SAT303 - Deep Q-Learning on CartPole-v1 using PyTorch

1. Introduction

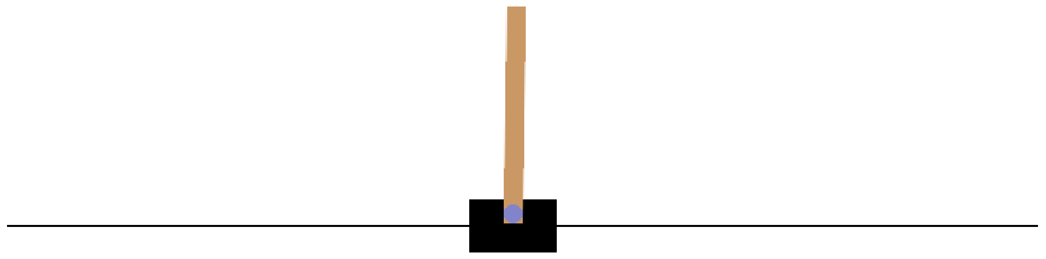
Document: https://gymnasium.org.cn/environments/classic_control/cart_pole/

Main Framework

- **train.py**
- **agents/ --**algorithms
- **cartpole_dqn.py**

(TODO: implement other algorithms)

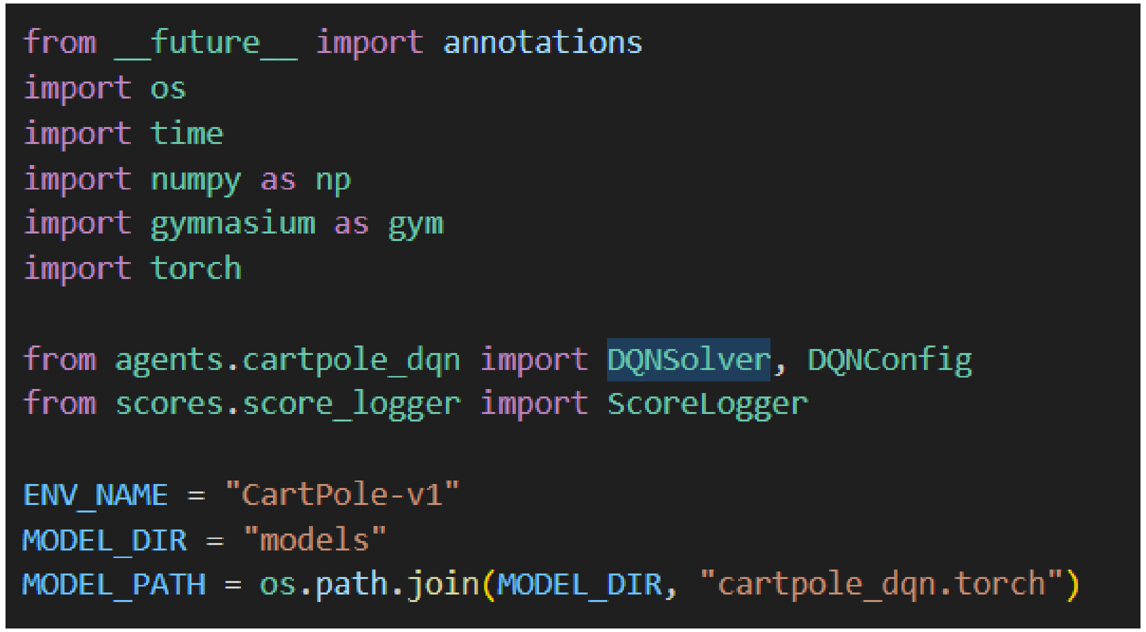
- **scores/score_logger.py**
- **assets/ --two photos**
- **models/ --for saving models**
- **requirement.txt** (pip install -r requiremnet.txt)



2. train.py explanation

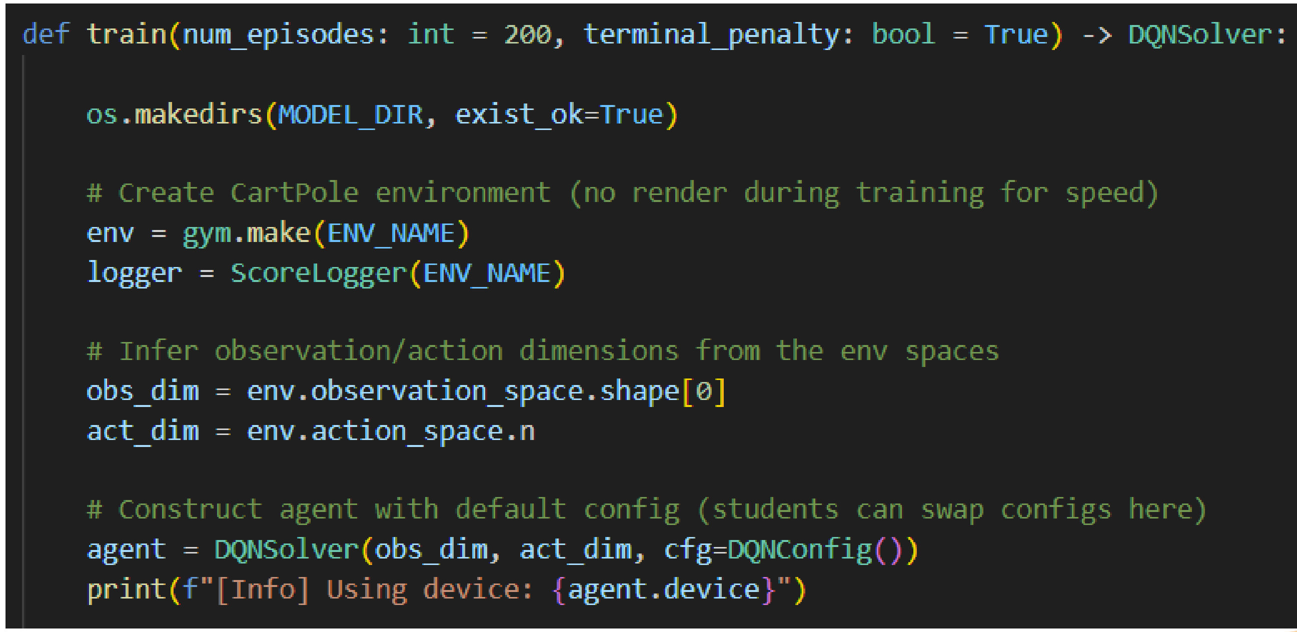
train() and **evaluate()**

import libraries and algorithms
define environment name and model storage path



```python
from __future__ import annotations
import os
import time
import numpy as np
import gymnasium as gym
import torch

from agents.cartpole_dqn import DQNSolver, DQNConfig
from scores.score_logger import ScoreLogger

ENV_NAME = "CartPole-v1"
MODEL_DIR = "models"
MODEL_PATH = os.path.join(MODEL_DIR, "cartpole_dqn.torch")
```

create environment
initialize dimensional and instantiate the **agent**



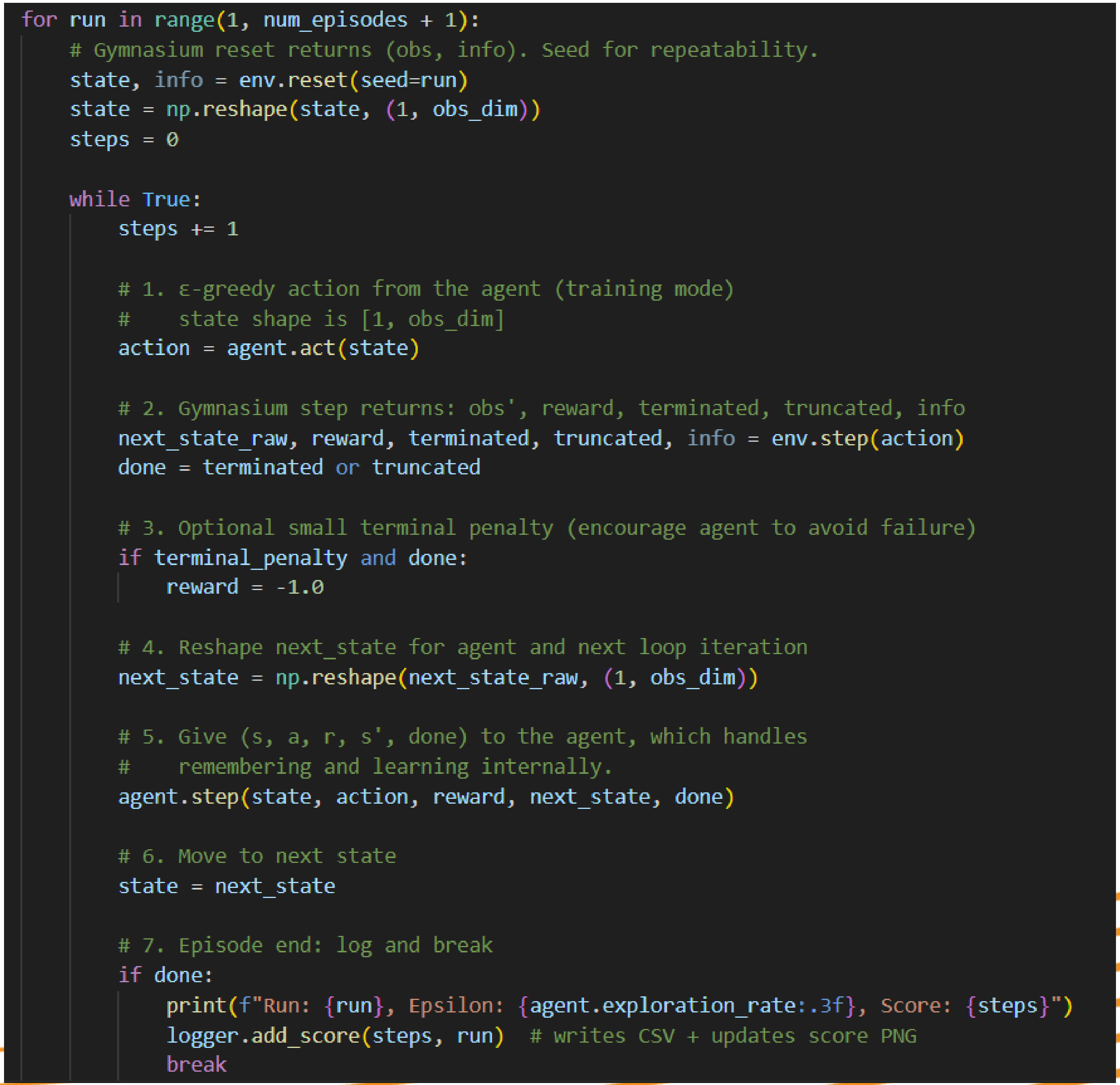
```python
def train(num_episodes: int = 200, terminal_penalty: bool = True) -> DQNSolver:

    os.makedirs(MODEL_DIR, exist_ok=True)

    # Create CartPole environment (no render during training for speed)
    env = gym.make(ENV_NAME)
    logger = ScoreLogger(ENV_NAME)

    # Infer observation/action dimensions from the env spaces
    obs_dim = env.observation_space.shape[0]
    act_dim = env.action_space.n

    # Construct agent with default config (students can swap configs here)
    agent = DQNSolver(obs_dim, act_dim, cfg=DQNConfig())
    print(f"[Info] Using device: {agent.device}")
```

2. train.py explanation

Train process:

- For the entire loop, one **episode represents one game**
- The inner loop "while True" iterates through each action in the environment and the feedback it provides
- **Loop:**

1. obtain action
2. take action to obtain next state, reward, and whether to terminate.
3. (s, a, r, s', done) pass to agent for remembering and learning
4. new state is assigned to the state



```python
for run in range(1, num_episodes + 1):
    # Gymnasium reset returns (obs, info). Seed for repeatability.
    state, info = env.reset(seed=run)
    state = np.reshape(state, (1, obs_dim))
    steps = 0

    while True:
        steps += 1

        # 1. ε-greedy action from the agent (training mode)
        #    state shape is [1, obs_dim]
        action = agent.act(state)

        # 2. Gymnasium step returns: obs', reward, terminated, truncated, info
        next_state_raw, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated

        # 3. Optional small terminal penalty (encourage agent to avoid failure)
        if terminal_penalty and done:
            reward = -1.0

        # 4. Reshape next_state for agent and next loop iteration
        next_state = np.reshape(next_state_raw, (1, obs_dim))

        # 5. Give (s, a, r, s', done) to the agent, which handles
        #    remembering and learning internally.
        agent.step(state, action, reward, next_state, done)

        # 6. Move to next state
        state = next_state

        # 7. Episode end: log and break
        if done:
            print(f"Run: {run}, Epsilon: {agent.exploration_rate:.3f}, Score: {steps}")
            logger.add_score(steps, run)  # writes CSV + updates score PNG
            break
```

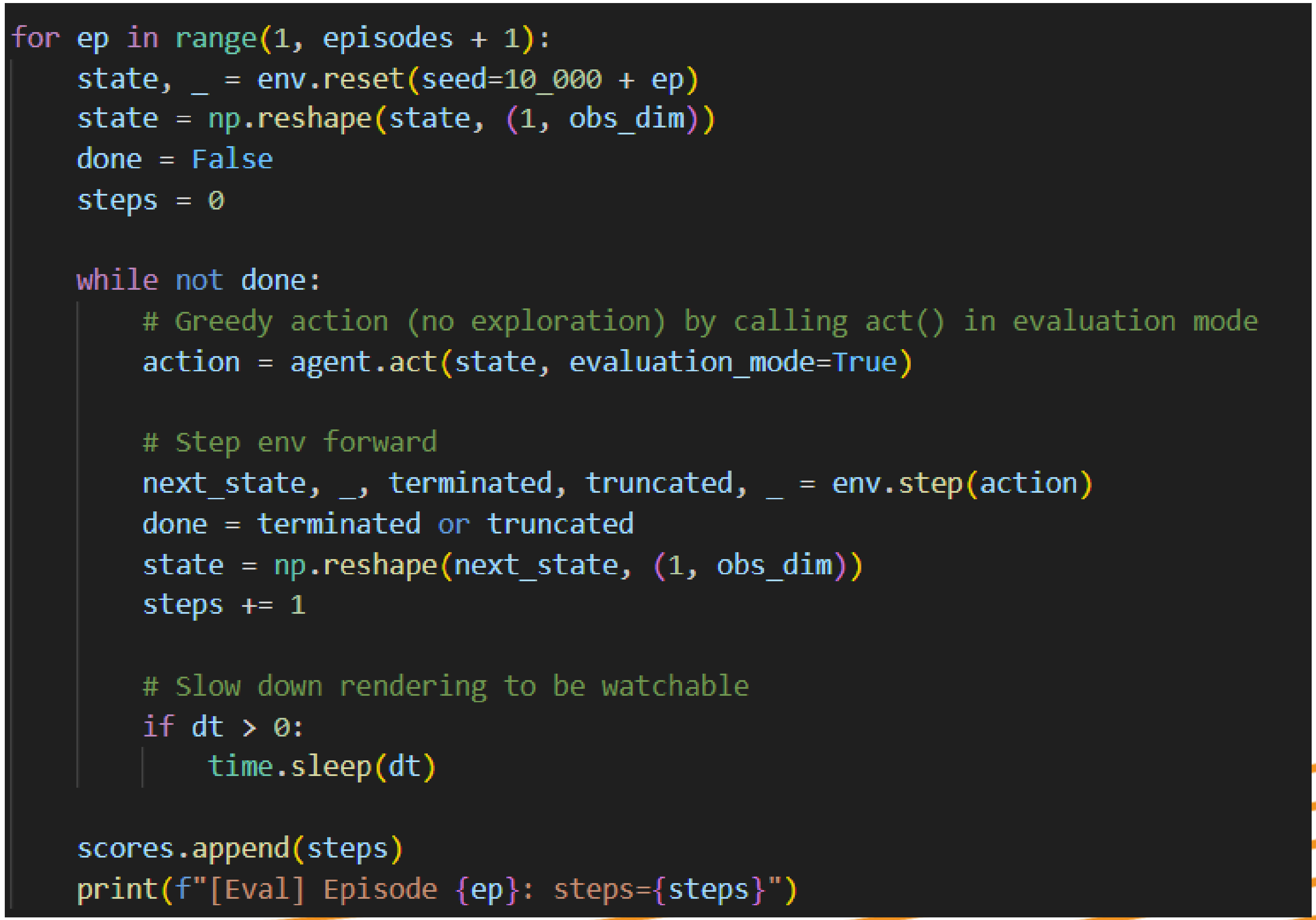
2. train.py explanation

evaluate()

- loads the model parameters from "model_path"
- uses the algorithm, runs episodes times(see **code** on the right)
- **obtains the average score**

- "render" parameter: graphical interface whether is rendered (**False** for quick evaluation or many episodes)

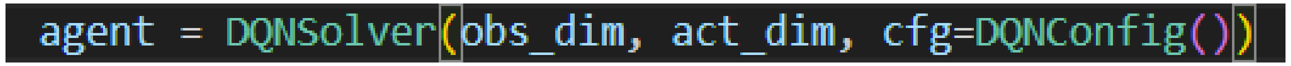
- **increasing the number of training episodes** will result in more thorough training and better performance



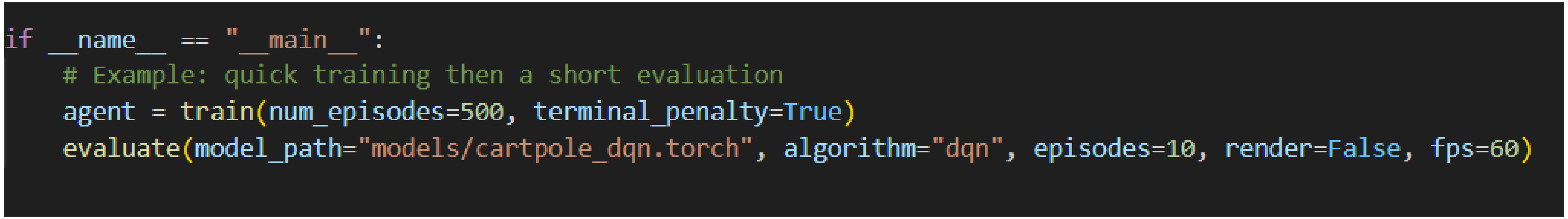
```python
for ep in range(1, episodes + 1):
    state, _ = env.reset(seed=10_000 + ep)
    state = np.reshape(state, (1, obs_dim))
    done = False
    steps = 0

    while not done:
        # Greedy action (no exploration) by calling act() in evaluation mode
        action = agent.act(state, evaluation_mode=True)

        # Step env forward
        next_state, _, terminated, truncated, _ = env.step(action)
        done = terminated or truncated
        state = np.reshape(next_state, (1, obs_dim))
        steps += 1

        # Slow down rendering to be watchable
        if dt > 0:
            time.sleep(dt)

    scores.append(steps)
    print(f"[Eval] Episode {ep}: steps={steps}")
```

2. train.py explanation

Running **python .\train.py** in the terminal will train and evaluate **the current algorithm**
If you want to change the algorithm, you need to modify agent instantiation and algorithm name **dqn**



```
agent = DQNSolver(obs_dim, act_dim, cfg=DQNConfig())
```



```
if __name__ == "__main__":
    # Example: quick training then a short evaluation
    agent = train(num_episodes=500, terminal_penalty=True)
    evaluate(model_path="models/cartpole_dqn.torch", algorithm="dqn", episodes=10, render=False, fps=60)
```

3. Adapt new algorithms

- implement a new algorithm (place it in the agent folder, e.g., agent/a2c_agent.py)
- **Five public methods that an Agent must implement**

① **__init__(self, obs_dim, act_dim, cfg=None)**

Purpose: Initialize the agent

② **act(self, state, evaluation_mode=False)**

Purpose: Returns an action (int) based on the current state.

evaluation_mode=False (When training)

③ **step(self, state, action, reward, next_state, done)**

Purpose: To learn by utilizing information obtained after interacting with the environment.

④ **save(self, path) / load(self, path)**

Purpose: To save/load model weights or policies for use in the evaluation phase.

⑤ **class [algo]Config**

Purpose: Pass in configuration parameters, you can refer to dqn.

3. Adapt new algorithms

Modify the initialization section at the top of train.py and in the evaluate function

① **Import:**

```
# from agents.cartpole_dqn import DQNSolver, DQNConfig
from agents.a2c_agent import A2CAgent, A2CConfig                    # <-- New
```

② **Initial:** In train() method

```
# agent = DQNSolver(obs_dim, act_dim, cfg=DQNConfig())
agent = A2CAgent(obs_dim, act_dim, cfg=A2CConfig())            # <-- From new
```

③ **evaluate:**

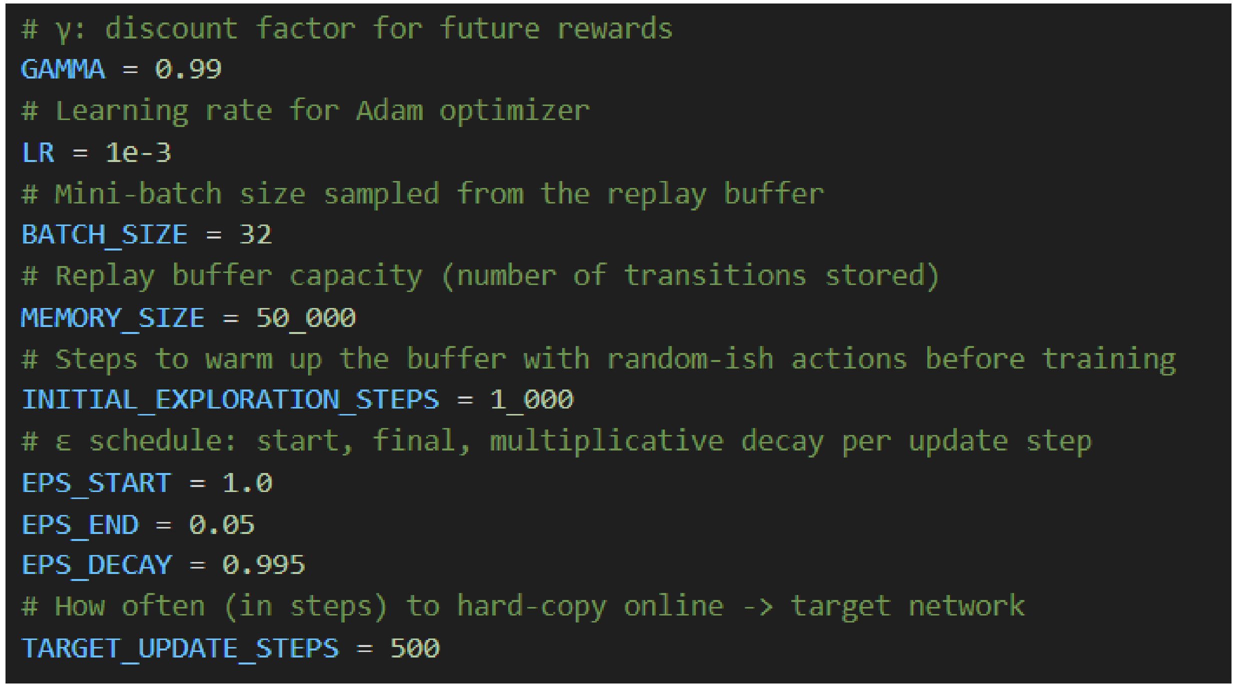
```
# (If you add PPO/A2C later, pick their agent classes by 'algorithm' here)
if algorithm.lower() == "dqn":
   agent = DQNSolver(obs_dim, act_dim, cfg=DQNConfig())
elif algorithm.lower() == "a2c":                                          # <-- New batch
   agent = A2CAgent(obs_dim, act_dim, cfg=A2CConfig())
```

4. Parameter adjustment

① search for information on **which parameters in your implemented algorithm affect model learning and convergence**

② compare values of a particular parameter

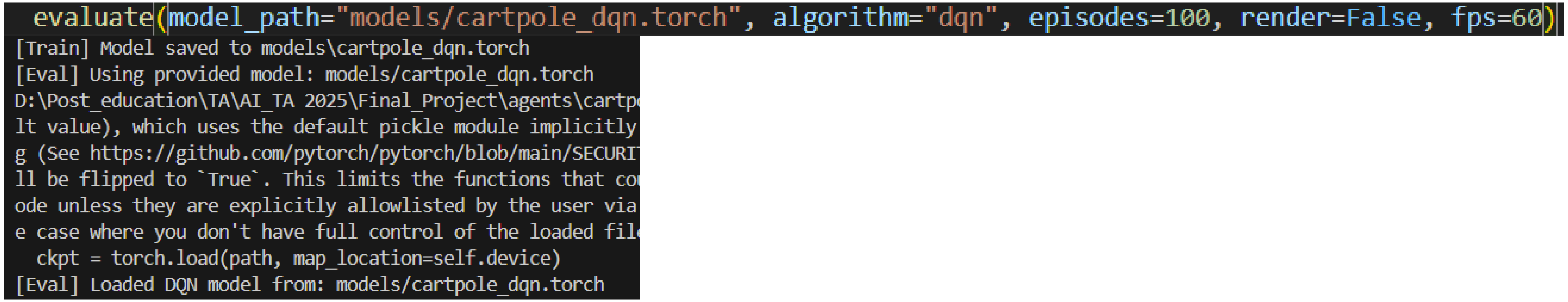
Alternatively, you can adjust the training framework to automate parameter comparisons



```
# γ: discount factor for future rewards
GAMMA = 0.99
# Learning rate for Adam optimizer
LR = 1e-3
# Mini-batch size sampled from the replay buffer
BATCH_SIZE = 32
# Replay buffer capacity (number of transitions stored)
MEMORY_SIZE = 50_000
# Steps to warm up the buffer with random-ish actions before training
INITIAL_EXPLORATION_STEPS = 1_000
# ε schedule: start, final, multiplicative decay per update step
EPS_START = 1.0
EPS_END = 0.05
EPS_DECAY = 0.995
# How often (in steps) to hard-copy online -> target network
TARGET_UPDATE_STEPS = 500
```

5. Grading

evaluated **100** episodes and **the average score** is higher than **475**



```
  evaluate(model_path="models/cartpole_dqn.torch", algorithm="dqn", episodes=100, render=False, fps=60)
```

```
[Train] Model saved to models\cartpole_dqn.torch
[Eval] Using provided model: models/cartpole_dqn.torch
D:\Post_education\TA\AI_TA 2025\Final_Project\agents\cartp
lt value), which uses the default pickle module implicitly
g (See https://github.com/pytorch/pytorch/blob/main/SECURI
ll be flipped to `True`. This limits the functions that co
ode unless they are explicitly allowlisted by the user via
e case where you don't have full control of the loaded fil
  ckpt = torch.load(path, map_location=self.device)
[Eval] Loaded DQN model from: models/cartpole_dqn.torch
```

Task Requirements:

1. Implement two algorithms other than DQN, ensuring model performance is maintained
2. Complete a task report including the implementation and understanding of the algorithms, the parameter tuning process, and analysis
3. Implement advanced algorithms by learning imitation learning, offline RL, etc., and provide analysis (**Advanced**)