Learn to Play CartPole with PyTorch

Code: *agent.py*, *cartpole.py*

This program implements REINFORCE, a fundamental algorithm in reinforcement learning. REINFORCE is a policy-based algorithm, meaning that it's model-free, direct parametrization of agent's (stochastic) policy and make no use of value functions. The algorithm is carried out in an episodic way. We compute the discounted sum of rewards(returns) of each action at the end of the episdoe and backpropagate the return-guided log probability of the action taken(score function). By nudging parameters of the policy toward the direction more probable each time, the agent gradually learns how to take action at each step for a given scenario(state).

The code presented here is taken from 莫煩's site. It's a great site where you can learn a lot from.

Another purpose of this post to demonstrate how easy it can be to implement RL algos in PyTorch than it were to be done in TensorFlow. I modified the code quite a lot but maintained the original structure so readers can make clear comparison. One of the major incovenience to build NN with PyTorch is the lack of visualization tools as TensorBoard serves for TensorFlow. I've seen comment on the PyTorch forum that there will be a TensorBoard integration with PyTorch in the future. But for now what you do is print out the model in the console to see if you built your model right.

The code is arranged in two files. *agent.py* contains the definition of an agent and a RL interface so that agent can receive state information, take action at each step and learn from the roll-out at the end of the episode. *cartpole.py* makes use of the OpenAI gym library to create an easy-to-use environment to implement a typical RL environment-agent training paradigm.

Here's the code. They are available on my Github in the folder named after the article title.

agent.py

```
import numpy as np
from sklearn import preprocessing

import torch
import torch.nn as nn
from torch.autograd import Variable


class PolicyGradient:
    def __init__(self, n_actions, n_features, learning_rate=0.01,
reward_decay=0.95):
        self.n_actions = n_actions
        self.n_features = n_features
        self.lr = learning_rate
        self.gamma = reward_decay

        # Record action/reward at each step
```

```
        self.ep_as, self.ep_rs = [], []

        # Define model, use high level wrappers
        # More complicated model can be build from deriving from
torch.nn.Module
        self.model = nn.Sequential(
            nn.Linear(self.n_features, 10),
            nn.Tanh(),
            nn.Linear(10, self.n_actions),
            nn.Softmax()
        )

        # # Print model
        # print(self.model)
        # for param in self.model.parameters(): print(param)

        self.optimizer = torch.optim.Adam(self.model.parameters(),
lr=self.lr)

    def choose_action(self, obs):
        # Cast np array to torch variable, very important to use type
np.float32
        s = Variable(torch.from_numpy(obs.astype(np.float32))).unsqueeze(0)
        prob_weights = self.model.forward(s)

        prob_weights_numpy = prob_weights.data.numpy()
        action = np.random.choice(range(prob_weights_numpy.shape[1]),
p=prob_weights_numpy[0])

        # One-hot encoding
        _one_hot = np.zeros((1, 2))
        _one_hot[0][action] = 1
        one_hot = Variable(torch.from_numpy(_one_hot.astype(np.float32)))

        # Append log prob of the action taken
        self.ep_as.append(torch.log(torch.sum(prob_weights * one_hot)))

        return action

    def store_reward(self, r):
        self.ep_rs.append(r)

    def learn(self):
        # Compute return
        vt = self._compute_vt()

        # Learn
        loss = Variable(torch.zeros(1, 1))
```

```
        for i in range(len(self.ep_as)): loss += float(vt[i]) *
self.ep_as[i]

        self.optimizer.zero_grad()
        (-loss).backward(retain_variables=True)  # Min -reward = Max reward
        self.optimizer.step()

        # Empty action/reward list after an episode ends
        self.ep_as, self.ep_rs = [], []

    def _compute_vt(self):
        vt = np.zeros_like(self.ep_rs)

        running_sum = 0
        for t in reversed(range(0, len(self.ep_rs))):
            running_sum = running_sum * self.gamma + self.ep_rs[t]  # sum of
discounted rewards
            vt[t] = running_sum

        '''
        Meaning of z score scaling:
            1. Scale reward magnitude
            2. Encourage actions from the former half -> actions preventing
the pole from falling
               Discourage actions from the latter half -> actions causing
the pole to fall off
        '''
        vt = preprocessing.scale(vt)

        return vt
```

cartpole.py

```
import gym
from agent import PolicyGradient

RENDER = False
DISPLAY_REWARD_THRESHOLD = 400  # Render environment if total episode reward
exceeds this threshold

env = gym.make('CartPole-v0')
env.seed(1)     # reproducible, vanilla PG has high variance
env = env.unwrapped

# # Basic info about the envir
# print(env.action_space)
# print(env.observation_space)
# print(env.observation_space.high)
```

```python
# print(env.observation_space.low)

RL = PolicyGradient(
    n_actions=env.action_space.n,
    n_features=env.observation_space.shape[0],
    learning_rate=0.02,
    reward_decay=0.99,
)


running_reward = 0
for i_episode in range(3000):
    observation = env.reset()

    while True:
        if RENDER: env.render()

        action = RL.choose_action(observation)
        observation_, reward, done, info = env.step(action)

        RL.store_reward(reward)

        ep_rs_sum = sum(RL.ep_rs)

        # End the ep if done or get enough reward
        if done or ep_rs_sum > 2 * DISPLAY_REWARD_THRESHOLD:
            running_reward = running_reward * 0.9 + ep_rs_sum * 0.1
            if running_reward > DISPLAY_REWARD_THRESHOLD: RENDER = True

            print("Episode:", i_episode, "  Reward:", int(running_reward))
            vt = RL.learn()

            break

        observation = observation_

        # Quit if agent performs stably
        if running_reward > 2 * DISPLAY_REWARD_THRESHOLD: break
```