AI Capstone HW2 report

111550056 陳晉祿

In this homework, I chose to work on 3 different environments from gymnasium:

1. Assault from Atari
2. MUJOCO-Walker2D
3. MUJOCO-Humanoid

GitHub link: https://github.com/Luluboy168/gymnasium_RL

Video link: https://www.youtube.com/playlist?list=PLKX1mYo4U0xmhWVFMrlRrHDUmlXVuIcKy

1. Assault-v5 (https://ale.farama.org/environments/assault/)

This is a simple shooting game. In this game, you have to control the vehicle and try to kill enemy using weapons. You can shoot at front, left and right. However, you need to wait for the weapons to cool down or you will die from the overheat.

Approach: Deep Q Network(DQN)

1. Environment Setup

The agent is trained on the ALE/Assault-v5 Atari environment. A vectorized setup with 32 parallel environments is used. Observations from each environment are preprocessed to gray scale and resize from (210, 160) to (84, 84). Four consecutive frames are stacked together to form the input state, allowing the agent to capture temporal dynamics.

2. Neural Network Architectures using tensorflow & keras

Two different Q-network architectures are explored: (Results and comparisons in experiment part)

(1) Standard DQN (Vanilla): A convolutional neural network with three convolutional layers followed by two fully connected layers. The final output layer predicts Q-values for all possible actions.

(2) Dueling DQN: This architecture separates the estimation of the state-value function and the advantage function. The network splits into two streams after the shared convolutional base: one for computing the scalar value of the state, and one for computing the advantage of each action. The two streams are combined to produce the final Q-values, improving learning efficiency in states where actions have similar effects.

Both networks use the Huber loss and are optimized using the Adam optimizer.

3. Experience Replay Buffer

A replay buffer is implemented using a Python deque to store tuples of (state, action, reward, next_state, done). Transitions are sampled uniformly to break temporal correlations during training. This allows the agent to learn from diverse past experiences and stabilize training.

4. Target Network

To further stabilize training, a target network is maintained as a copy of the primary Q-network. The target network is updated with the weights of the primary network every fixed number of steps (target_update_freq). This decouples the target Q-value computation from the current Q-network's rapidly changing parameters.

5. Action Selection: ε-Greedy Policy

The agent selects actions using an ε-greedy strategy. With probability ε, a random action is chosen to encourage exploration; otherwise, the action with the highest predicted Q-value is selected. The ε value decays gradually over time from 1.0 to a minimum of 0.1, balancing exploration and exploitation. Then I implement a re-exploration strategy, to help the agent learn more about the environment and prevent overfitting. The ε will increase to 1 – (episode / num_episodes)

every 200 episodes

6. Training Process

For each episode, the agent interacts with all vectorized environments simultaneously. Transitions are stored in the replay buffer. Once a minimum replay size is reached, the agent samples mini-batches and performs gradient updates on the primary Q-network every 4 (update_every) steps.

The target Q-value is calculated using the Bellman equation:

$$target = r + \gamma \cdot \max_{a'} Q_{target}(s', a')$$

The current Q-value is then trained to minimize the Huber loss between the predicted and target values.

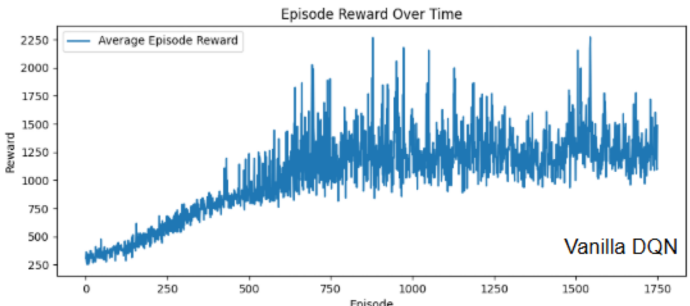
7. Logging and Model Saving

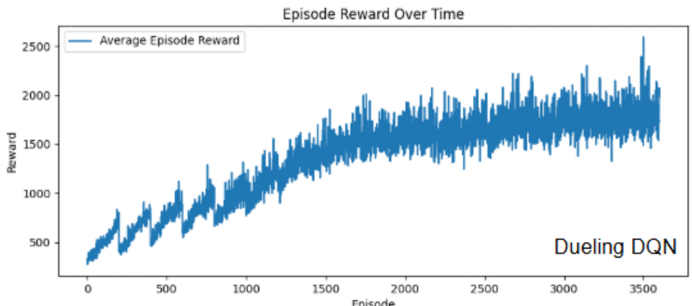
Training metrics such as episode rewards and loss values are recorded. Every 50 episodes, the model is saved to disk, and performance plots (reward vs. episode, loss vs. episode) are generated and saved as images.

Experiment & Results (Gameplay video)

1. Vanilla(standard) DQN v.s. Dueling DQN

In vanilla DQN, Single Q-network that directly estimates Q(s, a) for all actions. In dueling DQN Two separate streams: one estimates the state value V(s), and the other estimates the advantage A(s, a). The final Q-values are computed as: **Q(s, a) = V(s) + (A(s, a) − mean(A(s, ·)))**





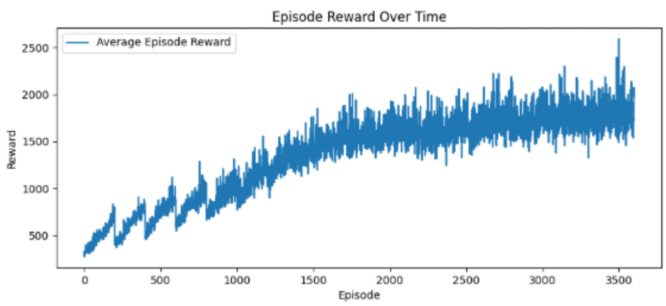
Above is the episode reward plot during training. In this plot, we can clearly see that vanilla DQN soon became highly unstable, with fluctuating rewards and an early plateau. In contrast, the Dueling DQN demonstrates a more stable and consistent learning curve, with smoother reward progression and higher final performance. (Note that in the right one, I'm also using re-exploration. I increase epsilon every 200 episodes).

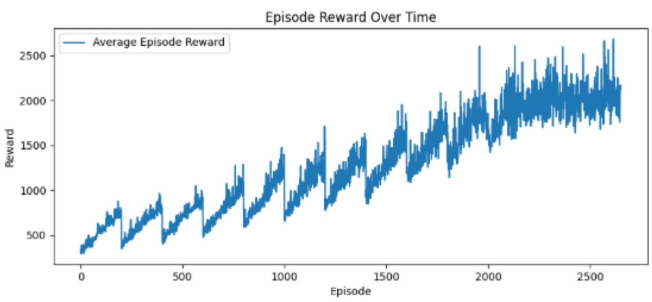
2. Wider convolution layer

Because the observation from the environment is image frame, I think wider the convolution layer might increase the agent's performance. So, I modify the model into wider CNN and I also add a Dropout layer in value stream. And the result is indeed better. The one (right) with wider convolution layer can get around 2000 avg scores, while the other one (left) can only get around 1600 avg scores.

```
# dueling DQN networks
def build_q_network(input_shape, num_actions):
    inputs = Input(shape=input_shape)

    x = Conv2D(32, 8, strides=4, activation='relu')(inputs)
    x = Conv2D(64, 4, strides=2, activation='relu')(x)
    x = Conv2D(64, 3, strides=1, activation='relu')(x)
    x = Flatten()(x)

    # Value stream
    value = Dense(512, activation='relu')(x)
    value = Dense(1)(value)

    # Advantage stream
    advantage = Dense(512, activation='relu')(x)
    advantage = Dense(num_actions)(advantage)
```

```
# dueling DQN networks
def build_q_network(input_shape, num_actions):
    inputs = Input(shape=input_shape)

    x = Conv2D(64, 8, strides=4, activation='relu')(inputs)
    x = Conv2D(128, 4, strides=2, activation='relu')(x)
    x = Conv2D(128, 3, strides=1, activation='relu')(x)
    x = Flatten()(x)

    # Value stream
    value = Dense(1024, activation='relu')(x)
    value = Dropout(0.1)(value)
    value = Dense(1)(value)

    # Advantage stream
    advantage = Dense(1024, activation='relu')(x)
    advantage = Dropout(0.1)(advantage)
    advantage = Dense(num_actions)(advantage)
```





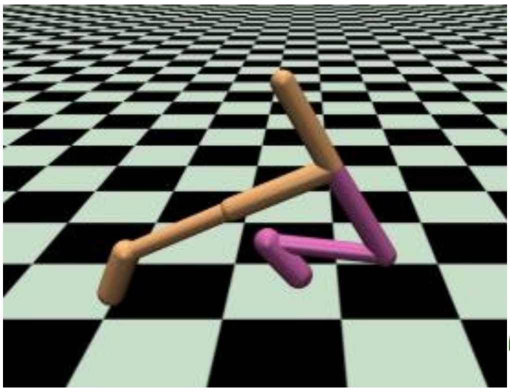
2. Walker2D-v5 (https://gymnasium.farama.org/environments/mujoco/walker2d/)

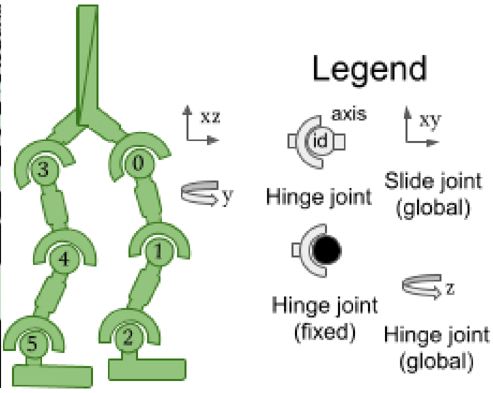
We're training a virtual robot (Walker2D) to walk stably and efficiently, like a bipedal creature balancing and moving forward. The robot lives inside a simulated environment called Walker2d-v5, and we use reinforcement learning to train an agent the can make the robot move forward.

Approach: Proximal policy optimization(PPO)

1. Environment setup

This agent is trained on Walker2D-v5 in Mujoco environment. A vectorized setup with 32 parallel environments is used. This environment has obs_dim=17 and act_dim=6 (controls over joints).





2. Model

In this approach, I use pytorch to build a Actor-Critic network. The model is placed in "model.py". The ActorCritic model has:

- A shared base: fully connected + ReLu
- Actor head: Outputs the mean of a Gaussian distribution over actions.
- log_std: A learnable parameter representing the log of standard deviation (shared across observations).
- Critic head: Outputs the estimated state value.
- Output: mean of action distribution + standard deviation + critic value estimate

3. Main training loop

(1) Rollout Collection

The agent interacts with environments for 2048 steps. And actions are sampled from a normal distribution defined by the model. Then store observations, actions, reward, dones, logprobs, and value predictions.

(2) Generalized Advantage Estimation(GAE) Calculation

GAE is used to compute the advantage function and returns, which balances bias and variance using gamma (discount factor) and lambda

(3) Data Flattening & Normalization

Move collected rollout data to GPU. Normalize and clip advantages to prevent instability.

(4) PPO update

During training, the PPO algorithm updates the policy over several epochs. For each epoch, the batch is randomly shuffled and divided into smaller mini-batches. Each mini-batch is used to compute the new policy's performance: how different its current behavior is from the one used during data collection, and how good its value predictions are. PPO ensures that policy updates are conservative using a clipping technique—this helps the model improve steadily without making drastic changes that could ruin previously learned behavior. The total loss combines three things: the policy loss (how well the policy is improving), the value loss (how accurate the critic is), and an entropy bonus (which encourages the policy to stay exploratory). After computing the gradients of this total loss, the optimizer updates the network parameters.

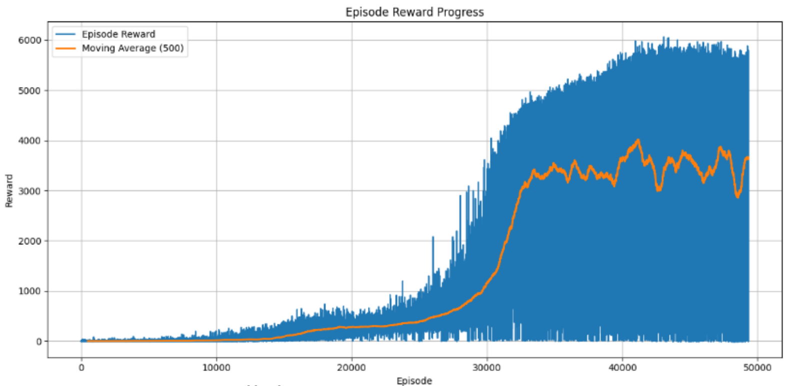
4. Logging and Model Saving

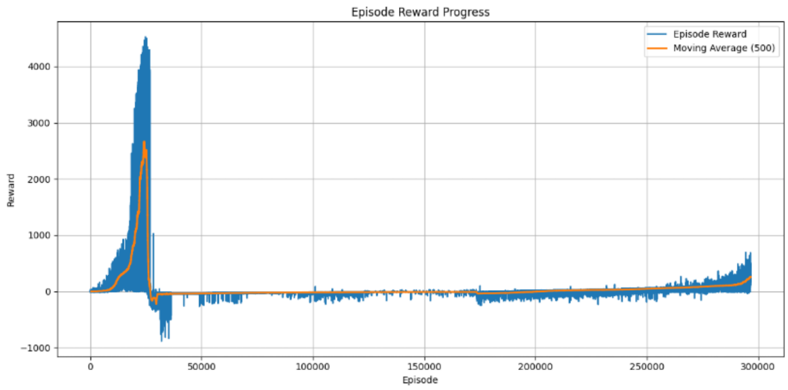
Training metrics such as episode rewards and loss values are recorded. Every 50 episodes, the model is saved to disk, and performance plots (reward vs. episode, loss vs. episode) are generated and saved as images.

Experiment & Results ([Gameplay video])

1. Different parameters

In this environment, I try to experiment the effect of different parameters. And I encounter a cool thing during training, I got a gradian explosion after few hundred updates. This might because the environment is much more complex, so the agent might get unstable rewards every episode. To solve this potential problem, I reduce learning rate(3e-4→1e-4) and clip epsilon(0.2→0.1). It turns out that the model trains very good. And the agent learns pretty fast. As you can see from the plots below, the right one got exploded.





2. Parallel environment

In this experiment, I try using single environment and vectorized environments. Turns out that vectorized environment is faster than using just a single one. Using 32 environments only needs 10 times more time than using single one, but we can get 32 times more episodes per update. And the full training only takes about one and a half hour (with i7-12700 & RTX3080ti) which is pretty amazing short amount of time. Below is the running time of each update, the algorithm takes about 250 updates to converge.

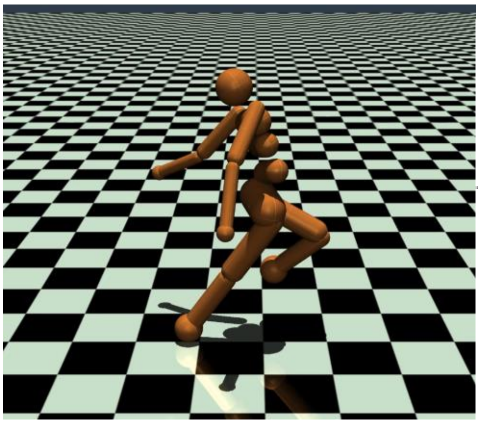
```
Update     1 | Avg Loss:  2.039 | Episodes: 3014 | Recent Avg Reward: -1.092 | Time: 22.91
Update     2 | Avg Loss:  3.424 | Episodes: 5779 | Recent Avg Reward:  0.690 | Time: 21.98
Update     3 | Avg Loss:  5.863 | Episodes: 8105 | Recent Avg Reward: 12.030 | Time: 19.58
Update     4 | Avg Loss:  9.753 | Episodes: 10054 | Recent Avg Reward: 15.649 | Time: 20.79
Update     5 | Avg Loss: 15.993 | Episodes: 11608 | Recent Avg Reward: 12.385 | Time: 23.37
```

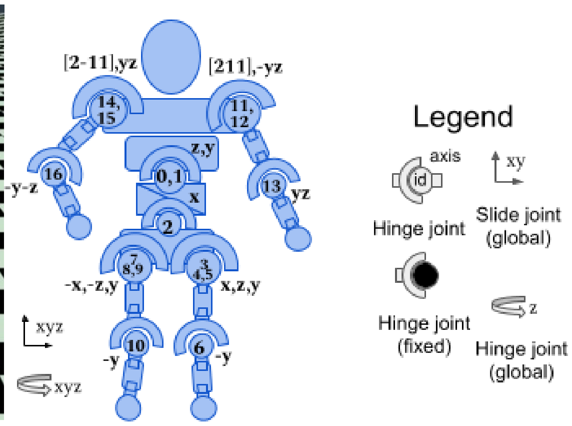
32 env↑ ↓single env

```
Update     1 | Avg Loss:  1.572 | Episodes:  119 | Recent Avg Reward:  1.198 | Time: 2.11
Update     2 | Avg Loss:  2.135 | Episodes:  235 | Recent Avg Reward: -0.752 | Time: 1.84
Update     3 | Avg Loss:  3.167 | Episodes:  338 | Recent Avg Reward:  4.217 | Time: 1.83
Update     4 | Avg Loss:  5.415 | Episodes:  415 | Recent Avg Reward:  5.629 | Time: 1.83
Update     5 | Avg Loss:  7.438 | Episodes:  483 | Recent Avg Reward: 12.929 | Time: 1.83
```

3. More complex environment

I have heard that PPO is a very strong reinforcement learning algorithm. So, I wonder that if I use the exact same algorithm with a much more complex environment. Can PPO handle it? In this experiment, I use the same code with environment Humanoid-v5. This environment is much more complex with 384 observation space and 17 action space. In this implementation, I only changed envs = gym.make_vec("Walker2d-v5", num_envs=num_envs, vectorization_mode="sync") to envs = gym.make_vec("Humanoid-v5", num_envs=num_envs, vectorization_mode="sync"). Other parts of algorithm remain the same. The result is not bad actually, the humanoid robot can actually balance itself and move forward, getting a pretty good score from the environment.





Gameplay video: https://www.youtube.com/watch?v=zPHR8DiUPH0

Discussion

1. Learned from this project:
   In this project, I gained a solid understanding of various reinforcement learning algorithms, including DQN, Dueling DQN, and PPO. More importantly, I learned how to build a reinforcement learning agent from scratch. This hands-on experience helped me grasp not only the theoretical concepts but also the practical challenges involved in implementation. In fact, it took me over a week just to get the model to start learning properly—reinforcement learning models are indeed much harder to train. This experience reinforced the saying, “Machine learning is more about practice than theory.” Through these experiments, I also learned the fundamentals of hyperparameter tuning and developed a deeper intuition for why and how these models work. In addition, I realized how powerful RL is. It can easily deal with new environments and perform really well.
2. Remaining questions:
   In Atari environment, I get around 2200 scores at the end of this project. But if we search this game online, we can find that agents trained by others can get a much higher score. This means that though my agent is powerful enough, there is still lots of spaces for improvement. Maybe I can try more different algorithms such as A3C or spend more time tuning my current model. The result of the Atari game can still get better.
   In Mujoco sim environment, my managed to get the highest score. However, if we watch that video clip, we can see that its movements are strange. I think there might be ways to solve this problem and make it look more nature. In the future, if I have time, I think I can use custom reward instead of reward given by OpenAI gym environment to see if the agent can be even better.

References

- https://gymnasium.farama.org/
- https://github.com/Harsha1997/DeepLearning-in-Atari-Games
- https://ithelp.ithome.com.tw/articles/10225812
- https://www.youtube.com/playlist?list=PLJV_el3uVTsODxQFgzMzPLa16h6B8kWM_
- https://github.com/PawelMlyniec/Walker-2D
- https://hackmd.io/@RL666/rJUDS6K05
- https://medium.com/intro-to-artificial-intelligence/the-actor-critic-reinforcement-learning-algorithm-c8095a655c14
- https://andy6804tw.github.io/2022/04/03/python-video-save/

Appendix

Here's the (training) code screenshots. To view the full code, please check out my [GitHub Repo].

1. Assault-v5

```python
import gymnasium as gym
import numpy as np
import tensorflow as tf
from tensorflow.keras import Input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Conv2D, Dense, Flatten, Lambda, Dropout
import matplotlib.pyplot as plt
import random
import ale_py
import os
import time
from collections import deque
from utils import preprocess_observation

gpus = tf.config.list_physical_devices('GPU')
if gpus:
    try:
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
    except RuntimeError as e:
        print("Error setting memory growth:", e)

# Create necessary directories
os.makedirs("models", exist_ok=True)
os.makedirs("figures", exist_ok=True)

# Build vectorized environment
num_envs = 32
gym.register_envs(ale_py)
envs = gym.make_vec("ALE/Assault-v5", num_envs=num_envs, vectorization_mode="async")

# Hyperparameters
input_shape = (84, 84, 4)  # grayscale, stacked
num_actions = envs.single_action_space.n
gamma = 0.99
batch_size = 1024
replay_capacity = 350000
min_replay_size = 10000
epsilon = 1.0
epsilon_min = 0.1
epsilon_decay = 0.999995
target_update_freq = 1000
update_every = 4

# dueling DQN networks
def build_q_network(input_shape, num_actions):
    inputs = Input(shape=input_shape)

    x = Conv2D(64, 8, strides=4, activation='relu')(inputs)
    x = Conv2D(128, 4, strides=2, activation='relu')(x)
    x = Conv2D(128, 3, strides=1, activation='relu')(x)
    x = Flatten()(x)

    # Value stream
    value = Dense(1024, activation='relu')(x)
    value = Dropout(0.1)(value)
    value = Dense(1)(value)

    # Advantage stream
    advantage = Dense(1024, activation='relu')(x)
    advantage = Dropout(0.1)(advantage)
    advantage = Dense(num_actions)(advantage)

    # Combine streams with Lambda to safely handle symbolic tensors
    def combine_streams(inputs):
        value, advantage = inputs
        return value + (advantage - tf.reduce_mean(advantage, axis=1, keepdims=True))

    q_values = Lambda(combine_streams, output_shape=(int(num_actions),))([value, advantage])

    model = Model(inputs=inputs, outputs=q_values)
    model.compile(optimizer=tf.keras.optimizers.Adam(0.00025), loss='huber')
    return model

primary_network = build_q_network(input_shape, num_actions)
target_network = build_q_network(input_shape, num_actions)
target_network.set_weights(primary_network.get_weights())

# Replay buffer
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def add_batch(self, states, actions, rewards, next_states, dones):
        for s, a, r, ns, d in zip(states, actions, rewards, next_states, dones):
            self.buffer.append((s, a, np.float32(r), ns, np.float32(d)))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = map(np.array, zip(*batch))
        return states, actions, rewards, next_states, dones

    def __len__(self):
        return len(self.buffer)

replay_buffer = ReplayBuffer(replay_capacity)

@tf.function
def train_step(states, actions, rewards, next_states, dones):
    rewards = tf.cast(rewards, tf.float32)
    dones = tf.cast(dones, tf.float32)
    next_q_values = target_network(next_states)
    max_next_q = tf.reduce_max(next_q_values, axis=1)
    target_q = rewards + (1. - dones) * gamma * max_next_q
    with tf.GradientTape() as tape:
        q_values = primary_network(states)
        indices = tf.stack([tf.range(tf.shape(actions)[0]), tf.cast(actions, tf.int32)], axis=1)
        q_selected = tf.gather_nd(q_values, indices)
        loss = tf.keras.losses.Huber()(target_q, q_selected)
    grads = tape.gradient(loss, primary_network.trainable_variables)
    primary_network.optimizer.apply_gradients(zip(grads, primary_network.trainable_variables))
    return loss

# Training loop
total_steps = 0
num_episodes = 5000
loss_history = []
reward_history = []

for episode in range(num_episodes):
    start_time = time.time()
    obs, _ = envs.reset()
    obs = preprocess_observation(obs)
    frames = np.repeat(obs, 4, axis=-1)  # (8, 84, 84, 4)

    done_flags = np.zeros(num_envs, dtype=bool)
    episode_rewards = np.zeros(num_envs)
    episode_losses = []

    # re-explore
    if((episode) % 200 == 0):
        epsilon = max(1.0 - (episode / num_episodes) * 2, epsilon_min)

    while not np.all(done_flags):
        total_steps += 1

        # ε-greedy action selection
        q_values = primary_network(frames)
        greedy_actions = np.argmax(q_values.numpy(), axis=1)
        random_actions = np.random.randint(num_actions, size=num_envs)
        actions = np.where(np.random.rand(num_envs) < epsilon, random_actions, greedy_actions)

        next_obs, rewards, terminations, truncations, _ = envs.step(actions)
        dones = np.logical_or(terminations, truncations)
        next_obs = preprocess_observation(next_obs)
        next_frames = np.concatenate([frames[..., 1:], next_obs], axis=-1)  # (8, 84, 84, 4)

        replay_buffer.add_batch(frames, actions, rewards, next_frames, dones)
        frames = next_frames
        done_flags = np.logical_or(done_flags, dones)
        episode_rewards += rewards

        if len(replay_buffer) >= min_replay_size and total_steps % update_every == 0:
            states_b, actions_b, rewards_b, next_states_b, dones_b = replay_buffer.sample(batch_size)
            loss = train_step(states_b, actions_b, rewards_b, next_states_b, dones_b)
            episode_losses.append(loss.numpy())

            if total_steps % target_update_freq == 0:
                target_network.set_weights(primary_network.get_weights())

        epsilon = max(epsilon * epsilon_decay, epsilon_min)

    avg_loss = np.mean(episode_losses) if episode_losses else 0
    loss_history.append(avg_loss)
    reward_history.append(np.mean(episode_rewards))
    elapsed = time.time() - start_time
    print(f"Episode {episode + 1}, Reward: {np.mean(episode_rewards):.2f}, Epsilon: {epsilon:.3f}, Total steps: {total_steps}, Time elapsed: {elapsed}")

    if (episode + 1) % 50 == 0:
        model_path = f"models/dqn_assault_model_{episode + 1}.keras"
        primary_network.save(model_path)
        print(f"Saved model to {model_path}")
        # Plotting
        plt.figure(figsize=(10, 4))
        plt.plot(loss_history, label="Avg Loss per Episode")
        plt.xlabel("Episode")
        plt.ylabel("Loss")
        plt.title("Training Loss Over Episodes")
        plt.legend()
        plt.savefig("figures/loss_over_episodes.png")

        plt.figure(figsize=(10, 4))
        plt.plot(reward_history, label="Average Episode Reward")
        plt.xlabel("Episode")
        plt.ylabel("Reward")
        plt.title("Episode Reward Over Time")
        plt.legend()
        plt.savefig("figures/reward_over_episodes.png")

envs.close()
```

2. Walker2D & Humanoid

```python
import gymnasium as gym
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import time

from model import ActorCritic

# Use a non-interactive backend so plots don't show automatically
plt.switch_backend('Agg')

# ---------- Environment Setup ----------

num_envs = 32
# Create a vectorized environment using gym.make_vec (without any wrappers)
envs = gym.make_vec("Walker2d-v5", num_envs=num_envs, vectorization_mode="sync")

obs_space = envs.single_observation_space
act_space = envs.single_action_space

obs_dim = obs_space.shape[0]
act_dim = act_space.shape[0]

episode_rewards = np.zeros(num_envs)  # one per sub-environment
all_episode_rewards = []
episode_count = 0


device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = ActorCritic(obs_dim, act_dim).to(device)
optimizer = optim.Adam(model.parameters(), lr=3e-4)

# ---------- Hyperparameters ----------
total_updates    = 1000       # total training updates
steps_per_update = 2048       # total timesteps per update
ppo_epochs       = 10         # PPO epochs per update
mini_batch_size  = 128        # mini-batch size for PPO updates
gamma            = 0.99       # discount factor
gae_lambda       = 0.95       # GAE lambda
clip_epsilon     = 0.1        # PPO clip parameter
value_loss_coef  = 0.25       # weight for the value loss
entropy_coef     = 0.01       # weight for the entropy bonus

# Storage for logging and plotting
loss_history      = []
plot_updates      = []
avg_reward_history= []
next_plot_episode = 50

# ---------- Utility: Generalized Advantage Estimation (GAE) ----------
def compute_gae(rewards, values, dones, next_value, gamma=0.99, lam=0.95):
    T = len(rewards)
    advantages = np.zeros(T, dtype=np.float32)
    lastgaelam = 0
    for t in reversed(range(T)):
        if t == T - 1:
            next_non_terminal = 1.0 - dones[-1]
            next_val = next_value
        else:
            next_non_terminal = 1.0 - dones[t+1]
            next_val = values[t+1]
        delta = rewards[t] + gamma * next_val * next_non_terminal - values[t]
        lastgaelam = delta + gamma * lam * next_non_terminal * lastgaelam
        advantages[t] = lastgaelam
    returns = advantages + values
    return advantages, returns

# ---------- Main Training Loop ----------
obs, _ = envs.reset()
for update in range(total_updates):
    start_time = time.time()
    # Buffers for rollout
    obs_buffer      = []
    actions_buffer  = []
    logprobs_buffer = []
    rewards_buffer  = []
    dones_buffer    = []
    values_buffer   = []

    # Collect rollout of fixed steps_per_update
    for step in range(steps_per_update):
        obs_tensor = torch.FloatTensor(obs).to(device)
        with torch.no_grad():
            mean, std, value = model(obs_tensor)
        dist = torch.distributions.Normal(mean, std)
        action = dist.sample()
        logprob = dist.log_prob(action).sum(axis=-1)

        # Store step data
        obs_buffer.append(obs)
        actions_buffer.append(action.cpu().numpy())
        logprobs_buffer.append(logprob.cpu().numpy())
        values_buffer.append(value.cpu().numpy())

        # Step the environments
        actions_np = action.cpu().numpy()
        next_obs, rewards, terminations, truncations, infos = envs.step(actions_np)
        rewards_buffer.append(rewards)
        dones_buffer.append(terminations | truncations)

        # Update per-environment cumulative rewards
        episode_rewards += rewards
        # If an environment is done or truncated, record its episodic reward and reset that counter.
        for i in range(num_envs):
            if terminations[i] or truncations[i]:
                all_episode_rewards.append(episode_rewards[i])
                episode_count += 1
                episode_rewards[i] = 0  # reset the cumulative reward

        obs = next_obs

    # Convert buffers to numpy arrays
    obs_buffer      = np.array(obs_buffer)
    actions_buffer  = np.array(actions_buffer)
    logprobs_buffer = np.array(logprobs_buffer)
    rewards_buffer  = np.array(rewards_buffer)
    dones_buffer    = np.array(dones_buffer, dtype=np.float32)
    values_buffer   = np.array(values_buffer)

    # get value estimates for the last observations
    obs_tensor = torch.FloatTensor(obs).to(device)
    with torch.no_grad():
        _, _, next_values = model(obs_tensor)
    next_values = next_values.cpu().numpy()

    # Compute advantages and returns for each sub-environment separately.
    advantages_buffer = np.zeros_like(rewards_buffer)
    returns_buffer    = np.zeros_like(rewards_buffer)
    for env_idx in range(num_envs):
        adv, ret = compute_gae(
            rewards=rewards_buffer[:, env_idx],
            values=values_buffer[:, env_idx],
            dones=dones_buffer[:, env_idx],
            next_value=next_values[env_idx],
            gamma=gamma,
            lam=gae_lambda
        )
        advantages_buffer[:, env_idx] = adv
        returns_buffer[:, env_idx] = ret

    # Flatten the trajectory
    batch_obs       = torch.FloatTensor(obs_buffer.reshape(-1, obs_dim)).to(device)
    batch_actions   = torch.FloatTensor(actions_buffer.reshape(-1, act_dim)).to(device)
    batch_logprobs  = torch.FloatTensor(logprobs_buffer.reshape(-1)).to(device)
    batch_returns   = torch.FloatTensor(returns_buffer.reshape(-1)).to(device)
    batch_advantages= torch.FloatTensor(advantages_buffer.reshape(-1)).to(device)
    # Normalize advantages
    batch_advantages = (batch_advantages - batch_advantages.mean()) / (batch_advantages.std() + 1e-8)
    batch_advantages = torch.clamp(batch_advantages, -10, 10)

    # ---------- PPO Policy Update ----------
    total_loss = 0
    batch_size = batch_obs.shape[0]
    indices = np.arange(batch_size)

    for epoch in range(ppo_epochs):
        np.random.shuffle(indices)
        for start in range(0, batch_size, mini_batch_size):
            end = start + mini_batch_size
            mb_idx = indices[start:end]

            minibatch_obs       = batch_obs[mb_idx]
            minibatch_actions   = batch_actions[mb_idx]
            minibatch_logprobs  = batch_logprobs[mb_idx]
            minibatch_returns   = batch_returns[mb_idx]
            minibatch_advantages= batch_advantages[mb_idx]

            mean, std, values = model(minibatch_obs)
            dist = torch.distributions.Normal(mean, std)
            new_logprobs = dist.log_prob(minibatch_actions).sum(axis=-1)
            entropy = dist.entropy().sum(axis=-1).mean()

            ratio = torch.exp(new_logprobs - minibatch_logprobs)
            surr1 = ratio * minibatch_advantages
            surr2 = torch.clamp(ratio, 1.0 - clip_epsilon, 1.0 + clip_epsilon) * minibatch_advantages
            policy_loss = -torch.min(surr1, surr2).mean()
            value_loss = ((minibatch_returns - values) ** 2).mean()
            loss = policy_loss + value_loss_coef * value_loss - entropy_coef * entropy

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

    avg_loss = total_loss / (ppo_epochs * (batch_size // mini_batch_size))
    loss_history.append(avg_loss)

    # Print
    recent_avg_reward = (np.mean(all_episode_rewards[-10:])
                         if len(all_episode_rewards) >= 10 else
                         np.mean(all_episode_rewards) if all_episode_rewards else 0)
    time_elapsed = time.time() - start_time
    print(f"Update {update+1:4d} | Avg Loss: {avg_loss:6.3f} | Episodes: {episode_count:4d} | Recent Avg Reward: {recent_avg_reward:6.3f} | Time: {time_elapsed:.2f}")

    # Save model every 50 updates and save the training plot
    if (update + 1) % 50 == 0:
        model_path = f"models/ppo_walker2d_update_{update+1}.pth"
        torch.save(model.state_dict(), model_path)
        print(f"Model saved at update {update+1} as '{model_path}'")

        # Plot
        plt.figure(figsize=(12, 6))
        plt.plot(all_episode_rewards, label="Episode Reward")

        # Moving average
        if len(all_episode_rewards) >= 500:
            rolling = pd.Series(all_episode_rewards).rolling(window=500).mean()
            plt.plot(rolling, label="Moving Average (500)", linewidth=2)
        plt.xlabel("Episode")
        plt.ylabel("Reward")
        plt.title("Episode Reward Progress")
        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.savefig("figures/episode_rewards.png")
        plt.close()

        plt.figure(figsize=(10, 5))
        updates_axis = np.arange(1, len(loss_history) + 1)
        plt.plot(updates_axis, loss_history, marker='o')
        plt.xlabel("Update")
        plt.ylabel("Average Loss")
        plt.title("Loss Progress")
        plt.grid(True)
        plt.tight_layout()
        plt.savefig("figures/loss_plot.png")
        plt.close()

# ---------- Final Save ----------
torch.save(model.state_dict(), "models/ppo_walker2d_model.pth")
print("Final model saved as 'ppo_walker2d_model.pth'")
```