Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

IMPROVING DEEP QUESTION ANSWERING: THE ALBERT MODEL

Relatore:
Chiar.mo Prof.
FABIO TAMBURINI

Correlatore:
Chiar.mo Prof.
PHILIPP CIMIANO

Presentata da:
MATTEO DEL VECCHIO

Sessione III
Anno Accademico 2018/2019

"The computer is incredibly fast, accurate, and stupid. Man is incredibly slow, inaccurate, and brilliant. The marriage of the two is a force beyond calculation."
– Leo M. Cherne

To my grandparents
To my family
To my friends

Sommario

L'*Elaborazione del Linguaggio Naturale*, o *Natural Language Processing* (NLP), è un settore dell'Intelligenza Artificiale che si riferisce all'abilità dei computer di capire il linguaggio umano, spesso in forma scritta e principalmente usando applicazioni di Machine Learning e Deep Leaning per estrarre pattern. In modo più specifico, due sottoinsiemi principali sono *Natural Language Understanding* (NLU) e *Natural Language Generation* (NLG). Il primo permette alle macchine di capire un linguaggio, mentre il secondo permette loro di scrivere usando quel linguaggio. Tale settore è diventato di particolare interesse a causa dei progressi fatti negli ultimi anni, in seguito all'affermarsi di nuove tecnologie, quali nuove GPU più performanti, le Google Tensor Processing Units (TPUs) [11], nuovi algoritmi o altri migliorati.

L'analisi delle lingue è molto complessa, a causa delle loro differenze, astrazioni ed ambiguità; di conseguenza, la loro elaborazione è spesso molto onerosa, in termini di modellazione del problema e di risorse. Individuare tutte le frasi in un testo è qualcosa che può essere messo in pratica con poche righe di codice, ma capire se una data frase è sarcastica o meno? Ciò è qualcosa di difficile anche per gli umani stessi e quindi richiede meccanismi molto complessi per essere affrontato. Questo tipo di informazione, infatti, pone il problema di una rappresentazione che sia effettivamente significativa.

La maggior parte del lavoro di ricerca riguarda il trovare e capire tutte le caratteristiche di un testo, in modo tale da poter sviluppare dei modelli sofisticati che affrontino problemi quali la Traduzione Automatica, il Riassumere ed il Question Answering.

Questa tesi si concentrerà su uno dei modelli allo stato dell'arte recentemente reso pubblico e ne approfondirà le performance sul problema del Question Answering. Inoltre, verranno mostrate alcune idee per migliorare tale modello, dopo aver descritto i cambiamenti importanti che hanno permesso il training ed il fine-tuning di grandi modelli linguistici.

In particolare, questo lavoro è strutturato come di seguito:

- Il Capitolo 1 contiene un'introduzione sulle Reti Neurali, il loro uso e il ruolo che esse svolgono nell'ambito della rappresentazione del testo
- Il Capitolo 2 descrive il problema del Question Answering, alcune differenze tra i vecchi ed i nuovi modelli che lo affrontano ed alcuni dei dataset utilizzati
- Il Capitolo 3 introduce in dettaglio l'architettura del Transformer [41], il quale può essere considerato il modello alla base di tutti quelli allo stato dell'arte
- Il Capitolo 4 descrive ALBERT [21], il modello su cui si concentra questa tesi, e tutte le caratteristiche e decisioni che relative alla sua progettazione
- Il Capitolo 5 mostra le idee sviluppate per migliorare ALBERT [21], con un'attenzione principale al training ed ai risultati
- Le Appendici A e B mostrano alcune informazioni riguardo gli hyperparameters del modello e del codice.

Introduction

Natural Language Processing (NLP) is a field of Artificial Intelligence referring to the ability of computers to understand human speech and language, often in a written form, mainly by using Machine Learning and Deep Learning methods to extract patterns. More specifically, two of the principal subsets are *Natural Language Understanding* (NLU) and *Natural Language Generation* (NLG). The former empowers machines to understand a language while the latter empowers them to write using that language. It gained a particular interest because of advancements made in the last few years, due to the rise of new technologies, such as new high-performing GPUs, Google Tensor Processing Units (TPUs) [11], new algorithms or improved ones.

Languages are challenging by definition, because of their differences, their abstractions and their ambiguities; consequently, their processing is often very demanding, in terms of modelling the problem and resources. Retrieving all sentences in a given text is something that can be easily accomplished with just few lines of code, but what about checking whether a given sentence conveys a message with sarcasm or not? This is something difficult for humans too and therefore, it requires complex modelling mechanisms to be addressed. This kind of information, in fact, poses the problem of its encoding and representation in a meaningful way.

The majority of research involves finding and understanding all characteristics of text, in order to develop sophisticated models to address tasks such as Machine Translation, Text Summarization and Question Answering.

This work will focus on one of the recently released state-of-the-art models and investigate its performance on the Question Answering task. In addition, some ideas will be experimented in order to improve the model, after exploring breakthrough changes that made training and fine-tuning of huge language models possible.

In particular, this work is structured as follows:

- Chapter 1 contains an introduction about Neural Networks, their usage and what role they play in text representation
- Chapter 2 describes the Question Answering problem, differences between old and new models tackling it and the datasets used
- Chapter 3 introduces in detail the Transformer [41] architecture that can be considered the basic building blocks of all current state-of-the-art models
- Chapter 4 describes ALBERT [21], the model this work focuses on and the necessary steps involved in its design
- Chapter 5 shows the ideas employed in order to improve ALBERT [21], with an additional focus on training pipeline and results
- Appendices A and B cover significant information such as model hyperparameters and code.

Table of Contents

List of Figures

List of Codes

List of Tables

Chapter 1

Neural Networks

A *Neural Network* is a computational model whose basic building block is the *artificial neuron*. At the beginning, its concept was highly inspired to the biological neuron, as proposed and logically described by McCulloch-Pitts [25], but later on it shifted to a simplified representation, without taking into account all the characteristics of the biological neuron, as described in Section 1.1.1. Generally speaking, these small computational units can be grouped and stacked together to form what is recognised as a Neural Network. As a consequence, there are several types of them with different characteristics but one of the main classification is: *shallow* versus *deep*.

A network is defined as *shallow* when it only has one hidden layer between the input and output layers; it is considered *deep* otherwise, as shown in Figure 1.1. This qualification, however, is quite broad as there are no proved evidence about the number that configures either one type or the other. In addition, even though this concept has been investigated for decades, it is still unclear when it is better to use shallow networks instead of deep ones.

According to the Universal Approximation Theorem [7], it is always possible to create a network capable of approximating any complex relation between input and output. In particular, any shallow network with a finite number of neurons in its hidden layer should be enough. However, in the reality, deep networks are more used because they require less resources.

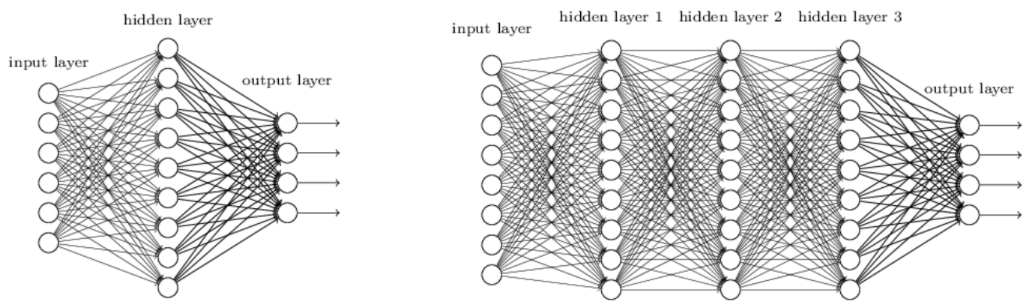


Figure 1.1: An illustrated comparison of a shallow network (left) and a deep network (right). Image taken from [28].

Again, a proof about why this happens is still not available and this belief only follows after a trial-and-error approach on many tasks. In particular, it is believed deep networks may represent a composition of several functions, where previous layers encode simpler features on top of which following layers build more complex ones. For this reason, a neural network can also be defined as a function approximator which tries to model a function between the input and output, but depending on millions of parameters.

The following sections will describe in detail many of the concepts that contribute to define this kind of model and some of its different types.

1.1 Basic Concepts

1.1.1 Neuron

The artificial neuron is the smallest computational unit in a neural network; according to the Figure 1.2, it is easily modelled from the biological neuron. All the input signals of a neuron coming from its *dendrites* are joined with their respective *synapse* and then, processed by the *cell body* itself, before being sent to other neurons through the *axon*.

[1]Image taken from `https://is.gd/Ry7PcF`

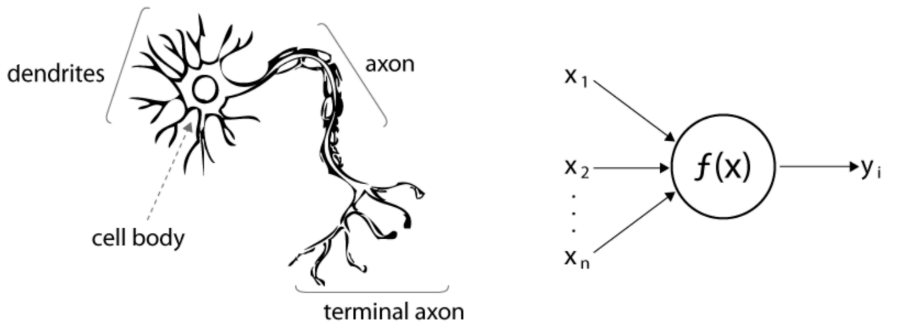


Figure 1.2: The artificial neuron model (right) compared to its biological counterpart (left)[1]

Even though this could seem quite difficult to be modelled, its mathematical translation is very simple: every input signal is represented as a real number, $x_1, x_2, \ldots, x_n$, every synapse with a *weight* factor $w_1, w_2, \ldots, w_n$ and the output is just the weighted product of all signals. In order to model the firing mechanism of the biological neuron, the so-called *bias* term b is introduced to model a threshold which is associated to every neuron. Given z the output of a neuron, Equation 1.1 shows this mathematical representation.

$$z = b + \sum_i w_i x_i \tag{1.1}$$

After applying some simple algebraic concepts, Equation 1.1 can be rewritten in a more compact way, shown in Equation 1.2, where x is the input vector, w is the weights vector and the summation has been replaced with the dot product.

$$z = w \cdot x + b \tag{1.2}$$

1.1.2 Activation Function

The neuron output, as described earlier, is just a linear combination of its input. In the reality, another processing step is employed which consists of applying a non-linear function $f(\cdot)$ to z. It is called the *activation function* and plays an important role in the process of letting a neural network learn. As a consequence, the updated mathematical representation is shown in Equation 1.3.

$$y = f(z) = f(w \cdot x + b) \tag{1.3}$$

During the decades, many activation functions have been used for modelling the learning process of a neural network, in particular the sigmoid function (1.4) and the hyperbolic tangent (1.5). They both have the advantage of mapping the input to a fixed range, $[0, 1]$ and $[-1, 1]$ respectively; however, they both suffer of the *vanishing gradient problem*, which almost prevent large networks to learn at all.

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{1.4}$$

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{1.5}$$

Following the work of Glorot at al. [10] and Krizhevsky at al. [19], a simple activation function such as the Rectified Linear Unit (ReLU) proved to be very beneficial when training particularly large networks and it became the *de-facto* standard. It is very simple, as shown in Equation 1.6, and avoids the vanishing gradient problem because positive input values very close to 1, linearly approach 1, as it can be see in Figure 1.3. However, it has the *exploding gradient problem*, which can almost be considered as the opposite of the vanishing one. Both problems will be described in Section 1.3.5.

$$ReLU(x) = max(0, x) \tag{1.6}$$

Even though ReLU is considered to be the most used activation function nowadays, currently state-of-the-art models, such as the ones described in the following chapters, made Gaussian Error Linear Unit (GELU) [13] popular. It is just a combination of other functions and approximated numbers but the interesting fact is its plot, which is quite similar to ReLU's one.

$$GELU(x) = 0.5x(1 + tanh(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3))) \tag{1.7}$$

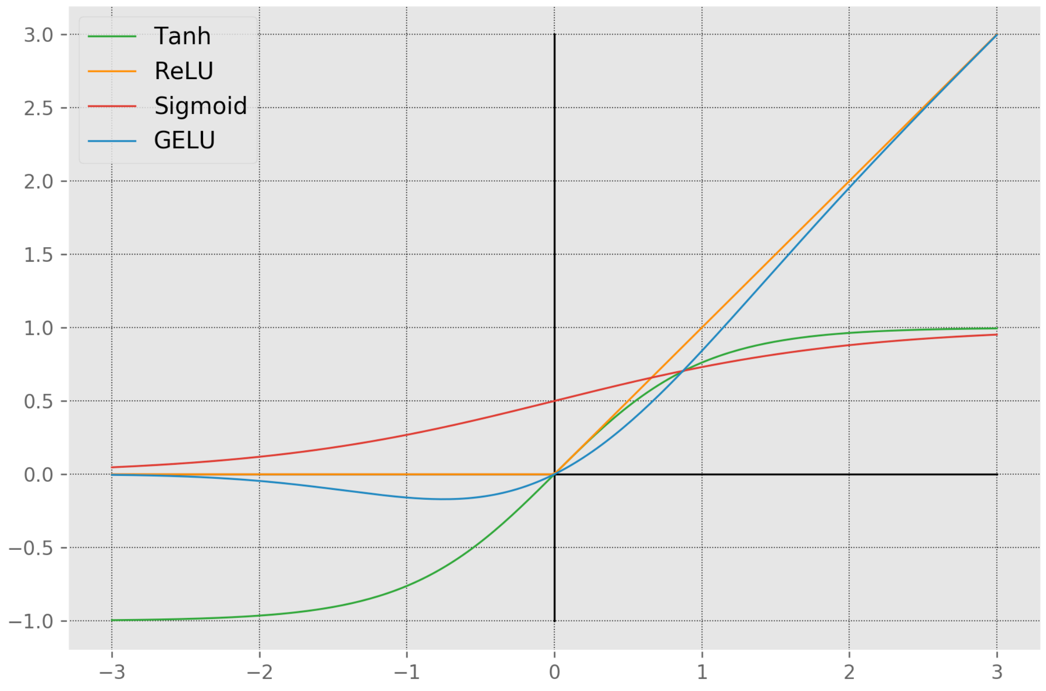


Figure 1.3: Comparison of Sigmoid, Tanh, ReLU and GELU activation functions

1.1.3 Main Layers

In the following layer descriptions, a *forward pass* means the act of feeding an input to the layer and computing the output. More information about this behaviour are described in Section 1.3.

In addition, a *feature* is every kind of relevant information extracted from data, which is useful to compute the output. As it will be shown, the purpose of many layers is exactly to learn how to extract these features, in order to obtain a *feature map*.

Fully Connected Layer

The *Fully Connected Layer* is the simplest type of layers in a neural network. As its name suggests, it has full connections with all the outputs from the previous layer but no connections between neurons in the layer itself. The forward pass of a fully connected layer is just one matrix multiplication with weights, followed by a bias offset and an activation function.

Weights matrix associated to this layer has a shape of (n, m) and a bias vector b, where n is the number of neurons in the current layer (k), while m is the same number but referring to the previous layer $(k-1)$. According to the Equations 1.1 and 1.3, this operation will compute:

$$y^{(k)} = f(\begin{pmatrix} w_{1,1} & \dots & w_{1,m} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \dots & w_{n,m} \end{pmatrix} \begin{pmatrix} y_1^{(k-1)} \\ \vdots \\ y_m^{(k-1)} \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}) \tag{1.8}$$

Supposing a simple network with an input layer of shape $(2, 1)$, a fully connected layer of shape $(3, 1)$ and an output layer of shape $(1, 1)$, the result of the fully connected layer is given from the following equation:

$$y^1 = f(\begin{pmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{pmatrix} \begin{pmatrix} y_1^0 \\ y_2^0 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}) = \begin{pmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \end{pmatrix} \tag{1.9}$$

Convolutional Layer

As its name suggests, this layer applies the operation of *convolution* to its input. In the context of neural networks, however, the definition is slightly different from the pure mathematical one: it is a linear operation that involves the multiplication (dot product) of the input and a smaller weights matrix, called *filter* or *kernel*. Because of its smaller size, a filter can be thought as a light that slides over the input, applying the convolution on the highlighted portion only. This sliding approach involves just two dimensions, width and height, from left to right, top to bottom. Even though the filter considers two dimensions only, it will extend through all the other dimensions, such as depth. In fact, every filter will produce a different feature map which are stacked together along the depth dimension.

This mechanism is governed by some hyperparameters:

- *Depth* (not to be confused with the dimension): states the number of different kernels to learn that are applied on the same portion of input
- *Stride*: states the amount of neurons/pixel the kernel has to slide from the previous position
- *Zero-Padding*: states the number of padding neurons at the borders in order to make the filter fit the input; this will also allow to arbitrarily control output size

Given an input of size (Wi, Hi, Di), a kernel width KW, a stride S, a padding P and the number of kernels to be applied K, the output will be of size (Wo, Ho, Do), where $Do = K$ and Wo given by the Equation 1.10 (similarly for Ho). An example of a convolution is shown in Figure 1.4: given the input of shape $(3, 3, 3)$, two kernels of shape $(2, 2, 3)$ are applied with $S = 1$ and $P = 0$.

$$Wo = \frac{Wi - KW + 2P}{S} + 1 \tag{1.10}$$

Pooling Layer

A *Pooling Layer* applies a downsampling operation to its input, in order to reduce input dimensionality and the associated number of parameters. Similarly to what happens with convolutional filters, this layer also supposes the usage of a particular filter all over the input; in other words, it applies a function, generally *max* or *average*, on portions of input data.

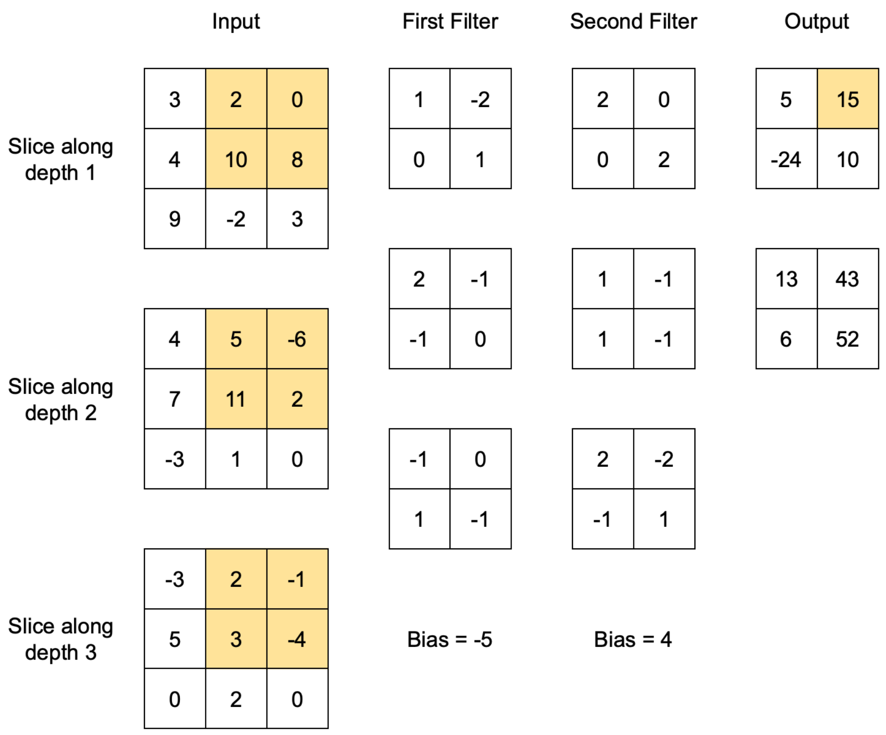


Figure 1.4: Example of convolutional layer output; the highlighted numbers are the ones who contribute to compute 15 in the first output slice, since the first kernel is applied

A pooling operation depends on two hyperparameters: filter width K and stride S. Given an input of shape (W_1, H_1), the output dimensions are computed with the Equation 1.11 (similarly for the height). Figure 1.5 shows an example of both max and average pooling with $K = 2$, $S = 2$ on an input of shape $(4, 4)$; the colours refer to the portion on which the function has been applied. Note that if the input shape is composed of additional dimensions, such as the depth, this operation will leave them untouched.

This layer with $K = 2$ is the most common one, together with the *overlapping pooling*, where $K = 3$ and $S = 2$; however, increasing filter size usually results in information loss.

$$W_2 = \frac{W_1 - K}{S} + 1 \tag{1.11}$$

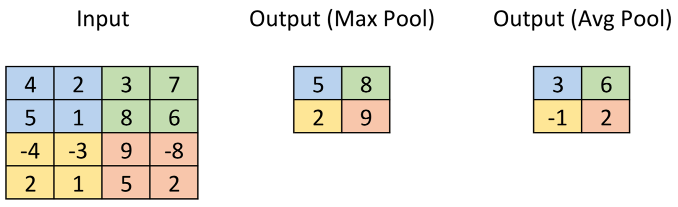


Figure 1.5: Example of max and average pooling operations

1.2 Types of Neural Networks

Before describing some of the main types of neural networks, layers can be classified based on the position they have in the architecture. For instance, the *input layer* is the first one which expects model input; the *output layer* is the last one and is designed according to the task the model has to tackle; and finally, all the others, if present, are called *hidden layers*.

1.2.1 Feed-Forward Network

A *Feed-Forward Network* is the simplest type of network where the computation flows only from the input layer to the output one and, in particular, iteratively from one layer to the next one. In addition, its structure does not expect back connections or cycles. Strictly speaking, a network of this type is composed of fully connected layers but this term is often used to only state the iterative, one-directional way of computation.

The simplest example is the *Single Layer Perceptron* network, which is only composed of an input and an output fully connected layers, and its generalization, called *Multi Layer Perceptron*, because it has at least an hidden layer in between.

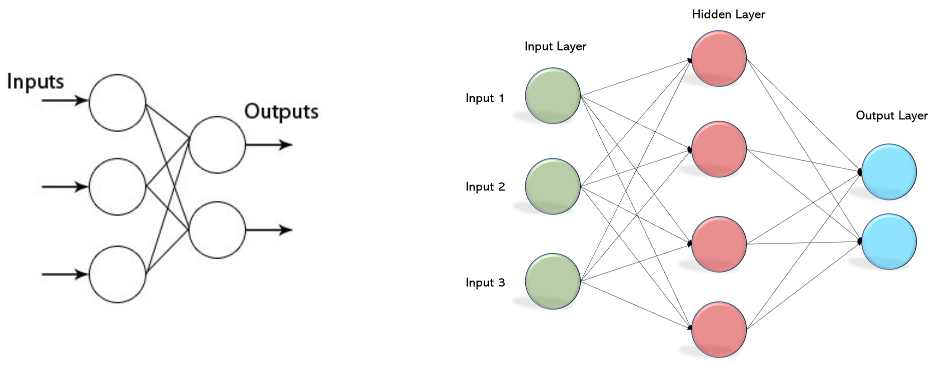


Figure 1.6: Single (left) and Multi (right) Layer Perceptron networks[2]

1.2.2 Convolutional Neural Network

A *Convolutional Neural Network* is a deep feed-forward network mainly composed by convolutional and pooling layers. They also usually have some final fully connected layers depending the task the model has to address, particularly for classification.

This kind of network is highly inspired to the humans' visual cortex and the concept of receptive field because smaller portions of data are considered while being processed, as described in Section 1.1.3. Because of their usual architecture and the mathematical operations applied, Convolutional Networks have the following main characteristics:

- They are *local connected*, as every neuron of a layer is connected to only a local region of its input, as opposed to fully connected layers. As a consequence, number of parameters is highly reduced
- Instead of vectors and matrices, they work with data treated as *3D volumes* $(width, height, depth)$, according to a specific spacial arrangement
- Under the assumption that if a learned feature is useful on a specific spacial position, it will also be useful on another position, filter applied on data slices along the depth dimension can share weights and biases

[2]Images taken from `https://is.gd/TOok7j` and `https://is.gd/KDlpRO`

- They are *translation invariant*: if the network learns to detect a particular feature, it will continue to detect it even if it gets translated in another position

Their main and most effective applications involve Computer Vision, with image classification, analysis and recognition, and Natural Language Processing, with sentence classification and feature extraction.

Important early works regarding convolutional networks have been proposed by Y. LeCun, such as the LeNet [22] architecture. Subsequent famous works are AlexNet [18], GoogLeNet [40] and VGGNet [37].

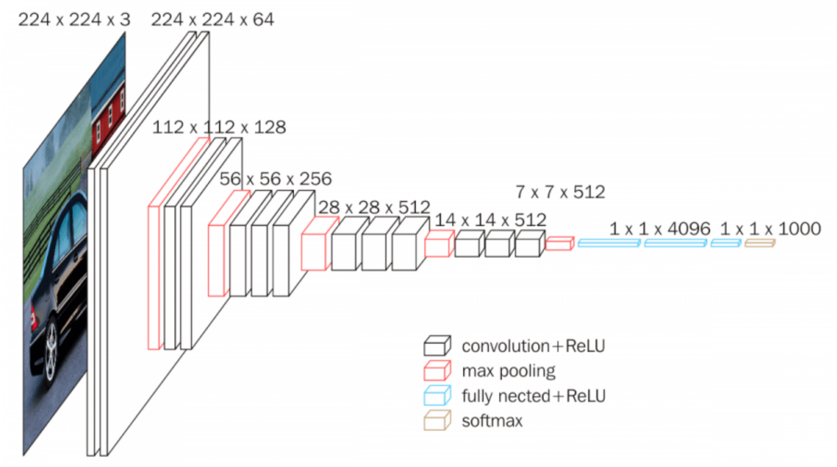


Figure 1.7: Convolutional Neural Network example: the VGG16 architecture[3]

1.2.3 Recurrent Neural Network

Traditional networks as the ones described earlier suppose every input is independent from the others. Consequently, sequential information are poorly handled. On the contrary, *Recurrent Neural Networks* perform the same computation on every element of an input sequence, where the output

[3]Image taken from `https://is.gd/pGYy31`

depends on the previous ones. In particular, they can be viewed as networks with a memory of all that has been computed so far. In order to make this possible, a Recurrent Network can be considered very similar to a Feed-Forward one, with the main difference that the former allows the presence of cycles in its architecture. Actually, this is what characterizes them the most, as shown in Figure 1.8.

Because text and language are inherently sequential, this type of networks became popular for Natural Language Processing tasks, such as, Next Word Prediction or Sequence Labelling.

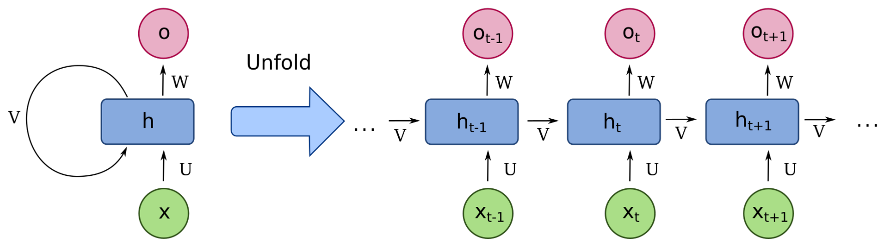


Figure 1.8: Recurrent Neural Network example; on the right the extended, unfolded version of the network[4]

For every token in a sequence, it is fed to an hidden layer to compute the related output but the hidden state of the previous token is fed too. By doing so, the predicted output value takes into account also information about previous tokens, just like an informed *context*.

From an analytical point of view, the equations to compute outputs slightly change, as they have to take into account new and different weights and biases. Generally speaking, given a time step or token position t, the following equations apply:

$$a^t = g(U \cdot x^t + V \cdot a^{t-1} + b_a) \tag{1.12}$$

[4]Image taken from `https://is.gd/utjo8r`

$$y^t = f(W \cdot a^t + b_y) \tag{1.13}$$

W, U and V are learnable weights matrices, b_a and b_y are bias vectors for computing hidden state activations and outputs, respectively. Such matrices are shared across time steps/token positions so the model will not increase its number of parameters with the size of input.

Because of their structure, Recurrent Networks have the advantage of allowing inputs of an arbitrary dimension; in the reality, however, this poses the problem of long-term dependencies, as it is difficult to keep track of contextual information for rather long sequences. In addition, they are quite expensive to train and slow to compute, due to their sequential nature, and they only get information from the past context instead of the future one too.

Many state-of-the-art models described later in this work focused on tackling many of these drawbacks, in order to obtain feasible and powerful language models.

1.3 How Neural Networks Learn

After modelling a neural network and its architecture, how is it possible to make them learn? How can they really understand their input and decide which answer to return? As it has been said earlier, everything is inspired to the human brain but the reality consists in only *learning algorithms* which accordingly update numbers.

1.3.1 Hebbian Learning

One of the first learning algorithms has been proposed by D. Hebb in 1949, following the neuroscientific theory often summarized as *"cells that fire together, wire together"*. In other words, the fact that two neurons fire together has the consequence of strengthening the synapse between them, or weakening it otherwise.

From the point of view of artificial neural networks, nodes that tend to be both positive or both negative at the same time have a positive weight connecting them, a negative one otherwise.

In general, this idea does not take into account a ground truth value, so tasks which require a supervised learning approach are not feasible with this learning algorithm.

1.3.2 Gradient Descent

Another learning algorithm is *Gradient Descent*, which is an instance of local optimization algorithms. On the contrary of Hebbian Learning, this algorithm supposes the knowledge of ground truth values, in order to compare them with the model predicted output. This comparison is made through a *loss function*, which has to be continuous and differentiable.

The general idea of gradient descent is to minimize the loss function, by discovering the lowest minimum point of that function, as shown in Figure 1.9. Ideally, when this point is reached, the loss is optimal so model accuracy should be as high as possible.

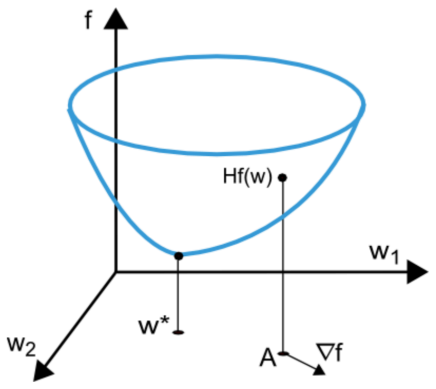


Figure 1.9: A simple cost function representation; in the reality, more and more parameters are involved[5]

The minimum loss value is iteratively found by computing the *gradient* of the loss function. Given a N-dimensional space, where N states the number of parameters the model depends upon, the gradient of the loss function with respect to the parameters is a vector whose components are partial derivatives of the loss function. Equation 1.14 shows this formulation, where L is the selected loss function, y is the expected output value and $f(x;\theta)$ is the predicted output value which depends on model parameters θ.

$$\nabla_\theta L(f(x;\theta),y) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\theta),y) \\ \frac{\partial}{\partial w_2} L(f(x;\theta),y) \\ \vdots \\ \frac{\partial}{\partial w_N} L(f(x;\theta),y) \end{bmatrix} \tag{1.14}$$

After computing the gradient, weights and biases are updated according to the rule in Equation 1.15, where ∇L is the gradient and η is the *learning rate*. This hyperparameter is particularly important when training neural networks because it states the magnitude of the gradient updates on the weights. In fact, when this value is too small, the model will train very slowly; when it is too big, the model may not learn as the minimum value could be skipped, resulting in a loss function divergence.

$$\theta_{t+1} = \theta_t - \eta \nabla L(f(x;\theta),y) \tag{1.15}$$

1.3.3 Backpropagation

Since the loss function is computed at the end of the neural network, the derivatives shown in the previous section are only valid for the last layer, even though the majority of parameters belongs to intermediate hidden layers. *Backpropagation*, then, is an algorithm to appropriately compute the gradient taking into account the architecture of the network. In other words,

[5]Image taken from `https://is.gd/Qpbkim`

it propagates the information about the loss back to the input layer by using the concept of computation graph.

Having a mathematical expression, its *computation graph* is a representation in which the computation is broken down into separate operations and each one of them represents a node in the graph. In the forward pass, input values are passed left to right in the graph, computing the operations in the nodes values flow through; in the backward pass, instead, derivatives with respect to the output function are computed.

This kind of differentiation uses the *chain rule*: supposing a composite function $f(x) = g(u(v(x)))$, its derivative with respect to x is:

$$\frac{\partial f}{\partial x} = \frac{\partial g}{\partial u} \cdot \frac{\partial u}{\partial v} \cdot \frac{\partial v}{\partial x} \tag{1.16}$$

Each partial derivative is calculated along the edges of the graph, right to left, taking into account the operations in the nodes, until the input is reached.

1.3.4 Stochastic Gradient Descent

As the name suggests, this is a modification of the gradient descent algorithm which overcomes a drawback: plain gradient descent considers the entire training dataset when computing one update. As anyone can image, this can be very slow or even unfeasible. On the contrary, stochastic gradient descent considers only one sample from the data to update the weight; this sample is chosen randomly among all the dataset. As a consequence, this algorithm is faster but the minimum it usually finds may not be the optimal one because of the variations caused by each sample.

1.3.5 Undesired Effects

Vanishing/Exploding Gradients

Vanishing or *exploding gradient* is a common problem when dealing with deep neural networks that actually prevented them from being trained until some discoveries in the last few years. This phenomenon happens when gradient-based learning methods and backpropagation are used to train a network, as described in Section 1.3.

Gradients at a specific layer are computed as the multiplication of gradients of prior layers, because of the chain rule. For this reason, having gradients in the range $(-1, 1)$ will result in numbers that gets smaller and smaller, until they vanish and will not update the weights. As a consequence, initial layers in the architecture will not receive a meaningful update, resulting in a long training time and considerable performance loss. On the contrary, if weights initialization and gradient computation produce large numbers, exploding gradients will happen, as their multiplication will easily tend to infinity.

Overfitting/Underfitting

Another undesired effect when developing machine learning or statistical models is the *overfitting* or *underfitting* of data. The purpose of these models is to find the best approximated function that describes the data. In particular, it should be able to generalize, in order to correctly be applied on new data.

Speaking about overfitting, it can happen under different situations, such as when using a model with too many parameters or when it is trained for too long or when the training data used is not enough. In fact, in such cases, the model will not focus on the relevant relation that describes the data but it will learn also statistical noise and useless information.

Underfitting, on the contrary, can be seen as the opposite problem: it happens when the model can not adequately capture the underlying structure of the data because the representative power is too low or is has been trained for too few time.

1.4 Normalization and Regularization

1.4.1 Dropout

Dropout [38] is a regularization technique used to prevent a model from *overfitting* while training. It is different from other techniques, such as L1 and L2 regularization as it does not change the cost function, but instead, it relies on changing the network itself and updating only a subset of weights and biases.

Large neural networks often suffer of *co-adaptation*: stronger connections between a subset of neurons is learned and they overlook the weaker ones. The direct consequence of this behaviour is that the model will also learn some statistical noise from the training dataset. In other words, it will perform really well on that dataset but its generalization capabilities will reduce considerably.

Dropout tries to prevent this by randomly deactivating some neurons and their connections with a probability sampled from a Bernoulli distribution of parameter p. As a consequence, only a subset of neurons is trained, breaking the co-adaptation problem and making them more robust because they have to learn features without relying too much on other neurons. After every training iteration, deactivated neurons are restored and a new subset is sampled. From a practical perspective, this means that multiple different models are trained at the same time: given the model with n units, it can be thought as a collection of 2^n smaller networks.

At test time, no units are deactivated and all weights are multiplied by p to guarantee coherence on the output between both training and test time. In addition, considering all weights together acts as a form of averaging between all smaller models.

A parameter $p = 0.5$ is quite frequent in the Computer Vision field, even though this technique has been overcome by others, such as Batch Normalization [15]; in NLP, instead, a value $p = 0.1$ is more common despite difficulties of its application due to the nature of problems.

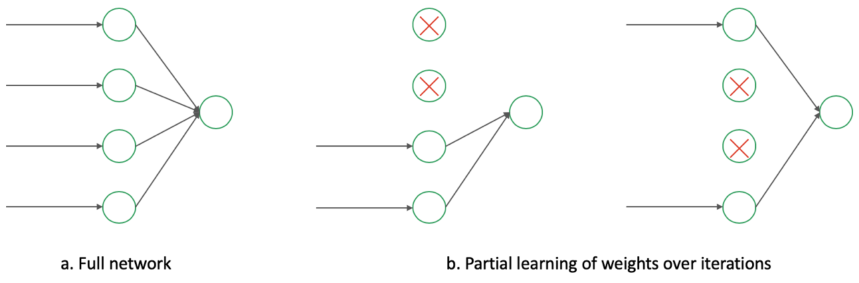


Figure 1.10: Dropout intuition; neurons in b) will result in two different models[6]

1.4.2 Batch Normalization

Because of how neural networks are structured, the output of every layer is passed as input to the following layer and so on, until the output one. This has implications on the distributions of inputs, leading to the concept of *Internal Covariate Shift* [15], which is the change in the distribution of network activations due to the change in network parameters during training. These shifts can be problematic because the discrepancy between subsequent layer activations can be quite pronounced.

Batch Normalization [15] then, aims to limit the shifting problem by normalizing the output of each layer; in particular, the network will learn γ and β parameters to transform the input distribution in order to have zero mean and unit variance.

Given a mini-batch $\mathcal{B} = x_1, \cdots, x_m$, the first step involves the computation of mini-batch mean and variance, which are then used to normalize mini-batch values, as shown below.

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{1.17}$$

[6]Image taken from `https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space`

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \tag{1.18}$$

$$\widehat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{1.19}$$

$$BN_{\gamma,\beta} \equiv \gamma\widehat{x}_i + \beta \tag{1.20}$$

Because of its formulation, one of Batch Normalization drawbacks is that it can not be applied to settings where mini-batch size is not big enough to obtain good results, for instance at least 32 or 64. In this case, other forms of normalization are preferred.

1.4.3 Layer Normalization

Layer Normalization [2] has been introduced to address some drawbacks of Batch Normalization, such as the overcomplication needed to apply it in *Recurrent Neural Networks* (RNNs) or the requirement of big mini-batches.

Recalling, a *mini-batch* is a set of multiple examples with the same number of features and it is represented as a matrix/tensor where one axis corresponds to the batch and the others to feature dimensions. On the opposite of Batch Normalization, which normalizes over the batch dimension, Layer Normalization learns γ and β parameters (mean and variance) over features, as it is shown in the equations below. As a consequence, these values are independent from other examples, allowing an arbitrary mini-batch size.

$$\mu_i = \frac{1}{m}\sum_{i=1}^{m} x_{ij} \tag{1.21}$$

$$\sigma_i^2 = \frac{1}{m}\sum_{i=1}^{m}(x_{ij} - \mu_i)^2 \tag{1.22}$$

$$\widehat{x}_{ij} = \frac{x_{ij} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \tag{1.23}$$

$$LN_{\gamma,\beta} \equiv \gamma \widehat{x}_{ij} + \beta \tag{1.24}$$

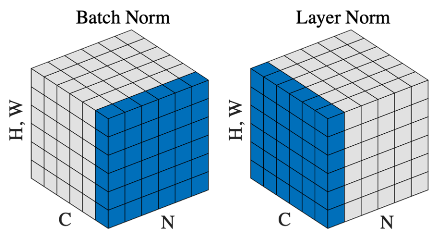


Figure 1.11: Visualization of difference between Batch and Layer Normalization[7]

1.5 Transfer Learning and Fine Tuning

Transfer Learning is the process of training a model on a large-scale dataset and then, use the *learned knowledge* as a base for a downstream task. This type of learning has been quite popular since the ImageNet 2015 Challenge [33], where the winning model showed how powerful such technique could be for Computer Vision. Later on, research in that field literally exploded but other fields, such as NLP, still struggled. In fact, examples of successful transfer learning applied to text were only few and usually too specific to be generalized.

In the last few years, however, *pretrained language models* showed their power, enabling a generic transfer learning for NLP too. The general procedure, in fact, consists in pretraining a model on a large *unlabelled* dataset and then, adapting its learned representation on a supervised target task with a new, smaller *labelled* dataset.

The introduction of BERT [8], whose description will follow in the next chapters, marked huge progress in many state-of-the-art baselines for many

[7]Image taken from [44]

different tasks, such as Question Answering and Sentence Classification. Following transfer learning, a fine tuning phase is usually applied: in this case, models are slightly adjusted to improve their performance on a smaller and more specific task. In other words, transfer learning makes a model learn general understanding abilities while fine tuning leverages them to tackle one precise problem.

1.6 Embeddings

From a practical point of view, neural networks expect a numerical input to process when they have to fulfil a task. This kind of input, however, is not composed by random numbers but instead, they semantically represent a high-level concept or object. A simple example is an image, where every number in its encoded representation states the intensity of every base colour for each pixel. Some inputs are intrinsically easy to be represented by numbers, while other not, such as text. It is not immediate to transform a text in a series of numbers because of all the meanings it conveys, the relations between words, its structure, its similarities and so on.

Vector semantics is a model that tries to encode all these characteristics by taking into account the environment in which every words is usually employed and a vector representation. Such vectors are called *embeddings* because they embed a word in a particular high-dimensional vector space. Embeddings can be either *sparse* or *dense*: the main difference is the dimensionality of the vector; the former, in fact, is usually as big as the vocabulary, while the latter has a fixed size, usually about hundreds of values.

Given a vocabulary of size V, an example of sparse embedding is the *one-hot encoding*: a vector of size V is associated to every word in the vocabulary. The encoding for the i-th word will have a 1 in the i-th index and zeros anywhere else. Generally speaking, sparse embeddings can become unfeasible to be used in tasks where the vocabulary size is very big, making dense ones more suitable for them.

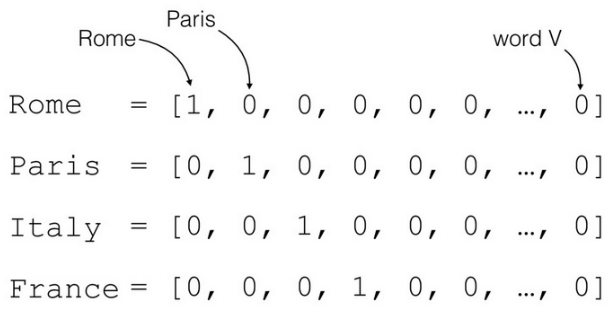

Rome
Paris
word V
Rome = [1, 0, 0, 0, 0, 0, …, 0]
Paris = [0, 1, 0, 0, 0, 0, …, 0]
Italy = [0, 0, 1, 0, 0, 0, …, 0]
France = [0, 0, 0, 1, 0, 0, …, 0]

Figure 1.12: Example of one-hot encoding of words[8]

Some other kinds of embeddings follow from the distributional semantics research area and the so-called *distributional hypothesis*, which states that *"words occurring in similar contexts tend to have similar meaning"*. This hypothesis is the base of many statistical methods for analysing the relations between terms and documents, mainly focusing on counting their occurrences.

1.6.1 Word2Vec

Word2Vec [26] is a set of models designed to compute dense and short word embeddings. The intuition consists of replacing count statistics between words with a binary classifier whose task is to predict whether a word x is likely to show up near another word y.

Given a large corpus of text, Word2Vec will produce a vector space in which word vectors are positioned, according the assumption that word sharing a similar context in the corpus will be closely located in the vector space.

Continuous bag-of-words (CBOW) or *Skip-gram* are the two models used by Word2Vec: the former predicts the next word by looking at some surrounding context words, while the latter does the opposite, predicting a set of context words given the current word. Once the learning phase for the classifier is done, the embeddings consists of the weights the model has learned

[8]Image taken from `https://is.gd/IG6W4P`

for every word.

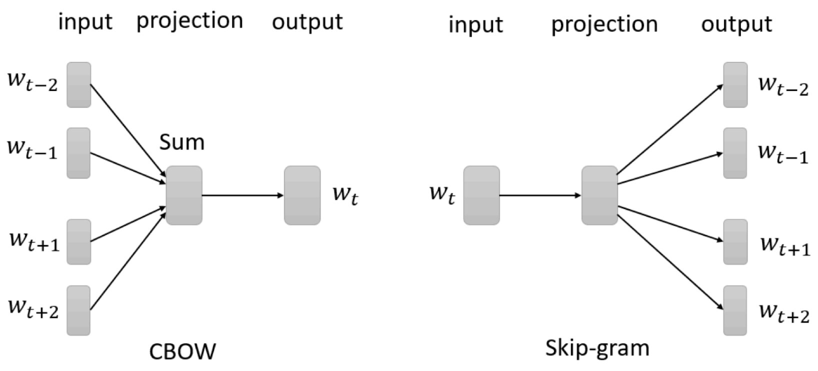


Figure 1.13: Word2Vec CBOW and Skip-gram block-diagram example[9]

[9]Image taken from `https://is.gd/x3WTdD`

Chapter 2

Question Answering

Question Answering (QA) is the field belonging to Natural Language Processing whose purpose is to create a system which is capable of automatically answer questions made by humans using natural language. Every QA system can be classified as *closed-domain*, when only questions regarding a specific and narrow topic can be asked or when they are limited (i.e. *What is the capital of Italy?*), or as *open-domain*, when basically anything can be requested (i.e. *What was your last travel trip experience like?*).

Since the early stages in the 1960s, a *knowledge base* has been the primary source of information, shifting to *information retrieval* (IR) approaches in the late 1980s. By definition, a knowledge base is a way of storing complex structured and unstructured data in a form of facts, assumptions and rules; for this reason, this data representation usually requires an expert that has enough knowledge to model it in an appropriate way. Many expert systems, in fact, rely on a knowledge base and an inference model to reason and provide answers. Due to the design, this kind of system is difficult to scale and they are only used for narrow and specific topics.

Nowadays, even though many IR systems are available and used, there is a growing interest in *machine reading comprehension* (MR) models. Watson [9], from IBM, is an example of QA system based on a huge knowledge base which became famous when it defeated Jeopardy top-player in 2011, putting the spotlight on how powerful machines can be.

IR and MR models are quite different: the former involve searching for documents, deciding the relevant ones and identifying passages of text that can provide an answer, while the latter assume a question and a passage will be given and their purpose is to understand them, in order to look for the correct answer.

2.1 Evaluation Metrics

After developing a system, it has to be evaluated according to some metrics; their choice usually depends on the particular task and data format the model returns, in order to obtain real and meaningful information. In other words, the purpose of evaluation is answering the question *how well does the system work?*. Regardless task type, either it is generic such as *classification* or *regression*, or more specific, the main idea is always to compare the predicted output from the model and the *ground truth*, representing the real expected values.

Confusion Matrix

Before digging into some of the main metrics, it is important to understand the Confusion Matrix, which is a tabular or visual representation of the relations between predicted labels and real ones from the ground truth, of size $(num\ labels, num\ labels)$. Figure 2.1 shows the confusion matrix on a subset of the Iris Dataset [1], restricted to two classes only.

This representation is particularly useful because it allows to obtain the required information to compute multiple metrics:

- **True Positive** (TP): number of correctly predicted examples of the positive class
- **False Positive** (FP): number of examples whose predicted class is the positive one but the actual one was another

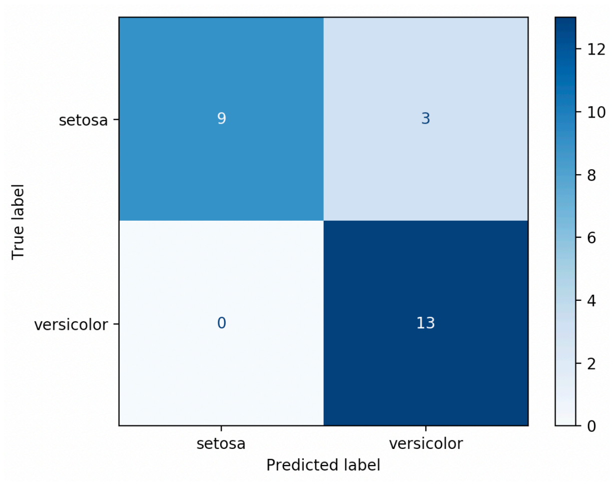


Figure 2.1: Confusion Matrix computed on a subset of the Iris Dataset

- **False Negative** (FN): number of examples whose actual class is the positive one but another one was predicted
- **True Negative** (TN): number of correctly predicted examples of all the other classes

Accuracy

Accuracy is one of the simplest metrics as it states the ratio between correctly predicted values and total number of predictions.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{\#\ correct\ predictions}{\#\ total\ predictions} \tag{2.1}$$

Precision

Precision is a classification measure that intuitively states the ability of a classifier not to label as positive samples that are negative; it is also called *Positive Predictive Value* (PPV). This metric can be more meaningful than accuracy when classes in a dataset are imbalanced: in fact, in that case the model would correctly predict the most frequent class, thus resulting in a high accuracy rate; in the reality, however, it may not be learning at all the least frequent classes.

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

Recall

Recall intuitively states the ability of a classifier to find all the positive examples; it is also called *Sensitivity.* If used alone, it may not be very meaningful, as it is easy to obtain the maximum recall; to avoid this scenario, it may be necessary to compute also statistics about non-relevant examples, such as precision.

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

F1 Score

The F1 Score, or F-measure, is the harmonic mean between precision and recall. In case of binary classification, it can be seen as a measure of a test's accuracy. It is mainly used in settings where both precision and recall are important, such as in Natural Language Processing.

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.4}$$

Generally speaking, the F_β score is given by the Equation 2.5, where β states how many times recall is more important than precision; for this reason, the F_1 gives the same importance to them.

$$F_\beta = (1 + \beta^2) \cdot \frac{Precision \cdot Recall}{(\beta^2 \cdot Precision) + Recall} \tag{2.5}$$

Exact Match

In Natural Language Processing, Exact Match (EM) score is a simple but very useful metric which states the ratio between correctly predicted values and total number of predictions, with the constraint that the predicted value must be exactly the same as the expected one.

Considering the example in Figure 2.1, we treat the “versicolor” class as the positive one, obtaining the following metrics:

$$TP = 13 \qquad FP = 3 \qquad FN = 0 \qquad TN = 9$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{13 + 9}{13 + 9 + 0 + 3} = 0.88 = 88\%$$

$$Precision = \frac{TP}{TP + FP} = \frac{13}{13 + 3} = 0.8125 = 81.25\%$$

$$Recall = \frac{TP}{TP + FN} = \frac{13}{13 + 0} = 1.0 = 100\%$$

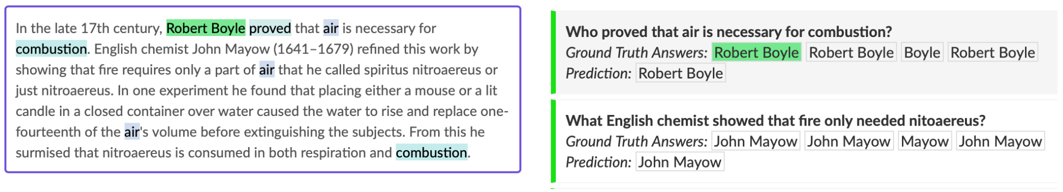
$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = 2 \cdot \frac{0.8125 \cdot 1.0}{0.8125 + 1.0} = 0.8965 = 89.65\%$$

2.2 Datasets

Many dissimilar dataset are available, such as TriviaQA [16], WikiQA [45], WikiReading [14], MSMarco [4] and SQuAD [30], with diversities on the way questions and answers have been created, the number of questions and the source documents considered. For instance, WikiQA is very similar to SQuAD, as it also considers Wikipedia documents but its purpose is sentence selection instead of span extraction. Some of them also assume presence of questions whose answers are not in the related text passage; the intuition behind this choice is that models should also be able to evaluate when information provided are enough to return a meaningful answer, otherwise it is better to avoid answering. However, due to every dataset key aspects, not all machine reading question answering datasets can be used for the span extraction task; in this work, we mainly focused on the SQuAD one.

2.2.1 SQuAD Dataset

Stanford Question Answering Dataset (SQuAD) is one of the biggest reading comprehension dataset that is available to the community. In fact, it is a collection of more than 100000 question-answer pairs, created by crowdworkers on a set of 400+ Wikipedia articles. In particular, answers have been annotated manually by selecting a span of text in the passage, together with the starting index. For this reason, this dataset can be used with every model capable of span extraction.



In the late 17th century, Robert Boyle proved that air is necessary for combustion. English chemist John Mayow (1641–1679) refined this work by showing that fire requires only a part of air that he called spiritus nitroaereus or just nitroaereus. In one experiment he found that placing either a mouse or a lit candle in a closed container over water caused the water to rise and replace one-fourteenth of the air's volume before extinguishing the subjects. From this he surmised that nitroaereus is consumed in both respiration and combustion.

Who proved that air is necessary for combustion?
Ground Truth Answers: Robert Boyle Robert Boyle Boyle Robert Boyle
Prediction: Robert Boyle

What English chemist showed that fire only needed nitoaereus?
Ground Truth Answers: John Mayow John Mayow Mayow John Mayow
Prediction: John Mayow

Figure 2.2: Example of paragraph from SQuAD v2.0 dataset, with questions and answers

One thing characterizing SQuAD v1.1 [31] is the assumption that every question has at least an answer in the provided text; in other words, a model trained on it will always predict an answer, even though it is not relevant. This assumption, of course, oversimplifies the reality as situations of uncertainty and lack of information could arise in every moment.

Consequently, SQuAD v2.0 [30] has been released as a brand-new dataset during 2018. The inclusion of about 40000 new pairs of unanswerable questions is supposed to tackle the assumption described earlier and these new information are marked by the attribute `is_impossible` set to *True*.

The train dataset, however, is not balanced with respect to the question type: as you can see from the summarizing table below, the ratio is 67%-33% in favour of answerable questions. Instead, the dev dataset is well balanced.

In addition to train and dev, an evaluation script on the dev set has been provided, allowing researchers to evaluate their models before submitting results to the official leaderboard [39]; this is necessary because the real

	Train Dataset		Dev Dataset	
	Questions	Unanswerable	Questions	Unanswerable
SQuAD v1.1	87599	–	10570	–
SQuAD v2.0	130319	43498	11873	5945

Table 2.1: Questions distribution by type across datasets.

figures appearing online are evaluated on a test set that has been kept closed-source. According to the scores shown online, SQuAD v1.1 is considered solved as many models exceeded the human performance of 82.304 EM and 91.221 F1 measure. Version 2.0, instead, is more challenging (86.831 EM and 89.452 F1 measure) and new models are being developed week after week.

The dataset is released as a JSON object, whose every inner object represents a specific topic; each one of them is a collection of one or more text paragraphs with the associated question objects and answers, when available. An example of SQuAD v2.0 object about Normans, with both an impossible and a possible questions can be seen in Appendix B.1.

Results Comparison

Following the brief introduction about the dataset, Table 2.2 shows a comparison between current state-of-the-art models and previous approaches tackling this problem. In particular, some of them will be described in detail in the following chapters, in order to understand breakthrough changes which led to performance improvements.

Question answering approaches to this dataset changed a lot during the decade; chronologically, in fact, the following systems have been implemented to address this task:

- Sliding Window Approach [32] (2013): baseline system involving the usage of lexical features; bag of words are created taking into account both the question and the hypothesized answer

- Logistic Regression model [31] (2016): extracts several types of feature for every possible answer, most of which are lexical ones, such as matching word and bigram frequencies and lexical variations; they all contribute to make the model pick the right sentence
- BiDAF [35] (2016): this model leverages many different "blocks", such as CNNs, LSTMs and attention mechanisms, focusing on both context and question information in a bidirectional way. It will compute probabilities for every token to be the start and the end of the answer span

Model	**SQuAD v1.1**		**SQuAD v2.0**	
	F1	EM	F1	EM
*Human Performance **	91.2	82.3	89.45	86.8
Sliding Window	20.2	13.2	-	-
Logistic Regressor	51.0	40.0	-	-
BiDAF (single)	77.3	67.7	-	-
$BERT_{BASE}$	88.5	80.8	-	-
$BERT_{LARGE}$	90.9	84.1	81.9	78.7
$RoBERTa$	94.6	88.9	89.4	86.5

Table 2.2: Performance comparison of state-of-the-art models on both SQuAD v1.1 and v2.0 dev datasets. * as stated in [31], [30].

2.2.2 OLP

In addition to SQuAD, another dataset has been used to experiment with the investigated models: it is the OLP dataset [6] [5] developed by a workgroup at Bielefeld University. As it is still under development, it is not publicly available yet. As opposed to SQuAD, OLP is a Question Answering dataset that focuses on free text and knowledge base integration.

One of the main differences between SQuAD and OLP is the fact that the latter is composed of closed-domain questions; in particular, it is about after-game comments of football matches. For this reason, many questions required some degree of knowledge and semantic processing, as the real answer was not explicitly stated in the text.

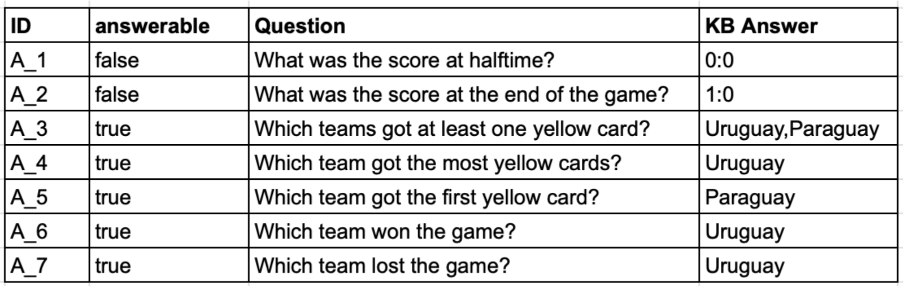
In the next sections, there will be a description of the work done to preprocess, normalize and convert OLP to a SQuAD-like format, in order to feed them to our Question Answering models.

2.2.3 Preprocessing

As it can been seen from Figures 2.3, 2.4, 2.5, the structure is very different, as it is composed of three kind of CSV files: Question, Annotation and Tokens.

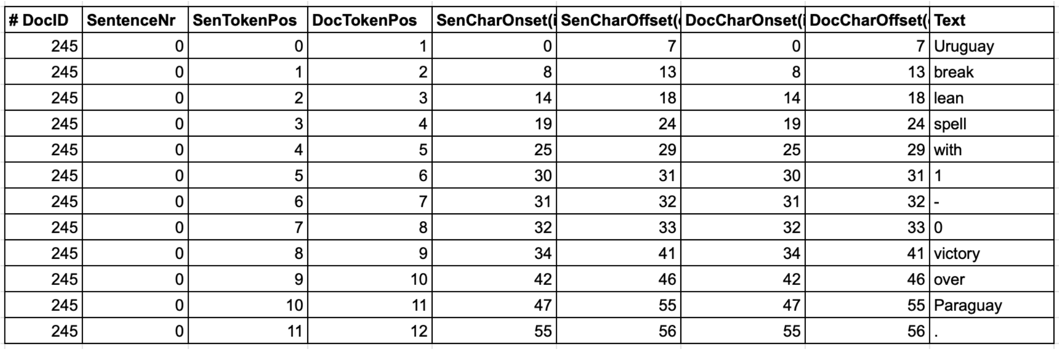
Given a specific topic, the first file contains a list of all the questions about that article, with information such as the ID and the correct answer got from a Knowledge Base; the Tokens file, instead, contains the entire text passage, token by token, with specific offsets (character-based, token-based and sentence-based); finally, the Annotation file contains all the information about the hand-made annotations on the text. This file can be considered the way of linking together questions and texts, as both IDs, offsets and answers are stated.

With support from Simone Preite and Frank Grimm, we tried to deeply understand how the dataset was composed and to which extent it was different from SQuAD, in order to write the appropriate logic to do a conversion. We wrote many Python scripts to automatically normalize and build the dataset in the proper format but we also had to manually fix mismatches between annotations and questions or wrong knowledge base answers. In addition, we removed duplicate entries and cleaned offsets values, as multiple whitespaces caused them to be unreliable. In addition, we added `is_impossible` information to every question by checking whether the answer was explicitly stated in the text or not.



ID	answerable	Question	KB Answer
A_1	false	What was the score at halftime?	0:0
A_2	false	What was the score at the end of the game?	1:0
A_3	true	Which teams got at least one yellow card?	Uruguay,Paraguay
A_4	true	Which team got the most yellow cards?	Uruguay
A_5	true	Which team got the first yellow card?	Paraguay
A_6	true	Which team won the game?	Uruguay
A_7	true	Which team lost the game?	Uruguay

Figure 2.3: Example of questions file from OLP dataset

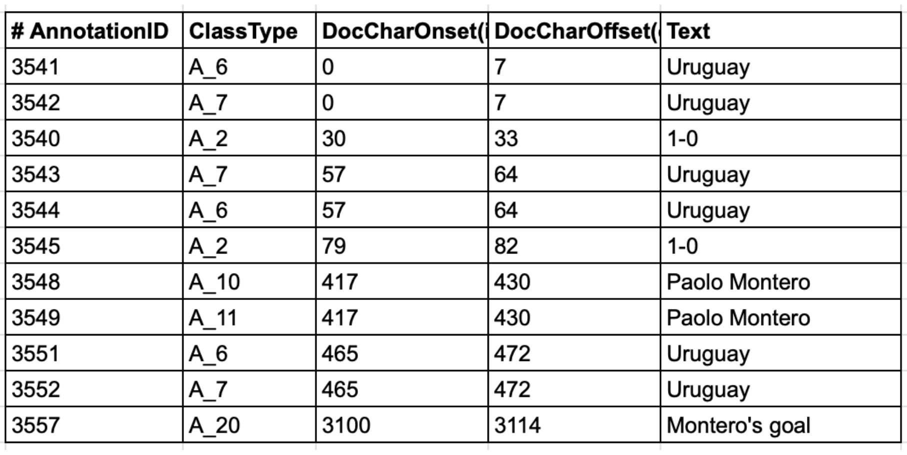


# DocID	SentenceNr	SenTokenPos	DocTokenPos	SenCharOnset(i	SenCharOffset(	DocCharOnset(i	DocCharOffset(	Text
245	0	0	1	0	7	0	7	Uruguay
245	0	1	2	8	13	8	13	break
245	0	2	3	14	18	14	18	lean
245	0	3	4	19	24	19	24	spell
245	0	4	5	25	29	25	29	with
245	0	5	6	30	31	30	31	1
245	0	6	7	31	32	31	32	-
245	0	7	8	32	33	32	33	0
245	0	8	9	34	41	34	41	victory
245	0	9	10	42	46	42	46	over
245	0	10	11	47	55	47	55	Paraguay
245	0	11	12	55	56	55	56	.

Figure 2.4: Example of tokens file from OLP dataset

2.2.4 Conversion

After preprocessing OLP, the next step was its conversion: we wrote another Python script to automatically generate the JSON file with the SQuAD-like format, starting from the new data. Unfortunately, we had to take into account maximum sequence length of text paragraphs: OLP texts, in fact, were usually longer than the ones in SQuAD, as shown in Table 2.3, or the ones used for pre-train the models. While converting the dataset, then, we added an option to automatically split long paragraphs into shorter ones belonging to the same topic. This operation, however, involved also a careful handling of questions and annotations, in order to avoid dangling references. For instance, splitting took into account entire sentences, avoiding cut phrases between paragraphs and questions about the same topic



# AnnotationID	ClassType	DocCharOnset(	DocCharOffset(	Text
3541	A_6	0	7	Uruguay
3542	A_7	0	7	Uruguay
3540	A_2	30	33	1-0
3543	A_7	57	64	Uruguay
3544	A_6	57	64	Uruguay
3545	A_2	79	82	1-0
3548	A_10	417	430	Paolo Montero
3549	A_11	417	430	Paolo Montero
3551	A_6	465	472	Uruguay
3552	A_7	465	472	Uruguay
3557	A_20	3100	3114	Montero's goal

Figure 2.5: Example of annotations file from OLP dataset

Tokens Per Paragraph	0-999	1000-1999	2000-2999	3000-3999	>4000
Number of Paragraphs	101	26	4	1	1

Table 2.3: Distribution of OLP paragraphs by number of tokens

were replicated in every resulting paragraph. We finally wrote also the appropriate logic to split the resulting dataset into a train and dev splits, so that it can be used to fine-tune a Question Answering model, defaulting to a 70%-30% split.

Chapter 3

Transformers

In this chapter there will be a discussion about the Transformer Architecture [41] and its benefits, making it the basic building block of almost all the current *state-of-the-art* models available.

Seq2seq is one of the most frequent tasks in NLP; in fact, as its name suggests, it involves sequences processing. The main examples are the translation of a sentence between two different languages or the summarization of a long text. They used to be addressed through *Recurrent Neural Networks* (RNNs): two networks of this type were stacked one after the other; the first one acted as an encoder while the following one as a decoder. The encoder part in only used to obtain a hidden state representation of the input, called *context vector* which is then fed to the decoder, that used it as additional information to predict target tokens.

Unfortunately, the longer the sequence, the more difficult it was to encode the meaning in a fixed-size vector. In addition, this model is sequential by definition, as each state needs the previous one as input; for this reason, its training can't take advantage of parallel computation, resulting in long training time, often without the expected results. On the other side, a Transformer can execute everything in a parallel fashion because of its design and its Attention mechanism let it process the entire sequence at once, in order to get a better representation of the relationships between tokens.

3.1 Input

Text sequences are fed as input to the model through their embedding representation. In this case, their size is $d_{input} = 512$ but this is just an *hyperparameter* that can be set, as it indicates the longest representable sequence. In particular, a vector of that size will be created for every token appearing in the sentence. Then, assuming a sequence of T tokens, input will be a matrix of size $T \times d_{input}$. Every sentence will be processed as a sort of set of token: this, however, has the drawback of losing information about their order.

3.1.1 Positional Embeddings

Given the parallel nature of this architecture, the model needs a way to encode tokens ordering of the given sentence. The intuition here is that a particular pattern can be instilled in the embeddings: for each token, a positional embedding conforming to that pattern is added to the normal one. Following that, the model will be able to learn the pattern and rebuild the original word ordering.

In this case, the addition is given by a sinusoidal signal with different frequencies and phases based on position (*sin* for even positions and *cos* for odd ones), but this is not the only possible choice: one benefit of the chosen one is the ability of scaling well with sequences of different length. In particular, given a token at position j in the sentence, every element i of its positional embedding vector is defined as:

$$PE(i,j) = \begin{cases} sin(w_k \cdot j), & \text{if } i = 2k \\ cos(w_k \cdot j), & \text{if } i = 2k+1 \end{cases} \quad with \;\; w_k = \frac{1}{10000^{\frac{2k}{d_{emb}}}} \tag{3.1}$$

The resulting positional embedding vector will be:

$$\vec{pe}_j = \begin{bmatrix} sin(w_1 \cdot j) & cos(w_1 \cdot j) & \cdots & sin(w_{\frac{d_{emb}}{2}} \cdot j) & cos(w_{\frac{d_{emb}}{2}} \cdot j) \end{bmatrix} \tag{3.2}$$

3.2 Architecture

A Transformer can be just considered as a collection of blocks where information flow through; a visual representation is shown in Figure 3.1. The two main parts are the *Encoder stack* (left side in the image) and the *Decoder stack* (right side in the image). Both stacks have the same number of layers, which are $N = 6$; the data is fed to the first encoder, processed by the entire Encoder stack and the passed to the Decoder stack which returns the output representation. Ideally, it is a simple, sequential architecture but, as it will be described in the following sections, the reality is quite different, with many parallelized computations. This is due to the fact that each token can flow through the architecture almost independently from the other ones. Furthermore, every encoder in the stack do not share its learned weights with the other encoders.

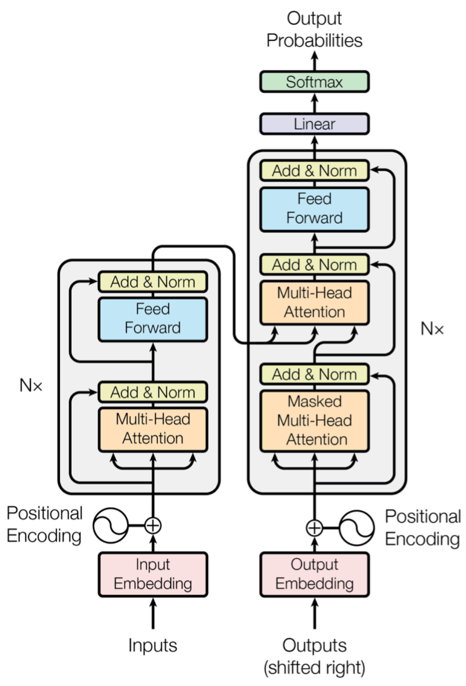


Figure 3.1: Encoder-Decoder representation of a Transformer model architecture[1]

Finally, on top of the architecture there are a Fully Connected layer together with a Softmax one, so that the output from the Decoder stack can be converted into a *logits* vector, where each value refers to a word in the vocabulary. The Softmax transforms this vector in a probability distribution from which the next token can be predicted.

3.2.1 Encoder

Every Encoder is a stack of three types of layers: *Self-Attention*, *Feed-Forward Network* and *Add & Normalization*, with skip connections. Self-Attention layer is the most important mechanism that makes a Transformer a powerful model and it will be described in detail in the next section.

The Feed-Forward Network is basically a stack of two Fully Connected layers with a ReLU [27] activation function in between. Assuming input is of size $d_{input} = d_{emb} = 512$, the first layer will project it to a dimension 4 times bigger, having a inner size of $d_{inner} = 2048$; the second layer, instead, will output it back to the original size. Given $W_i,\ b_i,\ i \in \{1, 2\}$ the weights and biases of the the layers and x the normalized attention scores, this network will apply the following transformation:

$$FFNLayer(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{3.3}$$

Both the Feed-Forward Network and the Self-Attention Layer have a skip connection around them: the purpose of the Add & Norm Layer is to sum the processed output from the previous layer with the same, unprocessed input got from the residual connection, and then, to apply a $LayerNormalization$ [2] on the resulting data, in order to normalize across features and independently from examples. As it has been shown in [12], skip connections are useful for mitigating the *vanishing/exploding gradient problem*, letting huge deep neural networks to be successfully trained.

[1]Image taken from [41]

[2]Image taken from `http://jalammar.github.io/illustrated-transformer/`

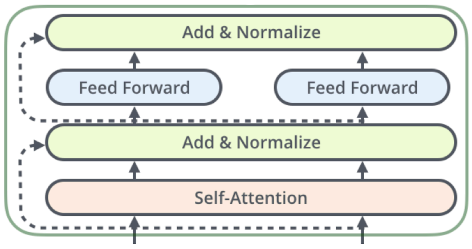


Figure 3.2: Encoder block of Transformer architecture[2]

3.2.2 Decoder

The Decoder is conceptually very similar to the Encoder counterpart: it takes a shifted masked embedding as input together with the Encoder stack output. The latter is given to every Decoder in the stack while the former only to the first one. During the decoding phase, it does not make sense to know words after the current one: for this reason, the embeddings gets shifted and masked with zeros in order to let the model focus on only the previous tokens. This new information is then used by the Attention mechanism to incorporate it with contextual data for the whole sentence, given by the Encoder stack. Finally, the Feed-Forward Network and the Layer Normalization will apply the same transformations as described earlier.

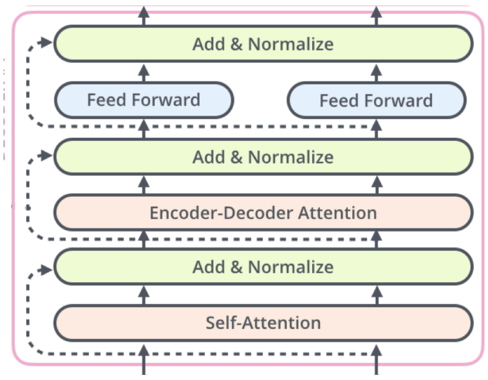


Figure 3.3: Decoder block of Transformer architecture[3]

[3]Image taken from `http://jalammar.github.io/illustrated-transformer/`

3.3 Self-Attention

Generally speaking, *Attention* is a mechanism introduced by [3] which automatically decides which part of text is relevant to achieve a specific task: if Machine Translation is taken into account, this method will choose the most important words to translate the current one. In other words, it considers both input and output. Many different types of Attention have been proposed over the years, focusing in particular on *Self-Attention*. It relates different positions of a single sequence in order to obtain a general representation of the whole sentence. As a consequence, every word is encoded based on all the other words.

This explanation could seem unclear: given the sentence *"The postman delivered a letter, it is on the ground."*, the Self-Attention mechanism will try to associate the word *"it"* with the one it is referring to, *"letter"*. Assuming a certain degree of abstraction, one can say that every word will *query* every other word how much their relationship is relevant and update its *value* consequently, based on every *key* answer.

Before moving on, word embedding will be assumed to change meaning based on the current layer in the stack: in particular, it is the embedding itself when it is fed to the first Encoder but it is also considered as the intermediate output between two Encoders/Decoders.

Given a word embedding Emb of size $d_{emb} = 512$, their query Q, key K and value V representations will be obtained by multiplying it with the corresponding matrix $W_Q \in \mathbb{R}^{d_{emb} \times d_k}$, $W_K \in \mathbb{R}^{d_{emb} \times d_k}$ and $W_V \in \mathbb{R}^{d_{emb} \times d_v}$. These matrices are *learned* by the model during its training phase. The resulting vectors are of size d_k, d_k and d_v, respectively.

$$Q = Emb\ W_Q \qquad K = Emb\ W_K \qquad V = Emb\ W_V \tag{3.4}$$

The following phase consists in calculating the *attention score* which determines how much focus to place on other parts of the input, while encoding a word in a specific position. First, the dot product between Q and K is computed for the currently encoded word and it is scaled by a factor $\sqrt{d_k}$ to stabilize gradients. In the meantime, the same operations are applied to the other words. Next step, in fact, applies a softmax function over all the results, obtaining a probability distribution that sums to 1, used to choose the relevant word to consider. Finally, everything will be multiplied by V and summed to get the score for the current word. As it can be seen, every word seems to follow a precise and almost independent path, which can be parallelized.

3.3.1 Matrix Generalization

Those calculations can be easily generalized: instead of feeding a single word embedding as input, an embedding matrix can be used, as stated in Section 3.1; this allows the processing of an entire sentence at the same time. As a consequence, all the previously defined vectors will become matrices and the steps can be summarized by the following equation:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{3.5}$$

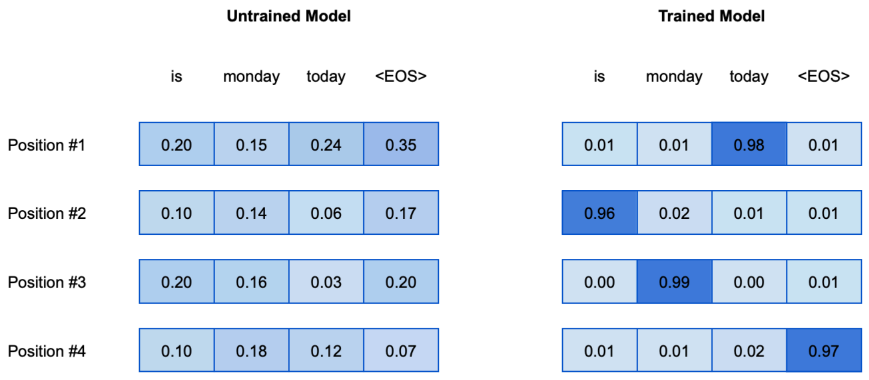
3.3.2 Multi-Head Attention

In order to make the model even more powerful, Self-Attention in every Encoder and Decoder is replicated M times. The advantage is that the model can increase its ability of focusing on different positions, as multiple different sets of $Q/K/V$ matrices are considered, called *heads*. The Attention matrix is now the concatenation of the resulting matrices from all the heads available. Now, however, a new learned matrix is needed to condense all the information in a new one with the proper size expected from the Feed-Forward Network. In particular, that matrix is $W_O \in \mathbb{R}^{Md_v \times d_{emb}}$.

One thing worth noticing is the relation between the embedding size and the number of heads: d_{emb} must be divisible by M, as $Q/K/V$ dimensionality depends on that result. Following that, the original Transformer employed $M = 8$ and $d_k = d_v = 64$.

3.4 Training Intuition

As many Deep Neural Networks, the intuition the training is based upon is quite simple: given some input, the network executes the *forward-pass*, it compares the output with the expected one (*ground truth*), calculates the gradients and executes the *backward-pass* to update all the weights and biases with these gradients. When handling text, this is applied to its embedding representation; in particular, gradients are computed using a loss function (often Cross-Entropy Loss) between model output and the one-hot encoding of the expected word. In general, every output represents the Softmax result for that word. Gradients will then be used to shift that probability distribution in order to make it similar to the expected one, as shown in the following figures.



Untrained Model

	is	monday	today	<EOS>
Position #1	0.20	0.15	0.24	0.35
Position #2	0.10	0.14	0.06	0.17
Position #3	0.20	0.16	0.03	0.20
Position #4	0.10	0.18	0.12	0.07

Trained Model

	is	monday	today	<EOS>
Position #1	0.01	0.01	0.98	0.01
Position #2	0.96	0.02	0.01	0.01
Position #3	0.00	0.99	0.00	0.01
Position #4	0.01	0.01	0.02	0.97

Figure 3.4: Starting from the untrained model on the left, training will update weights in order to obtain a model that produces the output on the right, when predicting the sentence *"Today is Monday"*

3.5 State Of The Art

In this section, an overview about the state-of-the-art models will be provided, with a particular focus on models thought for addressing the Question

Answering problem on the SQuAD dataset.

3.5.1 BERT

Introduced in late 2018, *Bidirectional Encoder Representations from Transformers* (BERT) marked the start of a new era for many NLP tasks. The model takes into account two different phases: *pre-training* and *fine-tuning.* During pre-training, the model is trained in order to make it understand bidirectional representations of text, by conditioning all layers to look at both left and right contexts. On the other side, the fine-tuning phase makes the model specific for a particular downstream task, by just slightly training all parameters.

BERT architecture is based on multiple Transformer Encoder blocks, as they were described in Section 3.2.1, and it has been developed in two variants, in which the number of layers, attention heads and the hidden size change, as shown in Table 3.1. One change with respect to the original Transformer architecture is that ReLU activation function across the entire model has been replaced with GELU [13].

An important characteristic of BERT is its versatility because the architecture has been designed in a generalized way, in order to tackle multiple tasks: both single sentence and pairs can be used (a pair is packed into a single sequence, separated by the special `[SEP]` token). Every sequence has the special `[CLS]` token as the first one. Input representation consists of the sum between token, positional (as for Transformers in Section 3.1.1) and sequence embeddings. This last type is the particular way BERT uses to describe whether a token belongs to the first or second sentence. On the contrary, output representation is given by the final hidden vectors: the one belonging to the `[CLS]` tag is denoted as $C \in \mathbb{R}^H$, while given the i-th input token, its output is given by $T_i \in \mathbb{R}^H$.

Pre-training is based on two tasks, *Masked Language Modelling* (MLM) and *Next Sentence Prediction* (NSP), and the training corpus is made up of BookCorpus (800M words) and part of English Wikipedia (2500M words).

Model	Layers L	Hidden Size H	Attention Heads A	Parameters
$BERT_{BASE}$	12	768	12	110M
$BERT_{LARGE}$	24	1024	16	340M

Table 3.1: Differences in BERT architecture variants.

Task-specific architecture modifications consist in adding just one linear layer on top of BERT, which will use the output representations as input: in case of token-level tasks, such as Question Answering, all token representations T_i are fed, while for classification tasks, such as sentiment analysis, just the `[CLS]` representation C is fed.

Masked Language Modelling

Bidirectional conditioning is made possible because some input tokens are randomly masked and then, the language model is asked to predict them. In particular, a softmax operation over the entire vocabulary is applied to the final hidden representation of the masked token. One consequence of this behaviour is a mismatch between pre-training and fine-tuning data, as masked tokens are not available while fine-tuning. For this reason, masking operations are slightly changed taking into account the following assumptions and its representation T_i will be used for prediction with cross entropy loss:

- Only 15% of token positions are masked
- After choosing the i-th token, it is replaced:
  - with `[MASK]` 80% of the time, e.g. `the post office is closed` $\rightarrow$ `the post [MASK] is closed`
  - with a random token 10% of the time, e.g. `the post office is closed` $\rightarrow$ `the post beach is closed`
  - leave the i-th token unchanged the remaining 10% of the time

Next Sentence Prediction

This task is particularly important for problems like Question Answering, as it lets the model understand the relation between two different sentences. During pre-training, sentence pairs (A, B) are chosen in a way that 50% of the time B is the actual sentence following A (labelled as `IsNext`), otherwise B is a random one from the corpus (labelled as `NotNext`). The binary classifier will then use `[CLS]` output representation C to predict the appropriate label.

```
[CLS] the post [MASK] is closed [SEP] I can not send the [MASK] [SEP]
Label = IsNext

[CLS] the post office is [MASK] [SEP] bringing the [MASK] together [SEP]
Label = NotNext
```

Code 3.1: Next Sentence Prediction binary classification example on masked sentences

WordPiece

WordPiece Model [34], [43] is a tokenization algorithm very similar to the *Byte-Pair Encoding* (BPE), whose idea was at the core of a compression algorithm. This model will create a language model with a vocabulary of a predefined size, starting by treating every char as a unicode element of the vocabulary; the model is trained on it and new subwords are added by merging the ones which increase the likelihood on the training data the most. This process continues until the desired vocabulary size is reached. Below there is an example of tokenized text using the WordPiece model; in particular, one can notice how suffixes and prefixes are treated.

Out-of-Vocabulary (OOV) words can not happen in this scenario as words not present in the vocabulary will be split into subwords, while rare words will be brought back to the case of being split.

Even though this model is quite famous and used by BERT [8], the real implementation is closed-source and it is used internally at Google. Many available implementations are not guaranteed to be as exactly as the original one.

```
INPUT: Lorem ipsum dolor sit amet.
TOKENIZED: 'lore', '##m', 'ip', '##sum', 'do', '##lor', 'sit', 'am', '##et', '.'
```

Code 3.2: Example of sentence tokenization using WordPiece

Fine-Tuning on SQuAD

SQuAD paragraphs and questions are modelled as sentence pairs, such that the input representation is `[CLS] question tokens [SEP] paragraph tokens [SEP]`, with a maximum sequence length of 512. A start vector $S \in \mathbb{R}^H$ and an end vector $E \in \mathbb{R}^H$ are introduced, in order to compute start/end word probabilities. Given a word i, its probability of being a start word is computed through the Equation 3.6. In other words, the dot product between the start vector S and output representation T_i is computed before applying a softmax over all words in the paragraph. The same happens for the end word.

$$Ps_i = \frac{e^{S \cdot T_i}}{\sum_j e^{S \cdot T_j}} \tag{3.6}$$

The predicted answer span is the one who maximizes the span score, defined as $S \cdot T_i + E \cdot T_j$, $j \geq i$, where i and j are token positions in the paragraph. The train objective is then the sum of log-likelihoods of the correct start and end positions.

The introduction of SQuAD v2 with impossible answers requires some adjustments on how answer spans are predicted: for questions without an answer, the span is defined on the `[CLS]` token.

$$\begin{gathered} \hat{s_{i,j}} = max_{j \geq i} S \cdot T_i + E \cdot T_j \\ s_{imp} = S \cdot C + E \cdot C \end{gathered} \tag{3.7}$$

In addition to the most probable answer span from text, an impossible span s_{imp} in calculated, by computing the dot product with the `[CLS]` output representation C. The final answer span prediction is given when $\hat{s_{i,j}} >$

$s_{imp} + \tau$, where τ is a threshold selected on the dev dataset, in order to maximize F1 score.

3.5.2 RoBERTa

Almost an year after the introduction of BERT, Facebook AI and University of Washington released RoBERTa, which stands for *Robustly Optimized BERT Approach* [23]. As its names suggests, their work focuses on a thorough evaluation of hyperparameters and design choices, proving that BERT-like models are very sensitive to these settings; for instance, they shows training is very sensitive to the Adam [17] epsilon term.

Experimental setups they conducted involved the usage of a larger pre-training dataset and a different training procedure: static vs. dynamic token masking, Next Sentence Prediction objective removal, different input format, larger batches and a BPE-based subtoken vocabulary. After an extensive ablation study, the final configuration uses *dynamic masking* (tokens are dynamically masked when a sequence is fed to the model), large mini-batch, a byte-level BPE vocabulary of about 50K units, and *Full-Sentences* input format without NSP (input consists of full sentences from one or more documents). Furthermore, they showed an increased pre-training dataset and a longer training time further improve performances, without overfitting risk on downstream tasks; as it can be seen in the results comparison on SQuAD datasets in Section 2.2.1, the model sets a new state-of-the-art score. In addition, unanswerable questions from SQuAD v2 dataset are first selected by a binary classifier, jointly trained while fine-tuning.

Chapter 4

ALBERT

Since the introduction of BERT [8], the world of NLP tasks and *Natural Language Understanding* faced a dramatically improvement in research and end-use cases. This is why it is considered one of the milestones of the latest years. Continuing on this trend, Google Research released ALBERT [21], a new model that quickly gained the attention of NLP community because of its improvements, establishing new state-of-the-art performances on many tasks. In particular, ALBERT stands for *"A Lite BERT"*: most of the work behind this model is focused on a more efficient usage of the Transformer architecture, as it will be described in this chapter. In addition, this model will be used as the base for a custom one, developed for the purpose of this thesis.

4.1 Yet Another Model. Why?

Before going on explaining ALBERT, a question should arise: *why do we all need another NLP model? Efficiency.* As shown earlier, Transformer-based models are all particularly well-performing on reading comprehension and classification tasks but their weak point is definitely their size. In fact, number of parameters usually ranges from millions to even billions, as it happens for the MegatronLM [36] model developed by NVIDIA (8.3 billion).

Having a large network is very important for achieving state-of-the-art performance. At the contrary, having larger and larger models can not be the answer for keeping improving performances, as memory limitations are quite restrictive. Even considering just $BERT_{BASE}$, batch size and maximum sequence length are two of the most relevant hyperparameters with respect to memory usage; for this reason, a quite good GPU, or even better a TPU, is mandatory to carry out a training with valid results. At the same time, inference is also affected by this phenomenon, even if in a lower magnitude. As a consequence, it is even more important to start investigating techniques to lower the number of parameters and memory usage, without harming performance.

From this point of view, ALBERT proposes two techniques, *Factorized embedding parametrization* and *Cross-layer parameter sharing*.

4.2 Factorized Embedding Parametrization

In models such as BERT, all input and internal representations depend on the hidden size H and the vocabulary size V. For instance, the embedding matrix has size $V \times H$.

Recalling that WordPiece embeddings are meant to learn *context-independent* representation, on the contrary, embeddings from hidden layers are meant to learn *context-dependent* representations. For this reason, increasing the hidden size H allows the model to have an higher representation capacity to learn these context-dependent information. As a consequence, the embedding matrix will become bigger and bigger, resulting in million or billion of parameters which are sparsely updated during training.

Given a new parameter E which represents the vocabulary embedding size, the intuition is to untie V from H by decomposing the embedding matrix into two smaller matrices, depending on E. Therefore, embedding parameters are reduced from $O(V \times H)$ to $O(V \times E + E \times H)$, making the reduction even more pronounced when $H \gg E$, as shown in Table 4.1. From a practical point of view, this matrix decomposition involves the projection of word embeddings in an intermediate space of size E, instead of directly into the hidden space of size H, resulting in a more efficient usage of parameters. In ALBERT, the default vocabulary embedding size is $E = 128$, taken into account model performance and size.

Hidden Size H	**BERT**	**ALBERT**	**Reduction Factor**
128	3.84M	3.85M	0.99
384	11.52M	3.89M	2.96
768	23.04M	3.94M	5.85
1024	30.72M	3.97M	7.74
2048	61.44M	4.1M	14.98
4096	122.88M	4.36M	28.16

Table 4.1: Factorization effect on embedding parameters supposing a vocabulary size $V = 30000$ and vocabulary embedding size $E = 128$

4.3 Cross-Layer Parameter Sharing

In addition to the Factorized Embedding Parametrization, the other important advancement in ALBERT is Cross-Layer Parameter Sharing. It is a parameter reduction technique which consists in avoiding multiple different parameters by sharing most of them between layers.

Generally speaking, given the number of layers L, sharing takes place by dividing their weights in N groups, each of them sized M, according to the relation $L = N \times M$; only weights belonging to the same group are shared. Furthermore, four different settings have been investigated: *all-shared* (ALBERT setting), *shared-FFN* (only Feed-Forward Network weights are shared), *shared-attention* (only attention heads are shared), *not-shared* (BERT setting), under the assumption of using an *ALBERT-base* configuration with vocabulary embedding size $E = 128, 768$.

The effect of this technique with respect to the number of parameters is shown in Table 4.2. As before, default vocabulary embedding size is $E = 128$ and $E = 768$ has been used for comparison only. Before choosing the default sharing strategy, the effects on downstream tasks has been investigated too and the results are shown in Table 4.3.

Model	Setting	Parameters	Reduction Factor
ALBERT Base E=768	all-shared	31M	1.0
	shared-attention	83M	2.68
	shared-FFN	57M	1.84
	not-shared	108M	3.48
ALBERT Base E=128	all-shared	12M	1.0
	shared-attention	64M	5.33
	shared-FFN	38M	3.17
	not-shared	89M	7.42

Table 4.2: Cross-layer parameter sharing reduction effect on total number of parameters

According to this figures, sharing Feed-Forward Network weights can be considered the worst case scenario, as performance drops under both configurations. In addition, also the *all-shared* setting results in a little drop on average. Taking into account only the scenario where $E = 128$, sharing attention weights only seems to be the best case, even though it is just 0.3 points above the *all-shared* strategy. For this reason, the default setting chosen for ALBERT is the *all-shared* one because the huge parameter reduction is worth a slight decrease in performance. From a practical point of view, this is implemented by using $N = 1$ groups of size $M = L$.

Model	Setting	SQuAD v1.1		SQuAD v2.0		Avg
		F1	EM	F1	EM	
ALBERT Base E=768	all-shared	88.6	81.5	79.2	76.6	81.47
	shared-attention	89.9	82.7	80.0	77.2	82.45
	shared-FFN	89.2	82.1	78.2	75.4	81.22
	not-shared	90.4	83.2	80.4	77.6	**82.9**
ALBERT Base E=128	all-shared	89.3	82.3	80.0	77.1	82.17
	shared-attention	89.9	82.8	80.7	77.9	**82.82**
	shared-FFN	88.9	81.6	78.6	75.6	81.17
	not-shared	89.9	82.8	80.3	77.3	82.57

Table 4.3: Cross-layer parameter sharing effect on SQuAD task

4.4 Sentence Order Prediction

Following many studies, such as [23], Next Sentence Prediction loss has been proven to be often unreliable, thus the decision of removing it. It was designed to merge *topic* and *coherence* predictions in a single task but, apparently, predicting coherence is much more complicated than predicting topic shifts. For this reason, NSP was unwittingly focusing on topics during training, as negative examples were created by sampling two segments from different documents.

ALBERT, instead, models a Sentence Order Prediction loss focusing just on coherence. In particular, the only change with respect to NSP is how negative examples are created: as it can be seen from Code 4.1, given a positive example consisting of two consecutive segments of text, the negative one is obtained by swapping their order. Even though the change is particularly simple, it forces the model to focus on discourse-level coherence properties.

```
Positive Example:
[CLS] the post [MASK] is closed [SEP] and I can not send the [MASK] [SEP]


```

```
Negative Example:
[CLS] and I can not send the [MASK] [SEP] the post [MASK] is closed [SEP]
```

Code 4.1: Sentence Order Prediction positive and negative examples for learning coherence

4.5 Minor Changes

4.5.1 Masked Language Modelling

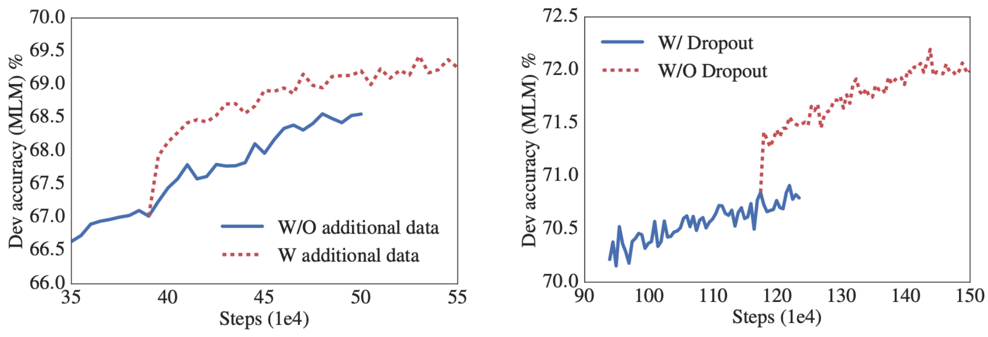
Masked Language Modelling objective has been slightly updated with respect to the one used in BERT, by employing *n-gram masking*. Instead of masking single tokens in a segment, up to n consecutive words are selected randomly, with $n = 3$.

4.5.2 Dropout and Data Augmentation

In order to keep the comparison between ALBERT and other BERT-like models as meaningful as possible, pre-training data was the same: a concatenation of BookCorpus and part of English Wikipedia. However, ALBERT was also pre-trained with additional data, the same used in [23] and [46]. According to Figure 4.1(a), MLM accuracy improves with the augmented pre-training dataset, even though performance on SQuAD task are slightly worse. Since no signs of overfitting were observed during pre-training, dropout effects were also investigated, in order to increase model representative capacity. As shown in Figure 4.1(b), by removing dropout, MLM achieves a significant boost in accuracy, together with downstream tasks, even if in a lower measure.

4.5.3 SentencePiece

SentencePiece [20] is a new tokenizer/detokenizer developed by Google specifically designed for neural-based text processing. In particular, it per-



(a) Additional data (b) Dropout

Figure 4.1: Dropout and additional data effects on MLM accuracy while pre-training. Images taken from [21]

	SQuAD v1.1		**SQuAD v2.0**	
	F1	EM	F1	EM
No additional data *	**89.3**	**82.3**	**80.0**	**77.1**
With additional data *	88.8	81.7	79.1	76.3
With dropout †	94.7	89.2	89.6	86.9
Without dropout †	**94.8**	**89.5**	**89.9**	**87.2**

Table 4.4: Dropout and additional pre-training data effects on ALBERT for downstream tasks. * refers to *ALBERT-base* configuration; † refers to *ALBERT-xxlarge* configuration

forms a language-independent subword processing. Because of these characteristics, it aims at becoming a new open-source standard in NLP community: it removes the burden of relying on hand-crafted rules for text segmentation, mostly for non-whitespace-separated languages such as Chinese or Japanese, and it also allows direct training from raw sentences, as it automatically handles vocabulary mapping. Besides custom normalization and an efficient implementation of segmentation, the main contributions are two: *lossless tokenization* and *self-contained model*. The former encodes all the necessary information for the detokenization in the tokenized output, for instance solving whitespace ambiguity problems, while the latter makes experiments easily reproducible, as the model contains the vocabulary and segmentation/normalization parameters.

Below an example of tokenized text using SentencePiece is shown.

```
INPUT: Lorem ipsum dolor sit amet.
TOKENIZED: '_lore', 'm', '_i', 'ps', 'um', '_do', 'lor', '_sit', '_a', 'met', '.'
```

Code 4.2: Example of sentence tokenization using SentencePiece

4.6 Configurations and Results

All ALBERT configurations investigated are reported in Table 4.5. As it can be seen, even considering the *xxlarge* configuration, which has a Hidden Size H 4 times bigger than the one used in $BERT_{LARGE}$, the difference in number of parameters is quite pronounced. In addition, big configurations, such as *large*, *xlarge* and *xxlarge*, have been explicitly created to explore the effects of network depth and width, in order to verify whether it is worthy to keep increasing model size to obtain better performance on downstream tasks. In particular, results show this assumption doesn't hold as increasing model capacity, for instance by using a larger amount of Transformer layers, often results in a smaller performance gain; the same applies to Hidden Size H. Furthermore, the *xxlarge* configuration uses only 12 Transformer layers, as it would have been computationally more expensive to employ 24 layers, without a significant increase in performance.

Config	**Parameters**	**Layers** L	**Hidden** H	**Embedding** E
base	12M	12	768	128
large	18M	24	1024	128
xlarge	60M	24	2048	128
xxlarge	235M	12	4096	128

Table 4.5: Overview of different ALBERT configurations

The ALBERT configuration taken into account for an evaluation on SQuAD considers the following settings: *xxlarge* configuration, combined MLM and SOP losses, no dropout, additional pre-training data and additional training time (1M and 1.5M steps).

Even though $ALBERT\ xxlarge$ has fewer parameters than $BERT_{LARGE}$, it is computationally more expensive to be trained, but it obtains significantly better results. Due to hardware restrictions, the work explored in the following chapter considers the $ALBERT\ base$ configuration only, unless otherwise specified. For a complete overview of the experimental results about ALBERT, its configurations and investigations, please refer to [21].

Model	**SQuAD v1.1**		**SQuAD v2.0**	
	F1	EM	F1	EM
$BERT_{LARGE}$	90.9	84.1	81.8	79.0
XLNet	94.5	89.0	88.8	86.1
RoBERTa	94.6	88.9	89.4	86.5
ALBERT (1M)	94.8	89.2	89.9	87.2
ALBERT (1.5M)	**94.8**	**89.3**	**90.2**	**87.4**

Table 4.6: Comparison on SQuAD dataset assuming the best performing ALBERT configuration

Chapter 5

ALBERT Improvements

This chapter includes a detailed description of all the work done for this thesis, with the purpose of investigating and improving ALBERT performance on the Question Answering task.

5.1 Settings

Before describing efforts and ideas for this project, it necessary to introduce the environment used. The majority of work consisted in writing Python scripts handling trainings and results collection, but also editing existing ones to adjust models.

As of today, there are two main frameworks for Deep Learning and Neural Networks development and deployment available to the Python community which are Tensorflow [24] and PyTorch [29]. Although ALBERT official code released by Google Research is Tensorflow-based, Pytorch has been preferred due to its simplicity and previous experience; in particular, version 1.3.1 has been employed. Other tools includes Tensorboard for training visualization, regex and Numpy.

In order to avoid a complete porting of ALBERT from Tensorflow to PyTorch, Hugging's Face Transformers [42] library has been used: it is a collection of all state-of-the-art NLP models completely written in PyTorch, continuously updated and community-driven.

Most of the research and trainings have been carried out using the GPU Cluster at CITEC, Bielefeld University, mainly through 2x NVIDIA P100 16GB GPUs.

5.2 Main Ideas

Due to the limited resources available, a complete pre-training of ALBERT was unfeasible: in fact, it took more than 32 hours employing 64 to 512 Google Cloud TPUs V3, depending on the configuration. As a consequence, all kind of improvements regarding the real representation capacity of this models have been excluded from the beginning; examples of this kind of changes could have been an updated loss function involving an additional/updated objective task, as it has been done with SOP, or data augmentation. The challenge then, consisted of finding reasonable but, hopefully, effective strategies to be applied while fine-tuning the model, after loading pre-trained weights.

5.2.1 Binary Classifier

As the main focus was on Question Answering and, in particular, on the SQuAD v2.0 dataset, one of the first ideas involved a better handling of unanswerable questions. Instead of letting the model predict them by returning an answer span on the `[CLS]` token, the idea was to add a *binary classifier* to predict question type and then, the classic answer span extractor. For Question Answering task, `[CLS]` token is not used, as it is an invalid position with respect to the text from which the answer has to be found; on the contrary, it is employed for every kind of classification task, such as Sequence Classification, because it contains a representation of the entire input. Consequently, the immediate extension was to implement the classifier on top of the `[CLS]` output while keeping the span extractor on the rest.

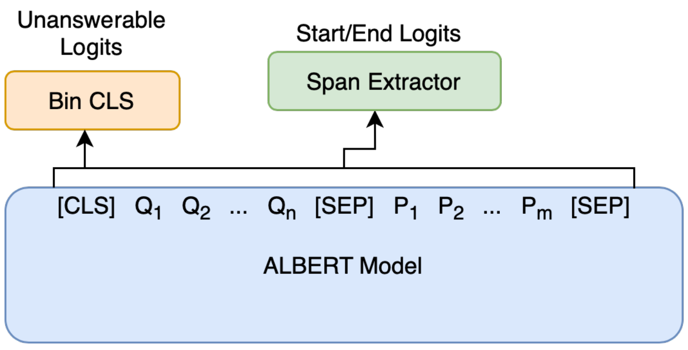


Figure 5.1: Schema for the binary classifier idea

Joint Loss

In order to let the model leverage the binary classifier output, it had to be trained on both answerable and unanswerable questions. For this reason, the usual loss function for the Question Answering task has been modified, as shown in the Equation 5.1, to take into account the classifier loss and jointly learn them.

$$\mathcal{L}oss = \mathcal{L}_{SPAN} + \alpha\mathcal{L}_{CLS} \tag{5.1}$$

In particular, $\mathcal{L}_{SPAN}$ is the span extractor loss, e.g. Cross Entropy Loss on start and end positions, while $\mathcal{L}_{CLS}$ is a Binary Cross Entropy Loss, applied on `[CLS]` output logits; α is a scaling factor applied to investigate the contribution of the classifier on the whole task.

5.2.2 Hidden States Concatenation

Another idea concerned Transformer layers output: to which extent is every output relevant for the final model decision? Even though every Transformer layer takes into account the previous output, the Question Answering system only considers the final Transformer output. The investigation in this case consisted in concatenating some of the last layers output, in order to understand whether intermediate representations could be effective; from another point of view, this operation could be considered almost as a *skip*

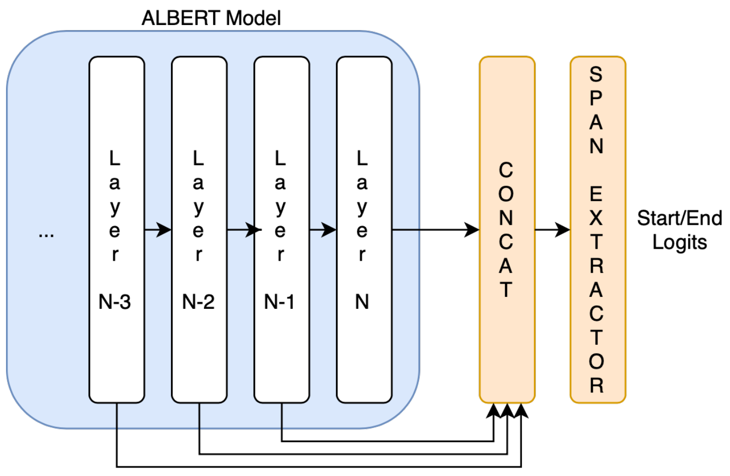


Figure 5.2: Schema for the $CL = 4$ hidden states concatenation

connection but for Encoder outputs.

However, the majority of work was on the Binary Classifier idea, thus less effort has been put into this one: given CL as the number of concatenated outputs, only few trainings were conducted with $CL = 2, 4$.

5.3 Implementation

The principal step before implementing the aforementioned ideas was an analysis of both Tensorflow-based official code and PyTorch implementation from Hugging Face, in order to get a proper understanding of their design choices. During this work, in fact, some mismatches in the configurations were found between the published ALBERT paper and the ones used by both codebases; for instance, dropout rates in Transformer sub layers were wrong, as Google confirmed to me on a GitHub issue[1], or the usage of a different GELU approximation. The updated configuration can be seen in Appendix B.2.

The span extractor is implemented using a single Fully Connected layer

[1]`https://github.com/google-research/ALBERT/issues/23`

which reduces input last dimension from Hidden Size H to 2, for Start and End logits; the same implementation as in Section 3.5.1.

Unanswerable Questions Classifier

By default, Hugging Face's ALBERT model already returns a *pooled output* with shape $(BatchSize, H)$, which is a representation of the entire input, processed and compressed into the `[CLS]` token. In other words, according to BERT implementation, it contains all the relevant information to perform a classification task. Since the aim is to get a boolean information whether the question can be answered or not, the pooled output is projected to this desired value through a Fully Connected layer which changes the shape from $(BatchSize, H)$ to $(BatchSize, 1)$; a dropout probability of 0.1 is applied to the pooled output before feeding it to this layer.

As stated earlier, a Binary Cross Entropy Loss is computed during training between the `is_impossible` data from the ground truth and the logits computed by the classifier. In addition to editing the model, the fine-tuning script has been updated too, in order to get appropriate metrics with respect to the classifier. During the prediction phase, instead, accuracy is evaluated in the following way: output logits are converted back to the range $[0, 1]$ by applying the Sigmoid function and every value lower than 0.5 has been approximated to 0, otherwise 1. For every batch, the accuracy is then the ratio between correct predictions over the total and the final classifier accuracy is the mean of all batch accuracies.

A complete overview of these implementations is shown in Appendix B.4.

Output Concatenation

In case of ALBERT Base configuration, every Transformer returns an output of size $H = 768$ and the final span extractor expects an input of that size. In order to consider multiple different output together, they have been concatenated along last dimension: for instance, given CL as the number of concatenated outputs and an output shape $(BatchSize, SeqLen, H)$, the shape of the resulting tensor will be $(BatchSize, SeqLen, CL \times H)$. Due to this change, another intermediate representation is needed in order to make the last dimension of the proper size. This is done by adding a Fully Connected layer between ALBERT model and the span extractor. The disadvantage of this approach, however, is the increasing number of learned parameters: for instance, assuming concatenation of the last $CL = 4$ hidden states, the new intermediate layer has to reduce the dimension from $H \times CL = 768 \times 4 = 3072$ back to 768. An operation like this introduces about 2.3M of parameters.

A complete overview of these implementations is shown in Appendix B.5.

5.4 Training Pipeline

From a broad point a view, the training pipeline that has been followed to carry out results and evaluations can be summarized with the following list:

- Model creation and selection
- Hyperparameters investigation depending on available resources
- Fine-tuning execution on SQuAD v2.0 dataset
- Evaluation during and after fine-tuning through metrics and Tensorboard data
- Choice of most promising models to use them as starting point for fine-tuning and evaluation on OLP dataset

Due to limited hardware resources, the Base configuration was the only feasible one that made extensive experiments possible, particularly with respect to some hyperparameters, such as *batch size* and *maximum sequence length*. In fact, the mean GPU memory usage was about 15 GB each.

As it will be shown in the following results, some hyperparameters were fixed in order to guarantee a fair comparison between different experiments, while some others were investigated and changed accordingly. For instance, *maximum sequence length* has been fixed to 512 to make the improved model consistent with the pre-training.

A complete description of the hyperparameters changed and their values can be seen in Appendix A, in addition to some graphs provided by Tensorboard.

5.5 Results

Later in this section, a complete overview of model configurations and results will be provided, both on SQuAD and OLP datasets, with an appropriate discussion.

5.5.1 Configurations

Table 5.1 shows the investigated configurations that have been fine-tuned on SQuAD v2.0. Unless otherwise specified, all of them used a Batch Size of 32 while fine-tuning and the initial pre-trained weights from ALBERT were the version `albert-base-v2`[2]. Furthermore, for an additional analysis, few ALBERT large configurations were used, as shown in Table 5.2 but hyperparameters have been changed in order to successfully proceed with the fine-tuning; unfortunately, this settings can be difficult to compare with the other ones. In this case, the initial weights were changed accordingly to `albert-large-v2`[3].

A side note regarding FP16: even though half-precision could be useful to speed up trainings and optimize GPU memory usage, this option has only been employed once because of lack of these benefits, as shown in Table 5.3. This is mainly due to the fact that specific GPU architectures are necessary, such as NVIDIA Volta GPUs.

[2]`https://tfhub.dev/google/albert_base/2`

[3]`https://tfhub.dev/google/albert_large/2`

Config	**BinCLS**	**Concat**	**MSL**	**LR**	**TE**	**WP**	**FP16**
base_binCls	✓	✗	512	3e-5	3.0	0.2	✗
base_binCls_384seq	✓	✗	384	3e-5	3.0	0.2	✗
base_binCls_4epochs	✓	✗	512	3e-5	4.0	0.2	✗
base_binCls_fp16	✓	✗	512	3e-5	3.0	0.2	✓
base_binCls_highLR	✓	✗	512	5e-5	3.0	0.2	✗
base_concat2	✗	✓	512	3e-5	3.0	0.1	✗
base_concat2_highLR	✗	✓	512	5e-5	3.0	0.1	✗
base_concat4	✗	✓	512	3e-5	3.0	0.1	✗
base_concat4_highLR	✗	✓	512	5e-5	3.0	0.1	✗

Table 5.1: Main Base configuration investigated while improving ALBERT. BinCLS: binary classifier for unanswerable questions. Concat: last hidden states concatenation. MSL: Max sequence length. LR: Learning rate. TE: Train epochs. WP: Warmup steps proportion. FP16: Half-precision.

Config	**BinCLS**	**Concat**	**BS**	**MSL**	**LR**	**TE**	**WP**
large_binCls	✓	✗	12	512	3e-5	3.0	0.2
large_binCls_highLR	✓	✗	12	512	5e-5	3.0	0.2
large_binCls_hLR_2ep	✓	✗	12	512	5e-5	2.0	0.2

Table 5.2: An overview of few Large configurations. BS: Batch size.

All models employed the `uncased` vocabulary and the optimizer was AdamW, which is a slightly modified version of Adam allowing the usage of warmup steps; in particular, that number was a percentage of total steps (according to warmup proportion hyperparameter) during which the learning rate was increased linearly until the maximum value. In case of base configurations, one training epoch took approximately 1 hour, which increased to 4 hours for large-based ones.

The following tables present results on SQuAD v2.0 dataset by comparing them with the appropriate baseline; moreover, every table shows an overview taking into account each improvement separately, in order to understand their contribution. *Best F1* and *Best EM* scores refer to F1 and EM computed after finding the best null threshold on the dev set, as stated in the official SQuAD evaluation script. This threshold is computed according to the predictions, in order to choose the best value discriminating unanswerable predictions from answerable ones.

For a complete description about these two metrics, please refer to Section 2.1.

Binary Classifier for Unanswerable Questions

Config	**Best**		**HasAns**		**NoAns**		**CLS**
	F1	EM	F1	EM	F1	EM	**Acc.**
ALBERT Base	80.0	77.1	-	-	-	-	-
base_binCls	82.63	79.38	83.65	76.64	**79.98**	**79.98**	**85.67**
base_binCls_384seq	**82.75**	**79.46**	**83.92**	**76.94**	79.8	79.8	85.04
base_binCls_4epochs	81.91	78.72	83.28	76.45	78.86	78.86	84.91
base_binCls_fp16	81.88	78.67	83.24	76.32	79.56	79.56	85.35
base_binCls_highLR	81.47	78.34	82.75	75.67	79.33	79.33	84.84

Table 5.3: Results on SQuAD v2.0 dataset for ALBERT base configurations with binary classifier for unanswerable questions

As it can be seen from these results, using a higher learning rate or training for an additional epoch instead of the usual 3 hurts performances as both settings will likely increase model overfitting chances.

base_binCls proves to be the best performing configuration with respect to the binary classifier accuracy and performances on unanswerable questions. Overall, *base_binCls_384seq* seems to perform better than *base_binCls* on answerable questions, leading to better F1/EM scores; taking a closer look, however, the difference between these two configurations is not so pronounced so the *base_binCls* has been preferred because of its better contribution in tackling unanswerable questions.

In addition to these figures, different scaling factor values have been experimented, in order to understand the best weight to give to the binary classifier loss, according to the joint loss in Section 5.2.1.

Even though ALBERT was pre-trained with longer sequences, *base_binCls_384seq* performs slightly better than *base_binCls*, and even more with respect to the plain ALBERT base configuration. Since the difference between these two best performing settings is not so marked, *base_binCls* will be used in subsequent experiments.

Table 5.4 shows the effect of different scaling factors on binary classifier loss; as it can be seen, having a 2× factor results in higher scores, but still lower than the plain *base_binCls* configuration shown in Table 5.3, which uses the same scale of the span extractor loss ($\alpha = 0.5$).

Config	**Factor**	**Best**		**HasAns**		**NoAns**		**CLS**
	α	F1	EM	F1	EM	F1	EM	**Acc.**
ALBERT Base	-	80.0	77.1	-	-	-	-	-
base_binCls	1.0	82.27	**79.17**	83.29	76.32	79.53	79.53	85.28
base_binCls	2.0	**82.4**	79.03	**83.5**	**76.33**	**79.61**	**79.61**	**85.57**
base_binCls	6.0	81.47	78.03	82.49	75.19	79.41	79.41	85.24

Table 5.4: Results on SQuAD v2.0 dataset for ALBERT base configurations after applying a scaling factor α to binary classifier loss

Finally, a large-based configuration allows to reach better performance, even considering the lower batch size used during fine-tuning. Generally speaking, the same considerations on learning rate applies to these settings too; however, these configurations require more training time and memory for about a +1.5 F1 improvement.

Looking at the figures regarding the binary classifier accuracy, it seems to be difficult to obtain a value greater than 85%; the objective of performing binary classification, in fact, seems to be too easy with respect to the span

Config	**Best**		**HasAns**		**NoAns**		**CLS**
	F1	EM	F1	EM	F1	EM	**Acc.**
ALBERT Large	82.3	79.4	-	-	-	-	-
large_binCls	**83.94**	**80.84**	**84.83**	**78.32**	**81.82**	**81.82**	**86.7**
large_binCls_highLR	82.94	79.69	83.88	76.75	80.49	80.49	85.79
large_binCls_hLR_2ep	82.86	79.69	83.52	76.62	80.59	80.59	85.67

Table 5.5: Results on SQuAD v2.0 dataset for ALBERT large configurations with binary classifier for unanswerable questions

extraction one. As a consequence, the model tends to focus on the latter one, preventing the classifier to reach better performance. However, this setting requires further investigation, considering also more complex architectures as classifier on top of the `[CLS]` output.

Hidden States Concatenation

Config	**Best**		**HasAns**		**NoAns**	
	F1	EM	F1	EM	F1	EM
base_concat2	**82.40**	79.21	82.06	75.54	**82.34**	**82.34**
base_concat2_highLR	81.82	78.73	81.3	74.58	81.78	81.78
base_concat4	82.32	**79.3**	**82.29**	**75.67**	81.45	81.45
base_concat4_highLR	81.54	78.57	81.27	74.71	81.19	81.19

Table 5.6: Results on SQuAD v2.0 dataset for ALBERT base configurations with hidden states concatenation

Considering the hidden states concatenation instead, it seems that concatenating the last four hidden states returns the best results, particularly on answerable questions; on unanswerable questions instead, this configuration performs slightly worse probably because the model puts more efforts on finding the answer span. Moreover, in both cases, having an higher learning rate hurts performance. In this case too, *base_concat2* has been preferred because of its better performances on unanswerable questions, the very little difference on answerable ones and also the lower introduction of new parameters.

In conclusion, looking at those figures, the following configurations are considered as the best ones and will be used also with the OLP dataset, as shown in the following section: *base_binCls*, *base_binCls_384seq*, *base_concat2* and *large_binCls*.

5.5.2 OLP Results

Differently from SQuAD, experiments on OLP dataset happened in two phases: a first plain evaluation using models fine-tuned only on SQuAD and another one after performing a second step of fine-tuning, on OLP itself too; results are shown in Table 5.7 and 5.8, respectively. In addition, two versions of the dataset have been employed, by changing the maximum sequence length of text passages.

Leveraging on Question Answering knowledge provided by SQuAD, OLP does not seem to be handled well, as all performances are way worse than expected. Considering how the evaluation script has been set up, it automatically computes the best null score threshold, in order to compute best metrics. In particular, *Best F1* and *Best EM* scores become meaningless, since the same null threshold is chosen among all models and due to single low scores on answerable and unanswerable questions.

After applying the second step of fine-tuning, scores greatly improve, even outperforming previous figures on SQuAD. For instance, configurations based on a maximum sequence length of 384 result to be the best performing ones on OLP. This, however, shows that even though the datasets share the same format, a fine-tuning step is still required to make the model understand the peculiarity of each of them.

Config	**MSL**	**Best**		**HasAns**		**NoAns**		**CLS**
		F1	EM	F1	EM	F1	EM	**Acc.**
ALBERT Base	-	**80.0**	**77.1**	-	-	-	-	-
base_binCls	384	62.53	62.53	**29.78**	**24.23**	31.58	31.58	50.13
base_binCls	512	62.86	62.86	26.94	21.36	**37.15**	**37.15**	**51.94**
base_binCls_384seq	384	62.53	62.53	**19.17**	**15.33**	53.54	53.54	53.23
base_binCls_384seq	512	62.86	62.86	17.4	13.03	**58.1**	**58.1**	**53.79**
base_concat2	384	62.53	62.53	**31.84**	**26.57**	28.61	28.61	-
base_concat2	512	62.86	62.86	30.61	25.45	**32.23**	**32.23**	-
ALBERT Large	-	**82.3**	**79.4**	-	-	-	-	-
large_binCls	384	62.53	62.53	**27.98**	**23.8**	38.06	38.06	49.59
large_binCls	512	62.86	62.86	26.66	22.58	**41.81**	**41.81**	**51.15**

Table 5.7: Results after plain evaluation on OLP dataset

Config	**MSL**	**Best**		**HasAns**		**NoAns**		**CLS**
		F1	EM	F1	EM	F1	EM	**Acc.**
ALBERT Base	-	80.0	77.1	-	-	-	-	-
base_binCls	384	85.92	84.41	72.21	68.18	94.05	94.05	83.25
base_binCls	512	81.49	79.74	65.62	60.9	90.78	90.78	84.03
base_binCls_384seq	384	**87.05**	**85.23**	73.17	68.32	95.36	95.36	**84.07**
base_binCls_384seq	512	82.02	80.53	67.86	63.79	90.15	90.15	83.65
base_concat2	384	81.33	79.92	59.25	55.18	94.49	94.49	-
base_concat2	512	78.32	76.59	52.17	47.42	93.64	93.64	-
ALBERT Large	-	82.3	79.4	-	-	-	-	-
large_binCls	384	**86.76**	**85.28**	74.63	69.93	93.0	93.0	83.96
large_binCls	512	87.08	85.59	73.05	68.79	94.36	94.36	**86.08**

Table 5.8: Results after fine-tuning each model on OLP dataset

Conclusions

The work of this thesis has been the right way to investigate and understand Natural Language Processing and its impact on a specific task, such as Question Answering. After introducing some basic information and concepts, this research focused on state-of-the-art, Transformer-based models to address the task.

First with BERT, then with ALBERT, the purpose was to figure out possible improvements for them, considering the difficulty of the task and the available resources. In the end, we proposed some different ideas and experiments that showed important advancements on Question Answering, particularly on the SQuAD v2.0 dataset and also on the OLP one. In fact, they involved the usage of a binary classifier focusing on unanswerable questions and a concatenation of last hidden layers, in order to have a more representative output on which extracting answer spans.

As results look promising, further investigation is needed to better understand the contribution of every ideas and how much they apply when scaling to bigger model architectures.

Future Developments

As previously described across chapters, Natural Language Processing and, in particular, Question Answering task is gaining more and more attention from the community, mainly because of the release of new state-of-the-art models such as ALBERT. Research will keep going on until they will reach human performance, or even more.

This work focused on fine-tuning ALBERT and trying to improve its performance on the Question Answering task. Even though huge performance advancements can not be expected without a different pre-training, there are still some ideas to be explored while fine-tuning. In particular, the ones explored in this work can be extended with a more careful hyperparameter tuning or with more expressive final representations for the binary classifier. Furthermore, with appropriate resources, it could be interesting to understand how to apply attention mechanisms specifically designed for span extraction, or even transformers.

Another room for improvement could be investigating new types of normalization, both on the transformer architecture or only on the final layers.

Appendix A

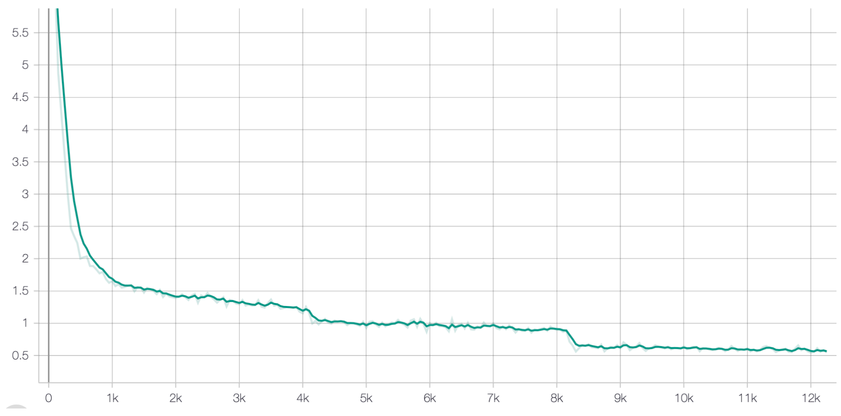
Additional Models Info

A.1 Hyperparameters Overview

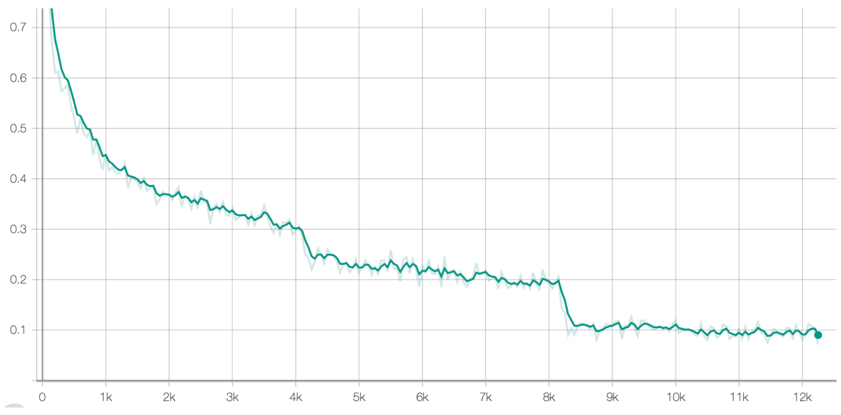
Hyperparameters	**SQuAD**	**OLP**
Batch Size	32, 12	16, 12
Max Sequence Length	512, 384	512, 384
Learning Rate	3e-5, 5e-5	2e-5
Train Epochs	3.0, 4.0	3.0
Warmup Proportion	0.1, 0.2	0.0

Table A.1: Overview of training hyperparameters used while fine-tuning on SQuAD and OLP datasets

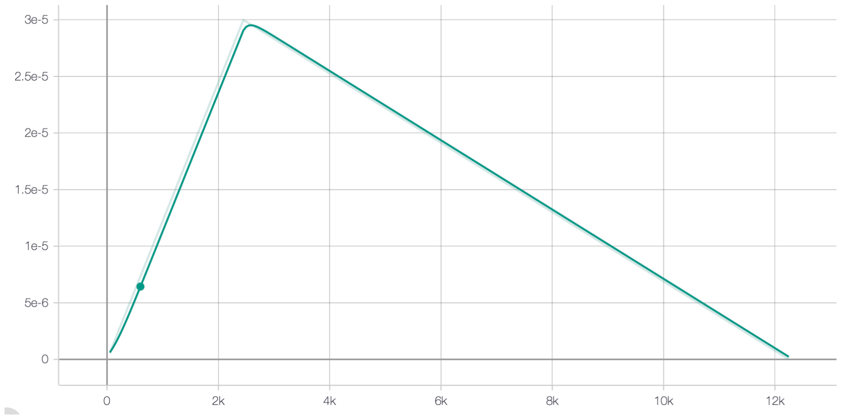
A.2 Tensorboard Data



(a) Joint loss

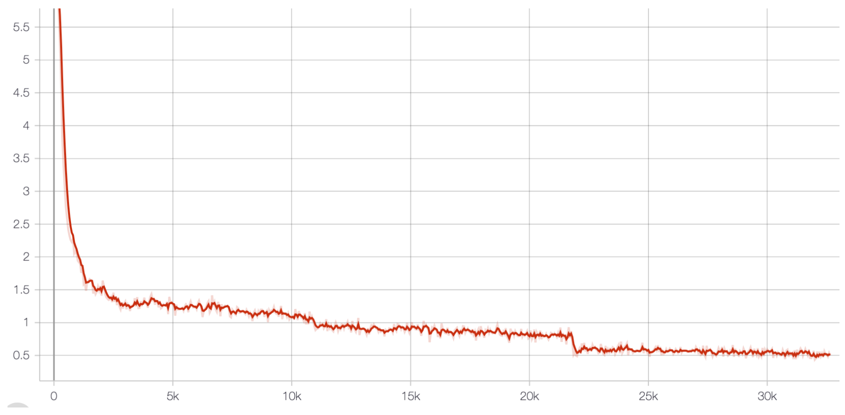


(b) Classifier loss

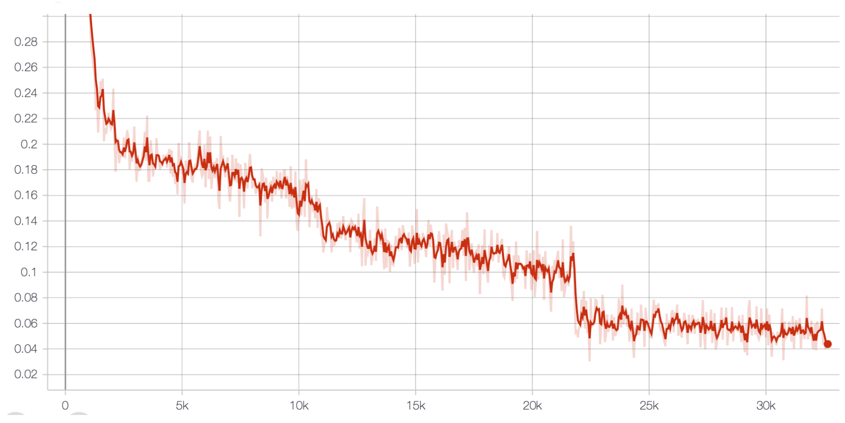


(c) Learning rate with warmup

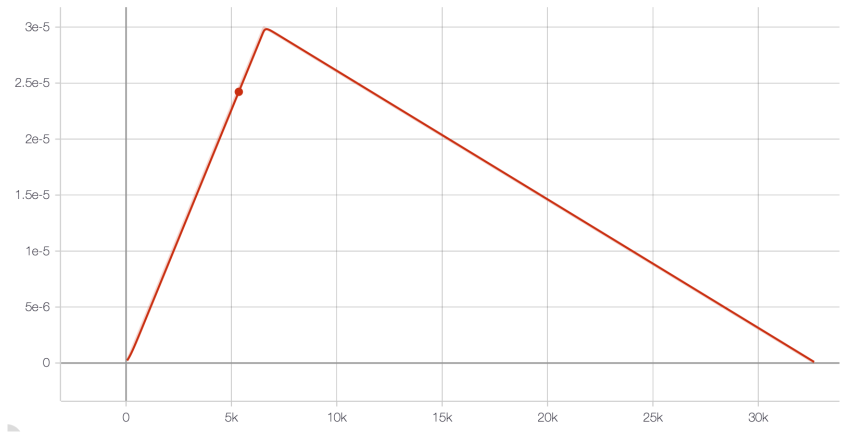
Figure A.1: Training metrics for *base_binCls* configuration



(a) Joint loss



(b) Classifier loss



(c) Learning rate with warmup

Figure A.2: Training metrics for *large_binCls* configuration

Appendix B

Code

B.1 SQuAD v2.0 Example

```json
{
  "title": "Normans",
  "paragraphs": [{
    "qas": [{
      "question": "When were the Normans in Normandy?",
      "id": "56ddde6b9a695914005b9629",
      "is_impossible": false,
      "answers": [
        {"text": "in the 10th and 11th centuries",
   "answer_start": 87},
        {"text": "10th and 11th centuries", "answer_start": 94}
      ]
    },
    {
      "question": "Who gave their name to Normandy in the
   1000's and 1100's",
      "id": "5ad39d53604f3c001a3fe8d1",
      "is_impossible": true,
      "answers": [],
      "plausible_answers": [
        {"text": "Normans", "answer_start": 4}
      ]
    }],
    "context": "The Normans (Norman: Nourmands; French:
   Normands; Latin: Normanni) were the people who in the 10th
   and 11th centuries gave their name to Normandy, a region in
   France. They were descended from Norse (\"Norman\" comes
   from \"Norseman\") raiders and pirates from Denmark..."
  }]
}
```

Code B.1: An example of JSON object regarding Normans from v2.0 dataset

B.2 ALBERT Base Updated Configuration

```
{
  "attention_probs_dropout_prob": 0,
  "hidden_act": "gelu_new",
  "hidden_dropout_prob": 0,
  "embedding_size": 128,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "max_position_embeddings": 512,
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "num_hidden_groups": 1,
  "num_labels": 2,
  "net_structure_type": 0,
  "output_attentions": false,
  "output_hidden_states": true,
  "gap_size": 0,
  "num_memory_blocks": 0,
  "inner_group_num": 1,
  "down_scale_factor": 1,
  "type_vocab_size": 2,
  "vocab_size": 30000
}
```

Code B.2: Updated configuration for ALBERT Base model following mismatches fix

B.3 ALBERT Large Updated Configuration

```
{
  "attention_probs_dropout_prob": 0,
  "hidden_act": "gelu_new",
  "hidden_dropout_prob": 0,
  "embedding_size": 128,
  "hidden_size": 1024,
  "initializer_range": 0.02,
  "intermediate_size": 4096,
  "max_position_embeddings": 512,
  "num_attention_heads": 16,
  "num_hidden_layers": 24,
  "num_hidden_groups": 1,
  "num_labels": 2,
  "net_structure_type": 0,
  "output_attentions": false,
  "output_hidden_states": true,
  "gap_size": 0,
  "num_memory_blocks": 0,
  "inner_group_num": 1,
  "down_scale_factor": 1,
  "type_vocab_size": 2,
  "vocab_size": 30000
}
```

Code B.3: Updated configuration for ALBERT Large model following mismatches fix

B.4 Binary Classifier Code

B.4.1 Layer Definitions

```python
def __init__(self, config):
  super(AlbertForQuestionAnswering, self).__init__(config)

  # ALBERT Transformer model
  self.albert = AlbertModel(config)
  # Span extractor
  self.qa_outputs = nn.Linear(config.hidden_size,
   config.num_labels)

  # Binary classifier with dropout
  self.classDrop = nn.Dropout(0.1)
  self.classifier = nn.Linear(config.hidden_size, 1)
```

Code B.4: Class definition of ALBERT model for Question Answering with the addition of the binary classifier for unanswerable questions

B.4.2 Forward Pass

```python
def forward(self, input_ids=None, attention_mask=None,
   token_type_ids=None, position_ids=None, head_mask=None,
   inputs_embeds=None, start_positions=None,
   end_positions=None, impossibleData=None,):

  outputs = self.albert(input_ids=input_ids,
   attention_mask=attention_mask,
   token_type_ids=token_type_ids, position_ids=position_ids,
   head_mask=head_mask, inputs_embeds=inputs_embeds,)

  sequence_output = outputs[0]
  pooled_output = outputs[1]

  # Computing classifier logits
  clsLogits = self.classifier(self.classDrop(pooled_output))
  clsLogits = clsLogits.squeeze(-1)
```

```

# Computing logits for span extraction
logits = self.qa_outputs(sequence_output)
start_logits, end_logits = logits.split(1, dim=-1)
start_logits = start_logits.squeeze(-1)
end_logits = end_logits.squeeze(-1)

outputs = (start_logits, end_logits, clsLogits,) + outputs[2:]

# if training
if start_positions is not None and end_positions is not None
  and impossibleData is not None:
  if len(start_positions.size()) > 1:
    start_positions = start_positions.squeeze(-1)
  if len(end_positions.size()) > 1:
    end_positions = end_positions.squeeze(-1)
  ignored_index = start_logits.size(1)
  start_positions.clamp_(0, ignored_index)
  end_positions.clamp_(0, ignored_index)

  if len(impossibleData.size()) > 1:
    impossibleData = impossibleData.squeeze(-1)

  # Computing binary classifier loss
  clLossFun = nn.BCEWithLogitsLoss()
  clsLoss = clLossFun(clsLogits, impossibleData)

  # Computing span extractor loss
  loss_fct = CrossEntropyLoss(ignore_index=ignored_index)
  start_loss = loss_fct(start_logits, start_positions)
  end_loss = loss_fct(end_logits, end_positions)
  span_loss = (start_loss + end_loss) / 2

  # Computing joint loss
  combined_loss = span_loss + 0.5 * clsLoss

  total_loss = (combined_loss, span_loss, clsLoss)
  outputs = (total_loss,) + outputs

  return outputs
```

Code B.5: Forward pass for computing model outputs and losses while training

B.4.3 Accuracy Evaluation

```
# Computing accuracy for a batch
# outputs is from a forward pass
sigmoidedIsImp = sigmoid(outputs[2])
isImpLogits = sigmoidedIsImp.detach().cpu().numpy()
isImpGT = isImpGT.detach().cpu().numpy()
tmp_acc = flat_accuracy(isImpLogits, isImpGT)

# Function definition for batch accuracy
def flat_accuracy(preds, labels):
  preds = np.where(preds >= 0.5, 1, 0).flatten()
  labelsFlat = labels.flatten()
  return np.sum(preds == labelsFlat) / len(labelsFlat)
```

Code B.6: Code executed during evaluation for computing binary classifier accuracy on a single batch

B.5 Hidden State Concatenation Code

B.5.1 Layer Definitions

```
def __init__(self, config):
  super(AlbertForQuestionAnswering, self).__init__(config)
  self.CL = config.num_concatenated_hidden_states

  # ALBERT Transformer model
  self.albert = AlbertModel(config)
  # Intermediate layer for concatenated input
  self.middle = nn.Linear(self.CL * config.hidden_size,
    config.hidden_size)
  # Span extractor
```

```
    self.qa_outputs = nn.Linear(config.hidden_size,
      config.num_labels)

    self.init_weights()
```

Code B.7: Definition of the intermediate Linear layer handling the concatenated hidden states input

B.5.2 Forward Pass

```
def forward(self, input_ids=None, attention_mask=None,
    token_type_ids=None, position_ids=None, head_mask=None,
    inputs_embeds=None, start_positions=None,
    end_positions=None,):

    outputs = self.albert( input_ids=input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids, position_ids=position_ids,
      head_mask=head_mask, inputs_embeds=inputs_embeds,)

    sequence_output = outputs[0]
    hidden_states = outputs[2]

    # Concatenating last 4 hidden states along last dimension
    concat = torch.cat((sequence_output, hidden_states[-2]), 2)
    concat = torch.cat((concat, hidden_states[-3]), 2)
    concat = torch.cat((concat, hidden_states[-4]), 2)
    # Projecting back to Hidden Size H
    middleOutput = self.middle(concat)
    # Computing span logits
    logits = self.qa_outputs(middleOutput)

    start_logits, end_logits = logits.split(1, dim=-1)
    start_logits = start_logits.squeeze(-1)
    end_logits = end_logits.squeeze(-1)

    outputs = (start_logits, end_logits,) + outputs[2:]

```

```
# if training
if start_positions is not None and end_positions is not None:
  if len(start_positions.size()) > 1:
    start_positions = start_positions.squeeze(-1)
  if len(end_positions.size()) > 1:
    end_positions = end_positions.squeeze(-1)
  ignored_index = start_logits.size(1)
  start_positions.clamp_(0, ignored_index)
  end_positions.clamp_(0, ignored_index)

  # Computing span loss
  loss_fct = CrossEntropyLoss(ignore_index=ignored_index)
  start_loss = loss_fct(start_logits, start_positions)
  end_loss = loss_fct(end_logits, end_positions)
  span_loss = (start_loss + end_loss) / 2

  outputs = (span_loss,) + outputs

return outputs
```

Code B.8: Forward pass of ALBERT for Question Answering with concatenated hidden states and loss computation while training

Bibliography

[1] Edgar Anderson. “The Species Problem in Iris”. In: *Annals of the Missouri Botanical Garden* 23.3 (1936), pp. 457–509. ISSN: 00266493. URL: `http://www.jstor.org/stable/2394164`.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization.* 2016. arXiv: `1607.06450 [stat.ML]`.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate.* 2014. arXiv: `1409.0473 [cs.CL]`.

[4] Payal Bajaj et al. *MS MARCO: A Human Generated MAchine Reading COmprehension Dataset.* 2016. arXiv: `1611.09268 [cs.CL]`.

[5] Paul Buitelaar, Philipp Cimiano, and Berenike Loos, eds. *Proceedings of the 2nd Workshop on Ontology Learning and Population: Bridging the Gap between Text and Knowledge.* Sydney, Australia: Association for Computational Linguistics, July 2006. URL: `https://www.aclweb.org/anthology/W06-0500`.

[6] Paul Buitelaar et al. “Ontology-Based Information Extraction and Integration from Heterogeneous Data Sources”. In: *Int. J. Hum.-Comput. Stud.* 66.11 (Nov. 2008), pp. 759–788. ISSN: 1071-5819. DOI: `10.1016/j.ijhcs.2008.07.007`. URL: `https://doi.org/10.1016/j.ijhcs.2008.07.007`.

[7] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of Control, Signals, and Systems* 2.4 (Dec. 1989), pp. 303–314. DOI: 10.1007/bf02551274. URL: https://doi.org/10.1007/bf02551274.

[8] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2018. arXiv: 1810.04805 [cs.CL].

[9] D. A. Ferrucci. “Introduction to This is Watson”. In: *IBM J. Res. Dev.* 56.3 (May 2012), pp. 235–249. ISSN: 0018-8646. DOI: 10.1147/JRD.2012.2184356. URL: http://dx.doi.org/10.1147/JRD.2012.2184356.

[10] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Networks”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Nov. 2011, pp. 315–323. URL: http://proceedings.mlr.press/v15/glorot11a.html.

[11] Google Cloud. *Google Tensor Processing Units (TPUs)*. https://cloud.google.com/tpu/. 2016.

[12] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[13] Dan Hendrycks and Kevin Gimpel. *Gaussian Error Linear Units (GELUs)*. 2016. arXiv: 1606.08415 [cs.LG].

[14] Daniel Hewlett et al. “WikiReading: A Novel Large-scale Language Understanding Task over Wikipedia”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1535–1545. DOI: 10.18653/v1/P16-1145. URL: https://www.aclweb.org/anthology/P16-1145.

[15] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* 2015. arXiv: `1502.03167 [cs.LG]`.

[16] Mandar Joshi 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.* Vancouver, Canada: Association for Computational Linguistics, July 2017.

[17] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2014. arXiv: `1412.6980 [cs.LG]`.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25.* Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1.* NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105.

[20] Taku Kudo and John Richardson. *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing.* 2018. arXiv: `1808.06226 [cs.CL]`.

[21] Zhenzhong Lan et al. *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations.* 2019. arXiv: `1909.11942 [cs.CL]`.

[22] Yann Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE.* 1998, pp. 2278–2324.

[23] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach.* 2019. arXiv: `1907.11692 [cs.CL]`.

[24] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[25] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* 5.4 (Dec. 1943), pp. 115–133. DOI: `10.1007/bf02478259`. URL: `https://doi.org/10.1007/bf02478259`.

[26] Tomas Mikolov et al. *Distributed Representations of Words and Phrases and their Compositionality*. 2013. arXiv: `1310.4546 [cs.CL]`.

[27] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *ICML*. 2010.

[28] Michael A. Nielsen. "Neural Networks and Deep Learning". In: *Determination Press* (2015).

[29] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[30] Pranav Rajpurkar, Robin Jia, and Percy Liang. *Know What You Don't Know: Unanswerable Questions for SQuAD*. 2018. arXiv: `1806.03822 [cs.CL]`.

[31] Pranav Rajpurkar et al. *SQuAD: 100,000+ Questions for Machine Comprehension of Text*. 2016. arXiv: `1606.05250 [cs.CL]`.

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

[33] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y`.

[34] M. Schuster and K. Nakajima. "Japanese and Korean voice search". In: *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Mar. 2012, pp. 5149–5152. DOI: `10.1109/ICASSP.2012.6289079`.

[35] Min Joon Seo et al. "Bidirectional Attention Flow for Machine Comprehension". In: *CoRR* abs/1611.01603 (2016). arXiv: `1611.01603`. URL: `http://arxiv.org/abs/1611.01603`.

[36] Mohammad Shoeybi et al. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*. 2019. arXiv: `1909.08053 [cs.CL]`.

[37] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. arXiv: `1409.1556 [cs.CV]`.

[38] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: `http://jmlr.org/papers/v15/srivastava14a.html`.

[39] Stanford NLP Group. *SQuAD v2.0 Leaderboard*. `https://rajpurkar.github.io/SQuAD-explorer/`. 2018.

[40] Christian Szegedy et al. "Going Deeper with Convolutions". In: *Computer Vision and Pattern Recognition (CVPR)*. 2015. URL: `http://arxiv.org/abs/1409.4842`.

[41] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: `1706.03762 [cs.CL]`.

[42] Thomas Wolf et al. "HuggingFace's Transformers: State-of-the-art Natural Language Processing". In: *ArXiv* abs/1910.03771 (2019).

[43] Yonghui Wu et al. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. 2016. arXiv: 1609.08144 [cs.CL].

[44] Yuxin Wu and Kaiming He. *Group Normalization*. 2018. arXiv: 1803.08494 [cs.CV].

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

[46] Zhilin Yang et al. *XLNet: Generalized Autoregressive Pretraining for Language Understanding*. 2019. arXiv: 1906.08237 [cs.CL].

Acknowledgements

I would like to seize the opportunity to thank all the people that supported me while writing this thesis. In particular, I would like to thank my supervisors, Prof. Dr. Fabio Tamburini from Alma Mater Studiorum - University of Bologna and Prof. Dr. Philipp Cimiano from Bielefeld University; Frank Grimm for his invaluable support throughout the research and work done, either in Bielefeld and Bologna.

A special thank goes also to Simone Preite, not only for the joint project work on the data later used in this thesis but also for the friendship we share.

But the biggest thanks go to my family: my sister Martina, my mother Maria Rosaria, my father Giovanni and my grandparents for being always supportive and for letting me think to my future without any constraints; and to my dear friends, with whom I shared my studies, travels, joy, laughs and bad moments too. It would not have been the same without you!

Last, but not least, Giulia, Anna and Simone, you deserve a special mention because of all the time we spent together, parties, travels, black humour and the whole year in Bielefeld, Germany that went by more as an adventure, with you. I am so grateful for having met you!

Thank you so much to all of you!