Deep Learning for NLP

Agenda

1. Word Embeddings
2. RNN
3. LSTM

Word Embeddings

One-hot encoding

How are you

This is you

Unique words :- How, are, you, this, is

How :- [1, 0, 0, 0, 0]

are :- [0, 1, 0, 0, 0]

.

.

is :- [0, 0, 0, 0, 1]

good player = 2^1/2

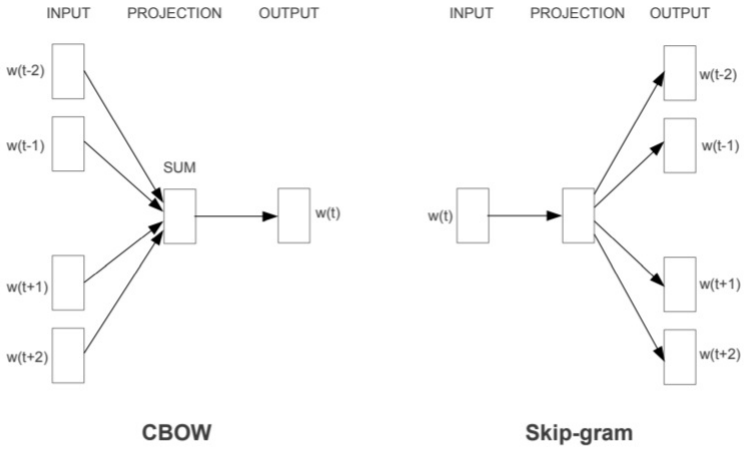
great player = 2^1/2

bad player = 2^1/2

These are also sparse vectors

50000 words in vocabulary -> 50000 sized vector

Word2vec



Best batsman in world Virat Kohli

batsman in - Best

Best in world - batsman

Best batsman world Virat - in

Two ways to solve the same problem

Input is one-hot encoded vector

The output of hidden layer is used as word-embedding

king + woman - man = queen

Word2vec properties

1. Can find similarities between words
2. woman+king-man closer to queen

How does Word2Vec produce word embeddings?

Word2Vec uses a trick you may have seen elsewhere in machine learning. Word2Vec is a simple neural network with a single hidden layer, and like all neural networks, it has weights, and during training, its goal is to adjust those weights to reduce a loss function. However, Word2Vec is not going to be used for the task it was trained on, instead, we will just take its hidden weights, use them as our word embeddings, and toss the rest of the model *(this came out a little bit evil)*.

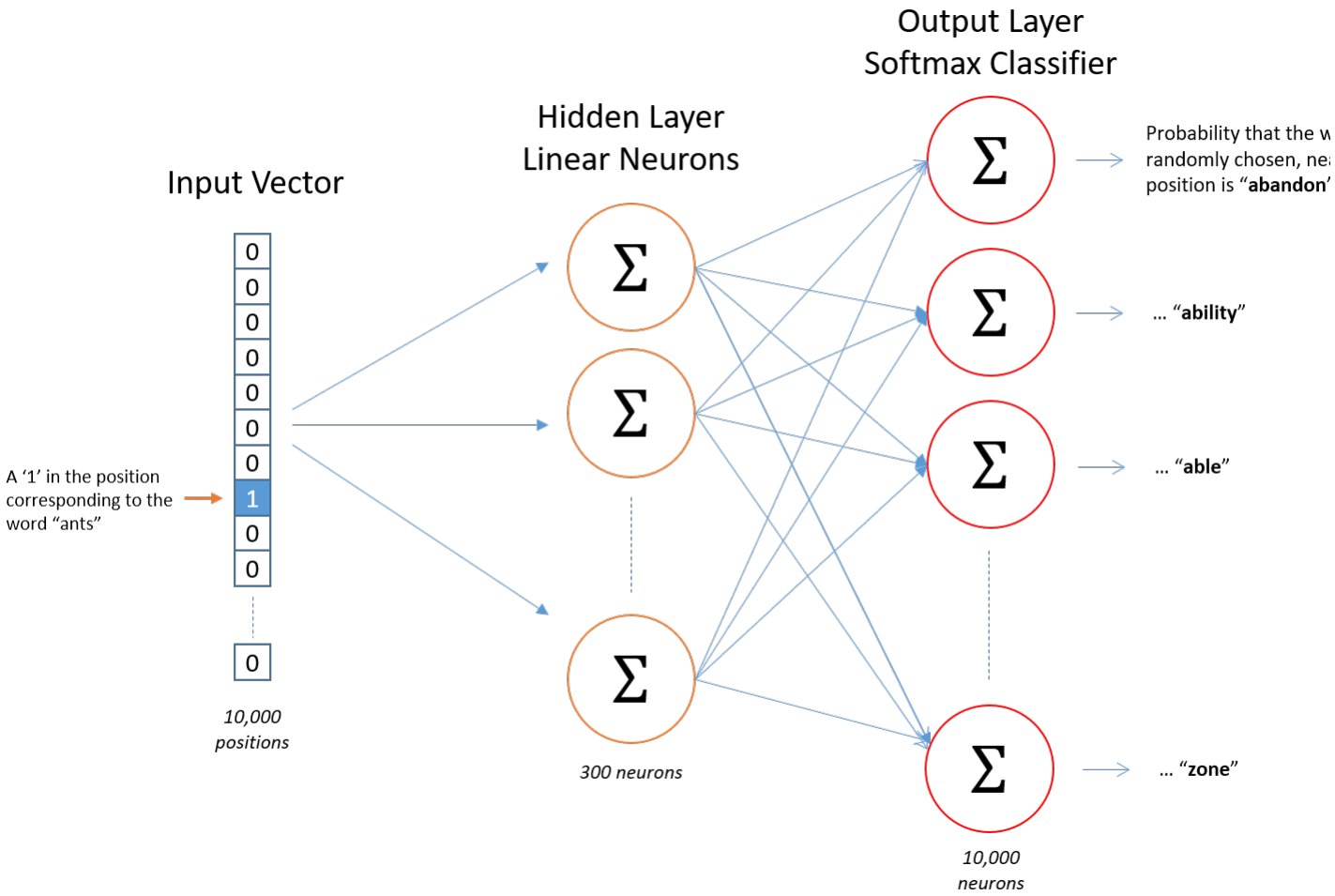
Architecture

The architecture is similar to an autoencoder's one, you take a large input vector, compress it down to a smaller dense vector and then instead of decompressing it back to the original input vector as you do with autoencoders, you output probabilities of target words.

First of all, we cannot feed a word as string into a neural network. Instead, we feed words as one-hot vectors, which is basically a vector of the same length as the vocabulary, filled with zeros except at the index that represents the word we want to represent, which is assigned "1".

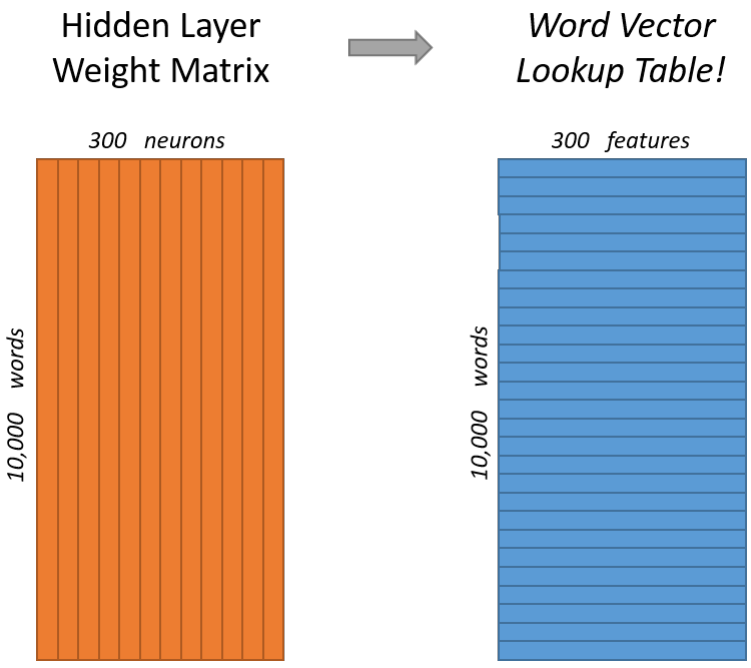
The hidden layer is a standard fully-connected (Dense) layer whose weights are the word embeddings.

The output layer outputs probabilities for the target words from the vocabulary.



> The input to this network is a one-hot vector representing the input word, and the label is also a one-hot vector representing the target word, however, the network's output is a probability distribution of target words, not *necessarily* a one-hot vector like the labels.

The rows of the hidden layer weight matrix, are actually the word vectors (word embeddings) we want!



The hidden layer operates as a lookup table. The output of the hidden layer is just the "word vector" for the input word. More concretely, if you multiply a `1 x 10,000 one-hot vector` by a `10,000 x 300 matrix` , it will effectively just select the matrix row corresponding to the '1'.

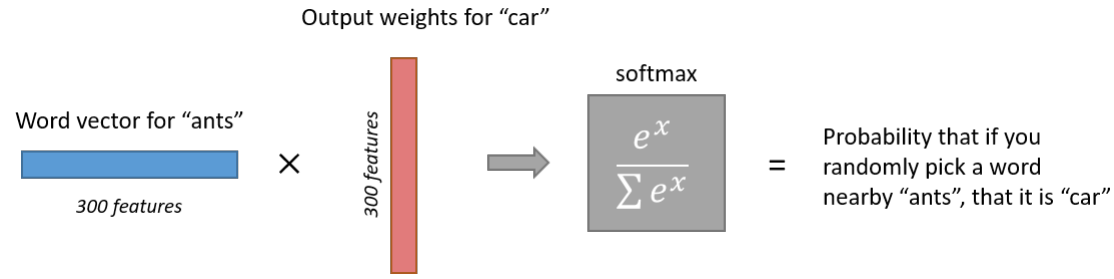
$$\begin{bmatrix}0 & 0 & 0 & 1 & 0\end{bmatrix} \times \begin{bmatrix}17 & 24 & 1\\ 23 & 5 & 7\\ 4 & 6 & 13\\ 10 & 12 & 19\\ 11 & 18 & 25\end{bmatrix} = \begin{bmatrix}10 & 12 & 19\end{bmatrix}$$

The end goal of all of this is to learn this hidden layer weight matrix and then toss the output layer when we're done!

The output layer is simply a softmax activation function:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Here is a high-level illustration of the architecture:



RNN

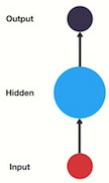
Repeat A to Z

Now try Z to A

Now start somewhere from middle

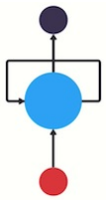
Human brain has the sequential memory nature by default

RNN’s have this abstract concept of sequential memory, but how the heck does an RNN replicate this concept? Well, let’s look at a traditional neural network also known as a feed-forward neural network. It has its input layer, hidden layer, and the output layer.



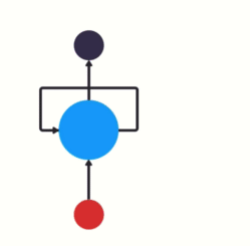
Feed Forward Neural Network

How do we get a feed-forward neural network to be able to use previous information to effect later ones? What if we add a loop in the neural network that can pass prior information forward?



Recurrent Neural Network

And that’s essentially what a recurrent neural network does. An RNN has a looping mechanism that acts as a highway to allow information to flow from one step to the next.



Passing Hidden State to next time step

This information is the hidden state, which is a representation of previous inputs. Let’s run through an RNN use case to have a better understanding of how this works.

Let’s say we want to build a chatbot. They’re pretty popular nowadays. Let’s say the chatbot can classify intentions from the users inputted text.

Classifying intents from users inputs

To tackle this problem. First, we are going to encode the sequence of text using an RNN. Then, we are going to feed the RNN output into a feed-forward neural network which will classify the intents.

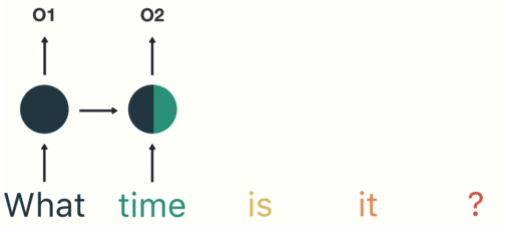
Ok, so a user types in… ***what time is it?***. To start, we break up the sentence into individual words. RNN’s work sequentially so we feed it one word at a time.

Breaking up a sentence into word sequences

The first step is to feed “What” into the RNN. The RNN encodes “What” and produces an output.

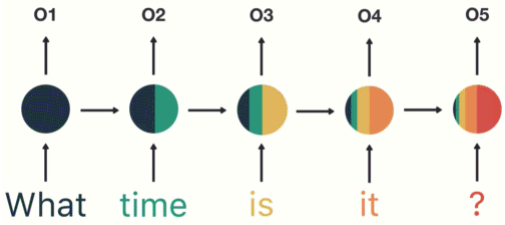
For the next step, we feed the word “time” and the hidden state from the previous step. The RNN now has information on both the word “What” and “time.”

We repeat this process, until the final step. You can see by the final step the RNN has encoded information from all the words in previous steps.



Since the final output was created from the rest of the sequence, we should be able to take the final output and pass it to the feed-forward layer to classify an intent.

![Image for post](https://miro.medium.com/freeze/max/60/1*3bKRTcqSbto3CXfwshVwmQ.gif?q=20



For those of you who like looking at code here is some python showcasing the control flow.

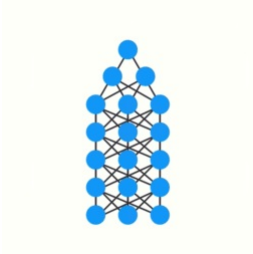
Vanishing Gradient

You may have noticed the odd distribution of colors in the hidden states. That is to illustrate an issue with RNN’s known as short-term memory.

Final hidden state of the RNN

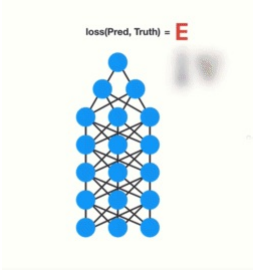
Short-term memory is caused by the infamous vanishing gradient problem, which is also prevalent in other neural network architectures. As the RNN processes more steps, it has troubles retaining information from previous steps. As you can see, the information from the word “what” and “time” is almost non-existent at the final time step. Short-Term memory and the vanishing gradient is due to the nature of back-propagation; an algorithm used to train and optimize neural networks. To understand why this is, let’s take a look at the effects of back propagation on a deep feed-forward neural network.

Training a neural network has three major steps. First, it does a forward pass and makes a prediction. Second, it compares the prediction to the ground truth using a loss function. The loss function outputs an error value which is an estimate of how poorly the network is performing. Last, it uses that error value to do back propagation which calculates the gradients for each node in the network.



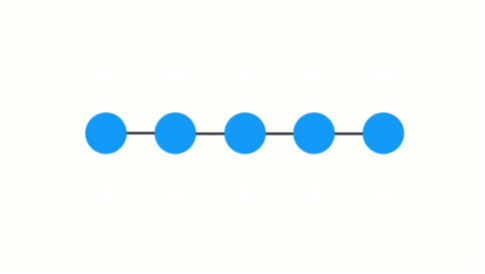
The gradient is the value used to adjust the networks internal weights, allowing the network to learn. The bigger the gradient, the bigger the adjustments and vice versa. Here is where the problem lies. When doing back propagation, each node in a layer calculates it’s gradient with respect to the effects of the gradients, in the layer before it. So if the adjustments to the layers before it is small, then adjustments to the current layer will be even smaller.

That causes gradients to exponentially shrink as it back propagates down. The earlier layers fail to do any learning as the internal weights are barely being adjusted due to extremely small gradients. And that’s the vanishing gradient problem.



Gradients shrink as it back-propagates down

Let's see how this applies to recurrent neural networks. You can think of each time step in a recurrent neural network as a layer. To train a recurrent neural network, you use an application of back-propagation called back-propagation through time. The gradient values will exponentially shrink as it propagates through each time step.

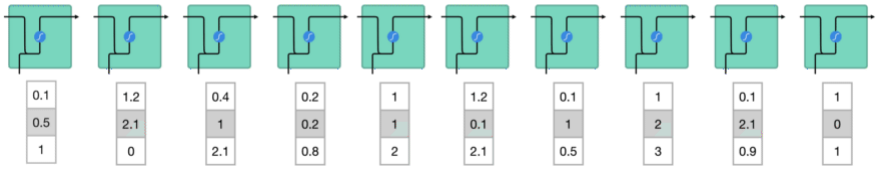


Gradients shrink as it back-propagates through time

Again, the gradient is used to make adjustments in the neural networks weights thus allowing it to learn. Small gradients mean small adjustments. That causes the early layers not to learn.

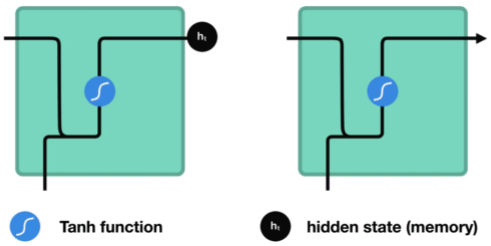
Because of vanishing gradients, the RNN doesn't learn the long-range dependencies across time steps. That means that there is a possibility that the word "what" and "time" are not considered when trying to predict the user's intention. The network then has to make the best guess with "is it?". That's pretty ambiguous and would be difficult even for a human. So not being able to learn on earlier time steps causes the network to have a short-term memory.

An RNN works like this; First words get transformed into machine-readable vectors. Then the RNN processes the sequence of vectors one by one.



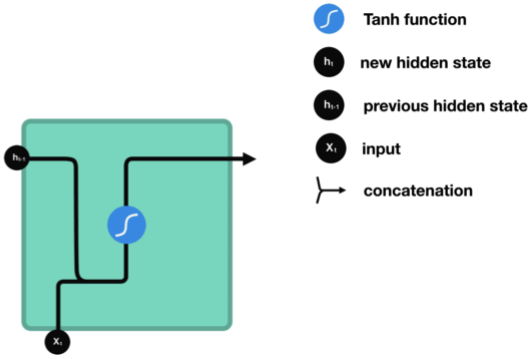
Processing sequence one by one

While processing, it passes the previous hidden state to the next step of the sequence. The hidden state acts as the neural networks memory. It holds information on previous data the network has seen before.



Passing hidden state to next time step

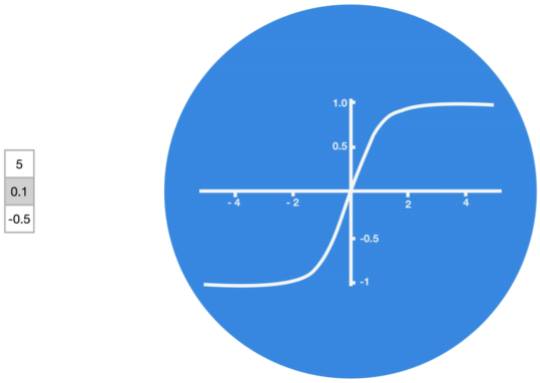
Let's look at a cell of the RNN to see how you would calculate the hidden state. First, the input and previous hidden state are combined to form a vector. That vector now has information on the current input and previous inputs. The vector goes through the tanh activation, and the output is the new hidden state, or the memory of the network.



RNN Cell

Tanh activation

The tanh activation is used to help regulate the values flowing through the network. The tanh function squishes values to always be between -1 and 1.



Tanh squishes values to be between -1 and 1

When vectors are flowing through a neural network, it undergoes many transformations due to various math operations. So imagine a value that continues to be multiplied by let's say 3. You can see how some values can explode and become astronomical, causing other values to seem insignificant.

vector transformations without tanh

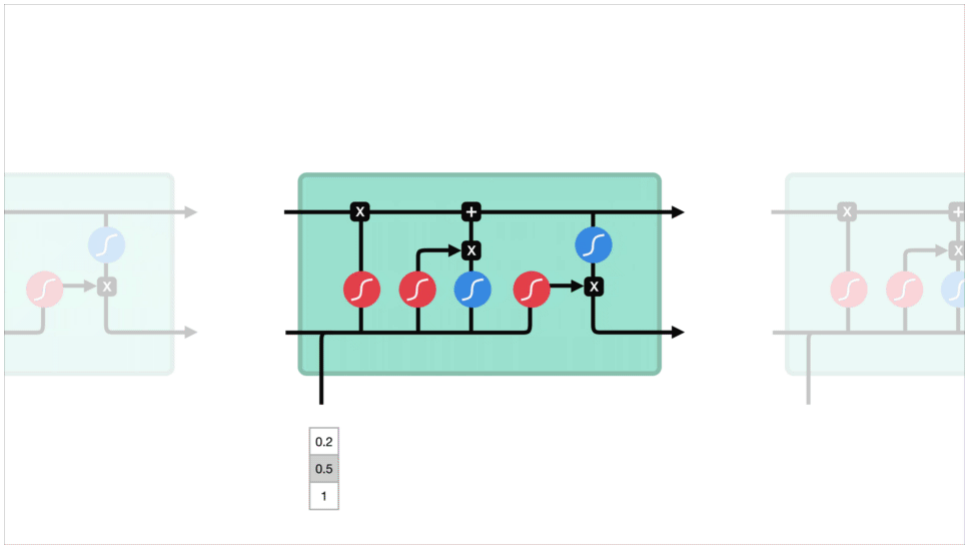
A tanh function ensures that the values stay between -1 and 1, thus regulating the output of the neural network. You can see how the same values from above remain between the boundaries allowed by the tanh function.

vector transformations with tanh

So that's an RNN. It has very few operations internally but works pretty well given the right circumstances (like short sequences). RNN's uses a lot less computational resources than it's evolved variants, LSTM's and GRU's.

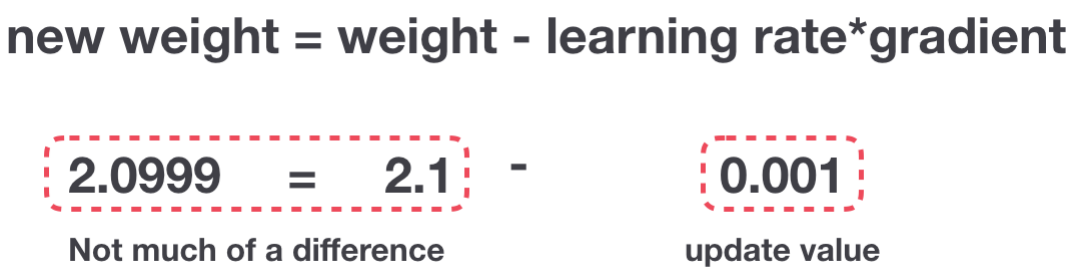
LSTM

LSTM to the rescue



Recurrent Neural Networks suffer from short-term memory. If a sequence is long enough, they'll have a hard time carrying information from earlier time steps to later ones. So if you are trying to process a paragraph of text to do predictions, RNN's may leave out important information from the beginning.

During back propagation, recurrent neural networks suffer from the vanishing gradient problem. Gradients are values used to update a neural networks weights. The vanishing gradient problem is when the gradient shrinks as it back propagates through time. If a gradient value becomes extremely small, it doesn't contribute too much learning.

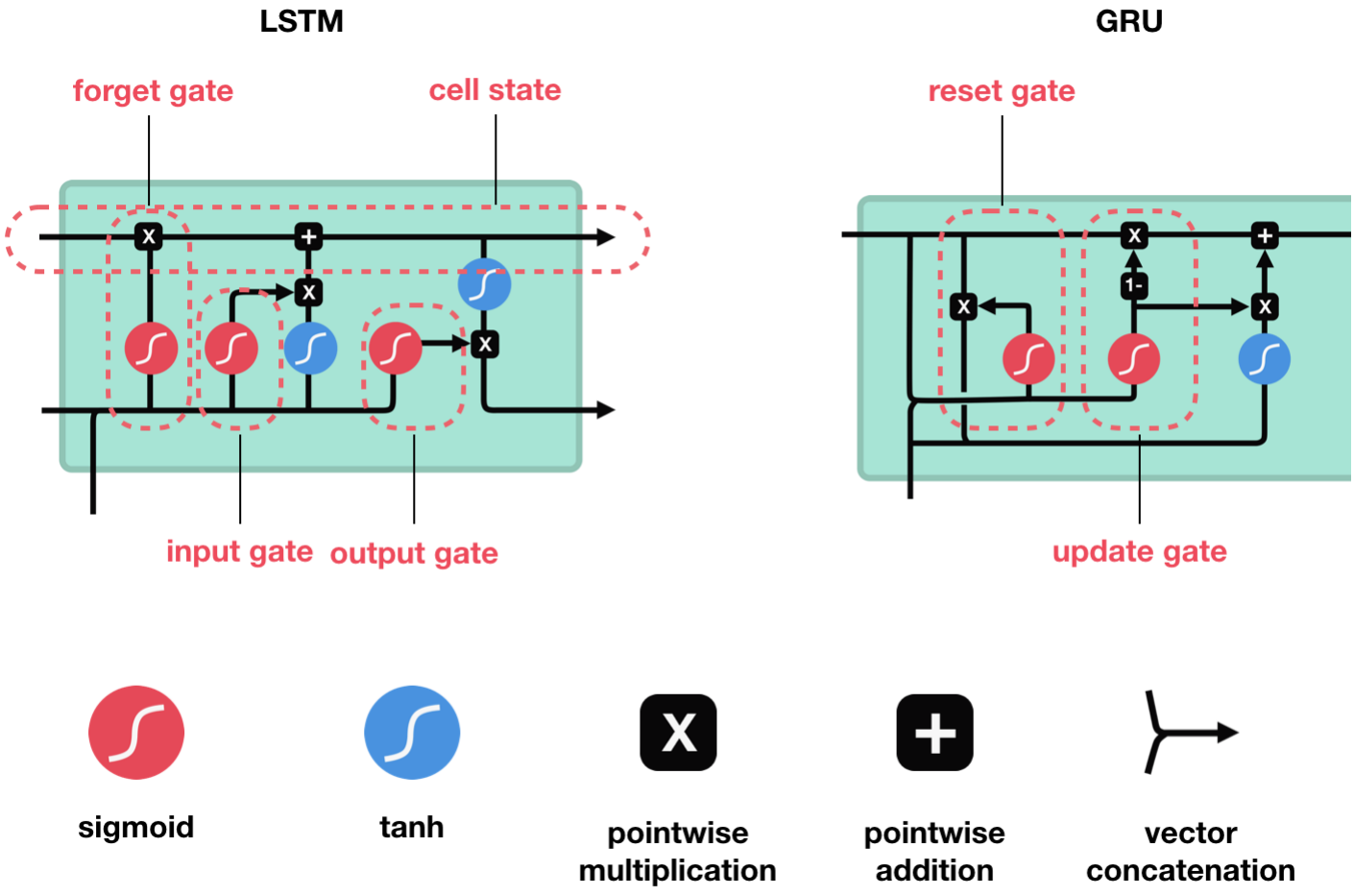


Gradient Update Rule

So in recurrent neural networks, layers that get a small gradient update stops learning. Those are usually the earlier layers. So because these layers don't learn, RNN's can forget what it seen in longer sequences, thus having a short-term memory. If you want to know more about the mechanics of recurrent neural networks in general, you can read my previous post here.

LSTM's and GRU's as a solution

LSTM 's and GRU's were created as the solution to short-term memory. They have internal mechanisms called gates that can regulate the flow of information.

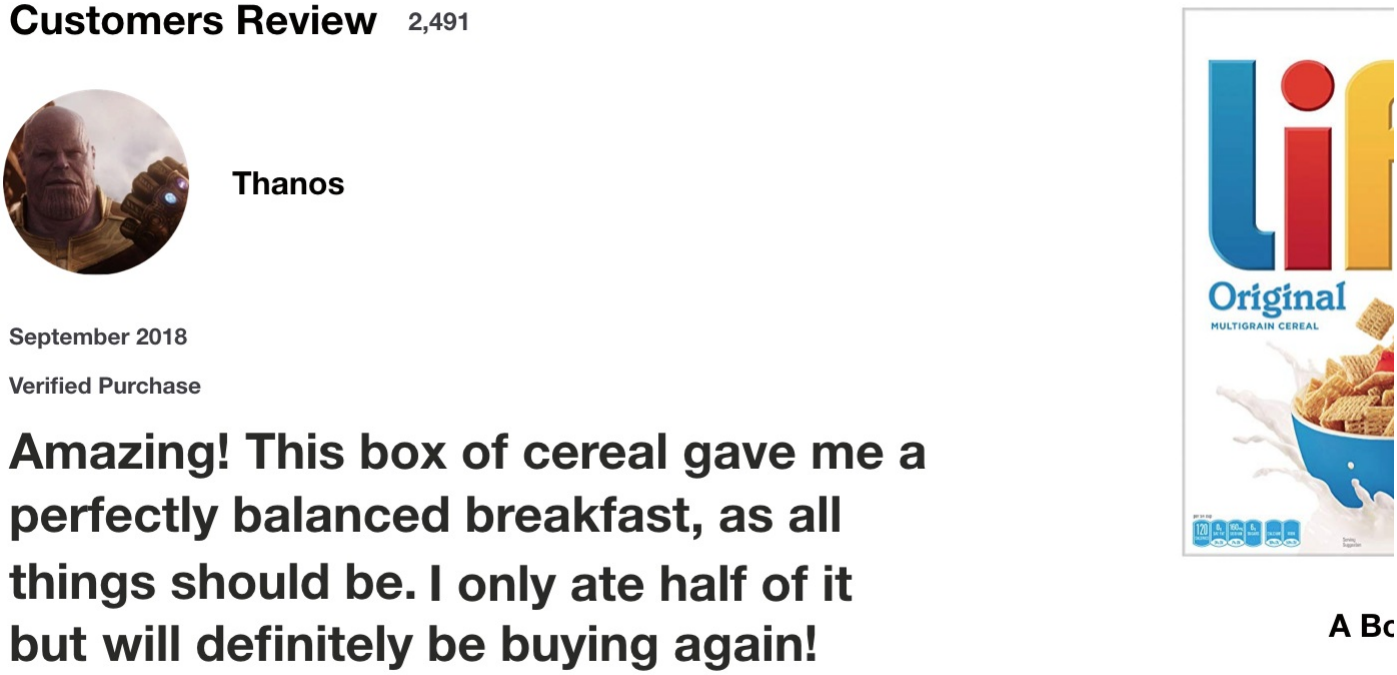


These gates can learn which data in a sequence is important to keep or throw away. By doing that, it can pass relevant information down the long chain of sequences to make predictions. Almost all state of the art results based on recurrent neural networks are achieved with these two networks. LSTM's and GRU's can be found in speech recognition, speech synthesis, and text generation. You can even use them to generate captions for videos.

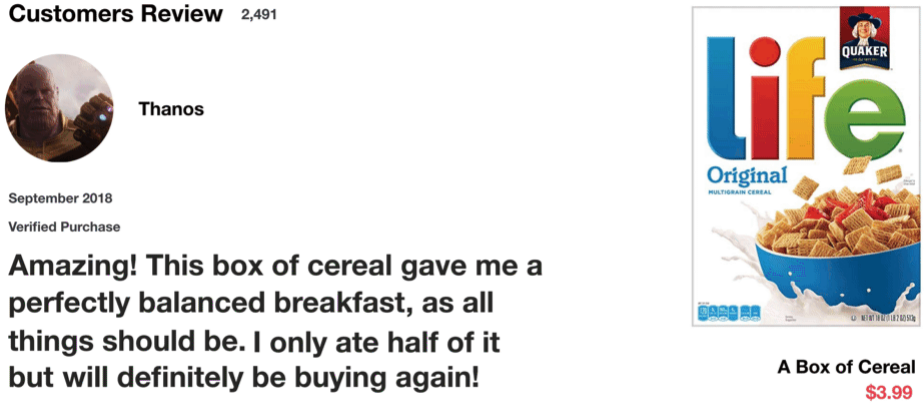
Ok, so by the end of this post you should have a solid understanding of why LSTM's and GRU's are good at processing long sequences. I am going to approach this with intuitive explanations and illustrations and avoid as much math as possible.

Intuition

Ok, Let's start with a thought experiment. Let's say you're looking at reviews online to determine if you want to buy Life cereal (don't ask me why). You'll first read the review then determine if someone thought it was good or if it was bad.

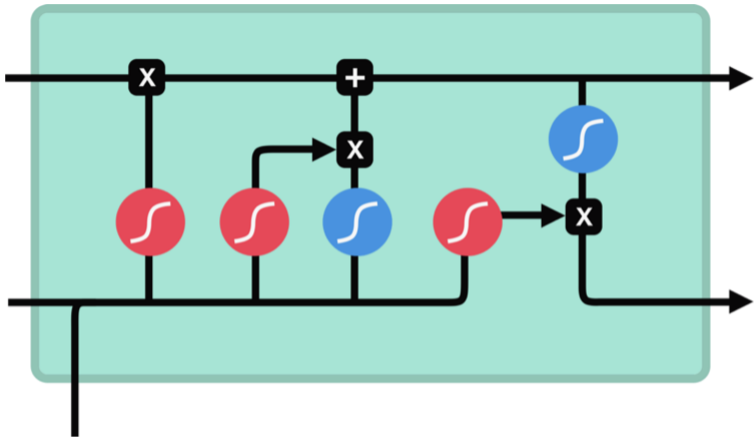


When you read the review, your brain subconsciously only remembers important keywords. You pick up words like "amazing" and "perfectly balanced breakfast". You don't care much for words like "this", "gave", "all", "should", etc. If a friend asks you the next day what the review said, you probably wouldn't remember it word for word. You might remember the main points though like "will definitely be buying again". If you're a lot like me, the other words will fade away from memory.



And that is essentially what an LSTM or GRU does. It can learn to keep only relevant information to make predictions, and forget non relevant data. In this case, the words you remembered made you judge that it was good.

An LSTM has a similar control flow as a recurrent neural network. It processes data passing on information as it propagates forward. The differences are the operations within the LSTM's cells.



LSTM Cell and It's Operations

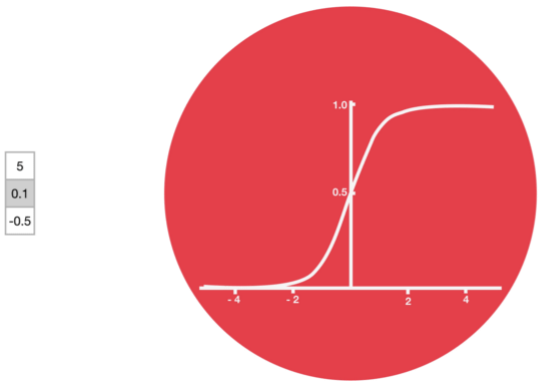
These operations are used to allow the LSTM to keep or forget information. Now looking at these operations can get a little overwhelming so we'll go over this step by step.

Core Concept

The core concept of LSTM's are the cell state, and it's various gates. The cell state act as a transport highway that transfers relative information all the way down the sequence chain. You can think of it as the "memory" of the network. The cell state, in theory, can carry relevant information throughout the processing of the sequence. So even information from the earlier time steps can make it's way to later time steps, reducing the effects of short-term memory. As the cell state goes on its journey, information get's added or removed to the cell state via gates. The gates are different neural networks that decide which information is allowed on the cell state. The gates can learn what information is relevant to keep or forget during training.

Sigmoid

Gates contains sigmoid activations. A sigmoid activation is similar to the tanh activation. Instead of squishing values between -1 and 1, it squishes values between 0 and 1. That is helpful to update or forget data because any number getting multiplied by 0 is 0, causing values to disappears or be "forgotten." Any number multiplied by 1 is the same value therefore that value stay's the same or is "kept." The network can learn which data is not important therefore can be forgotten or which data is important to keep.

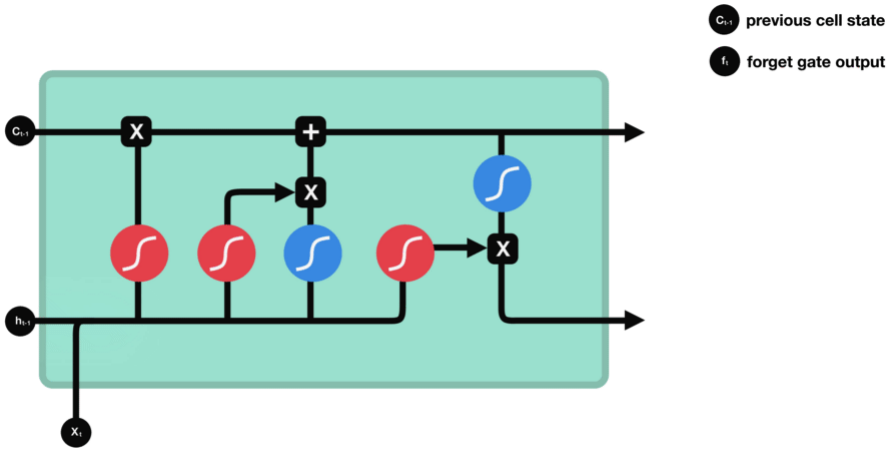


Sigmoid squishes values to be between 0 and 1

Let's dig a little deeper into what the various gates are doing, shall we? So we have three different gates that regulate information flow in an LSTM cell. A forget gate, input gate, and output gate.

Forget gate

First, we have the forget gate. This gate decides what information should be thrown away or kept. Information from the previous hidden state and information from the current input is passed through the sigmoid function. Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.



Forget gate operations

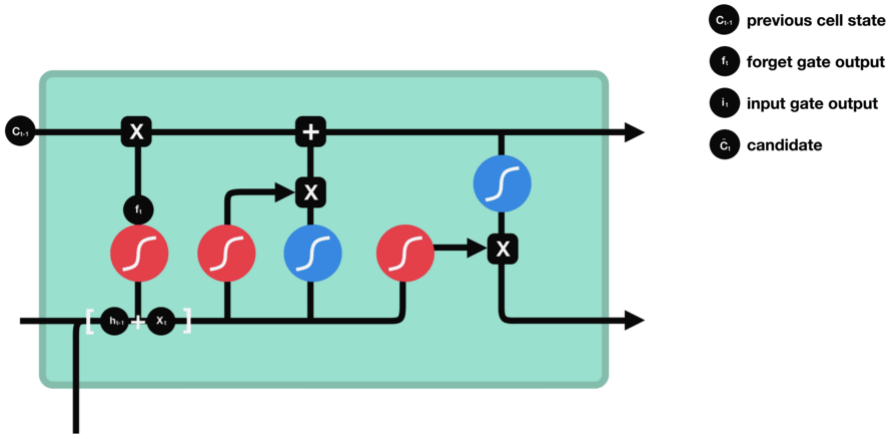
Input Gate

To update the cell state, we have the input gate. First, we pass the previous hidden state and current input into a sigmoid function. That decides which values will be updated by transforming the values to be between 0 and 1. 0 means not important, and 1 means important. You also pass the hidden state and current input into the tanh function to squish values between -1 and 1 to help regulate the network. Then you multiply the tanh output with the sigmoid output. The sigmoid output will decide which information is important to keep from the tanh output.

sigmoid(xt+ht) = it

tanh(xh+ht) = C~t

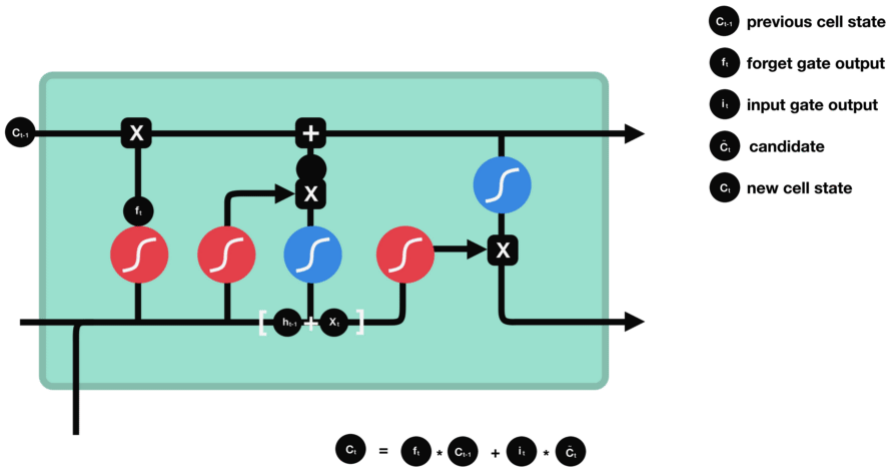
Ct-1xft + itxC~t



Input gate operations

Cell State

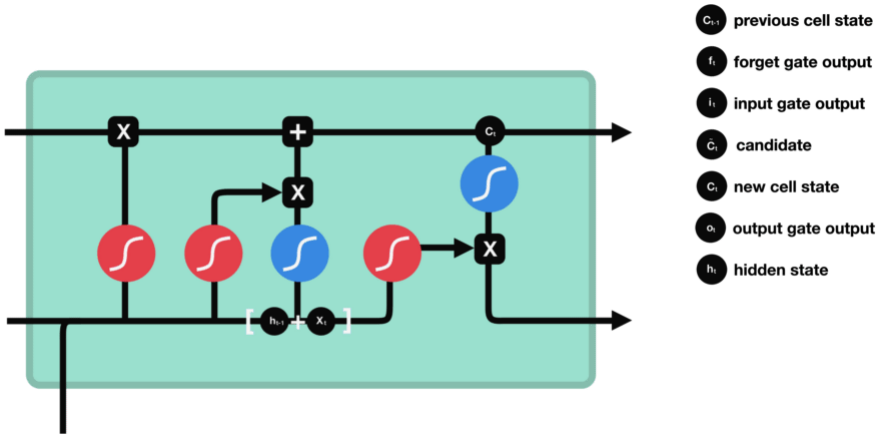
Now we should have enough information to calculate the cell state. First, the cell state gets pointwise multiplied by the forget vector. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0. Then we take the output from the input gate and do a pointwise addition which updates the cell state to new values that the neural network finds relevant. That gives us our new cell state.



Calculating cell state

Output Gate

Last we have the output gate. The output gate decides what the next hidden state should be. Remember that the hidden state contains information on previous inputs. The hidden state is also used for predictions. First, we pass the previous hidden state and the current input into a sigmoid function. Then we pass the newly modified cell state to the tanh function. We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry. The output is the hidden state. The new cell state and the new hidden is then carried over to the next time step.



output gate operations

To review, the Forget gate decides what is relevant to keep from prior steps. The input gate decides what information is relevant to add from the current step. The output gate determines what the next hidden state should be.