گذارش D3QN

قصد دارم که در این گذارش به توضیح وکد و الگوریتم D3QN بپردازم

بخش اول :

این بخش شامل فراخوانی پکیج های مورد نیاز برای الگوریتم هست که این پکیج ها به شرح زیر میباشند

```
import pygame,random,gymnasium as gym,numpy as np,matplotlib.pyplot as plt
import tensorflow as tf,os,warnings
losses ,from tensorflow.keras import optimizers
from tensorflow.keras import Model
from collections import deque
from tensorflow.python.framework import random_seed
from IPython.display import clear_output
```

سپس برای داشتن تجارب قابل پیش بینی و دوری از تصادفی بودن نتایج از seed استفاده میکنیم به شکل زیر تا با هر بار ران نتایج مشابهی داشته باشم

```
seed = 1
np.random.seed(seed)
np.random.default_rng(seed)
os.environ['PYTHONHASHSEED'] = str(seed)
random_seed.set_seed(seed)
random.seed(seed)
np.random.seed(seed)
tf.random.set_seed(seed)
```

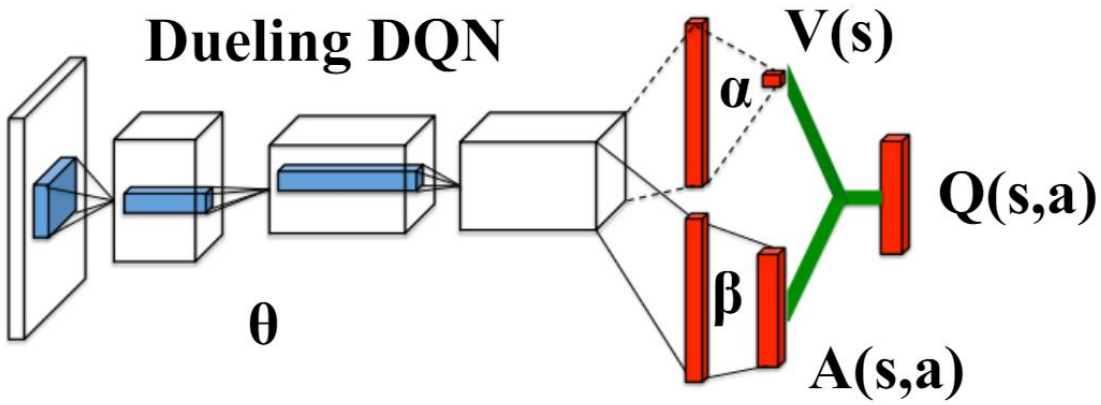
بخش 2:

این بخش شامل تعریف شبکه عصبی میباشد و محاسبه ارزش بر مبنای الگوریتم Dueling و با استفاده از محاسبه advantage می باشد که خود این با استفاده mean که از ارزش های عمل ها به دست می اید

```
class Network(Model):
int):  :hidden_size ,int :action_size ,int :state_size ,def __init__(self
init__()__.(self ,super(Network

self.num_action = action_size
activation='relu')#  ,self.layer1 = tf.keras.layers.Dense(hidden_size
Define the first hidden layer with ReLU activation
```

```
activation='relu')#  ,self.layer2 = tf.keras.layers.Dense(hidden_size
Define the second hidden layer with ReLU activation
self.state = tf.keras.layers.Dense(self.num_action)# Define the
output layer for state values
self.action = tf.keras.layers.Dense(self.num_action)# Define the
output layer for action values

state): ,def call(self
layer1 = self.layer1(state) # Pass the input state through
the first hidden layer
Pass the result through the second  #  (layer2 = self.layer2(layer1
hidden layer
state = self.state(layer2) # Compute the state values
action = self.action(layer2) # Compute the action values
keepdims=True)# Calculate the  ,mean = tf.keras.backend.mean(action
mean of the action values
mean)# Calculate the advantage by subtracting  - advantage = (action
the mean action value
advantage # Compute the final Q-values by adding  + value = state
state values and advantages

return value
```

و در این جا خود الگوریتم doeling رو داریم:



$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$$

بخش 3:

در این بخش به تعریف کلاس Agent و توابه ها میپردازم که جزایات کد کد را در این فایل مانند بخش قبل اورده نشد است چون این کلاس به نصبت بزرگ است و مانند بخش قبل فقط کلایات ان توضیح داده میشود همچنین برای توضیح هات بیشتر به خوبی به کد کامنت اضافه شده است.

```
class DQNAgent:
```

```
def __init__(
```

خوب در بخش اول این کلاس طبق معمول متد init را داریم که در وظیفه تعریف هایپر پارامتر ها را دارد و همچنین تعریف مواردی مانند optimizer این نتکه مهم است که بدانیم Adam یکی از بهترین optimizer هاست و در چند مقاله بررسی شده است

تابع بعدی تابع get_action است که به صورت E-greedy یک عمل را انتخواب میکند در صورتی که بخواهد بجای عمل تصادفی از انچه تا کنون یاد گرفته است استفاده کند.

```
epsilon) ,state ,def get_action(self
```

که برای توضیحات بیشتر به ان کامت اضافه شده است.

تابع بعدی وظیفه ذخیره کردن تجارب در یک استک را دارد برای استفاده در یادگیری مدل

```
done): ,next_state ,reward ,action ,state ,def append_sample(self
```

این تابع که یکی از مهم ترین توابع این کد است عمل یاد گیری را انجام میدهد که شامل چند بخش میباشد.

```
def train_step(self):
```

این تابع اپتدا به طور تصادفی یک مینی دسته را از حافظه پخش نمونه برداری میکند سپس با استفاده از مدل اصلی DQN، مقادیر Q را برای حالت های بعدی محاسبه می کن وبهترین اقدامات را برای حالت های بعدی انتخاب می کند سپس مقادیر Q هدف را با استفاده از مدل DQN هدف محاسبه می کند و یک ماسک در حالت ترمینال روی مقادیر Q هدف اعمال می کند و با توجه به گاما و پاداش مقادیر هدف را به روز میکند.

در بخش بعدی کد مقادیر Q را برای حالات فعلی محاسبه می کنیم و مقادیر Q مربوط به اقدامات انجام شده را جمع آوری می شود سپس خطاهای TD و loss را محاسبه میکنیم و از ان برای به روز کردن وزن ها استفاده میکنیم.

توضیه هر بخش مهم کد به صورت کامت به ان اضافه شده است

تابع مهم بعدی وظیفه با روز رسانی وزن های شبکه هدف را دارد که به صورت سخت رو نرم قابل پیاده سازی است کد برای هر دو حالت در این تابع هست ولی من توی تجاربم فقط از به روز رسانی سخت استفاده کردم و یک بار هم که این کار ور کردم تفاوت چندانی مشاهد نکردم

```
def _target_hard_update(self):
```

تابع بعدی تابعی است که در بخش 2 تمرین خواسته شده بود و در هر بار به روز رسانی مقدار گاما را افزایش میدهد وتا به حد 1 یک برسد خوب هدف این تابع کمک به کاوش است که باتوجه به تابع پاراشی که این محیط دارد چدادن تفاوتی اینجاد نمی کند و نمودار مربوط به ان با بقیه نمودار ها مقایسه خواهد شد

```
def update_Gamma(self):
self.gamma) - 0.985 * (1 - self.gamma = 1
```

دو تابع بعدی کار ذخیره سازی و باز گذاری وزن های شبکه برای اموزش را انجام مید دهند

```
phat): ,def load(self
self.dqn = load_model(phat)#
:'custom_objects={'Network ,self.dqn = tf.keras.models.load_model(phat
Network})
phat): ,def save(self
self.dqn.save(phat)
```

سپس یک instans از کلاس agent ساخته شده که مقادر مورد نیاز به ان داده شده

تابع بعدی برای ایجاد نمودار ها ایجاد شده است که به توضیحه ان نمی پیردازم

```
if __name__ == "__main__":
```

و در اخر با ساتفاده از شرط بالا چک میکنیم که کد های قبلی درست کار می کنند

سپس وارد حلقه اموزش یا تست میشویم

که ساختار ان مانند اکثر ایگوریتم های RL هست تنها این بخش از ان فبل ذکر هست که عمل امورش را اتجام میدهد البته وقتی که زمانش فرا برسد و سپس وقتی که شرط برای به روز سانی شبکه هدف فرا برسد انرا به روز رسانی میکنیم

```
if (update_cnt >= agent.batch_size):
agent.train_step()
agent.update_Gamma()#
if update_cnt % agent.target_update == 0:
target_hard_update()_.agent
```

در کد بالا وقتی که بخواهیم که گاما را به صورت تدریجی افزایش بدیم خط مربوط به ان را از کامنت بودن در میاریم

در تاهنکام تست هم وزن ها رو باز کذاری میکینم و با یک گاما و Epsilon ثابت عمل میکند و مانند حلقه اموزش یک عمل به صورت حریسانه انتخاب میکند و انرا به محیط میدهیم و پاراش و مکان بعدی را می گیریم

در یک فایل جدا به صورت جداگانه نتایج دو الگوریتم را مقایسه میکنم

تشکر از توجه شما

پایان