All you need to know about Recurrent Neural Networks

A beginner's guide into the implementation and data manipulation inside a RNN in TensorFlow

Photo by Laurent Naville on Unsplash

Introduction

Researchers came up with neural networks to model the behaviour of a human brain. But if you actually think about it, normal neural networks don't really do that much justice to its original intention. The reason for this statement is that feedforward vanilla neural networks cannot remember the things it learns. Each iteration you train the network it starts fresh, it doesn't remember what it saw in the previous iteration when you are processing the current set of data. This is a big disadvantage when identifying correlations and data patterns. This is where Recurrent Neural Networks (RNN)came into the picture. RNNs have a very unique architecture that helps them to model memory units (hidden state) that enable them to persist data, thus being able to model short term dependencies. Due to this reason, RNNs are extensively used in time-series forecasting to identify data correlations and patterns.

Even though RNNs have been around for some time, everyone seems to have their own confusing way of explaining it's architecture and no one really explains what happens behind the scenes. So let's bridge the gap, shall we? This post is aimed at explaining the RNN architecture in a more granular level by going through its functionality.

Is this for you?

If you have blindly made simple RNN models using TensorFlow before and if you have been finding it hard to understand about what

the inner workings of a RNN look like, then this article is just for you.

End Goal?

We will basically be explaining what happens behind the curtains when these two lines of TensorFlow code that are responsible for the declaration of the RNN and initiating the execution is run.

```
cell =
tf.contrib.rnn.BasicRNNCell(rnn_size,activation=tf.nn.tanh)

val1, state = tf.nn.dynamic_rnn(cell, inputs, dtype=tf.float32)
```

RNN Architecture

If you ever searched for architectural information about RNNs, the architecture diagrams you might get are rather confusing if you start looking into them as a beginner. I will use an example approach to explain the RNN architecture.

Before we get down to business, an important thing to note is that the RNN input needs to have 3 dimensions. Typically it would be batch size, the number of steps and number of features. The number of steps depicts the number of time steps/segments you will be feeding in one line of input of a batch of data that will be fed into the RNN.

The RNN unit in TensorFlow is called the “RNN cell”. This name itself has created a lot of confusion among people. There are many questions on Stackoverflow that inquire if “RNN cell” refers to one single cell or the whole layer. Well, it’s more like the whole layer.

The reason for this is that the connections in RNNs are recurrent, thus following a "feeding to itself" approach. Basically, the RNN layer is comprised of a single rolled RNN cell that unrolls according to the "number of steps" value (number of time steps/segments) you provide.

As we mentioned earlier the main speciality in RNNs is the ability to model short term dependencies. This is due to the hidden state in the RNN. It retains information from one time step to another flowing through the unrolled RNN units. Each unrolled RNN unit has a hidden state. The current time steps hidden state is calculated using information of the previous time step's hidden state and the current input. This process helps to retain information on what the model saw in the previous time step when processing the current time steps information. Also, something to note is that all the connections in RNN have weights and biases. The biases can be optional in some architectures. This process will be explained further in later parts of the article.

Since you now have a basic idea, let's break down the execution process with an example. Say your batch size is 6, RNN size is 7, the number of time steps/segments you would include in one input line is 5 and the number of features in one time step is 3. If this is the case, your input tensor (matrix) shape for one batch would look something like this:

Tensor shape of one batch = (6,5,3)

The data inside a batch would look something like this:

```
[[[1,2,3],[4,5,6],[7,8,9],[10,11,12],[13,14,15]],
[[2,3,4],[5,6,7],[8,9,10],[11,12,13],[14,15,16]],
[[3,4,5],[6,7,8],[9,10,11],[12,13,14],[15,16,17]],
[[4,5,6],[7,8,9],[10,11,12],[13,14,15],[16,17,18]],
[[5,6,7],[8,9,10],[11,12,13],[14,15,16],[17,18,19]],
[[6,7,8],[9,10,11],[12,13,14],[15,16,17],[18,19,20]]]
```

Fig 01 : Data representation inside a batch of data

Note: The data segmentation method used here is called the sliding window approach and is mostly used when doing time series analysis. You don’t have to worry about the data pre-processing process here.

When first feeding the data into the RNN. It will have a rolled architecture as shown below:

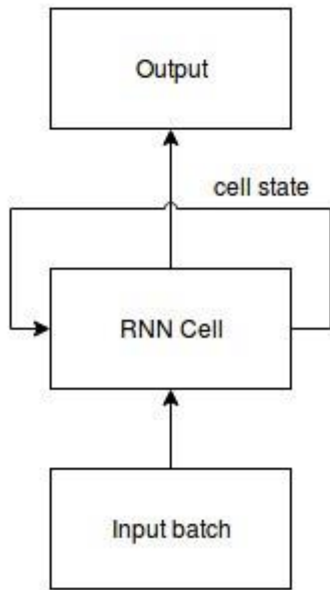

Fig 02: Rolled version of RNN

But when the RNN starts to process the data it will unroll and produce outputs as shown below:

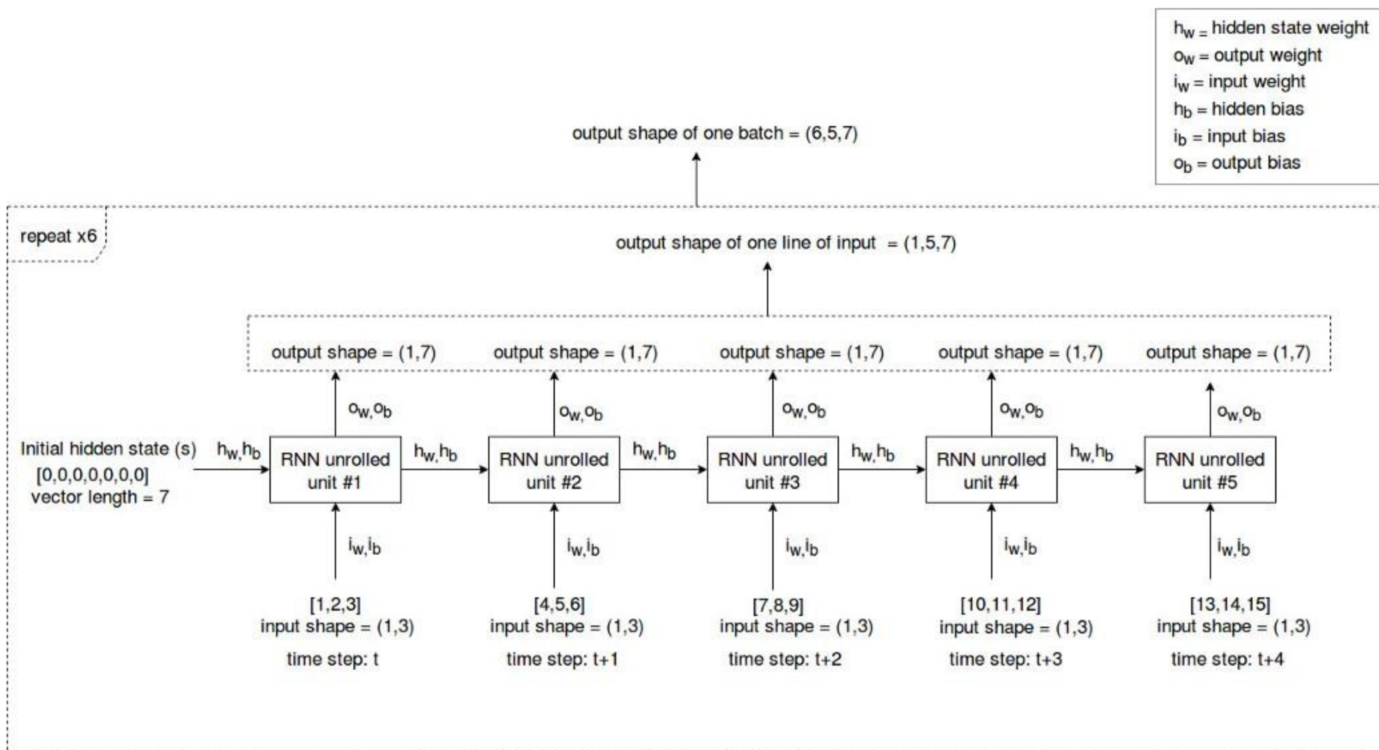

Fig 03: Unrolled version of RNN

Processing a batch:

When you feed a batch of data into the RNN cell it starts the processing from the 1st line of input. Likewise, the RNN cell will sequentially process all the input lines in the batch of data that was fed and give one output at the end which includes all the outputs of all the input lines.

Processing a single line of input:

In order to process a line of input, the RNN cell unrolls “number of steps” times. You can see this in the above figure (Fig 03). Since we defined “number of steps” as 5, the RNN cell has been unrolled 5 times.

The execution process is as follows:

- First, the initial hidden state (S), which is typically a vector of zeros and the hidden state weight (h) is multiplied and then the hidden state bias is added to the result. In the meantime, the input at the time step t ([1,2,3]) and the input weight (i) is multiplied and the input bias is added to that result. We can obtain the hidden state at time step t by sending the addition of the above two results through an activation function, typically tanh (f).

$$S_t = f((i_w[1,2,3]+i_b)+(h_w S_{intial}+h_b))$$

Fig 04: Hidden state calculation of time step t

- Then, to obtain the output at time step t, the hidden state (S) at time step t is multiplied by the output weight (O) at time step t and then the output bias is added to the result.

$$Output_t = (S_t O_w) + O_b$$

Fig 05: Output calculation of time step t

- When calculating the hidden state at time step t+1, the hidden state (S) at time step t is multiplied by the hidden state weight (h) and the hidden state bias is added to the result. Then as mentioned before the input at time step t+1 ([4,5,6]) will get multiplied by the input weight (i) and the input bias will be added to the result. These two results will then be sent through an activation function, typically tanh (f).

$$S_{t+1} = f((i_w[4,5,6]+i_b)+(h_w S_t+h_b))$$

Fig 06: Hidden state calculation of time step t+1

- Then, to obtain the output at time step t+1, the hidden state (S) at time step t+1 is multiplied by the output weight (O) at time step t+1 and then the output bias is added to the result. As you can see, when producing the output of time step t+1 it not only uses the input data of time step t+1 but also uses information of data in time step t via the hidden state at time step t+1.

$$Output_{t+1} = (S_{t+1} O_w) + O_b$$

Fig 07: Output calculation of time step t+1

- This process will repeat for all the time steps

After processing all time steps in one line of input in the batch, we will have 5 outputs of shape (1,7). So when all these outputs are concatenated together. the shape becomes (1,5,7). When all the input lines of the batch are done processing we get 6 outputs of size (1,5,7). Thus, the final output of the whole batch would be (6,5,7).

Note: All the hidden state weights, output weights and input weights have the same value throughout all the connections in a RNN.

Coming back to the 2 lines of code we stated earlier:

```
cell =
tf.contrib.rnn.BasicRNNCell(rnn_size,activation=tf.nn.tanh)

val1, state = tf.nn.dynamic_rnn(cell, inputs, dtype=tf.float32)
```

The 1st line basically defines the activation function and the RNN size of the RNN cell that we want to create. The 2nd line executes the processing procedure of the input data by feeding it into the RNN. The processing will happen according to what we discussed earlier. Finally, the output (value with shape (6,5,7)) of that batch will be assigned to the “val1” variable. The final value of the hidden state will be assigned to the “state” variable.

We have now come to the end of the article. In this article, we discussed the data manipulation and representation process inside of a RNN in TensorFlow. With all the provided information, I hope

that now you have a good understanding of how RNNs work in TensorFlow.