LunarLanderDQN2025Assignment

March 10, 2025

0.1 Lunar lander with DQN-style neural function approximator using PyTorch

0.1.1 Christian Igel, 2025

If you have suggestions for improvement, let me know.

I took inspiration from https://github.com/udacity/deep-learning/blob/master/reinforcement/Q-learning-cart.ipynb.

Imports:

[1]:
```
import gymnasium as gym

from tqdm.notebook import tqdm  # Progress bar

import torch
import torch.nn as nn
import torch.nn.functional as F

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Create the game environment (you need the `gym` package):

[2]:
```
env_visual = gym.make('LunarLander-v3', render_mode="human")
action_size = 4
state_size = 8
```

Let's just test the environment first:

[3]:
```
test_episodes = 0
for _ in range(test_episodes):
    R = 0
    state, _ = env_visual.reset()  # Environment starts in a random state, cart␣
 ↪and pole are moving
    print("initial state:", state)
    while True:  # Environment sets "truncated" to true after 500 steps
        env_visual.render()
        state, reward, terminated, truncated, _ = env_visual.step(env_visual.
 ↪action_space.sample()) #  Take a random action
```

```
        R += reward  # Accumulate reward
        if terminated or truncated:
            print("return: ", R)
            env_visual.reset()
            break
```

[4]:
```
#env.close()  # Closes the visualization window
```

Define Q network architecture:

[5]:
```
class QNetwork(nn.Module):
    def __init__(self, state_size=8, action_size=4, hidden_size=10, bias=True):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(state_size, hidden_size, bias)
        self.fc2 = nn.Linear(hidden_size, hidden_size, bias)
        self.output_layer = nn.Linear(hidden_size + state_size, action_size,␣
 ↪bias)

    def forward(self, x_input):
        x = F.tanh(self.fc1(x_input))
        x = F.tanh(self.fc2(x))
        x = torch.cat((x_input, x), dim=1)
        x = self.output_layer(x)
        return x
```

Data structure for storing experiences:

[6]:
```
from collections import deque
class Memory():
    def __init__(self, max_size = 1000):
        self.buffer = deque(maxlen=max_size)

    def add(self, experience):
        self.buffer.append(experience)

    def sample(self, batch_size):
        idx = np.random.choice(np.arange(len(self.buffer)),
                               size=batch_size,
                               replace=False)
        return [self.buffer[ii] for ii in idx]
```

Define basic constants:

[7]:
```
train_episodes = 400          # Max number of episodes to learn from
gamma = 0.99                  # Future reward discount
learning_rate = 0.001         # Q-network learning rate
tau = .01                     # learning rate for target network

# Exploration parameters
```

```python
explore_start = 1.0            # Exploration probability at start
explore_stop = 0.0001          # Minimum exploration probability
decay_rate = 0.05              # Exponential decay rate for exploration prob

# Network parameters
hidden_size = 64               # Number of units in each Q-network hidden layer

# Memory parameters
memory_size = 10000            # Memory capacity
batch_size = 128               # Experience mini-batch size
pretrain_length = batch_size   # Number experiences to pretrain the memory

log_path = "/tmp/deep_Q_network"
```

Instantiate network:

[8]:
```python
mainQN = QNetwork(hidden_size=hidden_size)
print(mainQN)
```

```
QNetwork(
  (fc1): Linear(in_features=8, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=64, bias=True)
  (output_layer): Linear(in_features=72, out_features=4, bias=True)
)
```

Initialize the experience memory:

[9]:
```python
# Initialize the simulation
env = gym.make('LunarLander-v3')
state = env.reset()[0]

memory = Memory(max_size=memory_size)

# Make a bunch of random actions and store the experiences
for _ in tqdm(range(pretrain_length)):
    # Make a random action
    action = env.action_space.sample()
    next_state, reward, terminated, truncated, _ = env.step(action)

    if terminated or truncated:
        # The simulation fails, so no next state
        next_state = np.zeros(state.shape)
        # Add experience to memory
        memory.add((state, action, reward, next_state))

        # Start new episode
        env.reset()
        # Take one random step to get the pole and cart moving
```

```
        state, reward, terminated, truncated, _ = env.step(env.action_space.
↪sample())
    else:
        # Add experience to memory
        memory.add((state, action, reward, next_state))
        state = next_state
```

```
  0%|          | 0/128 [00:00<?, ?it/s]
```

Now train with experiences:

```
[10]: total_reward_list = []  # Returns for the individual episodes

optimizer = torch.optim.AdamW(mainQN.parameters(), lr=learning_rate) # AdamW␣
↪uses weight decay by default
loss_fn = torch.nn.MSELoss()

for ep in range(train_episodes):
    total_reward = 0  # Return / accumulated rewards
    state = env.reset()[0]  # Reset and get initial state
    while True:
        # Explore or exploit
        explore_p = explore_stop + (explore_start - explore_stop)*np.
↪exp(-decay_rate*ep)
        if explore_p > np.random.rand():
            # Pick a random action
            action = env.action_space.sample()
        else:
            # Get action from Q-network
            state_tensor = torch.from_numpy(np.resize(state, (1, state_size)).
↪astype(np.float32))
            Qs = mainQN(state_tensor)
            action = torch.argmax(Qs).item()

        # Take action, get new state and reward
        next_state, reward, terminated, truncated, _ = env.step(action)

        total_reward += reward  # Return / accumulated rewards

        if terminated or truncated:
            # Episode ends because of failure, so no next state
            next_state = np.zeros(state.shape)

            print('Episode: {}'.format(ep), 'Total reward: {}'.
↪format(total_reward),
                  'Training loss: {:.4f}'.format(loss), 'Explore P: {:.4f}'.
↪format(explore_p))
            total_reward_list.append((ep, total_reward))
```

```
            # Add experience to memory
            memory.add((state, action, reward, next_state))
            break; # End of episode
        else:
            # Add experience to memory
            memory.add((state, action, reward, next_state))
            state = next_state

        # Sample mini-batch from memory
        batch = memory.sample(batch_size)
        next_states_np = np.array([each[3] for each in batch], dtype=np.float32)
        next_states = torch.as_tensor(next_states_np)  # as_tensor does not␣
↪copy the data
        rewards     = torch.as_tensor(np.array([each[2] for each in batch],␣
↪dtype=np.float32))
        states      = torch.as_tensor(np.array([each[0] for each in batch],␣
↪dtype=np.float32))
        actions     = torch.as_tensor(np.array([each[1] for each in batch]))

        # Compute Q values for all actions in the new state
        target_Qs = mainQN(next_states)

        # Set target_Qs to 0 for states where episode ended because of failure
        episode_ends = (next_states_np == np.zeros(states[0].shape)).all(axis=1)
        target_Qs[episode_ends] = torch.zeros(action_size)

        # Compute targets

        # Network learning starts here
        optimizer.zero_grad()

        # Compute the Q values of the actions taken
        main_Qs = mainQN(states)  # Q values for all action in each state
        Q = torch.gather(main_Qs, 1, actions.unsqueeze(-1)).squeeze()  # Only␣
↪the Q values for the actions taken

        # Gradient-based update
        loss = loss_fn(Q, y)
        loss.backward()
        optimizer.step()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[10], line 67
     64 Q = torch.gather(main_Qs, 1, actions.unsqueeze(-1)).squeeze()  # Only␣
 ↪the Q values for the actions taken
```

```
     66 # Gradient-based update
---> 67 loss = loss_fn(Q, y)
     68 loss.backward()
     69 optimizer.step()

NameError: name 'y' is not defined
```

Save policy network:

[ ]:
```
torch.save(mainQN, log_path)
```

Plot learning process:

[ ]:
```
# Moving average for smoothing plot
def running_mean(x, N):
    cumsum = np.cumsum(np.insert(x, 0, x[0]*np.ones(N)))
    return (cumsum[N:] - cumsum[:-N]) / N

eps, rews = np.array(total_reward_list).T
smoothed_rews = running_mean(rews, 10)

plt.plot(eps, smoothed_rews)
plt.grid()
plt.plot(eps, rews, color='grey', alpha=0.3)
plt.xlabel('Episode')
plt.ylabel('Accumulated Reward')
plt.savefig('deepQ.pdf')
```

Evaluate stored policy:

[ ]:
```
testQN = torch.load(log_path)

test_episodes = 5

for ep in range(test_episodes):
    state = env_visual.reset()[0]
    print("initial state:", state)
    R = 0
    while True:
        # Get action from Q-network
        # Hm, the following line could perhaps be more elegant ...
        state_tensor = torch.from_numpy(np.resize(state, (1, state_size)).
↪astype(np.float32))
        Qs = testQN(state_tensor)
        action = torch.argmax(Qs).item()

        # Take action, get new state and reward
        next_state, reward, terminated, truncated, _ = env_visual.step(action)
```

```
        R += reward

        if terminated or truncated:
            print("reward:", R)
            break
        else:
            state = next_state
```

[ ]:

```

```