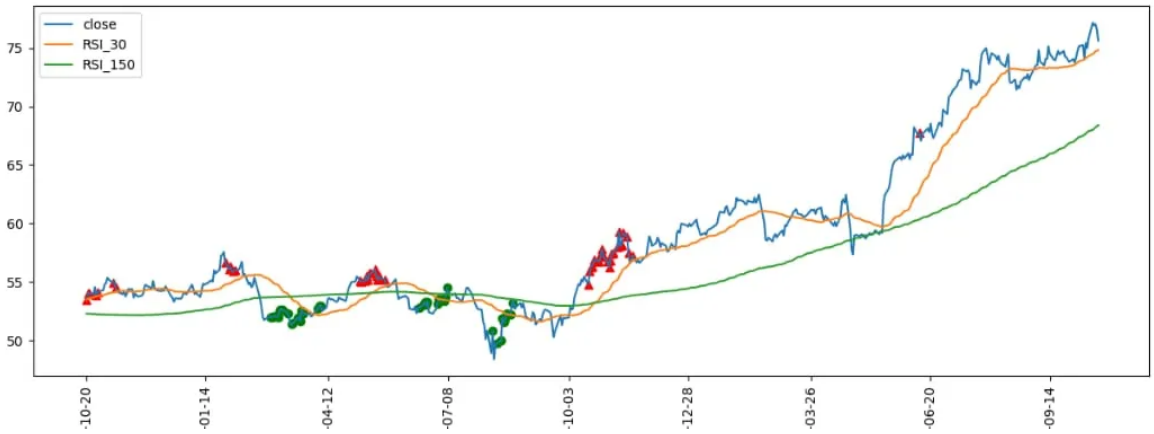


Stock Buy/Sell Prediction Using Convolutional Neural Network

Inspired from Research Paper titled 'Algorithmic Financial Trading with Deep Convolutional Neural Networks: Time Series to Image Conversion'

Asutosh Nayak · Follow
Published in Towards Data Science
11 min read · Jan 19, 2020

This project is loosely based on a research paper titled "*Algorithmic Financial Trading with Deep Convolutional Neural Networks: Time Series to Image Conversion Approach*". I say 'loosely' because although I have borrowed the core idea from the paper, there are some things that I have done (or had to do) different as we will see later. The link I have shared above is a preprint of the paper. The paid/main paper may have more details. This paper was suggested by one of the readers of my previous article on stock price prediction and it immediately caught my attention. **Here is the link to the Github repo and main training notebook on Kaggle.**

There is one thing I would like the readers to know — I am not here to claim that I have a ready to use trading model (although I am exploring this method further for my personal use). The idea of converting a conventional tabular or time-series data to image, and training a classification model on it, just seemed too exciting to resist from trying it out and sharing it with the community. Like my previous article this is an account of my experience with the project.

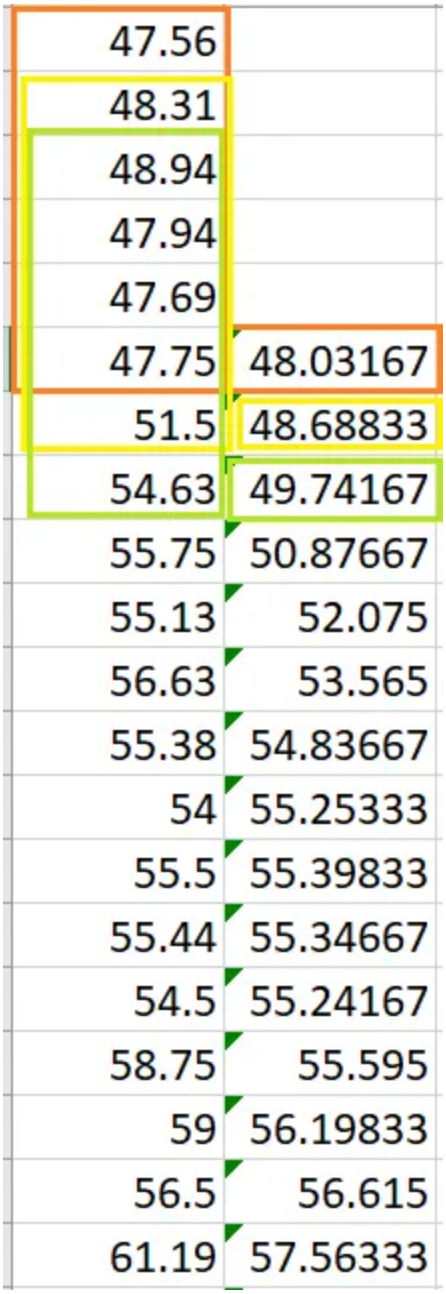
1. What does the research paper say?

In this section I will explain the idea presented in the paper. I will discuss the code and implementation in the next section.

> *The idea is fairly simple: Calculate 15 technical indicators with 15 different period lengths (explained below) for each day in your trading data. Then convert the 225 (15*15) new features into 15x15 images. Label the data as buy/sell/hold based the algorithm provided in the paper. Then train a Convolutional Neural Network like any other image classification problem.*

Feature Engineering: If you are not aware of what a technical indicator is, I would suggest you check the link provided above. I would explain the concept of technical indicators and time period with a Simple Moving Average (SMA) since it's simpler. This should be enough for you to understand the idea.

A moving average for a list of numbers is like arithmetic average but instead of calculating the average of all the numbers, we calculate the average of the first 'n' numbers (n is referred as window size or time period) and then move (or slide) the window by 1 index, thus excluding the first element and including the n+1 element and calculate their average. This process continues. Here is an example to drive this point home:



47.56	
48.31	
48.94	
47.94	
47.69	
47.75	48.03167
51.5	48.68833
54.63	49.74167
55.75	50.87667
55.13	52.075
56.63	53.565
55.38	54.83667
54	55.25333
55.5	55.39833
55.44	55.34667
54.5	55.24167
58.75	55.595
59	56.19833
56.5	56.615
61.19	57.56333

SMA example on excel sheet

This is an example of SMA on window size of 6. The SMA of first 6 elements is shown in orange. Now consider the first column above as the close price of your chosen stock. Now calculate SMA on close price for 14 other window sizes (7 to 20) concatenated on right side of sma_6. Now you have 15 new features for each row of your dataset. Repeat this process for 14 other technical indicators and drop the null rows.

Some of the indicators used are extensions of SMA. For instance, WMA (Weighted

Moving Average) is the average of previous 'n' days with more weight given to the recent past days. Similarly HMA (Hull Moving Average) is an extension of WMA with following steps:

Calculation

1. Calculate a Weighted Moving Average with period n / 2 and multiply it by 2
2. Calculate a Weighted Moving Average for period n and subtract if from step 1
3. Calculate a Weighted Moving Average with period sqrt(n) using the data from step 2

HMA= WMA(2*WMA(n/2) – WMA(n)),sqrt(n))

Image source: Fidelity.com

Now you have 225 new features. If you reshape these numbers into a 15x15 array, you have an image! (Albeit, at this point, it's a single channel. More on this later). **There is one thing to keep in mind though. While constructing these images we should keep the related technical indicators spatially close.** The intuition is, when training for human face recognition, you would not label a picture as human face if it has one eye below the nose. Related pixels should be close by. I am not posting the code to calculate all the indicators for brevity. You can find them in utils.py file.

Labeling: What's left now is to label this dataset. For that, the authors used following algorithm:

Algorithm 1 Labelling Method

```
1:  procedure LABELLING()
2:      windowSize = 11 days
3:      while(counterRow < numberOfDaysInFile)
4:          counterRow++
5:          If (counterRow > windowSize)
6:              windowBeginIndex = counterRow − windowSize
7:              windowEndIndex = windowBeginIndex + windowSize − 1
8:              windowMiddleIndex = (windowBeginIndex + windowEndIndex)/2
9:              for (i = windowBeginIndex; i <= windowEndIndex; i++)
10:                 number = closePriceList.get(i)
11:                 if(number < min)
12:                   min = number
13:                   minIndex = closePriceList.indexOf(min)
14:                 if(number > max)
15:                   max = number
16:                   maxIndex = closePriceList.indexOf(max)
17:             if(maxIndex == windowMiddleIndex)
18:                 result = "SELL"
19:             elif(minIndex == windowMiddleIndex)
20:                 result = "BUY"
21:             else
22:                 result = "HOLD"
```

Algorithm used to label the dataset as buy/sell/hold

At first glance, it may seem formidable, but all it says is this: use a window of 11 days on close price. If the middle number is maximum within the window, label the middle day as 'sell' or, if the middle number is minimum then label the middle day as 'buy', else label as 'hold'. Slide the window like explained earlier and repeat. **The idea is to buy at troughs and sell at crests for any 11 day window.** The competency of this algorithm is a different matter and I will get into that toward the end.

Training: Authors have used rolling window training, which is similar to the sliding window concept we saw above. If you have stock history data for the year 2000 to 2019 and you decide to train on 5 years data and test on 1 year data then, slice the data for 2000–2004 from dataset for training and 2005 year's data for testing. Train and test your model on this data. Next select 2001–2005 as training data and 2006 as test data. Use the same model to retrain on this data. Repeat until you reach the end.

Computational Performance Evaluation: Authors have provided two types of model evaluations in the paper, computational and financial evaluation. Computational evaluation includes confusion matrix, F1 score, class wise precision etc. Financial evaluation is done by applying the model prediction to real world trading and measure the profit made. I will only discuss the computational evaluation. Financial evaluation can be done by either real world trading or backtesting on held out data, which I may discuss in the future articles.

2. Implementation

As mentioned at the beginning of this article, I have not followed the research paper strictly because it didn't produce expected results. I will mention the differences as and when they come up. But with the changes I made the result was at par with the paper or better in some cases.

The data processing related code can be found in **data_generator.py**

Data Source: I usually get stock data from Alpha Vantage which provides historical stock data for free. I had used it for my previous project as well. Here is how you can download the data.

```
url = "https://www.alphavantage.co/query?
function=TIME_SERIES_DAILY_ADJUSTED&outputsize=full&apikey=api_key&d
atatype=csv&symbol=company_code"
urllib.request.urlretrieve(url, path_to_save)
```

Data looks like this:

A	B	C	D	E	F	G	H	I
timestamp	open	high	low	close	adjusted_close	volume	dividend_amount	split_coefficient
27-10-1999	91.5	91.625	89.688	90.875	29.332	54416600	0	1
28-10-1999	90	90.875	89.313	89.875	29.0092	70570400	0	1
29-10-1999	91.438	94	91.25	92.563	29.8767	79452600	0	1
01-11-1999	93.25	94.188	92.125	92.375	29.8162	53261200	0	1
02-11-1999	92.75	94.5	91.938	92.563	29.8767	46349000	0	1
03-11-1999	92.938	93.5	91.5	92	29.6952	44517000	0	1
04-11-1999	92.313	92.75	90.313	91.75	29.6145	54239400	0	1
05-11-1999	91.813	92.875	90.5	91.563	29.5539	70167400	0	1
08-11-1999	84.8113	90.75	84.375	89.938	29.0294	2.44E+08	0	1
09-11-1999	89.75	89.875	86.438	88.875	28.6865	1.1E+08	0	1
10-11-1999	88.125	89.125	86.438	87.125	28.1216	69385400	0	1
11-11-1999	88.25	90.438	88.25	89.625	28.9286	69274600	0	1
12-11-1999	89.75	90	87.063	89.188	28.7873	49414200	0	1
15-11-1999	88.25	88.5	86.938	87	28.0813	47080400	0	1
16-11-1999	86.938	87.75	85.875	87.313	28.1822	59165200	0	1
17-11-1999	86.438	87.063	85	85	27.4357	66819000	0	1
18-11-1999	84.938	85.813	84.5	84.938	27.4156	64493200	0	1
19-11-1999	84.438	86.563	84.375	86	27.7585	58226000	0	1
22-11-1999	89.625	90.3718	88.438	89.813	28.9891	90596600	0	1
23-11-1999	89.25	91.375	88.375	89.625	28.9286	70787400	0	1
24-11-1999	89.563	92.25	89.5	91.688	29.5943	53771000	0	1
26-11-1999	91.625	93.3699	91	91.125	29.4127	28514200	0	1

Feature Engineering: The first deviation from the paper is the technical indicators I used. I couldn't find library/implementation for some of the indicators that were mentioned in the paper, like PSI. Some indicators were just not clear; for example, PPO is calculated using EMA of period 12 and 26. How can we calculate PPO for different periods? I tried to use most of the indicators mentioned in the paper for which I found open source implementations to avoid any programming errors. I have implemented some indicators like WMA, HMA, etc, although they are slow and need optimization. Since I have to run it only once and save the data, it's not an issue for me. You can use different indicators of your choice though. They have also adjusted the prices (open, high, low etc) with adjust ratio. But I haven't followed this one because I couldn't find any reference on how to do that adjustment. All the functions for constructing these features are in **utils.py** file.

Labeling the data: For this blog, I have used the original labeling algorithm that the authors have used. Here is a direct implementation of it:

The dataset looks like this after feature construction and labeling:

	timestamp	open	high	low	close	adjusted_close	volume	rsi_6	rsi_7	rsi_8	...	eom_19	eom_20	eom_21	eom_22	eom_23	
0	2000-03-01	102.00	105.50	100.06	100.25	67.2395	10807800	10.769231	8.571429	12.876254	...	-23.656988	-23.656988	-23.656988	-23.656988	-23.656988	-2
1	2000-03-02	100.50	105.44	99.50	103.12	69.1645	11192900	29.354583	26.695174	21.606529	...	-16.451501	-16.451501	-16.451501	-16.451501	-16.451501	-
2	2000-03-03	107.25	110.00	106.06	108.00	72.4376	10162800	50.545209	45.207481	42.141929	...	215.554768	215.554768	215.554768	215.554768	215.554768	2
3	2000-03-06	109.94	111.00	101.00	103.06	69.1243	10747400	32.388833	37.235432	34.363373	...	-188.882893	-188.882893	-188.882893	-188.882893	-188.882893	-18
4	2000-03-07	106.00	107.00	101.69	103.00	69.0840	10035100	37.782761	33.681008	38.370550	...	-87.573118	-87.573118	-87.573118	-87.573118	-87.573118	-

5 rows × 450 columns

Normalization: I used MinMaxScaler from Sklearn to normalize the data in the range of [0, 1], although the paper used [-1, 1] range (second deviation). This is just a personal preference.

Feature Selection: After calculating these indicators, grouping them in the image based on their types (momentum, oscillator, etc), and training many CNN architectures, I realized the model just isn't learning enough. Maybe the features weren't good enough. So I decided to go with many other indicators without strictly following the rule of calculating them with different periods. Then I used feature selection technique to chose 225 high-quality features. In fact, I used two feature selection methods f_classif and mutual_info_classif and chose the common features

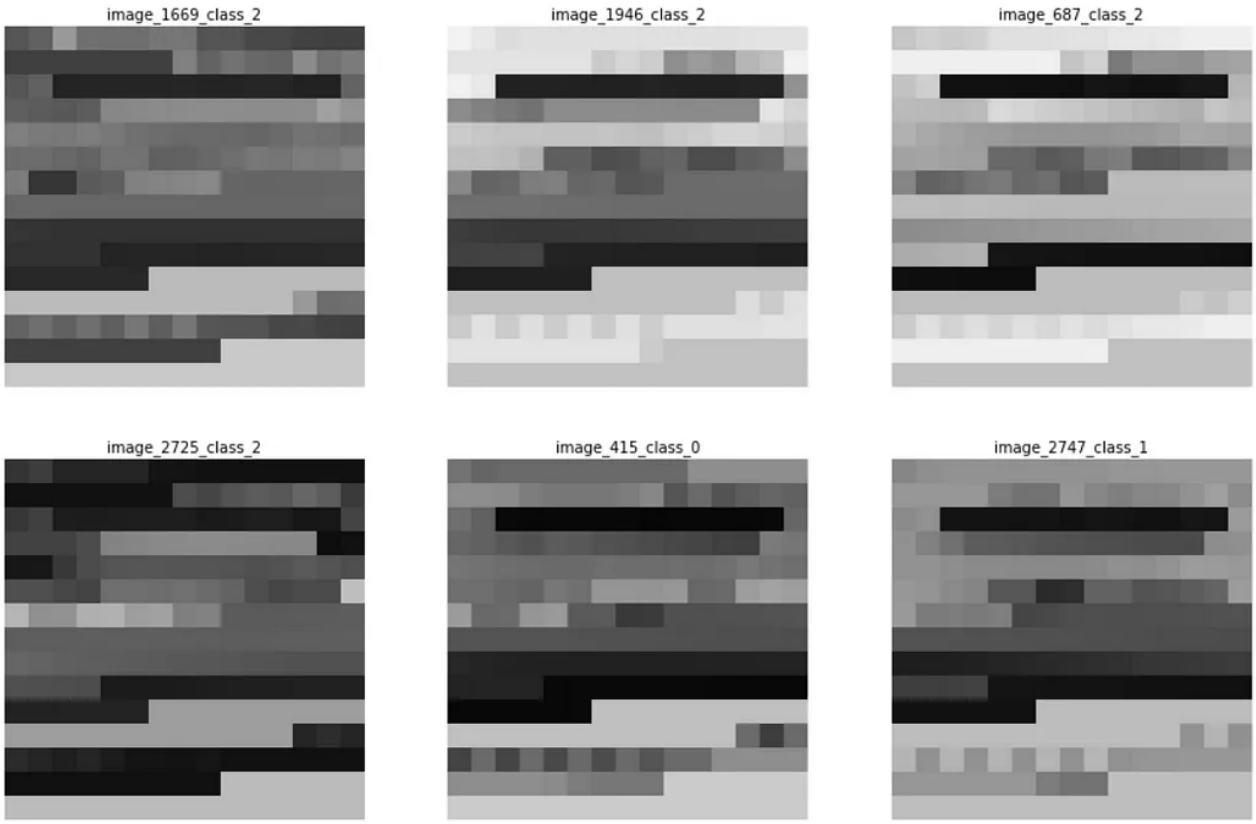
from both of their results. There is no mention of feature selection in the original paper, so third deviation.

At the end I am sorting indices list found intersection of both f_classif and mutual_info_classif. This is to ensure that related features are in close proximity in the image, since I had appended similar type of indicators closely. Feature selection significantly improved the performance of the model.

Reshaping the data as image: As of now we have a tabular data with 225 features. We

need to convert it as images like this:

This is what the images look like:



Training images

Handling Class Imbalance: One more reason why these kinds of problems are tricky to solve is that data is massively imbalanced. Number of instances of 'hold' action will always be *much* greater than buy/sell. In fact the labeling algorithm presented in the paper produces somewhat generous number of buy/sell instances. Any other real world strategy would produce much fewer instances. And to further complicate things, classification of 'hold' event would not be straight forward (more on this towards the end).

```
_labels, _counts = np.unique(y_train, return_counts=True)
print("percentage of class 0 = {}, class 1 = {}".format(_counts[0]/len(y_train) * 100, _counts[1]/len(y_train) * 100))
```

```
percentage of class 0 = 6.363961387200572, class 1 = 6.149445834823024
```

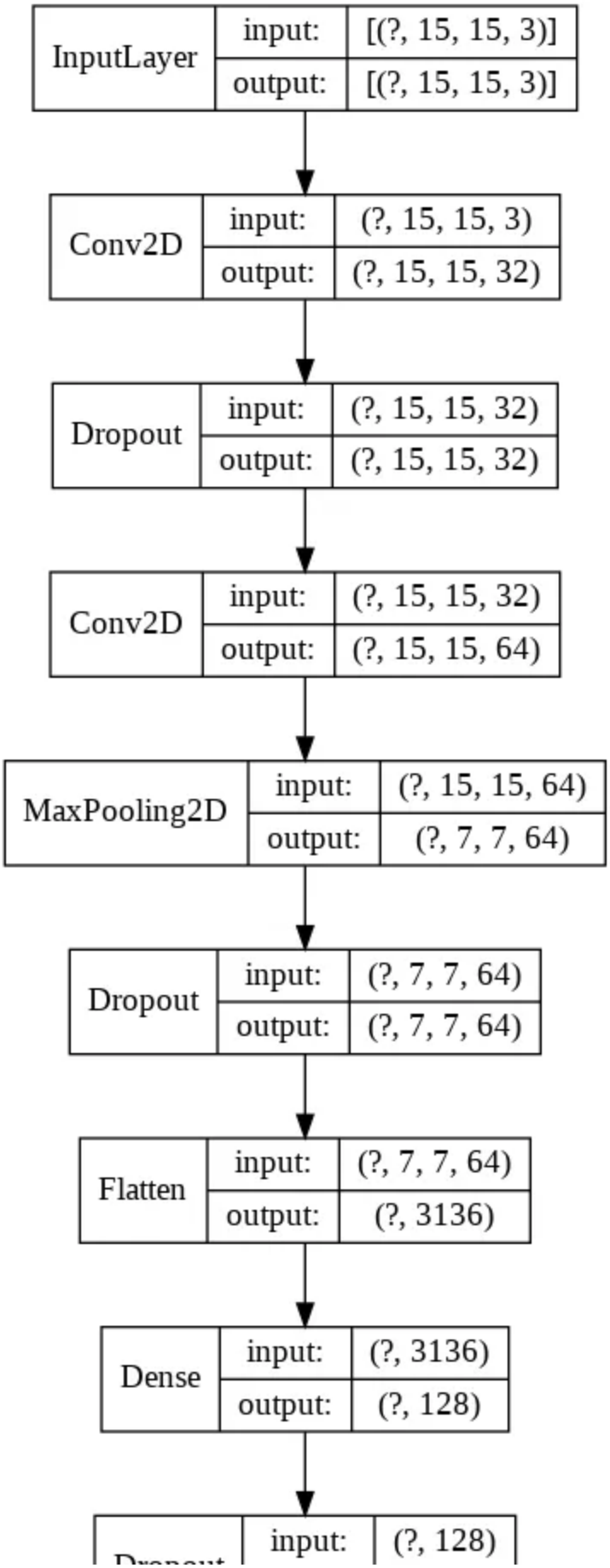
This is really less for model to learn anything significant. The paper mentions only "resampling" as a way of tackling this problem. I tried oversampling, synthetic data generation (SMOTE, ADASYN) but none of them gave any satisfactory result. Finally

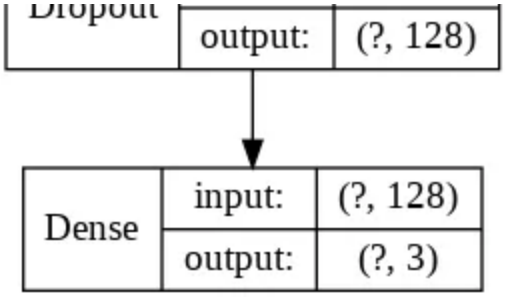
I settled for “sample weights”, wherein you tell the model to pay more attention to some samples (fourth deviation). This comes handy while dealing with class imbalance. Here is how you can calculate sample weight:

This array of sample weights is then passed to Keras ‘fit’ function. You can also look into ‘class_weights’ parameter.

Training: All the training related code can be found in “**stock_keras.ipynb**”. The model architecture mentioned in the paper had some points missing. For example,

they didn't mention the strides they had used. But trying with stride=1 and padding=same, I realized the model was just too big, especially for training on 5 years of data. I didn't have any luck with sliding window training no matter how small a network I used. So I trained with full training data with cross validation (fifth deviation). But I have included the code for sliding/rolling window training in the project (in "train.py" file). So, I used a very similar model with small differences like dropouts etc. This is the model I trained with (I have not tried extensive hyperparameter tuning):


InputLayer
input: [(?, 15, 15, 3)]
output: [(?, 15, 15, 3)]
Conv2D
input: (?, 15, 15, 3)
output: (?, 15, 15, 32)
Dropout
input: (?, 15, 15, 32)
output: (?, 15, 15, 32)
Conv2D
input: (?, 15, 15, 32)
output: (?, 15, 15, 64)
MaxPooling2D
input: (?, 15, 15, 64)
output: (?, 7, 7, 64)
Dropout
input: (?, 7, 7, 64)
output: (?, 7, 7, 64)
Flatten
input: (?, 7, 7, 64)
output: (?, 3136)
Dense
input: (?, 3136)
output: (?, 128)
input: (?, 128)



Keras model training was done with EarlyStopping and ReduceLROnPlateau callbacks like this:

```
import os

best_model_path = os.path.join('.', 'best_model_keras')
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1,
                   patience=100, min_delta=0.0001)

# csv_logger = CSVLogger(os.path.join(OUTPUT_PATH, 'log_training_batch.log'), append=True)

rlp = ReduceLROnPlateau(monitor='val_loss', factor=0.02, patience=20, verbose=1, mode='min',
                        min_delta=0.001, cooldown=1, min_lr=0.0001)
mcp = ModelCheckpoint(best_model_path, monitor='val_f1_metric', verbose=1,
                      save_best_only=True, save_weights_only=False, mode='max', period=1)  # val_f1_metric
```

As you can see above I have used F1 score as metric. For test data evaluation I have also used confusion matrix, Sklearn's weighted F1 score and Kappa (which I got to know about recently, have to dig deeper).

On Walmart data the above model gave the following result:

```
baseline acc: 87.5
[[ 49   0  15]
 [  0  47  14]
 [195 179 501]]
F1 score (weighted) 0.6643640827534588
F1 score (macro) 0.45295840345749205
F1 score (micro) 0.597
cohen's Kappa 0.20488982890333984
Recall of class 0 = 0.77
Recall of class 1 = 0.77
Recall of class 2 = 0.57
Recall avg 0.7033333333333333
```

This result somewhat varies every time I run it, which may be due to Keras weight

initialization. This is actually a known behavior, with a long thread of discussions here. In short you have to set random seed for both numpy and tensorflow. I have set random seed for numpy only. So I am not sure if it will fix this issue. I will update here once I try it out. But most of the time and for most other CNN architectures I have tried, precision of class 0 and class 1 (buy/sell) is less than class 2 (with class 0/1 being 70s).

The authors got following results:

Table 2: Confusion Matrix of Test Data (Dow-30)

		Predicted		
		Hold	Buy	Sell
Actual	Hold	52364	18684	23592
	Buy	1268	5175	3
	Sell	1217	8	5059

Table 3: Evaluation Of Test Data (Dow-30)

Total Accuracy: 0.58			
	Hold	Buy	Sell
Recall	0.55	0.80	0.81
Precision	0.95	0.22	0.18
F1 Score	0.70	0.34	0.29

Result for Dow-30 presented in the paper

If you notice, “hold” class scores are significantly worse that “buy/sell”, both in our result and the paper’s. I think this result is quite promising given that model can identify most of the buy/sell instances. Here is what the authors have to say about it:

> “However, a lot of false entry and exit points are also generated. This is mainly due to the fact that “Buy” and “Sell” points appear much less frequent than

> "Hold" points, it is not easy for the neural network to catch the "seldom" entry and exit points without jeopardizing the general distribution of the dominant "Hold" values. In other words, in order to be able to catch most of the "Buy" and "Sell" points (recall), the model has a trade-off by generating false alarms for non-existent entry and exit points (precision). Besides, Hold points are not as clear as "Buy" and "Sell" (hills and valleys). It is quite possible for the neural network to confuse some of the "Hold" points with "Buy" and "Sell" points, especially if they are close to the top of the hill or bottom of the valley on sliding windows."

3. Further Improvements

- There is definitely a lot of room for better network architecture and hyperparameter tuning.
- Using CNN with same architecture on other datasets didn't give as impressive precision for buy and sell. But by playing around with hyperparameters we can definitely improve it to similar figures as Walmart.
- Although these results seem good enough, there is no guarantee that it would give you profits on real world trading because it would be limited by the strategy you choose to label your data. For example, I backtested above trading strategy (with original labels and not model predictions!) but I didn't make much profit. But that depends on the labeling of the data. If someone uses a better strategy to label the training data, it may perform better.
- Exploring other technical indicators may further improve the result.

4. Conclusion

I started working on this project with a very skeptical mind. I was not sure if the images would have enough information/patterns for the ConvNet to find. But since the results seem to be much better than random prediction, this approach seems promising. I especially loved the way they converted the time series problem to image classification.

UPDATE- 12/7/2020: Major update- There was a bug in label creation, which was assigning labels to last day of the window instead of middle item. I have also a updated this article with new results. New model updated in "stock_keras.ipynb"

Code fix is available on GitHub as well. Please note that since I have moved to PyTorch and I don't have a working Tensorflow environment anymore, I trained this model on cloud and had to copy paste the fixes. So, I couldn't test the final code completely (the training part).

Inserted the code gists which were missing due changes to my GitHub account.

UPDATE- 23/2/2020: I have just discovered a bug in my model creation function "create_model_cnn", where I use the following check to add MaxPool layers:

```
if params["conv2d_layers"]['conv2d_mp_1'] == 1
replace this with
if params["conv2d_layers"]['conv2d_mp_1'] >= 0
```

Do the same for "conv2d_mp_2" as well. There is nothing wrong with the model or program as such, it's just that I had been exploring the hyperparameters search space without any MaxPools :-(. Need to explore if model can perform better with MaxPool layers.

UPDATE- 09/02/2020: Added explanation for some of the more complicated

technical indicators in “Feature Engineering” section.

Deep Learning | Convolution Neural Net | Stock Market | Image Classification | Data Imbalance

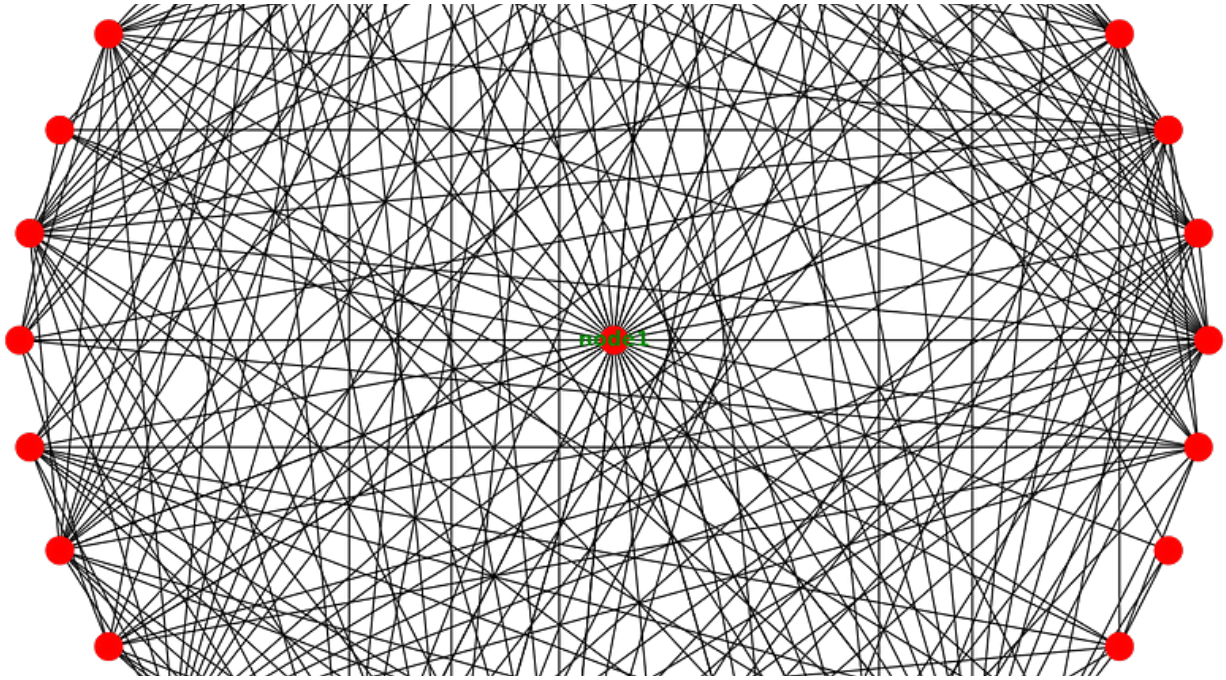
Follow

Written by Asutosh Nayak

589 Followers · Writer for Towards Data Science

Machine Learning | Android | Bibliophile | Aspiring Writer. Feel free to connect on LinkedIn https://www.linkedin.com/in/nayakasu92/

More from Asutosh Nayak and Towards Data Science



Asutosh Nayak in Towards Data Science

Shortest Path Distance with Deep Learning

Implementation of the research paper “Shortest Path Distance Approximation using Deep Learning Techniques”

15 min read · Jun 24, 2020

Cristian Leo in Towards Data Science

The Math behind Adam Optimizer

Why is Adam the most popular optimizer in Deep Learning? Let's understand it by diving into its math, and recreating the algorithm.

16 min read · Jan 30, 2024

Sheila Teo in Towards Data Science

How I Won Singapore's GPT-4 Prompt Engineering Competition

A deep dive into the strategies I learned for harnessing the power of Large Language Models (LLMs)

· 23 min read · Dec 28, 2023

 Asutosh Nayak in Towards Data Science

Predicting Stock Price with LSTM

Machine learning has found its applications in many interesting fields over these years. Taming stock market is one of them. I had been...

8 min read · Mar 18, 2019

 1K  35 

See all from Asutosh Nayak

See all from Towards Data Science

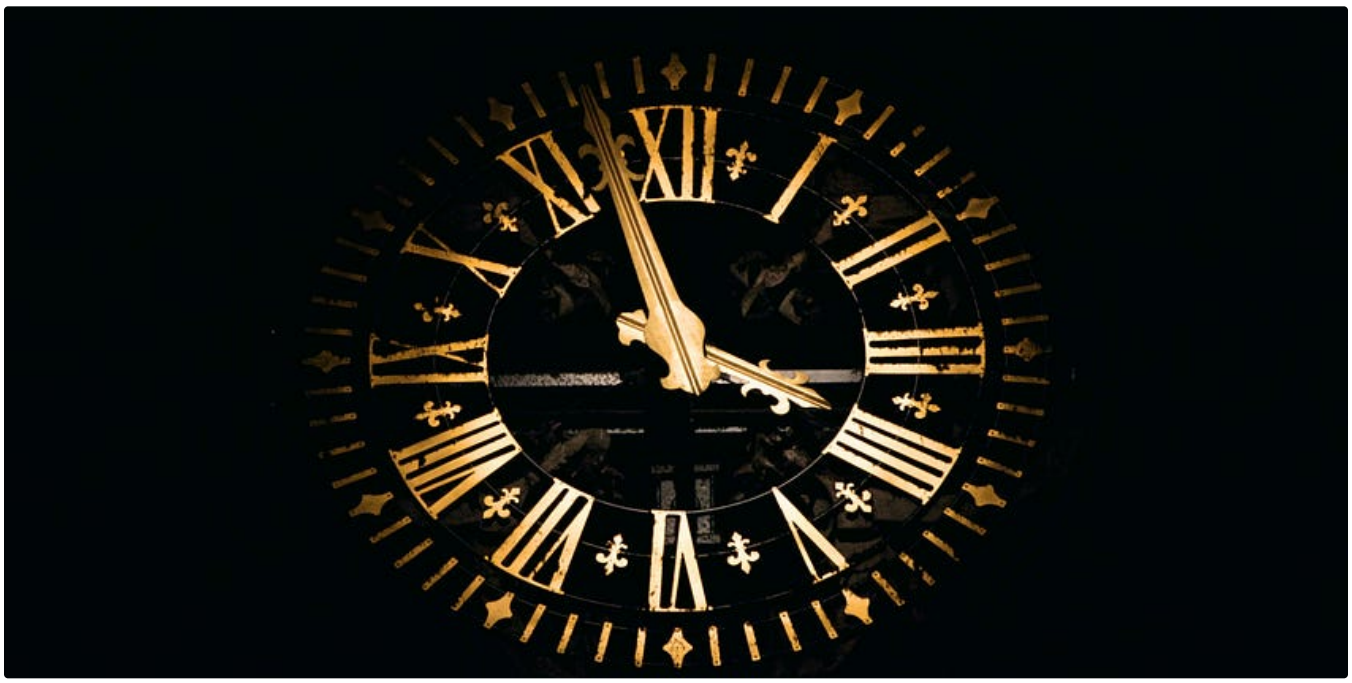
Recommended from Medium

Bruce Yang ByFinTech

Beginner's Guide to FinGPT: Training with LoRA & ChatGLM2–6B

Cost-Effective FinGPT Training: One Notebook, $10 GPU

10 min read · Oct 6, 2023



Jermaine Matthew

Time Series With LSTM

Time series prediction involves predicting future values of a time-dependent variable based on previous observations using statistical or...

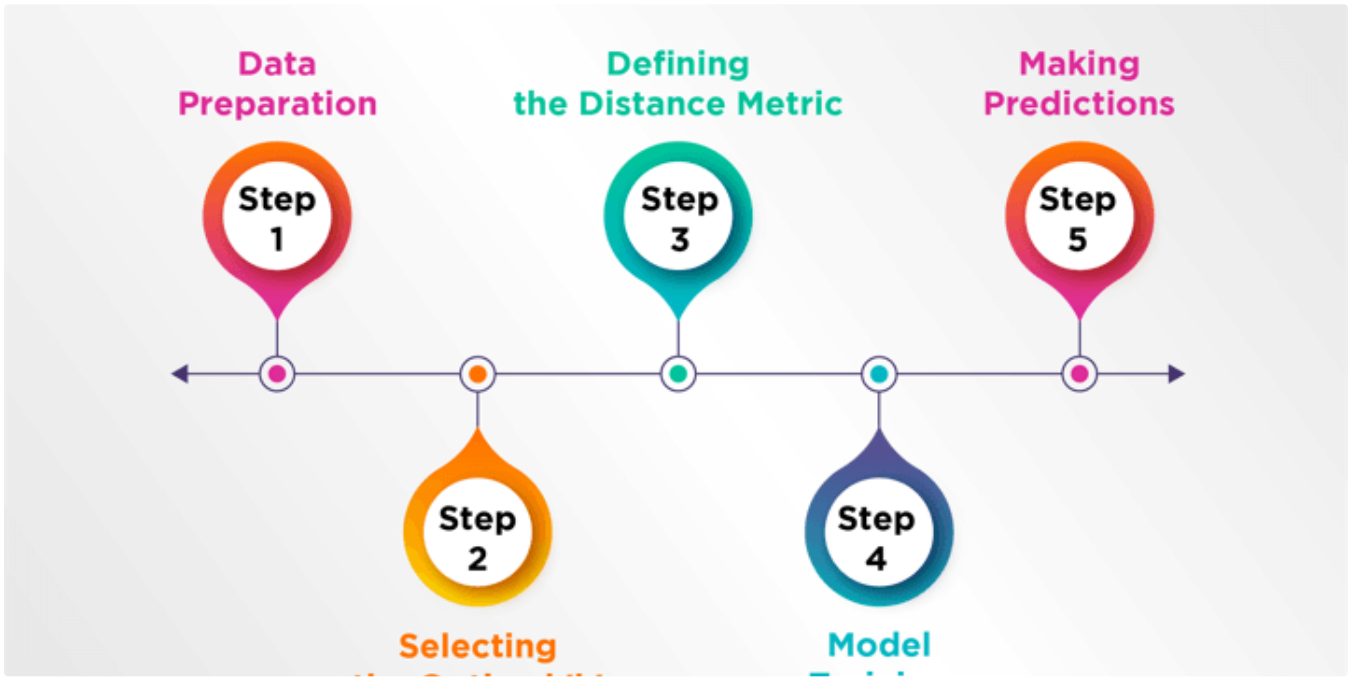
· 13 min read · Oct 18, 2023

Nikhil Adithyan in DataDrivenInvestor

Stock Price Prediction with Quantum Machine Learning in Python

An overview of the challenges and opportunities

17 min read · Jan 23, 2024



Puranam Pradeep Picasso - ImbueDesk Profile

Hyper Optimized Algorithmic Strategy Vs/+ Machine Learning Models Part -1 (K-Nearest Neighbors)

How useful is a Machine Learning Model for trading? A practical approach

19 min read · Jan 3, 2024

35

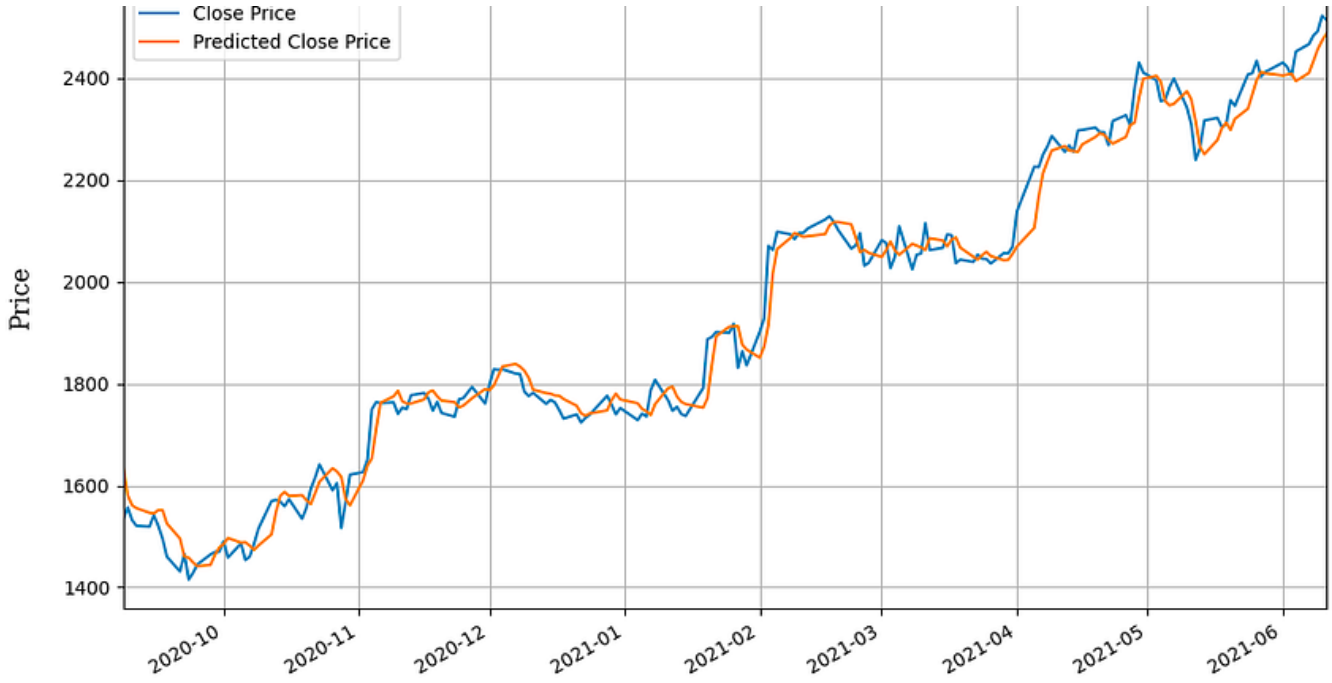
Chedy Smaoui

How I code a Python Stock Screener & A.I. Sentiment Analysis to pick stocks.

Finding stocks to invest in can be a long and tedious process. What if we used both A.I. and Python to create a program that can speed...

8 min read · Dec 21, 2023

232



Nickolas Discolll

Deciphering Market Trends: An Exploration of LSTM and GRU in Predicting Google's Stock Prices

I am going to show you, how you can forecast stock price using LSTM and GRU.

· 11 min read · Oct 21, 2023

215 1

See more recommendations