Deep Learning Project 3

Rishabh Gupta

MTech(CSA) - 15960

1 The Task

The task was to do Natural Language Inference on the Stanford's SNLI dataset. Given a pair of sentences, a *premise* and a *hypothesis*, classify whether it is an *entailment*, is *neutral* or is a *contradiction*. This is a classification problem with three classes.

2 SNLI dataset

The dataset contains 550,152 training pairs and 10,000 testing pairs. Some of these doesn't have a *gold_label*, so I removed them. I also created a validation set of almost same size as test set and kept the test set aside. Final count of each sets are:

Train	Validate	Test
538,379	10,988	9,824

Table 1: Dataset size

There was no class imbalance in the data.

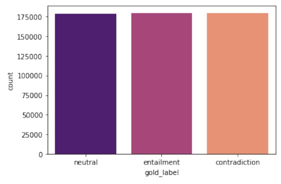


Fig. 1: Train Set

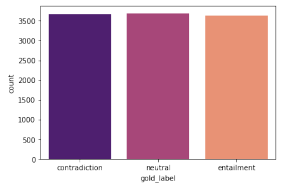


Fig. 2: Validation Set

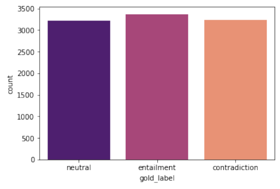


Fig. 3: Test Set

3 tfidf and logistic regression

The first task was to use tfidf features and logistic regression. For preprocessing the text data, I used *nltk* python library. First tokenized the sentences into words, then removed stop words and then did stemming. I used *SnowballStemmer* from *nltk*. The total size of my vocabulary was 20959.

For tfidf, I used *TfidfVectorizer* from *sklearn* library. Since idf values depend on the number of documents, it would not be a good idea to compute idf on validate set (or test set). So only the train data was used for idf computation, and then same idf values were used in validate and test set. The tf values were individually computed. (i.e. *TfidfVectorizer* was *fit_and_transform* on train data, but only *transform* on validate and test data). A sentence is considered as a document for tfidf computation.

Then I concatenated the feature vectors from premise and hypothesis, resulting in a vector of size 2*vocab_size.

I used logistic regression from *sklearn* library. Tried with different values of C. *max_iter* set to 500. Table 2 summarizes the results (in *w/o SWs* column).

I further observed that since the sentences length are not too long, there are chances that STOP WORDS might convey important information. For example *a* in premise and *one* in hypothesis can be important factors. So I tried without removing the stop words. The vocabulary size increased to 21038. This slightly improved the accuracies. The results are in Table 2 (in *with SWs* column).

Value	Training Set Accuracy		Validation Set Accuracy	
of C	w/o SWs	with SWs	w/o SWs	with SWs
0.1	62.29	64.60	61.52	63.49
1	64.50	66.75	62.29	63.60
10	65.85	68.33	62.24	**64.12**
100	66.11	68.67	61.72	63.03

Table 2: Accuracies of logistic regression model

I chose the final model which gave the highest validation accuracy (C=10 and not removing stop words). And trained the model on whole training data (train + validation set). The test accuracy of the final model is: **64.95%** (This includes all the 10,000 test samples, including those with empty *gold_label*. If we remove those samples from test set with empty *gold_label*, we get an accuracy of 66.11%).

4 Deep Models

The second task was to use Neural networks suited for sequential data like RNN, GRU, LSTM. I chose to use LSTMs. Library used: *PyTorch*.

My first model has an embedding layer, then a layer LSTM and then a sequential layer. In the forward pass, first we compute the embeddings for premise and hypothesis separately, then pass them to the LSTM layers, and then concatenate them both and pass the concatenated vector to the sequential layers. Since it was a classification task, I used Cross-entropy loss. The pre-processing was similar to what was done for logistic regression model. And I didn't remove the stop words (from previous experience).

I started by choosing the batch size accordingly to my GPU. I set it to 128, that was the best my GPU can handle.

Next thing was to select the learning rate and the optimization algorithm. I chose Adam. And tried different learning rates for 5 epochs to see which one converges better. The plot below shows convergence of different learning rates. I selected *lr = 0.001*, which is also the default in the library.

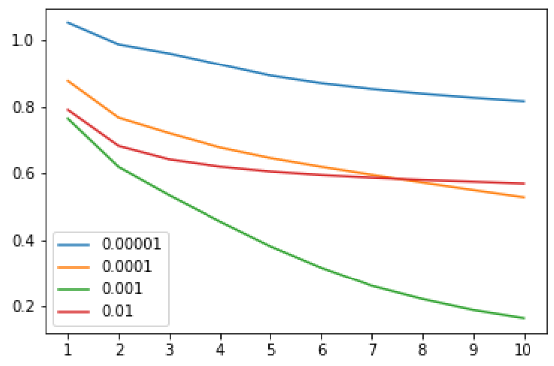


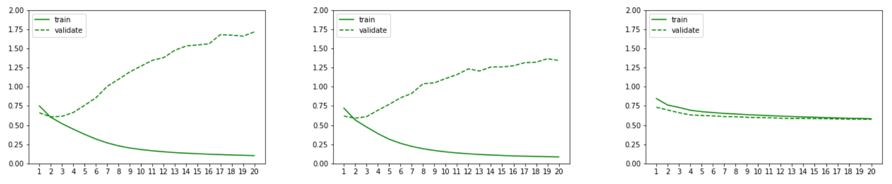
Fig. 4: Training loss for different learning rates

Then I tried with various modifications in the model, such as varying number of LSTM layers, varying number of hidden units, changing dropout rates. I also tried with bi-directional LSTM layers. The results are summarised in table 3. All these ran for 20 epochs.

No.	embedding	word	LSTM				Sequential Layers			Accuracy	
	dim	dropout	bi-dir	#layers	dim	dropout	#layers	dim	dropout	train	val
1	128	0.0	No	1	256	0.0	1	256	0.0	95.46%	71.59%
2	256	0.0	No	2	512	0.0	2	512	0.0	96.51%	74.14%
3	256	0.5	No	2	512	0.5	2	512	0.5	79.83%	76.43%
4	256	0.0	Yes	2	512	0.0	2	512	0.0	97.14%	74.80%
5	256	0.5	Yes	2	512	0.5	2	512	0.5	79.87%	76.63%
6	512	0.1	Yes	2	256	0.3	2	256	0.5	88.21%	76.68%
7	512	0.1	Yes	2	512	0.3	3	512	0.5	87.58%	76.60%
8	512	0.1	Yes	2	512	0.3	2	512	0.5	89.82%	76.65%

Table 3: Train and validation accuracy of different LSTM models

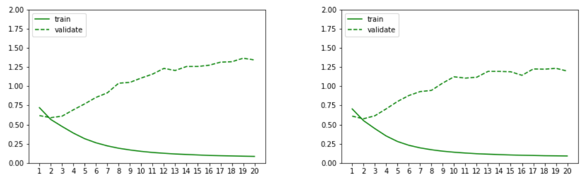
There were a lot of design choices to be made. Like, whether to increase or decrease hidden dimension, increase/decrease dropout, use bidirectional or not. And these design choices were made based on observations on previous models. I used the plots of train and validation losses for each of the previous model to make changes in the next model. All the plots (one for each model) is shown in the following figures [fig. 5, 6, 7]. The scale of each axis is same in subfigures in a given figure (for easier comparison), but different between two figures.



(a) Model 1 (b) Model 2 (c) Model 3

Fig. 5: Train and validate loss - Models 1, 2, 3

We can clearly see that initial models (1 and 2) are hugely overfitting and we can expect that they won't generalize well. Adding dropout as regularization significantly overcame this problem (fig. 5). Using Bi-directional LSTM also shows better results, as we can see by comparison of model 2 and 4 (which are same, except one is bi-directional and one is not) (fig. 6).

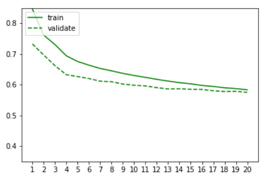


(a) Model 2 (b) Model 4

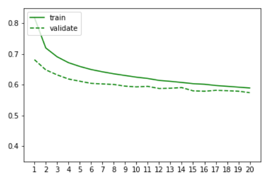
Fig. 6: Train and validate loss - Models 2, 4

Comparing the regularized models show that the regularization is very tight (fig. 7). Loosening the regularization reduces the training error but the validation error still plateaus at similar loss value (model 6). So I increased the model capacity while keeping same regularization (model 7).

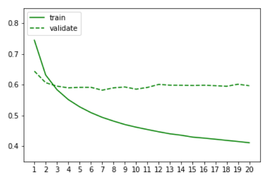
After analyzing the plots as well as numerical loss values, I came to the conclusion that model 6 performs the best, given we don't train it for too long



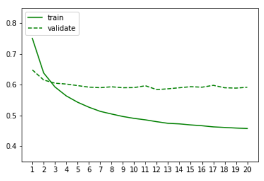
(a) Model 3



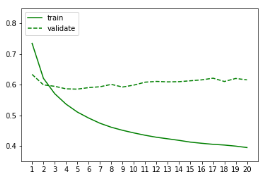
(b) Model 5



(c) Model 6



(d) Model 7



(e) Model 8

Fig. 7: Train and validate loss - Models 3, 5, 6, 7, 8

(do *early stopping*). So I trained my final model number 6 for 7 epochs on the full train data (train + validate). The final results are as follows:

Train Accuracy: 88.44%

Test Accuracy : 77.29%

Note: The above test accuracy is on all 10,000 test samples including one without gold label. If we remove those test samples, we get an accuracy of 78.67%.

The following plot is the train and test losses for each epoch for the final model.

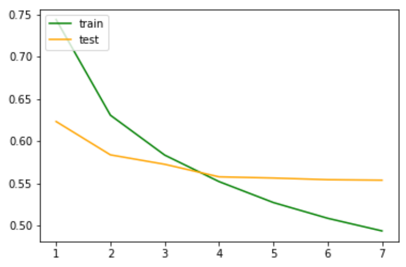


Fig. 8: Train and test loss of Final model

5 Files

There is one *main.py* file which contains the code for testing both the models and generate the expected output files. The files *part_1_training.ipynb* and *part_2_training.py* contains code for training both the models. The *plots* directory contains all the plots related to both the tasks. The models and other preprocessed files like tfidf values are saved in *model* directory.