Brian Rafferty, Kaiyu Chen, Richard Estes

CS 582

Professor Konopka

May 17, 2019

Artificial Led Zeppelin

We were assigned a semester long project in the beginning of the year where we were given free rein to choose the topic, while making sure that it fell into the category of speech recognition or speech processing. After reviewing previously accomplished projects, we were enthralled by the idea that a program could learn from a dataset to successfully generate original text, and it was in that moment when we decided to synthesize song lyrics from the legendary rock band Led Zeppelin. All we knew when we got started was that we were required to find a database that contained all known text material (in our case, this was a website that hosted song lyrics), analyze the data, train a machine learning model to recognize key phrases, and attempt to generate something coherent.

As excited as we were to start, the initial challenge of breaking our problem up into solvable solutions proved to be daunting. Many of the early weeks were spent researching topics like deep learning algorithms, recurrent neural networks, and synthetic generation. Ultimately, we decided that neural networks would serve as an adequate system to develop our project around. The concept of neural network is simple enough: “Neural networks are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. They interpret sensory data through a kind of machine perception, labeling or clustering raw input” (Skymind, 2015). In broader terms, they mimic the human brain’s ability to learn new information through a series of steps, by clustering and classifying data that may otherwise seem unrelated to each other. As magical as neural networks might seem, there are a handful of pitfalls we had to avoid when developing a solution for our project. Among those is the case of overfitting, “[a] situation when our neural network is so closely fitted to the training set that it is difficult to generalize and make predictions for new data” (Skalshi, 2018). This is a common issue for neural networks, and it prompted us to search for a specific model that would maximize effectiveness as well as minimize any shortcomings we may face.

In the case of our project, we decided that a Recurrent Neural Network (RNN) would be the best place to start spending our time into researching. RNNs are a “powerful set of artificial neural network algorithms especially useful for processing sequential data such as sound, time series (sensor) data or written natural language” (Skymind, 2015). Rather than using a feed-forward approach, RNNs use a feedback loop. “For example, if a net is exposed to a word letter by letter, and it is asked to guess each following letter, the first letter of a word will help determine what a recurrent net thinks the second letter will be, etc.” (Skymind, 2015). In fact, this was the very foundation our project would be based on. The characters of the song lyrics would be tokenized individually and run through a series of algorithms that would calculate values associated with preceding letters. Over the course of many tests, trial runs, and training, we hoped to have at least a proof of concept model that fulfilled our basic requirements.

At the very root, an RNN is a data-structure full of connections between nodes that form a directed graph which represents a temporal sequence. RNNs use a sequence as input and classify that sequence in several ways to learn from it. By splitting the sequence into tokens, the individual tokens can be pushed through the digraph between nodes and in turn develop a model that can accurately predict new information based upon what it is given. Consider this example sentence: “Where did you study last night?” It is a straight forward question in the English language and meaning can easily be derived, however, if the sequence was manipulated and the sentence was changed to: “are Where night? you last study”, there is no way to understand any underlying meaning. With this simple example, we can see that if a sequence is not ordered, then we would never understand its meaning. That is the core of the RNN, for it takes a sequence of tokens, and operates under the assumption that the sequence of those tokens is crucial to connote meaning. Under the structure of an RNN, sequences are the crux of not only their training methods, but also their methods used for synthetic output generation. RNNs on their own to do not understand the meaning behind the tokens used, they merely memorize optimal sequences between tokens based upon an input sequence, thus the meaning is derived from the order of the sequence alone.

For RNNs, every token of a given sequence plays a vital role in determining what the next token in the sequence will be, and they do this with Late Short-Term Memory (LSTM) nodes. LSTM nodes have the unique ability to remember past information contained within sequences via internal loops. These nodes make up the hidden states within the RNN and have a ‘recurrent’ structure that connect their output into the next LSTM node in the layer. Each LSTM node is connected in multiple locations, which aids in their ability to gather information and learn. LSTM nodes gradually get ‘smarter’ as the training process continues, for the output token they receive progressively gets more accurate at the time steps continue. Thus, by the end of the training session, the input token and recurrent output token will direct the LSTM node to output increasingly predictable results. The increasing predictability over the course of training an RNN model is calculated and quantified via the use of a phenomenon known as Value Loss.

LSTM nodes generate an output, which is a vector of real numbers that represent probabilities for a given token. Each index within the vector represents a unique token, and the index containing the highest probability will be selected as the output for that time-step within the sequence. Value loss is the difference between the target, or expected output, and the actual output that is currently generated by the model. The numerical scalar that represents Value Loss at each time-step is a positive real number, and is calculated by subtracting the target vector from the vector generated. Let’s imagine that the target vector of the LSTM at the current time-step is [0, 1, 0] and the output that is generated by the LSTM is [0.12, -0.987, 0.56]. That would mean that the rounded vector generated by the LSTM would be [0, 0, 1] since the third index is the highest, and would be deemed incorrect due to its misalignment with the target vector. To determine the Value Loss of this time-step, the value at the correct index represented by the target vector [0, 1, 0] would be subtracted by the correct index contained within the generated vector: 1 - -0.987 = 1.987. The Value Loss for this time-step is quite high, and correlates with the fact that the prediction and ultimately the generation in the example time-step was not what was expected. In the beginning of training, Value Losses are quite high, like in this given example, since the model has not yet traversed enough of the input sequence to contain the necessary knowledge to accurately predict future tokens.

RNNs are the current state of the art tool for sequential data analysis and are used by massive technology companies like Apple with Siri and Google with Voice search (Donges). Recurrent Neural Networks have a wide field of applications for researchers as well, such as speech synthesis, computational biology, earth science, and astrobiology. In the field of speech recognition, RNNs are used to model end-to-end networking such as verbal and textual speech recognition. Remarkably, RNNs recognition rate is higher than Hidden Markov Models and feedforward deep neural networks under the same conditions. Furthermore, RNNs can solve the problem that regular neural networks fail to answer, for they can use data from previous iterations to enhance their calculations. In the realm of speech processing, RNNs are widely used for machine translation, and there is a multitude of research on the topics of translation between languages via LSTM models. RNNs are also currently used for generating text, including online customer service in the form of chat-bots. For our project, we tried to do our best to utilize the powerful tools of RNNs and LSTM nodes to do similar work by writing artificial Led Zeppelin lyrics.

Implementing the RNN was now understood on a theoretical level, and it was time to create a working model of our very own. The implementation research led us to a webpage created by Albert Lai, where he lists the order of operations for successfully building an RNN. He uses his RNN to generate synthetic Shakespeare text, so we decided to build a similar model to generate our artificial lyrics. We closely followed Lai’s guidelines to create a viable product to demonstrate to the class. First, we began by compiling a massive text document containing all the lyrics ever written by Led Zeppelin, and conducted the necessary preprocessing of removing extraneous characters and symbols. The address to reach the website that contained the lyrics for Led Zeppelin is cited below for your convenience. Now that the dataset was prepared, it was time to begin coding the RNN, and we decided to create ours in Python. Python is a scripting language that includes many scientific libraries that help in the production of a RNN, including Numpy and PyTorch. There were struggles within the group to correctly download PyTorch and implement the correct classpath for the necessary driver, so we concluded that downloading a scientific Python application would be a smart alternative. After some brief research, we picked Anaconda, as it came with many scientific libraries, IDE’s, and PyTorch all rolled into one convenient package. Now that we had the required libraries and IDE’s all sorted out, we began to code.

Our coding began by reading in the dataset from the file containing all the Led Zeppelin lyrics. With the lyrics saved to a variable, they were then pushed through an encoding process. It began by placing the lyrics into a set, which eliminated all duplicates and created a list containing each unique character. Afterwards, the set was used to put each unique character into a dictionary, automatically generating their respective integer keys. Once the characters were mapped to integers, the same process was done in reverse, so that each integer had a corresponding character as its key. By having both dictionaries, looking up future characters or integers would be conducted in O(1) runtime within our RNN. The last step of the encoding process was to turn all the characters contained in the original Led Zeppelin lyrics into their appropriate integers. This was streamlined via the Numpy library, which referenced our dictionaries and transformed each character into its corresponding integer before placing it at the end of the encoded Numpy array. Now that our data was read in and correctly encoded, it was time to produce our RNN model.

The RNN model was generated by creating an RNN class, with the help of the PyTorch built-in neural network (nn) library. The RNN class created our model to be used as an object for future reference within the training sessions, and began with a constructor that defined the parameters required for proper functionality. The list of key parameters for our RNN include: dropProbability, hiddenLayers, hiddenStates, and learningRate. These parameters are found in every RNN, but they are dynamically created in ours so that if the RNN is trained repeatedly with differing input sources, the model can change too. The previously encoded lyric data is also passed into the RNN constructor to allow for the model to store the complete sequence of information. The constructor continues by defining the Late Short-Term Memory (LSTM) nodes, which came built-in with the nn library. The LSTM nodes are essential to our RNN, for they facilitate the learning process, and luckily their internal algorithms and functionality are predefined. Next, the dropout parameter is set using the built-in nn method, which will help the nodes pick where to send their output within the hidden layers. Dropout layers are critical within an RNN, for they help nodes learn faster during training by forcing them to connect to layers that they would not normally connect with. Dropout occurs by dynamically turning nodes off within the hidden layers, so that there are limited options for where the output can be passed. Lastly, within the constructor is the parameter fc, which connects all the nodes together internally and sets the stage for the hidden layers to communicate their output during future training. This RNN class creates the model that will be used as an object throughout the training process in the rest of our code.

Once the RNN model was created, it was time to begin writing the necessary code to train it. We created a method called teachNetwork(), which took in parameters, including: the RNN model, the encoded lyric data, the number of training sessions, the number of sequences to be processed, and the maximum length per sequence. The method also required some of the built-in variables required in training RNN’s, including gradientClipping and learningRate. Gradient clipping, normalizing data that exceeds allowable values, is crucial when training an RNN, especially when using LSTM nodes, to avoid what are known as “exploding gradients.” As Jason Brownlee explains, exploding gradients are, “large updates to weights during training [that] can cause a numerical overflow or underflow”(Brownlee, 2019). For this same reason, the learningRate variable is set to be 0.001, so that the volume of information calculated does not overwhelm the model during training. The teachNetwork() method begins by setting the RNN into training mode, by using the built-in nn train() method. Once the RNN was set to start training, it commenced computing optimum characters based upon the input data. The process of computing optimum characters is conducted with the built-in Adam Algorithm, and the output of which contains the expected output vector based upon the input given. The RNN had the goal of trying to replicate those optimum characters, and the process was accomplished by establishing criteria that could gauge whether the predicted output was diverging from the expected output. The criteria for which the divergence could be quantified was established using the built-in Cross Entropy Loss function provided by nn. Now that the training method was initialized, it was time to code the loops that would iterate through the encoded array and update the internal parameters of the LSTM nodes as the model trained.

The training method was set to iterate as many times as indicated by the number of training sessions in the parameters. The outer loop begins by locating the nearest LSTM node within the hidden layer, by calling the init\_hidden() method, and officially starts the process of traversing and training the model. The init\_hidden() method is contained within the recurrentNN class, and is used to collect the output vector contained in the current LSTM node, and return its value. Once the initial LSTM value is collected, it is time to prepare the encoded integer array for being sent forward through the RNN. RNN’s require vectors of information be passed through them, thus the encoded array was iterated through and each element was converted into its corresponding One Hot Vector (OHV). OHV’s are the vector representation of an integer value, containing all zeros except for a 1 located at the index corresponding to the given integer value. If the integer is 21, then only the 21st index of its OHV will contain a 1. Once the OHV’s are created, the LSTM output vector collected previously is referenced, and compared to all previous LSTM output vectors. If it is the first time running, then both the first output vector and the set it is compared to are null, but if it is deep within the training session, then the current output vector has a substantial set to compare itself with. As the set grows, the LSTMs get better at calculating the correct output for the desired characters, since they can use all previous vectors to determine the matrices for their calculation. Before continuing the training, the gradients contained within the RNN must be reset to zero, otherwise there would be no ability to accumulate correct gradients during the backward propagation later. Using the past LSTM vectors and the OHV of the input character, the optimal character at this time-step is generated. This generated character sets the stage to see if the training sessions are improving the reliability of the model via the introduction of Value Loss. The Value Loss of each time-step is calculated based upon how divergent the generated character is from the target character. Once the Value Loss is calculated, the weights (probabilities) for each hidden node must be reconfigured, to help train the model more accurately in future iterations, thus the built-in backward() method is used to instantiate backward propagation. As previously stated, these new weights assigned to the hidden nodes are accumulated gradients which are normalized to deliver more manageable values ranging from -1 to 1. Luckily, there is a built-in function for nn called clip\_grad\_norm\_ which could handle all our needs for compressing the gradients to avoid exponential numbers. Once the gradient compressing is accomplished and new weights were calculated for the hidden LSTM nodes, the built-in PyTorch method step() is used to update all the weights for every LSTM node of the network.

A standard procedure amongst RNN’s is to print statistics to the console periodically throughout each training step so that the user knows how the training of the model is progressing. That is why at the end of the training method, we added a conditional statement to determine where training currently resided, and what current loss statistics were available. The RNN’s training is momentarily paused by being set to evaluation mode, so that loss statistics contained within can be collected. All loss statistics available at each corresponding update are appended to an array containing the Value Loss data. Once all the losses for each time step are appended, the average Value Loss is outputted using the built-in Numpy array mean function. Once the current loss statistics are done being collected, the RNN is put back into train mode and allowed to continue iterating through the encoded dataset. Over the course of training, the LSTM’s will get incrementally diminished loss statistics, which indicates that the generated characters are more closely matching the target characters. After the training session, the model is trained to accurately predict new characters based upon current ones.

Now that training the model was complete, the final stages of our code are now needed, which includes the driver portion of the program. First, the trained model is saved using the standard protocol, and the model can be both retrieved and used in separate programs if so desired by the user. Now that the trained model is saved, all there is left to do is give the model an initial sequence of characters for it to read in and generate corresponding output. In our code, we did so by creating a method called generateLyrics() and passed in “The” as our first sequence. The first sequence is immediately placed into a set and iterated through to get the prediction of new characters started. The initial sequence is also placed into a list which we use as a staging data structure to accumulate and append all generated characters that will comprise our artificial Led Zeppelin lyrics. Every character that is generated and appended together is determined using a method called chooseNext(), which takes the latest character in the sequence as a parameter, then compares it to the trained model’s LSTM set of vectors, which returns the final output vector corresponding to the next character to be generated. Some internal calculations occur within the chooseNext() method, including: turning the current character’s integer value into a OHV, running the OHV through the model to get back the generated vector, and comparing the calculated softmax (vector of probabilities) with the generated vector to see if it does indeed have the optimal probability for the next character in the sequence. When all is said, and done, the built-in Numpy random function is used to increase the level of variation contained within the generated outputs, so that there will be more unique sequences of characters and fewer identical copies of the original lyrics. Numpy random is used in the effort to avoid any chance of accidentally overfitting the original data.

The program produced lyrics that were similar, but noticeably different in many areas from the original dataset. Since the RNN read and then generated a single character at a time, there is little-to-no correct grammar in the output, and many words have unintelligible spelling. We found that as we tested our RNN, using more training steps and longer sequences of characters to be read in each step did nothing but detract from the final output of model. Over a long period of testing, we found that reducing the number of training steps and compressing the sequences to between 10-20 characters at a time yielded improving results. Once we had improved the accuracy of the model, it began to produce more acceptable results, including some relatively complete sentences that resembled Led Zeppelin lyrics. We hypothesize that results improved with diminished sequence lengths because many the stanzas within the original Led Zeppelin lyrics are relatively short in length, so the short sequences better suited the original text.

The RNN we created trained the LSTM nodes and predicted future characters based upon existing ones. Thanks to the capabilities of Python and the scientific libraries of PyTorch and Numpy, the process of implementing this RNN was much more straightforward than it would have been otherwise. We used built-in functions whenever possible, and learned a multitude about the underlying concepts of machine learning along the way. In a future project, we could use our RNN to model more than just the lyrics of Led Zeppelin. RNN’s have the capability to learn some incredibly difficult sequences and we could implement them to create not only the lyrics, but also the music and maybe even the album cover-art in the future. Creating this RNN has proved to be equally challenging and rewarding, and we hope to be able to continue using them in the future.

Bibliography

Britz, D. (2016, July 08). Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs.

Retrieved from <http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/>

Che, Z., Purushotham, S., Cho, K., Sontag, D., & Liu, Y. (2018, April 17). Recurrent Neural

Networks for Multivariate Time Series with Missing Values. Retrieved from <https://www.nature.com/articles/s41598-018-24271-9>

Donges, N., & Donges, N. (2018, February 25). Recurrent Neural Networks and LSTM.

Retrieved from <https://towardsdatascience.com/recurrent-neural-networks-and-lstm-4b601dd822a5>

Jason Brownlee, How to Avoid Exploding Gradients in Neural Networks with Gradient

Clipping. (n.d.). Retrieved from <https://machinelearningmastery.com/how-to-avoid->exploding-gradients-in-neural-networks-with-gradient-clipping/

Lai, Albert. “Writing Like Shakespeare with Machine Learning in Pytorch.” *Towards Data*

*Science*, Towards Data Science, 8 Feb. 2019, towardsdatascience.com/writing-like-shakespeare-with-machine-learning-in-pytorch-d77f851d910c

Eugenio, C. (2018, April 13). The fall of RNN / LSTM. Retrieved from <https://towardsdatascience.com/the-fall-of-rnn-lstm-2d1594c74ce0>

N.A., Retrieved from <https://www.allthelyrics.com/lyrics/led_zeppelin>

N.A., A Beginner's Guide to Neural Networks and Deep Learning. (n.d.). Retrieved from

<https://skymind.ai/wiki/neural-network>

N.A., Recurrent Networks. (n.d.). Retrieved from <https://skymind.ai/wiki/recurrent-network-rnn>

Schmidhuber, J. (2017). Recurrent Neural Networks. Retrieved from <http://people.idsia.ch/~juergen/rnn.html>

Skalski, P., & Skalski, P. (2018, September 07). Preventing Deep Neural Network from

Overfitting. Retrieved from <https://towardsdatascience.com/preventing-deep-neural->network-from-overfitting-953458db800a