RNNs

Sylesh Suresh and Justin Zhang

January 2018

1 Introduction

Recurrent Neural Networks (RNNs) are a type of neural network that specializes on sequential data; that is, data that is dependant on prior data. For instance, RNNs are often used for natural language processing, because a given word is more likely to appear depending on the prior words. In this vein, RNNs have been applied to tasks like machine translation, image caption generation, and word prediction. However, RNNs have also seen moderate successes in image generation and audio generation.

2 Basic RNNs

In order to handle sequential data, a recurrent neural network has to be able to take in vectors of variable lengths. To do this, we design our network in the appropriate way. A RNN might look like the following:

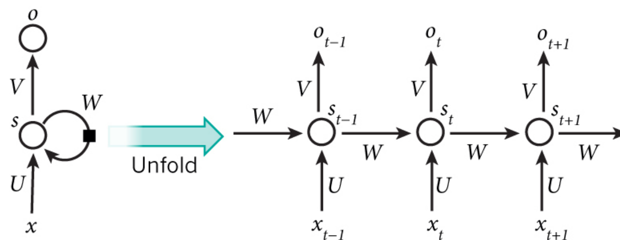

Figure 1: An unrolled RNN.

The figure above shows a RNN being *unrolled*, that is, being written out for the entire sequence x. If, for example, we had a sentence composed of five words, we would have the network be unrolled into five layers, one for each word. Note that the number of layers in the network matches the length of the sequence, so we can handle variable-length vectors unlike standard feedfoward neural networks. Moreover, the hidden state s_t depends on not only x_t but also the state of the previous layer s_{t-1}, allowing RNNs to process sequential data

better than feedforward networks. The output of a layer, o_t, is calculated from the layer's hidden state and is thus likewise reliant on the previous hidden state. Specifically,

$$s_t = f(Ux_t + Ws_{t-1})$$

where U and W are weight matrices. The first hidden state s_0 is usually initialized to $f(Ux_0)$. The function f is typically the tanh function or the ReLU function. The output o_t is calculated as such:

$$o_t = softmax(Vs_t)$$

where V is also a weight matrix. Note that U, V, and W are the same matrices for each layer as we are performing essentially the same operation at each time step t, meaning we only need to learn a few parameters. Also note that although we have an output for each layer, we may not necessarily use every one. For example, in sentiment analysis, we would only want the final output, the sentiment of the entire sentence, not the sentiment after each individual word.

2.1 Backpropagation Through Time

The equations for the hidden state s_t and predicted output o_t are:

$$s_t = \tanh(Ux_t + Ws_{t-1})$$

$$o_t = softmax(Vs_t)$$

Our total loss (error) function is:

$$L(y, \hat{y}) = \sum_t E_t \tag{1}$$

This is simply the sum of the losses at each time step. We can define the loss at each time step as:

$$E_t = -y_t \log o_t. \tag{2}$$

where y_t is the ground truth at a particular time step t. Note that the loss functions are dot products between the vectors y_t and the resulting vector of the element-wise logarithm of o_t. The weight matrices to be trained are U, V, and W. For V, by the chain rule, we find:

$$\frac{\partial E_t}{\partial V} = \frac{\partial E_t}{\partial o_t} \frac{\partial o_t}{\partial (Vs_t)} \frac{\partial (Vs_t)}{\partial V}$$

From this, we can derive:

$$\frac{\partial E_t}{\partial V} = (o_t - y_t) \times s_t$$

Where $\times$ is the cross product. For W, we can similarly apply the chain rule:

$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial o_t} \frac{\partial o_t}{\partial s_t} \frac{\partial s_t}{\partial W}$$

We can use the chain rule on the $\frac{\partial s_t}{\partial W}$ term again as s_t depends on s_{t-1} and we find:

$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial o_t}\frac{\partial o_t}{\partial s_t}\sum_{i=0}^{t}\frac{\partial s_t}{\partial s_i}\frac{\partial s_i}{\partial W}$$

From this, we can derive:

$$\frac{\partial E_t}{\partial W} = (o_t - y_t)V\sum_{i=0}^{t}\frac{\partial s_t}{\partial s_i}\frac{\partial s_i}{\partial W}$$

The partial derivative with respect to U is essentially identical:

$$\frac{\partial E_t}{\partial U} = (o_t - y_t)V\sum_{i=0}^{t}\frac{\partial s_t}{\partial s_i}\frac{\partial s_i}{\partial U}$$

2.2 Problems with Long-term Dependency

The $\sum_{i=0}^{t}\frac{\partial s_t}{\partial s_i}\frac{\partial s_i}{\partial W}$ term causes the vanishing gradient problem, which means that the network tends to forget words over time, which can prove troublesome. Let's take a word predictor, for instance. Given part of a sentence, predict the next word. This structure would work quite well for a sentence like "I ate some delicious ice ____"; "ice" provides a significant cue that the next word might be "cream." However, given a sentence like "I went to the ice cream store, but I forgot to bring my ____", it will likely have trouble guessing "wallet."

This is the problem of long-term dependencies; the network will "forget" words over time. We can remedy this issue with other types of RNNs such as LSTMs, which we will cover later on.

3 Types of RNNs

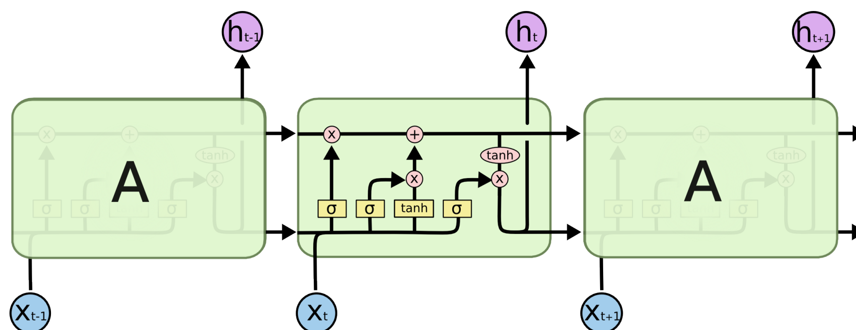

Figure 2: An LSTM cell.

An Long Short Term Memory network (LSTM) attempts to alleviate this issue by having a "forget gate," which allows the network to select parts of an input

vector to "remember" (i.e. pass on to the next cell) or "forget." It additionally has a mechanism to store and update the cell state. Let's walk through the cell step-by-step.

3.1 LSTM walk-through

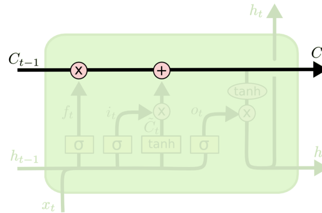

Figure 3: Cell state.

First off, the cell state. This is the pathway in which information flows from cell-to-cell. It was designed so that this would have little interaction, so as to store long-term data with little change.

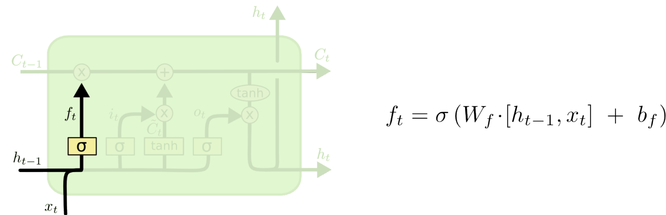

Figure 4: The forget gate.

This is the forget gate. We use a sigmoid activation conditioned on the input and prior output, which gives us outputs between 0 and 1. Since we're multiplying this value to the cell state, an output of 0 would correspond to completely forgetting everything, 1 remembering everything. This is done so as to update old information with more the more recent context.

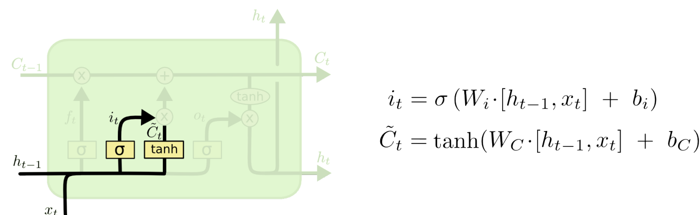

Figure 5: Updating the cell state.

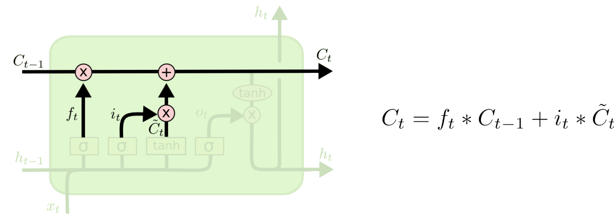

Figure 6: Outputting the cell state.

Next, we add a vector to the cell state based on the input and previous output, so as to update the state. Note that we're adding values between -1 and 1; tanh outputs -1 to 1, and the sigmoid outputs 0 to 1, which determines the magnitude of the update.

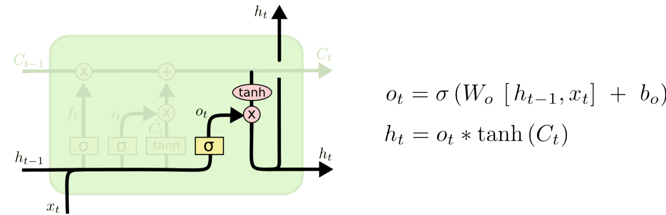

Figure 7: Outputting the final value.

Lastly, we want to output h_i. This is determined by the cell state, which is run through tanh to bound the values between -1 and 1. The sigmoid, whose output is conditioned on the previous cell output and current input, allows the cell to select which parts of the cell state to output (the sigmoid will output 0 for the parts it does not want to output, and vice-versa).

Then, the process repeats, with the next input.

3.2 GRUs

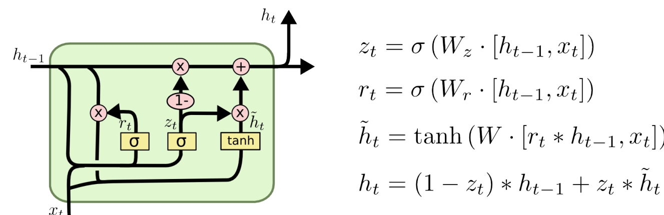

Figure 8: A GRU cell.

A Gated Recurrent Unit (GRU) is an alternative to an LSTM that is computationally simpler. It opts to combine the update and forget mechanisms into

an "update" mechanism, and also combines the hidden state and the cell state. GRUs are becoming increasingly popular as the perform as well as LSTMs on some tasks at a significantly lower computational cost.

4 Conclusion

There are many other types of RNNs; LSTMs and GRUs are two of the most popular. Additionally, there have been advancements on RNNs themselves; "attention" mechanisms are popular in tasks such as machine translation and caption generation.

For NLP tasks, we recommend investigating word vectors (Word2Vec, skip-thought vectors, etc.) to encode words to vectors. Also, using RNNs as the basis for an encoder/decoder model is widely used to translate sequences (e.g. machine translation). We encourage you to further investigate these areas!