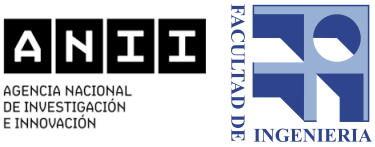
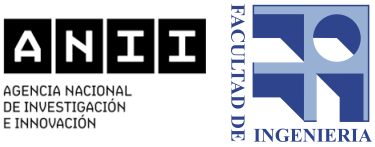


Web Application Attacks Detection Using Deep Learning

Nicolás Montes De Marco

Programa de Posgrado en Ingeniería Matemática
Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
Setiembre de 2021



UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Web Application Attacks Detection Using Deep Learning

Nicolás Montes De Marco

Tesis de Maestría presentada al Programa de Posgrado en Ingeniería Matemática, Facultad de Ingeniería de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de Magíster en Ingeniería Matemática.

Directores:
Dr. Gustavo Betarte
Dr. Álvaro Pardo

Director académico:
Dr. Gustavo Betarte

Montevideo – Uruguay
Setiembre de 2021

Montes De Marco, Nicolás

Web Application Attacks Detection
Using Deep Learning / Nicolás Montes De Marco. - Montevideo: Universidad de la República, Facultad de Ingeniería, 2021.

x, 98 p. 29, 7cm.

Directores:

Gustavo Betarte

Álvaro Pardo

Director académico:

Gustavo Betarte

Tesis de Maestría – Universidad de la República, Programa en Ingeniería Matemática, 2021.

Referencias bibliográficas: p. 91 – 98.

1. Web Application Firewall, 2. Detección de Anomalías, 3. Aprendizaje Profundo. I. Betarte, Gustavo, Pardo, Álvaro, . II. Universidad de la República, Programa de Posgrado en Ingeniería Matemática. III. Título.

Agradecimientos

Quiero agradecer a mis tutores Gustavo Betarte y Álvaro Pardo. Ellos me motivaron en todo momento y me apoyaron dedicando mucho de su tiempo. También a Rodrigo Martinez, el cual me ayudó muchísimo a lo largo de toda la maestría. Siempre me tuvieron mucha paciencia y una gran disposición para ayudarme y brindarme todo su conocimiento. Quiero agradecer también al tribunal conformado por Tamara Rezk, Luis Chiuruzo y Marcelo Fiori. Por toda su dedicación y las excelentes devoluciones que me brindaron.

Agradezco a los que hacen posible la Maestría en Ingeniería Matemática, que tiene como objetivo formar y capacitar recursos humanos para la investigación y el desarrollo científico y tecnológico. Quiero agradecer también la financiación recibida (para mi maestría) de la Agencia Nacional de Investigación e Innovación (ANII) [1] en el contexto del proyecto de investigación aplicada: Fondo María Viñas FMV 1 2017 136337. También al centro *Information and Communication Technologies for Verticals* [2] (ICT4V) que financió mi proyecto final de grado, el cual me permitió construir bases sólidas de conocimiento para luego seguir con el trabajo de maestría.

[1] https://www.anii.org.uy/
[2] https://ict4v.org

RESUMEN

En esta tesis se explora el uso de técnicas de aprendizaje profundo para mejorar el rendimiento de Web Application Firewalls (WAFs por su siglas en inglés). Dichos sistemas son utilizados para detectar y prevenir ataques a aplicaciones web. Normalmente un WAF inspecciona las solicitudes del Protocolo de Transferencia de Hipertexto (HTTP) que se intercambian entre el cliente y el servidor, con el objetivo de detectar ataques y bloquear potenciales amenazas. En el enfoque que se propone, modelamos el problema como un caso supervisado de una clase y construimos un extractor de características (features) utilizando una técnica de aprendizaje profundo. Concretamente, tratamos las solicitudes HTTP como texto y entrenamos un modelo de lenguaje profundo con una arquitectura basada en un tipo de redes neuronales que se denominan *Transformers*. El uso de modelos de lenguaje previamente entrenados ha producido mejoras significativas en un conjunto diverso de tareas de Procesamiento de Lenguaje Natural (PLN) porque son capaces de realizar aprendizaje por transferencia. En nuestro enfoque utilizamos el modelo previamente entrenado como un extractor de características para mapear las solicitudes HTTP en vectores numéricos. Luego, estos vectores se utilizan para entrenar un clasificador de una clase. También utilizamos una métrica de evaluación establecida para definir un punto operativo (de forma automática) para el modelo de una clase. Los resultados experimentales muestran que el enfoque propuesto supera a aquellos obtenidos con ModSecurity, un WAF ampliamente utilizado. La capacidad de detección de ataques de ModSecurity depende fuertemente de la aplicación de reglas configuradas con el Core Rule Set (CRS) de OWASP, el conjunto de reglas más ampliamente utilizadas para la prevención de ataques a aplicaciones Web. Una de las principales ventajas de nuestro enfoque es que no se requiere la participación de un experto en seguridad para el proceso de extracción de características. Palabras claves:

Web Application Firewall, Detección de Anomalías, Aprendizaje Profundo.

ABSTRACT

In this thesis, we present the use of deep learning techniques to improve the performance of Web Application Firewalls (WAFs), systems that are used to detect and prevent attacks to web applications. Typically a WAF inspects the HyperText Transfer Protocol (HTTP) requests that are exchanged between client and server to spot attacks and block potential threats. We model the problem as a one-class supervised case and build a *feature extractor* using a deep learning technique. We treat the HTTP requests as text and train a deep language model with a transformer encoder architecture which is a self-attention-based neural network. The use of pre-trained language models has yielded significant improvements on a diverse set of NLP tasks because they are capable of doing *transfer learning*. We use the pre-trained model as a *feature extractor* to map the HTTP requests into feature vectors. Then, these vectors are used to train a one-class classifier. We also use an established performance metric to define an operational point for the one-class model automatically. The experimental results show that the proposed approach outperforms the ones of the classic rule-based ModSecurity configured with a vanilla CRS and does not require the participation of a security expert to define the features.

Keywords:
Web Application Firewall, Anomaly Detection, Deep Learning, Transformers.

Contents

Chapter 1

Introduction

Web applications have changed our way of life, leveraging daily operations such as bank transfers, booking a flight and making online purchases. A web application is a piece of software based on a client-server architecture that embodies a coordinated set of functions. The information flowing between the client (which runs on the user's web browser) and the application server is transmitted using the HTTP protocol [10]. It is quite usual for a web application code to contain vulnerabilities like the ones listed and described in the OWASP top 10 [52]. Hence, the (automation of) detection of attacks on web applications is a quite active area of research [69].

It has become a regular security practice to deploy a Web Application Firewall (WAF) [24] to analyze the *request/response* flow through the communication channel to identify attacks that exploit vulnerabilities of web applications. A WAF is a piece of software that intercepts and inspects all the traffic between the web server and its clients, searching for attacks inside the HTTP packet contents. An implementation of an open source WAF that has become a *de facto* standard is ModSecurity [76], which allows the analysis of the users requests and the application responses by enabling real-time web application monitoring, logging and access control. The actions ModSecurity undertakes are driven by rules that specify, by means of regular expressions, the contents of the HTTP packets to be analyzed and eventually flagged as potential attacks. ModSecurity comes equipped with a default set of rules, known as the Open Web Application Security Project (OWASP) Core Rule Set (CRS) [51], for handling the most usual vulnerabilities included in the OWASP Top 10 [52]. This rule-based approach, however, has some drawbacks: rules are static

and rigid by nature, so the CRS usually produces a rather high rate of false positives, which in some cases may be close to 40% [22].

Rule tuning is a time consuming and error prone task that has to be manually performed for each specific web application. In traditional networks firewalls and Intrusion Detection Systems (IDS), the approach based on rules has been successfully complemented with other machine learning-based tools, anomaly detection and other statistical approaches which provide higher levels of flexibility and adaptability. Those approaches take advantage of sample data, from which the normal behavior of the web application can be learned in order to spot suspicious situations which fall out of this nominal use (anomalies) and which could correspond to ongoing attacks [45].

The systematic review presented in [69] analyzes the available scientific literature focused on detecting web attacks using machine learning techniques. However, one of the most challenging problems when applying machine learning models to web security is how to extract features from the raw data [58] (the HTTP request). This task, also known as *feature extraction*, consists of transforming the input text into some numerical form that a machine learning model can use to learn, as figure 1.1 shows. Researchers have spent a lot of time and effort exploring different *feature extraction* methods and it is still an open challenge [58].

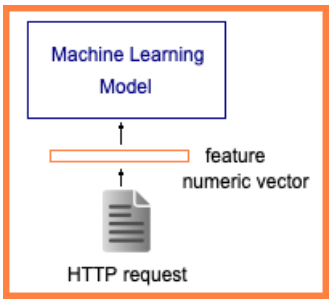


Figure 1.1: Example of a HTTP request encoding into a numeric vector.

Recently, deep learning techniques have shown to be quite successful in many fields achieving outstanding performance in many classification tasks. Furthermore, researchers have discovered that the convenient features for the classification tasks can also be obtained with deep learning. A concept called *transfer learning* has been instrumental in the success of deep learning in computer vision. As it is explained in [8], creating a good deep learning network for computer vision tasks can take millions of parameters and be very expensive

to train. Several research works have shown that deep networks can learn hierarchical feature representations (simple features like edges at the lowest layers with gradually more complex features at higher layers). Rather than training a new network from scratch each time, the lower layers of a pre-trained network with generalized image features could be copied and transferred for use in another network with a different task.

The *transfer learning* paradigm has also been successful in the area of NLP, where it has been used as a *feature extraction* technique. To be more specific, machine learning algorithms cannot work on the raw text directly. So, some *feature extraction* technique is needed to convert the raw text into a vector of features [1], namely, transform the input text into some numerical form that a machine learning model can use to learn. Advanced approaches, for *feature extraction*, in NLP rely on building *deep neural networks language models* that take raw text as input and learn to induce features as part of the learning process. The weights of the neural language networks already encode a lot of information about the language. Despite being trained with only a language modeling goal, they learn highly transferable and task-agnostic properties of the language [20]. These language networks can be used as input to another downstream task with the *transfer learning* approach. It is the opinion of well-known researchers that the introduction of deep pre-trained language networks signals the same shift in NLP to *transfer learning* that the one experimented by the field of computer vision [8].

1.1. Problem statement

This thesis investigates the use of a *transfer learning* technique combined with an anomaly detection model to improve the performance of web application firewalls (WAFs). We focus on a scenario where only valid requests are known, and no requests tagged as attacks are necessary. We believe that is a quite realistic approach, where valid traffic could be collected, for instance, from performing functional testing of the application. More specifically, this dissertation addresses three main problems, summarized as follows:

- As already mentioned, ModSecurity combined with the Open Web Application Security Project (OWASP) Core Rule Set (CRS) requires rule tuning tasks before being able to protect web applications. Therefore,

we will propose a learning framework to complement the rule-based approach to:

  - improve the Open Web Application Security Project (OWASP) Core Rule Set (CRS) attack detection and
  - diminish the false positives generated by the Open Web Application Security Project (OWASP) Core Rule Set (CRS) to reduce the initial time-consuming tuning task.

- In order to protect web applications by using anomaly detection techniques, the raw data (i.e., the HTTP requests) must be converted into a numeric feature vector. The learning framework that we will propose uses a *transfer learning* approach to extract the features, namely, transforming the HTTP request into some numerical form. For this, we treat the HTTP request as text and train a deep language network, which in turn is used (as a *feature extractor*) as the input for an anomaly detection model.
- Many anomaly detection models, also known as one-class classifiers, require setting up one or more parameters to optimize the model's performance. We will propose an approach that automatically selects, using an established performance metric, an operational point for the one-class model (without the need for data labeled as attacks).

1.2. Contributions

Inspired by the success of *transfer learning* for NLP tasks, also successfully applied to other modalities such as programming languages, this thesis contributes to validating the following research questions:

1. Can the powerful approach of deep language networks be used as a feature extractor of HTTP requests in order to address web application attack detection? Can they replace the participation of a security expert to define the features?
2. Once the deep language network from HTTP requests (as a feature extractor) is built, can these features be used as an input to a one-class classifier and get an operational model automatically without the need for data labeled as attacks?

To validate these research questions, we put forward an approach that uses a *deep language model* to address web application attack detection. It consists of a two-step learning framework: first, we build a *feature extractor* with a *deep language model* trained in a self-supervised fashion; then, we train a one-class supervised model with these features as input.

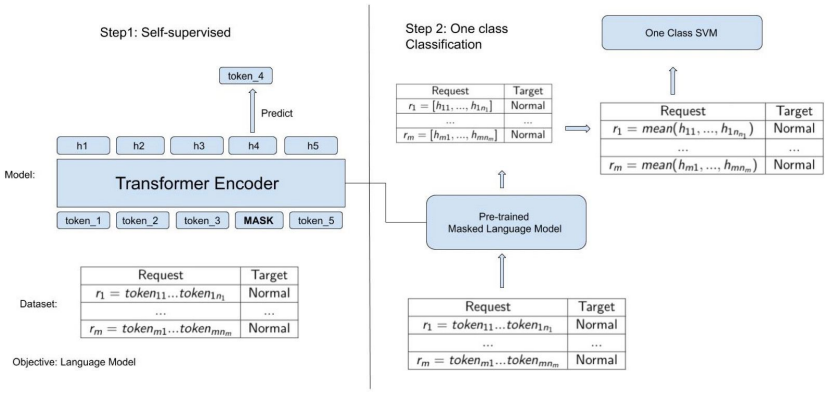


Figure 1.2: Proposed architecture. Left: deep pre-trained language model used to extract the contextual representation of each token $token_{ij}$ in the request r_i. Right: each request r_i is represented as the mean of token deep contextualized representation h_{ij} and how they are used to train a one-class classifier.

We treat HTTP requests as raw text and train a deep language network. The model architecture used to train it is called *Robustly Optimization Bidirectional Encoder Representations from Transformers* (RoBERTa) [43]. The architecture comes directly from the transformer model proposed by [78]; it is just the encoder portion of the model. RoBERTa [43] can operate in huge amounts of text and it is called self-supervised because the optimization of the network does not require labels (we shall explain in detail this in Section 4). This pre-trained language model can extract feature vectors that shall then be used to train a one-class classifier to detect attacks to web applications. As shown in figure 1.2, in the first step, we create a deep pre-trained language model from scratch using a set of HTTP requests from the web application that we aim to protect. In a second step, we use the *feature-based* strategy to transform each HTTP request into a feature vector. Once we have obtained the pre-trained model, we convert each HTTP request into a numeric representation using the weights of the last layer of the network, also known as *feature extraction*. Finally, with these representations as input, we build a One-Class Classification model (OCC).

In this thesis, we will expose the proposed learning framework in detail and discuss the experimental results, showing that our approach outperforms the classic rule-based ModSecurity configured with a vanilla CRS and does not require the participation of a security expert to define the features. To the best of our knowledge, our proposed framework is the first attempt in using transformer-based language representation of HTTP requests to address the problem of web applications attack detection. The work gave rise to an article that was accepted in the 25-th Iberoamerican Congress on Pattern Recognition:

Montes, N., Betarte, G., Martínez, R., Pardo, A. (2021, February). Web application attacks detection using deep learning. [1] In 25-th Iberoamerican Congress on Pattern Recognition.

1.3. Thesis Outline

The rest of this document is structured as follows: Chapter 2 discusses the related works. In Chapter 3 we provide a primer on web applications security and protection mechanisms. We also introduce the concept of *word embeddings* representations and describe the most common methods used to create them. These concepts underlie the more powerful pre-trained language representation models such as BERT [17], RoBERTa [43], XLM [38] or XLNet [81]. We propose a learning framework that uses a RoBERTa language model as a feature extractor to addres web application attack detection. Chapter 4 explains RoBERTa, the language representation model which is at the core of this thesis. There we describe in detail the self-attention mechanism, the model architecture, its pre-training procedure and the application to downstream tasks. Chapter 5 exposes the One-Class SVM, the model in charge of classification with the features extracted with the RoBERTa model. It also explains the performance metrics used to automatically define an operating point for the one-class SVM model. Chapter 6 presents the implementation of the learning framework and the experimental results. Finally, in Chapter 7 we put forward further work and conclusions.

[1] https://link.springer.com/chapter/10.1007/978-3-030-93420-0_22

Chapter 2

Related Work

The work presented in this thesis has been developed as a member of the research team WAF Mind[1][2]. In the context of the projects developed by that research group several works have been proposed [9, 10, 45] that put forward solutions where the rule-based detection approach of ModSecurity was successfully complemented with machine learning-based and anomaly detection models to improve its detection capabilities, with particular focus on the task of diminishing false positives. The main results are presented in [45]. In that work, it was implemented an anomaly detection framework that experiments with two approaches for the *feature extraction* step: a classic information retrieval technique and an *expert-assisted* mechanism, where the features to be extracted are defined by a security expert. Then, with those features as input, a one-class model based on a Gaussian mixture model is used to detect attacks.

The present thesis had as a main objective to enhance the work presented in [45] by, in particular, validating whether it is possible to extract features with the help of a deep learning technique. We put forward an approach that uses a *deep language model* to address web application attack detection. It consists of a two-step learning framework: first, we build a *feature extractor* with a RoBERTa-based language model [43] trained in a self-supervised fashion; then, we train a one-class supervised model with these features as input. We operate in a scenario where only valid requests are known, and no requests tagged as attacks are necessary. As already mentioned, we believe that is a realistic

[1] https://www.fing.edu.uy/inco/proyectos/wafmind/

[2] This thesis was supported by a grant given from ANII (http://anii.org.uy) and was done in the context of project FMV_1_2017_136337 (Fondo María Viñas, ANII) and project WAFINTL from ICT4V center (http://ict4v.org).

approach, where valid traffic could be collected, for instance, from performing functional testing of the application.

One of the main goals of this thesis is to evaluate if the features extracted using a deep learning technique are comparable with those extracted by a security expert and to assess whether it is possible to use them in a production environment to detect attacks on web applications.

In what follows we shall first discuss research that have dealt with *feature extraction* in NLP. In particular we shall focus on work that has recently been applied in tasks related to programming languages, which in turn inspired us to use these techniques in our approach. Then, we shall review related work concerning the application of machine learning techniques to increment web applications security.

2.1. Transfer Learning in NLP

As already pointed out, machine learning algorithms cannot work on the raw text directly. So, some *feature extraction* technique is needed to convert raw text into a vector of features [1], namely, transform the input raw text into some numerical form that a machine learning model can use to learn. Advanced approaches in NLP rely on building *deep neural networks language models* that take raw text as inputs and learn to induce features as part of the learning process. Those features can be used as input of another downstream task with the *transfer learning* approach.

Deep neural networks language models trained in a self-supervised fashion such as BERT [17] and RoBERTa [43] have become ubiquitous in NLP, and have led to significant improvements in many tasks. Those models rely on a two-step learning approach. First, they learn deep contextual word representations from the raw text in a self-supervised way (stage referred to as pre-training). Then, this pre-trained language model can be applied to NLP tasks by choosing between two learning strategies: *feature-based* and *fine-tuning*. The quality of pre-training mainly comes from the Masked Language Modeling (MLM) objective (inspired in the *the cloze task* or *fill in the blanks* proposed by [70]), which consists of randomly masking words from an input text, and training a model to recover the original input [62] (we will explain all these concepts detailed in Section 4).

Recent studies showed that pre-training methods developed for NLP are

also effective for programming languages (Programming Language Processing -PLP-). For example, [21] proposed CodeBERT, a RoBERTa-based language model trained on source code. The model presented in that work performs well on downstream code generation tasks and outperforms previous pre-training approaches. More recently, [37] applied the unsupervised machine translation principles of [38] (that uses contextual representations of tokens as a fundamental building block) to monolingual source code from GitHub. The work presented in [37] showed that the resulting model, TransCoder, was able to translate source code between Python, Java, and C++, in a fully unsupervised way.

Recently in [62] it has been proposed to use a code-specific objective to better pre-train models designed to be *fine-tuned* on code generation tasks: code *deobfuscation*. The authors of that work introduce a new pre-training objective, called DOBF, that leverages the structural aspect of programming languages and pre-trains a model to recover the original version of obfuscated source code.

2.2. Machine learning and web security

The systematic review presented in [69] compares several works that use machine learning and pattern recognition techniques to improve the security of web applications.

In [36], the authors propose an anomaly detection approach which models the specific characteristics of the URL parameters. This work focuses on parameter length and order generating a probabilistic grammar of each parameter. In our anomaly detection approach, we work using the whole requests, not only the URL parameters. We manage to capture the normal behavior of a user of the application by training a deep language model using a set of HTTP requests from the web application that we aim to protect. In a second step those features are used as input for an anomaly detection model. This allows us to capture the behavior of the data sent in the normal use of the application. Since the dataset we have used not only has attacks in the URL parameters but also in the body and headers, it is difficult to compare our work and approach to that developed in the referred work.

Several authors have proposed to use anomaly detection techniques that work over a simplified version of the application's parameter values. In [14]

the authors abstract away numbers and alphanumeric sequences, representing each category with a single symbol. Torrano, Perez, and Marañón [74] present an anomaly detection technique where instead of using the tokens themselves, they use a simplification that only considers the frequencies of three sets of symbols: characters, numbers, and special symbols. As was mentioned, in our anomaly detection approach, we analyze not only the parameters values but the whole request without any further simplification. We decided to use the entire HTTP request as *raw text* because we are interested in modeling a language that, despite being trained with only a language modeling goal, they learn highly transferable and task-agnostic properties of the language. Then, use it as a feature extractor with the *transfer learning* approach.

In [83] an approach that uses word embeddings to represent the URLs is proposed. This approach has three steps. Firstly, the authors extract features with a classic information retrieval approach and apply an ensemble clustering model to separate anomalies from normal samples. Then, they use *word2vec* to get the semantical representations of anomalies. Finally, another multi-clustering approach clusters anomalies (with the features generated with *word2vec*) into specific types. In our approach, we are replacing static embeddings (*word2vec*) with deep contextualized representations. Moreover, we use these representations to get the semantical presentations of normal data and use it as input to build the one-class model.

The work [82] proposes a method that uses Bidirectional Long Short-Term Memory (Bi-LSTM) with an attention mechanism to model HTTP traffic. However, this approach is supervised (while ours uses only normal data for training), as they train the Bi-LSTM network to predict if a request is anomalous or not.

The work [58] proposes a model which learns semantic of malicious segments in payload using Recurrent Neural Network (RNN) with an attention mechanism. The authors transform the payload to a hidden state sequence using a recurrent neural network. The authors then use the attention mechanism to weigh the hidden states as the feature vector for further detection. The approach presented in [58] is similar to our approach because, in that work, it is possible to use the hidden states of the network as features for a second classifier (as *feature extraction*). However, there exists an essential difference with our work because in [58] they learn the weights of the RNN in a supervised way, optimizing the parameters to predict if a request is abnormal

or not. On the contrary, in our approach, we do it in a self-supervised way. In other words, we do not need data labeled as attacks for language modeling or to learn the classifier.

Although some works use deep neural networks to represent requests, none of them do it in an unsupervised (or self-supervised) way, as we propose in our learning framework.

In [46] static approaches to detect XSS vulnerabilities in source code using neural networks are explored. The authors compare two different code representation techniques based on NLP and PLP and experiment with models based on different neural network architectures for static analysis detection in PHP and Node.js source code. The authors experiment with two code representation approaches to classify source code into two categories: XSS-safe or unsafe.

In the first approach, the authors designed a system using *word2vec* [48] to represent the source code. The method uses source code as a *corpus* and the *words* as the different tokens in the source code. To create a vectorial representation of a given source code file, the authors concatenate each of the vectors associated with the tokens that appear in a single file. Thus, each file is represented as the concatenation of the *word2vec* embeddings of the tokens present in the file. These concatenated embeddings will be the input of a second deep learning classifier that predict if the source code is XSS-safe or unsafe. The second approach is based on the Abstract Syntax Tree (AST) representation proposed by*code2vec* [2]. The authors of *code2vec* present a technique to represent embeddings to be used in deep learning models that seek to analyze source code. In [46], the authors used the neural network based on attention architecture originally proposed by *word2vec* [2] to implement the AST representation for JavaScript and PHP source code. The authors of [46] modify the attentional-based neural network proposed by [2] and adapt it to predict whether the file is XSS-safe or unsafe (a binary classification problem).

There exists an essential difference between our approach and the one proposed by [46]. In that work the authors explore statics methods to detect XSS vulnerabilities in source code. Instead, in our approach, we analyze the *requests/response* flow through the communication channel to identify attacks that exploit vulnerabilities of web applications. To be more precise, our approach focuses on the traffic between the web server and its clients, searching for attacks inside the HTTP packet contents. The main similarity of the ap-

proaches is that both explore deep learning methods as a *feature extractor*. However, as already mentioned, the essential difference is that [46] focuses on modeling the source code, and we focus on modeling the HTTP traffic.

Chapter 3

Background

This section provides a concise primer on web applications, including the HTTP protocol, security issues, and protection mechanisms. Then, it points out the challenge of extracting features when applying machine learning models for web application security and provides an overview of different approaches for feature extraction in NLP. Finally, it dives into the concept of *word embeddings* and some standard methods used to create them. These representations underlie the advanced deep language representation models like RoBERTa [43], which is proposed to use in this work to extract features.

3.1. Web Applications Security

A web application is a client-server software where the application is hosted in a web server (for example, `www.example.com`), and the client is, usually, a web browser (for example, *Firefox*). The interaction between the client and the server is performed over the HTTP(s) protocol. Figure 3.1 illustrates a typical situation (taken from [3]) where a web browser sends to the server a request for a certain resource (the page `search.php`).

1. A network packet is sent from the web browser (*Firefox*) to the web server (`www.example.com`). The packet contains a request for the page (`search.php`) and is sent according to the HTTP protocol (it is contained in the portion of the network packet which is known as *HTTP payload*). The information provided to the web server with the request are:
    - **The name of the HTTP method**. A different method is used in consequence of the kind of action the client is requiring to the web

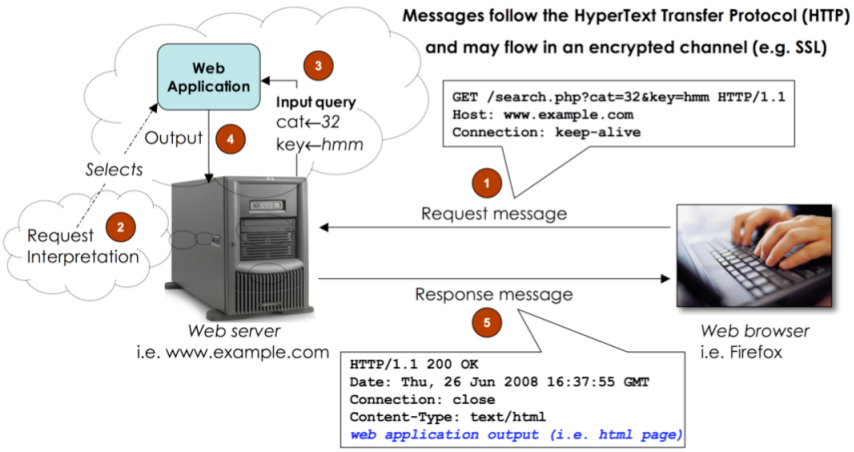


Figure 3.1: Example of the interaction between a client and a web application [3]

server. In particular `GET` is the method tipically used to retrieve information from the web server and the `POST` method is used to submit an entity to the specified resource.

- **The *request URI*** that identifies a page (or a set of pages) on a web server by the path and/or query parameters (also called *attributes*). For instance, if `www.example.com` is developed in static HTML and it has a page on that site called `about.html` located in a sub-directory on the site, the *request URI* for that page might be `/prag/eng/home.html`. On the other hand, if `www.example.com` is developed in `php`, the *request URI* for that page might look like `/pages.php?group=prag&lang=eng&page=about.php`. The example in figure 3.1 contains a request for the application `search.php`. The application receives two attributes: `cat` with value `32` and `key` with value `hmm`.
- **The version of the HTTP protocol**. In the example the request is sent according to the version 1.1 of the protocol.
- **The headers** are just two in the example. The `Host` header which specifies the host of the resource being requested and the `Connection` header that allows the sender to specify options that are desired for that particular connection (in the example the option `Connection: keep-alive` is sent).

2. When the HTTP request is received it is interpreted by a *parsing engine* which extracts the name of the application requested and the input

attributes.

3. The application is called and the attributes are passed as arguments.
4. The application generates the resource required.
5. The resource required is sent back to the web client within a response message.

As was already mentioned, it is pretty usual for a web application code to contain vulnerabilities like the ones listed and described in the OWASP top 10 [52]. There exist several types of vulnerabilities. In this work, we are going to focus on vulnerabilities related to input validation. When the software does not correctly validate inputs, an attacker can send crafted payloads that may affect the application control or data flow [45]. The two most important vulnerabilities related to improper input validation are Injection and Cross-Site Scripting (XSS).

The most critical web application security risk is injection. Injection occurs when data sent by the user to the application is used, without proper validation, to construct an instruction that an interpreter executes in the backend producing unexpected (by the designer of the application) results. There are different types of injections depending on the interpreter being exploited: SQL, LDAP, XPath, NoSQL queries, OS commands, XML parsers, SMTP headers, among others. An SQL injection occurs when an attacker sends SQL code as input to the application and it results in a query that returns, in an unauthorized manner, sensitive data. Let us consider an example taken from [77]: it is related to a web application that lets a user list all his registered credit cards of a given type. A possible pseudo-code for a similar application is the following:

```
uname = getAuthenticatedUser();
cctype = getUserInput();
result = sql("SELECT nb FROM creditcards WHERE user = '" +
uname + "' AND type = '" + cctype + "';");
print(result);
```

If a user *Bob* performs a search for all his *VISA* cards the following *query* would be executed:

```
"SELECT nb FROM credit cards WHERE user = 'bob' AND
type = 'VISA';
```

Bob could manipulate the structure of the SQL query if the input supplied is not properly sanitized. For example, if the value `' OR user = 'alice` is passed to the `cctype` attribute the following *query* would result:

```
"SELECT nb FROM creditcards WHERE user = 'bob' AND
type = '' OR user = 'alice';
```

This *query* will return to *Bob* a list of all the credit cards belonging to *Alice*.

Injection and XSS have always been included in the OWASP Top 10 since the first publication in 2007. Nevertheless, applications still suffer from these types of vulnerabilities. Thus, input validation is critical for software security. One such validation consists of verifying and filtering all data that flow to the system before they are effectively used. Unfortunately, these procedures usually are not introduced at the design stage of applications leaving then unattended vulnerabilities that might be critical, especially to web applications. When data is processed without proper validation an attacker could lead the system to unpredictable states and exploit this for his benefit. This data may also produce unexpected results that the attacker could analyze to infer further information to proceed with his activity[45].

The protection of a web application requires the ability to distinguish the behavior of a valid user from that of a malicious agent. A technological tool for performing that task is a WAF. In this work we shall focus in the WAF ModSecurity. This tool intercepts and inspects all the traffic between the web server and its clients, searching for attacks inside the HTTP requests.

ModSecurity is a de facto standard WAF in the open community. It is open source, flexible and extensible. It comes equipped with a default set of rules, known as the OWASP Core Rule Set (OWASP CRS) [51], for handling the most usual vulnerabilities. However, an approach only based on rules has some drawbacks: rules are static so the CRS usually produces a relatively high rate of false positives, which in some cases may be close to 40%. Such a high rate of false positives might potentially lead to a denial of service of the application.

As already mentioned in section 2, several authors have experimented on detecting web attacks using machine learning techniques. However, one of the most challenging problems when applying machine learning models to web security is how to extract features from the raw data [58]. This task also known

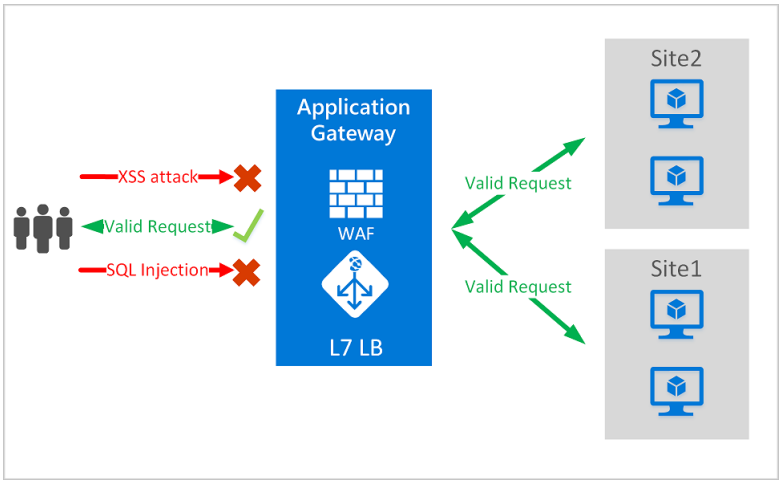


Figure 3.2: A Web Application Firewall

as *feature extraction*, consists of transforming the input text into some numerical form that a machine learning model can use to learn. Researchers have spent a lot of time and effort exploring different *feature extraction* methods, and it is still an open challenge [69].

In what follows we proceed to give an overview of different approaches for performing feature extraction in NLP. In particular we shall focus in what are denominated *word embeddings* and some standard methods that can be used to create those embeddings. These representations, in turn, constitute the basis of advanced deep language representation models like RoBERTa [43], which is the one we have favoured in our work to perform feature extraction.

3.2. Text encoding approaches in NLP

Many machine learning algorithms require the input to be represented as a numeric feature vector. In [30] Jain et al. present a review on statistical pattern recognition and define a general architecture of a pattern recognition system. As figure 3.3 shows, the system is operated in one of two modes: *training mode*, where training instances are used to adjusts the models parameters, and a *classification mode*, in which new instances are classified by the system.

The pre-processing module is the first module of the pipeline and is responsible for cleaning the raw data to remove noise. Then the *feature extraction/selection* module transforms the resulting data into a numeric vector. Finally the *learning module* train a classification model using the extracted

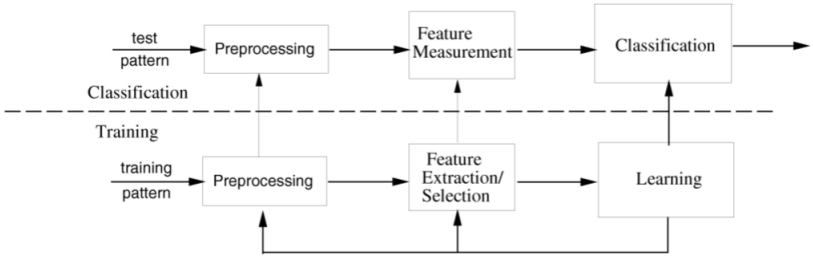


Figure 3.3: Statistical pattern recognition architecture [30]

features. After training the learning model, the system can be used to classify new instances using the classification mode.

The rest of this section is devoted to outline some of the classic and advanced approaches to carry out *feature extraction* in NLP.

Bag of words (BoW) [84] is a classic *feature extraction* technique used in information retrieval (IR) systems. This model is used to transform the input text into numerical form (text encoding). This model encodes an input text (an HTTP request or a sentence) as the sum of the one-hot encoding vector of each token present in the input text. The one-hot encoding transforms each token in a vector of dimension $R^{|V|}$, where $|V|$ is the size of the vocabulary (all set of *words/tokens* used to model the text language). This vector contains 1s in the index corresponding to each word that appears in the vocabulary and 0s elsewhere. For example, if the word *at* is the entry number i on the vocabulary, the *i-th* element of this output vector is a 1 and 0s elsewhere.

$$w^{aardvark} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, w^{a} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, w^{at} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \cdots w^{zebra} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

Figure 3.4: One hot representation [32]

The Bag of Words model transforms an input text into a vector of numbers (with dimension $R^{|V|}$ where $|V|$ is the size of the vocabulary), where each position of the vector corresponds to a word. The value of the position corresponds to the times the corresponding word appears in the text.

Figure 3.6 shows a small example for a Bag of Words matrix showing the frequency of four words in four plays by Shakespeare (documents).

Each cell in the BoW matrix represents the number of times a particular

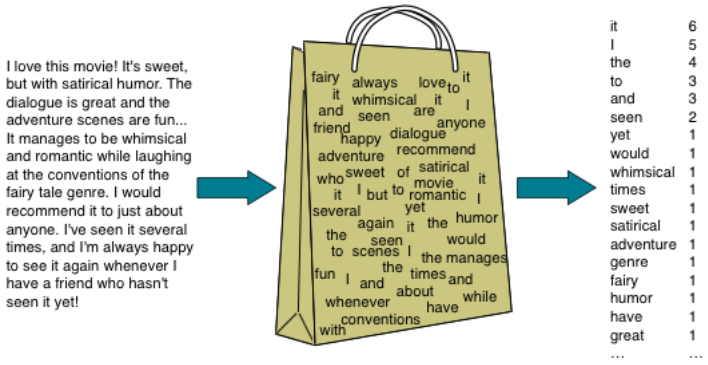


Figure 3.5: Bag of word example, taken from [32]

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 3.6: The Bow (or term-document) matrix for four words in four Shakespeare plays. Each cell contains the number of times the (row) word occurs in the (column) document. Taken from [32]

word (defined by the row) occurs in a specific document (defined by the column). Thus, *fool* appeared 58 times in the *Twelfth Night* document. With this model, each document is represented as a count vector, a column in figure 3.6. So *As You Like It* is represented as the list $[1, 114, 36, 20]$ (the first column vector in figure 3.6) and *Julius Caesar* is represented as the list $[7, 62, 1, 2]$ (the third column vector). In the example, the document vectors are of dimension 4 just so they fit on the page; in real term-document matrices, the vectors representing each document would have dimensionality $|V|$, the vocabulary size.

One important concept of this model is how to split the text documents into tokens that represents the words of the model. Most of the BoW algorithms use a tokenizer to implement the way to split the text. Two classic implementations are the n-gram and word tokenizer. The first one splits the text into tokens of a fixed length n-grams. For example, n-grams of size 1 represents each letter of the text. On the other side, the word tokenizer uses a set of characters known as the delimiters and spit the text each time one of this characters appears. For example to split an English document into its words, usually the space and punctuation signs are used as the delimiter characters (other variants for modern tokenization schemes will be presented in detail in

4.3.1).

The set of words/tokens used in the BoWs model is called the vocabulary and it can be fixed o learned during the training phase. In a fixed vocabulary approach, usually an expert uses its knowledge to define the set of features that better characterize the problem. Using this set of features, the BoW model captures from the input text the values of the features defined by the expert. The second approach corresponds to the case where the algorithm defines the vocabulary by it self. In this case, it uses the training set to learn the features to be used (this usually results into too many words).

This traditional model have some drawbacks: the size of the vector representation is as large as the vocabulary size and is sparse (has many zeros). Also, the semantic similarity of these feature vectors is pretty rigid and heavily relies on the literal lexicographical match between shared words [32].

Term Frequency - Inverse Document Frequency (TF-IDF) is a statistical measure that evaluates how relevant a word is to a document in a collection of documents. This is done by multiplying two metrics: how many times a word appears in a document, and the inverse document frequency of the word across a set of documents [79]. A common variant for the BoW approach consists of calculating the TF-IDF score for each token in the text instead of using the absolute frequency

TF-IDF was invented for document search and information retrieval. It works by increasing proportionally to the number of times a word appears in a document, but is offset by the number of documents that contain the word. So, words that are common in every document, such as *this*, *what*, and *if*, rank low even though they may appear many times, since they don't mean much to that document in particular. However, if the certain word appears many times in a document, while not appearing many times in others, it probably means that it's very relevant.

The TF-IDF score is calculated as follows, suppose a set of N documents $docs = (d_1, ..., d_N)$, lets define f_{ij} as the frequency of the term i in document j. The Term Frequency (TF_{ij}) is calculated as follows:

$$TF_{ij} = \frac{f_{ij}}{\max_k f_{kj}}. \tag{3.1}$$

If the word i appears in n_i of the N documents of the collection, the Inverse

Document Frequency (IDF_i) is defined as:

$$IDF_i = \log \frac{N}{n_i}. \tag{3.2}$$

Then the $TF - IDF$ score of the word i in the document j is calculated as: $TF_{ij} \times IDF_i$. The words with highest TF-IDF score are usually the words that better characterize the documents. It has many uses, one of them is use it as a text encoder like the BoW. Instead of using the absolute frequency (as in BoW) calculate the (TF-IDF) score for each token in the text. But, however it still suffers the same fundamental drawbacks such as high-dimensionality and sparsity like the BoW approach.

Vector semantics. Previous numeric representations of an input text (BoW and the TF-IDF variant) treat words as atomic units. They rely upon the one-hot encoding that seems to fail to encapsulate the true semantic meaning of the words. Now, we present approaches that use a distributed vector representation of words. First, we will show a high-level view of the distributed vector representation of words and then we show how to represent a text with this approach. In what follows we will expose the most common learning models to create them. These representations underlie the advanced deep language representation models like RoBERTa [43], the model that is proposed to use in this work to extract features (that will go in-depth in the next chapter).

As [32] explains, words that occur in similar contexts tend to have similar meanings. This link between similarity in how words are distributed and similarity in what they mean is called *the distributional hypothesis*. The hypothesis was first formulated in the 1950s by linguists like Joos [31], Harris [25], and Firth [80], who noticed that words which are synonyms (like *oculist* and *eye-doctor*) tended to occur in the same environment (e.g., near words like *eye* or *examined*) with the amount of meaning difference between two words *corresponding roughly to the amount of difference in their environments* (Harris, [25]).

The idea of vector semantics is to represent a word as a point in a multidimensional semantic space that is derived from the distributions of word neighbors. This representation is also known as an *embedding* or word representation. The term *embedding* derives from its mathematical sense as a mapping from one space or structure to another [32]. Research in NLP has compared the effectiveness of embedding methods for encoding semantic meaning. These

representations are used in every natural language processing application that makes use of meaning.

Figure 3.7: A two-dimensional (t-SNE) projection of embeddings for some words and phrases, showing that words with similar meanings are nearby in space. The original 60- dimensional embeddings were trained for sentiment analysis. Taken from [32].

Figure 3.7 shows a visualization of embeddings learned for sentiment analysis, showing the location of selected words projected down from 60-dimensional space into a two dimensional space. Notice the distinct regions containing positive words, negative words, and neutral words. *The fine-grained model of word similarity of vector semantics offers enormous power to NLP applications. NLP applications like the sentiment classifiers with the traditional approaches (like BoW) depend on the same words appearing in the training and test sets. But by representing words as embeddings, classifiers can assign sentiment as long as it sees some words with similar meanings* [32].

The word embeddings approach has proved to be a foundational building block for NLP and has inspired more sophisticated text representation schemes since them. For example, following these successful techniques, researchers have tried to extend the models to go beyond word level to achieve phrase-level or sentence-level representations such as the works presented in [33] and in [39].

The most straightforward approach to represent a text document may be using a weighted average (*centroid*) of the embeddings for all the words in the document. But it is possible to use another pooling technique such as the median or concatenate all the word embeddings in the document, as figure 3.8 shows. The *centroid* is the multidimensional version of the mean. Given k word embedding vectors $d = (w_1, w_2, ..., w_k)$ with dimensions d_m, the *centroid* is a vector $\hat{d} \in d_m$ that represent the document:

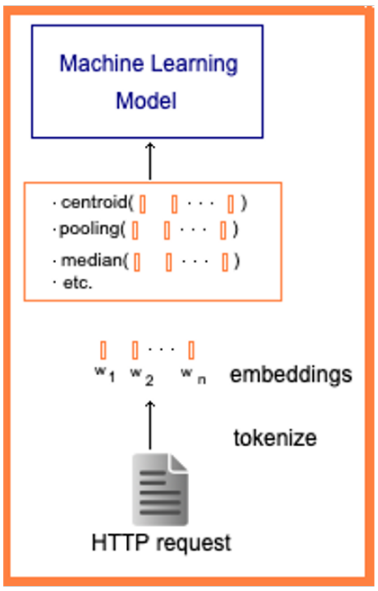


Figure 3.8: Example of a HTTP request encoding into a numeric vector with embeddings.

$$\hat{d} = \frac{w_1 + w_2 + ... + w_k}{k}. \tag{3.3}$$

In the previous approaches (like BoW or their variant TF-IDF), we saw how to represent a word (and a text) as a sparse, long vector with dimensions corresponding to words in the vocabulary. Then, a more powerful word representation was introduced: *embeddings* (short and dense vectors). Unlike the vectors that have been seen so far, *embeddings* are short, with the number of dimensions d ranging from $50 - 1000$, rather than the much more extensive vocabulary size $|V|$. Also, the vectors are dense: instead of vector entries being sparse, mostly-zero counts (or functions of counts), the values will be real-valued numbers that can be negative.

It turns out that dense vectors work better in every NLP task than sparse vectors. As explained in [32], *representing words as* $300 - dimensional$ *dense vectors requires our classifiers to learn far fewer weights than if we represented words as 50,000-dimensional vectors, and the smaller parameter space possibly helps with generalization and avoiding over-fitting. Also, dense vectors may also do a better job of capturing synonymy.*

Representation Learning. Word embedding models can be learned automatically from text without supervision. The *representation learning* task consist in automatically learning useful representations of words from a train-

ing input text. Finding such *self-supervised* ways to learn representations of the input, instead of creating representations by hand via feature engineering, is an important focus of NLP research [6]. Basically, word embedding models fall into two main categories. On one hand, *count-based* models use corpus-wide statistics such as words counts and frequencies to build word representations. On the other hand, *prediction-based* models learn embeddings by their predictive ability. A third category has appeared, *deep-contextual* models (like RoBERTa [43]), which have the particularity of using local context to create a word representation.

In this section we are going to present some of the more popular methods of the three categories. Then, in section 4 will delve into RoBERTa [43] model, that we propose to extract features from HTTP request that then will be used to train a classifier to detect attacks to web applications.

3.2.1. Count-based models

Count-based models are based on a *co-occurrence matrix*. This matrix is built by looping over a massive dataset ($\hat{d}$) composed of D textual documents ($d_1, ..., d_D$) in order to accumulate word co-occurrence counts (i.e., the number of times two or more words occur together in the dataset). In practice, there exists two types of co-occurrence: *word-document matrix* and *term-term matrix* (or *word-word matrix*). A *word-document matrix* $X \in N^{V \times D}$ (when V is the number of words in the vocabulary and D is the total number of documents in the dataset) is such that the entry X_{ij} corresponds to the number of times word i appears in document j. The *word-word matrix* $X' \in N^{V \times V}$ each cell records the number of times the row (target) word and the column (context) word co-occur in some context in the dataset $\hat{d}$. The context could be the document, in which case the cell represents the number of times the two words appear in the same document (for each one of the D documents). As in [32] explain it is most common, however, to use smaller contexts, generally a window around the word, for example of 4 words to the left and 4 words to the right, in which case the cell represents the number of times (in some training corpus) the column word occurs in such a ± 4 word window around the row word.

Figure 3.9 shows a simplified subset of the *word-word co-occurrence* matrix for four words computed from the *Wikipedia corpus* [15]. Note in figure 3.9 the

	aardvark	...	computer	data	result	pie	sugar	...
cherry	0	...	2	8	9	442	25	
strawberry	0	...	0	0	1	60	19	
digital	0	...	1670	1683	85	5	4	
information	0	...	3325	3982	378	5	13	

Figure 3.9: Co-occurrence vectors for four words in the *Wikipedia corpus*, showing six of the dimensions (hand-picked for pedagogical purposes). The vector for digitalis outlined in red. Note that a real vector would have vastly more dimensions and thus be much sparser [32].

two words *cherry* and *strawberry* are more similar to each other (both *pie* and *sugar* tend to occur in their window) than they are to other words like *digital*; conversely, *digital* and *information* are more similar to each other than, say, to *strawberry* (figure 3.10 shows a spatial visualization).

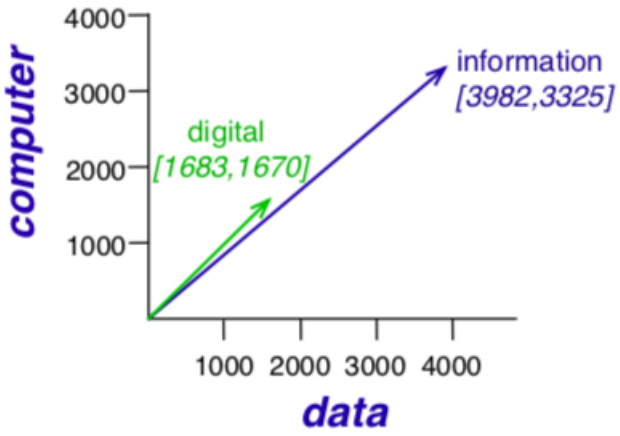


Figure 3.10: A spatial visualization of word vectors for *digital* and *information*, showing just two of the dimensions, corresponding to the words *data* and *computer* [32].

Note that $|V|$, the length of the vector, is generally the size of the vocabulary, usually between 10,000 and 50,000 words (using the most frequent words in the training corpus; keeping words after about the most frequent 50,000 or so is generally not helpful). The vector semantics model that has been described so far represents a target word as a vector with dimensions corresponding to all the words in the vocabulary (length $|V|$, with vocabularies of 20,000 to 50,000), which is also sparse (most values are zero). The values in each dimension are the frequency with which the target word co-occurs with each neighboring context word, there may be variants such as calculating the TF-

IDF (saw in the last section) weight instead of the absolute frequency. There are others alternative weighting function to TF-IDF, such as PPMI (positive pointwise mutual information) that we are not going to explain for reasons of extension, but that is very well presented in [32].

Once the co-occurrence matrix has been computed, its dimensionality is typically reduced using Singular Value Decomposition (SVD) such that X (respectively X') is factorized into USV^T, where U and V are orthonormal matrices. Finally, the rows of U are used as the word embeddings for all words in the vocabulary. Singular Value Decomposition (SVD) is a method for finding the most important dimensions of a data set, those dimensions along which the data varies the most. Well-known count-based models include Latent semantic indexing LSI) model [19] recast as LSA (latent semantic analysis) [16]. In LSA singular value decomposition (SVD) is applied to a *term-document matrix* (each cell weighted by log frequency and normalized by entropy), and then the first 300-dimensions are used as the LSA embedding.

A number of alternative matrix models followed on from the early SVD work, including Probabilistic Latent Semantic Indexing (PLSI) [27], Latent Dirichlet Allocation (LDA) [11], and Non-negative Matrix Factorization (NMF) [40]. As [32] explain, the LSA community seems to have first used the concept of *embedding* in the work presented in [19], in a variant of its mathematical meaning as a mapping from one space or mathematical structure to another. In LSA, the word embedding seems to have described the mapping from the space of sparse count vectors to the latent space of SVD dense vectors.

Although the *count-based* methods effectively leverage global statistical information, they are primarily used to capture word similarities and do poorly on tasks such as word analogy (this task will be explained at the end of the next sub-section 3.2.2), indicating a sub-optimal vector space structure [44]. By the next decade, Bengio et al. [6] showed that neural language models could also be used to develop *embeddings* as part of the task of word prediction (that is why its name *prediction-based*, which will be seen below).

3.2.2. Prediction-based models

The intuition of *prediction-based models* is that instead of counting how often each word w_i occurs near other (for example w_j) the idea is train a classifier on a prediction task: *Is word w_i likely to show up near w_j?*. Actually

this prediction task doesn't matter; instead, the learned classifier weights are taken as the *word embeddings*. As [32] explain, the revolutionary intuition here is that we can just use running text as implicitly supervised training data for such a classifier; a word w_i that occurs near the target word w_j acts as gold *correct answer* to the question *Is word* w_i *likely to show up near* w_j*?*. This method, often called **self-supervision**, avoids the need for any sort of *hand-labeled* supervision signal. This idea was first proposed in the task of **neural language modeling**, when Bengio et al. [6] and Collobert et al. [13] showed that a neural language model (a neural network that learned to predict the next word from prior words) could just use the next word in running text as its supervision signal, and could be used to learn an embedding representation for each word as part of doing this prediction task.

This section describes the simple feed-forward neural Language Model (LM): predicting upcoming words from prior word context, first introduced by Bengio et al. [6]. A feed-forward neural LM is a standard feed-forward network that takes as input at time t a representation of n previous words $(w_{t-n}, ..., w_{t-2}, w_{t-1})$ and outputs a probability distribution over the possible next word w_t. Thus, like the n-gram LM the feed-forward neural LM approximates the probability of a word given the entire prior context $P(w_t|w_1, ..., w_{t-1})$ by approximating based on the n previous words:

$$P(w_t|w_1, ..., w_{t-1}) \approx P(w_t|w_{t-n}, ..., w_{t-1}). \tag{3.4}$$

In neural language models, the prior context is represented by embeddings of the previous words. The model learns simultaneously (1) a distributed representation for each word along with (2) the probability function for word sequences, expressed in terms of these representations. Representing the prior context as embeddings, rather than by exact words as used in n-gram language models, allows neural language models to generalize to unseen data much better than n-gram language models [6]. Generalization is obtained because a sequence of words that has never been seen before gets high probability if it is made of words that are similar (in the sense of having a nearby representation) to words forming an already sentence.

Let us consider an example taken from [32], suppose it has been seen this sentence in training: *I have to make sure when I get home to feed the cat*, but it has never seen the word *dog* after the words *feed the*. Suppose for a

sentence in the test set the language model is trying to predict what comes after the prefix *I forgot when I got home to feed the*. An n-gram language model will predict *cat*, but not *dog*. However a neural LM, which can make use of the fact that *cat* and *dog* have similar *embeddings*, will be able to assign a reasonably high probability to *dog* as well as *cat*, merely because they have similar vectors. The idea is discovering some similarities between words to obtain generalization from training sequences to new sequences.

In a nutshell, the idea of the neural language models can be summarized as follows [6]:

1. associate with each word in the vocabulary a distributed *word feature vector* (a real valued vector in R^d),
2. express the joint *probability function* of word sequences in terms of the *feature vectors* of these words in the sequence, and
3. learn simultaneously the *word feature vectors* and the parameters of that *probability function*.

The key idea of the neural language model is find a representation for words that is helpful in representing compactly the probability distribution of word sequences from raw text [6]. The network learn jointly the representation (*words features*) and the prediction model. To illustrate these concepts, let us consider a toy example taken from [32] described in figure 3.11, that builds a feed-forward language model to estimate the probability of the word in time t (described as w_t), using the last three words $(w_{t-3}, w_{t-2}, w_{t-1})$ as context. This probability is described as $P(w_t = i|w_{t-3}, w_{t-2}, w_{t-1}; \vec{\boldsymbol{\theta}})$, where i could be any word of the vocabulary $|V|$ and $\vec{\boldsymbol{\theta}}$ are all the parameters of the network.

Following with the toy example of the figure 3.11 we assume (for simplicity) that only exist a training sequence (*for all the fish*) and $\vec{\boldsymbol{\theta}}$ represents all the network parameters. The optimization would be the following:

$$\arg\max_{\vec{\boldsymbol{\theta}}} P\left(w_4 = fish|w_1 = for, w_2 = all, w_3 = the; \vec{\boldsymbol{\theta}}\right). \tag{3.5}$$

Now, we shall delve into the architecture exposed in figure 3.11 (explaining it with the toy example). First, we will show all the layers of the network: the **input layer**, the **projection layer**, the **hidden layer** and the **output layer**. Then, we will explain (intuitively) how we can learn the parameters $\vec{\boldsymbol{\theta}}$ for the network, assuming that only the input sequence of the toy example is

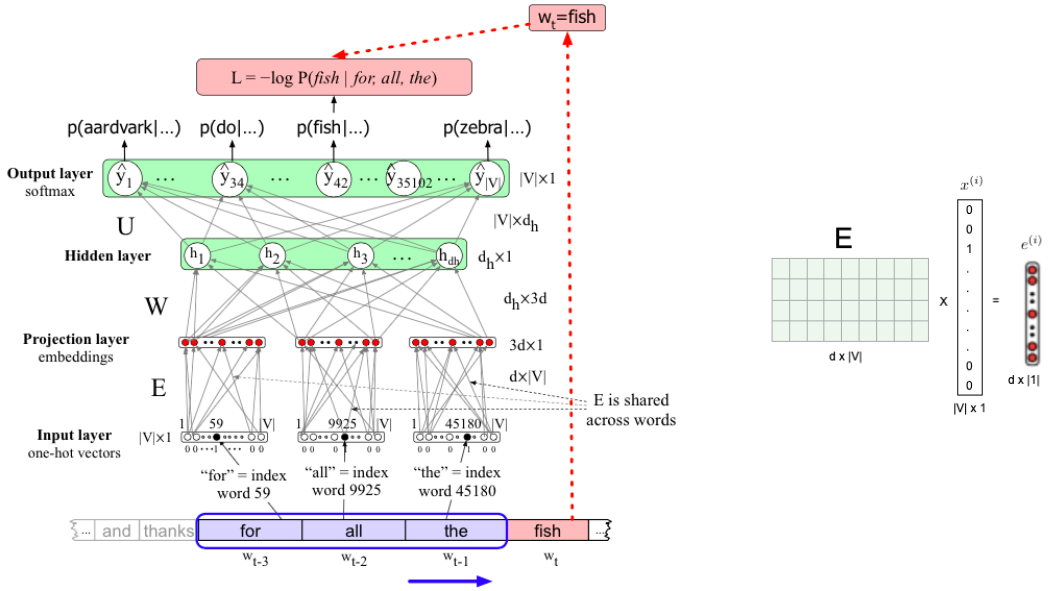


Figure 3.11: Left:Learning all the way back to embeddings. Notice that the embedding matrix E is shared among the 3 context words [32]. Right: Embedding matrix.

available. Then, we will show the general **training process** with a complete training set.

1. **Input Layer**. The context words of the example ($w_1 = for, w_2 = all, w_3 = the$) are first described as one-hot vectors $x^{example} = (x_{59}, x_{9925}, x_{45180})$ the index correspond the position of word in the vocabulary V. For example, if the word *for* is the index 59 in the vocabulary, this word is represented as a one-hot vector of dimension $R^{|V|\times 1}$ (with all zeros and a one in the position 59).
2. **Projection Layer**. This layer is the key part of the architecture, because allows the network to represent the context words (first represented as one-hot vectors $x_i \in R^{|V|\times 1}$) to real valued vectors $e_i \in R^{d\times 1}$. The procedure is really simple and consist on multiply a weighted embedding matrix $E \in R^{d\times|V|}$ against each one-hot vector $x_i \in R^{|V|\times 1}$ to place it into a different sub-space in $R^{d\times 1}$, as figure 3.12 shows. This simple (but powerful) technique allows to map a vector representation of an object (for example a one-hot vector $x_i \in R^{|V|\times 1}$) to a real valued vector $e_i \in R^{d\times 1}$.

   The embedding matrix $E \in R^{d\times|V|}$ has a column for each word, each of d

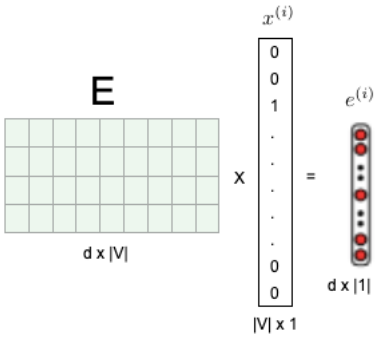


Figure 3.12: Embedding matrix of the feed-forward language model.

dimensions, and hence has dimension $d \times |V|$. The weights of this matrix are initialized randomly and are updated during training. Once the one-hot context vectors $x_i \in R^{|V|\times 1}$ are mapping into real valued vectors embeddings $e_i \in R^{d\times 1}$ then pass to the next layer of the network. For example, given the three context words x=(*for, all the*), the model first create a one-hot vectors $x_i \in R^{|V|\times 1}$ for each one. Then multiply the embedding matrix $E \in R^{d\times|V|}$ by each one-hot vector x_i to get the real valued vector e_i in R^d for each token. For example, to get the embedding of the word *the* (index 45180 in the vocabulary) the embedding matrix $E \in R^{d\times|V|}$ is multiplied by the one-hot vector for the word *the* (a vector of dimension $|V|$ with all zeros and a one in the position 45180). Doing this calculation for each token $(x_{59}, x_{9925}, x_{45180})$ allows us to obtain the three real-valued vectors $(e_{59}, e_{9925}, e_{45180})$ each one with dimension $R^{d\times 1}$. The next step is to concatenate the three built-in context words $e = (e_{59}, e_{9925}, e_{45180}) \in R^{3d\times 1}$ and pass it as input for the next hidden layer.

3. **Hidden Layer.** This layer consists in multiply $W \in R^{d_h\times 3d}$ by $e = (e_{59}, e_{9925}, e_{45180}) \in R^{3d\times 1}$ (and add $b \in R^{d_h\times 1}$) and pass through an activation function to get the hidden layer $h \in R^{d_h\times 1}$.
4. **Output Layer.** The output layer consists in multiply $U \in R^{|V|\times d_h}$ by $h \in R^{d_h\times 1}$ resulting in a vector $z \in R^{|V|\times 1}$. However, z can't be the classifier's output since it's a vector of real-valued numbers, while what the model need for classification is a vector of probabilities. There is a convenient function, called *softmax*, for normalizing a vector of real values. Normalizing is converting it to a vector $\hat{y} \in R^{|V|\times 1}$ that encodes a probability distribution (all the numbers lie between 0 and 1 and sum

to 1). For a vector $z \in R^{|V|\times 1}$, the softmax is defined as:

$$\hat{y}_i = softmax(z_i) = \frac{\exp z_i}{\sum_{j=1}^{|V|} \exp z_j} \; 1 \leq i \leq |V|. \tag{3.6}$$

Each node $\hat{y}_i$ in the output vector $\hat{y}$ estimates the probability $P(w_t = i|w_{t-1}, w_{t-2}, w_{t-3}; \vec{\boldsymbol{\theta}})$ when i represent each word of the vocabulary $|V|$ and $\vec{\boldsymbol{\theta}} = (E, W, b, U)$ are all the parameters of the model. The prediction depends on the context words x=$(w_{t-3}, w_{t-2}, w_{t-1})$ and the parameters $\vec{\boldsymbol{\theta}}$. The probability vector $\hat{y}$ is the output of parametric function $f(x, \vec{\boldsymbol{\theta}})$. The model now needs to set up all the parameters $\vec{\boldsymbol{\theta}}$.

Training. To explain the training step intuitively, we will continue with the toy example and the equation 3.5:

$$\arg\max_{\vec{\boldsymbol{\theta}}} P\left(w_4 = fish|w_1 = for, w_2 = all, w_3 = the; \vec{\boldsymbol{\theta}}\right). \tag{3.7}$$

For the sequence of the example $(w_1 = for, w_2 = all, w_3 = the, w_4 = fish)$, this equation can be rewritten with the concepts that we saw until now. The context words of the example $(w_1 = for, w_2 = all, w_3 = the)$ are first described as one-hot vectors $x = (x_{59}, x_{9925}, x_{45180})$ also the word that the model try to predict $(w_4 = fish)$ can be expressed in a one-hot vector. This word can be described as a vector $y \in R^{|V|\times 1}$ with all zeros and a one on the position 42 (the index of the word *fish* in the vocabulary). Therefore, the equation 3.7 can be expressed as a parametric function:

$$\arg\max_{\vec{\boldsymbol{\theta}}} \; y \approx \hat{y} = f(x, \vec{\boldsymbol{\theta}}). \tag{3.8}$$

For the context words $x = (x_{59}, x_{9925}, x_{45180})$ in the example and the initial parameters $\vec{\boldsymbol{\theta}}$, the network output a vector $\hat{y} = f(x, \vec{\boldsymbol{\theta}})$. For example, suppose that in the position 42 (corresponding to the word *fish*) of the output vector $\hat{y}$ there are a 0.75. That means that for a context words $x = (x_{59}, x_{9925}, x_{45180})$ the network predict that there are 75% of probability that the next word is *fish*. This vector of probabilities is described as:

$$\hat{y}^T = [0.05, 0.01, ..., 0.75, 0, 0]. \tag{3.9}$$

To find the best parameters $\vec{\boldsymbol{\theta}}$, the model first need a metric to measure

how close the prediction $\hat{y} \in R^{|V|\times 1}$ is to the true gold label $y \in R^{|V|\times 1}$ (a kind of *distance* between the prediction output and the gold output) as figure 3.13 shows.

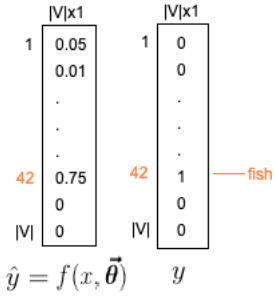


Figure 3.13: The prediction output $\hat{y} \in R^{|V|\times 1}$ and the gold output $y \in R^{|V|\times 1}$ vectors

This measure is called the *loss function* or the *cost function* $L(\vec{\theta})$. There are many ways to measure the *loss function*. One widely used in classification problems is the *cross entropy loss function* (L_{CE}), defined as:

$$\begin{aligned} L(\vec{\theta}) &= L_{CE}(y, f(x, \vec{\theta})) \\ &= -\langle y, \log\ f(x, \vec{\theta})\rangle. \end{aligned} \tag{3.10}$$

Once it has been defined the *loss function*, the optimization process consist into find the optimal $\vec{\theta}$ that minimizes it:

$$\arg\min_{\vec{\theta}} L_{CE}(y, f(x, \vec{\theta})). \tag{3.11}$$

General training process. Generally, training proceeds by taking as input a corpus with sentences in raw text, for example D=[[*the cat is on the mat*],...,[*the dog is cute*]] and tokenize it, resulting in a set D=[[*the, cat, is, on, the, mat*],...,[*the, dog, is, cute*]]. Then, with this tokenized set is build (automatically) a new set of sequences to train the network $D^{train} = [(x^{(1)} = \{the, cat, is\},\ y^{(1)} = on), (x^{(2)} = \{cat, is, on\},\ y^{(2)} = the), ..., (x^{(m)} = \{the, dog, is\},\ y^{(m)} = cute)]$.

The training consist on starting with random weights $\vec{\theta}$, and then iteratively try to predict the word in time t (w_t) using the context words ($w_{t-3}, w_{t-2}, w_{t-1}$) for each training sequence in D^{train} (in this example, three words were used as context, but this can vary). The prediction of the word in time t is compared with the real word. The *cross-entropy loss* (explained before) is used

to measure how close the prediction is to the true gold label. The goal is to find the set of weights ($\vec{\boldsymbol{\theta}}$) which minimizes the *loss function*, averaged over all examples in D^{train}:

$$\hat{\theta} = \frac{1}{m}\sum_{i=1}^{m} \arg\min_{\vec{\boldsymbol{\theta}}} L_{CE}(y^{(i)}, f(x^{(i)}, \vec{\boldsymbol{\theta}})). \tag{3.12}$$

A well-know method used to finds a minimum of a function is called stochastic gradient descent (SGC). The method consist in figuring out in which direction (in the space of the parameters $\vec{\boldsymbol{\theta}}$) the function's slope is rising the most steeply and moving in the opposite direction. The intuition taken textually from [32] is: *if you are hiking in a canyon and trying to descend most quickly down to the river at the bottom, you might look around yourself 360 degrees, find the direction where the ground is sloping the steepest, and walk downhill in that direction*, the algorithm explained at high level is exposed in the following figure taken from [32].

function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x, y) **returns** θ
where: L is the loss function
f is a function parameterized by θ
x is the set of training inputs $x^{(1)}$, $x^{(2)}$,..., $x^{(m)}$
y is the set of training outputs (labels) $y^{(1)}$, $y^{(2)}$,..., $y^{(m)}$

$\theta \leftarrow 0$
repeat til done # see caption
For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)
1. Optional (for reporting): # How are we doing on this tuple?
Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$ # What is our estimated output $\hat{y}$?
Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)}$ from the true output $y^{(i)}$?
2. $g \leftarrow \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss?
3. $\theta \leftarrow \theta - \eta\, g$ # Go the other way instead
return θ

Figure 3.14: The stochastic gradient descent algorithm. Step 1 (computing the loss) reports how well we are doing on the current tuple. The algorithm can terminate when it converges (or when the gradient norm $< \epsilon$) or when progress halts (for example, when the loss starts going up on a held-out set) [32].

Training thus not only sets the weights W and U of the network, but also as we're predicting upcoming words, we're learning the embeddings E for each words that best predict upcoming words [32]. In the training step, all the parameters $\vec{\boldsymbol{\theta}} = (E, W, U)$ of the model are learned. Training the parameters to minimize the *loss* will result in an algorithm for language modeling (a word predictor) and a new set of embeddings E that can be used as word represen-

tations for other tasks. Finding such self-supervised ways to learn representations of the words, instead of creating representations by hand via feature engineering, is an important focus of NLP research [6].

The classic feed-forward language networks (that we explained in this section) have been progressively replaced with recurrent neural networks [49] and long short-term memory networks [23] for language modeling. However, the general building blocks of Bengio et al.[6] (the classic feed-forward LM that we saw in this section) are still found in most neural language and word embeddings models nowadays [44].

Semantic properties of embeddings.

As in [32] explain, it's useful to distinguish two kinds of similarity or association between words [66]. Two words have **first-order co-occurrence** (sometimes called syntagmatic association) if they are typically nearby each other. Thus *wrote* is a first-order associate of *book* or *poem*. Two words have **second-order co-occurrence** (sometimes called paradigmatic association) if they have similar neighbors. Thus *wrote* is a second-order associate of words like *said* or *remarked*.

Word analogy

Another semantic property of *embeddings* is their ability to capture **relational meanings**. In an important early vector space model of cognition, [63] proposed the parallelogram model for solving simple analogy problems of the form a is to b as a^* is to what?. In such problems, a system given a problem like *apple* : *tree* :: *grape* :?, i.e., *apple* is to *tree* as *grape* is to ____, and must fill in the word *vine*. In the parallelogram model, illustrated in figure 3.15, the vector from the word *apple* to the word *tree*($=\overrightarrow{apple} - \overrightarrow{tree}$) is added to the vector for grape($\overrightarrow{grape}$); the nearest word to that point is returned.

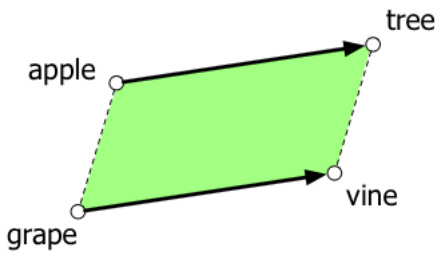


Figure 3.15: The parallelogram model for analogy problems [63]: the location of $\overrightarrow{vine}$ can be found by subtracting $\overrightarrow{tree}$ from $\overrightarrow{apple}$ and adding $\overrightarrow{grape}$.

For example, the result of the expression $\overrightarrow{king} - \overrightarrow{man} + \overrightarrow{woman}$ is a

vector close to $\overrightarrow{queen}$. Similarly, $\overrightarrow{Paris} - \overrightarrow{France} + \overrightarrow{Italy}$ results in a vector that is close to $\overrightarrow{Rome}$. For a $a : b :: a^* : b^*$ problem, meaning the algorithm is given a, b, and a^* and must find b^*, the parallelogram method is thus:

$$\hat{b}^* = \arg\max_x \ distance(x, a - a^* + b). \tag{3.13}$$

with the distance function defined either as cosine or as Euclidean distance.

In the present decade, with the rapid development in deep learning, the research of static word embedding took off with models like: Word2Vec [39] (Context Bag of Words and Skip-gram), fastText [12] and GloVe [56]. However, because they create a single representation for each word, a notable problem with static word embeddings is that all senses of a polysemous [1] word must share a single vector. Therefore, given the limitations of static word embeddings, recent work has tried to create context-sensitive word representations.

3.2.3. Contextual word embeddings

Until now, all word embeddings techniques were missing a crucial element for fully capturing semantic and syntactic meanings of words: *local context*. These methods actually learn to capture the general (most common) context of words in their representations. This posed several problems, most notably that all senses of a polysemous word had to share the same representation. For example, consider the word *mouse*. It has multiple word senses, one referring to a *rodent* and another to a *device*. The classic static embedding approach map the word *mouse* to a unique real-valued vector.

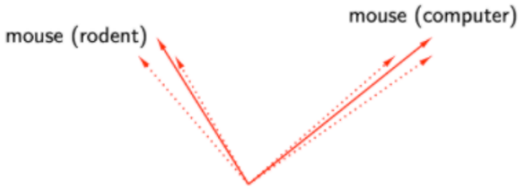


Figure 3.16: Contextual representations create one representation of *mouse* per word sense [20]

To address this problem, researchers have taken an increasing interest in

[1] Polysemous words are words with two or more meanings (for example, *run* is the largest polysemous entry in the Oxford English Dictionary with 645 meanings [44].)

models capable of creating *contextual word representations*. The idea is simple: a token is assigned a representation that is a function of the entire input sentence. These models can effectively create one representation of the same word (for example, *mouse*) per word sense as figure 3.16 shown.

Replacing static word embeddings with contextualized word representations has yielded significant improvements on many NLP tasks [20]. Early work focused on learning context-dependent representations included CoVe [47], ELMo [57] and ULMFiT [28], that all rely on bidirectional Long Short-Term Memory (LSTM) neural networks to encode the context. LSTM is a reccurrent neural network architecture that unlike standard feed-forward neural networks has feedback connections. More recently the introduction of the Transformer architecture [78] resulted in a series of powerful pre-trained language representations such as BERT [17], XLNet [81] and RoBERTa [43], that proved their effectiveness in a wide variety of language tasks. Transformers is an attention-based architecture for modeling sequential data which is an alternative to recurrent neural networks and is capable of capturing long range dependencies in sequential data.

In this work we propose the use of transformer-based contextualized representation of HTTP requests to extract feature vectors that then will be used to train a classifier to detect attacks to web applications. In a first step we create a RoBERTa [43] language model from scratch using a set of HTTP requests from the web application that we aim to protect. In a second step, we use the *feature-based* strategy to transform each HTTP request into a feature vector. Once we have obtained the pre-trained model, we convert each HTTP request into a numeric representation using the weights of the last layer of the network, also known as *feature extraction*. With these representations as input, we build a One-Class Classification model (OCC).

The next chapter explains RoBERTa [43], the transformer-based language model which is at the core of this thesis. It describes the model architecture, its pre-training procedure, and the application to other downstream tasks.

Chapter 4

A RoBERTa-based language model

This chapter explains the language representation model, which is at the core of this thesis. This model relies on a two-step learning framework. First, it learns contextual token representations from a large amount of text (or a set of HTTP requests) in a *self-supervised* way. This stage is commonly referred as pre-training. Then, these pre-trained representations can be applied to downstream tasks by choosing between two learning strategies: *feature-based* and *fine-tuning*. The model architecture used to train it is called *Robustly Optimization Bidirectional Encoder Representations from Transformers* (RoBERTa) [43]. As already mentioned, the architecture comes directly from the transformer model proposed by [78], it is just the encoder portion of the model.

Figure 4.1 shows the two-step learning framework that was presented in section 1.2. In the first step, a RoBERTa language model is created using a set of HTTP requests. In a second step, the *feature-based* strategy is used to transform the HTTP request into a feature vector. With these representations as input, a One-class classification model (OCC) is performed to predict if the request is normal or not. This chapter aims to delve into the internal architecture of the RoBERTa language model. First, we shall show the key components to get a high-level view, and then each one of them will be explored in-depth.

The internal architecture of the transformer encoder is composed of a stack of $\boldsymbol{L}$ identical encoder layers (the left side of figure 4.2 shows an example

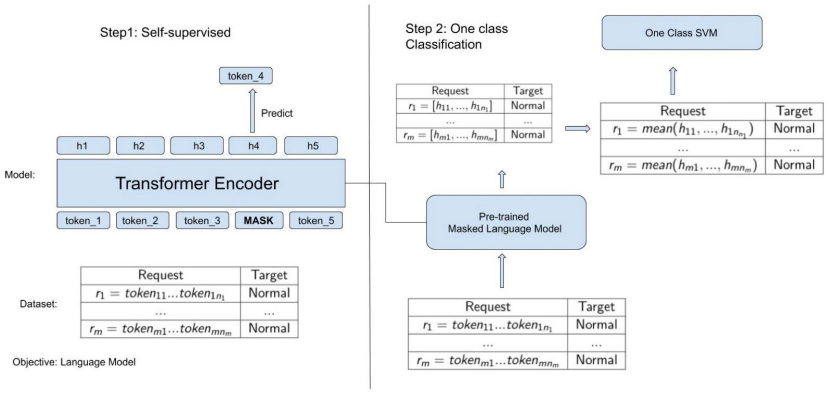


Figure 4.1: Two-step learning framework. Left: transformer encoder used to extract the contextual representation of each token $token_{ij}$ in the request r_i. Right: each request r_i is represented as the mean of token deep contextualized representation h_{ij} and how they are used to train a one-class classifier.

of a stack of 12 encoder layers). Each encoder layer contains two types of sub-layers, as the right side of of figure 4.2 exhibits. The first one is a multi-head self-attention layer (the essential piece of the internal architecture) which helps to look at other tokens in the sequence while encoding a specific token. The second sub-layer is simply a feed-forward network (FFN) applied to each position separately and identically.

The encoder is optimized with a *self-supervised* technique, a mechanism that has dramatically improved the state-of-the-art of a wide variety of NLP tasks. The terminology *self-supervised* means that supervisions used for training the network are automatically collected from raw data without manual annotation. For that reason, training this model can use considerable amounts of training data.

The *self-supervised* strategy that RoBERTa uses is based on the Masked Language Modeling (MLM) objective, which consists of randomly masking words from an input text and training a model to recover the original input. The MLM objective allows the model to make predictions by leveraging left and right contexts. The procedure consists of recovering the initial sentence (or the HTTP request) given the masked one. In contrast to denoising-autoencoders, these models only predict the masked words rather than reconstructing the entire input [17]. Despite being trained with only a language modeling goal, the network can learn highly transferable and task-agnostic properties of the language [20]. In an attempt to predict the masked tokens, the model should

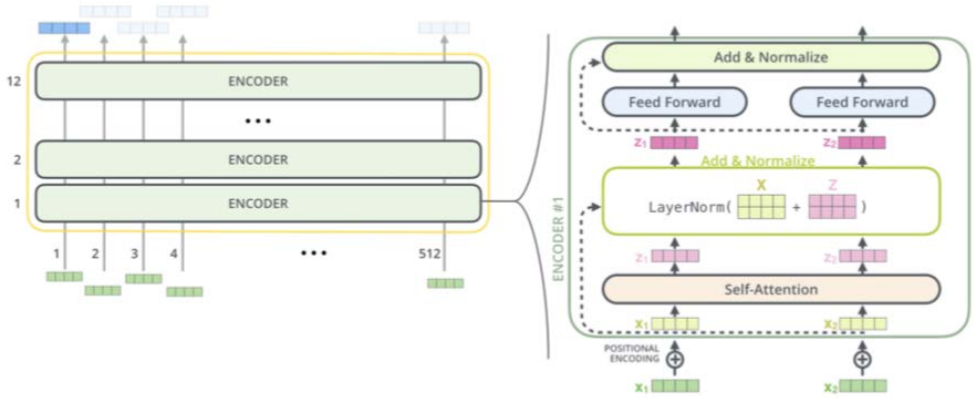


Figure 4.2: Internal architecture of the Transformer Encoder (Jay Alammar, 2018 [73]). Left: the transformer encoder composed of a stack of $\boldsymbol{L = 12}$ identical encoder layers. Right: the self-attention and the feed-forward sub-layers of each encoder layer.

extract some information from the language, not only structural information but also some semantic information. This information is encoded in the weights of the encoding layers [20].

Figure 4.3 shows the training procedure explained above step by step using the model architecture as a reference. The first thing that the encoder does is splitting the raw text into tokens and convert each one into a real-valued vector; these vectors are the encoder's inputs. The encoder is composed of a stack of encoder layers with their respective sub-layers, self-attention and feed-forward. The learning task consists of masking some tokens of the input, recovering the numeric representation of them (based on this context), and predicting which is the original token masked out. Assuming a very simplified example of a sequence $x = (x_1, ..., x_N)$ of N tokens where $\vec{\boldsymbol{\theta}}$ represents all the network parameters of the encoder. In the case of the toy example of figure 4.3 with a sequence of three tokens $x = (x_1, x_2, x_3)$, the optimization would be the following:

$$\arg\max_{\vec{\boldsymbol{\theta}}} P\left(x_2 | x_1, x_3; \vec{\boldsymbol{\theta}}\right). \tag{4.1}$$

That is, the result of the learning process consists of finding the value of the parameters $\vec{\boldsymbol{\theta}}$ that maximizes the probability of predicting the original value of the masked element x_2 using x_1 and x_2 as context.

The remainder of this chapter present each part of the internal architec-

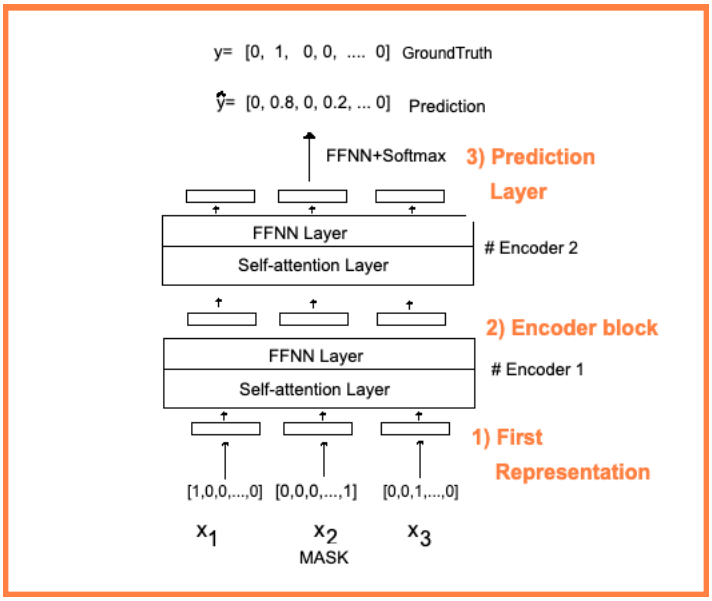


Figure 4.3: General architecture of the transformer encoder together with the MLM training model illustrated.

ture of the RoBERTa model. The most important part is the *self-attention* mechanism and is the first part that will be described in section 4.1. It will be then explained how this mechanism fits into the whole architecture. Section 4.2 describes the other layers of the model (such as the feed-forward). Then in section 4.3 we will go in-depth into the input representation (the process of *tokenization* and the first real-valued vector representations). The *self-supervised* learning procedure is presented in section 4.4. Finally, in section 4.5 we will explain how to apply this architecture to downstream tasks.

4.1. Self-Attention

The transformer encoder is built of stacks of network layers consisting of simple linear layers, feed-forward networks, and custom connections around them. In addition to these standard components, the key innovation of transformers is the use of self-attention layers [32]. This section describes how self-attention works and how to fit it into larger transformer blocks. Self-attention allows a network to directly extract and use information from arbitrarily large contexts without the need to pass it through intermediate recurrent connections as in Recurrent Neural Networks (RNNs).

In order to focus on the attention mechanism, it is going to be assumed that

the input sequence, which can be a sentence of words (or an HTTP request), is a sequence of tokens. Also, it will be assumed that each token is represented as a real-valued vector of dimension R^{d_m}. Later on, in section 4.3 we will detail how the tokenization process is performed and how each token (a discrete symbol in a dictionary) is converted into a real-valued vector to be processed by the self-attention layer. For the moment, it is assumed that a sentence with words (or an HTTP request) is represented by a sequence of N real-valued vectors $(x_1, ..., x_N)$ of dimension d_m, where N is the number of tokens.

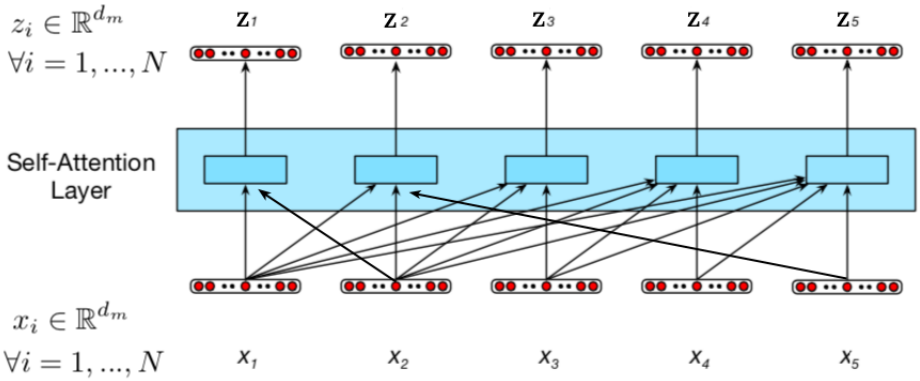


Figure 4.4: Information flow in the self-attention layer. When processing each element of the sequence (x_i), the model attends to all the inputs including the current one to obtain the new representation (z_i) [32].

The encoders' input first goes through a self-attention layer (as the figure of the whole architecture 4.3 shows). The main goal of this step is to introduce contextual information in the encoding process. Thus, while it encodes a given token in the sentence, it considers other tokens as contextual information. The self-attention layer maps input sequences $(x_1, ..., x_N)$ $x_i \in R^{d_m}$ to output sequences of the same length $(z_1, ..., z_N)$ $z_i \in R^{d_m}$, as figure 4.4 exhibits. When processing each real-valued token $x_i \in R^{d_m}$ in the input, the model has access to all of the tokens in the input sequence, including the one under consideration. In this way, the new representation combines information of the specific token and its context. As we can see in figure 4.4 each new representation z_i depends on the whole input sequence $(x_1, ..., x_N)$.

The attention mechanism's prominent ability is to compare a token of interest, for example the x_i vector, against a collection of other tokens in the sequence and reveal their respective relevance [32]. In the self-attention layer, the set of comparisons are to other tokens within a given input sequence $(x_1, ..., x_N)$. These comparisons, calculated as compatibility scores, help to

compute the new representation $z_i \in R^{d_m}$, of the current token $x_i \in R^{d_m}$. These scores mean how much to attend to the tokens in the context.

The simplest form of getting a compatibility score, between x_i and all the others tokens in the sequence, is a dot product. As [78] mention, there are other ways to calculate these scores, such as additive attention [5] in which the compatibility function is calculated with a feed-forward neural network. Nevertheless, [78] explains that while the two methods are similar in theoretical complexity, the dot product is much faster and more space-efficient in practice. For that reason, the original transformer implementation uses the dot-product score to calculate the compatibility between x_i and all the others tokens in the sequence $(x_1, ..., x_N)$, as seen in the following equation:

$$score(x_i, x_j) = \langle x_i, x_j \rangle \qquad \forall j = 1, ..., N. \tag{4.2}$$

The result of the dot product is a scalar value that measures the similarity between vectors. Similar vectors produce scores with large magnitudes. Then, a normalization process, with the *softmax* function, is carried out in order to create a vector of weights $\alpha_i = (\alpha_{i1}, ..., \alpha_{iN})$ where each $\alpha_{ij} \in [0, 1]$, $\forall j = 1, ..., N$ and $\sum_{j=1}^{N} \alpha_{ij} = 1$. Each α_{ij} indicates the proportional relevance of each context tokens $(x_1, ..., x_N)$ to the input token x_i (the current focus of attention).

$$\begin{aligned} \alpha_{ij} &= softmax(score(x_i, x_j)) \qquad \forall j = 1, ..., N. \\ &= \frac{\exp^{score(x_i, x_j)}}{\sum_{j=1}^{N} \exp^{score(x_i, x_j)}} \qquad \forall j = 1, ..., N. \end{aligned} \tag{4.3}$$

Given these proportional scores (α_{ij}) an output value z_i is generated by taking the sum of all context tokens $(x_1, ..., x_N)$ values weighted by their respective α_{ij} score.

$$z_i = \sum_{j=1}^{N} \alpha_{ij} x_j \qquad \forall i = 1, ..., N. \tag{4.4}$$

The steps embodied in equations 4.2 through 4.4 represent the core of an attention-based approach: a set of comparisons to relevant tokens in some context, a normalization of those scores (to provide a probability distribution), followed by a weighted sum using this distribution. The output z_i $\forall i = 1, ..., N$

is the result of this straightforward computation.

As explained in [32] this simple mechanism unfortunately provides no opportunity for learning, because everything is directly based on the original real-valued input vectors $(x_1, ..., x_N)$. In particular, there are no opportunities to learn the diverse ways that tokens can contribute to the representation of other tokens in large sequence inputs. To allow for this kind of learning, transformers include additional parameters in the form of a set of weight matrices that operate over the real-valued input vectors $(x_1, ..., x_N)$. That is, the real-valued input vectors are projected onto several different vector spaces. A projection is a simple (but powerful) technique in machine learning where a vector representation of an object is multiplied by a weighted matrix to project it into a different sub-space.

Each real-valued token representation (x_i) of the sequence $(x_1, ..., x_N)$ are projected in three different vector spaces. The authors of the original transformer architecture [78] used the names *query*, *key* and *value* for these new sub-spaces, in order to define different roles that each token (x_i) of the input $(x_1, ..., x_N)$ plays during the course of the attention process. The names *query*, *key* and *value* are concepts that come from *retrieval systems*, there are abstractions defined by the authors of [78] to calculate and think about the attention mechanism.

As already explained, the attention mechanism's prominent ability is to compare a token of interest (the current focus of attention), for example the x_i real-valued vector, against a collection of other tokens in the sequence $(x_1, ..., x_N)$ and reveal their respective relevance. As follows, we shall explain briefly the different roles that each projection plays during the attention process. Then, we shall expose how attention is calculated; this will clarify the role each of these projections plays.

- The *query* represents the projection of the current token x_i (that we aim to get a new representation) used to score against all the other tokens in the context $(x_1, ..., x_N)$ to reveal their respective relevance.
- Every token in the context is projected onto the *key* space to calculate the relevance of each one against the *query*. As already mentioned, the relevance is calculated as a compatibility score with the dot product.
- Finally, *value* vectors are actual token representations. Once the relevance of each context token has been calculated (in the form of compat-

ibility scores), the next step consists on adding the *values* that represent these tokens to get a new representation. Their respective compatibility score weights the summation.

In order to define mathematically the different roles that play the input tokens in each step introduced before, transformers introduce three weights matrices W^Q, W^K, and W^V. These matrices are initialized randomly and updated (learned) during training, as we will show in section 4.4. Given each input value x_i of dimension R^{d_m} and the three matrices with dimensions: $W^Q \in R^{d_m \times d_q}$, $W^K \in R^{d_m \times d_k}$ and $W^V \in R^{d_m \times d_v}$. The projection of the tokens (represented as a real-values vectors) into the new projection spaces are as follows:

$$q_i = x_i^T W^Q; \quad k_i = x_i^T W^K; \quad v_i = x_i^T W^V \qquad \forall i = 1, ..., N. \tag{4.5}$$

Given these projections, the score between a current focus of attention (x_i) and the tokens in the input context ($x_1, ..., x_N$) consists of a dot product between its *query* vector (q_i) and the *key* vectors ($k_1, ..., k_N$). The updated version of the score showed in equation 4.2 that allows the comparison between the tokens is:

$$score(x_i, x_j) = \langle q_i, k_j \rangle \qquad \forall j = 1, ..., N. \tag{4.6}$$

The ensuing *softmax* calculation resulting in α_{ij} remains the same

$$\alpha_{ij} = softmax(score(x_i, x_j)) \qquad \forall j = 1, ..., N. \tag{4.7}$$

The output calculation for z_i is now based on a weighted sum over the real-valued vectors ($v_1, ..., v_N$).

$$z_i = \sum_{j=1}^{N} \alpha_{ij} v_j. \tag{4.8}$$

Figure 4.5 shows a simple example when the input sequence is $x = (x_1, x_2, x_3)$ and shows how to compute z_3, the new third token representation of the sequence using the self-attention mechanism.

1. The first step consists of generating the *query*, *key* and *value* vectors

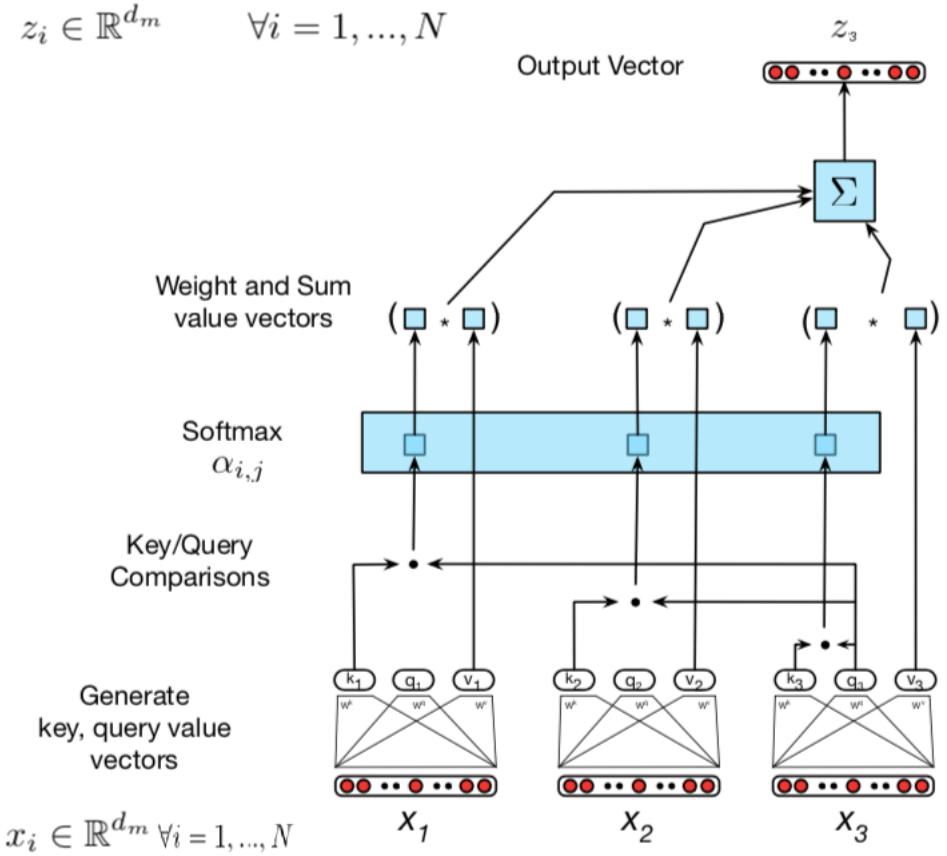


Figure 4.5: Calculation of the value of the third token (x_3) of a sequence using self-attention [32].

with the weight matrices (W^Q, W^K, and W^V) introduced before:

- The *query* is calculated as: $q_3 = x_3^T W^Q$.
- The *keys* are calculated as: $k_1 = x_1^T W^K$, $k_2 = x_2^T W^K$, $k_3 = x_3^T W^K$.
- Finally, the *values* are calculated as: $v_1 = x_1^T W^V$, $v_2 = x_2^T W^V$, $v_3 = x_3^T W^V$.

2. The second step embodies calculating the relevance of each context tokens (with the *key* projections) against the *query* projection q_3, this is calculated with three scores: $score(x_3, x_1) = \langle q_3, k_1 \rangle$, $score(x_3, x_2) = \langle q_3, k_2 \rangle$ and $score(x_3, x_3) = \langle q_3, k_3 \rangle$.
3. The next step consist of transforming each compatibility score to weights:

- $\alpha_{31} = softmax(score(x_3, x_1))$.
- $\alpha_{32} = softmax(score(x_3, x_2))$.
- $\alpha_{33} = softmax(score(x_3, x_3))$.

4. The final step comprises of calculating the weighted sum (or convex combination) of the tokens representations (*values*). The weights was calculated in the previous step (with the compatibility scores). Therefore, the weighted sum is:

$$z_3 = \alpha_{31} v_1 + \alpha_{32} v_2 + \alpha_{33} v_3. \tag{4.9}$$

Note that if there would exist a token x_4 when computing the output of x_3, the mechanism also allows taking it into account. The attention is calculated using all the tokens in the sentence and not just the tokens preceding the given token. Therefore, the transformer encoder provides RoBERTa its *bidirectional* nature, as it uses tokens from both directions to attend a given token.

The self-attention layer lies at the core of what's called a transformer block. In addition to the self-attention layer, each transformer block includes feed-forward layers, residual connections, and normalizing layers (that will be detailed in 4.2). The main idea is to build $\boldsymbol{L}$ blocks that can then be stacked, the output of the block i is the input of the block $i+1$. Figure 4.6 illustrates a typical stack of $\boldsymbol{L}$ transformer blocks (with their respective sub-layers).

Note that while the attention mechanisms are the same across different blocks, the transformer uses different parameters from block to block. That is, each attention layer has its own weights matrices. For example, the attention layer of the encoder block n has its own set of weighted matrices (W^{Q_n}, W^{K_n}, and W^{V_n}); all of them are learned during training (as we will show in section 4.4).

So far, we described the simplest version of the self-attention method. However, there are some practical concepts to make the approach more powerful which will be discussed in what follows. One practical consideration arises in computing the weights α_{ij}. Until now, we used a dot product as a comparison mechanism in combination with the exponential as described in equation 4.3. The result of the dot product can be an arbitrarily large (positive or negative) value. As [32] explains, the application of the exponential function to such large values can lead to numerical issues and to an effective loss of gradients during training. To avoid this, the dot product needs to be scaled in a suitable fashion. A scaled dot-product approach divides the result of the dot product by a factor related to the size of the embeddings before passing them through the *softmax*. A typical approach is to divide the dot product by the square root of the dimension of the *query* and *key* vectors, leading to update the scoring

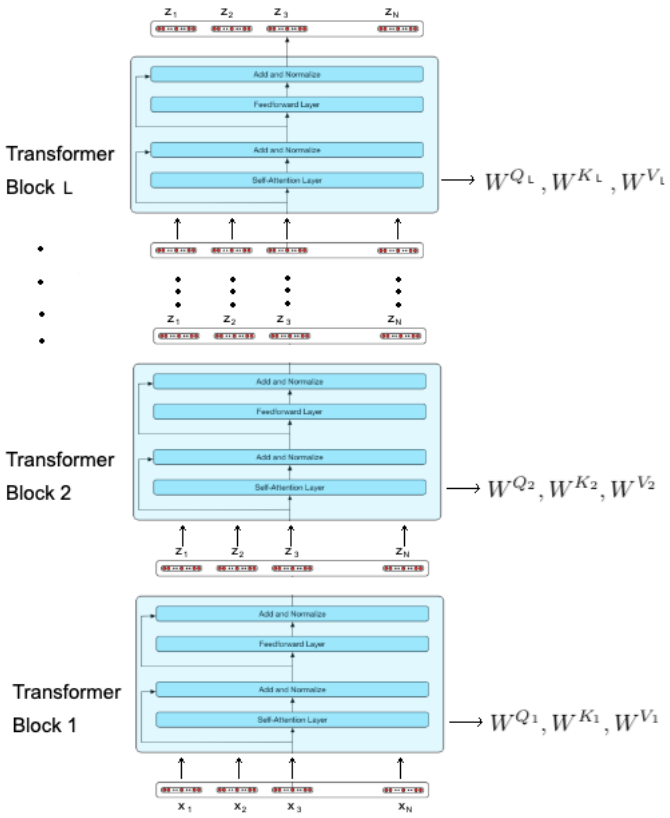


Figure 4.6: Transformer block with multiple self-attention layers [32].

function one more time.

$$scôre(x_i, x_j) = \frac{score(x_i, x_j)}{\sqrt{d_k}} \qquad \forall j = 1, ..., N. \tag{4.10}$$

$$\begin{aligned} \alpha_{ij} &= softmax(score\hat{(}x_i, x_j)) \qquad \forall j = 1, ..., N. \\ &= \frac{\exp^{score\hat{(}x_i, x_j))}}{\sum_{j=1}^{N} \exp^{score(x_i, x_j)}} \qquad \forall j = 1, ..., N. \end{aligned} \tag{4.11}$$

Another practical consideration is in the self-attention process. To make the narrative more straightforward, each vector output $(z_1, ..., z_N)$ was computed separately. However, since each output $(z_i \ \ \forall i = 1, ..., N)$ is calculated independently, this entire process can be done in parallel by taking advantage of efficient matrix multiplication routines. These can be done by packing the input values into a single matrix, $X{=}(x_1, ..., x_N)^T$ with dimension $R^{N \times d_m}$, and multiplying it by the W^Q, W^K and W^V weight matrices (with dimensions

$R^{d_m \times d_q}$; $R^{d_m \times d_k}$ and $R^{d_m \times d_v}$ respectively) to produce new matrices containing all the *key*, *query* and *value* vectors.

$$Q = XW^Q; \quad K = XW^K; \quad V = XW^V. \tag{4.12}$$

With the matrices calculated in equation 4.12 (with dimensions $R^{N \times d_q}$; $R^{N \times d_k}$ and $R^{N \times d_v}$ respectively), all the *query-key* comparisons can be calculated simultaneously by multiplying K and Q in a single matrix multiplication. Taking this one step further, following with multiplications of matrices, also can scale the scores, take the *softmax*, and then multiply the result by V, thus reducing the entire self-attention step for a complete sequence in the following calculation:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{4.13}$$

Figure 4.7 shows visually all the process described in equation 4.13 in a simple matrix multiplications:

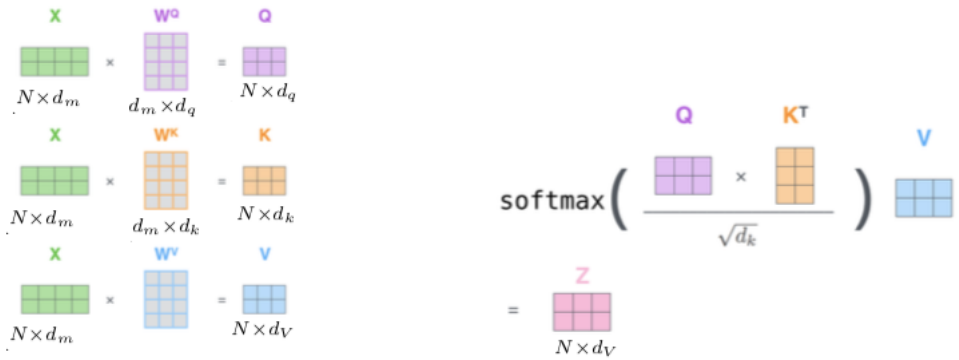


Figure 4.7: Entire process of self-attention with matrices multiplications [71].

As a result, the matrix $Z{=}(z_1, ..., z_N)^T$ with dimension $N \times d_v$ contains all the output vectors. Note that the dimension of each output vector is d_v, in order to these tokens can be processed by the next layer of the encoder as an input, the dimension of d_v should be equal to d_m (the dimension of the first representation of each token x_i).

The last practical consideration is the use of multi-head self-attention. As explained in [32], different words in a sentence (or tokens in a HTTP request) can be related to each other in many different ways simultaneously. For example, distinct syntactic, semantic, and discourse relationships can hold between

verbs and their arguments in a sentence. It would be difficult for a single transformer block to learn to capture all of the different kinds of parallel relations among its inputs. Transformers address this issue with multi-head self-attention layers. These are sets of self-attention layers, called *heads*, that reside in parallel layers at the same depth in a model, each with its own set of parameters. Given these distinct sets of parameters, each *head* can learn different aspects of the relationships that exist among inputs at the same level of abstraction.

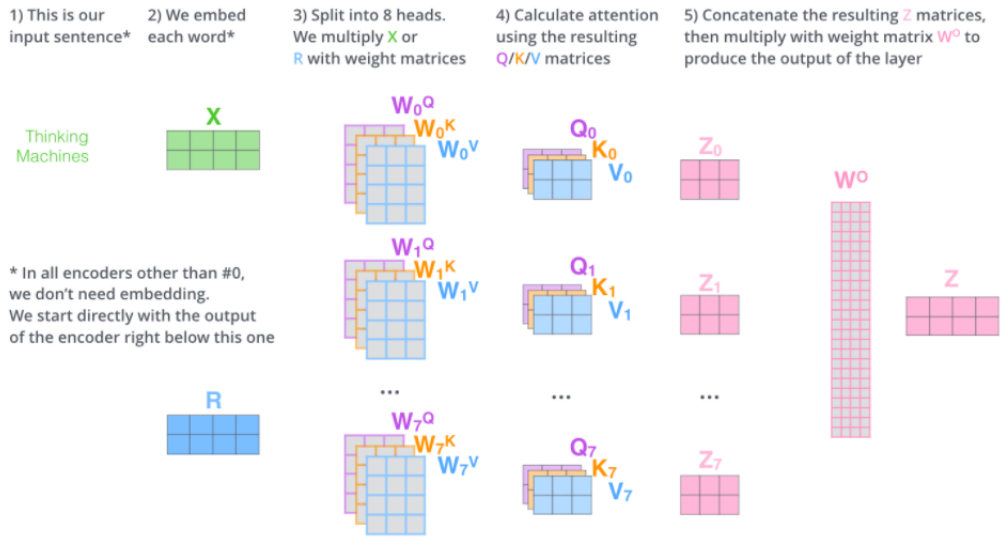


Figure 4.8: Example of the calculus of a self-attention layer with seven *heads* [71].

To implement this notion, the output Z_i of each $head_i$, in a self-attention layer is equipped with its own set of *key*, *query* and *value* matrices: W_i^K, W_i^Q and W_i^V. These are used to project the inputs $(x_1, ..., x_N)$ to the layer separately for each *head*, with the rest of the self-attention computation remaining unchanged, as figure 4.8 shows.

The output of a multi-head layer with h *heads* consists of h matrices $(Z_1, ..., Z_h)$ of the same dimension $R^{N \times d_v}$ that contains all the output vectors for each *head*, as shown in figure 4.9. To make use of these vectors in further processing, them are combined and reduced down to the original input dimension d_m. This is accomplished by concatenating the outputs from each *head* and then using yet another linear projection, W^O, to reduce it to the original output dimension. As a result, the matrix $Z{=}(z_1, ..., z_N)^T$ with dimension $R^{N \times d_m}$ contains all the output vectors. The following figure shows

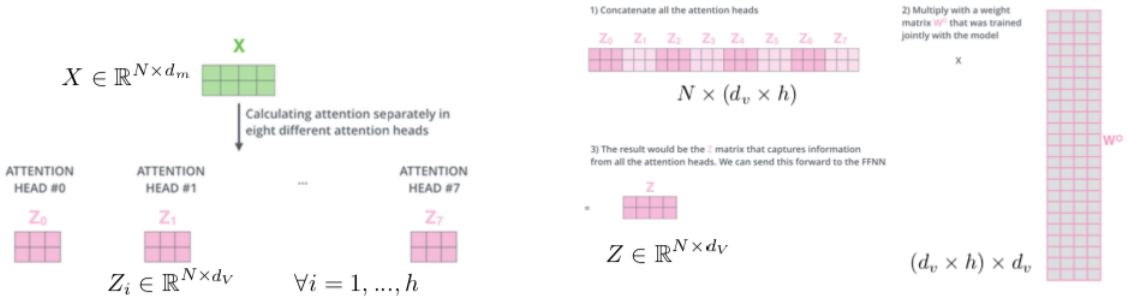


Figure 4.9: Example of the combination of the seven *heads* [71].

the procedure incorporated on the self-attention layer.

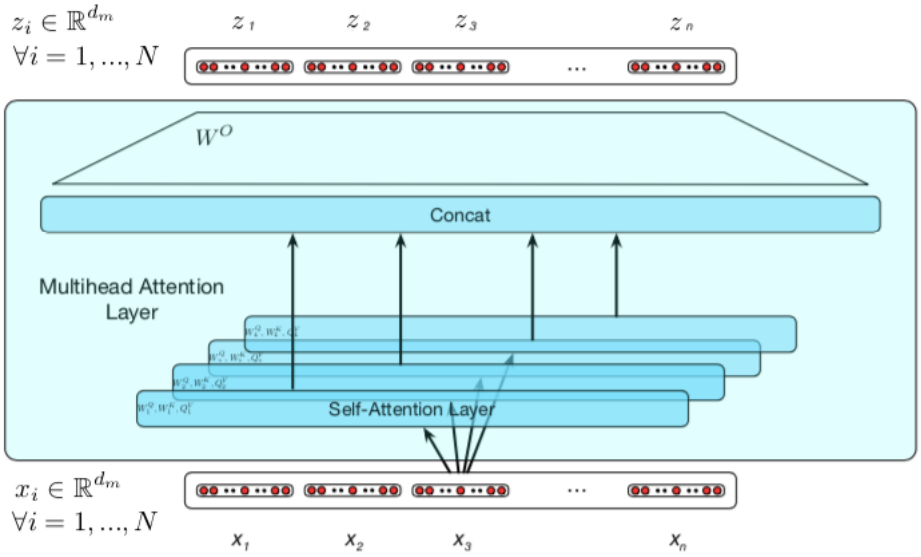


Figure 4.10: Example of the self-attention layer with Multi-head attention [32].

4.2. Fed-forward Layer

The RoBERTa architecture is composed of a stack of $\boldsymbol{L}$ identical transformer encoder layers. Each encoder layer contains two types of sub-layers as was shown on figure 4.6. The first one is a multi-head self-attention layer, which helps to look at other tokens in the sequence while encoding a specific token (as we was seen in the previous section). The second layer, that will be discussed in this section, is a simple position-wise fully connected layer *feed-forward network (FFN)*. This layer is applied to each position $(z_1, ..., z_N)$ separately and consists of two linear transformations $(W_1 \in R^{d_m \times d_{ff}}, b_1 \in R^{d_{ff}})$,

$(W_2 \in R^{d_{ff} \times d_m}, b_2 \in R^{d_m})$ such that:

$$FFN(z_i) = (z_i W_1 + b_1)W_2 + b_2 \qquad \forall i = 1, ..., N. \tag{4.14}$$

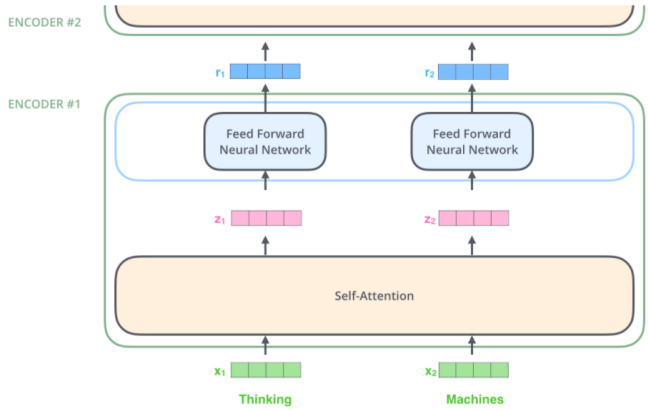


Figure 4.11: *FFN* layer after the self-attention layer [71].

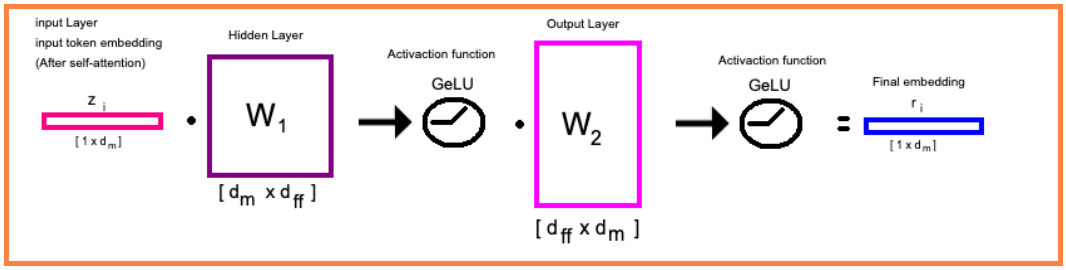


Figure 4.12: *FFN* as a two linear transformations for each z_i in the sequence (the bias vectors, b_1 and b_2, was omitted in the visualization for simplicity).

This neural network receives a vector of dimension R^{d_m} (the token vector after self-attention) and outputs another vector of the same dimension as shown in figure 4.11. Specifically the network has d_{ff} hidden layer neurons and d_m output neurons. The original transformer's convention sets the number of hidden neurons to be four times the input representation size, so there are $d_{ff} = 4 \times d_m$ hidden neurons. Note that while the linear transformations are the same across different positions within the same sub-layer, RoBERTa uses different parameters from block to block, as shown in figure 4.6.

The *FFN* also uses a *Gaussian Error Linear Unit* (GeLU)[26] activation function, as can be seen in figure 4.12. The activation function of a node defines the output of that node given an input or set of inputs. The GeLU activation is defined as:

$$GeLU(z) = 0.5z\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(z + 0.044715z^3\right)\right)\right). \tag{4.15}$$

It was shown in [17] that this activation works better than the standard ReLU [35] in the case of transformer encoders. In addition, an encoder layer employs a residual connection [42] around each of the two sub-layers, followed by a layer normalization [4] such that the output of each sub-layer is:

$$LayerNorm(x_i + Sublayer(z_i)) \qquad \forall i = 1, ..., N. \tag{4.16}$$

Where $Sublayer(z_i)$ represents the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model produce outputs of the same dimension R^{d_m}.

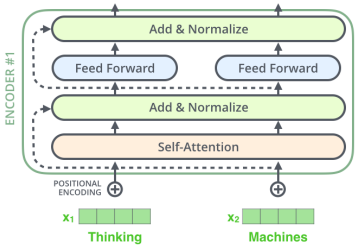


Figure 4.13: Residual connection around each of the two sub-layers [71].

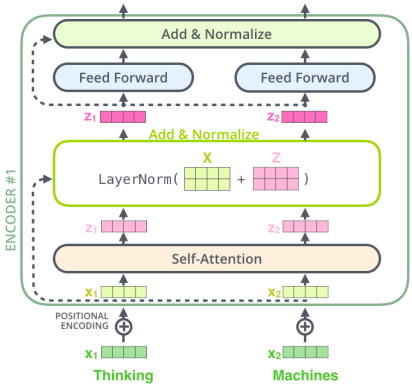


Figure 4.14: Residual connection around the self-attention layer. In the FNN layer is the same operation [71].

4.3. Input Representation

In order to focus on the main pieces of the encoder architecture we made some assumptions concerning the model's input. It was assumed that the raw input text for the model (that can be a sentence or an HTTP request) was a sequence of tokens. Also, it was assumed that each token is first represented as a real-valued vector of dimension R^{d_m}. This section will detail how the *tokenization* process works and then will explain how each token (first represented as a discrete symbol in a dictionary) is converted into a real-valued vector to

be processed by the encoder layers. As a result of these two steps, a raw input text will be represented by a sequence of N numerical vectors $(x_1, ..., x_N)$ of dimension R^{d_m} (where N is the number of tokens). These input representations are constructed by summing two different types of embeddings: token and positional embeddings. A visualization of this construction can be seen in figure 4.15.

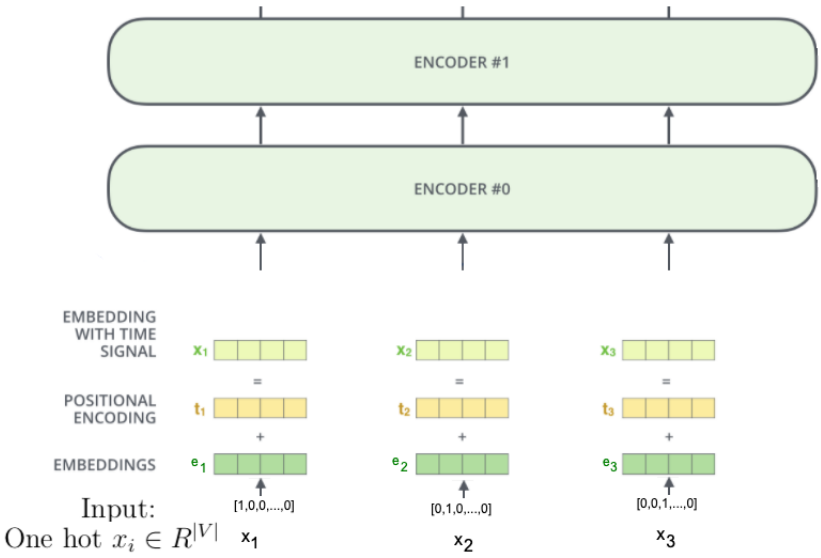


Figure 4.15: Input Representations [71].

4.3.1. Tokenizer

The tokenizer takes the raw text string and breaks it into tokens that the model can ingest. In [43] the authors use a Byte-Pair Encoding (BPE) [67] tokenizer, an hybrid method between character and token-level representations. Instead of defining tokens as words (delimited by spaces, for example) or as characters, this mechanism infers automatically the tokens from a training corpus. This is especially useful when dealing with *unknown* words, a fundamental problem in text processing.

To deal with this *unknown* word problem, the tokenizer automatically induce sets of tokens that include units smaller than words, called *sub-words* (that can be arbitrary *sub-strings*). As [32] says in modern tokenization schemes, most tokens are words, but some tokens are frequently occurring *sub-words*. Thus, some unseen words can be represented by some sequence

of known *sub-words* units or even as a sequence of individual characters if necessary.

The tokenization scheme has two parts: a *token learner* and a *token segmenter*. The *token learner* takes a (large) raw training set of strings and induces a vocabulary (a set of tokens). The *token segmenter* takes a raw test string and segments it into the tokens in the vocabulary. Below is the detailed explanation of the scheme extracted from [32].

The BPE *token learner* begins with a vocabulary that is just the set of all individual characters. It then examines the training corpus, chooses the two symbols that are most frequently adjacent (say 'A', 'B'), adds a new merged symbol 'AB' to the vocabulary, and replaces every adjacent 'A' 'B' in the corpus with the new 'AB'. It continues this process of counting and merging, creating new longer character strings, until k merges have been done (creating k novel tokens); k is thus is a parameter of the algorithm. The resulting vocabulary consists of the original set of characters plus k new symbols.

function BYTE-PAIR ENCODING(strings C, number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters
for $i = 1$ **to** k **do** # merge tokens til k times
  $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
  $t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
  $V \leftarrow V + t_{NEW}$ # update the vocabulary
  Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus
return V

Figure 4.16: BPE encoding algorithm [32].

Once the vocabulary was learned, the *token segmenter* is used to tokenize a test string. The *token segmenter* just runs on the test data, making the merges that have learned from the training data, greedily, in the order that learned them. The procedure is as follows, first segment each test sentence word into characters. Then apply the first rule: replace every instance of *'A'*, *'B'* in the test corpus with *'AB'*, and then the second rule, and so on. By the end, if there are a new (*unknown*) word would be merged into sub-words or even they can be built with individual characters.

4.3.2. Token Embedding

Once the raw input string has been *tokenized*, the next step is to send it into an embedding layer to get the first dense representation of them. Each token (first represented as a discrete symbol in a dictionary) needs to be mapped to a real-valued (dense) vector, this is because neural networks learn through numbers. To do that, the model includes a token embedding layer. This layer can be thought of as a *lookup table* to grab a learned vector representation of each token. This *lookup table*, also known as embedding matrix, consists of a weighted matrix (W^E) that is learned during training. This embedding layer is inherited from the traditional neural networks-based language models detailed in section 3.2.2.

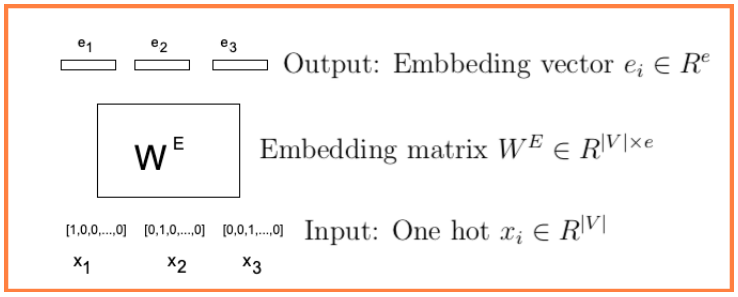


Figure 4.17: Token embedding layer.

Once the string has been tokenized, the vocabulary *id* of each token are used to retrieve the corresponding embedding in the learned token embedding matrix (W^E). That matrix has a row for each token, each is a vector of dimension R^e, and hence the matrix has a dimension of $R^{|V| \times e}$ (where $|V|$ is the size of the vocabulary). For example, to obtain the first dense representation (e_1) for the first token in the sequence (x_1) through the embedding layer the method is straightforward. Simply multiply the one-hot vector corresponding of the token x_1 by the W^E matrix to get the token embedding vector. Note that the dimension R^e of the embedding vector is arbitrary; the transformer uses a dimension $R^e = R^{d_m}$ to be accepted to the next layer of the encoder (that takes R^{d_m} dimensional input representations).

4.3.3. Positional Embedding

As each token in a sentence simultaneously flows through the transformer's encoder stack, the model itself doesn't have any sense of *position/order* for

each token. In order to inject some information about the relative or absolute position of the tokens in the input sequence, RoBERTa uses positional embeddings, as in the original transformer implementation. These embeddings have the same dimension R^{d_m} as the token embeddings so that they can easily be summed up as figure 4.15 shows. They are computed *deterministically* using *sine* and *cosine* functions of different frequencies:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right). \tag{4.17}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right). \tag{4.18}$$

where *pos* is the position of the token in the input sequence and i is the index in the positional embedding of dimension R^{d_m}. Hence, each dimension of the positional embedding corresponds to a sinusoid, and the wavelengths form a geometric progression from 2π to $10000 \times 2\pi$. In [78] the authors choose this function because allow the model to easily learn the relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

In [75] the authors intuitively explain how this combination of *sines* and *cosines* could represent a *position/order*. They propose the following example: suppose you want to represent a number (which means the token position in the sequence) in a binary format that, would be:

0 :	0 0 0 0	8 :	1 0 0 0
1 :	0 0 0 1	9 :	1 0 0 1
2 :	0 0 1 0	10 :	1 0 1 0
3 :	0 0 1 1	11 :	1 0 1 1
4 :	0 1 0 0	12 :	1 1 0 0
5 :	0 1 0 1	13 :	1 1 0 1
6 :	0 1 1 0	14 :	1 1 1 0
7 :	0 1 1 1	15 :	1 1 1 1

Figure 4.18: Positional encoding representation example with binary digits extracted from [75]

Figure 4.18 allows spotting the rate of change between different bits. The least significant bit (LSB) bit is alternating on every number; the second-lowest bit is rotating on every two numbers, and so on. In [75] explains that using binary values would be a waste of space in the world of floats (also, neural networks get along better with floating numbers). So instead using binary

digits, their float continuous counterparts can be used: *sinusoidal functions*. Indeed, they are the equivalent to alternating bits. Moreover, by decreasing their frequencies, we can go from red bits (most significant bit -MSB-) to orange ones (LSB).

4.4. Training Procedure

Recapitulating what was seen throughout the chapter, the first thing that the encoder does is split the raw text into tokens and convert it into a first numeric representation. These vectors are the inputs of the stack of the encoder layers (composed of the respective sub-layers, self-attention, and feed-forward layer). The learning task consists of masking some tokens for the input, recovering the numeric representation of them (based on this context), and predicting which is the original token masked out. In a very simplified example of a sequence $x = (x_1, ..., x_N)$ of N tokens where $\vec{\boldsymbol{\theta}}$ represents all the network parameters of the encoder, the optimization would be the following:

$$\arg\max_{\vec{\boldsymbol{\theta}}} P\left(x_j | x_{1:j-1}, x_{j+1:N}; \vec{\boldsymbol{\theta}}\right). \tag{4.19}$$

For an input that contains one or more mask tokens, the model will generate the most likely substitution for each, for example:

- Input: *I have watched this **[MASK]** and it was awesome.*
- Output: *I have watched this **movie** and it was awesome.*"

Figure 4.19 shows the procedure at a high level. The prediction is given by the final hidden vectors corresponding to the masked tokens. To get a vector of probabilities, it is added a feed-forward layer (over the hidden masked vector) and a *softmax* function (like the one seen in the section 3.2.2). This prediction layer outputs a vector of dimension $R^{|V|}$ (where $|V|$ is the vocabulary size). Each entry of the output vector represents the probability of each word in the vocabulary. For example, if the word *movie* is the entry number i on the vocabulary, the *i-th* element of this output vector represents the probability of the masked word is *movie*.

Masked Language Modeling (MLM) is a fill-in-the-blank task (also referred as a Cloze task [68] in the literature), where a model uses the context words surrounding a mask token to try to predict what the masked word should

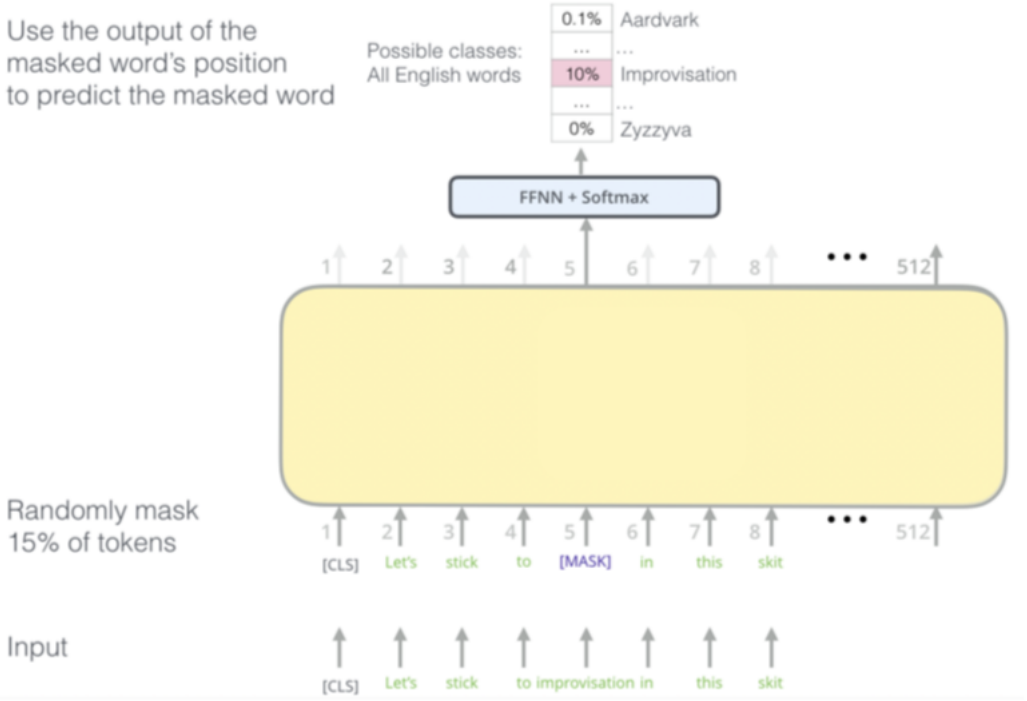


Figure 4.19: Prediction Layer for the optimization process [71]

be. This is a great way to train a language model in a self-supervised setting (without human-annotated labels). The MLM training objective was chosen over the traditional Language models (LM) objective (which aims at predicting the next word given the sequence of previous words, that we present in section 3.2.2), because of the bidirectionality of RoBERTa (that is uses both left and right context in the sequence to predict the target word).

Formally, given an input sequence $x = [x_1, x_2, ..., x_N]$ of N tokens, the MLM first selects a random set of k positions (integers between 1 and N) to mask out $m = [m_1, ..., m_k]$. The tokens in the selected positions are then replaced with a **[MASK]** token, resulting in the masked input sequence:

$$x^{masked} = [x_1, x_2, \textbf{[MASK]}, x_4, ..., \textbf{[MASK]}, x_N]. \tag{4.20}$$

RoBERTa learns to predict the original identities of the k masked-out tokens by computing an output word distribution over the vocabulary $|V|$:

$$\hat{y}(h) \ \ \forall \ \ h = 1, ..., k. \tag{4.21}$$

More precisely, given the h-th masked word x_{m_h} from sequence x, the MLM loss function (used to optimize network parameters $\vec{\boldsymbol{\theta}}$) is the cross-entropy between the predicted probability distribution $\hat{y}(h)$, and the ground true, $y(h)$, which is simply the one-hot vector for x_{m_h}. Therefore, we have:

$$\begin{aligned} L_{MLM}^{(h)}(\vec{\boldsymbol{\theta}}) &= CrossEntropy(y^{(h)}, \hat{y^{(h)}}) \\ &= -\langle y^{(h)}, \log \hat{y}^{(h)} \rangle. \end{aligned} \tag{4.22}$$

The overall loss of the sequence is simply the average loss for all the k masked-out tokens in x^{masked}:

$$\begin{aligned} L_{MLM}(\vec{\boldsymbol{\theta}}) &= \frac{1}{k} \sum_{h=1}^{k} L_{MLM}^{(h)}(\vec{\boldsymbol{\theta}}) \\ &\arg\min_{\vec{\boldsymbol{\theta}}} L_{MLM}(\vec{\boldsymbol{\theta}}). \end{aligned} \tag{4.23}$$

The masking scheme works as follows: RoBERTa selects 15% of all tokens in each training sequence at random. If the *i-th* token is chosen, it is replaced with:

- the [**MASK**] token 80% of the time
- a random token 10% of the time;
- the unchanged *i-th* token 10% of the time.

The selected tokens are not always replaced with [**MASK**]. In [17] the authors explained that while the [**MASK**] token tells nothing about the meaning of the word it replaced, RoBERTa has to infer what word was there purely by looking at the context. This might give RoBERTa a tendency to ignore the input embedding and infer everything from its context, which we would not want in the inference phase when we need the feature vector of a certain token.

4.5. Downstream Tasks

The use of pre-trained language models has yielded significant improvements on a diverse set of NLP tasks because they are capable of doing *transfer*

learning. The concept of *transfer learning* has been instrumental in the success of deep learning in computer vision. As explained in [8], creating a good deep learning network for computer vision tasks can take millions of parameters and be very expensive to train. Researchers discovered that deep networks learn hierarchical feature representations (simple features like edges at the lowest layers with gradually more complex features at higher layers). Therefore, rather than training a new network from scratch each time, the lower layers of a trained network with generalized image features could be copied and transfered for use in another network with a different task. It soon became common practice to download a pre-trained deep network and quickly retrain it for the new task or add additional layers on top (vastly preferable to the expensive process of training a network from scratch).

In NLP, the weights of the pre-trained language networks already encode a lot of information about the language. Despite being trained with only a language modeling goal, they learn highly transferable and task-agnostic properties of the language [20]. This information can be used as input of another downstream task with the *transfer learning* approach. For many, the introduction of deep pre-trained language networks signals the same shift in NLP to *transfer learning* that the one experimented by the field of computer vision [8].

There are two strategies for applying pre-trained RoBERTa language representations to downstream NLP tasks: *fine-tuning* and *feature-based*. On one hand, the *fine-tuning* approach introduces minimal task-specific parameters to the pre-trained model, and re-trained it on the downstream tasks by simply *fine-tuning* all parameters. On the other hand, the *feature-based* approach uses task-specific architectures that include the pre-trained representations as input features for learning the downstream task.

4.5.1. Fine-tuning

This approach consist on adding a prediction layer (like a *softmax*) after the pre-trianed network, then use downstream task data to learn the new layer's weights and slightly change the weights of the whole pre-trained model.

Figure 4.20 illustrates a *fine-tuning* process (for a *sentence classification* task) composed of two steps. The left side of the figure shows an example of the first step. The model is trained with a language modeling goal with data

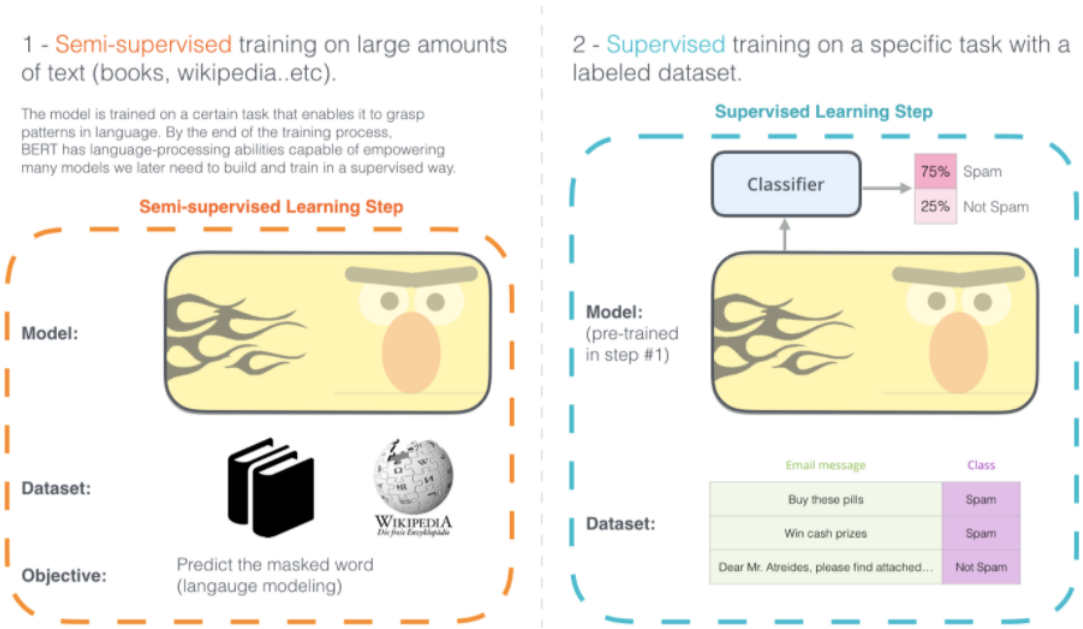


Figure 4.20: Fine tuning approach composted with two steps [71].

scraped from *Wikipedia*. This task enables the model to grasp patterns in the language. The right side shows the *fine-tuning* step, it consist in a *sentence classification* task with another dataset with e-mail messages labeled with two classes, *spam* and not *spam*. The strategy consists of adding an additional layer (to the pre-trained model) and re-train all the parameters of the architecture to predict if an e-mail is *spam* or not. The same approach can be used in the security domain. The pre-trained model can be learned with the flow of HTTP requests of an application. Then, in the next step *fine-tuning* the network with a tagged data set, composed of requests of the application (with *attacks* and *normal* instances).

As already explained, the *fine-tuning* strategy (for a *sentence classification* task) consists of adding a prediction layer to the top of the pre-trained model and re-train the whole architecture optimizing for the downstream task. Until now, we mentioned this mechanism at a high level, but it is important to explain specifically in which part of the pre-trained architecture is adding, as figure 4.21 shows.

Note that a **[CLS]** token (*classification token*) is seen at the beginning of the sequence in figure 4.21, which has been omitted for simplicity until now. This token is used for the prediction task (only in the *fine-tuning* approach for *sentence classification*). The final hidden state of the **[CLS]** token is taken

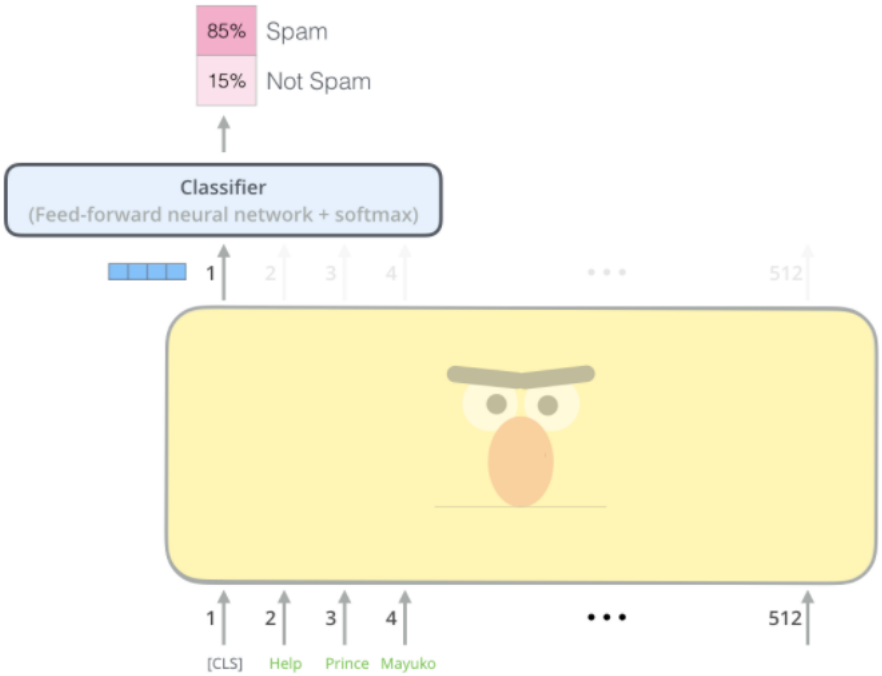


Figure 4.21: Fine tuning approach for text classification [72]

as the fixed-dimensional pooled representation of the input sequence (as the aggregate representation). It is processed by the encoder like any other token, but it is intended to be used only in the *fine-tuning* approach. The hidden state of the **[CLS]** is connected to the classification layer to make the predictions. Therefore, the classification layer is the only new parameter added and has a dimension of $H \times K$, where K is the number of labels to classify (in the example of the figure 4.21 are $K = 2$, *spam* and *not spam*) and H is the size of the hidden state. The label probabilities are computed with a standard *feed forward neural network* and a *softmax* activation function.

Figure 4.20 shows an example of a *fine-tuning* approach for a *sentence classification* task in pre-defined categories (*spam* and *not spam*). However, it is also possible to use the same *fine-tuning* approach but to predict the category of each token (*token classification*). The *token classification task* is similar to *sentence classification*, except each token within the text receives a prediction. A common use of this task is called *Named Entity Recognition* (NER), a subtask of *information extraction* that seeks to locate and classify named entities mentioned in unstructured text into pre-defined categories such as *person names*, *organizations*, *locations*, *medical codes*, *time expressions*, *quantities*,

monetary values, *percentages*, among others. This task requires the data be labeled at the token level. The training consists of adding a prediction layer to the top of each token representation of the pre-trained model and re-train the whole architecture optimizing for the downstream task (predicting each token's category).

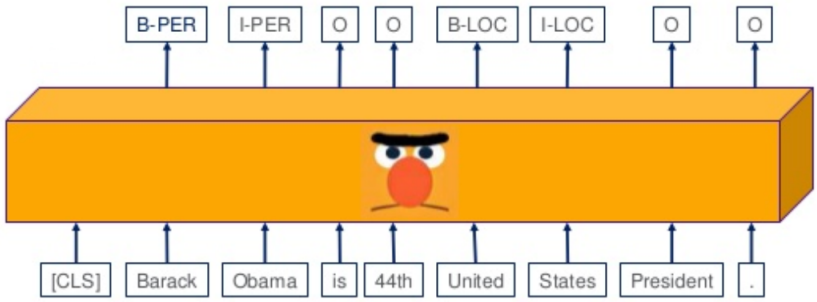


Figure 4.22: Fine tuning approach for *token classification* [64]

Figure 4.22 shows an example for a *token classification* task. For example, the entity *Barack Obama* is classified as a person name (PER) and *United States* is classified as a location (LOC). In order to tagging the entities is used the BIO tagging scheme [60] (short for *beginning*, *inside* and *outside*). Here the prefix *B-* before a tag (for example B-PER) denotes the *beginning* of an entity, and the prefix *I-* before a tag stands for *inside* and is used for all words comprising the entity except the first one, and *O* means the absence of an entity.

4.5.2. Feature-based

In addition to the *fine-tuning* approach, where a simple output layer is added to the pre-trained model and all parameters are jointly re-trained on a downstream task, RoBERTa can also be used with a *feature-based* approach. In this case the token representations are extracted from the pre-trained model and serve as inputs to other task-specific architectures.

As the right side of the figure 4.23 illustrates, the pre-trained model takes a request r_i and tokenizes it to obtain a representation $r_i = \{token_{i1}, ..., token_{in_i}\}$, where n_i is the number of tokens in the request r_i (the model has a *max length* of tokens as inputs to be processed). Then, generates

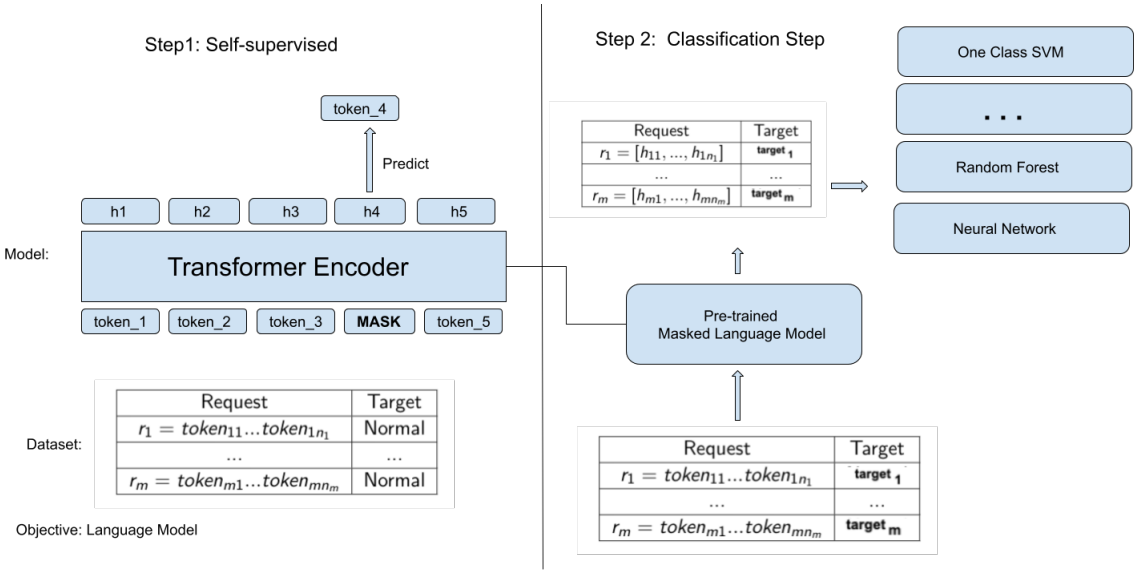


Figure 4.23: Feature-based approach.

as output a deep contextual representation of each token. This deep representation is obtained using the weights of the encoder's layers. In this way, each request r_i is transformed into a numeric vector $\{h_{i1}, ..., h_{in_i}\}$. Each $h_{ij} \in R^H$ is the vector representation for the $token_{ij}$ in r_i (where $\boldsymbol{H} = 768$ is the size of the encoder hidden layer). These representations can be used as input of other classificator (such as a One-class SVM, Random Forest, etc.).

A benefit of this approach is to pre-compute an expensive representation of the training data once and then run many experiments with cheaper models with these representations as inputs. Furthermore, there are no restrictions about the nature of the second classifier. If the new dataset contains only normal data, the second classifier may be a one-class classifier (like a SVM one class, for example). But if the new dataset has multiples classes, the second classifier may be a Random Forest or a Neural Network, or any other classfier.

There is a consideration about the deep representation obtained with the weights of the encoder layers. As mentioned before, the encoder is composed of a stack of $\boldsymbol{L}$ identical layers. The output of each encoder layer along each token's path can be used as a feature representation of that token, as figure 4.24 exhibits. Which one of these vector works best as a contextualized embedding is an open question and may depend on the downstream task as in [72] explain. For example, [17] lead a study for a NER task using the *feature-based* approach. In the study lead by [17] the authors applied a *feature-based*

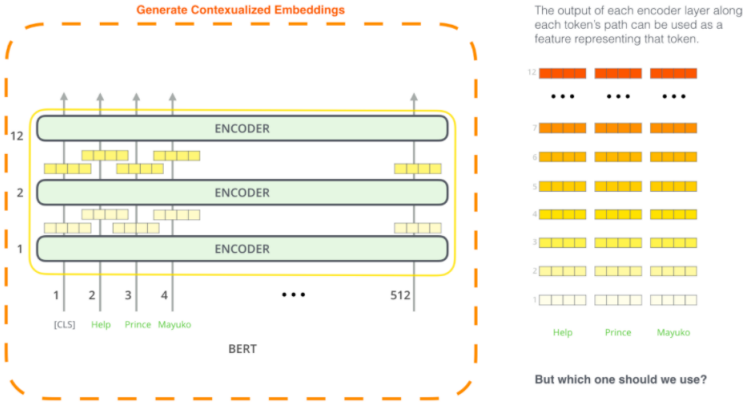


Figure 4.24: Feature-based approach [72].

approach by extracting the activations from one or more layers without *fine-tuning* any parameters of a pre-trained Transformer, then used these contextual embeddings as inputs to a randomly initialized two-layer 768-dimensional BiLSTM before the classification layer (that predict each tokens category).

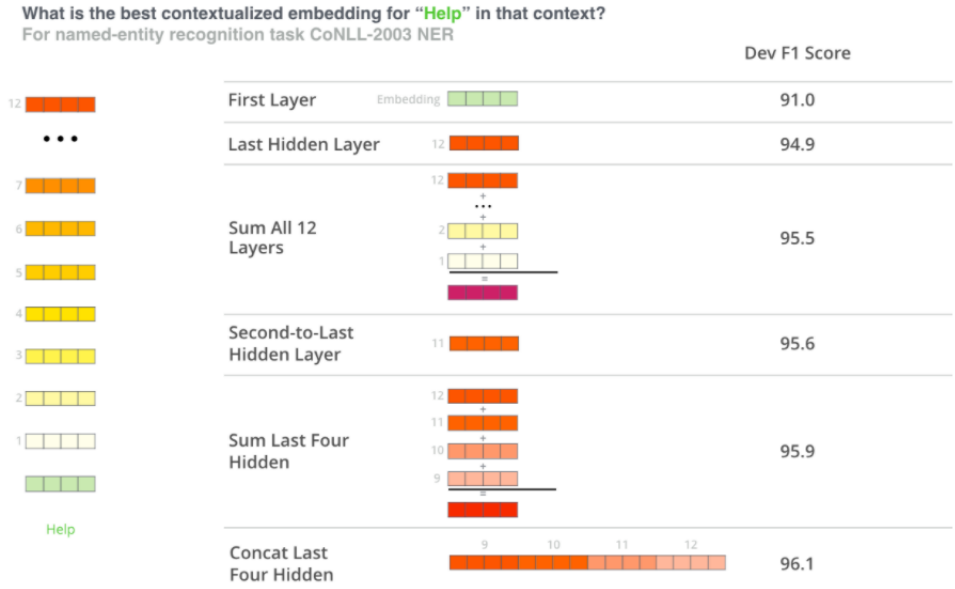


Figure 4.25: Results on named entity recognition (NER) using BERT embeddings with a *feature-based* approach [72].

The results presented by [17] showed that the best performing method for this NER task is concatenating the last four hidden layers (for the pre-trained Transformer) as the contextual word embeddings, as figure 4.25 shows. The authors showed that the results of concatenating the last four hidden layers are

only 0.3 F_1 score behind *fine-tuning* the entire model. Therefore, the authors demonstrated empirically that the pre-trained Transformer is effective for both *fine-tuning* and the *feature-based* approach.

Chapter 5

A two step learning architecture

The extraction of features from raw data is one of the most challenging problems when applying anomaly detection models to web security [58].

Inspired by the success of pre-trained contextual embeddings for natural languages, also recently successfully applied to other modalities such as programming languages, in this thesis we propose to use a transformer-based language representation of HTTP requests to address the problem of web applications attack detection. Our approach consists in a two-step learning framework: first, we treat the HTTP requests as raw text and pre-train a RoBERTa [43] language model as a *feature extractor*; then, we train a one-class learning model with these features.

This section presents the main components of the proposed learning architecture. We shall go through each element of the framework and explain the design decisions. In 5.1 we shall explain the input of the model: the HTTP requests. In 5.2 we delve into the RoBERTa language model, explaining some decisions concerned with the architecture and the training setup. Then in 5.3 we review the OCC-SVM and present an established performance metric that we use to select automatically an operational point for the OCC-SVM model (without the need for data labeled as attacks). Finally, in 5.4 we summarize the whole architecture with all the points depicted before.

5.1. Input HTTP data

The input to our system are the raw HTTP requests as any web application server receives them. The first component is the parser. It is responsible for

cleaning and parsing the requests (based on the HTTP protocol structure) before sending them to the tokenizer (described in 4.3.1). In this specific problem, the samples are instances of the semi-structured text protocol HTTP request. This protocol is used to exchange information between the client and the server. To take advantage of the protocol structure, the parser has three main objectives: *information decoding*, *headers filtering* and *special characters preservation*. For the experiments of this thesis, we have used the parser implemented by [45] in the context of the WAF Mind project. As follows, we summarize the explanation of the three main objectives of the parser (the full explanation can be found in [45]).

5.1.1. Information decoding

As the HTTP protocol is text-based when non-ASCII information has to be exchanged some encoding has to be employed (for example URLs can not contain spaces). The main encoding mechanism used in the Web is defined in the RFC3986 [7], namely a method of encoding for non-ASCII characters transformation in the URL known as the URL encoding format. URL encoding converts non-ASCII characters into a format that can be transmitted over the Internet.

The URL encoding mechanism substitutes specials characters (non-ASCII) by three characters: the special character % and two hexadecimal numbers corresponding to the US-ASCII value of the character. The URL encoding format could be used in the URI for the query string and the parameters sent in the body.

The first action of the parser is to decode all the URL encoded information so the learning algorithm can work with the real information. For example, the following encoded URI:

```
/user/login?destination=search%2Fnode%2Flimite%20de%20cursadas
```

will be decoded as:

```
/user/login?destination=search/node/limite de cursadas.
```

5.1.2. Headers filtering

HTTP headers let the client and the server pass additional information with an HTTP request or response. For example, it contains more information

about the resource to be fetched or about the client requesting the resource [29]. The parser filters the information contained in headers specific to the training requests. As it is explained in [45], the objective of this task is to eliminate information in the requests that are specific to the training environment that could lead to the machine learning algorithm to learn these particularities to classify new requests. Examples of these cases are the value of a cookie or the timestamp of the last time the web page was modified. The full headers filtered during the pre-processing are presented in figure 5.1.

host	x-forwarded-for	via	client-ip	referer
if-none-match	set-cookie	last-modified	if-modified-since	—
proxy-authorization	if-range	if-match	from	upgrade
authorization	max-forwards	ua-cpu	ua-disp	ua-os
cookie	etag	accept-charset	accept-encoding	if-unmodified-since
ua-color	ua-pixels	x-serial-number	pragma	range
cache-control	accept-language	trailer	expect	

Figure 5.1: Headers filtered during the parsing process [45].

5.1.3. Special characters preservation

Several attacks make use of specially crafted input to make the server to execute a not expected functionality. Most of these crafted inputs use *special characters* as part of the payload (for instance, `. , ;< >=/`). But also, some of those *special characters* can be part of the HTTP protocol. Therefore, it is important to distinguish whether it is part of the protocol or the parameter value. For example, the *destination* parameter and the value parameter (exposed earlier) is separated by the *special character* `=`, but also has two `/` *special characters* on the value parameter:

```
destination=search/node/limite de cursadas.
```

The parser uses the knowledge of the HTTP protocol to pre-process the request to preserve the *special characters* on the parameter value and deletes the *special characters* if they are part of the protocol. For that, the parser analyzes the parameters from the query string, the body, and headers and split them in name and parameter value as a different text string using the space character as a separator. So, for example, the destination *parameter* that we saw earlier would be transformed as:

```
destination search/node/limite de cursadas.
```

In this way, the three *special characters* / in the parameter value would be keeps in the text. But the *special character* = is deleted when it is part of the protocol.

5.1.4. Parsing result

Our model inputs are the requests after being processed by the parser implemented by [45]. Figure 5.2 shows an example of a request before being processed by the parser. Then, figure 5.3 shows the request after the transformation. We can see that the original 16 lines of the request get transformed into one string of words separated by the space character.

```
POST /user/login?destination=search%2Fnode%2Flimite%20de%20cursadas
    HTTP/1.1
Host: www.fing.edu.uy
Connection: keep-alive
Content-Length: 200
Cache-Control: max-age=0
Origin: https://www.fing.edu.uy
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36
    (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36
Content-Type: application/x-www-form-urlencoded
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
    webp,*/*;q=0.8
Referer: https://www.fing.edu.uy/user/login?destination=search%2
    Fnode%2Flimite+de+cursadas
Accept-Encoding: gzip, deflate, br
Accept-Language: es-ES,es;q=0.8,en;q=0.6
Cookie: SESS77d4c056e4744b8992994833 51de0e63=2
    pp35g1jp42mjm9216g04m1bm2; _pk_ref.1.af1c=%5B%22%22%2C%22%22%2
    C1489595176%2C%22https%3A%2F%2Fwww.google.com.uy%2F%22%5D;
    has_js=1; _pk_id.1.af1c=9bd9ba4509731 89b
    .1487191441.5.1489595186.1489595176.; _pk_ses.1.af1c=*

name=jose.perez&pass=**********&form_build_id=form-
    Kh0_uekNLVtroZXTLGt-71ZmRx3TJEwP7jz3K2DWQI0&form_id=user_login&
    securelogin_original_baseurl=https%3A%2F%2Fwww.fing.edu.uy&op=
    Iniciar+sesi%C3%B3n
```

Figure 5.2: HTTP Request before parsing [45]

```
POST HTTP/1.1  destination search/node/limite de cursadas connection
    : keep-alive content-length: 200 origin: https://www.fing.edu.uy
     upgrade-insecure-requests: 1 user-agent: mozilla/5.0 (windows
    nt 10.0; wow64) applewebkit/537.36 (khtml, like gecko) chrome
    /56.0.2924.87 safari/537.36 content-type: application/x-www-form
    -urlencoded accept: text/html,application/xhtml+xml,application/
    xml;q=0.9,image/webp,*/*;q=0.8  name jose.perez pass **********
    form_build_id form-Kh0_uekNLVtroZXTLGt-71ZmRx3TJEwP7jz3K2DWQI0
    form_id user_login securelogin_original_baseurl https://www.fing
    .edu.uy op Iniciar sesi\'on
```

Figure 5.3: Parsed HTTP Request, the input for the model [45]

5.2. HTTP language model with RoBERTa

The language model for the HTTP requests was trained in a self-supervised fashion. We use a Robustly Optimization Bidirectional Encoder Representations from Transformers architecture (RoBERTa) [43] explained in section 4. Using this model each HTTP request is transformed into a real-valued vector that captures the contextual information of each token present in the request.

5.2.1. Tokenizer

Our model inputs are the HTTP requests after being processed by the parser described in section 5.1. In order to perform the tokenization of the HTTP request we use a Byte-Pair Encoding (BPE)[67] tokenizer, a hybrid between character and token-level representations. It relies on sub-word units, which are extracted by performing statistical analysis of a training corpus. We use the same *tokenizer* learned by [59], a clever implementation of BPE that uses *bytes* instead of Unicode characters as the basic *sub-words* units. In that work, the authors learned a *sub-words* vocabulary of $\approx 50K$ units that can still encode any input string without introducing any *unknown* tokens. We could have chosen another pre-trained BPE tokenizer instead of the one trained by [59]. The key point was to use a BPE tokenizer trained on a vast corpus ($\approx$ 40 GB of text data scraped from the web). Thus it can tokenize any word (and any character) of any language without using the *unknown* token. This tokenizer allowed us to tokenize the data for the applications we want to protect without recognizing any token as *unknown*, since if a token is not in the vocabulary, it is built with individual characters.

5.2.2. The proposed architecture

The language model that we train is composed of a stack of $\boldsymbol{L}$ identical transformer encoder layers. As explained in section 4, each encoder layer contains two types of sub-layers. The first one is a multi-head self-attention mechanism, which helps looking at other tokens in the sequence while encoding a specific token. The second sub-layer is simply a feed-forward network (FFN), which is applied to each position (token representation from previous layer) separately and identically.

We denote the number transformer encoder blocks as $\boldsymbol{L}$, the hidden size

as $\boldsymbol{H}$, and the number of self attention heads as $\boldsymbol{A}$. Our model architecture (depicted in figure 5.4) uses the following set of parameters ($\boldsymbol{L} = 12$, $\boldsymbol{H} = 768$, $\boldsymbol{A} = 12$, Total Parameters = 125M). We chose this set of hyper-parameters because we use the same configuration used in the original implementation of RoBERTa reported in [43].

```
RobertaModel(
  (decoder): RobertaEncoder(
    (sentence_encoder): TransformerSentenceEncoder(
      (embed_tokens): Embedding(50265, 768, padding_idx=1)
      (embed_positions): LearnedPositionalEmbedding(2050, 768, padding_idx=1)
      (layers): ModuleList(
        (0): TransformerSentenceEncoderLayer(
          (self_attn): MultiheadAttention(
            (k_proj): Linear(in_features=768, out_features=768, bias=True)
            (v_proj): Linear(in_features=768, out_features=768, bias=True)
            (q_proj): Linear(in_features=768, out_features=768, bias=True)
            (out_proj): Linear(in_features=768, out_features=768, bias=True)
          )
          (self_attn_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
          (fc1): Linear(in_features=768, out_features=3072, bias=True)
          (fc2): Linear(in_features=3072, out_features=768, bias=True)
          (final_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        )
        (1): TransformerSentenceEncoderLayer(
          (self_attn): MultiheadAttention(
            (k_proj): Linear(in_features=768, out_features=768, bias=True)
            (v_proj): Linear(in_features=768, out_features=768, bias=True)
            (q_proj): Linear(in_features=768, out_features=768, bias=True)
            (out_proj): Linear(in_features=768, out_features=768, bias=True)
          )
          (self_attn_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
          (fc1): Linear(in_features=768, out_features=3072, bias=True)
          (fc2): Linear(in_features=3072, out_features=768, bias=True)
          (final_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        )
        ⋮
        (11): TransformerSentenceEncoderLayer(
          (self_attn): MultiheadAttention(
            (k_proj): Linear(in_features=768, out_features=768, bias=True)
            (v_proj): Linear(in_features=768, out_features=768, bias=True)
            (q_proj): Linear(in_features=768, out_features=768, bias=True)
            (out_proj): Linear(in_features=768, out_features=768, bias=True)
          )
          (self_attn_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True
          (fc1): Linear(in_features=768, out_features=3072, bias=True)
          (fc2): Linear(in_features=3072, out_features=768, bias=True)
          (final_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        )
      )
      (emb_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    )
    (lm_head): RobertaLMHead(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    )
  )
  (classification_heads): ModuleDict()
)
```

Figure 5.4: RoBERTa architecture for HTTP modeling.

5.2.3. Training strategy

In order to learn the deep contextualized representation of tokens the training strategy is the self-supervised learning approach, described in section 4.4. As already explained, the model *masks* some of the tokens from the input, and then the goal is to predict the original *masked* token based only on its context.

In an attempt to predict the *masked* tokens, the model should be able to extract some information from the language, not only structural information but some semantic information as well. This information is encoded in the weights of the encoding layers [20].

```
model roberta_base, criterion MaskedLmLoss
num. model params: 125876313 (num. trained: 125876313)
training on 1 GPUs
max tokens per GPU = 2048 and max sentences per GPU = 16
```

Figure 5.5: Training strategy.

5.2.4. Implementation Details

We use the library FAIRSEQ [50] to train the RoBERTa networks. It is an open-source sequence modeling toolkit. It is based on PyTorch [55] and supports distributed training across multiple GPUs and machines.

```
%%bash

TOTAL_UPDATES=125000    # Total number of training steps
WARMUP_UPDATES=10000    # Warmup the learning rate over this many updates
PEAK_LR=0.0005          # Peak learning rate, adjust as needed
TOKENS_PER_SAMPLE=2048   # Max sequence length
MAX_POSITIONS=2048       # Num. positional embeddings (usually same as above)
MAX_SENTENCES=16        # Number of sequences per batch (batch size)
UPDATE_FREQ=16          # Increase the batch size 16x
DATA_DIR=./modSecurity-bin

fairseq-train --fp16 $DATA_DIR \
    --task masked_lm --criterion masked_lm \
    --arch roberta_base --sample-break-mode complete --tokens-per-sample $TOKENS_PER_SAMPLE \
    --optimizer adam --adam-betas '(0.9,0.98)' --adam-eps 1e-6 --clip-norm 0.0 \
    --lr-scheduler polynomial_decay --lr $PEAK_LR --warmup-updates $WARMUP_UPDATES --total-num-update $TOTAL_UPDATES \
    --dropout 0.1 --attention-dropout 0.1 --weight-decay 0.01 \
    --max-sentences $MAX_SENTENCES --update-freq $UPDATE_FREQ \
    --max-update $TOTAL_UPDATES --log-format simple --log-interval 1 \
    --skip-invalid-size-inputs-valid-test --max-tokens 2048 \
    --save-dir ./checkpoints_sc4_modSecurity \
    --no-epoch-checkpoints
```

Figure 5.6: FAIRSEQ training setup for RoBERTa language model.

Figure 5.6 shows some implementation details with some decisions that we set up to train the model in our experiments. The total number of training updates is 125.000, and the batch size is 16. The model has a *max length* of 2048 tokens as inputs to be processed. Therefore, if an HTTP request (after the *tokenization* process) exceeds the 2048 tokens, there is a truncation and only processes the first 2048 tokens. The network is optimized with Adam [34] using the following parameters: $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 1e-6$ and L_2 weight decay of 0.01. We chose this optimizer, and these parameters, to follow the original implementation of RoBERTa reported in [43]. The Adam [34] optimizer (the name is derived from *adaptive moment estimation*) is a variant

of the classical stochastic gradient descent (SCD), explained in section 3.2.2. As already mentioned, SGD is a procedure to update all the network weights iterative based on training data. Adam is a method for efficient stochastic optimization.

5.3. One-Class Classification

The pre-trained RoBERTa model takes a request r_i and tokenizes it to obtain a representation $r_i = \{token_{i1}, ..., token_{in_i}\}$, where n_i is the number of tokens in the request r_i (as already mentioned, the model has a *max length* of 2048 tokens as inputs to be processed). Then, generates as output a deep contextual representation of each token. This deep representation is obtained using the weights of the encoder's last layer. In this way, each request r_i is transformed into a set of numeric vectors $r_i = \{h_{i1}, ..., h_{in_i}\}$. Each $h_{ij} \in R^H$ is the vector representation for the $token_{ij}$ in r_i (where H = 768 is the size of the encoder hidden layer).

In order to get a representation of the full request, the most common technique is to average token representations to produce a vector $\bar{r}_i \in R^H$ such that: $\bar{r}_i = \frac{1}{n_i} \sum_{j=1}^{n_i} h_{ij}$. With these procedure we transform a set of normal HTTP requests $D = \{r_1, .., r_m\}$ into a numeric form $\bar{D} = \{\bar{r}_1, .., \bar{r}_m\}$, each $\bar{r}_i \in R^H$. We perform a One-Class Classification model (OCC), with a One-Class Support Vector Machine (OCSVM), with these representations.

Once we have the feature vector mentioned above, we apply the well-know OCSVM classifier introduced in [65] with a Radial Basis Function (RBF) Kernel. The authors present an algorithm that returns a function f that takes the value $+1$ in a *small* region capturing most of the training data points and -1 elsewhere. The strategy is to map the data into the feature space corresponding to the kernel and to separate them from the origin with maximum margin. For a new point x, the value $f(x)$ is determined by evaluating which side of the hyper-plane it falls on in the feature space. For each sample, we can calculate the signed distance to the separating hyper-plane. The distance is positive for an inlier (considered as normal) and negative for an outlier (a possible attack). We can set a threshold (θ) and classify a sample as normal if the distance to the hyper-plane is greater than θ. Varying θ between -1 and $+1$ we obtain a ROC with different operational points.

5.3.1. Estimation of the optimal operational point.

We must set up two parameters to optimize the performance of the OCSVM: ν and γ. The γ parameter is required by the RBF kernel to define a frontier. The ν parameter corresponds to the probability of finding a new, but normal, observation outside the frontier. To find the optimal parameters we use a traditional grid-search method. In the case of supervised classification, we can use performance metrics such as F-score or the overall accuracy to evaluate each configuration of parameters. However, these metrics rely on positive and negative samples so it is not possible to use them in an anomaly detection scenario. Nevertheless, [41] introduces a performance measure, $\hat{F}$, that can be estimated from normal and unlabeled examples. They show that $\hat{F}$ is proportional to the square of the geometric mean of precision and recall. Thus, they argue that has roughly the same behaviour as the F_1-score (the harmonic mean of the precision and recall). Therefore, we use a grid-search with the $\hat{F}$ metric for selecting the best parameters (ν and γ) and the threshold (θ) of the OCSVM classifier, using a validation set with only normal and unlabeled examples.

5.3.2. Explanation of the performance metric

Now, we proceed to explain why the $\hat{F}$ presented in [41] is proportional to the square of the geometric mean of the *precision* and *recall*. First, we will refresh some (classic) definitions that will be used throughout the explanation.

- The F_1-score (the harmonic mean of the *precision* and *recall*) is defined as:

$$F_1 - score = \frac{2 * Precision * Recall}{(Precision + Recall)}. \tag{5.1}$$

- The geometric mean is defined as:

$$geometric\ mean = \left(\prod_{i=1}^{n} x_i \right)^{\frac{1}{n}}. \tag{5.2}$$

- The definition of the conditional probability is:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}. \tag{5.3}$$

- The Bayes's rule states:

$$P(B|A) = \frac{P(A \cap B)}{P(A)}. \tag{5.4}$$

As was already described, the One-class-SVM returns a parametric function f that takes the value 1 in a *small* region capturing most of the training data points and -1 elsewhere. Now let's redefine the function f with the parameters (ν, γ, and θ) that we must set up. For a new data point x, the classification $f_{\nu,\gamma,\theta}(x)$ is determined by evaluating which side of the hyper-plane it falls on. When θ is the signed distance for the hyperplane and ν and γ the parameters described before, the parametric function is defined as:

$$f_{\nu,\gamma,\theta}(x) = \begin{cases} \geq \theta, & \text{predict as normal (output a 1)} \\ < \theta, & \text{predict as abnormal (output a -1).} \end{cases} \tag{5.5}$$

In order to define *precision* and *recall* in a probabilistic way, we assume a complete dataset $d = (X, Y)$ when $X = (x_1, ..., x_n)\ \forall\ x_i \in R^d$ is the feature space and $Y = (y_1, ..., y_n)\ \forall\ y_i \in [1, -1]$ the ground truth label for each sample. Using the definition 5.5 of the parametric function and the definition of the conditional probability 5.3 we can define *precision* and *recall* in a probabilistic way[1]:

- Precision

$$P(Y = 1|f_{\nu,\gamma,\theta}(x) = 1). \tag{5.6}$$

- Recall

$$P(f_{\nu,\gamma,\theta}(x) = 1|Y = 1). \tag{5.7}$$

Precision and *recall* are metrics to estimate the performance of a classifier, for example $f_{\nu,\gamma,\theta}(x)$. *Precision* estimates the probability of a sample is really normal given that the classifier outputs it as normal. *Recall* estimates the probability that the classifier says a sample it is normal when it is really normal. Note that using the definition of the Bayes rule 5.4 in the probabilistic definitions of *precision* 5.6 and *recall* 5.7 we can obtain the following equality:

[1]Note that to follow the terminology of [41], the *positive class* (1) is the *normal* one in our application. With samples of this class, the OC-SVM was trained.

$$P(Y = 1 \mid f_{\nu,\gamma,\theta}(x) = 1)P(f_{\nu,\gamma,\theta}(x) = 1) = P(f_{\nu,\gamma,\theta}(x) = 1 \mid Y = 1)P(Y = 1) \tag{5.8}$$

$$\iff \frac{P(Y = 1|f_{\nu,\gamma,\theta}(x) = 1)}{P(Y = 1)} = \frac{P(f_{\nu,\gamma,\theta}(x) = 1|Y = 1)}{P(f_{\nu,\gamma,\theta}(x) = 1)}. \tag{5.9}$$

To pass form equation 5.8 to 5.9 simply divide both sides of 5.8 by the term $P(Y = 1) \times P(f_{\nu,\gamma,\theta}(x) = 1)$. Then, using the definition of the *precision* 5.6 and *recall* 5.7 in the equation 5.9 we get:

$$\frac{precision}{P(Y = 1)} = \frac{recall}{P(f_{\nu,\gamma,\theta}(x) = 1)}. \tag{5.10}$$

Multiplying both sides by *recall*, we get:

$$\frac{(precision \times recall)}{P(Y = 1)} = \frac{recall^2}{P(f_{\nu,\gamma,\theta}(x) = 1)}. \tag{5.11}$$

In [41], they call the right term of 5.11 as $\hat{F} = \frac{Recall^2}{P(f_{\nu,\gamma,\theta}(x) = 1)}$ and point out that:

$$\hat{F} \propto (precision \times recall). \tag{5.12}$$

Thus, the performance measure $\hat{F}$ is proportional to the square of the geometric mean of *precision* and *recall*. Furthermore, this metric can be estimated only by positive and unlabeled data.

$$\begin{aligned} \hat{F} &= \frac{recall^2}{P(f_{\nu,\gamma,\theta}(x) = 1)} \\ &= \frac{P(f_{\nu,\gamma,\theta}(x) = 1|Y = 1)^2}{P(f_{\nu,\gamma,\theta}(x) = 1)}. \end{aligned}$$

- $P(f_{\nu,\gamma,\theta}(x) = 1|Y = 1)$ is the probability of classifying a positive sample correctly, is the True Positive Rate (TPR)
- $P(f_{\nu,\gamma,\theta}(x) = 1)$ is the probability of classifying a sample as positive (PPP)

$$\hat{F} = \frac{TPR^2}{PPP}. \tag{5.13}$$

With these explanations in [40] it is argued that $\hat{F}$ has roughly the same behavior as the F_1-score in the sense that is large when both *precision* and *recall* are large and is small if either *precision* or *recall* is small. The best parameters ν, γ and the threshold θ are those that maximize the $\hat{F}$ metric:

$$\max_{\nu,\gamma,\theta} \frac{P(f_{\nu,\gamma,\theta}(x) = 1|Y = 1)^2}{P(f_{\nu,\gamma,\theta}(x) = 1)}. \tag{5.14}$$

5.4. Summary

This work investigates the use of a *transfer learning* technique combined with an anomaly detection model to improve the performance of web application firewalls (WAFs). Specifically, we have proposed a learning architecture composed of the two-step learning framework depicted in figure 5.7.

In a first step, we create a deep pre-trained language model using only normal HTTP requests to the web application that we aim to protect. In a second step, we use this model as a *feature extractor* and train a one-class classifier. Each web application has its own model (both the pre-trained language model and the one-class classifier). As follows, we describe the main components of the proposed learning architecture depicted in figure 5.7.

The first step (depicted in the left side of figure 5.7) consists of creating a RoBERTa language model (explained in detail in chapter 4) from scratch using a set of HTTP requests from the web application we aim to protect. As already explained, in section 5.1, the model inputs are the HTTP requests after being processed by the parser implemented by [45]. Section 5.2 presents the decisions concerned with the design of the RoBERTa architecture. Such as the tokenizer 5.2.1, the configuration of the hyper-parameters of the network 5.2.2 and the training strategy 5.2.3. The library used for the implementation and the experimental setup was explained in section 5.2.4.

In a second step (depicted in the right side of figure 5.7), we use the *feature-based* strategy (explained in detail on section 4.5.2) to transform each HTTP request (after being processed by the parser described in section 5.1) into a feature numeric vector. To be more precise, once we have trained the RoBERTa language model (in the first step), we convert each HTTP request into a nu-

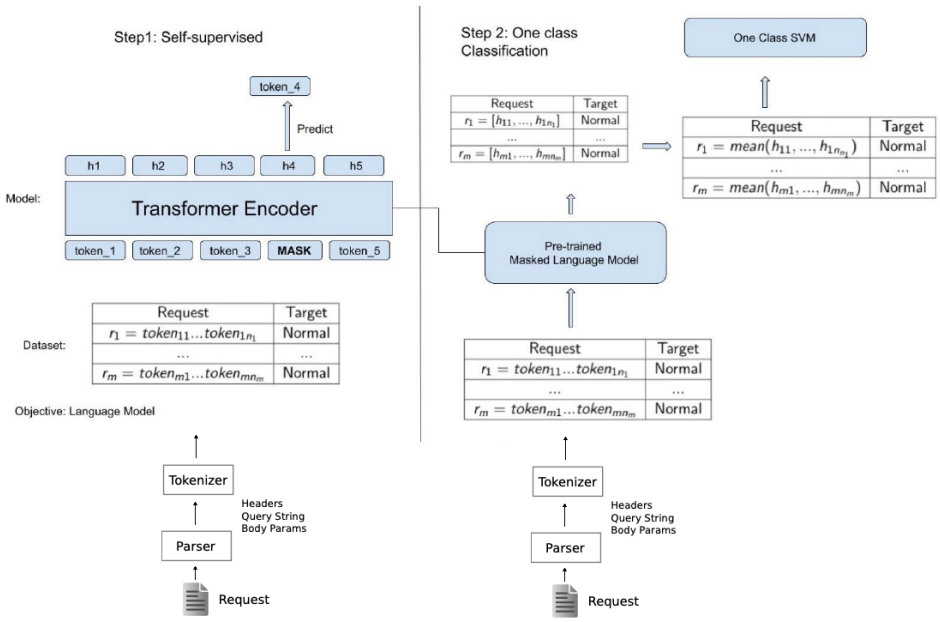


Figure 5.7: Proposed architecture. Left: transformer encoder used to extract the contextual representation of each token $token_{ij}$ in the request r_i. Right: each request r_i is represented as the mean of token deep contextualized representation h_{ij} and how they are used to train a one-class classifier.

meric representation using the weights of the last layer of the network (method known as *feature extraction*). With this representation as input, we build a One-Class Classification model (OCC) detailed in section 5.3.

As already explained, we must set up three parameters to optimize the performance of the OCSVM: ν, γ, and a threshold θ. We use the method proposed by [41] that allowed us to automatically select an operational point for the one-class model (without the need for data labeled as attacks) explained in 5.3.1. This method allows us to work in a scenario where only valid requests are used, and no requests tagged as attacks are necessary. As already mentioned, we believe that is a realistic approach, where valid traffic could be collected, for instance, from performing functional testing of the application.

Chapter 6

Experimental results

This section is devoted to present and discuss the outcomes of the experiments carried out with the proposed learning framework. We recall once more an essential premise of our work: false positives of ModSecurity configured with OWASP CRS out of the box often lead to a denial of service to valid users. Therefore, the main goals of the work presented in this thesis are the following:

- Reduce the number of false positives (FP) generated by ModSecurity without decreasing the True Positive Rate (TPR)
- Validate the use of transformer-based language models for HTTP requests to *extract features* (in a self-supervised fashion) to address web application attack detection. Our goal is to validate if this *feature extractor* can replace the participation of a security expert to define the features. Then, use these features as an input to an anomaly detection model (in a second step)
- Automatically obtain an operational point for the anomaly detection model without the need of labeled attacks for training. For that we use the performance metric proposed by [41] explained in section 5.3.2.

As already mentioned in section 2, the present thesis had as a main objective to enhance the work presented in [45]. In that work, was implemented an anomaly detection framework that experiments with two approaches for the *feature extraction* step: a classic information retrieval technique and an *expert-assisted* mechanism, where the features to be extracted are defined by a security expert. Then, with these features as input, a one-class model based

on a Gaussian mixture model is used to detect attacks. Therefore, one of the main objective of this thesis is validating whether it is possible to extract features with the help of deep learning techniques (to avoid the security expert in the *feature extraction* step).

The performance of the proposed framework is analyzed in terms of True Positive Rate (TPR) and False Positive Rate (FPR). In our case, TPR and FPR indicate the ratio of requests correctly and incorrectly classified as attacks, respectively[1].

This chapter is organized as follows, Section 6.1 describes the datasets used to conduct the experiments that were carried out. Then, Section 6.2 explains the baseline used to compare and evaluate the proposed learning framework. Finally, Section 6.3 concludes the evaluation by presenting and discussing the results of the experiments.

6.1. Datasets

We now proceed to describe the characteristics of the datasets that we have used in our experiments. Since our main objective is to improve the performance of ModSecurity, the first requirement is to use a dataset that each instance corresponds to a complete HTTP request. Also, to be aligned with [45] we shall use the same datasets used in that work. The experiments were performed on two different data sets that are briefly described in what follows.

6.1.1. CSIC2010

This dataset was developed at the *Information Security Institute* of the Spanish Research National Council (CSIC) in 2010 to test systems that were intenede to protect web application from attacks. This dataset also contains full requests and the instances are labeled as *Normal Traffic* and *Anomalous Traffic*. It was automatically developed based on real request to an e-commerce application. The dataset contains 36.000 valid requests for training, other valid 36.000 request for testing and 25.000 requests of anomalous traffic. In

[1]Note that there is a difference with the previous explanations of section 5.3, where the normal class (that is, an instance that is not an attack) was taken as the positive one (to be concise with the paper [41]). Now, to can compare with works [10] and [45], the positive class is the attack.

order to generate the anomalous traffic there were used tools such as Paros (which later became OWASP Zed Attack Proxy (ZAP) [53]) and w3af [61]. In addition to that, some valid requests were modified with typos errors in parameter names. Using this tagging mechanism, the anomalous traffic does not necessary corresponds to web application attacks, but it could also be requests generated by the crawler used by the different tools or typos errors in the parameter names. Unfortunately, we do not know how real attacks and anomalous traffic distribute in this dataset. However, this dataset seems to be a reference in the ModSecurity community, as may be understood from a note written by Christian Folini[1] who is the project leader of the OWASP CRS project.

6.1.2. DRUPAL

To experiment with a real-life application based on real requests and real attacks, we experimented with a dataset created in the work presented in [45] that captured incoming traffic to the public Website of Facultad de Ingeniería - Universidad de la República del Uruguay[2] which is based on the Drupal portal [18]. This dataset was developed using an instance of ModSecurity which recorded the incoming traffic for a period of six days to the website. This dataset specially marked the requests that were generated to the Drupal portal, discarding all other requests (e.g. access to static resources as images, PDF among others). The instances of this dataset where labeled as *Valid* and *Attacks*. As the web site is protected by an instance of ModSecurity featuring the OWASP CRS, which has been tuned for several years by a team of security and infrastructure experts. Therefore, this instance of ModSecurity was used as the labeling tool: those requests that were accepted by ModSecurity were considered as *valid traffic*, while those requests that ModSecurity rejected were tagged as *attacks*. After the tagging process, the only post-processing of this dataset consisted in blurring password values in the request. The train and testing datasets contains: 65.582 *valid* request and 1.287 real *attacks*.

[1] https://github.com/SpiderLabs/owasp-modsecurity-crs/issues/1016#issuecomment-366602493

[2] This dataset is not public, but it is available on demand to other researchers by writing to the authors

6.2. Baseline

To evaluate our proposed framework we compare our experimental results against:

- ModSecurity configured with the OWASP CRS out of the box
- The *expert-assisted* approach (for *feature extraction*) combined with a one-class model presented in [45]

In Section 6.2.1 we explain the baseline used to compare and evaluate the proposed learning framework against ModSecurity with the OWASP CRS out of the box. Then, in Section 6.2.2 we briefly describe the approach used in [45] and the performance results that were achieved in that work.

6.2.1. ModSecurity CRS baseline

As already mentioned, one of the main objectives of this work is to decrease the number of false positives of ModSecurity configured with the OWASP CRS out of the box. Therefore, in figure 6.1, we present the results of ModSecurity for each dataset configured with an OWASP CRS. These results will be used as one of the baselines when we evaluate the detection capabilities of our approach.

	DRUPAL		CSIC 2010	
ModSecurity configuration	**TPR**	**FPR**	**TPR**	**FPR**
ModSecurity OWASP CRS v3 -PL 1	29.55%	15.57%	26.62%	0.00%
ModSecurity OWASP CRS v3 -PL 2	77.89%	49.93%	29.48%	0.00%
ModSecurity OWASP CRS v3 -PL 3	78.42%	56.82%	52.61%	13.95%

Table 6.1: Summarized results of the ModSecurity baseline [45].

This baseline was generated by [45] using the ModSecurity requests generator. The ModSecurity evaluator was configured using a virtual machine with `CentOS 7`, an `Apache HTTP Server 2.4`, configured with `MODSECURITY2.7`. The experiments were developed with a version three of OWASP CRS (OWASP CRS v3), focusing on diminishing the number of false positives generated by it. Besides, we experiment with different levels of paranoia levels (PL), implemented by [45]. The PL is a configuration value that indicates how paranoid the WAF should be from a security perspective. This value ranges from 1 to 4, and it is defined as:

- PL 1: is the default paranoia level and is advised for beginners as it should face rarely FP. This setup is for applications with standard security requirements
- PL 2: is advised for moderated to experienced administrators and installations of elevated security requirements. This level comes with FP to be handled by the administrator
- PL 3: is aimed at administrators with experience at the handling of FP and at installations with a high security requirement
- PL 4: is the highest level and is advised for experienced administrators protecting installations with very high security requirements

The PL allows to change the behavior of ModSecurity depending on the main objective, have less FP, or detect most of the attacks. By analyzing the TPR and FPR values, we observe that, in general, PL 3 and PL 4 have a minor increment in TPR by producing a high amount of FP. For this reason and the expertise required for tuning a WAF with high PL, in what follows, we are going to compare our results with PL 1 and PL 2.

6.2.2. Expert-assisted baseline

In this section, we summarize the anomaly detection approach presented in [45] that explores two alternatives to extract features from the raw data (the HTTP requests), then use it as an input for a one-class model. The approaches for *feature extraction* are:

- A classic information retrieval approach by applying a bag of words model with the TF-IDF scheme,
- and an *expert-assisted* approach in which a security expert selected the features to be used.

The experimental results of [45] conclude that the use of features blindly generated from the bag of words model did not work. In [45] it is explained that the main problem was the large amount and sparsity of the extracted features because few of them were active in each instance. To address this difficulty, [45] incorporated into the analysis the experience of a security expert to identify the features manually. The approach consists of using the whole input document as a text string and count, for each token of the dictionary

constructed by the expert, the number of appearances of the feature as a sub-string of the document. Once the features were computed with the *expert-assisted* technique, in a second step a one-class classifier based on a Gaussian mixture model was trained.

The Gaussian mixture model, used as a one-class classifier, requires setting up a parameter λ to optimize the model's performance. The experiments in [45] have been carried out by varying the λ to obtain 500 different operational points and calculated for each λ the different performance metrics (TPR and FPR). As there are 500 different operational points, to apply these results to protect a real application, it is required to define the value of λ. The results of [45] in Table 6.2 were obtained with a parameter λ that was manually selected (using performance metrics such as TPR and FPR that rely on data labeled as normal and attack).

	DRUPAL		CSIC 2010	
Method	**TPR**	**FPR**	**TPR**	**FPR**
One-class classifier from [45] (*)	94.43%	6.00%	39.63%	5.37%

Table 6.2: TPR and FPR for each dataset. (*) One-class classifier using features manually selected by an expert (HTTP tokens). The operational point was manually selected by the authors.

As mentioned before, in this thesis, we propose a framework (in continuation of the work presented in [45]) to extract features with deep learning, to avoid the security expert in the *feature extraction* step. Also, we continue the work [45] by proposing a method to automatically select an operational point for a one-class model (without the need for data labeled as attacks).

Next, we present the results of the proposed approach and the comparison with the baselines explained until now.

6.3. Results

This section present and discuss the results of the experiments against both baselines (ModSecurity and [45]). The evaluation was performed, on each of the datasets, using 70% of the valid requests for training and the rest of the dataset (30% of valid and 100% of attacks) for testing.

The results of our proposal (`RoBERTa + OCSVM`) in Table 6.3 were obtained with parameters ν, γ and θ found automatically (as explained in Section 5.3).

	DRUPAL		CSIC 2010	
Method	**TPR**	**FPR**	**TPR**	**FPR**
ModSecurity OWASP CRS v3 -PL 1	29.55%	15.57%	26.62%	0.00%
ModSecurity OWASP CRS v3 -PL 2	77.89%	49.93%	29.48%	0.00%
`EA + Gaussian Mixture Model` from [45] (*)	94.43%	6.00%	39.63%	5.37%
RoBERTa + OCSVM (+)	95.00%	3.73%	47.10%	7.54%

Table 6.3: TPR and FPR for each dataset. (*) One-class classifier using features manually selected by an expert (HTTP tokens). The operational point was manually selected by the authors. (+) The operational point was automatically selected (see Section 5.3).

The best parameters for both datasets are: $\nu = 0.05$ and $\gamma = 0.5$. The optimal threshold for DRUPAL is $\theta = 0$ and for CSIC2010 is $\theta = -0.2$.

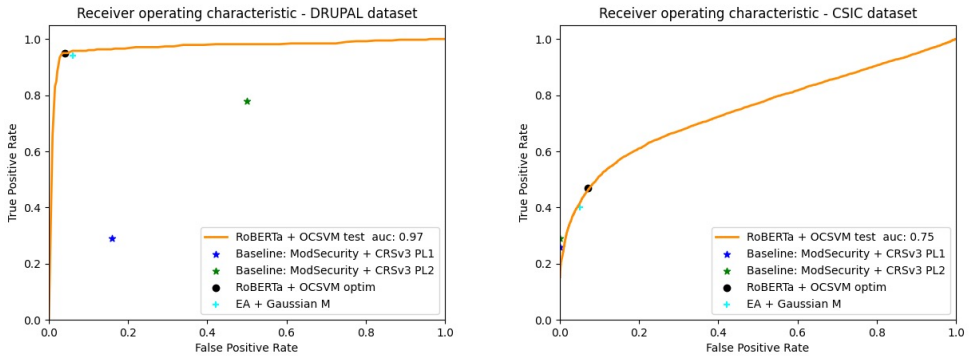


Figure 6.1: One-class ROC curve varying the θ value

In Figure 6.1 we present the results in terms of a ROC curve (constructed in terms of TPR and FPR) for each dataset.

- The solid line represents the different operation points of the `RoBERTa + OCSVM` model
- The stars represents the performance of ModSecurity (with the configuration PL1 and PL2 explained in 6.2.1)
- The plus symbol represents the operating point achieved by [45], we call it `EA + Gaussian Mixture Model` (for *Expert-Assisted* and Gaussian mixture model)
- The circle represents the operating point achieved by `RoBERTa + OCSVM` with the estimated ν, γ and θ as explained above

6.3.1. ModSecurity CRS comparison

This subsection compares the results of our proposed framework (described in table 6.3) against ModSecurity with the OWASP CRS out of the box [51].

In the DRUPAL dataset, the ROC curve shows that several points outperform all configurations of ModSecurity. If we compare the results with the ModSecurity CRS baseline, the best configuration (`ModSecurity OWASP CRS v3 with PL 2`) detects 77% of the attacks, whereas `RoBERTa + OCSVM` detects 95.00%. The FPR of ModSecurity is 49.93% and `RoBERTa + OCSVM` is 3.37%.

In the case of the CSIC2010 dataset, the TPR is higher than all versions of ModSecurity. However, it must be noticed that CSIC2010 is a synthetic dataset constructed adding some attacks and anomalous requests to a set of normal ones. For that reason, ModSecurity with paranoia levels 1 and 2 does not produce any false positives at the expense of extremely low TPR. With a more strict paranoia level (PL 3) FPR is 13.95% and TPR is 52.61% (as we can see in 6.1). Our method `RoBERTa + OCSVM` produces a lower FPR (7.54%) with a similar TPR (47.10%) for this dataset.

6.3.2. Expert-assisted comparison

The comparison with the *Expert-assisted* baseline explained in 6.2.2, shows that the results of our proposed framework have a similar performance to the `EA + Gaussian mixture` model [45] in both datasets.

In the DRUPAL dataset, the results are a little better than the `EA + Gaussian mixture model`; the TPR increase very little 0.57%; and the FPR decrease by 2.27%.

In the CESIC2010 dataset, the TPR increases by 7.47%; however, the FPR also increases 2.17%.

The results show a performance very similar to the *expert-assisted* approach quantitatively in both datasets. However, we were able to move forward in the research qualitatively in the following directions. The proposed method allows to avoid the security expert in the *feature extraction* step. Also, with the method proposed by [41] allowed us to automatically select an operational point for the one-class model (without the need for data labeled as attacks). Therefore, we work in a scenario where only valid requests are used, and no requests tagged as attacks are necessary. As already mentioned, we believe

that is a realistic approach, where valid traffic could be collected, for instance, from performing functional testing of the application.

Chapter 7

Conclusion and further work

To the best of our knowledge our proposed framework is the first attempt in using transformer-based language representation of HTTP requests to address the problem of web applications attack detection.

We used two different datasets to pre-train a deep language model for the HTTP requests without requiring a security expert to define the set of features. We have proposed a two-step learning approach consisting of first mapping the HTTP requests into a continuous space using a transformer encoder and then applying an OCSVM to discriminate normal traffic from attacks. In addition to that, we have adapted a performance metric proposed by [41] to automatically obtain the parameters of the OCSVM.

The results presented in Section 6.3 are quite promising. They outperform ModSecurity using the most widely adopted rules and are slightly better than those shown in [45] with the advantage of not requiring the participation of a security expert to define the features. We have also automatically achieved an operational model (without the need for data labeled as attacks) that can be used in a production environment.

In conclusion, the proposed learning framework allowed us to validate the use of transformer-based language models for HTTP requests to *extract features*, in a self-supervised manner, to address web application attack detection. We have decided to design the framework separating the problem into two steps and then integrating them. This procedure allowed us to delve into each one separately. For the *feature extraction* step we have analyzed different approaches, as exposed in chapter 3. After exploring all the alternatives, we have decided to experiment with the RoBERTa architecture. Then for the second

step, we have used a classic one-class model. However, we believe that it could be interesting as future work to use an end-to-end neural networks system, so as to, for instance, be able to perform *fine-tuning* of the transformer-network by adding a prediction layer capable of being optimized for an anomaly detection problem. For doing that it would be needed to modify the architecture of RoBERTa so as for it to allow us to perform a one-class classification procedre. An alternative procedute would be to use a Convolutional Neural Network, optimized for performing a one-class classification, like the one presented in [54]) for the second step.

Another possible improvement would be to refine the hyper-parameters of the network for the language model. As we have explained in section 5.2, the RoBERTa language model that we train is composed of a stack of **L=12** identical transformer encoder layers, each bearing a hidden size **H=768** and 12 (twelve) self-attention heads. Our model embodies that set of hyper-parameters because we have chosen to use the same configuration defined for the original implementation of RoBERTa, which is reported in [43]. In future work, however, it would be interesting to experiment with another configuration of hyper-parameters and check whether the performance can be improved. The RoBERTa language model, in addition, could be trained with the DRUPAL dataset including more HTTP requests.

Another component that can be modified to achieve an improvement is the tokenizer of the language model. As we have explained in section 4.3.1, in this work, we use the same tokenizer learned by Radford [59], a clever implementation of BPE that uses bytes instead of Unicode characters as the basic sub-words units. The authors learned a *sub-words* vocabulary of $\approx 50K$ units that can still encode any input string without introducing any *unknown* tokens. We could have chosen another pre-trained BPE tokenizer instead of the one trained by [59]. The key point was to use a BPE tokenizer trained on a vast corpus ($\approx$ 40 GB of text data scraped from the web). Thus it can tokenize any word (and any character) of any language without using the *unknown* token. That tokenizer allowed us to process the data for the two applications we want to protect without recognizing any token as *unknown*, since if a token is not in the vocabulary, it is built with individual characters. An improvement would be to learn our own tokenizer from scratch taking as input a significant HTTP traffic of the application requests we try to protect.

Bibliography

[1] *A Gentle Introduction to the Bag-of-Words Model*. https://machinelearningmastery.com/gentle-introduction-bag-words-model/. (Accessed on 08/16/2021).

[2] Alon U et al. "code2vec: Learning distributed representations of code". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.

[3] Ariu D. "Host and Network based Anomaly Detectors for HTTP Attacks". Dept. of Electrical and Electronic, Engineering University of Cagliari, 2010.

[4] Ba JL, Kiros JR, and Hinton GE. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[5] Bahdanau D, Cho K, and Bengio Y. "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (2014).

[6] Bengio Y et al. "A neural probabilistic language model". In: *The journal of machine learning research* 3 (2003), pp. 1137–1155.

[7] Berners-Lee T, Fielding R, and Masinter L. "RFC 2396: uniform resource identifiers (URI)". In: *IETF RFC* (1998).

[8] *BERT Fine-Tuning Tutorial with PyTorch · Chris McCormick*. https://mccormickml.com/2019/07/22/BERT-fine-tuning/. (Accessed on 04/09/2021).

[9] Betarte G, Martınez R, and Pardo Á. "Web Application Attacks Detection Using Machine Learning Techniques". In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2018, pp. 1065–1072.

[10] Betarte G et al. “Improving Web Application Firewalls through Anomaly Detection”. In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2018, pp. 779–784.

[11] Blei DM, Ng AY, and Jordan MI. “Latent dirichlet allocation”. In: *the Journal of machine Learning research* 3 (2003), pp. 993–1022.

[12] Bojanowski P et al. “Enriching word vectors with subword information”. In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 135–146.

[13] Collobert R et al. “Natural language processing (almost) from scratch”. In: *Journal of machine learning research* 12.ARTICLE (2011), pp. 2493–2537.

[14] Corona I, Ariu D, and Giacinto G. “HMM-Web: A Framework for the Detection of Attacks Against Web Applications”. In: *Proceedings of ICC 2009*. 2009, pp. 1–6.

[15] Davies M and FERREIRA M. “The Wikipedia Corpus: 4.6 million articles, 1.9 billion words”. In: *Adapted from Wikipedia. Accessed February* 15 (2015).

[16] Deerwester S et al. “Indexing by latent semantic analysis”. In: *Journal of the American society for information science* 41.6 (1990), pp. 391–407.

[17] Devlin J et al. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).

[18] *Drupal - Open Source CMS — Drupal.org*. https://www.drupal.org/. (Accessed on 06/06/2021).

[19] Dumais ST et al. “Using latent semantic analysis to improve access to textual information”. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1988, pp. 281–285.

[20] Ethayarajh K. “How contextual are contextualized word representations? comparing the geometry of BERT, ELMo, and GPT-2 embeddings”. In: *arXiv preprint arXiv:1909.00512* (2019).

[21] Feng Z et al. “Codebert: A pre-trained model for programming and natural languages”. In: *arXiv preprint arXiv:2002.08155* (2020).

[22] Folini C. *Handling False Positives with the OWASP ModSecurity Core Rule Set*. 2016. URL: https://www.netnea.com/cms/apache-tutorial-8_handling-false-positives-modsecurity-core-rule-set/.

[23] Graves A. "Generating sequences with recurrent neural networks". In: *arXiv preprint arXiv:1308.0850* (2013).

[24] Hacker AJ. "Importance of Web Application Firewall Technology for Protecting Web-based Resources". In: *ICSA Labs an Independent Verizon Business* (2008).

[25] Harris ZS. "Distributional structure". In: *Word* 10.2-3 (1954), pp. 146–162.

[26] Hendrycks D and Gimpel K. "Gaussian error linear units (gelus)". In: *arXiv preprint arXiv:1606.08415* (2016).

[27] Hofmann T. "Probabilistic latent semantic indexing". In: *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. 1999, pp. 50–57.

[28] Howard J and Ruder S. "Universal language model fine-tuning for text classification". In: *arXiv preprint arXiv:1801.06146* (2018).

[29] *HTTP headers - HTTP — MDN*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers. (Accessed on 08/10/2021).

[30] Jain AK, Duin RPW, and Mao J. "Statistical pattern recognition: A review". In: *IEEE Transactions on pattern analysis and machine intelligence* 22.1 (2000), pp. 4–37.

[31] Joos M. "Description of language design". In: *The Journal of the Acoustical Society of America* 22.6 (1950), pp. 701–707.

[32] Jurafsky D and Martin J. *Speech & language processing*. Pearson Education India, 3rd Edition - Draft December, 2020.

[33] Kayal S and Tsatsaronis G. "Eigensent: Spectral sentence embeddings using higher-order dynamic mode decomposition". In: *Proceedings of the 57th annual meeting of the association for computational linguistics*. 2019, pp. 4536–4546.

[34] Kingma DP and Ba J. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[35] Krizhevsky A and Hinton G. “Convolutional deep belief networks on cifar-10”. In: *Unpublished manuscript* 40.7 (2010), pp. 1–9.

[36] Kruegel C and Vigna G. “Anomaly detection of web-based attacks”. In: *Proceedings of CCS 2003*. ACM. 2003, pp. 251–261. ISBN: 1-58113-738-9.

[37] Lachaux MA et al. “Unsupervised translation of programming languages”. In: *arXiv preprint arXiv:2006.03511* (2020).

[38] Lample G et al. “Phrase-based & neural unsupervised machine translation”. In: *arXiv preprint arXiv:1804.07755* (2018).

[39] Le Q and Mikolov T. “Distributed representations of sentences and documents”. In: *International conference on machine learning*. PMLR. 2014, pp. 1188–1196.

[40] Lee DD and Seung HS. “Learning the parts of objects by non-negative matrix factorization”. In: *Nature* 401.6755 (1999), pp. 788–791.

[41] Lee WS and Liu B. “Learning with positive and unlabeled examples using weighted logistic regression”. In: *ICML*. Vol. 3. 2003, pp. 448–455.

[42] Li H et al. “Pruning filters for efficient convnets”. In: *arXiv preprint arXiv:1608.08710* (2016).

[43] Liu Y et al. “Roberta: A robustly optimized bert pretraining approach”. In: *arXiv preprint arXiv:1907.11692* (2019).

[44] Louis A. “NetBERT: A Pre-trained Language Representation Model for Computer Networking”. Faculté des Sciences appliquées, 2020.

[45] Martínez R. “Enhancing web application attack detection using machine learning”. MA thesis. Facultad de Ingeniería,UdelaR - Área Informática del Pedeciba, Uruguay, 2019.

[46] Maurel H, Vidal S, and Rezk T. “Statically Identifying XSS using Deep Learning”. In: *SECRYPT 2021-18th International Conference on Security and Cryptography*. 2021.

[47] McCann B et al. “Learned in translation: Contextualized word vectors”. In: *arXiv preprint arXiv:1708.00107* (2017).

[48] Mikolov T et al. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013).

[49] Mikolov T et al. “Recurrent neural network based language model”. In: *Eleventh annual conference of the international speech communication association*. 2010.

[50] Ott M et al. “fairseq: A fast, extensible toolkit for sequence modeling”. In: *arXiv preprint arXiv:1904.01038* (2019).

[51] OWASP. *OWASP ModSecurity Core Rule Set Project*. URL: https://coreruleset.org. Last visited on 14/02/2021.

[52] OWASP. *OWASP Top Ten Project*. URL: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. Last visited on 14/02/2021.

[53] *OWASP ZAP Zed Attack Proxy — OWASP*. https://owasp.org/www-project-zap/. (Accessed on 06/06/2021).

[54] Oza P and Patel VM. “One-Class Convolutional Neural Network”. In: *IEEE Signal Processing Letters* 26.2 (2019), pp. 277–281. DOI: 10.1109/LSP.2018.2889273.

[55] Paszke A et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by Wallach H 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.

[56] Pennington J, Socher R, and Manning CD. “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[57] Peters ME et al. “Deep contextualized word representations”. In: *arXiv preprint arXiv:1802.05365* (2018).

[58] Qin ZQ, Ma XK, and Wang YJ. “Attentional payload anomaly detector for web applications”. In: *International Conference on Neural Information Processing*. Springer. 2018, pp. 588–599.

[59] Radford A et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* 1.8 (2019), p. 9.

[60] Ramshaw LA and Marcus MP. “Text chunking using transformation-based learning”. In: *Natural language processing using very large corpora*. Springer, 1999, pp. 157–176.

[61] Riancho A. “w3af-web application attack and audit framework”. In: *World Wide Web electronic publication* (2011), p. 21.

[62] Roziere B et al. “DOBF: A Deobfuscation Pre-Training Objective for Programming Languages”. In: *arXiv preprint arXiv:2102.07492* (2021).

[63] Rumelhart DE and Abrahamson AA. “A model for analogical reasoning”. In: *Cognitive Psychology* 5.1 (1973), pp. 1–28.

[64] *Salmon Run: Adding a Transformer based NER model into NERDS*. http://sujitpal.blogspot.com/2020/01/adding-transformer-based-ner-model-into.html. (Accessed on 08/19/2021).

[65] Schölkopf B et al. “Estimating the support of a high-dimensional distribution”. In: *Neural computation* 13.7 (2001), pp. 1443–1471.

[66] Schütze H and Pedersen J. “A vector model for syntagmatic and paradigmatic relatedness”. In: *Proceedings of the 9th Annual Conference of the UW Centre for the New OED and Text Research*. Citeseer. 1993, pp. 104–113.

[67] Sennrich R, Haddow B, and Birch A. “Neural machine translation of rare words with subword units”. In: *arXiv preprint arXiv:1508.07909* (2015).

[68] Staub A et al. “The influence of cloze probability and item constraint on cloze task response time”. In: *Journal of Memory and Language* 82 (2015), pp. 1–17.

[69] Sureda Riera T et al. “Prevention and Fighting against Web Attacks through Anomaly Detection Technology. A Systematic Review”. In: *Sustainability* 12.12 (2020). ISSN: 2071-1050. DOI: 10.3390/su12124945. URL: https://www.mdpi.com/2071-1050/12/12/4945.

[70] Taylor WL. ““Cloze procedure”: A new tool for measuring readability”. In: *Journalism quarterly* 30.4 (1953), pp. 415–433.

[71] *The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning) – Jay Alammar – Visualizing machine learning one concept at a time*. http://jalammar.github.io/illustrated-bert/. (Accessed on 04/09/2021).

[72] *The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning) – Jay Alammar – Visualizing machine learning one concept at a time.* http://jalammar.github.io/illustrated-bert/. (Accessed on 08/17/2021).

[73] *The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time.* http://jalammar.github.io/illustrated-transformer/. (Accessed on 02/14/2021).

[74] Torrano-Gimenez C, Perez-Villegas A, Marañón GÁ, et al. "An anomaly-based approach for intrusion detection in web traffic". In: *Journal of Information Assurance and Security* 5.4 (2010), pp. 446–454.

[75] *Transformer Architecture: The Positional Encoding - Amirhossein Kazemnejad's Blog.* https://kazemnejad.com/blog/transformer_architecture_positional_encoding/. (Accessed on 04/07/2021).

[76] Trustwave Holdings I. *ModSecurity: Open Source Web Application Firewall.* URL: http://www.modsecurity.org/.

[77] Valeur F, Mutz D, and Vigna G. "A learning-based approach to the detection of SQL attacks". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2005, pp. 123–140.

[78] Vaswani A et al. "Attention is all you need". In: *arXiv preprint arXiv:1706.03762* (2017).

[79] *What is TF-IDF?* https://monkeylearn.com/blog/what-is-tf-idf. (Accessed on 04/17/2021).

[80] Widdowson H. "Jr firth, 1957, papers in linguistics 1934–51". In: *International Journal of Applied Linguistics* 17.3 (2007), pp. 402–413.

[81] Yang Z et al. "Xlnet: Generalized autoregressive pretraining for language understanding". In: *arXiv preprint arXiv:1906.08237* (2019).

[82] Yu Y et al. "DeepHTTP: semantics-structure model with attention for anomalous HTTP traffic detection and pattern mining". In: *arXiv preprint arXiv:1810.12751* (2018).

[83] Yuan G et al. "A deep learning enabled subspace spectral ensemble clustering approach for web anomaly detection". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2017, pp. 3896–3903.

[84] Zhang Y, Jin R, and Zhou ZH. "Understanding bag-of-words model: a statistical framework". In: *International Journal of Machine Learning and Cybernetics* 1.1-4 (2010), pp. 43–52.