Tutorial 6

March 15, 2024

1 Tutorial: Actor Critic Implementation

[1]:
```
#Import required libraries

import argparse
import gym
import numpy as np
from itertools import count
from collections import namedtuple

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Categorical

import matplotlib.pyplot as plt
```

[2]:
```
#Set constants for training
seed = 543
log_interval = 10
gamma = 0.99

env = gym.make('CartPole-v1')
env.reset(seed=seed)
torch.manual_seed(seed)

SavedAction = namedtuple('SavedAction', ['log_prob', 'value'])
```

[3]:
```
env = gym.make('CartPole-v1')
env.reset(seed=seed)
torch.manual_seed(seed)


SavedAction = namedtuple('SavedAction', ['log_prob', 'value'])
```

```
class Policy(nn.Module):
    """
    implements both actor and critic in one model
    """
    def __init__(self):
        super(Policy, self).__init__()
        self.affine1 = nn.Linear(4, 128)

        # actor's layer
        self.action_head = nn.Linear(128, 2)

        # critic's layer
        self.value_head = nn.Linear(128, 1)

        # action & reward buffer
        self.saved_actions = []
        self.rewards = []

    def forward(self, x):
        """
        forward of both actor and critic
        """
        x = F.relu(self.affine1(x))

        # actor: choses action to take from state s_t
        # by returning probability of each action
        action_prob = F.softmax(self.action_head(x), dim=-1)

        # critic: evaluates being in the state s_t
        state_values = self.value_head(x)

        # return values for both actor and critic as a tuple of 2 values:
        # 1. a list with the probability of each action over the action space
        # 2. the value from state s_t
        return action_prob, state_values
```

[4]:
```
model = Policy()
optimizer = optim.Adam(model.parameters(), lr=3e-2)
eps = np.finfo(np.float32).eps.item()
```

[5]:
```
def select_action(state):
    state = torch.from_numpy(state).float()
    probs, state_value = model(state)

    # create a categorical distribution over the list of probabilities of actions
    m = Categorical(probs)
```

```
    # and sample an action using the distribution
    action = m.sample()

    # save to action buffer
    model.saved_actions.append(SavedAction(m.log_prob(action), state_value))

    # the action to take (left or right)
    return action.item()


def finish_episode():
    """
    Training code. Calculates actor and critic loss and performs backprop.
    """
    R = 0
    saved_actions = model.saved_actions
    policy_losses = [] # list to save actor (policy) loss
    value_losses = [] # list to save critic (value) loss
    returns = [] # list to save the true values

    # calculate the true value using rewards returned from the environment
    for r in model.rewards[::-1]:
        # calculate the discounted value
        R = r + gamma * R
        returns.insert(0, R)

    returns = torch.tensor(returns)
    returns = (returns - returns.mean()) / (returns.std() + eps)

    for (log_prob, value), R in zip(saved_actions, returns):
        advantage = R - value.item()

        # calculate actor (policy) loss
        policy_losses.append(-log_prob * advantage)

        # calculate critic (value) loss using L1 smooth loss
        value_losses.append(F.smooth_l1_loss(value, torch.tensor([R])))

    # reset gradients
    optimizer.zero_grad()

    # sum up all the values of policy_losses and value_losses
    loss = torch.stack(policy_losses).sum() + torch.stack(value_losses).sum()

    # perform backprop
    loss.backward()
    optimizer.step()
```

```
    # reset rewards and action buffer
    del model.rewards[:]
    del model.saved_actions[:]


def train():
    rewards = []
    running_reward = 10

    # run infinitely many episodes
    for i_episode in range(2000):

        # reset environment and episode reward
        state, _ = env.reset()
        ep_reward = 0

        # for each episode, only run 9999 steps so that we don't
        # infinite loop while learning
        for t in range(1, 10000):

            # select action from policy
            action = select_action(state)

            # take the action
            state, reward, done, _, _ = env.step(action)

            model.rewards.append(reward)
            ep_reward += reward
            if done:
                break

        # update cumulative reward
        running_reward = 0.05 * ep_reward + (1 - 0.05) * running_reward
        rewards.append(running_reward)

        # perform backprop
        finish_episode()

        # log results
        if i_episode % log_interval == 0:
            print('Episode {}\tLast reward: {:.2f}\tAverage reward: {:.2f}'.
↪format(
                  i_episode, ep_reward, running_reward))

                # check if we have "solved" the cart pole problem
        if running_reward > env.spec.reward_threshold:
```

```
            print("Solved! Running reward is now {} and "
                  "the last episode runs to {} time steps!".
↪format(running_reward, t))
            break
    return rewards
```

[6]:
```
def plot(shared_rewards, unshared_rewards, threshold):
    """
    Plot the results of shared and unshared architectures with a horizontal␣
↪threshold line.

    Args:
        shared_rewards (list): List of average rewards for shared architecture.
        unshared_rewards (list): List of average rewards for unshared␣
↪architecture.
        threshold (float): Threshold value for the horizontal line.
    """
    # Plotting
    plt.figure(figsize=(10, 6))
    plt.plot(shared_rewards, label='Shared Architecture')
    plt.plot(unshared_rewards, label='Unshared Architecture')
    plt.axhline(y=threshold, color='r', linestyle='--', label='Threshold')
    plt.title('Comparison of Shared vs. Unshared Architectures')
    plt.xlabel('Episode')
    plt.ylabel('Average Reward')
    plt.legend()
    plt.grid(True)
    plt.show()
```

[7]:
```
shared_rewards = train()
```

```
/opt/miniconda3/envs/torch/lib/python3.8/site-
packages/gym/utils/passive_env_checker.py:233: DeprecationWarning: `np.bool8` is
a deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):

Episode 0       Last reward: 35.00      Average reward: 11.25
Episode 10      Last reward: 11.00      Average reward: 10.84
Episode 20      Last reward: 10.00      Average reward: 10.33
Episode 30      Last reward: 10.00      Average reward: 10.06
Episode 40      Last reward: 10.00      Average reward: 9.79
Episode 50      Last reward: 10.00      Average reward: 9.59
Episode 60      Last reward: 8.00       Average reward: 9.56
Episode 70      Last reward: 9.00       Average reward: 9.44
Episode 80      Last reward: 11.00      Average reward: 9.65
Episode 90      Last reward: 11.00      Average reward: 9.49
Episode 100     Last reward: 10.00      Average reward: 9.58
Episode 110     Last reward: 9.00       Average reward: 9.51
```

```
Episode 120     Last reward: 9.00       Average reward: 9.50
Episode 130     Last reward: 11.00      Average reward: 9.45
Episode 140     Last reward: 9.00       Average reward: 10.21
Episode 150     Last reward: 18.00      Average reward: 23.84
Episode 160     Last reward: 62.00      Average reward: 45.89
Episode 170     Last reward: 35.00      Average reward: 39.74
Episode 180     Last reward: 72.00      Average reward: 57.18
Episode 190     Last reward: 42.00      Average reward: 60.63
Episode 200     Last reward: 270.00     Average reward: 68.62
Episode 210     Last reward: 19.00      Average reward: 86.29
Episode 220     Last reward: 20.00      Average reward: 62.44
Episode 230     Last reward: 28.00      Average reward: 50.23
Episode 240     Last reward: 70.00      Average reward: 61.97
Episode 250     Last reward: 73.00      Average reward: 66.60
Episode 260     Last reward: 123.00     Average reward: 77.98
Episode 270     Last reward: 74.00      Average reward: 107.29
Episode 280     Last reward: 101.00     Average reward: 139.53
Episode 290     Last reward: 133.00     Average reward: 140.83
Episode 300     Last reward: 257.00     Average reward: 145.04
Episode 310     Last reward: 196.00     Average reward: 181.36
Episode 320     Last reward: 284.00     Average reward: 209.24
Episode 330     Last reward: 129.00     Average reward: 328.66
Episode 340     Last reward: 239.00     Average reward: 264.53
Episode 350     Last reward: 180.00     Average reward: 257.65
Episode 360     Last reward: 219.00     Average reward: 234.69
Solved! Running reward is now 921.6914894386209 and the last episode runs to
9999 time steps!
```

2 TODO: Write a policy class similar to the above, without using shared features for the actor and critic and compare their performance.

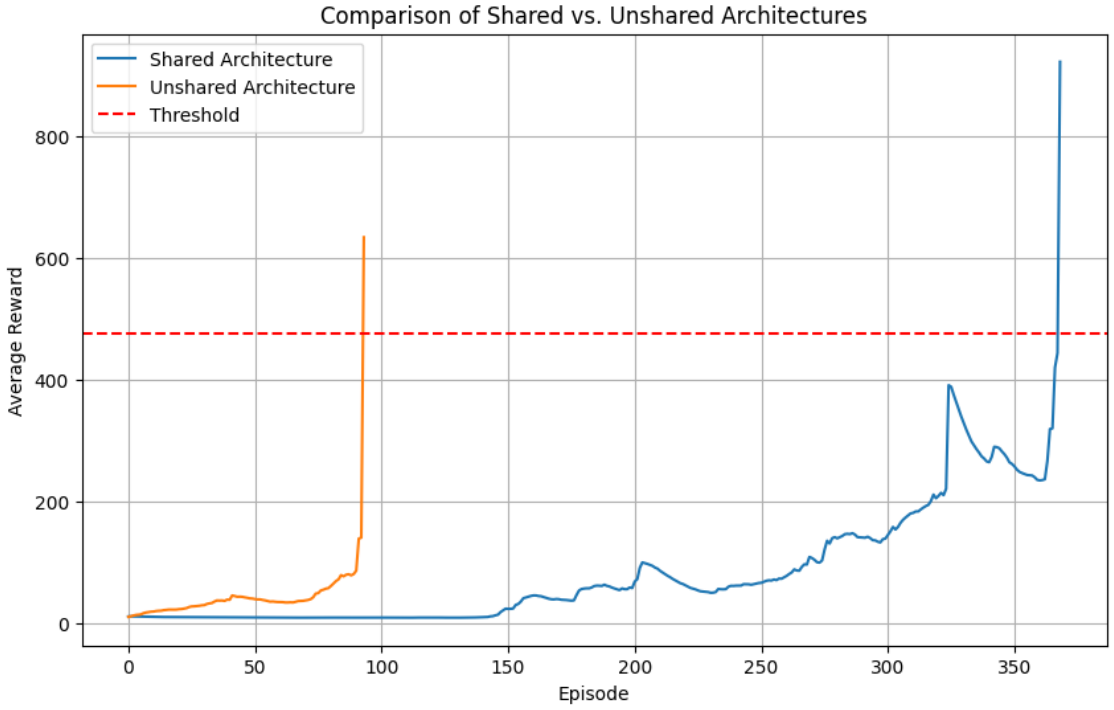
[8]:
```
#TODO: Write a policy class similar to the above, without using shared features␣
 ↪for the actor and critic and compare their
# performance.

class UnsharedPolicy(nn.Module):
        def __init__(self):
                super(UnsharedPolicy, self).__init__()
                #TODO: Fill in.
                # Actor network layers
                self.actor_fc1 = nn.Linear(4, 128)
                self.actor_fc2 = nn.Linear(128, 2)  # 2 actions in CartPole-v1

                # Critic network layers
                self.critic_fc1 = nn.Linear(4, 128)
                self.critic_fc2 = nn.Linear(128, 1)

                # Action & reward buffer
                self.saved_actions = []
                self.rewards = []

        def forward(self, x):
                # TODO: Fill in. For your networks, use the same hidden_size for␣
 ↪the layers as the previous policy, that is 128.
                # Actor network
                actor_x = F.relu(self.actor_fc1(x))
                action_probs = F.softmax(self.actor_fc2(actor_x), dim=-1)

                # Critic network
                critic_x = F.relu(self.critic_fc1(x))
                state_value = self.critic_fc2(critic_x)

                # return values for both actor and critic as a tuple of 2 values:
                # 1. a list with the probability of each action over the action␣
 ↪space
                # 2. the value from state s_t
                return action_probs, state_value
```

[9]:
```
model = UnsharedPolicy()
optimizer = optim.Adam(model.parameters(), lr=3e-2)
eps = np.finfo(np.float32).eps.item()
unshared_rewards = train()
```

```
Episode 0       Last reward: 24.00      Average reward: 10.70
```

```
Episode 10      Last reward: 24.00      Average reward: 19.47
Episode 20      Last reward: 35.00      Average reward: 23.33
Episode 30      Last reward: 36.00      Average reward: 29.98
Episode 40      Last reward: 33.00      Average reward: 38.66
Episode 50      Last reward: 24.00      Average reward: 39.87
Episode 60      Last reward: 34.00      Average reward: 35.00
Episode 70      Last reward: 51.00      Average reward: 37.74
Episode 80      Last reward: 130.00     Average reward: 61.41
Episode 90      Last reward: 192.00     Average reward: 86.55
Solved! Running reward is now 633.6832365471276 and the last episode runs to
9999 time steps!
```

[10]:
```
# Plot results
plot(shared_rewards, unshared_rewards, env.spec.reward_threshold)
```



Comparision between the performance of shared and unshared networks in Actor-Critic Method

Implementation: Each of the policy classes concerning a shared and an unshared network in the Actor-Critic Method is implemented for CartPole-v1. A brief overview regarding the network has been provided followingly.

1. *Shared Network:* For the shared architecture, a fully connected network of size 4 (Input Layer) $\times$128 (Hidden Layer) $\times$ 3 (Output Layer) is used, where 2 of the 3 neurons in the output layer are represented for the actor and the remaining one is used for the critic.

2. *Unshared Network:* Two analogous fully connected networks, one of size 4 (Input Layer) $\times$128 (Hidden Layer) $\times$ 2 (Output Layer) and the other of size 4 (Input Layer) $\times$128 (Hidden Layer) $\times$ 1 (Output Layer) is used to represent the actor and the critic respectively.

Observations and Inferences: Some observations are made concerning the learning in both paradigms and based on them the following inferences are drawn.

1. *Convergence Speed:* The unshared network appears to converge faster, as indicated by the shorter time to achieve a "solved" state. The shared network achieves a running reward of 921.69 after 360 episodes, while the unshared network achieves a running reward of 633.68 after only 90 episodes. This suggests that the unshared network learns more efficiently and reaches a satisfactory performance level in fewer episodes.

2. *Average Reward Trajectories:* Analyzing the average reward trajectories over episodes, it's observed that the unshared network generally maintains a smoother progression compared to the shared network. The average reward in the unshared network steadily increases over episodes, reflecting consistent improvement in performance. In contrast, the average reward in the shared network exhibits more fluctuations, with periods of stagnation followed by sudden spikes in performance. This indicates that the learning process in the unshared network is more stable and reliable.

3. *Final Performance:* Despite the faster convergence of the unshared network, both networks achieve high levels of performance by the end of training. The shared network reaches a running reward of 921.69, while the unshared network reaches a slightly lower running reward of 633.68. Although the unshared network converges faster, the shared network ultimately achieves a higher level of performance. This suggests that the shared architecture may have certain advantages in terms of final performance, even though it may take longer to converge.

4. *Sensitivity to Architecture:* The results highlight the sensitivity of the reinforcement learning process to the choice of architecture. Despite using similar network sizes and training procedures, the shared and unshared architectures exhibit distinct learning dynamics and performance trajectories. This underscores the importance of carefully selecting and designing the neural network architecture for reinforcement learning tasks to achieve optimal performance.

Overall, while the unshared network demonstrates faster convergence and smoother learning dynamics, the shared network ultimately achieves higher levels of performance in this specific case. These findings emphasize the trade-offs and considerations involved in choosing between shared and unshared architectures for actor-critic reinforcement learning tasks. Followingly we mention some general overview of the architectures that might be the reason for this behaviour.

1. *Parameter Separation:* In unshared networks, the parameters of the actor and critic are completely separate, allowing each network to specialize in its respective task without interference

from the other. This separation may lead to more efficient learning as each network can focus exclusively on optimizing its objective.

2. *Reduced Interference:* Shared networks require coordination between the actor and critic during training, as updates to one network may affect the other. This interdependence can introduce additional complexities and potential conflicts during training, leading to slower convergence.

3. *Gradient Interference:* In shared networks, updates to one set of parameters can inadvertently impact the gradients used to update the other set of parameters. This phenomenon, known as gradient interference or catastrophic forgetting, can hinder convergence by introducing noise or conflicting signals into the training process.

4. *Exploration Strategies:* Unshared networks may enable more effective exploration strategies by allowing the actor to explore independently of the critic's evaluations. This autonomy in exploration can lead to better discovery of optimal policies and higher rewards.

5. *Capacity Allocation:* Note that we allocate 2 neurons for Actor while 1 neuron for the critic. Shared networks need to allocate capacity to represent both the policy and value function within the same architecture. This shared capacity allocation may result in a suboptimal representation of either the actor or critic, hindering learning efficiency and performance.

In summary, the differences in architecture, particularly regarding parameter sharing, capacity allocation, and gradient interference, likely contribute to the observed behaviour of faster convergence in the unshared network compared to the shared network. These architectural choices influence the learning dynamics and efficiency of the actor-critic method, ultimately affecting the rate of convergence and the performance of the learning agent.

[ ]: