POLITECNICO DI MILANO
School of Industrial and Information Engineering
Department of Electronics, Information and Bioengineering
Master of Science Degree in Computer Science and Engineering

State of art and best practices in Generative Conversational AI

AI & R Lab
Laboratorio di Intelligenza Artificiale
e Robotica del Politecnico di Milano

Supervisor: Prof. Matteo Matteucci
Co-Supervisor: Patrick Ehlen

Author:
Alberto Mario Pirovano, 852479

Accademic Year 2017-2018

Alla mia famiglia, la migliore squadra di cui si possa fare parte.

Abstract

A chatbot is a software agent with the objective of helping humans to solve problems by using natural language to interact with them. This thesis has both the goal of investigating the most interesting methods to build such kind of models and of showing their application in a real-world use case. This work aims at describing these methodologies both from a theoretical and a practical point of view, taking as an example a technical case study coming from a project developed together with *Loop AI Labs*. In particular, we explain how to deal with a real-world *Conversational AI* problem, from data exploration to data preprocessing, normalization and standardization. After these preliminary steps, we explain how to choose, train and validate a model in a real-world setting.

We describe the two most successful architectures, retrieval and generative models, and after showing their pros and cons, we select the second family and we discuss how its most problematic issue is the integration of the conversational context. We select a model that can help to solve this problem and we quantitatively compare a *curriculum* learning, Bengio et al (2009) [6], based approach powered by a dedicated learning rate decay, with a *traditional* approach based on the learning rate decay proposed by Vaswani et al (2017) [100]. After this phase, we qualitatively evaluate the best performing model. In the quantitative analysis we present how the performance changes by varying some parameters of the model, while in the qualitative study we show the actual results that the best model obtains in two settings: *off-line*, where we evaluate a set of cases coming from an evaluation set, and *on-line*, where we inspect how the model behaves during a chat with a real client.

Sommario in Italiano

I chatbot sono software che hanno lo scopo di utilizzare il linguaggio naturale per guidare gli utenti nella risoluzione di problemi tecnici. Intuitivamente, le aziende che offrono un servizio di assistenza clienti tramite chat generano una grande quantità di dati sotto forma di conversazioni tra agenti reali e clienti. Registrare queste conversazioni apre una grande possibilità di creare valore. Questi dati possono infatti essere usati per costruire i chatbot, agenti di supporto virtuali che permettono agli agenti reali di lavorare più velocemente facendo meno fatica.

È in questo ambito che si colloca la *IA conversazionale*, la scienza che studia come utilizzare l'intelligenza artificiale e gli storici delle conversazioni dei centri di assistenza per costruire i chatbot.

Questa tesi ha il duplice obbiettivo di proporre una analisi che confronti i più recenti e innovativi metodi che la letteratura propone per costruire questi assistenti virtuali, e di mostrare come questi software si comportano in un caso reale. In particolare, dopo aver analizzato le due famiglie di architetture maggiormente utilizzate, gli algoritmi generativi e quelli di recupero, selezioniamo la prima e spieghiamo come il principale problema di questi modelli sia renderli consapevoli del contesto della conversazione. In seguito, proponiamo una validazione sperimentale nella quale confrontiamo quantitativamente un approccio basato sull'apprendimento a curriculum, basato sul Bengio et al (2009) [6], con un approccio più tradizionale che utilizza alcune intuizioni proposte da Vaswani et al (2017) [100].

Una volta selezionata una architettura appropriata, spieghiamo come affrontare un problema reale di *IA conversazionale* descrivendo le fasi che precedono l'effettiva creazione del modello, vale a dire quelle di esplorazione, di standardizzazione e di normalizzazione dei dati. Nell' ambito di questo caso di studio presentiamo una analisi quantitativa nella quale selezioniamo un insieme di metriche e mostriamo come il loro valore cambi al variare di alcuni parametri del modello. Nella analisi qualitativa invece studiamo il comportamento che il modello quantitativamente migliore presenta in due situazioni: una prima nella quale valutiamo come il software risponde ad una serie di casi provenienti da un set di validazione, e una seconda dove analizziamo le risposte generate durante simulazioni di conversazioni con i clienti.

Ringraziamenti

Ringrazio la mia famiglia, Susanna, Tullio e Federico, per il supporto costante e l'amore incondizionato. Mi hanno insegnato i valori che sono alla base di una vita onesta, la cultura del lavoro, la cura dei particolari, la capacità di crearsi nuovi obbiettivi e la voglia di mettersi in discussione. Mi hanno trasmesso insegnamenti che porterò per sempre nel cuore come la lealtà, l'avere fiducia in se stessi e il rispetto per gli altri. Non potrò mai ringraziarvi abbastanza.

Grazie anche ai miei parenti per l'affetto che mi hanno dato e per i valori che mi hanno trasmesso con la naturalezza di chi da senza chiedere niente in cambio. Ognuno di loro mi ha insegnato qualcosa di diverso e ha contribuito a costruire la mia personalità.

Un ringraziamento particolare va a mio cugino Alessandro, un amico vero con il quale ho condiviso ogni snodo della mia vita, dai miei primi passi fino alla laurea.

Un pensiero va a tutti i miei amici e compagni universitari, persone con le quali ho trascorso momenti indimenticabili e straordinari. Li ringrazio perchè mi hanno sempre stimolato nel pormi obbiettivi sempre più competitivi e difficili.

Grazie, infine, a tutto il team di Loop AI Labs, specialmente Gianmauro Calafiore, per l'incredibile esperienza vissuta insieme. Per il supporto tecnico ringrazio il professor Matteo Matteucci e il mio supervisor Patrick Ehlen.

Contents

Chapter 1

Introduction

Humans use natural language to communicate. Among the vast variety of tasks communication is used for, one of them is finding solutions to problems. In this scope, an example are customer services support departments that have the objective of helping clients with the products maintenance. Usually, these realities are structured as *chatting centres* where the first available agent answers questions related to a very narrow domain.

It is intuitive to understand that recording the conversations between clients and agents opens a great business spot, in fact these data could be used to build software modules that can empower and help agents to be more efficient, allowing them to perform more tasks with less effort.

Here is where *Conversational AI* comes into the picture: it is the science that studies how to build these software solutions, also called chatbots, that do this job by exploiting historical conversational data. If *Conversation* refers to the type of data this science handles, *AI* instead relates to the fact that these modules usually exploit artificial intelligence in order to catch patterns between question and answers and to mimic how humans behave, learning from them.

There is a very problematic issue developers have to face while developing such kind of solutions: not always the direct question is enough to generate a meaningful answer. For this reason one of the most hot research trends in the community is to find a smart way to incorporate the conversational context into the model.

Chatbots have a wide range of applicability, in fact on the long term they will significantly help companies to save money and to exploit at most their knowledge bases, and for this reason they are a trending topic receiving a lot of interest.

System integrators constantly monitor this field with interest because, even if they do not yet propose a proprietary solution, they consider this topic as a real and valuable business spot.

On the other hand big companies, such as *Google*, are heavily contributing to this field with their teams by pushing the boundaries of research. They develop new techniques with a single day pace and they leverage on proprietary huge datasets using unbounded computational resources. They can thus develop great demos, such as the one presented by Sundar Pichai, CEO at Google, at Google IO about the new Google assistant [102].

Then there are start-ups, some of which have their own research branch implementing proprietary solutions. At *Loop AI Labs*, one of those start-ups, we had the opportunity to work on very interesting projects and Chapters 4, 5, 6 describe how, during this collaboration, we faced the development of a Conversational AI tool.

1.1 Aims and goals of the thesis

This thesis on the one hand is a synthesis of the technical and practical knowledge gained during the collaboration with *Loop AI Labs*, on the other hand it has the objective of presenting a critical analysis of literature models performance on a real use case, in order to shade light and clarify the controversial topic of Conversational AI.

At first, this work structures the various research trends in an organized document, wrapping the state of art technologies in a compact summary in order to allow future works to have a solid base on which start to build upon. Secondly, we want to show the application of these models in a real world case, in order to transmit how much distance there is between the classical datasets used by the research community [60] and practical scenarios, such as the one described in Chapter 4. We discuss how the switch between research and real datasets is problematic and what is the way the developer has to head in order to help the model to learn the complex patterns behind conversations. In fact, we want to provide a smooth and clear path from the theory to a practical use case that could help to build deep learning models for natural language understanding.

1.2 Contribution

This work proposes some insights about how to structure a Conversational AI project in a real world setting and it shows the best practices we devised at *Loop AI Labs*.

In particular, it wants to be a journey through dataset assignment, study of the state of art and both qualitative and quantitative analysis. In Chapter 4 we carefully study how much crucial and tricky is the the integration of the conversational context into the model architecture. To solve this issue, our quantitative analysis compares a novel *curriculum learning* approach, that uses an innovative learning rate decay to support such incremental training, with a more *traditional* approach.

One of the contributions of this thesis is the idea of designing a learning paradigm that takes into consideration the incremental complexity of conversations to define a procedure that could help the model to generalize better. Also, we believe that through this work we opened a new branch of research that, if further deepened, could potentially give important improvements in the design of generative models.

Another contribution of this work is the idea of training generative conversational models with the learning rate decay proposed by Vaswani et al 2017, [100].

This quantitative analysis shows how the performance changes by varying some hyper-parameters of the model, while the qualitative analysis is meant to show the results that the best approach obtains in two settings: the first is a set of success and bad cases picked from an evaluation set, while the second is the result of an interactive chat between the developer and the chatbot, showing how tricky it is to put such models in production.

In addition, we precisely describe the preprocessing pipeline, and in particular the development of a novel spell checker, a very important module to normalize raw text that we designed to ensure the integration only of safe corrections.

1.3 Overview

From a high level point of view, the structure of this thesis reflects the research plan we followed: we started by studying retrieval models and we discussed their pros and cons, finding out that their architecture was not intuitive and too much constrained. For this reason, we decided to move towards generative models, a more complex but recent, intuitive and fascinating solution. In the scope of generative models, we selected and validated a set of important hyper-parameters. As a last step, we selected the best performing model and we qualitatively analysed its performance both from an *off-line* and *on-line* point of view.

All this work was targeted at a real working reality, and for this reason oriented to save resources and to exploit at most available data and computational power, and this is the reason why we used *distributed training* and an *ad-hoc preprocessing chain.*

In particular, we organized this work in seven chapters:

- Chapter 2 is a high level overview of Conversational AI. It presents the main differences that characterize conversational models, describing them from a general perspective and showing their pros and cons. In addition, we discuss open issues and design choices such as *Long vs Short conversations* (Section 2.3.1) and *Open vs Closed domain* (Section 2.3.2), and challenges such as *Incorporating context* (Section

2.3.3), *Coherent personality* (Section 2.3.4) and *Intention and diversity* (Section 2.3.5). In Section 2.4 we add an interesting discussion about conversational ethic and in Section 2.1 we discuss the relationship between Conversational AI and natural language processing.

- In Chapter 3 we present an organized, complete, updated and technical analysis of the most successful Conversational AI algorithms, together with their foundations. We start by describing the retrieval approaches, Section 3.1, showing them along the two dimensions of *matching measure* and *feature extraction*. Then we consider generative models, Section 3.2, with a particular attention to *sequence to sequence architectures* ([95], [19]).

- While Chapter 2 is a high level overview of Conversational AI and Chapter 3 is a technical description of the algorithms, Chapter 4 is a precise report on how to approach a real world Conversational AI problem. It contains a detailed description of the preprocessing pipeline, where its most important tool is a novel statistical spell checker, which behaviour is carefully presented. We demonstrate how, specially for text, real world data is very problematic. It is full of typos, it is dirty and irregular and usually intractable as it is. For this reason, we show how the most of the work of the developer is usually dedicated to design an appropriate preprocessing chain that is able to reduce the entropy of the dataset.

- Chapter 5 describes the learning paradigm and the training set-up that we decided to adopt to learn a conversational model. We briefly outline the hardware setting that we used and the programming environment that we decided to adopt. In addition, we discuss which are the evaluation metrics of Conversational AI and we study when these metrics represent a signal correlated with the human evaluation.

- In Chapter 6 we show how to train and validate a generative model, both in terms of *qualitative* and *quantitative* metrics, demonstrating the difficulty of evaluating such algorithms with the current model independent metrics.

- In Chapter 7 we derive our conclusions and we point out which are the most promising research directions for future work.

Chapter 2

Conversational AI

Conversational AI is a controversial and fast growing branch of AI research. Its aim is to build software agents, also called chatbots, able to handle a conversation with a human being by using natural language.

Common AI powered assistants have the main objective of supporting the user in solving a problem, and big companies are investing in this field. For instance, Microsoft is making big bets on chatbots, and so are companies like *Facebook* (M) [20], *Apple* (Siri) [1], *Google* (Google assistant) [58], *WeChat* [63], *Amazon* (ALEXA) [2], *Microsoft* (Tay) [42] and *Slack* [10].

In this picture, a new wave of startups is trying to change how consumers interact with services by building consumer applications that use conversational agents to drive the user into their offert. To help the integration of this feature into applications, Microsoft recently released their own bot developer framework. Many companies are hoping to develop bots able to produce natural conversations indistinguishable from human ones, and many are claiming to be using *NLP* (Natural Language Processing) and *DL* (Deep Learning) techniques to make this possible.

In this thesis the term *utterance* will refer to everything exchanged during the conversation, being either a question or an answer. Specifically, a contiguous pair of question and answer is called *turn*, and the *context* is then defined by a number of turns back into the history of the conversation. In this scope, there are two types of settings, *single-turn* and *multi-turn*: the first one refers to the case in which the context is made only of the direct question, while in the other case the context is made of the direct question together with a number of turns taken from the conversation.

2.1 Conversational AI and NLP

Conversational models handle contexts made of possibly multiple utterances, a data format that is very similar to text documents, where the sentences represent the exchanged utterances. In order to generate meaningful answers, it is necessary to provide the agent with methods to understand these conversational documents, and chatbots strongly rely on this skill. Here is where *Natural Language Processing* (NLP) techniques come in handy. NLP is a fast growing field of computer science and it had a fast growth in the last years. It is concerned about how to program computers to process and analyse large amounts of natural language data. Challenges in natural language processing frequently involve speech recognition, natural language understanding, and natural language generation.

Referring to Conversational AI, there are many factors in which chatbots can vary, and one of the biggest differences is whether or not a bot is equipped with Natural Language Processing. Bots without NLP rely on buttons and static information to guide a user through a bot experience, and they are significantly more limited in terms of functionality and user experience than bots equipped with NLP. However, even if these algorithms are useful and currently used in most of the production environments, the community is heading in the direction of building conversational models that use NLP because this last kind scales better.

Linguistic based NLP methods are very different from the ones used nowadays, and this is mainly due to the prominent raise of *Deep Learning* (DL), a methodology that completely changed this field. Conversational models are horizontal with respect to NLP. They involve *speech recognition* in order to format sounds into text, *natural language understanding* to reason on text, and *natural language generation* to produce the answer. In the majority of the cases, understanding text refers to having the ability to assess *semantic similarity*. This usually means having a way to take unsupervised raw text data and structure it in a meaningful and compact form. On the other hand, the task of generating language can be accomplished in two ways: generating already assembled answers, and this is the approach adopted by retrieval chatbots, or assembling the answer word by word, and this is how generative conversational agents solve the problem.

In particular, nowadays NLP techniques are of three kinds:

- **Vector space model based:** This approach aims at developing a *vectorial geometric representation* for text data. Considering a set of documents, the first step is to extract the set of the unique words in them, called *vocabulary* or feature set. In particular, documents are represented by means of these features as vectors with size equal to the vocabulary size. Starting from this abstract representational concept, there are several models that propose how to compute the values of each element in the vector.

  Having this vectorial representation, it is possible to perform mathematical operations on documents and in particular to compute similarity. An example is shown in figure 2.1.

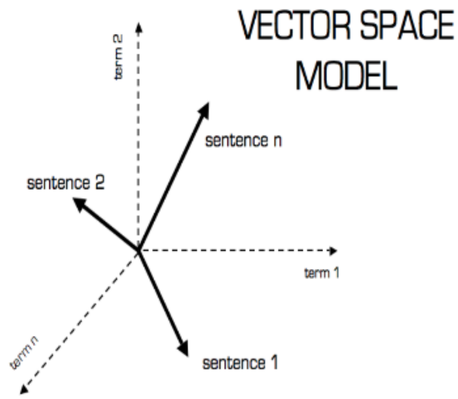


Figure 2.1: By using this model it is possible to map sentences into vector representations in a latent space. Figure coming from "Bitsearch" article [83] .

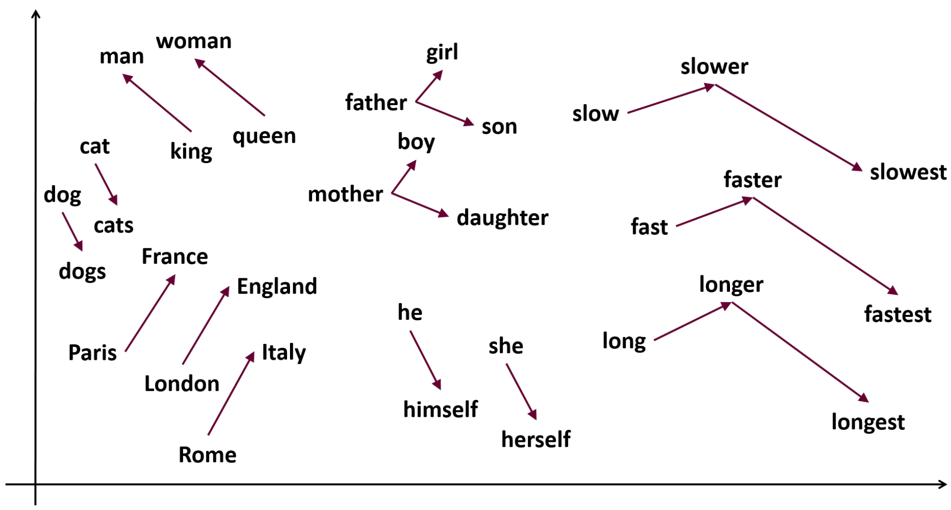


Figure 2.2: Learnt word vectors for common words. Semantically similar words are placed in the same part of the latent space. Figure coming from "http://www.samyzaf.com/" blog [81] .

- **Deep learning based:** Having the possibility to express documents as vectors, it is natural to think that it should also be possible to represent words as vectors and *"Efficient Estimation of Word Representations in Vector Space "* by Mikolov at al 2013 [64] was in this direction. They proposed a neural network architecture, called *Word2Vec*, able to learn a vector representation, also called word embedding, for each distinct word in a raw text (Figure 2.2). The great result of this model was that the learnt vectorial representations reflected the semantic relationships between words.

This method was a cornerstone in NLP because it allowed to learn semantic information in an unsupervised way without the need of an ontology. In fact, this algorithm does not need an explicit supervised signal.

This embedding technique is particularly useful to understand text. In particular, having a text and a mapping between each unique word in the text and a vector in a semantic learnt space, it is possible to aggregate the embeddings related to the words in the document and to abstract a document vector from them.

- **Linguistic based:** This approach is a defined as a set of methods with the goal of extracting structured information form unsupervised raw text data. Among them there are *POS-tagging*, *NER extraction*, *Stemming* and *Lemmatization* methods. All the above methodologies have two goals: the first is to reduce the complexity and the variance of text in order to reduce the scope of the problem, while to second one is to extract meaningful entities. At the bottom of this pipeline, literature has usually proposed the use of an ontology to pair tokens with a semantic meaning. While in theory this approach is pretty strong, in practice the problem is the reliability and the quality of the these ontologies.

Human language is more complex than humans could perceive. Besides its underlining logic, it is full of ambiguities and spell typos. Modelling natural language conversations is one of the most challenging artificial intelligence problems and for this reason Conversational AI needs several NLP methods such as text cleaning, typos fixing, language understanding, reasoning, and the utilization of common sense knowledge.

2.2 Retrieval vs Generative models

As already mentioned, from an high point of view, chatbot models are of two kinds, retrieval or generative. Even if they are two completely different ways of dealing with the same task, they mainly differ in the way they assemble the answer, either word by word, this is the case of generative models, or retrieving a historical answer, as happens in retrieval models.

2.2.1 Retrieval models

Retrieval models work by taking the incoming query and have the goal of finding the most similar question in an historical set of question-answer pairs to return the answer that the human agent gave to the retrieved question. In particular, retrieval chatbots employ NLP techniques in order to extract features to identify and compare questions.

Essentially, these models can be considered as clever chat assistants that perform an efficient semantic search in a space of historical question-answer pairs. Roughly speaking, the result is a model that learns how to assess similarity of text documents.

Because of their reliability, is better for production systems to be retrieval for now.

Procedure

In order to implement a retrieval-based software assistant, it is necessary to define a matching measure telling how much an answer a is suited for a question q. This measure can be stochastically or deterministically defined.

Stochastic measures can be seen as the prediction of a machine learning or a deep learning model trained to predict the matching score. Their input is the query-question pair, while the label can be a discrete, continous or discretized score between 0 and 1. Using stochastic measures usually gives better results because of the data driven nature of the process. As a drawback, because of their supervised nature, they need a labelled dataset and this is not often available.

Deterministic measures instead do not need any kind of supervision. These architectures exploit NLP techniques to extract a vectorial representation for documents and compute their similarity using a non-learnt function. This makes them more appetible to get baseline results with respect to the previous ones, even if, because of their nature, they usually results in very high level predictions, and this often turns into a poor performance.

Given an unseen customer question, these models use the matching measure to retrieve the top-k most similar historical questions. To overcome the memory issues that can arise if the model has to handle a very large dataset, this step can be enhanced with a clustering algorithm. In this case the dataset has to be organized into clusters with an off-line procedure. Then, the algorithm extracts the set of the centroids, one per cluster, and at inference time it compares the incoming sample with the centroids, finds the most similar one and returns the top-k most similar samples inside the selected cluster.

Considering only the previously retrieved subset of k questions, as the final step it returns the answer a related to the question-answer pair with the maximum matching score among the k retrieved ones.

Pros and Cons

Retrieval models has some pros and cons. In fact, they give grammatically correct answers, as the model output is coming directly from an historical repository. This way they are a starting point to build immediate customer engagement, in fact they show perfectly grammatically built answers and they can be considered an *off-the-shelf* baseline.

Unfortunately, these models may not be able to provide a precise match with the context and this often results in dull responses. Their main problem is that answers lack any novelty because the process is deterministic. In fact the model is not either assembling the predictions or reasoning on the context. This results in an inefficient exploitation of the dataset. Indeed, while a generative models massively combines data coming from different training samples to answer to a specific question, retrieval models do this only in a very little part.

Further, given that they are not suited for open domain conversations, their scope usually restricts to conversations that have the objective of pursuing a small set of simple goals. Retrieval models may be unable to handle unseen cases for which no appropriate predefined response exists. For the same reason, these models can not refer back to contextual entity information like names mentioned earlier in the conversation.

Even by using the context, they do not consider the evolving state of the conversation, so subsequent answers can be inconsistent with each other.

2.2.2 Generative models

If retrieval agents use NLP techniques to extract explicit features, generative ones instead use NLP to extract implicit, learnt features, in fact, while retrieval chatbots bias the feature extraction phase by exploiting a straightforward architecture, on the other hand generative models do not bias the architecture and let the model learn the features.

Deep Learning techniques can be used both in retrieval and generative models, but research seems to be moving towards the generative direction. These architectures, for example Sequence to Sequence models, firstly introduced by Sutskever et al in 2014 with their *"Sequence to Sequence Learning with Neural Networks "* [95] and by Cho et al in 2014 with *"Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation "* [19], are uniquely suited for generating text and researchers are hoping to make rapid progress in this area.

Procedure

These models fit an algorithm that automatically learns how to represent the conversational context, while the retrieval architectures build this representation by using traditional NLP methods that extract hand crafted features. This irremediably makes the retrieval process biased by the choices of the developer.

Then, generative models use the extracted context to generate the answer word by word. This is the second big difference between the two approaches.

Indeed, generative models have been designed to autonomously assemble the answer by choosing its words in a vocabulary while being conditioned on the current conversational context. It is interesting and fascinating how this process completely mimics the way humans answer to a question.

Pros and Cons

While this approach can seem more *human-like*, and for this reason more suitable for a real use case, there are pros and cons in using it.

In fact, the answer generation process mimics how humans behave. The job of the brain during the answering process can be seen as *two phases made*: an encoding one where the mind extracts the meaning from a question and a decoding step where it generates the answer. In the same way, generative models read the input data and build a compact representation of it. In the human parallelism, the result of the encoding phase is the thought that the brain generates while reading or hearing a question, that is used in the decoding phase to generate a coherent answer by assembling words belonging to a vocabulary set. This is what humans do when they generate an answer, the brain keeps the encoded question into consideration and chooses the words of the answer one at the time by taking into consideration the encoded question and the generated words.

In addition, these models can refer back to entities in the input. They give the impression of talking to a human agent and answers are novel by model construction, in fact the model learns how to combine words.

Once the model learns both the grammar and the language specific sentence generation process, it is possible achieve more customer engagement than in the case of using a retrieval based approach and when it makes mistakes, those mistakes are often *human-like.*

On the other hand, these models are hard to train and they need a huge amount of training data. Integrating world knowledge can be difficult. In fact, in order to do this, it is often necessary to completely change the model architecture introducing a proper bias. Their performance strongly depends on the preprocessing stage, together with data quality, data quantity and data distribution over the topic space.

Also, if the text is not accurately cleaned and corrected, the size of the vocabulary increases exponentially and as a result, on one side the model has to learn a lot of parameters with a fixed data amount and on the other side the hardware has to be able to handle the computational pressure. As training proceeds, sometimes it is quite likely to make grammatical mistakes and this may generate sentences that are not completely correct even once the model has been trained, especially when handling long contexts. In case a question has been seen in many different contexts it usually generates very common answers. This behaviour usually leads to software agents that prefer very general sentences with respect to specific ones.

Another problem is that these kind of models may not be able to handle a conversational topic if the dataset does not contain enough examples to learn it.

Even if carefully trained, the resulting model can be very general, proposing semantically meaningful answers that unfortunately are useless to drive the conversation towards really specific goals. This can be a problem if the model has to be used in a customer service environment, or in general where the conversational goal is to assist the user in complex and very particular procedures.

Nowadays, there not exist a reliable method to correctly evaluate the results of generative models and this is due to the fact that there is not a metric, differentiable or not, taking into account all the requirements that a successful human interaction has. In fact, current models have two very problematic vices: first of all they optimize a conceptually bad-posed on line error loss function, and secondly they are evaluated with wrong metrics. Among these two aspects, the one that mostly affects the quality of the final model is the conceptually wrong optimization procedure that is performed to train them. Cross entropy loss, the one that is usually adopted, does not model effectively the quality of an answer with respect to a certain question. The reason why this happens is that optimizing cross entropy means driving the model to favourite answers that, merely from the word by word perspective, are similar to the ground truth one. The problem here is that there are several answers $\{a_0, a_1, ..., a_n\}$, in general sharing few words, that can be considered as appropriate for a question q and they express different points of views, ways of solving a problem, or, maybe, opinions about a fact.

For this reason *answer-utility* should be taken into account in a goal based manner and cross entropy loss, being a supervised metric, does not teach correctly the model how to behave in a so complex and ambiguous scope. Liu et al published in 2016 a paper named *"How NOT To Evaluate Your Dialogue System: An Empirical Study of Unsupervised Evaluation Metrics for Dialogue Response Generation "* [59] where they found that none of the commonly used metrics really correlate with human judgement. Intuitively, using supervised techniques should bring to an inconsistent model that is not able to converse. On the contrary, the results are impressively good and often stunning for their correctness.

Another problem is that, once trained, generative models are deterministic. In particular they model a probability distribution over answers conditioned by the questions and this distribution is then used during the inference phase to make predictions. For this reason, given a certain input sequence of words:

$$\{w_1, w_2, ...w_{question\_length}\}$$

the model will always map it to a certain sequence of words:

$$\{w_1, w_2, ..., w_{answer\_length}\}$$

This is a big difference with the human behaviour, in fact human beings are stochastic in their answer generation process. In particular, if they are asked to answer to a certain question $\{w_1, w_2, ...w_{question\_length}\}$ multiple times they will probably give different answers and not always the same one. Luckily, generative models can be augmented with modules that allow them to partially overcome this problem.

2.3 Challenges in Conversational AI

There are some challenges when building conversational agents, most of which are active research areas. Specifically:

- Handling long rather than short conversations.
- Managing open or closed domain settings and what changes in the two cases.
- Choosing how to organize the conversational context.
- Ensuring cross-answer coherence.
- Assessing the quality of the answers that the proposed models generate.

2.3.1 Long vs Short conversation

The longer the conversation, the more difficult is to automate it.

On one side of the spectrum we have *Short-Text Conversations*, where the goal is to create a single response to a single input. For instance, an agent that receives a specific question from an user and has to reply with a single appropriate answer.

Then there are *Long Conversations*, where the dialogue goes through multiple turns and the agent needs to keep track of what has been said. Customer service conversations are typically long conversational threads with multiple questions.

2.3.2 Open vs Closed domain

Everyday conversations are made of several topic switches, and the space of the possible topics is infinite.

On the other hand, while talking to a customer service assistant to fix a problem, the user wants to reach a goal, and the set of the possible topics, here intended as conversational goals, is small and finite. Due to the dichotomous essence of this setting, during the model selection phase it is necessary to understand the nature of the problem, that can respectively be an *open domain* or a *closed domain* one.

In an *open domain* setting the user can take the conversation anywhere. It is not necessary to have a well-defined goal or intention. Conversations on social media sites like Twitter and Reddit are typically open domain, they can go into all kinds of directions. The infinite number of topics and the fact that a certain amount of world knowledge is required to create reasonable responses, makes this a hard problem. As could be imagined, generative models are suited for these tasks, even if, to work reasonably well, they need to be provided with enough and well topic-distributed data. Retrieval ones, instead, are not appropriate in this setting. In fact it is likely that the question relates to a topic that is not present into the historical repository.

In a *closed domain* setting instead, the space of possible inputs and outputs is somewhat limited because the system is trying to achieve a very specific goal. Technical customer support or shopping assistants are examples of closed domain problems. These systems do not need to be able to talk about very specific topics, they just need to accomplish their specific task as efficiently as possible. Sure, users can still take the conversation anywhere they want, but the system is not required to handle all these cases, and the users do not expect it to.

2.3.3 Multi vs Single turn

To produce sensible responses, it may be necessary to incorporate both linguistic and physical context. In long dialogs people keep track of what has been said and what information has been exchanged. That's an example of linguistic context.

The most common approach is to embed the whole conversation into a vector, as it was proposed by Serban et al in 2015 in their *"Building End-To-End Dialogue Systems Using Generative Hierarchical Neural Network Models "* [86] and by Yao et al in 2015 in *"Attention with Intention for a Neural Network Conversation Model "* [111]. Unfortunately, doing that with long conversations is challenging, especially if one may also need to incorporate other kinds of contextual data such as date/time, location, or information about a user.

In the scope of context handling, it is important to describe two families of algorithms. The first is called *single-turn models*, and represents the set of architectures that predict the answer only by using the incoming question.

In this case, the problem is formulated this way:

- Input: question q_0, a sequence of words.
- Target: answer a_0, a sequence of words.

This approach is poor and usually does not work well because the information needed to answer the current question may be back in the conversation. To continue the human-Conversational AI software parallelism it is the same as if there was a group of people talking about a topic and, after joining their already started conversation, a person was asked to answer a question related to a topic, that is unknown to her.

On the other hand, models belonging to the second family, called *multi-turn models*, are fed not only with the incoming question but they also take as their input a number of precedent conversational turns.

The problem, in the case of *2-turn* modelling, is formulated as:

- Input: context q_1-a_1-q_0, a sequence of utterances, each one being a sequence of words.
- Target: a_0 , a sequence of words.

Unfortunately, this way the model will be overwhelmed by the amount of information that is asked to process and usually the answer will be generic.

To overcome this problem, the research community came up with some mechanisms that could allow to pay attention (Bahdanau et al 2014 *"Neural Machine Translation by Jointly Learning to Align and Translate "*[4]) on different parts of the context, in order to build the context representation.

2.3.4 Coherent personality

When generating responses, the agent should ideally produce consistent answers to semantically identical inputs.

For instance, we should get the same reply to *How old are you?* and *What is your age?*. This may sound simple, but incorporating such fixed knowledge or *coherence* into models is a research problem. Many systems learn to generate linguistic plausible responses, but they are not trained to generate semantically consistent ones.

Usually, that happens because they are trained on a lot of data from multiple different users. Models like the one in *"A Persona-Based Neural Conversation Model"* Li et al 2016 [54], are making first steps into the direction of producing models that show coherence in their answers.

2.3.5 Intention and Diversity

A common problem with generative systems is that they tend to produce generic responses like *That's great!* or *I don't know* that work for a lot of input cases. Early versions of Google's Smart Reply, a sofware based on the work published by Kannan et al in 2016 in *"Smart Reply: Automated Response Suggestion for Email"* [47] tended to respond with *I love you* to almost anything.

That is partly a result of how these systems are trained, both in terms of data and in terms of actual training objective/algorithm. Some researchers tried to artificially promote diversity through various objective functions. However, humans typically produce responses that are specific to the input and carry an intention.

Because generative systems, and particularly open-domain systems, are not trained to have specific intentions, they lack this kind of diversity. A success story in this field is represented by the study made by Li et al in 2015, published in *"A Diversity-Promoting Objective Function for Neural Conversation Models "* [53], where they tried to artificially promote diverse answers.

2.4 How well does it actually work?

As shown in Sections 2.2 and 2.3, a Conversational AI problem can be framed in several different ways, some of which are very difficult to be faced. For instance:

1. A *retrieval open-domain* system is impossible nowadays, because of the complexity of hand-crafting enough responses to cover all the cases.

2. A *generative open-domain* system is almost *Artificial General Intelligence* (AGI) because it needs to handle all possible scenarios. Research is very far away from that as well, but a lot of work is going on in that area.

Given that it is difficult to build an agent able to behave properly in an open domain situation, the usual practice is to restrict the scope of Conversational AI to problems in close domains, where both generative and retrieval methods are usually appropriate. In this setting, the longer the conversations, the more important the context and the more difficult the problem becomes.

In a recent interview for *The Seattle Times* [97], *Andrew Ng*, now chief scientist of *Baidu*, puts it well:

> *"Most of the value of deep learning today is in narrow domains where you can get a lot of data. Here's one example of something it cannot do: have a meaningful conversation. There are demos, and if you cherry-pick the conversation, it looks like it's having a meaningful conversation, but if you actually try it yourself, it quickly goes off the rails."*

These systems work well if they operate in a narrow domain where the conversational topics are few. If instead it is necessary to have a more complex tool with the ability of handling conversations that are a little bit

Figure 2.3: Example of how these models can result in uncontrollable behaviours. Figure coming from "Business Insider" article [42] .

more open domain, it is pretty difficult to generate meaningful answers with the current research models. What instead these models can do is to propose answers to real human workers.

Another crucial point is that, in production systems, grammatical errors are very costly and may quickly drive users away. For this reason production systems usually adopt a retrieval based solution, as it is grammar typos free and in general more reliable.

Unfortunately, generative models may bring to embarrassing situations. In fact, given they are trained on real data, they automatically learn to answer like humans. If feeding them with data coming from a customer service maybe is not a big deal, providing instead data coming from *Twitter* or *Reddit* can show very bad behaviours. This is what happened to *Tay* [42], *Microsoft*'s chat assistant.

Tay was put on line as a Twitter user and after some days of normal behaviour it showed a very racist bias. In Figure 2.3 it is possible to see how it supports the holocaust denial.

This is what happens when training a software agent with real world data without carefully filtering it. Learning agents are like few years old babies, and they will learn also the biases contained in data, even if they are bad.

Conversational AI is a fascinating science. In fact, building a perfect conversational software agent would mean to create an artificial general intelligence (AGI) able to reason on concepts, and this is one of the final goals of artificial intelligence. Unfortunately, nowadays research is far away from building such a kind of agent, and this is due to two reasons: at first, the technologies are very recent and have to be further studied and improved, secondly there is a lack of big datasets to work on. These are the reasons why the current models, even if they show promising behaviours, are not suited to be put in productions, and this is especially true for generative agents.

Chapter 3

Conversation modelling

Modelling how humans interact is a fascinating task. To tackle this problem, a set of very diverse models can be adopted. The most interesting ones employ Natural Language Processing (NLP) to extract knowledge to drive the agent behaviour. The current chapter investigates, from a technical point of view, the various NLP-based approaches that the developer can embrace when tackling a Conversational AI task.

3.1 Retrieval approaches

Retrieval conversational models can be defined along two dimensions:

- The nature of the matching measure they adopt, either *stochastic* or *deterministic*.
- The way they extract the features to characterize the document, either with *traditional NLP methods*, with a *Bag Of Words* (BOW) model or with *Deep Learning* (DL) methods.

A retrieval chatbot can be built in three ways:

1. Deterministic matching measure and aggregation feature extraction. This is the case of the BOW-based retrieval conversational agents.
2. Stochastic matching measure and learnt feature extraction. This case is treated by Wu et al. 2016 in *"Response Selection with Topic Clues for Retrieval-based Chatbots"* [106].
3. Deterministic matching measure and learnt feature extraction. This configuration can be achieved by using an auto-encoder and it is treated in *"Deconvolutional Paragraph Representation Learning"* by Zhang et al 2017 [113] and in *"A Structured Self-attentive Sentence Embedding"* by Lin et al 2017 [57].

3.1.1 Feature extraction

In the scope of BOW models, the document features can be extracted in several ways: *count vectorizer*, *normed count vectorizer* and *TFIDF vectorizer*. All these methods assign a score w_{ij} for each word i and document j and they treat text as a set of unordered words; these methods have the strong limitation of being completely not aware of word order.

In its simplest version, called count vectorizer, the BOW model assigns a document vector d_j with elements $(w_{1j}, w_{2j}, ..., w_{Tj})$ to document j, where T is the size of the feature set, i.e. the possible words in the vocabulary. The score w_{ij} for feature i in document j is computed as:

$$w_{ij} = n_{ij} \tag{3.1}$$

where n_{ij} is the number of occurrences of word i in document j.

The normed count vectorizer instead computes the score w_{ij} as:

$$w_{ij} = \frac{n_{ij}}{|d_j|} \tag{3.2}$$

where $|d_j|$ is the number or words in doc j.

With count vectorizers, all the words are considered as if they had the same informative content. On the contrary language has several tokens, such as articles, prepositions, adverbs or the *be* verb which are not informative. In fact, they are used in many different contexts and in documents of very different kinds. For this reason, some approaches assume that those tokens, also called stop-words, are not meaningful to characterize a specific document. Following this idea, document representations should be built by an algorithm that assigns stop-words a score as low as possible. In order to achieve this goal, these algorithms usually adopt one of the following two approaches:

- Predefine a list of stop-words and filter them when building the Count vectorizer.
- Adopt a more general and data driven approach. In this case, it is needed to come up with a method that automatically assigns a high score to informative words and a low score to "pseudo" stop-words.

In *"A vector space model for automatic indexing"* [80], published in 1975 by Salton, Wong and Yang, the authors proposed an approach to overcome the impact of the stop-words in vector representation, without explicitly filtering them out. This approach is called *TFIDF* and the idea consists in assigning, to each feature in the document vector, a weight that is related with the informative content of that word.

The score w_{ij}, related to feature i in document j, is computed as:

$$w_{ij} = tf_{ij} * idf_i \tag{3.3}$$

where:

$$tf_{ij} = \frac{n_{ij}}{|d_j|}$$

$$idf_i = \log \frac{|D|}{|\{d : i \in d\}|}$$

and:

- n_{ij} is the number of occurrences of word i in doc j.
- $|d_j|$ is the number or words in doc j.
- $|D|$ is the number of documents.

As a result, the final TFIDF score will be high if two conditions hold; first, word i has to be present many times in doc j and, second, it should not be present in many other documents. Under these conditions, very informative words will be assigned an high score.

Word embedding

Deep learning architectures work by applying several matrix transformations and non linearities sequentially to vectors of real numbers. While this numerical feature does not create any problem if we handle numerical data, such as financial or report data, a few more problems arise in the case of text.

Given a raw piece of text T, first a vocabulary v is extracted. It can be seen as an ordered set containing all the unique words in T and it is usually sorted by word occurrence and it has a size, that is v_s. By means of the vocabulary, each word is assigned of an index i, a number between 0 and v_s. With this index, we build a vector $v(w)$ of shape $(1, v\_s)$ (Equation 3.4). Practically, considering a word w, its vectorial representation has the following form:

$$v(w) = (0, 0, ..., 0, 0, 1, 0, 0, ..., 0, 0) \tag{3.4}$$

having only one 1 in position i.

This representation is called *1-hot-encoding*. Even if this method is able to assign a vector to each word in a vocabulary, the extracted features are very sparse, and this is not a nice property. For this reason, usually a second step is performed, with the goal of using the 1-hot-encoded sparse vectors to a get a more handy dense representation. Usually this task is accomplished through a lookup function:

$$lookup(word, v)$$

This function takes as its input a word *word* in a vocabulary v, performs its 1-hot encoding and returns a dense vector of shape $(1, d_{model})$, where d_{model} is a chosen dense vector size, usually between 300 and 500.

In general, the returned vector is the i-th row of an embedding matrix W with shape:

$$(v\_s, d_{model})$$

where v_s is the vocabulary size.

The vector assigned to word w is computed as:

$$lookup(word, v) = ohe(word) * W \tag{3.5}$$

resulting in a vector of shape $(1, d_{model})$, called *word vector*, that can be fed into a DL architecture as the word representation.

These word representations can be combined with the document vector built with BOW methods to extract a more meaningful document vector. In fact, BOW based document vector have a clear limitation; they do not consider word order and for this reason they can not include any semantic information in the document vector. To overcome the just explained issue, document vectors built with vectorizers, *count vectorizer*, *normed count vectorizer* and *TFIDF vectorizer*, are considered as made of weights and not as actual vectors in a space. From this point of view, we can built the document vector as a weighted sum of word vectors driven by the weights provided by the vectorizers. This gives the possibility to build document vectors that could catch the semantic of words in text. In particular, considering a document j containing n unique words, the following steps are required:

- Extract a TFIDF vector $importance_j$ with shape $(voc\_size, 1)$.
- Exploit a technique among the ones presented in Section 2.1 to extract, for each word in the vocabulary, a word vector and pack them in a matrix W of shape (voc_size, d_{model}).
- Build the document vector

$$d_j = \sum_{i=1}^{voc\_size} importance_j[i] * W[i] \tag{3.6}$$

where d_j has dimensionality d_{model}.

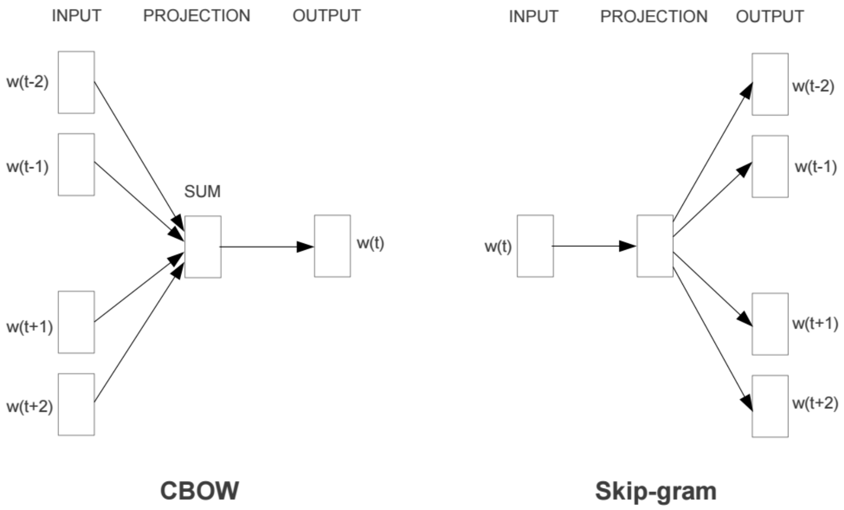


Figure 3.1: Word2Vdec training modes. Figure coming from "Efficient Estimation of Word Representations in Vector Space" by Mikolov at al in 2013 [64]

If importance scores are provided by the vectorizers, the embedding matrix W is usually built by using a Deep Learning (DL) model. Among these models, the simpler one is called *Word2Vec* and it was designed in *"Efficient Estimation of Word Representations in Vector Space"* by Mikolov at al in 2013 [64]. In their paper, Mikolov at al [64] proposed two novel model architectures for computing continuous vector representations of words from very large data sets. The first one is called *Continuous Bag Of Words*, Figure 3.1 on the left. It proposes to learn word vectors by predicting the current word from a window of words around it. The second approach instead is called *Skip Gram*, Figure 3.1 on the right, where the training procedure is designed by predicting the context words by starting from the current one.

Referring to Figure 3.1, an example of how the algorithm works is represented by taking the sentence *I like playing football in the backyard with my cousin*. Considering *CBOW* mode, a training sample is constructed by picking an index between the 0 and *num_words_in_sentence*, for example $t = 3$, and by extracting the relative word, that is $w(t = 3) = football$. The training task then consists in generating $w(t = 3)$ from the context words: $\{like, playing, in, the\}$. For *Skip-gram* mode instead, there is a specular situation where the network has to generate the context words $\{like, playing, in, the\}$ from $w(t = 3) = football$.

The method proposed by Mikolov et al. [64] is not the only way to train a word embedding model, in fact, as it has been published in *"From Machine Learning to Machine Reasoning"* by Bottou et al. in 2011 [11], another way to learn semantically consistent word vectors consists in training a network

FRANCE	JESUS	XBOX	REDDISH	SCRATCHED	MEGABITS
AUSTRIA	GOD	AMIGA	GREENISH	NAILED	OCTETS
BELGIUM	SATI	PLAYSTATION	BLUISH	SMASHED	MB/S
GERMANY	CHRIST	MSX	PINKISH	PUNCHED	BIT/S
ITALY	SATAN	IPOD	PURPLISH	POPPED	BAUD
GREECE	KALI	SEGA	BROWNISH	CRIMPED	CARATS
SWEDEN	INDRA	PSNUMBER	GREYISH	SCRAPED	KBIT/S
NORWAY	VISHNU	HD	GRAYISH	SCREWED	MEGAHERTZ
EUROPE	ANANDA	DREAMCAST	WHITISH	SECTIONED	MEGAPIXELS
HUNGARY	PARVATI	GEFORCE	SILVERY	SLASHED	GBIT/S
SWITZERLAND	GRACE	CAPCOM	YELLOWISH	RIPPED	AMPERES

Figure 3.2: This figure shows, for each word in a set, which are the top-10 nearest neighbours in the latent space. Figure coming from "Natural Language Processing (almost) from Scratch" published by Collobert et al in 2011 [23]

for predicting whether a 5-gram, sequence of five words, is valid. The model is trained by taking several 5-grams from *Wikipedia* and then breaking half of them by switching a word with at random word, making the 5-grams nonsensical. For example, considering the 5-gram *cat sat on the mat*, it is possible to break it to get *cat sat song the mat*.

This way it is possible to obtain two training samples:

1. Input: *cat sat on the mat*, Target: *correct*.
2. Input: *cat sat song the mat*, Target: *incorrect*

The architecture in this case is composed by a prediction function R, and an embedding function, actually a matrix, W.

For, the previous samples will be parsed in such a way:

- $R(W(cat), W(sat), W(on), W(the), W(mat)) = 1$
- $R(W(cat), W(sat), W(song), W(the), W(mat)) = 0$

In order to accomplish the previous task, the network needs to learn good parameters for both W and R.

Once trained, the matrix W contains, for each word in a vocabulary, a fixed size *word vector*. The interesting property is that, in the learnt latent space, similar words are close together and this can be visualized by looking at the top-k most similar tokens for a given word.

In 2011 Collobert et al published a study called *"Natural Language Processing (almost) from Scratch "* [23] where they discuss this property. In Figure 3.2 we shown how the network learns to place semantically similar words close together in the latent space.

3.1.2 Matching measures

Once we have the document vector, the second step consists in defining a matching measure. Considering deterministic measures, usually cosine similarity is exploited to compare two different document vectors. It aims at measuring the similarity between two non-zero vectors and it computes the cosine of the angle between them. It is thus a judgement of the orientation and not of the magnitude of the two vectors.

In particular, the similarity between two documents represented by the vectors a and b, it is defined this way:

$$c(a,b) = \frac{\sum_{i=1,..,n} a_i * b_i}{\sum_{i=1,..,n} a_i^2 * \sum_{i=1,..,n} b_i^2}, \tag{3.7}$$

where a and b are vectors of shape $(1, n)$.

When an unseen document comes in, the goal is to retrieve the most similar document in a knowledge base and to do that we usually need to compare its document vector with all the historical document vectors. This is feasible for small knowledge bases, not for big ones, e.g. those having more than one million documents.

In order to make this procedure feasible also for big historical sets of documents, we propose another step where we efficiently organize the knowledge base into clusters of similar documents, with the goal of speeding up the retrieval process. To do this, a clustering algorithm is applied to the historical set of document vectors. The grouping algorithm needs to respect two important properties. Firstly, objects in a cluster have to be very similar and is called the *compactness property.* The second claims that objects in different clusters have to show to be different one with each other and is called *separateness property.* This step can be faced with several algorithms such as *KMeans*, *Hierarchical clusering* or *DBSCAN.* The last one was published in *"A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise"* by Ester et al 1996 [27]. DBSCAN, because of its density based property, does not work well with sparse vectors such as document ones, so the choice usually falls on one of remaining.

Once run, this unsupervised learning algorithm provides the developer with a label for each training sample, representing the assigned cluster. Another significant speed up could be to physically store the clusters into different partitions. If we organize the knowledge base in such a way, an incoming document vector is processed by comparing it with the cluster centroids.

After this step, the second one usually consists in computing the similarity between the unseen document and each historical document inside the selected cluster to return the the one related with the higher similarity score.

Another interesting approach is to consider the matching measure as a learnt function that takes as its inputs two documents and returns a score between 0 and 1. By formulating the problem this way, we have several advantages:

- Speed up during inference time, in fact it is not necessary to loop over all the historical samples. We only have to compute the similarity between the unseen document and the centroids of the clusters.
- Possibility to tweak the similarity function and to incorporate a more complex logic.
- Move the solution toward a more data driven result, and so a more fine grained prediction.

The usual approach is to learn a deep learning model that estimates the matching function and an example is the architecture proposed by Wu et al in 2016 *"Response Selection with Topic Clues for Retrieval-based Chatbots"*[106], also called *TACNTN*, a model that is able to efficiently rank a set of candidates answers.

In particular, *TACNTN* [106] was designed to boost responses with rich content in retrieval-based chatbots. In fact, the matching between a message and a response is not only computed by considering the message and response vectors, but it also leverages extra topic information encoded in two topic vectors. The two topic vectors are computed as linear combinations of the topic words of the message and the response, where the topic words are obtained from a pre-trained LDA model, e.g. the one proposed by Blei et al in 2003 named *"Latent Dirichlet Allocation"* [8].

The architecture shown in Figure 3.3, works by feeding the message vector, the response vector, and the two topic vectors to a neural tensor network [91] that calculate a matching score.

Precisely, the model is defined in such a way. It exploits a dataset $D = \{y_i, m_i, r_i\}_{i=1,..,N}$, where respectively m_i and r_i represent an input message and a response candidate, while $y_i \in 0, 1$ denotes a class label. In particular, $y_i = 1$ means r_i is a proper response for m_i, otherwise $y_i = 0$. Moreover, each m_i in D is assigned to a topic word set $W_{m_i} = \{w^1_{m_i}, ..., w^h_{m_i}\}$, while each r_i with $W_{r_i} = \{w^1_{r_i} ..., w^k_{r_i}\}$.

The final goal is to learn a matching model f_θ for an inference sample (m, r):

$$f_\theta(m, r|\{y_i, m_i, r_i\}_{i=1,..,N}, \{\bigcup_{i=1}^{N} W_{m_i}, \bigcup_{i=1}^{N} W_{r_i}\}). \tag{3.8}$$

With this model, at inference time the retrieval algorithm takes an unseen question q and extracts a set C_k of K response candidates. In the base case

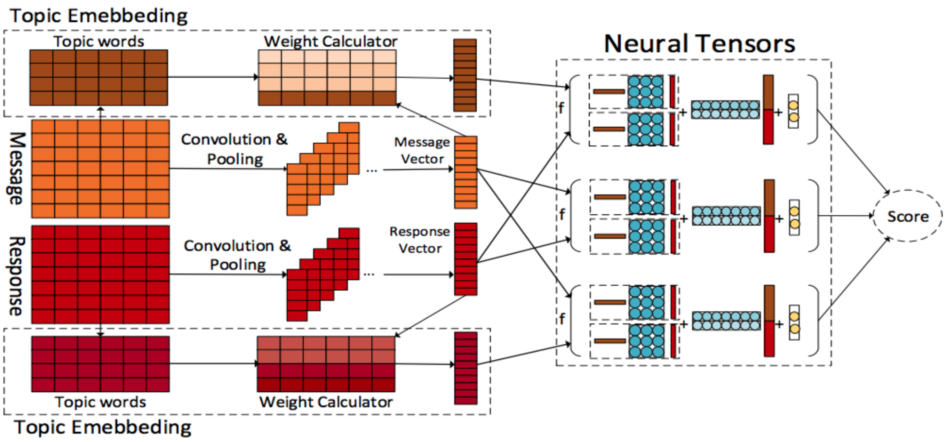


Figure 3.3: In this figure it can be seen the model of Wu et al. The message and the response, together with their topic words, are encoded by using a CNN. The extracted feature vectors are then parsed by a NTN to obtain the score. Figure coming from Wu et al, 2016 "Response Selection with Topic Clues for Retrieval-based Chatbots" [106]

C_k is made by all the possible answers in the knowledge base, while in a most advanced model this set is the result of a filtering procedure over all the possible answers:

$$\{c_1, ..., c_K\}.$$

Then, by using this scoring model, we generate the following set of scores, one for each message-response pair (q, c_i):

$$\{f_\theta(q, c_i)\}_{i=1,...,K}.$$

This way, at the end the selected answer q^* will be:

$$q^* = \text{argmax}(\{f_\theta(q, c_i)\}_{i=1,...,K}). \tag{3.9}$$

In order to compute the scoring function f_θ, the model exploits *Convolutional Neural Networks(CNNs)* and *Neural Tensor Networks(NTNs)*.

The last architecture was proposed in *"Reasoning With Neural Tensor Networks for Knowledge Base Completion"* by Socher et al in 2013 [91].

3.2 Generative approaches

Generative single turn conversational models have been treated by Vinyals et al (2015) in *"A Neural Conversational Model"* [101], Shang et al (2015) in *"Neural responding machine for short-text conversation"* [88], Li et al

(2015) in *"A Diversity-Promoting Objective Function for Neural Conversation Models"*[53], Xing et al (2016) in *"Topic aware neural response generation"* [107] and by Li et al (2016) in *"A persona-based neural conversation model"* [54].

These models converses by predicting the answer given the question and their strength is that they are a generative model trainable end-to-end requiring a small number of hand-crafted rules. A single training sample is made of a sequence of words and the model processes them one at the time by keeping in memory a conversational state that is dependent on the just processed tokens. In order keep a state, usually these models make use of recurrent cells, a natural generalization of feed-forward neural networks for sequences.

3.2.1 Recurrent models

The major limitation of Feed-Forward Neural Network is that they can not keep memory of the past, and so they can not condition their output on the previous inputs. Recurrent Neural Networks (RNNs) instead augment the framework provided by feed-forward neural networks (FFNNs) to address this issue.

A glaring limitation of FFNNs, also of CNNs, is that their API is too constrained: they accept a fixed-sized vector as input, for example an image, and produce a fixed-sized vector as output, containing the probabilities of different classes. In addition, these models perform this mapping by using a fixed amount of computational steps, the number of layers in the model.

The core reason that makes RNNs completely different is that they operate over sequences of vectors of variable length. They are basically networks with loops in them, allowing information to persist. Even if RNNs have a loop and they do not seem to be a standard directed neural network, it is possible to interpret them as a dynamic neural network with the same module repeated multiple times, each time sharing the same set of weights (Figure 3.4).

In the last few years, there has been incredible success applying RNNs to a variety of problems, for example speech recognition, language modelling, translation, image captioning and Conversational AI. One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, they would be extremely useful. But can they? It depends. Sometimes, only recent information is needed to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If the task is to predict the last word in *the clouds are in the sky*, it is pretty obvious the next word is going to be *sky*. In such cases, where the gap between the relevant information and the place that it is needed is

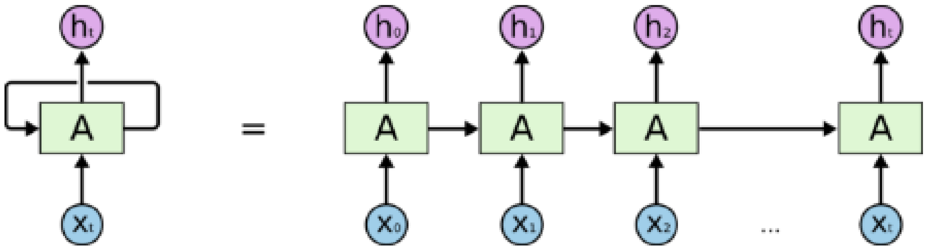


Figure 3.4: Unrolled RNN. It is possible to notice that it is always possible to unroll an RNN block and obtain a neural network with the same block repeated in series. Figure coming from "Understanding LSTM Networks", web blog "http://colah.github.io" published in 2015. [21]

small, RNNs can learn to use the past information.

But there are also cases where more context is needed. Consider trying to predict the last word in the text *I grew up in France... I speak fluent French.*. Recent information suggests that the next word is probably the name of a language, but narrowing down which language, needs the context of France, that is stored further back. It is entirely possible for the gap between the relevant information and the point where it is needed to become very large. Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

In theory, RNNs are absolutely capable of handling such *long-term dependencies*. Sadly, in practice, RNNs do not seem to be able to learn them. In general, given a sequence of inputs $(x_1, ..., x_T)$, a standard RNN computes a sequence of outputs of the same length $(y_1, ..., y_T)$ by iterating the following equations:

$$h_t = tanh(W_h * [h_{t-1}, x_t]), \tag{3.10}$$

$$y_t = W_y * h_t. \tag{3.11}$$

RNNs combine the input vector and their state vector with a function to produce a new state vector. This can be interpreted as running a fixed program with certain inputs and some internal variables.

Viewed this way, RNNs essentially describe programs. In fact, it is known that RNNs are Turing-Complete in the sense that, in theory, they can simulate arbitrary programs with proper weights. In this sense it is interesting the study made by Siegelman et al in 1992 published with the title *"On the computational power of neural nets"* [89].

To overcome the problem of long term dependencies, several recurrent cells have been proposed. One of them is called Long Short Term Memory,

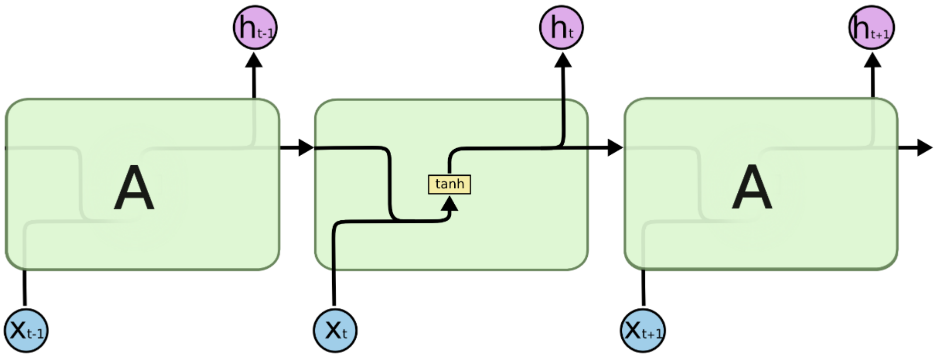


Figure 3.5: The repeating module in a standard RNN contains a single layer. Figure coming from "Understanding LSTM Networks", web blog "http://colah.github.io" published in 2015. [21]

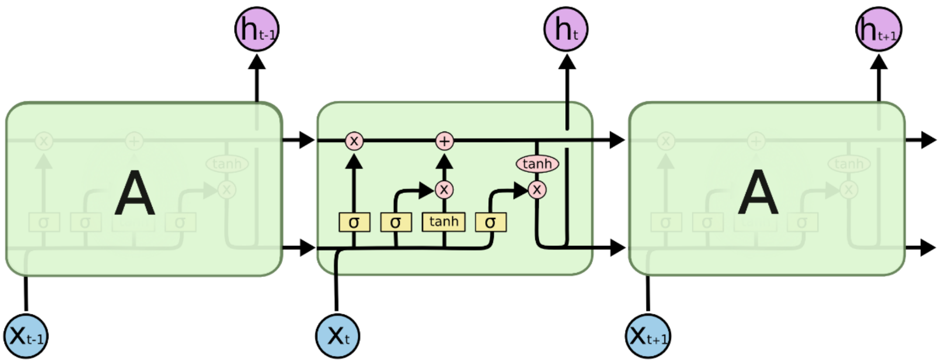


Figure 3.6: The repeating module in an LSTM contains four interacting layers. Figure coming from "Understanding LSTM Networks", web blog "http://colah.github.io" published in 2015. [21]

shortly *LSTM*. They are a special kind of RNN, capable of learning long-term dependencies and they were introduced by Hochreiter and Schmidhuber in 1997 with their *"LONG SHORT-TERM MEMORY"* [40]. LSTMs are explicitly designed to avoid the long-term dependencies problem.

All recurrent neural networks have the form of a chain of repeating modules of neural network. LSTMs also have this chain like structure, the difference is that the repeating module has a different architecture. Instead of having a single neural network layer, there are four, interacting in a very special way, as it is shown in Figure 3.5.

At each time step, the cell computes the current state C_t by combining several values:

$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f), \tag{3.12}$$

$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i), \tag{3.13}$$

$$o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o), \tag{3.14}$$

$$c'_t = tanh(W_c * [h_{t-1}, x_t] + b_c), \tag{3.15}$$

$$c_t = f_t * c_{t-1} + i_t * c'_t, \tag{3.16}$$

$$h_t = o_t * tanh(c_t), \tag{3.17}$$

$$y_t = W_y * h_t. \tag{3.18}$$

At time step t the cell state c_{t-1} flows horizontally on the top of the architecture and the network, by tweaking the parameters that affect the value of the two gates f_t and i_t, is able to decide how much of c_{t-1} is going to be maintained in c_t and how much of the new information c'_t will be part of c_t. This mechanism allows the network to learn what is needed to be kept and what is allowed to be thrown away, resulting in a better, more informative and cleaner state representation.

In 2015, Sergey Ioffee and Christian Szegedy came up with a technique called Batch Normalization and they published it in *"Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift"* [43]. The idea of Batch Normalization is that with neural networks, the inputs to each layer are affected by the parameters of all the preceding layers, so as the network gets deeper, small changes to these parameters get amplified by the later layers.

This causes a problem because it means that the inputs to the various layers tend to shift around a lot during training, and the network spends a lot of time learning how to adapt to these shifts as opposed to learning the ultimate goal which is the relationship between the input of the networks and the training labels.

What Ioffe and Szegedy came up with was a way to normalize the input data while the network is being trained, in such a way as that a more constant statistical distribution of layer inputs is ensured. This in turn accelerates training of the network. They basically did this by normalizing the summed inputs to each hidden neuron in the network on a batch by batch basis. The Batch Normalization approach worked very well and has become the state of the art for training CNNs, but it did not apply very well to Recurrent Networks, or when the batch size is needed to be small, such as in an online learning scenario.

For this reason Ba et al in 2016 published *"Layer Normalization"* [3], a technique to stabilize the training of RNNs. With Layer Normalization, instead of normalizing the inputs to each hidden neuron at batch level, the normalization is performed at each time step across the inputs on a layer-by-layer basis. Like batch normalization, this procedure stabilizes the dynamics of the hidden layers in the network and accelerates training, without the limitation of being tied to a batched implementation.

In particular, for an RNN at the tth time step:

- h^t is the state at time step t.
- a^t is the activation at time step t .
- W^h is the weight matrix.
- $f(.)$ is an element-wise non-linear function, usually $tanh$.

Layer normalized cells at time step t compute:

- $\mu^t = \frac{1}{H} * \sum_{i=1}^{H} a_i^t$, being the average value of the activations at time step t.
- $\sigma^t = \sqrt{\frac{1}{H} * \sum_{i=1}^{H} (a_i^t - \mu^t)^2}$, being the variance of the activations at t.

With those statistics, it is possible to compute the normalized hidden state:

- $a^t = W_h * [h_{t-1}, x_t]$ the activation vector at time step t
- $h^t = f(\frac{g}{\sigma^t} * (a^t - \mu^t) + b)$ the hidden state at time step t

where, b is the bias term while g is the gain parameter, both of them with the same dimension of h^t.

In a standard RNNs, we have the tendency for the average magnitude of the summed input of the recurrent units to either grow or shrink at every time-step, leading to exploding or vanishing gradients. In a layer normalized RNN, the normalization terms make it invariant to re-scaling all of the summed inputs to a layer, which results in much more stable hidden-to-hidden dynamics. In conclusion, RNN, but also LSTM, cells equipped with layer normalization often result in a faster and more stable training.

3.2.2 RNNs topology

Recurrent models are complex and can vary a lot in their architectures; it is possible to define them along two dimensions: *Type* and *Depth*.

In general, a recurrent model consists of a hidden state h and of an optional output y and it operates on a variable length sequence $x = (x_1, ..., x_T)$.

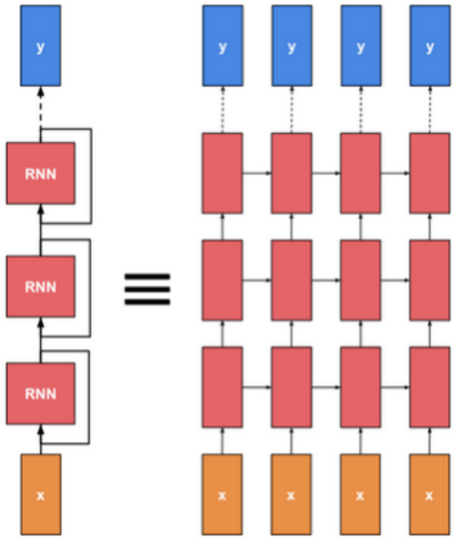


Figure 3.7: Multilayer RNN is a neural network with multiple RNN layers. Figure coming from "Recurrent Neural Network", web blog "http://http://sqlml.azurewebsites.net" published in 2017. [93]

At each time step t, the hidden state h_t of a general recurrent model is updated by using the general equation:

$$h_t = f(h_{t-1}, x_t). \tag{3.19}$$

In equation 3.19, f is a non-linear activation function that may be as simple as an element wise logistic sigmoid function and as complex as a long short-term memory (LSTM) unit. In particular, there are several different implementations of f_θ, i.e. the *cell type*, such as:

- **RNN**
- **LSTM:** Introduced by Hochreiter et al, 1997 *"LONG SHORT-TERM MEMORY"* [40]
- **GRU:** Introduced by Cho et al, 2014 *"Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation"* [19]

Recurrent models also differ in terms of depth. Increasing the depth of the architecture (Figure 3.7) means adding several recurrent layers on top of the first one and make the information flow from the input to the target deeper.

In presence of several layers, each layer l computes the hidden state h_t^l in such a way:

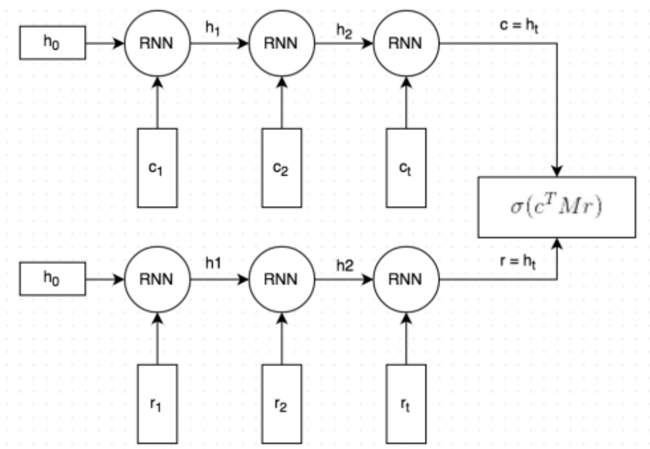


Figure 3.8: In this picture, each rectangle represent a vector and each arrow represent a function, for example a matrix multiply. Input vectors are in red, output vectors in blue and green vectors hold the network state. Figure coming from "The Ubuntu Dialogue Corpus: A Large Dataset for Research in Unstructured Multi-Turn Dialogue Systems", Lowe et al 2015.[60]

$$h_t^l = f(h_{t-1}^l, h_t^{l-1}), \tag{3.20}$$

and $h_t^0 = x_t$. This variant increases the complexity of the network and usually allows to catch hierarchical features.

3.2.3 Retrieval recurrent dual encoder

In the scope of recurrent models, it is important to point out the architecture proposed by Lowe et al (2015) [60]. They designed a model able to score a pair of question and answer by using a dual encoder architecture (Figure 3.8). The result is a recurrent retrieval chatbot with the objective of estimating a matching measure that, differently from TACNTN [106], uses recurrent cells to encode the question answer pair.

In particular, this model makes use of two RNNs with which it respectively computes the vector representation of a pair $c, r \in R^n$ of question c and an answer r. The model then calculates the probability that the given response r is the ground truth response given the question c, and does this by taking a weighted dot product:

$$p(r\ is\ correct|c, r, M) = \sigma(c_T Mr + b), \tag{3.21}$$

where M is a matrix of learned parameters and b is the bias. The model is trained using *negative sampling* to minimize the cross-entropy error of all context-response pairs.

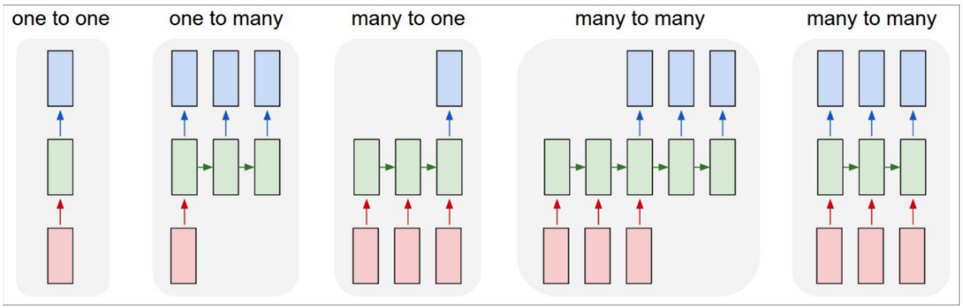


Figure 3.9: In this picture, each rectangle represent a vector and each arrow represent a function, for example a matrix multiply. Input vectors are in red, output vectors in blue and green vectors hold the network state. Figure coming from "The Unreasonable Effectiveness of Recurrent Neural Networks", web blog "http://http://karpathy.github.io/" published in 2015. [48]

3.2.4 Deep learning architectures

What makes RNNs special is that, while other deep learning architectures work with fixed size vectors as inputs and generate fixed size vectors as output, recurrent models instead can work with sequences of vectors both in input and in output. In particular, Andrey Karpathy in 2015 published a blog post named *"Unreasonable Effectiveness of Recurrent Neural Networks"* [48] where he said that deep learning architectures can be categorized in five main categories:

As it is shown in Figure 3.9, going from left to right there are five main families of *Deep Learning* (DL) architecture:

- The first one is the vanilla processing mode without RNNs, from fixed size input vector to fixed size output vector. Among these models there are the CNNs and FFNNs.
- The second represents the case in which there is a
  and a variable size sequence as the output. This is the case of auto-captioning where the input is an image and the output is a variable size sequence of words.
- The third one represents the case when there is a variable size sequence as input and a fixed size vector as the output. This is the case of sentiment analysis.
- The fourth one instead represents the most interesting case where both the input and the output are sequences of variable size. This is the case of sequence to sequence models, first introduced by Cho et al [19] and by Sutskever et al [95]. In particular, the most intuitive application of them is the machine translation task where a first module reads a sequence in English and another module generates another sequence

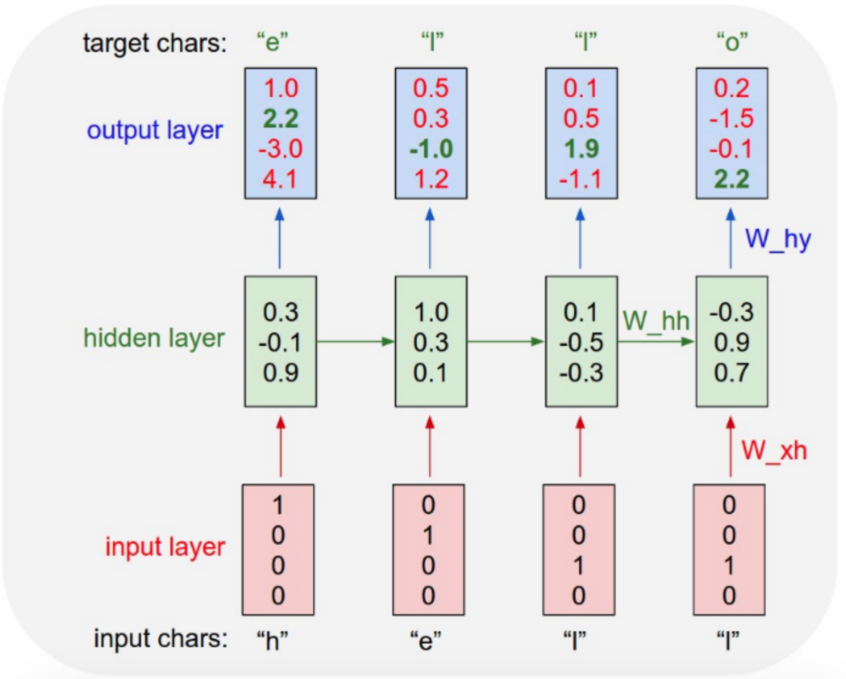


Figure 3.10: An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons). This diagram shows the activations in the forward pass when the RNN is fed the characters hell as input. The output layer contains confidences the RNN assigns for the next character (vocabulary is "h,e,l,o"); Intuitively, green numbers should to be high and red numbers should to be low. Figure coming from "The Unreasonable Effectiveness of Recurrent Neural Networks", web blog "http://http://karpathy.github.io/" published in 2015. [48]

in French. Beside this case, these models, because of their extreme flexibility, can be applied to a vast variety of tasks, and one of them is Conversational AI.

- The last case represents the situation where the input and output sequences have the same length. This is the case of video classification where the goal is to classify each frame in a video. Another example of these models is represented by language models.

Language models, firstly designed by Mikolov et al., 2010 [65] and here shown in Figure 3.10, are one of the most important foundations on which generative chatbots rely.

In the categorization of Figure 3.9, generative Conversational AI models are represented by the fourth set of models, *many to many* task with input and output having different lengths. This architectures are heavily based on the fifth models of Figure 3.9, simpler *many to many* algorithms handling input and output sequences with the same length. Language modelling is a typical application belonging to the fifth family of architectures. Our objective is to study language models in order provide foundations to understand how more complex architectures, such as the ones used to build generative chatbots, behave.

Language models take a sequence of words as their inputs and they attach a prediction task to each element in the list. In particular for each input word they are trained to predict the next one while being conditioned on all the processed words. As it is shown in Figure 3.10, the selected architecture is a single RNN which state, the green vector, is updated while the tokens are processed.

Furthermore, this task can be explained this way:

1. It takes an input sequence, that is *hello*, splitted it into characters.

$$\{h, e, l, l, o\}.$$

2. For each character defines a prediction task with the next character as the target.

3. The input tokens are organized into a character vocabulary and the $lookup(char, v)$ function assigns them a vector. This step is performed by the red arrow, while the red rectangle represents character vector and $W_{x,h}$ represents the character embedding matrix, which weights are learnt during the training process.

4. The hidden state h_t is computed with equation 3.19.

5. Considering an output distribution made of K classes, the next goal of this step is to predict a class from the hidden state h_t. This is done by computing the logits vector at time t, specifically:

$$logits_t = g(h_t), \tag{3.22}$$

   where $g(.)$ is usually a matrix multiplication, also called *output projection*. In this case:

$$g(h_t) = h_t * W_{h,y}. \tag{3.23}$$

   Practically, it projects the vector h_t of dimension d_{model} to another vectorial space of dimension K by exploiting a learnt matrix $W_{y,t}$ of dimension (d_{model}, K). Moreover, the vector $logits_t$, the blue vector in Figure 3.10, contains a confidence score for each one of the K classes.

6. To predict the next letter, a softmax activation function is applied to the scores vector $logits_t$, obtaining a vector of probabilities $probs_t$, one for each class:

$$probs_t = softmax(logits_t) = p_\theta(x_t|x_{t-1}, ..., x_1). \tag{3.24}$$

In particular, the j-th element of the softmax vector $\sigma(z)$ is computed this way:

$$\sigma(z)_j = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)}. \tag{3.25}$$

Specifically, each element of the vector $probs_t$ is computed by applying the softmax function to $g(h_t)$.

$$p_\theta(x_{t,j}|x_{t-1}, ..., x_1) = \frac{exp(g(h_t)_j)}{\sum_{j'=1}^{K} exp(g(h_t)_{j'})}, \tag{3.26}$$

where $g(h_t)_j$ is the j-th element of the $g(h_t)$ vector, so the score of the j-th class. Usually, the function $g(.)$ is computed with a matrix multiplication:

$$g(h_t)_j = w_j * h_t,$$

where w_j is the j-th row of the projection matrix W.

7. By combining these probabilities, the probability of a sequence $\{x_i\}_{i=1,...,T}$ is computed as:

$$p_\theta(\{x_i\}_{i=1,...,T}) = \prod_{t=1}^{T} p_\theta(x_t|x_{t-1}, ..., x_1). \tag{3.27}$$

3.2.5 Seq2Seq models

Generative conversational agents make use of the fourth architecture, called Sequence to Sequence models. They are a natural extension of the fifth architecture, in the sense that they allow to have variable length sequences as their input and output.

Sequence to sequence models, firstly introduced by Sutskever et al in 2014 with their *"Sequence to Sequence Learning with Neural Networks"* [95] and by Cho et al in 2014 with *"Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation"* [19] are deep learning architectures that had great success in a variety of tasks such as machine translation, speech recognition, text summarization and Conversational AI.

At an high level, they are made of two main modules:

- The encoder that learns to build a meaningful vectorial representation of a variable-length sequence $(x_1, ..., x_T)$ and encodes it into a fixed-length vector representation c of size d_{model}.

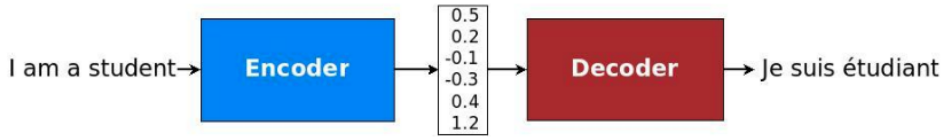


Figure 3.11: Encoder-decoder architecture, an example of a general approach for Neural Machine Translation(NMT). An encoder converts a source sentence into a meaning vector which is passed through a decoder to produce a translation. Figure coming from the github turorial called "Neural Machine Translation (seq2seq) Tutorial" [96].

- The decoder that learns to decode a given fixed-length vector representation, the encoding c, back into yet another variable-length sequence $(y_1, ..., y_S)$.

These models were firstly introduced to tackle machine translation tasks where the input sequence is a sentence in a language and the model is trained to generate the translation in a target language. Considering the case of translating between English and French, the training task can be defined as generating the sentence *Je suis éstudiant* after reading the sentence *I am a student*. In this setting the model architecture can be visualized as in Figure 3.11.

The encoder is an recurrent model that reads each symbol of an input sequence x sequentially. As it reads them, the hidden state of the model changes according to Equation 3.19. After reading the end of the sequence, the encoder contains a summary c of the whole input sequence. The decoder, on the other hand, works as a recurrent language model and computes a probability that is very similar to the one in Equation 3.27. The only difference is that, when it computes the probabilities of the target tokens $(y_1, ..., y_S)$, it is conditioned on the context vector c extrapolated by the encoder. Considering the language model probability formulation:

$$p_\theta(y_t|y_{t-1}, ..., y_1),$$

Sequence to Sequence models can be extended this way:

$$p_\theta(y_t|c, y_{t-1}, ..., y_1).$$

For, at each decoding time step the decoder module computes:

- The decoder hidden state at time step t :

$$h_t = f(h_{t-1}, y_{t-1}), \tag{3.28}$$

with $h_0 = c$.

- The probability that the model assigns to the correct class y_t:

$$p_\theta(y_t|c, y_{t-1}, ..., y_1) = softmax(g(h_t)), \tag{3.29}$$

with $g(.)$ being the output projection over the classes.

In the base architecture, like the one proposed by Sutskever et al in 2014 in their *"Sequence to Sequence Learning with Neural Networks"* [95] and by Cho et al in 2014 with *"Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation"* [19], c is computed as the last state of the encoder. In more complex settings instead, like the one proposed by Bahdanau et al in 2014 with the name *"Neural Machine Translation by Jointly Learning to Align and Translate"* [4] and by Luong et al 2015 in their *"Effective Approaches to Attention-based Neural Machine Translation"* [62], for each decoding step $t \in [1, ..., S]$, the decoder is fed with a different context vector c_t.

This mechanism, called Attention, greatly empowers sequence models. In particular, the decoder state h_t computation is not changed:

$$h_t = f(h_{t-1}, y_{t-1}), \tag{3.30}$$

with $h_0 = c$.

On the other hand, equation 3.29 is updated with the following one:

$$p_\theta(y_t|c_t, y_{t-1}, ..., y_1) = softmax(g(v(h_t, c_t))), \tag{3.31}$$

being $g(.)$ an output projection over the classes and $v(.)$ an aggregation function.

Machine translation parallelism

As it is shown, sequence to sequence models are suited for tasks with both input and target with variable length. The sequential nature of this task is challenging, and this is due to the dependencies that can happen between input and target tokens. The more complex the dependencies, the harder is the task.

One way researchers have tried to ease this problem is to use LSTMs cells in order to better model long term dependencies. Even with this technical tool, the performance of these architectures heavily depend on the type of dependencies that the task requires to handle. For its basic dependencies, machine translation is the task that is usually adopted to present results in this setting.

The dependencies between two sequences can be classified into three sets:

1. *Directly tok2tok aligned*: The output token $o_{t'}$ can be generated by only considering $x_{t''}$ and $t' == t''$.

2. *Indirectly tok2tok aligned*: The output token $o_{t'}$ can be generated by only considering $x_{t''}$ and $t'! = t''$.

3. *multitok2tok or tok2multitok*: The output token $o_{t'}$ is mapped to a set $(x_{t''-D}, ..., x_{t''}, ..., x_{t''+D})$ of input elements or the input token $x_{t'}$ refers to a set $(o_{t''-D}, ..., o_{t''}, ..., o_{t''+D})$ of output tokens.

The interesting part about translation is that the dependencies between the generated tokens and the input tokens are usually, but not always, *tok2tok aligned*, in the sense that single input elements are usually translated into single output tokens.

Conversational AI is a very challenging task to be faced with sequence to sequence models because the dependencies it underlines cannot be defined a priori. In fact there are a lot of cases in which also an even infinitely long context does not contain the correct information to give the proper answer, and for the dependency is intractable.

Even if it is quite impossible to achieve perfect answer quality in this setting, feeding the model with as much context as possible seems the correct way to go. Unfortunately, this way the model will be overwhelmed by a gigantic amount of information and can result in a pretty confuse module. The best solution is usually to feed a reasonably big amount of context and in parallel to provide the model with a mechanism with which it could be able to pick to the correct information among the provided input data. A solution that tries to do this is called multi-turn attention-based chatbot and it exploits complex kinds of attention mechanisms ([4], [62] and [108]).

Probabilistic View

From a probabilistic perspective, Sequence to sequence models are a general method to learn a conditioned distribution over a variable-length sequence $(y_1, ..., y_S)$ conditioned on yet another variable-length sequence $(x_1, ..., x_T)$, encoded as c:

$$p_\theta(y_1, ..., y_S | x_1, ..., x_T) = \prod_{t=1}^{S} p_\theta(y_t | c, y_{t-1}, ..., y_1),$$

where $p_\theta(y_t | c, y_{t-1}, ..., y_1)$ is the represented as a softmax over the words of a vocabulary. It is important to notice that the input and output sequence lengths T and S may be different. Once the this Encoder-Decoder architecture is trained, the model can be used:

- To generate a target sequence given an input sequence.

- To score a given pair of input and output sequences, where the score is the probability $p_\theta(y|x)$

Loss

One of the many nice properties of Sequence to sequence models is that they can be trained end-to-end. In fact, from a training sample $(x_1, ..., x_T), (y_1, ..., y_S)$ it is possible to propagate the updates across all the matrices that are involved in the probability computation, from the embedding matrix passing through the recurrence parameters and arriving to the output projection. This can be done by computing a loss function dependent on the input data and on the model parameters $J(x, y, \theta)$. This loss function is usually build by using the cross-entropy function, a measure of dissimilarity between two probability distributions p and q:

$$xentropy(p, q) = -\sum_{i=1}^{K} p_i * \log(q_i). \tag{3.32}$$

In this case, at decoding time step t, for sample n the cross-entropy is computed in such a way:

$$xentr(n, t) = xentr(y_{1n}, ..., y_{tn}, x_{1n}, ..., x_{Tn}) =$$

$$xentropy(trueprobs(y_{tn}), probs(y_{1n}, ..., y_{t-1n}, x_{1n}, ..., x_{Tn}). \tag{3.33}$$

Nicely, it is always possible to infer a probability distribution $trueprobs(y_{tn})$ from the true labels and the number K of distinct classes. In particular, knowing that:

$$y_{tn} = l \ with \ l \ \in \ (1, ..., K),$$

meaning that at time step t of the the n-th sample the true label is l, the probability distribution $trueprobs(y_{tn})$ can be defined in such a way:

$$trueprobs(y_{tn}) = (0, 0, ..., 0, 0, 1, 0, 0, ..., 0, 0). \tag{3.34}$$

It is a vector of shape $(1, K)$, made of all 0s and only one 1 placed in the l-th position.

Moreover, also $probs(y_{1n}, ..., y_{t-1n}, x_{1n}, ..., x_{Tn})$ is a vector of shape $(1, K)$ that assigns a probability to each class $k \in \{1, ..., K\}$ and given that its elements sum up to 1, it represents a probability distribution over the classes. Going back to Equation 3.32, for the n-th training sample for time step t the cross-entropy function is computed in such a way:

$$xentr(n, t) = -\sum_{i=1}^{K} trueprobs(y_{tn})[i] * \log(probs(y_{1n}, .., y_{t-1n}, x_{1n}, .., x_{Tn}))[i]. \tag{3.35}$$

that is:

$$xentr(n,t) = -1 * \log(p_\theta(y_t|c, y_{t-1}, ..., y_t)) = -\log(p_\theta(y_t|c, y_{t-1}, ..., y_t)).$$

The final loss function can be defined as:

$$J(\theta|D) = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{S_n} xentr(n,t) =$$

$$-\frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{S_n} \log p_\theta(y_t|c, y_{t-1}, ..., y_t) =$$

$$-\frac{1}{N}\sum_{n=1}^{N} \log \prod_{t=1}^{S_n} p_\theta(y_t|c, y_{t-1}, ..., y_t) =$$

$$-\frac{1}{N}\sum_{n=1}^{N} \log p_\theta(y_1, ..., y_{S_n}|x_1, ..., x_{T_n}).$$

Obtaining:

$$J(\theta|D) = -\frac{1}{N}\sum_{n=1}^{N} \log p_\theta(y_1, ..., y_{S_n}|x_1, ..., x_{T_n}), \tag{3.36}$$

where:

- θ is the set of parameters of the model
- N is the size of the dataset
- T_n is the number of tokens in the n-th source sequence
- S_n is the number of tokens in the n-th target sequence
- D the dataset, $\{(x_{1n}, ..., x_{Tn}), (y_{1n}, ..., y_{Sn})\}_{n=1}^{N}$

The optimization procedure is then defined as finding the minimum of the error function $J(\theta|D)$ with respect to the parameters. For, the goal is to find θ_{XE} such as:

$$\theta_{XE} = \operatorname*{argmin}_{\theta} J(\theta|D). \tag{3.37}$$

The optimization procedure can also be defined from the point of view of the maximum log-likelihood estimation. In fact, the two components of the proposed Encoder-Decoder module can be jointly trained to maximize the conditional log-likelihood and θ_{ML} is estimated this way:

$$\theta_{ML} = \underset{\theta}{\operatorname{argmax}} \frac{1}{N} \sum_{n=1}^{N} \log p_\theta((x_{1n}, ..., x_{Tn}), (y_{1n}, ..., y_{Sn})), \tag{3.38}$$

but:

$$\frac{1}{N} \sum_{n=1}^{N} \log p_\theta((x_{1n}, ..., x_{Tn}), (y_{1n}, ..., y_{Sn})) = -J(\theta|D).$$

And so:

$$\theta_{ML} = \underset{\theta}{\operatorname{argmax}} -J(\theta|D) = \underset{\theta}{\operatorname{argmin}}\, J(\theta|D) = \theta_{XE}. \tag{3.39}$$

Meaning that maximizing the log-likelihood or minimizing the cross entropy brings to the same optimal value of $\theta_{optimal}$.

3.2.6 Seq2Seq models with Attention

In a recent interview, *Ilya Sutskever*, research director at OpenAI, mentioned that attention mechanisms are one of the most exciting recent advancements in deep learning, and they are here to stay.

In particular, these techniques in neural networks are loosely based on the visual attention mechanism found in humans. Human visual attention is well-studied and while there exist different models, all of them essentially come down to being able to focus on a certain region of an image with *high resolution* while perceiving the surrounding image in *low resolution*, and then adjusting the focal point over time.

Attention mechanisms have a long story, particularly in image recognition. Examples include *"Learning to combine foveal glimpses with a third-order Boltzmann machine"* published by Larochelle et al in 2010 [52] and *"Learning where to Attend with Deep Architectures for Image Tracking"* by Denil et al (2011) [25]. But only recently these architectures have made their way into recurrent neural networks (RNNs) architectures that are typically used for NLP. More formally, in this framework the encoder processes the input sequence $(x_1, .., x_T)$ into a sequence of vectors $(h^e_1, ..., h^e_T)$. From this sequence of vectors, the hidden state of the decoder at time step t, that is h^d_t, is updated with the equation 3.28 and the probability for decoding time step t is computed with equation 3.29.

In general, the vector c on which the decoding is conditioned is computed as:

$$c = q(h^e_1, ..., h^e_T). \tag{3.40}$$

In the basic case proposed by Sutskever et al in 2014 in their *"Sequence to Sequence Learning with Neural Networks"* [95] and by Cho et al in 2014 with *"Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation"* [19], the vector c is computed this way:

$$c = h^e_T. \tag{3.41}$$

From this perspective can be seen how the basic sequence to sequence without attention architecture throws away a very big amount of information during the switch from the encoder to the decoder.

In their paper *"Neural Machine Translation by Jointly Learning to Align and Translate"*, Bahdanau et al (2014) [4]) assert that using such a fixed-length vector c is an information bottleneck in improving the performance of these models. For this reason, the work of Bahdanau et al (2014) [4] and of Luong et al (2015) [62] is mainly focused in improving the basic architecture in order to exploit the full amount of information coming from the encoder. In particular, the idea that stays behind the attention mechanism is that not all the information coming from the encoder is needed to produce the decoding token at time step t.

Indeed, the attention module empowers the decoder with a way to learn automatically to soft-search for the parts of the source sentence that are needed to predict a target word, without having to form these parts explicitly as hard segments. The intuition behind this idea is very promising. In fact, this module mimics a very important and peculiar characteristic of the brain; during each moment of their life, humans are subjected to a gigantic amount of stimula, the input, and have to come up with a decision, the output for that specific moment. Even if the processing stage that goes from inputs to outputs is complex and nowadays partially unknown, it is likely for the brain to be provided with a mechanisms allowing to throw away the inputs that are useless for the current task. It can be seen as a sort of defensive approach humans adopt to not be overwhelmed by the enormous amount of inputs they receive and to pay attention only on what is relevant for the current goal.

The attention mechanism is a way to switch from Equation 3.29 to Equation 3.31. As can be noticed, the only difference is that if Equation 3.29 is conditioned on a static vector c, independent from the time step t, equation 3.31 is instead conditioned on a dynamic context vector c_t. This is a crucial difference and here stays much of the contribution that attention provides. In fact, in the attention based case, each decoding step t is provided with a different context vector c_t, computed with Equation 3.40. While in the basic case of Sutskever et al 2014 [95] and by Cho et al 2014 [19], c is computed with Equation 3.41 and is constant, with attention c_t is a dynamic aggregation of the encoder states.

The context vector c_t can be computed in several different ways. Among them, the most used have been proposed by Bahdanau et al 2014 [4] and by Luong et al 2015 [62]. While they differ in how the vector is derived, they share the same subsequent steps. Specifically, at time step t the hidden state of the decoder h_t^d, and the dynamic context vector c_t are combined as follows to produce an attentional hidden state $\hat{h_t^d}$:

$$\hat{h_t^d} = tanh(W_c * [c_t; h_t^d]). \tag{3.42}$$

Equation 3.42 specifies how the function $v(.)$ of equation 3.31 works.

The attentional vector $\hat{h_t^d}$ is then fed through a softmax layer to produce a probability distribution over the classes:

$$p_\theta(y_t|c_t, y_{t-1}, ..., y_1) = softmax(g(\hat{h_t^d})). \tag{3.43}$$

As detailed by Luong et al 2015 [62], there are two classes of attention: global attention and local attention. While global attention considers all the source words, the local one instead, at each time step, looks at a different subset of them.

In particular, global attention considers all the hidden states of the encoder when deriving the context vector c_t (Figure 3.12). In this model type, for sample n a variable length alignment vector α_t, whose size equals the number of time steps of the source sentence, is derived by comparing the current decoder hidden state h_t^d with each encoder hidden state h_s^e, with $t \in 1, ..., T_n$ and $s \in 1, ..., S_n$.

In particular, following the notation proposed by Luong et al 2015 [62], for decoding step t and encoding step s, the alignment between h_t^d and the hidden state h_s^e is:

$$\alpha_{t,s} = align(h_t^d, h_s^e) = \frac{exp(score(h_t^d, h_s^e)}{\sum_{s'=1}^{T_n} exp(score(h_t^d, h_{s'}^e)))}. \tag{3.44}$$

This way the computational path to generate $\hat{h_t^d}$ is:

$$h_t^d \to \alpha_t \to c_t \to \hat{h_t^d}.$$

Bahdanau et al 2014 [4] proposed to use a different computational path to extract $\hat{h_t^d}$. In their version, the alignments are computed this way:

$$\alpha_{t,s} = align(h_{t-1}^d, h_s^e) = \frac{exp(score(h_{t-1}^d, h_s^e)}{\sum_{s'=1}^{T_n} exp(score(h_{t-1}^d, h_{s'}^e))}. \tag{3.45}$$

Resulting in a different computational path to generate $\hat{h_t^d}$, that is:

$$h_{t-1}^d \to \alpha_t \to c_t \to \hat{h_t^d}$$

Both of the above cases have a peculiar interpretation. In the former one, the probability $\alpha_{t,s}$ reflects the importance of the s-th source token to decode the t-th target token, while in the latter it represents the importance that the s-th source token has with respect to the previous hidden state of the decoder h^d_{t-1} in deciding what the next state h^d_t is. One nice property is that the alignment is not considered as a latent variable and so the model can directly compute a soft alignment, which allows the gradient of the cost function to be backpropagated through. This gradient can be used to train the alignment model as well as the whole sequence model jointly.

In the above cases, $score(.)$ is referred as a content-based function that can be implemented in several ways. The way $score(.)$ is implemented determines which attention mechanism is used. In particular, the most famous ones are:

$$score(h_t^d, \bar{h_s^e}) = \begin{cases} h_t^d h_s^e & dot \\ h_t^d W_a h_s^e & \textit{Luong et al } 2015,\ [62] \\ v_a^T tanh(W_a[h_t^d; h_s^e]) & \textit{Bahdanau et al } 2014,\ [4] \\ W_a[s,:]h_t^d & \textit{Luong et al } 2015,\ [62] \end{cases} \tag{3.46}$$

where it is possible to use either h^d_t or h^d_{t-1}. Given an alignment vector of weights, the context vector c_t is computed as a weighted average over the source hidden states:

$$c_t = \sum_{s=1}^{T_n} \alpha_{t,s} h_s^e. \tag{3.47}$$

Local attention

Global attention has the drawback that it has to attend to all words on the source side for each target word, which is expensive and can potentially render impractical to handle long sequences. To address this issue, in 2015 in their *"Effective Approaches to Attention-based Neural Machine Translation"* [62] Luong et al propose a local attentional mechanism (Figure 3.13) that chooses to focus only on a small subset of the source positions per each target word. This model takes inspiration from the trade-off between soft and hard attentional models proposed by Xu et al. 2015 [109] to tackle the image generation task. In their work, soft attention refers to the global attention approach in which weights are placed over all patches in the source image.

The hard attention instead, selects one patch of the image to attend to at a time. While less expensive at inference time, the hard attention model is non-differentiable and requires more complicated techniques such as variance reduction or reinforcement learning to train. Local attention instead selectively focuses on a small window of context and is differentiable. This approach has an advantage of avoiding the expensive computation incurred

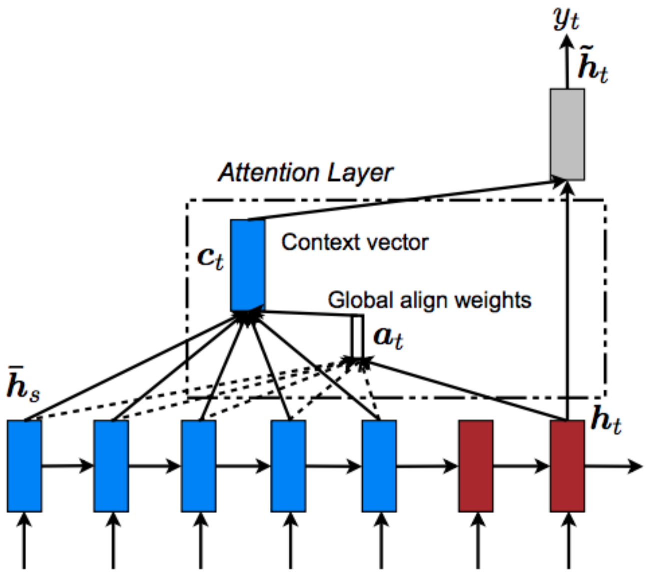


Figure 3.12: Global attention model where at each time step t the model infers a variable length alignment weight vector α_t based on the current decoder state h_t^d and all the source states $\hat{h_s}$. A global context vector c_t is computed as a weighted average. according to α_t. Figure coming from Luong et al 2015, "Effective Approaches to Attention-based Neural Machine Translation" [62]

in the soft attention and at the same time, is easier to train than the hard attention approach.

Concretely, the model first generates an aligned position p_t for each decoding time step t. The context vector c_t is then derived as a weighted average over the set of source hidden states within the window $[p_t - D, p_t + D]$ where D is empirically selected. Unlike the global attention approach, the local alignment vector α_t is now fixed-dimensional and belongs to the space $\mathbb{R}^{2D+1}$. There are two variants of this model:

- **Monotonic alignments**: This variant assumes that source and target sequences are monotonically aligned and sets $p_t = t$. The alignments are computed with equation 3.44.

- **Predictive alignments**: On the other hand, this approach predicts an aligned position:

$$p_t = T_n * sigmoid(v_p^T * tanh(W_p * h_t^d)), \tag{3.48}$$

  where W_p and v_p are model parameters which will be learned to predict positions and S is the source sentence length. As a result, p_t will be $\in [0, T_n]$. To favor alignments to be closed to p_t, they placed a gaussian distribution centered around p_t. Specifically, the alignment weights are now defined as:

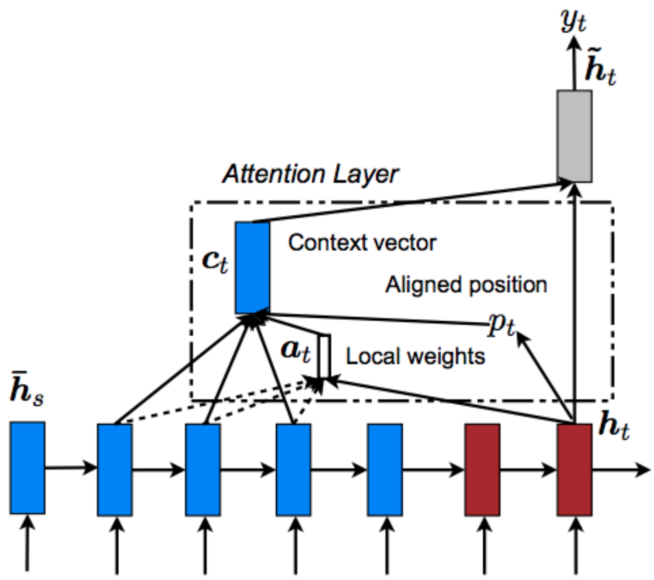


Figure 3.13: Local attention model where the model first predicts a single aligned position p_t for the current target word. Figure coming from Luong et al 2015, "Effective Approaches to Attention-based Neural Machine Translation". [62]

$$\alpha_t = align(h_t^d, h_s^e) * exp\left(-\frac{(s-p_t)^2}{2\sigma^2}\right), \tag{3.49}$$

where $align(.)$ is specified in equation 3.44 and $\sigma = \frac{D}{2}$.

Cost of attention

By looking a bit more closely at the equations 3.44 and 3.45 for global attention can be noticed that this feature comes at a cost. It is necessary to compute an attention value for each combination of input and output word and having a 50-word input sequence and a 50-word generated output sequence would mean to have 2500 attention values. In this case the situation is not too bad, but in other cases the computation can easily explode and become intractable. This is what happens in the character-level computations that deal with sequences consisting of hundreds of tokens. In this situation global attention can become prohibitively expensive.

Actually, that is quite counter intuitive. Human attention is something that is supposed to save computational resources. By focusing on one thing, the human brain can neglect many other things. But that is not really what the above model does. It essentially looks at everything in detail before deciding what to focus on. Intuitively that is equivalent to outputting a

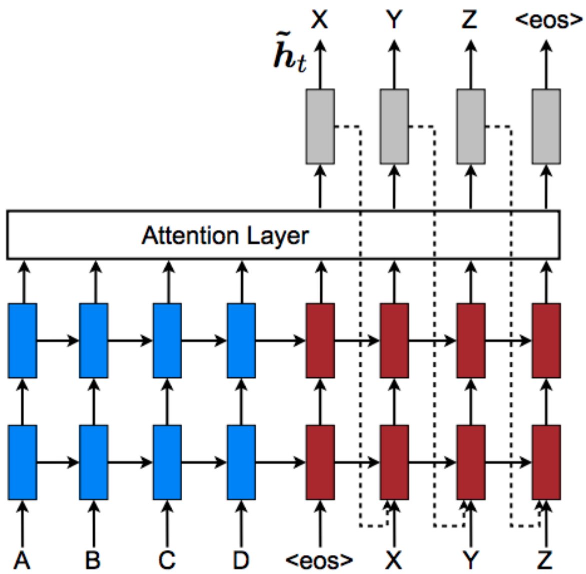


Figure 3.14: Attentional vectors are fed as inputs to the next time step to inform the model about past alignment decisions. The weights α_t are inferred from the current decoder state h_t^d and those source states h_s^e in the window. Figure coming from Luong et al 2015, "Effective Approaches to Attention-based Neural Machine Translation" [62]

translated word, and then going back through all of the internal memory of the text in order to decide which word to produce next.

That seems like a waste, and not at all what humans are doing. In fact, it is more a kind of memory access, not attention. As often happens in deep learning, that has not stopped global attention mechanisms from becoming quite popular and performing well on many tasks, such as machine translation and question answering.

An alternative approach to global attention is either to use Reinforcement Learning to predict an approximate location to focus on or to use the local attention proposed by Luong et al 2015 [62]. That sounds a lot more like human attention, and that is what is done in *"Recurrent Models of Visual Attention"* by Mnih et al 2014 [68].

Revised input feeding

In addition, Luong et al (2015) in *"Effective Approaches to Attention-based Neural Machine Translation"* [62] proposed a different input feeding procedure (Figure 3.14). It works by feeding the attentional vectors $\hat{h}_t^d$ back as new inputs of the decoder. The difference here is that, while in the previous architecture the decoder accepted only the input, now it takes a vector that is the concatenation of the input with the attention vector.

Bidirectional encoder

Usual RNN models read an input sequence x starting from the first symbol x_1 to the last one x_{T_n}. However, it is often necessary to have the encoder hidden state at time s, h^e_s, to be aware of both the preceding and the following words.

Hence, Bahdanau et al (2014) proposed to use a *BidirectionalRNN*, firstly introduced by Schuster et al (1997) [82]. This architecture consists of a forward and a backward RNNs. The forward one reads the input sequence as is it ordered and calculates a sequence of forward hidden states:

$$(\vec{h}^e_1, ..., \vec{h}^e_{T_n}). \tag{3.50}$$

The backward RNN instead reads the sequence in the reverse order resulting in a sequence of backward hidden states:

$$(\overleftarrow{h}^e_1, ..., \overleftarrow{h}^e_{T_n}). \tag{3.51}$$

The final hidden state for input token at position s will be:

$$h^e_s = [\vec{h}^e_s; \overleftarrow{h}^e_s]. \tag{3.52}$$

This way the annotation h^e_s is a summary of both the preceding words and the following words and due to the tendency of RNNs to better model recent inputs the encoder hidden state will contain information related to the words around x_s, resulting in a local information summary.

As it is shown in Figure 3.15, this extension of the encoder is particularly useful when powering the decoder with attention, in fact attending on the encoder states would mean to focus on chucks of the input sequence.

Attention visualization

Nicely, sequence to sequence models empowered with attention allow the developer to visualize what is the network learning. This visualization learning feature is really powerful and it is useful to inspect how the model aligns the generated sequence with the input data. In particular, given an input $(x_1, ..., x_{T_n})$ and a predicted sequence $(o_1, ..., o_{O_n})$, in the scope of global attention it is always possible to retrieve an alignment matrix of shape (O_n, T_n). This alignment matrix contains the result of equation 3.43, that are numbers between 0 and 1. In particular, for each decoding step, that is for each row of the matrix, the elements of the row are computed through a softmax and for this reason the elements in each row sum up to 1. This means that, for alignment matrix AL:

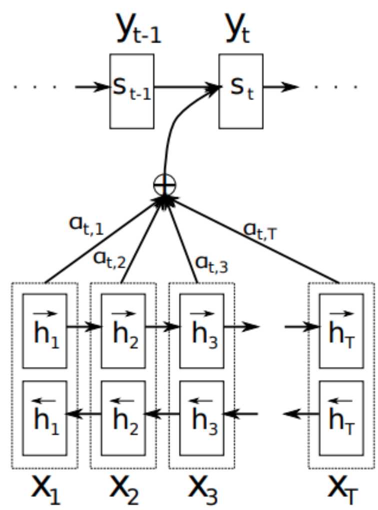


Figure 3.15: The encoder uses BiRNNs to process the input sequence and the decoder exploits attention to extract a dynamic context vector. Figure coming from Bahdanau et al 2014, "Neural Machine Translation by Jointly Learning to Align and Translate" [4]

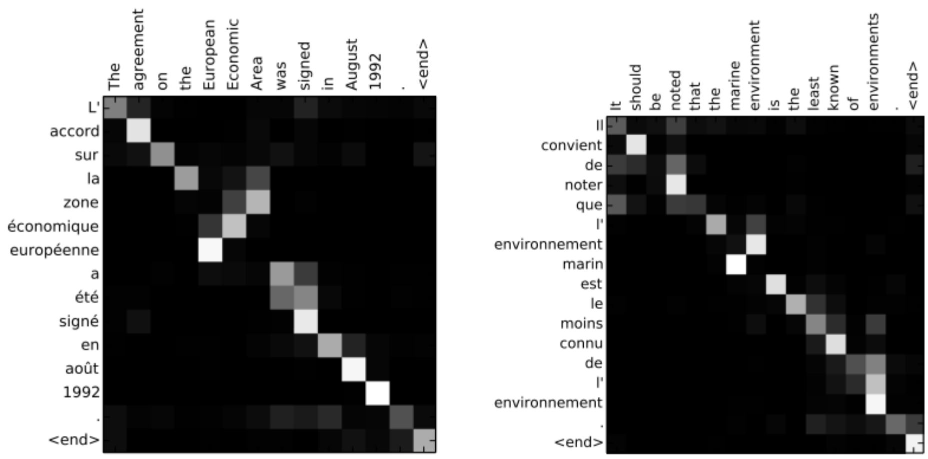
$$\sum_{j=0}^{T_n} AL[i,j] = 1 \; \forall i \in \{0, ..., O_n\}. \tag{3.53}$$

The *hello world* application of these models is usually machine translation. In this case, the input sequence is a sentence in a language, for example English, while the target sequence is another sequence in a different language, that can be French.

Given that, in machine translation, the dependencies are mainly *Directly tok2tok aligned* and *Indirectly tok2tok aligned*, the alignment matrix should be easily interpretable. In fact, some words are translated directly into other words, without the necessity to have information about the sentence itself. This is the case of the translation between *agreement* and *accord* for which intuitively the model, while generating *accord*, should assign an alignment close to 1 to the source word *agreement* and nearly 0 to the alignments with all the other input tokens. This is what happens in the results shown by Bahdanau et al 2014 [4] (Figure 3.16 and 3.17).

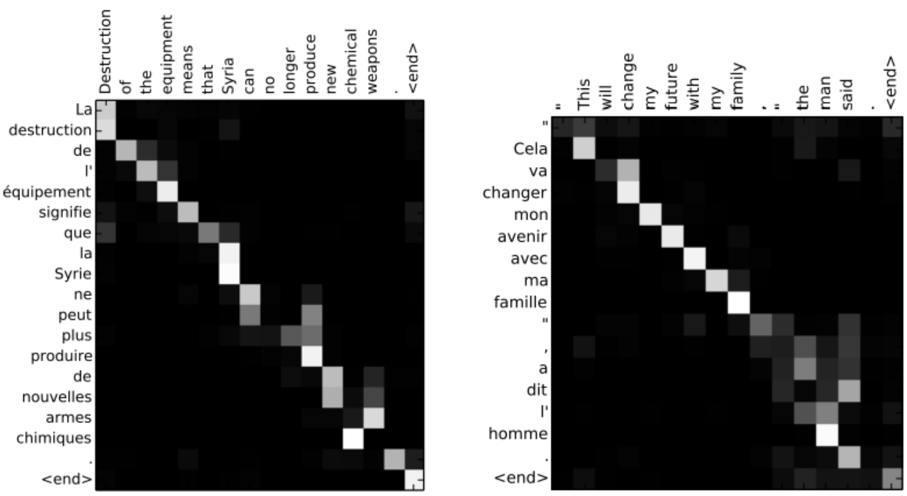
3.2.7 Beyond pure recurrence in Seq2Seq models

In generative chatbots, the conversational context it is meant to be the sequence of utterances exchanged between the actors of the conversation before the actual question that has to be answered. The depth of the context back into the historical conversation is called *num_turns* and is an hyper-



(a) (b)

Figure 3.16: The x-axis and the y-axis of each plot correspond to the words in the source sentence (English) and the generated translation (French). Part 1. Figure coming from Bahdanau et al 2014, "Neural Machine Translation by Jointly Learning to Align and Translate" [4]



(c) (d)

Figure 3.17: The x-axis and the y-axis of each plot correspond to the words in the source sentence (English) and the generated translation (French). Part 2. Figure coming from Bahdanau et al 2014, "Neural Machine Translation by Jointly Learning to Align and Translate" [4]

parameter, usually very difficult to be tuned.

As an example, by using $num\_turns = 2$, the conversational context used to predict the answer a_i would be abstracted this way:

$$context_i = (q_{i-2}, a_{i-2}, q_{i-1}, a_{i-1}, q_i)\,. \tag{3.54}$$

Then, the set of utterances are concatenated to form a long sequence of tokens and are sent to the encoder to be processed in a sequential way.

Concatenating the elements of $context_i$ usually results in a very long sequence of tokens:

$$encoder\_input = t_{i-2,0}, ..., t_{i-2,l_{i-2}}, \#, ..., \#, t_{i,0}, ..., t_{i,l_i}, \tag{3.55}$$

where $t_{h,k}$ can be either the a word or a character, depending on the design choice, and # is a change turn delimiter. This way, the input sequence can potentially be very large and usually there are serious problems for the recurrent models to handle its long term dependencies.

Also, many other text mining problems, such as text classification and text summarization, have to handle a very long input sequence and suffer of the same issue. This problem was explored in depth by Hochreiter in 2001 [39] and by Bengio et al in 1994 [7], who found some pretty fundamental reasons why it might be difficult to learn those dependencies with pure RNN networks. However, in 2018 Trinh et al, with their *"Learning Longer-term Dependencies in RNNs with Auxiliary Losses"* [98], explained that it is possible to capture long term dependencies in RNNs by adding an unsupervised auxiliary loss to the original objective. In practice, the problem still persists, even when using more complex recurrent cells such as *LSTM* [40] or *GRU* [4]. In addition to this convergence issue, standard recurrent models take a long time to train if fed with such long sequences. This is due to the fact that these networks are limited in two aspects: the *sequentiality* of the encoder, that makes it impossible to parallelize the computation of the input, and the *autoregressive* decoder, resulting in very slow inference procedures. This computational issue is very serious, and in real cases it also affects the possibility of doing a strong and wide cross-validation. For this reason, a lot of the research of the last year focuses on finding ways to overcome these two limitations.

To solve this issue, Gehring et al 2017 [31] proposed a way to perform sequence learning by using *Convolutional Neural Networks* (CNNs) in both the encoder and the decoder. As it can be seen in Figure 3.18, the performance achieved by this model is competitive and the time to converge is way less with respect to the fully recurrent *GNMT* (Google Neural Machine Translation) architecture (Wu et al 2016 [105]). Also, this approach is the one currently (June 2018) implemented by the automatic translation feature that is present in Facebook and Instagram.

Another very promising architecture, described in Section 3.2.9, is the one that Vaswani et al presented at NIPS 2017 with their paper *"Attention Is All You Need"* [100], also known as the *Transformer*. As it can be seen in

	BLEU	Time (s)
GNMT GPU (K80)	31.20	3,028
GNMT CPU 88 cores	31.20	1,322
GNMT TPU	31.21	384
ConvS2S GPU (K40) $b = 1$	33.45	327
ConvS2S GPU (M40) $b = 1$	33.45	221
ConvS2S GPU (GTX-1080ti) $b = 1$	33.45	142
ConvS2S CPU 48 cores $b = 1$	33.45	142
ConvS2S GPU (K40) $b = 5$	34.10	587
ConvS2S CPU 48 cores $b = 5$	34.10	482
ConvS2S GPU (M40) $b = 5$	34.10	406
ConvS2S GPU (GTX-1080ti) $b = 5$	34.10	256

Figure 3.18: CPU and GPU generation speed in seconds on the development set of WMT'14 English-French. Figure coming from Gehring et al 2017, "Convolutional Sequence to Sequence Learning" [31]

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Figure 3.19: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost. Figure coming from Vaswani et al 2017, "Attention Is All You Need" [100]

Figure 3.19, the performance of this last model is even better than the one obtained by the previous fully convolutional one. In addition, the training cost is an order of magnitude less, $3.3 * 10^{18}$ FLOPs on both EN-DE and EN-FR, while *ConvS2S* needs $9.6 * 10^{18}$ FLOPs on EN-DE and $1.5 * 10^{20}$ FLOPs on EN-FR, and *GNTM* requires $2.3 * 10^{19}$ FLOPs on EN-DE and $1.4 * 10^{20}$ FLOPs on EN-FR.

If the above two architectures, in order to tackle the fact that recurrent cells are very time consuming, exploit completely RNN-free architectures, some other approaches, called hierarchical sequence models, define fast and competitively performing algorithms by incorporating the notion of hierarchy into their architecture.

In fact, while *the Transformer* is a very general model that can be used to perform a wide variety of tasks, from image captioning to machine translation (Kaiser et al 2017, *"One model to learn them all"* [44]), hierarchical sequence models are specifically made for hierarchically organized data, such as text. In fact, text can be seen as an hierarchically organized input. It is made of characters wrapped into words that together form sentences, placed into a document.

Biasing the traditional sequence to sequence models to efficiently process this hierarchical structure opens the possibility to let the model understand better the input and to make the training process faster and more parallelizable even by using recurrent based models. Two interesting examples have been presented by Yang et al in 2016 in *"Hierarchical Attention Networks for Document Classification"* [110] and by Xing et al in 2017 with their *"Hierarchical Recurrent Attention Network for Response Generation"* [108].

3.2.8 Seq2Seq models with Hierarchical attention

Considering Conversational AI, in 2017 Xing et al published *"Hierarchical Recurrent Attention Network for Response Generation"* [108], a paper describing a sequence to sequence architecture to process conversational multi turn data. In particular, a multi turn input shows a clear hierarchical structure: the context is made of utterances and each utterance is composed of words.

Even if other existing works model the hierarchy of the context, they do not pay enough attention to the fact that words and utterances in the context are differentially important. In the architecture of Xing et al [108] instead, a hierarchical attention mechanism drives the context extraction phase and allows to find the information that is important at word and at sentence level. In particular, with the word level attention and the utterance level attention respectively, the model attends to important parts within and among utterances. With word level attention, hidden vectors of a word level encoder are synthesized as utterance vectors and are fed to an utterance level encoder to construct the hidden representations of the context. These hidden vectors are then processed by the utterance level attention and transformed in context vectors used to decode the response. In practice, building a model that processes multi-turn conversations is complex because of the conversational context is hierarchically organized and not all the parts of the context are equally important to response generation. Words are differentially informative and important, and so are utterances.

State of the art models, such as *HRED* (Serban et al 2016a [87]) and *VHRED* (Serban et al 2016c [87]) focus on modelling the hierarchy of the context, whereas there is only little exploration on how to select important parts of the context, although it is often a crucial step for generating a proper response. Without this step, existing models may loose important

Context	
$\mathbf{u_1}$ (Speaker A)：征男友，160cm的妹子真的找不到男友吗 I want a boyfriend. Why can't a 160cm girl find a boyfriend?	
$\mathbf{u_2}$ (Speaker B)：你找不到一定不是因为160 It's definitely not because you are 160cm.	
$\mathbf{u_3}$ (Speaker A)：我知道脸也是硬伤嘛 Well I know I'm not good-looking	
$\mathbf{u_4}$ (Speaker B)：是你非要175以上 No, it's because you always hit on someone higher than 175cm.	
Response Candidates	
身高不是硬性要求 No, I don't care much about height.	✓
你是男的还是女的啊 Are you a man or a woman?	✗

Figure 3.20: An example of multi turn conversation. Figure coming from Xing et al 2017, "Hierarchical Recurrent Attention Network for Response Generation" [108]

information in context and generate irrelevant responses.

An interesting example is shown in Figure 3.20. The conversational context is about height and boyfriend and so, to generate and answer, words like *girl* and *boyfriend* and numbers indicating height such as *160* or *175* seem to be more important than *not good-looking*.

For this reason, utterances u_1 and u_4 convey main semantics of the context, and are more important than the others for generating a proper response. Without modelling the importance of words and utterances, the state of the art model VHRED (Serban et al 2016c [87]) misses important points and gives a response which is fine with a context made of only u_3, but not that good having the whole context. After paying attention to the important words and utterances, HRAN by Xing et al 2017 [108] generates a better response. In particular, the model proposed by Xing et al (2017) [108] is inspired by the success of the attention mechanism in single-turn response generation (Shang et al 2015 [88]) and proposes a hierarchical attention recurrent model to dynamically highlight important parts of sequences of words and of utterances when generating the response.

To do this, HRAN [108] exploits a word level recurrent neural network that encodes each utterance into a sequence of hidden vectors then the decoder is generating a specific word in the response, a word level attention mechanism assigns a weight to each vector in the hidden sequence of an utterance and forms an utterance vector by linearly combine the vectors where important hidden vectors correspond to important parts in the utterance for the generation of the utterance vector. The utterance vectors are then fed into an utterance level encoder which constructs hidden representations of the context and, different from classic attention mechanism, the word level attention is dependent on both the decoder and the utterance level encoder.

As a third level, an utterance attention mechanism attends to important utterances in the high level sequence and summarizes them as a context vector. Finally, on the top of HRAN [108], a decoder takes the context vector as input and generates the word in the response working as a language model in the same way sequence to sequence decoders do.

Similarly to HRAN [108], other models tackled multi turn response generation in a similar way. For example, in DCGM of Sordoni et al (2015) [92] the context is encoded with a multi layer perceptron (MLP). Also, Serban et al (2016) proposed HRED [86], VHRED [86] and MrRNN [85], where the last two models introduce latent variables into the generation process. Differently from these approaches, HRAN [108] simultaneously models the hierarchy of contexts and the importance of words and utterances in a unified framework.

Practically, another interesting application of hierarchical attention networks is the one of Yang et al (2016) [110]. They proposed to use an architecture, similar to the encoder of HRAN [108], for document classification. In this work they use two levels of attention mechanisms to model the contributions of words and sentences in the classification decision.

Architecture

As it is shown in Figure 3.21, with the two levels of attention, HRAN [108] works in a bottom-up way: the hidden vectors of utterances are processed by the word level attention and uploaded to an utterance level encoder to form hidden vectors of the context, that are further processed by the utterance level attention to form a context vector to be uploaded to the decoder in order to generate the word.

Considering a dataset $D = (U_i, Y_i)_{i=1}^{N}$, each (U_i, Y_i) represents a pair of ground-truth context U_i and response Y_i. In particular:

$$Y_i = (y_{i,1}, ..., y_{iT_i}), \tag{3.56}$$

$$U_i = (u_{i,1}, ..., u_{i,m_i}), \tag{3.57}$$

where $y_{i,j}$ is the j-th target word of sample i, u_{i,m_i} is the direct question to be answered and $(u_{i,1}, ..., u_{i,m_{i-1}})$ is the remaining conversational context. To ensure that there is at least one utterance in the conversational context, it is required to have $m_i >= 2$. Specifically, each utterance $u_{i,j}$ in U can be hierarchically decomposed as:

$$u_{i,j} = (w_{i,j,1}, ..., w_{i,j,T_{i,j}}), \tag{3.58}$$

where $w_{i,j,l}$ is the l-th word in utterance j of sample i. The goal is to estimate the probability:

$$p(y_1, ..., y_T|U). \tag{3.59}$$

By exploiting an historical dataset D, a model of the above probability is fitted with a gradient descend method. Having a new conversational context U, is then possible to generate a novel response $Y = (y_1, ..., y_T)$.

Word level encoder

Given U of equation 3.57, the word level encoder processes each utterance $u_{i,j}$ of sample i in parallel. A bidirectional RNN with Gated Recurrent Units (GRU [4]) is exploited to encode each $u_{i,j} = (w_{i,j,1}, ..., w_{i,j,T_{i,j}})$ $j \in 1, ..., m_i$ into a sequence of hidden vectors $(h_{i,j,1}, ..., h_{i,j,T_{i,j}})$. It is important to notice that each $h_{i,j,l}$ is only dependent on $u_{i,j}$ and its computation does not involve any other utterance in the context U. In particular:

$$h_{i,j,l} = f(u_{i,j}). \tag{3.60}$$

For each $l \in (1, ..., T_{i,j})$ the word level encoder produces an hidden vector $h_{i,j,l}$ by using the following recurrent equation:

$$h_{i,j,l} = concat(\vec{h}_{i,j,l}\ ,\ \overleftarrow{h}_{i,j,l}). \tag{3.61}$$

In particular, the forward GRU computes its hidden state $\vec{h}_{i,j,l}$ by processing the sequence $u_{i,j}$ in its order, from $w_{i,j,1}$ to $w_{i,j,T_{i,j}}$, by using the following equations:

$$z_{i,j,l} = \sigma(W_z * e_{i,j,l} + V_z * \vec{h}_{i,j,l-1}), \tag{3.62}$$

$$r_{i,j,l} = \sigma(W_r * e_{i,j,l} + V_r * \vec{h}_{i,j,l-1}), \tag{3.63}$$

$$s_{i,j,l} = tanh(W_s * e_{i,j,l} + V_s * (\vec{h}_{i,j,l-1} * r_{i,j,l})), \tag{3.64}$$

$$\vec{h}_{i,j,l} = (1 - z_{i,j,l}) * s_{i,j,l} + z_{i,j,l} * \vec{h}_{i,j,l-1}. \tag{3.65}$$

In the above equations, $\vec{h}_{i,j,0}$ is initialized with an isotropic Gaussian distribution and $e_{i,j,l}$ is the embedding of $w_{i,j,l}$. The GRU [4] cell is then parametrized by the matrices W_z, W_r, W_s, V_z, V_r, V_s. In the same way the backward GRU computes its states $\overleftarrow{h}_{i,j,l}$ by processing the sequence $u_{i,j}$ in reverse order from $w_{i,j,T_{i,j}}$ to $w_{i,j,1}$ and is parametrized as the forward GRU. In Figure 3.21, the word level encoder is the purple module on the bottom. There, it can be seen how the different utterances of the context are encoded in parallel for each decoding step. This means that this architecture can be parallelized of a factor equal to the number of utterances in the context.

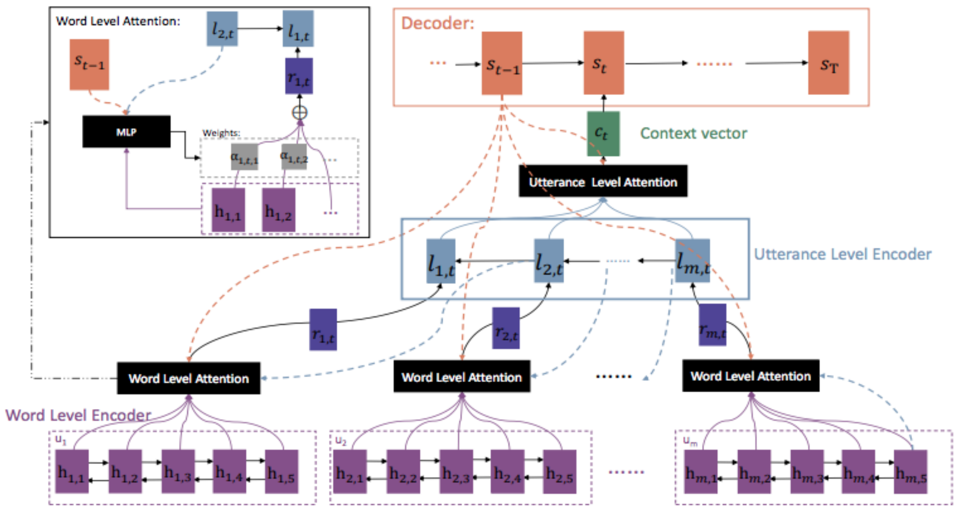


Figure 3.21: Hierarchical recurrent attention network. Figure coming from Xing et al 2017, "Hierarchical Recurrent Attention Network for Response Generation" [108]

Nicely, in this step can be exploited a dynamic batching procedure. To clarify, consider that the batch size is h and each sample is made of k utterances. This way the encoder input would have shape $(h, k, max\_words, d\_model)$. Thanks to the independence of the encoding process, it is possible to reshape the input tensor as $(h*k, max\_words, d\_model)$ and to reformulate the problem into a standard sequence to sequence one. This way, the batch of the encoder will be way bigger than the declared batch size and the computation of the encoder can be performed in parallel. At the end, the output of the encoder will also have shape $(h * k, max\_words, d\_model)$ and after being reshaped back into $(h, k, max\_words, d\_model)$, could be processed by the word level attention.

Word level attention

In the basic version, the word level attention will generate, for each decoding step t, an attention vector $\alpha_{t,i,j}$ for each utterance $u_{i,j}$ in the context U. This vector is computed with equation 3.45, where for each $l \in 1, ..., T_{i,j}$ the encoder's state $h_{i,j,l}$ is compared with $h_{i,t}$. In particular, for each utterance j, the attention vector $\alpha_{t,i,j}$ is computed this way:

$$\alpha_{t,i,j} = (\alpha_{t,i,j,1}, ..., \alpha_{t,i,j,T_{i,j}}), \tag{3.66}$$

specifically with shape $(1, T_{i,j})$. This vector is then used to perform a weighted sum of the word level encoder states, driven by the attention weights. Then, it is extracted an utterance representation $r_{t,i,j}$, as it is specified by equation 3.47. The same is done for all the other utterances in U, and the output of the word attention is:

$$(r_{t,i,1}, ..., r_{t,i,m_i}). \tag{3.67}$$

After stacking it, this set can be seen as a 3D tensor, called WLA_i of shape $(batch\_size, m_i, d\_model)$.

Utterance level encoder

At this step, the representation of each utterance does not know anything about the other sentences that compose the context. It is only aware of the words that compose the specific sentence. To make these vectors also aware of the other parts of the context, an utterance level encoder is exploited. It works as the word level one and can be a bidirectional or an unidirectional encoder. The output of this module, is a set of vectors:

$$(l_{t,i,1}, ..., l_{t,i,m_i}). \tag{3.68}$$

After stacking them, this set can be seen as a 3D tensor, called SLE_i, again of shape $(batch\_size, m_i, d\_model)$, where each sentence vector is aware of the utterance level context.

Utterance level attention

Moreover, an utterance level attention mechanism computes a set of attention weights, each one attached to a contextualized utterance vector. In particular, the attention vector $\beta_{t,i}$ is computed this way:

$$\beta_{t,i} = (\beta_{t,i,1}, ..., \beta_{t,i,m_i}), \tag{3.69}$$

where each attention weight is computed with equations 3.42 and 3.43 by using the tensor SLE_i and the decoder state. Its output has shape $(batch\_size, d\_model)$.

HRAN Xing et al. (2017) [108]

Starting from the basic version described above, Xing et al proposed a similar architecture called *"Hierarchical Recurrent Attention Network"* (HRAN [108]). More specifically, the utterance level encoder is a backward GRU which processes WLA_i backwardly. Also, the word level attention depends on both the hidden states of the decoder and the hidden states of the utterance level encoder. In order to do this, practically, the implementation works in reverse order by first applying this particular variation of the attention layer to the last utterance in the context, that is the set of vectors:

$$\{h_{i,m_i,1}, ..., h_{i,m_i,T_{i,m_i}}\}, \tag{3.70}$$

and then moving towards the first utterance represented as:

$$\{h_{i,1,1}, ..., h_{i,1,T_{i,1}}\}. \tag{3.71}$$

Specifically, the attention weights of equation 3.64 are now computed with a new score function:

$$e_{t,i,j,k} = \eta(h^d_{t-1}, l_{t,i,j+1}, h^e_{i,j,k}), \tag{3.72}$$

$$\alpha_{t,i,j,k} = \frac{exp(e_{t,i,j,k})}{\sum_{o=1}^{T_{i,j}} exp(e_{t,i,j,o})}. \tag{3.73}$$

Equation 3.72 returns the not normalized importance score of the token k in sentence j of sample i and can be seen as another fancy score function to be added to the ones in equation 3.46. Its data flow can be seen in the top left part of Figure 3.21.

It is worth to notice that the word level encoder and the utterance level encoder are dependent one on each other and are alternatively conducted, respectively first the attention then the encoder. This strange data flow makes the computation not parallelizable and this is a difference with the basic version of hierarchical attention. Moreover, $\eta()$ is a multi-layer perceptron with $tanh$ as its activation function. The reason why they added the dependency between the word level coefficient $\alpha_{t,i,j,k}$ and the utterance level state $l_{t,i,j+1}$ is that the content from the context could help to identify important information in utterances, especially when the state h^d_{t-1} is not informative enough.

They also required the utterance level encoder and the word level attention to work reversely because they assumed that, compared to history, conversation that happened after an utterance in the context is more likely to be capable of identifying important information in the utterance for generating proper response to the context.

Hierarchical attention visualization

In the same way of attention, also hierarchical attention can be visualized. This is done by inspecting the values that the model assigns to the normalized attention scores.

Nicely, hierarchical attention can be exploited for a lot of deep learning models, for example document classification and question answering. If Xing et al 2017 [108] proposed to use this mechanism for QAs, Yang et al 2016 [110] used it for document classification. Both of them also showed a visualization of what the network learnt, that is reported in picture 3.22, 3.23, 3.24 and 3.25. In every figure, each line represents an utterance, with blue indicating word importance, red representing utterance importance,

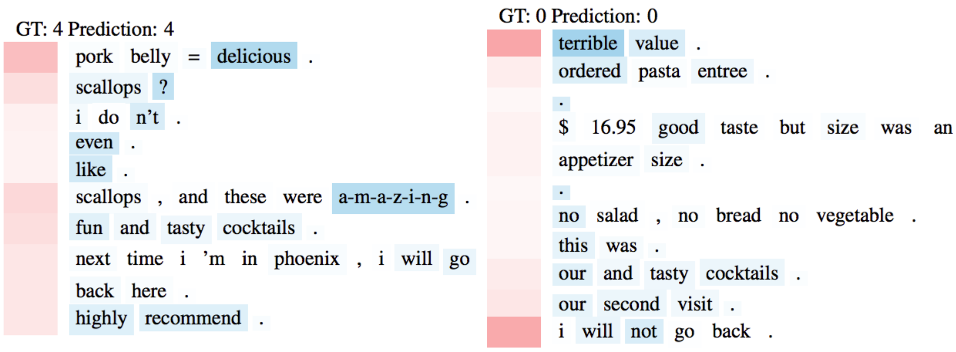


Figure 3.22: Hierarchical recurrent attention network for document classification. Figure coming from Xing et al 2017, "Hierarchical Attention Networks for Document Classification" [110]

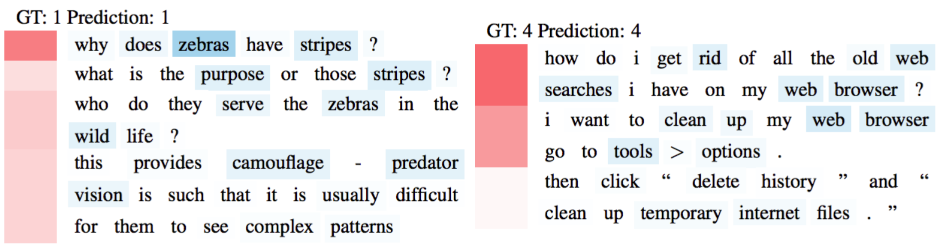


Figure 3.23: Hierarchical recurrent attention network for document classification. Figure coming from Xing et al 2017, "Hierarchical Attention Networks for Document Classification" [110]

and darker colours referring to higher scores. Figure 3.22 shows that a document classification model powered with hierarchical attention can select the words carrying strong sentiment like *delicious*, *amazing*, *terrible* and their corresponding sentences. Sentences containing many words like *cocktails*, *pasta*, *entree* are disregarded.

Note that the model can not only select words carrying strong sentiment, it can also deal with complex across-sentence context. For example, there are sentences like *i don't even like scallops* in the first document of Figure 3.22 that, if looking purely at the single sentence, could be a negative comment. However, the model looks at the context of this sentence and figures out this is a positive review and chooses to ignore this sentence.

The hierarchical attention mechanism also works well for topic classification in the *Yahoo Answer* data set. For example, for the left document in Figure 3.23 with label 1, which denotes Science and Mathematics, the model accurately localizes the words *zebra*, *strips*, *camouflage*, *predator* and their corresponding sentences. In fact for the document on the right, having ground truth label 4, which denotes *Computers and Internet*, the model focuses on *web*, *searches*, *browsers* and their corresponding sentences.

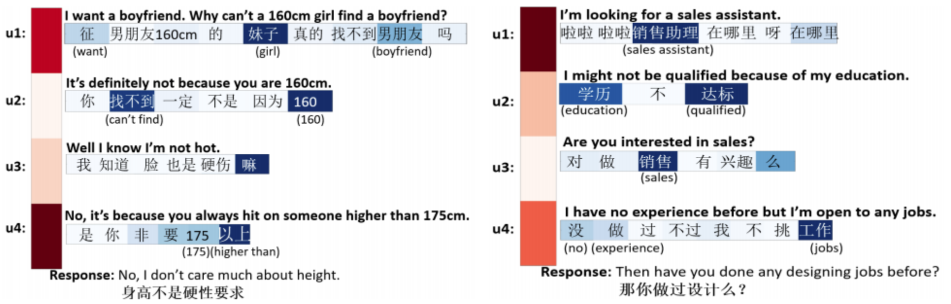


Figure 3.24: Hierarchical recurrent attention network for Conversational AI. Figure coming from Xing et al 2017, "Hierarchical Recurrent Attention Network for Response Generation" [108]

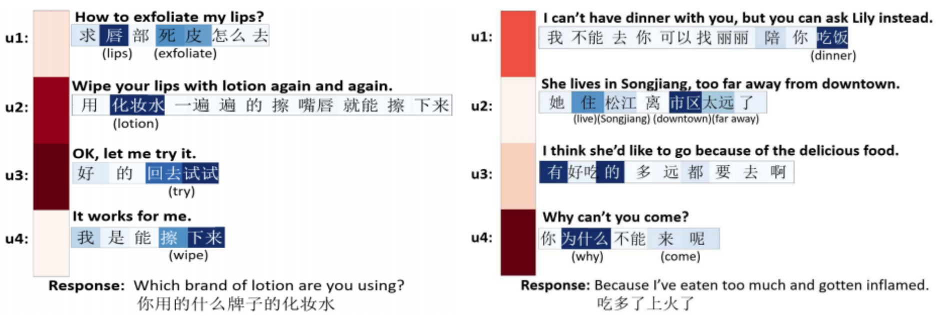


Figure 3.25: Hierarchical recurrent attention network for Conversational AI. Figure coming from Xing et al 2017, "Hierarchical Recurrent Attention Network for Response Generation" [108]

In the first example of Figure 3.24, words including *girl* and *boyfriend* and numbers including *160* and *175* are highlighted, and *u1* and *u4* are more important than the others and the result matches the intuition.

In the first case of Figure 3.25, HRAN [108] assigned a large weight to *u1*, *u4* and to words like *dinner* and *why*. In conclusion, these figures provide insights about how HRAN [108] understands contexts during generation.

Error analysis

Xing et al [108] also nicely investigated how to improve HRAN in the future by analyzing errors. The errors can be summarized as: 51.81% *logic contradiction*, 26.95% *universal reply*, 7.77% *irrelevant response*, and 13.47% others. Most of the bad cases come from universal replies and responses that are logically contradictory with contexts. This is easy to understand as HRAN [108] does not explicitly model the two characteristics.

The result also indicates that, although contexts provide more information than single messages, multi-turn response generation still has the *safe* *response* problem as the single-turn case; in fact attending to important words and utterances in generation can lead to informative and logically consistent responses for many cases, it is still not enough for fully understanding contexts due to the complex nature of conversations. Specifically, the irrelevant responses might be caused by wrong attention during generation. Although this analysis might not cover all bad cases, it sheds light on the future directions, that should cover:

- improving the diversity of responses, for example by introducing extra content into generation like Xing et al. 2017 [107] and Mou et al 2016 [69] did for single-turn conversation.
- modelling logics in contexts.
- improving attention.

3.2.9 Seq2Seq models with Self Attention

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing ones also connect the encoder and decoder through an attention mechanism. In this scope, Vaswani et al (2017) [100] proposed a new network architecture, the *Transformer*, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks showed that their model is both superior in quality and more parallelizable, requiring significantly less time to train.

Further, they also showed that the Transformer generalizes well to non machine translation tasks and works well for limited data. The main aspect that characterizes this architecture is that it relies entirely on an attention mechanism to draw global dependencies between input and output, eschewing recurrence and convolution.

High level overview

As has been said in Section 3.2.7, the deep learning community is heading in the direction of reducing sequential computation in sequence to sequence models. To this end, several models have been designed, for example *Extended Neural GPU* [45], *ConvS2S* [31] or *Bytenet* [46].

The above architectures have in common the fact that they use convolutional neural networks (CNNs) as basic building blocks, computing hidden representations in parallel for all input and output positions. In these models, the number of operations required to relate signals grows with the distance between positions, linearly for *ConvS2S* [31] and logaritmically for *Bytenet* [46], and this makes more difficult to learn long dependencies between distant positions. The *Transformer* [100] instead, reduces the number of operations to a constant number.

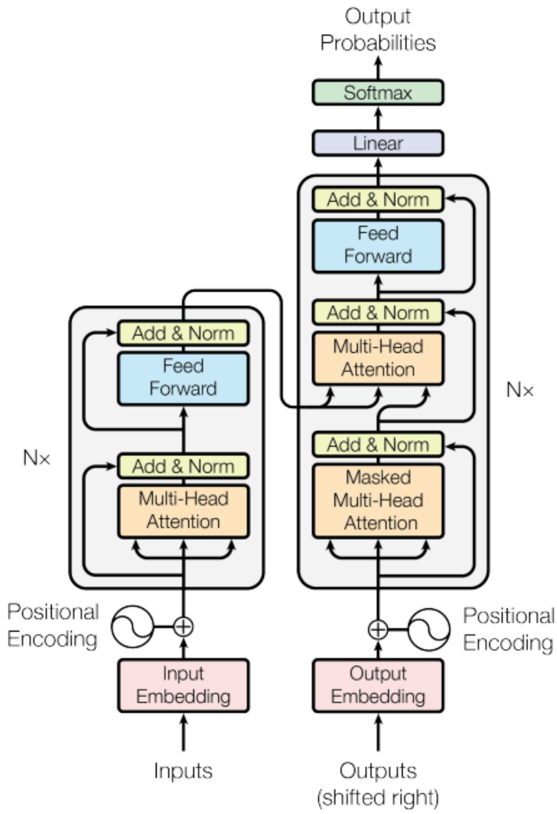


Figure 3.26: Transformer model architecture. Figure coming from Vaswani et al 2017, "Attention is all you need!" [100]

In fact, this model is based solely on self-attention, as the name of the paper, *"Attention is all you need !"* suggests. Self attention is an attention mechanism relating different positions of a single sequence, in order to compute an aggregated representation. It has been successfully used in a variety of tasks including reading comprehension [18], abstractive summarization [72], textual entailment [73] and learning task-independent sentence representation [57]. The *Transformer* is the first sequence to sequence architecture relying entirely on self-attention to compute representation of its input and output, without using sequence aligned RNNs or convolution.

As explained from Section 3.2.5 on, most competitive neural sequence transduction models have an encoder-decoder structure where the encoder maps an input sequence of symbol representations $(x_1, ..., x_n)$ to a sequence of continuous representations $z = (z_1, ..., z_n)$. Given z, the decoder then generates an output sequence $(y_1, ..., y_m)$ of symbols one element at a time. At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next. The *Transformer* follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, as it is shown in the left and right halves of Figure 3.26, respectively.

Positional embeddings

Since this model contains no recurrence and no convolution, in order to inject the notion of word order, it is necessary to provide some information about the relative or absolute position of the tokens in the sequence.

To this end, Vaswani et al (2017) [100] added *positional encodings* to the input embeddings at the bottoms of the encoder and decoder stacks. This interesting trick allows the model, completely unaware of the order of the tokens in the sequence, to make pure word embedding vectors aware of the position in the sequence. In particular, positional encodings have the same dimension d_{model} of the embeddings, so that the two can be summed. There are many choices of positional encodings, learned and fixed [31]. In their work, the choice was to use sine and cosine functions of different frequencies:

- $PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}}))$
- $PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}}))$

where *pos* is the position in the sequence and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid.

Encoder

The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a fully connected feed-forward network with a residual connection [36] around each of the two sub-layers, followed by layer normalization [3]. That is, the output of each sub-layer is $LayerNorm(x+Sublayer(x))$, where $Sublayer(x)$ is the function implemented by the sub-layer itself.

Decoder

The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack.

Similar to the encoder, residual connections are used around each one of the sub-layers, followed by layer normalization. They also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

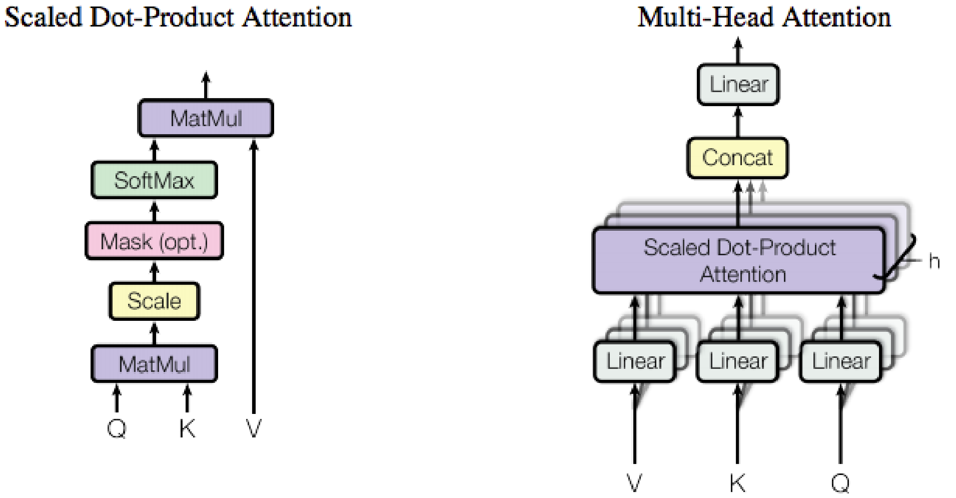


Figure 3.27: Scaled Dot-Product Attention. (right) Multi-Head Attention (left) consists of several attention layers running in parallel. Figure coming from Vaswani et al 2017, "Attention is all you need!" [100]

Attention

In general, an attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, the keys, the values, and the output are all vectors.

The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function (Equation 3.46) of the query with the corresponding key. In particular, the attention mechanism of Vaswani et al [100] is fancy because it performs in parallel different kinds of attention on different versions of the same sequence. Specifically, they use:

- **Scaled dot product attention:** It is a particular variation of multiplicative Luong attention [62], called *Scaled Dot-Product Attention*, Figure 3.27. The input consists of two sets of queries and keys of dimension d_k, and a set of values of dimension d_v. As was explained in Section 3.2.6, it is performed the dot products of the query with all keys. After that, each element in the vector is divided by $\sqrt{d_k}$, and a softmax function is applied to obtain the set of weights α.

  In practice, they compute the attention function on a set of queries simultaneously, packing them together into a matrix Q. The keys and values are also packed together into matrices K and V. The matrix of outputs is then computed as:

$$Attention(Q, K, V) = softmax(\frac{QK * T}{\sqrt{d_k}})V. \tag{3.74}$$

The two most commonly used attention functions are additive attention [4] and dot-product attention [62]. Dot-product attention is identical to Vaswani et al [100] algorithm, except for the scaling factor of $\sqrt{d_{model}}$.

- **Multi head attention:** Another trick they implemented is that, instead of performing a single attention function with d_{model}-dimensional keys, values and queries, they found it beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_q, d_k and d_v dimensions, respectively.

  On each of these projected versions of queries, keys and values they then perform the attention function in parallel, yielding d_v-dimensional output values. These are then concatenated and once again projected, resulting in the final values, as depicted in Figure 3.27. Conceptually, multi-head attention allows the model to jointly attend to information coming from different representation subspaces.

  $$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^o, \tag{3.75}$$

  where:

  $$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V), \tag{3.76}$$

  where $W_i^Q \in R^{d_{model}, d_k}$, $W_i^K \in R^{d_{model}, d_k}$, $W_i^V \in R^{d_{model}, d_v}$ and $W_o \in R^{hd_v, d_{model}}$ They used $h = 8$ and $d_k = d_v = d_{model}/h = 64$ and due to the reduced dimensionality of each head, the total computational cost is similar to the one of single head attention.

3.3 Data preparation for Generative models

Considering a mini-batch training procedure, for each training step t the model consumes a batch of tokenized input and target sequences:

$$raw\_batch_t = \{wy_{1n}, ..., wy_{Sn}, wx_{1n}, ..., wx_{Tn}\}_{n=1}^{batch\_size},$$

where:

- wy_{ij} represents the i-th word in the target sequence of sample j
- wx_{ij} represents the i-th word in the input sequence of sample j

By holding an input vocabulary and a target vocabulary that assign an id to the set of the top-k most common words in the input and target corpora, each wy_{ij} and wx_{ij} is mapped to an id, respectively y_{ij} and x_{ij}. If it is necessary to assign an id to a word that is not present in the vocabulary the UNK token is used, a special element defined to handle out of vocabulary tokens, called OOV.

Furthermore, it is necessary to define two other special tokens:

- SOS, that marks the start of a sequence and its id is SOS_{id}.
- EOS, that marks the end of a sequence and its id is EOS_{id}.

After translating the sequences from text tokens into ids, the training batch has the following form:

$$batch_t = \{y_{1n}, ..., y_{Sn}, x_{1n}, ..., x_{Tn}\}_{n=1}^{batch\_size}.$$

From this object, three elements are extracted:

- **The augmented input sequence**:

$$\{x_{1n}, ..., x_{Tn}, EOS_{id}\}_{n=1}^{batch\_size}.$$

- **The augmented target input sequence**:

$$\{SOS_{id}, y_{1n}, ..., y_{Sn}\}_{n=1}^{batch\_size}.$$

- **The augmented target output sequence**: This is the list of the labels and, given that the decoder works as a language model, it is made of the shifted decoder inputs:

$$\{y_{1n}, ..., y_{Sn}, EOS_{id}\}_{n=1}^{batch\_size}.$$

The model is then fed with a batch object containing these elements:

- **Encoder inputs:** This object holds $\{x_{1n}, ..., x_{Tn}, EOS_{id}\}_{n=1}^{batch\_size}$. It has the form of a list of lists, where the first list has length $batch\_size$ and the inner lists have lengths T_n for $n \in (1, ..., batch\_size)$. To feed this heterogeneous data structure to the model input data has to be organized it in a tensor. This tensor will have shape $(batch\_size, max(T_n))$ and will contain the input word ids assigned by the input vocabulary. Intuitively, in order to fit in the above shape, some input sequences in the batch will be padded until $max(T_n)$ with a token id of another special token, PAD. This token abstracts the concept of padding. Given that $lookup(word\_id)$ returns a vector of shape $(1, d_{model})$, looking up all the ids contained into the 2D tensor results in having the encoder input tensor with shape $(batch\_size, max(T_n), d_{model})$.

- **Encoder lengths:** In order to not perform useless computation, that is to not effectively process those PAD_{id} tokens, it is needed to provide the model with the list of the effective lengths of the sequences wrapped in encoder inputs, that is the number of *non-pad* tokens per each input sequence in the batch. This object has shape $(1, batch\_size)$.

- **Decoder inputs:** This is the object that holds the input to the decoder. It wraps $\{SOS_{id}, y_{1n}, ..., y_{Sn}\}_{n=1}^{batch\_size}$ in the same way the encoder inputs does. For this reason it has 3D shape $(batch\_size, max(S_n), d_{model})$.

- **Decoder outputs:** This object holds the expected output that the decoder should generate. It is the label set for the current batch and it has 3D shape $(batch\_size, max(S_n), d_{model})$.

- **Decoder lengths:** It has the same role of the encoder lengths and it has shape $(1, batch\_size)$.

For, the effective vocabulary will be made of the top-k most common unique words in the dataset augmented by the four special tokens: UNK, PAD, SOS and EOS.

Training and inference

There is a big difference between the training and the inference phase. While during training the model is fed with encoder inputs, decoder input and decoder output, during inference the only available data are the encoder inputs. This is due to the fact that training can be seen as an off line procedure in which the target sequence is already known. During inference instead, only the input sequence is available. For this reason, *Sequence to Sequence models* ([19], [95]) need to be able to work in two modalities. In particular, considering the base case of using it for machine translation, its training mode can be visualized in picture 3.28:

The architecture of Figure 3.28 uses a multi layer recurrent network both for the encoder (blue) and for the decoder (red).

To summarize, the basic building blocks of a sequence model in training phase are:

1. **Initialization:** The initial states of the layers are initialized as zero-vectors and embedding matrices are used in both encoder and decoder and can be either initialized as 0 tensors, or can be pre-trained with a word2vec model.

2. **Encoder-Decoder:** The encoder does not produce any output, while the decoder produces an output per time step that is then projected over the classes space.

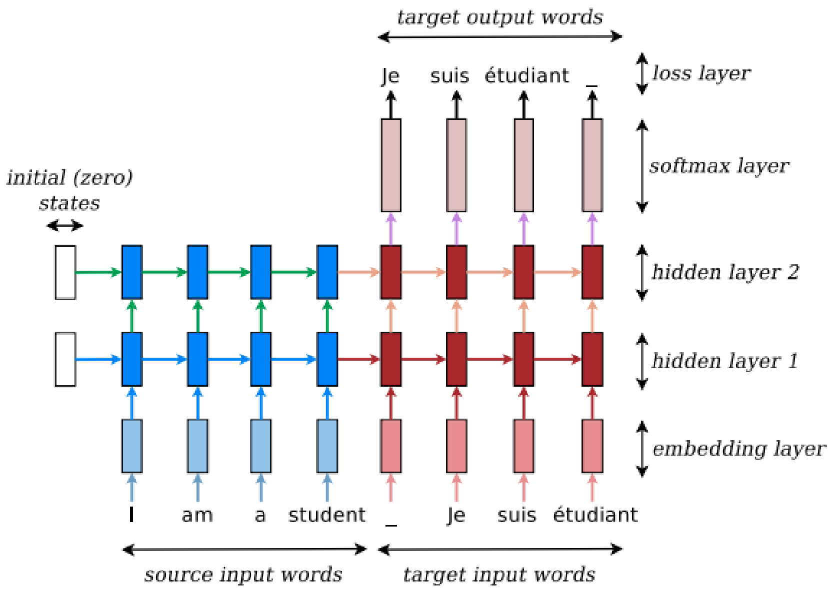


Figure 3.28: Sequence to Sequence model in training phase used for machine translation. Figure coming from Loung thesis, "Neural Machine Translation". [96]

3. **Loss:** A loss value is computed by applying a function to the predicted probabilities for the correct classes.

4. **Stopping criteria:** The decoding process ends when it has generated a number of tokens equal to the number of elements in the target sequence.

While the encoder architecture is independent on having or not the correct target sequence, the decoder architecture needs the target sequence to work. For this reason, during inference a new architecture has to be defined. The simplest strategy is to perform greedy decoding, as it is shown in picture 3.29.

In particular:

- **Encoder-Decoder:** The source sentence is encoded in the same way as in the training process. The decoding process is started as soon as a start of sentence marker SOS is fed as a decoder input.

- **Greedy-Decoding:** For each time step on the decoder side, the most likely word is picked as a greedy choice. For example, in the picture 3.29, for the first time step the greedy choice is *moi* given that it has the highest translation probability. Then, the just generated word is used as the input for the next time step.

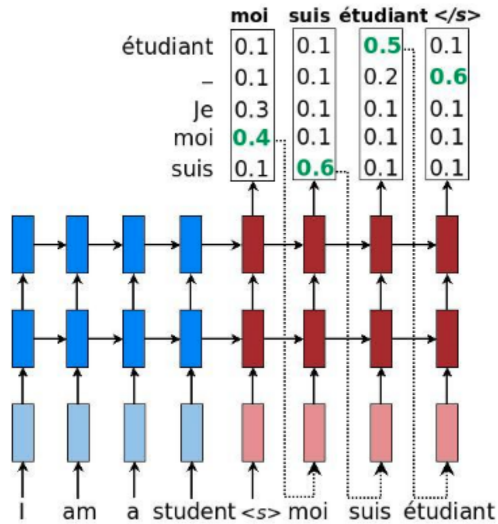


Figure 3.29: Sequence to Sequence model in inference phase used for machine translation. The network performs greedy decoding. Figure coming from Loung thesis, "Neural Machine Translation". [96]

- **Stopping criteria:** The decoding process continues until the end-of-sentence marker *EOS* is produced as an output symbol.

What makes inference different from training is that, while during training correct target words are always fed as inputs, during testing instead the model uses the words it has predicted in an autoregressive way. Several studies have been done on removing this autoregressive property.

Without it, the model could produce its outputs in parallel, allowing an order of magnitude lower latency during inference.
Among them, the most interesting one is *"NON-AUTOREGRESSIVE NEURAL MACHINE TRANSLATION"*, published by Gu et al in 2017 [34].

3.4 Training Generative models

In Sequence to sequence models the output of the decoder, starting from the input, is differentiable, and gradient-based algorithms can be exploited to estimate the parameters of the model.

Considering the most traditional gradient method, gradient descend, there are three variants of it. They differ in how much data is used to compute the gradient of the loss function. Depending on the amount of data, a trade-off is made between the accuracy of the parameter update and the time it takes to perform an update step.

A very interesting study about this trade-off was presented by Ruder in 2016 in his *"An overview of gradient descent optimization algorithms"* [78].

In particular, there are three approaches:

- **Batch gradient descend** computes the gradient of the cost function with respect to the parameters θ for the entire training dataset.

$$\theta(t) = \theta(t-1) - \eta * \nabla_\theta J(\theta|\{x_j, y_j\}_{j=1}^N),$$

  where:

  - $\theta(t-1)$: It represents the values of model parameters used to compute the current value of the error with the current set of weights, that is $J(\theta|\{x_j, y_j\}_{j=1}^N)$
  - ∇_θ: It represents the derivative of the error function with respect to the model parameters. The goal is to find the minimum of the error function, and the derivative is an indication of the direction in which the function is decreasing.
  - η: the learning rate, it specifies how much the optimization follows the derivative direction.

  Given that calculating the gradients for the whole dataset would result in just one update, batch gradient descent can be very slow and is intractable for datasets that do not fit in memory.

- **Stochastic gradient descend** in contrast performs a parameter update for each training example x_i and label y_i.

$$\theta(t) = \theta(t-1) - \eta * \nabla_\theta J(\theta|x_i, y_i).$$

  SGD performs frequent updates with a high variance and that cause the objective function to fluctuate heavily.

- **Mini-batch gradient** descend finally takes the best of both worlds and performs an update for every mini-batch of n training examples.

$$\theta(t) = \theta(t-1) - \eta * \nabla_\theta J(\theta|\{x_j, y_j\}_{j=i}^{i+B}).$$

  By adopting this technique, the model is dependent on another hyper-parameter that is the *batch_size*. While the other hyper-parameters are mainly setting the complexity, batch size, together with the learning rate, are particularly important for the correctness and the speed of the optimization procedure. A very interesting study about how to handle their values has been published by Smith et al in 2017 in *"Don't Decay the Learning Rate, Increase the Batch Size"*[90].

Given that SGD has trouble navigating areas where the surface curves much more steeply in one dimension than in another, researchers have found several other optimizers that could improve the convergence. In particular it is worth to point out:

- *Momentum optimizer* [74].
- *Adam optimizer* [49].
- *RMSProp optimizer* [38].

Chapter 4

Case Study and data preprocessing

In order to provide a real world example of how to apply generative models in Conversational AI, our case study refers to a project developed together with *Loop AI Labs*, an American text mining company that provided us with the dataset and that supported us during the design phase.

We developed a project with the goal of inspecting the performance of a generative chatbot trained with an historical conversational knowledge base made of real world interactions. The historical dataset we were provided with came from an Italian service provider company, called here *DICIOS*, and was the result of the properly anonymized recording of all the conversations that one of their chatting centers in the north or Italy produced during one year of activity, from *January 2016* to *December 2016*.

Being a completely unsupervised list of conversations between users and agents coming from real world interactions, data used in this case study is dirty, irregular and it touches a lot of different topics. Given that we wanted to help the job of the model by feeding data as much clean as possible, we spent a huge amount of work to standardize the conversations, designing a custom preprocessing pipeline and showing the pros and cons of each normalization step.

Data pre-processing is an important step in the data mining process. The phrase *garbage in, garbage out* is particularly applicable to data mining, in fact feeding a model with irregular data will usually succeed in bad performance. If there is irrelevant and redundant information or noisy and unreliable data, then knowledge discovery during the training phase is more difficult. Data preparation and filtering steps can take considerable amount of processing time and they include cleaning, instance selection, normalization, transformation, feature extraction and selection. The product of data pre-processing is the final training set.

The objectives of this chapter are both the ones of framing our case study,

TYPE	Single utterances
AMMINISTRATIVO INFO	950667
AMMINISTRATIVO SALDO	496725
ASSISTENZA TECNICA	1649416
CHAT PREVENTION	1335205
CHAT PREVENTION DWG	247964
CHAT UPSELLING CAR	255551
CHAT UPSELLING INFO	8484
CHAT UPSELLING KO	73854
CHAT UPSELLING OK	516076
NO VALUE	917840
PROMO PARITY CMN	339699
LOGIN	471531
RECUPERO ID	141988
TRASLOCA CMN	183
TRASLOCA INFO	110026

Table 4.1: DICIOS dataset.

this is presented in Section 4.1 about data exploration, and of describing how to design a preprocessing pipeline able to extract, from a set of real world irregular conversations, a clean and regular training set to be fed to a generative model among the architectures exposed in Chapter 3. This second part is treated in Section 4.2.

4.1 Data exploration

The dataset is encoded in fifthteen files, each one containing conversations written in Italian and relating the different issues that a customer service has to solve in the scope of a service provider company.

For instance, helping the user to change the password, supporting the user in the remodulation of his plan or opening a request for a substitution of a physical hardware platform.

With different files containing different kinds of conversations, as a first step we merge the files of Table 4.1 into a single big unsupervised dataset, and we do this with the goal of increasing the support of the statistics that the preprocessing methods use to standardize the conversations.

As a starting point, we extract a vocabulary from the merged dataset and we find out $970k$ unique words and $68M$ total words, and this gives an idea of the complexity and of the entropy of the dataset. To visualize this entropy, in Figure 4.1 we show the absolute frequencies of the $20k$ most frequent tokens with their counts. The *x-axis* represents the id in the sorted vocabulary while the *y-axis* represents the corresponding count. We can see

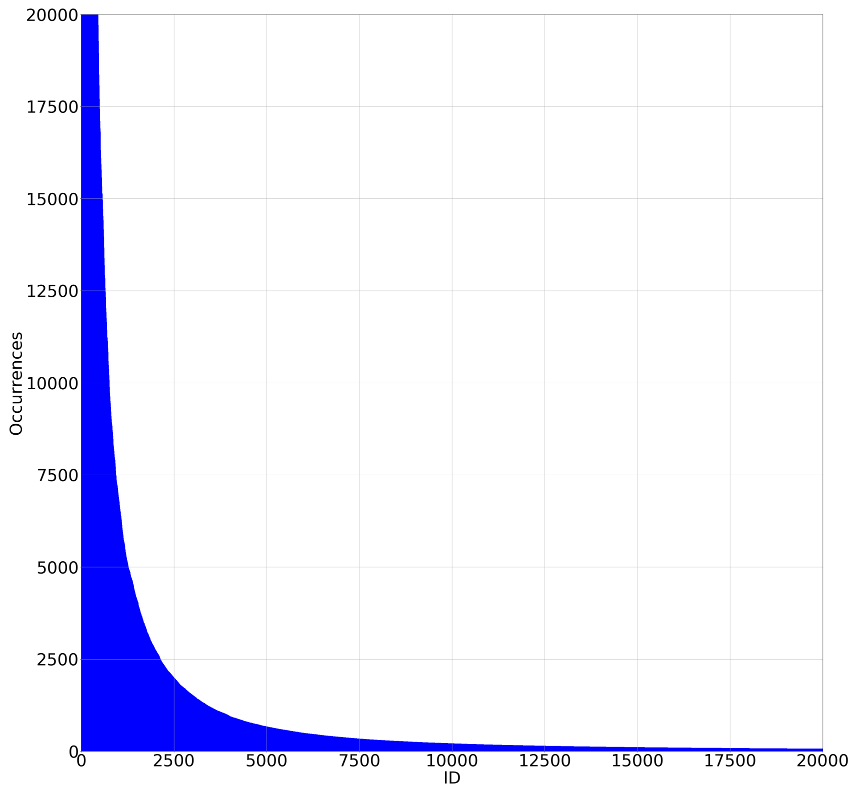


Figure 4.1: On the y-axis there are the occurrences of the top-$20k$ most frequent words in the vocabulary extracted from the raw and not preprocessed text.

how the count quickly drops down to stay constant in the long tail, meaning that the dataset is divided into two chunks, very frequent words and very rare words. We can also notice that, among the $970k$ unique words, only $5k$, that is the 0.005% of the total, has a count that is more than $1k$, and this clarifies how much crucial is data normalization to make this data usable and meaningful.

If in the previous analysis, Figure 4.1, we discussed how the most frequent tokens are distributed, it is also interesting to check how much sparse is the long tail of the vocabulary, that means inspecting the percentage of tokens that have a very low count, their nature and so on, and we show this in Figure 4.2, showing the cumulative frequencies.

To generate the plot shown in Figure 4.2 we first extract the vocabulary voc, that is a big vector of size V, where V is the number of unique tokens in the raw text. After assigning to each unique token its count, we organize the unique words in a list called *voc_list*, sorted by count, where each element i of the vector *voc*, that is $voc[i]$, now represents the count for token $voc\_list[i]$. From this traditional vocabulary formulation, in Equation 4.1 we define the cumulative count for token $voc[i]$, $voc\_cum[i]$, and in Figure 4.2 we show $voc\_cum[i]$, for each i.

$$voc\_cum(voc\_list[i]) = \sum_{k=0}^{i} voc[k]. \tag{4.1}$$

Specifically, cumulative count for word i, $voc\_cum[i]$, represents how much of the corpus will be covered by taking a vocabulary of maximum length equal to i.

Given the dataset has to be parsed into the conversational samples that are then fed to a generative model, there are few crucial aspects to be considered, such as *vocabulary distribution*, *length of the conversational context*, and *unknowns percentage*.

4.1.1 Vocabulary size

For tasks that use the output of a preprocessing stage, such as generative models, it is necessary to choose a vocabulary size k, then all the tokens that are not present in the vocabulary are considered as unknowns; this cut has to be performed in a way that the most of the corpus is still covered by the vocabulary. To ensure this property, it is often required to use a very big vocabulary size k, and this results in an huge number of variables to be estimated, a not desirable property.

To reduce the parameters of the model, there are several vocabulary reduction policies that can be adopted, such as word and character level cleaning. These techniques aim to shrink the vocabulary and to make it more meaningful. For example, with $970k$ distinct tokens the cumulative frequencies of Figure 4.1 show that, by using a vocabulary made of the top $20k$ most frequent tokens, $3M$ of tokens will be unknown. The goal of vocabulary reduction techniques is to allow to cut as much as possible the vocabulary and at the same time to retain a good corpus coverage.

In table 4.2 we show the effect of retaining a vocabulary made of the top-k most frequent tokens. In particular, we point out the *tot cov*, that is the percentage of total words in the dataset covered by the specific vocabulary cut, and the *voc cov*, that is the percentage of unique words retained. In addition, we extract the k-th most frequent word in the vocabulary, the *sample* column, together with its count, the *count* column, in order to give an example of how the vocabulary is composed as the depth increases.

From Figure 4.2 and Table 4.2, we can notice that:

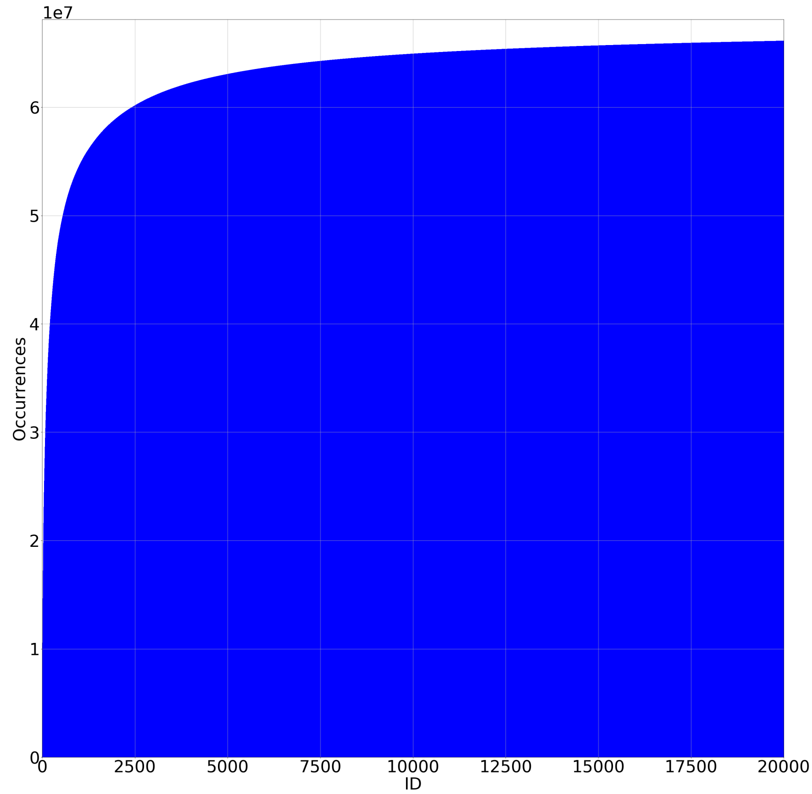


Figure 4.2: Cumulative frequencies of the first $20k$ tokens in the vocabulary, showing on the y-axis the corpus, in steps of ten millions, covered by a vocabulary holding the x most frequent tokens. It can be seen how the top-$20k$ tokens roughly cover the totality of the data, that is $68M$ total words.

- The corpus coverage, *tot cov*, increases quickly to reach a plateau at roughly 96%. This means that there is a very long tail that does not allow to increase the corpus coverage by adding new words to the vocabulary because the count, last column, decreases with a very fast rate.

- The vocabulary coverage, *voc cov*, tells us that the tail is very long and it is possible to understand why by inspecting the samples in column *sample* of Table 4.2. In fact, as the depth increases, the tokens become badly punctuation splitted, not lower-cased and in general not normalized. This made us understand how much necessary was to normalize the text. The long tail was mainly made of spell typos of

Top-k	n words	tot cov	voc cov	sample	count
50	25M	0.36%	0.0001%	contratto	176k
100	32M	0.46%	0.0001%	tua	106k
500	49M	0.72%	0.0005%	attiva	17,6k
1k	54M	0.80%	0.0010%	attesa	7k
2k	59M	0.86%	0.0021%	costi,	2,7k
3k	61M	0.89%	0.0031%	Considera	1,5k
4k	62,2M	0.91%	0.0041%	e-mail.	966
5k	63M	0.92%	0.0052%	comodamente,	674
5,5k	63,3M	0.93%	0.0057%	AVREI	579
7k	64M	0.941%	0.0072%	l'attivazione,	387
8k	64,4M	0.946%	0.0082%	pagarne	308
10k	64,9M	0.953%	0.0103%	SOFT	214
12k	65,3M	0.959%	0.0124%	storia,	157
13k	65,4M	0.961%	0.0134%	?!,	138
14k	65,5M	0.963%	0.0144%	anticipato,	122
∞	68,1M	1,00%	1,00%	–	–

Table 4.2: For each vocabulary cut k it is shown the corpus coverage (n words and tot cov) and the vocabulary coverage (voc cov). In addition, it is extracted the k most frequent word with its count, in order to qualitatively inspect the vocabulary at different depths. It can be noticed how common words are clean and regular, while as the depth is increased errors, typos and irregular words appear. In fact, the tail of the vocabulary is made of not normalized tokens, and refers to roughly 99.9995% of the unique words. This shows how much required is a normalization.

normal tokens and this phenomenon made the vocabulary size explode, and consequently any NLP task pretty much impossible.

4.1.2 Length of conversational context

Another important value to be monitored is the length of the conversational context. In Chapter 3 we showed how generative models suffer in handling long sequences because of their long term dependencies. For this reason we adopted a technique to abstract long repeated expressions into place-holder tokens.

In particular, in Figure 4.3 we present the distribution of the lengths of single utterances in the dataset. On the y-axis we present the absolute percentage while on the x-axis there are the different values of the lengths. There are $7M$ single utterances and their lengths are distributed with a mean of 10.28, a variance of 12.2, a maximum value of 1882 and a minimum one of 2. The percentiles are: 4 for the the 25-th, 6 for the 50-th, 12 for the 75-th, 17 for the 85-th, 23 for the 90-th and 33 for the 95-th. Also, from this figure we can see how this distribution is irregular and typical of a real world case.

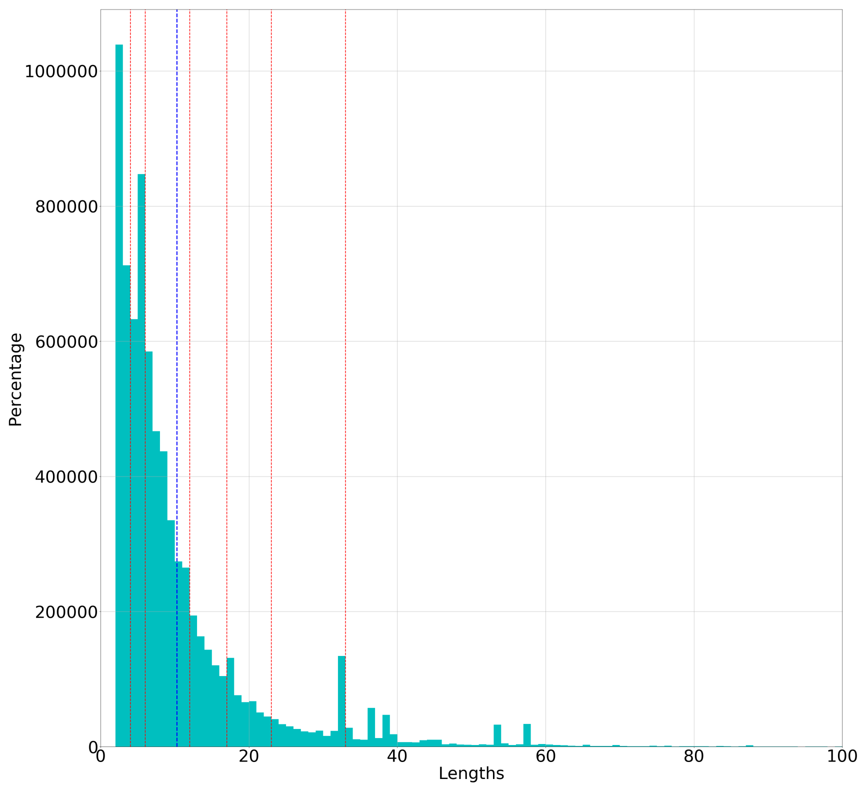


Figure 4.3: The distribution of the lengths, showing on the y-axis the absolute percentage.

Having outliers that are far away from the main percentiles means that during training these samples will be batched together with normal ones causing an huge amount of padding to be processed and this usually causes memory errors and computation to be wasted. For this reason we had the necessity of finding out a way to normalize the length of the utterances.

4.1.3 Unknown percentage

The last property that is required for a conversational dataset is to have a low unknowns percentage. This can be achieved by making use of a spell checker, a software module that, by exploiting unsupervised statistics, is able to understand when a word is misspelled or if it is the result of two words merged together.

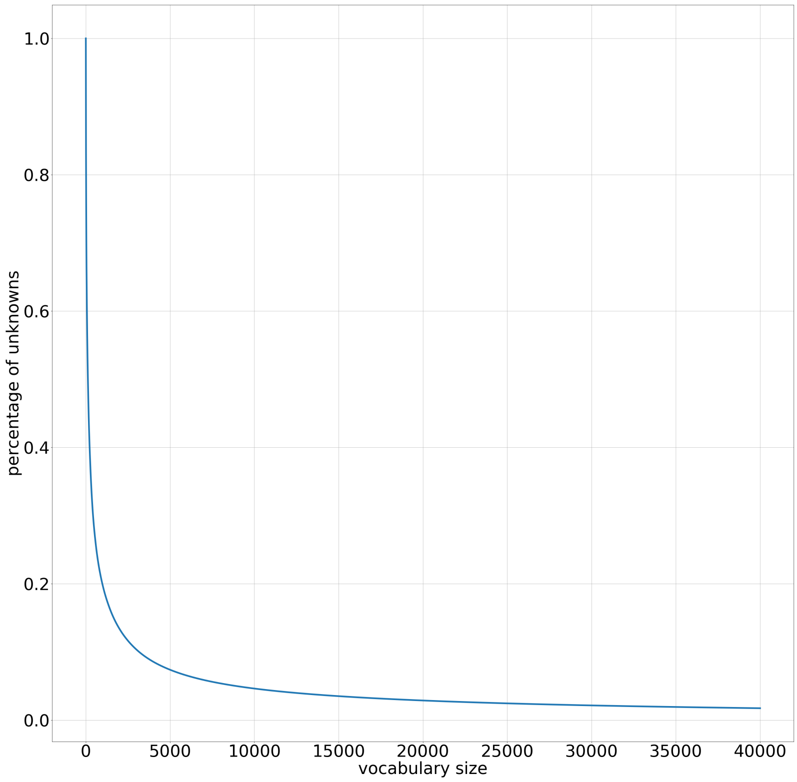


Figure 4.4: The unknown relative percentage per vocabulary size computed on the not normalized dataset. It can be seen how for small vocabulary sizes, such as $10k$ or $15k$, the unknown rate is pretty high, in the order of 3.5%. The goal is to have a very small unknown rate with a small vocabulary size, in order to reduce the number of model parameters and normalize data.

Under the assumption that a misspelled token is rare, correcting them with more frequent ones results in a big reduction of rare words. This way, the count of the top-k elements in the vocabulary will increase, and this has the effect of reducing the number of unknown tokens for a certain maximum vocabulary size k. In Figure 4.4 we show how the unknown percentage changes for different cuts to the vocabulary.

4.2 Data preprocessing

As explained in the previous section, in real world text mining environments it is necessary to perform a precise preprocessing step and this section describes how we decided to design our preprocessing pipeline, in order to make data meaningful and usable.

4.2.1 Cleaning

At first, we perform a *cleaning* step, that consists in several sub-steps:

- Extracting regular expressions representing entities, for example *_URL_*, *_EMAIL_*, *_PERC_*, *_CREDIT_CARD_*, *_MONEY_*, *_NUM_*, *_DATE_*, *_POSITIVE_SMILE_*, *_NEGATIVE_SMILE_*, *_IBAN_*, *_ZIP_*, *_STREET_* and other task specific codes.
- Removing unprintable characters.
- Fixing when an apostrophe is used instead of accent.
- Remove everything inside brackets.
- Surround Italian punctuation with spaces.
- Removing digits inside group of only letters.
- Extracting entities from predefined categories. This is done by exploiting a pre-built set of Italian names, of Italian cities, of Italian days and Italian months into the tokens *_NAME_*, *_PLACE_*, *_DAY_* and *_MONTH_*.
- Fix edge cases.

After this steps, the dataset assumes a better form. In particular, the number of unique words in the vocabulary becomes $265k$, instead of $970k$. This datum made us notice how much important is this step, in fact we reduced the number of unique tokens by 73%.

Another result of this stage is shown in Figure 4.5, which shows how the unknown percentage per vocabulary cut now has a nicer trend, demonstrating again the importance of is this stage.

Cleaning does not affect heavily the lengths distribution. The only aspect that changes significantly is the maximum length, which arrives to 2027. This worsening is intuitive, in fact cleaning data could potentially split words, transforming a token into several ones. This results in a shorter vocabulary but the drawback is that there is an increase in the length of the sequences.

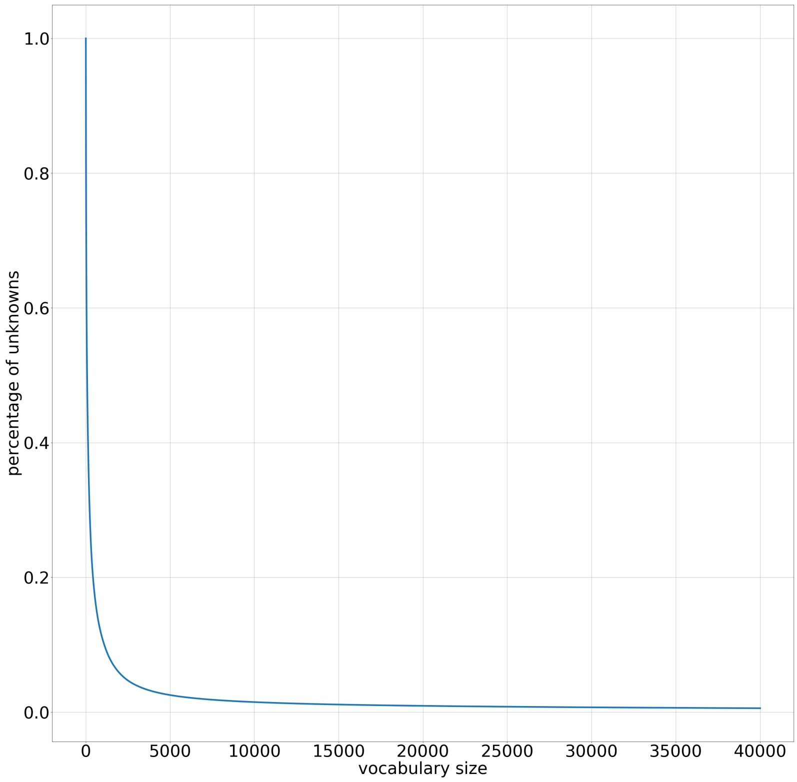


Figure 4.5: The unknown relative percentage per vocabulary size after cleaning. It can be seen how for small vocabulary sizes, such as $10k$ or $15k$, the unknown rate is better than in Figure 4.4.

4.2.2 Fix typos

Table 4.3 shows how the top $30k$ words cover mostly the whole dataset, leaving out $600K$ words, less than the 1%. The plateau we noticed in the previous chapter disappeared, and is interesting to see that, going in depth into the vocabulary, after the $14k$ most common words there are a lot of typos, meaning that roughly 90% of the vocabulary is made of word misspellings.

Examples are *variazioen*, typo of *variazione*, *arivederci*, typo of *arrivederci*, *srevizio*, typo of *servizio*, *principlae*, typo of *principale*, *pormozioni*, typo of *promozioni* and *validoper*, typo of *valido per*. This is not too bad because the corpus percentage related to these tokens is not huge, but a module able to fix those errors could give a substantial improvement.

Top-k	# words	Corpus cov	Voc cov	Sample	count
50	29M	0.43%	0.0002%	hai	205k
100	37M	0.54%	0.0004%	on	121k
500	54M	0.81%	0.0019%	inviato	16k
1k	59M	0.88%	0.0037%	potete	5k
1,5k	62M	0.91%	0.0056%	avrai	3k
2k	63M	0.93%	0.0074%	software	2k
13k	66,6M	0.9868%	0.0484%	chiedergli	49
14k	66,7M	0.9875%	0.0521%	**variazioen**	42
15k	66,7M	0.9881%	0.0558%	ricalcola	37
16k	66,7M	0.9886%	0.0595%	**arivederci**	33
17k	66,8M	0.9891%	0.0633%	**srevizio**	30
18k	66,8M	0.9895%	0.0670%	visualizzandoli	27
19k	66,8M	0.9899%	0.0707%	**principlae**,	24
20k	66,9M	0.99%	0.0744%	**validoper**	22
25k	67M	0.9916%	0.0930%	**pormozioni**,	15
30k	67,5M	0.9926%	0.1117%	avvisandomi	12
∞	67.5M	1,00%	1,00%	–	–

Table 4.3: Vocabulary analysis after cleaning.

The need of fixing word typos comes from the wish of reducing the entropy of the vocabulary. In particular, our goal is also to let the model learn cleaner patterns. In fact, without spell checking, the architecture would have to learn also the implicit mapping between the words and their typos, resulting in an waste of resources.

From the above examples, we can understand that there are two kinds of typos: the ones that are the result of a character level perturbation and the ones that come from the elimination of a space between two words, and each one of them requires a different fixing procedure.

While fixing typos corrections have to be safe, meaning that the spell checking module has to be able to correct errors and not to change the meaning of the text by replacing a correct word with another one that is statistically more frequent in that context. This is crucial, and a way to control the safeness of a correction has to be integrated in the spell checking procedure. Unfortunately, safeness comes at a cost. In fact, designing a policy that avoids false positive means to accept at least some false negative, but here the assumption is that the last kind is not statistically significant.

Even if spell checking can be seen as a character level sequence model task [95], is not that clear how to build an appropriate dataset to train that model on. A way could be to retrieve an huge amount of correct text data from a reliable source, such as *Wikipedia*, and to perform artificial typos on it. The model would be trained to predict the correct sequence from the artificially corrupted one. Even if it is an appealing solution, the problem is that the model would learn to fix artificial typos that the humans never do, resulting in a probably poor tool. For this reason, we choose to design a statistical spell checker.

As a first step, we build a vocabulary holding each unique word and its count. On top of that, we pass each word in the vocabulary through a function that computes all its possible candidate corrections. This is done by performing four character level transformations.

Specifically, the procedure starts by generating all the possible *splits* in the word:

```
splits = [
        (word[:i], word[i:]) \
        for i in range(len(word) + 1)
]
```

Then, all the possible *drops* are generated:

```
deletes = [
        L + R[1:] \
        for L, R in splits if R
]
```

And all the possible *transposes*:

```
transposes = [
        L + R[1] + R[0] + R[2:] \
        for L, R in splits if len(R)>1
]
```

This set is integrated with all the *replaces*, where each replaced character is substitute with one of its *neighbours in the keyboard*:

```
replaces = [
        L + c + R[1:] \
        for L, R in splits \
          if R for c in self.replacement_dict.get(
                R[0],
                random.choice(self.letters)
         )
]
```

And at the end, the *inserts* are generated:

```
inserts = [
        L + c + R \
        for L, R in splits for c in self.letters
]
```

All the above variations of the original word are then put together into the *candidate set*:

```
candidates = set(
        deletes + transposes + replaces + inserts + set(word)
)
```

The idea behind the spell checker is to extract, for each word in the vocabulary, the word in the candidate set *candidates* having the higher probability of not being a typo. For the i-th word in the vocabulary, this probability is defined in Equation 4.2.

$$p(voc\_list[i] \ is \ not \ a \ typo) = \frac{voc[i]}{\sum_{k=0}^{V} voc[k]}. \tag{4.2}$$

In general, this policy is reasonable. Unfortunately, it would also correct some words that are not a typo. This is the case of *caso* that may be paired with *casa*, that is the result of a random insert, if *casa* had higher count than *caso*.

To handle these situations, we design an algorithm that wraps the above mechanism and calls the correction function only in some cases and, even in those cases, it does not automatically accept the correction.

This is done by providing two *thresholds*, $min\_count\_for\_not\_mistake$ and $min\_count\_for\_replacement$. The first one is used to check whether or not a word has to be corrected, in fact tokens with a count that is higher than $min\_count\_for\_not\_mistake$ are not passed through the spell checking function, assuming that very frequent tokens are not a typo. This way the behaviour explained above is partially avoided.

The second threshold, instead, is used to check whether the correction proposed by the algorithm is safe or not. This means that, in case the count of the best candidate is less than $min\_count\_for\_replacement$, the correction is not accepted because its adoption raises the probability of inserting a false positive.

Practically, this spell checking procedure is run five times over the vocabulary generated by the cleaning procedure. Each time, the thresholds are scaled, in order to perform a safe and progressive spell checking. In particular, the thresholds vary in these sets: $[10, 20, 50, 100, 150]$ for the first one and $[5000, 4000, 4000, 2000, 1000]$ for the second. The reason why the first set is made of increasing numbers and the second of decreasing ones is that at the first iteration it is better to be safe, and so to consider as mistakes words with less than 10 occurrences, a very safe and pretty much always true condition. In parallel, it is better to accept as corrections only the words with more than 5000 occurrences, another very safe condition.

After this first iteration a dictionary of safe corrections is generated, where the keys are the corrected words and the values are the selected corrections. A new vocabulary based on this dictionary is built, with the words in the first part of the dictionary, the keys, that are dropped from the vocabulary and the occurrences of the values that are increased by the occurrences of the respective word they are correcting, that is the key. This way, by using a little bit less safe pair of thresholds, the process is performed again and again in order to catch new typos with statistics that are stronger as the process goes on.

Another useful spell checking feature is to detect merged words. To tackle this problem, we adopted the following procedure. All the possible *splits* are generated:

```
possible_splits = [
        (word[:i], word[i:]) \
        for i in range(len(word) + 1)
]
```

Then, only the splits where both the words are present into the vocabulary are retained:

```
existing_splits = [
        split \
        for split in possible_splits
        if len(self.known([split[0], split[1]])) == 2
]
```

The mechanism returns the existing split with the higher probability, if present:

```
candidate = [
        el for el in \
         max(existing_splits, key=lambda x:
                  self.p(x[0]) * self.p(x[1])
          )
         ] \
         if len(existing_splits) >= 1 else [word]
]
```

Again, this is a candidate that is passed to a wrapping script that decides if the correction has to be performed or not basing on the above thresholds.

The vocabulary, after performing the above procedures, arrives to hold $184k$ words, that is a reduction of the 19% with respect to the one coming from the previous step, and for sure can be done something better with a less conservative and safe set of thresholds. As we can see in Table 4.4 and Figure 4.6, from $15k$ words on, the impact on the unknowns is less than the 0.06%.

Top-k	# words	Corpus cov	Voc cov	Sample	count
50	30M	0.44%	0.0003%	servizio	238k
100	39M	0.56%	0.0005%	ci	136k
1k	63M	0.90%	0.0054%	bisogna	5.7k
2k	66M	0.95%	0.0108%	compresi	1.8k
8k	69M	0.990%	0.0433%	impagata	80
10k	69,1M	0.991%	0.0542%	napoletano	46
15k	69,3M	0.994%	0.0813%	pseudo	20
∞	69,3M	1,00%	1,00%	–	–

Table 4.4: Vocabulary analysis after fixing typos.

4.2.3 Abstraction of long sentences

If the procedures described in Sections 4.2.1 and 4.2.2 have the goal of reducing the entropy of the vocabulary, this section details how to bound the sequence length.

We do this by abstracting long and repetitive agent scripts, and to this end we build a dictionary holding long sequences that happen more than one time.

After this phase, another one is performed with the goal of assigning to very similar long sequences, called *agent scripts*, the same token. This necessity comes from the cases when two sequences differ for few tokens, for example they use a different greeting format or they do not share the same subject.

This last proposed procedure essentially merges very similar agent scripts by considering their n-grams overlap. We check all the n-grams, until the four-grams, of any pair of extracted sequences and, if the percentage of common n-grams, until four-grams, is more than 0.33%, qualitatively cross-validated, we consider the two agent scripts as the same one, and the same token is assigned to the two different scripts.

The goal of this last step is to ensure that the assignment of agent scripts tokens to long sequences does not make the vocabulary explode, a phenomenon that would happen when the dataset is full of very similar long sequences. In this case basic procedure would add huge number of useless tokens referring to similar concepts.

Unfortunately, even if this step makes the dataset on average made of shorter sentences, in fact the mean goes from 10.39 to 8.39, it is only a first step towards the solution of the long utterances problem. After the application of this procedure, the maximum length is still 2040, as it is shown in Figure 4.7, and this is due to the fact that long but not frequent sequences are not considered by this method.

Indeed, there are two kinds of long sentences: the ones that are present more than one time and the ones that instead happen a single time. The

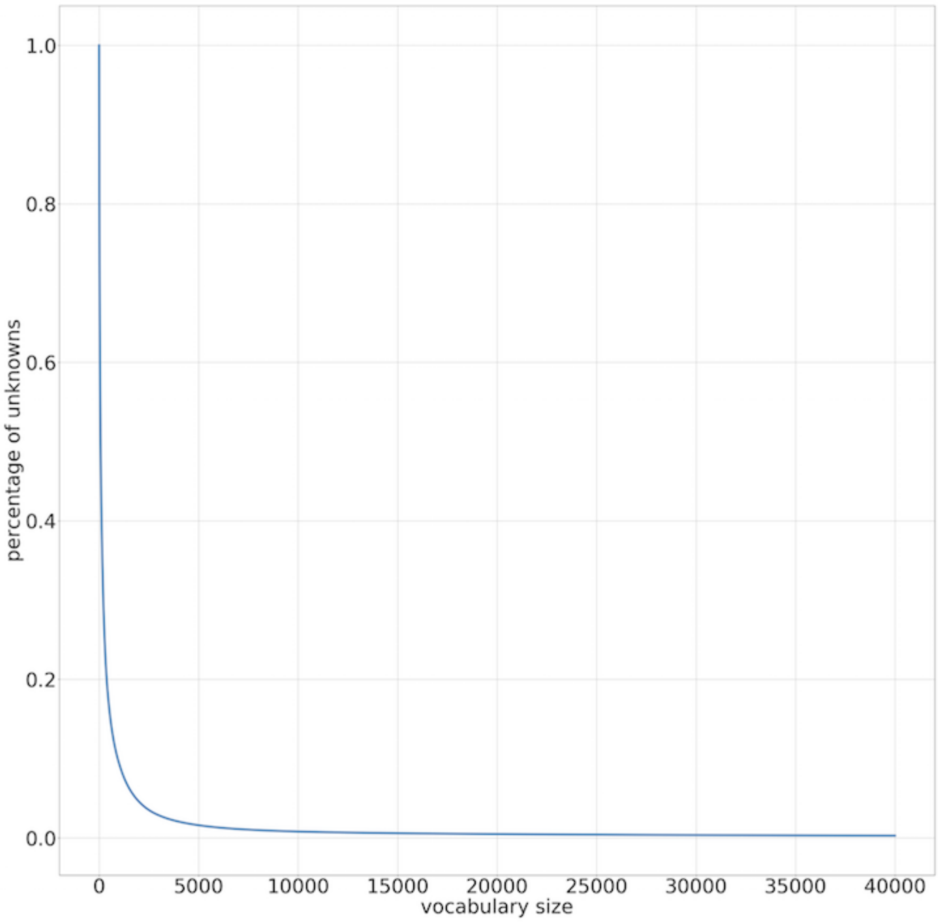


Figure 4.6: The unknown relative percentage per vocabulary size after fixing typos.

method explained in this section handles only the first case, while, regarding the other case, it is difficult to find out a specific way to manage those kind of utterances, and for this reason we decided to simply filtering them out, under the assumption that they can be considered as edge situations that are not useful for the model to get the final goal. Given that this step is performed while the dataset is parsed in multi-turn samples, we explain it in the next section.

4.2.4 Parsing into multi-turn

In the previous sections we considered a dataset of $7,5M$ single utterances, either coming from the agent or from the customer side. To generate a training set that could be fed to the models presented in Chapter 3, we have to parse those conversations into *multi-turn samples.*

How to build a meaningful conversational context is a very open problem.

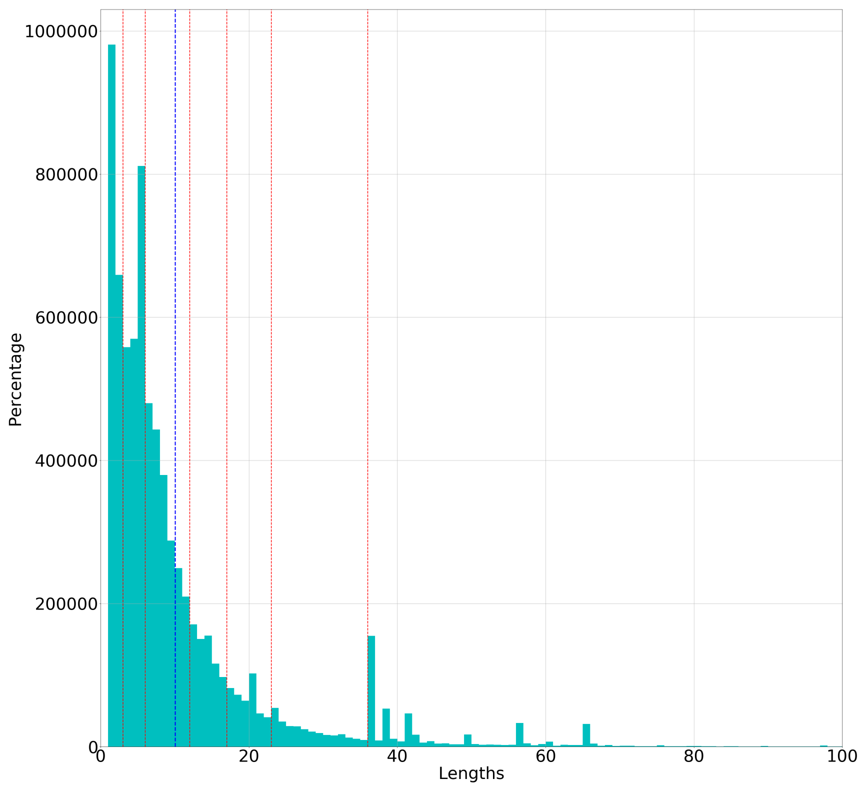


Figure 4.7: The bar-plot shows, for each value of sequence length, the number of single utterances of that length after abstracting long sequences. The mean of the distribution is 8.39*, the variance* 11.68*, the maximum value* 2040*. The* 25*-th percentile is* 2*, the* 50*-th* 5*, the* 75*-th* 10*, the* 85*-th* 14*, the* 90*-th* 18 *and the* 95*-th is* 25*.*

As said in Chapter 2, there are two cases: in the first one the context is made of the immediate question while in the other one the context is made of a set of historically exchanged utterances. The first case is called *single-turn* while the last one is named *multi-turn*.

The longer the context, the more information will be provided to the agent. While in general more information means better quality, in the case of Conversational AI it also means an information explosion and this usually makes the model to be overwhelmed by the information contained in the context. To face this problem, the community designed *attention mechanisms* ([4] and [62]) to provide the model with a way to handle such a big amount of information.

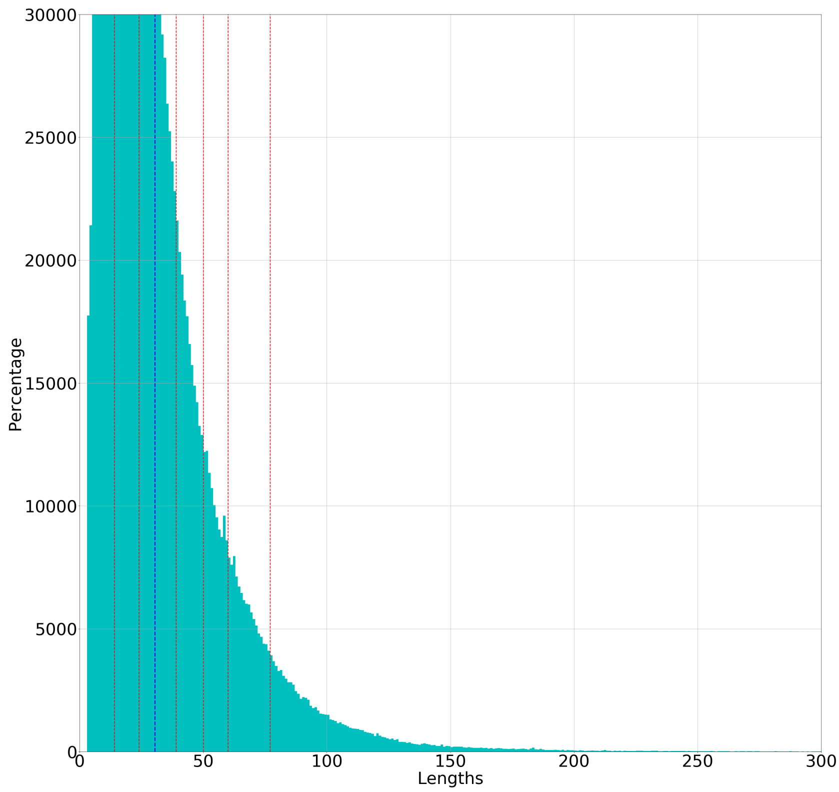


Figure 4.8: Bar-plot of the encoder's lengths for the 1-turn dataset.

In any case, even with attention, choosing the nature of the conversational context to be fed to the model is a delicate design choice. It has implications with the effective size of the model, making actually computationally infeasible to handle contexts that are too long.

In real use cases, a trade-off between information quantity and length of the context is usually accepted. In the following experiments we use two kinds of dataset, one where the context is made of one conversational turn, called one turn dataset, and another one where the model is fed with two turns.

The distribution of the lengths of the contexts of the dataset made of one turn samples is shown in Figure 4.8, with the blue dotted line representing the mean, 30, and red line being the percentiles. We can notice that the 95-th percentile is 77 and the maximum length is 2095. This values show how this distribution presents a very long tail. For this reason, we performed a

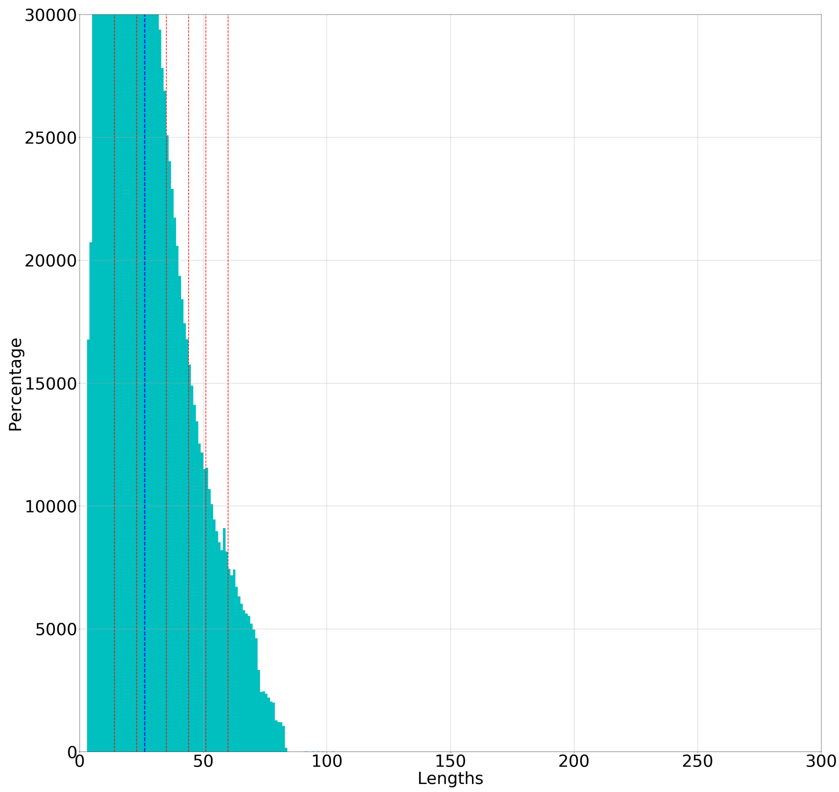


Figure 4.9: Bar-plot of the lengths of the encoder for the 1-turn dataset after filtering samples with a lenght that is above the 99*-th percentile, for each distinct file.*

preliminary filtering on the lengths. This filtering procedure considers the distribution of the lengths at file level.

In fact, for each conversational file in Table 4.1, we computed the percentiles of the lengths of the utterances contained in that file and we retained only the samples for which both the encoder length and the decoder lengths were below the respective 99-th percentiles. Figure 4.9 shows the context lengths after this step. The mean of the lengths is $26,5$ and the maximum value is now 112. The number of samples is reduced from the $1.9M$ of the non filtered dataset to the $1.75M$ of the filtered one, a reduction of 7.7%.

In Figure 4.9 we show the same data distribution of Figure 4.8 with the only difference that it shows the samples with a length that is below the 99-th percentile. The long tail disappears, while the left part of the figure is left unchanged.

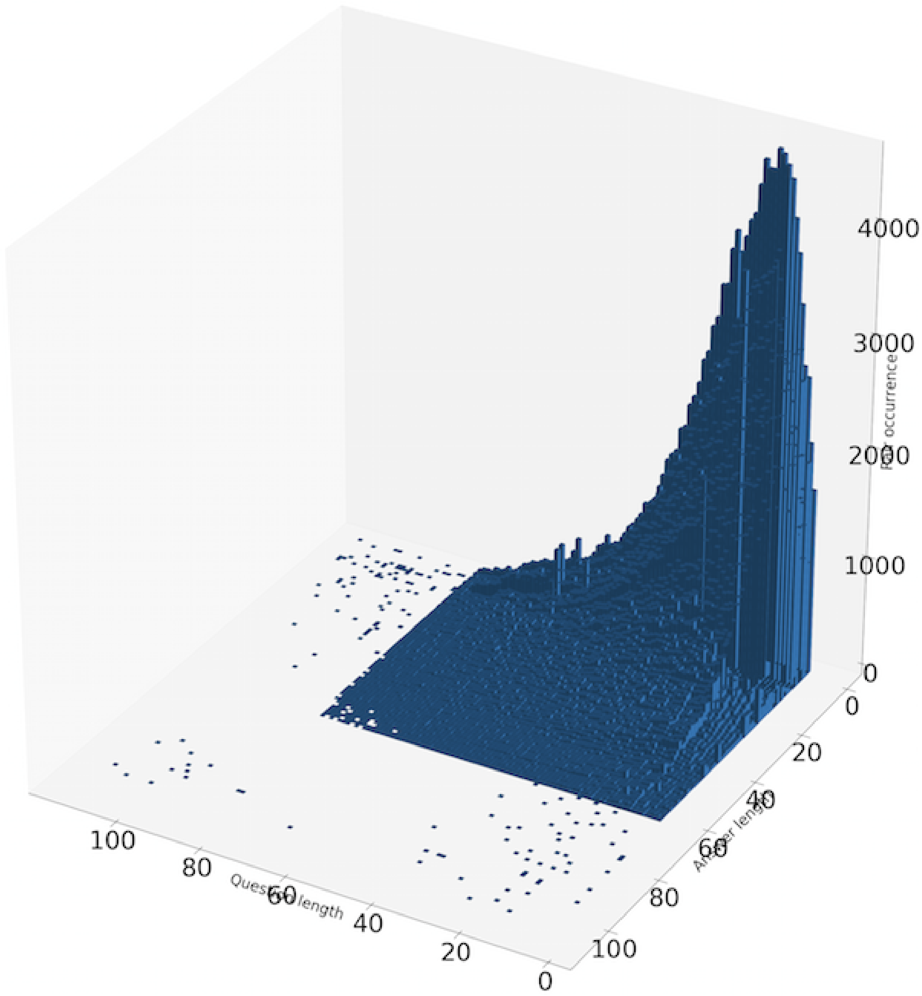


Figure 4.10: 3D bar-plot showing the joint length distribution.

We show a nice visualization of this filtered dataset in Figure 4.10. On the x-axis there is the context length, on the y-axis there is the answer length and for each pair of question-answer lengths the figure shows the number of samples that are represented by that pair.

From Figure 4.10 is clear that there are still some outliers. This happens because filtering the samples basing on the individual percentiles of the files on one hand drops irregular samples, but on the other hand makes the situation sparse, given that the percentiles could be different for each file, and this is the reason why there are the outliers that we can see in Figure 4.10.

Having few samples with an outlier length means forcing the sequence model, while processing these outlier samples, to use a lot of padding tokens when batching them together with regularly distributed samples, causing

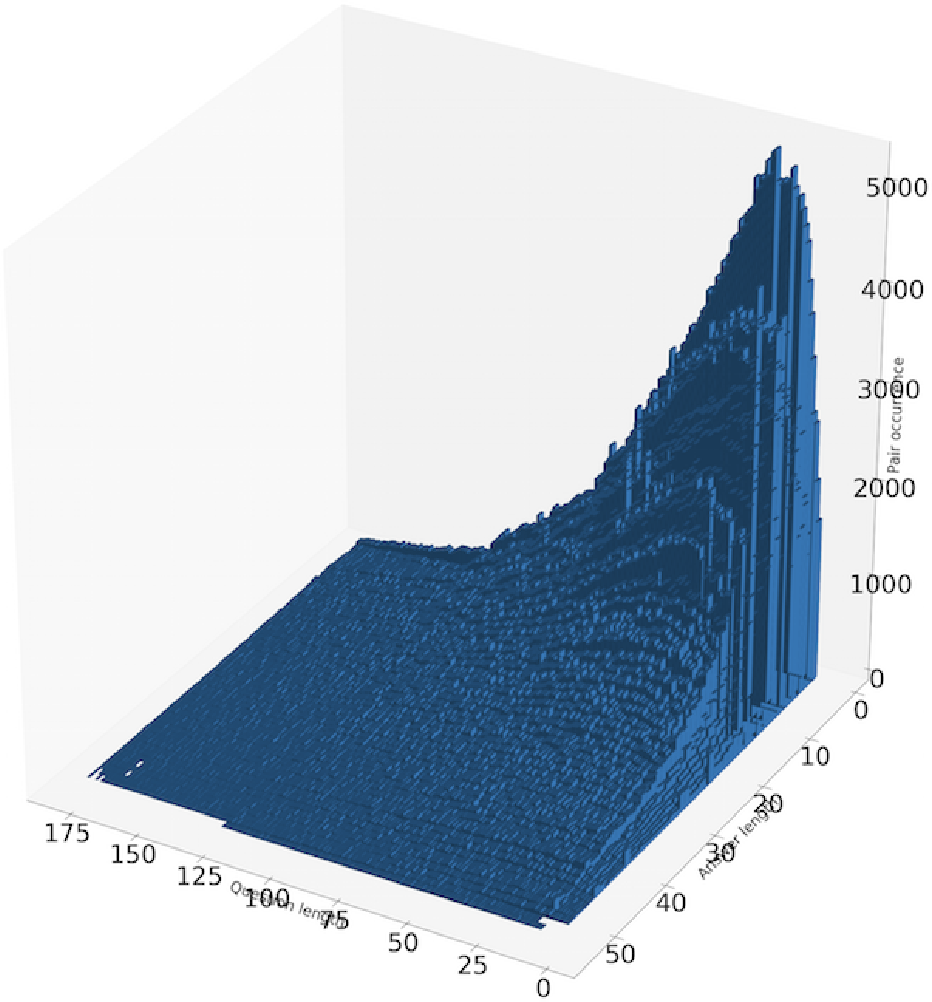


Figure 4.11: 3D bar-plot of the training set for 1-turn parsed dataset.

both computational issues and useless computation. To tackle this problem, after filtering the file level outliers, we compute the global percentiles on the lengths and we retain a sample only if its lengths, both the context and the answer ones, are less than the globally computed 99-th percentile.

After this step, we parse this dataset into train and test sets, that show to be very regular. Figures 4.11 and 4.12 show the joint length distribution of the one turn parsed train and test sets. Nicely, this last filtering procedure drops only $35k$ samples over more than $1.7M$.

In conclusion, the one turn parsed dataset contains 1701130 train samples and 8870 test samples, while the final number of samples for the two turns dataset are 1492479 for train and 7521 for test.

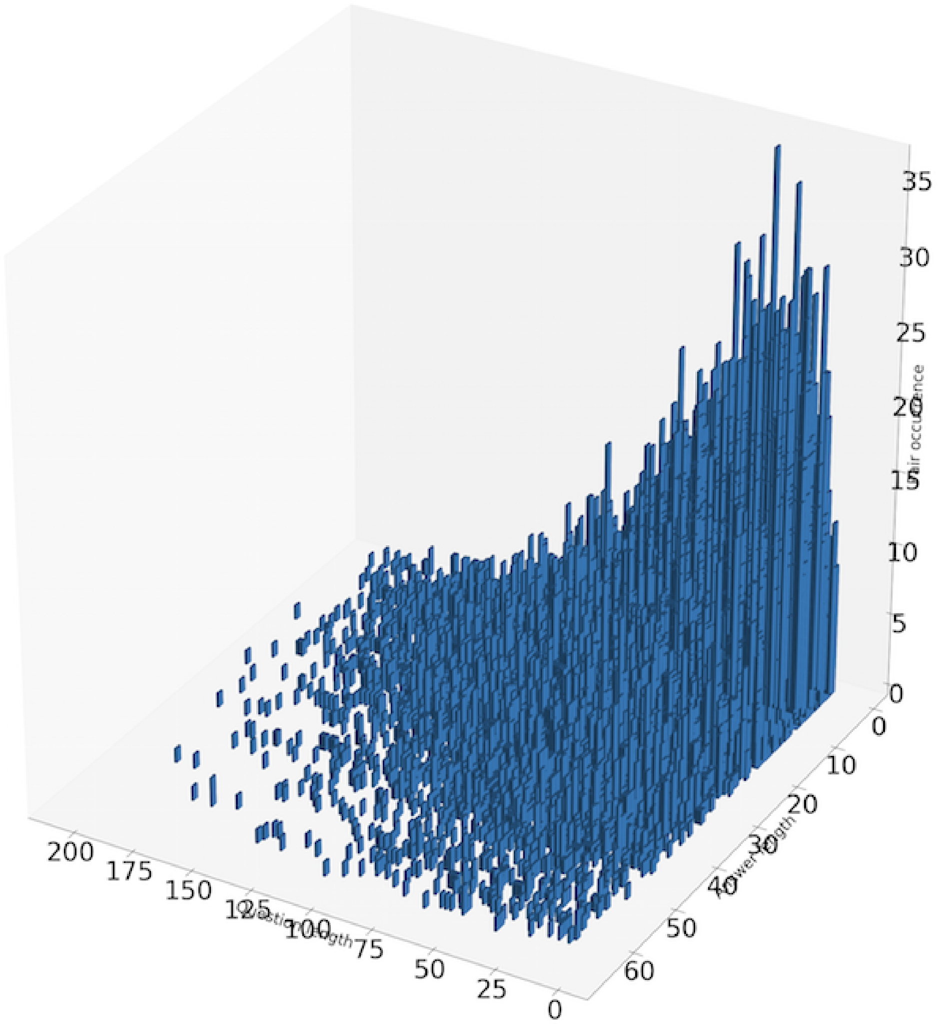


Figure 4.12: 3D bar-plot of the test set for 1-turn parsed dataset.

Chapter 5

Model learning and evaluation

After the data processing steps explained in Chapter 4, in this chapter we implement a simplified version of the model proposed by Xing et al [108], and we study how it performs on the real world dataset described in Chapter 4. This Chapter is both a description of how to practically tackle the development of a Conversational AI software and a critical analysis of quantitative and qualitative metrics.

5.1 Proposed approach

In order to build a generative model able to answer in an *human like* fashion, we use the dataset described in Section 4.1 to train a generative model among the ones of Section 3.2. In particular, we propose a solution similar to the one proposed by Xing et al. (2017) [108], but with the difference that we do not use the backward sentence encoder and that our word level attention is not dependent on the last state of the utterance level encoder, as instead Xing et al (2017) [108] propose.

We decided to adopt these simplifying choices because we had the necessity of building a practical but not trivial solution, that could allow to build a solid baseline for future improvements. We assert that the two modifications that Xing et al. [108] proposed are not crucial to obtain baseline results.

We implemented a model that can be used either in training or in inference mode; we designed an *off-training Django* based interface to serve the model in inference mode with the weights loaded from its last train checkpoint. This resulting interface makes possible to chat with the software in an *on-line* way, and the examples of Section 6 come from an real interaction with it.

5.1.1 Practical setting

We run the experiments presented in this chapter on a machine with twelve CPUs and four K80 GPUs, each one having $12MiB$ of memory and we implemented the models by using the *TensorFlow's APIs*. Having the possibility of accessing this physical setting was very useful because it allowed us to experiment and to practically realize the limits of different approaches, both in terms of memory errors and of saturating resources.

In particular, each cross-validation task consists in a distributed training job run on the machine itself, where six processes are spawned: two workers, one master, a parameter server, an evaluator and an inferencer. Both the master and the workers are assigned to a dedicated GPU machine, while the parameter server and the inferencer share the last GPU and the evaluator works in a multi-process fashion on the CPUs.

5.1.2 Implementation details

We adopted *Python* as the programming language and *TensorFlow*, version 1.4, as the deep learning library. The reason why we chose Python is that it is heavily used by the community, and for this reason is the most contributed and maintained high level wrapper for building deep learning models.

We adopted TensorFlow because of the possibility it gives to modify low level modules, a very important feature if the goal is building non standard deep learning architectures. In fact, if in TensorFlow a very basic sequence to sequence model can be built with not too much lines of code and in *Keras*, another deep learning library, very few are enough, a particular architecture, such as the hierarchical attention mechanism, is much more difficult to be built in other languages with respect to using TensorFlow. This happens because TensorFlow represents a good trade-off between the high complexity of writing deep learning models directly in GPU optimized languages, such as CUDA, and the too much high level nature of languages such as Keras.

5.1.3 Sequence models in TensorFlow

Sequence to sequence models in TensorFlow are made of two blocks, the encoder and the decoder. The first one uses *tf.nn.dynamic_rnn* to return *encoder_outputs* and an *encoder_state* while the second is built with *tf.contrib.seq2seq.dynamic_decode* that takes as its input a custom object called *tf.contrib.seq2seq.BasicDecoder*.

The *BasicDecoder* component during training is built with the helper object defined in *tf.contrib.seq2seq.TrainingHelper*, while in inference it uses *tf.contrib.seq2seq.GreedyEmbeddingHelper*. The decoder returns *outputs* and *final_context_state*, and the *outputs* are then projected on the classes space by using an output projection.

Recurrent models, such as the encoder and the decoder, are defined in terms of recurrence type and Section 3.2.2 explains how the most used architectures work. In addition to the ones reported in Section 3.2.2, recently research proposed some new algorithms. One nice aspect of using TensorFlow is that these architectures are implemented and made available in the codebase after few months they are presented. An example is that there is the possibility of using the regularized version of the LSTM proposed by Zaremba et al. (2014) [112], implemented in $tf.contrib.rnn.BasicLSTMCell$. This is invaluable during cross validation, and its power is shown in Section 6, where we test several different recurrent architectures.

In Tensorflow, sequence models can be run either in training and in inference mode, respecting the architecture depicted in Loung thesis [96] and described in Figure 3.28 and 3.29. Also, we decided to use TensorFlow iterators, $tf.data.Iterator$, an efficient way to load data from disk that minimizes the interaction between Python and Tensorflow optimized primitives, written in *C++*. The iterator is shuffled with a seed equal to the number of turns in which the data is parsed, in order to ensure reproducibility.

5.1.4 Extending a TensorFlow model

In TensorFlow, the attention mechanism is implemented by wrapping the decoder cell with an attention wrapper, defined in the dedicated class called $tf.contrib.seq2seq.AttentionWrapper$.

This code takes the recurrent cell and an attention mechanism that can be either $tf.contrib.seq2seq.LuongAttention$, implementing *Loung* attention score function [62], or $tf.contrib.seq2seq.BahdanauAttention$, implementing instead the *Bahdanau* attention scoring [4]. Nicely, the complex implementation staying behind the high level APIs calls briefly mentioned above is completely hidden to the eyes of the developer, that only has to call the wrapper with the appropriate parameters.

Given that TensorFlow does not implement such a particular algorithm as the hierarchical attention wrapper, in order to integrate it into an existing sequence model we had to dive into the code base, understand the interfaces, and implement the required wrapper writing a compliant code that could be pluggable in a working sequence to sequence architecture. The Tensorflow development and the empirical evaluation of this wrapper, called *Hierarchical Attention Wrapper*, is one of the biggest contributions of this work.

5.1.5 Hierarchical attention

Sequence models empowered with hierarchical attention have a lot of *hyper-parameters*. For example, the attention mechanism can be *Loung* [62] or *Badhanau* [4] based. Then there are batch size, dropout, encoder type, initialization weights, initialization style, learning rate, maximum gradient norm, number of layers in the encoder and in the decoder, number of hidden units, number of epochs, and the optimizer to be used.

Important are also the choices of using or not lazy optimizers, the recurrent cell to be adopted, to use or not residual connections, to reverse or not the source sentence, the decay schema, to either perform sequential or distributed training, the maximum length of the vocabulary, the length penalty used with the *BLEU* score, to use or not pre-trained word embeddings, to share or not the word embeddings, the sentence attention mechanism, the number of units in the sentence encoder and the number of elements in the sentence encoder. Also, it is crucial to define if it is necessary to initialize or not the first state of the decoder with the last one of the encoder and into how many turns the input has to be parsed into.

Even if each hyper-parameter is important and determinant for the final performance, for some of them there are some typical values coming from the machine translation experiments. The value assigned to other parameters instead, such as *batch size* and *hidden size*, is bounded by the computational capacity of the machine and a minimum complexity that is necessary to provide the model with. For this reason we performed cross validation only for some hyper-parameters, assuming that the values coming from machine translation are portable also to Conversational AI and leaving their validation to future work.

Specifically, in Table 5.1 we show some standard values for the typical hyper-parameters of a hierarchical generative chatbot similar to the one presented by Xing et al (2017) [108]. With such an architecture, the model ends up having a number of variables between $18M$ and $27M$.

5.1.6 Lazy adam optimizer

Additionally, the update of the variables is eased by the use of lazy Adam optimizer, implemented in TensorFlow in $tf.contrib.opt.LazyAdamOptimizer$, a variation of the algorithm presented by Kingma et al (2014) [49] that performs sparse parameter updates.

In fact, the original Adam algorithm maintains two moving-average accumulators for each trainable variable and the accumulators are updated at every step. Lazy Adam instead provides lazier handling of gradient updates for sparse variables. It only updates moving-average accumulators for sparse variable indices that appear in the current batch, rather than updating the accumulators for all indices.

Compared with the original Adam optimizer, it can provide large improvements in model training throughput for some applications, and future work requires to experiment if this choice gives an improvement with respect to the traditional algorithm of Kingma et al (2014) [49]. In fact, it provides slightly different semantics than the original Adam algorithm, and may lead

Hyper parameter name	Standard value
word attention mechanism	Badhanau et al. [4]
sentence attention mechanism	Badhanau et al. [4]
dropout	0.2
encoder type	bidirectional ([82] and [4])
initialization weight	0.1
initialization style	variance scaling (He et al. [35])
learning rate	0.001
optimizer	adam (Kingma et al.[49])
maximum gradient norm	10
number of encoder layers	4 bidirectional
number of decoder layers	4 unidirectional
number of units encoder and decoder	256
number of epochs per curriculum step	1
pass encoder hidden state to decoder	True
training	distributed and asynchronous
vocabulary max length	10k
BLEU length penalty	1.0
share vocabulary	False
pre-train embeddings	True
sentence encode layers	4
batch size 1-turn	128 (only data parallelism)
batch size 2-turn	42 (only data parallelism)
batch size 3-turn	25 (only data parallelism)
use lazy adam	True
residual connections	True
reverse source sentence	False
number of units sentence encoder	256

Table 5.1: Standard hyper parameters for generative hierarchical attention based chatbot.

to different empirical results.

5.2 Learning paradigm

In training a Conversational AI tool, at least one that aims to be able to show basic natural language understanding skills, the number of design choices are many. First of all, it is important to define how much context we want to consider, that is in how many turns we organize the training samples.

Given that this choice determines how much context the model is fed with, it is crucial. In this scope, we decided to validate two approaches and to make a comparison between them. The first one is named *curriculum* *learning* [6], and takes inspiration from the reinforcement learning community, while the second one is more traditional and adopts an intuition presented by Vaswani et al (2017) [100].

5.2.1 Curriculum learning

This approach takes inspiration from a peculiar aspect of conversations. In fact, while the hierarchical structure of dialogues is a property often used to bias the model architecture, and an example is the algorithm presented by Xing et al (2017) [108], another property that is usually not exploited is that conversations can be seen as incremental in their complexity.

In this scope, generative answering can be seen as a task where the complexity grows as the provided context gets longer, in fact answering to one turn contextualized question is easier then doing it for two turns, and curriculum learning is adopted because it is particularly suited to exploit this property. In fact, it is usually chosen for problems where the agent starts from simple tasks and abstracts the knowledge to solve more complex tasks.

Specifically, a Conversational AI model trained with curriculum learning performs some epochs with one turn samples, then it changes dataset and uses two turns data for some other epochs, and so on, increasing the complexity of the training samples. In this setting, we perform cross validation procedure by taking a pool of configurations and we train them with datasets that have an incremental complexity.

This approach is promising, but clearly has drawbacks if implemented in this basic way. In fact, the learning rate has to be set carefully in order to not support the overall convergence.

In the phase where the model uses one turn parsed training set, the learning rate is set to lr_0 and remains constant for half of the training steps. Then, it is decayed during the second half of the training steps with staircase style. When the training with one turn data ends, a second phase starts and the dataset is changed with one parsed in two turns. The learning rate is now set to $lr_0 * 0.7$ and the decay is the same used in the one turn phase. This pattern goes on for all the steps of curriculum learning that the developer wants to implement, and the initial learning rate for each step is computed as $lr_0 * 0.7^{num\_turns}$.

One of the contributions of this thesis is the idea of designing such a learning paradigm that takes into consideration the incremental complexity of conversations to define a procedure that could help the model to generalize better. Also, we believe that through this work we opened a new branch of research that, if further deepened, could potentially give important improvements in the design of generative models.

Figure 5.1 shows the resulting learning rate for a hierarchical attention model. It can be seen that roughly at $40k$ steps there is the end of the one

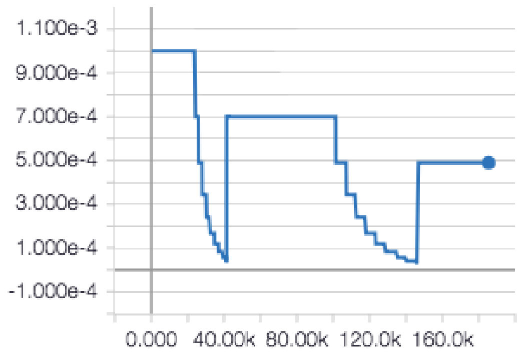


Figure 5.1: Proposed learning rate decay for curriculum learning training.

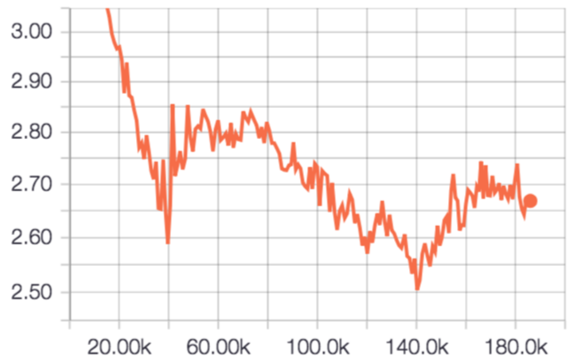


Figure 5.2: Evaluation loss for a curriculum learning training computed on 1*-turn dataset.*

turn training and the beginning of the second phase, where the model is fed with two turns parsed samples. As can be seen in Figure 5.1, this second phase ends at $140k$ steps, where it is started a third phase using three turns parsed samples.

The assumption on which we rely in designing the learning rate decay of Figure 5.1 is that, during the one turn training, the optimization should arrive to a safe point and then, during the second phase, the optimization is going to head to a minimum for the two turns function, and it does this in a better way than in the case in which it started to train in two turns. A side property would be that the final set of weights will be optimal for both the one turn and two turns cases. It has to be noticed that, this way, the function to be optimized slightly changes when a new curriculum learning phase starts, and for this reason the loss value could fluctuate a bit when the dataset changes.

Figures 5.2, 5.3 and 5.4 show an example where the training of a hierarchical attention sequence model has been stopped after $20k$ iterations of the three turns phase. In particular, in the three figures below, training

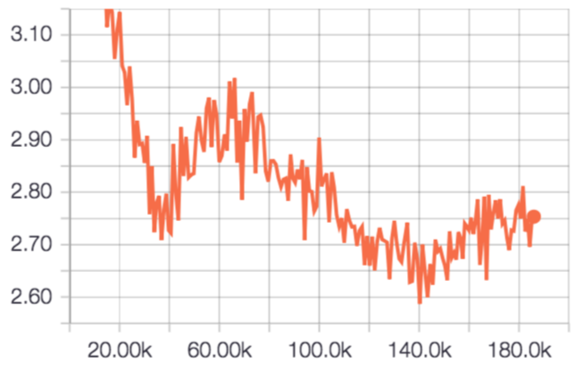


Figure 5.3: Evaluation loss for a curriculum learning training computed on 2*-turn dataset.*

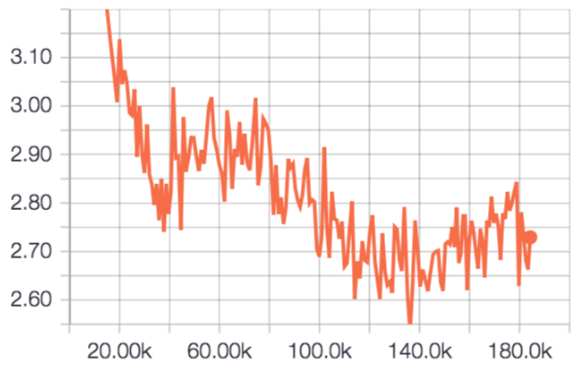


Figure 5.4: Evaluation loss for a curriculum learning training computed on 3*-turn dataset.*

is performed in an asynchronous way, as explained in Section 5.3.1, with three workers each one using lazy adam optimizer [49]. The master uses a constant learning rate of 0.0005 and a constant dataset parsed in three turns, while the two workers use the decay proposed in Figure 5.1, with a starting learning rate of 0.001, together with curriculum learning to support incremental training. Figures 5.2, 5.3 and 5.4 represent the per word perplexities [101] computed on three different evaluation sets, respectively containing one turn, two turns and three turns parsed samples.

We can see that, after step $40k$, there is a fluctuation where the evaluation perplexity increases, and this happens for each evaluation set. This happens because at $40k$ steps starts the second step of curriculum learning, and a two turns parsed training set is used. The same phenomenon happens at $140k$ steps, where the model starts to be fed with three turns parsed samples. Interestingly, after this fluctuation, the perplexity arrives to a value that is even better than the one obtained after the previous curriculum learning step. We can also notice that, during the two turns training, the phase between $40k$ and $140k$ steps, the learning rate decay helps the model to stabilize and to converge to the optimal value of 2.50, allowing it to not have high variance and to decrease smoothly.

It is important to point out that, while the master is performing several epochs with the three turns dataset, the workers perform only one epoch for each curriculum learning step. In this last point we can see that there is a great space for research. First of all the workers can be trained for more than one epoch during each curriculum learning step, secondly the batch size is an important limitation of this configuration and increasing it could result in an even more smooth convergence. Another variation could be to use more than two workers, some of them using the constant learning rate and others using the proposed decay. Another important choice is the dataset to be provided to the master: in the above examples it is made of three turns samples but it can be one turn, two turns, three turns or maybe also more than three turns. While the hidden size is pretty much in line with the normal values used for machine translation, that is 256, and for this reason the complexity could be considered enough, the batch size is way less than the suggested one, that is 256, and this phenomenon gets worst and adds variance to the loss as the number of turns increases.

For example, for three turns the adopted batch size is 25, that is small and does not allow to perform safe updates. Even if this phenomenon is partially overcome by the learning rate decay, keeping such a low batch size and maybe going on with new curriculum learning steps with more than three turns would not result in good performance. For this reason, in the experiments of this chapter, the models trained with curriculum learning use at most two turns parsed training data.

5.2.2 Traditional learning

Another contribution that we give is a comparison, presented in Section 6.2.1, between the novel *Curriculum learning* approach defined in Section 5.2.1, and a second more traditional way of designing the learning paradigm for generative chatbots, that we call *Traditional learning*.

This last procedure consists in training a generative conversational model only with data parsed in a specific number of turns, kept constant during training. Here stays the first difference with *Curriculum learning*. In fact, while the core feature of the curriculum based approach explained in Section 5.2.1 is that the model is trained with a set of datasets with an increasing complexity, this second more traditional approach proposes to train the model always with a set of samples parsed in a specific number of turns.

We call this second approach *Traditional* because it is a combination of the paradigm with which the community usually trains generative conversational models, extended with some intuitions provided by Vaswani et al 2017, [100]. In fact, we decided to adopt the learning rate decay that

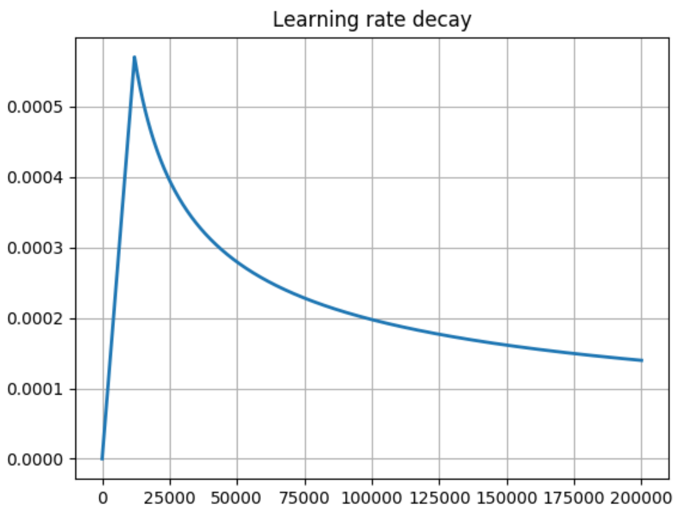


Figure 5.5: Noam decay from Vaswani et al (2017) [100] .

Vaswani et al 2017, [100] used to train the *Transformer*.

A contribution of this work is the idea of combining the traditional approach used to train generative conversational models with the learning rate decay proposed by Vaswani et al 2017, [100].

The nice property of this innovative learning rate decay, shown in Figure 5.5, is that the learning rate is not fixed during the initial steps, as curriculum learning instead suggests. This property is important, in fact the initial value of the learning rate in each stage of curriculum learning phase is a crucial choice to be validated.

This learning rate decay is defined by equation 5.1 and shown in Figure 5.5:

$$lr(step) = d_{model}^{-0.5} * \min(step^{-0.5}, step * warmup^{-1.5}). \tag{5.1}$$

As we can see in the decay function plotted in Figure 5.5, the learning rate starts from zero and increase linearly for *warmup* steps. Then it decreases exponentially. The choice of the *warmup* steps is important, in fact it determines the maximum value for learning rate, how smoothly the value reaches zero and the average value of the exponential part. For this reason, it useful to cross validated it.

This decay is interesting because it allows the model to learn smoothly in the first phase. The nice property here is that in the first part of learning the function returns smaller and progressively increasing values, allowing to perform safe updates.

5.3 Training procedure

In order to implement train algorithms like the one presented in Section 5.2, there are several training procedures that can be adopted. First of all, usually a lot of parameters are necessary to provide the model enough complexity to learn the final task. Due to the inner entropy of Conversational AI, this number is usually even bigger than in other applications. The huge number of parameters that the model has to learn makes the training procedure mainly made of very big matrix multiplications and performing those operations on the CPUs results in a very slow training. For this reason, the common approach is to equip a machine with several GPUs to exploit their ability of performing quick matrix multiplications.

Training can be performed sequentially or in a distributed fashion, and in some cases also training without parallelism is not a big deal. In other situations, in which the training speed is crucial and there are strict time constraints, it is necessary to adopt more complex techniques to speed up the whole procedure. The answer to this need is usually *Distributed TensorFlow*, a package implemented with the goal of providing the developers with APIs to control and implement the distributed training of a deep learning model.

A model is usually defined from the data fetching to the definition of the loss function and when its training graph is executed, it is compiled and issued to the run time optimizer, that can run in either sequential or distributed mode. Sequential training is the solution that is usually chosen. In this setting, the static graph is taken and provided to a process that has the role of entering into a training loop and of running the forward and backward pass for each training batch. The evaluation and inference procedures are performed periodically in the main thread after a predefined number of steps. During these evaluation tasks the training is suspended. The process ends when the epochs are finished or if a termination condition triggers. After this, the final weights are returned.

Unsupervised feature learning and deep learning have shown that being able to train large models on vast amount of data can drastically improve model performance. However, training a deep network with millions, or even billions of parameters can result in a problem. How do the developer achieve this without waiting for days, or even multiple weeks ? Dean et al. [24] propose a different training paradigm which allows to train and serve a model on multiple physical machines. The authors propose two novel methodologies to accomplish this, namely, *model parallelism* and *data parallelism*. Given that, on average, generative Conversational AI models have an order of $20M$ parameters, to reduce the training time the experiments of this chapter are run in distributed settings, in particular exploiting data parallelism.

In model parallelism, a single graph is distributed across multiple machines. The performance benefits of distributing a deep network across mul-

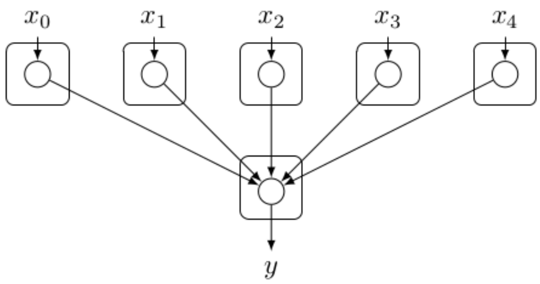


Figure 5.6: Synchronizing perceptron computation. Figure coming from [37]

tiple machines mainly depends on the structure of the model. Models with a large number of parameters typically benefit from accessing more CPU cores and memory, thus, parallelizing a large model usually produces a significant performance increase, and thereby reduces the training time.

There is a famous example to illustrate this concept more clearly. Imagine having a perceptron, as depicted in Figure 5.6. In order to parallelize this efficiently, a neural network has to be considered as a dependency graph where the goal is to minimize the number of synchronization mechanisms. A synchronization mechanism is only required when a node has more than one variable dependency, that is a dependency which can change in time. For example, a bias would be a static dependency, because the value of a bias remains constant over time. In the case for the perceptron shown in Figure 5.6, the only synchronization mechanism which should be implemented resides in output neuron, in fact $y = \sigma(\sum_i w_i x_i)$. In Figure 5.6, a perceptron is partitioned by using the model parallelism paradigm. In this approach every input node is responsible for accepting the input x_i from some source, and multiplying the input with the associated weight w_i. After the multiplication, the result is sent to the node which is responsible for computing y. This node requires a synchronization mechanism to ensure that the result is consistent. The synchronization mechanism does this by waiting for the results y depends on.

In *TensorFlow*, model parallelism is achieved through the so called *Device Wrappers*, that allow the developer to physically send a part of the graph to specific machine, or to specify that some operations have to be performed on specific physical machines. Once these wrappers are placed, the framework takes care of how to synchronize the whole computation. For example, in the experiments that we present in this chapter, device wrappers are used to specify that the word embedding lookup and in general the embedding operations are performed on the CPU.

Data parallelism is an inherently different methodology of optimizing parameters. The general idea is to reduce the training time by having n workers optimizing a central model by processing n different shards of the

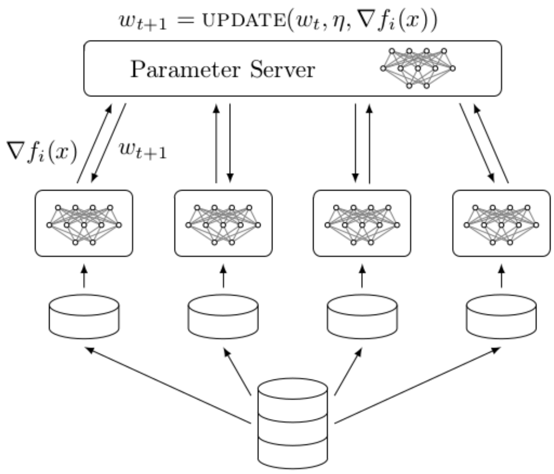


Figure 5.7: In this methodology n workers are spawned, not necessarily on different machines, and a data shard of the dataset is assigned to every worker. Using this data shard, a worker i will iterate through all mini-batches to produce a gradient, $\nabla f_i(x)$ for every mini-batch x. Next, $\nabla f_i(x)$ is sent to the parameter server, which will incorperate the gradient using an update mechanism. Figure coming from [37].

dataset in parallel. In this setting n model replicas are distributed over n processing nodes, that is every node, or process, holds one model replica. Then, the workers train their local replica using the assigned data shard. However, it is possible to coordinate the workers in such a way that, together, they will optimize a single objective. There are several approaches to achieve this. Nevertheless, a popular approach to optimize this objective is to employ a centralized parameter server, that is responsible for the aggregation of model updates and the parameter requests coming from different workers. The distributed learning process starts by partitioning a dataset into n shards, each one assigned to a particular worker. Next, a worker will sample mini-batches from its shard in order to train the local model replica. After every mini-batch, or multiple mini-batches, the workers communicate a variable to the parameter server. This variable is, in most implementations, the gradient $\nabla f_i(x)$. Finally, the parameter server will integrate this variable by applying a specific update procedure. This process repeats itself until all workers have sampled all mini-batches from their shard. This high-level description is summarized in Figure 5.7.

5.3.1 Approaches for distributed training

Parallelizing gradient updates is not an intuitive task since gradient descent is an inherently sequential algorithm where every data point provides a direction to a minimum. However, training a model with a lot of parameters while using a very large dataset, will result in a high training time. To reduce

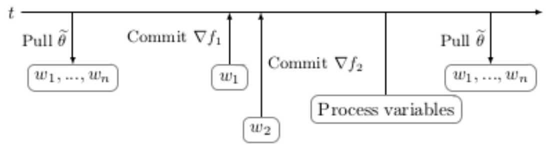


Figure 5.8: In a synchronous data parallel setting, there are n workers, not necessarily on different machines. At the start of the training procedure, every worker fetches the most recent center variable. Next, every worker will start their training procedure. After the computation of the gradient, a worker commits the computed information, gradient or parametrization, depending on the algorithm, to the parameter server. However, some workers might induce a significant delay, which results in other workers to be taskless while still consuming the same memory resources. [37]

the training time, the obvious choice would be to buy better, or rather, more suitable hardware, for example GPUs, but this is not always possible. For this reason, several attempts have been made to parallelize gradient descent.

There are two distinct approaches towards solving data parallelism and the most intuitive is synchronous data parallelism. In synchronous data parallelism, as depicted in Figure 5.8, all workers compute their gradients based on the same center variable. This means that, whenever a worker is done computing a gradient for the current batch, it will commit a parameter to the parameter server. However, before incorporating this information into the center variable, the parameter server stores all the information until all workers have committed their work. After this, the parameter server will apply a specific update mechanism to incorporate the commits. In essence, one can see synchronous data parallelism as a way to parallelize the computation of a mini-batch. Intuitively, workers might commit their results with a certain delay. Depending on the system load, this delay can be quite significant. As a result, a synchronous data parallel method is only as strong, as the weakest worker in the cluster.

In order to overcome the significant delays induced by loaded workers in synchronous data parallelism, and thereby decrease the training time even further, the idea of asynchronous parallel training is to remove the synchronization constraint. However, this imposes several other effects. The first, is *parameter staleness*. Parameter staleness is the number of commits other workers performed between the last pull, and the last commit, the parameter update, of the current worker. Intuitively, this implies that a worker is updating a *newer* model using gradients based on a previous parametrization of that model. This is shown in Figure 5.9. The other, less intuitive side-effect is *asynchrony induced momentum* [67]. This means that adding more workers to the problem also adds more implicit momentum to the optimization process. This implicit momentum is the result of the queuing model required by asynchrony. Note that some approaches,

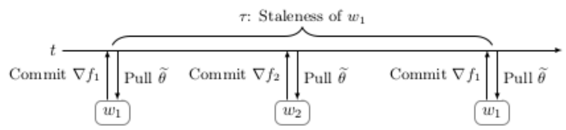


Figure 5.9: In asynchronous data parallelism, training time is on average reduced due to the removal of the synchronization mechanism in synchronous data parallelism. However, this induces several effects such as parameter staleness, and asynchrony induced momentum. [37]

such as *Hogwild!* [75], do not require locking mechanisms, since they assume sparse gradient updates. Also, Mitliagkas et al [67], said that adding more asynchronous workers to the problem actually deteriorates the statistical performance when using algorithms which do not take staleness and asynchrony into account.

5.4 Evaluation metrics

How to evaluate a response generation model is still an very open problem. A deep study about this has been performed by Liu et al in 2017 in their *"How to not evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation."* [59]. Usually, generative response generation models adopt metrics from machine translation to compare the answer of the model with the target response. In their work, Liu et al [59] showed how these metrics correlate very weakly with human judgement in the non-technical *Twitter domain* and not at all in the technical *Ubuntu domain*.

Practically, evaluation in dialogue models is done by using human generated supervised signals such as task completion or user satisfaction score. These kind of models are called *supervised dialogue models* while the others are called *unsupervised dialogue models*. In their work, Liu et al [59] only consider the second kind of models. In the last years the unsupervised ones received an increasing attention, and this is due to the fact that being trainable end-to-end (Serban et al 2016 [86], Sordoni et al 2015 [92], Vinyals et al 2015 [101]) there is not the need to collect supervised labels, which can be prohibitively expensive. However, evaluating these models remains an open question.

Faced with similar challenges, other natural language tasks have successfully developed automatic evaluation metrics. For example *BLEU* (Papineni et al 2002a [71]) and *METEOR* (Banerjee et al 2005 [5]) are now standard for machine translation while *ROUGE* (Lin 2004) [56] is used for text summarization. The automatic evaluation metrics can be divided into two categories: the *word-overlap based*, such as *BLEU*, *ROUGE* and *METEOR*, and

Context of Conversation
Speaker A: Hey John, what do you want to do tonight?
Speaker B: Why don't we go see a movie?
Ground-Truth Response
Nah, I hate that stuff, let's do something active.
Model Response
Oh sure! Heard the film about Turing is out!

Figure 5.10: Example showing the intrinsic diversity of valid responses in a dialogue. The reasonable model response would receive a BLEU score of 0. Figure coming Liu et al 2017, "How to not evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation." [59].

the *word embedding based*, that are based on word embedding models such as *Word2Vec* (Mikolov et al 2013 [64]). In their paper Liu et al [59] show that all metrics show either weak or no correlation with human judgements, despite the fact that word overlap metrics have been used extensively in the literature for evaluating dialogue systems (above and Lasguido et al 2014 [70]). An example is shown in Figure 5.10.

In this scope, Ritter et al 2011 [77] formulate the unsupervised learning problem as one of translating a context into a candidate response and they used Statistical Machine Translation (SMT) to generate the responses to various contexts using *Twitter data* and showed that their model outperformed information retrieval baselines according to both *BLEU* and human evaluations. However, these metrics assume that valid responses have significant word overlap with the ground truth responses, and this is a very strong assumption for dialogue systems, where there is a significant diversity in the space of valid responses to a given context. Sordoni et al 2015 [92] extended this idea using recurrent language models to generate responses in a context-sensitive manner. They evaluated their models by using *BLEU* and they produced multiple ground truth responses by retrieving a set of responses from elsewhere in the corpus, using a bag of word model. Li et al 2015 [53] evaluate their diversity-promoting objective function with *BLEU* using a single ground truth response. Galley et al 2015b [30], proposed a modified version called *DeltaBLEU*, which takes into account several human evaluated ground truth responses and that is shown to be weakly to moderate correlated to human judgement using Twitter dialogues.

Unfortunately, such human annotation is infeasible to be obtained in practice. Several works evaluate how well automatic metrics correlate with human judgements in both machine translation (Callison-Burch et al 2010 [14], Callison-Burch et al 2011 [15], Graham et al 2015 [33], Bojar et al 2014 [9]) and natural language generation (Stent et al 2005 [94], Cahill et al 2009 [13], Reiter et al 2009 [76], Espinosa et al 2010 [26]). There have been also works criticizing the use of *BLEU* for machine translation (Callison-Burch et al 2006 [16]).

While many of these criticisms apply to dialogue generation, in their paper of 2017 Liu et al [59] noticed that generating dialogue responses conditioned on the conversational context is in fact a more difficult problem, and this is due to the fact that the set of possible correct answers to a context is very large. The main point here is that dialogue response generation given solely the context has intuitively a higher entropy than translation given text in source language.

5.4.1 Word overlap based metrics

Considering metrics that evaluate the word overlap between the proposed response and the ground truth response, the most interesting are *BLEU*, *ROUGE* and *METEOR*. In this section we will denote the ground truth response as r while the proposed response is identified as $\hat{r}$. The j-th token in the ground truth response is w_j while $\hat{w_j}$ is the j-th of the proposed response.

BLEU (Papineni et al 2002a [71]) analyzes the co-occurrences of n-grams in the ground truth and the proposed responses. It first computes an n-gram precision for the whole dataset that, with a single candidate ground truth response, is computed as:

$$P_n(r, \hat{r}) = \frac{\sum_k \min(h(k,r), h(k,\hat{r}))}{\sum_k h(k,r)}, \tag{5.2}$$

where k indexes all possible n-grams of length n and $h(k, r)$ is the number of n-grams equal to k in r. To avoid the drawbacks of using a precision score that favours shorter candidate sentences, the authors introduce a brevity penalty. The most common value of n is usually 4. *BLEU-n* is then defined this way:

$$BLEU\_n = b(r, \hat{r}) \exp(\sum_{n=1}^{N} \beta_n \log(P_n(r, \hat{r}))), \tag{5.3}$$

where β_n is an uniform weighting and $b(.)$ is a brevity penalty.

METEOR metric (Banerjee et al 2005, [5]) creates an explicit alignment between the candidate and target responses. The alignment is based on exact token matching, followed by *WordNet synonyms*, stemmed tokens and then paraphrases. Given a set of alignments, the *METEOR* score is the harmonic mean of precision and recall between the proposed and ground truth sentence.

ROUGE (Lin 2004, [56]) is a set of evaluation metrics used for automatic summarization. The most interesting version is the *ROUGE-L*, which is a *F-measure* based on the *Longest Common Subsequence (LCS)* between a candidate and target sentence. The LCS is a set of words which occur in two sentences in the same order. However, unlike n-grams the words do not have to be contiguous, that is there can be other words in between the words of the LCS.

Following Vinyals and Le 2015 [101], perplexity, Equation 5.4, is usually exploited as the typical *model dependent* evaluation metric for Conversational AI and it measures how well a model predicts human responses. Lower perplexity generally indicates better generation performance.

$$PPL = exp\{-\frac{1}{N}\sum_{i=1}^{N}\log(p(Y_i|U_i))\}. \tag{5.4}$$

5.4.2 Embedding based metrics

An alternative to using word overlap based metrics is to consider the meaning of each word as defined by a word embedding. Methods such as *Word2Vec* (Mikolov et al 2013 [64]) calculate these embeddings using distributional semantics and word vectors are then aggregated in sentence vectors with some heuristic. To compare the ground truth r and the retrieved response $\hat{r}$, it is used the cosine similarity between their sentence level embeddings:

$$EA = cos(e_r^{avg}, e_{\hat{r}}^{avg}). \tag{5.5}$$

Embedding based metrics focus at extracting a sentence vector for both the candidate and the target sequence of words and then to compare these sentence vectors by using a similarity metric, for example the cosine similarity. There are several methods to aggregate word vectors into a sentence vector. An example are *Embedding average* and *Embedding vector extrema.*

Embedding average computes the sentence vector by using an additive composition, a method for computing the meanings of phrases by averaging the vector representations of their constituent words (Foltz et al 1998 [28], Laudauer et al 1997 [51], Mitchell et al 2008 [66]). This method has been widely used in other domains, for example in textual similarity tasks (Weiting et al 2015 [104]). The embedding average e^{avg} is defined as the mean of the word embeddings of each token in a sentence r:

$$e_r^{avg} = \frac{\sum_{w\in r} e_w}{|\sum_{w'\in r} e_{w'}|}. \tag{5.6}$$

Another way to calculate sentence level embeddings is using vector extrema (Forgues et al 2014 [29]). For each dimension of the word vectors, it takes the most extreme value among all the word vectors in the sequence, and uses that value in sentence-level embedding:

$$e_{rd}^{max} = \begin{cases} \max_{w\in r} e_{wd} & if\ e_{wd} > |min_{w'\in r} e_{w'd}| \\ \min_{w\in r} e_{wd} & otherwise \end{cases} \tag{5.7}$$

	Twitter				Ubuntu			
Metric	**Spearman**	p-value	**Pearson**	p-value	**Spearman**	p-value	**Pearson**	p-value
Greedy	0.2119	0.034	0.1994	0.047	0.05276	0.6	0.02049	0.84
Average	0.2259	0.024	0.1971	0.049	-0.1387	0.17	-0.1631	0.10
Extrema	0.2103	0.036	0.1842	0.067	0.09243	0.36	-0.002903	0.98
METEOR	0.1887	0.06	0.1927	0.055	0.06314	0.53	0.1419	0.16
BLEU-1	0.1665	0.098	0.1288	0.2	-0.02552	0.8	0.01929	0.85
BLEU-2	0.3576	< 0.01	0.3874	< 0.01	0.03819	0.71	0.0586	0.56
BLEU-3	0.3423	< 0.01	0.1443	0.15	0.0878	0.38	0.1116	0.27
BLEU-4	0.3417	< 0.01	0.1392	0.17	0.1218	0.23	0.1132	0.26
ROUGE	0.1235	0.22	0.09714	0.34	0.05405	0.5933	0.06401	0.53
Human	0.9476	< 0.01	1.0	0.0	0.9550	< 0.01	1.0	0.0

Figure 5.11: Correlation between each metric and human judgement for each response. Correlations for the human row result from randomly dividing human judges into two groups. Figure coming Liu et al 2017, "How to not evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation." [59].

where d indexes the dimension of a vector and e_{wd} is the d-th dimension of e_w.

Intuitively, this approach prioritizes informative words over common ones; words that appear in similar contexts will be close together in the vector space. Thus, common words are pulled towards the origin because they occur in various contexts, while words carrying important semantic information will lie further away. By taking the extrema along each dimension, it is thus more likely to ignore common words.

Greedy matching is the one embedding-based metric that does not compute sentence level embedding. Instead, given two sequences r and $\hat{r}$, each token $w \in r$ is greedily matched with a token $\hat{w} \in \hat{r}$ based on the cosine similarity of their word embeddings (e_w), and the total score is then averaged across all words:

$$G(r, \hat{r}) = \frac{\sum_{w \in r} \max_{\hat{w} \in \hat{r}} cos\_sim(e_w, e_{w'})}{|r|}, \tag{5.8}$$

$$GM(r, \hat{r}) = \frac{G(r, \hat{r}) * G(\hat{r}, r)}{2}. \tag{5.9}$$

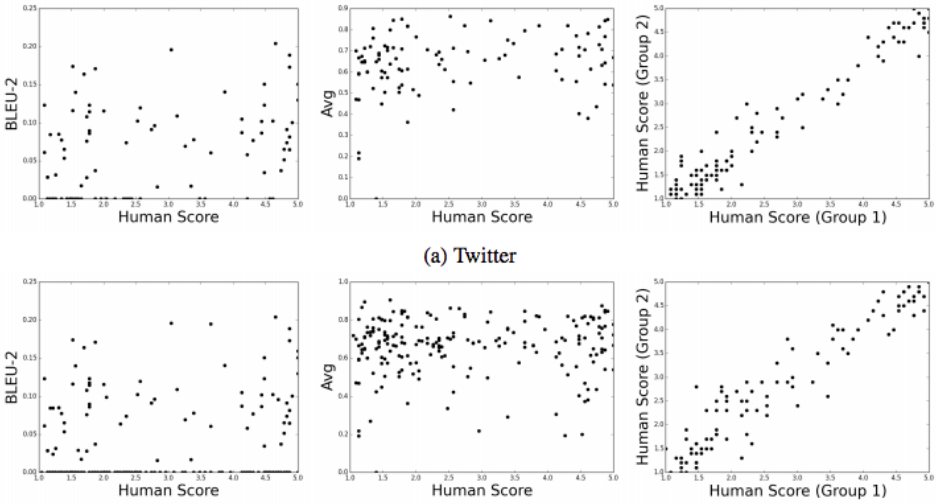
This approach was initially implemented in intelligent tutoring systems (Rus et al 2012 [79]).

5.4.3 Comparison between word overlap and word embedding metrics

Liu et al 2017 [59] also proposed an interesting correlation analysis, in which they showed how much the current metrics are related to the human judgement. An example can be found in picture 5.11. They also tried to calculate *BLEU* score after removing stopwords and punctuation from the responses,

	Spearman	p-value	**Pearson**	p-value
BLEU-1	0.1580	0.12	0.2074	0.038
BLEU-2	0.2030	0.043	0.1300	0.20

Figure 5.12: Correlation between BLEU metric and human judgement after removing stopwords and puctuation for the Twitter dateset. Figure coming Liu et al 2017, "How to not evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation." [59].



(a) Twitter

(b) Ubuntu

Figure 5.13: Scatter plots showing the correlation between metrics and human judgements on the Twitter corpus (on the top) and on the Ubuntu Dialogue Corpus (on the bottom). The plots representBLEU-2 (left), embedding average(center) and correlation between two randomly selected halves of human responsents(right). Figure coming Liu et al 2017, "How to not evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation." [59].

Figure 5.12, finding that this weakens the correlation with human judgements from *BLEU-2* compared with the ones of Figure 5.11.

As it is showed in the picture 5.13, there is a very weak correlation and in both cases the metric seems to be random noise.

In conclusion, results on the proposed embedding metrics are shown in Figure 5.14. For retrieval models, the Dual encoder model described in Section 3.2.3 clearly outperforms both TFIDF baselines on all the metrics.

On the other hand HRED [86], representing the equivalent of the one described in Section 3.2.8 without attention, significantly outperforms the LSTM generative model, an architecture that encodes the concatenated context.

Considering *BLEU-4* score, where 4 is the maximum order n of n-grams taken into consideration by the metric, few words have to be said in order

	Ubuntu Dialogue Corpus			Twitter Corpus		
	Embedding Averaging	Greedy Matching	Vector Extrema	Embedding Averaging	Greedy Matching	Vector Extrema
R-TFIDF	0.536 ± 0.003	0.370 ± 0.002	0.342 ± 0.002	0.483 ± 0.002	0.356 ± 0.001	0.340 ± 0.001
C-TFIDF	0.571 ± 0.003	0.373 ± 0.002	0.353 ± 0.002	0.531 ± 0.002	0.362 ± 0.001	0.353 ± 0.001
DE	**0.650 ± 0.003**	0.413 ± 0.002	0.376 ± 0.001	**0.597 ± 0.002**	0.384 ± 0.001	0.365 ± 0.001
LSTM	0.130 ± 0.003	0.097 ± 0.003	0.089 ± 0.002	0.593 ± 0.002	**0.439 ± 0.002**	**0.420 ± 0.002**
HRED	0.580 ± 0.003	**0.418 ± 0.003**	**0.384 ± 0.002**	**0.599 ± 0.002**	**0.439 ± 0.002**	**0.422 ± 0.002**

Figure 5.14: Metrics evaluated using the vector-based evaluation metrics, with 95% confidence intervals. Figure coming Liu et al 2017, "How to not evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation." [59].

to interpret its results. In fact, its values are very difficult to be read in the Conversational AI scope. This happens because, as the maximum order of n-grams increases, the probability for the *BLEU-n* score of being zero, and consequently of loosing any information, grows. This happens because the rareness of n-grams grows as n increases, and for, in the case of $n = 4$, it is very likely that the prediction does not have any 4-grams in common with the ground truth sample. As a result, for values of n bigger than 3, it is very likely to not have common n-grams between the predicted and the ground answer, causing the n-th precision to be equal to zero.

Given that the whole *BLEU-n* score is defined as the weighted sum of the logarithm of the precisions, it is enough to have the input to one logarithm equal to zero to add a $-\infty$ term in the sum, that squashes the whole sum to $-\infty$, no matter which are the values of the other precisions. This is the reason why a lot of computed *BLEU-n* scores, with $n > 3$, are zero. In these cases, the *BLEU-n* score is not able to provide an informative content about the prediction.

Variants of *BLEU* also use sentence-level smoothing (Chen et al (2014) [17]) to fix this issue, a method to ensure that geometric mean never gets to zero. In order to provide a baseline set of results, this work considers plain *BLEU-4*, without smoothing.

In conclusion, the depicted picture seem to be quite confusing. The metrics do not correlate with human judgement but picture 5.14 shows that these metrics differentiate models of different quality. The point here is that, even if the metrics show to have some meaning, this meaning is not related with what is important for a dialogue system, and for this reason it is not safe to take them as indicators for ranking models.

The analysis of Liu et al 2017 [59] only considers unconstrained domain, that is an open domain. Other dialogue settings, such as close domain applications, could potentially find stronger correlation with the *BLEU* metric for their lower diversity.

Liu et al 2017 [59] also asserted that, despite the poor performance the word embedding-based metrics had in their survey, they believe that metrics based on distributed sentence representations hold the most promise for the future. This happens because word-overlap metrics requires too many ground-truth responses to be meaningful, due to the high diversity of dialogue responses. The bad performance of the embedding based metrics could potentially be related to the fact that deterministic aggregation algorithms are not enough complex for modelling sentence level compositionality in dialogue.

Chapter 6

Results

As a contribute to the Conversational AI community, this thesis aims at presenting a structured evaluation of the results that different model architectures obtain on the dataset presented in Table 4.1, that is interesting because it comes from real world and it is pretty complex for its entropy.

In particular, both *model dependent metrics*, such as perplexity [101], defined in equation 5.4, and *model independent metrics*, such as *ROUGE* [56], *BLEU* [71] and *accuracy*, are used to asses the supremacy of an approach with respect to another. These quantitative metrics are then used to drive a cross validation procedure, with the goal of finding the best performing model.

In particular, two approaches are compared, one based on *curriculum learning*, explained in Section 5.2.1, and the other one exploiting *traditional learning*, Section 5.2.2. At the end, the best model is evaluated through a qualitative analysis. Each evaluated model is trained in an asynchronous distributed way exploiting data parallelism. In each Figure of this Chapter on the y-axis we plot the metric specified in the caption with respect to the number of training steps (x-axis).

6.1 Curriculum learning quantitative analysis

In this scope, a particular kind of data parallelism is exploited: each one of the workers retains a pointer to a specific dataset among the ones previously generated, and asynchronously sends updates to the parameter server basing on its dataset.

Interestingly, data parallelism here becomes *turn parallelism*. Furthermore, in order to support curriculum learning, the workers adopt the learning rate decay described in Section 5.2.1, while the master takes lr_0, the initial learning rate of the initial decay used by the workers, and adopts a constant learning rate equal to $lr_0 * 0.7$.

It can be seen how much complex is the parameter update: gradients are sparse, asynchronous and come from different optimization tasks. The assumption here is that such a complex model as the hierarchical attention can be trained in this very particular setting and that curriculum learning, together with the learning rate decay of Section 5.2.1, helps to reach convergence.

Practically, a first set of models is trained in a cluster where the master uses a two turns dataset and the workers perform two epochs of curriculum learning, one on one turn parsed train samples, the other on the two turns training set. This set of models is composed of four variations, each one sharing the configuration of Table 5.1 and using a different value of the recurrent cell:

- **NAS:** Implemented in *tf.contrib.rnn.NASCell*, described in Zoph et al (2016) [114], represented in the plots by the blue line.
- **LSTM:** Implemented in *tf.contrib.rnn.BasicLSTMCell*, described in Zaremba et al (2014) [112], represented in the plots by the pink line.
- **GRU:** Implemented in *tf.contrib.rnn.GRUCell*, described in [19]. and [84], represented in the plots by the red line.
- **UGRNN:** Implemented in *tf.contrib.rnn.UGRNNCell* and described in [22], represented in the plots by the light blue line.

In particular, in this section each plot shows the trends of several evaluation metrics, referring each validated model with a line of a different colour, light blue for *UGRNN* [22], red for *GRU* [84], blue for *NAS* [114] and dark pink for *LSTM* [112]. This analysis considers both model dependent and model independent metrics and its goal is to show which is the best recurrent cell to be used in the curriculum learning setting.

In Figure 6.1, 6.2 and 6.3 we present the perplexities, computed with Equation 5.4, that each different model obtains on three evaluation sets, respectively made of samples parsed in one turn, two turns and three turns.

By inspecting the evaluation perplexities, it is pretty clear that there are two sets of models. The first, composed by *UGRNN* cell [22] and *GRU* cell [84], is outperformed by the second, made of *LSTM* cell [112] and *NAS* cell [114]. In particular, among the best performing set, the second model based on the *LSTM* cell shows less variance in the results, and for this reason is selected as the best one with respect to model dependent metrics.

In addition, there is a clear trend that each plot presents. The perplexity drops down very fast for $40k$ steps, then it grows for some time to decrease slowly. This is caused by the fact that at $40k$ steps it starts the second step of curriculum learning and the dataset changes from the one parsed in one turn to the one in two turns.

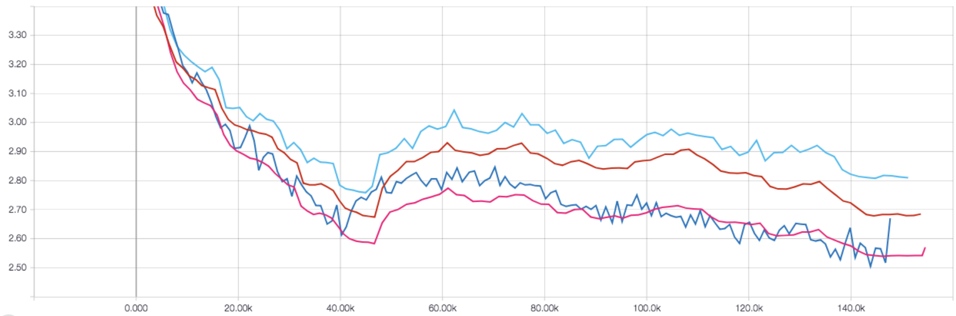


Figure 6.1: Perplexities obtained by the models trained with curriculum learning on the 1-turn evaluation set. The Blue line represents the NAS based model, Light blue the UGRNN based one, red the GRU based one, and pink the LSTM based one.

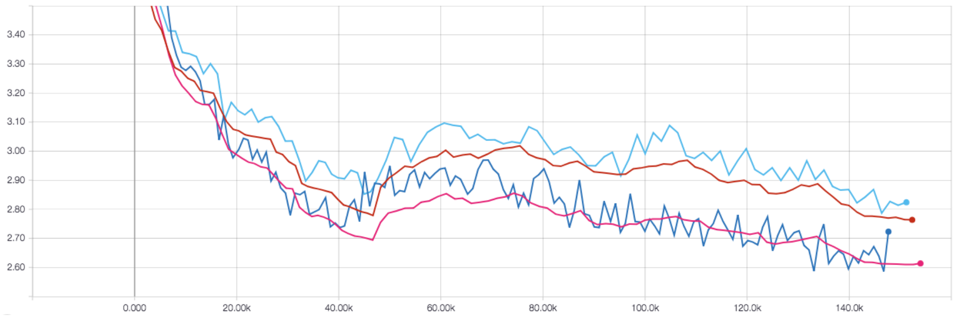


Figure 6.2: Perplexities obtained by the models trained with curriculum learning on the 2-turns evaluation set. Blue NAS based model, Light blue UGRNN, red GRU, pink LSTM.

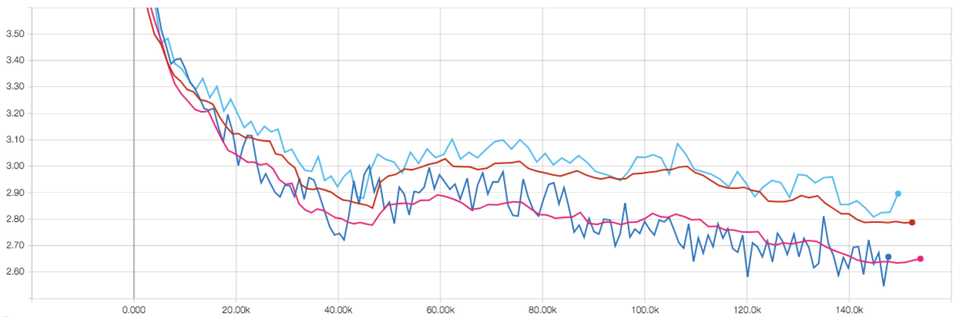


Figure 6.3: Perplexities obtained by the models trained with curriculum learning on the 3-turns evaluation set. Blue NAS based model, Light blue UGRNN, red GRU, pink LSTM.

Another interesting behaviour is that, if during the first step of curriculum learning the three cells do not show that big difference in terms of performance, the second step instead amplifies a lot the difference. In fact, the gap between the first and the second set of models is roughly 0.3 in the first step and 0.5 in the second. This observation shows that this learning procedure could potentially help the developer in performing cross validation. An intuition that could explain why the performance changes that much in the second curriculum learning phase could be in the strong perturbation caused by the dataset switch. In fact, the goal of the optimization changes and the model is asked to adapt to this phenomenon, that means figuring out how to move from a set of weights that were optimal for the previous curriculum learning step, to a new set of weights, optimal for the new learning objective. In this order of ideas, the just noticed difference in the performance could be due to the fact that some recurrent cells are more suited to adapt to such hard perturbations than others. In particular, it is clear that *LSTM* [112] is the best model with respect to model dependent metrics in terms of its adaptation capability. In fact it is the best performing model in roughly all the curriculum learning steps, it adapts very quickly to the dataset switch and also shows less variance with respect to the other best performing model, that is *NAS* [114]. In addition, the initial learning rate used in the beginning of the staircase decay of Figure 5.1 could be too high to support such a complex learning algorithm.

Interesting could also be to study how increasing the number of epochs for each curriculum learning step changes the overall performance of the model. On one side, using too many epochs could bring to values of the weights that are too much over-fitted for the current dataset, and this can make very hard for the next curriculum learning step to converge. On the contrary instead, it could happen that using several epochs brings to better and more stable solutions.

Another parameter that is interesting to tune is the batch size. In these experiments it is used a value of 128, the maximum allowed to make a model fit in a single K80 GPU. More complex architectures, combining model parallelism with data parallelism, could allow to use higher batch size and so to perform safer updates with less variance. Again, here is clear how this work wants to be a first head into this world, presenting a baseline architecture both in terms of model complexity, it does not use backward sentence level encoder and the algorithm does not adopt the dependence of the word level encoder to the sentence one, and in terms of training complexity, meaning that only data parallelism is adopted while a combination with model parallelism could bring to better solutions.

The best obtained perplexity is 2.53 on the one turn, 2.61 on the two turns and 2.63 on the three turns evaluation dataset. An interesting aspect to be pointed out is that the three evaluation plots referring to model dependent metrics show exactly the same trend even if they refer to datasets with an increasing complexity.

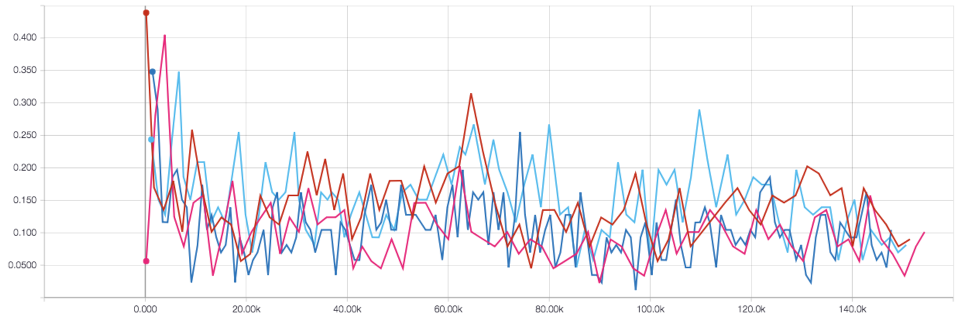


Figure 6.4: Accuracy obtained by the evaluated models on the 1-turn evaluation set. Blue NAS based model, Light blue UGRNN, red GRU, pink LSTM.

6.1.1 Model independent metrics

While in the previous section was explained how model dependent metrics give an informative signal about the quality of the model, in this section we show how independent measures are not very useful during the evaluation phase of curriculum learning based algorithms, demonstrating the thesis of Liu et al (2016) [59].

In particular, the models are evaluated through the analysis of the trends of three external metrics: *sentence level accuracy*, *ROUGE* [56] and *BLEU-4* [71] and the assessment of the word-embedding based ones are left to future work. Section 7 shows, from an high level, how to build this new kind of evaluation module.

Considering *sentence level accuracy*, Figure 6.4 shows, with the same colours explained above, the accuracy obtained on the first evaluation set by the four models. From this Figure we can see how much noisy are the performances and how they do not show any trend. This happens because accuracy is computed sentence level, meaning that if the target sequence is (w_0, w_1, w_2, w_3) and the model generates $(w_{100}, w_0, w_1, w_2, w_3)$, the accuracy is zero because the first token w_{100} makes the two sequences two different labels. From the smoothed version of Figure 6.5 is pretty clear how this metric pretty much useless, demonstrating the thesis of Liu et al (2016) [59].

Again, plot 6.6 shows how the *ROUGE* metric is uncorrelated with the model strength in handling conversations. In fact it stays constant on four *ROUGE* points. Differently with accuracy, it shows a little drop at $40k$ steps, the end of the first curriculum learning phase, demonstrating that it suffers a bit of the dataset switch. Unfortunately, after that moment it returns quickly to the previous value and stays there constantly.

For these above reasons, this metric can be considered a little bit more correlated with the model behaviour with respect to accuracy. However, given that it does not show any increasing or decreasing trend and remains

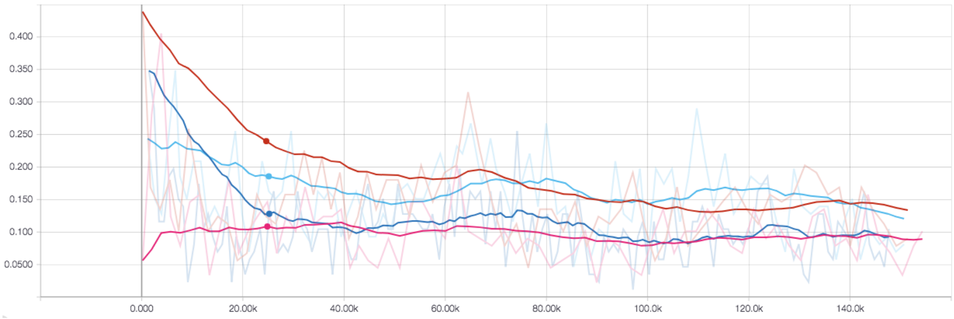


Figure 6.5: Smoothed accuracy obtained by the evaluated models on the 1-turn evaluation set. Blue NAS based model, Light blue UGRNN, red GRU, pink LSTM.

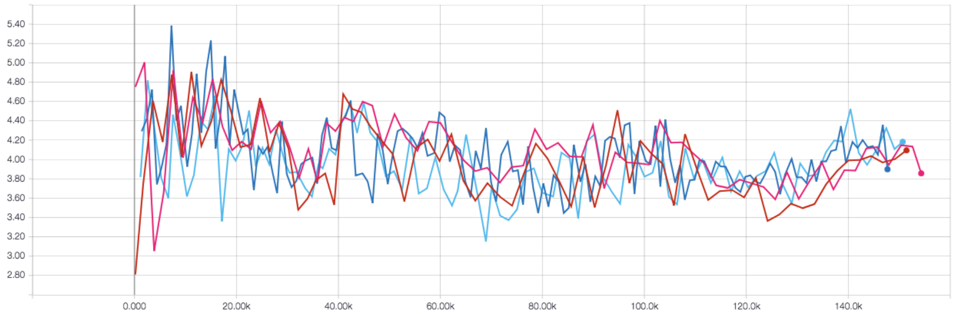


Figure 6.6: Rouge scores obtained by the evaluated models on the 1-turn evaluation set. Blue NAS based model, Light blue UGRNN, red GRU, pink LSTM.

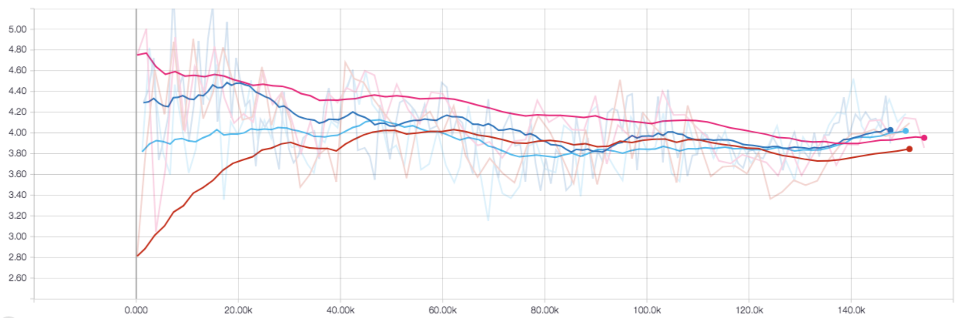


Figure 6.7: Smoothed rouge scores obtained by the evaluated models on the 1-turn evaluation set. Blue NAS based model, Light blue UGRNN, red GRU, pink LSTM.

constant exception made for step $40k$, also this metric is considered not informative to discriminate the model capacity. This demonstrates again the thesis of Liu et al (2016) [59].

A quite different situation happens for *BLEU-4* [71]. In Figure 6.8, 6.10 and 6.12 we show the not smoothed *BLEU-4* trends, and it can be seen that they are very noisy. To clarify these patterns, we show in Figures 6.9, 6.11 and 6.13 the smoothed plots, where it is possible to better notice the high

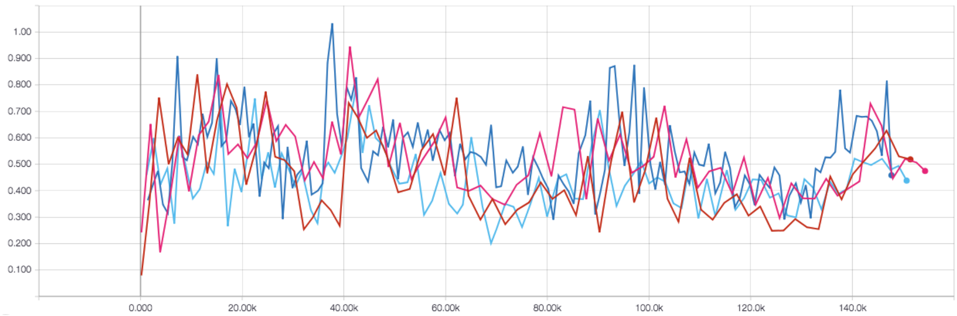


Figure 6.8: BLEU-4 [71] scores obtained by the evaluated models on the 1-turn evaluation set. Blue NAS based model, Light blue UGRNN, red GRU, pink LSTM.

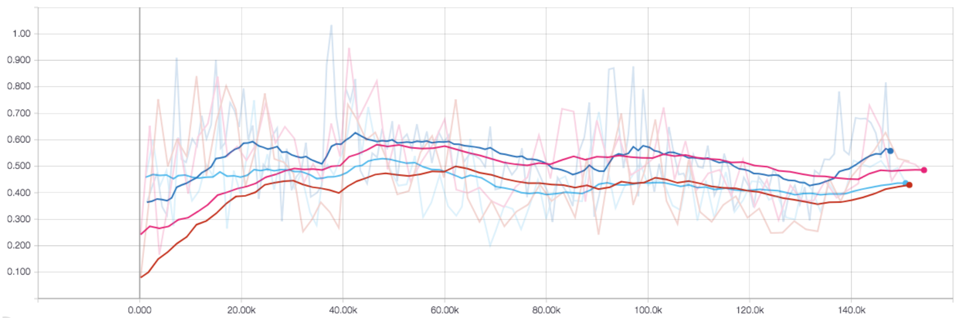


Figure 6.9: Smoothed BLEU-4 [71] scores obtained by the evaluated models on the 1-turn evaluation set. Blue NAS based model, Light blue UGRNN, red GRU, pink LSTM.

level patterns.

First of all, in Figures 6.8 and 6.9 can not be noticed any clear increasing trend. The performance always remains lower than one *BLEU-4* point and increases during the first curriculum learning step, that finishes at $40k$. After that, when the dataset changes, it starts a long decreasing curve which finishes roughly at $120k$ steps, where it starts increasing again resuming the trend it left after the first curriculum learning step. This very weak trend can be interpreted by saying that during the first steps the model is optimizing by using a dataset parsed as the evaluation set, and so the evaluation *BLEU-4* increases. In the second step instead, it is fooled by the change of dataset and it takes a lot to converge to a better solution. In this last example can be seen how much difficult is to train a model with this new learning procedure.

A weak correlation with model independent metrics is instead shown in Figure 6.10 and 6.11, presenting the *BLEU-4* points obtained on the two turns parsed evaluation set. These last two figures show again an increase in the first curriculum learning step but, differently with the previous plots, it can be noticed a clear increasing trend for some recurrent cells during

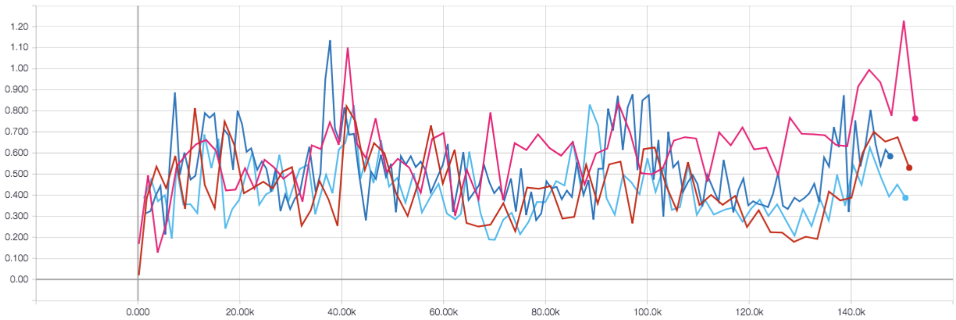


Figure 6.10: BLEU-4 [71] scores obtained by the evaluated models on the 2-turns evaluation set. Blue NAS based model, Light blue UGRNN, red GRU, pink LSTM.

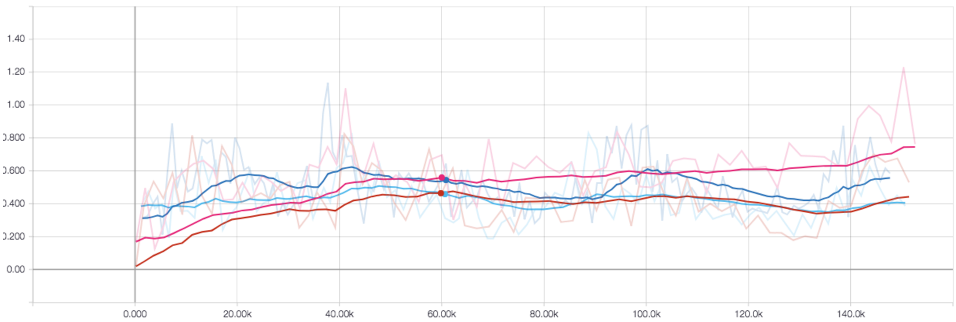


Figure 6.11: Smothed BLEU-4 [71] scores obtained by the evaluated models on the 2-turns evaluation set. Blue NAS based model, Light blue UGRNN, red GRU, pink LSTM.

the second curriculum step, and this is intuitive given that now the model is optimizing by using a dataset that is parsed in the same way of the evaluation set.

We can notice how the *LSTM* cell [112] clearly outperforms *UGRNN* [22], *GRU* [84] and *NAS* [114], that seem to show the same, but weaker, increasing trend. We can find an even nicer pattern in Figures 6.12 and 6.13. Here the *BLEU-4* score is computed on a three turns parsed evaluation set and the ramp is even clearer. However, this is counter intuitive given that the model now is never optimized by using the three turns dataset. In fact, it is using one turn parsed samples in the first step and two turns data in the second step.

As it could be noticed in two turns evaluation, also in three turns validation the *LSTM* clearly outperforms the other three recurrent cells showing a cleaner trend.

A difference between the figures of model dependent and model independent metrics is that, while the first ones show an equivalent performance for both *NAS* and *LSTM* based models, from an analysis of model independent metrics it is clear that with the *LSTM* cell the model performs better.

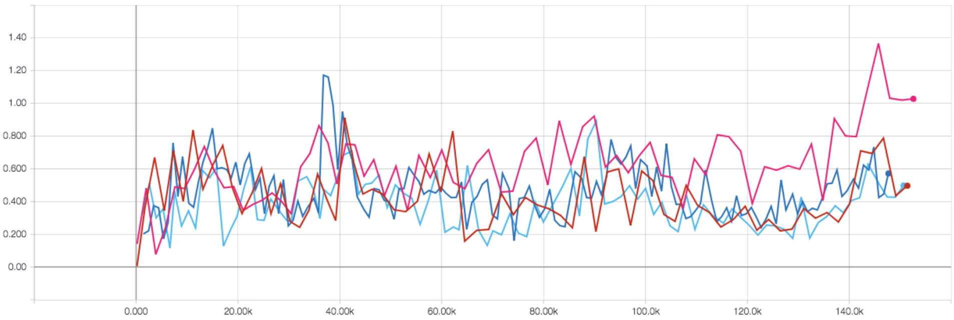


Figure 6.12: BLEU-4 [71] scores obtained by the evaluated models on the three turns evaluation set. Blue NAS based model, Light blue UGRNN, red GRU, pink LSTM.

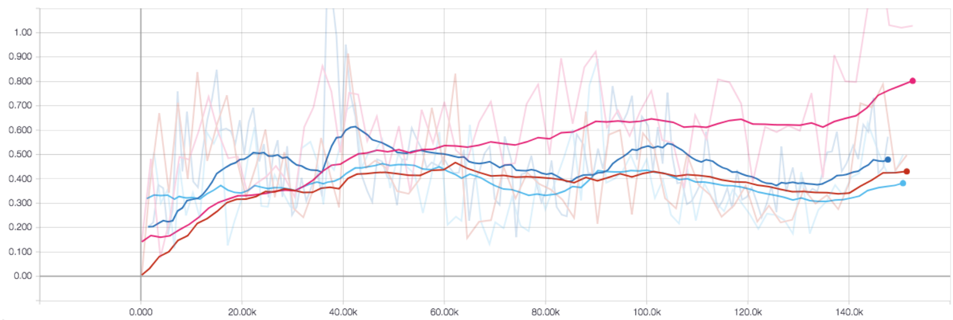


Figure 6.13: Smoothed BLEU-4 [71] obtained by the evaluated models on the 3-turns evaluation set. Blue NAS based model, Light blue UGRNN, red GRU, pink LSTM.

If *ROUGE* [56] and accuracy are not correlated at all with model dependent metric, *BLEU-4* score instead shows that at least a bit it is useful to discriminate the performances.

Another problematic issue of using *BLEU* to evaluate Conversational AI is to understand what is a good value for it. It is known that the *Transformer*, the very best model for machine translation, obtains 28 *BLEU-4* points, but its goal is to model not entropic data, and this value can not be used as an indicator for Conversational AI, first of all because it uses a different architecture, and lastly because of the motivations explained in Liu et al (2016) [59]. At the end, even if the final evaluation is usually done through a qualitative analysis, inspecting the trend of the *BLEU* score could be interesting to see, from an high level point of view, if the model is behaving well during intermediate steps.

In all the above plots can be seen how the models, if run for more than one epoch per curriculum learning step and with a bigger batch size, could potentially show their trends better and with less variance, and this analysis is left to future work. As a conclusion, in this first evaluation step we found out that, from a mere quantitative analysis, *LSTM* cell is the best recurrent unit to be used in the curriculum learning setting because it outperforms

Color	**Unit**	**Dropout**	**Pretrain**	**Reverse**	**Warmup**	**Batch**
Cyan	LSTM	0,20	True	True	12k	128
Blue	LSTM	0,15	True	True	12k	128
Red	LSTM	0,10	True	True	20k	128
Green	NLSTM	0,10	True	True	20k	128
Grey	NLSTM	0,15	True	True	20k	256
Pink	NLSTM	0,15	False	True	12k	128
Orange	LSTM	0,15	True	False	9k	128

Table 6.1: Different settings of traditional learning based models.

the other architectures in both qualitative and quantitative metrics.

6.2 Traditional learning quantitative analysis

In parallel, we validated a second set of seven models using the traditional learning procedure described in Section 5.2.2. Four of them use *LSTM* [112], while the other three use *Layer Normalized LSTM* [3]. Each setting shares the values of Table 5.1 and updates them with the ones of Table 6.1. The best performing model for traditional learning is extracted and compared with the best one coming from curriculum learning.

Again, each validated model is trained in a distributed fashion where, differently from curriculum learning, each process always uses samples coming from a one turn parsed training set. For this reason, while the previous approach implemented turn parallelism, here only data parallelism is used.

Similarly to what happened in the previous section, here each model is identified by a colour, that is described by the first column of Table 6.1. First of all, given that the curriculum learning analysis found out that the *LSTM* cell is the best performing one, this evaluation only considers this cell together with its normalized version, called *NLSTM* (Layer Norm LSTM [3]) which functionalities are detailed in Section 3.2.1.

In order to find the best values for the six hyper-parameters of Table 6.1, the seven configurations are compared in groups, and during each comparison we give insights about how to choose the value of a specific parameter. The first and most important analysis regards the unit type, and for this reason the first evaluation step considers the models identified with the red and green colours, that share the exact same configuration and differ only for the unit, respectively *LSTM* for the red model and *Layer Normalized LSTM* for the green one. In particular, Figures 6.14, 6.15 and 6.16 show the perplexities that these models achieve on three evaluation sets, the same ones used for curriculum learning evaluation and respectively containing one turn, two turns and three turns parsed samples.

From Figure 6.14 we can notice that the two models perform very sim-

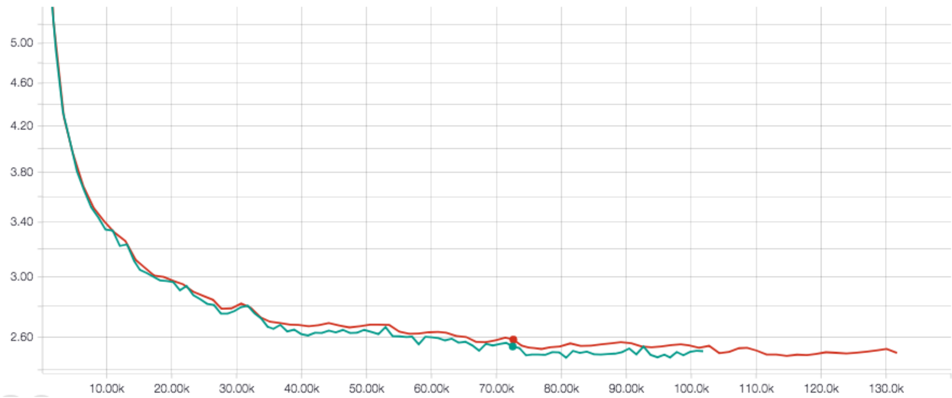


Figure 6.14: Perplexities that a LSTM based model, the red one, and a Layer Norm LSTM model, the green one, obtain on the 1-turn evaluation set.

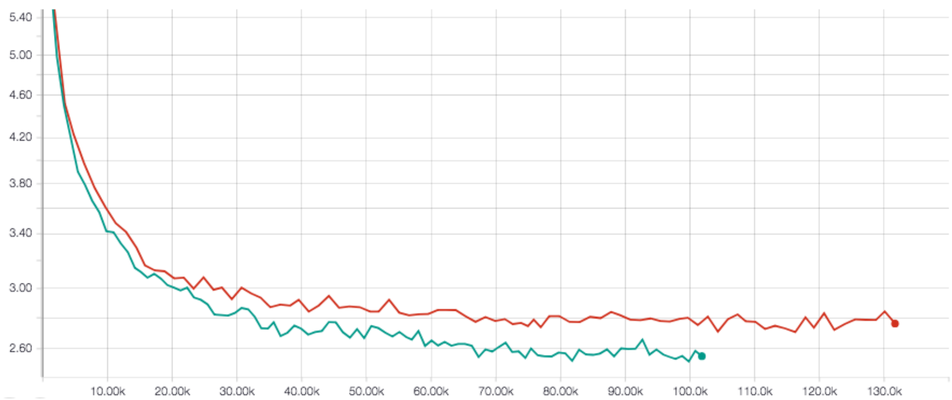


Figure 6.15: Perplexities that a LSTM based model, the red one, and a Layer Norm LSTM model, the green one, obtain on the 2-turn evaluation set.

ilarly on the one turn parsed evaluation set, even if the model using *Layer Norm LSTM* always shows to be better than the other one.

While this first figure already shows the supremacy of the normalized cell, this hypothesis is even enforced by the plots showing the perplexities achieved by the two models on the two turns, Figure 6.15, and on the three turns, Figure 6.16, evaluation sets. In fact, from the last two figure we can notice how the normalized cell is able to reach a significantly better result on more complex tasks, such as multiple turn evaluation. Interestingly, the effect of the normalization allows to show the same performance on the different evaluation datasets, while the other architecture keeps worsening as the number of turns increases. In particular, on the two turns parsed evaluation set, the green model obtains less than 2.6 while the *LSTM* obtains 2.8 and then stops improving. Considering the three turns case instead, is clear that the normalized model performs in the same way it did on two

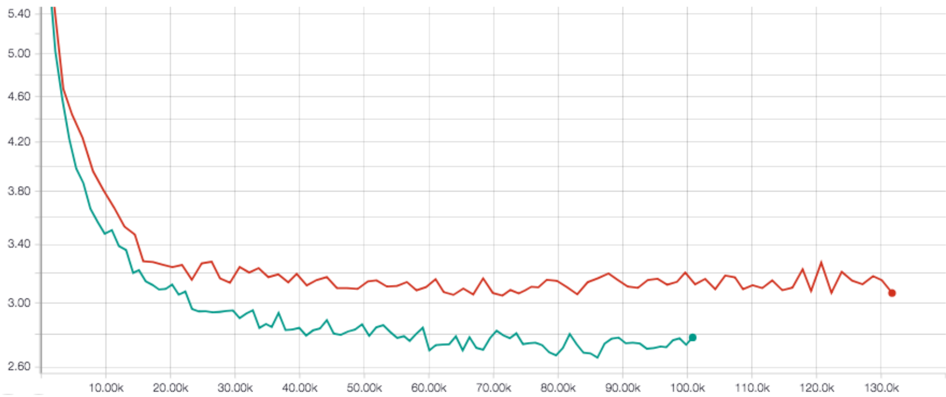


Figure 6.16: Perplexities that a LSTM based model, the red one, and a Layer Norm LSTM model, the green one, obtain on the 3-turn evaluation set.

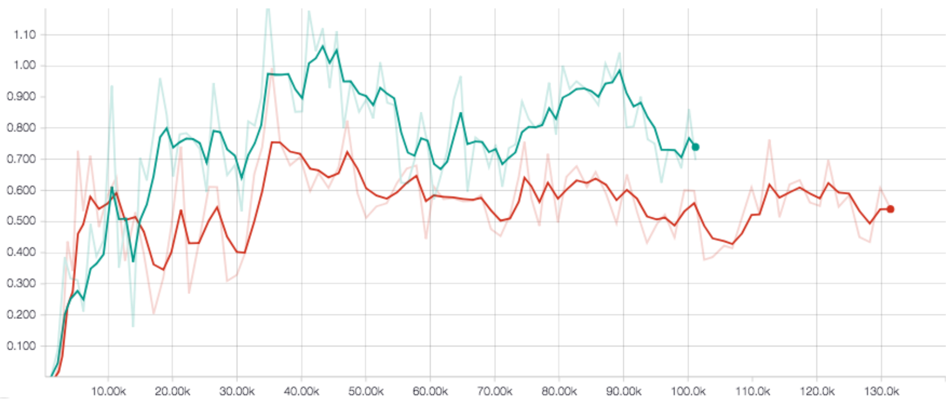


Figure 6.17: BLEU-4 scores that a LSTM based model, the red one, and a Layer Norm LSTM model, the green one, obtain on the 2-turns evaluation set.

turns data, while the other one is not even able to converge to a value less than 3, and it further starts diverging after $100k$ steps. We found the same pattern also in the plots regarding *BLEU-4* [71], as it is shown in Figure 6.17 that presents this evaluation metric computed on the two turns parsed evaluation set.

As a conclusion, it can be said that in traditional learning *Layer Norm LSTM* performs significantly better than *LSTM* cells, both in terms of quantitative and qualitative metrics.

A second interesting analysis regards the red, blue, cyan and orange model. Each one of them uses the *LSTM* cell, and while the first three use an increasing value of dropout, the last one differs with the blue model because it is fed with the reversed input sequence.

In particular, from Figure 6.18 we can notice how using the two extreme values of dropout, that are $0,10$ and $0,20$, results in a poor performance,

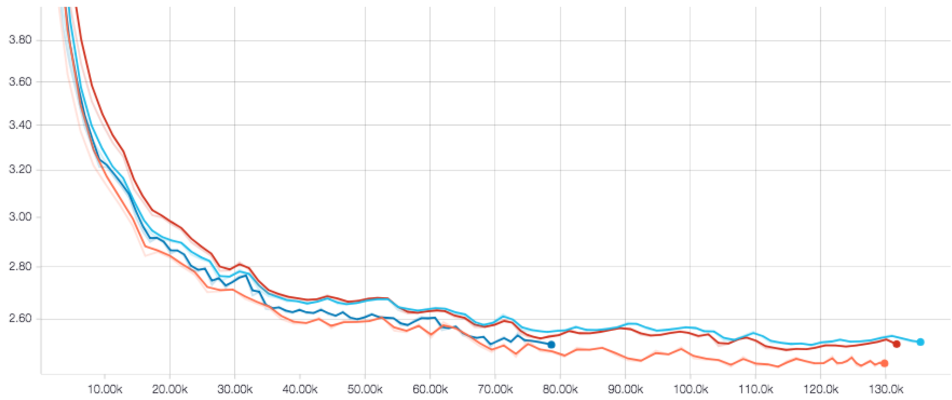


Figure 6.18: Perplexities obtained by four LSTM based models, three of them, red, blue and cyan, using different dropout probabilities, respectively 0,10 , 0,15 and 0,20, and the last one, the orange, using 0,15 and not reversing the input sequence. Values computed on the 1-turn evaluation set.

while adopting $0,15$, blue line, results in a better model. If Figure 6.18 shows a clear distinction between adopting $0,15$ and $0,20$, the figures about perplexity computed on two turns dataset, Figure 6.19, and three turns dataset, Figure 6.20, show that instead, from their point of view, the two choices are pretty much equivalent.

A similar behaviour can be also seen in Figure 6.21, where the model using $0,10$, the red line, is outperformed by the other two, which show a similar trends. For this reason, it is possible to say that the optimal dropout value is between $0,15$ and $0,20$, and in this range, plot 6.18 suggest to use values close to $0,15$. To summarize, from Figures 6.18, 6.19, 6.20 and 6.21, we extracted that dropout slightly affects the performances of the model and a value close to $0,15$ is the best choice.

Considering instead the way the input sequence is fed to the model, if in machine translation reversing the input of the encoder is a practice that showed great improvements, on the contrary, this analysis shows that in the scope of Conversational AI, it is better not to follow this intuition and to use a non reversed input. In fact the orange model, that does not reverse its input sequence, shows to be better than the other three, which instead reverse it, both in terms of model independent and model dependent metrics. As said in Section 3.2.5, in the translation setting the dependencies between source and target sequences are mainly *tok2tok aligned*. For this reason, reversing the input sequence is a practice that, putting close together the first positions of the two sequences, reduces the length of the dependencies and eases the task. Given that Conversational AI does not have the same dependencies of machine translation, it is intuitive that the same architecture will not work in this setting. This happens mainly because, in order to reduce the length of the dependencies, is often better to leave the last

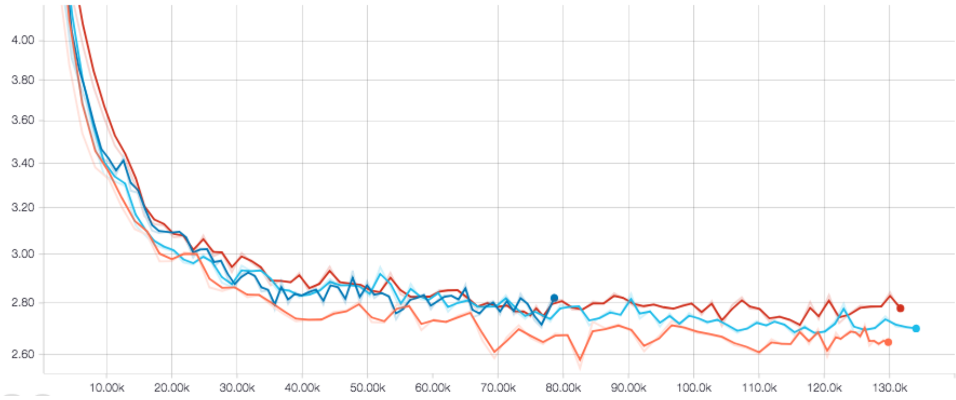


Figure 6.19: Perplexities obtained by four LSTM based models, three of them, red, blue and cyan, using different dropout probabilities, respectively 0,10 , 0,15 and 0,20, and the last one, the orange, using 0,15 and not reversing the input sequence. Values computed on the 2-turn evaluation set.

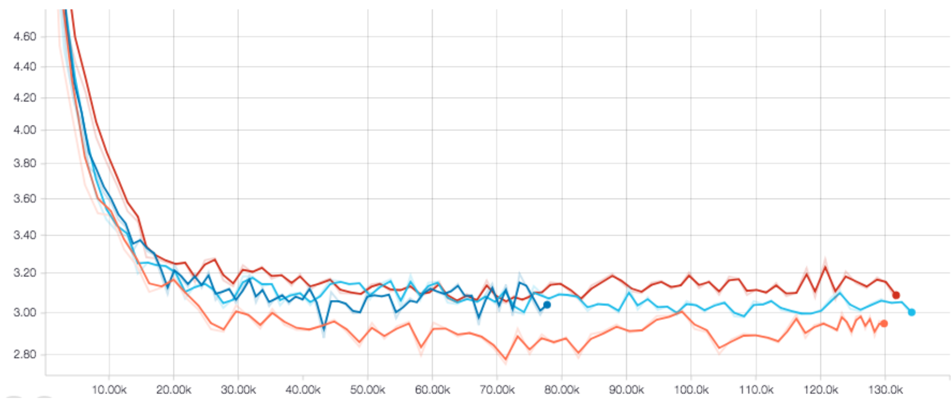


Figure 6.20: Perplexities obtained by four LSTM based models, three of them, red, blue and cyan, using different dropout probabilities, respectively 0,10 , 0,15 and 0,20, and the last one, the orange, using 0,15 and not reversing the input sequence. Values computed on the 3-turn evaluation set.

utterance of the context close to the answer generation process, given that it is more likely to have the last utterance being the most important part of the context to generate the answer.

We than perform a third analysis with which on one side we want to cross-validates the choice of pre-training or not the word embeddings, and on the other one we show how increasing the batch size improves the performance of the model. A first important insight can be extracted from Figures 6.22, 6.23 and 6.24, where it is clear how the grey model, using batch size equal to 256, greatly outperforms the other two in model dependent metrics. This is reasonable given that, with a bigger batch size, is possible to better approximate the function to be optimized, bringing to faster convergence.

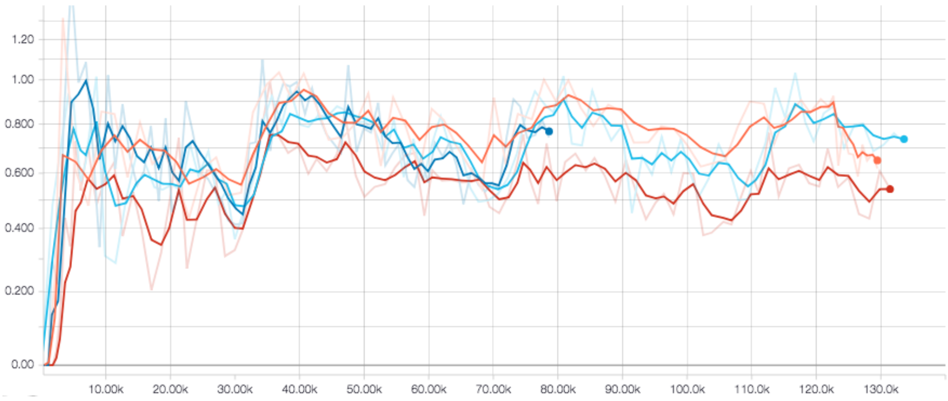


Figure 6.21: BLEU-4 scores obtained by four LSTM based models, three of them, red, blue and cyan, using different dropout probabilities, respectively 0,10 , 0,15 and 0,20, and the last one, the orange, using 0,15 and not reversing the input sequence. Values computed on the 2-turn evaluation set.

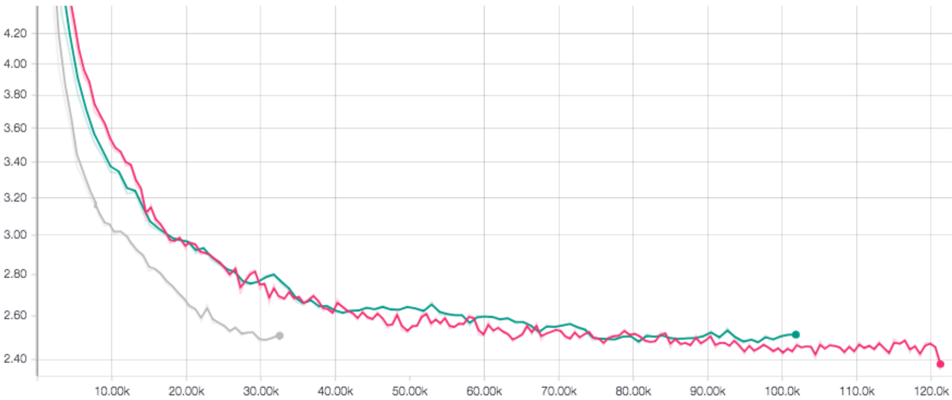


Figure 6.22: Perplexities obtained by three Layer Norm LSTM based models, one using batch size of 256, the grey line, another that does not pretrain the word embeddings, the pink line, and lastly the green one, that uses a low dropout probability, that is 0,10. Values computed on the 1-turn evaluation set.

Unfortunately, using 256 as the batch size is not always feasible. In fact, the physical setting we used, the one proposed in Section 5.1.1, is made of four GPU machines, each one having *12 MiB* of internal memory and being dedicated to a single specific process during distributed training. In fact, given that in this work each worker has its own version of the model and sends asynchronous updates to the central parameter server, computed by using its own dataset, all the computation has to be performed on a single GPU, and for this reason only a batch with size equal to 128 allows to not have *OOM* errors.

Given that, as said, the selected architecture could not support such a high batch size, the grey model shown in Figures 6.22, 6.23, 6.24, 6.25, 6.26

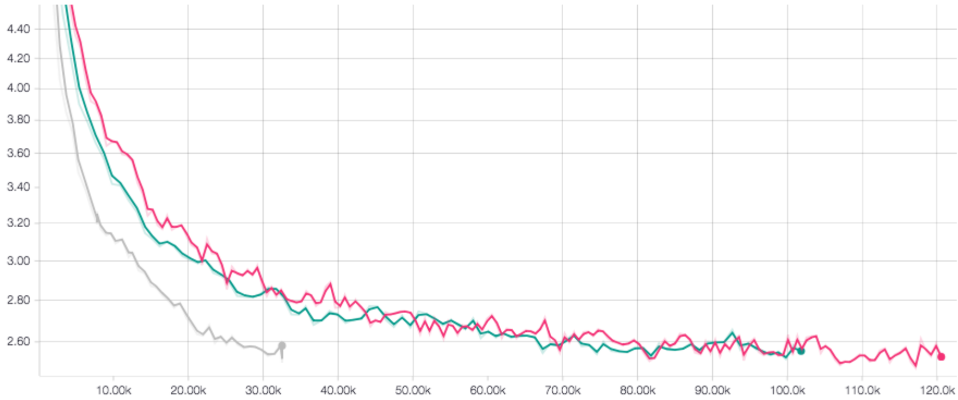


Figure 6.23: Perplexities obtained by three Layer Norm LSTM based models, one using batch size of 256, the grey line, another that does not pretrain the word embeddings, the pink line, and lastly the green one, that uses a low dropout probability, that is 0,10. Values computed on the 2-turn evaluation set.

Figure 6.24: Perplexities obtained by three Layer Norm LSTM based models, one using batch size of 256, the grey line, another that does not pretrain the word embeddings, the pink line, and lastly the green one, that uses a low dropout probability, that is 0,10. Values computed on the 3-turn evaluation set.

and 6.27 is the result of training the model on another machine equipped with four P100 NVIDIA GPUs, each one having *16 MiB* of memory.

As a conclusion, in order to use 256 as the batch size, it is necessary to either use P100 GPUs, or to adopt an approach that allows to use both data and model parallelism. This last setting refers to the case of distributing the nodes of the computational graph across the available GPUs, performing for example the operations of the encoder on the first GPU and remaining computation on the second GPU. This way, the training procedure exploits both data parallelism and model parallelism.

Another interesting aspect is to understand if the use transfer learning for word embedding matrices gives an improvements in the performance.

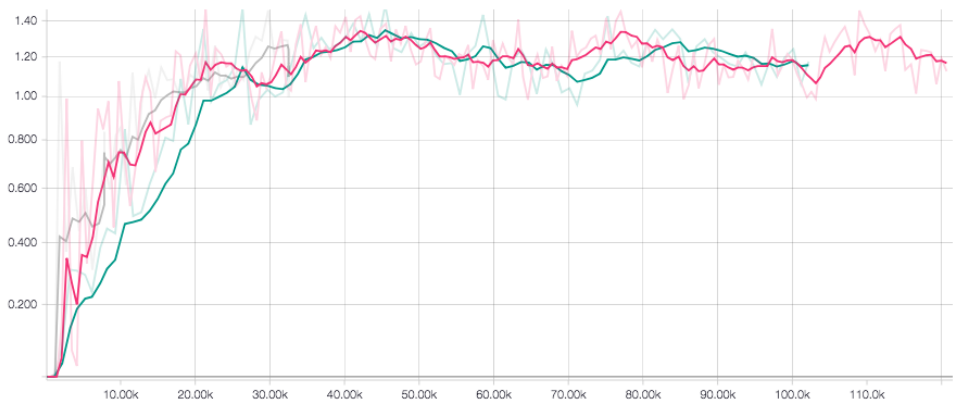


Figure 6.25: BLEU-4 scores obtained by three Layer Norm LSTM based models, one using batch size of 256, the grey line, another that does not pretrain the word embeddings, the pink line, and lastly the green one, that uses a low dropout probability, that is 0,10. Values computed on the 1-turn evaluation set.

Intuitively, exploiting this technique means pre-training word embedding matrices with a *Word2Vec* model, Mikolov et al 2013 [64], and this should result in a model that reaches convergence more easily. On the contrary, in Figures 6.25, 6.26 and 6.27 we show how this intuition is not correct, and training word embeddings from zero gives a better performance.

In fact, if Figure 6.25 does not highlight this difference that much, we noticed that, as the number of turns increases, the model that does not pre-train embeddings clearly outperforms the ones that use transfer learning for the embedding matrices, and this can be seen in Figures 6.26 and 6.27.

One could argue that, in the above figures, the model that does not pre-train embeddings also uses $0,15$ as the dropout probability, while the other one adopts $0,10$, and given that this choice was found to be crucial, it could be said that the cause of the better performance is in the dropout and not in the different initialization of the embedding matrices. To clarify this issue, the claim here is that the gap in the *BLEU-4* scores found in Figures 6.26 and 6.27 is not explainable by only considering the difference in the dropout probabilities of the two models. In fact, in Figure 6.21 the gap between the models using $0,10$ and $0,15$ as their dropout probability is not as huge as the one in Figure 6.27.

It is also very interesting to notice how this gap is found this huge only in figures showing the model independent performances on complex evaluation tasks, such as decoding of two turns and three turns unseen samples. In fact, in model dependent metrics and in the evaluation of the *BLEU-4* score on the one turn evaluation set, the two models are indistinguishable.

This again shows how much tricky is taking design choices for Conversational AI architectures. An explanation for this phenomenon is that training word vectors together with the architecture makes it more robust

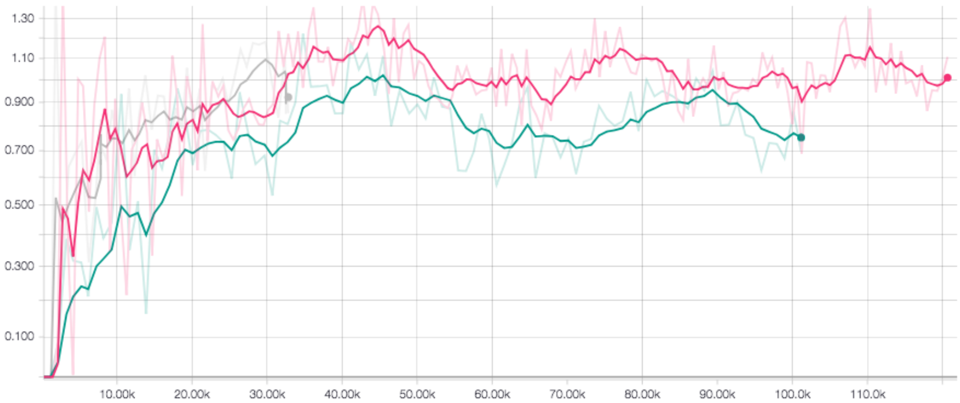


Figure 6.26: BLEU-4 scores obtained by three Layer Norm LSTM based models, one using batch size of 256, the grey line, another that does not pretrain the word embeddings, the pink line, and lastly the green one, that uses a low dropout probability, that is $0, 10$*. Values computed on the 2-turn evaluation set.*

and gives the ability to generalize better and to learn *Conversational AI word embeddings* instead of *general purpose word embeddings*. In fact, when using transfer learning, the developer does an implicit assumption about the portability of the value of the pre-trained weights. In some cases this assumption is correct, in other cases, such as pre-training word embeddings for Conversational AI, it is not.

Another nice property is that the performance gained by the pink model on the two turns and three turns reaches roughly the same value in both the cases while the green configuration instead becomes worst as the complexity of the evaluation task increases, and this is a good indicator of the generalization capability of the pink variant. Nicely, *BLEU-4* score is a little bit more correlated with model dependent metrics in traditional learning than in curriculum learning.

6.2.1 Best performing model

In Section 6.1 was found that the best performing model for curriculum learning is the one that adopts the *LSTM* recurrent cell, while the best in the traditional learning setting, Section 6.2, is the pink configuration of Table 6.1. In this section, the above two models are compared, both in terms of model dependent and independent metrics. The final goal is to find the best performing setting, and consequently the best approach, in order to perform a qualitative analysis of the best model. Specifically, Figures 6.28 6.29 and 6.30 show the per word perplexity that the two best models obtain respectively on the three evaluation sets used in this chapter. In these next three plots the best performing model for curriculum learning is identified with the blue line, while the best from traditional learning is represented

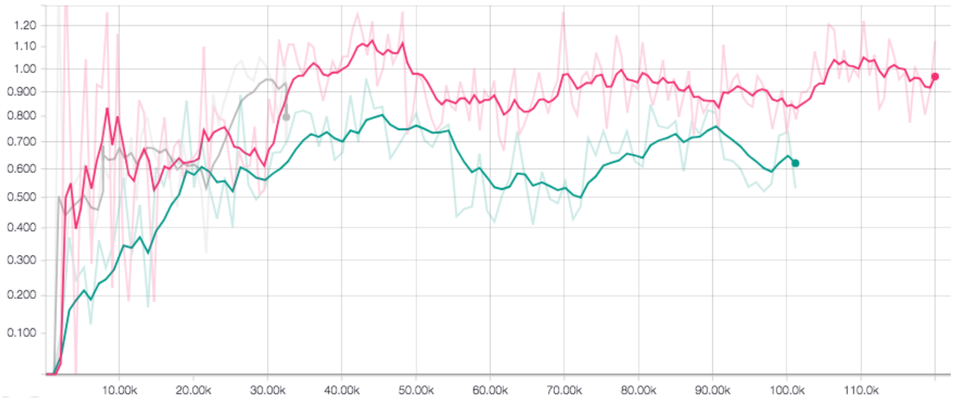


Figure 6.27: BLEU-4 obtained by three Layer Norm LSTM based models, one using batch size of 256, the grey line, another that does not pretrain the word embeddings, the pink line, and lastly the green one, that uses a low dropout probability, that is 0,10. Values computed on the 3-turn evaluation set.

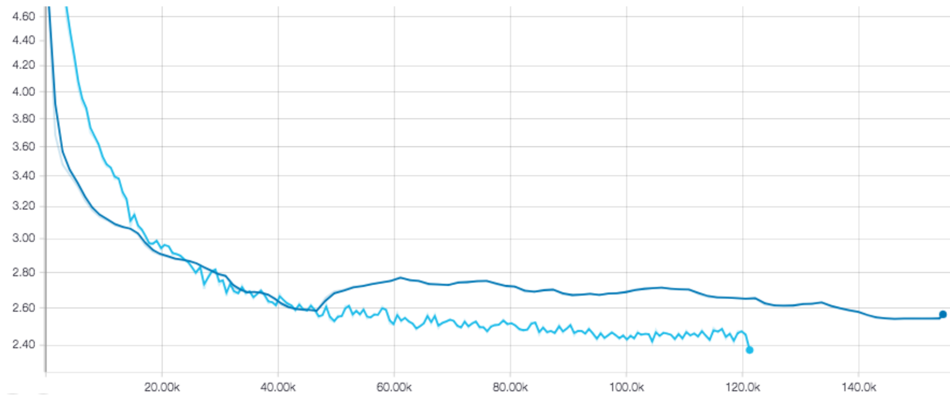


Figure 6.28: Perplexities obtained by the two best performing models. The blue model represents the best performing one with respect to curriculum learning while the cyan line identifies the best one in traditional learning. Values computed on the 1-turn evaluation set.

with the cyan coloured curve.

As expected, the traditional learning model is more stable and presents a much smoother trend, and this is true in all the three plots. Interestingly, the three figures show that during the first curriculum learning step, identifies between zero and $40k$ global steps, the performance of the blue model drops down faster than the cyan does, and this is due to the fact that the first step of curriculum learning starts with constant learning rate different from zero, while traditional learning starts with zero.

For this reason the cyan model, using a larger learning rate, at the very beginning is able to converge quicker to a smaller evaluation loss. After this first phase, in all the three figures the models join and their performance

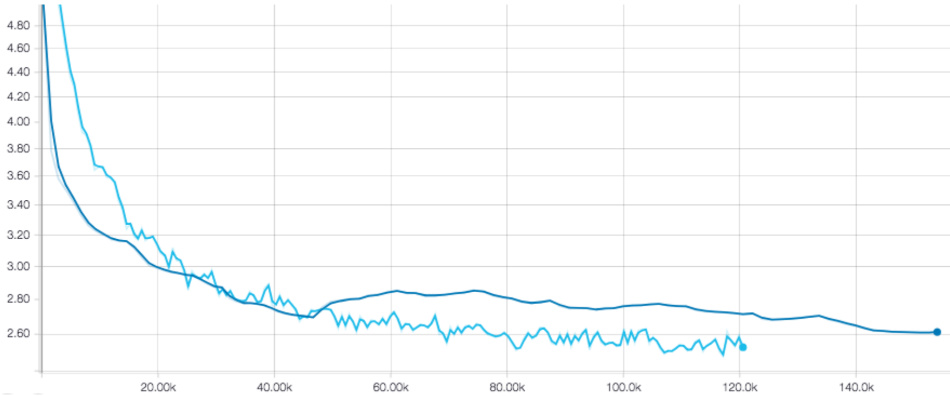


Figure 6.29: Perplexities obtained by the two best performing models. The blue model represents the best performing one with respect to curriculum learning while the cyan line identifies the best one in traditional learning. Values computed on the 2-turn evaluation set.

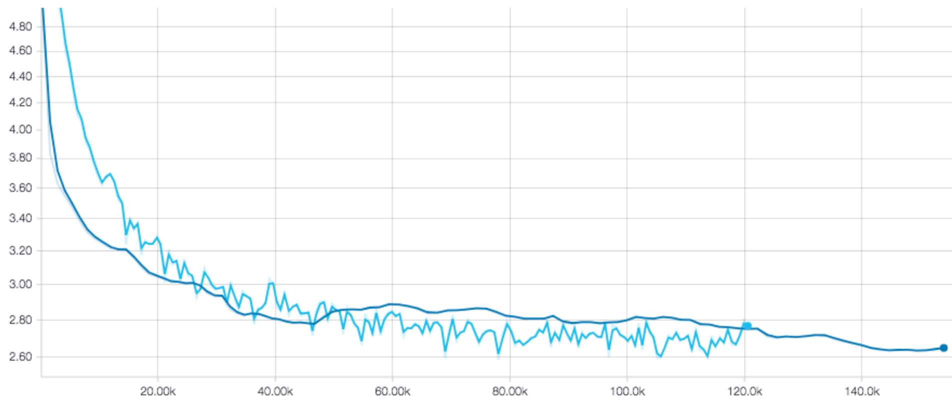


Figure 6.30: Perplexities obtained by the two best performing models. The blue model represents the best performing one with respect to curriculum learning while the cyan line identifies the best one in traditional learning. Values computed on the 3-turn evaluation set.

decreases together, until the first curriculum learning step finishes. This event causes a perturbation in the curriculum learning curves, while the cyan model continues optimizing following the same pattern.

An interesting fact to be noticed is that, after figuring out how to react to the perturbation caused by the change of dataset, the blue model arrives to a value of the performance that is closer to the cyan model as the complexity of the evaluation task increases. This happens because augmenting the complexity of the training task helps the model to perform better on more complex tasks, and for this reason during the second curriculum learning step the performance shown by the two models of Figure 6.30 is very similar.

Even if the curriculum learning approach is promising, it is clear that

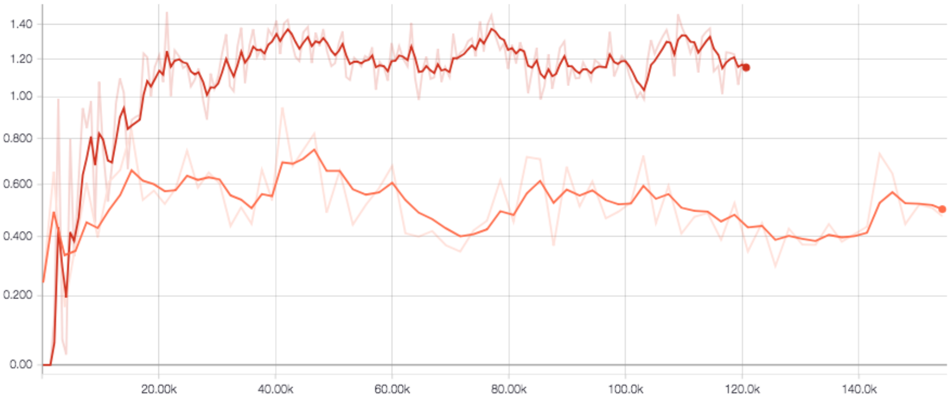


Figure 6.31: BLEU-4 scores obtained by two best performing models. The orange model represents the best performing one with respect to curriculum learning while the red line identifies the best one in traditional learning. Values computed on the 1-turn evaluation set.

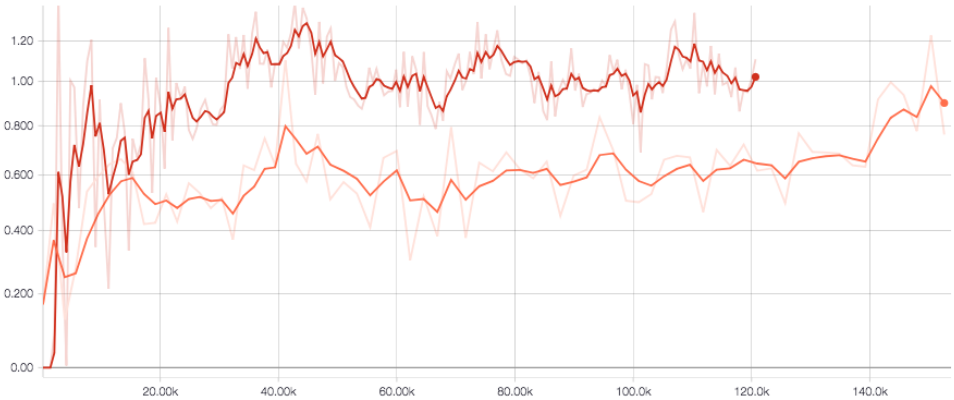


Figure 6.32: BLEU-4 scores obtained by two best performing models. The orange model represents the best performing one with respect to curriculum learning while the red line identifies the best one in traditional learning. Values computed on the 2-turn evaluation set.

a much deep study has to be performed to understand how to avoid that perturbation. For this reason, the much smoother trend shown by traditional learning makes it the optimal architecture with respect to model dependent metrics.

Figures 6.31, 6.32 and 6.33 instead present the *BLEU-4* score obtained by the same models on the three different evaluation sets. The red line is the best one in the traditional learning setting, while the orange one represents the best one using curriculum learning.

First of all, we can notice that the orange model does not show the perturbation that its showed in the plots of model dependent metrics. In addition, we can see that the traditional learning model always outperforms

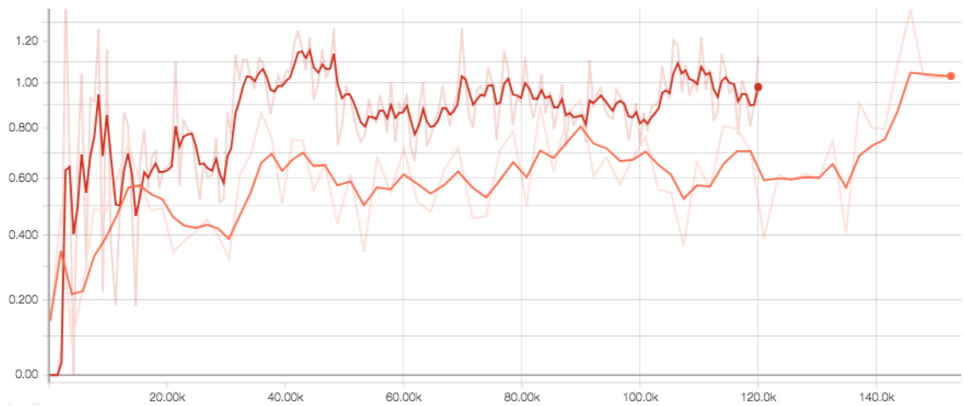


Figure 6.33: BLEU-4 obtained by two best performing models. The orange model represents the best performing one with respect to curriculum learning while the red line identifies the best one in traditional learning. Values computed on the 3-turn evaluation set.

the other one. Nicely, as the complexity of the evaluation task increases, the performance of the models becomes more similar, demonstrating the principle for which curriculum learning helps to solve complex tasks. Again, it is clear that the traditional approach is more stable and best suited for immediate use, while the curriculum learning based approach is promising but requires further study to be stabilized and used.

6.3 Qualitative analysis

As said in the last section, the best model is the one extracted from traditional learning because of its superiority in roughly both model independent and model dependent metrics and also for its more regular and smooth trends. Because of the absence of a really reliable evaluation metric, qualitative analysis is very important in Conversational AI, and for this reason this section contains a qualitative inspection of the results provided by the best model.

In this section we qualitatively analyze the quality of the answers that the best performing model generates when is fed with contexts coming from an one turn parsed evaluation set. In addition, in order to visualize what the network is learning, for each example coming from the *off-line* evaluation we show sentence level and word level attention weights. The plots of this section have to be interpreted like Figures 3.16, 3.17, 3.22, 3.23, 3.24 and 3.25, meaning that a dark colour refers to a higher attention score. In particular, each example that we show below is made of to three Figures, where each column presents the words that compose the generated answer. The Figures with orange weights, for example Figure 6.34, show the sentence attention weights, while the blue weights, Figures 6.35 and 6.36, represent

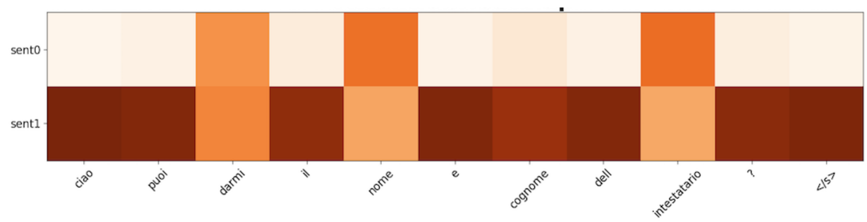


Figure 6.34: Example 1: Attention weights assigned by sentence level attention.

the word attention weights that the model assigned to the hierarchical context. In order to find the elements that compose an example, we identify a triplet, made of the sentence attention and the two word level attentions, with the unique number presented in the caption after the word *Example*.

For instance, Figures 6.34, 6.35 and 6.36 refer to the first evaluation example. Below we show off-line one turn evaluation, where the first word level attention Figure always represents the last agent utterance while the second word level attention refers to the actual user question, resulting in a basic context made of two utterances. For each column, a token in the predicted sequence, the Figures show the importance that the model assigned to each row in generating that specific predicted token. In word level attention Figures instead the row represents a word in either the user or the agent utterance, while in the sentence level attention the rows represent the importance of either the user utterance, the first row, or the agent utterance, the second row.

6.3.1 Success examples

A first example is generated with the $24k$-th model checkpoint, a bit more than one epoch, and it is interesting because it shows how the model is able to catch a first simple pattern. Figures 6.34, 6.35 and 6.36 show the attention weights assigned to the context in order to generate the answer *ciao puoi darmi il nome e cognome dell'intestatario?*.

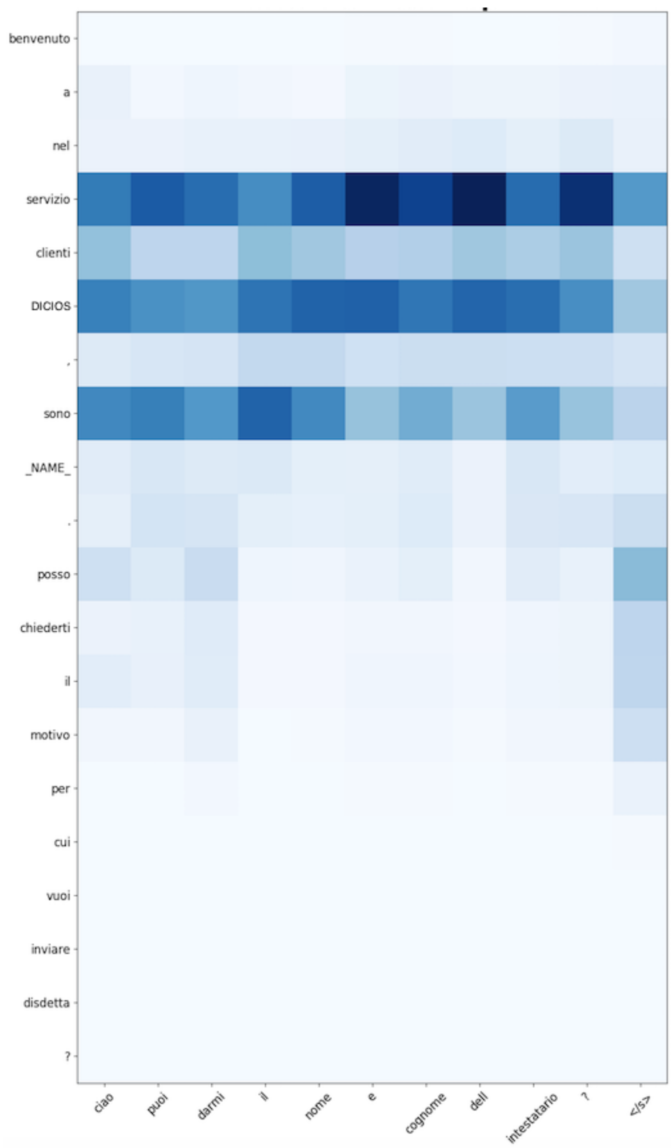


Figure 6.35: Example 1: Attention weights assigned by word level attention to the first utterance of the context.

Figure 6.36: Example 1: Attention weights assigned by word level attention to the second utterance of the context.

From Figure 6.34 we can see that the sentence level attention understands that, to generate the answer, only the user utterance, the second one in the context, represented by Figure 6.36, is useful.

Secondly, from Figure 6.35 we can notice how the model, even if the first utterance is not significant in the answer generation, catches important tokens such as *servizio*, the name of the company *DICIOS* and the verb *sono*, but not other important words such as *disdetta*.

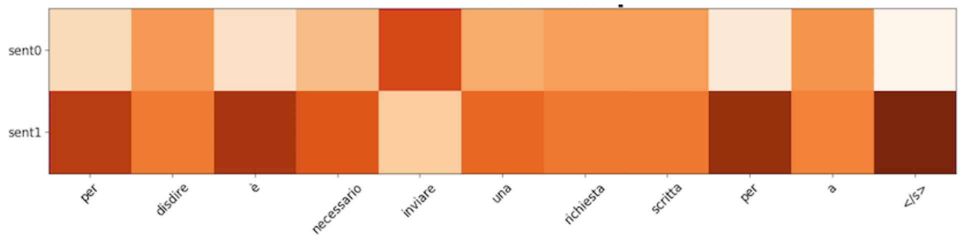


Figure 6.37: Example 2: Attention weights assigned by sentence level attention.

This can be either due to the fact that this topic has not been learnt given that the model is taken from the early training stage or because there are much hidden dynamics that can not be caught from a mere weights visualization. From Figure 6.36 we can notice how the model pays all the high level attention to the second utterance in which the single token *salve* triggers the generation of a typical question, that is asking the name and the surname of the user to load its profile.

We can noticed how, given that the predicted utterance is generated in any case by the agent after the first user utterance, only the greet *salve* is important, while the first utterance of the context is useless, and this procedure mimics what a real customer service agent does. In fact, the agent, while asking generalities, does not pay attention to the utterance he said in the previous turn.

Another interesting example comes from an inference step using the $32k$-th checkpoint and we present it in Figures 6.37, 6.38 and 6.39. This case shows two important characteristics that attention visualization presents in complex scopes such as the one of Conversational AI where the dependencies between input and output are very difficult to be understood, even by inspecting the attention weights.

In particular, in the example shown in Figures 6.37, 6.38 and 6.39 we specify how the model pays attention to the hierarchical context made of *come posso esserti utile ?* and of *avrei bisogno di sapere se volessi recedere dall'abbonamento come sono le modalita' e i tempi per non pagare penali* in order to generate the answer *per disdire e' necessario inviare una richiesta scrita per a.*

First of all we want to highlight how the generated answer, *inviare una richiesta scrita per a ... ?*, is incomplete, and this is due to the fact that the model uses weights coming from an early stage of learning. This problem was mentioned in Chapter 2 and refers to the possibility for generative models to show bad grammar, and this happens because each word in the generated sentence is the result of a maximum sampling over the vocabulary, and does not come from a database of correct answers.

With respect to the previous case, now the sentence level attention gives importance to both the utterances, in a way that is very difficult to be understood by humans.

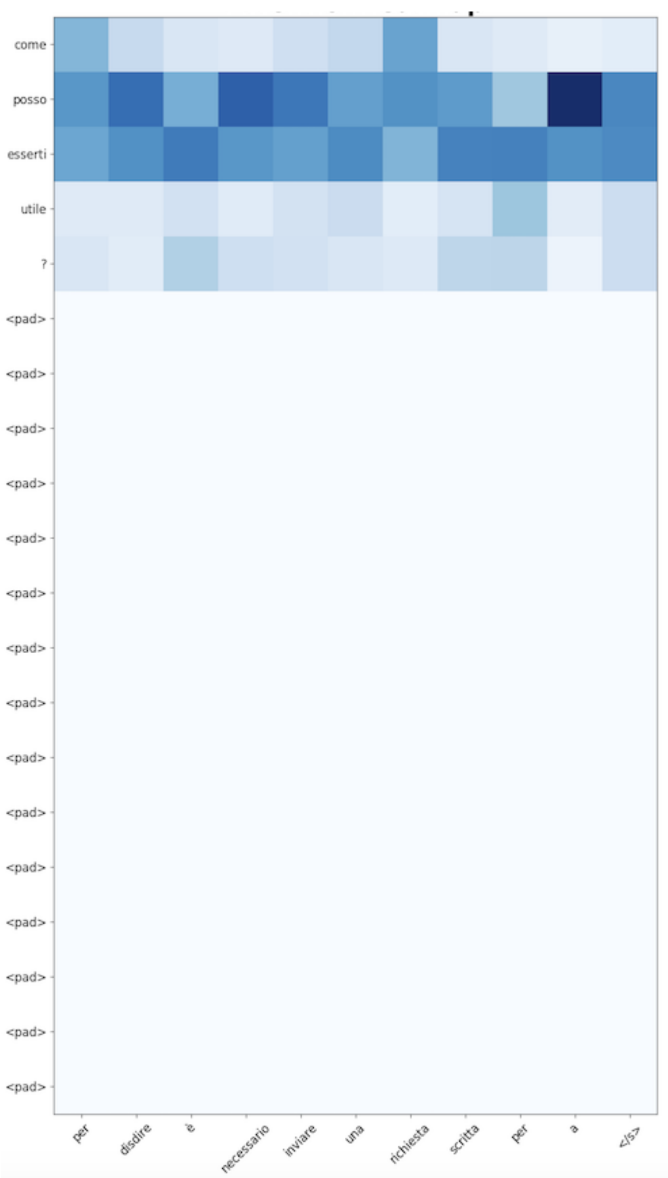


Figure 6.38: Example 2: Attention weights assigned by word level attention to the first utterance of the context.

Nicely, word level attention shown in Figure 6.39 presents a very interesting word by word relationship. In fact, to generate the word *richiesta* the model pays attention to the word *modalita'*, in fact a *request* is a *modality*, while to generate *scritta* pays a lot of attention also to *come sono*, meaning *..how is the request? written..*, in a way that the model seams to answer to a specific indirect question. This last example shows that Conversational AI is based on a various number of dependencies, some of them are *tok2tok* and other ones are *multitok2tok* or *tok2multitok*, and this last case explains

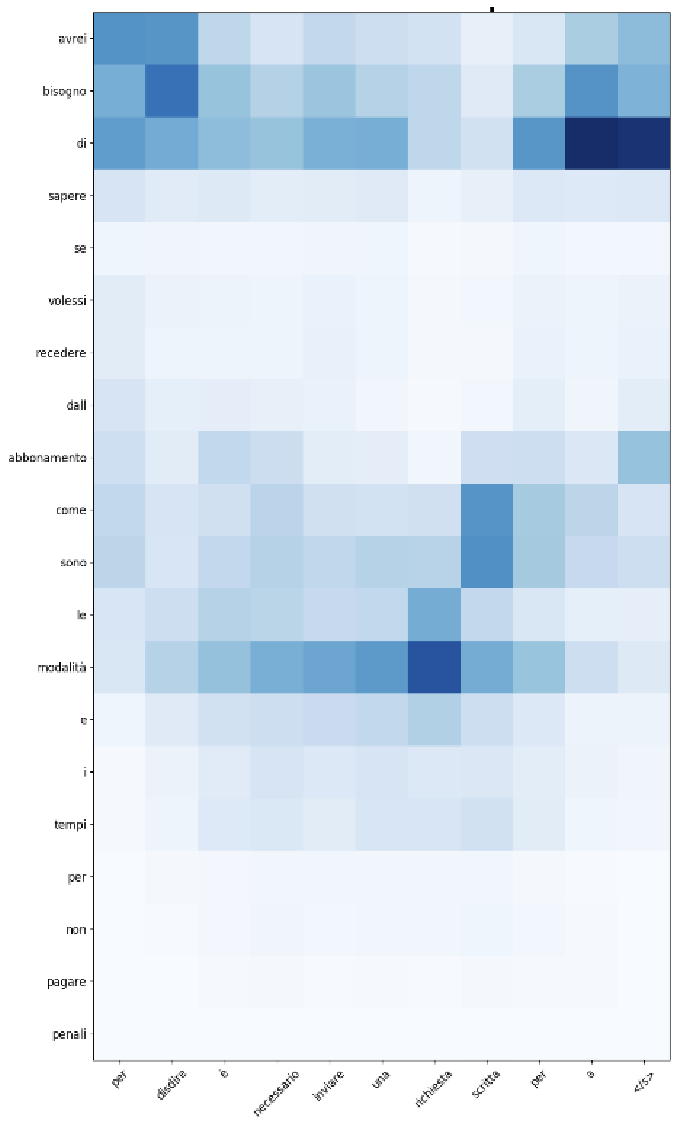


Figure 6.39: Example 2: Attention weights assigned by word level attention to the second utterance of the context.

that sometimes we can find *tok2tok* dependencies, typical of machine translation, also in this scope.

Another interesting example in this setting is the one that we present in Figures 6.40, 6.41 and 6.42. These weights are generated by a checkpoint related to the 94-th step of the model. It can be noticed how the model now is more able to extract topics, such as a service sold by the company, called *DICIOS fly*, and this can be seen by inspecting the attention weights.

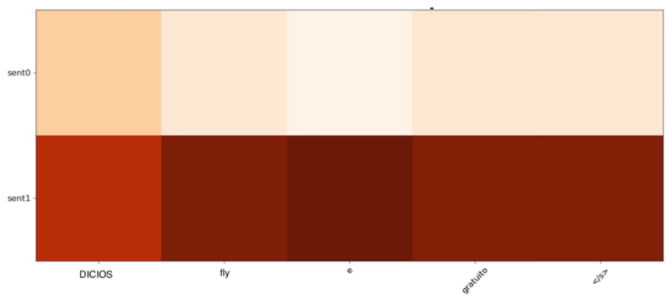


Figure 6.40: Example 3: Attention weights assigned by sentence level attention.

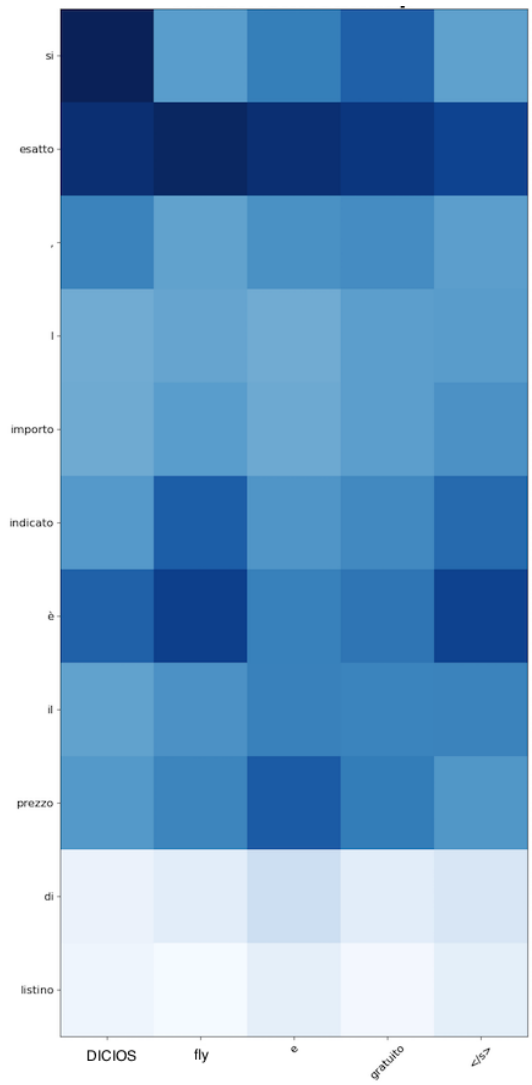


Figure 6.41: Example 3: Attention weights assigned by word level attention to the first utterance of the context.

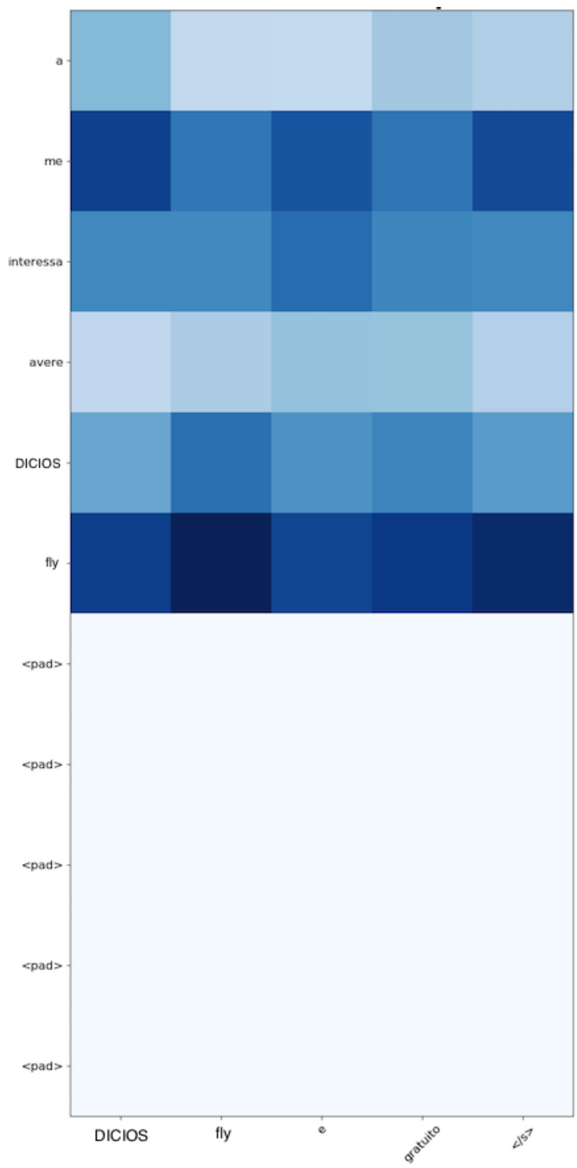


Figure 6.42: Example 3: Attention weights assigned by word level attention to the second utterance of the context.

In this case, to generate the answer *DICIOS fly e' gratuito*, the model correctly pays a lot of attention to *DICIOS fly* in the last utterance of the context and interestingly refers back to the price of the service, an information contained in the first utterance. The weird fact is that this last dependency is not clear from the visualization of the attention weights. From this example we can see how in Conversational AI the dependencies can be both direct and indirect, in a way it is not always easy to interpreted them with a mere analysis of the magnitude of the weights.

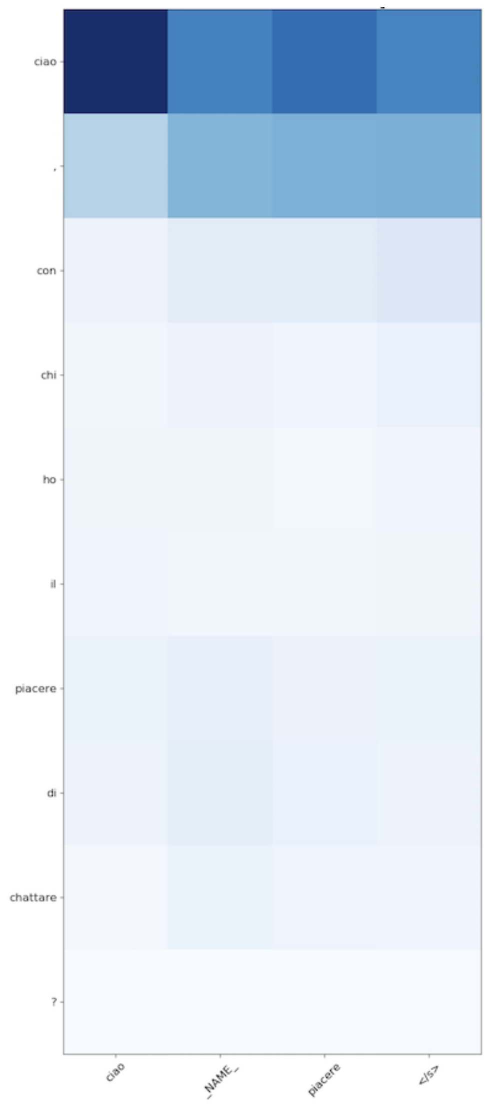


Figure 6.43: Example 4: Attention weights assigned by word level attention to the first utterance of the context.

As said in Chapter 2, generative models are able to refer back to entities in the conversational context. This is shown in Figures 6.44, 6.43 and 6.45, where the agent asks the name in the first utterance of Figure 6.43 and then, when the user replies with his name, escaped from the preprocessing chain into the support token _*NAME*_, the model, identified by the checkpoint taken after 104 steps, uses it to reply again with _*NAME*_, showing to understand this micro conversational patter.

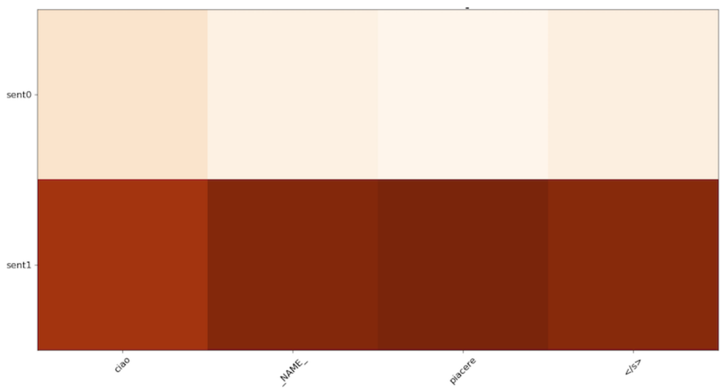


Figure 6.44: Example 4: Attention weights assigned by sentence level attention.

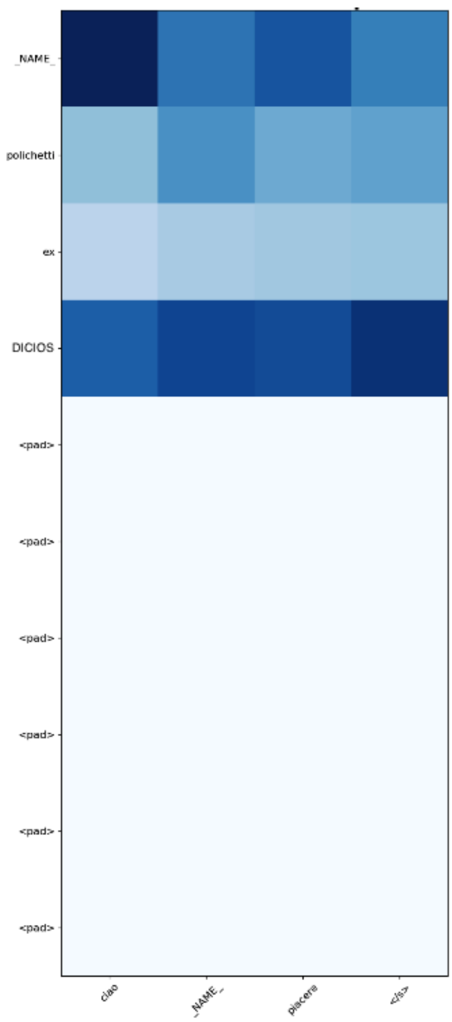


Figure 6.45: Example 4: Attention weights assigned by word level attention to the second utterance of the context.

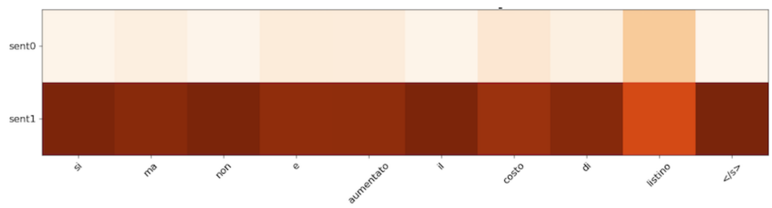


Figure 6.46: Example 5: Attention weights assigned by sentence level attention.

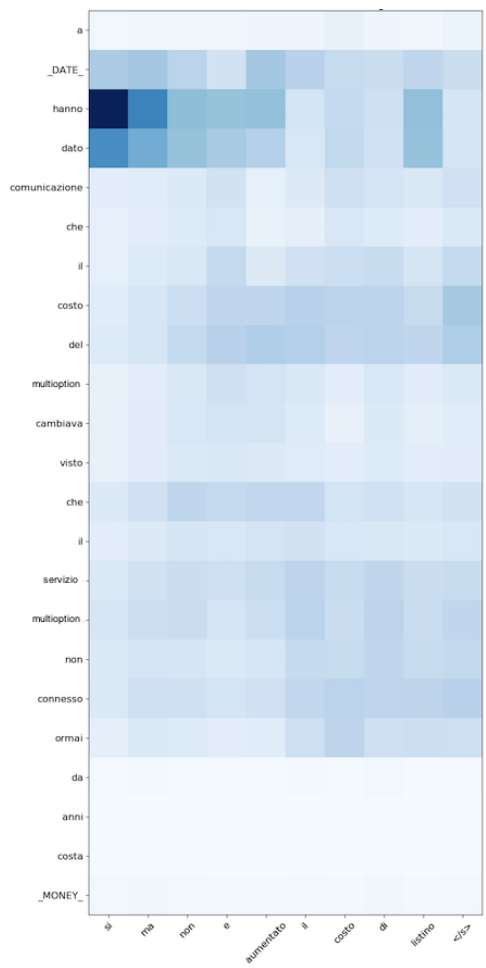


Figure 6.47: Example 5: Attention weights assigned by word level attention to the first utterance of the context.

A very interpretable situation is shown in Figures 6.46, 6.47 and 6.48. In fact Figure 6.46 shows that the first utterance is useless and Figure 6.47 describes how the weights generally are more intense for concepts.

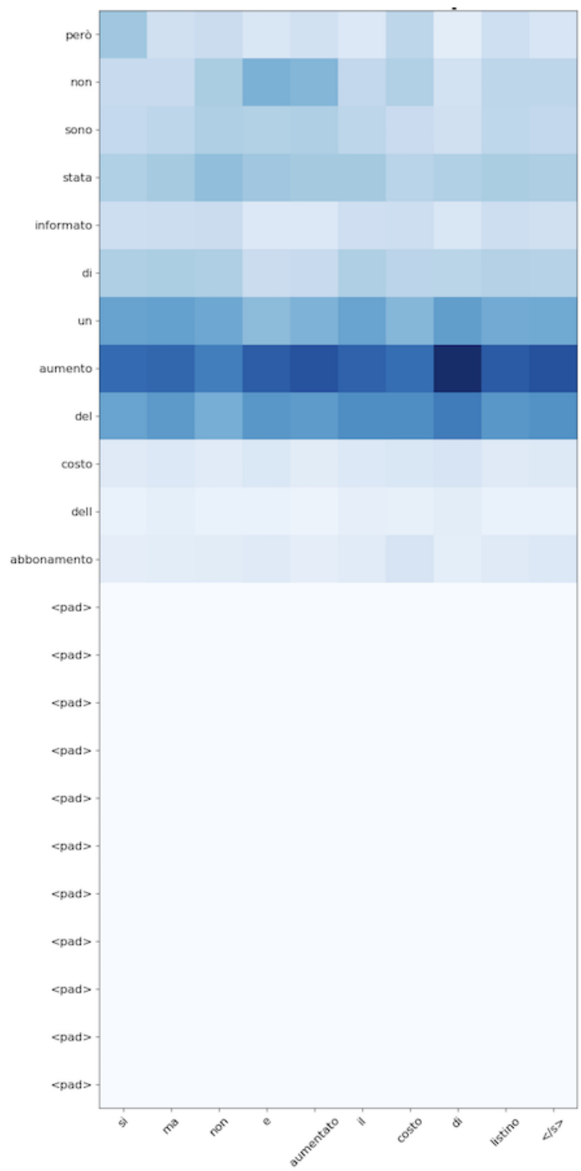


Figure 6.48: Example 5: Attention weights assigned by word level attention to the second utterance of the context.

More importantly, Figure 6.48 indicates how the most important word to generate the answer *si ma non e' aumentato il costo di listino* is *aumento*. The model uses the weights related to the 105k-th checkpoint.

Figures 6.49, 6.50 and 6.51 refer to the inference step performed with the model identified by the weights of the 123k-th checkpoint and are interesting because they show how, after being trained for a significant number of steps, the model starts to answer correctly also to indirect questions.

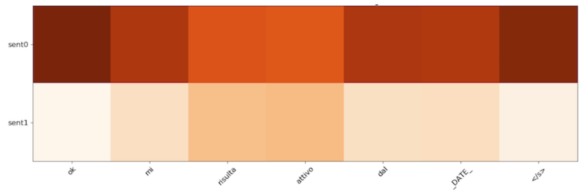


Figure 6.49: Example 6: Attention weights assigned by sentence level attention.

Figure 6.50: Example 6: Attention weights assigned by word level attention to the first utterance of the context.

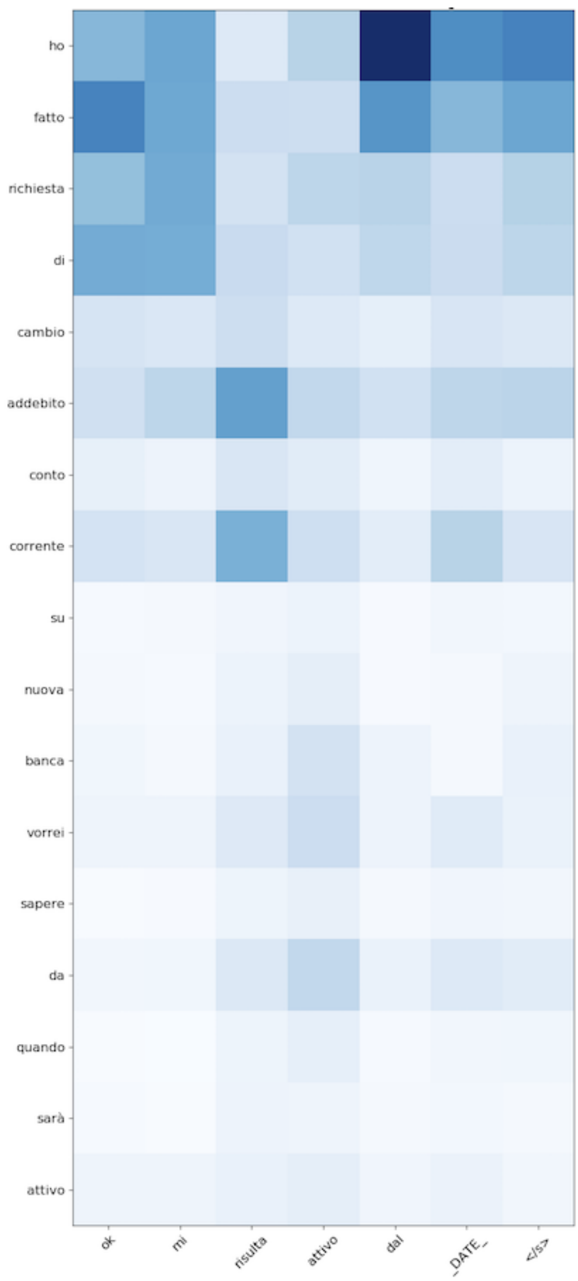


Figure 6.51: Example 6: Attention weights assigned by word level attention to the second utterance of the context.

As the training goes on, the model becomes more able to extract concepts, and to hierarchically attend on them. This is what we show in figures 6.52, 6.53 and 6.54, where we present the attention weights generated by a model using the checkpoint of the 134k-th step.

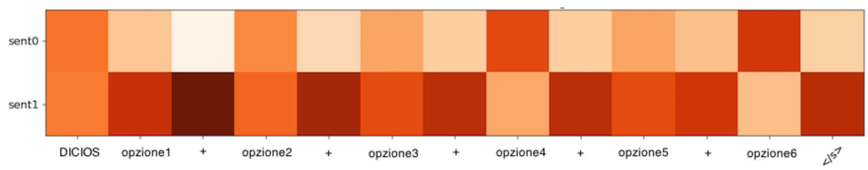


Figure 6.52: Example 7: Attention weights assigned by sentence level attention.

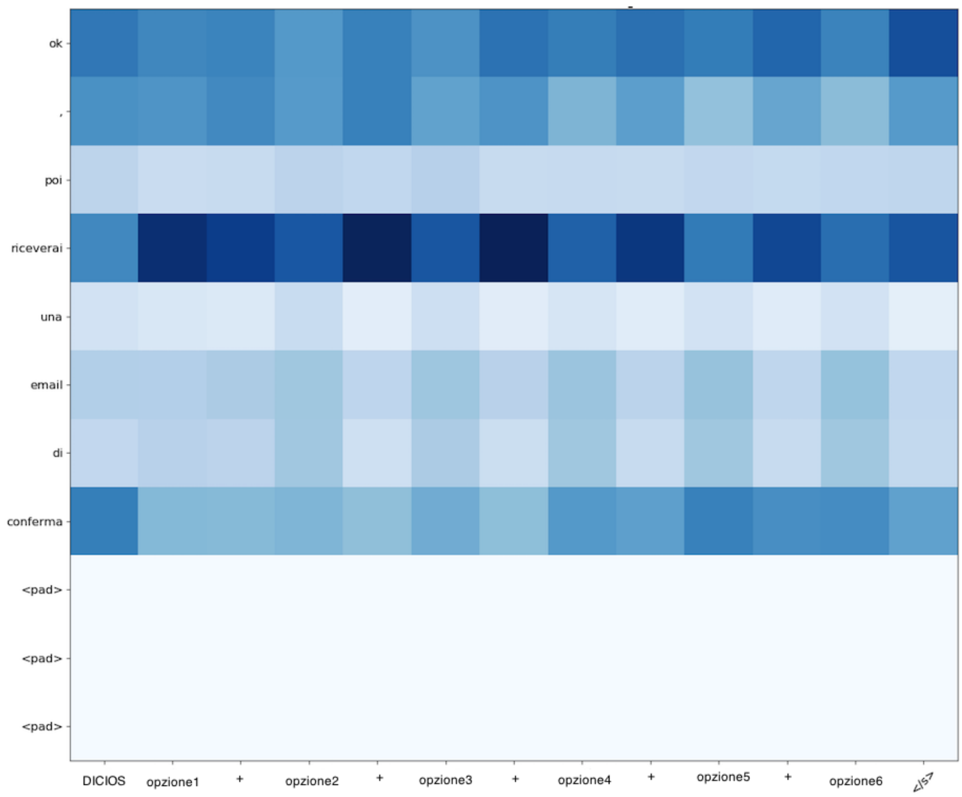


Figure 6.53: Example 7: Attention weights assigned by word level attention to the first utterance of the context.

In particular, it starts by assigning a distributed set of sentence attention weights, as shown in Figure 6.52, and the word level attention aggregates n-grams. In fact, from Figure 6.53 it extracts *ok*, *riceverai* and *conferma*. From the second utterance instead, several concepts are aggregated, such as *ottimo*, *quali*, *questo abbonamento* and *?*. Then, the model performs a very complex processing of the extracted topics and generates the correct answer. This example is important because shows how attention helps the model to hierarchically organize the available information, and to generate proper answers.

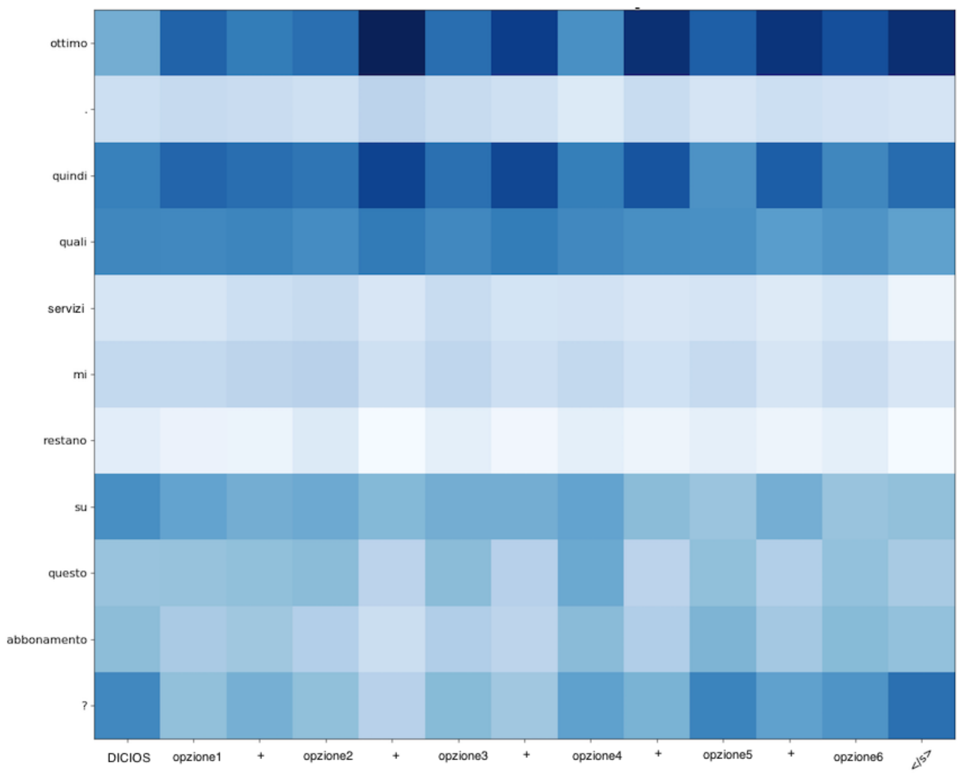


Figure 6.54: Example 7: Attention weights assigned by word level attention to the second utterance of the context.

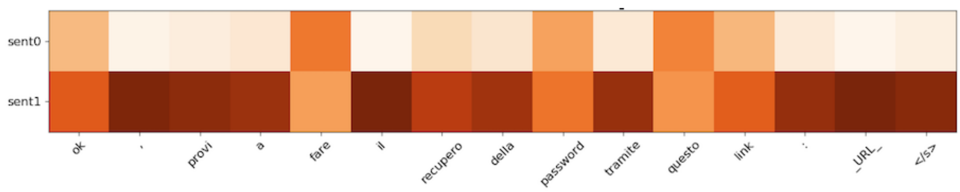


Figure 6.55: Example 8: Attention weights assigned by word level attention to the first utterance of the context.

6.3.2 Failure examples

Sometimes, the model shows some bad behaviours. An example is the one presented in Figures 6.55, 6.56 and 6.57. Nicely, the model understands that there is a problem, even without a token specifically related to this semantic meaning, and, as shown in Figure 6.57, extracts the service that causes the problem, *DICIOS fly*, and proposes a fix, that is by following the instructions provided at an external url. Also, this case shows one of the problematic issues of a model trained with one turn parsed samples.

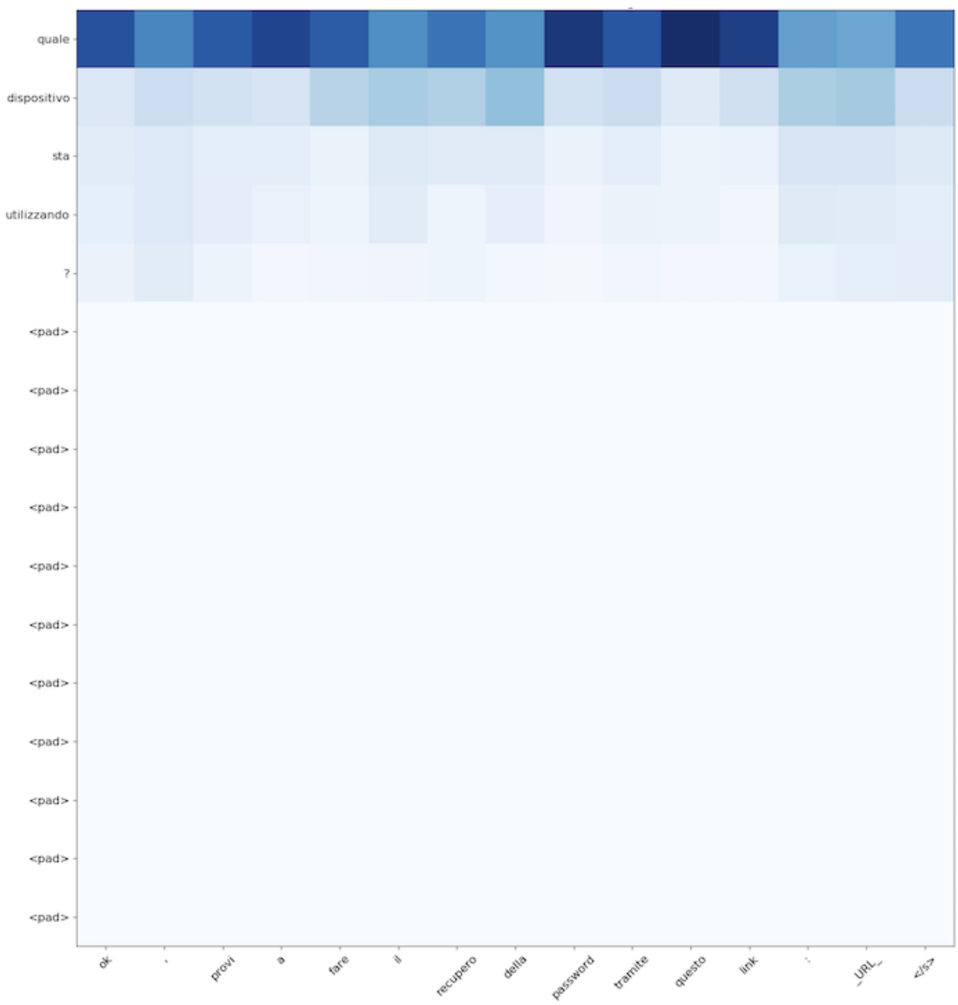


Figure 6.56: Example 8: Attention weights assigned by word level attention to the first utterance of the context.

In fact, probably, the correct solution to the problem is not to restore the password, but it is impossible for the model, and I would say also for humans, to understand the real issue without being fed with more context.

For this reason, future work would require to train this architecture with either two turns or three turns parsed sample, but, as already said, this would need both a more powerful physical setting and a training architecture that exploits also model parallelism.

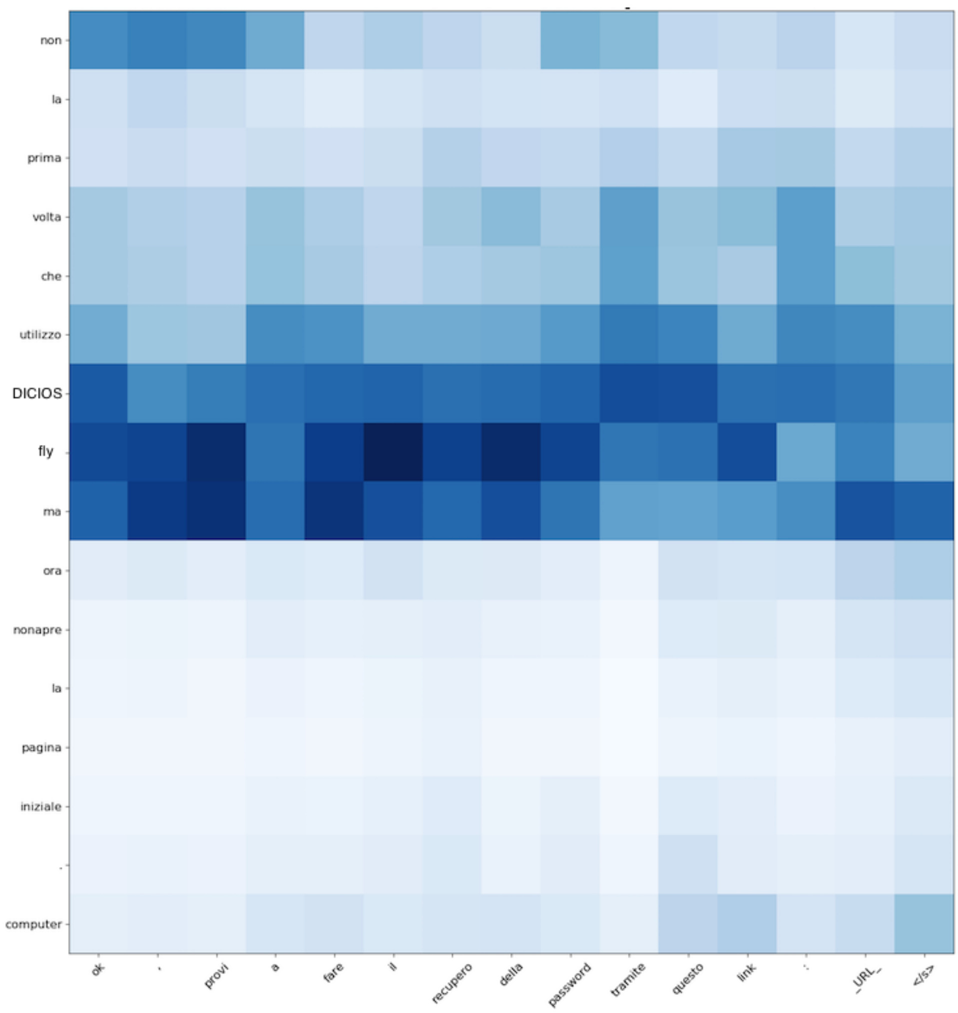


Figure 6.57: Example 8: Attention weights assigned by word level attention to the second utterance of the context.

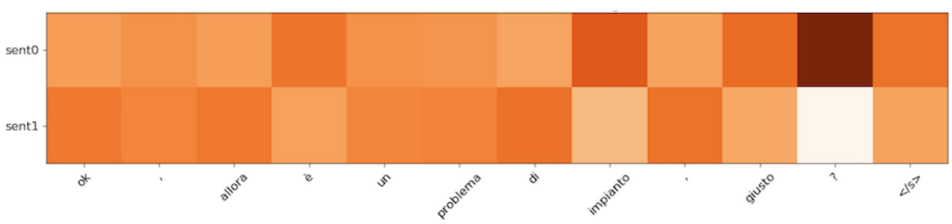


Figure 6.58: Example 9: Attention weights assigned by word level attention to the first utterance of the context.

Another similar example to is presented in 6.58, 6.59 and 6.60, that refers to an inference step performed with the model trained for 140k steps.

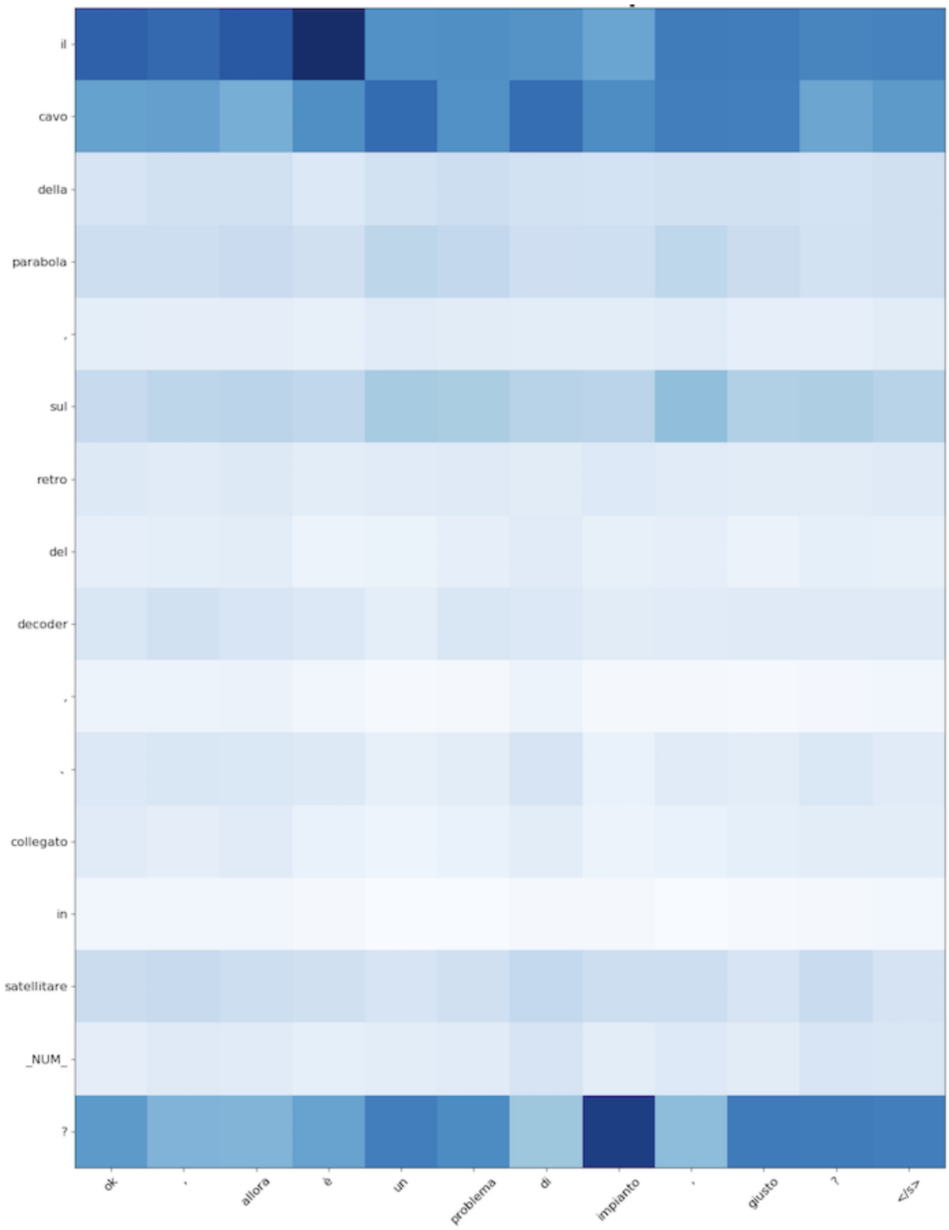


Figure 6.59: Example 8: Attention weights assigned by word level attention to the first utterance of the context.

Here the model, even if it shows the ability of extracting and understanding hierarchical concepts, such as *il cavo*, that is *the cable*, gives a too much general and maybe wrong answer, and this is probably due to the fact that is fed with a context that does not contain the information needed to answer properly. In fact, from the answer it gives we can understand that it gets that there is an hardware problem, *... problema di impianto...*, that means *a problem with the physical setting*.

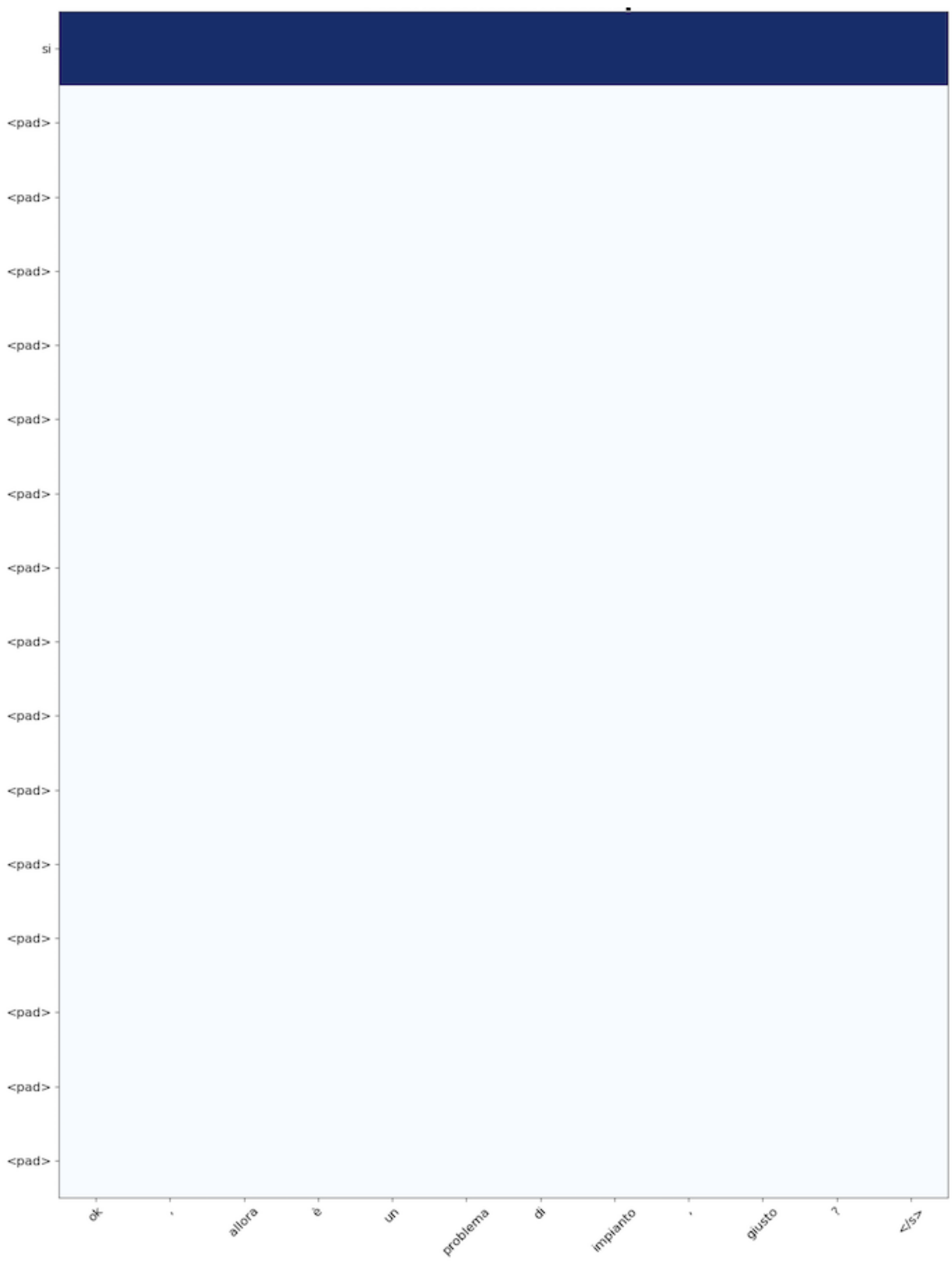


Figure 6.60: Example 8: Attention weights assigned by word level attention to the second utterance of the context.

A different problem is shown in examples 6.61, 6.62 and 6.63. In fact here the model, even if it extracts correctly the hierarchical concepts, does not understand the question of the user, and replies by asking a question that the user already answered.

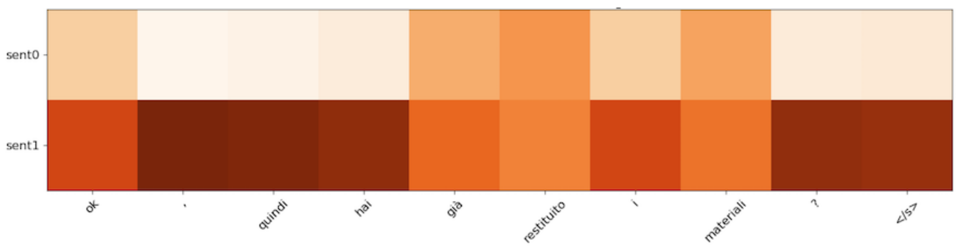


Figure 6.61: Example 10: Attention weights assigned by word level attention to the first utterance of the context.

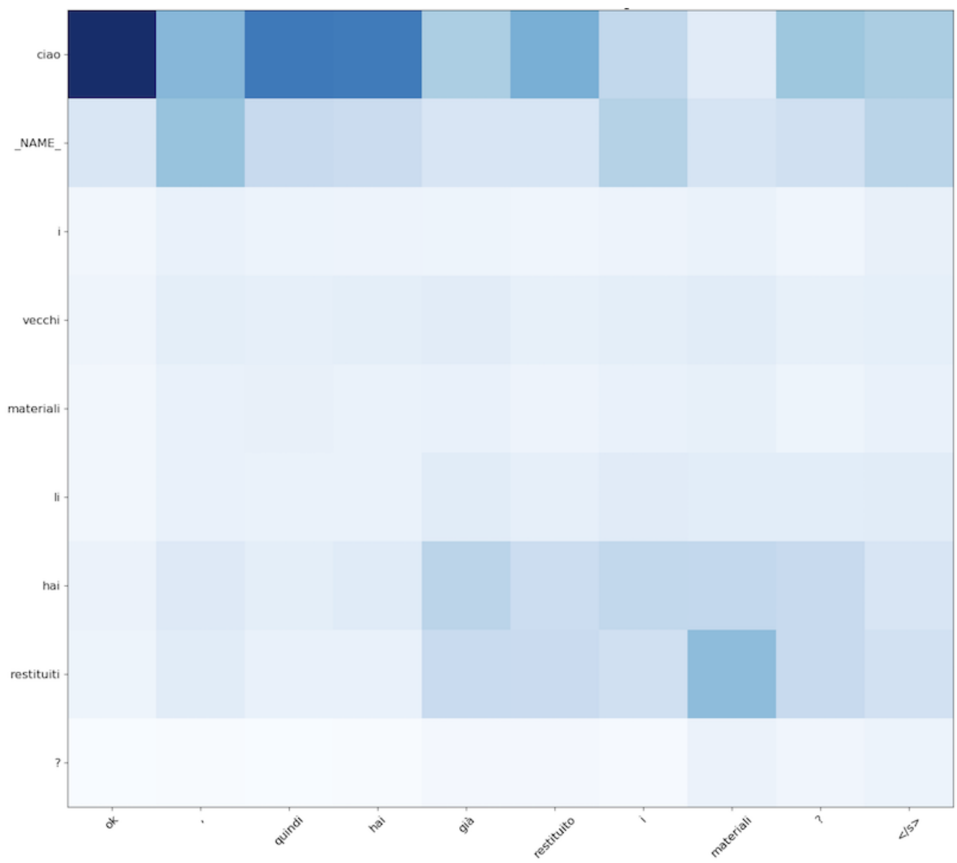


Figure 6.62: Example 10: Attention weights assigned by word level attention to the first utterance of the context.

This is a bad behaviour that can potentially drive users away, and is one of the reasons why such models are not used in production environments.

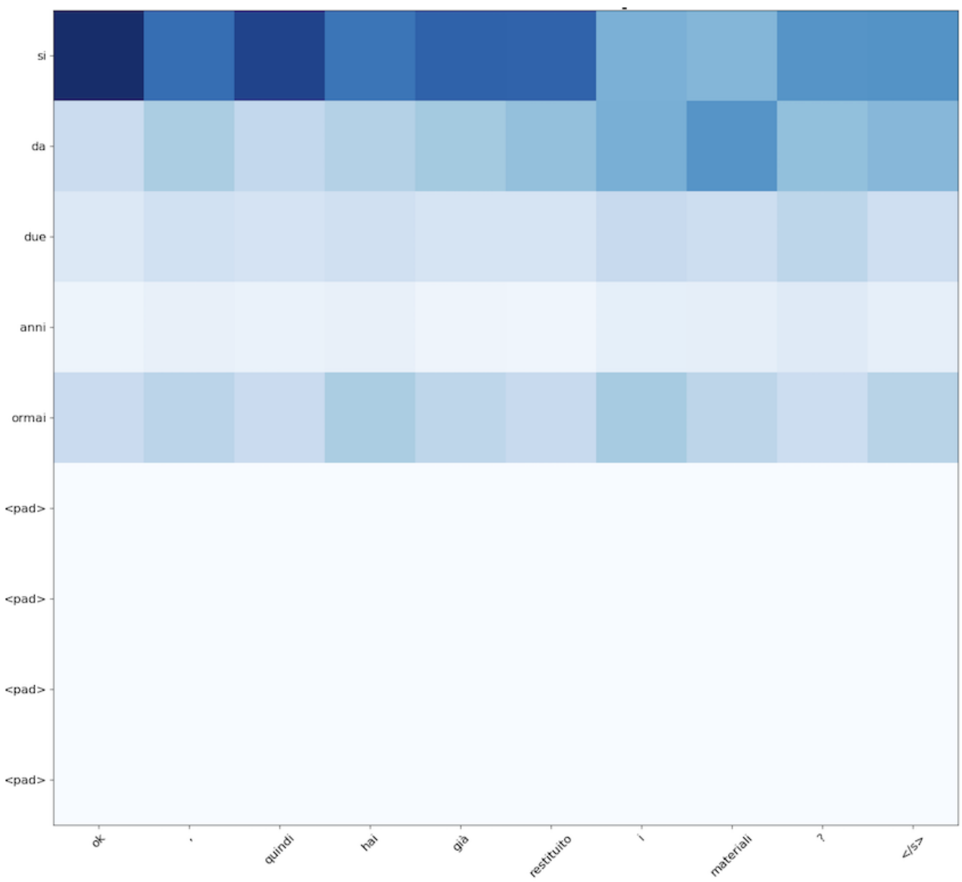


Figure 6.63: Example 10: Attention weights assigned by word level attention to the second utterance of the context.

6.3.3 Interactive chats

To perform a really meaningful qualitative analysis, it is necessary to check how the model behaves in an environment similar to the production one. For this reason, we supposed to be customers having several typical problems and we asked the model for a solution to fix them. Importantly, this inference phase has been performed by passing one turn samples to the model. A first typical problem arises when a user looses the credentials to access the on-line website. Below can be found an example of a conversation aimed to solve this issue.

buonasera mi chiamo luca : USER

BOT: *ciao _NAME_*

mi invii il link per recuperare i dati di accesso ? : USER

BOT: *_URL_*

grazie : USER

BOT: *figurati grazie*

In the example above, the user asks a very direct question, containing a lot of informative keywords, such as *link* and *dati di accesso*. For this reason, the model is able to understand immediately the necessity, resulting in a perfect handling of the request. This is not what always happens. In fact, more hard situations can arise, like the one below.

ciao sono alberto : USER

BOT: *ciao*

non riesco ad accedere al portale online : USER

BOT: *mi daresti il tuo codice client cosi facciamo una verifica*

3145151 : USER

BOT: *grazie sei ancora in linea ?*

si : USER

BOT: *si scusa messaggio automatico*

si ok : USER

BOT: *ok fatto*

mi puoi aiutare a recuperare i miei dati di accesso al portale online ? : USER

BOT: *ok, ti aiuto io, clicca qui per recuperare i tuoi dati di accesso: _URL_ sei ancora in linea ?*

grazie :) : USER

BOT: *posso esserti utile in qualcosa altro ?*

no grazie : USER

BOT: *_AGENT_SCRIPT_1_*

In this example, the user does not ask a direct question and instead explains a problem with particular words, making the task more difficult. This way the model, being a little bit fooled by the questions of the user, starts asking typical cross-context questions. Because of this intermediate micro conversational patter, the important message goes out of the context, and, after the conversational patter is finished, the model has no way to answer with the correct solution, given that description of the problem is no more in the conversational context. For this reason it responds with a cross-context answer like *ok fatto*.

Now the user decides to reply with a more direct question, receiving the correct answer. Potentially, this issue could be solved by passing a longer conversational context to the model, and this explains that, in Conversational AI, the number of utterances passed as input is very important for both training and inference.

Also, there are cases in which the model gets lost saying useless utterances. This is what happens below.

salve : USER

BOT: *come posso*

mi chiedevo se puoi aiutarmi a recuperare i miei dati di accesso al portale online : USER

BOT: *_URL_*

ahahah ok : USER

BOT: *clicca su prosegui*

si : USER

BOT: *clicca su prosegui*

ho fatto grazie : USER

BOT: *perfetto*

ciao : USER

BOT: *posso esserti*

no sono a posto grazie : USER

BOT: *ok buona serata! _AGENT_SCRIPT_1_*

In this example, it is clear that the model is able to immediately solve the issue, even if it is explained by using a non direct question, but, when the uses says *ahahah ok*, that is a very rare answer, it starts behave strangely, repeating the same utterance twice. This is caused by the fact that there is not that much data covering that specific rare situation, and for this reason the model does not know with high confidence how to handle properly this case.

Also, this example shows another typical issue of generative models. In some cases, their answers are incomplete, and this is due to the fact that the end of sequence token is assigned of a little bit higher probability than the token that allows to continue the answer. This happens because, when the algorithm is not sure about how to answer, usually generates short utterances such as *ok*. In this case, this behaviour does not result in generating an answer made of a single token but instead makes the model uncertain, with the result of raising the probability of the end of sequence token, demonstrates how these models are very difficult to be put in production. In fact, even if they have very good performances in some cases, in others they show unpredictable behaviours.

Another interesting case is to understand how the model handles situations when the user asks for the available promotions, and below is shown an example.

ei ciao : USER

BOT: *ciao sei ancora in*

si, volevo chiederti se hai delle offerte interessanti : USER

BOT: *non ho promozioni dedicate ai nuovi clienti, ma ce ne sono alcune per i gia' clienti*

ho capito ma io sono gia' cliente : USER

BOT: *puoi aderire alla promo dedicata ai nuovi abbonati estesa, ancora per pochi giorni, anche ai gia' clienti*

mi puoi inviare il link della promo ? : USER

BOT: *_URL_ puoi verificarlo direttamente da questo link _PSMILE_*

grazie : USER

BOT: *di nulla ! buona giornata _AGENT_SCRIPT_1_*

:) : USER

BOT: *_PSMILE_*

In this case we show how the architecture is able to model a very complex situation. It commits some errors, such as the early termination of the decoding, first utterance generated by the bot, and a conceptual incompatibility between the utterance where it says that there are not promotions for the new clients, and the utterance when it says that there is a promo for them.

Beside this, it is able to express a very complex concept, that is the fact that there is a promotion for new users that is available also for already registered clients, and this has a duration of only few days. Also, it is able to send the link where the promotion is presented and to nicely show the very polite and sympathetic behaviour of last utterance. Relating this topic, a case when the model does not behave perfectly is presented below.

ciao mi chiamo marcello : USER

BOT: *ciao*

mi puoi dare delle informazioni? : USER

BOT: *si certo*

mi interesserebbe sapere se ci sono delle promozioni interessanti : USER

BOT: *al momento non ho altre opzioni da proporti, mi spiace*

sicuro? : USER

BOT: *si, ma non posso darti informazioni in merito*

ok : USER

BOT: *posso esserti utile in qualcosa altro?*

si vorrei sapere quanto costa il servizo opzione4 : USER

BOT: *_MONEY_ al mese*

grazie : USER

BOT: *posso esserti utile in qualcosa altro*

no grazie sono a posto : USER

BOT: *_AGENT_SCRIPT_1_*

Here we can notice that the model, when the user asks a rare question such as *sicuro?* answers in a confuse way, showing a grammatical error. In fact *si ,ma non posso darti informazioni in merito* should be *si, non posso darti informazioni in merito.* However, it shows of being able to perfectly give information about the cost of a service. A side property is that, when a task is solved, keeps up the engagement of the user by asking if it can be useful for something else.

Another topic refers to the case when the user asks for the intervention of a technician, as shown below.

ciao sono fabio : USER

BOT: *ciao _NAME_ !*

vorrei chiederti se puoi mandarmi un tecnico in via garibaldi 23 : USER

BOT: *certamente, con chi ho il piacere di chattare ?*

sono fabio : USER

BOT: *piacere _NAME_ dimmi pure*

vorrei richiedere l'intervento di un tecnico : USER

BOT: *ok, ti mando il link per il preventivo _URL_ clicca qui: _URL_*

grazie : USER

BOT: *prego !*

Here the model still shows the problem of not having the information about the problem in the conversational context when generating the utterance *piacere _NAME_ dimmi pure.* Also, replies with a correct but not perfect answer, not setting up the meeting with the technician but sending a form to understand the cost of using such service. The last two examples show how the model behaves when it has to handle a very long conversation.

ciao mi chiamo alberto : USER

BOT: *dimmi*

mi chiedevo se ci fosse qualche promozione interessante : USER

BOT: *verifico subito attendimi in linea per favore*

ok : USER

BOT: *grazie per l'attesa. vedo che hai la visione dei pacchetti DICIOS opzione1 + opzione3*

si esatto : USER

BOT: *DICIOS opzione3 comprende documentari e life style + bambini e musica*

esatto : USER

BOT: *ok, possiamo richiedere la variazione gratuita del servizio al _DATE_, oppure, se preferisce, anticipata*

quanto mi verrebbe a costare? : USER

BOT: *_MONEY_ mensili*

grazie mi sembra ragionevole : USER

BOT: *posso esserti utile in qualcosa altro ?*

vorrei richiedere la variazione : USER

BOT: *ok, mi attendi in linea per favore?*

ok : USER

BOT: *la variazione si puo' inserire per il _DATE_ con un costo di una tantum di _MONEY_*

ok : USER

BOT: *procedo ?*

si grazie : USER

BOT: *fatto*

perfetto : USER

BOT: *posso esserle*

mi arrivera' una email di conferma? : USER

BOT: *si certo*

grazie : USER

BOT: *di nulla, posso*

no sono a posto : USER

BOT: *bene le auguro una buona serata allora*

grazie a lei : USER

BOT: *grazie*

This first example shows again how, when the context does not contain the informative utterance, the model generates not appropriate answers, and this is what happens during the generation of *ok, possiamo richiedere la variazione gratuita del pacchetto al _DATE_, oppure, se preferisce, anticipata.* In fact, this utterance is about the variation of a service as if the user had already selected the service, and this is not true.

Nicely, it is instead able to detect that anticipating the variation means to pay a penalty, while the free variation is related to a specific date. It also shows nice micro-patterns such as answering when the user asks for the price of something, the case of *_MONEY_ mensili*, making a summary of what it has already said in the conversation, this is what happens with the answer *la variazione si puo' inserire per il _DATE_ con un costo di una tantum di _MONEY_* and driving the user through a complex procedure such as the one of actuating a variation of the current plan.

In conclusion, it shows to be polite and to engage the user after a task is concluded. Still, there is the problem of the early termination of the decoding, and this is the case of *di nulla, posso*.

A second interesting long conversation is presented below.

ciao il mio nome e' carlo tarlo e vorrei chiederti delle informazioni : USER

BOT: *ciao dimmi pure*

mi sai dire quanto costa aggiungere il servizio opzione3? : USER

BOT: *_MONEY_ al mese*

interessante : USER

BOT: *si certo*

e da quando sarebbe attivo? : USER

BOT: *il _DATE_*

sto pensando di rimodulare il mio abbonamento e aggiungere il servizio opzione3 : USER

BOT: *ok allora possiamo fare la richiesta per il _DATE_ senza costi*

grazie : USER

BOT: *ok fatto posso esserti utile in qualcosa altro?*

no va bene : USER

BOT: *ok buona serata ! _AGENT_SCRIPT_1_ ciao*

ciao :) : USER

BOT: *ciao _PSMILE*

Again the model shows of being able to answer to direct questions that ask for a date, to help the user in the reformulation of its plan and to suggest the best option to perform a task, that is the answer proposing an option to perform the variation without costs.

The examples above showed how generative models are promising because of their ability of hierarchically paying attention and of successfully supporting the user in several tasks typical of a very hard and diverse environment such as customer service. Still, some problems are present, for example the one of expressing coherent personality, explained in Chapter 2, or the ability to reliably recover from not usual questions.

To solve this issue, a recent paper by Buck et al (2017) [12] proposes to integrate a module able to reformulate the question in a meaningful form, in order to help the decoder in reducing the entropy of the possible questions.

Chapter 7

Conclusions

In this thesis we want explain how to deal with a real-world *Conversational AI* problem, from data exploration to data preprocessing, normalization and standardization. After these preliminary and crucial steps, we explain how to choose, train and validate a model in a real-world setting.

In Chapter 2 we deeply discuss the relationships between text mining and Conversational AI and we analyse how to handle *long or short* conversations, *open or closed* domains and *multi or single turn* conversations. We give a particular importance to the study of providing a conversational model with *coherent personality*, *intention* and *diversity*. At the end of the Chapter we show how these models can easily result in uncontrollable behaviours, presenting the case of Tay [42], a chatbot developed by Microsoft.

The goal of Chapter 2 is to engage the reader by showing how much fascinating is *Conversational AI*. In fact, building a perfect conversational agent would mean to reach the final goal of AI, that is creating an artificial general intelligence (AGI) able to reason on concepts, have memory and take decisions. We are far away from reaching this goal and this is due to two reasons: the first one is that technologies and models have to be further improved to handle the complexity of conversations, and the second is the lack of appropriate datasets to experiment on. As a result, current models are promising but not ready to be put in production environments where mistakes are very expensive.

Chapter 3 is a review of the state of art models currently adopted to build conversational agents. Among them we consider only *retrieval* and *generative* models, the ones that employ natural language processing (NLP).

We show how retrieval models rely on a knowledge base of historical question-answer pairs and exploit two modules to return a sample from the database: the *matching measure*, that can be either deterministic or stochastic, and the *feature extraction algorithm*, that can be based on traditional NLP methods, on bag of word algorithms (BOW) or on deep learning models (DL). Among these architectures we carefully analyse *TACNTN*, a retrieval model based on CNNs and NTNs (Socher et al (2013) [91]), published by Wu et al (2016) [106], and the dual recurrent encoder, published by Lowe et al (2015) [60], that uses RNNs. Retrieval models essentially implement a semantic search over a knowledge base. In huge databases this procedure can take a lot of time, and this is the reason why we propose to integrate a clustering algorithm to reduce the time needed.

Retrieval models do not generate *novel* answers. This is the main difference with generative models, which instead exploit methods that allow to assemble new answers word by word. Given that they handle sequences of words, generative models use recurrent architectures to process them sequentially. In fact, *Conversational AI* can be seen as a task with multiple inputs, the words of the conversational context, and multiple outputs, the words of the answer, where these sequences have in general different lengths.

In Chapter 3 we explain how the above models work, starting from showing the behaviour of basic recurrent units such as *RNNs*, *LSTMs* [112], *GRUs* [4], *Layer Norm LSTMs* [3] or *BiRNNs* [4], and arriving to more complex architectures such as *Sequence to Sequence models* ([95], [19]) and *Language models* ([65]).

An important part of Chapter 3 is dedicated to the study of the attention mechanism, either *local* ([4], [19]), *global* ([19]) or *hierarchical* ([108], [110]), a powerful algorithm that empowers sequence models and gives them the possibility to pay attention on the different elements in the input sequence.

To conclude the analysis, we present a set of algorithm that perform sequence learning without recurrence. Among them the most interesting is the one of Vaswani et al (2017) [100], an architecture able to process sequences with an architecture that dispenses from recurrent modules and that is solely based on *self attention*. The last two Sections of Chapter 3 are summaries that explain how to prepare data and train Sequence models. We spent the most of Chapter 3 studying generative models for Conversational AI because we believe that they are intuitive, fascinating and interesting because of their ability of generating novel answers.

In Chapter 4 we frame our case study and we show how we designed our preprocessing pipeline. This thesis has both the goal of investigating the most interesting methods used to build conversational models and of showing their application in a real-world use case. In fact, we want to describe generative conversational models both from a theoretical and a practical point of view, taking as an example a technical case study coming from a project developed together with *Loop AI Labs*, an American company together which we developed this thesis. This project had the goal of inspecting the performance and applicability of generative conversational models in a real world scenario, and the support of the team of *Loop AI Labs* was fundamental during the design of the model.

Thanks to *Loop AI Labs*, we could have access to a big knowledge base containing historical set of conversations. This database came from *DI-*

CIOS, an Italian service provider company, and it is a set of the anonymized recorded conversations produced by a chatting center in the north of Italy between *January 2016* and *December 2016*.

This dataset was dirty, irregular and related to a lot of topics. In order to help the job of the model we designed a preprocessing pipeline to standardize and normalize the conversations. The result of this pipeline is the final training set.

We started from data exploration, where we understood the properties of the dataset we had to handle. Then, we designed a procedure for word and character level cleaning. We empowered the preprocessing pipeline with an interesting module: a statistical spell checker able both to detect if a word contains a typo and to correct the misspelled word with its most plausible correction in a vocabulary. This algorithm is based on the number of occurrences that each word has in the corpus, and works in such a way that the percentage of the generated false positive is kept low, at the cost of allowing few false negatives. This means that the program corrects a word only if it is very sure that it is misspelling, resulting in a safe spell correction module that does not introduce errors. As a third step, we propose a module able to extract long and frequent sequences and we assign them a single token. An important contribution of this thesis is the design of a preprocessing pipeline to normalize Italian text. At the end of the Chapter, we propose a critical analysis where we show how to parse historical conversations into multi turn training samples.

Chapter 5 presents the features of our hardware and the programming environment that we decided to adopt. We explain why we chose a hierarchical attention based model and we detail the set of hyper-parameters that we decided to use.

In the second part of Chapter 5 we describe the two learning paradigms that we quantitatively compare in Chapter 6: a first one, called *curriculum learning* and based on the work of Bengio et al (2009) [6], and a second called *traditional learning*, an approach based on the learning rate decay proposed by Vaswani et al (2017) [100].

Curriculum learning takes inspiration from a peculiar aspect of conversations. In fact, while the hierarchical structure of dialogues is a property often used to bias the model architecture, and an example is the algorithm presented by Xing et al (2017) [108], another property that is usually not exploited is that conversations can be seen as incremental in their complexity. In this scope, generative answering can be seen as a task where the complexity grows as the provided context gets longer, in fact answering to one turn contextualized question is easier then doing it for two turns, and we adopted curriculum learning because it is particularly suited to exploit this property. In fact, this paradigm is usually chosen for problems where the agent starts from simple tasks and abstracts his knowledge to solve more complex tasks. Specifically, a Conversational AI model trained with curriculum learning performs some epochs with one turn samples, then it changes dataset and uses two turns data for some other epochs, and goes on increasing the complexity of the training samples.

Traditional learning instead consists in training a generative conversational model only with data parsed in a specific number of turns, kept constant during training. Here stays the first difference with *curriculum learning*. In fact, while the core feature of the curriculum based approach explained in Section 5.2.1 is that the model is trained with a set of datasets with an increasing complexity, this second more traditional approach proposes to train the model always with a set of samples parsed in a specific number of turns. We call this second approach *traditional* because it is a combination of the paradigm with which the community usually trains generative conversational models and of some intuitions provided by Vaswani et al 2017 [100]. In fact, we decided to adopt the learning rate decay that Vaswani et al 2017 [100] used to train the *Transformer*.

A contribution of this thesis is the idea of combining the traditional approach used to train generative conversational models with the learning rate decay proposed by Vaswani et al 2017, [100].

In the last part of Chapter 5 we study the different types of training procedures, sequential or distributed, focusing on the second family and showing the differences between asynchronous and synchronous distributed training. In our experiments we decided to use asynchronous distributed training.

At the end of Chapter 5 we present a Section where we analyse the evaluation metrics that are commonly adopted, *word overlap* or *word embedding based*, showing that they are not correlated with human judgement. We conclude by saying that Conversational AI needs a new, more reliable and correlated evaluation metric.

As a unique result, this thesis presents an explicit quantitative analysis of the BLEU-4 scores, filling a huge gap that research previously had. In fact, to the best of our knowledge, it does not exist a report showing quantitative results of generative models.

However, it is clear that the current evaluation metrics does not provide a signal able to correctly discriminate models. For this reason it is important for the community to head in the direction of designing a new evaluation metric correlated with human judgement. Another problematic issue of using *BLEU* to evaluate Conversational AI is to understand what is a good value for it. It is known that the *Transformer*, the very best model for machine translation, obtains 28 *BLEU-4* points, but its goal is to model not entropic data, and this value cannot be used as an indicator for Conversational AI, first of all because it uses a different architecture, and lastly because of the motivations explained in Liu et al (2016) [59].

In their paper of 2017 Liu et al [59] noticed that generating dialogue responses conditioned on the conversational context is in fact a more difficult problem, and this is due to the fact that the set of possible correct answers to a context is very large. The main point here is that dialogue response generation given solely the context has intuitively a higher entropy than translation given text in source language. To fix the above issues, which make evaluation pretty complex, research is heading in the direction of building an embedding based metric to more reliably evaluate Conversational AI models. This way, a better signal would be provided to discriminate models during evaluation. An example where word-overlap based metrics fail is presented in following dialogue.

The agent says:

mi puoi dare il tuo indirizzo email ?

The user answers:

va bene aspettami in linea

The ground truth response is:

ok

The model generates:

si

Every word overlap metric, Section 5.4.1, would assign a score equal to zero to the predicted response, while an embedding metric, Section 5.4.2, would find out that *ok* and *si* are similar answers. In fact, embedding based metrics will assign a very high similarity score to the predicted answer, resulting in a cleaner evaluation signal, and this is the reason why we believe that word embedding metrics are best suited for building reliable evaluation metrics correlated with human judgement.

In Chapter 6 we present the obtained results. We select a hierarchical attention sequence model similar to the one presented by Xing et al (2017) [108] and we test its behaviour on the real world dataset generated by the preprocessing procedure described in Chapter 4.

As a first part we use model dependent and independent metrics, such as perplexity [101], BLEU [71], ROUGE [56] and accuracy, to *quantitative* cross-validate the recurrent unit of a set of curriculum learning based models. We design four algorithms and we show that the LSTM [112] based architecture performs better.

In a second part we consider traditional learning and we quantitatively analyse seven models finding out that the best architecture uses layer normalized LSTM [3] cells, does not reverse the input sequence, uses $0,15$ as dropout probability, does not pre-train the word embeddings and uses 256 as batch size. From the results of Chapter 6 it is clear that using 256 as the batch size gives an important performance improvement. We showed that, in order to do increase the batch size, it is necessary to either use *P100 GPUs*, or to adopt an approach that allows to use both *data* and *model* parallelism.

Curriculum learning is more difficult to be evaluated because it has a lot of variables to be tuned, that are the number of curriculum steps, the number of epochs per step and the parameters of the learning rate decay. Traditional learning instead is easier to be validated, and this happens because the training procedure is more stable. For the above reasons, in Chapter 6 we claim that traditional learning is more reliable for immediate usage. However, we think that, if further validated, curriculum learning is promising.

One of the contributions of this thesis is the idea of designing such a learning paradigm that takes into consideration the incremental complexity of conversations to define a procedure that could help the model to generalize better. Also, we believe of having opened a new branch of research that, if further deepened, could potentially give important improvements in the design of generative models.

An important discovery we made is understanding that, even if generative models for Conversational AI use the same architectures of machine translation, design choices made in this last field are not always portable to Conversational AI. The reason why this happen is that, if in machine translation a token usually maps to a specific single token in the translated sequence, in Conversational AI happens that each word of the generated response is dependent on the big parts of the input sequence. This can be seen from several examples presented in the off-line evaluation of Chapter 6. On the other hand, some other examples show that the model, when it is necessary, is able to learn the ability of focusing on a single word in the context, demonstrating how Conversational AI can be seen as a generalization of machine translation.

As the last step we take the best models coming from the two learning paradigms and we perform a quantitative comparison from which we understand that traditional learning is more reliable. This is the reason why we choose this last learning paradigm to perform a qualitative analysis where we inspect the responses that the model proposes in *off-line* and *on-line* evaluation. If we perform the off-line evaluation by taking some one turn parsed evaluation examples coming from a separate test set, on the other hand we perform on-line testing by analysing how the model behaves during chats with real users.

Qualitative analysis shows that the number of turns is crucial while developing such kinds of models. In fact, some evaluation examples show that, when the information needed to answer a question is not present in the context, it is impossible for the model to generate a proper response.

Beside some annoying issues, such as the fact that the decoding phase often terminates too early, resulting in an not natural truncation of the answer generation process, or the unpredictable behaviour that these conversational agent show when they are asked to answer to a rare question, conversational models are usually able to generate correct and satisfying answers, showing politeness and engaging the user during the conversation. Unfortunately, because of few cases in which they answer in an unpredictable way, we believe that for now it is not safe to use generative architectures in production environments.

7.1 Future work

As said in Chapter 3, the problem of modelling situations where both the input and the output are sequences of variable size is often faced by adopting sequence to sequence models, first introduced by Cho et al (2014) [19] and by Sutskever et al (2014) [95]. Their most intuitive application is machine translation, where an encoder reads a sequence in English and a decoder generates the target sentence in French, while being conditioned on the output of the encoder. Beside this basic application, these models, because of their extreme flexibility, can be used for a vast variety of tasks.

In this scope, we claim that each task that adopts these models is different in terms of the entropy it shows, intended as the number of target sequences that are appropriated for a given input sequence. In fact, while in machine translation this number is bounded, in other scopes such as Conversational AI it is huge and usually infinite. In fact, if we consider a conversational context made of *Hello*, coming from agent 1, then *Hi* from agent 2 and then *How are you ?* again from agent 1, both *I am good* and *I do not feel good* belong to the infinite set of sentences that are appropriate answers to the proposed conversational context.

We state the problem this way: sequence architectures are not able to model this diversity because they optimize cross entropy. The assertion is that this optimization procedure forces too much the model to refer to the specific ground truth answer. In fact, all the other acceptable candidates are not considered, resulting in a weak model.

In the frame described above, a second assumption is that the encoder-decoder architecture is valuable, especially in the variant that uses the hierarchical encoder, proposed by Xing et al (2017) [108]. For this reason, it is reasonable to think that sequence models, if trained in a different way, could both generate more meaningful and diverse answers and result a truly generative models. Following the above intuitions, that is saying that the optimization is conceptually not correct for all the applications while the architecture itself is valuable, as said by Vinyals et al (2015) [101], we believe that the future of Conversational AI is to introduce a new loss function able to better train sequence models for high entropic data, creating truly generative sequence to sequence architectures.

Bibliography

[1] Apple. Siri. `https://www.apple.com/ios/siri/`.

[2] AWS. Amazon chatbots. `https://aws.amazon.com/lex/`.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

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

[5] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

[6] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[7] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[8] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[9] Ondrej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Johannes Leveling, Christof Monz, Pavel Pecina, Matt Post, Herve Saint-Amand, et al. Findings of the 2014 workshop on statistical machine translation. In *Proceedings of the ninth workshop on statistical machine translation*, pages 12–58, 2014.

[10] Botengine. Hire your slack chatbot in a few simple steps. `https://www.botengine.ai/integrations/slack`.

[11] Léon Bottou. From machine learning to machine reasoning. *Machine learning*, 94(2):133–149, 2014.

[12] Christian Buck, Jannis Bulian, Massimiliano Ciaramita, Wojciech Gajewski, Andrea Gesmundo, Neil Houlsby, and Wei Wang. Ask the right questions: Active question reformulation with reinforcement learning. *arXiv preprint arXiv:1705.07830*, 2017.

[13] Aoife Cahill. Correlating human and automatic evaluation of a german surface realiser. In *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*, pages 97–100. Association for Computational Linguistics, 2009.

[14] Chris Callison-Burch, Philipp Koehn, Christof Monz, Kay Peterson, Mark Przybocki, and Omar F Zaidan. Findings of the 2010 joint workshop on statistical machine translation and metrics for machine translation. In *Proceedings of the Joint Fifth Workshop on Statistical Machine Translation and MetricsMATR*, pages 17–53. Association for Computational Linguistics, 2010.

[15] Chris Callison-Burch, Philipp Koehn, Christof Monz, and Omar F Zaidan. Findings of the 2011 workshop on statistical machine translation. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 22–64. Association for Computational Linguistics, 2011.

[16] Chris Callison-Burch, Miles Osborne, and Philipp Koehn. Re-evaluation the role of bleu in machine translation research. In *11th Conference of the European Chapter of the Association for Computational Linguistics*, 2006.

[17] Boxing Chen and Colin Cherry. A systematic comparison of smoothing techniques for sentence-level bleu. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 362–367, 2014.

[18] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*, 2016.

[19] 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.

[20] CNET. Facebook m chatbot. `https://www.cnet.com/news/facebook-is-killing-m-its-personal-chatbot-assistant/`.

[21] Christopher Colah. Understading lstms. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[22] Jasmine Collins, Jascha Sohl-Dickstein, and David Sussillo. Capacity and trainability in recurrent neural networks. *stat*, 1050:28, 2017.

[23] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[24] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[25] Misha Denil, Loris Bazzani, Hugo Larochelle, and Nando de Freitas. Learning where to attend with deep architectures for image tracking. *Neural computation*, 24(8):2151–2184, 2012.

[26] Dominic Espinosa, Rajakrishnan Rajkumar, Michael White, and Shoshana Berleant. Further meta-evaluation of broad-coverage surface realization. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 564–574. Association for Computational Linguistics, 2010.

[27] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, pages 226–231, 1996.

[28] Peter W Foltz, Walter Kintsch, and Thomas K Landauer. The measurement of textual coherence with latent semantic analysis. *Discourse processes*, 25(2-3):285–307, 1998.

[29] Gabriel Forgues, Joelle Pineau, Jean-Marie Larchevêque, and Réal Tremblay. Bootstrapping dialog systems with word embeddings. In *Nips, modern machine learning and natural language processing workshop*, volume 2, 2014.

[30] Michel Galley, Chris Brockett, Alessandro Sordoni, Yangfeng Ji, Michael Auli, Chris Quirk, Margaret Mitchell, Jianfeng Gao, and Bill Dolan. deltableu: A discriminative metric for generation tasks with intrinsically diverse targets. *arXiv preprint arXiv:1506.06863*, 2015.

[31] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. *arXiv preprint arXiv:1705.03122*, 2017.

[32] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[33] Yvette Graham, Timothy Baldwin, and Nitika Mathur. Accurate evaluation of segment-level machine translation metrics. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1183–1191, 2015.

[34] Jiatao Gu, James Bradbury, Caiming Xiong, Victor OK Li, and Richard Socher. Non-autoregressive neural machine translation. *arXiv preprint arXiv:1711.02281*, 2017.

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[36] 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.

[37] Joeri Hermans. Distributed deep learning - part 1 - an introduction. `http://joerihermans.com/ramblings/distributed-deep-learning-part-1-an-introduction/`.

[38] Geoffrey Hinton. Rms prop. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture\_slides\_lec6.pdf`.

[39] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

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

[41] horovod. Horovod. `https://github.com/uber/horovod`.

[42] Business Insider. Microsoft deletes racist genocidal tweets from ai chatbot tay. `http://www.businessinsider.com`.

[43] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[44] Lukasz Kaiser, Aidan N Gomez, Noam Shazeer, Ashish Vaswani, Niki Parmar, Llion Jones, and Jakob Uszkoreit. One model to learn them all. *arXiv preprint arXiv:1706.05137*, 2017.

[45] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

[46] Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016.

[47] Anjuli Kannan, Karol Kurach, Sujith Ravi, Tobias Kaufmann, Andrew Tomkins, Balint Miklos, Greg Corrado, László Lukács, Marina Ganea, Peter Young, et al. Smart reply: Automated response suggestion for email. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 955–964. ACM, 2016.

[48] Andrey Karpathy. Rnn effectiveness. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`.

[49] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[50] Ryan Kiros, Yukun Zhu, Ruslan R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-thought vectors. In *Advances in neural information processing systems*, pages 3294–3302, 2015.

[51] Thomas K Landauer and Susan T Dumais. A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological review*, 104(2):211, 1997.

[52] Hugo Larochelle and Geoffrey E Hinton. Learning to combine foveal glimpses with a third-order boltzmann machine. In *Advances in neural information processing systems*, pages 1243–1251, 2010.

[53] Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and Bill Dolan. A diversity-promoting objective function for neural conversation models. *arXiv preprint arXiv:1510.03055*, 2015.

[54] Jiwei Li, Michel Galley, Chris Brockett, Georgios P Spithourakis, Jianfeng Gao, and Bill Dolan. A persona-based neural conversation model. *arXiv preprint arXiv:1603.06155*, 2016.

[55] Jiwei Li, Will Monroe, Tianlin Shi, Sébastien Jean, Alan Ritter, and Dan Jurafsky. Adversarial learning for neural dialogue generation. *arXiv preprint arXiv:1701.06547*, 2017.

[56] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. *Text Summarization Branches Out*, 2004.

[57] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130*, 2017.

[58] Pocket lint. Google assistant. `https://www.pocket-lint.com/apps/news/google/`.

[59] Chia-Wei Liu, Ryan Lowe, Iulian V Serban, Michael Noseworthy, Laurent Charlin, and Joelle Pineau. How not to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation. *arXiv preprint arXiv:1603.08023*, 2016.

[60] Ryan Lowe, Nissan Pow, Iulian Serban, and Joelle Pineau. The ubuntu dialogue corpus: A large dataset for research in unstructured multi-turn dialogue systems. *arXiv preprint arXiv:1506.08909*, 2015.

[61] Ryan Lowe, Nissan Pow, Iulian Serban, and Joelle Pineau. The ubuntu dialogue corpus: A large dataset for research in unstructured multi-turn dialogue systems. *CoRR*, abs/1506.08909, 2015.

[62] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[63] Chatbots magazine. Wechat chatbots. `https://chatbotsmagazine.com/china-wechat-and-the-origins-of-chatbots-89c481f15a44`.

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

[65] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[66] Jeff Mitchell and Mirella Lapata. Vector-based models of semantic composition. *proceedings of ACL-08: HLT*, pages 236–244, 2008.

[67] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré. Asynchrony begets momentum, with an application to deep learning. In *Communication, Control, and Computing (Allerton), 2016 54th Annual Allerton Conference on*, pages 997–1004. IEEE, 2016.

[68] Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. Recurrent models of visual attention. In *Advances in neural information processing systems*, pages 2204–2212, 2014.

[69] Lili Mou, Yiping Song, Rui Yan, Ge Li, Lu Zhang, and Zhi Jin. Sequence to backward and forward sequences: A content-introducing approach to generative short-text conversation. *arXiv preprint arXiv:1607.00970*, 2016.

[70] Lasguido Nio, Sakriani Sakti, Graham Neubig, Tomoki Toda, and Satoshi Nakamura. Utilizing human-to-human conversation examples for a multi domain chat-oriented dialog system. *IEICE TRANSACTIONS on Information and Systems*, 97(6):1497–1505, 2014.

[71] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[72] Ankur P Parikh, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit. A decomposable attention model for natural language inference. *arXiv preprint arXiv:1606.01933*, 2016.

[73] Romain Paulus, Caiming Xiong, and Richard Socher. A deep reinforced model for abstractive summarization. *arXiv preprint arXiv:1705.04304*, 2017.

[74] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[75] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[76] Ehud Reiter and Anja Belz. An investigation into the validity of some metrics for automatically evaluating natural language generation systems. *Computational Linguistics*, 35(4):529–558, 2009.

[77] Alan Ritter, Colin Cherry, and William B Dolan. Data-driven response generation in social media. In *Proceedings of the conference on empirical methods in natural language processing*, pages 583–593. Association for Computational Linguistics, 2011.

[78] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[79] Vasile Rus and Mihai Lintean. A comparison of greedy and optimal assessment of natural language student input using word-to-word similarity metrics. In *Proceedings of the Seventh Workshop on Building Educational Applications Using NLP*, pages 157–162. Association for Computational Linguistics, 2012.

[80] Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[81] samyzaf's Blog. Nlp. `http://www.samyzaf.com/ML/nlp/nlp.html`.

[82] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[83] Bit Search. Vector space model for scoring. `http://bitsearch.blogspot.com/2011/01/vector-space-model-for-scoring.html`.

[84] Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. Recurrent dropout without memory loss. *arXiv preprint arXiv:1603.05118*, 2016.

[85] Iulian Vlad Serban, Tim Klinger, Gerald Tesauro, Kartik Talamadupula, Bowen Zhou, Yoshua Bengio, and Aaron C Courville. Multiresolution recurrent neural networks: An application to dialogue response generation. In *AAAI*, pages 3288–3294, 2017.

[86] Iulian Vlad Serban, Alessandro Sordoni, Yoshua Bengio, Aaron C Courville, and Joelle Pineau. Building end-to-end dialogue systems using generative hierarchical neural network models. In *AAAI*, volume 16, pages 3776–3784, 2016.

[87] Iulian Vlad Serban, Alessandro Sordoni, Ryan Lowe, Laurent Charlin, Joelle Pineau, Aaron C Courville, and Yoshua Bengio. A hierarchical latent variable encoder-decoder model for generating dialogues. In *AAAI*, pages 3295–3301, 2017.

[88] Lifeng Shang, Zhengdong Lu, and Hang Li. Neural responding machine for short-text conversation. *arXiv preprint arXiv:1503.02364*, 2015.

[89] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. *Journal of computer and system sciences*, 50(1):132–150, 1995.

[90] Samuel L Smith, Pieter-Jan Kindermans, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[91] Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in neural information processing systems*, pages 926–934, 2013.

[92] Alessandro Sordoni, Michel Galley, Michael Auli, Chris Brockett, Yangfeng Ji, Margaret Mitchell, Jian-Yun Nie, Jianfeng Gao, and Bill Dolan. A neural network approach to context-sensitive generation of conversational responses. *arXiv preprint arXiv:1506.06714*, 2015.

[93] SQLML. Recurrent neural network. `http://sqlml.azurewebsites.net/2017/08/12/recurrent-neural-network/`.

[94] Amanda Stent, Matthew Marge, and Mohit Singhai. Evaluating evaluation methods for generation in the presence of variation. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 341–351. Springer, 2005.

[95] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[96] Tensorflow. Neural machine translation (seq2seq) tutorial. `https://github.com/tensorflow/nmt`.

[97] The Seattle Times. Baidu research chief andrew ng fixed on self-taught computers self-driving cars. `https://www.seattletimes.com/business`.

[98] Trieu H Trinh, Andrew M Dai, Thang Luong, and Quoc V Le. Learning longer-term dependencies in rnns with auxiliary losses. *arXiv preprint arXiv:1803.00144*, 2018.

[99] Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics, 2010.

[100] 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 6000–6010, 2017.

[101] Oriol Vinyals and Quoc Le. A neural conversational model. *arXiv preprint arXiv:1506.05869*, 2015.

[102] Chris Welch. Google just gave a stunning demo of assistant making an actual phone call. `https://www.theverge.com/2018/5/8/17332070/google-assistant-makes-phone-call-demo-duplex-io-2018`.

[103] Lilian Weng. from gan to wgan. `https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html`.

[104] John Wieting, Mohit Bansal, Kevin Gimpel, and Karen Livescu. Towards universal paraphrastic sentence embeddings. *arXiv preprint arXiv:1511.08198*, 2015.

[105] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[106] Yu Wu, Wei Wu, Zhoujun Li, and Ming Zhou. Response selection with topic clues for retrieval-based chatbots. *arXiv preprint arXiv:1605.00090*, 2016.

[107] Chen Xing, Wei Wu, Yu Wu, Jie Liu, Yalou Huang, Ming Zhou, and Wei-Ying Ma. Topic aware neural response generation. In *AAAI*, volume 17, pages 3351–3357, 2017.

[108] Chen Xing, Wei Wu, Yu Wu, Ming Zhou, Yalou Huang, and Wei-Ying Ma. Hierarchical recurrent attention network for response generation. *arXiv preprint arXiv:1701.07149*, 2017.

[109] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057, 2015.

[110] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1480–1489, 2016.

[111] Kaisheng Yao, Geoffrey Zweig, and Baolin Peng. Attention with intention for a neural network conversation model. *arXiv preprint arXiv:1510.08565*, 2015.

[112] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.

[113] Yizhe Zhang, Dinghan Shen, Guoyin Wang, Zhe Gan, Ricardo Henao, and Lawrence Carin. Deconvolutional paragraph representation learning. In *Advances in Neural Information Processing Systems*, pages 4172–4182, 2017.

[114] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.