UNIKASSEL
VERSITÄT

Master Thesis

On the Undecidability of the Output Reachability Problem for Transformer Sequence Classifiers

---

Submitted in partial fulfillment of the requirements
for the degree of

Master of Science

at the
Theoretical Computer Science / Formal Methods Group

Eric Alsmann

First Examiner	:	Prof. Dr. Martin Lange
Second Examiner	:	Prof. Dr. Bernhard Sick
Supervisor	:	Marco Sälzer, M.Sc.

Department of Electrical Engineering and Computer Science
University of Kassel

Abstract

Transformer models have revolutionised natural language processing, achieving remarkable performance in a variety of tasks. However, as they become increasingly integral to critical applications, ensuring their reliability and trustworthiness is becoming increasingly important. This paper addresses the challenging problem of verifying transformer models. We establish the undecidability of the output reachability problem for transformer sequence classifiers, highlighting the complexities arising from their massive parameter counts and complex structures. We also show that a restricted version of this problem is NP-complete, and discuss another decidability result for models with limited precision.

Declaration

I confirm that the submitted thesis is original work and was written by me without further assistance.

Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

(Eric Alsmann)

Contents

1 Introduction

In recent years, the field of natural language processing (NLP) has seen significant breakthroughs with the introduction of transformer models. Transformer models rely heavily on the architecture developed by Vaswani et al. in 2017 [VSP+17]. In the short time since its introduction, large language models (LLMs) based on this architecture have significantly advanced the state of the art in several NLP tasks, including machine translation, sentiment analysis, question answering, and speech generation. These models have shown exceptional ability to capture linguistic patterns and understand natural language. The main reasons for the success of transformer models can be attributed to their ability to effectively model long-range dependencies within sequences, overcoming previous limitations of recurrent neural networks (RNNs) and similar models. By using a self-attention mechanism, transformer models are able to capture the global context and cross-references between all elements in a sequence simultaneously. This enables the model to efficiently process input sequences of any length, resulting in superior performance to previous models.

While transformer models have achieved unprecedented performance in a wide range of NLP applications, their increasing use in real-world scenarios requires verification and validation techniques. As these models are increasingly relied upon for critical decision-making tasks such as question answering, autonomous systems, medical diagnosis, image generation and financial forecasting, the need to ensure their reliability, robustness and trustworthiness becomes particularly important. The risks of unverified transformer models, or models trained on unverified data, are many. They range from biased decision making and ethical dilemmas to potential security vulnerabilities. Undetected biases in the training data can propagate through the model, leading to discriminatory behaviour. Adversarial attacks can exploit weaknesses in the model's decision boundary, resulting in incorrect or malicious outputs. This is particularly important in tasks such as hate speech detection or fake news detection, where models need to be robust against the use of synonyms or pertubations.

The verification of transformer models is a particularly big challenge. The reason for this is in particular the massive number of parameters that make up these models. Parameter numbers of several trillion are not uncommon. Furthermore, the complex internal structure makes formal verification of the models difficult. In principle, the models are

built from two structures. One is the self-attention mechanism and the other is neural networks. Verification of neural networks is already known to be NP-hard. Verification of the self-attention mechanism presents us with even greater challenges, because of the cross-position dependencies and internal calculations based on non-rational functions and the scalar product.

Aim and Structure of This Thesis

This paper deals specifically with transformer models for sequence classification. Sequence classification means that an input sentence is classified. These models are used in various areas such as fake news and hate speech detection or the detection of fake product reviews. After the necessary mathematical foundations have been defined, the thesis first deals with the question of which verification problems are interesting in relation to this model. Then, the undecidability of the output reachability problem for transformer sequence classifiers is shown. In particular, it is discussed which mechanisms of the transformer cause the undecidability and thus also make the verification of such models so difficult. Subsequently, it is shown that if the length of the input is restricted, the problem becomes NP-complete. In the last chapter, transformer sequence classifiers that work with limited precision are considered. It is discussed how limited precision affects the power of the transformer and a first approach for a decidability proof of the output reachability problem for these models is given.

Related Work

The formal analysis of transformer models began with Hsieh et al. [HCJ+19] who experimentally investigated the robustness of transformers to small input pertubations. Furthermore, they did the first formal analysis on how sensitive the self attention mechanism is to small changes in the input. Perez et al. [PBM21] showed that the whole sequence-to-sequence transformer model is turing-complete, which was the first characterisation of transformers in terms of their computational power. They also proposed the first general formalisation of the transformer model, abstracting away the specific choice of functions and parameters and therefore, provided a mathematical framework to work with when analysing transformer models. Further research [CCP23] [HAF22] [MS23b] [MS23a] has been dealing with analysing the expressiveness of sequence classifiction transformers with arbitrary as well as limited precision with respect to circuit complexity and logics. The only notable works regarding the verification of transformers are from Shi et al. [SZC+19] and Bonaert et al. [BDBV21] who provided approximative algorithms for ensuring a special kind of adversarial robustness properties using well known techniques from the verification of neural networks like Interval Bound Propagation and Zonotopes.

2 Preliminaries

This chapter introduces the mathematical prerequisites needed for the following chapters. We introduce neural networks and give a detailed introduction into the transformer model.

2.1 Mathematical Notations

Big bold letters like $\boldsymbol{A}, \boldsymbol{B}$ indicate matrices and small bold symbols indicate vectors like $\boldsymbol{v}, \boldsymbol{w}$. For some matrix $\boldsymbol{A} \in \mathbb{K}^{n \times m}$ we denote by $(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m)$ the ordered set of column vectors, all from $\mathbb{K}^n$. Scalars are denoted by small normal symbols like x, y. Respectively we define by $(a_1, \ldots, a_n)$ the ordered set of entries in some vector $\boldsymbol{a}$. We define $[n] = \{1, \ldots, n\}$ and $[n]_0 = \{0, \ldots, n\}$. Sometimes we arrange vectors in groups, for example to write the vector which consists of the entries of vector $\boldsymbol{a}$ and a additional value x we write $[\![\boldsymbol{a}, x]\!] = (a_1, \ldots, a_n, x)^T$. Furthermore, for some set M we write $\mathcal{P}(M) = \{A \mid A \subseteq M\}$ for the power set of M.

2.2 Neural Networks

A deep neural network (DNN) with n inputs and m outputs computes a function $f : \mathbb{R}^n \to \mathbb{R}^m$. It is composed of an input layer and several hidden layers. Each hidden layer is composed of neurons. A neuron v with input dimension k can be seen as a function $v : \mathbb{R}^k \to \mathbb{R}$ with

$$v(\boldsymbol{x}) = \sigma\left(\sum_{i=0}^{k-1} w_i x_i + b_i\right),\ w_i, b_i \in \mathbb{R} \tag{2.1}$$

The activation function σ is either the identity function or the rectified linear unit (ReLu) $\text{relu}(x) = \max(0, x)$. A hidden layer l is an ordered set of neurons $(v_0^l, \ldots, v_{k_l-1}^l)$ all with

the same input dimension . k_l is the number of neurons in layer l. It computes a function $v^l : \mathbb{R}^{k_{l-1}} \to \mathbb{R}^{k_l}$ with

$$v^l(\boldsymbol{x}) = \begin{pmatrix} v_0^l(\boldsymbol{x}) \\ \vdots \\ v_{k_l-1}^l(\boldsymbol{x}) \end{pmatrix}$$

A DNN N with n inputs, m outputs and l hidden layers, therefore computes the function $f_N : \mathbb{R}^n \to \mathbb{R}^m$ with

$$f_N(\boldsymbol{x}) = v^{l-1} \circ v^{l-2} \circ \cdots \circ v^0(\boldsymbol{x}).$$

Moreover, we define the size of a DNN N as the total number of neurons, occuring in N.x

2.2.1 Finite Sample Expressivity

The following results characterise the expressibility of neural networks in terms of their ability to remember a finite set of input-output pairs. We will need this later in some places to consider neural networks in transformers as a kind of black-box functions that can check finitely many conditions. The results go back to [ZBH+21], but we prove them constructively in a somewhat simpler way in this context.

Lemma 2.2.1. *Let $\mathbb{B} \subseteq \mathbb{R}$ and $|\mathbb{B}| = n < \infty$. For every function $f : \mathbb{B} \to \mathbb{R}$, there is a DNN N_f of size $\mathcal{O}(n)$ with $f(x) = N_f(x)$ for all $x \in \mathbb{B}$.*

Proof. Due to the finiteness of $\mathbb{B}$, we can construct a DNN which memorizes the finitely many input-output pairs of f. Let $(x_1, \ldots, x_k)$ be a enumeration of all elements in $\mathbb{B}$ with $x_i \leq x_{i+1}$ for all $0 \leq i \leq k-1$. The constructed DNN will have one hidden layer with n neurons. The i-th neuron will compute the function

$$f_i(x) = \text{relu}\left(\frac{x}{x_i - x_{i-1}} - \frac{x_{i-1}}{x_i - x_{i-1}}\right)$$

The function has the property $f_i(x_i) = 1$ and $f_i(x_j) = 0$ for all $j < i$. Therefore, if the input is x_i, $f_j(x_i) = 0$ for all $j \geq i$, because $x_i < x_j$. To get the output value y_i, we have to define the weight w_i for the output layer as follows:

$$w_i = y_i - \sum_{j=1}^{i-1} w_j \cdot f_j(x_i)$$

Then the output is

$$\begin{aligned} N(x_i) &= f_i(x_i) \cdot w_i + \sum_{j=1}^{n} w_j \cdot f_j(x_i) \\ &= w_i + \sum_{j=1}^{i-1} w_j \cdot f_j(x_i) \\ &= y_i - \sum_{j=1}^{i-1} w_j \cdot f_j(x_i) + \sum_{j=1}^{i-1} w_j \cdot f_j(x_i) \\ &= y_i \end{aligned}$$

□

Lemma 2.2.2. *Lemma 2.2.1 also holds when* $\mathbb{B} \subseteq \mathbb{R}^m$.

Proof. We add a hidden layer with a single neuron behind the input layer. This neuron now maps each vector from the finite input set to a unique real number. So we get a bijective function $\phi : \mathbb{B} \to \mathbb{F}$, where $\mathbb{F} \subseteq \mathbb{R}$ and $|\mathbb{F}| < \infty$. We can therefore write the function f as a composition of the functions $f' : \mathbb{F} \to \mathbb{R}$ and ϕ with $f(\boldsymbol{x}) = f'(\phi(\boldsymbol{x}))$. f' can be computed by a neuronal network after Lemma 2.2.1 and ϕ can be represented as a linear map, which is obviously computable by a neural network. □

Theorem 2.2.3. *Lemma 2.2.1 also holds for functions* $f : \mathbb{B} \subseteq \mathbb{R}^n \to \mathbb{R}^m$ *and the resulting DNN is of size* $\mathcal{O}(|\mathbb{B}| \cdot m)$.

Proof. We define a function f_i for all output components $1 \leq i \leq m$ as in Lemma 2.2.2. The final DNN f is obtained by a join of all component neural networks. □

2.3 Transformers

A transformer model is designed to process sequences of data, such as sentences or time series data, by capturing dependencies and relationships between elements in the sequence. Transformers rely on a mechanism called self-attention to weigh the importance of different elements in the input sequence when making predictions. This self-attention mechanism allows the model to consider all positions in the input sequence simultaneously and enables it to capture long-range dependencies, making it highly effective for tasks involving context and understanding relationships between elements in the sequence.

In this section we present a formalisation of the transformer model. Transformer were introduced by Vaswani et al. in 2017 [VSP+17]. The transformer model consists of an encoder-decoder structure, where both the encoder and decoder are composed of multiple layers. Each layer in the encoder and decoder consists of two sub-layers: a multi-head self-attention mechanism and a position-wise feed-forward neural network. In this work we will consider encoder-only models. Such models are used for sequence classification, where a input word is classified.

The following mathematical notations used to define the transformer model are based on work from Perez et. al. [PBM21], who first proposed a formalisation of transformer models, which can be used to simplify formal proofs.

2.3.1 Word Embeddings

For this work, we only consider transformers that operate on words. Let Σ be some finite input alphabet. For some input word $w \in \Sigma^*$, each symbol (sometimes also called token) is mapped to a numerical vector of embedding dimension d_e. These initial embeddings are learned during training in form of a neural network, and the model adjusts them to capture meaningful features of the words. Hence, the word embedding is a function $f_{\text{emb}} : \Sigma \to \mathbb{R}^{d_e}$, which naturally extends to words by symbol-wise application $f_{\text{emb}}(w)$:

$$\begin{array}{ccccccccc} w & = & w_1 & w_2 & w_3 & \cdots & w_n & & \\ & & \downarrow & \downarrow & \downarrow & \cdots & \downarrow & f_{\text{emb}} & \\ & & \boldsymbol{x}_1 & \boldsymbol{x}_2 & \boldsymbol{x}_3 & \cdots & \boldsymbol{x}_n & & \end{array}$$

2.3.2 Positional Encoding

Positional encoding is another crucial component of transformer models, working hand in hand with word embeddings to enable these models to process sequential data such as text. In transformers, word embeddings on their own do not contain any information about the order or the position of the symbols within a sequence. Positional encoding is designed to overcome this limitation by injecting positional information into the embeddings. To each word embedding vector, positional encoding adds unique positional values. These values encode the relative positions of words, allowing the model to distinguish between words in different positions within a sequence. This positional awareness is essential for transformers to effectively capture dependencies between symbols that are not purely based on their semantics.

Mathematically the positional encoding is simply a function $\text{pos} : \mathbb{N} \to \mathbb{R}^{d_e}$, which assigns every position a vector of positional information, which is then added to the respective

word embedding vector. This means for some input word $w = a_1 \cdots a_n$, the input into the transformer is a vector sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$, with $\boldsymbol{x}_i = f_{\text{emb}}(a_i) + \text{pos}(n)$:

$$
\begin{array}{ccccccccccc}
w & = & & w_1 & & w_2 & & w_3 & \cdots & w_n & \\
 & & & \downarrow & & \downarrow & & \downarrow & \cdots & \downarrow & f_{\text{emb}} \\
 & & & \boldsymbol{a}_1 & & \boldsymbol{a}_2 & & \boldsymbol{a}_3 & \cdots & \boldsymbol{a}_n & \\
 & & & \oplus & & \oplus & & \oplus & \cdots & \oplus & \\
 & & & \text{pos}(1) & & \text{pos}(2) & & \text{pos}(3) & \cdots & \text{pos}(n) & \\
 & & & \downarrow & & \downarrow & & \downarrow & \cdots & \downarrow & \\
\boldsymbol{X} & = & (& \boldsymbol{x}_1 & & \boldsymbol{x}_2 & & \boldsymbol{x}_3 & \cdots & \boldsymbol{x}_n &) \\
\end{array}
$$

2.3.3 Self-Attention

The key concept behind transformer models is self-attention. Basically, self-attention allows the transformer to attend to different positions across the whole input sequence to capture relevant contextual information. An importance score is assigned to each position. This importance score is determined by attending to every other position in relation to the position currently under consideration.

In detail, each position in the input sequence is linearly transformed into a query, key and value vector. Then, for each position, the query vector is evaluated against each key vector of every other position. These scores determine the importance of each position in the sequence relative to the position currently in focus. The score values are then normalised for each position and used to weight the corresponding value vectors. The weighted sum then produces the final representation for each position in the input sequence.

Definition 2.3.1. For $d, n > 0$, let $\text{score} : \mathbb{R}^d \to \mathbb{R}$ be a scoring and $\rho : \mathbb{R}^n \to \mathbb{R}^n$ be a normalization function. Assume that $\boldsymbol{q} \in \mathbb{R}^{d_e}$ and $\boldsymbol{K} \in \mathbb{R}^{n \times d_e}, \boldsymbol{V} \in \mathbb{R}^{n \times d_h}$. Attention is a function $f^{\text{att}} : \mathbb{R}^{d_e} \times \mathbb{R}^{n \times d_e} \times \mathbb{R}^{n \times d_h} \to \mathbb{R}^{d_h}$ with

$$
\begin{aligned}
\boldsymbol{s} &= \rho\left(\text{score}(\boldsymbol{q}, \boldsymbol{k}_1), \ldots, \text{score}(\boldsymbol{q}, \boldsymbol{k}_n)\right) \\
f^{\text{att}}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) &= s_1 \boldsymbol{v}_1 + \cdots + s_n \boldsymbol{v}_n
\end{aligned}
$$

In the original work of Vaswani et. al. [VSP+17], the scalar product $\langle \boldsymbol{q}, \boldsymbol{k}_i \rangle$ is used as the score function and the softmax function as the normalisation where:

$$
\text{softmax}(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}},\ 1 \leq i \leq n.
$$

In recent work [PBM21] on the formal analysis of transformer models, the hardmax function has often been used for normalisation. The hardmax function is defined as $\text{hardmax}(x_i) = 1$ is x_i is the maximum value in $\boldsymbol{x}$, otherwise $\text{hardmax}(x_i) = 0$. If the maximum value is not unique and r is the number of occurences of the maximum value then $\text{hardmax}(x_i) = \frac{1}{r}$ for all such positions. The advantage of using hardmax is that we can easily construct transformers that pay attention to a certain position, or all positions that fulfil a certain property. A basic property required by all possible normalisation functions is averaging. This means that every normalisation function ρ must have the following form:

$$\rho(\boldsymbol{x})_i = \frac{f(x_i)}{\sum_{k=1}^{n} f(x_k)}$$

for some function f. For soft attention f is the exponential function and for hard attention f is the function, which is 1 if x_i is the maximal value in $\boldsymbol{x}$ and 0 otherwise. This averaging allows the transformer to kind of count specific positions, as we will see later. We call an attention-head for softmax normalisation hard attention and with hardmax normalisation soft attention.

In order to allow a transformer model to attend one word to different positions and be able to have several representation subspaces, the model uses several attention heads. This works by generating several query, key and value vectors for each input token and aggregating the embedding of each attention head together. The aggregation works by applying a linear projection to get a vector of the desired output dimension.

Definition 2.3.2. For $h, d_h > 0$, let f_{att} be an attention function. Assume that $\boldsymbol{q}_i \in \mathbb{R}^d$, $\boldsymbol{K}_i \in \mathbb{R}^{n \times d}, \boldsymbol{V}_i \in \mathbb{R}^{n \times d_h}$, for all $1 \leq i \leq h$ and W is a linear map from $\mathbb{R}^{h \cdot d_h}$ to $\mathbb{R}^{d_h}$. Multi-Head Attention with h Heads is a function $f^{\text{att}}_{\text{multi}}$ with:

$$f^{\text{att}}_{\text{multi}}(\boldsymbol{q}_1, \ldots, \boldsymbol{q}_h, \boldsymbol{K}_1, \ldots, \boldsymbol{K}_h, \boldsymbol{V}_1, \ldots, \boldsymbol{V}_h) = W([\![\boldsymbol{a_1}, \ldots, \boldsymbol{a_h}]\!])$$

where $\boldsymbol{a}_i = f^{\text{att}}(\boldsymbol{q_i}, \boldsymbol{K_i}, \boldsymbol{V_i})$.

2.3.4 Encoders

An encoder layer of a transformer is divided into three phases. The input sequence is first passed through a self-attention layer and then the output of each position passes an additional neural network.

Definition 2.3.3. Let $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ be some input sequence with $\boldsymbol{x}_i \in \mathbb{R}^{d_e}$. A transformer encoder layer is parameterized in

$$\theta = (Q_1, \ldots, Q_h, K_1, \ldots, K_h, V_1, \ldots, V_h, W, \mathcal{N})$$

where Q_i, K_i, V_i, W are linear maps and $\mathcal{N}$ is a neural network, which yields an output sequence $\boldsymbol{Z} = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)$ with $\boldsymbol{z}_i \in \mathbb{R}^{d_e}$ with

$$\begin{aligned}\boldsymbol{a}_i &= f^{\text{att}}_{\text{multi}}(Q_1(\boldsymbol{x}_i), \ldots, Q_h(\boldsymbol{x}_i), K_1(\boldsymbol{X}), \ldots, K_h(\boldsymbol{X}), V_1(\boldsymbol{X}), \ldots, V_h(\boldsymbol{X})) + \boldsymbol{x}_i \\ \boldsymbol{z}_i &= \mathcal{N}(\boldsymbol{a}_i) + \boldsymbol{a}_i\end{aligned}$$

A transformer can have several encoder layers. The first layer gets the input sequence as input and the other ones get the output of the previous one. We denote a l-layer transformer encoder with parameters $\theta_1, \ldots, \theta_l$ as $\text{TEnc}(\boldsymbol{X}; \theta_1, \ldots, \theta_l)$. We may omit the parameters if they are not important in the context.

2.3.5 Sequence Classification

One application for transformer models is sequence classification or sometimes called sentiment analysis. A sequence classification transformer only uses encoder layers and one final DNN for the classification.

Definition 2.3.4. A transformer sequence classification model T_{seq} is a function $f : \Sigma^* \to \mathbb{R}$. Supppose we have a embedding with positional encoding f_{emb}, a l-layer encoder $\text{TEnc}(\theta_1, \ldots, \theta_l)$ and a DNN $\mathcal{N}_{\text{class}} : \mathbb{R}^d \to \mathbb{R}$. Then

$$T_{\text{seq}}(w) = \mathcal{N}_{\text{class}}\left(\sum_{i=0}^{n} \boldsymbol{z}_i\right) \tag{2.2}$$

$$\boldsymbol{Z} = \text{TEnc}(f_{\text{emb}}(w); \theta_1, \ldots, \theta_l) \tag{2.3}$$

This definition of sequence classification can be seen as a language acceptor. Let $t \in \mathbb{R}$ be some threshold value, then a transformer sequence classifier T_{seq} defines a language $L_{T_{\text{seq}}}$ over Σ by:

$$L_{T_{\text{seq}}} = \{w \in \Sigma^* \mid T_{\text{seq}}(w) \geq t\}$$

3 Verification Problems for Transformer Models

This chapter discusses the various verification problems that are of interest in the context of transformers. The recent success of large language models (LLMs), such as GPT [BMR+20] or BART [LLG+20], and their applications in various safety-critical environments, such as hate speech detection [MNO20], fake news detection [KGN21], autonomous driving [PCG21], mathematical problem solving [CKB+21] and financial forecasting [YSPK21], have led to the need for techniques to verify such models. For example, it is desirable that the models are invulnerable to word substitutions and the use of synonyms.

Word embedding plays a central role in the robustness of such models. A priori, we assume that during word embedding learning, semantically similar words (synonyms) are mapped to similar embedding vectors with respect to a given norm. To ensure this assumption, there are techniques to ensure during the training process that this assumption is met and that the transformer is invariant to small changes in the input [WGL19] [JRGL19]. However, it is difficult to formally check whether, for a given input sequence, a trained transformer model is robust to the use of synonyms and perturbations. As the length of the text increases, the number of possible transformations scales exponentially. Therefore, existing research on verifying transformers tries to skip the embedding part and assumes similarity of synonyms after embedding. This makes the problem easier to approach, because there are already well-studied techniques from the verification of neural networks like Interval Bound Propagation, Mixed Integer Programming, SMT solving and others that can possibly be adapted to transformers. While this solves one of the challenges, another difficulty that makes verification of transformer models difficult is the complex self-attention mechanism with the use of the non-rational functions and cross-position dependencies. A recently studied verification problem for adversarial robustness of fixed input sequences is the following. Given an input sequence after embedding $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x_n})$, a set of perturbed positions P and some small value of ϵ, the question is whether, for all perturbed input sequences $\widetilde{\boldsymbol{X}}$ with $\|\boldsymbol{x}_i - \widetilde{\boldsymbol{x}}_i\| < \epsilon$ for all $\boldsymbol{x}_i \in P$, the output of the transformer model is the same as for $\boldsymbol{X}$. Or one could ask for the maximum value of ϵ for which this is the case. Existing work uses well-known

techniques from the verification of neural networks, such as Interval Bounded Propagation [SZC+19] or Zonotopes [BDBV21], to approximate this problem. To date, there is no existing sound and complete method for solving this problem. There is also no existing work that additionally considers permutations in the input sequence.

The focus of this work is on the verification of already trained models. The verification problem we consider is a more simple problem called output reachability. The output reachability problem is well known in the field of neural networks [BTT+18]. Intuitively, an output reachability property wants to ensure that certain misbehaviour does not occur. Given an input specification and an output specification, the output reachability problem asks whether there is an input that satisfies the input specification such that the output satisfies the output specification. For neural networks this problem is known to be NP-complete [SL21] [KBD+17]. This already hints at NP-hardness as a first lower bound on the complexity of the output reachability problem for transformers, since neural networks are used as part of the transformer model. For the scope of this work, we will work with a simplified version of the output reachability problem that does not consider any specification and is defined as follows:

Definition 3.0.1 (TransReach)**.** The output reachability problem for transformer models is the following: Given some transformer model T_{class} over some alphabet Σ, is there a word $w \in \Sigma^*$, such that T_{class} accepts w?

4 Undecidability of TransReach for Transformer Sequence Classifiers

This chapter aims to show the undecidability of the output reachability problem for transformer sequence classifiers. We will show the undecidability result using a reduction of Post's Correspondence Problem (PCP) [Pos46]. We briefly review the definition of the PCP.

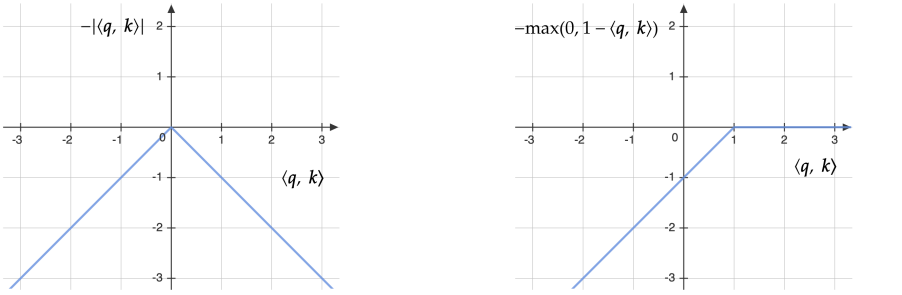
Definition 4.0.1 (PCP)**.** Let Γ be some finite alphabet. An instance of the PCP is an ordered set $P = ((\alpha_1, \beta_1), \ldots, (\alpha_k, \beta_k)) \subseteq \Gamma^+ \times \Gamma^+$. The decision problem asks if there is a finite non-empty sequence of indices $i_1, \ldots, i_l$ from $[k]$, such that $\alpha_{i_1} \cdots \alpha_{i_l} = \beta_{i_1} \cdots \beta_{i_l}$.

Theorem 4.0.2 ([Pos46])**.** *The PCP is undecidable over all* Γ *with* $|\Gamma| \geq 2$*.*

We will always assume in the following that all PCP instances are defined over the alphabet $\{0, 1\}$.

As stated in the Preliminaries, our transformer model is generic in terms of the normalisation and scoring functions. The original transformer model of Vaswani et al. [VSP+17] uses soft attention for normalisation and the standard scalar product for scoring. However, previous works regarding the expressibility of transformer have shown, that on the one hand, using soft attention is difficult to handle formally because of the exponential function and using the standard scalar product is rather weak, because it does not allow or at least makes it quite difficult to attend to specific positions. Therefore, we make some assumptions on these functions, which are presented in the following.

Hard Attention. The first change is the use of hard attention. This has been a standard assumption so far when investigating the expressibility of Transformer models. Hard-attention allows targeted access to certain positions in a self-attention layer. So far, no technique is known that can also apply this property to soft-attention. The answer to the question whether the following arguments and results from other papers can also be applied to soft-attention is probably also related to Tarski's Exponential Function Problem [Wil97] , which asks whether the theory of real numbers is decidable together with



(a) Max-Scoring Function (b) Threshold-Scoring Function

Figure 4.1: Scoring Functions

the exponential function. The problem has not yet been solved and therefore contributes to the difficulty of dealing with the exponential function when using soft attention.

Scoring Functions. We will use two types of scoring functions, both of which add an additional component to the scalar product. This can be motivated by different insights. The scalar product used by Vaswani et al. [VSP+17] is also known as multiplicative attention. Another form of scoring function is additive attention [BCB16] where the sum is taken and then a DNN is applied. Our modification can be interpreted as a combination of these two techniques. We calculate the scalar product and apply a DNN to it. We call the two modified scoring functions max-scoring and threshold scoring. They are defined as follows:

$$\begin{aligned} \text{score}_{\max}(\boldsymbol{u}, \boldsymbol{v}) &= -|\langle \boldsymbol{u}, \boldsymbol{v} \rangle| = \text{rel}(\langle \boldsymbol{u}, \boldsymbol{v} \rangle) - \text{rel}(-\langle \boldsymbol{u}, \boldsymbol{v} \rangle) \\ \text{score}_{\text{thres}}(\boldsymbol{u}, \boldsymbol{v}) &= -\text{rel}(1 - \langle \boldsymbol{u}, \boldsymbol{v} \rangle) \end{aligned}$$

The two functions can be seen in Figure 4.1. We will use them in two different ways. The $\text{score}_{\max}$ function becomes maximum when the scalar product takes a value as close to zero as possible. We will use this to identify a specific position that has a certain property using scoring. Together with the hardmax normalisation, we can attend to a single specific position. The threshold scoring, on the other hand, allows us to focus on several positions at once. This function maps the value to 0 when the scalar product has a value greater than or equal to 1 and is negative otherwise. This allows us to focus on a set of items that all fulfil a certain property. Together with the hardmax function, it is then possible to average over these positions. This averaging can be used as inverse counting of positions, which satisfy a given constraint.

In order to simplify the definition of some neural networks, used in the following argument, we introduce the notation of multiplexer programs. These can be directly translated into neural networks with which properties of discrete vectors can be checked very easily.

Multiplexer Programs

Definition 4.0.3. Let $X = \{x_1, \ldots, x_n\}$ be a set of variables. A multiplexer program P is inductively defined by the following grammar:

$$P := x_i - x_j = k \mid x_i = k \mid \neg P \mid P \wedge P \mid P \vee P \mid P \to P$$

for all $1 \leq i, j \leq n$, $k \in \mathbb{N}$. We interpret such a program over the theory of discrete non-negative vectors. An interpretation for a multiplexer program P is a vector $\bar{\boldsymbol{x}} \in \mathbb{N}^n$ We say $\bar{\boldsymbol{x}} \models P$ if and only if $\bar{\boldsymbol{x}}$ fits to the set of variables and satisfies all conditions in P with the usual interpretation.

The goal is to construct a DNN that, given an interpretation $\bar{\boldsymbol{x}}$, outputs 0 if $\bar{\boldsymbol{x}} \models P$ and 1 if $\bar{\boldsymbol{x}} \not\models P$. It should be noted here that neural networks are always defined over the real numbers. With regard to the multiplexer programmes, however, we are only interested in the behaviour over the natural numbers. The behaviour over the non-natural numbers does not matter to us here. So in the following, when we say that a DNN calculates a discrete function, we don't care about the behaviour over all other values. The following lemma shows that it is sufficient for the DNN to output 0 in the positive case and a number other than 0 in the negative case.

Lemma 4.0.4. *For every discrete function $f : \mathbb{N}^n \to \mathbb{N}$ calculated by some DNN $\mathcal{N}_f$ there is a DNN $\mathcal{N}_{f'}$ computing the function $f' : \mathbb{N}^n \to \{0, 1\}$ with:*

$$f'(\boldsymbol{x}) = \begin{cases} 0 & , f(\boldsymbol{x}) = 0 \\ 1 & , f(\boldsymbol{x}) \neq 0. \end{cases}$$

Proof. Let $f : \mathbb{N}^n \to \mathbb{N}$ be some function, which is calculated by a DNN $\mathcal{N}_f$. Define the function f' with

$$f'(\boldsymbol{x}) = \mathrm{rel}(f(\boldsymbol{x}) - \mathrm{rel}(f(\boldsymbol{x}) - 1))$$

This function outputs 0 if the input was 0. For all other natural number inputs $x \geq 1$, the function calculates $x - (x - 1) = 1$. It can be seen that the function can also be easily implemented as a neural network. □

Lemma 4.0.5. *For every multiplexer program P over variables $X = \{x_1, \ldots, x_n\}$ there is a DNN $\mathcal{N}_P$, with n inputs and one output, such that:*

$$\mathcal{N}_P(\bar{\boldsymbol{x}}) = \begin{cases} 0 & , \bar{\boldsymbol{x}} \models P \\ 1 & , \bar{\boldsymbol{x}} \not\models P \end{cases}$$

for all $\bar{\boldsymbol{x}} \in \mathbb{N}^n$.

Proof. We prove this by induction over the structure of P. There are three base cases:

1. $P := x_i - x_j = k$. Let f be the following function

   $$f(\bar{\boldsymbol{x}}) = \mathrm{rel}(\mathrm{rel}(\bar{x}_i - \bar{x}_j - k) + \mathrm{rel}(\bar{x}_j - \bar{x}_i + k)) = \begin{cases} 0 & , \bar{x}_i = \bar{x}_j + k \\ \geq 1 & , \text{otherwise.} \end{cases}$$

   which can easily be verified. This function can obviously be calculated by a neural network. According to Lemma 4.0.4 this function can be modified to output exactly one in case, φ does not hold.

2. $P := x_i = k$. Let f be the following function

   $$f(\bar{\boldsymbol{x}}) = \mathrm{rel}(\mathrm{rel}(\bar{x}_i - k) + \mathrm{rel}(k - \bar{x}_i)) = |\bar{x}_i - k|$$

   This function can obviously be calculated by a neural network. According to Lemma 4.0.4 this function can be modified to output exactly one in case, φ does not hold.

Let $\mathcal{N}_P, \mathcal{N}_{P'}$ be neural networks for arbitrary subexpressions P, P'. There are three inductive cases:

1. $\neg P$. Let f be the function

   $$f(\bar{\boldsymbol{x}}) = \mathrm{rel}(1 - \mathcal{N}_P(\bar{\boldsymbol{x}})) = \begin{cases} 1 & , \bar{\boldsymbol{x}} \not\models P \\ 0 & , \bar{\boldsymbol{x}} \models P \end{cases}$$

2. $P \wedge P'$. Let f be the function

   $$f(\bar{\boldsymbol{x}}) = \mathrm{rel}(\mathcal{N}_P(\bar{\boldsymbol{x}}) + \mathcal{N}_{P'}(\bar{\boldsymbol{x}})) = \begin{cases} 0 & , \bar{\boldsymbol{x}} \models P \wedge \bar{\boldsymbol{x}} \models P' \\ > 0 & , \text{otherwise.} \end{cases}$$

   By using Lemma 4.0.4, we get the desired function.

3. $P \to P'$. Let f be the function

$$f(\bar{\boldsymbol{x}}) = \text{rel}(\mathcal{N}_{P'}(\bar{\boldsymbol{x}}) - \mathcal{N}_P(\bar{\boldsymbol{x}})) = \begin{cases} \mathcal{N}_{P'}(\bar{\boldsymbol{x}}) & , \bar{\boldsymbol{x}} \models P \\ 0 & , \bar{\boldsymbol{x}} \not\models P \end{cases}$$

□

Observation 4.0.6. *The size of the resulting DNN coming from Lemma 4.0.5 is linear in size of the formula.*

By closer inspection of the inductive translation, one can see that for each case a constant number of nodes are added to the DNN independently from any parameters occuring in the formula.

4.1 Overview of the Reduction

In this section we present the basic ideas we will need to show the undecidability of the output reachability problem for transformer sequence classifiers. The goal is to construct a transformer for a given PCP instance P that outputs one for an input word if and only if the input word corresponds to a correct encoding of a solution for the PCP instance. In this first section we will perform the reduction on a running example in order to formally define the reduction and prove its correctness in the next sections.

We use the following PCP instance as an example $P = ((001, 00), (1, 011), (10, 11))$. A possible solution of this PCP instance is the following:

$$\boldsymbol{s} = (1, 3, 1, 2) \Rightarrow \begin{pmatrix} 001 \\ 00 \end{pmatrix} \begin{pmatrix} 10 \\ 11 \end{pmatrix} \begin{pmatrix} 001 \\ 00 \end{pmatrix} \begin{pmatrix} 1 \\ 011 \end{pmatrix}$$

The first step is to translate this solution into a word over a finite alphabet. We use multitrack characters to encode the solution. We use these to store multiple pieces of information in one symbol. We encode each symbol of the solution separately and store additional information, such as whether the symbol belongs to a tile type, whether it is the beginning of a lower or upper tile. By saying lower tile we mean the upper part of a tile in sequence. For the above example the upper tiles are $001, 10, 001, 1$ and the lower tiles are $00, 11, 00, 011$. In Figure 4.3, we see what the word representation of the solution looks like. We also require, for technical reasons, that the word encoding of a PCP instance always ends with a distinguished end marker. This end marker is start

symbol for top and bottom and also has the unique tile type zero, which encodes the end. Therefore, we can represent every possible solution of the PCP instance using a finite alphabet Σ.

Obviously, not every word over Σ is an encoding of a possible solution. For this reason, the transformer must check two things. First, that the input word corresponds to a correct encoding of a possible solution and second, if the input word encodes a possible solution, that this solution is also correct with respect to the given PCP instance. In Figure 4.2, we see the schematic structure of the transformer for a PCP instance. The resulting transformer will have three encoder layers. The first layer is responsible for validating the input word. In this layer, we only check whether the input word is actually a possible solution for the PCP instance. However, we do not check whether the upper and lower tiles belong together. This is done in encoder layers two and three. The matching of the upper and lower tiles is more complicated and takes place in two phases. First, we use the power of self-attention to number the tiles in the input top and bottom. In the second phase, we check whether tiles with the same number at the top and bottom also have the same tile type. If all these checks are positive, the input word is indeed a valid solution for the PCP instance. If a check is negative, the input word is either not coded correctly or the top and bottom tiles do not match correctly. In the following, we will now schematically go into more detail about the individual components of the Transformer.

Word Encoding. The first step is to translate a solution, as seen in Figure 4.3, into a word. As mentioned above, we use multitrack characters to co-encode additional information for each symbol of the solution word. What this looks like can be seen in Figure 4.3. We additionally encode the following information:

1. A marker whether the current symbol is the start of a new upper or lower tile.
2. The type of tile.
3. The ID of the tile, but this is only to be specified a priori for the first tile, since otherwise we would need an infinite alphabet. We assign the remaining ids with the help of the transformer.

Word Embedding. Having shown how we can encode each solution of the PCP instance over the alphabet Σ, we now outline the embedding function we will use to translate the word into a sequence of reel-valued vectors. In Figure 4.4, we see the sequence of vectors for our example after the word embedding. The translation is done character by character. Basically, we translate the multitrack characters one-to-one into the vectors and add some auxiliary dimensions that will be needed in later steps. These include

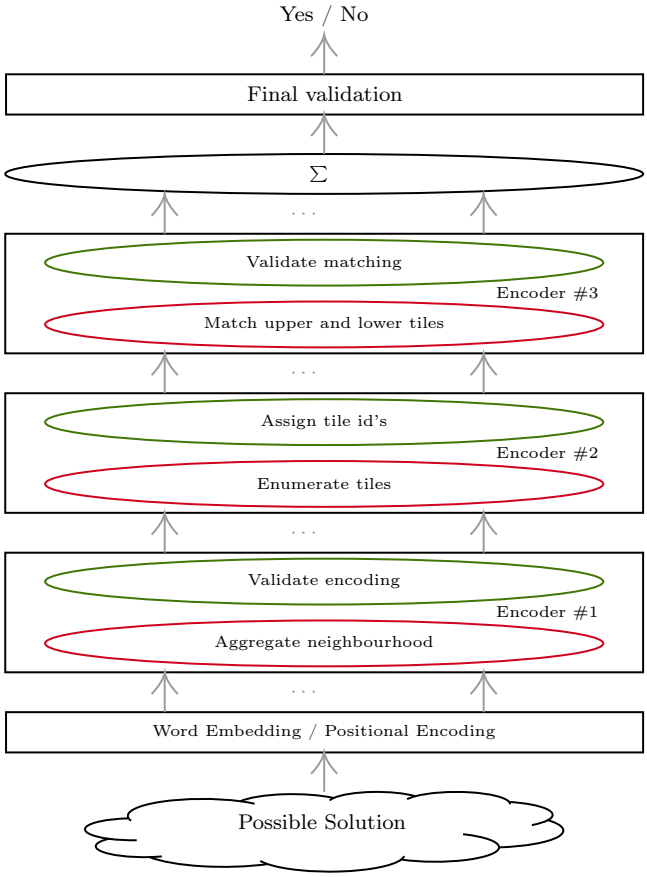


Figure 4.2: Example of the word embedding of a PCP solution.

symbol	0	0	1	1	0	0	0	1	1	0	⇒	$\begin{bmatrix}0\\1\\1\\1\\1\\1\\1\end{bmatrix}$ $\begin{bmatrix}0\\0\\0\\1\\1\\0\\0\end{bmatrix}$ $\begin{bmatrix}1\\0\\1\\1\\3\\3\\0\end{bmatrix}$ $\begin{bmatrix}1\\1\\0\\3\\3\\0\\0\end{bmatrix}$ $\begin{bmatrix}0\\0\\1\\3\\1\\0\\0\end{bmatrix}$ $\begin{bmatrix}0\\1\\0\\1\\1\\0\\0\end{bmatrix}$ $\begin{bmatrix}0\\0\\1\\1\\2\\0\\0\end{bmatrix}$ $\begin{bmatrix}1\\0\\0\\1\\2\\0\\0\end{bmatrix}$ $\begin{bmatrix}1\\1\\0\\2\\2\\0\\0\end{bmatrix}$ $\begin{bmatrix}0\\1\\1\\0\\0\\0\\0\end{bmatrix}$
$\text{start}_\top$	∘			∘		∘			∘	∘		
$\text{start}_\bot$	∘		∘		∘		∘			∘		
$\text{type}_\top$	1	1	1	3	3	1	1	1	2	0		
$\text{type}_\bot$	1	1	3	3	1	1	2	2	2	0		
$\text{id}_\top$	1	0	0	0	0	0	0	0	0	0		
$\text{id}_\bot$	1	0	0	0	0	0	0	0	0	0		

Figure 4.3: Example of the word encoding of a PCP solution.

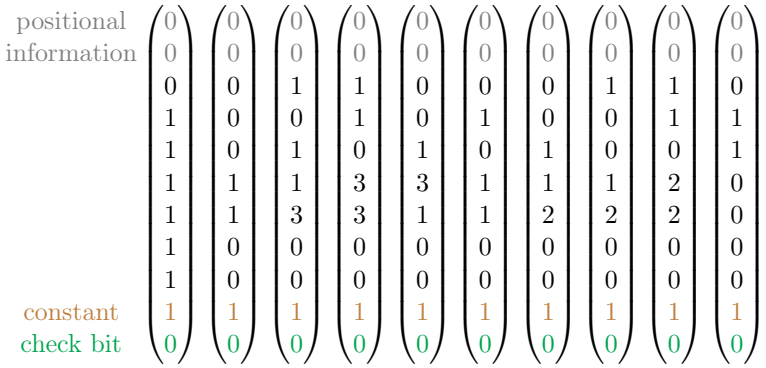


Figure 4.4: Example of the word embedding of a PCP solution.

two dimensions for the positional encoding, which will be explained in more detail in the next paragraph, a dimension that always has the value one, since we will need this in calculations, and a dimension that will have the value 0 at the beginning. This last dimension is the check dimension. We will use it at the end in the last step to check whether all phases of the transformer have been passed through successfully. If a check of the transformer fails, there will be a non-zero number in one of the vectors in the check dimension at the end. This allows us to check at the end whether it is really a correct solution of the PCP instance or not.

Positional Encoding. After we have embedded the input word, we now add positional information to the vectors. We will see that these are necessary in order to specifically access neighbours or address special positions with the help of self-attention. We will add two types of positional information to the vectors. One is a simple numbering of the positions and additionally the reciprocal of the numbering, so that these values are all smaller than 1. We will see later in the construction of the encoder layer that we need these reciprocals to check for several properties when scoring. In Figure 4.5 we see how the vectors look after the positional encoding.

Encoder 1: Validation. Within the first encoder, the input word is validated. This only checks whether the start markers, tile types and symbols are consistent with the PCP instance. It is also checked whether the ID of the start position is correct and the other ids are set to zero and also whether the end marker is present. The matching of the upper and lower tiles only happens in the later layers.

To check these properties, the neighbourhood must be looked at from each position. It is easy to see that for each position we only need to look at the right neighbours up to the length of the longest tile in the PCP instance. This is because we need to

1	2	3	4	5	6	7	8	9	10
$1/2$	$1/3$	$1/4$	$1/5$	$1/6$	$1/7$	$1/8$	$1/9$	$1/10$	$1/11$
0	0	1	1	0	0	0	1	1	0
1	0	0	1	0	1	0	0	1	1
1	0	1	0	1	0	1	0	0	1
1	1	1	3	3	1	1	1	2	0
1	1	3	3	1	1	2	2	2	0
1	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0

Figure 4.5: Example of the positional encoding of a PCP solution.

check whether the current tile of the position is consistent and whether a new tile starts after it. So the distance to the furthest neighbour needed is limited by the PCP instance. Therefore, we use self-attention with a corresponding number of attention heads to collect the information of the required neighbours and then check the properties with the help of a neural network.

Encoder 2: Tile Enumeration. After ensuring that the input word is coded correctly, the next step is to determine whether the upper and lower tiles match. This means whether the sequence of the upper tile types matches the sequence of the lower tile types. To check this, the transformer must be able to explicitly address each tile in the input word. We realise this by using self-attention to number the tiles. Basically, at each position we count the number of upper and lower start markers that have occurred up to that position. We then write this result into the vector representation as tile ID. In the following encoder, we must then check whether for each tile ID at the top or bottom there is a corresponding tile ID on the other side and whether the tile type matches in each case. In Figure 4.6 we can see what the new vector representation of the input word looks like after passing through the encoder.

Encoder 3: Tile Matching. After the tiles have been numbered, we can now match the upper and lower tiles and check whether the tile types actually match. To do this, we again use self-attention to find the corresponding start tile with the same ID on the other track for each position. Once this is found, the type is compared with the help of a neural network. Figure 4.6 shows the matching for our example.

1	2	3	4	5	6	7	8	9	10
$1/2$	$1/3$	$1/4$	$1/5$	$1/6$	$1/7$	$1/8$	$1/9$	$1/10$	$1/11$
0	0	1	1	0	0	0	1	1	0
1	0	0	1	0	1	0	0	1	1
1	0	1	0	1	0	1	0	0	1
1	1	1	3	3	1	1	1	2	0
1	1	3	3	1	1	2	2	2	0
1	1	1	$1/2$	$1/2$	$1/3$	$1/3$	$1/3$	$1/4$	$1/5$
1	1	$1/2$	$1/2$	$1/3$	$1/3$	$1/4$	$1/4$	$1/4$	$1/5$
1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0

Figure 4.6: Example of the tile enumeration of a PCP solution.

Final Classification. After all encoders have been passed through, the vectors are summed up and finally fed into a DNN for classification. As mentioned above, all encoders work in such a way that if a check fails, the check bit of the vector is set to a non-zero value. Conversely, this means that after all vectors have been added up, it is only necessary to check whether the value of the check bit is zero. If this is the case, the input word actually corresponds to a solution of the PCP instance. If the value is not zero, the input word was either not coded correctly or the coding did not correspond to a solution.

4.2 Undecidability Proof

In this section, the above reduction is now formally defined. The section is divided into the individual components in the same way as the previous one.

4.2.1 Word Encoding

Suppose we are given a PCP instance $P = ((\alpha_1, \beta_1), \ldots, (\alpha_k, \beta_k))$. We define for the remaining chapter c as the maximum length of a tile in P:

$$c = \max\{|\alpha_i|, |\beta_i| \mid 1 \leq i \leq k\}$$

Suppose we have given a solution $\boldsymbol{s}$ to P by $\boldsymbol{s} = (i_1, \ldots, i_l)$ with $i_j \in [k]$. We encode solutions by multi-track symbols with 7 tracks over the finite alphabet

$$\Sigma = \{0,1\}^3 \times [k]_0^2 \times \{0,1\}^2$$

The first track will encode the solution word. The second and third tracks are there to signal whether an upper or lower tile begins at the position of the current symbol. The fourth and fifth tracks encode the respective tile type to which the current top and bottom symbol belongs. The last two tracks are needed later for technical reasons to match the upper and lower tiles.

Let $w_{\boldsymbol{s}} = \alpha_{i_1} \cdots \alpha_{i_l} = \beta_{i_1} \cdots \beta_{i_l}$ be the solution word of the solution $\boldsymbol{s}$. Furthermore, $w_{\boldsymbol{s}}^{\top}$ is a word with $|w_{\boldsymbol{s}}^{\top}| = |w_{\boldsymbol{s}}|$ and

$$(w_{\boldsymbol{s}}^{\top})_j = \begin{cases} 1 & , j = 1 + \sum_{t=1}^{m} |\alpha_{i_t}| \text{ for some } m \in [l-1]_0 \\ 0 & , \text{ otherwise.} \end{cases}$$

The word $w_{\boldsymbol{s}}^{\top}$ specifies the positions in the solution word, where a new upper tile begins. We define $w_{\boldsymbol{s}}^{\perp}$ accordingly for the lower tiles β. We define the word of tile types $w_{\boldsymbol{s}}^{\top,\text{type}}$ by

$$w_{\boldsymbol{s}}^{\top,\text{type}} = \underbrace{i_1 \cdots i_1}_{|\alpha_{i_1}|} \cdots \underbrace{i_l \cdots i_l}_{|\alpha_{i_l}|}$$

and $w_{\boldsymbol{s}}^{\perp,\text{type}}$ accordingly with β. We define a function $f_{\text{conv}} : [k]^* \to \Sigma^*$ with:

$$f_{\text{conv}}(\boldsymbol{s}) = f_{\text{conv}}((i_1, \ldots, i_l)) = \begin{pmatrix} (w_{\boldsymbol{s}})_1 \\ (w_{\boldsymbol{s}}^{\top})_1 \\ (w_{\boldsymbol{s}}^{\perp})_1 \\ (w_{\boldsymbol{s}}^{\top,\text{type}})_1 \\ (w_{\boldsymbol{s}}^{\perp,\text{type}})_1 \\ 1 \\ 1 \end{pmatrix} \begin{pmatrix} (w_{\boldsymbol{s}})_2 \\ (w_{\boldsymbol{s}}^{\top})_2 \\ (w_{\boldsymbol{s}}^{\perp})_2 \\ (w_{\boldsymbol{s}}^{\top,\text{type}})_2 \\ (w_{\boldsymbol{s}}^{\perp,\text{type}})_2 \\ 0 \\ 0 \end{pmatrix} \cdots \begin{pmatrix} (w_{\boldsymbol{s}})_{|w_{\boldsymbol{s}}|} \\ (w_{\boldsymbol{s}}^{\top})_{|w_{\boldsymbol{s}}|} \\ (w_{\boldsymbol{s}}^{\perp})_{|w_{\boldsymbol{s}}|} \\ (w_{\boldsymbol{s}}^{\top,\text{type}})_{|w_{\boldsymbol{s}}|} \\ (w_{\boldsymbol{s}}^{\perp,\text{type}})_{|w_{\boldsymbol{s}}|} \\ 0 \\ 0 \end{pmatrix} \underbrace{\begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}}_{\text{end marker}}$$

4.2.2 Word Embedding

In this section we will formally define the embedding function of our transformer. We will translate input words into vectors with embedding dimension $d_e = 11$. For this purpose, we interpret the multitrack symbols from the input alphabet as real vectors in the following and add 4 additional dimensions. We define the function $f_{\text{emb}} : \Sigma \to \mathbb{R}^{d_e}$ with:

$$f_{\text{emb}}(a) = [\![0, 0, a_{\mathbb{R}}, 1, 0]\!] \in \mathbb{R}^{d_e}$$

where $a_{\mathbb{R}}$ is the interpretation of a as a real vector. This function naturally extends to words over Σ by symbol-wise application.

Lemma 4.2.1. *f_{emb} can be computed by a DNN N_{emb}.*

Proof. $f_{\text{emb}} : \Sigma \to \mathbb{R}^{d_e}$ is a function defined over a finite domain. The input into the DNN will be a one-hot encoding of the symbol. Therefore, N_{emb} computes a function $N_{\text{emb}} : \mathbb{R}^{|\Sigma|} \supset A \to \mathbb{R}^{d_e}$ with $|A| < \infty$. We can construct this DNN by Theorem 2.2.3. □

In the following sections we will always use the abbreviations presented to refer to specific positions in the vectors. Here is an overview over all abbreviations: Let $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$

be a encoding of a word over Σ:

$$
\begin{aligned}
l^{(i)} &: \text{ the symbol at position } i \text{ (dimension: 3)} \\
s_{\top}^{(i)} &: \text{ upper start tile marker at position } i \text{ (dimension: 4)} \\
s_{\bot}^{(i)} &: \text{ lower start tile marker at position } i \text{ (dimension: 5)} \\
t_{\text{type},\top}^{(i)} &: \text{ type of upper tile at position } i \text{ (dimension: 6)} \\
t_{\text{type},\bot}^{(i)} &: \text{ type of lower tile at position } i \text{ (dimension: 7)} \\
t_{\text{id},\top}^{(i)} &: \text{ ID of upper tile at position } i \text{ (dimension: 8)} \\
t_{\text{id},\bot}^{(i)} &: \text{ ID of lower tile at position } i \text{ (dimension: 9)} \\
t_{\text{check}}^{(i)} &: \text{ check bit of position } i \text{ (dimension: 11)}
\end{aligned}
$$

4.2.3 Positional Encoding

As the last step of embedding, we add positional information to the input vectors. These are necessary in order to access special positions or to be able to address relative positions. We define the function $\text{pos} : \mathbb{N} \to \mathbb{R}^{d_e}$ as follows:

$$\text{pos}(n) = [\![n, \frac{1}{n+1}, 0, \ldots, 0]\!]$$

We add a simple numbering to all vectors on the one hand and an inverse numbering on the other hand, where $n+1$ ensures that all these values are smaller than 1. We use the following abbreviations for these two values:

$$
\begin{aligned}
p^{(i)} &: \text{ absolute position of position } i \text{ (dimension: 1)} \\
p^{-1,(i)} &: \text{ inverse absolute position of position } i \text{ (dimension: 2)}
\end{aligned}
$$

When the position of the vector talked about is clear, we also just write i or $\frac{1}{i+1}$.

This allows us to define the final input into our transformer. We define the function $f_{\text{in}} : \Sigma^* \to (\mathbb{R}^{d_e})^*$ as follows. Let $w \in \Sigma^*$ be a word with $w = a_1 \cdots a_n$.

$$f_{\text{in}}(w) = f_{\text{in}}(a_1, \ldots, a_n) = f_{\text{emb}}(a_1) + \text{pos}(1), \ldots, f_{\text{emb}}(a_n) + \text{pos}(n)$$

4.2.4 Encoder 1: Aggregating Neighborhood Information

In this section we start to construct the first encoder of the transformer. As described above, the first encoder is responsible for validating the input word. This includes cor-

rectly specifying the start positions, matching tile types and symbols, and correctly encoding the start and end positions. To check these conditions with the help of a neural network, we need neighbourhood information. We need to look to the right at most to the start of the next tile. This distance is determined by the length of the maximum tile c. We will now use self-attention with $c+1$ many attention heads to aggregate the information of the c nearest right neighbours including self-information.

Lemma 4.2.2. *There is a self-attention layer f_{agg} with $c+1$ attention heads and appropriate linear maps, such that for each input vector $\boldsymbol{x}_i$, $f_{agg}(\boldsymbol{q}_i, \boldsymbol{K}, \boldsymbol{V})$ is a vector, which contains the information defined below for $\boldsymbol{x}_i$ and its c right-neigbors.*

Proof. We will use max-scoring for all attention heads. Furthermore, we define linear maps $Q^h_{\text{agg}}, K^h_{\text{agg}}, V^h_{\text{agg}}$ for all $0 \leq h \leq c$ with:

$$Q^h_{\text{agg}}(\boldsymbol{x}_i) = \boldsymbol{q}_i = \begin{pmatrix} 1 \\ i+h \end{pmatrix}$$

$$K^h_{\text{agg}}(\boldsymbol{x}_i) = \boldsymbol{k}_i = \begin{pmatrix} i \\ -1 \end{pmatrix}$$

$$V^h_{\text{agg}}(\boldsymbol{x}_i) = \boldsymbol{v}_i = [\![\underbrace{\mathbf{0}, \ldots, \mathbf{0}}_{h}, \boldsymbol{s}_i, \underbrace{\mathbf{0}, \ldots, \mathbf{0}}_{c-h}]\!]$$

$$\boldsymbol{s}_i = \begin{pmatrix} i \\ l^{(i)} \\ s^{(i)}_{\top} \\ s^{(i)}_{\bot} \\ t^{(i)}_{\text{id},\top} \\ t^{(i)}_{\text{id},\bot} \\ t^{(i)}_{\text{type},\top} \\ t^{(i)}_{\text{type},\bot} \end{pmatrix}$$

For each attention head f^j_{agg} and $\boldsymbol{q}_i, \boldsymbol{k}_j$ defined as above holds:

$$\text{score}_{\max}(\boldsymbol{q}_i, \boldsymbol{k}_j) = -|\langle \boldsymbol{q}_i, \boldsymbol{k}_j \rangle| = -|j - (i+h)| = \begin{cases} 0 & , j = i+h \\ < 0 & , \text{otherwise.} \end{cases}$$

It is easy to see, that by applying the hardmax normalisation, we will attend to exactly one position p_i.

$$p_i = \underset{1 \leq j \leq n}{\text{argmax}}\, \text{score}_{\max}(\boldsymbol{q_i}, \boldsymbol{k_j}) = \begin{cases} i+h & , i+h \leq n \\ n & , \text{otherwise.} \end{cases}$$

This means, that during the aggregation we only aggregate the information of neighbors to the right and if there is no more neighbor the information of the last position will be

aggregated. Therefore it follows for the attention head f^h_{agg}:

$$f^h_{\text{agg}}(\boldsymbol{q_i}, \boldsymbol{K}, \boldsymbol{V}) = \boldsymbol{a}_i^{(h)} = \begin{cases} \boldsymbol{v}_{i+h} & , i+h \leq n \\ \boldsymbol{v}_n & , i+h > n \end{cases}$$

The aggregation of all attention heads works by simply summing up the output of all attention heads. Therefore,

$$W\left(\boldsymbol{a}_i^{(0)}, \ldots, \boldsymbol{a}_i^{(c)}\right) = \boldsymbol{a}_i^{(0)} + \cdots + \boldsymbol{a}_i^{(c)} = [\![\boldsymbol{s}_i, \boldsymbol{s}_{\min(i+1,n)}, \ldots, \boldsymbol{s}_{\min(i+c,n)}]\!]$$

□

The output of the self-attention layer for each position is therefore a vector that additionally contains the information of the nearest c neighbours. This vector is now transferred position by position to a neural network, which checks the coding of the word. We will define the DNN in the next section.

4.2.5 Encoder 1: Validate Encoding

After defining the attention function for the encoder in the previous section, in this section we will construct the DNN that checks whether the input word was a correct encoding of a possible solution for the PCP instance. We take advantage of the fact that after the self-attention phase we have a vector for each position that also contains the information of the c nearest right neighbours. c is the length of the maximum tile in the given PCP instance. The DNN now checks the following conditions at each position of the input:

1. The first position must be a start position for an upper and lower tile. Furthermore, the tile type must match top and bottom.
2. There must be an end marker at the last position. This end marker has the exclusive tile type zero. Furthermore, no other end marker may occur in the input sequence.
3. For each position i, if this position is a start position for an upper tile and the tile type is k, then for all successor positions up to $i + |\alpha_k|$, these are not start positions, the tile type is also k and the symbols match α_k. Furthermore, the position $i + |\alpha_k| + 1$ must again be a starting position. The same applies to lower tiles.
4. The tile ID is one for the first position. For all other positions the tile ID must be zero.

Lemma 4.2.3. *Let $P = ((\alpha_1, \beta_1), \ldots, (\alpha_k, \beta_k))$ be some PCP instance. There is a DNN N^P_{valid}, for which $N^P_{valid}(\boldsymbol{x}_i) = \boldsymbol{y_i}$ and for $\boldsymbol{y_i}$ holds that*

$$t_{check} = \begin{cases} 0 & , \boldsymbol{x}_i \textit{ fulfills conditions (1) to (4)} \\ 1 & , \textit{otherwise.} \end{cases}$$

Proof. First, we construct a multiplexer program Q_P that checks the above conditions for a vector $\boldsymbol{x}_i$. According to Lemma 4.0.5, we can translate Q_P into a DNN N^P such that

$$N^P(\boldsymbol{x}_i) = \begin{cases} 0 & , \boldsymbol{x}_i \text{ fulfills conditions (1) to (4)} \\ 1 & , \text{otherwise.} \end{cases}$$

Finally, we modify the DNN so that the output vector has dimension d_e and the output of N^P is at the place of the check bit. We will now give multiplexer programs for each condition mentioned above:

$$\begin{aligned}
Q_1 &:= \Big(p^{(i)} = 1\Big) \rightarrow \Big((s^{(i)}_{\top} = 1) \wedge (s^{(i)}_{\bot} = 1) \wedge (t^{(i)}_{\text{type},\top} = t^{(i)}_{\text{type},\bot})\Big) \\
Q_2 &:= \Big(p^{(i)} - p^{(i+1)} = 0\Big) \rightarrow \Big((t^{(i)}_{\text{type},\top} = 0) \wedge (t^{(i)}_{\text{type},\bot} = 0)\Big) \\
&\quad \wedge \Big((t^{(i)}_{\text{type},\top} = 0) \vee (t^{(i)}_{\text{type},\bot} = 0)\Big) \rightarrow \Big(p^{(i)} - p^{(i+1)} = 0\Big) \\
Q_3 &:= \bigwedge_{j=1}^{k} \left((s^{(i)}_{\top} = 1 \wedge t^{(i)}_{\text{type},\top} = j) \rightarrow \bigwedge_{m=0}^{|\alpha_j|-1} (s^{(i+m)}_{\top} = 0 \wedge t^{(i+m)}_{\text{type},\top} = j \wedge l^{(i+m)} = \alpha_{j,m}) \right) \\
&\quad \wedge \left((s^{(i)}_{\bot} = 1 \wedge t^{(i)}_{\text{type},\bot} = j) \rightarrow \bigwedge_{m=0}^{|\beta_j|-1} (s^{(i+m)}_{\bot} = 0 \wedge t^{(i+m)}_{\text{type},\bot} = j \wedge l^{(i+m)} = \beta_{j,m}) \right) \\
&\quad \wedge \Big((s^{(i)}_{\top} = 1 \wedge t^{(i)}_{\text{type},\top} = j) \rightarrow (s^{(i+|\alpha_j|)}_{\top} = 1)\Big) \\
&\quad \wedge \Big((s^{(i)}_{\bot} = 1 \wedge t^{(i)}_{\text{type},\bot} = j) \rightarrow (s^{(i+|\beta_j|)}_{\bot} = 1)\Big) \\
Q_4 &:= \Big((p^{(i)} = 1) \rightarrow (t^{(i)}_{\text{id},\top} = 1) \wedge (t^{(i)}_{\text{id},\bot} = 1)\Big) \\
&\quad \wedge \Big(\neg(p^{(i)} = 1) \rightarrow (t^{(i)}_{\text{id},\top} = 0) \wedge (t^{(i)}_{\text{id},\bot} = 0)\Big)
\end{aligned}$$

□

Now we can define the entire first encoder:

Lemma 4.2.4. *Let P be some PCP instance. There is a transformer encoder layer $TEnc^P_{valid}$, such that for an input sequence $\boldsymbol{X}$, the check bit of all vectors in $\boldsymbol{Z} =$*

$TEnc^P_{valid}(\boldsymbol{X})$ are equal to zero if and only if the underlying input word, was a valid encoding for a possible solution to the PCP instance P.

Proof. Follows directly from Lemmas 4.2.2 and 4.2.3. □

4.2.6 Encoder 2: Enumerating the Tiles

To match the top and bottom tiles, we now need to assign a unique ID to each tile in the solution. To do this, we will use the strength of self-attention to count the number of start tiles seen so far at each position. In principle, this can be interpreted as numbering the tiles in the solution. Each position is then given the ID $\frac{1}{k}$, where k is the number of the tile to which that position belongs. In order to check the correct matching, we only have to check whether there is a tile ID at the bottom for each tile ID at the top, so that the two ids are the same and the tile type is identical.

Lemma 4.2.5. *There is a transformer encoder $TEnc_{enum}$ with two attention heads such that $TEnc_{enum}(\boldsymbol{X}) = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)$ and for all $\boldsymbol{z}_i$ hold: $t^{(i)}_{id,\top} = \frac{1}{k_i^\top}$ and $t^{(i)}_{id,\bot} = \frac{1}{k_i^\bot}$, where $k_i^\top, k_i^\bot$ are the number of upper or lower starting tiles which occured up to position i.*

Proof. Let $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ be the current representation of the input tokens, after passing the aggregation and validation layer. We will now construct a encoder layer with two attention heads $f^\top_{\text{att}}, f^\bot_{\text{att}}$ and parameters $\theta = (Q^\top_{\text{enum}}, K^\top_{\text{enum}}, V^\top_{\text{enum}}, Q^\bot_{\text{enum}}, K^\bot_{\text{enum}}, V^\bot_{\text{enum}}, W, N)$, which uses the threshold-scoring function $\text{score}_{\text{thres}}$ mentioned earlier. Let $Q^\top_{\text{enum}}$, $K^\top_{\text{enum}}$, $V^\top_{\text{enum}}$ be a linear maps with:

$$Q^\top_{\text{enum}}(\boldsymbol{z}_i) = \boldsymbol{q}_i^\top = \begin{pmatrix} 1 \\ \frac{1}{i+1} \\ 1 \end{pmatrix} \qquad Q^\bot_{\text{enum}}(\boldsymbol{z}_i) = \boldsymbol{q}_i^\bot = \begin{pmatrix} 1 \\ \frac{1}{i+1} \\ 1 \end{pmatrix}$$

$$K^\top_{\text{enum}}(\boldsymbol{z}_i) = \boldsymbol{k}_i^\top = \begin{pmatrix} \frac{1}{i+1} \\ -1 \\ s^{(i)}_\top \end{pmatrix} \qquad K^\bot_{\text{enum}}(\boldsymbol{z}_i) = \boldsymbol{k}_i^\bot = \begin{pmatrix} \frac{1}{i+1} \\ -1 \\ s^{(i)}_\bot \end{pmatrix}$$

$$V^\top_{\text{enum}}(\boldsymbol{z}_i) = \boldsymbol{v}_i^\top = \begin{pmatrix} t^{(i)}_{\text{id},\top} \\ 0 \end{pmatrix} \qquad V^\bot_{\text{enum}}(\boldsymbol{z}_i) = \boldsymbol{v}_i^\bot = \begin{pmatrix} 0 \\ t^{(i)}_{\text{id},\bot} \end{pmatrix}$$

We show that $f^\top_{\text{att}}(\boldsymbol{q}_i, \boldsymbol{K}, \boldsymbol{V}) = \frac{1}{k_\top}$. It follows by the definitions above that:

$$\text{score}_{\text{thres}}(\boldsymbol{q}_i^\top, \boldsymbol{k}_j^\top) = -\max\left(0, -\left(\frac{1}{j+1} - \frac{1}{i+1} + s^{(j)}_\top\right)\right)$$

We know that $s_\top^{(i)} \in \{0, 1\}$. We will use that to show:

$$\frac{1}{j+1} - \frac{1}{i+1} + s_\top^{(j)} = \begin{cases} \geq 1 & , (j \leq i) \wedge (s_\top^{(j)} = 1) \\ < 1 & , \text{otherwise.} \end{cases}$$

There are four cases to distinguish:

1. $j \leq i, s_\top^{(j)} = 0$. Then:
$$\underbrace{\frac{1}{j+1} - \frac{1}{i+1}}_{\in[0,1)} + 0 \in [0, 1)$$

2. $j \leq i, s_\top^{(j)} = 1$. Then:
$$\underbrace{\frac{1}{j+1} - \frac{1}{i+1}}_{\in[0,1)} + 1 \in [1, 2)$$

3. $j > i, s_\top^{(j)} = 0$. Then:
$$\underbrace{\frac{1}{j+1} - \frac{1}{i+1}}_{\in(-1,0)} + 0 \in (-1, 0)$$

4. $j > i, s_\top^{(j)} = 1$. Then:
$$\underbrace{\frac{1}{j+1} - \frac{1}{i+1}}_{\in(-1,0)} + 1 \in (0, 1)$$

Therefore, it follows that:

$$\text{score}(\boldsymbol{q}_i^\top, \boldsymbol{k}_j^\top) = \begin{cases} 0 & , (j \leq i) \wedge (s_\top^{(j)} = 1) \\ < 0 & , \text{otherwise.} \end{cases}$$

We can now calculate the output of the attention head $f^{\top}_{\text{att}}$.

$$\begin{aligned} \boldsymbol{s} &= \text{hardmax}\left(\text{score}(\boldsymbol{q}_i^{\top}, \boldsymbol{k}_1^{\top}), \ldots, \text{score}(\boldsymbol{q}_i^{\top}, \boldsymbol{k}_n^{\top})\right) \\ f^{\top}_{\text{att}}(\boldsymbol{q}_i, \boldsymbol{K}, \boldsymbol{V}) &= s_1 \begin{pmatrix} t^{(1)}_{\text{id},\top} \\ 0 \end{pmatrix} + \cdots + s_n \begin{pmatrix} t^{(n)}_{\text{id},\top} \\ 0 \end{pmatrix} \\ &= \frac{1}{k_i^{\top}} \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \cdots + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right) \qquad (4.1) \\ &= \begin{pmatrix} \frac{1}{k_i^{\top}} \\ 0 \end{pmatrix} \end{aligned}$$

Where $k_i^{\top} = |\{j \mid 1 \leq j \leq i \text{ and } s^{(j)}_{\top} = 1\}|$ is the number of positions left of i which are starting positions for upper tiles. Equation 4.1 comes from the fact that after the aggregation layer it is ensured that $t^{(1)}_{\text{id},\top} = 1, s^{(1)}_{\top} = 1$ and $t^{(j)}_{\text{id},\top} = 0$ for all $j > 1$. This shows the claim for $f^{\top}_{\text{att}}$. The proof for $f^{\bot}_{\text{att}}$ is analogous, by replacing all $\top$ with $\bot$ symbols.

Up to this point we have shown that for every input token $\boldsymbol{x}_i$:

$$\begin{aligned} \boldsymbol{a}_1^{(i)} &= f^{\top}_{\text{att}}(Q^{\top}_{\text{enum}}(\boldsymbol{x}_i), K^{\top}_{\text{enum}}(\boldsymbol{X}), V^{\top}_{\text{enum}}(\boldsymbol{X})) = \begin{pmatrix} \frac{1}{k_i^{\top}} \\ 0 \end{pmatrix} \\ \boldsymbol{a}_2^{(i)} &= f^{\bot}_{\text{att}}(Q^{\bot}_{\text{enum}}(\boldsymbol{x}_i), K^{\bot}_{\text{enum}}(\boldsymbol{X}), V^{\bot}_{\text{enum}}(\boldsymbol{X})) = \begin{pmatrix} 0 \\ \frac{1}{k_i^{\bot}} \end{pmatrix} \end{aligned}$$

We now define the linear map W_{enum}, which just sums the outputs of the two attention heads. Therefore,

$$W_{\text{enum}}([\![\boldsymbol{a}_1^{(i)}, \boldsymbol{a}_2^{(i)}]\!]) = \boldsymbol{a}_1^{(i)} + \boldsymbol{a}_2^{(i)} = \begin{pmatrix} \frac{1}{k_i^{\top}} \\ \frac{1}{k_i^{\bot}} \end{pmatrix}$$

The last part of the transformer encoder is a DNN N_{enum}, which gets the output of $W_{\text{enum}}([\![\boldsymbol{a}_1^{(i)}, \boldsymbol{a}_2^{(i)}]\!])$ as input and outputs a vector of dimension d_e. The new representation of the input token $\boldsymbol{z}_i$ is then:

$$\boldsymbol{z}_i = N_{\text{enum}}(W_{\text{enum}}([\![\boldsymbol{a}_1^{(i)}, \boldsymbol{a}_2^{(i)}]\!])) + \boldsymbol{x}_i$$

Through the residual connection we will add the information that has not been changed back to the representation. We add the values for the entries $t_{\text{id},\top}, t_{\text{id},\bot}$ to the correct position through the DNN and set all other entries of the vector to 0. The only problem here are the entries of the first token. According to the prerequisite, these are 1, but

the new representation must also be 1, because exactly one start tile was seen at the first position. To remove the effect of the residual connection for these tokens, we will construct the DNN in such a way that it returns 0 for input 1, and for inputs $\leq \frac{1}{2}$ the input is simply passed through. Let $f : \mathbb{R} \to \mathbb{R}$ the function with:

$$f(x) = \begin{cases} x & , x \leq \frac{1}{2} \\ -x + 1 & , \frac{1}{2} < x \leq 1 \\ 0 & , x > 1 \end{cases}$$

The function has the desired property and is piecewise-linear, hence computable by a neural network.

In summary the complete encoder layer $\text{TEnc}_{\text{enum}}$ only changes the entries $t_{\text{id},\top}, t_{\text{id},\bot}$ in each vector. Moreover, after passing the encoder the entries $t_{\text{id},\top}, t_{\text{id},\bot}$ contain a unique ID for each tile contained in the input word.

□

4.2.7 Encoder 3: Tile Matching

Lemma 4.2.6. *There is a transformer encoder $TEnc_{match}$ with three attention heads such that $TEnc_{match}(\boldsymbol{X}) = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)$ and for all $\boldsymbol{z}_i$ hold: $t^{(i)}_{check} = 0$ if and only if the input word was a correct matching of upper and lower tiles.*

Proof. Let $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ be the current representation of the input word, after passing encoder one and two. We will now construct an encoder layer with three attention heads $f^{\text{self}}_{\text{match}}, f^{\top}_{\text{match}}, f^{\bot}_{\text{match}}$ and parameters $\theta = (Q^*_{\text{match}}, K^*_{\text{match}}, V^*_{\text{match}}, W_{\text{match}}, N_{\text{match}})$, which use the max-scoring function mentioned earlier. Let Q^*_{match}, K^*_{match}, V^*_{match} be

linear maps with:

$$Q^{\top}_{\text{match}}(\boldsymbol{x}_i) = \boldsymbol{q}_i = \begin{pmatrix} t^{(i)}_{\text{id},\top} \\ -1 \\ 1 \end{pmatrix} \qquad Q^{\perp}_{\text{match}}(\boldsymbol{z}_i) = \boldsymbol{q}_i = \begin{pmatrix} 1 \\ \frac{1}{i+1} \\ 1 \end{pmatrix}$$

$$K^{\top}_{\text{match}}(\boldsymbol{x}_i) = \boldsymbol{k}_i = \begin{pmatrix} 1 \\ t^{(i)}_{\text{id},\perp} \\ s^{(i)}_{\perp} - 1 \end{pmatrix} \qquad K^{\perp}_{\text{match}}(\boldsymbol{z}_i) = \boldsymbol{k}_i = \begin{pmatrix} \frac{1}{i+1} \\ -1 \\ s^{(i)}_{\perp} \end{pmatrix}$$

$$V^{\top}_{\text{match}}(\boldsymbol{z}_i) = \boldsymbol{v}_i = [\![\mathbf{0}_4, t^{(i)}_{\text{id},\perp}, t^{(i)}_{\text{type},\perp}, 0, 0]\!] \qquad V^{\perp}_{\text{match}}(\boldsymbol{z}_i) = \boldsymbol{v}_i = [\![\mathbf{0}_4, 0, 0, t^{(i)}_{\text{id},\top}, t^{(i)}_{\text{type},\top}]\!]$$

$$Q^{\text{self}}_{\text{match}}(\boldsymbol{x}_i) = \boldsymbol{q}_i = \begin{pmatrix} i \\ -1 \end{pmatrix} \qquad K^{\text{self}}_{\text{match}}(\boldsymbol{x}_i) = \boldsymbol{q}_i = \begin{pmatrix} 1 \\ i \end{pmatrix}$$

$$V^{\text{self}}_{\text{match}}(\boldsymbol{z}_i) = \boldsymbol{v}_i = [\![t^{(i)}_{\text{id},\top}, t^{(i)}_{\text{type},\top}, t^{(i)}_{\text{id},\perp}, t^{(i)}_{\text{type},\perp}, \mathbf{0}_4]\!]$$

The first attention head $f^{\text{self}}_{\text{match}}$, collects the ID and type of the tile at some position and the other two attention heads $f^{\top}_{\text{match}}, f^{\perp}_{\text{match}}$ collect the information of the respecting upper and lower start tile with the same ID. Then the information gets aggregated together by a linear projection W_{match} and the neural network, checks if the ids and tile types match. We first show, that the first attention head $f^{\text{self}}_{\text{match}}$ collects the information of the position itself.

$$\text{score}_{\max}(\boldsymbol{q}_i, \boldsymbol{k}_j) = -|\boldsymbol{q}_i, \boldsymbol{k}_j| = -|i-j| = \begin{cases} 0 & , i = j \\ < 0 & , \text{otherwise.} \end{cases}$$

Therefore, it follows directly that:

$$f^{\text{self}}_{\text{match}}(\boldsymbol{q}_i, \boldsymbol{K}, \boldsymbol{V}) = \boldsymbol{a}^{(i)}_{\text{self}} = [\![t^{(i)}_{\text{id},\top}, t^{(i)}_{\text{type},\top}, t^{(i)}_{\text{id},\perp}, t^{(i)}_{\text{type},\perp}, \mathbf{0}_4]\!]$$

This easily shows, that the attention head collects the ID and type information of the position itself, for every position i. For the attention head $f^{\top}_{\text{match}}$, it holds:

$$\text{score}_{\max}(\boldsymbol{q}_i, \boldsymbol{k}_j) = -|t^{(i)}_{\text{id},\top} - t^{(j)}_{\text{id},\perp} + s^{(j)}_{\perp} - 1| = \begin{cases} 0 & , t^{(i)}_{\text{id},\top} = t^{(j)}_{\text{id},\perp} \wedge s^{(j)}_{\perp} = 1 \\ < 0 & , \text{otherwise.} \end{cases}$$

The score is therefore maximum if and only if the position j being scored against has the same lower tile ID and is a starting tile. It follows that this attention head calculates the following:

$$f^{\top}_{\text{match}}(\boldsymbol{q}_i, \boldsymbol{K}, \boldsymbol{V}) = \boldsymbol{a}^{(i)}_{\top} = [\![\mathbf{0}_4, t^{(j)}_{\text{id},\perp}, t^{(j)}_{\text{type},\perp}, 0, 0]\!]$$

where j is the position described above. The same applies analogously to the attention

head $f^{\perp}_{\text{match}}$ with

$$f^{\perp}_{\text{match}}(\boldsymbol{q}_i, \boldsymbol{K}, \boldsymbol{V}) = \boldsymbol{a}^{(i)}_{\perp} = [\![\mathbf{0}_4, 0, 0, t^{(k)}_{\text{id},\top}, t^{(k)}_{\text{type},\top}]\!]$$

As mentioned above, the linear map W_{match} just sums the output of all three attention heads together:

$$W_{\text{match}}(\boldsymbol{a}^{(i)}_{\text{self}}, \boldsymbol{a}^{(i)}_{\top}, \boldsymbol{a}^{(i)}_{\perp}) = [\![\boldsymbol{a}^{(i)}_{\text{self}}, \boldsymbol{a}^{(i)}_{\top}, \boldsymbol{a}^{(i)}_{\perp}]\!] = \begin{pmatrix} t^{(i)}_{\text{id},\top} \\ t^{(i)}_{\text{type},\top} \\ t^{(i)}_{\text{id},\perp} \\ t^{(i)}_{\text{type},\perp} \\ t^{(j)}_{\text{id},\perp} \\ t^{(j)}_{\text{type},\perp} \\ t^{(k)}_{\text{id},\top} \\ t^{(k)}_{\text{type},\top} \end{pmatrix}$$

This vector is now given into the DNN N_{match}, we define the DNN via a multiplexer program P_{match} as follows:

$$P_{\text{match}} := t^{(i)}_{\text{id},\top} = t^{(j)}_{\text{id},\perp} \wedge t^{(i)}_{\text{id},\perp} = t^{(k)}_{\text{id},\top} \wedge t^{(i)}_{\text{type},\top} = t^{(j)}_{\text{type},\perp} \wedge t^{(i)}_{\text{type},\top} = t^{(j)}_{\text{type},\perp}$$

We translate P_{match} into a DNN and modify it to output a vector of dimension d_e and write the output of P_{match} to the check bit. By the resdiual connection we then get as the output of the whole encoder at position i:

$$\boldsymbol{z}_i = N_{\text{match}}([\![\boldsymbol{a}^{(i)}_{\text{self}}, \boldsymbol{a}^{(i)}_{\top}, \boldsymbol{a}^{(i)}_{\perp}]\!]) + \boldsymbol{x}_i$$

Therefore, the encoder changes at most only the check bit of a position i in case of an incorrect matching. □

4.2.8 Final Classification

The last thing we need to do is define the neural network, which takes the sum of all the terminating vectors and then classifies them. As already mentioned, at the end we only have to check the check bits of the vectors. If all of these bits are zero, then the value at the position of the check bit is also zero after the sum. If, however, a check has failed, then the sum at the end is not equal to zero. It follows that the DNN should classify the vector at the end as zero if the value at the position of the check bit is not equal to zero or as one if the value is zero. Let N_{class} be this neural network. It can be defined as

follows:

$$N_{\text{class}}(\boldsymbol{y}) = \text{rel}(-y_{\text{check}} + 1) - \text{rel}(0, y_{\text{check}} - 1) = 1 - y_{\text{check}}$$

If we now set the classification threhsold to one, we get the desired behaviour.

4.2.9 Correctness

Now that we have defined everything necessary, we can come to the main result of this chapter.

Theorem 4.2.7. *For every PCP instance P there is a transformer sequence classifier T_P, such that:*

$$P \text{ is solvable if and only if there is a word } w \in \Sigma^*, \text{ such that } T_P(w) = 1$$

Proof. Let $P = ((\alpha_1, \beta_1), \ldots, (\alpha_k), \beta_k)$ be a PCP instance. Let T_P be the transformer sequence classifier as defined in the previous sections.

- $\rightarrow$: Assume P is solvable. Then there is a solution $\boldsymbol{s} = (i_1, \ldots, i_l)$ with $l \geq 1$, such that $\alpha_{i_1} \cdots \alpha_{i_l} = \beta_{i_1} \cdots \beta_{i_l}$. Let $f_{\text{conv}}(\boldsymbol{s}) = w_{\boldsymbol{s}}$ be the encoded solution as defined in section 4.2.1. Moreover, let $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ be the input sequence after the word embedding and positional encoding as defined in Lemma 4.2.1 and Section 4.2.3. By Lemma 4.2.4 the output of the first encoder is $\text{TEnc}^P_{\text{valid}}(\boldsymbol{X}) = \boldsymbol{X}$, because $\boldsymbol{X}$ is by assumption a valid encoding of a solution to P. Therefore, the check bit of the input sequence has not been changed. The second encoder layer defined as in Lemma 4.2.5 enumerates the upper and lower tiles. The upper and lower enumeration of the tiles must match, because the input was a valid encoding of a solution. Therefore, the third encoder as defined in Lemma 4.2.6, does not change the check bit of any position in the sequence. It follows that the output after all encoder layers, has not changed the check bit of any position and therefore, the input word gets accepted by the last classification layer.
- $\leftarrow$: Assume P is not solvable. Let $w \in \Sigma^*$ be some input word. There are two cases. Either w is a syntactically correct looking encoding of a upper and lower tile sequence with respect to Section 4.2.1 or w does not adhere to the encoding scheme. If w does not adhere to the encoding scheme, the validation happening in the first encoder, will alter the check bit of some positions in the input sequence. This will result in a false classification at the final classification. In the other case the input sequence correctly encodes a possible solution, where the upper and lower tile sequence encode the same word and matches the underlying PCP instance, but

the upper and lower tiles are mismatched. This means that at some position in the input sequence, the tile type at the start of an upper tile does not match the tile type at the respecting lower tile. This is checked in encoder layer 3 as described in Lemma 4.2.6. Therefore, the output classification of such an input sequence is also false.

□

Corollary 4.2.8. *The output reachability problem for transformer sequence classifiers is undecidable.*

4.3 Bounded Output Reachability

As we saw in the last section, the TransReach problem is undecidable. In the following, we consider a restricted version of this problem. We restrict the maximum length of the input word. This restriction makes the problem trivially decidable, because now all possible inputs can be tested. However, we want to find a characterisation in terms of computational complexity of this problem. In doing so, we will use the reduction from the last section to show the NP-completeness of this problem. We use a variant of the PCP that also limits the maximum length of a solution. This problem is known to be NP-complete. We show that the reduction from the last section is computable in polynomial time and that the constrained TransReach problem is in NP. Thus we conclude the NP-completeness of the bounded TransReach problem.

Definition 4.3.1 (Bounded Output Reachability Problem k-TransReach)**.** Given a transformer sequence classifier T and a $k \geq 1$ in unary. Is there a word $w \in \Sigma^*$ with $|w| \leq k$, such that T accepts w?

Definition 4.3.2 (Bounded PCP k-PCP)**.** Given an instace of the PCP problem $P = ((\alpha_1, \beta_1), \ldots, (\alpha_k, \beta_k))$ and $k \geq 1$. Is there a solution to P $\boldsymbol{s} = (i_1, \ldots, i_l)$, such that $|\alpha_{i_1} \cdots \alpha_{i_l}| \leq k$?

Proposition 4.3.3 (Garey and Johnson [GJ79])**.** *k-PCP is NP-complete.*

Theorem 4.3.4. *The reduction from k-PCP to k-TransReach is computable in polynomial time.*

Proof. By revisiting the reduction from the previous section, one can see that the only parts of the transformer, which are dependent on the PCP instance P are the DNN for the word embedding N_{emb} from Lemma 4.2.1 and the DNN for checking the valid encoding N_{valid} from Lemma 4.2.3.

The size of N_{emb} is linear in the size of the input alphabet Σ, because the input into the DNN is a one-hot encoding of the symbol. The size of the input alphabet is linear in size of the PCP instance.

We defined the DNN from Lemma 4.2.3 as a multiplexer program. On closer inspection one can see that the size of the formula is at most quadratic in the size of the PCP instance, because for each tile type, the corresponding symbols of the tile have to be defined. By Observation 4.0.6 the DNN is of linear size in the size of the formula.

Therefore, the reduction can be computed in polynomial time. □

Theorem 4.3.5. *The bounded output reachability problem for sequence classification transformers is in NP.*

Proof. Let T be a transformer sequence classifier and $k \geq 1$. Existence of a solution can be decided as follows. Guess a word $w \in \Sigma^*$ with length $\leq k$. Then compute the output $T(w)$ of the transformer. The length of w is of linear size in k. The computation of the output can be done in polynomial time. This is, because the computations of the neural networks and linear maps in T can be done in linear time $\mathcal{O}(|T|)$. The self-attention can be computed in $\mathcal{O}(|w|^2)$, because each positions has to be scored against each other position. This has to be done for every attention head in every encoder layer. In summmary this gives us polynomial time $\mathcal{O}(|w|^2 \cdot |T| \cdot a)$ in summary, where a is the total number of attention heads in T. □

Corollary 4.3.6. *k-TransReach is NP-complete.*

5 TransReach for Transformers with Limited Precision

Having demonstrated in the preceding chapter that transformer sequence classifiers possess an undecidable output reachability problem, this chapter aims to explore a constraint on the model in order to make the verification problem decidable. Essentially, two main factors can be identified that lead to the undecidability in the last chapter. Firstly, the arbitrary length of the input words, and secondly, the need for arbitrary precision in the internal computations of the transformer. Restricting the word length makes the TransReach problem easily decidable as all possible words can be tested. As shown in Section 4.3 this problem is NP-complete. However, arbitrary precision allows a position in the input sequence to attend to positions that are arbitrarily far away. By limiting the precision, a position cannot reference positions that are arbitrarily far away, because due to the small amount of representable numbers, positions can no longer have unique ids up to any length. While this gives us an indication that such transformers may have less power, it is not apparent whether the output reachability problem becomes decidable with this restriction.

In the following, we first define what we mean by transformers with limited precision and why these models are interesting in practice. Then we discuss what effects the precision constraint has on the power of the transformer and why the decidability of TransReach over limited precision presumably follows from this. Afterwards, first approaches of a decidability proof are presented.

5.1 Limited Precision

The practical relevance of limited precision transformers is obvious, as the assumption of arbitrary precision is unrealistic in practice. The hardware used to apply transformer models always operates with limited precision. Most computers employ floating-point arithmetic, where each number represented by a bit string of finite bit width p. It is apparent that computers can only represent a finite set of numbers using such a representation. The question that naturally arises is to what extent this limits the capabilities

of the transformer. This work solely focuses on floating point representations of numbers. We refer to the set of p precision numbers as $\mathbb{D}_p$. The amount of numbers that can be represented with p precision is obviously exponential in p, $|\mathbb{D}_p| \in O(2^p)$. Floating-point arithmetic operations work as usual, but results outside the representable space are rounded to the nearest representable number. When referring to p-precision transformers, this implies transformers that operate on $\mathbb{D}_p$ using floating-point arithmetic. Transformers with limited precision have already been investigated in terms of expressibility [MS23a] [CCP23]. In order to obtain a decidability result regarding the output reachability problem for transformers with limited precision, we first have to investigate how finite precision affects the different components of the transformer.

Positional Encoding. At the word level, additional positional information is added to each position after word embedding. This positional information allows regular transformers to address specific positions and aggregate the information. When limited precision is used, these positional information cannot be unique for every possible input length. It follows that vectors must repeat in embeddings of words, that exceed a certain length.

Self-Attention. During the self-attention mechanism the transformer cannot see the order in which the input sequence is given, because the calculations are done independently for each position. The only information that is taken into consideration is the positional information added through the positional encoding. As explained above, in transformers with limited precision this positional information is no longer unique for each position. But this means that the calculations done for two positions with the exact same embedding, are also the same. Hence the output from the self-attention for these two vectors must be equal. This observation was formally proven by Perez et al. [PBM21].

Proposition 5.1.1 (Perez et al. [PBM21])**.** *Let TEnc be some transformer encoder layer and let* $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ *be an input sequence. Furthermore, let* $\boldsymbol{Z} = TEnc(\boldsymbol{X})$*. Then for every pair of indices* $(i, j) \in [n] \times [n]$ *the following holds: if* $\boldsymbol{x}_i = \boldsymbol{x}_j$ *then* $\boldsymbol{z}_i = \boldsymbol{z}_j$*.*

Self-Attention can also be used to count the number of items with certain properties. This works through the interaction of scoring and normalisation in each attention head. Scoring is used to check the respective property and averaging in normalisation counts the occurrences. However, with limited precision transformers, it is not possible to count indefinitely, as only a finite amount of representable numbers is available. Merrill and Sabharwal [MS23a] have shown another property that limits the counting range of the

self-attention depending on the available precision. In particular, they proved that transformers with finite precision are much less expressive, since only a constant number of other positions can be attended to during self-attention. The constant depends on the precision.

Proposition 5.1.2 (Merrill and Sabharwal [MS23a]). *Let $\boldsymbol{a} \in \mathbb{R}^n$ with $\sum_{i=1}^{n} a_i = 1$ and $\widetilde{\boldsymbol{a}}$ be the nearest p-precision approximation to $\boldsymbol{a}$. Then:*

1. *$\widetilde{\boldsymbol{a}}$ has at most 2^{2^p} many nonzero entries.*
2. *If $n > 2^{2^p}$ and $\boldsymbol{a}$ is uniform $\left(a_i = \frac{1}{n}\right)$, then $\widetilde{\boldsymbol{a}} = \mathbf{0}$.*

It follows for any p-precision transformer that if the length of the input word is greater than 2^{2^p}, the self-attention of each position cannot depend on all other position. In the special case of uniform normalisation like hardmax, the scoring vector even gets to zero, which means that the output for this position is the zero vector. We will first address decidability under the assumption of hardmax as a normalisation function, as this case is easier to consider for the time being. In the case of general normalisation functions, more work has to be done, as we will see later, because it has to be calculated beforehand which entries become zero during normalisation.

5.2 Capturing the Behaviour of Self-Attention

We will now use these results to develop a first approach to the decidability result. So far, we have seen two limitations of transformers with limited precision related to the self-attention mechanism. Firstly, the transformer can no longer distinguish positions in the input up to any length and secondly, the possibilities of averaging are also limited by the limited precision and thus the ability of the transformer to count certain positions. In the following, we will take advantage of these two limitations to reduce the behaviour of the self-attention mechanism to finitely many different cases. We show that for all longer inputs, they behave similarly to shorter inputs. Specifically, we want to find an upper bound on the length of the input sequence that we need to look at, so that longer input sequences behave equivalently to shorter input sequences. From Proposition 5.1.1, we have already identified $|\mathbb{D}_p|$ as a lower bound on this word length, since up to this length, the transformer can distinguish positions with precision p. But as we have seen, an attention head can not only distinguish positions, but also count. The counting threshold is double-exponential in the precision p. If this counting threshold is exceeded, the output of the attention head for this position is by Proposition 5.1.2 is zero. If it is true for an input sequence that each position attends to more positions than the counting

threshold is large, the output for all positions is the zero vector. In summary, the output for an input sequence $\boldsymbol{X}$ depends on the following factors:

- The set of disjunct vectors vals($\boldsymbol{X}$) occuring in $\boldsymbol{X}$
- The set of vectors $A_{\boldsymbol{v}}$ each vector $\boldsymbol{v}$ in vals($\boldsymbol{X}$) attends to
- The number of occurences of each vector in vals($\boldsymbol{X}$)

Let f_{att} be some p precision transformer encoder attention head with linear maps Q, K, V. Let d_Q be the dimension of the query vectors and d_V be the dimension of the value vectors. Furthermore, let score : $\mathbb{D}_p^{d_Q} \times \mathbb{D}_p^{d_Q} \to \mathbb{D}_p$ be the scoring function used in the attention head. We introduce the notion of the attention map $m_{\text{att}} : \mathbb{D}_p^{d_Q} \times \mathcal{P}(\mathbb{D}_p^{d_Q}) \to \mathcal{P}(\mathbb{D}_p^{d_Q})$. For some input sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ with vals($\boldsymbol{X}$) $= M$, $m_{\text{att}}(\boldsymbol{v}, M) = A$ is the set of vectors, the vector $\boldsymbol{v}$ attends to, when M is the set of other vectors occuring in the input sequence $\boldsymbol{X}$.

$$m_{\text{att}}(\boldsymbol{v}, M) = \underset{\boldsymbol{w} \in M}{\operatorname{argmax}} \operatorname{score}(\boldsymbol{v}, \boldsymbol{w})$$

We need the Attention Map to determine for an input sequence whether or not it already exceeds the Counting Threshold for all positions. We define $h(M, \boldsymbol{X})$ as the frequency of occurrences of vectors of M in $\boldsymbol{X}$. We use this notation in order to formally prove that it suffices to only consider input sequences where each position attends to at most $2^{2^p} + 1$ positions.

Lemma 5.2.1. *Let m_{att} be an attention map of a transformer attention head f_{att}, $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ be an input sequence and its corresponding output $f_{att}(\boldsymbol{X}) = \boldsymbol{Z} = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)$. If $m_{att}(\boldsymbol{x}_i, vals(\boldsymbol{X})) = A$ and $h(A, \boldsymbol{X}) > 2^{2^p}$ for some $i \in [n]$, then $\boldsymbol{z}_i = \boldsymbol{0}$.*

Proof. This Lemma is a direct consequence of Propositions 5.1.1 and 5.1.2. □

The idea is to characterise the behaviour of an attention head for each possible input sequence by increasing the frequencies for each possible set $M \subseteq \mathbb{D}_p$ of occurring vectors until each position attends to $2^{2^p}+1$ other positions. It is important to note that the order of the input sequence does not matter for the attention head. Order will only become important when we look for input words that map to specific sequences. The worst case is that every vector only attends to itself. In this case we have to consider all input sequences up to length $|\mathbb{D}_p| \cdot 2^{2^p}$. Every input sequence which is not considered, has at least one positions which attends to more than $2^{2^p} + 1$ other positions. Therefore, with Lemma 5.2.1 it follows that this position will be mapped to the zero vector. The complexity of calculating all these input sequences and their corresponding output sequences is immense. We have to calculate the output for all input sequences up to length $|\mathbb{D}_p| \cdot 2^{2^p}$

in the worst case. The computational cost per input sequences is quadratic in the length of the input sequences, because during scoring, each position is scored against each other positions. So the complexity of calculating all possible sequences is:

$$|\mathbb{D}_p^{d_Q}|^{|\mathbb{D}_p^{d_Q}|\cdot 2^{2^p}} \cdot \mathcal{O}((|\mathbb{D}_p^{d_Q}| \cdot 2^{2^p})^2)$$

d_Q is a constant coming from the transformer and with $\mathbb{D}_p \in \mathcal{O}(2^p)$ we get:

$$2^{\mathcal{O}(p)\cdot 2^{\mathcal{O}(p)}\cdot 2^{2^p}} \cdot \mathcal{O}((2^{\mathcal{O}(p)} \cdot 2^{2^p})^2) \in \mathcal{O}\left(2^{2^{2^p}}\right)$$

which gives us triple-exponential computational cost.

5.3 Discussion of Further Steps

In the following, we will discuss how this result for individual attention heads can be extended to the other components of the transformer in order to decide the output reachability problem. The next largest component of the transformer is an encoder layer. As we have just seen, we can precompute the behaviour of the individual attention heads on all input sequences. The output sequences of the individual attention heads are then combined with each other. However, since the individual attention heads score individual positions differently with each other, two input sequences that are similar with respect to one attention head are not necessarily similar with respect to the others. Therefore, to characterise the behaviour of an entire encoder, all input sequences must be considered up to a limit where each of the vectors occurring attends to more than 2^{2^p} of other positions in each attention head. This has to be done for all encoder layers. After we have captured the input output behaviour of all encoder layers we have to determine what kind of input sequences output one in the last classification layer. This can be done inductively in reverse order from classification to word embedding. The first step is to calculate all vectors $\mathbb{F} \subseteq \mathbb{D}_p^d$ which the final DNN classifier accepts. Naively, this can be done by simply testing all possible inputs from $\mathbb{D}_p^d$. Next we consider the last encoder layer of the transformer. We have to check for all input sequences, whether the sum of the output sequence belongs to $\mathbb{F}$. This step is tricky, because we also have to consider sequences exceeding the counting treshold. For input sequences in which no vector attends to more than 2^{2^p} position, this is not an issue. But for every input sequence $\boldsymbol{X}$ in which $T_{\boldsymbol{X}}$ is the set of vectors attending to more than 2^{2^p} positions, we know, that adding vectors from $T_{\boldsymbol{X}}$ to the sequence, does not change the output of the corresponding vector. Then the question is, if there is a function $f_{\text{ext}} : T_{\boldsymbol{X}} \to \mathbb{N}$ such that if $\boldsymbol{X}$ is extended by vectors in $T_{\boldsymbol{X}}$ with frequencies given by f_{ext} the sum of the output

sequence is in $\mathbb{F}$. There may be infinitely many possibilities for such a function, but one can show that the frequencies repeat periodically. After having determined all such input sequences into the last encoder layer, which the final DNN classifier accepts, we just have to propagate these input sequence backwards through the remaining encoder layers. This means we check for every possible input sequence, if this sequence is in the set of possible output sequences of the previous encoder layer. After we done that, we have a set of possibly infinite many input sequences, which are accepted by the transformer. The last step is to match these input sequences to the corresponding input words, while also take the positional encoding into consideration. This requires further investigation of positional encoding and more pigeonhole arguments to map the correct input words with the sequences.

In summary, in this chapter we have taken the first steps towards decidability of the TransReach problem for p-precision transformers. Although, this problem is most likely decidable, the practical application of this algorithm is very questionable. Already for the first decidability result for the self-attention mechanism we have triple-exponential complexity. This is not practical in reality for models with many parameters.

6 Conclusion

In this thesis, the output reachability problem for transformer sequence classifiers was investigated. First, the undecidability of the problem was shown. On closer analysis, two main factors were found to be the cause of the undecidability. The first is that the input to a transformer can be arbitrarily long, and the second is that the transformer can compare positions in the input that are arbitrarily far apart and can also count indefinitely. If either of these capabilities is restricted, the problem becomes decidable. It has been shown that the problem is NP-complete when the input length is bounded. Then, in the last chapter, a first sketch of a proof of decidability was given under the assumption that the transformer works with limited precision. With this restriction, the input can still be arbitrarily long, but the ability to count and distinguish positions is severely limited.

Further Research

While this work has only provided a first insight into the computational complexity of verifying transformer models, there are several interesting next steps for research.

- While proving undecidability, we made several non-standard assumptions. Most importantly, we used hard attention and an improved scalar product equipped with a DNN for scoring. It would be interesting to investigate whether this result can be extended to soft attention or the standard scalar product for scoring.
- The investigation of the adversarial robustness verification problem discussed in Chapter 3. This problem is much more relevant from a practical point of view. There are already approximate algorithms for this problem, but finding a sound and complete algorithm and determining the computational complexity could be a very interesting task.
- The use of hardmax normalisation and the advanced scalar product could also be analysed in terms of expressiveness. Intuitively, it seems that the advanced scalar product is somewhat better able to attend to certain positions than the standard. This could be investigated both formally and experimentally. This would also give us more insights from a linguistic or language-theoretic perspective, in

terms of understanding what makes transformer models so capable of capturing the semantics of natural languages.

- Another interesting direction would be to develop more practical algorithms for verifying transformer models. A very successful approach from the verification of neural networks is SMT solving [KBD+17]. This technique has proven to be quite scalable to very large neural networks.
- As we have only considered sequence classifiers in this paper, sequence-to-sequence models such as translator and generative models are also very common. Verification problems for these models include avoiding abusive, biased or inappropriate language in the output. It would be interesting to find formal definitions for these problems and to study them from a theoretic perspective.
- The previous point could also be interesting in terms of whether transformer models can be trained to avoid specific behaviour. This robustness by design has already been worked on in terms of word embeddings, but it would also be interesting to develop algorithms for general training of transformer models.
- The study of limited precision transformers should be extended. Although these transformer models are much weaker in terms of power, they are much closer to the hardware on which these models are used in practice. The verification of these transformers seems to be easier. Limited precision can also be motivated by recent research on quantized neural networks [DKSL20]. It has been shown that low-precision neural networks are less vulnerable to adversarial attacks than full-precision neural networks. There has also been work to do verification of quantized neural networks with SMT solving [BHL+20]. It could be investigated if this framework could also be applied to transformer models.

Bibliography

[BCB16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate, May 2016. arXiv:1409.0473, doi:10.48550/arXiv.1409.0473.

[BDBV21] Gregory Bonaert, Dimitar I. Dimitrov, Maximilian Baader, and Martin Vechev. Fast and precise certification of transformers. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 466–481, New York, NY, USA, June 2021. Association for Computing Machinery. doi:10.1145/3453483.3454056.

[BHL+20] Marek Baranowski, Shaobo He, Mathias Lechner, Thanh Son Nguyen, and Zvonimir Rakamarić. An SMT Theory of Fixed-Point Arithmetic. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, Lecture Notes in Computer Science, pages 13–31, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-51074-9.

[BMR+20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[BTT+18] Rudy Bunel, Ilker Turkaslan, Philip H.S. Torr, Pushmeet Kohli, and M. Pawan Kumar. A unified view of piecewise linear neural network verification. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 4795–4804, Red Hook, NY, USA, December 2018. Curran Associates Inc. doi:10.48550/arXiv.1711.00455.

[CCP23] David Chiang, Peter Cholak, and Anand Pillay. Tighter Bounds on the Expressivity of Transformer Encoders. In *Proceedings of the 40th International Conference on Machine Learning*, pages 5544–5562. PMLR, July 2023.

[CKB+21] Karl Cobbe, V. Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and J. Schulman. Training Verifiers to Solve Math Word Problems. *ArXiv*, October 2021.

[DKSL20] Kirsty Duncan, Ekaterina Komendantskaya, Robert Stewart, and Michael Lones. Relative Robustness of Quantized Neural Networks Against Adversarial Attacks. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2020. doi:10.1109/IJCNN48605.2020.9207596.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. A Series of Books in the Mathematical Sciences. W. H. Freeman & Company, New York, 1979.

[HAF22] Yiding Hao, Dana Angluin, and Robert Frank. Formal Language Recognition by Hard Attention Transformers: Perspectives from Circuit Complexity. *Transactions of the Association for Computational Linguistics*, 10:800–810, July 2022. doi:10.1162/tacl_a_00490.

[HCJ+19] Yu-Lun Hsieh, Minhao Cheng, Da-Cheng Juan, Wei Wei, Wen-Lian Hsu, and Cho-Jui Hsieh. On the Robustness of Self-Attentive Models. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1520–1529, Florence, Italy, 2019. Association for Computational Linguistics. doi:10.18653/v1/P19-1147.

[JRGL19] Robin Jia, Aditi Raghunathan, Kerem Göksel, and Percy Liang. Certified Robustness to Adversarial Word Substitutions. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4129–4142, Hong Kong, China, November 2019. Association for Computational Linguistics. doi:10.18653/v1/D19-1423.

[KBD+17] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 97–117, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-63387-9_5.

[KGN21] Rohit Kumar Kaliyar, Anurag Goswami, and Pratik Narang. FakeBERT: Fake news detection in social media with a BERT-based deep learning approach. *Multimedia Tools and Applications*, 80(8):11765–11788, March 2021. doi:10.1007/s11042-020-10183-2.

[LLG+20] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online, 2020. Association for Computational Linguistics. doi:10.18653/v1/2020.acl-main.703.

[MNO20] Raymond T Mutanga, Nalindren Naicker, and Oludayo O. Hate Speech Detection in Twitter using Transformer Methods. *International Journal of Advanced Computer Science and Applications*, 11(9), 2020. doi:10.14569/IJACSA.2020.0110972.

[MS23a] William Merrill and Ashish Sabharwal. A Logic for Expressing Log-Precision Transformers, May 2023. arXiv:2210.02671.

[MS23b] William Merrill and Ashish Sabharwal. Transformers Can Be Expressed In First-Order Logic with Majority, January 2023. arXiv:2210.02671, doi:10.48550/arXiv.2210.02671.

[PBM21] Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is turing complete. *The Journal of Machine Learning Research*, 22(1):75:3463–75:3497, January 2021.

[PCG21] Aditya Prakash, Kashyap Chitta, and Andreas Geiger. Multi-Modal Fusion Transformer for End-to-End Autonomous Driving. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7073–7083, June 2021. doi:10.1109/CVPR46437.2021.00700.

[Pos46] Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946. doi:10.1090/S0002-9904-1946-08555-9.

[SL21] Marco Sälzer and Martin Lange. Reachability is NP-Complete Even for the Simplest Neural Networks. In Paul C. Bell, Patrick Totzke, and Igor Potapov, editors, *Reachability Problems*, Lecture Notes in Computer Science, pages 149–164, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-89716-1_10.

[SZC+19] Zhouxing Shi, Huan Zhang, Kai-Wei Chang, Minlie Huang, and Cho-Jui Hsieh. Robustness Verification for Transformers. In *International Conference on Learning Representations*, September 2019.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 6000–6010, Red Hook, NY, USA, December 2017. Curran Associates Inc.

[WGL19] Dilin Wang, Chengyue Gong, and Qiang Liu. Improving Neural Language Modeling via Adversarial Training. In *International Conference on Machine Learning*, May 2019.

[Wil97] A. J. Wilkie. Schanuel's Conjecture and the Decidability of the Real Exponential Field. In Bradd T. Hart, Alistair H. Lachlan, and Matthew A. Valeriote, editors, *Algebraic Model Theory*, pages 223–230. Springer Netherlands, Dordrecht, 1997. doi:10.1007/978-94-015-8923-9_11.

[YSPK21] Jaemin Yoo, Yejun Soun, Yong-chan Park, and U Kang. Accurate Multivariate Stock Movement Prediction via Data-Axis Transformer with Multi-Level Contexts. *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2037–2045, August 2021. doi:10.1145/3447548.3467297.

[ZBH+21] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, February 2021. doi:10.1145/3446776.