report_2403967

December 3, 2021

0.1 Countering COVID with The Best Stratergy

Khawaja Muhhammad Saad Butt - 2403967b
University of Glasgow

0.2 Abstract

COVID-19 has been an epidemic which took the world by storm and has had it's svere consequences. Governemnts across the world had taken restrictive measures and in this report/assessed exercise I will present one of the optimal stratergies which the Governemnts can take to counter the COVID-19 problem. The report demonstrate how and which stratergy should the Government opt for and how would it impact the entire year in terms of COVID situation. To figure out such a policy, the code below demonstrates different agents that have been implemented along with their evaluation.

[6]:
```
# Obtain the notebook JSON as a string
from google.colab import _message
notebook_json = _message.blocking_request('get_ipynb', request='',␣
↪timeout_sec=10)

# collate all text in Markdown cells
all_text = ''
for cell in notebook_json['ipynb']['cells']:
    if cell['cell_type'] == "markdown":
        all_text += ' '.join(cell['source'])
# replace # and \n by empty space
all_text = all_text.replace('#', '').replace('\n', '')
# find main section and reference & appendix section
before_eof, eof, after_eof = all_text.partition('===EOF===') # please do not␣
↪temper with this

# count words per section, our counting method is simple and probably plays in␣
↪your favor
report_word_count = len(before_eof.split())
remaining_word_count = len(after_eof.split())

print("Your report currently has {} words".format(report_word_count))
```

```
print("Your reference and appendix currently have {} words".
 ↪format(remaining_word_count))

# Please respect this convention and work with it, not against it.
# We will run an independant word count check on all notebooks submitted
```

```
Your report currently has 4828 words
Your reference and appendix currently have 377 words
```

1 1. Introduction

Your mission is to design, implement, evaluate and document a number of virtual agents which can learn (optimal) COVID-19 mitigation policies.

1.1 1.1 Motivation

COVID-19 has been an epidemic which took the world by storm and has had it's svere consequences. Governemnts across the world had taken restrictive measures and in this report/assessed exercise I will present one of the optimal stratergies which the Governemnts can take to counter the COVID-19 problem.

To demonstrate the stratergies the Governments can take and the impact they would have on the the COVID situation is presented below and later evaluated to show it's success.

Such an optimaml policy can be implemented by the some latest algortihm such Reinforcement Learning which is explored in the report below with comparison of Diterministic agents and Rnadom agent.

1.2 1.2 Task Environment

ViRL is an environment which would be our main course of attention. ViRL is an Epidemics Reinforcement Learning Environment which helps in exploring the several policies that help to reduce the spread of COVID-19 virus.

ViRL is available at https://git.dcs.gla.ac.uk/SebastianStein/virl, the readme gives more information about the environment.

External libraries, like ViRL, can be installed directly from inside the notebook as follow:

[ ]:
```
!git clone https://git.dcs.gla.ac.uk/SebastianStein/virl.git
```

```
fatal: destination path 'virl' already exists and is not an empty directory.
```

Once cloned from GitLab, you can add the virl folder to the path where Python can look for libraries (sys.path)

```
[ ]: ## to import virl, we add the virl folder cloned above to the path where Python
     ↪can look for libraries (sys.path)
     import sys
     sys.path.append('virl')
     import virl
```

The ViRL library can now be used directly from this notebook

The are all the libraries required to solve the assessed exercise

1.3 Note: Usually the first run, the imports will cause an error. In that case, please restart runtime and then run it again. That will fix it.

```
[ ]: from matplotlib import pyplot as plt
     import numpy as np
     import random
     import numpy as np
     import tensorflow as tf
     from tensorflow import keras
     from tensorflow.keras import layers

     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Dense
     from tensorflow.keras.optimizers import Adam
     from tensorflow.keras import backend as K

     import matplotlib.pyplot as plt
     import time
     import itertools
     import matplotlib
     from mpl_toolkits.mplot3d import Axes3D
     from collections import deque
     import pandas as pd
     import statistics

     import numpy as np
     import sys
     import os
     import random
     from collections import namedtuple
     import collections
     import copy

     !pip install tensorflow==1.14.0
     !pip install keras==2.2.4
```

```
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:516: FutureWarning: Passing
```

```
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:517: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:518: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:519: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:520: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:525: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:541: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:542: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:543: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:544: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
```

```
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:545: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:550: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])

Requirement already satisfied: tensorflow==1.14.0 in /usr/local/lib/python3.7
/dist-packages (1.14.0)
Requirement already satisfied: tensorflow-estimator<1.15.0rc0,>=1.14.0rc0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow==1.14.0) (1.14.0)
Requirement already satisfied: absl-py>=0.7.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow==1.14.0) (0.12.0)
Requirement already satisfied: wrapt>=1.11.1 in /usr/local/lib/python3.7/dist-
packages (from tensorflow==1.14.0) (1.13.3)
Requirement already satisfied: wheel>=0.26 in /usr/local/lib/python3.7/dist-
packages (from tensorflow==1.14.0) (0.37.0)
Requirement already satisfied: protobuf>=3.6.1 in /usr/local/lib/python3.7/dist-
packages (from tensorflow==1.14.0) (3.17.3)
Requirement already satisfied: keras-applications>=1.0.6 in
/usr/local/lib/python3.7/dist-packages (from tensorflow==1.14.0) (1.0.8)
Requirement already satisfied: gast>=0.2.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow==1.14.0) (0.4.0)
Requirement already satisfied: keras-preprocessing>=1.0.5 in
/usr/local/lib/python3.7/dist-packages (from tensorflow==1.14.0) (1.1.2)
Requirement already satisfied: numpy<2.0,>=1.14.5 in /usr/local/lib/python3.7
/dist-packages (from tensorflow==1.14.0) (1.19.5)
Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow==1.14.0) (1.15.0)
Requirement already satisfied: astor>=0.6.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow==1.14.0) (0.8.1)
Requirement already satisfied: tensorboard<1.15.0,>=1.14.0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow==1.14.0) (1.14.0)
Requirement already satisfied: grpcio>=1.8.6 in /usr/local/lib/python3.7/dist-
packages (from tensorflow==1.14.0) (1.42.0)
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.7
/dist-packages (from tensorflow==1.14.0) (1.1.0)
Requirement already satisfied: google-pasta>=0.1.6 in /usr/local/lib/python3.7
/dist-packages (from tensorflow==1.14.0) (0.2.0)
Requirement already satisfied: h5py in /usr/local/lib/python3.7/dist-packages
(from keras-applications>=1.0.6->tensorflow==1.14.0) (3.1.0)
Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.7
/dist-packages (from tensorboard<1.15.0,>=1.14.0->tensorflow==1.14.0) (1.0.1)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/dist-
```

```
packages (from tensorboard<1.15.0,>=1.14.0->tensorflow==1.14.0) (3.3.6)
Requirement already satisfied: setuptools>=41.0.0 in /usr/local/lib/python3.7
/dist-packages (from tensorboard<1.15.0,>=1.14.0->tensorflow==1.14.0) (57.4.0)
Requirement already satisfied: importlib-metadata>=4.4 in
/usr/local/lib/python3.7/dist-packages (from
markdown>=2.6.8->tensorboard<1.15.0,>=1.14.0->tensorflow==1.14.0) (4.8.2)
Requirement already satisfied: typing-extensions>=3.6.4 in
/usr/local/lib/python3.7/dist-packages (from importlib-
metadata>=4.4->markdown>=2.6.8->tensorboard<1.15.0,>=1.14.0->tensorflow==1.14.0)
(3.10.0.2)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-
packages (from importlib-
metadata>=4.4->markdown>=2.6.8->tensorboard<1.15.0,>=1.14.0->tensorflow==1.14.0)
(3.6.0)
Requirement already satisfied: cached-property in /usr/local/lib/python3.7/dist-
packages (from h5py->keras-applications>=1.0.6->tensorflow==1.14.0) (1.5.2)
Requirement already satisfied: keras==2.2.4 in /usr/local/lib/python3.7/dist-
packages (2.2.4)
Requirement already satisfied: keras-preprocessing>=1.0.5 in
/usr/local/lib/python3.7/dist-packages (from keras==2.2.4) (1.1.2)
Requirement already satisfied: keras-applications>=1.0.6 in
/usr/local/lib/python3.7/dist-packages (from keras==2.2.4) (1.0.8)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.7/dist-packages
(from keras==2.2.4) (3.13)
Requirement already satisfied: h5py in /usr/local/lib/python3.7/dist-packages
(from keras==2.2.4) (3.1.0)
Requirement already satisfied: scipy>=0.14 in /usr/local/lib/python3.7/dist-
packages (from keras==2.2.4) (1.4.1)
Requirement already satisfied: numpy>=1.9.1 in /usr/local/lib/python3.7/dist-
packages (from keras==2.2.4) (1.19.5)
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.7/dist-
packages (from keras==2.2.4) (1.15.0)
Requirement already satisfied: cached-property in /usr/local/lib/python3.7/dist-
packages (from h5py->keras==2.2.4) (1.5.2)
```

The ViRL proposes 4 non-medical policy interventions for the agents that are entrusted to be controlling the spread of the COVID-19 virus. 0. no intervention (remove all restrictions) 1. impose a full lockdown 2. implement track & trace 3. enforce social distancing and face masks

After running each episode for 52 weeks, the agent makes a decision to take based on the evidence obtained from the state of the epidemic as the number of persons that are: 0. susceptibles 1. infectious 2. quarantined 3. recovereds

Every interference can affect the infection rate dissimilarly (the number of concurrently infected and hospitalized persons) and on the economic opportunity cost (summarized single scalar reward at each time step).

Relationship between the reward and the state of epidemic are hard coded into the simulator, using a bunch of parameters which make the simulation possible. The reward values are presented in negative float number, and the sum of all the rewards are then compared to check whether the outcome was optimal or not. The higher the reward vlaue, the better the outcome.

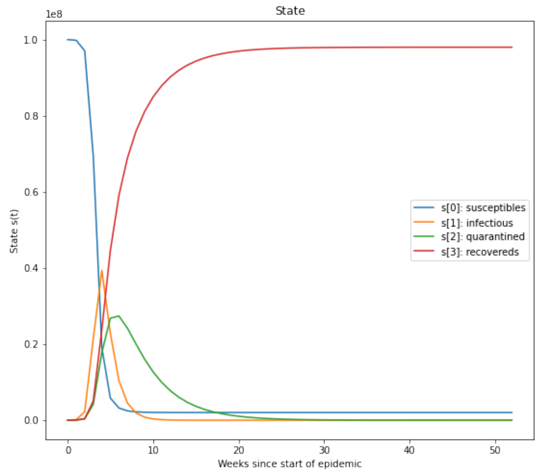
Below is an example of running an dummy agent on the ViRL environment. This agent will always take the same action every week, irrespectively of the current state of the population.

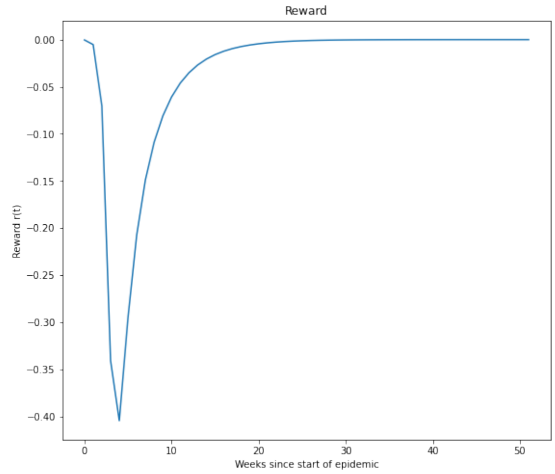
[ ]:
```
env = virl.Epidemic()

states = []
rewards = []
done = False

s = env.reset() # reset the environment before using it and log the starting␣
 ↪state
states.append(s)
while not done:
    s, r, done, i = env.step(action=0) # deterministic agent doing action 0␣
 ↪(no-intervention)
    states.append(s)
    rewards.append(r)
```

As a example, you can now plot the evolution of states and reward for the 52 weeks of this epidemic simulation.

[ ]:
```
# start a figure with 2 subplot
fig, axes = plt.subplots(1, 2, figsize=(20, 8))
labels = ['s[0]: susceptibles', 's[1]: infectious', 's[2]: quarantined', 's[3]:␣
 ↪recovereds']
states = np.array(states)

# plot state evolution on the left subplot
for i in range(4):
    axes[0].plot(states[:,i], label=labels[i]);
axes[0].set_title('State')
axes[0].set_xlabel('Weeks since start of epidemic')
axes[0].set_ylabel('State s(t)')
axes[0].legend()

# plot reward evolution on the right subplot
axes[1].plot(rewards);
axes[1].set_title('Reward')
axes[1].set_xlabel('Weeks since start of epidemic')
axes[1].set_ylabel('Reward r(t)')

print('Total reward for this episode is ', np.sum(rewards))
```

Total reward for this episode is -1.9231823993453754





It can be seen in the graoh how this policy creates an overflow of cases at the start of the epidemic as the reward value goes very low in the first weeks. Later, as the patients start to recover the graphs goes back up to low negative value.

2 1.3 PEAS Anlysis

The following presents the PEAS analysis for our problem

2.1 Performance Measure

- Rewards

Closer to the zero, the better the rewards

2.2 Environment

- ViRL

Epidemics Reinforcement Learning Environment ## Action

The Virtual environment ViRL takes these 4 actions and implements a policy:

- [0]no intervention (remove all restrictions)
- [1]impose a full lockdown
- [2]implement track & trace
- [3]enforce social distancing and face masks

2.3 Sensor

- [0]susceptibles
- [1]infectious
- [2]quarantined
- [3]recovereds

3 2. Method and Implementation

In this experiment we'll implement three different types of agents named Random, Determinsitc, and Q-Learning with Neural Netwrok Function Approximation where we'll compare their rewards and see how each agent performed in comparison to the other.

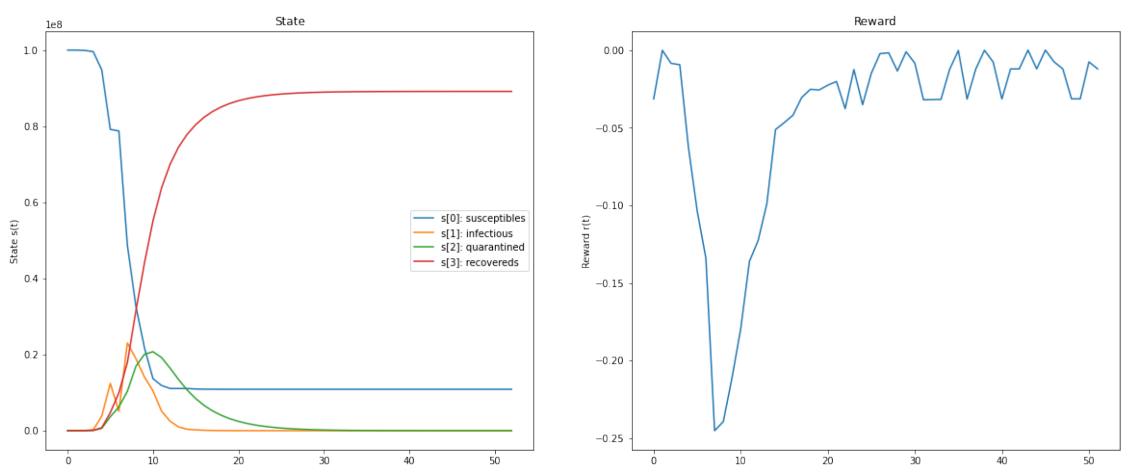
3.1 2.1 Random Agent

The Random agent simple takes a random action from env.action_space.n and returns it to the en.step(). Env.step() then performs action for the step passed by the random agent and appends to the state and rewards denoted by the action for the current week. This process is repeated in the same manner for the next 52 weeks.

[ ]:
```
# add code for your random agent

def random_agent(s):
  action = np.random.choice(env.action_space.n)
  return action

env = virl.Epidemic()

R_states = []
R_rewards = []
R_action = []
done = False

s = env.reset()
R_states.append(s)


while not done:
      action = random_agent(s)
      s, r, done, i = env.step(action=action)
      R_action.append(action)
      R_states.append(s)
      R_rewards.append(r)



print(sum(R_rewards))
```

```
fig, axes = plt.subplots(1, 2, figsize=(20, 8))
labels = ['s[0]: susceptibles', 's[1]: infectious', 's[2]: quarantined', 's[3]:␣
 ↪recovereds']
states = np.array(R_states)

# plot state evolution on the left subplot
for i in range(4):
    axes[0].plot(states[:,i], label=labels[i]);
axes[0].set_title('State')
axes[0].set_xlabel('Weeks since start of epidemic')
axes[0].set_ylabel('State s(t)')
axes[0].legend()

# plot reward evolution on the right subplot
axes[1].plot(R_rewards);
axes[1].set_title('Reward')
axes[1].set_xlabel('Weeks since start of epidemic')
axes[1].set_ylabel('Reward r(t)')

print('Total reward for this episode is ', np.sum(R_rewards))
```

```
-2.305388877560294
Total reward for this episode is  -2.3053888775602944
```



3.2 2.2 Deterministic Agent

Deterministic agent is an agent where the current state and the selected action of the agent can fully decide the next state of the environment. In deterministc, the agent doesn't need to worry about unpredictability.

Following is a code presented where a determministic agent is implemented where it access the 4 actions of the environment, compares them to a certain number of the pateints either suspected, infected, recovered or infectious and then determines a action which is suppose to be implemented.

In our case, the suspecticles greater than 5000000, infectious greater than 10000, quarantine greater than 300000 and recovered greater than 500000 then action "impose full lock down" would be implemeted. Otherwise track and trace down would be implemented.

As the results show, the deterministic agent performs better than the random agent as the rewards for the deterministic are lower than random agent.

[ ]:
```
def deterministic_agent(s):
  action = 0
  if s[0] > 5000000 and s[1] > 10000 and s[2] > 300000 and s[3] > 500000:
    action = 1
  else:
    action = 2
  return action
```

[ ]:
```
# test deterministic agent on default environment
env = virl.Epidemic()

D_states = []
D_rewards = []
D_action = []

done = False

s = env.reset()
D_states.append(s)
while not done:
    action = deterministic_agent(s)
    D_action.append(action)
    s, r, done, i = env.step(action=action)
    D_states.append(s)
    D_rewards.append(r)
print(sum(D_rewards))
```

```
-1.209125303870942
```

[ ]:
```
'''
# start a figure with 2 subplot
fig, axes = plt.subplots(1, 2, figsize=(20, 8))
labels = ['s[0]: susceptibles', 's[1]: infectious', 's[2]: quarantined', 's[3]:␣
↪recovereds']
states = np.array(D_states)
```

```
# plot state evolution on the left subplot
for i in range(4):
    axes[0].plot(states[:,i], label=labels[i]);
axes[0].set_title('State')
axes[0].set_xlabel('Weeks since start of epidemic')
axes[0].set_ylabel('State s(t)')
axes[0].legend()

# plot reward evolution on the right subplot
axes[1].plot(D_rewards);
axes[1].set_title('Reward')
axes[1].set_xlabel('Weeks since start of epidemic')
axes[1].set_ylabel('Reward r(t)')
print('Total reward for this episode is ', np.sum(D_rewards))
'''
```

[ ]:
```
"\n# start a figure with 2 subplot\nfig, axes = plt.subplots(1, 2, figsize=(20,
8))\nlabels = ['s[0]: susceptibles', 's[1]: infectious', 's[2]: quarantined',
's[3]: recovereds']\nstates = np.array(D_states)\n\n# plot state evolution on
the left subplot\nfor i in range(4):\n    axes[0].plot(states[:,i],
label=labels[i]);\naxes[0].set_title('State')\naxes[0].set_xlabel('Weeks since
start of epidemic')\naxes[0].set_ylabel('State s(t)')\naxes[0].legend()\n\n#
plot reward evolution on the right subplot\naxes[1].plot(D_rewards);\naxes[1].se
t_title('Reward')\naxes[1].set_xlabel('Weeks since start of
epidemic')\naxes[1].set_ylabel('Reward r(t)')\nprint('Total reward for this
episode is ', np.sum(D_rewards))\n"
```

3.3 2.3 Q-Learning with Function Approximation

Q-Learning is an reinforcement algorithm that looks to find the most suitable action for every given state. Q-learninng is said to be an off policy RL algorithm as it learns from actions which are not in the current policy. Basically, q-learnings looks forward to learn a policy whihch would maximzie the rewards [1].

As our third agent, is going to be a Q-Learning with Function Approximization which would use neural network. For each episode, the agent takes an action (either the maximum reawrd from the Q-Table or randomly), then observes the reward of the action chosen to update the temporal difference of Q-table using the formula (AIMA, page 844):

`(,)(,)+[()+  (,)(,)]`

Where:

- R(s) : reward
- Q(s,a): final action value after state s and action a
- s′ : next state
- s : current state

- a : current action (lockdown, restrictions etc)
- a' : next action
- y : discount factor from 0 to 1
- : learning effect from 0 to 1

  A major problem when naively applying neural network is that they tend to overfit quite badly (due to the flexibility) which is especially the case in RL where the observation are highly correlated (due to the sequential behavior and nature of the policy). In order to break this correlation (within a single update) we implement a so-called replay buffer which saves all observation for a certain period back in time. We then sample from this buffer when making our updates. Below we implement such a buffer.

  RL observations are highly correlated (because of the policy's nature and sequential beahviour). In order to interrupt the correlation , replay buffer is implemented which saves observations for a brief history in time. After that, we sample this buffer when making the updates.

  Finally our model is implement and our reinforcement learning agent that every week takes an action based on its experience from playing against the simulator for hundreds of training episodes. The Q-learning agent bascially follows a policy and update the function approximator.

  Later in your report,the agent is trained and reported how it learns and how it performs on the ViRL simulator.

4 NOTE:

The following code is taken Lab 8 solutions

[ ]:
```
class NNFunctionApproximatorJointKeras():
    """ A basic MLP neural network approximator and estimator using Keras
    """

    def __init__(self, alpha, d_states, n_actions, nn_config, verbose=False):
        self.alpha = alpha
        self.nn_config = nn_config      # determines the size of the hidden␣
↪layer (if any)
        self.n_actions = n_actions
        self.d_states = d_states
        self.verbose=verbose # Print debug information
        self.n_layers = len(nn_config)
        self.model = self._build_model()

    def _huber_loss(self,y_true, y_pred, clip_delta=1.0):
        """
        Huber loss (for use in Keras), see https://en.wikipedia.org/wiki/
↪Huber_loss
        The huber loss tends to provide more robust learning in RL settings␣
↪where there are
```

```
        often "outliers" before the functions has converged.

        Note: There i sa huber loss which is likely quicker, but we want to␣
↪show you the core implementation here
        """
        error = y_true - y_pred
        cond  = K.abs(error) <= clip_delta
        squared_loss = 0.5 * K.square(error)
        quadratic_loss = 0.5 * K.square(clip_delta) + clip_delta * (K.
↪abs(error) - clip_delta)
        return K.mean(tf.where(cond, squared_loss, quadratic_loss))

    def _build_model(self):
        # Neural Net for Deep-Q learning
        model = Sequential()
        for ilayer in self.nn_config:
            model.add(Dense(ilayer, input_dim=self.d_states, activation='relu'))
        model.add(Dense(self.n_actions, activation='linear'))
        model.compile(loss=self._huber_loss, # define a special loss function
                      optimizer=Adam(lr=self.alpha, clipnorm=10.)) # specify␣
↪the optimiser, we clip the gradient of the norm which can make traning more␣
↪robust
        return model

    def predict(self, s, a=None):
        if a==None:
            return self._predict_nn(s)
        else:
            return self._predict_nn(s)[a]

    def _predict_nn(self,state_hat):
        """
        Predict the output of the neural netwwork (note: these can be vectors)
        """
        x = self.model.predict(state_hat)
        return x

    def update(self, states, td_target):
        self.model.fit(states, td_target, epochs=1, verbose=0) # take one␣
↪gradient step usign the optimiser
        return
```

[ ]:
```
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state',␣
↪'reward','is_not_terminal_state'))

class ReplayMemory():
```

```python
    """
    Implement a replay buffer using the deque collection
    """

    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = deque(maxlen=capacity)

    def push(self, *args):
        """Saves a transition."""
        self.memory.append(Transition(*args))

    def pop(self):
        return self.memoery.pop()

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

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

[ ]:
```python
# Note: This a an inline implementation for teaching purposes (you may want
# to split you own "production" code into more smaller parts" and optimise the␣
↪performance!)

# Keep track of some stats
EpisodeStats = namedtuple("Stats",["episode_lengths", "episode_rewards"])

# Main Q-learner
def q_learning_nn(env, func_approximator, func_approximator_target,␣
↪num_episodes,max_steps_per_episode=500,discount_factor=0.95, epsilon_init=0.
↪01, epsilon_decay=0.99995,epsilon_min=0.01,use_batch_updates=True,␣
↪show=False, fn_model_in=None, fn_model_out=None):
    """
    Q-Learning algorithm for Q-learning using Function Approximations.
    Finds the optimal greedy policy while following an explorative greedy␣
↪policy.

    Args:
        env: OpenAI environment.
        func_approximator: Action-Value function estimator, behavior policy (i.
↪e. the function which determines the next action)
        func_approximator_target: Action-Value function estimator, updated less␣
↪frequenty than the behavior policy
        num_episodes: Number of episodes to run for.
        max_steps_per_episode: Max number of steps per episodes
        discount_factor: Gamma discount factor.
```

```
        epsilon_init: Exploration strategy; chance the sample a random action.␣
↪Float between 0 and 1.
        epsilon_decay: Each episode, epsilon is decayed by this factor
        epislon_min: Min epsilon value
        use_batch_updates=True,
        show: Render the environment (mainly for test/demo)
        fn_model_in: Load the model from the file if not None
        fn_model_out: File name of the saved model, saves the best model in the␣
↪last 100 episodes

    Returns:
        An EpisodeStats object with two numpy arrays for episode_lengths and␣
↪episode_rewards.
    """

    memory = ReplayMemory(BUFFER_SIZE) # init the replay memory
    n_actions = env.action_space.n
    d_states  = env.observation_space.shape[0]
    best_reward = 0
    Q_action = []

    # Synch the target and behavior network
    if not fn_model_in is None:
        func_approximator.model.load_weights(fn_model_in)
    func_approximator_target.model.set_weights(func_approximator.model.
↪get_weights())

    # Keeps track of useful statistics
    stats = EpisodeStats(
        episode_lengths=np.zeros(num_episodes),
        episode_rewards=np.zeros(num_episodes))

    epsilon = epsilon_init
    states = np.zeros(shape=(2500,4)) # Should match the number of episodes
    for i_episode in range(num_episodes):
        sys.stdout.flush()
        # Reset the environment and pick the first action
        state = env.reset()
        state = np.reshape(state, [1, d_states]) # reshape to the a 1xd_state␣
↪numpy array

        # One step in the environment
        for t in range(max_steps_per_episode):#itertools.count():
             #
            if(show):
                env.render()
```

```
            # Select an action usign and epsilon greedy policy based on the␣
↪main behavior network
            if np.random.rand() <= epsilon:
                action = random.randrange(n_actions)
            else:
                act_values = func_approximator.predict(state)[0]
                action = np.argmax(act_values)  # returns action                    ␣
↪

            # Take a step
            next_state, reward, done, _ = env.step(action)

            next_state = np.reshape(next_state, [1, d_states] )

            # Add observation to the replay buffer
            if done:
                memory.push(state, action, next_state, reward, 0.0)
            else:
                memory.push(state, action, next_state, reward, 1.0)

            # Update statistics
            stats.episode_rewards[i_episode] += reward
            stats.episode_lengths[i_episode] = t

            #print(done)
            #print(len(memory))
            #print(func_approximator.alpha)
            #print()
            # Update network (if learning is on, i.e. alpha>0 and we have␣
↪enough samples in memory)
            if func_approximator.alpha > 0.0 and len(memory) >= BATCH_SIZE:

                # Fetch a bacth from the replay buffer and extract as numpy␣
↪arrays
                transitions = memory.sample(BATCH_SIZE)
                batch = Transition(*zip(*transitions))
                train_rewards = np.array(batch.reward)
                train_states = np.array(batch.state)
                train_next_state = np.array(batch.next_state)
                train_actions = np.array(batch.action)
                train_is_not_terminal_state = np.array(batch.
↪is_not_terminal_state) #

                if(use_batch_updates):
                    # Do a single gradient step computed based on the full␣
↪batch
```

```
                    train_td_targets    = func_approximator.
↪predict(train_states.reshape(BATCH_SIZE,4)) # predict current values for the␣
↪given states
                    q_values_next       = func_approximator_target.predict(np.
↪array(batch.next_state).reshape(BATCH_SIZE,d_states))
                    train_td_targetstmp = train_rewards + discount_factor *␣
↪train_is_not_terminal_state * np.amax(q_values_next,axis=1)
                    train_td_targets[ (np.arange(BATCH_SIZE), train_actions.
↪reshape(BATCH_SIZE,).astype(int))] = train_td_targetstmp
                    func_approximator.update(train_states.
↪reshape(BATCH_SIZE,d_states), train_td_targets) # Update the function␣
↪approximator using our target
                else:
                    # Do update in a truely online sense where a gradient step␣
↪is performaed per observation
                    for s in range(train_rewards.shape[0]):
                        target = func_approximator.predict(train_states[s])[0]
                        q_next = func_approximator_target.
↪predict(train_next_state[s])[0]
                        target[train_actions[s]] = train_rewards[s] +␣
↪discount_factor * train_is_not_terminal_state[s] * np.amax(q_next)
                        func_approximator.update(train_states[s], target.
↪reshape(1,n_actions)) # Update the function approximator using our target   ␣
↪
                if epsilon > epsilon_min:

                    epsilon *= epsilon_decay

            state = next_state
            states[i_episode] = state
            Q_action.append(action)
            '''
            # Synch the target and behavior network
            func_approximator_target.model.set_weights(func_approximator.model.
↪get_weights())

            print("\repisode: {}/{}, score: {}, epsilon: {:.2}".
↪format(i_episode, num_episodes, reward, epsilon), end="")                  ␣
↪

            # Save the best model so far
            if fn_model_out is not None and (t >= best_reward):
                func_approximator.model.save_weights(fn_model_out)
                best_reward = t
            '''
```

```
            if done:
                # Synch the target and behavior network
                func_approximator_target.model.set_weights(func_approximator.
↪model.get_weights())

                print("\repisode: {}/{}, score: {}, epsilon: {:.2}".
↪format(i_episode, num_episodes, t, epsilon), end="")

                # Save the best model so far
                if fn_model_out is not None and (t >= best_reward):
                    func_approximator.model.save_weights(fn_model_out)
                    best_reward = t

                break
    return stats,states,Q_action
```

The learning rate, which is the leap the code takes to find out the optimal policy, is chosen to low (0.000001) and the discount factor equal to 0.95 whihch helps in producing high rewards for our Q-learning model. The batch size is chosen to be 64 and initial epsilon value is 0.1 and can go to a max low of 0.01 with a decay rate of 0.9995. The epsilon value dictates the randomness is finding specific actions base on the Q values we already have. The max_iter per epsisode is 52 which makes sure our env.action() returns a true in our done varaible (episode reaching to it's conclusion).

Along with the other specified parameters, our model is trained below and produces a decent reward along with an increasing and converging graph. The agent is giving instances of learning as it is trying to learn but as of now I couldn't produce a perfect learning model. It requires the model roughly around 1500 episodes before it reaches convergence. It takes several runs for the model to finally drop onto a model which converges and for the one which I have saved is the best one one in my recent tries. It takes roughly 22 mins for the model to converge and on average the model is performing better than random agent but worse than deterministic agent.

As per to encapsulate the reward value, I selected the last value in our list of rewards value beacuse once the graph converges, the last value reflect the approximate reward value which shows convergence. So from here all the reward values being compared and analysed for Q-Learning are basically the last values as depicted from the graph.

[ ]:
```
env = virl.Epidemic()
d_states    = env.observation_space.shape[0]
n_actions   = env.action_space.n

alpha= 0.000001
nn_config   = [64,64]
BATCH_SIZE  = 64
BUFFER_SIZE = 10000
```