ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA
CAMPUS OF CESENA

School of Engineering
Master's degree in Computer Science and Engineering

JOB RECOMMENDATION BASED ON DEEP LEARNING METHODS FOR NATURAL LANGUAGE PROCESSING

Project in
Text mining

Rapporteur
Prof. Gianluca Moro

Presented by
Lorenzo Valgimigli

Third Deegre Session
Accademic year 2018-2019

KEY WORDS

Natural Language Processing

Recommendation System

Deep Neural Networks

Job Embeddings

Python

Index

List of Figures

Chapter 1

Introduction

From the very beginning of the history human kind had always tried to figure out how to solve the problems they had to face every day learning from experience, improving their intelligence and their skills. History tells us how they handle them, how they grew improving technology, techniques and personal skills but hard to solve problems still exist.

Today, different from the past, we have more tools and more experience to handle such problems and to try to find solutions. One of the most important tools is the computer that has lead us to new solutions and new ways to approach them. The computer has worked fine to face problems which can be rigorous and formally described and for years it has been used to handle these kinds of tasks. However, there are problems which humans can face more or less easily but they can't be described using rigorous ways. Let's think, for examples of a man or woman's face and the goal is to guess if it is smiling or not. This task is incredibly easy for anyone, but it is impossible to describe with some rigorousness. For the human mind it is simple to face those kind of problems, but for a such schematics technology as computers it is impossible.

To face them **Artificial Neural Networks** were born. ANNs are inspired to simulate the human brain and how it works. Indeed, they use as basic unit the **Neuron** that, like in the brain, is linked to more neurons and receives inputs from other neurons. Another similarity between the human mind and ANN is that both learn from experience but in the first case the experiences are limited and it must learn fast from them. The second case is different in the fact that it can live a lot of experiences in a few seconds. This makes the ANN very powerful and, for some tasks, they surpass humans by reaching better results. From the first promising results Neural Networks has been deployed in many fields: Natural Language Understanding, Events Prevision, Medical Diagnostic System, Products Recommendations and so on.

To face more complex problems Artificial Neural Networks has been modified

by adding inner Neuron Layers to let them model those problems better and better. These modified artificial neural networks are called **Deep Neural Networks** and, nowadays, they represent a very promising solution for tasks that before were impossible to solve.

1.1 Natural Language Processing

Natural Language Processing is the point where the linguistic field and computer science come together. It concerns all methods, algorithms and tools used by software to process and extract information from human language data. This field includes different tasks as **Natural Language Understanding**, **Speech Recognition** and **Natural Language Generation**.

For humans understanding their own language is easy and natural, but it is a very hard task for a computer that doesn't have the ability to deeply understand the semantic of a word. In the human mind exists a special dictionary that links a word to its meaning composed by experiences and sensations. If we read the word ice-cream we get access to a lot of personal information like last time we tasted it or our favourite flavours or ice-cream shop and all of these well define the concept of ice-cream. But computers don't have such a pool of experiences and this in not the only problem they face. In fact, this field raises other issues:

- **Syntax**. Syntax structure differs from language to language
- **Polysemy**. Words can have different meaning according to their usage.
- **Synonymy**. Different words have the same meaning.
- **Irony**. Sentences written to have a meaning that has to be extracted form the sentence indirectly.
- **Orthographic errors**. If sentences are hand-written they can contain errors to be corrected.
- **Abbreviations** or **Special Symbols** as the emojis. If the aims are tweets abbreviations and special symbols are very common and they have important meaning within the phrase.

To handle all of these problems data preprocessing is a common way to proceed. This phase is composed by well-known steps which aim to clean the text, remove synonymy and errors. These steps can vary from task to task but the most common ones are:

- **Case folding**. All words are turned in lower or upper case.
- **Segmentation**. Sentences are split in lists for the composed words.
- **Normalization**. This step aims to reduce the dictionary size using technique as **Lemmatizations** or **Stemming**.

After this phase data appears divided into basic units called **Tokens**, but to teach a computer the meaning of these tokens is still a problem to face. The basic idea is to create relationships between words looking at the ones that come before the target word and the ones that follow it. To better understand this concept it necessary to define the Context of a word C_{w_k} as the set of the N words before w_k and the M words after.

$$C_{w_k} = [w_{k-n}, \ldots, w_{k-1}, w_{k+1}, \ldots, w_{k+m}]$$

In this way for each word (or token) it's possible to create a context and then join the contexts of the same tokens which have more recurrences and lastly list those tokens which appear several times. For each word, looking at its context, a list of features can be created using the frequencies of the presence of other words in its context. In this way two words could have a relationship if they share the same or similar context. For example the word *ice-cream* and the word *cold* appear to be related in some way because the second is often in the context of the first. This leads computer to create relationships between words and these relationships are the keys to understand the meaning of them.

1.1.1 Word Embeddings

Using this context is also possible, for each word, to create a vector of features composed by real numbers that describes the word itself. This vector is called **Word Embeddings** and maps the word into a vector space. An intuitive way to create such a vector could be by using a matrix with words in rows and columns. Cell c_{ij} contains a natural number that indicates how many times the word in the column wj appears in the context of the word in the row wi. This kind of matrix is known as **Co-occurrence Matrix**. In this way for each word in the dataset it possible to create a high dimensional vector that brings information about relationships with other words. At this point, using some advanced techniques as **Singular Value Decomposition** each vector is mapped to another vector with less features but bringing under light relationships between words (Latent Semantic Analysis). This phase is called **Dimensionality reduction** and aims to create a vector space where

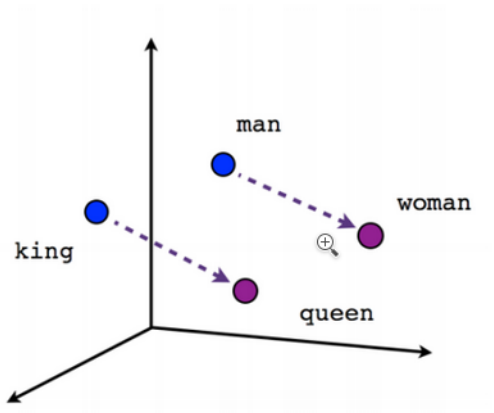


Figure 1.1: Vector space representations. In the figure is possible to see how the relationships between words are contained within the boundaries of this space

similar words are put close together while different words are placed at some distance. A graphical view of this concept can be found at figure 1.1.

The above explained process is just the basic concept behind the algorithms used in Natural Language Understanding tasks. It is just an intuition of a more complex word, that is the ground on which, works by many researchers has been built. One of this work highlights the importance of studying the frequency of the term in a document in relation to the frequency of that term in the whole corpus [5]. The study of that frequency is fundamental aspect for the Word Embedding creation.

Word Embeddings are the key to solving NLP problems and to create good ones is still an on going task and a lot of techniques have been created to further improve them.

1.2 Deep Neural Network in NLP

In today's Deep Neural Network era a lot of old techniques have become useless. The Deep Neural Networks bring with them some advantages which make these new models the most favored in many fields of data science. First of all, they don't need a strong formalism to describe a problem but a lot of examples (x, y) where x is the input of the net and y is the desired output to train the network. Thanks to the new technologies and the coming of **big data**, Deep Neural Networks achieved exactly what they wanted and they were

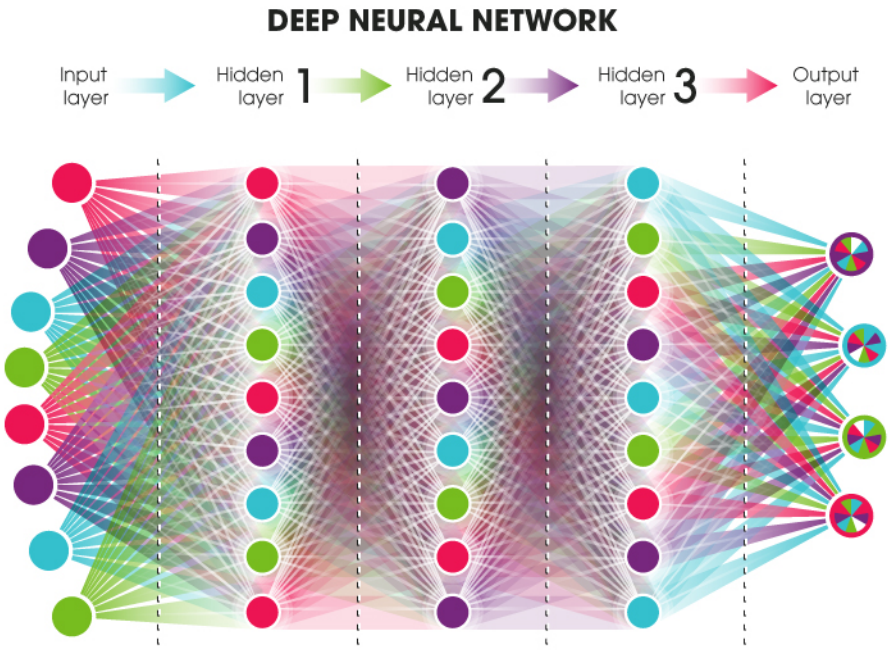


Figure 1.2: Deep Neural Network overview

able to expand limitlessly, leading research into unexplored territory. Today they are used to reach state of the art performance, growing day by day, ever improving their results. They are also used in concrete fields like market. One example is the work done by Professor Gianluca Moro regarding how to predict the stock market Dow Jones Index using Tweets [6].

For each data science field a specific Neural Network exists as **Recurrent Neural Network** for data represented by sequence like sentences or **Convolutional Neural Network** for matrices of data and they are widely used in Imagine Processing and so on. The following paragraph will try to explain how deep neural network changed the NLP field.

1.2.1 New Word Embeddings

The basic and most important element for each Natural Language Processing task is word representation in a vector space that shows the relationships between words. This representation is called **Word Embedding** and the creation of this is a very well studied topic by lot of researchers.

A good way to generate these vectors comes from Deep Neural Networks. The main ingredient to generate them is the Context of a word C_{w_0} so Deep Neural Networks are trained to recreate the right word belonging to a specific

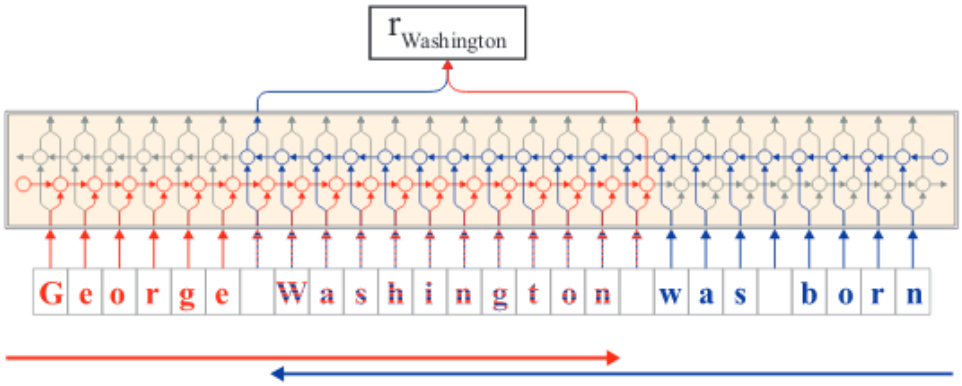


Figure 1.3: Intuitive representation of a model that extracts contextual word embeddings from a sentence. *Figure from Contextual String Embeddings Paper* [7]

context. For example, masking a word in a sentence and the model has to read the words prior to it and after it and then generate a probability distribution over the dictionary to guess the masked word. Using a large dataset (more than 100 GB) and a lot of training time the model can achieve very good results. This kind of approach forces model to learn relationships between words, to guess which words are nearer to others and how often they occur, but without creating any explicit data structure with this information. In this way, the network learns internally how to model the language. As we know, Deep Neural Networks are by definition, Multi-layered Neural Networks and each layer has a hidden state composed by the state of each neurons. Using these hidden states, which somehow contain what model learned about the language, it is possible to create vectors to use as word embeddings. This is the basic idea on how to use Deep Neural Networks to generate word embeddings but each implementation brings differences and variations which need to be studied.

1.2.2 Contextual Word Embeddings

Using the previously explained approach, the model can read the words prior to the target word and the ones after it and therefore generate a probability distribution over the entire dictionary according to what it has seen.

Let's define $X_{0:T} = (x_0, x_1, \ldots, x_T)$ the language dictionary, w_j the masked word and $C_j = (w_0, \ldots, w_{j-1}) \cup (w_{j+1}, \ldots, w_n)$ its context. So the model tries to predict the right probability distribution over $X_{0:T}$.

$$P(X_{0:T}|C_j)$$

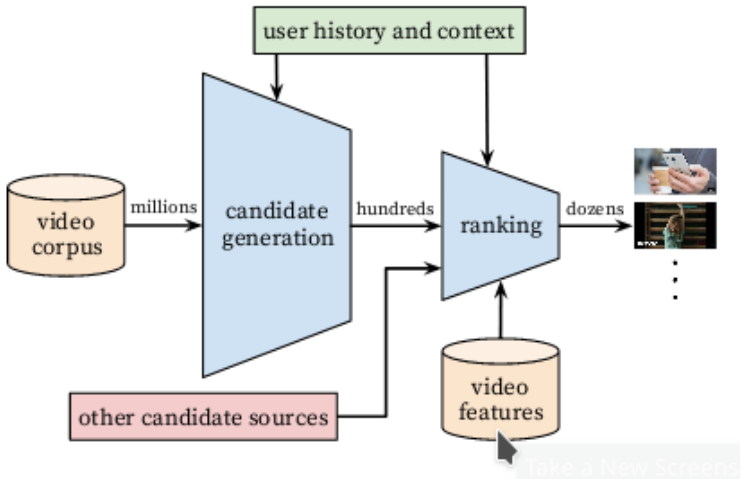


Figure 1.4: Picture from paper *Deep Neural Networks for YouTube Recommendations*[1]. High level representation of the Neural Networks used by You Tube to recommend videos

From this description it's easy to see that the context heavily influences the result of the model. This means that the same word in a different sentence, generates different word embeddings. Using old approaches each word was mapped to a unique vector. With this new technique, however, each word gets a vector strongly correlated to the context and such vectors are called **Contextual Word Embeddings**. This allows Deep Neural Networks to create better embeddings and reach a new state of the art performance in this field.

1.3 Recommendation Systems

A **Recommendation System** or **Recommender System** is a technology that seeks to predict the preference a user would give to an item in the system. For example **You Tube Recommendation System** [1] is a Neural Network model that tries to predict the favourite videos in the corpus for each user. Improving and deploying recommendation systems are often the main objectives of global companies such as **Amazon**, **Netflix**, ... It's a fundamental solution to enhance the user's experience especially when considering the number of available items grows day by day and for this reason much research and study goes into this field.

1.3.1 Collaborative Filtering

There are two main approaches for the design of a recommendation system: **Collaborative**[8], **Content-Base**. In the first case the idea behind it is: two users with similar histories will probably make similar decisions. In this system the user is defined by a series of items $e_0, e_1, \ldots, e_n \in E$ that is to say that the collection of the previously selected items like videos viewed on You Tube, films watched on Netflix or items purchased on Amazon. If two users have a similar items history, this approach assumes that if item e_k will be the next choise for the first user it will probably be the next choise even for the second user. Matching user items histories the system tries to suggest items with the highest probability of being selected.

1.3.2 Content-Based Filtering

The second approach is based on the description of the item to profile the user preferences. The idea behind it is: the user that has been using the system to buy food and not electronic devices will probably keep doing so in the future. The system will then recommend pizzas rather than smart phones. Also in this case the user is defined by items he previously selected. The model, using these items and the correlating data, tries to create a **User-specific Classifier** that has to determine whether for each item the user will like it or not. Usually, Recommendation Systems incorporate both of these approaches to achieve the best results[9].

1.4 This project

The goal of this project is to test new technologies as Deep Neural Networks tries to manage problems that classic machine learning algorithms can't. One of these problems is how to interpret textual data, maybe hand-written, in tasks which aren't NLP tasks. In particular this work focuses on job recommendation systems where data is not well structured and most of them are hand written by users. In fact, the entire job history is written by each users using his terminology, his descriptions, his detail level, ... It can be seen immediately this kind of approach brings a lot of problems:

- Two different jobs can be addressed with the same name. For example a medical assistant or a university professor assistant can be referred using just *Assistant* but they are very different in particular if the system has to model concepts such as *Careers*.

- The same job can be named differently. Two Java developers can refer to themselves using terminologies as *Java Developer*, *Java Programmer* or *Object Oriented Programmer using Java*. So the same job can lead to a erroneous representation of the job histories of two users because they are written in different ways.

- Not all users express themselves with the same degree of verbosity. Some can use informative job titles adding extra information while others can do the opposite using short and meaningless job titles. This means the model's approach is to focus where there is more information and ignore those where there is less.

If the goal is to create a model capable to work with this data it is necessary that it can read and understand the sequence of the jobs in a job history. It has to model complex data and work with them. The idea behind this project is to create a system that takes the job history and perhaps some further information about the user and therefore predict the last job of this user. It's impossible know which is the best job for each user so this project is based on this assertion: the last job of a user was one of the best jobs for him before doing it. Let's define user u as a list of jobs that he did until time t: $u_0 =< j_0, j_1, ..., j_n >$. So at the given time $t-1$ user is $u_0 =< j_0, j_1, ..., j_{n-1} >$. In this case it is possible to assume that the job j_n is one of the best jobs for him because it is the one he did next. However, guessing the last job is one of the many possibilities: in fact the data also contains job posting and job applications for each user and other information such as level of education, time, location, ... All of them compose a variegate picture that brings possibilities and problems. So the question still remains: can Deep Neural Networks make use of these opportunities by overcoming problems?

Chapter 2

Domain of the project

The project was born from recent European interesting in the field of the Job Recommendation. Some projects in this direction have been financed by governs and by private companies in the last few years highlighting how much important is this aspect. Find the best worker for a given jobs or find the best job for a worker is the key point for an healthy working system that the big companies with thousand of employs know well. Today, part of this work relies on personal specialized in finding the best workers in the market, meeting them and selecting a few of them according to little information. This is a very expansive mechanism that doesn't guarantee the best results. Therefore one question arises: how new Deep Neural Networks Technologies can be applied to this field and what results they can achieve?

2.1 Job Recommendation System

The efficiency of the Artificial Neural Networks and the Machine Learning Models in Recommendation Field is not deniable. Today many systems relies on this kind of technology to improve the User Experience on one side and to optimize the server computational cost on the other. People interact with this system every day even without knowing it. It is enough to think about Amazon.com Recommendation System that uses a **Item based collaborative filtering**[10] to recommend items to each user according to his past history. Similar examples can be found everywhere in the web, but all of those systems use well structured data which have a very accessible and explicit information. Automatize these kinds of tasks has been a target for many researchers for years. Job seeking was tricky, tedious and time consuming process because people looking for new position had to collect information from many different sources. To automatize this process*Job Recommendataion systems* were proposed. They have been used and improved year after year with the best machine learning

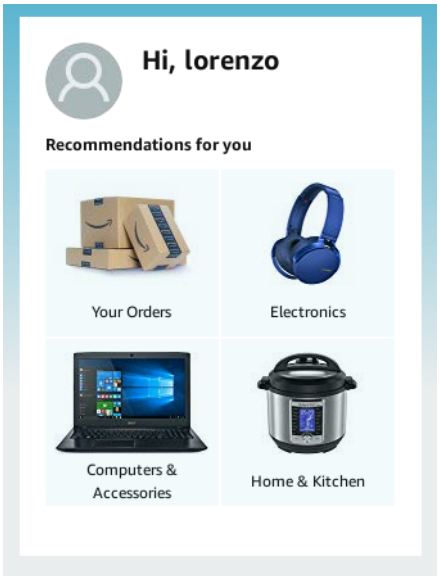


Figure 2.1: Recommendations on Amazon

algorithms [11].

But now, the question is how Deep Neural Networks can be applied in Job Recommendation tasks even if data are unstructured. First of all it is useful, in order to understand the problem, to formally describe the task. Let's define a worker w_k as a user u_k with all information about his working history. This information can vary from dataset to dataset but they often are:

- **Jobs History**. This is probably the most important information about the user because it contains all jobs done by the user. Sometimes this job history comes with other useful information for each job like salary, duration, rating, ...
- **Education Level**. This information allows the system to better classify the user. Often it contains information as final vote, graduation, extra experiences, ...
- **User's preferences**. The user can indicate some preference on which field he would like to work. Sometime, this information extrapolated by using user's researches.
- **User's skills**. User can select or write some skills he thinks to have. This information can be useful to pair the user with a job that requires explicitly or implicitly some skills.

Now let's define a company c_k as the element in the system that produces job postings. Each company can have a description about itself and some other

textual information but, the key points are the job postings it produces. They are defined as $j \in J$ and they have some information as:

- **Description of the position**. This could be a textual information about the job and it gives a brief introduction about duties and tasks employ has to fulfill.
- **Structured information**. These are a series of information like salary, location, work hours which help to formally define the boundary of the job.
- **Hard and soft skills**. Job posting can indicate which skills are required (Hard Skills) and which are welcome (Soft Skill). This information can help the system to find the right worker for this job posting.
- **Experience required**. This information tells how many years and which kind of experience the candidate should have. It can be used to filter a huge quantity of candidates.

Once job postings $j \in J$ and user $u \in U$ are defined it is necessary to define a function of satisfaction $S(j, u) \rightarrow s$ that, for each pair user u_i and job posting j_k, produces a real number that indicates how good is the user for the selected job. The perfect Job Recommender System JRS is the one that is capable of generating the pairs (j_k, u_i) that maximize the function S. Let's define $D_{KX2} \in \mathbf{D}$ as the matrix that contains into the rows the pairs (j_k, u_i) and $\mathbf{D}$ is the space of all possible matrices D. JRS has to generate D_{KX2_n} that:

$$S(D_{KX2_n}) >= S(D_{KX2_i}) \forall D_{KX2_i} \in \mathbf{D}$$

Obviously there are some problems and limitations designer has to face to get good results. First of them is the data quality. Data can be structured and gives explicit information or can be unstructured as textual data. In this second case system needs a set of tools to retrieve latent information from them. The quantity of the data is also an important piece of this puzzle. In order to get a good collaborative system a huge amount of data is needed.

2.2 LinkedIn: an example

Job Recommendation system already exists and they are very important for job seeking. In this filed, one of the most important and famous service is **LinkedIn**. It's an American company that operates via web site or mobile apps and try to help worker find best jobs and company find best candidates. One of the key point of its success is the Recommendation System[12] inside.

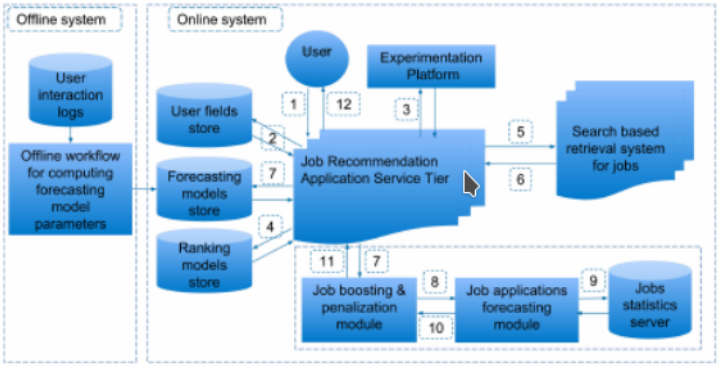


Figure 2.2: Overview of the LinkedIn recommendation system

It had been improved in the past years and fed with tons of data and now it is capable to suggest very good recommendations. But a growing dataset with a complex query to satisfy represent a unique and hard challenge that machine learning experts have to manage. One piece of this system is called **Recruiter System** and, like the name says, it is the component that recruits candidates for a given job posting. This system has to respond to queries respecting the following criteria:

- **Relevance**. The results must be relevant for the positions.
- **Query Intelligence**. The query shouldn't look for just specific criteria but also for similar ones.
- **Personalization**. Possibility to personalize searches using customized search criteria

The initial recommendation experience in *LinkedIn Recruiter* was based on a linear regression model. This model was easy to interpret and debug but failed to find non-linear correlations. So engineers decided to improve experience deploying a new and most efficient model: **Gradient Boosted Decision Trees** [13] that combined different models in a complex tree structure. It improved the recommendation system in general but it failed to address some key challenges.

To solve this problem, LinkedIn added a series of context-aware features based on a technique called **Pairwise optimization**. Essentially, this method made model be capable of comparing candidates' context finding the one who best fit current search context. This is just a little piece of the LinkedIn structure of Recommendations System but it is useful to underline the role Deep Neural Network can play in this field.

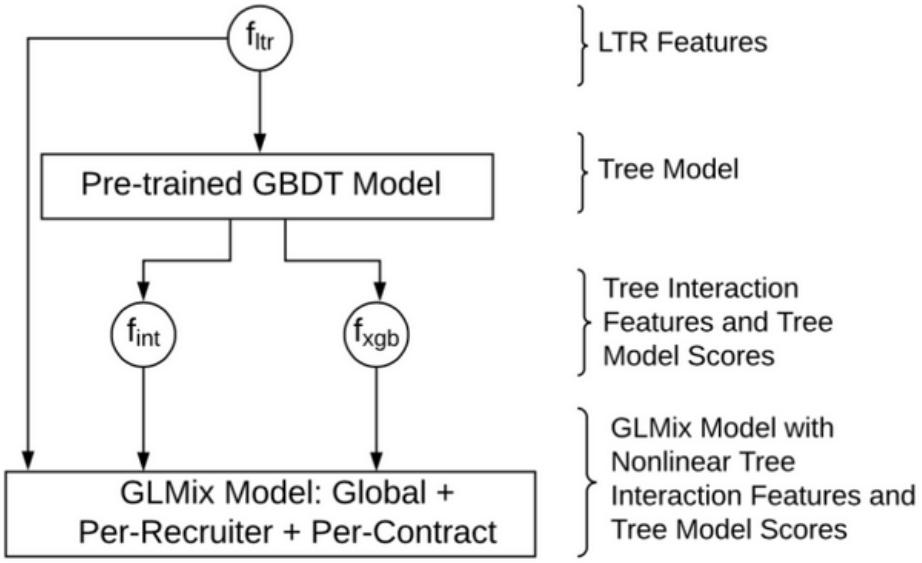


Figure 2.3: Abstract overview of LinkedIn Internal structure for Recruiter system

2.3 The data

The dataset used during the course of this project is provided by **Kaggle** (https://www.kaggle.com/). Kaggle is a web site where many datasets are stored and it provides not just data but tutorials, resources to develop your own model and global competitions. To get access to all of these benefits an account is needed but it is completely free. The dataset for this work comes from a kaggle challenge called **Job Recommendation Challenge** and sponsored by **careerbuilder** (https://www.careerbuilder.com/) that is an online service for job postings. This competition was about creating a model that was capable to predict for a given user which jobs he would apply. The prediction was based on user's previous applications and some other related information.

It took place 7 years ago with the technologies of that period and now, it would have been interesting adapt the best solution to the new technologies and check the results but no solution had been released so this comparison is impossible. Data within the dataset were collected by carrerbuilder and stored in its internal database. They were about users, job postings, job application that user made to job posting.

In total the applications spanned 13 weeks. This period had been divided in seven parts called windows and each users or job posting has been assigned

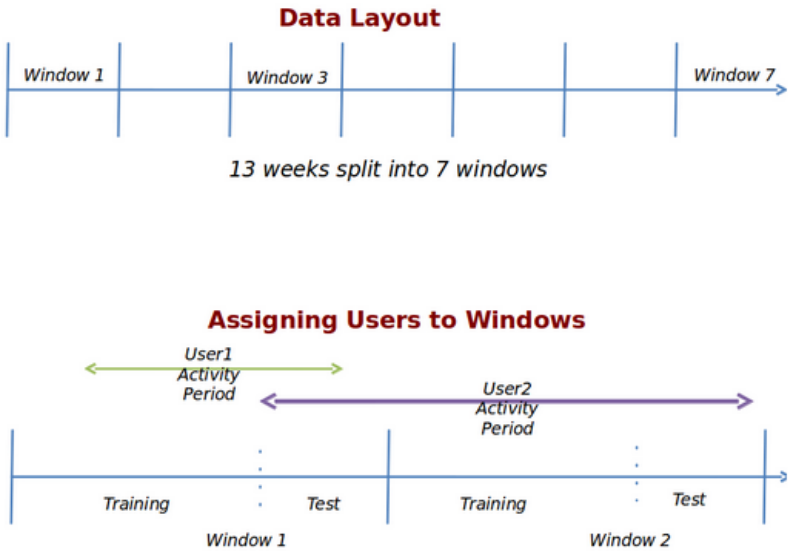


Figure 2.4: Overview of the time windows from which data has been taken.

to only one of these windows. Job is assigned to a window with a probability proportional to the time it was public on the site in that window. Each user is assigned to a window with a probability proportional to the number of applications made by him in that window and during that window. Each window is split in two parts: train split and test split. All data are stored in relational tables contained in the following file:

- **Window_dates.tsv** that contains information about timing of each window.
- **users.tsv** that contains information about users. Each user is identified by ID called UserID.
- **test_users.tsv** contains a list of the Test UserIDs and windows.
- **user_history.tsv** contains information about users' work history.
- **jobs.tsv** contains information about job postings.
- **splitjobs.zip** contains the same information of jobs.tsv but with jobs grouped in the timing windows.
- **apps.tsv** contains information about applications.
- **popular_jobs.csv** is a submission file.
- **popular_jobs.py** is a python script that produces popular_jobs.csv

	UserID	WindowID	Split	City	State	Country	ZipCode	DegreeType	Major	GraduationDate
0	47	1	Train	Paramount	CA	US	90723	High School	NaN	1999-06-01 00:00:00
1	72	1	Train	La Mesa	CA	US	91941	Master's	Anthropology	2011-01-01 00:00:00
2	80	1	Train	Williamstown	NJ	US	08094	High School	Not Applicable	1985-06-01 00:00:00
3	98	1	Train	Astoria	NY	US	11105	Master's	Journalism	2007-05-01 00:00:00
4	123	1	Train	Baton Rouge	LA	US	70808	Bachelor's	Agricultural Business	2011-05-01 00:00:00

Figure 2.5: Users table view using Pandas

2.3.1 window_dates.tsv

This file, as the others, contains data and it is a tsv format file that means tab separated value. Each value inside the file is separated from the others by using a *TAB*. In this file there is a relational table that contains information about time and windowing. It is divided in 4 columns:

- **Window** this column contains the window ID as an integer.
- **Train Start**. It is a date and it represents the beginning time of the train split.
- **Train End / Test Start**. It is a date and it represents the end of train split and the beginning of the test split.
- **Test End**. It is a date and it represents the end of the test split.

This table contains 7 rows, one for each window.

2.3.2 users.tsv

This file contains information about all the users in the system and these infromation are spread over 15 columns:

- **UserID** and **WindowID**. These two columns contain two integers which identify the user and the window in which user has been assigned to.
- **Split** that contains a string that represents the split (train or test).
- **City, State, Country** and **ZipCode** contain information about where user lives. Users are from 10734 different cities, from 221 states and from 120 countries.

	UserID	WindowID	Split	Sequence	JobTitle
0	47	1	Train	1	National Space Communication Programs-Special ...
1	47	1	Train	2	Detention Officer
2	47	1	Train	3	Passenger Screener, TSA
3	72	1	Train	1	Lecturer, Department of Anthropology
4	72	1	Train	2	Student Assistant

Figure 2.6: user_history table view using Pandas

- **DegreeType, Major** and **Graduation Date**. They contain information about the education level of the user. DegreeType is a value from seven: None, High School, Associate's, Bachelor's, Master's, PhD, Vocational.

- **WorkHistoryCount, TotalYearsExperience, CurrentlyEmployed, ManagedOthers, ManagedHowMany**. They contain information about professional career of the users.

This table contains a lot of information about users but some of them like *Major* are hand-written by the users. In the whole dataset users are 389708 and this table contains NaN values.

2.3.3 user_history.tsv

This file contains a table in which there are users' job histories. This table has four columns:

- **UserID** and **WindowID**. These two columns contain two integers that identify the user and the window in which user has been assigned.

- **Split** that contains a string that represents the split (train or test).

- **Job Title** that contains the title of the job hand-written by the user.

- **Sequence**. It is an integer number that indicates the position of the job inside the sequence. 1 for the first, 2 for the second and so on.

This file contains one job history for each user, but in this table users are 375531 that means not all the users in the system have submitted their job histories. There are 14177 users without any indication of their past jobs.

2.3.4 jobs.tsv

This file contains a table with information about Job Postings made by a Company. It has the following columns:

- **JobID** and **WindowID**. These two columns contain two integers that identify the Job and the window in which job posting has been assigned.
- **Title, Description** and **Requirements**. These contains information about the kind of work, but the requirements are not obligatory so many case this field is empty or contains indication that invites lecturer to check the description.
- **City, State, Zip5** and **Country** contain information about where company resides. Job Posting are from 11074 different cities, from 60 states and from 66 countries.
- **StartDate** and **EndDate**. These two columns are the beginning and the end of the period job posting is public.

The title, description and requirements fields are hand-written by the Company. The total number of job posting in the table is 1091923.

2.3.5 apps.tsv

This file contains a table with the application submitted by the users to a job posting in the related table. It is composed by 5 columns:

- **JobID, UserID** and **WindowID**. These three columns contain integers which identify the Job posting, the users and the window which job application has been done.
- **Split** that contains a string that represents the split (train or test).
- **ApplicationDate** contains the dates when application has been done.

This table contains 1603111 job applications.

2.3.6 Other files

The other files in the dataset contain the same information or information obtainable from the file described above. **test_users.tsv** is a table containing users for whom to predict the applications in a given window for the challenge submission. **splitjobs.zip** contains the same table as jobs.tsv but job posting are grouped in windows according to their visible period.

2.4 Contribution of this work

During the progresses of the project a lot of technologies have been tested but, after all of this work, two models were created. These models reached very good results opening the path to new researches in this field. Today, the most used method to approach this kind of task is to convert jobs and structured information into some vector representations using specific models or algorithms and to use another model for the prediction or the recommendation. All of this system is based on very structured data.

In this work the best results have been reached using the same model for both the duties. This model is RoBERTa3.6 model that was fine-tuned and trained for the specific task. It is capable to understand, given a job history and some user information, if the last job is part or not of the given job history with a precision of 74%. It is also capable reading a users' job history and a job posting, to say with a precision of 92.527% if the given job posting has been submitted by the users in input.

These results underline that this kind of approach and new technologies can help to create very precise Recommendation System, even without well structured data, but they still have some limitations. For both tasks the model is the same, it is RoBERTa default model, but it has been trained on different task specific datasets.

Chapter 3

Available Technologies

The technologies for downstream NLP tasks are growing faster, changing every day and pushing the State-Of-Art a little forward. Researchers from public and private groups as Google, Zalando, Amazon are developing more and more accurate algorithms day by day. In this chapter the most important technologies in the scope of this project will be explained in detail. This is important in order to fully understand the context of this work.

3.1 Attention Mechanism

[14] For years Recurrent Neural Networks like LSTM[15] or GRU[16] have represented the best solution to work with information modelled by sequences of data. Despite achieving better performances compared to purely statistical methods, the RNNs-based network suffers from two serious drawbacks. They are forgetful meaning old information tends to disappear after multiple steps and, second, there is no explicit word alignment. To address these problems, Attention Mechanism was introduced in neural network in particular in neural network translation machine. The basic idea of this new mechanism is to create a Context c for each data in the sequence. Each data is represented by a vector v and this context contains information about similarity with other data in the sequence in order empower latent relationship between data. In other word, using attention mechanism, model wonders on which other part of the input sequence should pay attention to fully understand a given data.

3.1.1 First implementation

To formally discuss about attention mechanism it is necessary to define $V = \{\vec{v_i}\} \in \mathbb{R}^{nxd_v}$ as a sequence of vectors. First of all it is needed to compute a vector $\vec{\alpha}$ called **Attention Score**

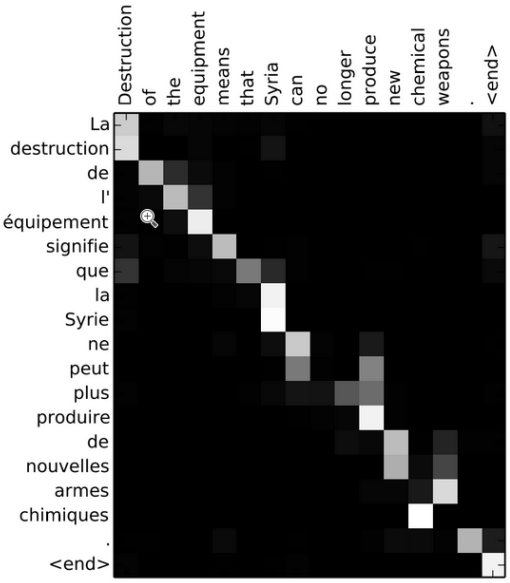


Figure 3.1: Graphical representation of attention during translation

$$e_i = a(\vec{u}, \vec{v_i}) \tag{3.1}$$

$$\alpha_i = \frac{e_i}{\sum_i e_i} \tag{3.2}$$

In the first equation $\vec{u} \in \mathbb{R}_u^d$ is the vector in the input that is going to be matched with all the others. It is called **Pattern Vector**. This match is performed using a function $a(\vec{u}, \vec{v})$ that produces a scalar score e_i that indicates the quality of the match. After that, those scores are normalized to create the final α that can be used to compute the context vector c:

$$c = \sum_i \alpha_i \vec{v_i} \tag{3.3}$$

3.1.2 Multi-Dimensional Attention

The previous kind of attention mechanism can be seem as a 1D attention because $\vec{\alpha}$ is a vector containing the normalized score. **Multi-Dimensional Attention** is proposed to improve the previous simple mechanism to work with more complex relationships and to capture multiple interactions between terms in different representation spaces. The idea is to map vectors to K different

vector spaces end to compute attention for each of them. In order to do that it's needed $\mathbf{W} = W_k \in \mathbb{R}^{KXdXd}$ that is a 3D tensor composed by K matrices W_{dXd} and each of them maps vectors V into a different vector space. At the end all matching scores are concatenated to create the final e_i

$$\vec{e1} = concat(a(\vec{u}^T W_k \vec{v_i})) \tag{3.4}$$

The equation 3.4 shows that in multi-dimensional attention $\vec{e_i}$ is a vector containing all match scores from all different vector spaces.

3.1.3 Self Attention

Self attention is a special attention mechanism where the pattern vector u is not independent from V, like in the classic implementations, but its a part of it. In this approach V is split in many v that, one per step, become the u. Self attention was created to better model the latent relationships between the various parts of the same sequence. This kind of attention is wide used in **Transformers**3.4.

$$a = softmax(\frac{\vec{u}K^T}{\sqrt{d_k}})V \tag{3.5}$$

The above equation is the typology of Self Attention adopted by Trasformers. In this kind of mechanism $\vec{u}$ is the part of the input sequence, $K_n xd$ is the matrix of the keys with one row for each data in the input sequence and $V_n xd$ is a matrix of the values similar to K. Both are created multiplying the input sequence to two matrices: W_k for keys and W_v for the values. This procedure is repeated for each part of the sequence.

3.1.4 Conclusion

Attention mechanism proved itself to reach better result even than Recurrent Neural Networks. It is used with success in Natural Language Processing Tasks to create Context Word Embeddings, but also in image recognition tasks used in combination with a Convolutional Neural Network[17]. Despite its wide usage a deep formal and mathematical justification about it success is still needed and searched. It could be useful to find it in order to improve this mechanism and to apply it into different fields.

Compared to its wide usage in various NLP tasks, attempts to explore its mathematical justification still remain scarce. Recent works that explore its application in embedding pre-training have attained great success and might be a prospective area of future research - Dichao Hu[14]

3.2 Flair and Contextual String Embedding

[7] A crucial component of each Natural Language Processing Task is the word embeddings. Their quality impacts heavily on the results of the model and recent studies have underlined their importance. Find the best word embeddings is the goal for a lot researchers nowadays. The state of art methods to create them are three:

- **Classical word embeddings** pre-trained over very large dataset. They aim to capture latent semantic and syntactic similarities
- **Character-level features**[18]. They are not pre-trained but generated using task data to better represent task features.
- **Contextual Word Embedding**[19] that represent word in the context to address problems as polysemous.

The idea behind the **Contextual string embeddings** is to combine the attributes of the above embeddings in order to create best vector representations. According to the result of recent works[20] that show how natural languages can be modelled using probability distribution over characters instead words, these Contextual String Embeddings come from a Character Level Neural Model. This kind of vectors reached state of art in some NLP task as NER, PoS, ...

3.2.1 Experiments and Results

The proposed embeddings have been compared with the other kinds in the following tasks:

- **Named entity recognition** using both CoNLL03 English and German. Data contains entity of four types *PER* for person, *ORG* for organization, *LOC* for location and *MISC* for miscellaneous names.
- **Chunking** using CoNLL2000. The task consists in dividing a text in syntactically correlated parts. It is an intermediate step toward full parsing.
- **Part of Speech tagging** using data from **The Penn Treebank Project** (https://web.archive.org/web/19970614160127/http://www.cis.upenn.edu/ treebank/)

Furthermore the contextual string embeddings used for testing and evaluations are generated using 4 different approaches.

- **Proposed**. Just Contextual String Embeddings alone

Approach	NER-English F1-score	NER-German F1-score	Chunking F1-score	POS Accuracy
proposed				
PROPOSED	91.97±0.04	85.78 ± 0.18	96.68±0.03	97.73±0.02
PROPOSED$_{+\text{WORD}}$	93.07±0.10	88.20 ± 0.21	96.70±0.04	97.82±0.02
PROPOSED$_{+\text{CHAR}}$	91.92±0.03	85.88 ± 0.20	**96.72**±0.05	97.8±0.01
PROPOSED$_{+\text{WORD+CHAR}}$	**93.09**±0.12	**88.32** ± 0.20	96.71±0.07	97.76±0.01
PROPOSED$_{+\text{ALL}}$	92.72±0.09	n/a	96.65±0.05	**97.85**±0.01
baselines				
HUANG	88.54±0.08	82.32 ± 0.35	95.4±0.08	96.94±0.02
LAMPLE	89.3±0.23	83.78 ± 0.39	95.34±0.06	97.02±0.03
PETERS	92.34±0.09	n/a	96.69±0.05	97.81 ± 0.02
best published				
	92.22±0.10 (Peters et al., 2018)	78.76 (Lample et al., 2016)	96.37±0.05 (Peters et al., 2017)	97.64 (Choi, 2016)
	91.93±0.19 (Peters et al., 2017)	77.20 (Seyler et al., 2017)	95.96±0.08 (Liu et al., 2017)	97.55 (Ma and Hovy, 2016)
	91.71±0.10 (Liu et al., 2017)	76.22 (Gillick et al., 2015)	95.77 (Hashimoto et al., 2016)	97.53±0.03 (Liu et al., 2017)
	91.21 (Ma and Hovy, 2016)	75.72 (Qi et al., 2009)	95.56 Søgaard et al. (2016)	97.30 (Lample et al., 2016)

Figure 3.2: Results obtained by using Contextual String Embeddings

- **Proposed + word**. An extension in which they concatenate pre-trained static word embeddings with Contextual String Embeddings.
- **Proposed + char**. Similar extension in which they concatenate task trained character embeddings to their Contextual String Embeddings.
- **Proposed + word + char** where concatenation is made by using Contextual String Embeddings, word embedding, task trained character embeddings.
- **Proposed + all** putting every kind of embedding together.

In each task Contextual String Embeddings help model to reach state of art. Results are shown in the figure3.2

3.2.2 Model

The selected model for the generation of these kind of Contextual String Embeddings is a bidirectional Long Short Term Memory (LSTM). It is a famous variant of Recurrent Neural Network the got recent success. Atomic input units are the characters so, at each step, the network has an internal representation of the current character given by its hidden states. The target of this model, as each character level models, is to estimate a good probability distribution

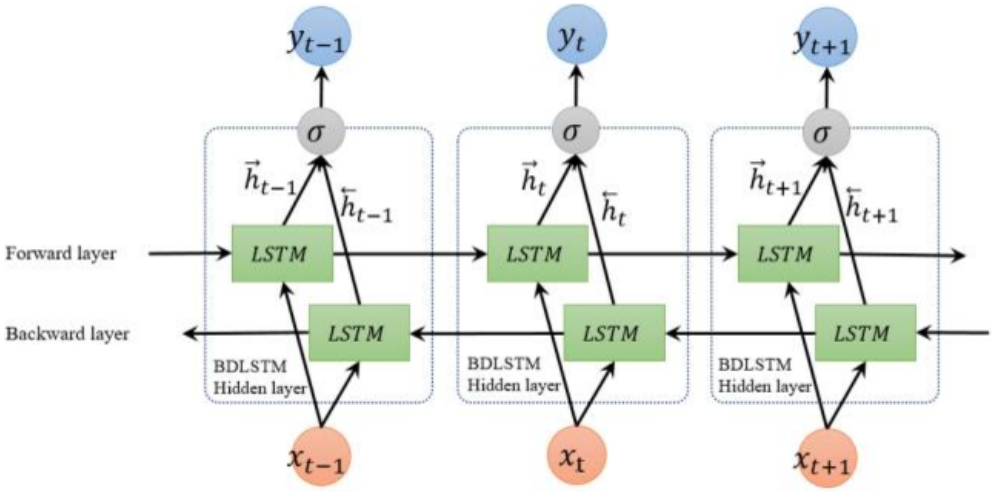


Figure 3.3: Bidirection LSTM Neural Network

over characters of the input language.

$$P(C) = \prod_{t=0}^{T} P(c_t|c_{0:t-1}) \tag{3.6}$$

The above equation shows the target function in which C is the set of all characters in the input sentence $< c_0, ..., c_n >$ and $P(c_t|c_{0:t-1}$ is the probability assigned to c_t to appears after $c_{0:t-1}$. Training the model to generate a good probability distribution forces it to well understand characters, relationships between them, words, sentence, semantic and syntactic rules.

3.2.3 Contextual Word Embedding Extraction

From the model described above it's possible to extract, using its internal hidden layers, information about input characters which can be manipulated to create Word Representation Vectors. To do this it is necessary to define input sentences as a sequences of characters $< c_0, ..., c_n >$. To create the representation vector of a word w_0 in the sentence model must be used in both direction: forward and backward. Word is composed by characters and it is a sub-sequence of the sentence: $< c_0, ..., c_b, ..., c_f, ..., c_n >$ where c_b is the first character in the word and c_f the last. Model is fed in forward direction by giving to it all characters from c_0 to c_f. In this way it can get information from the context prior to the target word. The same process is repeated but in the backward direction using characters from c_n to c_b. From these two steps, after the last character in the sequence, it is possible to get two word

vector representations by taking the hidden states: $\vec{h_f}$ and $\vec{h_b}$. Then they are concatenated to create the final Contextual Word Embedding.

$$\vec{w_0} = \left[\vec{h_f}, \vec{h_b}\right] \tag{3.7}$$

3.2.4 How to use

The model to produce those embeddings and the code for training or fine-tuning it can be found at the link https://github.com/zalandoresearch/flair

3.3 ELMo - Embeddings for Language Models

3.3.1 Overview

Pre-trained word representation are the key components in many NLU tasks. A good representation of the word and its characteristics leads model to achieve better results. But a representation to be considered a good one has to be capable to model:

- Complex characteristics of word usage (**Syntax** and **Semantics**)
- How these uses vary across linguistic contexts (**Polysemy**)

ELMo wants to address both of these challenges using a **Deep Contextualized Word Representation**. In order to do that ELMo is based on a Bidirectional Deep Neural Network (biLM) capable to model a word according to the context into it is inserted. But differently from other models ELMo applies a function to create the embeddings using all of the internal layers of the net rather than using just to the last one (like FlairEmbeddings. See chapter 3.2). ELMo Team claims that different layers give word representations specialized on different aspects.

Using intrinsic evaluations, we show that the higher-level LSTM states capture context-dependent aspects of word meaning (e.g., they can be used with-out modification to perform well on supervised word sense disambiguation tasks) while lower-level states model aspects of syntax (e.g., they can be used to do part-of-speech tagging)
From ELMo Paper [21]

This strategy demonstrates to work extremely well in practice improving the State-of-Art in different tasks.

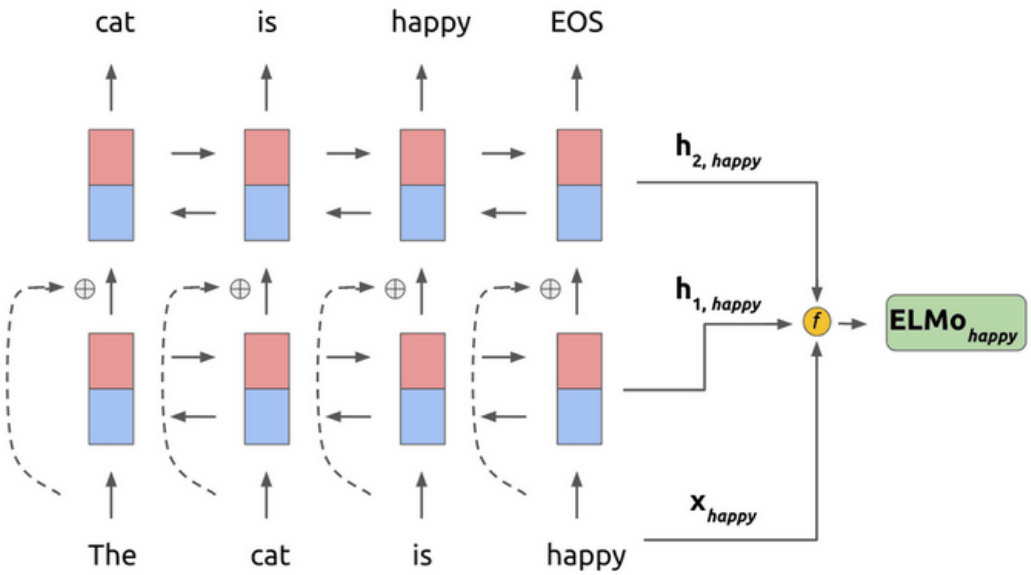


Figure 3.4: LSTM Layers in ELMo Model

3.3.2 Model

ELMo Model is based on different kinds of neural network working together. There are LSTM neural network and CNN. It takes in input a sequence of words from a sentence. Each of them is turned into a matrix and it is given to a **Character-level Convolutional Neural Network** that, applying some filters, produces a feature vector for the input word. Feature Vector is a vector representation context-independent. To put context information into the word representation, the vector is passed to a **BiLSTM** with N layers. All of them produce hidden states and then, those hidden state are concatenated performing a weighted sum into a unique vector. The following lines explain the above key concept in detail in order to give to the reader a deep understanding of the ELMo model.

- **Character-level CNN**. ELMo needs context-independent word embeddings to be trained. Authors chose to use Character-CNN based model. They trained a 2048 channel char-ngram CNN followed by two highway layers and a linear projection down to 512 dimension. A good overview of this part is in the paper **Character-Aware Neural Language Model**[2]. According to this paper the input sentence is split in word $[k_0, k_1, ..., k_j] \in V$ and each word is composed by $[c_1, c_2, ..., c_l]$ characters where l is the length of the word k. Characters are turned into fixed-sized vector multiplying the one-hot encoding of the character

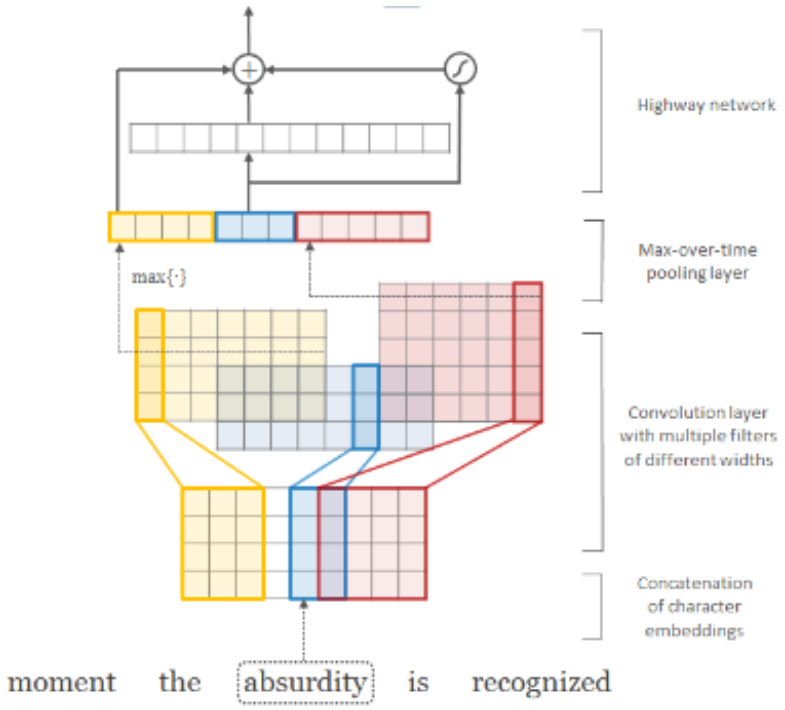


Figure 3.5: Character-Level CNN used in a lot of models including ELMo. Picture from paper [2]

by a weight matrix W_v. In this way a word k is represented by a matrix C_{dxl} where each column is the vectoring representation of each character. After that filters H_{dxw} with $w \in W$ (weights of filters) are applied and for each filter a feature vector is computed. The feature vector is generated using this formula:

$$\mathbf{f}^k[i] = tanh(< \mathbf{C}^k[*, i : i + w - 1], \mathbf{H} > +b)$$

From each feature vector the max value y_i is extracted to create the final representation of the word k. If $H_0, H_1, ..., H_i$ are the filters used the final vector y^k for the word k will be $[y_0^k, y_1^k, ..., y_i^k]$. After this CNN there are placed two **HighWay Neural Network** [3] which give to the model the possibility to choose to apply affine function followed by a non linear function (as in a classical layer) or not. To give to the network this ability a layer in the Highway Neural Network performs this function:

$$y = F(x, W_f) * T(x, Wt) + x * (1 - T(x, W_t))$$

So the layer is controlled by the paramenters $T()$ and it is the net itself to choose the right value into it.

$$y = \begin{cases} x & \text{if } T(x, W_t) = 0 \\ F(x, W_f) & \text{if } T(x, W_t) = 1 \end{cases}$$

- **ML-biLSTM Layers**. ELMo uses a Multi Layers biLSTM neural network. A biLSTM is the combination of two LSTM Neural Networks that have learned how to extract information from input sequence in both the directions: from beginning to the end and backward. This architecture performs very well in NLP tasks and it is wide deployed. What differs in ELMo model from others is how output is compute. In ELMo y is a weight sum of all hidden states.

$$ELMo(k) = \gamma \sum_{j=0}^{L} s_j * h_{k,j}$$

Where k is the token in input, L the number of hidden layers, h the state of a hidden layer, s a weight vector and γ a scalar used according to the task model is performing.

3.3.3 Performances of ELMo

Authors suggest to use ELMo with another Model in order to improve the system. The Idea behind is to use ELMo just to produce better Word

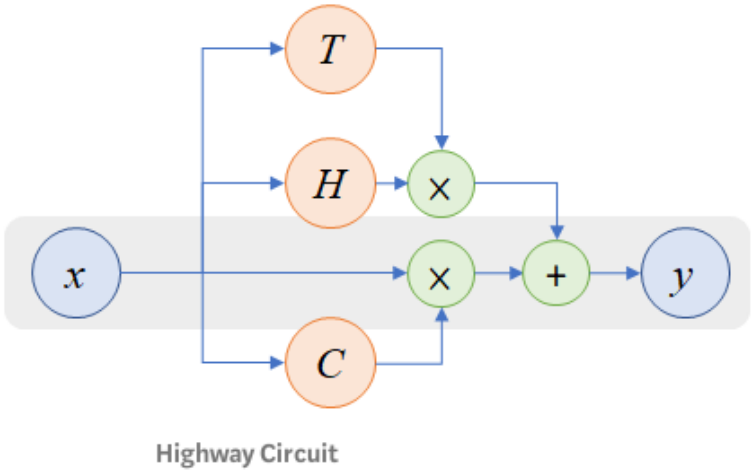


Figure 3.6: Highway model. Picture from paper [3]

Embeddings so, if there is another model for a specific task, ELMo can be added anyway to improve the performances. In many benchmarks for NLP Tasks ELMo has been added to the best model and it helped to get new States of the Art.

- **SQuAD**. Stanford Question Answering Dataset contains more than 100K crowd sourced question-answer pairs where the answer is a span in a given Wikipedia paragraph. The previous best model was a Bidirectional Attention Flow Model[22] with 81% of F-measure. After adding ELMo as embedding components the F-measure improved by 4.7%.

- **Semantic Role Labeling**. This task is performed on OneNotes benchmark with more than 2.9 million words and it consists to understand which words are the subjects, the verbs and the objects. The best score is obtained using a deep biLSTM interleaved neural network [23]. Adding ELMo the F-measure jumped up by 3.2% from 81.4% to 84.6%

3.3.4 How to use

Elmo can be used trough the **AllenNLP** packages for python from this site https://allennlp.org/elmo. There are also free pre-trained model available *Small* with 13.6M parameters, *Medium* with 28.0M parameters, *Original* and *Original 5.5B* both with 93.6M parameters.

3.4 Transformers

A new kind of Neural Network Architecture has recently proved his power in Sequence to Sequence tasks as translation. It is called Transformer [24] and it is created to solve a specific problem. All sequence translation models are based on complex recurrent neural network as RNN, LSTM, BiLSTM, GRU .... They work reading just one word per step and precluding any form of parallelization within training examples. This leads to a critical performance for wide input sentences. Transformer wants to overcome this limits but without worsening performances.

3.4.1 Overview

Transformers are build without using Recurrent Neural Network in order to untied them from input sequence constrain, however the ordering of the word is a important information for Natural Language Processing. So Transformers employ a mechanism called **Attention** that is often used with advanced Recurrent Neural Network. Attention helps a Neural Network to focus on interesting elements and to ignore others. Trsformers use a special kind of attention called **Self-Attention**, sometimes **Intra-Attention** and it is a mechanism that creates relationships between pieces of a single input sequence. For each piece of the sequence (like words for sentence or number for numeric series) it creates a vector representation of its relationships with other pieces and this vector is called Context Vector. This kind of attention has been used successfully in a number of NLP tasks (please refer to section 3.1.

3.4.2 Model

The Transformer architecture is based on Encoder-Decoder architecture. This kind of Neural Notwork are widely employed in Sequence to Sequence tasks because their good performances. The encoder maps a input sequence of embeddings $x_0, x_1, ...x_n$ to an output sequence $z_0, z_1, ...z_n$. These vectors become the input of the decoder that generates a sequence of symbols $y_0, y_1, ...y_m$. Model is auto-regressive, consuming the previously generated symbols as additional input at each step. This mechanism is shown in 3.7. Transformer encoder is composed by a stack of $N = 6$ layers and each layer contains two sub-layers: **Multi-Head Self-Attention Layer** and **Feed Forward Neural Network**. The output of the first sub-layer is combined with his input and becomes the input of the second sub-layer $o_i = LayerNorm(x_i + sublayer(x_i))$. This technique is called **Residual Connection**. The output of the entire layer is the input of the next layer in the stack. Transformer decoder, as encoder, is

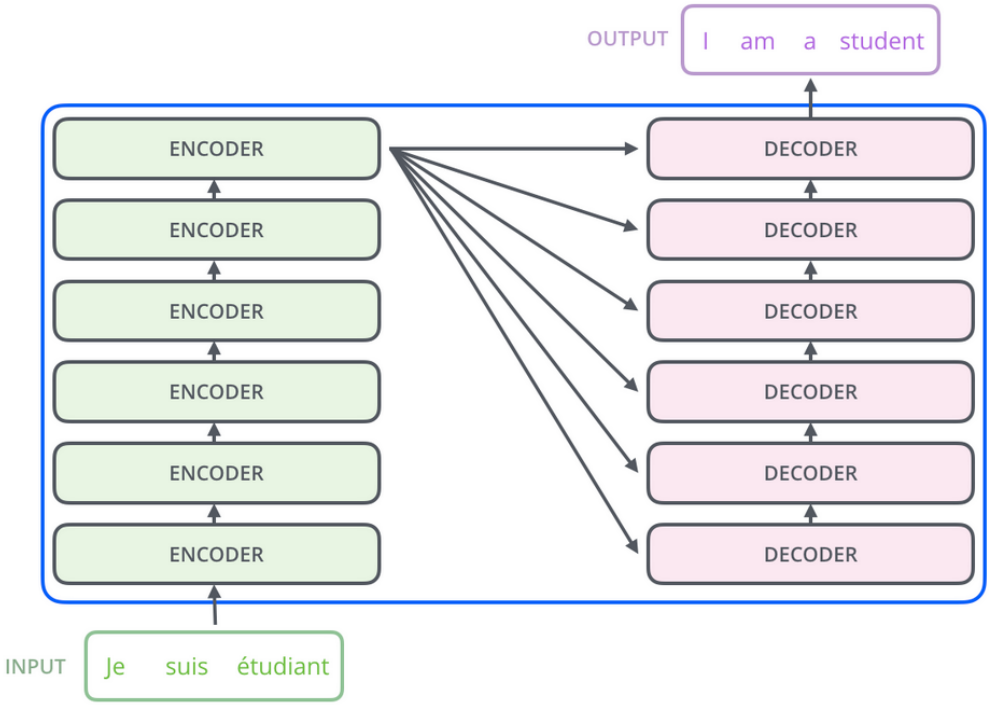


Figure 3.7: Encoder-Decoder Architecture of Transformer

composed by a stack of $N = 6$ layers. Each layer contains three sub-layer: two **Multi-Head Self Attention** layers and a **Feed Forward Neural Network**. Between each sub layer there is a **Residual Connection**. At the end of the Decoder stack there is a **Linear Layer** that takes in input z_i from the decoder with size d and generates and output y with size equal to natural language vocabulary size. After this layer there is a **SoftMax Layer** that generates a probability distribution over the vocabulary. A visual representation of the architecture is in the figure3.8

3.4.3 Attention Mechanism in Transformer

In both encoder and decoder there are sub-layers called **Multi-Head Attention**. Attention is representable as a function $Attention(K, V, Q)$ that produces as result Z.

- K_{nxd} It's a matrix containing in each row a key k that represents a value.
- V_{nxd} It's a matrix containing in each row a vector of values v.
- Q_{qxd} It's a matrix containing in each row a query q.

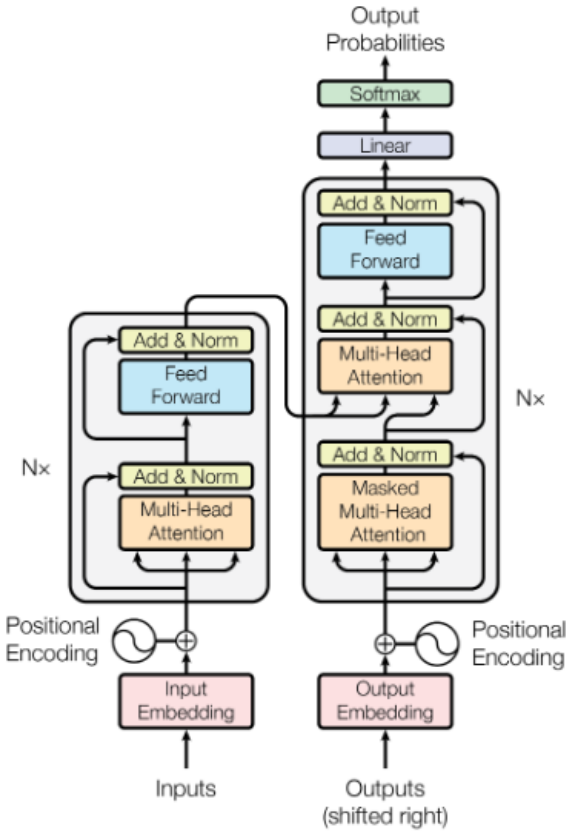


Figure 3.8: Model Architecture of Transformer

- Z_{qxd} It's a matrix containing for each query a representation with attention z.

In general NLP tasks q is a word and model looks for related words in a given Sentence. k and v are vectors associated to those words. To compute Z this formula is used:

$$Z_{qxd} = SoftMax(\frac{Q_{qxd} * K^T_{dxn}}{\sqrt{d_k}}) * V_{nxd} \tag{3.8}$$

where d is the size of the model. Transformer has $d = 512$and uses this different kinds of this mechanism in each layer this way:

- `Encoder`: It use **Self attention**. It is performed using just input sequence X. Each x_i in input is multiplied by three matrices W_k, W_v, W_q randomly initialized. W_k is used to generate K, W_v to generate V and W_q to generate Q. Transformer, during training phase, has to modify those matrix in order to improve the model.

- `Decoder`: there are two layers here which perform Attention. The first takes as input the decoder output tokens it has produced until now and works like the encoder. It converts them into vectors and perform a self attention. The second layer use a **Simple Attention** in fact it computes K and V using the output of encoder and Q with the output from previous layer.

3.4.4 Multi-Head Attention

Transformers adds another level of complexity to attention mechanism employing **Multi-Head Attention**. Instead of compute a single attention function they got better result to linearly project the queries, keys, values h times. To do it transformer uses in each Attention Level h triples of W_k, W_v, W_q. In this way h Z are computed but at the end of each Multi-Head Attention Layer all Z are concatenated and multiplied by a weight matrix to create the output vector with the right size.

$$Z_{qxd} = [Z^0_{qxd}, Z^1_{qxd}, ..., Z^h_{qxd}] * W_{hdxd}$$

3.4.5 Feed Forward Neural Network

Each layer in both, Encoder and Decoder, has a sub-layer containing a **Feed Forward Neural Network**. This Neural Network consists in two layers with

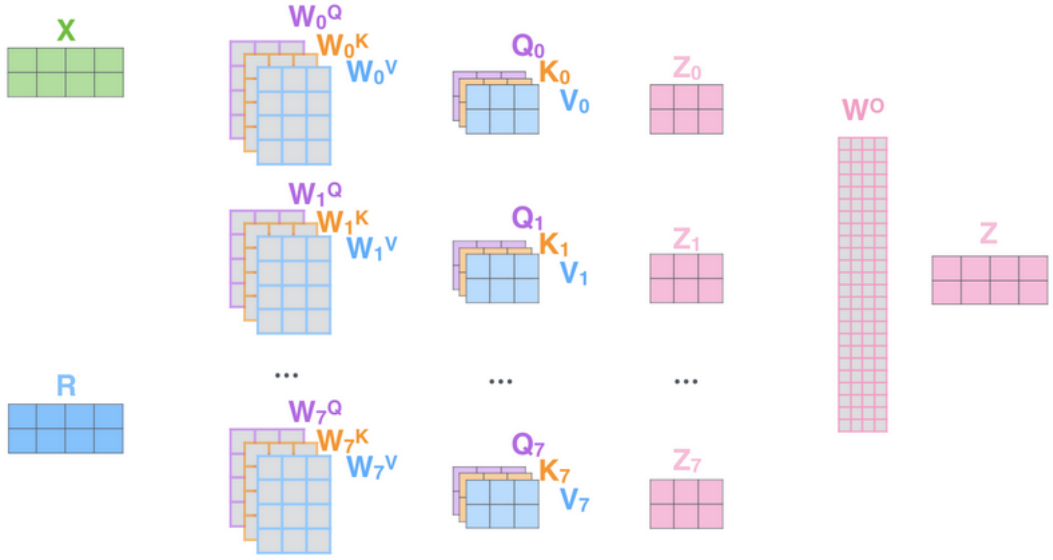


Figure 3.9: Multi-Head Attention

`ReLu` activation function. It can be represent as:

$$FFN(X) = max(0, YW_1 + b_1)W_2 + b_2 \tag{3.9}$$

The output of this Sub-Layer is given directly to the Attention sub-layer of the next layer.

3.4.6 Input and Output

Transformer takes in inputs embedding representations of the words in a sentence. But the problem is that transformer can't know the position of the word in input sentence but this information can be very interesting. To fill that lack a positional information is injected in each word embedding at the bottom of the encoder and decoder stacks. It is called **Positional Encoding** and it is a vector with the same size of the embeddings and each cell is filled following Sine and Cosine function at different frequencies. The value is computed using this formula:

$$PE_{(pos,i)} = \begin{cases} sin(\frac{pos}{10000^{\frac{k}{d_{model}}}}), & \text{if } i = 2k \\ cos(\frac{pos}{10000^{\frac{k}{d_{model}}}}) & \text{if } i = 2k+1 \end{cases} \tag{3.10}$$

According to this formula all even dimensions are encoded using a *sine* function and all odd dimensions using *cosine*. The positional encoding vectors,

generated from the formula above, are summed to the embedding vectors. This allows to each word to be mapped to a different frequency function according to its position in the sentence. So model can get this information directly from the word encoding.

$$x_i = E(w_i) + PE(w_i, i) \tag{3.11}$$

These vectors are the input for the transformer, differently it happens on the other side of the it. The output of the transformer is a probability distribution $P(x)$ over the entire output vocabulary. To do that, Transformer uses two layers: a **Linear Layer** and a **Softmax Layer**. The first, which is a fully connected neural network, takes in input the output O_i from last layer of the decoder and it produces a vector with size equal to the number of words in the output vocabulary. The SoftMax layer turns this vector in the probability distribution.

3.4.7 How to use

Transformers are freely available using the package **tensor2tensor**. It comes from repository https://github.com/tensorflow/tensor2tensor

3.5 Bidirectional Encoder Representation from Transformer

Bidirectional Encoder Representation from Transformer[4] or **Bert** is a new language representation model created by **Google AI Language Team**.

3.5.1 Overview

Unlike recent language representation models, BERT model is pre-trained on unlabeled text and it was forced to retrieve information from both, left or right, contexts in all layers. It can be fine-tuned adding an output layer to create state-of-art models for a wide range of tasks such as Question Answering and Language Inference.

3.5.2 BERT Performance

Bert has proven to excel in Natural languages Understand Tasks and sometimes it has overcame the current State of The Art. Next lines contains

System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	**Average** -
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
$BERT_{BASE}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
$BERT_{LARGE}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

Figure 3.10: Accuracy for GLUE Task. Picture took from Bert Paper[4]

Bert results.

- **General Language Understanding Evaluation** : BERT Score 80,5% (7,7% Absolute improvement). GLUE [25] is a benchmark for the evaluation of a model on Natural Language Understanding (NLU). It comprehends different tasks as Question Answering, Sentiment Analysis and Textual Entailment. All single task results are in the picture
- **Multi-Genre Natural Language Interface** (MultiNLI). BERT accuracy 86,7. It collects 433000 sentence pairs annotated with textual entailment information. It pairs a written or spoken text with a Hypothesis. Task wants the model to classify the entailment between text and Hypothesis as ENTAILMENT, CONTRADICTION, NEUTRAL.
- **Stanford Question Answering Dataset 1.1** (SQuAD v1.1 []): Question answering Test F-Score 93.2. Previous F-Score was 91.2 held by Humans. This dataset is a collection of 100K crowd-sourced question/answering pairs. Given a sentence from Wikipedia and a question, the task is to predict the answer text span in the given passage. Sometimes question could be unanswerable and model should not answer.
- **Stanford Question Answering Dataset 2.0** (SQuAD v2.0 []) BERT F1-Score 83.1. Previous best score was 78.0. This new version contains 100k more data and adding questions which need more complex answers.

3.5.3 Model

BERT architecture is just the encoder piece of the **Transformer** architecture (see section 3.4). The idea is the Transformer uses encoder to generate an embedding representation z_i of a word and decoder, to fulfill the task, has to associate to that vector a probability distribution over the output dictionary. It could be worth to use the z_i representation as embedding for NLP tasks. Following this idea, BERT Team deploys two models: **base** and **large**. The

first is composed by the Encoder with $L = 12$ layers, $H = 768$ hidden size (or model dimension) and $A = 12$ heads in multi-head attention mechanism. The second, a little bit bigger, is an Encoder with $L = 24$ layers, $H = 1024$ hidden size and $A = 16$ heads.

3.5.4 Inputs and Outputs

BERT input is a pair of sentences and in order to create the input for BERT there are some steps to follow:

- **Tokenization**. Sentence is split in tokens using **WordPiece Tokens** []. It is a vocabulary of 30000 tokens and after that special tokens are added to the sentence: *[CLS]* at the beginning of each sentence and *[SEP]* to separate the first sentence to the second. Once all tokens are present they are converted into their IDs from vocabulary.

- **Token Embedding**. IDs are scalars. Each of them is multiplied with vector of size d_{model} to generate the embedding of that token. During train phase these vectors are modified by optimizer to generate even better embedding.

- **Position Embedding** As in Transformer model the positional embedding is add to each Token Embedding. See the section 3.4 for more details.

- **Segment Embedding**. When BERT deals with sentence pairs *[SEP]* tokens help to divide the first sentences to the seconds. To improve this separation an extra vector is used. Each sentence has its vector and it is summed to each token of the sentence. The vectors of the two sentences are different and they are modified during training phase.

3.5.5 BERT Framework

The idea behind BERT is to pre-train Encoder in order to make it learn how to represent words, latent semantics and all useful information. Once it has been trained, it can be deployed to do some NLP tasks just adding a Neural Network at the end of the encoder. This Neural Network takes as input some or all outputs from the Encoder and produces a result. This proceedings is called **Fine-Tuning** and it is performed trough this steps:

- `Pre-training`: during this phase model is trained using unlabeled data and two different tasks. The data comes from **BookCorpus[26]** and English Wikipedia while the tasks are:

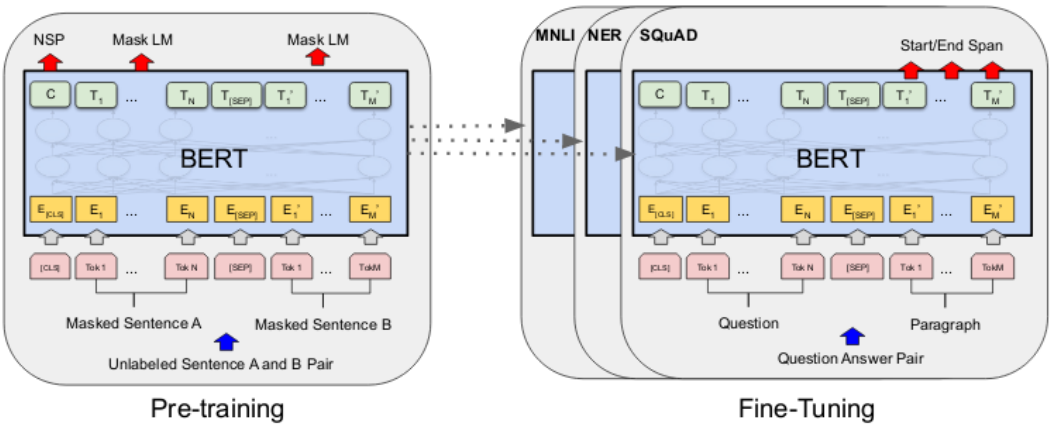


Figure 3.11: Bert Model. Picture from Bert Paper[4]

- **Masked LM** Masked Language Model. In this task a token from inputs is masked using MASK token and Bert must be able to retrieves it. This process forces Bert to put attention on both directions: the piece of sentence prior to the masked word and the one after. MLM works masking some percentage of the input tokens randomly and model must predict them. The outputs corresponding to the MASKs are given to a SoftMax Layer to predict, using a probability distribution, the target word. Problem is that MASK token does not appear during fine-tuning phase. To mitigate this they take 15% of the whole tokens and the 80% of them is turned into MASK tokens, 10% of them is turned in random token and the others are left unchanged. In this way Model is obliged to learn the context of each word.
- **Next Sentence Prediction (NSP)**. In this task BERT is fed with two sentences separated by a special token **SEP** and, at the beginning of them, there is a another special token called **CLS**. BERT must learn to classify the pair A and B as `IsNext` if the sentence B is the actual next sentence of A, or `NotNext`. To do this, the representation of CLS token is given to a classifier layer that has the duty to label the pair. For this task is used a dataset of A,B sentences where in the 50% of the time B is the actual B and 50% of the time B is replaced with a random one.

- `Fine-tuning`: During this phase the model is trained using supervisioned training data from other downstream tasks. The parameters of the Encoder are initialized using those from pre-training phase while an

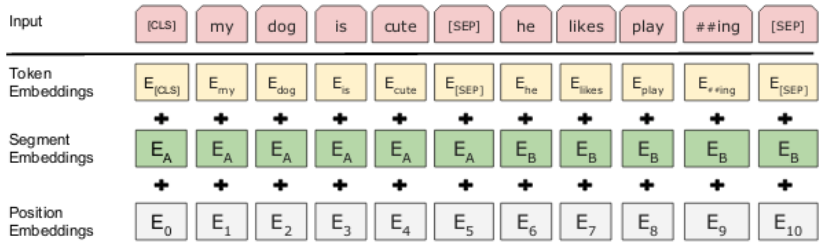


Figure 3.12: Bert Inputs. Picture took from Bert Paper[4]

extra layer is added at the end of the pre-trained model. According to the task this extra layer could be a `SoftMax`, `Sentiment Classifier`, `Entailment Classifier`, ... This gives to BERT the power to be used in different tasks without long training sessions.

3.5.6 How to use

BERT is freely provided by google to anyone. It comes in two versions:

- `base` : Trained using BooksCorpus (800M words) and English Wikipedia (2.5 Billions words). It is a model with 12 layers, 768-hidden size, 12 heads (for multi-head attention), 110M Parameters
- `Large`. It is trained like base version, but it deploys a bigger model with: 24 layers, 1024-hidden size, 16 heads (for multi-head attention), 340M Parameters

You can download them directly from https://github.com/google-research/bert

3.6 ROBERTA

3.6.1 Overview

After the wide success of BERT a lot of studies have been conducted in order to improve it or to overcome its limitations as the computational cost for training the model or just to finetuning it. A lot of advanced models have followed and one of them, from *facebook AI* and *University of Washington*, has reached some good results. It's called RoBeRTa that means: **A Robustly Optimized BERT Pretraining Approach**.

	MNLI	QNLI	QQP	RTE	SST	MRPC	CoLA	STS	WNLI	Avg
Single-task single models on dev										
$\text{BERT}_{\text{LARGE}}$	86.6/-	92.3	91.3	70.4	93.2	88.0	60.6	90.0	-	-
$\text{XLNet}_{\text{LARGE}}$	89.8/-	93.9	91.8	83.8	95.6	89.2	63.6	91.8	-	-
RoBERTa	**90.2/90.2**	**94.7**	**92.2**	**86.6**	**96.4**	**90.9**	**68.0**	**92.4**	**91.3**	-
Ensembles on test (from leaderboard as of July 25, 2019)										
ALICE	88.2/87.9	95.7	**90.7**	83.5	95.2	92.6	**68.6**	91.1	80.8	86.3
MT-DNN	87.9/87.4	96.0	89.9	86.3	96.5	92.7	68.4	91.1	89.0	87.6
XLNet	90.2/89.8	98.6	90.3	86.3	**96.8**	**93.0**	67.8	91.6	**90.4**	88.4
RoBERTa	**90.8/90.2**	**98.9**	90.2	**88.2**	96.7	92.3	67.8	**92.2**	89.0	**88.5**

Figure 3.13: GLUE Results for ROBERTA model

We present a replication study of BERT pretraining that carefully measures the impact of many key hyperparameters and training data size. We find that BERT was significantly undertrained, and can match or exceed the performance of every model published after it. From paper[27]

The idea behind this work is that BERT is good, but it could be better with a well studied training process. This idea leads the researchers to develop this new model that has the same architecture of BERT, but a completely different training phase.

3.6.2 ROBERTA Performances

ROBERTA training process demonstrates to improve BERT results and to reach states of art for some tasks:

- **General Language Understanding Evaluation**[25]. For this task model was fine tuned using two approaches:*Single task* where model is fine-tuned before each tasks and **Esambles**. As it is shown in figure it surpasses BERT in all tasks and some times it reaches the state of the art.

- **RACE**. In this task model has to predict the right answer from four and a given textual passage that contains information about answer. It reaches a new state of art for both tasks. Results are in figure

This proves that preforming a good training sessions it is possible to reach better results.

Model	Accuracy	Middle	High
Single models on test (as of July 25, 2019)			
$\text{BERT}_{\text{LARGE}}$	72.0	76.6	70.1
$\text{XLNet}_{\text{LARGE}}$	81.7	85.4	80.2
RoBERTa	**83.2**	**86.5**	**81.3**

Figure 3.14: RACE results for ROBERTA

3.6.3 Differences from BERT

The Artificial Neural Network behind is the same for BERT and ROBERTA in fact all differences lie in the training phase. For a detailed view of the model see relative section itemize

Train Data Size. The first factors that is different from BERT is the amount of data used for training. As *Baevski*[28] underlines in his work, the quantity of data can improve the results of the model. So ROBERTA was trained using more data than BERT for a total of 160GB of English corpora.

- **Bookcorpus**[26] plus **Wikipedia** that is the original dataset used to train BERT.(16GB)
- **CC-News**. It contains 63 million English news and articles. (76 GB)
- **OpenWebText**. It contains web contents extract from URLs shared on Reddit. (38GB)
- **Stories**.[29] It contains story-like text style of Winograd schemas (31GB)

Dynamic Masking. BERT relies on masking technique for training. During preprocessing phase some random words were hidden under a token MASK and model has to retrieve those words. But in this way masking was static because, in each training epoch, model has to work on the same masked words. ROBERTA training used a dynamic masking to avoid to reuse the same masks. To do that dataset was duplicated 10 times so that each sequence was masked in 10 different ways.

NSP Loss removed. BERT is trained using two different losses: Mask prediction and Next Sequence Prediction that means model has to guess the masked word and, in the meanwhile, to guess if the sentence B is the next sentence of sentence A. Some studies as *Lample and Conneau*[30] has questioned the necessity of the NSP loss so ROBERTA was trained without that Loss Function.

BPE[31]. Byte-Pair Encoding is method to create language dictionary with a size between 10K - 100K. BERT uses a character-level BPE vocabulary of size 30K. For ROBERTA the vocabulary raise to 50K sub-word units.

All of this training differences make ROBERTA different and better than BERT. Furthermore researchers underline that training phase still needs to be well studied.

3.6.4 How to use

Authors released the model freely using directly *pytorch*. They released the code for pretraining and finetuning the model at https://github.com/pytorch/fairseq

3.7 Sentence BERT

After the success of BERT some researchers started to study limits and issues of this big model in order to make it better. Roberta was created to address a bad training phase but others still exists. One of them is that Bert (and Roberta) inputs are sentence pairs and that makes Bert extremely slow in similarity comparison task. Let's imagine having a dataset of 1M strings and we want to find the most similar string in the database to a given one. In order to fulfill this task Bert has to produce a similarity score between each couple of sentences composed by the target string and the others in the dataset and then it will be possible to find the one with the higher score. Let's imagine now that we want to find the most similar sentence for each one in the dataset. This leads us to a task with a very elevate computational cost: $1MX1M$ and if we didn't have a lot of resources we would never get the result. The heart of this problem lies in Bert's lack of creating word embeddings but just to compute and to use them internally. Some studies in this direction have been conducted but it was impossible to find solid way to extract embeddings from the hidden layer of the model. For this reason researchers have developed **Sentence Bert**[32]. It is a modification of the BERT network using siamese and triplet networks that is able to force the model to create meaningful embeddings.

Researchers have started to input individual sentences into BERT and to derive fixed-size sentence embeddings. The most commonly used approach is to average the BERT output layer(known as BERT embeddings) or by using the out-put of the first token (the [CLS] token). As we will show, this common practice yields rather bad sentence embeddings, often worse than averaging GloVe embeddings
- Nils Reimers and Iryna Gurevych[32]

Model	MR	CR	SUBJ	MPQA	SST	TREC	MRPC	Avg.
Avg. GloVe embeddings	77.25	78.30	91.17	87.85	80.18	83.0	72.87	81.52
Avg. fast-text embeddings	77.96	79.23	91.68	87.81	82.15	83.6	74.49	82.42
Avg. BERT embeddings	78.66	86.25	94.37	88.66	84.40	92.8	69.45	84.94
BERT CLS-vector	78.68	84.85	94.21	88.23	84.13	91.4	71.13	84.66
InferSent - GloVe	81.57	86.54	92.50	**90.38**	84.18	88.2	75.77	85.59
Universal Sentence Encoder	80.09	85.19	93.98	86.70	86.38	**93.2**	70.14	85.10
SBERT-NLI-base	83.64	89.43	94.39	89.86	88.96	89.6	**76.00**	87.41
SBERT-NLI-large	**84.88**	**90.07**	**94.52**	90.33	**90.66**	87.4	75.94	**87.69**

Figure 3.15: SentEval results for SBERT

Model	STS12	STS13	STS14	STS15	STS16	STSb	SICK-R	Avg.
Avg. GloVe embeddings	55.14	70.66	59.73	68.25	63.66	58.02	53.76	61.32
Avg. BERT embeddings	38.78	57.98	57.98	63.15	61.06	46.35	58.40	54.81
BERT CLS-vector	20.16	30.01	20.09	36.88	38.08	16.50	42.63	29.19
InferSent - Glove	52.86	66.75	62.15	72.77	66.87	68.03	65.65	65.01
Universal Sentence Encoder	64.49	67.80	64.61	76.83	73.18	74.92	**76.69**	71.22
SBERT-NLI-base	70.97	76.53	73.19	79.09	74.30	77.03	72.91	74.89
SBERT-NLI-large	72.27	**78.46**	**74.90**	80.99	76.25	**79.23**	73.75	76.55
SRoBERTa-NLI-base	71.54	72.49	70.80	78.74	73.69	77.77	74.46	74.21
SRoBERTa-NLI-large	**74.53**	77.00	73.18	**81.85**	**76.82**	79.10	74.29	**76.68**

Figure 3.16: Results of SBERT and other models in Semantic Textual Similarity (STS) benchmark

3.7.1 Sentece-Bert Results

The word embeddings extracted from this model were compared with others extrated directly from BERT or from other models. Sentence Bert has always produced better embeddings than BERT, in fact, they were tested in different Benchmarks and the vectors created by SBERT have got the higher results.

- **Semantic Textual Similarity**. It as dataset that contains pairs of sentences with a score from 0 to 5 that represents semantic relatedness. As it is shown in figure 3.16 SBERT has proven itself to create good embeddings while BERT, on the other hand, reached worse results even than Glove.

- **Argument facet similarity**[33]. It is a dataset composed by 6000 sentential argument pairs from social media divided in three controversial topics: *gun control*, **gay marriage** and **death penalty**. Each pair has a value from 0 to 5 where 0 means *different topic* and 5 **the same topic**. For this task SBERT worked better than Glove but not than BERT that, reading both sentence in the same time it can use the attention on them while SBERT can't.

- **SentEval**[34]. It is a popular toolkit to evaluate the quality of a sentence

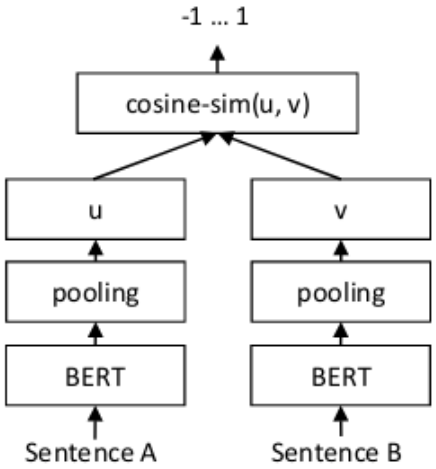


Figure 3.17: SBERT Architecture for compute cosine similarity

embeddings using a logistic regression classifier already trained. This benchmark is composed by more tasks:

- **MR**: sentiment prediction for movie Reviews
- **CR**: sentiment prediction of customer product reviews
- **SUBJ**: subjectivity prediction of sentences from movie reviews.
- **MPQA**: phrase level opinion polarity classification from newswire.
- **SST** Stanford sentiment treebank with binary labels.
- **TREC**: fine grained question-type classification from TREC.
- **MRPC**: Microsoft Research Paraphrase Corpus from parallel news source.

In these embedding oriented tasks SBERT has reached the state of art for 5 out of 7 of them. Results are shown in the figure 3.15

3.7.2 Model

SBERT model is based on BERT (or ROBERTA) model adding a pooling operation to the output of BERT to derive a fixed sized sentence embedding. The pooling strategy tried are three **CLS** using as vector the output of class

token, **MEAN** and **MAX** and the selected is **MEAN**. In order to fine-tuned the model SBERT use a siamese and triplet network[35] to update weights. In this way, the model is forced to create sentence embeddings semantically meaningful and they can be compared by using consine similarity. This network has been trained using different kinds of Objective functions with differences for each of them:

- **Classification**. The embeddings u and v are concatenated with the element-wise difference and multiply with the trainable weight matrix W_t

$$o = softmax(W_t concat(u, v, |u - v|) \tag{3.12}$$

- **Regression**. The cosine similarity between u and v is computed.

- **Triplet Objective function.** Given an anchor sentence a a positive sentence b and a negative sentence c, the distance between a and b must be smaller than the distance between a and c.

$$max(||s_a - s_b|| - ||s_a - s_c|| + \epsilon, 0) \tag{3.13}$$

3.7.3 How to use

The code to use this model is freely distributed. All code, documentations and examples can be found at https://github.com/UKPLab/sentence-transformers

3.8 SHA-RNN Model

The direction of recent works is to use more and more resources, computational power and memory with massive parallelization system to improve the results and reach new states of the art. This approach makes this field accessible just for the big farms but a bit too far for single researchers or little organizations. That seams like the field of the research and the new technologies is a private world for big company as **Google, Facebook, Zalando, ...**. The author of this work **S. Merity** asks himself if this is the only direction and, in response, he presents a very light model with very good results to demonstrate that using more resources is the easiest way but not the only one.

This work has undergone no intensive hyper-parameter optimization and lived entirely on a commodity desktop machine that made the author's small studio apartment far too warm in the midst of a San Franciscan summer. The final results are achie-able in plus or minus 24 hours on a single GPU as the author is impatient. The attention mechanism is also readily extended to large contexts with minimal computation. - Stephen Merity[36]

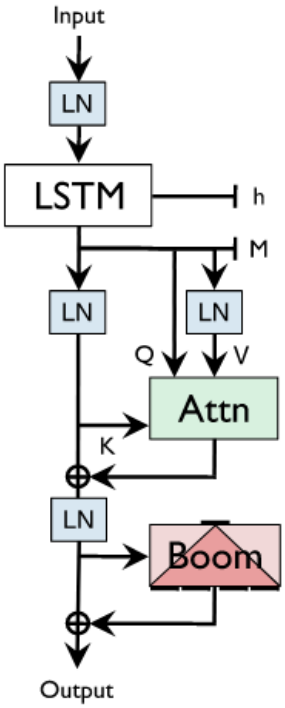


Figure 3.18: sha-rnn architecture

Model	Heads	Valid	Test	Params
Large RHN (Zilly et al., 2016)	0	–	1.27	46M
3 layer AWD-LSTM (Merity et al., 2018b)	0	–	1.232	47M
T12 (12 layer) (Al-Rfou et al., 2019)	24	–	1.11	44M
LSTM (Melis et al., 2019)	0	1.182	1.195	48M
Mogrifier LSTM (Melis et al., 2019)	0	1.135	1.146	48M
4 layer SHA-LSTM ($h = 1024$, no attention head)	0	1.312	1.330	51M
4 layer SHA-LSTM ($h = 1024$, single attention head)	1	1.100	1.076	52M
4 layer SHA-LSTM ($h = 1024$, attention head per layer)	4	1.096	1.068	5[illegible]M
T64 (64 layer) (Al-Rfou et al., 2019)	128	–	1.06	235M
Transformer-XL (12 layer) (Dai et al., 2019)	160	–	1.06	41M
Transformer-XL (18 layer) (Dai et al., 2019)	160	–	1.03	88M
Adaptive Transformer (12 layer) (Sukhbaatar et al., 2019)	96	1.04	1.02	39M
Sparse Transformer (30 layer) (Child et al., 2019)	240	–	0.99	95M

Figure 3.19: Results on enwink8 task got by SHA-RNN

3.8.1 The sha-rnn model

Model architecture comes from `AWD-LSTM` one of author's recent works. It is based on Long Short Term Memory Model (LSTM) but it applies single attention layer on it. Precisely, model consists in a trainable embedding layer, one or more layers of stacked single head attention recurrent neural networks and a softmax classifier. Embedding and softmax classifier utilize tied weights as suggested in the paper `Hakan`[37]. Internally The single head attenetion recurrent layer is composed by LSTM as first sub layer and single head attention after. They are linked by a residual connection. At the end of the model there is a **Boom Layer**. This layer performs a dimensional stretching of the input vector and then reduces the size back to the initial one. To do that the input vector $\vec{v} \in \mathbb{R}^H$ is multiplied with a matrix M_{HXM} where M is $N * H$ so it is a multiple of the initial size of the input vector. This create a vector $u \in \mathbb{R}^{(N*H)}$ and this vector is split in N vectors with size H. After that, these vectors are summed and they generate the output vector $o \in \mathbb{R}^H$

3.8.2 Model results

The model is tested against the `enwik8` dataset or **The Hutter Prize Wikipedia**. It is a byte level dataset creating using the first 100 million bytes from Wikipedia XML dump. The task is to create a model capable to compress the dataset (1GB) less than 116MB losslessly. The model performed well on this task without reaching the state of art but getting very close results. The results can be seen in the figure 3.19.

Chapter 4

Experiments and Results

During the progression of this project many tests have been conducted and some have failed but others have succeed. Each technology brought vantages and limits that were used in this work trying to reach better and better results or underlining new problems to face. This chapter contains all majors experiments done with their results and all of the models trained. Particular focus is on ROBERTA model and it will be explained in detail and all steps performed will be shown: how it was fine-tuned, how it was trained and how it works. It reached some results and they will be discussed deeply.

4.1 Development Environment

To develop the entire project it was used a physical server provided by *Professor G. Moro* and *Google Colab*. The latter is a cloud service created to develop machine learning model on virtual environments. The models used in this work are very huge with millions of parameters and dataset contains millions of data, therefore the development environment must have the necessary resources. The two solutions used in the project are well explained in the following sections.

4.1.1 Server

The server provided for the project by professor *Gianluca Moro* is a powerful environment built to develop and train Deep Neural Networks. It has 32GBs of ram and a powerful CPU. It is an **Intel(R) Core(TM) i5-6400** that reaches the frequency of 2.70GHz with 4 cores. This server also has a powerful GPU **Nvidia TITAN Xp** with 12GB of dedicated Memory and full Cuda support. The operative system on this machine is Ubuntu and it comes with

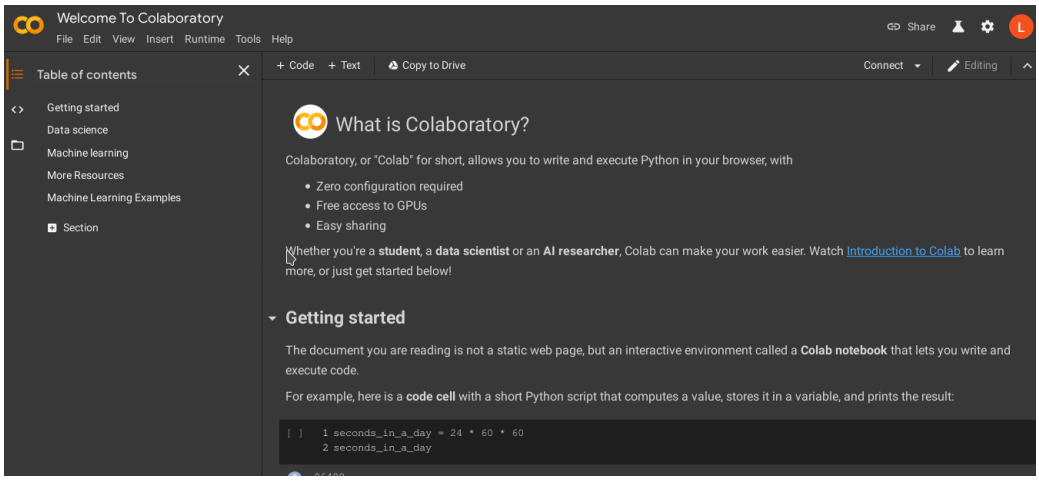


Figure 4.1: Colab Interface

all most important python frameworks as **Pytorch, TensorFlow, Numpy, ...** pre-installed.

4.1.2 Google Colaboratory

Google Colab is a cloud service provided by google (https://colab.research.google.com) and it is a virtual machine that has enough resources to train machine learning model and Deep Neural Networks. It comes with all frameworks you need and *pip* tool installed in order to let you catch all frameworks that are not present in the default environment. This service is based on **Jupyter Notebook**, in fact the basic interactions you can perform are to add code cells or text cells and to run them. But all notebooks the user can create are launched on virtual machine with a lot of resources. User can also choose which hardware deploy between three configurations:

- **None**. This configuration allows user to use just CPU
- **GPU Accelleration**. User can use GPU and Cuda.
- **TPU Accelleration** User can use TPU Technology.

The hardware of this environment is composed by a cpu **Intel(R) Xeon(R) 2.30GHz** with 2 available cores, 12GBs of ram but, in case user needs more, they can be expanded to 25GBs and a GPU **NVIDIA Tesla P4** with 8GBs of dedicated memory. This service is free but there is a limit about usage time. This service allows users to run short GPU tasks for free while for long GPU tasks user has to pay a fee. For these reasons some tasks as data preprocessing

or small neural network train have been conducted on Colaboratory while longer tasks as Fine-Tuning or big Network train have been executed on the server.

4.1.3 Python and Frameworks

To the project development the chosen programming language is Python version 3. It is one of the most successful language in the Data Science field because it is easy to use and learn but powerful whit a lot of well done frameworks created by the community. Furthermore python and all tools are free and that makes their usage, their solidity and their community really strong. The principal frameworks used for this project are:

- **Pickle**. It is a library containing functions to write and read python object from/to a file.
  https://docs.python.org/3/library/pickle.html

- **Numpy** to work with multidimensional vectors thank its *ndarray* object.
  https://numpy.org/

- **Pandas** to work with tables and relational data stored in *tsv* files.
  https://pandas.pydata.org/

- **SciPy**. The scientific version of numpy.
  https://www.scipy.org/

- **Pytorch**. Framework created by *Facebook* with the idea to develop high level functions to create and work with neural networks.
  https://pytorch.org.

- **TensorFlow**. Famous framework for Machine learning and Neural Network models creation.
  https://www.tensorflow.org/

- **Keras**. It is a framework born to provide to developer very high level functions to easily and with a few line of code create model and train them.
  https://keras.io/

- **Transformers**. Framework that provides easy access to state of art models of NLP tasks as Transformers, BERT, ROBERTA, ...
  https://github.com/huggingface/transformers

- **Sentence-transformers**. It's a Framework that contains the implementation of Sentence-Bert.
  https://github.com/UKPLab/sentence-transformers
- **Flair**. It's a very simple framework for state of the art NLP. It also contains Contextual Embeddings.
  https://github.com/flairNLP/flair

4.2 ROBERTA Model and Set-Up

The selected model for the project was ROBERTA Model and it was selected because it recently reached the state of art in many NLP tasks. It demonstrated itself robust and very precise beating a lot of other models. One possible alternative was represented by BERT model, but ROBERTA demonstrated its limitations getting better results and in many tasks it reached the state of the art surpassing the BERT Model. To better understand its advantages read the section 3.6 about ROBERTA. The selected model comes in more versions:

- **robera-base**. 12-layers, 768 hidden size, 12 heads, 125M parameters.
- **robera-large**. 24-layers, 768 hidden size, 16 heads, 355M parameters.
- **robera-large-mnli**. 24-layers, 768 hidden size, 16 heads, 355M parameters but it has been trained on MNLI dataset.

The better versions which have highest results, are the large ones. The roberta-large version got better result and it detains the state of art in some NLP tasks but it has a deep limitation. It is very big with a lot of parameters and the available resources deployed for this project wasn't enough so it was chose to work with a smaller model. So the final model is **Roberta Base**.

To get the model there are a lot of ways. You can develop the model on your own using pytorch or tensorflow but it can be hard, otherwise you can download a pre-build model trough one of the many repository online. For the project it was used **Transformers** as basic framework to get and to work with ROBERTA pre-build model. Trasformers is a easy framework to work with state of art models including ROBERTA and it provides numerous tools to help users in all phases of model development. Using it, every one can get with a few lines of code a pretrained models ready to use.

```
In [1]:  pip install transformers
```

The above code is for install transformers in your virtual environment using **Pip** tool.

```
In [2]: from transformers import RobertaModel
In [ 3]: model = RobertaModel.from_pretrained(
                "roberta-base")
```

The above code can be used to download a pre-trained base model. This model is pre-trained using general sentences about everything and this model works fine for NLP tasks on general arguments. In the case you want to specialize the model on a specific domain as Jobs like the case of this project you need some more steps. There are two main possibilities: training the network or fine tuning the model. In the first case all weights previously learned by the Neural Network are dropped and the model is trained against a dataset on that domain.

This option required a lot of data and a lot of time, in particular for neural network complex as ROBERTA and it often is not the best solution. The second option, fine tuning, means to use data about specific domain to train the network but without dropping previously learned weights. This forces network to adjust the knowledge learned before to fit it in the new domain. This approach is very fast and requires less data, in fact it is the most chosen solution when a complex pre-trained network need to be specialized.

4.2.1 Fine Tuning

The idea, now, is to use a well studied technique called **Cross-Domain** that force model to move previously learned acknowledged on one domain to another one. This approach is widely used because it permit model to work in domain where data are not enough for train. A lot of studies were conducted about that topic, applying this technique to all the fields. For examples, the work of Professor Moro[38] uses it in the sentiment analysis with success. The idea is to train model using a lot of reviews from heterogeneous source domains and that transfer that knowledge to a specific domain.

To specialized the network on the jobs domain it was necessary some textual data about jobs and a fine tuning script. Dataset contained a table called jobs.tsv that had a lot of job postings inside and, each of them, had title and description. The idea was to create an unique text file with all job titles and descriptions from the entire table that contained 1091923 different job postings.

ROBERTA, for the fine tuning phase, needs some text to read and nothing more so that made this task very easy. The fine-tuning script to use in order to automatize this process, was the one provided by **Transformer Framework Repository**. It is a pytorch script that loads the model and data and wants some training parameters, then it executes and manages the fine-tuning phase for the target network.

The data

The table selected to get some textual information about jobs was the one containing job postings: *jobs.tsv*. This table contains for each job posting: the title, the description and some requirements. Those three columns could be used to create a file containing a lot of text about jobs. However, requirements field often contains only some references to the description and, for this reason, it was dropped. First of all tables were loaded in a `pandas.Dataframe` that is an object for modeling relational tables. Useless columns and rows containing NaN cells were dropped. After that, all `JobTitles` and `JobDescriptions` were combined and saved in a textual file and at last the obtained dataset was split in two pieces: trainset and testset stored on two files: `jobs_descriptions_for_ft_train`(1,7G) and `jobs_descriptions_for_ft_eval`(87M).

Training Script

The repository of Transformers Framework also contains some useful scripts including the one for fine tuning. This scripts can be used for training a model and can be heavily customized using input parameters. The command used in this project to execute the fine-tuning of the ROBERTA model was:

```
python3 run_lm_finetuning.py
    --do_train
    --train_data_file=jobs_descriptions_for_ft_train
    --output_dir=roberta_ft_outputs
    --model_type=roberta
    --model_name_or_path=roberta-base
    --do_eval
    --eval_data_file=jobs_descriptions_for_ft_eval
    --mlm
    --save_step=300000
    --num_train_epochs=5
    --overwrite_output_dir
```

The above scripts performs the fine-tuning of the specified model through the parameter `-model_type` and `-model_name_or_path` to specify the version of the model to download or the directory where model was previously stored. Using the flag `-do_train` the scripts performs a training session and similar the with the flag `-do_eval` scripts performs evaluation after training. `-train_data_file` is used to tell to the script where training data are located and `-eval_data_file` do the same but for the evaluation data. `-output_dir` is used to specify the directory to store results and the new

model, `-overwrite_output_dir` is a flag that tells to the scripts to overwrite output directory in case it already exists and `-save_step` indicates the saving frequency for the model. Finally the flag `-mlm` tells to train the model using a masked language modeling and the parameter `-num_train_epochs` to specify how many training epochs perform. A lot of other parameters are set by default and can be changed. The most importants of them are:

- `-mlm_probability` default: 0.15 and it is the ratio of tokens to mask for masked language modeling loss.
- `-block_size` default: -1. The size of the block in which tokens are split before training. If -1 it will be used the largest for the selected model.
- `-per_gpu_train_batch_size` default 4. It is the batch size per GPU for training.
- `-gradient_accumulation_steps` default 1. It indicates number of update steps to accumulate before performing a backward pass.
- `-learning_rate` default $5e^{-5}$. It is the learning rate to use during train.
- `-weight_decay` default 0.0. It is the weight decay to apply.

This script produces a fine-tuned model saved into the output directory that can be used and load by using pytorch or other Transformers tools. Running the script the model `roberta-base` was downloaded and the file with the text for the train was opened. The text was turned into tokens using ROBERTA dictionary with 50k elements during a process called **Tokenization** and then, tokens were replaced with their ids. So the entire text was turned into a list of ids. After this phase the dataset was split in subsequences of 510 ids and two special tokens `<CLS>`(id 0) and `<SEP>` (id 2), converted into their ids, were added. One was placed at the beginning of the sub-sequence and the other at the end of it. An example of that phase can be seen at figure 4.2.

Now training could take place, this sub-sequences were the input of the model. First of all the 15% of the ids was masked with a `<MASK> token` because the task was: the model had to read the other ids in the input sequence and to try to guess the one under the mask. Model was trained for five epochs and for each epoch it did read the entire dataset. It was a big dataset with milions of sub-sequence, therefore it required some times to be complete.

4.3 Multiple Losses Training

After fine-tuning phase model had to be trained to fulfill a specif task. Model, thank to the previous training, obtained some knowledge about jobs

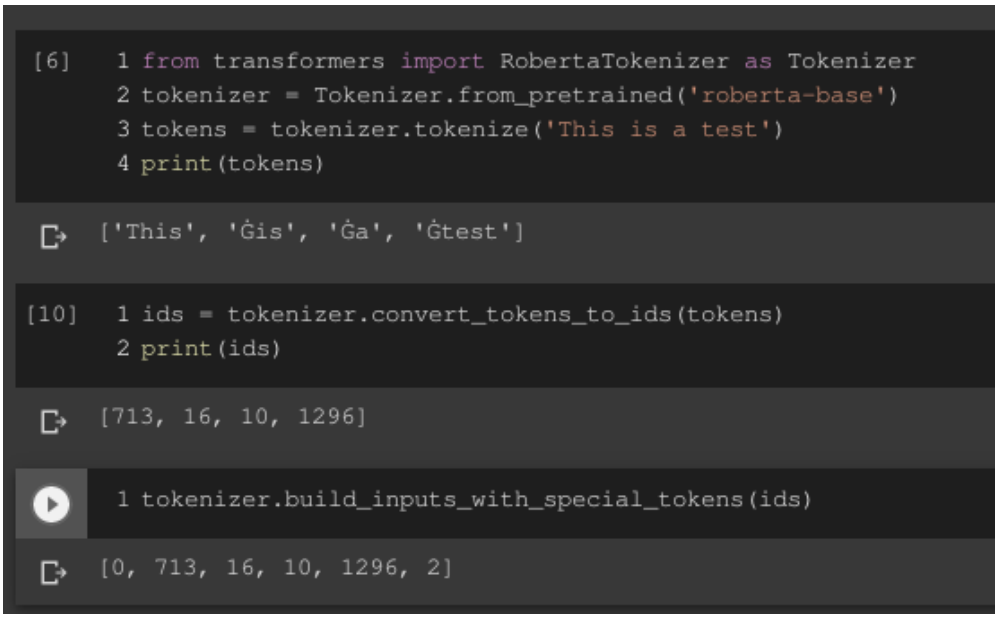


Figure 4.2: Tokenization performed by Roberta Tokenizer

in general, but it had to use those kind of notions to reach good results in some useful tasks. To do that model was trained again, but using a task level datasets. The tasks designed for that model were three and where binary classification tasks:

- **Job Title - Description**. For this task model had to express a probability that indicated how much a given description corresponded to a given title.
- **User - Job Application task**. For this task model had to process a user and an application and it had to express a real number that indicated the probability the user did that application.
- **Last Job Classification**. For this task model got in input a job history without the last job and a single job. It has to express a probability level that the single job is the last job of the given job history.

The following sections contain a detailed description of the above tasks and their results.

4.3.1 Job Title - Description

The first task for the model was to express a real number that was the probability that a given description was about a given job title. Those data,

job description and job title, came from Job Posting and the idea behind that task was to force the model to internally link information from description to the job title.

The problem was job history contained just a list of job title very short with no other information. To fulfill any kind of task on that data it was necessary the model was capable to know what a job title meant. This task was designed to let the model get that kind of knowledge by training it to link a job title to the information contained in the description.

This task was fundamental for the later experiments so it was performed as first. The dataset for this task was created using data from **Job Posting** table and the model, used for the classification, was the one fine-tuned in the phase before. After the training phase model achieved very good results guessing with a precision of 97,67%.

Dataset

For this task dataset came from job posting table. This table contained a lot of information about jobs and, for each job, it had the job title and the job description. That made that table the only solution to create this dataset. The idea was to create inputs that contained bot job title and a job description and, for each input, to assign a binary flag that expressed if the description was about title or not.

```
INPUT: <CLS> Job Title <SEP><SEP> Job Description <SEP>
FLAG: [0,1]
```

In order to create this dataset the following steps were performed:

1. **Dataset Preprocessing**. Dataset contained NaN cells and they have to be removed. Dataset also had more data then we need, so the rows containing NaN values were just dropped.

```
In [1]:
import pandas as pd
dataset = pd.read_csv(
    "jobs.zip",
    sep = '\t'
    error_bad_lines = False
)

In [2]:
dataset = dataset[['Title', 'Description']]
```

```
In [3]:
dataset.dropna( inplace=True )
```

After that, dataset still remained too large so it was selected just a part of it. For this task it was chosen to use 90000 jobs (60000 for training and 30000 for test).

2. **Create true and false examples**. Now dataset was composed by `Job Title` and its `Job Description` so it was necessary to turn dataset into a list of tuple containg job title, job description, flag.

   ```
   ( Job Title, Job Description, Flag)
   ```

   Now dataset had to be split in two parts of 45000 elements which the first part contained the true samples and the second the false ones. In order to create that list of tuples, the elements form the first true part were taken and placed in a tuple adding the right flag and then appended to a list. Elements from second part were added in the same way but before, the descriptions were randomly shuffled. Doing that each job title was paired to the description of another job.

3. **Roberta Tokenization**. To feed the model with data they had to be turned into tokens. To do this it was used the **tokenizer** from the library **Transformers**. However, before this phase, HTML tags had to be removed by using the following code:

   ```
   In [1]:
   re.sub('<[^>]*>',' ', desc, flags=re.I|re.U)
   ```

   After HTML tags elimination the tokenization process started. To tokenize a sentence it was possible to use the following command:

   ```
   In [2]:
   from transformers import RobertaTokenizer as tokenizer
   tokenizer.encode(jobTitle, jobDescription)
   ```

   There were cases where description was too long and the block size limit (512 ids) was reached. In this case description was truncated taking a piece of it from a random point, but enough large to fit the block size. in other cases Job Title and Job Description were too short and the block size was less than 512. For these cases, ROBERTA Team suggests to pad the block with zeros and add a mask. Mask was an array composed by 0 and 1 where 1 represented data from dataset and 0 represented pad values. After this step data appeared:

```
( RobertaInputBlock, Mask, Flag )
```

After the above steps dataset was ready to be used to train model.

Classification Model

The classification model was composed by two parts: **ROBERTA model** and a **binary classifier**. The model fine-tuned was just ROBERTA model without any classifier but, according to the paper, one of the key points of BERT and so ROBERTA was: they could be trained adding other layers on the top of them using all or some of the model outputs. The model produces vectors with 768 dimensions, one per input tokens, so it produces embeddings also for the `<CLS>` and `<SEP>` tokens.

For the classification tasks authors suggest to use the output vector from the first token `<CLS>` and to give it in input to the classifier. This chain doesn't interrupt the back-propagation and both, model and classifier, can be trained together. For the project the selected classifier was the one that came with the **RobertaForSequenceClassification** model included in the **Transformers** repository. It contains a lot of tools and models to cover most NLP tasks. The selected model it was composted by ROBERTA pre-trained model and a classifier, but it was possible to change ROBERTA model with the one fine-tuned, from the previous task, and than to train the whole model again for this new task.

```
In [1]:
from transformers import RobertaForSequenceClassification
model = RobertaForSequenceClassification.from_pretrained(
            'path_to_finetuned_model'
)
```

The classifier mounted on the top of ROBERTA was a two layers classifier. It was composed by a dense layer with a non-linear activation function: **Tanh**.

$$tanh(x) = \frac{\exp^{x} - \exp^{-x}}{\exp^{x} + \exp^{-x}} \tag{4.1}$$

This first layer was composed by 768 neurons so it produced as output a vector of the same size of the input one, but performing the following equation:

$$tanh(\vec{X} * \mathbf{W} + \mathbf{B}) \tag{4.2}$$

The second layer was composed by as many neurons as the number of the classes were, but for binary classification just one was needed. So, at the end, classifier produced a real number that could be used as affinity ration between input couples. For this task it represented the probability of the description to be related somehow to the given job title.

Training

Model was trained using the trainset composed by 60000 samples balanced between positives and negatives for just 2 epochs using a batch size of 8 elements. Greater batch size would had led to a Cuda memory crash so it was impossible test with different batch size. The target function was **CrossEntropyFunction** that works fine with classification problems.

$$H(p,q) = -\sum_{x \in X} p(x) * \log q(x) \tag{4.3}$$

The selected optimizer for the train was **AdamW** with a learning rate of $5e^{-5}$ and an ϵ equal to $1e^{-8}$. After the train model was tested using the testset composed by 30000 data of the same kind of the trainset and it achieved very good results:

- `Precision: 97,67%`
- `Accuracy: 97,45%`
- `Recall: 97,22%`

As said before, the model produced a real number that had be used to decide if the input sample was in the class or not. To do so, it was used an empirical value of 0.6 and all the samples for which model generated a number lower were considered 0 and the others 1.

4.3.2 Users to Job Application Task

The second task was about train a model to guess if a given user submitted a given application. For this task the inputs were pairs user-application and, likely to the first task, the output was the probability that the user performed the given application. The users was defined by combining some of its studies information and its job history while for the application was used just the job title. For this task model reached good results with an accuracy of 91,95%.

Dataset

Differently from the first task this dataset did not come from a single table but it needed data from more tables:

- **users.tsv**. This table contained information about study level.
- **user_history.tsv**. This one contained the job history of each user.

- **apps.tsv**. This table contained the userids of the users that submitted applications and the jobid of the job posting.
- **jobs.tsv**. This table contained the job title of the jobs.

To create the dataset it was necessary to merge data from all of the above tables. Final data was in the following format:

`Input= <CLS> Users information, job history <SEP><SEP> Job Title <SEP>`

In order to create that dataset these following steps were performed:

1. **Dataset Preprocessing**. Tables contained Nan values and they had to be removed. Dataset had more data then it was necessary, so the rows with Nan cells were simply dropped.

```
In [1]:
import pandas as pd
dataset = pd.read_csv(
    "table.tsv",
    sep = '\t'
    error_bad_lines = False
)

In [2]:
dataset.dropna( inplace=True )
```

2. **Group applications and job histories by user**. The next step after cleaning the data by the NaN values was to group data from apps.tsv table by the users. In this way it was easier to get from each user its own applications.

```
In [3]:
grouped_app = apps.groupby( by=UserID )
grouped_jobs = job_history.groupby( by=UserID )
```

3. **Join the tables**. Once all tables were indexed by UserID they had to be merged. To do this it was possible to use pandas the merge command, but in the project this join was made by using a loop cycle over all the job histories, getting the users and its applications and creating the new dataset. From each user table it was kept the State, the DegreeType, the Major, the TotalYearsExperience and the WorkHistoryCount while, from the jobs, just the JobTitle of the job application.

4. **True and false sample creation**. As in the previous task the dataset, by now, contained only positive examples of users and their applications. In order to create also negative items, dataset was reduced to 90000 samples and half of them were turned into negative samples just shuffling the job applications.

5. **Roberta Tokenization**. To feed the model with the data they had to be turned into tokens. To do this it was possible to use the **tokenizer** from the library **Transformers**. However, before this phase, HTML tags had be removed by using the following code:

```
In [1]:
re.sub('<[^>]*>',' ', desc, flags=re.I|re.U)
```

After the HTML tags elimination step, the tokenization process started. To tokenize a sentence it used the following command:

```
In [2]:
from transformers import RobertaTokenizer as tokenizer
tokenizer.encode(users, jobTitle)
```

The lists of tokens, generated by this way, often were smaller than block size (512) so they were padded with zeros and a mask vector was created for each sample. This vector contained 512 cells with 1 if the same cell in the input vector was an ids or 0 if it was a padding value.

After those steps the data were in the right format to be used by the model.

Training

The model was the same of the previous task. After it, the model was saved to be used for this task and, doing this, a chain of losses was created in order to train a single model. For this task, model was trained on a training set of 60000 samples that was balanced between positives and negatives. It was trained for just 2 epochs using a batch size of 8 elements. Greater batch size would have led to a Cuda memory crash and, even for this task, it was impossible test the model with different batch size. The target function was **CrossEntropyFunction** that worked fine with classification problem. See equation 4.3. The selected optimizer for the training was **AdamW**, like in the previous task, with a learning rate of $5e^{-6}$ and an ϵ equal to $1e^{-8}$. After training phase model was tested using the other 30000 data and, again, it reached very good results.

- `Precision: 94,71%`
- `Accuracy: 91,95%`
- `Recall: 88,86%`
- `F-Measure: 91,69%`

As for the previous task it was used 0.6 as split value and all values produced by the model below of it were considered 0 and all values above were considered 1.

4.3.3 Job History to last Job Task

The last task was to train the net to guess if a given job was the last job of a given job history. This was the most complex task because, to fulfill the task, model had to understand the meaning of the job titles hand-written in the job history. The idea was that: the other 2 tasks, on which the model were trained until now, helped the model to better understand job titles and to get good results even for this task. Dataset for this task was composed by samples in this format:

```
<CLS> USER user's info JOB job history <SEP><SEP> last job <SEP>
```

Also for this task more tables had to be merged in order to give to the model some user's information for a better classification. After the train model reached a good result. It classifies the input samples with an accuracy of 75%

Dataset

The data for this dataset came from the following tables:

- **users.tsv**. This table contained information about users.
- **user_history.tsv**. This table contained the job histories of each user.

To create the dataset for that task the following steps were performed:

1. **Dataset Preprocessing**. Like the previous tasks, tables contained Nan values so the rows with those values had to be dropped.

```
In [1]:
import pandas as pd
dataset = pd.read_csv(
    "table.tsv",
```

```
    sep = '\t'
    error_bad_lines = False
)

In [2]:
dataset.dropna( inplace=True )
```

2. **Creating Job History**. Users_history table contained a row for each job indicating the user that did that job. It was better to groups jobs by the user that did them using the command `groupby` provided by pandas dataframe.

```
In [3]:
grouped_jobs = job_history.groupby( by=UserID )
```

3. **Last job extraction**. Now performing a loop over the job histories it was possible to extract the last job from them and to create a list of tuples with that format:

```
( UserID, JobHistory, LastJob)
```

4. **Add user's information and flag**. For each tuple, using the UserID, it is possible to find the related user and his information and put them in the dataset. The selected information of the user used in this task were: **DegreeType**, **Major**. In the same time it was possible to add the flag that, for now, it was positive creating data in this format:

```
(User + JobHistory, LastJob, Flag)
```

5. **Creating negative samples**. By now, the job histories are more than 300k so dataset needed to be reduced to 90000 samples. After that reduction, from those selected samples which were just positives, it was necessary to create negative samples. Dataset was split in two equal parts and one part it was turned into negative samples shuffling the last jobs.

6. **Tokenization**. Like the other tasks, the entire dataset had be converted into tokens ids. To do this it was used the default Tokenizer provided by transformers framework.

```
In [4]:
from transformers import RobertaTokenizer as tokenizer
tokenizer.encode(JobHistory, LastJob)
```

Training

Once dataset was created, it was split in the trainset, with 60000 samples, and testset, with the other 30000 samples, then model was trained. Model was the one from previous task that was ready to be trained for the third and last task. For this task, the model was trained for 2 epochs using a batch size of 8 elements. As in the previous tasks the loss function was **CrossEntropyFunction** and the optimizer was **AdamW** with a learning rate of $5e^{-6}$ and a ϵ of $1e^{-8}$. After this train phase, the model was tested using the testset and it reached the following results:

- `Precision: 73,34%`
- `Accuracy: 74,59%`
- `Recall: 77.25%`

For this task the empirical split point used was 0.4.

4.4 Further tests and considerations

Model reached very good results in two tasks: to express the probability that a given user submitted a given job application and that a given job history with some user's information contained, as last job, a given one. These two tasks were tested using a balanced dataset, but the reality it's different.

The first task, that link a user to its job application, in a real scenario had find the right applications between a lot of wrong ones. They aren't necessarily wrong and the application done are not the only ones right. therefore, the number of good job postings for a given user is very smaller than the wrong ones. The same concept is correct even for the other task, so the model were tested with unbalanced datasets from both of these tasks.

Another parameter out of the model is the split point for the classification. Samples can be positive or negative so they can flagged as 0 or 1, but the output of the model is a real number. It was selected empirically a number called **Split Point** equal to 0.6 for both the tasks and for all the samples model produced a value. If that value was above the Split Point it was considered 1 otherwise it was considered 0. It was interesting to use different Split Points with the unbalanced datasets.

4.4.1 User to Job Application tests

For these tests 5 datasets were created by using unseen data. The datasets had different rations between positive and negative samples: 1-1, 1-2, 1-4, 1-8,

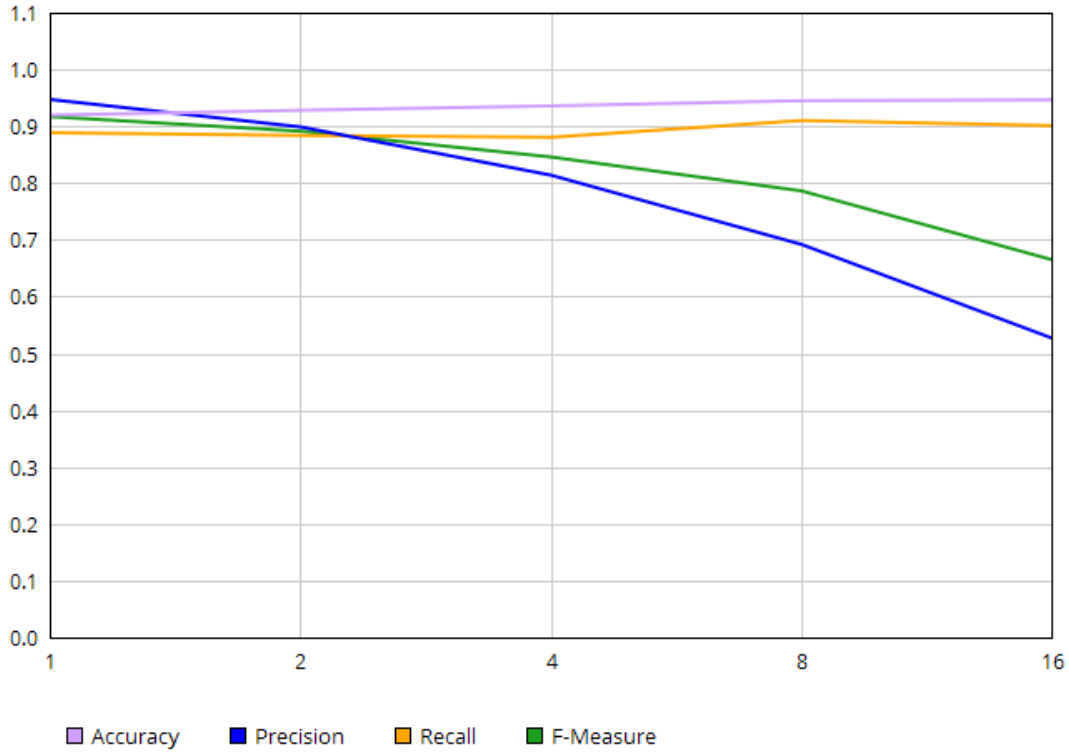


Figure 4.3: Model results tested by using unbalanced datasets

1-16. All of them were composed by 60000 never seen before samples and the metrics to evaluate the model were:

- **Accuracy**: $\frac{TP+TN}{TP+TN+FP+FN}$
- **Precision**: $\frac{TP}{TP+FP}$
- **Recall**: $\frac{TP}{TP+FN}$
- **F-Measure**: $2 * \frac{Precision*Recall}{Precision+Recall}$

As it is possible to see in the figure4.3 model demonstrated itself to work really good with balanced or slightly unbalanced datasets (1-1, 1-2, 1-4), but with strong unbalanced dataset the precision and so the F-Measure fell down.
Metrics for dataset 1-4:

- `Accuracy`: 93.58%
- `Precision`: 81.37%

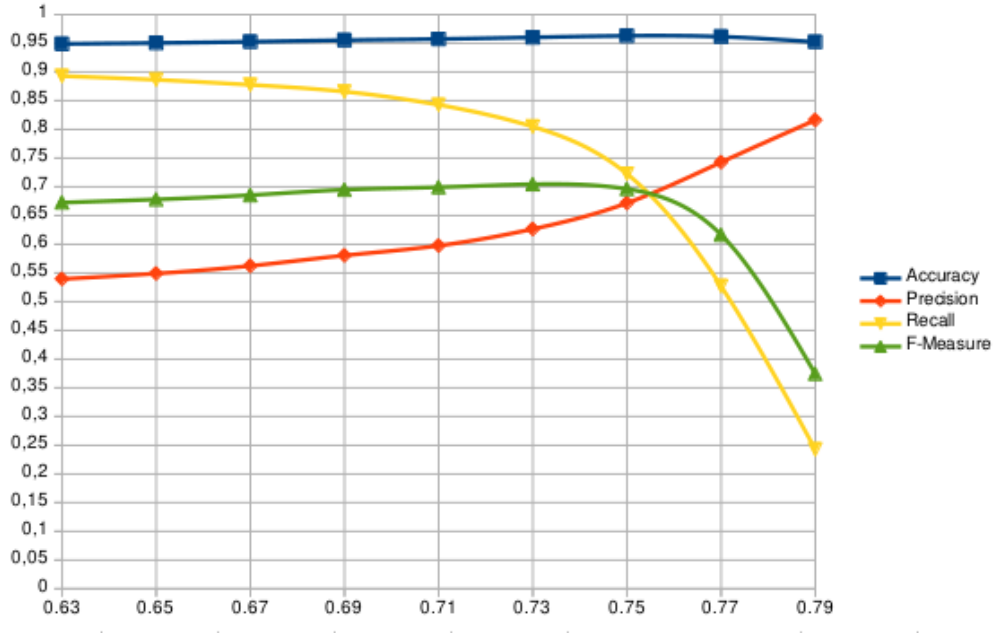


Figure 4.4: Metrics variation using different split points

- `Recall`: 88.06%
- `F-Measure`: 84.58%

Metrics for dataset 1-16:

- `Accuracy`: 94.66%
- `Precision`: 52.68%
- `Recall`: 90.11%
- `F-Measure`: 66.49%

This probably happened because the model tended to classify as positive class easier then negative. This brought to have almost all of the positive samples selected but with some negative samples within.

This is good, or not to bad, if system has to provide to an user a list of job postings and user has to choose between them because it guarantees the right job postings are shown. But for a company looking for the best candidates the situation is harder because it doesn't want to select a wrong one. For this case it would be better to get less positive samples but without negative ones within.

For all these tests it was used the same Split Point 0.6. It was also interesting to check result variations by using different values. By now, the model had favored the positive class, so the idea was to use higher threshold in order to help a better recognition of negative samples and to improve precision even worsening a bit the recall.

The model was tested using the dataset 1-16 but with different split points: 0.63, 0.65, 0.67, 0.69, 0.7, 0.71, 0.73, 0.75, 0.77, 0.79, 0.8. As it is possible to see from the results shown in the figure 4.4, the value of the Split Point played a very important role in the classification. Raising the value to 0.73 as results demonstrate, F-Measure was maximized so the ration between precision and recall was maximum.
Metrix for 0.73 split point:

- `Accuracy`: 96.03%
- `Precision`: 62,65%
- `Recall`: 80,48%
- `F-Measure`: 70,45%

Metrix for 0.63 split point:

- `Accuracy`: 94.87%
- `Precision`: 53,95%
- `Recall`: 89,29%
- `F-Measure`: 67,26%

Increasing the split point over 0.73 precision kept to raise but recall fell down quickly. This demonstrates that for unbalanced model to study the split value can lead to a better results.

4.4.2 Job History Last Job Test

Even for this task model was tested against unbalanced datasets in the same proportions of the previous: 1-1, 1-2, 1-4, 1-8, 1-16. But the previous task model worked really fine for balanced model so it had a good start point. For the second task it was impossible to say the same. It got some good results but much worse than in the other task.

Like for the other tests metrics used for the evaluations were: `Accuracy`, `Precision`, `Recall` and `F-Measure`. After these tests results showed the

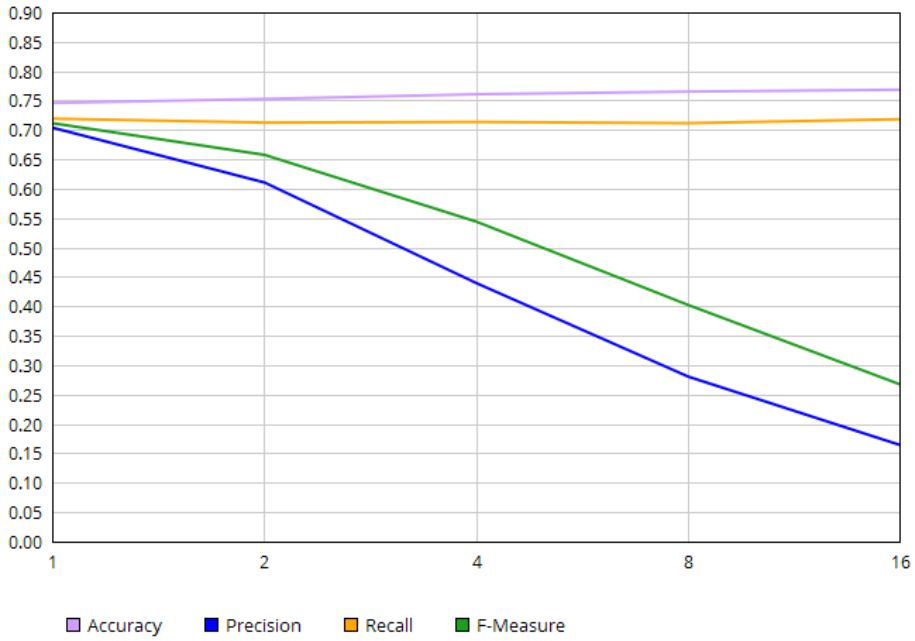


Figure 4.5: Results of the model in last job classification task using unbalanced datasets

difficulty of the model working with unbalanced datasets. From the dataset 1-2 the results started worsening a lot, in particular the Precision.
Metrics for dataset 1-2

- `Accuracy`: 75.27%
- `Precision`: 61.06%
- `Recall`: 71.25%
- `F-Measure`: 65.76%

Metrics for dataset 1-16

- `Accuracy`: 76.84%
- `Precision`: 16.42%
- `Recall`: 71.83%
- `F-Measure`: 26.73%

This results show the difficulty to work with job written in the same job history. It can be possible to see the job posting as the last job of the current job history but in the future. If users and company accept to work together the job posting, somehow, becomes the last job of the users job history.

Under this light the tasks are very similar but the results are so different. One of the main difference between the two tasks is the first has to classify the job posting title that have more words and more information, while the second case it has to classify a job title hand-written by the user that has no gain adding more information. This could be a point to start for a future work for improving this model.

Chapter 5

Other Experiments and future works

During the evolution of this project other experiments were done and other models were tested without reaching good results. They need to be explained anyway to specify what technology worked and what didn't for future studies in this field.

The initial idea was to predict the last job from each job history. Job histories were defined as matrices J_{NXD} where N is the number of the jobs contained in the job history and D is the dimension of the vector used to represent jobs called **Job Embedding**. A good representation of the jobs leads to create a Jobs Vector Space where similar jobs are closed together and different jobs are placed at some distance computed by using *Cosine distance*. This vector space is fundamental in order to create a model capable of predict the last job.

$$J_{NXD} = \begin{pmatrix} j_{1,1} & j_{1,2} & \cdots & j_{1,d} \\ j_{2,1} & j_{2,2} & \cdots & j_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ j_{n,1} & j_{n,2} & \cdots & j_{n,d} \end{pmatrix} \tag{5.1}$$

Last job extracted from the job history is $\vec{j}_n$ and the other jobs in sequence are the vectors $\vec{j}_0, \vec{j}_1, \ldots, \vec{j}_{n-1}$ that represent the jobs user did before the job to predict. As in NPL task where the model has to create a probability distribution over a natural language dictionary after reading some words to predict the last, in this case the model has to read the previous jobs and predict the last. These tasks look really similar but they have some deep differences and they deserve to be well described to fully understand the results.

- **Dictionary Size**. The words in natural language can be collected into a dictionary with 30K (as BERT Dictionary) to 50K (as ROBERTA

Dictionary) items. For each words exists tons of examples and usages which model can be trained with. **Job History** are hand-written and that fact leads to have an enormous dictionary because each job can be written in a lot of different ways. Our dataset has a dictionary of 270k jobs for 360k Job Histories. That means that most jobs have one occurrence and that makes a prediction really hard.

- **Context Size**. For NLP context size is limited by just the model input limits however the context can be very long and full of useful information. Using the Job Histories context is limited to the size of them which generally have 6 or 7 jobs and this makes context to be very short.
- **Dataset Size**. Last difference, but it played a key role in the project, was the dataset size. Natural Language Model are trained over dozens of GBs of textual data with billions of samples. Dataset for the project contains only 360K job history which aren't few but not enough to overcome the problems above.

One of the first ideas to approach the task was to convert all the jobs inside the job histories into Job Embeddings hopping to create some density based clusters and, somehow, to reduce the size of the jobs dictionary. After that the idea was to create a model that has to read the sequence of jobs in the job history and to try to predict the embedding of the last job. Then it has to take the prediction and to check the closest K jobs in order to find the target job using cosine similarity.

This task has been repeated using different models and technologies but without reaching any good results. After this first difficulty it was decided to try different paths, one of them was to test this new technology very promising called **BERT** and the model **ROBERTA** explained in the section 4.2.

Other paths were to avoid explicit Job Embeddings trying to use an Embedding layer inside the model and producing a probability distribution from the last layer output. This strategy was too prone to dataset problems and no good results were achieved. The only good results obtained are from ROBERTA model in binary classifications task but predicting the last job, using this dataset, is still an open task and it needs further experiments and investigations.

5.1 Last Job Prediction using Job Embeddings

For this task some embedding models were used like **Flair**, **BERT**, **Sentence-Bert**. All of them promise to create the best word embeddings but none of them could help in this task. The task was to guess the last jobs after reading the previous jobs in the job history. All jobs were converted in job embeddings

before the task using one specific model. The Neural Network deployed for the prediction was based on an LSTM and the loss function used was **cosine distance** between the output of the model and the job embeddings of the target jobs.

$$X = \begin{pmatrix} \vec{j_1} \\ \vec{j_2} \\ \vdots \\ \vec{j_{n-1}} \end{pmatrix} \tag{5.2}$$

$$Y = \vec{j_n} \tag{5.3}$$

$$Y' = Model(X; \sigma) \tag{5.4}$$

$$loss = ConsineDistance(Y', Y) \tag{5.5}$$

During tests the model was evaluated by using a special function called **KJobsAccuracy** that checked if within the K closest jobs to the predicted vectors there was the target job. So, given a certain K, it was possible for the model to evaluate its accuracy.

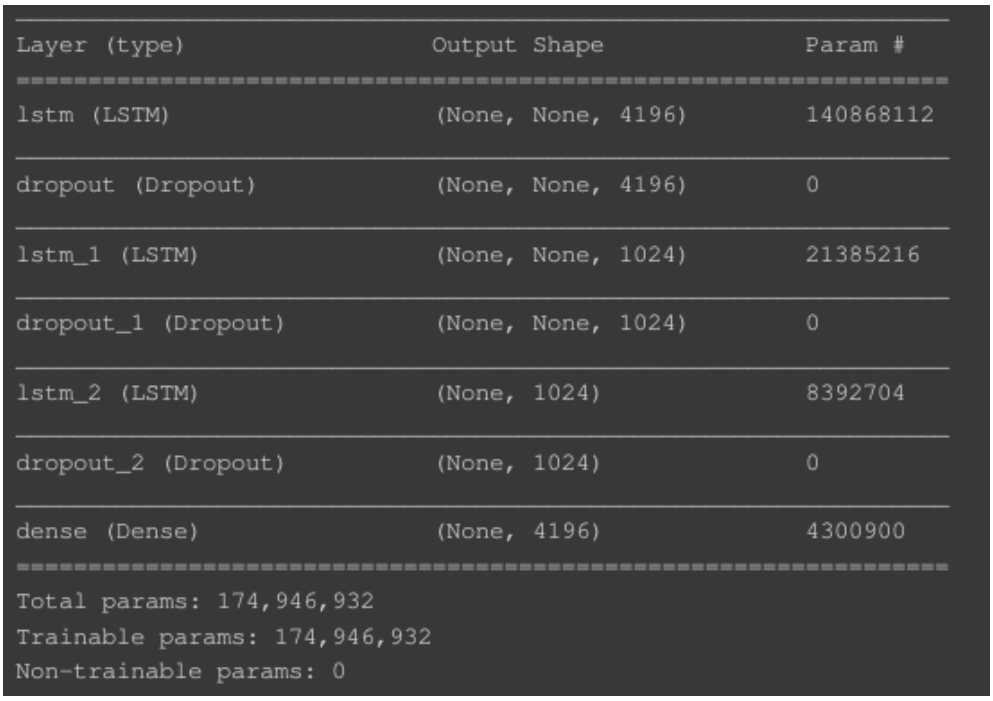
$$KJ = KClosestJobs(Y', J, K) \tag{5.6}$$

$$KJobsAccuracy = \frac{\sum_i^N res_i}{N} \tag{5.7}$$

$$res = \begin{cases} 1 & \text{if } Y \in KJ \\ 0 & \text{if } Ynot \in KJ \end{cases} \tag{5.8}$$

First test was performed by using **Flair** contextual embeddings created by chaining *Forward Vectors* (2048 dimensions), *Bakward Vectors* (2048 dimensions) and **Glove Vectors** (100 dimensions). In that way the final vectors had 4196 dimensions. Each Job Embedding was created converting all words in the job title into vectors and mixing them using mean pooling function. After that job histories with less than 3 jobs were dropped and the others were zero padded or truncated in order to have 5 jobs plus the target job in each job history.

The model, shown in picture 5.1, was created using **Keras** with an input shape of 4196 size, two hidden layers of 1024 neurons and a Dense Layer of 4196



Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, None, 4196)	140868112
dropout (Dropout)	(None, None, 4196)	0
lstm_1 (LSTM)	(None, None, 1024)	21385216
dropout_1 (Dropout)	(None, None, 1024)	0
lstm_2 (LSTM)	(None, 1024)	8392704
dropout_2 (Dropout)	(None, 1024)	0
dense (Dense)	(None, 4196)	4300900

Total params: 174,946,932
Trainable params: 174,946,932
Non-trainable params: 0

Figure 5.1: Keras representation of the model used for job prediction

neurons. So this model could take in input a sequence of 5 jobs embeddings with 4196 dimensions and it could produce a vector of 4196 size like the other Job Embeddings. The activation function for all the layers was Tanh and the output was compared to target job by using **Cosine Distance** as loss function.

$$CosineDistance(\vec{y}, \vec{y'}) = \frac{\vec{y}\vec{y'}}{||\vec{y}||||\vec{y'}||} \tag{5.9}$$

. The model was trained with 18000 job histories for 60 epochs and tested using others 10000. It was trained using a learning rate of $1e^{-3}$ and **RMSProp** for optimizer. The average of cosine distances produced during test was 0.327 and this showed model didn't learn how to predict the last job. In the training set cosine distance reached 0.074 and this fact was caused probably to an over-fitting. Model was also evaluated using *KJobsAccuracy* function with 10k jobs from test set but result was 25,3% with K=500.

To handle this problem other two experiments were conducted. In the first case 2 dropout layers (with 0.2 value of dropout) were added to the model after the two hidden layers and in the second case, model (with dropout) was trained using 60000 data. The train phase was equal for both of them, but results didn't improve. The first reached 25 % of accuracy using *KJobsAccuracy* while the other reached 27%. Maybe, flair contextual word embeddings were not

sufficient to represent jobs correctly so it was decided to test model with other Job Embeddings.

Job embeddings were recreated by using first BERT and then Sentece-BERT. It's important underline that BERT Model has no official embedding extraction methods. Using BERT3.5 to generate word embeddings is more similar to an empirical procedure because it has a lot of internal layers and the embeddings can be extracted from all of them or concatenating some of them.

Many different embeddings were created and best results were obtained concatenating the first two layers using mean function. Sentence-BERT3.7, differently, is a model that uses BERT but it is trained to create vectors which can be compared using cosine distance. So jobs in job histories were turned into Job Embeddings using both of them and tested using LSTM model written in Keras.

The model, equal for both the tasks, was composed by three layers of LSTM and a dense layer as output layer. The first layer had an input shape of 768 as the input vector size, the second and the third have 1536 neurons while the last dense layer has 768 neurons. The output was compered with target by using *Cosine Distance* as loss function. BERT embeddings with this model reached an accuracy using *KJobsAccuracy* function of 19,2% and, using Sentence-BERT, they reached 27,1%.

None of the obtained results was good enough so these results brought to light the problem about job embeddings quality. Jobs hand-written have some problems like polysemy or synonymy that lead model to hardly create feature vectors that well represent the jobs. How can a model predict the last job in a meaningless vector space?

5.2 Possible solutions and future works

Jobs in a Job History are sequences of data a Recurrent Neural Network can use. These kind of Networks showed good results in sequence tasks so probably the problem is not the predictor model but the jobs representation that does not provide enough information to fulfill the task.

However ROBERTA Model reached good results in the binary classification tasks using jobs in job history (See section 4) and creating internally the job embeddings using attention mechanism. It was impossible extract these embeddings but somehow and somewhere these useful information were computed and turned in some feature vectors. Maybe it was the Attention mechanism the key point to solve the problem but the concept that this works wants to highlight is: the best results come from model that internally creates job embeddings.

This fact suggests that granting model more autonomy better results it

can reach and this fact is also highlighted by the history of the Deep Neural Networks models. The first and easiest model, the **Feed Forward Model** has just the power on its weights and nothing more in fact it was used with good results in very easy tasks . The Neural Networks that come after starts to get more tools to decide alone how to handle data. Recurrent Neural Networks are capable to handle sequences and to decide how much information keep and how much forget from the old data. Attention mechanism allows network to learn which data have more information and which are useless and then select just the useful ones. Convolutional Neural Networks, for example, can decide to which features give more importance and to which give less.

All of the above networks reached good results first in his field and then in others improving all deep neural network world. From them is possible to see that the common characteristic from all is they get some autonomy. In the future, networks with more autonomy can obtain better and better results and maybe, this can be a future direction also for researches in Job Recommendation filed. Following this idea the future network for this task may have more autonomy to map job title hand-written to a vector space created directly within the network and not before and then to decide which works are important and which meaningless. Following this idea a proposed model that resume those concepts may looks like the one in the figure 5.2.

This model is composed by the following layers:

- **Embeddings Layer**. This layer has the duty to create best embeddings for Job Title representation. Results showed BERT was very good to extract these features so this layer is just a BERT sub-layer composed by **Self-attention Mechanism** and a feed forward neural network. (Check section 3.5 relative to BERT for more information)

- **Recurrent Neural Network**. This layer is composed by a Recurrent Neural Network like **LSTM** or **GRU**. The idea is to have a model that can check the sequentially of the jobs in the job history and models concepts like careers.

- **Output Layer**. In the picture 5.2 the last layer is a softmax layer that produces a distribution over a job dictionary. This requires a well done Job Dictionary maybe created using some clustering technique.

The model proposed is just an idea to assemble all concepts expressed above but more investigations and tests must be conduct in this direction to prove the goodness of what has been said.

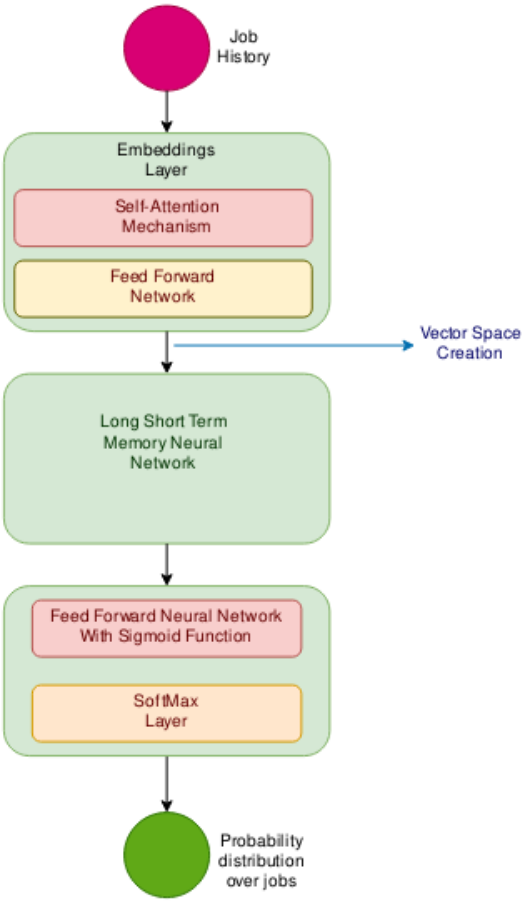


Figure 5.2: Idea for future models for job prevision

Conclusions and future prospects

The initial problem was about the Job Recommendation system and how new technologies could help it. New Deep Neural Networks have shown themselves to surpass results ever achieved before becoming one of the most promising technologies of this historical moment and bringing solution in a numerous fields from artificial vision to decision support.

This project wants to focus on the Job Recommendation world because it has been subject to recent attention from major companies. This field relies on old machine learning models that have limits which can't be easily overcome and on well structured data. It is important to test this system with the Deep Neural Networks. The dataset used comes from a Kaggle Competition and brings some issues which need to be addressed. One of them is data in Job Histories are Job Titles hand-written by each users introducing **polysemy** calling the same job with different names and **synonymy** calling two different jobs with the same job title. Another problem is the lack of a context that can help better understand those titles and improve the predictions.

The initial task was to create a model capable of predicting the last job of each job history by reading the previous job titles, but things turned out to be harder than imagined. Using different kinds of **Job Embeddings** it was impossible create a good vector space to well represent the job titles and none of the models were able to predict the last job. So the task was modified to become a binary classification task in order to test BERT Model. The task was to predict if a given Job Title was the last job of a given Job History.

A model based on the BERT model and called ROBERTA (please refer to 3.6) was used for this task. It used an attention mechanism to handle fixed sized sequences and got very impressive results in some NLP tasks. The model was fine-tuned and trained with two other tasks to better prepare it for the last job classification. The other two tasks were: guess if a job description was about a given job title and guess if a certain job application was submitted by a given user. The model demonstrated itself to be very efficient in these tasks reaching very good results (over 90% of accuracy in a balanced dataset)(Please refer to 4). With no doubt about its potential it was tested against the Job Classification Task. The results were good, but did not achieve high accuracy:

74%.

These results underline how inaccurate hand-written job titles can be represented as vectors for job recommendation task. For sure a lot of models and new technologies can be tested in this task, perhaps reaching even better results. Indeed, this work has highlighted the need of further research and a specialized Deep Neural Networks field. Out of all the possibilities one of them seems to be more interesting: Autonomy. The model that reached the best results was ROBERTA and it was the one that created embeddings internally instead of getting them in input. Maybe if the model were able to extract features by itself, it would be possible to achieve what today seems impossible and obtain better results even in more demanding tasks such as the one dealt with in this project.

Thanksgivings

The first person I want to thank is my Professor Gianluca Moro for all the support he gave to me, feeding my personal interesting for this marvellous word and making this work possible. The second person I want to thank is my girlfriend Lia for the moral support and for the patience she always had, accompanying me during this project. Thank to my family for its sustain, to all of my friends and to all the people wanting or not wanting helped me during this piece of my life.

Bibliography

[1] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In Shilad Sen, Werner Geyer, Jill Freyne, and Pablo Castells, editors, *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*, pages 191–198. ACM, 2016.

[2] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. Character-aware neural language models. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 2741–2749. AAAI Press, 2016.

[3] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *CoRR*, abs/1505.00387, 2015.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[5] Giacomo Domeniconi, Gianluca Moro, Roberto Pasolini, and Claudio Sartori. A study on term weighting for text categorization: A novel supervised variant of tf.idf. In Markus Helfert, Andreas Holzinger, Orlando Belo, and Chiara Francalanci, editors, *DATA 2015 - Proceedings of 4th International Conference on Data Management Technologies and Applications, Colmar, Alsace, France, 20-22 July, 2015*, pages 26–37. SciTePress, 2015.

[6] Giacomo Domeniconi, Gianluca Moro, Andrea Pagliarani, and Roberto Pasolini. Learning to predict the stock market dow jones index detecting

and mining relevant tweets. In Ana L. N. Fred and Joaquim Filipe, editors, *Proceedings of the 9th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - (Volume 1), Funchal, Madeira, Portugal, November 1-3, 2017*, pages 165–172. SciTePress, 2017.

[7] Alan Akbik, Duncan Blythe, and Roland Vollgraf. Contextual string embeddings for sequence labeling. In *COLING 2018, 27th International Conference on Computational Linguistics*, pages 1638–1649, 2018.

[8] Jonathan L. Herlocker, Joseph A. Konstan, and John Riedl. Explaining collaborative filtering recommendations. In Wendy A. Kellogg and Steve Whittaker, editors, *CSCW 2000, Proceeding on the ACM 2000 Conference on Computer Supported Cooperative Work, Philadelphia, PA, USA, December 2-6, 2000*, pages 241–250. ACM, 2000.

[9] Justin Basilico and Thomas Hofmann. Unifying collaborative and content-based filtering. In Carla E. Brodley, editor, *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*, volume 69 of *ACM International Conference Proceeding Series*. ACM, 2004.

[10] Brent Smith and Greg Linden. Two decades of recommender systems at amazon.com. *IEEE Internet Computing*, 21(3):12–18, 2017.

[11] Giacomo Domeniconi, Gianluca Moro, Andrea Pagliarani, Karin Pasini, and Roberto Pasolini. Job recommendation from semantic similarity of linkedin users' skills. In Maria De Marsico, Gabriella Sanniti di Baja, and Ana L. N. Fred, editors, *Proceedings of the 5th International Conference on Pattern Recognition Applications and Methods, ICPRAM 2016, Rome, Italy, February 24-26, 2016*, pages 270–277. SciTePress, 2016.

[12] Krishnaram Kenthapadi, Benjamin Le, and Ganesh Venkataraman. Personalized job recommendation system at linkedin: Practical challenges and lessons learned. In Paolo Cremonesi, Francesco Ricci, Shlomo Berkovsky, and Alexander Tuzhilin, editors, *Proceedings of the Eleventh ACM Conference on Recommender Systems, RecSys 2017, Como, Italy, August 27-31, 2017*, pages 346–347. ACM, 2017.

[13] Si Si, Huan Zhang, S. Sathiya Keerthi, Dhruv Mahajan, Inderjit S. Dhillon, and Cho-Jui Hsieh. Gradient boosted decision trees for high dimensional sparse output. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017,*

Sydney, NSW, Australia, 6-11 August 2017, volume 70 of *Proceedings of Machine Learning Research*, pages 3182–3190. PMLR, 2017.

[14] Dichao Hu. An introductory survey on attention mechanisms in NLP problems. *CoRR*, abs/1811.05544, 2018.

[15] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM neural networks for language modeling. In *INTERSPEECH 2012, 13th Annual Conference of the International Speech Communication Association, Portland, Oregon, USA, September 9-13, 2012*, pages 194–197. ISCA, 2012.

[16] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. In Dekai Wu, Marine Carpuat, Xavier Carreras, and Eva Maria Vecchi, editors, *Proceedings of SSST@EMNLP 2014, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar, 25 October 2014*, pages 103–111. Association for Computational Linguistics, 2014.

[17] Zichao Yang, Xiaodong He, Jianfeng Gao, Li Deng, and Alexander J. Smola. Stacked attention networks for image question answering. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 21–29. IEEE Computer Society, 2016.

[18] Xuezhe Ma and Eduard H. Hovy. End-to-end sequence labeling via bi-directional lstm-cnns-crf. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016.

[19] Matthew E. Peters, Mark Neumann, Luke Zettlemoyer, and Wen-tau Yih. Dissecting contextual word embeddings: Architecture and representation. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 1499–1509. Association for Computational Linguistics, 2018.

[20] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. Visualizing and understanding recurrent networks. *CoRR*, abs/1506.02078, 2015.

[21] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proc. of NAACL*, 2018.

[22] Min Joon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[23] Peng Zhou, Zhenyu Qi, Suncong Zheng, Jiaming Xu, Hongyun Bao, and Bo Xu. Text classification improved by integrating bidirectional LSTM with two-dimensional max pooling. In Nicoletta Calzolari, Yuji Matsumoto, and Rashmi Prasad, editors, *COLING 2016, 26th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers, December 11-16, 2016, Osaka, Japan*, pages 3485–3495. ACL, 2016.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[25] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[26] Yukun Zhu, Ryan Kiros, Richard S. Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 19–27. IEEE Computer Society, 2015.

[27] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[28] Alexei Baevski, Sergey Edunov, Yinhan Liu, Luke Zettlemoyer, and Michael Auli. Cloze-driven pretraining of self-attention networks. In

Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 5359–5368. Association for Computational Linguistics, 2019.

[29] Trieu H. Trinh and Quoc V. Le. A simple method for commonsense reasoning. *CoRR*, abs/1806.02847, 2018.

[30] Alexis Conneau and Guillaume Lample. Cross-lingual language model pretraining. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 7057–7067, 2019.

[31] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016.

[32] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 3980–3990. Association for Computational Linguistics, 2019.

[33] Amita Misra, Brian Ecker, and Marilyn A. Walker. Measuring the similarity of sentential arguments in dialogue. In *Proceedings of the SIGDIAL 2016 Conference, The 17th Annual Meeting of the Special Interest Group on Discourse and Dialogue, 13-15 September 2016, Los Angeles, CA, USA*, pages 276–287. The Association for Computer Linguistics, 2016.

[34] Alexis Conneau and Douwe Kiela. Senteval: An evaluation toolkit for universal sentence representations. In Nicoletta Calzolari, Khalid Choukri, Christopher Cieri, Thierry Declerck, Sara Goggi, Kôiti Hasida, Hitoshi Isahara, Bente Maegaard, Joseph Mariani, Hélène Mazo, Asunción Moreno, Jan Odijk, Stelios Piperidis, and Takenobu Tokunaga, editors, *Proceedings of the Eleventh International Conference on Language Resources and*

Evaluation, LREC 2018, Miyazaki, Japan, May 7-12, 2018. European Language Resources Association (ELRA), 2018.

[35] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 815–823. IEEE Computer Society, 2015.

[36] Stephen Merity. Single headed attention RNN: stop thinking with your head. *CoRR*, abs/1911.11423, 2019.

[37] Hakan Inan, Khashayar Khosravi, and Richard Socher. Tying word vectors and word classifiers: A loss framework for language modeling. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[38] Giacomo Domeniconi, Gianluca Moro, Andrea Pagliarani, and Roberto Pasolini. On deep learning in cross-domain sentiment classification. In Ana L. N. Fred and Joaquim Filipe, editors, *Proceedings of the 9th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - (Volume 1), Funchal, Madeira, Portugal, November 1-3, 2017*, pages 50–60. SciTePress, 2017.