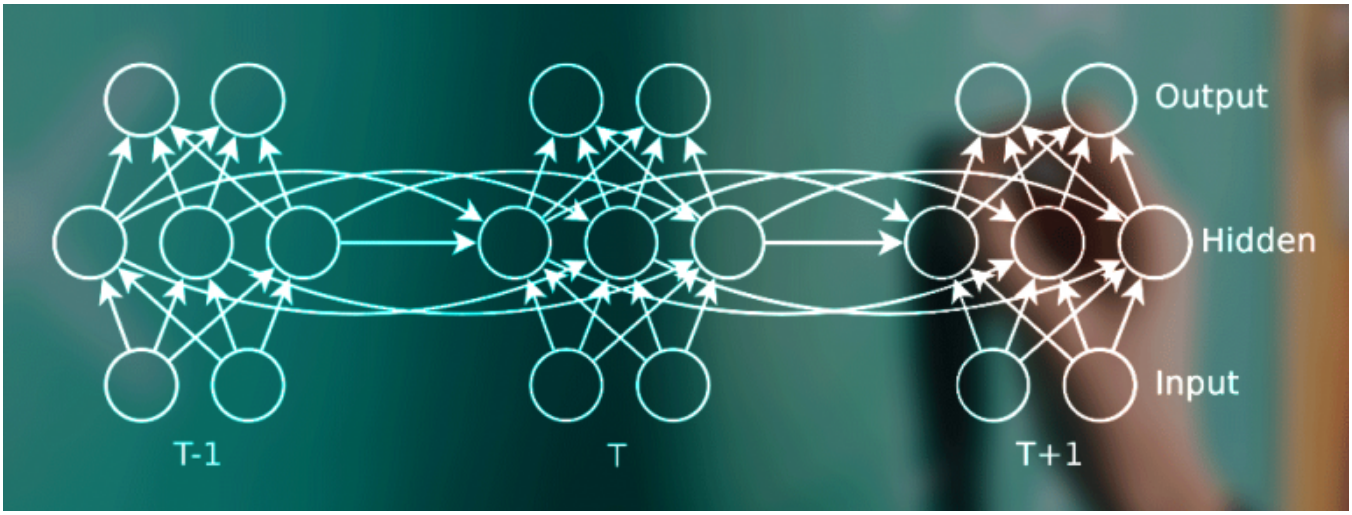
Textual Entailment Task

Gaëtan Bellière
University of Stavanger
Stavanger, Norway
g5bellie@enib.fr

Luc Castelain
University of Stavanger
Stavanger, Norway
l6castel@enib.fr

Lucas Bahuaud
University of Stavanger
Stavanger, Norway
l5bahuau@enib.fr

Thomas Favé
University of Stavanger
Stavanger, Norway
t5fave@enib.fr



ABSTRACT

As part of our studies at the University of Stavanger, we had the opportunity to realize a mini-project. The theme we chose for this project was a Natural Language Processing topic : Textual entailment task. Given two sentences, the goal is to classify whether the second sentence follows from the first sentence or if they are unrelated. And also, given one claim, being able to extract data from Wikipedia to support or refute the claim. For that we present multiple methods : Bi-LSTM architecture (with and without attention), Bi-GRU and BERT. LSTM and GRU are a neural network architecture when BERT is an open source machine learning framework for NLP. BERT is already pre-trained for textual entailment.

KEYWORDS

datasets, neural networks, textual entailment, LSTM, GRU, BERT, classification, NLP

ACM Reference Format:
Gaëtan Bellière, Luc Castelain, Lucas Bahuaud, and Thomas Favé. 2020. Textual Entailment Task. In *Proceedings of (Textual entailment task).* , 7 pages.

Textual entailment task, April 2020, Stavanger, Norway

.

1 INTRODUCTION

Textual entailment in natural language processing is a directional relation between text fragments. This relation holds the truth of one text fragment follows from another text. Textual entailment measures natural language understanding and it is an active area of research.

Many natural language processing applications, like question answering, information extraction, summarization, multi-document summarization, and evaluation of machine translation systems, need to recognize that a particular target meaning can be inferred from different text variants. Typically entailment is used as part of a larger system, for example in a prediction system to filter out trivial or obvious predictions.

This project focus on the creation of a textual entailment task. It will not approach any of the language processing applications above.

To complete this task, we work with three main methods, LSTM, GRU and BERT, on three different datasets, both of them have labels, which means that this is a supervised task. LSTM and GRU are a recurrent neural network architecture when BERT is an open source machine learning framework for natural language processing. Both of them are implemented with python so we can compare their efficiency.

2 DATA SETS

We have three datasets to work on. The first one is a dataset from a fake news classification challenge from Kaggle. The dataset is divided into two csv files : train and test. The train and test dataset contain respectively 320552 and 80126 rows. This dataset is unbalanced, 70 percent of the training pairs are unrelated. The English titles are machine translated from the Chinese titles related. Each row of the train dataset contains the following fields :

- id - the id of each news pair.
- tid1 - the id of the news title 1.
- tid2 - the id of the news title 2.
- title1_zh - the news title 1 in Chinese.
- title2_zh - the news title 2 in Chinese.
- title1_en - the news title 1 in English.
- title2_en - the news title 2 in English.
- label - indicates the relation between the news pair: agreed/-disagreed/unrelated.

The second dataset is from the FEVER (Fact Extraction and VERification) challenge. The train and test files are in jsonl and have respectively 145449 and 19998 rows. This file is a bit more balanced, with a repartition of training labels like : 55-25-20. Each row of the train file contains the following data fields:

- id -
- verifiable -
- label -
- claim -
- evidence - [Annotation ID, Evidence ID, Wikipedia URL, sentence ID]

The third dataset is The Stanford Natural Language Inference (SNLI) Corpus. This one is perfectly balanced, it contains more than 500000 rows. We only kept 3 columns from all those available :

- gold label -
- sentence1 -
- sentence2 -

3 PREPROCESSING

Before working on the raw data, the preprocessing is an important step in the data mining process. If there are too many irrelevant or redundant information in our dataset, then the training phase can be more difficult and take a considerable amount of time. In order to faster our algorithms we should do some preprocessing on our data, which means remove the noises from our data that are irrelevant or redundant.

3.1 Data cleaning

First, the data has to be cleaned from all kind of punctuation. If the data is from HTML/XML sources, we need to get rid of all the tags, HTML entities, punctuation, non-alphabets, and any other kind of characters which might not be a part of the language. The general methods of such cleaning involve regular expressions, which can be used to filter out most of the unwanted texts. We also remove unnecessary characters in the data to speed up the future executions of our algorithms.

The data also has to be put to lowercase, because if not done, the words "money" and "Money" won't have the same meaning for the model.

Then we need to deal with the "N/A" and "NaN" cells by just replacing their value to "undefined" in our case because there was problems with only some categorical values, so we didn't have to build another method to deal with missing numerical values.

It is also possible to remove what is called the stop words from the texts. The stop words are the most common words used in a language and they don't help to understand the meaning of a sentence, like "a", "the", "do", "can", etc... Removing these doesn't change the meaning of a sentence and also reduces the time computation of a model.

3.2 Tokenizer

In language processing, strings are not implicitly segmented on spaces as a natural language speaker would do. Each word in the raw input should be explicitly splitted into token. Tokenizer brand sections of a string of input characters. Appart of the global definition, tokenization is implemented differently depends of the used method. Each is detailled in *section 4 - LSTM Algorithm* and *section 5 - BERT*.

3.3 Pad_sequences

To compare two sentences in these models, the compared sentences need to have the same length. To ensure that all sequences of tokens have the same length we pad sequences to the same length using zero-padding. An argument is taken to set the sequences length. Sequences that are shorter than the taken argument are padded with value at the end or at the beginning. Sequences that are longer than the taken argument are truncated to fit the desired length.

4 NEURAL NETWORK MODELS

4.1 LSTM algorithm

LSTM is a Recurrent Neural Network, with the faculty to remember short and long term dependencies, it's therefore very useful in NLP. It is a complex cell containing multiple activation gates.

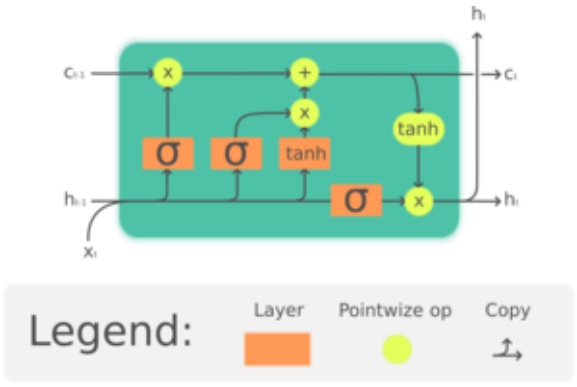


Figure 1: Inside LSTM cell

In order to compare the meaning of 2 sentences the models implemented were siamese LSTM. Two sentences can have the same

meaning without using the same words, or a word can have different meanings according to the context. This is where LSTM is useful, it "remembers" the context around a word to better understand the meaning of a sentence, furthermore, using a Bi-LSTM layer helps get the context of a word in reading the sentence forward and backward.

As mentioned earlier, it's impossible to feed a model with raw text data, so the text was tokenized using the method from Keras preprocessing. Each word is now a number, and each sentence a list of numbers. To feed this to a model every sequences need to have the same size, the size of the 99 percentile is 30, the sequences were zero padded, so that if they had less than 30 words it would add zeros at the end and if it was longer than 30 the last words were deleted. So a sequence input would look like something like : [156,1233,45,459,...,789,12]. For the labels the tokenization was different, the method used was the one hot encoding, so it would look like this : [1. 0. 0.],[0. 0. 1.], [0. 1. 0.].

4.2 GRU algorithm

GRU is very similar to LSTM from the outside, the inputs and outputs may differ but the key idea is the same. Nonetheless, the architectures inside are different. GRU has less gates, which often makes it faster to compute than LSTM.

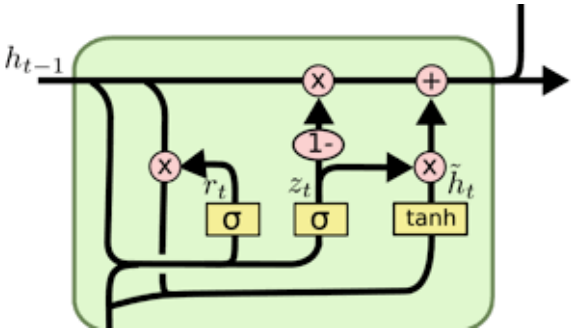


Figure 2: Inside GRU cell

4.3 Models architectures

Several architectures were implemented, with more or less hidden layers, in order to compare their performance. The hidden layers were Bi-LSTM or Bi-GRU. One architecture had an attention layer between the last bi-LSTM layer and the concatenation layer. An attention layer is used in NLP in order to give more importance to some words than others, and this way it gives an output based on the most relevant words and not on an entire sentence. Two different methods were tested for the embedding layer, either it was using a classic embedding layer from Keras layers with random weights or using a pre-trained embedding matrix based on GLOVE that has been developed by Stanford. The Glove.6B.100D.txt file was mainly used, with 6 billions tokens. Then, the two layers were concatenated, and finally we applied a dense layer with a softmax activation function to classify our sentences.

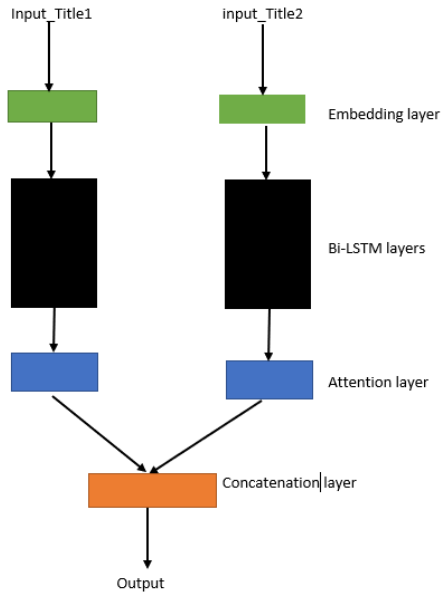


Figure 3: model architecture

4.4 Possible improvements

The first sumbission file that was posted on Kaggle received a 60 percent accuracy score, with tensorboard it improved to 73 percent. With the LSTM model on the Kaggle dataset, the best score that was reached is 85 percent accuracy on training and 73 percent accuracy on testing. We believed that one way to improve this score would be to use the Chinese titles instead of the English, because they were machine translated from the Chinese titles and the translation was a bit approximate making it difficult for the model to understand the meaning of the sentences. Even with Chinese text, LSTM and GRU were not able to get a higher score than English text.

In order to find the best model, multiples parameters had to be tested, like the number of units per layers, the dropout rate, the learning rate, the batch size, the cell type of the hidden state, the optimizer, using attention layer or not. To do so, we used Tensorboard, it is a tool that computes the model multiple times but changes one parameter every time, and then it gives the best parameters according to the metric that you're focusing on. It gave the following results for the Kaggle dataset :

embedding	learning_rate	optimizer	dropout	num_units	accuracy	f1_macro	loss	precisi
from_scrat...	0.00300...	rmsprop	0.10000	100.00	0.71588	0.71214	0.559...	0.7073
from_scrat...	0.00100...	rmsprop	0.30000	125.00	0.75087	0.75057	0.559...	0.7498
pretrained	0.00100...	rmsprop	0.10000	125.00	0.73687	0.73683	0.561...	0.7360
from_scrat...	0.00300...	rmsprop	0.30000	100.00	0.72237	0.71818	0.569...	0.7140

Figure 4: tensorboard testing multiples parameters

The focus was put on the loss function, we can see a first pattern, the optimizer rmsprop occupies the first places, the other tested was adam, we can also see that the embedding layer gives better

result from scratch that the one using the embedded matrix. We have to be careful with these results, because since comparing multiple parameters was time consuming, the computation using tensorboard were made on a smaller training set. When a parameter seemed to be better for multiple metrics (accuracy, loss, precision, recall..) it was fixed. Then it was possible to compute tensorboard on the whole training set for the parameters that were giving similar results. In the end, the best implementation gave these parameters as the best (for the Kaggle once again) : Once the best parameters

Show Metrics	dropout	num_units	optimizer	accuracy	f1_macro	loss	precision	recall
☐	0.300...	125.00	rmsprop	0.69266	0.69169	0.61598	0.69012	0.69357
☐	0.100...	100.00	rmsprop	0.73059	0.72879	0.57837	0.72607	0.73226
☐	0.100...	125.00	adam	0.68896	0.68746	0.67032	0.68542	0.69035
☐	0.300...	100.00	rmsprop	0.67839	0.67847	0.71723	0.67839	0.67839
☐	0.100...	100.00	adam	0.71456	0.71410	0.60289	0.71298	0.71531
☐	0.300...	100.00	adam	0.71281	0.71190	0.61448	0.71030	0.71378
☐	0.100...	125.00	rmsprop	0.69097	0.69005	0.66448	0.68837	0.69183
☐	0.300...	125.00	adam	0.72936	0.72928	0.60058	0.72863	0.72996

Figure 5: final tensorboard run

were found, we performed a training on more epochs, using an early stopping callback to avoid overfitting, and finally we could evaluate the model on testing data to measure accuracy and other metrics. An other possibility to improve the model accuracy would be to use an embedding matrix based on a glove file with more words, we used the smaller one the with 6 billions tokens, but with the 840B file, we might get a better accuracy score because the model would be able to handle way more words. On the SNLI model, some research papers claim a 85 percent accuracy on training and 77 percent accuracy on testing, with a simple LSTM layer of a 100 units, but using the 840B tokens file.

4.5 Limitations

LSTM and GRU models have a hard time to make good predictions with a training set that has one label over represented, this is the case for the Kaggle dataset, almost 70 percent of the records have the label "unrelated", this is one of the reasons why LSTM models could hardly have a better accuracy on predictions than 73 percent. But even with a well balanced dataset like SNLI, we didn't manage to have a better accuracy than 75 percent on training and 70 percent on testing. Some paper research claim 84 percent accuracy on training and 79 percent accuracy on testing, while using the 840B glove file, when we used the 6B glove file.

5 BERT

5.1 Description

BERT was released in 2018 by Google and has soon become a milestone in the NLP field by presenting state-of-the-art results in a wide variety of tasks. This breakthrough was the result of Google research on transformers: models that process words in relation to all the other words in a sentence, rather than one-by-one in order. It was mostly created to understand deeper the intent behind user's queries in Google Search.

BERT is designed to pre-train deep bidirectional representations from unlabeled text by processing on both side of the sentence at each layer. To be able to do that, BERT has been trained with two different methods : random word masking and next sentence prediction. We won't explain these methods here but they are well detailed in the author's paper [2]. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks without having to handcraft many more features.

5.2 Model

For our project we used a BERT pre-trained model, added an untrained classifier layer and finally trained the whole model with our labelled dataset. Using a pre-trained model shows some advantages compared to using a model from scratch:

- It requires less data. Indeed as the model is already trained, you don't need too much data to train it for a specific task. As we had a lot of data it is not the main point for us.
- Thanks to the pre-training, the model contains a lot of information about the requested language, in other words, the first layers of the model are already well set and just need to be tuned a little to match the expectations. It means that we don't need a lot of epochs to fit our model to the task, no more than 4 epochs are recommended.
- Finally it has shown really good results for a lot of different task in NLP, [insert examples]

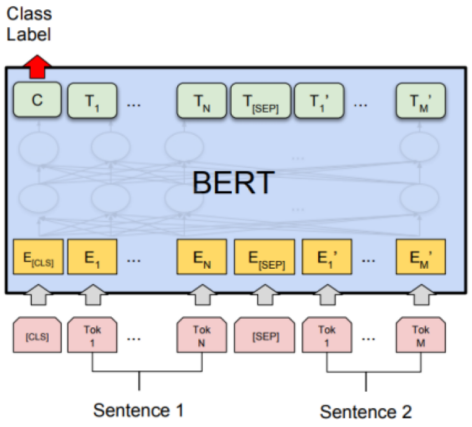


Figure 6: structure with tokens

We first used the "bert-base-uncased" model which is defined as : 12-layer (transformers), 768-hidden, 12-heads, 110M parameters, trained on lower-cased English text. We took this model at first so we could use the English sentences and make the whole process more understandable to us. But later on we switched the model to "bert-base-chinese" which is the same model but trained on Chinese texts. This one has shown better results (+0.04 on Kaggle score), we assume that it is because the original dataset is in Chinese and the English texts are just translations from the Chinese ones, which include some mistakes and non-sense that makes the first model harder to train because it's already trained on proper English texts.

5.3 Input Formatting

5.3.1 Tokenization. Sentences need to be modified so they can be well interpreted by the model, to do that special tokens need to be put in the sentence :

CLS : first element of the input, it needs to be put before the first sentence. It will store the information about the sentence(s) and be taken as the output of the last layer of BERT sequence before classification.

SEP : separation token that needs to be put at the end of each sentence (even if there is only one sentence). Helps the model to distinguish the different sentences in the input.

5.3.2 Embedding. BERT requires embeddings as input which are built from the tokenized sentences. It needs 3 layers of embedding to provide important metadata:

- token embedding : layer that matches each word token or subword-token to its id in the dictionary.
- sentence embedding : says for each token if it belongs to Sentence 1 or Sentence 2
- positional embedding : keep track of the position of each token in the sentence

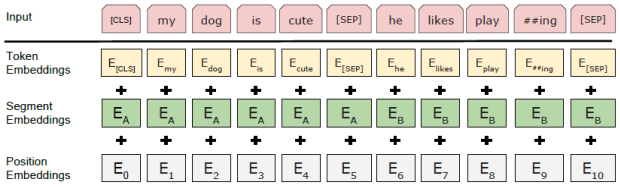


Figure 7: example of sentence and embedding layers

To compute all these prerequisite the "Transformers" library provides a class BertTokenizer that will encode our sentences to the correct form and provide the embedding layers needed to feed BERT. To use the encoding method from that class you need to provide different piece of information:

- Sentence1.
- Sentence2
- max sequence length : this will truncate the input sequence if it's longer than the value or add [PAD] tokens if its length is lower than the value.

The max sequence length will directly impact the running time so it has to be wisely defined. Our approach was to compute the mean of every pair of sentences length and compare it with the median to be sure that we don't get influenced by possible outliers. We picked a value a bit higher than the mean because we have to think of the CLS and SEP tokens, plus when encoding words the tokenizer actually split the words he does not know into subwords which increases the total sequence length.

Example : playing -> [play, ##ing]

5.4 Model Training

Transformers library provides a lot of classes to implement BERT for specific tasks. For that project we used BertForSequenceClassification which, as mentioned before, build a model based on BERT

Table 1: Training speed per epoch

Batch size	Learning rate	Time/epoch
64	5e-5	35min
128	1e-3	25min

(12 transformer layers) plus an extra layer on top for classification. To speed up the training we load the model on a GPU. Each of the 12 transformer layer is composed of an Attention Layer and a Dense layer (feed-forward layer) which are then merged into a final Output layer. Each layer is initialized with dropout to 0.1 to avoid overfitting issues. The Attention layers here are keys for our task because it's what gives sense to the words, a particular meaning in a sentence. So, having those layers at each step of the feed forward computation increases the ability of the algorithm to understand the input sentences and give a better classification. Attention mechanism in BERT is well explained in the paper [7]

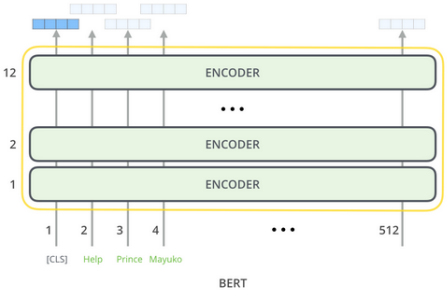


Figure 8: model architecture

As an Optimizer for the model, we first used Adam which we saw in the lectures and which is recommended by BERT authors. But later on we discovered that a new Optimizer was used to speed up BERT initial training time from 3 days to 76 minutes. It's called LAMB optimizer standing for 'Layer-wise Adaptive Moments optimizer for Batch training'. This optimizer allows us to increase the batch size and the learning rate for our algorithm which speeds it up greatly without losing in performance. Indeed increasing the learning rate speeds up the training time of an algorithm but it can result in the training being more unstable (gradient get stuck in a local minimum). LAMB handles that problem better than Adam and helped us to increase our performance. We complemented the optimizer with a linear scheduler to update the learning rate at each step.

During the training loop we take care to clip the gradient to a maximum value at each step so we prevent exploding gradient problem, and to clear out the previous gradient before performing backward propagation. If we don't, the gradients will accumulate and misguide the gradient descent.

We keep track of some metrics (accuracy, precision, loss) at each step and print the result for each epoch.

Table 2: Results for different number of epochs

N° epochs	Kaggle score
2	0.82464
4	0.81946

5.5 Parameters tuning

Once the model was well set-up and working, the principal task was to test and find optimal parameters. In our case what we could change (hyper-parameters) were: the batch size, the learning rate, number of epoch. In the previous section the method to select the batch size and the learning rate was explained. Concerning the number of epoch, thanks to the pretraining and the use of a good embedding , it doesn't need to be high. In fact we discovered than no more than 1 or 2 epochs were needed in our case.

The table above shows that for a greater number of epoch we had a lower result on Kaggle (corresponding to our testing dataset), which means that we were overfitting. Indeed we could see that the accuracy during the training was going up as the number of epochs was increasing but ultimately was lower on the testing.

5.6 Results

Once the training is complete, we just run the model with our testing data and put our results in a csv file so we can put it on Kaggle and see the final score. Our best shot gave us a private score of 0.86509 which would have placed us to the 24th place on the leaderboard (out of 95 teams). To do so we used a batch size of 128 with a learning rate of 1e-3 and trained the model for 1 epoch.

In order to verify the robustness of the algorithm we tried it on the SNLI dataset and without having to modify much of it we had pretty decent results. With the same hyper-parameters as told above we reached a 88 percent accuracy on training. It can probably get much better but it seems to be a nice start and proves the efficiency of the model without much fine tuning.

5.7 Possible improvements

It could be great to make a good tensorboard to be able to find the best hyper-parameters configuration to optimize our task. We went step by step to select best values for our features but we maybe missed some strong combinations. Also we found out later that an optimized version of Bert was created RoBERTa, it builds on BERT and modifies key hyperparameters, removing the next-sentence pretraining objective and training with much larger mini-batches and learning rates. So we would have like to test it and compare the results that we had with BERT for our datasets.

5.8 Limitations

BERT algorithm takes some time to train, even if it doesn't require much epochs it's a 12+1 layers algorithm with a big embedding. A good GPU is key to optimize time spent during the training phase, as we worked on Google Colab we had to deal with the GPU that was given to us. Sometimes it was great and we could make it to 30 min/epoch but it could go up to more than 1 hour per epoch with other GPU's. Sadly we found out near the end of the project that we could run our code on the Kaggle platform that allocates everytime the best GPU that Google Colab could give us. Also it comes as a full package so it's important to first understand well what it requires to work efficiently and if needed to modify a bit its structure.

6 OVERALL PROBLEMS ENCOUNTERED

During this project we had a hard time to use a good framework to implement BERT, we had lots of version issues with our libraries etc... But then we were told to work on Google Colab which helped a lot with those issues, it also helped by giving us greater GPU's than the ones on our machines, this made us win a great amount of time as the algorithm is time consuming.

We struggled with the fever dataset. The pairs sentences were a claim (that was fully given), and evidences from Wikipedia articles, but they were not given, we had a wikipedia url and the number of the line where the sentence was. We didn't manage to retrieve efficiently those sentences, some attempts were made but the estimated time to retrieve every sentences was 1000 hours. With those pairs sentences, we would have been able to train our models and compare their efficiency on a new dataset. We tried two approaches, one using wikipedia's api for python and the other using wikipedia's jsonlines dump. None of these approaches get to the end, so we didn't manage to work on the dataset fever, this is why instead we used the SNLI dataset.

Most of the problem we had came from a lack of experience or knowledge from us, so it was a good experience to search for solutions and discover new tools.

7 CONCLUSION

This project was a great first experience for us in the NLP field. We had some hard times with the use of new techniques, libraries etc.. But we managed to get some good results, even though we are aware that it could be better. We have spotted some interesting points to work on to make the models more accurate.

8 TASKS SHARING

Thomas worked on BERT and on the Kaggle dataset. Gaëtan and Luc implemented LSTM, both worked on FEVER dataset but one used the wikipedia's api to extract the sentences when the other used jsonlines wikipedia's dump. As we were struggling on the FEVER dataset, Gaëtan found the SNLI Corpus dataset and adapted it for our task. In the mean time, Luc tried to get the FEVER dataset to work. Lucas worked on the GRU method. After testing LSTM on SNLI Corpus dataset, Gaëtan helped Lucas on GRU. Thomas also worked on SNLI to make it works and to test it on BERT. We shared a lot of our work, firstly to help everyone to understand the maximum of each method, and secondly to help each other. We were sharing a lot of code to help to go further in all methods. Three of us were living together, it helped a lot to work together and there was a lot of communication by message and often calls with the last one until the end of the quarantine. Working like that allowed us to improve some methods and if someone was struggling we could try to help him, we were not confined in our only task. For every methods, it was a lot of understanding, improving and debugging little by little. We all worked together on the report. Each one wrote more precisely on the method he worked on.

9 BIBLIOGRAPHY

REFERENCES

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova (2019) *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, Google AI Language. DOI: https://arxiv.org/pdf/1810.04805.pdf .

[2] Leslie N. Smith (2018) *A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay*, US Naval Research Laboratory, Washington, DC, USA. DOI: https://arxiv.org/pdf/1803.09820.pdf .

[3] Rani Horev" (10th November 2018) *BERT Explained: State of the art language model for NLP*. In: *Medium*. DOI: https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270 .

[4] Dima Shulga (5th June 2019) *BERT to the rescue!*. In: *Medium*. DOI: https://towardsdatascience.com/bert-to-the-rescue-17671379687f .

[5] Chris McCormick and Nick Ryan (20th March 2020) *BERT Fine-Tuning Tutorial with PyTorch*. URL: http://mccormickml.com/2019/07/22/BERT-fine-tuning/history .

[6] Keita Kurita (7th January 2019) *Paper Dissected: "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" Explained*. URL: http://mlexplained.com/2019/01/07/paper-dissected-bert-pre-training-of-deep-bidirectional-transformers-for-language-understanding-explained/ .

[7] Kevin Clark, Urvashi Khandelwal, Omer Levy, Christopher D. Manning (2019) *What Does BERT Look At? An Analysis of BERT's Attention*, Computer Science Department, Stanford University, Facebook AI Research. DOI: https://arxiv.org/pdf/1906.04341.pdf .

[8] Jeffrey Pennington, Richard Socher, Christopher D. Manning (October 2015) *GloVe: Global Vectors for Word Representation*. URL: https://nlp.stanford.edu/projects/glove/?fbclid=IwAR2e-3sd5hEqDkbTOcJz1w3E5-kCB69bHgNVO8pRnn_IdeQFx5L1WxpfKrM .

[9] The Stanford NLP Group (2015) *The Stanford Natural Language Inference (SNLI) Corpus*. URL: https://nlp.stanford.edu/projects/snli/?fbclid=IwAR1ATCthjRfuZ-s2bOHlXn9hDEcAq4NG0OTff5Y3WjnK9iQktDWIYuSTM4k .

[10] Edward Ma (13th Octobe 2018) *Learning sentence embeddings by Natural Language Inference*. In: *Medium*. DOI: https://towardsdatascience.com/learning-sentence-embeddings-by-natural-language-inference-a50b4661a0b8 .

[11] Michael Phi (24th September 2018) *Illustrated Guide to LSTM's and GRU's: A step by step explanation*. In: *Medium*. DOI: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21 .

[12] Likejazz (22nd March 2018) *Siamese-LSTM*. URL: https://github.com/likejazz/Siamese-LSTM/blob/master/train.py?fbclid=IwAR311zFW_1WZ0eUwPF3 .

[13] François Chollet (4th December 2020) *The Sequential model*. URL: https://keras.io/guides/sequential_model/?fbclid=IwAR1fDcTH-P2O0VRdWByYe9kAIsIYY9-gqh7sVmNWS6Ot7utIeb_Bgo3MB1M .

[14] Steven Hewitt (17th July 2017) *Textual entailment with TensorFlow Using neural networks to explore natural language*. In: O'reilly Media. DOI: https://www.oreilly.com/content/textual-entailment-with-tensorflow/?fbclid=IwAR1B3Di2fATeMxd9460hofU0yqqMDDogw0liT-jslx5VMjjR6zNI889c30Y .

[15] Juan Pablo Mejia (30th August 2018) *The Ultimate Guide to Recurrent Neural Networks (RNN)*. In: *SuperDataScience*. DOI: https://www.superdatascience.com/blogs/the-ultimate-guide-to-recurrent-neural-networks-rnn .