Recurrent Neural Nets for Time Series forecasting

Piotr Zawiła-Niedźwiecki

June 2020

1 Introduction

As part of this assignment we will discuss different Neural Networks architectures, that are viable in Time Series Forecasting. In lectures we've heard about regular Neural Nets, in the following sections we will talk about specific types of Convolutional Neural Networks, and different kinds of Recurrent Neural Networks. Those kinds of networks try to tackle processing data provided in longer sequences, which makes them a perfect fit for Time Series forecasting. What is most important the Nets mentioned below can work on sequences of arbitrary lengths, rather than on fixed-sized inputs, allowing for much deeper understanding of patterns.

2 Overview of architectures

In this section I will shortly explain State of the Art Neural Networks, when it comes to Time Series forecasting. I will not dive deeper into explanations of their inner workings. First I will talk about Convolutional Neural Networks, as they weren't part of the time series lectures I will skip mathematical nuances behind them, and not explain them in-depth later. Recurrent Neural Nets and their variations will be explained more in-depth below, but that section requires initial knowledge of ideas such as activation functions, backpropagation or vanishing gradient problem. For those of you who need some preliminary reading I suggest this link:

`https://towardsdatascience.com/simple-introduction-to-neural-networks-ac1d7c3d7a2c`

As the ideas there are presented in an accessible way for inexperienced reader.

2.1 Recurrent Neural Networks

Till this point we've only dealt with feedforward neural nets. What does this mean? Those are networks, that process the data throughout all the layers moving only forward from the input to the output. Recurrent neural nets (RNN), are very similar to them with the exception that they can also

"learn from themselves" as they also pass their own predictions backwards. This allows them to learn longer patterns in the data, and give multiple sensible predictions that are correlated with each other. Time series are one of the prime examples of the RNN aplications. Other worth mentioning are used within autonomous driving systems (`https://blogs.nvidia.com/blog/2019/05/22/drive-labs-predicting-future-motion/`), that learn to intercept other cars movements, or audio generation (example of google doodle: `https://www.google.com/doodles/celebrating-johann-sebastian-bach`).

Vanilla RNN's aren't that effective at those tasks, and simple recurrent neurons are used more as a part of so-called "Memory Cells". Those cells are combination of different neuron layers, that are operated by gateways allowing them to remember or forget certain information from the data they were fed. Most notable "Memory Cells" are LSTM's and GRU. Long Short-Term Memory (LSTM) cell was proposed back in 1997, and since then was systemically improved by research community (including one researcher of Polish origin, and former MIMUW student, Wojciech Zaremba, co-founder of OpenAI). Gated Recurrent Unit (GRU) cell was proposed back in 2014, and is a simplification of LSTM cell, that performs as good as it's older brother.

LSTM as a more advanced cell, has multiple smaller neuron layers within it. We can simply see it as a neuron that can adapt during training way more than regular RNN's. It can save it's own memory and if it finds something unimportant at some point during training, it will remove it from memory, and fill in the missing spot with new important information. It has a long-term memory of patterns that is preserved throughout whole training, and short-term memory that is used when it sees new data. Using the short-term memory it decides what he should forget from the long-term memory, and it applies this procedure every time it is fed new data. So for example it may find that during analysis of weather data it doesn't need to remember what was the weather 10 minutes ago, but only 1 hour intervals would give it useful information.

GRU as a cell is simpler than LSTM, and has only one "memory". We can see it is as a LSTM without short-term memory, and it uses it's only "memory" to memorize important information. This would seem as an inferior solution compared to LSTM, but the GRU cells achieve comparable results, with obviously less computation time.

What's worth noting is that these nets because of their architecture, are very deep networks. This can result with VERY long training time, and could cause vanishing gradient problems. Vanishing gradients is very well-known problem in Deep Learning community, but RNN based architectures suffer from it even more. Let's portray the problem, imagine a student who is studying for a very long exam session, if he has to learn 100 pages of calculus, around page 30 he would already start forgetting what he has read at the very start. LSTM and GRU can deal with the above problem, as they select which lemmas and definitions they actually need to remember, resulting in more optimal memory usage.

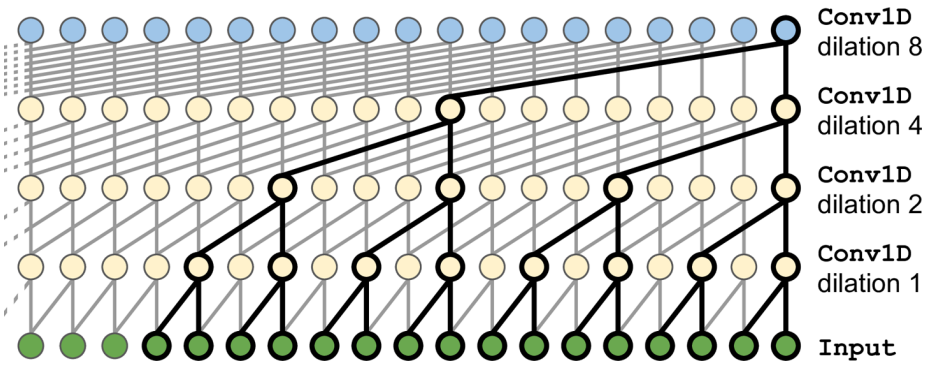

Figure 1: WaveNet

2.2 1D Convolutional Neural Networks

Another kind of Neural Networks applicable to Time Series, are Convolutional Neural Networks. Specifically 1D Convolutional Neural Networks (1D CNN). Initially these kinds of Nets were developed for image classification, where our models expect two-dimensional inputs representing an image's pixels, and their color channels (taking values that represent intensity of colours).

The same process designed for multi-dimensional data can be applied to one-dimensional sequences of data such as time series. CNN works well for identifying simple patterns hidden within data, which can be then used to form more complex patterns within higher layers. 1D CNN is especially very effective when you expect to derive features from short fixed-length segments of the overall data, and when the location of the feature within the segment is not relevant. This particular kind of Neural Networks was proven to be very effective by DeepMind researchers in 2016 when they developed WaveNet. we will not dive into deep mathematical formulations of it's inner workings, but we will rather give intuition how it tackles Time Series problems.

In the figure (1), we can see how the simplified WaveNet architecture process the incoming data. Let's see how it does it step by step. Input layer would be our Time Series data. Then each further layer is a 1D CNN with increasing dilation rate. Dilation rate is a parameter, that tells our layer to look up only combinations of observations that are Dilation Rate away from each other. What does this effectively mean? First 1D CNN layer looks for patterns that could be found between neighbour observations, next layer would look for dependencies between those pairs of neighbors, and so on. After applying multiple such layers, Neural Net would effectively learn not only small patterns between observations, but also long term dependencies. For example if we would monitor weather data over the years, first layer would identify that the weather from hour to hour doesn't change that much, second layer would find that in two hours intervals changes can be more visible. 6th layer (with dilation rate o 32) would already learn dependencies between days, while 11th

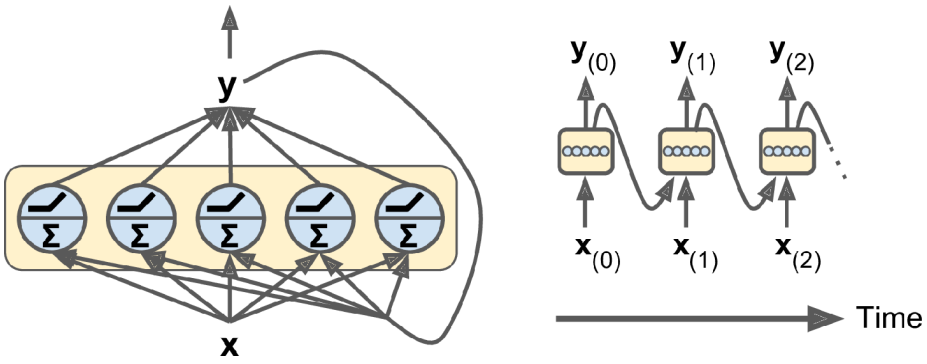

Figure 2: Recurrent neuron (left), unrolled through time (right)

(dilation rate of 1024) layer would learn patterns between months. If we had data for a few dozen years, we could even learn patterns that would be visible within decades. What's the advantage of such an architecture in terms of Time Series forecasting? As all the subtleties within data can be learnt by the Neural Nets, we do not need to remove seasonality or upward/downward trends in the data. Those would be learnt by the Neural Net itself (example how good WaveNet works can be seen in my project where I compared different Neural Nets). Applications of this Neural Nets are much wider than only time series, it is used in Text-To-Speech services, or generating music.

In-depth explanation of mathematical formulations of CNN's can be found in this article:

`http://www.cogprints.org/5869/1/cnn_tutorial.pdf`

3 Recurrent Neural Networks

Another architecture that was created to "predict the future", are Recurrent Neural Networks. In the following sections we will look at the fundamental concepts underlying RNN's, what problems do they face, and how they can be fixed.

3.1 Recurrent Neurons and Layers

Figure (2) on the left shows a singular recurrent neuron, that receives inputs, produces an output and sends the output back to itself. At each time step t, recurrent neuron receives the inputs $x_{(t)}$ and it's output from previous time step, $y_{(t-1)}$. As there is no output at first time step it is generally set to 0. Right side of the figure represents small network consisting of one neuron, against time axis. This is called unrolling the network through time.

This concept is easily generalized to a layer of recurrent neurons. At each time step t, every neuron receives both the input vector $x_{(t)}$, and output vector from the previous time step $y_{(t-1)}$.

Each recurrent neuron has two sets of weights: one for the inputs $x_{(t)}$ and the other for the previous time steps outputs, $y_{(t-1)}$. We will denote those as w_x and w_y. If we consider the whole recurrent layer instead of just one recurrent neuron, we can place all the weight vectors in two weight matrices, W_x and W_y. The output vector of the whole recurrent layer can then be computed as follows:

$$y(t) = \phi(W_x^T x(t) + W_y^T y(t-1) + b) \tag{1}$$

Where b is the bias vector, and $\phi(\cdot)$ is chosen activation function (e.g. Tanh or Relu). This equation is easily generalized for mini-batches:

$$y(t) = \phi(X_{(t)} W_x + Y_{(t-1)} W_y + b) \tag{2}$$

Where in this equation:

- $Y_{(t)}$ is a $m \times n\_neurons$ matrix made of layer's outputs at time step t for every observation in the mini-batch (where, m - the number of observations in the mini-batch, n_neurons - the number of neurons).
- $X_{(t)}$ is an $m \times n\_inputs$ matrix made of the inputs for all observations (where, n_inputs - the number of input features).
- W_x is an $n\_inputs \times n\_neurons$ matrix made of weights of the inputs of the current time step.
- W_y is an $n\_neurons \times n\_neurons$ matrix made of weights of the outputs of the previous time step.
- b is a vector of size n_neurons containing each neuron's bias term.

Notice that $Y_{(t)}$ is a function of $X_{(t)}$ and $Y_{(t\check{}1)}$, which itself is a function of $X_{(t-1)}$ and $Y_{(t-2)}$ (as it is fed it's own previous output and information from current step), which itself is a function of $X_{(t-2)}$ and $Y_{(t-3)}$, and so on. This makes $Y_{(t)}$ a function of all the inputs since time $t = 0$ (as there are no previous outputs at the first time those, as previously mentioned, are usually set to 0).

3.2 Memory Cells

Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, you could view it as if it had some kind of memory. A part of a neural network that preserves some state across time steps is called a memory cell (or simply a cell). A single recurrent neuron, or a layer of recurrent neurons, is a very basic cell, capable of learning short patterns.

In general a cell's state at time step t, denoted $h_{(t)}$ (the "h" stands for "hidden"), is a function of some inputs at that time step and it's state at the previous time step: $h_{(t)} = f(h_{(t\check{}1)}, x_{(t)})$. Its output at time step t, denoted $y_{(t)}$, is also a function of the previous state and the current inputs. In the case of the basic cells we have discussed so far, the output is simply equal to the state, but in more complex cells this is not always the case, which we will show later.

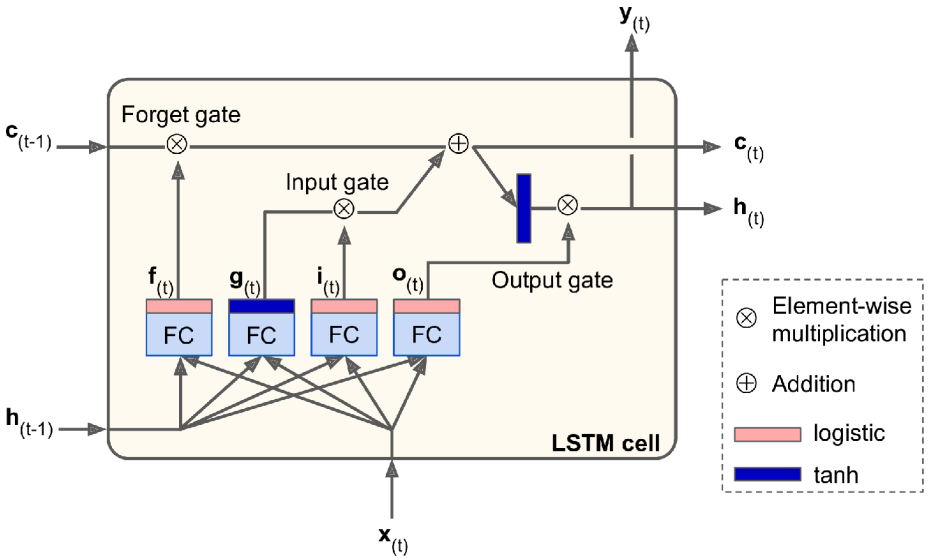

Figure 3: LSTM cell architecture

3.3 Input and Output Sequences

An RNN can take a sequence of inputs and produce a sequence of outputs. This type of sequence-to-sequence network is useful for predicting time series such as temperature values, you feed the model temperature values over the last N hours, and it must output the forecasted temperature for next few hours.

3.4 Training RNN's

To train a RNN, the trick is to unroll it through time (like we did in figure (2) on the right), and then simply use regular backpropagation. This strategy is called backpropagation through time.

In-depth formulation of backpropagation through time can be found in the link below. I will skip explaining it, as it is very similar to regular backpropagation.

`https://d2l.ai/chapter_recurrent-neural-networks/bptt.html`

4 LSTM Cells

The idea behind further research about RNN's was to handle problem of dealing with very long forecasting sequences, where our unrolled RNN's would become very deep networks. Therefore suffering from unstable gradients problems. To handle this problem, various types of cells with long-term memory have been introduced, and their astounding success resulted in basic RNN cells not being used anymore. Let's take look at LSTM first.

How does LSTM's architecture work? At first brief look at figure (3), it looks like a regular cell, except that it's state is split into two vectors: $h_{(t)}$ and $c_{(t)}$ (where "c" stands for "cell"). We can think of $h(t)$ as the short-term state and $c_{(t)}$ as long-term state.

So what's the idea behind LSTM? It's that the network can learn what information to store in the long-term state, what to delete, and what to use from it. Let's look what happens with long-term state $c_{(t-1)}$ while it goes through the network from left to right. It first passes forget gate, deleting some memories, and then it adds some information via the addition operation. The result of those operations becomes an output. Furthermore, after the addition operations, the long-term state is copied and goes through tanh function, afterwards the results is filtered out by output gate producing short-term state $h_{(t)}$ (which becomes output for the given time step, $y_{(t)}$).

4.1 Inner workings of LSTM

Having brief idea of the logic, let's look at the origin of new memories and how the aforementioned gates work.

Firstly, the current input vector $x_{(t)}$ and the previous short-term state $h_{(t-1)}$ are given as input to four different fully connected layers. Each one of those layers serves a different purpose:

Main workhorse layer, is the one with tanh activation that outputs $g_{(t)}$. In a basic RNN cell it would be the only layer included (processing current inputs $x_{(t)}$, and previous outputs $h_{(t-1)}$), and its outputs would be the only ones for RNN cell. In case of LSTM its output doesn't go straight out, but instead its most important memories are stored in the long-term state, while the rest are dropped.

Three other layers, are gate controllers. As they are logistic functions, their outputs range from 0 to 1. Those outputs are fed to different gates with element-wise multiplication. Therefore if logistic outputs are 0's, they close that part of the gate, and if they are equal to 1's they open it. To be more specific:

- Forget gate (connected to $f_{(t)}$) controls, which long-term states should be erased.
- Input gate (connected to $i_{(t)}$) controls, which parts of $g_{(t)}$ will be added to the memory of the long-term state.
- Output gate (connected to $o_{(t)}$) controls, what memories of the long-term state will be given as an output both to $h_{(t)}$ and $y_{(t)}$, at given time step.

Summing it up, LSTM cell learns to identify important inputs via the input gate. Then stores it in long-term state, keeping it there for as long as it is needed via forget gate. Then it extracts to outputs, chosen saved memories via output gate. Computations of each of the aforementioned gate and output functions

are as follow:

$$\begin{aligned}
\mathbf{i}_{(t)} &= \sigma\left(\mathbf{W}_{xi}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hi}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_i\right) \\
\mathbf{f}_{(t)} &= \sigma\left(\mathbf{W}_{xf}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hf}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_f\right) \\
\mathbf{o}_{(t)} &= \sigma\left(\mathbf{W}_{xo}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{ho}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_o\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hg}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_g\right) \\
\mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \\
\mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)
\end{aligned} \tag{3}$$

In all of the above equations:

- W_{xi}, W_{xf}, W_{xo} and W_{xg} are matrices containing weights of the connections of each layer, to the input vector $x_{(t)}$.
- W_{hi}, W_{hf}, W_{ho} and W_{hg} are matrices containing weights of the the connections of each layer, to the previous short-term state $h_{(t-1)}$
- b_i, b_f, b_o and b_g are the bias terms for the respective four layers.

Worth noting is that initially b_f needs to be set to vector of 1's, as otherwise we would forget everything at the beginning of the training!

5 GRU

Figure (4) shows GRU cell's architectures. Comparing it to LSTM we can immediately see that both state vectors are merged into one, $h_{(t)}$. There is only one gate controller $z_{(t)}$, that controls both the forget and input gates. Main difference between how the gates work is that if controller outputs a 1, the forget gate is open, and the input gate is closed. Conversely if opposite value is returned. In this cell, whenever new memory needs to be stored, it's storing location is cleared first.

Furthermore, there is no output gate, this results in full state vector being output at every time step. But there is also a new gate controller $r_{(t)}$ that controls, which part of the previous output will be shown to the main layer, $g_{(t)}$. The computation are as follows:

$$\begin{aligned}
\mathbf{z}_{(t)} &= \sigma\left(\mathbf{W}_{xz}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hz}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_z\right) \\
\mathbf{r}_{(t)} &= \sigma\left(\mathbf{W}_{xr}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hr}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_r\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hg}^{\top}\left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right) + \mathbf{b}_g\right) \\
\mathbf{h}_{(t)} &= \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + \left(1 - \mathbf{z}_{(t)}\right) \otimes \mathbf{g}_{(t)}
\end{aligned} \tag{4}$$

6 Conclusions

Different Neural Network architectures show great promise when it comes to forecasting longer time periods. They face different problems than the mathematically formulated models such as ARIMA, but can be a great asset to

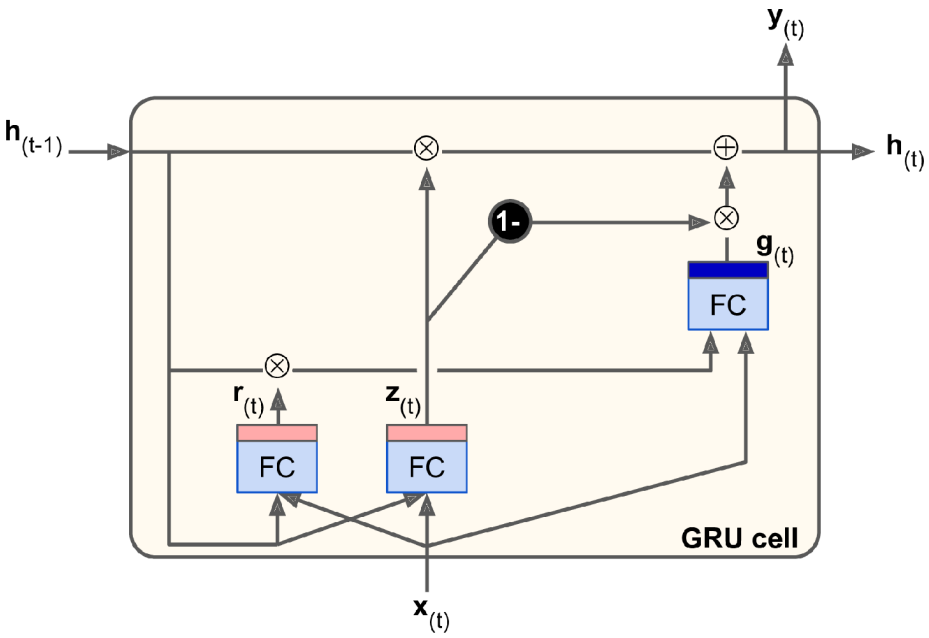

Figure 4: GRU cell architecture

mathematicians' arsenal when it comes to handling different time series problems. Current implementations within different programming languages, make the computations and creating efficient architectures, fairly effortless.

Above mentioned architectures can tackle many other problems, such as those mentioned in the section about WaveNet. Ideas presented in the sections of all the above types of nets, can be combined which could result in even better performance. As in every modelling task, finding best architecture takes a lot of work and trial, but it can be rewarded with results far surpassing those of the ARIMA/SARIMAX/GARCH models.

7 Sources

All of the figures come from the books listed below. Explanations of LSTM and GRU architectures were heavily inspired using those, and I've tried to sum up mathematical formulations used within those sources, whilst removing all code annotations:

- `https://www.amazon.com/Hands-Machine-Learning-Scikit-Learn-TensorFlow/dp/1491962291`
- `https://www.mimuw.edu.pl/~noble/courses/TimeSeries/1909arxivpreprintRecurrentNeuralNets.pdf`

Sources that helped me understand the problems better, and helped me understand logic of WaveNet:

- https://keras.io/api/layers/recurrent_layers/
- https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-exp
- https://d2l.ai/chapter_recurrent-neural-networks/bptt.html
- http://www.cogprints.org/5869/1/cnn_tutorial.pdf
- https://arxiv.org/abs/1609.03499