A Comparative Study on Deep Learning Approaches and Attention Mechanisms for Reading Comprehension on SQuAD Dataset

Author:
Elena MARTINA
33464991

Supervisor:
Dr. Tim BLACKWELL

A thesis submitted in fulfilment of the requirements for BSc Computer Science Degree

Goldsmiths, University of London
May 17, 2019

Abstract

Machine Reading Comprehension (MC) is a Natural Language Processing task where the machine is required to predict an answer to a specific query about a given context. Such problems not only have many practical applications, as automated customer service and virtual assistants, but their progress also has a significant value for the field of Natural Language Processing. This paper constitutes a study on Machine Comprehension and its applications in Question Answering (QA). In particular, it analyses and compare modern Neural Networks approaches to QA with a speculation on Attention Mechanisms. Along with the research, 4 separate models were implemented, of which only the most complex, R-Net, performed well on the dataset in question, SQuAD. Although the model was run for only 10 epochs, due to the resource constraints of the project, the implemented R-Net model was able to reach 10.5 EM score and 16.9 F1 score, proving the effectiveness of complex models featuring Attention Mechanisms on MC tasks.

Acknowledgements

First and foremost, I want to thank my supervisor, Dr. Tim Blackwell, for always steering me in the right the direction whenever he thought I needed it, and for always brightening the most stressful periods with his positivity.

I would also like to express my very profound gratitude to my flatmate R. M., who made the results of this thesis possible, not only because of his GPU, which was used 24/7 for several months, but also for his support and kindness.

Finally, I must thank my parents and R. for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	**A**rtificial **I**ntelligence
API	**A**pplication **P**rogramming **I**nterface
BERT	**B**idirectional **E**ncoder **R**epresentations from **T**ransformers
BGD	**B**atch **G**radient **D**escent
BiDAF	**Bi**-**D**irectional **A**ttention **F**low
BiGRU	**Bi**directional **G**ated **R**ecurrent **U**nit
BiRNN	**Bi**directional **R**ecurrent **N**eural **N**etwork
DL	**D**eep **L**earning
EM	**E**xact **M**atch
GD	**G**radient **D**escent
GPU	**G**raphical **P**rocessing **U**nit
GRU	**G**ated **R**ecurrent **U**nit
LSTM	**L**ong **S**hort-**T**erm **M**emory
MC	**M**achine Reading **C**omprehension
MT	**M**achine **T**ranslation
NLP	**N**atural **L**anguage **P**rocessing
NLTK	**N**atural **L**anguage **T**ool**K**it
OWQA	**O**ne **W**ord **Q**uestion **A**nswering
POS	**P**art **O**f **S**peech
QA	**Q**uestion **A**nswering
QAS	**Q**uestion **A**nswering **S**ystem
RC	**R**eading **C**omprehension
ReLU	**Re**ctified **L**inear **U**nit
RMSProp	**R**oot **M**ean **S**quare **P**ropagation
RNN	**R**ecurrent **N**eural **N**etwork
Seq2Seq	**Seq**uence **2** **Seq**uence
SGD	**S**tochastic **G**radient **D**escent
SQuAD	**S**tanford **Q**uestion **A**nswering **D**ataset
TREC	**T**ext **Re**trieval **C**onference

Chapter 1

Introduction

1.1 Overview

As human beings, we are naturally driven by an innate desire for knowledge. It is therefore not surprising how we started asking questions to computers as soon as we created them (Jurafsky et al., 2018). The first Question Answering (QA) systems were indeed built in the very early 1960s, such as BASEBALL and LUNAR (Green et al., 1986; Woods, 1978), both able to answer questions strictly relative to their specific domains. Over the decade, the early models evolved into expert systems, which became very popular in the 1970s to answer questions within a specific area of knowledge (Pundge, 2016). In contrast, the architecture of modern QA systems allows them to not be restricted to only a specific area of expertise, but to deal also with more generalised open-domain questions.

These modern systems achieved notable improvements over the years: from IBM's Watson, which won the US television game-show "Jeopardy!" in 2011, surpassing humans at answering the questions in the game (Markoff, 2011), to Facebook Research's DrQA (Chen et al., 2017) and Google AI's BERT (Devlin et al., 2018), current state-of-the-art model for most Natural Language Processing (NLP) tasks, including QA. What these models have in common is being trained over incredibly large natural language text corpora, mostly using Wikipedia as knowledge source (Chen et al., 2017). The existence of such extensive datasets is one of the main reason why those advancements were possible (Chollet, 2018). This is what makes SQuAD (Rajpurkar et al., 2016), the NLP dataset object of this paper's speculation, absolutely invaluable.

1.2 Motivation

Despite the incredible improvements of AI applications in Natural Language Processing (NLP) tasks such as Question Answering, we are still far away from being able to reproduce human-level Reading Comprehension skills using AI. As a matter of fact, Machine Reading Comprehension, the NLP topic tackled in this project, is an AI-Hard problem (Yampolskiy, 2013). This is not surprising, as understanding natural language and extracting meaningful information from it is indeed a purely perceptual task. And as Moravec's paradox says, "*it is comparatively easy to make computers exhibit adult level performance on intelligence tests or playing checkers, and difficult or impossible to give them the skills of a one-year-old when it comes to perception and mobility*" (Moravec, 1988).

The challenge presented by NLP tasks and its importance in the advancement of AI towards building a multipurpose intelligence is what makes it extremely fascinating. Understanding natural language would mean understanding humans and our prime, innate way of expressing ourselves, which despite having evolved in many different forms, has always characterised us. Such Question Answering models not only have a significant research value for the field of NLP, but also have many practical applications, such as automated customer service and virtual assistants.

This paper arises as a study on these challenges presented by Machine Reading Comprehension and NLP as a whole. It analyses the approaches used nowadays to solve MC Question Answering tasks, firstly from a historical and theoretical point of view, to then move to the direct implementation of a few selected models.

1.3 Structure of the Report

This paper opens with **Chapter 1**, an introduction to Question Answering and its role in Natural Language Processing; this chapter presents an overview of the topic and the motivation behind this research. **Chapter 2** states the overall aim of this study and the research and implementation objectives that need to be met.

The research findings and results are explained in **Chapter 3**; this particular chapter is divided in two main sections: *Literature Review* and *Algorithms and Model Architectures*. The former is a review of the history of Question Answering from the 1950s to the present, with an emphasis on modern approaches to QA; the latter, instead, is a report of the algorithms and model architectures that have been examined for the fulfilment of this paper's objectives.

The core of the implementation process is presented in **Chapter 4**, divided in three main sections: *Methodology*, an overview of the methodology used for the implementation, the problem statement and the metrics used for model evaluation; *Data Engineering*, which explains how the data was preprocessed, and *Model Architectures*, in which the structure of the models is described.

This paper comes to a conclusion with an evaluation of the results obtained throughout the paper in **Chapter 5**, and finally the conclusion in **Chapter 6**.

Chapter 2

Research Aim and Objectives

The purpose of this paper is to investigate and compare Deep Learning approaches to the task of Machine Reading Comprehension on the Question-Answering dataset "SQuAD", and evaluate the impact of Attention Mechanisms in their performance.

The fulfilment of the overall aim needs to be achieved by exploring the field in question over two different aspects:

- An extensive research, in order to have a comprehensive understanding of Attention, the field and the main mechanisms involved in it;
- The implementation of selected models, with and without Attention Mechanisms, in order to gain further insight into the effects of Attention and be able to directly compare their performance.

On one hand, the research to be conducted aims at addressing the following points at issue:

- What is a QA system? How did QA systems evolve overtime?
- What datasets are the most popular to train QA models, and why?
- What are the latest/best approaches so far? How are they implemented?
- What is Attention?
- Are Attention Mechanisms a valid solution to Machine Reading Comprehension problems, and why?

On the other hand, the models to be implemented and compared are the following:

1. Model 1 - Word Embeddings (Baseline model)
2. Model 2 - Recurrent Neural Networks (RNNs)
3. Model 3 - RNNs with Attention Mechanisms
4. Model 4 - R-Net Model (Wang et al., 2017)

Chapter 3

Background Research

3.1 Literature Review

3.1.1 Overview: NLP, QA, and MC

Natural Language Processing (NLP), sub-field of Computer Science and Artificial Intelligence, is a theory-motivated range of computational techniques concerned with automating the analysis and representation of natural (human) languages (Young et al., 2018). NLP enables computers to perform a wide range of natural language related tasks, such as natural language understanding, speech recognition, and natural language generation. In particular, natural language understanding deals with all Machine Reading Comprehension (MRC or MC, "Machine Comprehension") tasks, in which the machine is taught to comprehend text. Question Answering (QA), for example, is a Machine Comprehension task.

3.1.2 Question Answering Systems

A Question Answering (QA) system is a system that, given a specific natural language question, is able to automatically find an answer, in a corpus or in the Web (Jousse et al., 2005). As illustrated in Figure 3.1, QA systems are more sophisticated and complete, compared with search engines such as Google or Yahoo, which usually lack in understanding the intent of a question and are able to return only query-relevant documents. QA systems provide the users with a more direct and precise access to the desired information, and leverage human-machine interactions that support natural language.

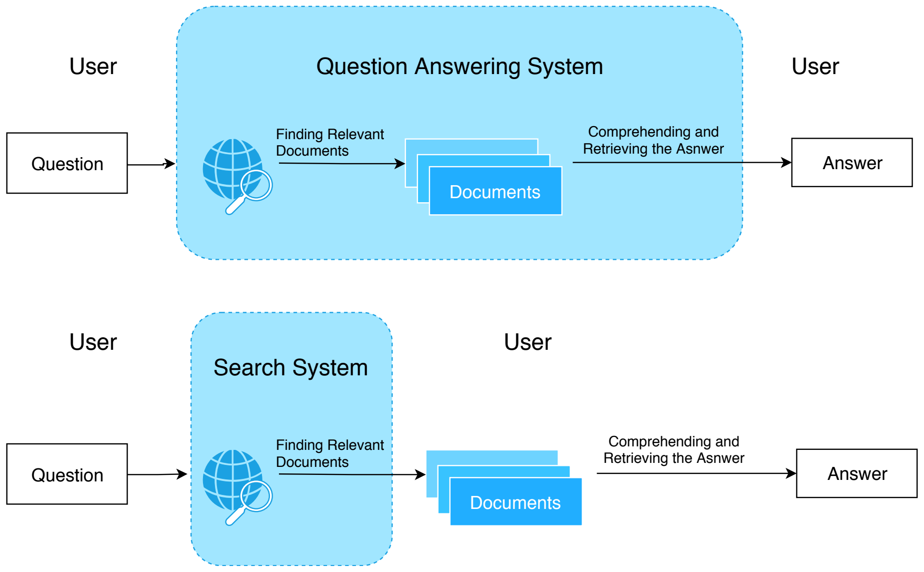

User
Question Answering System
User
Question
Finding Relevant Documents
Documents
Comprehending and Retrieving the Asnwer
Answer
User
Search System
User
Question
Finding Relevant Documents
Documents
Comprehending and Retrieving the Asnwer
Answer

Figure 3.1: Search Systems VS Question Answering Systems

As described in (Jousse et al., 2005), a QA system performs the three following steps: question analysis, information retrieval and answer extraction. Initially, the question is analysed and translated into the form of an abstract concept that the machine can interpret and use as a basis for answer retrieval; in most cases, the question is classified into a category. The information retrieval task consists in a filtering process to gradually reduce the scope of the corpus within the relevant documents containing the information. Finally, based on the retrieval information, an answer is extracted among the possible candidates after calculating a confidence score.
This paper will focus on this last step, answer extraction, as it has been the major point of interest within the QA research field over the past decade. Modern Neural Network QA models are indeed trained on large dataset in which the passage has already been selected from the documents relevant to the the query. Performing answer extraction tasks is what Reading Comprehension techniques are about.

3.1.3 Early Question Answering Systems

The first idea of a QA system was undoubtedly used in the Turing Test, developed by Alan Turing in 1950, with the aim of testing a machine's ability to exhibit intelligent behaviour similar to that of a human. In his 1950 paper, "Computing Machinery and Intelligence"(Turing, 1950), he proposed what he called "The Imitation Game" (later renamed "The Turing Test"), in which the machine was required to generate human-like responses to a series of natural language questions asked by human testers. A human evaluator would then judge the answers: if the evaluator could not reliably distinguish the machine's response from the human's one, the machine was said to have passed the test.

Naturally, early QA systems such as Turing's and the ones subsequently developed between the 1960s and 1970s were mainly restricted to a limited domain and to the technologies available at the time. Projects such as BASEBALL and LUNAR (Green et al., 1986, Woods, 1978) were indeed able to only process structured data regarding a very restricted scope, due to the limitation of intelligent technologies and quantity of data (AliCloud, 2017). It was after the rise of the internet and the development of Natural Language Processing techniques, that QA systems started being able to deal with open-domain questions. Examples of this development stage were QA retrieval systems such as Ask Jeeves [1] and START [2], in which the processing flow mainly consisted in question analysis, document and passage retrieval, candidate answer extraction, and answer validation (AliCloud, 2017).

Despite the development in this research field seen between the 1960s and 1970s, Question Answering systems started becoming popular only in the late 1990s. This progress was mainly promoted at the Text Retrieval Conference (TREC) held in 1999, where the Question Answering Track (QA Track - Voorhees et al., 1999) was introduced and the first large-scale evaluation of such systems took place.

3.1.4 The "Data Revolution"

"*If Deep Learning is the steam engine of the AI revolution, then data is its coal: the raw material that powers our intelligent machines, without which nothing would be possible*"(Chollet, 2018). The rise of the internet made it possible to collect and distribute very large datasets which played an essential role in the progress of QA and Reading Comprehension research as well as Machine Learning in general. Community Question Answering websites, for example, became a stable and reliable source of question-answer pairs derived from real user interactions (AliCloud, 2017); also Wikipedia became a key dataset for the field of Natural Language Processing (Chollet, 2018).

This proper "Data Revolution" allowed the development of Machine Learning and its applications in NLP. The advantages of introducing such techniques are obvious: the building of hard-coded systems used for natural language tasks were long and tedious, in particular in a field in which the structure of the system needs to adapt to some specificities, such as the domain or the language used. Hard-coded patterns are indeed restricted to one language and (most of the time) domain specific (Jousse et al., 2005); being able to learn these patterns automatically using Machine Learning and, successively, Deep Learning, was a breakthrough in NLP.

[1] `http://www.ask.com`
[2] `http://start.csail.mit.edu`

Existing datasets are mainly divided into two categories, according to whether they are manually labelled or not (Wang et al., 2017). The automatically labelled datasets can be very large, a very useful feature for training complex models, but often come in cloze style (Hermann et al., 2015; *The Children's Book Test* - Hill et al., 2016). On the other hand, manually labelled datasets (Berant et al., 2014; *MCTest* - Richardson et al., 2013; *WikiQA* - Yang et al., 2015) are always in high-quality, but often too small in size for modern models.

However, datasets such as SQuAD(Rajpurkar et al., 2016) and MS MARCO (Nguyen et al., 2016) are different from the ones listed above, as they are manually labelled but both large and high-quality.

The SQuAD Dataset

The Stanford Question Answering Dataset (SQuAD) is a large and high-quality Reading Comprehension dataset. It is based on 536 Wikipedia articles with 107,785 crowdsourced question-answer pairs. In contrast to other datasets, SQuAD requires the machine to select the answer from all possible spans in the paragraph ('context'), instead of providing a list of answer choices for each question. The answers in SQuAD often include non-entities and can be much longer phrases, which is more challenging than cloze-style datasets (Wang et al., 2017). Thus, this results in the system having to cope with a large number of candidates for the answer, and allows the dataset to feature a diverse range of question-answer types; Rajpurkar et al. also show that SQuAD requires different forms of logical reasoning, including multi-sentence reasoning.

3.1.5 Modern Approaches to QA

For a long time, core NLP techniques were dominated by machine learning approaches that used linear models such as support vector machines (Zhang et al., 2003), conditional random fields (Yao et al., 2013), and logistic regression, trained over very high dimensional yet very sparse feature vectors (Goldberg, 2016). However, these QA systems, based on shallow models, still featured high artificial dependency, lacking data representation and generalisation capability for data processing in different fields (AliCloud, 2017).
More recently, Neural Networks architectures started to be applied to natural language and have superseded linear, shallow models, yielding outstanding state-of-the-art results. Since 2015, in particular, several powerful deep learning models were introduced to solve Machine Reading Comprehension (Wang et al., 2017). It is not a coincidence that Attention was introduced into RC problems in that same year by (Hermann et al., 2015), to favour the interaction between questions and passages. Attention Mechanisms were actually firstly applied in Recurrent Networks for the NLP task of Machine Translation (MT) by (Bahdanau et al., 2014), as an improvement of Sequence2Sequence (Seq2Seq) models, revolutionising Neural Network approaches to the problem.

The performance improvements reached with the use of Attention have been so significant that it has become "*the new vanilla*" in many NLP tasks such as MT, QA and RC (Genthial et al., 2019). As a matter of fact, modern Neural Network-based state-of-the-art models for Reading Comprehension use variants of Attention: (Seo et al., 2016) used a "*Bi-directional Attention Flow*" (BiDAF) mechanism, which combines a "*context-to-question*" Attention with a "*question-to-context*" Attention to match questions and contexts mutually, obtaining a query-aware passage representation. (Wang et al., 2017) of Microsoft Research Asia, in their "*R-Net*" model, introduced "*self-matching*" Attention, which refines the representation of each passage by matching it against itself, aggregating evidence relevant to the current passage word and question. Most recently, (Devlin et al., 2018) from Google AI Language introduced a model called BERT (Bidirectional Encoder Representations from Transformers), a pre-trained model that uses an Attention mechanism called the "*Transformer*", from the famous Google Brain's paper "*Attention Is All You Need*" (Vaswani et al., 2017). BERT pre-trains deep bidirectional representations by "*jointly conditioning on both left and right context in all layers*", reaching human-level performance with 93.2% F1 score on the SQuAD dataset (Devlin et al., 2018).

Summary

This study showed how the Question Answering problem has had a great interest since the 1960s, when Artificial Intelligence was still at its earliest stages. Advancements in AI and the introduction of Neural Networks in the field brought invaluable improvements in the performance of Question Answering systems in Reading Comprehension tasks. After the introduction of Neural Network architectures and RNNs in particular, the main great innovation was shown to be Attention Mechanisms, introduced as an extension to RNNs in 2015. All modern state-of-the-art models use Attention variations in their architectures. This research will continue in the next section with a study on Natural Language Processing Deep Learning techniques, focusing on the recently introduced Attention Mechanisms. One of the aforementioned state-of-the-art models, R-Net, will be also described in detailed to then be implemented in Chapter 5.

3.2 Algorithms and Model Architectures

3.2.1 Gradient Descent

Gradient Descent (GD) is by far the most common algorithm to optimise neural networks. It aims at minimising an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$ by repeatedly updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ with respect to the parameters (Ruder, 2016). In other words, the minimisation of the objective function (the loss function) is obtained by taking steps in the negative direction of the function's gradient. The size of the steps taken to reach a (local) minimum is determined by the learning rate η. The amount of data used in order to compute the gradient of the function creates a trade-off between the accuracy of the parameter update and the time taken to perform an update. For this reason, three variants of Gradient Descent can be distinguished according to the amount of data used in each update: Batch GD, Mini-Batch GD and Stochastic GD.

Batch Gradient Descent

Batch Gradient Descent (BGD), the most common form of GD, performs the updates by calculating the gradient of the cost function for the entire training dataset each time:

$$\theta = \theta - \eta \times \nabla_\theta J(\theta)$$

Running every iteration on all the data available allows this approach to be more accurate: Batch Gradient Descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces (Ruder, 2016). However, it is often very time-inefficient for large datasets and it causes carrying out redundant computations.

Stochastic Gradient Descent

In order to overcome the computational redundancy of Batch Gradient Descent, a different method was proposed, Stochastic Gradient Descent (SGD). In this variant, a single parameter update is performed with only one training sample-target pair $x^{(i)}$ and $y^{(i)}$ (batch size of 1):

$$\theta = \theta - \eta \times \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

The term "Stochastic" refers to the fact that the sample-target pair is selected randomly (Chollet, 2018). Due to its stochastic nature, the path created by SGD towards the global minimum is not as "direct" as in BGD, but it rather "jumps" to different locations and, possibly, a potentially better local minima. This may seem to cause SGD to keep overshooting and prevent the convergence to the global minimum. However, it has been shown that when slowly decreasing the learning rate, SGD shows the same convergence behaviour as BGD, and it almost certainly converges to a local or global minimum for non-convex and convex optimisation respectively (Ruder, 2016).

Mini-Batch Gradient Descent

Mini-Batch Gradient Descent is a middle ground between BGD and SGD, combining the best characteristics of both variations. It performs an update for every mini-batch of n training samples:

$$\theta = \theta - \eta \times \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

This variant allows a more stable convergence, as it reduces the variance of the parameter updates, and it is also less expensive than BGD. The term SGD is usually employed for this optimisation algorithm, as it is the most typically used when training a neural network and the most used in deep learning libraries such as Keras.

3.2.2 Activation Functions

Activation functions are essential components of Neural Networks, as they determine the output behaviour of each neuron and introduce non-linear properties to the network. Non-linearity allows to get access to a much richer hypothesis space. Without activation functions, the network would consist in a set of linear operations, which means that each layer would only able to learn linear transformations of the input data. Such a hypothesis space would be too restricted. Some popular activation functions are ReLU (Rectified Linear Unit), Sigmoid, Tanh and Softmax.

ReLU

A ReLU (Rectified Linear Unit) function zeros out negative values. The formula is simple: $max(0, z)$.

Sigmoid

The Sigmoid activation function takes a real-valued input and coverts it into a value between 0 and 1. As it is within the interval [0, 1], its output can be interpreted as a probability.

Tanh

Like Sigmoid, Tanh “squashes” a real-valued input to an interval, [-1, 1]. Its output, however, is zero-centred. For this reason, the Tanh non-linearity is often preferred to the Sigmoid non-linearity.

Softmax

Softmax is used as the activation function of the outputs of all the models described in this paper. It is mainly used in Multiclass Classification problems, as it outputs a probability distribution of each target class over all possible target classes.

3.2.3 Text Vectorisation

One Hot Encoding

One hot encoding is the most basic way to turn a word into a vector (Chollet, 2018). Each word ("token") is associated to a unique integer i, and then converted into a binary tensor of the same size of the vocabulary. This vector is filled with zeros, except for the i^{th} entry, which is 1.

Word Embeddings

Word Embeddings are one of the most common ways of vectorising text sequences into numeric tensors. Their popularity depends on many factors. Firstly, the vectors created by Word Embeddings are dense, low-dimensional floating-point vectors: they allow to efficiently convert large text sequences into smaller, condensed representations. Secondly, as Word Embeddings are learned from the data, they are able to map human language into a geometric space: the geometric relationships between word vectors reflect the semantic relationship between the words they represent (Chollet, 2018).

There are two ways for obtaining word embeddings:

- Learn Word Embeddings during training, within the model itself. This is possible using the Keras `Embedding` layer. As they are learned directly from the data in question, they often overcome the performance of pre-trained word embeddings.
- Use pre-trained Word Embeddings, such as GloVe[3] or Word2Vec[4]. These Word Embeddings, precomputed using a different machine learning task, are very useful with small datasets, as the data available is not enough for training an `Embedding` layer.

[3] `https://nlp.stanford.edu/projects/glove/`
[4] `https://code.google.com/archive/p/word2vec/`

3.2.4 Recurrent Neural Networks

Biological intelligence processes information incrementally while maintaining an internal model of what it is processing, memorising past information and constantly updating as new information comes in (Chollet, 2018). Recurrent Neural Networks (RNNs) (Elman, 1990) are based on the same principle: they process sequences by iterating through each element, while maintaining a *state* with information about what it has seen so far. In this way, they are able to capture the sequential and ordered nature of natural language, in which the semantic meaning of words depends also on previous and subsequent words in the sentence.

In order to *memorise* the elements already encountered in the sequence, the RNN architecture does not process each data point in a single step; rather, the network internally loops over sequence elements. At each timestep t, it considers both the current input and the state obtained from the previous input at the previous timestep, and combines them to obtain the output at timestep t. Specifically, given an input to the network x_t at timestep t, its hidden state representation s_t is calculated as:

$$s_t = f(Wx_t + Us_{t-1} + b)$$

where the function f is non-linear activation function (commonly tanh or ReLU[5]) and b is the bias vector. The output at position t is the same as the hidden state in that position (Goldberg, 2016).

Theoretically, the simple RNN structure just described above should retain at timestep t, in its last hidden state, all the information seen from all the previous timesteps. However, in practice, such long-term dependencies appear impossible to learn, due to the *vanishing gradient problem* (Chollet, 2018). This problem was overcome with the development of more complex networks such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), which are the most used RNNs in NLP applications. As often in Machine Learning, there is a trade-off between computational expensiveness and representational power: on one hand, GRU layers, although they use the same principles as LSTM, are cheaper to run; on the other hand, LSTM layers have more representational power than GRU.

[5]See Section 3.2.2: *Activation Functions*

3.2.5 Attention Mechanisms

Attention: the Concept

As seen above, RNNs are able to learn sequential data by having an internal memory of all its past inputs. However, they have a limitation: when building the memory, all data points are treated equally. (Bahdanau et al., 2014) noticed this flaw while applying Sequence2Sequence models in Machine Translation. Unlike RNNs, when people read a sentence such as “The cat is on the mat”, we process all the words, but focus on them to different extents. We assign more significance to the words “cat”, “on”, “mat”, as they are the key words for understanding the meaning of the sentence. Neural networks can achieve this same behaviour using Attention Mechanisms.

“*Given a set of vector values, and a vector query, Attention is a technique to compute a weighted sum of the values, dependent on the query*”(Socher et al., 2019)

In other words, it allows the model to “*focus the attention*” on selective parts of the input values (in QA, the passage), obtaining a new representation of those based on how much they relate to the given query (the question).

Attention: Equations

Given an encoder (passage) hidden states $h_1, ..., h_N \in \mathbb{R}^h$ returned from a recurrent network, and a decoder (query) hidden state $s_t \in \mathbb{R}^h$, the attention scores e^t on timestep t can be calculated as the dot product between the decoder hidden state at timestep t and each encoder hidden state:

$$e^t = [s_t^T h_1, ..., s_t^T h_N]$$

To get the attention distribution α^t along the encoder states (a probability distribution that sums to 1) for this step, the softmax of the attention scores is taken:

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

The final attention output a^t can then be obtained by simply using the attention distribution α^t to take a weighted sum of the the encoder hidden states:

$$a_t = \sum_{i=1}^{N} a_i^t h_i$$

Attention: an Example

Given an example passage:

"*dan lives with his pets.*
the dog plays in the garden.
the cat is on the mat.
the hamster runs in the wheel.
the cat is sleeping."

and a question:

"*where is the cat?*",

Figure 3.2 illustrates an example of a soft alignment matrix, representing the correspondence between the words in the passage and question. In particular, it is a hypothetical visualisation of the attention distribution (α^t, as referred to in the formulas above) that would be obtained from the hidden states of the context and question in this example. The colour shade indicates the relation between the words in the table: the higher the intensity of the blue in a cell, the more relevant the correlation between the corresponding row and column. In this alignment matrix, the correlations are based on the semantic relationships between the words. Same words (i.e. "cat" and "cat", "is" and "is") are highly related; nouns are moderately related with other nouns, verbs with verbs; the word "where" is highly correlated with prepositions ("in", "on") and moderately with nouns indicating places ("garden","mat"), and so on.

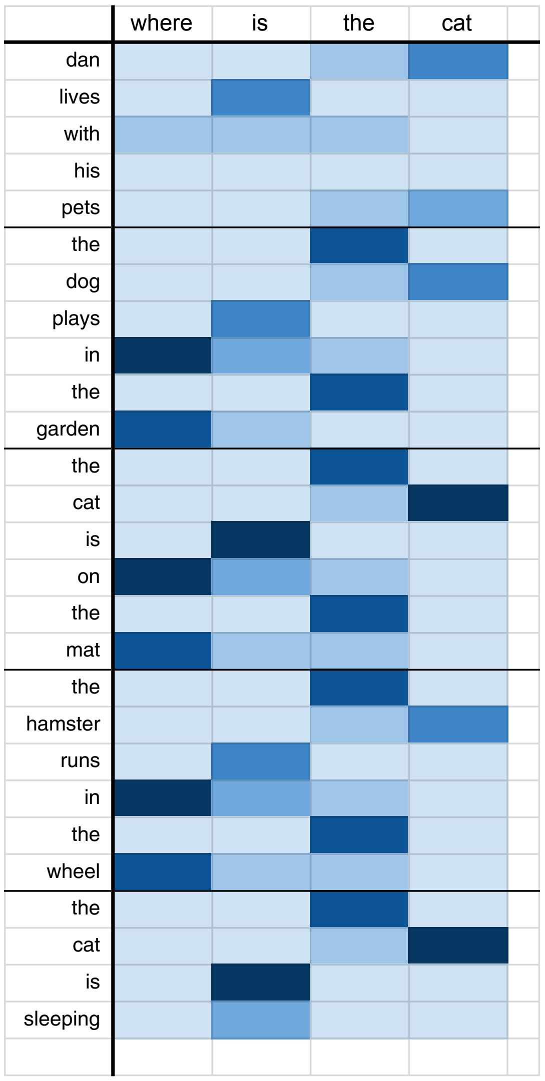


Figure 3.2: Example of alignment matrix between a context and query

This probability distribution from the matrix is used in the weighted sum of the passage hidden states to weight each state accordingly to their relevance to the query. In this case, the query word "where" would concentrate the focus (have an higher weight/score) on prepositions and nouns indicating places; the word "the", appearing often and uniformly

in every sentence, would not particularly affect the weights nor highlight any particular part of the passage; in contrast, the query word "cat" would strongly highlight the places in the context that mention the word "cat", and so on.
The attention output therefore returns the passage itself, but in a new, question-aware representation.

Attention: Variants

Attention Mechanisms always involve the steps seen above:

1. Computing the attention scores e;
2. Taking softmax to get the attention distribution α;
3. Using attention distribution to take the weighted sum of the passage states with the attention distribution, to obtain the attention output a.

However, there are multiple ways in which the attention scores e can be computed. Based on the formula to obtain e, it is possible to divide several Attention variations (Socher et al., 2019):

- Basic dot-product Attention (the one previously described): $e_i = s^T h_i$;
- Multiplicative Attention: $e_i = s^T W h_i$, which considers also a weight matrix W;
- Additive Attention: $e_i = v^T \tanh(W_1 h_i + W_2 s)$, where W_1 and W_2 are weight matrices and v is a weight vector. This variant is one of the most commonly used, and it will be discussed in details when explaining the R-Net structure in Section 3.2.6;
- Self-Attention (also called Intra-Attention): variant of Additive Attention, in which the two inputs consist of the same sequence: it relates different positions of the sequence in order to compute a representation of itself.

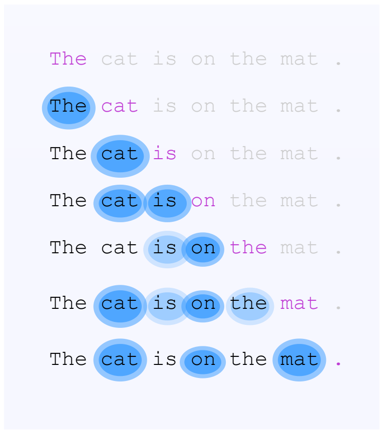


Figure 3.3: Self-Attention example

Disclosure

This research on Attention Mechanisms was possible thanks to the online lectures from Stanford University master program's CS224n course, *"Natural Language Processing with Deep Learning"* (Socher et al., 2019).

3.2.6 R-NET Model

R-NET (Wang et al., 2017) is an end-to-end neural networks model for reading comprehension style question answering, developed by the Natural Language Computing Group of Microsoft Research Asia. It has been tested on the SQuAD (Rajpurkar et al., 2016) and MS MARCO (Nguyen et al., 2016) datasets, reaching state-of-the-art results on SQuAD: 72.3 EM score and 80.6 F1 score, demonstrating to have the best performance among single models, as of August 25, 2017 (Wang et al., 2017).
As illustrated in the diagram in Figure 3.4, the model mainly consists in four parts:

1. Question and Passage Encoding, in which the questions and contexts are preprocessed and encoded separately;
2. Question-Passage Matching, where a question-aware representation for the contexts is obtained;
3. Passage Self-Matching, where self-matching attention is applied on the passage to get its final representation;
4. Answer Prediction, where the interval containing the answer is predicted.

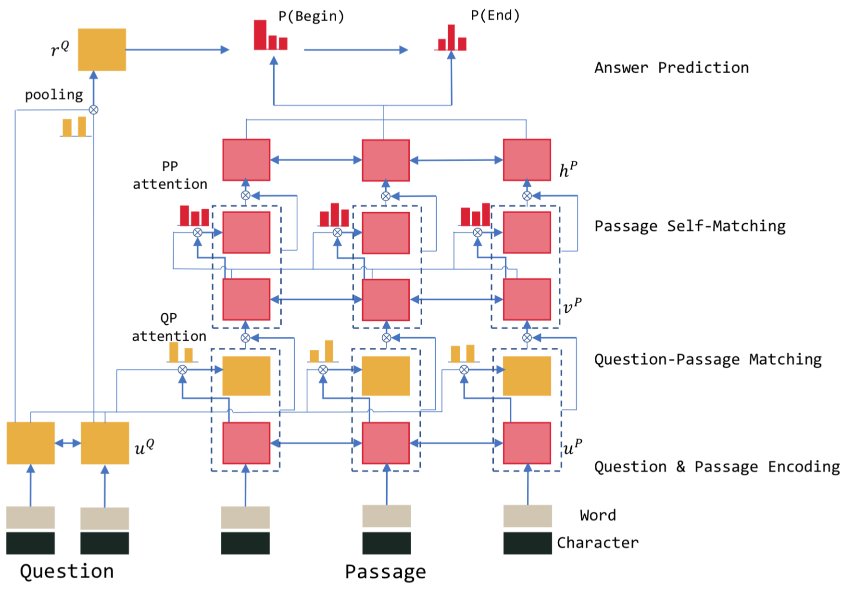


Figure 3.4: R-NET structure overview. From the official R-NET paper (Wang et al., 2017)

Disclosure

This description was elaborated from the original R-Net paper (Wang et al., 2017) and the work of (Mirakyan et al., 2017) in their blog post[6].

Question and Passage Encoding

In this step, each passage $P = \{w_t^P\}_{t=1}^n$ and question $Q = \{w_t^Q\}_{t=1}^m$ are converted into their respective word-level embeddings ($\{e_t^P\}_{t=1}^n$ $\{e_t^Q\}_{t=1}^m$) and character-level embeddings ($\{c_t^P\}_{t=1}^n$ $\{c_t^Q\}_{t=1}^m$). The character-level embeddings, useful to deal with out-of-vocab tokens, are generated by first parsing each character into an Embedding layer; then, a Bi-directional RNN runs over those embeddings to produce a vector for the word. The process is repeated for each word. The embedded contexts and questions are then encoded using a Bi-directional RNN, to produce the new representations $u_1^P, ..., u_n^P$ and $u_1^Q, ..., u_n^Q$.

Question-Passage Matching

In order to obtain a question-aware representation for the passages, the encoded passages and questions are input into a gated attention-based recurrent network layer, a variant of Additive Attention-based RNNs. Figure 3.5 illustrates the operations each RNN cell of this layer. Each cell has three inputs, characterised by a different colour:

- The previous GRU state (v_{t-1}^P), in violet;
- A matrix representation of the question (u^Q), in light blue;
- A vector representation of the passage at the (u_t^P) at the t^{th} word, in blue.

Firstly, the dot products between each input and their respective weight are calculated, and their results summed together:

$$s_t = v^T \tanh(W_u^Q u^Q + W_u^P u_t^P + W_v^P v_{t-1}^P) c_t = u^Q \text{softmax}(s_t)$$

As the outputs of these dot products have not all the same shape, to allow them to be summed together, the two vectors are broadcasted (repeated several times) into the same shape as the question matrix. The result of the summation is parsed into a *tanh* activation function, that is then multiplied (dot product) with an additional weight v^T. At the end of this operations, a *softmax* activation is applied to the output, which produces an attention vector: a vector of probabilities scores indicating the importance of each word of the question. After computing the dot product between the question u^Q and the attention vector, the output c_t is obtained. This output is a new representation of the question, which highlights the parts of the question that are relevant to the current word of the passage,

[6]`http://yerevann.github.io/2017/08/25/challenges-of-reproducing-r-net-neural-network-using-keras`

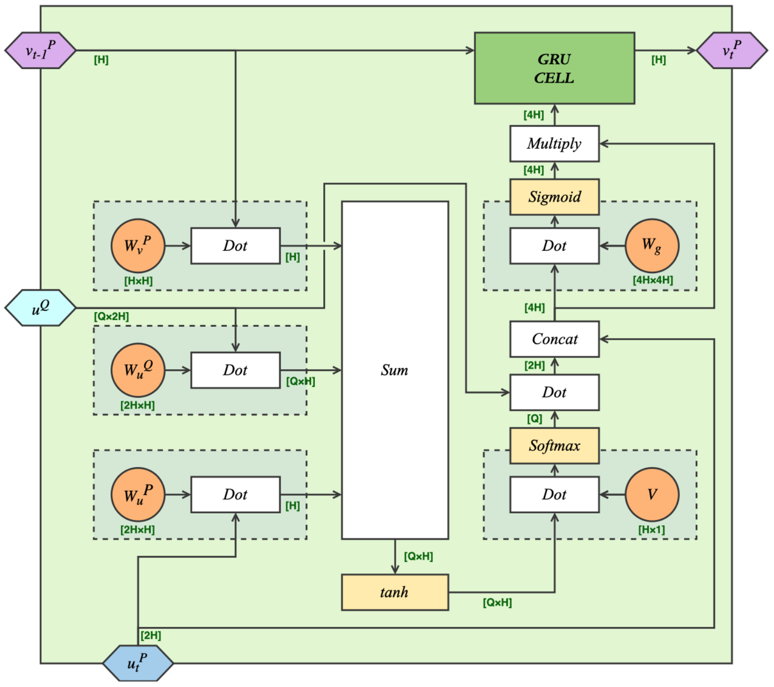


Figure 3.5: Detail of a cell of the gated attention-based RNNs. Illustration taken from (Mirakyan et al., 2017)

based on the current word, the whole question and the previous state of the recurrent cell. An additional gate consisting of another dot product is then applied to the concatenated vector, before returning the final state t of the RNN cell.

Passage Self-Matching

After the question-passage matching, the R-Net authors apply a self-matching mechanisms on the contexts. The self-attention module attempts to augment the passage vector representations with information from other relevant parts of the passage (Mirakyan et al., 2017). Each self-attention cell takes as inputs the current word vector along with the entire context matrix. The structure of the cell is then the same as the one described above in the previous step. This time, however, each word of the passage is matched with the entire passage matrix.

Answer Prediction

Finally, the start and end positions of the answer are predicted. The encoded question vector from step 2 (question-passage matching) creates the first hidden state , while the selfmatched passage from step 3 is the input of the pointer network, a variation of the sequence-to-sequence model with attention. The attention mechanism is utilised as a pointer to select the start position and end position from the passage.

Chapter 4

Implementation

The code for this project implementation can be found in this public GitHub repository: `https://github.com/elemar03/SQuAD_question_answering`

This Chapter starts by firstly explaining the methodology used in the implementation process, for then illustrating the first stage of the project: *Data Engineering*. It then moves to the description and the implementation process of the 4 models selected within the research aims[1]. For each model, the chapter focuses on their structure, how their parameters have been tuned, and the results obtained. These models were all applied to two different tasks: a "QA task" on the whole SQuAD dataset, and a second task, which will be referred to as "OWQA task" (One Word Question Answering). For this OWQA task, only a subset of SQuAD dataset was considered, containing only one-word answers.

4.1 Methodology

The model architectures were build in Keras[2] [3], a high-level neural networks API written in Python running on top of TensorFlow[4] as backend. The models were trained and tested on a GPU *GeForce GTX 1080* using Jupyter Notebooks[5] and TensorBoard, a TensorFlow browser-based visualisation tool.
In order to build these models, the methodology used was consistent with the steps described by F. Chollet in the "Universal Machine-Learning Workflow" (Chollet, 2018):

1. Define the problem

[1]See Chapter 2:*Research Aims*.
[2]Keras Version: 2.2.4.
[3]`https://github.com/keras-team/keras`
[4]`https://tensorflow.org`
[5]`https://jupyter.org`

2. Identify a method of evaluation to measure success and performance
3. Preprocess and vectorise the text corpus in the dataset in order to be able to feed it into the model
4. Establish a baseline model
5. Develop a model able to beat the baseline
6. Gradually refine the model architecture by using hyperparameter tuning and regularisation
7. Repeat steps 5 and 6 on different, increasingly powerful models in order to achieve results and be able to compare their performance

4.1.1 Problem Definition

QA Task

Given a word sequence passage $P = \{p_1, p_2, ..., p_N\}$ with length N, and a word sequence question $Q = \{q_1, q_2, ..., q_M\}$ with length M, a model is required to learn a function $f : (p, q) \to \{a_s, a_e\}$, with the condition $1 \leq a_s \leq a_e \leq N$. The labels $\{a_s, a_e\}$ are a pair of scalar indices pointing respectively to the start position and end position of the answer to the question q in the context p.

OWQA Task

Given a word sequence passage $P = \{p_1, p_2, ..., p_N\}$ with length N, and a word sequence question $Q = \{q_1, q_2, ..., q_M\}$ with length M, a model is required to learn a function $f : (p, q) \to \{a_i\}$, with the condition $1 \leq a_i \leq N$. The label $\{a_i\}$ is a scalar index pointing respectively to the position of the answer to the question q in the context p.

4.1.2 Performance Evaluation Methods

The organisers of the SQuAD dataset established two different metrics in order to evaluate the performance of a QA system on the benchmark: the Exact Match (EM) metric and the F1 score. For the QA task, each model was evaluated by these two metrics. For the OWQA task instead, EM, Top2-EM and Top3-EM scores were used.

EM Score

The Exact Match (EM) metric measures the percentage of predictions that match the ground truth answers exactly. Given a ground truth answer and a prediction, 1 is returned if answer and prediction are exactly the same, 0 otherwise.

F1 Score

The F1 score is a looser metric in respect to EM; it measures a model's overall accuracy by combining precision and recall scores. It is calculated as $F1 = 2 \times \frac{precision \times recall}{precision + recall}$. The F1 score ranges from 0 (lowest) to 1 (highest); a good F1 score for a model means having low predictions of false-positives and false-negatives.

Top2-EM Score and Top3-EM Score

As the dataset of OWQA is formed by only one-word answers, the F1 score cannot be applied to it. The predicted answers can only either correspond the the right or the wrong answer, and therefore the EM score is the only reliable performance indicator. For this reason, "Top2-EM" and "Top3EM", two variants of the EM score, were created exclusively in this project for the purpose of evaluating the model performance on the OWQA task. Instead of considering only the *most probable* answer chosen by the model, these scores consider the top-2 and top-3 most probable answers, respectively. In this way, when a near miss occurs, it can be observed. If someone were to guess a person who their friend met today, they would create a list of most probable people they could have seen. Based on their knowledge, they would likely be left with a few choices that have no clear winner. If their guess is not right at the first attempt, it does not mean that the right person is not among their list at all. Similarly, if the model did not find the right answer with its first guess, it still might have considered the ground truth answer. These scores, although not official, proved to be helpful in understanding the reasoning behind the different models implemented in this project.

4.1.3 Constraints

The models in this project will not aim to be a state-of-the-art model on the task, as current state-of-the-art Machine Comprehension models are outside both the scope and the resources available for this project. In particular, there are some considerations about the challenges this task involves that need to be pointed out:

- All the applications of Deep Learning in NLP and QA are extremely recent. Hence, there is a considerable lack of exhaustive and in-depth materials on the subject, as well as a scarcity of publicly available model implementations;
- Deep Learning QA state-of-the-art models are computationally expensive to run. The resources available for this project were limited: as mentioned before, the models were trained on a GPU *GeForce GTX 1080.*

4.2 Data Engineering

4.2.1 Preprocessing the SQuAD Dataset

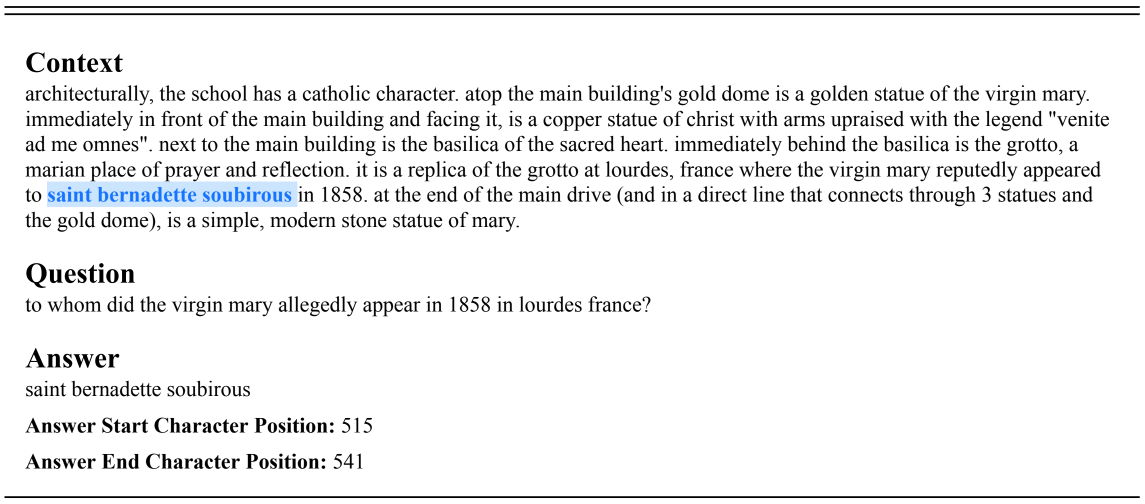
The Data Preprocessing task was the result of a long and complex autonomous analysis of the SQuAD dataset itself and the problem requirements, which needed more time than expected in the project planning. Not only did the dataset require a considerable amount of processing in order to be ready to be fed into a Deep Learning model, but it also appeared hard to find explicit guidelines on the required procedures.
Despite it being the most popular QA dataset, and the many Deep Learning implementations based on it, the SQuAD dataset had barely any useful reference on its required or suggested preprocessing procedure. Every recent publication on SQuAD mostly concentrate on the model structures rather than the preprocessing techniques required for those models to perform well.

4.2.2 SQuAD dataset structure

The SQuAD dataset has a particular nested structure. The hierarchy is the following:

$$data \rightarrow paragraphs \rightarrow context \rightarrow qas \rightarrow question, answer\_text, answer\_start, answer\_end$$

The data itself is a list of 536 articles sampled from Wikipedia, containing in total 23215 paragraphs (Rajpurkar et al., 2016). A single paragraph is referred as *'context'*. Each of these contexts contains a set of 3-to-5 questions about the passage, along with their respective answers. The answers are given both in their text form (`answer_text`), which will not be used directly in the model, and as a pair of indices (`answer_start, answer_end`), representing the start and end location at character level of the answer in the context. An example is shown in Figure 4.1. In the SQuAD dataset, each answer is not openly formulated, but it can always be found inside the corresponding passage. The dataset was already split in a training and development (test) sets by its creators, with 87599 question-answer pairs in the former, and 10570 in the latter.



Context
architecturally, the school has a catholic character. atop the main building's gold dome is a golden statue of the virgin mary. immediately in front of the main building and facing it, is a copper statue of christ with arms upraised with the legend "venite ad me omnes". next to the main building is the basilica of the sacred heart. immediately behind the basilica is the grotto, a marian place of prayer and reflection. it is a replica of the grotto at lourdes, france where the virgin mary reputedly appeared to **saint bernadette soubirous** in 1858. at the end of the main drive (and in a direct line that connects through 3 statues and the gold dome), is a simple, modern stone statue of mary.

Question
to whom did the virgin mary allegedly appear in 1858 in lourdes france?

Answer
saint bernadette soubirous

Answer Start Character Position: 515

Answer End Character Position: 541

Figure 4.1: Example of a question-answer pair for a sample passage from the SQuAD dataset. The answer is a segment of text from the passage.

4.2.3 Importing and structuring the data

The first step in preprocessing the SQuAD dataset consisted in importing the training and development datasets from their respective `.json` files. The two datasets are separately parsed into a function, `import_dataset()`, which has the following functionalities:

- Takes the name of the `.json` file to import as a parameter and reads its content;
- Converts the complex nested structure of the dataset into a python dictionary, easier to read and use;
- If a value is given for the optional parameter `contexts_max_len`, the function discards all the samples in which the context is longer than `contexts_max_len`. This is a key functionality in order to make the program less computationally expensive later.

The data was stored by the creators in a nested structure, in which multiple QA pairs were inside a single context. For development purposes, the data needed to be converted into a data structure which resulted clearer and simpler: five separate lists were created for contexts, questions, answer texts, answer starts, and answer ends. In order for the contexts to match all their questions, each passage in the 'contexts' list was repeated as many times as the number of questions corresponding to it. Finally, those lists were all stored into two Python dictionaries, one for the training and one for the development data. The idea of storing them in a dictionary turned out to be essential during the development of the project, as having all the lists as separate and sparse variables appeared to be confusing and unnecessary: in total, the lists would've been seven for each training/development set. By storing all the lists in a dictionary, they were still easily accessible and also more readable. There was also a clearer distinction between training and dev data.

4.2.4 Tokenization

Like all other Neural Networks, Deep Learning models are not able to process raw text - they only work with numeric tensors (Chollet, 2018). The first step for converting text into vectors is called *Tokenization*, and it consists in breaking down the text into different units called *tokens*. These tokens are obtained by segmenting the text into words, characters or n-grams of words [6].

In this case, the tokenization for the SQuAD dataset was made at word-level, and it was applied to contexts and questions. As shown in the code snippet in Listing 4.1, each sequence (a single context or question) in the list of sequences `text` is tokenized into words using the NLTK tokenization function `word_tokenize`. The choice of using this particular NLTK tokenizer was due to the fact that it splits tokens based on white space and punctuation, without removing punctuation. The tokens are then all converted to lower case to unify the same words. Also, as suggested by (Seo et al., 2016), some replacements are made in order to standardise the quotation marks over the sequences.

Listing 4.1: Tokenization Function

```
def tokenize(text):
    return [[token.replace("‘‘", '"').replace("''", '"').lower()
                      for token in word_tokenize(sequence)]
                           for sequence in text]
```

4.2.5 Mapping the answer location from Character-level to Word-level

However, now that the contexts and the questions are tokenized at word level, a problem arises. The answers, which were encoded as a pair of indices indicating the character where the answer starts and ends in the paragraph, no longer relate to anything in the tokenized version of the contexts. Therefore, the start and end character indices of the answers need to be mapped to word-level indices.
The function `map_char_to_token_index()` is responsible for the generation of new word-level indices. By going through the characters of the tokenized and non-tokenized contexts

[6]N-grams are overlapping groups of N consecutive words (or characters) that can be extracted from a sentence. For example, the sentence ”The cat sat on the mat” can be decomposed in the following set of 2-grams, creating a set which is called a *bag-of-2-grams*: { “The”, “The cat”, “cat”, ‘cat sat‘”, “sat”, “sat on”, “on”, “on the”, “the”, “the mat”, “mat” }.

in parallel, and keeping track of the index of the current token, it is able to find which tokens contain the character index of the answer start and end. The contexts, questions and answers are now all encoded at word-level.

4.2.6 Vectorization

The last step required in order for the data to be ready to be parsed into a Neural Network consists in vectorizing the sequences of tokens. In this process, called *Vectorization*, numeric vectors are associated with the generated tokens. The two main vectorization methods are *One Hot Encoding* and *Word Embeddings* (Chollet, 2018), explained in details in Section 3.2.3: *Word Embeddings*.

Vectorizing Contexts and Questions

Word Embeddings were chosen to encode contexts and questions, due to the following reasons:

- They are more dense and lower-dimensional than One Hot encoding vectors: they pack more information into far fewer dimensions; this was essential given the size of the dataset and the not insignificant length of the contexts.
- They are learned from the data. This allows Word Embeddings to reflect the semantic relationships between words, mapping human language into a geometric space.

Initially, the context and question tokens were parsed into a Keras Tokenizer. The Tokenizer allowed to perform the last steps required in order to then use Word Embeddings:

1. Firstly, the Keras function `fit_on_texts()` builds the word index: a large indexed vocabulary including all the words encountered in the data, where each word has an index assigned to it;
2. Secondly, the method `text_to_sequences()` is used to turn the tokens into lists of integer indices corresponding to them;
3. Finally, all question sequences obtained from the previous steps must be of the same length in order to be parsed into an Embedding Layer. The smaller questions are therefore padded with zeros to the same length of the largest question using the `pad_sequences()` function. The same process applies for the context sequences. The contexts, however, are padded until the maximum value of the answer end: the final parts, not containing any answer, of some the longest contexts are truncated. This slightly improves accuracy and time performance of the model.

The context and question sequences are now ready for Word Embeddings, which will happen in the first part of the models.

Vectorizing the Labels

The labels instead, were One Hot encoded. The implementation is shown in Listing **??** in the Appendix and the result is illustrated in Figure 4.2.

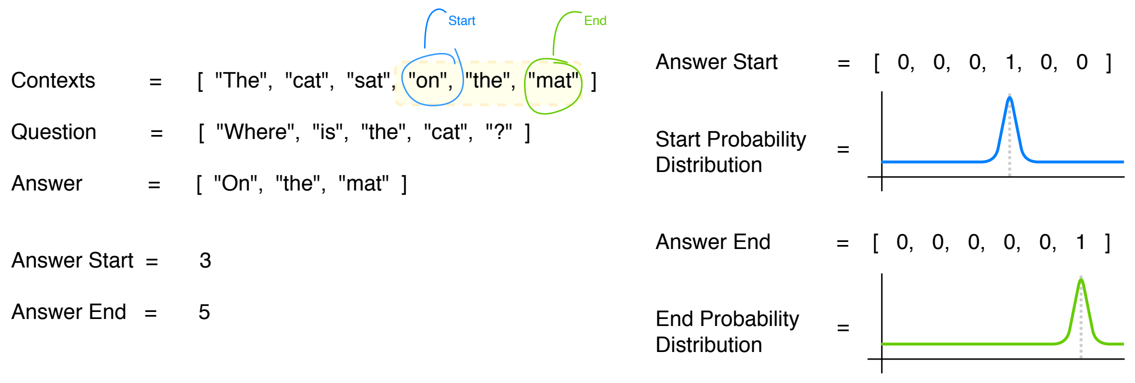

Start
End
Contexts = ["The", "cat", "sat", "on", "the", "mat"]
Question = ["Where", "is", "the", "cat", "?"]
Answer = ["On", "the", "mat"]
Answer Start = 3
Answer End = 5
Answer Start = [0, 0, 0, 1, 0, 0]
Start Probability Distribution =
Answer End = [0, 0, 0, 0, 0, 1]
End Probability Distribution =

Figure 4.2: Example of dataset structure and labels before (left) and after (right) One-Hot encoding

Before One Hot encoding, the labels are in the form of two indices, indicating the start and end of the answer in the context (Figure 4.2, on the left). For each data point, these two indices are transformed into two sequences of the same length of the contexts, and filled with zeros. The first sequence, representing the answer start index a_s_i, will have a 1 at $a\_s_i^{th}$ term; similarly, the second sequence, representing the answer end index a_e_i, will have a 1 at $a\_e_i^{th}$ term.

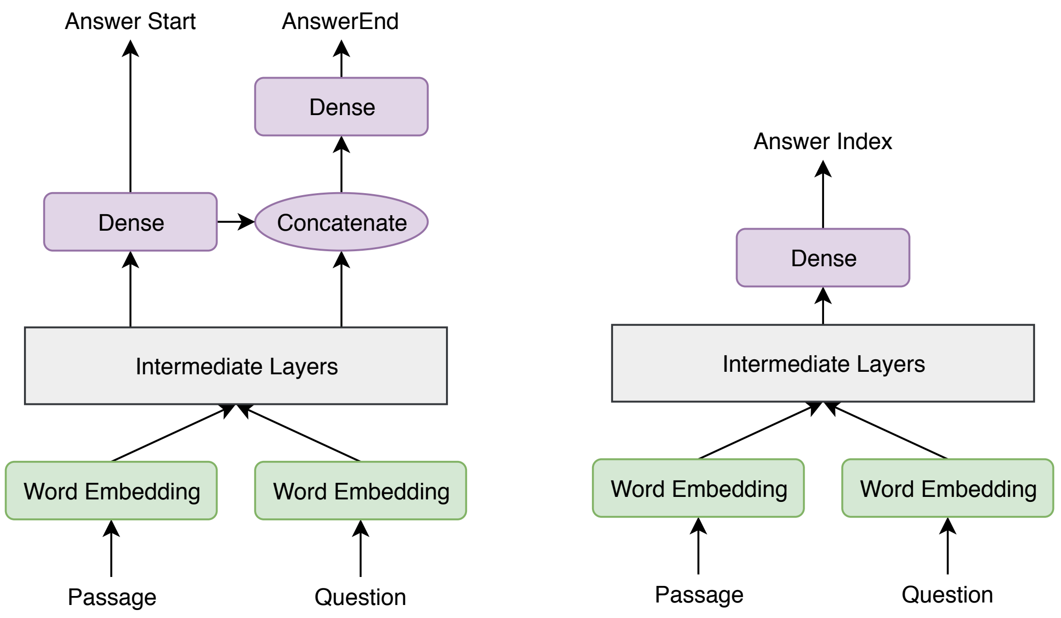
In this way, the structure obtained by One Hot encoding the labels architecturally maps the answer starts and ends to their respective contexts: the two sequences represent two probability distributions (Figure 4.2, on the right) over the context's words. The items in the sequences indicate all the possible indices of the contexts in which the answer could start/end: the zeros indicate the "wrong" starts/ends, the ones indicate the correct start/end.

Finally, the preprocessed data is saved into Numpy files to be retrieved rapidly without going through the data preprocessing each time.

4.3 Model Architectures

After the preprocessing phase, the data is finally ready to be fed into the neural network.

Figures 4.3 illustrates the input and output structure common to each of the models. Figure 4.3a, on the left, describes the structure of the basic multi-output model used for the QA task, while Figure 4.3b, on the right, illustrates the single-output model architecture which will be used in the OWQA problem.



(a) Multi-output basic model architecture for the QA task

(b) Single output basic model architecture for the OWQA task

Figure 4.3: Basic model architecture

Each model takes two inputs: the passage tokens and the question tokens. Keras Functional API[7] allows to create models with multiple inputs and outputs. The two inputs are then separately fed into their respective Embedding Layer, which returns the corresponding Word Embeddings[8]. The embedded passages and questions are then parsed into the Intermediate Layers of the models.

[7]`https://keras.io/getting-started/functional-api-guide/`
[8]See Section 3.2.3: *Word Embeddings.*

The Intermediate Layers form the main part of the model architectures, and they are specific to each different model. For now, they will be considered as "black boxes" of neural network layers that take the embedded passages and questions as inputs and return an encoded version of them.
Finally, the encoded passages and questions are returned from the Intermediate Layers, and they are used to generate the output(s) of the model. As showed in Figure 4.3, the model architecture for the two tasks, QA and OWQA, differs in the structure of the output layers.

On one hand, the QA problem requires the model to output two values, corresponding to the start and end position of the answer in the context. Also, the value of the second output (the end position) should depend on the value predicted for the start position. In this way, the model can learn the dependencies between the two values: for example, the end position is always after the start, or the two values are never too far apart from each other.
In order to achieve this functionality, the model firstly predicts the answer start using a Keras Dense Layer as output layer. The Dense Layer is a fully connected neural layer, in which each neuron receives input from all the neurons in the previous layer, thus named "densely connected". In this case, the Dense layer is used with an output dimension equal to the fixed length of the passages, and a *Softmax* activation[9]. This means that the layer returns a probability distribution over all the words in the passage, indicating the probability of the answer start being on that specific word.
The output of the answer start Dense layer is then merged with the encoded passages and questions, which were already used as the input for predicting the answer start. The two are merged using the Keras "concatenate" layer, which concatenates the two inputs into a single tensor. Finally, this single tensor is fed into a second Dense layer, which is used to predict the answer end, and shares the same properties of the first Dense Layer.

On the other hand, in the OWQA task the model only has to predict a single value, corresponding to the index of the one-word answer in the passage. As illustrated in Figure 4.3b, the output layer consists only of a Dense layer featuring the same characteristics as the Dense output layers used in the QA task.

4.3.1 Model Parameters and Hyperparameters

Every Deep Learning model is characterised by its architecture, parameters and hyperparameters. While model parameters are internal to the model and/or estimated from the data, hyperparameters can be tuned in order to improve the model performance.
Some of these variables were common to every model and remained unchanged during the multiple training runs:

[9]See Section 3.2.2: *Activation Functions.*

Loss Function. The loss function measures the performance on the training data, acting as a feedback signal for learning the weight tensors. Referred to as the *objective function* in Section 3.2.1: *Gradient Descent*, it is the function which the training phase attempts to minimise. The loss function used for every model is *categorical_crossentropy*, as it compares the distribution of the predictions (the output layer softmax activation) with the true distribution of the labels, where the probability of the answer start/end index is set to 1, and 0 for the other indices.
Optimizer. The Optimizer is the mechanism through which the network updates itself based on the data and its loss function. The optimizer that was used throughout this project is RMSProp (Root Mean Square Propagation), variant of SGD [10]. In all the models, its performance was similar, but slightly better than Adam's, the second optimizer considered.

Other hyperparameters, instead, were changed and tuned more often during the various runs of the different models:
Learning Rate. The Learning Rate hyperparameter controls the "speed" of the training, deciding by how much the weights of the network are adjusted. It was one of the most important parameters to tune, as it had more visible effect on the training than any other parameter. The values chosen ranged from 0.0005 to 0.005.
Batch Size. This parameter indicates the number of training samples in one forward/backward pass. The smaller the batch size, the less accurate the estimate of the gradient is; however, an higher batch size needs more memory space. Depending on the resources available and the complexity of the model, the batch ranged from 1024 to 50.
Dropout Rate. When using dropout in/after layers in the neural network, a probability p (corresponding to the dropout rate) of units in the layer are temporarily ignored in calculations. Dropout helped reducing overfitting, and it was essential in almost every model. The dropout rates used range between 0.2 and 0.5.

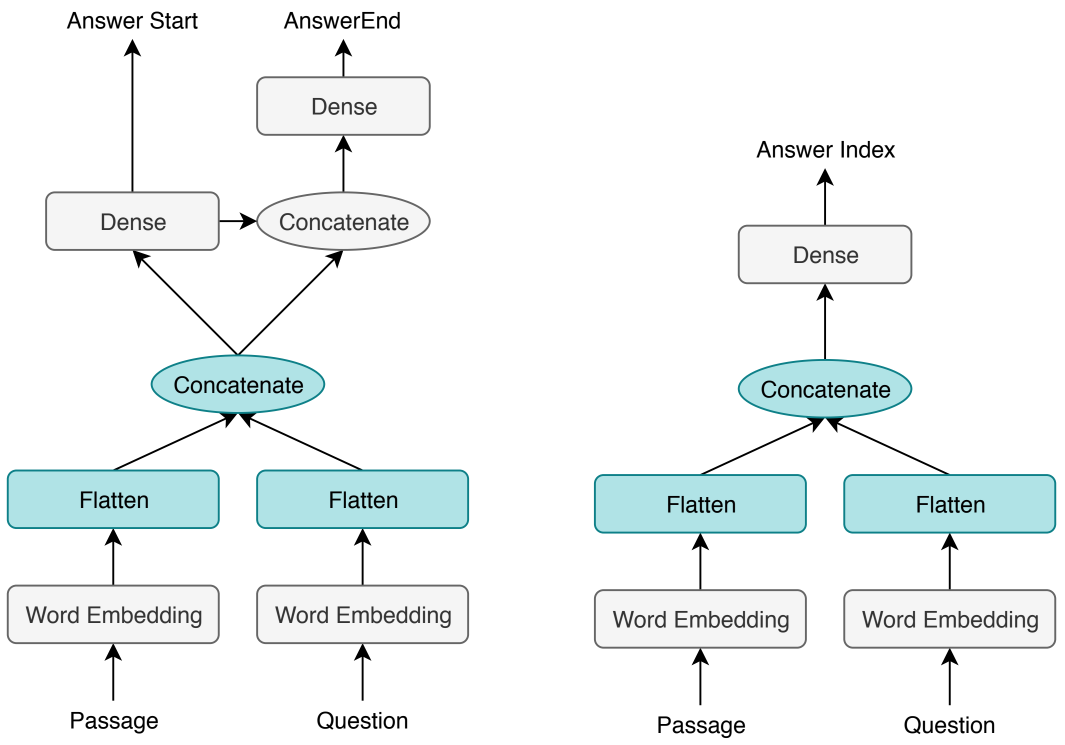
4.3.2 Model Callbacks

Keras API provides a set of functions, called callbacks, that allow to interact with the training model process automatically. Callbacks can also be used to view internal states of the model during training. In particular, two callbacks were used for the models:
TensorBoard. It allows to save the logs of the model in a directory, to then be examined with the TensorBoard visualisation tool.
ModelCheckpoint. It allows to save the model and its weights during training as a checkpoint file, after every epoch. This callback function was particularly useful in this project as the epochs of most models turned out to take a long time to run, and it allowed to save them for later re-use.

[10]See Section 3.2.1: *Gradient Descent.*

4.3.3 Model 1: Baseline

The first model represent a simplistic Deep Learning baseline for the two tasks. It focuses on the effectiveness of the representations created from the Embedding Layer alone, which are then input in the Dense output layer/layers. As Word Embeddings are at the base of every model described in this project, it was reasonable to set it as a starting point for this comparative research.



(a) Baseline model structure for the QA task *(b) Baseline model structure for the OWQA task*

Figure 4.4: Model 1 internal architecture

As illustrated in Figure 4.4, the two Embedding layers take as input the 2D tensors of the question and context sequences, of shapes (*samples, question_maxlength*), (*samples, context_maxlength*) respectively. Each Embedding layer then returns a 3D floating-point tensor of shape (*samples, sequence_length, embedding_dimension*). Such a 3D tensor would be ready to be processed by an RNN layer, however in this model, the embedded sequences are simply parsed into the Dense output layer, which takes only a 2D tensor as input. In order to reduce the dimensionality of the embedded sequences, the outputs of the Embedding layers are parsed into a Flatten Layer, which transforms the 3D tensors into 2D tensors of shape (*samples, sequence_maxlength * embedding_dimension*).

Parameters and Hyperparameters

The parameters chosen for the model are listed in Table 4.1.

	OWQA task	QA task
Learning Rate	0.001	0.001
Batch Size	1024	1024
Embedding Dimension	256	256

Table 4.1: Model 1 - Hyperparameters

Results

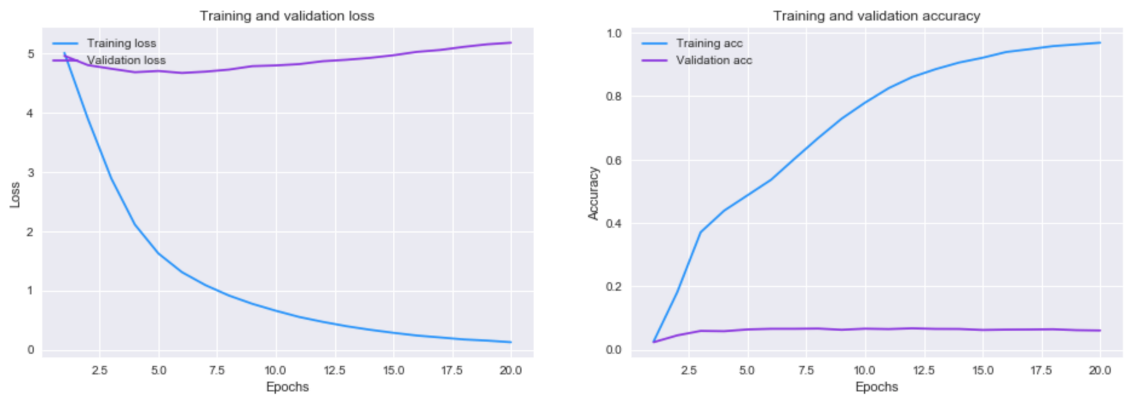
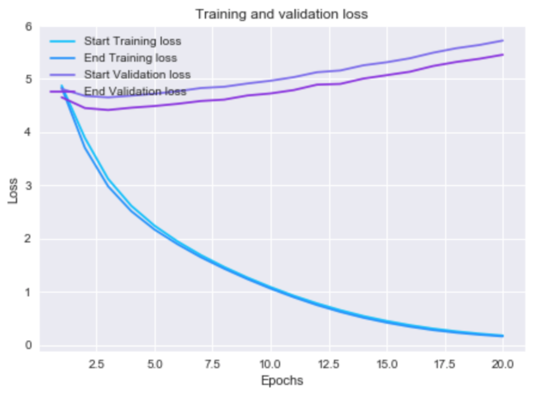


Figure 4.5: Plots of the Training and Validation Accuracy and Loss of model 1 on the OWQA task over 20 epochs.



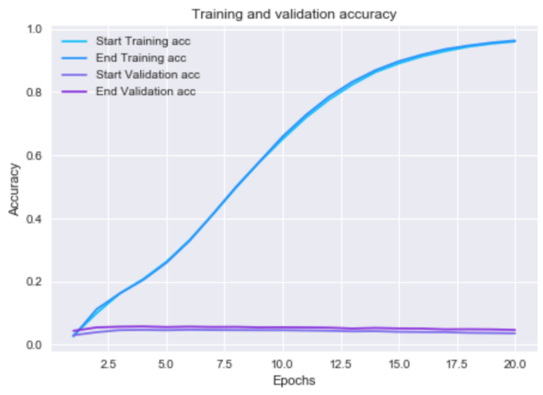


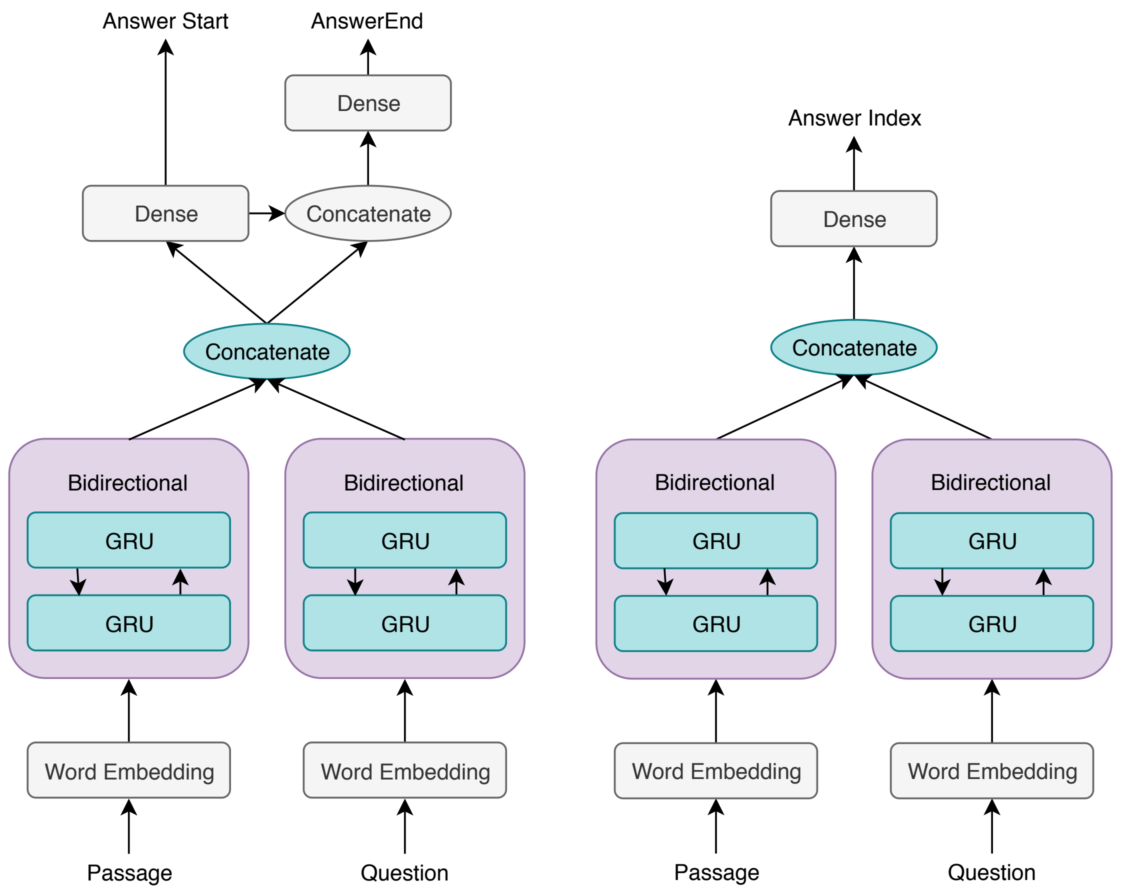
Figure 4.6: Plots of the Training and Validation Accuracy and Loss of model 1 on the QA task over 20 epochs.

	OWQA task		QA task	
	Training Set	Development Set	Training Set	Development Set
Total params	65,994,004	-	96,311,871	-
Time p.e.	5s	-	11s	-
Epochs	20	-	20	-
Running Time	1min 40s	-	3min 40s	-
EM	78.3974237	7.894736842	77.9986812	1.2950683
Top2-EM	80.6630043	12.15856095	-	-
Top3-EM	81.4548209	16.25582944	-	-
F1	-	-	79.2226117	5.6589017

Table 4.2: Training running time and performance results on the Training and Development sets of model 1 on the two tasks.

4.3.4 Model 2: RNNs

The second model has the purpose of demonstrating the performance of RNNs on the two tasks. The network structure is similar to the one just presented for model 1. On top of the Embedding layers, however, a RNN module is used, as illustrated in figure 4.7. In particular, the Gated Recurrent Unit (GRU) Layer[11] was chosen for both tasks. Although GRU cells are demonstrated to be slightly outperformed by LSTM cells in most NLP tasks such as Machine Translation (Britz et al., 2017), they are computationally cheaper to run (Chollet, 2018). In projects with resource constraints, such as this[12], GRU layers are the best option.



(a) Bi-RNN model structure for the QA task *(b) Bi-RNN model structure for the OWQA task*

Figure 4.7: Model 2 internal architecture

[11]see Section 3.2.4: *RNNs*
[12]see Section **??**: *Constraints*

From Figure 4.7, it can be observed that the GRU module does not consist of a single GRU layer, but rather a Bidirectional GRU (Bi-GRU) layer. A Bidirectional RNN traverses the input sequence in both directions, using two RNN hidden layers going in opposite directions, for then concatenating the two resulting outputs (both cell outputs and final hidden states). The output o_t corresponding to the t^{th} word is the result of the concatenation of vectors $[o_t^{(f)}, o_t^{(b)}]$, where $o_t^{(f)}$ is the output of the forward-direction RNN on word t and $o_t^{(b)}$ is the corresponding output from the reverse direction RNN(Genthial et al., 2019).

A Bidirectional RNN Layer is more effective than a single-directional one on NLP problems, as it accounts for the dependencies in sentences in both directions: a word can have a dependency on another word before or after it. Forward RNNs, instead, only consider information from words *before* the current word. Also on the tasks considered in this paper, the Bidirectional RNN was more effective.

Parameters and Hyperparameters

	OWQA task	QA task
Learning Rate	0.0005	0.001
Batch Size	512	512
Embedding Dimension	300	256
Bi-GRU Output Dimension	(128+128) = 256	(75+75) = 150
Bi-GRU Dropout Rate	0.5	0.3

Table 4.3: Model 2 - Hyperparameters

Results

	OWQA task		QA task	
	Training Set	Development Set	Training Set	Development Set
Total params	51,924,276	-	53,678,655	-
Time p.e.	29s	-	97s	-
Epochs	50	-	50	-
Running Time	24min 10s	-	1h 20min 50s	-
EM	66.845993	1.5323117	74.325988	1.6680480
Top2-EM	69.547262	2.9980013	-	-
Top3-EM	70.752036	4.1305796	-	-
F1	-	-	76.7267705	6.3948697

Table 4.4: Training running time and performance results on the Training and Development sets of model 2 on the QA and OWQA tasks.

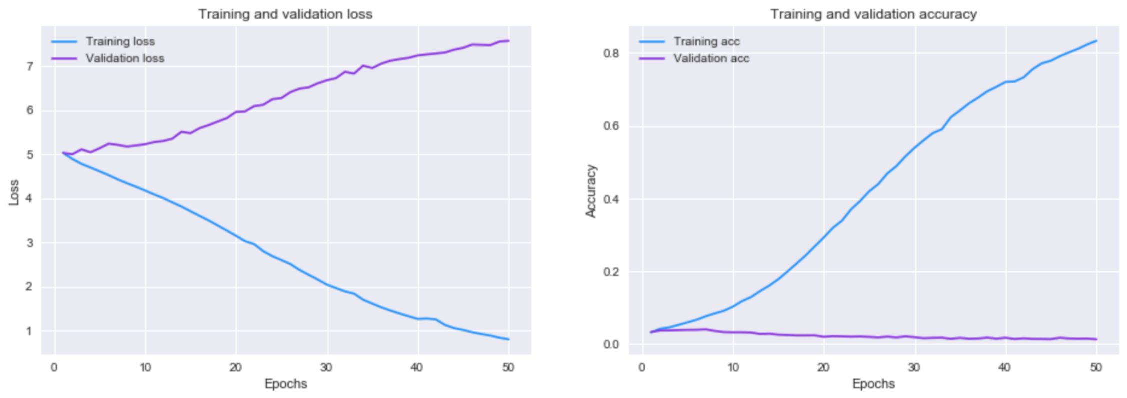


Figure 4.8: Plots of the Training and Validation Accuracy and Loss of model 2 on the OWQA task over 50 epochs.

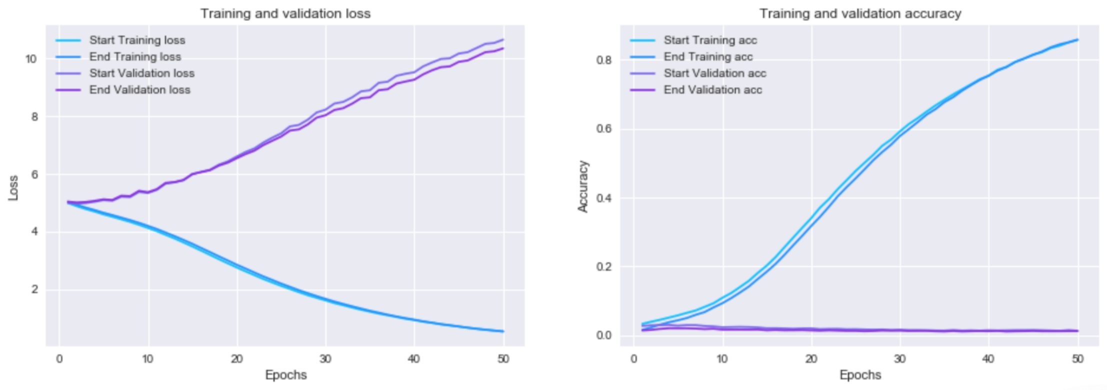
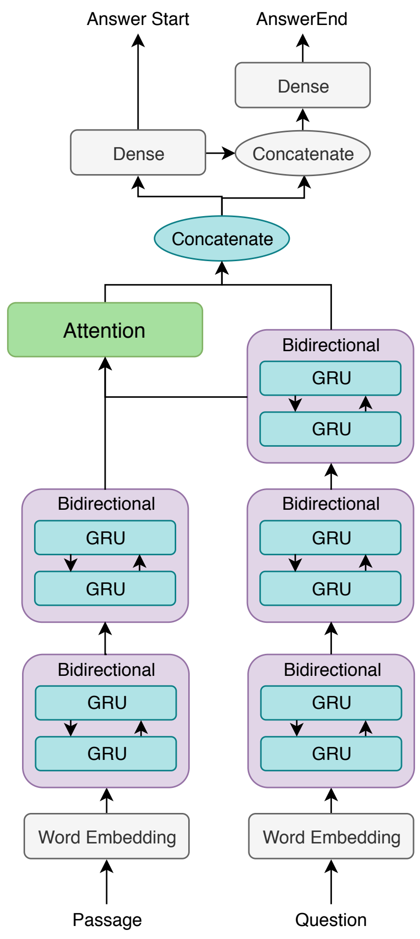


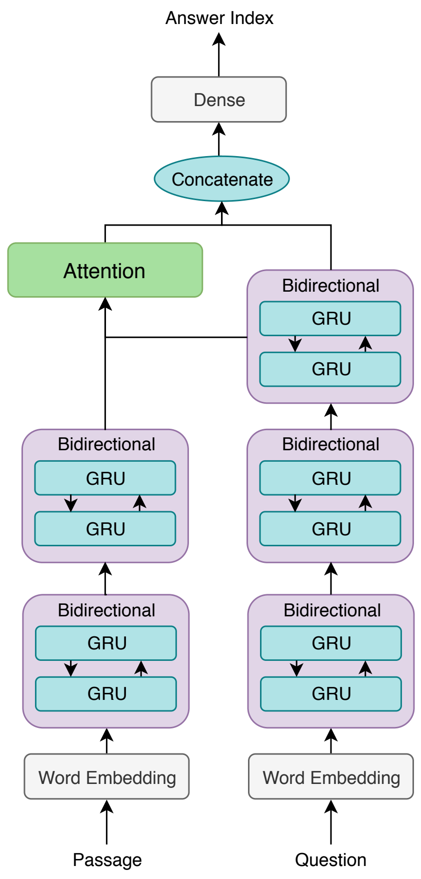
Figure 4.9: Plots of the Training and Validation Accuracy and Loss of model 2 on the QA task over 50 epochs.

4.3.5 Model 3: Attention

The third model, illustrated in Figure 4.10, introduces Attention Mechanisms[13] to the RNNs from model 2. The architecture includes a stack of two Bi-GRU layers after the passage Embedding, and a stack of three Bi-GRU after the question Embeddings. The outputs from the RNN stacks are parsed into the Attention Layer, which will return a new representation of the contexts dependent on the question. The output of the Attention Layer is then concatenated to the encoded question and parsed into the output layers to predict the answer, as in previous models.



(a) Attention model structure for the QA task



(b) Attention model structure for the OWQA task

Figure 4.10: Model 3 internal architecture

[13]see Section 3.2.5: *Attention Mechanisms*

Similarly to model 2, the Embedded questions and passages are input into the Bi-GRU. In this model, the Bi-GRUs are stacked together to increase the network capacity. To stack RNNs on top of each other, all intermediate layers need to return their full sequence of outputs (a 3D tensor), including their states, rather than only their output at the last timestep. This is done in Keras by specifying the parameter *return_sequences=True* in the intermediate layers (Chollet, 2018). As mentioned before, the embedded questions are encoded in a stack of three Bi-GRUs. The first two, as intermediate layers, have the parameter `return_sequences=True`. The third Bi-GRU instead outputs the encoded questions. The embedded contexts, on the other hand, are parsed in a stack of two Bi-GRUs, considered both intermediate layers, as their hidden states are input into the Attention layer.
The Attention Layer therefore takes as inputs the encoded questions and the contexts hidden states and outputs. This allows to compare each word and state of the passage with the question matrix and produce an attention score, used to create a new, question-aware representation of the passage. This new representation of the passage (the output of the Attention Layer) is then used with the encoded questions to predict the answers.

Disclosure

The Attention Layer implemented in this model was taken from Wentao Zhu's Recurrent Attention in Keras[14]. This particular Attention custom layer was chosen as it was implemented in Keras, and it was also originally applied to the same dataset, SQuAD. Although the repository did not contain comments or results on the dataset, the implemented Attention proved to perform correctly an Additive Attention Mechanism, and presented all the necessary feature for Attention to operate[15].

Parameters and Hyperparameters

	OWQA task	QA task
Learning Rate	0.001	0.005
Batch Size	50	1024
Embedding Dimension	256	64
Bi-GRU Hidden Units	75, 75, 75	64, 96, 128
Attention Hidden Units	75	128
Bi-GRU Dropout Rate	0.3	0.3

Table 4.5: Model 3 - Hyperparameters

[14]GitHub Repository: `https://github.com/wentaozhu/recurrent-attention-for-QA-SQUAD-based-on-keras`.
[15]See Section 3.2.5: *Attention Mechanisms*

Results

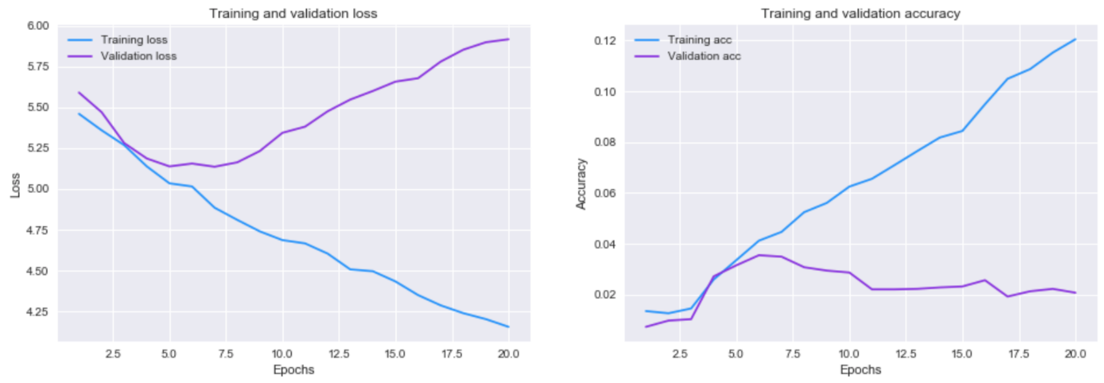


Figure 4.11: Plots of the Training and Validation Accuracy and Loss of model 3 on the OWQA task over 20 epochs.

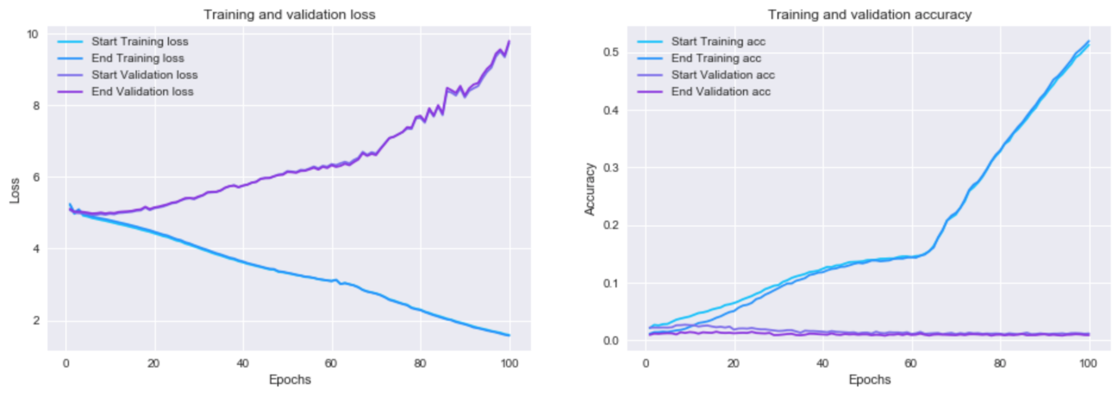


Figure 4.12: Plots of the Training and Validation Accuracy and Loss of model 3 on the QA task over 100 epochs.

	OWQA task		QA task	
	Training Set	Development Set	Training Set	Development Set
Total params	44,622,607	-	14,077,887	-
Time p.e.	11min 46s	-	3min 25s	-
Epochs	20	-	100	-
Running Time	3h 55min 20s	-	5h 41min 40s	-
EM	19.4620193	2.431712191	38.4699603	0.3937007
Top2-EM	28.0697101	3.997335109	-	-
Top3-EM	33.8700511	4.996668887	-	-
F1	-	-	46.5139066	3.6789665

Table 4.6: Training running time and performance results on the Training and Development sets of model 3 on the two tasks.

4.3.6 Model 4: R-Net

Finally, the most complex and large model out of the four. Conceived by the Natural Language Computing Group of Microsoft Research Asia (Wang et al., 2017), R-Net[16] was the state-of-the-art model on SQuAD as of August 25, 2017, reaching 72.3% EM and 80.7% F1 scores. Unfortunately, Microsoft researchers did not release the code along with their technical report. However, an R-Net implementation was reproduced and released in Keras 2.06 by YerevaNN, a non-profit computer science and mathematics research lab, in October 2017. Their R-Net version reached 57.52% EM and 67.42% F1 scores. In this project, the R-Net model was implemented by using the custom layers created by YerevaNN in their GitHub repository[17]. The model was tested on the QA task only.

Parameters

Some of the parameters used were different from the ones in the R-Net original paper, mostly due to the resource constraints of this project. The Embedding and Hidden States dimensions were drastically reduced in order for the model to compile without an OOM (Out Of Memory) error. In particular, the Embedding dimension was reduced from 300 to 64, while the hidden state dimension of every intermediate layer was modified from 75 to 32. Also, the maximum number of epochs for which the model was able to run without crushing due to a “Resources Exhausted” error was between 12 and 13. The model was therefore trained for “only” 10 epochs, taking 2 hours each.

	OWQA task	R-Net (original)
Optimizer	Adadelta	Adadelta
Learning Rate	1 (default)	1 (default)
Batch Size	50	50
Embedding Dimension	64	300
Hidden units	32	75
Dropout Rate	0.2	0.2

Table 4.7: Model 4 VS Original R-Net - Hyperparameters

[16]See Section 3.2.6: *R-Net Model*

[17]`https://github.com/YerevaNN/R-NET-in-Keras`

Results

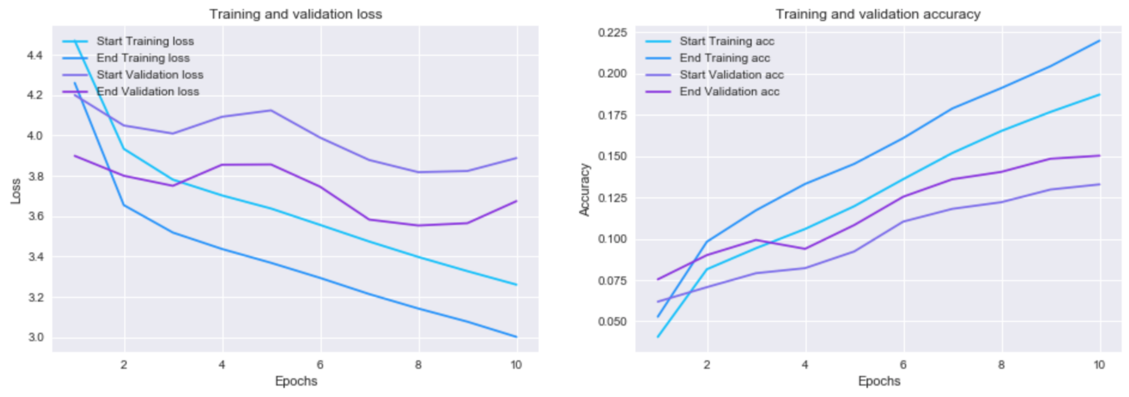


Figure 4.13: Plots of the Training and Validation Accuracy and Loss of model 4 on the QA task over 10 epochs.

	QA task	
	Training Set	Development Set
Total params:	13,410,112	-
Time p.e.	1h 48min 29s	-
Epochs	10	-
Running Time	18h 4min 50s	-
EM	15.555444	10.547036
F1	22.7805155	16.872682

Table 4.8: Training running time and performance results on the Training and Development sets of model 4 on the QA task.

Chapter 5

Evaluation

The models were tested on two separate tasks, both based on the same dataset, SQuAD. The first task (QA) included the whole SQuAD dataset. In the second task (OWQA) instead, only a subset of SQuAD samples was considered, made only of one-word answers.

Selecting only a subset of one-word answers allowed to reduce the complexity of the problem, and obtain better results from the models. While for the QA task the models were required to predict the start and end positions of the answer in the context, in the OWQA task the start and end positions of the answer coincide. With answer start and stop points being the same, the number of possible choices, corresponding to the length value of the contexts, are exactly half than in the QA task. It is therefore not surprising that the models, as showed from the results in Table 5.1, performed better on the OWQA task.

	OWQA Task			QA Task			
	Model 1	Model 2	Model 3	Model 1	Model 2	Model 3	Model 4
Parameters	65,994,004	51,924,276	44,622,607	96,311,871	53,678,655	14,077,887	130,410,112
Time p.e.	5s	29s	11min 46s	11s	97s	3min 25s	1h 48min 29s
Epochs	20	50	20	20	50	100	10
Time tot	1min 40s	24min 10s	3h 55min 20s	3min 40s	1h 20min 50s	5h 41min 40s	18h 4min 50s
EM	7.894736842	1.5323117	2.431712191	1.2950683	1.6680480	0.3937007	10.547036
Top2-EM	12.15856095	2.9980013	3.997335109	-	-	-	-
Top3-EM	16.25582944	4.1305796	4.996668887	-	-	-	-
F1	-	-	-	5.6589017	6.3948697	3.6789665	16.872682

Table 5.1: Results from all the models on the Development (test) Set.

What is surprising, however, is the performance of the baseline model (model 1) on the simpler task. This first model, composed by only Word Embedding layers and a Dense classifier on top, was particularly fast to train, due to its simplicity: its training time corresponded to 5 seconds per epoch on the OWQA task, 11 seconds on QA. This allowed the model to be trained with more parameters than the other models, given the resource constraints of the project; the resulting total number of parameters used for QA task, for example, was 96M (96,311,871). Such a large number of parameters allowed a proportionally large hypothesis space. Therefore, despite its simplicity and low-dimensionality, model 1 resulted as the best-performing model on the OWQA task, based on the performance scores EM, Top2-EM and Top3-EM.

Performance metrics aside, however, the training loss and accuracy plots of the first model showed the opposite. As it can be observed in Figure 5.1, although all the three models dramatically overfit, their level of overfitting is different. Model 1 (on the left) overfits badly since the first epoch; the validation loss is not improving, but remaining a flat line since the start. In model 2 and 3 plots, instead, the validation loss follows the training loss, and the two lines do not separate each other as in model 1. This two models, and in particular model 3, implementing attention, show a substantial margin of improvement; as a matter of fact, the number of parameters they were trained with was lower than model 1. A more powerful machine, able to run these expensive models with a larger number of parameters, would definitely achieve proportionally higher results. However, with the resources constraints of this project, this was not possible.

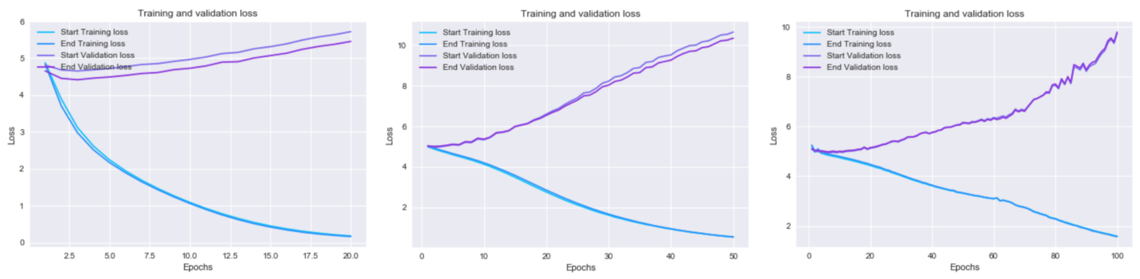


Figure 5.1: Plots of the Train Losses in models 1 (left), 2 (centre) and 3 (right).

On the OWQA task, model 3 was evaluated by also printing the answer predictions versus the ground truth answers, along with the probability distribution output from the softmax activation of the output layer. Two of these plots are visible in Figure **??**. These two particular examples were questions that the model guessed right, as it can be observed from the Figure: The predictions “four” and “2015” are the same as the ground truth answers. It was possible to notice that most of the correct predictions were numbers, such as the ones displayed below. Apparently, the model found easier to answer to question including “When”, “How many?”, and indicating a numerical value. This is not surprising, as this type of questions require less interpretation of the text, and are therefore easier to guess for the model.

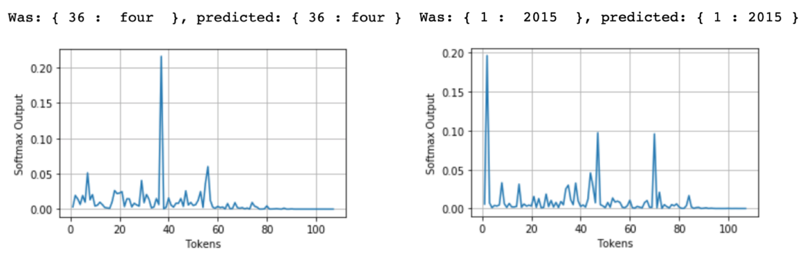


Figure 5.2: Plots of the softmax activation over the context words (tokens) of two answers guessed right by model 3.

The last model instead, R-Net (model 4), despite the fact that it was trained for only 10 epochs (which took nearly 19 hours) due to the resource constraints, it performed better than any other model on the QA task. In only 10 epochs, it reached a 10.5% EM and a 16.9% F1 score, confirming the effectiveness of complex models featuring Attention Mechanisms on SQuAD. SQuAD has indeed been demonstrated to be a fairly complex dataset, in which many questions require interpretation, word paraphrasing, and sometimes long answers. A small model such as model 3, even with Attention Mechanisms, it appeared not able to perform well on such an advanced dataset.

Chapter 6

Conclusion

This paper has completed a study on Question Answering Systems and the approaches used to solve the challenges presented by Machine Reading Comprehension in order to build state-of-the-art QA models. The research included also insights about the mechanisms behind these approaches, from basic NLP techniques such as Word Embeddings, to more recent and exclusive tools such as Attention.

In particular, the research depicted Attention Mechanisms as the greatest improvement in NLP after Recurrent Neural Networks, referring to it as "*the new vanilla*" (Genthial et al., 2019) in tasks such as Machine Translation and Machine Comprehension. As a matter of fact, all modern MC state-of-the-art models use Attention variations in their architectures. A study into Attention Mechanisms proved their theoretical effectiveness on Machine Comprehension problems, justifying the hype highlighted by modern research papers during the literature review process.

In order to directly examine and investigate further the results obtained from the research, four models were implemented in Keras. The models were selected to be increasingly complex, ranging from the simplest architecture with only Word Embeddings to a state-of-the-art model, R-Net.

The results obtained from the models, discussed and compared in Section 5: *Evaluation*, proved the research results to a certain extent. The last model, R-Net, with its complex attention-based structure and its custom layers, proved to overcome every other model in any aspect. With datasets such as SQuAD, only powerful models with hundreds of millions of parameters such as BERT (Devlin et al., 2018) were demonstrated to have human-level performance. But, however "good" the results achieved by these models are in Machine Comprehension, Artificial Intelligence is still far from fully understanding and interpreting human language.

As reported in this paper, Neural Networks are able to map semantic relationships of words in a geometrical space using Word Embeddings and POS tagging; take into consideration the order of the words in a sequence using RNNs; even return a probability score of the relations between separate sequences with Attention Mechanisms. The way these features are obtained, however, has no relations to the way our minds think and process the same information. And even those models mentioned earlier, yielding impressive results on NLP tasks, do not understand natural language in the same way as humans do (yet). Maybe they never will. Maybe it won't even be necessary. "Is there some ideal word-embedding space that would perfectly map human language and could be used for any NLP task?" asks Chollet in his book (Chollet, 2018). Possibly. But would it be possible for every language? And would it even solve NLP problems? Probably not, or at least not alone. An intelligent AI able to understand natural languages still seems far away. The progress obtained in the field with Deep Learning in about a decade, however, remains absolutely impressive, and proves that there is hope to see even greater advancements in the next few years.

Bibliography

AliCloud (2017). *QA Systems and Deep Learning Technologies - Part 1.* URL: https://www.alibabacloud.com/blog/qa-systems-and-deep-learning-technologies-e28093-part-1_71899.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). "Neural Machine Translation by Jointly Learning to Align and Translate". In: pp. 1–15. URL: http://arxiv.org/abs/1409.0473.

Berant, Jonathan, Vivek Srikumar, Pei-Chun Chen, Abby Vander Linden, Brittany Harding, Brad Huang, Peter Clark, and Christopher D Manning (2014). "Modeling Biological Processes for Reading Comprehension". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).* Doha, Qatar: Association for Computational Linguistics, pp. 1499–1510.

Britz, Denny, Anna Goldie, Minh-Thang Luong, and Quoc Le (2017). "Massive Exploration of Neural Machine Translation Architectures". In: URL: https://github.com/moses-.

Chen, Danqi, Adam Fisch, Jason Weston, and Antoine Bordes (2017). "Reading Wikipedia to Answer Open-Domain Questions". In: DOI: 10.18653/v1/P17-1171. URL: http://trec.nist.gov/data/qamain.html%20http://arxiv.org/abs/1704.00051.

Chollet, François (2018). *Deep Learning with Python.* Shelter Island, NY: Co., Manning Publications. ISBN: 9781617294433.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova (2018). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: ISSN: 0140-525X. DOI: arXiv:1811.03600v2.

Elman, Jeffrey L (1990). "Finding Structure in Time". In: *Cognitive Science* 14.2, pp. 179–211. DOI: 10.1207/s15516709cog1402{_}1. URL: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1402_1.

Genthial, Guillaume, Lucas Liu, Barak Oshri, and Kushal Ranjan (2019). "Neural Machine Translation, Seq2seq and Attention". In: *CS224n: Natural Language Processing with Deep Learning - Lecture Notes* Part VI. URL: http://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes06-NMT_seq2seq_attention.pdf.

Goldberg, Yoav (2016). "A primer on neural network models for natural language processing". In: *Journal of Artificial Intelligence Research* 57, pp. 345–420. ISSN: 10769757. DOI: 10.1613/jair.4992.

Green, Bert F, Alice K Wolf, Carol Chomsky, and Kenneth Laughery (1986). "An Automatic Question Answerer". In: *the Western Joint Computer Conference 19*, pp. 219–224.

Hermann, Karl Moritz, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom (2015). "Teaching Machines to Read and Comprehend". In: *Advances in Neural Information Processing Systems 28*. Ed. by C Cortes, N D Lawrence, D D Lee, M Sugiyama, and R Garnett. Curran Associates, Inc., pp. 1693–1701. URL: http://papers.nips.cc/paper/5945-teaching-machines-to-read-and-comprehend.pdf.

Hill, Felix, Antoine Bordes, Sumit Chopra, and Jason Weston (2016). "The Goldilocks Principle: Reading Children's Books with Explicit Memory Representations". In: *International Conference on Learning Representations*, pp. 1–13. URL: http://arxiv.org/abs/1511.02301.

Jousse, Florent, Isabelle Tellier, Marc Tommasi, and Patrick Marty (2005). *Learning to Extract Answers in Question Answering: Experimental Studies*. Tech. rep. URL: www.grappa.univ-lille3.fr.

Jurafsky, Daniel and James H Martin (2018). *Speech and Language Processing (2018)*. 3rd ed. Vol. 1. URL: https://dl.acm.org/citation.cfm?id=1214993.

Markoff, John (2011). *Computer Wins on 'Jeopardy!': Trivial, It's Not*. Yorktown Heights, New York.

Mirakyan, Martin, Karen Hambardzumyan, and Hrant Khachatrian (2017). *Challenges of reproducing R-NET neural network using Keras*. URL: http://yerevann.github.io/2017/08/25/challenges-of-reproducing-r-net-neural-network-using-keras/#implementation-details.

Moravec, Hans (1988). *Mind children.* Cambridge (MA US): Harvard University Press, p. 15. ISBN: 9780674576186.

Nguyen, Tri, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng (2016). "MS MARCO: A Human Generated MAchine Reading COmprehension Dataset". In: *CoRR* abs/1611.0. URL: `http://arxiv.org/abs/1611.09268`.

Pundge, Ajitkumar M. (2016). "Information Retrieval and Question Answering Systems: A Review". In: *Information Retrieval and Question Answering Systems: A Review*, pp. 14–16.

Rajpurkar, Pranav, Jian Zhang, Konstantin Lopyrev, and Percy Liang (2016). "SQuAD: 100,000+ Questions for Machine Comprehension of Text". In: ISSN: 9781941643327. DOI: `10.18653/v1/D16-1264`. URL: `http://arxiv.org/abs/1606.05250`.

Richardson, Matthew, Christopher J C Burges, and Erin Renshaw (2013). "MCTest: A Challenge Dataset for the Open-Domain Machine Comprehension of Text". In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, Washington, USA: Association for Computational Linguistics, pp. 193–203. URL: `https://www.aclweb.org/anthology/D13-1020`.

Ruder, Sebastian (2016). "An overview of gradient descent optimization algorithms". In: pp. 1–14. URL: `http://arxiv.org/abs/1609.04747`.

Seo, Minjoon, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi (2016). "Bidirectional Attention Flow for Machine Comprehension". In: ISSN: 09226389. DOI: `10.1002/2014GB005021`.

Socher, Richard and Abigail See (2019). *Lecture 8: Machine Translation, Sequence-to-sequence and Attention [Lecture Slides].*

Turing, Alan M (1950). "I. - Computing Machinery and Intelligence". In: *Mind* LIX.236, pp. 433–460. ISSN: 0026-4423. DOI: `10.1093/mind/LIX.236.433`. URL: `https://doi.org/10.1093/mind/LIX.236.433`.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin (2017). "Attention Is All You Need". In: *arXiv e-prints*, arXiv:1706.03762.

Voorhees, Ellen M and Dawn M Tice (1999). *The TREC-8 Question Answering Track*. Tech. rep. URL: https://pdfs.semanticscholar.org/74e0/3acd5532fbad4c770e9293d2a788b11364f7.pdf.

Wang, Wenhui, Nan Yang, Furu Wei, Baobao Chang, and Ming Zhou (2017). *R-NET : Machine Reading Comprehension with Self-Matching Networks*. Tech. rep., pp. 1–11. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/05/r-net.pdf.

Woods, W. A. (1978). "Semantics and Quantification in Natural Language Question Answering". In: *Advances in Computers*. DOI: 10.1016/S0065-2458(08)60390-3.

Yampolskiy, Roman V (2013). "Turing test as a defining feature of AI-completeness". In: *Studies in Computational Intelligence* 427, pp. 3–17. ISSN: 1860949X. DOI: 10.1007/978-3-642-29694-9{_}1.

Yang, Yi, Wen-tau Yih, and Christopher Meek (2015). "WikiQA: A Challenge Dataset for Open-Domain Question Answering". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, pp. 2013–2018. DOI: 10.18653/v1/D15-1237. URL: https://www.aclweb.org/anthology/D15-1237.

Yao, Xuchen, Benjamin Van Durme, Chris Callison-burch, and Peter Clark (2013). "Answer extraction as sequence tagging with tree edit distance". In: *In North American Chapter of the Association for Computational Linguistics (NAACL*.

Young, Tom, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria (2018). "Recent trends in deep learning based natural language processing [Review Article]". In: *IEEE Computational Intelligence Magazine* 13.3, pp. 55–75. ISSN: 15566048. DOI: 10.1109/MCI.2018.2840738.

Zhang, Dell and Wee Sun Lee (2003). "Question Classification Using Support Vector Machines". In: *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval*. SIGIR '03. New York, NY, USA: ACM, pp. 26–32. ISBN: 1-58113-646-3. DOI: 10.1145/860435.860443. URL: http://doi.acm.org/10.1145/860435.860443.