University of Tehran
Electrical and Computer Engineering Department
Neural Networks and Deep Learning
Extra Homework

Name	Danial Saeedi
Student No	810198571
Date	Wednesday, December 15, 2021

Table of contents

1. Solving CartPole Problem using Policy Gradient Method

Part 1

Import Dependencies

First of all, we have to import dependencies:

In [ ]:
```
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch
import gym
from torch.autograd import Variable
import random
```

Define Parameters

Through trial and error, the chosen learning rate and gamma parameters are 0.01 and 0.99 respectively.

In [ ]:
```
LR = 0.01
GAMMA = 0.99
```

Creating an environment in OpenAI Gym

In [ ]:
```
env = gym.make('CartPole-v0').unwrapped
history = []
```

Setting up the policy network

The input and output size of policy network is 4 and 2 respectively because the dimension of observation space is 4 and there are 2 possible actions(either going left or right) in this problem. The chosen optimizer is Adam.

```
In [ ]: class Network(nn.Module):
            def __init__(self):
                super(Network, self).__init__()
                # define forward pass with one hidden layer with ReLU activation and sofmax after output layer
                self.l1 = nn.Linear(4, 150)
                self.l2 = nn.Linear(150, 2)
            def forward(self, x):
                x = F.relu(self.l1(x))
                x = F.softmax(self.l2(x))
                return x
```

```
In [ ]: model = Network()

        use_cuda = torch.cuda.is_available()
        if use_cuda:
            model.cuda()
        FloatTensor = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
        LongTensor = torch.cuda.LongTensor if use_cuda else torch.LongTensor

        optim = torch.optim.Adam(model.parameters(), lr=LR)
```

Calculate Discount Rewards

Here, we calculate discount reward based on this formula:

$$R_t = \sum_{k=t}^{T} \gamma^{(k-t)} r_k(s_k, a_k)$$

```
In [ ]: def calculate_discount_rewards(r):
            discounted_r = torch.zeros(r.size())
            running_add = 0
            for t in reversed(range(len(r))):
                running_add = running_add * GAMMA + r[t]
                discounted_r[t] = running_add

            return discounted_r
```

Training

The summary of section:

- Calculate the probability of the action taken at each time step.
- Multiply the probability by the discounted return (the sum of rewards).
- Use this probability-weighted return to backpropagate and minimize the loss.

In [ ]:
```
for e in range(10000):
    complete = run_episode(model, e, env)

    if complete:
        break
```

```
[Episode     0] reward: 12.0
[Episode     1] reward: 14.0
[Episode     2] reward: 31.0
[Episode     3] reward: 42.0
[Episode     4] reward: 33.0
[Episode     5] reward: 25.0
[Episode     6] reward: 27.0
[Episode     7] reward: 25.0
[Episode     8] reward: 33.0
[Episode     9] reward: 36.0
[Episode    10] reward: 75.0
[Episode    11] reward: 50.0
[Episode    12] reward: 39.0
[Episode    13] reward: 73.0
[Episode    14] reward: 31.0
[Episode    15] reward: 28.0
[Episode    16] reward: 53.0
[Episode    17] reward: 83.0
[Episode    18] reward: 100.0
[Episode    19] reward: 41.0
[Episode    20] reward: 103.0
[Episode    21] reward: 169.0
[Episode    22] reward: 159.0
```

Run Episode

In [ ]:
```
def run_episode(net, e, env):
    state = env.reset()
    reward_sum = 0
    xs = FloatTensor([])
    ys = FloatTensor([])
    rewards = FloatTensor([])
    steps = 0

    while True:
        x = FloatTensor([state])
        xs = torch.cat([xs, x])

        action_prob = net(Variable(x))

        # select an action depends on probability
        action = 0 if random.random() < action_prob.data[0][0] else 1

        y = FloatTensor([[1, 0]] if action == 0 else [[0, 1]])
        ys = torch.cat([ys, y])

        state, reward, done, _ = env.step(action)
        rewards = torch.cat([rewards, FloatTensor([[reward]])])
        reward_sum += reward
        steps += 1

        if done or steps >= 500:
            adv = calculate_discount_rewards(rewards)
            adv = (adv - adv.mean())/(adv.std() + 1e-7)
            loss = learn(xs, ys, adv)
            history.append(reward_sum)
            print("[Episode {:>5}] reward: {}".format(e, reward_sum))
            if sum(history[-5:])/5 > 490:
                return True
            else:
                return False
```

Learning Function

```
In [ ]: def learn(x, y, adv):
            # calculate probabilities of taking each action
            action_pred = model(Variable(x))
            y = Variable(y, requires_grad=True)
            adv = Variable(adv).cuda()
            log_lik = -y * torch.log(action_pred)
            log_lik_adv = log_lik * adv
            loss = torch.sum(log_lik_adv, 1).mean()

            optim.zero_grad()
            loss.backward()
            optim.step()

            return loss.data
```

Plotting average reward per episode

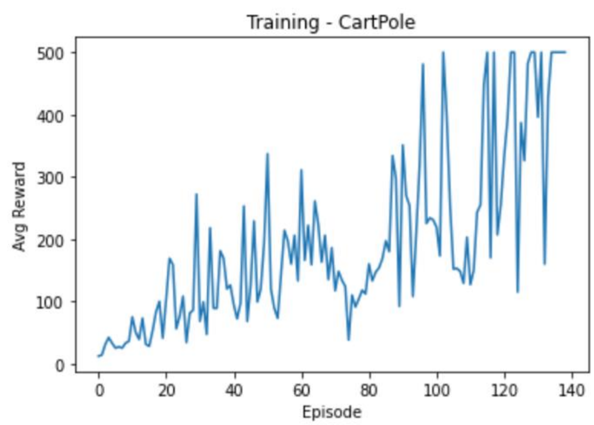

Figure 1 Average reward per episode

Part 2

These type of problem in RL require techniques such as mathematical programming and heuristic reduction to keep them manageable. A linear program with tons of actions might be solvable within seconds. Lots of real-life problems are convex or even (approximately) linear. This is a very powerful property that often makes problem solving considerably easier.

But linear programming can't solve every problem. The solution proposed here is based on this paper. The paper proposed a new policy architecture called Wolpertinger. This architecture avoids the heavy cost of evaluating all actions while **retaining generalization over actions**. The solution is based on actor-critic framework. We use multi-layer neural networks as function approximators for both our actor and critic functions. Training this policy is based on Deep Deterministic Policy Gradient.

Algorithm 1 Wolpertinger Policy

State s previously received from environment.
$\hat{\mathbf{a}} = f_{\theta^\pi}(\mathbf{s})$ {Receive proto-action from actor.}
$\mathcal{A}_k = g_k(\hat{\mathbf{a}})$ {Retrieve k approximately closest actions.}
$\mathbf{a} = \arg\max_{\mathbf{a}_j \in \mathcal{A}_k} Q_{\theta^Q}(\mathbf{s}, \mathbf{a}_j)$
Apply $\mathbf{a}$ to environment; receive $r, \mathbf{s}'$.

Figure 2 Wolpertinger Policy Algorithm

1) Action Generation

This architecture reasons over actions within a **continuous space** $\mathbb{R}^n$, and then maps this output to the discrete action Discrete Action Spaces set $\mathcal{A}$:

$$f_{\theta^\pi} : \mathcal{S} \to \mathbb{R}^n$$
$$f_{\theta^\pi}(\mathbf{s}) = \hat{\mathbf{a}}.$$

fθ is a function parametrized by θ^π, mapping from the state representation space $\mathbb{R}^m$ to the action representation space $\mathbb{R}^n$. We need to be able to map from a to an element in $\mathcal{A}$

$$g : \mathbb{R}^n \to \mathcal{A}$$
$$g_k(\hat{\mathbf{a}}) = \overset{k}{\underset{\mathbf{a}\in\mathcal{A}}{\arg\min}}\, |\,\mathbf{a} - \hat{\mathbf{a}}|_2.$$

gk is a k-nearest-neighbor mapping from a continuous space to a discrete set. It returns the k actions in A that are closest to a by D distance. This is called proto-action. In the bottom half of Figure 2 we can see proto action.

2) Action Refinement

Certain actions may be near each other in the action embedding space, but in certain states they must be distinguished as one has a particularly low long-term value relative to its neighbors.

The model refines the choice of action by selecting the highest-scoring action according to Q-value:

$$\pi_\theta(\mathbf{s}) = \underset{a \in g_k \circ f_{\theta^\pi}(\mathbf{s})}{\arg\max} \; Q_{\theta^Q}(\mathbf{s}, \mathbf{a})$$

The size of the generated action set, k, is task specific, and allows for an explicit trade-off between policy quality and speed.

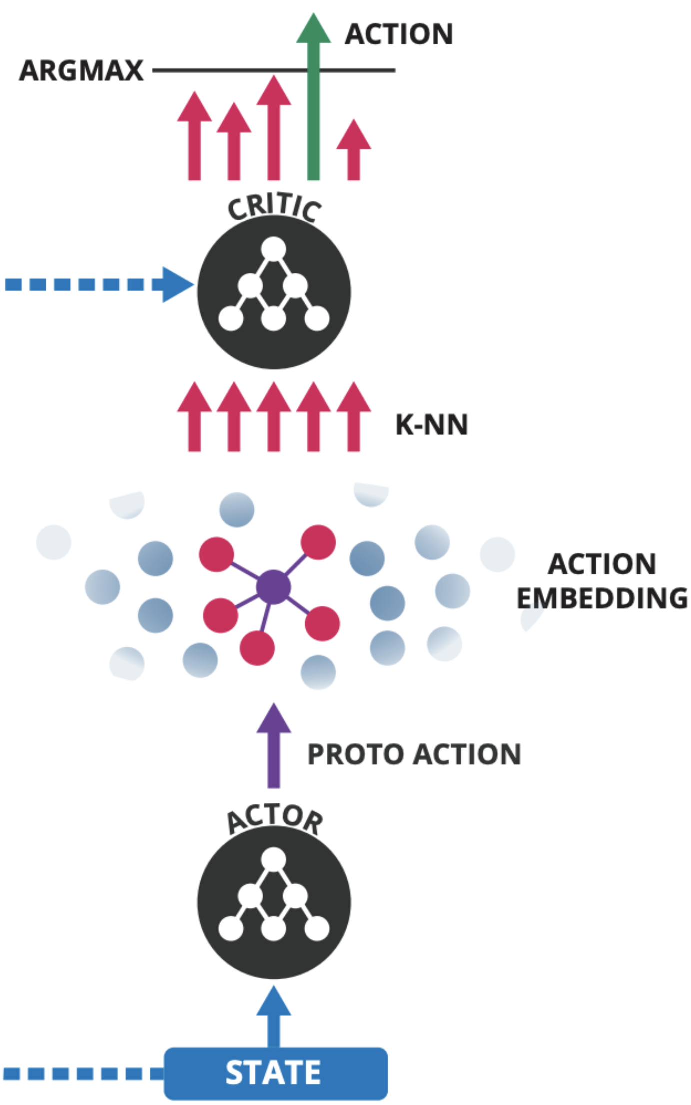

Figure 3 Wolpertinger Architecture