Tutorial: Actor Critic Implementation

```
#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

#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'])
```

```
/usr/local/lib/python3.10/dist-packages/gym/core.py:317:
DeprecationWarning: WARN: Initializing wrapper in old step API which
returns one bool instead of two. It is recommended to set
`new_step_api=True` to use new step API. This will be the default
behaviour in future.
  deprecation(
/usr/local/lib/python3.10/dist-packages/gym/wrappers/step_api_compatib
ility.py:39: DeprecationWarning: WARN: Initializing environment in old
step API which returns one bool instead of two. It is recommended to
set `new_step_api=True` to use new step API. This will be the default
behaviour in future.
  deprecation(
```

```
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

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

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():
    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

        # 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
```

```
train()
```

```
Episode 0  Last reward: 22.00    Average reward: 10.60
Episode 10 Last reward: 28.00    Average reward: 16.78
Episode 20 Last reward: 42.00    Average reward: 33.66
Episode 30 Last reward: 21.00    Average reward: 31.73
Episode 40 Last reward: 26.00    Average reward: 29.00
Episode 50 Last reward: 150.00   Average reward: 64.74
Episode 60 Last reward: 85.00    Average reward: 87.97
Episode 70 Last reward: 234.00   Average reward: 153.18
Episode 80 Last reward: 44.00    Average reward: 144.46
Episode 90 Last reward: 44.00    Average reward: 101.80
Episode 100      Last reward: 133.00   Average reward: 89.91
Episode 110      Last reward: 60.00    Average reward: 88.15
Episode 120      Last reward: 339.00   Average reward: 178.64
Episode 130      Last reward: 105.00   Average reward: 177.38
Episode 140      Last reward: 34.00    Average reward: 138.97
Episode 150      Last reward: 500.00   Average reward: 155.96
Episode 160      Last reward: 500.00   Average reward: 267.45
Episode 170      Last reward: 500.00   Average reward: 360.77
Episode 180      Last reward: 140.00   Average reward: 312.40
Episode 190      Last reward: 119.00   Average reward: 242.36
Episode 200      Last reward: 154.00   Average reward: 202.31
Episode 210      Last reward: 240.00   Average reward: 192.62
Episode 220      Last reward: 500.00   Average reward: 307.03
Episode 230      Last reward: 500.00   Average reward: 357.78
Episode 240      Last reward: 500.00   Average reward: 390.83
Episode 250      Last reward: 500.00   Average reward: 421.68
Episode 260      Last reward: 165.00   Average reward: 428.27
Episode 270      Last reward: 500.00   Average reward: 438.85
Episode 280      Last reward: 500.00   Average reward: 452.60
Episode 290      Last reward: 500.00   Average reward: 471.62
Solved! Running reward is now 475.6650523591305 and the last episode
runs to 500 time steps!
```

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__()
        # Define separate layers for actor and critic since they do
not share features
        self.actor_affine1 = nn.Linear(4, 128) # Actor layer
        self.critic_affine1 = nn.Linear(4, 128) # Critic layer
        # Actor's output layer
        self.action_head = nn.Linear(128, 2)

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

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

    def forward(self, x):
        # Separate pathways for actor and critic
        actor_x = F.relu(self.actor_affine1(x))
        critic_x = F.relu(self.critic_affine1(x))

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

        # Critic: evaluates being in the state s_t
        state_values = self.value_head(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_prob, state_values

model = UnsharedPolicy()
# Learning parameter changes from 3e-2 to 3e-3
optimizer = optim.Adam(model.parameters(), lr=3e-3)
eps = np.finfo(np.float32).eps.item()

train()

Episode 0  Last reward: 53.00    Average reward: 12.15
Episode 10 Last reward: 11.00    Average reward: 15.03
Episode 20 Last reward: 62.00    Average reward: 19.29
Episode 30 Last reward: 11.00    Average reward: 20.63
Episode 40 Last reward: 33.00    Average reward: 21.77
Episode 50 Last reward: 16.00    Average reward: 26.82
Episode 60 Last reward: 137.00   Average reward: 39.76
Episode 70 Last reward: 34.00    Average reward: 48.42
Episode 80 Last reward: 17.00    Average reward: 47.86
Episode 90 Last reward: 80.00    Average reward: 66.07
Episode 100     Last reward: 121.00   Average reward: 88.49
Episode 110     Last reward: 118.00   Average reward: 110.36
Episode 120     Last reward: 199.00   Average reward: 139.14
Episode 130     Last reward: 172.00   Average reward: 173.34
Episode 140     Last reward: 173.00   Average reward: 161.62
Episode 150     Last reward: 191.00   Average reward: 181.37
Episode 160     Last reward: 500.00   Average reward: 271.40
Episode 170     Last reward: 285.00   Average reward: 271.58
Episode 180     Last reward: 249.00   Average reward: 272.62
Episode 190     Last reward: 471.00   Average reward: 314.10
Episode 200     Last reward: 500.00   Average reward: 365.90
Episode 210     Last reward: 500.00   Average reward: 406.24
Episode 220     Last reward: 287.00   Average reward: 406.90
Episode 230     Last reward: 296.00   Average reward: 402.68
Episode 240     Last reward: 435.00   Average reward: 348.99
Episode 250     Last reward: 264.00   Average reward: 310.02
Episode 260     Last reward: 500.00   Average reward: 328.94
Episode 270     Last reward: 460.00   Average reward: 395.58
Episode 280     Last reward: 122.00   Average reward: 307.10
Episode 290     Last reward: 116.00   Average reward: 229.85
Episode 300     Last reward: 125.00   Average reward: 180.34
Episode 310     Last reward: 123.00   Average reward: 150.25
Episode 320     Last reward: 129.00   Average reward: 138.72
Episode 330     Last reward: 241.00   Average reward: 148.81
Episode 340     Last reward: 259.00   Average reward: 174.95
Episode 350     Last reward: 346.00   Average reward: 227.90
Episode 360     Last reward: 500.00   Average reward: 308.41
Episode 370     Last reward: 500.00   Average reward: 376.00
Episode 380     Last reward: 500.00   Average reward: 402.86
Episode 390     Last reward: 500.00   Average reward: 419.73
Episode 400     Last reward: 500.00   Average reward: 448.29
Episode 410     Last reward: 500.00   Average reward: 458.46
Episode 420     Last reward: 331.00   Average reward: 459.46
Episode 430     Last reward: 500.00   Average reward: 472.01
Solved! Running reward is now 476.00371339359236 and the last episode
runs to 500 time steps!
```

Comparison

Shared Policy:

- Episode 290: Last reward of 500.00, Average reward of 471.62 Solved! Running reward is now 475.67, and the last episode runs to 500 time steps.

Unshared Policy:

- Episode 430: Last reward of 500.00, Average reward of 472.01 Solved! Running reward is now 476.00, and the last episode runs to 500 time steps.

Both policies achieved similar performance in terms of solving the environment, with the unshared policy slightly outperforming the shared policy by a small margin in terms of the running reward. However, the difference in performance is relatively minor, indicating that both approaches are effective for solving the CartPole environment within the specified constraints.