````
# Reinforcement learning training example

```bash
pip install -r requirements.txt
# For REINFORCE:
python reinforce.py
# For actor critic:
python actor_critic.py
```
torch
numpy
gym
import argparse
import gym
import numpy as np
from itertools import count

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


parser =
argparse.ArgumentParser(description='PyTorch
REINFORCE example')
parser.add_argument('--gamma', type=float,
default=0.99, metavar='G',
                    help='discount factor (default:
0.99)')
parser.add_argument('--seed', type=int,
default=543, metavar='N',
                    help='random seed (default:
543)')
parser.add_argument('--render', action='store_true',
                    help='render the environment')
````

```
parser.add_argument('--log-interval', type=int, 
default=10, metavar='N',
                    help='interval between training 
status logs (default: 10)')
args = parser.parse_args()


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


class Policy(nn.Module):
    def __init__(self):
        super(Policy, self).__init__()
        self.affine1 = nn.Linear(4, 128)
        self.dropout = nn.Dropout(p=0.6)
        self.affine2 = nn.Linear(128, 2)

        self.saved_log_probs = []
        self.rewards = []

    def forward(self, x):
        x = self.affine1(x)
        x = self.dropout(x)
        x = F.relu(x)
        action_scores = self.affine2(x)
        return F.softmax(action_scores, dim=1)


policy = Policy()
optimizer = optim.Adam(policy.parameters(), lr=1e-2)
eps = np.finfo(np.float32).eps.item()


def select_action(state):
    state = 
```

```
torch.from_numpy(state).float().unsqueeze(0)
    probs = policy(state)
    m = Categorical(probs)
    action = m.sample()

policy.saved_log_probs.append(m.log_prob(action))
    return action.item()


def finish_episode():
    R = 0
    policy_loss = []
    returns = []
    for r in policy.rewards[::-1]:
        R = r + args.gamma * R
        returns.insert(0, R)
    returns = torch.tensor(returns)
    returns = (returns - returns.mean()) /
(returns.std() + eps)
    for log_prob, R in zip(policy.saved_log_probs,
returns):
        policy_loss.append(-log_prob * R)
    optimizer.zero_grad()
    policy_loss = torch.cat(policy_loss).sum()
    policy_loss.backward()
    optimizer.step()
    del policy.rewards[:]
    del policy.saved_log_probs[:]


def main():
    running_reward = 10
    for i_episode in count(1):
        state, ep_reward = env.reset(), 0
        for t in range(1, 10000):  # Don't infinite
loop while learning
            action = select_action(state)
```

```
            state, reward, done, _ =
env.step(action)
            if args.render:
                env.render()
            policy.rewards.append(reward)
            ep_reward += reward
            if done:
                break

        running_reward = 0.05 * ep_reward + (1 -
0.05) * running_reward
        finish_episode()
        if i_episode % args.log_interval == 0:
            print('Episode {}\tLast reward: {:.2f}
\tAverage reward: {:.2f}'.format(
                  i_episode, ep_reward,
running_reward))
        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


if __name__ == '__main__':
    main()
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

# Cart Pole

parser =
argparse.ArgumentParser(description='PyTorch actor-
critic example')
parser.add_argument('--gamma', type=float,
default=0.99, metavar='G',
                    help='discount factor (default:
0.99)')
parser.add_argument('--seed', type=int,
default=543, metavar='N',
                    help='random seed (default:
543)')
parser.add_argument('--render', action='store_true',
                    help='render the environment')
parser.add_argument('--log-interval', type=int,
default=10, metavar='N',
                    help='interval between training
status logs (default: 10)')
args = parser.parse_args()


env = gym.make('CartPole-v0')
env.seed(args.seed)
torch.manual_seed(args.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 tupel 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. Calcultes 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 + args.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 main():
    running_reward = 10

    # run inifinitely many episodes
    for i_episode in count(1):

        # 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)

            if args.render:
                env.render()

            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 % args.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


if __name__ == '__main__':
    main()
```