UNIVERSIDAD DE MURCIA

FINAL DEGREE PROJECT

Syntactic analysis with natural language processing using large language models

Author:
Francisco José CORTÉS DELGADO
franciscojose.cortesd@um.es

Supervisors:
Eduardo MARTÍNEZ GRACIÁ
edumart@um.es
Rafael VALENCIA GARCÍA
valencia@um.es

June 2023

This bachelor's degree project is rendered entirely in English, aiming to reach a broad and diverse audience. However, I'd like to take a moment to express my deepest gratitude in my native language, Spanish, to all the individuals without whose support this work would not have been possible.

En primer lugar, me gustaría agradecer a mis tutores de este trabajo fin de grado (TFG), a Rafael Valencia y a Eduardo Martínez Graciá. Especial agradecimiento a Eduardo, ya que sin él este trabajo tan ambicioso no habría sido posible. Me acogió como alumno interno desde el momento en el que, en segundo de carrera, al final de uno de sus seminarios de Autómatas y lenguajes formales, me acerqué junto a mi compañero, amigo y cofundador de MiSintaxis [18] Gonzalo Cánovas López de Molina, con una idea feliz que se me había ocurrido. Aprovechar lo que nos había enseñado sobre las gramáticas libres de contexto para realizar un analizador sintáctico automático para nuestra aplicación. En tercero de carrera, mientras estaba en Polonia de Erasmus, Gonzalo empezó esta tarea. A mi vuelta nos dimos cuenta de que necesitábamos la potencia de los grandes modelos del lenguaje y descubrimos que esto daba para un TFG. Lo que me lleva a darle las gracias también a Gonzalo por seguirme desde que nos conocemos en esta aventura emprendedora.

En segundo lugar quiero dar las gracias a todos los que trabajaron en el desarrollo del corpus de AnCora, a los que participan en el grupo Plan de Tecnologías del Lenguaje - Gobierno de España y a los que trabajaron para desarrollar GPT2 y BLOOM. Gracias al grupo de investigación Sistemas Inteligentes y Telemática en concreto a Antonio Skármeta y Luis Bernal por permitirme entrenar los modelos en una máquina de su grupo. Gracias también a Alejandra por ayudarme a entender el corpus de Ancora. Gracias a todos los lingüistas, especialmente a Pascual Cantos, Santiago Roca y a Ana Bravo.

En tercer lugar, quiero dar las gracias a mis padres, a mi padre, Gabriel Cortés García, por contagiarme sus ganas de aprender. Y es que sin su influencia no habría estudiado esta carrera. A sus 30 años cambió su vida laboral del mundo de las letras a la informática. A sus 60 años sigue entrenando su cuerpo y mente. Con ganas de adquirir nuevos conocimientos en distintos ámbitos, desde nuevas lenguas a nuevas tecnologías. Con 15 años le pregunté qué tenía que hacer para crear una empresa tecnológica y su respuesta fue aprender a programar. Al día siguiente me regaló el libro que él había usado para aprender a programar, "Piensa en java" [6]. Gracias a mi madre María José Delgado Franco por transmitirme también sus ganas de aprender y su amor por la economía, lo que hizo que desde los 13 años, me interesara por el mundo del emprendimiento.

Gracias a mis amigos y al resto de mi familia, en especial a mis hermanos Gabri y Guille. En primero, segundo y tercero de carrera, tuve un problema en el brazo derecho que me impedía escribir. Fueron mi familia y amigos los que me ayudaron a preparar los exámenes y a realizar mis tareas del día a día. Sin todos ellos no habría podido estudiar esta carrera. En especial gracias de corazón a Cristina Fresneda Riquelme por acompañarme y hacerme mejor persona estos últimos 6 años de mi vida.

Quiero dar las gracias a todos los integrantes de MiSintaxis. En especial a aquellos que llevan más tiempo, Gonzalo Cánovas López de Molina, Laura Mateo Galindo, Tomás Bernal Beltrán y Mario Rodríguez Béjar. Por confiar en este proyecto y en este loco que quiere acercar la sintaxis al mayor número de estudiantes posibles. Podemos decir con orgullo que hemos superado las 70.000 descargas en estos 3 últimos años. Gracias también a Jose Antonio García Cartagena por su ayuda en AWS y por enseñarnos cómo debe trabajar una compañía de software.

Quiero expresar mi gratitud a la comunidad emprendedora de la Región de Murcia, así como a todas las instituciones que me han apoyado con MiSintaxis. Me siento enormemente orgulloso de lo que estamos construyendo en nuestra región. También deseo agradecer a la comunidad universitaria de ingeniería informática, la cual, a diferencia de otras, se distingue por compartir su conocimiento de forma altruista, sin el deseo de pisotear a los compañeros. Este espíritu se ve reflejado en las iniciativas de código abierto que han impulsado grandes proyectos como Linux, Github o Hugging Face.

Espero poder contribuir a la comunidad con este trabajo de fin de grado. También quiero agradecer a los divulgadores como Carlos Santana Vega (DotCSV), quien me motivó a aprender sobre Inteligencia Artificial, a Juan Sensio de SensioCoders, cuyos cursos he realizado para prepararme para afrontar este TFG. Agradezco a José Ignacio Gentile (Nate Gentile), uno de los canales que me han motivado durante mi carrera. Mi agradecimiento también para Lex Fridman y Javier Santaolalla.

Quisiera expresar mi gratitud a todos los youtubers que hacen videotutoriales; su trabajo es a menudo infravalorado. Si no fuera por ellos, no habría aprendido muchas de las cosas que me han permitido llevar a cabo este trabajo.

Por último, quiero hacer mención especial a mis amigos de toda la vida que siempre confiaron en mí y a mis amigos de la carrera, Carmona, Felipe, Gonzalo, Juanjo, Silvia, César, Pablo y a mi compañera de prácticas Raquel. Por apoyarme durante todo este viaje.

Contents

List of Figures

Summary

This document presents an approach for automatic constituency parsing of Spanish sentences based on fine-tuning Large Language Models like Bloom or GPT2 using the seq2seq approach. Furthermore, it aims to ensure the widespread accessibility of this system. To achieve this, we use the Amazon Web Services platform for hosting and distribution. The successful completion of this project will benefit MiSintaxis [18] application, thus providing quality education to its thousands of users worldwide.

In this project, we initially delve into the history of Spanish grammar studies, exploring its components and the methodologies employed in teaching it at the elementary and secondary education levels. This analysis serves as a foundational understanding, informing the subsequent stages of our research and development.

Subsequently, we present a review of the state-of-the-art developments in Natural Language Modeling and Parsing. We traverse the history of Neural Networks and their application in the realm of Natural Language Modeling, discussing the evolution of various architectures that laid the foundation for the advent of Transformers. We meticulously explore the intricacies of Transformer architecture, focusing on the critical elements that propelled the success of Large Language Models. In addition, we introduce the Hugging Face ecosystem, a notable platform that fosters the accessibility and usability of these advanced models. We also shed light on traditional parsing algorithms, delineating their role and significance in the broader context of language parsing.

Using an automatic process, we converted the Spanish AnCora corpus using our grammar notation based on the recommendations of the *Nueva gramática BÁSICA de la lengua española* [15]. This process resulted in a Spanish corpus comprising 500,000 words spread across 17,300 sentences, thus encompassing the entirety of AnCora.

We fine tuned Hugging Face models bloom-560m, bloom-1b1, gpt2-base-bne and gpt2-larg-bne with this customized corpus and compared them using the F_1 metric over the test dataset from the Ancora corpus, obtaining the following scores: 0.8141 for gpt2-larg-bne, 0.7939 for bloom-560m, 0.7790 for bloom-1b1, 0.7234 for gpt2-base-bne. With a simplified test dataset that we called the Argos dataset, we obtain the following F_1 scores: 0.9123 for bloom-1b1, 0.8642 for bloom-560m, 0.8321 for gpt2-larg-bne, 0.8190 for gpt2-base-bne.

Finally, we present Amazon Web Services and how to deploy a large language model for a real use scenario.

Resumen extendido

Este trabajo fin de grado aborda el análisis sintáctico automático de frases en español reentrenando grandes modelos del lenguaje como Bloom o GPT2, tomando un enfoque secuencia a secuencia como el usado en procesos de traducción entre idiomas. Además, en este trabajo tratamos no solo de abrir una nueva línea de investigación en la tarea del análisis sintáctico, en concreto en el análisis de constituyentes, sino que también tratamos de hacer llegar esta herramienta a los miles de usuarios que hoy en día usan nuestra aplicación MiSintaxis [18], que cuenta con más de 70,000 descargas. Para ello se ha realizado un estudio de los distintos servicios que ofrece Amazon Web Services (AWS), creando la arquitectura basada en microservicios que mejor se adaptaba a nuestro objetivo y realizando además un análisis de costes.

En primer lugar, en el proyecto se presenta cómo se ha abordado el estudio de la gramática del español a lo largo de la historia. Se describe, además, la problemática de la enseñanza de la sintaxis, principalmente en España, que dio lugar a que apareciese MiSintaxis. En el trabajo definimos la sintaxis como la parte de la gramática a la que corresponde el análisis de la manera en que las palabras se combinan y se disponen linealmente, así como el de los grupos que forman. Nosotros nos centramos en la parte de la sintaxis conocida como el análisis de constituyentes, donde se identifica la estructura jerárquica de la frase en forma de un árbol en el que un nodo intermedio puede tener múltiples nodos hijos; en las hojas del árbol se encuentran las palabras de la frase.

Actualmente, la enseñanza de la sintaxis está incluida dentro de la materia de Lengua Castellana y Literatura de la ESO y Bachillerato, y en los exámenes de acceso a la universidad en España se pide un análisis sintáctico como el que se desarrolla en este trabajo mediante modelos del lenguaje. Los ejercicios incluidos en los libros de texto habitualmente no incluyen una solución que permita al estudiante un estudio autónomo. Por otra parte, la exposición de los contenidos es extensa y a veces trata aspectos complejos sin suficientes ejemplos. Gracias al modelo del lenguaje que se ha entrenado en este trabajo, y a su puesta en producción como microservicio en AWS, los estudiantes tendrán una cantidad infinita de ejemplos.

En este proyecto se trata de explicar el estado del arte en la tarea de análisis de constituyentes. Para ello empezamos hablando de n-gramas, útiles en tareas de procesamiento del lenguaje natural (PLN) porque permiten capturar dependencias locales, analizando la frecuencia de aparición de secuencias de n palabras en un corpus. Esto se usa para encontrar relaciones semánticas entre las palabras. Los n-gramas se ven superados por las redes neuronales en la tarea que tratamos de desempeñar.

Las redes neuronales son un tipo de algoritmo que aprende a reconocer patrones entre los datos sobre los que se entrena. En este trabajo se cubre desde el origen del perceptrón o neurona en 1958 hasta la llegada de la arquitectura Transformers en 2017. Primero presentamos formalmente el perceptrón donde cada nodo de entrada se conecta con un nodo de salida mediante lo que denominamos *peso*, pasando por una función de activación. Con una función de pérdida tratamos de disminuir en cada paso del entrenamiento la diferencia entre la salida dada y la esperada.

Después presentamos el perceptrón multicapa que surge ante la incapacidad del perceptrón de resolver problemas no lineales. A partir de la unión de varios perceptrones en distintas capas, formando una estructura similar a una red neuronal, se pueden capturar mediante funciones no lineales patrones y relaciones más complejas entre los datos. Para garantizar el aprendizaje se necesita una buena función de activación no lineal. Aquí presentamos las funciones ReLU, tangente hiperbólica y sigmoide.

Antes de continuar con el estado del arte explicamos cómo se entrenan modelos del lenguaje para tareas de procesamiento de lenguaje natural. Para realizar el entrenamiento se suelen recopilar grandes conjuntos de texto (datasets), que son normalizados y tokenizados. La tokenización es una parte clave del proceso de entrenamiento en la que se pasan las palabras o datos a una representación basada en vectores numéricos o *embeddings* que pueden ser utilizados en operaciones matemáticas. Esta tokenización puede realizarse sobre palabras enteras o partes de palabras. Los modelos del lenguaje se basan en redes neuronales que pueden aprender a reconocer patrones entre los datos y, en el caso del lenguaje natural, pueden tratar de predecir la probabilidad de la siguiente palabra en una secuencia, cumpliendo el objetivo de la tarea de modelado del lenguaje.

Para entrenar las estructuras compuestas por redes neuronales se presentan distintos ejemplos del dataset, dejando palabras en blanco para las cuales conocemos su respuesta. A continuación, el modelo trata de adivinar la palabra. Hacemos esto en repetidas ocasiones, por ejemplo quitando algunas palabras de un texto conocido y luego mediante una función de pérdida optimizamos los pesos de la red, para minimizar la diferencia entre las predicciones y los datos reales. Para realizar un buen entrenamiento se divide el dataset en tres partes: dataset de entrenamiento, de evaluación y de test. La proporción suele ser del 80, 10 y 10 por ciento respectivamente. Una vez el modelo esté entrenado, puede usarse para distintas tareas de PLN.

Las distintas arquitecturas del deep learning se basan en redes neuronales, como por ejemplo las redes neuronales recurrentes, que permiten tratar con datos lineales como el texto, las cuales mantienen la información en el tiempo gracias a un estado oculto que contiene información sobre datos anteriores, pero que se desvanece con el tiempo. Las LSTM, que vienen a mejorar los resultados de las redes neuronales recurrentes ante este problema de desvanecimiento con celdas de memoria que permiten de forma selectiva olvidar o recordar información sobre largos periodos de tiempo. Los Encoder-Decoder donde el "Encoder" es responsable de procesar la secuencia de entrada y codificarla en un vector de longitud fija usando una red neuronal recurrente como una LSTM. Y el "Decoder" el cual se encarga de generar la secuencia basándose en la representación del "Encoder" usando también una red neuronal recurrente.

Los modelos usados en este trabajo, así como el famoso ChatGPT, están basados en la conocida arquitectura Transformers, que desde que apareció en 2017 con el artículo *Attention is all you need* [20], ha revolucionado el mundo del PLN, convirtiéndose en el actual estado del arte. En este proyecto se explica su estructura al completo y la llamada *Atención*. Esta estructura usa un mecanismo conocido como self-attention que permite conocer la importancia de diferentes posiciones en una cadena de entrada a la hora de hacer predicciones, permitiendo saber a qué prestar atención cuando se realiza la predicción. Esto le da la posibilidad de capturar relaciones de dependencia entre partes de la secuencia muy lejanas entre sí. Otro mecanismo usado en los Transformers es el *positional encoding*: como el modelo procesa todas las palabras en paralelo en vez de secuencialmente, se pierde la posición de un elemento de la secuencia de entrada, por ejemplo, de una palabra en una frase; gracias al *posicional encoding* conseguimos mantener la información posicional.

La capacidad de tratar con gran cantidad de información en paralelo, permitiendo prestar atención a zonas concretas de la entrada y sin perder su información posicional, hace a la arquitectura Transformer interesante en tareas que van desde la visión por computación hasta tareas de PLN.

A continuación, se presenta el ecosistema Hugging Face, que cuenta con un Hub donde se almacenan distintos modelos de inteligencia artificial, así como el framework usado en este trabajo para reentrenar los modelos. Hugging Face a tomado un rol importantísimo en la democratización del estado del arte en las distintas tareas de inteligencia artificial y ha permitido la realización de este trabajo.

Después se continúa explicando la tarea de PLN conocida como *parsing* que engloba el analisis de dependencias donde se muestran las relaciones entre la cabeza de la frase y todas las otras palabras, y el análisis de constituyentes del que hemos hablado anteriormente y el cual vamos a llevar a cabo de forma automática.

Un algoritmo famoso para esta tarea es el Cocke–Younger–Kasami (CYK), un algoritmo de programación dinámica basado en el uso de gramáticas libres de contexto para modelar el lenguaje natural puesto en forma normal de Chomsky. Este algoritmo puede ser usado junto con grandes modelos del lenguaje que determinan las mejores alternativas para hacer el análisis de una frase.

Nosotros en concreto tomamos la idea de realizar el análisis de constituyentes basándonos en la tarea traducción de secuencia a secuencia con modelos del lenguaje en la que, dada una entrada, el modelo debe dar una salida concreta. En nuestro caso, dada una frase, el modelo debe de generar su análisis de constituyentes. Hay trabajos importantes que tratan de conseguir realizar la tarea del análisis de constituyentes de un lenguaje. En concreto, la tesis doctoral de Chiruzzo [3] trata de lograrlo para el español. En la mayoría de trabajos se trata de abordar esta tarea con redes LSTM, o mediante mecanismos de atención en los más recientes. En este trabajo se propone un enfoque distinto: reentrenar un modelo de generación de texto para la tarea específica de, dada una frase, generar su análisis de constituyentes.

Para reentrenar un modelo del lenguaje necesitamos, como hemos comentado antes, un dataset que, en el caso del lenguaje natural, se suele llamar *corpus*. Nosotros usamos el de AnCora-ES [12], con aproximadamente 500.000 palabras y 17.300 frases, en su mayoría compuesto por artículos de prensa. Contiene muchos niveles de anotación en formato XML, como morfológica, sintáctica o semántica, al igual que identificación de entidades y correferencias entre constituyentes. Nosotros nos centramos en las etiquetas y atributos XML que identifican funciones sintácticas y, en algunos casos, morfológicas.

Este corpus ha sido adaptado para el entrenamiento de los modelos, para llevar a cabo la tarea que queremos en MiSintaxis, o sea el análisis sintáctico que se realiza en las aulas españolas, basándonos en las etiquetas usadas en la *Nueva gramática BÁSICA de la lengua española* [15], inspirándonos en el formato del Peen Treebank [11] donde las estructuras sintácticas se delimitan con paréntesis que contienen un primer elemento que etiqueta la estructura y una lista de elementos separados por espacios simples que contienen estructuras sintácticas que representan al contenido. Únicamente se ha tomado la precaución de cambiar los paréntesis por corchetes porque en español se usan los paréntesis como signos de puntuación. Un ejemplo de la notación usada es la siguiente:

```
<s>Yo tengo un amigo alto</s><s>[O.Simple [GN/S [N Yo]] [GV/PV [NP tengo]
[GN/CD [Det un] [N amigo] [GAdj/CN [Adj alto]]]] [Punc .]]</s>
```

Cuatro modelos del Hub de Hugging Face han sido reentrenados; bigscience/bloom-560m, bigscience/bloom-1b1, PlanTL-GOB-ES/gpt2-base-bne y PlanTL-GOB-ES/gpt2-larg-bne, con este corpus y han sido comparados con la métrica F1. Se han obtenido los siguientes resultados para la parte del dataset de Ancora reservado al test: 0.7790 para bloom-1b1, 0.8141 para gpt2-larg-bne, 0.7939 para bloom-560m, 0.7234 para gpt2-base-bne; y estos resultados para el dataset de Argos (39 frases analizadas extraídas del libro de texto del Proyecto ARGOS Murcia [7]): 0.9123 para bloom-1b1, 0.8642 para bloom-560m, 0.8321 para gpt2-larg-bne, 0.8190 para gpt2-base-bne.

Hemos realizado un estudio de los servicios que ofrece AWS para seleccionar aquellos que se emplearán para la puesta en producción de este proyecto como un microservicio cloud consumible a partir de una API. Se han estudiado los siguientes servicios: Amazon Lambda functions, API Gateway, S3 Bucket, Amazon Elastic Compute Cloud (EC2), ECR y Amazon Sagemaker. Para crear la arquitectura del microservicio hemos usado los servicios indicados a continuación.

S3 Bucket es el servicio de almacenamiento de AWS, que hemos usado para almacenar el modelo de inteligencia artificial. API Gateway, es el servicio que permite a los desarrolladores crear, publicar, monitorizar, mantener y securizar APIs a cualquier escala; en el caso de este trabajo, se ha empleado para crear una API REST. Lambda, es el servicio de pago por uso que aporta una unidad de cómputo que ejecuta código solo cuando se requiere y que escala de forma automática. En nuestra arquitectura sirve para llamar al endpoint de Sagemaker, un servicio *serverless* orientado a tareas de *machine learning* que ofrece herramientas para construir, desplegar y escalar modelos de inteligencia artificial. En nuestro caso sirve para realizar la inferencia con nuestro modelo de inteligencia artificial, cargándolo desde S3 en una de las instancias que ofrece AWS; la inferencia se realizar a partir de la frase que le envía Lambda, que a su vez recibe de la API Rest.

Finalmente, se ha realizado un análisis de costes que nos indica que durante el primer año de vida el servicio costaría entre 2220.48 US$ y 2553.08 US$ respectivamente para entre 1 y 100 millones de llamadas a la API anuales.

Chapter 1

Introduction

May 2017. Just a month before the academic earthquake in the world of Deep Learning that was about to be caused by the publication of the paper *Attention Is All You Need* [20]. At the age of 15, I uploaded my first mobile application on the Google Play Store. It was a modest application centered around teaching syntax that had aided me in passing my final high school exams. Who would have thought that the application would reach 30,000 downloads when I entered university? Who would have said that this small project would motivate me to create *MiSintaxis* [18], an application with over 70,000 downloads on Google Play? That same year, two events occurred that have led to this final degree project. Firstly, the creation of the transformer architecture by a team at Google, secondly, the creation of *MiSintaxis* app which currently maintains a presence in both Spain and Latin America.

1.1 Study of Spanish grammar

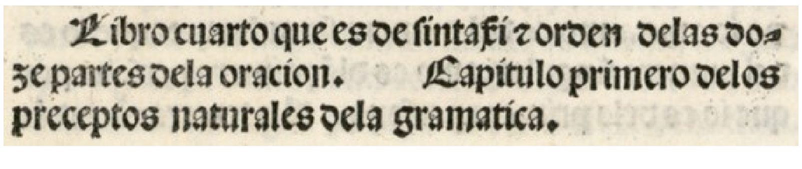

Libro cuarto que es de ſintaſí ⁊ orden delas do-
ze partes dela oracion. Capitulo primero delos
preceptos naturales dela gramatica.

Figure 1.1: Fragment from the Grammar of Elio Antonio de Nebrija (1492). National Library of Spain. *Fourth book, which is about syntax and the order of the twelve parts of speech. First chapter on the natural precepts of grammar.*

> "I wish that, as it is easy to prove the usefulness of Grammar, so would its composition be! But experience shows the opposite, and even without it, one can infer the difficulty it has due to the multitude of opinions and disputes that reign among grammarians. Neither the ancients nor the moderns have been able to agree on many of its main points, nor on the method of writing it."

These words appear in the prologue of the *Grammar of the Castilian Language* published by the Royal Spanish Academy in 1796 in Madrid. Possibly any teacher in charge of explaining this subject in a high school could confirm that even today, it is still very difficult to avoid doubts in the study of Spanish grammar. But this is natural. Language use is a living and complex phenomenon due to characteristics such as ambiguity, and it cannot be definitively fixed in a grammar.

However, the discipline of Spanish linguistics has the merit of being the first to follow the example of the Greek and Latin classics in proposing a systematic study of a Romance language. Elio Antonio de Nebrija does this in a very symbolic year, 1492, with the publication of his Grammar in Salamanca. According to experts [14], the Sevillian grammarian's vision was very modern, since his goals were:

1. To fix the Castilian language as much as possible to ensure its survival.
2. To facilitate the learning of Castilian for foreigners.
3. To use the grammar of Castilian as a first step in the study of the more complex Latin grammar.

There is an extensive list of Spanish grammars that have been published since Nebrija's initial one. In the most recent one by the Royal Spanish Academy, the *New Grammar of the Spanish Language*, published in 2011 [15], we can find a definition of grammar that allows us to situate the development of this work:

> "Grammar studies the structure of words, the ways they are linked, and the meanings that result from these combinations. It includes *Morphology*, which deals with the structure of words, their internal constitution and their variations, and *Syntax*, which is responsible for analyzing how words are combined and arranged linearly, as well as the groups they form."

Of the two parts that make up grammar, this work focuses on Spanish syntax. In the previously mentioned document, we find an extension of the definition of syntax that is included below:

> "*Syntax* is the part of grammar that studies the ways in which words and groups of words are combined, as well as the sentences that these units create. It is, therefore, a discipline of combinatorial or syntagmatic nature. Syntax is also concerned with the study of the various relationships that occur within all syntactic groups, including sentences. Among these links are the various classes of functions that are recognized, but also other dependency relationships. The meaning of all units thus constituted is also the object of study of syntax."

This definition refers to two types of analysis addressed by *Computational linguistics* [9]: constituent analysis and dependency analysis. In the first case, the hierarchical structure of the sentence is identified in the form of a tree in which an intermediate node can have multiple child nodes; the leaves of the tree contain the words of the sentence. This is the classic type of syntactic analysis that we have studied in Spanish language lessons and is the focus of this work. In the second case, there are only binary relationships between words, labeled with the type of the relationship.

1.1.1 Current grammar teaching methodology

Considering that non-university education competencies are transferred to the Autonomous Communities, we will focus in this section on the regulations established in the *Region of Murcia*. Currently, grammar teaching is included within the subject of *Castilian Language and Literature* in the Baccalaureate [2]. This subject is organized into four blocks: Languages and their speakers, Communication, Literary Education, and Reflection on Language. It is this last block that proposes to address the systematic learning of grammar.

More specifically, it is the subject *Castilian Language and Literature II*, in the second year of Baccalaureate, that further develops the study of grammar, including the following basic knowledge:

- Reflection and explanation of the form of words and their relationship with their sentential function.
- Distinction between the form (grammatical category) and function of words (syntactic functions of simple and compound sentences).
- Relationship between semantic structure (verbal meanings and arguments) and syntactic structure (subject, predicate, and complements) of simple and compound sentences depending on communicative purpose.

It is interesting to note that the decree of the Region of Murcia proposes methodological guidelines that include the following two points:

- The new educational reality demands the incorporation of *innovative teaching methodologies* that place students as agents of their own learning. For this, it is essential to implement pedagogical proposals that allow students to *construct meaningful knowledge autonomously*, with initiative and creativity, based on their learnings and experiences.
- The use of *digital technologies*, active and contextualized methodologies that facilitate the participation and involvement of students, the acquisition and use of knowledge in real situations, as well as the *evaluation of their own work* and that of their peers, will be promoted.

If we take a textbook currently used for the second year of Baccalaureate [7], we can find units dedicated to the study of grammar in which, after a presentation of the not always up-to-date theory, a few examples of syntactic analysis and a list of exercises are provided. The exercises often do not include a solution that allows the student to study independently. Moreover, the presentation of the content is extensive, sometimes addressing complex aspects without sufficient examples and not allowing the student to easily establish a connection between a manageable set of theoretical knowledge applied to specific exercises. In other words, the student may soon lose the thread between theory and its practical application. Obviously, experienced teachers can solve many of these problems in the classroom, using the blackboard or auxiliary materials prepared by them. However, independent study and motivation remain a challenge for the student.

An additional problem presented by a classic textbook is the difficulty in adapting the teaching of grammar to the changes proposed by the Royal Spanish Academy. In February 2020, a Glossary of Grammatical Terms [16] was published, which introduces changes in the terms used to define Spanish grammar. For example, it uses the term *syntagma* instead of *group* or *noun* instead of *substantive*.

A tool like the one proposed in this work can overcome the limitations of a methodology based on a classic textbook and can make the guidelines set out in the decree a reality, especially in terms of promoting autonomous learning.

1.1.2 EBAU Test

EBAU[1] is the Universty access tests that Spanish eighteen years old students have to pass in order to enroll into a degree in a Faculty.

In the syntactic analysis test of the EBAU, all types of sentences may appear, and the criteria of the new Spanish grammar must be applied. Additionally, from the year 2024 the new glossary of grammatical terms must be used. Given a sentence, the students do not have to perform a complete syntactic analysis of it. They must specify the type of relationship it presents (for example, subordination) and of what kind it is (for example, relative subordinate). Additionally, they must identify the syntactic function of five units indicated in the sentence (for example, subject, direct object, etc.).

Although a complete analysis does not have to be performed, the students should practise to do so if they want to ensure obtaining the full score of the exercise. Therefore, a digital tool that facilitates the verification of the analysis of any sentence can be of great utility for a student preparing for the EBAU test.

In conclusion, the teaching of grammar remains a challenge due to the complexity and constant evolution of language. Current teaching methodologies aim to promote autonomous learning, the use of digital technologies, and active, contextualized approaches. However, traditional textbooks may not be sufficient for meeting these goals. A digital tool that supports the study of grammar, like the one proposed in this work, can help students to better understand and apply grammatical concepts, ultimately enhancing their learning experience and preparing them for tests such as the EBAU.

[1]Evaluación del Bachillerato para el Acceso a la Universidad.

1.2 Structure of this work

The rest of the document has been structured following the regulations of the Final Degree Project of the Faculty of Computer Science of the University of Murcia. Chapter 2 presents a state of the art of Natural Language Parsing. Since current techniques are based on Artificial Intelligence and more specifically on the use of Neural Networks, this chapter includes a description of the development of this discipline that has led to the spectacular advances that we can observe today. Chapter 3 presents the specific objectives of this work and the methodology used in its development. Chapter 4 presents the resolution of these objectives. Finally, chapter 5 presents the conclusions and future directions of research.

Chapter 2

State of the Art

This chapter presents the state of the art in Natural Language Modeling and Parsing. Section 2.1 summarizes the main characteristics of Neural Networks and their application to the problem of Natural Language Modeling, discussing various architectures that preceded the arrival of transformers. Section 2.2 presents the inner workings of Transformers, the architecture that has been key to the current success of Large Language Models. Section 2.3 explores the Hugging Face ecosystem which provides access to a large number of pre-trained models, thanks to which it has been possible to develop this work. Finally, section 2.4 describes and compares constituency and dependency parsing, introduces the classical CYK parsing algorithm and presents two alternatives to develop constituency parsing, one based on the CYK algorithm, and another one based on transformers. The rest of the project focuses on this second alternative.

2.1 Neural Networks in Language Modeling

Language modeling is a key area of research in Natural Language Processing (NLP) and is essential for many applications. In this subsection, the reader will find an explanation of what language modeling is and its importance in NLP.

Before going into detail using the appropriate mathematical notation, at a high level, language modeling can be described as the task of predicting the probability of a sequence of words in a given language. More specifically, language modeling involves training an statistical model to predict the probability of the next word in a sequence, given the previous words. This is known as the language modeling objective.

Language modeling is important because it enables machines to understand[1] and generate natural language. Language models can be used for a variety of tasks, such as speech recognition, machine translation, text summarization, and chatbot dialog generation. In these tasks, the

[1]The meaning of *understanding* in the field of Artificial Intelligence can be associated with generating a computational representation of the text that *somehow* contains its semantics. An interesting field of study is being developed to try to explain this *somehow*.

language model is used to understand an input and generate an output in natural language text.

There are several types of language models, including n-gram models, recurrent neural network (RNN) models, and transformer models. N-gram models are a classical proposal based on statistical language modeling that use a fixed window of n words to predict the next word in the sequence. RNN models are neural network-based language models that use a sequence of previous words to predict the next word. Transformer models, which are currently state-of-the-art in language modeling, use self-attention mechanisms and more complex neural network architectures to better capture long-range dependencies in the text.

Training a language model involves feeding the model with a large dataset of text in the language it is intended to learn. The model is then trained to predict the probability of the next word in a sequence, given the previous words. The training process involves adjusting the weights of the model to minimize the difference between the predicted and true outputs. Words are represented as vectors in a big-dimensional space, which is convenient to measure the error in the prediction.

In recent years, pre-trained language models such as BERT, GPT-2, and T5 have revolutionized NLP. These models are pre-trained on large datasets and can be fine-tuned on specific tasks to achieve excellent performance. Pre-trained language models have enabled significant progress in a variety of NLP applications, including text classification, question answering, and language generation.

2.1.1 Historical introduction

To understand where we go we need to understand where we came from. NLP systems are not new at all. We can find them in the literature since 1940's. Most of the classical proposals were based on grammar rules and n-grams. It is interesting to mention that the first references of n-grams come from Claude Shannon [4]. In the next section the reader can find a description of classical NLP systems based on n-grams and neural networks. We postpone the description of grammars in the context of NLP to section 2.4.2.

2.1.1.1 N-Grams

N-grams are a type of language model that is commonly used in NLP tasks. An n-gram is a contiguous sequence of n items from a given sample of text. In the context of language modeling, the items are typically words or characters.

N-grams are useful in NLP tasks because they can capture local dependencies between words or characters in a text. By analyzing the frequency and co-occurrence of n-grams in a corpus, we can gain insights into the semantic relationships between words and their contextual meaning.

It is easy to grasp the intuitiveness of n-grams if we think in the development of a text generation application that tries to mimic how we humans write. Suppose the application requests $n-1$ words from the user, and from this input it starts generating the sequence of new words. To continue the sequence, the application could select the word w_n that maximizes the conditional probability $P(w_n|w_{1:n-1})$. But, how is $P(w_n|w_{1:n-1})$ calculated for any valid sequence of n words? Even having a very large corpus, we have to take into account that humans use languages in a very creative way. We could be confronted with sequences of words that do not appear in the corpus.

This is were n-grams can help us. We can approximate the conditional probability $P(w_n|w_{1:n-1})$ using only the last word or the few last words. For example, in the sentence *"Learning syntax is awesome"* the probability of the word *awesome* based on the previous words could be approximated with:

$$P(\textit{awesome} \mid \textit{Learning syntax is}) \approx P(\textit{awesome} \mid \textit{is}) \tag{2.1}$$

When the approximation is done with the last word, we use the term *bigram*. If we use the two last words, the term is *trigram*, etc. If $C(w_{n-1}, w_n)$ represents the count of occurrences of the bigram in the corpus, and $C(w_n)$ the count of occurrences of the word w_n, we can estimate the conditional probability $P(w_n|w_{n-1})$ with Eq. 2.2:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}, w_n)}{\sum_w C(w_{n-1}, w)} = \frac{C(w_{n-1}, w_n)}{C(w_{n-1})} \tag{2.2}$$

An example of a real bigram count from the Berkeley Restaurant Project can be found in table 2.1.

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Table 2.1: Bigram counts for eight of the words (out of V = 1446) in the Berkeley Restaurant Project corpus of 9332 sentences [8].

In general, if N is the size of the n-gram ($N = 2$ for bigrams, $N = 3$ for trigrams, etc.), the approximation of the conditional probability of a word w_n given a history of previous words is:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-N+1:n-1}) \tag{2.3}$$

and the estimation of the conditional probabilities $P(w_n|w_{n-N+1:n-1})$ with the corpus could be done using the following equation:

$$P(w_n|w_{n-N+1:n-1}) = \frac{C(w_{n-N+1:n-1}, w_n)}{C(w_{n-N+1:n-1})} \tag{2.4}$$

Beyond the simple text generation task proposed previously, there are many interesting applications of n-grams. One example is *language identification*. By analyzing the frequency of n-grams in a given text, we can determine the language of the text with a high degree of accuracy. This is because different languages have different distributions of n-grams, which can be used to identify the language of a text.

Sentiment analysis is another NLP task that can benefit from the use of n-grams. In this task, we can identify the n-grams that are most strongly associated with positive or negative sentiment. This can be useful for automatically classifying the sentiment of new texts.

The comparison of the frequency of n-grams in a corpus of texts written by different authors can help us in the *authorship attribution* task. We can identify the n-grams that are most strongly associated with each author.

Finally, n-grams are also used in machine translation, where they can help us in identifying common phrases or expressions in different languages. By aligning the n-grams in the source and target languages, machine translation systems can generate more accurate translations.

In the next subsection we will explain neural networks which are more powerful than n-grams because they can preserve long-term information, share parameters across similar n-grams, and use past memory for predicting words or sentiments. N-grams are simpler and faster than neural networks, but they suffer from data sparsity, computational complexity, and lack of semantic understanding. Therefore, we consider that they are not useful in order to achieve our goal.

2.1.1.2 Neuronal Networks

Neural networks are a type of machine learning algorithm that can learn to recognize patterns in data, and they have been instrumental in achieving state-of-the-art performance in many NLP tasks. In this subsection, we will provide an overview of neural networks and their application in neural language models.

The history of neural networks can be traced back to the perceptron, which was first introduced by Frank Rosenblatt in 1958. It was inspired by the biological neuron, which is a fundamental building block of the nervous system.

A perceptron consists of one or more input nodes, which receive information from the environment, and one output node, which produces the network's output. Each input node is connected to the output node through a weight, which determines the strength of the input's effect on the output.

The perceptron works by taking in input data, multiplying each input by its corresponding weight, and summing these products to produce a single output. This output is then given to an activation function which calculates the final result as shown in Eq 2.5.

$$\hat{y} = f(\mathbf{w} \cdot \mathbf{x}) = f(w_0 + w_1 x_1 + ... + w_m x_m) \tag{2.5}$$

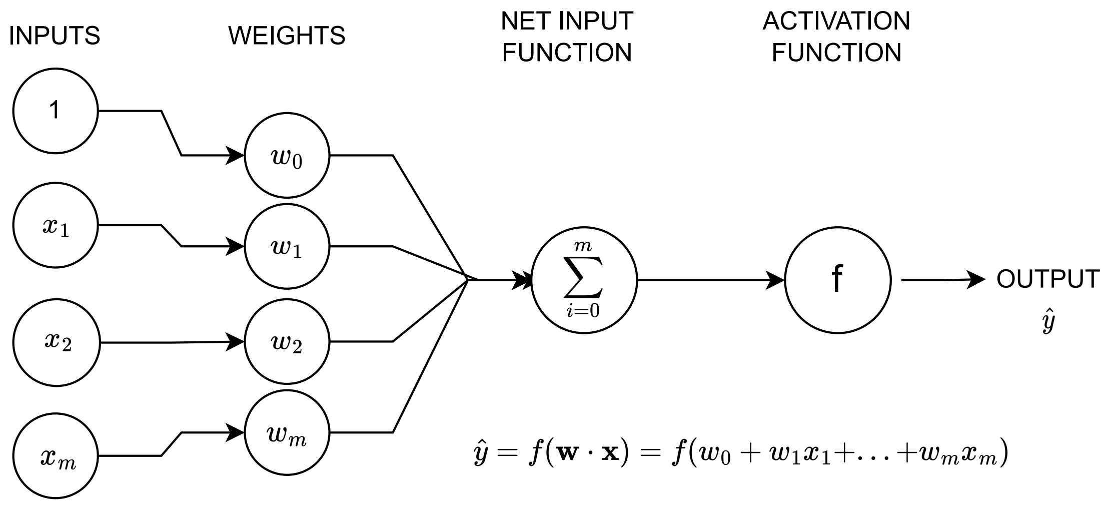


Figure 2.1: Rosenblatts perceptron schema

The perceptron's learning algorithm involves adjusting the weights based on the errors made by the network during training. If the perceptron misclassifies an input, the weights are adjusted to reduce the error. This process is repeated until the network is able to accurately classify the training data. In figure 2.1 we can see the Rosenblatts perceptron schema.

In order to calculate the error, we use the gradient descent algorithm for the perceptron using the mean squared error (MSE) loss function is implemented as follows:

1. Calculate the model output $\hat{y}$.

2. Calculate the derivative of the loss function with respect to the model parameters as shown in Eq. 2.7.

$$MSE(\hat{y}, y) = \frac{1}{N} \sum_{j=1}^{N} (\hat{y}^{(j)} - y^{(j)})^2 \tag{2.6}$$

$$\frac{\partial MSE}{\partial w} = \frac{2}{N} \frac{\partial \hat{y}}{\partial w} (\hat{y} - y) \tag{2.7}$$

   where $\frac{\partial \hat{y}}{\partial w} = x$.

3. Update the parameters as shown in Eq 2.8

$$w \leftarrow w - \eta \frac{\partial MSE}{\partial w} \tag{2.8}$$

   where η is the learning rate.

4. Repeat until convergence.

We will initialize our model with random values for the weights, and we will iteratively update their values in the direction of the negative slope of the loss function. As you can see, there are several hyperparameters in the optimization process that will affect the optimization process. These are:

- Weight initialization
- Chosen value of the learning rate
- Data used for gradient calculation

Perceptrons were limited in their capacity to handle complex tasks because they are only able to produce linear decision boundaries. In other words, they can only separate input data into two classes using a straight line or a plane, depending on the dimensionality of the input.

This limitation means that perceptrons are only able to solve linearly separable problems. For example, a perceptron could accurately classify whether a point on a 2D plane falls above or below a straight line, but it would not be able to classify points that require a more complex decision boundary, such as a circle or a spiral.

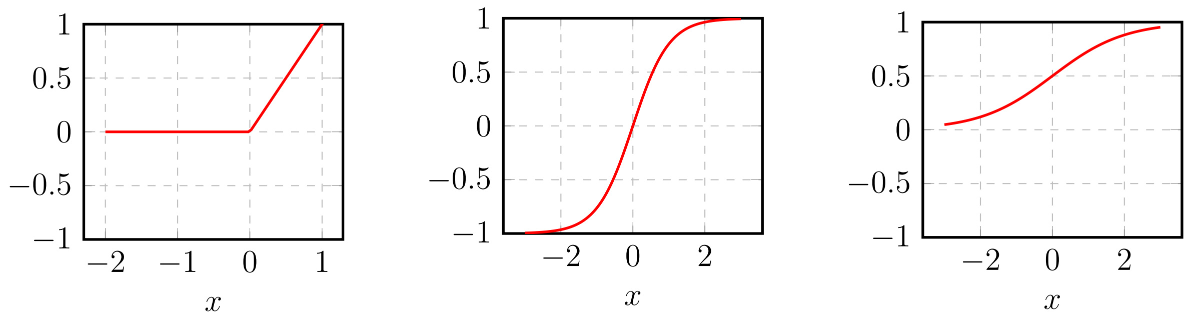
Additionally, perceptrons are only able to learn from labelled data and require a large number of training examples in order to learn to classify input data accurately. This makes them less suited for tasks such as unsupervised learning or learning from small datasets.

Finally, the perceptron's binary output limits its ability to represent complex relationships between inputs. While it can learn to associate inputs with outputs, it cannot capture more nuanced relationships such as partial matches or fuzzy logic. This is why this idea was left behind and others artificial intelligence algorithms were studied.

In the 1980s, researchers took up this idea and began to explore the use of multilayer perceptrons (MLPs) for more complex tasks. MLPs were similar to the perceptron. The multilayer perceptron was developed as a way to overcome its limitations. It is a neural network that consists of multiple layers of neurons, allowing it to learn to classify non-linearly separable patterns by combining multiple linear decision boundaries. This makes the multilayer perceptron more powerful than the perceptron and enabled it to handle more complex tasks. Researchers thought that adding more neurons will fix all problems. Nonetheless they discover that when you add two straight lines together, the result is another straight line. So when you add more neurons to the equation, you are not really improving their performance. This is a limitation because the system is not "learning". Linear functions have a fixed slope and can only represent simple relationships that can be described by a straight line. However, many real-world problems have complex and nonlinear relationships that cannot be accurately modelled using linear functions alone.

In order to overcome this linear regression problem they used non-linear functions also called activation functions.

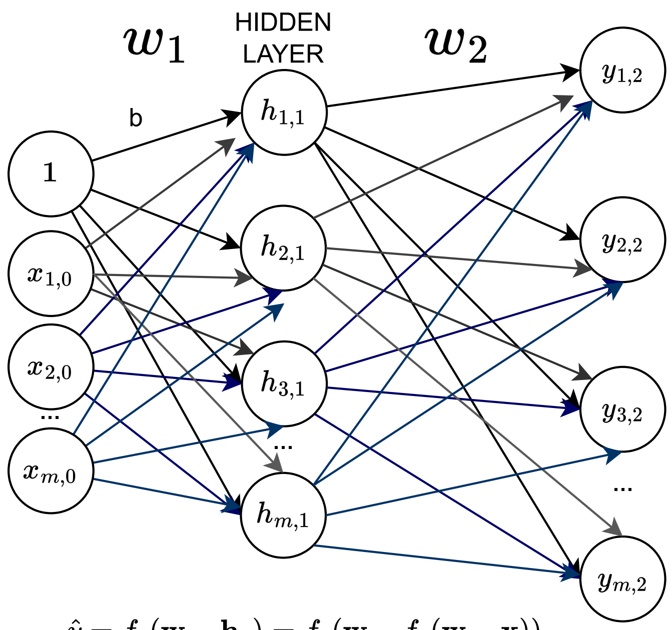
Nonlinear functions can capture more complex and varied relationships, including curves, exponential relationships, and other intricate patterns. There are different activation functions but in figure 2.2 the three most popular non-linear functions are shown. The sigmoid, the tanh and the rectified linear unit or ReLU.



(a) ReLU $= \max(0, x)$ (b) Tanh $= \frac{e^x - e^{-x}}{e^x + e^{-x}}$ (c) Sigmoid $= \frac{1}{1+e^{-x}}$

Figure 2.2: Most relevant activation functions

The multilayer perceptron was a significant advancement in neural network research because it introduced the concept of deep learning. By stacking multiple layers of neurons, the multilayer perceptron could learn to recognize complex patterns in data, leading to a new era of research in artificial intelligence.



$$\hat{y} = f_2(\mathbf{w}_2 \cdot \mathbf{h}_1) = f_2(\mathbf{w}_2 \cdot f_1(\mathbf{w}_1 \cdot \mathbf{x}))$$

Figure 2.3: Graphical example of multilayer perceptron

As we can see in figure 2.3 a MLP is a sequence of Perceptrons, the outputs of which are also the inputs for the next layer (this continues until reaching the output). We can calculate the output of the first layer as shown in Eq 2.9

$$h_1 = f_1(\mathbf{w_1} \cdot \mathbf{x}) \tag{2.9}$$

where h_1 is the hidden state of the first layer. As you can see, the expression is the same as the one we used in the Perceptron model, in which w_1 are the weights of the first layer and x the inputs. Now, if we consider h_1 as the inputs for the next layer, we can find the output as shown in Eq 2.10

$$\hat{y} = f_2(\mathbf{w_2} \cdot \mathbf{h_1}) = f_2(\mathbf{w_2} \cdot f_1(\mathbf{w_1} \cdot \mathbf{x})) \tag{2.10}$$

in which we have simply applied the same expression again. You can intuit that if we have more layers, we simply repeat the expression for each layer, using the outputs of one layer as inputs to the next in a recurrent manner until we reach the output.

For training, we use the MSE, Eq 2.7

We use nonlinear activation functions in the hidden layers of an MLP because a linear combination of linear functions is still a linear function, which would mean that our MLP would not be better than a simple Perceptron. For example, in the case of an MLP with a hidden layer of just one neuron, one input, and one output, the equation would be Eq 2.11

$$\hat{y} = w_2^1 h_1 + w_2^0 = w_2^1(w_1^1 x_1 + w_1^0) + w_2^0 = w^1 x + w^0 \tag{2.11}$$

To find the derivative with respect to the weights of the previous layers, we have to use the backpropagation algorithm, which essentially involves applying the chain rule of the derivative backward in the MLP until we reach the first layer, as shown in Eqs 2.12 2.13

$$\frac{\partial MSE}{\partial w_2} = \frac{\partial MSE}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2} = \frac{2}{N}(\hat{y} - y)h_1 \tag{2.12}$$

$$\frac{\partial MSE}{\partial w_1} = \frac{\partial MSE}{\partial h_1} \frac{\partial h_1}{\partial w_1} = \frac{\partial MSE}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_1} \frac{\partial h_1}{\partial w_1} = \frac{2}{N}(\hat{y} - y)\mathbf{w}_2 f_1'(\mathbf{w_1} \cdot \mathbf{x})x \tag{2.13}$$

And again, if we have more than two layers, we simply repeat the reasoning for each layer until we reach the first one.

The development of the multilayer perceptron was a crucial step in the creation and advancement of neural networks. Despite their success, MLPs still had limitations when it came to processing sequential data.

Neural language models are a type of neural network that can be trained to assign probabilities to word sequences and predict upcoming words. At a high level, the language modeling objective is to predict the probability of the next word in a sequence, given the previous words in the sequence. Neural language models achieve this by processing the previous words in the sequence through a series of neural network layers to generate a probability distribution over the next word.

There are several types of neural language models, including feedforward neural language models, recurrent neural language models, and transformer-based neural language models. Feedforward neural language models process the previous words in the sequence through a series of fully connected layers, while recurrent neural language models based on recurrent neuronal network (RNN) use a recurrent connection to maintain a memory of previous words in the sequence. Transformer-based neural language models, which are currently state-of-the-art in language modeling, use self-attention mechanisms to better capture long-range dependencies in the text. We will deep into RNN and specially into transformers lately in this work.

One of the advantages of neural language models over traditional n-gram models is that they can capture long-term dependencies in the text, as they can use information from all previous words in the sequence. Additionally, neural language models can share parameters across similar n-grams, reducing the number of parameters that need to be learned. Furthermore, they can be used to generate natural language text, a task that was previously difficult for traditional n-gram models.

Training a neural language model involves feeding the model a large dataset of text in the language it is intended to learn. The model is then trained to predict the probability of the next word in a sequence, given the previous words in the sequence. The training process involves adjusting the weights of the model to minimize the difference between the predicted and true outputs.

2.1.2 Training Language Models

Language models are a type of artificial intelligence model that can be trained to understand and generate natural language. These models are a crucial component of many modern NLP applications such as machine translation, text summarization, and chatbots. In this subsection, we will explain the general process of training a language model.

The first step in training a language model is to gather a large dataset of text. This can be any type of text, such as books, articles, or web pages, as long as it is in the language that the model is intended to learn. Once the dataset is collected, it is preprocessed to clean and normalize the text. This includes tasks such as tokenization, where the text is split into individual words or subwords, and normalization of the text to ensure consistency in formatting and spelling.

In order to *understand* words, a computer needs to translate them to numbers. For this purpose, embeddings were created. In NLP, an embedding is a numerical representation of a word or text that captures its meaning and context. Embeddings are created using algorithms that analyze large amounts of language data to identify patterns and relationships between words. By using embeddings, NLP models can better understand the meaning of words and phrases in context, which improves their accuracy and performance.

After preprocessing the text, the next step is to build the model architecture. The most common type of language model used today is the neural network-based language model. Neural networks are a type of machine learning algorithm that can learn to recognize patterns in data. In the case of language models, the neural network is designed to predict the probability of the next word in a sequence, given the previous words in the sequence. This is known as the language modeling objective.

Once the model architecture is built, the next step is to train the model. Training a neural network involves presenting it with examples from the dataset and adjusting the weights of the network to minimize the difference between the predicted outputs and the true outputs.

To achieve this, a loss function is used to measure the difference between the predicted outputs and the true outputs. The choice of loss function depends on the specific task and the type of output. For example, in binary classification tasks, the binary cross-entropy loss function is commonly used, while in multiclass classification tasks, the categorical cross-entropy loss function is often used. In regression tasks, the mean squared error loss function is commonly used. During training, the model adjusts its weights using an optimization algorithm such as stochastic gradient descent to minimize the loss function. The goal is to find the set of weights that results in the lowest possible loss, indicating the best possible fit of the model to the data.

The training process involves feeding the model a sequence of words and asking it to predict the next word. The output of the model is then compared to the true next word in the sequence, and the weights of the network are adjusted using backpropagation to minimize the difference between the predicted and true outputs.

If the model is trained and evaluated on the same dataset, it may perform well on the training set but not generalize well to new data. This is known as overfitting. By splitting the dataset into separate training, validation, and testing sets, we can estimate how well the model will perform on new data and avoid overfitting. Additionally, using a validation set during training allows us to monitor the model's performance and make adjustments to prevent overfitting, which occurs when a statistical model fits exactly against its training data or underfitting, which occurs when it is unable to capture the relationship between the input and output variables accurately. The training set is used to train the model, the validation set is used to tune hyperparameters and make adjustments to the model during training to avoid overfitting or underfitting, and the test set is used to provide an unbiased evaluation of the final model.

The training process can take a long time, especially for large models and datasets. To speed up training, techniques such as parallelization and distributed training are often used. These techniques involve splitting the dataset and model across multiple processors or computers to train them in parallel.

Once the model is trained, it can be used for a variety of tasks, such as text generation or classification. In some cases, the trained model can be fine-tuned on a specific task to improve its performance. This involves training the model on a smaller dataset specific to the task, while keeping the weights learned during the language modeling training fixed.

In conclusion, training a language model involves preprocessing a large dataset of text, building a neural network architecture, and training the model to predict the probability of the next word in a sequence. The training process can take a long time and involves adjusting the weights of the network to minimize the difference between the predicted and true outputs. Once trained, the model can be used for a variety of NLP tasks.

2.1.3 RNN Networks

Recurrent neural networks (RNNs) are a type of neural network that is designed to process sequential data such as text, speech, and time-series data. The key feature of an RNN is its ability to maintain a hidden state that contains information about previous inputs, allowing the network to maintain a memory of the context in which each word appears.

At each time step t, an RNN takes an input vector x_t and a hidden state vector h_{t-1} as inputs and produces an output vector y_t and an updated hidden state vector h_t as outputs. The updated hidden state h_t is calculated as shown in Eq 2.14:

$$h_t = g(Uh_{t-1} + Wx_t) \tag{2.14}$$

W is a weight matrix where $\mathbf{W} \in \mathbb{R}^{d_h \times d_{in}}$. U is another weight matrix where $\mathbf{U} \in \mathbb{R}^{d_h \times d_h}$. The function g is typically a non-linear activation function such as the hyperbolic tangent (tanh) or the rectified linear unit (ReLU).

The output vector y_t is calculated as shown in Eq 2.15

$$y_t = softmax(Vh_t) \tag{2.15}$$

V is a weight matrix where $\mathbf{V} \in \mathbb{R}^{d_{out} \times d_h}$ and $softmax$ is another non-linear activation function that provides a probability distribution over the possible output classes.

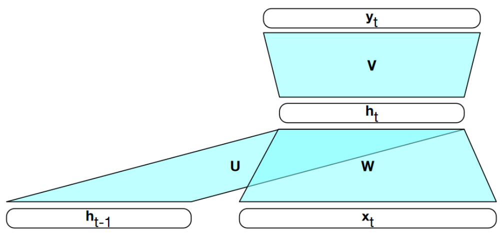


Figure 2.4: Jurafsky Dan, 2023 [9].Simple recurrent neural network illustrated as a feedforward network.

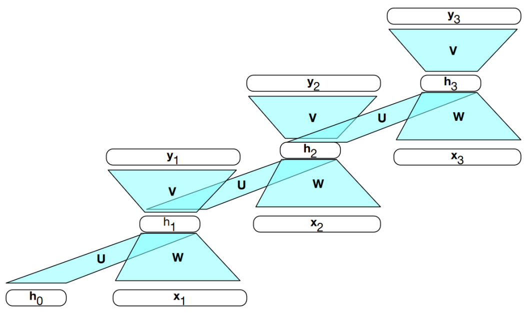


Figure 2.5: Jurafsky Dan, 2023 [9]. A simple recurrent neural network shown unrolled in time. Network layers are recalculated for each time step, while the weights U, V and W are shared across all time steps.

The parameters of an RNN (i.e., the weight matrices and bias vectors) are learned by minimizing a loss function using an optimization algorithm such as stochastic gradient descent (SGD) or Adam.

One of the main advantages of RNNs over other neural network architectures is their ability to capture long-term dependencies in the text. This is because the hidden state of an RNN contains information from all previous time steps, allowing it to maintain a memory of the context in which each word appears.

However, RNNs have some limitations. One major issue is the vanishing gradient problem, which occurs when the gradient of the loss function with respect to the weights of the network becomes very small, making it difficult to update the weights effectively during training. Another issue is the difficulty of capturing long-term dependencies that are more than a few time steps away, as the information can become diluted or lost as it is passed through the network.

In NLP it means that when you work with long sentences the first will affect to less to the inference, in other words, they will be "forgot" when we process the whole input and we will take into account only the last words of the sentence. That´s not correct because we need to take into account the hole context in order to properly understand a sentence.

I will try to illustrate it with an example: "The quick brown fox jumps over the lazy dog, which is lying on the grass in the park, surrounded by trees that are swaying gently in the wind on this beautiful sunny day."

In a neural network trained to predict the next word in a sequence of words, each word in this sentence would be considered as an input to the network, and the network would be trained to predict the next word based on the context provided by the preceding words.

However, during backpropagation, the gradients calculated for the earlier layers of the network can become very small, due to repeated multiplication by small weights in each layer. This means that the earlier layers of the network may not receive enough information to update their weights effectively, resulting in slow convergence or even failure to converge altogether.

In the case of this sentence, the gradients for the earlier layers may become very small due to the length of the sentence and the number of layers in the network, making it difficult for the network to learn meaningful representations of the input. This is an example of the vanishing gradient problem.

Despite these limitations, RNNs have been used successfully in a variety of NLP tasks, including language modeling, machine translation, sentiment analysis, and speech recognition. In particular, the long short-term memory (LSTM) and gated recurrent unit (GRU) architectures have been developed to address some of the limitations of basic RNNs.

2.1.4 LSTM Networks

Long short-term memory (LSTM) networks are a type of RNN that were specifically designed to address the vanishing gradient problem and the difficulty of capturing long-term dependencies. They were introduced by Hochreiter and Schmidhuber in 1997.

As shown in Figure 2.6 LSTMs have a similar structure to basic RNNs, but with additional memory cells and gating mechanisms that allow them to selectively remember or forget information over longer periods of time.

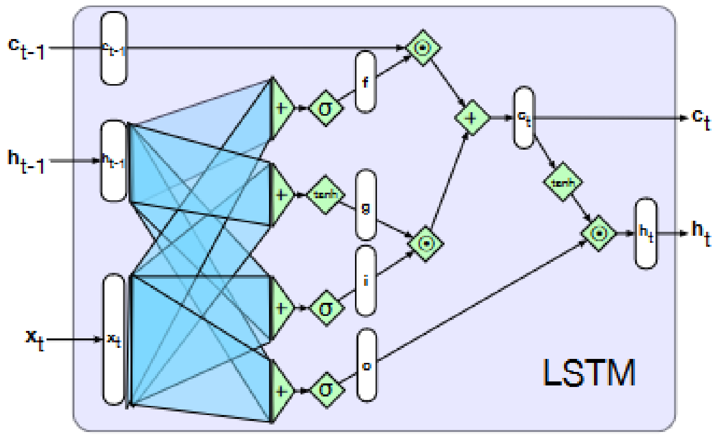


Figure 2.6: Jurafsky Dan, 2023 [9]. A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input x, the previous hidden state $ht - 1$, and the previous context $ct - 1$. The outputs are a new hidden state ht and an updated context ct.

At each time step t, an LSTM takes an input vector x_t and a hidden state vector h_{t-1} as inputs and produces an output vector y_t and an updated hidden state vector h_t as outputs.

The parameters of an LSTM (i.e., the weight matrices and bias vectors) are learned by minimizing a loss function using an optimization algorithm such as stochastic gradient descent (SGD) or Adam [2].

LSTMs have been used successfully in a variety of NLP tasks, including language modeling, machine translation, sentiment analysis, and speech recognition. They have shown improved performance over basic RNNs in tasks that require capturing long-term dependencies.

2.1.5 Encoder-Decoder Model

The Encoder-Decoder model is a neural network architecture commonly used in NLP tasks such as machine translation and text summarization. It consists of two main components: an encoder and a decoder.

[2]Providing a detailed explanation or delving deeply into the mechanics of Long Short-Term Memory (LSTM) falls outside the purview of this particular project, reader can read more about LSTM in chapter 9 of Jurafsky Dan, 2023 [9] book.

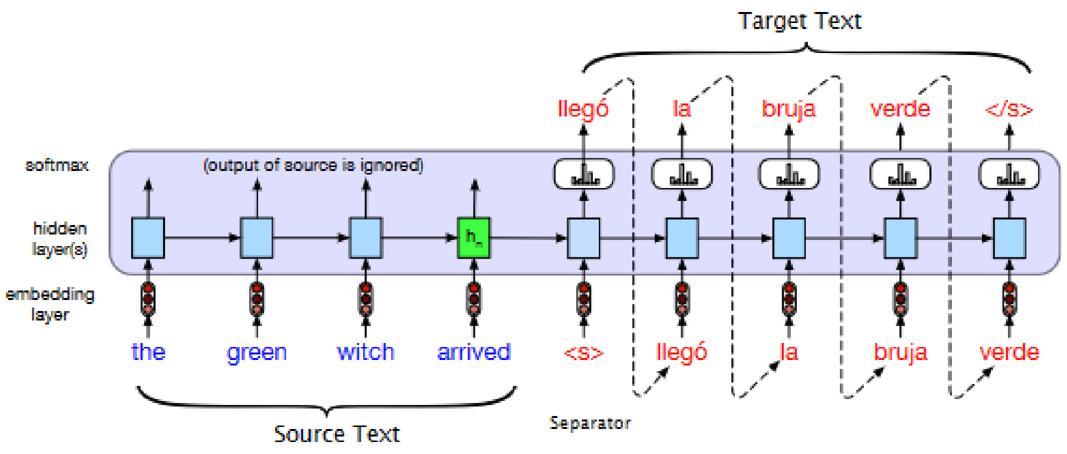


Figure 2.7: Jurafsky Dan, 2023 [9]. Translating a single sentence (inference time) in the basic RNN version of encoder-decoder approach to machine translation. Source and target sentences are concatenated with a separator token in between, and the decoder uses context information from the encoder's last hidden state.

The encoder is responsible for processing the input sequence and encoding it into a fixed-length vector representation. This is typically done using a RNN such as a long short-term memory (LSTM) network or a gated recurrent unit (GRU) network. The encoder RNN takes in the input sequence one token at a time and generates a hidden state for each token. The final hidden state of the encoder RNN, which contains information about the entire input sequence, is used as the input to the decoder.

The decoder is responsible for generating the output sequence based on the encoded input representation. Like the encoder, the decoder is typically implemented using an RNN such as an LSTM or GRU network. The decoder RNN takes in the final hidden state of the encoder as its initial state, and generates an output token one at a time until the end of sequence token is generated. At each time step, the decoder RNN uses the current output token and the previous hidden state to generate the next output token and update its hidden state. The output tokens generated by the decoder are the final output sequence.

The Encoder-Decoder model can be trained end-to-end using backpropagation through time (BPTT), where the objective is to minimize the difference between the predicted output sequence and the true output sequence. This is typically done using a sequence-to-sequence loss function such as cross-entropy loss.

Mathematically, the Encoder-Decoder model can be represented as follows:

Given an input sequence $x = (x_1, x_2, ..., x_n)$ and an output sequence $y = (y_1, y_2, ..., y_m)$, the Encoder-Decoder model computes the probability of the output sequence given the input 2.16 sequence as follows:

$$p(y|x) = \prod_{i=1}^{m} p(y_i|y_{<i}, x) \tag{2.16}$$

where $P(y_i|y_{1:i-1}, x)$ is the probability of the i^{th} output token given the previously generated tokens $y_{1:i-1}$ and the input sequence x.

The Encoder-Decoder model can be trained using maximum likelihood estimation (MLE), where the objective is to maximize the log-likelihood of the true output sequence given the input sequence as shown in Eq 2.17:

$$\log\left(P(y|x)\right) = \sum_{i=1}^{m} \log\left(P(y_i|y_{1:i-1}, x)\right) \tag{2.17}$$

The Encoder-Decoder model has been widely used in NLP tasks and has achieved state-of-the-art performance on machine translation and text summarization tasks. However they have been defeated by Transformers which will be discussed in the next section. RNNs inherently involve sequential computation, which prevents parallelization within sequences. This sequential nature leads to increased training times. In contrast, Transformers, because of the self-attention mechanism, can process data in parallel, leading to faster training times.

2.2 Transformers

Since the article "Attention is all you need" [20], transformers have been everywhere. We have seen them in a wide range of applications, revolutionizing natural language processing (NLP), computer vision, and various other domains. Transformers have emerged as a powerful and versatile architecture for sequence modeling tasks.

The key innovation introduced by the transformer architecture is the self-attention mechanism, which allows the model to weigh the importance of different positions within a sequence when making predictions. This mechanism enables transformers to capture long-range dependencies in data, making them particularly effective for tasks that involve processing sequential or hierarchical information.

In NLP, transformers have achieved remarkable success in tasks such as machine translation, language modeling, sentiment analysis, question answering, and text generation. Models like OpenAI's GPT (Generative Pre-trained Transformer)(which is the base of famous ChatGPT) and Google's BERT (Bidirectional Encoder Representations from Transformers) have set new benchmarks in these areas.

Transformers have also made significant contributions to computer vision tasks. The Vision Transformer (ViT) introduced by Google demonstrated that transformers can be applied effectively to image classification tasks by transforming images into sequences of patches. This approach has challenged the dominance of convolutional neural networks (CNNs) in computer vision and has shown promising results in tasks such as object detection and image segmentation as well.

Beyond NLP and computer vision, transformers have been applied to various other domains, including speech recognition, recommendation systems, reinforcement learning, and even music generation. Their ability to capture complex patterns and dependencies in sequential data has proven valuable across different fields.

The success of transformers can be attributed to their parallelizability, scalability, and the ability to capture both local and global dependencies effectively. However, transformers also come with certain challenges, such as their computational requirements and the need for large amounts of training data. Researchers continue to explore ways to improve and optimize transformer architectures to address these limitations.

Transformers have revolutionized the field of deep learning and have become a cornerstone of modern AI research and applications. Their ability to capture rich contextual information and model complex sequences has unlocked new possibilities in various domains, paving the way for exciting advancements in artificial intelligence.

Overall, transformers have captivated me over the past months, and with "MiSintaxis" [18], they have motivated me to undertake this final degree work. I will attempt to provide a concise summary of their structure, even though it may not do them full justice.

2.2.1 Structure

The transformer model, as proposed by Vaswani et al.[20], consists of an encoder-decoder structure that adopts the self-attention mechanism. A transformer model's architecture is entirely based on attention mechanisms, dispensing with recurrence and convolutions entirely.

As shown in Figure 2.8, the encoder and decoder are composed of stacks of identical layers, each consisting of two primary sub-layers: a self-attention layer and a position-wise fully connected feed-forward network. Depending on the specific model variant, the number of layers can vary, but typical implementations use 6, 12, or even 24 layers.

The **encoder** takes a sequence of continuous representations as input, which are embedded versions of the words in the input sentence. Each word is transformed into a fixed-sized vector through an embedding layer. The encoder processes the input sequence and maps it into a continuous representation that holds the semantic meaning of the input with respect to other words in the sequence. This representation is then passed to the decoder.

The **decoder** generates an output sequence by predicting the next output word based on the given input word and the continuous representation from the encoder. It also consists of self-attention and feed-forward networks, with an additional third sub-layer that performs multi-head attention over the encoder's output. The decoder has an auto-regressive property, meaning it consumes its previous outputs along with the encoder's output to generate predictions for the next word.

A unique aspect of transformers is that they do not inherently understand the order of the sequence, unlike recurrent neural networks (RNNs). To address this, a mechanism called

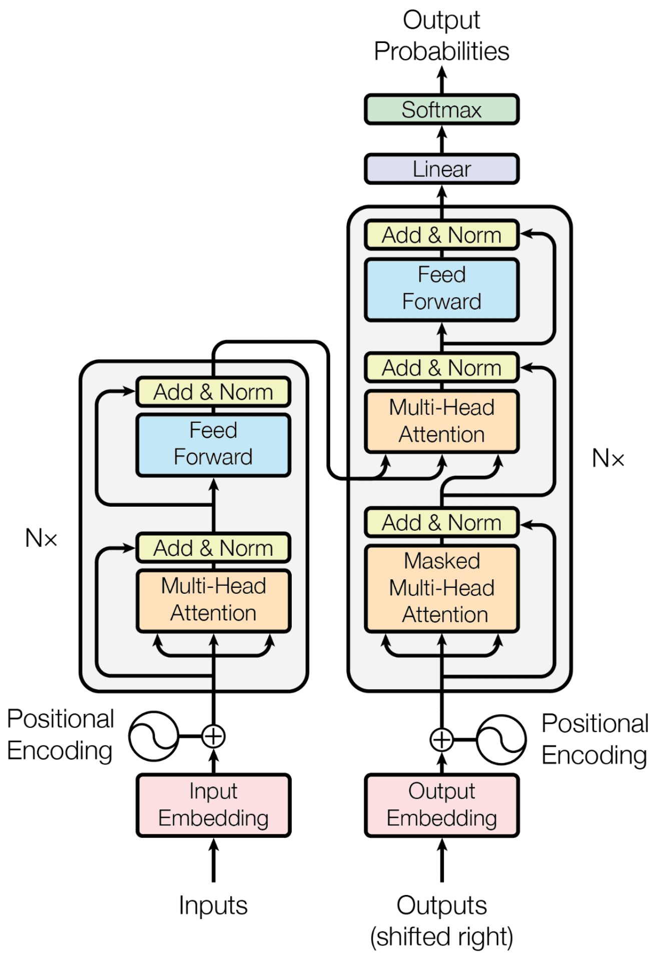


Figure 2.8: Transformer model architecture (Vaswani et al., 2017)

positional encoding is used. Positional encodings are added to the input embeddings at the bottoms of the encoder and decoder stacks, allowing the model to consider the position of the words in the sequence.

The **self-attention mechanism**, also known as scaled dot-product attention, is the most distinctive feature of transformers. It allows the model to weigh the relevance of each word in the sequence for generating an output for a given input word.

In the following sections, we will delve deeper into these key components: the attention mechanism, multi-head attention, and positional encoding, which are instrumental to the functionality and effectiveness of the transformer model.

2.2.2 Attention

The attention mechanism Vaswani et al., 2017 [20]. Is a vital component of the transformer model, which allows it to focus on different parts of the input sequence when producing an output.

This mechanism provides the model with the ability to weigh the importance of each word in the context of the entire sequence, making it highly effective for tasks involving sequential data.

2.2.2.1 Scaled Dot-Product Attention

The transformer model uses a specific type of attention mechanism known as *scaled dot-product attention*. The input to this mechanism is a set of queries (Q), keys (K), and values (V), which are derived from the input data. These sets are all vectors produced in the model's intermediate stages.

The attention score between a query and a key is calculated using their dot product, followed by scaling down by the square root of the query's dimensionality. This score determines the amount of attention to be given to the corresponding value. Formally, the scaled dot-product attention can be expressed as shown in Eq 2.18

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{2.18}$$

Where:

- Q is the matrix of queries.
- K is the matrix of keys.
- V is the matrix of values.
- d_k is the dimensionality of the queries and keys.

The softmax operation ensures that all the attention scores are normalized between 0 and 1, summing up to 1.

To help you understand it consider that a word possesses a set of attributes that define it - this set forms the query vector. A word also has a unique set of attributes, known as the key vector, which helps it identify its semantic relationship with other words. The word utilizes its key to determine its association with other words, much like using a key to unlock a door.

The mechanism through which this process takes place involves the dot product between the key of the word in focus and the keys of all other words in the sequence. The dot product, in this context, serves as an indicator of the degree of relation or similarity between the word in focus and the other words in the sequence. The higher the dot product, the stronger the relation.

2.2.2.2 Mechanism

The attention mechanism operates by associating a weight, or "attention score", with each word in the input sequence. These scores determine the importance of each word in the context of the entire sequence.

For a given query (usually the representation of a word in the sequence), the attention mechanism calculates the attention scores with all keys (representations of all words in the sequence). These scores are then used to create a weighted sum of the values (also the representations of all words), with high scoring words contributing more to the final result.

Intuitively, the mechanism "attends" to all parts of the input, but it gives more consideration to the parts that are more relevant to the current word being processed. In this way, the model can focus on different parts of the input when generating the output, hence the term "attention".

The attention mechanism allows the model to capture long-range dependencies in the data effectively. It ensures that even words far apart in the sequence can directly impact each other's output representations, unlike in traditional recurrent or convolutional neural networks where such interaction is mediated through multiple intermediate computations.

In the next sections, we will elaborate further on how the transformer uses multiple attention heads in parallel and incorporates positional information into the model.

2.2.3 Multi-Head Attention

The attention mechanism explained before, is used in another crucial component of the transformer model, the Multi-Head Attention. It operates by running multiple scaled dot-product attentions in parallel, each with its independent set of learned parameters, allowing the model to capture different types of information from the same input sequence.

2.2.3.1 Mechanism

The process starts by linearly projecting the queries, keys, and values hh times (where hh is the number of attention heads) with different, learned linear projections. Then, each of these projected sets of queries, keys, and values are fed into a different scaled dot-product attention mechanism, yielding hh different output vectors. These output vectors are then concatenated and linearly transformed to produce the final output.

Formally, the multi-head attention can be expressed as shown in Eq 2.19

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W_O \tag{2.19}$$

Where each head is defined as shown in Eq 2.20

$$\text{head}_i = \text{Attention}(QW_{Qi}, KW_{Ki}, VW_{Vi}) \tag{2.20}$$

Here:

- Q, K, and V are the input queries, keys, and values
- W_{Qi}, W_{Ki}, and W_{Vi} are the learned linear projections for the ith head
- W_O is the final linear transformation applied after concatenation

2.2.3.2 Benefits

The advantage of using multiple attention heads is that it allows the model to capture different types of attention or relationships from the input data. For instance, one attention head might focus on syntactic relationships (like subje ct-verb agreement), while another might focus on semantic relationships (like word synonymy).

This ability to learn and apply different types of attention makes transformers particularly powerful and versatile for various tasks, as they can process and understand the input data from multiple perspectives simultaneously.

It's worth noting that the number of attention heads, hh, is a hyperparameter of the model and can be tuned for optimal performance.

In the next section, we will explore another important feature of the transformer model, the positional encoding, which enables the model to make use of the order information in the sequence.

2.2.4 Positional Encoding

Positional Encoding is a crucial part of the transformer architecture. As the model processes all input words in parallel rather than sequentially, it lacks inherent knowledge of the relative or absolute position of words in a sequence. The positional encoding is a method to inject this positional information into the model.

2.2.4.1 Mechanism

In the original transformer model proposed by Vaswani et al. [20], positional encodings are added to the input embeddings. The positional encoding uses sine and cosine functions of different frequencies to encode the position p of a word in the sequence, with dimensions $2i$ and $2i + 1$ in the encoding vector respectively.

The positional encoding for a position p and a dimension i is defined as shown in Eqs 2.21 2.22:

For even $i = 2k$:

$$PE_{(p,2k)} = \sin\left(\frac{p}{10000^{2k/d}}\right) \tag{2.21}$$

And for odd $i = 2k + 1$:

$$PE_{(p,2k+1)} = \cos\left(\frac{p}{10000^{2k/d}}\right) \tag{2.22}$$

Where d is the dimensionality of the input embeddings.

2.2.4.2 Benefits

The use of these sinusoidal functions allows the model to infer the relative positions of words in a sequence. This is due to the specific properties of sine and cosine functions, allowing the model to deduce the position of a word based on the sum or difference of the positional encodings.

Importantly, because the positional encoding is added to the word embeddings, the model can learn to attend to it when it's beneficial. In other words, the model is not forced to use the positional encoding but can learn to do so when it helps improve performance.

As such, the positional encoding plays a vital role in enabling the transformer model to understand the order of words in a sequence, contributing to its effectiveness in handling various sequence-to-sequence tasks.

With the completion of this section, we have discussed the key structural components of transformers - self-attention, multi-head attention, and positional encoding. These elements combined allow transformers to process sequential data in parallel and capture complex, long-range dependencies, making them a powerful tool for various sequence modeling tasks. Future work continues to focus on optimizing and extending these components to develop more efficient and capable transformer-based models.

2.3 Available Models and Hugging Face

As discussed in Tunstall et al. 2022 [19], the Hugging Face ecosystem, comprising its Transformers, Tokenizers, Datasets, and Accelerate libraries, as well as the Hub, provides a comprehensive set of tools for researchers and developers working with transformer models. By offering easy access to a wide variety of pretrained models, efficient tokenization algorithms, a diverse range of datasets, and a platform for sharing and collaboration, it has played a pivotal role in democratizing state-of-the-art NLP. Moreover, the concept of fine-tuning has made it possible to harness these advanced models for specific tasks, significantly improving the efficiency and performance of NLP applications.

This section will delve into the Hugging Face ecosystem, which have had a pivotal role in the project's realization. We will also discuss the concept of fine-tuning. Which was necessary to achieve our goal.

2.3.1 Hugging Face Transformers

Prior to the advent of the Hugging Face Transformers, many research labs released their models using disparate frameworks, typically PyTorch or TensorFlow. This posed a significant challenge for NLP practitioners, who often found it difficult to integrate these models into their applications due to compatibility issues.

The introduction of the Hugging Face Transformers library, however, transformed the landscape by offering a unified API across different architectures. This consolidation fostered an unprecedented surge in transformer-based research and provided NLP practitioners with a straightforward way to incorporate these models into real-world applications.

Today, the Hugging Face Transformers library stands as a comprehensive, user-friendly resource offering thousands of pre-trained models for a plethora of NLP tasks. It houses a multitude of renowned transformer architectures, including but not limited to BERT, GPT, and BLOOM, thereby empowering developers and researchers to leverage state-of-the-art NLP technologies with ease.

The library offers pre-trained models that can be used for tasks such as text classification, named entity recognition, question answering, and language generation, among others. All these models can be fine-tuned on a specific task with a user's own data to achieve state-of-the-art results.

This library has greatly democratized access to transformer-based models by simplifying their use and allowing researchers and developers to build sophisticated applications without having to train these models from scratch, which often requires large amounts of data and computational resources.

2.3.2 Hugging Face Tokenizers

The Hugging Face Tokenizers library is an integral part of the Hugging Face ecosystem. It provides highly optimized implementations of tokenization algorithms, which are the first step in any NLP pipeline.

Tokenization is the process of converting a text into a sequence of tokens, which can be words, subwords, or characters, depending on the algorithm. The tokens are then used as input for a model. The Tokenizers library supports all major tokenization methods used in modern NLP.

This library not only provides efficient tokenization but also handles other important aspects such as encoding the tokens into the format required by the transformer models, and decoding the output of these models back into text.

2.3.3 Hugging Face Datasets

Hugging Face Datasets is a library for loading and preprocessing datasets. It includes a wide variety of datasets for different NLP tasks, making it easier for researchers and developers to access and use these datasets in their projects.

The library also provides powerful tools for preprocessing the data, such as tokenization and batching, which are essential steps when preparing the data for a transformer model. Furthermore, it allows the data to be cached and loaded efficiently, making it particularly suitable for large-scale datasets.

2.3.4 Hugging Face Accelerate

Hugging Face Accelerate is a library that simplifies the use of hardware accelerators like GPUs and TPUs in deep learning projects. It provides a high-level, easy-to-use API for distributing the computation across multiple devices or machines, allowing the models to be trained and used more efficiently.

With Accelerate, developers can easily switch between different hardware setups without changing the model's code. This flexibility makes it easier to scale up the computation when needed, for example, when training large models or processing large datasets.

2.3.5 Hugging Face Hub

The Hugging Face Hub is a platform that hosts pre-trained models and datasets contributed by the community. It allows users to share their models and datasets with others, facilitating collaboration and knowledge sharing in the field of NLP.

The Hub integrates seamlessly with the Hugging Face libraries, allowing users to directly load models and datasets from the Hub into their projects. This integration simplifies the process of using and sharing NLP resources, making the Hugging Face ecosystem even more accessible and user-friendly.

In summary, the Hugging Face ecosystem, with its diverse range of libraries and the Hub, provides a powerful set of tools for working with transformer models. It has been instrumental in enabling us to utilize state-of-the-art models and datasets in our project, and we anticipate that it will continue to drive progress in the field of NLP in the future.

2.3.6 Generative Models available

Hugging Face offers a diverse array of generative models, including the renowned BERT, the GPT family from which the widely recognized ChatGPT originates, and open-source models like Bloom.

For the purposes of our study, we decided to focus on testing with GPT and different variants of Bloom. This choice was primarily driven by the extensive documentation and information available on the Hugging Face Hub, which significantly simplifies the model selection and implementation process.

These models will be discussed in subsequent sections.

2.3.7 Fine-tuning Transformer Models

Fine-tuning is a critical concept when working with pre-trained transformer models, especially in the context of transfer learning. While pre-trained models are trained on large amounts of general-domain data, they can be adapted or "fine-tuned" to perform specific tasks or operate on specific-domain data, achieving high performance with comparatively less data and computational resources.

To fine-tune a transformer model, we initialize the model with the pre-trained weights and then continue training on our specific task's data. This allows the model to leverage the learned representations from the pre-training phase and adjust them to the nuances of the new task.

Fine-tuning can be more efficient than training a model from scratch, particularly when the task-specific data is limited. This method also allows us to leverage the power of state-of-the-art transformer models, such as BERT or GPT, which have been trained on extensive corpora capturing a wide range of linguistic knowledge.

Hugging Face's Transformers library simplifies the fine-tuning process by providing an easy-to-use API for loading pre-trained models and fine-tuning them on custom datasets. With a few lines of code, we can load a model, fine-tune it on our data, and evaluate its performance.

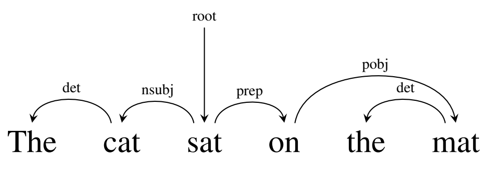
In this project, we leveraged fine-tuning to adapt a pre-trained transformer model to our specific task, enabling us to achieve high-quality results despite the limitations in data and computational resources. We found the Hugging Face ecosystem's support for fine-tuning instrumental in achieving this goal.

2.4 Parsing

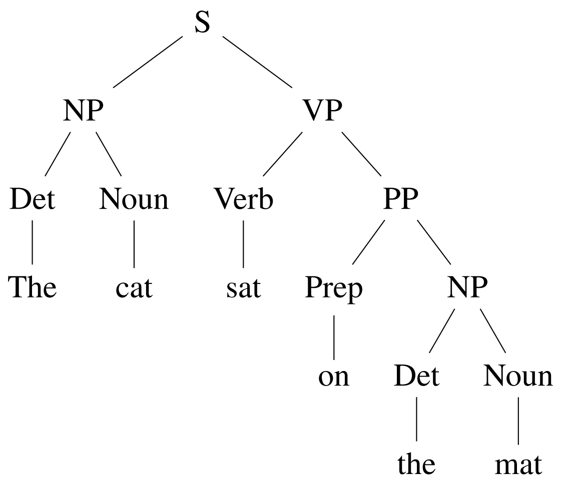
Parsing is the task of performing Syntactic Analysis and generating a parse tree from a given sentence. This tree serves to highlight the syntactical structure of a sentence, based on formal rules in grammar. Within the realm of NLP, there exist two fundamental approaches to Parsing, namely Dependency Parsing and Constituency Parsing.

2.4.1 Constituency parsing vs. dependency parsing

Dependency Parsing refers to the process of extracting a grammatical structure that provides clear definitions of the relationships between "head" words and all other words in a sentence. This can be thought of as a graph, where the words in the sentence serve as nodes and the dependencies between them serve as edges. Below 2.4.1 is a graphical representation of the dependency analysis for the sentence "The cat sat on the mat", where each word is connected with arrows representing dependencies.



In this structure, there exists a defined root word, and all other words can be reached by traversing the graph from this root word, as it serves as the starting point for the text data. The Stanza NLP model utilizes this approach to Dependency Parsing, and generates a structure that resembles the one depicted in figure 2.4.1.



Constituency Parsing is a parsing technique based on context-free grammar. It deals with identifying the types of phrases in a given text data. This approach breaks down the text into sub-phrases, known as constituents, based on a grammar category. These constituents essentially serve as their own grammatical units.

In Spanish, some examples of phrase units of grammar categories include Sujeto (S), Predicado Verbal (PV) or Complemento Directo (CD). By utilizing Constituency Parsing, we can identify the various constituents within a sentence, and understand how they relate to one another in terms of the grammatical structure of the text data.

Constituency parsing is often used in NLP to help machines understand the structure of a sentence and its meaning.

Language modeling and constituency parsing are both important tasks in NLP and are often used in conjunction with each other. Language models can be used to help improve constituency parsing accuracy by providing information about the likelihood of certain words or phrases occurring in a particular context.

In particular, language models can be used to help guide the parsing process by providing a score for each possible parse tree. The score represents the likelihood of the parse tree being the correct structure for the sentence, given the language model's understanding of the language. The parser can then use this score to guide its search for the most likely parse tree.

Similarly, constituency parsing can be used to improve language modeling by providing more accurate information about the grammatical structure of the language. This can help language models better understand the relationships between words and phrases in a sentence, leading to more accurate predictions.

Overall, the combination of language modeling and constituency parsing can help machines better understand the structure and meaning of natural language text. By breaking down sentences into their constituent parts and analysing their grammatical structure, machines can gain a deeper understanding of language and better perform a variety of NLP tasks.

2.4.2 Cocke-Yourgen-Kasami algorithm

Although this work focuses on the use of large neural network language models for the analysis of Spanish constituents, other techniques have traditionally been used for this purpose, the most representative of which is the Cocke-Kasami-Younger (CYK) algorithm. This is a dynamic programming algorithm based on the use of a context-free grammar to model natural language, put into Chomsky Normal Form (CNF). In this case, the production rules of the grammar are of the form $A \to BC \mid w$, being A, B and C non-terminals of the grammar and being w a string of constituents (language tokens are the terminal symbols of the grammar). Any context-free grammar can be transformed to CNF.

With a CNF grammar, each non-terminal node in the syntactic tree of a sentence will have exactly two children. Because of this, we can use a two-dimensional table to encode the structure of the syntax tree. The cell $[i, j]$ of the matrix contains the set of non-terminals that can derive all the constituents of the sentence going from position i to position j in left-to-right order[3]. The term *span* is used to refer to the portion of consecutive tokens occupying positions i through j in the input phrase. Obviously, the matrix used is of upper triangular form, since it is meaningless to use the entries $[i, j]$ where $i >= j$.

The algorithm proceeds in the order of the columns of the table, filling in the entries from left to right and from bottom to top, i.e. from the first constituent and in increasing span. In column j it starts by filling in the entry $[j-1, j]$, then $[j-2, j]$, until it reaches $[0, j]$ which represents the analysis from the beginning of the sentence to the constituent at position j. It is beyond the scope of this document to describe the algorithm in detail. The reader can find a good description of it in chapter 17 of Jurafski's excellent book [8].

If the input is n long and the algorithm ends by including in cell $[0, n]$ the initial symbol of the grammar, then it has been verified that the input string can be generated by the grammar. With the use of appropriate data structures to link depending cells, the string of derivations needed to validate the input can be retrieved from the table. The algorithm has an order of complexity $O(n^3)$ in time and $O(n^2)$ in space.

2.4.3 Parsing based on language models

In a natural language described with an ambiguous grammar, the number of derivations for a given string can be very large. The CYK algorithm cannot determine which of the derivations is the best. This section explains how to use large neural network-based language models to determine the best alternatives for parsing a sentence. The first option is to combine the CYK algorithm with a neural model that allows to compute a score of the assignment of a syntactic label to a given span of the sentence. The second option is to rely on a large model of the language tuned to generate, from a sentence prompt, an output containing the constituent parsing of that sentence.

[3]There are languages such as Arabic or Hebrew that use the opposite order, but it is possible to adapt the algorithm to deal with them.

2.4.3.1 Parsing based on scope labeling

The first alternative has been proposed by Kitaev [10]. The fundamental idea is to combine the CYK parsing with a classifier that scores the assignment of label l (non terminal of the grammar) to the span from i to j. This score can be represented with $s(i, j, l)$. Figure 2.9 represents the architecture of the system.

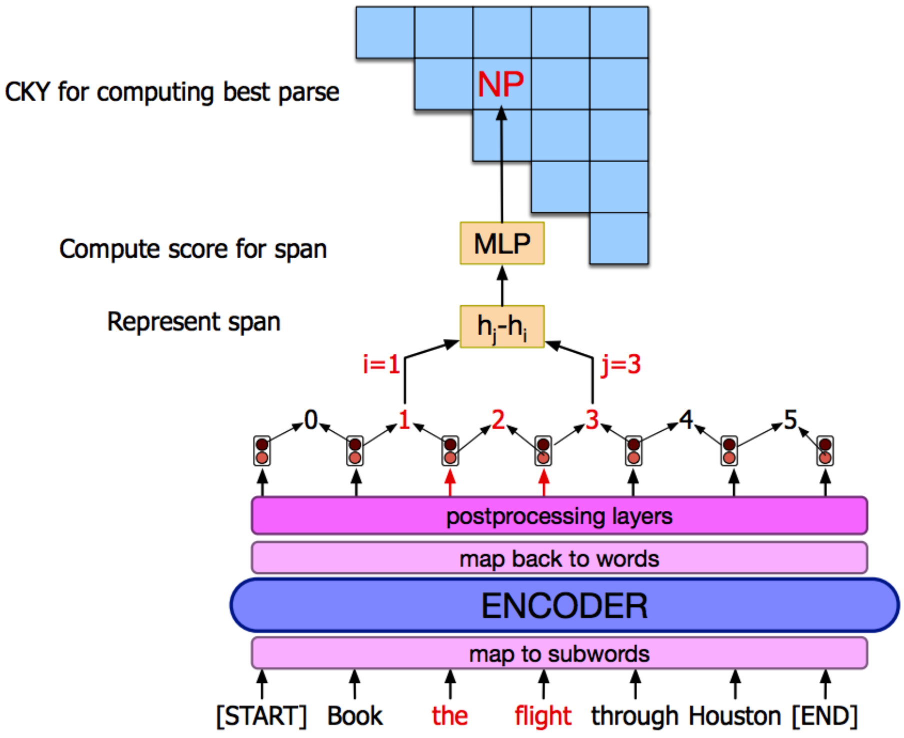


Figure 2.9: Jurafsky Dan, 2023 [9]. Architecture of constituency scope labeling based on CYK and a neural network clasifier.

The classifier receives the complete sentence, creates a representation of the spans with the embeddings of the initial and final words, and generates the scores of all the possible label assignments to each span $\forall l\ s(i, j, l)$. Then, with each parsing tree T of the sentence generated with the CYK algorithm, it computes a score for the whole tree $s(T)$:

$$s(T) = \sum_{(i,j,l)\in T} s(i, j, l)$$

Finally, the tree with the maximum score is selected. The role of the grammar is to reduce the search space, constraining possible combinations of constituents. Nevertheless, a good classifier would assign a very poor score to rare combinations. Taking this into account, some alternatives have been used that skip the parsing of CYK and simply use the scores in a greedy search to choose the highest scoring label for each span.

2.4.3.2 Parsing based on Seq2Seq

The approach of using only a language model for constituent analysis without the support of a grammar achieves an important result in Vinyals' work [21], which opens a very interesting line of research in which this problem is treated as a *sequence to sequence* (Seq2Seq) problem that can leverage the use of the attention mechanism.

In the Seq2Seq approach, the model is trained to work as a translator that converts an input sequence into another output sequence. Starting from a large model of the language, a fine-tuning is performed to re-train the model in a specific translation task with a sufficiently large corpus of examples which, in our case, are constituent analysis of Spanish sentences. The hypothesis of this approach is that the model learns the grammar of the language from the training dataset. Chapter 4 is devoted to the development of this approach.

2.4.3.3 Recent studies

Currently, all the top-performing parsing techniques employ neural network frameworks and main research primarily utilized The English Penn Treebank (PTB) corpus [11], a significant corpus in English language research.

Recent studies have utilized Bi-Directional LSTM in their research, achieving significant advancements in the field like shown in the work of Cross & Huang, 2016 [5]. A notable improvement has been observed when the LSTM encoder was replaced with a self-attentive architecture, allowing the model to capture global context without using recurrent connections Kitaev & Klein (2018) [10]. This self-attentive architecture, introduced by Vaswani et al. (2017) [20] relies on a repeated neural attention mechanism and has been found to outperform LSTM-based approaches. This novel encoder-decoder design provides a means to incrementally build up a labelled parse tree, thereby improving the effectiveness of syntactic analysis. Indeed, in the work of Mrini et al. (2020) [13], they point out that while attention mechanisms have enhanced performance in NLP tasks, and have retained explainability in models, self-attention is currently widespread, but its interpretability remains a challenge due to the abundance of attention distributions. They proposed a new self-attention variant, known as the Label Attention Layer, where attention heads symbolize labels. Interestingly, their model demands fewer self-attention layers compared to existing works, and they found that the Label Attention heads have learned relations between syntactic categories, thereby revealing pathways to scrutinize errors. Using this mechanism, they improve the parsing performance over the English and Chinese Penn Treebanks. Another approach is the already referenced Vinyals' work [21].

Turning to Spanish, a noteworthy development comes from the doctoral thesis of Luis Chiruzzo [3]. In his work, Chiruzzo develops a statistical Head-Driven Phrase Structure Grammar (HPSG) parser specifically for Spanish. HPSG, as a deep linguistic formalism, elegantly combines syntactic and semantic information, allowing it to model many linguistic phenomena. In HPSG, every component of a sentence, ranging from the sentence itself down to the individual words, is represented as a complex feature structure known as a *sign*. These signs are collections of attributes (features) and their corresponding values, which may be atomic (like strings or symbols), or complex (other signs or sets/lists of signs).

The *Head-Driven* part of the name indicates the central role of the head (the core element) in determining the properties of the phrase (group of words). The head of a phrase passes its properties up to the entire phrase, and in turn imposes constraints on what other elements may or must be included in the phrase.

HPSG grammar is particularly useful because it unifies syntactic and semantic information, making it possible to model a wide range of linguistic phenomena in a consistent and straightforward way. This formalism is particularly powerful in handling phenomena such as word order variation, unbounded dependencies, and complex predicate phrases, among others.

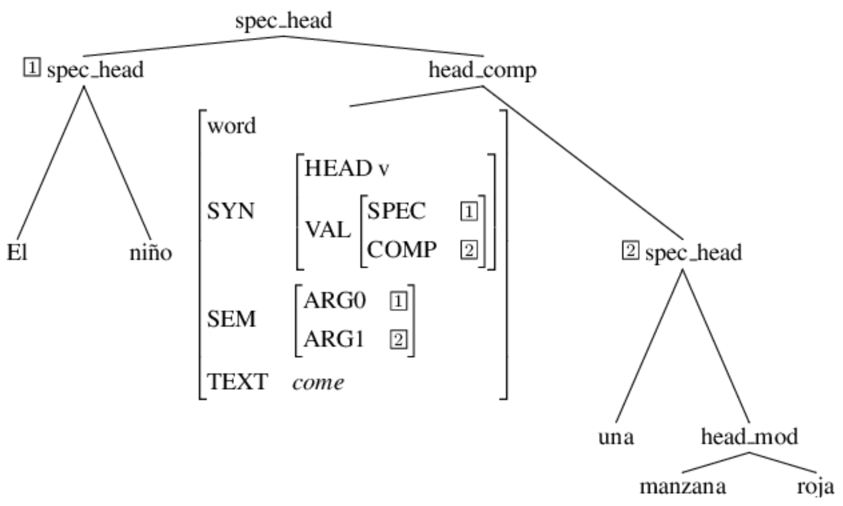


Figure 2.10: Chiruzzo 2020 [3]. Simplified tree for “El niño come una manzana roja” “The boy eats a red apple”. Only the node for “come” is expanded, but each of the nodes in the tree is a feature structure that contains the corresponding combinatorial and semantic information.

Chiruzzo’s research encompasses multiple stages: the design of the HPSG grammar, corpus construction, implementation of parsing algorithms, and evaluation of parser performance. A simplified yet powerful HPSG grammar for Spanish was created, which models morphosyntactic information of words, syntactic combinatorial valence, and semantic argument structures in its lexical entries. This grammar uses thirteen broad rules for managing various linguistic structures, such as specifiers, complements, modifiers, clitics, relative clauses, punctuation symbols, and coordinations. The relative clause that modifies a noun phrase represents the only long-range dependency modeled in this standard HPSG simplification. Semantic role labeling is used as the semantic representation.

The Spanish AnCora corpus was transformed using a semi-automatic process and analyzed using the grammar implementation, resulting in a Spanish HPSG corpus of 517,237 words in 17,328 sentences. The implemented statistical parsing algorithms, including a bottom-up baseline using bi-lexical comparisons or a multilayer perceptron, a CKY approach using the results of a supertagger, and a top-down approach encoding word sequences with an LSTM network, were trained over this corpus.

Evaluations indicated that the LSTM top-down approach is the best performing parser over the test data, achieving the highest scores according to constituency metrics, dependency metrics, and SRL. However, the comparison against external parsers might be skewed due to the required post-processing needed to adapt them to the research format. The LSTM top-down parser also excelled in identifying specific language phenomena, surpassing the baseline models in most metrics.

Chiruzzo's work represents a significant advancement in Spanish language parsing, utilizing a combination of deep linguistic formalism and state-of-the-art machine learning techniques. His findings provide a strong foundation for future research, highlighting the value of innovative grammar designs and LSTM top-down approaches for language parsing.

Chapter 3

Analysis of goals and Methodology

Having dealt with the state of the art, in this chapter we analyse the goals of the project, establishing the specific elements that will allow us to achieve these goals, as well as the development methodology.

3.1 Goals

The principal goals of this research are dual-faceted. The first aim is to create an artificial intelligence mechanism capable of performing constituency parsing on Spanish sentences, aligning with the labelling system outlined in the *Nueva gramatica BASICA de la lengua española* [15]. Our second ambition is to ensure widespread accessibility of this system. To this end, we leverage the Amazon Web Services platform for hosting and distribution purposes. The successful completion of this research will have implications for the MiSintaxis application, enabling its thousands of global users to benefit from quality education. As I dreamed about when I was 15 years old.

3.1.1 Syntactic Analysis of Spanish using Seq2Seq

In the scope of this research, my objective is to augment and refine preceding studies, notably the work of Chiruzzo discussed in 2.4.3.3, through the implementation of significant alterations to the current methodology. My primary focus will be on substituting the LSTM with Attention-based models, following the approach of Kitaev et al. (2018) [10] and applying sequence to sequence like in Vinyals' work [21]. Attention-based models, like self-attention, offer a promising alternative to LSTM by summarizing large global contexts without the necessity of recurrent connections. This may lead to improved efficiency and performance in parsing tasks.

However, the crux of this research does not solely rest on the utilization of Attention-based models. I intend to further this approach by harnessing the power of large language models

(LLMs), which have exhibited remarkable capabilities in various natural language processing tasks. Some of the state-of-the-art LLMs, such as ChatGPT, have shown that they can generate human-like text, making them a promising tool for improving the efficiency and accuracy of constituency parsing. By fine-tuning these LLMs in seq2seq task to handle constituency parsing, I hope to generate results that supersede the current state-of-the-art methods.

Another pivotal component of my work involves making enhancements to the corpus used for parsing. Currently, the Spanish AnCora corpus, utilized by Chiruzzo, is widely used in parsing research. However, I propose to modify this corpus to align it with the labelling system detailed in the *Nueva gramatica BASICA de la lengua española*. This comprehensive guide provides a initial framework for understanding and analysing Spanish grammar, and aligning our corpus with its principles may further improve the accuracy of our parsing methods.

Through these methodological advancements, my objectives extend beyond making significant contributions to the field of Spanish language parsing and pushing the boundaries of current possibilities. I also aim to democratize access to this cutting-edge technology through the development of our application, MiSintaxis [18]. This software harnesses the power of our improved parsing techniques, bringing high-level language analysis to users' fingertips.

The democratization process is not only about creating powerful and accurate language models, but also about ensuring they are scalable, maintainable, and accessible. Therefore, in addition to refining language models, this work also entails incorporating them into a user-friendly application and efficiently deploying this application in a way that can handle potentially high usage loads. For this task, I have opted to leverage Amazon Web Services (AWS), a robust and scalable cloud computing platform. In the following subsection, I will delve into how AWS was used to integrate and deploy the MiSintaxis application, ensuring its performance, reliability, and accessibility to users worldwide.

3.1.2 Integration Analysis in AWS

Amazon Web Services (AWS) provides a vast assortment of services, creating an extensive range of possibilities for developing and deploying cloud-based applications. After the process of training various models, my primary objective was to identify the most fitting architecture that would allow me to establish a microservices-based API. The architecture I sought had to be scalable, to accommodate fluctuating usage loads, and affordable, to ensure sustainability and cost-effectiveness.

Microservices architecture offers several advantages, making it a suitable choice for this task. It allows for the development of applications as a collection of loosely coupled services, which can be developed, deployed, and scaled independently. This offers enhanced scalability, as each service can be scaled individually based on its own demand, leading to cost-effective use of resources.

Selecting the right architecture required an understanding of the nature of our application and the potential usage patterns. In addition, it involved evaluating the various AWS services in terms of their capabilities, costs, and how well they fit into the proposed architecture. This careful evaluation led to an architecture that not only meets our current requirements, but also has the flexibility to evolve with future needs. In the section 4.3, I will describe the process of choosing the architecture, and the specifics of its implementation in AWS.

3.2 Methodology

In this section, I will elaborate on the methodologies and practices that have been integral to the smooth running and successful completion of this project. As a student researcher in the subject Compilers since my third year of undergraduate studies, I have developed a keen understanding and appreciation of systematic work approaches.

The inspiration to adopt Agile methodology for this project came from my reading of the book *Lean Startup* [17]. This seminal work emphasized the importance of being responsive, flexible, and customer-focused in project management - values that aligned perfectly with the principles of Agile. Agile methodology proved to be a linchpin in this project, shaping the way we executed and managed our tasks. Known for its iterative approach and its focus on customer satisfaction, Agile was instrumental in fostering an efficient and flexible work environment. It placed significant emphasis on adaptive planning, evolutionary development, prompt delivery, and continual improvement, allowing for the accommodation of change in a smooth and efficient manner.

One of the pivotal components of our Agile implementation was the continual collaboration with linguists. This feedback loop was integral to the success of our project, as it allowed us to obtain immediate and critical insight into the performance of our models. Linguists played the role of the "customer" in our Agile approach, as they were able to provide expert assessment of the syntactic and semantic inferences made by the models.

Every iteration of model training was followed by a thorough review of its performance. Based on the feedback received, I made necessary adjustments to the model configurations, training methodologies, or even to the corpus itself. This process of continual improvement was underpinned by the Agile methodology and was crucial in achieving our project goals.

In essence, the Agile methodology facilitated a symbiotic relationship between development and linguistic expertise, resulting in a project that was not only trying to be technologically sound but also linguistically accurate.

I primarily used Notion, a unified workspace where all the tasks, schedules, and deadlines were meticulously documented. Notion allowed me to visualize my workflow, the work to be done, and the timelines.

Building on the Agile methodology, I adopted a Git workflow, akin to our practices in MiSintaxis [18], providing a robust and efficient framework for managing the codebase. Git, with its distributed version control system, facilitated simultaneous work on different features without conflicts. While this is typically beneficial for team projects, in this case - as the sole developer

- it served as a tool for code versioning and documentation, enabling me to track and revert to earlier iterations when necessary.

Additionally, the flexibility of Git supported my working dynamics, allowing seamless transition between coding on my personal computer and the high-performance machine used for model training. To maintain code quality and minimize bugs, all changes to the code were meticulously reviewed before being merged via pull requests, ensuring that each update to the codebase adhered to the highest standards of quality and consistency. This strict review process also served as a checkpoint to reassess the alignment of code updates with project objectives.

In addition to Agile and Git workflow practices, regular meetings with my co-advisor, Eduardo, played a pivotal role in the successful progression of this project. Scheduled on a weekly basis, these meetings provided a consistent platform for dialogue and exchange of ideas, enhancing the overall development process.

Each meeting functioned as a condensed sprint planning session, often extending over a span of five hours. During these intensive discussions, we evaluated the progress made, navigated through any roadblocks, and strategized for the week ahead. This systematic evaluation and forward planning not only ensured sustained momentum, but also facilitated the effective tracking of the project. Our close collaboration expedited the resolution of problems and fostered the continuous enhancement of the project, playing a pivotal role in the ultimate success of our efforts.

In conclusion, these methodologies and practices allowed me to work efficiently, manage time effectively, ensure the quality of the work, and adapt to changes quickly and effectively, thereby facilitating the successful completion of the project.

Chapter 4

Design and Resolution

In this chapter, we will elaborate on the process of implementing the artificial intelligence model for constituency parsing. This journey involves multiple steps such as model training, evaluation, and finally, deployment on the Amazon Web Services (AWS) platform.

In the section 4.1 on *Constituency Parsing*, we will provide an overview of the methodological foundation of constituency parsing and its significance in the context of our project. We will then discuss the AnCora corpus in detail, emphasizing its role as the primary training data resource for our model. Subsequently, we will illustrate the necessary modifications undertaken to adapt the AnCora corpus to fit the requirements of the *Nueva gramatica BASICA de la lengua española* [15] which is used in our app MiSintaxis [18].

Further, in the *Fine-tuning of Hugging Face models* subsection 4.1.3, we will explore the process of tailoring pre-trained Hugging Face models to suit the specific demands of our task. This includes the alterations made to enhance the performance of these models. The subsequent subsection, *Inference with the generated model* 4.1.4, will elucidate how the fine-tuned model is leveraged to generate predictions for novel inputs.

In the *Evaluation* section 4.2, we will outline the strategies employed to assess the performance of the model, primarily focusing on metrics like recall, precision, and the F_1 score. This evaluation extends to an analysis of the efficiency and effectiveness of our models from various perspectives, such as RAM and CPU usage, model file size, average processing time per sentence, and overall F_1 score. We will delve into an examination of both successful and erroneous predictions, thereby offering insights into the model's strengths and limitations.

Lastly, the *Study of integration in AWS* section 4.3 will depict the integration of our model into the AWS environment. We will explore different AWS services such as Lambda functions, API Gateway, S3 Bucket, EC2, ECR Container, and Sagemaker, delineating their respective roles in model deployment and management. Additionally, we will conduct a cost analysis to comprehend the financial considerations involved in utilizing these services.

In essence, this chapter encapsulates the entire implementation process of our constituency parsing AI model, from its inception and development to evaluation and deployment in AWS.

4.1 Constituency parsing

This section describes the whole process of training a language model for the task of parsing Spanish sentences. First, the AnCora corpus, used as input data for training, is described in section 4.1.1. As usual in a computational learning task, the data cannot be used as it is available, but has to be adapted to the parsing format used in the study of Spanish grammar in high schools. This adaptation is the subject of section 4.1.2. The fine-tuning of Hugging Face models with the corpus prepared for constituent analysis is described in section 4.1.3. Finally, in section 4.1.4 we show how to perform inference with the trained model.

4.1.1 Description of the AnCora corpus

The AnCora-ES corpus [12] is used as input data to prepare the language model for the constituent analysis. The corpus contains approximately 500,000 words in 17,300 sentences annotated using an XML format[1]. Most of the corpus comes from press articles, including feeds from Spanish EFE news agency and the 'El Periódico' newspaper. There are multiple levels of annotation, including morphological, syntactic and semantic information, as well as entity identification and co-reference between constituents. This amount of information generates a really verbose content, as can be seen in the following partial example.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<article lng="es">
  <sentence>
    <sn arg="arg1" entity="entity1" entityref="nne" func="suj" homophoricDD="yes" tem="tem">
      <spec gen="f" num="s">
        <d gen="f" lem="el" num="s" pos="da0fs0" postype="article" wd="La"/>
      </spec>
      <grup.nom gen="f" num="s">
        <n gen="f" lem="final" num="s" pos="ncfs000" postype="common" sense="16:05550251"
            wd="final"/>
        <sp>
          <prep>
            <s lem="de" pos="sps00" postype="preposition" wd="de"/>
          </prep>
          <sn>
            <grup.nom gen="f" num="s">
              <n gen="f" lem="copa" num="s" pos="ncfs000" postype="common" sense="16:02533651"
                wd="copa"/>
            </grup.nom>
          </sn>
        </sp>
        <sp>
          <prep>
            <s lem="entre" pos="sps00" postype="preposition" wd="entre"/>
          </prep>
          <sn complex="yes">
            <grup.nom coord="yes">
              <grup.nom entity="entity2">
                <n lem="Inglaterra" ne="organization" pos="np00000" postype="proper"
                    wd="Inglaterra"/>
              </grup.nom>
              <conj conjunctiontype="coordinating">
                <c lem="y" pos="cc" postype="coordinating" wd="y"/>
```

[1]The corpus can be downloaded from `http://clic.ub.edu/corpus/es/ancora`.

```
            </conj>
            <grup.nom>
              <n lem="Alemania" ne="organization" pos="np00000" postype="proper"
                  wd="Alemania"/>
            </grup.nom>
          </grup.nom>
        </sn>
      </sp>
    </grup.nom>
  </sn>
```

The example shows the beginning of one of the 1635 XML files in the corpus. Each file corresponds to a newspaper article. We have focused our attention on some of the elements present in the AnCora XML format. These elements are the tag identifiers and some of the attributes associated with the tags. All tags representing leaf nodes in the parse tree contain the attribute `wd` (word) whose value is the word in that leaf node. Some other attributes contain syntactic information, such as `func` (function) and `tem` (theme). Table 4.1 lists the identifiers of XML elements in the AnCora files.

Table 4.1: Tags of the AnCora corpus.

Tag	**Description**
article	Root node of an xml file.
sentence	First level sentence inside an article.
S	Coordinated or subordinated sentence starting from second level.
sn	Nominal group.
grup.nom	Contents of the nominal group.
n	Core of the nominal group. It is a leaf node.
conj	Conjunctions group.
c	Core of the conjunctions group. It is a leaf node.
grup.verb	Verbal group.
v	Core of the verbal group. It is a leaf node.
sadv	Adverbial group.
grup.adv	Contents of the adverbial group.
r	Core of the adverbial group. It is a leaf node.
sp	Prepositional group.
prep	Contents of the prepositional group.
s	Core of the prepositional group. It is a leaf node.
spec	Determinant group.
d	Core of the determinant group. It is a leaf node.
sa	Adjectival group.
grup.a	Contents of the adjectival group.
a	Core of the adjectival group. It is a leaf node.
f	Punctuation mark.
neg	Negative modifier.
infinitiu	Infinitive.
z	Number or ammount. It is a leaf node.
participi	Participle.

relatiu	Simplex or complex relative.
p	Pronoun. It is a leaf node.
morfema.pronominal	Reflexive pronoun or impersonal verb mark.
gerundi	Gerund.
inc	Peripheral group.
morfema.verbal	Verbal morpheme.
w	Date, it is a leaf node.
interjeccio	Group with interjection.
i	Core of interjection.

In relation to the `func` and `tem` attributes, a non-exhaustive list of values for identifying the syntactic functions of the groups of a sentence can be found in table 4.2.

Table 4.2: Values of `func` and `tem` attributes of the AnCora corpus.

func	tem	Description
ao		Apposition.
atr		Attribute.
cag		Agent.
cc	cau	Circumstantial complement of cause.
cc	loc	Circumstantial complement of place.
cc	ext	Circumstantial complement of quantity.
cc	fin	Circumstantial complement of finality.
cc	ins	Circumstantial complement of instrument.
cc	mnr	Circumstantial complement of mode.
cc	tmp	Circumstantial complement of time.
cd		Direct complement.
ci		Indirect complement.
cn		Noun complement.
cpred		Predicative complement.
creg		Regime complement.
suj		Subject.

4.1.2 Adaptation of the AnCora corpus to MiSintaxis

Once we have a detailed knowledge of how the syntactic information of the sentences is represented in AnCora, we must perform a transformation of this information to a suitable format to carry out the syntactic analysis required in MiSintaxis. The output notation of this transformation process must be simple and adequate to measure the accuracy of the constituent analysis. For this reason we have chosen to use the common Peen Treebank notation [11] in which syntactic structures are delimited with parenthesis.

The adaptation from AnCora to MiSintaxis is performed with a simple translation based on the

XML tree of each article in the corpus. We use the standard `xml` package included in the Python distribution to parse the XML source and retrieve the root node of the tree. Then, we perform a traversal of the tree with recursive methods, one for each tag, that is in charge of generating a string in MiSintaxis form, using the abbreviations indicated in table 4.3. The following python code is a summary of this translation. A dictionary in the `Converter` class associates each tag with the name of a method. Usually, we use a method name that corresponds to the tag name, but in some cases we employ a common method for several tags.

```
import xml.etree.ElementTree as ET
class Converter:

    tag_method = {
        'S': 'S', 'sn': 'sn', 'grup.nom': 'grup_nom', 'n': 'n',
        'conj': 'conj', 'c': 'default', ... }

    ...

    self.call(node, path) {
        method = self.tag_method[node.tag]
        try:
            m = getattr(self, method)
            return m(node, path)
    }

    self.article(root, path) {
        output = ''
        for child in root:
            sentence_misintaxis = self.call(child, path)
            sentence_text = self.serialize_text(sentence_misintaxis)
            sentence_tree = self.serialize_tree(sentence_misintaxis)
            output += f'{self.start_tok}{sentence_text}{self.end_tok}'
            output += f'{self.start_tok}{sentence_tree}{self.end_tok}\n'
        return output
    }

    def translate(self, path):
        tree = ET.parse(path)
        root = tree.getroot()
        return self.article(root, path)
}
```

As a result, we obtain a complete translation of the AnCora corpus into a MiSintaxis corpus where each sentence is represented in two formats: in plain text and in treebank format, but using brackets instead of parenthesis to take into account that Spanish can use parenthesis as punctuation marks. The plain and treebank versions of the sentence are separated with <s> and </s> to mark the start and end of each part. For example, the following is the translation of the AnCora sentence shown at the beginning of the section 4.1.1

```
<s>La final de copa entre Inglaterra y Alemania ha tenido un efecto
positivo , aunque haya pasado casi inadvertido .</s><s>[O.Compuesta
[GN/S [Det La] [N final] [GPrep/CN [E de] [GN/T [N copa]]] [GPrep/CN
[E entre] [GN/T [N Inglaterra y Alemania]]]] [GV/PV [NP ha tenido]
[GN/CD [Det un] [N efecto] [GAdj/CN [Adj positivo]]] [OS.Adverbial/AP
[Punt ,] [nx aunque] [SO él/ella] [GV/PV [NP haya pasado] [GAdj/PVO
[GAdv [Adv casi]] [Adj inadvertido]]]]] [Punt .]]</s>
```

Table 4.3: Abbreviations of syntactic functions in MiSintaxis

Syntactic function	Abbreviation	Syntactic function	Abbreviation
Apposition	AP	Grupo verbal	GV
Attribute	ATR	Impersonal mark	MImp
Agent complement	CAg	Reflexive mark	PasRef
Circumstancial complement	CC	Nexus	nx
Regime complement	CR	Nucleus	N
Adjectival complement	CAdj	Predicate nucleus	NP
Adverbial complement	CAdv	Sentence	O
Name complement	CN	Subordinate sentence	OS
Direct complement	CD	Nominal predicate	PN
Indirect complement	CI	Verbal predicate	PV
Predicative complement	PVO	Subject	S
Determinant	Det	Omitted subject	SO
Link	E	Term	T
Nominal group	GN	Vocative	Voc

4.1.3 Fine-tuning of Hugging Face models

Thanks to the generous support of the Intelligent Systems and Telematics research group, we have had access to a computer with an Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz, 82 GB of RAM, and an NVIDIA A100 GPU with 40 GB of RAM, 6912 FP32 CUDA Cores and a CUDA version 12 driver to fine-tune the models.

We have used four models from Hugging Face to fine-tune with the data described in the previous section. Table 4.4 lists the models with some basic characteristics about them.

Table 4.4: Characteristics of the models used to fine-tune

Model	Release date	Parameters	Max Input
PlanTL-GOB-ES/gpt2-base-bne	20.Oct.2021	117M	512 tokens
PlanTL-GOB-ES/gpt2-larg-bne	20.Oct.2021	774M	512 tokens
bigscience/bloom-560m	11.July.2022	559M	2048 tokens
bigscience/bloom-1b1	11.July.2022	1065M	2048 tokens

The tuning is done using the Python `transformers` package. We have to take into account that some sentences in the corpus have more than 512 tokens (the maximum is 1239 tokens). As a consequence, when we fine-tune the gpt2 models, we filter the dataset to use those sentences with a maximum of 512 tokens. In the case of bloom, we can use the complete dataset. We employ 80% of the sentences during the training, and keep 20% to test. The learning rate is initialized with 5e-5 (default value) and we use a weight decay of 0.01. The training is performed during 5 epochs. The following code shows the procedure of training for bloom-560m.

```
model_name = 'bigscience/bloom-560m'
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name).cuda()
dataset = AncoraDataset(good_list, tokenizer)

main_size = int(0.8 * len(dataset))
test_size = len(dataset) - main_size
main_dataset, test_dataset = random_split(dataset, [main_size, test_size])
train_size = int(0.8 * len(main_dataset))
eval_size = len(main_dataset) - train_size
train_dataset, eval_dataset = random_split(main_dataset, [train_size,
    eval_size])

training_args = TrainingArguments(output_dir='./results',
    num_train_epochs=5, logging_steps=100,
    save_strategy=IntervalStrategy.NO,
    per_device_train_batch_size=2, per_device_eval_batch_size=2,
    warmup_steps=100, weight_decay=0.01, logging_dir='./logs',
    optim='adamw_torch')
Trainer(model=model, args=training_args, train_dataset=train_dataset,
    eval_dataset=eval_dataset, data_collator=lambda data:
        {'input_ids': torch.stack([f[0] for f in data]),
         'attention_mask': torch.stack([f[1] for f in data]),
         'labels': torch.stack([f[0] for f in data])}).train()
```

The gpt2 and bloom models have a decoder-only architecture. For this reason, we instantiate them with `AutoModelForCausalLM` instead of `AutoModelForSeq2Seq`, that will be adequate for encoder-decoder architectures.

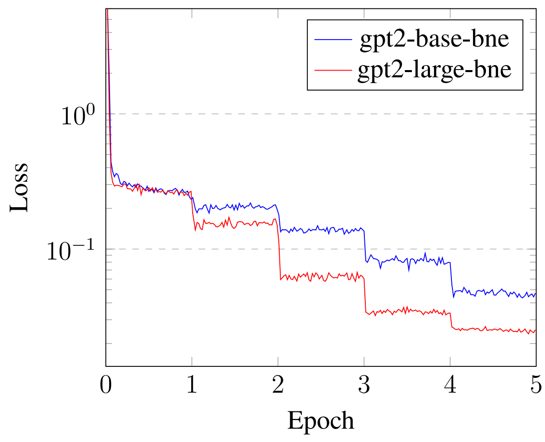
Table 4.5 summarizes the main information regarding the fine-tuning of each model. Train, evaluation and test size are measured in sentences, and runtime in seconds. Obviously, the time spent in the fine-tuning and the amount of memory used depend directly on the number of parameters of each model. The smaller model is fine-tuned in 33,29 minutes whereas the bigger model needs 10,26 hours. It has to be considered the fact that we have not used the `Accelerate` package to speed up the fine-tuning. This remains as a future line of improvement of this project.

Table 4.5: Fine-tuning of models

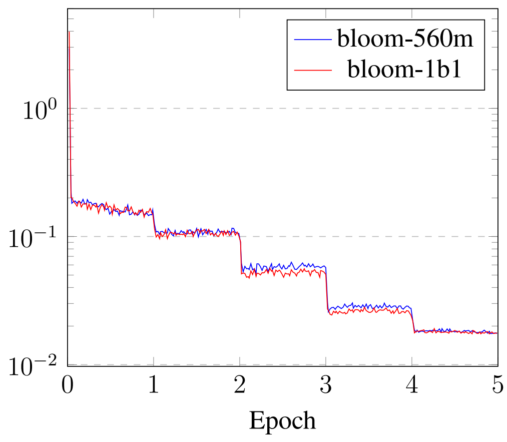
Model	**Train**	**Evaluation**	**Test**	**Runtime**	**Final loss**	**Memory use**
gpt2-base-bne	9660	2416	3019	1997.6506	0.0472	4295 MB
gpt2-larg-bne	9660	2416	3019	11268.1895	0.0253	19883 MB
bloom-560m	11072	2768	3460	22212.9313	0.0175	23307 MB
bloom-1b1	11072	2768	3460	36971.6241	0.0177	32867 MB

Figure 4.1 shows the evolution of the loss during the fine-tuning. As we can see, the plots have significant down-steps at the beginning of each epoch, and the plot for bigger models is always below that of smaller ones. It seems there is a small amount of improvement in loss if we use more epochs, but at the same time there is a danger of overfitting. For this reason, we have

considered that 5 epochs is a good trade-off. As a first approximation to the results shown in the plots, we can consider that bigger models could have better inference results than smaller ones, but we have to wait until the evaluation of tests with each model.



(a) GPT2 models



(b) Bloom models

Figure 4.1: Evolution of loss during fine-tuning

4.1.4 Inference with the generated models

Once the models have been prepared, we can move on to implementing inference with them. Given a sentence w in Spanish, the prompt we introduce to the model is <s>w</s><s> where <s> and </s> are the tokens to start and end a sentence. The model will correctly generate the parsing of the sentence if it returns an output of the form <s>w</s><s>p</s> where p is the string containing a correct constituent parsing using tree bank notation with MiSintaxis tags. We can benefit the use of <s> and </s> to extract the parsing string p from the output with a simple regular expression.

The code to perform the inference is similar in all four models. The model and its tokenizer are loaded from the folder that contains the fine-tuned version. Then we generate the output with a beam search that will be explained below. These few lines of code show how to do the inference with bloom-560m:

```python
sentence = 'Tengo un perro marrón.'
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model_name = './bloom-560m'
model = AutoModelForCausalLM.from_pretrained(model_name).to(device)
model.eval()
tokenizer = AutoTokenizer.from_pretrained(model_name)
prompt = f'<s>{sentence}</s><s>'
inputs = tokenizer(prompt, return_tensors="pt")
input_ids = inputs['input_ids'].to(device)
output = model.generate(input_ids, max_length=2048, num_beams=4,
                        num_return_sequences=1)
parsed_sentence = tokenizer.decode(output[0])
```

We have studied two possible ways to perform the inference: *greedy search* and *beam search*. With greedy search, at each time step, the model selects the next token with the highest probability. The major drawback of this strategy is that the model could miss high probability tokens that lay behind low probability ones. For this reason, we have chosen to use beam search. In this case, the inference keeps a number of hypothesis (beams) at each time step and eventually chooses the hypothesis that has the overall highest probability. In our case, we have performed the inference with 4 beams. It is pending an in-depth study of how the number of beams parameter influences the outcome.

The Hugging Face `generate` method can receive many parameters, but most of them are used to avoid repetitive generation. In our case, we are seeking the best and in many occasions unique parse of the input sentence, and this implies that intuitively the repetition is not a problem at all, but a virtue! Anyway, it will be necessary to further analyze other possibilities of generation.

4.2 Evaluation

This section is focused on the evaluation of the trained models: section 4.2.1 describes the metrics traditionally used to evaluate the results of constituent analysis; section 4.2.2 shows the results obtained for each of the models prepared; section 4.2.5 presents some examples of inputs that generate model failures; and section 4.2.4 does the same with complex inputs that are analysed with full accuracy.

4.2.1 Performance measures: recall, precision and F1

The standard tool for evaluating the quality of a parse tree for a sentence are the PARSEVAL metrics [1]. Having a *gold reference* of the parse tree that has been hand-labeled, the metric calculates how many constituents in the hypothesis generated by the model correspond to constituents in the gold standard. There are two possible metrics: recall (R) and precision (P).

$$R = \frac{\text{number of correct constituents in hypothesis}}{\text{number of constituents in gold reference}} \tag{4.1}$$

$$P = \frac{\text{number of correct constituents in hypothesis}}{\text{number of constituents in hypothesis}} \tag{4.2}$$

Both measures are usually combined to obtain the F_1 metric, the harmonic mean of the precision and recall:

$$F_1 = \frac{2 \times P \times R}{P + R} \tag{4.3}$$

It is easy to see that an exact hypothesis will have $F_1 = 1$. In this project we use PYEVALB[2], a Python package to calculate 4.1, 4.2 and 4.3 equations.

[2]`https://pypi.org/project/PYEVALB/`

4.2.2 Evaluation of the trained models with test dataset

Table 4.6 shows the results of the evaluation with mean R, P and F1 values for each model using the test dataset for each of them. The table includes the amount of memory measured in the GPU after loading the model and the tokenizer with the `pynvml` package[3]. We also show the inference mean time per sentence.

Table 4.6: Inference with AnCora test dataset

Model	Memory use	Mean time	R	P	F1
gpt2-base-bne	1984 MB	1.9420 secs.	0.7195	0.7275	0.7234
gpt2-large-bne	4582 MB	5.2488 secs.	0.8131	0.8153	0.8141
bloom-560m	3606 MB	2.9910 secs.	0.7958	0.7968	0.7963
bloom-1b1	5584 MB	3.0467 secs.	0.7793	0.7790	0.7792

The results confirm the intuition: bigger models obtain better values, but need more time and computational resources. The exception to this is bloom-1b1. To understand the circumstances under which bloom fails, a more in-depth error analysis is necessary. Consequently, we are presented with the typical engineering trade-off between the quality of inference and the allocation of resources. Some of the models spent so much time per sentence that are not usable to be integrated in a production setting. Of course, in batch mode we could speed up the inference per sentence, but this will not be a real scenario for the final application in which a student will use MiSintaxis app to retrieve the parsing of a single sentence online.

We have to take into account that the time spent in the inference is proportional to the maximum size configured in the generation method. Even if a model has been trained to process sentences with a big amount of tokens, as 512 with gpt2 and 2048 with bloom, in a real scenario the generation will be limited to much lower values in the maximum size parameter. Therefore, the maximum time has to be considered as the value in the worst case scenario with the hardware configuration used.

On the other hand, the results for R, P and F_1 can be improved by having a larger training corpus, with more quality on the preparation of the data having the advice of linguists and, possibly, by slightly increasing the number of epochs.

4.2.3 Evaluation of the trained models with Argos dataset

The previous section has shown the evaluation results of the models with the test sets extracted from AnCora. As indicated in the description of the AnCora dataset (see section 4.1.1) the sentences have been extracted from press articles. They are usually very long sentences. In this section an evaluation is performed with a little set of sentences that are closer to those that will actually be used in the MiSintaxis application.

[3]`https://pypi.org/project/pynvml`

We have named *Argos dataset* the 39 sentences contained in a Spanish language teaching textbook used in high schools in the Region of Murcia [7]. These sentences are included in chapter 7 of the book with a detailed component analysis that we have codified with the notation used in MiSintaxis. The Argos dataset includes simple, coordinated and subordinate sentences, in some cases with complex structures:

1. Mi hermano ganó la medalla de oro en esa competición.
2. ¿En esa ciudad, cómo se aplastó la rebelión militar?
3. El hermano de Juan habló de su viaje a Alemania.
4. Eran muy celosas de su intimidad.
5. El árbol fue talado ayer por los operarios municipales.
6. Ernesto estudia inglés los martes y yo practico natación los sábados.
7. El director nació en Suiza pero vivió durante toda su vida en Alemania.
8. O lee alguna novela o sale con su hijo a la plaza.
9. En derredor vibraba la noche estrellada; en el cielo palpitaban los astros.
10. Es admirable que haya donado los beneficios a esa asociación.
11. Ese político asegura que el AVE llegará a Murcia en 2020.
12. El documento explica cómo se ha de aplicar la ley.
13. Carlos no sabe si esta noche vendrá su tío Manolo.
14. No daba crédito a que te inscribieses en la carrera popular.
15. Confían en que los trabajadores cualificados permanezcan en España durante este periodo.
16. A su padre le molesta muchísimo eso, que siempre llegas tarde.
17. Trabajamos con el método que nos propuso.
18. Madrid ciudad, cuya contaminación es preocupante, es ejemplo de que debemos caminar hacia un modelo diferente de gestión urbanística.
19. Le molestaban las noches tempestuosas en las que el viento era muy fuerte.
20. Condujeron hasta el lugar donde lo habían abandonado.
21. Quienes compraron esos pisos fueron engañados por el promotor.
22. Cuando vuelva Luis, llama a Elena.
23. El que opine lo contrario debe respetar lo acordado por la mayoría.
24. Se juzga fácilmente a quienes actúan de otra manera.
25. Llegó muy tarde, lo que no gustó a su profesor.
26. Lo que ocurrió les desagradó mucho.
27. Mientras ves eso, yo haré la compra.
28. Las chicas tenemos que estar muy concienciadas porque somos las principales víctimas de este problema.

29. Tu hermano Luis no está en casa, porque no coge el teléfono.
30. Tengo que mandarles un WhatsApp enseguida para que vengan pronto.
31. Para que lo sepas, yo no pedí tu destitución esta mañana.
32. Vuestra estrategia no ha servido, de modo que es necesaria otra visión de las cosas.
33. Si eres alérgico a algún medicamento, coméntaselo al doctor.
34. Aunque tiene sueño, no puede dormirse.
35. Los golpes fueron tan fuertes que tuvo que ser ingresado en el hospital.
36. Lee tantos libros como Pedro.
37. Fue el músico más brillante que he tenido.
38. La posibilidad de presentar su caso ante el resto la animaba.
39. Defendiendo la opinión de que nada se puede hacer , no se consigue nada.

Table 4.7: Inference with Argos test dataset

Model	**Mean time**	**R**	**P**	**F1**
gpt2-base-bne	0.9983 secs.	0.8258	0.8122	0.8190
gpt2-large-bne	2.6515 secs.	0.8380	0.8263	0.8321
bloom-560m	1.1967 secs.	0.8654	0.8631	0.8642
bloom-1b1	1.2655 secs.	0.9130	0.9115	0.9123

The table 4.7 shows the results of the evaluation of the four models with these sentences. It can be seen that they obtain better values than with the AnCora test dataset, and that the bloom models outperform the gpt2 models, with quite reasonable response times.

4.2.4 Success cases

This section shows some examples of good results in constituent analysis obtained by the trained models. The Python package `nltk`[4] has been used to generate the parse tree images. This will not be the representation used in the MiSintaxis application, but it allows us to easily check the constituent analysis obtained. In MiSintaxis the parsing will be displayed using a matrix in which the words will occupy the first row and the root symbol will occupy the last row. The syntactic structure will be built in the intermediate rows by combining the cells corresponding to all the words in the span of each label.

The examples shown in the figure 4.2 correspond to outputs from the gpt-large model. The other three models also obtain mostly good results for most of the sentences in the Argos dataset, as it is shown with the F_1 metric.

[4]`https://www.nltk.org/index.html`

We can see as examples the simple sentence 3 (subfigure 4.2a), the disjunctive coordinating sentence 8 (subfigure 4.2b) and sentence 39 with a gerund conditional construction at the beginning (subfigure 4.2c).

In sentence 3 we can see how the model has been able to recognize the prepositional group "*de su viaje a Alemania*" as a regime complement (CR). In sentence 8 we can see how the model has determined that there is an omitted subject in the two sub-sentences, and that it is a third person singular. It can not determine the gender with the other constituents of the sentence. The subject is usually generated in the proximity of the verb, but it can occupy different positions, as it happens in the AnCora corpus. In sentence 39 *se* has been correctly identified as a reflexive passive mark.

4.2.5 Error cases

In this section we present three examples of sentences incorrectly analyzed by the bloom-1b1 model. The other three models also make some errors similar to those indicated here. The figure 4.3 shows the trees for sentences 13, 22 and 30.

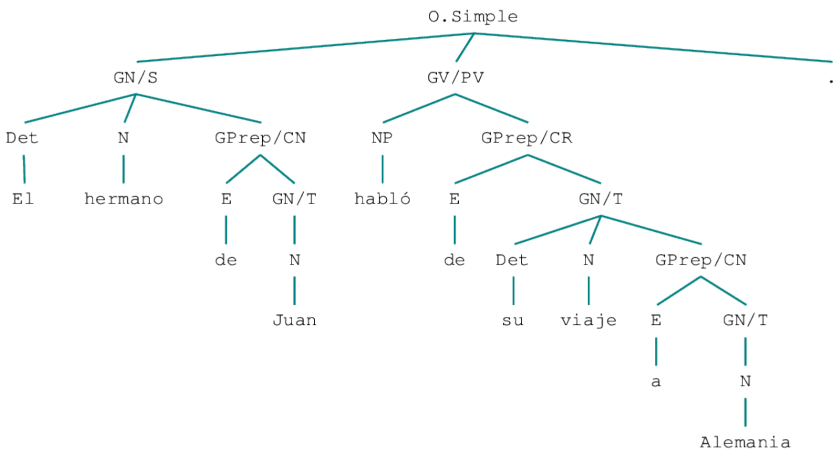
In sentence 13 (subfigure 4.3a) we encounter a problem related to an omitted subject generated by the model that is not correct. An in-depth analysis would be necessary, but possibly the training corpus contains sentences with conditional constructions in which the subject is omitted. In this case, the correct subject is "*su tío Manolo*", but it is behind the verb. By adding the omitted subject, the model marks "*su tío Manolo*" as a direct complement in a totally erroneous way. Possibly this would not happen if the token information were used bidirectionally in the generation process.

In sentence 22 (subfigure 4.3b) we have a case of ambiguity again involving an omitted subject. The verb *llama* in this sentence could be conjugated in the present tense or in the imperative. In the first case, the omitted subject would be a third person singular. But if it were imperative, it would be a second person singular.

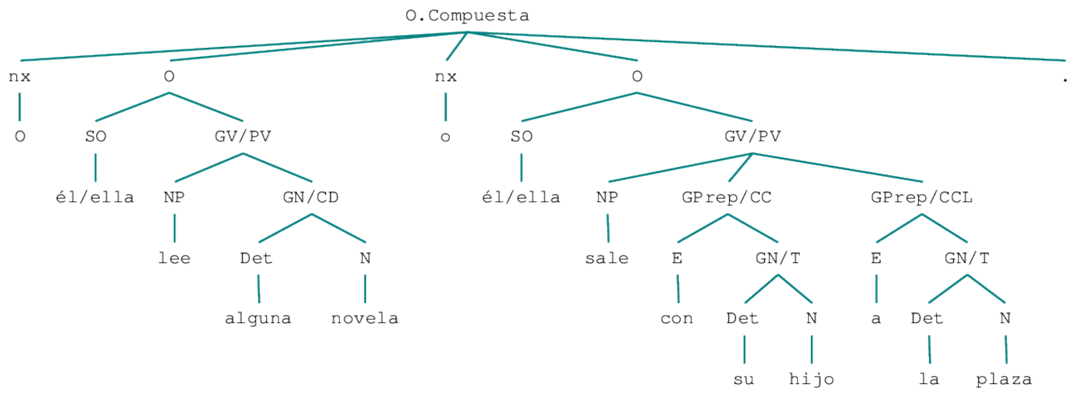
In sentence 30 (subfigure 4.3c) we encounter a serious problem: the model generates an output in which one word is different in comparison with the prompt. In this case, instead of *Whatsapp* the model has generated *Nash*. What a strange neural connection between these two words! In other similar cases, the word is not completely different, but its morphological construction does vary (change of verb tense, for example).

By making use of a model in the framework of the MiSintaxis application, the latter problem would always be detected before displaying the result. There is the possibility of asking the model to generate several sentences as output (see `num_return_sequences` parameter in the code shown in the section 4.1.4) ordered from highest to lowest probability. If the first sentence contains any erroneous word, the second option could be offered to the user. The other problems can only be partially solved by improving the training, which implies essentially a better training corpus. However, ambiguity will always be present in the use of a natural language. In this case, generating several sentences and presenting them to the user may be a solution.

(a) Sentence 3



(b) Sentence 8



(c) Sentence 39

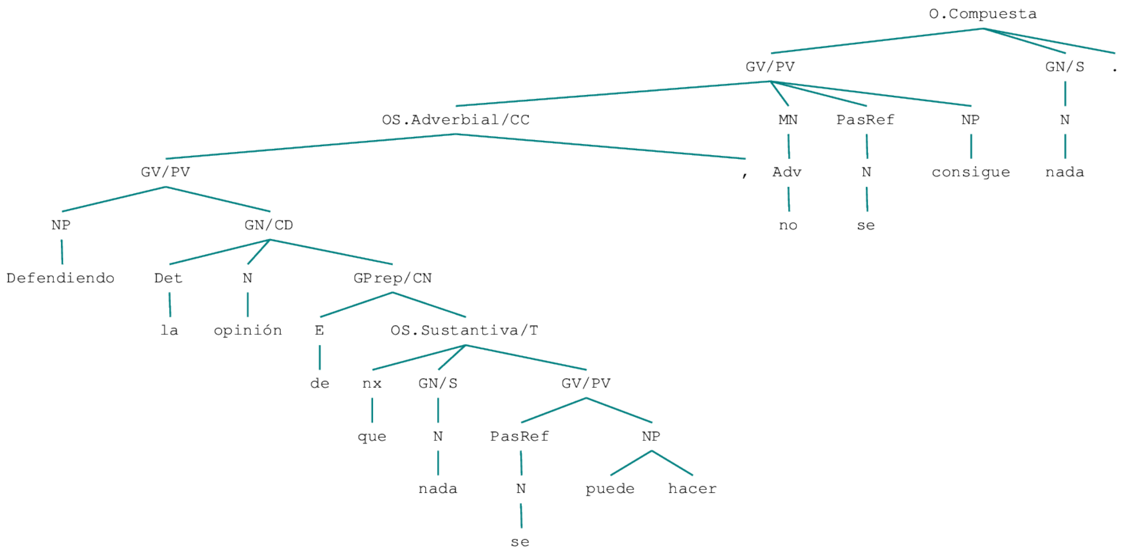
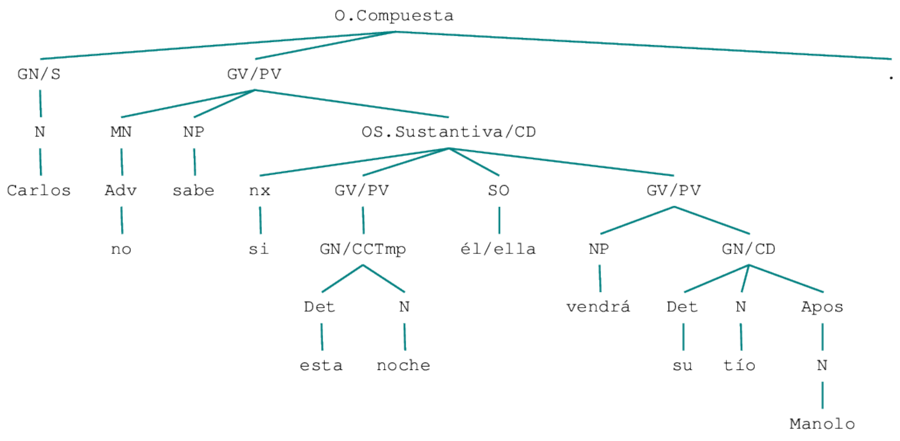
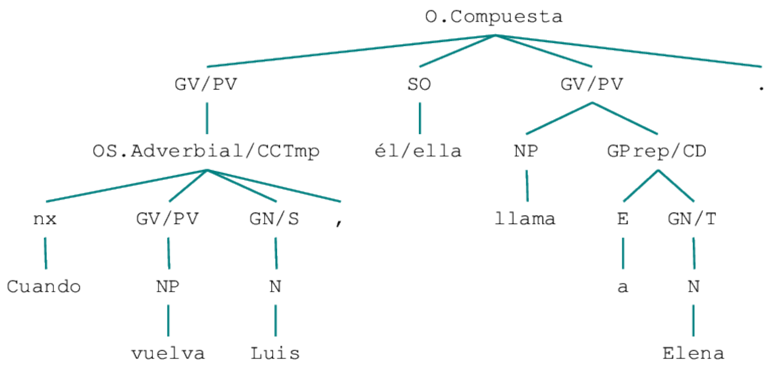


Figure 4.2: Success parsing of sentences from Argos dataset generated with gpt2-large

(a) Sentence 13



(b) Sentence 22



(c) Sentence 30

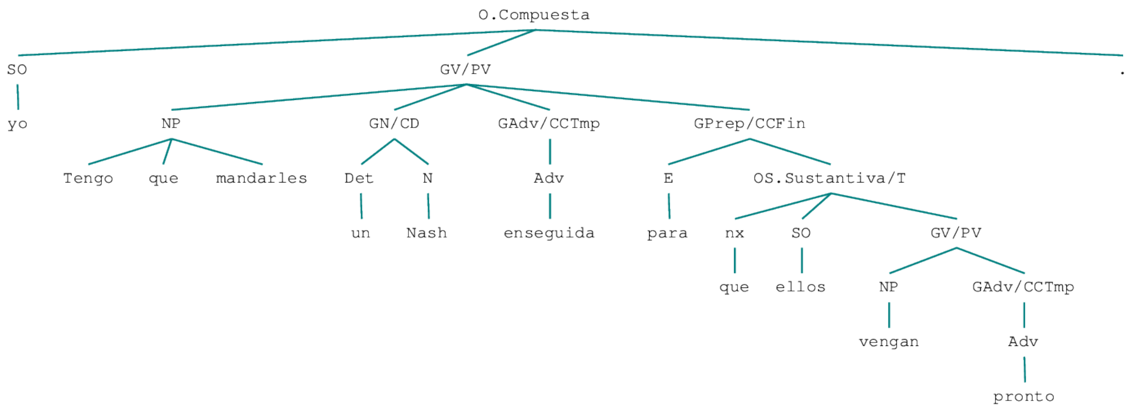


Figure 4.3: Erroneous parsing of sentences from Argos dataset generated with bloom-1b1

4.3 Study of integration in AWS

To effectively embed our constituency parsing AI model within a functional application, an exhaustive study of the Amazon Web Services (AWS) ecosystem was necessitated. This study centred on Amazon Lambda functions, API Gateway, S3 buckets, Amazon Elastic Compute Cloud (EC2), ECR, Amazon Sagemaker, the ultimate architectural framework, and a comprehensive cost analysis. The objective was to establish an optimal strategy for the deployment and management of our model, ensuring scalability, affordability, and efficiency.

4.3.1 Lambda functions

Amazon Lambda, a part of Amazon Web Services (AWS), is a compute service that allows us to run our code without provisioning or managing servers. It executes the code only when required and scales automatically, from a few requests per day to thousands per second.

In the context of this project, Lambda functions were used to handle various backend tasks such as receiving requests from our frontend service, preprocessing the input, invoking our model, post-processing the output, and finally returning the parsed sentence to the user. The code for these tasks was written in Python, which is one of the supported languages for AWS Lambda.

The use of Lambda functions was advantageous in several ways. Firstly, they eliminated the need to manage the underlying compute resources, allowing us to focus more on the core logic of our application. Secondly, they provided automatic scaling based on the incoming request volume, which helped handle varying workloads efficiently. A limitation of lambda is that you can not import most libraries directly, in order to import libraries as for example Hugging Face or torch we had to use lambda layers, which was not our case because we use lambda to trigger our Sagemaker endpoint 4.3.6. Finally, with Lambda's pay-per-use pricing model, we only paid for the compute time consumed, resulting in cost savings.

Fortunately, we have the free plan which allow us to obtain our goal without expending any money. Nonetheless, if we want to use it in production we need to talk about pricing.[5] The AWS Lambda free tier includes one million free requests per month and 400,000 GB-seconds of compute time per month, usable for functions powered by both x86, and Graviton2 processors, in aggregate. Additionally, the free tier includes 100 GiB of HTTP response streaming per month, beyond the first 6 MB per request, which are free.

Duration cost depends on the amount of memory you allocate to your function. You can allocate any amount of memory to your function between 128 MB and 10,240 MB, in 1 MB increments. Table 4.9 contains a few examples.

[5]04/06/2023 pricing could change, you can consult pricing in `https://aws.amazon.com/lambda/pricing/?nc1=h_ls`.

Architecture	**Duration**	**Requests**
x86 Price		
First 6 Billion GB-seconds/month	US$0.0000166667 for every GB-second	US$0.20 per 1M requests
Next 9 Billion GB-seconds/month	US$0.000015 for every GB-second	US$0.20 per 1M requests
Over 15 Billion GB-seconds/month	US$0.0000133334 for every GB-second	US$0.20 per 1M requests
Arm Price		
First 7.5 Billion GB-seconds/month	US$0.0000133334 for every GB-second	US$0.20 per 1M requests
Next 11.25 Billion GB-seconds/month	US$0.0000120001 for every GB-second	US$0.20 per 1M requests
Over 18.75 Billion GB-seconds/month	US$0.0000106667 for every GB-second	US$0.20 per 1M requests

Table 4.8: Comparison of x86 and Arm Prices

Memory (MB)	**Price per 1ms**
128	US$0.0000000021
512	US$0.0000000083
1024	US$0.0000000167
1536	US$0.0000000250
2048	US$0.0000000333
3072	US$0.0000000500
4096	US$0.0000000667
5120	US$0.0000000833
6144	US$0.0000001000
7168	US$0.0000001167
8192	US$0.0000001333
9216	US$0.0000001500
10240	US$0.0000001667

Table 4.9: Memory and Corresponding Price per 1ms

4.3.2 API Gateway

Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. In our project, we utilized API Gateway to build a RESTful API that acted as a front-facing interface to our clients. This API accepted parsing requests and routed them to our AWS Lambda functions. In addition, the API Gateway helped us manage traffic to our application, handle version control, and monitor application performance and health. For REST APIs, the API Gateway free tier includes one million API calls per month for up to 12 months. But in case of 'dying from success' table 4.10 shows API Rest pricing[6].

[6]04/06/2023 pricing could change, you can consult pricing in `https://aws.amazon.com/api-gateway/pricing/`.

Number of Requests (per month)	**Price (per million)**
First 333 million	US$3.50
Next 667 million	US$3.19
Next 19 billion	US$2.71
Over 20 billion	US$1.72

Table 4.10: Price per Million for Different Number of Requests per Month

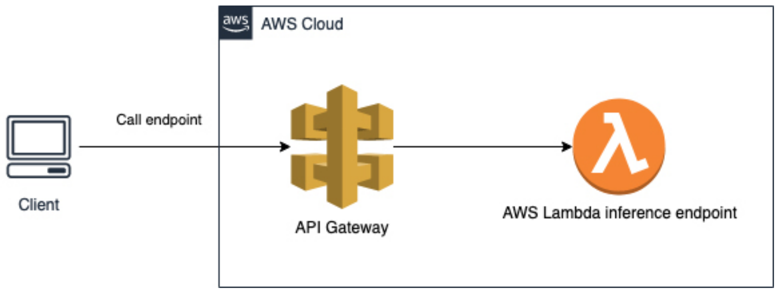


Figure 4.4: AWS architecture of API Gateway with lambda.

4.3.3 S3 bucket

Amazon Simple Storage Service (S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. In the context of this project, we used an S3 bucket as a central repository to store our trained models and other project-related artefacts such as lambda layers. This approach not only provided a reliable and secure storage solution, but also facilitated easy access to these resources when required. The rate you're charged depends on your objects' size, how long you stored the objects during the month, and the storage, for example class—S3 Standard. Data transferred from an Amazon S3 bucket to any AWS service(s) within the same AWS Region as the S3 bucket (including to a different account in the same AWS Region) is free of charges. As part of the AWS Free Tier, you can get started with Amazon S3 for free. Upon sign-up, new AWS customers receive 5 GB of Amazon S3 storage in the S3 Standard storage class; 20,000 GET Requests; 2,000 PUT, COPY, POST, or LIST Requests; and 100 GB of Data Transfer Out each month. After Free trial, table 4.11 shows pricing[7].

S3 Standard - General purpose storage for any type of data, typically used for frequently accessed data	
Amount of Data	**Price per GB**
First 50 TB / Month	US$0.023
Next 450 TB / Month	US$0.022
Over 500 TB / Month	US$0.021

Table 4.11: S3 Standard - Pricing for Different Amounts of Data

[7]04/06/2023 pricing could change, you can consult pricing in `https://aws.amazon.com/s3/pricing/`.

4.3.4 EC2

Initially, Amazon Elastic Compute Cloud (EC2) was under consideration for hosting our application's server and running the model's inferences. EC2 instances provide resizeable compute capacity in the cloud and allow for easy scaling. They also offer the flexibility to choose the instance type that best fits the workload requirements, from CPU-intensive jobs to memory-intensive ones.

However, after conducting a thorough cost-benefit analysis, we decided against deploying our application on EC2. Despite its capabilities, EC2 instances also come with overhead costs related to maintenance and management. Moreover, ensuring high availability and scalability with EC2 would necessitate configuring auto-scaling groups and load balancers, which adds to the complexity and cost.

Instead, we opted for a serverless architecture, utilizing AWS Lambda for running our code and Amazon Sagemaker for hosting the model. This decision not only saved us the trouble of managing servers, but also made our application more cost-effective[8]. Additionally, both these services offer automatic scaling, which is essential for our needs.

This does not mean that EC2 was not a suitable option, but for our specific needs and considering our resources, other AWS services proved to be more beneficial.

4.3.5 ECR Container

The first approach that we tried, involved utilizing Amazon Elastic Container Registry (ECR), an AWS service that offers storage, management, and deployment capabilities for Docker container images. This strategy was underpinned by the objective of crafting a resilient, isolated ecosystem for our application that would bolster scalability and streamline deployment processes.

In pursuit of this objective, we constructed a Docker image incorporating our model and successfully uploaded this image into an Amazon ECR repository, a process facilitated by the AWS Serverless Application Model (SAM) CLI.[9]

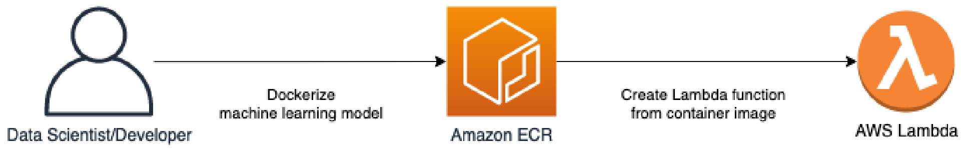

Data Scientist/Developer
Dockerize machine learning model
Amazon ECR
Create Lambda function from container image
AWS Lambda

Figure 4.5: AWS architecture of ECR with lambda.

A primary attraction of this approach was the potential for cost savings; Amazon ECR has 500

[8]Consult pricing in `https://aws.amazon.com/ec2/pricing/on-demand/`.

[9]We followed this tutorial `https://aws.amazon.com/es/blogs/machine-learning/using-container-images-to-run-pytorch-models-in-aws-lambda/`.

MB per month of storage for your private repositories for one year as part of the free trial[10]. Storage is US$0.10 per GB / month for data stored in private or public repositories. With ECR the only significant charges are for the /tmp Lambda memory utilized, implementing a warm up policy would keep the model primed for use as required, thereby averting cold starts.

However, we encountered a significant obstacle while attempting to deploy this Docker image on AWS Lambda. AWS Lambda imposes a soft limit of, 3008 MB on the size of the deployment package. Given that our Docker image, which contained the complete constituency parsing model, exceeded this restriction, it was incompatible with a Lambda function.

In response to this issue, we filed a request to exceed the soft limit and increase the available storage to the maximum limit of 10240 MB. This expansion would provide adequate memory for both the model and the necessary libraries. We were also able to configure ephemeral storage for functions operating on both x86 and Arm architectures. By default, each Lambda function receives 512 MB of ephemeral storage at no extra cost. We incurred charges only for the additional ephemeral storage configured. As of the time of this project, the cost in the eu-west-1 region was US$0.000000034 per GB-second.

To resolve this issue, we decided to pivot away from the containerized approach for deploying our model and instead leveraged Amazon Sagemaker. Amazon Sagemaker offered us the ability to host our model without any size restrictions, while also providing advantages such as auto-scaling, managed infrastructure, and seamless integration with other AWS services. This decision allowed us to maintain the benefits of isolation and scalability that the container approach would have offered, without the size constraints imposed by AWS Lambda.

It's important to note that this decision was highly contextual and dependent on the specific requirements and constraints of our project. The use of ECR and Docker containers may still be a valuable strategy for different types of projects and use cases, but for our specific scenario, the serverless approach via AWS Lambda and Sagemaker proved to be more beneficial.

4.3.6 Sagemaker

Amazon Sagemaker emerged as the solution to the challenges we faced with AWS Lambda and ECR. Sagemaker, a comprehensive machine learning service on AWS, provides tools to build, train, and deploy machine learning models swiftly and at scale.

In our case, Sagemaker was used to host our constituency parsing model without any size limitations, a fundamental requirement that couldn't be fulfilled by AWS Lambda due to its inherent restrictions. Unlike ECR, Sagemaker offered automatic scaling capabilities which adjusted to the incoming request volume, thereby providing a scalable solution to manage varying workloads effectively.

Moreover, Sagemaker's fully managed infrastructure simplified model deployment and operation, freeing us from the need to manually manage servers. This aspect was instrumental in

[10] 04/06/2023 pricing could change, you can consult pricing in `https://aws.amazon.com/ecr/pricing/`.

allowing us to focus on the core logic of our application, rather than the intricacies of server management and maintenance.

Additionally, Sagemaker's integration capabilities with other AWS services added a layer of simplicity to our process. It was straightforward to link our Sagemaker endpoint with AWS Lambda, which was then tasked with preprocessing the input, invoking our model, and post-processing the output. To achieve this, we developed an inference script and bundled it with the model scripts into a tar file. This tar file was subsequently uploaded to S3. Sagemaker was then used to instantiate a Hugging Face Model Class and to deploy the endpoint. This endpoint was designed to be triggered by a Lambda function, which is accessible thanks to the API Gateway service, thus effectively bridging the interaction between our model and the user interface.

```python
from sagemaker.huggingface.model import HuggingFaceModel


# create Hugging Face Model Class
huggingface_model = HuggingFaceModel(
   model_data=s3_location,       # path to your model and script
   role=role,                    # iam role with permissions to create an
    Endpoint
   transformers_version="4.26",  # transformers version used
   pytorch_version="1.13",        # pytorch version used
   py_version='py39',            # python version used
)

# deploy the endpoint endpoint
predictor = huggingface_model.deploy(
    initial_instance_count=1,
    instance_type="ml.m5.xlarge" #For CPU 4cores 16GiB 0,257 USD/hour
    )
```

Crucially, while Sagemaker does have associated costs[11], its pay-per-use model meant we were only billed for the compute time we actually consumed, and Amazon SageMaker has a lot of functionalities which are free to try. This pricing model helped us effectively manage our costs while still enjoying the benefits of a powerful and scalable solution. As we can see in our code 4.3.6 we are using ml.m5.xlarge instance which as shown in table 4.12 has a price of US$0.257 per hour, which is the most expensive service that we are going to use.

In summary, adopting Sagemaker was instrumental in overcoming the deployment issues we encountered with AWS Lambda and ECR. While the choice to use Sagemaker was very specific to our project's needs and constraints, it proved to be an efficient, scalable, and cost-effective solution for hosting our constituency parsing model.

[11]check pricing at `https://aws.amazon.com/sagemaker/pricing/`.

Instances	**vCPU**	**Memory**	**Price per Hour**	**Price per month**
Standard Instances				
ml.t3.medium	2	4 GiB	US$0.055	US$39.6
ml.t3.large	2	8 GiB	US$0.109	US$78.48
ml.t3.xlarge	4	16 GiB	US$0.219	US$157.68
ml.t3.2xlarge	8	32 GiB	US$0.438	US$315.36
ml.m5.large	2	8 GiB	US$0.128	US$92.16
ml.m5.xlarge	4	16 GiB	US$0.257	US$185.04
ml.m5.2xlarge	8	32 GiB	US$0.514	US$370.08
Compute Optimized				
ml.c5.xlarge	4	8 GiB	US$0.23	US$165.6
ml.c5.2xlarge	8	16 GiB	US$0.461	US$ 331.92
Memory Optimized				
ml.r5.large	2	16 GiB	US$0.169	US$121.68
Accelerated Computing				
ml.g5.xlarge	4	16 GiB	US$1.57	US$1130.4

Table 4.12: Price for Different Standard Instances

4.3.7 Final architecture

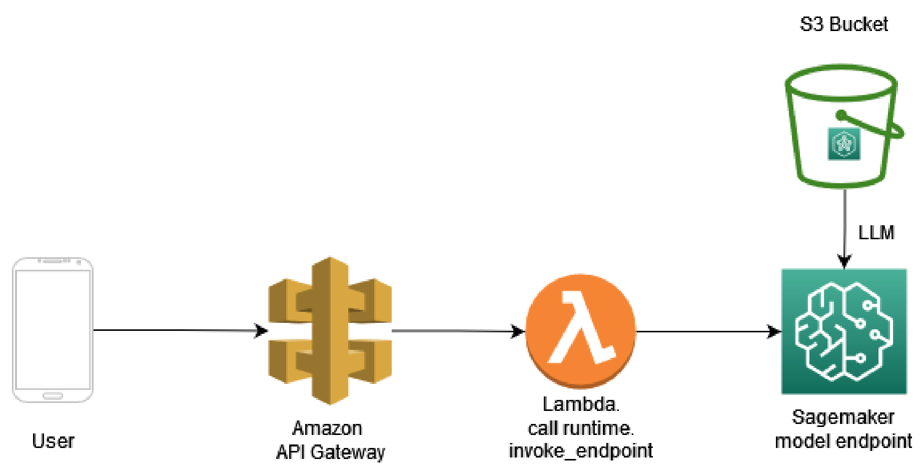


Figure 4.6: AWS architecture of Sagemaker with lambda.
`https://acortar.link/ZoNDX3`

The final architecture of our solution is a harmonious blend of different Amazon Web Services, each serving a unique role in the functioning of our application. Here, we provide a detailed breakdown of the architecture and the role of each AWS service employed in our system.

The front end of our application, primarily responsible for user interaction, communicates with an AWS Lambda function via an API Gateway. This Lambda function is crucial in managing the preprocessing of input, invoking our model hosted on Sagemaker, and post-processing the output. Once a user submits a sentence for parsing, the request is passed on to the Lambda function through the API Gateway.

The Lambda function, written in Python, takes the request and pass it on to our model deployed on Sagemaker via a Sagemaker endpoint.

Sagemaker, hosting our trained constituency parsing model, performs the computation and returns the parsed sentence to the Lambda function. The Lambda function then post-processes the output and sends it back to the user through the API Gateway.

To ensure high availability and data durability, we used Amazon S3 for storing our trained models as well as the inference script. This means even if a model needs to be redeployed or updated, we have a secure and reliable storage solution.

Furthermore, we employed AWS CloudWatch for monitoring our system. CloudWatch provides us with data and actionable insights to monitor our applications, understand and respond to system-wide performance changes, optimize resource utilization, and get a unified view of operational health.

Overall, this architecture has been designed with scalability, efficiency, and cost-effectiveness in mind. Each component has been selected and configured in a way that best fits our specific requirements, and together they provide a robust infrastructure for our constituency parsing model application.

4.3.8 Cost Analysis

We were fortunate to have the backing of the AWS EdTech program for this project. Their generous support provided us with an allocation of US$5000 to explore and utilize their services. This not only facilitated the implementation of our project, but also gave us the opportunity to delve into and experiment with various AWS offerings, furthering our understanding and knowledge of the platform's capabilities.

As part of our real-world application testing, we deployed the bigscience/bloom-560m model on the ml.m5.xlarge instance. The results from the AWS Lambda function were as follows: The total duration of the function execution was 12185.62 ms, with a billed duration of 12186 ms. The function utilized a memory size of 130 MB, with the maximum memory used capped at 68 MB. When we upload this into production, we will have to test the different models with different instances of Sagemaker in order to take into account the time that lambda takes to give a response. Its cost can be seen in column "Duration" of table 4.8 and table 4.9.

In order to make a cost analysis for our first year, we have to consider fixed cost and variables cost. Our fixed cost will be Sagemaker instance, ml.m5.xlarge has a cost of US$185.04 per month and US$2220.48 per year. As S3 bucket has a free trial of 1 year 5 GB, knowing that the sum of our model and the lambda layer don't exceed the limit, we don't have to worry about it. Our variables costs would be API gateway service and lambda service, which will be increased with the number of calls. Lambda free trial includes one million requests per month so 12 million requests per year 400,000 GB-seconds of compute time per month so 4,800,000 GB-seconds per year. API Gateway free trial includes one million API calls for the first year. If we exceed the free trial for lambda services we should consult table 4.8 and for API Gateway consult table 4.10. Variables cost for the first 100 Million request for lambda are 0.2 per 1M request and for API Gateway 3.5 per 1M request.

The figure 4.7 depicts the cost analysis for our first year of operations. The x-axis represents the volume of requests made in millions, while the y-axis denotes the cost in USD. The fixed cost of the operation, shown in blue, remains constant at US$2220.48 irrespective of the number of requests. On the other hand, the variable cost, represented in green, increases linearly as the number of requests grows, starting from zero for no requests and reaching US$332.6 for a hundred million requests. The total cost, displayed in red, combines these two factors, starting from the base fixed cost and increasing steadily with the variable cost, peaking at US$2553.08 for a hundred million requests. This graph provides a clear visualization of the financial considerations associated with running our AWS services at different scales of operation.

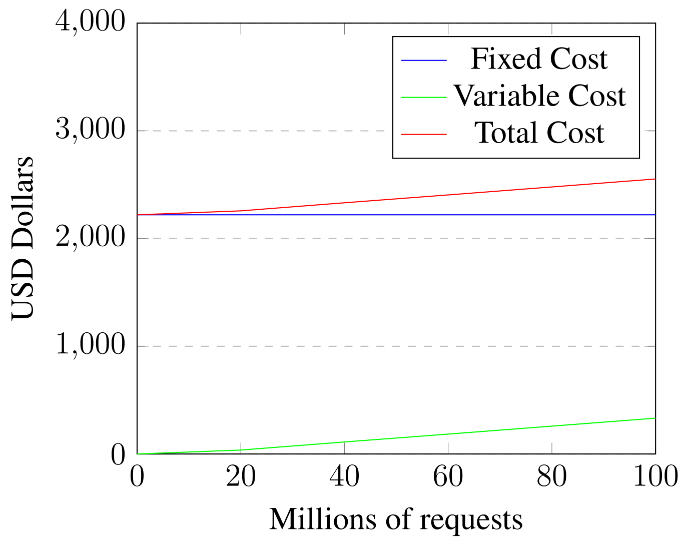


Figure 4.7: Cost analysis for our first year

Chapter 5

Conclusions and Future Directions

This chapter outlines the conclusions and prospective paths for further exploration and development related to this project.

5.1 Conclusions

Through this project, it is evident that transformers have established themselves as the gold standard in the vast majority of natural language processing (NLP) tasks. The task of constituency parsing can indeed be enhanced by fine-tuning transformer-based Large Language Models specifically for this task.

However, a significant corpus is essential to achieve success in this task. We utilized a modified version of the AnCora corpus for this study, but our model faced challenges in handling ambiguous sentences due to the limited presence of such instances in AnCora [12]. Additionally, the AnCora corpus predominantly comprises newspaper sentences, which may not mirror the regular queries a student might ask for analysis. Therefore, to improve response quality and model performance, it's crucial to expand our corpus.

In pursuit of keeping this work open and accessible, I opted to deploy the model on the Amazon Web Services (AWS) platform. AWS offers the necessary services for deploying our model, enabling widespread use through our application, MiSintaxis. As per the cost analysis, we can see that for less than 2600$ for a hundred million requests, our model can remain accessible during the first year. Consequently, students learning Spanish worldwide stand to gain from this tool.

Given that GPT-2 and Bloom are multilingual, they can be trained on different languages. The prerequisite for this expansion is a corpus of similar or greater volume than AnCora, like The Penn Treebank [11]. This approach would enable us to extend the benefits of our project to students of various languages

5.2 Future Directions

This study has laid the groundwork for new research in the realm of fine-tuning Large Language Models (LLMs) to master the task of constituency parsing. Future research could potentially explore alternative methods, such as combining LLMs with the Cocke–Younger–Kasami (CYK) algorithm.

The immediate next step I envisage is to enrich the corpus with ambiguous sentences and sample sentences that students are likely to pose. In addition to the aforementioned, there are also certain linguistic components that the designers of AnCora overlooked, such as the *Circumstantial Complement of Company* (in Spanish, "Complemento Circustancial de Compañía"). Future endeavors should consider incorporating these elements, further enhancing the scope and effectiveness of the parsing model. This extension of the model will ensure that it captures a broader range of grammatical constructs prevalent in everyday Spanish, thereby increasing its utility and accuracy.

To improve the quality of sentences, I propose employing a technique known as Reinforcement Learning from Human Feedback (RLHF), aided by the users of MiSintaxis. Since the model can generate multiple responses during an initial phase, I would solicit users to select the response they deem most accurate from the options provided.

However, this approach raises the concern that users, who are typically students, may not possess comprehensive expertise in syntax. To mitigate this issue, I would store potential high-quality sentences in a database, subjecting them to a review process. If these sentences meet our criteria, they would be integrated into the corpus.

An alternative solution is to prioritize the feedback of students who demonstrate superior proficiency. This can be achieved by examining their responses to exercises and giving consideration only to the sentences proposed by the top-performing students. Alternatively, we could devise an algorithm that assigns relevance scores to our model based on the user profile. This strategy would help to refine our model based on high-quality input, thus improving its performance and utility in the long run.

Furthermore, when transitioning this model into a production environment, it will be crucial to experiment with different Sagemaker instances, testing various models. This is particularly important because the response time of AWS Lambda functions can directly impact the overall operational costs on AWS, as discussed in Section 4.3.8. Therefore, careful selection and optimization of Sagemaker instances will be essential to ensure the balance of cost-effectiveness and performance in our deployed application.

Bibliography

[1] E. Black, S. Abney, D. Flickenger, C. Gdaniec, R. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B. Santorini, and T. Strzalkowski. A procedure for quantitatively comparing the syntactic coverage of English grammars. In *Speech and Natural Language: Proceedings of a Workshop Held at Pacific Grove, California, February 19-22, 1991*, 1991. URL: https://aclanthology.org/H91-1060.

[2] BORM. Decreto nº 251/2022, de 22 de diciembre, por el que se establece la ordenación y el currículo de bachillerato en la comunidad autónoma de la región de murcia. 24 de diciembre de 2022.

[3] Luis Chiruzzo. *Statistical Deep parsing for Spanish.* PhD thesis, Universidad de la República (Uruguay). Facultad de Ingeniería, 2020.

[4] Eugene Chiu, Jocelyn Lin, Brok Mcferron, Noshirwan Petigara, and Satwiksai Seshasai. Mathematical theory of claude shannon. *Work Pap*, 2001.

[5] James Cross and Liang Huang. Incremental parsing with minimal features using bi-directional lstm, 2016. arXiv:1606.06406.

[6] Bruce Eckel and Bruce Eckel. *Piensa en Java.* Prentice Hall, Madrid, 2 edition, 2002.

[7] Juan Antonio Belmote Sánchez, Antonio Fernández del Amor, Inmaculada Perán Rex, Agustín Pérez Leal. *Lengua Castellana y Literatura. Proyecto ARGOS Murcia.* Sansy Ediciones, 2018.

[8] Daniel Jurafsky, Chuck Wooters, Gary Tajchman, Jonathan Segal, Andreas Stolcke, Eric Fosler, and Nelson Morgan. The berkeley restaurant project. In *Third International Conference on Spoken Language Processing*, 1994.

[9] Martin James H. Jurafsky Dan. *Speech and language processing*. Draft book, 2023. URL: https://web.stanford.edu/~jurafsky/slp3/.

[10] Nikita Kitaev and Dan Klein. Constituency parsing with a self-attentive encoder, 2018. arXiv:1805.01052.

[11] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19(2):313–330, jun 1993.

[12] Mariona Taulé, Aina Peris, Horacio Rodríguez. Iarg-ancora: Spanish corpus annotated with implicit arguments. In *Language Resources and Evaluation, Vol. 50(3): 549-584, Springer-Verlag, Netherlands*, 2016. URL: `https://link.springer.com/article/10.1007/s10579-015-9334-3`.

[13] Khalil Mrini, Franck Dernoncourt, Quan Tran, Trung Bui, Walter Chang, and Ndapa Nakashole. Rethinking self-attention: Towards interpretability in neural parsing, 2020. `arXiv:1911.03875`.

[14] Manuel Peñalver Castillo. Nebrija: de la gramática de ayer a la gramática de hoy. *Cauce, 1992,(14-15): 221-232*, 1992.

[15] Real Academia Española, Asociación de Academias de la Lengua Española ASALE. *Nueva gramatica BASICA de la lengua española*. Espasa Libros, 2011.

[16] Real Academia Española de la Lengua. *Glosario de térrminos gramaticales*. Ediciones Universidad de Salamanca, 2020.

[17] Eric Ries. *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Business, 2011.

[18] MiSintaxis team. Misintaxis, 2023. Accessed: 2023-05-17. URL: `https://misintaxis.com/`.

[19] Lewis Tunstall, Leandro von Werra, and Thomas Wolf. *Natural Language Processing with Transformers, Revised Edition*. O'Reilly Media, Inc., May 2022.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. `arXiv:1706.03762`.

[21] Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. *Advances in neural information processing systems*, 28, 2015.