AI

Snake Game

T

- AI
  Snake Game
- T
- Model (1)
- Model (2)
- Model (3)
- Model
- Model
- Model

Model (1)

```
import os
import time

import torch
import torch.nn.functional as F
from torch import nn, optim
```

Why use `torch` instead of `tf`?

Model (2)

```
class LinearQNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LinearQNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Model (3)

```
    def save(self, filename=None):
        if filename is None:
            filename = f"model_{time.time()}.pth"

        if not os.path.isdir("./models"):
            os.mkdir("./models")

        torch.save(self.state_dict(), f"./models/{filename}")
```

Model

```
class QTrainer:
    def __init__(self, model, learning_rate, gamma) -> None:
        self.learning_rate = learning_rate
        self.model = model
        self.optimizer = optim.Adam(self.model.parameters(), lr=learning_rate)
        self.gamma = gamma
        self.criterion = nn.MSELoss()
```

Model

```
    def train_step(self, state, action, reward, next_state, game_over):
        state = torch.tensor(state, dtype=torch.float)
        next_state = torch.tensor(next_state, dtype=torch.float)
        action = torch.tensor(action, dtype=torch.float)
        reward = torch.tensor(reward, dtype=torch.float)
```

Model

```
        if len(state.shape) == 1:
            state = torch.unsqueeze(state, 0)
            next_state = torch.unsqueeze(next_state, 0)
            action = torch.unsqueeze(action, 0)
            reward = torch.unsqueeze(reward, 0)

            done = (game_over,)

            pred = self.model(state)

            target = pred.clone()
```

Model

```
        for i in range(len(done)):
            Q_new = reward[i]
            if not done[i]:
                Q_new = reward[i] + self.gamma * torch.max(
                    self.model(next_state[i])
                )

            target[i][torch.argmax(action).item()] = Q_new

        self.optimizer.zero_grad()
        loss = self.criterion(target, pred)
        loss.backward()

        self.optimizer.step()
```

Helper

```
import matplotlib.pyplot as plt
from IPython import display
plt.ion()

def plot(scores, mean_scores):
    display.clear_output(wait=True)
    display.display(plt.gcf())
    plt.clf()
    plt.title("Training...")
    plt.xlabel("Number of Games")
    plt.ylabel("Score")
    plt.plot(scores)
    plt.plot(mean_scores)
    plt.ylim(ymin=0)
    plt.text(len(scores) - 1, scores[-1], str(scores[-1]))
    plt.text(len(mean_scores) - 1, mean_scores[-1], str(mean_scores[-1]))
    plt.show(block=False)
    plt.pause(0.1)
```

Game

```
def play_step(self, action: Action) -> tuple[int, bool, int]:
        # 1. collect user input
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()

        # 2. move
        self.frame_iteration += 1
        self._move(action)  # update the head
        self.snake.insert(0, self.head)

        # 3. check if game over
        reward = 0
        game_over = False
        if self.is_collision() or self.frame_iteration > TIMEOUT * len(self.snake):
            game_over = True
            reward = -10
            print("Timeout | ", end="")
            return reward, game_over, self.score
```

Game

```
        # 4. place new food or just move
        if self.head == self.food:
            self.score += 1
            reward = 10
            self._place_food()
        else:
            self.snake.pop()

        # 5. update ui and clock
        self._update_ui()
        self.clock.tick(SPEED)
        # 6. return game over and score
        return reward, game_over, self.score
```

Agent

```
    def remember(self, state, action, reward, next_state, game_over):
        self.memory.append((state, action, reward, next_state, game_over))

    def train_long_memory(self):
        if len(self.memory) > BATCH_SIZE:
            sample = random.sample(self.memory, BATCH_SIZE)
        else:
            sample = self.memory

        state, action, reward, next_state, game_over = zip(*sample)
        self.trainer.train_step(state, action, reward, next_state, game_over)

    def train_short_memory(self, state, action, reward, next_state, game_over):
        self.trainer.train_step(state, action, reward, next_state, game_over)
```

Agent

```
    def get_action(self, state):
        self.epsilon = 80 - self.games
        final_move = [0, 0, 0]

        if random.randint(0, 200) < self.epsilon:
            move = random.randint(0, 2)
            final_move[move] = 1

        else:
            state0 = torch.tensor(state, dtype=torch.float)
            prediction = self.model(state0)
            move = torch.argmax(prediction).item()
            final_move[move] = 1

        return final_move
```

Agent

```
while True:
    # get old state
    old_state = agent.get_state(game)

    # get move
    final_move = agent.get_action(old_state)

    # preform move and get new state
    reward, game_over, score = game.play_step(final_move)
    new_state = agent.get_state(game)

    # train short memory
    agent.train_short_memory(old_state, final_move, reward, new_state, game_over)

    # remember
    agent.remember(old_state, final_move, reward, new_state, game_over)
```

Agent

```
        # remember
        agent.remember(old_state, final_move, reward, new_state, game_over)

        if game_over:
            # train long memory, plot results
            game.reset()
            agent.games += 1
            agent.train_long_memory()
            if score > record:
                record = score
                agent.model.save()

            plot_scores.append(score)

            total_score += score
            mean_score = total_score / agent.games
            plot_mean_scores.append(mean_score)
            plot(plot_scores, plot_mean_scores)
```