


POLITECNICO DI TORINO

Master degree in Computer Engineering

Master Thesis

Deep Learning for Session Aware Conversational Agents

Supervisor
Prof. Maurizio Morisio

Candidate
Matteo Antonio Senese

LINKS Foundation Tutor
Dr. Giuseppe Rizzo

October 2019

Come alla fine di ogni viaggio, ci si guarda intorno per fissare in mente gli ultimi istanti di quello che sembrava un eterno oggi, un po' per realizzare dove si é arrivati, un po' per paura di perderli per sempre. Realizzare che l'essenza del viaggio non é la vetta di oggi, ma i sassi dove cadesti ieri, non la calma della fine, ma il trambusto della tempesta. Se oggi sono qui a scrivere queste righe è grazie alle persone che ho avuto il privilegio di avere dietro e di fianco a me. Voglio dedicare questo lavoro a tutti coloro che si sono spesi per insegnarmi anche una sola virgola di quello che avevano imparato, a tutti i miei professori, passati e futuri. Ai miei genitori che mi hanno permesso di essere chi volevo essere, che mi hanno insegnato l'onestà e il rispetto, ad apprezzare le cose vere, e a cui sarò eternamente in debito. Ai miei fratelli che mi hanno sostenuto anche solo con uno sguardo o con un gesto non richiesto. A mia nonna, i miei zii e i miei cugini che mi hanno cresciuto e continuano a farlo sempre. Agli amici di una vita e a quelli che ho scoperto in questi anni. Grazie per essere meglio di me e non farmelo pesare. Grazie per essere stati parte di un tratto importante e per continuare ad esserlo per quelli futuri. Al mio amico Davide, per tutte le volte che mi ha dato senza volere nulla in cambio, per tutte le cose che ha condiviso con me, per la sua genuinità e per tutte le risate. Grazie a Sara per essermi sempre stata vicina nonostante i momenti, nonostante i chilometri, per avermi supportato e ispirato, per tutti i baci e tutte le birre, e per tutte quelle che verranno. A Giuseppe, grazie al quale questo lavoro è stato possibile, per la fiducia concessami, per il saper essere stato una brillante guida e un grande amico (.. e per tutto il caffè).
Pronto per quello che verrà.

«To see the world, things dangerous to come to, to see behind walls, to draw closer, to find each other and to feel. That is the purpose of Life.»
The Secret Life of Walter Mitty.

- Matteo Antonio Senese, Ottobre 2019 -

Summary

Abstract

In the last 2 years the state of NLP research has made a huge step forward. Since the release of *ELMo* [1], a new race for the leading scoreboards of all the main linguistic tasks has begun. Several models came out every 2 months achieving promising results in all the major NLP applications, from *QA* to text classification, passing through *NER*. These great research discoveries coincide with an increasing trend for voice and textual technologies in the customer care market. One of the next biggest challenges in this scenario will be the handling of multi-turn conversations, a type of conversations that differs from single-turn by the presence of the concept of session. A session is a set of related QA between the user and the agent to fulfill a single user request. A conversational agent has to be aware about the session to effectively carry on the conversation and understand when the goal can be achieved.
The proposed work focused on three main parts: *i)* the study of the state of the art deep learning techniques for NLP *ii)* the presentation of a model, ***MTSI-BERT*** (Multi Turn Single Intent BERT), using one of such NLP milestones in a multi-turn conversation scenario *iii)* The study of a real case scenario.
The work takes in consideration both *Recurrent Neural Networks* and attention based models, as well as word embedding such as *Word2Vec* and *Glove*. The proposed model, based on *BERT* and *biLSTM*, achieves promising results in conversation intent classification, knowledge base action prediction and end of dialogue session detection, to determine the right moment to fulfill the user request. The study about the realization of *PuffBot*, an intelligent chatbot to support and monitor asthmatics children, shows how this type of technique could be an important piece in the development of future chatbots.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter acts as an introduction for the proposed work. The first part describes the cultural and folkloric vision of conversational AI in the current and past century. The second part presents the major milestones in the history of conversational AI of the last 70 years and an important philosophical premise for the following work, in the first half.
The third and last parts contain the description of the task proposed in this work.

1.1 Popular expectations in conversational AI

Since the creation of computational systems one of the biggest desire of humankind is to replicate human intelligence. Many novels and films imagine a future where humans are side by side with robots able to accomplish various tasks and to interact in a natural and fluent way with humans. *Isaac Asimov* was an American writer and professor of biochemistry at *Boston University*, he wrote several futuristic books about the coexistence of humans and artificial agents that replace computers with an increased efficiency and facilitation of interaction. *Asimov* described very accurately this futuristic society and has introduced, maybe for the first time, the possible dangers and ethical problems that a technology like that could bring to human race. In 1942, in a short story called *Runaround*, *Asimov* introduced the so called *Three Laws of Robotics* that are basically the fundamental rules that an intelligent agent created by humans has to always respect in order to preserve the human safety. The three rules are:

1. *A robot may not injure a human being or, through inaction, allow a human being to come to harm.*
2. *A robot must obey orders given it by human beings except where such orders would conflict with the First Law.*

3. *A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.*

These three rules inspire a lot of other novels and films. *Blade Runner* of 1982 by *Riddley Scott*, set in a dark and dusty *Los Angeles* of a dystopian future in the year 2019, tells about the story of a set of *replicants* that rebel against humans. The film introduces a series of ethical concerns about how thin is the separation between a real human intelligence and a bioengineered replicant enslaved to work all its life on a inhospitable planet at the edge of the galaxy. While different novels have gone so far thinking about bioengineering or complete robots that can move, see, talk, understand and think also in a more abstract way, other ones simply limit the scenario to the presence of an intelligent assistant able to see, talk and understand humans. *2001: A Space Odyssey* is a 1968 film by *Stanley Kubrick* which imagines a space mission having on board 5 humans and a sentient computer called *HAL 9000* (Heuristically Programmed ALgorithmic Computer). *HAL* can help the astronauts with the control of the spaceship and all the technical stuff related to the mission. It is capable of speech, understand and other many human like capabilities. It even has a configurable sarcasm level to allow empathy with the astronauts group. All the mentioned films and novels describe an intelligent agent able to compete with humans and accomplish several difficult tasks even in a more efficient way compared to a human being. This kind of agents are artificial general intelligent (AGI) systems which nowadays are strongly desired by many philanthropists and AI pioneers. Even though AI research achieved incredible results in the last 20 years, the so called AGI is still light years away from the current state of the art. The last decade was characterized by a rebirth of neural networks thanks to the available computational power needed to train them. This type of technology is inspired by human brains where different units fire together to make signals flow when a particular stimulus arrived. The field of *Machine Learning* is born and promises a great margin of applicability and research for new intelligent technologies. It has revolutionized all the digital fields that were stuck because of the presence of complicated rules that regulate the environment. *Machine Learning* aims to exploit the large amount of available digital data to overcome hand-coded rules and allow machines to autonomously understand the patterns that regulate a specific phenomenon. A great success of deep learning (a variant of machine learning exploiting deep neural networks) is for sure *Computer Vision* that is capable today of distinguish faces, animals, detect anomalies in cellular tissues as well as analyze real-time recording to discover obstacles in the way. All of this applications bring us now to test several prototypes of self-driving cars that promises to have a great impact on the habits of

tomorrow. The vision is one of the fundamental tasks to achieve the AGI and machines now prove they can distinguish several patterns in images to detect high level features starting from basic shape like line, circles, corners. A huge amount of animal species have the ability to see and elaborate the information inside their brain to understand what surrounds them. The gazelle needs to recognize the lion hidden in the grass to have a chance of survival. Anyway, while the vision is a type of ability shared among different intelligent life forms, only humans seem to have the power of communicate through a formal defined language, composed of thousand of different words, to express both thoughts and sentiments. Maybe the language is what gives us the competitive advantage over the thousand of species that used to hunt and kill us. The language is for sure what allow us to create societies based on the exchange of favors and lead by a group of persons that organizes through laws the right and duties to survive as a community. It is so important and so present in our lives that influences also our thoughts, which are often expressed by meaning of words. Unfortunately programming machines able to understand and produce human language it is a very difficult problem that still today is far from being solved. One of the great difficulties, for a machine, in understanding human languages derived from the existence of two levels of interpretations: the syntactic and semantic level. The syntax is the field of linguistic which studies the relations between the elements of an expression. The semantic considers instead the relations between the expression and the extra-linguistic world. While we are quite good today in analyzing sentences in a syntactic way, we can not say the same for the semantic. To correctly implement the semantic level into a machine we need first to define the concept of meaning. The field in charge of studying the language in order to make machines able to use it is called *Natural Language Processing* (*NLP*). *NLP* achieves great results in extracting syntactical information from sentences like existing relationships among words. In the last decade researchers has tried first attempts to extract semantic information from words by using machine learning techniques. The main idea is that words that often co-occurs are similar. Following this, the algorithm learns a N-dimensional vector representation for each of the word, keeping close in that space all the co-occurring words. Today we are capable of building technologies that understand human languages in a very limited context and provide limited interaction.

The following work is an attempt to list the current state of the art for NLP, to better understand which is the way the AI researcher are following to reach such expectations, and to propose a new model for multi-turn session identification.

1.2 History of conversational AI

1.2.1 '50 - Turing test and the born of philosophical AI

The first attempt to theorize the feasibility of such a project was made by *Alan Turing* in a 1950 seminal paper entitled *"Computing Machinery and Intelligence"*[13]. The paper main idea is trying to interpret and answer to the question *"Can machines think?"*. That sort of pretty philosophical question requires to define the act of *"think"* as a property in order to understand if a *"machine"* could have this type of property (properly given by its programmer). Such a task is not trivial and still today has not a formal solution. Due to this, *Turing* changed the question into *"Can machines do what we (as thinking entities) can do?"*, making easily to test without incurring in philosophical issues. As any scientific work, *Turing* also formulates a way to test this ability of a machines that quickly became known as *Turing test*. The test *Turing* proposed was the *"Imitation game"* that involves three players A, B and C. Player A is a machine while player B and C are humans. Player A and B are not in the same room of C and interact with him only through pairs of written notes. Both A and B try to convince C that they are humans. The computer wins the game if the judge cannot tell which is which. What *Turing* want to focus on is: by providing enough memory, enough computational speed, and an appropriate program, can the C machine imitate a human being in such a game? The paper proposed by *Turing* receives a lot of criticism among religious man who see intelligence as a god's gift, among philosopher which *Turing* responses with the *Problem of other minds* but also among mathematicians. The latter quoted the *Gödel's incompleteness theorem* to assert that there are some limits to what a computer system based on logic can answer. Another interesting objection was made approximately 100 years before by the mathematician *Ada Lovelace* which states that computers are incapable of originality because they can only do what we tell it. The last part of the paper is of crucial importance because *Turing* introduces for the first time the concept of learning machine. He said how difficult could be to build a machine that mimic human intelligence at once, and so propose to build a machine that has the basic ability of child and then simulate the human growth by making machine able to learn itself, through a mechanism of penalties and rewards, what is important and what is not. He proposes to mimic the natural selection for the learning phase, giving birth to genetic algorithms.

In 1980 the philosopher *John Searle* proposes a thought experiment known as *Chinese room* [14]. In this experiment he imagines that exists a

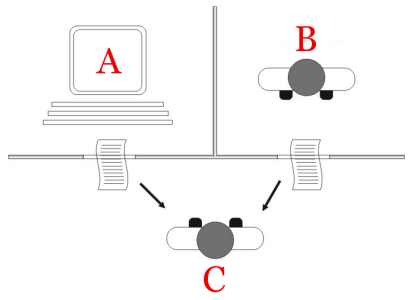

Figure 1.1: The Turing test. C has to understand the real nature of A and B interacting with them only through written notes

machine able to pass the *Turing test* in Chinese language. This machine has got a set of rules, do not matter how complicated, that allow it to pass the test against a Chinese speaker on the other side. *Searle* imagines to take the part of the machine and "enroll" its set of rules in order to execute it manually by itself. Each time *Searle* receives a bunch of Chinese characters it follows the instruction code of the machine and produces a Chinese response. With this experiment he proves how the *Turing test* can be passed even without truly understand, since neither *Searle* understands Chinese but he was able to communicate with the other side by simply using a set of hard coded rules. This experiment was a milestone for philosophical AI because it outlines the difference between simulate an intelligence versus be intelligent. The crucial element that a machines misses is the consciousness which is a concept for which the current philosophy still does not have found an explanation. The concept of consciousness is even better exposed in the 1978 thought experiment made by the philosopher *Ned J. Block* known as *China Brain*. The author imagines to take all the population of China and make each single person to act like a single neuron of human brain. According to functionalism this system should have a mind, but is quite intuitive to show how this system could create a mind capable of thoughts and feelings.

What is evident to anyone today is that there is a difference between what we called brain and what we call mind. When we talk about brain we usually refer to the biological organ able to react to input by means of complex signal flow. Mind is instead something more abstract that we still do not understand and it is the one responsible for thoughts, feelings and the part in which consciousness resides. In order to build intelligent machine as the one we have described, only when we will understand how mind works, we will be able to answer to the question *Can machines think?*.

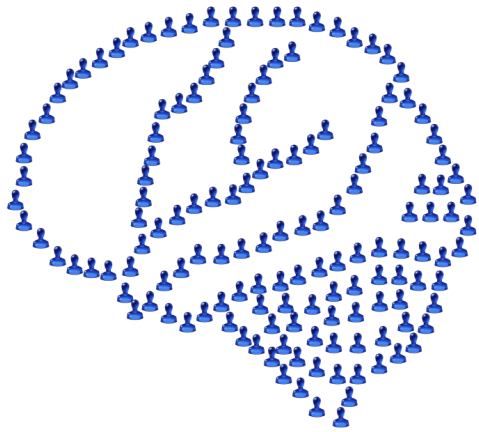

Figure 1.2: The China Brain. Can an entity composed by several different working units act like a mind?

For what belongs to us we just leave aside all the philosophical and ethical issues and we follow the opinion of *Peter Norvig* observing that most AI researchers "don't care about the strong AI hypothesis—as long as the program works, they don't care whether you call it a simulation of intelligence or real intelligence." [15]
What we are trying today is not to recreate human intelligence but simply to create useful systems that act intelligently.

1.2.2 '66 - ELIZA and first attempts to pass the test

In 1966 *Joseph Weizenbaum* created at *MIT Artificial Intelligence Laboratory* a program called *ELIZA*[16]. ELIZA whose name was inspired by the ingenue in *George Bernard Shaw's* Pygmalion, is a computer program that simulates conversations between humans. It is based on pattern matching and substitution methodology through a series of custom scripts which can be added to personalize the agent. Each script defines a set of keywords and associate to them a rank and a set of rules for transformation. The script searches for keywords inside the user input and put them in a queue sorted by keyword's rank, then the rule associated with the keyword having the highest rank is applied and the transformation is performed. For instance the person keyword such as "I", "You" could be transformed in "You", "I" in order to redirect the sentence to the user.The most famous script was the DOCTOR, a script that simulates a Rogerian psychotherapist by simply answered back to the user what they have just said.
Despite the very rude mechanism that empowers the chatbot, a lot of people

attributed to it the ability to think and understand what they told. A lot of experts said how this type of chatbot could help many people suffering from psychological disorder in daily life. In reality the real purpose of this experiment was to show the superficiality of conversations between humans and programmable machines. ELIZA was probably also the first chatbot that has ever attended (then failed) a *Turing test* and today is seen as one of the earliest realizations of Turing's idea.

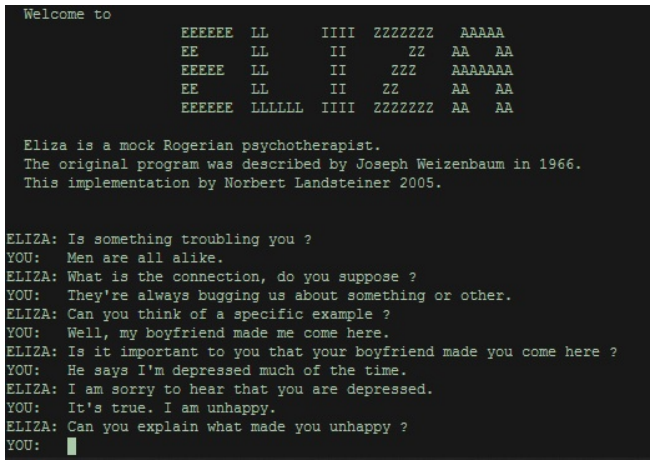

Figure 1.3: ELIZA just only create the illusion of understanding what user said, it does not have any contextual mechanism to be aware on current conversation.

In 1972 the psychiatrist *Kenneth Colby* implemented *PARRY* a chatbot that simulates a person affected by paranoid schizophrenia. It was subject to a variation of the Turing test where a group of psychiatrists have to distinguish which are transcripts of a real patients and which are made by *PARRY*. The psychiatrists were able to correctly identify the patient only 48% of the time. Even if the results were promising for the time, the test was limited to a fixed domain with non real time interaction (the transcript were already written by other psychiatrists). Researchers also connect *PARRY* and *ELIZA* together to make them talk to each other, an artificial patient with an artificial doctor.

1.2.3 '95 - Jabberwacky and A.L.I.C.E heuristic based chatbots

A.L.I.C.E (Artificial Linguistic Internet Computer Entity) is a chatbot implemented by *Richard Wallace* in 1995. Differently from ELIZA and

PARRY it exploits heuristic pattern matching rules that make it more robust. Even if it was not able to pass the Turing test, A.L.I.C.E. wons the *Loebner Prizer* three times (2000, 2001 and 2004), an annual competition that awards computer programs considered more human-like by the judges. It was developed in Java and it uses AIML (Artificial Intelligence Markup Language) as XML Schema, the code is now open sourced.
In 1997 Jaberwacky came out developed by *Rollo Carpenter*, a British AI scientist. It participates to several *Loebner Prize* by collecting a third and a second place. In October 2008 Jaberwacky evolves in *CleverBot*, a bot that have held more than 150 million conversations. *CleverBot* is constantly learning, it extracts important keywords from the user text and seeks for matching keyword in previous conversations, when the keyword matches then the bot replies how a human responded to that input when it was asked. It was judged to be 59.3% human in a 2011 formal Turing test.

1.2.4 2011 - IBM Watson and *Jeopardy!* champions defeat

Starting from 2005 the IBM launched the *DeepQA* project whose aim was to develop a machine that can compete with humans in *Jeopardy!*, an American quiz show where participants are presented with general clues in the form of answers and they have to respond in the form of questions. The name of the machine was choosen as the first IBM CEO *Thomas J. Watson* and in 2010 was ready to compete with human players. The first official game on television was made in February 2011 against the champions *Ken Jennings* and *Brad Rutter*. After 3 games *IBM Watson* won the contest with $1 million dollar prize then given to charity. The *IBM Watson* has a natural language interface and a rude *text-to-speech* system that enabled it to reply with a mechanical voice. It was written in almost 4 years by 25 IBM researchers with different contributions from universities among which the *University of Trento*. Its hardware is composed by a cluster of ninety IBM power 750 servers, each of which uses a 3.5 GHz POWER7 eight-core processor plus 16 terabytes of RAM. *Watson* has different encyclopedias in RAM to make them accessible faster than the disk. Each time it receives an input it extracts the main keywords and searches for related answers in memory running in parallel hundreds of language analysis algorithm exploiting the *Apache Hadoop* framework for distributed computations. The main innovation of *Watson* was not the introduction of new sophisticated algorithm but its possibility to run them very quick and simultaneously. *IBM* is trying today to move *Watson* from a pure AI experiment to a commercial product able to help in different domains like business, healthcare and education through specific functionality offered via API.

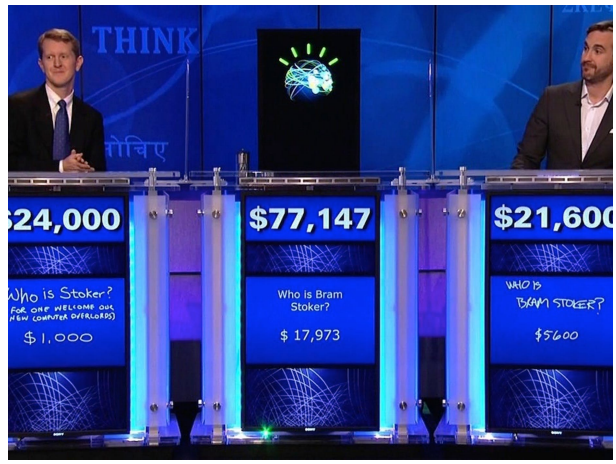

Figure 1.4: *Jeopardy!* game in February 2011. *Jennings* conscious of *Watson* victory wrote the emblematic phrase.

1.2.5 2014 - Siri, Google Assistant and the rise of voice based virtual assistants

In October 2011 *Apple* integrated *Siri* on the *Iphone 4S*, a voice virtual assistant able to answer general knowledge questions and to facilitate the interaction with the smartphone. When *Siri* receives a voice input it transforms it to text via a *speech-to-text* system developed by *Nuance*, analyzes the query and then performs the response action based on the intent detected in the query. *Siri* can answer simple questions by simply search on the web and it can handle some tasks via underlying OS like scheduling appointment or reminders. The presence on the market of a virtual assistant like that brings the major smartphone company to develop in house alternatives to remain competitive. One notorious effort was made by *Google* which introduces, in 2016, the assistant on its *Android OS*. *Google Assistant* provides a natural voice interface and a sophisticated *text-to-speech* system using *WaveNet* [17], a deep generative model able to mimic any human voice with incredible accuracy. *Google Assistant* is probably the most advanced virtual assistant since now also thanks to the knowledge base that it can exploit. Google has developed a knowledge graph which aims is to represent knowledge in a structured way by linking together related entities. Traversing this graph means to extract knowledge then used to shape the query's response. The Google graph has an enormous size covering 570 million entities and 18 billion facts and is continue expanding [18].
According to the report of *Canalys* the smart speaker shipments grew by 187% and home assistant devices are by now present in different homes.

Smart assistants exploit the IOT technologies to interface with different devices in the room like lamps, to make easier to turn them on by simply pronounce a little command for the assistant. Market analysts confirm how voice will be the predominant way for web search in 2020, this to underline the importance of developing intelligent technologies to overcome actual machine understanding limits.

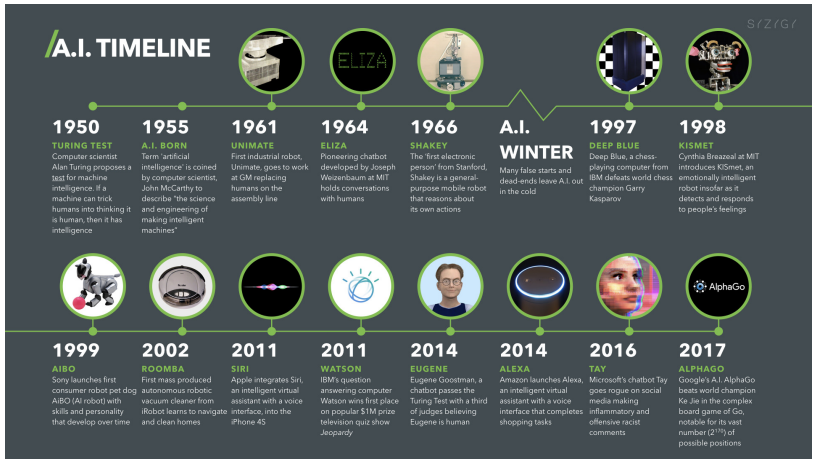

Figure 1.5: AI timeline taken from *SYZYGY* international survey *Public Perceptions of Artificial Intelligence: What Marketers Need to Know*

1.3 Natural Language Understanding and Human Computer Interaction

Natural Language Understanding (NLU) is a sub-task of NLP that deals with the comprehension of human written text. NLP is typical considered as a pre-processing for NLU, for instance Automatic-Speech-Recognition (ASR) can be used before NLU algorithms. NLU is considered to be an AI-hard problem for which solution have not yet been found. The task of comprehension is everything except trivial, it involves human-language comprehension together with reasoning processes to achieve a desired final action. During the computer science history, different attempts to create a NLU system have been made. A famous example is the *SHRDLU*[19], a system for English language understanding developed by *Terry Winograd* at MIT in 1971. *SHRDLU* is able to understand and reply to human written questions related to a small world it knows, composed by colored boxes an pyramid. The questions are related to the color, shape, size of objects

in this world for which *SHRDLU* was able to reason on having some predefined rules. A more recent study of 2012 [20] proposed a way to build a NLU system for robots, in order to make more natural interactions with them, for ease of use. The so called Human Computer Interaction (HCI) is the final aim for which NLU is studied for, it consists in make easier the interaction with machines using only natural language. While the above examples are all based on a limited knowledge world, consisting in a small environment, today challenges have a wider perspective. NLU is at the base of all the modern voice search technologies, it has to correctly retrieve the requested information from an enormous data source which is the web. An usual task is the semantic parsing, that is the technique to transform a natural language request to machine comprehensible code, like an SQL query. Having so much data introduces the needs, for the algorithms, to solve also other different problems related to words ambiguity. Other uses of today NLU interfaces are sentiment analysis or automated trading, both of crucial importance for the future business companies.
The next section defines the main task of this work which belongs to Natural Language Understanding field. Also different understanding tasks like intents classification and conversations are better explored.

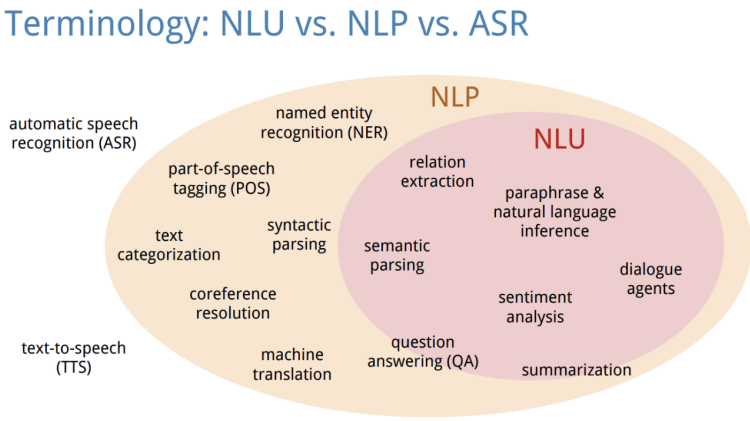

Figure 1.6: NLU is a subset of NLP. Source [3]

1.4 Task description

The majority of conversational agents are today based on *single-turn* interaction. This type of implementation allows the user to request something to

the agent by pronouncing a simple natural language command. The command has to be clear, in order to allow the agent to comprehend it correctly, and complete, meaning that it has to contain all the information, needed to the agent, to achieve the desired goal. A real case example is a user asking to a smart assistant for the nearest restaurant. The assistant will likely query *Google Maps*, retrieves a list of restaurants in the area, generates a syntactically valid response and then deliver it to the user. If the user tries to continue the interaction about one of the listed restaurants, the assistant will not be able to correlate the new interaction with the previous response. This happens because a single-turn agent simply relies on a single-utterance context, each time a new command is expressed the agent will likely forget what has been said before.

Today the rise of smart speakers on the market pushes AI companies to produce more natural paradigms of conversations, which can improve user experience and satisfaction. The Holy Grail of modern conversational AI is the multi-turn paradigm. Such paradigm relies on conversations having multiple pairs of related QA belonging to the same session, thus allowing more fluent and natural interactions between user and agent. A session is a set of utterances between the two parties which are related to a single topic or intent. Thinking about the previous example, would be better if, after the response with the list of restaurants, the agent would answer with "Do you need more information?" and then carries on the conversation to provide a final answer as satisfying as possible. Only at the end of conversation the agent will have all the information to proceed with the final goal, which could be, for instance, the restaurant reservation. Another possible scenario is the possibility to carry on tasks having missing information at the beginning. When the user asks for a web search, today agents need to know also, since the first interaction, the content to search on the Web. A multi-turn conversational agent should be able in practice to ask the user for missing information needed to complete the task, without surrender immediately with an "I don't understand". In such a way more human like interaction are possible with the result in facilitation of usage which could lead to a greater pervasiveness of this technology.

This paradigm is still today pretty far to be solved because the difficulty for the assistant to keep the context and correctly references it whenever is needed. A work in this way is contained in a paper by *Yun-Nung Chen et al.* [21] which propose and end-to-end memory network with knowledge carryover for multi-turn understanding. Even if the promising results, the cited work seems to be not aware of the presence of session, thus not implement any kind of context flush when the session expires.

The proposed work consists in the classification of three main components

which could be useful in a multi-turn scenario: intent, knowledge base action and end-of-session. The proposed method is described in chapter 3 while the results are presented in chapter 5. In chapter 6 an interesting use case is shown and final conclusions are reported in chapter 7. The three aspects are instead better described in the following subsections.

1.4.1 Intent classification

Today voice agents are always intent based. An intent represents the will of the user, expressed in natural language. Typically a conversational agent is domain specific and can handle only a certain number of different intents. Some common examples of intents are "weather", "booking", "scheduling" or something else like "write email". A multi-turn session can be identified by a single or a multiple number of intents. While a real natural interaction could be achieved with a multi-intent session, this paradigm seems today to be still too complex to be handled and for this reason is out of the scope of this work. For what concerns this work, only single intent session are studied.
The intent is the feature that triggers the session and identify it from the beginning to the end. Such classification should be done with the first user interaction and will influences all the intra-session utterances. The type of detected intent will bring the agent to satisfy a different goal. More pragmatically, an intent can be seen as a different API to be called. A today agent has a different set of APIs which identify all the different tasks it can handle. Each API is invoked when the correspondent intent is identified in the user request. When the user asks for the weather, the Natural Language Understanding interface will detect the "weather" intent and will deliver it to the interpreter, which is the module responsible for "interpret" the NLU interface and transform the request from natural language to machine language. Finally the service manager calls the correspondent API that returns a result that has then to be formulated as natural language (and maybe passed to a Text-To-Speech module).
The intent is of crucial importance, if an agent misses it, the user experience will be very poor.

1.4.2 POS Tagging and Name Entity Recognition

If the intent can be seen as the "conceptual API", what misses now from the picture are the parameters to sent with this API. POS (Part of Speech) Tagging is the task of giving a grammatical tag to each word in a sentence. This technique is widely used to analyze better the meaning of the sentence and to perform words disambiguation (distinguish "book" as a verb and as a

noun). The next step after POS Tagging is the so called NER (Name Entity Recognition). NER is the technique to associate each word or set of words to a concept in the conversational domain. The NER module is of fundamental importance for an agent since it allows it to understand which entities the user is referring to. Retrieving the example of weather intent, an important parameter to fulfill the request is the city to search for the weather. Entities are part of the speech that commonly refers to existing concepts in the conversational domain world, such as cities, football teams, days of the week, months. The entities are the parameters the API receives. The final API to fulfill the "weather" intent will then have the following shape:

$$fetch\_weather(city = "New\ York",\ day = "Tomorrow")$$

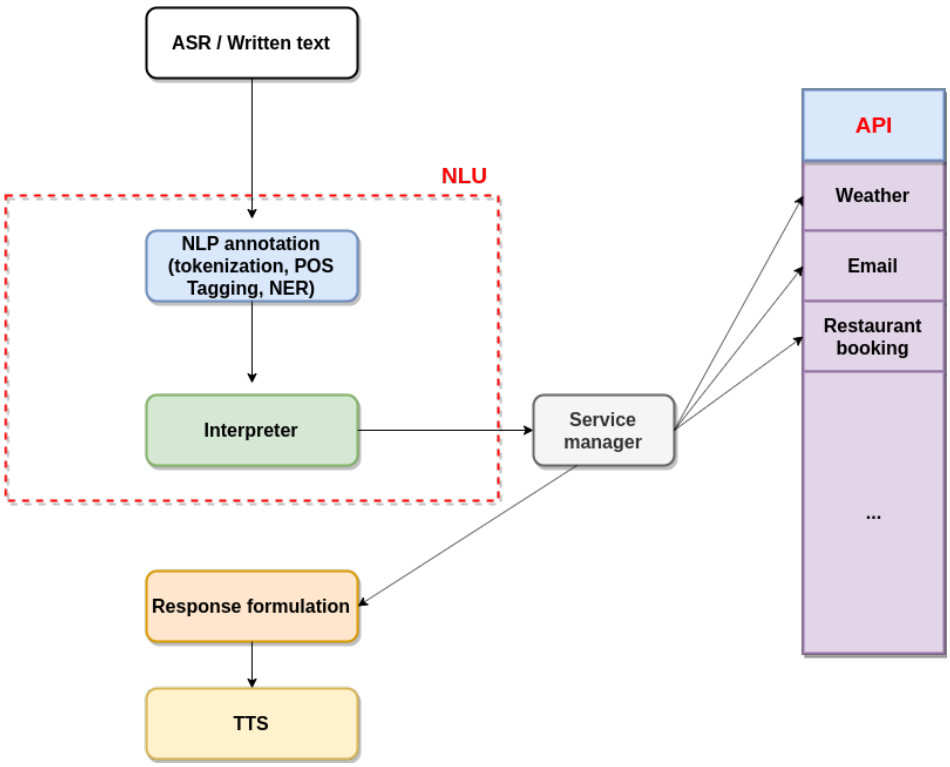

Figure 1.7: Common structure of today smart speakers. The NLU module is the one responsible for transforming the intent from natural language to machine code. The diagram was inspired by the slides of cs224U-2019, Stanford

1.4.3 Knowledge base action prediction

A next generation conversational agent should be developed together with a knowledge base, a structure containing all the needed information to carry on conversations and achieve tasks. Knowledge graphs are ones of the most used knowledge bases in today smart assistants. For instance *Google Assistant* exploits the Google knowledge graph, containing more than 500 million different entities and facts extracted from Wikipedia. A knowledge graph is a structure for linking related concepts, in such a way reasoning technique can be applied by traversing the graph. If the user asks for the city where George Washington was born, the Google graph not only replies with the name of the city, but also with population and location, since the concept of city is related to George Washington but also with other different concepts (state, population, foundation year etc.). While the previous example is about a world knowledge, a real knowledge base has to contain different levels of knowledge. The domain knowledge represents all the information needed for the agent to understand and carry on information about the particular domain it was created to work on. *PuffBot*, for instance, contains domain knowledge about different pains and emotional status related to the asthma, these information were given by domain experts of Trento and are useful to deploy *PuffBot* in such scenario. While world and domain knowledge are typically "hardcoded" inside a conversational agent, a today research hot-topic is to find a way to make neural networks able to exploit such ground knowledge.
The last level of knowledge an assistant needs in a multi-turn paradigm is the discourse knowledge. This knowledge is of extreme importance and allows the agent to better understand what is the user desired goal to achieve. A discourse knowledge should contain all the information extracted from the current session, in this way the agent can understand the current status of conversation and predict the next step. Thinking about the restaurant reservation, the agent can store all the information given by the user inside a suitable data structure and understand when all the needed information to ask for the reservation are already present.
In this work a step towards these knowledge bases is done. Two different actions were distinguished: fetch and insert. A fetch is the action of "reading" information from the knowledge base and is needed whenever the user requests something that it does not know like "What's the weather like in NYC?" or "Remind me of tomorrow appointment". An insert instead is an action to be performed when the user gives a new item or an additional information, for instance when he want to schedule a new event. In such scenario the agent has to insert the new scheduling in the knowledge base to be able to fetch it in future questions.

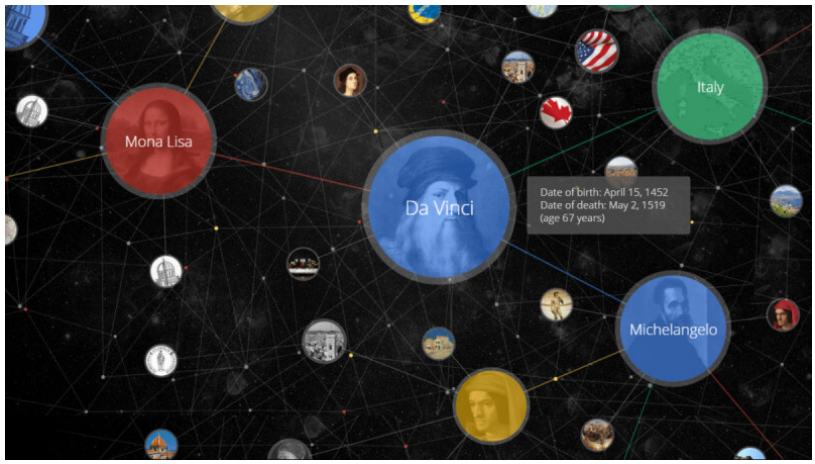

Figure 1.8: The Google knowledge graph links together related entities to form a net of knowledge.

1.4.4 End of session detection

Another issue of multi-turn conversations is to understand the moment in which a session expires and a new one starts. When this happens the assistant has to discard the context of the previous session and start a new one. For instance, if after the restaurant reservation the user needs to know about the weather for that night, the assistant has to understand that this question is not related with the previous restaurant session. The agent has so to be aware of the existence of a conversation session. In the case of the proposed *PuffBot*, the agent must know when the current diagnostic session is finished, in order to make a little resume of the patient conditions. The resume will use all the information detected inside the session by fetching the knowledge base.
Multiple solutions could be adopted. A timer can be set to wait for a new user request, if the request arrives after the timer expires then the agent can discard the previous context. Another possibility is the one proposed by *Mensio et al.* [22], they predict the sequence of intents in a multi-turn QA, in this way is possible to understand when a new session starts by looking the change of intent. This last method is however weak against consecutive session with the same intent. The proposed work starts from this consideration to try to predict the end of session. Instead of focusing on intents succession, the base is to try to catch discontinuities inside a QAQ triplet. Each time a new user utterance is available, the model uses the BERT self-attention layers between the previous command and the current one. If discontinuities are observed, then the end of session is predicted and

the context is flushed.

Chapter 2

Deep Learning for NLP

This chapter presents all the major NLP deep learning architectures and techniques widely used in the last 20 years. The first part contains a description of the *word embedding* technique empowering the first "era" of transfer learning in NLP. In the second section the main focus will be on *Recurrent Neural Networks*, an architecture that is still today crucial for the development of the field, along with the two main cell variants. The third part presents the concept of *attention* and motivate its cumbersome presence in the today NLP revolution. Finally the last section shows the new NLP great discovery, the *transfer learning*, and it will lists some of the today most used models.

2.1 Word embedding

The technique of word embedding has its root back in 1960. The underlying idea is that the meaning of a word is related with the words that often co-occurs with it.

"a word is characterized by the company it keeps" (J.R. Firth)

This technique aims to represent the semantic level of words by means of N-dimensional vectors. These vectors contain a statistical representation of their features, related to all the other words in the considered vocabulary. Words that often co-occur together will have close vector representations by means of euclidean, cosine ore other distance measures. The used of these embeddings has empowered NLP models of the last 20 years, making faster the training and more accurate predictions. This first level of transfer learning is today often surpassed by modern contextual word embedding models. The main problem of word embedding is the presence of a

single representation for each word, without caring about the current context. Such problem is evident dealing with polysemy (words with different meanings) like "book", this word will have the same fixed representation both in sentences where it is a noun and sentences where it is a verb.
In literature exists two types of word embedding:

- Count based
- Predictive methods

The count based methods applies directly statistic exploiting the count of co-occurences, typically followed by a dimensionality reduction technique such as SVD. The predictive methods instead use machine learning models and train them to produce the embedding for each words. While the count based are considerably faster to train, the predictive has proved to be able to capture more complex patterns beyond word similarity, which leads to improved performance on different tasks.
In the following sections two algorithm are shown: *i)Word2Vec*, a predictive method *ii) Glove*, a hybrid approach.

2.1.1 Word2Vec

Word2Vec is a set of machine learning models whose aim to produce a vector representation for words. It was proposed by a group of researchers lead by *Thomas Mikolov*[23] at Google in 2013. The technique is based on *Neural Probabilistic Language Model*, a task in which a machine learning model is trained to predict the next word w_t given the history $h = \{w_1, w_2, ..., w_{t-1}\}$. Such a model learns the conditional probability of the next word, given all the previous ones.

$$P(w_t|h) = softmax(score(w_t, h)) = \frac{exp(score(w_t, h))}{\sum_{w' in\ vocab} exp(score(w', h))} \quad (2.1)$$

The $score(w_t, h)$ computes the compatibility of target word and the context h, dot product is commonly used. The model is trained to maximize the log-likelihood of such conditional probability.

$$J = log\ P(w_t|h) = score(w_t, h) = score(w_t, h) - log \sum_{w' in\ vocab} exp(score(w', h)) \quad (2.2)$$

Since the model is trained on an entire corpus, the final goal is to find the right parameters θ that maximizes the average of (2.2).

$$\arg\max_{\theta} \frac{1}{T} \sum_{t=1}^{T} \sum_{j \epsilon c, j \neq 0} log\ P(w_{t+j}|w_t; \theta) \quad (2.3)$$

The equation (2.3) expresses in a compact form the necessity to maximize the probability of seeing a particular word within a window having size c of the current word w_t.
The algorithm came up in 2 different flavours:

- Skip-Gram
- CBOW (Continuous bag of words)

While both variants work with a size c window, the objective function is different. The CBOW is trained to predict the center word of the window, given the surrounding context (the so called "bag of words"). The Skip-Gram is instead trained to predict the context, given the center word. CBOW relies on the "Bag of words" assumption, where the order of context words does not influence the prediction of the center word. Skip-Gram instead has a mechanism to weight nearby words more than distant ones. While CBOW is pretty much faster, the Skip-Gram handles better infrequent words.
In this section only the Skip-Gram algorithm is taken into consideration.

Skip-Gram

The Skip-Gram algorithm defines the conditional probability through the softmax function. In equation (2.4) w_i represents a context word (outside word) and w_t the target one (also referred as central word), both *one-hot encoded*. N and k are respectively the number of words in the vocabulary and the embedding dimension. θ will represent the trained lookup matrix after the learning phase.

$$P(w_i|w_t,\theta) = \frac{exp(\theta w_i)}{\sum_t exp(\theta w_t)} \quad ,\theta \epsilon \mathbb{R}^{NxK} \, , w_i \epsilon \mathbb{R}^N \tag{2.4}$$

The architecture for the Skip-Gram is a shallow 3-layer neural network. It has one input layer, one hidden layer and one output layer. The hidden layer presents a lookup matrix for the center word, from there the embedding for the input word can be retrieved by means of a simple matrix multiplication. Because w_t is one-hot encoded, the result of this operation will be the $i-th$ column of the embedding matrix, where i is the index of the entry of w_t containing the one. The output layer contains the lookup matrix for all the context words. The embeddings for the context and center word are dot product, the results are then followed by a softmax and an argmax, in order to predict each context words. The backpropagation is then performed in order to correct the network weights, which are the lookup matrices for center and context words. Optimizing such structure means to tune the

embeddings for words that often co-occur, trying to make them as similar as possible, in this way the dot product will have a higher result.
After the training the embedding for each word can be obtained by simply removing the output layer and feeding a valid index for the vocabulary word. The reader could have notice that during training, Skip-Gram, tunes

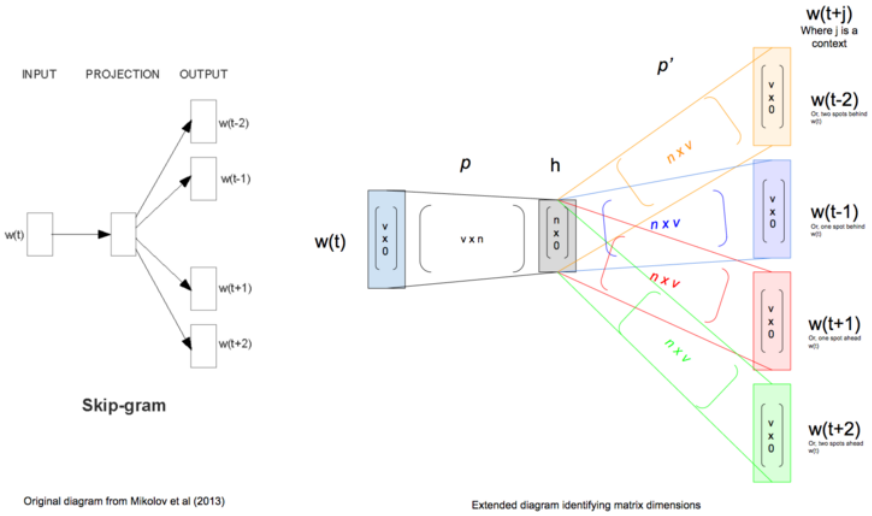

Figure 2.1: Skip-Gram architecture in an original diagram from *Mikolov et al.*, 2013

two different lookup matrices, even if at the end only one will be used. The reason why the proposed model does not have a unique matrix is merely mathematical. If the model had one single lookup matrix, then each time dot product is performed, the highest score will be assigned to the center word, considering that it is equal to itself. This is a typical problem of language models that must be avoided, a model that predicts the input word as the next word of itself has no predictive power.
During the development of Word2Vec there were two main problems to solve. The first problem was related to the presence of a very large corpus, using *batch Gradient Descent* the weights would be updated every epoch, and so too seldom. To solve this problem, Word2Vec uses *Stochastic Gradient Descent* that, differently from batch Gradient Descent, updates weights at each sample. The error is typically noisier but the model converges faster and is robust against local minimums. The second problem was instead more tricky to solve, it is related to the computation of denominator

in the conditional probability. Since words of vocabulary are a huge number, iterate over them each time is not feasible. Additionally it would be useless to compute the gradient for all the words in the vocabulary if we only have fixed size windows, the vector of gradients will be full of zeros. To overcome this problem, Word2Vec uses *negative sampling*. Instead of iterating over the entire vocabulary each time, just going to train a binary logistic regression for a true pair (center word and a word in its window) against several noise pairs (center word paired with random word), which is the so called negative sample.

Word2Vec embeddings are still used today because the built multi dimensional space, in which those vectors live, owns very interesting properties. Such space allows the use of mathematical and logical operators on word vectors. A now multi-cited example is the one involving the following operation:

$$e_{king} - e_{man} + e_{woman} \approx e_{queen}$$

Logical operations can also be performed:

$$e_{Rome} : e_{Italy} = e_x : e_{France}, \quad x \approx Paris$$

Such dreamy results are possible because of the rich geometry on which that space was built. Related words are close to each other and this brings a lot of benefits to the final representation. Even if Word2Vec can capture

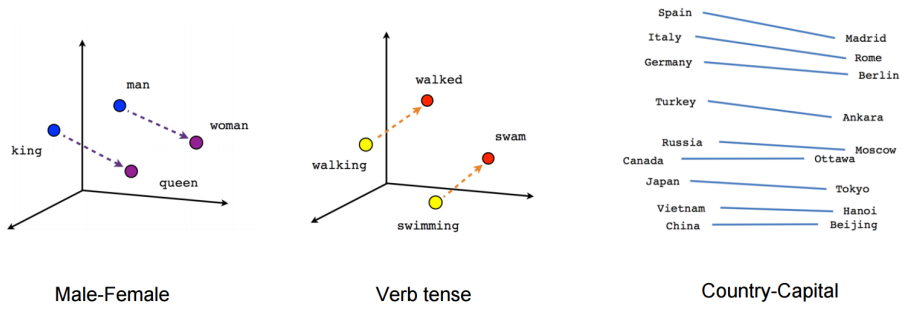

Figure 2.2: Some example tho show the rich geometry of Word2Vec space after a PCA, for dimensionality reduction.

very complex patterns in data, it not scales with corpus size and has a very slow training time. For this reason *Glove* was proposed.

2.1.2 Glove

Glove is a word embedding technique published by *Pennington et al.*[24] at *Stanford* in 2014. The idea of Glove is to take the pros from both count based and predictive models and put them together into a single algorithm. The crucial concept underlying this technique is that the ratio of co-occurrence probabilities can encode the meaning of words better than the raw co-occurrence probability matrix, the one typically built in a count based algorithm. The ratio probability of co-occurrence is defined in equation (2.5) where X_{ij} represents the number of times word j occurs in the context of word i and k is instead a probe word.

$$P(j|i) = \frac{X_{ij}}{X_i} = \frac{X_{ij}}{\sum_k X_{ik}} \tag{2.5}$$

To capture ratios of co-occurrence probabilities the equation (2.6) has to be satisfied, which means that the dot product of the two embedding must be higher as possible in order to maximize the log likelihood.

$$w_i \cdot w_j = logP(i|j) \tag{2.6}$$

The loss function for Glove was identified in the formula (2.7).

$$J = \sum_{i,j=1}^{V} F(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - logX_{ij})^2 \tag{2.7}$$

Glove is often preferred to Word2Vec especially when a little retraining is needed. The algorithm is much faster to train, it is scalable to huge corpora and it can achieve very good performance even with little amount of data and small vectors.

2.2 Recurrent Neural Networks

Recurrent Neural Networks also identified with the acronym *RNNs* are a family of neural network architectures designed specifically for time sequence analysis, based on the work of *David Rumelhart* in 1986. The main difference with a simple FFNN (Feed-Forward Neural Network) is the presence of a recurrent link which allows sharing of information between consecutive inputs belonging to the same sequence. Each hidden unit of RNN produces a hidden state which is passed to itself in the future timestep (and to the next layer in case of multi-layer RNN). A RNN receives the current

input and the previous hidden state, which contains information about the previous processed input. Each unit has two weight matrices, one for the input and one for the hidden state, which are shared for any element of the sequence. The core idea is that, to correctly analyze time sequences, the prediction on input x_t should depend on what was seen previously. Thanks to this sort of "statistical memory", *RNN* are used in different domains involving time sequence analysis, such as speech recognition, natural language understanding, natural language generation, bio-informatic and several others. One of the main task for which RNN are used for is language modeling. Language modeling is the task of predicting the next word w_{t+1} given the set of previous ones $w_1, w_2, \ldots, w_t$, it was proved how RNN are quite effective there with the respect to other techniques such *n-grams*. The recurrent weights matrix will learn the peculiarity of the language, like the syntactic and semantic level, the input weights instead will learn how to correctly handle the current word. Another great peculiarity of RNNs

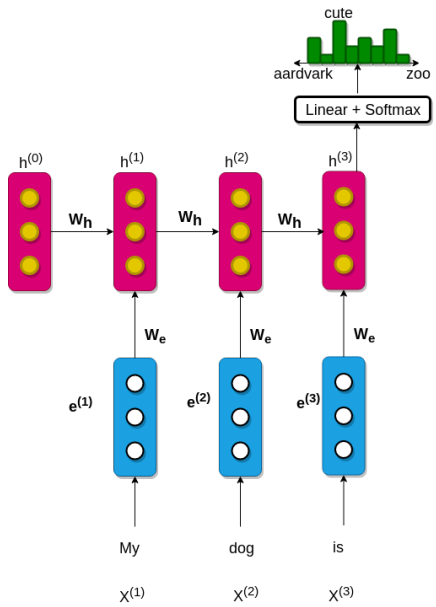

Figure 2.3: RNN are widely used for language modeling.

are their being general purpose encoders. To encode a sentence of length N inside a fixed length vector we can feed the entire sentence once and then use the last hidden state of the network as sentence digest.
RNNs allow having variable length input and output. Thanks to this structural property they can be used in different configurations such the ones listed in figure (2.4) Despite all this great things a RNN can do, they have

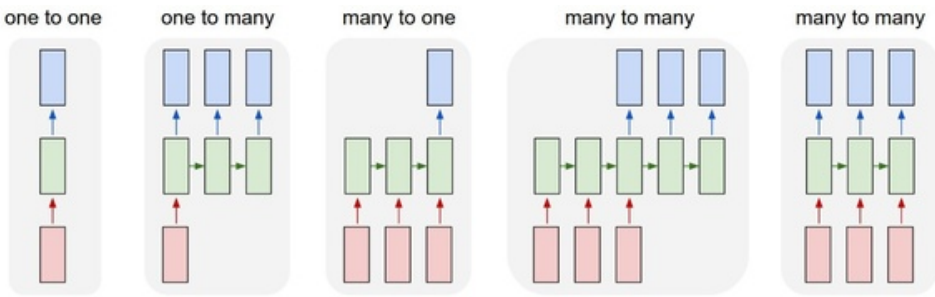

Figure 2.4: RNN are very versatile, they can have variable length input and output, then allowing different architectures.

some major limitations that have lead today to prefer attention based models for NLP tasks.

- Since each hidden unit needs the output of the previous one, they cannot be parallelized. This lead to long training time.
- They suffer from both exploding and vanishing gradient

While for the first issue does not seem to exist a solution, the second can be in part mitigated. Exploding gradient can be avoid by using *gradient clipping*, which consists in cutting the gradient above a certain defined threshold. Vanishing gradient is instead not trivial to solve and represents the main problem of such architecture.
The following section describes deeper the internal structure of a RNN vanilla unit. Other two section are then dedicated to the presentation of the two main cell variants designed trying to overcome the limits of the vanilla: *LSTM* and *GRU*.

2.2.1 Vanilla RNN

The vanilla RNN hidden unit consists of a layer receiving two vectors, the input X_t (at timestep t) and the hidden-state H_{t-1} produced by the unit itself at the previous timestep $(t-1)$. The hidden-state is a vector that should represents the history of all the input elaborated by this unit during time. A unit can be potentially composed by many neurons, like FFNN, allowing the layer to produce a more complex output. A single unit in the vanilla RNN has two different weights matrices, one to process hidden-state and another to process input. The novelty with respect to FFNN is the hidden-state matrix. This matrix is the one responsible for information to flow across timestep, for this reason is also called *recurrent weights matrix*. The vanilla RNN cell uses a tanh non linearity to produce the hidden state. Each unit can also produce an output by feeding the new hidden-state into

an additional linear layer. When working with NLP, the real input for the

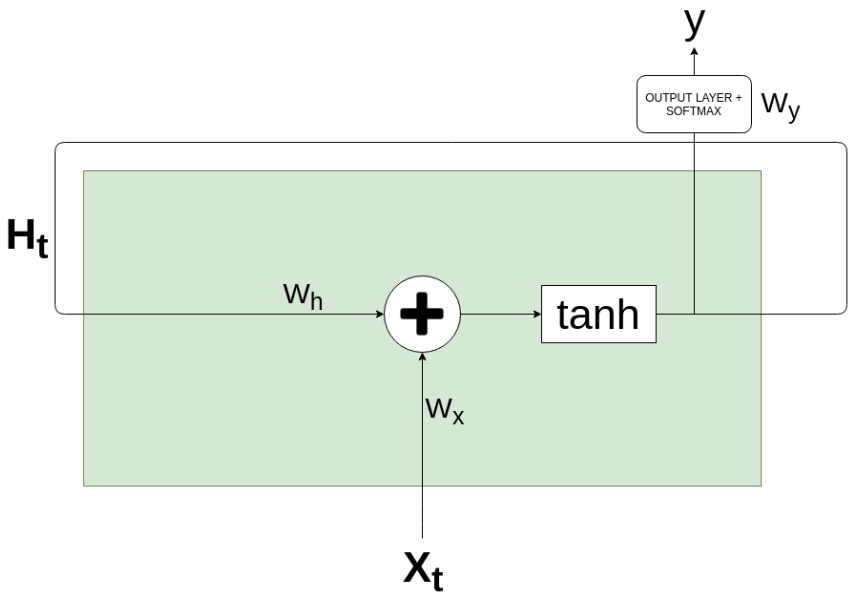

Figure 2.5: vanilla RNN cell structure. Each cell can have an output, thus allowing multiple output.

RNN cell of the first layer is typically the embedding of the word. More formally the equations that control the behaviour of a RNN are expressed in (2.8). X is one-hot encoded, E represents the lookup matrix, W_h and W_e are respectively the recurrent and input (embedding) weights.

$$\begin{aligned} h^{(t)} &= \tanh(W_h \cdot h^{(t-1)} + W_e \cdot e^{(t)} + b_1) \\ e^{(t)} &= E \cdot x^{(t)} \ , \ x^{(t)} \epsilon \mathbb{R} \end{aligned} \tag{2.8}$$

As other Neural Networks architectures, also the RNN were influenced by deep learning at the beginning of the *XXI* century. In a 2013 paper, *Razvan Pascanu* illustrates different ways to make deep RNNs, anyway this work will always refer to the ones having multiple hidden-layers between input and output. A deep RNN can be created by stacking together different recurrent units, thus each unit receives the hidden-state from the previous layer and the hidden-state produced by itself at previous timestep. Anyway, due to the vanishing problem that will be described later, the number of layer hardly exceeds 3 or 4. If the number of layers is greater than 8 it is a good rule to insert residual connection (also called highways) between hidden-layer to make gradient flows better. Another interesting aspect of

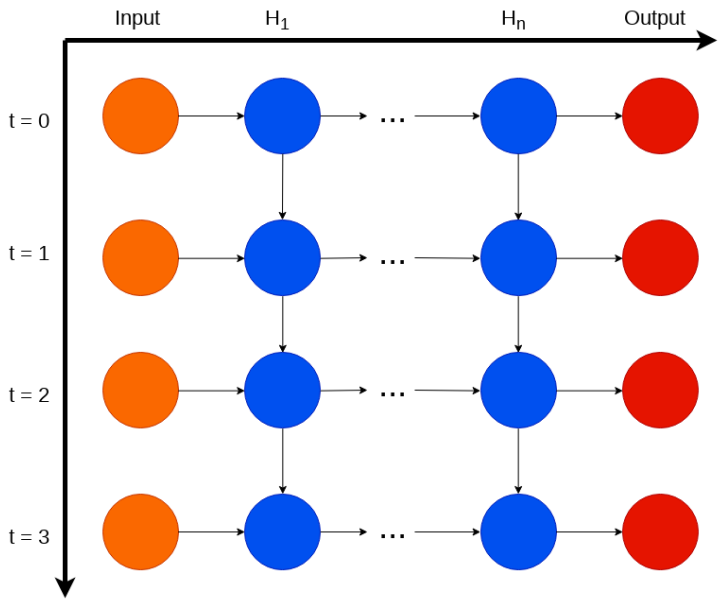

Figure 2.6: Deep Recurrent Neural Networks unfolded in time

this family of architectures is the fact that RNNs can approximate any algorithms. The work proposed by *Siegelmann* and *Sondag*[25] proves that it is possible to use RNNs to simulate a pushdown automaton with two stacks, given that every *Turing machine* can be simulated by such type of automaton and, for definition, a *Turing machine* can compute every computational function, then a RNN can compute every computable function.

BPTT

To train a Recurrent Neural Network a particular type of backpropagation algorithm is used, the so called Back Propagation Through Time (BPTT). The basic idea is to unfold RNN in time to make the gradient flows. Such algorithm is similar to the traditional backpropagation by taking in consideration two aspects:

- The loss can be computed at each timestep (because we have one output for timestep)
- Information travel through time in RNN thus the output at timestep t depends on the information elaborated in $t-1$, $t-2$ etc...

The network is fed by an instance of itself of the past sharing the same parameters. Figure (2.7) shows effectively the dependency of the current prediction on all the previous timesteps. The final loss value is the average

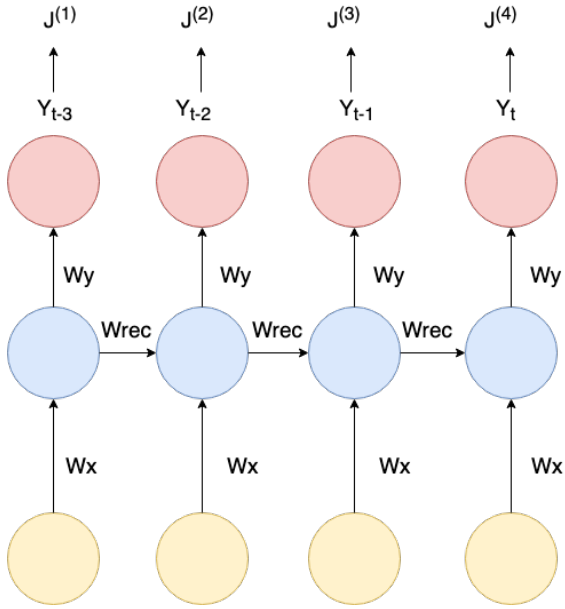

Figure 2.7: A single layer RNN unfolded in time.

of all the output losses.

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta) \tag{2.9}$$

By defining a learning rate η the standard equation for weights update, using gradient descent, ca be written.

$$W_i = W_i - \eta * \frac{\partial J}{\partial W_i} \tag{2.10}$$

Limits of the vanilla architecture

Vanilla RNNs present two main issues related to their structure: exploding and vanishing gradient. A paper by *Pascanu* of 2013[26], has shown possible problem during BPTT that could bring to computational errors or impossibility to learn patterns in long sequences. An intuition of such problem could be found in (2.11), supposing a backpropagation over $t = 4$ timesteps, the value of the current loss with the respect to distant hidden-states could

vanish, as each single term is very small.

$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial J^{(4)}}{\partial h^{(4)}} \cdot \frac{\partial J^{(4)}}{\partial h^{(3)}} \cdot \frac{\partial J^{(4)}}{\partial h^{(2)}} \cdot \frac{\partial J^{(4)}}{\partial h^{(1)}} \tag{2.11}$$

A formal definition of the problem is shown in equation (2.14) by putting together (2.12) and (2.13). Equation (2.12) is computed applying the chain rule for partial derivatives. Equation (2.14) shows the derivative of the loss computed by two timestep i and j, with the constraint $i \geq j$. The core of the problem is in the presence of the term $W_h^{(i-j)}$, representing the recurrent weights matrix raised to the distance between timestep i and j. If W_h is small, then vanishing gradient issue appears as the distance between the two timestep increases. If W_h is big, then we could have exploding gradient.

$$\frac{\partial h^{(t)}}{\partial h^{(t-1)}} = diag(\tanh'(W_h h^{(t-1)} + W_x x^{(t)} + b_1))W_h \tag{2.12}$$

$$\frac{\partial J^{(i)}}{\partial h^{(j)}} = \frac{\partial J^{(i)}}{\partial h^{(i)}} \prod_{j<t\leq i} \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \tag{2.13}$$

$$\frac{\partial J^{(i)}}{\partial h^{(j)}} = \frac{\partial J^{(i)}}{\partial h^{(i)}} \cdot W_h^{(i-j)} \cdot \prod_{j<t\leq i} diag(\tanh'(W_h h^{(t-1)} + W_x x^{(t)} + b_1)) \tag{2.14}$$

Exploding gradient can cause the raise of error in computations, for this reason is good habit to always perform gradient clipping when working with RNNs. Vanishing gradient instead prevents the model to learn long-term dependencies in the input sequence. This is evident since the loss at timestep t depends too little from far input. A real case issue with RNN is the difficult to handle syntactic recency versus sequential recency. An example is the sentence "The writer of these stories _" where the underscore indicate something missing that has to be predicted. Syntactic recency means to relate the prediction with the singular noun "writer" by predicting "is". Sequential recency means to relate the prediction with the plural noun "stories" by predicting "are". To properly handle syntactic recency vanilla RNNs are not the most suitable choice.
In order to allow RNN to handle longer sequences, many cells variant were proposed. The next sections will focus on the two most famous, LSTM and GRU.

2.2.2 LSTM

Long Short-Term Memory (LSTM) is a type of recurrent cell proposed firstly by *Sepp Hochreiter* in 1997 [27] to overcome the short-term memory of the vanilla RNN. A LSTM unit, differently from the vanilla RNN, has two different memory mechanisms: the hidden-state and the cell-state. The hidden state is responsible for storing short-term information, while the cell-state long-term ones. Both vectors have the same dimension. The crucial aspect of LSTM cell is the ability to read/write/erase information contained in the cell-state. The three operations are done using three different gates, each of them part of the training phase. At each timestep each element of the gates can be open (1), closed (0) or in an intermediary state between the two. The gates computation is dynamic, it relies on the value of the current hidden-state.
Three gates are present in the LSTM cell: forget, input and output gate. The forget gate is responsible of what is kept and what is forgotten from the previous cell-state $c^{(t-1)}$. The input gate instead decides which parts of the new cell will be written into the new cell $c^{(t)}$. Finally the output gate controls which parts of the cell will be part of the new hidden-state $h_{(t)}$. The equations that control the behaviour of the LSTM gates are reported in (2.15), where $f(t)$, $i(t)$ and $o(t)$ are respectively forget, input and output gates.

$$\begin{aligned} f(t) &= \sigma(W_{fh}h^{(t-1)} + W_{fx} \cdot x^{(t)} + b_f) \\ i(t) &= \sigma(W_{ih}h^{(t-1)} + W_{ix} \cdot x^{(t)} + b_i) \\ o(t) &= \sigma(W_{oh}h^{(t-1)} + W_{ox} \cdot x^{(t)} + b_o) \end{aligned} \tag{2.15}$$

The results of such gates are then combined to produce the output of the cell. The equation controlling the produced output are the ones in (2.16), there $\tilde{c}^{(t)}$ is the new content to be written in the cell-state.

$$\begin{aligned} \tilde{c}(t) &= \tanh(W_{ch}h^{(t-1)} + W_{cx} \cdot x^{(t)} + b_c) \\ c(t) &= f^{(t)} \cdot c^{(t-1)} + i^{(t)} \cdot \tilde{c}(t) \\ h(t) &= o^{(t)} \cdot \tanh c^{(t)} \end{aligned} \tag{2.16}$$

Finally figure (2.8) shows the entire structure of a LSTM cell. LSTM can handle longer sequences respect to the vanilla cell, for this reason they are one of the most used RNN still today. The main reason is the memory cell, which can improve the ability of the network to remember things over time. An intuitive example is to set the forget gate to 1 (remember everything), in this configuration the info in the cell is preserved indefinitely, this behaviour is not possible with a vanilla RNN. However LSTM does not guarantee the total absence of vanishing gradient but it reduces it by putting these

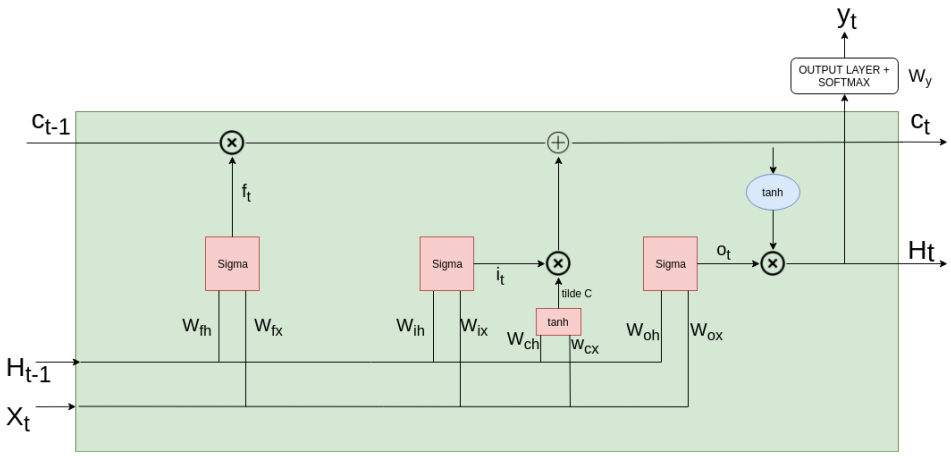

Figure 2.8: A diagram showing the internal structure of a LSTM cell.

shortcuts between distant input, which help gradient to flow better climbing over the intermediary timestep gradients.
Different LSTM cells can be stacked together to form a deep neural network and they are typically used in a bidirectional fashion. Bidirectional LSTM is composed by two LSTM, one reading the input from right to left, the other reading the input from left to right. LSTM networks need a lot of computational power to be trained, for this reason in last 5 years a simplified version called GRU was introduced.

2.2.3 GRU

Gated Recurrent Unit (GRU) is a type of recurrent cell proposed in 2014 by *Cho et al.*[28] to overcome LSTM long training time. A GRU cell has, at each timestep, input and hidden-state only, the cell-state of LSTM was removed. Similar to LSTM, it presents two gates (versus 3 of the LSTM): update and reset gate. The update gate controls which parts of the hidden-state have to be updated versus which parts have to be preserved. This gate plays the role of both the forget and the input gate of LSTM. The reset gate instead controls which parts of the previous hidden-state will be used to compute the new content. Basically the mechanism that regulates the GRU is to insert new contents only when forget. The equations for the gates behaviour are presented in (2.17)

$$\begin{aligned} u(t) &= \sigma(W_{uh}h^{(t-1)} + W_{ux} \cdot x^{(t)} + b_u) \\ r(t) &= \sigma(W_{rh}h^{(t-1)} + W_{rx} \cdot x^{(t)} + b_r) \end{aligned} \tag{2.17}$$

As in LSTM, the gates are used to produced an output. Equations in (2.18) represents the mechanism to produce the next hidden-state. $\tilde{h}(t)$ is the new hidden state content selected from the previous hidden-state by the reset gate.

$$\begin{aligned} \tilde{h}(t) &= \tanh(W_{hh}(r^{(t)} \cdot h^{(t-1)}) + W_{hx} \cdot x^{(t)} + b_h) \\ h(t) &= (1 - u^{(t)}) \cdot h^{(t-1)} + u^{(t)} \cdot \tilde{h}^{(t)} \end{aligned} \tag{2.18}$$

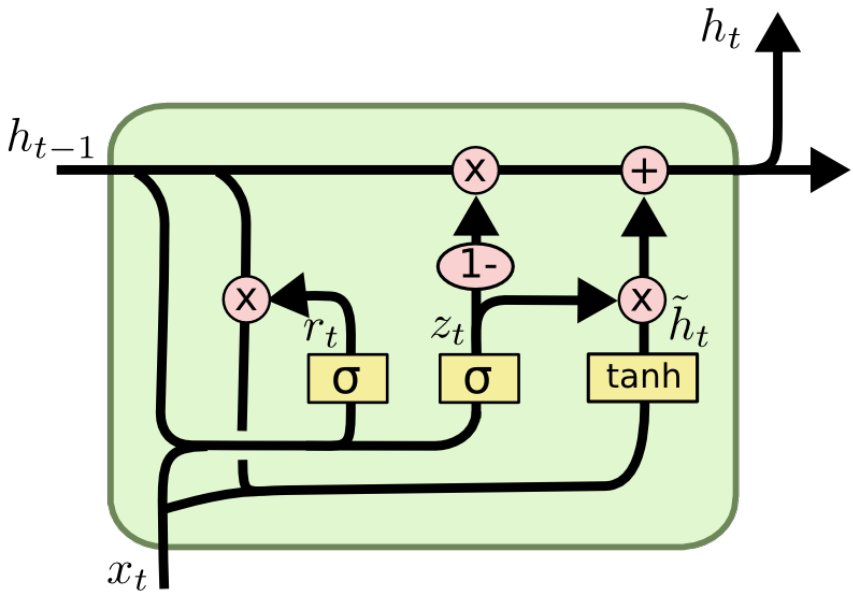

Figure 2.9: A diagram showing the internal structure of a GRU cell. Source [4]

GRU has proved to be faster to compute respect to LSTM and, even with less parameters, has the same performance.

2.3 Attention

Attention is a today widely adopted technique initially introduced for Neural Machine Translation. The underlying concept is to allow the network to focus on the most important pieces of input information at each timestep. It was inspired by the mechanism of human attention. When looking at one image, humans always focus on small portions (the ones containing information) instead of focusing on the entire pixels. Attention is today replacing all the recurrent neural networks based models for NLP and is at the base of new transfer learning era. In the following sections, the main aspects of the attention are reported.

2.3.1 RNN with attention mechanisms

The first use of Attention was on translation tasks. Neural Machine Translation (NMT) typically exploits an *encoder-decoder* architecture for sequence-to-sequence (seq2seq). The encoder receives all the words from source language and produces a sentence encoding as last hidden state. This vector is then passed as initial hidden state for the decoder that is trained to produce a sequence of words as output. The idea is that the encoder encodes the input sentence at higher level, while the decoder extract one word at time from this vector in the target language.

In a paper by *Bahdanau et al.* [29] has been shown how encoding the

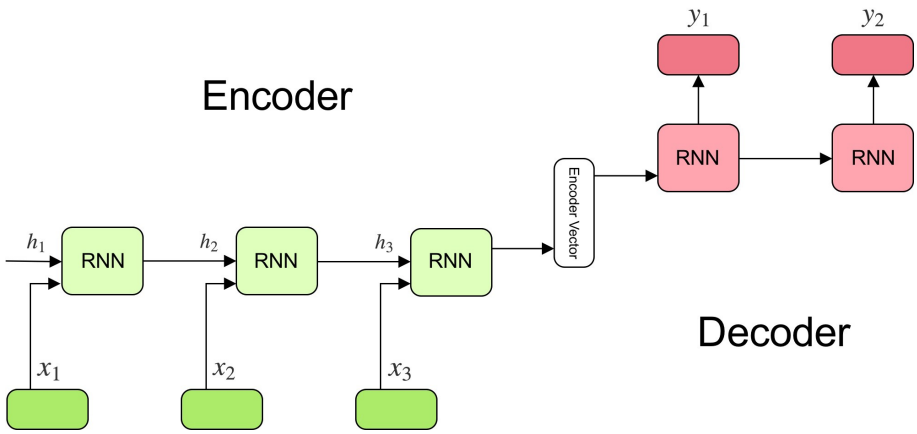

Figure 2.10: A traditional encoder-decoder based on RNNs. Source [5]

entire input sentence in a fixed length vector is a great bottleneck for such architecture. The paper then proposes an alternative method that enables the encoder to focus each time only on relevant pieces of the input sequence. This mechanism took the name of attention (also called soft-alignment in NMT). The new encoder-decoder architecture integrates a way to directly connect the decoder to the source sequence. Each time a new input is processed, the decoder receives a weighted average of the encoder hidden states, where the weights for the average are the attention scores for each input word. These weights are computed by an attention layer which performs a dot product between the current decoder hidden-state and each encoder hidden-states. The final results are then summed into a single attention vector which put higher importance to the hidden-states that mostly match the current decoder hidden-state. In the original architecture, the decoder receives, at each timestep, the previous decoder hidden-state concatenated with the attention vector and the previous predicted token as input (a simplified diagram is shown in figure 2.11). In a more formal way the attention

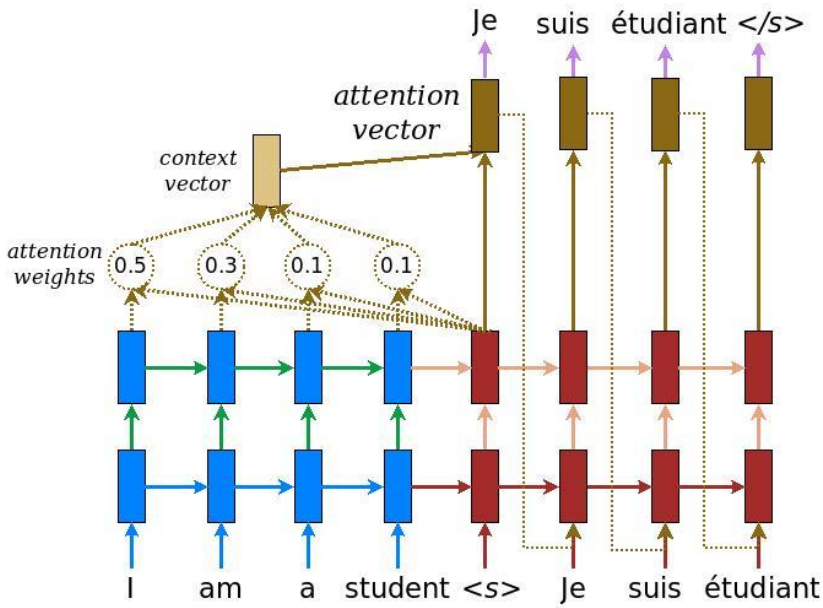

Figure 2.11: 2-layer encoder-decoder for NMT with attention mechanisms. Source [6]

score $e^{(t)}$ is computed as in 2.19, where s_t is the decoder hidden-state at timestep t. The dot product computes the similarity between each encoder state and the current decoder one.

$$e^{(t)} = [s_t \cdot h_1, \ldots, s^t \cdot h_N] \tag{2.19}$$

The softmax is then applied to the attention score $e^{(t)}$, to retrieve values between $[0, 1]$ for each encoder state. The final sum will be equal to one as an ordinary probability function. The α^t are the weights for each encoder states (2.20).

$$\alpha^t = softmax(e^t) \tag{2.20}$$

The attention weights α_i are then used to compute the weighted average corresponding with the final attention output a_t (2.21).

$$a_t = \sum_{i=1}^{N} \alpha_i \cdot h_i \ , \ a_t \, \epsilon \, \mathbb{R}^{NxK} \tag{2.21}$$

The final attention output a_t is then concatenated with the current decoder hidden-state s_t and fed, together with the previous predicted word, to decoder RNN.

$$[a_t; s_t] \, \epsilon \, \mathbb{R}^{NxK} \tag{2.22}$$

One important aspect of this mechanism is that it does not introduce any other parameters to be learned, but it simply forces the encoder and decoder interpretation for a word (or a bunch of words) to be similar (high dot product) if the concept is the same, this mechanism is called soft-alignment. The attention does this by introducing some direct connections between encoder and decoder, thus making the gradient flows better and towards the right input states.
While the above example takes in consideration an NLP problem, attention is a general Deep Learning technique used in different domains like Computer Vision. A more general definition of attention involves the concepts of query and values that will be then mentioned again in the next section. Given a *query* vector and a set of vectors called *values*, the attention is a mechanism to compute a weighted sum of values, dependent on the query. Based on the value of the query, this technique will pay higher attention on different values. This concept is similar to the one seen with LSTM and GRU, the architecture should have the possibility to decide what information has to flow.
Even if the concept remains the same, there are different variants of attention which simply exchanges the way the attention score $e^{(t)}$ is computed. Some of them are listed here below.

- Basic dot product $e_i^{(t)} = s^t \cdot h_i$, $(d1 = d2)$
- Additive attention $e_i = v^t \cdot tanh(W_1 \cdot h_i + W_2 \cdot s^t)$
- Multiplicative attention $e_i^{(t)} = s^t \cdot W \cdot h_i$

Another important peculiarity of attention is the interpretability it gives by inspecting the attention scores gave to each state. This aspect is crucial in a neural network architecture which is commonly has a very poor level of interpretation.
In the next section self attention mechanism, which is at the base of *Transformer*, will be analyzed.

2.3.2 Self-attention and Transformers

The *Transformer* is an encoder-decoder architecture introduced by *Vaswani et al.* [7] in 2017 at *Google Brain* team, originally used for NMT. The main novelty introduced is the total absence of any recurrent unit, totally replaced with attention mechanisms. This new architecture overcomes the two main problems with LSTM, as well as bringing new state-of-the-art results:

- Difficulty to learn long-term dependencies when sentences are too long. The probability of keeping the context from a word decreases exponentially with the distance from it.
- Hard to parallelize given the recurrence, which translates in a very slow training time.

The basic idea is to overcome the LSTM problems by simply putting attention connections between different words. The original *Transformer* is composed by 6 encoders and 6 decoders. Each block contains a *Multi-Head* unit to allow each word to attend to all of the others in the input sequence. The model exploits the *self-attention* technique to discover intra-sentence relations. It also includes a positional embedding system to embed, inside each word, the information about its position in the sequence. In the next sections all the main components of a *Transformer* are explained. A complete image of *Transformer* is reported in (2.12).

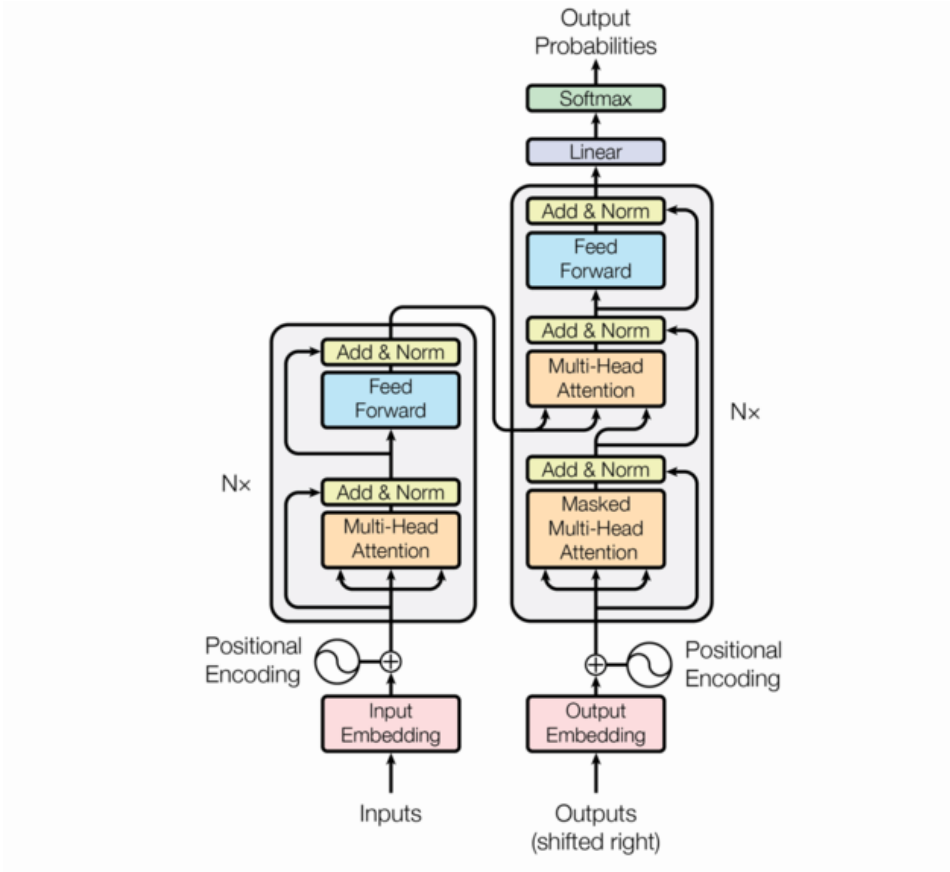

Figure 2.12: The transformer proposed by *Vaswani* [7]

Encoder

The encoder learns the dependencies between any pair of input words (or subwords as in BERT) through a self-attention mechanism. Self-attention

consists of three main components:

- Query Q
- Key K
- Values V

The attention score for a pair of input X_i and X_j is the weighted sum of the *Values*, where the weight are computed as a normalized dot product between *Query* and *Key* as in (2.23) (d_k is the dimension of the key).

$$Attention(Q, K, V) = softmax(\frac{Q \cdot K^T}{\sqrt{d_k}})V \tag{2.23}$$

Q, K and V are projections of the original input along three different matrices: W^q, W^k and W^v. These matrices are the ones that the Transformer has to learn and they are used to represent the three components in different flavours. Equations (2.24) represent the creation of Q, K and V for the computation of the attendance score of input X_i with X_j (attention is computed also between a word and itself, so when $X_i = X_j$).

$$\begin{aligned} Q &= X_i \times W_q \\ K &= X_j \times W_k \\ V &= X_j \times W_v \end{aligned} \tag{2.24}$$

For an entire sentence of length N the output vector Z_i (corresponding to query of X_i) contains a number of entries equal to the length of input sequence, each of them weighted based on the attention score between X_i and X_j (2.25).

$$Z_i = \sum_{j=1}^{N} [softmax(\frac{Q_i \cdot K_j^T}{\sqrt{d_k}})] \times V_j \tag{2.25}$$

One of the great idea of Transformer is not to limit the attention score to only one "point of view". The *Multi-Head Attention* module performs several attention computations in parallel, each head has it own W^q, W^k and W^v. Each head produces a different Z_i that are then concatenated, multiplied by a learned matrix W^o, and sent to a 2-layer FFNN to squeeze them and produce a single output for each pair (X_i, X_j). The basic idea of multi-head is that there are different ways a word can attend to another, for instance the heads can represents the questions "To whom?", "For what?" etc. Equation (2.26) describes the computations for the $head_i$, while (2.13) shows the final *Multi-Head attention* block.

$$head_i = Attention(X_1 \times W_i^q,\ X_2 \times W_i^k,\ X_2 \times W_i^v) \tag{2.26}$$

The final encoder block contains also a layer for normalization and dropout,

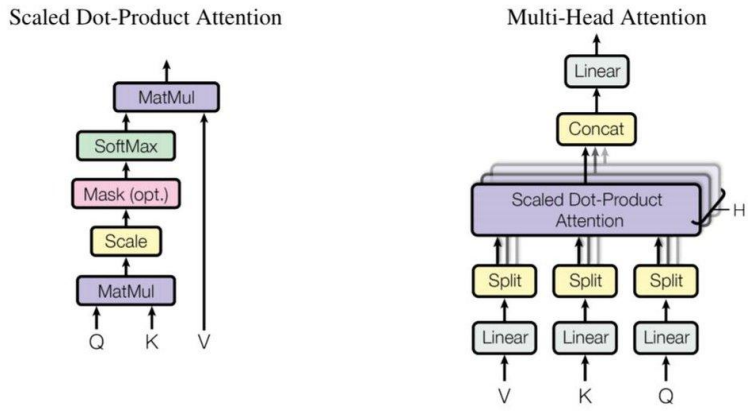

Figure 2.13: The multi-head attention block from the paper of *Vaswani et al.* [7]

in order to avoid overfitting. After multi-head and FFNN a residual connection is present to make gradient flow better (because of this deep architecture) and carry on position information given by positional encoding.

Positional embedding

Even if they are slow, RNNs give a positional information about input words due to their sequential structure. For this reason, *Transformer* needs to implement a way for introduce positions for each embedding, in order to affect the score between two words with their distance. *Vaswani et al.* includes a positional embedding mechanism by adding, to the original embedding, a sinusoidal function dependent on the position of the word inside the sentence. The positional encoding to be added to embedding is shown in (2.27) , where d_{model} is equal to 512 in the paper and $i \, \epsilon \, [0, 255]$ is the position of the word. In this way, the repetition of a word in the sentence will lead to different embedding, dependent on the position.

$$\begin{aligned} PE(pos, 2i) &= \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \\ PE(pos, 2i+1) &= \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \end{aligned} \tag{2.27}$$

Decoder

The decoder is the module responsible for generating a sequence, conditioned by the encoder output, which minimizes a given loss function. The

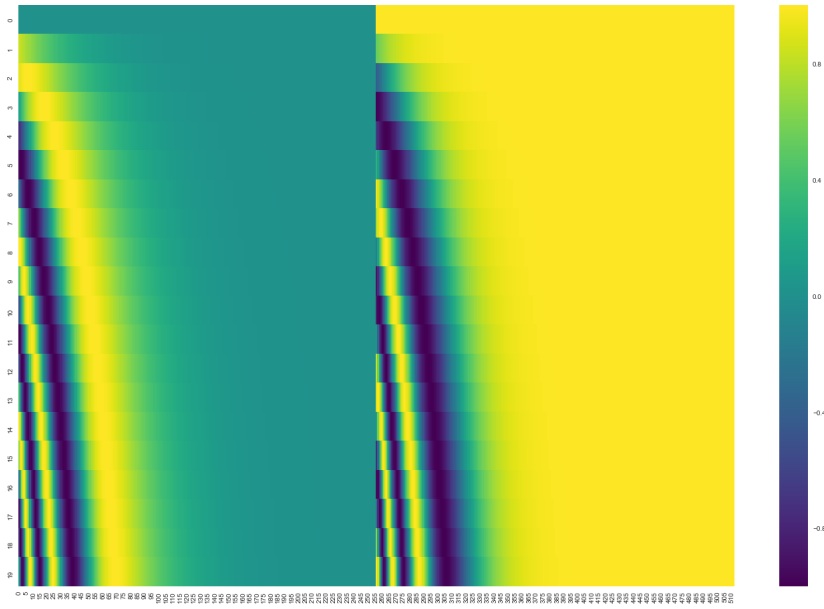

Figure 2.14: Positional encoding spectre for Transformer. The values of the left half are generated by one function (which uses sine), and the right half is generated by another function (which uses cosine). Source [8].

Transformer decoder is similar to the encoder block. The *Masked Multi-Head Attention* is the module responsible for applying attention on the decoder generated sequence. Additionally to the encoder, it integrates a way for masking out some words. This functionality is needed during the training, where the *Transformer* has access to the entire target sequence, in order to compute the loss at each step. To avoid the decoder to look at the future, before attention computation, a mask is applied on the future words. In order to do so, the *Masked Multi-Head Attention*, before computing the softmax, put the attention score (the normalized dot product between query and keys) to $-inf$. In this way the softmax output for future words is approximately 0 and only earlier words are considered.
Another difference respect to the encoder is the presence of *encoder-decoder Multi-Head Attention*. This module works as the *Multi-Head Attention* of the encoder with the difference that the query is composed starting from the decoder sequence, while keys and values are composed starting from the last encoder output. The aim is to allow each word of the decoder to attend to the ones of the input, so learning the alignment. After finishing the encoder phase, the decoder one begins. At each decoder step, the last

decoder block is a vector which depends on the entire encoder output (by focusing more on important features) and on previous generated words. This vector is sent to a linear layer and then to a softmax one, to produce the probability distribution of this vector over the entire vocabulary set. The softmax will produce a vector having dimensionality *vocab_size*, where the highest probability is assigned to a certain word.

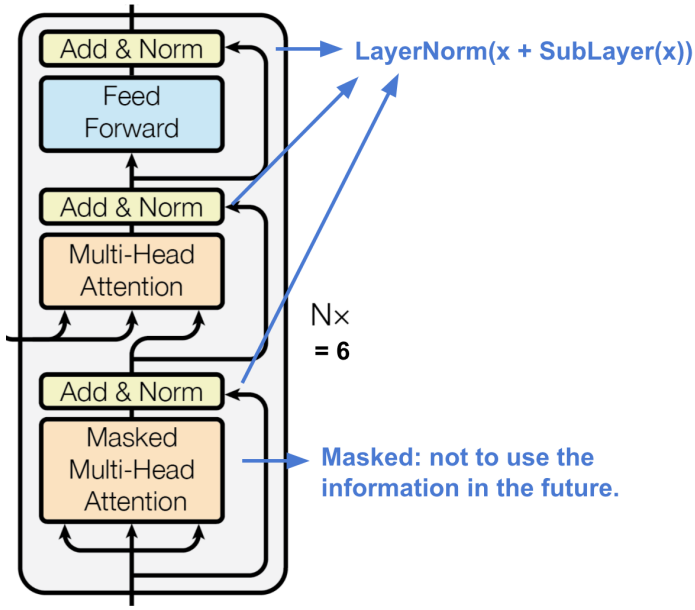

Figure 2.15: Transformer decoder architecture. Source [9]

Architecture details

The *Transformer* can be finally decomposed in various fundamental blocks, each of them composed by two different sublayers: the self-attention and the FFNN. The self-attention sublayer has to learn 3 different matrices W^q, W^k and W^v, for each head, plus an additional matrix W^o used to compact different heads output together. The matrix dimensions are $W^q \epsilon \mathbb{R}^{d_{model} x d_k}$, $W^k \epsilon \mathbb{R}^{d_{model} x d_k}$, $W^v \epsilon \mathbb{R}^{d_{model} x d_v}$, $W^o \epsilon \mathbb{R}^{h d_v x d_{model}}$, where d_{model} is set to 512, h is the number of heads equal to 8, and dv and dk are the dimensions of keys and values, both set to 64. The FFNN receives one input for each element in the sequence, each of one having dimension d_{model}. The FFNN hidden layer has dimension 2048. The final output for a decoder block is one vector for each decoding step, each of them having dimension d_{model}. Additionally the embedding layer has to be learned, it produces embedding having size d_{model}, which are shared between encoder and decoder. The

total number of blocks for both encoder and decoder was choose to be 6.
By reducing the size of each vector inside the self-attention sublayer, it is possible to compute different attention heads simultaneously with the same complexity of computing only one with the original dimension.
Each encoder receives as input the output vectors of the previous one, except for the first one, receiving the embedding (the same for the decoder).

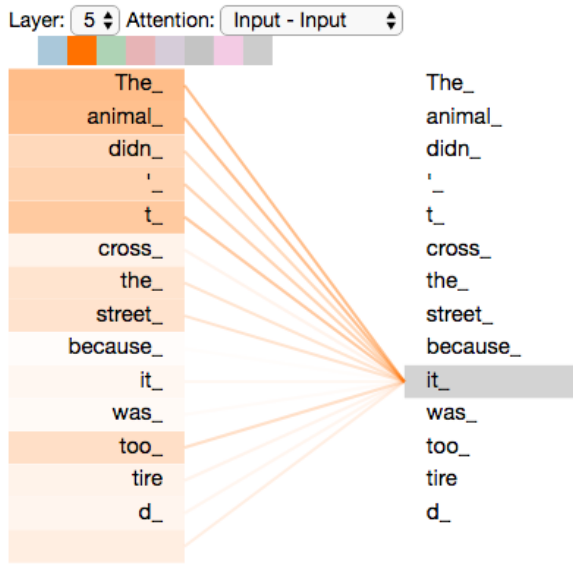

Figure 2.16: Transformer self attention can be easily exploited for coreference resolution. The word "it" has an high attention score for the words "The animal". Source [8]

2.4 Transfer learning

Transfer learning is a technique born with the necessity, in modern Deep Learning, of having a huge size labelled data and computational resources for the training of a neural network classifier. Transfer learning consists in re-using already huge trained model by adapt (fine-tuning) them on different, but similar, tasks. The basic idea is that a deep neural network learns representations, at different level of abstraction, for each hidden layer. In the literature this representation was often referred to the concept latent space (or hidden space), which is the space in which this representations of features live. The first hidden layers could learn representations which can be re-used for other similar tasks. An example could be computer vision

tasks. The first layer of a CNN will learn to distinguish the basic shape like angles, round etc. These features are "context independent" and are useful for any computer vision tasks. Since today, the predominant way to do transfer learning was on computer vision. *ResNet*[30] is one of the most famous pre-trained models, it was trained on a huge image corpus and it is widely exploited for fine tuning tasks. In this way researchers can use a pre-trained model and avoid long training time, only few epochs for fine tuning should be enough. While for computer vision transfer learning has been available for some years, NLP struggled since 2018 to publish the first real transfer learning model. Before that date the only available transfer learning technique was the use of pre-trained word embedding. The first real breakthrough in this direction was *ELMo* [1], a contextual embedding model, based on neural language model, for fine tuning tasks. Despite the today countless new model, this work will describe **BERT** [10] which represents for sure the major breakthrough in this new NLP era.

2.4.1 Contextual embedding

The original word embedding were characterized by the presence of a single fixed vector for each word. Such vector include a statistical distribution for the word by means of its co-occurrences with other ones. These vectors are easy to load in a model, since only word-vector mapping is needed, and produces good results compared with other encoding techniques like one-hot. Even if these aspects, some problems remain still open. One of the them is the word disambiguation. Very often the language presents some word which can have multiple meaning based on the context of use. A possible example is the word "bank" that can have a different meanings when is nearby the word "river" or the word "money". Both the concepts are expressed with a single word but they have a total different aim. The first is referring to the bank of the river, while the second one with the bank as an institution for money deposit. This language peculiarity is called polysemy. To handle this, the modern tools have to take care of the context in which the word appears. Contextual embedding are a powerful tool to do that, they can create an embedding of the word based on "original embedding" and on current contextual information. By doing so, the embedding are unique for each context, even if they are similar for similar context. Thus the embedding of the word bank will have a different shape when is referring to the bank of the river and when is referring to the bank as the institution. This tools are today widely used in NLP because they achieve great results and have replaced the leaderboards in all the major tasks. Even this promising introduction, contextual word embedding words requires to download the whole pre-trained model, which is typically very huge. This

lead the today NLP research to exploit very large computational resources which put a cost boundary for several research projects.

2.4.2 BERT

BERT (Bidirectional Encoder Representations from Transformer)[10] is a deep model for contextual embedding extraction from text, proposed by *Jacob Devlin* at Google AI in 2018. The model uses a stack of 12 (or 24) transformer encoders to produce an embedding for each input word, by performing self attention among them. The main novelty BERT introduces is the bidirectionality. Until then, language modelling did not use a real deep bidirectionality, this is due to the fact that, in a deep RNNs, bidirectionality allow the word to see itself in the second layer. This phenomenon can be easily proved thinking of a language modelling task with a bidirectional LSTM. The final hidden state will consists in the concatenation of the last backward state and the last forward state. Passing this hidden-state to a second layer means to allow the forward LSTM so see also the future (already encoded in the vector), and so the language modelling task deviate to a simple "copy" of a word read from the hidden-state. BERT overcomes this problem by avoid the standard language modelling task and proposing a Masked Language Model (MLM). A masked language model is the task to predict a masked word that could be in any position inside a given sequence of words. The probability distribution is slightly different from the one of a language model, but in this way BERT can exploit both left and right context for the word prediction. This intuition is one of the most important because gives to BERT the possibility to include, inside the embedding, the information from the entire context simultaneously. Other models like ELMo train two LSTM (forward and backward) independently and then concatenate them.
BERT implements a slightly modified version of the original transformer encoder, in order to make possible to mask out some token, it uses the masked multi-head attention, which is typically used in the transformer decoder. By exploiting several multi-head attention layer, BERT builds a contextual embedding for each word and then compute the loss for the masked one.
To allow BERT to be versatile on different tasks, *J. Devlin* proposes to train BERT jointly also on another task, the next sentence prediction. The BERT model can handle a pair of sequences separated by a special token, the [SEP]. The second learning task consists in predicting if the second sequence could be an acceptable continuation of the next one. This binary classification task gives to BERT the ability to work also at sentence level and not only

with words, thus allowing it to be exploited also on QA tasks. The token used for this task is the [CLS], which will represents a digest of the entire input sequence. BERT embedding is learned from scratch, it consists in

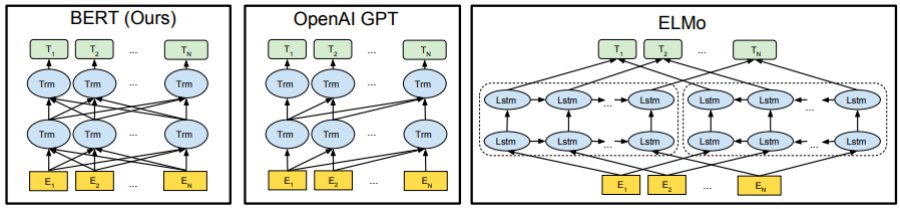

Figure 2.17: BERT uses bidirectionality to predict the masked words. Other model uses only left-directional transformers or bidirectional LSTM trained independently. Source [10]

three main pieces then summed up together. Like the original transformer, there is positional embedding. This information is of crucial importance in this type of tasks, the transformer has to know the relative position of the masked word in the sequence. In order to work with a pair of sequences, BERT needs to include also an information of "position at sentence level". Two segment embedding are learned, one for the sentence A and one for the sentence B. The last embedding is the one related to the single token. BERT has a huge vocabulary of 30 thousand English words, each of them should be mapped to a given embedding. In reality BERT works with subwords, for instance it will tokenize the word "playing" in "play" and "##ing", this trick is used to decrease the vocabulary size by learn the most used part of words. These three embedding, the positional, the segment and the subword, are then summed together to create the final embedding that is fed to BERT. BERT came out in two different variants, the base and the large. The base

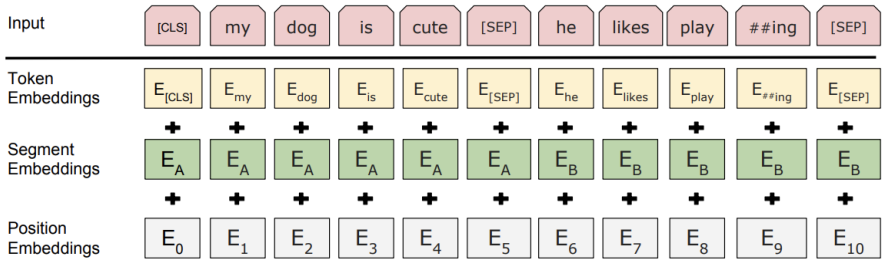

Figure 2.18: BERT embedding computation is the sum of three different components. Source [10]

is composed by 12 encoders, 768 dimensional contextual embedding and

12 attention heads for a total of 110M parameters. The large is composed by 24 encoders, 1024 dimensional embedding and 16 attention heads for a total of 340M parameters. The base version is much less expensive and was trained to be compared to other models of the same size such OpenAI-GPT. The large is instead much more expensive and it typically beats the base version scores.
BERT was built for fine-tuning tasks and today is the most used pre-trained model for NLP tasks. It was the first breakthrough in this new NLP era and it was of inspiration of tons of new contextual embedding models, and more have to come. BERT can be easily fine-tuned of different tasks like QA, sentence classification, NER etc. In the following chapter a way to exploit BERT for multi-turn session detection and classification will be presented.

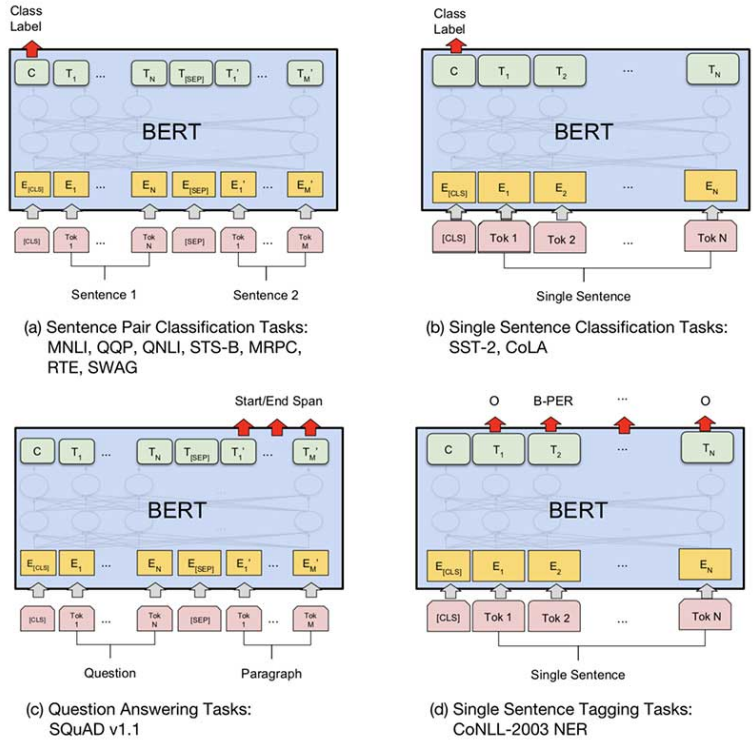

Figure 2.19: BERT can be used for several NLP tasks by exploiting its flexible input sequence. Source [11]

Chapter 3

Methodology

In this chapter the core part of the proposed work is reported. **MTSI-BERT**, which stands for Multi-Turn Single-Intent BERT, is a BERT based model for session classification in a multi-turn conversational scenario. Developed in Python programming language, it exploits the *PyTorch* framework, together with the *PyTorch-Transformer*, a package containing the BERT implementation for PyTorch. In the following section a deep overview of MTSI-BERT, together with the architecture structure, is done. The results of the model will then be presented in the next chapter 5.

3.1 Approach

MTSI-BERT (Multi-Turn Single-Intent BERT) is a joint model for intent classification, action extraction and end-of-session prediction on multi-turn conversational sessions. It was created with the idea to test if BERT could be useful in conversational scenario, where multiple utterances are presents, each one with strictly relationships with the others. The model development followed the flow of a natural dialogue between an user and an agent by modeling the input accordingly. In the following subsections all the aspects of MTSI-BERT will be explained. In the fist part the explanation of the input shape and the dialogue management is given. In the second instead a focus on the neural architecture is done.

3.1.1 Model input

Input shape

MTSI-BERT was trained to make it consistent with a conversational scenario. Such scenario requires the agent to wait for the first user interaction, understand the intent, the action and extract the useful entities. Then it

has to formulate a coherent answer to reply to him and then come back to wait for the new user interaction. While the answer generation was not part of this work, the main focus remains on the session classification. The new user interaction could be related to the previous question-answer pair or not. In the first case a dialogue continuation is happening, which is the aspect characterizing a multi-turn conversation. If, instead, the new user interaction is not related with the previous exchanged utterances, then a new session begins, with a new intent and a new action. The agent needs to detect such change in order to flush the "statistical history" and trigger the intent and action classification modules.
In order to allow the agent to work in this scenario, the input has to be shaped to simulate a conversation. MTSI-BERT was trained with a *QAQ triplet* as input. Such triplet consists in the concatenation of the last three exchanged utterances, where Q represents the user question and A the agent answer. A more formal and complete representation is possible by defining a conversation as an ordered sequence of QA, where eventually the A can be optional (for instance when the user ends the conversation). Formula (3.1) tries to give a formal description of the conversation. $CONV$ represents the sequence of QA pairs, where each answer is identified by the related source question. Such sequence must contain at least one pair of QA to be considered a conversation, the "+" sign is the regex symbol to indicate "at least one". A definition of Q and A is proposed by using *regex* expression. The regex $[a-z, .?!]$ represents the set of alphabetic characters and the most important punctuation symbols (not exhaustive), while the symbols $+$ and $*$ respectively mean "at least one" and "zero or more repetitions", thus indicates the optional answer.

$$\begin{aligned} CONV &= [(Q^{(i)}, A^{(i)})^{+}],\ \forall i \in [0, N] \\ Q^{(i)} &= [a-z, .?!]^{+}\ and\ A^{(i)} = [a-z, .?!]^{*} \end{aligned} \tag{3.1}$$

The session is instead identified as a sub-sequence of $CONV$. A single $CONV$ sequence can contain several non-overlapping sessions. All the QA in a single session are related each other and aim at satisfying a single intent with a single knowledge-base action. Thus in order to identify sessions, discontinuities in the $CONV$ sequence have to be found. To do so, MTSI-BERT receives a series of utterances with the following shape: if the question is the first of this session, than it will be analyzed individually (the answer is not yet formulated), otherwise the user question is compared with the previous QA pair. MTSI-BERT analyzes the first question of the session and extract intent, entities and action, then, for the rest of the session, it analyzes a triplet of QAQ to extract the entities from the last question and catch the dependencies between the last question and the previous QA pair.

The input is formatted to be fed to BERT and so with [CLS] and [SEP] tokens. The triplet is characterized by the presence of the [SEP] token between the previous QA pair and the new user question. In this way is possible to take advantage of the next sentence prediction original training task of BERT. An input example through time is available at (3.1). The figure shows an end-of-session at $t = 3$, thus the input is flushed and the new session utterance is analyzed individually.

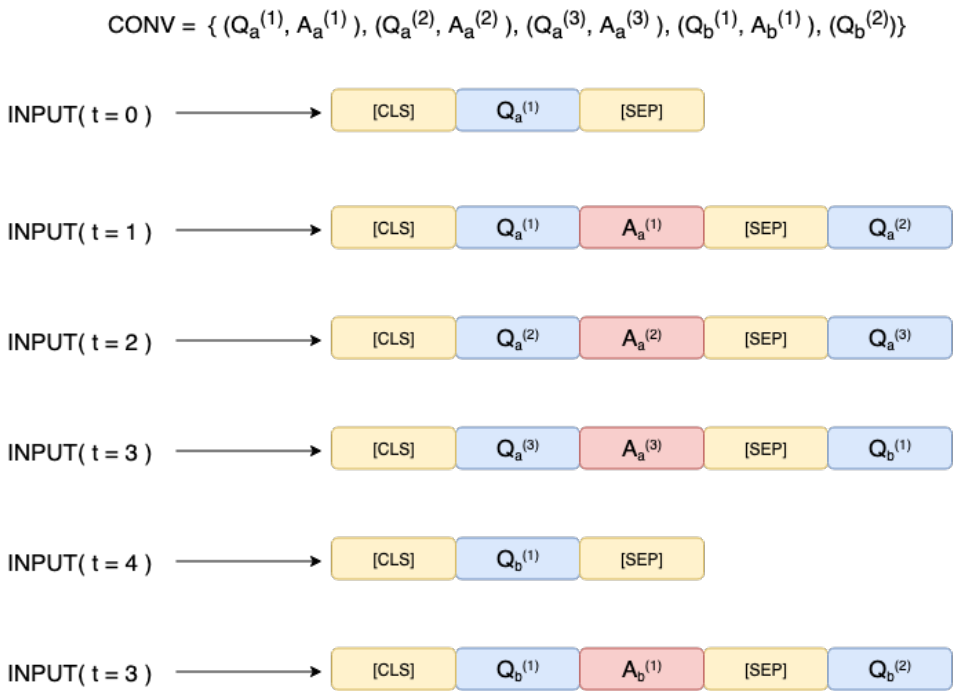

Figure 3.1: MTSI-BERT input. The blue boxes corresponds to user utterances, the red boxes to agent utterances and the yellow boxes to BERT tokens. When a new session is detected, the input is flushed and the first utterance is analyzed individually.

Dialogue granularity

To create the *CONV* sequence, all the sessions are concatenated to form a single macro-dialogue. In terms of input shape, the dimensions will be DIALOGUES_NUM x DIALOGUE_LEN x SENTENCE_LEN, thus the batch size correspond to the number of dialogues. Unfortunately an issue arises with BERT implementation. BERT actually accepts an input of dimension BATCH_SIZE x SEQ_LEN which has one dimension less than

the desired one. A possible way is to loop, for each dialogue, on BERT by feeding it with a new dialogue each time. While this way seems to be trivial, it has two main drawbacks. The first one is that loops are inefficient on GPUs, thus the input should always be vectorized since GPUs are faster to do matrices multiplications instead of iterations. The second and most important issue is related to how *PyTorch* library works. The dynamic graph of PyTorch implies that the computational graph is instantiated each time the forward pass is called. This means that a number of computational graphs equal to the number of dialogues in one batch are created. Since the BERT graph is computational expensive, a batch size of 3 already occupies more than 10 GB of GPU vram. For these reasons the number of dialogue processed each time was set to be 1 and, in order to allow the end-of-session task, the current session is concatenated with the first sentence of another randomly chosen session. The input shape will then be equal to figure (3.2).

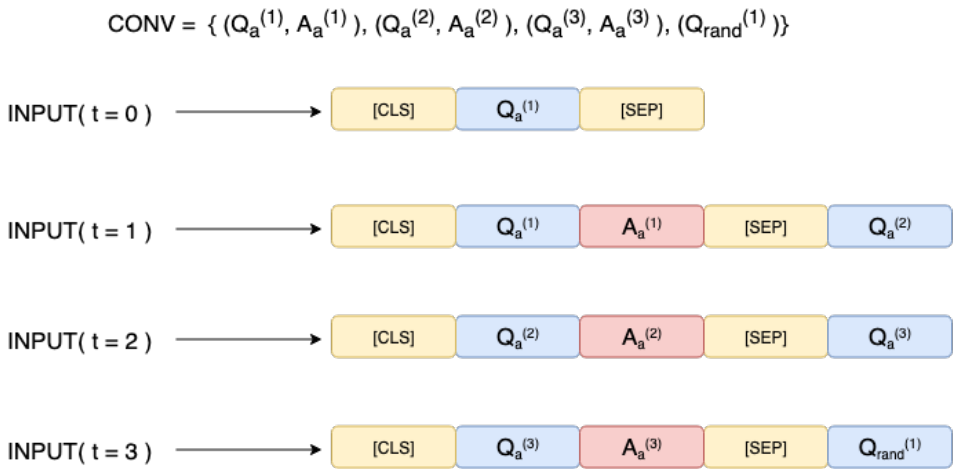

Figure 3.2: MTSI-BERT real input consists in a single session concatenated with the first sentence of another session randomly chosen. In this way is possible to train the model on the end-of-session detection task even with BERT input limitations.

3.1.2 Architecture

Joint model

A joint model is a particular model which was trained to perform more than one single prediction task. It is based on the same idea of transfer learning, the latent space representation can be used for similar tasks. Joint NLP

models were often used to do intent classification and NER, by exploiting the same initial layers of the model. Even BERT was trained as a joint model, it aimed to predict both the masked words and the next sentence flag. *MTSI-BERT* reuses the same idea to perform the three different tasks jointly: intent and action classification, end-of-session prediction. The model has a common basis composed by pre-trained BERT encoders then followed by different branches, one for each task. BERT produces the contextual embeddings for each input token, which are then used to fed the upper neural networks.
Since the model can be trained to optimize a single objective function, the loss must be unique. Thus the three different tasks must produce, at the end, one single loss to minimize. In order to do so, the final loss could be created by combining the three original losses in different ways. In this work the most basic was used, consisting in gives the same importance to all the three losses. Equation 3.2 shows the final loss of MTSI-BERT, consisting in the sum of the three single-task losses.

$$\mathscr{L}_{joint} = \mathscr{L}_i + \mathscr{L}_a + \mathscr{L}_{eos} \tag{3.2}$$

To minimize (3.2), all the individual losses have to be minimized, in such a way is possible to train the model on the three tasks.
During the training some conflicts between tasks came out, which have lead to the adoption of different architectures. The best three architectures are presented in the following subsections of this work.

MTSI-BERT Basic

The first and most simple architecture was the one which exploits only the *[CLS]* token contextual embedding. Such token is the first in the sequence and was used, in BERT training, for the next sentence prediction task. Such embedding contains semantic information about the whole input sequence and so it can be used for sentence classification tasks. The most basic architecture for sentence classification is the one involving only that contextual embedding. MTSI-BERT basic does exactly this thing by feeding the single *[CLS]* embedding to three different classification layers (also called output layers), one for end-of-sentence and the other two for intent and action classification. The three classifiers are then followed by a softmax function that outputs the probability distribution for that particular prediction.
Even if simple and relatively fast compared to the others, such architecture presents a main drawback. By looking at the training loss, it was noticed that the three single losses decreases in the fist epochs and then, after 4-5 epochs, the two losses corresponding to intent and action continue to decrease, while the one corresponding to the end-of-sentence stops decreasing.

Such behaviour have lead to think that the end-of-sentence task is less compatible with the other two, while intent and action goes well together. The reason of that could be found in the fact that, with this architecture, BERT is forced to collect information regarding end-of-session, intent and action on the same contextual embedding. If two of them are incompatible, then the model adjust its weights to decrease only one, trying to minimize the final loss. So, since the final loss is the sum of the three single losses, the model prefers to minimize two of them compatible together and discard the incompatible one. A figure of such architecture is proposed in (3.3).

MTSI-BERT BiLSTM

Since the [CLS] contains information about next sentence prediction and intent classification, there is a better way to optimize the end-of-sentence without pulling down the other two losses. In order to do so the end-of-sentence must be "separated" from the other two tasks, even by still sharing the same BERT infrastructure. The end-of-sentence branch keeps the same infrastructure, considering that the [CLS] contextual embedding was trained on a very similar task. The intent and action classifications instead have to follow another way to correctly capture the semantic information of the sentence. A possible way is to encode the sentence inside a single hidden vector through a RNN. Such method performs quite well with reasonably short sentences, which is typically the case of a smart speaker dealing with the first user utterance. To improve such representation a biLSTM can be used. The biLSTM encodes the information in two ways, one in forward direction and one in backward direction. Finally the two final hidden states are concatenated together to form a resulting encoding vector having twice the size of the original one, thus containing more information. This vector is called *sentence encoding* and is then fed to two different linear layers, one for the intent and one for the action. These are then followed by a softmax that produces the probability distributions for the predictions. The illustrated architecture is depicted in figure (3.4).

MTSI-BERT BiLSTM + Residual

In order to make larger the separation between end-of-sentence and intent, action tasks, different layers can be added. In this way the gradient will affect mainly the last specific layers and less the layers in common. A problem that must be avoided in this case is to increase too much the depth of the network. The reason why this could be a problem is related with the vanishing gradient effect. When the network is too deep, the gradient will affect mainly the superficial layers and less the first ones. To make

the gradient flows better, residual connections (also known as highways) can be used. The last proposed architecture for MTSI-BERT follows these specifics and consists in the MTSI-BERT BiLSTM with the addition of three task-specific layers, each of them followed by a $ReLU$ non-linear activation function. The three layers do not squeeze the input but simply represent it at high level, in this way is also possible to apply residual connection to sum the produced output with the input given to the network. The three Feed Forward Neural Networks are task-specific, they have the additional aim of helping the branches to improve the learning of the task by differentiating more the computations for each of them. In this way each branch has more parameters to specialize on the given task. Additionally the FFNNs are quite small (only 3 layers) and so the training time overhead is not relevant.

NER

In this work, the Name Entity Recognition task had not the same importance of the other three. The reason is the presence of different papers made by many researchers in the past year which uses a joint architecture to perform the NER. What instead was not addressed so much is a system for session classification like the one here proposed. This is the main reason why the NER was kept aside in favor of the other tasks. A little model for NER was anyway developed by using the *spaCy* open-source library for advanced NLP applications. The model extracts the main entities from each utterance of the session, in order to allow possible future storage of information, or better, to allow the extraction of information of crucial importance for eventual reasoning processes. Despite this, a future improvements of MTSI-BERT could be to integrate the NER also, trying to understand if it can be compatible with the end-of-sentence, intent and action classification. Different experiments using BERT are already available on the web. They fed to a linear layer the BERT contextual embedding of each token and output a BIO tag (Beginning, Inside, Outside) for entities classification.

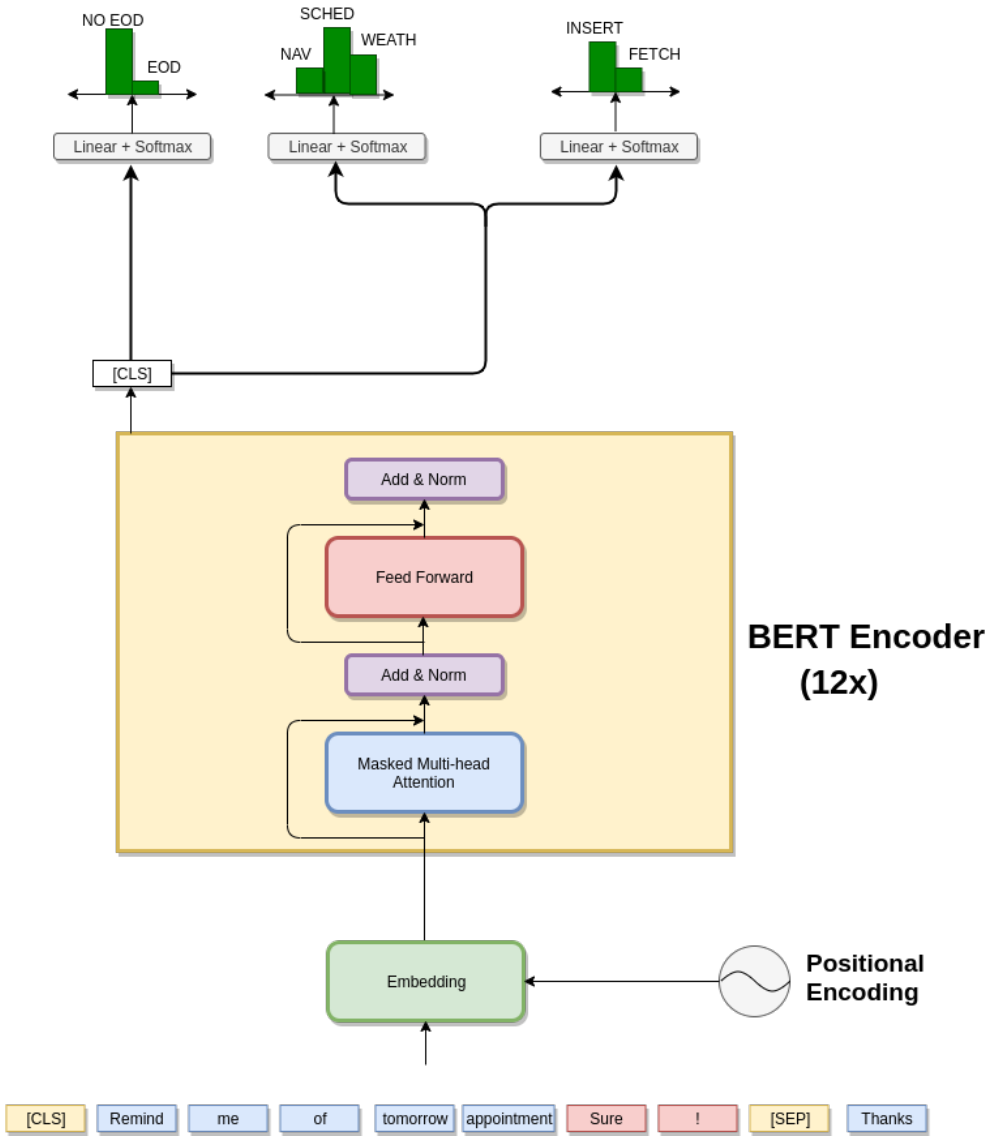

Figure 3.3: MTSI-BERT Basic.

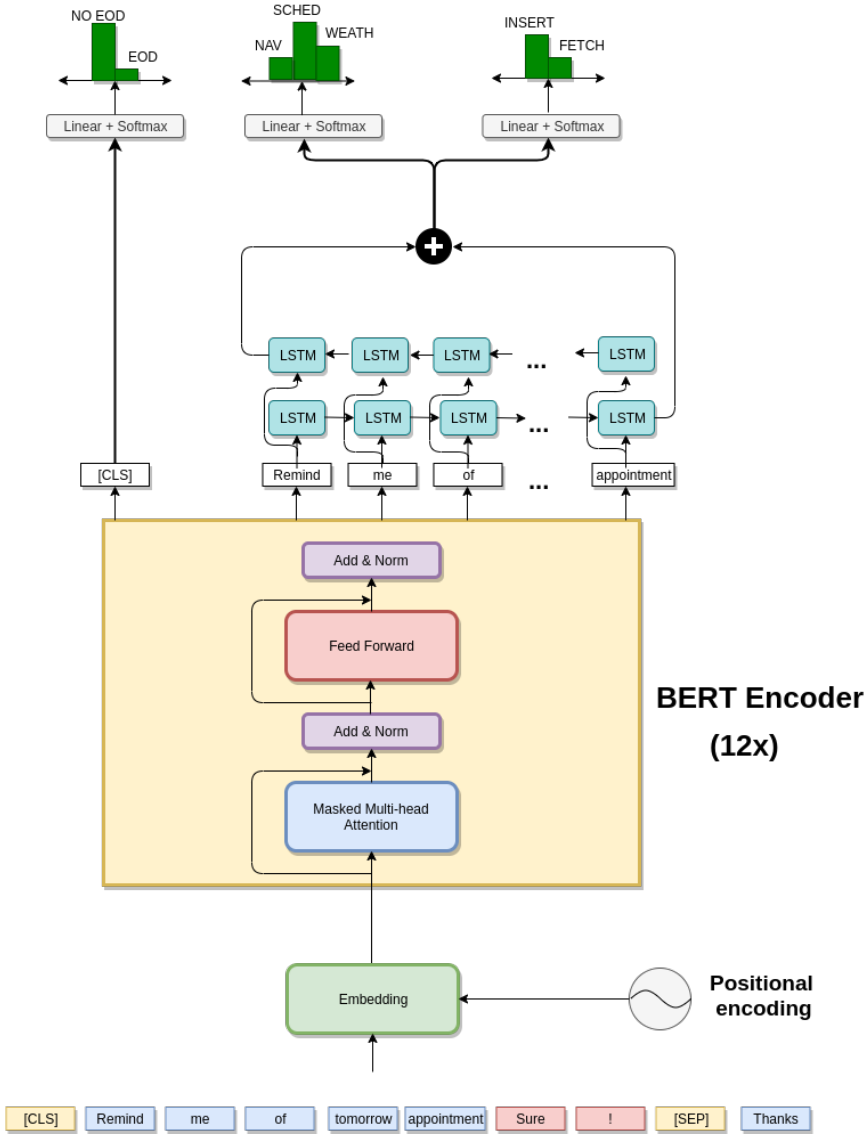

Figure 3.4: MTSI-BERT BiLSTM.

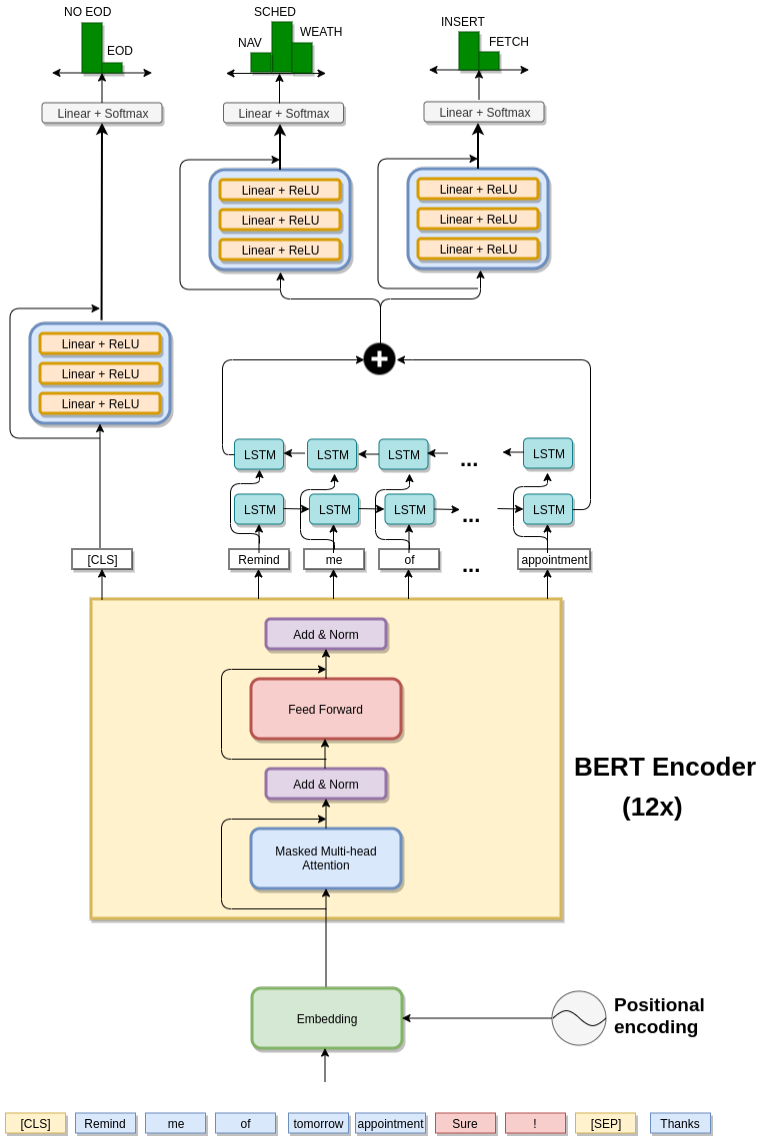

Figure 3.5: MTSI-BERT BiLSTM + Residual.

Chapter 4

Experimental setup

This chapter reports all the aspects related to the experimental setup made for MTSI-BERT, which includes the dataset, the various used frameworks and the running environment used for model training.

4.1 Dataset

A great problem of today Natural Language Processing is the small availability of labeled data. The proposed work needed a dataset containing a relevant number of conversations between an user and an agent. Furthermore the tasks of action and intent classification require additional labels for them. The *KVRET* dataset (Key-Value Retrieval dataset) [31] is the one finally chosen. The dataset was presented by Stanford student *Mihail Eric* in 2017. Collected in a Wizard-Of-Oz fashion using 241 Amazon Turk workers, it simulates conversations between a driver and a car-integrated personal assistant. The dataset is formatted as a *JSON* file, where each object corresponds to a single conversation and is divided in two sections: dialogue and scenario. The dialogue is a set of interactions between the driver and the agent. Each interaction is composed of the turn, which could be "driver" or "assistant", and the data, containing the utterance and the end of dialogue flag (True if this is the last interaction of the dialogue). If the interaction is made by the assistant, the data will also contain information about the user request and the detected slot of the previous question. The second section instead contains the information about the whole dialogue. The knowledge base is a fictitious database enriched with information needed to the assistant to reply to user questions. Each item in the knowledge base is represented by a key-value pair. The dialogues which do not require a knowledge base have no items inside it. The knowledge base has also a name to identify the data category for that particular dialogue.

The scenario section also contains the task entry, this in turn contains the intent of the dialogue. The KVRET dataset contains dialogues belonging to three different domains (or intents): weather, navigate and scheduling. The weather intent is associated to all the dialogues consisting in the user questions about the weather in a particular city and in a particular day of the week. The knowledge base will then contain the weather information for a set of cities and for all the days of the week. The navigate intent is associated to all the dialogues dealing with directions. The user typically requests indications to reach a particular point of interest, which could be referring to a specific entity (San Francisco) or a generic one (the nearest gas station). The knowledge base contains various points of interest, together with the street name, the distance and other information about the traffic. The scheduling intent corresponds to the dialogues involving an appointments scheduling or a simply reminder. In the case the user requests to schedule a new event, the knowledge base will be empty, while it will contains various appointment during the week otherwise. Finally a dialogues ID is presents, helping to uniquely identify a certain dialogue.

Turn	**Utterance**
ASSISTANT	send me to the nearest gas station
DRIVER	The nearest gas station is Valero at 200 Alester Ave, 7 miles away. Setting directions now.
ASSISTANT	Where is Valero and is there any traffic on the way?
DRIVER	Valero is at 200 Alester Ave and moderate traffic is being noted.
ASSISTANT	Thank you
DRIVER	You're welcome!

Table 4.1: An example of dialogue in the KVRET dataset.

The real task for which the dataset was created for is the key-value retrieval using Natural Language Understanding techniques. The agent has to correctly identify the intent and the entities of a dialogue in order to correctly fetch the knowledge base and provide the requested information. Despite this, the proposed work has a different aim. The detection of intent, knowledge base action and end of session is still possible with some small tricks. By carefully analyze the dataset, it can be noticed that the intent is

always clearly expressed in the first user utterance, for each dialogue. The first user interaction defines the task of the dialogue. For what concerns the action, as already explained, it refers to an insert or a fetch on the knowledge base. While this information is not directly reported, it can be extracted by looking at the entries of the knowledge base. When the knowledge base has some entries, it means that the task is about retrieving something from that. When instead the knowledge base contains no item, the dialogue corresponds to an insertion of a new item. Finally the concept of session can be adopted by concatenating together different dialogues, to form a bigger dialogue where each session is single-intent. In this way is possible to predict the end of session by concatenating a session with the first utterance of a random one, which expresses a new user request and so a new intent.
The dataset is already divided in three files, one for the training, one for the validation and one for the testing. They contain respectively 2425, 302 and 304 dialogues (sessions).

Dialogue type	**# Dialogues**
Training dialogues	2425
Validation dialogues	302
Test dialogues	304
Intent type	**# Intents**
Calendar Scheduling	1034
Navigation	1000
Weather	997
Slots	**# Slots**
# Slots type	15
Scheduling slots	79
Weather slots	65
POI slots	140

Table 4.2: KVRET statistics about dialogues and labels type. Source [2]

The number of intents is almost balanced as shown in table (4.2) and so they do not need particular attention during the training. A totally different discourse should be done for the slots. The NER, done with *spaCy*, encountered some problems with the slot labelling, since a lot of them are missing or wrongly reported. This was another reason why spaCy was chosen, instead of integrating the NER into the MTSI-BERT model.
Other task specific statistics have been made, especially regarding more

practical aspects needed to understand if the proposed model could have worked on this dataset. A technical limitation of BERT is the maximum length of input set to 512 tokens per time. A QAQ triplet is fed to BERT each time. Its length is variable also because the second question can be randomly chosen from the entire dataset. The maximum QAQ triplet length found is 125 tokens for the training set and 129, 78 for validation and test set ([CLS] and [SEP] included). These numbers are noticeably smaller the 512 tokens of BERT and gives a flexibility of application. If instead the sentences are taken individually, the max number of tokens reach 113 for training set and 111, 50 for test set. Another interesting statistic is the maximum number of sentences per each dialogues, this number was found to be equal to 12 for all the three dataset partitions. The dataset also contains some examples of subsequent utterances made by the same actor. These cases are intended to simulate a situation in which the assistant does not seem to work and so the user has to repeat, more than once, the utterance (table 4.3). In order to make the QAQ triplet construction always feasible, these cases are avoided and only the last user utterance is taken.

Partition	**Dialogue ids**
Training set	ca4a934e-8a4e-48b6-90bd-fd9f20b07180, 23dc5ff5-807b-435d-b540-7917d9fdaff2, 024d1329-6a76-4aea-a806-222bdfd252f1, 3bb3d971-d069-4208-92f4-6ce2b055ed76, 6dc75860-9ae0-4ac3-b30d-32c2379afbbf, a006f8e9-318e-4173-abcf-7dd58d3ecb97
Validation dialogues	120c9192-09e0-41d7-93d0-03c899e11571
Test dialogues	38e7f22d-5914-4fee-a696-c83bbd1be451

Table 4.3: Subsequent same actor utterances in dialogues.

For what concerns the other two labels, the action and end of session, other statics have been made. The number of "fetch" against the "insert" was found to be equal to 2012 vs 413. Validation and test set instead has 242 vs 60, 256 vs 60. This high unbalancing needed a particular care during the loss computation as described in section 3.1. The computation of end-of-session and intra-session was instead trivial. It can be easily deducted that the number of end-of-session is equal to the number of dialogues, while the intra-session to the total number of user utterances in the dataset. A final complete report of the above statistics is shown in table (4.4).

Max # tokens per dialogue	
Training set	152
Validation set	149
Test set	120
Max # tokens per sentence	
Training set	113
Validation set	111
Test set	50
Max # tokens per *QAQ* triplet	
Training set	125
Validation set	129
Test set	78
Max # sentence per dialogue	
Training set	12
Validation set	12
Test set	12
Max # user utterances per dialogue	
Training set	6
Validation set	6
Test set	7 (6 after preprocessing)
FETCH vs INSERT	
Training set	2012 vs 413
Validation set	242 vs 60
Test set	256 vs 48
INTRA SESSION vs EOS (after preprocessing)	
Training set	6406 vs 2415
Validation set	748 vs 301
Test set	810 vs 302

Table 4.4: Task specific statistics on KVRET dataset.

4.2 Training settings

One of the most important things to choose, after the architecture, are the loss function and parameters for the training. By choosing the right parameters the final model will be significantly better and will achieve very

good performance. Before deciding the parameters, the loss function has to be chosen. In this work the *Cross Entropy loss* was used during training. *CrossEntropy loss* is a way to compute the "distance" between two probability distributions $p(x)$ and $q(x)$, where q is the probability outputs from the network and p the true one. The formula is presented in (4.1), where $\hat{y}$ is the predicted probability and y is the true one. It penalizes only the prediction for the correct class without taking care how the rest of probability (up to 1) is splitted among the others labels. Another important thing to notice is that $L(\hat{y}, y) \neq L(y, \hat{y})$, so particular attention has to be given to the loss input parameters.

$$L(\hat{y}, y) = -y \cdot log(\hat{y}) \tag{4.1}$$

In the proposed dataset, the classes are very unbalanced and so a slightly modification to this loss have to be done. In particular, to improve the learning phase, higher penalization should be given to the less frequent classes like end-of-sentence. If this trick was not taken, then the model will simply content to predict the most frequent class to decrease the loss. In order to avoid such behaviour, the classes were weighted during the loss computation with a factor of $\frac{support_x}{support_C}$, where x is the class taken in consideration and C the most frequent class. This weighting was done for the action and the end-of-sentence which are very unbalanced as shown in table (4.4).
After the choice of the loss function, one of the most import parameters to choose is the learning rate. The learning rate tells how much you want to move, during parameters optimization, from the current weight value. If the learning rate is too high, then the network will not be able to reach a good local minimum because the step is too large each time. In the case of pretrained model, where a good local minimum was already reached, a big learning rate will cause to go up the hill an loosing the minimum area. When instead the learning rate is too small, the loss moves slower, this can cause the model to not converge after long training time. In the proposed work, different learning rate values were chosen, one for BERT and another for the neural networks on top of BERT. The learning rate for BERT was set to be $5e^{-5}$ as suggested in the paper, this small value ensures keeping BERT in the local minimum found with the pretraining. The learning rate value for the neural networks on top of BERT was choose to be higher. A value of $1e^{-3}$ gives good results and allows the networks to learn the three tasks without any particular problem. Another important thing to care of is changing the value of the learning rate during epochs. In the first epochs, the networks have typically to move faster from the random weights initial configuration to reach a local minimum area. When the epochs increases,

the learning rate has to decrease in order to allow the model to explore better the local minimum area without surpassing it. This allow the model to converge avoiding big oscillations of the loss. In this work the learning rate was multiplied, during training, by 0.5 every 5 epochs in the 20 epochs training.
Other important parameters to set are the ones related to regularization techniques. Regularization techniques aim to avoid overfitting, a phenomenon consisting in the inability of the model to abstract the prediction on unseen samples. Two regularization techniques were used during the training: *dropout* and *weight decay*. The dropout is a technique that consists in shutting down, with a certain probability, a given unit during the training. This feature enables to change slightly the connections between units and forces the network to learn sparse representation. The dropout on BERT output was set to be 0.5. Weight decay is another regularization technique aims to avoid the network to be too complex. The underlying idea is that lot of parameters means lot of connections between units which this can lead to overfitting the training data. To prevent this, the degree of freedom of the network must be limited. Weight decay introduces a penalization for high weights in the loss function. In this way, when gradient descent is applied, the weight is updated in function of its norm also (squared of the weight), thus penalizing more high weights. The network is then forced to find a compromise between the value of weights and the distance from the prediction to the ground truth. This compromise avoid the network to put all the weights to zero, which implies to learn nothing. The rate of penalization was set to be 0.1. In equation (4.2) the formula for loss with L_2 penalization and the relative gradient descent are shown.

$$\begin{aligned} \hat{J}(\mathbf{w}) &= J(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^2 \\ W_i &= W_i - \eta\frac{\partial J}{\partial W_i} - \eta\lambda W_i \end{aligned} \tag{4.2}$$

Finally the batch size is another crucial parameter, different values of batch size ends up with different results since it influences the way *gradient descent* optimization algorithm is applied. Three variants of gradient descent can be distinguished:

- Batch Gradient Descent
- Stochastic Gradient Descent (or SGD)
- Mini-Batch Gradient Descent

The *Batch Gradient Descent* consists in the computation of the error on the whole training set (the so called batch). The errors for each single sample are added up and the optimization step is performed at the end of each epoch. If a lot of data are present (typical of Deep Learning application), the training

will be computationally unfeasible. The *Stochastic Gradient Descent* tries to overcome this limitation by applying the optimization step after each sample. In this way the error will be noisier than batch gradient descent, but the model will converge faster. The *Mini-Batch Gradient Descent* mixes the two approaches by computing the error on a subset of the training set called mini-batch. This last approach is the most used one in Deep Learning applications because it provides a less noisier error (compared to the SGD) and it is very well optimized for GPU computations. The mini-batch, in fact, exploits GPU libraries for vectorization improving the training performance on GPUs. The batch size is typically a power of 2 because of the alignment of GPU virtual processor (VP) onto physical ones (PP). Values for batches typically range from 8 to 256 samples. In this work SGD was initially tried (because of BERT input limitations) with very poor results. The batch size equal to one provides too noisy errors and so bad optimization steps. Then, even keeping the batch size equal to 1, and so without the possibility to exploit the GPU vectorization, the adoption of batch was made. All the three architectures of MTSI-BERT were trained with mini-batch gradient descent, by accumulating the loss on mini-batches of size 16, with a significantly improvement in model performances.

4.3 Frameworks and libraries

4.3.1 PyTorch

Despite an increasing availability of open source Deep Learning framework, today research works uses *TensorFlow* and *PyTorch* libraries. These two frameworks provide implementation for several neural networks architectures like CNN, RNN, simple FFNN and recently also Transformer modules, together with different losses and optimizers. Both supports the use of GPUs for neural networks execution and have several optimizations to make them faster. TensorFlow (`https://www.tensorflow.org/`) was develop by *Google Brain* for internal use and then released as open source library in 2015. It allows to run the program on more than one CPU or GPU. In the last years it also allow to run the code on Google TPU (Tensor Processing Unit), a specific hardware chip for machine learning built ad-hoc for TensorFlow. The framework is based on a static graph paradigm. A static graph means that all the things related to a neural network have to be declared before and this seems to be more unnatural and tedious most of the times. The neural network lives inside a TensorFlow session and can communicate with the outside world only with special mechanisms. The

debugging with Python tools is not possible, special commands for debugging are needed. Another problem is the absence of modules, which brings to write boiler code every time a new architecture has to be implemented, and the complex way to distribute the work on several CPUs or GPUs. *PyTorch* (`https://pytorch.org/`) is an open source machine learning library based on *Torch* library. It is developed by a community but primarily by *Facebook AI*. It has a more pythonic way of use and is based on dynamic graph. A dynamic graph means that a graph is instantiated each time a forward method is called. This brings the possibility to debug with native Python tools and other important improvements. PyTorch has the concept of module, which make it more similar to a framework than to a library. The modules help in creating new architectures without taking care of boiler code. It supports the use of multiple CPUs and GPUs with a simple wrapper called *nn.DataParallel* and TPU support is coming with the next version. PyTorch is a new library with promising future. Today different researchers and universities prefers it to TensorFlow for its great improvements in flexibility and ease of use.
For all these reasons, the preferred Deep Learning library for this work was chosen to be *Pytorch*.

4.3.2 PyTorch-Transformers

The BERT original model was released in TensorFlow at `https://github.com/google-research/bert`. Fortunately, during the first year, some portings has been done. The most famous one was done by *Hugging Face* available at `https://github.com/huggingface/pytorch-transformers`. The library includes all the state of the art NLP models based on Transformer architecture, together with the pretrained weights converted from TensorFlow file format, for fine tuning tasks. At the time of writing, the following models are available:

- BERT
- DistilBERT
- Transformer-XL
- XLNet
- GPT
- GPT-2
- XLM
- RoBERTa

The modules typically share a common Transformer structure and are all documented in a pretty good way. The documentation is available at `https://huggingface.co/pytorch-transformers/`.
For what concerns this work, the BERT model only was needed. The BERT

model is well documented, with different examples, and ease of use. It is released in different flavours, each one can use pretrained weight for downstream tasks. The available BERT modules are the following:

- BertModel
- BertForPreTraining
- BertForMaskedLM
- BertForNextSentencePrediction
- BertForSequenceClassification
- BertForMultipleChoice
- BertForTokenClassification
- BertForQuestionAnswering

The various models mount some layers on the top for different downstream tasks (except for the BertModel). The pretrained models are available for both large and base version of BERT, with cased or uncased version and in English, Chinese, German or multilingual (which consists in a mixed words vocabulary, including Italian). In the proposed work only the BertModel was used, which is the one corresponding to the raw BERT model, without additional downstream layers on top. The library also provides other tools like the BertTokenizer, allowing to easily divide the sentence into subwords ready to be fed to BERT. The structure was done to be as most similar to the TensorFlow counterpart, in order to achieve equal performances.

4.3.3 spaCy

spaCy is a open source library for helping in the development of advanced NLP applications. It was developed by *Matthew Honnibal* now member of *Explosion AI* in Python and Cython programming languages. It provides statistical machine learning model, together with Deep Learning ones, fully compatible with the major machine learning frameworks like TensorFlow, PyTorch, Keras or Scikit-learn. It provides built in functionalities for tokenization, word similarity measuring (through word vectors), dependency parsing, sentence classification, POS, NER and many others. It does not provide software as a service, it is instead use to build powerful NLP pipelines for applications or Deep Learning pre-processing. It allows to perform the training of different included model (such as BERT) for different tasks. In this work, *spaCy* was used exclusively for NER. The official website is available at `https://spacy.io/`.

4.4 Running environment

In this subsection a little overview of the main tools used for the training of the model. The **MTSI-BERT** required at least 3 GB to be instantiated and up to 5 for the effective training. The memory consumption was optimized during the development, anyway the GPU memory resources remained out of the possibilities of a normal machine. For this reason, the training and the testing was done on two different platforms, described below.

HPC@POLITO

Computational resources were provided by HPC@POLITO, a project of Academic Computing within the Department of Control and Computer Engineering at the Politecnico di Torino (http://www.hpc.polito.it).
The *HPC@POLITO* is the high performance computing infrastructure hosted by the DAUIN (Deparment of Control and Computer Engineering) deparment at *Politecnico di Torino* university. The project started in 2008 with small hardware and open source software. In 2011 the resulting system, called CASPER, has come to life. In 2015 a new cluster, HACTAR, joins the infrastructure. HACTAR has GPUs capabilities and for this reason was the one chosen in this work, all the hardware specifics are shown in figure (4.1). In summer 2019 a new cluster, called LEGION, has come. This has a capabilities of 8x nVidia Tesla V100 SXM2 with 32 GB and 5120 cuda cores, which will help several future Deep Learning studies.
All the clusters owns a SLURM (`https://slurm.schedmd.com/overview.html`) scheduler, an open source software for Linux clusters management.
The training phase with HACTAR cluster required 25 hours on 100 epochs and 1 hour and an half on 20 epochs.
All the information about the HPC@POLITO are available at `http://hpc.polito.it/index.php`.

4.4.1 Google Colab

Google Colab (Colaboratory) is a free jupyter notebook that runs in the cloud, offered by *Google*. It offers the possibility to use computational resources for free, in particular a CPU, a NVIDIA TESLA K40 GPU and a TPU. While the TPU is not yet supported by PyTorch, the GPUs has the capability to run the **MTSI-BERT** model. The use of Colab was limited to testing only, due to time limitation (12 hours of continuos training) and some performance slowdown when the cloud was full. It allows to easily test the model, which otherwise would have to wait for the queue on HPC@POLITO. All the basic information about the service offered

Architecture	Linux Infiniband-QDR MIMD Distributed Shared-Memory Cluster
Node Interconnect	Infiniband QDR 40 Gb/s
Service Network	Gigabit Ethernet 1 Gb/s
CPU Model	2x Intel Xeon E5-2680 v3 2.50 GHz 12 cores
GPU Model	2x nVidia Tesla K40 - 12 GB - 2880 cuda cores
Sustained performance (Rmax)	**20.13 TFLOPS** (last update: june 2018)
Peak performance (Rpeak)	**25.61 TFLOPS** (last update: june 2018)
Computing Cores	696
Number of Nodes	29
Total RAM Memory	3.6 TB DDR4 REGISTERED ECC
OS	Centos 7.4.1708 - OpenHPC 1.3.4
Scheduler	SLURM 17.11.5

Figure 4.1: The HACTAR hardware specifics. Image take from HPC@POLITO website

by Google are available at the following url `https://colab.research.google.com/notebooks/welcome.ipynb`.

Chapter 5

Results

In this chapter the results of MTSI-BERT are presented. The chapter is divided in two sections, in the first a description of the results after the training phase, together with some considerations, are reported. The second one instead contains the results of the testing phase.

5.1 Training results

The training phase was repeated two times, once on 20 epochs and another one on 100 epochs, to see if the network needs more times to continue the loss decreasing.
During training the most important result to look at is the decreasing of the loss. Even if the loss decreasing is not always related to the final testing scores, this is an important parameter to take care of. The loss trend tells if the trained model is going to converge or not. The trend of the losses for the three tasks was promising and is reported in figure (5.1). The plot shows how the loss for the action decreases faster compared to the other two. The action could be the easier to detect for the network, considering that it is a binary classification task compared to the three possible classes for the intent. The end-of-session should be instead the most difficult task of the three and the achieved loss value confirms the good job done by the proposed model.
Another important thing is to compare the loss on the validation set with the one on the test set. The depth of Deep Learning models introduces so much parameters that they are, in principle, able to just memorize the training set. While this trend make the loss converging, it produces very bad performance on unseen samples, because of the inability of the model to abstract what it has seen. This phenomenon is also known as overfitting and is characterized by an increasing trend of the validation loss versus

a decreasing trend of the training one. MTSI-BERT avoided overfitting thanks to dropout and weight decay regularization techniques. The plot reporting the trends for both validation and test losses is shown in (5.1a). The plot shows how the validation and the training losses decrease together. Finally a table reporting the training time for the 20 and 100 epochs on single GPU is available at (5.1). The table reflects the number of parameters of the three MTSI-BERT variants. The Basic variant is the fastest one, the Deep is the slowest, while BiLSTM is a compromise between the two.

Architecture	**20 epochs**	**100 epochs**
MTSI-BERT Basic	2:00h	10:19h
MTSI-BERT BiLSTM	2:17h	11:21h
MTSI-BERT Deep + Residual	2:40h	11:35h

Table 5.1: Training time for the three MTSI-BERT variants on 20 and 100 epochs training.

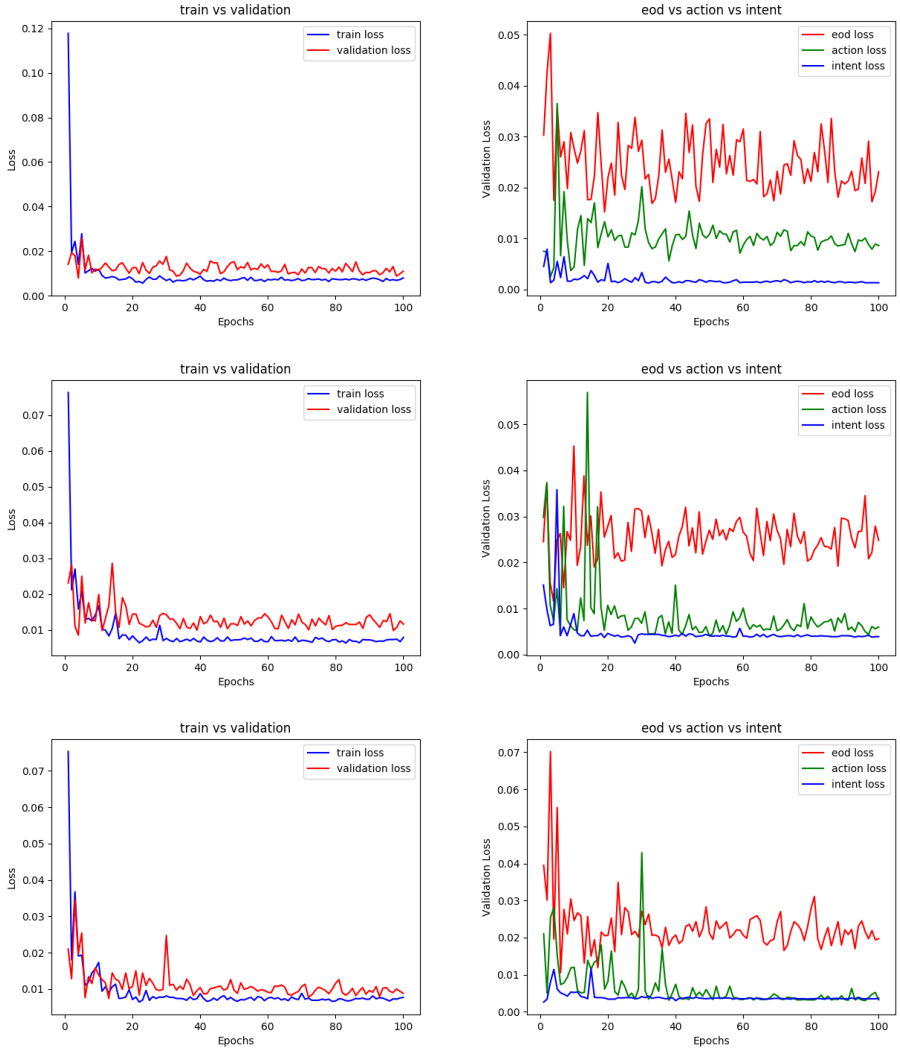

Figure 5.1: (a) losses trend for MTSI-BERT basic architecture. (b) losses trend for MTSI-BERT biLSTM architecture. (c) losses trend for MTSI-BERT deep + residual architecture.

5.2 Testing results

5.2.1 Measures

The testing phase is the most important in machine learning, is the moment in which the real value of a model is assessed. The way to measure the

performance of a model is today widely discussed, anyway the MTSI-BERT performance was tested with the traditional machine learning scores which are:

- precision
- recall
- F1

Before understanding what the three score measures actually mean, the confusion matrix has to be introduced. Figure (5.2) represents the confusion matrix. It contains four different entries for a binary classification task:

- True Positive (TP)
- False Positive (FP)
- True Negative (TN)
- False Negative (FN)

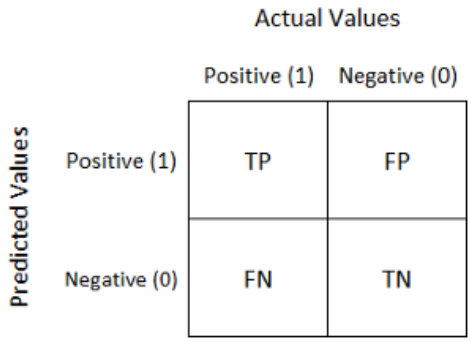

Figure 5.2: The confusion matrix. Source [12].

The entries beginning with the "True" word are all the ones correctly predicted by the model. The True Positive are the correctly predicted belonging to the positive class (label 1), the True Negative are instead the correctly predicted belonging to the negative class (label 0). The entries beginning with the "False" word are the mispredicted ones, the False Positive are the mispredicted belonging to the positive class (label 1), the False Negative are the mispredicted belonging to the negative class (label 0).
The precision is the ratio between the correctly classified samples and the total number of classified samples. Formula (5.1) represents the precision for the positive class. Unfortunately, the meaning of the precision score for unbalanced classes is quite ambiguous. The precision score will be high if the model simply predict the most frequent class without taking care of the less frequent one. For instance let's think about a binary classification problem where the positive class has support 99 and negative class has support 1. If the model predicts correctly only the positive ones, than it will have a

precision of 99% but no predictive power. So, in order to give a measure of the predictive power of a model, the recall concept has to be used.

$$Precision = \frac{TP}{TP + FP} \tag{5.1}$$

The recall measures the ability of the model to correctly predict the samples belonging to a class, given all the other samples of that class. Then it simply the ability of the model to "recall" such a class. Formula (5.2) shows the recall for the positive class. These metric is used in some scenarios in which the classes importance is not the same. For instance think about a patient cancer detector, the system has to predict at a first stage if a patient could have the cancer or not. If a cancer is predicted, then the patient has to be subject to additional human visits, if not it can go home. In such scenario a False Positive is not a great problem, since the misprediction will then be revealed with the further human visits. A False Negative has instead to be absolutely avoided, otherwise a patient with the cancer will be sent home without any further visit.

$$Recall = \frac{TP}{TP + FN} \tag{5.2}$$

The F1-score is a measure defined as function of precision and recall, balancing between the two. The F1 weighs the precision of the model with its recall. In this way is possible to give a meaningful measure with unbalanced classes. The formula for F1-score is reported in (5.3).

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{5.3}$$

While these metrics are defined for binary classification problems, the extension to multi-class is trivial by considering binary problems for each class c of the type: class c versus not class c. Once all the precision for each class have been computed, an average of them have to be made. There are three ways to compute the average of a measure in multi-class classification problems:

- Macro average
- Macro average
- Weighted average

With macro average the F1-score for each class is computed first, then all the scores are averaged. This type of average is used in case of unbalanced class, since it penalize more errors on the minority class. Micro average consists in computing the average by taking in consideration the total true positive, false positive and false negative, thus not favouring any particular

class. Weighted average is instead another option present in the *sklearn* library which computes the average of all the F1, computed individually for each label, by weighing each factor with the support of that class, thus favouring the majority class. The equations for the listed average methods are reported at (5.4). The scores for this work take in consideration the macro average only for precision, recall and F1, due to presence of unbalanced classes.

$$\begin{aligned} F1_{macro\_avg} &= F1_{class1} + F1_{class2} + F1_{class3} \\ F1_{micro\_avg} &= F1_{class1+class2+class3} \\ F1_{weighted\_avg} &= W_1 \cdot F1_{class1} + W_2 \cdot F1_{class2} + W_3 \cdot F1_{class3} \end{aligned} \tag{5.4}$$

5.2.2 Reference SOTA model

In order to have a performance comparison for *MTSI-BERT*, another model was developed. This model is similar to the proposed one except for two aspects:

- No joint tasks
- Non contextual embeddings

The model performs the tasks prediction by using three different disjoint networks: one for the end-of-session, one for the intent and another one for the action. The networks for the intent and action prediction are the same, except for the output layer. They are composed by a biLSTM which performs the sentence encoding, followed by a 3-layer feed forward neural network to classify the sentence. The end-of-session uses the same biLSTM architecture. The encodings for each utterance are then grouped by 3 and concatenated to create a window. The resulting windows, having size 3 times the single utterance embedding, are then fed to another 3-layer feed forward neural network to classify the end-of-sentence.
The embeddings for this model are the one provided in Spacy with 300-dimensionality. The sentence embedding has instead dimension 1536 (768 times 2 for the biLSTM), to match the embedding of MTSI-BERT model. Stop words were completely removed. A picture showing the architecture is shown in (5.3).

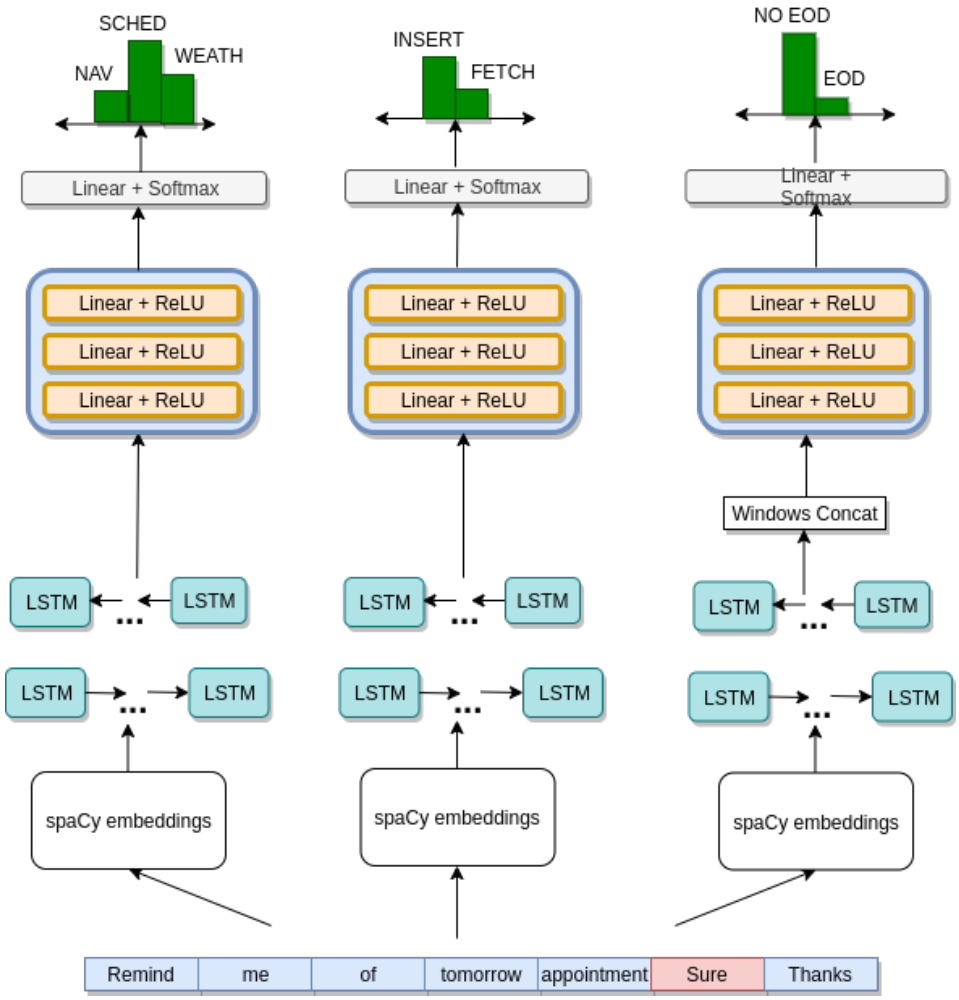

Figure 5.3: Reference SOTA architecture.

5.2.3 Model results

Finally in this section the results achieved by MTSI-BERT are reported. Before showing them a little premise have to be made. Since the end-of-session (EOS in the table) is computed between the current session and the first utterance of a random one, the final prediction will have a little randomness. To capture it and have a better vision of the final accuracy, the scores for this task were made by averaging the scores on 10 different and independent runs. Then the mean of the scores was made and the standard deviation computed. To better capture the stability and variability of the end-of-sentence accuracy, this is the only class presenting three precision digit when the standard deviation is in the order of 10^{-3}.
In both the training experiments (20 and 100 epochs) the training parameters are reported. In such tables, the milestones are the epochs where the learning rate is decreased, γ instead is the learning rate decreasing factor. The end-of-sentence prediction is three percentage point higher in MTSI-BERT, compared with the NO-BERT reference architecture. The action and intent tasks can be considered solved by all the three architectures, even if some of them shows better behaviour in some cases. The MTSI-BERT BiLSTM and MTSI-BERT Deep + Residual are the ones with the higher score on the EOS, anyway the second one shows the best loss trend of the three. The loss variance is smaller and so more stable. Since the model will be, in the future, deployed also in other scenarios, this robustness will make the deep architecture the preferred one.

20 epochs

Parameter	**Value**
mini-batch size	16
BERT learning rate	$5e^{-5}$
NN learning rate	$1e^{-3}$
weight decay	0.1
milestones	5, 10, 15, 20
γ	0.5

Table 5.2: Training settings for MTSI-BERT on 20 epochs.

Architecture	Task	Precision	Recall	F1
MTSI-BERT Basic	Intent	1.00	1.0	1.00
	Action	0.99	0.99	0.99
	EOS	0.981 ± 0.001	0.99	0.986 ± 0.001
MTSI-BERT BiLSTM	Intent	1.00	1.00	1.00
	Action	0.99	0.99	0.99
	EOS	0.99	0.995 ± 0.001	0.99
MTSI-BERT Deep	Intent	1.00	1.00	1.00
	Action	1.00	0.99	0.99
	EOS	0.98	0.99	0.99
NO-BERT	Intent	1.00	1.00	1.00
	Action	1.00	0.99	0.99
	EOS	0.961 ± 0.001	0.964 ± 0.001	0.962 ± 0.001

Table 5.3: Results for the MTSI-BERT on test set after a training of 20 epochs.

Architecture	Task	Precision	Recall	Accuracy
MTSI-BERT Basic	Intent	1.00	1.00	1.00
	Action	1.00	1.00	1.00
	EOS	0.98	0.99	0.99
MTSI-BERT BiLSTM	Intent	1.00	1.00	1.00
	Action	0.99	0.99	0.99
	EOS	0.99	0.992 ± 0.001	0.992 ± 0.001
MTSI-BERT Deep	Intent	0.99	1.00	0.99
	Action	1.00	1.00	1.00
	EOS	0.98	0.99	0.99
NO-BERT	Intent	1.00	1.00	1.00
	Action	0.99	1.00	0.99
	EOS	0.96	0.96	0.96

Table 5.4: Results for the MTSI-BERT on validation set after a training of 20 epochs.

100 epochs

Parameter	Value
mini-batch size	16
BERT learning rate	$5e^{-5}$
NN learning rate	$1e^{-3}$
weight decay	0.1
milestones	5, 10, 15, 20, 30, 40, 50, 75
γ	0.5

Table 5.5: Training settings for MTSI-BERT on 100 epochs.

Architecture	Task	Precision	Recall	F1
MTSI-BERT Basic	Intent	1.00	1.00	1.00
	Action	1.00	1.00	1.00
	EOS	0.988 ± 0.001	0.99	0.991 ± 0.001
MTSI-BERT BiLSTM	Intent	1.00	1.00	1.00
	Action	1.00	1.00	1.00
	EOS	0.99	0.997 ± 0.001	0.99
MTSI-BERT Deep	Intent	1.00	1.00	1.00
	Action	1.00	1.00	1.00
	EOS	0.99	1.00	0.99
NO-BERT	Intent	1.00	1.00	1.00
	Action	1.00	0.99	0.99
	EOS	0.955 ± 0.001	0.96	0.96

Table 5.6: Results for the MTSI-BERT on test set after a training of 100 epochs.

Architecture	Task	Precision	Recall	Accuracy
MTSI-BERT Basic	Intent	1.00	1.00	1.00
	Action	0.99	1.00	0.99
	EOS	0.99	0.99	0.991 ± 0.001
MTSI-BERT BiLSTM	Intent	1.00	1.00	1.00
	Action	1.00	1.00	1.00
	EOS	0.99	0.994 ± 0.001	0.99
MTSI-BERT Deep	Intent	1.00	1.00	1.00
	Action	1.00	1.00	1.00
	EOS	0.99	0.994 ± 0.001	0.99
NO-BERT	Intent	1.00	1.00	1.00
	Action	0.99	1.00	0.99
	EOS	0.961 ± 0.001	0.965 ± 0.001	0.963 ± 0.001

Table 5.7: Results for the MTSI-BERT on validation set after a training of 100 epochs.

Chapter 6

Use case

In this chapter a real case scenario is studied. *PuffBot* is a chatbot for asthma suffering users to support doctors in daily care of their patients. The chatbot will rely on a multi-turn paradigm to correctly understand user healt status while it will also exploit *Convology*, an ontology specific for multi-turn conversational agent. While the project is still in the first phase of development, a brief overview of it is given. The next sections will present the PuffBot assistant together with a use case scenario, a first prototype using rapid-prototyping tools and the Convology ontology.

6.1 PuffBot

PuffBot is a multi-platform chatbot born from an idea of two Italian researchers belongs to *LINKS Foundation* and *Fondazione Bruno Kessler*. The aim of PuffBot is to monitor the status of the patients through natural conversations via text and voice. The multi-platform implementation allows the user to interact with the bot via several devices like Telegram, Amazon Alexa etc.
PuffBot is designed specifically for helping doctors in the support of its asthma suffering patients by creating an empathic channel with them, especially for children. It ask periodically for the health status of the user by monitoring its breath, how many "puffs" he has done in the last period or for any particular emergency situation he encountered recently. The patient can also interact for first with PuffBot. It will extract all the relevant information from the user dialogue to make a little resume of his situation. PuffBot relies on a knowledge-base based on Convology, an ontology specific for multi-turn interactions. With Convology, PuffBot can store all the information regarding the user status and suggest him treatments specific for its conditions. The treatments PuffBot suggests are all defined by domain

expert of the Trentino ASL and cover only light cases, for emergency one the visit of the doctor is always suggested. PuffBot classifies the user status, by means of reasoning processes on the knowledge-base, in 4 different categories. The green zone is the one associated with the normal situation. Events associated to this category are, for instance, related to absence of cough or cough during sport activities. The typical PuffBot suggestion will be to continue with the current therapy. The yellow zone is associated with mild cases, for which doctor intervention is typically not needed. Events falling into this category are cough with rhinitis, cough during night and other types of coughs. In this case PuffBot suggest the user to increase the frequency of the puffs and will monitor again the situation after a while, if the conditions worsen then the patients will be classified in the red zone. The orange zone is the last PuffBot can handle and is the one that needs the most attention, together with the red zone. This zone typically consists in persistence cough and sense of chest constriction. PuffBot will suggest to increase the frequency of puffs and to follow a particular therapy, of course prescribed by domain experts. PuffBot continues to monitor the patients to see if condition improves or not, in the worst case it classifies the user with the red zone. The red zone is the one consisting in a emergency situation for which the doctor visit is needed. PuffBot will suggest the user to call a doctor and will continue to monitor the status of the user after the doctor visit.
PuffBot monitoring is triggered by particular external situations like humidity changing, pollen in the air or detection of high values of pollution in the city where the patient lives. The PuffBot conversational paradigm is a multi-turn. Each session is designed to identify a single user intent like the breath condition, cough, sport activities or particular allergies. All the information extracted from each single session are then inserted in the knowledge base together with some important parameters such as date and hour. PuffBot interactions are divided in two main phases: on-boarding and monitoring. The on-boarding phase starts when the user interact with the bot for the first time, thus a set of question related to the name, the city, sport practiced and eventual allergies are made to him in order to create an user profile to be saved on a database. These informations are then used by PuffBot to monitor the weather and humidity level for that city or to take care about particular user allergies. The second phase instead is the real monitoring discussed above. A little prototype of PuffBot has been done using rapid prototyping techniques as *DialogFlow* and the *Alexa SDK* released by *Amazon*.
In the following subsections a use case description and the a little overview of PuffBot together with a proposed infrastructure architecture is done.

The last section instead contains a brief description of Convology.

6.1.1 Use Case Scenario

Martin is a university student in Turin. In the last month he suffered from cough and thoracic compression during the whole day. The doctor prescribes him a pulmonologist visit. With the visit, Martin discovered his asthma due to a pollen allergy. The doctor prescribes a therapy, gives him his inhalator and suggest an application called *PuffBot*, an intelligent assistant capable of supporting him with diagnostic of his situation. Martin starts the bot on *Telegram* and the skill on *Amazon Alexa*. PuffBot begins the conversation by presenting itself and asks some personal information about Martin. It asks for his name, the city where he lives, sport activities he practices and eventual allergy it has. Finally PuffBot saves all the Martin information on the database and creates the user profile.
Martin interacts with PuffBot twice a week, by updating it with new events related to his asthma. He interacts with him both at home, using the *Alexa smart speaker*, and when he is out with friends, using the *Telegram* chat. PuffBot frequently asks him about breath condition and, in case of anomalies, suggest him to continue daily the therapy and try to increase the puffs frequency. In the month of April, Puffbot, knowing the pollen allergy of Martin and the pollen quantity in the air of Turin, interacts with him to know if everything is fine, if the asthma is under control and if he suffered from cough during the night. He warns Martin about the high percentage of pollen in the air and suggest him to always carry the inhalator with him and try to increase the puffs to 4-6 times per day, whenever is needed. After a week Martin says that its situation is getting worse, he has dry cough during all night and this make difficult for him to sleep. PuffBot consults all the data it has about the Martin situation and understand that this anomaly could be related with the high percentage of pollen in the air and the great humidity in the city of Turin during the last 2 days. It extracts from his domain knowledge, given by Trentino ASL experts, a possible therapy consisting in increasing the puffs to 4-6 hours daily and, if the condition improves, then reduce the puffs to 8-10 hours per day. Martin follows the instructions for the next 3 days and the situation seems to improve, so it decreases the puffs frequency as suggested. After 5 days it encounters new problems not only with the cough but he also feels pain during the breath and a sense of thoracic constriction. PuffBot catches this new information and makes a new diagnosis, based on the previous state of Martin and all his precedent history of the last 2 months. PuffBot extracts a new therapy consisting in increase the puffs to 2 per hours, for the first 2 hours, then 3 every 4-6 hours. PuffBot also notices the possible

seriousness of the situation and invite Martin to immediately talk to it if the situation worsens. Martin follows the instruction of PuffBot but the situation does not seem to improve, thus he talks with PuffBot the next day which suggests to use the puffs for time intervals smaller than 3 hours while calling a doctor as soon as possible. Martin calls the doctor and makes an appointment for the afternoon of the same day. After the visit, PuffBot asks him for the doctor diagnosis and saves the new therapy in order to continue the monitoring of Martin.
Martin is happy with the bot support during the day and refers it to the doctor. The doctor is helped by PuffBot in the care of its patients and continue to suggest it to all its asthma suffering patients.

6.1.2 Rapid prototyping

Rapid prototyping tools for chatbot development are tools that helps in the creation of a conversational agent, without caring about machine learning algorithms. The are commonly offered as web services hosted by companies which allow the use of their intelligent algorithm to support the development. *DialogFlow* and *Alexa SDK* are ones of the most famous rapid-prototyping services freely available. In the next sub sections a little overview of both is made.

DialogFlow

DialogFlow is a web platform for make easier the chatbot development (url: `https://dialogflow.com`). It consists in a dashboard where the user can create a new project or open a existing one. DialogFlow is empowered by the *Google* machine learning techniques and facilitate the design of multi-platform chatbot. With a single click it is possible to activate the porting of our project to different chat platform like Telegram, Messenger, Slack, Google Assistant and Amazon Alexa. DialogFlow is based on webhooks which are called each time a particular intent is detected. The developer has to insert some possible user utterances for each intent by high lightning the slots that have to be detected. Thus it has to write the webhook to be called when that particular intent is recognized. The webhook receives a JSON file containing all the useful information for the detected intent (Slots, original utterance etc.). With the Blaze plan of Firebase (pay as you go), it is possible to call an external service from the webhook, if the latter is hosted on a server having a non-self signed certificate. This last detail allow the use of DialogFlow Natural Language Understanding as an interface for external applications. DialogFlow makes possible also to handle a sort of multi-turn conversation through the mechanism of contexts. A context is a

particular variable that can be passed from an utterance to another. The maximum number of contexts is anyway limited to 5. DialogFlow provides also a speech-to-text and text-to-speech systems for the deployment on a voice assistant like Google Assistant and Alexa. Despite this, the conversion of the project to an Alexa skill is today very poor.
The DialogFlow platform was the selected one for the development of the first PuffBot prototype for presentation purposes.

Alexa SDK

Alexa SDK is the Software Development Kit for the development of skills on *Amazon Alexa* (url: `https://developer.amazon.com`). A skill is a sort of little program for the Alexa speaker to handle particular type of conversations, enabling so the speaker customization. The skill can be developed using the Amazon dashboard web service, which is very similar to the DialogFlow one and easy to use. Despite the design similarity, the tools provided by Amazon Alexa seems to be very poor compared to the one offered by the Google service. It follows the same paradigm of DialogFlow, the developer has to insert some example of user utterances for each intent and then develop the correspondent webhook to be called. It offers compatibility with Amazon products like Echo Show but not with external services. The Italian version still does not support the fallback intent. The fallback intent is the intent triggered when a user utterance is not recognized, its absence make useless the development of a skill considering that Alexa will detect the closest intent even if a total random utterance was said. Nevertheless a small prototype was done and deployed on an Alexa speaker.

6.1.3 Proposed infrastructure

The first phase of PuffBot development consisted in the feasibility study. This study takes in consideration the possible infrastructure to host the chatbot web-service. The working team is composed by 3 people having skill covering NLU and knowledge representations. The development of a multi-platform system, together with the voice system, is out of the scope of this work. For this reason two main issues must be solved by using external services:

1. Multi-platform integration
2. Voice systems, which includes both speech-to-text and text-to-speech

As already explained in section 6.1.2, DialogFlow provides multi-platform support with just a few clicks together with a reliable STT and TTS systems. What has to be developed internally is the domain Natural Language

Understanding module, whose is the core part of the project. An infrastructure proposal for the described web-service consists in the use of DialogFlow for redirection, via webhook, of the processed speech to the internal server, containing the NLU module. By using DialogFlow as a proxy, the module can simply work on the NLU task, while all the other pre-processing (SST) or post-processing (TTS) are made by the DialogFlow framework. This solution gives also the compatibility with multiple messaging applications without the need of changing the internal logic. All the compatibility issues are moved to DialogFlow. The drawback is the need for having a non self-signed certificate for the server, which has to be addressed from DialogFlow webhooks.
In the next sub-section the specific service integration with Telegram application is described.

Telegram integration

Telegram is a messaging application available for Android, iOS, Windows Phone, Windows, Linux and macOS. PuffBot will be hosted on the internal server and will use DialogFlow for the integration of Telegram. The Telegram integration in DialogFlow consists in the creation of a bot for the chat. A Telegram bot is a special chat agent programmed to accomplish a specific task. A bot is easy to create on the Telegram platform, in a way that everyone can use it. Each bot has its own unique client access token, used to contact it. To start the conversation, the user has to search for the bot, open the chat and initiate the bot session with the "*/start*" command. Each time the user sends a message to the bot, this will be redirected to DialogFlow. A Telegram message, in DialogFlow, is a JSON payload containing all the information which characterize that message. The JSON file contains the message ID, the chat ID, the timestamp and other information as reported in figure (6.1). The chat ID is the one identifying the user. The first time the user starts the bot, the chat ID will be stored in the internal server database and will then be enriched with the user personal information. By doing so, PuffBot has a way to contact the user when a particular event occurs (doctor appointment, weather changing, pollen detected in the air). In DialogFlow there are two main ways to trigger an intent.

1. by sending an user query (for that particular intent) to the bot
2. by linking an event to the intent and then triggering such event

while the first one is the common way to trigger an intent and can be done also by *POST request* (by sending directly the query utterance in the JSON payload), the second is the way chosen in this work for the triggering. By linking an event to an intent, the latter can be triggered to invoke the former. For instance, when the Telegram start command is sent,

Figure 6.1: An example of DialogFlow JSON payload for a Telegram message.

DialogFlow triggers the TELEGRAM_WELCOME event which is by default associated to that particular command. In figure (6.2) the manual

```
curl -H "Authorization: Bearer <CLIENT_ACCESS_TOKEN>"
"https://api.dialogflow.com/v1/query?v=<API_VERSION>&e=<E
VENT>&timezone=Europe/Paris&lang=it&sessionId=1234567890"
```

Figure 6.2: How to manually trigger an event, and so the associated intent.

triggering of the TELEGRAM_WELCOME event is reported. Where the <CLIENT_ACCESS_TOKEN> is the one of the bot to contact (can be retrieved through Telegram), the <API_VERSION> corresponds with the DialogFlow API version to use (available versions can be found on the docs) and the <EVENT> is the name of the event to trigger. When a particular event is triggered (like the weather changing), PuffBot has then to contact the user. To do so, the Telegram *REST API* can be used. Figure (6.3)

shown the way to manually send a message, where the TELEGRAM_TOKEN is the token identifying the telegram bot, the CHAT_ID is the ID of the chat on which the message has to be sent (it corresponds to the user ID if the group is private) and the TEXT_TO_SENT is the formatted string to be sent as message. The TELEGRAM_TOKEN can be found in the integration tab of DialogFlow console, by clicking on the telegram icon.

```
curl
"https://api.telegram.org/bot<TELEGRAM_TOKEN>/sendMessage
?chat_id=<CHAT_ID>&text=<TEXT_TO_SENT>"
```

Figure 6.3: How to use the Telegram *REST API* to send a message to a particular user.

Finally the triggering mechanism allow PuffBot to contact the user when a particular event occurred. By storing all the chat id at server side, it is possible to send a message to a particular user with the Telegram *REST API*. The *send_message* is called inside the webhook correspondent to a particular intent. This intent is then associated to an event which is triggered every time is needed.
Using DialogFlow as a proxy removes the needing for implementing a Telegram interface for PuffBot service. The only thing needed is a non self-signed certificate for the server, in order to be called via API from the DialogFlow webhook.
A final schematic representation of the Telegram integration is available in figure (6.4).

The prototype developed for Telegram is available and presents the two phases of on-boarding and first diagnosis. The current list of intents implemented in PuffBot is almost 40, 12 of them are part of the on-boarding phase. All the intents were defined with the supervision of the Trentino ASL, to better model the domain specific information. Furthermore, intents were structured in a hierarchical fashion, in order to group together similar intents. For instance, several sub-intents are related to the macro-intent "cough", some of them are "cough frequency", "last cough episode" and so on. A conversation sample is shown in figure (6.5). Unfortunately for the English speakers, the prototype was developed in Italian only. Anyway the project is still work in progress and English and Chinese porting are coming. The picture shows the on-boarding phase, consisting in asking basic information about the user. Here were reported questions about name, city, weather of the city, practiced sports, smoking habit, work place information and if the user often suffers from colds. Some important information, such as the city weather, are in project to be retrieved through third party APIs.

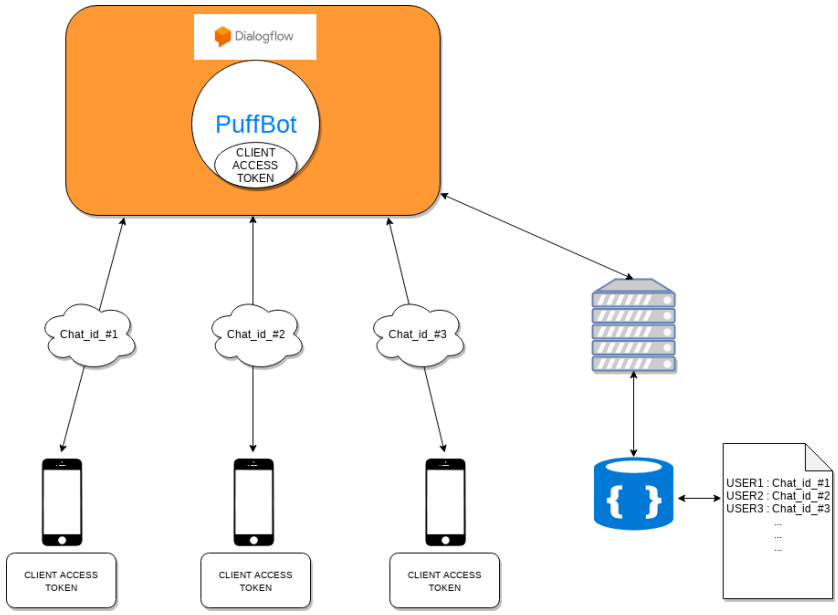

Figure 6.4: Proposed web-service infrastructure for Telegram integration.

The second phase instead consists in asking questions about the cough, for which the user has suffered in the last period. At the end of the diagnosis, PuffBot resumes all the symptoms in the last message, together with a little suggestion, consisting of, in this particular case, a little walk in the park.

6.2 Convology

Convology (CONVersational OntOLOGY) is an ontology developed by *Mauro Dragoni* at Fondazione Bruno Kessler. Convology is a top level ontology which aim is to model conversation for building knowledge-bases as support of multi-turn conversational agents. It was developed to be highly reusable and is downloadable at the following url `https://perkapp.fbk.eu/convology/`.
The building process of Convology followed the METHONTOLOGY [32], which is composed by seven stages:

1. Specification
2. Knowledge acquisition
3. Conceptualization
4. Integration
5. Implementation
6. Evaluation

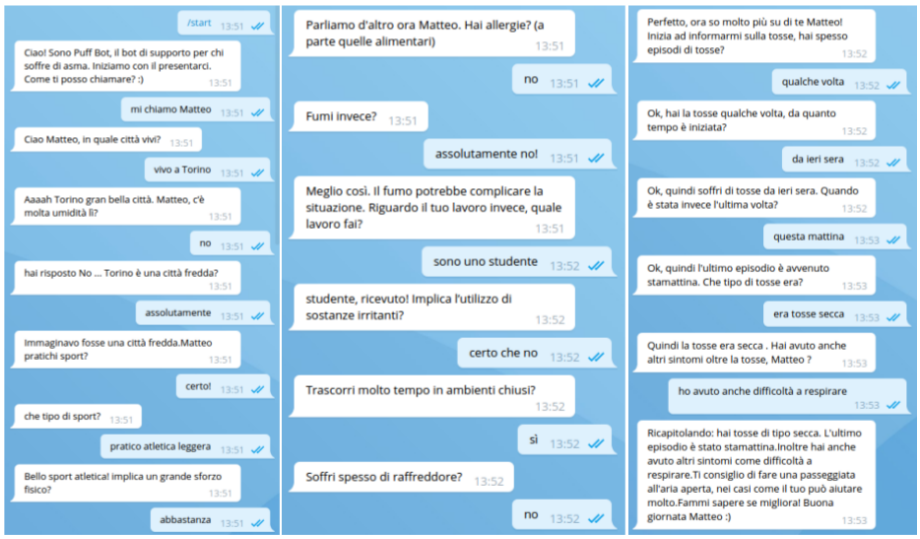

Figure 6.5: A sample Italian conversation with the Telegram first prototype of PuffBot.

7. Documentation

The process involves a total number of four knowledge engineers and two domain experts from the Trentino ASL. Convology is a meta-model to describe the conversation from the agent point of view. By using it, is possible to create conversational agents able to access a knowledge base through a multi-turn conversational paradigm.
The ontology consists in five top-level concepts:

- Actor
- ConversationItem
- Dialog
- Event
- Status

The **Dialog** concept is instantiated each time a new multi-turn interaction, between user and one or more *Agent*, is started. It is the only concept in Convology that does not subsume any other. It has the *hasId* property allowing an efficient retrieval of the dialog during reasoning time.
The **Actor** is the concept defines the party role inside a conversation. Two party roles are available: *User* and *Agent*. An *User* role identifies the user of the conversation. An *User* concept is created whenever a new user starts the conversation with an *Agent*. Each *User* can be associated with different instances of *Dialog* and *Agent*. The *Agent* role identifies the conversational agent instead. Convology can has different instances of *Agent* even if the

application is the same. This can improve the performance in the case of PuffBot, for instance, where we can have different instances of *Agent* with different user statuses.
The **ConversationItem** represents the entities of the conversation. Convology presents four types of ConversationItem:

- Question
- Intent
- Feedback
- DialogAction

Question represents a question sent from the User to an Agent. It can also be associated with the *UserEvent* through the *hasTriggerQuestion* property. *Intent* represents the intent detected by a NLU module for a particular Question. When the Intent is detected, a *StatusItem* instance is created to allow the inference of the user status in PuffBot. *Feedback* represents a simple utterance from the User to the Agent which not requires a reply and is not associated with an Intent. The *DialogAction* instead is associated with Question or Feedback and represents the next conversational action to perform. These type of concepts are typically defined by domain experts.
Event represents an event that can occur during the conversation, in this way reasoning processes can be triggered. Convology presents three different kinds of events:

- EventQuestion
- EventAnswer
- UserEvent

EventQuestion notifies the submission of a Question from a particular Actor. Instances of this type are associated with *hasTimestamp* property, the *Actor* instance that receives the question and the one which has sent the question. The *EventAnswer* represents an answer provided by a specific *Actor* at a specific timestamp (*hasTimestamp* property). The *UserEvent* represents an event related to a specific user, for instance the detection of a particular intent.
Status represents the relevant status of the user. The possible user statuses are defined by domain experts. A complete overview of Convology is shown in figure (6.6).

6.2.1 Convology in PuffBot

The main goal of PuffBot is to perform a real-time inference of the *UserStatus* through a set of *Questions*. Since multi-turn conversation are hard to handle, in order to make possible to track different session simultaneously, the concept of *Dialog* were exploited, each of them identified by Convology

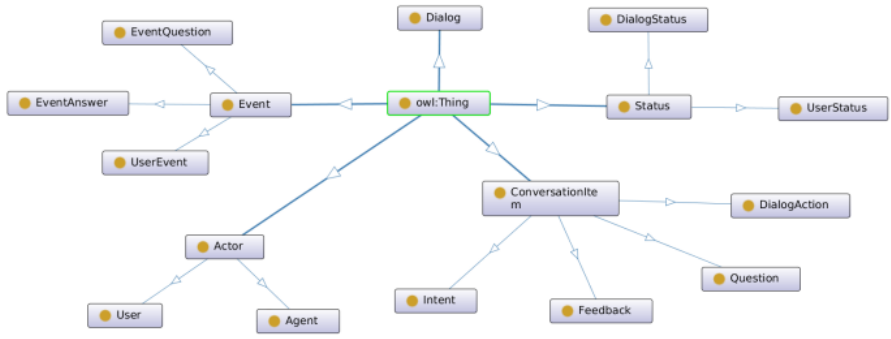

Figure 6.6: A hierarchical view of Convology.

with a unique identifier. In this way a fast and easy data retrieval is possible during reasoning time. When PuffBot detects a known intent, an *Intent* instance is created inside the knowledge base, which triggers the reasoner to decide the next *DialogAction* to perform. For example, a possible action is to ask a further question to the user, in order to better understand, with higher accuracy, the *UserStatus*. Once PuffBot classifies the status of the user with a certain confidence, the reasoner triggers the dispatch of an advice to the user, consisting in a status summary. This advice is typically an instance of the *Feedback* concept.
Figure (6.7) briefly describes an example of the reasoning process with Convology. The red circles represents the detection of *UserEvent* concepts triggered by the user answers. Each *UserEvent* is then linked to the correspondent *Intent* recognized by the Natural Language Understanding module (which is not part of Convology). The link (green arrows) between an *UserEvent* and and *Intent* is the *hasRelevantIntent* property of the *UserEvent*. The tables in the right part of the image represents the *UserStatus* which are LowRisk, MediumRisk and HighRisk. Each *UserStatus* is characterized by a series of symptomps, defined by domain experts. Each detected *Intent* is related to a particular symptom through the *activate* object property. Finally the SPARQL-based reasoner starts and tries to infer the status of the user. If the information are not enough, the reasoner tries to infer the next *DialogAction* to continue the conversation.

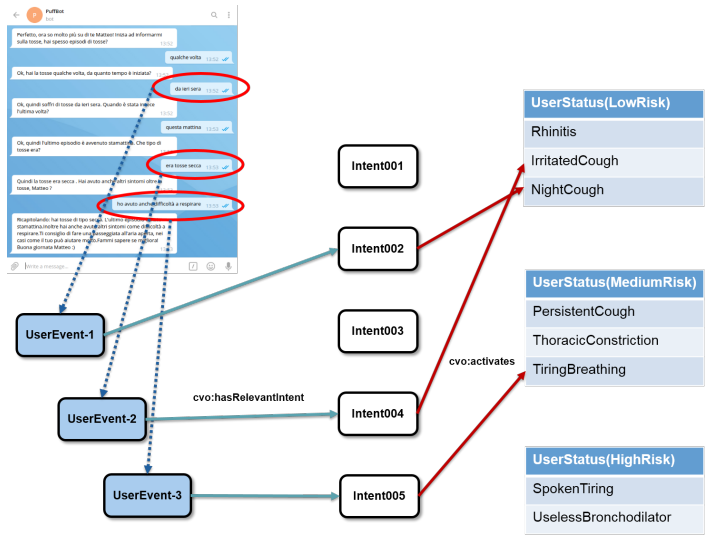

Figure 6.7: Reasoning example in Convology.

Chapter 7

Conclusions

In this work, it is proposed an overview of the major Deep Learning techniques in modern Natural Language Processing. Recurrent Neural Networks are going, by now, to be replaced by attention based models like *Transformer*. New models like *BERT* start a new era of *Transfer Learning* comparable to the one Computer Vision had in the past 5 years. This new trend opens different scenarios where voice search technologies will replace most of today textual searches. To empowering this new technology, a great breakthrough in the field of *Natural Language Understanding* has to be done. The scenario taken into consideration in this work is the one involving a multi-turn conversational paradigm. Such paradigm consists in a simulation of a real conversation between humans, where different utterances can be exchanged in order to fulfill a desired goal. Such paradigm is not trivial to handle because it requires to solve a lot of open questions for the understanding field. For instance the coreference resolution have to be done easily, together with the carry of the dialogue context through the conversation.
To analyze this paradigm from a more practical perspective, some approximations of the real world were done. The entire conversation was divided in subsections called sessions. Each session is characterized by a single intent to be fulfilled through a single action on a knowledge base, a structured representations for data which enables dialogue tracking, fast data retrieval and reasoning processes. Then a model to classify these sessions was proposed. *MTSI-BERT* is a joint model, based on BERT, for intent and action classification within a dialogue session and end-of-session detection. Such model have reached very good results on *KVRET* dataset, a dataset containing some dialogues between a driver and an integrated car assistant. Thus providing a fertile soil for further studies. Finally an use case was described. *PuffBot* is a chatbot for helping and monitoring asthma suffering

children. It is based on a multi-turn scenario and it has a knowledge base for saving all the status information about the patient, in order to infer its emergency code and support him accordingly. A little prototype and an architecture proposal to make it work on Telegram have been made.
This work was only the starting point for my studies in this new and exciting area. Future studies will cover other aspects of *Natural Language Understanding* field. MTSI-BERT can be improved by integrating, together with the other already studied tasks, the *Name Entity Recognition*, actually performed via *spaCy*. Many open questions are still opened for which this work did not have the presumption to answer. This work not focused on the way to correctly formulate a response for the user. The session classification was done by assuming correct agent response. Anyway wrong agent replies will affect the way this model performs and could lead to unwanted system behaviour. The agent reply must not only be grammatically correct, but it needs to be also semantically coherent. A way to generate a natural language text given some constraints (e.g. the knowledge base features extracted after the reasoning process) has already to be studied. A dialogue tracker system has also to be implemented. Another core problem in such scenario is to find a way, for the agent, to carry on the conversation dynamically if some information is still missing. This maybe could be the most important module to design.
A lot of work still have to be done, in order to allow Conversational AI to handle goal-oriented multi-turn conversations, anyway, as always, the best has yet to come.

Glossary

Artifial Neural Networks A family of architectures consisting in several logistic regression that works as a chain. They are inspired by human neurons behaviours..

Artificial Intelligence A family of techniques aim to simulate human intelligence to produce smarter algorithm..

BERT A huge stack of Transformer encoders trained on MLM and next sentence prediction tasks by Google AI. It produces contextual embedding task and it is used for fine-tuning tasks..

Deep Learning A family of machine learning algorithm raised in the last years. They are based on deep neural network architectures. These architecture can reach even more than 100 layers and requires great computational resources..

Encoder-Decoder A deep learning architecture for sequence-to-sequence composed by two fundamental modules. The encoder encodes the source sequence at a more abstract way. The decoder produces the target sequence starting from the representation produced by the encoder..

Language Model A NLP task aiming at predicting the next word in a sequence, given all the preceding ones..

Machine Learning A types of algorithms belonging to AI family which aim is to allow machines to learn a task, without explicit programming them..

Machine Neural Translation The task of translating a sentence from a source language to a target one by exploiting neural network techniques..

Masked Language Model A variant of LM whose aim is to predict a word inside a sequence taking in consideration both the right and the left context..

n-gram An n-gram is a contiguous and ordered sequence of n items from a bigger sample of text..

Natural Language Understanding An NLP task whose aim is to make

machines able to understand human language..

NLP Natural Language Processing is a field of AI aiming to give to machines the ability to work with human language..

Recurrent Neural Networks A family of neural networks architectures aiming to model time sequences. They exploit both the current input and the previous history..

Sequence-to-Sequence The task of associating a sequence of objects to another sequence of objects. An example could the machine translation..

Transformer A encoder-decoder architecture based on attention mechanisms, without any recurrent unit. It was originally introduced for translation tasks..

Bibliography

[1] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[2] Mihail Eric. Kvret website - `https://nlp.stanford.edu/blog/a-new-multi-turn-multi-domain-task-oriented-dialogue-dataset/`.

[3] Bill MacCartney. Understanding natural language understanding - `"https://nlp.stanford.edu/~wcmac/papers/20140716-UNLU.pdf"`.

[4] Saurabh Rathor. Simple rnn vs gru vs lstm :- difference lies in more flexible control (medium).

[5] Simeon Kostadinov. Understanding encoder-decoder sequence to sequence model (medium).

[6] Neural machine translation with attention - `"https://www.tensorflow.org/beta/tutorials/text/nmt_with_attention"`.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[8] Jay Alammar. The illustrated transformer - `"https://jalammar.github.io/illustrated-transformer"`.

[9] Lilian Weng. Attention? attention! - `"https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html"`.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[11] Lilian Weng. Generalized language models: Bert & openai gpt-2 - `"https://www.topbots.com/generalized-language-models-bert-openai-gpt2/"`.

[12] Sarang Narkhede. Understanding confusion matrix - `"https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62"`.

[13] Alan M Turing. Computing machinery and intelligence. In *Parsing the Turing Test*, pages 23–65. Springer, 2009.

[14] John Searle. Chinese room argument, the. *Encyclopedia of cognitive science*, 2006.

[15] P Russel Norvig and S Artificial Intelligence. *A modern approach*. Prentice Hall, 2002.

[16] Joseph Weizenbaum et al. Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966.

[17] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[18] C Newton. Google's knowledge graph tripled in size in seven months.

[19] Terry Winograd. Procedures as a representation for data in a computer program for understanding natural language. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1971.

[20] Cynthia Matuszek, Nicholas FitzGerald, Luke Zettlemoyer, Liefeng Bo, and Dieter Fox. A joint model of language and perception for grounded attribute learning. *arXiv preprint arXiv:1206.6423*, 2012.

[21] Yun-Nung Chen, Dilek Hakkani-Tür, Gökhan Tür, Jianfeng Gao, and Li Deng. End-to-end memory networks with knowledge carryover for multi-turn spoken language understanding. In *Interspeech*, pages 3245–3249, 2016.

[22] Martino Mensio, Giuseppe Rizzo, and Maurizio Morisio. Multi-turn qa: A rnn contextual approach to intent classification for goal-oriented systems. In *Companion Proceedings of the The Web Conference 2018*, pages 1075–1080. International World Wide Web Conferences Steering Committee, 2018.

[23] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[24] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[25] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 440–449. ACM, 1992.

[26] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty

of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013.

[27] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[28] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[29] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[31] Mihail Eric and Christopher D Manning. Key-value retrieval networks for task-oriented dialogue. *arXiv preprint arXiv:1705.05414*, 2017.

[32] Mariano Fernández-López, Asunción Gómez-Pérez, and Natalia Juristo. Methontology: from ontological art towards ontological engineering. 1997.