Introduction to RNNs, Sequence to Sequence Language Translation and Attention

Omer Sakarya Jun 23, 2019 · 7 min read

The goal of this post is to briefly introduce RNNs (Recurrent Neural Networks), sequence to sequence language translation (seq2seq) and Attention. I will try to make it as simple as possible. You only need to know:

1. Linear algebra
2. Neural networks

In case you feel rusty on the topics, feel free to review them before you start reading by clicking on the links above.

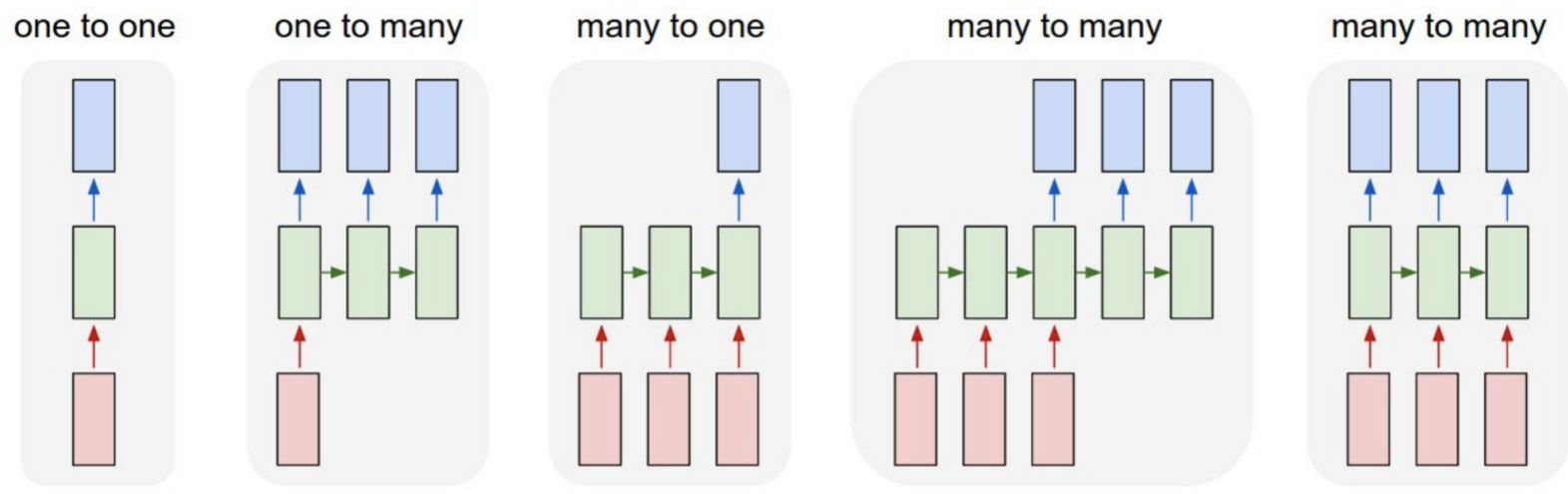
human brain, neurons are connected in a more complex way then they do in a feed-forward neural networks. Turns out that, researchers continuously find new ways to align neurons to get them to do something useful. RNNs are one of them.

In a neural network, you have one output per input. You have an image and you have a label, for instance. There is no way you can input image after image on Neural Networks and get an output based on all of them. The nature of neural networks make them unable to process sequential data.

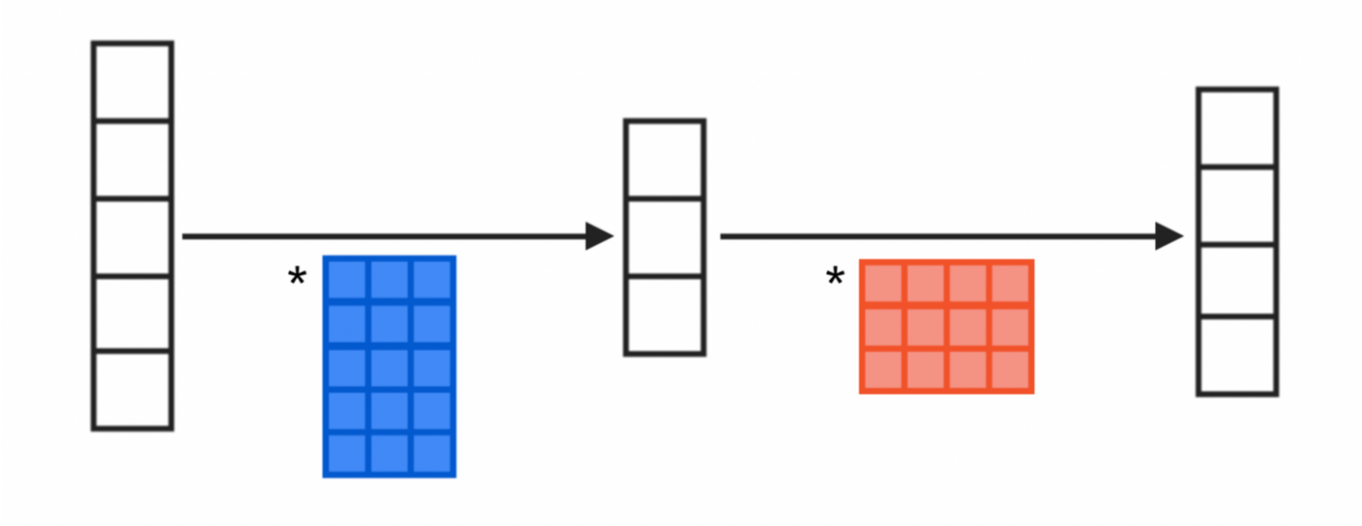
On the other hand, RNNs work very well with sequential data. They have a mechanism of "remembering" the previous inputs and producing an output based on all of the inputs. This makes them well-suited for sequential type of data such as text, audio, video or any time-series data.

The diagram below shows five variations of RNNs. Red vectors are inputs whereas blue vectors are the outputs. First one is a one input to one output situation. Which is essentially a neural network. The others are capable of sequential inputs and outputs, such as:

one to many: image -> caption sentence
many to one: sentence -> sentiment (positive / negative label)
many to many: a sentence in English -> a sentence in Turkish
the other many to many: frames of video -> coordinates of bounding boxes around an object

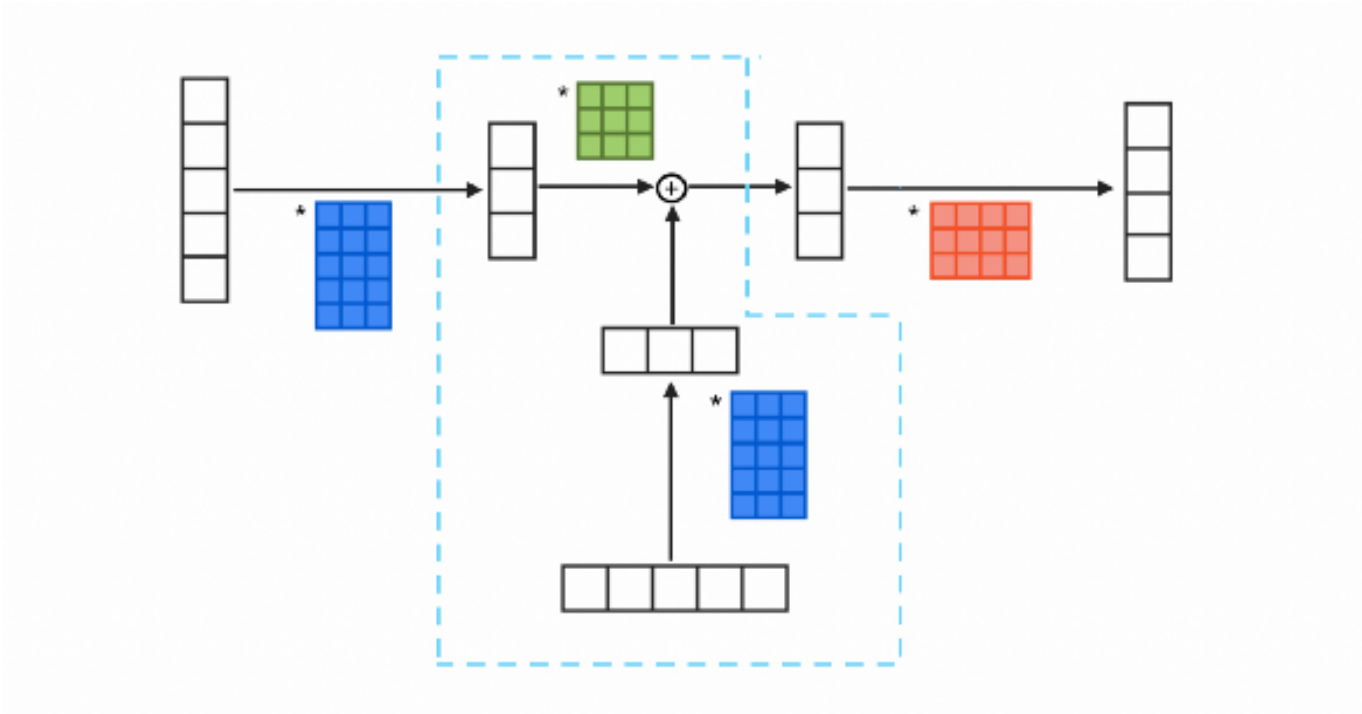


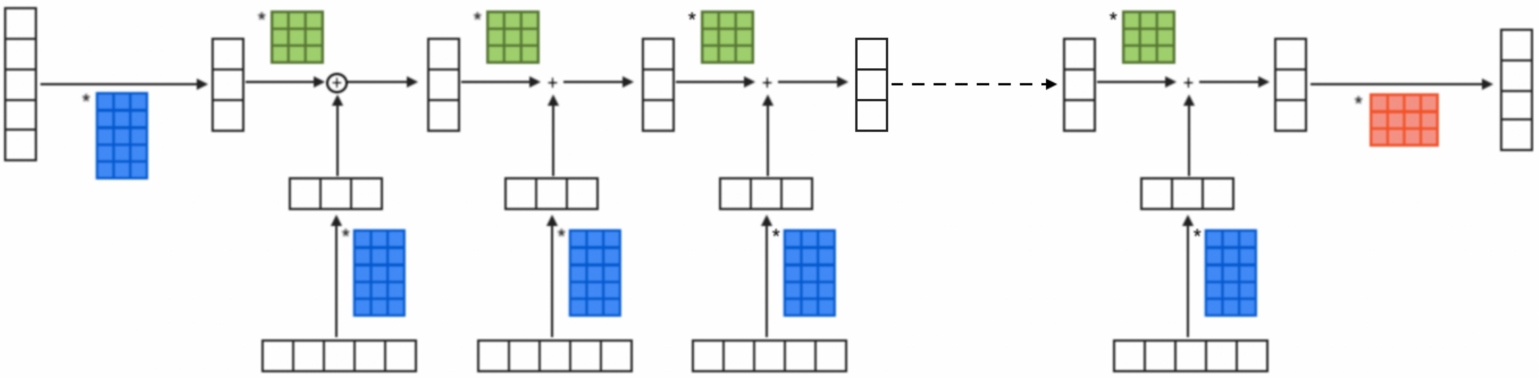
Let's start to look under the hood by looking at a feed-forward neural network with one hidden layer. Since each layer is essentially a linear transformation and therefore a



An input of size 5 goes through a matrix multiplication with a 5x3 matrix, producing the input for the hidden layer, vector of size 3. Then, it gets multiplied by a 3x4 matrix, producing an out put of 4. Pay attention to the vector of size 3 here. In RNNs, we will call it the *hidden state*.

Let's take a look at slightly modified version of that. We have added just the area inside the dashed rectangle. Also, we have added one more parameter matrix, 3x3 in this case. What happens is that, we are adding one more input from the bottom. All inputs need to be of the same size. Because it also goes through the same, blue matrix multiplication as the bottom one. Notice that, the hidden output we had before gets multiplied by a square matrix before being combined with the new input. We never know what that operation is. It is just something that the RNN learns to do itself while training.





What we are essentially doing is that, we are applying a recurrent formula to every input from the sequence. What we have at the very end depends on the whole input sequence.

$$h_t = f_W(h_{t-1}, x_t)$$

new state — old state — input vector at some time step

some function with parameters W

Notice: the same function and the same set of parameters are used at every time step.

To delve deeper:

To picture it with the parameters we initially had:

What you have just read is an explanation of Vanilla RNNs, the most basic form of RNNs. It's easy to learn, however, they are hard to train. The reason is that, the values in the hidden states tend to exponentially explode or vanish. More advanced RNNs such as LSTM and GRUs mitigate that problem by adopting more sophisticated mechanisms. There is an excellent blog post by Michael Nguyen on LSTMs and GRUs. I would strongly suggest you to check it out for further reading.

Part 2: RNNs with words: Language Models

Language Models try to predict the next word, given a sequence of words. It might sound useless at first, as it did to me. However, they are usually components of other more useful Natural Language Processing systems.

Let's look at a diagram very similar to our Vanilla RNN.

Here, the inputs to the RNN are word embeddings, not actual words. If you don't know what a word embedding is, think hash function, but for vectors. Instead of representing words from vocabulary as one-hot encoded vectors, embeddings represent them as lower dimension vectors. It provides the benefit of faster linear operations because of lower dimension vectors. That's what *E* means in the diagram.

We, Wh, and U are blue, green and orange matrices in the previous diagrams, respectively. This diagram depicts training of a Language Model. "The students opened their exams…" are from the training corpus; the sentence actually exists. We are penalizing the Language model by a loss function comparing pairs of y_hat^(1) and x^(2), y_hat^(2) and x^(3) and so on. Notice that as we go on towards the right, model also keeps the hidden state so that it keeps track of the past words.

After trained for a while, a language model can generate text by predicting the next for over and over. Notice that, the predicted words, y_hat^(i) are the input for the next step of operation.

Part 3: Sequence to Sequence Machine Translation

RNNs are also capable of doing natural language translation, aka. machine translation. It involves two RNNs, one for the source language and one for the target language. One of them is called an *encoder,* and the other one *decoder.* The reason is that, the first one encodes the sentence into a vector and the second one converts the encoded vector into a sentence in target language.

Encoder

The encoder is an RNN. Given the source sentence, it produces an encoding. Notice the benefit of RNNs here. The size of input sentence can be any size.

Decoder

The decoder is a separete RNN. Given the encoded sentence, it produces the translated sentence in target language. Notice the purple, dashed arrows. Every predicted word is an input to the next prediction. This is the recurring nature of RNNs and it is very interesting to me that they can do a significant work such as language translation by being connected like that.

At each step of the decoder, given the hidden state and input word vector as input, the argmax of a probability distribution of words is chosen as the most probable word. This is called greedy decoding. The problem with that is, the decoder is going one by one. At one point, if the sentence does not make sense, it can not go back.

Also, the whole sentence being encoded in a single vector is hard to translate into a sentence. After the encoding step, it is essentially a mapping from sentence to a vector. It it really hard for the decoder to derive an order for the words in the target language. That's where attention comes into play.

Part 4: Attention

Attention lets the decoder to focus on specific parts of the input sentence for each output word. This helps the input and output sentences to align with one another.

In the picture, the blue ovals are simply dot products with the sentence encoding and hidden states.

For the French sentence above, the highest attention score belongs to “les”. This will cause the decoder to start the translated sentence with “the”. When I first saw that diagram, I thought that, if the correct translation started with “pauvres”, there is no way the decoder would get it right. However, since this neural network is supposed to be trained end-to-end, it will also learn to produce correct attention scores along the way.

Closing Note

For complete beginners, this might be a lot of information in a short text. Rachel Thomas’s way of introducing to RNN at a class at USF have inspired me to write this blog post. I wanted to sum up what caught my interest on a high-level. Feel free to ping me if you have any questions! 🎉 🍇

Machine Learning Rnn Recurrent Neural Network Seq2seq Attention