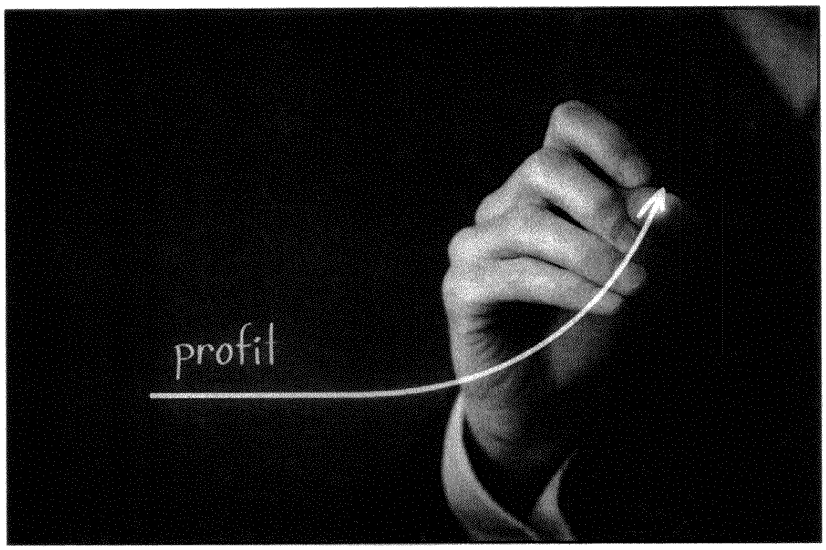
Optimising deep learning trading bots using state-of-the-art techniques

Let's teach our deep RL agents to make even more money using feature engineering and Bayesian optimisation

Adam King

Jun 4 · 17 min read



In the last article, we used deep reinforcement learning to create Bitcoin trading bots that don't lose money. Although the agents were profitable, the results weren't all that impressive, so this time we're going to step it up a notch and massively improve our model's profitability.

As a reminder, the purpose of this series of articles is to experiment with state-of-the-art deep reinforcement learning technologies to see if we can create profitable Bitcoin trading bots. It seems to be the status quo to quickly shut down any attempts to create reinforcement learning algorithms, as it is "the wrong way to go about building a trading algorithm". However, recent advances in the field have shown that RL agents are often capable of learning much more than supervised learning agents within the same problem domain. For this reason, I am writing these articles to see just how profitable we can make these trading agents, or if the status quo exists for a reason.

We will first improve our model's policy network and make the input data set stationary, so we can learn more from less data. Next, we will use advanced feature engineering to improve our agent's observation space and fine tune our reward function to produce more attractive strategies. Finally, we will use a technique called Bayesian optimisation to zone

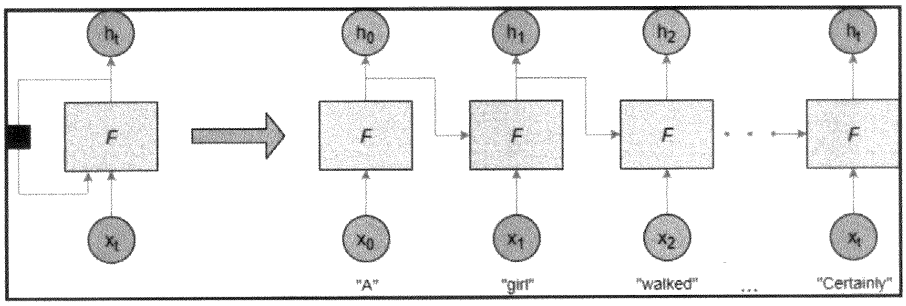
in on the most profitable hyper-parameters, before training and testing the final agents profitability. Hold on to your seats everyone, this is going to be a wild ride.

Modifications

The first thing we need to do to improve the profitability of our model, is make a couple improvements on the code we wrote in the last article. If you do not yet have the code, you can grab it from my **GitHub**.

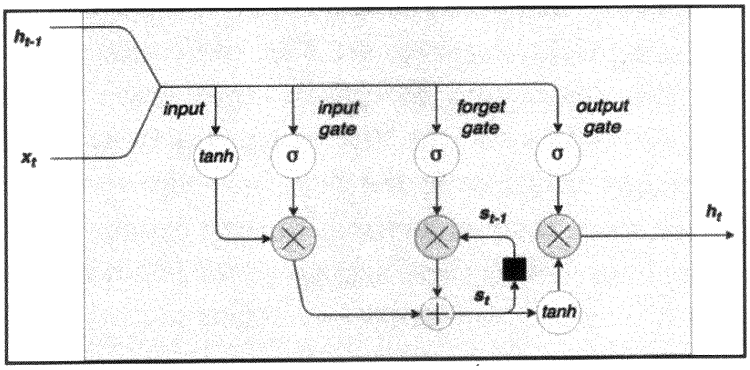
Recurrent Networks

The first change we need to make is to update our policy to use a recurrent, Long Short-Term Memory (LSTM) network, in place of our previous, Multi-Layer Perceptron (MLP) network. Since recurrent networks are capable of maintaining internal state over time, we no longer need a sliding "look-back" window to capture the motion of the price action. Instead, it is inherently captured by the recursive nature of the network. At each time step, the input from the data set is passed into the algorithm, along with the output from the last time step.



Source: https://adventuresinmachinelearning.com/recurrent-neural-networks-lstm-tutorial-tensorflow/

This allows the LSTM to maintain an internal state that gets updated at each time step as the agent "remembers" and "forgets" specific data relationships.



Source: https://adventuresinmachinelearning.com/recurrent-neural-networks-lstm-tutorial-tensorflow/

To better allow the agent to remember, let's modify the code as follows:

```
from stable_baselines.common.policies import MlpLstmPolicy
model = PPO2(MlpLstmPolicy, train_env, tensorboard_log="./tensorboard")
```

Here we update our PPO2 model to use the MlpLstmPolicy, to take advantage of its recurrent nature.

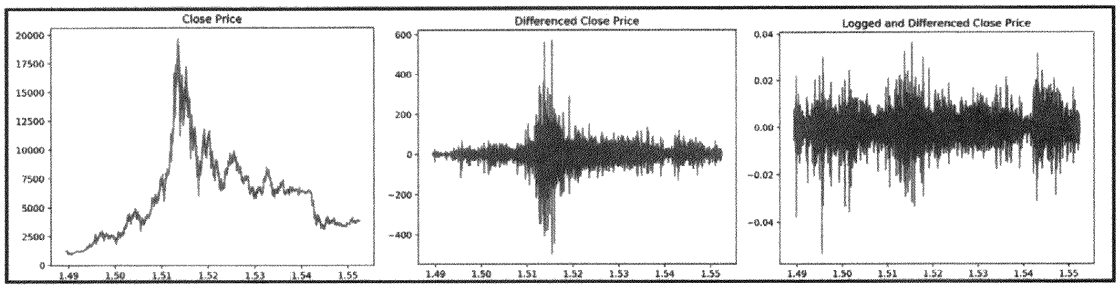
Stationary Data

It was also pointed out to me on the last article that our time series data is not stationary, and therefore, any machine learning model is going to have a hard time predicting future values.

A *stationary time series* is one whose mean, variance, and auto-correlation (lagged correlation with itself) are **constant**.

The bottom line is that our time series contains an obvious trend and seasonality, which both impact our algorithms ability to predict the time series accurately. We can fix this by using differencing and transformation techniques to produce a more normal distribution from our existing time series.

Differencing is the process of subtracting the derivative (rate of return) at each time step from the value at that time step. This has the desired result of removing the trend in our case, however, the data still has a clear seasonality to it. We can attempt to remove that by taking the logarithm at each time step before differencing, which produces the final, **stationary** time series, shown below on the right.




```
df['diffed'] = df['Close'] - df['Close'].shift(1)
df['logged_and_diffed'] = np.log(df['Close']) - np.log(df['Close']).shift(1)
```

We can verify the produced time series is stationary by running it through an Augmented Dickey-Fuller Test:

```
from statsmodels.tsa.stattools import adfuller
df['logged_and_diffed'] = np.log(df['Close']) - np.log(df['Close']).shift(1)
result = adfuller(df['logged_and_diffed'].values[1:], autolag="AIC")
print('p-value: %f' % logged_and_diffed_result[1])
```

Here we run the Augmented Dicker-Fuller Test on our transformed data set to ensure stationarity.

Doing this gives us a p-value of 0.00, allowing us to reject the test's null hypothesis and confirm our time series is stationary.

Now that we've got that out of the way, we are going to further update our observation space using a bit of feature engineering.

Feature Engineering

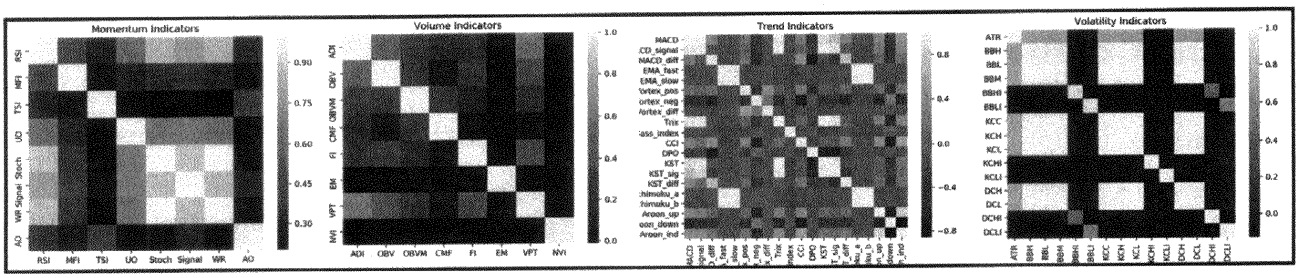
To further improve our model, we are going to be doing a bit of feature engineering.

Feature engineering is the process of using domain-specific knowledge to create additional input data that improves a machine learning model.

In our case, we are going to be adding some common, yet insightful technical indicators to our data set, as well as the output from the StatsModels SARIMAX prediction model. The technical indicators should add some relevant, though lagging information to our data set, which will be complimented well by the forecasted data from our prediction model. This combination of features should provide a nice balance of useful observations for our model to learn from.

Technical Analysis

To choose our set of technical indicators, we are going to compare the correlation of all 32 indicators (58 features) available in the `ta` library. We can use `pandas` to find the correlation between each indicator of the same type (momentum, volume, trend, volatility), then select only the least correlated indicators from each type to use as features. That way, we can get as much benefit out of these technical indicators as possible, without adding too much noise to our observation space.



Seaborn heatmap of technical indicator correlation on BTC data set.

It turns out that the volatility indicators are all highly correlated, as well as a couple of the momentum indicators. When we remove all duplicate features (features with an absolute mean correlation > 0.5 within their group), we are left with 38 technical features to add to our observation space. This is perfect, so we'll create a utility method named `add_indicators` to add these features to our data frame, and call it within our environment's initialization to avoid having to calculate these values on each time step.

```
def __init__(self, df, initial_balance=10000, commission=0.0003, reward_func='profit', **
    super(BitcoinTradingEnv, self).__init__()

    self.df = df.fillna(method='bfill')
    self.df = add_indicators(self.df.reset_index())
    self.stationary_df = log_and_difference(
        self.df, ['Open', 'High', 'Low', 'Close', 'Volume BTC', 'Volume USD'])

    ...
```

Here we initialize our environment, adding the indicators to our data frame before making it stationary.

Statistical Analysis

Next we need to add our prediction model. We've chosen to use the Seasonal Auto Regressive Integrated Moving Average (SARIMA) model to provide price predictions because it can be calculated very quickly at each step, and it is decently accurate on our stationary data set. As a bonus, it's pretty simple to implement and it allows us to create a confidence interval for its future predictions, which is often much more insightful than a single value. For example, our agent can be learn to be more cautious trusting predictions when the confidence interval is small and take more risk when the interval is large.

```
def _next_observation(self):
    ...

    obs = scaled_features[-1]

    past_df = self.stationary_df['Close'][:self.current_step + self.n_forecasts + 1]
    forecast_model = SARIMAX(past_df.values)
    model_fit = forecast_model.fit(method='bfgs', disp=False)
    forecast = model_fit.get_forecast(steps=self.n_forecasts, alpha=(1 - self.confidence_i

    obs = np.insert(obs, len(obs), forecast.predicted_mean, axis=0)
    obs = np.insert(obs, len(obs), forecast.conf_int().flatten(), axis=0)

    ...
```

Here we add the SARIMAX predictions and confidence intervals to our observation space.

Now that we've updated our policy to use a more applicable, recurrent network and improved our observation space through contextual feature engineering, it's time to optimize all of the things.

Reward Optimization

One might think our reward function from the previous article (i.e. rewarding incremental net worth gains) is the best we can do, however, further inspection shows this is far from the truth. While our simple reward function from last time was able to profit, it produced volatile strategies that often lead to stark losses in capital. To improve on this, we are going to need to consider other metrics to reward, besides simply unrealized profit.

A simple improvement to this strategy, as mentioned by Sean O'Gordman in the comments of my last article, is to not only reward profits from holding BTC while it is *increasing* in price, but also reward profits from not holding BTC while it is *decreasing* in price. For example, we could reward our agent for any incremental *increase* in net worth while it is holding a BTC/USD position, and again for the incremental *decrease* in value of BTC/USD while it is not holding any positions.

While this strategy is great at rewarding increased returns, it fails to take into account the risk of producing those high returns. Investors have long since discovered this flaw with simple profit measures, and have traditionally turned to risk-adjusted return metrics to account for it.

Volatility-Based Metrics

The most common risk-adjusted return metric is the Sharpe ratio. This is a simple ratio of a portfolio's excess returns to volatility, measured over a specific period of time. To maintain a high Sharpe ratio, an investment must have both high returns and low volatility (i.e. risk). The math for this goes as follows:

$$Sharpe = \frac{R_P - R_B}{\sigma_P} = \frac{\text{Portfolio returns} - \text{Benchmark returns}}{\text{Standard deviation of portfolio}}$$

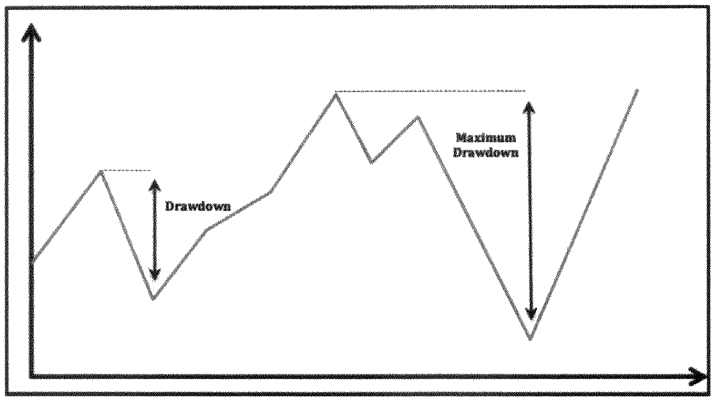
This metric has stood the test of time, however it too is flawed for our purposes, as it penalizes upside volatility. For Bitcoin, this can be problematic as upside volatility (wild upwards price movement) can often be quite profitable to be a part of. This leads us to the first rewards metric we will be testing with our agents.

The Sortino ratio is very similar to the Sharpe ratio, except it only considers downside volatility as risk, rather than overall volatility. As a result, this ratio does not penalize upside volatility. Here's the math:

$$Sortino = \frac{R_P - R_B}{\sigma_D} = \frac{\text{Portfolio returns} - \text{Benchmark returns}}{\text{Standard deviation of downside}}$$

Additional Metrics

The second rewards metric that we will be testing on this data set will be the Calmar ratio. All of our metrics up to this point have failed to take into account *drawdown*.



Drawdown is the measure of a specific **loss in value** to a portfolio, from **peak to trough**.

Large drawdowns can be detrimental to successful trading strategies, as long periods of high returns can be quickly reversed by a sudden, large drawdown.

To encourage strategies that actively prevent large drawdowns, we can use a rewards metric that specifically accounts for these losses in capital, such as the Calmar ratio. This ratio is identical to the Sharpe ratio, except that it uses maximum drawdown in place of the portfolio value's standard deviation.

$$Calmar = \frac{R_P - R_B}{\mu_D} = \frac{\text{Portfolio returns} - \text{Benchmark returns}}{\text{Maximum drawdown of portfolio}}$$

Our final metric, used heavily in the hedge fund industry, is the Omega ratio. On paper, the Omega ratio should be better than both the Sortino and Calmar ratios at measuring risk vs. return, as it is able to account for the entirety of the risk over return distribution in a single metric. To find it, we need to calculate the probability distributions of a portfolio moving above or below a specific benchmark, and then take the ratio of the two. The higher the ratio, the higher the probability of upside potential over downside potential.

$$Omega = \frac{\int_{R_B}^{b} 1 - F(R_P)dx}{\int_{a}^{R_B} F(R_P)dx} = \frac{\text{Upside Potential}}{\text{Downside Potential}}$$

If this looks complicated, don't worry. It get's simpler in code.

The Code

While writing the code for each of these rewards metrics sounds *really fun,* I have opted to use the `empyrical` library to calculate them instead. Luckily enough, this library just happens to include the three rewards metrics we've defined above. Getting a ratio at each time step is as simple as providing the list of returns and benchmark returns for a time period to the corresponding Empyrical function.

```
import numpy as np
from empyrical import sortino_ratio, calmar_ratio, omega_ratio

def _reward(self):
      length = min(self.current_step, self.reward_len)
      returns = np.diff(self.net_worths)[-length:]

      if self.reward_func == 'sortino':
          reward = sortino_ratio(returns)
      elif self.reward_func == 'calmar':
          reward = calmar_ratio(returns)
      elif self.reward_func == 'omega':
          reward = omega_ratio(returns)
      else
          reward = np.mean(returns)

      return reward if abs(reward) != inf and not np.isnan(reward) else 0
```

Here we set the reward at each time step based on our pre-defined reward function

Now that we've decided how to measure a successful trading strategy, it's time to figure out which of these metrics produces the most appealing results. Let's plug each of these reward functions into Optuna and use good old Bayesian optimisation to find the best strategy for our data set.

The Toolset

Any great technician needs a great toolset. Instead of re-inventing the wheel, we are going to take advantage of the pain and suffering of the programmers that have come before us. For today's job, our most important tool is going to be the `optuna` library, which implements Bayesian optimization using Tree-structured Parzen Estimators (TPEs). TPEs are parallelizable, which allows us to take advantage of our GPU, dramatically decreasing our overall search time. In a nutshell,

Bayesian optimization is a technique for efficiently searching a hyperspace to find the set of parameters that maximize a given objective function.

In simpler terms, Bayesian optimization is an efficient method for improving any black box model. It works by modeling the objective function you want to optimize using a surrogate function, or a distribution of surrogate functions. That distribution improves over time as the algorithm explores the hyperspace and zones in on the areas that produce the most value.

How does this apply to our Bitcoin trading bots? Essentially, we can use this technique to find the set of hyper-parameters that make our model the most profitable. We are searching for a needle in a haystack and Bayesian optimization is our magnet. Let's get started.

Implementing Optuna

Optimizing hyper-parameters with Optuna is fairly simple. First, we'll need to create an `optuna` study, which is the parent container for all of our hyper-parameter trials. A *trial* contains a specific configuration of hyper-parameters and its resulting cost from the objective function. We can then call `study.optimize()` and pass in our objective function, and Optuna will use Bayesian optimization to find the configuration of hyper-parameters that produces the lowest cost.

```
import optuna

def optimize(n_trials = 5000, n_jobs = 4):
    study = optuna.create_study(study_name='optimize_profit', storage='sqlite:///params.db', load_if_exists=True)
    study.optimize(objective_fn, n_trials=n_trials, n_jobs=n_jobs)
```

In this case, our objective function consists of training and testing our PPO2 model on our Bitcoin trading environment. The cost we return from our function is the average reward over the testing period, negated. We need to negate the average reward, because Optuna interprets lower return value as better trials. The `optimize` function provides a trial object to our objective function, which we then use to specify each variable to optimize.

```
def objective_fn(trial):
    env_params = optimize_envs(trial)
    agent_params = optimize_ppo2(trial)

    train_env, validation_env = initialize_envs(**env_params)
    model = PPO2(MlpLstmPolicy, train_env, **agent_params)

    model.learn(len(train_env.df))

    rewards, done = [], False

    obs = validation_env.reset()
    for i in range(len(validation_env.df)):
        action, _ = model.predict(obs)
        obs, reward, done, _ = validation_env.step(action)
        rewards += reward

    return -np.mean(rewards)
```

The `optimize_ppo2()` and `optimize_envs()` methods take in a trial object and return a dictionary of parameters to test. The search space for each of our variables is defined by the specific `suggest` function we call on the trial, and the parameters we pass in to that function.

For example, `trial.suggest_loguniform('n_steps', 16, 2048)` will suggest a new float between 16–2048 in a logarithmic manner (16, 32, 64, ..., 1024, 2048). Further, `trial.suggest_uniform('cliprange', 0.1, 0.4)` will suggest floats in

a simple, additive manner (0.1, 0.2, 0.3, 0.4). We don't use it here, but Optuna also provides a method for suggesting categorical variables: `suggest_categorical('categorical', ['option_one', 'option_two'])`.

```
def optimize_ppo2(trial):
    return {
        'n_steps': int(trial.suggest_loguniform('n_steps', 16, 2048)),
        'gamma': trial.suggest_loguniform('gamma', 0.9, 0.9999),
        'learning_rate': trial.suggest_loguniform('learning_rate', 1e-5, 1.),
        'ent_coef': trial.suggest_loguniform('ent_coef', 1e-8, 1e-1),
        'cliprange': trial.suggest_uniform('cliprange', 0.1, 0.4),
        'noptepochs': int(trial.suggest_loguniform('noptepochs', 1, 48)),
        'lam': trial.suggest_uniform('lam', 0.8, 1.)
    }
```

```
def optimize_envs(trial):
    return {
        'reward_len': int(trial.suggest_loguniform('reward_len', 1, 200)),
        'forecast_len': int(trial.suggest_loguniform('forecast_len', 1, 200)),
        'confidence_interval': trial.suggest_uniform('confidence_interval', 0.7, 0.99),
    }
```

Later, after running our optimization function overnight with a decent CPU/GPU combination, we can load up the study from the sqlite database we told Optuna to create. The study keeps track of the best trial from its tests, which we can use to grab the best set of hyper-parameters for our environment.

```
study = optuna.load_study(study_name='optimize_profit', storage='sqlite:///params.db')
params = study.best_trial.params

env_params = {
    'reward_len': int(params['reward_len']),
    'forecast_len': int(params['forecast_len']),
    'confidence_interval': params['confidence_interval']
}

train_env = DummyVecEnv([lambda: BitcoinTradingEnv(train_df, **env_params)])

model_params = {
    'n_steps': int(params['n_steps']),
    'gamma': params['gamma'],
    'learning_rate': params['learning_rate'],
    'ent_coef': params['ent_coef'],
    'cliprange': params['cliprange'],
    'noptepochs': int(params['noptepochs']),
    'lam': params['lam']
}

model = PPO2(MlpLstmPolicy, train_env, **model_params)
```

We've revamped our model, improved our feature set, and optimized all of our hyper-parameters. Now it's time to see how our agents do with their new reward mechanisms. I have trained an agent to optimize each of our four return metrics: simple profit, the Sortino ratio, the Calmar ratio, and the Omega ratio. Let's run each of these optimized agents on a test environment, which is initialized with price data they've not been trained on, and see profitable they are.

Benchmarking

Before we look at the results, we need to know what a successful trading strategy looks like. For this treason, we are going to benchmark against a couple common, yet effective strategies for trading Bitcoin profitably. Believe it or not, one of the most effective strategies for trading BTC over the last ten years has been to simply buy and hold. The other two strategies we will be testing use very simple, yet effective technical analysis to create buy and sell signals.

1. Buy and hold

The idea is to buy as much as possible and Hold On for Dear Life (BHODL). While this strategy is not particularly complex, it has seen very high success rates in the past.

2. RSI divergence

When consecutive closing price continues to rise as the RSI continues to drop, a negative trend reversal (sell) is signaled. A positive trend reversal (buy) is signaled when closing price consecutively drops as the RSI consecutively rises.

3. Simple Moving Average (SMA) Crossover

When the longer-term SMA crosses above the shorter-term SMA, a negative trend reversal (sell) is signaled. A positive trend reversal (buy) is signaled when the shorter-term SMA crosses above the longer-term SMA.

The purpose of testing against these simple benchmarks is to prove that our RL agents are actually creating alpha over the market. If we can't beat these simple benchmarks, then we are wasting countless hours of development time and GPU cycles, just to make a cool science project. Let's prove that this is not the case.

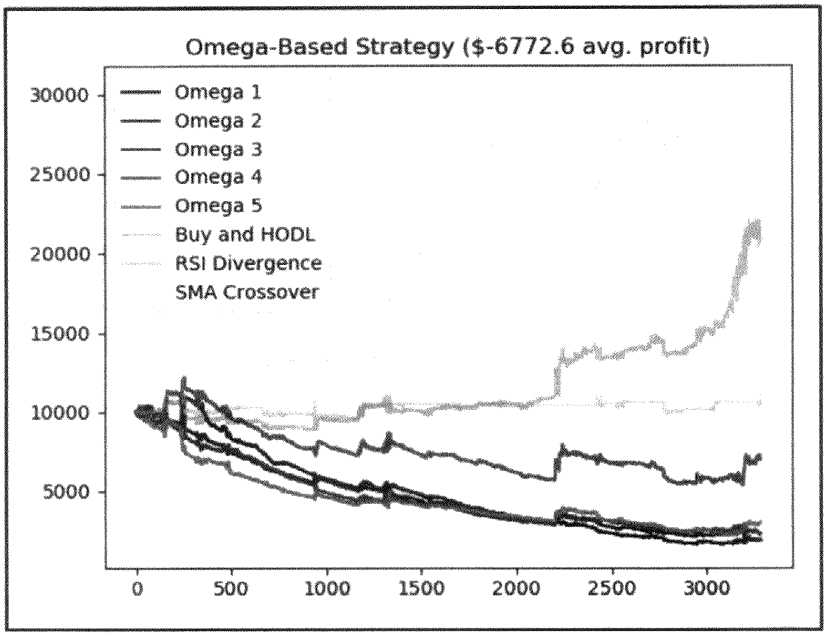
The Results

I must preface this section by stating that the positive profits in this section are the direct result of incorrect code. Due to the way dates were being sorted at the time, the agent was able to see the price 12 hours in advance at all times, an obvious form of look-ahead bias. This has since been fixed, though the time has yet to be invested to replace each of the result sets below. Please understand that these results are completely invalid and highly unlikely to be reproduced.

That being said, there is still a large amount of research that went into this article and the purpose was never to make massive amounts of money, rather to see what was possible with the current state-of-the-art reinforcement learning and optimization techniques. So in attempt to keep this article as close to the original as possible, I will leave the old (invalid) results here until I have the time to replace them with new, valid results.

The agents were trained on the first 80% of the data set (hourly OHCLV data from CryptoDataDownload), and tested on the final 20% to see how the strategies generalize to fresh data. This simple cross validation is enough for what we need, as when we eventually release these algorithms into the wild, we can train on the entire data set and treat new incoming data as the new test set.

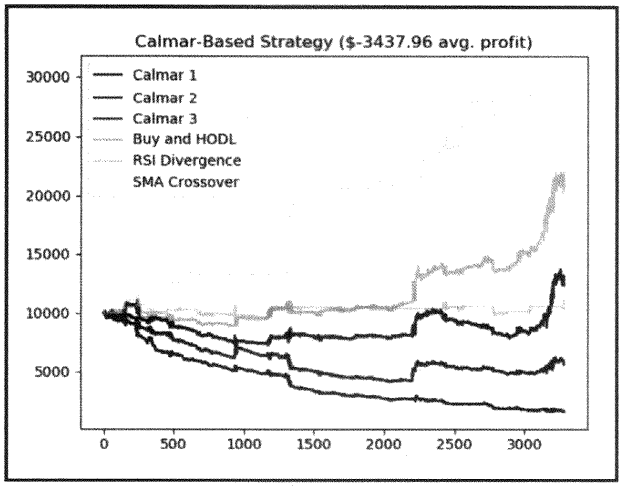
Let's quickly move through the losers so we can get to the good stuff. First, we've got the Omega strategy, which ends up being fairly useless trading against our data set.



Average net worth of Omega-based agents over 3500 hours of trading

Watching this agent trade, it was clear this reward mechanism produces strategies that over-trade and are not capable of capitalizing on market opportunities.

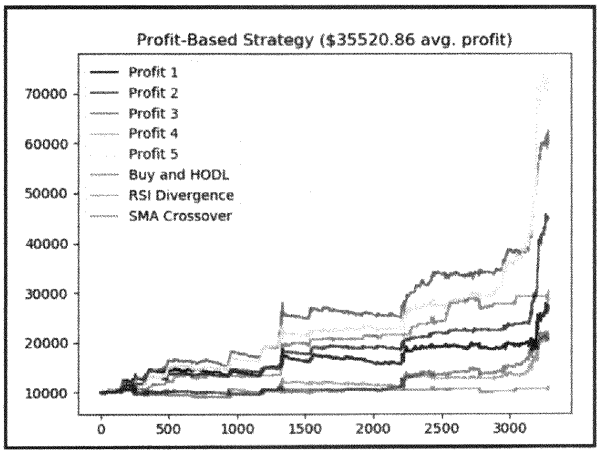
The Calmar-based strategies came in with a small improvement over the Omega-based strategies, but ultimately the results were very similar. It's starting to look like we've put in a ton of time and effort, just to make things worse...



Average net worth of Calmar-based agents over 3500 hours of trading

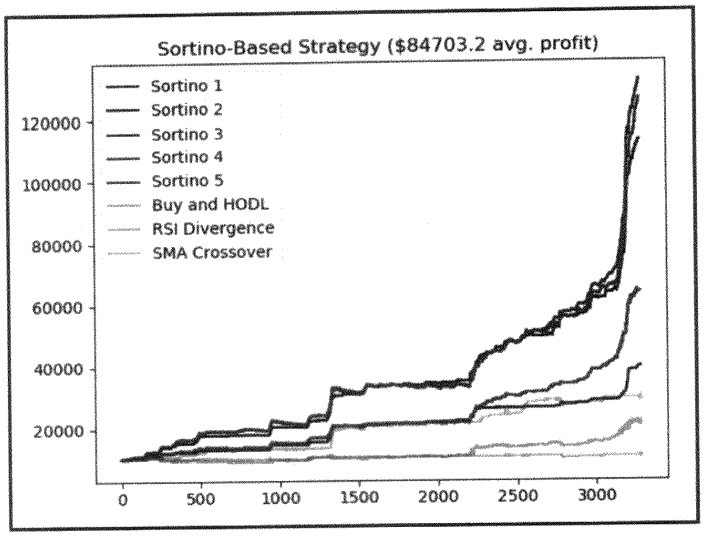
Remember our old friend, simple incremental profit? While this reward mechanism didn't prove to be too successful in our last article, all the modifications and optimizations we've done seem to have massively improved the success of the agents.

The average profit is just over **350%** of the initial account balance, over our four month test period. If you are unaware of average market returns, these kind of results would be absolutely insane. Surely this is the best we can do with reinforcement learning... right?



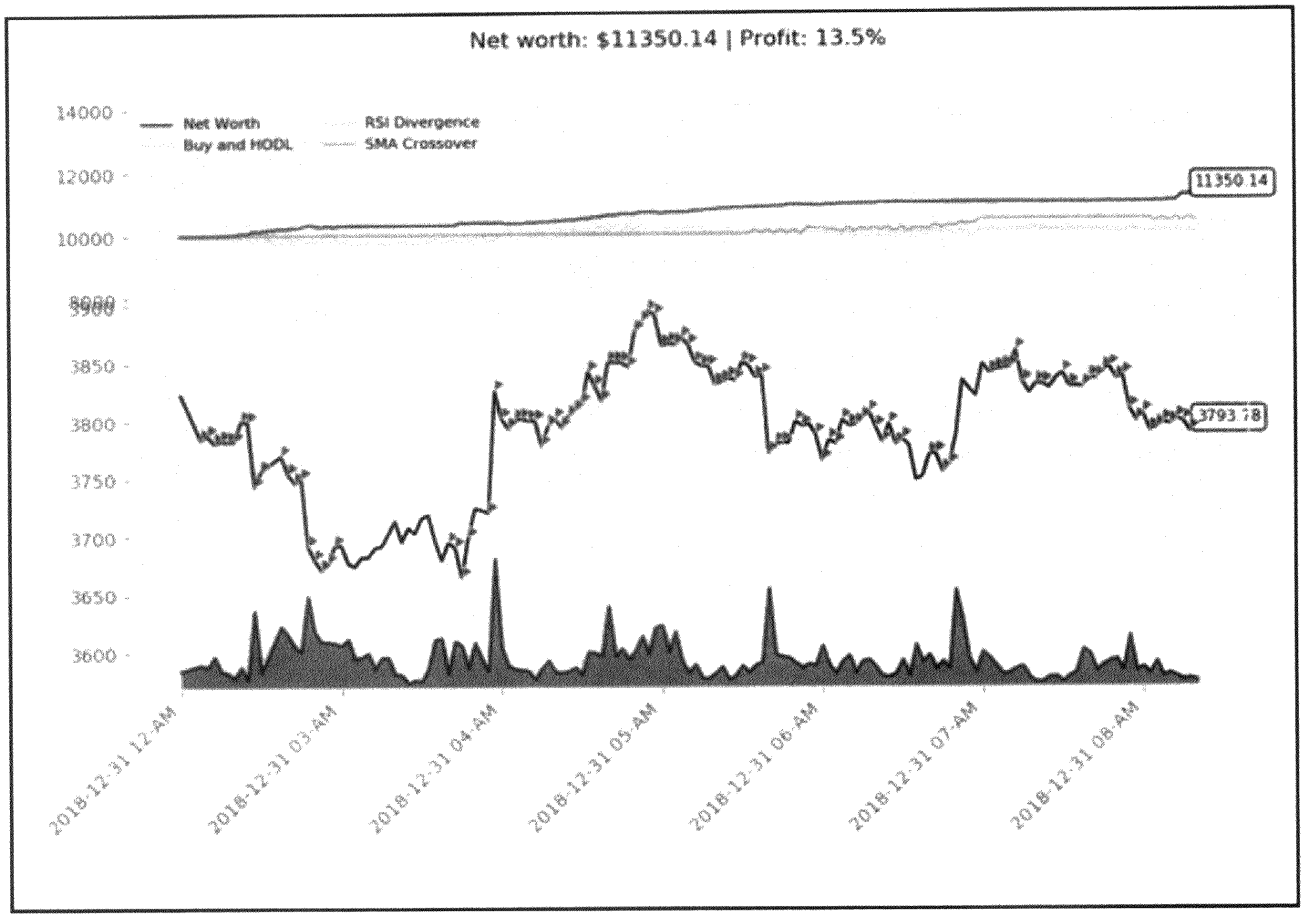
Average net worth of Profit-based agents over 3500 hours of trading

Wrong. The average profit produced by agents rewarded by the Sortino ratio was nearly **850%**. When I saw the success of these strategies, I had to quickly check to make sure there were no bugs. [**Editor's note**: **Brace yourself for the irony of the following sentence.**] After a thorough inspection, it is clear that the code is bug free [**it is not**] and these agents are just very good at trading Bitcoin [**they might not be**].



Average net worth of Sortino-based agents over 3500 hours of trading

Instead of over-trading and under-capitalizing, these agents seem to understand the importance of buying low and selling high, while minimizing the risk of holding BTC. Regardless of what specific strategy the agents have learned, our trading bots have clearly learned to trade Bitcoin profitably. If you don't believe me, see for yourself.



One of the Sortino-based agents trading BTC/USD. Green triangles signal buys, red triangles signal sells.

Now, I am no fool. I understand that the success in these tests may not [**read: will not**] generalize to live trading. That being said, these results are far more impressive than any algorithmic trading strategies I've seen to date (this should have been the first clue that something was wrong...). It is truly amazing considering these agents were given no prior knowledge of how markets worked or how to trade profitably, and instead learned to be massively successful through trial and error alone (along with some good old look-ahead bias). Lots, and lots, of trial and error.

Conclusion

In this article, we've optimized our reinforcement learning agents to make even better decisions while trading Bitcoin, and therefore, make a ton more money! It took quite a bit of work, but we've managed to accomplish it by doing the following:

- Upgrade the existing model to use a recurrent, LSTM policy network with stationary data
- Engineer 40+ new features for the agent to learn from using domain-specific technical and statistical analysis
- Improve the agent's reward system to account for risk, instead of simply profit
- Fine tuned the model's hyper-parameters using Bayesian optimization
- Benchmarked against common trading strategies to ensure the bots are always beating the market

A highly profitable trading bot is great, in theory. However, I've received quite a bit of feedback claiming these agents are simply learning to fit a curve, and therefore, would never be profitable trading on live data. While our method of training/testing on separate data sets should address this issue, it is true that our model *could* be overfitting to this data set and *might* not generalize to new data very well. That being said, I've got a feeling these agents are learning quite a bit more than simple curve fitting, and as a result, will be able to profit in live trading situations.