Recurrent Neural Networks Overview

Benjamin Santana
Universidad Panamericana
Email: 0211518@up.edu.mx

I. Introduction

Traditional machine learning methods assume data points to be independently and identically distributed, but when talking about language, speech and time series, one data item depends on the item that precede or follow it. These data points are what we would call sequence data.

Sequential information is everywhere. For example speech, we can think of it as a sequence of basic units called phonemes. In English, words are not random. They may depend on the words that come before and after them. This means that in order to understand human language we need to understand sequences.

Modeling sequences involves maintaining a hidden state. When the model *process* each item (for example a word in a sentence) the hidden state is updated. This hidden state represents everything seen by the sequence so far. We usually represent this hidden state as a vector.

Conventional deep networks (also called feed-forward) fail to capture the sequence information in training. We use *Recurrent Neural Networks* for these types of problems. As you probably guessed for the title, the goal of this brief paper is to get an understanding of them, how they work and showcase an example.

II. Recurrent Neural Networks

The purpose of a recurrent neural network is to model sequences of tensors. Recurrent Neural Networks (RNN) are a family of models, meaning there are different forms on how to make an RNN. Nevertheless, the goal of them all is basically the same, we want to learn a representation of a sequence. This is done by maintaining a hidden state vector that captures the state of the sequence. We get this vector from both the input vector and the previous hidden state vector. We are going to focus on a type of RNN architecture called the *gated network.*

We will use this architecture as opposed to the Vanilla/Elman architecture (which is like the basic one) because of a couple of issues that it presents. Vanilla RNNs have a hard time retaining long-range information because we update the hidden state vector no matter what at each time step, therefore we have no control over which values are retained and which are discarded, we leave this decision completely to the input. This is not ideal, we want the RNN to update when needed. The second issue with Vanilla RNNs is the tendency to cause gradients to spiral out of control, this issue is not particular to RNN, and we can solve it using *ReLUs* and that kind of stuff, but those methods do not work as well as the technique called gating.

To understand gating, let us think of the sum $a+b$, say you want to control how much b gets added to a. So you can write Equation 1, this way if λ is 0 then nothing of b is added, but if λ is 1 all of b is added. You could even say that λ act as a *gate*. How does this look in a recurrent neural network?

$$a + \lambda b, \tag{1}$$

In a Vanilla RNN, the recurrent update could be calculated something like Equation 2. Where h_{t-1} is the previous-hidden state, and the current input is x_t.

$$h_t = h_t + F(h_{t-1}, x_t) \tag{2}$$

This has the two issues mentioned above (about using Vanilla RNNs). How would our gate (λ in Equation 1) look here? We could have a function of the previous hidden state vector h_{t-1} and the current input x_t that produces a value between 0 and 1. See Equation 3

$$h_t = h_t + \lambda(h_{t-1}, x_t)F(h_{t-1}, x_t) \tag{3}$$

Now, it is clear that this function λ is controls how much of the current input gets to update the state of h_{t-1}. λ is usually a sigmoid function to get a value between 0 and 1.

A. Long Short-Term Memory (LSTM)

In Long Short-Term Memory (LSTM) this basic intuition is extended to incorporate not only conditional updates, but also intentional forgetting of the values in previous hidden states. The forget part happens by multiplying the previous hidden state value h_{t-1} with another function μ, see Equation 4.

$$h_t = \mu(h_{t-1}, x_t)h_t + \lambda(h_{t-1}, x_t)F(h_{t-1}, x_t) \quad (4)$$

So μ is another gating function. In an actual LSTM description, this becomes complicated because the gating function is parameterized, which leads to a complex sequence of operations. I do not plan to cover all the theory here, please refer to [1] for more information.

B. Gated Recurrent Units (GRU)

In Gated Recurrent Units, the intuition of conditional updated and intentional forgetting is also extended. This is achieved by introducing two gating mechanisms, an update gate and a reset gate.

- *Update Gate*, this controls to which extent the previous hidden state is updated with new information in extent the previous hidden state is updated with new information.
- *Reset Gate*, it determines how much of the previous hidden state should be ignored when considering new input.

The GRU architecture introduces a more concise update mechanism compared to LSTM, as it combines the forget and input gates into a single update gate. However, it achieves a similar goal of selectively updating and forgetting information over time.

It's worth noting that, like LSTM, GRU also utilizes learned parameters to facilitate these gating mechanisms. These parameters are optimized during the training process to capture the relevant patterns and dependencies in the sequential data.

Fortunately, when coding (in PyTorch) you can simply replace the `nn.RNN` with `nn.LSTM` or `nn.LSTMCell` to switch to an LSTM, as you might have already guess `nn.GRU` changes to a GRU cell.

III. Example: Generate Surnames

I got this example from Natural Language Processing with PyTorch [2]

In this example we will go over a simple sequence prediction task, we will be generating surnames using an RNN. Meaning the RNN will compute a probability over a set of possible characters in the surname. The dataset is a collection of last names and their country of origin. The model utilizes a specific nationality embedding as the initial hidden state of the RNN to allow the model to bias its predictions of sequences. I will walk you through to each step of the example.

A. Preprocessing the data

We create a class that basically uses a pandas dataframe to load the dataset, then a *vectorizer* is constructed which encapsulates the token-to-integer mappings required for the model.

There are three data structures that transforms each surnames sequence of characters into its vectorized forms: A maps the individuals tokens to integers, B coordinates the integer mappings, C groups B's results into mini-batches.

Let us talk a bit more about B. In a sequence prediction problem , the training part expects two sequences of integers, the first represents the token input and the other represents what we want the model to predict. Commonly, we want to predict the sequence we are training on, such as with the surnames in this example. This means that we only have a single sequence of tokens to work with, and construct the input and output by staggering the sequence.

All this means is that this approach allows us to use a single sequence of tokens for training our model to predict the same sequence.

Since we want the model to consider the nationality of the surname it is generating, the model has to be influenced in some way, we use a technique called parameterization.

We start by assigning each nationality a vector. The model uses this vector to initialize the hidden state of the RNN. This way, we add a bias to its predictions.

Say, for example, we encounter an Irish surname. The Irish nationality vector represents Irish naming conventions, it may bias the model to generate surname sequences that often start with *Mc* or *O* since these are common prefixes in the Irish surnames.

B. The Actual Layers

The network is defined by these layers:

- *Character Embedding Layer*, this layer maps character indices to character embeddings. It takes as input a tensor of character indices and returns a tensor of corresponding character embeddings.
- *Nationality Embedding Layer*, this layer maps nationality indices to nationality embeddings. It takes as input a tensor of nationality indices and returns a tensor of corresponding nationality embeddings.
- *Recurrent Layer (GRU)*, it is the same as explained in Section II-B
- *Fully Connected Layer*, this layer is a linear transformation that maps the hidden states produced by the recurrent layer to the character vocabulary size. It takes as input the hidden states and produces output logits for each character in the vocabulary.
- Then we apply a softmax activation, and we reshape the output tensor to the original shape

C. Training and Results

The training routine follows the standard formula. For a single batch of data, apply the model and compute the prediction vectors to then compute a loss value. Using the loss value and an optimizer, compute the gradients and update the weights of the model using those gradients.

Then we can do a sampling from the model, we get the following results:

```
Sampled for Polish:
-  Ardbanis
-  Emitdo
-  Altillad
Sampled for Portuguese:
-  Berte
-  Balras
-  Tenta
Sampled for Russian:
-  Gadekok
-  Nanamov
-  Zanev
Sampled for Scottish:
-  Pirer
-  Serignern
-  Sarg
Sampled for Spanish:
-  Carcha
-  Aritrele
-  Bariie
```

What can we learn from inspecting the output? We can see that first the surnames appear to follow several morphological patterns, then the names appear to be of one nationality or another. The model is indeed picking up on some patterns of orthography for surnames. I will not lie to you, I have not heard anyone's surname being Carcha, but I can see it happening.

D. Tips

Before jumping into the conclusion part, the book offered some advice in the form of tips and tricks when using recurrent neural networks that I wanted to record here for future reference.

- *Use GRUs over LSTMs*, GRUs provide almost comparable performance to LSTMs and need far fewer parameters and compute resources.
- *Use Adam as optimizer*, it is reliable and typically converges faster than the alternatives. If for some reason your models are not converging with Adam, switch to stochastic gradient descent.
- *Early Stopping*, with sequence models, it is easy to overfit, the book recommends that we stop the training procedure early.

IV. Conclusion

In conclusion, recurrent neural networks (RNNs) have proven to be a useful class of models for sequential data processing like speech (Natural Language Processing). This brief paper has explored some key ideas behind them and a short example. In the future, we could try to create sentences based on a book or some text. It is the next logical step, we go from character to words and so on.

I am not going to lie, it was quite hard to wrap my head around this topic, I do not know why but talking about how the data points are *sequences* and all that, was confusing for me. It was surprising also, since I have studied time series from a statistical point of view before. But natural language processing was a challenging but fun topic.

I want to get more projects under my belt, try to solve other problems using the techniques learned here. Sadly the course time is short and even though we saw really cool applications and projects, I think we just grasp the surface of all RNN can do. Will definetly do more projects in the in the future.

References

[1] C. Olah, "Understanding lstm networks," http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015.

[2] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*. O'Reilly Media, 2009. [Online]. Available: https://learning.oreilly.com/library/view/natural-language-processing/9781491978221/