-value function, which estimates the maximum expected future rewards an agent can get by taking a particular action in a given state. The Q-value function is defined for each state-action pair in the agent's environment.

During training, the agent uses the Q-value function to choose actions that maximize expected future rewards, and updates the Q-value function by performing a Bellman update step using the observed rewards and the estimated Q-values for the next state. This update is done in a way that the Q-value function converges to the true Q-value function over time.



Once the Q-value function has been learned, the agent can use it to determine the optimal policy by selecting the action with the highest Q-value in each state. Q-learning is known for its simplicity, scalability, and ability to handle large state-action spaces.

Agent

```
 7   import numpy as np
 8   import torch
 9   import random
10   from collections import deque
11   from game_AI import Game
12   from model import Linear_QNet, QTrainer
13
```

IMPORTING MODULES

Brief description of the imported modules:

1. Pygame: Pygame is a cross-platform library designed for creating games. It provides functionality for handling graphics, sound, and user input, among other things. In this case, it is being used to create the visual interface for the 1D car obstacle avoidance game.
2. random: The random module provides a suite of functions for generating random numbers and selecting random elements from sequences. It is likely being used in the game to generate random positions for the obstacles.
3. numpy: NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. In this case, it is likely being used to manage the state of the game (e.g., the position of the car and obstacles) as a multidimensional array.
4. PyTorch: PyTorch is an open-source machine learning framework that provides a Python interface for building and training deep neural networks. It includes support for tensor computations with strong GPU acceleration, automatic differentiation for building complex models, and a range of built-in optimization algorithms.
5. os: The os module provides a way to interact with the operating system, including functions for reading and writing files, creating and removing directories, and getting information about the system environment.

```python
def train():
    game = Game()
    agent = Agent()
    record = 0
    # total_scores = 0
    # plot_scores = []
    # plot_mean_scores =[]
    while True:
        #get current state
        state_old = game.get_state()

        # get action or move
        move = agent.get_action(state_old)

        # perform move and get new state
        reward,g_o,score = game.play_step(move)
        next_state = game.get_state()

        #train short memory
        agent.train_short_memory(state_old,move,reward,next_state,g_o)

        #remember
        agent.remember(state_old,move,reward,next_state,g_o)
```

Game is an object of car game environment in which the agent will play.

Get_state function will return current state of the game environment which includes safe position of our vehicle based on distance from left and right edges of the road, and distance from obstacle car.

```python
    def get_action(self,state):
        # Epsilon greedy approach
        self.epsilon = 50 - self.n_games
        final_move = [0,0,0]
        if random.randint(0,200) < self.epsilon:
            move = random.randint(0,2)
            final_move[move] = 1
            self.k.append(0)
        else:
            self.k.append(1)
            state0 =  torch.tensor(np.array(state), dtype=torch.float)
            prediction = self.model(state0)
            move = torch.argmax(prediction).item()
            final_move[move] = 1

        return final_move
```

It takes state as in input and applies Epsilon greedy approach and returns the best action bas decided by this greedy approach.

In epsilon-greedy, the agent selects the action with the highest estimated value (i.e., the action that is believed to have the highest expected reward) with probability (1 - epsilon),

and selects a random action (i.e., explores) with probability epsilon. A higher value of epsilon encourages more exploration, while a lower value of epsilon encourages more exploitation of the best-known action.

```
47      def train_short_memory(self,state,action,reward,next_state,g_o):
48          self.trainer.train_step(state,action,reward,next_state,g_o)
49
50
51      def remember(self,state,action,reward,next_state,g_o):
52          self.memory.append((state,action,reward,next_state,g_o))
53
54
55      def train_long_memory(self):
56          if len(self.memory) > BATCH_SIZE:
57              mini_sample = random.sample(self.memory,BATCH_SIZE)
58          else:
59              mini_sample = self.memory
60
61          states,actions,rewards,next_states,g_os = zip(*mini_sample)
62          self.trainer.train_step(states,actions,rewards,next_states,g_os)
```

After every step the short memory function and trains the model based on the rewards it receives and improves the model.

Remember function appends every step to the deque.

train_long_memory zips a batch from deque and sends it to train_step function; which trains the model based on the rewards.

MODEL:

```
8   class Linear_QNet(nn.Module):
9       def __init__(self, input_size , hidden_size, output_size):
10          super().__init__()
11          self.linear1 = nn.Linear(input_size,hidden_size)
12          self.linear2 = nn.Linear(hidden_size,output_size)
13
14      def forward(self, x):
15          x = F.relu(self.linear1(x))
16          x = self.linear2(x)
17          return x
18
19      def save(self, file_name = 'model.pth'):
20          model_folder_path = './model'
21          if not os.path.exists(model_folder_path):
22              os.makedirs(model_folder_path)
23
24          file_name = os.path.join(model_folder_path,file_name)
25          torch.save(self.state_dict(),file_name)
```

Class Linear_Qnet defines a torch model with single hidden layer and its forward function modifies weights of the model, and save function saves the model.

Game Environment

```
def play_step(self,action):
    # to be called after getting a output of neural network
    self.move(action)
    self.reward = 0
    if self.x>690-car_width or self.x<110:
            self.reward = -10
            return self.reward,True,self.score
```

This function takes action/move as an input, and calls the move function which update the environment, and returns reward, whether the game is over or not, and current score based on current state of environment.

```
def update_ui(self):
    self.gamedisplays.fill(gray)
    rel_y=self.y2%backgroundpic.get_rect().width
    self.gamedisplays.blit(backgroundpic,(0,rel_y-backgroundpic.get_rect().width))
    self.gamedisplays.blit(backgroundpic,(700,rel_y-backgroundpic.get_rect().width))
    if rel_y<800:
            self.gamedisplays.blit(backgroundpic,(0,rel_y))
            self.gamedisplays.blit(backgroundpic,(700,rel_y))
            self.gamedisplays.blit(yellow_strip,(400,rel_y))
            self.gamedisplays.blit(yellow_strip,(400,rel_y+100))
            self.gamedisplays.blit(yellow_strip,(400,rel_y+200))
            self.gamedisplays.blit(yellow_strip,(400,rel_y+300))
            self.gamedisplays.blit(yellow_strip,(400,rel_y+400))
            self.gamedisplays.blit(yellow_strip,(400,rel_y+500))
            self.gamedisplays.blit(yellow_strip,(400,rel_y-100))
            self.gamedisplays.blit(strip,(120,rel_y-200))
            self.gamedisplays.blit(strip,(120,rel_y+20))
            self.gamedisplays.blit(strip,(120,rel_y+30))
            self.gamedisplays.blit(strip,(680,rel_y-100))
            self.gamedisplays.blit(strip,(680,rel_y+20))
            self.gamedisplays.blit(strip,(680,rel_y+30))
```

This code snippet is responsible for updating the game's UI by redrawing the background image and other graphical elements based on the game's state, creating a scrolling effect for the background, and rendering road markings and obstacles on the screen.