HW2 Report: CNN vs. LSTM

Sheryll Martutartus

Abstract

In HW2, our goal was to better understand Convolutional Neural Networks (CNN) and Long Short-Term Memory (LSTM) models by creating such models and testing them on a dataset provided by Yelp. Our classification was on sentiment detection based on the Yelp review provided- a written response as well as atmosphere rating ("cool", "funny", "useful") and a five-star rating provided by the user. With the help of the given *Towards Data Science* guideline, I was able to extract the necessary data, utilized Word2Vec to build embeddings, and build both models to train/test the data. Finally, after running the data through the models, we obtained and analyze the results to compare the differences in CNNs and LSTMs.

Contents

1. Background

Before we dive into the experimentation and results of our testing, we want to provide some background information on the types of models we are using as well as the dataset.

1.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of deep learning model commonly used for image classification and natural language processing. They consist of three main layers: convolutional layer, pooling layer, and fully connected layer [1]. The convolutional layer is important in feature extraction. It works by taking an input (for example, the image), sliding a filter of weights over top the image, and applying a dot product of the filter and image to obtain a feature map. Typically, after the convolutional layer, ReLU (or any type of activation function) is applied to the feature map to introduce non-linearity. Next is the pooling layer. The pooling layer aids in dimensionality reduction as it slides over the feature map, but it applies an aggregation function instead of weights; some common types of pooling layers are max pooling (taking the maximum value) or average pooling (taking the average value). After performing several rounds of these layers, the final layer in the network is the fully connected layer. This layer is descriptive of its name as it takes into consideration all of the nodes and connects them in the output layer, which is then typically applied to SoftMax operation to obtain the classification. While CNNs are commonly found in image and video classification, they can also be utilized for natural language processing as well.

1.2 Long Short-Term Memory

Long Short-Term Memory (LSTM) networks are also a type of neural networks, but it incorporates recurrency to learn the order and dependencies within the data to predict sequences [2]. LSTMs operate by utilizing gates and history of the previous inputs to predict the next sequence. It involves three types of gates- forget gate, input gate, and output gate- that all essentially decide ‘how much’ of each input we want to take for the next prediction. Each gate executes different operations (sigmoid, tanh) and utilizes both the current input value and the history of previous states, which is where the recurrency of the network comes in. Finally, after each gate calculates its output, they are all joined

together to predict the next state in the sequence. Overall, the recurrency within the model makes it optimal for language modeling, speech recognition, and machine translation [2].

1.3 Yelp Dataset

In order to experiment with these two types of neural networks, we utilized the Yelp reviews dataset, which can be found at [3]. Consisting of multiple JSON files, we focused primarily on the reviews.json file. Each entry of this file consisted of a review_id, user_id, business_id, stars, date, text, useful, funny, and cool rating. In our project, we focused primarily on the text review to extract the sentiment behind the rating (good, bad, neutral) along with other features such as the number of stars and the useful/funny/cool ratings given. To aid us in our CNN creation and sentiment extraction, we followed along with the provided code at [4]. This code example walks us through how to load and extract important information from the dataset, process the text and retrieve Word2Vec embeddings, and then apply these embeddings to the CNN to get the sentiment predictions. We also used this as a model in creating the LSTM.

2. Experimentation

As mentioned before, we followed the implementation of [4] to get our word embeddings and create a CNN. After loading in the dataset, we began by analyzing the data and creating a couple other keys in the dictionary for each review that would be of use to us. One extra piece of information we extracted was mapping the number of stars given in the review to a sentiment. For example, 2 or less stars received a -1 (bad review), 3 stars received 0 (neutral review), and 4 or more received 1 (good review); this is how we will base our prediction on. From the text reviews given, we preprocessed the data by removing all the stop words (words that occur a lot such as ‘the’ yet hold little meaning), tokenized each word from each review (making each word as its own element in a list), then finally stemmed each of the tokenized words (getting the core of the word and extracting pre/suffixes). After obtaining all the necessary information, we split the data into a training and testing set by a 70-30 split.

Next is processing each review to obtain a word embedding by utilizing Word2Vec. In our Word2Vec model, we utilized skip-gram method with a window size

of 3 to create a vector size of 500 for each review. We used our stemmed words as input and saved our model for later use in the CNN.

Our last step was to create the CNN model. Our CNN implementation consisted of an embedding layer, four convolutional layers (each with different filter sizes) followed by a tanh activation and max pooling layer repeated in each iteration. Finally, we pulled it all together in the fully connected layer and obtained our output after a SoftMax layer. We trained the model for 30 epochs, and then tested its prediction capabilities on the test data. We applied a similar set up for the LSTM model (data preprocessing, Word2Vec creation, etc.), yet instead of the convolutional layers, we used an LSTM layer followed by the same tanh, max pooling, and fully connected layers.

3. Results

Within the Yelp dataset, there were about 15,000 reviews, yet for testing and time purposes, we only extracted a subset of these reviews. I began by extracting only 100 reviews just to make sure the CNN and LSTM runs effectively. With just 100 reviews, we achieved about a 76% accuracy on the testing data as shown below in the classification report.

	precision	recall	f1-score	support
	0.767	1.000	0.868	23.000
accuracy	0.767	0.767	0.767	0.767
weighted avg	0.588	0.767	0.665	30.000
macro avg	0.256	0.333	0.289	30.000

Figure 1: Classification report from 100 reviews

What I found interesting in the results were that both the CNN and LSTM produced the same accuracy and almost the same results for the classification report. Though the word embeddings and most of the model's layers were the same, I was surprised to see that both the convolutional layer and LSTM layer produced the same accuracy for the testing data. Just to make sure there wasn't an error in these results, I checked the loss calculated at each epoch, and they did have different degrees of loss as shown in the graphs below.

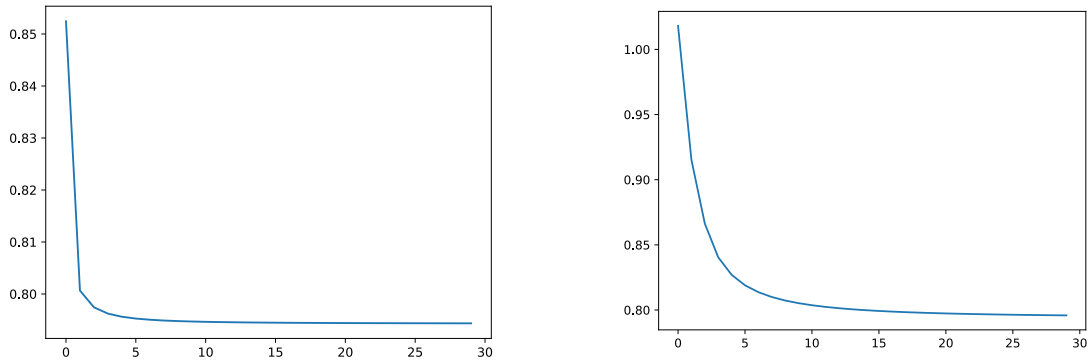

Figure 2: Loss calculated per epoch of training. Left for CNN and right for LSTM.

As stated before, both models resulted in the same accuracy of about 76% which is why both graphs end around that mark. However, we can see a difference in the starting points and the shape of the curve for each model. The LSTM model starts at about 100% loss, while the CNN starts at about 85%. The CNN drops quicker to about 80% accuracy in less than 5 epochs, while it takes the LSTM longer, about 15 epochs, to reach the 80% mark. This leads us to believe there is a difference between the convolutional layer and the LSTM, yet by very little.

Though we did focus our initial test on only 100 of the reviews, we wanted to test if increasing the dataset size would improve the accuracy. We increased the dataset size to 1,000 samples, yet we noticed it decreased the testing accuracy to about 66%. Again, we increased the size to about 2,000 then 3,000 samples, and we noticed for both values, the accuracy was at 66%. Therefore, we believe that with these word embeddings and model structures, the accuracy will plateau at that range, yet it is interesting how the accuracy decreased from 76% to 66%. Typically, we observe the accuracy increasing as we increase the dataset size, yet it was the opposite. This could be due to overfitting with only 100 samples and having 1,000+ samples create a better generalization for the data since we are exposed to a greater variety. It could also be that with 100 samples, we are only exposed to a certain limited range of data, and as we increase the data, we get a greater variety, so we need to increase our generalization ability.

One last observation we discovered was when we changed the activation function from tanh to ReLU. Tanh places the values between [-1,1], while ReLU takes the max(0,x), therefore, ReLU has a greater range of values. For the CNN, it resulted in the same testing accuracy of about 76% with the same results of the classification report. However, we did notice a difference in the loss per epoch once again as shown in the graphs below.

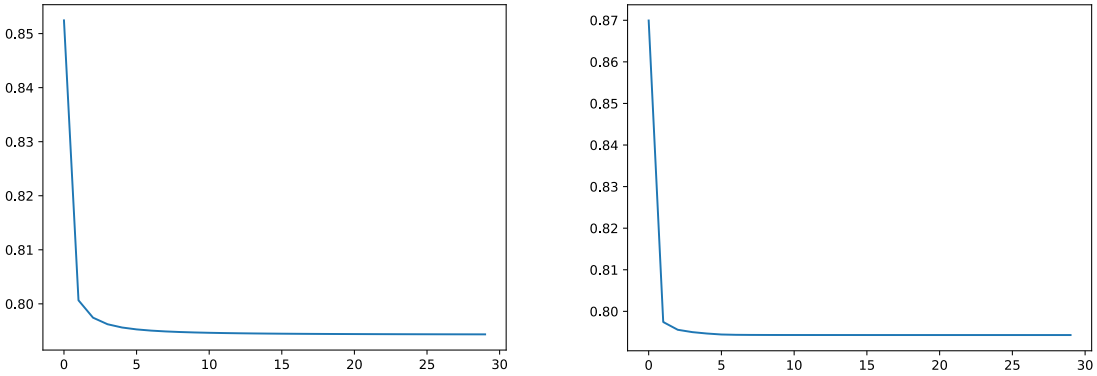

Figure 3: Loss per epoch of the training data. Left is CNN with Tanh activation and right is CNN with ReLU activation.

Above we can see a slight difference in the loss within the first couple epochs between using the tanh and ReLU activation function. At around ~2 epochs, they both drop drastically from 85%, with tanh dropping to around 80% and ReLU dropping to about 77%. ReLU looks to plateau right after that drop whereas tanh has a slight curve, then begins to hit the same plateau yet at around 5+ epochs. ReLU could be a better activation function than tanh in this example because instead of normalizing the values between [-1,1], it allows for a greater range of value, which means greater representation of values and information passed through.

In terms of trying ReLU for the LSTM, it seemed to perform faster per epoch than the ReLU for CNN, which is surprising as tanh seemed to perform faster per epoch on the CNN than in LSTM. Again, the accuracy and classification report were about the same for using ReLU, yet contrary to the CNN, the loss produced a similar graph for ReLU vs. tanh. The values were slightly different, yet not enough to distinguish themselves in the

graph. It's interesting to see how different activation functions affect models differently. One possible explanation is because LSTM is a recurrent model, and it's recurrency of values circulating within the model and its weights may not be greatly affected by the type of activation function used; CNNs do not utilize recurrency so the values outputted from the activation function may result in different values traversing the rest of the graph, whereas LSTM utilizes different gates to decide what to keep and what to forget, which may end up keeping/discarding the same information despite the activation functions.

4. Analysis

Relating back to the main focus of the project, we want to compare the results of the CNN and LSTM on sentiment classification and analyze how/why they are different. In the previous section, we analyzed the classification and accuracy results, seeing that they were both similar in the accuracy in which they classified the sentiment of the reviews. Another way we can analyze their difference is in how long it takes each of them to train per epoch. We began to mention it in the previous section, but we observed a difference in training time between the two different activation functions. With the tanh activation function, we found them to be very close in the execution time per epoch, yet CNN was slightly quicker. I believe CNN performs quicker because there is no recurrency like there is in LSTM, which could add to the execution time. The main difference in the structures we used was the convolutional layer and the LSTM layer. As the difference was in only one layer, we do not expect to see a great difference in execution time, yet a baseline structure was necessary for comparison. The convolutional layer just involves sliding the filter over the input and computing the dot product, which is less computationally/time expensive than passing it through the three gates and performing the computations in what to keep/forget from the memory.

Additionally, we can analyze the number of parameters utilized in each model, which could help us understand the accuracy results and the time execution difference. To reiterate, the only difference in the neural network model was the convolutional layer was replaced by a LSTM layer, with the rest (activation function, pooling, fully connected layer) staying the same. Therefore, we will only analyze the difference in the parameters for the convolutional layer and for the LSTM layer. There were four convolutional layers in the CNN, each with an input of 1 and output of 10, a filter size

varying ({1,2,3,5}, 500), padding varying of ({0,1,2,4}, 0) and a stride of 1x1. To calculate the number of parameters for the convolutional layer, we take the width of the filter **m** by the height **n** by the number of filters **d**, add 1 because of the bias, and then multiply by the shape of the current filter **k** [5]**.** So, in this case we have varying filter sizes, so let's start with 1 (height) x 500 (width) = 500 x 1 (input from previous layer) = 500 + 1 (bias) = 501 x 10 (output dimension) = 5,010 parameters. That's just for the one convolutional layer with the filter of 1x500. Continuing with the same process, we get 10,010 for 2x500 filter, 15,010 for 3x500 filter, and 25,010 parameters for 5x500 filter.

After calculating the number of parameters for the convolutional layers, we want to compare it with the number of parameters for a LSTM layer. Based on [6], we can calculate the number of parameters in a LSTM by taking the number of activation functions 4 * (**n**+**m**+1), where **n** is the input dimension, **m** is the number of LSTM units in a layer, and 1 is the bias. Then finally, take all this multiplied by **m** again to get the number of parameters. In our case, we have 4 * (500+10+1) * 10 = 20,440 parameters. This is about mid-range of the parameters used for the CNN. Depending on the size of the CNN and its filters, LSTMs usually utilize more parameters as there's more gates, more history saved, and more calculations done in the layer rather than based on just the filter's dimension. The fact that LSTM has more parameters helps to explain why CNNs perform quicker per epoch than LSTMs do (less calculations).

Overall, this project helped us learn more about CNNs and LSTMs by having to create and implement our own models using PyTorch; I also got the chance to experiment with using Keras for creating the LSTM model (even though the end model was utilizing PyTorch). Not only did we learn about how to construct our own neural networks and how the inputs need to line up for each layer, but we also got experience with language preprocessing and creating word embeddings from Word2Vec. Finally, we used the classification results to help us better understand how each model operates differently and why we could obtain different results and execution times. In further experiments, I would like to focus on just on type of model and see how varying the structure and layers could impact the classification.

5. Citations

[1] https://www.ibm.com/topics/convolutional-neural-networks

[2] https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/

[3] https://www.yelp.com/dataset

[4] https://towardsdatascience.com/sentiment-classification-using-cnn-in-pytorch-fba3c6840430

[5] https://towardsdatascience.com/understanding-and-calculating-the-number-of-parameters-in-convolution-neural-networks-cnns-fc88790d530d

[6] https://medium.com/@priyadarshi.cse/calculating-number-of-parameters-in-a-lstm-unit-layer-7e491978e1e4