Machine Learning Engineer Nanodegree

Capstone Project

Pedro Sousa
pjgs.sousa@gmail.com

I. Definition

1. Project Overview

In this project we studied the possibility of using some of the recent advances in Computer Vision with Deep Learning to "look" at financial markets data in order to predict their outcome.

We gathered historical data for Nasdaq100 composite, which is composed of the top 106 biggest companies in the Nasdaq market.

We studied some common indicators used in stock Technical Analysis and used a deep learning model with the VGG16[1] architecture to look at the "images" created with these indicator. To create said images we encoded some indicators using Markov Transition Fields[10] to use as our image's channels.

We also provide some evaluation and comparison of our model against our benchmark.

2. Problem Statement

There is an endless offering of resources about the stock market that range from the best investment strategy to the latest stock pick to trade. There is also extensive research about technical indicators that might help an investor to find "buy" or "sell" signals in a stock's price. Unfortunately, no single indicator is capable of predicting a stock's price accurately.

This, coupled with simply being just too much information and resources about the stock market, makes it impossible for a human investor to cover it back to back and take into account every detail when it comes to the buying or selling moment.

This poses a good challenge for a machine learning problem. Given all the publicly available information about the stock market and each company's historical stock price, one should be able to feed in all the past historical prices across the market into a machine learning model in order to learn patterns and fluctuations and get in return a forecast for its future price. More specifically, we would like to be able to predict any stock's future price for the next day, 30 days and 60 days. Given this, our further discussion is about how we built a multi label regression model and what we found most interesting about our results and our journey.

3. Metrics

On our problem as a whole we will be using 3 different metrics to help us optimize and evaluate our model.

We will approach our problem as a Regression task and will use Mean Squared Error as our loss function and R^2 as our performance metric. This performance will be on the price returns that we are trying to predict.

We will also evaluate both our final model and our benchmark model using Accuracy. For this we will discretize our model's predictions into either a GAIN or LOSS labels. That is, discretizing the labels according to their numeric signal and evaluating the models as if they outputed one hot encoded labels (with

“gain”/”positive” being the true value). This will both help us directly compare our results against our benchmark but also has a chance of being an easier evaluation metric to understand by someone not familiar with Machine Learning or model evaluation.

For our loss, we first considered both the mean squared error[18] and the mean absolute error[19] shown in the formulas below mainly since those are two of the regression losses already implemented in the Keras [15] library.

$$MSE = \frac{1}{n}\sum_{n=1}^{i}(\hat{Y}_i - Y_i)^2$$

Formula 1. Mean Squared Error of $\hat{Y}$ predictions from Y targets

$$MAE = \frac{1}{n}\sum_{n=1}^{i}|\hat{Y}_i - Y_i|$$

Formula 2. Mean Absolute Error of $\hat{Y}$ predictions from Y targets

If we just used the error itself (the subtraction of prediction and target) we wouldn’t be able to really understand the error of our whole model if it errors distribution had zero mean, since the overestimation and the underestimations would zero out one another.

Both MSE and MAE handle this problem, although they handle it differently. MSE will penalize more bigger error values (due to the squaring of the error) while MAE will treat big and small errors equally (since it is merely calculating the absolute values). Still because of the difference in the squaring vs the absolute values, the MAE will have the same variance and standard deviation as our data, while MSE will have the same as the errors themselves.

This means that MSE is better suited for problems where we want our model to be better contained and behaved in the range of prediction values that it outputs, since too far off predictions will have bigger/worse loss values (which we would like to be our case).

On the other hand, it also means that MAE is more robust to outliers than MSE. Even so, after looking at the distributions of our labels (presented later in the Data Exploration section) we observed that the spread for each of our them is actually ‘closed’ enough (the standard deviation is, at most, 0.17), so we felt confident enough that we should see most our data in a shorter range of values.

Another consideration we later had after looking at our labels distributions , was that they are also similiar to one another, all with mean values close to zero. Had it not be the case and they could actually serve as ‘outliers’ to one another and in that scenario, MSE shouldn’t be our choice (since we will be trying to predict multiple labels in the same model).

II. Analysis

4. Data Set and Inputs

For our problem, we will be using daily stock prices downloaded using panda's data-reader[13].

We will focus our efforts with tickers listed in the Nasdaq100 composite index. For each ticker/company we will be receiving:

Table 1. Input Columns present in the raw datafiles

Column	Description	DataType
Date	The day the data row corresponds to	Date
Open	The price at which the stock started the day	Float
Close	The price at which the stock finished the day	Float
Low	The lowest price seen during the day	Float
High	The highest price seen during the day	Float
Volume	The number of shares traded during the day	Integer

We will also be downloading the official companies list from the nasdaq.com website that will serve as the source for our main list of companies [14].

In total we will be getting the daily stock prices for 106 companies and we will only be using data from 2009-01-01 onwards.

5. Data Exploration

After fetching all the historical prices necessary we will have a csv file for each ticker which will allow us to perform both the features and labels calculations necessary for our model.

We loaded up the data for all our tickers in order to have a feeling for our mean ticker price and behaviour, as well as calculate and analyze the mean returns for each of the three time spans that we are interested in.

In the image below we can see that even though all our price variables have means on the 120's, all of them show skewed distributions to the right. The same happens to the volume distribution.

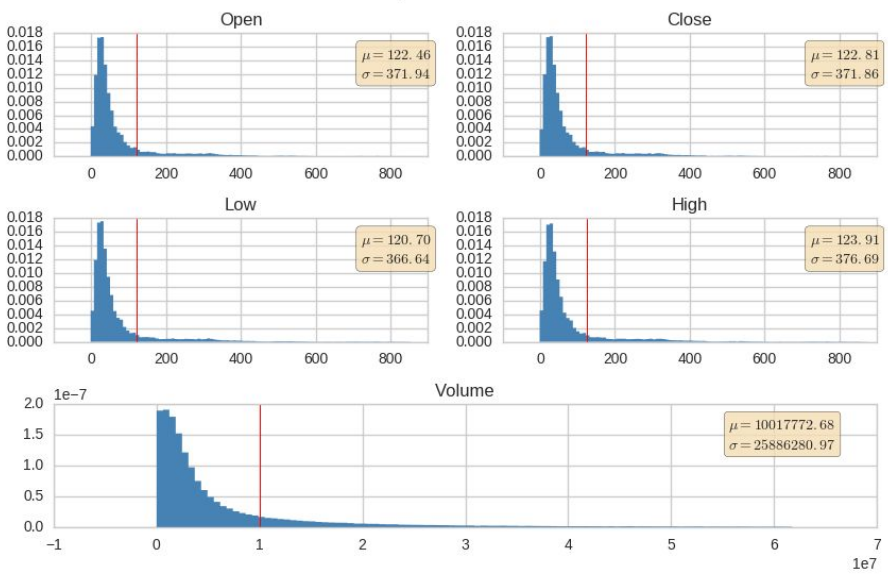

Image 1. Histograms of the each of the prices and volume

Even though our raw features' distributions are skewed to the right, after analyzing our target labels we can see that the same does not happen on the latter. All our labels show a somewhat bell shaped curve. Here we can also look at the means and standard deviations which as we discussed in the Metrics section, are close enough to one another and therefore allowed us to use the MSE loss.

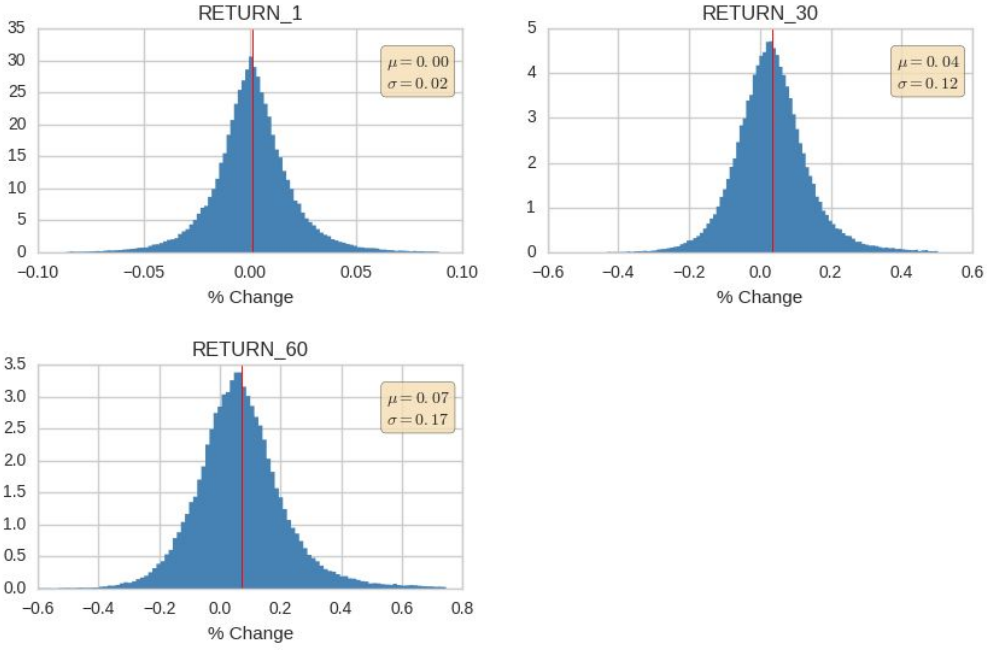

Image 2. Labels' Histograms for % of the close price Change

6. Exploratory Visualization

Instead of just using the raw features we made a study of some common technical analysis indicators. During this study we considered about using all of the indicators we found to build our model, however, during our experiments, we started to see that we were feeding in too much information into our models and eventually falling into the curse of dimensionality trap.

So, after implementing all the indicators we had learned about we still had some questions to answer. Namely, which parameters should we use to each indicator? Or should we use the same indicator multiple times with different parameters? In this last case we would easily fall to the curse of dimensionality[20], if we didn't settle down how much is enough, bur in that case, considering PCA as a means to do feature analysis and selection, how should we normalize and standardize these indicators? For instance, we could just use a linear scale standardization on each one, but maybe it would also make sense to normalize all price/volume/volatility indicators together towards the same price/volume/volatility? Also, we had all these dates, tickers and indicators, how should we organize them?

After experimenting a while and also reading about MTF encoding [10] we decided to only chose 3 indicators to feed into our problems and to just analyze them in terms whether the indicator was based on price or on volume and whether or not it was based on moving averages (being inherently "smoothed"). This analysis is presented in the table below. We also implemented and calculated these indicators for each of our 106 tickers and did a descriptive analysis on each of them. In the following figure 4 we present this analysis.

In the end of the Implementation section we explain why we ended with this approach and also revisit the questions we mentioned above.

Table 2. Technical Indicators Summary

Indicator	Price	Volume	Moving Average
MACD[2]	x		x
CMF[3]	x	x	x
DMF[4]	x	x	
Aroon Up[5]	x		
Aroon Down[5]	x		
RSI[6]	x		x
Stochastic Oscilator[7]	x		x
ADX[8]	x		x
OBV[9]		x	

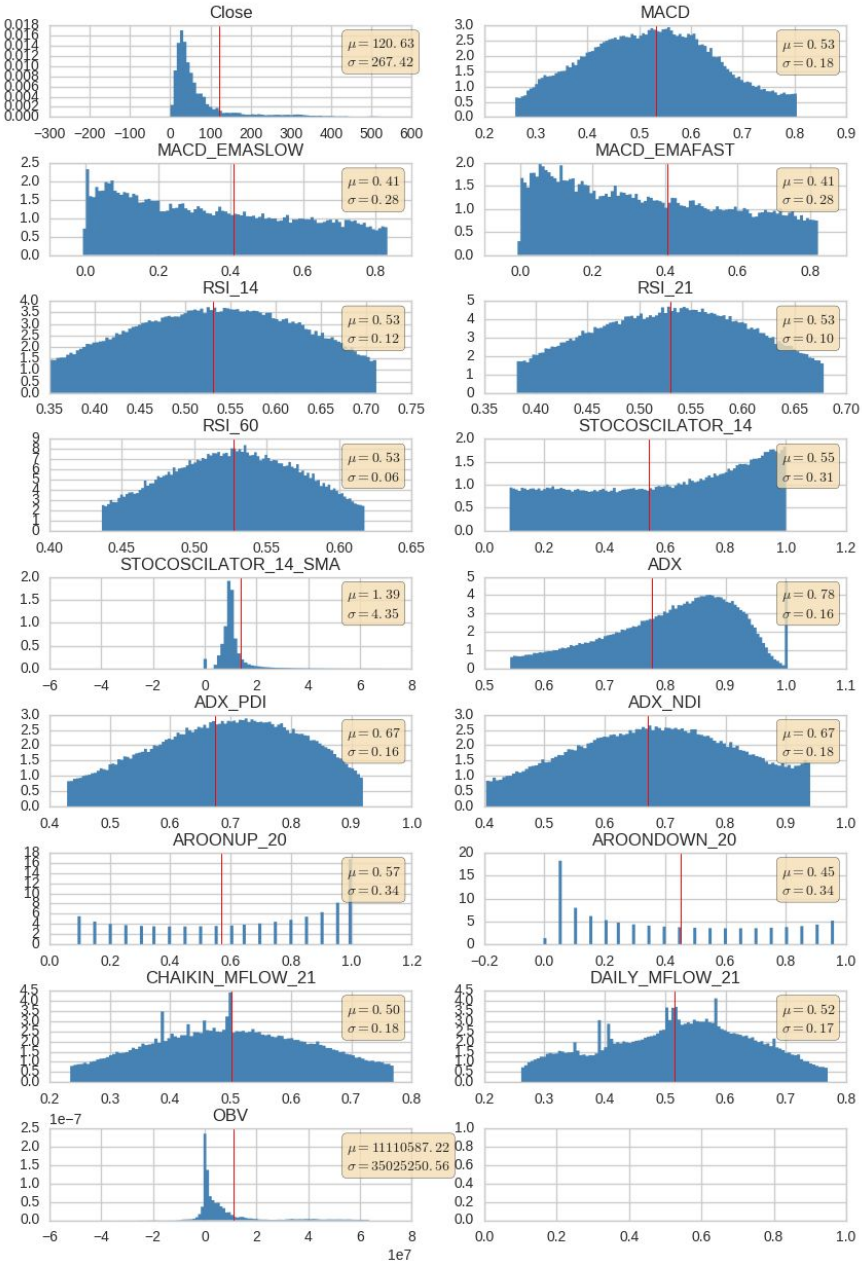

Image 3. Histograms for each of the technical indicator studied

We decided to use RSI and Chaikin Money Flow as our extra features in the images channels.

We chose RSI for being an indicator on price that is actually already normalized by definition, works also as is, without the need of another secondary indicator and also because it shows an uniform distribution when we plot it to the whole market.. Notice that we saw a skewed distribution of the close price in the previous section, but RSI actually 'behaves better' in terms of distribution.

From the indicators on volume we chose Chaikin money flow for its uniform distribution (if compared to OBV) and for being a simple moving average of the Chaikin daily money flow.

From the image above we can see that we actually tried some specific parameters for this analysis. For instance, both our chaikin money flow indicators are for a time window of 21 days or our RSI indicators are for 14, 21 or 60 days. We chose this more or less empirically from what we found to be usual values either in tools

that we found or articles about the topic. So, we ended up using values for these indicators that us humans usually use also.

This indicator choosing process is one of the key decisions in building the dataset that we will be training our model on, more or less, we can be injecting huge amounts of bias to our process fixing these values so soon. For that reason we decided to actually incorporate just the Close price column. In the end, the "statement" to our model in some way will be "look at this, easy to read by a human and usually relevant indicator, also, look at this other indicator. By the way, this what the original data looks like (the Close variable), now train to predict this (the returns).

7. Algorithms and Techniques

a) Deep Learning with Vgg16

Our model will be a Convolutional Neural Network as studied in the Deep Learning course. As an architecture for our model we picked up Vgg16[1].

A Convolutional Neural Network (CNN) is composed of many different modules and layers, such as:

- Perceptrons

Perceptrons are most atomic processing units of a neural network. They can be defined as the same formula of a line [21] as they are really just that. The slope in the line definition is usually called weight, and the intercept is called bias. This means that a perceptron takes in a numeric input, multiplies in by a weight (by how much our perceptron 'matters') and adds a bias to the final result,which therefore the perceptron's output. When we couple together a set of one or more perceptrons we call this a layer. In a neural network model, our layers inputs and outputs can either be our raw data and our target labels, or even another layers' outputs and inputs.

- Activations

The previous definition means that by changing the a perceptron's weights and bias we can make our network emulate a great range of mathematical functions. Adding more perceptrons to a layer (also referred as making the network wider) or adding more layers (referred as making the network deeper) allows us to represent more complex functions. Unfortunately, the combination of multiple linear modules is only able to create a linear system. To target this issue, we can use another processing unit called activation, which is a nonlinear function that is applied to the outputs of our perceptrons. One of this such common functions are called Rectified Linear units [17] and are defined by y=max(0, x).

- Convolutional Layers

Convolutional layers are a specific kind of neural layer that is usually used to process image data or spatially dependent data.

- Filters

On a deep neural network setting, a filter is a set of weights, or perceptrons, that are applied across an entire image. For this to happen the filter can be thought of has a window that starts by getting has input a patch from the top left corner of pixels from the image, each input is computed with the filter's weights which is is passed on to the next layer. This patch is slided across the image by an arbitrary step size, which is usually called stride. Multiple blocks of filters compose a convolutional layer and the output of these layers are usually called feature maps. A convolutional layer will usually expand the number of channels into has many channels has filters it has.

- Pooling Layers

Pooling layers are also window patches but applied on the feature maps and also with a stride equal to the patch size. These perform very simple operations (such as max or mean calculations) of the pixel values on the maps below their window. Because of this, they are intended for feature consolidation and their application result on a dimension reduction of the feature maps.

- Fully Connected Layers

Fully connected layers are sets of perceptrons to which all inputs are fed into all perceptrons

The image below represents the original architecture of a Vgg16 model.

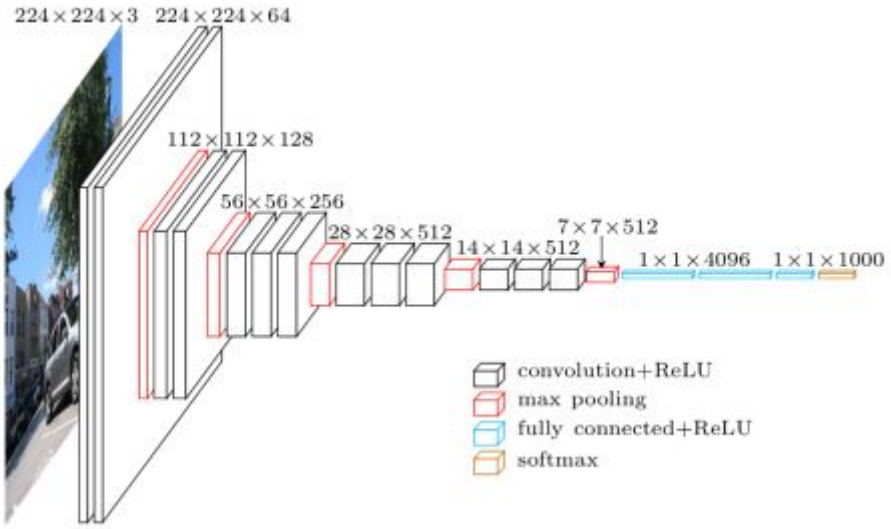

Image 4. Original VGG16 Architecture

In the original architecture for Vgg16, the output layer is a 1x1000 Fully connected layer which has one output weight for each of the classes present in the original ImageNet Dataset. For our model, we made some changes to this architecture.

First of all, we expanded the Fully Connected blocks to four instead of one. So, instead of the 1x4096 + ReLu like the picture above, we have 4x4096 fully connected layers all with relu activations.

We also changed the output layer. This last layer is a 1x3 fully connected layer (one output for each of our target spans, instead of 1000 categories from ImageNet) and without activation (since we are looking to build a regression model).

Given that the original ImageNet dataset consists of natural images from the real world, which are remarkably different than our synthesized/encoded images, we will be training our model from scratch and will not use any pre trained weights for the convolutional layers.

b) Markov Transition Fields

In order to encode our time series as images we took inspiration from a method called Markov Transition Field [10]. In the article, the author defines X, with x_i timestep observations as our time series representation and determines Q quantiles into which each $x_i \in X$ is mapped into $q_j \in [1, Q]$.

With this sort of digitized signal we can define a Markov Chain Transition Matrix [] $W_{Q,Q}$ that has each w_i, j given by the normalized frequency with which a point in quantile q_j is followed by a point in quantile q_i.

A Markov Transition Field is a matrix $M_{t,t}$ in which t defines the the number of time steps on our time series.

M_{ij} in the MTF matrix denotes the transition probability of $q_i \longrightarrow q_j$. In other words, we need to look up what is the digitized state we are in i, what digitized state we are in j and look in Q what is the probability of that transition.

As we mentioned, we took some inspiration from this and did some changes to this approach: Instead of quantiles we implemented our approach with bins. Also, instead of encoding the entire time series, we encode a moving time window, creating a different image/encoding to (possibly, if we so choose) each date.

In other words, for one given date, say D, we can look at a time window of T days back that goes from D-T to T. This is the data that we will be encoding. We discretize the range of values into B bins, each bin will be as a "state", and then build a matrix that is a cross tabulation of these bins and that tells us the conditional probability of our time serie jumping to a bin/state i given it was in bin/state j. After this, our markov transition field will be a second cross tabulation of the size of our time window that starts on D-T. For all pairs of time steps in this, we find the corresponding bins and look for that transition probability on the first matrix. So we will be encoding for each time step more less how usual it was that our time series went through some bin in any other time step.

8. Benchmark

We based our benchmark against another model created by us. For this we trained a single model for each ticker. But in this case, our feature space consists only of the ticker's own daily data: Open, Close, High, Low, Volume (all of them normalized). For our labels we also used the calculated returns for each of our 3 time spans.

Each model is just a Fully Connected deep neural network with 3 Fully Connected blocks with ReLu activations (just like the Fully Connected stage on the Vgg implementation that we implemented for the main model).

Like our main model we also trained our benchmark using Mean Squared Error and computed R^2 and Accuracy to evaluate it.

We used the same rule for accuracy that we mention in the Metrics section. In fact, the results that we got at the time we wrote our Capstone Proposal for our benchmark showed very poor results in the R^2 score which is why we decided to also report its accuracy this way.

We would like to mention that it would be appropriate for us to define our benchmark as a classification model (classifying LOSS/GAIN labels) and eventually use a logistic function or softmax as its output. Although, we decided to actually use an equivalent set of fully connected with regression so that our benchmark architecture would be quite like our main model but without the convolutions.

We did the same train/test split as in the main model and also split our training data between 80% training and 20% validation. For each training session, we set up a limit of 1000 epochs, with an early stopping policy of 30 epochs on the validation R^2.

After averaging the results from the models of all the tickers we got the following results:

- Accuracy: 0.13911
- R^2: -1.92856

III. Methodology

In the following section we will describe the approaches we took in order to implement our model. Succinctly, the image below identifies the main modules we built for our data pipeline, for its management and for our results presentation, which will be addressed throughout.

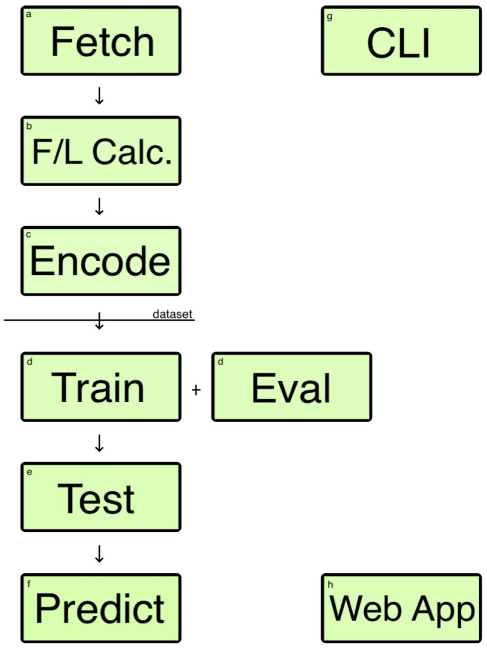

Image 5. main modules of the solution

9. Regression vs Classification

Given that we want to know our tickers' returns for different time spans, from our dataset we calculated a new return variable for each of them. So instead of predicting one label, we will be predicted 3: 1 Day Return, 30 Days return and 60 Days return.

Initially, we considered to discretize our labels and create a multiclass classifier measured using accuracy. With that, we would reduce the space of our target labels and achieve both a simpler problem to solve and make it simpler for anyone (even without prior knowledge about machine learning or model evaluation metric) to find it easier to understand what our performance metric actually meant.

Although, a few problems could arise from there. For one, we would be able to directly influence our problem's complexity (for example, discretizing into only two buckets as we ended up doing as we already mentioned for the comparison with our benchmark); we could also be introducing some discretization bias or error in the case of ignoring some important range values that could be key for our model's performance; we could be unbalancing our dataset.

Considering this, our approach was to set up our labels for a regression model and to train it as one. As stated in a previous section we used mean squared error during training to guide our weights updates and R^2 to measure its performance.

In the end, we still offer a comparison of our discretized labels, but this is done merely for evaluation and comparison purposes against our benchmark, not to evaluate the main model itself.

10. Data Preprocessing

In order to feed the data into our model we add to take into consideration:

a) Data Fetch

Using the full listing in for the nasdaq100 index we used pandas data reader in order to use Google Finance API to download and store to disk the raw data with the daily prices for each ticker.

b) Feature and Label Calculations

As stated in the previous section, we used Close, RSI and CMF as our features. We used a time window of 60 for the RSI indicator and a time window of 21 for RSI. Each of these were then treated in the Feature encoding process.

For our labels we created 3 label column in which we calculated the returns on the Close column for each of our time spans (1, 30 and 60 days). We could have used the prices themselves, but we saw in the Data Exploration section, using the returns we get uniform distributions for our labels, which also was one of the factors while choosing the indicators for our features.

Both our features and labels calculations were done at once for each ticker and stored to disk.

c) Feature Encodings

Each of the features calculated previously we planned to treat each one as a separate time series and encoded using our version of Markov Matrix Field. After encoding these, we were able to use each encoding result, aligned by date, as a different channel of an RGB image. Which were then the input to our model. The standardization was performed on the fly during training and consisting in setting the images to have zero mean and unit variance (we computed the values taking into account our train dataset).

So, after having all features and labels calculated, we then generated our MTF images. For the Markov transition Matrix (Q) and our Markov transition field (M) we ended up using the sizes of 50 bins and a time window of 224, generating 224x224x3 images (we explain why we chose these values in the Refinement section).

On this stage, from all the possible observations of ticker+date pair, we only computed a subset. The reason for this was because with the 224 encoding size, we are actually looking back 224 days in each image. This mean that two contiguous dates will look almost the same, so we would be unnecessarily training almost the same data with very similiar images. Instead, we used simple random sampling from all the pairs with a chance of ~1.51 of each pair being chosen to be encoded, resulting in 2509 images being generated.

This images and the labels calculated in the step before were our dataset to be fed into our model.

Since this was one of the most time consuming tasks in our preprocessing stages we implemented the encoding process with a sort of parallelization in mind in order to make use of multi-core environments. Our CLI tool also made it easy to use this functionality.

12. Implementation

d) Create train/test split

For our final model, we used cross validation to evaluate our training progress. Our data split was based the date ranges that we had on disk after running the encoding process. For those images, independently of their ticker, the ones corresponding to dates between 2010-02-15 and 2014-12-31 were used for training, and the ones in the range of 2016-02-15 and 2016-12-31 were used for testing. On each training sessions we used a random split of 80-20 (80% for training and 20% for validation). During training all the test features are ignored and we only keep those that are for training or validation.

Since loading all the encoded images at once into memory would be infeasible we loaded batches of 32 images at a time. We also had to implement this by hand, even though Keras[15] has a Class called ImageDataGenerator, that lets us pass in a list of file paths to load into memory later, it is not designed to work with regression tasks. It expects the images to be organized by class folder. The only way for us to do this would be to actually create multiple folders for the various continuous values that we found.

For our train sessions we set a limit of 2000 epochs with an Early Stopping[25] policy if the validation R^2 score didn't improve for more than 30 epochs.

We also allow our models to be fine tuned. The training process creates a new model and randomly initializes the weights by default, but we can reload and retrain previous models.

e) Test

We implemented our test process almost identical to the training process. Except it does not perform any training, and throws away the train and validation data and runs only with the testing. In our implementation we are able to name our models and store each stage's progress to disk. This means that during testing we don't need to re-evaluate neither the training nor the validation data. It is all stored and updated into disk using pandas library. This allows us to run everything in our CLI and then later to just load the results in a jupyter notebook.

f) Predict

In our pipeline the prediction module is apart from the evaluation or testing module (and the evaluation processes only store scores results). This module is able to handle in itself almost the whole pipeline (to the exception of training). Given any pair of date+ticker, or any list of pairs, it is able to re-fetch the raw data and re calculate the engineering and encoding processes before loading a previously trained model into our gpu and running a prediction. All this only happens if the requested pair does not exist in disk, in which case we just run and return the prediction. This module is very useful to query our models from either inside a jupyter notebook or to prepare all the data needed for the Web App.

g) Command Line Interface

We built a simple CLI module with python. With this CLI we were able to let longer processes running in the command line of our server. Our CLI, coupled with a tmux server made sure we could run our processes overnight without them hanging due to connection drops over ssh or the jupyter kernels hanging.
We were also able to run multiple trials with the same parameters, so our reported results were more reliable. Even though we didn't do it, due to the fact that we really only thought about it during the redaction of this report, we could have rather use it as a sort of cross validation with bagging[16] since we were randomly re-setting our validation split and then also storing the weights files at each different trial.
Our CLI is able to setup the directory structure for our product and run any process on our data pipeline (fetch, feature engineering, feature encoding, train, evaluate).

h) Web App

We built a web application using Flask[26] as our backend and Knockout.js[27] coupled with Twitter Bootstrap[28] and Plotly.js[29] as our frontend.
Our web app allows the users to browse the price data for our list of stocks and also query some of the predictions previously given by our model.
Initially, we planned our Web App to query our models in runtime, which was also the reason that we came up with the data pipeline as is. Although since our data preprocessing steps turned out to be quite time consuming (especially the encoding process), we decided to query pre-processed/offline predictions. We only provide these offline predictions to the tickers AAPL, GOOG and NFLX (Apple, Google and Netflix) in the deployed environment. Although, through the CLI, in our or any other development environment it is possible to run and predict any pair of
Date+ticker context. Another reason we didn't deploy our model in our web server was also because of the weight files. The VGG16's require a bit more of disk space than we initially realized and we only really noticed that cost when it came to the deploying phase, already using VGG16 and also with a bagging approach with 5 models to deploy into the server.

13. Refinement

In order to create our final model we experimented with some parameters: the bin size (the number of bins used when discretizing the signal in the encoding process), the encoding size (the final dimensions of the images), the filter size of our convolutions and the optimizer for the backpropagation.

We did this to find the model that best generalized our data and we looked for the best validation R-Squared Score.

For our first goal, we tried to settle down what kind of dimensions for the model would work better. So we started with an encoding size of 224 (the original VGG16 image size) and a filter size of 3x3 (also the values from the original architecture). Instead of performing a full grid search on all the parameters combinations, we performed a Greedy Search. With the mentioned parameters we tried to find which bin size was best (this was experiment 1), then chose the best bin size and tested for the best encoding size (this was experiment 2). After that, we used the best bin and encoding size to look for the best filter size (this was experiment 3).

Having the dimensions of the model settled, we then proceeded to try which optimizer [11] [12] [30] would be better for our problem (this was experiment 4).

As stated in another section, we ran each of our experiments on 20% of our data (approximately 300 images) with an 80-20 train-validation split, we used early stopping with a tolerance of 30 epochs on the R-Squared validation.

For each experiment we ran 5 trials and averaged the results of the best performing epoch from each trial.

All trials from experiments 1 through 3 started from scratch with preset weights to the default uniform distributions provided by keras[15]. On experiment 4 we fine-tuned the models of our best filter size from experiment 3. On this finetune process, we reset the weights to the fully connected layers and turned off training for the convolutional layers and then proceeded also with 5 trials using the same parameters to the data split and early stopping policies.

We present the results for each experiment below.

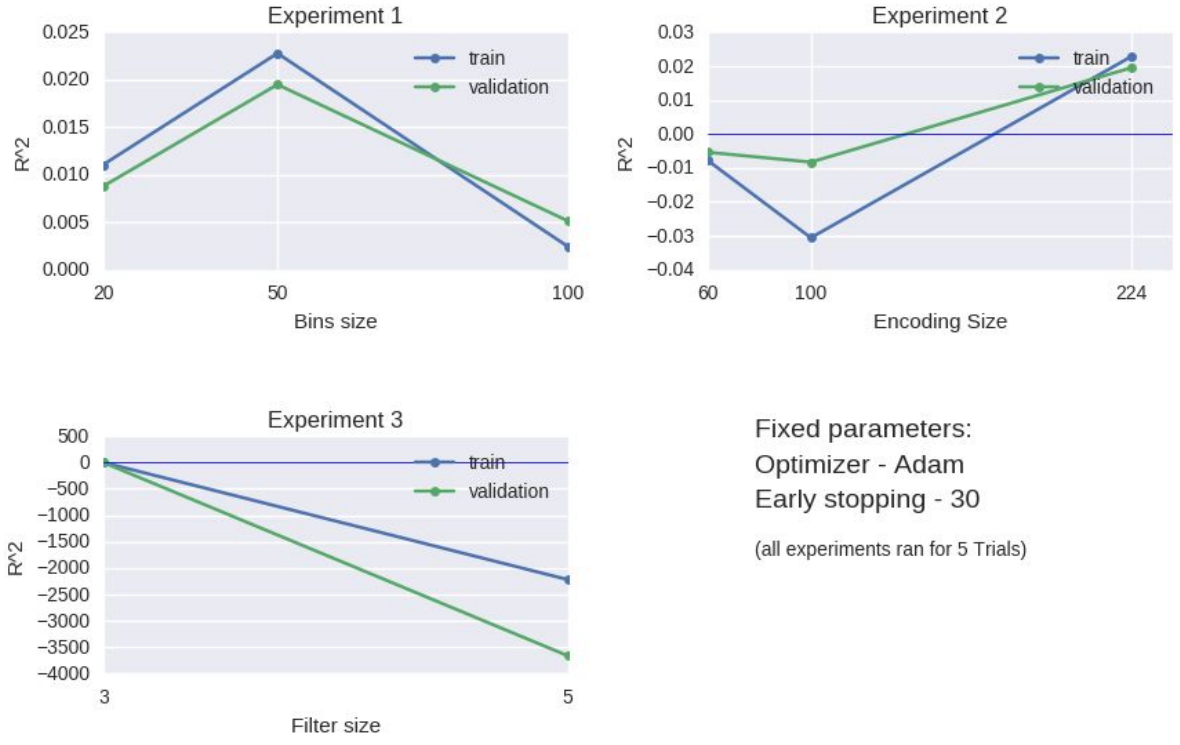

Image 6. Results for experiment 1, 2 and 3

On experiment 1, we were expecting that the bigger bin size would perform better, since that was the one with less information loss during the discretization. Although, that wasn't the case and the bin size of 50 showed better results, so we chose it to move on to Experiment 2.

On experiment 2, one could propose that the model would benefit from a smaller image. Since a (bit) smaller bin size (with less information) had a better performance and also since we were not seeing too great results (our

r-squared score was in ~0.02 for experiment 1, which also shows an extremely high bias), maybe a smaller image with less information (that also covers a smaller timespan on the raw data) would also be a benefit. Although that turned out not to be the case, since the 224 encoding size ended up being the only one that somewhat represented the data.
Finally in experiment 3, we were not sure of what to expect from the change of filter sizes. We initially planned to experiment with 3x3, 5x5 and 7x7 filters, although the 7x7 filters were not manageable with a 224 encoding size in our environment, so we ended up not considering it for our fine tuning.

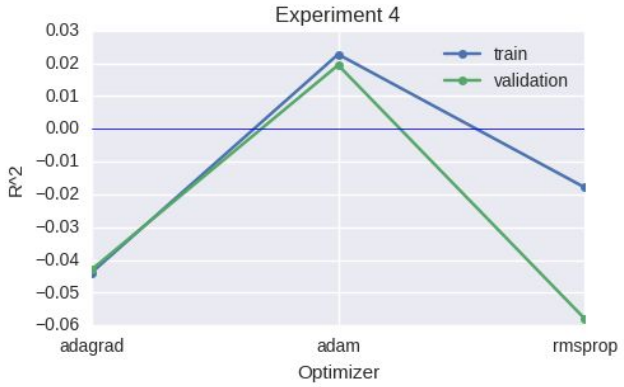

Image 7. Results for experiment 4

On experiment 4, from the three optimizers we experimented, adam ended up showing a better performance than the other two.

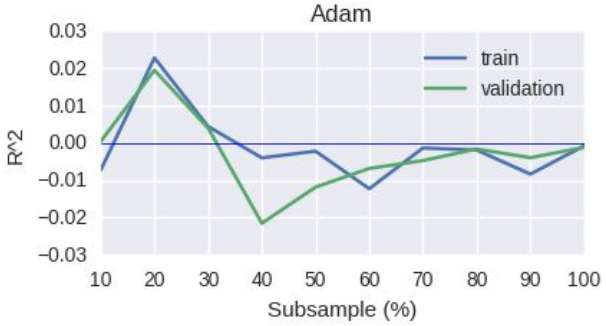

Image 8. Results for experiment 4 with multiple sample sizes

After experiment 4 we decided to run create and run our model with multiple sample sizes.

As already stated before, our model suffers from high bias (it starts off with a poor score and it only converges to a even worse score). In the case of adam, we can also see a problem of high variance from a sample size 30% to 50%.

IV. Results

14. Model Evaluation and Validation

In order to summarize what we had so far after the fine tuning process:

- 224x224x3 images
- Each image channel represented:
    - the Close Price
    - the RSI indicator with a 60 days window
    - The Chaikin Money Flow indicator with a 21 days window
- Each image channel had been discretized using 50 bins
- Our model was a VGG16 architecture
    - With 3x3 filters
    - With 4 fully connected block
    - Trained using Mean squared error with the adam optimizer
    - Standardizing the input images to have zero mean and unit variance
    - Performing regression for 3 different outputs

Although initially we planned on creating only one model, since we had already set up our solution to run multiple trials at once we made use of this in order to evaluate our model. The results we present below are the final solution as an ensemble of the 5 trials we had run and stored in the end of Experiment 4 from our refinement stage. This means that our results are the mean values of the 5 models and the training process ended up following a bagging approach in its train/validation split.

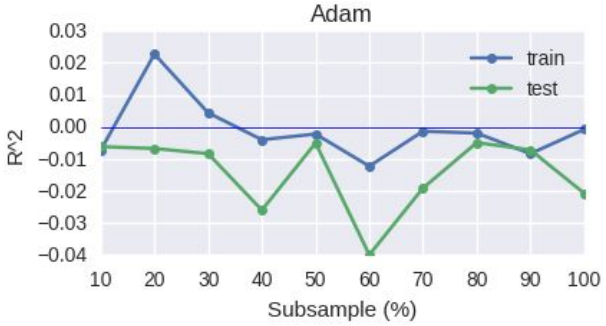

Image 9. Test Results

Above we can see again the problems of our model. As we mentioned on the previous section it is highly biased on the training data since even on training it is not able to properly represent the data. Evaluating it against our test split we actually observe even more variability on subsamples 20, 60 and 100.

Something we found that would be interesting to analyze was how good/bad was it for each of the individual outputs. To do this we used the same models but just did a split on the labels before evaluating (we did not retrain for this).

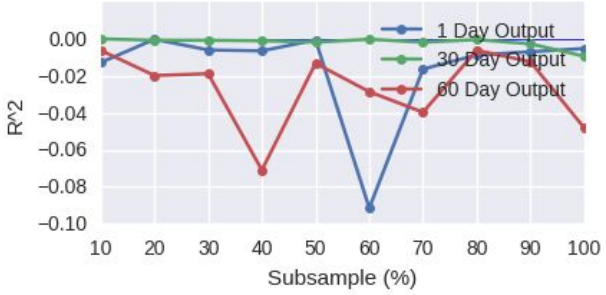

Image 10. Multi Output Test Results

At first we were expecting to see at least one of the outputs better enough to say that it would clearly represent the data, although that is not the case. We did find however that our 30 day output performs better than the other two, with the 60 day output being the worst.

With the aggregated results our performance over the multiple subsample sizes looks a bit jittery. But here we can also see that it actually decreases as we increase the training subsample size.

We think that this reflects two major problems on our approach. After looking at our results we think that this might reflect at least one problem in our approach with regards to the time window parameters we used for our features calculations which possibly made our model to become better suited for the 30 days time span prediction (this was a source of bias).

15. Justification

With the results we obtained, our model does not meet our main goal. Even though we didn't settle down on an appropriate range of values, an R^2 score below zero or marginally above means that our model does not appropriately represents our data. With negative values, an usual example is that a better way to predict our data would be just a line defined by zero slope and an intercept on the data's mean.

Although, comparing it to our benchmark we did see some improvements. Comparing each of the models trained with the full datasets we went from an accuracy of 0.13911 to an accuracy of 0.64315 on our main model.

V. Conclusion

16. Free-Form Visualization

In this section we will show an example of each step for our encoding process. In order for in to be easier understand the images we first present our example with an encoding of size 10x10 with 4 bins discretization.

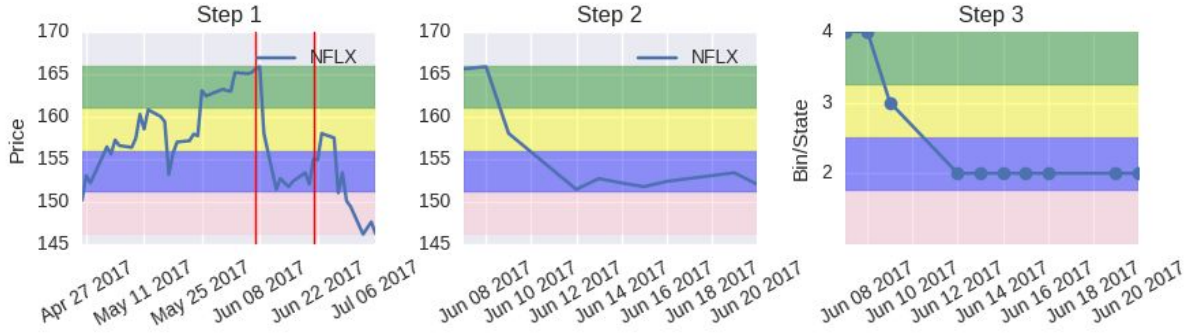

Image 11. Time series visualization

On the image above, in step 1, we can see an example of the close price series for Netflix. We can also see the 4 bins being identified based on the full series (these are the 4 different horizontal shades) and arbitrarily, we will be encoding the date 2017-06-21 (denoted by the space between the red line). For step two we have the slice of the series that we will have for the encoding. Step 3 shows the same series after being discretized into each bin. After this we can look at each matrix (below). For this example we can have easily identifiable probabilities, the values present here are 0, 0.5 and 1.0 as identified in the colorbar.

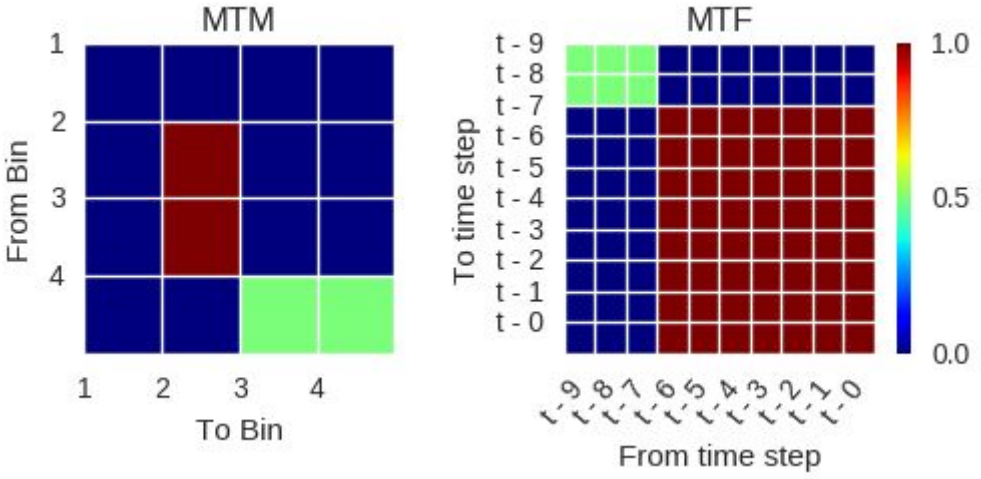

Image 12. Markov Transition Matrix and Markov T. Field encoding

17. Reflection

Looking back even before the beginning of our capstone we do have to admit some uncertainty regarding the subject that we should choose. Since we still didn't have Deep Learning on our syllabus at the time, and given all the buzz around the topic and also because a lot of the competitions on Kaggle about computer vision we just decided on what technology we would like to use rather than what problem we would like to solve. It also started, to some extent, as a problem of focus on what to do and what to pursue even before being a problem of how to do it.

To the model and our solution itself, we do think that our main performance on the R^2 score did not impress, but maybe that also had to do a bit with our somewhat cloudy formulation of our goal (only by now did we really realized that we wanted 'to predict a stock's return' but we didn't actually settled on by how far off).

Since we had never had contact with deep learning, this was regardless a great experience. We did learn a lot, both from the Deep Learning materials but also a lot of other resources ([31],[32],[33], to name just a few). Although, no tutorial or annotated notebook could really give us the insight of how massive a deep learning process could be. We did develop most of our solution with a really small subset of our data, but every time we had to tweak or fix something and had to re-run the whole, full process on the full dataset, that was a huge amount of time, and memory and disk space that we didn't properly account for in the beginning of the capstone. Also adding to this fact, we didn't perform an exhaustive analysis of all the existing computer vision architectures and VGG16 is one of the most taxing architectures in terms of weight storage and memory space compared to architectures such as Resnet or Inception v3. Definitely a consideration to have in future projects.

We did try to synthesize and streamline the most we could with our simple CLI, but still, any new experiment could really take a toll and sometimes over parameterization also becomes over engineering, which can also lead to a loss of focus of the real task at hand.

18. Improvement

After finishing this project we do see some improvements that could be made.

- The solution proposal

Given that we saw such discrepancy between the Regression performance and the classification we used to compare our benchmark. It would possibly be best change it into a classification problem (and not necessarily just a gain/loss scenario, it could be around other threshold values)
Still on the proposal, focusing on one target time span could also simplify the problem

- The label encoding

Finding out a way to encode our data into an image was a considerable stepstone in our project. But we didn't really implement the whole framework presented in [10]. This could also be interesting to experiment.
In the same paper, there is also another encoding technique, called Gramian Angular Field which is also another path that could be taken.

- RNN's and LSTM

The de facto standard for time series prediction are Recurrent Neural Networks [34] with LSTM [35] modules. If we had think about it differently at the time of submitting our proposal, this could actually be a more relevant benchmark for our problem.

- Different computer vision architectures

As we stated in the Reflection section. One could try to use the same set of features and labels with a different computer vision architecture, such as Inception V3, Resnet50 or Vgg19, to name a few. Although, regarding the issue of memory and storage allocated, Vgg19 would fall to the same category.

- Reinforcement Learning

Besides all improvements we just mentioned, there is also a huge amount of decisions that we made towards which indicators to use, which parameters to use and how to refine our problem. With the use of our CLI we ended up write some scripts to perform either a grid search or a greedy search, of some sorts, over the parameters that we needed. But all this could also be made by implementing an agent with Reinforcement Learning to make use of our CLI and perform that exploration autonomously.

VI. References

[1] - SIMONYAN K., ZISSERMAN A. - Very Deep Convolutional Networks for Large-Scale Image Recognition

[2] - (online resource) - MACD indicator - http://traderhq.com/ultimate-guide-to-the-macd-indicator/

[3] - (online resource) - CMF indicator - http://traderhq.com/chaikin-money-flow-ultimate-guide/

[4] - (onlne resource) - DMF indicator - http://traderhq.com/chaikin-money-flow-ultimate-guide/

[5] - (online resource) - Aroon indicator - https://www.tradingview.com/wiki/Aroon

[6] - (online resource) - RSI indicator - http://traderhq.com/relative-strength-index-ultimate-guide-rsi/

[7] - (online resource) - Stochastic Oscilator - http://traderhq.com/stochastic-oscillator-ultimate-guide/

[8] - (online resource) - ADX indicator - http://traderhq.com/average-directional-index-indicator-guide/

[9] - (online resource) - OBV indicator - http://traderhq.com/trading-indicators/understanding-on-balance-volume-and-how-to-use-it/

[10] - WANG Z, OATES T - Encoding Time Series as Images for Visual Inspection and Classification Using Tiled Convolutional Neural Networks

[11] - KINGMA D. P., LEI BA J - Adam: A Method for Stochastic Optimization

[12] - DUCHI J. et al. - Adaptive Subgradient Methods for Online Learning and Stochastic Optimization

[13] - (library) - Pandas data reader - https://pandas-datareader.readthedocs.io/en/latest/

[14] - (online resource) - Nasdaq Listing - http://www.nasdaq.com/screening/company-list.aspx

[15] - (library) - KERAS - https://keras.io/

[16] - (online resource) - Bagging - https://en.wikipedia.org/wiki/Bootstrap_aggregating

[17] - (online resource) - Rectified Linear Unit - https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

18 - (online resource) - Mean Squared Error - https://en.wikipedia.org/wiki/Mean_squared_error

19 - (online resource) - Mean Absolute Error - https://en.wikipedia.org/wiki/Mean_absolute_error

20 - (online resource) - Curse of Dimensionality - https://en.wikipedia.org/wiki/Curse_of_dimensionality

21 - (online resource) - Line - https://en.wikipedia.org/wiki/Line_(geometry)

22 - CANNY J. - A Computational Approach to Edge Detection

23 - (online resource) - Pooling - https://en.wikipedia.org/wiki/Convolutional_neural_network#Pooling

24 - (online resource) - Fully Connected - https://en.wikipedia.org/wiki/Convolutional_neural_network#Fully_connected

25 - (online resource) - Early Stopping - https://en.wikipedia.org/wiki/Early_stopping

26 - (library) - Flask - http://flask.pocoo.org/

27 - (library) - Knockout.js - http://knockoutjs.com/

28 - (library) - Twitter Bootstrap - http://getbootstrap.com

29 - (library) - Plotly.js - http://https://plot.ly

[30] - (online resource) - RmsProp - http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[31] - (online resource) - HVASSLabs Tensorflow tutorial - https://github.com/Hvass-Labs/TensorFlow-Tutorials

[32] - (online resource) - Creative Applications of deep learning with tensorflow - https://github.com/pkmital/CADL

[33] - (online resource) - Practical Deep Learning For Coders - http://course.fast.ai/

34 - (online resource) - Recurrent neural networks - https://en.wikipedia.org/wiki/Recurrent_neural_network

35 - (online resource) - Long short-term memory - https://en.wikipedia.org/wiki/Long_short-term_memory