Natural language inference

José Ángel García Sánchez[1], Sarra Ben Yahia[1], Amande Edo[1], Giulia Dall'Omo[1]

[1*]Master of Science MoSEF Data Science, École d'Économie de la Sorbonne, Université Paris 1 Panthéon-Sorbonne, 112 Boulevard de l'Hôpital, Paris, 75013, Ile-de-France, France.

Abstract

This paper describes our work on a Natural Language Inference (NLI) project, where the goal is to predict the relationship between two given statements, whether they entail each other, contradict each other, or are neutral. As part of the Advanced Machine Learning course, we were guided by Professor Grégory Futhazar in exploring Deep Learning models for this task. Specifically, we aimed to compare the performance of a basic Deep Learning model with pre-trained transformer-based language models. Our experiments were conducted on the Stanford Natural Language Inference (SNLI) dataset, a large-scale benchmark for NLI tasks. Although our model did not outperform the state-of-the-art, we provide a detailed analysis of our approach and highlight areas for further improvement. Our work contributes to the field of Natural Language Processing and demonstrates the potential of Deep Learning models for NLI tasks.

Keywords: NLI, Transformers, Hugging Face, AI, Deep Learning, Machine Learning

Contents

1 Introduction

1.1 Problem presentation

In this paper, the problem we are addressing is a **Natural Language Inference** problem. As outlined in abstract, Natural Language Inference involves reading a pair of sentences and judging the **relationship** between their meanings, such as **entailment**, **neutral** and **contradiction**. Base on the Stanford Natural Language Inference (SNLI) dataset, we will trying to evaluate the relationship between a pair of **premises** and **hypotheses** within the dataset.

1.2 Database presentation

The database used in this paper is the **Stanford Natural Language Inference (SNLI)**. The SNLI corpus (version 1.0) is a dataset consisting of **570000 english sentence pairs** that have been manually labeled for balanced classification with three labels : Entailment, Contradiction, Neutral.
The dataset supports the task of natural language inference (NLI), which is also known as recognizing textual entailment (RTE).

The dataset comprises **english sentences** from two sources : user-generated text on the website Flickr, and text written by crowdworkers from Amazon Mechanical Turk.

Here is an example of how the database is constructed:

```
'premise': 'Children smiling and waving at camera',
'hypothesis': 'The kids are frowning',
'label': 2
```

Each instance in the database consists of :

- A **premise** : a string used to determine the truthfulness of the hypothesis.

- An **hypothesis** : a string that may be true, false, or whose truth conditions may not be knowable when compared to the premise.

- A **label** : an integer whose value may be either **0, 1 or 2** indicating respectively that the hypothesis neither entail or that indicating that the hypothesis and the premise is neutral nor that they contradict each other.
  Also, **observations which don't have any label are marked with -1**.

Furthermore, it is important to note that **each premise may appear three times in the database**, each with a different hypothesis and label.
When constructing the train, test, and validation split, it is important to ensure that the same premise does not appear in multiple subsets, even if the hypotheses differ. This is because such an overlap would lead to data leakage and can potentially inflate

the accuracy of the model. Additionally, it is essential to maintain the same label distribution across all subsets to ensure that they are representative of the entire dataset. This will allow for a fair evaluation of the model's performance on unseen data.

Here we will make an analysis of the labels distribution.

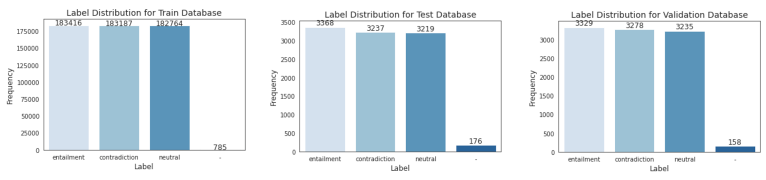

Fig. 1 Labels Distribution

From the graph above, that represent the labels distribution in each type of database, we can see that the distribution is almost the same.

The dataset is composed of three samples :

- A **train sample** composed of **550152 rows** and three columns (as mentioned before)
- A **test sample** composed of **10000 rows** and three columns as in the test sample
- A **validation sample** composed of **10000 rows** and three columns as in the test and train samples

We will now present the specificity of each samples in terms of vocabulary.
The **number of words in the vocabulary** of the different sample is:

- **40436 words** (36590 unique vocabulary words) in the train sample
- **6685 words** (6578 unique vocabulary words) in the test sample
- **6580 words** (6454 unique vocabulary words) in the validation sample

The **average number of words in a sentence** is respectively for each sample :

- **20 words in average** in the train sample sentences
- **21 words in average** in the test sample sentences
- **22 words in average** in the validation sample sentences

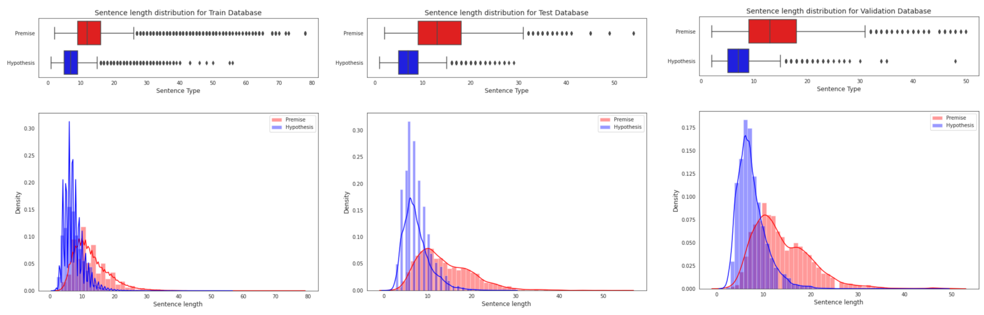

Fig. 2 Sentence length

Through the graph above, we can see that the premises are generally longer than the hypothesis.

2 Presentation of the models implemented

2.1 LSTM from scratch Architecture

In this project, we aim to develop a Long Short-term Memory (LSTM) model from scratch. The main idea behind this model is to incorporate a memory concept that enables it to learn from data while keeping track of both short and long-term dependencies. While there are already well-established LSTM models in the field, such as the 450D DR-BiLSTM Ensemble which is considered the best for the SNLI dataset and ranked 17 on the paperswithcode leaderboard, our goal is to develop a Naive LSTM that outperforms the hazard metrics. Specifically, we aim to improve the accuracy, which is currently at 0.33 due to the three possible labels.

2.1.1 Why LSTM?

LSTM (Long Short-term Memory) networks are designed to address the issue of long-term dependency in recurrent neural networks (RNNs) caused by the vanishing gradient problem. In traditional RNNs, during each iteration of training, each neural network weight receives a proportionally small partial derivative of the error function with respect to the current weight. In some cases, this gradient is so small that it prevents the weight from changing its value, which can hinder further training of the neural network. For example, when applying the tanh activation function, whose gradient is between (0, 1], backpropagation computes gradients using the chain rule, resulting in multiplying many of these small numbers to compute gradients for the early layers of the network, which can cause the gradient to decrease exponentially.

To overcome this problem, LSTMs use a gating mechanism to regulate the flow of information through the network. This gating mechanism enables LSTMs to selectively store, read, or forget information in their cell states, allowing them to learn and retain long-term dependencies more effectively than traditional RNNs. As a result, LSTMs have become a popular choice for many applications where long-term dependencies are important, such as language translation, speech recognition, image captioning and for our case NLI. LSTMs have been shown to be effective in NLI tasks because they can capture the complex semantic relationships between words and phrases in both the premise and hypothesis, allowing them to learn and reason about the relationships between them.

2.1.2 How LSTM works?

The LSTM architecture is composed of several gates that regulate the flow of information through the network. These gates are implemented using mathematical operations that selectively control how information is processed and passed through the network.

At each time step t, the LSTM cell has three inputs: the current input x_t, the previous hidden state h_{t-1}, and the previous cell state c_{t-1}. The LSTM cell has four main components: the forget gate, the input gate, the cell state, and the output gate.

1. Forget Gate
   The forget gate controls how much of the previous cell state should be retained and how much should be discarded. The forget gate is implemented using a sigmoid function that outputs a value between 0 and 1. The mathematical equation for the forget gate at time step t is:

   $$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

   where σ is the sigmoid function, W_f is the weight matrix for the forget gate, b_f is the bias vector for the forget gate, and $[h_{t-1}, x_t]$ represents the concatenation of the previous hidden state and the current input.

2. Input Gate
   The input gate controls how much of the new information should be added to the cell state. The input gate is implemented using a sigmoid function and a hyperbolic tangent (tanh) function. The sigmoid function determines which values should be updated, while the tanh function creates a vector of new candidate values that could be added to the cell state. The mathematical equation for the input gate at time step t is:

   $$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{c}t &= \tanh(W_c \cdot [ht-1, x_t] + b_c) \end{aligned}$$

   where i_t is the input gate, W_i and b_i are the weight matrix and bias vector for the input gate, respectively. $\tilde{c}_t$ is the candidate cell state, and W_c and b_c are the weight matrix and bias vector for computing the candidate cell state.

3. Cell State
   The cell state stores the long-term memory of the network. The cell state is updated using the forget gate and the input gate. The forget gate determines which parts of the previous cell state should be discarded, and the input gate determines which parts of the candidate cell state should be added to the current cell state. The mathematical equation for the cell state at time step t is:

   $$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t$$

4. Output Gate
   The output gate determines which parts of the current cell state should be outputted as the hidden state. The output gate is implemented using a sigmoid function and a hyperbolic tangent (tanh) function. The sigmoid function determines which

values should be outputted, and the tanh function applies a non-linear transformation to the cell state to create the hidden state. The mathematical equation for the output gate at time step t is:

$$
\begin{aligned}
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
h_t &= o_t \cdot \tanh(c_t)
\end{aligned}
$$

where o_t is the output gate, W_o and b_o are the weight matrix and bias vector for the output gate, respectively. h_t is the hidden state outputted by the LSTM cell.

The LSTM architecture is:

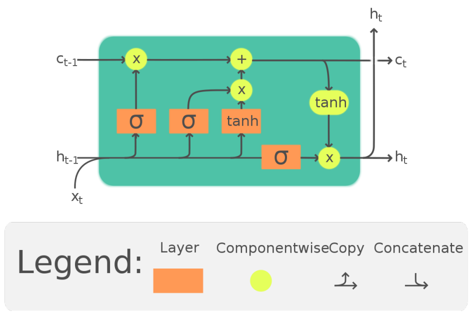

Fig. 3 LSTM architecture

The limited capacity for complexity due to long training times is a significant limitation of LSTM models from scratch. As they need to be trained on large data and as our hardware limits us, we will continue with the fine-tuning of transformers.

2.2 Transformers Architectures

2.2.1 Cross-Encoder for Natural Language Inference - Hugging Face

Presentation of the pre-trained model

The cross-encoder/nli-roberta-base model is a pre-trained transformer model for the Natural Language Inference (NLI) task. The model uses a RoBERTa architecture based on transformer neural networks, which are state-of-the-art natural language processing models. The uniqueness of transformer neural networks is that they can capture contextual information about word sequences through multi-head attention.

The cross-encoder/nli-roberta-base model is a "cross-encoder" type model, which means it processes both input sentences at the same time to determine their relationship. The model takes two sentences as input in the form of character strings and generates a single output indicating whether the two sentences are contradictory, entailment, or neutral.

To train the model, a supervised classification method is used. This means that the model is trained on a large annotated dataset, where each pair of sentences is associated with a label (contradiction, entailment, or neutral). The model is trained to minimize a cross-entropy loss function between the predictions and the actual labels.

The model is pre-trained on a large corpus of unannotated text data (using the self-supervised MLM technique) to learn highly informative sentence representations. Then, it is fine-tuned on an annotated dataset for the specific task of Natural Language Inference (NLI). Here is a high-level representation of the model's architecture:
- Input: The model receives a pair of sentences (premise and hypothesis) as input. The tokenizer processes these sentences into token IDs, which are fed into the model.

- Embeddings: The model generates input embeddings by combining:
1) Token embeddings: These are learned vector representations of individual tokens.
2) Position embeddings: These represent the position of each token in the input sequence.
3) Segment embeddings: These embeddings indicate which sentence a token belongs to (either the premise or hypothesis).

- Transformer layers: The transformer layers are composed of multiple self-attention and feed-forward sub-layers. Each transformer layer processes the input embeddings and generates contextualized token representations. These layers can be stacked, with the output of one layer serving as input to the next layer.

- Pooling: After the final transformer layer, a pooling layer is applied to convert the contextualized token representations into a single fixed-size vector representation. In RoBERTa, this is done by taking the representation of the first token (the [CLS] token) or by using mean-pooling or max-pooling techniques.

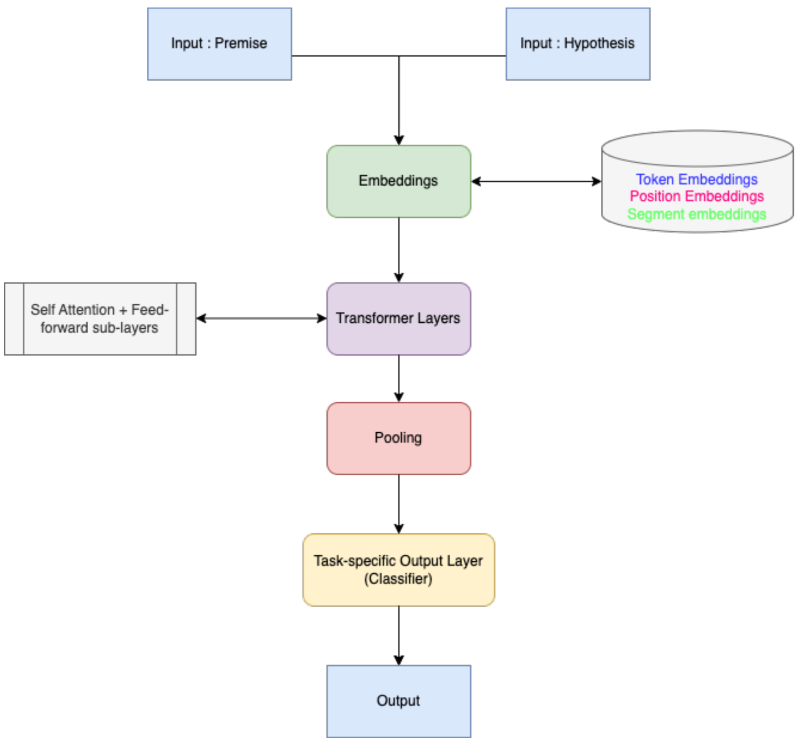

Fig. 4 Cross Encoder architecture

- Task-specific Output Layer (Classifier): A linear classifier is added on top of the pooled vector representation. This classifier is specific to the NLI task and predicts the relationship between the input sentences. It has three output units, one for each class (entailment, neutral, and contradiction), and uses a softmax activation function to convert the output units into probabilities.

- NLI Prediction: The model outputs the predicted relationship between the premise and hypothesis sentences as an index corresponding to the class label with the highest probability.

It should be noted that this model was trained on the SNLI and MultiNLI datasets. The Multi-Genre Natural Language Inference (MultiNLI) corpus is a collection of 433k sentence pairs annotated with textual entailment information, sourced from crowd-sourcing. The corpus is modeled on the SNLI corpus but differs in that it covers a range of spoken and written text genres and supports a unique cross-genre generalization evaluation.

The model without fine-tuning achieves an accuracy of 87.47.

2.2.2 Semantics-aware BERT - semBERT

Overview of the model

The **Semantics-aware BERT** (semBERT) model is the combination of a pre-trained **semantic role labeler** and **pre-trained BERT transformer model** with an addition of a **dense layer** in order to perform Natural Language Understanding (NLU) tasks.

Why semBERT ?

In the state of art, they proposed to incorporate explicit contextual semantics from pre-trained semantic role labeling, and introduce an improved language representation model, Semantics-aware BERT (SemBERT), which is capable of explicitly absorbing contextual semantics over a BERT backbone. SemBERT keeps the convenient usability of its BERT precursor in a light fine-tuning way without substantial task-specific modifications. Compared with BERT, semantics-aware BERT is as simple in concept but more powerful. It obtains new state-of-the-art or substantially improves results on ten reading comprehension and language inference task.

Model Presentation

Indeed, **semBERT model architecture is based on BERT model and have the particularity to in parallel passes the inputs data through a semantic role labeler to extract predicate-argument structure information from the input sentences**. Also, we added at the top of the BERT model **a dense layer with three units and a softmax activation function** for the NLI task.

To present the model architecture deeper, we've implemented the following steps :

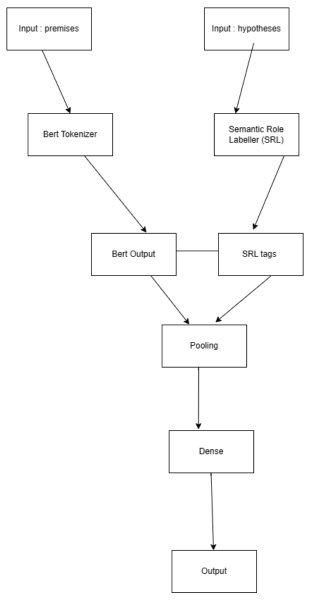

Fig. 5 Overview of model architecture

1) First, we use a pair of sentences (premises and hypothesis) that we tokenized using the BertTokenizer in order to transform text into a format that can be processed by the BERT model.

2) The train premises and hypotheses has been processed through a semantic role labeller using AllenNLP library annotate every token of input sequence with semantic labels. Then, the resulting SRL labels are mapped to unique integers and one-hot encoded to be better understood by the model.

3) Then, we extract the outputs of pre-trained BERT model is used to obtain a representation of the input sequence (contextualized embeddings), which is then concatenated with the semantic role labeling (SRL) representation. This will be the inputs of the dense layer we will apply for classification.

4) We apply a pooling layer both the last hidden state and the one-hot encoded SRL labels concatenated in the step 3 for a dimensional purpose.

5) These pooled representations are concatenated and passed through a dense layer with a softmax activation function, which outputs the label corresponding to the relationship bewteen the premises and hypothesis.

6) Finally, the model is compiled with the sparse categorical crossentropy loss function, the adam optimizer, and the accuracy metric, and it is trained on the concatenated representations for 10 epochs with a batch size of 32.

However, we have tested this model on a very small subset of the SNLI train dataset because the Google Colab RAM memory were crashing despite the connection to a

GPU. In consequences, we weren't able to train this model on the 550 000 rows of the train sample and don't get the real performance of the model despite the fact that we have an accuracy of 50 in average on the train sample used that were containing 15 rows.

2.3 Most performing model : Hyperparameters Fine tuning

2.3.1 How we attempted to improve the model

We chose to fine-tune the cross-encoder transformer.

Our code fine-tunes only the last layer of the model. The reason is that the model parameter passed to Trainer contains the pre-trained weights of the model, and the fine-tuning is done only on the last layer that is randomly initialized. The other layers of the model are frozen and kept as they were pre-trained.

The reason for fine-tuning only the last layer is that it typically contains the most task-specific information and adjusting it can improve the model's performance on the target task without losing the general knowledge learned from pre-training.

In the code, the model object passed to the Trainer is a pre-trained language model, which consists of multiple layers. Let's denote the parameters of this model as $\theta_{pretrain}$.

The fine-tuning process aims to optimize the parameters of the last layer θ_{last} to fit the model to a specific task. During the fine-tuning, the pre-trained parameters $\theta_{pretrain}$ are frozen, and only the parameters of the last layer are updated. The fine-tuning process can be formulated as a minimization problem, where the objective function L is the task-specific loss function, and the parameters θ are the parameters of the last layer:

$$\theta_{finetune} = \operatorname*{argmin}_{\theta} L(\theta, \theta_{pretrain})$$

The fine-tuning is performed using gradient descent or one of its variants, where the gradients of the loss function with respect to the last layer parameters are computed using backpropagation:

$$\nabla_{\theta} L(\theta, \theta_{pretrain}) = \nabla_{\theta_{last}} L(\theta_{last}, \theta_{pretrain})$$

The gradients of the loss function with respect to the pre-trained parameters $\theta_{pretrain}$ are not computed, and thus they are not updated during the fine-tuning.

Therefore, the code fine-tunes only the last layer of the pre-trained language model, while keeping the pre-trained parameters of the other layers fixed.

We had the idea of making the model more specific to the SNLI dataset by fine-tuning it on a sample of the train dataset. Fine-tuning a pre-trained model on a smaller sample of the same dataset can be useful in certain situations. The main idea behind this approach is to adapt the pre-trained model to the specific characteristics or distribution of the smaller sample, which could potentially differ from the overall distribution of the entire dataset.

When fine-tuning a model on a smaller sample, we are essentially performing a form of transfer learning. The model has already learned general language patterns and features during its pre-training phase on a large dataset, like SNLI and MultiNLI. By fine-tuning the model on a smaller sample, you are allowing it to adapt and specialize in the specific patterns present in the sample. This can lead to better performance on the target task, especially if the smaller sample has unique characteristics or a slightly different distribution compared to the overall dataset.

With this method, we have reached an accuracy of 0.89. However, there are limitations and challenges to this method:

- Overfitting: Since the sample size is smaller, the model might overfit to the specific characteristics of the sample, reducing its generalization capability to the entire dataset or similar data. To mitigate overfitting, regularization techniques can be applied, early stopping can be used, or the model's complexity can be reduced. Nevertheless, we were able to take into account overfitting by implementing hyperparameters, which we will detail later.

- Representativeness: The smaller sample may not be representative of the entire dataset, leading to biased or suboptimal performance on the target task. It is crucial to ensure that the smaller sample is representative of the overall dataset, capturing its diversity and distribution. We tried to put the most representative sample that our machines could take. However, we think that with a higher-performance machine that can take a greater sample, the results would be more promising.

2.3.2 Hyperparametrisation for overfitting and computation time

Here are the parameters that we chose :

```
training_args = TrainingArguments(
    output_dir='./results', # directory to save model checkpoints and results
    evaluation_strategy='steps', # evaluate every eval_steps
    eval_steps=500, # evaluate every 500 steps
    load_best_model_at_end=True, # load the best model at the end of training
    per_device_train_batch_size=8, # batch size for training
    per_device_eval_batch_size=32, # batch size for evaluation
    num_train_epochs=1, # number of training epochs
    metric_for_best_model='eval_loss', # metric used to determine the best model
    warmup_steps=0.1 # number of warmup steps for learning rate scheduler
)
```

Following are the explanations for our choices of parameters :

outputdir : This parameter specifies the directory where the model outputs, such as checkpoints and logs, will be saved during training and evaluation.

evaluationstrategy : This parameter specifies the evaluation strategy during training. In this case, the evaluation will be performed at regular intervals based on the number of training steps specified by *evalsteps*.

evalsteps : This parameter specifies the number of training steps after which evaluation will be performed. In this case, evaluation will be performed every 500 training steps.

loadbestmodelatend: This hyperparameter sets whether the best model checkpoint based on the evaluation metric specified by *metricforbestmodel* will be loaded and used at the end of training. Loading the best model can prevent overfitting by ensuring that the model with the best performance on the validation set is used for inference, but it may increase computation time because more model checkpoints need to be saved and evaluated.

perdevicetrainbatchsize: This hyperparameter sets the batch size per device during training to 8. A larger batch size can lead to faster convergence during training, but it can also increase the risk of overfitting because the model is learning from more data at each step. In this case, a batch size of 8 is relatively small, which may help reduce the risk of overfitting but may slow down training because it takes longer to process each batch.

perdeviceevalbatchsize: This hyperparameter sets the batch size per device during evaluation to 32. A larger batch size can speed up evaluation, but it may not improve the quality of the evaluation because the model is not learning from the data during evaluation. In this case, a batch size of 32 is relatively large, which may speed up evaluation but may not improve the quality of the evaluation significantly.

numtrainepochs: This hyperparameter sets the number of training epochs to 1. Increasing the number of epochs can lead to overfitting because the model has more opportunities to learn from the training data, but it can also improve the quality of the model's predictions. In this case, setting the number of epochs to 1 may reduce the risk of overfitting. Moreover, if there is only a small amount of training data available, setting the value at 1 may be a good choice to prevent the model from overfitting to the limited data. In this case, training for more than one epoch may result in the model memorizing the training data.

metricforbestmodel: This hyperparameter sets the evaluation metric used to select the best model checkpoint to the evaluation loss. Choosing an appropriate evaluation metric can help prevent overfitting by ensuring that the model's performance is evaluated on a relevant metric. In this case, using the evaluation loss as the evaluation metric is appropriate because it directly measures the model's performance on the task.

warmupsteps: This hyperparameter sets the number of warmup steps during training to 0.1 times the total number of training steps. Warmup steps can help prevent overfitting by gradually increasing the learning rate and allowing the model to learn from easier examples before tackling more difficult examples. In this case, setting the number of warmup steps to 0.1 may help prevent overfitting without significantly increasing training time.

2.4 Results of the most performing model

Our most perfoming model was the fine-tuned Cross-Encoder for Natural Language Inference (with Roberta Base). Here are the results :

Table 1 Metrics for the train and test datasets.

	Train Dataset	**Test Dataset**
Evaluation Loss	0.3315	0.3217
Evaluation Accuracy	0.9037	0.8927
Evaluation Runtime (seconds)	427.12	286.10
Evaluation Samples per Second	23.04	34.34
Evaluation Steps per Second	0.72	1.07
Epoch	1.0	1.0

Based on the provided metrics, it can be concluded that the model achieved a higher accuracy on the training dataset compared to the test dataset, which suggests that the model may have overfit a trifle to the training data.

However, the evaluation loss for the test dataset was slightly lower than the evaluation loss for the train dataset, indicating that the model was able to generalize somewhat to the test data despite the overfitting issue.

The evaluation runtime was longer for the training dataset compared to the test dataset, which is due to the larger size of the training dataset. Finally, the evaluation samples per second and evaluation steps per second were higher for the test dataset compared to the train dataset, which could be because the test dataset is smaller or because the model's performance on the test data was better.

Overall, these metrics provide a snapshot of the model's performance on the provided datasets, but additional evaluation and testing may be necessary to fully understand its strengths and weaknesses.

2.5 Conclusion

In this project, we developed an LSTM model from scratch for the task of natural language inference (NLI) using the SNLI dataset. Unfortunately, the model's accuracy was quite low, only achieving 0.33, indicating that it struggled to capture the complex relationships between the sentences in the dataset.

To address this issue, we decided to experiment with pre-trained transformer models, which have been shown to perform well on a variety of natural language processing tasks. We evaluated two models, cross-encoder/roberta-base and sembERT, and found that the cross-encoder model performed better. We then fine-tuned the cross-encoder model using the SNLI dataset and achieved better results.

While our project showed promising results, there are several potential ways to improve it. One approach would be to try different transformer models or variations of the existing models to see if they perform better. For example, we could experiment with models like GPT, or XLNet to see how they perform on the NLI task.

Another avenue for improvement is to explore other pre-processing or data augmentation techniques that could help the model better capture the nuances of natural language. For example, we could use techniques like sentence or word embedding, stemming, or lemmatization to preprocess the data. We could also explore data augmentation techniques like back-translation or paraphrasing to increase the diversity of the training data.

Finally, it may be useful to explore ensembling techniques, such as combining the predictions of multiple models, to further boost performance. For example, we could ensemble our fine-tuned cross-encoder model with other transformer models to see if we can achieve better results.

Overall, this project provides a valuable learning experience in developing and fine-tuning deep learning models for natural language processing tasks. By exploring different models and techniques, we can gain a deeper understanding of the strengths and weaknesses of different approaches and improve our models' performance over time.

2.6 Contributions

José Ángel García Sánchez

During my internship, I had the opportunity to work with a Seq2SQL model using TAPEX and fine-tuning it on my data. Although it was a great learning experience, I wanted to challenge myself further and explore the depths of deep learning. This led me to dive into building a custom RNN (LSTM) from scratch.

One of the major challenges I faced while working on this model was to ensure that each of the arrays had an appropriate shape. I spent a lot of time experimenting with different shapes and sizes to achieve the best possible results. It was a great feeling to finally get it right and see the model working seamlessly.

Working on this custom RNN gave me a better understanding of how deep learning works and how to build a model that fits my specific needs. It also taught me the importance of research and staying up-to-date with the latest developments in the field. I spent a considerable amount of time reading articles, papers, and blogs to gain a deeper understanding of the subject.

Overall, the experience was challenging yet rewarding. It gave me the confidence to take on more complex projects and expand my knowledge in the field of deep learning. I'm excited to continue my journey of learning and exploring new technologies in the future.

Sarra Ben Yahia

My initial foray into transformers involved creating a chatbot as a toy project. While the project is still ongoing, you can check out the code for the model here https://github.com/sarrabenyahia/chatbot-mental-health/blob/main/creating_patterns.py. To improve the chatbot's generalization capabilities and learning performance, I employed data augmentation techniques to generate paraphrases of my question instances.

However, this was an exploratory endeavor, and I heavily relied on advice from ChatGPT without a deep understanding of the underlying mechanisms, even though the results were satisfactory.

Taking this course and working on this project has afforded me an introduction to the architecture and fine-tuning of transformers. Through several experiments, I gained deeper insights into the intricacies of this domain, although the results were not always promising.

The Natural Language Inference (NLI) topic was particularly intriguing, and I want to express my appreciation to our professor for introducing us to this concept that I was previously unfamiliar with. The hyper-parameterization process to address overfitting during fine-tuning was particularly noteworthy for me. The process of hyperparameter tuning is a fundamental practice in machine learning and deep learning used to optimize model performance, and fine-tuning transformers is no exception.

While concepts like regularization and learning rate are relevant to both machine learning and deep learning, the techniques used to implement them may differ in the two domains. For example, in the context of fine-tuning transformers, we employed regularization techniques like weight decay and dropout to prevent overfitting, and we also tuned the learning rate to improve the model's convergence speed and performance. Another specific technique we used in this project is warm-up steps.

Similarly, we also had to consider the impact of batch size on memory requirements and computation time during the training process. Larger batch sizes can improve training efficiency, but they also require more memory and computation time, which can be a bottleneck when dealing with larger datasets or when resources are limited.

In general, I am satisfied with my/our progress on this project, despite the time constraints. I was particularly taken aback by the models' running time, which could surpass an hour if not optimized for batch size and other relevant parameters.

Amande Edo

Working on the natural language inference (NLI) task has been for me very interesting first of all because it was the first time i was facing natural language processing tasks and I learned to manipulate words instead of numbers like in machine learning tasks.

It was also very challenging because while implementing the semBERT model because I've implement it in a different manner then in the actual state of art which has lead me to a deep reflexion about what i wanted to achieve through the different steps of the model. For example, I've chosen to introduce the sentences tag as inputs of the model based on a dictionary that was attributed a distinct integer value to each tags that a sentence was composed of so that we can better distinguish the specificity of the premises and hypothesis.

Furthermore, I've learned a lot about neural networks and the particularity of inputs dimensions matching because of the bugs fixing i've been through the project.

To conclude, another aspect of the project I wasn't prepared to is the impact of the memory taken when constructing a deep learning model. For instance, while using Google Colab notebook I've faced a lot of RAM troubleshot that I wasn't expected to and which slows me down during building the model and makes me realize the importance of GPU in deep learning area.

Giulia Dall'Omo

Working on natural language inference (NLI) tasks based on the Stanford Natural Language Inference (SNLI) dataset can be an interesting and valuable experience for several reasons. Firstly, the SNLI dataset is widely used as a benchmark dataset for NLI tasks, making it an essential dataset to work with in the field of natural language processing (NLP). Moreover, NLI is a fundamental task in many NLP applications such as question-answering systems, chatbots, and machine translation. Therefore, working on the SNLI dataset can provide practical experience in building NLI models that can be used in real-world applications.

Working on a complex deep learning task for the first time was truly a remarkable experience for me. I had the opportunity to work with transformers and models that take a considerable amount of time to run. This practical exposure allowed me to learn new skills and techniques, and most importantly, to apply what I have learned in class.

Although I believe there is still so much to learn, this project has piqued my interest in a subject that I previously had little knowledge of. The subject is highly intriguing, and with the recent release of ChatGPT, it is the talk of the town. As a naturally curious person, I found myself pondering the inner workings of ChatGPT and how it could be so powerful. This initial exposure to NLI has sparked my curiosity, and I am excited to explore it further in the future.

In conclusion, this project was not only a platform for learning new things, but it also introduced me to model structures that I had never come across before, such as pre-trained models. However, there is still so much to learn, and I am grateful for my classmates' unwavering support throughout the project's design. I extend my heartfelt gratitude to them for their assistance.