Stock Propagation

Liam Sefton, Giacomo Radaelli, Dalton Kohl

Yfinance API

- Open-source tool that uses Yahoo's publicly available APIs
- Allows users to download market data from Yahoo! Finance
- Ticker module
    - ticker.get_financials()
    - ticker.get_balance_sheet()
    - ticker.get_cashflow()
- yfinance.download()

Data Mining

- Used stock data from S&P 500
- Selected several attributes from balance sheet, cashflow, and financial statement of each stock
- Extracted SPY ETF data
  - tracks the S&P 500
- Created CSV file containing extracted data for each stock
  - Three samples per stock (each sample represents different year)

Show: **Income Statement** | Balance Sheet | Cash Flow

Income Statement All numbers in thousands

Get access to 40+ years of historical data with Yahoo Finance Plus Essen

Breakdown	TTM	3/31/2021
Total Revenue	10,754	9,168
Cost of Revenue	7,476	6,461
Gross Profit	3,278	2,707
Operating Expense	3,371	3,323
Operating Income	-93	-616

Problems with Yfinance API

- Extract close data from next day if there was no close data for current day
- Extremely slow data extraction
    - Approximately 6 samples per minute (2 stocks)
    - ~4.17 hours to complete

Cleaning Dirty Data

- Out of the initial 1500 samples, only ~900 had clean data
- The majority of lines are either too long, too short or contain numbers with multiple decimal points
- Even removing the lines with mistakes described above, lines containing “nan” values can slip through


```
754384629153,0.3475814931987807,0.
553199947,-0.08648162571462736,0.1
34277740872173,-0.0679665951061211
342784120109,-0.0651450183399034,0
77331268648261,nan,0.5884039290635
1562349735534541,nan,0.57621796351
732180293501048,-0.002402166317260
65975876,-0.22700278379214353,0.57
5718079,-0.2221262437214064,0.5817
382541167,-0.20245200258862445,0.5
26136,0.09595366740369768,0.500658
324117149,0.0862578764605227,0.431
```

Data Preprocessing

- Using the raw information from companies of vastly different sizes is not a good idea
- Rescaling the data is a possible option, but there is some degree of information loss with basic rescaling
- Representing the data in a way that keeps their values close together while still preserving the full information would be ideal

Data Preprocessing

- Our solution was to change the data we were mining for, some examples:
  - Net assets -> (Net assets)/(Total assets)
  - Cash -> (Cash)/(Total assets)
  - Current liabilities -> (Current liabilities)/(Total liabilities)
  - Gross profit -> (Gross profit)/(Total revenue)
- Pricing data was also changed
  - Price 1yr before “current” -> % increase to “current”
  - SPY price 1yr before “current” -> % increase in SPY price to “current”
  - Price 1yr after “current” -> % increase from “current” to 1yr after “current”
- This way the information is preserved while also changing the range in values from something like [-1000000, 1000000] to something about [-1, 2]

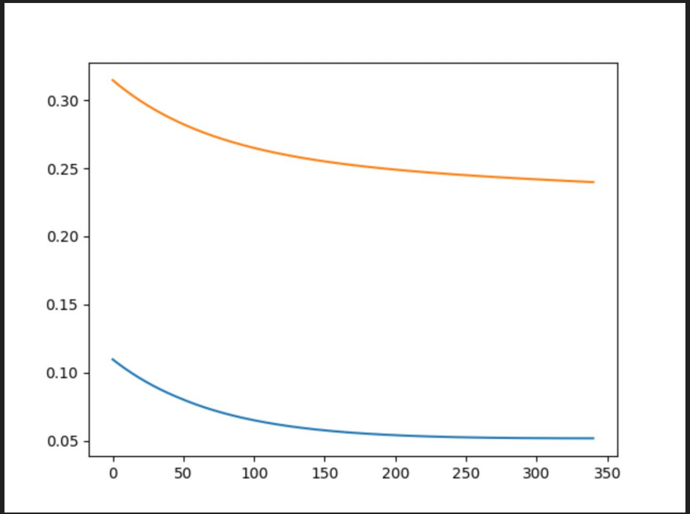
Network Architecture (using pytorch)

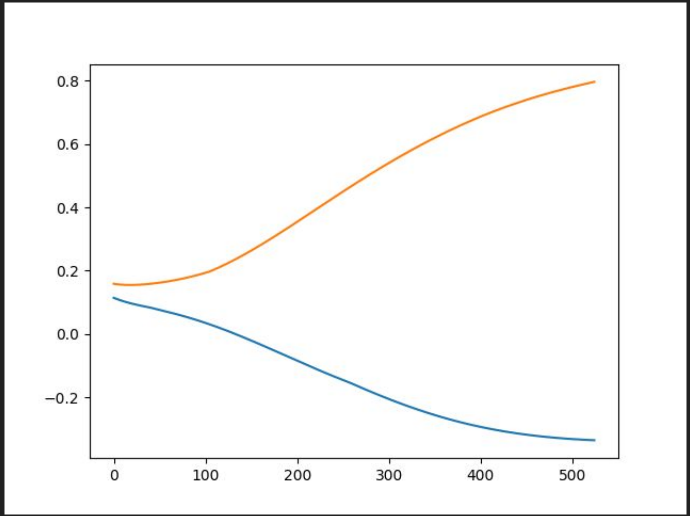
```
class StockPropagation(nn.Module):
    def __init__(self):
        super(StockPropagation, self).__init__()
        #This creates the layers
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(19, 15),
            nn.Tanh(),
            nn.Linear(15, 10),
            nn.Tanh(),
            nn.Linear(10, 5),
            nn.Tanh(),
            nn.Linear(5, 1)
        )

    def forward(self, x):
        #forward function must be defined, so we give it the layers we created above
        return self.linear_relu_stack(x)
```

Network Architecture

- For the learning algorithm (called the optimizer in pytorch), we settled on using Adam, a version of Stochastic Gradient Descent that stores past gradients and adapts the learning rate for each parameter individually.
- (We don't really understand the math, we just did a lot of experiments)
- Top right figure is the output variance with SGD, bottom right is from Adam

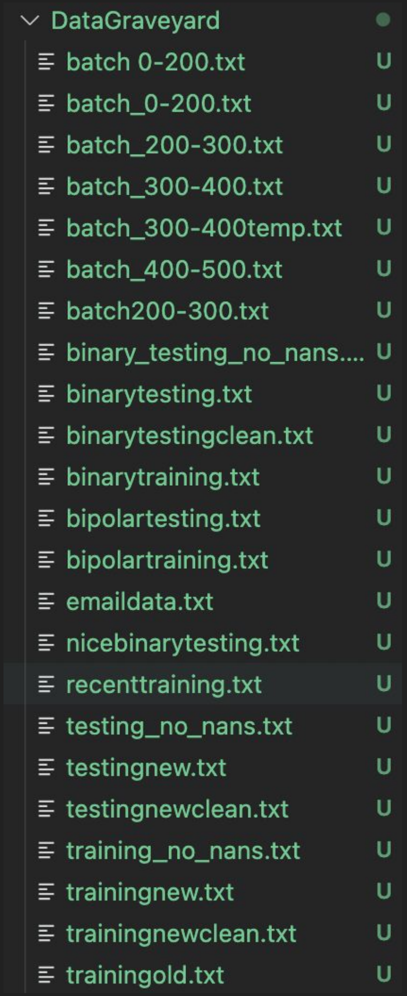
```
optimizer = optim.Adam(model.parameters(), lr=.000005)
```



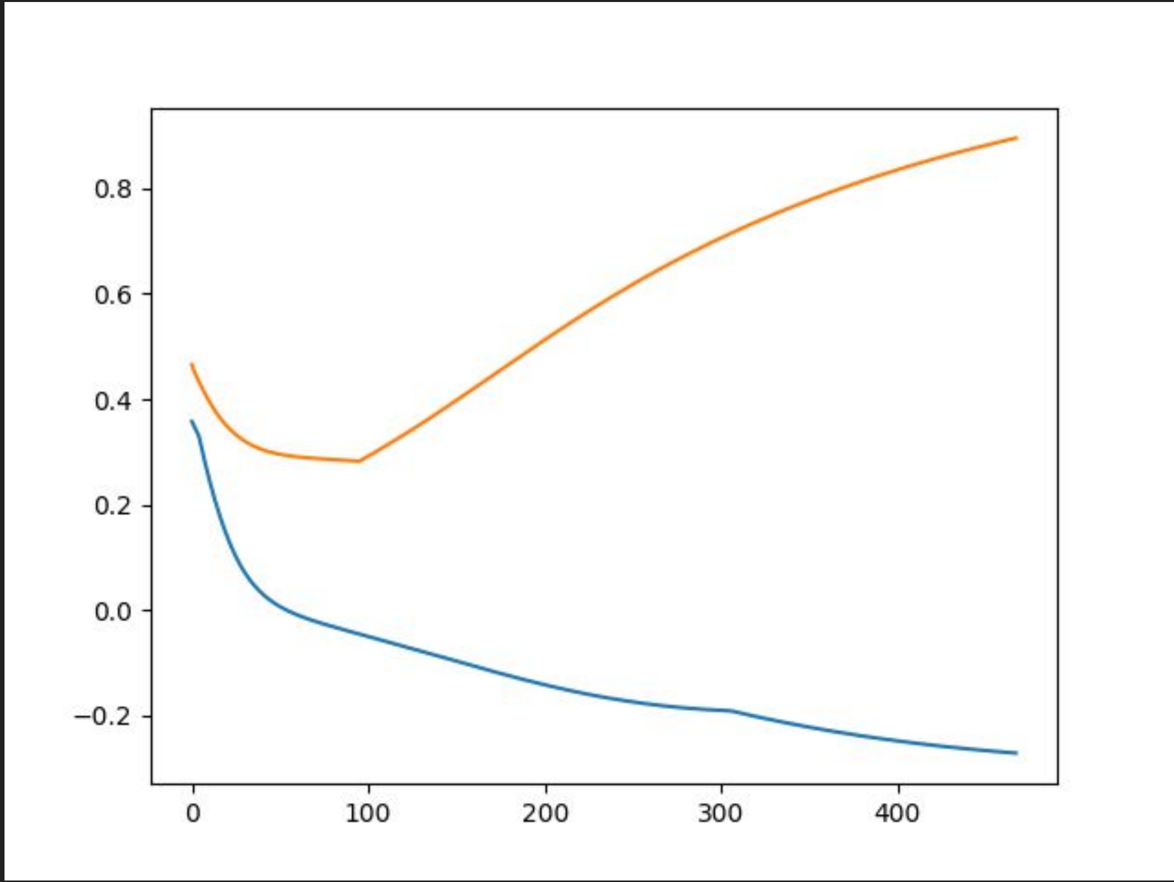


Network Architecture - Activation functions

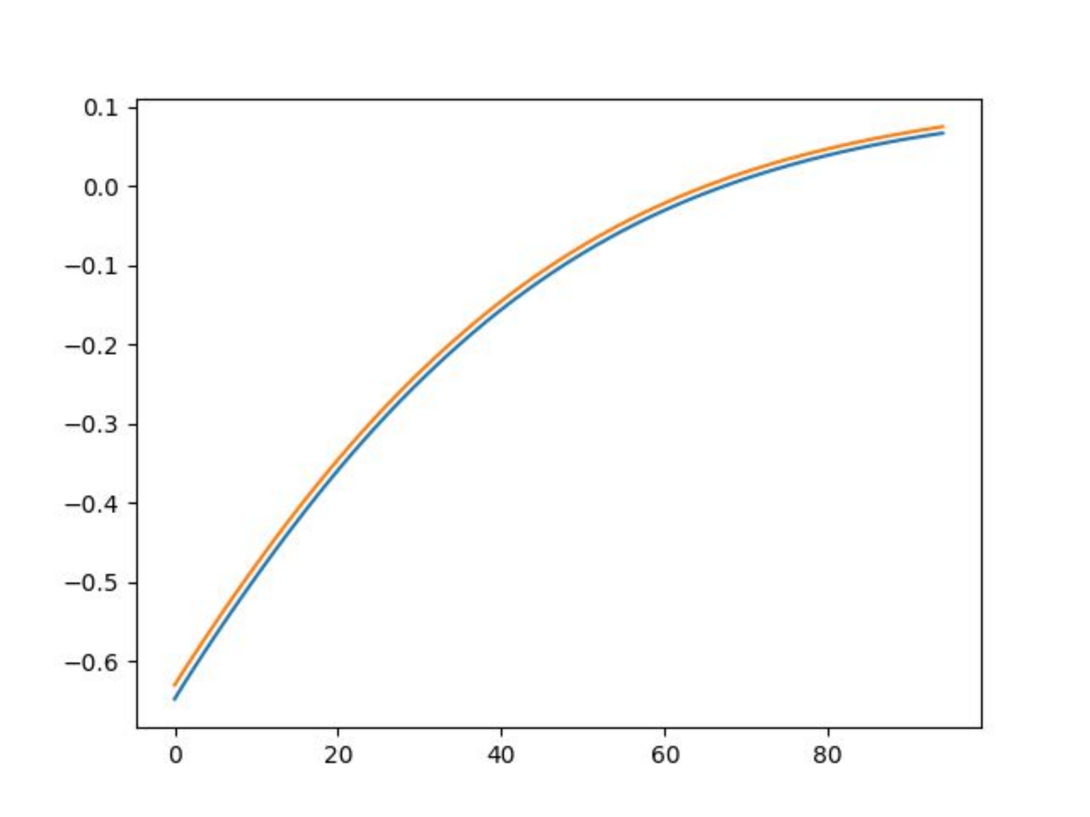
- Sigmoid results were around ~40%-50% (bad)
- ReLU results were around ~45%-55%
- Tanh+ReLU results were around ~55%-65%
- Keep in mind that if the net output all buy signals it would get 45% correct (the proportion of samples in the testing set that have the buy signal as their target value)
- Nets with pure ReLU or Sigmoid would cause output clumping, shown on the next couple of slides



Tanh (2 hidden layers, layer count: [19, 15, 10, 5], Adam)



Sigmoid (same architecture)



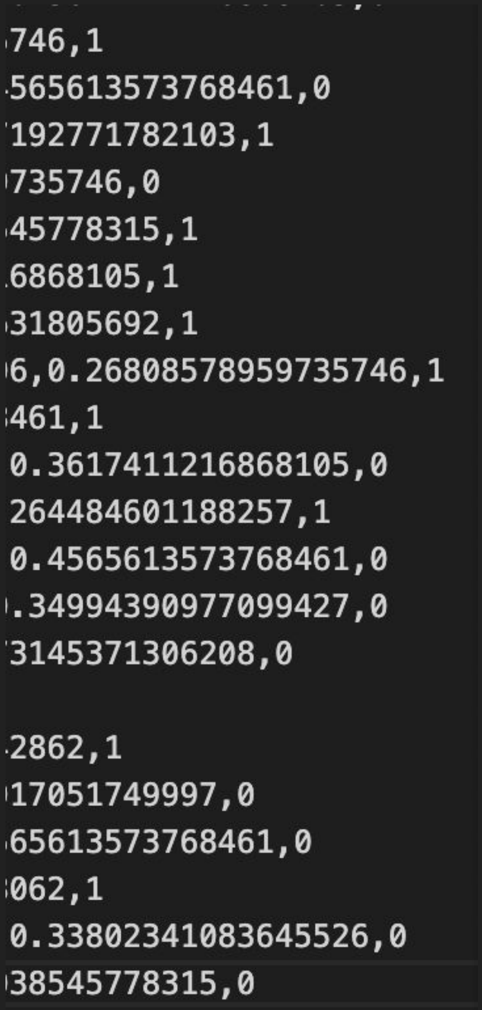
Training Methodology

- For the training and validation sets, the target value represents the % increase over 1yr
- The loss function (we chose to use mean squared errors) compares the predicted % increase with the target value
- We graph the change in the loss function by taking the average of |target - output| for each epoch
- The training ends when either the graphed loss converges (behaves asymptotically) or the graphed loss on the validation set increases (overfitting)

03996,0.12846025758969645
3959735746,0.19990049063814058
461,-0.0826433500189548
782103,0.44813829245480796
9735746,0.13448607366928095
58461,-0.018951157751754134
0118500557178,-0.04593845281042486
368105,0.11592842854266855
09404047,-0.4475540798684363
3467,-0.10699593718330183
7411216868105,-0.27738151846568393
.28251121076233177
3573768461,0.0827574922572097
1782103,0.3324271267309674
5,-0.0814404802526049
315,0.20078015082496356
05,0.20391878940496422
592,0.19964464565608583
5808578959735746,0.4001293391232213
.3516283944926257
7411216868105,-0.42749734833189146
4601188257,0.16208964233345377
5613573768461,-0.06889972732243915
4390977099427,0.022212881296708988
71306208,0.032893532189304374

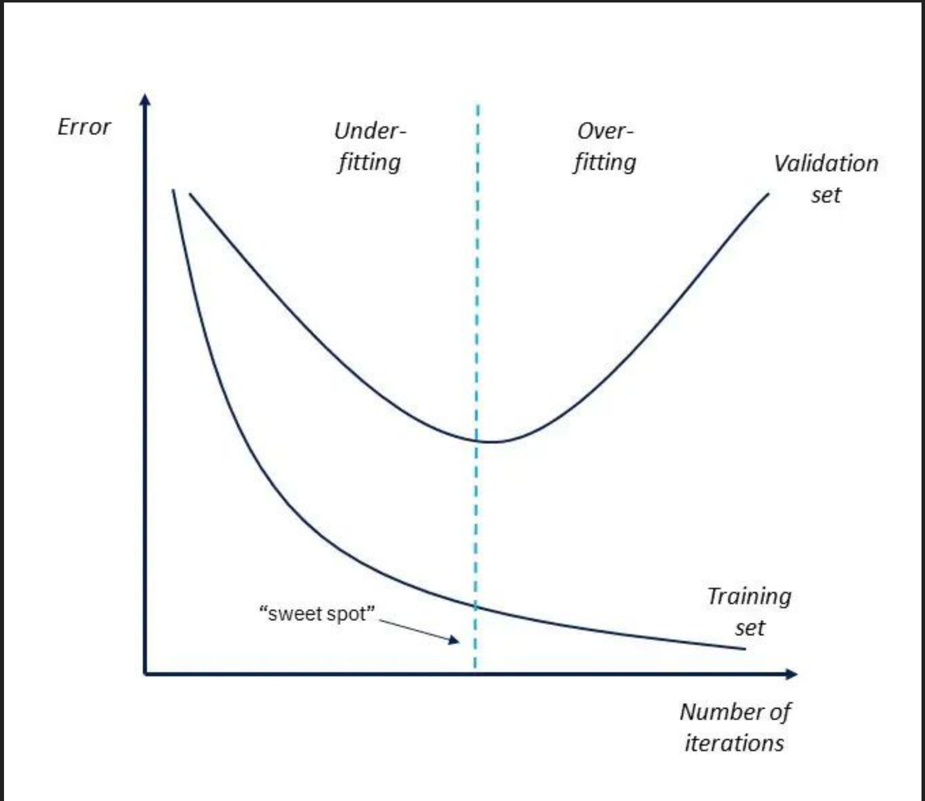
Testing Methodology

- In the testing set we actually change the target value:
  - Any target value greater than .1 -> 1
  - Any target value less than .1 -> 0
- We also created “nice” testing sets where we made these changes:
  - Nice testing set 1 (for stocks that at least increase 5%)
    - Any target value greater than .05 -> 1
    - Any target value less than .05 -> 0
  - Very nice testing set 2 (for stocks that at least don’t decrease)
    - Any target value greater than 0 -> 1
    - Any target value less than or equal to 0 -> 0



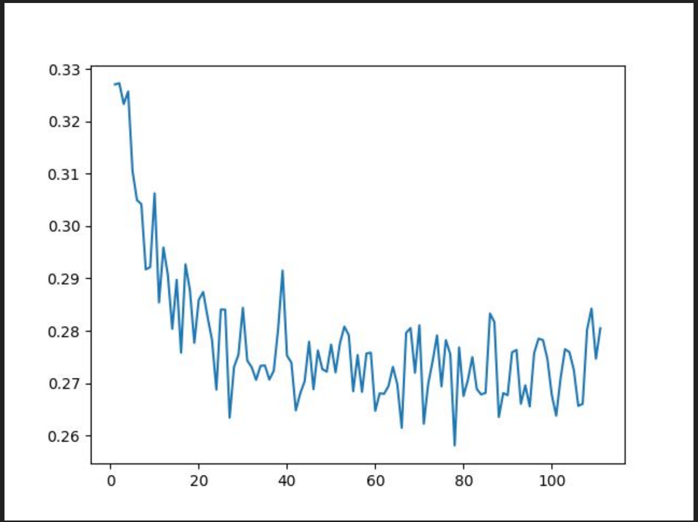
Overfitting prevention

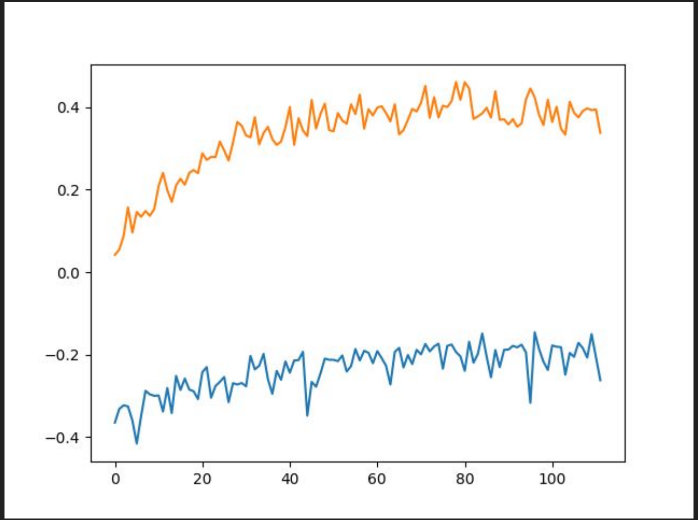
- Our main overfitting prevention comes from our validation set
- We end training when the error on the validation set increases
- Our validation set is just the testing set without binary outputs



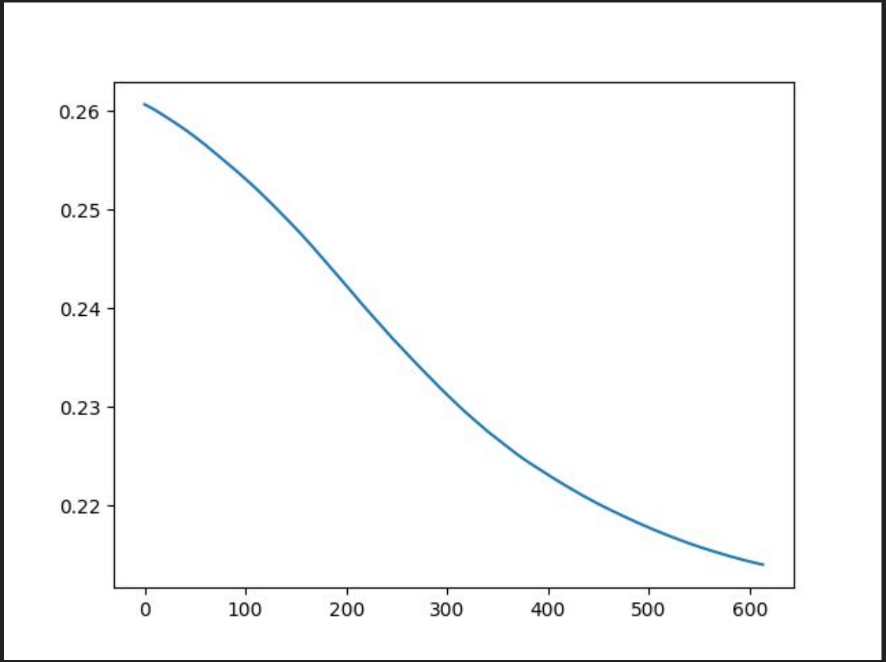
Overfitting prevention

- Another common method to prevent overfitting is dropout, we also tried this approach, but found much worse overall results
- The average accuracy of the net when dropout is applied in training is around 50%
- Top right graph is validation set error by epoch
- Bottom right graph is variation spread by epoch





Results



We found the most success with the following:

- Architecture: 2 hidden layers with 15 neurons in the first hidden layer and 10 in the second hidden layer
- Loss function: Mean Squared Errors
- Learning algorithm: Adam + backpropagation
- Activation function: Tanh
- Learning rate: 0.000005

Results

100 run result (13 hours of running):

- Sample mean: 0.7295137663611784
- Sample standard deviation: 0.015129025774311732
- 99% confidence interval: [0.7291, 0.7299]
- Interpretation: Our net can predict 10% stock price increases over a 1 year period with ~73% accuracy (~80% accuracy for very nice testing set and ~75% for the nice testing set)
- Keep in mind, the proportion of the testing set that has 10% increases is 45%, so this is not a result of the net outputting all buy signals or outputting buy signals randomly

Possible extensions

Ways the accuracy could be improved is with:

- Data changes
  - More data
  - Different inputs
  - Higher quality data (bloomberg terminal)
- Full exploration of pytorch optimizers, loss functions and activation functions via automation
- More computing power for longer and faster training
  - Currently an 800 epoch run takes more than 10 minutes, so testing changes becomes very slow, especially since a few runs must be conducted for a reasonable sample size