PredictionUsingDQN

October 17, 2020

1 Stock price prediction using Deep Reinforcement Learning

[31]:

```
# importing the dependencies
import numpy as np # linear algebra
import pandas as pd # for the dataframe

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf # for Deep learning
from glob import glob # for file handling
from tqdm import tqdm # for the progress bar
from collections import deque # for simpler implementation of memory
import os
import matplotlib.pyplot as plt

%matplotlib inline
tf.logging.set_verbosity(tf.logging.ERROR)
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)


from tensorflow import keras
from tensorflow.keras import layers
if type(tf.contrib) != type(tf): tf.contrib._warning = None

import warnings
warnings.filterwarnings("ignore")
```

[32]:

```
# For GPU
config = tf.ConfigProto(log_device_placement=True)
config.graph_options.optimizer_options.global_jit_level = tf.OptimizerOptions.
 ↪ON_1
tf.config.optimizer.set_jit(True)
tf.config.threading.set_intra_op_parallelism_threads(8)  # Number of physical␣
 ↪cores.
```

Load the stock prices and set required parameters

[33]:

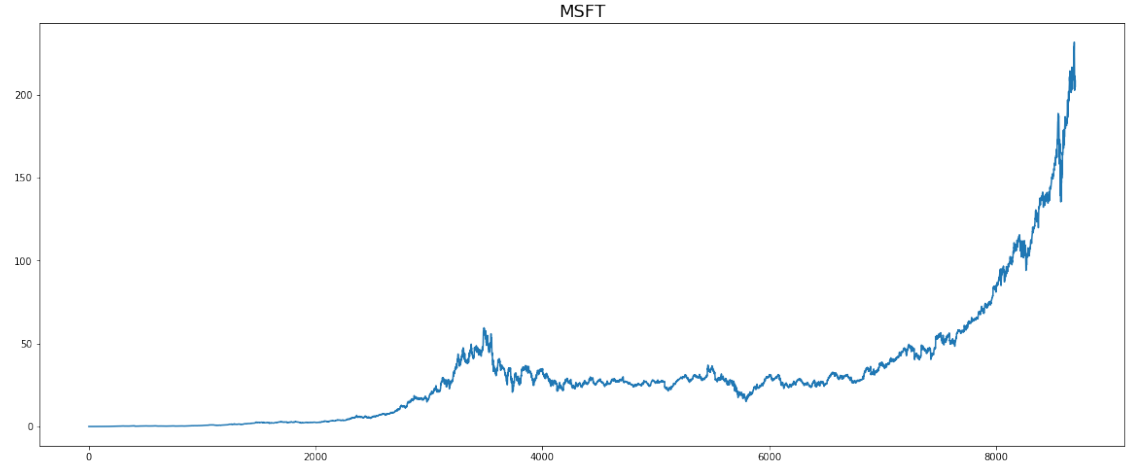
```
# load the csv file

path_folder = './stock_data/MSFT.csv'


stock_name  = 'MSFT'

data = pd.read_csv(path_folder)

# constants
LOG = False
episode_count = 15
window_size = 100
data = data['Close'].values # what our data is
len_data = len(data) - 1 # total information length
batch_size = 10 # minibatch size

# logs
loss_global = []
profits_global = []
```

[34]:

```
plt.figure(figsize = (20, 8))
plt.plot(data)
plt.title(stock_name  , fontsize = 18)
```

[34]:

```
Text(0.5, 1.0, 'MSFT')
```



2 Deep Q-Learning algorithm

[35]:

```
class StocksDQN():
    # init functions
    def __init__(self, input_dim, scope, is_eval = False, epsilon_decay_steps =␣
↪1000):
        # input_dim: state size
        # is_eval: is being evaluated
        # scope: scope of the model
        self.state_size = input_dim
        self.action_space = 3 # sell, sit, buy
        self.memory = deque(maxlen = 10000)
        self.inventory = [] # holdings that we have
        self.scope = scope # name of scope
        self.is_eval = is_eval # whether in training or deployment
        self.gamma = 0.99 # discount factor
        self.h1_size = 64
        self.h2_size = 32
        self.h3_size = 8
        # epsilon greedy policy
        self.epsilon = 1.0
        self.epsilon_end = 0.01
        self.epsilon_decay_val = (self.epsilon - self.epsilon_end)/
↪epsilon_decay_steps
        self._build_model()
        self.initialize_network()

    def _build_model(self):

        self.input_placeholder = tf.placeholder(tf.float32, [None, self.
↪state_size], name = 'inputs')
        self.target_placeholder = tf.placeholder(tf.float32, [None, self.
↪action_space], name = 'target_value')

        # layers
        h1 = tf.contrib.layers.fully_connected(self.input_placeholder, self.
↪h1_size)
        h2 = tf.contrib.layers.fully_connected(h1, self.h2_size)
        h3 = tf.contrib.layers.fully_connected(h2, self.h3_size)
        self.action_pred = tf.contrib.layers.fully_connected(h3, self.
↪action_space,activation_fn = tf.nn.softmax)
        self.loss = tf.reduce_mean(tf.square(self.target_placeholder - self.
↪action_pred))
        self.update_step = tf.train.AdamOptimizer(0.001).minimize(self.loss)
    # Initialize the network
    def initialize_network(self):
        self.sess = tf.Session(config=config)
        self.sess.run(tf.global_variables_initializer())

    # Functions for taking actions using epsilon greedy policy
    def act(self, state):
        if not self.is_eval and np.random.random() <= self.epsilon:
            return np.random.randint(self.action_space)

        # else use the model to predict action
        action_dist = self.sess.run(self.action_pred, feed_dict = {self.
↪input_placeholder: state})
        return np.argmax(action_dist[0])

    def experience_replay(self, batch_size):
        mini_batch = []
        mem_len = len(self.memory)
        for i in range(mem_len - batch_size + 1, mem_len):
            mini_batch.append(self.memory[i])

        loss_log = []

        for state, action, reward, next_state, done in mini_batch:
            target_s = reward
            if not done:
                # get predictions from model
                pred = self.sess.run(self.action_pred, feed_dict = {self.
↪input_placeholder: next_state})

                # get the target value to be fit
                target_s = reward + self.gamma*np.amax(pred[0])

            # target value to be fit upon
            target_y = self.sess.run(self.action_pred, feed_dict = {self.
↪input_placeholder: state})
            target_y[0][action] = target_s

            # train the model
            feed_dict = {self.input_placeholder: state, self.target_placeholder:
↪ target_y}
            loss, _ = self.sess.run([self.loss, self.update_step], feed_dict =␣
↪feed_dict)

            # add to logs
            loss_log.append(loss)
        # reduce the value of epsilon
        if self.epsilon > self.epsilon_end:
            self.epsilon -= self.epsilon_decay_val

        # return loss
        return loss_log
```

2.0.1 To get price of profit, stock prices according to window size

Below is the code for some helper functions

[36]:

```
# function to properly return the string of price
def format_price(price):
    return ("-$" if price < 0 else "$") + "{0:.2f}".format(abs(price))

def sigmoid(x):
    return 1/(1 + np.exp(-x))

# function to get the state
def get_state(data, t, n):
    d = t - n + 1
    if d >= 0:
        block = data[d:t+1]
    else:
        # pad with t0
        block = -d*[data[0]] + data[0:t+1].tolist()

    # get results
    res = []
    for i in range(n - 1):
        res.append(sigmoid(block[i + 1] - block[i]))

    # return numpy array
    return np.array([res])
```

[37]:

```
# define the agent
agent = StocksDQN(window_size, 'model_pre')
```

```
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efdf2ad3b10>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efdf2ad3b10>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efdf29b0650>> could not be transformed and will be executed as-is.
```

```
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efdf29b0650>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efe03652050>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efe03652050>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efdf2981810>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efdf2981810>>: AssertionError:
Bad argument number for Name: 3, expecting 4
```

[38]:

```python
# run things
for e in range(episode_count):
    state = get_state(data, 0, window_size + 1)

    # init values for new episode
    total_profit = 0.0 # total profit in this epoch
    agent.inventory = [] # reset the inventory
    total_loss = [] # at each step what was the total loss
    mean_loss = [] # at each step what was the mean loss

    for t in tqdm(range(len_data)):
        action = agent.act(state)

        # next state
        next_state = get_state(data, t + 1, window_size + 1)
        reward = 0

        # now go according to actions
        if action == 2:
            # buy
            agent.inventory.append(data[t])
            if LOG:
                print('Buy:' + format_price(data[t]))


        elif action == 0 and len(agent.inventory) > 0:
            bought_price = agent.inventory.pop(0) # remove the first element␣
↪and return the value
            profit = data[t] - bought_price # profit this iteration
            reward = max(data[t] - bought_price, 0) # reward
            total_profit += profit # add to total profit
            if LOG:
                print("Sell: " + format_price(data[t]) + " | Profit: " +␣
↪format_price(profit))

        # condition for finish
        done = t == len_data - 1
        agent.memory.append((state, action, reward, next_state, done))
        state = next_state

        if done and LOG:
            print("Total Profit: " + format_price(total_profit))

        # train the model
        if len(agent.memory) > batch_size:
            losses = agent.experience_replay(batch_size)
            total_loss.append(np.sum(losses))
            mean_loss.append(np.mean(losses))

    # add the mean loss to global loss
    loss_global.append(np.mean(mean_loss))
    profits_global.append(total_profit)

    print('[*]Episode: {0}, loss: {1}, profits: {2}'.format(e, loss_global[-1],␣
↪profits_global[-1]))
```

```
100%|      | 8699/8699 [01:14<00:00, 116.02it/s]
  0%|          | 11/8699 [00:00<01:21, 106.66it/s][*]Episode: 0, loss:
0.012476674281060696, profits: 84.12195700000007
100%|      | 8699/8699 [01:15<00:00, 115.00it/s]
  0%|          | 12/8699 [00:00<01:16, 112.83it/s][*]Episode: 1, loss:
0.0044120922684669495, profits: -30.62554799999996
100%|      | 8699/8699 [01:18<00:00, 110.18it/s]
  0%|          | 13/8699 [00:00<01:07, 129.00it/s][*]Episode: 2, loss:
0.007566284388303757, profits: 29.229172999999985
100%|      | 8699/8699 [01:24<00:00, 103.40it/s]
  0%|          | 12/8699 [00:00<01:13, 118.78it/s][*]Episode: 3, loss:
0.007851461879909039, profits: -77.77503600000013
100%|      | 8699/8699 [01:21<00:00, 106.75it/s]
  0%|          | 11/8699 [00:00<01:22, 105.85it/s][*]Episode: 4, loss:
0.006167199462652206, profits: -13.571717999999947
100%|      | 8699/8699 [01:21<00:00, 106.46it/s]
  0%|          | 13/8699 [00:00<01:07, 128.85it/s][*]Episode: 5, loss:
```

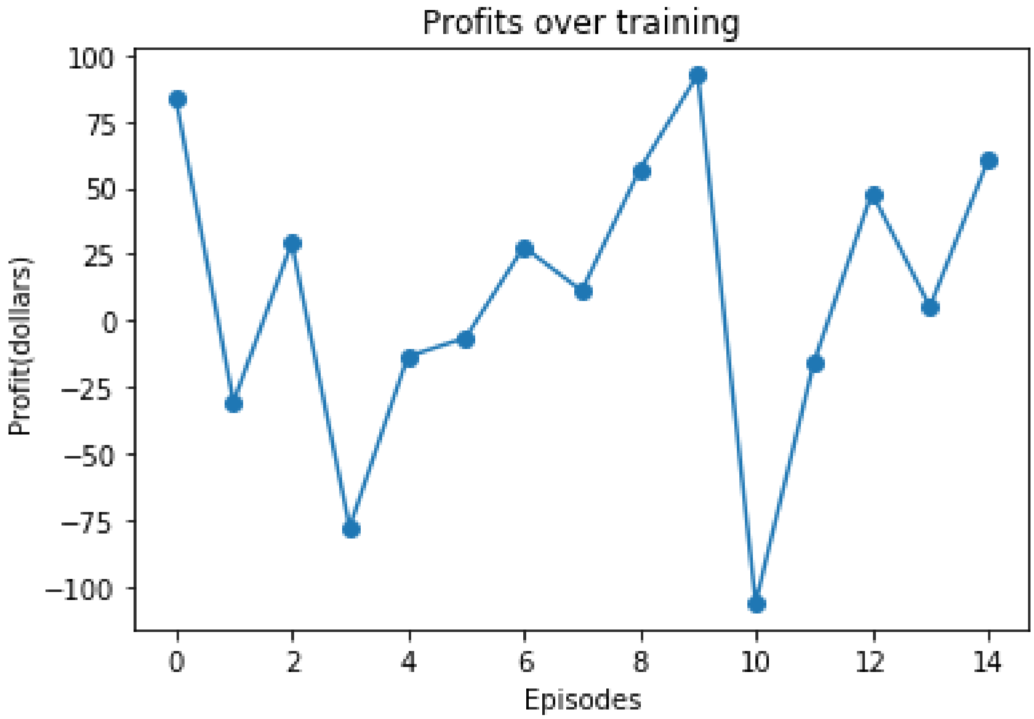
```
0.01807945780456066, profits: -6.4667889999999275
100%|      | 8699/8699 [01:26<00:00, 100.39it/s]
  0%|           | 12/8699 [00:00<01:15, 115.68it/s][*]Episode: 6, loss:
0.0065415995195508, profits: 27.763485999999986
100%|      | 8699/8699 [01:21<00:00, 107.34it/s]
  0%|           | 13/8699 [00:00<01:07, 128.39it/s][*]Episode: 7, loss:
0.004649381153285503, profits: 11.239652999999965
100%|      | 8699/8699 [01:20<00:00, 108.14it/s]
  0%|           | 12/8699 [00:00<01:12, 119.79it/s][*]Episode: 8, loss:
0.01602264493703842, profits: 56.88193500000013
100%|      | 8699/8699 [01:27<00:00, 99.97it/s]
  0%|           | 10/8699 [00:00<01:30, 95.96it/s][*]Episode: 9, loss:
0.014853036031126976, profits: 92.60075600000003
100%|      | 8699/8699 [01:21<00:00, 106.67it/s]
  0%|           | 14/8699 [00:00<01:06, 131.52it/s][*]Episode: 10, loss:
0.004825407639145851, profits: -106.15372900000008
100%|      | 8699/8699 [01:20<00:00, 107.87it/s]
  0%|           | 13/8699 [00:00<01:11, 121.37it/s][*]Episode: 11, loss:
0.014983797445893288, profits: -15.777920000000023
100%|      | 8699/8699 [01:23<00:00, 104.01it/s]
  0%|           | 12/8699 [00:00<01:19, 109.43it/s][*]Episode: 12, loss:
0.005404008086770773, profits: 48.10088600000002
100%|      | 8699/8699 [01:24<00:00, 102.39it/s]
  0%|           | 13/8699 [00:00<01:06, 129.87it/s][*]Episode: 13, loss:
0.003640985582023859, profits: 5.198355000000056
100%|      | 8699/8699 [01:23<00:00, 104.69it/s][*]Episode: 14, loss:
0.007000233046710491, profits: 60.271384000000005
```

[39]:

```
plt.title('Profits over training')
plt.plot(profits_global, '-o')
plt.xlabel('Episodes')
plt.ylabel('Profit(dollars)')
```

[39]:

```
Text(0, 0.5, 'Profit(dollars)')
```



2.1 Double Q learning (with target network)

[40]:

```
# functions to copy parameters between two networks
def copy_parameters(q_network, target_network, sess):
    # q_network (source) to target_network (target)
    # sess: tensorflow session

    # source
    source_params = [t for t in tf.trainable_variables() if t.name.
↪startswith(q_network.scope)]
    source_params = sorted(source_params, key = lambda v: v.name)

    # target
    target_params = [t for t in tf.trainable_variables() if t.name.
↪startswith(target_network.scope)]
    target_params = sorted(target_params, key = lambda v: v.name)

    # do assign operations in loop
    for s_v, t_v in zip(source_params, target_params):
        op = t_v.assign(s_v)
        sess.run(op)
```

[41]:

```
class DoubleDQN():
    # initialization function
    def __init__(self, input_dim, scope, is_eval = False):
        self.state_size = input_dim
        self.action_space = 3 # sell, sit, buy
        self.scope = scope # name of scope
        self.h1_size = 64
        self.h2_size = 32
        self.h3_size = 8
        self._build_model()


    def _build_model(self):

        self.input_placeholder = tf.placeholder(tf.float32, [None, self.state_size], name = 'inputs')
        self.q_placeholder = tf.placeholder(tf.float32, [None, self.action_space], name = 'q_value')
        # layers
        h1 = tf.contrib.layers.fully_connected(self.input_placeholder, self.h1_size)
        h2 = tf.contrib.layers.fully_connected(h1, self.h2_size)
        h3 = tf.contrib.layers.fully_connected(h2, self.h3_size)
        self.action_pred = tf.contrib.layers.fully_connected(h3, self.action_space, activation_fn = tf.nn.softmax)
        self.loss = tf.reduce_mean(tf.square(self.q_placeholder - self.action_pred))
        self.update_step = tf.train.AdamOptimizer(0.001).minimize(self.loss)
```

[42]:

```
def train_dqn(q_network,
              target_network,
              sess,
              data,
              max_mem_size = 750,
              num_episodes = 15,
              train_target_every = 10,
              gamma = 0.99,
              epsilon_start = 0.99,
              epsilon_end = 0.001,
              epsilon_decay = 0.995):
    # function variables
    train_global_step = 0 # global step needed in parameter update
    train_loss = [] # training loss in each episode
    train_profits = [] # for profits in each episode
```

```python
    # memory_buffer
    memory_buffer = []

    # initialize variables
    sess.run(tf.global_variables_initializer())

    # init stuff
    epsilon = epsilon_start

    # iterate over each episode
    for ep in range(num_episodes):
        # for each training episode
        state = get_state(data, 0, window_size + 1)

        # init values for new episode
        total_profit = 0.0 # total profit in this episode
        # q_network.inventory = [] # holdings by q_network
        inventory = [] # inventory for this episode
        ep_loss = [] # total loss in this episode

        for t in tqdm(range(len_data)):
            # take action according to epsilon greedy policy
            if np.random.random() > epsilon:
                action = np.random.randint(q_network.action_space)
            else:
                feed_dict = {q_network.input_placeholder: state}
                action = sess.run(q_network.action_pred, feed_dict = feed_dict)
                action = np.argmax(action[0])

            # next state
            next_state = get_state(data, t + 1, window_size + 1)
            reward = 0

            # now go according to the actions
            if action == 2:
                # buy
                inventory.append(data[t])
                if LOG:
                    print('Buy:' + format_price(data[t]))

            elif action == 0 and len(inventory) > 0:
                bought_price = inventory.pop(0) # remove the first element and return the value
                profit = data[t] - bought_price # profit this transaction
                reward = max(data[t] - bought_price, 0) # reward
                total_profit += profit # add to total profit
                if LOG:
                    print("Sell: " + format_price(data[t]) + " | Profit: " +␣
↪format_price(profit))

            # condition for done
            done = t == len_data - 1

            # add to memory and make sure it's of fixed size
            memory_buffer.append((state, action, reward, next_state, done))
            if len(memory_buffer) > max_mem_size:
                memory_buffer.pop(0)

            # update state
            state = next_state

            # train the model
            if len(memory_buffer) > batch_size:

                # sample minibatches here
                mini_batch = memory_buffer[-batch_size:]

                # calculate q_value and target_values
                for state_t, action_t, reward_t, next_state_t, done_t in␣
↪mini_batch:
                    # condition for calculating y_j
                    if done_t:
                        target_pred = reward

                    else:
                        feed_dict = {target_network.input_placeholder:␣
↪next_state_t}
                        target_pred = sess.run(target_network.action_pred,␣
↪feed_dict = feed_dict)
                        target_value = reward_t + gamma*np.amax(target_pred[0])

                    # q_value
                    feed_dict = {q_network.input_placeholder: state_t}
                    q_values = sess.run(q_network.action_pred, feed_dict =␣
↪feed_dict)
                    q_values[0][action_t] = target_value

                    # drop epsilon value after every action taken
                    if epsilon > epsilon_end:
                        epsilon *= epsilon_decay

                    # update the q_network parameters
                    feed_dict = {q_network.input_placeholder: state_t,
                                 q_network.q_placeholder: q_values}
                    loss, _ = sess.run([q_network.loss, q_network.update_step],␣
↪feed_dict = feed_dict)

                    # update the lists
                    ep_loss.append(loss)

                    # update target network
                    train_global_step += 1
                    if ep % train_target_every == 0:
                        copy_parameters(q_network, target_network, sess)

        # update the outer values
        train_loss.append(ep_loss)
        train_profits.append(total_profit)

        # print val
        print('[*]Episode: {0}, loss: {1}, profits: {2}, epsilon: {3}'\
              .format(ep + 1, np.mean(train_loss[-1]), train_profits[-1],␣
↪epsilon))

    # return the values
    return train_loss, train_profits
```

[43]:
```
# run the model
q_network = DoubleDQN(window_size, 'q_network')
target_network = DoubleDQN( window_size, 'target_network')
sess = tf.Session(config=config)
loss, profits = train_dqn(q_network, target_network, sess, data)
```

```
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efebc51cb10>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efebc51cb10>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efebc57f0d0>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efebc57f0d0>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efebc51cb10>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
```

```
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efebc51cb10>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efebc51cdd0>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efebc51cdd0>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efebc51cb10>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efebc51cb10>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efebc598f50>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efebc598f50>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efdf428b350>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efdf428b350>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efdf6511690>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efdf6511690>>: AssertionError:
Bad argument number for Name: 3, expecting 4
100%|     | 8699/8699 [01:42<00:00, 85.06it/s]
  0%|          | 11/8699 [00:00<01:21, 106.61it/s][*]Episode: 1, loss:
1.4275182485580444, profits: 2993.3464360000007, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:27<00:00, 99.53it/s]
  0%|          | 11/8699 [00:00<01:26, 100.99it/s][*]Episode: 2, loss:
9.927663803100586, profits: 9750.301731999996, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:33<00:00, 92.78it/s]
  0%|          | 14/8699 [00:00<01:06, 129.72it/s][*]Episode: 3, loss:
7.074610710144043, profits: 8723.679133999996, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:28<00:00, 97.78it/s]
```

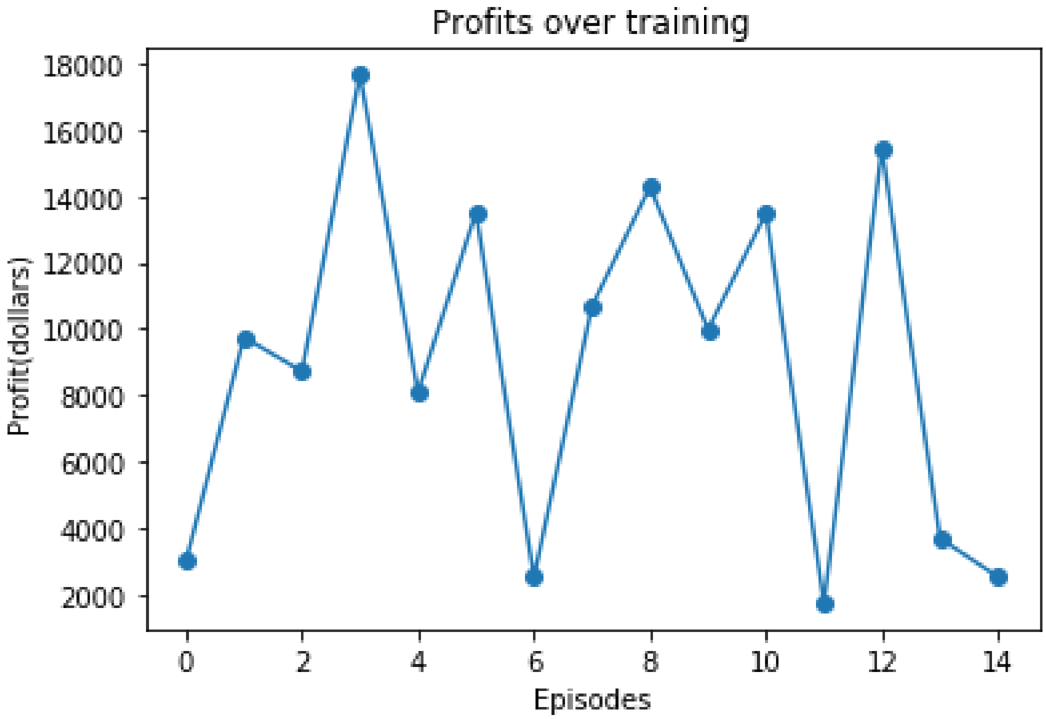
```
  0%|          | 12/8699 [00:00<01:13, 118.14it/s][*]Episode: 4, loss:
24.565065383911133, profits: 17672.34877999999, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:25<00:00, 101.51it/s]
  0%|          | 12/8699 [00:00<01:13, 118.63it/s][*]Episode: 5, loss:
10.313729286193848, profits: 8092.977806, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:15<00:00, 114.86it/s]
  0%|          | 12/8699 [00:00<01:13, 118.36it/s][*]Episode: 6, loss:
19.200298309326172, profits: 13522.266027000001, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:35<00:00, 91.52it/s]
  0%|          | 9/8699 [00:00<01:39, 86.99it/s][*]Episode: 7, loss:
1.6806341409683228, profits: 2515.1280350000006, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:45<00:00, 82.45it/s]
  0%|          | 12/8699 [00:00<01:16, 113.20it/s][*]Episode: 8, loss:
11.728540420532227, profits: 10682.958250000007, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:27<00:00, 99.33it/s]
  0%|          | 7/8699 [00:00<02:12, 65.47it/s][*]Episode: 9, loss:
17.8087100982666, profits: 14287.960070000005, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:37<00:00, 89.02it/s]
  0%|          | 10/8699 [00:00<01:34, 91.91it/s][*]Episode: 10, loss:
10.17485237121582, profits: 9984.962120999999, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:54<00:00, 76.17it/s]
  0%|          | 11/8699 [00:00<01:24, 102.80it/s][*]Episode: 11, loss:
19.195152282714844, profits: 13496.860477999995, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:42<00:00, 84.52it/s]
  0%|          | 9/8699 [00:00<01:41, 85.40it/s][*]Episode: 12, loss:
1.3077878952026367, profits: 1747.857188999999, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:39<00:00, 87.79it/s]
  0%|          | 12/8699 [00:00<01:15, 114.53it/s][*]Episode: 13, loss:
26.976255416870117, profits: 15438.182648000004, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:35<00:00, 91.41it/s]
  0%|          | 9/8699 [00:00<01:42, 84.81it/s][*]Episode: 14, loss:
2.6746416091918945, profits: 3683.9557210000003, epsilon: 0.0009954452565571535
100%|     | 8699/8699 [01:39<00:00, 87.76it/s]
[*]Episode: 15, loss: 1.5041004419326782, profits: 2519.489125999999, epsilon:
0.0009954452565571535
```

[44]:

```
plt.title('Profits over training')
plt.plot(profits, '-o')
plt.xlabel('Episodes')
plt.ylabel('Profit(dollars)')
```

[44]:

```
Text(0, 0.5, 'Profit(dollars)')
```



```
[ ]:
```

2.2 Dueling DQN

```
[45]: class DDQN():
          def __init__(self, input_dim, scope):
              self.state_size = input_dim
              self.scope = scope
              self.action_space = 3

              # placeholders
              self.input_placeholder = tf.placeholder(tf.float32, [None, self.
       ↪state_size], name = 'inputs')
              self.q_placeholder = tf.placeholder(tf.float32, [None, self.
       ↪action_space], name = 'q_value')

              # build model
              self._build_model()
              self._build_loss()

          def _build_model(self):
              # layers
```

```
        h1 = tf.contrib.layers.fully_connected(self.input_placeholder, 64)
        common_h2 = tf.contrib.layers.fully_connected(h1, 32)

        # value network layers
        val_h3 = tf.contrib.layers.fully_connected(common_h2, 8)
        self.value = tf.contrib.layers.fully_connected(val_h3, 1)

        # advantage network layers
        adv_h3 = tf.contrib.layers.fully_connected(common_h2, 16)
        self.advantage = tf.contrib.layers.fully_connected(adv_h3, self.
↪action_space)

        # get the final q value
        # tensorflow automatically perform the calculation of type [1,1] +␣
↪[1,3] = [1,3]
        # Q(s,a) = V(s) + (A(s,a) - 1/|A|(sum(A(s,a))))
        self.action_pred = self.value + (self.advantage - tf.reduce_mean(self.
↪advantage, axis = 1, keepdims = True))

    def _build_loss(self):
        self.loss = tf.reduce_mean(tf.square(self.action_pred - self.
↪q_placeholder))
        self.train_step = tf.train.AdamOptimizer().minimize(self.loss)
```

[47]:

```
def train_ddqn(network,
               sess,
               data,
               max_mem_size = 1000,
               num_episodes = 15,
               gamma = 0.99,
               epsilon_start = 0.99,
               epsilon_end = 0.001,
               epsilon_decay = 0.995):
    # function variables
    train_global_step = 0 # global step needed in parameter update
    train_loss = [] # training loss in each episode
    train_profits = [] # for profits in each episode

    # memory_buffer
    memory_buffer = []

    # initialize variables
    sess.run(tf.global_variables_initializer())

    # init stuff
    epsilon = epsilon_start
    # iterate over each episode
    for ep in range(num_episodes):
        # for each training episode
        state = get_state(data, 0, window_size + 1)

        # init values for new episode
        total_profit = 0.0 # total profit in this episode
        # q_network.inventory = [] # holdings by q_network
        inventory = [] # inventory for this episode
        ep_loss = [] # total loss in this episode

        for t in tqdm(range(len_data)):
            # take action according to epsilon greedy policy
            if np.random.random() > epsilon:
                action = np.random.randint(q_network.action_space)
            else:
                feed_dict = {q_network.input_placeholder: state}
                action = sess.run(q_network.action_pred, feed_dict = feed_dict)
                action = np.argmax(action[0])

            # next state
            next_state = get_state(data, t + 1, window_size + 1)
            reward = 0

            # now go according to the actions
            if action == 2:
                # buy
                inventory.append(data[t])
                if LOG:
                    print('Buy:' + format_price(data[t]))

            elif action == 0 and len(inventory) > 0:
                bought_price = inventory.pop(0) # remove the first element and␣
↪return the value
                profit = data[t] - bought_price # profit this transaction
                reward = max(data[t] - bought_price, 0) # reward
                total_profit += profit # add to total profit
                if LOG:
                    print("Sell: " + format_price(data[t]) + " | Profit: " +␣
↪format_price(profit))

            # condition for done
            done = t == len_data - 1

            # add to memory and make sure it's of fixed size
            memory_buffer.append((state, action, reward, next_state, done))
            if len(memory_buffer) > max_mem_size:
                memory_buffer.pop(0)

            # update state
            state = next_state

            # train the model
            if len(memory_buffer) > batch_size:

                # sample minibatches here
                mini_batch = memory_buffer[-batch_size:]

                # calculate q_value and target_values
                for state_t, action_t, reward_t, next_state_t, done_t in␣
↪mini_batch:
                    # condition for calculating y_j
                    if done_t:
                        target_pred = reward

                    else:
                        feed_dict = {network.input_placeholder: next_state_t}
                        network_pred = sess.run(network.action_pred, feed_dict␣
↪= feed_dict)
                        target_value = reward_t + gamma*np.amax(network_pred[0])

                    # q_value
                    feed_dict = {network.input_placeholder: state_t}
                    q_values = sess.run(network.action_pred, feed_dict =␣
↪feed_dict)
                    q_values[0][action_t] = target_value

                    # drop epsilon value after every action taken
                    if epsilon > epsilon_end:
                        epsilon *= epsilon_decay

                    # update the q_network parameters
                    feed_dict = {network.input_placeholder: state_t,
                                 network.q_placeholder: q_values}
                    loss, _ = sess.run([q_network.loss, q_network.train_step],␣
↪feed_dict = feed_dict)

                    # update the lists
                    ep_loss.append(loss)

        # update the outer values
        train_loss.append(ep_loss)
        train_profits.append(total_profit)
        # print val
        print('[*]Episode: {0}, loss: {1}, profits: {2}, epsilon: {3}'\
              .format(ep + 1, np.mean(train_loss[-1]), train_profits[-1],␣
↪epsilon))

    # return the values
    return train_loss, train_profits
```

[48]:

```
# run the model
q_network = DDQN(window_size, 'q_network')
sess = tf.Session(config=config)
loss, profits = train_ddqn(q_network, sess, data)
```

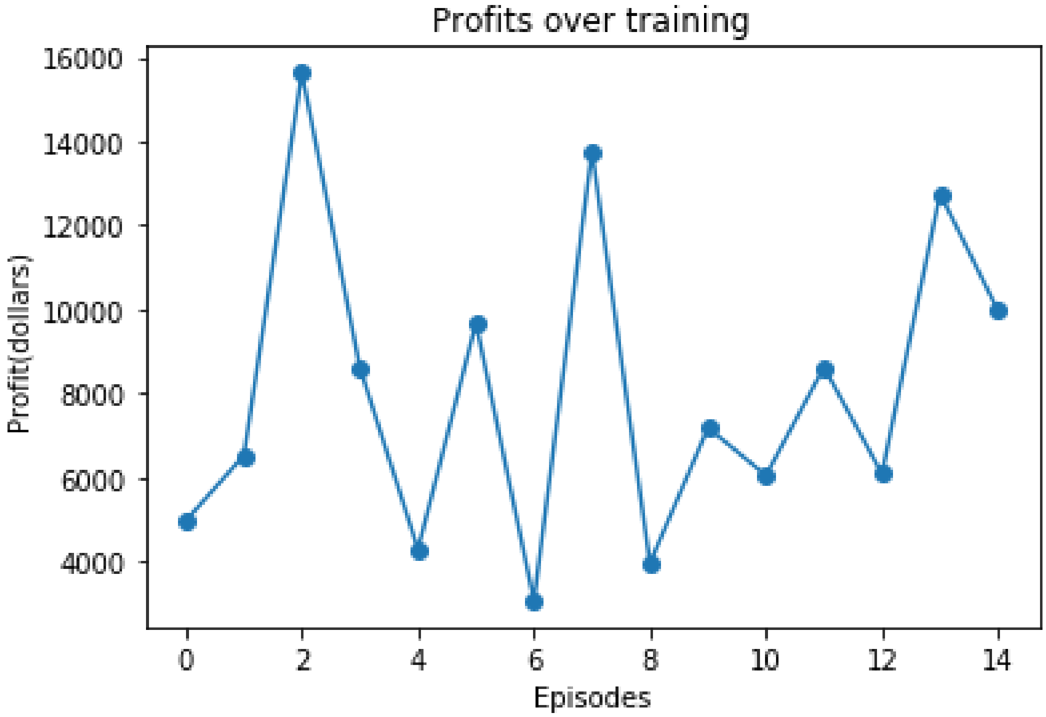
```
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efea833c190>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efea833c190>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efea833cd10>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efea833cd10>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efebc599110>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efebc599110>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efec7163e10>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efec7163e10>>: AssertionError:
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efebc508310>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efebc508310>>: AssertionError:
```

```
Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method Dense.call of <tensorflow.python.layers.core.Dense
object at 0x7efebc5c2190>> could not be transformed and will be executed as-is.
Please report this to the AutgoGraph team. When filing the bug, set the
verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full
output. Cause: converting <bound method Dense.call of
<tensorflow.python.layers.core.Dense object at 0x7efebc5c2190>>: AssertionError:
Bad argument number for Name: 3, expecting 4
100%|      | 8699/8699 [01:31<00:00, 94.84it/s]
  0%|          | 12/8699 [00:00<01:18, 110.41it/s][*]Episode: 1, loss:
3.6638989448547363, profits: 4978.8708830000005, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:35<00:00, 90.62it/s]
  0%|          | 10/8699 [00:00<01:35, 90.98it/s][*]Episode: 2, loss:
6.761518478393555, profits: 6531.6548379999995, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:36<00:00, 90.59it/s]
  0%|          | 9/8699 [00:00<01:38, 87.80it/s][*]Episode: 3, loss:
25.375991821289062, profits: 15638.322314000003, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:42<00:00, 85.28it/s]
  0%|          | 12/8699 [00:00<01:17, 112.39it/s][*]Episode: 4, loss:
10.004467010498047, profits: 8623.802422999996, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:43<00:00, 84.19it/s]
  0%|          | 9/8699 [00:00<01:41, 85.72it/s][*]Episode: 5, loss:
2.648190975189209, profits: 4296.925802000002, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:30<00:00, 95.69it/s]
  0%|          | 11/8699 [00:00<01:24, 102.23it/s][*]Episode: 6, loss:
9.680620193481445, profits: 9680.754473, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:53<00:00, 76.34it/s]
  0%|          | 10/8699 [00:00<01:33, 93.28it/s][*]Episode: 7, loss:
2.010627031326294, profits: 3074.5014719999995, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:41<00:00, 86.05it/s]
  0%|          | 11/8699 [00:00<01:19, 109.66it/s][*]Episode: 8, loss:
17.88360595703125, profits: 13771.761279999993, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:33<00:00, 92.64it/s]
  0%|          | 8/8699 [00:00<01:55, 75.06it/s][*]Episode: 9, loss:
3.625286340713501, profits: 3973.128112999998, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:36<00:00, 90.34it/s]
  0%|          | 12/8699 [00:00<01:15, 114.81it/s][*]Episode: 10, loss:
5.181102275848389, profits: 7186.825887, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:30<00:00, 96.13it/s]
  0%|          | 11/8699 [00:00<01:21, 107.05it/s][*]Episode: 11, loss:
7.5532402992248535, profits: 6055.1132590000025, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:30<00:00, 96.53it/s]
  0%|          | 10/8699 [00:00<01:30, 96.43it/s][*]Episode: 12, loss:
9.387152671813965, profits: 8628.528960000001, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:34<00:00, 91.76it/s]
  0%|          | 11/8699 [00:00<01:29, 97.10it/s][*]Episode: 13, loss:
8.608474731445312, profits: 6093.363259999999, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:33<00:00, 93.24it/s]
```

```
  0%|          | 12/8699 [00:00<01:15, 114.87it/s][*]Episode: 14, loss:
18.909765243530273, profits: 12755.859637999994, epsilon: 0.0009954452565571535
100%|      | 8699/8699 [01:34<00:00, 91.61it/s]
[*]Episode: 15, loss: 12.265534400939941, profits: 9973.962029000008, epsilon:
0.0009954452565571535
```

```
[49]: plt.title('Profits over training')
      plt.plot(profits, '-o')
      plt.xlabel('Episodes')
      plt.ylabel('Profit(dollars)')
```

```
[49]: Text(0, 0.5, 'Profit(dollars)')
```



2.3 Conclusions

- Both Dueling DQN and Double DQN give better profits than vanilla DQN.
- Although more tuning is neccessary better predictions.
- Algorithms perform better for the companies which have good and bad phases in the stock market.
- This can be explained because agent can learn all possible scenarios - good and bad stock rate in the market.