Course: COSC 4337 Data Science II
Professor: Ricardo Vilalta
TA: Shaila Zaman
Group members: Kiet Dinh, Giai Tran, Cuong Phan

The Stanford Natural Language Inference (SNLI) Corpus
Data Modeling

Overview both models

The Stanford Natural Language Inference Corpus (SNLI) is a project mainly handling text-processing in machine learning tasks. The whole dataset contains 570k human-written English sentence pairs along with their labels for balanced classification tasks. In the past few decades, human-beings were using machine language (binary code) to communicate with a machine and have them to handle tasks which required hard work to be accomplished by humans.

Nowadays, with advances in technology, human beings are able to talk with a machine by our language and force the machine to understand what we are saying. This is where the Natural Language Processing (NLP) comes from.

In general, we have built two models for the SNLI classification task. Our models are able to distinguish the meanings of two input sentences and classify the relationship between them by a label. The implementation processes are much complicated compared to numerical data. In addition, we acknowledge that our model can be used further for implementing NLP applications such as chat-bot, question & answering, search engine like Google. Therefore, we learnt and used Tensorflow, a popular open-source framework for machine learning. Tensorflow has a strong set of libraries supporting text classification. In addition, Tensorflow provides a strong visualization library for improving, testing models as well.

Two models are using a pre-trained model called GloVe (Global Vectors for Word Representation) associated with this dataset that we obtain from Stanford NLP group website. GloVe is an unsupervised learning algorithm for obtaining vector representation for words. In module 1, we used Word2vec to transform word to vectors (Pennington et al. 2). However, the advantage of GloVe is that, unlike Word2vec, it does not rely just on local context information of words but incorporates word co-occurrence to obtain word vectors. For example, consider this

sentence from the training set: *A person is outdoors, on a horse.* The words that are next to each other in this sentence are marked as 1 while the others are marked as 0, which generates a co-occurrence matrix showing semantic relationships (Venkatachalam).

Moreover, distances between words could be computed as well. GloVe is used to combine with the training data to build a vocabulary for our own model. Acquiring a fast processing model is one of our highest priorities, we transform all of csv files to text files with a txt extension.

	A	person	is	outdoors	on	horse
A	0	1	0	0	1	1
person	1	0	1	0	0	0
is	0	1	0	1	0	0
outdoors	0	0	1	0	1	0
on	1	0	0	1	0	0
horse	1	0	0	0	0	0

Figure 1. Co-occurrence matrix example.

```
heutral ||| A person on a horse jumps over a broken down airplane. ||| A person is training his horse for a competition.
```

We dropped all feature and only three of them are remain: 'Gold_Label' ||| 'Sentence1' ||| 'Sentence2'. We will use the same text file data for both models.

Embedding layer, one of the most powerful layers in Tensorflow for NLP. The layer encodes each word with a unique number. Word embeddings provide us a way to use an efficient, dense representation in which similar words have similar encoding (Word Embeddings: TensorFlow Core). Instead of specifying the values for the embedding manually, they are trainable. The weights for the embedding layer are randomly initialized. After that, they are adjusted via backpropagation. The ways we encode the input embedding layer are different from each model.

1. Model 1 - Attention method

Model Structure

Model_1 which contains SNLI_Train and SNLI_Test file is a complex implementation of SNLI Corpus based mainly on Attention concept. Attention, also known as Positional Encoding,

is a very powerful method in NLP enabling the machine to understand not only the meaning of each word but only the structure of the whole input sentence (Vaswani).

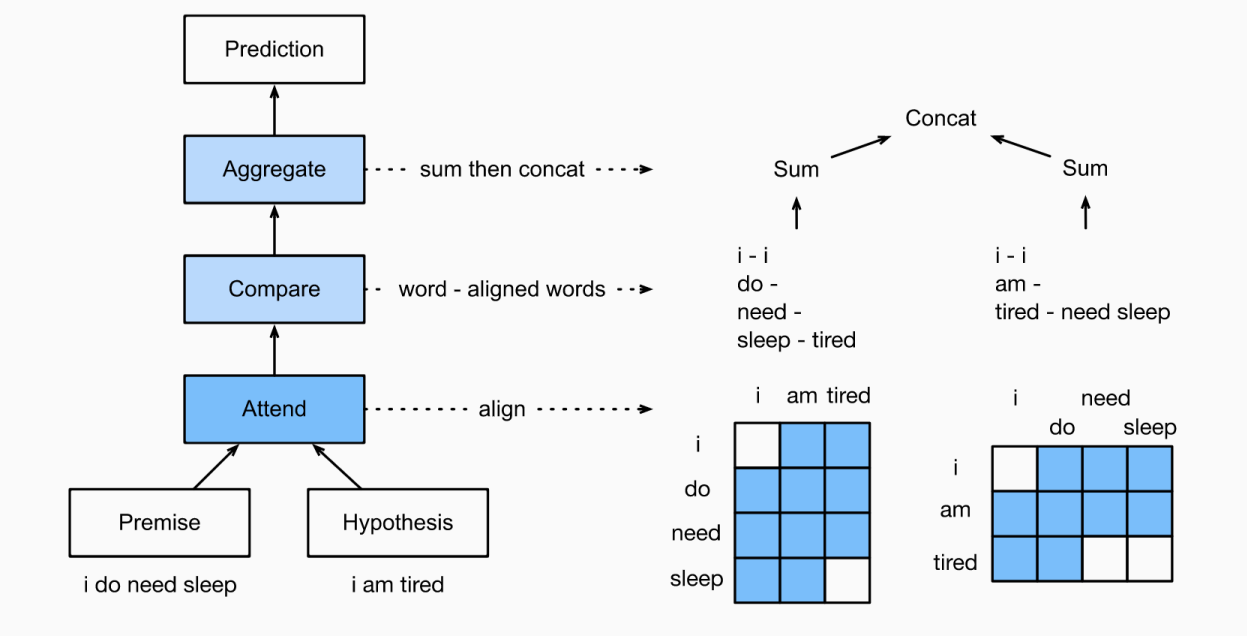


Figure 2. NLI using attention mechanisms (15.5. Natural Language Inference: Using Attention)

In general, the premise and hypothesis sentence will be fitted into our embedding layer. Before this fitting happens, we must transform both sentences for that.

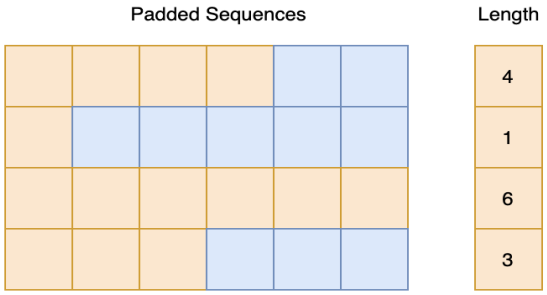


Figure 3. Padded Sequences

The process is called padding. The length of padded sequences is a fixed length which is defined in the model named max_length_sentence variable. Each marked light orange position in padded sequences represents an existing word vector.

Each marked light blue position is a blank represented by “0”. Furthermore, a mask sequence is constructed as well after padded sequences. This mask sequence has the same fixed length with padded sequences. In other words, they are corresponding to each other. Instead of storing word vectors, the mask will write “1” in each position the word existed, “0” otherwise. Then, the returning result will be used to build our embedding layer. The next layer, Attention layer will align words in one text sequence to each word in the other sequence. The attention mechanisms

that we are using here is called soft alignment. This type of alignment uses weighted average, where large weights are associated with the words to be aligned. Then, the model compares one sentence to its soft alignment with the other and aggregates the representations induced from both sentences and their representation. Finally, the model classifies the output labels.

The model will not perform training the whole dataset in epoch due to the large number of samples. In order to reduce the training time and improve in performance, we divided the whole dataset into sub-batch based on the idea of divide and conquer algorithm. This is kind of similar to ensemble learning but we only use one model while ensemble learning performs on different models. After that, a feed forward network is used to take concatenated word embedding and corresponding normalized alignment vector to generate the "comparison vector". The comparison vectors for each sentence are summed to create two aggregate comparison vectors representing each sentence which is then fed through another feed forward network for final classification.

Parameter tuning

In this tuning parameters process, we tried to implement and compare three main optimizers for compiling a Keras model: **Adam**, **Adadelta**, and **Adagrad**. **Adam** algorithm is computationally efficient, has little memory requirements, and well-suited for problems that are large in terms of data and/or parameters. **Adagrad** is an optimizer that is adapted relative to how frequently a parameter gets updated during training while **Adadelta** is a more robust extension of **Adagrad** that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients (Optimizers - Keras Documentation)

Train model 1 with the following configurations and parameters:

batch_size: ***500****, clip_value: 5.0, drop_out: 0.8, learning_rate:* ***0.1****, log: log/log.2020_04_04_01_36_46, max_length_sen1:* ***50****, max_length_sen2:* ***50****, num_class: 3, num_epoch: 200, optimizer:* **Adam**, *vocab_size: 56221*

=> Training result with optimizer **Adam**: No optimization for a long time, auto-stopping. Model training ends at epoch: 91 with a low accuracy of 33.31%. Time Taken: 22:55:54

We keep the same above configurations but change the optimizer to **Adadelta**

=> Training result with optimizer **Adadelta**: Finish the training dataset with 200 epoches, with a significant improvement to achieve the training accuracy of 75.97%. Time Taken: 2 days, 4:37:30.

The same above configurations but with optimizer: **Adagrad**

=> Training result with optimizer **Adagrad**: Finish the training model with 200 epoches, yield a training accuracy of 74.13%. Total training time taken: 3 days, 5:38:20.

With some different parameters values in clip_value, and learning_rate

batch_size: ***500****, clip_value:* ***4.0****, drop_out: 0.8, learning_rate:* ***0.01****, log: log/log.2020_04_06_14_16_51, max_length_sen1:* ***50****, max_length_sen2:* ***50****, num_class: 3, num_epoch: 200, optimizer:* **Adagrad**, *vocab_size: 56221*

=> Training result: achieves a higher accuracy of 75.25% at epoch 86.

The training model that yields that best result with 75.40% in training and 76.77% in testing is with the following parameters: *batch_size:* ***200****, clip_value: 5.0, drop_out: 0.8, learning_rate:* ***0.05****, log: log/log.2020_04_02_21_12_40, max_length_sen1:* ***100****, max_length_sen2:* ***100****, num_class: 3, num_epoch: 200, optimizer:* ***Adagrad****, vocab_size: 56221*

Performance Evaluation

Overall the training phrase, we perform at least five tuning hyper-parameters for a better training and testing accuracy. The optimizer is the main approach used today for training a deep learning or machine learning model in order to minimize its error rate. Despite the widely used **Adam** optimizer, it performs the worst with our default set of hyper-parameters. Recent research papers have noted that it can fail to converge to an optimal solution under specific settings. The SNLI_4_K model gets 74.84% on testing while the SNLI_3_K model performs better and has 76.16% accuracy on testing. Our best model is SNLI_0_G which outperforms SNLI_3_K a little bit and has 76.77% accuracy on prediction.

SNLI_0_G	SNLI_3_K	SNLI_4_K
Confusion Matrix... [[2866 293 209] [603 2168 448]	Confusion Matrix... [[2808 355 205] [593 2203 423]	Confusion Matrix... [[2880 337 151] [695 2186 338]

[350 379 2508]]	[356 410 2471]]	[475 476 2286]]

By observing the confusion matrix of three models, we realize that all models have trouble to predict label 1, which is Neutral. The F-1 score of three models in the testing report shows that the prediction on label 1 is the worst as well among the rest. The plot below is a summary of training and testing accuracy.

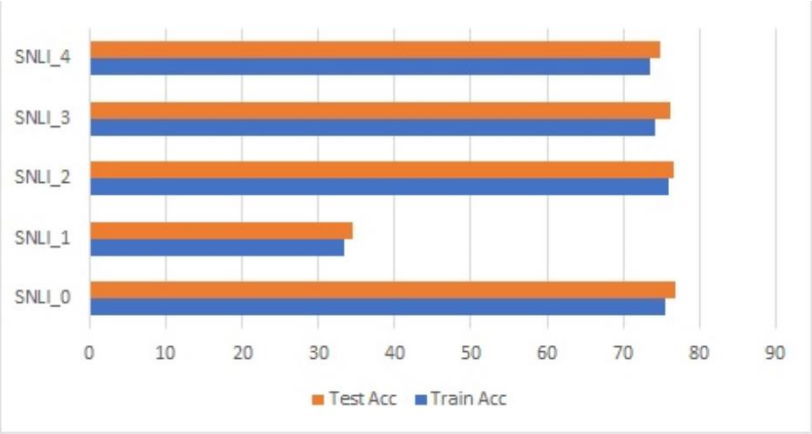
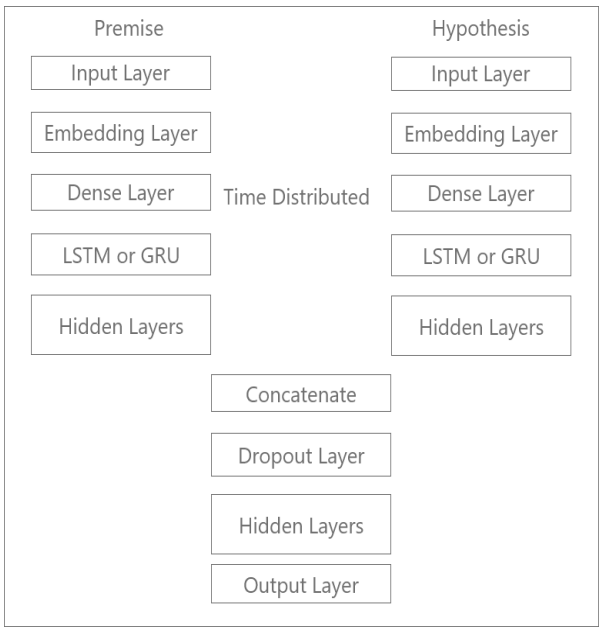


Figure 4. Overall Model 1's performance.

2. Model 2 - RNN (LSTM, GRU)

Model Structure



In this model, we are using the modern recurrent units which are GRU (gated recurrent unit) and LSTM (Long Short-Term Memory). These modern models are able to avoid vanishing and exploding gradients and learn long-term dependencies that a simple neural network cannot. Different from the first model, model 2 is trained by library Keras functional API. At first, we try to use the keras sequential model, however; because the dataset has 2 inputs (sentence 1 and sentence 2), Keras functional API is a better approach for multiple inputs

Figure 5. NLI using RNN with LSTM & GRU model.

Briefly explain the structural of model

From the last preprocessing milestone. We have the clean version of each dataset (train, test, and dev). However, in this model, we decided to drop all others but 3 columns: gold_label, sentence1, and sentence2. By using the pre-trained word embedding vectors GloVe, We're able to convert all the vocabulary that appear in the dataset into sequences of numbers that have the same lengths by using pad_sequences. At the starting point, we were using the default NN model in Keras Library for tuning the parameter. and then adding recurrent units such as LSTM or GRU after to explore the change in accuracy.

The model consists of 3 core layers: the input layer, hidden layers, and output layer. The input layers will be separately used for sentence 1 and sentence 2. The data will be processed through hidden layers (embedding, dense) and RNN layers (either LSTM or GRU) . The model then combines sentence 1 and sentence 2 in one layer (concatenate layer). However, this does not yield a high accuracy in the test dataset and causes overfitting. We added 3 more hidden layer groups each including a dense, dropout, and batch normalization layer.

Parameter tuning

*Different tuning values are detailed in parenthesis.

Enable RNN layers: None (recurrent.LSTM, recurrent.GRU) (recurrent library in Keras)

Train Embedding: False (a parameter in embedding layer. We set it to false because we don't want to modify the GloVe Weights)

GloVe Enable: True (False to not using the pre-trained vectors)

Embedding layer dimension/ hidden layer size: 300 (The GloVe file is 300-dimension version)

Optimizer: rmsprop (adagrad, adadelta, adam)

Max Sequence Length: 42 (the length is switched between 79 - maximum number of words in a sentence - and 42 - the median/mean of the number of words in a sentence)

Batch size: 512

patience: 4 (8)

Number of Epoch: 20 (each takes 20-30 minutes depending on the resource, so we pick 5 to tune LSTM and GRU and 20 for other tuning parameters)

Loss Function: Categorical Cross Entropy

Activation Function: Relu for hidden layers and Softmax for output layer

The tuning values above results in 80% in accuracy.

Performance Evaluation

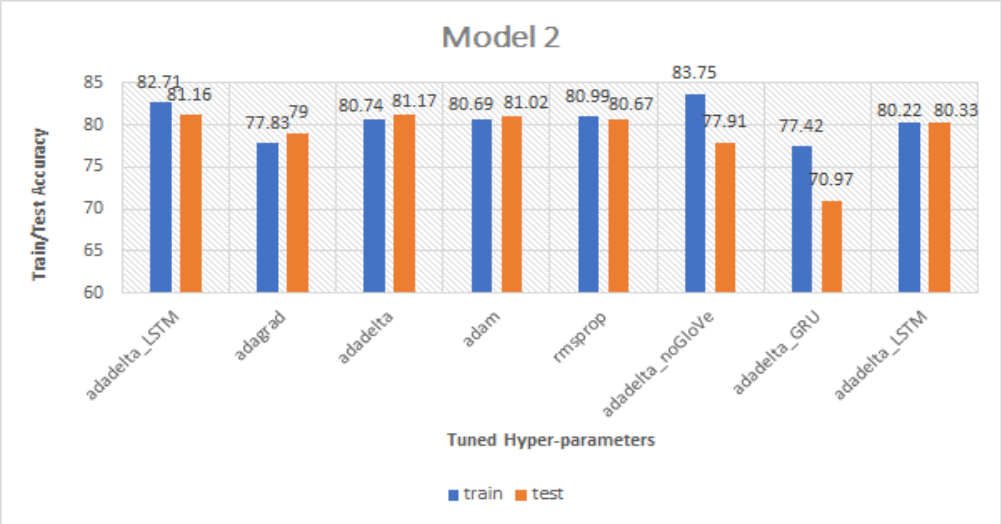


Figure 6. Overall Model 2's performance.

Optimizer: after 20 epochs, adadelta and adam yield a slightly higher accuracy (about 1-3%) than rmsprop and adam.

GloVe Enable: after 20 epochs, using GloVe results in a higher accuracy overall. Not using GloVe gives a higher accuracy in the train dataset only, and lower accuracy in the test set (about 7-10%) which is overfitting.

Max Sequence Length: after 20 epochs, 42-number sequence is performing better than 79-number sequence (77% to 80%). The number of 0 after pad sequence may be the noise causing the descent in accuracy.

RNN Unit: From the model, the LSTM unit performs better than the GRU and NN.

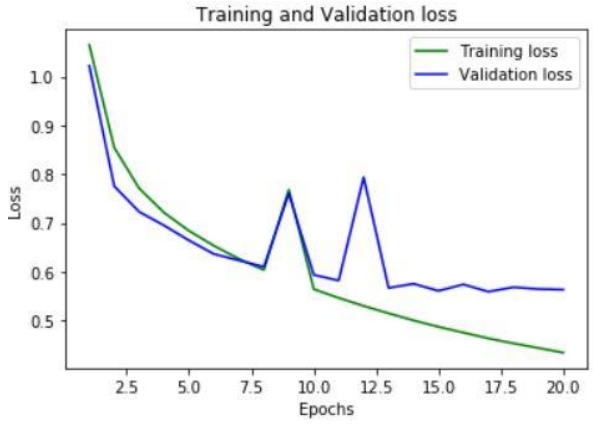


Figure 7. Training without pre-trained GlovVe model.

```
Precision, Recall and F1-score...
              precision    recall  f1-score   support

           0       0.87      0.79      0.83      3237
           1       0.79      0.75      0.77      3219
           2       0.78      0.88      0.83      3368

    accuracy                           0.81      9824
   macro avg       0.81      0.81      0.81      9824
weighted avg       0.81      0.81      0.81      9824

Confusion Matrix...
[[2570  354  313]
 [ 270 2429  520]
 [ 106  287 2975]]
Time usage: 1:13:36
```

Observing the F1-score and confusion matrix of this model, both first model and second model yields a similarity in result. It performs well on predicting label 0 (contradiction) and 2 (entailment), and does not on predicting label 1 (neutral).

Model Comparison

	Model 1	**Model 2**
Advantages	Model 1 is an implementation of a popular method- Attention method. Attention takes the advantage of not only using the word vector presentation but also the word's position in the whole sentence. Compared to model 2, model 1 fits for large scale dataset with high complexity such as the Translation Language model. Last but not least, model 1 performs pretty well without underfitting or overfitting due to the training and testing accuracy. Moreover, it's very convenient because model 1 exports its model after each training time.	By using the Keras functional API, the complexity and length of the code is shorter and simpler. Training time of each epoch take longer time to run but yield the effectiveness in accuracy The model is not underfit/overfit because training and testing acc is closed to each other, can also conclude this by observing plots in plot folder (each training have their train loss and validation loss closed to each other) Able to use the advantages of LSTM and GRU, support RNN (short term memory term). Model 2 is suitable for building sentiment analysis systems.

Disadvantages	By dividing the whole dataset into sub-batch and considering training them separately, model 1 consumes significantly a lot of time for training. Therefore, we need much more time for tuning hyper-parameters for better model's accuracy. The training time for model 1 is 15 days, 17:25:39.	The training time of model 2 will take longer than model 1 if both models train the same amount of epochs. Each training time of model 2 is shorter than model 1 because we limited the number of epochs to 20.

3. Conclusion:

In conclusion, both models perform very well. All hyper-parameters are tuned carefully to prevent underfitting and overfitting. Model 1 is run on testing data and received 76.77% accuracy while model 2 archives 81.17% accuracy. Both test scores satisfy us. By working more on tuning hyper-parameters, we believe that we could obtain higher accuracy on testing. We could bring the test accuracy up to around 80% and around 85% testing accuracy for model 2

References

"Module: Tf.keras.layers: TensorFlow Core v2.1.0." *TensorFlow*, www.tensorflow.org/api_docs/python/tf/keras/layers.

Optimizers - Keras Documentation, keras.io/optimizers/.

Pennington, Jeffrey, et al. *GloVe: Global Vectors for Word Representation*. 2003, nlp.stanford.edu/pubs/glove.pdf.

Pennington, Jeffrey. "GloVe: Global Vectors for Word Representation." *GloVe: Global Vectors for Word Representation*, 2014, nlp.stanford.edu/projects/glove/.

"Recurrent Neural Networks (RNN) with Keras: TensorFlow Core." *TensorFlow*, www.tensorflow.org/guide/keras/rnn.

"Text Classification with an RNN: TensorFlow Core." *TensorFlow*, www.tensorflow.org/tutorials/text/text_classification_rnn.

Vaswani, Ashish, et al. *Attention Is All You Need*. 2017, arxiv.org/pdf/1706.03762.pdf.

Venkatachalam, Mahendran. "Attention in Neural Networks." *Medium*, Towards Data Science, 7 July 2019, towardsdatascience.com/attention-in-neural-networks-e66920838742.

"Word Embeddings: TensorFlow Core." *TensorFlow*, www.tensorflow.org/tutorials/text/word_embeddings.