Università degli Studi dell'Insubria
Dipartimento di Scienze Teoriche e Applicate DiSTA
Master's Degree in Computer Science

Investigating the role of Word Embeddings in sentiment analysis and text generation.

Advisor: **Prof.ssa Elisabetta Binaghi**

Co-Advisors: **Samuele Martinelli, Mirko Puliafito**

Candidate: **Dario Bertolino**

Academic year 2018 – 2019

Index

INTRODUCTION 3

1. EVOLUTION OF NLP TECHNIQUES 6

1.1 MACHINE LEARNING APPROACH 6

1.2 NEURAL NETWORKS 9

1.3 DEEP LEARNING APPROACH 15

1.3.1 Convolutional Neural Networks 16

1.3.2 Recurrent Neural Networks 19

1.4 WORD EMBEDDINGS 21

2. SENTIMENT ANALYSIS: PRACTICAL TESTS 27

2.1 WORD2VEC 27

2.2 PRACTICAL TESTS 32

2.2.1 Data preparation 33

2.2.1 Experiment One: Pre-trained Embeddings in NNs 36

2.2.2 Experiment Two: Pre-trained Embeddings in CNNs 44

2.3 RESULTS DISCUSSION 47

3. TEXT GENERATION: PRACTICAL TESTS 49

3.1 LSTMS 50

3.2 ENCODER-DECODER ARCHITECTURE 52

3.3 ATTENTION MECHANISM 55

3.4 THE TRANSFORMER 57

3.5 PRACTICAL TESTS 61

3.5.1 Experiment One: LSTM *62*

3.5.2 Experiment Two: Encoder-Decoder LSTM *66*

3.5.3 Experiment Three: The Transformer *72*

3.6 RESULTS DISCUSSION 79

4. CONCLUSIONS AND FUTURE WORK 82

4.1 CONCLUSIONS 82

4.2 FUTURE WORK 83

REFERENCES 85

LIST OF FIGURES 90

Introduction

The automatic comprehension of human language is becoming more important every day, by now software interact with people, analyze their responses and even generate their own. The applications of such technology are everywhere, customer support is often performed by a chatbot and complex neural networks are used to generated articles, conversations and even books. How can a computer be programmed to understand, process, and generate language just like a person? In the 80s, natural language experts used to define complex rules to recognize syntax and semantics of text. It was popular in document classification to use these rules to manually build custom systems able to take text categorization decisions [1]. This approach has been abandoned over time in favor of machine learning algorithms, general inductive processes which automatically build classifiers by learning. During decades, Natural language processing (NLP) expanded its range of tasks, from document classification to automatic text summarization, sentiment analysis, topic extraction, named entity recognition, text mining, machine translation, automatic question answering and others. In 2018, T. Young described NLP as a theory-motivated range of computational techniques for the automatic analysis and representation of human language [2]. He also showed that Deep learning and word embeddings are two keywords in this field today. The former enables multi-level automatic feature representation learning, the latter completely changed the way text is represented, transforming each word in a dense vector that incapsulate information about the context in which the word usually appears. The aim of this work is to collect information on how deep models are structured and which is the influence of word embeddings in sentiment analysis and text generation. Chapter 1

contains an overview on the evolution of NLP techniques over time, from custom classifiers to machine learning and deep learning. This chapter is also considered a preliminary study that justify all experiments done. Chapter 2 is dedicated to practical tests on sentiment analysis, using reviews from an Italian chatbot service. The task is performed with simple neural networks and convolutional neural networks, focusing on how to produce and manage word embeddings, each network is trained with and without them in order to measure any influence. Chapter 3 first describes three complex Deep Learning models: LSTMs, Encoder-Decoder architecture and the Transformer. A text generation task is then performed with increasing complexity, starting from a training with Aesop fables, ending up trying with sci-fi books. Chapter 4 contains a brief discussion about results, the experiments showed how word embeddings can boost performances of convolutional neural networks in a text categorization tasks such as sentiment analysis. More of that, the creation of this vector representation for words is a relatively short process and it helps to reduce the training time. Recurrent Neural Networks instead, are not a good match with pre-trained embeddings because the computational heaviness of recurrent layers seems to put in a shade the vector representation of words in input. In reverse, tests with text generation have also shown that they can have a positive influence in networks with positional encoding layers, improving the semantic correctness of the generated text. Google Word2Vec is the only set of techniques used in this work, but many other solutions already exist, opening to further tests and future works, like described in Chapter 5. For instance, in 2016 D. Tang tried towards the creation of word embeddings that are specific for sentiment analysis. Text generation was performed with small models and Python code running on single GPU, in order to obtain better results in the generation of complex

text such as sci-fi stories, larger models need to be trained, probably in a distributed environment. However, experiments presented in this document can help understand the right path to reach this objective.

1. Evolution of NLP techniques

Document classification is one of the first tasks that Natural Language Processing has tried to solve, early solutions were based on a Knowledge Engineering (KE) approach. By building a classifier in such a way, all effort was dedicated to the construction of single classifier for a very specific problem. In 2001, the research community dominant approach to text classification was based on machine learning (ML) techniques: a general inductive process automatically builds a classifier by learning, from a set of pre-classified documents, the characteristics of the categories [1]. The advantages of the ML approach over the KE are evident because the engineering effort goes toward the construction not of a classifier, but of an automatic builder of classifiers, the learner. This means that all that is needed are manually classified documents to make the automatic builder learn from them. More of that, if a classifier already exists and the original set of categories is updated, or if the classifier is ported to a completely different domain, it could be trained again. All these benefits were a very good effectiveness, considerable savings in terms of expert labor power, and straightforward portability to different domains.

1.1 Machine Learning approach

F. Sebastiani formalized Text Categorization (TC) problem to explain how the machine learning approach works. A resume of its explanation is now reported. TC is the task of assigning a Boolean value to each pair $< d_j , c_i > \in D \times C$, where D is a domain of documents and C is a set of pre-defined categories. The goal is to approximate the target function $\breve{\Phi}: D \times C \rightarrow \{T, F\}$ by means of a function $\Phi: D \times C \rightarrow \{T, F\}$ called the

classifier. This process works with categories that are just symbolic labels and no metadata such as publication date or document type. The ML approach for document classification relies on the availability of an initial corpus $\Omega = \{d_1, \ldots, d_{|\Omega|}\} \subset \mathrm{D}$ of documents pre-classified under $\mathrm{C} = \{c_1, \ldots, c_{|\mathrm{C}|}\}$. The values of the target function $\breve{\Phi}: D \times C \rightarrow \{T, F\}$ are known for every pair $< d_j , c_i > \in \Omega \times C$. A document d_j is a positive example of c_i if $\breve{\Phi}(d_j , c_i) = T$, a negative example of c_i if $\breve{\Phi}(d_j , c_i) = F$. The case in which exactly one category must be assigned to each document is often called the single-label case, while the case in which any number of categories from the predefined set may be assigned to the same document is dubbed the multilabel case. A special case of single-label TC is binary TC, in which each document must be assigned either to category c_i or to its complement $\bar{c}_i$. Solving the binary case also means solving the multilabel case, which is also representative of important TC applications, including automated indexing for Boolean systems, indeed one needs only transform the problem of multilabel classification into independent problems of binary classification. However, this requires that categories be stochastically independent of each other. The inductive construction of a ranking classifier for category $c_i \in C$ usually consists in the definition of a function $CSV_i: D \rightarrow [0,1]$ that, given a document d_j, returns a categorization status value for it, that is, a number between 0 and 1. Documents are then ranked according to their CSV_i value for document-ranking TC while for category-ranking TC is usually tackled by ranking, for a given document d_j , its CSV_i scores for the different categories in $C = \{c_1, \ldots , c_{|C|}\}$. The inductive construction of text classifiers has been tackled in many ways and F. Sebastiani described the most popular until 2002 [1]. A **Probabilistic Classifier** view $CSV_i(d_j)$ in terms of $P(c_i \mid \overrightarrow{d_j})$, that is, the

probability that a document represented by a vector $\overrightarrow{d_j} = < w_{1j}, \ldots, w_{|T|j} >$ of (binary or weighted) terms belongs to c_i, and compute this probability by an application of Bayes' theorem, given by:

$$P(c_i \mid \overrightarrow{d_j}) = \frac{P(c_i)P(\overrightarrow{d_j} \mid c_i)}{P(\overrightarrow{d_j})}$$

A **Decision tree (DT) text classifier** is a tree in which internal nodes are labeled by terms, branches departing from the mare labeled by tests on the weight that the term has in the test document, and leaves are labeled by categories. Such a classifier categorizes a test document d_j by recursively testing for the weights that the terms labeling the internal nodes have in vector $\overrightarrow{d_j}$ until a leaf node is reached; the label of this node is then assigned to d_j. Most such classifiers use binary document representations, and thus consist of binary trees.

A **Decision rule classifier** for category c_i built by an inductive rule learning method consists of a $DNF\ rule$, that is, of a conditional rule with a premise in disjunctive normal form (DNF). The literals in the premise denote the presence (non-negated keyword) or absence (negated keyword) of the keyword in the test document d_j, while the clause head denotes the decision to classify d_j under c_i. DNF rules are like DTs in that they can encode any Boolean function. However, an advantage of DNF rule learners is that they tend to generate more compact classifiers than DT learners.

Various TC efforts have used **Regression models**. Regression denotes the approximation of a real-valued (instead than binary, as in the case of classification) function ϕ by means of a function ϕ that fits the training data. For instance, consider Linear Least-Squares Fit (LLSF), in which each

document d_j has two vectors associated to it: an input vector $I(d_j)$ of $|T|$ weighted terms, and an output vector $O(d_j)$ of $|C|$ weights representing the categories. Weights for this latter vector are binary for training documents and are nonbinary CSV's for test documents. Classification is somehow the task of determining an output vector $O(d_j)$ for test document d_j, given its input vector $I(d_j)$; hence, building a classifer boils down to computing a $|C| \times |T|$ matrix $\widehat{M}$ such that $\widehat{M}I(d_j) = O(d_j)$.

A **Linear Classifier** for category c_i is a vector $\overrightarrow{c_i} = < w_{1_i}, \ \dots, w_{|T|_j} >$ belonging to the same $|T| - dimensional\ space$ in which documents are also represented, and such that $CSVi(d_j)$ corresponds to the dot product $\sum_{k=1}^{|T|} w_{ki} w_{kj}$ of $\overrightarrow{d_j}$ and E $\overrightarrow{c_i}$. Note that when both classifier and document weights are cosine-normalized, the dot product between the two vectors corresponds to their cosine similarity, that is:

$$S(c_i, d_j) = \cos(\alpha) = \frac{\sum_{k=1}^{|T|} w_{ki} \cdot w_{kj}}{\sqrt{\sum_{k=1}^{|T|} {w_{ki}}^2} \cdot \sqrt{\sum_{k=1}^{|T|} {w_{kj}}^2}};$$

which represents the cosine of the angle α that separates the two vectors. This is the similarity measure between query and document computed by standard vector space IR engines, which means in turn that once a linear classifier has been built, classification can be performed by invoking such an engine.

1.2 Neural Networks

Neural Networks are a class of models within the general machine learning literature, a specific set of algorithms that have revolutionized machine learning. They are themselves general function approximations, which is

why they can be applied to almost any machine learning problem about learning a complex mapping from the input to the output space. Neural Networks are the connection model between Machine Learning and Deep Learning.

A **Neural Network (NN) text classifier** is a network of units, where the input units represent terms, the output unit(s) represent the category or categories of interest, and the weights on the edges connecting units represent dependence relations [1]. For classifying a test document d_j, its term weight $w_{k\,j}$ are loaded into the input units; the activation of these units is propagated forward through the network, and the value of the output unit(s) determines the categorization decision(s). A typical way of training NNs is backpropagation, whereby the term weights of a training document are loaded into the input units, and if a misclassification occurs the error is "backpropagated" so, as to change the parameters of the network and eliminate or minimize the error. A Neural Network is basically a function that represent an approximation of another highly computationally expensive function [3], as shown in figure 1.

Figure 1 - Representation of a Neural Network as a function

The approximation is obtained by a supervised training of the network with the backpropagation process. The structure of the network is described in figure 2. It is composed by three major levels: input, hidden and output. The first and the last one simply applies linear function to determine if a neuron is active or not for that step while the hidden stratus, that could be composed by multiple layers, apply a non-linear function to data in order to transform them. One of the most used is the sigmoid

function because of its derivate easy to calculate, $f(x) = f(x)(1 - f(x))$. The derivative of the non-linear function applied by the hidden layer, is crucial during the backpropagation process.

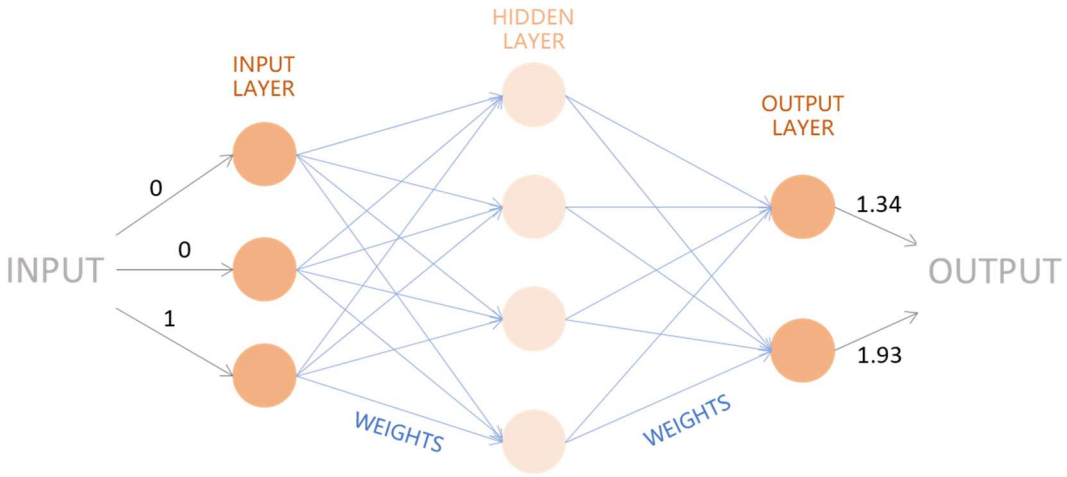


Figure 2 – Basic structure of a Neural Network: Input layer, Hidden layer and Output layer

Another important element in the network is the weight assigned to each connection between nodes, values that are adjusted at each training step to better approximate the highly computationally expensive function. R. Colvin described backpropagation in a simple way to model Neural Networks with process algebra [3].

To understand the mathematical behavior of a neural network, he considered this naming convention:

$\mathcal{N}$	Set of neuron identifiers
I, H, O	Input, hidden and output identifiers, mutually exclusive.
i, h, o	A generic input, hidden or output neuron identifier.
x	A general variable; an input list, one entry per input neuron
y	A general variable; an output list, one entry per output neuron
z	The activation (output) of a neuron

$x_i\ y_i\ z_i$	Where x is a function (or list), let x_i abbreviate $x(i)$
w_{hi}	The weight from neuron i to neuron h
δ_h	The approximate error at neuron h
f	An activation function used by hidden neurons
$\dot{f}$	The derivative of function f.
μ	The learning rate
κ	A value of any type

First step is the activation of the neurons in the hidden layer, where each hidden neuron receives inputs from connections and calculate its activation, $\forall\ h\ \in H \bullet z_h = f(\sum_{i \in I} w_{hi} \cdot x_i)$, as shown in figure 3.

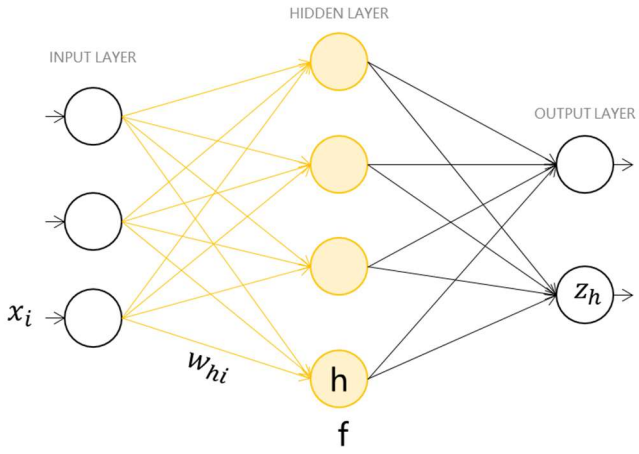


Figure 3 – Math behind hidden neurons receiving input and computing their activation

Then output neurons receive inputs from the hidden neurons and calculate their outputs in the same way, except for the activation non-linear function, $\forall o \in O \bullet z_o = \sum_{h \in H} w_{oh} \cdot z_h$, as shown in figure 4. The process just described is the feedforward behavior of the network:

$$\forall o \in O \bullet y'_o = \sum_{h \in H} w_{oh} \cdot f(\sum_{i \in I} w_{hi} \cdot x_i)$$

In one single function is pointed out why Neural Networks find large application on modern calculators, because this "parallel" computation of neurons can be executed with matrix multiplications, perfect for a Graphical Process Units – GPU. After the mismatch between network output and expected value has been calculated, the backpropagation process for that input data can start, $\forall o \in O \bullet \delta_o = z_o - t_o$. This process is described in figure 5.

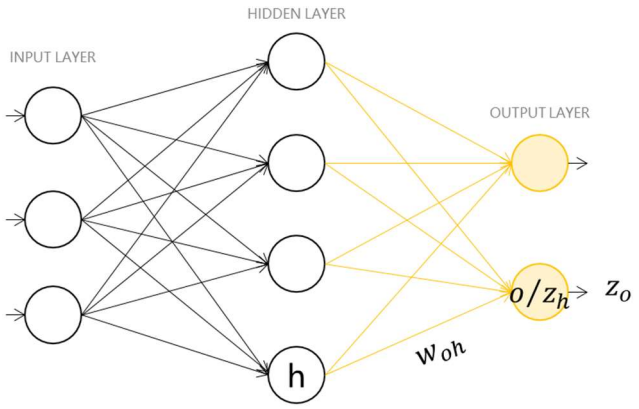


Figure 4 – Math behind output neurons receiving input and computing their activation

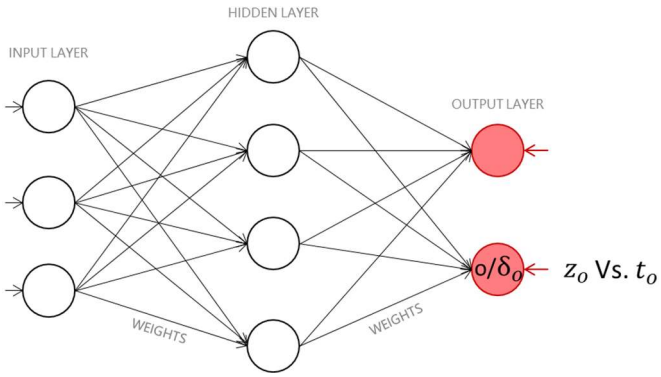


Figure 5 – Math behind Neural network producing an output, comparing it with a target valu.

Connections weighs are updated simply subtracting the mismatch multiplied by a learning rate and the previous input to that connection, $\forall\, o \in O, h \in H \bullet w'_{oh} = w_{oh} - \mu \cdot \delta_o \cdot z_h$, as described in figure 6.

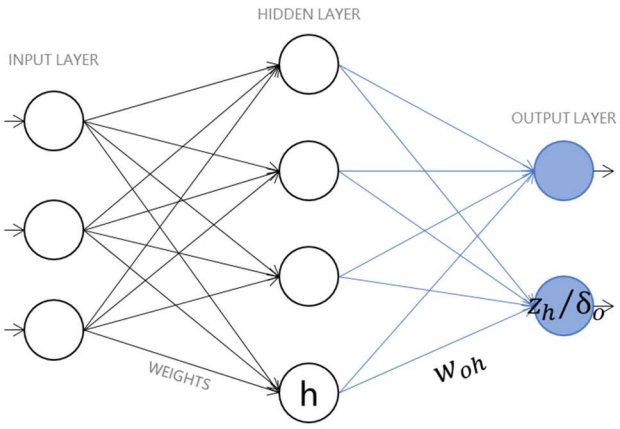


Figure 6 Math behind connection weights getting updated in a Neural Network

The error on the Hidden nodes is calculated with the derivative of the non-linear activation function in order to turn back from that process. So, the derivate is multiplied with the sum of all the products between error from the output node connected and the weights of each connection to that hidden node, $\forall\, h \in H \bullet \delta_h = z_h(1 - z_h) \cdot (\sum_{o \in O} w_{oh} \cdot \delta_o)$. This can be observed in figure 7.

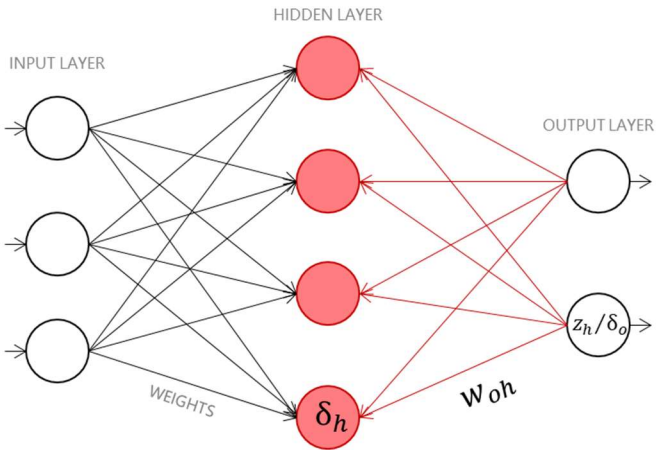


Figure 7 – Math behind Backpropagation of the error in hidden layers of a Neural Network.

The connections weights update remains always the same. Even if we add an enormous number of hidden layers, the behavior of the entire network remains the same. Backpropagation is just a mathematical optimization, which is, essentially, what of all neural networking is. Feeding the network with a right number of samples will automatically adjust all the weights to approximate and recognize a given problem/function. Deep Learning basically is the composition of more different types of Neural Networks that, once composed results in a fully trainable network that can backpropagate end-to-end.

1.3 Deep Learning approach

For decades, Natural language processing (NLP) has been based on models trained with high dimensional, sparse and hand-crafted features. Deep Learning models introduced multi-level automatic feature representation learning, employing multiple processing neural network layers to learn hierarchical representations of data, producing state-of-the-art results in many domains, as in NLP. Neural Networks can resolve very different type of problems. However, knowing the format of the input (text, image or sequence) it is possible to obtain better performance [4]. Considering a network able to determine if an image contains a human image, do it really matter if the human is in the right or the left corner of the picture? Or if a network is analyzing a text that talks about houses, do the meaning of the word house change from a sentence to another? The answer is no, but a neural network needs a way to do not relearn the same concept every time it comes in input surrounded by a different context. Things that don't change on average across time or space are everywhere, because of statistical invariance phenomenon. When two inputs can contain the same kind of information, a network needs to share weights and train them

jointly for those inputs. This very important idea is the base concepts behind new kind of deep neural networks able to compute the statistical invariance of the data.

1.3.1 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a neural network that shares weights across space. CNNs are great for image processing but they can also be applied to text data. An image has three dimensions, width, height and depth. The fundamental idea is to consider a small patch of the image and run a neural network with *k* outputs on it [4]. This idea is represented in figure 8. The running neural network is slided across the image without changing its weights, obtaining as output a new image with different height, width and depth.

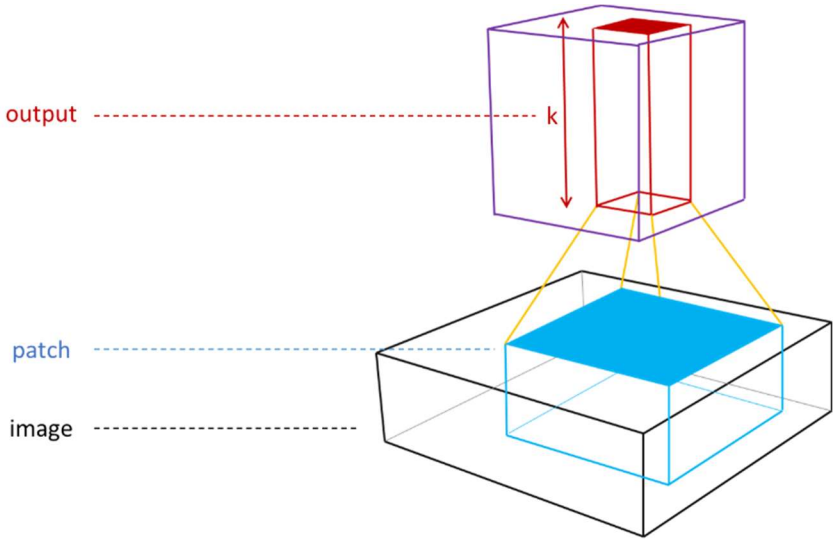


Figure 8 - Graphical representation of a CNN patch applied to a tridimensional input.

This operation is called *convolution*, bearing in mind that if the patch has the same size of the image, it will be no different than a regular neural network layer, but because the presence of the small patch instead, many fewer weights are shared across space. A CNN is basically a deep network

composed by stacks of convolutions instead of stacks of matrix multiplications.

Few parameters must be considered while designing a CNN:

- **Depth**: refers to the third dimension of an image and each layer of the CNN has the scope to increase it learning regularities about training images. In the very first layers, the regularities are curves and edges, then when you go deeper along the layers it starts learning higher levels of regularities such as colors, shapes, objects.
- **Patch / Kernel**: subsection of an input image.
- **FeatureMap**: is the output of one convolution. For instance, in figure 8 it was mapping three features to k features.
- **Stride**: is the number of pixels that are shifted each time the filter moves on the input image. With a stride of one the output width and length will remain the same as the input. With a stride of 2 the it will be half sized. A graphical representation of the stride can be found in figure 9.
- **Padding**: represent the action of passing the edges of the input image. Choosing a valid stride that lets the filter don't pass the edges results in a *valid padding*. Another solution is to apply a padding of zeros around the image and this operation take the name of *same padding*.

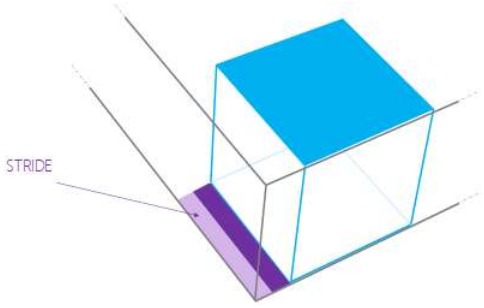


Figure 9 - Graphical representation of stride parameter in CNNs

At the end of the stack of convolutions, few simple fully connected layers receive the data so that a classifier can be trained. The training process of this type of structure is quite different from a simple neural network backpropagation. Because the weights are shared, calculating the error, derivatives must be summed up:

$$\frac{\Delta £}{\Delta W} = \sum \frac{\Delta £}{\Delta W}(X_{\mathrm{n}})$$

Where £ is the error, W is the shared weight and X_{n} are the inputs sharing weights. This is the base concept behind a CNN but in practice other features, mechanism and architectures exists:

- **Pooling**, an operation applied with a very small stride, for example one, that take a certain number of convolutions in a neighborhood and combines them somehow. This is done to reduce the loss of information caused by the stride operation itself. *Max pooling* compute the maximum of all responses around a point in the feature map, $y = \max(X_i)$. A famous application sequence in 'LENET-5' by Yan Lecun '98 and 'ALEXNET' by Alex Krizhevsky '12**.**
- **1x1 convolution**, applying convolutions operations to pixels one by one helps the networks to go deeper in very inexpensive way and without changing the structure.
- **Inception architecture** is a combination of the average pooling and 1x1 convolutions that creates neural networks that are at the same time smaller and better than convnets that simply uses pyramid of convolutions. Instead of applying just one of these mechanisms, it is possible to create what it is called an *inception module:* average pooling, 1x1 convolution and normal convolutions are calculated for the same fraction of the input in parallel and their outputs are simply concatenated.

Stacks of convolutions applied to images can help recognize that two humans within an image are object of the same nature. Instead of image pixels, the input to most NLP tasks are sentences or documents represented as a matrix. Each row of the matrix corresponds to one token, typically a word, but it could be a character. That is, each row is a vector that represents a word [5]. Stacks of convolutions applied to text help to recognize that even if the word "house" twice in a text, the meaning is always the same, the two tokens can share weights in the network [4].

1.3.2 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a neural network that shares weights across time [4]. When considering learning the mean of speech and real text, one of the most important problem is the length of the sequence of words. The goal is to be able to apply a classifier at each word in the sequence, which is capable of consider the classification states of the previous words. As shown in figure 10, the architecture of an RNN end up with a part of the classifier connected to the input at each time step and another part called the recurrent connection of the same classifier connecting to past at each step.

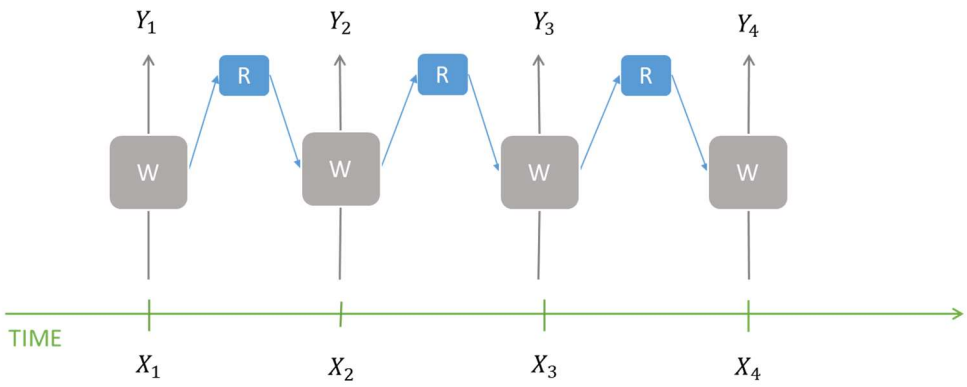


Figure 10 - Graphical representation of the workflow of an RNN over time

Considering the simplest neural network structure again, an RNN presents every neuron in the hidden layer connected to each other neuron in the hidden layer, as can be seen in figure 11.

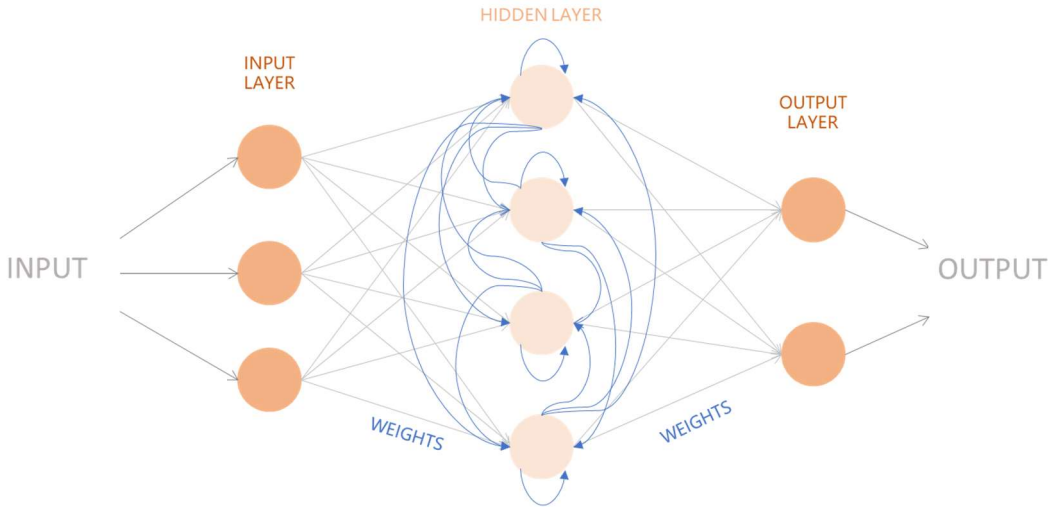


Figure 11 – Simplest structure of RNNs: hidden layers are connected to each other's

The essential difference with classic neural networks is that RNNs contain cycles, and as such have a form of short-term memory. This means that the output for a given input may differ depending on the preceding inputs. The input to the network remains as a list of binary values, but now all inputs occur in a discrete time series, so that each input occurs before or after every other input. After (or in parallel) a hidden neuron receives input from the environment, it then receives the previous activation of each hidden neuron. The activation of the hidden neuron combines the standard input with the input from the other hidden neurons. Many variants on this approach exists in literature, however this example in figure 11 has the benefit of showing formally the ordering of events and those that occur in parallel. The backpropagation process in a RNNs needs to backpropagate multiple derivatives through time all the way to beginning of the sequence, as shown in figure 12. Derivatives through time need to be applied to the same parameters at once and this is not good for stochastic gradient

descent that can explode or vanish. To prevent gradient from exploding it can be applied what is called the gradient clipping, that calculate the gradient norm and shrink steps when the norm grows too big. However, vanishing gradient is the biggest problem because it makes forget more distant past causing a memory loss in RNNs.

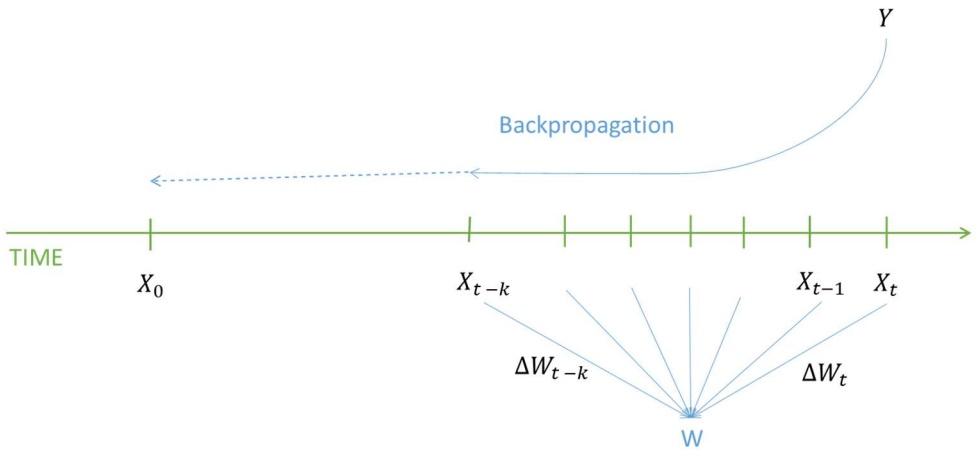


Figure 12 - Backpropagation process in RNNs involving multiple derivatives through time

1.4 Word Embeddings

The way data is encoded is a crucial process for data analysis in general, not only for NLP. Categorical data are input features that represent one or more discrete items from a finite set of choices [6], e.g. the set of movies a user has watched, the set of words in a document, or the occupation of a person. Categorical data is efficiently represented via sparse tensors, which are vectors with very few non-zero elements. As shown in figure 13, a movie recommendation model can assign a unique ID to each possible movie, and then represent each user by a sparse tensor of the movies they have watched. Each row of the matrix is an example capturing a user's movie-viewing history, represented as sparse tensors because each user only watches a small fraction of all possible movies.

	film1	film2	film3	film4	film5	film6	film7	film8	. . .	filmN
userID1	1	1	0	0	0	0	1	1	. . .	1
userID2	0	0	0	1	0	0	0	0	. . .	0
userID3	0	1	0	1	0	0	0	0	. . .	1
userID4	0	1	0	1	1	0	0	0	. . .	0
userID5	0	0	0	0	1	1	1	0	. . .	1
userID6	1	0	0	0	0	0	1	0	. . .	1
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .
userIDn	1	0	0	1	0	1	1	0	. . .	1

Figure 13 - Sparse vectors representing users in a movie recommendation model

Words, sentences, and documents can be represented as sparse vectors where each word in the vocabulary plays the same role of the movies in this recommendation example. In order to use such representations within a machine learning system, sparse vectors need to be represented as vectors of numbers so that semantically similar items (movies or words) have similar distances in the vector space. How represent a word as a vector of numbers? The simplest way is to define a Neural Network input layer in which a node corresponds to a word in a vocabulary, or at least a node for every word that appears in the data the network will be analyzing. If 500,000 unique words appear in the data, a word could be represented with a length 500,000 vector and assign each word to a slot in that vector. If the word "horse" is at index 1247, then to feed "horse" into the network, a 1 must be copied into the 1247th input node and 0s into all the rest. The representation just described is called a *one-hot encoding*, because only one index has a non-zero value. More typically a vector might contain counts of the words in a larger chunk of text. In that case the representation is called *bag of words*. In a bag-of-words vector, several of the 500,000 nodes would have non-zero value. However, determine the non-zero values, one node per word gives very *sparse* input vectors, very large vectors with relatively few non-zero values. Sparse representations have a couple of problems

that can make it hard for a model to learn effectively. Huge input vectors result in a very big number of weights for a neural network. If there are M words in the vocabulary and N nodes in the first layer of the network above the input, M x N weights will be necessary to train that layer. Large number of weights causes further problem such as the amount of data needed, the more weights in the model, the more data are needed to train it effectively. Or the amount of computation, the more weights, the more computation required to train and use the model. It's easy to exceed the capabilities of a hardware, more of that, a significant lack of meaningful relations between vectors must be faced. Feeding the pixel values of RGB channels into an image classifier, it makes sense to talk about "close" values. Reddish blue is close to pure blue, both semantically and in terms of the geometric distance between vectors. But a vector with a 1 at index 1247 for "horse" is not any closer to a vector with a 1 at index 50,430 for "antelope" than it is to a vector with a 1 at index 238 for "television". The solution to these problems is to use *Embeddings*, which translate large sparse vectors into a lower-dimensional space that preserves semantic relationships [6]. Core problems of sparse input data can be resolved by mapping high-dimensional data into a lower-dimensional space able to give a machine learning system the opportunity to detect patterns that may help with the learning task. While enough dimensions to encode rich semantic relations are good, an embedding space that is small enough can allow to train the system more quickly. A useful embedding may be on the order of hundreds of dimensions. This is likely several orders of magnitude smaller than the size of a typical vocabulary for an NLP task. Consider the objective of classify documents, understand what they are talking about like politics, medicine or other topics. The first problem to be solved with deep learning is to search for words that are significantly rare. For instance,

cholecystectomy is a word with a very low probability to be named in the entire English vocabulary and it tells very much about the topic of a text. Words relation is another important problem because to share parameters between different words with the same meaning, "home" and "house" for example, the only way that is to learn that they are related. Unfortunately, these two problems require way too much of label data for any task that really matters. The approach that produces very good results is to apply unsupervised learning to everyday text, which can be found everywhere on the web, exploiting a very simple concept: "*Similar words, appears in similar contexts*" [6]. In this way words with the same meaning are classified without even know which the mean is. Words are classified in small vectors called word embeddings, which will be close to each other if they appear in similar contexts and far from each other if they don't. A simple example of this classification can be seen in figure 14.

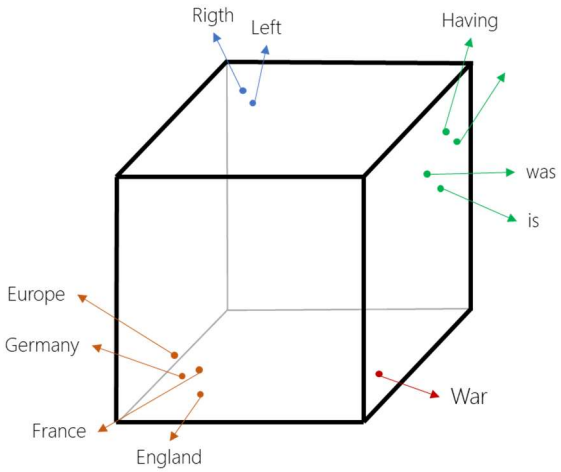


Figure 14 - Graphical example of a word embeddings space, in which similar words are represented by close vectors

For example, Word2vec is a surprisingly simple model that works very well, based on a logistic linear classifier, in which distance between vectors is calculated using a cosine distance. Figure 15 describe the cosine distance

between two vectors. This type of distance is perfect because length of the embedded vector is not relevant to the classification, in fact it's often better to normalize all embedded vector to simply have unit norm.

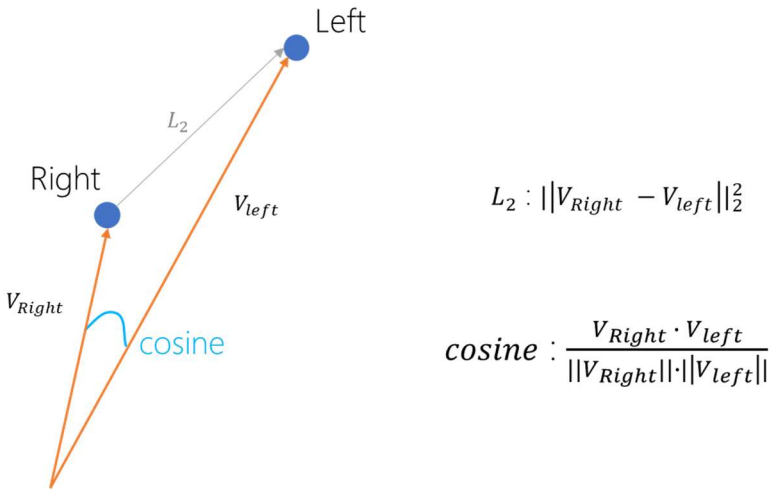


Figure 15 - Cosine distance formula and graphical example

Another important thing to be taken in mind during comparing process is that the linear logistic classifier produces a SoftMax output that is then compared to a target, another word that appear in the same context classified. The problem, of course is that there might be many words in the vocabulary and the comparing process could be very inefficient. One solution is to perform a sampled SoftMax, described in figure 16. The idea is to randomly sample the vectors eliminating part of the non-classified parts, acting like they simply do not exist, resulting in faster execution with no costs in performance. However, there are many existing mathematical techniques for capturing the important structure of a high-dimensional space in a low dimensional space. In theory, many techniques could be used to create an embedding for a machine learning system. For example, Principal Component Analysis - PCA has been used to create word embeddings. Given a set of instances like bag of words vectors, PCA tries to find highly correlated dimensions that can be collapsed into a single

dimension. Another technique to learn an embedding is make it a part of the neural network for the target task. This approach gets an embedding well customized for a specific system but may take longer than training the embedding separately. In general, sparse data (or dense data to embed), it is possible to create an embedding unit that is just a special type of hidden unit of size d. This embedding layer can be combined with any other features and hidden layers, as in any DNN, the final layer will be the loss that is being optimized.

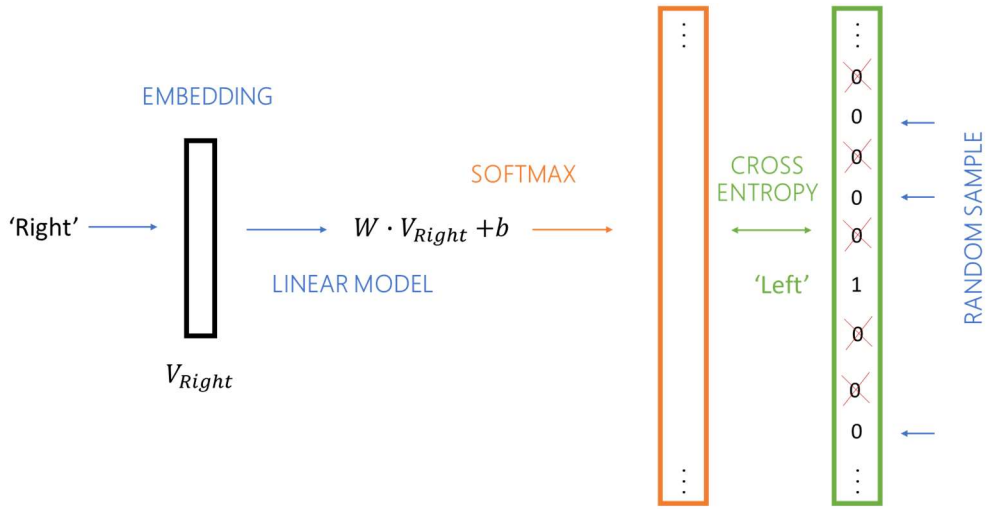


Figure 16 – Graphical example of Sampled SoftMax

2. Sentiment analysis: practical tests

The aim of this work is to build a neural network to perform a sentiment analysis on Italian sentences. Sentiment analysis is a data mining process which identifies and extracts subjective information from text. It could help to understand the social sentiment of clients, respect a business product or service. It could be a simple classification task that analyses a sentence and tells whether the underlying sentiment is positive, negative or neutral. Potentiality of deep learning techniques made this simple classification task evolve, creating new more complex sentiment analysis, e.g. Intent Analysis and Contextual Semantic Search [7]. The scenario of this experiment involves the presence of a chatbot for customer service. Whenever a customer talks with the chatbot to get help, the system asks him to leave a review in order to evaluate the chatbot work. The sentiment information from the review of a client can be used in different ways, for instance as a feature to increase the accuracy of a classifier which automatically evaluate the chatbot sessions. Another important goal of this test is to make practice with word embeddings, learning how to generate and use them correctly.

2.1 Word2Vec

Word2Vec is a famous collection of two-layer neural networks invented in 2013 by T. Mikolov et al. [8]. These models process text and create word vector representations. The purpose and usefulness of Word2vec is to group the vectors of similar words together in a vector space, detecting similarities mathematically. It creates vectors that are distributed numerical representations of word features. Considering the context of a

word as the other words that surround it, two models differ from the target to be predicted. The former, called Continuous Bag of Words (CBOW), it uses the context to predict a target word. The latter is the reverse of CBOW, called SKIP-GRAM, it uses a word to predict the target context. A graphical representation of these two models can be found in figure 17. Once trained on text, a word2Vec model provides a layer which can be put into deep networks to improve their performances. So, instead of re-use the entire network for the task it was trained on, the goal is actually just to learn the weights of the hidden layer [9]. The skip-gram model will be described in detail to understand how the vector representation of words is managed by a neural network.

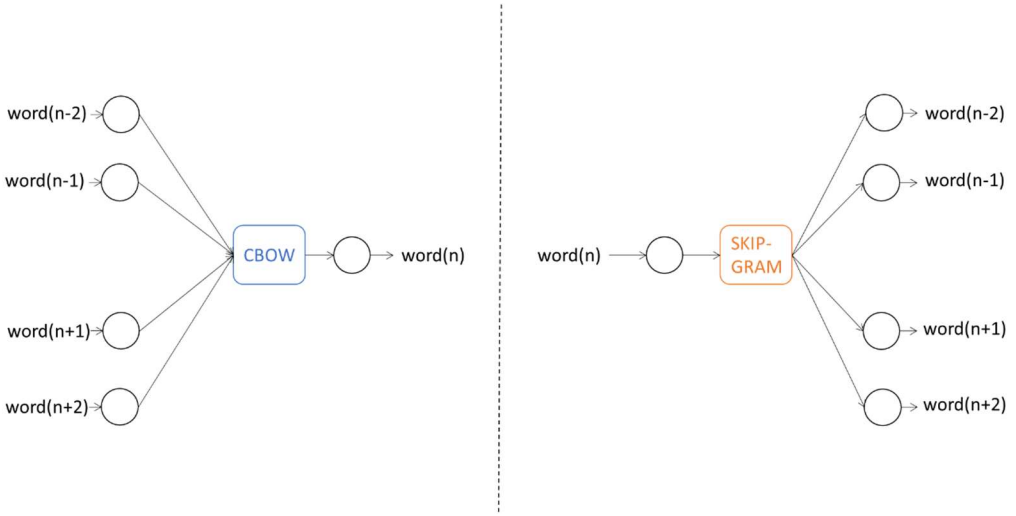
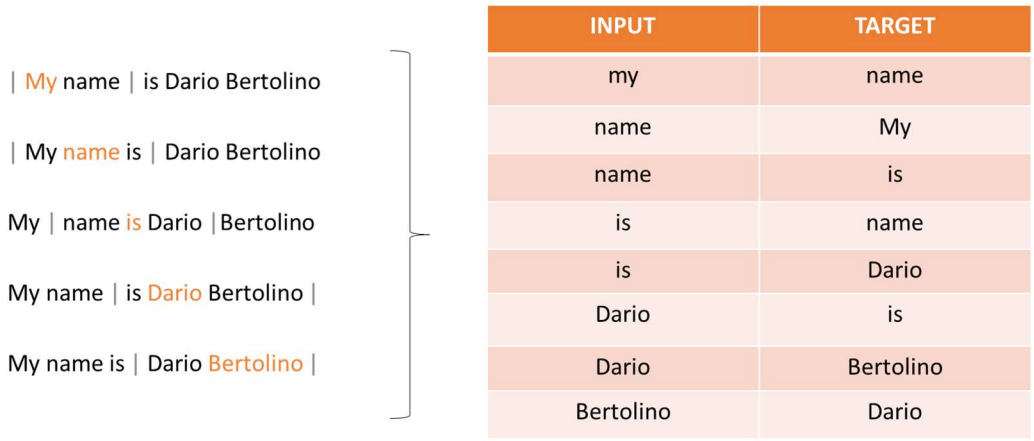


Figure 17 - Word2Vec CBOW vs. SKIP-GRAM

Considering a simple phrase “My name is Dario Bertolino”, the model receives as input each word, one after the other. The target of each word in input is a collection of words that surround it. For example, figure 18 describes the creation of input/target pairs, considering a sliding context of three words. When considering an input in the sequence, so many input/target pairs are created as is the number of words in the context of

the input one. In this way the network is going to learn the statistics from the number of times each pair shows up [9].



INPUT	TARGET
my	name
name	My
name	is
is	name
is	Dario
Dario	is
Dario	Bertolino
Bertolino	Dario

Figure 18 - Extraction of inputs and targets from text for a Word2Vec SKIP-GRAM model

The creation of inputs and targets for a CBOW model is very similar. This process can be automated with a simple software and it lets any Word2Vec model to be trained on any kind of text using an unsupervised method of learning. Naturally, the position of a single word in the final vector space will be defined by the type of text and the quantity of words used to train the network, so it will differ from literature, news, sci-fi books and so on. However, the same kind of words usually appear in the same context, the syntactic structure of a language is always recognized because articles will be always close to other articles or prepositions will be always close to other prepositions. The same rule is valid also for more abstract concepts like words which indicates names of animals or names of states. Train a Word2Vec model with enough text means to extract the syntactic structure of a language, e.g. Italian in this practical test, creating a statistical distribution of words. In order to perform a sentiment analysis on reviews which have a maximum length of 30 words it would probably be a good choice to train word embeddings on a great number of short phrases, like posts from Twitter rather than on a collection of books, because of the

similarity of the statistical distribution of words. This hypothesis will be tested in further experimentations. The structure of a skip-gram neural network to represent a vocabulary of 10'000 of words in a vector space of 300 dimensions is described by figure 19. The input layer is composed by so many neurons as is the vocabulary size, because every word in input is a one-hot encoded vector. One input will activate just one neuron in the input layer, which will be automatically activate so many connections as is the number of dimensions desired for the vector space. These connections bring to a single hidden layer, which goal is in fact to project every word in the vector space.

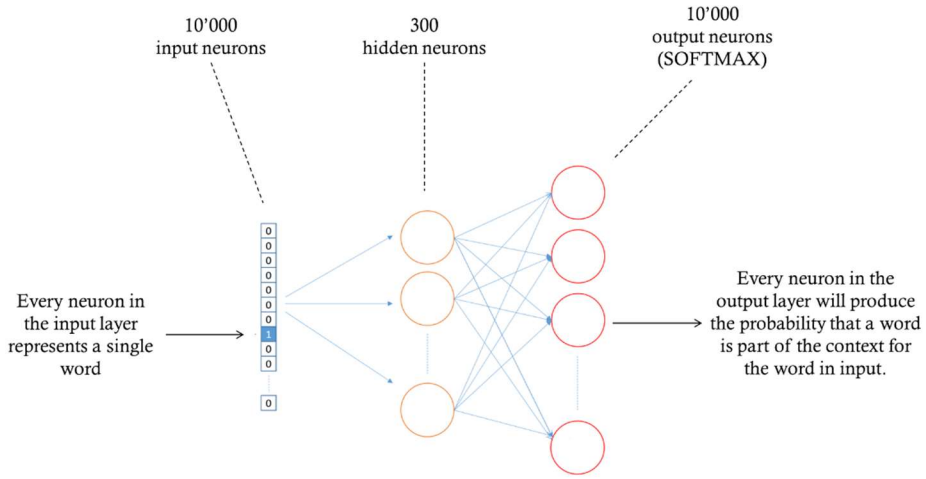


Figure 19 - Structure of Word2Vec skip-gram model for a vector space of 300 dimensions and a vocabulary of 10'000 words

The weights of the connections from the input layer to the hidden one represents the coordinates of every single word in the final vector space, as shown in figure 20. Train the network means to update these weights, a process which results in moving the words within the vector space. Once the model is trained, the matrix of weights from the input layer to the hidden one contains all word vectors. This is the part of the network that can be extracted, saved and reused in other networks, even deep ones. The

structure of the skip-gram model ends with a SoftMax layer as output, which contains so many neurons as is the vocabulary size, so each neuron corresponds again to a unique word.

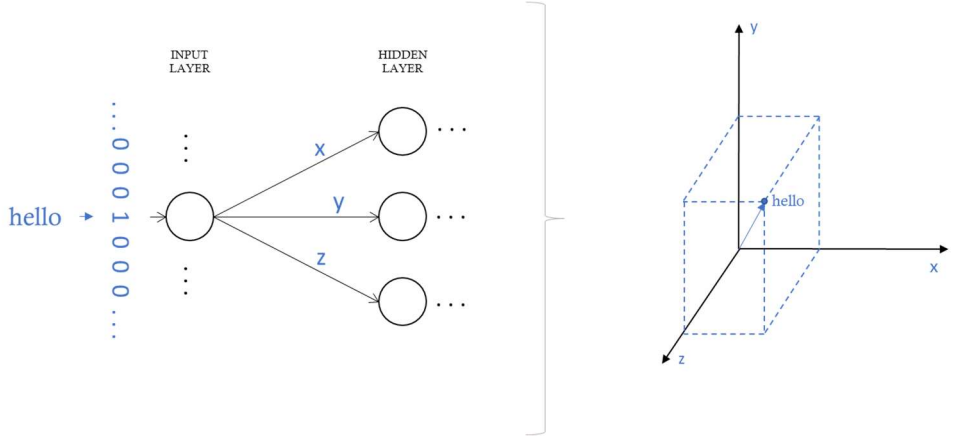


Figure 20 – Graphical representation of a three dimensions vector space created by the weights from input to hidden layer in a Word2vec skip-gram model

The output is a probability distribution in which each value represents the probability of a single word to randomly be part of the context of the input word. When the word vector for a single word reach the hidden layer, then gets fed to the output layer. Each output neuron has a weight vector which it multiplies against the word vector from the hidden layer, then it applies the exponential function exp(x) to the result. Finally, in order to get the outputs to sum up to 1, the result is divided by the sum of the results from all output nodes [9]. The process is summed up in figure 21. If two different words have similar contexts, then the model needs to output very similar results for these two words [9]. The only way for the network to output similar context predictions for two different words is if the word vectors are similar, so the weights that produce the output are similar. If two different words continuously appears in similar contexts, backpropagation will adjust the network weights to learn similar word vectors for these two words. Other standard models to create word embeddings exists, for example GloVe by Socher's research group at Stanford Global Vectors,

which is one of the best in terms of accuracy, sometimes even better than Word2Vec [10], or many custom models have been developed each of which is often dedicated to a particular NLP task, such as the one developed by D. Tang et al. for the Sentiment analysis [11].

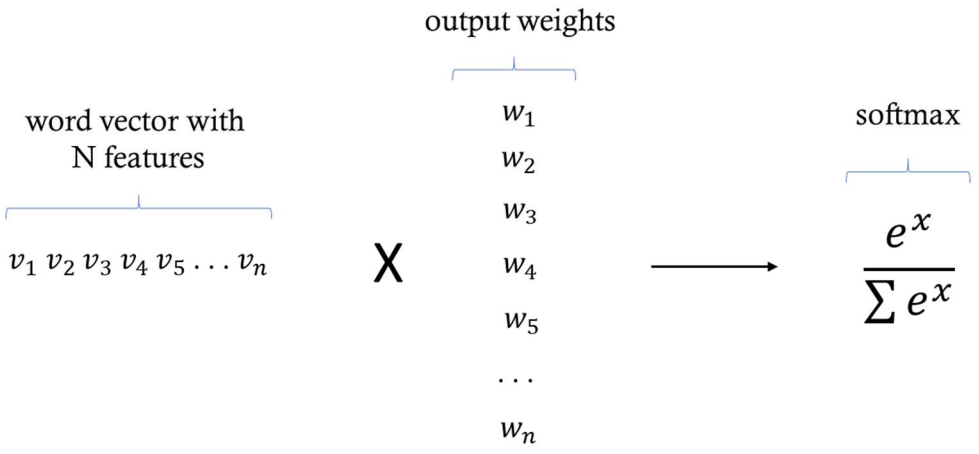


Figure 21 - Mathematical computation of SoftMax output in a Word2Vec skip-gram model

In this work only Word2Vec models will be used, while CBOW is not described in detail because the structure of the model is very similar to the skip-gram one, being the same task, but reversed.

2.2 Practical Tests

In this practical test, a simple classification task on 2789 reviews will be performed. The amount of available text could seem not to be enough, especially for a Deep Neural Network. The focus of the experiment will be on the presence of a pre-trained word embeddings layer in the network. The idea is to understand if such a layer, pre-trained on a huge amount of text, can boost performances of a classifier, which can't be train on a lot of data, so the amount of reviews is good for this purpose. Koppel and Schler, in 2006, showed that in every polarity problem, three categories must be identified: positive, negative and neutral. The introduction of the neutral

category can even improve the overall accuracy of a classifier [12]. However, because of this test is the first in this field for the author, it was decided to maintain the classification problem as a binary one, in order to make it so simple as possible.

2.2.1 Data preparation

Reviews from the chatbot service were completely unlabeled, so how to manually label 2789 phrases as positive or negative, accurately? The goal is to create an Extract, Transform and Load process (ETL) to generate a dataset that can be used to train a neural network, make it learn how to recognize positive and negative sentiment in Italian sentences. In order to file off the bias of a single person respect the actual positivity/negativity of a sentence, each review was first manually labeled with 0 (negative) or 1 (positive), by three different people. This is the only manual action performed in the entire process, which produced three different files containing different sets of labeled reviews. Each person has labeled approximately the first 1300 reviews out of 2789, however the generated sets were not equal. A review was labeled only if the person was completely sure about its value, so each comment can differ not only for the value assigned, but it could be missing, as shown in figure 22. The ETL objectives are to select all labeled comments in the sets and then merge them into one unique set with average values as final labels. The entire process was implemented with a lightweight ETL framework for Python 3.5+, called Bonobo [13]. This framework helps a programmer in creating computational graphs to process data as streams of independent rows. The custom data flow defined is composed by three sub-jobs (one for each set of reviews), each of which selects all reviews with a label, clean the text deleting punctuation, backslashes, numbers and others useless characters.

SET ONE:	SET TWO:	SET THREE:
review one → 0	review one → 0	review one → 0
review two → 0	review two → 0	review two → n/a
review three → 0	review three → 0	review three → 0
review four→ 1	review four→ 1	review four→ 1
review five→ n/a	review five→ 0	review five→ 1
review six → 1	review six → 1	review six → 1
review seven→ 0	review seven→ 0	review seven→ 0
. . .	. . .	. . .

Figure 22 - Differences between three sets of reviews labeled by three different people.

The main job merges the three results of the sub-jobs, cleaned and selected sets, filtering out all reviews that didn't present a label in all three sets and computing the average value for each label, reducing the bias of individual people. After the execution of this part of the ETL process, a dataset containing 1015 cleaned and labeled reviews is ready to be used. The next task consists in prepare the dataset to perform a k-fold cross validation. In Machine Learning, it is a good practice to divide a dataset in two parts, a training set and a test set. The second one is used to test network on data that has never seen before. K-fold cross validation is one of the best practices to understand if a model performs well on new data. The idea is to initially split the entire dataset into k sub-datasets and train the model k times, using a different sub-dataset as the test set each time, as shown in figure 23 with k equals to 4. The performance of the model is then an approximation of the k models performances (P_i) trained and tested on different combinations of the dataset:

$$P_{tot} = \frac{1}{4} \sum_{i=1}^{4} P_i$$

In subsequent experiments, this k-fold divided dataset is used to test different neural networks. More of that, each network will be trained with and without a pre-trained embedding layer. The goal is to understand which one of the tested networks is the best and if the pre-trained embedding layer can have a good influence on performance, especially with a small amount of training data. In these experiments, the dataset is divided in 10 sub-datasets, so the training sets contains approximately 900 samples each time, while the test sets contain approximately 100 samples.

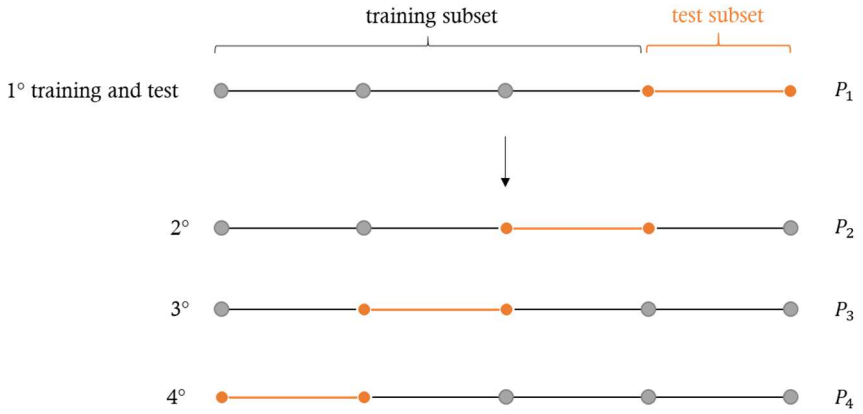


Figure 23 - K-fold cross validation

The last step is to correctly encode the reviews, so that they could be fed into a network. All sentences must have the same length, so in order to understand which is the right one to better represent all, a little investigation have been performed, calculating the maximum and average length in the dataset. As described in figure 24, the lengths were quite unbalanced with a max value of 186 words against an average value of 13,42. More of that, plotting values in a Histogram, it can be observed that a great number of reviews contains less than 30 words. For this reason, reviews with more than 30 words have been truncated while reviews with less than 30 words have been filled with a padding value. All words have been also encoded, mapping every single word to a numerical value.

Encoding operations have been implemented using a Machine Learning framework called Tensorflow, which is an end-to-end open source platform containing a comprehensive, flexible ecosystem of tools, libraries and supported by community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications [14]. The map containing the encoded words is implemented with an object called Tokenizer that can be exported in a *.pickle* file. The tokenizer object is created and saved just one time during the ETL process because all models used in subsequent experiments must share the same vocabulary and encoding.

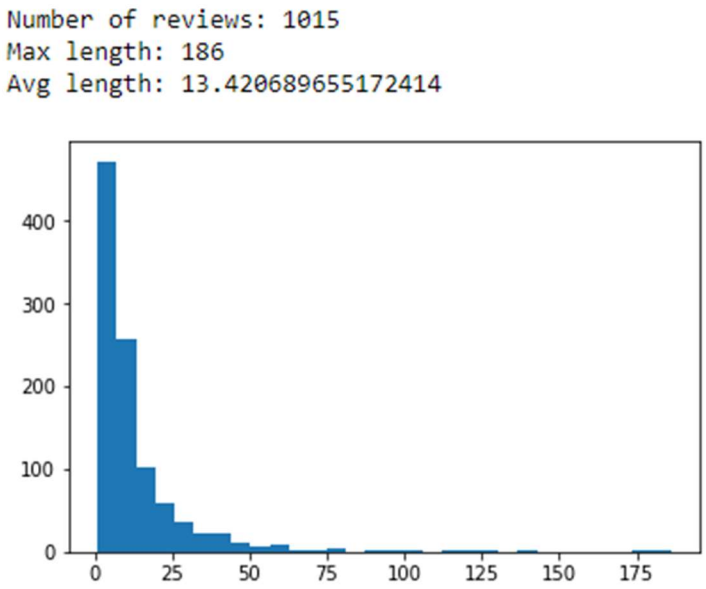


Figure 24 - Histogram about chatbot reviews length

2.2.1 Experiment One: Pre-trained Embeddings in NNs

The goal of this first experiment is to compare the effects of word embedding layers, pre-training them on different kind of texts with different Word2Vec algorithms. Naturally, a model without pre-training is tested too. The chosen model is the simplest possible: Neural Network with an embedding layer and single output node. It was implemented with the

Keras library, which is included in Tensorflow. It allows to compose a network exploiting ready out of the box layers. Here the code used to create the model for the experiment:

```
# define neural network
model = Sequential()
model.add(Embedding(vocabSize, EMBEDDING_DIM, input_length=PAD_LEN))
model.add(Flatten())
model.add(Dense(1, activation='ReLU'))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
```

A Sequential model is initialized, creating a linear stack of layers, initially empty. Then all desired layers can be added using the $add()$ function. First layer added is the Embedding one, which is defined passing as parameters the dimension of the vocabulary (dimension of a single input word), the number of neurons in the layer (number of dimensions in the vector space) and the constant length of input sequences. The second is a layer named Flatten, which has the scope of flatten the 2D output matrix of the embedding layer to a 1D vector. This must be done in order to connect the last layer, called Dense, which is one regular fully connected layer. It contains just a single output neuron because the training targets are values between 0 and 1, the positivity scores created during the ETL process. The activation function of the output node is the ReLU, $f(x) = max(0, x)$, one of the most used. It improves neural networks by speeding up training. Once the network is complete, the $compile()$ function is invoked to configure the network, passing three important arguments, the optimizer, which specifies the training procedure, the loss function, which will be minimized during optimization, and metrics to be calculated. The chosen optimizer is Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments [15]. It maintains a per-parameter learning rate,

which improves performance on problems with sparse gradients (e.g. NLP), and it also adapts these learning rates based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). Adam is a combination of two other important algorithms, AdaGrad and RMSProp. It is very popular in Deep Learning because it achieves good results fast. The second parameter, the loss function, it is filled with the binary cross-entropy, which is perfect for the binary classification problem performed. The only calculated metric is the accuracy, which indicates the percentage of misclassification. One important aspect about implementing a Neural Network using Keras is that it is simple to change the weights matrix of the Embedding layer:

```
# define neural network
model = keras.models.Sequential()
model.add(keras.layers.Embedding(
       vocabSize,
       128,
       weights=[embeddingMatrix],
       input_length=PAD_LEN,
       trainable=True
))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(1, activation='relu'))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
```

The only difference between the first model and the second is a parameter passed during the definition of the layer, `weights=[embeddingMatrix]`, where the variable `embeddingMatrix` contains a NumPy matrix. It was changed more than once during the experiments using different types of pre-trained word embeddings, generated using Word2Vec models. The network was tested with four different initializations of the embedding layer. The first one is a random initialization, making the network learn the word representations itself, during the training on 1015 labeled reviews. The dimension of the layer has been incremented gradually from 8 neurons to 300 to observe the

influence of the vector space size. The model obtained pretty good results with each dimension because of the ease of the task, however, as can be observed in figure 25, the best overall accuracy is produced by the model with 64 neurons in the embedding layer, but the real increase takes place from 8 to 16, after which the value remains stable at around 86%.

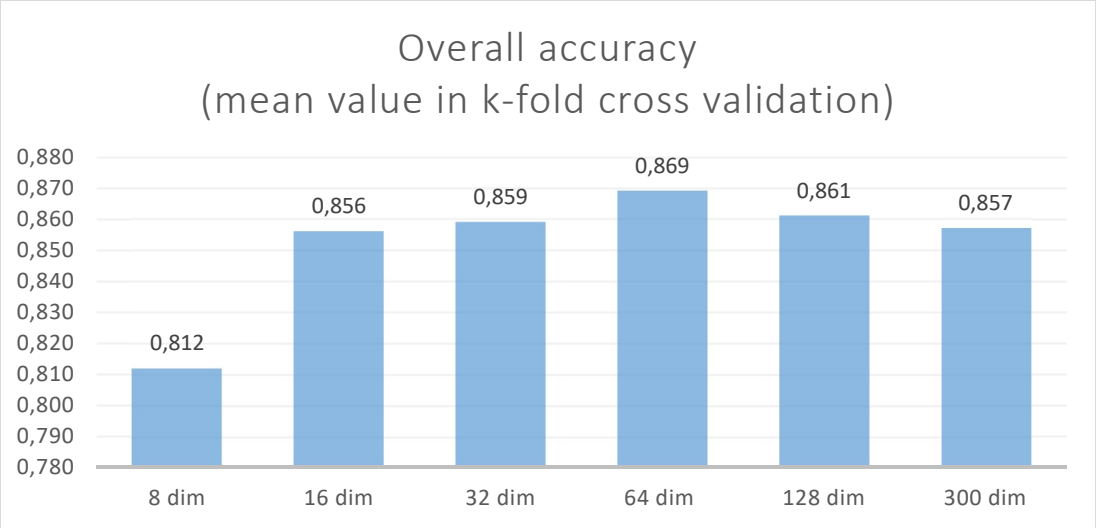


Figure 25 - Histogram about the influence of the embedding layer dimension on the sentiment analysis overall accuracy

The second test is pre-train the embedding layer weights on the same reviews, instead of making the networks learn the word representation itself, a skip-gram Word2Vec model have been trained on all 2789 (because it can be trained even on unlabeled text). Figure 26 shows a 2D distribution of 100 calculated vectors. The distances between words do not seem to be well calculated. For example, the word “telefonica”, which is an adjective, is closer than “hai” to “avete”, but these last two words are very similar verbs that appears in similar contexts for sure. Probably the amount of text used to train the Word2Vec model is not enough. Pre-trained word vectors with 64 dimensions have been compared with the random initialization, results can be observed in figure 27. Besides the overall accuracy, other

metrics have been calculated, to better show how the insertion of the pre-trained layer has clearly worsened the performances.

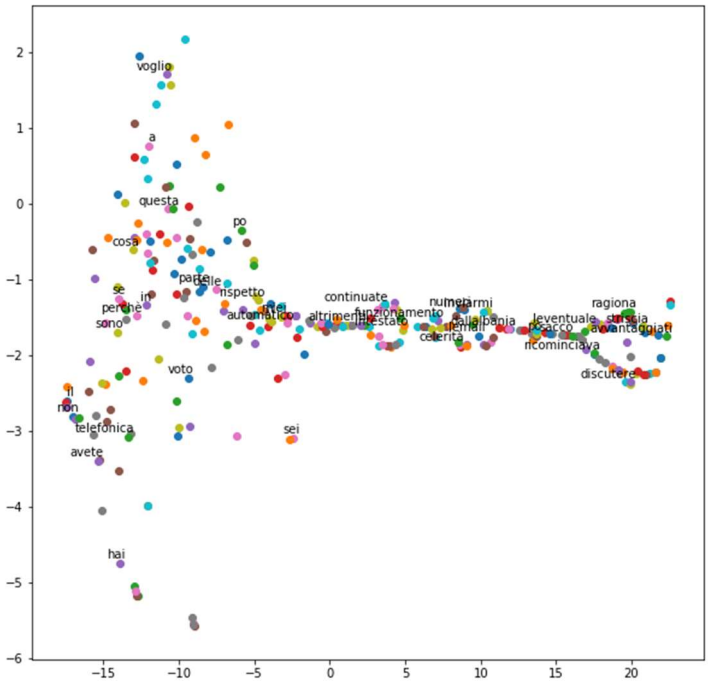


Figure 26 - 2D representation of word embeddings with 64 dimensions, obtained with Principal Components Analysis - PCA

These new metrics are calculated for single classes. Precision answers to the question, "What proportion of identifications was actually correct?" while Recall answers to, "What proportion of identifications was identified correctly?" [6]:

$$Precision = \frac{true\ positives}{true\ positives + fals\ \ positives} \qquad Recall = \frac{true\ positives}{true\ positives + fal\ \ \ negatives}$$

A third metric, the f1 score The F1 score, is the harmonic average of the precision and recall:

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

If F1 score is high, the model has good results in precision and recall, on the contrary if it is low it indicates that the model has problems with false

positives or false negatives. F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0.

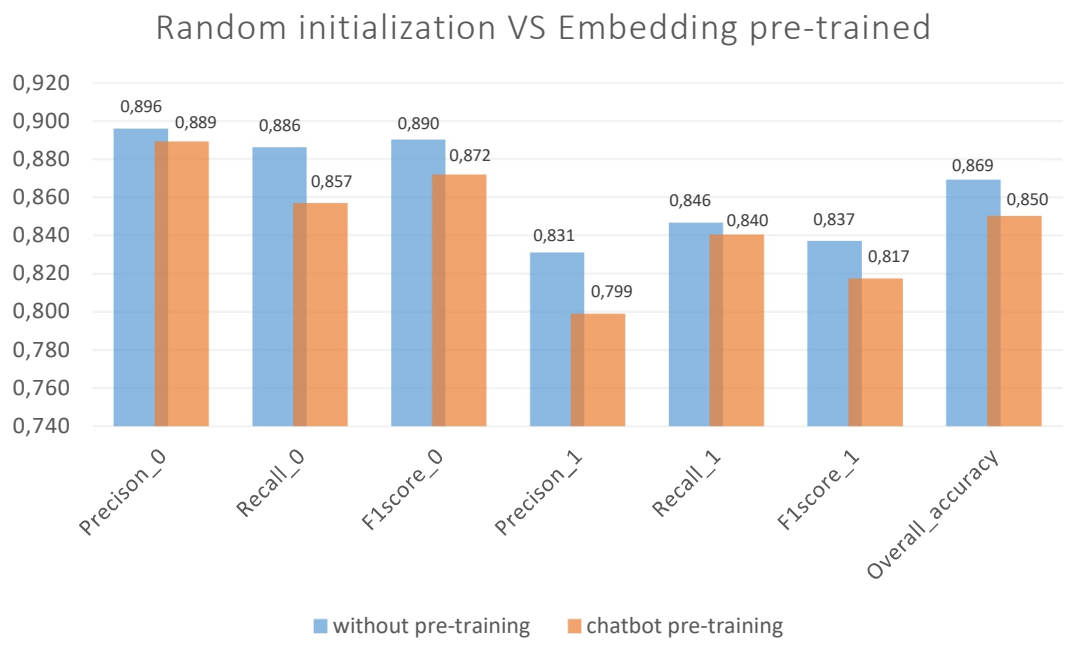


Figure 27 – NN performances comparison between a random initialized embedding layer and a pre-trained one, on a low number of chatbot reviews (2789)

Results suggests that 2789 chatbot reviews are probably not enough to create good word vectors with a Word2Vec model, so another two pre-trained layers have been tested. First weights come from a project of the ItaliaNLP Lab at the Istituto di Linguistica Computazionale "Antonio Zampolli" (ILC-CNR). The aim of the project was to train word embeddings on a collection of 46.935.207 italian tweets, using CBOW Word2Vec model. Word vectors have been released as an SQLite database, only for research purposes. The last word vectors have been retrieved from a project of the Human Language Technologies (HLT), Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", Consiglio Nazionale delle Ricerche - Pisa, Italy. The aim of the project was to train Italian word embeddings with two different models (SKIP-GRAM Word2Vec and Glove) on a text corpus from

the Italian Wikipedia [16]. Twitter word embeddings have 128 dimensions while Wikipedia embeddings have 300, so they have been compared with randomly initialized layers of the same dimension. As can be observed in figure 28 and figure 29, neither of them led to improvements in the network's performance, indeed as in the previous test, performance worsened. Probably low complexity of the task is not a good test field.

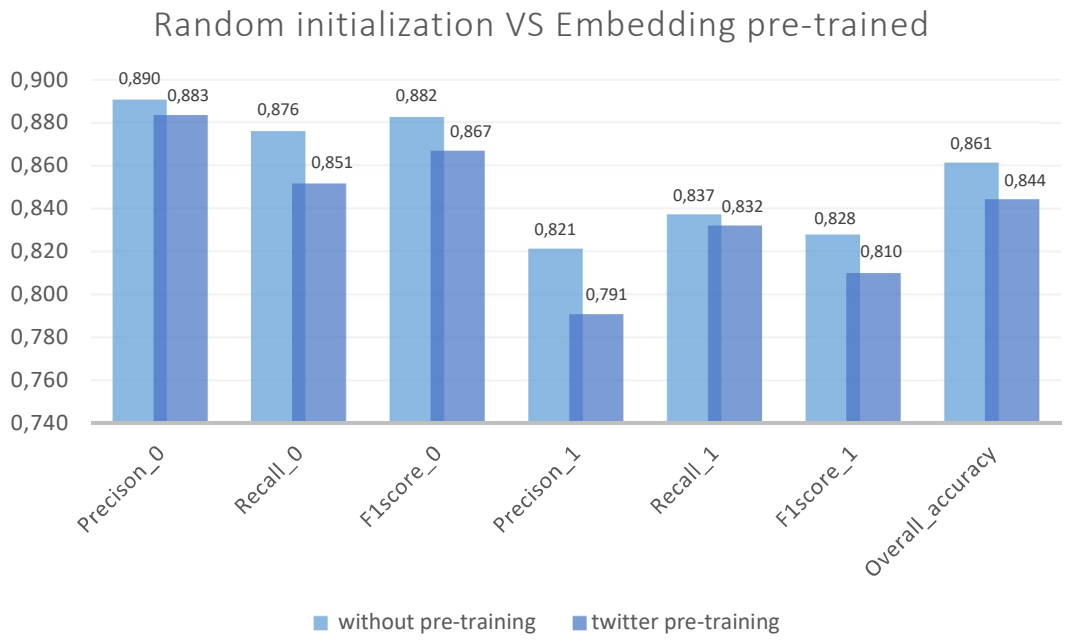


Figure 28 -NN performances comparison between a random initialized embedding layer and a pre-trained one on big Twitter corpus

Nor the results obtained from the application of CBOW vectors, nor the results obtained with the SKIP-GRAM ones have improved the performance of the NN. Vectors obtained from twits should be good word features for this task, because they have been created on short sentences, like the chatbot reviews. On the contrary, Wikipedia embeddings have been trained on various kind of texts. However, all models achieved good results and the variations in performance are not very significant, in any case. Pre-trained word embeddings have not a good influence in NNs for simple NLP

tasks, so the next steps will be to test these layers in more complex models and more complex tasks.

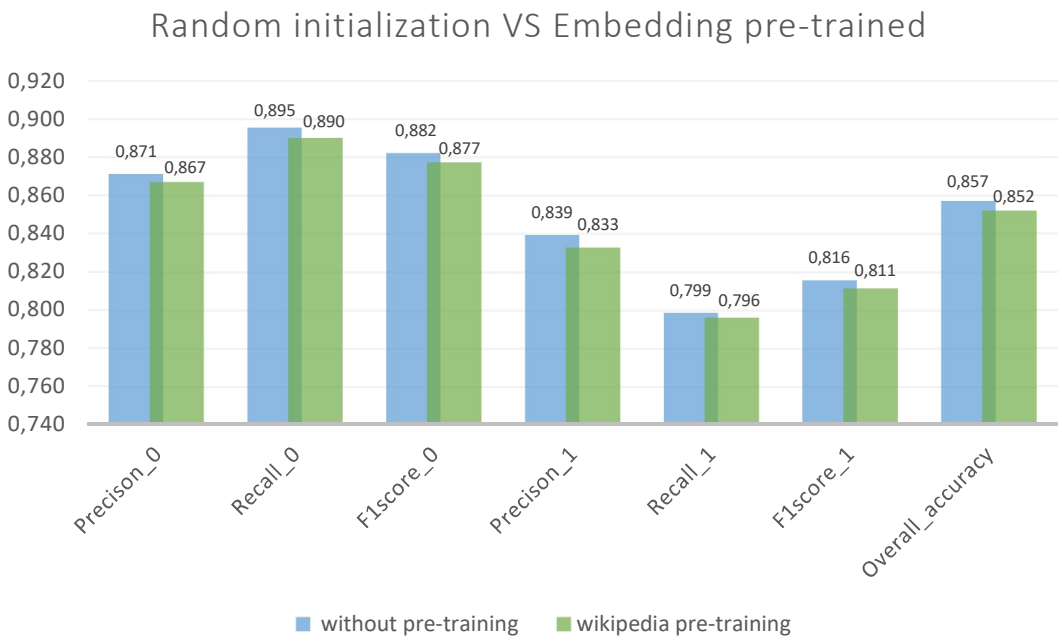


Figure 29 – NN performances comparison between a random initialized embedding layer and a pre-trained one on big Wikipedia corpus

2.2.2 Experiment Two: Pre-trained Embeddings in CNNs

The purpose of this experimentation is to find out whether a convolutional neural network can improve its performance by applying a layer of pre-trained word embeddings. The idea is to re-propose the task of Experiment One, but making the network evolve in a deep model. Once again, the network has been implemented using the Keras library:

```
# define neural network
model = Sequential()
model.add(Embedding(vocabSize, EMBEDDING_DIM, input_length=PAD_LEN))
model.add(Conv1D(128, 2, activation='relu'))
model.add(GlobalMaxPooling1D())
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
```

The architecture, shown is figure 30, is taken from Y. Kim work on sentence classifications [17]. The input layer is a sentence comprised of word embeddings. That's followed by a convolutional layer with multiple filters, then a max-pooling layer, and finally the classifier neuron. The convolution operation, which is applied to a window of 2 words to produce a new feature, is applied to each possible window of words in the sentence to produce a feature map. Then the max-overtime pooling operation over the feature map extracts the maximum value as the feature for each filter.

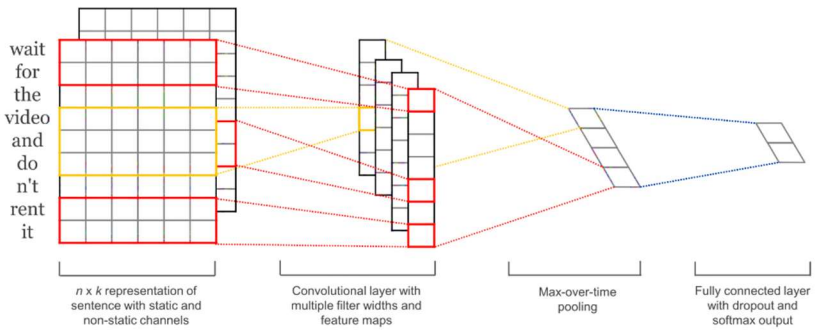


Figure 30 - - CNN architecture from Y. Kim work on sentence classification [17]

The network captures the most important feature for each feature map. Metrics, optimizer and loss function remain the same of the previous network. Again, each model is tested applying the k-cross validation technique, training and testing each model ten times on different combinations of the original dataset. All metrics showed in the graphs are an average value of the ten iterations. The three types of pre-trained embeddings (chatbot reviews, Twitter corpus, Wikipedia corpus) are again compared with randomly initialized layers of the same size (64, 128, 300). As shown in figure 31, vectors created with SKIP-GRAM algorithm and chatbot reviews text, do not improve the CNN performance. While recall of class 0 and precision of class 1 have been increased, precision of class 0 and recall of 1 have been decreased. The network has not improved its ability to avoid misclassifications and the overall accuracy almost remains at the same value (0,857 vs 0,856).

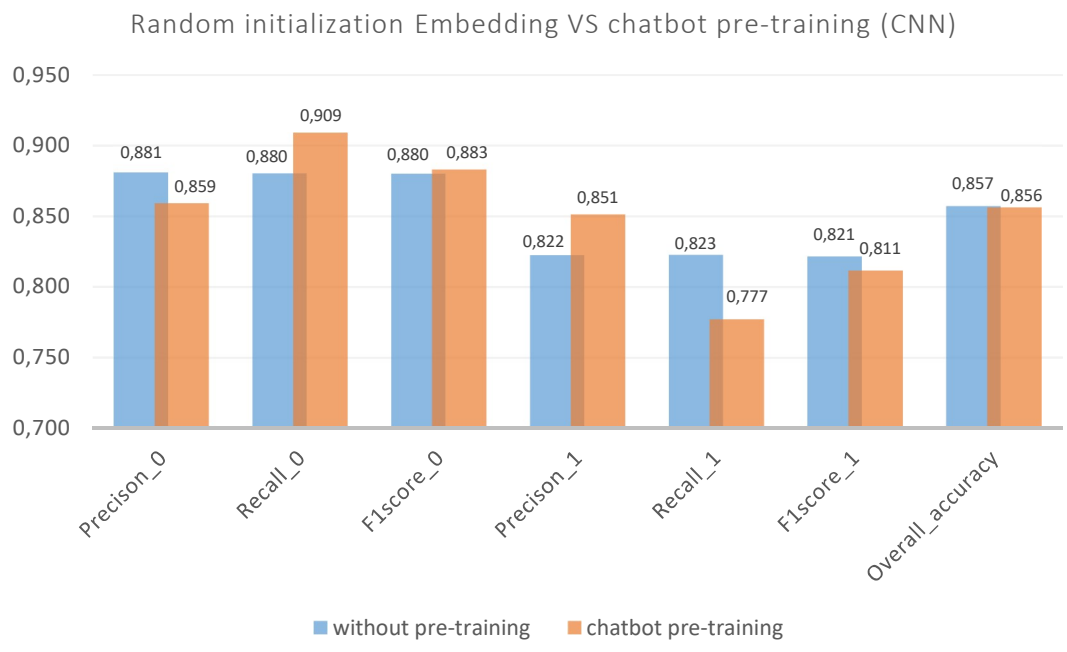


Figure 31 - CNN performances comparison between a random initialized embedding layer and pre-trained one on 2789 chatbot reviews

CNN performance has not been improved, but it has not even deteriorated as in the case of the simple neural network. The incorporation of Twitter and Wikipedia pre-trained layers led to an improvement in the CNN performance. As show in figure 32 and figure 33, in both cases the values of all the metrics have been improved, so the CNN enhanced its ability to avoid misclassifications for each class and the overall accuracy. Surprisingly the best results until now have been achieved with the Wikipedia pre-trained layer, despite the Twitter dataset is composed by short sentences like the reviews. Moreover, the twitter vectors set is missing only 379 words of reviews vocabulary (4617 words), against the 780 missing words for the Wikipedia vectors set. The last difference between Twitter vectors and Wikipedia ones is the model used to generate them, respectively SKIP-GRAM and CBOW. So, this result suggests that the SKIP-GRAM algorithm could be a better choice for create word features to be used in sentence classification tasks.

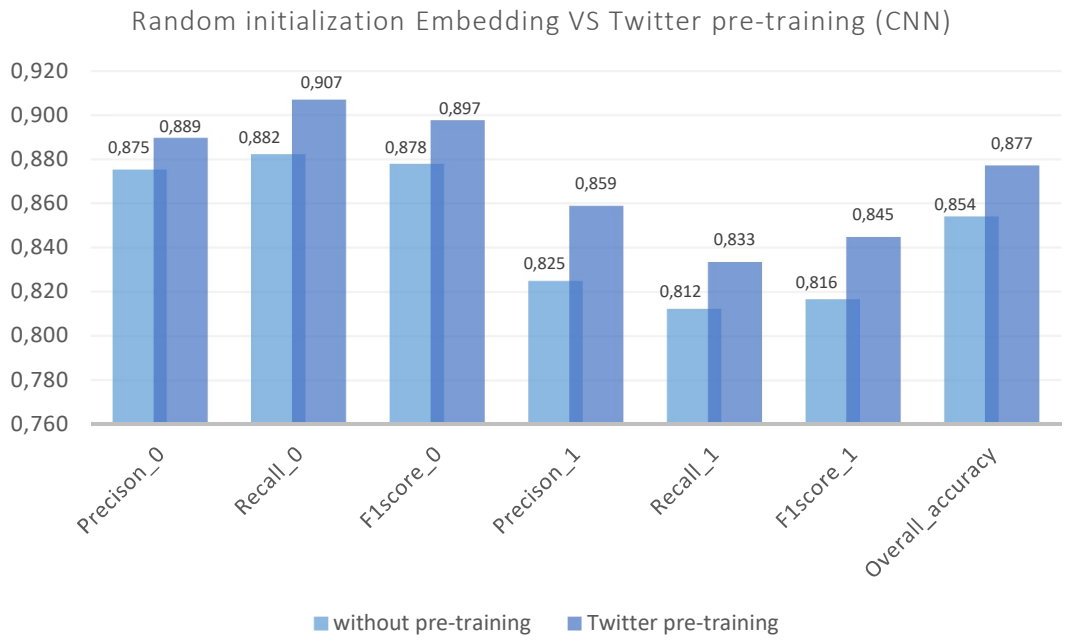


Figure 32- CNN performances comparison between a random initialized embedding layer and pre-trained one on big Twitter corpuss

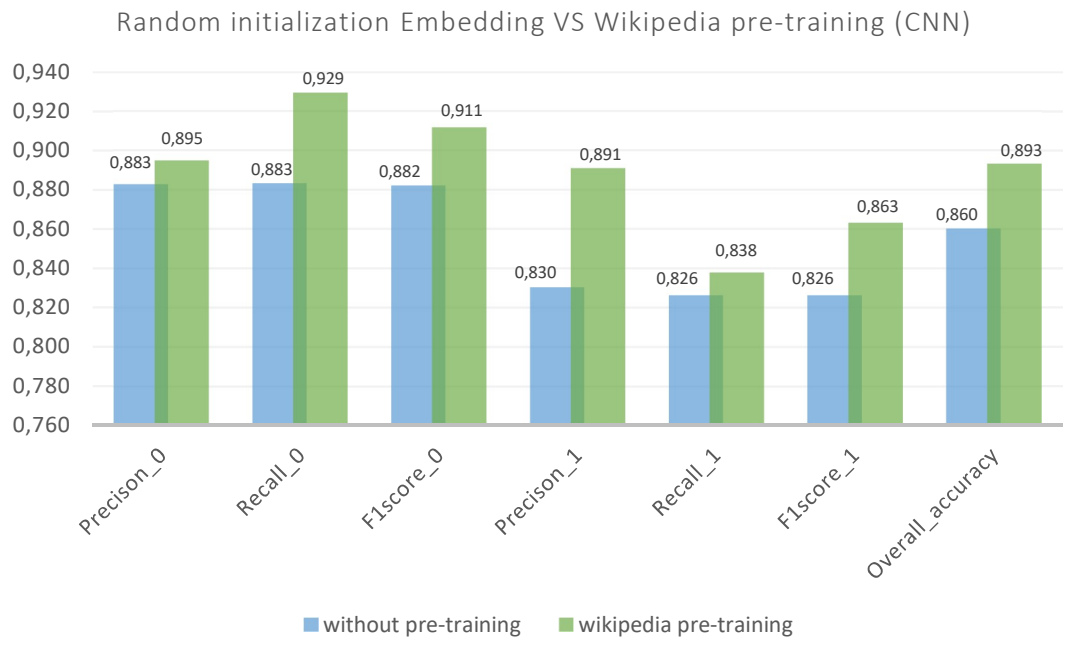


Figure 33 - CNN performances comparison between a random initialized embedding layer and pre-trained one on a big Wikipedia corpus

2.3 Results discussion

The incorporations of pre-trained word vectors have led to performance improvements in CNNs, but not in simple NNs. The convolutions applied to 2 words at a time seems to take advantage from the pre-training of word embeddings coming from the previous layer. Word2Vec models need to be trained on a huge amount of text in order to produce good features. Vectors trained on 2789 reviews with a skip-gram algorithm did not produce improvements in any case and the distances between word vectors weren't optimized at all. The sentiment classification task that has been performed is probably too simple to observe significant differences between randomly initialized layers and pre-trained ones. Metrics have always varied no more than 5-10% and all models, with and without pre-trained layers, always achieved an overall accuracy between 80% and 90%. However, this

simplicity of the task also allowed to gain a first experience in the field of Natural Language Processing and Deep Learning, without too much effort. All the arguments in the preliminary study about NLP techniques have been addressed from a practical point of view, representing a first step towards experimentations with more complex models and tasks. Word vectors have been successfully created and incorporated in NNs and CNNs, using the Python programming language and Data Analysis frameworks such as Bonobo and Tensorflow. The natural next step should be to test CNNs in more complex classification tasks, but in next chapters the experience gained from this test will be applied to a new NLP task and to different Deep Learning models.

3. Text Generation: practical tests

Pre-trained word embeddings had a good influence on sentence classification with Convolutional Neural Networks. The goal of this experiment is to find out if these layers can increase performance of other types of DNNs besides CNNs. Deep Neural Networks have achieved excellent performance on a variety of difficult learning tasks related to Natural Language Processing. The evolution trend of Deep Learning models is incredibly fast. For instance, let's consider GPT-2 model by Open-AI, released on February 19th, 2019 [18]. This model has demonstrated that when a big model is trained on a sufficiently large and diverse dataset it is able to perform well across many domains and datasets. The diversity of tasks that GPT-2 is able to perform suggests that high-capacity models trained on varied text corpus begin to learn how to perform a surprising number of tasks without the need for explicit supervision [19]. GPT-2 larger model and training data were not released because of concerns about potential abuse. Indeed, besides achieving state-of-the-art performance on many language modeling benchmarks, and performing rudimentary reading comprehension, machine translation, question answering, and summarization, it can generate realistic text in a variety of styles, from news, articles to fan fiction, based off some seed text. Because of the strong interests in recent GPT-2 results, a part of these last experiments will be dedicated to understanding the GPT-2 model, called the Transformer. The architecture of such a model is an Encoder-Decoder, used to solve general sequence to sequence problems. In 2014, I. Sutskever et al. published a work on Sequence to Sequence Learning in which they found that reversing the order of the words in all source sentences, but not target ones, increased performances in the Long short-term Memory

networks – LSTM, an architecture evolution of RNNs. The first few words in the source language results very close to the first few words in the target language, so the problem's minimal time lag is greatly reduced [20]. The Transformer model was presented in 2017 and it is an encoder-decoder model based solely on attention mechanism, not recurrence, which allow modeling of dependencies without regard to their distance in the input or output sentence. Exploiting the experience gained from previous sentiment analysis tests, text generation will be performed first with LSTM and then with the Transformer model. In doing so, the focus will be always on the role of pre-trained word embeddings, pointing out if they could help getting this task easier.

3.1 LSTMs

RNNs compute multiple derivatives through time applying them to the same parameters at once. This operation is not good for stochastic gradient descent that can explode or vanish. In order to prevent it to explode, the gradient clipping can be applied, which calculate the gradient, shrinking steps when the norm grows too big [4]. Vanishing gradient is more difficult to prevent and it make forget more distant past causing a memory loss in RNNs. One solution to the vanishing gradient problem is a relevant change on the structure of the network that gives life to a specific type of RNN, the LSTM. Let's reconsider the typical unit repeated in the structure of a standard RNN. Each node, like the one shown in figure 34, will have a very simple internal structure like, for example, a single tanh neural network layer. In LSTM one or more units are substituted by a machine, shown in figure 35, which structure is a basic memory cell, with three different gates.

The first gates aim is to decide if the memory can be written, the second one to decide if it can be read while last one to decide if can be cleared.

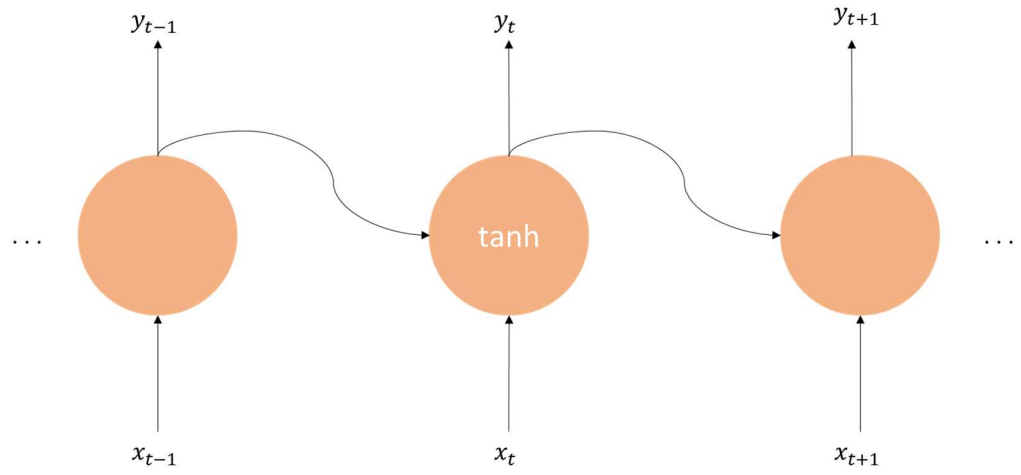


Figure 34 – Representation of typical hidden node of a recurrent neural network, through time.

The gates can be implemented in practice to support continuous decisions, like in Neural Networks, multiplying the value to be written with a value between 0 and 1 so that the memory is partially written or read. The multiplicative factor is a continuous function that also differentiable, meaning that it can backpropagate through the network [4].

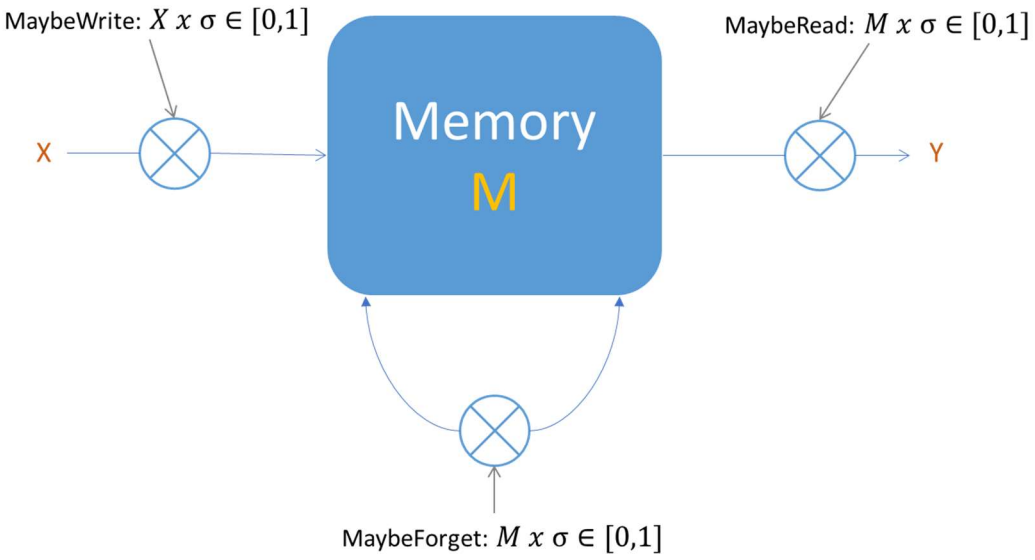


Figure 35 - LSTM unit machine structure

In order to understand how this machine is implemented as a unit of a RNN, it has to be considered that the internal structure of the unit, instead of having one single neural network layer, it has four, interacting and emulating somehow the behavior of the memory cell machine just described. The first step is to decide what information is going to be thrown away from the cell state. This decision is made by a first sigmoid layer, called the "forget gate layer." Referring figure 34, it looks at $y_{t-1}$and x_t , then it outputs a number between 0 and 1 for each number in the cell state $C_t - 1$. Naturally, value 1 means that the information will be completely kept, while a value 0 means to completely forget it. The second step, divided in two parts, is to decide what new information is going to be stored in the cell state. First, a sigmoid layer called the "input gate layer" decides which values will be updated, while a $tanh$ layer creates a vector of new candidate values, $\widetilde{C_t}$, that could be added to the state. Second, the two values are combined to create an update to the state. The third step is to update the old cell state, $C_t - 1$, into the new cell state C_t. The last step is to decide what is the output, which will be based on the cell state just updated but it will be a filtered by another sigmoid function. LSTM are very good neural network architectures to understand short sequences of text or to generate them.

3.2 Encoder-Decoder Architecture

In word embeddings each word is encoded with a vector of fixed dimensionality. In the experiment one, Twitter embeddings were encoded with 128 dimensions while Wikipedia ones were encoded with 300. The NN received in input a vector of a fixed dimension of 30 words. The necessity to encode input and output to a pre-decided and fixed

dimensionality is a big problem for tasks like machine translation and text generation. Let's consider a simple text generation task in which if the model receives in input an English sentence it must generate a sequence of words that make sense not only in terms of grammar but also it must be related to the source one. It's impossible to think of fixed number of words in input and output every time, the sequences must vary their length. Deep Neural Networks such as RNNs and DNNs, do not resolve this inconvenience and they can only be applied to problems in which input and output can be encoded with vector of fixed dimensionality too. The Encoder-Decoder architecture is used to solve general sequence to sequence problems. The goal of the LSTM is to estimate the conditional probability $p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T)$ where $x_1, \ldots, x_T$ is an input sequence and $y_1, \ldots, y_{T'}$ is its corresponding output sequence whose length T' may differ from T. The LSTM compute this conditional probability by first obtaining the fixed-dimensional representation v of the input sequence given by the last hidden state of the LSTM itself. Then the probability of the output is computed as follow:

$$p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T) = \prod_{t=1}^{T'} p(y_t | v, y_1, \ldots, y_{t-1})$$

Where $p(y_t | v, y_1, \ldots, y_{t-1})$ distribution is represented with softmax over all the words in the vocabulary. The basic idea behind the Encoder-Decoder architecture is that one LSTM read the input sentence, one timestep at a time, to obtain large fixed-dimensional vector representation, and that another LSTM is used to extract the output sentence from that vector [20]. Simeon Kostadinov gave a short and concise explanation of the sequence to sequence model [21]. As shown in figure 36, The architecture consists of 3 parts: encoder, intermediate (encoder) vector and decoder. The encoder

is a stack of several recurrent units (LSTM cells, for instance) where each accepts a single element of the input sequence, collects information for that element and propagates it forward. The Encoder vector is the final hidden state produced from the encoder part of the model, which aim is to encapsulate the information for all input elements in order to help the decoder make accurate predictions. It acts as the initial hidden state of the decoder part of the model.

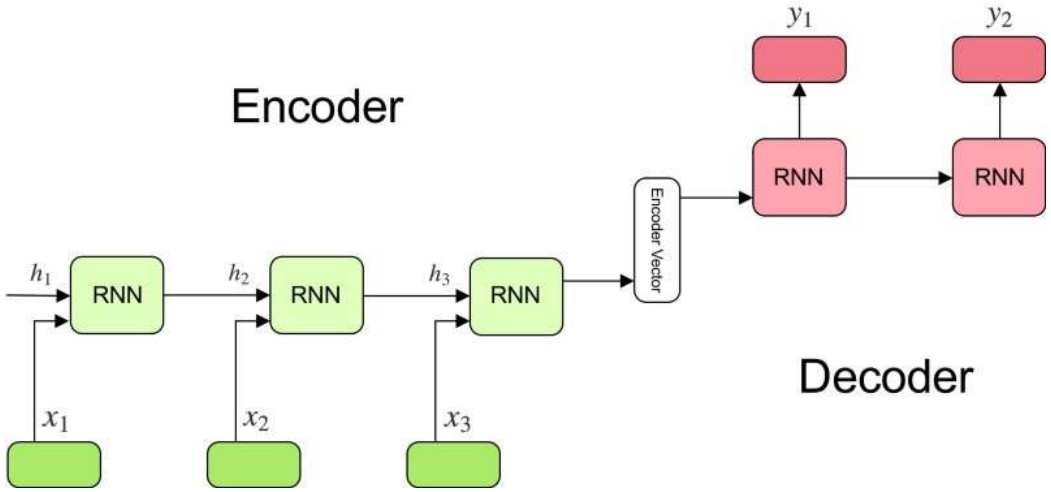


Figure 36 -Graphical representation of the Encoder-Decoder architecture [21]

The Decoder is a stack of several recurrent units where each predicts an output y_t at a time step t. Each recurrent unit accepts a hidden state from the previous unit and produces and output as well as its own hidden state. In text generation problems, the vocabulary usually contains an end phrase token and start phrase token. The output sequence is a collection of all words from the generated sentence, plus the end token. Each word is represented as y_i where i is the order of that word. Each RNN uses the previous hidden state to compute the next one. The encoder decoder architecture can be implemented with different types of deep layers, not only recurrent ones. For example, the Transformer model is an implementation of the Encoder-Decoder architecture with attention layers.

3.3 Attention mechanism

The traditional Encoder-Decoder architecture often encode information that is completely useless to the task it is trying to learn. Thinking of text summarization where a very long sentence could arrive as the input, it is unrealistic to expect a fixed-size vector to encode all needed information, plus each part of the output sequence is highly related to a certain segment of the input one. Indeed, reversing the input sequence usually increase the performances of the Encoder-Decoder model because of the short distance between input and output parts related to each other [20]. The attention mechanism attempts to ease the above problem by allowing the decoder to refer the input sequence. Specifically, starting decoding, in addition to the last hidden state and the generated token, the decoder is also conditioned on a "context" vector calculated on the input hidden state sequence [2]. An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as weighted sum of the values, where the weights assigned to each value is computed by a compatibility function of the query with the corresponding key [22]. The core of Probabilistic Language Model is to assign a probability to a sentence. Due to the nature of sentences that consist of different numbers of words, RNN is naturally introduced to model the conditional probability among words:

$$P(w_1, w_2, w_3, \dots, w_n) \approx \prod_i P(w_i | w_{i-k}, \dots, w_{i-1})$$

Considering an encoder decoder architecture, composed for instance by LSTM cells sequentially. As shown in paragraph 3.2, encoder vector incapsulate the information for all input elements. This vector is passed as a single hidden state to the decoder and it is produced by the last LSTM cell

in the Encoder. Attention mechanism try to answer one simple question: “is one hidden state really enough?”. Like the basic encoder-decoder architecture, it plugs a context vector into the gap between encoder and decoder. According to the schematic shown in figure 37, the context vector takes all encoder cells outputs (not only the last one) as input to compute the probability distribution of source language words for each single word decoder wants to generate (not just the first one). By utilizing attention mechanism, it is possible for decoder to capture global information rather than solely to infer based on one hidden state [23].

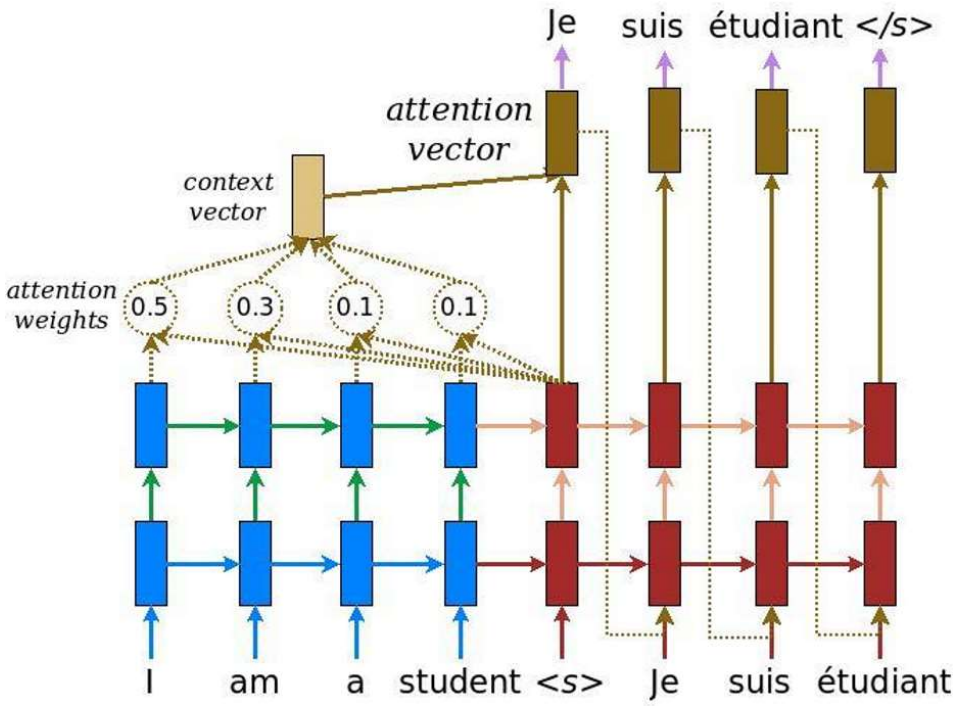


Figure 37 - Graphical representation of the Attention Mechanism in the Encoder-Decoder architecture. The encoder is highlighted in blue while the Decoder in red [23]

To build an attention context vector is simple respect a recurrent computation such as LSTM cells. For a fixed target word, first, it loops over all encoders’ states to compare target and source states, generating scores for each state in encoders. Then a SoftMax layer, for instance, normalizes all scores, generating the probability distribution conditioned on target

states. At last, the weights are introduced to make context vector easy to train. The math behind these computations can be understood keeping two key points in mind [23]:

- During decoding, context vectors are computed for every output word. So, a 2D matrix whose size is the number of target words multiplied by the number of source words will be calculated each time. The equation,

$$\alpha_{ts} = \frac{\exp\left(score(h_t, \overline{h_s}\right)}{\sum_{s'}^{S} \exp\left(score(h_t, \overline{h_{s'}}\right)}$$

  demonstrates how to compute a single value given one target word and a set of source word.
- Once context vector is computed,

$$c_t = \sum_{s} \alpha_{ts} \cdot \overline{h_s}$$

  attention vector could be computed by context vector, target word, and attention function f (tanh for instance),

$$a_t = f(c_t, h_t) = \tanh\left(W_c[c_t;\ h_t]\right)$$

3.4 The Transformer

The Transformer model has an Encoder-Decoder architecture in which recurrence and/or convolutions are substituted by attention. Global dependencies between input and output are entirely managed by attention mechanism described in paragraph 3.3. The architecture, shown in figure 38, is designed by stacked self-attention, fully connected layers for both the encoder and decoder. V. Ashish et al. Stacked six encoders and six decoders in the original paper, but this is only a demonstration number that can be

changed according to the task. In [23], a fair explanation of the model structure can be found. A resume containing key information will be reported in this work.

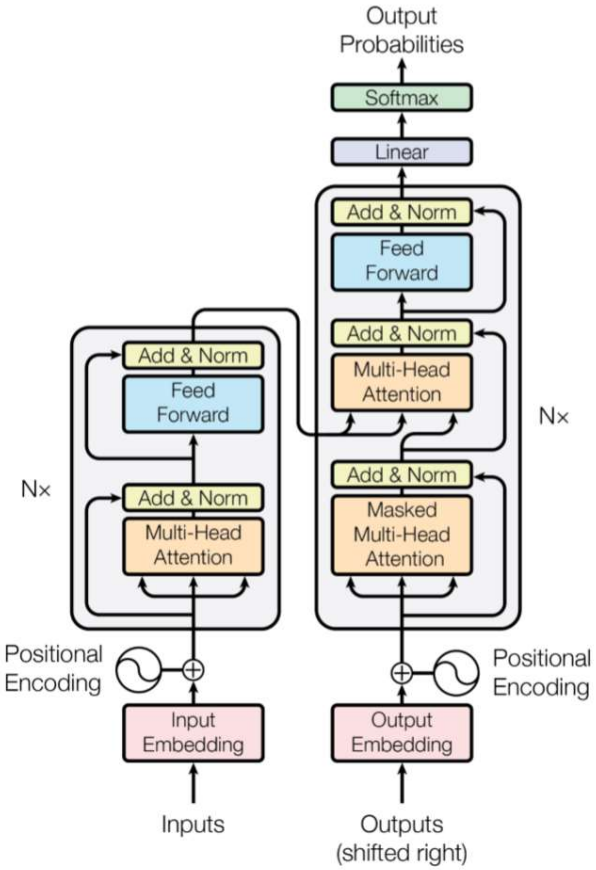


Figure 38 - Transformer model architecture [22], on the left, the Encoder while on the right, the Decoder

The encoders are all identical in structure, but they do not share weights. Each one is composed by two sub-layers, multi-headed attention and feedforward. The encoder's inputs first flow through a multi-headed attention layer, an improvement of a simple self-attention layer which helps the model to process each word, allowing it to look at other positions in the input sequence to obtain a better encoding of that word, as shown in figure 39. While self-attention makes use of a single representation subspace, multi-headed layers use multiple. The outputs are fed to a feed-

forward neural networks. The exact same feed-forward layer is independently applied to each position. Before the first encoder, an Embedding layer transforms input word into vectors. After embedding the words in the input sequence, each of them flows through each of the two sub-layers of the encoder. One key property of the Transformer is that the word in each position flows through its own path in the encoder. There are dependencies between these paths in the multi-headed attention layer. The feed-forward layer does not have those dependencies and can execute the various paths in parallel. For this reason, the computation of such a model is faster than RNNs. Figure 40 describes the flowing of words into the first encoder of the model. To conclude, in order to account for the order of the words in the input sequence the transformer adds a vector to each input embedding. These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence.

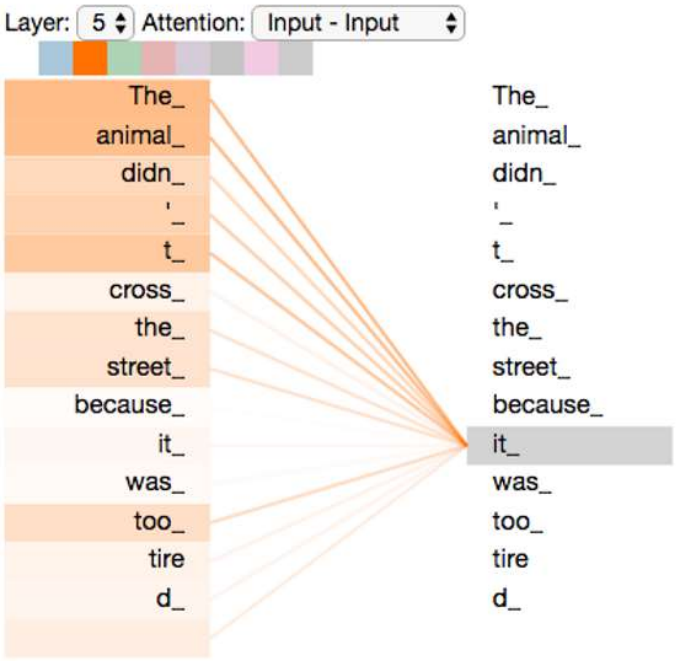


Figure 39 – Graphical representation of the self-attention mechanism [23]

.

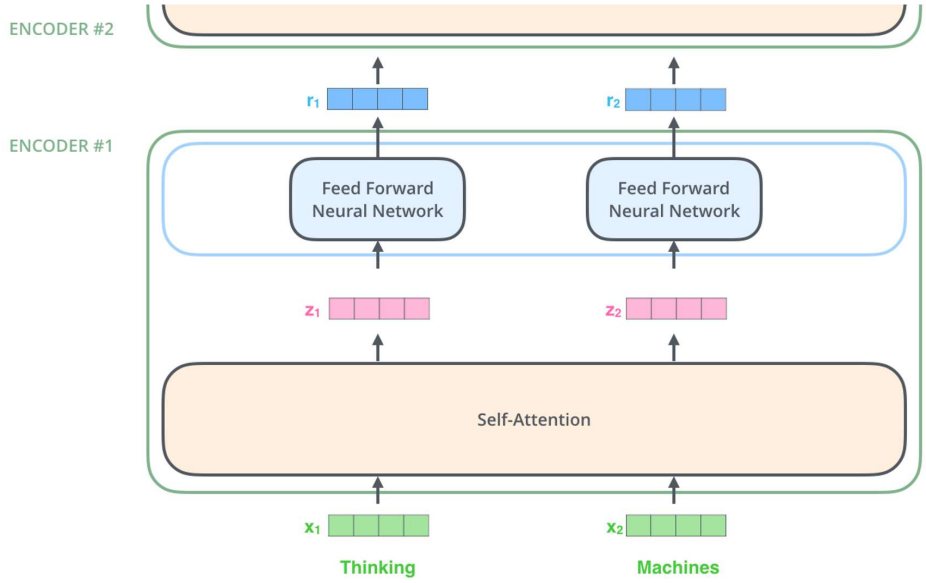


Figure 40 – In the Transformer, the word at each position passes through a self-encoding process. Then, they each pass through a feed-forward neural network (the exact same network) with each vector flowing through it separately [23]

The decoder-side basically works how the components of encoders work as well, having both those sub-layers, plus an "encoder-decoder" attention layer between them, as shown in figure 41. Encoder starts by processing the input sequence, then output of the top encoder is transformed into a set of attention vectors. These are used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence.

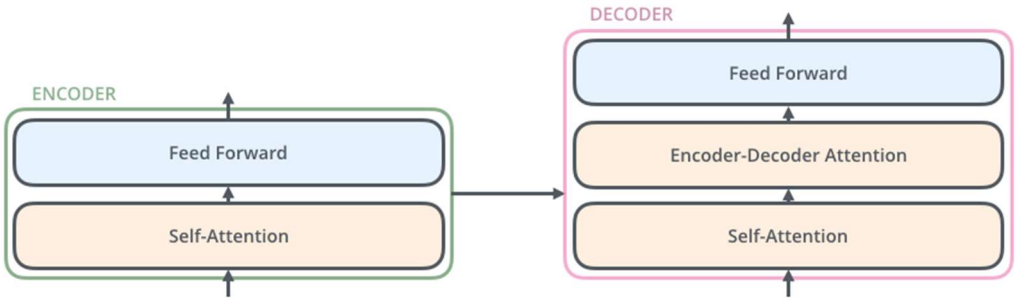


Figure 41 - Encoder structure vs. Decoder structure in the Transformer [23]

The process repeats these steps until a special symbol is reached indicating the transformer decoder has completed its output. The output of each step is fed to the bottom decoder in the next time step. Like for the encoder inputs, decoder inputs are embedded and added with positional encoding to indicate the position of each word.

3.5 Practical Tests

The Transformer model is more complex than the description given in paragraph 3.4, but the key features explained can help in understanding substantial differences respect CNNS and RNNs. The transformer is based on attention mechanism between encoder and decoder, and self-attention layers in each encoder/decoder, so the focus is entirely on the relations of words in input. Convolutions summarizes and captures the most important feature for each input. Recurrence lets the network "remember" a certain number of previous inputs, influencing actual decisions. In further experiments, the focus will be again on NLP and word embeddings. Making use of the experience of previous tests in sentiment analysis, the influence of pre-trained word vectors will be investigated in RNNs and Transformer applied to text generation. Again, the tests are organized with increasing complexity, from a character-based generation with LSTM to a word based-generation using Encoder-Decoder architectures. First, a dataset containing fables will be tested because of the ease of the language. Each fable follows a simple narration pattern often containing animals. They were created to teach moral concepts to children so, the syntax of the sentences is very simple. For sure, fables are a good choice to build toy examples of text generation. In the end, more complex datasets such as international news and sci-fi books are also tested.

3.5.1 Experiment One: LSTM

The first experiment is based on the simplest example, a NN with 1024 LSTM cells, performing a char-based generation, which implementation is inspired to a Tensorflow official tutorial [24]. The network is composed by three layers:

- Embedding
- LSTM
- Dense layer (with vocabulary size dimensionality)

The chosen dataset is a JSON file containing 147 Aesop Fables, it is open-source and available on GitHub [25]. Figure 42 shows an example of its structure.

```
{
  "stories":[
    {
      "number": "01",
      "title": "THE WOLF AND THE KID",
      "story": [
        "There was once a little Kid whose growing horns made him t
hink he was a grown-
up Billy Goat and able to take care of himself.",
        "So one evening when the flock started home from the pastur
e and his mother called, the Kid paid no heed and kept right on nib
bling the tender grass.",
        ...
      ],
      "moral": "Do not let anything turn you from your purpose.",
      "characters": []
    }, ...
}
```

Figure 42 - Structure of the Aesop dataset, in which fable are stored as lists of sentences.

As a first step fables were extracted and cleaned bringing all characters to lower case, substituting any English language abbreviations. The task on which the model is trained on is the prediction of just one character. Inputs are sequence of characters and predictions are the most probable following

character at each time step. In order to obtain this pre-process, the text is divided in sequences of a maximum length, each input sequence has a corresponding target of the same length but shifted one character to the right. Considering a maximum length of 30 characters:

- 'there was once a little kid wh'
- 'ose growing horns made him thin'
- 'k he was a grownup billy goat a'
- 'nd able to take care of himself'
- ' . so one evening when the flo'
- . . .

Preprocess of the first sentence:

- *Input data*: 'there was once a little kid w'
- *Target data*: 'here was once a little kid wh'

Time steps:

- input: 't', expected output: 'h'
- input: 'h', expected output: 'e'
- . . .

In order to generate a sentence with a fixed dimensionality, the network generates a prediction distribution of the next character using the start string and the LSTM state. It uses a multinomial distribution to calculate the index of the predicted character, then it is used as the next input to the model. The new LSTM state is fed back into the network so that it now has more context. Predicting the next character again and again, the modified states are continuously fed back into the model, which is how it learns as it gets more context from the previously predicted words. As shown in figure 42, four identical networks were trained and tested with increasing

number of epochs, observing the variation of the loss function, which became 0,3602 after 100 epochs.

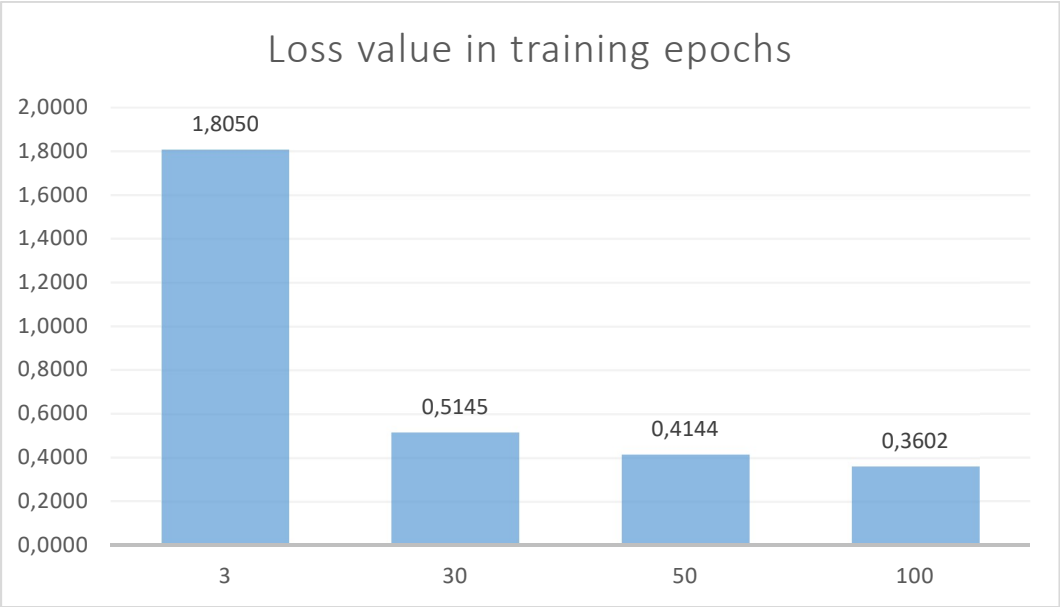


Figure 43 - Loss function through epochs in LSTM for text generation training on Aesop fables

The network started to generate sentences containing correct words and apparently good syntax structure, at least in the beginning of the generation. However, after few words, the network started to generate no-sense words too, as can be observed in figure 44.

Initial sentence	There was once a little Bear
Generation	him , and the king of beasts is like wax in the ground roon with them , and the snakes kindly appearance , he bears . so he about to d . just then the cat let go a good lesson learned . a friving her good boasted now the cranes were going to straid and strck . besides , the animals were carried one of them . it bear whire the lion had little happen i cannot tell you how gecorners . i see ! mother mole saw sure i can soon gnaw this stalks out of the ground with creature . a thirsty

Figure 44 - Results of a character-based generation of Aesop fables, performed with a NN containing LSTM cells

What is completely missing in these results is the semantic logic of the generated sentence, apart from the contextual relation of words. The same test was performed on a word-based generation task and the network showed similar results, reported in figure 45.

Initial sentence	There was once a little Bear
Generated 1	got inside than that of any of their neighbors , the farmer and his son were on their way to market with them . a hedgehog happened by . let me go ! the weasel had not been for that terrible monster , the beetle flew to the ass could not hear him , i got a huge bear crashed out of the brush near them . a frightful din in the cave , kicking and braying , and needed a weasel walked into a pit where the sun had a
Generated 2	. pull them out and ill divide them between and save you . so the earthen pot in the pile of broken dishes from the table . there they flew out of him . please as she grew older , especially if anything disturbed . if you will please go home with her to do . when the wolf felt that the harder to endure . silence ! , as he expected , giving their work and claws . but what he would have to might .

Figure 45 - Results of a word-based generation of Aesop fables performed with a NN containing LSTM cells

After few rare context-related words the generation become completely random. Respect the character-based generation, no wrong words could be generated, but the semantic logic of the generation is again completely missing. More of that, it was observed that two generations from the same starting sentence are completely different. As a final test, the bigger model was compared with one identical network, but containing a pre-trained embedding matrix. The words vectors were calculated with Google's Word2Vec skip-gram model on the text8 dataset, containing the first 10^9 bytes of the English Wikipedia dump on Mar. 3, 2006 [26]. This dataset

was chosen because of the good results obtained in sentiment analysis tests with CNNs, paragraph 2.3.2; results are reported in figure 46. No significant improvements were observed introducing a pre-trained vector matrix, probably because the LSTM recurrent layer has a much bigger impact on the generation than the Embedding one. Vector representations didn't improve the memory of the network.

Initial sentence	There was once a little Bear
Generation 1	. the crane soared in freedom far up into the blue sky and felt a desire to fly , as was his custom hide , that one day as i was passing near a large cave on the mountain side , when he suddenly took it into his basket . how foolish i should be , all birds have feathers ! i am nothing at all , replied the dog . what ! , snarled the wolf as he crept into the kite flew down to the boaster
Generation 2	every not see what good friends we shall become . the waves washed it up on shore . but his plans were very much changed when he met a lion and furiously began to tear it with their teeth . and when they returned next day to look for visitors . and after he had been walking . wishing also to rest in a wolf and began to his life , and the goats out to feed , the wild goats scampered the animals respectfully made way for him , an ass

Figure 46 - Results of a word-based generation of Aesop fables performed with a NN containing LSTM cells and pre-trained word embeddings (on text8 dataset [26])

3.5.2 Experiment Two: Encoder-Decoder LSTM

Implementing the Encoder-Decoder architecture a network can generate variable dimension sequences, meaning that it will be the model itself to decide how many words must be generated given an input sequence. However, in order to achieve the result, text must be pre-processed in a way that let the model understand where a sequence starts and where it

ends. The first step consists in modifying the vocabulary used for the task, adding three new tokens:

- '<PAD>' encoded with 0
- '<START>' encoded with 1
- '<END>' encoded with 2

Text is divided into sequences of words, respecting a maximum length decided a priori. Each sequence will generate as many samples as its number of words. For example, say the maximum length is 4 and the text is "Hello my name is Dario and I love to code".

The sequences are:

- "Hello my name is "
- "Dario and I love"
- "to code"

Then with each sequence:

- EncoderInput: "START Hello END"
  DecoderInput: "START my name is END"
  Target: "my name is END"
- EncoderInput: "START Hello my END"
  DecoderInput: "START name is END"
  Target: "name is END"
- EncoderInput: "START Hello my name END"
  DecoderInput: "START is END"
  Target: "is END"
- EncoderInput: "START Hello my name is END"
  DecoderInput: "START END”
  Target: "END"

In this way, sequences of every length can be preprocessed together. The purpose of the padding token is to be able to encode even sentences shorter than the maximum length. The encoder simply takes the input data, and train on it then it passes the last state of its recurrent layer as an initial state to the first recurrent layer of the decoder part. So, the encoder model is a Neural Network composed by:

- Input layer
- Embeddings layer
- Recurrent Layer (Long Short Memory Network)

The decoder takes the last state of encoder's last recurrent layer and uses it as an initial state to its first recurrent layer, the input of the decoder is the sequences that we want to get. So, the decoder model will be a simple Neural Network composed by:

- Input layer
- Embeddings layer
- Recurrent Layer (Long Short Memory Network)
- Dense output (to predict the next word)

The model has been trained with and without text8 pre-trained word vectors, obtained training a Word2Vec skip-gram model. Results have demonstrated the hypothesis from the experiment one, according to which LSTM cell have a much higher impact on the generation of words than the embedding layers. In fact, the network is always generating text entirely seen during training, remembering sentences of the training dataset. If the training dataset contains input and targets created with a short maximum length, the network tries to mix the sentences, but without logical sense, as shown in figure 47. If the maximum length is big (respect the total length of a fable), 100 words in figure 48, the network again tends to re-propose

pieces of fables entirely seen during training, without any contextual connection with the initial sentence. Giving the network two words or thirty do not change the results.

phrase	generated
The Cock	bank , i know that he is the prettiest , the sweetest , the dearest darling in the world . <END>
A Dog and a Wolf	dogs with much to eat because of the watchfulness of the shepherds . but one night he found a sheep skin that had been cast <END>
An eagle was given permission to fly over the country.	fell to come out . his great this time very exclaimed the well in the foxes the <END>
A dog was talking to a bear asking for some food. The bear who was hungry too said no.	show the two of i . what <END>

Figure 47- Results of Encoder-Decoder LSTM used to generate Aesop fables. The dataset used to train the network contains input-target samples with maximum of 30 words. Colored sections of the generated sentences indicate a series of words entirely seen during training

The results can be also interpreted as an overfitting of the network, but they have been achieved after 10 epochs of training, for the dataset with 30 as maximum length, and just 2 epochs of training for the dataset with 100 as maximum length. The dataset contains only 147 fable, from which about 25000 input/target sample can be extracted. However, the same network can achieve very good results in a Machine Translation task, as show in [27]. Translating an English sentence to a French one is a task very similar to remember which is the translation of each combination of words seen during the training, while generating a completely new fable after seeing some during training is a different goal. The use of an RNN such as LSTMs in text generation is probably not a good choice because the network does not learn how to generate new text, but simply remember the text seen during training.

phrase	generated
The Cock	the fox , who had never seen anything of the world , almost came to grief the very first time he ventured out . and this is the story he told his mother about his adventures . i was strolling along very peaceably when , just as i turned the corner into the next yard , i saw two strange creatures . one of them had a very kind and gracious look , but the other was the most fearful monster you can imagine . you should have seen him , on top of his head and in <END>
A Dog and a Wolf	the fox , who had gathered to elect a new ruler , the monkey was asked to dance . this he did so well , with a thousand funny capers and grimaces , that the animals were carried entirely off their feet with enthusiasm , and then and there , elected him their king . the fox did not vote for the monkey and was much disgusted with the animals for electing so unworthy a ruler . one day he found a trap with a bit of meat in it . hurrying to king monkey <END>
An eagle was given permission to fly over the country.	the wolf , and all the savage beasts recited the most wicked deeds , all were excused and made to appear very saintlike and innocent . it was now the ass turn to confess . i remember , he said guiltily , that one day as i was passing a field belonging to some priests , i was so tempted by the tender grass and my hunger , that i could not resist nibbling a bit of it . i had no right to do it , i admit <END>
A dog was talking to a bear asking for some food. The bear who was hungry too said no.	the wolf , but all his efforts were in vain . then the sun began to shine . at first his beams were gentle , and in the pleasant warmth after the bitter cold of the north wind , the traveler unfastened his cloak and let it hang loosely from his shoulders . the sun rays grew warmer and warmer . the man took off his cap and mopped his brow . <END>

Figure 48 – Results of Encoder-Decoder LSTM used to generate Aesop fables. The dataset used to train the network contains input-target samples with maximum of 100 words. Colored sections of the generated sentences indicate a series of words entirely seen during training

As can be observed in Figure 49, the incorporation of pre-trained word vectors seems to have no influence on RNN performance and generated results, like for a simple NN in sentiment analysis.

phrase	generated
The Cock	mouse , who had to take them to the found , there he was able to do . lifting his wings he tried to rise from the ground . but the weight of his magnificent train held him down . instead of flying up to greet the first rays of the morning sun or to bathe in the rosy light among the floating clouds at sunset , he would have to walk the ground more encumbered and oppressed than any common barnyard fowl .
A Dog and a Wolf	tree , and the animals respectfully made way for him , an ass brayed a scornful remark as he passed . the lion felt a flash of anger . but when he turned his head and saw who had spoken , he walked quietly on . he would not honor the fool with even so much as a stroke of his claws .
An eagle was given permission to fly over the country.	when the fox saw the mouse , he swam to the bank and croaked : won you pay me a visit ? i can promise you a good time if you do . the mouse did not need much coaxing , for he was very anxious to see the world and everything in it . but though he could swim a little , he did not dare risk going into the pond without some help . the frog had a <END>
A dog was talking to a bear asking for some food. The bear who was hungry too said no.	they had so jupiter to do that any animal that he chose for a meal , should be so brazen as to wear such dangerous things as horns to scratch him while he ate . so he commanded that all animals with horns should leave his domains within twentyfour hours . the command struck terror among the beasts . all those who were so unfortunate as to have horns , began to pack up and move out . even the hare , <END>

Figure 49 – Results of Encoder-Decoder LSTM, with pre-trained embedding layer (Word2Vec skip-gram on text8 [26]) used to generate Aesop fables. Input-target samples with maximum of 100 words. Colored sections of the generated sentences indicate a series of words entirely seen during training

3.5.3 Experiment Three: The Transformer

The aim of this last experiment is to understand first, if the transformer model is better than RNNs in a text generation task, and second, if pre-trained word vectors can influence its performance. The model was not built from scratch using Tensorflow libraries like for the Encode-Decoder in experiment two. The open source repository in [28], provides a Keras implementation of the general Transformer:

```
# Build the model
model = get_model(
    token_num=len(word2idx),
    embed_dim=EMBEDDING_DIM,
    encoder_num=ENCODERS,
    decoder_num=DECODERS,
    head_num=HEADS_ATTENTION,
    hidden_dim=HIDDEN_DIM,
    attention_activation=ACTIVATION_FUNCTION,
    feed_forward_activation=ACTIVATION_FUNCTION,
    dropout_rate=DROPOUT_RATE,
    embed_weights=embeddingMatrix,
)
```

To create the model is enough to invoke a simple function passing some important parameters like activation function of layers, dropout rate, pre-trained embedding matrix and number of encoders and decoders in the architecture. All parameters were taken from [22], apart from the number of encoders and decoders, six in the paper, only one in this test because of small dataset of 147 fables. The model has been tested with and without a pre-trained embedding matrix obtained from the text8 dataset. The presence of generated word vectors has a good influence on the decreasing of the loss function already from the first epoch, as shown in figure 50. Looking at the generated text after just 40 epochs, a huge difference between the transformer and LSTM emerged. In fact, in this case the generated sentences are composed by groups of 2-3 words seen during the

training, all mixed together. The network tries to mix pieces of training text to create new arrangements.

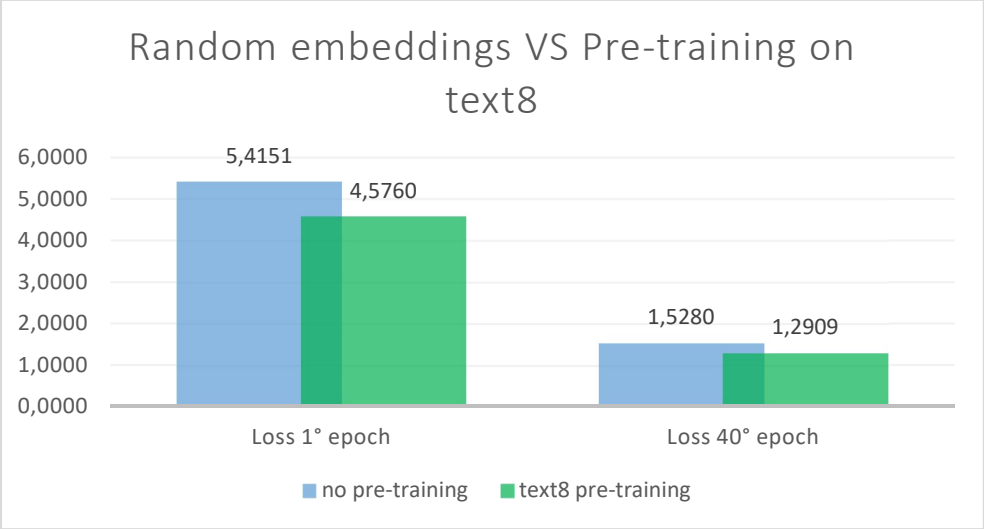


Figure 50 – Transformer model from [22] with only one encoder and one decoder was tested with and without a pre-trained embedding matrix, the plot shows loss function values after the first and 40° epoch

No-pretraining

Initial sentence	**genereation**
The Dog	<START> the fox was very hurried any bones about it would be to fire <END>
There was once a little	<START> of the animals who had become entangled in the accustomed <END>
An eagle was given permission to fly over the country	<START> . <END>

Text8 pre-trained embeddings

Initial sentence	**generation**
The Dog	<START> was very angry to try to make love to me but j should <END>
There was once a little	<START> very kind ass and yet unable to get out again <END>
An eagle was given permission to fly over the country	<START> road leading down in each <END>

Figure 51 - Results from Transformer model [22] with only one encoder and one decoder, used to generate text after a training of 40 epochs on 147 Fables. The training dataset contains sentences with a maximum length of 15 words. Colored sections indicate a series of words entirely seen during training

Results in figure 51 suggest that the presence of the embedding matrix improves also the quality of the generated sentences, but further tests with longer maximum length of the sentences and more epochs of training have demonstrated that the dataset of 147 fables is too small to train well the model. However, the ability of the Transformer to mix pieces of training sentences has encouraged to test the model on different datasets. The first tested dataset contains 515'000 hotel reviews and it was chosen because of the strong similarity between reviews, each talk about rooms and hotel services [29]. The idea is that seeing a great number of different opinions about the same entity could give the model a lot to learn on how to generate a sentence in many ways. Only 4000 reviews were used to train the model, while the all 515'000 were used to generate word vectors with a Word2Vec skip-gram model. Decision justified by the number of training words:

- Fables dataset → 147 fables → 27195 words
- Reviews dataset → 4000 reviews → 52000 words

Training the model on 4000 reviews is the equivalent of training the model on a dataset that is approximately double the fables dataset. Considering that reviews max length is 30 words, each one is preprocessed as a complete input for the model, while most of the fables did require to be separated in chucks of 100 words. The results show the important role of pre-trained embeddings (on all 515'000 reviews), figure 52 shows that after 100 epochs of training, the model with random initialized word vectors cannot generate meaningful reviews, while the same model with pre-trained layers on the entire dataset was able to generate sentences, not completely correct from a syntactic point of view, but certainly rich of contextual information respect the initial sentence. The fables dataset test

and the reviews dataset test can be considered as small experiments done to observe the behavior of the model and to infer general rules.

Without pre-trained embeddings

Initial sentence	**generation**
I would come with someone else	the hotel and the location was mush its rooms and the location was much
I only stayed for	the bed the bed was the hotel the rate i would peeling staff were very on
A lovely breakfast	and the staff were very helpful and friendly
The breakfast	was very good and the staff were very helpful and fresh
Good location and the room	was very good staff very helpful breakfast good choice
Very kind staff	and comfortable bed

Pre-trained embeddings on 515k reviews

Initial sentence	**generation**
I would come with someone else	can t have been better for me
I only stayed for	a while one night we arrived we were upgraded to the rooms were very nice in
A lovely breakfast	and the staff were very helpful location was excellent choices
The breakfast	was very good
Good location and the room	very clean and nice place to stay in great breakfast
Very kind staff	good location

Figure 52 – Results from Transformer model [22] with only one encoder and one decoder, used to generate text after a training of 100 epochs on 4000 hotel reviews. Colored sections indicate a series of words entirely seen during training

For instance, it can be assumed that pre-trained word vectors can be used to increase quality of text generation with attention mechanism, or to reduce the dimension of the training dataset needed for the task. A third dataset was tested, bigger than previous ones, it contains a collection of sci-fi books equivalent to 30'524'481 of words [30]. After 9 hours of training with a single GPU (Nvidia RTX 2060) the model performed just 3 epochs bringing the loss value from 6,2713 to 6,0429. In order to train the

Transformer model on such a dataset, it must be run on a distributed computing cluster. Bigger models trained faster, so another solution to train on a big dataset is to increase the number of encoders and decoders, but how much? A last test will try to point out the behavior of the model in relation to its dimension and to the dimension of the training dataset. The training corpus is a single sci-fi book composed by 12714 words, "contagion" by Katherine MacLean, which was retrieved from Project Gutenberg [31], while the embedding matrix was pre-trained on the sci-fi corpus of 30'524'481 words. Figure 53 shows the loss function variation respect 1 encoders/decoders, 3 and 6.

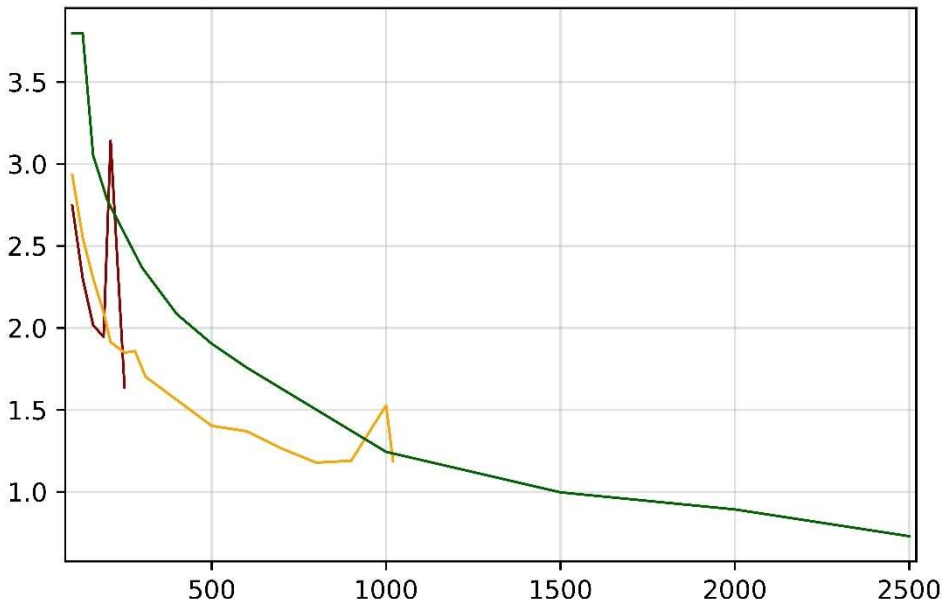


Figure 53 - Transformer model from [22] with pre-trained skip-gram vectors on sci-fi corpus of 30'524'481 words, trained on single sci-fi book of 12714 to perform a text generation task. The plot compares the loss function decreasing for 1 encoder, 1 decoder (GREEN), 3 encoders, 3 decoders (ORANGE), 6 encoders, 6 decoders (RED)

The graph represents the loss function values at steps of 30 epochs until the firsts ~250, then at steps of 100 epochs until ~1000 and ends up with

steps of 500 epochs. Globally, loss values go from epoch 100 to epoch 2500. The loss function of the bigger model starts with the lowest value between the three, but it is clearly unstable. This means that a bigger number of encoders/decoders can train faster but are not suited to learn small problems such as train on a single book. Figure 54 shows the instability of the loss function in its last 50 epochs.

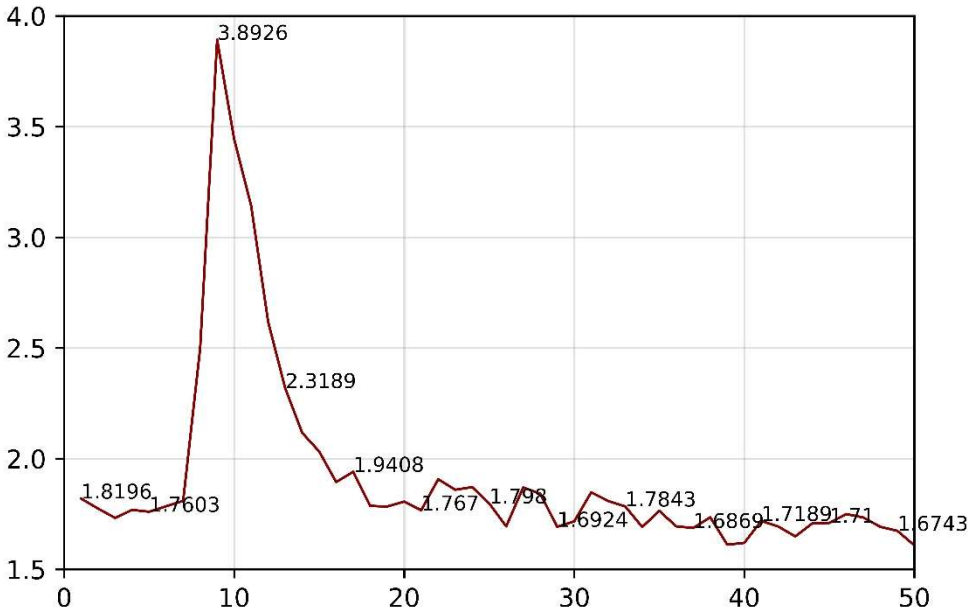


Figure 54 – Instability of the loss function decrease during last 50 epochs of training using a transformer model with 6 encoders and 6 decoders

The second model composed by 3 encoders and 3 decoders is again unstable and can't go under ~1.100. Its last 50 epochs out of 1000 are shown in figure 55. The last smaller model produces a more stable decrease of the loss function and in respect the previous two it reaches values under the 1. However, it showed a little instability too, during last epochs, showed in figure 56. A generation test with smaller model have led to no-sense generated sentences, even if the loss function has fallen below 1. These results suggest that if the Transformer model has too much

encoders/decoders in respect to training dataset dimension it could led to a great instability of the training. The single book containing 12714 words is a smaller dataset compared with fables (27195 words) and reviews (52000 words). So, why the smaller model with one encoder and one decoder can't learn to generate from the book? The book dataset has a significant difference in respect to the others. All training fables contain a very simple language because they are written for children, more of that they all contain stories about animals and share a lot of linguistic similarities. The hotel reviews all contain opinions about the same entities, rooms, breakfast, location and so on. The dimension of the dataset and model must be scaled with the number of entities that compare in texts, one single book probably contains a lot of different situations like description of places, dialogs, narrative scenes, character's thoughts and many others.

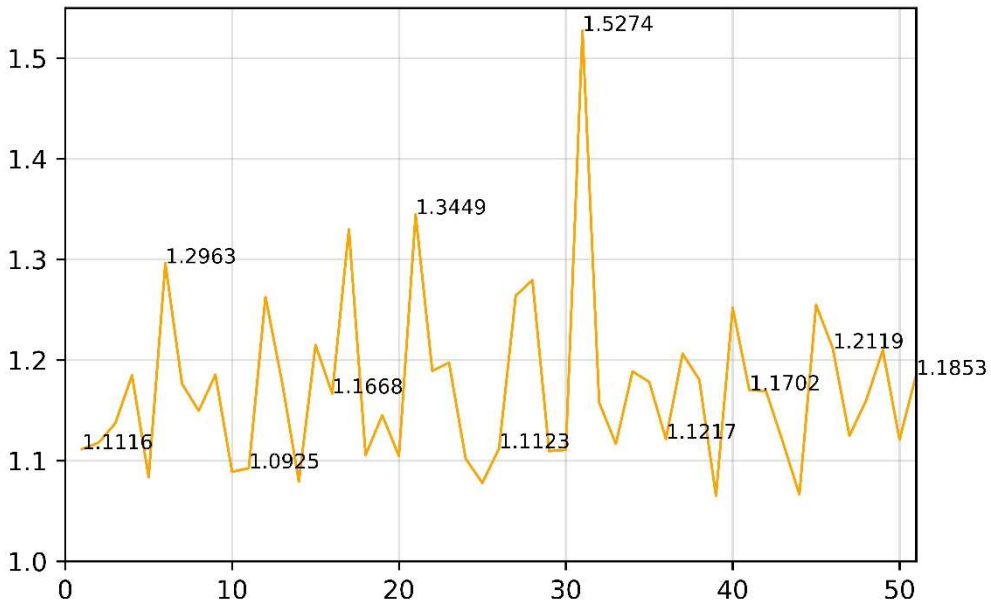


Figure 55 - Instability of the loss function decrease during last 50 epochs of training using a transformer model with 3 encoders and 3 decoders

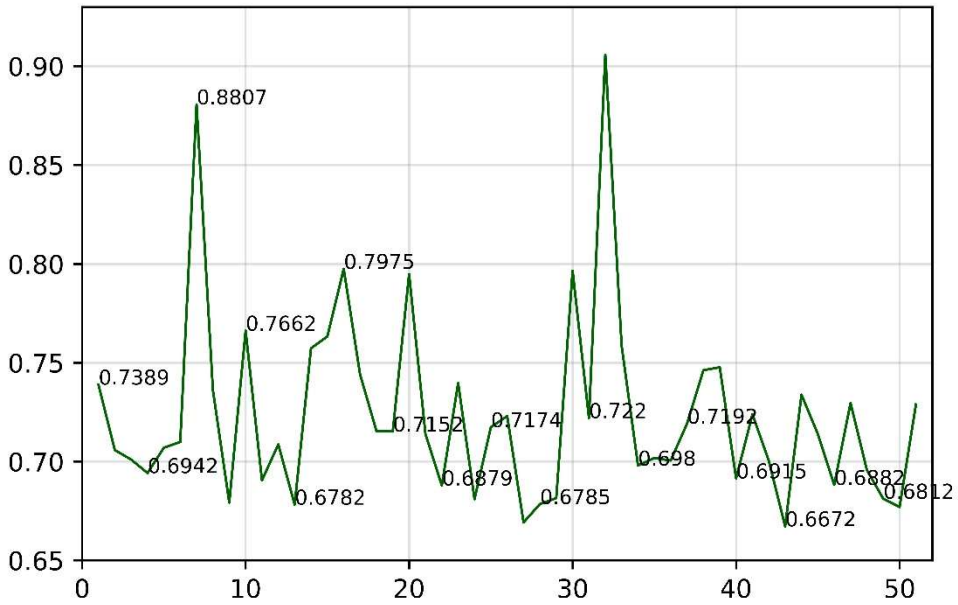


Figure 56 - Instability of the loss function decrease during last 50 epochs of training using a transformer model with 3 encoders and 3 decoders

3.6 Results discussion

Experiment one and Experiment two have demonstrated that a Deep Neural Network with just Recurrent layers is not good in performing a text generation task. The ability of expand the neural network memory is not a feature that enable the creation of new contents. However, they have represented a good first test field in implementing complex architectures (Encoder-Decoder) with the Keras library and Python programming language. Pre-trained word embeddings have been again produced with the Word2vec skip-gram model, and they didn't improve the LSTMs performance in any case. Representation of words as vectors in a multi-dimensional space can't influence the training phase of a recurrent neural network learning a text generation task. On the contrary, the Transformer

model, which wasn't built from scratch for simplicity, showed encouraging results in the text generation field. Even if tests in experiment three were only toy tests of text generation, developed using small datasets of reviews and fables, the Transformer has shown the ability to mix words, trying to create new sentences never seen during the training phase. The fables dataset, containing only 147 short stories, is not big enough to teach to a such complex network the Aesop style of writing, but the test with hotel reviews demonstrated two important characteristics. First, if the training dataset contains enough sentences on the same entities (staff, room, cleaning, for instance), the network learns better how to mix words to create new phrase about those entities, which are semantically correct. Second, the use of pre-trained word embeddings can play an important role in the decrease the training necessary to achieve good results. A transformer with a pre-trained embedding matrix, one encoder and one decoder, generated good reviews examples after a small training on just 4000 reviews out 515'000. The last very important inferred rule is that the dimension of the dataset and model must be scaled with the number of entities that compare in texts, because even if the book used in the last test, 'Contagion' by Katherine MacLean, is smaller than the fables, and reviews dataset, the model with one encoder and one decoder couldn't learn nothing from it, even after 2500 epochs of training. The entities, type of narrations and linguistic pattern that appear in a single book can be so many, that in order to achieve good results in generating a book, a very large dataset must be prepared, and so a very large model. The larger model used in this work (6 encoders and 6 decoder) contains about 39 millions of parameters, while the GPT-2 model from [19] contains about 1500 millions of parameters. Moreover, it was trained using google private infrastructures on a dataset, not released yet, which involved millions of

people around the world for its creation. Differences in dimensions between GPT-2 and this work are significant and cannot be filled for obvious reasons. However, GPT-2 was trained to become a language model able to perform well in different NLP tasks. Results of experiment three suggests that maybe with a well-defined and structured training, taking advantage of ad-hoc pre-trained word embeddings, the Transformer can achieve good results in specific text generation tasks.

4. Conclusions and future work

" The face of a loved one is not stored once but on the order of thousands of times.
Some of the repetitions are largely the same image of the face,
whereas most show different perspective of it, different lighting, different expressions.
None of these repeated patterns are stored as images per se.
Rather they are stores as lists of features,
where the constituent elements of a pattern are themselves patterns. "

Ray Kurzweil
(How to create a Mind: the secret of human Thought revealed)

4.1 Conclusions

During last years, Natural Language processing has enormously evolved and grown, expanding its potentiality and application fields. The backpropagation process played a key role in the transition from Machine Learning to Deep Learning. Statistical models have been substituted by complex deep neural networks, which automatically learn how to understand and manage languages. By now, one single model (GPT-2) can perform many tasks such as, reading comprehension, machine translation, question answering, summarization and text generation, without specific trainings. Together with the evolutions of models, the arrival of advanced text representation techniques boosted this significant growth. Word embeddings are present inside most successful networks, they have been studied a lot and different variations of the same idea already exist. Deep Neural Networks often require a huge amount of labeled data to be trained well, while the creation of word embeddings is obtained through an unsupervised training and, unlabeled text is everywhere. This work explored the benefits of pre-trained word vectors in terms of performance

improvement and workload reduction for Deep Neural Networks. Two tasks have been performed during experiments, sentiment analysis and text generation, respectively part of two different sub-domains of NPL: Natural Language Understanding (NLU) and Natural Language Generation (NLG). The use of pre-trained word vectors can improve results of both, but not using any type of neural network. For instance, the simple multilayer perceptron is not influenced by pre-trained word vectors in a sentence classification task, but CNNs are. In the same way, RNNs are not influenced in a text generation task, but Transformer is. Each specific problem must be addressed making a preliminary study to find the best model, the right dimension of it and the amount of data required. The increasing complexity of tests allowed to approach the analyzed problems from the simplest point of view, achieving important skills in programming ETL processes and building trainable Neural Networks.

4.2 Future Work

Many different adaptations, tests, and experiments have been left for the future due to lack of time and resources (i.e. the experiments with Deep Neural Networks are usually very time consuming, requiring even days to finish a single run). Future work concerns further analysis of word embeddings, deepening on other creation techniques besides Word2Vec models. For instance, specific sentiment analysis word embeddings, such as the ones described in [11], or simply other standard models like GloVe can be tested and compared with the ones used in this work. The sentiment analysis test was performed on a binary classification using a very small dataset. One future objective certainly is to find out if pre-trained word vectors can improve CNNs performance even if the complexity of the task

is increased, adding a neutral class or more sentiments to recognize besides positive and negative, or exploring more deep sentiment analysis like intent Analysis or Contextual Semantic Search. A second future objective is to identify the right dimension of the Transformer model to obtain good specific text generations, paying attention to the training dataset, e.g. type of text, number of entities, languages. One important problem felt during text generation experiments is the amount of memory and resources needed to train models that can generate long stories, not just simple sentences connected to one input. Lara J. Martin et al. addressed this problem in [32], moving the problem from text generation to event generation. They invented a way to encode a sequence of events, not words, and then train deep neural networks to generate those sequences. An idea to further evolve this work is to explore the possibility of generating text from an encoded event, instead of a seed piece of text.

References

[1] F. Sebastiani, "Machine Learning in Automated Text Categorization," *Consiglio Nazionale delle Ricerche, Italy,* 2002.

[2] T. Young, D. Hazarika, S. Poria and E. Cambria, "Recent Trends in Deep Learning Based Natural Language Processing," *arXiv.org,* 2018.

[3] R. Colvin, Modeling Neural Networks using Process Algebra, The University of Queensland, The Queensland Brain Institute, The Science of Learning Centre, 2013.

[4] Google, "Deep Learning crash course," [Online]. Available: https://eu.udacity.com/course/deep-learning--ud730.

[5] D. Britz, "Understanding Convolutional Neural Networks for NLP," [Online]. Available: http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/.

[6] Google, "Machine learning crash course," [Online]. Available: https://developers.google.com/machine-learning/crash-course/.

[7] S. Gupta, "Sentiment Analysis: Concept, Analysis and Applications," 2018. [Online]. Available: https://towardsdatascience.com/sentiment-analysis-concept-analysis-and-applications-6c94d6f58c17.

[8] T. Mikolov, "Efficient Estimation of Word Representations in Vector Space," 2013.

[9] C. McCormick, "Word2Vec tutorial: the skip-gram model," 2016. [Online]. Available: http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/.

[10] J. Pennington, R. Socher and C. Manning, "GloVe: Global Vectors for Word Representation," [Online]. Available: https://nlp.stanford.edu/projects/glove/.

[11] D. Tang, Q. Bing, N. Yang, W. Furu, T. Liu and M. Zhou, "Sentiment Embeddings with Applications to Sentiment Analysis," *IEEE Transactions on Knowledge and Data Engineering,* 2016.

[12] M. Koppel and J. Schler, "The Importance of Neutral Examples for Learning Sentiment," *Computational Intelligence,* 2006.

[13] R. Dorgueil, "Bonobo project," [Online]. Available: https://www.bonobo-project.org.

[14] Open-source, "An end-to-end open source machine learning platform," [Online]. Available: https://www.tensorflow.org.

[15] D. P. Kingma and J. Lei Ba, "Adam: a method for stochastic optimization," *arXiv.org,* 2014.

[16] I. d. S. e. T. d. ". Faedo", "Italian Word Embeddings from Wikipedia.," [Online]. Available: http://hlt.isti.cnr.it/wordembeddings/.

[17] Y. Kim, "Convolutional Neural Networks for Sentence Classification," *arXiv, New York University,* 2014.

[18] OpenAI, "Better Language Models and Their Implications," 2019. [Online]. Available: https://openai.com/blog/better-language-models/.

[19] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei and I. Sutskever, "Language Modes are Unsupervised Multitask Learners," 2019.

[20] I. Sutskever, O. Vinyals and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," *Advances in neural information processing systems,* 2014.

[21] S. Kostadinov, "Encoder-Decoder sequence to sequence model," Towards Data Science, [Online]. Available: https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model-679e04af4346.

[22] V. Ashish, S. Noam, P. Niki, U. Jakob, J. Llion, G. Aidan, K. Lukasz and P. Illia, "Attention is all you need," *Neural Information Processing Systems Conference,* 2017.

[23] Synced, "A Brief Overview of Attention Mechanism," Medium, 2017. [Online]. Available: https://medium.com/syncedreview/a-brief-overview-of-attention-mechanism-13c578ba9129.

[24] Open-source, "Tensorflow, char-based text generation with LSTM," [Online]. Available: https://www.tensorflow.org/tutorials/sequences/text_generation.

[25] Open-source, "Github, machine learning project on Aesop fables," [Online]. Available: https://github.com/itayniv/aesop-fables-stories.

[26] M. Mahoney, "Text8 dataset," [Online]. Available: http://mattmahoney.net/dc/textdata.html.

[27] D. Maheshwari, "Neural Machine Translation using word level seq2seq model," Medium.com, 2018. [Online]. Available: https://medium.com/@dev.elect.iitd/neural-machine-translation-using-word-level-seq2seq-model-47538cba8cd7.

[28] Open-source, "keras-transformer," 2019. [Online]. Available: https://github.com/kpot/keras-transformer.

[29] J. Liu, "Hotel reviews data in europe," Kaggle, 2017. [Online]. Available: https://www.kaggle.com/jiashenliu/515k-hotel-reviews-data-in-europe.

[30] J. Klaas, "Scifi stories text corpus," Kaggle, 2018. [Online]. Available: https://www.kaggle.com/jannesklaas/scifi-stories-text-corpus#internet_archive_scifi_v3.txt.

[31] K. MacLean, ""Contagion"," Project Gutenberg, [Online]. Available: https://www.gutenberg.org/ebooks/50774.

[32] L. Martin, P. Ammanabrolu, X. Wang, W. Hancock, S. Singh, B. Harrison and M. Riedl, "Event Representations for Automated Story Generation with Deep Neural Nets," *The Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18),* 2018.

List of Figures

FIGURE 1 - REPRESENTATION OF A NEURAL NETWORK AS A FUNCTION 10

FIGURE 2 – BASIC STRUCTURE OF A NEURAL NETWORK: INPUT LAYER, HIDDEN LAYER AND OUTPUT LAYER 11

FIGURE 3 – MATH BEHIND HIDDEN NEURONS RECEIVING INPUT AND COMPUTING THEIR ACTIVATION 12

FIGURE 4 – MATH BEHIND OUTPUT NEURONS RECEIVING INPUT AND COMPUTING THEIR ACTIVATION 13

FIGURE 5 – MATH BEHIND NEURAL NETWORK PRODUCING AN OUTPUT, COMPARING IT WITH A TARGET VALU. 13

FIGURE 6 MATH BEHIND CONNECTION WEIGHTS GETTING UPDATED IN A NEURAL NETWORK 14

FIGURE 7 – MATH BEHIND BACKPROPAGATION OF THE ERROR IN HIDDEN LAYERS OF A NEURAL NETWORK. 14

FIGURE 8 - GRAPHICAL REPRESENTATION OF A CNN PATCH APPLIED TO A TRIDIMENSIONAL INPUT. 16

FIGURE 9 - GRAPHICAL REPRESENTATION OF STRIDE PARAMETER IN CNNS 17

FIGURE 10 - GRAPHICAL REPRESENTATION OF THE WORKFLOW OF AN RNN OVER TIME 19

FIGURE 11 – SIMPLEST STRUCTURE OF RNNS: HIDDEN LAYERS ARE CONNECTED TO EACH OTHER'S 20

FIGURE 12 - BACKPROPAGATION PROCESS IN RNNS INVOLVING MULTIPLE DERIVATIVES THROUGH TIME 21

FIGURE 13 - SPARSE VECTORS REPRESENTING USERS IN A MOVIE RECOMMENDATION MODEL 22

FIGURE 14 - GRAPHICAL EXAMPLE OF A WORD EMBEDDINGS SPACE, IN WHICH SIMILAR WORDS 24

FIGURE 15 - COSINE DISTANCE FORMULA AND GRAPHICAL EXAMPLE 25

FIGURE 16 – GRAPHICAL EXAMPLE OF SAMPLED SOFTMAX 26

FIGURE 17 - WORD2VEC CBOW VS. SKIP-GRAM 28

FIGURE 18 - EXTRACTION OF INPUTS AND TARGETS FROM TEXT FOR A WORD2VEC SKIP-GRAM MODEL 29

FIGURE 19 - STRUCTURE OF WORD2VEC SKIP-GRAM MODEL FOR A VECTOR SPACE OF 300 DIMENSIONS AND A VOCABULARY OF 10'000 WORDS 30

FIGURE 20 – GRAPHICAL REPRESENTATION OF A THREE DIMENSIONS VECTOR SPACE CREATED BY THE WEIGHTS 31

FIGURE 21 - MATHEMATICAL COMPUTATION OF SOFTMAX OUTPUT IN A WORD2VEC SKIP-GRAM MODEL 32

FIGURE 22 - DIFFERENCES BETWEEN THREE SETS OF REVIEWS LABELED BY THREE DIFFERENT PEOPLE. 34

FIGURE 23 - K-FOLD CROSS VALIDATION 35

FIGURE 24 - HISTOGRAM ABOUT CHATBOT REVIEWS LENGTH 36

FIGURE 25 - HISTOGRAM ABOUT THE INFLUENCE OF THE EMBEDDING LAYER DIMENSION 39

FIGURE 26 - 2D REPRESENTATION OF WORD EMBEDDINGS WITH 64 DIMENSIONS, .. 40

FIGURE 27 – NN PERFORMANCES COMPARISON BETWEEN A RANDOM INITIALIZED EMBEDDING LAYER 41

FIGURE 28 -NN PERFORMANCES COMPARISON BETWEEN A RANDOM INITIALIZED EMBEDDING LAYER AND 42

FIGURE 29 – NN PERFORMANCES COMPARISON BETWEEN A RANDOM INITIALIZED EMBEDDING LAYER AND 43

FIGURE 30 - - CNN ARCHITECTURE FROM Y. KIM WORK ON SENTENCE CLASSIFICATION [17] 44

FIGURE 31 - CNN PERFORMANCES COMPARISON BETWEEN A RANDOM INITIALIZED EMBEDDING LAYER AND 45

FIGURE 32- CNN PERFORMANCES COMPARISON BETWEEN A RANDOM INITIALIZED EMBEDDING LAYER 46

FIGURE 33 - CNN PERFORMANCES COMPARISON BETWEEN A RANDOM INITIALIZED EMBEDDING LAYER 47

FIGURE 34 – REPRESENTATION OF TYPICAL HIDDEN NODE OF A RECURRENT NEURAL NETWORK, THROUGH TIME. 51

FIGURE 35 - LSTM UNIT MACHINE STRUCTURE 51

FIGURE 36 -GRAPHICAL REPRESENTATION OF THE ENCODER-DECODER ARCHITECTURE [21] 54

FIGURE 37 - GRAPHICAL REPRESENTATION OF THE ATTENTION MECHANISM IN THE ENCODER-DECODER ARCHITECTURE. THE ENCODER IS HIGHLIGHTED IN BLUE WHILE THE DECODER IN RED [23] 56

FIGURE 38 - TRANSFORMER MODEL ARCHITECTURE [22], ON THE LEFT, THE ENCODER WHILE ON THE RIGHT, THE DECODER 58

FIGURE 39 – GRAPHICAL REPRESENTATION OF THE SELF-ATTENTION MECHANISM [23] 59

FIGURE 40 – IN THE TRANSFORMER, THE WORD AT EACH POSITION PASSES THROUGH A SELF-ENCODING PROCESS. 60

FIGURE 41 - ENCODER STRUCTURE VS. DECODER STRUCTURE IN THE TRANSFORMER [23] 60

FIGURE 42 - STRUCTURE OF THE AESOP DATASET, IN WHICH FABLE ARE STORED AS LISTS OF SENTENCES. 62

FIGURE 43 - LOSS FUNCTION THROUGH EPOCHS IN LSTM FOR TEXT GENERATION TRAINING ON AESOP FABLES 64

FIGURE 44 - RESULTS OF A CHARACTER-BASED GENERATION OF AESOP FABLES, ... 64

FIGURE 45 - RESULTS OF A WORD-BASED GENERATION OF AESOP FABLES PERFORMED WITH A NN CONTAINING LSTM CELLS 65

FIGURE 46 - RESULTS OF A WORD-BASED GENERATION OF AESOP FABLES PERFORMED WITH A NN CONTAINING LSTM CELLS AND PRE-TRAINED WORD EMBEDDINGS (ON TEXT8 DATASET [26]) 66

FIGURE 47- RESULTS OF ENCODER-DECODER LSTM USED TO GENERATE AESOP FABLES. THE DATASET USED TO TRAIN THE NETWORK CONTAINS INPUT-TARGET

samples with maximum of 30 words. Colored sections of the generated sentences indicate a series of words entirely seen during training....69

Figure 48 – Results of Encoder-Decoder LSTM used to generate Aesop fables. The dataset used to train the network contains input-target samples with maximum of 100 words. Colored sections of the generated sentences indicate a series of words entirely seen during training....70

Figure 49 – Results of Encoder-Decoder LSTM, with pre-trained embedding layer (Word2Vec skip-gram on text8 [26]) used to generate Aesop fables. Input-target samples with maximum of 100 words. Colored sections of the generated sentences indicate a series of words entirely seen during training....71

Figure 50 – Transformer model from [22] with only one encoder and one decoder was tested with and without a pre-trained embedding matrix, the plot shows loss function values after the first and 40° epoch....73

Figure 51 - Results from Transformer model [22] with only one encoder and one decoder, used to generate text after a training of 40 epochs on 147 Fables. The training dataset contains sentences with a maximum length of 15 words. Colored sections indicate a series of words entirely seen during training....73

Figure 52 – Results from Transformer model [22] with only one encoder and one decoder, used to generate text after a training of 100

epochs on 4000 hotel reviews. Colored sections indicate a series of words entirely seen during training 75

Figure 53 - Transformer model from [22] with pre-trained skip-gram vectors on sci-fi corpus of 30'524'481 words, trained on single sci-fi book of 12714 to perform a text generation task. The plot compares the loss function decreasing for 1 encoder, 1 decoder (GREEN), 3 encoders, 3 decoders (ORANGE), ... 76

Figure 54 – Instability of the loss function decrease during last 50 epochs of training using a transformer model with 6 encoders and 6 decoders 77

Figure 55 - Instability of the loss function decrease during last 50 epochs of training using a transformer model with 3 encoders and 3 decoders 78

Figure 56 - Instability of the loss function decrease during last 50 epochs of training using a transformer model with 3 encoders and 3 decoders 79