Patrick Gruber
pgruber2

Latent space encoding using LSTMs: Finding similar word context

Recurrent Neural Networks (RNNs) have been the state of art Natural Language Processing (NLP) elements of Deep Learning (DL) for a long time. One specific example of RNNs are the Long Short Term Memory Networks (often called LSTMs). They were first introduced by Hochreiter and Schmidhuber in 1997 [1]. By making the networks "recurrent", each node in the network can persist information as it processes data. As such, they can pass signals to other nodes that inform them of what they processed themselves, which adds some context for other nodes down the line. This is a common problem in NLP, as words that come later in a sentence, often derive their exact meaning from earlier words. On their output, LSTMs can learn a numerical representation of the input data, which in turn can be analyzed further. LSTMs are a great tool for NLP, as they are able to encode language semantics and maintain an internal state of what has already been processed.

Processing the meaning behind a sentence is often not as easy as it sounds [2]. A single word often has multiple meanings and can therefore be interpreted in different ways. The word "date" for example, changes depending on the context in which it is used. "They went on a dinner date" is different from "They picked a wedding date" or "He really likes eating dates". This means that for sentiment analysis, we should really consider the entire context in which a word is used. Words such as "good" or "bad" are oftentimes great indicators about the sentiment of a sentence, but what happens if we review a restaurant as "A good place to get ripped off"? The usage of "good" should suggest positive sentiment, but it is followed with the words "ripped off", which should make us reconsider. In fact, the use of "good" seems to underline the negative sentiment of "ripped off". This is where LSTMs come in and allow us to parse the sentence as a whole (and often from both directions), and analyze the word context more accurately.

As previously mentioned, RNN's main feature is that they can pass information along to other nodes. This differs from normal feed forward connected DL structures (where data is also passed along to other nodes at the next level) in the way that the data also flows horizontally, along each individual time step.

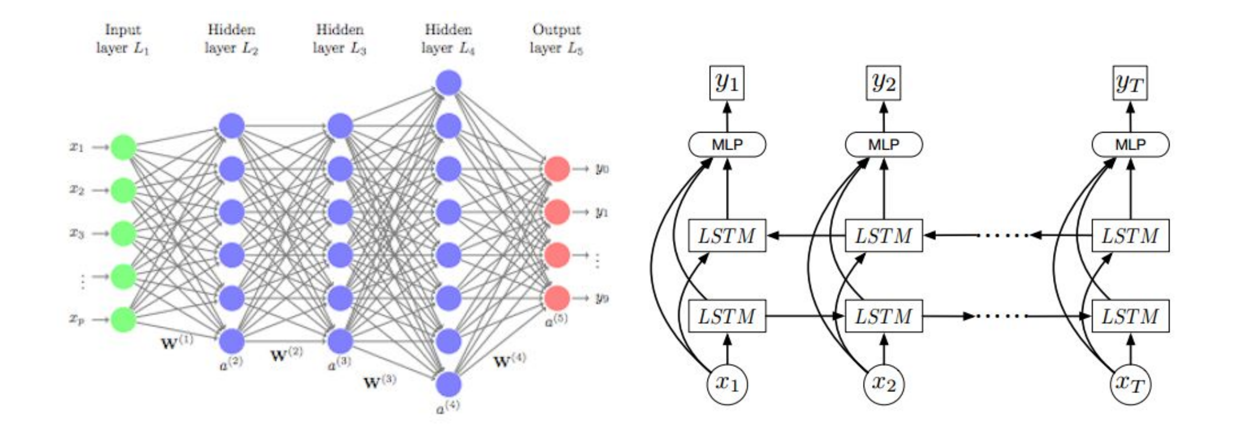
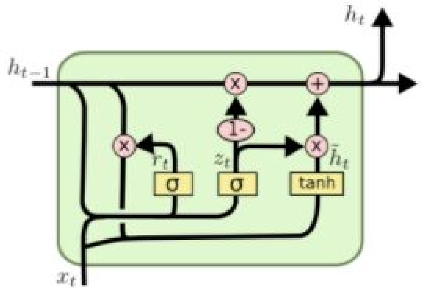


Figure 1: Feed forward vs RNN structure example (https://uc-r.github.io/feedforward_DNN and https://datascience.stackexchange.com/questions/40171/stacking-lstm-layers)

In traditional DL, we pass data into the depth of the layered network, where each layer extracts additional information from our input data. There is no link from the node processing x_1 to the node processing x_2. A deeper network structure can still be achieved with LSTMs, but within each layer, each node also passes information to the next in line (for example from x_1 to the node that processes x_2). This allows each node to learn more about what is going on in the current context.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 2: The inner workings of each LSTM node. (https://colah.github.io/posts/2015-08-Understanding-LSTMs/)

An in-depth look at a single LSTM cell (shown in Figure 2), shows the data flow from h_{t-1} to h_t, or simply put, from the earlier cell to the next in line. The cell includes data provided at timestep x_t to make these decisions. Multiple things are happening here (we won't go too much into detail, as other sources on the web explain the detailed process already); The cell decides what it needs to forget (represented as r_t), based on the data it got and the state it is in, then it needs to decide what it needs to pass along (a combination of z_t and $\bar{h}_t$, where the former decides what values to update, while the latter decides what new values to assign). We output h_t and also pass it along to the next LSTM cell [3]. The same concept can be used in both directions (bi-directional) to analyze word context from both ends of the sentence and make sure that the LSTM node has enough context to process each word.

In our case we want to use LSTMs to create a latent space, where related search phrases are closer together in a multi-dimensional space. For this, we need to modify the neural network structure slightly. We will use an Autoencoder concept, where half of the network is used to encode data, while the other half will decode the data [4]. This approach is often called unsupervised learning, as the data does not need to be labelled. Additionally, we will make use of the benefits that LSTMs provide, in the hope that the latent space yields queries that are syntactically similar to user input [5]. For this example, we use the freely available IMDB dataset, which contains sentiment information for IMDB user movie reviews. The encoder structure will be led by an embedding layer, a bi-directional LSTM layer, followed by a batch normalization layer with a fully connected output layer. This will result in an output vector, which maps our data into a multidimensional latent space. For the decoder, we will take in the encoded vector, and provide it as an input to a set of fully connected layers, which then map the data to each of our LSTM nodes. The output of the decoder will again pass through fully connected layers, followed by a softmax layer. Figure 3 displays the architecture in detail. The resulting concept is fairly simple, both encoder and decoder have to work together to minimize the loss at training time. The encoder will try to find a representation in the latent space that is easy for the decoder to understand. The decoder constantly has to learn what the different representations in the latent space mean.

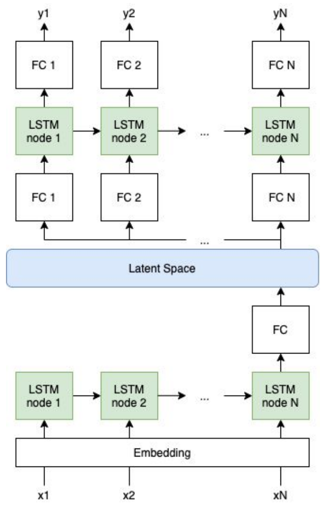


Figure 3: Simplified network structure without sentiment information.

After training the model for 50 epochs, we end up with a categorical cross entropy loss of about 1.8. We can also plot the data in relation to their sentiment in a three dimensional space. The model had no concept of sentiment, but it does seem as though it ended up naturally clustering phrases of similar sentiment in the resulting latent space. A plot of this can be seen in Figure 4, where positive sentiment phrases are displayed as blue, while negative ones are shown in red. Many of the phrases in the training data are clustered in the center of the space, however. This could potentially mean that the vast majority of the phrases are very similar. For the search phrase “great family movie”, the model returned similar phrases, such as “decent family movie”, “long long time”, “great dumb movie”, “an excellent movie”, “worth your time”, and “fun family movie”. This showcases that the LSTM did learn similar representations, and grouped them up in the latent space.

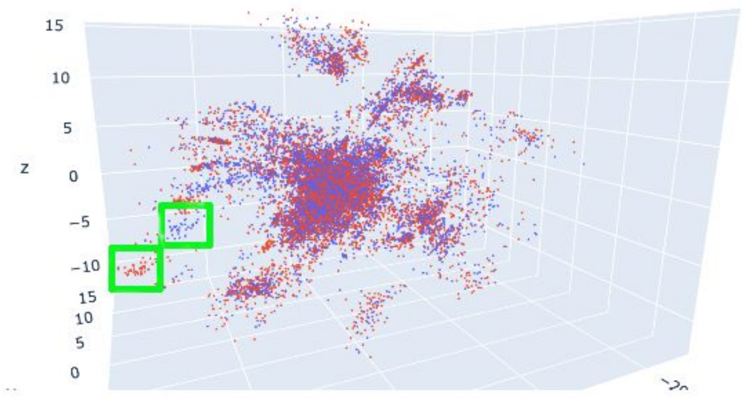


Figure 4: Three dimensional latent space of the LSTM encoder.

It is worth noting that making the LSTM layer bi-directional improved the results a bit. While the query "best movie ever", initially also returned "best stay away", bi-directional models returned phrases like "kids movie ever", "everyone must see", but unfortunately also "worst movie ever". Based on the way the model was trained, it is to be expected that sentiment is not a factor when computing similar phrases, the phrase structure, however, is very close to the ones that are returned. More details can be found in the supplemental material ("Latent space encoding using LSTMs - Finding similar word context (no sentiment).html" and "plot_no_sentiment.html").

A recommender system should have a concept of sentiment though. On top of that we already have some pretty reliable labelled data that we can use for this. To enable our model to acknowledge the sentiment of phrases, we need to modify our initial model slightly. Instead of having only one softmax output, we now use the last LSTM cell's output, and route it through a fully connected layer with a sigmoid activation. Then we add a binary cross entropy loss to the training. The idea here is to force the decoder to learn both syntactical and sentiment structures in the phrases to minimize the total loss. The encoder will react to the change by adjusting the latent space to reflect both criteria, and simplify decoding. Figure 5 shows the updated decoder of the new model.

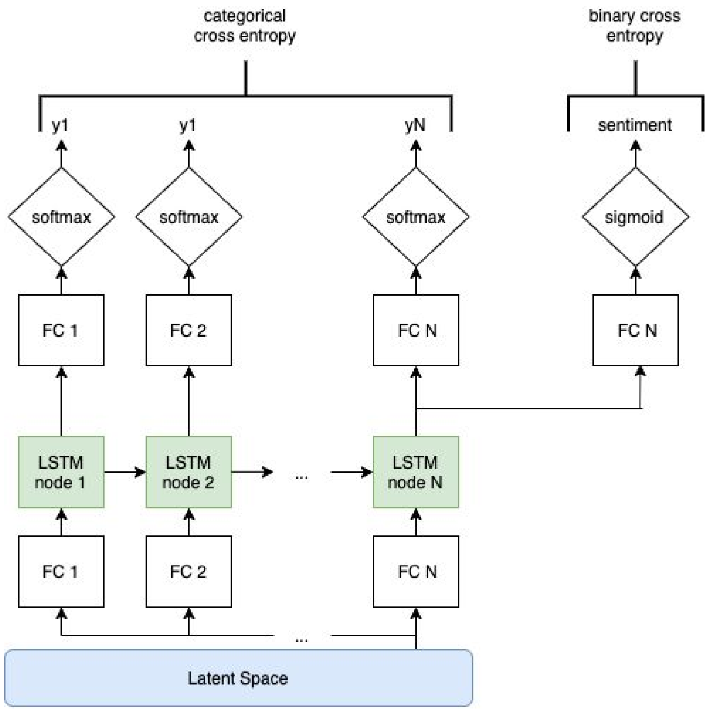


Figure 5: Update model with sentiment output.

If we consider the same queries as before, for "great family movie" we now get recommended phrases such as "great idea here", "good family films", "those fantastic films", "any comedy movie", and "its release though". While for the query "best movie ever", the model yields "modern cinema masterpiece", "family movie night", "top class cinema", "best films ever", and "personal level enjoy".

Every once in a while there are still some results whose sentiment is a bit off, but overall the results have drastically improved. Details can be found in the supplemental materials ("Latent space encoding using LSTMs - Finding similar word context.html" and "plot_sentiment.html"). As shown in Figure 6, the sentiment boundaries are now much clearer (blue = positive, red = negative).

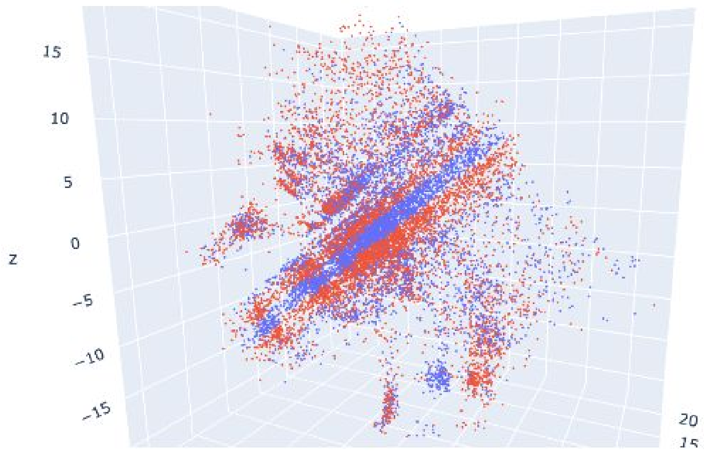


Figure 6: Latent space for sentiment LSTM encoding.

There are some limitations to this approach. In general, we don't always have the ability to add a heuristic for the model to help it steer into a direction we want. For the IMDB dataset, we were lucky that the data contained sentiment labels, which allowed us to improve the recommendations. Still, the results in the first approach were quite good, as the stated examples show. Furthermore, to keep the example simple, we only used three word phrases. Even with the added sentiment, it isn't always easy to detect whether a phrase is positive or negative. We also restricted the latent space to three dimensions. In reality when we are computing the distances, we would be able to introduce as many dimensions as we like. This would allow the encoder to create more complex clusters, and the results might be improved. On the other hand, we would not be able to plot the latent space directly from the encoder and make it available for exploration.

Long Short Term Memory networks are great tools for processing natural language. Oftentimes they do have the limitation of needing labelled data (supervised training), but they can also be used in an unsupervised manner. As we have seen in the first approach, they have the ability to extract some form of semantics from provided examples. The semantics might not always be helpful for our use case though and we still need to guide them into the right direction. Recently, there have also been a multitude of other network structures released that appear to improve on some of LSTMs weaknesses. With the introduction of deep learning, the future of NLP appears to be quite promising.

Sources:

[1] https://www.mitpressjournals.org/doi/abs/10.1162/neco.1997.9.8.1735
[2] https://www.coursera.org/learn/cs-410/lecture/rLpwp/lesson-1-1-natural-language-content-analysis
[3] https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[4] https://www.jeremyjordan.me/autoencoders/
[5] https://machinelearningmastery.com/lstm-autoencoders/