# *Portfolio Management with Machine Learning*

## **Forecasting returns and managing possitions**

## 1. Introduction

With the development of machine learning and the increasing disponibility of free high frequency financial data online, the oportunities to implement this techniques to forecast the behaviour and manage assets arise.

In this notebook I use machine learning algorithms to forecast the returns of assets of a defined universe, and with this forecast take possitions and define an optimal portfolio. The forecasting algorithm used is a Recurrent Neural Network, whose structure is explained with more detail below.

The data used are daily prices, taken from [Yahoo Finance](https://finance.yahoo.com) through the [yfinance](https://pypi.org/project/yfinance/) API. Also I use the [quandl](https://github.com/quandl/quandl-python) API for the money market rates, which extracts the data from the [Federal Reserve (FRED)](https://fred.stlouisfed.org).

## 2. Data Analysis and Visualization

The first step is to define the libraries that I'll use, which will be necessary for the data analysis, visualization and optimization:

# 1. Import libraries:  
%matplotlib inline  
import os  
import quandl  
import yfinance as yf  
import numpy as np  
import pandas as pd  
import matplotlib as mlp  
import matplotlib.pyplot as plt  
import seaborn as sns  
import scipy.optimize as spo  
from scipy.stats import kurtosis, skew  
import seaborn as sns  
from financial\_data import \*  
import tensorflow as tf  
mlp.style.use('seaborn')  
quandl.save\_key('HtwBLPt3k37yZHTvy15K')

Next, we must define the universe of the portfolio. For this I'll use the companies inside the S&P500 stock index as the stock universe. To mitigate the survivor bias I'll use only the stocks that where added before January 2016. First, let's see the S&P500 companies:

sp\_url = 'https://en.wikipedia.org/wiki/List\_of\_S%26P\_500\_companies'  
sp500 = pd.read\_html(sp\_url, header=0)[0]  
sp500.head()

Symbol Security SEC filings GICS Sector \  
0 MMM 3M Company reports Industrials   
1 ABT Abbott Laboratories reports Health Care   
2 ABBV AbbVie Inc. reports Health Care   
3 ABMD Abiomed reports Health Care   
4 ACN Accenture reports Information Technology   
  
 GICS Sub-Industry Headquarters Location Date first added \  
0 Industrial Conglomerates St. Paul, Minnesota 1976-08-09   
1 Health Care Equipment North Chicago, Illinois 1964-03-31   
2 Pharmaceuticals North Chicago, Illinois 2012-12-31   
3 Health Care Equipment Danvers, Massachusetts 2018-05-31   
4 IT Consulting & Other Services Dublin, Ireland 2011-07-06   
  
 CIK Founded   
0 66740 1902   
1 1800 1888   
2 1551152 2013 (1888)   
3 815094 1981   
4 1467373 1989

# Correct invalid dates:  
sp500.loc[sp500[sp500['Date first added']=='1983-11-30 (1957-03-04)'].index,'Date first added'] = '1983-11-30'  
  
# Filter firms that entered the index after December 2015:  
sp500['Date first added'] = pd.to\_datetime(sp500['Date first added'],format='%Y-%m-%d')  
sp500 = sp500[sp500['Date first added']<'2007-01-01']  
print("The number of stocks in the universe is:", sp500.shape[0])

The number of stocks in the universe is: 212

Now that we have defined the stock universe, limit the strain on my machine, this proyect will only take 10 random stocks from these universe. The stocks selected are:

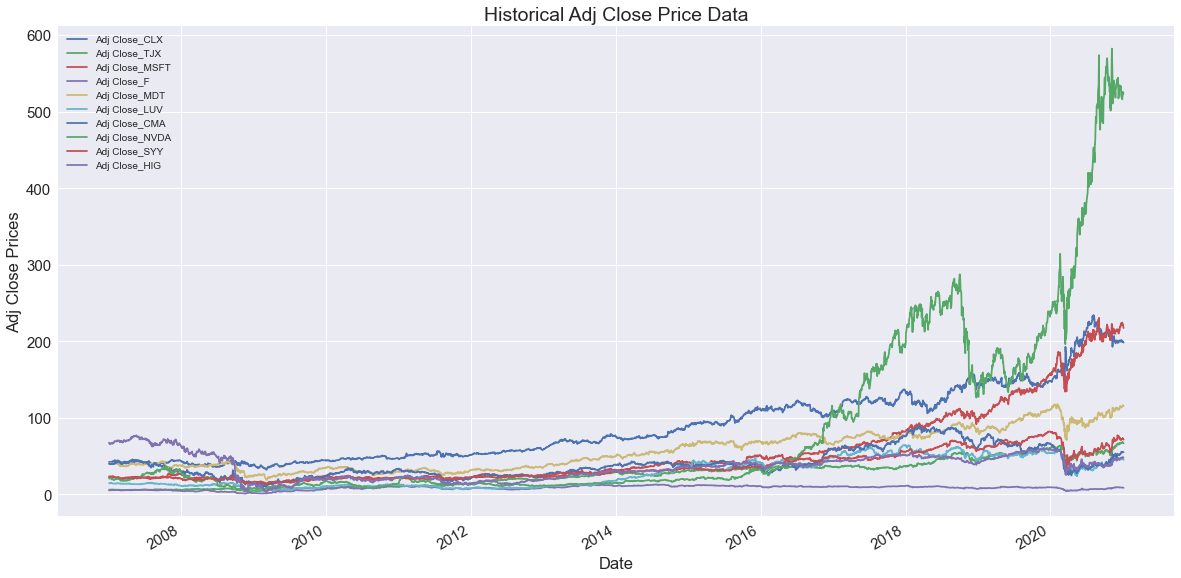
# Define a random seed:  
n\_stocks = 10  
np.random.seed(1792)  
universe\_tickers = sp500['Symbol'].unique()  
portfolio\_tickers = list(np.random.choice(universe\_tickers,replace=False,size=n\_stocks))  
sp500[sp500['Symbol'].isin(portfolio\_tickers)]

Symbol Security SEC filings GICS Sector \  
112 CLX The Clorox Company reports Consumer Staples   
119 CMA Comerica Inc. reports Financials   
197 F Ford Motor Company reports Consumer Discretionary   
221 HIG Hartford Financial Svc.Gp. reports Financials   
310 MDT Medtronic plc reports Health Care   
317 MSFT Microsoft Corp. reports Information Technology   
348 NVDA Nvidia Corporation reports Information Technology   
420 LUV Southwest Airlines reports Industrials   
429 SYY Sysco Corp. reports Consumer Staples   
443 TJX TJX Companies Inc. reports Consumer Discretionary   
  
 GICS Sub-Industry Headquarters Location \  
112 Household Products Oakland, California   
119 Diversified Banks Dallas, Texas   
197 Automobile Manufacturers Dearborn, Michigan   
221 Property & Casualty Insurance Hartford, Connecticut   
310 Health Care Equipment Dublin, Ireland   
317 Systems Software Redmond, Washington   
348 Semiconductors Santa Clara, California   
420 Airlines Dallas, Texas   
429 Food Distributors Houston, Texas   
443 Apparel Retail Framingham, Massachusetts   
  
 Date first added CIK Founded   
112 1969-03-31 21076 1913   
119 1995-12-01 28412 1849   
197 1957-03-04 37996 1903   
221 1957-03-04 874766 1810   
310 1986-10-31 1613103 1949   
317 1994-06-01 789019 1975   
348 2001-11-30 1045810 1993   
420 1994-07-01 92380 1967   
429 1986-12-31 96021 1969   
443 1985-09-30 109198 1987

Now that we have a set of randomly selected tickers that are part of the S&P500 universe,I'll construct an instance of a FinancialData() object, which was done to ease the handling of financial data and its analysis. Because of the distortions generaded by dividend payments and stock splits, I'll analize the adjusted close prices, which eliminate this distortions. The starting date for the analysis is ***2010-01-01***. In the graph below, you can see the historical price data for the 10 stocks selected:

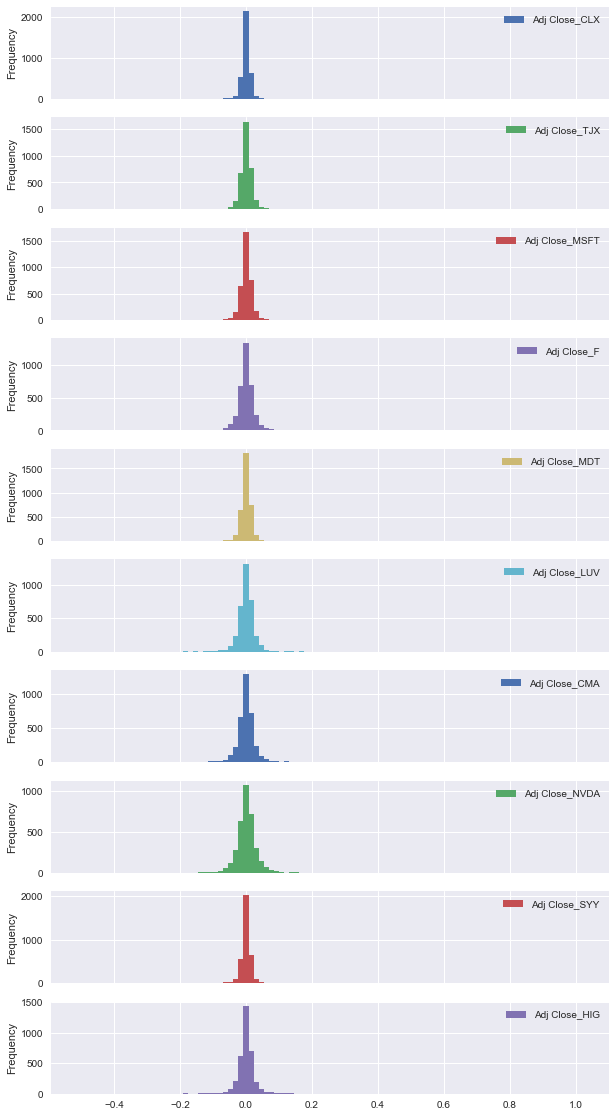
start\_date = '2007-01-04'  
end\_date = '2021-01-05'  
my\_portfolio = FinancialData(  
 tickers = portfolio\_tickers,  
 cols = ['Adj Close','Volume'],  
 start = start\_date,  
 end = end\_date)  
my\_portfolio.plot\_data(figsize=(20,10),fontsize=15);

[\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*100%\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*] 10 of 10 completed

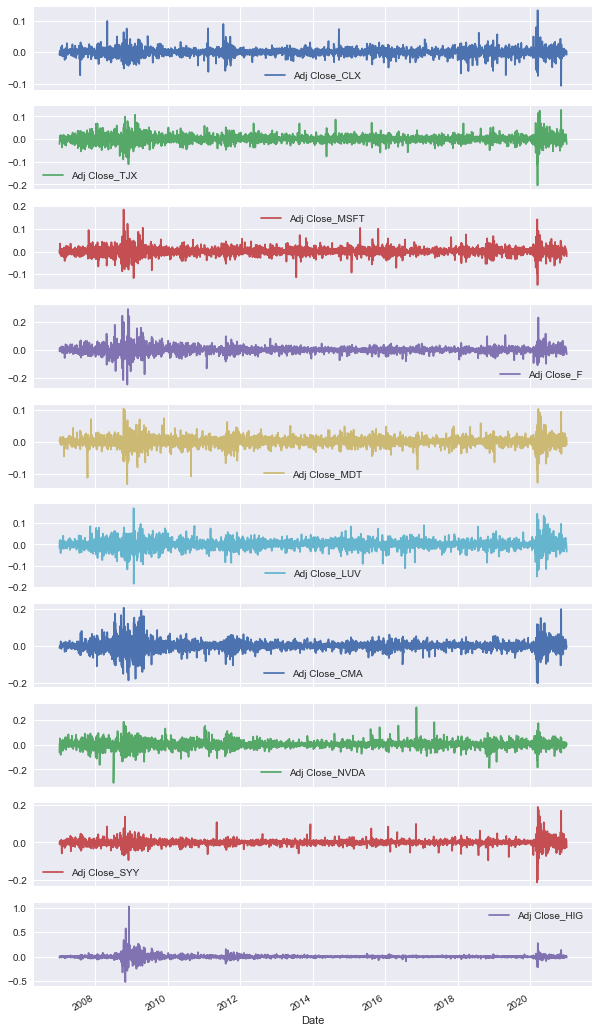


A key aspect are the returns of each of the assets that compose the portfolio. We can see their distribution and behavior in the graphs below:

port\_returns = my\_portfolio.get\_returns(plot=True,subplots=True,figsize=(10,20),kind='hist',bins=100)



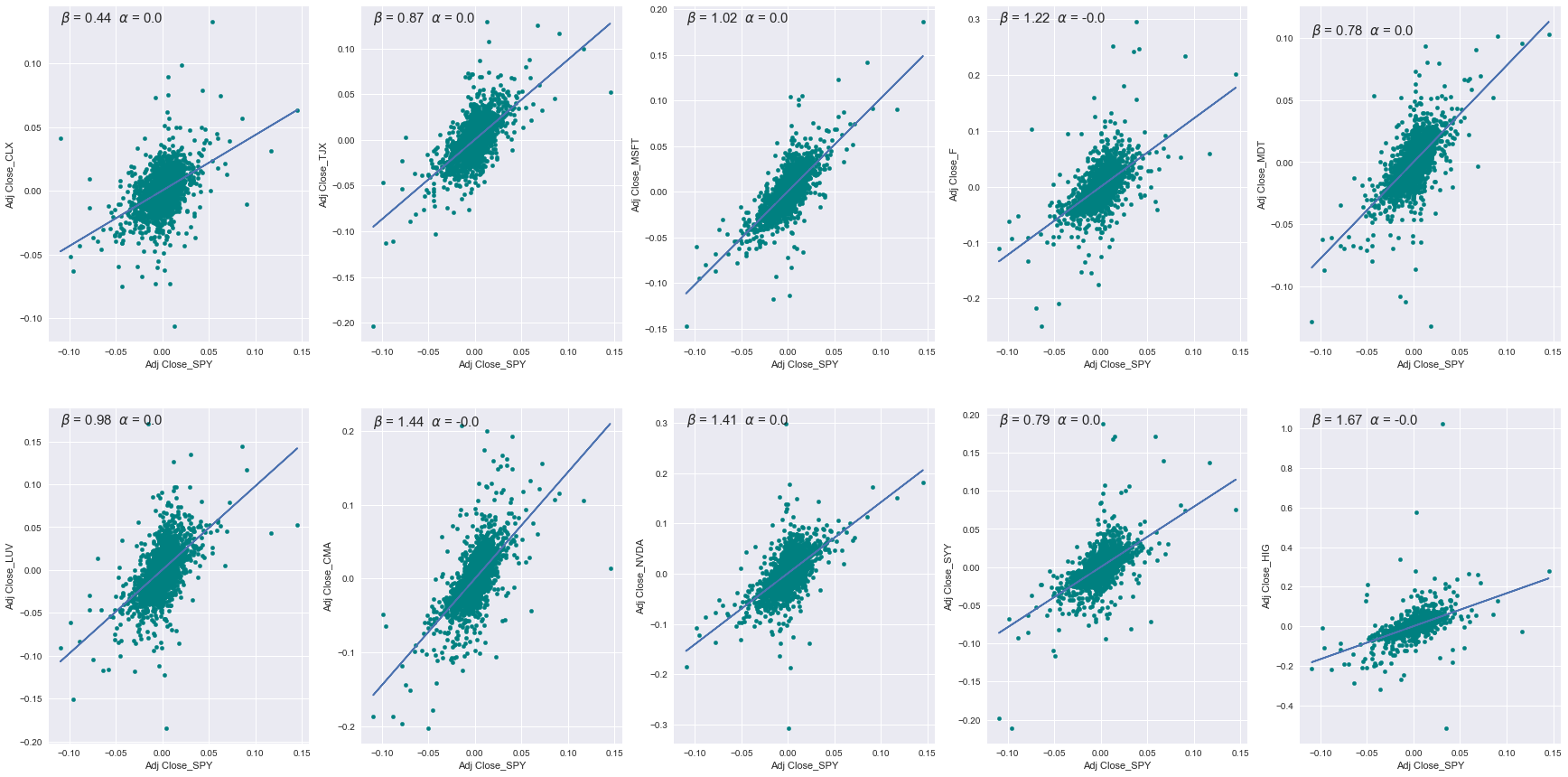
port\_returns = my\_portfolio.get\_returns(plot=True,subplots=True,figsize=(10,20))



The relation between the returns and the market portfolio (in this case I took the SPY ETF, as a proxy for the S&P500 market index), is also very important in the optimization process of portfolios. A visual way to see this relation is by plotting a scatter plot of the returns of each asset vs the returns of the market portfolio, plus the and its corresponding line that resulted from a linear regression:

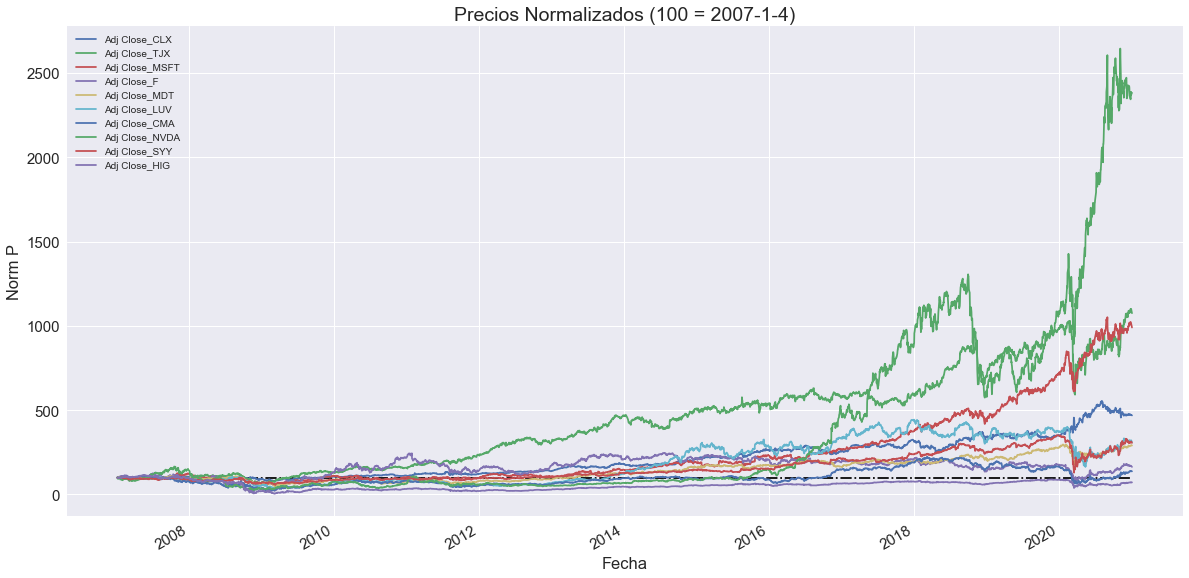
market = yf.download('SPY',start=start\_date,end=end\_date)['Adj Close'].rename('Adj Close\_SPY').pct\_change()  
  
portfolio\_alphas\_betas = my\_portfolio.find\_beta\_alpha(market=market,plot=True,nrows=2,ncols=5,figsize=(30,15),color='teal')

[\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*100%\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*] 1 of 1 completed



Finally, to have a good representation of the evolution of the adjusted prices for each stock, I normalize the prices with the first value available for analisis. This gives us the representation of the evolution of USD 100 if we invested that amount at the begining of the period. This puts in the same scale the evolution of each asset in the portfolio:

norm\_prices =my\_portfolio.get\_normalized\_prices(plot=True,figsize=(20,10),fontsize=15)



Finally, some additional metrics about the returns of the stocks selected for this exercise:

def abs\_mean(x):  
 """Computes the mean of the absolute values of x  
 Inputs:  
 -------  
 x (pandas DataFrame|Series)  
   
 Outputs:  
 --------  
 ab\_m (numerical value): the mean of the absolute values of x  
 """  
   
 ab = x.abs()  
 ab\_m = ab.mean()  
 return ab\_m  
   
port\_returns.agg(['min','max','mean','std','kurtosis','skew',abs\_mean]).transpose()

min max mean std kurtosis skew \  
Adj Close\_CLX -0.106245 0.132749 0.000518 0.012594 11.221270 0.252592   
Adj Close\_TJX -0.203995 0.129032 0.000832 0.017753 10.712812 -0.138494   
Adj Close\_MSFT -0.147390 0.186046 0.000812 0.017955 10.622950 0.344403   
Adj Close\_F -0.250000 0.295181 0.000511 0.027388 19.210743 0.817162   
Adj Close\_MDT -0.132344 0.103000 0.000423 0.015484 9.401576 -0.375849   
Adj Close\_LUV -0.184506 0.170644 0.000578 0.022633 6.500361 -0.149215   
Adj Close\_CMA -0.202907 0.206897 0.000496 0.028530 9.607306 0.208112   
Adj Close\_NVDA -0.307265 0.298067 0.001362 0.030337 9.584639 -0.011336   
Adj Close\_SYY -0.211061 0.187900 0.000469 0.017417 29.086396 0.341663   
Adj Close\_HIG -0.515609 1.023579 0.000705 0.041073 137.121892 4.661057   
  
 abs\_mean   
Adj Close\_CLX 0.008466   
Adj Close\_TJX 0.012052   
Adj Close\_MSFT 0.011966   
Adj Close\_F 0.016940   
Adj Close\_MDT 0.010424   
Adj Close\_LUV 0.015559   
Adj Close\_CMA 0.018100   
Adj Close\_NVDA 0.020652   
Adj Close\_SYY 0.010188   
Adj Close\_HIG 0.019199

## 3. Returns Forecast Model

Now that I have the data of prices and returns for the assets that will compose my toy portfolio, the next step is to make a model that forecasts the returns of each of the assets. This forecast will guide the possitions the portfolio should have with respect to each asset, to maximize the Sharpe Ratio.

To do this, due to the time series nature of the data, I'll use a *recurrent neural network* to forecast the returns of each asset. For this, we need first to define which other variables we'll use as predictive variables of the returns, beyond the past returns of the assets.

Using the [quandl API](https://github.com/quandl/quandl-python), which uses multiple data sources to provide a great variety of financial information, I extract other relevant information, which comes in a daily periodicity. I take 6 additional factors, which are:

* Energy prices, represented by the WTI spot price, taken from the [US Energy Information Administration](https://www.eia.gov).
* Spread between the 10-year Treasury Constant Maturity and the 2-year Treasury Constant Matiruty, taken from the [Federal Reserve of St. Louis](https://fred.stlouisfed.org)
* Spread between the 10-year Treasury and the 3-month Treasury, taken from the Fed.
* Secondary market rate of the 3-month Treasury, which is used also as the risk-free rate, taken from the Fed.
* The Treasury Inflation-Indexed Long-Term Average Yield, taken from the Fed.
* TED rate, which is the spread between the 3-month USD LIBOR and the 3-month Bill, taken from the Fed.

Additionally, using the FinancialData class I compute the technical factors, with a window of 20 days (an approximation of the number of trading days in a month):

1. *Momentum*: measures how much the prices of assets changed in a time window
2. *Simple moving average (SMA)*:
3. *Bollinger bands (BB)*:

# Extracting additional factors:  
new\_tickers = ['EIA/PET\_RWTC\_D','FRED/T10Y2Y','FRED/T10Y3M','FRED/DTB3','FRED/DLTIIT',  
 'FRED/TEDRATE']  
names = ['wti\_spot','10y2y\_spread','10y3m\_spread','3m\_rate','ltiit','ted\_spread']  
add\_factors = quandl.get(new\_tickers, start\_date=start\_date, end\_date=end\_date,  
 api\_key=quandl.ApiConfig.api\_key)  
add\_factors.columns = names  
add\_factors['var\_wti'] = add\_factors['wti\_spot'].pct\_change()  
  
# Fill NaN values:  
add\_factors.fillna(method='ffill',inplace=True)  
add\_factors.fillna(method='bfill',inplace=True)  
  
# Factors: momentum, simple moving average and bollinger bands:  
factors = my\_portfolio.rolling\_statistics()

Now that we have all the additional factors we want for the model, let's merge all the data into a unique dataframe total\_df, clean that dataframe and split the data into a train, validation and test subsets. The ordering of the split will follow chornological order, so that our validation and test subsets are at after the train subset. This is to see how well the model performs in more recent data.

# Merge dataframes to get the full data for modeling:  
total\_df = port\_returns.merge(factors,right\_index=True,left\_index=True,how='left')  
total\_df = total\_df.merge(add\_factors,right\_index=True,left\_index=True,how='left')  
  
# Drop NaN that result from rolling functions:  
total\_df.dropna(subset=factors.columns,inplace=True)  
  
# Define the label columns:  
label\_cols = total\_df.columns[:n\_stocks]  
  
# Define train (70%), val (20%) and test (10%) dataframes:   
train\_p, val\_p, test\_p = 0.7,0.2,0.1  
window\_size = 5  
num\_features = total\_df.shape[1]  
total\_size = len(total\_df)  
train\_size = int(total\_size\*train\_p)  
val\_size = int(total\_size\*val\_p)  
test\_size = int(total\_size\*test\_p)  
train\_df = total\_df.iloc[:train\_size,:]  
val\_df = total\_df.iloc[train\_size-window\_size:train\_size+val\_size,:]  
test\_df = total\_df.iloc[train\_size+val\_size-window\_size:,:]

Now that I have all the information needed for this exercise, I use the WindowGenerator object that can be found in the auxiliary module financial\_data. This object transforms the time series data into tensorflow.data.Dataset objects with the addecuate shape for the models. whose dimensions are:

Because the portfolio is formed by 10 stocks we expect to see 10 returns to be forcasted, as can be seen in the last dimension of the target shape. Another important thing to notice is that, to avoid sequence bias, inside the WindowGenerator object there is a shuffle=True attribute, which shuffles the data:

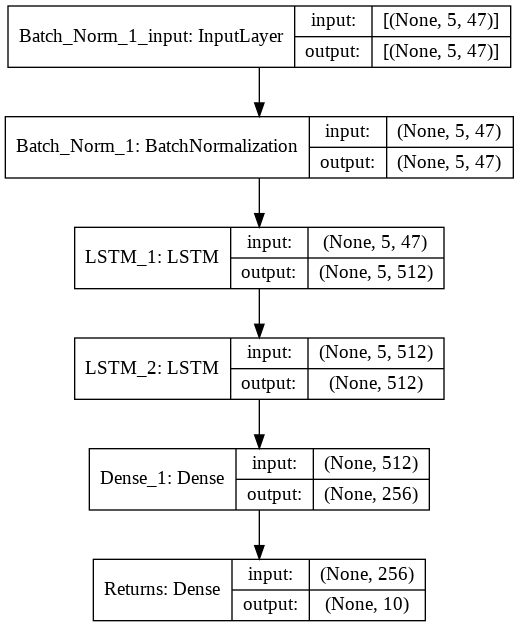
# Define the batch size:  
batch\_size = 512  
  
# Create an instance of the WindowGenerator object:  
my\_window = WindowGenerator(input\_width=window\_size,label\_width=1,shift=1,train\_df=train\_df,val\_df=val\_df,  
 test\_df=test\_df,label\_columns=label\_cols,  
 batch\_size=batch\_size,shuffle=True)  
  
# Print the shapes for one batch of each sub dataset:  
for example\_inputs, example\_labels in my\_window.train.take(1):  
 print("Train input shape:",example\_inputs.shape)  
 print("Train target shape:",example\_labels.shape)  
for example\_inputs, example\_labels in my\_window.val.take(1):  
 print("Validation input shape:",example\_inputs.shape)  
 print("Validation target shape:",example\_labels.shape)  
for example\_inputs, example\_labels in my\_window.test.take(1):  
 print("Test input shape:",example\_inputs.shape)  
 print("Test target shape:",example\_labels.shape)

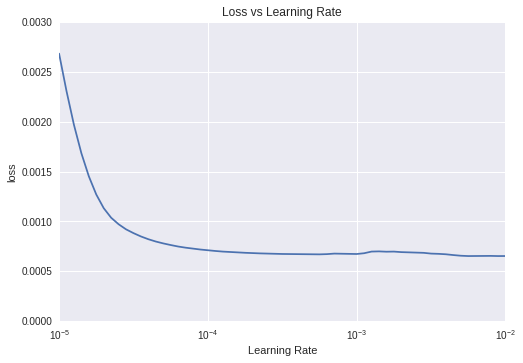
Train input shape: (512, 5, 47)  
Train target shape: (512, 1, 10)  
Validation input shape: (512, 5, 47)  
Validation target shape: (512, 1, 10)  
Test input shape: (351, 5, 47)  
Test target shape: (351, 1, 10)

The next step is to create the model. For this I'll use the [TensorFlow python API](https://www.tensorflow.org/api_docs/python/tf) to create the *Recurrent Neural Networks'* architecture.

from tensorflow.keras.layers import Dense, Conv1D, LSTM, GRU, Input, BatchNormalization  
from tensorflow.keras.models import Model, Sequential  
from tensorflow.keras.callbacks import ModelCheckpoint  
from tensorflow.keras.models import load\_model  
  
# # Model architecture creation:  
# model\_1 = Sequential([  
# BatchNormalization(  
# input\_shape = (window\_size,num\_features),  
# name = 'Batch\_Norm\_1'),  
# LSTM(512,return\_sequences=True,name='LSTM\_1'),  
# # BatchNormalization(),  
# LSTM(512,name='LSTM\_2'),  
# # BatchNormalization(momentum=0.8),  
# Dense(256,activation='relu',name='Dense\_1'),  
# Dense(n\_stocks,name='Returns')  
# ])  
  
# Learning Rate Schedule: used to decide the learning rate:  
# # lr\_schedule = tf.keras.callbacks.LearningRateScheduler(  
# # lambda epoch: 1e-8\*10\*\*(epoch/20))  
  
# Checkpoint callback to save the model:  
# checkpont\_rnn = ModelCheckpoint(  
# filepath='model\_1\_rnn',  
# save\_weights\_only=False,  
# save\_freq = 'epoch',  
# monitor = 'val\_loss',  
# save\_best\_only = True,  
# verbose = 0)  
  
# # Define optimizer  
# optimizer = tf.keras.optimizers.Adam(lr=1e-3)  
  
# # Compile Model  
# model\_1.compile(  
# loss=tf.keras.losses.Huber(),  
# metrics=[tf.metrics.RootMeanSquaredError(),'mae'],  
# optimizer=optimizer)  
  
# # Train model  
# history = model\_1.fit(  
# my\_window.train,  
# validation\_data=my\_window.val,  
# epochs=100)  
# model\_1.summary()

Because of the computation required for the training of a neural net, I used [Google Colab](https://colab.research.google.com/notebooks/welcome.ipynb?hl=es), in particular the GPU hardware accelerator capability. Thus, I trained and saved the model above, and loaded here to continue with the exercise



To decide which learning rate to use, I used a learning rate schedule to see which produced a smooth and fast minimization of the loss function. The results of this exercise can be seen below: 

I chose a learning rate of 0.001 as a result.

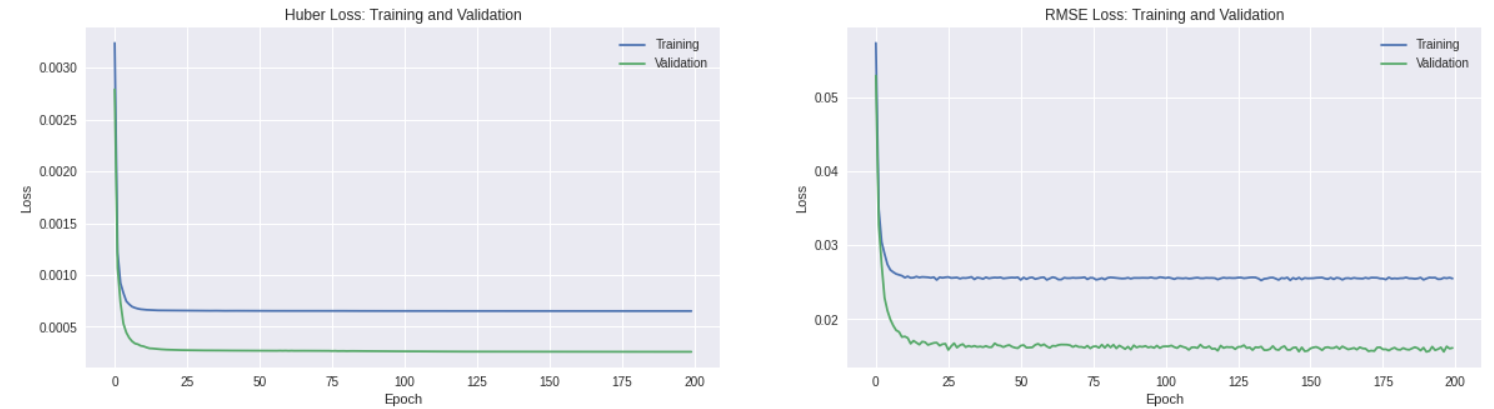
model\_1 = load\_model('rnn\_model\_1')

Below you can see the architecture I used for this model, that starts with a 1-dimension convolutional layer, followed by two recurrent layers with *Long Short Term Memory* cells, a normalization layer and ends with two dense layers, one with the a [ReLU (*Rectified Linear Unit*)](<https://en.wikipedia.org/wiki/Rectifier_(neural_networks)> and the output layer, which returns the 10 prediction of returns.

model\_1.summary()

Model: "sequential\_3"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Layer (type) Output Shape Param #   
=================================================================  
Batch\_Norm\_1 (BatchNormaliza (None, 5, 47) 188   
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
LSTM\_1 (LSTM) (None, 5, 512) 1146880   
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
LSTM\_2 (LSTM) (None, 512) 2099200   
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Dense\_1 (Dense) (None, 256) 131328   
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Returns (Dense) (None, 10) 2570   
=================================================================  
Total params: 3,380,166  
Trainable params: 3,380,072  
Non-trainable params: 94  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In the graphs below you can see the training loss (in this case I used the [Huber loss](https://en.wikipedia.org/wiki/Huber_loss), which is less sensitive to outliers). I used also the common metric *Root Mean Squared Errors (RMSE)*, which is in the right plot.



# plt.figure(figsize=(20,5))  
# plt.subplot(1,2,1)  
# plt.plot(history.history['loss'])  
# plt.plot(history.history['val\_loss'])  
# plt.xlabel('Epoch')  
# plt.ylabel('Loss')  
# plt.legend(['Training','Validation'])  
# plt.title('Huber Loss: Training and Validation')  
# plt.subplot(1,2,2)  
# plt.plot(history.history['root\_mean\_squared\_error'])  
# plt.plot(history.history['val\_root\_mean\_squared\_error'])  
# plt.xlabel('Epoch')  
# plt.ylabel('Loss')  
# plt.legend(['Training','Validation'])  
# plt.title('RMSE Loss: Training and Validation')

Finally, I evaluate the model for the lattest data, which is in the test set:

model\_1.evaluate(my\_window.test)

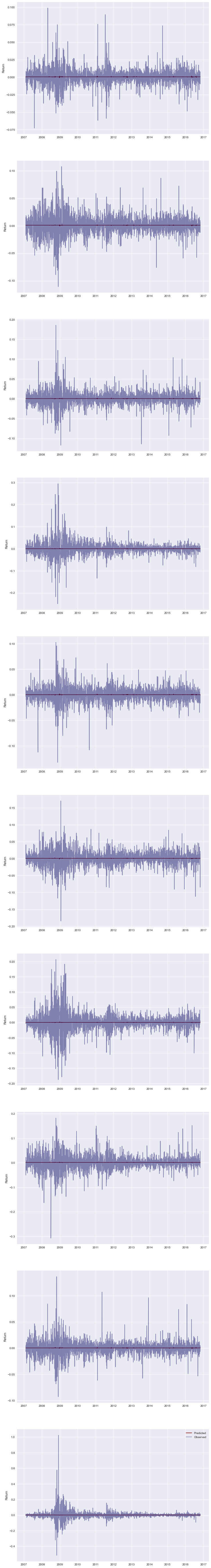
1/1 [==============================] - 2s 2s/step - loss: 0.0010 - root\_mean\_squared\_error: 0.0319 - mae: 0.0204

[0.0010197722585871816, 0.031933870166540146, 0.02037873864173889]

Now that I have the model let's see it's results compared with the real returns for each dataset (train, val, test):

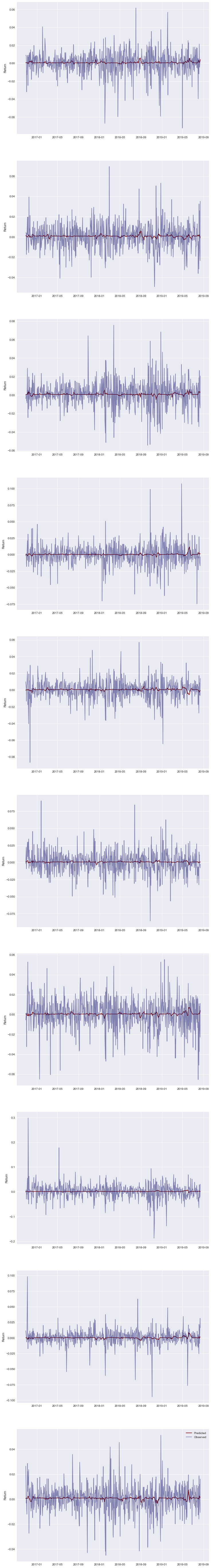
### ***Training Results***

# Create new WindowGenerator with the data not shuffled:  
my\_window\_2 = WindowGenerator(  
 input\_width=window\_size,  
 label\_width=1,  
 shift=1,  
 train\_df=train\_df,  
 val\_df=val\_df,  
 test\_df=test\_df,  
 label\_columns=label\_cols,  
 batch\_size=batch\_size,  
 shuffle=False)  
  
plot\_window(my\_window\_2.train,train\_df,window\_size,model\_1)



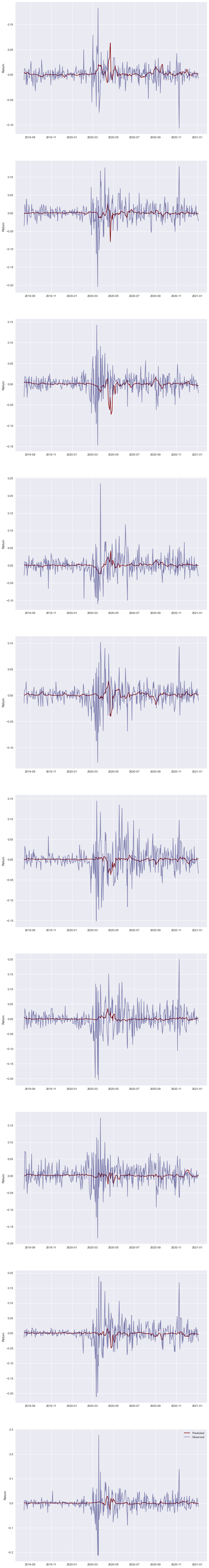
### ***Validation Results***

plot\_window(my\_window\_2.val,val\_df,window\_size,model\_1)



### ***Test Results***

plot\_window(my\_window\_2.test,test\_df,window\_size,model\_1)



## 4. Portfolio Optimization

Now that we have a way returns forecast model, we could use the forecasted returns to take possitions on the different assets that compose our portfolio, by allocating weights to the assets such that the Sharpe Ratio is maximized:

where is the return of the portfolio, is the risk-free rate (I used the 3-month Treasury bill daily rate as the risk-free rate).

columns = ['FR\_'+ticker for ticker in portfolio\_tickers]  
y\_train = model\_1.predict(my\_window\_2.train)  
y\_val = model\_1.predict(my\_window\_2.val)  
y\_test = model\_1.predict(my\_window\_2.test)  
y\_hat\_total = np.concatenate([y\_train,y\_val,y\_test],axis=0)  
ret\_hat\_df = pd.DataFrame(data=y\_hat\_total,index=total\_df.index[5:],columns=columns)  
rfr = add\_factors['3m\_rate'].agg(daily\_rate)  
ret\_hat\_df = ret\_hat\_df.merge(rfr.rename('rfr'),left\_index=True,right\_index=True,  
 how='left')  
# ret\_hat\_df.rolling(40).agg(lambda x: optimize\_portfolio(  
# returns = ret\_hat\_df[ret\_hat\_df.columns[:-1]],  
# rfr = ret\_hat\_df[ret\_hat\_df.columns[-1]]))  
opt\_weights = np.array([optimize\_portfolio(  
 returns = window[ret\_hat\_df.columns[:-1]],  
 rfr = window[ret\_hat\_df.columns[-1]]).x for window in ret\_hat\_df.rolling(40)])  
opt\_weights.shape

(3501, 10)

opt\_weights.shape

(3501, 10)

In the variable opt\_weights the optimal weights, given the predicted returns and the minimization function of the -Sharpe Ratio, are stored. This continual change assumes the investor is willing to reallocate daily the assets in the portfolio. This gives a demostration of how you could manage your portfolio by using Deep Learning to forecast returns, and based on those forecasts, manage your portfolio. Because of the demostration nature of this exercise, many available assets where leaved out, but with more computational power many more could be plugged in and analyzed and used in the management of the portfolio.

Finally, as can be seen in the results, the forecasting of the returns is not an easy endevor, due to the stocastic nature of them, resulting in difficult to predict jumps and falls. However, all of the work presented could be automated and improved, using NLP to measure the market's sentiment taking advantage of the live news feed (or Twitter information). You could also add image recognition, to exploit satellite data, or enhance the algorithm adding Reinforcement Learning.