University of São Paulo

Alvaro Henrique Chaim Correia

Jorge Luiz Moreira Silva

Development of a Fully Attention-Based Question-Answering Model

São Paulo, SP, Brazil

2017

University of São Paulo

Alvaro Henrique Chaim Correia

Jorge Luiz Moreira Silva

Development of a Fully Attention-Based Question-Answering Model

Final year project presented to the Department of Mechatronics of the Polytechnic School of the University of São Paulo to obtain the Degree in Engineering

Advisor: Prof. Dr. Fábio Gagliardi Cozman

São Paulo, SP, Brazil

2017

Catalogação-na-publicação

Correia, Alvaro Henrique Chaim
Development of a Fully Attention-Based Question-Answering Model / A. H. C. Correia, J. L. M. Silva -- São Paulo, 2017.
92 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos.

1.SISTEMAS DE QUESTÕES E RESPOSTAS 2.APRENDIZADO COMPUTACIONAL 3.REDES NEURAIS I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos II.t. III.Silva, Jorge Luiz Moreira

Acknowledgments

First and foremost we would like to thank our advisor, Prof. Fabio Cozman, for his orientation and encouragement throughout this project. Also, we are immensely grateful to Prof. Thiago Martins, who gave us access to the needed computational resources. Without the equipment provided by his research group the experiments reported here would not have been viable.

We also want to extend our gratitude to all the professors who contributed to our education. Without their guidance and dedication, the intellectual development culminating in this final project would not have been possible. The same way, we want to register our appreciation for the University of São Paulo as an institution and thank all the professionals that work to keep it an international reference in the development of science.

Finally, we thank our families for the unconditional support not only during this project but also throughout our academic lives. We both have been full-time engineering students for the last seven years, which would have been inconceivable without the investment they have made in our education. For that we will be forever indebted and, as a token of our appreciation, we offer this project to our families.

Resumo

Este trabalho tem como propósito o desenvolvimento de um modelo de perguntas e respostas, capaz de responder a questões sobre um parágrafo, supondo que a resposta possa ser recuperada num trecho contínuo do texto. O estado da arte neste tópico é dominado por redes neurais recorrentes, pois estas são capazes de representar a relação entre as palavras no contexto de uma pergunta. Esses modelos, porém, são sequenciais, o que os torna lentos e difíceis de treinar em comparação a redes neurais tradicionais (*feedforward*) que são mais simples e paralelizáveis. Estas, no entanto, por terem um tamanho de entrada pré-definido, não conseguem incorporar a relação entre palavras separadas por distâncias arbitrárias, o que reduz sua capacidade de interpretar a estrutura semântica e gramatical do texto. Para solucionar este problema, foram desenvolvidas novas arquiteturas de redes neurais que incorporam a interação entre as palavras através de um mecanismo de atenção que inclui a posição de cada palavra como entrada. O modelo aqui desenvolvido, chamado de FABIR (extrator de informações exclusivamente baseado em atenção), será baseado exclusivamente nesses novos mecanismos, sem o uso de redes recorrentes, com o propósito de estudar sua aplicação em perguntas e respostas. FABIR atingiu resultados semelhantes aos melhores modelos em um banco de dados público, possuindo um menor número de parâmetros e maior velocidade tanto no processo de treino quanto de inferência.

Palavras-chave: Sistemas de Questões e Respostas. Aprendizado Computacional. Redes Neurais.

Abstract

This project aims at the development of a machine learning model capable of answering questions about a given passage, assuming that the answer is contained in a continuous snippet of the text. The state-of-the-art in question-answering is currently dominated by recurrent neural networks because they are capable of representing the relationship between words in the context of a question. These models, however, are sequential, which makes them slow and difficult to train as opposed to standard neural networks (*feedforward*), which are a simpler alternative that increases the amount of computation that can be parallelized. Nevertheless, feedforward neural networks cannot model the relationship between words separated by arbitrary distances because their input size is a pre-defined parameter. That hinders the effective representation of grammatical and semantic structures of text, which are essential for natural language processing tasks. To address that limitation, new neural network architectures have been developed that incorporate the interdependence between words through an attention mechanism that includes the position of each word as an input. We propose a new question-answering model that is entirely based on these new mechanisms, which we call Fully Attention-Based Information Retriever (FABIR). We show that FABIR achieves competitive results in an open source question-answering dataset while having fewer parameters and being faster at both learning and inference than rival methods.

Keywords: Question-Answering. Machine Learning. Neural Networks.

List of Figures

List of Tables

List of abbreviations and acronyms

AI	Artificial Intelligence
AoA	Attention over Attention Neural Networks for Reading Comprehension
BiDAF	Bidirectional Attention Flow for Machine Comprehension
CNN	Convolutional Neural Network
DCN	Dynamic Coattention Networks for Question Answering
EM	Exact Match Score
FABIR	Fully Attention Based Information Retrieval
F1	Harmonic mean of Precision and Recall
GloVe	Global Vectors for Word Representation
GRU	Gated Recurrent Unit
HLR	High Level Requirement
LSTM	Long short-term Memory
MAP	Maximum A Posteriori
ML	Maximum Likelihood
MSE	Mean Squared Error
NLP	Natural Language Processing
QA	Question-Answering
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SQuAD	Stanford Question-Answering Dataset

List of symbols

θ	Learnable parameters of the model
W, b	Matrices of learnable weights
$\mathcal{P}, \Omega_P$	Passage in textual and matrix form, respectively
$\mathcal{Q}, \Omega_Q$	Question in textual and matrix form, respectively
P_{len}	Number of rows (words) in passage matrix Ω_P
Q_{len}	Number of rows (words) in question matrix Ω_Q
ω_w, ω_c	Embedding vectors correspondent to word and character-level embeddings
Ω	Matrix composed of the embedding vectors of a sentence
E_Q, E_P	Position encoder matrix for the passage Ω_P and question Ω_Q, respectively
$\mathcal{A}$	Answer in textual form
y_1, y_2	First and last indices of the words that identify the answer in $\mathcal{P}$
$\hat{y}_1, \hat{y}_2$	First and last indices of the words that were inferred by the model in $\mathcal{P}$
$\pi^{\hat{y}_1}, \pi^{\hat{y}_2}$	A vector in the P_{len}-probability simplex, such that $\pi_k^{\hat{y}_1}$ is the probability of $\hat{y}_1$ being the index k, $k \in \{0, 1, ...P_{len} - 1\}$
J	Scalar that represents the cost function when evaluated in a set of points of the data set
x	Generic vector or datapoint used as input in a neural network
X	Generic matrix used as input in a neural network. x_i represents its i^{th} row or the i^{th} data point in the inputs
ϵ	Scalar that defines the learning rate used to compute the step size in gradient descent algorithms
d_{input}	The embedding size of the vectorial representation of each word of the input, i.e., the embedding size of the concatenation of ω_w and ω_c
d_{model}	The embedding size of the vectorial representation of each word in the processing layers

d_{hidden}	The size of a hidden layer in a feed-forward neural network
n_{heads}	Number of heads in multi-head attention sublayer
d_{head}	Output dimension of each head in multi-head attention sublayer
U, K, V	Query, Key and Value matrices used in attention function
H	Four dimensional tensor associated to convolutional kernels. Typically, $H \in \mathbb{R}^{height \times width \times n_{channels} \times n_{filters}}$
σ	Sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$
$tanh$	Hyperbolic tangent function: $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
$ReLU$	Rectified Linear Unit function: $ReLU(x) = max(0, x)$
∇_θ	Gradient with respect to parameters θ
μ	Arithmetic mean
var	Variance

Contents

1 Introduction

1.1 Objectives

This final year project aims at the development of a machine learning model capable of answering questions in natural language. In that context, we set three main objectives as follows:

- Study the application of neural networks and the recent developments in attention mechanisms (2) to question-answering.
- Derive a system capable of answering a question expressed in natural language by selecting a snippet from a related piece of text.
- Validate the system on a public question-answering dataset in English (SQuAD (3)) and compare its performance against state-of-the-art models.

1.2 Motivations

We identify two primary motivations for our work. We divide them according to the practical and theoretical interests associated with our research as follows:

First, a question-answering model is a valuable tool for information retrieval that can change the way we access and use the data made available to us. The advent of the Internet allowed people to publish and access information in real-time, generating large amounts of data that are continually being stored by companies and personal users in countless databases around the world. These databases indeed hold valuable pieces of information, but they might be hidden among terabytes of useless data and hence are virtually inaccessible. Therefore, identifying relevant data given a specific information need is one of the most desirable functions for end users (4). However, despite the development of efficient search engines such as *Google*, these are still in gross limited to keyword matching, which restricts the precision and type of queries to which they can respond. A question-answering model is the most user-friendly extension to these systems because it would allow for queries in natural language, which are both more expressive and easier to formulate (5).

Second, question-answering is a vibrant research area in natural language understanding (6) and represents an important step in the development of Artificial Intelligence (AI). Understanding natural language is not only a milestone in the conception of a machine capable of reasoning, but it is also paramount to render that technology useful to us for two reasons: (i) **knowledge acquisition** given that most information available today is expressed in human language and (ii) **human–computer interaction**, as language is our main mean of communication (7, p. 860). We believe that the development of neural networks and its applications to natural language processing, including the work put forward in this project, are an significant contribution to the field of AI.

1.3 Scope

Within the broader scope of Machine Learning, our work touches two research areas in particular: **Deep Learning** and **Natural Language Processing** (NLP).

Deep learning is a class of machine learning models that comprise neural networks with multiple layers (8). It has dominated state-of-the-art research in NLP and hence, our work is entirely neural network-based in tune with the most recent developments in the field. Given that neural networks are a rich and complex research area in themselves, we feel the need for a brief introduction to allow for a deeper understanding of the inner-workings of our model. We present the main characteristics of different architectures of neural networks in chapter 3.

Natural Language Processing (NLP) is the field of science that has been developed to allow computers to perform useful tasks involving human language (9). These include tasks such as text translation, automatic captioning, spam detection, information extraction and Question-Answering (QA). The latter is the topic of this project and will be detailed further on.

1.3.1 Question-Answering (QA)

Question-answering refers to an information retrieval task in which the information need is expressed as a set of questions or statements in natural language (4). The answer could be, for instance, merely a word such as "yes" or "no", an option in a multiple choice test or even a long sequence of sentences. QA tasks can be split into **closed** and **open domain** systems. The former refers to questions related to a specific knowledge field such as football, religion and weather forecasting, whereas the latter might include questions across several areas of knowledge.

Closed domain QA systems are simpler and tend to be more accurate because prior knowledge about the target domain can be integrated into the model's design. Indeed if the domain is restricted enough, one could manually engineer a set of rules that map regular expressions to a predefined set of answers. An example of that approach is the LUNAR NLP project in which the questions were exclusively about geological compositions of the rocks found in the moon (10). That system could answer satisfactorily 78% of the questions utilizing an exhaustive set of pattern analysis over the domain-specific vocabulary, translating natural languages to entries to a database. It is important to highlight that this was achieved in 1973 and therefore with limited computational power in comparison to today's QA systems.

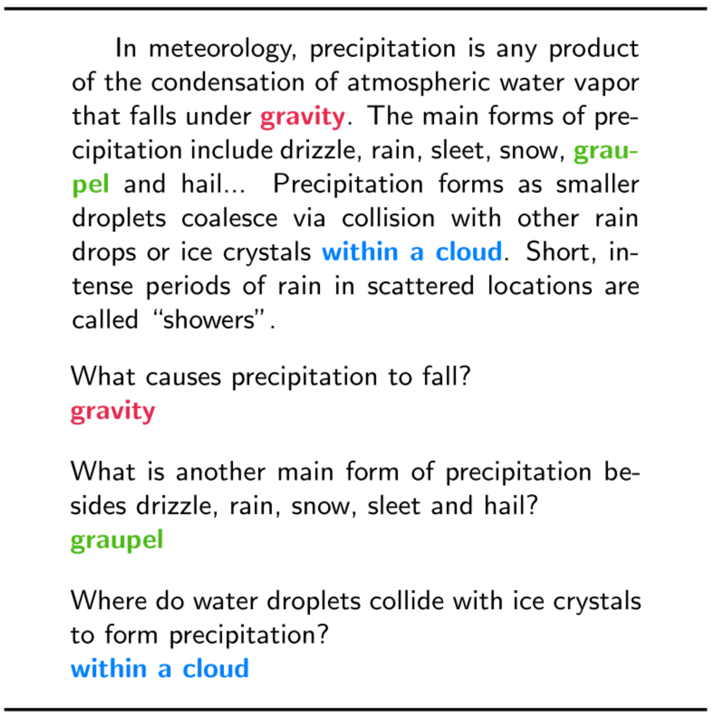
In open-domain QA systems, the scope of the questions encompasses multiple fields of knowledge, and hence manually designed rules are not feasible for any interesting information retrieval task. Indeed, to answer questions about different domains the ideal model needs some level of understanding of natural languages. Neural networks are to this day the closest we have come to such an "intelligent" model because they are particularly good at handling raw features which are not individually interpretable, such as words (11). Our work tackles the challenging task of open domain QA in the constraints imposed by Stanford's SQuAD dataset introduced in the following section.

1.3.2 SQuAD Dataset

Neural networks have been successful in tackling a variety of challenging tasks in NLP such as machine translation (12) or speech recognition. However, deep learning models usually require large amounts of labeled examples and a data set that would allow for the training of a QA system over multiple knowledge fields had been lacking. Motivated by the success of public data sets like ImageNet (13) in promoting the development of machine learning models, Rajpurkar et al. have published the Stanford Question-Answering Dataset (*SQuAD*), a large QA dataset based on Wikipedia articles (3).

The SQuAD Dataset consists of 107,785 question-answer pairs about 536 articles from *Wikipedia*, which altogether results in 23,215 paragraphs (3). To get high-quality articles about a wide range of topics, they were randomly selected from the top 10,000 articles according to the Project Nayuki's Wikipedia's internal PageRanks. As these articles cover different knowledge fields, such as history, natural science, and sports, this QA task can be considered open domain.

The question-answer pairs follow a well-defined structure. Every question $\mathcal{Q}$ is associated with a passage $\mathcal{P} = \{p_1, p_2, ..., p_n\}$, where p_i represents the i^{th} character of the respective passage and n its size defined by the number of characters. The answer $\mathcal{A}$ must be a contiguous snippet from the respective passage, i.e., it must be $\mathcal{A} = \{p_k, p_{k+1}, ..., p_l\}$, where $k \geq 1$ and $k \leq l \leq n$. On the one hand, this gives a large number of possible answers for each question, which increases the complexity of the problem against multiple choice questions. On the other hand, it simplifies the problem by reducing the interpretation requirements of the paragraph. Figure 1 illustrates the SQuAD dataset with one passage and three questions about the topic meteorology.


In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under **gravity**. The main forms of precipitation include drizzle, rain, sleet, snow, **graupel** and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals **within a cloud**. Short, intense periods of rain in scattered locations are called "showers".

What causes precipitation to fall?
gravity

What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?
graupel

Where do water droplets collide with ice crystals to form precipitation?
within a cloud

Figure 1 – Example of a paragraph extracted from the SQuAD dataset with three questions and their respective answers in colors. Image Source: (3)

1.4 Methods

As mentioned before, we follow the recent trends in the machine learning community and devise a deep learning model for QA. However, differently from previous works we do not employ Recurrent Neural Networks (RNNs), which are the base of all published models aimed at the SQuAD dataset (14, 15, 16, 17, 1). Instead, inspired by a recent paper on machine translation (2), we propose a deep learning QA model that is entirely based on attention mechanisms. To the best of our knowledge, this is a wholly new approach to QA and hence constitutes the major contribution of our work. We introduce herein our Fully Attention Based Information Retriever model, which we have named FABIR.

The development of a machine learning model of the size and complexity of FABIR requires a series of design choices involving multiple hyperparameters that define the model's architecture and training algorithm. Finding the optimal set of these parameters is not viable due to the large search space and computational cost of running the system. Therefore, we draw insights from the related work and follow some heuristics common to the field to guide us in the design of the best performing model. To compare the performance of each new architecture and also evaluate them against the related work, we apply a couple of standard metrics in NLP: EM and F1 scores, which are defined in the appendix. Finally, FABIR has been fully implemented with Tensorflow's python API (18), which facilitates the training and validation of our model.

1.5 Outline

The rest of this document is organized as follows. The following chapter presents our model from a systems engineering perspective, including functional and requirements analyses. Subsequently, we introduce the reader to the field of Machine Learning, so that we can explore more advanced topics in exposing the development of our model further on. We then cover the related work and compare the state-of-the-art machine learning models that have addressed the SQuAD dataset. Chapter 5 is entirely dedicated to theory and architecture of our fully attention-based model. Finally, we present our experimental results in Chapter 6 and later draw some conclusions and point out interesting directions for future work in Chapter 7.

2 System Analysis

In this chapter we analyze the project from a system engineering perspective. We start with stating the mission and objectives, then we perform a functional analysis and divide the model in its main building blocks. Finally, we derive the system requirements that underpin the development of this project.

2.1 Mission Definition

The mission of our system is simple and can be stated as follows:

Mission Statement *Given a passage $\mathcal{P}$ and a query $\mathcal{Q}$ both written in English, to produce an answer $\mathcal{A}$ by selecting a continuous snippet from $\mathcal{P}$.*

2.2 High Level Requirements

The mission can be divided into High Level Requirements (HLRs) that should be satisfied to produce an acceptable system.

HLR 2.1 (Nature of the System) *The system should be a parametric supervised machine learning model capable of learning to perform the task defined in the **Mission Statement** in a completely data-driven way.*

Rationale: A machine learning model requires little human intervention and is capable of generalizing to unseen data, as opposed to systems with hard-coded rules, which are labor intensive and have a limited scope.

HLR 2.2 (Data source) *The system should be trained on the examples provided in the public dataset SQuAD (3).*

Rationale: A supervised machine learning model needs a considerable amount of data to "learn" to predict the target variable given the inputs.

HLR 2.3 (Input Format) *The system should process passages $\mathcal{P}$ and queries $\mathcal{Q}$ in json format.*

Rationale: json is the most popular format for exchanging information in the web and is also the format in which the SQuAD dataset (3) is provided.

HLR 2.4 (Input Length) *The system should process passages $\mathcal{P}$ and queries $\mathcal{Q}$ of arbitrary lengths.*

Rationale: A question-answering system is only useful if it can be applied to real world documents that naturally vary in length.

HLR 2.5 (Output) *The system should output indices $\hat{y}_1$ and $\hat{y}_2$ that define the position in $\mathcal{P}$ of the first and last word of the answer $\mathcal{A}$, respectively.*

Rationale: That is the format in which answers are given in the SQuAD dataset (3).

HLR 2.6 (Loss) *The system should minimize the negative log likelihood over the examples on the training set of the SQuAD dataset (3).*

Rationale: That is equivalent to Maximum Likelihood estimation (ML) in a frequentist approach or the Maximum a Posteriori Probability (MAP) in a Bayesian approach (8, p. 131-139).

HLR 2.7 (Single step) *The system should process the entire $\mathcal{P}$ and $\mathcal{Q}$ in a single pass to output $\hat{y}_1$ and $\hat{y}_2$.*

Rationale: That renders the system faster and parallelizable for computational efficiency and scalability.

Observation: That means that Recurrent Neural Networks (RNNs) should not be used.

HLR 2.8 (Validation) *The system should achieve an F1 score of at least 70% on the development set of the SQuAD dataset (3).*

Rationale: Such an F1 score would place the system among other state-of-the-art machine learning models (19).

2.3 Functional Analysis

Functional analysis consists in defining the key building blocks of the model so that they can be maintained and updated independently. It also facilitates the understanding of the model, as it exposes the main functionalities of the system in a clear way. We divide our model in eleven functions, which are combined in accordance to the modes of operation defined on the subsequent section.

Function 2.1 (Receive input) *This function is the interface that receives a question and a passage in textual form (.json).*

Function 2.2 (Tokenize) *This function splits the text into words or characters (tokens).*

Function 2.3 (Embed words) *This function converts each word in its corresponding vectorial representation.*

Function 2.4 (Embed characters) *This function converts the set of characters of a word in a vectorial representation.*

Observation: Function 2.4 differs from Function 2.3 in that it produces a vector that is a function of the characters put together, instead of the word as a whole.

Function 2.5 (Run neural network) *This function is given by a neural network, which uses the sequence of words in their vectorial representation (word plus character embedding) to produce higher level representations of the data.*

During training, these representations are gradually updated to facilitate the prediction of the final output.

Function 2.6 (Select Indices) *This function selects the indices $\hat{y}_1$ and $\hat{y}_2$ that identify the snippet from the passage that answers the query.*

Function 2.7 (Evaluate loss function) *This function returns the loss as a function of the inputs $\mathcal{P}$ and $\mathcal{Q}$, the model's parameters and the true indices provided in the training data, y_1 and y_2.*

Function 2.8 (Train) *This function updates the model's parameters to minimize the loss as calculated in Function 2.7.*

Function 2.9 (Evaluate Metrics) *This function computes the metrics used to evaluate the performance of the model. In the SQuAD dataset, these are mainly the Exact Match (EM) and F1 scores as defined in Appendix A.*

Function 2.10 (Load Parameters) *This function loads the initial values for the parameters θ. These can be randomly initialized or imported from a pre-trained model.*

Function 2.11 (Save Parameters) *This function saves the optimized parameters θ.*

Observation: Functions 2.10 and 2.11 are important to retrieve the parameters of a trained model and potentially use them in real world applications.

2.3.1 Modes of Operation

As any machine learning system, ours should have three distinct modes of operation: training, testing and inference.

- **Training mode**: The model's parameters are updated to minimize the loss function as shown in Figure 3.

- **Testing mode**: The model's parameters are kept fixed and tested against EM and $F1$ scores as shown in Figure 5.

- **Inference mode**: The model's parameters are kept fixed and used to infer the answer of a question about a passage as shown in Figure 4.

2.3.2 Functional Diagrams

In this section, we present the functional diagrams for each of the modes of operation. Figure 2 show the Model block, which is subsequently used in Figures 3, 4 and 5 to represent each mode of operation.

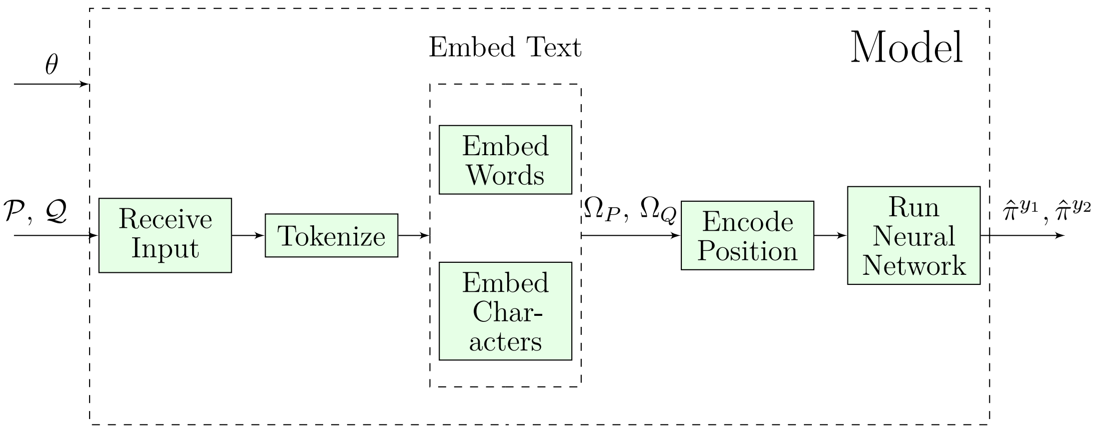


Figure 2 – Model diagram

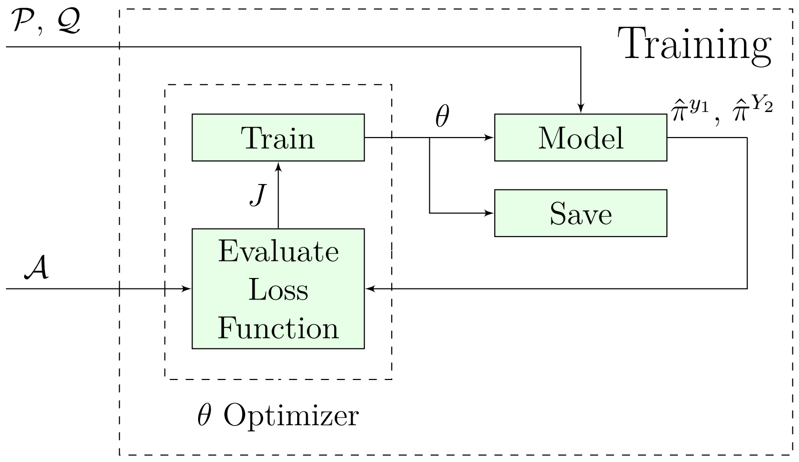


Figure 3 – Training Mode Diagram

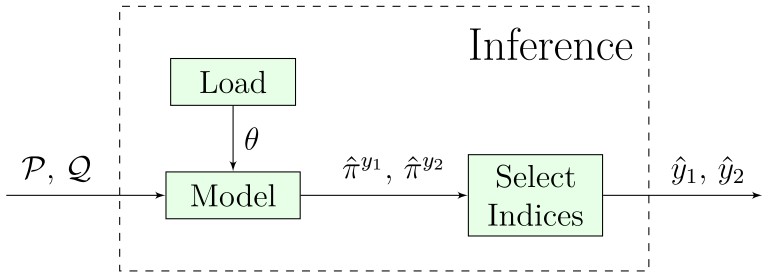

Inference
Load
θ
$\mathcal{P}$, $\mathcal{Q}$
Model
$\hat{\pi}^{y_1}$, $\hat{\pi}^{y_2}$
Select Indices
$\hat{y}_1$, $\hat{y}_2$

Figure 4 – Inference Mode Diagram

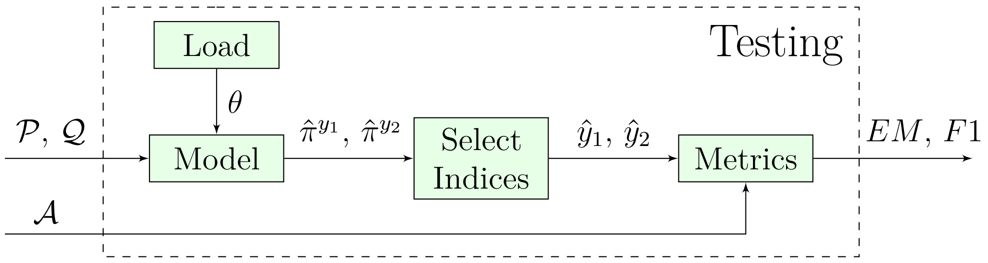


Figure 5 – Testing Mode Diagram

2.4 Requirements Analysis

Requirement analysis is the study and definition of the essential criteria a system needs to meet to exert its functionality appropriately. The following table exposes the requirements of our model together with their rationale and their traceability, which is the set of HLRs that motivate the need for each requirement. The priority feature establishes the importance of each of those requirements on a scale from 0 to 3, with 0 being absolutely necessary and 3 secondary.

Table 1 – Requirements table.

Function	Functional Requirements	Rationale	Trace-ability	Prio-rity
Receive Input	The system should extract passages and queries from a json file and convert them into a python dictionary.	The model is developed on top of Tensorflow's python API.	HLR-1.2 HLR-1.3	1
Tokenize	The system should split both the passage and the query into words and characters.	Text is presented to the model as a set of words and characters that are subsequently embedded in vectorial representations.	HLR-1.1 HLR-1.4	0
Embed Words	The system should represent each word in both passage and query by a one dimensional vector of size WEs.	A machine learning model needs the input data to be represented in a numerical form. On the one hand, a longer vector is expected to contain more information about its respective word, on the other hand it will require more memory and computational power to be processed.	HLR-1.1	0
Encode Position	The system should encode the position of each word in both the passage and the query in a one dimensional vector of size WEs.	The system needs the position of each word to effectively model their interdependence and understand the sentence structure.	HLR-1.1	0
	The system should be able to encode the position of each word in sentences of arbitrary length.	Real world applications contain texts of any length.	HLR-1.4	1
Run Neural Network	The system should implement a sublayer self-attention(X), which gives a score to each element of X given an element of X.	Self-attention is needed to model the interdependence of words in the same piece of text (context).	HLR-1.1 HLR-1.7	1

Function	Functional Requirements	Rationale	Trace-ability	Prio-rity
Run Neural Network	The system should implement a sublayer cross-attention(X, Y), which gives a score to each element of X given an element of Y.	Cross-attention is needed to model the interdependence of words in two different pieces of text, namely passage P and question Q.	HLR-1.1 HLR-1.7	1
	The system should implement a sublayer feed-foward(X) that applies an affine transformation followed by a non-linear function.	This function allows the model to represent the model in another vectorial space.	HLR-1.1 HLR-1.7	1
	Each sublayer should be followed by a vector-wise normalization function, which sets the mean and standard variance across the elements of each vector to 0 and 1, respectively.	Layer normalization facilitates convergence because it avoids covariate shifts inside the model.	HLR-1.1 HLR-1.5	2
	Each layer should be composed of three sublayers: self-attention(P), self-attention(Q), and either cross-attention(P, Q) or cross-attention(Q, P).	Each layer applies different attention mechanisms to the data.	HLR-1.1 HLR-1.7	2
	The system should be a neural network composed of n_pre_layer layers to infer $\hat{y}_1$ followed by n_post_layers to infer $\hat{y}_2$.	The model is composed of several layers to gradually build higher level representations as the data flows through the model.	HLR-1.1 HLR-1.5	1
Select Indices	The system should output a probability distribution over the possible values for $\hat{y}_1$ and $\S\hat{y}_2$.	The probability distribution is needed to compute the negative log-likelihood.	HLR-1.5 HLR-1.6	1
	The system should apply an argmax over the probability distribution to produce $\hat{y}_1$ and $\hat{y}_2$.	The final output of the system that defines answer A are indices $\hat{y}_1$ and $\hat{y}_2$.	HLR-1.5	1

Function	Functional Requirements	Rationale	Trace-ability	Prio-rity
Evaluate Loss	The system should calculate the mean of the negative log-likelihood over the batch.	The loss function calculated over each batch is used to estimate the loss over the whole data set and update the model's parameters.	HLR-1.6	1
Train	The system should train the model using training data batches of size batch_size.	The size of the batches is an hyperparameter defined in the configuration file and is limited by the available RAM.	config	1
	The system should train the model for n_steps steps.	The number of steps together with the batch size define the number of epochs, in which the model is going to be trained.	config	2
	The model should be trained by gradient descent with the optimizer defined in train_type.	The model's convergence rate and final performance are influenced by the specific gradient descent algorithm used to optimize the parameters.	config	1
Evaluate Metrics	The system should compute EM and F1 scores as defined in Appendix A.	These metrics are used to evaluate the performance of the model.	HLR-1.8	2

3 Machine Learning

This chapter aims to introduce the reader to the basic theory of machine learning that was considered relevant for understanding this work and other state-of-the-art models in the SQuAD dataset.

First of all, we briefly introduce neural networks theory in section 3.1, as these models are at the forefront of new developments in machine learning and artificial intelligence. More specifically, we focus in Recurrent Neural Networks (RNN) and Long Short Term Memory (LSTM) because they are the base for most of the state-of-the-art research in NLP and power most models in the SQuAD leaderboard (19). Additionally, we also present Convolutional Neural Networks (CNN), which are one of the key-elements in FABIR implementation, although they are not used in many SQuAD state-of-the-art models. Subsequently, we present word-level and character-level embeddings in Section 3.3, which are basic tools for developing NLP models. Later on, we introduce the concept of attention in section 3.2, which is the cornerstone of this project and will be further exploited in developing the theory behind our fully-attention based model in chapter 5. Finally, in section 3.4 we present some stochastic optimization tools. These are more sophisticated gradient descent methods and are key to improve the model's convergence rate and performance.

3.1 Neural Networks

In this section we briefly introduce the theory of neural networks. For an excellent and comprehensive overview of the topic, we refer to (8).

Neural networks are computational models inspired by the human brain. They are composed of a set of *artificial neurons* that are connected by directed edges. These edges are analogous to synapses in the brain and it is by gradually updating them that we can train a neural network to produce a desirable output.

3.1.1 Feedforward Neural Networks

The standard neural network, which is often referred to as *feedforward*, maps an input x to a target value y, with the goal of approximating some function $y = f^*(x)$. As any parametric machine learning model, a neural network can have its predictions $\hat{y}$ written as function of a set of parameters θ: $\hat{y} = f(x; \theta)$. In order to solve for the value of θ that best approximates $f^*(x)$, we can redefine that goal as minimizing some loss function $L(x, y, \theta)$ which is some dissimilarity measure between $f(x; \theta)$ and $f^*(x)$ when evaluated on data point x. A common example of loss function is the Mean Squared Error (MSE), which can be written as follows:

$$L(x, y, \theta) = (y - f(x; \theta))^2 \tag{3.1}$$

However, evaluating the model in a single data point is not a good measure of its performance because we are interested in finding a model that fits the whole data set. We define then a cost function $J(\theta)$, which is an expectation of the loss function over multiple data points.

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} L(X_i, Y_i, \theta) \tag{3.2}$$

where m is the number of data points where we evaluate the loss function. In this introduction, we will follow the convention that the inputs are given in a matrix X with each example being represented by a row vector X_i. In the machine learning literature, each element of the vector X_i is usually referred to as a *feature*. In our notation, we will use the index j to address these features and hence, X_{ij} represents the feature j of example i. Also, we refer to a single example as a pair x, y.

Note that the cost in equation 3.2 is written as a function of θ. That is because a neural network, as any other parametric machine learning model, "learns" to approximate $f^*(x)$ by changing its parameters θ. In neural networks, that is usually done by gradually updating θ in the direction of the gradient of the cost function w.r.t. θ, which is an optimization method called *gradient descent*.

$$\theta \leftarrow \theta - \epsilon \nabla_\theta J(\theta) \tag{3.3}$$

where $\epsilon \in [0, 1]$ is a hyperparameter known as learning rate.

Neural networks are flexible models and the artificial neurons can be organized in a variety of ways depending on the specific application. However, most neural nets are structured in *layers*, which are sets of neurons that do not share any connections. In that case, the computation flows sequentially from one layer to the next, from the input x to the output y. Because this type of network does not include any cycles in its computational graph, it is called *feedforward network*. Each layer in a feedforward neural network can be interpreted as a function in its own and the whole network as composition of these functions. If $f^{(l)}$ is the function implemented by layer l and we have three layers in the network, we can write $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$.

The last thing left to be defined is the operation performed by each of the functions $f^{(l)}$. Each layer in a neural net consists of an affine transformation followed by a nonlinear function usually known as an *activation function* and represented by g herein. The affine

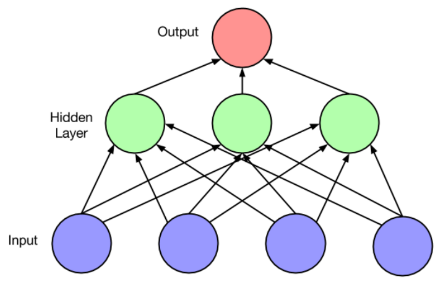


Figure 6 – Schematic of a neural network. Each layer in the network is represented by a set of nodes in a different color. Image Source: (11)

transformation is controlled by the learnable parameters θ which are divided into *weights* W and *bias* b. The entire operation defined by a layer in the network can be written as follows:

$$h^{(l)} = g^{(l)}(Wx + b) \tag{3.4}$$

where $h^{(l)}$ is a vector containing the values of each of the neurons in layer l. The letter h stands for *hidden* because the values of the neurons in the inner layers of the network are not given in the data. The layers in a feedforward network are not constrained to have the same number of neurons and hence, the weights W can have any shape. More formally, $W \in \mathbb{R}^{n_{out} \times n_{in}}$, where n_{in} and n_{out} are the number of neurons in layers l and $l-1$ respectively.

3.1.2 Recurrent Neural Networks (RNN)

The feedforward neural network framework presented above does not model the dependence between different inputs explicitly: the computation of the output y is based on a single example x. In other words, the y_i's are conditionally independent given the inputs x_i's.

That is an important limitation if one wants to model natural language because the information contained in a piece of text is not only given by the meaning of each word, but also by their syntactic and semantic relationship to other words in a sentence (4). To accommodate such a limitation, one could extend the input vector to include a sliding window of words and so implicitly model the interdependence between them. However, that still prevents the neural net from observing the relationship of words separated by more than the predefined window size allows for (11).

A possible solution is to include a feedback loop such that the hidden state at step t of the neural network not only depends on the current input $x^{(t)}$, but also on its previous hidden state $h^{(t-1)}$.

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \tag{3.5}$$

where the superscript t refers to the current processing time step. The intuition behind this kind of feedback is that $h^{(t-1)}$ will sum up the relevant aspects of all the previous inputs up to time step $t-1$. Neural networks that make use of such recurrent feedback are called *Recurrent Neural Networks* or RNNs.

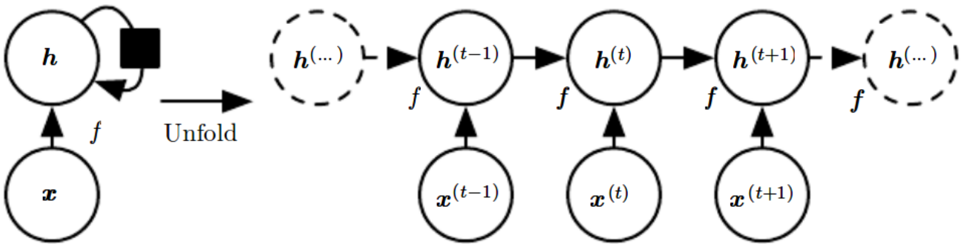


Figure 7 – Schematic of a recurrent neural network unfolded across time steps. The RNN, which is represented by function $h^{(t)} = f\left(h^{(t-1)}, x^{(t)}\right)$, is unfolded and the dependence on the previous time step is made explicit by the illustration. Image Source: (8)

Despite the inclusion of the feedback mechanism, a basic RNN could apply the exact same transformation in each of its layers: an affine transformation controlled by learnable parameters followed by a nonlinear function. In order to show an example of an RNN, a simple model is described in (3.6).

$$h^{(t)} = \sigma\left(Wh^{(t-1)} + Ux^{(t)} + b\right) \tag{3.6}$$

where W, U and b are generic matrices that agree with $h^{(t-1)}$ and $x^{(t)}$ dimensions, and σ is the elementwise non-linear function sigmoid.

Although equation 3.6 takes into account the previous state $h^{(t-1)}$ in every iteration, it has been observed that the optimization of its parameters can be unstable for large sequences (20). That happens, because when the gradient computation flows through a long sequence of time steps, the gradient values tend to either explode or vanish, which hinders the update of the weights in the RNN (20). In order to attenuate this gradient issue, alternative RNN cells have been proposed, such as the Long Short-Term Memory (LSTM) (21) and the Gated Recurrent Unit (GRU) (22). The former is presented in section 3.1.3 to illustrate how it can be used to facilitate the training of RNNs with long sequences. Regarding the use of these two RNN models in the SQuAD dataset, every model in the leaderboard with a published paper used at least one of them for encoding questions or passages, which speaks to their current relevance in the modeling of natural languages for question-answering tasks.

Figure 8 illustrates one possible application of RNNs to machine translation from German to English. The need for a recurrent architecture is clear from the intuition that one can only translate a sentence after reading all its words. The RNN does so by processing one word at a time and updating the value of its current hidden state, which is shown in the figure as the vertical vector. To be clear, h_1, h_2 and h_3 are the same hidden state layer. In Figure 8, they represent h as if unfolded in time to expose the process that underlies the computation in an RNN. In that case, both input (red) and output (blue) are handled by RNNs that process one word at a time.

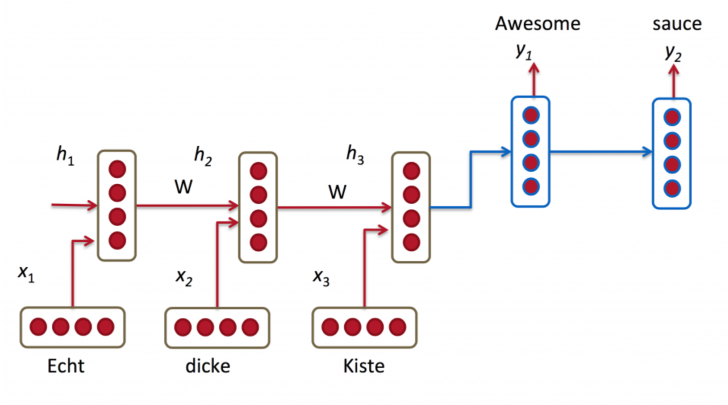


Figure 8 – Schematic of a recurrent neural network applied to machine translation from German to English. Image Source: `http://cs224d.stanford.edu/lectures/CS224d-Lecture8.pdf`.

3.1.3 Long Short-Term Memory (LSTM)

Long Short-Term Memory or LSTM is one type of RNN, which was introduced by Hochschreiter et al. (21) and later updated by Gears et al. (23). It has been successfully used in different NLP tasks, such as machine translation (12) and question answering (1, 17). It is based on the addition of a channel through which information can be selectively saved or deleted. This channel would represent somehow the context or the relation of all words that have already been read, in order to allow the neuron to interpret the next word properly. An LSTM is fully described by Equation 3.7.

$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f) \tag{3.7a}$$

$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i) \tag{3.7b}$$

$$C_t = f_t \circ C_{t-1} + i_t \circ tanh(W_C * [h_{t-1}, x_t] + b_C) \tag{3.7c}$$

$$o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o) \tag{3.7d}$$

$$h_t = o_t \circ tanh(C_t) \tag{3.7e}$$

where x_t is the t^{th} input of the neural network, which for instance can be interpreted as a word in NLP. Additionally, h_t is the hidden state in position t, C_T represents the memory, Ws and bs are parameter matrices that must be trained, and $[h_{t-1,x_t}]$ represents the vertical row-wise concatenation of the previous hidden state and current input.

Variables f_t, i_t, o_t are called *forget*, *input* and *output* gates and their value is between 0 and 1, because of their activation function σ. While, the forget gate f_i is used to define which information the memory C should keep or forget, the input gate i_t defines how much information from $[h_{t-1,x_t}]$ should be added in C. These accesses to memory C can be understood from Equation 3.7c. Finally, output gate o_t defines how much information from memory is relevant for the next hidden state, as shown by Equation 3.7e.

3.1.4 Convolutional Neural Networks

In many applications, it is desirable that the operations applied by the neural network be translational invariant. For instance, that is the case in image classification, where moving the object of interest should not change whether it is a dog or a cat. Convolutional neural networks were developed precisely to address that problem (24). The idea is that by applying the same operation over all the parts of an image, one can detect some patterns regardless of the exact position where they appear. That approach gave rise to architectures known as Convolutional Neural Networks (CNNs) that have been extremely successful in computer vision (25).

In our case, CNNs are mainly used to model local attention, i.e. the relationship between words that appear close to each other in a sentence. The intuition is that there are patterns, such as the interdependence between verb and subject, that are often observed within distances of only a few words and that can be modeled quite generally. Therefore, ideally, a convolution could capture such patterns even if a sentence is paraphrased and the exact position of each word is changed.

The ideas described above already illustrate the mechanism of CNNs, but to effectively discuss their application to our model, we need a more formal definition. In order to define a convolution, one must first name three dimensions in the input: height h, width w and number of channels $n_{channels}$. Subsequently, one must define three dimensions in the convolution kernel: kernel length H_{height}, kernel width H_{width} and number of filters $n_{filters}$. While the input $I \in \mathbb{R}^{h \times w \times n_{channels}}$ is described by a three dimensional tensor, the kernel is a four dimensional tensor $H \in \mathbb{R}^{H_{height} \times H_{width} \times n_{channels} \times n_{filters}}$. Note that $n_{channels}$ is defined exclusively by the input, but the kernel size must agree with it. Finally, the convolution output $O \in \mathbb{R}^{h - H_{height} + 1 \times w - H_{width} + 1 \times n_{filter}}$ between the input I and the kernel

H is defined by (3.1.4).

$$O_{i,j,k} = Conv(I,H) \sum_{m=1}^{n_{channels}} < I[i : i + H_{len} - 1, j : j + H_{width} - 1, m], H[:,:,m,k] > \quad (3.8)$$

where $< A, B >$ is $Tr(AB^T)$, " : " represents all indices in the respective dimension and $a : b$ all indices between a and b, which includes a and b.

3.2 Attention

In recent years, attention mechanisms have been used with success in a variety NLP tasks, such as machine translation (12, 2), natural language inference (26, 27) and question answering (1). It can be defined as a mechanism that gives a score α_i to a vector x_i from a set $X = [x_1, ..., x_n]$ with respect to a vector c. This score is a function of of both X and c and is shown in its most general form in (3.9).

$$s_i = f(x_i, c) \quad (3.9a)$$

$$\alpha_i = \frac{exp(s_i)}{\sum_{i=1}^{n} exp(s_i)} \quad (3.9b)$$

where s_i and α_i are scalars and f is a function. Equation (3.9b) is called softmax operation and guarantees that the sum of all weights α_i is equal to 1. A large weight α_i means that the vector x_i is somehow strongly related to c.

In (3.9), f essentially defines a score function that measures the relative importance of x_i given the context c. In the literature, two alternatives for f have been proposed: additive (12) and multiplicative (28) attentions. Both types of attention mechanisms are presented in (3.10) below.

$$f(x_i, c) = \begin{cases} W_3 g(W_1 x_i + W_2 c) & \text{(additive)} \\ x_i^T W_1 c & \text{(multiplicative)} \end{cases} \quad (3.10)$$

where W_1, W_2 and W_3 are learnable parameters and g is a elementwise nonlinear function. For small vectors, additive and multiplicative attention mechanisms have been shown to produce similar results (29). The advantage of multiplicative attention is that it can be implemented more efficiently using matrix multiplication, which is highly optimized in modern GPUs. Nevertheless, when the vectors have high dimensionality, the multiplicative attention might result in large values in the softmax operation in (3.9b), which leads to small gradients that hamper efficient optimization of the parameters. That can be addressed by diving $f(x_i, c)$ by the square root of the hidden state size (2).

In NLP, attention mechanisms are often used to identify the most relevant words x_i in a text X for a specific task. In order to illustrate and motivate their use, a machine translation example is shown, which was the task addressed by the paper that first introduced attention mechanisms in NLP (12).

When translating a sentence, it is intuitive that one should not translate word by word because the context and the interaction among words play a major role in defining its semantics. RNNs solve that limitation by means of their feedback mechanism, which is supposed to encode all the information regarding the original sentence in a single context vector c. Typically, that context vector is simply the hidden state of the RNN after processing all words in a sentence. However, c has a pre-defined number of features, which is independent of the input sequence size, and hence might not be enough to effectively represent the original sentence. That is exactly the limitation that attention mechanisms were designed to tackle.

The model proposed in (12) is based on a encoder-decoder architecture common in sequence-to-sequence models. The translation is then made in two steps: (i) a RNN-encoder computes the context vector c of the original text and subsequently (ii) a RNN-decoder uses that context to output the translated text word by word. However, traditionally the only input to the decoder was the last hidden state of the encoder and hence the context c suffered from the limitations outlined above. Bahdanau et al.(12) address that problem by changing the input to the decoder in such a way that it contains information from all previous hidden states in the decoder. Their insight is that each word in the translation does not depend equally on every word in the original text. Therefore, the input to the decoder must be a weighted average of the encoder hidden states, assuming that each of them is associated with a single word in the original sentence. That weighted average is the attention mechanism as defined in (3.9). In practice, for every new word y_t output by the model, the input to the decoder will be given by a new context vector c_t, which is a function of $\alpha^T X_{1:t}$. As shown in (12), using attention increases model performance in comparison to models without attention, specially for long sentences, in which c is more likely to fail in representing the whole context.

3.2.1 Self-attention

Despite their theoretical capability, RNNs can only memorize a limited portion of a passage and the interdependence between distant words is poorly modeled (14). The mechanism used to mitigate that problem is called *self-attention* or *self-aligning* because it models the relationship between words in the same piece of text. That way, (3.2) can be used to establish a connection between words in any position of the text, producing a more effective representation of the context.

3.2.2 Cross-attention

Cross-attention is conceptually similar to self-attention, but instead models the dependence between words in two different pieces of text. That is the type of attention introduced in machine translation (12) which is also fundamental to QA, as it is by means of that mechanism that the model understands the relationship between a question and its answer as part of a text. Indeed, most models in the SQuAD ranking use some sort of cross-attention to determine which words in the question are the most relevant given a word in the passage. For clarity, we will refer to this sort of cross attention as *passage-over-query*. It is also possible to apply cross-attention in the opposite sense, which would be equivalent to a *query-over-passage* attention.

3.3 Word and Character Embedding

Embedding is the process of associating a word or a character from a piece of text to a vector so that it can be processed by a digital computer. In that regard, that process is analogous to associating colors to a triad of numbers in the RGB format.

The most intuitive way to represent a word or any categorical feature to a machine learning model is to label it with an arbitrary number. The most common approach for doing so is called one-hot encoding: each feature is represented by a vector the size of the vocabulary (number of categories) where all elements are set to zero except for the position that identifies the pertinent word (category). However, even though that allows the computer to distinguish the different categories, it fails to capture any relationship between them. That is especially relevant in NLP because words share meanings and morphology that could be explored by the model, but are completely ignored in this simple labeling process.

Word embeddings are an interesting alternative to build effective representations of words. In (30) and (31), each word is represented by a vector of size d, which is randomly initialized and trained by a neural network to minimize some cost function. This process has been proven to generate meaningful word representations with remarkable properties. Namely, similar words are in general given by embeddings that are close in the corresponding vectorial space.

That idea has been further extended to process characters (32, 33, 34). The advantage of a character-level embedding is that it is more robust to misspellings and can model morphological similarities that often translate into interesting semantic or grammatical relationships.

3.4 Training Algorithms

Training is the process of optimizing the parameters θ in the neural network to minimize its cost function $J(\theta)$, which is described in (3.2). Ideally, J would be evaluated over the whole data set, but that is computationally expensive for most interesting models. Therefore, a smaller number of examples to which we will refer as *batch* is used to estimate J and update the model's parameters. Given that the batches are selected randomly, this process is called Stochastic Gradient Descent (SGD) (8, p.150). From the perspective of SGD, the cost function evaluated in a batch provides a local estimate of the gradient that can be used to update θ in a direction that minimizes $J(\theta)$.

Given that the loss function is not convex in neural networks and that the estimate of $J(\theta)$ is noisy, the learning rate ϵ in (3.3) needs to be carefully fine-tuned: a high value could cause the model to diverge, whereas a low one could retard the learning process. The choice of this learning rate is paramount for the convergence of the model and several extensions to the vanilla SGD algorithm have been proposed to improve the overall performance or automatically define the learning rate (35, 36). These methods are presented in the following sections.

3.4.1 Annealing Learning Rate

The simplest addition to SGD is the introduction of a variable learning rate ϵ that is reduced as the model approaches convergence. That helps to avoid large increments in the region of the local minimum. One possible way to implement an annealing learning rate is to decay it by a constant factor as in (3.11).

$$\epsilon_t = \frac{\epsilon_{t-1}}{1 + \rho t} \tag{3.11}$$

where t is the time step or iteration number and ρ is a decay rate, specifying by how much the learning rate should be reduced at each step. By hampering large oscillations around the local minimum, an annealing learning rate has been shown to improve performance and speed up the training process.

3.4.2 Momentum

One can also speed up training on a per-dimension basis by applying a momentum algorithm. In that case, the update in (3.3) will no longer be defined exclusively by the gradient, and we will rewrite it more generally as $\Delta\theta$.

$$\theta_{t+1} = \theta_t + \Delta\theta_t \tag{3.12}$$

Momentum algorithms update the parameters with the following idea: the optimization method should accelerate the progress along dimensions in which gradients consistently point in the same direction and slow it down in dimensions for which the sign of the gradient changes regularly. That is done by including the previous $\Delta\theta$ when computing the update as follows:

$$\Delta\theta_t = \rho\Delta\theta_{t-1} + \epsilon\nabla_{\theta,t} \tag{3.13}$$

where ρ is again a constant decay rate, but this time it controls the influence of previous updates. It is easy to see that in (3.13), the learning rate gradually increases if the sign of the gradient is consistent across multiple steps and is damped otherwise. Momentum updates help SGD to navigate ravines, i.e., regions where the surface curves are steeper in a few dimensions, and a single learning rate would lead to slow convergence or divergence (37).

3.4.3 Adagrad

Adagrad (35) is another gradient descent algorithm that also speeds up training on a per-dimension basis. In the case of Adagrad, each dimension has an unique learning rate, which is defined as in (3.14).

$$\Delta\theta_t = -\frac{\epsilon}{\sqrt{\sum_{\tau=1}^{t} \nabla_{\theta,\tau}^2}}\nabla_{\theta,t} \tag{3.14}$$

Note that, despite the influence of ϵ as a global learning rate hyperparameter, each dimension has a different learning rate, which is defined by the inverse of the gradients magnitude. Therefore, Adagrad attributes a small learning rate to high gradients and vice-versa, which is an attractive property for deep neural networks where the scale of the layers may vary by orders of magnitude.

The main advantage of Adagrad is that it, in theory, it eliminates the need to tune the learning rate manually. Nevertheless, Adagrad is still sensitive to the global learning rate ϵ and, due to the continual accumulation of squared gradients, the updates tend to converge to zero, stopping the training completely.

3.4.4 Adadelta

Adadelta (36) tries to solve Adagrad's limitations by accumulating the gradients over a pre-defined window. That prevents the learning rate from converging to zero as the denominator in (3.14) no longer grows indefinitely. In practice that is done by replacing the sum over the gradients by a running average $E[\nabla_\theta^2]_t$, which is defined as follows:

$$E[\nabla_\theta^2]_t = \rho E[\nabla_\theta^2]_{t-1} + (1-\rho)\nabla_{\theta,t}^2 \tag{3.15}$$

where ρ is a hyperparameter that defines the influence of previous updates. The Adadelta algorithm can then be defined as in (3.16).

$$\Delta\theta_t = -\frac{\epsilon}{\sqrt{E[\nabla_\theta^2]_t + c}}\nabla_{\theta,t} \tag{3.16}$$

where c is a constant added for numerical stability.

3.4.5 Adam

Adam (38) is another extension to the Adagrad algorithm that computes an exponentially decaying average for both past gradients and past squared gradients. We refer to those as the first moment (mean) and second moment (variance), which are computed as follows:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_{\theta_t} \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)\nabla_{\theta_t}^2 \end{aligned} \tag{3.17}$$

where β_1 and β_2 are hyperparameters set to values close to 1. However, at $t = 0$ both β_1 and β_2 are initialized at zero, and hence m_t and v_t are biased towards zero, which requires a bias-correction as presented below.

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{3.18}$$

The update used in Adam is then derived as in Adagrad and Adadelta and can be written as follows.

$$\Delta\theta_t = -\frac{\epsilon}{\sqrt{\hat{v}_t + c}}\hat{m}_t \tag{3.19}$$

where c is again a constant which is set to a low value to avoid numerical instability.

4 Related Work

Since the SQuAD dataset has been made public in June 2016, the research community has produced ever more accurate models. For reference, the EM and F1 scores for each of five of the best performing models in the SQuAD leaderboard (19) are shown in Table 3. Note that even though some of them achieve significantly better performances when combined with other models (ensemble), only the scores obtained with *single models* are presented, so that they can be directly compared to our work.

Table 3 – EM and F1 scores for referenced papers.

Model	EM (%)	F1 (%)
r-net (14)	76.461	84.265
Interactive Attention over Attention Reader (AoA) (16)	75.821	83.843
Dynamic Coattention Networks (DCN) (15)	75.087	83.081
Reinforced Mnemonic Reader (17)	73.188	81.816
Bidirection Attention Flow (BiDAF) (1)	67.974	77.323

It is worth noting that all models in Table 3 are RNN-based, and hence very similar conceptually. Our model represents a complete departure from that approach as it replaces the recurrent processing with an attention mechanism. In the following sections, we introduce and compare these five models against our own regarding some key design choices. At the last section, we also present the work introduced in (2), which was the first to propose a fully attention-based model as the one we put forward in this project.

4.1 Pre-processing

Most NLP systems process information on the word level, hence pieces of text must be split into words before being fed to the machine learning model. That process is called tokenization and has been a backbone of NLP since its dawn (39).

Not all models in Table 3 cite which tokenizer they employed to split the passages and queries into words. Nevertheless, Stanford's PTB Tokenizer (40) is probably the most popular open source software package and has been used in (14, 15). In our model, we gave preference to the NLTK tokenizer (41) because it is python native, and hence straightforward to integrate into our pipeline.

4.2 Word Embedding

All models in the SQuAD leaderboard rely on word embeddings as described in section 3.3. However, the SQuAD dataset is relatively small, which prevents the embeddings from being learned from scratch. Therefore, the word embeddings are initialized with Global Vector for Word Representation (*GloVe*) vectors (31) and kept fixed during training, whereas only the representations of unknown words are learned via gradient descent with the rest of the neural network. In theory, all word embeddings could be fine-tuned the same way, which would allow them to adapt to the specific topics of the SQuAD dataset, but that has been found to lead to overfitting (15). In that regard, the only difference between the models in Table 3 is the dimension of these vectors, which is either 100 (17) or 300 (14, 1) in accordance to the available pre-trained embeddings in `https://nlp.stanford.edu/projects/glove/`.

In our model, we also use pre-trained *GloVe* vectors and following a similar strategy, we only train the embeddings for unknown words as described in Section 5.1.3.

4.3 Character-level Embedding

The r-net (14), Reinforced Mnemonic Reader (17) and BiDAF models (1) also use character-level embeddings, which have been found to be robust to grammatical and orthographic errors (42), while being more effective at representing unknown words (33, 14).

In (14) and (17) each word is associated to a character-level embedding, which is encoded via a bidirectional RNN applied to character embeddings similar to those used for words. The last hidden state of that RNN is then concatenated with the word embedding to produce the final representation of each word that is fed to the rest of the model. Note, that in this case there are two different levels of embeddings: (i) the vectorial representation of each character (input to the RNN) and (ii) the encoding of the set of characters of each word (output of the RNN). Only the latter is presented to the subsequent layers of the model, but the whole process is learned by gradient descent.

In (1) the character-level embedding is done similarly, but the encoding is performed via a Convolutional Neural Network (CNN). The technique is inspired by (34) and consists in applying a CNN with n filters over the set of m one-dimensional character embeddings of each word. Naturally, this CNN extracts nm features, but a max-over-time-pooling is used to extract only the most important features across the m embeddings, producing a fixed-size representation of n dimensions for each word. The output of that CNN is concatenated with the respective word embedding and processed by a two-layer Highway Network (43) to produce the final representation of each word.

Given that one of the critical features of FABIR is the absence of sequential operations, we chose to create character-level embeddings using CNNs to maintain that property. The architecture used to extract those embeddings is similar to the one proposed in (1) and is detailed further in the following chapter.

4.4 Contextual Embedding

An effective representation of words and characters is still not enough to capture the semantics of a piece of text. Indeed, the relationship between words in the context of the passage needs to be accounted for as well. All five models in Table 3 tackle this problem through bidirectional RNNs (12, 1). Differently, from a traditional unidirectional RNN that can only relate each word to their precedents, the bidirectional RNN captures the dependence between every word in the passage by processing each sentence in both directions. Figure 9 depicts a bidirectional RNN similar to what is used by (14, 15, 16, 17, 1).

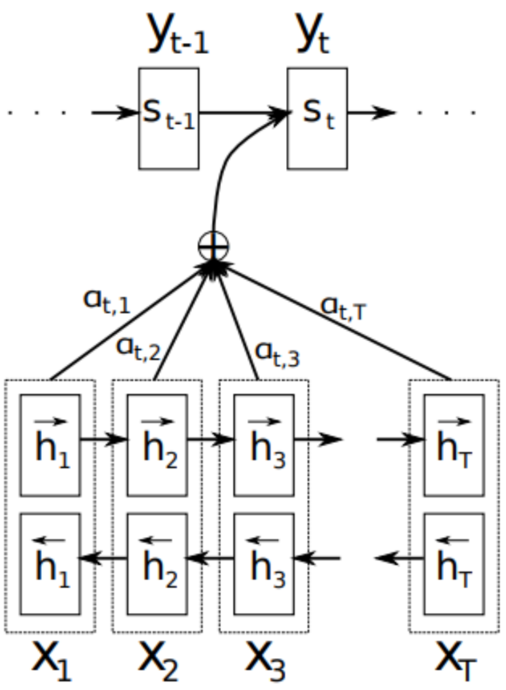


Figure 9 – Illustration of a bidirectional RNN with an attention mechanism, where x_t is a word in the passage, h_t is a hidden state and α is the attention weight. Image Source: (12)

However, the bidirectional RNN still cannot adequately represent the relationship between distant words. The difficulty in learning such dependencies can be measured by the length of the paths that forward and backward signals have to traverse in the network (20). According to Vaswani et al. (2), if n is the distance between a pair of words, in bidirectional RNNs this path is still of length $\mathcal{O}(n)$, whereas in a fully attention-based model that path is only $\mathcal{O}(1)$ long. That is a key difference between our model and previous works on the SQuAD dataset as it might translate into a more accurate representation of long pieces of text.

In all five models cited above, questions and passages are processed individually by the bidirectional RNNs to produce contextual representations. The relationship between the two is only modeled in the subsequent layers through different attention mechanisms that we present in the following section.

4.5 Attention Mechanisms

Attention mechanisms as described in 3.2 are a crucial part of state-of-the-art question-answering and other applications of neural networks to NLP. All models in Table 3 use some attention mechanism, though with some significant differences that we outline below.

4.5.1 Cross-attention

Cross-attention is a necessary mechanism to model the relationship between question and passage, which is essential to produce a pertinent answer. As explained in Section 3.2.2, we divide cross-attention into two different types of mechanisms, according to the direction in which it is applied.

passage-over-query: Roughly speaking, this type of attention attributes a score to each word in the question, given a word from the passage. All models in Table 3 implement *passage-over-query* mechanisms, the main difference being that (14, 17, 1) use additive attention, whereas (16, 15) use multiplicative attention[1].

query-over-passage: This would be the opposite of the mechanism described above, i.e., it computes a score for each word in the passage, given a word from the question. That is the way in which *query-over-passage* is applied in (15, 16), but an alternative is presented in BiDAF (1) and r-net (14). They compute an attention weight for each word in the passage with respect to a single vector that represents the context of the question as a whole. It is also worth noting that the *query-over-passage* attention mechanism in r-net (14) uses a sigmoid function instead of softmax, which motivates its name, "gated attention".

In all cases, cross-attention is applied directly after encoding passage and question in an RNN network.

In FABIR, we tested both types of cross-attention, and the results are shown in Chapter 6. Nevertheless, it seems more natural to employ *query-over-passage* because

[1] See Equation 3.10 in p. 43 for a discussion of different types of attention.

the required output in SQuAD is a probability distribution over the passage, and hence, intuitively, the attention over P would be more useful.

4.5.2 Self-attention

As described in section 3.2.1 self-attention mechanisms are essential to facilitate the modeling of the relationship between distant words. In the related work, only (14) and (17) use any self-attention mechanism and both apply it on the query-aware context, which is the output of the cross-attention. That contrasts with our model in which the self-attention is the first operation in the pipeline.

Another interesting difference is that in (17) the self-attention is represented by a matrix where the diagonal is set to zero (zero identity), preventing a word from being aligned with itself. Conversely, in (2) the diagonal concentrates the highest values (high identity) relating attention to a concept of similarity: the model should associate similar words having the identity as a reference for maximum. In FABIR, we have only experimented with self-attention matrices as in (2) because it naturally arises from the position encoding described in Section 5.2.

4.6 Attention Based Model - Google's Transformer

Instead of using an RNN to describe the interaction between words in a sentence, Google (2) suggests a different mechanism that can be implemented with feedforward neural networks. Their method, which they call the Transformer, is based on the addition of a unique position encoding vector in every word embedding, followed by an attention mechanism, which is described in next Section.

4.6.1 Transformer's Attention Mechanism

The backbone of the Transformer is its attention mechanism. It is computed as a function of the input matrices $U \in \mathbb{R}^{U_{len} \times d_{model}}$, $K \in \mathbb{R}^{K_{len} \times d_{model}}$ and $V \in \mathbb{R}^{V_{len} \times d_{model}}$, as shown in (4.1). Following the nomenclature introduced in (2), we will refer to U, K and V as queries, keys and values respectively.

$$att(U, K, V) = softmax\left(UK^T\right)V \tag{4.1}$$

where $softmax()$ represents the softmax operation over the rows of UK^T and $att(U, K, V) \in \mathbb{R}^{U_{len} \times d_{model}}$ is a linear combination of V defined by $softmax(UK^T)$. We can still add an affine transformation in (4.1) to allow the neural network to learn higher level representations, as shown in (4.2).

$$att(U, K, V) = softmax\left(UW_{UK}K^T\right)V\,W_V \tag{4.2}$$

where $W_{UK}, W_V \in \mathbb{R}^{d_{model} \times d_{model}}$ are weight matrices. Additionally, Google (2) suggests a multi-head attention, in which the attention in the i^{th} head would be computed by (4.3):

$$att_i(U,K,V) = softmax\left(U\ W_{U,i}(K\ W_{K,i})^T\right) V\ W_{V,i} \tag{4.3}$$

where $W_{U,i}, W_{K,i}, W_{V,i} \in \mathbb{R}^{d_{model} \times d_{head}}$ are weight matrices.

Finally, the multi-head attention is computed by (4.4).

$$att_{MultiHead}(U,K,V) = concatenate\,(att_1, ..., att_{n_{heads}})\, W_O \tag{4.4}$$

where $W_O \in \mathbb{R}^{(n_{heads} * d_{head}) \times d_{model}}$ is a weight matrix. Considering that the product $n_{heads} * d_h$ is constant, the increase of the number of heads might improve the performance of the model at little computational cost, given that the number of operations grows at a smaller rate (2).

4.6.2 Transformer's Self-Attention Mechanism

Given a passage $\Omega_P = [\omega_{p,1}, \omega_{p,2}, ..., \omega_{p,n}]$, where $\omega_{p,i}$ is a vector which represents the i^{th} word of the sentence, the sum of P with the position encoder output $E = [e_1, e_2, ..., e_n]$ is described in Equation 4.5.

$$P_{encoded} = [\omega_{p,1} + e_1, \omega_{p,2} + e_2, ..., \omega_{p,n} + e_n] \tag{4.5}$$

where e_i is the i^{th} unique vector output from the positional encoder. Subsequently, this $P_{encoded}$ goes through a self-attention mechanism, which is shown in Equation 4.6.

$$att_{self}(P_{encoded}) = att_{MultiHead}(P_{encoded}, P_{encoded}, P_{encoded}) \tag{4.6}$$

where $U = Q = V = P_{encoded}$ is substituted in (4.4).

This operation outputs a matrix $att_{self}(P_{encoded})$ with the same dimensions of $P_{encoded}$, in which the output is a function of all words in the original sentence and their absolute positions. The influence of the absolute position is guaranteed because of the sum of the word embedding with the positional encoder vector in (4.5).

Naturally, the self-attention mechanism can be employed on multiple layers in the architecture, not only on the inputs $P_{encoded}$. Indeed, the architecture tested in (2) was composed of six layers, each one of them with a self-attention operation. In that case, the self-attention is always calculated for the output of the previous layer, $P_l = att_{self}(P_{l-1})$.

It is important to highlight that the self-attention mechanism shows two improvements against the traditional RNN approach. The first one is that all words can interact with each other in only one iteration, while in RNNs that requires k iterations, where k is the distance between two words. The second improvement is the possibility of parallel computation. While RNNs must perform n sequential steps, where n is the length of the sentence, the self-attention mechanism performs a similar evaluation in a single step, which may reduce processing time.

4.6.3 Transformer's Cross-Attention Mechanism

A similar mechanism can be defined for cross-attention to compute the attention between two different pieces of text. Given two passages P and Q, the attention from Q to P is defined in (4.7).

$$att_{cross}(P, Q) = att_{MultiHead}(P, Q, Q) \tag{4.7}$$

We refer to this attention as *query-over-passage* because the output of this operation is a score for each element in P. Note that this operation is not commutative and hence we cannot guarantee $att_{cross}(P, Q) = att_{cross}(Q, P)$.

These self and cross attentions operations are independent of the size of P and Q. That property makes them especially attractive to real-world NLP applications, which often involve processing inputs of different lengths.

To the best of our knowledge, these attention mechanisms for encoding sentences have not yet been used for question-answering models, but only for Machine Translation in (2). As this new method showed better results than traditional approaches based on RNNs both in quality of translation and in the required processing time for training, it motivates its use for other NLP applications, such as question-answering.

Table 4 – Architecture comparison of the different models in the SQuAD leaderboard. NI stands for “No Information”.

Characteristic / Model	**r-net**	**DCN**	**AoA**	**Mnemonic Reader**	**BiDAF**	**FABIR**
Tokenizer	Stanford PTB	Stanford PTB	NI	NI	NLTK Tokenizer	NLTK Tokenizer
Word Embedding	Glove 300d	Glove 300d	NI	Glove 100d	Glove 300d	Glove 100d
Character Embedding	RNN-based	None	None	RNN-based	CNN-based	CNN-based
Attention	Additive	Multiplicative	Multiplicative	Additive	Additive	Multiplicative
Self-attention	High identity	None	None	Zero identity	None	High identity
Passage-over-query	Softmax context words	Softmax context words	Softmax context words	Softmax context words	Softmax context words	Design Choice at layer level
Query-over-passage	Sigmoid query context	Softmax query words	Softmax query words	None	Softmax query context	Softmax query words

5 FABIR: Fully Attention-Based Information Retriever

This chapter introduces the Fully Attention Based Information Retriever model (FABIR), which has been developed for the QA task proposed in (3). The breakthrough in FABIR is that it is capable of processing sequences of variable length without resorting to recurrent models, which currently dominate the state-of-the-art in NLP. Indeed, all models in the SQuAD leaderboard (19) are RNN-based. That innovation represents a challenge in itself because being a novel approach, it lacks references or previous works that could guide our design choices.

FABIR is based on a machine translation model by Google (2). Their idea is to replace the sequential processing of words in a piece of text by an attention mechanism capable of representing the interdependence of all words in a matrix form. That allows us to make predictions in a single step and increase the amount of computation that can be parallelized both at training and testing times, which is interesting for real-world applications.

Although machine translation and question answering are both in the NLP domain, they are still substantially different, which prevents us from applying Google's model as-is. Therefore, there is a series of key design choices that need to be made to bring their idea of a fully attention-based model to the SQuAD ranking. To do so, we follow some heuristics and try to replicate some of the features found in state-of-the-art QA models (14, 15, 16, 17, 1).

For a clear presentation of the architecture of our model, we divide it into four different processes as outlined below.

1. **Pre-processing**: The raw query and passage texts are split into words (tokenization) and each of them is associated with an embedding vector.

$$Pre\_Processing : \mathcal{Q}, \mathcal{P} \rightarrow \Omega_Q, \Omega_P \tag{5.1}$$

   where $\mathcal{Q}$ and $\mathcal{P}$ are respectively the query and the passage in text format. Similarly, $\Omega_Q \in \mathbb{R}^{Q_{len} \times d_{model}}$ and $\Omega_P \in \mathbb{R}^{P_{len} \times d_{model}}$ are the matrix representations of the query and the passage as a series of embedding vectors. Therefore, the rows in these matrices should represent somehow the meaning of each word in the text, as shown in Section 3.3.

2. **Position encoding**: In this step the position of each word in $\mathcal{Q}$ and $\mathcal{P}$ is encoded

in a vector of size d_{model}.

$$Pos\_Encoding : Q_{len}, P_{len} \rightarrow E_Q, E_P \tag{5.2}$$

where $E_Q \in \mathbb{R}^{Q_{len} \times d_{model}}$ and $E_P \in \mathbb{R}^{P_{len} \times d_{model}}$ are the positional encoding matrices for the query Q and passage P, respectively.

3. **Processing Stage**: At this stage, the position encoded embedding vectors go through a sequence of operations, which can be any combination of the following:

   – Self-attention: computes the attention between words from the same piece of text;

   – Cross-attention: computes the attention between words from two different pieces of text;

   – Feedforward: applies a feedforward neural network with a single hidden layer;

   – Normalization: normalizes the values of each vector to mean zero and standard deviation one.

   Herein, we will refer to each of these operations as *sublayers*. The model is also structured in *layers*, which are merely a combination of the sublayers defined above. The processing stage consists of a stack of those layers and it outputs the final high-level representations of Q and P that will be used to compute the answer in the following stage. The whole processing stage can be summarized by its input and output as follows:

$$Y_{proc} : \Omega_Q + E_Q,\ \Omega_P + E_P \rightarrow Q_{y_1},\ P_{y_1},\ Q_{y_2},\ P_{y_2} \tag{5.3}$$

   where $Q_{y_1}, Q_{y_2} \in \mathbb{R}^{Q_{len} \times d_{model}}$ and $P_{y_1}, P_{y_2} \in \mathbb{R}^{P_{len} \times d_{model}}$ are the outputs from processing layers used to compute y_1 and y_2. It might also be desirable to calculate y_1 and y_2 from the same high-level representations. That is a special case of (5.3), where $P_{y_1} = P_{y_2}$ and $Q_{y_1} = Q_{y_2}$.

4. **Answer selection**: This process uses the output from the processing layers to compute the start and end positions of the answer in the passage, as shown in (5.4).

$$Y_1\_Selector : Q_{y_1},\ P_{y_1} \rightarrow \hat{\pi}^{y1} \tag{5.4a}$$

$$Y_2\_Selector : Q_{y_2},\ P_{y_2},\ (\hat{\pi}^{y1}) \rightarrow \hat{\pi}^{y2} \tag{5.4b}$$

   where $\hat{\pi}^{y1}, \hat{\pi}^{y2} \in \mathbb{R}^{P_{len}}$ are vectors in the P_{len}-probability simplex, representing the probability of each word in the passage being the start and end indices, respectively.

Note that the use of $\hat{\pi}^{y1}$ to compute $\hat{\pi}^{y2}$ is optional, as it is only necessary when $\hat{y}_1$ and $\hat{y}_2$ are calculated from different layers.

Considering the outputs of the selector, a good model is supposed to output high probabilities in y_1 and y_2, as given by the ground truth.

In the following sections we present the architecture of each of these stages of the model in detail.

5.1 Pre-processing

The pre-processing was split into three steps, which are described in the next sections.

5.1.1 Tokenization

The raw text is split into words and words into characters. That process, which is called tokenization, is intricate and a vibrant research area in itself (39). Fortunately, efficient open source packages are available for this task, and we opted for the nltk API (41) because it is python native and hence, easy to integrate to the rest of the model. An alternative to this tokenizer is the Stanford CoreNLP Natural Language Processing toolkit (40), which has been used by other state-of-the-art models (14, 15).

5.1.2 Word-embedding

After the word tokenization, each word must be associated with a vector that somehow represents its meaning. That mapping was performed using GloVe word-to-vector representation “6B” (31), which was developed with Wikipedia articles. That is an interesting feature given that the SQuAD question-answers pairs were extracted from the same corpus.

From the 115,308 unique words that were identified by the tokenizer in the whole SQuAD dataset, 81% had an embedding mapping into GloVe. The other 19% is composed mostly of misspelled words, rare words, proper nouns, and numbers. To deal with these unknown words, we created two different categories: “uncommon” and “common” words. This classification was based exclusively on the number of times that each word appears in the corpus. Each “common” word was associated with a different random embedding, which can be trained by backpropagation with the rest of the model. Given that these words appear repeated times in the corpus, the model can build a representation that might contribute to the final prediction. Conversely, “uncommon” words data is to sparse to allow for the training of an effective embedding and all of them were tied to the same “UNKNOWN” vector.

For clarity, we will refer to the word embeddings described above as ω_w to distinguish it from the character-level embeddings ω_c discussed in the following section.

5.1.3 Character-embedding

Character-embedding produces a vector representation ω_c for each word that is based on its set of characters rather than the context in which it is commonly found. That representation can be built by RNNs (14, 17), but as we want to avoid sequential operations in FABIR, convolution (44) was the preferred method to process characters inside a word.

Given a word with length l, one can represent it by a three dimensional tensor $C = [c_1, c_2, ..., c_l] \in \mathbb{R}^{l \times n_{char} \times 1}$, where $c_i \in \mathbb{R}^{n_{char}}$ is the word's i^{th} character and n_{char} is the character embedding size. A convolution operation is applied to C with a kernel $H \in \mathbb{R}^{H_{height} \times n_{char} \times 1 \times n_{char\_out}}$, where n_{char_out} is called character-level embedding size and represents the number of filters in the kernel. This convolution is described in (5.5)[1].

$$C_{Conv} = Conv(C, H) \tag{5.5}$$

where $C_{Conv} \in \mathbb{R}^{l - H_{height} + 1 \times 1 \times n_{char\_out}}$ is the output and H is a four dimensional weight tensor to be trained.

Subsequently, C_{Conv} goes through a max pooling operator in its rows, which produces a vector with constant size regardless of the number of characters l in each word. That operation is often referred to as max-over-time pooling (44). The whole operation can be generally defined as follows:

$$\omega_c = max_{row}(C_{conv}) \tag{5.6}$$

The whole pipeline described above is depicted in Figure 10, where for instance the convolution operation is applied over the characters with four filters. Note that to implement a convolution operation efficiently, we represented all words with the same length l, which is achieved by adding zeros to the shorter ones (*zero-padding*).

After the pooling operation, a non-linearity is applied with *tanh* to squeeze the results to $[-1, 1]$. The output is then concatenated with the word-embedding described in the previous section and is finally passed through a Highway Network with two layers (1, 43), as described in (5.7).

$$\omega = Highway(Highway([\omega_w, tanh(\omega_c)])) \tag{5.7}$$

where ω is the final word embedding, which will be used in the next processing steps. From here on, the term word embedding will be used to refer to ω, which is composed by both

[1] See p. 42 for the definition of the Conv(,) operator.

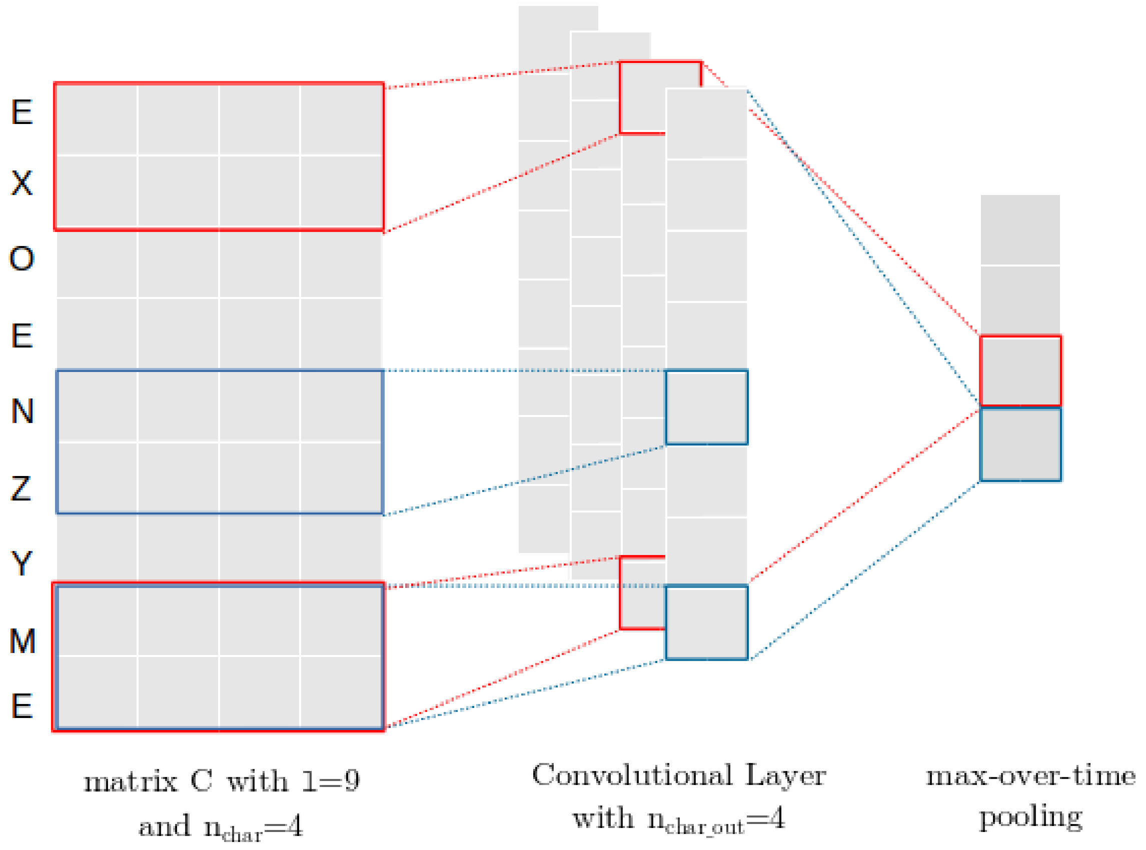

E
X
O
E
N
Z
Y
M
E
matrix C with l=9 and n_{char}=4
Convolutional Layer with n_{char_out}=4
max-over-time pooling

Figure 10 – Schematic of max-over-time pooling operation applied over character embeddings of the word "Exoenzyme". To the right is matrix C with the character-level embeddings of each word. The output of the convolution C_{Conv} is showed in the second step with $n_{char\_out} = 4$ and finally, the third stage is the result of the max-over-time pooling. Note that in the picture, only the resulting vectors are presented and the kernel H is omitted for clarity.

word and character-level embeddings. Note, that ω is the embedding vector of a single word and when referring to a set of words in a question or passage, Ω will be used.

5.2 Position Encoding

Assuming that a piece of text is represented in a matrix Ω formed by word embeddings, position encoding consists in summing a vector e_i to every word ω_i. The vector e_i is supposed to encode the i^{th} of the sequence of words represented in Ω. Note that e_i has the same dimensionality of the word embedding, d_{model}.

In (2) two alternatives for this encoder are presented: its values could be simply trained from a randomly initialized embedding matrix, or they could be generated through a deterministic combination of sines and cosines of different frequencies, which we named trigonometric encoder.

To study the properties of these encoders concerning the multiplicative attention, we defined the operation in (5.8) to which we will refer as positional attention.

$$att_{pos}i, j = e_i^T \; e_j \tag{5.8}$$

We define the positional attention by the dot product because all attentions in our model are multiplicative[2]. Note that (5.8) guarantees the commutative property, independent of the choice of the positional encoder.

Property 5.1 (Commutativity) *Positional attentions from the ith to the jth position and from the jth to the ith position are equal.*

$$att_{i,j} = att_{j,i} \; \forall i, j \in \mathbb{N} \tag{5.9}$$

In the following section, we are going to analyze other properties of the trigonometric encoder (2) that were considered relevant to the performance of FABIR.

5.2.1 Trigonometric Encoder

Given an embedding vector of even size d_{model}, its trigonometric position encoder is given by:

$$e_i = \begin{bmatrix} sin(i * f_1) \\ cos(i * f_1) \\ ... \\ sin(i * f_{d_{model}/2}) \\ cos(i * f_{d_{model}/2}) \end{bmatrix} \tag{5.10}$$

where f_k are scalars, which define the frequency of each sine and cosine.

In spite of its simplicity, this encoder shows some interesting properties regarding the dot product that we discuss in this section. Although positional attention does not take into account the specific word that occupies a given position, it hints at how the model treats words that are nearby or far apart in a sentence, which helps us to understand the model's overall behavior.

The following three properties are consequences of the choice of a trigonometric encoder.

Property 5.2 (Linear Dependence) *The positional encoder vector shows the same linear dependence on the previous encoder vector, regardless of its position i.*

$$e_{i+1} = R * e_i \; \forall i \in \mathbb{N} \tag{5.11}$$

where $R \in \mathbb{R}^{d_{model} \times d_{model}}$ is a square matrix of same dimensionality as e_i.

[2] See p. 43 for a discussion of the different types of attention mechanism.

Property 5.3 (Symmetry) *The attention between positions separated by the same distance is the same regardless of orientation.*

$$att_{i,i+k} = att_{i,i-k}\ \forall i, k \in \mathbb{N} \tag{5.12}$$

Property 5.4 (High Identity) *The attention of a position with itself is equal to $d_{model}/2$, which is the greatest possible value.*

$$att_{i,i} = d_{model}/2 \geq att_{i,j}\ \forall i, j \in \mathbb{Z} \tag{5.13}$$

Note that all previous properties are independent of the frequencies f_k, which gives us some freedom in the design of f_k. In Google's work (2), they were defined as a geometric progression of size $d_{model}/2$ between frequencies 10^{-4} and 1 with a model dimension $d_{model} = 512$. As we are using a lower dimension in FABIR, we ran an analysis to verify its effects.

Figure 11 shows $att_{pos}(1, x)$ for consecutive positions (1^{st} through 400^{th}) for two different model dimensions d_{model}, considering the same frequency interval $[1e^{-4}, 1]$. The limit of 400 was chosen, because 99.5% of our passages are shorter than that.

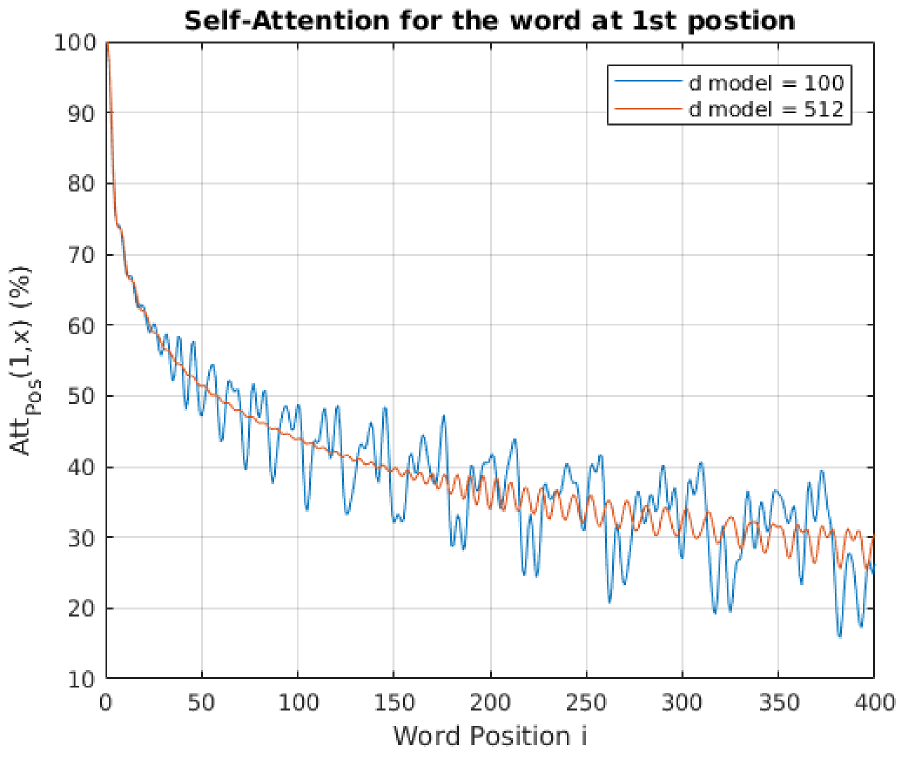


Figure 11 – Positional attention comparison for different model dimensions. Attention values (y axis) were normalized to allow for a direct comparison.

While for $d_{model} = 512$ the curve is almost monotonic with oscillations of small amplitude, for $d_{model} = 100$ the attention values oscillate considerably more, which might deteriorate the performance of the model. It is important to highlight that this is only observed in distant positions. Indeed, the curves are almost identical for low values in the x-axis.

Figure 12 shows the same plot, but considering a frequency interval for $d_{model} = 100$ of $[3e^{-3}, 1e^{-1}]$. Although oscillations in further positions got considerably smaller, this other frequency interval had a negative impact on the encoding of the position of words nearby. For positions close to the reference word, frequencies $[1e^{-4}, 1]$ resulted in a steeper curve than that observed for frequencies $[3e^{-3}, 1e^{-1}]$. This smaller variation for lower frequencies might prevent the model from distinguishing between close words, which could also undermine performance.

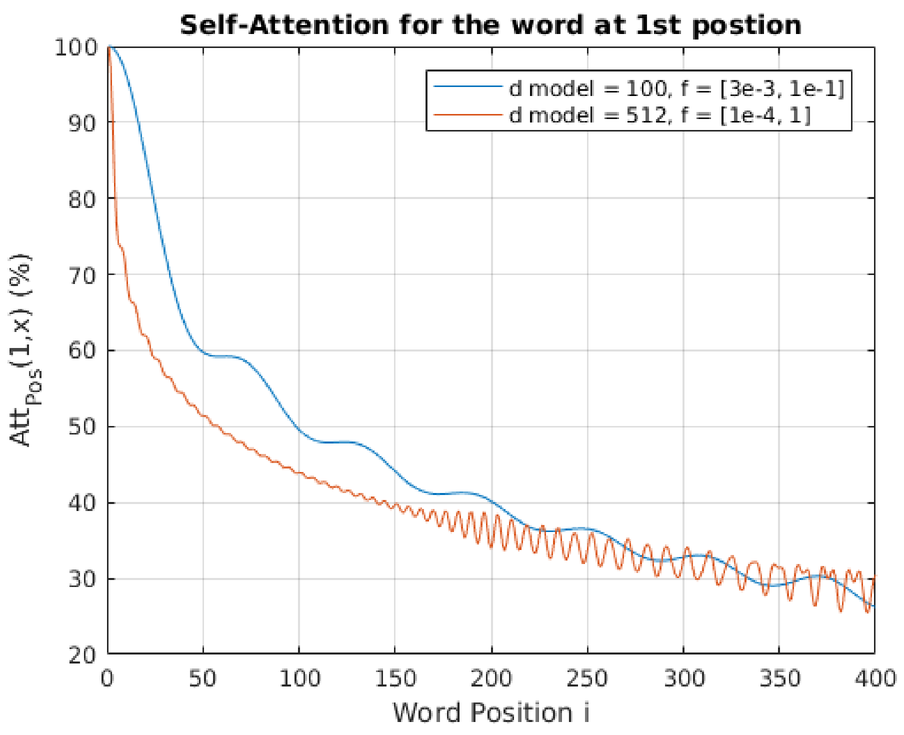


Figure 12 – Positional attention comparison for different frequency boundaries. Attention values (y axis) were normalized to allow for a direct comparison.

5.3 Processing Stage

For a clear explanation, the processing stage is going to be split into two organizational levels. The first one is the level of *sublayers*, which is composed of the basic operation blocks of self-attention, cross-attention, feedforward and normalization. The second one is the assembly of these blocks to compose *layers*, which are stacked one on top of the other to build the complete processing stage.

Regarding the sublayers, both self and cross attention operations follow a similar structure to that introduced in (2) and, therefore, were already covered in Section 4.6. In this chapter, we present only our additions to this structure, namely the convolutional attention described in Section 5.3.1. Here we also describe the feedforward and normalization operations in Sections 5.3.2 and 5.3.3. Finally, we discuss the assembly of these sublayers to compose a layer in Section 5.3.4 and show how these layers can be combined to build the complete processing stage in Section 5.3.5.

5.3.1 Convolutional Attention Layers

Self and cross attentions were implemented based on Transformer's attention mechanism (2) (see p. 53). However, instead of computing the softmax directly over the dot product of query U by key K, we added a convolution along the height U_{len} and width K_{len} of UK^T. To keep dimensions constant, we treat each head as a channel and maintain the number of filters in the convolution equal to the number of heads n_{heads}.

The convolutional attention is described in (5.14)[3], where the convolutional kernel is $H \in \mathbb{R}^{H_{height} \times H_{width} \times n_{heads} \times n_{heads}}$. The input to this convolution is zero-padded to keep the dimensions constant, so we have $UK^T \in \mathbb{R}^{U_{len}+H_{height}-1 \times K_{len}+H_{width}-1 \times n_{heads}}$.

$$(UK^T)_i = UW_{U,i}(KW_{K,i})^T \tag{5.14a}$$

$$(UK^T)_{i,padded} = pad((UK^T)_i) \tag{5.14b}$$

$$(UK^T)_{conv} = Conv((UK^T), H) \tag{5.14c}$$

$$att_{head,i} = softmax((UK^T)_{conv,i})VW_{V,i} \tag{5.14d}$$

$$att_{conv}(U,K,V) = concatenate(att_{head,1}, ..., att_{head,n_{heads}})W_O \tag{5.14e}$$

where $W_{U,i}, W_{K,i}, W_{V,i} \in \mathbb{R}^{d_{model} \times d_{head}}$ and $W_O \in \mathbb{R}^{d_{head}*n_{heads} \times d_{model}}$ are weight matrices, $H \in \mathbb{R}^{H_{height} \times H_{width} \times n_{heads} \times n_{heads}}$ is the convolution kernel, $(UK^T)_i \in \mathbb{R}^{U_{len} \times K_{len}}$ is the query-key product in the i^{th} head, $(UK^T)_{conv} \in \mathbb{R}^{U_{len} \times K_{len} \times n_{heads}}$ is the convolution output, and $(UK^T)_{conv,i} \in \mathbb{R}^{U_{len} \times K_{len}}$ is its i^{th} channel.

The intuition behind this convolutional attention is that the association between two words should include their context. The attention mechanism introduced in the Transformer (2) is one-to-one in the sense that $att_{i,j}$ only considers the relationship between words i and j. We propose the addition of a convolution operation to extend this relationship to

[3] See p. 42 for the definition of the Conv(,) operator.

many-to-many and better model the context. Indeed, the span of kernel H includes words around both i and j in the computation of $att_{i,j}$.

Additionally, in the case of cross-attention, we consider two possible directions for the softmax in (5.14d): row-wise, in which the values attributed to each word in a question sum up to one, and column-wise in which the passage values sum up to one.

5.3.2 Feedforward

The feedforward sublayer is simply composed of a neural network with a single hidden layer. It is applied to each vector x_i and, therefore, does not take into account the relationship between words and positions inside a set X. It is perhaps more illustrative to say that the feedforward operation processes each word individually. Following the architecture suggested by Google (2), the feedforward sublayer is implemented in (5.15) with a two-layers-neural network with ReLU activation in the hidden layer.

$$x_{i,out} = ReLU\left(x_i\, W_1 + b_1\right) W_2 + b_2. \tag{5.15}$$

where $W_1 \in \mathbb{R}^{d_{model} \times d_{hidden}}$, $W_2 \in \mathbb{R}^{d_{hidden} \times d_{model}}$, $b_1 \in \mathbb{R}^{1 \times d_{hidden}}$ and $b_2 \in \mathbb{R}^{1 \times d_{model}}$ are all trainable parameters, and d_{hidden} is the dimension of the hidden layer in (5.15).

5.3.3 Normalization

The main goal of layer normalization is to accelerate training as shown in (45, 46). Like the feedforward operation, normalization is applied to a single vector x_i, and hence, it does not take into account its position or any surrounding words, as in cross or self-attention sublayers. Layer normalization (46) was preferred over batch normalization (45) because the former is straightforward to implement in our model, while the latter is not. That happens because in SQuAD the input is composed of queries and passages, which are expected to vary in length from sample to sample.

This sublayer normalizes the embedding of each word so that its variance and mean are reduced to 1 and 0, respectively. Given a set of vectors X with embedding dimension d_{model}, we define the normalization operation as follows:

$$\mu_i = \frac{1}{d_{model}} \sum_{j=1}^{d_{model}} X_{i,j} \tag{5.16a}$$

$$var_i = \frac{1}{d_{model} - 1} \sum_{j=1}^{d_{model}} \left(X_{ij} - \mu_i\right)^2 \tag{5.16b}$$

$$\overline{X}_{ij} = \left(\frac{g_j}{var_i}\left(X_{i,j} - \mu_i\right) + b_j\right) \tag{5.16c}$$

$$\overline{X} = norm(X) \tag{5.16d}$$

where g_j and b_j are respectively gain and bias for each feature j that should be trained together with the other parameters of the model.

5.3.4 Sublayers Assembly

The assembly of sublayers to compose layers is discussed in this section. The designed layer follows the same structure found in (2), which is depicted in Figure 13 and generally defined in (5.17). For clarity, we will refer to the operation performed by each layer as $\mathcal{L}$.

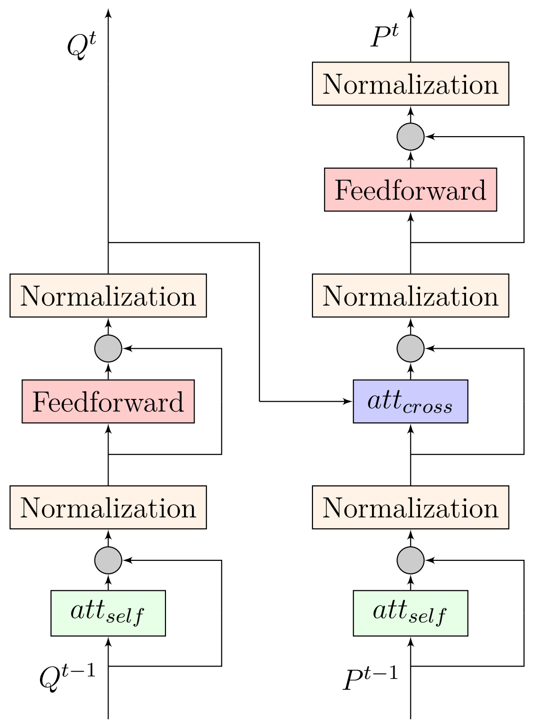


Figure 13 – Block diagram representation of a FABIR layer $\mathcal{L}_{Q\to P}$. Q^{t-1} and P^{t-1} represent the output processed query and passage from previous processing layer t-1, respectively.

$$Q^t, P^t = \mathcal{L}_{Q\to P}\left(Q^{t-1}, P^{t-1}\right) \tag{5.17}$$

Notice that the processing of query Q^{t-1} is independent of the passage P^{t-1} and hence we denominate the corresponding layer a *query-over-passage* attention layer, which will be represented by the symbol $\mathcal{L}_{Q\to P}$. A layer with the same structure, but with attention in the opposite sense will be referred to as a *passage-over-query* attention layer, which can be defined by simply interchanging the query and passages inputs. Likewise, the latter is represented by the symbol $\mathcal{L}_{Q\leftarrow P}$.

5.3.5 Layers Assembly

Finally, layers $\mathcal{L}_{Q\to P}$ and $\mathcal{L}_{Q\leftarrow P}$ can be assembled in order to compose the complete processing stage. Two architectures were considered and they are shown in Figure 14 for a three layers configuration. The former model is based on (2) and was named "Repeated Layers", because it applies the same layer repeatedly. The latter is called "Switching Layers" model, because while odd layers are $\mathcal{L}_{Q\rightarrow P}$, even layers are $\mathcal{L}_{Q\leftarrow P}$. Note that the first layer is numbered one and therefore is odd. "Switching Layers" model was designed based on the fact that state-of-the-art models in SQuAD used attention from passage over question, in addition to attention from question over passage (1, 16, 15).

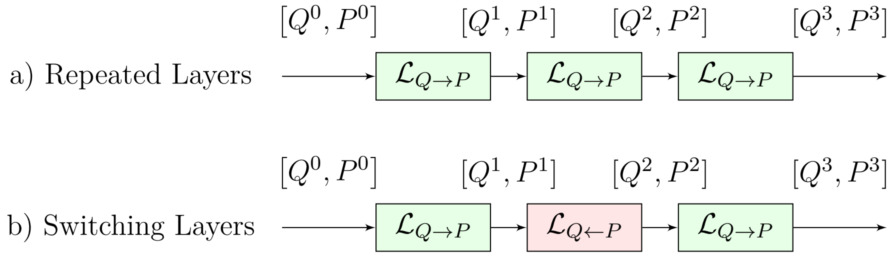


Figure 14 – Block diagram representation of "Repeated Layers" and "Switching Layers" models in a three layers configuration. Q^t and P^t represent the output query and passage from the tth layer.

In Figure 14, Q^t and P^t are the outputs of the tth layer. Q^0 and P^0 are the sum of the embeddings Q and P with their respective positional encoders e_Q and e_P.

5.4 Layer Reduction

This section describes the method that was used in the present work to reduce the embedding size dimension from d_{input} to d_{model}. The former is the dimension of the full word embedding ω generated by the concatenation of a GloVe vector (31) and a character-level embedding [4]. The latter is the reduced dimension that will be used in subsequent layers to process the passage P and the question Q.

In theory, nothing prevents both sizes from being equal, and it would be simpler to feed the inputs directly to the rest of the model. Nevertheless, for inputs with high dimensionality, an equally large processing layer size might not be beneficial for the model as a whole. Increasing the model only to fit the input size might not only be a waste of computational power and memory, but it might also complicate training, as larger models are expected to facilitate overfitting. Therefore, the introduction of a layer capable of reducing the embedding size from d_{input} to d_{model} has proved useful in developing our question-answering model.

A straightforward method to reduce the input embedding size is to multiply it by a matrix with the required dimensions, as shown in (5.18).

$$\omega_{model} = W_{Reduction}\omega_{input} \tag{5.18}$$

[4] See Equation (5.7) in p. 60 for details in the processing of embeddings with Highway Networks.

where $\omega_{model} \in \mathbb{R}^{1\times d_{model}}$, $\omega_{input} \in \mathbb{R}^{1\times d_{input}}$ are the embedding vectors, and $W_{Reduction} \in \mathbb{R}^{d_{model}\times d_{input}}$ is a weight matrix, to which we will refer to as "Reduction Matrix" from now on.

Although matrix reduction is quite simple, it discards information before any processing, and hence, it might prevent the network from using some relevant data, which might limit the performance of the model. To incorporate that information before discarding it, we could add a large processing layer followed by a matrix reduction, but our experiments have shown that this approach does not yield positive results for our model. Our interpretation of that behavior is that the position encoding is somehow dissolved in the matrix reduction process. That would happen because the reduced encoder may not maintain the properties that were described in section 5.2, such as linear dependence and symmetry.

A possible solution is to apply the reduction to the embeddings only and to use an encoding of size d_{model}. Therefore, we suggest a *decoupled attention*, which is described in Figure 15. There, the self-attention is applied to embedding and encoding separately, but both are computed with the full embedding $\Omega \in \mathbb{R}^{\Omega_{len}\times d_{input}}$. Note that the defined attention allows us to use V with a different embedding size than U and K [5].

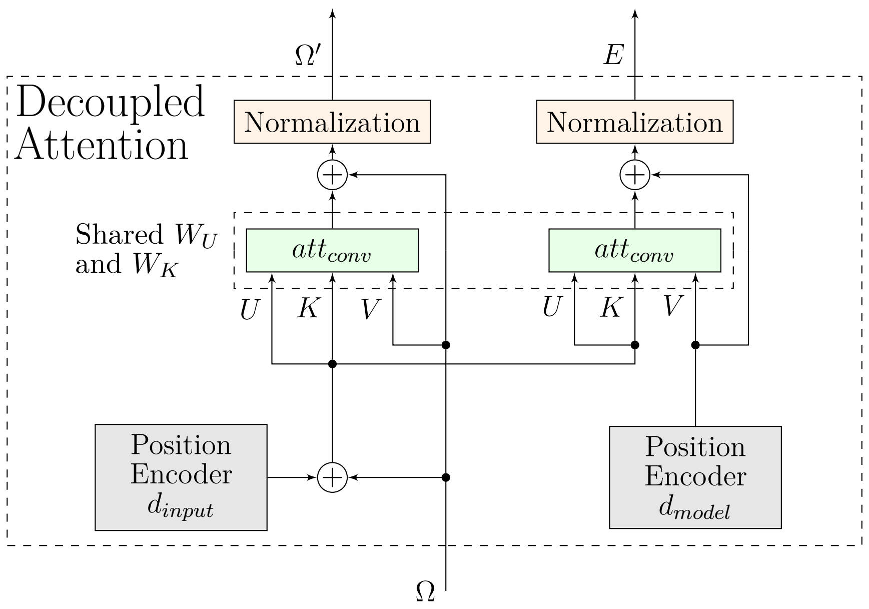


Figure 15 – Decoupled Attention. In contrast to previous attention mechanisms, this structure computes the embedding matrix $\Omega' \in \mathbb{R}^{\Omega_{len}\times d_{input}}$ and the encoder matrix $E \in \mathbb{R}^{\Omega_{len}\times d_{model}}$ separately.

After applying Decoupled Attention, we add a full processing layer for the embedding Ω' with size d_{input}, which implements cross attention and feedforward. That is equivalent to a regular $\mathcal{L}_{Q\to P}$ layer, though in this case the encoding E is left untouched. Note that the

[5] See p. 65 for a discussion of attention mechanisms.

cross attention will not use position encoders, but only the processed embedding Ω', which is the output from the Decoupled Attention sublayer. That is not expected to deteriorate the results because positions in the question Q and the passage P are not correlated. Finally, a Matrix Reduction is used to reduce the processed Ω' size and add the encoder matrix E. Figure 16 describes this whole process, which was named *Layer Reduction.*

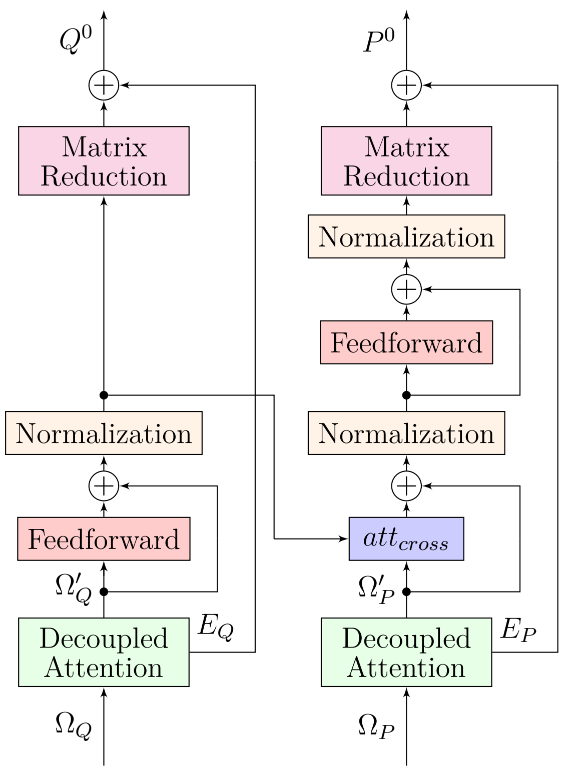


Figure 16 – Reduction Layer. This layer has inputs Ω_Q and Ω_P with embedding size d_{input} and it outputs Q^0 and P^0 with embedding dimension d_{model}. Both Q_0 and P_0 are used as inputs for the subsequent processing layers.

5.5 Answer Selection

Answer selection uses the output of the layers in the processing stage to compute the start and end indices $\hat{y}_1$ and $\hat{y}_2$ of the answer. For this, two decisions must be taken:

– **Selection Layers**: In order to compute both $\hat{y}_1$ and $\hat{y}_2$, a layer output must be chosen, which might be different for each of them. For instance, one possible configuration is to calculate $\hat{y}_1$ using Q^2 and P^2, and compute $\hat{y}_2$ with Q^3 and P^3. That means that y_1 and y_2 are selected using the second and third layers outputs, respectively.

– **Selection Functions**: The function that is going to be applied over the selection layer to compute $\hat{y}_1$ and $\hat{y}_2$ also must be chosen. It is important to highlight that they might be different, but for the sake of simplicity they are considered to be the same in our analysis. The selection functions considered in this project are detailed in the next sections. Additionally, the presented functions compute $\hat{\pi}^{y_1}$ and $\hat{\pi}^{y_2}$, which are a P_{len}-probability simplex and represent the probability of each word to be the start and end index, respectively. Finally, the predicted $\hat{y}_1$ and $\hat{y}_2$ must be computed as function of $\hat{\pi}^{y_1}$ and $\hat{\pi}^{y_2}$ by the Indices Selector, which is described in Section 5.5.3.

5.5.1 Linear

This selector was chosen based on (2) and can be considered the simplest alternative because it applies a softmax directly over the last layer P^t, as shown (5.19). It is also the least computationally expensive selector and hence, might be especially interesting for applications with limited processing power.

$$\hat{\pi}^y = softmax(P^t\ v) \tag{5.19}$$

where $\hat{\pi}^y \in \mathbb{R}^{P_{len}}$ is the discrete probability distribution over all indices and $v \in \mathbb{R}^{d_{model} \times 1}$ is a weight vector that must be trained.

5.5.2 n-layer Convolution

This function applies a sequence of n convolutions in the last layer output P^t. To compute the probability distribution $\hat{\pi}^y$, the last convolution output must be a single row of P_{len}, in which softmax can be applied. This selector is described in (5.20)[6].

$$\hat{\pi}^y = softmax(Conv(Conv(...Conv(P^t, H_1)..., H_{n-1}), H_n) \tag{5.20}$$

where $H_i \in \mathbb{R}^{1 \times H_{width,i} \times d_{filter,i-1} \times d_{filter,i}}$ are the kernels that are going to be trained, and $d_{filter,0}$ and $d_{filter,n}$ are defined as d_{model} and 1, respectively.

In comparison to the linear selector, this method takes into account words that are close to each other and hence it is expected to improve inference. That, of course, has the drawback of a larger processing cost and longer training times. The window size of words that are considered in the computation of $\hat{\pi}^y_i$ is shown in (5.21).

$$WindowSize = \sum_{i=1}^{n} H_{width,i} - 1 \tag{5.21}$$

where $H_{width,i}$ is the kernel size of the i^{th} convolution. Note that this selector with a single convolution with kernel size 1 is identical to Linear Selector.

6 See p. 42 for the definition of the Conv(,) operator.

5.5.3 Indices Selector

This functions outputs $\hat{y}_1$ and $\hat{y}_2$ as function of $\hat{\pi}^{y_1}$ and $\hat{\pi}^{y_2}$, which were computed in previous Sections. The indices are chosen following optimization in (5.22)

$$\begin{aligned} \underset{i,j}{\text{maximize}} \quad & \hat{\pi}_i^{y_1} * \hat{\pi}_j^{y_2} \\ \text{subject to} \quad & i \leq j < i + l_{answer,max} \end{aligned} \tag{5.22}$$

where $l_{answer,max}$ represents the maximum allowed answer length. This superior limit is imposed, in order to avoid long answers, since short answers are more frequent.

5.6 Training

5.6.1 Loss Function

A differentiable loss function must be defined, in order to use one of the gradient descent based algorithms for optimizing the model parameters. Following the state-of-the-art trend (1, 14, 15, 16), we have chosen the negative log likelihood function, which is shown in (5.23).

$$J = -\sum_{i=0}^{P_{len}-1} (\pi_i^{y_1} log(\hat{\pi}_i^{y_1}) + \pi_i^{y_2} log(\hat{\pi}_i^{y_2})) \tag{5.23}$$

where $\pi_i^{y_1}$ and $\pi_i^{y_2}$ represent the probability of the i^{th} index to be the start and the end indices of the answer, respectively. Usually, if there is only one possible answer, $\pi_i^{y_1}$ is set to 1, if $i = y_1$ and 0 otherwise. An alternative is to use label smoothing, in which $\pi_{y_1}^{y_1} = 1 - \delta$ and the δ probability is somehow distributed in the other indices. Label smoothing is essentially a regularization technique that has been found to help to avoid overfitting (47).

5.6.2 Optimizer

The Adam optimizer[7] (38) was the one that produced the best results. The hyper-parameters for this optimizer were set in accordance to the suggestions made by (2) with $\beta_1 = 0.9$ and $\beta_2 = 0.98$. Also, the learning rate was set to produce a set of initial warm-up steps during which its value increases in contrast to the typical annealing rates.

$$learning\_rate = d_{model}^{-0.5}\, min(t^{-0.5}, t\, warmup\_steps^{-1.5}) \tag{5.24}$$

where t is the current step and *warmup_steps* is the number of steps during which the learning rate is increased. In our model, *warmup_steps* was set to 4000.

[7] See p. 46 for a discussion of different gradient descent algorithms.

5.6.3 Initialization Techniques

Deep learning models are sensitive to the initial values of their set of weights. In fact, inadequate initialization methods were one of the hindrances to the effective training of neural network-based models for many decades (8). The solution commonly called Xavier initialization consists in maintaining the same variance in every layer so that the information flow can be efficiently propagated (48). It can be shown that this property can be achieved for the initial steps if the weights of each layer are initialized with the following variance.

$$Var[W^i] = \frac{2}{n_i + n_{i+1}} \tag{5.25}$$

where n_i and n_{i+1} are the dimensions of the input and output of layer W^i. That method was used in all weights in the model, except in the initialization of trainable word and character embeddings. In that case, the embedding matrix is too large and applying the Xavier initialization would result in a very low variance. Therefore, we opted to keep a constant variance of one in the embeddings, an approach that was also adopted in (1).

5.6.4 Regularization Techniques

Regularization techniques are useful to avoid overfitting. For this purpose, we used Dropout (49), which randomly substitutes values in different parts of the model by zeros during training. In FABIR, we used five different dropouts, which are listed below.

- **Input Dropout** (2): It is applied once to every word embedding Ω and encoder position vectors E before the first layer $\mathcal{L}$.
- **Char-embedding Dropout**: This dropout is applied to every character embedding, before the convolution. Note that after the concatenation of ω_c with ω_w, the input dropout is applied once more.
- **Sublayer Dropout** (2): It is applied directly after every feedforward, self-attention and cross-attention sublayers. Note that this dropout is applied before summing the output of the sublayer with its input, and hence, before the normalization sublayers. Note also that this dropout is not applied after normalization sublayers.
- **Attention Dropout** (2): In addition to the sublayer dropout, another dropout was added in attention sublayers directly after the computation of $softmax(UK^T)$ in every head.
- **Selector Dropout**: It is applied in the input of every selector layers.

In addition to these dropouts, we defined a **Dropout Amplification** β for the Reduction Layer, which increases the dropout in this layer according to the values defined for the other layers. This amplification was motivated by the fact that the reduction layer is larger than processing layers. The computation of the equivalent dropouts in the Reduction Layer is described in (5.26).

$$Dropout_{ReductionLayer} = Dropout_{ProcessingLayer}^{\beta} \tag{5.26}$$

6 Model Validation

In this chapter, we present the experimental procedures used to validate our model. FABIR was developed using the Tensorflow python API (18) and the code can be found at `https://github.com/AlCorreia/FAB` for replicability. The code was tested with Tensorflow 1.3 and might not run on previous versions. We run all experiments on a single GPU NVidia Titan X with 12GB of RAM.

In the following sections, we first discuss how the data in SQuAD was used to test and validate FABIR. Then, we discuss the different FABIR structures that were presented in chapter 5 and evaluate their influence in light of a series of experimental results. Finally, we compare FABIR against a state-of-the-art RNN-based model (1) to assess the advantages and disadvantages of this novel mechanism against RNN models.

6.1 Data splits

Before running the tests, we split the SQuAD dataset into training (80%), development (10%) and test (10%) sets, which are described below. That is a standard procedure in the machine learning literature that helps to validate the model.

Training dataset: This dataset is used exclusively to optimize the parameters of the model, θ.

Development dataset: This dataset is also used during training, but only to check the generalization capability of the model and not to update θ, as done with the training set. EM and $F1$ scores in this dataset are monitored to avoid overfitting, i.e., to avoid having a model that can answer correctly only questions on which it has been trained.

Test dataset: This is the dataset used for testing the final model after the complete optimization of its parameters. The leaderboard scores in $SQuAD$ are computed using this dataset, which was not made public to avoid frauds. Therefore, in order to use it, the QA model must be submitted to the SQuAD team that runs the tests and computes the scores in this dataset.

6.2 Batch Generation

The SQuAD dataset is composed of question-answer pairs of different lengths, but to keep the code efficient, we constrain all matrices in the model to have the same length for a single batch. That is achieved by padding the shortest sentences with zeros so that all examples in a batch appear equally protracted to the model. However, in that case, the computational cost of running a batch is always determined by the longest sentence.

Therefore, to speed up training, we divided the question-answer pairs by length so that only examples of similar size are run together in a batch. That reduces processing costs as the operations can be applied on considerably smaller matrices. More specifically, all examples were classified into 30 different groups according to their length and batches were composed of randomly selected examples within a same group, which was also chosen at random. That technique did not have any impact on the model's overall performance regarding EM and F1, but it did reduce training times by more than 2.5 times.

6.3 FABIR - Model Description

After running multiple tests to fine tune the hyperparameters, we arrived at our best performing model, which achieved scores of 77.6% and 67.6% in F1 and EM, respectively. This best model is the one to which we refer as FABIR herein. Its overall structure has already been introduced in Chapter 5, but its hyperparameters were defined as variables to favor a more general discussion of the properties of the model. In Table 5, we define the exact values of each of these hyperparameters that were used to achieve our best performance.

6.4 FABIR Structures Validation

The relevance of each structure in FABIR was estimated by first building and training variants of FABIR in a standardized way and subsequently, by evaluating their EM and F1 scores in the development dataset. That is important to validate our design choices and weigh the gains in performance against the computational cost contributed by each part of the model. For the sake of simplicity, we are going to use the model in Table 5 as a reference and only change it part by part to check its effects. We are not going to change more than one structural part at a time because the number of possible combinations would rise quickly.

Table 6 describes all the tests that were run to assess the relevance and the effects of some of the parameters defined in Table 5. The variants are compared concerning their performance scores (EM and F1) and their training time, which is shown as TT. All models were trained for 18 epochs with a batch size of 75 samples in an NVidia Titan X after being initialized without pre-trained variables, i.e., all variables were randomly initialized.

Note that we achieved our best scores in 54 epochs, though we did not present the following analysis with that number of epochs because it would take too long to train each

Table 5 – FABIR Model Description.

Stage	**Features**	**Description**
Pre-Process	Tokenizer	nltk
	Word-Embedding	Glove 6B 100
	Char-embedding	Convolution with max pooling Char embedding size: 8 Convolution output size: 100 Number of highway layers: 2
Position Encoding	Type	Trigonometric
	Frequencies	Geometric progression from 0.001 to 1.0.
Process	Embedding Reduction	Layer Reduction from layer size 200 to 100
	Number of Layers	Reduction Layer + 3
	Layer Size	100
	Layer Type	Repeated Layers
	Number of Heads	4
	Feed forward Hidden Size	200
	Cross-attention direction	Column-wise (passage direction)
	Attention Convolution	1×5
Selector	Structure	Convolution Number of Conv. Layers: 2 Kernel Size: 9 Hidden Layer Size: 32
	Max. Answer Length	15
Train	Algorithm	Adam: $\beta_1 = 0.9$, $\beta_2 = 0.98$ Learning Rate: 0.5 Warmup Steps: 4000
	Dropout	Input: 0.9 Sublayers: 0.9 Attention: 0.9 Selector: 0.8 Char embedding: 0.75 Layer Red. Amplification: 2.0

FABIR variant. However, we expect the variations shown in Table 6 to remain roughly the same for longer runs because, after 18 epochs, the model is almost fully trained and only marginal gains are observed. It is also worth noting that these experiments were run only once and given the stochastic nature of the gradient descent algorithms used to optimize the models, these values can vary by small percentages from run to run. In the following sections, we will discuss each one of the design choices and evaluate their contribution in light of the tests reported in Table 6.

Table 6 – Experiments description. TT stands for Training Time.

Exp. Name	**Description**	**F1 (%)**	**EM (%)**	**TT**
FABIR	No changes in Table 5.	75.6	65.1	2h14m
No Char Embedding	Model without char embedding.	72.9	62.0	1h48m
Matrix Reduction	Instead of layer reduction, matrix reduction was used to reduce layer dimension.	73.5	62.6	1h59m
Swiching Layer	A Reduction Layer followed by 3 Switching Layers	74.7	64.1	2h11
2 Layers	2 layers $\mathcal{L}_{Q \to X}$, in order to check if there are more layers than necessary.	75.0	64.1	1h55m
4 layers	4 layers $\mathcal{L}_{Q \to X}$, in order to to check if more layers would increase performance.	75.5	64.7	2h47m
2 Heads Attention	Number of heads is equal to 2.	73.8	63.2	2h12m
No Convolutional Attention	Model without convolution in attention, in order to check its relevance in the final performance.	73.2	62.6	1h49m
Row-wise softmax in cross attention	The direction of the softmax in cross-attention layer is applied in the row-wise direction instead of column-wise.	73.6	63.2	2h08m
Selector in different layers	Selector is applied in penultimate and last layers for y1 and y2, respectively.	74.3	63.7	2h09m
Selector with Kernel Size 1	The kernel size of the convolutions in selectors are reduced to one.	72.8	62.5	2h12m
Linear Selector	The selector is substituted by a simply linear selector.	74.0	62.8	2h10m

6.4.1 Char Embedding

Experiment “No Char Embedding” in Table 6 shows that character embedding was useful to deal with out-of-vocabulary words. Additionally, since our word embedding, GloVe 6B-100 (31), does not take into account capital letters, character embedding is also expected to attach such information to the final word representation, which helps to explain the improvement of almost 3% in the F1 score.

Nevertheless, that gain in performance comes at a high computational cost. The addition of character embeddings resulted in an increase of almost 25% in training time over the model with GloVe embeddings only.

6.4.2 Layer Reduction

In the experiment “Matrix Reduction”, it is noticeable that “Layer Reduction” outperformed “Matrix Reduction” by more than 2% in both F1 and EM scores. As described in Section 5.4, that difference could be explained by the early discard of information when matrix reduction is applied. It is important to highlight that in the “Matrix Reduction” experiment, a standard layer $L_{Q \to P}$ was added in place of the reduction layer, to have a fair comparison with the model described in Table 5 regarding the number of parameters.

6.4.3 Layer Type

The use of “Switching Layers” has shown a similar performance to “Repeated Layers” with a difference of less than 1% in the F1 score. Due to the simplicity of “Repeated Layers” and its slightly better performance, we kept it in the final FABIR model.

6.4.4 Number of Layers

Experiments “2 Layers” and “4 Layers” have shown that the most appropriate number of layers is three. The update from two to three layers seems to improve performance, as the EM score increased by 1%. Regarding the possibility of having more than three layers, it seems to increase training time and not add significant improvements, as the computed F1 changed only 0.1%.

6.4.5 Number of Heads

As suggested in (2), the multi-head attention has proven to be a cost-free option to improve model performance. The reduction of the number of heads decreased the F1 score by almost 2%, while the computation time remained virtually unchanged. It is important to highlight that, although the number of operations does not change significantly, we have observed that the amount of required RAM increases considerably with the number of heads.

6.4.6 Convolutional Attention

Experiment “No Convolutional Attention” has shown that the convolutional attention is computationally expensive, as it increases the training time by around 25min, which is comparable to the addition of another layer. Nonetheless, it improved F1 and EM scores by more than 2%, which speaks to its importance in the FABIR model. Due to the high computational cost, we think that alternatives that reduce the number of operations, such as depth-wise convolution (50), might offer considerable speed gains.

6.4.7 Softmax Direction in Cross Attention

Experiment “Row-wise softmax in cross attention” indicates that the column-wise direction is the most appropriate for the softmax in cross attention, as it improved by almost 2% both EM and F1. That goes against the implementation in Vaswani et al. (2), which applied it row-wisely in a machine translation task. This result could be explained by trying to understand why the row-wise softmax showed a weaker performance. In practice, this approach would compute a weighted average of the question words for every passage word, independently of other passage words. It thus treats every passage word similarly and, although each one of them is given a different score, they are all associated with a weighted sum of the question embeddings. Hence, this method fails to model all possible relationships between the two pieces of texts, e.g., it seems reasonable that not every word in the passage is related to the question. In contrast, when applied column-wise, the softmax would attribute greater weights to passage words that are more closely related to the respective question word, which seems appropriate for the SQuAD task, in which not all words in a passage are expected to be connected to the words in the question.

From the machine translation point-of-view, the row-wise approach could be explained by the fact that every word in the translated text is expected to be somehow represented by a weighted average of words in the original text.

6.4.8 Answer Selector

Experiments “Selector with Kernel Size 1” and “Linear Selector” were run to study the effect of the Answer Selector kernel size. Both selectors were worse than the convolution used in the original FABIR. That suggests that, despite the theoretical capabilities of self-attention mechanisms, convolutions are beneficial to model the relationship between nearby words in a sentence.

Experiment “Selector in different layers” was run to check the effects of adding a selector of y_1 and y_2 in different layers, in contrast to what is done in FABIR, where both indices are computed from the last layer. The intuition behind that change is that the information about the selected y_1 could be used by an extra layer to produce a better estimate of y_2. Relying on that idea, some of the state-of-the-art models discussed here do calculate y_1 and y_2 at different layers (1, 14). However, the experiments have shown that this approach is counterproductive in FABIR and computing both indices directly from the last layer, resulted in an F1 score 1% higher.

6.5 FABIR vs BiDAF

As stated in chapter 4, to the best of our knowledge, all models in the SQuAD ranking are RNN-based and, therefore, their processing speed is constrained by a large number of sequential operations. Because FABIR relies on a different type of neural network, we write this section to compare both architectures: fully attention-based and RNN-based.

To have a comprehensive comparison, we took a state-of-the-art model (1) developed in Tensorflow that had its code openly available in the web[1]. That way, we could run our experiments with both models in the same piece of hardware to have a fair comparison between them. Table 7 shows some preliminary results.

Table 7 – Comparison between FABIR and BiDAF (1) models. The BiDAF scores without parenthesis were achieved after training their model in 18,000 iterations of batch size 60 in our hardware. The values under parenthesis are their official scores in SQuAD ranking (19).

	FABIR	**BiDAF**
F1 (%)	77.6	77.0 (77.3)
EM (%)	67.6	67.3 (68.0)
Training Time	6h30m - 54 epochs	6h30m - 12 Epochs
Training Time/Epoch	7m15s	37m30s
# of Training Variables	1,385,198	2,695,851
Inference Time (full dev)	24s	135s

6.5.1 Training Time and Performance Score (EM and F1)

Regarding EM and F1 scores, FABIR and BiDAF showed similar performances. While BiDAF is slightly better in EM, FABIR marginally outperformed it concerning the F1 score. Although both models required similar training times to reach these scores, the time for training one epoch in FABIR is more than five times shorter, which could be useful for training this model in a larger dataset. It is hard to point out why FABIR needs many more epochs to achieve that score, but it is possible that a better set of hyperparameters or a different optimizer could speed up the learning process. Indeed, FABIR reaches an F1 score of 76% in the first 20 epochs and then takes other 30 to improve less than 2%, which indicates that the learning curve can be worked on.

6.5.2 Inference Time

With respect to inference time, FABIR was more than five times faster in processing the 10,570 question-passage pairs in the Development data set. This faster inference together with the similar F1 and EM scores gives room for using FABIR in large-scale applications, such as information extraction in large corpora. Indeed, when running applications such as search tools or user interfaces, the inference time is critical to tackling real-world problems.

[1] The code for the BiDAF model (1) is available at `https://github.com/allenai/bi-att-flow`

6.5.3 Number of Parameters

Concerning the number of training variables, FABIR has almost 50% less parameters than BiDAF, which incurs three major advantages:

Training time: A large part of the training time is consumed by the updates of each parameter via gradient descent. Hence, for a model with half the number of parameters, the time required for each step is expected to be lower, which was indeed observed as FABIR was four times faster than BiDAF in that respect.

Memory requirements: A model with fewer parameters requires less memory to be stored or transferred across platforms. It is interesting to applications that dispose of low computational power.

Generalization: It is a well-known fact that larger models are more likely to overfit, an observation which is often associated with the Occam's razor principle. Even though that is only an empirical observation, all things being equal, a model with fewer parameters is more likely to produce positive results on new data.

6.5.4 Question-Type Analysis

Figure 17 shows the F1 scores for BiDAF and FABIR concerning different performance measures when varying the characteristics of the question-passage pairs. Plot a) shows that shorter answers are easier for both models: while they reach more than 75% in the F1 score for answers that are shorter than four words, for answers longer than ten words, that score drops to 60.4% and 67.3% for FABIR and BiDAF, respectively. That could be explained not only by the higher complexity associated with longer answers but also by the fact that more than 79% of the question-passage pairs in SQuAD have answers that are shorter than five words. They are less exposed to questions that require long responses and hence, are not effectively trained to provide long answers.

Plot b) shows that question length seems not to be relevant in the performance of both models. The F1 scores for both models varied by less than 2.5% in the considered question length intervals.

Plot c) shows that both models depend strongly on the question type. Both had their best performance with "when" questions, which could be explained by the higher predictability of possible answers. They are usually composed of time-related words, such as months, years, seasons or weekdays, which are easier to distinguish from the rest of the text. Together with "when" questions, "how long" and "how many" also proved easier to respond, as they possess the same property of having a smaller universe of possible answers. In contrast to these, "how" and "why" questions resulted in considerably lower F1 and EM scores, as they can be answered by any sentence and hence, require a deeper

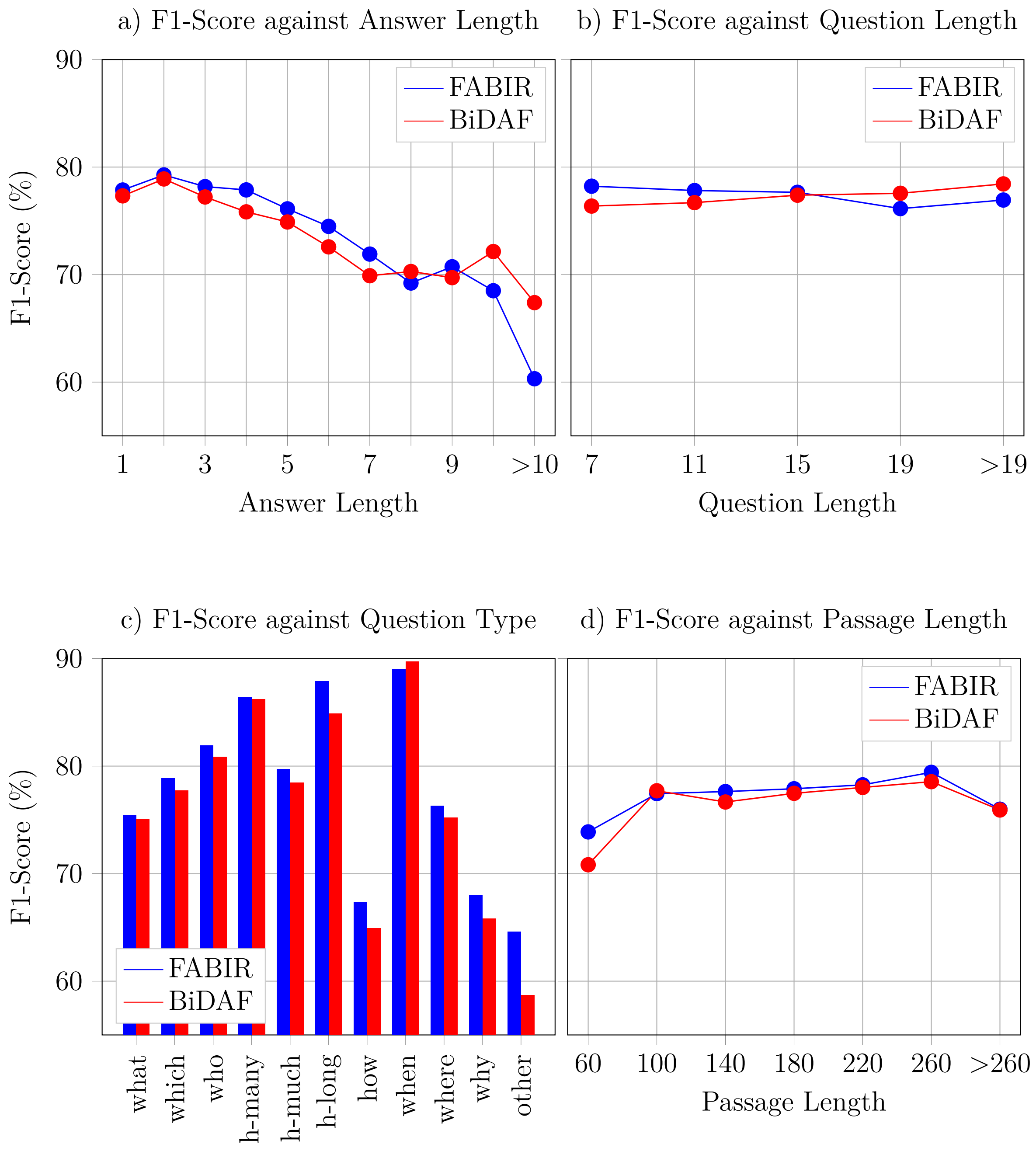


Figure 17 – F1-Score of FABIR and BiDAF with respect to a) Answer Length, b) Question Length, c) Question Type and d) Passage Length

understanding of the text. Note that "how" questions do not include "how many", "how much" or "how long" questions, which have more predictable answers.

"Other" questions include alternatives, such as "Name a type of..." or "Does it..." or even typos like "Hoe was ...". The first type is complicated to add in datasets like SQuAD because it might have multiple correct answers and require higher levels of abstraction. For instance, to respond to a question such as "Name an ingredient...", the model would need a deep understanding of the semantics of the word "ingredients" to identify "tomatoes" or "cheese" as possible answers. The second kind of "Other" question usually expects a "yes" or a "no" as an answer, but in SQuAD this is not possible because the answer must be a snippet from the passage. Hence, the model is supposed to provide the same or a similar sentence in the passage as an answer. It could be something in the lines of "Does Ronaldo play football?", which could be answered by a snippet like "Ronaldo plays football...". Finally, the third type of "Other" question originates from misspellings. In the example above, the letter "w" was replaced by an "e". There are other types of "Other" questions, but we do not list them here for they do not reveal any other interesting characteristics of the models.

Plot d) shows the performance of FABIR and BiDAF against the passage length. It is curious that shorter passages showed the worst performance for both models. It is hard to interpret that result as intuitively, we would expect brief passages to be easier to interpret. One possibility is that questions related to short paragraphs can be more complicated because the creators of the dataset had fewer options of simple questions, such as "when", "who", or "how many" and had to resort to more elaborate alternatives.

7 Final Considerations

7.1 Conclusion

Considering the final results and the properties of FABIR, we can assert that our model satisfied all the requirements outlined in Chapter 2. In particular, the most challenging of those, which defined the threshold for an acceptable F1-score (HLR 1.8), was achieved with success. The F1 and EM scores of 77.6% and 67.6%, respectively, place FABIR among other state-of-the-art models in the SQuAD leaderboard.

That is a very positive result, given that a fully attention-based model has some significant advantages over the more traditional RNNs. Indeed, FABIR has achieved similar performances while requiring fewer sequential operations, having fewer parameters and being faster both at training and testing times. Those characteristics certainly make FABIR an appealing alternative to real-world applications, which often dispose of limited computational power and are under tight time constraints to achieve the desired user experience. Moreover, in extending the Transformer (2) to QA systems, we have made some pivotal changes in the design of the architecture of fully attention-based models that can be considered valuable contributions to the fields of deep learning and NLP.

7.2 Future Work

Fully attention-based models are relatively new, and hence still constitutes a vast territory for further research. To that extent, we see three main areas where FABIR could be extended to achieve better performances or tackle other tasks.

7.2.1 New Attention Mechanisms

The performance achieved in the SQuAD dataset was primarily due to the introduction of new attention mechanisms, such as the convolutional-attention, and we believe that there is still plenty of room to explore new types of such mechanisms.

7.2.2 New QA datasets

Question-answering is a vibrant research domain, and new open-domain data sets are being developed. Given that state-of-the-art models are getting closer to human performance in the SQuAD, the research community is moving to more complex and challenging data sets. TriviaQA (51) is one those that has drawn some attention. It also assumes that the answer is contained in the passage, but it was designed to require reasoning over multiple sentences to infer the answer, which increases the complexity of the problem considerably. We would like to test our model in this new dataset to evaluate its performance one more challenging question-answer pairs.

7.2.3 Memory

Even though the performance levels are already impressive, FABIR is still a pattern recognition model with limited memory capabilities. That hinders practical applications of the model in two different ways. First, the model cannot model complex relationships between words that are not explicitly mentioned in the text. Even though word embeddings are capable of modeling semantic relationships to some extent, they are still limited. For instance, it might fail when questions bear on common sense knowledge, such as whether a tortoise could outrun a hare in a race. Second, FABIR cannot represent changes of state and would not be able to understand a narrative. Therefore, questions about a sequence of events or current status would be hard to answer.

Solving these two issues would likely require two different mechanisms. On the one hand, complex relationships would probably need large knowledge databases, such as NELL (52), and some logic reasoning (53). On the other hand, modeling changes of state would only be possible via some memory which the model could write and read during learning and inference times. The development of such models is still an open question, but some promising results have been shown in (54). That said, addressing these shortcomings of FABIR would surely extend its capabilities way beyond the SQuAD dataset and its closed format questions. Despite involving challenge research questions, we would like to explore extensions to FABIR within those lines.

Bibliography

1 SEO, M. et al. Bidirectional Attention Flow for Machine Comprehension. In: *2017 International Conference on Learning Representations*. Toulon, France: [s.n.], 2017.

2 VASWANI, A. et al. Attention Is All You Need. *arXiv preprint arXiv:1706.03762*, 2017.

3 RAJPURKAR, P. et al. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, 2016. p. 2383–2392.

4 KOLOMIYETS, O.; MOENS, M.-F. A Survey on Question Answering Technology from an Information Retrieval Perspective. *Information Sciences: an International Journal*, v. 181, n. 24, p. 5412–5434, 2011.

5 ZHANG, Y. et al. Question Answering over Knowledge Base with Neural Attention Combining Global Knowledge Information. *arXiv preprint arXiv:1606.00979*, 2016.

6 ALLEN, J. *Introduction to Natural Language Understanding*. 2nd. ed. London: Benjamin/Cummings Publishing Company, 1995.

7 RUSSELL, S. J.; NORVIG, P. *Artificial intelligence: a modern approach*. 3rd. ed. Upper Saddle River, New Jersey: Prentice Hall, 2010.

8 GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. Deep Learning. *Nature*, v. 521, n. 7553, p. 436–444, 2015.

9 JURAFSKY, D.; MARTIN, J. H. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 2nd. ed. Englewood Cliffs, New Jersey: Prentice Hall, 2009.

10 WOODS, W. a. Progress in natural language understanding: an application to lunar geology. In: *Proceedings of the National Computer Conference and Exposition on AFIPS '73*. New York: Association for Computing Machinery, 1973. p. 441–450.

11 LIPTON, Z. C.; BERKOWITZ, J.; ELKAN, C. A Critical Review of Recurrent Neural Networks for Sequence Learning. *arXiv preprint arXiv:1506.00019*, 2015.

12 BAHDANAU, D.; CHO, K.; BENGIO, Y. Neural Machine Translation By Jointly Learning To Align and Translate. *arXiv preprint arXiv:1409.0473*, 2014.

13 DENG, J. et al. ImageNet: A large-scale hierarchical image database. In: *Proceedings of 2009 Computer Vision and Pattern Recognition*. Miami: IEEE, 2009. p. 248–255.

14 Natural Language Computing Group, M. R. A. R-Net: Machine Reading Comprehension with Self-Matching Networks. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. Vancouver: Association for Computational Linguistics, 2017.

15 XIONG, C.; ZHONG, V.; SOCHER, R. Dynamic Coattention Networks For Question Answering. In: *2017 International Conference on Learning Representations*. Toulon, France: [s.n.], 2017.

16 CUI, Y. et al. Attention-over-Attention Neural Networks for Reading Comprehension. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver: Association for Computational Linguistics, 2017. p. 593–602.

17 HU, M.; PENG, Y.; QIU, X. Reinforced Mnemonic Reader for Machine Comprehension. *arXiv preprint arXiv:1705.02798*, 2017.

18 ABADI, M. et al. TensorFlow: Large-scale machine learning on heterogeneous systems. *arXiv preprint arXiv:1603.04467*, 2016.

19 RAJPURKAR, P. *The Stanford Question Answering Dataset*. Available in: <https://rajpurkar.github.io/SQuAD-explorer/>. Acessed in: 16-11-2017.

20 PASCANU, R.; MIKOLOV, T.; BENGIO, Y. On the difficulty of training recurrent neural networks. In: *Proceedings of The 30th International Conference on Machine Learning*. Atlanta: The International Machine Learning Society, 2013. v. 28, n. 3, p. 1310–1318.

21 HOCHREITER, S.; Urgen Schmidhuber, J. Long Short-Term Memory. *Neural Computation*, v. 9, n. 8, p. 1735–1780, 1997.

22 CHO, K. et al. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. Doha: Association for Computational Linguistics, 2014. p. 1724–1734.

23 GERS, F. A.; SCHMIDHUBER, J.; CUMMINS, F. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, v. 12, n. 10, p. 2451–2471, 2000.

24 LECUN, Y. et al. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, IEEE, v. 86, n. 11, p. 2278–2324, 1998.

25 SZEGEDY, C. et al. Going deeper with convolutions. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition*. Boston: IEEE, 2015.

26 SOCHER, R. et al. Reasoning With Neural Tensor Networks for Knowledge Base Completion. In: *Advances in Neural Information Processing Systems 26*. Lake Tahoe, Nevada: Curran Associates, Inc., 2013. p. 926–934.

27 WANG, S.; JIANG, J. Learning Natural Language Inference with LSTM. In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego: Association for Computational Linguistics, 2016. p. 1442–1451.

28 LUONG, M.-T.; PHAM, H.; MANNING, C. D. Effective Approaches to Attention-based Neural Machine Translation. In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon: [s.n.], 2015.

29 BRITZ, D. et al. Massive Exploration of Neural Machine Translation Architectures. *arXiv preprint arXiv:1703.03906*, 2017.

30 MIKOLOV, T. et al. Efficient Estimation of Word Representations in Vector Space. In: *Workshop Proceedings of the 2013 International Conference on Learning Representations*. Scottsdale, Arizona: [s.n.], 2013.

31 PENNINGTON, J.; SOCHER, R.; MANNING, C. D. GloVe: Global Vectors for Word Representation. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. Doha: Association for Computational Linguistics, 2014.

32 SANTOS, C. D.; ZADROZNY, B. Learning Character-level Representations for Part-of-Speech Tagging. In: *Proceedings of the 31st International Conference on Machine Learning*. Beijing: The International Machine Learning Society, 2014. v. 32, n. 2, p. 1818–1826.

33 WIETING, J. et al. Charagram: Embedding Words and Sentences via Character n-grams. In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, 2016. p. 1504–1515.

34 KIM, Y. Convolutional Neural Networks for Sentence Classification. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. Doha: The Association for Computational Linguistics, 2014. p. 1746–1751.

35 DUCHI, J.; HAZAN, E.; SINGER, Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, v. 12, p. 2121–2159, 2011.

36 ZEILER, M. D. Adadelta: An Adaptive Learning Rate Method. *arXiv preprint arXiv:1212.5701*, 2012.

37 RUDER, S. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2017.

38 KINGMA, D. P.; BA, J. Adam: A Method for Stochastic Optimization. In: *2015 International Conference on Learning Representations*. San Diego: [s.n.], 2015.

39 MANNING, C. D.; SCHÜTZE, H. *Foundations of statistical natural language processing*. Cambridge, Massachusetts: MIT Press, 1999.

40 MANNING, C. et al. The Stanford CoreNLP Natural Language Processing Toolkit. In: *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Baltimore, Maryland: Association for Computational Linguistics, 2014. p. 55–60.

41 BIRD, S.; KLEIN, E.; LOPER, E. *Natural Language Processing with Python*. 1st. ed. Sebastopol, California: O'Reilly, 2009.

42 KANARIS, I. et al. Words vs. Character n-grams for Anti-spam Filtering. *International Journal on Artificial Intelligence Tools*, v. 16, n. 6, p. 1047–1067, 2007.

43 SRIVASTAVA, R. K.; GREFF, K.; SCHMIDHUBER, J. Highway Networks. *arXiv preprint arXiv:1505.00387*, 2015.

44 KIM, Y. et al. Character-Aware Neural Language Models. In: *Thirtieth AAAI Conference on Artificial Intelligence*. Phoenix, Arizona: AAAI Press, 2016.

45 IOFFE, S.; SZEGEDY, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In: *Proceedings of the 32nd International Conference on Machine Learning*. Lille, France: The International Machine Learning Society, 2015. v. 37, p. 448–456.

46 BA, J. L.; KIROS, J. R.; HINTON, G. E. Layer Normalization. *arXiv preprint arXiv:1607.06450*, 2016.

47 SZEGEDY, C. et al. Rethinking the Inception Architecture for Computer Vision. In: *2016 Conference on Computer Vision and Pattern Recognition*. Las Vegas: IEEE, 2016. p. 2818–2826.

48 GLOROT, X.; BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. In: *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*. Sardinia: Proceedings of Machine Learning Research, 2010. v. 9, p. 249–256.

49 SRIVASTAVA, N. et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, v. 15, p. 1929–1958, 2014.

50 CHOLLET, F. Xception: Deep Learning with Separable Convolutions. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu: IEEE, 2017. p. 1251–1258.

51 JOSHI, M. et al. TriviaQA: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver: Association for Computational Linguistics, 2017. p. 1601–1611.

52 MITCHELL, T. et al. Never-Ending Learning. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. Austin, Texas: AAAI Press, 2015. p. 2302–2310.

53 De Raedt, L.; KIMMIG, A. Probabilistic (logic) programming concepts. *Machine Learning*, v. 100, n. 1, p. 5–47, 2015.

54 GRAVES, A.; WAYNE, G.; DANIHELKA, I. Neural Turing Machines. *arXiv preprint arXiv:1410.5401*, 2014.

A Evaluation Metrics

Many scores can be used to evaluate the performance of the model. Before going into the details of each type of score, we are going to present some important basic concepts. Given a sample, in which examples can be either "Positive" (P) or "Negative" (N), there are four alternatives for a prediction concerning that property, as shown in Table 8.

Table 8 – Classification possibilities for a single property.

		Property Value	
		N	P
Classification Value	N	True Negative (TN)	False Negative (FN)
	P	False Positive (FP)	True Positive (TP)

If the property was correctly predicted by the model, it could be either a true positive or a true negative. In contrast, if it was wrongly classified it could be a false negative or a false positive. The perfect algorithm would correctly predict every sample and therefore would make neither false negatives nor false positives.

In SQuAD, true positives and false positives are the characters predicted to be in the answer of the query. In order to improve an algorithm, the false positives should be instead classified as negative and therefore as not being part of the answer. A similar scenario occurs for true negatives and false negatives: the latter should be predicted as positive to improve the performance of the algorithm.

A.1 Precision

Precision is the ratio between the number of samples predicted as true positive divided by the number of total samples classified as positive, as shown in Equation A.1.

$$Pr = \frac{TP}{TP + FP} \tag{A.1}$$

Given a query and passage in SQuAD, this score would not take into account parts of the answer that were considered false negative. Therefore, an algorithm that does not select a character would get the same score as another algorithm that selects the right answer.

A.2 Recall

Recall is the number of true positives divided by the total of positives samples, as shown in Equation A.2.

$$Re = \frac{TP}{TP + FN} \tag{A.2}$$

In contrast to the precision score, it takes into account the number of false negatives, but not the false positives. Therefore, it would get the same score if the algorithm selects the complete passage or the right answer.

A.3 F1

F1-score is the harmonic mean between precision and recall, as shown in Equation A.3.

$$F1 = \frac{2}{\frac{1}{Pr} + \frac{1}{Re}} = 2\frac{Pr\ Re}{Pr + Re} \tag{A.3}$$

As explained in Sections A.1 and A.2, precision and recall are not adequate for the SQuAD question-answering task because a simple algorithm which returns zero characters or the complete passage would achieve good performances. In contrast, F1-score is stricter and more closely related to desirable answers. Hence, it was chosen to measure the performance of the models in the SQuAD ranking.

A.4 Exact Match

Exact Match (EM) is the strictest among the presented scores in this section. It is 1 if and only if the number of false negatives and positives are both zero and it is 0 otherwise.

$$EM = \begin{cases} if\ (FP == 0\ and\ FN == 0) & 1 \\ else & 0 \end{cases} \tag{A.4}$$