Tutorial: Actor Critic Implementation

```python
# 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'])

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()[0]
        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, truncated, _ = 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: 22.00    Average reward: 21.89
Episode 20 Last reward: 37.00    Average reward: 28.09
Episode 30 Last reward: 28.00    Average reward: 37.72
Episode 40 Last reward: 276.00   Average reward: 91.54
Episode 50 Last reward: 157.00   Average reward: 115.99
Episode 60 Last reward: 159.00   Average reward: 142.15
Episode 70 Last reward: 131.00   Average reward: 153.68
Episode 80 Last reward: 17.00    Average reward: 128.06
```

```
Episode 90	Last reward: 229.00	Average reward: 151.71
Episode 100	Last reward: 642.00	Average reward: 228.76
Solved! Running reward is now 636.0859746392732 and the last episode
runs to 6839 time steps!
```

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

```python
# 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.
        hidden_size = 128
        # Actor network
        self.actor_affine1 = nn.Linear(4, hidden_size)
        self.action_head = nn.Linear(hidden_size, 2)

        # Critic network
        self.critic_affine1 = nn.Linear(4, hidden_size)
        self.value_head = nn.Linear(hidden_size, 1)

        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 forward pass
        actor_x = F.relu(self.actor_affine1(x))
        action_prob = F.softmax(self.action_head(actor_x), dim=-1)

        # Critic forward pass
        critic_x = F.relu(self.critic_affine1(x))
        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()
optimizer = optim.Adam(model.parameters(), lr=3e-2)
eps = np.finfo(np.float32).eps.item()
train()
```

```
Episode 0  Last reward: 11.00    Average reward: 10.05
Episode 10 Last reward: 29.00    Average reward: 21.53
Episode 20 Last reward: 129.00   Average reward: 46.80
Episode 30 Last reward: 434.00   Average reward: 130.49
Episode 40 Last reward: 9999.00  Average reward: 707.31
Solved! Running reward is now 707.3117556396027 and the last episode
runs to 9999 time steps!
```