Human Languages and Machines

Recurrent Neural Networks and Word Vectors

Gabe Wilberscheid

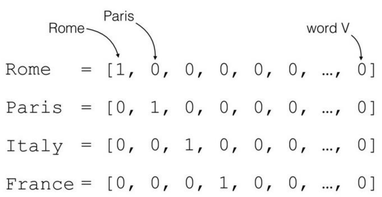
Department, Minnesota State University Moorhead, 1104 7th Avenue South, Moorhead, MN 56563

Abstract:

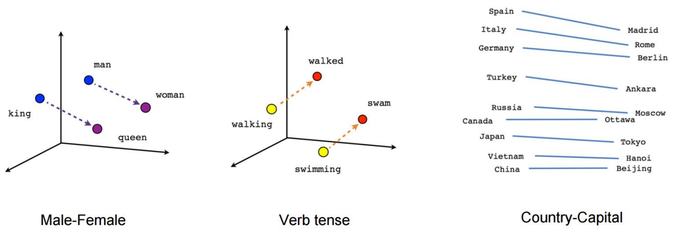
Recently groundbreaking work in the area of deep neural networks combined with huge datasets and paired with unparalleled compute thanks to new hardware like tensor processing units (TPU's) has allowed computers to understand human language like never before. Google, Facebook, and Microsoft have all had multiple research papers released, each progressing the abilities of researchers to analyze huge datasets of language using statistical machines. The winning combination has been to use high dimensional word vectors produced by neural networks and then feed the words vectors into a generator/discriminator pair that has what researchers call "attention".

Word Emeddings:

Traditional attempts at understanding human language failed largely because researchers did not know of a decent way to store lexemes (words) of a language. Early approaches used one-hot encoding and word stemming and lemmatization. In such models, the machine is given a vector for each base word (running, ran, and runs --> run) where the word vector is a sparse vector with a single value of one, and the rest are zeros.



While we can build some interesting models with this method, it is severely limited in that words have no relationship among other words. All the machine knows is yes or no a word appeared. Yet this is not at all how humans learn words, we know certain words have special meanings, or that certain words are closely related and can be used interchangeably with little to no impact on the meaning of the sentence. To overcome this drawback of one-hot encoding, we can find inspiration in both the fields of computer science and linguistics. In computer science it is common to store information in dense vectors, these are sometimes called Vector Space Models (VSM). Combining this model with the distributional hypothesis, that says words that appear in similar context over large samples have similar semantic meanings. We can then build a model that can store words in high dimensional, dense vector spaces where words relationship to one another holds meaning.



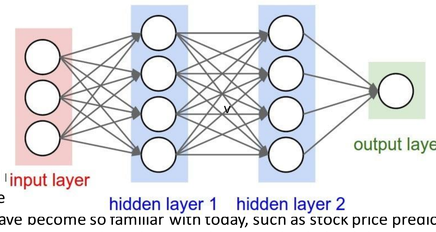
Male-Female Verb tense Country-Capital

These word vectors must be learned through analysis of huge text corpus. These word vectors are built by using neural networks that learn which words have similar meanings. Thankfully tech giants like Google and Facebook have done this using their supper computers and have released the results to the public. Google's word vectors were trained on 100 billion words from Google news. We can download them and use them freely in our projects. There are also some useful Python packages out there that make preprocessing text and using word vectors much easier. I used Spacy, as it is open source and built for industry-level word processing. Not to mention that the documentation is very helpful and there is an active community working to make it even better.

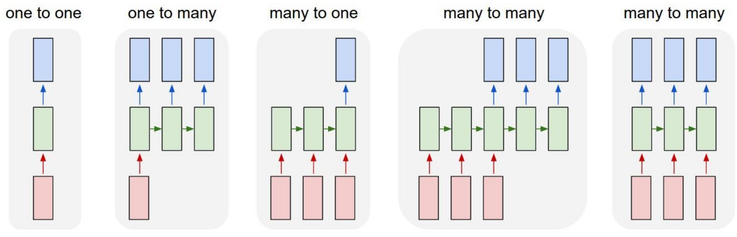
Recurrent Neural Networks:

Artificial neural networks (ANN) are nothing new; the original paper describing them was published back in 1958 by a psychologist as a way for computers to learn to recognize visual objects. The inspiration for artificial neural networks comes from the way we think the human brain works, with the simplest architecture being a perceptron, which is modeled after humans neurons. While ANN's showed some promise for a few decades after their invention, many researchers felt they had hit their limit. This set off what is known as the neural network winter, where nearly no attention was given to improving these algorithms and researches focused on other models such as generative algorithms and evolutionary algorithms.

Finally, several researchers, namely, Yoshua Bengio, Geoffrey Hinton, and Yann LeCun, made progress in the late 80's and early 90's that would forever change how ANN's are used. Thanks to more powerful computers and larger datasets, these researchers were able to come up with new features of ANN's that made them quickly reach state of the art in many tasks. The two most important types of ANN that were created during this time were convolutional NN's which are mainly used for image recognition tasks and recurrent NN's which are used for sequential data.



For my project, I used I NN's that can process sequential data, name odels that are used in all sorts of tasks we have become so familiar with today, such as stock price predictions, language translations, article summarizes and much more. The main idea behind these models is trying to learn a pattern through time, or the change of states.



I found several useful libraries that helped with parsing the text into a clean dataset that could then be input into a model. I examined word vectors and played around with some vector operations and used them to find synonyms to words. To find synonyms for words, Spacy has implemented a feature that looks for the most similar vector of a word. Spacy also has some useful features that can label each word in a sentence with their parts of speech.

Text Generation:

After I had played around with the preprocessing and working with word vectors I decided to build a recurrent neural network that was capable of generating text. To do this we first have to train a model on some piece of text. During this process, we feed some of the text into the model and have it predict what the next word or character will be. Then we tell the model if it was correct or not and we slide our input text one more step forward. We do this millions of times until the model starts to learn what the next word or character will be. There are three many ways we could go about this. First, we could have the model guess what the next character will be. When doing this we will often first change all letters to lowercase so the model has fewer characters to learn. It is also common to remove punctuation, as this can also stump the model and create unneeded complexity. This is by far the simplest method to train a model on how to output language but it has some pros as well. For one, the model starts off knowing no words and just guessing the next letter. After a time it will begin to learn common words, although the sentence itself may not make much sense. The pro to this model is that if there are some misspelled words in our input data, the model can handle it and so long as there are not a ton of misspelled words it should not mess the model up too much.

The second option would be to create a word-level model. This would be done by looking at all the words in the input data, giving each one a number and then having the model guess the next word in the sentence. While this model would not need to learn how to spell words and would do a better job at creating viable sentences, it is not without its own draw backs. For one, the model still starts off knowing nothing about the words it uses. It does not know that dogs and cats are more similar than dogs and apples. This type of model also has a serious flaw, that is according to the model the only words that exist are those that we told it about in the input data. If we wanted to train this model on a different text data and our new data had a word that was not in the previous model or a word that was misspelled, our model would crash.

The last option is the most complex but will lead to the best results. Using word vectors we give the model some understanding of words and their meaning right from the start. This makes it possible for the model to learn patterns that are similar in meaning, but completely different in spelling. Take for example the following two sentences. "I took my dog on a walk this morning." and "I jogged my dog at dawn." The previous two models would not have a clue that these sentences are similar because the words used are different. A word vector model would know that these sentences are in someway similar.

Due to time and hardware restrains I settled on building a character model with only lowercase letters and spaces, 27 total character the model had to learn. These have to be stored in memory and creating a word-level model with thousands of words will cause most machines to crash due to memory errors. I trained it on Alice in Wonderland for a few hours and here are some of the results.

Input text: "an angry voice the rabbit s pat pat whe"

Model output:"an angry voice the rabbit s pat pat where what they worst alice stilld poise beloud beight but thing was a causi there s tring alice who said the finish was know decl but the jury was at least when they had to know the middle and when she"

We can see the model has begun to spell words correctly and occasionally string a few words together that make sense, but it is a long way from writing like Lewis Carroll. In the future, I would like to train a word vector model on a larger dataset and see how the output would improve. It is interesting to see that we have come to a point where to build models that are comparable to human-level language. We first have a machine encode words into a 300-dimensional vector space, then we use these word vectors to train a different model to understand patterns of speech. It seems the best results come when humans try not to teach a model language, but rather let the models learn how we use language by observing us.

References:

"Example Script to Generate Text from Nietzsche's Writings." LSTM for Text Generation - Keras Documentation, keras.io/examples/lstm_text_generation/.

Gelly, Sylvain et al. "MemGEN: Memory is All You Need." ArXiv abs/1803.11203 (2018): n. Pag.

Géron Aurélien. Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media, Inc., 2019.

"SpaCy · Industrial-Strength Natural Language Processing in Python." · Industrial-Strength Natural Language Processing in Python, spacy.io/.

Stecanella, Rodrigo, and Bruno Stecanella. "The Beginner's Guide to Text Vectorization." MonkeyLearn Blog, 19 Nov. 2018, monkeylearn.com/blog/beginners-guide-text-vectorization/.