Deep Learning techniques for Natural Language Processing: A multilingual Encoder model for NLI task

Author:
Alessandro Manenti

Industrial Supervisor:
Dr. Dario Del Sorbo

Academic Supervisor:
Prof. Alfredo Braunstein

A thesis submitted in fulfillment of the requirements
for the Master's double-degree in Physics of Complex Systems

October 13, 2022

POLITECNICO DI TORINO and SORBONNE UNIVERSITÉ

Master's double-degree in Physics of Complex Systems

Deep Learning techniques for Natural Language Processing: A multilingual Encoder model for NLI task

Alessandro Manenti

In this manuscript, we build an Artificial Intelligence model that classifies the inference relations between pairs of English or Italian sentences with accuracies above 75%. To do so, we leverage pre-trained Transformer-based [1] [2] sentence Encoders [3] to encode sentences into high-dimensional vectors. Then, we build and test different algorithms to compare the encoded vectors and to infer the logical relations between texts. On the SNLI [4] validation set, the simple and fast dot-product reaches an accuracy of 50.13%; fine-tuning the encoder we obtain a 15.26% increment of performances; while relying on a more complex algorithms - the Support Vector Machines - we achieve an accuracy of 84.13%. At the end, we study 4 different Deep Learnig end-to-end models with 4 different attention heads (Fully Connected, Convolution, Convolution generalization and dot-product generalization). The best model on both the SNLI and on the MNLI dataset is the one that uses a series of Fully Connected Layers as comparing algorithm. It obtains 80.69% on the first and 77.00% on the latter.

We detail the theory behind Transformers and why they have outperformed state-of-the-art architectures on Natural Language Processing tasks. We stress the symmetries we exploited and the motivations that led us to improve models' performances on two Italian datasets (achieving an accuracy of 63.38% and 81.65% on Italian RTE-3[1] and ABE_ABSITA [5] datasets). While the model developed may still improve, it is already able to support quantitative text analysis in industrial environments.

[1] Unluckily the official website is down, but the dataset can be downloaded here: https://github.com/gilnoh/RTEFormatWork/tree/master/RTE3-ITdata-original-format/RTE3-ITA_V1_2012-10-04

Contents

Chapter 1

Introduction to Natural Language Processing

In this thesis, we aim to build an AI model that understands the language. In Sec. 1.1, we provide some intuition by describing the main tasks an AI may perform. Then, in Sec. 1.2, we briefly describe Deep Learning (DL) as it is the field of AI with the most accurate models for solving NLP tasks. We then start describing RNN and LSTM in Sec. 1.3, two of the first DL models that could process language in a complex and accurate way. Finally, we describe in depth the state-of-the-art models for Natural Language Processing: Transformers and Transformer-based architectures (Sec. 1.4) and state our research goal in Sec. 1.5

1.1 Natural Language Processing

Natural Language Processing is the branch of AI that deals with human language comprehension. A language is a form of information encoding that can vary a lot. For example the same piece of information contained in "I like skiing" is represented differently if it is in English or Italian, if it is an audio recording or a text. Furthermore, it is possible to express the same meaning by paraphrasing it with similar words. Data like this are called unstructured and are very difficult to process automatically. In this research, we specifically focus on text inputs.

We aim to build an AI architecture that can extract quantitative information from Italian sentences. There are many tasks and related datasets that we explored that allow us to extract pieces of information from text. Here we name a few of them:

- **Semantic textual similarity** (STS) [6]: Couples of sentences are labeled from 0 to 5 based on how similar they are (5 if they mean the same thing, as "I like skiing" and "I like sliding down the mountains with skis", 0 if completely dissimilar). We decided not to use this dataset as "similarity" is a vague notion and we saw that scores can be subjective. For example, in the dataset the sentences "A woman is playing the guitar" and "A man is playing guitar" have been labeled with an average score of 2.4, while "A woman is playing the flute" and "A man is playing a flute" have been labeled with a score of 2.75 (almost a 15% increment for the same "woman" $\longrightarrow$ "man" change, in the same context). Furthermore, the STS task is good for finding some sort of similarity, but there is no control on which part of the sentences to focus on.
- **Question-Answer** (QA) (an example dataset is TriviaQA [7]): Couples of question and answers are collected in pairs. This is a great task for building AI that learn notions about the world and can answer questions. Anyway, as we want

to extract precise pieces of information from reviews, QA systems wouldn't be useful.

- **Natural Language Inference** (NLI) (two example datasets are Stanford NLI [4] and Multi NLI [8]): NLI datasets are composed of sentence pairs ($sentence_1$, $sentence_2$) and labels. Each pair is labeled with: *entailment*, *contradiction* or *neutral* labels:

  Entailment: assuming that $sentence_1$ is true then $sentence_2$ is true.

  Neutral: assuming that $sentence_1$ is true then $sentence_2$ might be true.

  Contradiction: assuming that $sentence_1$ is true then $sentence_2$ is definitely false.

  For each class, we show some examples in Table 1.1

$sentence_1$	$sentence_2$	NLI label
"A soccer game with multiple males playing"	"Some men are playing a sport"	Entailment (1)
"An older and younger man smiling"	"Two men are smiling and laughing at the cats playing on the floor"	Neutral (0)
"A man inspects the uniform of a figure in some East Asian country"	"The man is sleeping"	Contradiction (-1)

TABLE 1.1: 3 examples from the SNLI inference task.

An architecture trained on NLI tasks learns the inference relations between sentences. Many practical problems can be mapped in an inference task and this is the main reason we used these datasets. In Chapter 2 we will explain more in detail how we will use NLI datasets to solve practical problems.

1.2 Introduction to Deep Learning

In this section, we briefly describe the high-level concepts of Deep Learning (DL). Deep Learning is a field of AI particularly suitable for understanding and processing the key features in unstructured data. This is due to the fact that, when we build a DL model, we are defining a series of interconnected layers that learn how to process information nonlinearly. Since the number of layers stacked together usually is big (the number of parameters can easily exceed 10^6) this field of AI is called Deep Learning.

A Deep Learning model is a sequence of parametric mathematical operations. Their processing unit is the neuron.

The Neuron

A neuron processes its input with a linear transformation followed by a non-linear activation function:

$$\text{Neuron}_{\vec{w},b}(x_1, x_2, ..., x_I) = \text{activation\_function}(w_1x_1 + w_2x_2 + ... + w_Ix_I + b) \quad (1.1)$$

EQUATION 1.1: The equation for a Neuron: the processing unit of a Deep Learning model

For every neuron in the architecture, parameters $\vec{w}$ and b may be different. Those parameters are updated during training and should converge to an optimal configuration[1]. There are many standard activation functions with different properties [9], but all of them are non-linear functions. The presence of non-linearity is crucial, as a series of linear transformations can be mapped in a single linear transformation.

Stacking together Neurons produces a Deep Learning architecture that for each input $\vec{x}$ produces an output $\vec{y}$. As the set of all $\vec{w}$ and b parameters are usually randomly initiated, the output is random at the beginning. To obtain useful results, the parameters need to be trained. This is done via the backpropagation algorithm.

The Loss function and the backpropagation algorithm

To reach an optimal configuration of the parameters $\vec{w}$ and $\vec{b}$, we train the model on a dataset, consistently called *training set*. We only describe supervised training, where training data have labels[2]. This way, whenever an input data $\vec{x}$ is processed and a prediction $\vec{y}_{pred}$ is made, the prediction can be compared with the true label $\vec{y}_{true}$ and the parameters can be changed to improve the model.

The loss function $\mathcal{L}_{\vec{w}\,\vec{b}}(\vec{y}_{pred}, \vec{y}_{true})$ is the function responsible for the numerical evaluation of the error made. There are many loss functions, with different properties and applications [10], but the purpose is always the same: estimating the distance between $\vec{y}_{pred}$ and $\vec{y}_{true}$. For each training data, the loss function produces a scalar called loss. Generally, the more $\vec{y}_{pred}$ is different from $\vec{y}_{true}$, the bigger the loss is.

Each model parameter $p_i \in [\vec{w}, \vec{b}]$ contributes to the generation of the output $\vec{y}_{pred}$. So, more in detail, each parameter p_i contributes to the loss. The backpropagation[3] algorithms - coupled with Gradient Descent and an optimization algorithm - tries to find the parameters $\vec{w^*}$ and $\vec{b^*}$ that are a minimum of $\mathcal{L}_{\vec{w}\,\vec{b}}$. The backpropagation algorithm calculates all the partial derivatives $\frac{\partial \mathcal{L}}{\partial p_i}$ with the chain rule. Then, at each training step, parameters are updated in the gradient direction $\vec{\nabla}_{\vec{w},\vec{b}}\,\mathcal{L}$[4]. This process is called Gradient Descent.
An important practical problem arises when we minimize the loss function. $\mathcal{L}_{\vec{w}\,\vec{b}}(\vec{y}_{pred}, \vec{y}_{true})$ is calculated on the training set: there is no guarantee that the parameters $\vec{w^*}$ and $\vec{b^*}$ that minimize $\mathcal{L}_{\vec{w}\,\vec{b}}$ on this set generalize well for other datasets. This is known as the bias-variance tradeoff [11]. Two other auxiliary datasets are

[1] As we will show later, the optimal configuration should be the one that minimizes the Loss function.

[2] There are other training schemes, such as unsupervised trainig, self-supervised training, ...

[3] The name backpropagation derives from the fact that gradients $\frac{\partial \mathcal{L}}{\partial p_i}$ are calculated from the output to the input.

[4] In reality, optimizers are used in order not to get stuck in local minima and perform better. Explicinting how they work is beyond the scope of this manuscript.

used for checking generalization: the validation and the test datasets. The validation set is used to identify the optimal model and optima hyperparameter, the validation set is used to infer real performance.

1.3 RNN and LSTM

Recurrent Neural Networks (RNN) are Deep Learning models that leverage the sequential structure of texts. This is due to the fact that the order of words in a sentence is important. Traditional Multilayer Perceptrons do not exploit this feature. On the other hand, RNNs have only two parameter matrices W_{hh} and W_{Xh} that are shared for all inputs. A sentence is decomposed into single tokens (see 1.4.1 for a detailed description of sentence pre-processing); for simplicity now imagine that each input $\vec{X}_i$ is a vector representation of a single word. Mathematically an RNN processes each input in the following way:

$$\vec{h}_i = \text{activation}(W_{hh}\vec{h}_{i-1} + W_{Xh}\vec{X}_i + \vec{b}_h) \tag{1.2}$$

EQUATION 1.2: Equation for processing input X_i based on previous hidden state $\vec{h}_{i-1}$. $\vec{h}_i$ is a hidden state that contains all the information of previously processed inputs $\vec{X}_1$, $\vec{X}_2$, ..., $\vec{X}_{i-1}$.

The computational graph of an RNN is shown in Figure 1.1. For sequence-to-sequence tasks, hidden layers can be used also as outputs. For sequence to vector tasks the last hidden layer is usually used.

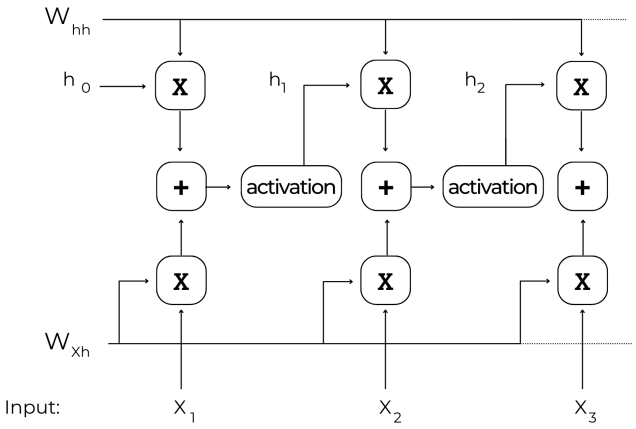

FIGURE 1.1: The general structure of an RNN. Bias $\vec{b}$ are omitted.

In Figure 1.1 and Equation 1.2 we can notice that each input $\vec{X}_i$ is processed in the same way and sequentially. In order to produce hidden state $\vec{h}_i$, $\vec{X}_i$ and $\vec{h}_{i-1}$ are processed with the same weight matrices.

Despite being great architectures for text processing RNNs have important drawbacks. Each time a hidden layer i is calculated, the information of all previous inputs $\{\vec{X}_0, \vec{X}_1, ..., \vec{X}_{i-1}\}$ gets diluted by a constant factor. This produces an exponential decay of information along the net, so, far inputs do not interact with each other. Moreover, when applying gradient descent, the gradients may easily explode or vanish.

This is due to the fact that the same parameter matrices W_{hh} and W_{Xh} are shared in series for many steps [12].

These problems are partially solved by the Long Short-Term Memory [13] Neural Networks. We do not discuss LSTM NN in detail, as for this research we used more recent architectures. Anyway, at a high-level, a LSTM may be thought of as a RNN with some additional memory cells that allow longer-range interactions between words.

1.4 Transformers

The structure of RNNs and LSTM does not allow them to be computed in parallel, as each input token is processed sequentially. Starting from this problem the authors of Ref. [1] built an architecture that could be parallelized more easily, that also handles the exponential decay of information along the net and the exploding/vanishing gradient problem : the Transformer.

1.4.1 The Transformer

The main feature of transformers is the attention mechanism: a series of calculations that allows the Network to focus on some part of the input data and create connections between them. These connections are showed in 1.4.1 paragraph **Multi-Head attention**. The attention mechanism does not sequentially process information. As we have seen with RNNs the output h_t could be written as a function of the input at time t (x_t) and the output at time t-1 (h_{t-1}). On the contrary, the attention mechanism compares all inputs at the same time with all of them, we will see how soon.

The first Transformer model was represented by the authors [1] in Figure 1.2

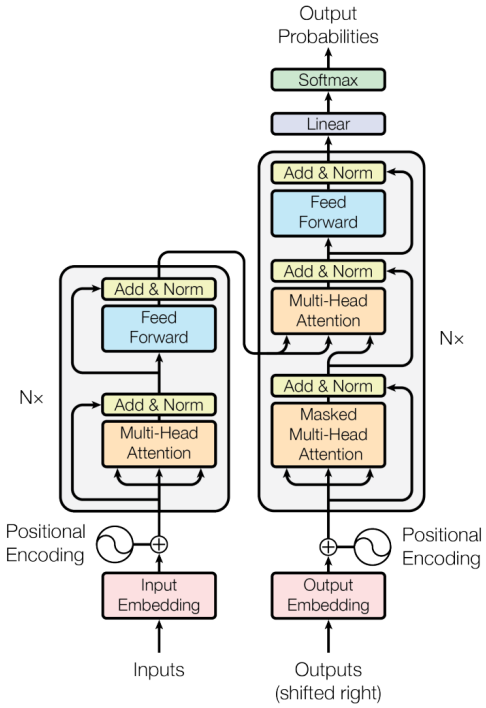

FIGURE 1.2: The structure of a Transformer. It can be divided into two parts: the Encoder, on the right, and the Decoder, on the left. [1].

The right part is the Encoder, used to give a vector representation of the meaning of the input text. On the left-hand side there is the Decoder:

- During training for translation task, the Decoder takes both the Encoder's output and the translated sentence[5] and tries to translate it.
- During training for monolingual data the Encoder receives a sentence of L tokens of a corpus (h_i, h_{i+1}, ..., h_{i+L-1}) and the Decoder receives the same sentence shifted (h_{i+1}, h_{i+2}, ..., h_{i+L}) (properly masked) and tries to predict token h_{i+L}.
- During inference, the Decoder takes the Encoder's output and auto-regressively processes its output to make predictions.

Since the Decoder is task-specific, its core structure is very similar to the Encoder and the Decoder is not used in newer Transformer-based architectures, we only detail the Encoder[6].

In this section we describe how the Encoder processes an input sequence step by step, making explicit input and outputs' dimensions, as it helps understanding the actual computations. We describe the left hand side of Figure 1.2: the *Input Embedding layer*, the *Positional Encoding layer*, the *Multi-Head attention*, the *Add Norm blocks* and the *Feed Forward layer*.

Input Embedding

Inputs are usually in the form of raw text that needs to be pre-processed. This computation is made in the Input Embedding cell. Two transformations occur at this stage: **tokenization** and **token embedding**.

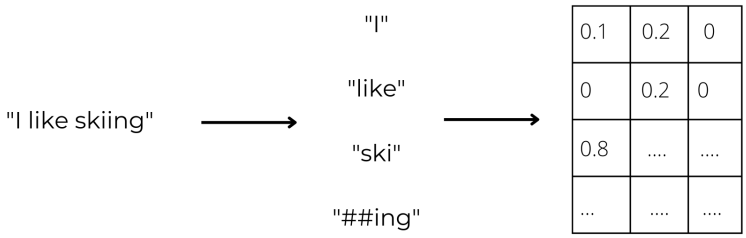

FIGURE 1.3: Input Embedding. Raw text is firstly tokenized (with byte-pair encoding or word-piece, then each token is mapped to a vector

As shown in Figure 1.3, a sentence is split into the most meaningful subtokens: this is **tokenization**. It makes it possible to learn the interaction between subwords (for example it can learn that the word "skiing" is related to "ski" and is in a gerund form). If no tokenization was used, each slightly different word would need a new entry in the dictionary[7].

[5] The translated sentence is properly masked so that it is possible to train in parallel predicting all output tokens at once. Masking simply makes the future tokens in the output sentence unaccesible to the Decoder.

[6] As we will see in the next sections, BERT [2] and Transformer-based sentence Encoders [3] use only the Encoder part, so, for this research, it is useful to focus only on this.

[7] This should be avoided since 1) it increases the degrees of freedom of the task, for no specific reason 2) does not allow out-of-distibution learning (words not seen during training are treated as unknowns) 3) can be computationally infeasible

In the original paper, they used both character-level byte pair encoding (BPE), as trained by [14], and WordPiece [15]. BPE [16] iteratively expands the tokens dictionary until it reaches a certain length. The dictionary is expanded adding the most frequent couple of tokens as a new token. WordPiece instead of adding new tokens using only the frequencies of couples of tokens, calculates a score as:

$$\text{score}(token_i, token_j) = \frac{\text{freq}(token_i \text{ before } token_j)}{\text{freq}(token_i)\text{freq}(token_j)}$$

And then chooses the couple of tokens with the highest score. This way, if two tokens always appear one after the other, they will be favored in being merged. [8]

After tokenization, each token is mapped to a vector of dimension d_{model} (in the original paper they tried different configurations with d_{model} = 256, 512, 1024). This process is called **token embedding**. Embeddings are usually randomly initiated and then learned during training, as they are vectors directly connected to the backpropagation computational graph. After training, embeddings should map similarly used tokens into close vectors (for example "like" embedding vector should be closer to "love" embedding vector than to the "chair" one). At the end of the Input Embedding cell the sentence is transformed to a $\mathbb{R}^{n_{tokens} \text{ x } d_{model}}$ matrix[9]

Positional Encoding

As we anticipated and as will see more in detail, all embedded tokens are processed in parallel. At this stage, the architecture has no information about the relative positions of words, and sentences like "I love what I understand" and "I understand what I love" would be the same. Therefore this information is encoded in the vectors adding to each of them a Positional Encoding vector as shown in Figure 1.4.

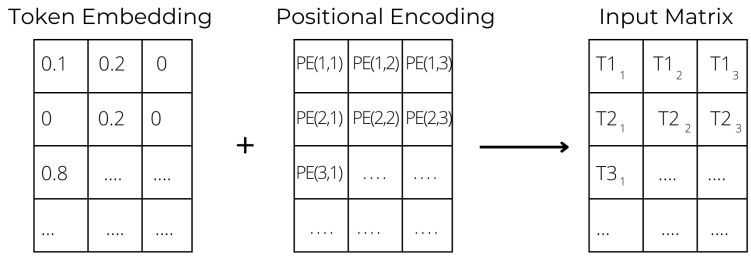

FIGURE 1.4: Positional Encoding. A Positional Encoding value PE(i,j) is added to each entry in the token embedding matrix. This produces the matrix

[8] A good tokenization is essential to obtain state-of-the-art results and shouldn't be neglected. It is worth noticing that other tokenizers exist. Two other famous ones are UnigramLM [17] and SentencePiece [18]. As proved in [19], using UnigramLM rather than BPE improves performances up to 1.4% in the MNLI task.

[9] Actually, the input gets padded (transformed to a fixed dimension by the addition of [PAD] tokens) to a matrix of shape $\mathbb{R}^{d_{pad} \text{ x } d_{model}}$, so it becomes n_{tokens} independent. Since it will be useful to keep in mind which dimension correspond to the tokens we will pretend that $n_{tokens} = d_{pad}$

The authors of Ref. [1] chose the following positional encoding function[10]:

$$PE(i,2j) = sin(\frac{i}{10000^{2j/d_{model}}})$$
$$PE(i,2j+1) = cos(\frac{i}{10000^{2j/d_{model}}})$$

We will see that newer Transformer-based architectures learn the positional encodings. So, the positional encoding block outputs a $\mathbb{R}^{n_{tokens} \text{ x } d_{model}}$ matrix.

Multi-Head attention

As we can see in Figure 1.2, after the Positional Encoding there is the Multi-Head attention. This part is the innovative structure of the Transformer. In the previous section we said that from the Positional Encoding a $\mathbb{R}^{n_{tokens} \text{ x } d_{model}}$ matrix arrives as input of the Transformer, where each row corresponds to a token. Now, this matrix gets projected with 3 different projection matrices $W^Q \in \mathbb{R}^{d_{model} \text{ x } d_k}$, $W^K \in \mathbb{R}^{d_{model} \text{ x } d_k}$ and $W^V \in \mathbb{R}^{d_{model} \text{ x } d_v}$. This process is run h times in parallel. We show this process in Figure 1.5.

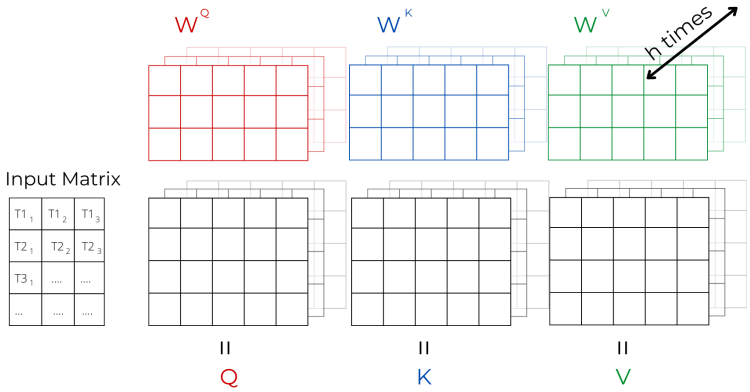

FIGURE 1.5: Linear transformation of input matrices. The matrices Q, K and V will go through the Attention Mechanism.

In Figure 1.5 we can see the linear projection described above, where we took $d_v = d_k$ for simplicity. The outputs of this step are $2h$ $\mathbb{R}^{n_{tokens} \text{ x } d_k}$ matrices and h $\mathbb{R}^{n_{tokens} \text{ x } d_v}$ matrices. We will focus on only one set, since all h sets will undergo the same transformations (with different weight matrices). It is important to notice that each row of the Q, K and V matrices corresponds to the transformed encoding of a particular token.[11]. The attention mechanism is the following:

$$\text{Attention}(Q,K,V) = \text{row\_softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{1.3}$$

EQUATION 1.3: Attention mechanism equation. row_softmax is the softmax function applied row by row.

[10] They also explored learned positional encodings obtaining similar results.

[11] The names Q, K and V derive from information retrieval. Q stands for Query, K for Key and V for Value. Queries are compared to Keys and then, a Value based on the matches.

A representation of this operation is shown in Figure 1.6.

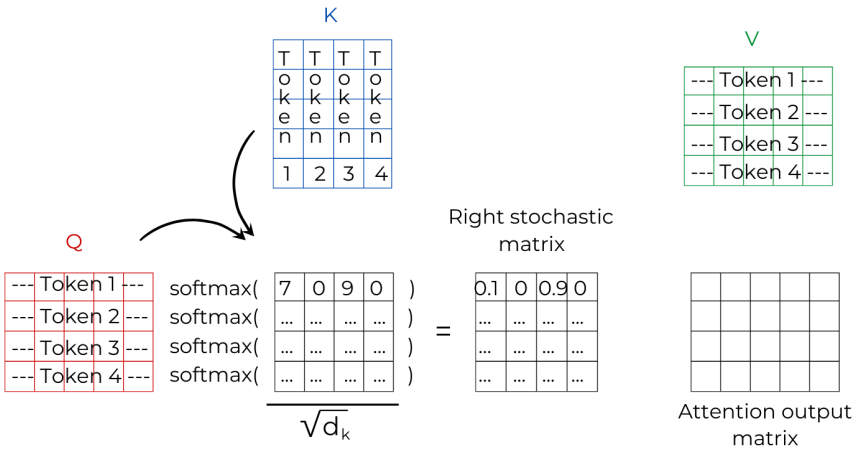

FIGURE 1.6: Visual representation of the Attention Mechanism. Numbers in the right stochastic matrix are approximated to the first significant digit.

From Figure 1.6 it is clear what the attention mechanism does: firstly it compares each transformed input token with each transformed input token. This produces a $\mathbb{R}^{n_{tokens} \times n_{tokens}}$ matrix that should entail the relations between tokens. Then, it is divided by $\sqrt{d_k}$[12] and a softmax function is applicated row by row. This way, a right stochastic matrix is produced: each row i should represent how much token i is related to token j[13].

Then, the right stochastic matrix selects the elements of the value matrix V. Take the example in Figure 1.6: If $token_1$ of Q has a high dot product with $token_1$ and $token_3$ of K, then the right stochastic matrix's row 1 has the highest probabilities in entry 1 and 3. Then, the attention output matrix has, as row 1, mainly the weighted sum of matrix V's $token_1$ and $token_3$.

In Figure 1.7 we show the attention mechanism for an input sentence.

[12]The authors of [1] decided to divide by $\sqrt{d_k}$ because if we take two d_k dimensional vectors randomly distributed with 0 mean and variance 1, their dot product has 0 mean and d_k variance.

[13]Whenever we talk about a "token" it is important to keep in mind that we refer to the linearly transformed token. We dropped the adjective "transformed" for simplicity.

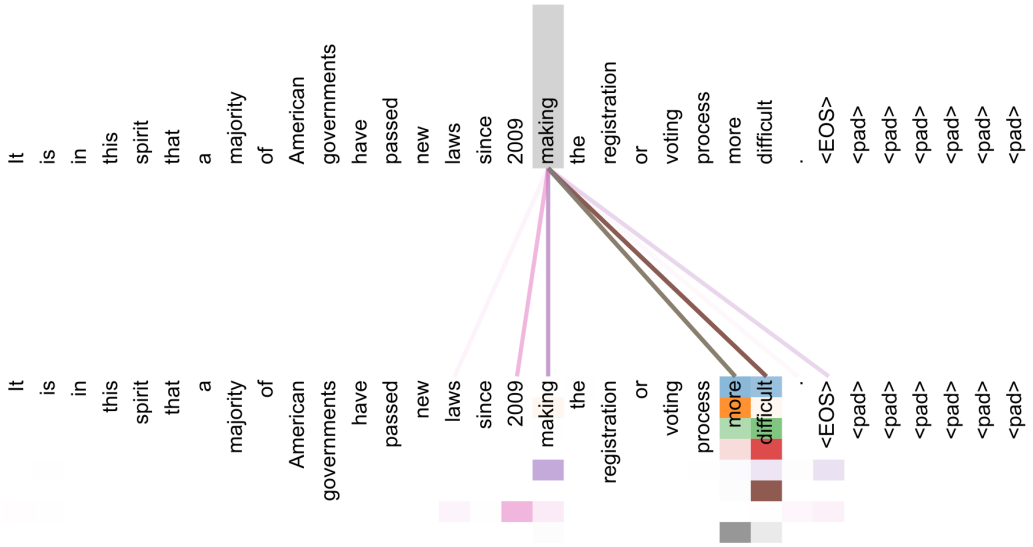

FIGURE 1.7: An example of the attention mechanism. Each color represent an attention matrix for the word "making". Figure taken from [1]

As we can see, the attention mechanism can learn long-range interactions between words. The output of each of the h attention heads is a $\mathbb{R}^{n_{tokens} \times d_v}$ matrix. These matrices are then concatenated in a $\mathbb{R}^{n_{tokens} \times hd_v}$ matrix and linearly projected with $W^0 \in \mathbb{R}^{hd_v \times d_{model}}$. The complete Multi-Head attention is:

$$\text{MultiHeadAttention}(Q, K, V) = \text{concat}(\text{Attention}_1(Q, K, V), \text{...}, \text{Attention}_h(Q, K, V))W^0 \tag{1.4}$$

with Attention_i(Q, K, V) from Equation (1.3)

EQUATION 1.4: Mathematical description of Multi-Head attention. Each attention head Attention_i(Q, K, V) has different weight matrices W_i^Q, W_i^V and W_i^K

Add and normalize

The output of the Multi-Head attention is a $\mathbb{R}^{n_{tokens} \times d_{model}}$. This output is summed to the output of the Positional encoding (a $\mathbb{R}^{n_{tokens} \times d_{model}}$ matrix) via a Residual Connection [20]. A Residual Connection is a processing step that sums the most recent output to the output of some precedent processing block. From Figure 1.2 The residual connection is depicted as the line that connects the Positional Encoding output to the Add & Norm layer. [14].

Then Layer Normalization is applied [21]:

[14] Note that a unique and fixed d_{model} facilitates these residual connections through the model.

1. Each row[15] of the $\mathbb{R}^{n_{tokens} \times d_{model}}$ matrix gets normalized with 0 mean and unitary variance[16]

2. Then, every column, is multiplied by a trainable parameter a_c, with c $\in$ {1, 2, ..., d_{model}}). Then, to every entry of the column the same trainable parameter b_c is added.

The final output is again a $\mathbb{R}^{n_{tokens} \times d_{model}}$ matrix.

Feed Forward

The $X \in \mathbb{R}^{n_{tokens} \times d_{model}}$ matrix is then transformed via a Feed-Forward Network:

$$\text{FFN}(X) = \max(0, XW_1 + b_1)W_2 + b_2 \tag{1.5}$$

EQUATION 1.5: Equation of a Feed Forward Network, with 1 hidden layer and ReLU activation function.

with $W_1 \in \mathbb{R}^{d_{model} \times d_{FF}}$, $b_1 \in \mathbb{R}^{1 \times d_{FF}}$, $W_2 \in \mathbb{R}^{d_{FF} \times d_{model}}$ and $b_2 \in \mathbb{R}^{1 \times d_{model}}$. The authors of [1] tried many hyperparameters' settings, the interesting thing is that the best results were obtained for $d_{FF} > d_{model}$[17].

In the original paper (and in the literature in general), the tendency is to deeply study the attention mechanism. On the other hand, the authors of [22] demonstrated how the FFN is equally important. In particular, they showed how this processing step (accounting for two thirds of the total number of parameters) behaves as a information retrieval system. Omitting the bias terms for semplicity, a FFN can be written in the following way:

$$\text{FFN}(X) = \text{activation}(XK_M)V_M \tag{1.6}$$

EQUATION 1.6: General form of a Feed Forward Network with 1 hidden layer and no bias.

They showed how, after training, K_M is responsible for capturing the patterns in the input, which then would be retrieved in the V_M matrix: a memory acquired during training.

Total Encoder

After the FFN, another Add and normalize block transforms the data and a $\mathbb{R}^{n_{tokens} \times d_{model}}$ matrix is produced. Then, the blocks from Multi-Head attention process the data again for N times (each time with different weight matrices) and a final $\mathbb{R}^{n_{tokens} \times d_{model}}$ matrix is returned as the output of the Encoder.

[15] A row is a $\mathbb{R}^{1 \times d_{model}}$ vector that is related to a specific mixture of tokens.

[16] Actually, when normalizing the norm a small ϵ term is added to the denominator for numerical stability ($\frac{x-\mu}{\sigma} \longrightarrow \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}$)

[17] An conventional choice is to set $d_{FF} = 4\, d_{model}$.

1.4.2 BERT

As we mentioned above, the Transformer introduced in [1] was trained:

- In an auto-regressive way for single-input tasks: the Decoder uses the Encoder's output as a fixed input, and each of its previously generated outputs to predict the next token.
- Using the "Language A" sentence as input of the Encoder and "Language B" sentence as input of the Decoder for translation tasks.

A Google research group [2] made two main changes to the original Transformer and obtained state-of-the-art results on NLP tasks (7.7% absolute improvement on the GLUE benchmark):

1. Instead of having an Encoder-Decoder structure for processing sentences, they used an unique Encoder that took as input couples of sentences divided by a [SEP] (separator) token.
2. The training procedure changed: they trained the model by substituting 15% of input words with: a [MASK] token (80% of the time), a random word (10% of the time) or the same word (10% of the time). Then the model learned in a self-supervised way to reconstruct the original sentence. They also added a "next sentence prediction task" where the model should understand if the two sentences separated by the [SEP] token were sequential originally.
   The model was then fine-tuned on task specific tasks changing only the last layer and fine-tuning all the parameters.

The authors named this model BERT (Bidirectional Encoder Representations from Transformers)[2]. This architecture laid the foundations for modern Transformers, and in particular for the Transformer-based sentence Encoders: the architecture we used for this work.

1.4.3 Transformer-based sentence Encoders

While BERT-like architectures can solve NLP tasks with high accuracy (often in the 80% - 90% range), in certain situations they are computationally too expensive. Consider the Semantic Textual Similarity task (STS), where we want to understand if couples of sentences are semantically similar[18]. If we want to find the two most semantically similar sentences in a database of N sentences the computational complexity is O(N^2). Since BERT-like architectures have $\sim 10^7$ - $\sim 10^{12}$ parameters, they become too expensive very fast (for as little as N = 10^3 we would need approximately 5 million calls that need $\sim$ 6.5 hours with BERT).

The authors of [3] proposed to use a Siamese Network [23] based on BERT to create Sentence-BERT (SBERT).

Siamese Networks transform unstructured data into vectors so that similar[19] sentences produce similar vectors and no relevant information is lost in the transformation. This is possible since during training the same network is used to produce vector embeddings, and only at the end a few classification layers are used. A scheme

[18] For example, despite using different words the sentence "I love experimenting with recipes" and "I'm fond of cooking" are similar.

[19] It is worth noticing that "similar" is task specific: we may be interested in similar meaning, similar sentiment and so forth.

of the training setup of a Siamese Network is in Figure 1.8

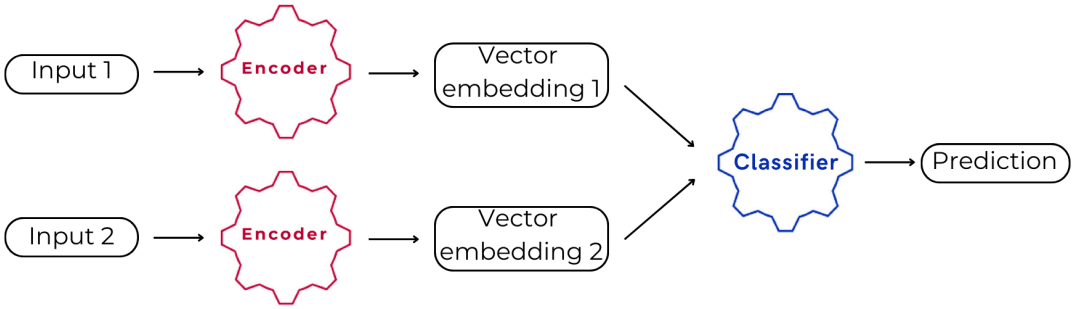

FIGURE 1.8: Training scheme of a Siamese Network.

In our research, we tried different pre-trained Encoders and, based on their proprieties and performances we selected and used 2 of them: a SBERT [20] and a TensorFlow multilingual Encoder [21]

1.5 Research goal

Ideally, we imagine having a large ($\sim 10^5$) corpus of Italian sentences, and we want to extract information from each sentence. So, the goal of this research is to build a model that:

- Can understand inference between pairs of Italian sentences. In other words, we train and test our models on NLI datasets. This way we can query each sentence in the dataset in the way we want.

  For example, consider having a set of hotel reviews and being interested in knowing what custumers think about the price. If two reviews are:

  POS_review: "This hotel is cheap"
  NEG_review: "This hotel is too expensive"

  Then, coupling each review in a NLI task with the ground truth sentence:

  GT_review: "I have paid too much"

  will give:

Review to classify (sentence 1 of NLI)	Ground truth review (sentence 2 of NLI)	NLI label
"This hotel is cheap"	"I have paid too much"	Contradiction (-1)
"This hotel is too expensive"	"I have paid too much"	Entailment (1)

TABLE 1.2: NLI inference task for two reviews

  This way we can query our dataset in a very general way without retraining the model each time.

[20] https://huggingface.co/sentence-transformers/paraphrase-multilingual-mpnet-base-v2
[21] https://tfhub.dev/google/universal-sentence-encoder-multilingual-large/3

- Is based on an Encoder: by encoding the sentences into vectors we can perform many operations on them very fast.

  Imagine that we have a dataset of N reviews and, for each of them, we want to query for Q pieces of information[22], then if we don't use an Encoder we would need O(NQ) calls to the Deep Learning BERT-based model. On the other hand, if we previously encode all the sentences and then compare them with a fast algorithm[23], we need to call the BERT-based Encoder only O(N + Q) times. [24]

- Understands Italian. The majority of the optimized and high-performance models published online are in English. This reflects the fact that research is published in English and important NLP datasets (as SNLI [4], STS [6], MNLI [8]) and benchmarks (as GLUE [24]) are all in English.

Transformers are computationally expensive to train from scratch and require the choice and tune of many hyperparameters (such as learning rate, gradient descent parameters, batch size, loss function, ...). As an example, the latest Google AI NLP architecture [25] was developed and trained from scratch by a team of 50+ people with the collaboration of other 8 teams. Furthermore, one complete training of BERT can take up to 4 days on 16 Cloud TPUs[25] (with 64 TPU chips total). So, we decided to test some multilingual transformer models and fine-tune them to solve our task. In the next chapter, we explain the models we developed and the improvements we made step by step.

[22] An example can be a dataset of hotel reviews. Not only we may be interested in knowing whether each review talks about the price or not, but we may want to know whether it talks about the cleanliness, about the staff and so forth. Moreover, for each topic and each review, we may be interested in knowing the sentiment.

[23] Fast means that the time complexity of the comparing algorithm must be much lower than the one of a BERT-like architecture. This is quite easy to obtain as Transformer architectures often have $\sim 10^7$ - $\sim 10^{12}$ parameters.

[24] Furthermore, the attention mechanism computational complexity scales quadratically with the sentence length L (due to the matrix in the attention mechanism), making each call of the non-Encoder model quadratically slower in space and time.

[25] TPUs or Tensor Processing Units are hardware components developed by Google that allow fast and low-energy training of Deep Learning architectures. They are built to make the tensor operations typical of DL extremely fast.

Chapter 2

Model development

When researching for a Machine Learning model, it is important to follow a scientific method, as acceptable results may also be achieved using an architecture as a "black box". Many times, in Deep Learning it is not well understood why some architectures perform better than others. For example, Transformers where invented as a NLP architecture that could be parallelized. Only at a latter time they started to understand why they outperformed RNN and LSTM. This is the reason why, during our research, we tried to understand the strength and the weaknesses of our models and improved them step by step.

In this chapter, we explain the basic structure of the models we used and the computational-informed or physics-informed motivations that brought us to make certain changes.

2.1 Model structure

As we stated in Chapter 1, we wanted to build a NLI classifier that takes two sentences as input, separately encodes them in a multidimensional-vector and compares them with a fast mathematical rule. Graphically the model structure we want to build is depicted in Figure 2.1:

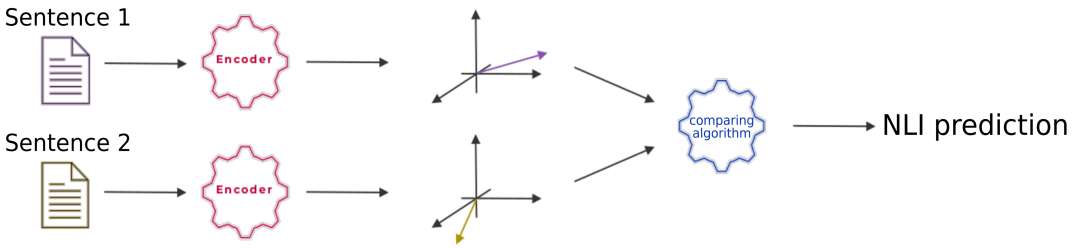

FIGURE 2.1: The general structure of the algorithms we tested. The Encoder is a transformer-based Deep Learning model, while the comparing algorithm is a fast rule.

Before the development of a final model, we tried different choices both for the encoder part and for the comparing algorithm. In the following sections we describe more deeply what we tried and why.

2.2 Encoders

We considered two different pre-trained, open-source alternatives as Transformer-based sentence Encoders available in the following repositories:

- The official SBERT website. Among the 4 multilingual alternative models, we opted for the most performing[1], according to reference [26], and that does not normalize output vectors. [2] We refer to this model as SBERT.
- TensorFlow Hub: the official repository for TF. Between the 7 multilingual encoders we chose the most performing on STS. [3] We refer to this model as TFH.

As we describe in Subsection 2.3.2, we fine-tuned the Transformer-based sentence Encoder model only once. In all the other cases, the Encoders we used were not re-trained. We will always specify the Encoder we used.

2.3 Comparing algorithms

A part from the study of Transformers, their properties, their strengths and weaknesses, the main research focus was on the comparing algorithm. It takes as input two encoded sentences and calculates their NLI label, in a fast way (fast with respect to a Transformer-based architecture).

2.3.1 Dot product

One of the fastest ways to compare vector pairs is via their dot-product, with this purpose, we checked whether the SNLI's validation set could be divided easily, by using the dot-product only. With both Encoders (SBERT and TFH) we encode all the sentences and, for each pair of encoded sentences, we calculate the relative dot product. We produce an histogram for each class and group them in a unique plot. Results for the SBERT model are shown in Figure 2.2. Results for the TFH model are shown in Figure 2.3.

We can clearly see that we cannot properly divide the 3 classes with two thresholds. Even with the best division (showed as black vertical lines) we obtain an accuracy of 57.87% for the SBERT encodings and 50.13% for the TFH encodings.

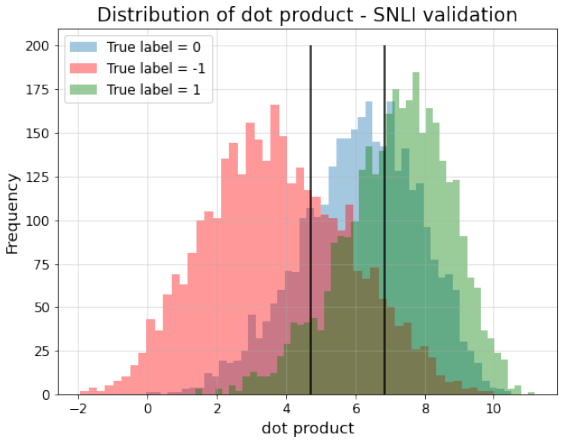

FIGURE 2.2: Histogram of dot product between SBERT sentence embeddings of SNLI validation set. Classifying thresholds are shown.

[1] https://huggingface.co/sentence-transformers/paraphrase-multilingual-mpnet-base-v2

[2] We look for not normalized vectors as we hypothesize that meaning is encoded in the direction of a vector while intensity in its length. So a sentence with a lot of information should be longer

[3] TFH: https://tfhub.dev/google/universal-sentence-encoder-multilingual-large/3

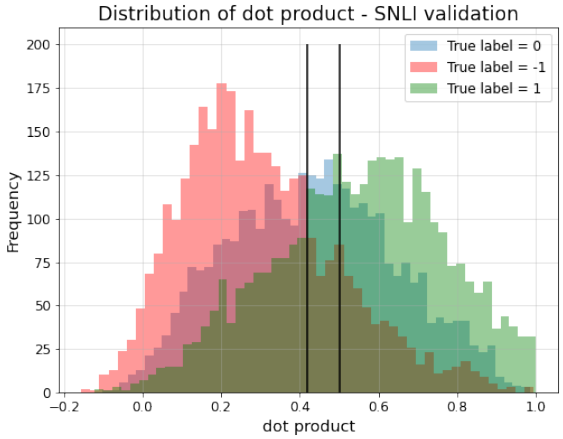

FIGURE 2.3: Histogram of dot product between TFH sentence embeddings of SNLI validation set. Classifying thresholds are shown.

Up to now, the Encoder was not fine-tuned on this comparing algorithm, so we expect to have a large margin of improvement

2.3.2 Dot product with fine-tuning

Using Keras Functional API we build an end-to-end model that takes couples of SNLI sentences, encodes them[4] and calculates the dot product between them. We used the following loss function:

$$\text{RootLoss}(y_{pred}, y_{true}) = \sqrt{|y_{pred} - y_{true}|} \tag{2.1}$$

EQUATION 2.1: Loss function used for the fine-tuning.

A standard choice for regression tasks like this would have been the Absolute Error Loss. We decided to use this loss function since {{-1}, {0}, {1}} are three different classes and this should be considered when building an adequate loss function. More in detail, we represented each class as a point on the continuos line with -1 and 1 as extrema. We spaced the classes linearly, but there is no guarantee that this linearity should hold in the loss function. For example, if a couple of sentences is labeled as "-1", the absolute mean error gives a penality that it is double if the true label is "1" rather than "0". To mitigate this effect more in favor of a multiclass classification task we employed Equation 2.1 the loss function.

We used Adam optimizer [27] with learning rate of 10^{-4}, $\beta_1 = 0.9$, $\beta_2 = 0.999$. We used a learning rate warmup over the first 10% of steps and trained for 5 epochs. These are the set of hyperparameters that were chosen for training BERT [2]. Since a complete training took $\equiv$ 4 hours on a Google Colab GPU we decided not to try other configurations.

We then repeated the analysis on the dot product distributions and got the results of Figure 2.4

[4]Since Keras Functional API is based on TensorFlow, for this section we Encoded sentences with the TFH model.

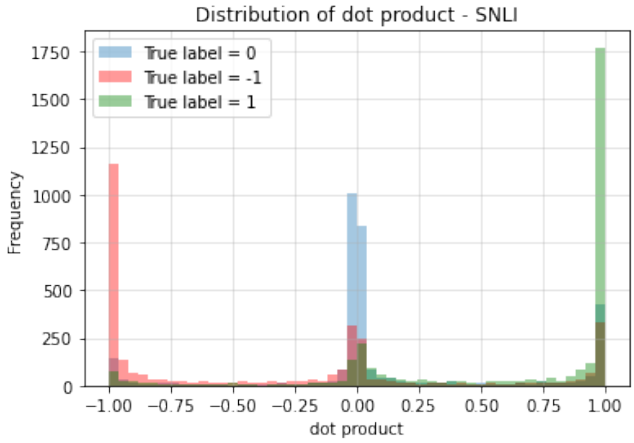

FIGURE 2.4: Histogram of dot products between sentence embeddings of SNLI validation dataset. Results after fine-tuning the TensorFlow Hub Encoder model.

It is straightforward to notice that now two thresholds can divide the distributions quite well. If we set them at -0.25 and at 0.25 we get an accuracy on the validation set of 65.39%. This is an increment of 15.26% from the previous accuracy.

Retraining the encoder on the SNLI dataset we lost the multilingualism of the Encoder. For this reason, we no longer can use this algorithm to treat italian sentences. Anyway, we showed how a fine-tuninig of the Transformer could considerably improve the accuracy, prooving the benefits of this approach.

Up to now, we have always considered the dot product, which is an average operation. The dot product between two vectors s_1 and s_2 can be written as:

$$< s_1 | s_2 >= \sum_{i=1}^{D} s_{1i} * s_{2i}$$

So, we looked at the distributions of element by element product $p_i \equiv s_{1i} * s_{2i}$ to see if only a few components were encoding all the inference relation. We didn't find any single component that could classify the data.

2.3.3 Support Vector Machine

We saw that no components p_i of the element-wise product vector $\vec{p}$ ($\vec{p} \equiv (s_{11} * s_{21}, s_{12} * s_{22}, ..., s_{1D} * s_{2D})$) can classify the SNLI data alone. The fact that no component of $\vec{p}$ is able to classify the SNLI dataset does not mean that a mixture of them can't. So, we decided to use Support Vector Machines (SVM) [28], a supervised machine learning algorithm that can handle high-dimensional data. SVM find the hyperplane that divides the data at best. Since data could be divided by an infinite number of planes SVM finds those that maximize its distance with respect to both classes. A nonlinear division can be learned by mapping the data to a high dimensional space with a non-linear function, this technique is called Kernel Method [28]. [5] [6]

[5] For multiclass classification tasks, an hyperplane is found for every pair of classes.

[6] Note that SVM can be seen as improved perceptrons. [29]

We chose SVM as they obtain high performances when the number of features is large (we recall that SBERT and TFH output vectors of whose dimensions respectively are 768 and 512).

First, we want to set the SVM's input. In particular, we want input vectors to have 2 properties:

1. *We want to include the element-wise product $\vec{p}$ as part of the input*. As we said, the dot-product is an average among the $\vec{p}$ components, but some components, if properly combined, may be more relevat than others for the classification. We believe so, since the dot product proved capable of classification, even with low performance. Moreover, the dot-product is the metric used in the original SBERT paper [3] when performing Semantic Textual Similarity.

2. *We want an input that preserves the asymmetry of the problem*. The NLI task is asymmetric under sentences permutation, but the $\vec{p}$ vector is. So, we must enrich the inputs with an asymmetric vector.

For these reasons we decide to input the vector $\vec{p}$ concatenated with the difference vector $\vec{d} \equiv \vec{s_1}$ - $\vec{s_2}$. [7].

Furthermore, we notice that an input as [$\vec{p}$, $\vec{d}$] satisfies other two properties:

1. *Almost all the information that was in $\vec{s_1}$ and $\vec{s_2}$ is preserved*. If we only keep the $\vec{p}$ vector OR the $\vec{d}$ vector as input, a lot of information from the original $\vec{s_1}$ and $\vec{s_2}$ encodings is lost. The information is lost as we cannot reconstruct $\vec{s_1}$ and $\vec{s_2}$ from $\vec{p}$. Consider the i^{th} component of $\vec{p}$, p_i: there is an infinite set of pairs (s_{1i}, s_{2i}) such that s_{1i} x $s_{2i} = p_i$.
   Using both $\vec{p}$ AND $\vec{d}$ the set of possible input vectors $\vec{s_1}$, $\vec{s_2}$ that identifies [$\vec{p}$ and $\vec{d}$] is reduced from ∞^D to at most 2^D, with D the dimension of the encoding space. This is due to the fact that, for every pair of transformed components p_i and d_i, we have to solve the system of equations 2.2 to obtain s_{1i} and s_{2i}:

$$\begin{cases} s_{1i} = d_i + s_{2i} \\ s_{2i}^2 + d_i s_{2i} - p_i = 0 \end{cases} \tag{2.2}$$

EQUATION 2.2: System of equations to obtain s_{1i} and s_{2i} from their product p_i and difference d_i.

2. *NLI is a relation task, an input that is a relation between $\vec{s_1}$ and $\vec{s_2}$ may be beneficial*. Encodings are vectors whose positions are connected with the meaning. It has been shown [30] that, for some encodings, simple arithmetic relations stand. For example $\overrightarrow{king}$ - $\overrightarrow{male} \simeq \overrightarrow{queen}$ - $\overrightarrow{female}$. In the NLI task we want to classify the implication relation between sentences. If this relation is (partially) made explicit in the preprocessing of $\vec{s_1}$ and $\vec{s_2}$ the comparing algorithm may significantly benefit, learning similar relations for sentences.

Since nonlinear Support Vector Machines struggle with big data (Memory complexity scales as O(N^3), where N is number of data), but we saw that nonlinearity

[7] $\vec{s_1}$ and $\vec{s_2}$ are the TensorFlow Hub encodings of SNLI's sentences.

was essential to obtain accuracies over 80%, we decided to take the maximum number of training data that didn't saturate the RAM (3 x 10^4). So, we trained a SVM on SBERT encodings of 3 x 10^4 pairs randomly sampled from SNLI training set. We chose a gaussian kernel with γ = 0.002 and C = 50. On the SNLI validation set the accuracy achieved is 84.13%. This is an 18.74% accuracy improvement.

2.3.4 SVM improvements strategies: Nystroem approximation and MNLI dataset

The last model described shows an accuracy of 84.13% on the SNLI validation set. In order to furhter generalize this model we trained it including MNLI dataset [8] as well (as frequently done in literature [2])[8]. MNLI has 2 validation sets: one matched (with the same genres[9] as the train dataset) and one mismatched (with genres different from training dataset).

As previously stated, memory constraints prevents us from using the whole dataset (we used only 6% of the SNLI training data). The main problem in the space complexity of SVM is the nonlinear transformation i.e. the kernelization (that scales as O(N^3)). We tried to approximate this nonlinear map with the so-called Nystroem approximation [31]. Nystroem fixes the dimensionality of the nonlinear space, so less memory is required. [10]

By decreasing the number of components used in the Nystroem approximation, we can use more and more data. We did many explorative experiments with the Nystroem approximation (tried different hyperparameters, tried using Stochastic Gradient Descend on Linear SVM using all the data and tried different number of Nystroem components m).
We have developed a prototype model based on SBERT encodings of a mixture of SNLI and MNLI data (34K SNLI data and 136K MNLI data). The hyperparameters are m = 3000, C = 1, γ = 0.0003. With this model we obtain an accuracy of 72.66% on the SNLI validation set and of 70.15% on MNLI matched validation set.

Better SNLI and MNLI data sampling

Up to now, SNLI and MNLI data were randomly sampled when fewer data were used. With this method high-density clusters can be oversampled at the expense of single points and low-density clusters.

So, we used Ball Trees [32] to sample more frequently those points on low density volumes in the hyperspace where $\vec{p} \otimes \vec{d}$ vectors lay. More in detail, the Ball Tree algorithm constructs a tree where siblings leaf nodes are close in the high-dimensional space[11]. We then uniformly sampled 170K vectors, extracting the same number of samples from different siblings leaf nodes. We then trained a model as done in 2.3.4, but obtained comparable results.

[8] In the SNLI paper [4] there is the explanation of the dataset creation: sentences are created from photo descriptions. So, the SNLI sentences are text descriptions of only visual images. As the dataset is so visual-specific, we hypothesize it can be a cause of bad generalization.

[9] A genre is the origin of the dataset. It can be a telephone call, a letter 9/11 Report, ... There are 5 genres for train set and matched set, and other 5 for the mismatches set.

[10] Space complexity of the Nystroem approximation scales as O(Nm^2), with N number of data and m the dimensionality of the nonlinear space.

[11] Note that the Ball Tree algorithm is an approximate algorithm

2.3.5 Deep Learning end-to-end models

The space complexity of previous algorithms allowed us to use only $\sim$ 15% of the data available. On such dataset, we trained end-to-end Deep Learning models: models where both the Encoder and the comparing algorithms are Deep Learning layers.

We trained 4 different deep learning comparing algorithms: *Fully Connected layers*, *Convolution layers + fully connected*, *Fully Connected layers on single vector components* and a *Learnable generalization of the dot product*. All models were trained on both the SNLI and on the MNLI datasets. We remark that retraining all the models on those datasets make the encoder monolingual.

Fully Connected layers

The concatenated vector [**p**, **d**] is passed to a feed forward neural network composed by 6 different Fully Connected layers. The dimensions of inner vectors are 1024, 512, 256, 128, 64, 3[12]. Then, the softmax function is applied to the final 3D vector in order to obtain the classification probabilities. For all layers GeLU activation function is used as it is the activation function used in the BERT model [2]. Every end-to-end model can be trained on a 16 Gb Pascal GPU in approximately 7 hours. Based on the results of Ref. [2] we decided not to change the activation function and try different ones (ReLU, tanh, sigmoid, ...).

With this comparing algorithm we obtain an accuracy of 80.69% on SNLI and 77.00 % on MNLI mismatched.

In a Fully Connected layer each component of the input vector is used to compute each component of the output vector. All these interactions may not be necessary from the beginning and it may be beneficial to process the encodings $\vec{s}_1$ and $\vec{s}_2$ differently, before using the Fully Connected layers.

Convolution layers + Fully Connected

In 2.3.3 we described how we decided to process the encodings $\vec{s}_1$ and $\vec{s}_2$ before using them as the inputs of the comparing algorithm. The choices we made were based on informed reasonings but we wanted to test if a network could learn more relevant operations. So, we built a series of 1D convolutions that preprocess $\vec{s}_1$, $\vec{s}_2$, $\vec{p}$ and $\vec{d}$. The convolution operation we used is depicted in Figure 2.5.

[12] We tried both deeper networks and shallower ones. Deeper networks are heavier and do not produce higher accuracies and converge more slowly. Shallower ones start losing predictive power.

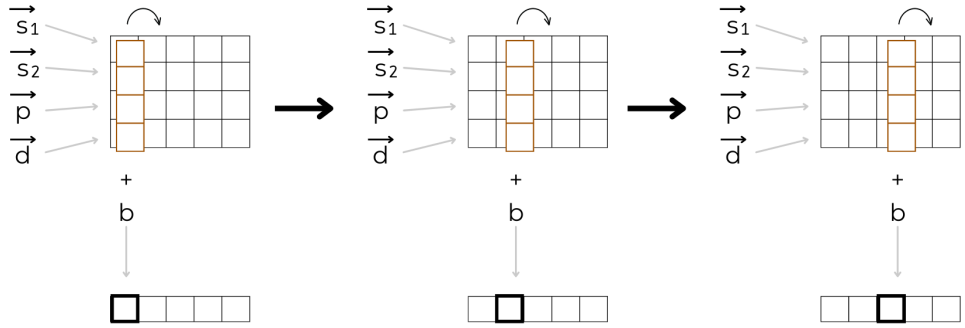

FIGURE 2.5: The convolution operation: $\vec{s}_1$, $\vec{s}_2$, $\vec{p}$ and $\vec{d}$ are stacked. Then a convolution consists in a 1D learnable matrix that is multiplied element-wisely column by column. A learnable bias term is added and an activation function is applied.

As described in previous Figure 2.5, for each pair of encodings a convolution matrix produces a vector with same dimensions as encodings' dimensions d_{model}. If C_1 convolutions are used, a $\mathbb{R}^{C_1 \times d_{model}}$ matrix is produced stacking the outputs. Then, the process can be repeated with a different number of convolutions C_2.

We noticed that a network with more than 3 convolutional sequences had convergence problems, in addition to an increased computation time. For this reason we decided to use $C_1 = 256$ and $C_2 = 1$. Then, we applied a series of FC layers of dimensions 512, 128, 64, 3.

With this network we we obtain an accuracy of 78.89% on SNLI and 75.37 % on MNLI mismatched.

Fully Connected layers on single vector components

The convolution works well when the input data present a translational invariance symmetry. This is due to the fact that the weight matrix is always the same, column independent. In our example different column may need to be processed differently. From this idea we tried to generalize a convolution applying a different weigth matrix for each column. This new operation is depicted in Figure 2.6.

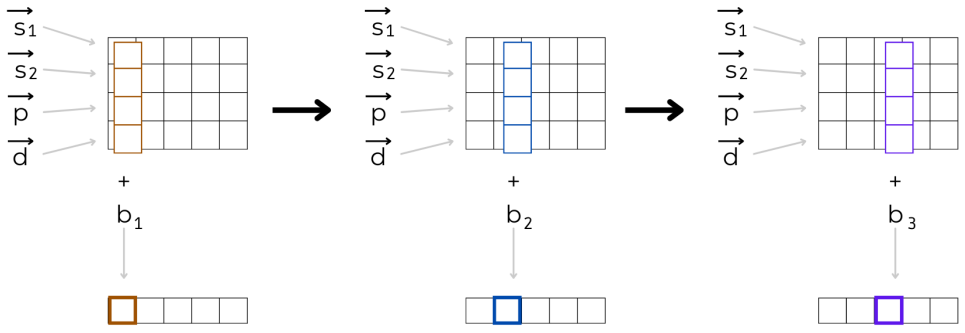

FIGURE 2.6: The convolution is generalized as the weight matrices and biases can be different. This can be seen also as a series of FC layers on each column.

Note that in Figure 2.6 the 1D weight matrix that is multiplied to the first column is different to the one that is multiplied by the second column. There is no convolution anymore.

Since in the computational graphs each node associated to a weight parameter is connected to fewer outputs, the gradient descend is more stable. We could use more generalized convolutions in series. In particular we used $C_1 = 16$, $C_2 = 16$, $C_3 = 8$, $C_4 = 4$, $C_5 = 2$, $C_6 = 1$. We then applied the usual series of FC layers of dimensions 512, 128, 64, 3. With this network we obtain an accuracy of 79.31% on SNLI and 75.02 % on MNLI mismatched.

Again, the accuracy decreased. We interpreted this as an indicator of the fact that it is important for different entries to be able to interact with one another. Furthermore we have a proof that processing the vectors $\vec{s}_1$ and $\vec{s}_1$ into $\vec{p}$ and $\vec{d}$ works better than making a network choose how to process them. This suggests that the properties used in 2.3.3 help preprocessing the encodings in the correct way.

Learnable generalization of the dot product

In the previous paragraph we showed that is important for different components to interact. So, we tried to generalize the dot-product with a learnable matrix. More in detail we can write the dot-product between $\vec{a} \in \mathbb{R}^{1 \times d_{model}}$ and $\vec{b} \in \mathbb{R}^{1 \times d_{model}}$ as:

$$< \vec{a} | \vec{b} >= \vec{a}\, \mathbb{1}\, \vec{b}^T$$

It is clear that the identity operator does not make any interaction between different components possible. So, we decided to try the following map:

$$\vec{a} \to [\vec{p}, \vec{d}] \tag{2.3}$$

$$\vec{b} \to [\vec{p}, \vec{d}] \tag{2.4}$$

$$\mathbb{1} \to \mathbb{P}, \text{ a parameter matrix.} \tag{2.5}$$

We then use the scalar obtained with this procedure as a classifier for the NLI task. We obtain an accuracy of 79.55% on SNLI and 75.74 % on MNLI mismatched. This result shows how a simple all components to all components interaction can be used to achieve high accuracies.

In Table 3.3 we group the results of all models previously described.

Each test described in the previous chapters helped us build a better model step by step. We obtained better and better results leveraging symmetries, geometric intuitions and literature. We have also developed a webapp to make the model developed in 2.3.4 available for non-technical users. We are developing an end-to-end model for Italian language via knowledge distillation [26]. [13]

[13] Knowledge distillation is a technique used to train a new model (student model) on a training dataset labelled by another model (teacher model). The teacher model we are using is the one developed in 2.3.5 - Fully Connected layers.

Chapter 3

Results

In this chapter, we present our results, i.e. the performance of the models developed - as they are at the current stage - and discuss the tests we developed to infer these performance.

3.1 Test datasets

3.1.1 English datasets: SNLI & MNLI

Both the SNLI [4] dataset and MNLI [8] dataset are NLI datasets. The detailed description of the Natural Language Inference task can be found in Sec. 1.1. Here we describe the test set for both datasets.

SNLI

SNLI has the canonical train-validation-test split of the dataset. The train set is used to train the models, for each trained model an accuracy is calculate on the validation set and these accuracies are used to choose the best model. Since the train and validation sets are used to choose the best model it may be possible that the model specialized on those data. In order to have an accuracy that is meaningful, the test set is used.
For the SNLI dataset we have simply used the SNLI test set.

MNLI

MNLI does not have the standard train-validation-test split of the dataset. Differently from the SNLI dataset the MNLI datasets are divided into genres. A genre is a particular source of the data. For example, if a sentence was extracted from a letter the genre is "Letters".
The train and matched validation sets have the same gentres: "Telephone", "Fiction", "Government", "Slate" and "Travel". For the mismatched validation set, the genres are different: "Letters", "Oxford University Press", "Face-to-face", "9/11 reports" and "Verbatim".
Since no test set is available for MNLI, the mismatched validation dataset is used to evaluate the generalization power of the model. A high accuracy on the mismatched set corresponds to a better generalization of the model.

3.1.2 Italian datasets: RTE-3 & ABE_ABSITA

We identified two open-source Italian datasets and the relative evaluating procedure to estimate how our models generalize to Italian and to free-text information extraction.

RTE-3

RTE-3 for Textual Entailment in Italian: an NLI Italian dataset. Unluckily the official website is down, but the dataset can be downloaded here: https://github.com/gilnoh/RTEFormatWork/tree/master/RTE3-ITdata-original-format/RTE3-ITA_V1_2012-10-04[1]

This dataset is composed of only two classes: *Entailment* and *Not Entailment*. Since our model produces 3 classes (*Entailment*, *Neutral*, *Contradiction*) we map the outputs in the following way:

$$\begin{aligned} \textit{Entailment} &\longrightarrow \textit{Entailment} \\ \textit{Neutral} &\longrightarrow \textit{Not Entailment} \\ \textit{Contradiction} &\longrightarrow \textit{Not Entailment} \end{aligned}$$

We chose this mapping as it is the one that maximizes the accuracy on the validation set.

ABE_ABSITA

ABE_ABSITA [5] is Subtopic-level Sentiment Analysis (SSA) dataset in Italian: A corpus of Italian hotel reviews that can have multiple topics (Price, Cleanliness, Staff, Location, ...) and sentiments for each topic (sentiment can be different for different topics in the same sentence). This dataset is useful to test our models on different tasks. Our aim is to build a general architecture that can be used for different tasks. On this dataset we can test our models on 3 different tasks:

1. **Topic recognition**: recognize if a given sentences talks about a topic.
2. **Aspect-based Sentiment Analysis**: Given a sentence with a certain topic, recognize the sentiment for that topic.
3. **General Sentiment Analysis**: Recognize whether or not the overall sentiment is positive.

For each task we used the following ground truths[2] as $sentence_2$ of the NLI task:

Task	Ground truth	English translation
Topic Recognition	"Parlo di pulizia"	"I'm talking about cleanliness"
Aspect-based	"La camera è pulita"	"The room is clean"
General	"Sono soddisfatto"	"I'm satisfied"

TABLE 3.1: Ground truths used fro different ABE_ABSITA tasks.

Since each task is a binary classification task we mapped the 3 outputs of our model in the following way:

[1] We weren't able to find a human labelled NLI datset for the Italian language, se we used RTE.

[2] Remember that a ground truth is a query sentence that, when used with a NLI model, can give information on other sentences. Look at Sec. 1.5 for clarification

Task	3-class output ⟶ task output
Topic Recognition	Entailment ⟶ Entailment Neutral ⟶ Entailment Contradiction ⟶ Contradiction
Aspect-based Sentiment Analysis	Entailment ⟶ Entailment Neutral ⟶ Contradiction Contradiction ⟶ Contradiction
General Sentiment Analysis	Entailment ⟶ Entailment Neutral ⟶ Entailment Contradiction ⟶ Contradiction

TABLE 3.2

For topic recognition and Aspect-based Sentiment Analysis we focused only on detecting and classifying the "Cleanliness" topic. There is no particular reason for this choice.

In the tests we made we used only 1 ground truth. All the accuracies described before were achived using only 1 ground truth. A unique ground truth may produce biased and inaccurate results. So, we are currently testing a more Ground Truth independent inference scheme that uses 5 sentences. For each task and for each sentence in the dataset we use 5 Ground Truths to produce an output. In order to have an unique label we implemented the following voting system: If one of the output is *Entailment* then we label the sentence as *Entailment*, otherwise label as the majority between *Neutral* and *Contraddictions*. Then label is mapped to one of the 2 classes using the mapping described in Table 3.2.

3.2 Results

In Table 3.3 we summarize the result that were shown in a scattered way in Chapter 2 and the results on the Italian datasets.

Encoder	Comparing algorithm	Dataset	Validation accuracy	RTE-3 val accuracy	ABE_ABSITA val accuracy
SBERT	dot product	SNLI	57.87^{A}%	–	–
TFH	dot product	SNLI	50.13^{A}%	–	–
TFH	dot product with fine-tuning	SNLI	65.39^{A}%	–	–
SBERT	SVM	SNLI (N = 30K)	84.13^{A}%	(in progress)	(in progress)
SBERT	Nystroem (m=3000) + Linear SVM	SNLI (N=34K) + MNLI (N=136K)	72.66^{A}% 70.15^{B}%	63.38%	67.00^{C1}% 86.00^{C2}% 76.00^{C3}%
SBERT	FC layers	SNLI + MNLI	80.69^{A}% 77.00^{B}%	(not doable)	(not doable)
SBERT	Convolution + FC	SNLI + MNLI	78.89^{A}% 75.37^{B}%	(not doable)	(not doable)
SBERT	FC layers on single components	SNLI + MNLI	79.31^{A}% 75.02^{B}%	(not doable)	(not doable)
SBERT	Generalization of dot-product	SNLI + MNLI	79.55^{A}% 75.74^{B}%	(not doable)	(not doable)

TABLE 3.3: Summary of the main models described in this thesis. All the models are the ones described in Chapter 2. ^{A}: SNLI validation, ^{B}: MNLI matched, ^{C1}: Topic recognition task, ^{C2}: Aspect-based Sentiment Analysis task, ^{C3}: General Sentiment Analysis task.

In Table 3.3 a validation accuracy is calculated on the MNLI matched validation dataset.

It may seem that the best performing model is the one that reaches an accuracy of 84.13% on the SNLI validation accuracy. But, as we already discussed in Sec. 2.3.4 this model was neither trained nor validated on the MNLI dataset. For this reason it is a SNLI specific model that is not general on the NLI task. The most general multilingual model we developed is the one that uses the Nystroem approximation and the Linear SVM. With that model we calculate the test accuracies. These accuracies are the one we expect to have on real-world data. We obtain an accuracy of 72.30% on SNLI test set and 70.52% on MNLI mismatched.

On the RTE-3 dataset we achieved an accuracy of 63.38%. This is a $\sim$ 7% drop from the MNLI results. Since RTE-3 is almost equivalent to a NLI task in italian, we attribute this 7% gap to a problem in the multilingualism of the Encoder.

On ABE_ABSITA we obtained different results. For the Topic recognition tasks and the General Sentiment Analysis task we obtained, with the best multilingual model, respectively 67% and 76%. The score on topic recognition may appear poor but still is acceptable fpr industrial purposes since accuracy may be practically increased by adding ground truths as described in Sec. 3.1.2. The 86% accuracy for the Aspect-based sentiment analysis shows that more specific queries leads to more accurate results.

Chapter 4

Conclusions and perspectives

4.1 Conclusions

In this work we built several models showing that it is possible to obtain high accuracies (over 84% on the SNLI dataset) on the NLI task with an sentence-Encoder-based architecture folowed by a comparing algorithm. We showed that, even if the train dataset for the SVM is in English, it is possible to generalize it to other languages, such as italian, as long as the Encoder is multilingual.

We also showed that various NLP problems (e.g. the tasks derived from the ABE_ABSITA dataset) may be mapped mapped into a NLI task. After this map we showed how we could perform sentiment analysis on a specific subtopic ("cleanliness"), obtaining an accuracy of 86%. We also obtained acceptable accuracies (67% and 76%) on other tasks (respectively, Topic Recognition and General Sentiment Analysis). In this way, we empirically proved the generality of the NLI task. We stress that all these results were obtained without retraining the model on the specific task. This allows us to make many explorative analysis when lacking task-specific datasets. Moreover, we may avoid the time and financial costs of a task-specific training.

We showed that, with the same comparing algorithm (dot product), the SBERT encoder gives more accurate results than the TFH one. We showed how both models can be improved with a fine-tuning on the comparing algorithm. However, this process makes the Encoder monolingual and should be limited to the target language or coupled with knowledge distillation [26] for task translation to the target language.

We showed how the best accuracies were achieved by pre-processing the inputs of the comparing algorithm in a geometric-informed way (keeping a dot-product-based input and exploiting the asymmetries of the NLI task): Instead of using the encoded sentences ($\vec{s_1}$ and $\vec{s_2}$), we showed the validity of pre-processing them by calculating their element-wise product ($\vec{p}$) and their difference ($\vec{d}$).

Using different end-to-end Deep Learning models we then obtained the most accurate algorithm on the NLI task (both the SNLI and MNLI accuracies raised of over 6%). Furthermore, comparing the different Deep Learning comparing algorithms we confirmed that the informed pre-processing we made (transforming the encodings $\vec{s_1}$ and $\vec{s_2}$ into the element-wise and difference vectors $\vec{p}$ and $\vec{d}$) to the encodings were solid. Indeed, when we let a simple network decide, that pre-processing resulted in worse accuracy (Sec. 2.3.5 - Fully Connected layers on single vector components).

We also showed that after the pre-processing it is useful to make different components of the input interact without any symmetry exploitation. This fact reflects the asymmetry of the encodings: different entries may encode different information that needs to be processed in different ways.

4.2 Perspectives

Up to now the best model working for Italian is the one using a Support Vector Machine (line 5 of Table 3.3). In order to build a better model for Italian we can follow 2 paths.

Use knowledge distillation on the best end-to-end Deep Learning model

When we fine-tunined all the Deep Learning end-to-end models we transformed the multilingual Encoder into a monolingual one, as both SNLI and MNLI are English datasets.

We have already seen and studied proper techniques to transfer learning knowledge from Monolingual Encoders [26] and how to properly tune some hyperparameters. For example, the authors of [33] described how to use a Kullback–Leibler divergence loss function for high performances in knowledge distillation.

We are now developing a model for the Italian language with these techniques. Then, these models should be tested on the public Italian datasets described before.

Use a better end-to-end model

We have seen that the best models so far on the SNLI and MNLI datasets are the end-to-end Deep Learning ones. We have tried 4 comparing algorithms but some improvements may be added.

The simplest improvements are those that change the hyperparameters of the existing network. For example it would be interesting studying what different learning rates produce, different parameters in the optimizer, and so on. Another interesting perspective is to use a different Encoder. We tried the SBERT encoder and the TFH one, but many more are present and, as more time passes, better models are available. In order to achieve better results more modern architecture should be considered.

List of acronyms

ABSITA Aspect-based Sentiment Analysis for Italian
ABSA Aspect-based Sentiment Analysis
AI Artificial Intelligence
ATE Aspect Term Extraction
BERT Bidirectional Encoder Representations from Transformers
CNN Convolutional Neural Network
DL Deep Learning
FC Fully Connected
LSTM Long Short Term Memory
MNLI Multi-genre Natural Languange Inference
NN Neural Network
NLI Natural Languange Inference
NLP Natural Languange Processing
RNN Recurrent Neural Network
RTE Recognizing Textual Entailment
SBERT Sentence-BERT
SNLI Stanford Natural Languange Inference
STS Semantic Textual Similarity
SVM Support Vector Machines
TFH TensorFlow Hub

Bibliography

[1] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[2] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[3] Nils Reimers and Iryna Gurevych. "Sentence-bert: Sentence embeddings using siamese bert-networks". In: *arXiv preprint arXiv:1908.10084* (2019).

[4] Samuel R Bowman et al. "A large annotated corpus for learning natural language inference". In: *arXiv preprint arXiv:1508.05326* (2015).

[5] Pierpaolo Basile et al. "Overview of the EVALITA 2018 Aspect-based Sentiment Analysis task (ABSITA)". In: *EVALITA Evaluation of NLP and Speech Tools for Italian*. CEUR. 2018, pp. 1–10.

[6] Daniel Cer et al. "Semeval-2017 task 1: Semantic textual similarity-multilingual and cross-lingual focused evaluation". In: *arXiv preprint arXiv:1708.00055* (2017).

[7] Mandar Joshi et al. "Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension". In: *arXiv preprint arXiv:1705.03551* (2017).

[8] Adina Williams, Nikita Nangia, and Samuel R Bowman. "A broad-coverage challenge corpus for sentence understanding through inference". In: *arXiv preprint arXiv:1704.05426* (2017).

[9] Bin Ding, Huimin Qian, and Jun Zhou. "Activation functions and their characteristics in deep neural networks". In: *2018 Chinese control and decision conference (CCDC)*. IEEE. 2018, pp. 1836–1841.

[10] Katarzyna Janocha and Wojciech Marian Czarnecki. "On loss functions for deep neural networks in classification". In: *arXiv preprint arXiv:1702.05659* (2017).

[11] Pankaj Mehta et al. "A high-bias, low-variance introduction to machine learning for physicists". In: *Physics reports* 810 (2019), pp. 1–124.

[12] Robert DiPietro and Gregory D Hager. "Deep learning: RNNs and LSTM". In: *Handbook of medical image computing and computer assisted intervention*. Elsevier, 2020, pp. 503–519.

[13] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[14] Denny Britz et al. "Massive exploration of neural machine translation architectures". In: *arXiv preprint arXiv:1703.03906* (2017).

[15] Yonghui Wu et al. "Google's neural machine translation system: Bridging the gap between human and machine translation". In: *arXiv preprint arXiv:1609.08144* (2016).

[16] Philip Gage. "A new algorithm for data compression". In: *C Users Journal* 12.2 (1994), pp. 23–38.

[17] Taku Kudo. "Subword regularization: Improving neural network translation models with multiple subword candidates". In: *arXiv preprint arXiv:1804.10959* (2018).

[18] Taku Kudo and John Richardson. "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing". In: *arXiv preprint arXiv:1808.06226* (2018).

[19] Kaj Bostrom and Greg Durrett. "Byte pair encoding is suboptimal for language model pretraining". In: *arXiv preprint arXiv:2004.03720* (2020).

[20] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[21] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[22] Mor Geva et al. "Transformer feed-forward layers are key-value memories". In: *arXiv preprint arXiv:2012.14913* (2020).

[23] Jane Bromley et al. "Signature verification using a" siamese" time delay neural network". In: *Advances in neural information processing systems* 6 (1993).

[24] Alex Wang et al. "GLUE: A multi-task benchmark and analysis platform for natural language understanding". In: *arXiv preprint arXiv:1804.07461* (2018).

[25] Aakanksha Chowdhery et al. "Palm: Scaling language modeling with pathways". In: *arXiv preprint arXiv:2204.02311* (2022).

[26] Nils Reimers and Iryna Gurevych. "Making monolingual sentence embeddings multilingual using knowledge distillation". In: *arXiv preprint arXiv:2004.09813* (2020).

[27] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[28] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine learning* 20.3 (1995), pp. 273–297.

[29] Stephen I Gallant et al. "Perceptron-based learning algorithms". In: *IEEE Transactions on neural networks* 1.2 (1990), pp. 179–191.

[30] Kawin Ethayarajh, David Duvenaud, and Graeme Hirst. *Towards Understanding Linear Word Analogies*. 2018. DOI: 10.48550/ARXIV.1810.04882. URL: https://arxiv.org/abs/1810.04882.

[31] Christopher Williams and Matthias Seeger. "Using the Nyström method to speed up kernel machines". In: *Advances in neural information processing systems* 13 (2000).

[32] Stephen M Omohundro. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[33] Taehyeon Kim et al. "Comparing kullback-leibler divergence and mean squared error loss in knowledge distillation". In: *arXiv preprint arXiv:2105.08919* (2021).