Recognizing Textual Entailment with PyTorch

Kevin Chen

Department of Computer Science

The University of Texas at Dallas

ning.chen@utdallas.edu

Abstract

This paper describes utilizing a deep learning model to recognize the textual entailment between two sentences. That is, the premise and the hypothesis are encoded and put through a neural network using a model that will be described in this paper. This output, in the training phase will help adjust the weights of the neural network to optimize the performance of the model. This model is then tested on another set of premises and hypothesis to evaluate the model.

Keywords: Neural network; Vocabulary; Premise; Hypothesis; Entailment; LSTM

1. Introduction

For this project, students were tasked with implementing a deep learning model which recognizes the textual entailment relation between two sentences; one of which is labeled the premise, one of which is labeled the hypothesis. Suppose that the premise is denoted t, while the hypothesis is denoted h. Entailment, denoted as $t \rightarrow h$, is defined as follows: t entails h if the meaning of h can be inferred from the meaning of t. [1] For example, consider some examples given below:

(1)

t: The wait time for a green card has risen from 21 months to 33 months in those same regions.

h: It takes longer to get green card.

$t \rightarrow h$ because the meaning of h can be inferred from the meaning of t.

(2)

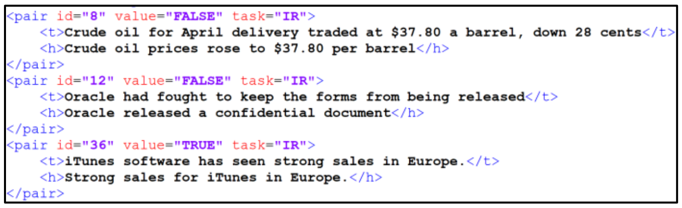
t: Oracle had fought to keep the forms from being released

h: Oracle released a confidential document

$\neg$ $(t \rightarrow h)$ because the meaning of h cannot be inferred from the meaning of t.

In the context of deep learning, the task of predicting textual entailment is set up as a binary classification problem where, given two input variables premise (*t*) and hypothesis (*h*), the expected output is *entails* or *not entails*.

To conduct this experiment, students are given two XML files, from the RTE-1 dataset: *train.xml* and *test.xml* [2]. An example of the data that is given in these files are illustrated in Figure 1. The `<t>` tag represents the premise, while the `<h>` tag represents the hypothesis. Within the `pair` tag is the property `value`, which represents the *entails* or *not entails* output. If the property is "`TRUE`", then *t* entails *h*, and if the property is "`FALSE`", then *t* does not entail *h*.



```xml
<pair id="8" value="FALSE" task="IR">
    <t>Crude oil for April delivery traded at $37.80 a barrel, down 28 cents</t>
    <h>Crude oil prices rose to $37.80 per barrel</h>
</pair>
<pair id="12" value="FALSE" task="IR">
    <t>Oracle had fought to keep the forms from being released</t>
    <h>Oracle released a confidential document</h>
</pair>
<pair id="36" value="TRUE" task="IR">
    <t>iTunes software has seen strong sales in Europe.</t>
    <h>Strong sales for iTunes in Europe.</h>
</pair>
```

Figure 1: A snippet of an XML file containing the premise, hypothesis and entails or not entails values data. This data is used to train and test the deep learning model, respectively.

2. Proposed Solution

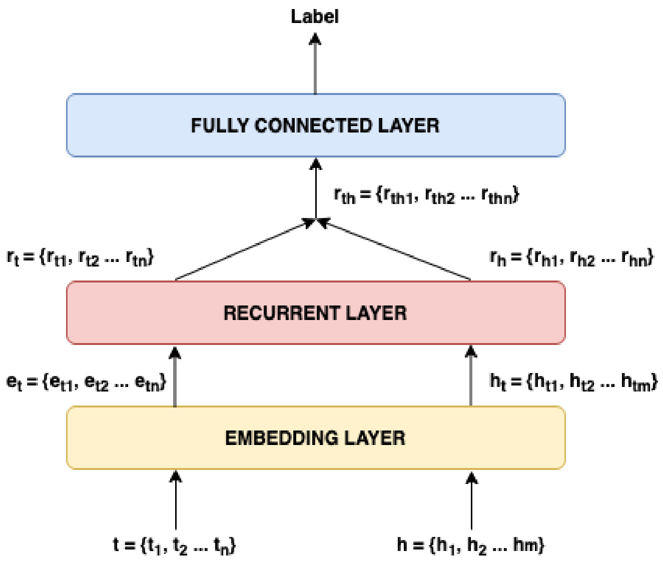

Label
FULLY CONNECTED LAYER
$r_{th} = \{r_{th1}, r_{th2} \dots r_{thn}\}$
$r_t = \{r_{t1}, r_{t2} \dots r_{tn}\}$
$r_h = \{r_{h1}, r_{h2} \dots r_{hn}\}$
RECURRENT LAYER
$e_t = \{e_{t1}, e_{t2} \dots e_{tn}\}$
$h_t = \{h_{t1}, h_{t2} \dots h_{tm}\}$
EMBEDDING LAYER
$t = \{t_1, t_2 \dots t_n\}$
$h = \{h_1, h_2 \dots hm\}$

Figure 2: A neural network architecture that will be used to predict entails or not entails (*label*) by making a forward pass with the premise (*t*) and hypothesis (*h*) through a series of layers.

Figure 2 provides a high-level overview of the model architecture, which will be implemented in the form of a neural network. Sub-sections describe the model and how it is implemented using Python and a popular machine learning library, PyTorch.

2.1. Embedding Layer

Let $t = \{t_1, t_2, \ldots, t_n\}$ denote the premise and $h = \{h_1, h_2, \ldots, h_n\}$ denote the hypothesis. The model first puts the integer encoding representations of both the premise and hypothesis through the embedding layer separately to get vectors e_t and e_h, the dense vector representations for t and h, respectively. Both vectors have dimension d_1. This is implemented in Python using the Embedding class within torch.nn in the PyTorch API.

2.2. Recurrent Layer

Let $e_t = \{e_{t_1}, e_{t_2}, \ldots, e_{t_n}\}$ denote the vector representation for the premise and $e_h = \{e_{h_1}, e_{h_2}, \ldots, e_{h_n}\}$ denote the vector representation for the hypothesis. These vector representations are both then passed through a recurrent layer, which is an LSTM layer, to get temporal sequences r_t and r_h, respectively for the premise and hypothesis, both of dimension d_2. The reason why the model uses an LSTM is because, in a traditional RNN architecture, there is the vanishing gradient problem, when gradients decrease as they get pushed back due to non-linearities. In Python, this recurrent layer is implemented using the LSTM class in the torch.nn within the PyTorch API.

2.3. Fully Connected Layer

Before the two temporal vectors r_t and r_h are passed into the fully connected layer, these two vector representations are concatenated together to obtain the vector r_{th}. Then, this concatenated vector r_{th} is passed through the fully connected layer to get vector f_{th}, of dimension 2, since there are two labels *entails* and *not entails*. This representation is then passed through the softmax function, to convert the previous vector f_{th} into a probability distribution, outputted as a vector p_{th}. Vector concatenation is accomplished using the built-in function torch.cat. The fully connected layer is implemented with the Linear class in torch.nn, which applies a linear transformation to the vectors. Finally, the softmax function is implemented by the Softmax class in torch.nn. All of these are within the PyTorch library.

3. Experimental Results

3.1. Setup

This project was implemented in Python, specifically utilizing the PyTorch library. The code was run on a combination of Google Colab and, when Colab crashed due to issues with sending to the GPU, used my own laptop which has an NVIDIA GeForce GTX 1650.

For both training and testing, I ended up using batches each of size 16. I used the CrossEntropyLoss within the torch.nn library for loss, and Adam from optim in PyTorch for the optimizer, for updating network weights. The learning rate is set at 0.001, because I found that setting the learning rate any higher will make the loss fluctuate to too high values, and any lower would not make the model go as low in terms of loss.

3.2. Observed Results

The neural network was tested with different combinations of number of LSTM layers, embedding dimension and hidden dimension. To make my observations as accurate as possible, I only changed one of these parameters at a time, to see the effect of changing each of these variables on model performance and throughput.

LSTM Layers	10	15	25
Model Performance (average)	Accuracy: 0.49975 Precision: 1.0 Recall: 0.49975 F1 score: 0.65807	Accuracy: 0.50025 Precision: 0.0 Recall: 0.0 F1 score: 0.0	Accuracy: 0.49529 Precision: 1.0 Recall: 0.49529 F1 score: 0.65484
Throughput (average)	0.03834 sec	0.06341 sec	0.10469 sec

Figure 3: Model performance and throughput for different number of LSTM layers. Note that hidden layer size and embedding vector size is constant at 256 and 500, respectively.

The observations made in Figure 3 with regards to the number of LSTM layers is as expected, in terms of throughput. Simply put, the more layers there are in the recurrent layer, the higher the throughput, and therefore the longer the model takes to both train and test. The throughput seems to almost double with each increment of LSTM layers. The model performance data, however, is basically useless; the model either outputs entails values with all 1's or with all 2's. Depending on which one it is, the precision is either 1.0 or 0.0. For this reason, I wouldn't really use this model performance to determine how the model performs. I kept the number of layers at 15, because it seems to provide the best in terms of balance of throughput and loss values.

Hidden Size	128	256	512
Model Performance (average)	Accuracy: 0.49752 Precision: 1.0 Recall: 0.49752 F1 score: 0.65677	Accuracy: 0.50025 Precision: 0.0 Recall: 0.0 F1 score: 0.0	Accuracy: 0.50868 Precision: 1.0 Recall: 0.50868 F1 score: 0.66278
Throughput (average)	0.07304 sec	0.06341 sec	0.14243 sec

Figure 4: Model performance and throughput for different number of hidden layers. Note that LSTM layer size and embedding vector size is constant at 15 and 500, respectively.

The observations made in Figure 4 with regards to hidden layer size is as expected in terms of throughput. The throughput increases with the hidden layer size increase; although, surprisingly for some reason, throughput on hidden layer size 256 is lower than 128. The higher the hidden layer size, the longer it takes for the model to be trained. Again, model performance here cannot really be determined at all; as stated before, the model only produces either all 1's or all 2's for all batches and thus any measure of performance is moot. Based off this data, I decided to stick with hidden layer size 256.

Embedding Size	250	500	750
Model Performance (average)	Accuracy: 0.50025 Precision: 0.0 Recall: 0.0 F1 score: 0.0	Accuracy: 0.50025 Precision: 0.0 Recall: 0.0 F1 score: 0.0	Accuracy: 0.50422 Precision: 1.0 Recall: 0.50422 F1 score: 0.65697
Throughput (average)	0.05364 sec	0.06312 sec	0.06344 sec

Figure 5: Model performance and throughput for different number of embedding layers. Note that hidden layer size and LSTM layer size is constant at 256 and 15, respectively.

In Figure 5, we see the effect of embedding size on both model performance and throughput. Surprisingly, the embedding size does not seem to significantly affect throughput. There is an increase in throughput with embedding size, but by a relatively negligible amount at higher embedding size. Again, there is no way to gauge model performance by any metric since the predictions are all the same; either all 1's or all 2's for every batch in the test data. Based off this data, I decided to switch the embedding size to 750, because it has a relatively negligible impact on model performance.

4. Conclusion

4.1. Thoughts on Observed Results

The results from the above section are quite disappointing. All the model performance data is almost useless, all because the model is predicting either all 1's or all 2's for every sentence in the batches. Precision values were either 1.0 or 0.0, depending on whether the output was all 1's or all 2's. Recall was either equal to accuracy or 0.0, also depending on the output. Accuracy was generally averaging around 0.5, likely because about half of the values in the test data was 1's and the other half 2's. Throughput, however, was interesting. Generally, I found embedding size to not significantly impact throughput, but hidden layer size and number of LSTM layers significantly impacted throughput. Generally, the more layers, the higher the throughput, which was not surprising, as the data must go through more processing and calculation through the layers.

4.2. Lessons Learned

Though most of the data didn't tell a whole lot about the quality of the neural network, there was a lot learned through constructing this model. Prior to this project, I had little experience with working with neural networks in Python and had never used PyTorch. Importing the data into Python and encoding the data from words into numbers was the easiest part of the project, though it took me a little bit longer to set up than I had expected. Setting up the neural network and getting it to work was the hardest part of this project. I initially couldn't figure out the difference between CPU and GPU tensors, but I eventually figured it out. My laptop was on Python 3.8, which kept having this weird error with GPU tensors, but I downgraded to Python 3.7 and it worked just fine. Most tutorials online were training models with only one input and one output, but for this project there are two inputs and one output, so I scrambled to try and figure out how to set up the model so it could take two inputs. I settled on creating a class, which made it a lot easier to put the parameters into the optimizer. Creating batches to train and test data was also foreign to me prior to this project, as I had never done it for my undergraduate

Machine Learning class. The loss and optimizer were not incredibly hard to setup, though I was not familiar with the concept of epochs. I found that putting the model through the GPU instead of the CPU significantly improved performance; one run through on the GPU took only about 3 minutes, while another run through with the same parameters took almost 44 minutes on the CPU. This project also taught me that, just because neural network takes longer to train and are more sophisticated, does not necessarily mean that it will perform better than a more simplistic model. It all depends on the input data, as well as the parameters given to the model and how the neural network model is set up. Based on the code, as I can find no fault in the code, I believe the unusual uniformity of the output is due to the small brevity of the data and not the model I used to train and test it. To summarize, other than the performance data, this project was an interesting insight into how to implement a deep learning model to determine *entails* or *not entails* for a sentence.

References

[1] Daniel Z Korman, Eric Mack, Jacob Jett, and Allen H Renear. Defining textual entailment. *Journal of the Association for Information Science and Technology*, 69(6):763-772, 2018.

[2] Ido Dagan, Oren Glickman, and Bernardo Magnini. The pascal recognising textual entailment challenge. In *Machine Learning Challenges Workshop*, pages 177-190. Springer, 2005.