University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

Gabriel Ogawa Cruz

# IBOVESPA index volatility forecasting using neural networks

São Paulo
November, 2018

# IBOVESPA index volatility forecasting using neural networks

Final bachelor's thesis submitted for
MAC0499 – Supervised Undergraduate Project.

Supervisor: Prof. Dr. Roberto Hirata Jr.

São Paulo
November, 2018

# Abstract

Volatility forecasting is a core part of risk management and options pricing in the stock exchange context. While using time series models is still common for such forecasting, this work will explore the accuracy of using a long short term memory (LSTM) recurrent neural network (RNN) for forecasting the volatility of Brazil's stock price index, IBOVESPA.

**Keywords:** volatility forecasting, neural networks, LSTM, RNN, DA-RNN.

# Contents

# List of Abbreviations

| | |
|---|---|
| B3 | Brazil's Stock Exchange |
| DA-RNN | Dual-Stage Attention-Based Recurrent Neural Network |
| D+X | X business days after the reference date |
| IBOVESPA | B3 stock prices index |
| LSTM | Long Short Term Memory |
| RNN | Recurrent Neural Network |
| VAR | Value at Risk |
| Vol | Volatility |

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In the financial market *volatility* is a term that frequently shows up. It can be defined as the measure of a financial instrument's price variation tendency over time, or a measure of price dispersion. For instance, an instrument with high volatility is prone to having sharp price movements much more often than a low volatility one. Being able to calculate and forecast an instrument's volatility is very desirable, as it has many uses such as: risk management, where volatility plays a main role in calculating measures like VAR (Value At Risk) [Jor00], a measure of how much one might lose given a set of investments, where more volatile assets mean a higher risk of loss; pricing options, contracts where the option seller can sell a buyer the right of buying or selling an instrument to the option seller in a future date (maturity), at a price (strike) set in the present, meaning that if the instrument is volatile there's a higher probability the price of the instrument at the option's maturity date will be better than the agreed upon strike, which in turn makes the option more valuable in the present, meaning volatility plays a pivotal role in pricing options [BS73] and forecasting or estimating an instruments volatility until its option's maturity date enables accurate option pricing; and for possibly being a measure of market health, as it has been found that often when the market volatility is high or quickly climbing the market can be said to be in poor health, while the opposite happens when volatility is low [Eas].

Given its uses, accurate volatility forecasting is highly desirable and studied, there are many published works on the matter such as some on general models evaluations and comparisons ([Roh07], [BF96], [AB98]), some focused on the commonly used ARCH (autoregressive conditional heteroskedasticity) time series models like GARCH ([BM96], [FVD], [HL]) and some introducing the use of neural networks for forecasting ([HI04], [DK]). Most of the published works are aimed towards US and European markets, which are in general more well behaved in comparison to emerging markets to a degree that very likely affects the accuracy of the models.

This works proposes using a recurrent neural network, more specifically a long short term memory network for forecasting an emerging market's index volatility using historical trading data of both the index itself and correlated series, based on the notion that such a network could be trained without much data manipulation and still produce forecasts as accurate as, or even more accurate than the more traditional ARCH models [GKD11].

IBOVESPA [BMF], Brazil's stock price index depicted in Figure 1.1, was chosen as the emerging market index of which the volatility will be forecast by a neural network, as it represents the overall situation of Brazil's stocks and as a not trivial time series to forecast, given the country's political and economic instability and incidents over the years. By using relevant related time series such as interest rates, stock prices and commodities prices when training the network, it is expected that complex relationships and patterns will be detected, enhancing the results accuracy. Even though the IBOVESPA index is being used in this work, it is also expected that the same methods could be applied to other instruments in the same market, given enough consideration when picking the input series.

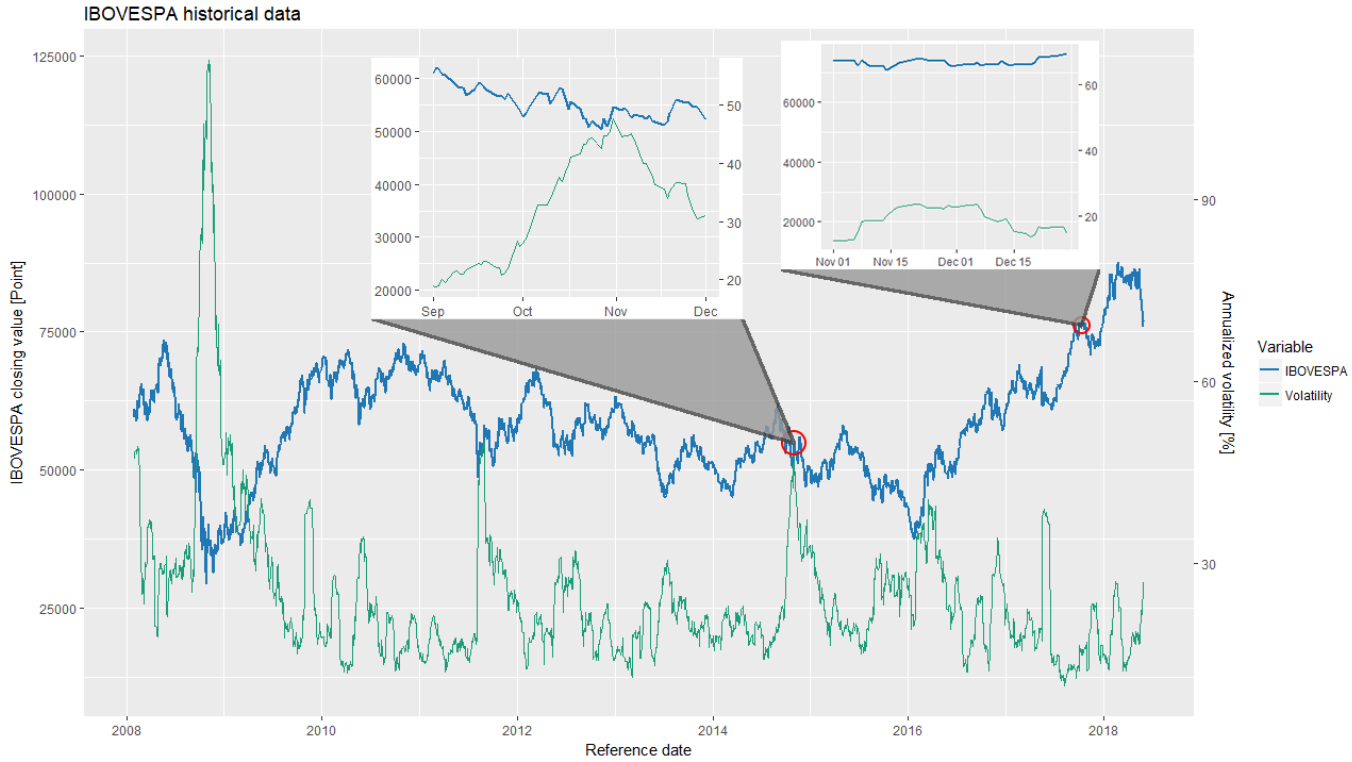


**Figure 1.1:** *IBOVESPA historical data and its volatility series. A period of high volatility and another of low volatility are shown in the zoomed cut-ins.*

The next section, Chapter 2, will be a brief description of neural networks, then more specifically of the recurrent neural network and the dual-stage attention based network that were implemented for the forecasts. In the section after that theoretical overview the practical implementation is shown in Chapter 3. Following that, we cover the series we aim to forecast, the dataset used for training and validating the neural network and which time series were used and why in Chapter 4. Then, in Chapter 5 the experiments details and the results of using the trained networks for forecasting the index volatility will be shown, including comparisons with other forecasting methods. Finally, conclusions drawn from the experiments results and further thoughts will be expressed in Chapter 6.

# Chapter 2

# Artificial Neural Network

## 2.1 Introduction

To put it simply, an **Artificial Neural Network** is a computational system that has the capacity to learn through experience. An artificial network is usually composed by many interconnected **layers** of artificial neurons, shown in Figure 2.1, which are represented as functions that generally take in many inputs, sum them, apply an activation function and then feed this output to other neurons, an structure modeled after biological neurons. In other words, a network can be seen as a set of algorithms that use a composition of many different functions to find patterns in raw data and use those to complete a task. For instance, in our case the network is supposed to find patterns in historical prices series and use them to predict a number, the index volatility forecast.

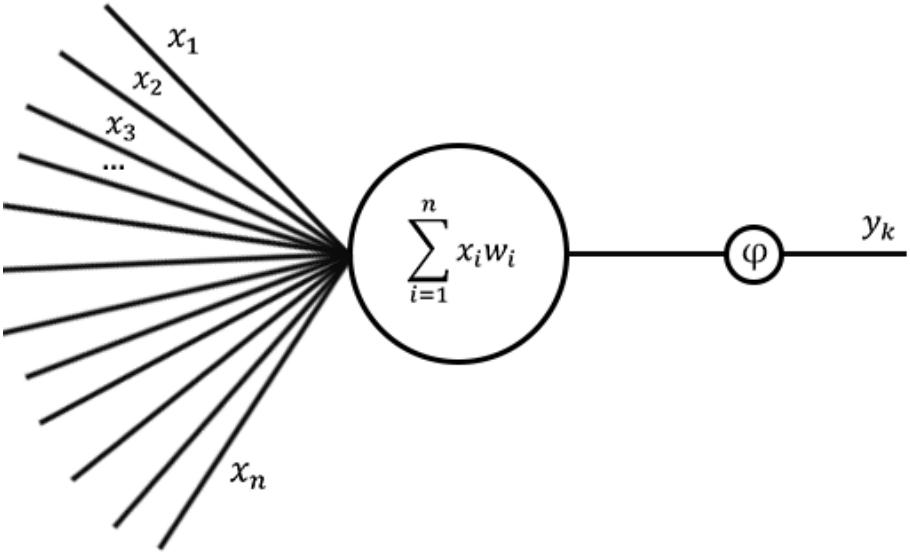


**Figure 2.1:** *Generic artificial neuron k representation. It takes the inputs $x_1 \dots x_n$, sums them with weights $w_1 \dots w_n$, and passes them through an activation function $\varphi$ to generate an output $y_k$.*

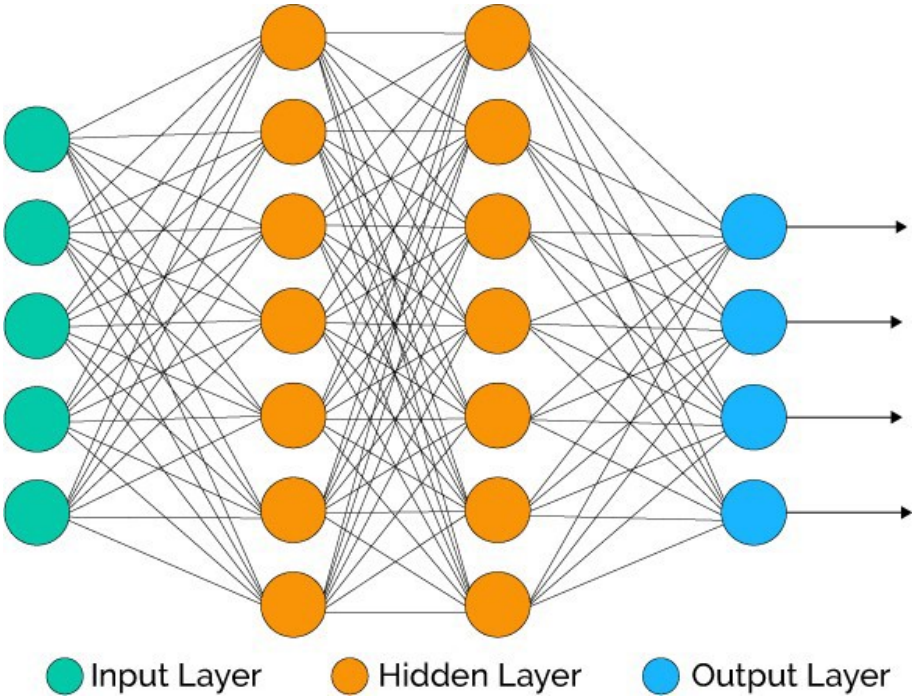


**Figure 2.2:** *Artificial Neural Network architecture example, circles represent artificial neurons. [McD]*

If we define the network's task as being finding an approximation to some function $f$, the network itself can be simplified to being a chain of function applications that given an input $x$ returns a value $y^*$ that approximates $f(x)$, such that, as an example, $y = f^{(*)}(x;\theta) = f^{(3)}(f^{(2)}(f^{(1)}(x;\theta)))$. These functions compose the network's **layers**, the outermost being the **output layer** because it returns the wanted output value, while the others are called **hidden layers**, this general architecture is exemplified in Figure 2.2. **Training** the network means finding the parameters $\theta$ (like the weights $w$ in Figure 2.1) that best approximate $f^{(*)}(x;\theta)$ to $f(x)$[GBC16].

Since our goal is to obtain index volatility forecasts based on various time series of correlated stocks, commodities and indicators, a **Recurrent Neural Network**, or **RNN** for short, is a good fit for the task. RNNs are neural networks specialized for processing sequences of values by sharing parameters through deep computational graphs in which each member of the output is a function of the previous members of the output[GBC16]. In this chapter we'll cover the way a RNN works, as well as the RNN types chosen for our problem, while further details on how neural networks in general work and are trained can be found in works such as [GBC16], [B+95], [RM99], [RHW86], [Sch15] and [BSF94].

## 2.2 Recurrent Neural Network

A **Recurrent Neural Network**, **RNN**, is, simply put, a neural network for processing sequences of variable length by sharing weights throughout time steps, allowing for information that appears at different time steps yet holds the same importance to have a consistent weight. Generally, given a sequence $X \in \mathbb{R}^N$ containing $x^{(t)}$ where $t$ ranges from 1 to $N$ time steps, an RNN will apply a function $f$ at each time step state $h^{(t)} \in \mathbb{R}^m$, where $m$ is the hidden state size, to obtain the state $h^{(t+1)}$, as illustrated in Figure 2.3, $f$ utilizes the same parameters $\theta \in \mathbb{R}^m$ for all time steps and also incorporates the $x^{(t)}$ signal. It should be noted the input can also be generalized to $k$ sequences with $N$ time steps such that $X \in \mathbb{R}^{kXN}$ and $x^{(t)} \in \mathbb{R}^k$ represents the values of $k$ series at time $t$. The recurrence can be represented by:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \tag{2.1}$$

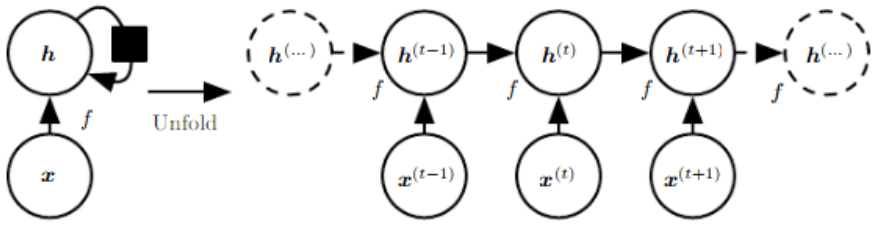


**Figure 2.3:** *Unfolded computational graph illustrating the application of $f$ over time steps. [GBC16]*

Training the network means trying to find the parameters $\theta$ such that from the states $h^t$ it's possible to generate an output that approximates the target series $y$ we're trying to predict or model. That is done by attempting to minimize a chosen loss criteria $L$ that compares the network outputs $o$ to target series values used for training. As most loss functions applied to neural networks are non-convex, training is usually done by iterative gradient-based optimizers that use gradient values to decrease loss. A generic RNN can be represented as the graphs in Figure 2.4.

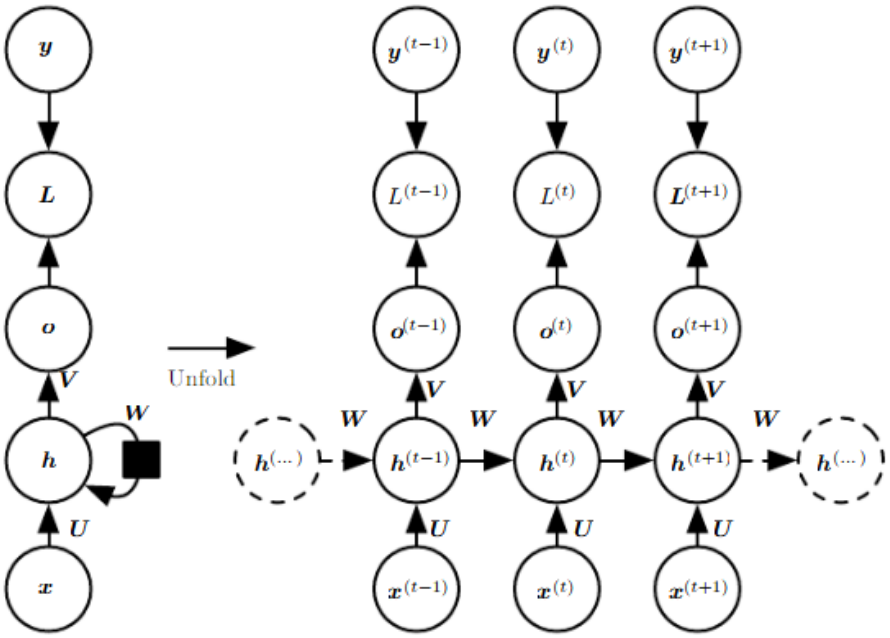


**Figure 2.4:** *Computational graphs for a generic RNN. [GBC16]*

In the example, the parameters $\theta$ are represented as weight matrices $W \in \mathbb{R}^{mXm}$, $U \in \mathbb{R}^{mXk}$ and $V \in \mathbb{R}^{mXm}$ and bias vectors $b, c \in \mathbb{R}^m$, and the state updates (forward pass) for each time step $t$ are applied following the equations:

$$a^{(t)} = b + Wh^{(t-1)} + Ux^t, \tag{2.2}$$

$$h^{(t)} = f(a^{(t)}), \tag{2.3}$$

$$o^{(t)} = c + Vh^t \tag{2.4}$$

As mentioned before, in order to minimize the chosen loss function for training the network gradients are used in the learning process. This means gradients for at least the cost function will have to be computed, which is usually done by a back-propagation algorithm. For a RNN like the one described above, computing the gradients is an expensive task because it has to be done for all past time steps for each time step, and that also brings up the issue of vanishing or exploding gradients and difficulty capturing long-term dependencies due to very small weights being given to long-term

states. There are many RNN architectures built to mitigate those issues, the one chosen for this work as the long short-term memory (LSTM) model for its success in many applications[GBC16].

The LSTM network is a gated RNN, illustrated in Figure 2.5, made to create paths through time that stop gradients from vanishing or exploding. Those paths are created dynamically through weights tuning based on the input, and the addition of gates allow for control of past values influence over state updates, minimizing the issue of long-term dependencies.

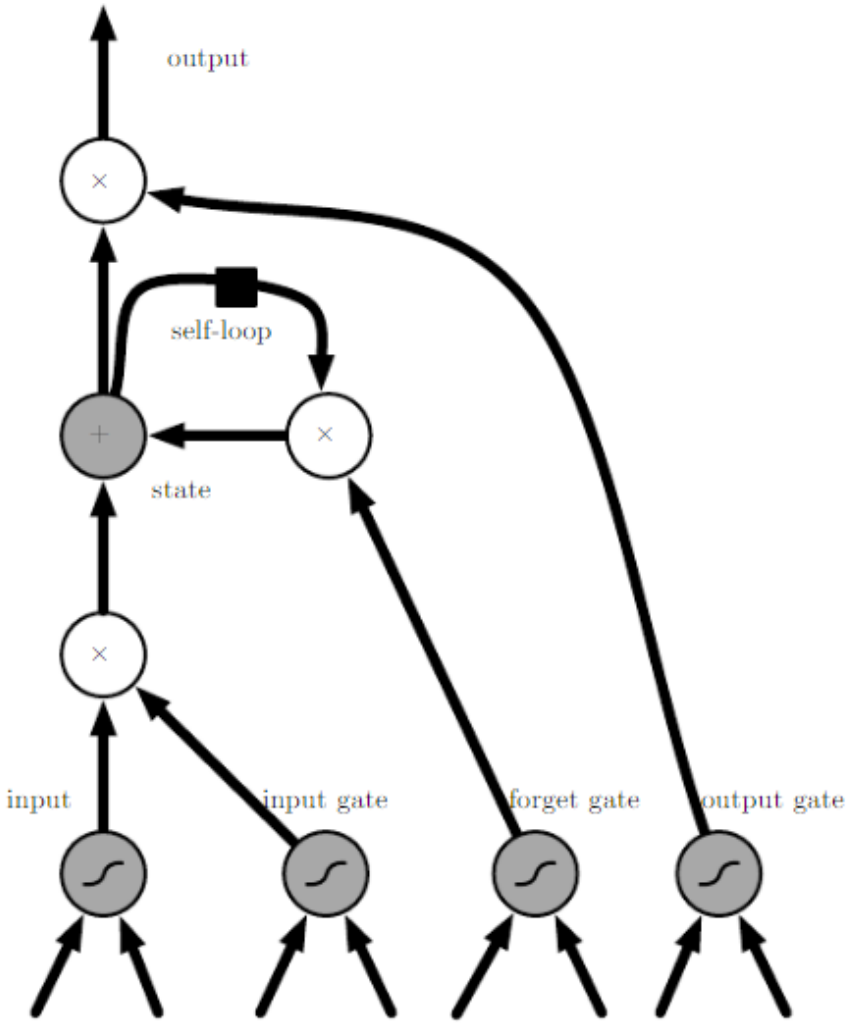


**Figure 2.5:** *LSTM cell block diagram. Cells replace the hidden units from the previously seen recurrent network. [GBC16]*

Here the state unit $s_i^{(t)}$, for time step $t$ and cell $i$, will accumulate information from the past up to a certain point, a point that is controlled by the forget gate unit $f_i^{(t)}$ which is updated by the sigmoid:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right) \tag{2.5}$$

where $x^{(t)} \in \mathbb{R}^k$ is the current input vector, $h^{(t)} \in \mathbb{R}^m$ the current hidden layer vector with outputs of all cells, and $b^f \in \mathbb{R}^m$, $U^f \in \mathbb{R}^{mXk}$, $W^f \in \mathbb{R}^{mXm}$ the biases, input weights and recurrent weights for the forget gates. The state unit is updated with:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma \left( b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)} \right) \tag{2.6}$$

where $g_i^{(t)}$ represents the external input gate unit, computed by:

$$g_i^{(t)} = \sigma \left( b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)} \right) \tag{2.7}$$

similarly to the forget gate but with its own parameters. The output $h_i^{(t)}$ is then computed with an activation function applied over the state (for instance, $tanh$) and controlled by the output gate $q_i^{(t)}$ which also functions like the other gates:

$$h_i^{(t)} = tanh \left( s_i^{(t)} \right) q_i^{(t)}, \tag{2.8}$$

$$q_i^{(t)} = \sigma \left( b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)} \right) \tag{2.9}$$

By tuning all the gate parameters during training the network is then capable of controlling the time scale for dependencies over time, making the LSTM model much more robust for handling long-term dependencies. In this work's implementation of a LSTM network, discussed in Section 3.1 two LSTM units are stacked, meaning the first unit takes in the $x$ series input and generates an output $o$, which is then fed into the second unit as that units $x$ input. This allows for a more complex representation of the input information by capturing it at different time scales, improving accuracy for complex time series predictions such as this works objective.

## 2.3 DA-RNN

The **Dual-Stage Attention-Based Recurrent Neural Network** or **DA-RNN** for short is an alternative network architecture proposed by [QSC+17] that also aims to better capture long-term dependencies, like the LSTM network, and on top of that introduce an attention mechanism that selects the most relevant input information.

The DA-RNN is split into two stages, an encoder that selects the relevant information from the input and a decoder that tries to select relevant encoder hidden states through time. By also using a LSTM unit, the DA-RNN is then capable of selecting relevant input features and capturing long-term dependencies of time series. The attention mechanism in the encoder unit is constructed with:

$$e_i^{(t)} = v_e^\top tanh \left( W_e [h_i^{(t-1)}; s_i^{(t-1)}] + U_e x_i \right), \tag{2.10}$$

$$\alpha_i^{(t)} = \frac{exp(e_i^{(t)})}{\sum_{j=1}^{n} exp(e_j^{(t)})} \tag{2.11}$$

where $v_e \in \mathbb{R}^T$, $W_e \in \mathbb{R}^{TX2m}$ and $U_e \in \mathbb{R}^{TXT}$ are parameters to learn, $x_i \in \mathbb{R}^T$ is the $i$-th input series, $T$ is the amount of time-steps and $\alpha_i^{(t)}$ is the attention weight for the $i$-th input feature at time $t$. The attention weight is then passed through a softmax function to ensure they sum to one, and the relevant features can then be extracted with:

$$\tilde{x}^{(t)} = (\alpha_i^{(t)} x_i^{(t)})^\top. \tag{2.12}$$

The hidden state is then updated through a LSTM unit by feeding it $\tilde{x}^{(t)}$, as described in

Section 2.2. The output $h^{(t)}$ of the LSTM unit is then used as the input for the decoder step, which computes the output $\hat{y}_T$ using a second attention mechanism based on the decoder hidden states $d^{(t)} \in \mathbb{R}^p$ and the previous LSTM unit cell states $s^{(t)} \in \mathbb{R}^p$, as follows:

$$l_i^{(t)} = v_d^\top tanh\left(W_d[d_i^{(t-1)}; s_i^{(t-1)}] + U_d h_i\right), \tag{2.13}$$

$$\beta_i^{(t)} = \frac{exp(l_i^{(t)})}{\sum_{j=1}^{n} exp(l_j^{(t)})} \tag{2.14}$$

where $v_d \in \mathbb{R}^m$, $W_d \in \mathbb{R}^{mX2p}$ and $U_d \in \mathbb{R}^{mXm}$ are parameters to learn, and $\beta_i^{(t)}$ weights each encoder hidden state. This gives us context vectors $c^(t)$ given by:

$$c^{(t)} = \sum_{i=1}^{T} \beta_i^{(t)} h_i \tag{2.15}$$

which can then be combined with the past values of the target series to compute:

$$\tilde{y}^{(t-1)} = \tilde{w}^\top [y^{(t-1)}; c^{(t-1)}] + \tilde{b} \tag{2.16}$$

where $\tilde{w} \in \mathbb{R}^{m+1}$ and $\tilde{b} \in \mathbb{R}$ are parameters. The decoder hidden state is then updated using $\tilde{y}^{(t-1)}$ through the decoder's LSTM unit. By repeating the update steps, in the end $\hat{y}_T$ can be calculated by using the final decoder hidden state in a linear function:

$$\hat{y}_T = v_y^\top \left(W_y[d_T; c_T] + n_w\right) + b_v \tag{2.17}$$

where $W_y \in \mathbb{R}^{pX(p+m)}$, $b_w \in \mathbb{R}^p$, $v_y \in \mathbb{R}^p$ and $b_v \in \mathbb{R}$ are parameters, and $d_T \in \mathbb{R}^p$ and $c_T \in \mathbb{R}^m$ are the final decoder hidden state and context vector. With this, the network generates a target series prediction for time $T$ given input values $x$ until time $T$ and target series values until time $T-1$.

# Chapter 3

# Model Implementation

## 3.1 RNN

The way a RNN works was described in Section 2.2, what follows is the PyTorch implementation of a network that utilizes a stacked LSTM with two layers (Listing 3.1), meaning there's one LSTM that takes in the input and a second LSTM that takes in the output of the first and computes the final results, which are then passed through a linear transformation to obtain the desired values. PyTorch is a Python library for tensor based deep learning that greatly facilitates the implementation of neural networks and which has a seamless integration with CUDA, allowing for quick model training.

```
class LSTM(nn.Module):
  """
  Recurrent LSTM network
  """
  def __init__(self, features, hidden_size, num_layers = 2, output_size = 1):
    super(LSTM, self).__init__()

    self.lstm = nn.LSTM(features, hidden_size, num_layers)
    self.lin = nn.Linear(hidden_size, output_size)

  def forward(self, x):
    """
    Computes forward pass
    """
    x, states = self.lstm(x)
    seq_len, batch, hidden = x.shape
    x = x.view(seq_len * batch, hidden)
    x = self.lin(x)
    x = x.view(seq_len, batch, -1)
    return x
```

**Listing 3.1:** *LSTM model definition with PyTorch*

The model was then trained using the Adam optimizer [KB14], an efficient optimization algorithm that works well for our case as it is appropriate for noisy non-stationary objectives like the index volatility series. The loss criterion chosen was the SmoothL1Loss for being less sensitive to outliers, defined as: [PyT]

$$loss(x, y) = \frac{1}{n} \sum iz_i \tag{3.1}$$

Where $z_i$ is given by:

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1. \\ |x_i - y_i| - 0.5, & \text{otherwise.} \end{cases} \tag{3.2}$$

The model is trained over a training dataset and a validation dataset, with the training function seeking to minimize validation error. The code for training the model is shown in Listing 3.2.

```
def train_model(model, train_x, train_y, valid_x, valid_y, state_path = "
    lstm_state.pkl", epochs = 500):
  """
  Train a PyTorch model with the Adam optimizer by
  minimizing validation set error
  """

  best_valid_loss = float("inf")
  criterion = nn.SmoothL1Loss()
  optimizer = torch.optim.Adam(model.parameters(), lr = 1e-2)

  train_loss = []
  valid_loss = []
  for epoch in range(epochs):
    pred = model(train_x)
    loss = criterion(pred, train_y)
    v_pred = model(valid_x)
    v_loss = criterion(v_pred, valid_y)
    valid_loss.append(float(v_loss))
    train_loss.append(float(loss))
    if (float(v_loss) < best_valid_loss):
      msg = "\ntrain_loss = {:.3f} | valid_loss = {:.3f} | epoch = {:.1f}\n".
          format(float(loss),float(v_loss), float(epoch))
      torch.save(model.state_dict(), state_path)
      best_valid_loss = float(v_loss)
      print(msg, end="")
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
  return train_loss, valid_loss
```

**Listing 3.2:** *Model training function*

After testing various parameter configurations two RNN models were chosen to be presented in this work, with parameters as follows (unmentioned parameters follow PyTorch's defaults):

- RNN-1
  - Input features: 8
  - Hidden size: 16
  - LSTM layers: 2
  - Adam learning rate: 0.01
  - Epochs: 500
- RNN-2
  - Input features: 8
  - Hidden size: 32
  - LSTM layers: 2
  - Adam learning rate: 0.01
  - Adam AMSGrad [RKK18] variant utilized
  - Epochs: 500

To produce the forecasts, for each date in the test set, the best model state found during training (the one that minimizes validation error) is loaded and evaluated using the input up to a day before the current test date, as shown in Listing 3.3. This simulates real usage in which the model would

be used daily to produce a current forecast. In reality as time passes it's possible the network should be retrained, but for our purposes since the test set isn't too big retraining wasn't done and instead the model parameters found during initial training were used for all dates in the test set.

```
with open(answer_file, "w") as file:
  for i in range(test_start, len(raw_data)):
    # at each date the input goes up to
    # the previous date
    x_test = raw_x_test[:(i + 1), ]
    test_x = torch.from_numpy(x_test.reshape(-1, 1, 8))
    model = LSTM(8, 16).double()
    model.load_state_dict(torch.load(state_file + ".pkl"))
    model = model.eval()
    # The model returns the whole series but we're
    # only interested in the last value which would be
    # the forecast on the current date
    pred_test = model(test_x).view(-1).data.numpy()
    file.write("{}\n".format(pred_test[-1]))
```

**Listing 3.3:** *RNN-1 forecasts generation*

## 3.2 DA-RNN

The DA-RNN was also implemented in PyTorch, strongly following [Zuo17]'s implementation of the network described in Section 2.3. The encoder and decoder were implemented separately as follows:

```
class encoder_RNN(nn.Module):
  """
  DA-RNN encoder module
  """
  def __init__(self, features, hidden_size, T, num_layers = 1, output_size = 1):
    super(encoder_RNN, self).__init__()

    self.lstm = nn.LSTM(features, hidden_size, num_layers)
    self.lstm.flatten_parameters()
    self.lin = nn.Linear(2 * hidden_size + T - 1, output_size)
    self.T = T
    self.features = features
    self.hidden = hidden_size

  def forward(self, x):
    """
    Computes forward pass
    """

    curr_hidden = Variable(x.data.new(1, x.size(0), self.hidden).zero_())
    curr_cell_state = Variable(x.data.new(1, x.size(0), self.hidden).zero_())
    # input with attention weights applied
    x_atn = Variable(x.data.new(x.size(0), self.T - 1, self.features).zero_())
    # encoded hidden state
    x_enc = Variable(x.data.new(x.size(0), self.T - 1, self.hidden).zero_())

    for t in range(self.T - 1):
      # find e^{k}_{t}
      e = torch.cat((curr_hidden.repeat(self.features, 1, 1).permute(1, 0, 2),
        curr_cell_state.repeat(self.features, 1, 1).permute(1, 0, 2),
        x.permute(0, 2, 1)), dim = 2)
      # ensure attention weights sum to 1
      a = ff.softmax(self.lin(e.view(-1, self.hidden * 2 + self.T - 1)).view(-1,
          self.features), dim = 1)
       # driving series extracted according to attention weights
      x_atn_tmp = torch.mul(a, x[:, t, :])
```

```
36        # hidden state update through LSTM unit
37        _, state = self.lstm(x_atn_tmp.unsqueeze(0), (curr_hidden, curr_cell_state
              ))
38        curr_hidden = state[0]
39        curr_cell_state = state[1]
40
41        x_atn[:, t, :] = x_atn_tmp
42        x_enc[:, t, :] = curr_hidden
43
44      return x_atn, x_enc
```

**Listing 3.4:** *DA-RNN encoder module implementation*

In Listing 3.4's encoder module implementation, lines 29-31 compute Equation 2.10, line 33 computes the attention weights from Equation 2.11 and line 35 computes $\tilde{x}^{(t)}$ from Equation 2.12, which is then passed through the LSTM module on line 37.

```
1  class decoder_RNN(nn.Module):
2    """
3    DA-RNN decoder module
4    """
5    def __init__(self, dec_hidden, enc_hidden, T):
6      super(decoder_RNN, self).__init__()
7
8      self.lstm = nn.LSTM(1, dec_hidden)
9      self.lstm.flatten_parameters()
10     self.lin_in = nn.Sequential(nn.Linear(2 * dec_hidden + enc_hidden,
           enc_hidden),
11                                 nn.Tanh(),
12                                 nn.Linear(enc_hidden, 1))
13     self.lin_upd = nn.Linear(enc_hidden + 1, 1)
14     self.lin_upd.weight.data.normal_()
15     self.lin_out = nn.Linear(dec_hidden + enc_hidden, 1)
16     self.T = T
17     self.hidden = dec_hidden
18     self.enc_hidden = enc_hidden
19
20
21   def forward(self, x, y):
22     """
23     Computes forward pass
24     """
25
26     curr_hidden = Variable(x.data.new(1, x.size(0), self.hidden).zero_())
27     curr_context = Variable(x.data.new(1, x.size(0), self.hidden).zero_())
28
29     for t in range(self.T - 1):
30      # find l^{i}_{t}
31       l = torch.cat((curr_hidden.repeat(self.T - 1, 1, 1).permute(1, 0, 2),
32         curr_context.repeat(self.T - 1, 1, 1).permute(1, 0, 2),
33         x), dim = 2)
34       # attention weights that sum 1
35       b = ff.softmax(self.lin_in(l.view(-1, 2 * self.hidden + self.enc_hidden)).
             view(-1, self.T - 1), dim = 1)
36       context_step = torch.bmm(b.unsqueeze(1), x)[:, 0, :]
37       if t < self.T - 1: # not last
38         # target extracted with attention weights
39         y_new = self.lin_upd(torch.cat((context_step, y[:, t].unsqueeze(1)), dim
               = 1))
40         # hidden state update through LSTM unit
41         _, state = self.lstm(y_new.unsqueeze(0), (curr_hidden, curr_context))
42         curr_hidden = state[0]
43         curr_context = state[1]
44
45     # last step isn't a hidden state update, it just produces the result
```

```
46      prediction = self.lin_out(torch.cat((curr_hidden[0], context_step),
47                                          dim = 1))
48
49      return prediction
```

**Listing 3.5:** *DA-RNN decoder module implementation*

In Listing 3.5's decoder module implementation, lines 31-33 compute Equation 2.13, line 35 computes the attention weights from Equation 2.14, line 36 finds the context vectors from Equation 2.15, and line 39 computes $\tilde{y}^{(t-1)}$ from Equation 2.16. After all states are updated over the time steps, the output is calculated in line 46 following Equation 2.17.

The network was trained using batches of 128 time steps each and minimizing the mean error over batches, with the Adam optimizer and SmoothL1Loss function as described for the RNN model. This time the dataset was only split between training and testing, only the last 60 days of the sample were excluded for testing while the rest was used for training. The implemented DA-RNN model training function is shown in Listing 3.6.

```
1  def train_model(encoder, decoder, train_x, train_y, T, features,
2   enc_state_path = "encoder_state.pkl", dec_state_path = "decoder_state.pkl",
        epochs = 500):
3    """
4    Train a DA-RNN model
5    """
6
7    best_loss = float("inf")
8    criterion = nn.SmoothL1Loss()
9    optimizer_enc = torch.optim.Adam(encoder.parameters(), lr = 1e-2)
10   optimizer_dec = torch.optim.Adam(decoder.parameters(), lr = 1e-2)
11   tsteps = int(train_x.shape[0] * 0.7)
12
13   for epoch in range(epochs):
14
15     idxs = np.array(range(tsteps - T))
16
17     idx = 0
18     losses = []
19
20     while (idx < tsteps):
21       train_idxs = idxs[idx:(idx + 128)]
22       tx = np.zeros((len(train_idxs), T - 1, features))
23       ty = np.zeros((len(train_idxs), T - 1))
24
25       for i in range(len(train_idxs)):
26         tx[i, :, :] = train_x[train_idxs[i]:(train_idxs[i] + T - 1), :]
27         ty[i, :] = train_y[train_idxs[i]:(train_idxs[i] + T - 1)]
28
29       tx_tensor = Variable(torch.from_numpy(tx))
30       ty_tensor = Variable(torch.from_numpy(ty))
31       target_y = Variable(torch.from_numpy(train_y[train_idxs + T]))
32       target_y = target_y.view(-1, 1)
33
34       weighted, encoded = encoder(tx_tensor)
35       pred = decoder(encoded, ty_tensor)
36       loss = criterion(pred, target_y)
37       losses.append(float(loss))
38
39       optimizer_enc.zero_grad()
40       optimizer_dec.zero_grad()
41       loss.backward()
42       optimizer_enc.step()
43       optimizer_dec.step()
44       idx += 128
45
46     if (np.mean(losses) < best_loss):
```

```
47          torch.save(encoder.state_dict(), enc_state_path)
48          torch.save(decoder.state_dict(), dec_state_path)
49          best_loss = np.mean(losses)
50          print(msg, end="")
```

**Listing 3.6:** *DA-RNN training function*

Producing forecasts with the DA-RNN isn't as straightforward as it was for the RNN, because of the fact that the decoder network requires past values for the target series. In order to train the model to forecast D+N value for volatility the target series used was the volatility series lagged by N days, which in turn means that for a current date the target series values for the previous N - 1 days aren't yet available, as illustrated in Figure 3.1. To remedy that, those values have to be populated with the networks own predictions. A possibly more accurate way of handling this issue would be to have a different network trained for forecasting each of 1 to N days forecast horizons, but as that is very expensive computationally only the N-day network was utilized for the whole forecast.

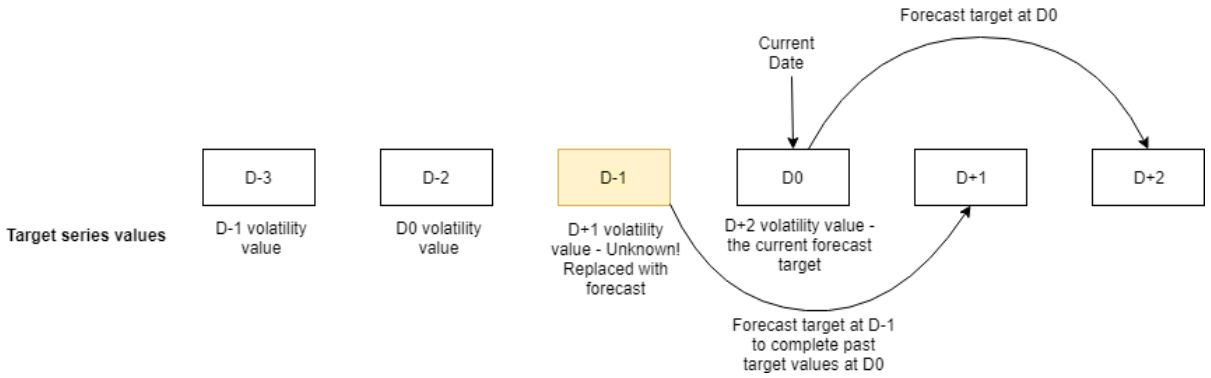


**Figure 3.1:** *Example of the DA-RNN forecasting issue for a 2-days prediction*

Listing 3.7 shows how the forecasts were computed given the saved encoder and decoder states from the best network found during training. Lines 11 to 43 deal with the above mentioned issue of needing N - 1 predictions to use as an input for the D+N forecast, by using the network multiple times generating a prediction for each unavailable date and appending it to the previous y values series. After that, lines 46-73 use the appended y series to generate the final prediction.

```
1  with open(answer_file, "w") as file:
2    for i in range(test_start, len(raw_data)):
3
4      # Values for the last N - 1 days are needed
5      # to predict the current date, since the
6      # target series uses lagged values the
7      # real values until D - (N - 1) aren't available
8      # for decoder input, meaning each prediction will
9      # carry into it N predictions
10
11     j = N - 1
12     curr_index = i - N
13     available_y = raw_y_test[:curr_index, ]
14
15     while j > 0:
16       x_test = raw_x_test[:curr_index, ]
17       encoder = encoder_RNN(8, 32, 10).double()
18       decoder = decoder_RNN(32, 32, 10).double()
19       encoder.load_state_dict(torch.load(enc_state_file))
20       decoder.load_state_dict(torch.load(dec_state_file))
21       encoder = encoder.eval()
22       decoder = decoder.eval()
23       tsteps = int(x_test.shape[0] * 0.7)
```

```
      y = np.zeros(x_test.shape[0] - tsteps)

      k = 0
      while (k < len(y)):
        idxs = np.array(range(len(y)))[k:(k + 128)]
        x = np.zeros((len(idxs), 9, x_test.shape[1]))
        yh = np.zeros((len(idxs), 9))

        for l in range(len(idxs)):
          x[l, :, :] = x_test[range(idxs[l] + tsteps - 10, idxs[l] + tsteps - 1)
              , :]
          yh[l, :] = available_y[range(idxs[l] + tsteps - 10, idxs[l] + tsteps -
               1)]

        yh = Variable(torch.from_numpy(yh))
        _, encoded = encoder(Variable(torch.from_numpy(x)))
        y[k:(k + 128)] = decoder(encoded, yh).data.numpy()[:, 0]
        k += 128

      curr_index += 1
      available_y = np.append(available_y, np.expand_dims(y[-1], axis = 1), axis
          =0)
      j -= 1


    x_test = raw_x_test[:(i + 1), ]
    y_test = available_y

    encoder = encoder_RNN(8, 32, 10).double()
    decoder = decoder_RNN(32, 32, 10).double()
    encoder.load_state_dict(torch.load(enc_state_file))
    decoder.load_state_dict(torch.load(dec_state_file))
    encoder = encoder.eval()
    decoder = decoder.eval()

    tsteps = int(x_test.shape[0] * 0.7)
    y = np.zeros(x_test.shape[0] - tsteps)

    k = 0

    while (k < len(y)):
      idxs = np.array(range(len(y)))[k:(k + 128)]
      x = np.zeros((len(idxs), 9, x_test.shape[1]))
      yh = np.zeros((len(idxs), 9))

      for j in range(len(idxs)):
        x[j, :, :] = x_test[range(idxs[j] + tsteps - 10, idxs[j] + tsteps - 1),
            :]
        yh[j, :] = y_test[range(idxs[j] + tsteps - 10, idxs[j] + tsteps - 1)]

      yh = Variable(torch.from_numpy(yh))
      _, encoded = encoder(Variable(torch.from_numpy(x)))
      y[k:(k + 128)] = decoder(encoded, yh).data.numpy()[:, 0]
      k += 128

    file.write("{}\n".format(y[-1]))
```

**Listing 3.7:** *DA-RNN forecasts generation*

# Chapter 4

# Dataset

The dataset consists of daily trading data for stocks, futures and options dating from 2008-01-02 to 2018-06-01, and diverse economic indicators as published by B3, Brazil's stock exchange on their public official website[1]. For ease of access, all the data was parsed from the text and XML files B3 publishes and inserted into a InfluxDB[2] time series database. While InfluxDB is more often used for higher frequency time-stamped data, the time, value and tags structure is very convenient for storing trading data, and the SQL-like queries make data retrieval simple. Both the dataset and the scripts used to build it are available publicly.[3]

## 4.1 Contents breakdown

The data was split into five InfluxDB measures:

- Stocks
- Options
- Futures
- Indic (Economic Indicators such as commodities and currency prices)
- Interest Rate

The stocks, options and futures measures contain open, maximum, minimum and close prices, trades quantities, traded quantities and trade volume for every listed contract in Brazil's stock exchange. The economic indicator measure contains all indicators B3 publishes, plus crude oil barrel prices (WTI).

The interest rate measure in particular was built from the interest rate curve (PRE) B3 publishes, which is calculated based on interest rate futures trading and/or offer books, according to B3's pricing manual [B3]. For each reference date in the measure, the published curve from that day was used to compute the 1, 3 and 5 years interest rates via exponential flat rate forward interpolation.

For each time series in each measure their log-returns and historical volatility were calculated as:

$$log\text{-}ret(t) = ln\left(\frac{value(t)}{value(t-1)}\right) \tag{4.1}$$

$$vol(t) = stddev(log\text{-}ret[t-20, t-1]) \tag{4.2}$$

[1] http://www.b3.com.br/en_us/

[2] Appendix A for further information.

[3] https://github.com/ogaw4/MAC0499-Ibovespa-Volatility

That means for each reference date in each time series there is a recent volatility measure given by the standard deviation of the past 20 business days logarithmic returns. While there are many different ways to measure volatility, the standard deviation suffices for the objectives of this work given that it is reasonable to expect that if the neural network can be trained to forecast this measure of volatility, then it should be able to be trained to forecast other measures, as all of them are supposed to be a measure of the dispersion of an assets returns. At some points the annualized volatility will be shown on graphs, it is calculated as $annualized\_vol(t) = vol(t) * \sqrt{252}$.

## 4.2 IBOVESPA

The IBOVESPA stock price index is composed by a weighted sum of the stocks that represent 85% of total trading on B3's stock exchange [BMF]. As such, the IBOVESPA is very representative of the overall stock market in Brazil, specially considering that B3 is the only stock exchange in the country. One of our goals is to predict how the index volatility will behave in the future, which by extension would be a measure of how volatile the market as a whole is predicted to be. The index composition has changed over the years as the stocks that compose it and their weights are adjusted based on free float market value, which means it's possible that its volatility behavior has changed as well. However, the stocks that account for the highest weights in the index have consistently been so for the period this dataset contains.

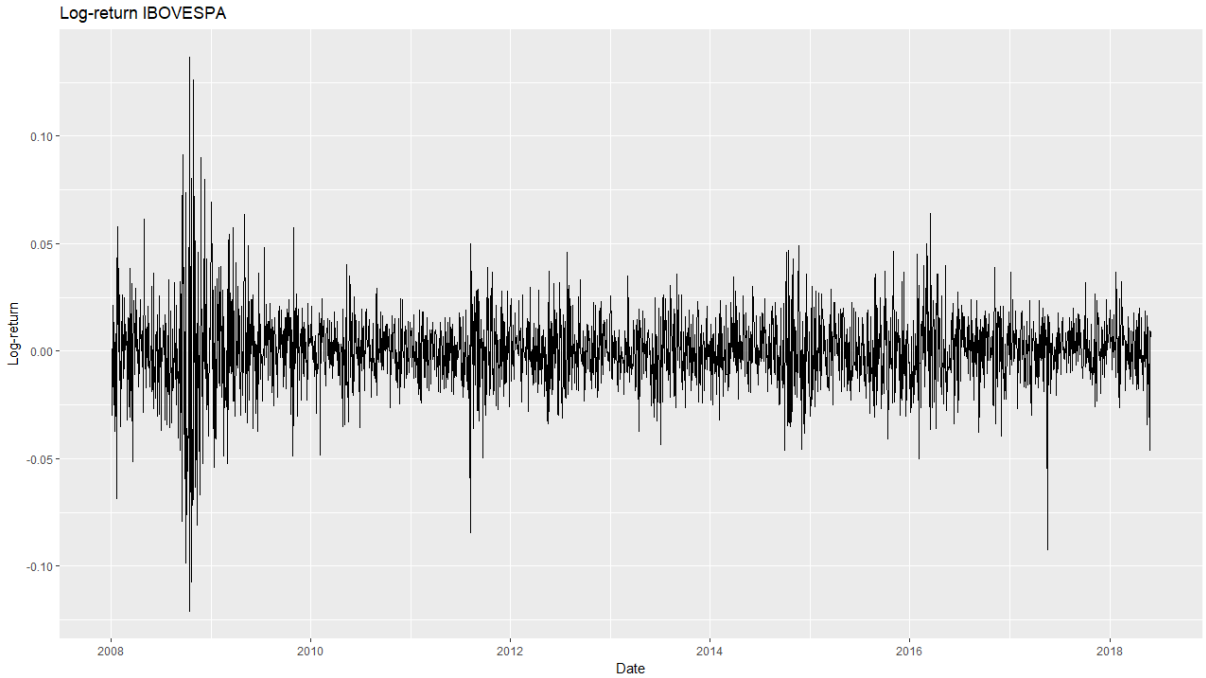


**Figure 4.1:** *IBOVESPA Log-returns.*

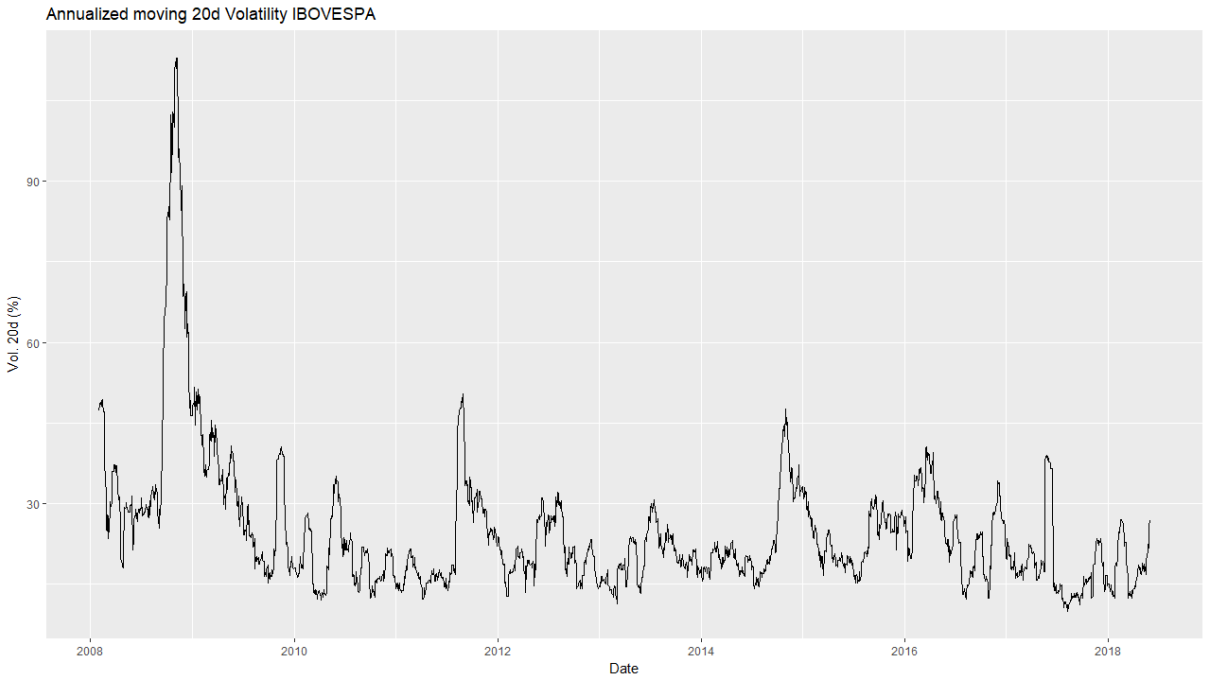


**Figure 4.2:** *IBOVESPA historical volatility.*

In Figure 4.1 the IBOVESPA log-returns are shown, which are then used to calculate IBOVESPA's volatility shown in Figure 4.2. We can see the IBOVESPA volatility itself is rather volatile, with some peaks that could be outliers in our data. While it is possible the neural network could be trained even with the outliers in the sample to a usable degree of precision, for the sake of comparisons it was chosen to also study the outliers in the data.

For outlier detection and adjustments the Chen and Liu iterative outlier estimation method [CLD03] was used, as implemented by the ***tsoutliers*** R package.

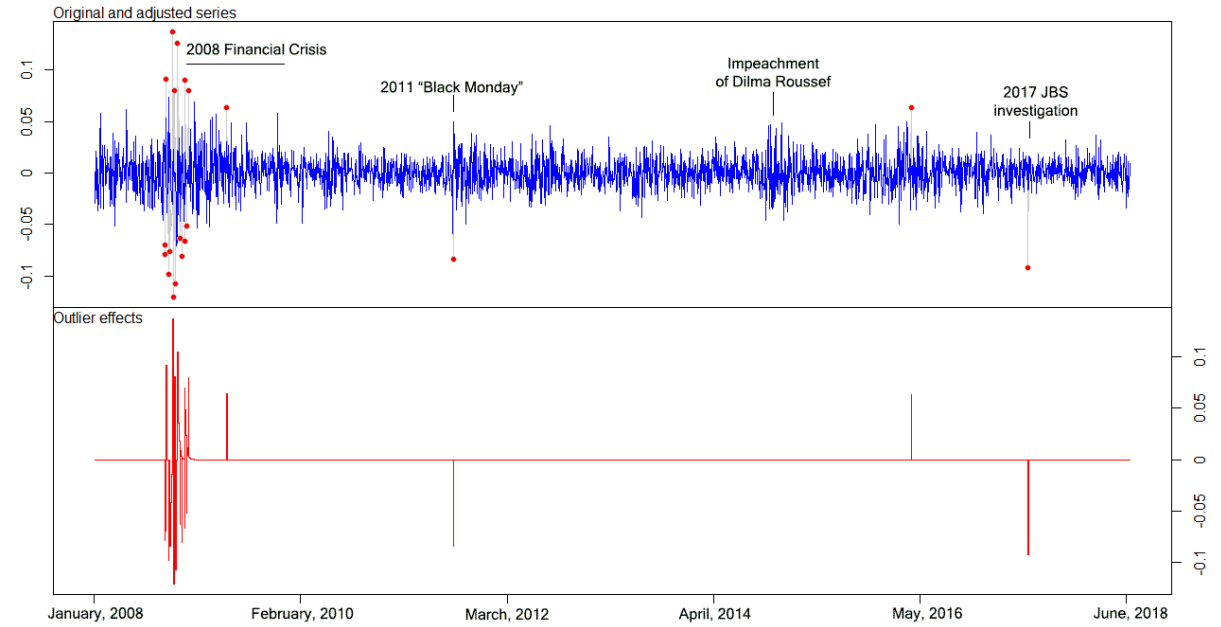


**Figure 4.3:** *IBOVESPA log-returns outliers.*

In Figure 4.3 we see the results of using the ***tsoutliers*** package to detect outliers on IBOVESPA log-returns, with the adjusted series in dark blue and outliers pointed out as red dots. Some labels

with the likely explanations[4] for the outliers were added, and it is interesting to note that most outliers in the returns series do correspond to suspicious peaks in the volatility series, except for Dilma Roussef's impeachment which was not detected as an outlier, likely due to the fact that returns were unstable for a long period of time causing the used outlier detection method to not consider it as one. Calculating the volatility over the now adjusted return series the strongest effect is on the 2008 crisis peak, which is brought down to more reasonable levels, as can be seen in Figure 4.4. The raw series will be mentioned as 'vanilla', while the adjusted series will be mentioned as 'cleaned' for the rest of this work. The model fits over vanilla and cleaned series will be compared against each other, as it will be interesting to see how much of an effect the outliers have on the forecasts.

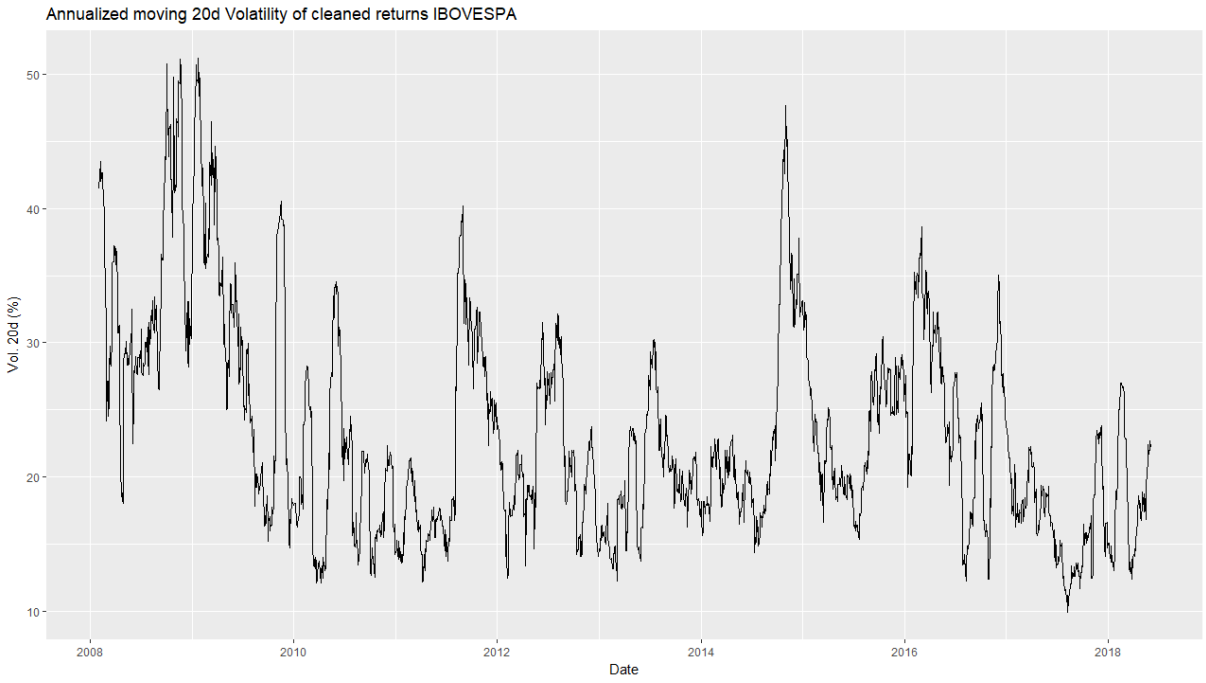


**Figure 4.4:** *IBOVESPA volatility calculated over adjusted returns.*

## 4.3 Other series

As the IBOVESPA index is an aggregate of several stocks from diverse economic sectors, it is not far-fetched to believe that it is possible to find returns series from other securities that are correlated to the index's own returns series. Such correlated series could then be used to improve the neural network accuracy by providing a good variety of explanatory variables. The goal is to choose variables that are correlated to the index so they're relevant, but not too correlated, neither to the index nor to each other, so they're not redundant and needlessly increasing the model's complexity[HI04].

First, the log-returns for the following series were chosen as candidates for being explanatory variables, based on stock market knowledge and their likelihood to be correlated to the index:

- Live cattle 330 arrobas spot price (BGI)
- Dollar exchange rate[5] (DOL)

[4] 2008 Financial Crisis: https://www.thebalance.com/2008-financial-crisis-3305679;
2011 "Black Monday": https://money.cnn.com/2011/08/08/markets/stock-market-loss/index.htm;
Impeachment of Dilma Roussef: https://www.bbc.com/news/world-latin-america-36028247;
2017 JBS investigation: https://www.reuters.com/article/us-brazil-corruption-jbs-idUSKCN18F2LQ

[5] PTAX bid-ask mid as published by B3

- Euro exchange rate[6] (EUR)
- Soybeans 60kg spot price (SOY)
- Arabica Coffee 6 metric tons spot price (ICF)
- Crude oil barrel spot price (WTI)
- 1, 3 and 5 years interest rates (PRE1Y, PRE3Y, PRE5Y)
- Petrobrás preferred stock spot price (PETR4)
- Vale common stock spot price (VALE3)
- Itaú preferred stock spot price (ITUB4)
- Bradesco Bank preferred stock spot price (BBDC4)

In order to evaluate the correlations between the chosen series and the index, the ***rmgarch*** R package[Gha14] was used by fitting a multivariate GARCH model to all series and analyzing the conditional correlations that are a by-product of the fits.

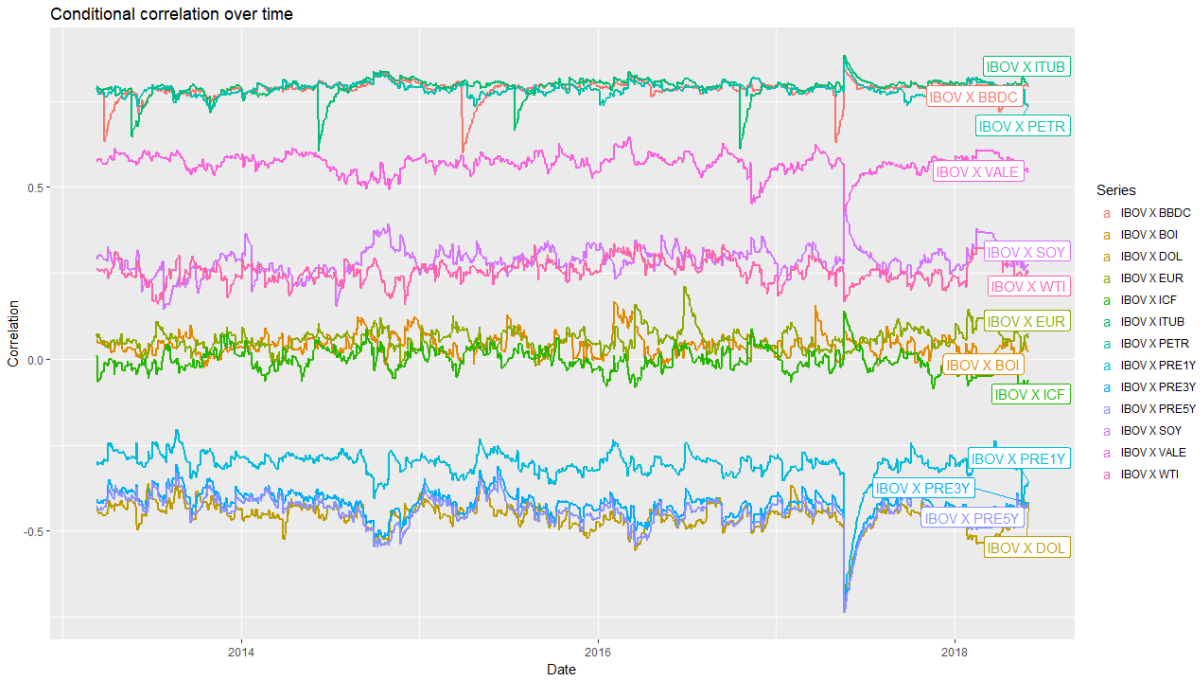


**Figure 4.5:** *IBOVESPA conditional correlation to various series over time.*

In Figure 4.5 it is very noticeable that the stock series have a very high positive correlation to the index, which is expected given they're part of it. As mentioned before, although the index composition changes over time the relevant stocks are consistent, as can be seen by the consistently high correlation over time. Commodities like soy and oil have a somewhat relevant positive correlation to the index, which is not unexpected given they're an important slice of Brazil's exports [MdI].

On the other side of the spectrum we have the dollar and interest rates with relevant negative correlations to the index. Indeed, a high interest rate or weak Brazilian currency are often indicators of poor economic health, and stock prices usually go down when the economic prospects are negative.

Not did we want series with relevant correlation to the index, we also wanted series that weren't too correlated to one another, to avoid redundancy. The same conditional correlation matrices the mGARCH fit produced were used to verify the correlations between the series, as exemplified in Figure 4.6. On cases where one or more series were too correlated to one another, preference was given to the longest series, then to the series with higher correlation to the index.

[6] PTAX bid-ask mid as published by B3

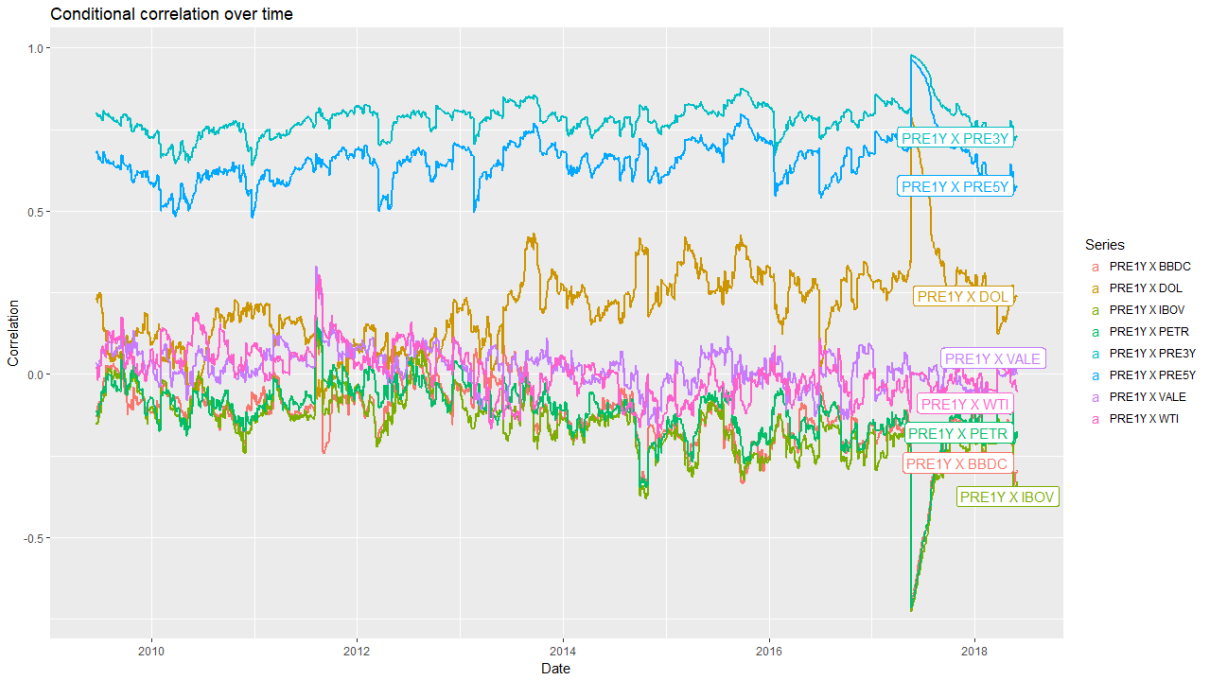


**Figure 4.6:** *1-year interest rate correlation to other series.*

Following the above criteria, the chosen variables for training the neural network are as follows:

- Dollar exchange rate (DOL)
- Crude oil barrel spot price (WTI)
- 5 years interest rates (PRE5Y)
- Petrobrás preferred stock spot price (PETR4)
- Vale common stock spot price (VALE3)
- Bradesco Bank preferred stock spot price (BBDC4)

It is important to note that due to the initial choice of series being quite arbitrary and not that vast, it is quite possible that there are many other suitable series that could be used as input. However, with our choices the commodities and exports sector (WTI, PETR4, VALE3), financial sector (BBDC4), country risk (PRE5Y) and currency fluctuations (DOL) are present, which seems to be a quite diverse coverage of economic sectors. The final choices and their correlations to the index are shown in Figure 4.7.

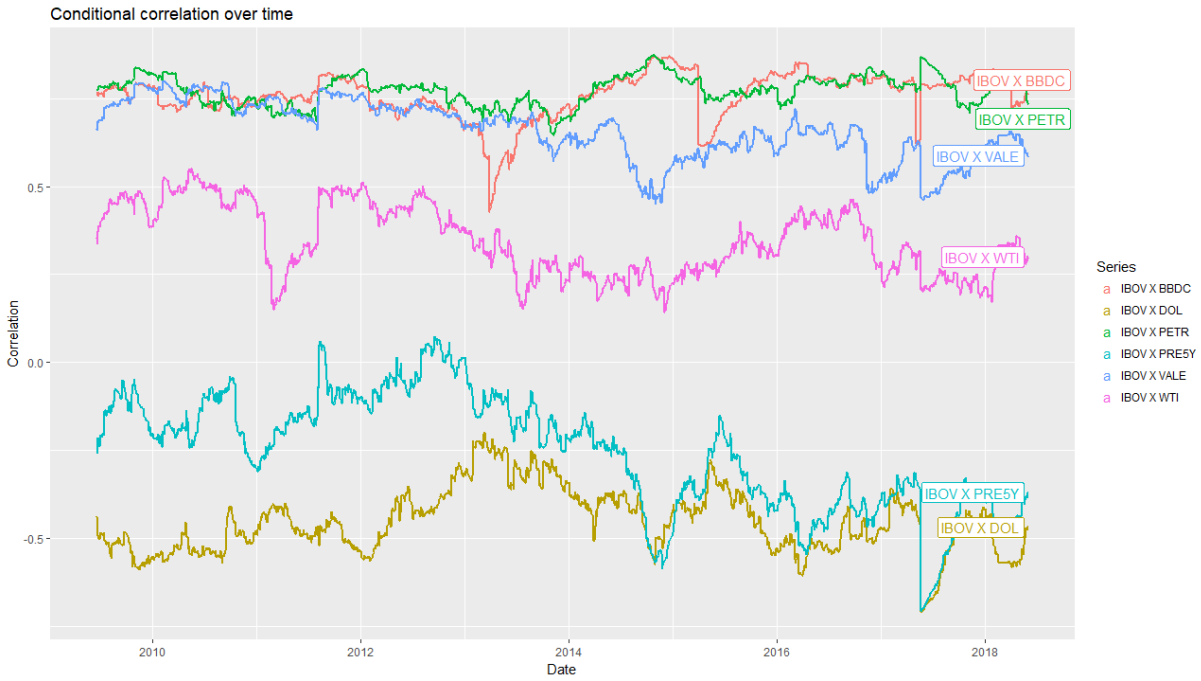


**Figure 4.7:** *IBOVESPA conditional correlation to chosen input series.*

# Chapter 5

# Experiments and Results

## 5.1 Experiments description

In order to evaluate the neural network's prediction performance a few experiments were carried out with standard models as a baseline (ARIMA, GARCH), although not much focus was given to perfecting their fits. As for the RNNs, the two with most promising configurations throughout testing were chosen, as well as the Dual-Stage Attention Based RNN (DA-RNN for short), as presented in Chapter 3. As described in Chapter 4, the original time series are referred to as Vanilla Series, while the series with outlier effects removed by the *tso* function in the *tsoutliers* R package are referred to as Cleaned Series.

### 5.1.1 ARIMA

The ARIMA model was simply fit using the *auto.arima* function in the *forecast* R package, then the forecasts were obtained by using the *forecast* function in the same package. For each date in the last 60 days of the IBOVESPA volatility time series an ARIMA fit was obtained with *auto.arima* over the whole time series up to that date, and then a forecast of 10 or 21 days was obtained with the *forecast* function.

As an example, for the 10-day forecast the time series ends at 2018-05-16, going 60 days back would be 2018-02-20. In the first iteration, the script then calls *auto.arima* on the period from 2008-02-01 to 2018-02-16 of the volatility time series, then uses a forecast of 10 days as the D+10 volatility forecast of 2018-02-20. The same is repeated for 2018-02-21, 2018-02-22 and so on until the end of the sample is reached. At each step the period used with *auto.arima* is increased by one day.

### 5.1.2 GARCH

The GARCH model was fit in a very similar way to the ARIMA model, except using the *garchFit* function from the *fGarch* package, and then the *predict* function on the resulting fit to obtain the forecasts. Unlike *auto.arima* where the model orders are dynamic, *garchFit* was always called to fit a GARCH(1, 1) model. For each date in the last 60 days of the IBOVESPA returns time series a GARCH(1, 1) fit was obtained with *garchFit* over the whole time series up to that date, and then a standard deviation forecast of 10 or 21 days was obtained with the *predict* function. It's important to notice the GARCH model isn't predicting exactly the same measure of volatility being used in the other models, but the results should be somewhat comparable nonetheless.

### 5.1.3 RNN

Both RNNs were trained by segregating the dataset into a training, validation and test sets. The validation dataset was comprised of the last 502 days of the sample, excluding the last 60 days for testing purposes, or about 20% of the dataset, while the training dataset used the sample that

remained after excluding the validation and the test periods, as shown in Figure 5.1. Other sizes for the validation and training sets were also tested, but ultimately using 20% of the dataset for validation seemed to yield the best results.

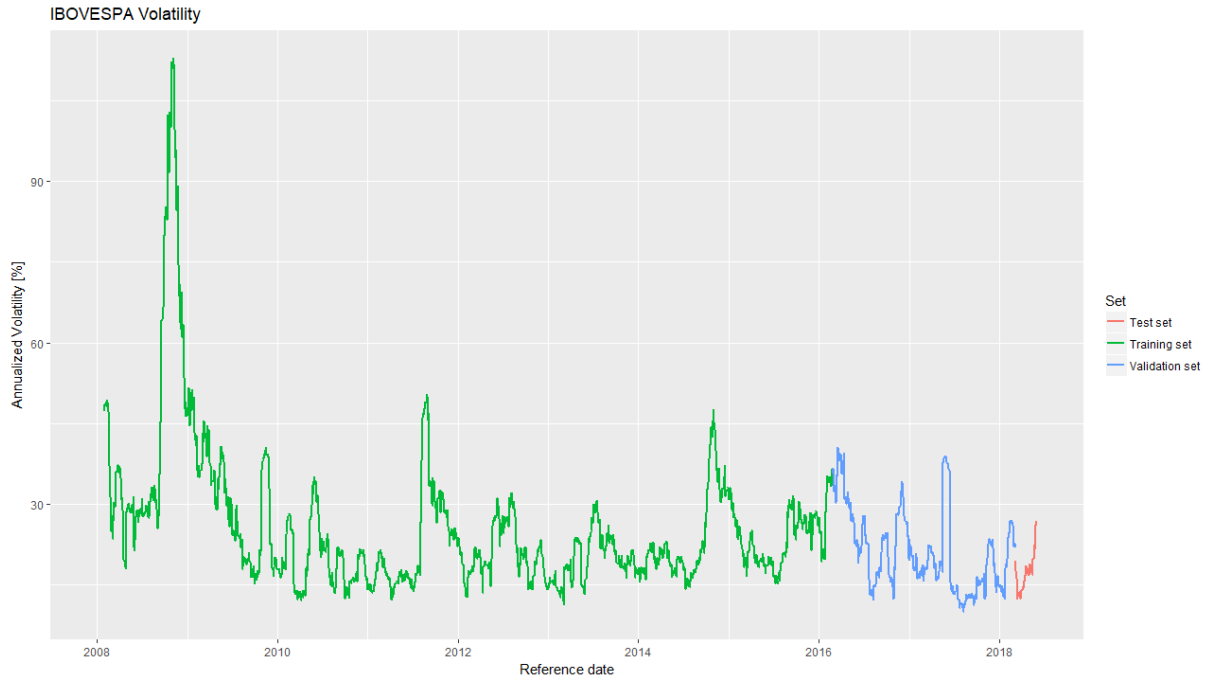


**Figure 5.1:** *Visualization of how the series was split for training. The graph shows only the IBOVESPA volatility series but the other inputs were split in the same way.*

After training the best performing network state was then used to generate the predictions for the last 60 days of sample.

### 5.1.4 DA-RNN

Unlike the RNNs the DA-RNN used the whole dataset save for the last 60 days for training, albeit with a different training procedure as explained in Section 3.2. The state with the lowest training error was then used to generate predictions for the last 60 days of sample.

## 5.2 Results comparison

Each model was used to predict the D+10 or D+21 days volatility in the period from 2018-02-20 to 2018-05-16 or 2018-01-30 to 2018-04-30 respectively, and the results were then compared using the error measures, similar to [QSC+17], defined as:

- Mean squares error: $mean\left((predictions - realValues)^2\right)$, where the predictions and real values are annualized volatility values;
- Mean percent error: $100 * mean\left(|\frac{(predictions - realValues)}{realValues}|\right)$

For each model a graph was generated containing the volatility series starting from 2017-06-01 until the last date available. It is important to note that the period used for testing had unusual volatility movements due to political and economical influence on the index, in particular the last days in the sample were heavily affected by the truckers' strike that took place during May 2018[DA18], which in turn means most models are likely to be unable to form good predictions for that period. On the opposite side of the test sample there's also a sharp drop in volatility levels that can be explained by the overall good economic prospects Brazil had around the beginning of

2018 and relative domestic and international stability in politics and economics, and that shows to be difficult to model.

### 5.2.1 10 days prediction

| Model | Mean squares error | Mean percent error |
|---|---|---|
| ARIMA Vanilla Series | 24.13 | 23.11% |
| ARIMA Cleaned Series | 293.75 | 80.30% |
| GARCH(1, 1) Vanilla Series | 27.07 | 29.00% |
| GARCH(1, 1) Cleaned Series | 22.53 | 24.85% |
| RNN-1 Vanilla Series | 19.92 | 22.89% |
| RNN-1 Cleaned Series | 19.65 | 20.25% |
| RNN-2 Vanilla Series | 11.96 | 16.06% |
| RNN-2 Cleaned Series | 16.80 | 17.42% |
| DA-RNN Vanilla Series | 21.52 | 21.97% |
| DA-RNN Cleaned Series | 22.67 | 21.20% |

**Table 5.1:** *Forecast accuracy by model - 10 days prediction*

Table 5.2 shows the best performing model was RNN-2 trained over the vanilla series, closely followed by the same network but trained over the cleaned series, and significantly more accurate than the other models. The graphs showing the test results for each model's predictions follow below, in figures 5.2, 5.3, 5.4, 5.6 and 5.8.

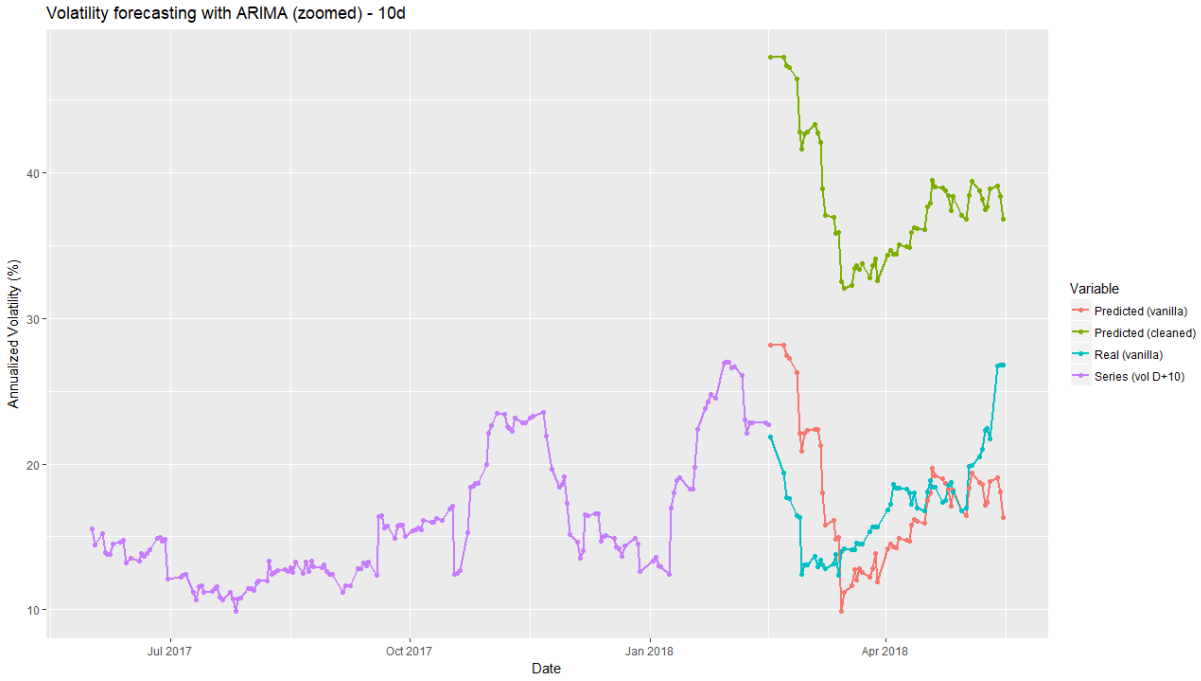


**Figure 5.2:** *ARIMA fit results for 10 days prediction*

The cleaned volatility series for IBOVESPA utilized in fitting the ARIMA model had a much higher volatility level than the vanilla series, which accounts for the difference in level between the predictions. Visually, it seems as if overall the ARIMA fit manages to follow reasonably well the direction of movements but fails to predict the levels accurately at most points.

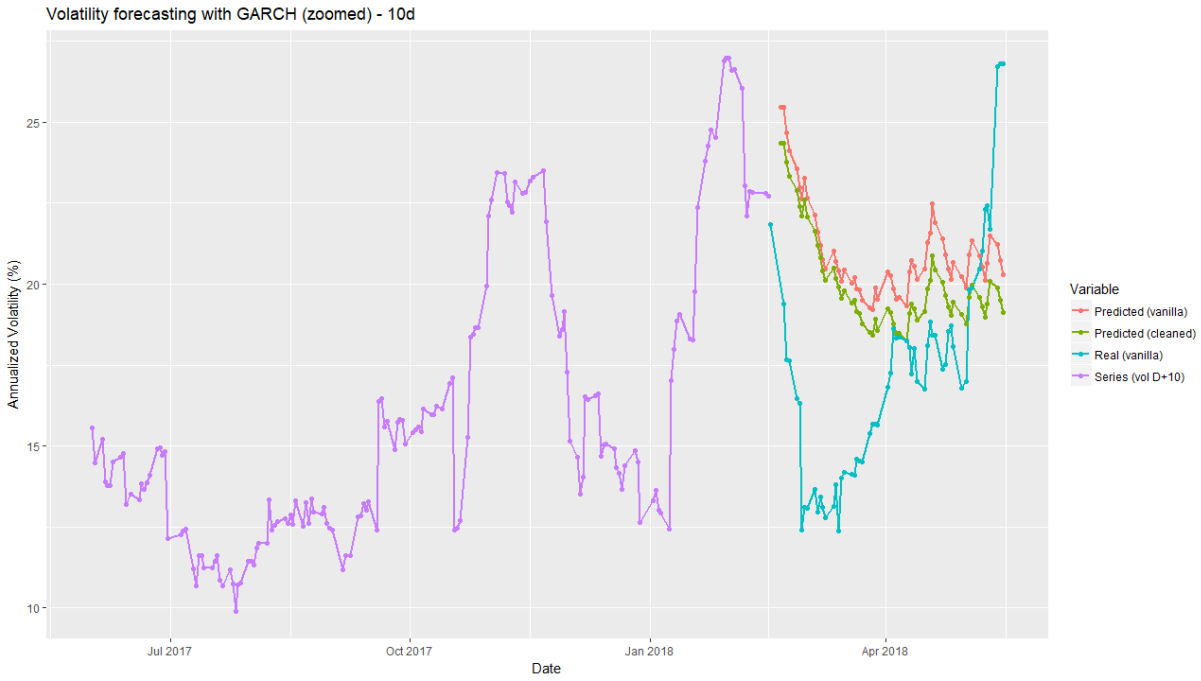


**Figure 5.3:** *GARCH fit results for 10 days prediction*

Much like the ARIMA predictions, the GARCH ones seem to overall follow the volatility movements directions while failing to predict the levels. The steep rise at the end of the sample is also not predicted.

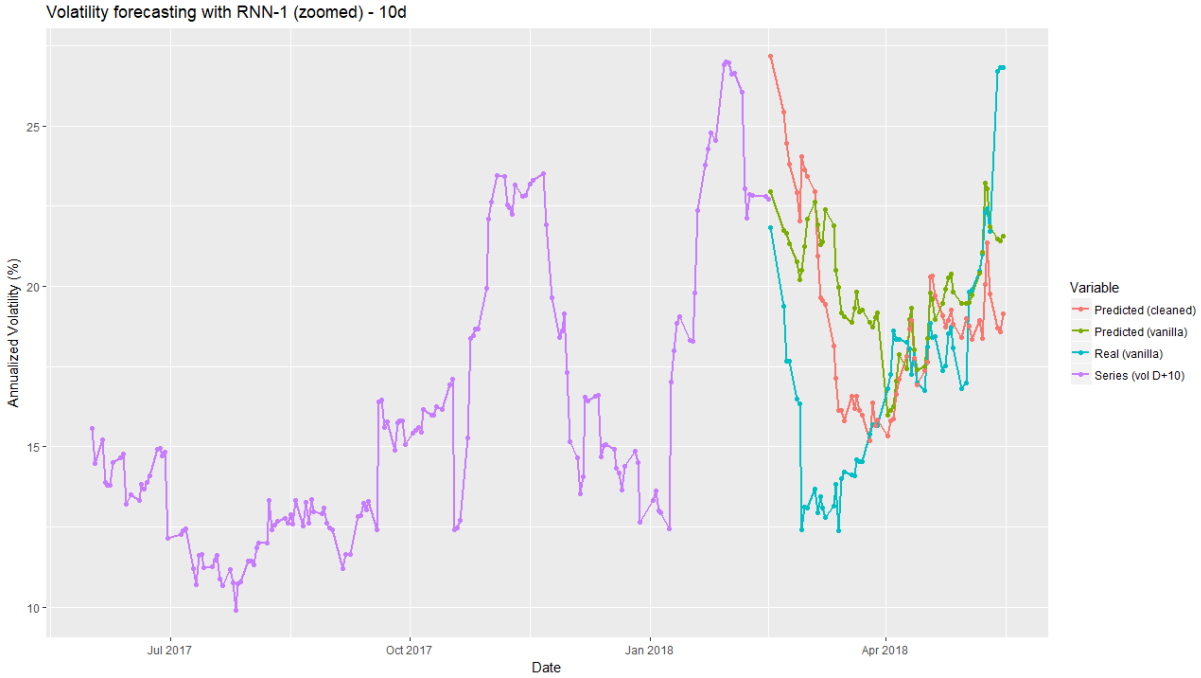


**Figure 5.4:** *RNN-1 model results for 10 days prediction*

For our first RNN model we can see benefits to using the cleaned series for training as it seems to better follow both the movements and levels of the volatility series. The predictions utilizing the cleaned series seem to follow the initial volatility levels decline very closely, and then follow most of the rise except for the last few days of the sample. It's visible in Figure 5.5 that during the short period of non-extreme variation in the sample the network's predictions seem to be quite accurate.

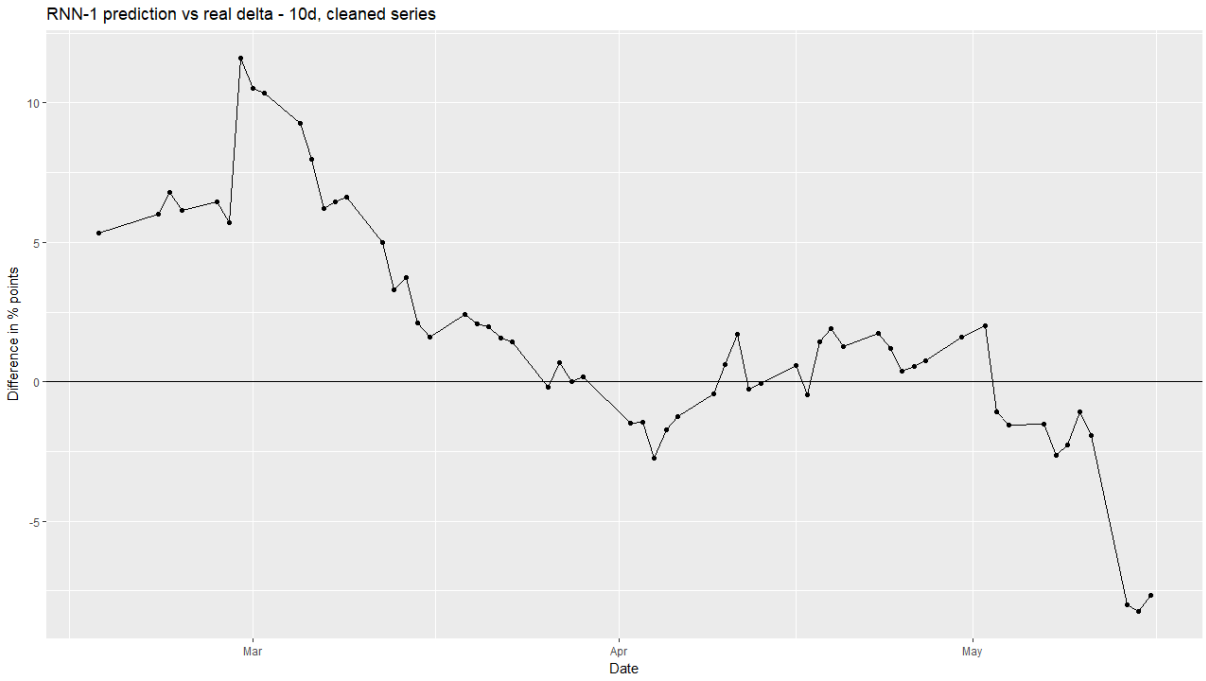


**Figure 5.5:** *Distance between RNN-1 model predictions and real values*

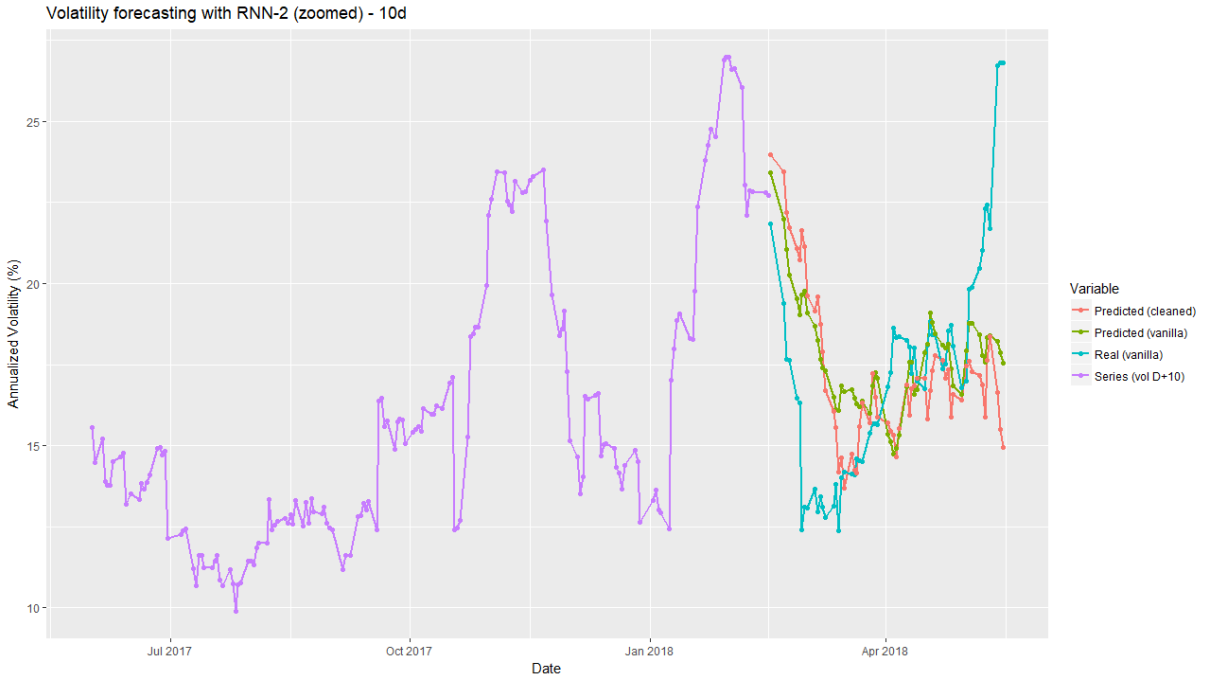


**Figure 5.6:** *RNN-2 model results for 10 days prediction*

The second RNN model results behavior is close to the first RNN ones, but with better approximation to the observed volatility levels in the period. Again, the model fails to predict the steep rise in volatility levels around the end of the sample. The results for the model trained over the vanilla series were the best observed between all models, and the proximity to real values can be more easily seen in Figure 5.7.

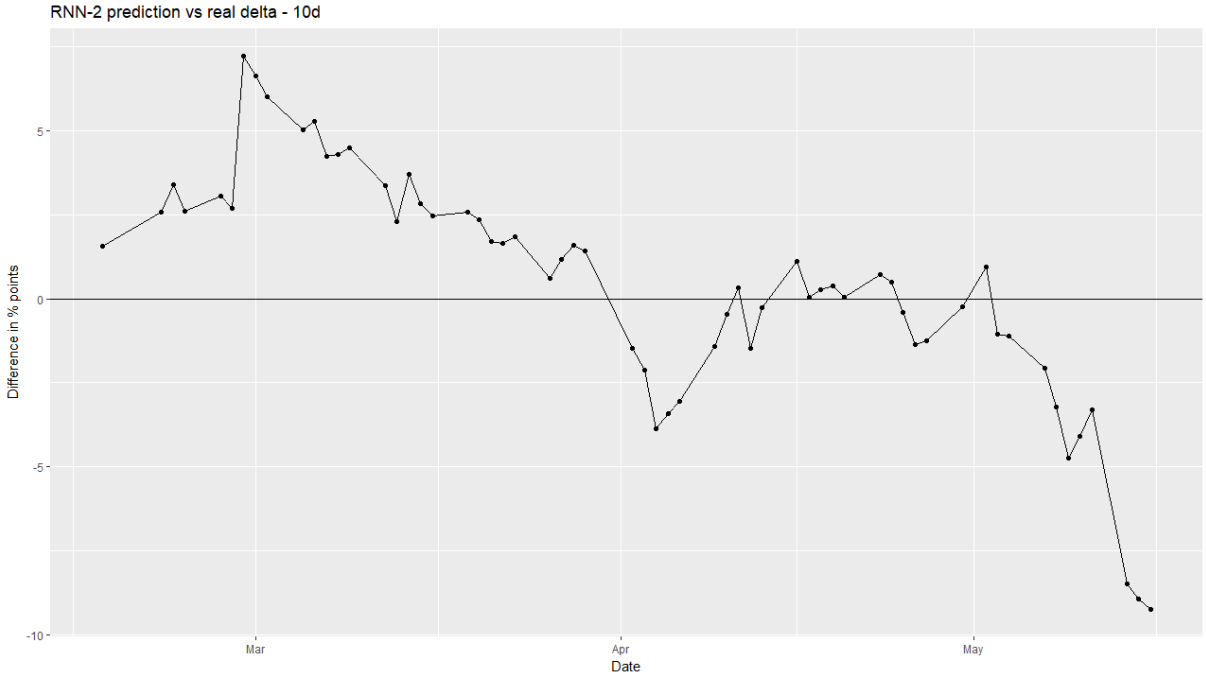


**Figure 5.7:** *Distance between RNN-2 model predictions and real values*

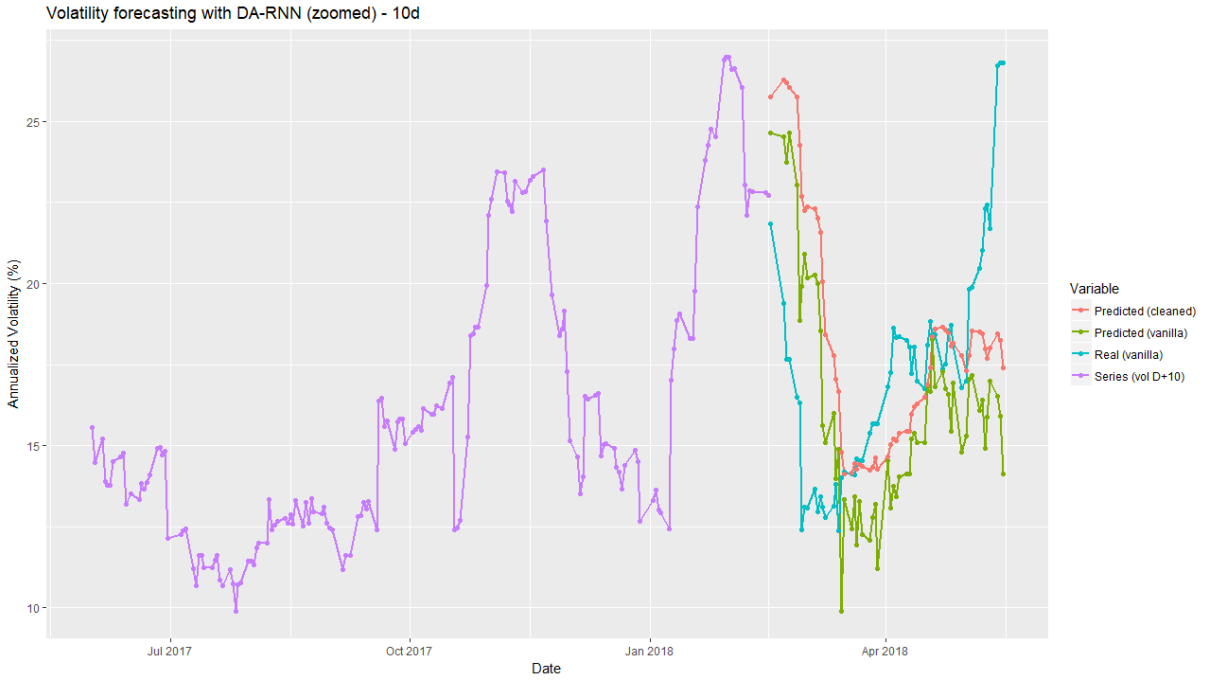


**Figure 5.8:** *DA-RNN model results for 10 days prediction*

The DA-RNN predictions appear to be somewhat more volatile when compared to the other neural networks and not particularly better in capturing both the levels and movements of the real volatility series.

### 5.2.2 21 days prediction

| Model | Mean squares error | Mean percent error |
|---|---|---|
| ARIMA Vanilla Series | 55.86 | 38.72% |
| ARIMA Cleaned Series | 511.45 | 135.85% |
| GARCH(1, 1) Vanilla Series | 50.67 | 39.93% |
| GARCH(1, 1) Cleaned Series | 38.15 | 32.74% |
| RNN-1 Vanilla Series | 25.34 | 24.50% |
| RNN-1 Cleaned Series | 43.77 | 33.35% |
| RNN-2 Vanilla Series | 41.22 | 33.79% |
| RNN-2 Cleaned Series | 44.61 | 35.56% |
| DA-RNN Vanilla Series | 50.11 | 35.71% |
| DA-RNN Cleaned Series | 63.07 | 38.69% |

**Table 5.2:** *Forecast accuracy by model - 21 days prediction*

Table 5.2 shows the best performing model was RNN-1 trained over the vanilla series, with a significantly more accurate result when compared to all other tested models. The graphs showing the test results for each model's predictions follow below, in figures 5.9, 5.10, 5.11, 5.13 and 5.14.

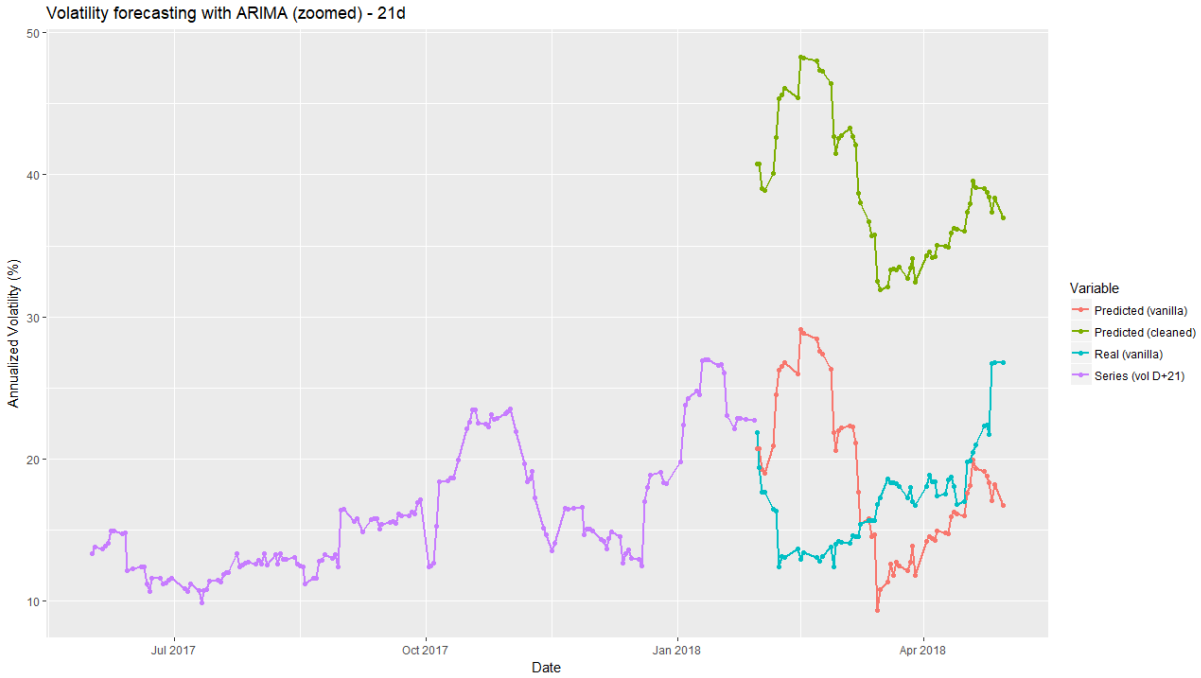


**Figure 5.9:** *ARIMA fit results for 21 days prediction*

When compared to the 10 days prediction it is quite clear even visually that the 21 days prediction is much further away from the real observed values. An interesting point to note that will be recurrent for every model is that the steep decline around the beginning of the sample is wrongly predicted as a sharp increase in volatility level. After that the model appears to somewhat follow the series dynamics, only to fail again at the increase in levels at the end of the sample.

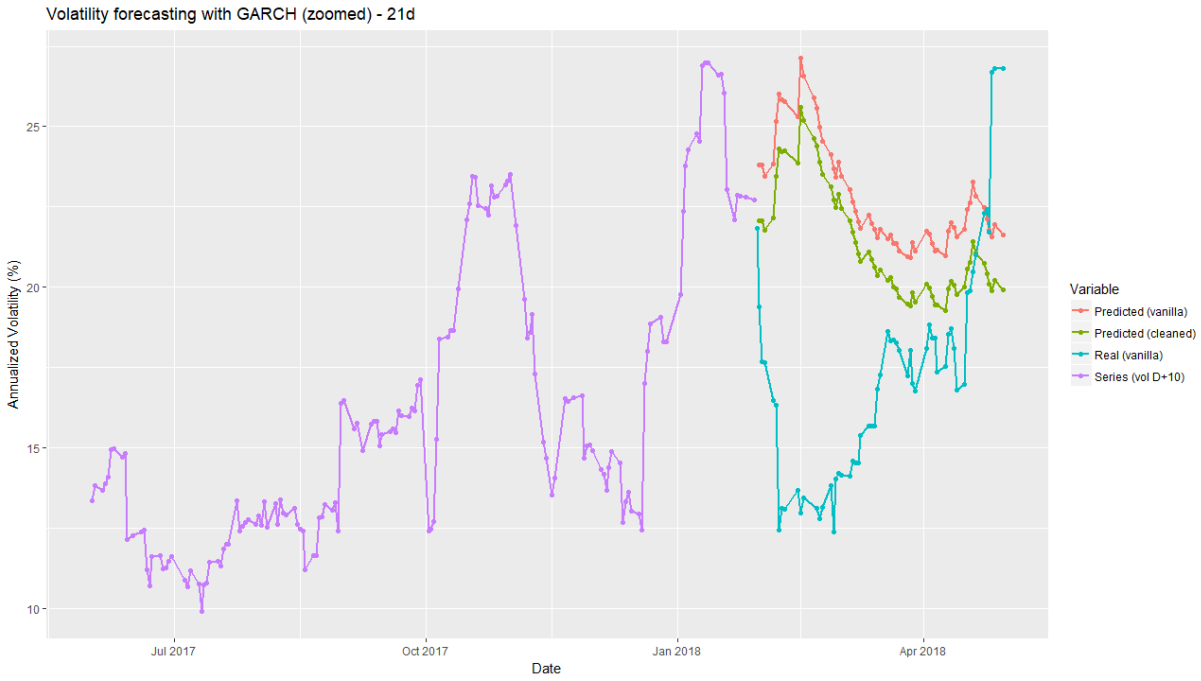


**Figure 5.10:** *GARCH fit results for 21 days prediction*

As like the 10 days prediction the GARCH fit seems to overestimate the volatility levels, and as mentioned for the ARIMA model the start of the sample is predicted as a sharp increase instead of a decline in volatility levels.

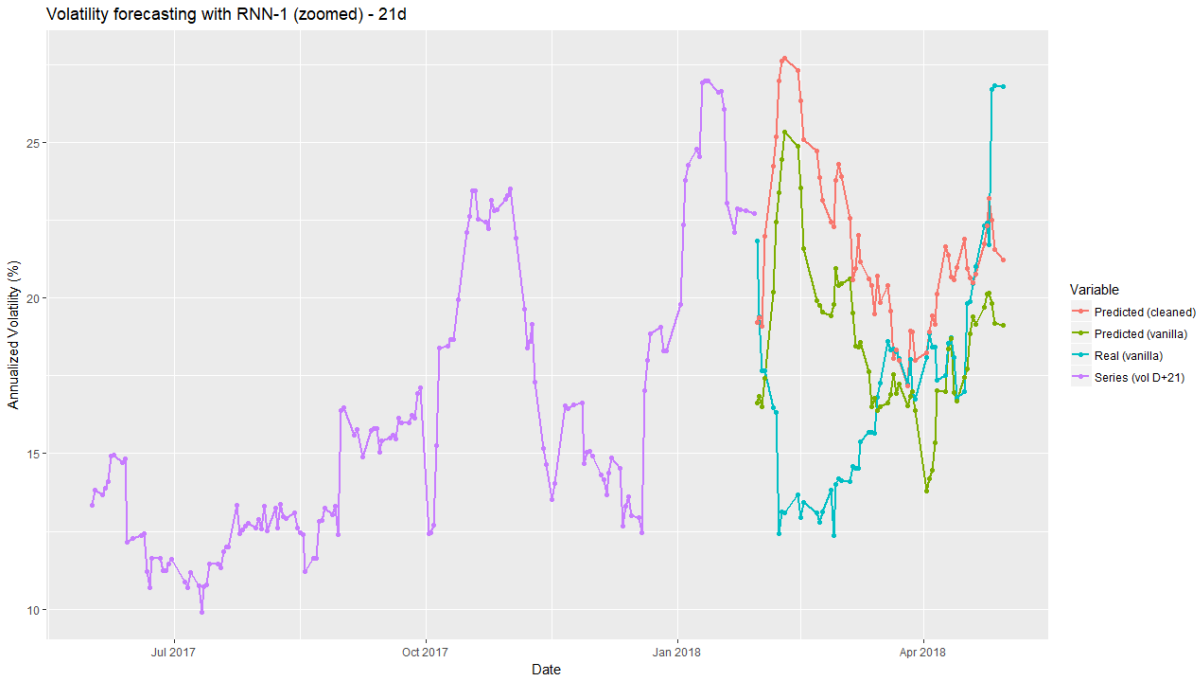


**Figure 5.11:** *RNN-1 model results for 21 days prediction*

The first RNN model also predicts the decline in levels as an increase. Afterwards the predictions overall follow the real series dynamics, but it's quite clear that the 21 days predictions are less accurate than the 10 days ones. Looking at the distance between predictions and real values at each date in Figure 5.12, for a short period the network gets closer to the real values, only to diverge again when the real series has strong volatility variations. The RNN-1 model trained over the vanilla series appears to be the best fit for 21 days prediction, indicating that for different forecast horizons

different models will be more or less effective.

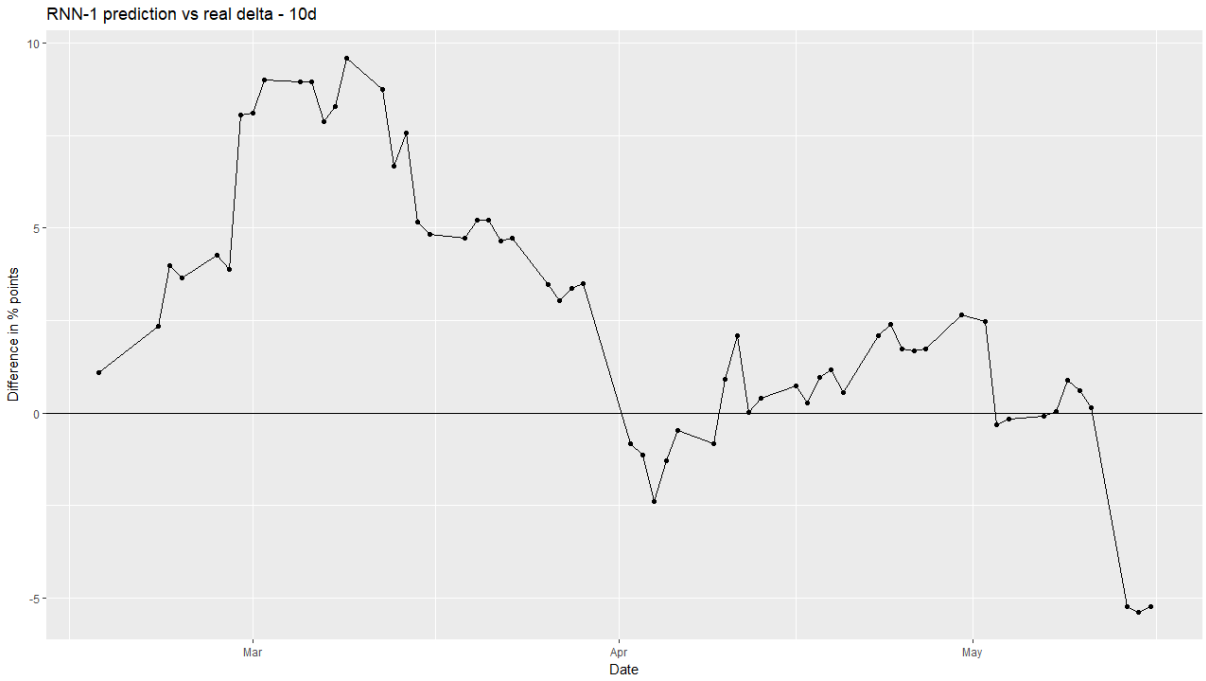


**Figure 5.12:** *Distance between RNN-1 model predictions and real values*

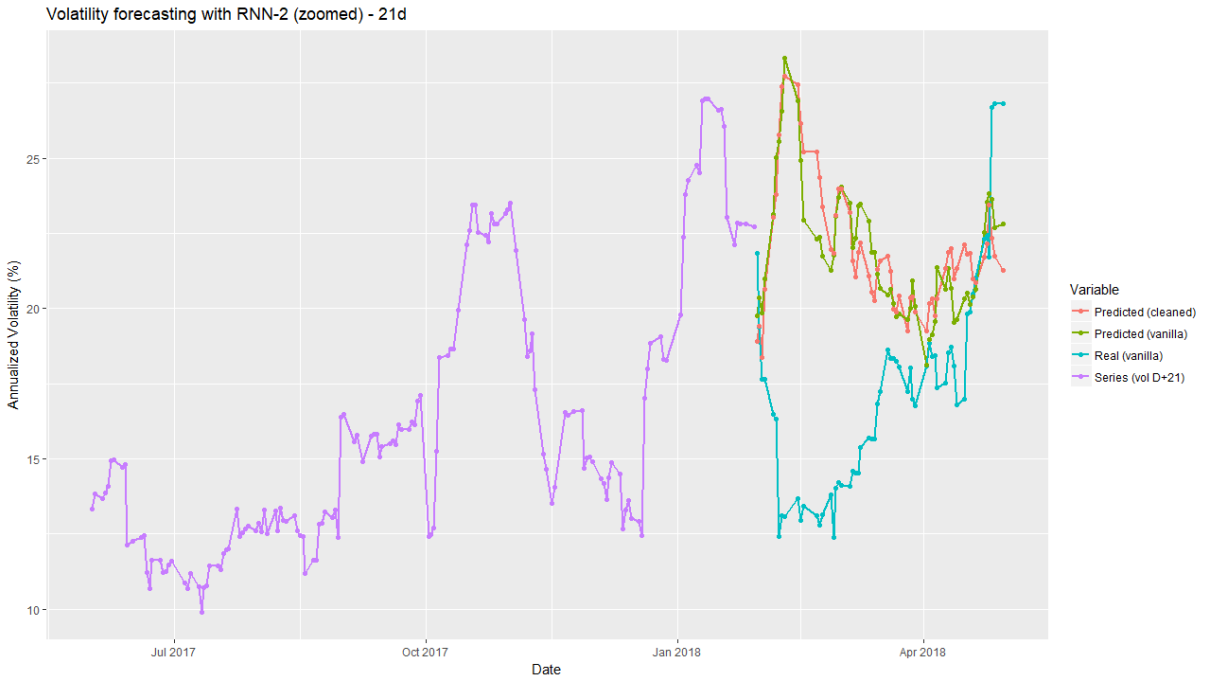


**Figure 5.13:** *RNN-2 model results for 21 days prediction*

The second RNN model this time fails to approach the real volatility levels and consistently overestimate them. That aside, the overall behavior is very similar to the RNN-1 results.

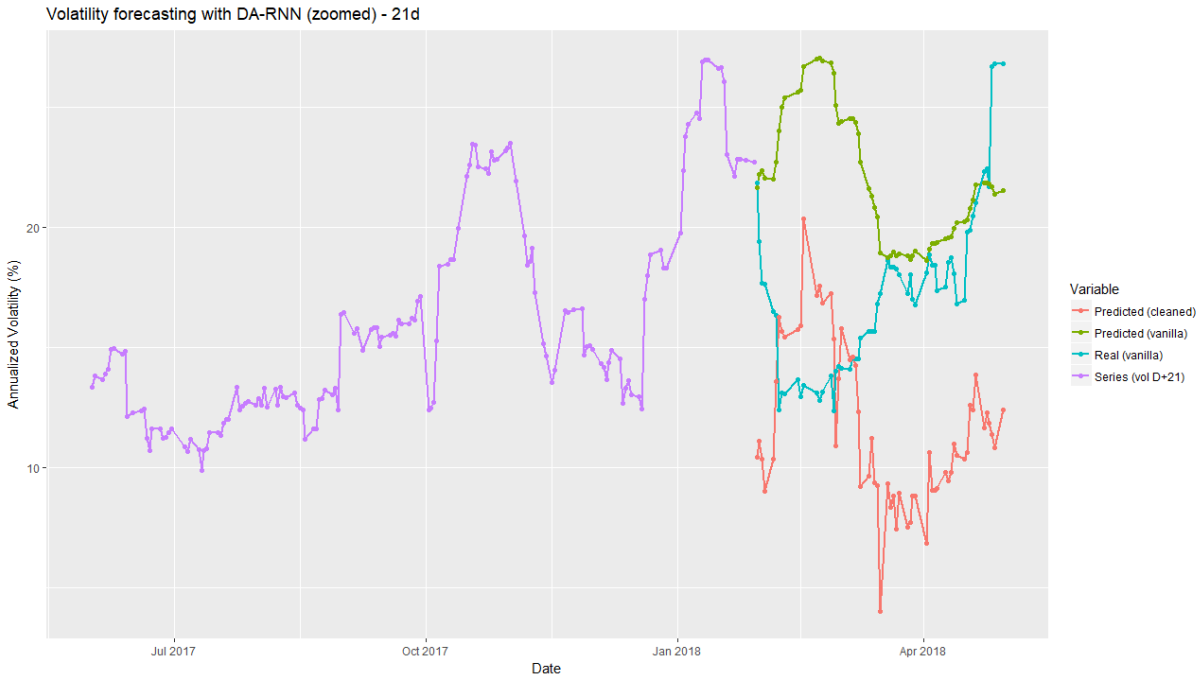


**Figure 5.14:** *DA-RNN model results for 21 days prediction*

The DA-RNN predictions are very poor, which is likely a direct consequence of needing to forecast 21 data points per date because of the necessity of having the past target series values as an input, as was discussed in Section 3.2.

# Chapter 6

# Conclusions

The proposed recurrent neural network shows to be significantly more accurate for forecasting the IBOVESPA index volatility when compared to more traditional time series models, both for predicting volatility levels and increases or decreases. It is also clear, however, that the neural networks do not behave ideally for periods of increased volatility variance, specially when that variance comes from external unforeseeable factors like politics, and also produce poorer results for longer forecast horizons, although the above deficiencies are seen throughout most models.

Regarding the effects of adjusting input series for outliers, while the effects of the adjustment on the forecasts are quite noticeable, it is not clear that adjusting the series will always be beneficial for the predictions accuracy. It is quite possible that a more careful outlier evaluation that doesn't rely on a fully automated process would improve results, which could be an avenue for future model improvement.

Other very important factors that can be further improved on to better the forecast accuracy are the exogenous factors choice, the network architecture and the training method and parameters. Six exogenous factors were chosen that covered multiple economic sectors, but it's very likely more viable series exist that would improve the neural network, for instance obtaining a measure of political uncertainty could help with forecasting the variance of volatility by introducing political dynamics into the model. As for the network's architecture and training parameters it is also possible there are better combinations than what was found throughout the development of this work, the impact those settings have on model quality is quite significant as can be seen by comparing RNN-1 and RNN-2 results.

In conclusion, we have managed to develop a RNN based model that produces adequate forecasts for short term index volatility save for sudden and intense level shifts with causes that go beyond the factors used for training the network. The model also has room for future improvements, with the addition or exclusion of exogenous factors and architectural or parametric tweaking.

# Appendix A

# InfluxDB

This appendix gives a brief introduction on the time series database, **InfluxDB**, which was used to store the dataset described in Chapter 4. As the dataset was publicized[1] as an InfluxDB dump, it will be necessary to set up a local instance in order to access the stock market data. Although there are multiple ways of retrieving data from the database we'll focus on using the *influxdbr*[Leu18] R package for quick and easy access through R scripts.

## A.1 Overview

InfluxDB is a free open source[2] time series database written in Go built for usage with large amounts of time-stamped data, that offers a query language similar to SQL for access, high performance for real-time data insertion and querying and automatic data compacting. Its features go above and beyond what are needed for storing daily stock market data, but it was chosen because of its simple set-up process, lack of external dependencies and easy data retrieval through HTTP APIs.

An InfluxDB database is composed of *measurements*, that work similarly to SQL tables. Because it's a time series database, every measure has a *time* column as its key storing timestamps, usually date and time, associated with some data. *Field values* are the data associated with timestamps, they can be strings, floats, integers or booleans and are named by a *field key*. A timestamp entry can be associated with not only fields, but also with *tags*, which have their own values and keys. Tags are indexed, meaning queries that use tag values as filters perform much better than queries that use field values as filters, which aren't indexed. A measurement is exemplified in Table A.1.

| time | value | returns | vol20d | variable |
|---|---|---|---|---|
| 2008-01-31 | 12.09 | 0.59% | 0.5% | PRE1Y |
| 2008-02-01 | 12.13 | 0.29% | 0.48% | PRE1Y |
| 2008-02-06 | 12.18 | 0.40% | 0.45% | PRE1Y |
| 2008-01-31 | 12.92 | 0.29% | 0.86% | PRE3Y |
| 2008-02-01 | 12.86 | -0.46% | 0.85% | PRE3Y |
| 2008-02-06 | 12.89 | 0.23% | 0.85% | PRE3Y |
| 2008-01-31 | 12.92 | 0.19% | 0.81% | PRE5Y |
| 2008-02-01 | 12.84 | -0.62% | 0.8% | PRE5Y |
| 2008-02-06 | 12.88 | 0.29% | 0.8% | PRE5Y |

**Table A.1:** *A few entries from the interest_rate measurement. value, returns and vol20d are fields and variable is a tag*

Data can be written and retrieved using a HTTP API with queries smiliar to SQL ones. We'll

[1] https://linux.ime.usp.br/~ogawa/mac0499/files/dataset.zip
[2] https://github.com/influxdata/influxdb

go over querying utilizing the *influxdbr* R package, but for example here's how to create a database in a default InfluxDB installation using a HTTP query:

```
curl -i -XPOST http://localhost:8086 --data-urlencode "q=CREATE DATABASE dbName"
```

And how to retrieve a JSON with data from the above mentioned interest_rate measurement:

```
curl -G 'http://localhost:8086/query' --data-urlencode "db=DadosHistBMFBOVESPA"
    --data-urlencode "q=SELECT \"value\" from \"interest_rate\" WHERE \"variable\
    "='PRE1Y'"
```

## A.2 Setup

In order to load an InfluxDB database with the data provided in the BMFBovespa dataset, the latest release should first be downloaded from https://portal.influxdata.com/downloads and installed following the documentation instructions. If the purpose of the installation is only to use the provided dataset all settings can be left at the default values.

Once installed, the influxd executable should be accessible wherever InfluxDB was installed, or on system path. The installation also has default meta and data folders that can be checked or set using the influxdb.conf file, or by running the influxd executable once for initial setup. After locating those folders, the dataset can be loaded into InfluxDB by running the commands:

```
influxd restore -metadir <meta folder path> <dump folder path>
influxd restore -database DadosHistBMFBOVESPA -datadir <data folder path> <dump
    folder path>
```

Where the dump folder path is the folder called "data_dump" available in the dataset zip. Once InfluxDB is done restoring the backup, simply run the influxd executable again to start the service and make the database accessible through the HTTP API.

## A.3 Accessing the data

We'll focus on utilizing the *influxdbr* R package for accessing the data, which means an R installation is required, as well as installing the package through the install.packages("influxdbr") function. Once the database service is running, a connection to it can be set up with the function influx_connection, which defaults to the same default values a new InfluxDB installation uses. If the InfluxDB configuration differs from default, the settings will have to be specified when setting up the connection. Once a connection is made, data can be written using the influx_write function or retrieved with the influx_select function. An usage example follows below in Listing A.1, functions definitions and further information are available in the *influxdb* documentation.

```r
library(influxdbr)

con <- influx_connection()

exampleData <- data.frame(refDate = as.Date("2008-01-31"),
                          value    = 12.09,
                          returns  = 0.0059,
                          vol20d   = 0.005,
                          variable = "PRE1Y")

# Writing data to the interest_rate measurement

influx_write(x           = exampleData,
             con         = con,
             db          = "DadosHistBMFBOVESPA",
             time_col    = c("refDate"),
             measurement = "interest_rate",
             consistency = "all",
             tag_cols    = c("variable"))

# Retrieving PRE1Y data from the interest_rate measurement as a data.frame,
# by default the data is returned as a xts object

irData <- influx_select(con, "DadosHistBMFBOVESPA",
                          field_keys   = "*",
                          measurement  = "interest_rate",
                          return_xts   = FALSE,
                          simplifyList = TRUE,
                          where        = "variable = 'PRE1Y'")
```

**Listing A.1:** *influxdbr usage example*

# Bibliography


[AB98] Torben G. Andersen e Tim Bollerslev. Answering the skeptics: Yes, standard volatility models do provide accurate forecasts. *International Economic Review*, 39(4):885–905, 1998. 1

[B+95] Christopher M Bishop et al. *Neural networks for pattern recognition*. Oxford university press, 1995. 4

[B3] B3. *B3 Pricing Manual, Futures Contracts*. 16

[BF96] Timothy J. Brailsford e Robert W. Faff. An evaluation of volatility forecasting techniques. *Journal of Banking & Finance*, 20(3):419 – 438, 1996. 1

[BM96] Tim Bollerslev e Hans Ole Mikkelsen. Modeling and pricing long memory in stock market volatility. *Journal of econometrics*, 73(1):151–184, 1996. 1

[BMF] BMFBOVESPA. Metodologia do Índice ibovespa. http://www.bmfbovespa.com.br/lumis/portal/file/fileDownload.jsp?fileId=8A828D29514A326701516E695D7F65C0. Last accessed: 2018-04-15. 1, 17

[BS73] Fischer Black e Myron Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973. 1

[BSF94] Y. Bengio, P. Simard e P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, March 1994. 4

[CLD03] An-Sing Chen, Mark T Leung e Hazem Daouk. Application of neural networks to an emerging financial market: forecasting and trading the taiwan stock index. *Computers & Operations Research*, 30(6):901–923, 2003. 18

[DA18] Shasta Darlington e Manuela Andreoni. Trucker's strike paralyzes brazil as president courts investprs. *The New York Times*, May 2018. 24

[DK] R. Glen Donaldson e Mark Kamstra. Forecast combining with neural networks. *Journal of Forecasting*, 15(1):49–61. 1

[Eas] Ed Easterling. Volatility in perspective. https://www.crestmontresearch.com/docs/Stock-Volatility-Perspective.pdf. Last accessed: 2018-04-15. 1

[FVD] Philip Hans Franses e Dick Van Dijk. Forecasting stock market volatility using (non-linear) garch models. *Journal of Forecasting*, 15(3):229–235. 1

[GBC16] Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org. v, 4, 5, 6

[Gha14] Alexios Ghalanos. *rmgarch: Multivariate GARCH models.*, 2014. R package version 1.2-8. 20

[GKD11] Erkam Guresen, Gulgun Kayakutlu e Tugrul U. Daim. Using artificial neural network models in stock market index prediction. *Expert Systems with Applications*, 38(8):10389 – 10397, 2011. 1

[HI04] Shaikh A. Hamid e Zahid Iqbal. Using neural networks for forecasting volatility of s&p 500 index futures prices. *Journal of Business Research*, 57(10):1116 – 1125, 2004. Selected Papers from the third Retail Seminar of the SMA. 1, 19

[HL] Peter R. Hansen e Asger Lunde. A forecast comparison of volatility models: does anything beat a garch(1,1)? *Journal of Applied Econometrics*, 20(7):873–889. 1

[Jor00] Philippe Jorion. Value at risk. 2000. 1

[KB14] Diederik P. Kingma e Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. 9

[Leu18] Dominik Leutnant. *influxdbr: An R interface to the InfluxDB time series database.*, 2018. R package version 0.14.2. 34

[McD] Conor McDonald. Machine learning fundamentals (ii): Neural networks. https://towardsdatascience.com/machine-learning-fundamentals-ii-neural-networks-f1e7b2cb3eef. Last accessed: 2018-11-17. v, 4

[MdI] Comércio Exterior e Serviços Ministério da Indústria. Brazil's external balance. http://www.mdic.gov.br/comercio-exterior/estatisticas-de-comercio-exterior/comex-vis/frame-brasil. Last accessed: 03/06/2018. 20

[PyT] PyTorch. *PyTorch documentation*. 9

[QSC+17] Yao Qin, Dongjin Song, Haifeng Cheng, Wei Cheng, Guofei Jiang e Garrison W. Cottrell. A dual-stage attention-based recurrent neural network for time series prediction. *CoRR*, abs/1704.02971, 2017. 7, 24

[RHW86] D. E. Rumelhart, G. E. Hinton e R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, páginas 318–362. MIT Press, Cambridge, MA, USA, 1986. 4

[RKK18] Sashank J. Reddi, Satyen Kale e Sanjiv Kumar. On the convergence of adam and beyond. Em *International Conference on Learning Representations*, 2018. 10

[RM99] Russell Reed e Robert J MarksII. *Neural smithing: supervised learning in feedforward artificial neural networks*. Mit Press, 1999. 4

[Roh07] Tae Hyup Roh. Forecasting the volatility of stock price index. *Expert Systems with Applications*, 33(4):916 – 922, 2007. 1

[Sch15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015. 4

[Zuo17] Chandler Zuo. A pytorch example to use rnn for financial prediction, Nov 2017. 11