Utrecht university
Faculty of Humanities
Department of linguistics

Utrecht University

MSc Thesis

On how transformers learn to understand and evaluate nested arithmetic expressions

Daan Grashoff
6977189

Time frame: December 2021

Supervisor
Dr. Denis Paperno

Second Supervisor
Dr. Meaghan Fowlie

Abstract

In this thesis, we studied whether self-attention networks can learn compositional semantics using an arithmetic language. The goal of language aims to evaluate the meaning of nested expressions. We find that self-attention networks can learn to evaluate these nested expressions by taking shortcuts on less complex expressions or utilizing deeper layers on complex expressions when the nested depth grows. The complexity is in whether expressions are left- (easy) or right-branching (hard) and whether, in the case of right-branching expressions, plus (easy) or minus (complex) operators are used. We find that increasing the number of heads does not always help with more complex expressions, whereas the number of layers does always help to generalize to deeper expressions. Finally, to help with the understanding of what the self-attention networks are doing, we analyzed the attention scores and found exciting patterns such as the numbers attending to the preceding operators and nested sub-expressions attending to preceding operators. These patterns may explain why in less complex expressions, the self-attention networks take shortcuts, but in more complex expressions, this is not possible by the way the self-attention networks try to solve them.

Contents

1 Introduction

One of the main topics in artificial intelligence research is the ability of machines to learn and understand the meaning of language. Language is everywhere, communication between humans residing in spoken text, documents, and many more. The ability for machines to understand language and work with it could be beneficial for all kinds of tasks, e.g., Sentiment Analysis, Questioning and Answering, Entity Recognition, and machine translation. Language can be complex since it contains hierarchical compositional semantics, which means that the meaning of a sentence is determined by combining the meanings of its words and sub-phrases, following a set of rules driven by the syntactic structure. For example, "dog bites man" and "man bites dog," which is a combination of the meaning of the words "man, dog, bites," and depending on the order, the meaning changes to something that happens all the time, or something happens rarely; like "man bites dog."

Humans can understand these concepts well and follow along with a sentence that might contain complex compositions of words. For example, the meaning of "It is not that it is not raining today," which contains a double negation about raining today, boils down to the meaning that it is indeed raining today. Of course, such complex compositions are not in all sentences, but a machine should be able to capture these compositions too, for example, correctly capture the sentiment of a text.

There has been much work done in the field of natural language processing (NLP), where Neural networks have been prominent in this field for a long time. Tasks in NLP can be seen as a sequence processing task where a sentence is a sequence of words and a word a sequence of characters. One of the architectures to do this is the Recurrent neural network (RNN) [17] which recurrently processes sequences, meaning given a sentence, it would process the sentence incrementally, word by word.

There are many studies done about the capability of recurrent neural network models to capture context-free languages [12] [7] [8] [22] [18] [13], and linguistic phenomena involving hierarchical structure [14] [9]. Other more general studies about RNN models showed, assuming arbitrary precision, that RNN models are Turing-complete [19], which means that RNNs can complete any algorithmic task formalized by Turing machines. Furthermore, studies have shown how RNNs process hierarchical structures to expose the linguistic properties encoded in the models, using probing and diagnostic tasks to help them visualize the internals of the models [11]. Now, most researches in NLP do not use the RNN architecture but rather the state-of-the-art Transformer architecture [21]. This architecture is not using recurrence to process sequences but instead processes sequences as a whole using self-attention. This enables transformers to do most of the computations in parallel and let them scale up to a larger amount of text and bigger models such as (GPT-2 [16]/GPT-3 [3] and BERT [5]). Transformers are similar to RNNs Turing complete. A study by Perez et al. showed that Transformers could emulate Turing machines' computation when using an unbounded number of autoregressive decoding steps. However, when considering incremental modeling of sequences, then lack of recurrence might suggest the limits of its expressiveness, as this removes the ability to process input sequentially. So it might be more challenging for Transformer models to process hierarchical structure in text.

The Transformer architecture is state of the art, and therefore we focus in this work on this architecture. In particular, how self-attention processes compositions structures in the text.

2 Literature overview

Studying the properties of language models has been emerging since the empirical successes of language models such as LSTMs, RNNs, and Transformers. A particular exciting topic is testing the ability of these models to generalize to hierarchical and compositional structures because hierarchical and compositional structures are essential to model natural language.

A study by Tran et al. [20] tested the abilities of LSTMs, and Transformers to learn hierarchical structures. For this, they used English subject-verb agreement and evaluated logical formulas. Their results suggested that LSTMs are better than Transformers at learning hierarchical structures. Other studies related to this worked instead with artificial languages that can represent nested structures [4]. One set of languages that are commonly used is the Dyck-n languages [15], which consist of well-balanced parentheses with n different types of brackets. An example sentence of a Dyck-2 language is "(()[])[]", which contains two different types of brackets that are balanced, namely "()", and "[]". The applications of such language are, for example, to parse expressions that must have the correctly nested sequence of brackets, such as found in arithmetic and algebraic. In

addition to that, it could also capture long-range and nested dependencies in English subject-verb agreement [23]. Like subject-verb agreement in the sentence, "(laws (the lawmaker) [writes] [and revises]) [pass]", can be captured with an dyck-2 sentence, (()[][])[] [23]. Weiss et al. [22], and Sennhauser et al. [18] show that with the use of Dyck-n languages, that LSTMs are pretty limited on the range of mechanisms they can learn. Nevertheless, their results suggest that LSTMs can recognize the Dyck-1 language using a counting mechanism but showed that LSTMs could not recognize Dyck-2 languages and beyond as it would require emulating a pushdown automaton.

Furthermore, a study by Bhattamishra et al. [2] suggests that Transformers with soft-attention and position masking are also capable of generalizing Dyck-1 languages as LSTMs can, but show that they are incapable of recognizing the language Dyck-n for n $>$ 1. Also, Hahn [10] shows that Transformers are like LSTMs limited in their ability to model the Dyck-2 language. In addition, he proves that hard-attention Transformers cannot model Dyck-n languages and suggests that Transformers using soft-attention cannot achieve perfect cross-entropies when the input is sufficiently long. However, he argues that a Transformer may still be able to model such a language with perfect accuracy when the length of expressions is $n \leq N$; where the number of heads and layers have to increase by N. Therefore when the expressions grow in length, the model has to increase the number of heads and layers.

On the contrary to the work by Hahn [10], the work by Bernardy et al. [1] shows that the Transformer encoder-only model can make good predictions on longer distances and deeper nesting of the Dyck-n language. They achieved this by using a random masking strategy and found out that the Transformer uses a simple parenthesis counting strategy to make good predictions.

Also, Ebrahimi et al. [6] observed that an encoder-only Transformer model can generalize to longer and deeper Dyck-n languages when a starting symbol is added to the expressions. They observed that for Dyck-1, the model achieved almost perfect performance ($> 98\%$) and no degradation when increasing the length of the expressions. The performance on Dyck-n$\geq$ 2 languages shows a nearly constant performance score of ($\sim 93\%$), although the performance dropped significantly on longer expressions.

Furthermore, the most recent study by Yao et al. [23], which was inspired by the work of Hahn et al. [10], and proves that an encoder-only Transformer model can actually recognize a subset of Dyck-n languages, namely Dyck-n,D, which is any Dyck-n language, where the maximum depth is bounded by D. Specifically, they showed that to model a Dyck-n,D language, a self-attention network is needed consisting of $D + 1$ layers and only three heads.

Next to the commonly used Dyck-n languages, Hupkes et al. [11] proposed using an arithmetic language, another artificial language consisting of nested arithmetic expressions. This study shows that RNN's learns to predict nested arithmetic expressions, although the performance drops when increasing the length of the expressions.

Previously summarized works suggest that transformers and recurrent networks have comparable capabilities for learning to recognize context-free languages. However, in practice, we want to model recognition (checking well-formedness) and interpretation of the language. Hupkes et al. analyzed trained recurrent and recursive models on the interpretation task. Characterizing Transformers' application to the same task is the gap that our work aims to fill.

3 Artificial language

This chapter introduces the artificial language used to analyze the Transformer models its ability to compute embedded structures. Additionally, we describe the tasks that the Transformer models have to compute and which strategies they could use to solve them.

3.1 Arithmetic language

The arithmetic language was first proposed by Hupkes et al. [11]. This language consists of embedded arithmetic expressions with a vocabulary that consists of integers in the range of $\{-10, ..., 10\}$, the operators plus(+) and minus(-), and parentheses (and). All these expressions evaluate to an integer between the same range, that is, below or equal 10, and above or equal -10. The expressions are split into subsets by the number of numerals they contain (see Table 1 for a formal description). For example ***L4*** contains expressions with exactly 4 numerals, such as $(2 + (3 - (1 + 2)))$. Other examples can be found in Table 2.

	Sentences	meanings
L_1	$\{-10, -9, ..., 9, 10\}$	$\{-10, -9, ..., 9, 10\}$
L_{m+n}	$\{(l_m \text{ op } l_n) \mid l_m \in L_M, l_n \in L_n, \text{op} \in \{+, -\}\}$	$\langle l_m \rangle$ op $\langle l_m \rangle$

Table 1: Formal description of the arithmetic language.

$\boldsymbol{L_1}$	-4, 1, 5
$\boldsymbol{L_2}$	(-4 + 1), (1 - 5), (5 + 2)
$\boldsymbol{L_3}$	(-4 + (1+1)), (1 - (5 + 2)), ((5-2) + 2)
$\boldsymbol{L_4}$	((-4-2) + (1+1)), (1 - (5 + (2-1)))

Table 2: Examples of arithmetic expressions with different levels of nested expressions.

3.1.1 Strategies

There are many ways a Transformer could compute an arithmetic expression. Therefore we stated two different strategies on how we think a Transformer evaluates nested expressions.

Recursive strategy The first strategy we propose is by recursively evaluating the nested expressions from innermost to outermost, which will compute the evaluation of the whole expression. Each token attends to the subsequent five tokens in the sequence to find an expression that can be evaluated. This flow is visualized in Figure 1, whereby for layer one, the attention is visualized. Each layer will update the first token to the evaluation of the sub-expression. This is seen in Figure 1 after the FNN step.

For a Transformer to evaluate an expression it needs at least L layers, where $\boldsymbol{L_{d-1}} \leq L$. Therefore the example in Figure 1 needs at least four layers to evaluate the $\boldsymbol{L_5}$ expression.

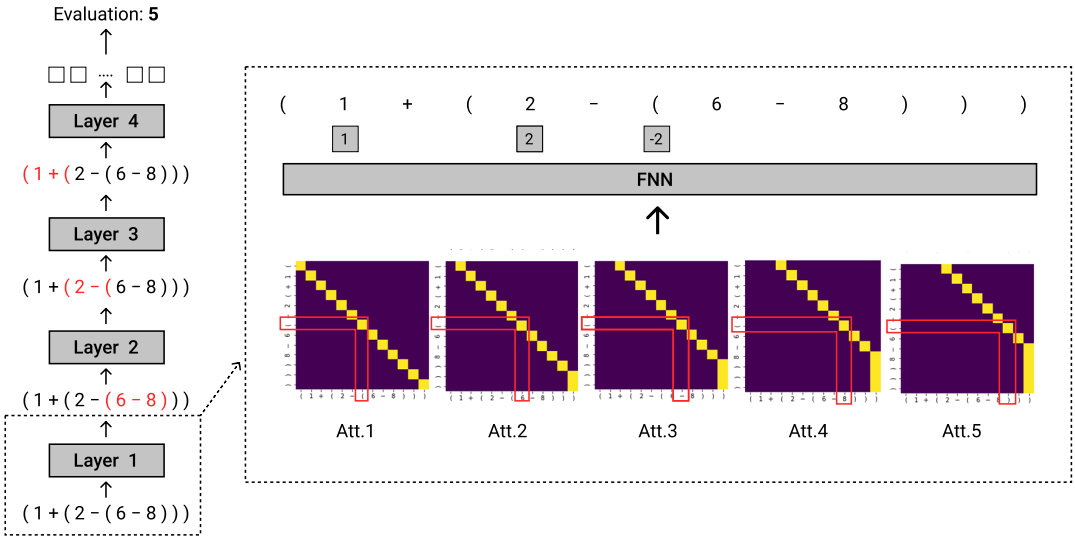


Figure 1: The flow of the **recurrent strategy** wherein the most nested opening parentheses attends with five heads to all the tokens in the most nested expression so that that expression can be evaluated.

Minus counting strategy The second strategy takes advantage of the fact that we can count the number of nested minus operators before each token and transform the tokens accordingly. This is possible because when we subtract a minus it becomes a positive, therefore, an expression like $(2 - (1 - 1))$ is the same as $(2 - 1 + 1)$ or $(2 + (-1 + 1))$. This strategy focuses primarily on right-branching expressions since nested minus operators only occur in right-branching expressions. The idea is that in each layer, one depth level is processed by attending to all its nested minus operators. That is, given an expression like $(1 - (2 - (6 - 8)))$, the first layer will evaluate token 2,

by counting how many minus operators are nested. In this case, there is only one minus operator nested. Thus it becomes -2 The overall flow of this is visualized in Figure 2, whereby layer three is further visualized. For this strategy to work, we need at least as many layers as maximum expression depth.

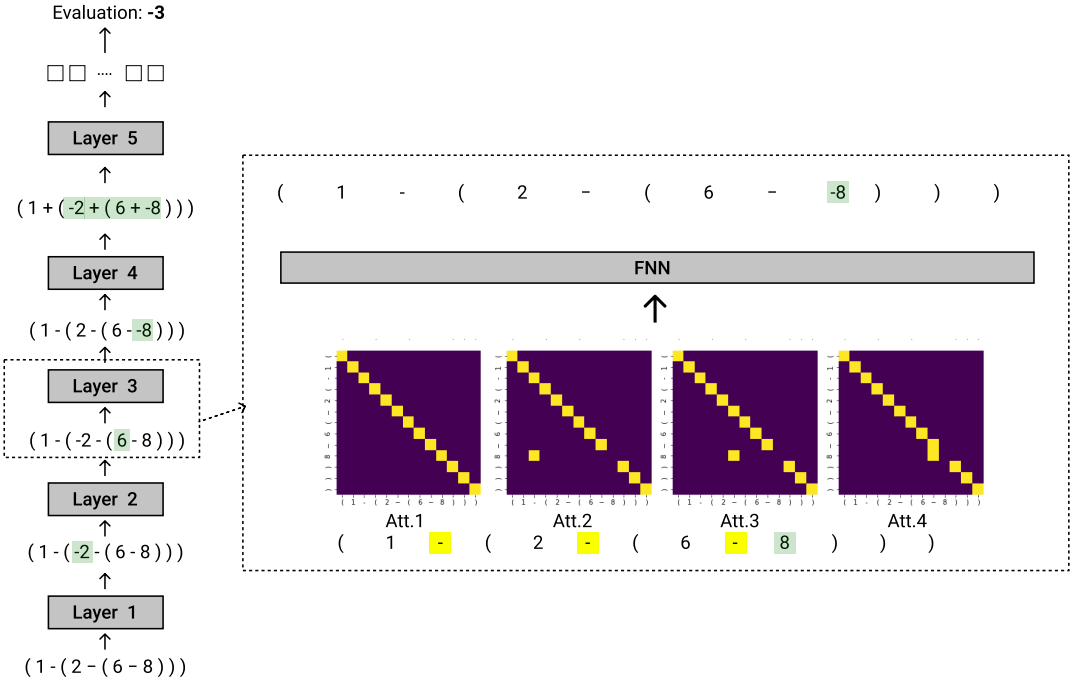


Figure 2: At each layer, the model attends with one head to the next numeral in the sequence and the rest to the nested minuses for that numeral.

3.1.2 Predictions

The two strategies that we proposed differ on how the expressions are evaluated. For example, the **recursive strategy** will not allow evaluations on expression with a higher level of depth than the number of layers in a model since the number of layers bounds it. Furthermore, we predict that when following the **minus counting strategy** the models need more layers and heads only whenever an expression is right-branching and includes minus operators. In this case, the models could take shortcuts whenever the expressions are left-branching or contain plus operators. Therefore we predict that a transformer following the **Minus counting strategy** might compute longer expressions with a low number of heads and layers when evaluating expressions with plus operators or strictly left-branching expressions.

4 Method

We introduced the arithmetic language in the previous chapter, consisting of different expressions with different lengths and depths. Furthermore, we proposed three strategies on how and what is needed in a Transformer to solve these expressions. Therefore, this chapter introduces the specifics of the Transformer architecture that we use to run the experiments and the parameters we used while training the Transformer models. Furthermore, we describe in this chapter how we build the dataset and evaluate the experiments.

4.1 Transformer architecture

Encoder-only model The Transformer model we consider is the encoder-only model from the original seq-to-seq architecture [21]. The encoder consists of multiple layers containing two blocks: a self-attention block and a feed-forward network (FFN). The model takes as input a sequence of symbol representations $s_1, s_2..., s_n \in \Sigma$ and generates a sequence of output vectors $y_1, y_2..., y_n$. First are these input symbol representations converted to a embedding vector using the function $f_e : \Sigma \rightarrow R^{d_{model}}$ and optionally a positional encoding. The position encoding proposed in the original paper uses sine and cosine functions of different frequencies, defined as:

$$PE_{(pos,2i)} = \sin\Big(pos/10000^{2i/d_{model}}\Big)$$

$$PE_{(pos,2i+1)} = \cos\Big(pos/10000^{2i/d_{model}}\Big)$$

where pos is the position and i the dimension. Another more straightforward method of keeping track of word positions is to embed the positions as words. A drawback is that the model needs to see sequences of every length instead of position encodings, which should also process sequences longer than those seen during training. So, for example, to embed the first word, we add a positional embedding that represents position one to the embedding of the first word. The final input vector where the embedding vectors and positional encoding or positional embedding are combined is denoted as a sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2..., \boldsymbol{x}_n)$.

Self-attention Let $\boldsymbol{X}_i := (\boldsymbol{x}_1, ..., \boldsymbol{x}_i)$ for $i \geq 1$. The input vectors in $\mathbf{X}$ will undergo linear transformations $Q(\cdot), K(\cdot)$, and $V(\cdot)$ inside of the self-attention block. These linear transformations will create query, key and value vectors respectively. The self-attention mechanism will take as input a *query* vector $Q(\boldsymbol{x}_i)$, *key* vectors $K(\boldsymbol{X}_i)$, and value vectors $V(\boldsymbol{X}_i)$. For $1 \leq i \leq n$, the output of the self-attention block is a vector:

$$\boldsymbol{a}_i = Att(Q(\boldsymbol{x}_i), K(\boldsymbol{X}_i), V(\boldsymbol{X}_i)) + \boldsymbol{x}_i$$

Where the subscript in $\boldsymbol{X}_i$ indicates the positional masking. The output of the self-attention block denoted by $\mathbf{z}_i$ is computed by $\mathbf{z}_i = O(\boldsymbol{a}_i) + \boldsymbol{a}_i$ where $O(\cdot)$ is usually defined as a FFN with ReLU activation. The operations $+\boldsymbol{x}_i$ and $+\boldsymbol{a}_i$ are the residual connections. To make the complete L-layer, we repeat this block above, which produces a vector $\boldsymbol{z}_i^L$. The final output is obtained by applying a projection layer with normalization or an FNN over the vectors $\boldsymbol{z}_i^L$. The final output is denoted by $\boldsymbol{y}_i = F(\boldsymbol{z}_i^L)$.

4.2 Experimental setup

4.2.1 Transformer implementation

For implementing the transformer, we consider the encoder-only model of the original seq-to-seq architecture [21]. This type of model is usually used for classification tasks. In total, we train 56 different Transformer models with different parameters. That is models with 2 to 8 layers that have 1 to 8 heads. Although the word embedding size is between 160 and 640, we have chosen to multiply the number of heads by 80 for the word embedding. Furthermore, we decided on simple position embedding for the positional encoding schemes because we work with fixed expression lengths. The rest of the parameters we present in Table 3.

4.2.2 Data selection

We arbitrarily sample sets of unique expressions from the arithmetic language introduced in the previous chapter for data selection. We do this in the range of expression lengths from $\boldsymbol{L1}$ to $\boldsymbol{L9}$. For the subsets of lengths $\boldsymbol{L1} - \boldsymbol{L3}$ we are limited by the possible unique expression we can generate. Therefore we only selected for $\boldsymbol{L1}$, $\boldsymbol{L2}$, and $\boldsymbol{L3}$, 21 expressions, 662 expressions, and 15911 expressions, respectively. Furthermore, for the rest of the expression lengths, we select each 16.800 expressions. The result is a dataset of 117.361 expressions. After this selection, we split the dataset into training, testing, and validation dataset, with 60% for training, 20% for the test, and 20% for the validation- set.

Number of layers in the encoder	2-8
Number of heads in the encoder	1-8
Word embedding size	160-640
Number of epochs	90-100
Vocab size	27
Batch size	16
Adam optimizer	$\beta 1 = 0.9$ $\beta 2 = 0.999$ ϵ =1e-08
Max pooling	True
Learning rate	0.0001
Warm-up steps	10.000
Max prediction length	66

Table 3: Parameters we chose for training the Transformer models.

4.2.3 Evaluation

For evaluating the performance during training, we use the **Log-Loss** method, which is one of the most crucial classification metrics based on probabilities. It is based on the likelihood function, which measures the likelihood of the observed outcome.

$$Logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} log(p_{ij})$$

Where N is the number of samples, M the number of classes, and y_{ij} is the predicted result of the classification.

After training the models, we use **accuracy** and **MSE** to analyze the performance on different lengths of expressions. Using the following metrics, we can calculate the accuracy of the models that measure the ratio of correctly predicted expressions. That is True positive (TP), True negative (TN), False positive (FP), and False negative (FN).

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}.$$

5 Results

This chapter will describe the results we got after conducting the experiments. We provide results of the 56 different Transformer models that we trained. First, we start with the overall performance of each model on the validation set. Next, we compare the differences between left- and right-branching expressions. Lastly, we dive deeper into the most challenging expressions, the right-branching expressions with only minus operators.

5.1 Validation set

This first section presents the results using the validation data-set described in Chapter 3. We start with the visualization of the accuracy scores. Each heatmap represents a different number of layers, with on the x-axis the number of heads and the y-axis the expression nesting depth. This is seen in Figure 3.

When looking at these heatmaps, we notice that the performance of the models with only two layers performs noticeably worse than models with three or more layers. Furthermore, we do not notice any significant differences as the layers or numbers of heads increases after two layers.

Left-branching We begin with the left-branching expressions, where we test the performance on expressions with a certain total nested depth and within some left-branching depth that is part of the whole expression. Each of the plots in Figures 4, and 5 represents a heatmap with a number of total nested depth, with on the y-axis the number of left-branching depth that fit the expression, and on the x-axis represents either the number of layers or the number of heads.

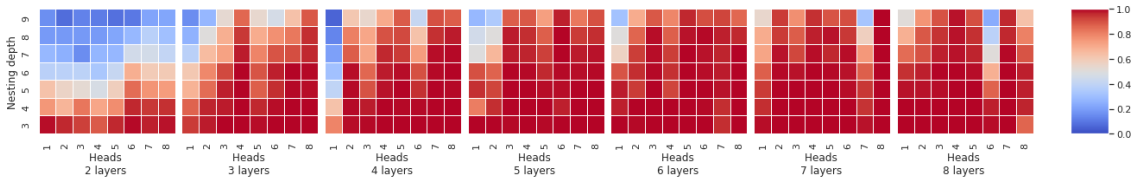


Figure 3: Each heatmap represents the accuracy scores of a given number of layers, with on the y-axis the expression depth and the x-axis the number of heads.

As a result, we notice that the models can easily evaluate expressions with a high left-branching depth, but whenever the expressions are mostly right-branching or mixed-branching, the models show that they perform noticeably worse. In addition, this is the case in both the heatmaps with different layers and different heads. Therefore show that mostly left-branching expressions are easier than right-branching expressions.

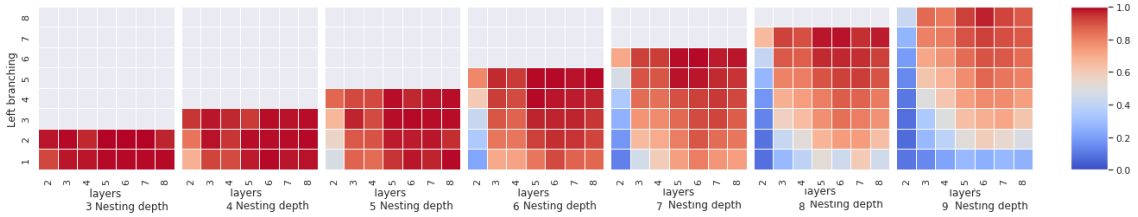


Figure 4: This heatmap shows the average accuracy scores given a number of layer depth, length of expression and embeddedness of that expression.

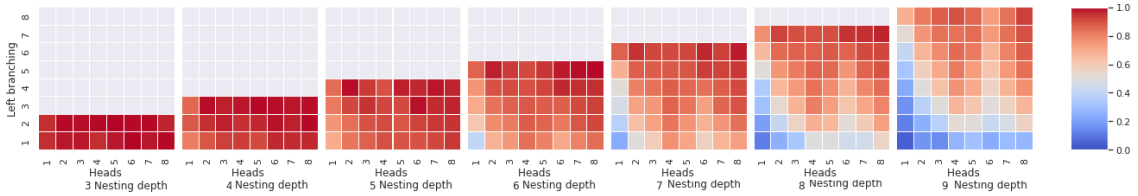


Figure 5: This heatmap shows the average accuracy scores given a number of model heads, length of expression and embeddedness of that expression.

Right-branching Opposite results are seen in the right-branching expressions, whereby the level of right-branching depth in expressions grows, the accuracy drops. This can be seen in the Figures 6, and 7. On the contrary, with the results on the left-branching expressions, we notice that the models have a harder time when the expressions are mostly right-branching.

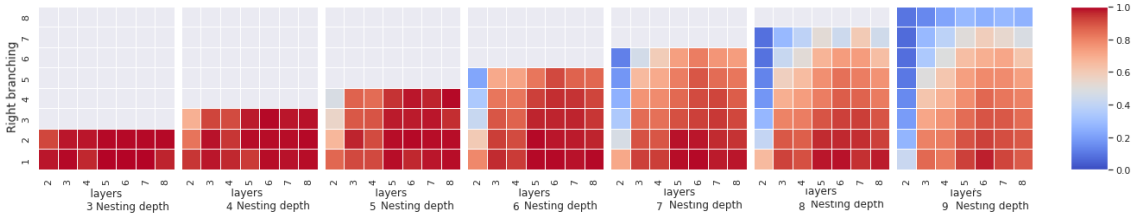


Figure 6: This heatmap shows the average accuracy scores given a number of layer depth, length of expression and embeddedness of that expression.

5.1.1 left- and right-branching expressions

To summarize, we notice left-, and right-branching expressions differ in difficulty, whereas left-branching expressions are much easier to solve than right-branching expressions. Therefore, we

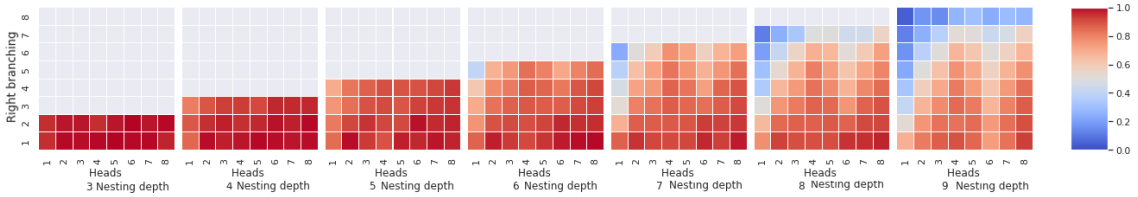


Figure 7: This heatmap shows the average accuracy scores given a number of model heads, length of expression and embeddedness of that expression.

continue on the right-branching expressions since we found that these expressions expose the limits of the models.

5.2 Right-branching

In this section, we go deeper into the performance of the models on right-branching expressions. However, the validation set we used to test the models only contained a small amount of right-branching expressions, especially on longer expressions. Therefore, we made a new data set containing only right-branching expressions to analyze the model performance.

We start with right-branching arithmetic expressions that randomly contain plus and minus operators. The visualization of the results is visible in Figure 8, where each plot shows the accuracy score of the right-branching depth level and with the corresponding number of heads given the number of layers. For example, the first plot shows the performance of a Transformer model with two layers and 1 to 8 heads. In this figure, we notice that the models have a more challenging time generalizing on higher levels of right-branching depth. However, it does not show a specific indication of how the number of heads and layers play a role here. We see that models with minimal two heads and three layers tend to generalize better on higher levels of right-branching depth, but it is not clear how the expression depth level affects the models' performance.

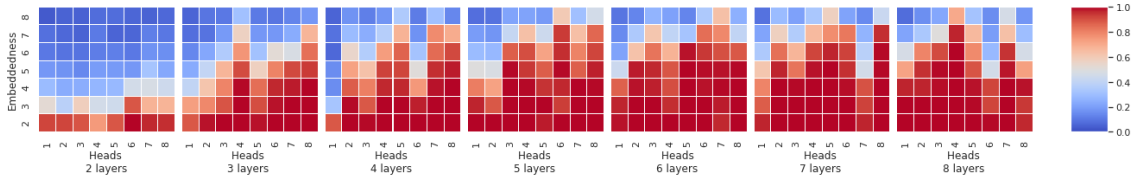


Figure 8: Each heatmap represents the accuracy scores of a given number of layers, with on the y-axis the expression depth and the x-axis the number of heads.

5.3 Right-branching expressions plus / minus operators

In the previous section, we used right-branching expressions with randomly mixed plus and minus operators to test the performance. To further analyze the performance of the different models, we split the right-branching expressions into two separate data sets, one with only plus operators and the second with only minus operators. We do this to see the exact effects on the models using different operators in an expression. Figures 9, and 10, show the accuracy scores of the expression with only plus and minus operators, respectively. When comparing these two figures, we observe a big difference between the performance of expressions with plus operators and minus operators. It is visible that the expressions with only plus operators are much easier than the expressions with only minus operators. The heatmaps of expressions with plus operators show that it only takes a model with three layers to evaluate these expressions. On the contrary, the heatmaps of expressions with minus operators show that increasing the number of layers and heads is crucial to evaluate a higher level of nested depth.

5.4 Right-branching expressions with minus operators

We highlighted in the previous section that the right-branching expressions with only minus operators are much more complex than expressions with plus operators. Therefore, we continue with

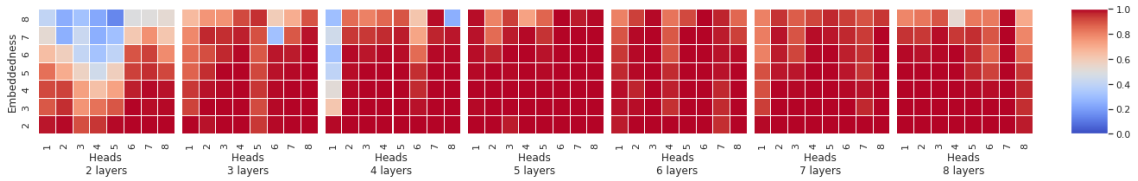


Figure 9: Visualisation of the performance on right branching expressions that only contain plus operators. Each plot shows the number of layer, and the different number of heads.

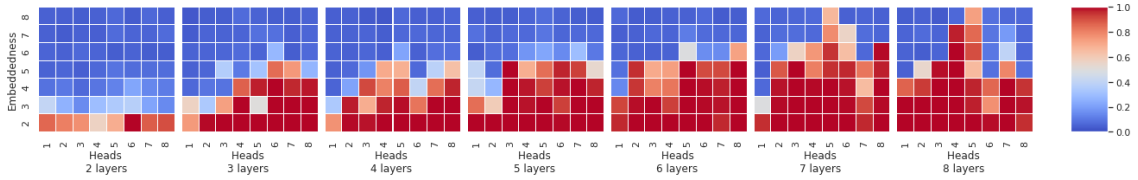
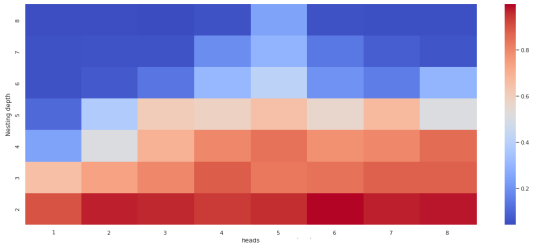


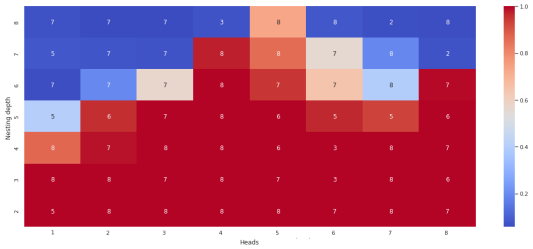
Figure 10: Visualisation of the performance on right branching expressions that only contain minus operators. Each plot shows the number of layer, and the different number of heads.

the right-branching expressions containing only minus operators and then precisely determine the effects of increasing the number of heads and layers. We begin with the performance scores when we increase the number of heads. Then, from these scores, we create two plots, one, the **average** of all the head-layer combinations and, second, the best performing head-layer combination for each number of heads, which are visualized in Figures 11A, and 11A, respectively. From these plots, we notice that until five heads, the performance increases but decreases when the number of heads surpasses five.

Furthermore, we did the exact visualization of the effects when we increased the layers. Likewise, as in the previous Figures, we take each layer's average and best performing head combinations. This is visualised in Figure 12A, and 12B. From these plots, we find that increasing the number of layers allows the models to generalize to a higher level of expression depth.



(a) Results of evaluation on right branching expressions with only minus operators, and averaged on the number of heads.

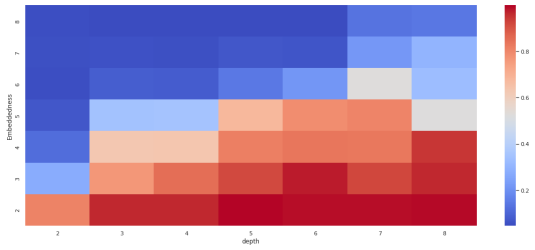


(b) Results of the best performing models based on the number of heads with the number of layers inside the heatmap.

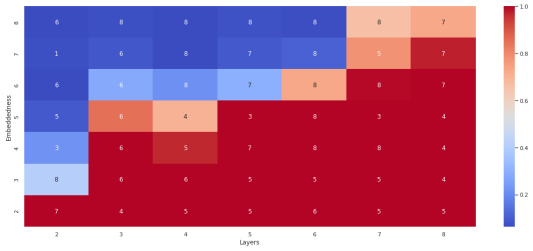
Figure 11: Results of averages and best performing models side by side.

5.5 Results summarise

To summarise, the performance of self-attention models attending to arithmetic expressions differ in performance between left- and right-branching expressions. Left branching expressions are relatively easier than the right-branching expressions, especially when we consider expressions with only minus operators, and this difference is again visualized in Figure 13. Furthermore, we see that models with at least three layers can evaluate deeper nested left-branching expressions. However, in the case of right-branching expressions, we need to increase the number of layers to evaluate more complex nested expressions. Also, the number of heads plays a role here, although after the number of heads surpasses five, we see a performance drop. Therefore, five heads might be the best parameter for these tasks.

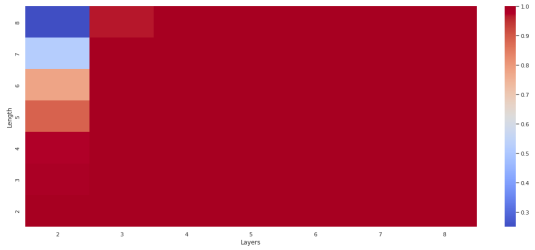


(a) Results of evaluation on right branching expressions with only minus operators, and averaged on the amount of layers.

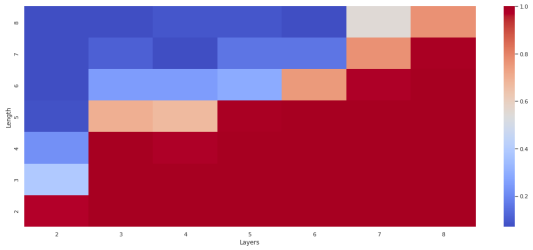


(b) Results of the best performing models based on the amount of layers with the number of heads inside of the heatmap.

Figure 12: Results of averages and best performing models on right-branching expressions side by side.



(a) Results of left branching expressions.



(b) Results of right branching expressions.

Figure 13: It shows two plots of accuracy scores from left-branching and right-branching expressions.

6 Discussion

In the previous section, we present the results of the performance of the models. In addition to those results, we present an analysis of the connection to the theoretical part. Furthermore, we present an analysis of the attention head activation of one of these models. We do this because previous work suggests that analyzing the attention scores could give some insight into the internal dynamics of self-attention models. So then, our goal is to find some patterns in the attention head activation that correlate with the strategies we described earlier.

6.1 Trained models

Almost all 56 trained models seemed to perform well on the validation set. However, the models with less than three layers or only one head performed noticeably worse. At the same time, the rest of the models could generalize well on nine numerals' expression lengths and eight's depth levels. The high performance was primarily because many of the expressions in the validation set consisted of either mostly left-branching expressions or right-branching expressions with plus operators, which are less complex. It looks like that the transformer models learn to take shortcuts to solve the whole expression, like we proposed in (**Minus counting strategy**), where we said that expressions consisting of plus operators and left-branching expressions would be the easiest for the model to solve due to the lack of semantic depth in these expressions and therefore complexity. In addition to that, the **Recurrent strategy** would not match the outcome of the performance of the models. That is because models with fewer layers could still solve expressions with higher nested depth and might be taking shortcuts to solve the expressions, which the **Recurrent strategy** does not allow. Furthermore, expressions that the model could not take shortcuts on are the expressions that are strictly right-branching with minus operators, and the results show that the models need to have equal or more layers than the expression depth was, thus when evaluating on deeper expression depth, the model will require a higher number of layers to do so.

6.2 Analysis attention head activation

We start by considering the model with three heads and five layers since the small number of heads and layers will allow us to get a clear idea about what patterns in the head activation present.

Attention heat-maps left-branching The left-branching expressions were the easiest for the model to process. We observed from the results that the models could process longer expressions without increasing the number of layers or heads. In addition, when visualizing the attention-heat-maps, we notice a clear pattern of numbers attending to its appropriate operator sign to allow the number to transform according to which operator it is attending. Furthermore, the models do not have difficulty evaluating long expressions because the transformed numbers can essentially be summed up for left-branching expressions. For example, Figure 14 shows the heatmap of layer one, head two, of left-branching expressions consisting of minus operators. Here we see four different heatmaps with different expression lengths showing the same pattern, wherein they show that the numerals attend to the appropriate operator symbol. Likewise, left-branching expressions containing plus or mixed operators show the same pattern as seen in Figure 15. According to this pattern, the transformer model can efficiently process longer expressions without increasing the number of layers or heads.

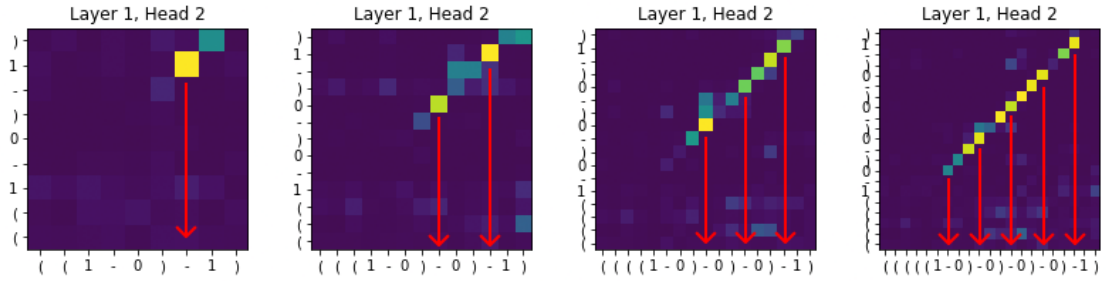


Figure 14: Heatmaps of four different right-branching expression lengths with only minus operators, the numerals attend to the designated operator signs.

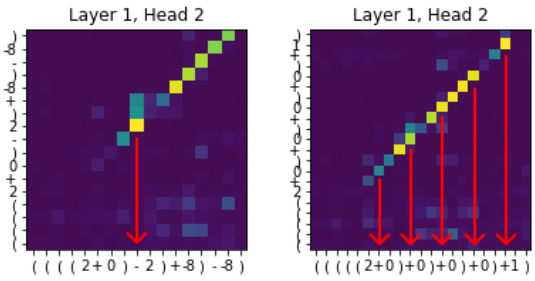
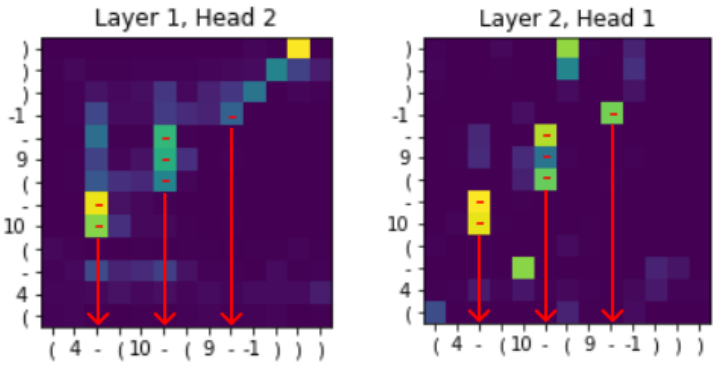


Figure 15: Heatmaps of 2 different right-branching expression lengths, wherein the numerals attend to the designated operator signs, one with mixed operators, and the other consists of only plus operators.

Attention heat-maps right-branching The right-branching expressions caused the greatest difficulties for the models, especially those containing only minus operators. Therefore we discuss only the right-branching expressions that contain minus operators. We observed an interesting pattern in 41 of the 56 models when analyzing the attention heatmaps on these expressions. It shows how the expression should be solved to get the correct answer, whereby each part of the expression gets transformed by the minus operator in front of it. Figure 16 represents two heatmaps of two different models that visualize this pattern. We see that the first nested numeral and operator pair (10 and -) attends to the first minus operator. Furthermore, in the most nested expression, we see that only the first numeral and minus symbol attends to the previous minus operator, and the second numeral attends to the minus operator in front of it. The same pattern occurs in more complex expressions, as seen in Figure 17, although these heatmaps are messier than on less complex expressions, they still show the same pattern. In addition, we noticed that

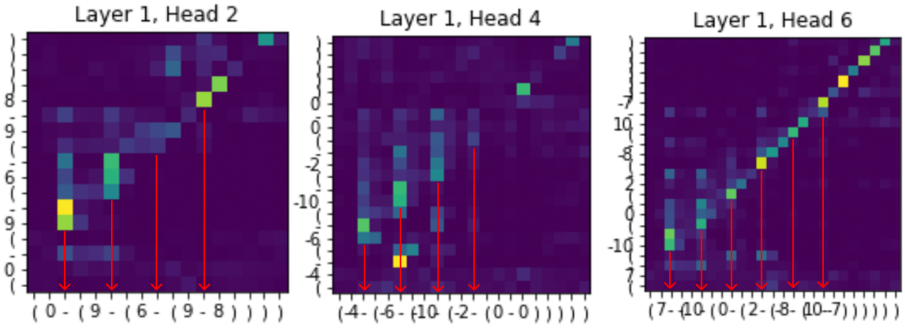
this pattern only occurs in either the first or second layer of the models. Looking at this pattern, we know that transforming those sub-parts with a minus symbol at once will not directly solve the expression but instead needs to be done sequentially. The first nested numeral and minus operator must be transformed by the first minus operator, transforming into -10 and a plus operator, respectively. After the transformation, the most nested expression transforms according to the new plus operator, which results in 9 and -. Lastly, the remaining -1 attends to the minus operator that stayed the same, creating 1. This pattern might indicate that the model knows what steps to take to solve the expression in the upper layers and uses the remaining layers to process that evaluation. Although, the attention heatmaps in the latter layers do not show a pattern that describes how this execution would process and, therefore, stays unknown.



(a) Attention heatmap of a model with three heads and seven layers.

(b) Attention heatmap of a model with three heads and eight layers.

Figure 16: There are two different attention score heatmaps of the evaluation on right-branching expressions of nested depth of four. It shows that the numbers and operators in sub-expressions attend to the preceding operator symbol.



(a) Attention heatmap of a model with six heads and five layers.

(b) Attention heatmap a model with two heads and six layers.

(c) Attention heatmap of a model with seven heads and seven layers.

Figure 17: There are three different attention score heatmaps of the evaluation on right-branching expressions of nested depth of four, five, and six, respectively. It shows that the numbers and operators in sub-expressions attend to the preceding operator symbol.

Summarize To summarize, we have seen that the left-branching expressions are easy to solve and explainable by the pattern in the attention heatmaps. Furthermore, show that they do not need more layers or heads to generalize to more prolonged expressions. Finally, as observed, the numbers attend to the according plus-or-minus operator to see if the operator is minus or plus. Furthermore, on right-branching expressions with only minus operators, we saw that the models need more layers to generalize to longer expression lengths. Although it is not exactly known how the models evaluate these expressions, but we found patterns that might indicate that the nested expressions attend to the operator in front of it but cannot transform each directly and thereby needs to do that sequentially, which shows the need for more layers in order to do transform each

nested expression accordingly.

7 Conclusion

In this work, we studied whether self-attention networks can learn compositionality using an arithmetic language, a simple language that allowed us to construct precise compositional structures and test the networks on these.

To understand how self-attention networks can learn the arithmetic language, we hypothesized two strategies beforehand that we believed a transformer could follow to evaluate the meaning of an arithmetic expression:

- **The recursive strategy** would allow a transformer to evaluate what is inside the innermost set of parentheses, replace that with the result, and repeat this until there is nothing to evaluate. We hypothesized that a transformer needs as many layers as nested expression depth.
- **The minus counting strategy** would allow a transformer to evaluate expressions by counting how many minuses are nested before each numeral, transforming the numeral according to an even or uneven count of minus operators, and summing everything together. Nevertheless, the Transformer needs at least as many layers as maximum expression depth for this strategy to work.

Furthermore, we demonstrated that self-attention networks with a low number of heads and layers could easily evaluate arithmetic expressions of any tested lengths. However, we highlighted that the high performance was due to simple expressions and showed that the performance drops when the arithmetic expressions become more complex, that is, right-branching expressions with only minus operators. To allow a transformer to generalize on more complex expressions, we showed that it needs to increase the number of layers as the expression depth grows.

When analyzing the attention heatmaps, we found patterns that explain the Transformer's strategies that it follows to evaluate arithmetic expressions. For example, numbers in left-branching expressions attend to the preceding operator. Therefore, when transforming these numbers by the preceding operator, it could sum the numbers together to get the result. Summing the numbers is possible due to the lack of complexity in the left-branching expressions and correlates with the need for a relatively low number of layers to evaluate these expressions. Therefore, it shows that the Transformer takes shortcuts to achieve its goal.

Simultaneously, when evaluating right-branching expressions, we found a pattern where the nested sub-expressions attend to the preceding operator. As a result, the Transformer cannot sum the sub-expressions directly together but instead needs to transform each sub-expression sequentially, which explains the need for more layers when the nesting of expressions grows. Therefore, it shows that Transformers simultaneously learn to take shortcuts and a more complete strategy.

When we compare the hypothesized strategies and the findings, we see that the Transformers take shortcuts whenever there is a lack of minus operators or the expressions are mostly left-branching. However, it is not solving these expressions recursively as we proposed in **The recursive strategy** but rather in one step. However, we see similarities in the learned strategy and **the recursive strategy** when evaluating right-branching expressions, but instead of evaluating the sub-expressions, it shows that it is transforming the numbers and operators according to the preceding operator. However, it is unclear whether the trained models follow the recursive strategy in the deeper layers, especially since they did not show any pattern to validate this.

The analysis allowed us to explore the limitations of a self-attention network. In particular, we find that the network takes shortcuts on less complex nested expressions but requires deeper networks on more complex nested expressions.

References

[1] Jean-Philippe Bernardy, Adam Ek, and Vladislav Maraev. Can the transformer learn nested recursion with symbol masking? In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 753–760, 2021.

[2] Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability of self-attention networks to recognize counter languages. *arXiv preprint arXiv:2009.11264*, 2020.

[3] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[4] Noam Chomsky and Marcel P Schützenberger. The algebraic theory of context-free languages. In *Studies in Logic and the Foundations of Mathematics*, volume 26, pages 118–161. Elsevier, 1959.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[6] Javid Ebrahimi, Dhruv Gelda, and Wei Zhang. How can self-attention networks recognize dyck-n languages? *arXiv preprint arXiv:2010.04303*, 2020.

[7] Felix A Gers and E Schmidhuber. Lstm recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

[8] André Grüning. Stack-like and queue-like dynamics in recurrent neural networks. *Connection Science*, 18(1):23–42, 2006.

[9] Kristina Gulordava, Piotr Bojanowski, Edouard Grave, Tal Linzen, and Marco Baroni. Colorless green recurrent networks dream hierarchically. *arXiv preprint arXiv:1803.11138*, 2018.

[10] Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171, 2020.

[11] Dieuwke Hupkes, Sara Veldhoen, and Willem Zuidema. Visualisation and'diagnostic classifiers' reveal how recurrent and recursive neural networks process hierarchical structure. *Journal of Artificial Intelligence Research*, 61:907–926, 2018.

[12] Yvonne Kalinke and Helko Lehmann. Computation in recurrent neural networks: From counters to iterated function systems. In Grigoris Antoniou and John Slaney, editors, *Advanced Topics in Artificial Intelligence*, pages 179–190, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[13] Samuel A Korsky and Robert C Berwick. On the computational power of rnns. *arXiv preprint arXiv:1906.06349*, 2019.

[14] Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. Assessing the ability of lstms to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4:521–535, 2016.

[15] George A Miller and Noam Chomsky. Finitary models of language users. 1963.

[16] Alec Radford, Jeffrey Wu, Dario Amodei, Daniela Amodei, Jack Clark, Miles Brundage, and Ilya Sutskever. Better language models and their implications. *OpenAI Blog https://openai.com/blog/better-language-models*, 2019.

[17] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[18] Luzi Sennhauser and Robert C Berwick. Evaluating the ability of lstms to learn context-free grammars. *arXiv preprint arXiv:1811.02611*, 2018.

[19] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. *Journal of computer and system sciences*, 50(1):132–150, 1995.

[20] Ke Tran, Arianna Bisazza, and Christof Monz. The importance of being recurrent for modeling hierarchical structure. *arXiv preprint arXiv:1803.03585*, 2018.

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

[22] Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision rnns for language recognition. *arXiv preprint arXiv:1805.04908*, 2018.

[23] Shunyu Yao, Binghui Peng, Christos Papadimitriou, and Karthik Narasimhan. Self-attention networks can process bounded hierarchical languages. *arXiv preprint arXiv:2105.11115*, 2021.

A Appendix

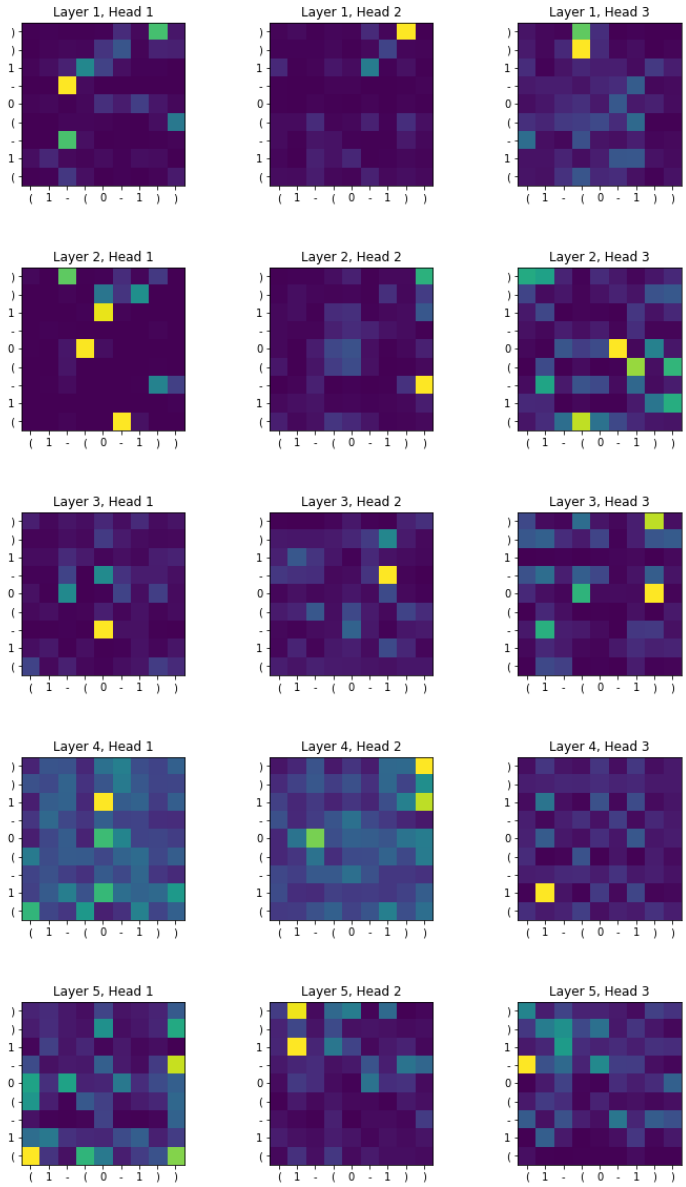


Figure 18: Full visualisation of the attention scores.

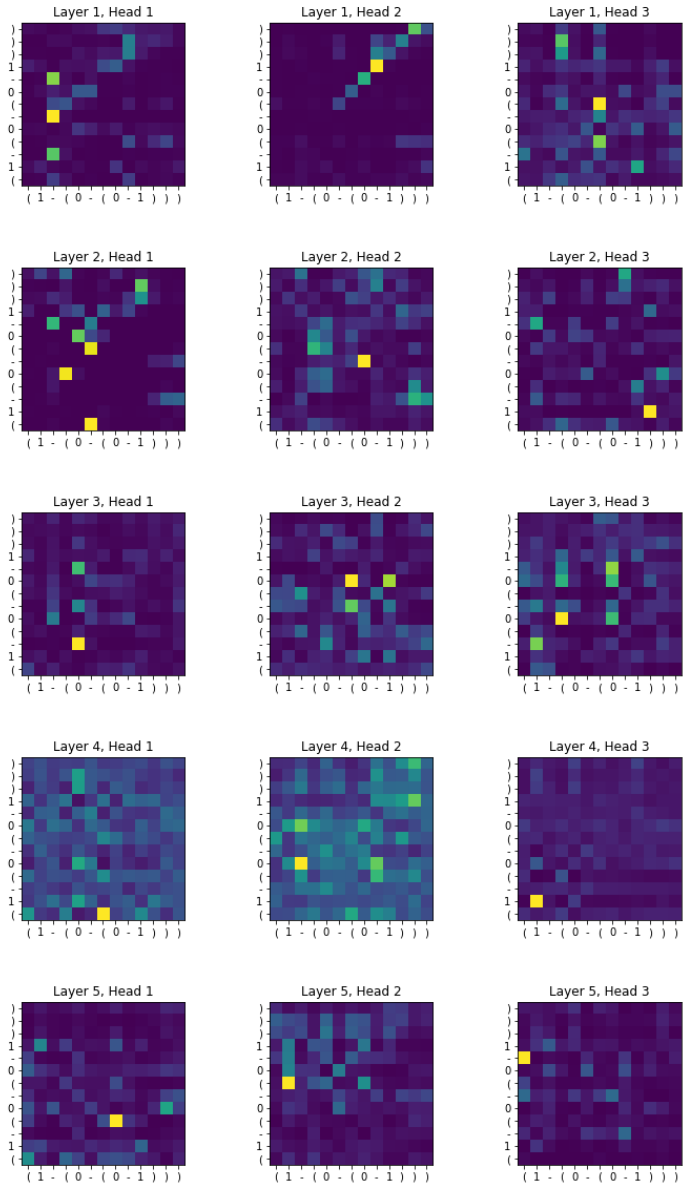


Figure 19: Full visualisation of the attention scores.

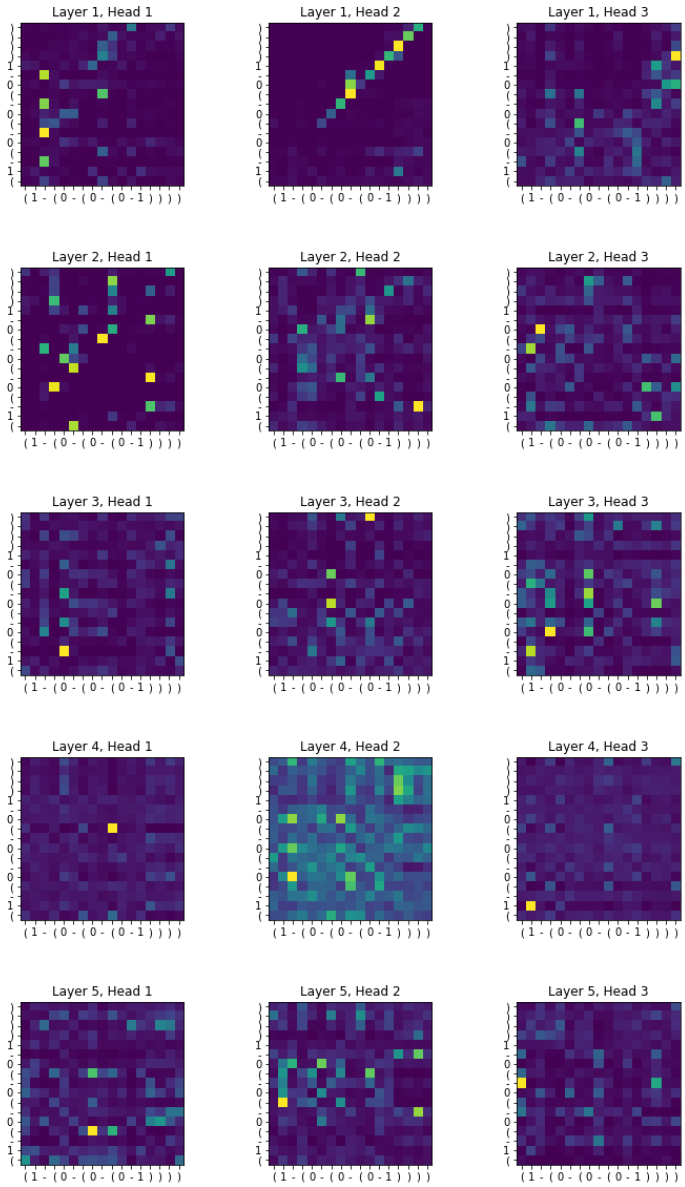


Figure 20: Full visualisation of the attention scores.

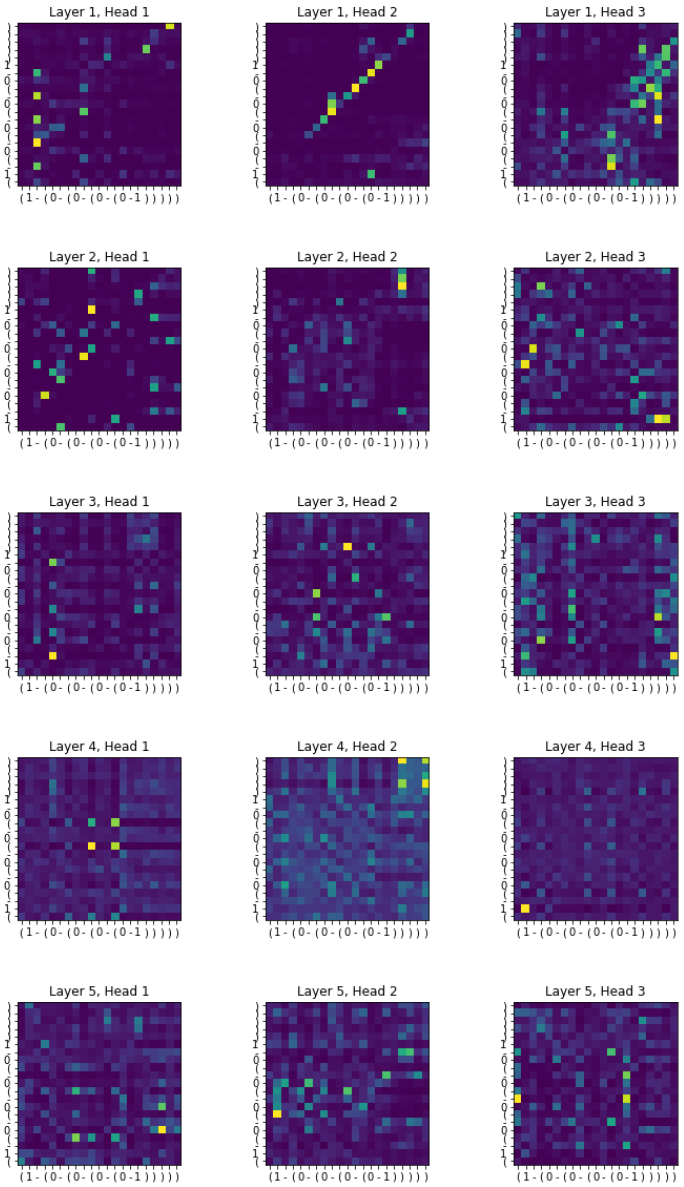


Figure 21: Full visualisation of the attention scores.