NLP 243 Assignment 1

Ethan Sin
UC Santa Cruz
elsin@ucsc.edu

1 Introduction

The task in this assignment is to take a corpus of movie reviews and apply different neural network techniques using PyTorch to assign slot tags to each word.

The dataset is given as a .csv with two main columns, 'utterances' and 'IOB Slot tags'. Supposedly, the utterances are already tokenized and separated by whitespace, with each token corresponding to a slot tag, which are also separated by whitespace.

For example, 'what is Tom Cruise's birthday' looks like 'what is tom cruise 's birthday', and 'O O B_person I_person O O' here the clitic has been separated from 'cruise', and the number of whitespace separated items in the slot tag line is the same as the number of tokens in the utterance. As for the slot tags, they will either be an 'O', indicating that the token has no relevant tag, or they will be a 'B' or 'I' tag. Each 'B' and 'I' tag has an underscore then a label for what type of token it is. The 'B' and the 'I' then inform if it is the beginning of that type of token or a continuation. In the Tom Cruise example above, you can see that it labels Tom Cruise as a person, has 'B' over 'tom', and 'I' over 'cruise', since 'B' is the beginning of that and the 'I' is a continuation labeling the same person. There are 27 labels in total including 'O', and there are 13 unique types of labels such as 'person' or 'movie'. It is worth noting that some of the lines are improperly formatted or include unwanted characters, in these cases the line of data will be ignored and not used in my training.

The final product for this task is to take a training dataset in this format and predict on a test dataset with the same format as the 'utterances', and our output should be tags in the same format as the 'IOB Slot tags' here.

2 Models

As we are tagging individual words, I decided to reformat the data drastically. Rather than taking in each utterance as a whole input, I created trigrams for each word with the significant word at the center. Then I associated the whole trigram with the label belonging to the significant word. As such, if the word in question was the beginning or the end of an utterance, I would need to pad the trigram with start and stop tokens. Thus, 'what is tom cruise 's birthday' should become several elements that look like, [("<s> what is", O), ("what is tom", O), ("is tom cruise", B_person), ..., ("'s birthday </s>", O)]. I would do this with every utterance, and each row would look similar to one of the example tuples above. This way, each trigram with a label can be simplified to a document classification task.

As for tokenization, I will not be using any method besides separating by whitespace, as the data could be ruined if we tokenize differently. Each token already corresponds to a tag, so I do not want to risk changing that.

The rest of the feature engineering is done using code provided by the starter code in section. I will be using word embeddings, specifically the wikipedia word embedding model provided by Gensim [2]. Each trigram will be encoded to three numbers that are indices for retrieving the appropriate word embedding from the model. When they are converted to tensors to for the model to train and predict on, they will be come a 3 by 600 tensor, with 3 being each word in the trigram, 300 of the weights in each row being frozen weights that represent the origial word embedding weights, and the other 300 will be initialized as the word embedding weights, but are trainable and will be adjusted in the backpropagation step of the model based on the loss. If they are unknown, start, or stop tokens, they are added to the wikipedia model at the end

and given different weights. The outputs will be encoded too, but simply as numbers.

2.1 Convolutional Neural Network

For this task, I only experimented with one type of model, the convolutional neural network, or CNN. This is method normally applied to images, but is shown in testing to have fairly good performance for NLP tasks like slot tagging when used in conjunction with models like conditional random fields [1]. In short, a CNN layer has a window, or kernel, that applies a filter of weights over a set of weights in the data, associating that product to some label in the data. The kernel then slides over to a new area of weights in the data, and keeps doing this until it has covered everything. In training, it will then associate those products it found with the label that the training data has. In prediction, it will predict based on what the product says the data most likely is. Practically, it finds patterns in different areas of the data that may give valuable information to what label the data has.

For our data, the filter is applied to a set of weights it sees in the word embeddings between two words, and patterns are found that ideally will make two different but similar sequence of tokens produce a similar product. For example, ideally 'was luke' and 'is vader' have similar products over their word embeddings and the filter, since 'was' and 'is' are similar in meaning and 'luke' and 'vader' are likely unknown tokens. If they successfully look similar to the model, then the model should be able to predict 'vader' as 'B_char' if the training data has 'luke' as 'B_char'.

The CNN can typically be a number of convolutional layers that is added to a deep neural network with other types of layers like linear layers [3], but for my implementation I am only experimenting with training a single convolutional layer to make predictions. This implementation of the model is also taken from the starter code on CNN, combined with the encoding portion from the LSTM code. For prediction, I modified the evaluation portion of the model to not take batches or calculate loss, and it will just return the predictions on whatever data is passed in. The hyperparameters that are tuneable are the kernel size, dropout, and the size of the n-gram, and other parameters changed are the learning rate and epochs.

2.2 Challenges

Before discussing the experiments, I will explain my challenges with building the model and what I have learned. Initially, my understanding of a CNN was still very hazy, so taking that concept and applying it to an NLP problem when my understanding of word embeddings was also weak proved to be a challenge. I spent a week wrestling with the starter code and parsing through various online resources to try to understand the concepts and how to apply them to this dataset. When I was able to solidify my growing understanding of the concepts in office hours, then I felt much more comfortable handling the starter code. I understand what my goal is in shaping the dataset, and how the classes in the starter code access the word2vec embeddings, encode the input data with them, and use them for training in a CNN. I understand now how the filter of a CNN interacts with the tensor made with the inputs and word embeddings, and the idea of how it recognizes what class the input belongs to using what it sees in groups of word embedding weights. I now feel more comfortable in the future if I were to implement this model on my own, and after working with the starter code I also have a better understanding of where to start with using PyTorch for these models.

3 Experiments

The data was split at a 90-10 ratio. The parameters that I ended up tuning are the kernel size, dropout, learning rate, and number of epochs. I changed each of these in increments from the default values that were in the starter code, starting with the learning rate and number of epochs. As mentioned before, this singular convolutional layer is the only model that I had time to test.

It is worth noting the parameters that the starter code defaulted to which were: 50% dropout, 1e-2 learning rate, and 5 epochs. I am only using trigrams in this experiment, and the model I modified from was originally built for a different dataset for article classification rather than these trigrams, so the original kernel sizes were too big for my data. I defaulted to a kernel size of 3 by 100 as such.

4 Results

As I spent most of my time figuring out how to get the code to work, I did not get to test too many different methods. The table above shows what I did test. Overall, what I found is fairly straightforward.

default	0.15188
learning rate to 1e-3	0.21771
20 epochs	0.287
learning rate to 1e-4	0.21989
kernel size from 3 to 2	0.3211
dropout to 0.3	0.34472
kernel try 2 and 3	0.35097
dropout to 0.1, epoch to 25	0.38637
no dropout	0.37153

Table 1: Performance Comparison

I increased the epochs each time it seemed that the other parameters I changed made the loss continuously go down even approaching the last epochs, and when adjusting the learning rate I found that 1e-3 was a great place for my testing speed and number of epochs. Performance improved significantly when I adjusted the kernel height from 3 to 2, essentially having the filter look over individual bigrams separately rather than only looking at the whole trigram all at once. I also lowered the dropout ratio, and each time I did the performance improved by a notable amount. Another small improvement came with having the model try both kernel heights of 2 and 3 during training. The only notable loss in performance came with eliminating the dropout altogether.

5 Conclusion

Overall, I am satisfied with my growing understanding of deep learning tools like CNNs. Though it is a rather uncommon technique to use in NLP, it may be useful in the future for specific tasks. If I had the chance to do this again, I would attempt to implement everything myself as I believe that would also make it easier to train multiple convolutional layers and use them in a larger, deeper network with linear layers too. I think that the task of using CNNs with NLP also forced me to understand more how word embeddings are represented, which will prove useful for me any other time I will want to use word embeddings as a tool. I am also now curious to learn how to train my own word embeddings, too. For tasks like this in movie datasets I believe that it can be very useful to train word embeddings that are specialized for this type of vocabulary for better results.

References

[1] B. Liu and I. Lane. Attention-based recurrent neural network models for joint intent detection and slot filling. *Interspeech*, 2016:2288–2292, 2016.

[2] R. Rehurek and P. Sojka. Gensim–python framework for vector space modelling. *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, 3(2), 2011.

[3] A. Tam. Building a convolutional neural network in pytorch, 2023.