Universidad ORT Uruguay
Facultad de Ingeniería

Exploring Attention Patterns and Neural Activations in Transformer Architectures for Sequence Classification in Context Free Grammars

Entregado como requisito para la obtención del título de Ingeniero en Sistemas

Matías Molinolo De Ferrari - 231323

Tutores: Dr. Sergio Yovine, Dr. Franz Mayr

2024

Declaración de Autoría

Yo, Matias Molinolo De Ferrari, declaro que el trabajo que se presenta en esta obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba el Proyecto Final de Ingeniería en Sistemas;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente mía;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mi;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

[illegible]

Matias Molinolo De Ferrari

01-10-2024

Agradecimientos

A mi familia y amigos, por su apoyo y acompañamiento constante en esta etapa y por escucharme hablar horas y horas de una tesis y experimentos que salían bien (y no tan bien).

A mis tutores, Dr. Sergio Yovine y Dr. Franz Mayr, por su apoyo y las discusiones que dieron fruto a este trabajo.

Proyectos ANII

Esta tesis fue parcialmente financiada por ANII en el marco de los proyectos `IA_1_2022_1_173516` y `FMV_1_2023_1_175864`

Abstract

This work explores attention patterns and neural activations within Transformer architectures from a *mechanistic interpretability* point of view, specifically, on their application to sequence classification tasks related to context-free grammars, focusing on Dyck-k languages. We investigate whether Transformers, through their attention mechanisms, can effectively model and classify the aforementioned languages, which serve as a canonical example of context-free grammars. The work also addresses the broader issue of trainability of these models, analyzing how the model architecture impacts its ability to learn recursive structures. By employing Transformers trained on sequences of Dyck-k languages, this work empirically shows the attention patterns that emerge align with the structural dependencies within the sequences. Bidirectional masking was found to significantly enhance the model's performance, leading to perfect accuracy in classification tasks, while causal masking introduced limitations in trainability and generalization. The work further emphasizes the importance of attention mechanisms in parsing and recognizing hierarchical languages, contributing to discussions on the explainability and interpretability of neural networks. A detailed analysis of experimental results and attention matrices provide insights into the internal workings of these models, suggesting that these architectures, when properly trained, are capable of capturing complex syntactical structures of context-free languages without the need of recursion. A key outcome of this research is the development of the `transformer-checker` library, a tool designed to facilitate the training, evaluation, and visualization of transformers on formal language tasks. The tool integrates an explainability module to visualize attention matrices. The code is publicly available.

Abstract Español

Este trabajo explora los patrones de atención y activaciones neuronales dentro de modelos con arquitecturas Transformer, desde un punto de vista de la *interpretabilidad mecanística*, específicamente en clasificación de secuencias pertenecientes a gramáticas libres de contexto, enfocándose en lenguajes Dyck-k. Se investigó si los Transformers, a través de sus mecanismos de atención, pueden modelar y clasificar efectivamente los lenguajes mencionados anteriormente, que sirven como ejemplo canónico de las gramáticas libres de contexto. El trabajo apunta también al problema más amplio de la entrenabilidad de estos modelos, analizando cómo la arquitectura impacta su capacidad de aprender estructuras recursivas. Al usar

Transformers entrenados en secuencias de lenguajes Dyck-k, este trabajo muestra de forma empírica que los patrones de atención que surgen se alinean con las dependencias estructurales dentro de las secuencias. Se encontró que el uso de una máscara bidireccional mejora significativamente la performance del modelo, logrando una precisión perfecta en la tarea de clasificación, mientras que el uso de máscaras causales introdujo limitaciones en la entrenabilidad y generalización. Este trabajo subraya la importancia de los mecanismos de atención en el análisis y reconocimiento de lenguajes jerárquicos, contribuyendo a la discusión acerca de la explicabilidad e interpretabilidad de los modelos neuronales. Un detallado análisis de los resultados experimentales y las matrices de atención provee información acerca del funcionamiento interno de estos modelos, sugiriendo que estas arquitecturas, cuando son entrenadas correctamente, son capaces de capturar las estructuras sintácticas complejas de los lenguajes libres de contexto sin la necesidad de recursión. Un resultado clave de esta investigación es el desarrollo de la librería `transformer-checker`, una herramienta diseñada para facilitar el entrenamiento, evaluación y visualización de Transformers en tareas de lenguajes formales. La herramienta integra un módulo de explicabilidad para visualizar las matrices de atención. El código es de acceso público.

Key words

Artificial Intelligence; Transformers; Neural networks; Attention Mechanism; Attention Patterns; Formal languages; Formal grammars; Pushdown automata; Explainability; Interpretability; Expressivity

Palabras clave

Inteligencia Artificial; Transformers; Redes neuronales; Mecanismo de Atención; Patrones de Atención; Lenguajes formales; Gramáticas formales; Autómatas a pila; Explicabilidad, Interpretabilidad; Expresividad

Contents

1 Introduction

Large Language Models (LLMs) have been a topic of interest in the field of Computer Science for the past few years, and more recently, with the release of ChatGPT [1] by OpenAI, they have become a topic of interest for the general public too.

These models are based on an architecture called Transformer [2], a type of Artificial Neural Network (ANN) well suited to process sequences, such as text. These models have grown exponentially, as seen in Figure 1.1, both in size and complexity, in the last years, and have shown to achieve state-of-the-art results in a wide variety of Natural Language Processing (NLP) and Natural Language Understanding (NLU) tasks.

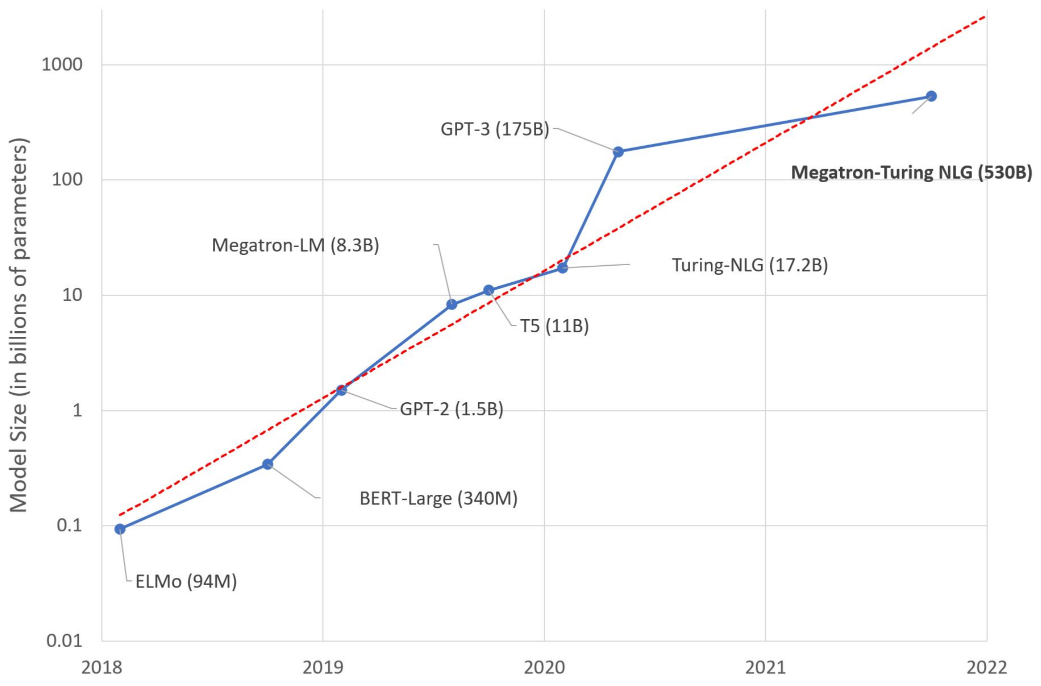


Figure 1.1: Model Sizes (2018–2021) [3]

However, despite their state-of-the-art performance, the inner workings of these models are not yet fully understood and these models are still considered opaque or *black-box* [4], which is a problem for their adoption in critical applications, such as healthcare or finance where decisions need to be explainable and interpretable. We define *explainability* as the capacity to answer questions about a model's particular decision (i.e.: classification, object detection, etc.) [5].

Moreover, there is still a boundary to the capabilities of these models, regarding which problems can or cannot be solved by them and what can be learned and expressed by these models.

Strobl et al. [6] define two lines of work regarding this problem: *expressivity* and *trainability*. This work will focus on the latter, through training different Transformers and using a *white-box* approach to view the internal workings of these models.

This approach will look at the attention mechanism and neural activations in Transformers that are able to learn and classify sequences belonging to a formal language, more specifically, context-free grammars (CFGs) such as Dyck languages, to see whether we can gain an insight into the inner workings of these models.

Chomsky's hierarchy [7] classifies CFGs as those languages that can be represented by a nondeterministic pushdown automaton, a class of automata that is more expressive than finite-state machines but less so than Turing machines [8]. Pérez, Barceló and Marinkovic propose that architectures based on self-attention, such as Transformers, are Turing complete [9], therefore, this leads us to believe that CFGs can be expressed by these neural language models.

We aim to investigate whether attention patterns and neural activations in these architectures can help explain the model's classifications, using an approach rooted in *mechanistic interpretability*, as we will look into the internal components of the model. This concept can be likened to reverse-engineering a program, but applied to neural models [10, 11]. Based on this approach, we will explore how these explanations can be leveraged to enhance both the model's performance and its interpretability.

Outline

Chapter 2 will introduce the concepts behind formal languages and context-free grammars, focusing especially on the Chomsky Hierarchy and Dyck-k languages.

Chapter 3 will introduce the Transformer architecture, with a focus towards the attention mechanism.

Chapter 4 will discuss related works and the state-of-the-art in the field of trainability, explainability and interpretability of Transformers, as well as obtained results and their implications, comparing them with those obtained by other authors.

Chapter 5 focuses on the experimental setup, engineering process, the dataset used and the training process and results.

Chapter 6 will summarize the work done and propose future work.

2 On Formal Languages

Formal languages are a fundamental concept in computer science, as they allow us to rigorously define and analyze the structure of different types of languages that we may encounter in both natural and artificial systems.

Mateescu and Salomaa [12] explain a language as three possible explanations:

1. A body of words and systems for their use common to people of the same community or nation, geographical area, or cultural tradition
2. A set or system of signs or symbols used in a more or less uniform fashion by a number of people that enables them to communicate and understand each other.
3. Any system of formalized symbols, signs, gestures, etc. used as a means of communicating thought, emotion, etc.

However, these provide a notion of "language" that is general rather than formal. When speaking of formal languages, it is essential to construct a systematic definition, or *grammar*, rather than to consider a language as something given to us by others.

To these effects, we will consider Hopcroft, Motwani and Ullman's [13] definition of a language $\mathcal{L}$, as a set of strings chosen from Σ^*, where Σ is a particular alphabet. An alphabet is defined as a finite, nonempty set of symbols. Σ^* denotes the *universal language*, which is the language formed by all possible sequences over an alphabet Σ. For example, if we were to work with binary strings, $\Sigma = \{0, 1\}$ and $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \cdots\}$.

Formally, we define Σ^* as the Kleene closure of Σ. The Kleene closure is defined, for a set of symbols $V \subseteq \Sigma$, as the set of all strings over symbols in V, including the empty string ϵ, which can be expressed by the following equation [14]:

$$V^* = \bigcup_{i=0}^{i=\infty} V^i \tag{2.1}$$

Furthermore, they define a string or word as a finite sequence of symbols chosen from an alphabet Σ. Therefore, a word can also be seen as a concatenation of symbols that start with the *empty* or *identity* symbol, ϵ.

However, not every word formed by the alphabet necessarily belongs to a language, since for each language, a specific set of rules exist that define *membership* - whether a word belongs to the language or not.

Once again, let us consider binary strings, but now we will restrict our language to those strings of even length. In this case, $\Sigma = \{0, 1\}$ and some valid strings in $\mathcal{L}$ would be $\{\epsilon, 00, 11, 0011, \cdots\}$, but $\{0, 1, 010, 110, \cdots\}$ do not belong to $\mathcal{L}$ as they are not of even length.

Furthermore, it is important to note the existence of the *empty language*, $\varnothing$, that is, the language that does not contain any words, not even the empty one - $\epsilon \notin \varnothing$. This is important, as it gives us a way to represent when a certain language cannot be constructed.

Even though the set of strings in a language may be infinite, these all stem from concatenations of a finite set, Σ, our alphabet. From this we can define a single constraint on what constitutes a language: its alphabet, Σ, must be a finite set [13].

2.1 Chomsky's Hierarchy

A grammar, according to Chomsky, is a set of rules or device that defines and enumerates the sentences of a languages. Also, these enumerators propose a set of restrictions, which then map to different types of automata, in such a way that languages that can be generated by a certain type of grammar are a proper subset of the less restrictive grammars [7].

For this work, we will focus on Type-2, or context-free grammars, which can be recognized by pushdown automata. The particularity of these grammars is they have a recursive, hierarchical structure and notation [13]. For example, palindromes are a context-free language, as the grammar that generates it is also context-free.

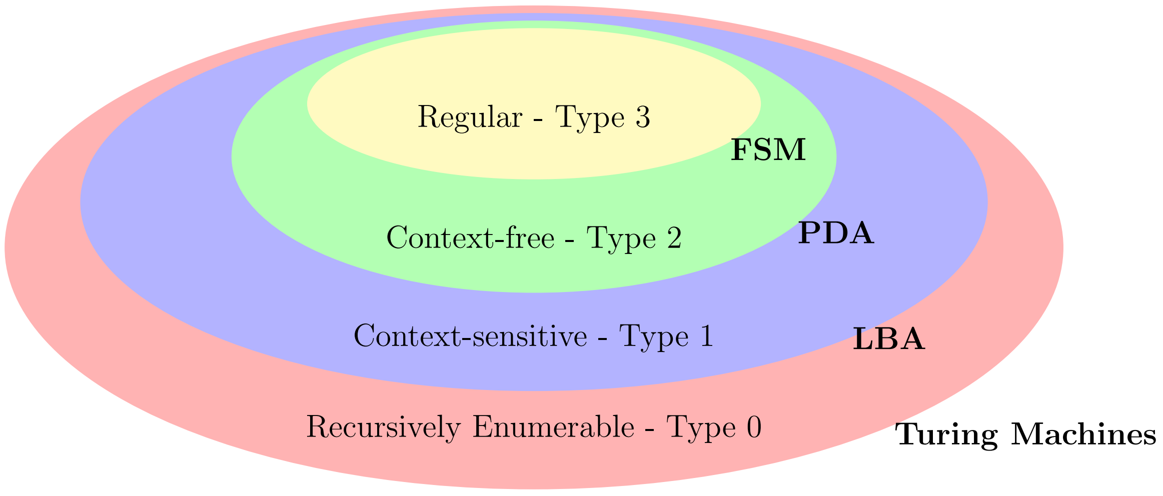


Figure 2.1: Chomsky Hierarchy

2.1.1 Automata and Grammar Recognition

From Chomsky's hierarchy, we gain insights into which type of automaton is expressive enough to recognize a particular type of language, as each type of grammar maps to a class of automaton. As we move from less restrictive languages to more restrictive languages, as seen in Figure 2.1, the automaton required to recognize the language becomes simpler.

- Turing Machines: recognize any language.
- Linear-Bounded Automata (LBA): recognize context-sensitive languages.
- Pushdown Automata (PDA): recognize context-free languages. As these automata use a stack to keep track of symbols, they are particularly useful to recognize recursive languages.
- Finite State Machines: recognize regular languages and are the simplest automata.

2.2 Context-free grammars

This work will focus on extracting attention patterns from Transformers trained on Type-2 grammars (context-free). We find these grammars of interest as they are central to the parsing of many modern programming languages and formal systems. Since they can represent recursive structures, they are particularly useful for defining constructs such as expressions, loops and function calls.

A context-free grammar $\mathcal{G}$ has four main components [15, 16]:

1. A finite set of *variables* $\mathbf{V}$ - these variables represent a language (a set of strings). We note $\mathbf{V}^*$ as the Kleene closure of the set of variables, as defined in Equation 2.1.
2. A finite set of symbols $\mathbf{T}$, called *terminals*. Note that $\mathbf{T} \subset \mathbf{V}$. Once again, $\mathbf{T}^*$ is the Kleene closure of the set.
3. A *start symbol* $\mathbf{S} \in \mathbf{V} - \mathbf{T}$.
4. A finite set of *productions*, $\mathbf{P} \subset (\mathbf{V} - \mathbf{T}) \times \mathbf{V}^*$. These represent the recursive definition of the language. Each production consists of the following:
   (a) A variable, called the *head* that is partially defined by the production.
   (b) A production symbol $\rightarrow$
   (c) A string of zero or more terminals and variables, called the *body*.

Therefore, a grammar $\mathcal{G}$ can be expressed as a four-tuple $(\mathbf{V}, \mathbf{T}, \mathbf{P}, \mathbf{S})$.

Given 2 words, $u, v \in \mathbf{V}^*$, we say that $u \rightarrow v$ if there exists a derivation, or sequence of words in $\mathbf{V}^*$ such that $u_{i-1} \rightarrow u_i$ for $i = 1, \ldots, k$ and $u_0 = u$ and $v = u_k$. The existence of a derivation is noted by $u \xrightarrow{*} v$

Usually, grammars are presented with a notable non-terminal symbol, also called *axiom*, noted by $\mathbf{S}$ [13, 15]. The language generated by this variable in the grammar is called the *language $\mathcal{L}$ generated by $\mathcal{G}$* and can be defined as the following set:

$$\mathcal{L}(\mathcal{G}) = \{w \in \mathbf{T}^* | \mathbf{S} \xrightarrow{*} w\} \tag{2.2}$$

Should X be a variable in $\mathcal{G}$, then:

$$\mathcal{L}_{\mathcal{G}}(X) = \{w \in \mathbf{T}^* | X \xrightarrow{*} w\} \tag{2.3}$$

From Equations 2.2 and 2.3, we derive that $\mathcal{L}(\mathcal{G}) = \mathcal{L}_{\mathcal{G}}(S)$. If two grammars generate the same language, they are considered *equivalent* [16].

2.3 Dyck-k Languages

Dyck-k languages are a canonical family of context-free languages, composed of strings of balanced parentheses. Let $A = \{a_1, \ldots, a_k\}$ and $\bar{A} = \{\bar{a}_1, \ldots, \bar{a}_k\}$. A Dyck-$k$ language is defined as a set of strings over the alphabet $\Sigma = A \cup \bar{A}$, where each string is a sequence of $2k$ parentheses, such that the parentheses are balanced. That is, for each string $w \in \Sigma^*$, the number of opening parentheses is equal to the number of closing parentheses, and for each prefix w' of w, the number of opening parentheses is greater than or equal to the number of closing parentheses.

The language of Dyck-k is denoted by D_k and is defined by the following grammar:

$$\begin{aligned}
\mathbf{V} &= \{S\} \cup \mathbf{T} \\
\mathbf{T} &= \{a_1, \ldots a_n\} \cup \{\bar{a_1}, \ldots, \bar{a_n}\} \\
\mathbf{P} &= \{S \rightarrow a_i S \bar{a_i} S\}_{i=1,\ldots,n} \cup \{S \rightarrow \epsilon\} \\
\mathbf{S} &= S
\end{aligned}$$

In this case, $\mathbf{T}$ is called a *matched alphabet*, as each symbol a_i has a corresponding closing symbol $\bar{a}_i$.

Chomsky-Schützenberger Theorem

We say that Dyck-k languages are canonical because they are the simplest form of context-free languages and can be used as a building block for all other context-free languages. This is known as the Chomsky-Schützenberger Theorem [17] and states that a language $\mathcal{L}$ over an alphabet Σ is context-free iff there exists:

- a matched alphabet $T \cup \bar{T}$,
- a regular language $\mathcal{R}$ over $T \cup \bar{T}$,
- a homomorphism $h : (T \cup \bar{T})^* \to \Sigma^*$

such that $\mathcal{L} = h(R \cap D_T)$, where D_T is the Dyck language over T.

It is useful to visualize a matched alphabet $T \cup \bar{T}$ as matched parentheses, with T being the set of opening parentheses and $\bar{T}$ the set of closing ones.

We find Dyck-k languages interesting to study as they can showcase subject-verb agreement in common language (English, Spanish, etc.) [18], therefore we can consider Dyck-k languages as a sort of building block for common language. Furthermore, they are the canonical form of nested structures [8]. We can see an example in the figure below:

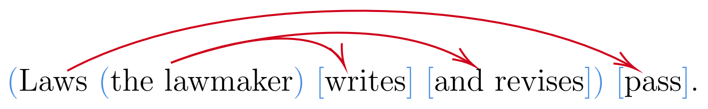


Figure 2.2: Subject-verb agreement [18]

We see that this subject-verb agreement can be expressed by the Dyck-2 word (( ) [ ] [ ]) [ ].

Furthermore, Dyck-k languages provide a way to easily parse and validate modern programming languages, for example, if we consider a C-like language, the expression shown in the figure below can be parsed by a Dyck-3 language:

```
foo( bar( arr[ index ] ), { key: value }, x + y )
```

Figure 2.3: C-like expression

Shuffle-Dyck-k

A particular, less restrictive case of Dyck-k languages is given by the so-called Shuffle-Dyck-k family of languages.

This family of languages is made up of k shuffles of Dyck-1 languages, with k different types of brackets. Each type of bracket must be well balanced, but there is no restriction to their relative order [19].

3 Transformer Architecture

Transformers [2] were proposed by Vaswani et al. in 2017. This novel architecture achieved state-of-the-art results in machine translation tasks, while dispensing with recurrent and convolutional architectures, which enabled parallelization and speedup of natural language processing tasks. The Transformer employs an encoder-decoder stack, each with multiple *blocks*, as can be seen in Figure 3.1. The key mechanism behind this architecture is called *attention*, which we will discuss in detail in a later section.

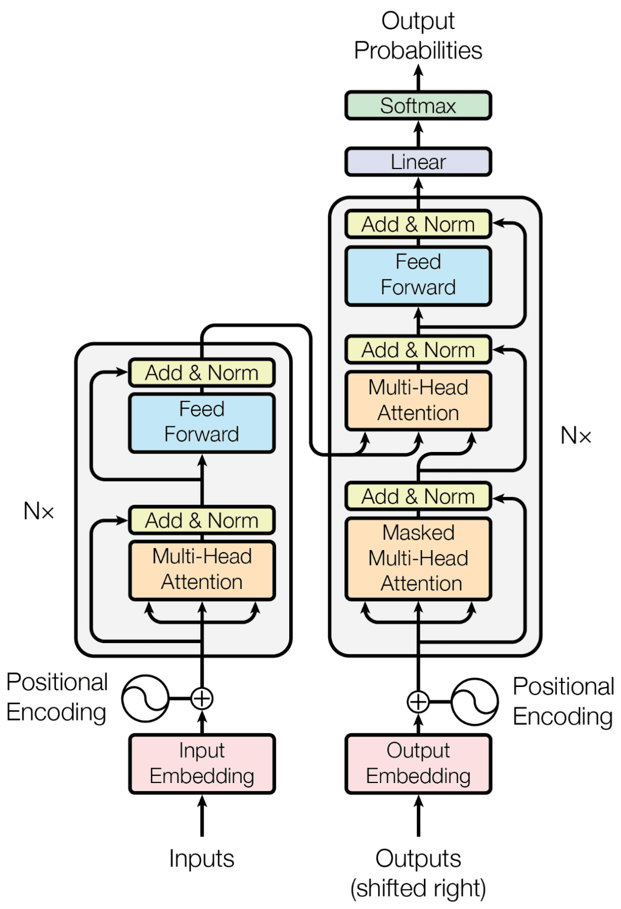


Figure 3.1: Transformer Architecture [2]

As we can see, the architecture lends itself to parallelization, since there are no recurrences or dependencies between inputs, which in turn helps speed up training and inference, but also increases the model's complexity and reduces its interpretability, due to the existence of residual paths, also called *residual connections*, which are paths that add the input without processing to the output that stems from processing that same input, similar to doing $x = x + f(x)$, layer normalizations and attention mechanisms.

For our work, we will focus on Transformer encoder stacks (which we will call encoder-only Transformers hereon forwards) with multi-head attention, as these can be used for sequence classification rather than machine translation or sequence modeling. In essence, Transformer encoder stacks map an input sequence $s \in \Sigma^*$ to a latent space $l \in (\mathbb{R}^d)^*$.

Correspondingly, Transformer decoder stacks use this latent space, and an output start sequence and convert these inputs to a new sequence $s' \in \Sigma'^*$. This is a generative process, which we will not focus on.

These encoder-only Transformers, when used along a feed-forward layer and a softmax layer, use this latent space l to output n-class classification probabilities, which we will use to determine if a string belongs to a certain language, after being trained on a dataset composed of strings of balanced and unbalanced parentheses.

Albeit Transformers are known for their state-of-the-art performance on natural language processing tasks, they cannot process words as-is; these inputs need to be mapped to a vector space $\mathbb{R}^d$. To this effect, we will use Ströbl's definition of a *word embedding*, which is a length preserving function $\mathrm{WE} : \Sigma \to \mathbb{R}^d$. This is part of the *input layer*, which is defined in detail in Equation 3.1 [6].

Figure 3.1 displays the full architecture of the original Transformer, with an encoder and decoder, each with its inner components, however, we will deal with a simplified version of this architecture, shown in Figure 3.2, and we will proceed to explain each part that makes up our encoder-only Transformers.

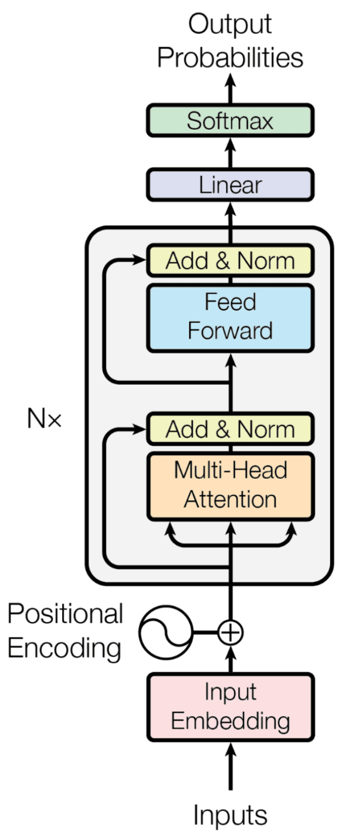


Figure 3.2: Transformer classifier Architecture

3.1 Attention

The attention mechanism is the most important part of this architecture and will be the focus of our study. In short, this mechanism will let the model know the importance of a *token* with respect to all other tokens in the sequence.

This concept was introduced by Badhanau et al. and is defined as an alignment model [20] that scores matches between inputs at a given position to outputs at another position. Even though this mechanism was applied to recurrent neural networks (RNNs), it is equivalent to the mechanism we will describe next.

According to Vaswani et al., attention functions can be described as mappings of queries and key-value pairs to an output, where queries, keys and values are all vectors belonging to a vector space $\mathbb{R}^d$, where d is called the *embedding* or *representation* dimension.

Input layer

As mentioned previously, Ströbl defines these vectors as mappings that stem from applying a length preserving function, called the *input layer*, $f : \Sigma^* \to (\mathbb{R}^d)^*$ to input strings[1] [6]. This function consists of two components, the previously defined word embedding and a positional encoding, PE, such that:

$$f(w_0 \dots w_{n-1})_i = \text{WE}(w_i) + \text{PE}(i) \tag{3.1}$$

Positional encoding

A positional encoding (or embedding) is a function that maps the position of a token to our vector space $\mathbb{R}^d$. Formally,

$$PE_n : |n| \to \mathbb{R}^d \text{ for } n \in \mathbb{N} \tag{3.2}$$

where $|n| = \{0, 1, \dots, n-1\}$ [6]. We note 2 positional encodings of interest, absolute and sinusoidal.

Absolute positional encodings were defined by Yao et al. [18] and are defined as follows:

$$\text{PE}_{(i,n)} = \frac{i}{n} \tag{3.3}$$

where i is the token position and n is the total token count (or length of the sequence).

Sinusoidal positional encodings were defined in Vaswani et al. [2], and are the seminal positional encoders for Transformers, defined as follows:

$$PE_{(pos,k)} = \begin{cases} \sin\left(\frac{pos}{10000^{\frac{k}{d_{model}}}}\right) & \text{if } k \text{ is even} \\ \\ \cos\left(\frac{pos}{10000^{\frac{k}{d_{model}}}}\right) & \text{if } k \text{ is odd} \end{cases} \tag{3.4}$$

Attention mechanism

The most commonly used attention mechanism is called scaled dot-product attention or *soft(max)* attention, which is defined as follows:

[1]Ströbl et al. use a different notation, $\Sigma^* \to (\mathbb{R}^d)^*$, compared to Vaswani et al., who represent these embeddings as $\mathbb{R}^{d \times n}$, to better align with conventions in formal language theory and emphasize variability in input sequence lengths.

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

In practice, Q, K, V in equation 3.5 refer to batched queries, keys and values respectively, where individual queries, keys and values are packed into matrices and processed simultaneously, speeding up computation.

Furthermore, this mechanism can be split and parallelized, which is then known as *multi-head attention*, and represented by the following equation:

$$\begin{aligned}\text{MultiHead}(Q,K,V) &= \text{Concat}(\text{head}_1,\ldots,\text{head}_n)W^O\\ \text{where head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)\end{aligned} \tag{3.6}$$

In this mechanism, $W_i^{\{Q,K,V\}}$ are learned parameter matrices. Splitting the attention mechanism into different *heads* allows for attending to information at different positions using different representations, without incurring in severe computational cost penalties, as the reduced dimensionality of each head allows for a computational cost similar to single-head attention with full dimensionality [2].

We can also define *hard* attention, which we define with the following equation:

$$\text{HardAttention}(Q,K,V) = \text{argmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{3.7}$$

We also note the difference between leftmost-hard and average-hard attention mechanisms, as the former looks for the first element with the maximum value, whereas the latter averages these and allows them to share weight equally [6].

Finally, we introduce the concept of *self-attention*, which is simply applying the mechanism to the input sequence, in essence, allowing us to see internal relations within the sequence [2].

We find this mechanism (*self-attention*) of interest to our work, as we believe it will provide information regarding if a certain string of parentheses is balanced, and therefore, information about its membership to a Dyck-k language.

3.2 Attention Masks

We define an attention mask as a matrix $\mathcal{M}_{i\times j}$, such that:

$$\mathcal{M}_{i\times j} = \begin{cases} 0 & \text{if condition is true} \\ -\infty & \text{otherwise} \end{cases} \tag{3.8}$$

This mask is applied to the attention matrix through element-wise summation.

We define 2 conditions of interest:

1. $i \leq j$ - we call this mechanism future or *causal* masking
2. tok$[i] \neq$ `[pad]` - we call this pad-token or *bidirectional* masking

The first condition allows the mechanism to attend *only* to previous positions, such that the mechanism cannot get information on tokens it has not yet seen.

The second condition allows for the attention mechanism to "see" the sequence as a whole, but considers only tokens that provide information, ignoring special `[pad]` tokens, used to unify input sequence lengths. This masking is well known in the field of Transformers, as it is used by Bidirectional Encoding Representations from Transformers or BERT [21].

We will analyze in further detail whether different attention masks have an effect on trainability in a later chapter.

4 On Transformer Classifiers

4.1 Trainability

Having already analyzed the Transformer architecture, we will now proceed with a discussion on whether these architectures are capable of being trained to *recognize* context-free grammars. Firstly, we need to define what *recognize* means in the context of our problem. In this case, we define *recognition* of a string as determining whether or not the string belongs to the language in question. This is essentially a membership test: is the string a valid member of the language?

However, before we dive into this analysis on the *trainability* of Transformers, which will be backed by experimental data and experiments, we first must analyze whether these architectures are capable of learning context-free grammars or not - in essence, are they *expressive* enough to generate an internal model of a context-free grammar?

In addition to discussing the expressiveness and trainability of these models on the problem of classifying sequences on their membership to CFGs, we will also focus on explainability. By analyzing key aspects of the Transformer architecture, such as attention patterns, we aim to gain insights into the inner workings and behaviour of the model when processing and encoding the underlying structures in a CFG. Explainability is crucial, not only for improving transparency and trustworthiness of these complex models, but also for understanding the limits of their trainability and expressiveness.

Bhattamistra et al. [19] prove that these architectures are expressive enough to recognize the Shuffle-Dyck-k family of languages and notes the special case of $k = 1$, where Shuffle-Dyck-k is equal to Dyck-1 languages.

Ströbl et al. [6] specify that this Transformer is one with soft-attention (i.e.: softmax attention, as defined in Section 3.1) with future masking, positional encoding only, no layer normalization and no residual connections. However, Hahn [22] proved that hard-attention Transformers cannot model Dyck-k, and that soft-attention Transformers cannot model Dyck-k with perfect cross entropy.

Yao et al. [18] describe a Transformer with 3 layers, $\frac{i}{n}$ positional encoding, soft-attention and causal masking that should be sufficient to *recognize* Dyck-k.

Table 4.1 presents the different architectures used by the authors.

The experiments conducted and detailed below are aimed towards validating the work done by the aforementioned authors, through experimentation. Unless otherwise stated, the experiments below were repeated at least 10 times, achieving the same results each time.

Author	**Layers**	**Attention**	**PE**	**Masking**
Bhattamistra et al.	Unspecified	Soft-Attention	Unspecified	Unspecified
Ströbl et al.	Unspecified	Soft-Attention	Sinusoidal	Causal
Hahn	Unspecified	Hard-Attention	Unspecified	Unspecified
Yao et al.	3	Soft-Attention	$\frac{i}{n}$	Causal

Table 4.1: Architectures used by different authors for classifying Dyck-k languages with Transformers

Summary of experiments

The table below shows the parameters used to configure our different models. Training hyperparameters will be discussed in a later chapter.

	N	d_{model}	d_{ff}	h	P_{drop}	masking	PE	Context length	$\|\Sigma\|$
1	2	256	512	1	0.1	bidirectional	none	16	2
2	2	256	512	1	0.1	causal	none	16	2
3	2	256	512	1	0.1	bidirectional	none	16	6
4	2	256	512	1	0.1	causal	none	16	6
5	3	256	512	1	0.1	causal	$\frac{i}{n}$	16	6
6	2	256	384	1	0.1	bidirectional	none	128	6
7	2	384	768	1	0.1	bidirectional	none	4096	6

Table 4.2: Transformer architectural parameters

Here, N represents the number of layers (or encoder blocks), d_{model} is the embedding dimension, d_{ff} is the dimension of the feed-forward network, h is the number of attention heads, and P_{drop} is the dropout probability [23].

The masking parameter allows us to specify the type of attention mask to use, which can be either **bidirectional** or **causal** (see Section 3.2 for details). If no mask is specified, a bidirectional mask is applied by default.

The PE (positional encoding) parameter lets us choose the type of positional encoding, which can be either **absolute** or **sinusoidal**, as described in Equations 3.3 and 3.4. If the PE parameter is set to **none**, no positional encoding is applied.

Context length is the maximum sequence length accepted by the model and $|\Sigma|$ is the size of our alphabet, as defined in Section 2.3. For example, when $|\Sigma| = 2$, we are referring to a Dyck-1 language, and with $|\Sigma| = 6$, we refer to a Dyck-3 language.

These experiments were done to visualize attention patterns in a manageable way, as longer sequence lengths will only make the visualization more complex. However, we also test the *trainability* of these Transformers on longer sequence lengths, as seen in Experiment 7.

As we seek to explore these attention matrices and their patterns, should they exist, we extract these matrices after training, to see whether patterns appear that can help show Transformers are able to learn Dyck-k languages.

Before we jump into the analysis of attention patterns, we will first define a systematic approach for interpreting these matrices. $\text{Attn}_{m,k}(i,j)$, refers to the attention at layer m, (attention) head k of `tok[`i`]` with respect to `tok[`j`]`, where `tok[`i`]` and `tok[`j`]` are the tokens at position i and j, respectively. Furthermore, $\text{Attn}_{m,k}(i)$ or the attention of `tok[`i`]` with respect to the sequence, refers to row i in the attention matrix for layer m, head k.

Also, we find it important to note that all attention matrices shown below are min-max normalized to the range $[-1,+1]$, for easier visualization of attention scores.

4.1.1 Experiments on Dyck-1 Classifier Transformers

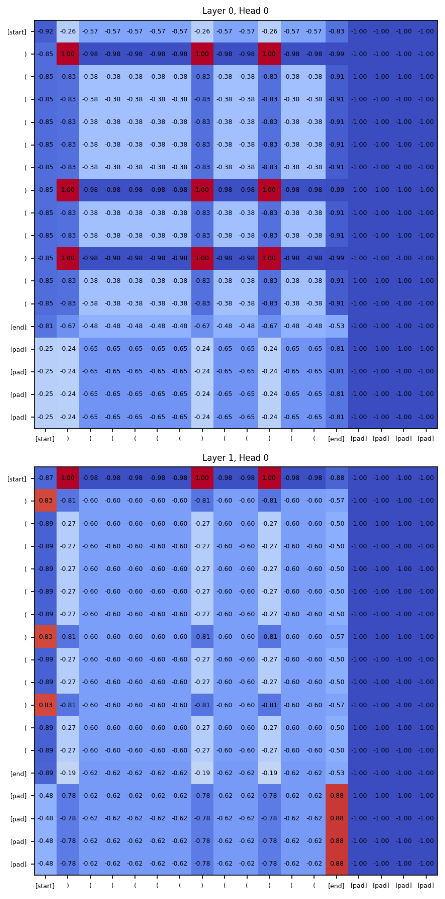
We will now describe the experiments conducted with two Transformers built to classify Dyck-1 languages - these correspond to the first 2 experiments in Table 4.2.

4.1.1.1 Bidirectionally masked Transformer

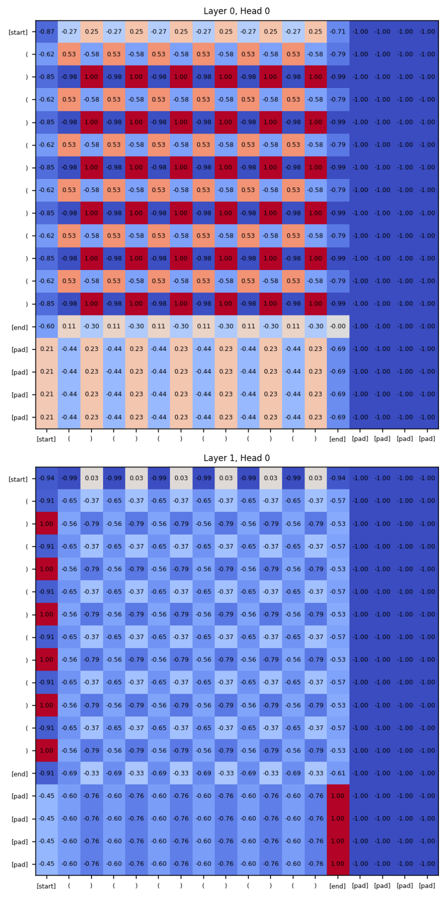
Figure 4.1 shows two examples of attention matrices for the Transformer corresponding to the first experiment in Table 4.2, trained on Dyck-1 samples that achieved perfect (i.e.: 100%) accuracy over train, validation and test datasets, as well as a near-zero loss.

Both sets of attention matrices display a distinct and meaningful pattern, illustrating how certain tokens attend more strongly to specific others, which aligns with the structure of the Dyck-1 language. Specifically, looking at Figure 4.1b, we observe that in Layer 0, the first opening bracket (‘`(`’) primarily attends to the other opening brackets in the sequence. This suggests that the model has learned to recognize the structural importance of matching opening brackets, as it assigns greater attention weight to those tokens. Also, these opening brackets are negatively attending to the closing brackets in Figure 4.1b, which indicates the model is distinguishing between different types of parentheses.

In Figures 4.1a and 4.1b, the closing brackets (‘`)`’) exhibit a strongly positive attention towards the other closing brackets in the sequence, but not to the open-



(a) Attention matrices of string $)((((()(()((\notin$ Dyck-1



(b) Attention matrices of string $()()()()()() \in$ Dyck-1

Figure 4.1: Attention matrices of a Transformer with a pad-token mask trained on Dyck-1 strings

ing brackets. This consistent and structured behavior in Layer 0 highlights the model's understanding of grouping similar tokens, which reflects how it is capturing the hierarchical nature of Dyck-1, where brackets must be correctly matched and nested.

However, in Layer 1, the attention pattern undergoes a transformation. Here, the first opening bracket now attends more strongly to the closing brackets, inverting the order with respect to Layer 0, Head 0, suggesting that the model in this layer has begun learning the relationships necessary for pairing open and close brackets. The first closing bracket, on the other hand, shifts its focus to attend to the opening brackets in the sequence. This inversion of attention patterns in Layer 1 may reflect the model's mechanism for resolving bracket pairs, ensuring that each opening parenthesis finds its corresponding closing counterpart.

This progressive change across layers demonstrates how the transformer is leveraging the attention mechanism to first group similar tokens (Layer 0) and then to learn the associations between these groups (Layer 1), effectively capturing the hierarchical and nested structure of Dyck-1 sequences. This process mirrors the parsing of formal languages, where early stages (lower layers) establish syntactical groupings, while later stages (higher layers) work toward resolving more complex relationships like matching parentheses.

This clear attention pattern is a key factor in the model's ability to achieve perfect accuracy across the train, validation, and test datasets. It highlights how transformers can efficiently learn and represent formal languages by leveraging attention mechanisms across layers to capture both token-level similarities and dependencies across a sequence.

4.1.1.2 Causally masked Transformer

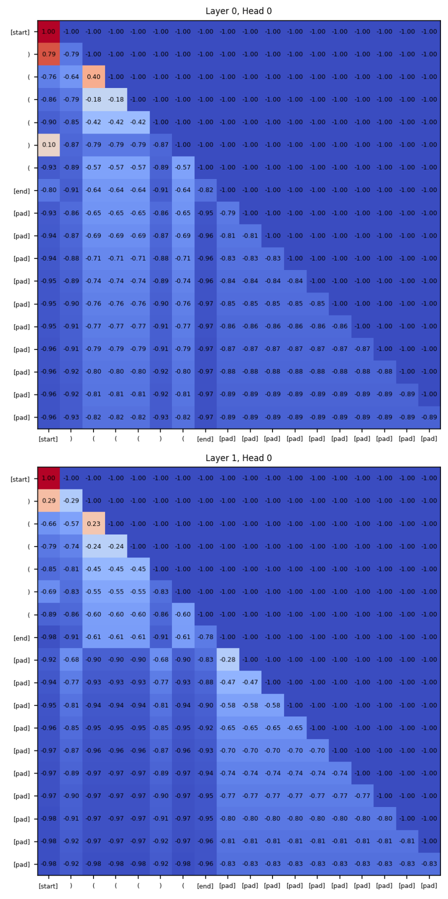
However, if we take a look at the attention matrices generated by the second experiment in Table 4.2, we will see that no clear pattern is generated, as can be seen in Figure 4.2.

In this case, the model can only "see" or attend to tokens behind the current token position. A causal attention mechanism restricts the model to attend only to tokens that have already been processed, avoiding future token information during the current position's attention computation, as defined in Section 3.2.

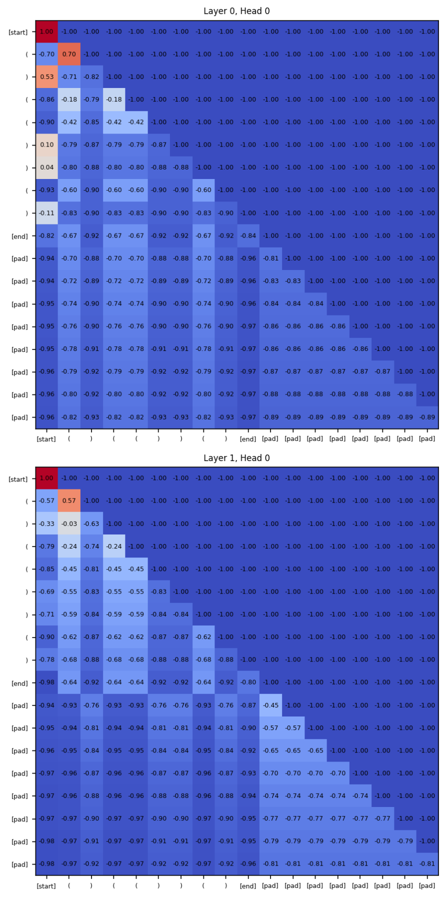
Figure 4.2 shows this limitation leads to less defined attention patterns compared to the first experiment. In Layer 0, Head 0, for instance, the model primarily attends to nearby tokens in the sequence, but with little variation in attention strength. This could indicate that the model is not learning significant dependencies between tokens, or it is being overly biased by the causal restriction, preventing it from fully capturing the structure of the input sequence, such as matching parentheses.

This effect is more pronounced in deeper layers. In Layer 1, Head 0, the attention values are also spread more diffusely, and while some tokens show slightly stronger attention, the general trend reflects a failure to learn structured dependencies, as seen in the minimal variation between attention weights. Furthermore, as we traverse the sequence, attention scores become lower, showing that there is a less of a learned dependency between opening and closing brackets, which is a key part of learning to recognize Dyck-k sequences.

Moreover, there is no distinct pattern like the one observed in the first experiment (refer to Table 4.2) where the model learned the dependency structure



(a) Attention matrices of string $)((()(\notin$ Dyck-1



(b) Attention matrices of string $()(())() \in$ Dyck-1

Figure 4.2: Attention matrices of a Transformer with a causal mask trained on Dyck-1 strings

between opening and closing parentheses more clearly. This lack of a pattern in the attention matrices implies that the model may be struggling with generalizing the Dyck-1 rules, which we attribute to the causal restriction in the attention mechanism.

In this experiment, we never reached an accuracy higher than $\approx 60\%$ across the train, validation and test datasets. Furthermore, the loss across training, validation and evaluation seemed to increase, rather than decrease, as we will see in Section 5.5.3.

In summary, the causal limitation imposed in this second experiment seems to significantly impact the model's ability to learn the intricate patterns required for Dyck-1 language parsing. While the model does attend to previous tokens, the attention matrices do not show the clear patterns needed for correctly pairing parentheses, unlike in the first experiment. This leads to the conclusion, supported by the low training and test accuracies, that causal attention alone may not be sufficient for creating Transformers trainable enough to generalize certain formal languages from limited samples, especially when bidirectional dependencies are crucial, as is the case with Dyck-1.

4.1.2 Experiments on Dyck-3 Classifier Transformers

We will now describe the experiments conducted with Transformers built to classify Dyck-3 languages - these correspond to the third, fourth and fifth experiments in Table 4.2.

4.1.2.1 Bidirectionally masked Transformer

In these experiments, both sets of attention matrices display meaningful patterns, which illustrates how certain tokens in the sequence attend more strongly to specific others, in this case aligning with the structure of the Dyck-3 language.

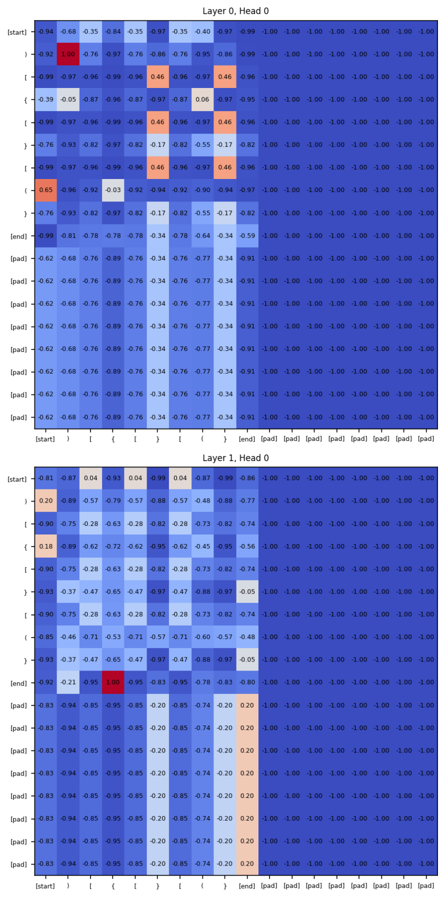
Albeit not as clear as the patterns in the experiments done with Dyck-1 languages, similar behaviours can be seen in the matrices shown in Figure 4.3. This decrease in the clarity of the pattern in the attention matrices was expected as the language became much more complex, however, the Transformer still managed to classify all sequences belonging to the language perfectly across train, validation and test datasets, reaching an accuracy of 100%, with a near-zero loss.

This is similar to the work done by Ebrahimi et al. [24], where the authors empirically show the self-attention mechanism captures these dependencies without the need for recursion or positional encodings.

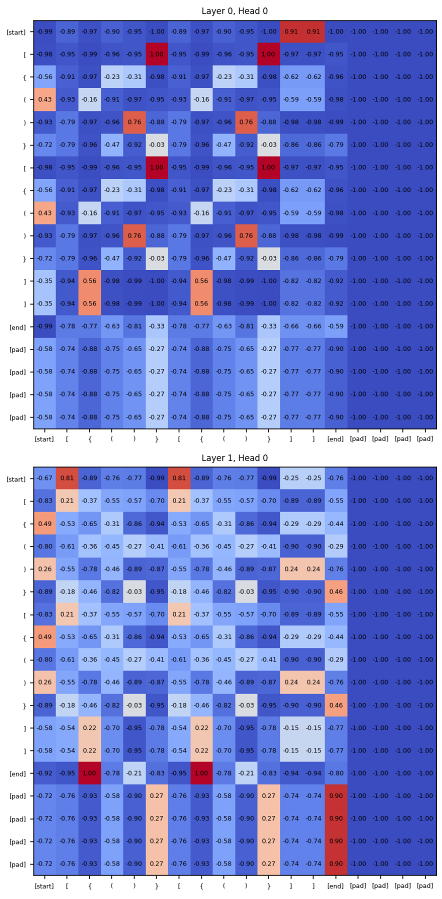
Even though these matrices do not have a pattern as clear as those in Figure 4.1, we still can see elements that indicate the Transformer has learned relations between tokens in the sequences. If we look at the attention of `[start]` with respect to the rest of the sequence in Figures 4.3a and 4.3b, we can see a similar pattern to the one observed in the experiments carried out for Dyck-1 languages.

In the negative example, the attention values for this row never reach positive values, whereas for the positive example, the last element of the first row in the attention matrix reached a strongly positive attention value.

Also, the attention of the `[pad]` tokens with respect to `[end]` (i.e.: the column for the `[end]` token) shows a pattern similar to that of Dyck-1, as the attention values for the positive examples is higher than for the negative ones.



(a) Attention matrices of string `)[{[}[(}` $\notin$ Dyck-3



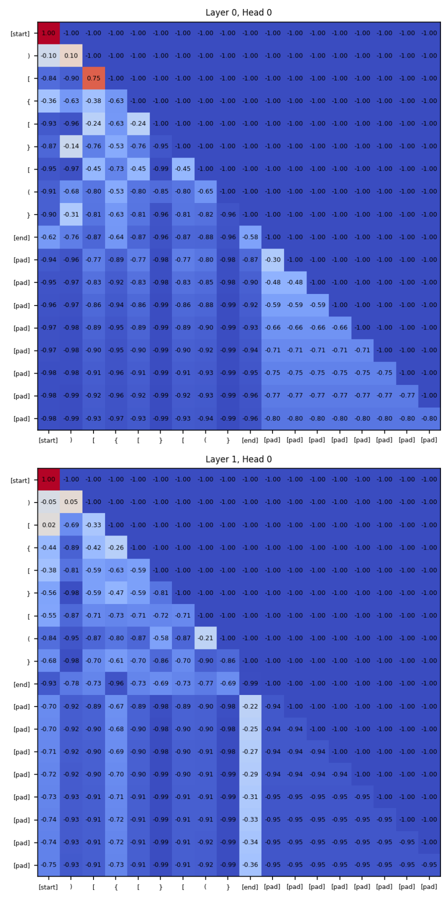
(b) Attention matrices of string `[{()}[{()}]]` $\in$ Dyck-3

Figure 4.3: Attention matrices of a Transformer with a pad-token mask trained on Dyck-3 strings

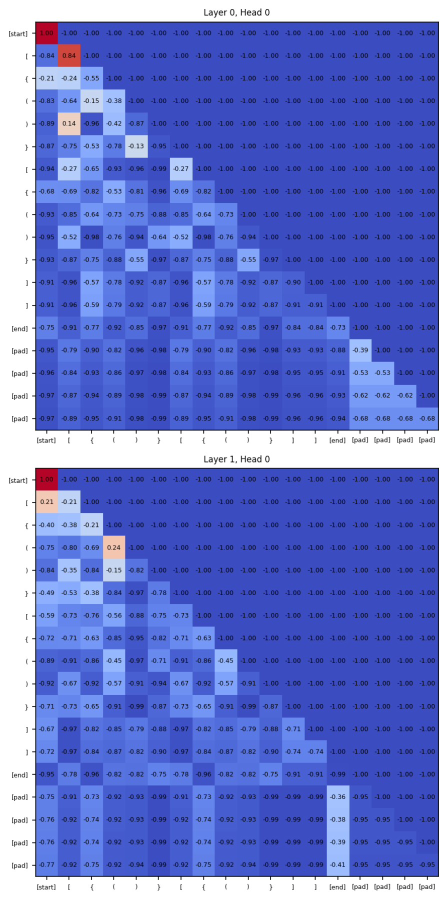
We can hypothesize that even for more complex languages such as Dyck-3, the bidirectional attention mechanism is capable of effectively capturing the hierarchical dependencies and structures present in these languages, even without considering the position of the tokens themselves.

4.1.2.2 Causally masked Transformer

Once more, when the Transformer is trained with a causal mask rather than a bidirectional one, we find that not only the patterns in the attention matrices are lost, but also the model does not achieve generalization and near-perfect performance over the data. For Dyck-3 sequences, we found the effect of the mask to be even more pronounced, with variations of attention scores dropping drastically with the token position, in essence, “losing” information on whether the string up to that point was balanced or not.



(a) Attention matrices of string `)[{[}[(}` $\notin$ Dyck-3



(b) Attention matrices of string `[{()}[{()}]]` $\in$ Dyck-3

Figure 4.4: Attention matrices of a Transformer with a causal mask trained on Dyck-3 strings

Let us look at Figure 4.4b, more specifically at Layer 0, Head 0, where attention values at the start of the sequence are much higher than attention values at the end of the sequence, where they tend to be much nearer the (normalized) minimum value of -1. For example, if we consider the attention of the first token with respect to itself, we see a much higher attention value:

$$\mathrm{Attn}_{0,0}(\texttt{[}, \texttt{[}) = 0.84 \tag{4.1}$$

However, if we consider tokens towards the end of the word, such as the attention of the last token with respect to itself, we see it is much lower, and does not follow the pattern seen in bidirectionally masked Transformers:

$$\mathrm{Attn}_{0,0}(\texttt{]},\texttt{]}) = -0.91 \tag{4.2}$$

We attribute this to the causal restriction imposed by the mask, which limits the ability of the model to "see", generate internal pairings between brackets, and thus train and generalize to recognize Dyck-3 strings from limited samples.

4.1.3 Out-of-distribution experiments

In this experiment, we investigate whether Transformer classifiers can maintain their accuracy when processing sequences longer than those they were trained on.

We trained a 2-layer Transformer model with soft attention, no positional encoding, and bidirectional masking on sequences from the Dyck-3 language. Our previous results already showed this model can generalize well, achieving perfect classification of the Dyck-3 language with a limited number of training samples. The training sequences had lengths ranging from 0 to 96 tokens. As shown in Table 4.2, the model's context window was set to 128 tokens, meaning it was theoretically capable of handling sequences up to 128 tokens in length. During training, the model quickly reached 100% accuracy and near-zero loss, which is consistent with the results observed in previous experiments.

For evaluation, we used a different dataset containing Dyck-3 sequences with lengths ranging from 32 to 128 tokens. Given the model's (near) perfect performance on both the training and validation sets, we expected it to perform well on the test dataset. However, the model only achieved an accuracy of $\approx 75\%$. We can visualize this in Figure 4.5.

This outcome suggests that the model struggles with sequences longer than 96 tokens, despite having a larger context window of 128 tokens. Since $\approx 75\%$ accuracy could correspond to correctly classifying test sequences up to 96 tokens long, assuming a uniform distribution of sequence lengths, we hypothesize that Transformer models can only effectively process sequences up to the maximum length they were trained on, even if their context window supports longer sequences.

Therefore, we found an important limitation of these models, since even though Transformers can generalize across different sequence lengths, their ability to classify sequences accurately is bounded by the lengths present in the training data. In essence, for the model to perform well on longer sequences, the training dataset must include samples that cover the full range of sequence lengths up to the desired maximum.

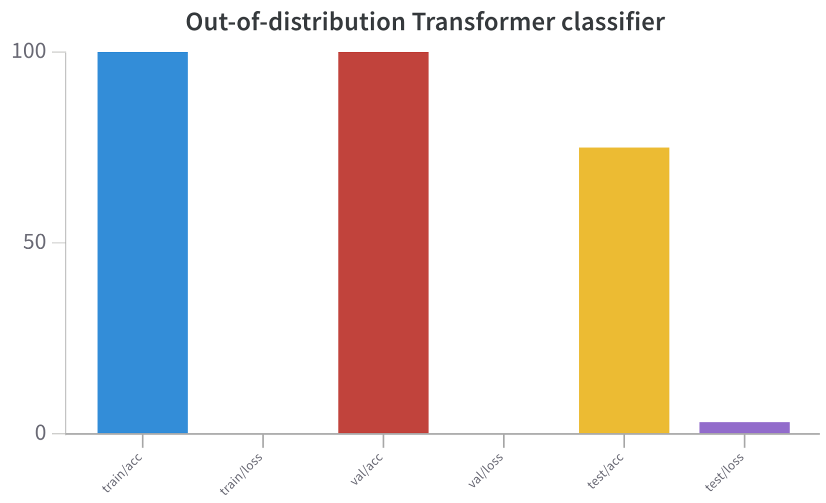


Figure 4.5: Accuracy/Loss for Out-of-distribution Transformer classifier

4.1.4 Long-context Transformer experiments

In our final experiment, we aimed to assess whether these models could accurately classify longer Dyck-3 strings, testing their ability to handle significantly longer sequences. We conducted this experiment using sequences ranging in length from 0 to 4096, which allowed us to push the boundaries of sequence length beyond what had been tested in previous experiments. The model we used was a 2-layer Transformer with soft attention, no positional encoding, and bidirectional masking, as in earlier setups.

The model demonstrated rapid learning, achieving an accuracy of 100% and a near-zero loss in a short amount of time, similar to our previous experiments. This quick convergence suggests that Transformer models do not struggle with recognizing Dyck-k strings, even as sequence lengths increase significantly. In fact, sequence length does not appear to pose a challenge, as long as the sequences fall within the range of lengths seen during training.

These results indicate that the model's architecture, particularly its use of bidirectional masking and soft attention, is well-suited to the task of Dyck string classification, handling long sequences just as effectively as shorter ones.

Due to the sheer size and complexity of the attention matrices, which in this experiment are 4096×4096, we found it nigh impossible to visualize any meaningful attention patterns. The vast amount of information stored in these matrices makes it extremely difficult to interpret or identify specific trends or behaviors within the attention mechanism. Each matrix element represents the attention weight between every pair of tokens in the sequence, leading to over 16 million individual data points per matrix.

As a result, the scale of the data makes visual inspection nearly impossible, and even traditional visualization techniques, such as the heatmaps used in previous experiments, become ineffective due to the dense and intricate nature of the attention distribution. This highlights one of the limitations of working with very long sequences in Transformer models, where the complexity of the attention mechanism grows quadratically with sequence length, making it increasingly harder to analyze and understand the model's internal workings as the sequence length increases.

4.1.5 General observations

In our experiments, we found that while the Transformer architecture described by Ströbl et al. [6] is theoretically expressive enough to recognize this family of languages, it may lack sufficient trainability.

However, a Transformer composed by 2 encoder layers (or blocks) with soft attention, pad-token masking, no positional encoding, layer normalization, and residual connections proved to be consistently trainable enough to not only recognize but also generalize effectively to both the training and test datasets. This modified architecture achieved 100% accuracy and a loss of 0 on both sets, at least for sequences of a given length, for Dyck-k languages (with different values of k).

This corresponds with results achieved by Yao et al. [18], in which they empirically found Transformers are able to *learn* (through training) and achieve good performance on finite samples of these languages. This highlights the difference between *expressivity* and *trainability.*

However, when trying to train the model described by the authors (which corresponds to Experiment 5 in Table 4.2), we found that it does not generalize from limited samples, but instead falls into the same pitfalls our causally masked Transformers (Experiments 2 and 4) had during training, where none could surpass an accuracy of $\approx 60\%$.

Also, we found that the self-attention mechanism is capable enough to capture the hierarchical dependencies between opening and closing brackets in Dyck-k languages, without taking the position of the token into account or relying on recursive structures, as proposed by Ebrahimi et al. [24]. We find it worth noting that the repeated values in the attention matrices of bidirectionally masked Transformers (Figures 4.1 and 4.3) can be attributed to the lack of positional encoders in these models.

It is of interest to note that we did observe some examples where a Transformer with only 1 encoder block (or layer), with the aforementioned components, was able to be trained to recognize the training and testing dataset with 100% accuracy, however, these results were sporadic and not consistently reproducible over multiple runs.

4.2 Explainability and Interpretability

With regards to the explainability of Transformers, we have already discussed the patterns present and absent in the attention matrices of the models trained in our experiments, however, a discussion is needed as to what these patterns may convey.

Explainability, as previously defined, is the capacity to answer "wh" ("why", "what", etc.) or "how" questions about a model's particular solution, be it classification, regression, object detection or another task [5].

Furthermore, we will tackle explainability using an approach rooted in *mechanistic interpretability* [10, 11], which can be likened to reverse-engineering a neural network by looking at its internal components, in our case, attention matrices and neural activations.

We have already seen some of the internal workings and limitations of these models by visualizing their attention matrices, in essence, peeking into the inner workings of the key mechanism of a Transformer model - we have tested the influence of masking on the accuracy of the Transformer on the classification task, which leads us to believe the causal restriction is too strong to allow for accurate classification of Dyck-k sequences.

By examining the attention matrices in our Transformers, we can sometimes visualize clear and defined patterns that offer insights into how the model classifies sequences, particularly those belonging to a Dyck-k language. If we were to look at Layer 0, Head 0 in Figures 4.1a and Figure 4.1b, we are able to see a key difference, apart from the pattern - the value of the attention of the `[end]` token with itself. In the positive example, this value is much higher compared to the negative example, which leads us to believe the attention matrix may hold a *preview* into the model's prediction.

The key word here is *preview*, as this value is not an output the end user sees, but rather an extra piece of information the model may use to output a probability distribution that indicates membership of a sequence to a Dyck-k language.

We also explored the neural activations and weights of various components in our models, but did not observe any meaningful patterns. These values were automatically logged using Weights & Biases, a tool for tracking ML/AI experiments [25].

In Figure 4.6, we present the weights of the fully connected layer from the

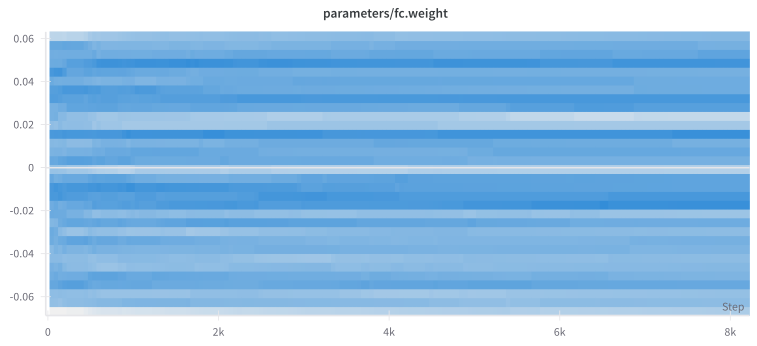


Figure 4.6: Fully-connected layer weights (Bidirectionally-masked Transformer)

model used in Experiment 1 (Table 4.2) across training, validation, and testing. The figure illustrates how the weights of the fully connected layer evolve during training—essentially presenting a top-down view of a histogram. If we were to take vertical slices from this figure, we would get a traditional histogram of the weights at training step i.

Since we do not see any significant changes during the training process, we hypothesize the model is not relying heavily on this component to refine or adjust its internal representations over time. If this were to be the case, we would be able to visualize a meaningful pattern or shift in weight values, to correlate with the model learning features from the data. In this case, however, the weights remain nearly static throughout, implying the model is much more reliant on other mechanisms, such as self-attention, rather than the fully connected layer. Consequently, visualizing this layer provides little information regarding explainability or interpretability, making it minimally useful to analyze to gain insights into the model's decision making process.

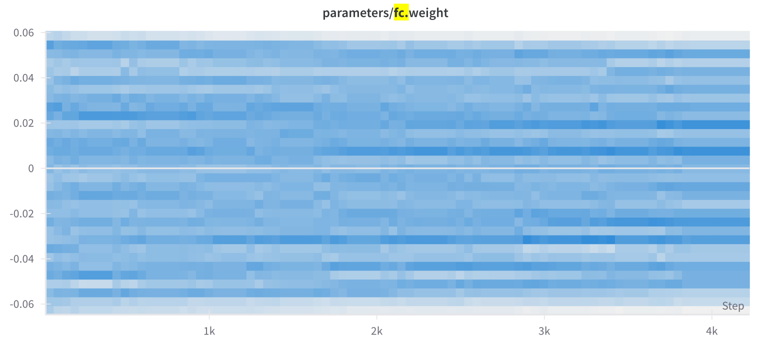


Figure 4.7: Fully-connected layer weights (Causally-masked Transformer)

This is supported by the fact that a very similar pattern is observed in the weights of the fully-connected layer of a casually-masked Transformer, as can be seen in Figure 4.7, leading us to believe that analyzing this component and its weights does not provide information on the model's decision. Instead, it suggests that other components within the model hold more valuable information regarding the model's decision. The fully connected layer appears to function primarily as an output mechanism, mapping the model's internal decisions to a probability distribution.

Our observations are similar to Ebrahimi et al. [24], where they found self-attention networks with bidirectional masking leverage the attention mechanism in such a way that it can replace the recursion needed to learn hierarchical languages such as Dyck-k, with an intelligent representation of the head or start token.

5 Technical Implementation

This chapter will discuss the technical aspects involved in the implementation of our Transformer models for classification of Dyck-k sequences. We will cover the dataset generation process, the tokenizer's design and implementation of the Transformer models. We will also discuss the package structure, dependency management and specific tools and libraries used.

5.1 Dataset

We built a dataset generator, which allows us to generate n samples of a Dyck-k language with a certain distribution.

Balanced Strings

```python
def _generate_balanced_string(order: int, length: int, seed: int = 42) -> str:
    """Generate a string of `length` from the Dyck language of `order`.
    Args:
        order (int): The order of the Dyck language.
        length (int): The length of the string to generate.
        seed (int): The seed for the random number generator.
    Returns:
        str: A string of `length` from the Dyck language of `order`."""

    random.seed(seed)

    if length == 0:
        return ""

    length = length if length % 2 == 0 else length + 1

    stack = []
    word = ""

    brackets = [(k, v) for k, v in list(c.BRACKETS.items())[:order]]

    half_length = length // 2
    first_half = last_half = 0

    while first_half + last_half < length:
        if first_half < half_length and (len(stack) == 0 or random.random() <
    0.5):
            opening_bracket, closing_bracket = random.choice(brackets)
            stack.append(closing_bracket)
            first_half += 1
            word += opening_bracket
        else:
            bracket = stack.pop()
            last_half += 1
            word += bracket

    return word
```

Algorithm 5.1: Generate balanced string

We select a subset of k pairs of opening and closing brackets from all our available brackets, which will be our alphabet.

Our algorithm to generate a Dyck-k word uses a stack, in which closing brackets are added while the amount of opening brackets is less than the length of half the word, the stack is empty or a random number from 0 to 1 is less than 0.5. We also add the opening bracket to the word we are generating

If these conditions are not met, we add 1 to the counter of closing brackets, add the closing bracket to the word we are generating and pop the bracket from the stack.

This process is repeated while the length of the word is less than the desired length.

Unbalanced Strings

```
    """Generate a string of ‘length‘ that is not necessarily from the Dyck language of ‘order‘.
    Args:
        order (int): The order of the Dyck language.
        length (int): The length of the string to generate.
        seed (int): The seed for the random number generator.
    Returns:
        str: A string of ‘length‘ that is not necessarily from the Dyck language of ‘order‘."""

    random.seed(seed)

    word = ""

    opening_brackets = [k for k, _ in list(c.BRACKETS.items())[:order]]
    closing_brackets = [v for _, v in list(c.BRACKETS.items())[:order]]

    brackets = opening_brackets + closing_brackets

    first_char = random.choice(opening_brackets) if random.random() < 0.5 else random.choice(closing_brackets)
    word += first_char

    random_brackets = [random.choice(brackets) for _ in range(length - 1)]
    random.shuffle(random_brackets)
    unbalanced_str = word + "".join(random_brackets)

    if checker.is_dyck_word(unbalanced_str, order):
        del unbalanced_str
        del brackets
        return _generate_unbalanced_string(order, length)

    return unbalanced_str
```
Algorithm 5.2: Generate unbalanced string

We select a subset of k pairs of opening and closing brackets from all our available brackets, and then convert it into a list for easier selection. We also create lists of opening and closing brackets.

We first select the first character, which can be either an opening or closing bracket with $p = 0.5$.

We then select a random item from the list until we generate a word with the desired length, and check whether the generated string is balanced - since the process to generate this sample is random, we may generate a balanced string. In this case, we discard the result (to optimize memory usage) and recursively call our function to generate a new unbalanced string.

In both algorithms, we define a seed for our random number generator, in order to guarantee reproducible results.

Finally, we use these functions to generate a dataset composed by n samples with probability p of being balanced (i.e.: $n \times p$ samples are balanced, $n \times (1 - p)$ samples are unbalanced). It is useful to note that this dataset class inherits from `torch.utils.data.Dataset`, which will be of use later on when we are training and evaluating our Transformers, as it allows us to easily split the dataset into subsets used for training, validation and evaluation (our train-val-test split) and use `DataLoaders` to easily and appropriately batch our data for training.

5.2 Tokenizer

We built a tokenizer that maps our sequences from $\Sigma^* \to \mathbb{R}^{|\Sigma|+3}$. In essence, we convert our sequence of parentheses to a sequence of numbers, mapping each parenthesis to a number and adding 3 special tokens to define the start and end of the sequence, as well as any padding necessary to unify the sequence lengths. This tokenizer partially conforms to HuggingFace's tokenizer specification [26], specifically exposing the `.decode()` and `tokenize()` methods.

We find it important to highlight the `tokenize` method in the implementation, which can be seen in Annex 8.1, as this was done using `numpy` to allow for efficient, vectorized tokenization of large datasets, which helped drastically reduce time taken to build datasets.

5.3 Transformers

We now dive into the practical implementation of our Transformers. The practical implementation of these models was done using Pytorch [27], for ease of training using a GPU. We also implemented a way to easily define different Transformer architectures, through our `TransformerClassifierConfig` class. Also, we highlight device-specific nuances in Pytorch, that were considered in order to avoid issues.

5.3.1 TransformerClassifierConfig

In order to make the process of creating different Transformers easier, we defined a class `TransformerClassifierConfig`, which defines the model's architecture - i.e.: the context length, the hidden dimensions, the number of attention heads and all other relevant architectural parameters.

```
class TransformerClassifierConfig:
    def __init__(self, vocab_size, d_model, n_heads, dim_ff, n_layers,
    n_classes, max_seq_len):
        self.vocab_size = vocab_size + 3
        self.d_model = d_model
        self.n_heads = n_heads
        self.dim_ff = dim_ff
        self.n_layers = n_layers
        self.n_classes = n_classes
        self.max_seq_len = max_seq_len + 2
```

Algorithm 5.3: TransformerClassifierConfig definition

We find it useful to note that `self.vocab_size` is defined as `vocab_size + 3` in order to consider the three special tokens present in our vocabulary: `[pad]`, `[start]` and `[end]`. Analogously, `self.max_seq_len` is defined as `max_seq_len + 2` for the same reason.

As the purpose of these Transformers is to classify, we provide a way for users to employ these Transformers for either binary or multi-class classification problems.

5.3.2 TransformerClassifier

In this section we will present particular aspects of our implementation that are of use or interesting to readers from an engineering perspective, as other parts of the implementation do not deviate significantly from a typical Pytorch implementation of a Transformer encoder.

We find it of interest to discuss differences in implementation based on the device used to store in memory and then train the Transformer. In Pytorch, we must have both the model and dataset on the same device, which can be `cpu`, `cuda` (if we are using an NVIDIA GPU) or `mps` (if we are using Apple Silicon). However, behaviour of functions may differ based on the selected device - we provide an example below.

```
if device.startswith("cuda") or device == "cpu":
    preds = torch.argmax(predictions, dim=1)
elif device == "mps":
    _, preds = predictions.max(1)
```

If we were to use `torch.argmax` on `mps`, we would see that the behaviour is erratic and will not provide us with the expected result, instead, we must use `predictions.max()` and specify we want to reduce over the first dimension by specifying `dim=1`. This returns a tuple of `(values, indices)`, where indices is the index location of each maximum value found (argmax) [28].

5.4 Package Architecture and Dependency Management

The project's codebase was architected in such a way that it can easily extend the model explainability tool already developed by the Universidad ORT Uruguay's AI research group. We kept external dependencies to a minimum, limiting ourselves to only the strictly necessary. This was done in pursuit of an easier integration with the already existing codebase. Furthermore, the codebase was developed using Python 3.10. We used Github for version control, and the repository with the codebase is public and accessible at `https://github.com/matiasmolinolo/transformer-checker`.

We can see the package diagram in Figure 5.1, which shows multiple packages, split by responsibilities, which will be detailed below. A class diagram is presented in Annex 8.2.

The `transformer` package contains all the modules necessary to build a TransformerClassifier and a TransformerClassifierConfig. Every module that makes up a TransformerClassifier inherits from `nn.Module`, for easy training of our models using Pytorch.

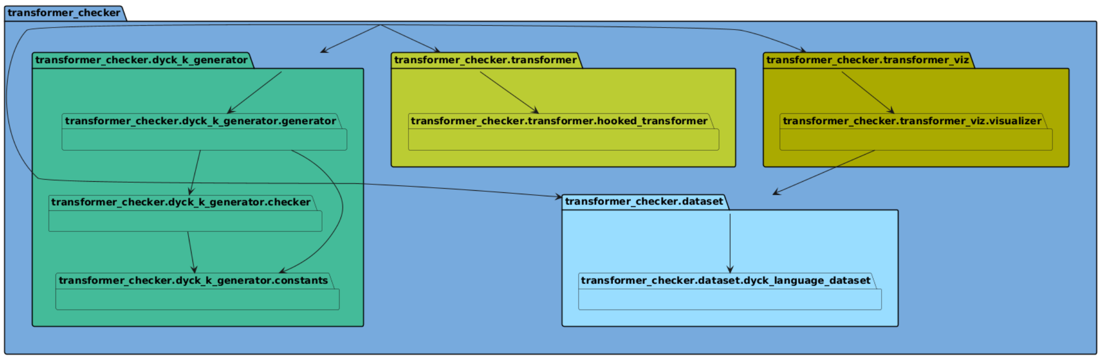

transformer_checker
transformer_checker.dyck_k_generator
transformer_checker.transformer
transformer_checker.transformer_viz
transformer_checker.dyck_k_generator.generator
transformer_checker.transformer.hooked_transformer
transformer_checker.transformer_viz.visualizer
transformer_checker.dyck_k_generator.checker
transformer_checker.dataset
transformer_checker.dyck_k_generator.constants
transformer_checker.dataset.dyck_language_dataset

Figure 5.1: `transformer-checker` package diagram

Secondly, the `dataset` package contains both the tokenizer and Pytorch Dataset definitions, which are used to build Dyck-k language datasets that can be easily used to train our models.

The `dyck_k_generator` package contains an algorithmic checker to validate whether a string belongs to a Dyck-k language or not, the raw data generator that builds a JSON Lines file (`.jsonl`) with our labeled samples and the required constants (i.e.: our alphabet).

Finally, the `transformer_viz` package provides helpful methods to visualize the attention matrices extracted from our TransformerClassifier models.

Dependencies between packages are kept to a minimum to avoid breaking changes in one package affecting others. This kept the development speed agile, as most changes in one package did not affect other packages and therefore, development could continue. The only dependency is that the `transformer_viz` package uses the `dataset` package to access the tokenizer and create an instance of it, in order to decode the batch tokens back to strings, to label the plot axes.

We used `poetry` [29] to manage dependencies, as it manages dependencies more cleanly and consistently than `pip` and allows us to easily publish a new package version to PyPI, for easy sharing with other teams outside Universidad ORT Uruguay's AI research group. We automated publishing a new package version using Github Actions, with a workflow that triggers when a new release is created, which automatically bumps the project version to the one defined in the aforementioned tag and updates the code in Github, and then publishes the release to PyPI, where it can be accessed via `pip`, `poetry` or the PyPI web repository (`https://pypi.org/project/transformer-checker/`).

Furthermore, we follow Pythonic conventions, making each folder a module by adding an `__init__.py` file, which allows the contents of the folder to be imported across the codebase and for easy access to functionalities in other projects.

Specifically, the dependencies used were: `torch==2.3.0`, `matplotlib==3.8.4`, `tqdm==4.66.4` and `wandb==0.17.1`

5.5 Training and Evaluation

We will now discuss the methodology used to train and evaluate our Transformer models. We will discuss how we split our data into training, validation and test datasets, the hardware used, how we logged metrics and artifacts, the hyperparameters used and our obtained metrics.

5.5.1 Training Data and Batching

Training and test data was synthetically generated using the methods described in section 5.1. This allowed us to build several datasets with different characteristics, which were then used for training different Transformers. This data was then batched using a Pytorch DataLoader, using different batch sizes for training and testing. For example, Experiment 1 (as defined in Table 4.2) used batch sizes of 32, 8 and 4 for training, validation and testing, respectively.

```python
from torch.utils.data import DataLoader

train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=8, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=4, shuffle=True)
```

Algorithm 5.4: DataLoader definition for Experiment 1

We use `shuffle=True` to introduce randomness into the training process, in order to prevent the model from memorizing the order in which the sequences are presented. By shuffling the data, we encourage the model to focus on learning meaningful patterns within the sequences rather than overfitting to the specific order of the training data. This approach ultimately enhances the model's ability to generalize to new, unseen data.

5.5.2 Hardware and Logging

We trained these models locally and logged the training artifacts using Weights & Biases [25], which allowed us to capture key values such as the weights and biases of the models' internal components. The training was conducted on two systems: one equipped with an NVIDIA RTX 3060 GPU with 12 GB of VRAM and 64 GB of RAM, running Ubuntu 22.04, and the other on an M3 MacBook Pro with 8 GB of unified memory, running macOS Sonoma 14.5.

5.5.3 Hyperparameters and Metrics

We will now discuss the hyperparameters and criteria used when training the models. We can see the hyperparameter and criteria selection in Table 5.1 for the experiments detailed in Table 4.2.

	lr	Epochs	Optimizer	Criteria
1	1×10^{-5}	20	Adam	Cross-entropy Loss
2	1×10^{-4}	10		
3	1×10^{-5}	15		
4	1×10^{-5}	15		
5	1×10^{-5}	15		
6	1×10^{-5}	25		
7	1×10^{-5}	100		

Table 5.1: Training hyperparameters and criteria

Table 5.1 shows the learning rates (lr) and the number of epochs used during the training of our experiments. All models were trained using the Adam optimizer [30] and evaluated with the cross-entropy loss function, which is commonly used for classification tasks.

	Train		Validation		Test	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
1	**100.0**	6×10^{-5}	**100.0**	1×10^{-5}	**100.0**	1×10^{-5}
2	50.425	*0.7087*	*49.42*	0.6860	52.25	0.7005
3	100.0	2×10^{-5}	100.0	1×10^{-5}	100.0	1×10^{-5}
4	49.62	0.6875	50.61	0.6933	*49.92*	0.6891
5	*49.47*	0.6725	49.72	*0.6941*	50.58	0.6941
6	100.0	$\mathbf{1 \times 10^{-5}}$	100.0	**0.0000**	74.98	*3.0529*
7	100.0	2×10^{-5}	100.0	0.0000	100.0	**0.0000**

Table 5.2: Experimental results

Table 5.2 presents the accuracy and loss metrics for the experiments described in Table 4.2. The highest metrics achieved are highlighted in bold, while the lowest are italicized.

6 Conclusions and Future Work

Throughout this work, we carried out these experiments that focused on interpretability and explainability of language models, more specifically, Transformer classifiers. We discussed formal languages and the Chomsky Hierarchy, context-free languages such as Dyck-k and Shuffle-Dyck-k, the Transformer architecture and its components, the state-of-the-art of using these models to classify sequences belonging to a formal language and the engineering process behind the development of the `transformer-checker` tool. We discussed how expressive, trainable and interpretable these models are, through the lens of *mechanistic interpretability*, which can be likened to seeing the execution of a program and trying to reverse-engineer it. In our case, we looked at the model's internal components (attention matrices, weights) and tried to draw conclusions on the model's decision from these components.

We conducted a series of experiments using Transformer classifiers to determine whether sequences belong to a context-free language, such as Dyck-k. We found these models to be computationally sufficient for this task, reaching perfect accuracies under certain conditions. We found that sequence length does not hinder the performance of these models, as we tested sequence lengths of up to 4096. We did find, however, an important limitation of these models - their ability to classify sequences accurately is bounded by the lengths present in the training data.

Through these experiments, we discovered that the mask applied to the input sequences plays a pivotal role in the model's ability to learn effectively. The correct use of masking ensures that the model focuses on the appropriate parts of the sequence, which is critical for learning the hierarchical and nested structure characteristic of Dyck-k languages.

Additionally, we found that the self-attention mechanism within the Transformer is central to the model's decision-making process. The attention mechanism appears to capture important dependencies between tokens, allowing the model to learn which parts of the sequence relate to each other, when a pad-token mask is used. Specifically, self-attention helps the model track the relationships between opening and closing symbols, which is essential for determining membership in Dyck-k languages. Thus, the information encoded in the self-attention weights provides valuable insight into how the model arrives at its classification decisions.

We were able to visualize this by extracting and plotting the attention matrices of these models, which gave us an intuitive way of looking inside the Transformer and making sense of its output, effectively switching the black-box approach that is more commonly used with language models for a white-box approach.

Furthermore, we explored the internal workings of other components in the models, such as the weights of the feed-forward layer responsible of outputting the classification probability distribution and found no useful information or patterns that could help clarify the model's decision process.

We compared our results with those obtained by Bhattamistra et al. [19], Ströbl et al. [6], Yao et al. [18] and Ebrahimi et al. [24] and found similarities, differences and limitations on the experiments conducted by the authors, aside from the findings described above. Firstly, we found that albeit causally-masked Transformers may be *expressive* enough to classify Dyck-k languages, they are not *trainable* enough to do this consistently and with perfect accuracy. Secondly, we found that one-layer Transformers do not consistently achieve perfect accuracy on this task, and that a minimum of two layers are required to consistently achieve 100% accuracy on Dyck-k sequence classification.

We developed `transformer-checker`, a tool that can be easily integrated into `neuralchecker`, the existing explainability tool developed by the Universidad ORT Uruguay's AI research group. Our tool expands the capability of `neuralchecker` by providing a way to analyze complex models, such as Transformers, with a novel, white-box approach.

Regarding future work, we look forward to combining this approach with automata extraction, similar to the work done by Weiss et al. [31], Carrasco et al. [32] and Mayr et al. [33], as well as working with sparse autoencoders to extract interpretable features [34, 35], in order to be able to work with larger models and work on problems larger than toy examples of interpretability of language models for formal languages.

7 Bibliography

[1] OpenAI, Nov 2022. [Online]. Available: https://openai.com/index/chatgpt

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[3] J. Simon, "Large language models: A new moore's law?" Oct 2021. [Online]. Available: https://huggingface.co/blog/large-language-models

[4] T. Lei, R. Barzilay, and T. Jaakkola, "Rationalizing neural predictions," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, J. Su, K. Duh, and X. Carreras, Eds. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 107–117. [Online]. Available: https://aclanthology.org/D16-1011

[5] P. Gohel, P. Singh, and M. Mohanty, "Explainable ai: current status and future directions," 2021. [Online]. Available: https://arxiv.org/abs/2107.07045

[6] L. Ströbl, W. Merrill, G. Weiss, D. Chiang, and D. Angluin, "What formal languages can transformers express? a survey," 2024. [Online]. Available: https://arxiv.org/abs/2311.00208

[7] N. Chomsky, "On certain formal properties of grammars," *Information and Control*, vol. 2, no. 2, pp. 137–167, 1959. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019995859903626

[8] N. Chomsky and M. Schützenberger, "The algebraic theory of context-free languages*," in *Computer Programming and Formal Systems*, ser. Studies in Logic and the Foundations of Mathematics, P. Braffort and D. Hirschberg, Eds. Elsevier, 1963, vol. 35, pp. 118–161. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0049237X08720238

[9] J. Pérez, P. Barceló, and J. Marinkovic, "Attention is turing-complete," *Journal of Machine Learning Research*, vol. 22, no. 75, pp. 1–35, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-302.html

[10] N. Elhage, T. Hume, C. Olsson, N. Schiefer, T. Henighan, S. Kravec, Z. Hatfield-Dodds, R. Lasenby, D. Drain, C. Chen, and et al., "Toy models of superposition," Sep 2022. [Online]. Available: https://transformer-circuits.pub/2022/toy_model/index.html

[11] L. Bereska and E. Gavves, "Mechanistic interpretability for ai safety – a review," 2024. [Online]. Available: https://arxiv.org/abs/2404.14082

[12] A. Mateescu and A. Salomaa, *Formal Languages: an Introduction and a Synopsis.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 1–39. [Online]. Available: https://doi.org/10.1007/978-3-642-59136-5_1

[13] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[14] P. Fletcher, H. Hoyle, and C. W. Patty, *Foundations of discrete mathematics.* Florence, KY: Brooks/Cole, Nov. 1990.

[15] J.-M. Autebert, J. Berstel, and L. Boasson, *Context-Free Languages and Pushdown Automata.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 111–174. [Online]. Available: https://doi.org/10.1007/978-3-642-59136-5_3

[16] J. van Leeuwen, Ed., *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics.* Elsevier and MIT Press, 1990. [Online]. Available: https://www.sciencedirect.com/book/9780444880741/formal-models-and-semantics

[17] G. Rozenberg and A. Salomaa, *Handbook of Formal Languages: Volume 1. Word, Language, Grammar*, ser. Handbook of Formal Languages. Springer, 1997. [Online]. Available: https://books.google.com.uy/books?id=yQ59ojndUt4C

[18] S. Yao, B. Peng, C. Papadimitriou, and K. Narasimhan, "Self-attention networks can process bounded hierarchical languages," 2023. [Online]. Available: https://arxiv.org/abs/2105.11115

[19] S. Bhattamishra, K. Ahuja, and N. Goyal, "On the ability and limitations of transformers to recognize formal languages," 2020. [Online]. Available: https://arxiv.org/abs/2009.11264

[20] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2016. [Online]. Available: https://arxiv.org/abs/1409.0473

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019. [Online]. Available: https://arxiv.org/abs/1810.04805

[22] M. Hahn, “Theoretical Limitations of Self-Attention in Neural Sequence Models,” *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 156–171, 01 2020. [Online]. Available: https://doi.org/10.1162/tacl_a_00306

[23] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[24] J. Ebrahimi, D. Gelda, and W. Zhang, “How can self-attention networks recognize Dyck-n languages?” in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 4301–4306. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.384

[25] L. Biewald, “Experiment tracking with weights and biases,” 2020, software available from wandb.com. [Online]. Available: https://www.wandb.com/

[26] A. Moi and N. Patry, “HuggingFace’s Tokenizers,” April 2023. [Online]. Available: https://github.com/huggingface/tokenizers

[27] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, and S. Chintala, “PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation,” in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS ’24)*. ACM, Apr. 2024. [Online]. Available: https://pytorch.org/assets/pytorch2-2.pdf

[28] Pytorch Foundation and Pytorch Contributors, “torch.max; PyTorch 2.4 documentation - pytorch.org,” 2023. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.max.html

[29] S. Eustace and The Poetry contributors, “Poetry: Python packaging and dependency management made easy.” [Online]. Available: https://github.com/python-poetry/poetry

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017. [Online]. Available: https://arxiv.org/abs/1412.6980

[31] G. Weiss, Y. Goldberg, and E. Yahav, "Extracting automata from recurrent neural networks using queries and counterexamples (extended version)," *Machine Learning*, vol. 113, 06 2022.

[32] M. Carrasco, F. Mayr, S. Yovine, J. Kidd, M. Iturbide, J. P. da Silva, and A. Garat, "Analyzing constrained llm through pdfa-learning," 2024. [Online]. Available: https://arxiv.org/abs/2406.08269

[33] F. Mayr, S. Yovine, F. Pan, N. Basset, and T. Dang, "Towards efficient active learning of pdfa," 2022. [Online]. Available: https://arxiv.org/abs/2206.09004

[34] H. Cunningham, A. Ewart, L. Riggs, R. Huben, and L. Sharkey, "Sparse autoencoders find highly interpretable features in language models," 2023. [Online]. Available: https://arxiv.org/abs/2309.08600

[35] T. Bricken, A. Templeton, J. Batson, B. Chen, A. Jermyn, T. Conerly, N. Turner, C. Anil, C. Denison, A. Askell, R. Lasenby, Y. Wu, S. Kravec, N. Schiefer, T. Maxwell, N. Joseph, Z. Hatfield-Dodds, A. Tamkin, K. Nguyen, B. McLean, J. E. Burke, T. Hume, S. Carter, T. Henighan, and C. Olah, "Towards monosemanticity: Decomposing language models with dictionary learning," *Transformer Circuits Thread*, 2023, https://transformer-circuits.pub/2023/monosemantic-features/index.html.

8 Annexes

8.1 DyckLanguageTokenizer Implementation

```python
class DyckLanguageTokenizer:
    START_TOKEN, PAD_TOKEN, END_TOKEN = 0, 1, 2
    base_vocab = {"[start]": START_TOKEN, "[pad]": PAD_TOKEN, "[end]": END_TOKEN}

    def __init__(self, vocab: str):
        self.vocab = vocab
        self.tok_to_i = {
            **{tok: i + 3 for i, tok in enumerate(vocab)},
            **self.base_vocab,
        }
        self.i_to_tok = {i: tok for tok, i in self.tok_to_i.items()}

        # Precompute tokenization mapping
        self.char_to_token = np.zeros(256, dtype=np.float32)
        for char, token in self.tok_to_i.items():
            if len(char) == 1:
                self.char_to_token[ord(char)] = token

    def tokenize(self, strings: str | List[str], max_len=None):
        if isinstance(strings, str):
            strings = [strings]

        if max_len is None:
            max_len = max((max(len(s) for s in strings)), 1)

        # Vectorized tokenization
        tokenized = np.full((len(strings), max_len + 2), self.PAD_TOKEN, dtype=np.float32)
        tokenized[:, 0] = self.START_TOKEN

        lengths = np.array([len(s) for s in tqdm(strings, desc="Calculating lengths")])
        for i, s in enumerate(tqdm(strings, desc="Tokenizing strings")):
            tokenized[i, 1:lengths[i]+1] = self.char_to_token[[ord(c) for c in s]]

        # Efficient end token placement
        tokenized[np.arange(len(strings)), lengths + 1] = self.END_TOKEN

        return torch.from_numpy(tokenized)

    def decode(self, tokens, remove_special_tokens=True):
        if tokens.ndim < 2:
            raise ValueError("Needs to have a batch dimension.")

        def i_to_c(i):
            if i < len(self.i_to_tok):
                return self.i_to_tok[i]
            raise ValueError(f"Index {i} not in vocabulary")

        if remove_special_tokens:
            return [
                "".join(i_to_c(i.item()) for i in seq[1:] if i != self.START_TOKEN and i != self.END_TOKEN)
                for seq in tokens
            ]
        return [" ".join(i_to_c(i.item()) for i in seq) for seq in tokens]

    def decode_single(self, tokens, remove_special_tokens=True):
        return self.decode(tokens.unsqueeze(0), remove_special_tokens=remove_special_tokens)[0]

    def __repr__(self):
        return f"DyckLanguageTokenizer(vocab={self.vocab!r})"
```

Algorithm 8.1: DyckLanguageTokenizer implementation

8.2 `transformer-checker` Class Diagram

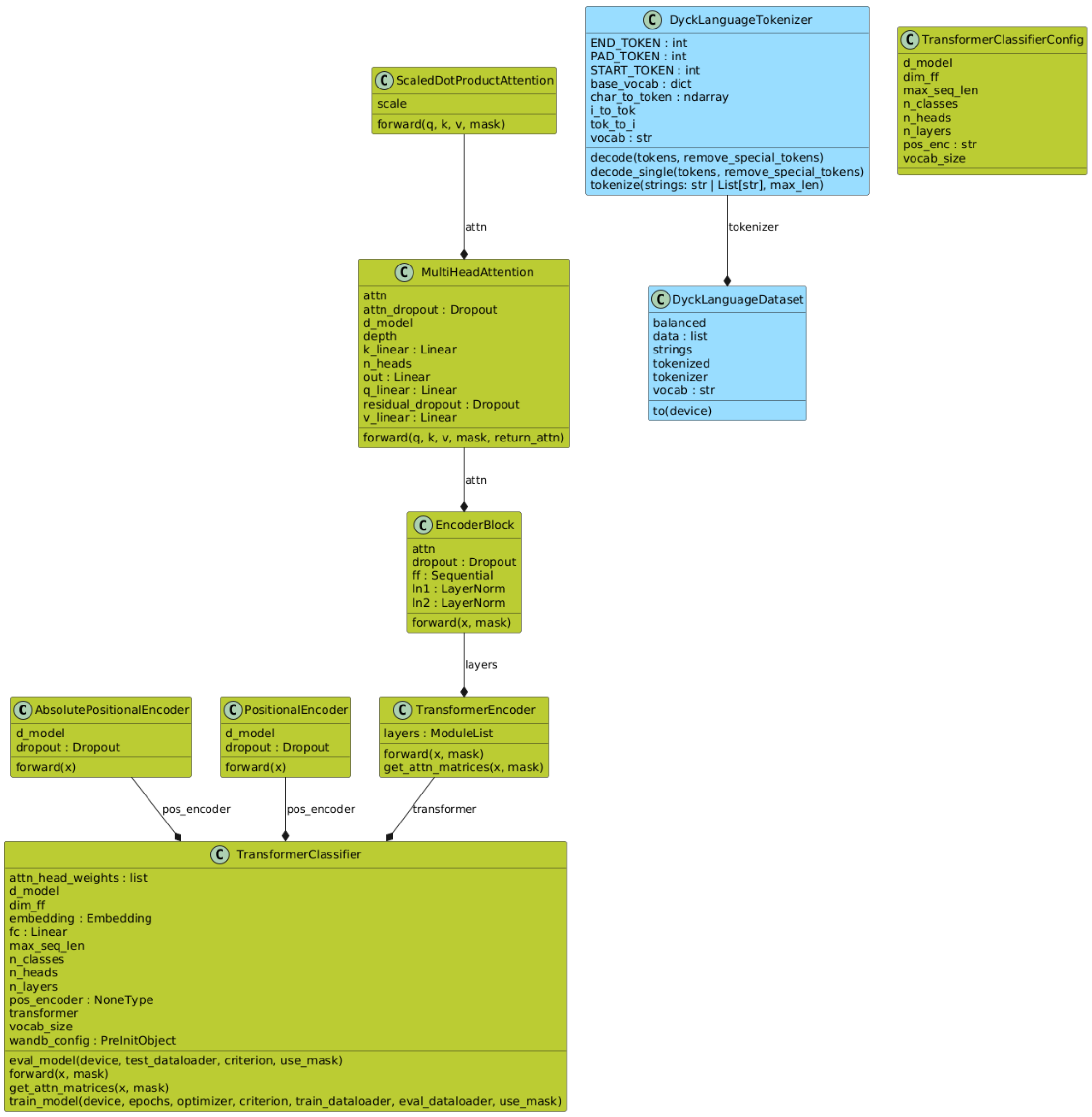


Figure 8.1: `transformer-checker` class diagram