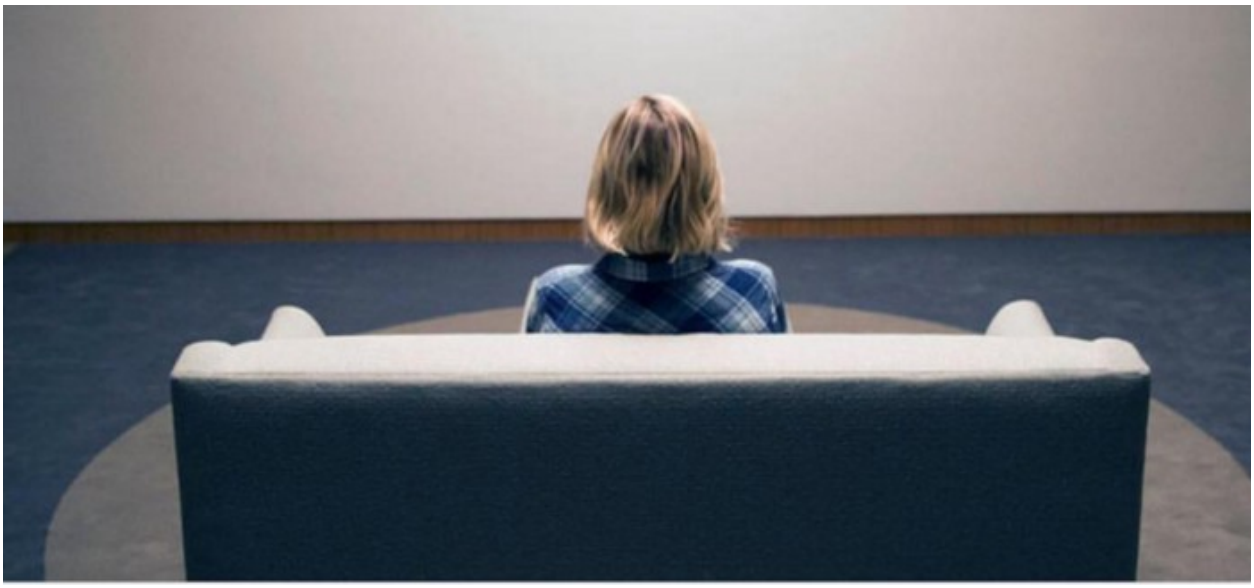
A blundering guide to making a deep actor-critic bot for stock trading

Tom Grek Follow
Sep 1, 2018 · 12 min read

Technical analysis lies somewhere on the scale of wishful thinking to crazy complex math. If there's a real trend in the numbers, irrespective of the fundamentals of a particular stock, then given a sufficient function approximator (… like a deep neural network) reinforcement learning should be able to figure it out.

Here's a fun and maybe profitable project in which I try to do that. I've been working with RL for less than six months and still figuring things out, but after making AI learners for a few basic games, time-series stock market data was at top of mind.

It wasn't easy and, instead of writing an article presenting myself as an expert with a solved problem, this article documents my process and blunders I made along the way. There were many.



Reinforcement learning is The Good Place

Do note that if you are totally new to RL, you could probably benefit from reading my previous article on Deep-Q learning first. This one is a bit different, but it also starts from a higher level with more implicit knowledge.

If you just want to skip to the code/notebook, it's here, but remember: past performance is no guarantee of future results.

Setting up the problem

One thing I've learned about RL is that framing the problem is really important for success. I decided that my bot:

- Would be given a starting amount of cash
- Would learn to maximize the value of its portfolio (shares and cash) at the end of a given time interval
- At each time step, could buy or sell shares or do nothing
- If it buys more than the money it has available, the run ends; if it sells more than it has, the run ends. It must learn the rules of the game, as well as how to play it well.

To keep things manageable, I decided only to work with a pair of presumably somewhat correlated stocks: AAPL and MSFT. Initially I wanted the bot to be able to pick how many shares it buys or sells at each timestep, and this led me down a rabbit hole of "continuous action space" reinforcement learning. In a discrete space the bot can get an idea of the value of each of its discrete actions given a current state. The more complex the space, the harder training is; in a continuous space the range of actions proliferates exponentially. For simplicity I pared down my ambitions for now so that the AI could only buy or sell a single stock per timestep.

As usual, we have a neural network for the bot and an environment that reacts to the bot's actions, returning a reward each step depending how good the bot's action was.

Before proceeding further, here's a Captain Obvious pro tip: *unit test everything as you go!* Trying to debug an environment and an AI simultaneously ("or do I just need to train it longer or adjust the hyperparameters?!") can be a bit of a nightmare.

Preparing the data

Quandl, a data platform, makes getting stock data really easy; if you exceed free limits you can sign up quickly for a free API key too:

```
msf = quandl.get('WIKI/MSFT', start_date="2014-01-01",
end_date="2018-08-20")
```

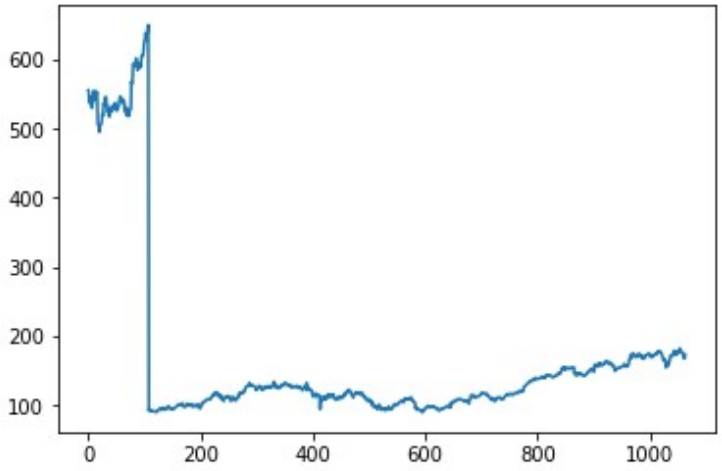
Date	Open	High	Low	Close	Volume	Ex-Dividend	Split Ratio	Adj. Open	Adj. High	Adj. Low	Adj. Close	Adj. Volume
2014-01-02	37.350	37.40	37.10	37.16	30632200.0	0.0	1.0	33.704254	33.749373	33.478656	33.532800	30632200.0
2014-01-03	37.200	37.22	36.60	36.91	31134800.0	0.0	1.0	33.568895	33.586943	33.027461	33.307202	31134800.0
2014-01-06	36.850	36.89	36.11	36.13	43603700.0	0.0	1.0	33.253059	33.289154	32.585290	32.603338	43603700.0
2014-01-07	36.325	36.49	36.21	36.41	35802800.0	0.0	1.0	32.779304	32.928198	32.675529	32.856007	35802800.0
2014-01-08	36.000	36.14	35.58	35.76	59971700.0	0.0	1.0	32.486028	32.612362	32.107024	32.269454	59971700.0

It returns a nice Pandas dataframe

I won't go through all the tedious steps of data preparation, you can follow along in my notebook here. But I do want to point out the value of not skipping at least a cursory EDA (exploratory data analysis) — I didn't discover at first that the date range I selected had a big discontinuity for AAPL.

```
plt.plot(range(0, len(apl_open)), apl_open)
```

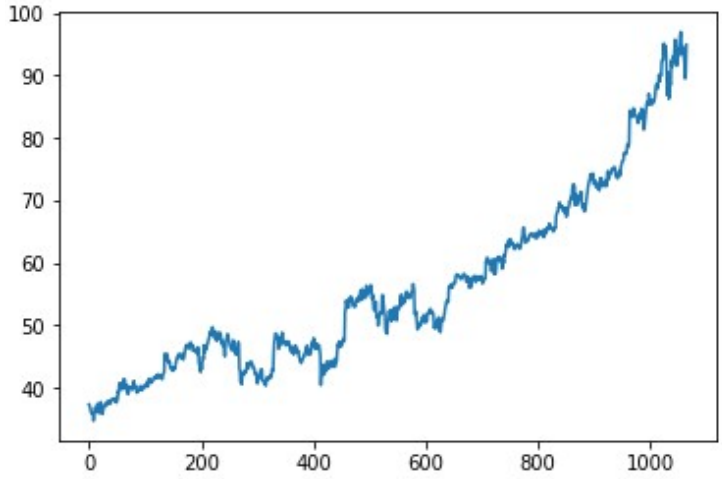
```
[<matplotlib.lines.Line2D at 0x22b96dcad30>]
```



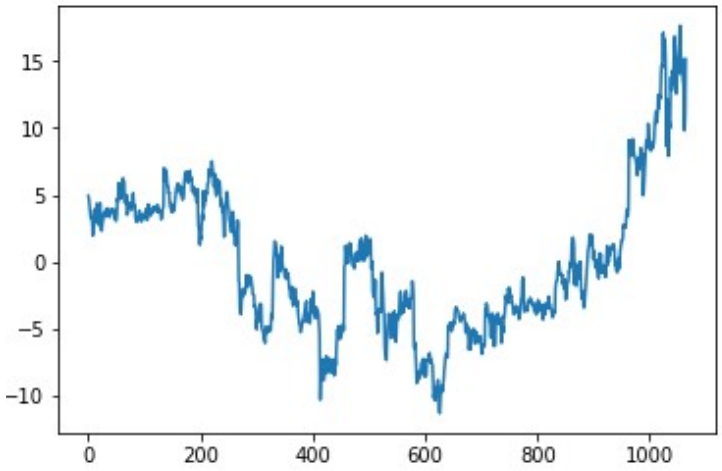
I turned out that on 9 June 2014, Apple stock was split in the ratio of 1:7. So I simply divided everything before that date by 7 to keep things consistent.

The other important thing is to remove trends from the data. At first, I didn't do this, and my AI simply learned to maximize its returns by buying a stock early on and holding it til the end of the game, because there was a general upwards trend. Not interesting! Additionally, AAPL and MSFT had very different means and stddev's, so it was like asking the AI to learn to trade apples (pun!) and oranges.

Apparently there are several different ways to remove trends like this; I went with one from SciPy's signal processing module. It fits a linear approximator to the data, then subtracts the estimated data from the real data. For example, MSFT was transformed from:



To:



You can see after the transformation there are some negative stock values which wouldn't make sense (better than free!). I dealt with this by just adding a constant amount to all the input data to shift it into the positive.

Last thing to note on the data is that I treated both stock prices just as a consecutive array. In the chart above you can see 100 "days" worth of data, but the actual time period is longer because of weekends and holidays. For simplicity I totally ignored the realities of time in this project, but in the real world I believe they are significant.

Let's leave aside the issue of train/test split for now; you will understand why when I come back to it later.

Actor-Critic RL

I moved away from Q-learning for the implementation of the trader bot, for two reasons:

- Everyone says actor-critic is better; and
- It's actually kind of more intuitive. Forget the Bellman equation, just use another neural net to calculate state values, and optimize it just like you optimize the main action-selecting (aka policy, aka actor) neural net.

In fact, the policy network and the value network can just sit as two different linear-layer heads on top of a main "understanding the world" neural net.

The code is quite straightforward, in pseudocode like this:

```
For 1..n episodes:
  Until environment says "done":
    Run the current state through the network
    That yields:
      a set of "probabilities" for each action
      a value for how good the state is

    Sample from those probabilities to pick an action

    Act upon the environment with that action
    Save the state, action, reward from environment to a buffer

  Adjust each saved reward by the value of the terminal state
  discounted by gamma each timestep

  Loss for the episode is the sum over all steps of:
    The log probability of the action we took * its reward
    The value of that action compared to what was expected

  Propagate the loss back and adjust network parameters
```

You can see that A-C does not attempt to optimize for the choice of action — we do not ever know what the objectively correct action would have been — but rather:

- The degree of certainty about each action, given its resulting reward
- The degree of surprise about each reward, given the state

As the network becomes more certain about a given action (*p -> 1*), loss decreases (*ln(p) -> 0*) and it learns slower. And on the flipside, unlikely actions that were sampled and which led unexpectedly to big rewards, produce a much bigger loss. Which, in backprop/gradient descent, causes the optimizer to increase the probability of that unlikely action in future.

In my A-C implementation, based on the official PyTorch code, there's no epsilon-greedy aspect to the policy ("with a decreasing probability, take a random action instead of your best guess in order to encourage exploration of the action space"). Instead, we sample from the policy network's outputs according to their probability, so there's always a small chance some unlikely action will be chosen.

In addition, the reward signals are perturbed slightly before feeding them in to gradient descent, so the backprop doesn't merely perfectly chase a local minimum, but leads to some slightly wrong weight updates that magnify at the output causing unexpected actions to be selected. I see that as a kind of regularization too. In my implementation, I let the perturbation be a random small amount that differs on each training step; empirically this worked better than just adding the same small constant each time. In future I might anneal the amount over time, as the network starts to converge.

The network I ultimately settled on was not complex, except that it was probably a mistake for me to use a recurrent layer, for reasons we'll come to:

```
class Policy(nn.Module):
    def __init__(self):
        super(Policy, self).__init__()
        self.input_layer = nn.Linear(8, 128)
        self.hidden_1 = nn.Linear(128, 128)
```

The environment code and much of the training loop code is not very interesting boilerplate; if you are interested, it's in the notebook on Github.

The state, and some blunders

The state I finally opted for was a vector consisting of:

```
[AAPL holdings,
MSFT holdings,
cash,
current timestep's AAPL opening price,
current timestep's MSFT opening price,
current portfolio value,
past 5 day average AAPL opening price,
past 5 day average MSFT opening price]
```

This certainly was not the first thing I tried! At first, I hand-engineered a bunch of features: many different moving averages, max and min prices seen, and so on. Later I dropped all that thinking that the recurrent layer would be able to figure those features out for itself.

"Look away, look away ... these RL methods will wreck your evening, your whole life and your day" — fellow blunderer Count Olaf from A Series Of Unfortunate Events. Actor Neil Patrick Harris has no critics!

You see, I wanted the AI to learn the fundamentals of a BUY signal and a SELL signal, rather than just learning to regurgitate the time series. So, each time the environment was reset, I set it to start at a different random timestep and use a different random stride through the data, and defined "done" as a different random number of steps in the future.

While I think this is the right approach, unfortunately it made it basically impossible for the RNN to learn anything useful because it had no idea where it was in the series each time. So I abandoned the varying start point, stride, and sequence length, and held them the same throughout all the training episodes.

In future I'd definitely keep the handcrafted features too; while in theory the RNN might be able to figure them out, training time can get really long.

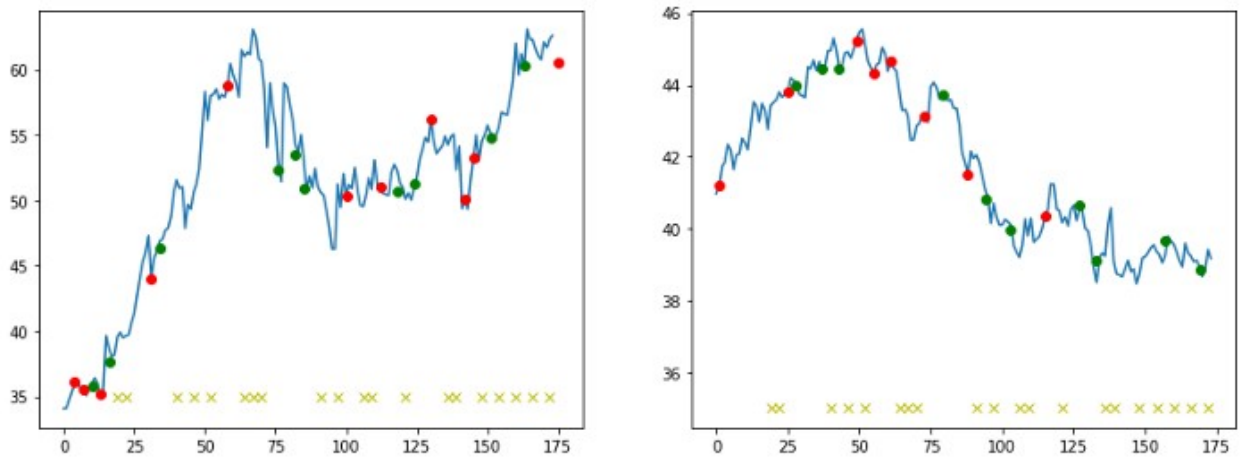
Results

The bot learned to not cheat or bankrupt itself, most of the time, and it was able to make a significant profit: here you can see 36% returns (on the data it trained on).

```
Failed goes: 1 / 50, Avg Rewards per successful game: 4098.34423828125
Avg % profit per game: 0.3571603298187256
Avg % profit per finished game: 0.36444932222366333
```

(Here, 0.36 means 36% profit)

(The starting portfolio value was on average about 3000, so 4000 rewards represents a 36% rise.) The bot learned the rules, too: it didn't bankrupt itself or try to sell more shares than it had, except one time. Here's how it looked on a section of training data:



Red: BUY, Green: SELL, Yellow Cross: do nothing

Tinkering to get better results

One important thing I learned is that if different states yield the same reward, it complicates training and slows convergence. So here's a useful trick:

Add a time element to the reward. As time moves on, actions that don't kill the game are successively more valuable; or, the longer the agent continues in its environment, the more reward it gets.

I was giving the agent a small positive reward for actions and a small negative reward for doing nothing. So, an action like "buy AAPL" might yield +0.1 reward at any time during the sequence. But we ultimately want the bot to keep going til the end of the sequence without bankrupting itself, which I encouraged by mixing in a time element, namely, subtracting the number of steps left to go from the reward. "buy AAPL" with 50 steps to go might yield a reward of -49.9; with 49 steps to the reward would be -48.9 (better), and so on.

While that helped it to learn to survive longer, I was puzzled why it seemed my bot would only learn *either* to survive, *or* to maximize gain, but struggled to learn to do both together. **It turned out that designing rewards is quite difficult!** The step reward needed to be a balance of how long the bot has survived plus how much gain it has made. The exact formula I used is in the notebook.

A couple of other empirical findings:

- Compensate for the infrequency of good rewards by increasing them a lot compared to all the negative rewards.
- There's no need to subtract the starting portfolio value from the ending value when calculating a reward. You get this for free as the bot always tries to optimize higher.
- Once the bot hits its maximum, it starts oscillating; at this point, you better stop training.
- The ideal place for training seems to be "on the edge" between reaching the terminal state half the time, and failing to get there half the time.

. . .

Train/Test split

At this point everything looks great and you want to test the model on a different time range of stock data in order to ensure it generalized well. After all, we'd like to be able to deploy this on our Robinhood trading accounts in future and make lots of money.

What I did next was to go back earlier in time to 2012 and grab the stock data from then to present; I then split it into roughly 2/3 train and reserved the last 1/3 for test.

Unfortunately, after accounting for trend, both stocks experienced slides during the 2012–2016/17 period and if you remember, the market was considered to be rather volatile then. This made it really difficult for the bot to learn how to make a profit. (From the original 2014–2018 data I used, simply buying a stock and holding onto it was a pretty good strategy).

To aid the bot, I set the stride to 1 and sequence length to the length of the training data, so it had as much sequential data as possible to train on. Now, there were enough action-reward pairs being processed that it was too slow to continue on CPU, so by adding 5 `.cuda()` statements I was able to move much of the work to the GPU.

Now, you can imagine the chances of the bot getting to the end of a sequence of ~1000 moves in order to get a meaningful reward. Starting off random, it has to play by rules it doesn't even know yet and not make a single false move. Even if it does that, it's pretty useless to us unless it made some trades and a profit during that time. Some hours' wasted training time confirmed this not to be a viable way to proceed.

To really make it work I had to discover and apply what I think is a cheap trick:

Relax the requirement for the agent to learn the rules at first; let it train in a very easy environment where it cannot die, and ramp up the difficulty over successive training periods.

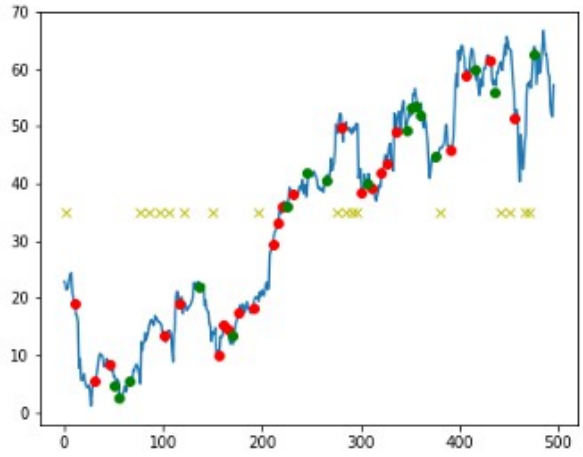
With the financial environment I've made here, that was not too difficult. I just had to start the agent off with lots of AAPL and MSFT shares and a big pot of cash. Train it for a while, then reduce the starting shares/cash, train it again, and so on. Eventually it learned to respect the rules.

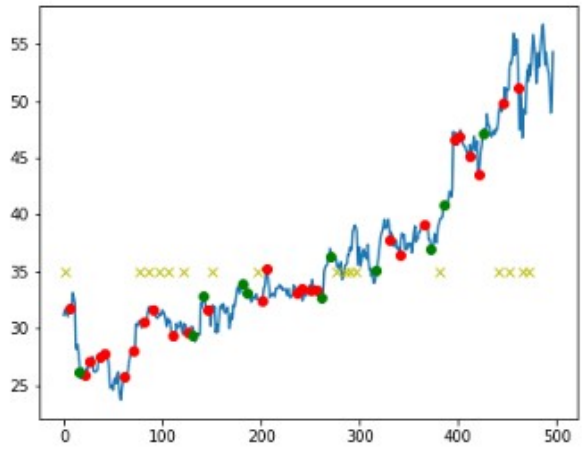
Results on the test data, which the bot had not trained on, were not great. I suppose that if they were, everyone would have done this already and we'd cease to have a functioning market.

```
Failed goes: 0 / 50, Avg Rewards per successful game: 7005.291015625
Avg % profit per game: -0.26925191283226013
Avg % profit per finished game: -0.26925191283226013
```

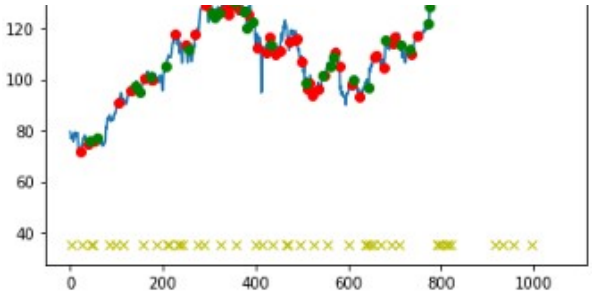
It has, however, learned to follow the rules

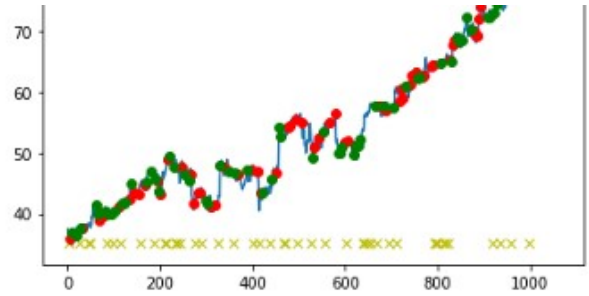
Here's one of its "trading strategies", graphed. Red means buy, green means sell, and left is AAPL and right MSFT. The yellow crosses are timesteps where the bot chose to take no action.





And here's a longer run, showing buys and sells in the context of the original, non-transformed share prices. As you can see, it's pretty jumbled.





I'm fascinated by RL's potential as a universal optimizer that can learn to function by itself, but in truth it's fiddly to get working well — particularly in high-entropy environments like trading.

Thus ends my tale of woe and blunder, for now. We didn't get rich, but we did figure out a state of the art AI technique, and surely those are two things that will prove correlated over time!