RL_Deep_Q_Week_4

November 10, 2020

1 Import Package

[1]:
```
import pyvirtualdisplay
import matplotlib.pyplot as plt
import tensorflow as tf
from tf_agents.agents.dqn import dqn_agent
from tf_agents.environments import suite_gym
from tf_agents.environments import tf_py_environment
from tf_agents.networks import q_network
from tf_agents.utils import common
from tf_agents.replay_buffers import tf_uniform_replay_buffer
from tf_agents.trajectories import trajectory
import PIL.Image
from tf_agents.policies import random_tf_policy
```

[2]:
```
tf.compat.v1.enable_v2_behavior()
# Set up a virtual display for rendering OpenAI gym environments.
#display = pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()
```

2 Hyperparameters

[3]:
```
num_iterations = 20000 # @param {type:"integer"}

initial_collect_steps = 100  # @param {type:"integer"}
collect_steps_per_iteration = 1  # @param {type:"integer"}
replay_buffer_max_length = 100000  # @param {type:"integer"}

batch_size = 64  # @param {type:"integer"}
learning_rate = 1e-3  # @param {type:"number"}
log_interval = 200  # @param {type:"integer"}

num_eval_episodes = 10  # @param {type:"integer"}
eval_interval = 1000  # @param {type:"integer"}
```

3 Environment

[4]:
```
env_name = 'CartPole-v0'
env = suite_gym.load(env_name)
```

[5]:
```
# Train and Evaluation(Testing)
train_py_env = suite_gym.load(env_name)
eval_py_env = suite_gym.load(env_name)

# Python to TF
train_env = tf_py_environment.TFPyEnvironment(train_py_env)
eval_env = tf_py_environment.TFPyEnvironment(eval_py_env)
```

4 Agent / Algorithm

Y(s,a,r,s)=r+ maxa Q −(s ,a)

L( )= (s,a,r,s) U(D)[(Y(s,a,r,s)−Q (s,a))2]

4.1 Create a QNetwork

Feed Forward network

[6]:
```
fc_layer_params = (100,) # initial_collect_steps? input layer?

q_net = q_network.QNetwork(
    train_env.observation_spec(),
    train_env.action_spec(),
    # A list of fully_connected parameters,
    # where each item is the number of units in the layer
    fc_layer_params=fc_layer_params)
#q_net
```

4.2 Instantiate a DqnAgent

Implements the DQN algorithm from

“Human level control through deep reinforcement learning” Mnih et al., 2015

https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

[7]:
```
optimizer = tf.compat.v1.train.AdamOptimizer(learning_rate=learning_rate) # Adam algorithm

train_step_counter = tf.Variable(0)

agent = dqn_agent.DqnAgent(
    train_env.time_step_spec(),
    train_env.action_spec(),
```

```
    q_network=q_net,
    optimizer=optimizer, # AdamOptimizer function
    td_errors_loss_fn=common.element_wise_squared_loss, # loss function
    train_step_counter=train_step_counter) # integer step counter

agent.initialize()
```

4.3 Policies/Rules

The rules will return an action to produce the desired rewards

[8]:
```
eval_policy = agent.policy # The main policy that is used for evaluation and␣
 ↪deployment.
collect_policy = agent.collect_policy # A second policy that is used for data␣
 ↪collection.
```

5 Evaluation

Computes the average return of a policy per episode.

[9]:
```
def compute_avg_return(environment, policy, num_episodes=10):

  total_return = 0.0
  for _ in range(num_episodes):

    time_step = environment.reset()
    episode_return = 0.0

    while not time_step.is_last():
      action_step = policy.action(time_step)
      time_step = environment.step(action_step.action)
      episode_return += time_step.reward
    total_return += episode_return

  avg_return = total_return / num_episodes
  return avg_return.numpy()[0]
```

Show baseline performance by randomly selection.

[10]:
```
random_policy = random_tf_policy.RandomTFPolicy(train_env.time_step_spec(),
                                                train_env.action_spec())
compute_avg_return(eval_env, random_policy, num_eval_episodes)
```

[10]:
```
18.7
```

6 Data Collection

6.1 Replay Buffer

Dictionary?

[11]:
```
replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec,
    batch_size=train_env.batch_size,
    # The maximum number of items that can be stored in a
    # single batch segment of the buffer
    max_length=replay_buffer_max_length)

agent.collect_data_spec, agent.collect_data_spec._fields
```

[11]:
```
(Trajectory(step_type=TensorSpec(shape=(), dtype=tf.int32, name='step_type'),
observation=BoundedTensorSpec(shape=(4,), dtype=tf.float32, name='observation',
minimum=array([-4.8000002e+00, -3.4028235e+38, -4.1887903e-01, -3.4028235e+38],
       dtype=float32), maximum=array([4.8000002e+00, 3.4028235e+38,
4.1887903e-01, 3.4028235e+38],
       dtype=float32)), action=BoundedTensorSpec(shape=(), dtype=tf.int64,
name='action', minimum=array(0), maximum=array(1)), policy_info=(),
next_step_type=TensorSpec(shape=(), dtype=tf.int32, name='step_type'),
reward=TensorSpec(shape=(), dtype=tf.float32, name='reward'),
discount=BoundedTensorSpec(shape=(), dtype=tf.float32, name='discount',
minimum=array(0., dtype=float32), maximum=array(1., dtype=float32))),
 ('step_type',
  'observation',
  'action',
  'policy_info',
  'next_step_type',
  'reward',
  'discount'))
```

Recording the data in the replay buffer.

[12]:
```
def collect_step(environment, policy, buffer):
  time_step = environment.current_time_step()
  action_step = policy.action(time_step)
  next_time_step = environment.step(action_step.action)
  traj = trajectory.from_transition(time_step, action_step, next_time_step)

  # Add trajectory to the replay buffer
  buffer.add_batch(traj)

def collect_data(env, policy, buffer, steps):
  for _ in range(steps):
    collect_step(env, policy, buffer)
```

```
collect_data(train_env, random_policy, replay_buffer, initial_collect_steps)
```

Covert the dictionary as a data set and return an iterator.

[13]:
```
dataset = replay_buffer.as_dataset(
    num_parallel_calls=3,
    sample_batch_size=batch_size,
    num_steps=2).prefetch(3)
#dataset
iterator = iter(dataset)
```

```
WARNING:tensorflow:From /opt/anaconda3/lib/python3.8/site-
packages/tensorflow/python/autograph/operators/control_flow.py:1004:
ReplayBuffer.get_next (from tf_agents.replay_buffers.replay_buffer) is
deprecated and will be removed in a future version.
Instructions for updating:
Use `as_dataset(…, single_deterministic_pass=False) instead.
```

7 Train

Two things must happen during the training loop:

collect data from the environment

use that data to train the agent's neural network(s)

[14]:
```
try:
  %%time
except:
  pass
# (Optional) Optimize by wrapping some of the code in a graph using TF function.
agent.train = common.function(agent.train)

# Reset the train step
agent.train_step_counter.assign(0)

# Evaluate the agent's policy once before training.
avg_return = compute_avg_return(eval_env, agent.policy, num_eval_episodes)
returns = [avg_return]

for _ in range(num_iterations):

  # Collect a few steps using collect_policy and save to the replay buffer.
  collect_data(train_env, agent.collect_policy, replay_buffer,␣
↪collect_steps_per_iteration)

  # Sample a batch of data from the buffer and update the agent's network.
```

```python
  experience, unused_info = next(iterator)
  train_loss = agent.train(experience).loss

  step = agent.train_step_counter.numpy()

  if step % log_interval == 0:
    print('step = {0}: loss = {1}'.format(step, train_loss))

  if step % eval_interval == 0:
    avg_return = compute_avg_return(eval_env, agent.policy, num_eval_episodes)
    print('step = {0}: Average Return = {1}'.format(step, avg_return))
    returns.append(avg_return)
```

```
WARNING:tensorflow:From /opt/anaconda3/lib/python3.8/site-
packages/tensorflow/python/util/dispatch.py:201: calling foldr_v2 (from
tensorflow.python.ops.functional_ops) with back_prop=False is deprecated and
will be removed in a future version.
Instructions for updating:
back_prop=False is deprecated. Consider using tf.stop_gradient instead.
Instead of:
results = tf.foldr(fn, elems, back_prop=False)
Use:
results = tf.nest.map_structure(tf.stop_gradient, tf.foldr(fn, elems))
step = 200: loss = 18.008861541748047
step = 400: loss = 5.086471080780029
step = 600: loss = 15.28239631652832
step = 800: loss = 9.468496322631836
step = 1000: loss = 11.550569534301758
step = 1000: Average Return = 15.699999809265137
step = 1200: loss = 35.28739929199219
step = 1400: loss = 2.0952863693237305
step = 1600: loss = 2.0557005405426025
step = 1800: loss = 25.886310577392578
step = 2000: loss = 5.963220119476318
step = 2000: Average Return = 24.299999237060547
step = 2200: loss = 11.770583152770996
step = 2400: loss = 8.902961730957031
step = 2600: loss = 37.822486877441406
step = 2800: loss = 6.025323867797852
step = 3000: loss = 9.689393043518066
step = 3000: Average Return = 110.4000015258789
step = 3200: loss = 22.36556625366211
step = 3400: loss = 32.306640625
step = 3600: loss = 15.72234058380127
step = 3800: loss = 6.93696403503418
step = 4000: loss = 50.098915100097656
step = 4000: Average Return = 90.0
```

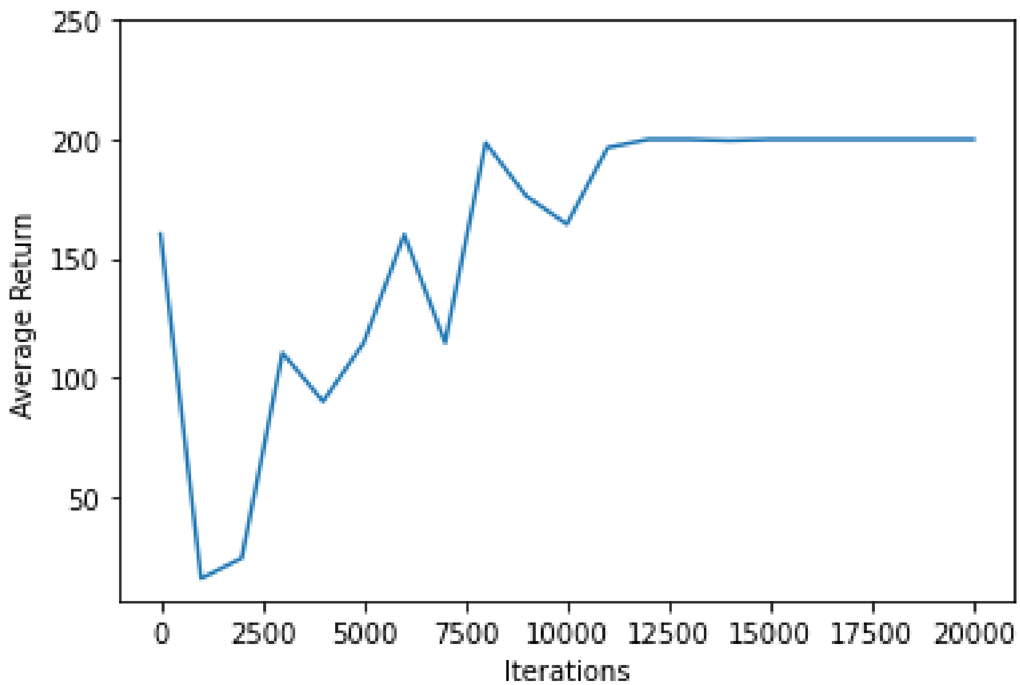
```
step = 4200: loss = 47.52417755126953
step = 4400: loss = 155.73275756835938
step = 4600: loss = 153.7744140625
step = 4800: loss = 112.13093566894531
step = 5000: loss = 15.244081497192383
step = 5000: Average Return = 114.5999984741211
step = 5200: loss = 21.695892333984375
step = 5400: loss = 5.562005996704102
step = 5600: loss = 6.435539245605469
step = 5800: loss = 59.24911117553711
step = 6000: loss = 7.506043910980225
step = 6000: Average Return = 160.0
step = 6200: loss = 65.04222869873047
step = 6400: loss = 217.68240356445312
step = 6600: loss = 174.95286560058594
step = 6800: loss = 164.0635528564453
step = 7000: loss = 10.535503387451172
step = 7000: Average Return = 114.69999694824219
step = 7200: loss = 117.93932342529297
step = 7400: loss = 67.07676696777344
step = 7600: loss = 10.39604377746582
step = 7800: loss = 67.12391662597656
step = 8000: loss = 105.36053466796875
step = 8000: Average Return = 198.5
step = 8200: loss = 118.29344177246094
step = 8400: loss = 140.25460815429688
step = 8600: loss = 30.344758987426758
step = 8800: loss = 11.699833869934082
step = 9000: loss = 476.17034912109375
step = 9000: Average Return = 176.1999969482422
step = 9200: loss = 451.31793212890625
step = 9400: loss = 20.707763671875
step = 9600: loss = 28.288009643554688
step = 9800: loss = 125.00628662109375
step = 10000: loss = 165.96885681152344
step = 10000: Average Return = 164.39999389648438
step = 10200: loss = 234.79196166992188
step = 10400: loss = 34.319217681884766
step = 10600: loss = 73.28331756591797
step = 10800: loss = 21.278406143188477
step = 11000: loss = 207.4669189453125
step = 11000: Average Return = 196.5
step = 11200: loss = 9.321070671081543
step = 11400: loss = 34.81235885620117
step = 11600: loss = 24.20108985900879
step = 11800: loss = 218.18382263183594
step = 12000: loss = 54.903507232666016
step = 12000: Average Return = 200.0
```

```
step = 12200: loss = 696.0888061523438
step = 12400: loss = 118.06753540039062
step = 12600: loss = 29.727664947509766
step = 12800: loss = 172.8981475830078
step = 13000: loss = 36.71021270751953
step = 13000: Average Return = 200.0
step = 13200: loss = 43.96297836303711
step = 13400: loss = 187.39060974121094
step = 13600: loss = 699.5609741210938
step = 13800: loss = 43.06932830810547
step = 14000: loss = 262.7250061035156
step = 14000: Average Return = 199.5
step = 14200: loss = 44.86981201171875
step = 14400: loss = 788.365966796875
step = 14600: loss = 186.86534118652344
step = 14800: loss = 34.16826248168945
step = 15000: loss = 68.5713882446289
step = 15000: Average Return = 200.0
step = 15200: loss = 36.6361083984375
step = 15400: loss = 67.6352767944336
step = 15600: loss = 88.42098999023438
step = 15800: loss = 66.49943542480469
step = 16000: loss = 34.3782958984375
step = 16000: Average Return = 200.0
step = 16200: loss = 51.016273498535156
step = 16400: loss = 979.0067138671875
step = 16600: loss = 20.1558780670166
step = 16800: loss = 74.05459594726562
step = 17000: loss = 27.95159912109375
step = 17000: Average Return = 200.0
step = 17200: loss = 149.70065307617188
step = 17400: loss = 16.747379302978516
step = 17600: loss = 678.1433715820312
step = 17800: loss = 41.89617919921875
step = 18000: loss = 609.5775756835938
step = 18000: Average Return = 200.0
step = 18200: loss = 454.809326171875
step = 18400: loss = 48.18207550048828
step = 18600: loss = 48.29536437988281
step = 18800: loss = 122.12271118164062
step = 19000: loss = 1473.701416015625
step = 19000: Average Return = 200.0
step = 19200: loss = 135.0766143798828
step = 19400: loss = 408.56280517578125
step = 19600: loss = 489.67938232421875
step = 19800: loss = 107.77105712890625
step = 20000: loss = 753.5126953125
step = 20000: Average Return = 200.0
```

8 Visualization

[15]:
```
iterations = range(0, num_iterations + 1, eval_interval)
plt.plot(iterations, returns)
plt.ylabel('Average Return')
plt.xlabel('Iterations')
plt.ylim(top=250)
```

[15]: (6.484999799728394, 250.0)



9 Reference

https://www.tensorflow.org/agents/tutorials/1_dqn_tutorial