به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

شبکه های عصبی و یادگیری عمیق

تمرین امتیازی

نام و نام خانوادگی	شایان واصف
شماره دانشجویی	810197603
تاریخ ارسال گزارش	بهمن ماه

فهرست گزارش سوالات

سوال 1 – Policy-Gradient implementation 3

- ایجاد کتابخانه های مورد نیاز : 3
- ایجاد محیط بازی :..... 4
- تعریف کلاس سیاست ها : 4
- ایجاد تابعی برای آموزش Agent :..... 5
- مشاهده نتیجه :..... 7
- رسم نمودار و انیمیشن بدست آمده :..... 8

سوال 2 – Dealing with Large Discrete Action space 9

سوال 1 – Policy-Gradient implementation

در ابتدا مراحل الگوریتم Policy-Gradient در قالب pseudocode می‌آوریم :

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

شکل 1-1 : شبه کد مربوط به الگوریتم Policy-Gradient

در ادامه مراحل لازم برای پیاده سازی سوال را به ترتیب ذکر می‌کنیم :

ایجاد کتابخانه های مورد نیاز :

در این مرحله تمامی کتابخانه های مورد نیاز برای پیاده سازی سوال را Import می‌کنیم . از جمله مهمترین آنها ، کتابخانه gym به منظور ایجاد محیط بازی و torch برای حل بهینه ساز می‌باشد .

```
import gym
import numpy as np
from collections import deque
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (16, 10)

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Categorical
torch.manual_seed(0)

import base64, io

# For visualization
from gym.wrappers.monitoring import video_recorder
from IPython.display import HTML
from IPython import display
import glob
```

شکل 1-2 : ایجاد کتابخانه های مورد نیاز

ایجاد محیط بازی :

با کمک دستور gym. Make محیط بازی Cart Pole را ایجاد می‌کنیم و سپس از طریق آن ، مجموعه حالت ها (States) و مجموعه اعمال امکان پذیر (Actions) را می‌خوانیم:

```
env = gym.make('CartPole-v0')
env.seed(0)

print('observation space:', env.observation_space)
print('action space:', env.action_space)
```

```
observation space: Box(-3.4028234663852886e+38, 3.4028234663852886e+38, (4,), float32)
action space: Discrete(2)
```

شکل 1-3 : ایجاد محیط بازی

تعریف کلاس سیاست ها :

در این مرحله کلاس خود را برای تعیین سیاست ها را تشکیل می‌دهیم . برخلاف روش های مبتنی بر مقدار (Value-based method) ، خروجی روش های مبتنی بر سیاست ، احتمال انجام یک عمل (action) خاص می‌باشد بنابراین برای اینکه خروجی به طور احتمالاتی قابل تفسیر باشد ، از تابع فعال ساز (activation function) sigmoid استفاده می‌کنیم .

```
class Policy(nn.Module):
    def __init__(self, state_size=4, action_size=2, hidden_size=32):
        super(Policy, self).__init__()
        self.fc1 = nn.Linear(state_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, action_size)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = self.fc2(x)
        #1 dimensional probability of action
        return F.softmax(x, dim=1)

    def act(self, state):
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        probs = self.forward(state).cpu()
        model = Categorical(probs)
        action = model.sample()
        return action.item(), model.log_prob(action)
```

شکل 1-4 : تعریف سیاست به صورت کلاس

ایجاد تابعی برای آموزش Agent :

در ابتدا Hyper parameter های مساله را به صورت زیر تعریف می‌کنیم :

- n_episodes: تعداد Iteration های مورد نیاز تا توقف آموزش
- max_t: تعداد state های پیمایش شده درون یک Iteration
- gamma: ضریب تخفیف که نشان می‌دهد چه مقدار از پاداش های آینده در زمان جال تاثیر پذیرد ، هر چه قدر این مقدار نزدیک به 1 باشد ، مساله پیچیده تر خواهد شد زیرا تعداد پارامتر های مساله افزایش می‌یابد.
- print_every: متغیری است که زمان چاپ شدن یک پیغام را کنترل می‌کند.

نحوه کار تابع نوشته شده بدین صورت است که در هر Iteration ، Agent به محل اول خود بازمیگردد (تمامی حالات reset می‌شوند) ، سپس طبق کلاس نوشته شده در بخش قبل که در مسیر forward تعادل میله و در قسمت act ، عمل اتخاذی را کنترل می‌کند ، برای ماکسیسم 1000 بار پیمایش حالت ، احتمال انجام هر action و reward مربوطه را تعیین می‌کنیم ، در نهایت برسی می‌کنیم که آیا شرط مساله برآورده شده است یا خیر .

```
def reinforce(policy, optimizer, n_episodes=1000, max_t=1000, gamma=1.0, print_every=100):
    scores_deque = deque(maxlen=100)
    scores = []
    for e in range(1, n_episodes):
        saved_log_probs = []
        rewards = []
        state = env.reset()
        # Collect trajectory
        for t in range(max_t):
            # Sample the action from current policy
            action, log_prob = policy.act(state)
            saved_log_probs.append(log_prob)
            state, reward, done, _ = env.step(action)
            rewards.append(reward)
            if done:
                break
```

شکل 1-5 : آموزش Agent (قسمت اول)

در صورتی که شرط مساله برقرار نبود ، میانگین پاداش های بدست آمده در هر مرحله را بدست آورده و به ترتیب ذخیره می‌کنیم :

```
# Calculate total expected reward
scores_deque.append(sum(rewards))
scores.append(sum(rewards))
```

شکل 1-6 : آموزش Agent (قسمت دوم)

بعد از اینکه آرایه از پاداش ها تشکیل شد ، طبق توضیحات اول سوال ، ضریب تخفیف را اثر می‌دهیم . (در اینجا gamma را برابر 1 گرفتیم ، بنابراین ، ارزش پاداش های زمان حال با آینده یکی می‌باشد) و پاداش اصلاح شده را بدست می‌آوریم :

```
# Recalculate the total reward applying discounted factor
discounts = [gamma ** i for i in range(len(rewards) + 1)]
R = sum([a * b for a,b in zip(discounts, rewards)])
```

شکل 1-7 : آموزش Agent (قسمت سوم)

حال نوبت به بهبود سیاست های اتخاذی می‌رسد . در ابتدا لیستی تهی برای ذخیره خطای حاصل از سیاست های پیشین ایجاد می‌کنیم . خطای ما به صورت منفی حاصل ضرب احتمال اتخاذ هر action در پاداش اصلاح شده مرحله قبل می‌باشد .

دلیل این تعریف این است که ما در نهایت می‌خواهیم احتمال انتخال هر action نزدیک به 1 باشد و همچنین پاداش دریافتی بیشینه شود . بنابراین معیار ضرب ، معیار مناسبی خواهد بود .

نکته نهایی این است که ما در جهت افزایش این دو حرکت می‌کنیم ، پس جهت مورد نظر صعود است (Gradient ascent) ، بنابراین علامت منفی را به این دلیل اضافه می‌کنیم :

```
# Calculate the loss
policy_loss = []
for log_prob in saved_log_probs:
    # using Gradient Ascent instead of Descent => need to insert negative rewards.
    policy_loss.append(-log_prob * R)
#concatenatenating whole policy loss in first dimension
policy_loss = torch.cat(policy_loss).sum()
```

شکل 1-8 : آموزش Agent (قسمت چهارم)

در نهایت ، مثل روال همیشه درتمامی مسائل ، دستورات مربوط به Backpropagation را می‌نویسم :

```
# Backpropagation
optimizer.zero_grad()
policy_loss.backward()
optimizer.step()
```

شکل 1-9 : آموزش Agent (قسمت پنجم)

در نهایت در هر 100 دور ، میانگین پاداش های بدست آمده را چاپ می‌کنیم . همچنین در صورتی که میانگین بدست آمده از یک یک threshold (در اینجا 195) بیشتر شد ، متوقف شویم

```
if e % print_every == 0:
    print('Episode {}\tAverage Score: {:.2f}'.format(e, np.mean(scores_deque)))
if np.mean(scores_deque) >= 195.0:
    print('Environment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(e - 100, np.mean(scores_deque)))
    break
```

شکل 1-10 : آموزش Agent (قسمت ششم)

➲ مشاهده نتیجه :

در این مرحله مدل را به gpu منتقل می‌کنیم و از بهینه ساز Adam به عنوان optimizer استفاده می‌کنیم.

```
policy = Policy().to(device)
optimizer = optim.Adam(policy.parameters(), lr=1e-2)
scores = reinforce(policy, optimizer, n_episodes=2000)
```

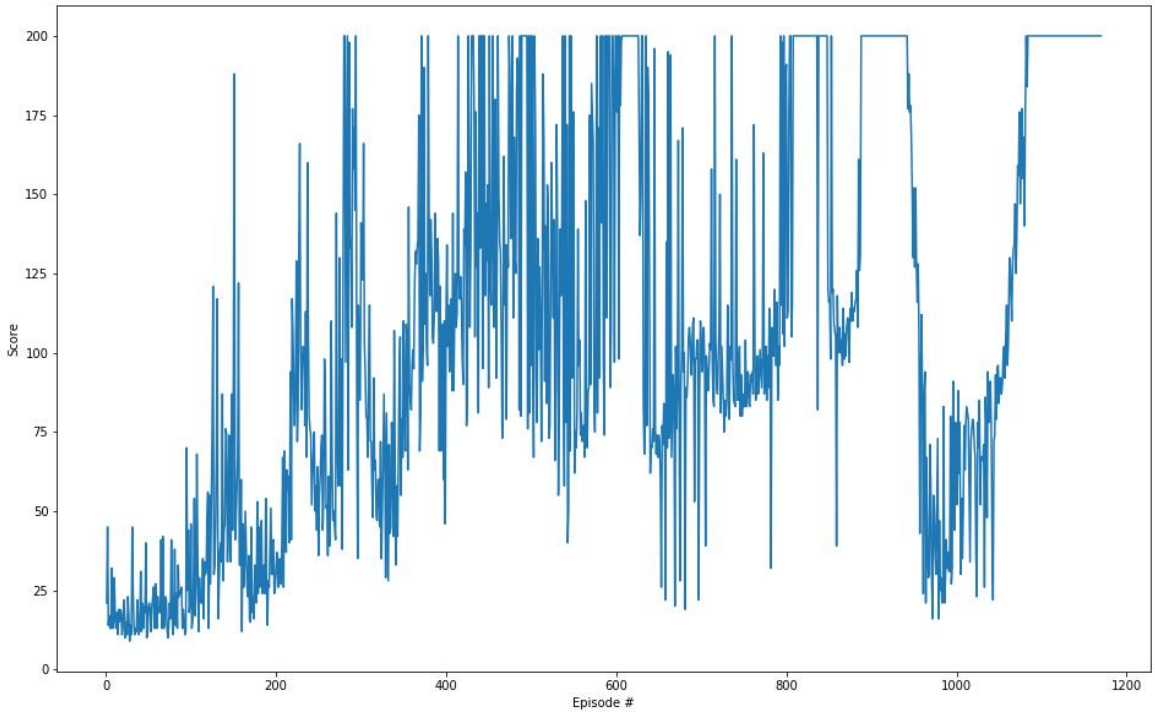
```
Episode 100     Average Score: 20.22
Episode 200     Average Score: 41.21
Episode 300     Average Score: 85.37
Episode 400     Average Score: 91.89
Episode 500     Average Score: 143.20
Episode 600     Average Score: 125.37
Episode 700     Average Score: 122.81
Episode 800     Average Score: 103.71
Episode 900     Average Score: 160.88
Episode 1000    Average Score: 126.33
Episode 1100    Average Score: 111.43
Environment solved in 1070 episodes!    Average Score: 195.44
```

شکل 1-11 : مشاهده نتیجه

همینطور که در شکل بالا مشخص است ، بعد از 1100 دور ، همگرا می‌شویم .

➲ رسم نمودار و انیمیشن بدست آمده :

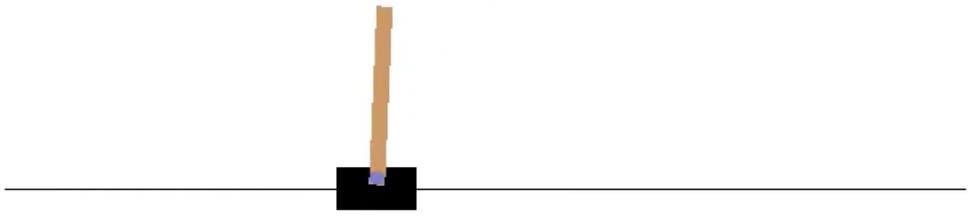
در نهایت نمودار میانگین پاداش دریافتی بر حسب تعداد Episode را رسم می‌کنیم:



شکل 1-12 : نمودار میانگین پاداش بر حسب دور

همچنین یک shot از انیمیشن بدست آمده از نحوه جابجایی cart و pole را آورده ایم (ویدیو بدست آمده در پوشه هموروک قرار داده شده است)

```
show_video('CartPole-v0')
```



شکل 1-13 : انیمشن بدست آمده

سوال 2 – Dealing with Large Discrete Action space

طبق مطالعات انجام شده ، جواب این سوال را به کمک این مقاله پاسخ می‌دهم.

طبق الگوریتم بحث شده در این مقاله ، در ابتدا یک محیط پیوسته از حالات (States) موجود تشکیل می‌دهیم. سپس به کمک تخمین نزدیک ترین همسایه ، نزدیکترین action های گسسته را در مقایس لگاریتمی زمان پیدا می‌کنیم :

our action set. Our policy produces a continuous action within this space, and then uses an approximate nearest-neighbor search to find the set of closest discrete actions in logarithmic time. We can either apply the closest action in this set directly to the environment, or fine-tune this selection by selecting the highest valued action in this set relative to a cost function. This approach allows for generalization over the action set in logarithmic time, which is necessary for making both learning and acting tractable in time.

شکل 2-1 : توضیحات کلی در مورد الگوریتم

همچنین در این قسمت مقاله ادعا می‌شود که این روش برای تعداد action بسیار زیاد هم کارا خواهد بود :

We demonstrate the effectiveness of our policy on various tasks with up to one million actions, but with the intent that our approach could scale well beyond millions of actions.

شکل 2-2 : ادعایی بر کارایی این روش

اسم الگوریتمی که در این مقاله به عنوان Proposed Method معرفی شده ، Wolpertinger می‌باشد

We propose a new policy architecture which we call the Wolpertinger architecture. This architecture avoids the heavy cost of evaluating all actions while retaining generalization over actions. This policy builds upon the actor-critic (Sutton & Barto, 1998) framework. We define both an efficient action-generating actor, and utilize the critic to refine our actor's choices for the full policy. We use multi-layer neural networks as function approximators for both our actor and critic functions. We train this policy using Deep Deterministic Policy Gradient (Lillicrap et al., 2015).

The Wolpertinger policy's algorithm is described fully in Algorithm 1 and illustrated in Figure 1. We will detail these in the following sections.

Algorithm 1 Wolpertinger Policy

State $\mathbf{s}$ 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 $r, \mathbf{s}'$.

.

شکل 2-3 : نحوه کار الگوریتم Wolpertinger

طبق توضیحات آورده شده ، این الگورتیم از بررسی یک جا تمامی action های موجود خودری می‌کند . روش کلی برگرفته از روش actor-critic می‌باشد که زیر مجموعه از روش Policy – Gradient می‌باشد . همچنین ذکر شده که هم actor و هم critic به صورت بهینه تعریف شده اند تا انتخاب actor را برای تمامی ساست ها بهبود بخشند. در نهایت در مورد معماری مدل ذکر شده که از شبکه های چند لایه (Multi-layer neural networks) هم برای actor و هم برای critic استفاده شده است .

در نهایت دیاگرام این الگوریتم را در زیر می‌آوریم :

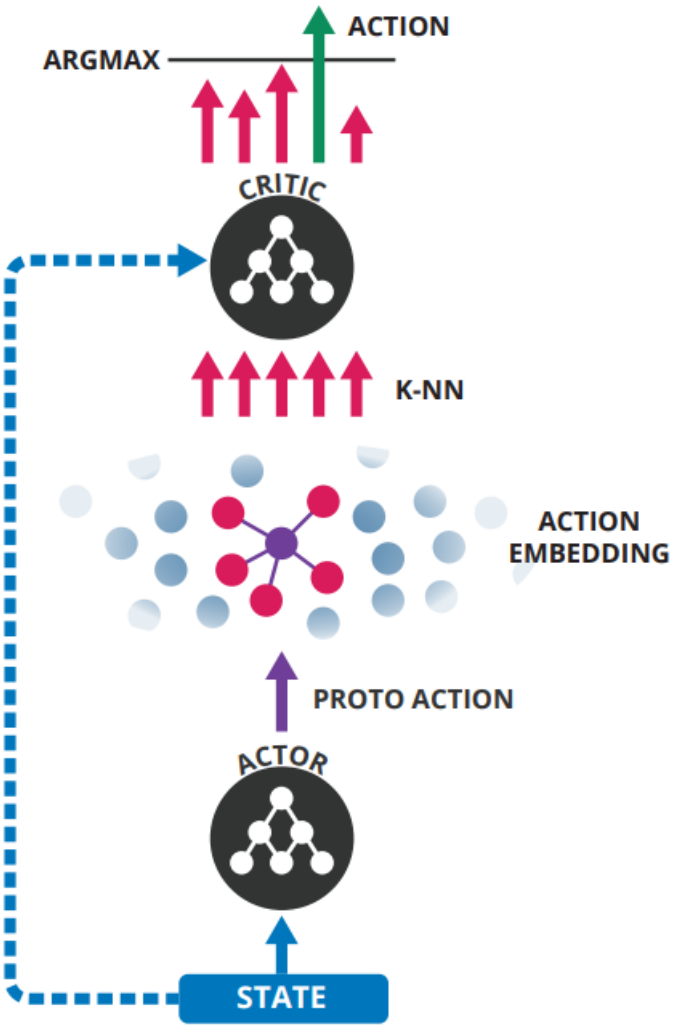


Figure 1. Wolpertinger Architecture

شکل 2-4 : دیاگرام الگوریتم Wolpertinger

طبق دیاگرام بالا ، با قرار گیری در هر state ، actor مجاز به انتخاب تعدادی action می‌باشد ، در ادامه طبق الگوریتم نزدیکترین همسایه ، نزدیکترین action ها به هم به critic داده می‌شود و critic با ماکسیمم گیری ، بهترین action موجود را انتخاب می‌کند و این loop تکرار می‌شود.