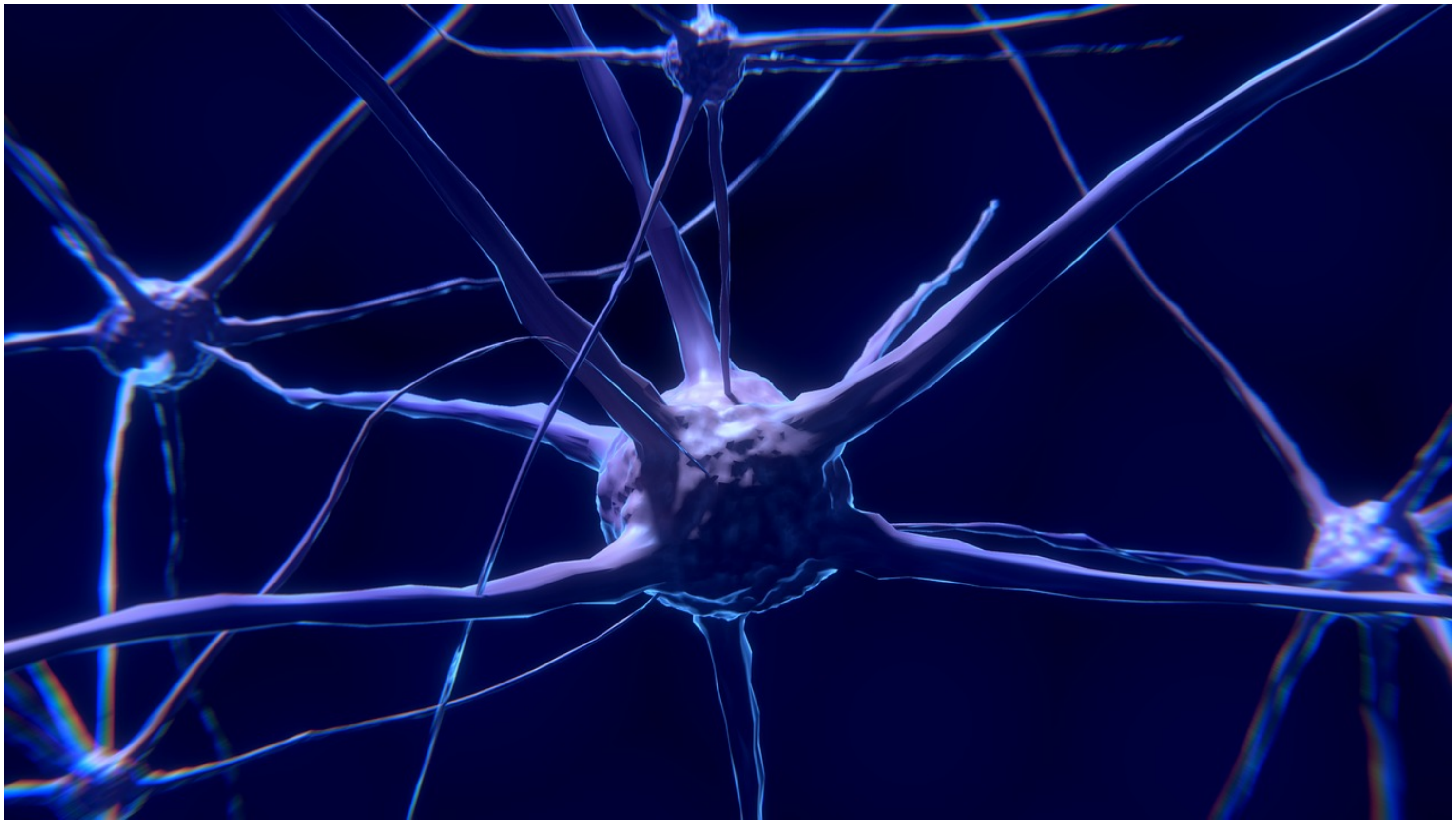

https://pixabay.com

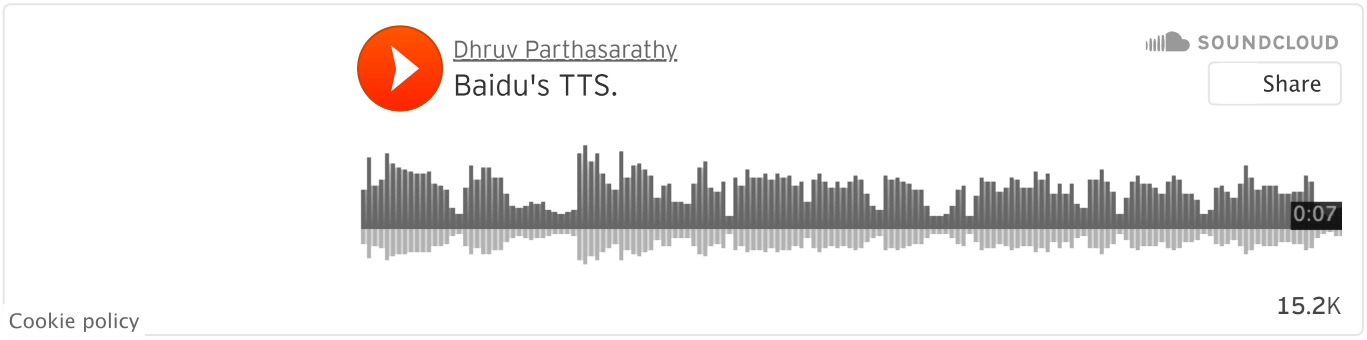
How Recurrent Neural Networks work

Simeon Kostadinov Follow
Dec 2, 2017 · 6 min read

You have definitely come across software that translates natural language (Google Translate) or turns your speech into text (Apple Siri) and probably, at first, you were curious how it works.

In the last couple of years, a considerable improvement in the science behind these systems has taken place. For example, in late 2016, Google introduced a new system behind their Google Translate which uses state-of-the-art machine learning techniques. The improvement is remarkable and you can test it yourself.

Another astonishing example is Baidu's most recent text to speech:



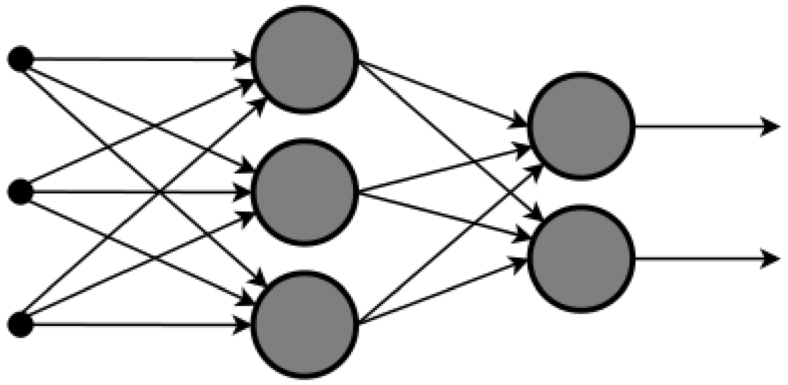
Credits to Dhruv Pathasarathy for the amazing demo.

So what do all the above have in common? They deal with **sequential data** to make predictions. Okay, but how that differs from the well-known cat image recognizers?

Imagine you want to say if there is a cat in a photo. You can train a **feedforward neural network** (typically CNN-Convolutional Neural Network) using multiple photos with and without cats.

> *In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network. — Wikipedia*

These networks are primarily used for pattern recognition and can be illustrated as follows:



Feedforward neural network

Conversely, in order to handle sequential data successfully, you need to use **recurrent (feedback) neural network**. It is able to 'memorize' parts of the inputs and use them to make accurate predictions. These networks are at the heart of speech recognition, translation and more. So let’s dive into a more detailed explanation.

What is a Recurrent Neural Network?

Training a typical neural network involves the following steps:

1. Input an example from a dataset.
2. The network will take that example and apply some complex computations to it using randomly initialised variables (called weights and biases).
3. A predicted result will be produced.
4. Comparing that result to the expected value will give us an error.
5. Propagating the error back through the same path will adjust the variables.
6. Steps 1–5 are repeated until we are confident to say that our variables are well-defined.
7. A predication is made by applying these variables to a new unseen input.

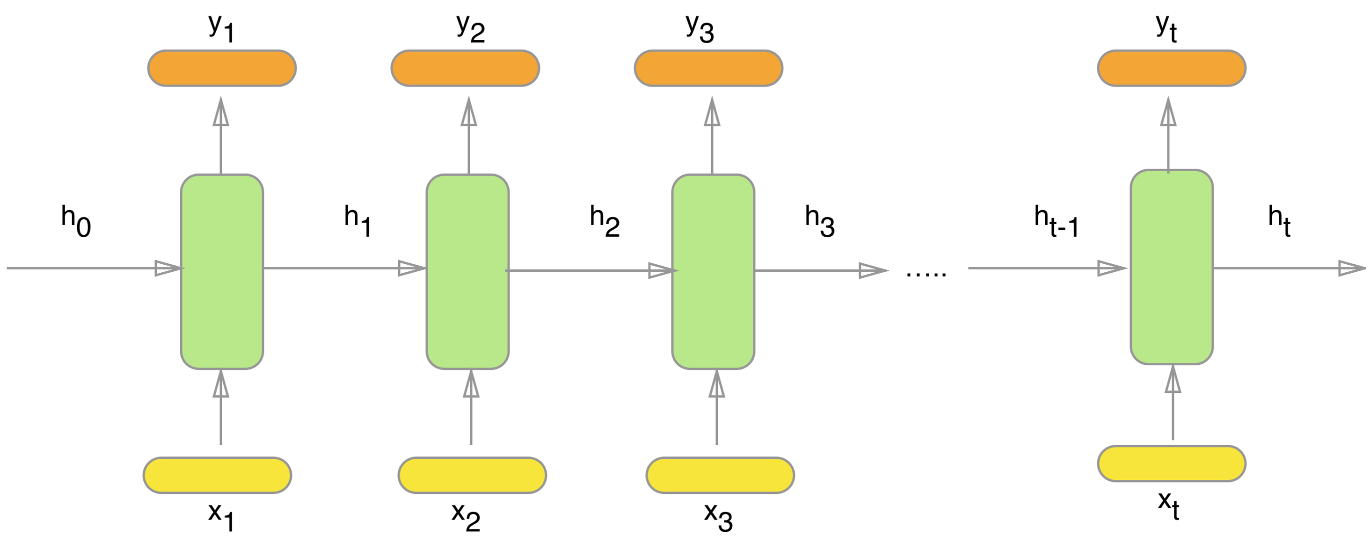
Of course, that is a quite naive explanation of a neural network, but, at least, gives a good overview and might be useful for someone completely new to the field.

Recurrent neural networks work similarly but, in order to get a clear understanding of the difference, we will go through the simplest model using the task of predicting the next word in a sequence based on the previous ones.

First, we need to train the network using a large dataset. For the purpose, we can choose any large text (“War and Peace” by Leo Tolstoy is a good choice). When done training, we can input the sentence “Napoleon was the Emperor of…” and expect a reasonable prediction based on the knowledge from the book.

So, how do we start? As explained above, we input one example at a time and produce one result, both of which are single words. The difference with a feedforward network comes in the fact that we also need to be informed about the previous inputs before evaluating the result. So you can view RNNs as multiple feedforward neural networks, passing information from one to the other.

Let’s examine the following schema:



Recurrent neural network

Here *x_1, x_2, x_3, …, x_t* represent the input words from the text, *y_1, y_2, y_3, …, y_t* represent the predicted next words and *h_0, h_1, h_2, h_3, …, h_t* hold the information for the previous input words.

Since plain text cannot be used in a neural network, we need to encode the words into vectors. The best approach is to use **word embeddings** (word2vec or GloVe) but for the purpose of this article, we will go for the one-hot encoded vectors. These are *(V,1)* vectors (*V* is the number of words in our vocabulary) where all the values are 0, except the one at the *i-th* position. For example, if our vocabulary is *apple, apricot, banana, …, king, … zebra* and the word is *banana*, then the vector is *[0, 0, 1, …, 0, …, 0]*.

Typically, the vocabulary contains all English words. That is why it is necessary to use word embeddings.

Let’s define the equations needed for training:

$$1)\quad h_t = f(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$$

$$2)\quad y_t = softmax(W^{(S)}h_t)$$

$$3)\quad J^{(t)}(\theta) = \sum_{i=1}^{|V|} (y'_{t_i} log\, y_{t_i})$$

- 1) —holds information about the previous words in the sequence. As you can see, *h_t* is calculated using the previous *h_(t-1)* vector and current word vector *x_t*. We also apply a non-linear activation function *f* (usually tanh or sigmoid) to the final summation. It is acceptable to assume that *h_0* is a vector of zeros.
- 2) — calculates the predicted word vector at a given time step *t*. We use the softmax function to produce a *(V,1)* vector with all elements summing up to 1. This probability distribution gives us the index of the most likely next word from the vocabulary.
- 3) — uses the cross-entropy loss function at each time step *t* to calculate the error between the predicted and actual word.

If you are wondering what these *W's* are, each of them represents the weights of the network at a certain stage. As mentioned above, the weights are matrices initialised with random elements, adjusted using the error from the loss function. We do this adjusting using back-propagation algorithm which updates the weights. I will leave the explanation of that process for a later article but, if you are curious how it works, Michael Nielsen's book is a must-read.

Once we have obtained the correct weights, predicting the next word in the sentence "Napoleon was the Emperor of…" is quite straightforward. Plugging each word at a different time step of the RNN would produce *h_1, h_2, h_3, h_4*. We can derive *y_5* using *h_4* and *x_5* (vector of the word "of"). If our training was successful, we should expect that the index of the largest number in *y_5* is the same as the index of the word "France" in our vocabulary.

Problems with a standard RNN

Unfortunately, if you implement the above steps, you won't be so delighted with the results. That is because the simplest RNN model has a major drawback, called **vanishing gradient problem,** which prevents it from being accurate.

In a nutshell, the problem comes from the fact that at each time step during training we are using the same weights to calculate *y_t*. That multiplication is also done during back-propagation. The further we move backwards, the bigger or smaller our error signal becomes. This means that **the network experiences difficulty in memorising words**

from far away in the sequence and makes predictions based on only the most recent ones.

That is why more powerful models like LSTM and GRU come in hand. Solving the above issue, they have become the accepted way of implementing recurrent neural networks.

Bonus

Finally, I would like to share my list with all resources that made me understand RNNs better:

Warm-up:

- Paperspace Blog — Recurrent Neural Networks
- WILD ML — Introduction to RNNs

Go deeper:

- Andrej Karpathy blog — The Unreasonable Effectiveness of Recurrent Neural Networks
- Stanford CS224n — Lecture 8: Recurrent Neural Networks and Language Models (recommend going through the whole course + exercises)

Advanced (grasping the details):

- Tensorflow — Recurrent Neural Networks
- arXiv paper — A Critical Review of Recurrent Neural Networks for Sequence Learning

I hope this article is leaving you with a good understanding of Recurrent neural networks and managed to contribute to your exciting Deep Learning journey.

Thank you for the reading. If you enjoyed the article, give it some claps 👏 . Hope you have a great day!

Machine Learning Recurrent Neural Network Lstm Speech Recognition Text To Speech