SAKI SS19 Homework 4

Author: Maximilian Lattka

Program code: https://github.com/nakami/oss-saki-ss19-exercise-4/releases/tag/oss-saki-ss19-exercise-4

Summary

For exercise 4, the instructors from senacor provided a stock market framework. The scenario contains a stock market supervising unit including stock of two companies, several traders which compete for the most valuable portfolio and two experts who offer action suggestions. As baseline, two simple traders are already realized: a Buy-And-Hold trader (BAH) which upfront buys as much stock as he possibly can and keeps it without further interaction and a Total-Trusting trader (TT) who blindly believes the suggestions of the respective experts. In comparison, the TT trader and therefore the expert suggestions lead to a higher final portfolio value in the end. Now, the students were instructed to implement a Deep-Q-Learning trader (DQL) to challenge the baseline by using a deep neural network in combination with Q learning.

Overview

Data: `datasets/*`

The process of data exploration for this task can be kept brief as the provided framework does all of the data handling. There are datasets for two periods (1962-2011 and 2012-2015) for both respective stocks (Company A and Company B) in the `datasets/` folder. The skeleton of the DQL trader (`deep_q_learning_trader.py`) suggests running trainings with the first period and testing with the second. The main file which also outputs the trader graphs in comparison only uses the training period. For each day, the `trade(...)`-function of each trader is called and returns a list of orders (buy, hold or sell arbitrary amounts stock of Company A or B) to be performed.

DQL trader implementation

Code: `traders/deep_q_learning_trader.py`

As the deep network was mostly implemented and some parameters were already given, I first set up meaningful starting conditions by introducing an input layer of 2 input neurons which represent a current state and an output layer of size 4 to stand for 4 action predictions. To keep things simple, the state only consists of two numerical values of the expert suggestions. Input neuron 0 stands for stock A and in particular holds expert A either suggesting to sell (0), hold (1) or buy (2). The second input neuron holds the vote for stock B. The output layer (size 4) on the other hand, has two neurons for A and two for B, each being a suggestion to sell A/B (neuron 0/2) or to buy A/B (neuron 1/3). I differentiate between an action for A and one for B, so when choosing the index of the neuron with higher output is taken (0 and 1 compared for A; 2 and 3 compared for B). Keeping it clear, the trader either buy or sell as much as possible. For the advanced techniques I realized all three suggestions the lecturers included in their slides. I implemented exploration (random decision making) by using an arbitrary but fixed `epsilon` value which leads to random choices and is reduced by a `epsilon_decay` factor in the long term. Also, my trader makes use of gamma parameter for Q-learning and a memory queue to re-learn past events. In regards to the rewards, I save per stock prices for both A and B respectively and observe the percentage in change over the last day. The selling or buying decisions are then weighted whether they were meaningful. Plus, I take for A and B each its current maximum of each prediction, scale it by gamma and increase the particular reward. I decreased the suggested batch_size of 64 to 16 as it turned out reasonable in my implementation regarding results and the required training time. Experimenting with the residual parameters, I chose some to lock in place and yet some others to be tweaked over multiple comparable instances of the DQL trader. See a more detailed description in the *Evaluation* section.

Evaluation

Code: `traders/deep_q_learning_trader*.py`

Data: `eval/*.png`

I trained my DQL trader with different parameters by using the code provided in deep_q_learning_trader.py and ran the stock exchange file right after. For each parameter variation, I saved the DQL file under a different name, saved both graph visualisations and renamed both data files of the respective model. For evaluation purpose, I collected the final portfolio values from each stock exchange graph. Of course, the decision making on the whole period and therefore the resulting graphs are not irrelevant. Yet, it would overdraw the scope of this report if done properly. Although I played with quite a few settings, one cannot play out all possible variations as there are many variables which can be chosen fairly arbitrary - yielding thousands of possible models. The provided baseline traders score as follows:

	final portfolio value (thousand)
buy_and_hold_trader	3.7
trusting_trader	22.3

The final portfolio value of the baseline BAH and TT trader

Throughout the experiments, I kept the following parameters fixed:

```
self.state_size = 2

self.hidden_size = 50

self.action_size = 4

self.epsilon_min = 0.01

self.batch_size = 16

self.min_size_of_memory_before_training = 1000

self.memory = deque(maxlen=2000)

self.learning_rate = 0.001
```

Gamma experiments

Parameter fixed: `self.epsilon = 0.0`

At first, I tried different gamma variations while disabling the random exploration completely. Choosing gamma to be 1.0 apparently is not a good idea and even underperforms the BAH trader baseline. Besides that I had different results that did not vary too much from each other. A DQL with gamma of 0.5 yielded the best result with 57.4k as final portfolio value.

	gamma	**final portfolio value (thousand)**
eps0_gamma0	0.0	53.0
eps0_gamma10	0.1	52.0
eps0_gamma50	0.5	57.4
eps0_gamma90	0.9	48.6
eps0_gamma100	1.0	2.5

The performance of my DQL trader implementation with different gamma factors

Epsilon experiments

Parameter fixed: `self.gamma = 0.5`

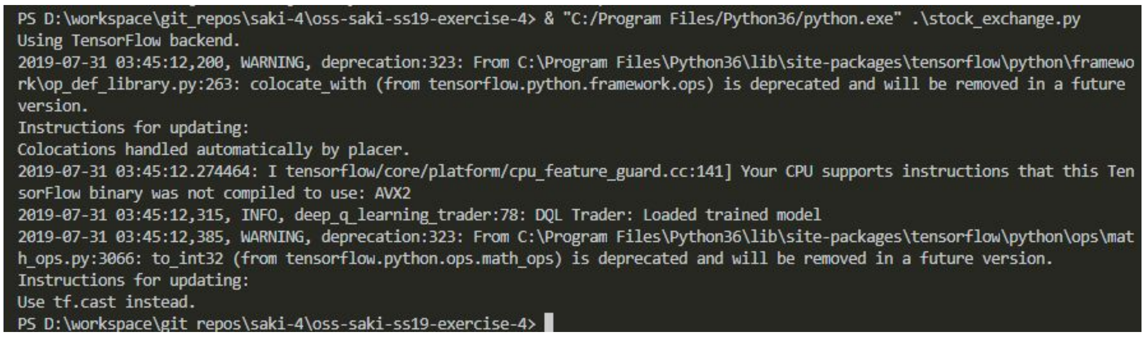
I experimented with different epsilon values and decay factors. Also, I took the most promising gamma factor of 0.5 from earlier experiments. Choosing a high decay factor and therefore enabling longer exploration seems to worsen results for my implementation and the data set. Basically, it seems that my implementation in this particular setting does not require exploration to optimize the portfolio progression, but may vary in a different setting. Keep in mind that random exploration plays a role in these traders/models and the results vary from run to run.

	epsilon	**epsilon_decay**	**final portfolio value (thousand)**
eps100_decay10_gamma50	1.0	0.1	52.7
eps100_decay50_gamma50	1.0	0.5	50.9
eps100_decay90_gamma50	1.0	0.9	41.7
eps50_decay10_gamma50	0.5	0.1	56.5
eps50_decay50_gamma50	0.5	0.5	43.1
eps50_decay90_gamma50	0.5	0.9	48.7

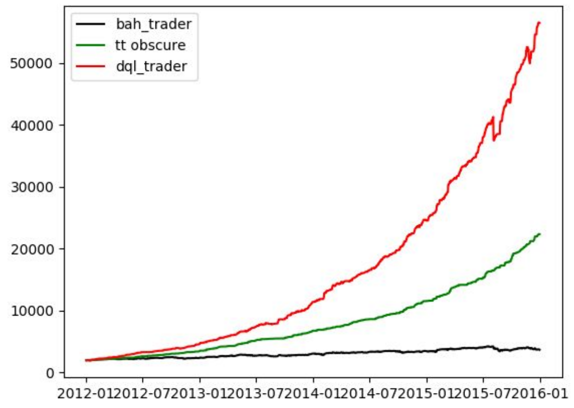
The performance of my DQL trader implementation with varying epsilon values and epsilon decay factors

The trader I submit in `deep_q_learning_trader.py` is eps50_decay10_gamma50 with a final portfolio value of 56.5k. I chose this variable setting over eps0_gamma50 (with a higher portfolio value of 57.4), because in contrast eps50_decay10_gamma50 includes exploration which may more likely result in better results for different data sets.

Screenshots



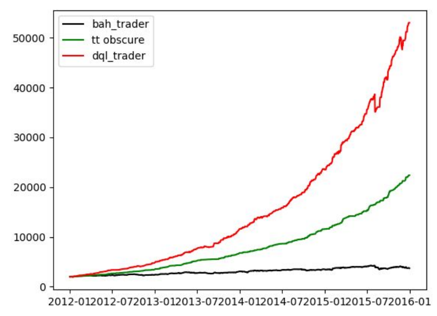
```
PS D:\workspace\git_repos\saki-4\oss-saki-ss19-exercise-4> & "C:/Program Files/Python36/python.exe" .\stock_exchange.py
Using TensorFlow backend.
2019-07-31 03:45:12,200, WARNING, deprecation:323: From C:\Program Files\Python36\lib\site-packages\tensorflow\python\framewo
rk\op_def_library.py:263: colocate_with (from tensorflow.python.framework.ops) is deprecated and will be removed in a future
version.
Instructions for updating:
Colocations handled automatically by placer.
2019-07-31 03:45:12.274464: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this Ten
sorFlow binary was not compiled to use: AVX2
2019-07-31 03:45:12,315, INFO, deep_q_learning_trader:78: DQL Trader: Loaded trained model
2019-07-31 03:45:12,385, WARNING, deprecation:323: From C:\Program Files\Python36\lib\site-packages\tensorflow\python\ops\mat
h_ops.py:3066: to_int32 (from tensorflow.python.ops.math_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
PS D:\workspace\git_repos\saki-4\oss-saki-ss19-exercise-4>
```



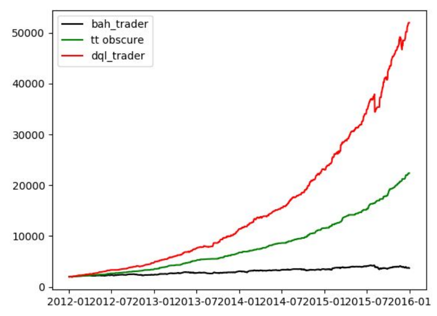
`deep_q_learning_trader.py`

(or eps50_decay10_gamma50)

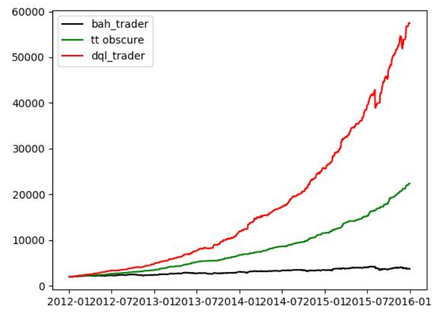
```
self.gamma = 0.5
self.epsilon = 0.5
self.epsilon_decay = 0.1
```



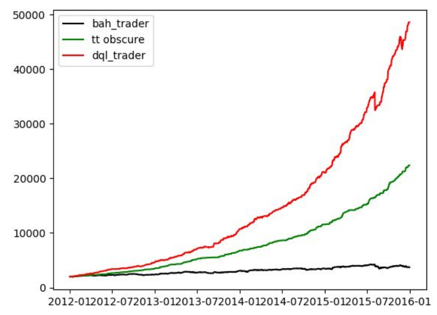
eps0_gamma0



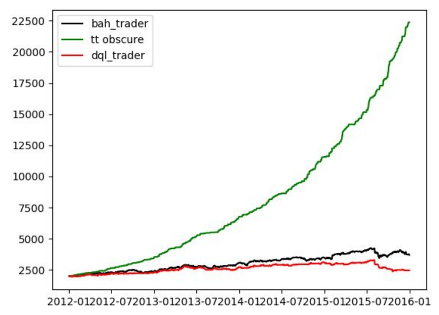
eps0_gamma10



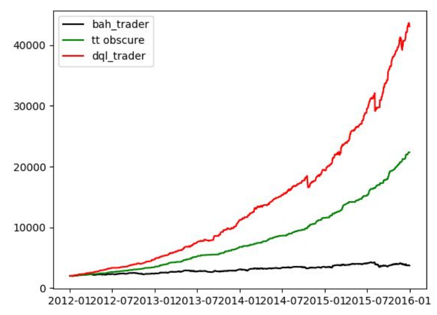
eps0_gamma50



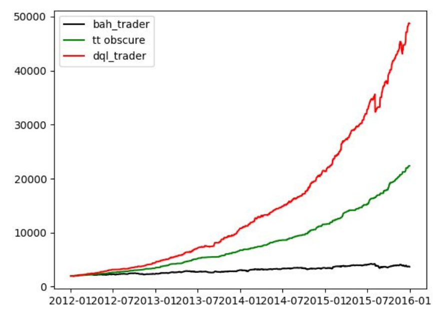
eps0_gamma90



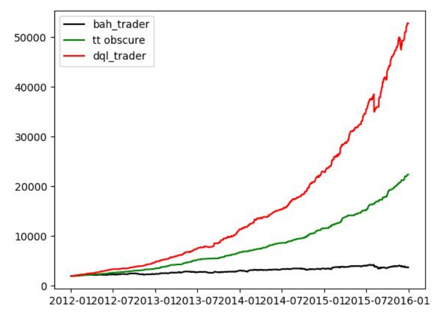
eps0_gamma100



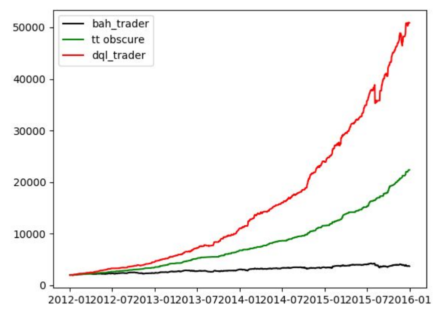
eps50_decay50_gamma50



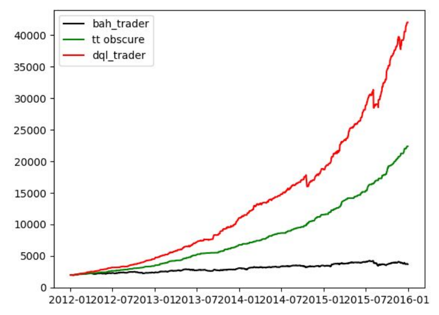
eps50_decay90_gamma50



eps100_decay10_gamma50



eps100_decay50_gamma50



eps100_decay90_gamma50