RL_PA2_DQN_CS21D407_CS22E005

March 27, 2023

0.0.1 Connecting the notebook with Google Drive

[1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

0.0.2 Installing packages for rendering the game on Colab

[2]:
```
!pip install gym pyvirtualdisplay > /dev/null 2>&1
!apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
!apt-get update > /dev/null 2>&1
!apt-get install cmake > /dev/null 2>&1
!pip install --upgrade setuptools 2>&1
!pip install ez_setup > /dev/null 2>&1
!pip install gym[atari] > /dev/null 2>&1
!pip install git+https://github.com/tensorflow/docs > /dev/null 2>&1
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: setuptools in /usr/local/lib/python3.9/dist-
packages (67.6.0)
```

0.0.3 Importing the necessary packages

[3]:
```
import numpy as np
import random
import torch
import torch.nn as nn
import torch.nn.functional as F
from collections import namedtuple, deque
import torch.optim as optim
from torch.distributions import Categorical
import gym
import matplotlib.pyplot as plt
import os
from os import listdir
from os.path import isfile, join
```

```
import re
```

1 DQN Implementation

[5]:
```
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

1.0.1 Q-Network

[6]:
```
class QNetwork(nn.Module):
  def __init__(self, input, output, layers=[]):
    super(QNetwork, self).__init__()
    self.hidden = nn.ModuleList()
    in_layer = input
    for out_layer in layers:
      self.hidden.append(nn.Linear(in_layer, out_layer))
      in_layer = out_layer
    self.out = nn.Linear(in_layer, output)

  def forward(self, x):
    for fc in self.hidden:
      x = F.relu(fc(x))
    return self.out(x)
```

1.0.2 Replay Buffer

[7]:
```
class ReplayBuffer(object):
  def __init__(self, buffer_size):
    self.memory = deque(maxlen=buffer_size)
    self.experience = namedtuple("Experience", field_names=["state", "action",
↪"reward", "next_state", "done"])

  def __len__(self):
    return len(self.memory)

  def add(self, state, action, reward, next_state, done):
    e = self.experience(state, action, reward, next_state, done)
    self.memory.append(e)

  def sample(self, batch_size):
    experiences = random.sample(self.memory, k=batch_size)
    states = torch.from_numpy(np.vstack([e.state for e in experiences if e is
↪not None])).float().to(DEVICE)
    actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is
↪not None])).long().to(DEVICE)
```

```
    rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is␣
↪not None])).float().to(DEVICE)
    next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences␣
↪if e is not None])).float().to(DEVICE)
    dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not␣
↪None]).astype(np.uint8)).float().to(DEVICE)
    return (states, actions, rewards, next_states, dones)
```

1.0.3 Implements the DQN Agent

[8]:
```
class DQN_Agent(object):
  def __init__(self, state_size, action_size, layers=[],
               gamma=0.9, lr=0.001, buffer_size=1e3,
               batch_size=64, update_target=20, truncation_limit=1):
    self.q_local = QNetwork(state_size, action_size, layers).to(DEVICE)
    self.q_target = QNetwork(state_size, action_size, layers).to(DEVICE)
    self.optimizer = optim.Adam(self.q_local.parameters(), lr=lr)
    self.memory = ReplayBuffer(buffer_size)
    self.gamma = gamma
    self.batch_size = batch_size
    self.update_target = update_target
    self.truncation_limit = truncation_limit
    self.t = 0

  def softmax(self, values, beta):
    """
    Implements the Softmax Exploration Strategy
    """
    values /= beta
    prob = np.exp(values - np.max(values))
    prob /= np.sum(prob)
    return np.random.choice(len(values), p=prob)

  def get_action(self, state, explore=0.0):
    """
    Returns an action for a given state
    """

    state = torch.from_numpy(state).float().unsqueeze(0).to(DEVICE)
    self.q_local.eval()
    with torch.no_grad():
        action_values = self.q_local(state)
    self.q_local.train()
    qvalues = action_values.cpu().data.numpy()[0]
    return self.softmax(qvalues, explore)
```

```
  def update(self, state, action, reward, next_state, done):
    """
    Updates the Q-network parameters
    """

    self.memory.add(state, action, reward, next_state, done)
    if len(self.memory) >= self.batch_size:
      exps = self.memory.sample(self.batch_size)
      self.train(exps)
    self.t = (self.t + 1) % self.update_target
    if self.t == 0:
      self.q_target.load_state_dict(self.q_local.state_dict())

  def train(self, exps):
    """
    Updates the Q-network parameters
    """

    states, actions, rewards, next_states, dones = exps
    q_targets_next = self.q_target(next_states).detach().max(1)[0].unsqueeze(1)
    targets = rewards + (self.gamma * q_targets_next * (1 - dones))
    expected = self.q_local(states).gather(1, actions)
    loss = F.mse_loss(targets, expected)
    self.optimizer.zero_grad()
    loss.backward()
    for param in self.q_local.parameters():
      param.grad.data.clamp_(-self.truncation_limit, self.truncation_limit)
    self.optimizer.step()
```

1.0.4 Runs a specified number of experiments(in our case 10) for a particular hyper-parameter configuration

[18]:
```
class DQN_Runner(object):
  def __init__(self, params):
    self.params = params
    self.env = gym.make(params['environment'])
    self.truncation_rwd = params["reward threshold"]
    self.explore_start = params['explore_start']
    self.explore_end = params['explore_end']
    self.decay = params['decay'] if 'decay' in params else 0.995
    self.runs = params['runs']
    self.episodes = params['episodes']
    self.max_t = params['max_t']
    self.save_path = params['save_path']
    self.name = f"hid layers:{params['layers']}, gamma:{params['gamma']}, lr:
↪{params['lr']}"+\
```

```python
                f"replay buffer:{params['buffer_size']}, batch size:
↪{params['batch_size']}"+\
                f"update target:{params['update_target']}, trunc limit:
↪{params['truncation_limit']}"+\
                f"explore end:{params['explore_end']}, decay:{params['decay']}"

  def reset(self):
    """
    Initialises a DQN agent
    """

    params = self.params
    self.env.reset()
    state_size, action_size = self.env.observation_space.shape[0], self.env.
↪action_space.n
    self.agent = DQN_Agent(state_size=state_size,
                           action_size=action_size,
                           layers=params["layers"],
                           gamma=params['gamma'],
                           lr=params['lr'],
                           buffer_size=params['buffer_size'],
                           batch_size=params['batch_size'],
                           update_target=params['update_target'],
                           truncation_limit=params['truncation_limit'])

  def single_run(self, print_after=100):
    """
    Implements a single run for the given hyperparameter configuaration
    """

    self.reset()
    explore = self.explore_start
    episodes_reward, episodes_step = [], []
    reward_window = deque(maxlen=100)
    for episode in range(1, self.episodes+1):
      state = self.env.reset()
      total_rwd, done = 0, False
      step = 0
      while (not done) and (step < self.max_t):
        action = self.agent.get_action(state, explore)
        next_state, rwd, done, _ = self.env.step(action)
        self.agent.update(state, action, rwd, next_state, done)
        state = next_state
        total_rwd += rwd
        step += 1
      episodes_reward.append(total_rwd)
      episodes_step.append(step)
```

```python
      reward_window.append(total_rwd)
      explore = max(self.explore_end, explore*self.decay)
      if (episode % print_after) == 0:
        print(f"Episode: {episode}\tAverage Reward: {np.mean(reward_window)}")
      if np.mean(reward_window) > self.truncation_rwd:
        print(f"Environment solved in {episode} episodes!\tAverage Reward: {np.
↪mean(reward_window)}")
        break
    episodes_reward.extend([total_rwd]*(self.episodes - episode))
    episodes_step.extend([step]*(self.episodes - episode))
    return episodes_reward, episodes_step

  def avg_run(self):
    """
    Averages the performance of the agent across a given number of runs
    """

    avg_rwd_list = np.zeros(self.episodes)
    avg_step_list = np.zeros(self.episodes)
    for run in range(self.runs):
      rwd_list, step_list = self.single_run()
      rwd_list, step_list = np.array(rwd_list), np.array(step_list)
      avg_rwd_list += (rwd_list - avg_rwd_list) / (run+1)
      avg_step_list += (step_list - avg_step_list) / (run+1)
      print("-"*10)
    return avg_rwd_list, avg_step_list

  def save_and_plot(self):
    """
    Plots the average rewards and average number of steps curve
    """
    avg_rwd_list, avg_step_list = self.avg_run()
    self.save(avg_rwd_list, f"{self.name}_rwd")
    self.save(avg_step_list, f"{self.name}_step")
    self.plot(avg_rwd_list, "Average Reward")
    self.plot(avg_step_list, "Average Step")

  def save(self, save_list, name):
    path = os.path.join(self.save_path, self.env.spec.id)
    if not os.path.exists(path):
      os.makedirs(path)
    filename = os.path.join(path, name)
    with open(filename, 'w') as fp:
      for rwd in save_list:
        fp.write(f"{rwd}\n")

  def plot(self, data_list, name):
```

```
    plt.title(self.name)
    plt.ylabel(name)
    plt.xlabel("Episode")
    index = 0
    for index in range(len(data_list)):
      if data_list[index] == data_list[index+1]:
        j = index+1
        value = data_list[j]
        flag = True
        while j < len(data_list)-1:
          if value != data_list[j+1]:
            flag = False
            break
          j += 1
        if flag and j == len(data_list)-1:
          data_list = data_list[:index+1]
          break
    plt.plot(np.arange(len(data_list)), data_list)
    plt.show()
```

1.0.5 Hyperparameter configuration can be specified here

[22]:
```
parameters = {
    "environment": "CartPole-v1",
    "reward threshold": 475,
    "layers": [128, 128],
    "gamma": 1,
    "lr": 1e-3,
    "batch_size": 256,
    "truncation_limit": 2,
    "buffer_size": int(1e5),
    "update_target": 20,
    "explore_start": 1.0,
    "explore_end": 0.1,
    "decay": 0.9,
    "runs": 10,
    "episodes": 3000,
    "max_t": 1000,
    "save_path": "/content/drive/MyDrive/PA2/DQN"
}
```

[1]:
```
DQN = DQN_Runner(parameters)
DQN.save_and_plot()
```

[2]:
```
from google.colab import drive
drive.mount('/content/drive')
!pip install nbconvert
```

```
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic
```

[4]:

```
!jupyter nbconvert --to pdf "/content/drive/MyDrive/Colab Notebooks/
↪RL_PA2_DQN_CS21D407_CS22E005.ipynb"
```

```
[NbConvertApp] Converting notebook /content/drive/MyDrive/Colab
Notebooks/RL_PA2_DQN_CS21D407_CS22E005.ipynb to pdf
[NbConvertApp] Writing 59980 bytes to notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', 'notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', 'notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 63471 bytes to /content/drive/MyDrive/Colab
Notebooks/RL_PA2_DQN_CS21D407_CS22E005.pdf
```