Eötvös Loránd University

Ádám Fraknói

Solving Mathematical Problems with Transformers

Master's Thesis

ELTE Faculty of Science
Mathematics MSc

Supervisors:

Zsolt Zombori
Alfréd Rényi Institute of Mathematics, and
Eötvös Loránd University

Researcher

András Kornai
Dept. of Algebra, Budapest University of
Technology and Economics

Professor

Budapest, 2024

Contents

1 Introduction **4**

2 Mathematical Framework **6**

2.1 Embedding and Positional Encoding 7
2.2 Layer Normalization and Softmax Function 8
2.3 Multi-Head Attention 9
2.4 Feed-Forward Network 11
2.5 Output Layer 11
2.6 Training 12
2.7 Initializations 13
2.8 Further Techniques 13

3 Complexity Class of Transformers **14**

3.1 Circuit Complexity 14
3.2 Upper and Lower Bounds for Transformers 15
3.3 Lower Bound for CoT Transformers 16

4 Modular Addition **19**

5 Noisy Majority Problem **22**

5.1 Problem Definition 23
5.2 Technical Details 23
5.3 Minimal Theoretical Transformer 24
5.4 Understanding Heads 27
5.5 Reducing the Number of Heads 30

6 Conclusion **35**

Acknowledgment

I would like to thank my supervisor, Zsolt Zombori, for his help. He guided me, managed the research project, and thoroughly reviewed and supported me during the work on this thesis. I would also like to thank András Kornai for his valuable guidance throughout the research process. I would also like to thank Pál Zsámboki, who developed the model architecture that I used to train the models and answered thoroughly my technical questions.

1 Introduction

A major turning point in the history of natural language processing (NLP) was the emergence of the first semantically meaningful word embeddings into vector space [31, 6, 5, 24]. It turned out that the same representation can be extremely useful for several downstream tasks, such as translation or sentiment classification, that the system was not trained on explicitly. The idea behind these embeddings is really simple: a good way to characterize the meaning of a word is through the meanings of other words that appear close to the given word in natural text. Arguably, finding the proper representation of words and sentences is at the core of the success surrounding present-day language models.

Numerous systems apply Deep Learning to aid mathematical reasoning, and inside the trained models, they also implicitly create vector embeddings for mathematical objects. There is, however, virtually no evidence that these embeddings are faithful to the semantics of the considered mathematical theory.

It is a conjecture that the lack of proper embeddings is a major bottleneck of learning assisted theorem proving systems and overcoming this problem is one of the next major challenges. Here, I made steps in this direction, to clarify issues of mechanistic interpretability [25] and contribute to AI safety [18, 34].

To get an idea of the differences between natural and formal languages, consider the natural and formal sentences "The weather is wonderful today" and "$3 * (6 - 2) = 24/2$". Each word in the English sentence has a rather small set of meanings and the context determines which one is applicable. The meaning of the sentence can be surprisingly well approximated by the set of the relevant meanings of the words: changing the word order only results in subtle changes. [17] (Section 1.3) provides a back-of-the-envelope estimation of the relative information content of various linguistic components in natural language, concluding that around $80 - 84\%$ comes from words, $12 - 16\%$ comes from the logical/grammatical structure and emotive content accounts for around $5 - 7\%$. For the arithmetic formula, however, many of the words are much more ambiguous, for example, the digit 2 can mean two, twenty, two hundred etc. Variables are an extreme example of terms whose meaning is entirely context-dependent. On the other hand, formal languages have unambiguously defined compositional semantics, i.e, we know exactly how the meaning of a complex expression is built up from those of the subexpressions. In summary, the bulk of the meaning in natural language comes from words, while in formal language it comes from logical structure. It is hence an interesting – and yet

open – question whether embedding methods for natural text will also work for embedding formulas. Terms of a mathematical theory often have a very clear structure and one can check how well that structure manifests in the embedding space. [40]

Transformers have gained a lot of attention because of their success in natural language processing [35], therefore in this thesis, I will focus on Transformers' capabilities in understanding formal languages.

First, I build a mathematical framework for analyzing Transformers in Section 2. [9] Computational complexity of the models is also interesting from a theoretical point of view, how it treats problems as formal languages. Therefore, in Section 3, I analyze the capabilities of Transformers across different architectures. Changing the definition of the architecture can have a significant impact on the model's complexity class. But it turns out that the most realistic architectures of Transformers are in the complexity class of TC^0, which is the class of problems that can be solved by polynomial-size Boolean circuits of constant depth using gates of AND, OR, NOT and majority. However, if we allow the model to have the chain-of-thoughts (CoT) mechanism, i.e. the model can take intermediate steps, then it can recognize any regular language. In Section 4, I cover the topic of modular addition. This a is first step to understand what mechanisms from NLP work well directly and what needs to be adapted to the particularities of the formal content. I have done some research to understand the embedding of three-digit sums modulo 1000. So I summarize the conclusions of this research and present the results from the literature in this section. It seems to be a simple task, but it turns out that Transformers sometimes learn it in a seemingly complicated way. In Section 5, I introduce the noisy majority problem, which is an even simpler task. It should determine whether zeros or ones form a majority in a sequence while ignoring noise symbols represented by twos. This problem is simple enough to be rigorously analyzed and reverse-engineered.

Larger models produce better results, but the priority is not to have a state-of-the-art model on a task. Instead, the goal is to analyze small models, understand how they work, and find best practices to improve them. Once we have a good understanding of small models and small problems, we can scale them up and move on to larger models and more complex problems.

Finally, in Section 6, I discuss implications and future research directions.

During this thesis I have done numerous experiments and research. In Section 4, I analyzed the embedding of modular addition, while in Section 5, I investigated a problem where the model has to determine whether zeros or ones form a majority in a sequence. The most significant experimental contribution is presented in Section

5.5, where I introduced a method that enables smaller models to achieve perfect test accuracy performance with comparable consistency to larger models. This is particularly noteworthy since small models typically struggle to achieve consistent results, even though they should theoretically be able to learn the task perfectly.

2 Mathematical Framework

Before I define Transformers, it is helpful to review the high-level structure of it.

Transformers have many variations for different tasks. A typical transformer-based model consists of two main components: the encoder and the decoder. The encoder processes the input data to extract meaningful representations, while the decoder generates the output based on these representations. Transformer architectures may vary based on specific use cases, such as encoder-only transformers (e.g., BERT [8]), decoder-only transformers (e.g., GPT [29]), or encoder-decoder transformers (e.g., T5 [30]). In this thesis, I focus solely on decoder-only transformers. In natural language processing (NLP), a token is a unit of text that models process. It is usually a word, but it can also be a sub-word or a character. In computer vision, a token can be a pixel, in audio processing it can be a sound, etc. In my case, a token will always be a character (e.g. a mathematical symbol or a number).

The input of the decoder-only transformer is a sequence of tokens, and the output is a probability distribution over the next token in the sequence. The model is trained to predict the next token in the sequence given the previous tokens. The training process involves minimizing the cross-entropy loss between the predicted and true next tokens.

Positional encoding is a technique that adds a vector to the input embedding to represent the positions of tokens in the sequence. Without positional encoding, the model cannot account for the order of tokens in the sequence. Sometimes positional embeddings are also learned, but in most cases, they are fixed.

The Transformer architecture consists of multiple layers of self-attention blocks, each containing a multi-head self-attention mechanism and a feed-forward network. They continuously refine the embedding of tokens to adapt to the context. The output of each self-attention mechanism and feed-forward network is added back to the input embedding, forming a residual connection. This embedding, which is modified and reused throughout the layers, is called the *residual stream*.

At the beginning of multi-head self-attention mechanism, the embedding is split into heads, and each head is processed separately. Then, the outputs of the attention

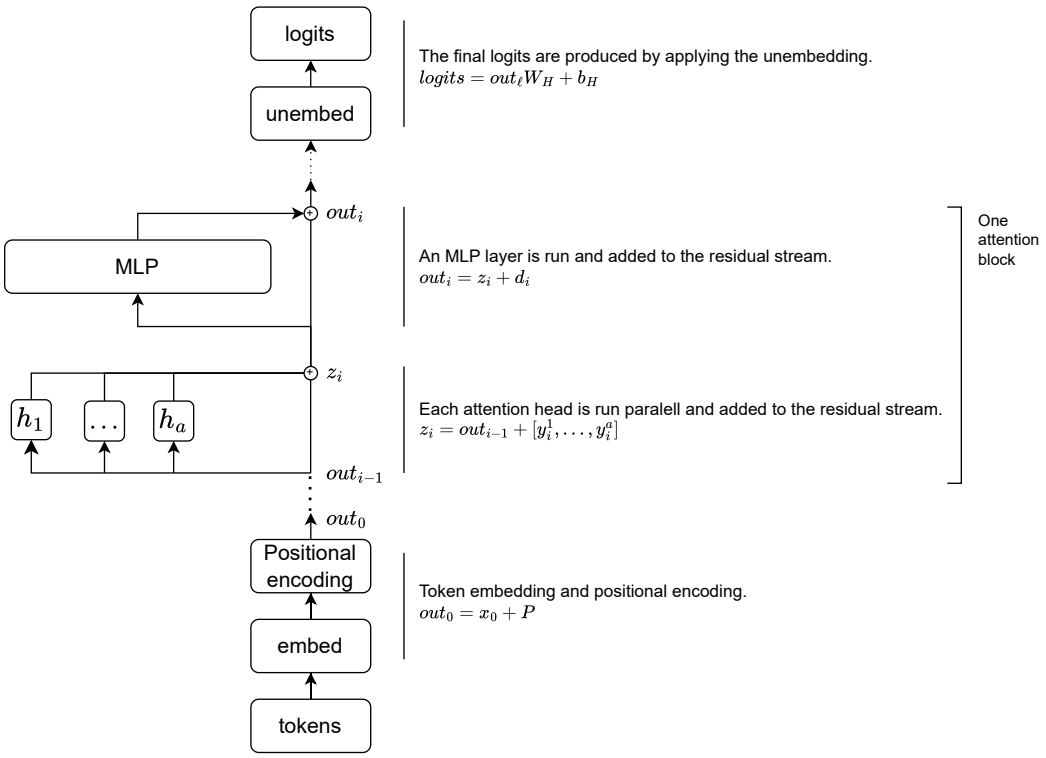

Figure 1: Decoder-only Transformer

heads are concatenated, linearly transformed to the original embedding dimension, and added to the original embedding.

This is followed by a *feed-forward network* (FFN), also called *multi-layer perceptron* (MLP), with an activation function (e.g. ReLU). The output of the MLP is added again to the residual steam.

Multi-head attention and the MLP are repeated ℓ times. The output of the last layer is extracted with a linear transformation to get the logits. Using the softmax function we can get the probabilities of the next token from logits.

The decoder-only transformer is illustrated in Figure 1.

2.1 Embedding and Positional Encoding

Let us define the architecture more formally. Let d be the embedding dimension, ℓ be the number of attention blocks (layers), a be the number of attention heads, d_h be one attention head's dimension. Let V be the vocabulary and v be its size. These are hyperparameters of the model that are set manually before training.

Let $W_E \in \mathbb{R}^{v \times d}$ be the token embedding matrix, where each row represents

the embedding of a token from vocabulary V. This embedding matrix is typically learned during the training.

Let $t = (t_1, \ldots, t_n)$ be the input sequence, where $t_i \in V$ is the i-th token and n is the length of the sequence. The token embedding is computed using W_E and t. The result is $\mathbf{x}_0 = (x_0^1, \ldots, x_0^n)$, where $x_0^i \in \mathbb{R}^d$ is the embedding of token t_i, i.e., the row in W_E corresponding to token t_i. Then, the positional encoding $P \in \mathbb{R}^{n \times d}$ is defined as follows:

$$P_{i,2j} = \sin\left(\frac{i}{N^{2j/d}}\right)$$
$$P_{i,2j+1} = \cos\left(\frac{i}{N^{2j/d}}\right)$$

Here, i denotes the i-th token, $0 \leq j \leq \frac{d}{2} - 1$, and N is significantly larger than the biggest j, e.g. $N = 10,000$. The positional encoding is added to the token embedding, i.e. $\mathbf{out}_0 = \mathbf{x}_0 + P$.

The input to the i-th attention block is the output of the previous block: $\mathbf{out}_{i-1}$.

2.2 Layer Normalization and Softmax Function

Before I define the self-attention block, I introduce some basic concepts.

A layer normalization [1] has two learnable parameters $\gamma, \beta \in \mathbb{R}^d$, and is defined as follows:

$$\mu = \frac{1}{d}\sum_{j=1}^{d} w_j, \quad \sigma^2 = \frac{1}{d}\sum_{j=1}^{d}(w_j - \mu)^2$$
$$\text{LayerNorm}(\mathbf{w}) = \frac{\mathbf{w} - \mu}{\sigma^2 + \epsilon} \odot \gamma + \beta,$$

where $\odot$ represents element-vise multiplication and ϵ is a small constant to prevent division by zero.

Layer normalization is applied before the self-attention block and before the feed-forward network. It scales the input to have zero mean and unit variance, and then scales it by γ and shifts it by β, thus the output has β mean and γ variance.

The softmax function is defined the following way:

$$\text{softmax}(\mathbf{w}) := \frac{e^{\mathbf{w}}}{\sum_{i=1}^{n} e^{w_i}}$$

The purpose of the softmax function is to normalize the logits to get probabilities (i.e. it sums the values for all tokens to 1).

2.3 Multi-Head Attention

Let $\mathbf{out}_{i-1} \in \mathbb{R}^{n \times d}$ be the output of the previous layer. The self-attention block begins with with a layer normalization, followed by the multi-head self-attention mechanism, then comes the feed-forward network with another layer normalization.

For the self-attention mechanism we have the following parameters for each head h: Let $W^h_{Q,i}, W^h_{K,i}, W^h_{V,i} \in \mathbb{R}^{d \times d_h}$, be the query, key, value matrices, $M \in \{0,1\}^{n \times n}$ be the masking matrix and $W^h_{O,i} \in \mathbb{R}^{d_h \times d}$ be the output matrix. The output of an attention is calculated as follows:

$$
\begin{aligned}
\mathbf{x}_{i-1} &= \text{LayerNorm}(\mathbf{out}_{i-1}) \\
q^h_i &= \mathbf{x}_{i-1} W^h_{Q,i} \in \mathbb{R}^{n \times d_h} \\
k^h_i &= \mathbf{x}_{i-1} W^h_{K,i} \in \mathbb{R}^{n \times d_h} \\
v^h_i &= \mathbf{x}_{i-1} W^h_{V,i} \in \mathbb{R}^{n \times d_h} \\
A &= \text{softmax}\left(\frac{q^h_i (k^h_i)^T + M}{\sqrt{d_h}}\right) \in \mathbb{R}^{n \times n} \\
r^h_i &= A v^h_i \in \mathbb{R}^{n \times d_h} \\
y^h_i &= r^h_i W^h_{O,i} \in \mathbb{R}^{n \times d_h} \\
\mathbf{z}_i &= [y^1_i, \ldots, y^a_i] + \mathbf{out}_{i-1} \in \mathbb{R}^{n \times d}
\end{aligned}
$$

where $[y^1_i, \ldots, y^a_i]$ denotes the concatenation of the heads' outputs. The $\frac{1}{\sqrt{d_h}}$ term is a scaling factor, which is used to prevent the dot product from becoming too large.

The intuition behind the attention is the following. Let us consider only one layer. For each head and for each token we create three vectors: query, key, and value. Their dimensions are d_h and they are calculated from their embedding with a linear transformation. From query and key vectors we get the attention matrix A. The A_{ij} value represents how much the i-th token should attend to the j-th token. This one is calculated by the dot product of the query and key vectors. The query vector represents what that token is looking for, while the key vector represents the context of that token. If the dot product of the query and key vectors are high, it means key vector's token is contextually important for understanding the query vector's token.

M denotes the masking matrix, which prevents tokens from attending to future tokens. Typically we have

$$M = \begin{pmatrix} 0 & -\infty & -\infty & \dots & -\infty \\ 0 & 0 & -\infty & \dots & -\infty \\ 0 & 0 & 0 & \ddots & -\infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{pmatrix}$$

where $-\infty$ means that future tokens will not influence the current token, because we are applying the softmax function. (Sometimes there are also $-\infty$ values on the diagonal.)

To obtain the attention weights, we apply the softmax function to the attention matrix, so that each row sums to 1. There are also other types of attention functions, e.g. hard attention, where we only take the maximum value from the attention matrix. In case of tie, we can take the average of the values, or we can take the leftmost value, etc.

After that, we compute the sum of the value vectors, weighted with the attention weights. This is multiplied by an output matrix (just to get the right dimension if $ad_h \neq d$), and then adding this information to the output of the previous layer (i.e. residual stream) we get the output of the head.

Note that the softmax function is the only part that is not linear. We can think about attention heads that splits into query-key and output-value circuits, where the former one describes how much a given query token "wants" to attend to a given key token, and the latter describes how a given token will affect the output logits if attended to. [9]

This whole method is called scaled dot-product attention. Each head is calculated independently and in parallel, therefore the process is also called multi-head attention. Head dimension is commonly defined as $d_h := d/a$, therefore the output matrix is only essential if $ad_h \neq d$. Note also, that query and key vectors can have other dimension than d_h (i.e. value vectors' dimension), since queries and keys are only multiplied with each other. The only requirement is that they should have the same dimension.

The complexity of the self-attention block is $O(n^2 \cdot d)$, where n is the sequence length, d is the embedding dimension, since calculating qk^T terms in the attention matrix requires $O(n^2 \cdot d)$ steps. This is a huge limitation if we want to process long sequences. Therefore, there are many techniques that try to optimize it, since in large language models the large context window is crucial. [7, 2, 38, 16, 3, 4]

2.4 Feed-Forward Network

The self-attention block is followed by a layer normalization and feed-forward network (FFN). If the FFN has one hidden layer with hidden dimension of m, then it consists of two linear transformations with a ReLU activation function, and finally there is a residual connection again. So we have learnable parameters of $W_{i,1}, W_{i,2}^T \in \mathbb{R}^{d\times m}$, $b_{i,1} \in \mathbb{R}^m, b_{i,2} \in \mathbb{R}^d$, and the calculation is the following:

$$\begin{aligned}\mathbf{c}_i &= \text{LayerNorm}(\mathbf{z}_i)\\ \mathbf{d}_i &= \max(0, \mathbf{c}_i W_{i,1} + b_{i,1})W_{i,2} + b_{i,2}\\ \mathbf{out}_i &= \mathbf{z}_i + \mathbf{d}_i\end{aligned}$$

Feed-forward layers constitute two-thirds of a transformer model's parameters. [10] showed that feed-forward layers operate as key-value memories: each key corresponds to input patterns while values predict next tokens' distribution. Lower layers learn shallow patterns while upper layers learn semantic ones. These outputs are added to the residual stream thus it is a composition of memories, which is subsequently refined throughout the layers.

2.5 Output Layer

The final output of the decoder-only transformer is followed by a linear transformation to get the logits for all tokens, and then a softmax function to get the probabilities for all token. I call this part of Transformer as *output layer*. Formally, let $W_H \in \mathbb{R}^{d\times v}, b_H \in \mathbb{R}^v$ be learnable parameters, then the final output is:

$$\begin{aligned}\mathbf{logits} &= \mathbf{out}_\ell W_H + b_H \in \mathbb{R}^{n\times v}\\ \mathbf{p} &= \text{softmax}(\mathbf{logits}) \in \mathbb{R}^{n\times v}\end{aligned}$$

In summary, $p_{i,j}$ represents probability of the j-th token at the $(i+1)$-th position given the input sequence $t = (t_1, \ldots, t_i)$. During generation, we focus on the vector $p_{n,.}$, which predicts the next token. We usually determine the next token with highest probability. Conversely, during training, we typically use the entire matrix $\mathbf{p}$, because we want to train to predict the whole sequence.

2.6 Training

Each deep learning model requires a loss function to optimize. Usually, it measures the difference between the predicted and the true values. In my case, the model was trained by minimizing the commonly used cross-entropy loss.

Originally, cross-entropy comes from information theory. The formal definition is the following:

Definition 2.1. *Assume we have have two probability distributions over the same set X: an estimated p and a true distribution q. Then the cross-entropy between p and q is*

$$H(p, q) = -\mathbb{E}_p[\log q] = -\sum_{x \in X} q(x) \log p(x)$$

where $\mathbb{E}_p$ denotes the expected value with respect to the distribution p. The second equality only holds if X is discrete.

However, the distribution p is unknown in our case. So we have to approximate it by taking a sample of sequences. Let $\mathbf{y} \in \{0, 1\}^{n \times v}$ be the one-hot encoded target sequence, i.e. $y_{i,j}$ represents whether the i-th token in the sequence is the j-th token in the vocabulary. Then we approximate the cross-entropy loss function the following way:

$$H(\mathbf{p}, \mathbf{y}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{v} y_{i,j} \log p_{i,j}$$

The model is trained by minimizing the loss function using the AdamW optimizer [21], which is a variant of the Adam optimizer [15], a stochastic optimization method using weight decay to prevent overfitting. The AdamW optimizer does the following: let t denote the time step, $\boldsymbol{\theta}_t$ the model parameters and $\nabla f_t(\boldsymbol{\theta}_{t-1})$ the gradient at step t. Let $\beta_1 = 0.9$, $\beta_2 = 0.999$ be the gradient decay rates, λ the weight decay, $\eta_t = 10^{-4}$ the learning rate, and $\epsilon = 10^{-8}$ a small constant. $\mathbf{m}_t, \mathbf{v}_t$ are the first and second moment estimates. Then the update rule is the following:

$$\begin{aligned}
\mathbf{g}_t &:= \nabla f_t(\boldsymbol{\theta}_{t-1}) + \lambda \boldsymbol{\theta}_{t-1} \\
\mathbf{m}_t &:= \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\mathbf{g}_t & \mathbf{v}_t &:= \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2 \\
\hat{\mathbf{m}}_t &:= \mathbf{m}_t / (1 - \beta_1^t) & \hat{\mathbf{v}}_t &:= \mathbf{v}_t / (1 - \beta_2^t) \\
\boldsymbol{\theta}_t &:= \boldsymbol{\theta}_{t-1} - \eta_t \left(\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} + \lambda \boldsymbol{\theta}_{t-1} \right)
\end{aligned}$$

First, the AdamW optimizer calculates the gradient of the loss function and adds the weight decay term. Then it calculates the first and second moments of the gradient and uses decay rates, i.e. takes into account the previous moments. After that, it applies bias correction to these moments, that helps correct the bias toward zero during the early time steps. Finally it updates the parameters with the bias corrected moments, learning rate, and weight decay term.

2.7 Initializations

Initialization of learnable parameters is also an important part of the training that can highly affect it. The parameters are initialized with random values in the following way:

$$
\begin{aligned}
W_E &\sim \mathcal{N}\left(0, \frac{1}{d}\right) \\
W^h_{Q,i}, W^h_{K,i}, W^h_{V,i}, W^h_{O,i}, W_{\ell,2} &\sim \mathcal{N}\left(0, \frac{2}{3}\sqrt{\ell} \cdot \frac{1}{d}\right) \quad \text{for } 1 \leq i \leq \ell \\
W_{i,1}, W_{i,2}, W_{\ell,1} &\sim \mathcal{N}\left(0, \frac{2}{3}\sqrt{\ell} \cdot \frac{2}{d}\right) \quad \text{for } 1 \leq i \leq \ell - 1
\end{aligned}
$$

The reason behind the factor $\frac{2}{d}$ is that it precedes a ReLU activation function. This initialization is called He Initialization or Kaiming Initialization and is designed for ReLU activations. [13] The factor $\frac{2}{3}\sqrt{\ell}$ is another common initialization from [14].

Another similar and commonly used method is the Xavier initialization, which is designed for sigmoid and tanh activations. [11] The difference between the He Initialization and the Xavier Initialization is that the standard deviation of the former one is equal to $\sqrt{\frac{2}{d_{in}}}$, while the latter is equal to $\sqrt{\frac{2}{d_{in}+d_{out}}}$, where d_{in} and d_{out} are the input and output dimensions of the layer.

2.8 Further Techniques

Dropout [32] is a regularization technique that randomly removes some of the neurons from the network during training. In each iteration during the training phase, each neuron is removed with a predefined probability p. This helps to prevent the model from overfitting. Formally, let $\mathbf{m} \in \{0,1\}^d$ be a random variable, where

$m_i = 1$ with probability p. Then during the traning phase we have

$$\text{Dropout}(\mathbf{w}) = \mathbf{w} \odot \mathbf{m},$$

where $\odot$ is the element-wise multiplication. During the validation phase we have

$$\text{Dropout}(\mathbf{w}) = (1 - p) \cdot \mathbf{w}.$$

This ensures that the expected value of the output is the same during training and during validation.

Dropout is used after calculating the embedding (after the positional encoding is applied), after multi-head attention and after the feed-forward network.

In practice, instead of forwarding one sequence through the model, we are forwarding a collection of input sequences, called *minibatches*. This changes the formalization so that every matrix has an additional 3rd dimension. For example, the input token embedding is $\mathbf{X}_0 \in \mathbb{R}^{b \times n \times d}$, where b is the minibatch size, and n is the maximal sequence length. If a sequence is shorter than n, we pad it with a special token, thus every sequence will have the same length.

3 Complexity Class of Transformers

Computational complexity theory is a highly studied and wide-ranged topic in mathematics and theoretical computer science. The question arises where Transformer models belong in this theory. In this section, I overview some of the results about transformers' computational complexity from [33], which is a detailed and recent survey on this topic. After that I show the main result from [20], that says a constant-depth transformer with T steps of chain-of-thoughts using constant-bit precision and $O(\log n)$ embedding size can solve any problem that is solvable by Boolean circuits of size T.

3.1 Circuit Complexity

Definition 3.1. *A* Boolean circuit *is a directed acyclic graph where each node is an AND, OR, or NOT gate. There are also input gates without fan-in (without incoming edges), and output gates without outgoing edges. AND and OR gates have two or more fan-in, while NOT gate has exactly one fan-in. The output of the circuit is the value of the output gate.*

Each Boolean circuit calculates a function $f : \{0,1\}^n \to \{0,1\}^m$, where n is the number of input gates and m is the number of output gates. The *size* of the circuit is the number of gates, and the *depth* of the circuit is the length of the longest path from an input gate to an output gate.

Let $\text{SIZE}[T(n)]$ denote the class of problems that can be solved by Boolean circuits of size $O(T(n))$. Now we can define $\text{P/poly} := \bigcup_{k\in\mathbb{N}^+} \text{SIZE}[n^k]$, i.e. as the class of problems that can be solved by polynomial-size Boolean circuits. Since any Turing Machine with at most $T(n)$ step can be simulated by a Boolean circuit of size $O(T(n))$, we have $\text{P} \subseteq \text{P/poly}$.

Two classical problems are the following:

$$\text{MAJORITY} = \{(x_1, \ldots, x_n) \in \{0,1\}^n : \sum_{i=1}^{n} x_i > \frac{n}{2}\}$$

$$\text{PARITY} = \{(x_1, \ldots, x_n) \in \{0,1\}^n : \sum_{i=1}^{n} x_i \text{ is odd}\}$$

Definition 3.2. *The complexity class* AC^i *is the class of problems that can be solved by polynomial-size Boolean circuits with* $O(\log^i n)$ *depth. The class* NC^i *is the same, but gates of AND and OR have fan-in exactly 2 (i.e. they are not allowed to have more than 2 inputs). The class* TC^i *is the same as* AC^i*, but we also have a MAJORITY gate that can have arbitrary fan-in. Let* $AC := \bigcup_{i\in\mathbb{N}^+} \text{AC}^i$*, and similarly for* NC *and* TC*.*

Specifically, AC^0 is the class of problems that can be solved by polynomial-size Boolean circuits of constant depth. We know that $\text{NC}^i \subseteq \text{AC}^i \subseteq \text{TC}^i \subseteq \text{NC}^{i+1}$ for all $i \in \mathbb{N}$. We also know that $\text{PARITY}, \text{MAJORITY} \notin \text{AC}^0$.

3.2 Upper and Lower Bounds for Transformers

There are many variants of Transformers, and therefore the complexity of the model can vary. Some aspects that can vary are the following:

- Architecture: encoder-only, decoder-only or encoder-decoder
- Attention pattern: leftmost-hard, rightmost-hard, avarage-hard, or softmax
- Position encoding
- Attention masking: none or (strict) future masking
- Layernorm: Inclusion or omission

- Precision: infinite, $O(\log n)$, $O(1)$, floating point
- Number of intermediate steps (chain-of-thought): zero, $O(\log n)$ or $O(n)$
- Uniformity: whether parameter values or number of parameters depend on n.

Paper [26] assumes infinite precision, and proves that Transformers are Turing Complete. However, this is not very realistic, since we have finite precision in practice.

Paper [23] shows that decoder-only Transformers with $O(\log n)$ precision and softmax attention are in log-uniform TC^0, that means we can construct a Turing Machine with logarithmic space that can create the circuits. However, [12] showed that leftmost hard attention is in AC^0, i.e., when we only attend to the leftmost token.

[20] uses floating number in other way than others. In their model, they are rounding the numbers to a floating number after every operation. They showed that even polynomial-depth constant-precision Transformers with $\log n$ steps of chain-of-thoughts are within AC^0. If they use logarithmic-preicision, but without exponents of the floating numbers, then their model is in TC^0.

The literature sometimes considers uniform complexity, and sometimes non-uniform complexity. The latter one allows different programs for each different input length, while in the former one, we need to have a Turing Machine that takes n as input and outputs the appropriate program for the given input length. The question arises whether having different transformer for each input is a realistic assumption. However, note that in practice transformers cannot scale up to arbitrary length of inputs. Furthermore, in many cases the model size should be also scaled up for longer input sequences.

3.3 Lower Bound for CoT Transformers

Chain-of-thought (CoT) is a method that allows the model to "think" by making intermediate reasoning steps. E.g. in a natural language problem, we force the model to explain its reasoning, and not just give the final answer. It turned out that it significantly improves the model's ability to solve complex reasoning problems. [36]

Now, let us formulate it. Let $TF(x_1, \ldots, x_n)$ denote the output of the transformer model. Then, let $x_{n+1} = TF^1(x_1, \ldots, x_n) := TF(x_1 \ldots, x_n)$. After that recursively define $x_{n+i} = TF^i(x_1, \ldots, x_n) := TF(x_1, \ldots, x_{n+i-1})$. So after $T(n)$ steps of chain-

of-thoughts the model output is $(x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+T(n)})$. We can define for each problem individually how we calculate the final output from these intermediate steps. (Note that the the TF transformer also had $n + T(n)$ input tokens, but the last $T(n)$ tokens were just padding tokens, so the model will ignore them.)

Definition 3.3. *Let* $\mathrm{CoT}[T(n), d(n)]$ *denote the class of problems that can be solved by a constant depth transformer with* $T(n)$ *steps of chain-of-thoughts using* $O(d(n))$ *embedding size and constant-bit precision.*

I define constant-bit precision here that the model can only use an s constant number of bits to represent any number, and after every (interim or final) calculation (addition, multiplication, inner product, softmax etc.) it will round to the closest number that we can represent. In case of tie, we can break it by picking the one with a smaller absolute value. Let us denote all operations with constant-bit precision with an $[\cdot]_s$ subscript. So e.g. $\mathrm{sum}_s(\mathbf{x}) := \left[\left((x_0 + x_1)_s + x_2\right)_s + \ldots x_n\right]_s$. We can define all operations in Transformers in the same way, but because of its clarity I omit it here.

Theorem 3.4. *For any polynomial function* $T : \mathbb{N} \to \mathbb{N}$*, we have* $\mathrm{SIZE}[T(n)] \subseteq \mathrm{CoT}[T(n), O(\log n)]$*. In particular,* $\mathrm{P/poly} = \mathrm{CoT}[T(n), O(\log n)]$*.*

The following proof is from [20].

Proof. Given a Boolean circuit with size of $O(T(n))$. We can assume that it only contains AND and NOT gates. Without loss of generality we can further assume that its size is $T(n)$. Take a topological order on the gates so that the input gates are numbered from 1 to n. We will develop a transformer model in such a way, that in each step of CoT, it will simulate the i-th gate of the Boolean circuit.

In a high-level, we will do the following: Let the Transformer contain two layers with one-head attention mechanism and feedforward network with two hidden layers. Store the information about Boolean circuits in the positional encoding. First, the model obtains all the information of the current gate (gate id, gate type, first input of the gate, second input of the gate) from the predefined position encoding by adding it to the embedding. Then, in the first attention phase the model queries the the first input. In the second attention phase the model queries the second input. In the last feedforward network, the model calculates the output based on the gate type and the inputs. In the next step, the model will get the previous output as well.

For each gate $i \in [n+1, n+T(n)]$ let $a(i)$ and $b(i)$ denote the first and second input gate's id. (NOT gate has only one input, so we can define $b(i) = 0$ for NOT gates.) Let $c(i) = 0$ if i-th gate is NOT and $c(i) = 1$ if it is AND. For gates $i \in [1, n]$ we define $a(i) = b(i) = c(i) = 0$.

Let $k := \lceil \log_2(T(n) + n) \rceil$ which is $O(\log n)$ since $T(n)$ is polynomial. Let $d(n) := 3k + 6$ be the token embedding dimension (thus $d(n) = O(\log n)$)) and the token embeddings $W_E(0) = \mathbf{0}$ and $W_E(1) = e_1$. For each $2 \leq i \leq n + T(n)$ let positional encoding the following:

$$(\theta_{PE}^{i-1})^T = [0, 0, 0, 0, c(i), i - 1_{(k2)}, a(i)_{(k2)}, b(i)_{(k2)}, 1]$$

where $a_{(k2)}$ and $b_{(k2)}$ denotes a special binary encoding defined later that maps number i to a k-dimensional signed binary vector.

Before defining the query, key and value vectors I will show what values we want to achieve. Since we have only one head, I will omit the head index. Let x_i denote the embedding of the i-th token. We want to achieve the following:

$$\begin{aligned}
out_0[i] &= [x_i, \quad 0,0, \quad 0,c(i), \quad (i-1_{(k2)})^T,(a(i)_{(k2)})^T, \quad (b(i)_{(k2)})^T,1]^T \\
y_0[i] &= [x_i, \quad x_{a(i)},0, \quad 0,c(i), \quad (i-1_{(k2)})^T,(a(i)_{(k2)})^T, \quad (b(i)_{(k2)})^T,1]^T \\
out_1[i] &= [x_i, \quad x_{a(i)},0, \quad 0,c(i), \quad (i-1_{(k2)})^T,(a(i)_{(k2)})^T, \quad (b(i)_{(k2)})^T,1]^T \\
y_1[i] &= [x_i, \quad x_{a(i)},x_{b(i)}, \quad 0,c(i), \quad (i-1_{(k2)})^T,(a(i)_{(k2)})^T, \quad (b(i)_{(k2)})^T,1]^T \\
out_2[i] &= [x_i, \quad x_{a(i)},x_{b(i)}, \quad g_i(x),c(i), \quad (i-1_{(k2)})^T,(a(i)_{(k2)})^T, \quad (b(i)_{(k2)})^T,1]^T
\end{aligned}$$

where

$$\begin{aligned}
g_i(x) := ReLU(1 - x_{a(i)} - c(i)) + ReLU(x_{a(i)} + x_{b(i)} + c(i) - 2) = \\
= \begin{cases} 1 - x_{a(i)1} & \text{if } c(i) = 0 \text{ (NOT gate)} \\ x_{a(i)} \cdot x_{b(i)} & \text{if } c(i) = 1 \text{ (AND gate)} \end{cases}
\end{aligned}$$

which is the desired output and it can indeed be calculated by the feedforward network. The output of this iteration (and the input of the next iteration) will be the fourth coordinate, i.e. $x_{i+1} := TF(x_1, \ldots, x_n, \ldots, x_{i-1})[4] = g_i(x)$.

Now we only need to define the attention mechanism. It will copy the value of two fan-in of the gate to the second and third coordinate.

We will use the fact that $[\exp(-B_s)]_s = 0$, where B_s represents the largest representable number with s bits of precision. (The exact representation of numbers is not important as long as this property holds.)

Let $a_{k2} := 2\,\mathrm{bin}_k(i) - (1, \ldots, 1)$, where $\mathrm{bin}_k(i)$ is the binary representation of i with k bits. So $a_{k2} \in \{-1, 1\}^k$. Now let us begin with the first attention: $q_i[2j-1] := (a(i)_{k2})[j]$, $q_i[2j] := 1$, $k_i[2j-1] := (i-1)_{k2}[j] \cdot B_s$, $k_i[2j] := -B_s$, $v_i := e_2$ for $1 \leq j \leq k$. We can construct these vectors, because the last coordinate of out_0 is always 1, so we can use it for the even coordinates.

Then we have the following: $[\exp(\langle q_i, k_{i'} \rangle_s)]_s = \mathbb{1}_{i'=a(i)}$. For this it is enough to to prove that $\langle q_i, k_{i'} \rangle_s = (1 - \mathbb{1}_{i'=a(i)})B_s$. We have to think about how we calculate the inner product with s bits of precision. It will start with $h_1 = (q_i[1] \cdot k_{i'}[1])_s$, and then $h_j = (h_{j-1} + (q_i[j] \cdot k_{i'}[j])_s)_s$. So if $i' = a(i)$, then $h_{2j-1} = B_s$ and $h_{2j} = 0$ for all j, thus $h_{2k} = \langle q_i, k_{i'} \rangle_s = 0$. If $i' \neq a(i)$, then there is a smallest j' when $h_{2j'-1} = -B_s$. After that $h_{j''} \leq 0$ for all $2j' - 1 \geq j'' \leq 2k$, and so $h_{2k} = -Bs$, which we needed.

So we proved that the first attention mechanism works as we wanted. The second attention mechanism is the same, but with $b(i)$. This finishes the proof. □

Corollary 3.5. *Every regular language belongs to* $\mathrm{CoT}[T(n), O(\log n)]$.

Proof. Regular languages can be recognized by finite automata, and it is known that finite automata can be simulated by NC^1 circuits with linear size. [19] Thus, by the previous theorem, the statement holds. □

4 Modular Addition

In this section, I summarize two articles [25, 27] which are considering modular addition in transformers. The first article considers the problem of adding two numbers modulo 113 with a one-layer transformer, while the second one considers adding two five-digit numbers with a one-layer transformer. Both articles reverse engineer the trained model to understand how the model works, furthermore, the first one also investigates the learning process.

In [25] the authors used a one-layer transformer with $d = 128$, $a = 4$ and hidden dimension of $m = 512$. The input to the model was the form of "a b =", where a and b are the numbers to be added with one-hot encoding. This means that each number between 1 and 113 was represented with a unique token. The output was the sum of the numbers modulo 113. They focused on the embedding matrix W_E, and the so-called neuron-logit map $W_L := W_{1,2}W_H$, i.e. the product of the output linear map of the feed-forward network and the output layer matrix. After reverse-engineering the model they found that the algorithm represents the number in a

circle and convert addition to rotation about a circle. More formally the algorithm is the following:

1. Embed input numbers a and b to sine and cosine functions with various frequencies $w_k = \frac{2k\pi}{P}, k \in \mathbb{N}$, i.e. to $\sin(w_k a)$, $\cos(w_k a)$, $\sin(w_k b)$ and $\cos(w_k b)$.

2. Compute the sine and cosine of the sum $a + b$ using trigonometric identities:

$$\cos(w_k(a+b)) = \cos(w_k a)\cos(w_k b) - \sin(w_k a)\sin(w_k b)$$
$$\sin(w_k(a+b)) = \sin(w_k a)\cos(w_k b) + \cos(w_k a)\sin(w_k b)$$

   This is computed in the attention and MLP layers.

3. For each output logit c, compute $\cos(w_k(a+b-c))$ using the trigonometric identity:

$$\cos(w_k(a+b-c)) = \cos(w_k(a+b))\cos(w_k c) + \sin(w_k(a+b))\sin(w_k c)$$

   This is a linear function of $\cos(w_k(a+b))$ and $\sin(w_k(a+b))$, and implemented in W_L.

4. The unembedding matrix adds together $\cos(w_k(a+b-c))$ for various k values. This causes constructive interference at $c^* = a + b \mod p$ (giving it a large logit), and destructive interference everywhere else (giving small logits).

They were also investigating the learning process. They found that it can be split into three phases: memorization of the training data, circuit formation, where the model generalizes the mechanism, and cleanup, when weight decay removes the memorization components. First, the model very quickly memorizes the training set, but could not solve the problem on the validation set. It seemed that nothing would happen after that, but after a long time it quickly learned the validation set as well. This phenomenon is called *grokking*, i.e. the model suddenly understands the problem. Two key condition was required to the grokking: the small training set, and the the weight decay which tried to minimize the norm of the model weights.

The paper [27] also investigated a one-layer transformer but trained on five-digit integer addition. They used decimal encoding, i.e. each digit in base 10 was represented by a unique token. Their key findings were the following:

The model uses three base functions that operate on individual digit pairs and two compound function that chain operations across digits:

- Base Add (BA): calculates sum of digits modulo 10 ignoring carry

- Make Carry 1 (MC1): evaluates if adding digits results in carry of 1 to the next column
- Make Sum 9 (MS9): evaluates if sum of digits is exactly 9
- Use Carry 1 (UC1): adds previous column's carry to the current sum
- Use Sum 9 (US9): propagates carry across multiple digits

However, these tasks occur in the training data with different frequencies. E.g. BA is much more common than US9. During training, tasks are learnt for each digit independently, progressively increasing per digit accuracy.

The model has three attention heads that learn different aspects. E.g. to predict the third digit from right, the first head calculates MC1 on first digit, the second head calculates MC1 and MS9 on second digit, and the third head calculates BA on third digit. The feed-forward network combines outputs from the three heads using trigrams to compute the final digit. Simple carry propagation works well but complex carry chains (like $99999 + 00001 = 100000$) where the MC1 must propagate across many digits, remain challenging. This one is challenging, because the model must output the result from left to right. In their follow up work [28] they introduced a method to improve this edge case.

In one of our previous work [40] we investigated the embedding of the tree digit-numbers in the modular addition task. We found that the encoding of the numbers was very important, the model could easily learn the problem with binary encoding, but using one-hot encoding caused grokking effect. I have put a lot of effort to understand the embedding of the numbers, but it turned out that it is very architecture dependent. Some Transformer architectures had some patterns in the embedding, while others had not. One common pattern was the grid pattern: the more-digit numbers was arranged in a grid just like in the decimal system. This pattern was part of the encoding, because a random initialization of a model already had this pattern. However, this pattern was slightly degraded during the training. Another interesting pattern is that some models represented the digits in circle or line. However, its meaning and usefulness was not clear.

Additionally, we can think, that equivalent formulas should be close to each other in the embedding space, because that would make the model easier to predict the output. I found that $123 + 456$ is not close to the number 579, but instead $123 + 456$ gets closer to $291 + 288$ during training. This means that the embedding of equivalent expressions is slightly getting closer to each other, but not to their values. However, this phenomenon only happens, if the dataset is enough complex,

otherwise it will not happen. It is also not clear why this happens, and why some models have this pattern, while others have not.

5 Noisy Majority Problem

A majority function or gate is a logical operator that receives a sequence of bits and returns the majority of the bits. In case of tie, the output should be 0. It is known that the majority function cannot be computed by AC^0 circuits, and some variants of Transformer are in AC^0.

However, in practice, Transformers can perform well on this problem and solve most of the input sequences. [39] In this thesis, I define a noisy majority problem, where the input sequence is a ternary (base 3) number. The goal is the same, but now the digits of 2 should be ignored, i.e. if we remove all digits of 2 from the input sequence, then the output should be the same as in the original majority problem. A detailed definition is given in the next section.

Theoretically, one attention head with dimension of one is enough to solve this problem. In section 5.3, I show a minimal setup created by hand that can solve this problem. However, it turns out that in practice, a transformer can barely learn the problem with this setting. Larger models have better results, but our goal is to find a method that more often solves the problem with the minimal setup. In Section 5.5, I present a method called "halving heads" that tries to achieve it.

Another direction is to understand the model's algorithm: What does one head do? When can we call a head "good"? How do heads interact with each other? These questions are handled in Section 5.4. Certainly, these two directions are not independent, because understanding the algorithm can help to find a better setup.

This project was part of a collaboration work with Zsolt Zombori, András Kornai, Pál Zsámboki and Máté Gedeon. My contribution to this project was the following:

- I elaborated a minimal setup model with one head, which can theoretically solve the noisy majority problem.
- I trained the minimal setup model with different learning rates.
- I implemented and investigated the learned accuracy of heads.
- I implemented and investigated the halving heads method.

5.1 Problem Definition

Let $\Sigma = \{[\text{BOS}], 0, 1, 2, =, 4, 5, [\text{EOS}]\}$ be the language of the problem. ([BOS] and [EOS] tokens represent the beginning and end of sequence.) Let $x = ([BOS], x_1, x_2, \ldots, x_n)$ be the input sequence over $\{0, 1, 2, =, 4, 5\}$ such that there can be only one '=' token in the sequence, and there are only '0', '1' and '2' tokens before the '=' token.

Let n_0 and n_1 the number of '0' and '1' tokens preceding the '=' token, respectively. The output token is determined as follows:

1a. If x_n is '=' and $n_0 \geq n_1$, then the output should be '4'.

1b. If x_n is '=' and $n_0 < n_1$, then the output should be '5'.

2. If x_n is '4' or '5' then the output should be '[EOS]'.

So the model has to learn two tasks: determine the majority of '0' and '1' tokens before the '=' token, and return an end of sequence token after that.

5.2 Technical Details

I trained thousands of one-layer attention-only generative transformer models with different configurations (e.g. dimensions number, head number, etc.). This means that the feed-forward network was omitted. The model also had no position encoding, nor an output matrix (unless otherwise stated).

The dataset consists of 10,000 sentences, split into training, validation, and test sets with a ratio of 75/15/15. The number of digits in each input sequence was randomly chosen from the intervals $[0, 100]$, $[101, 150]$ and $[151, 200]$ for the training, validation, and test sets, respectively. So the model was trained on shorter sequences but had to generalize to longer sequences.

The model was trained with the following hyperparameters: $\text{batch size} = 128, \text{learning rate} = 0.001, \text{weight decay} = 0.01, \text{dropout} = 0.1$. I used warmup training iterations with a parameter of $\frac{2}{1-\beta_2}$ as recommended in [22].

Now I only focus on the performances of the models, I do not consider the training time. Therefore, each model was trained for around 900 epochs, and the best model was selected based on the validation accuracy. The test accuracy was calculated on the best model. Validation and test accuracies are the proportion of correctly predicted sequences on the first task. The second task (i.e. predicting the [EOS] token) was not evaluated.

5.3 Minimal Theoretical Transformer

If we have $d = 1$, $d_h = 1$, we can construct a transformer that solves the problem, if $n_0 + n_1 > 0$. Below, I present a minimal setup that solves the problem. (Since I have one head and one layer, I will omit indices of heads and layers.) Let N be a large number. The embedding and the key, query, value matrices are the following:

$$W_E = \begin{pmatrix} 0 \\ N \\ N+1 \\ 0 \\ 1 \\ N^2 \\ N^2 \\ 0 \end{pmatrix} \begin{matrix} \text{'[BOS]'} \\ \text{'0'} \\ \text{'1'} \\ \text{'2'} \\ \text{'='} \\ \text{'4'} \\ \text{'5'} \\ \text{'[EOS]'} \end{matrix} \qquad W_K = W_Q = W_V = 1$$

We are only interested in query vectors of tokens '=', '4' and '5', so the relevant attentions will be the following:

	$k_{BOS} = 0$	$k_0 = N$	$k_1 = N+1$	$k_2 = 0$	$k_= = 1$	$k_4 = k_5 = N^2$
$q_= = 1$	0	N	$N+1$	0	1	–
$q_4 = q_5 = N^2$	0	N^3	$N^3 + N^2$	0	N^2	N^4

The attention of $q_= \cdot k_4$ and $q_= \cdot k_5$ will not be used at all, because '4' and '5' tokens come only after '=' token. For this reason I denoted its values with '-'.

Since we take a softmax function for each query vector, therefore if the input ends with '=', then only $A_{=,0}$ or $A_{=,1}$ will be significant. If the input is '4' or '5', then $A_{4,4}$ or $A_{5,5}$ will be significant only. Therefore, if $n_0 + n_1 > 0$, then the output of the attention mechanism in the first task will be:

$$\begin{aligned} \mathbf{y}_= &= \frac{v_{BOS} + n_0 e^N v_0 + n_1 e^{N+1} v_1 + n_2 v_2 + e \cdot v_=}{1 + n_0 e^N + n_1 e^{N+1} + n_2 + e} = \frac{e^N (n_0 N + n_1 e(N+1)) + e}{1 + n_0 e^N + n_1 e^{N+1} + n_2 + e} \\ &\approx \frac{n_0 N + n_1 e(N+1)}{n_0 + n_1 e}. \end{aligned}$$

Let a be defined such as $\mathbf{y}_= \geq a \Leftrightarrow n_0 \geq n_1$. We get that $a := \frac{e(n+1)+n}{1+e}$.

Without loss of generality we can assume that the last token is '4' in the second task. Then the output will be

$$\mathbf{y}_4 = \frac{v_{BOS} + n_0 e^N v_0 + n_1 e^{N+1} v_1 + n_2 v_2 + e \cdot v_= + e^{N^4} v_4}{+ n_0 e^N + n_1 e^{N+1} + n_2 + e + e^{N^4} v_4} \approx \frac{e^{N^4} N^2}{e^{N^4}} = N^2.$$

These are only approximations, and are only valid if $n_2 \ll e^N$. However, if we assume that the Transformer is not infinitely precise then with an appropriate large N value, the self attention mechanism outputs these values. Now, we need to define the output layer, and prove that it will produce the correct logits.

$$W_{out} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 4 \end{pmatrix},$$
$$b_{out} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & -a-1 & a+1 & -4 \cdot 3(N+1) \end{pmatrix}.$$

Let us denote the embedding of a token as $x_{<token>}$ and let $e_{<token>}$ denote a vector with 1 at the specified token and 0 elsewhere. Then the output layer produces the following logits

- In the first task, we have

  $$\left(x_= + \mathbf{y}_=\right) W_{out} + b_{out} = (y_= - a)\, e_4 + (a - y_=)\, e_5 + c \cdot e_{EOS}$$

  as logits, where $c = 4 \cdot y_= - 12(N+1) < 0$, which predicts indeed '4' or '5' depending on whether $n_0/n_1 \geq 1$ or not.

- In the second task, we have

  $$\left(x_4 + \mathbf{y}_4\right) W_{out} + b_{out} \approx 4N^2 - 12(N+1) e_{[EOS]} + (N^2 - a - 1) e_4 + + (a + 1 - N^2) e_5$$

  logits, which predicts '[EOS]' if N is large enough.

Note that I did not take advantage of positional encoding here, so we can even skip it if we want by slightly changing the output layer.

To summarize, we can construct a minimal setup that can solve the noisy majority problem. However, note that training a transformer with high test accuracy even with parameters of $d = 2, a = 1$ is not trivial. Most of the time it is not able to learn the problem, there is only a few cases when it can almost perfectly learn the problem.

Our goal was to identify methods that can consistently achieve high accuracy. One approach is to increase the learning rate. Figure 3 shows the test accuracy of the model with $d = 2, a = 1$ for different learning rates over 10-10 runs. We observe that higher learning rates can help, but too large learning rate causes instability.

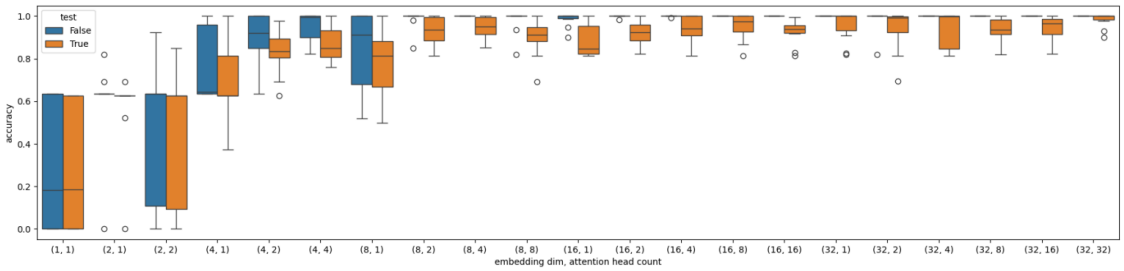

Figure 2: Test and validation accuracies of models with different dimensions and number of heads. 10 runs for each configuration.

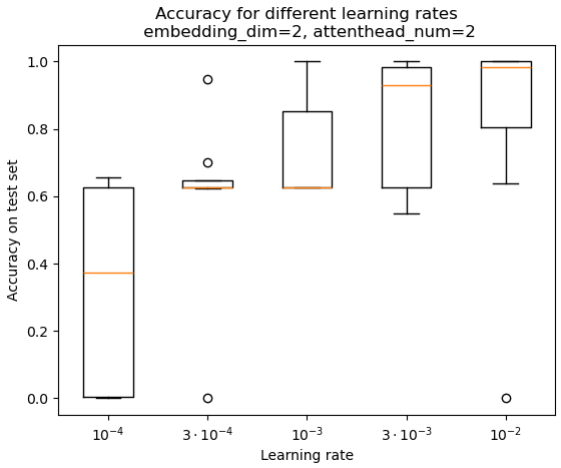

Figure 3: Test accuracies of the model with $d = 2, a = 2$ for different learning rates. 10 runs for each configuration.

5.4 Understanding Heads

I define two metrics to measure the performance of a subset of heads.

Definition 5.1. *Let $H \subseteq \{1, \ldots, a\}$ be a subset of heads. Let $A_\ell(H) \in [0, 1]$ be the model accuracy on test set when all heads are zeroed out except the heads in H. Formally $\boldsymbol{out}_1 := [r^1, \ldots, r^a] \odot e_H + \boldsymbol{out}_0$, where $e^i_H = 1$ if $i \in H$ and 0 otherwise. Let us call it* learned accuracy *of H.*

Definition 5.2. *Let $H = \{h_1, \ldots, h_j\} \subseteq \{1, \ldots, a\}$ be a subset of heads again. Let $\boldsymbol{r} := [r^{h_1}, \ldots, r^{h_j}] \in \mathbb{R}^{n \times j}$. We learn a linear separator $W_s \in \mathbb{R}^j, b_s \in \mathbb{R}$ on the head output of H on the training dataset. Expression of $\boldsymbol{r}W_s + b_s$ predicts the final token: if it is greater than or equal to 0, then the output is '4', otherwise '5'. Let $A_s(H) \in [0, 1]$ be the linear separator accuracy on test set. Let us call it* separation accuracy *of H.*

We expect that $A_s(H) \leq A_s(H')$ if $H \subset H'$, since the model can use more information. However, this one is not guaranteed in practice, as we see in some cases. There are multiple reasons for this: on the one hand the linear separator usually minimizes another expression than the $A_s(H)$, on the other hand finding the best separator requires many compute. But choosing the right separation algorithm highly affects the results. In this thesis, I use support vector classification (SVC) with a linear kernel. I use squared hinge loss as the loss function, and L_2 penalty. Formally, the optimization problem is the following:

$$||W_s||^2 + C \left[\frac{1}{n} \sum_{i=1}^{n} \max(0, 1 - y_i(\mathbf{r}W_s + b_s)) \right]$$

where $y_i \in \{-1, 1\}$ represents the true label ('4' or '5', respectively) of the i-th token. To reduce the impact of the penalty, I use $C = 1000$. This ensures that there is a smaller margin between the two classes.

Figure 5 and 4 shows some example runs with their learned accuracy and separation accuracies for each pair of heads. We can see that around more than half of the heads has high separation accuracy (above 95%), and the other half of the heads has around 50% separation accuracy, which is equivalent to random guess. Sometimes there are heads which are around 80% separation accuracy. If we consider head pairs, then good heads remain good (as expected), and sometimes two bad heads can achieve good separation accuracy. This suggests that two bad heads can sometimes complement each other.

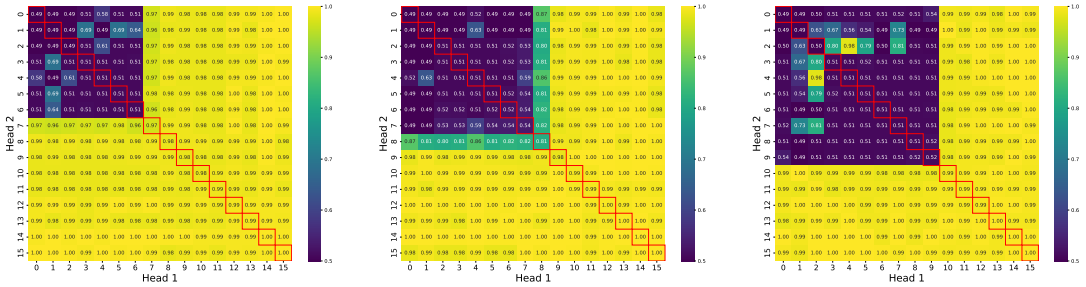

Figure 4: Separation accuracies for each pair of heads for three different runs with hyperparameters of $d = 32, a = 16$. The heads are sorted by their accuracies in the diagonal.

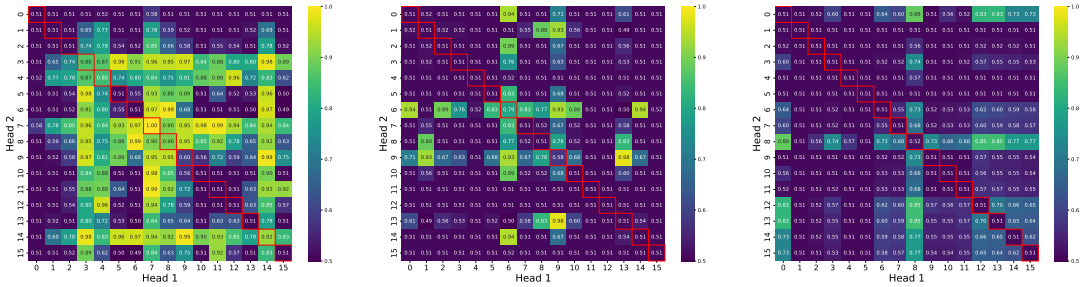

Figure 5: Learned accuracies for each pair of heads for three other runs with hyperparameters of $d = 32, a = 16$.

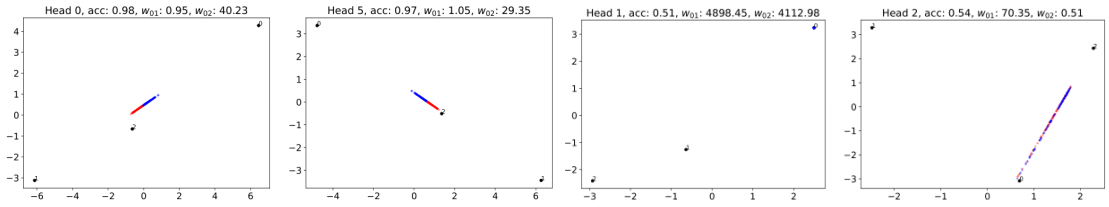

Figure 6: Two successful (left) and failed (right) heads' value vectors for tokens '0', '1' and '2', and their attention head outputs on test dataset.

Learned accuracy's definition tries to measure how much the heads corporate with each other in the models itself. Perfect heads are very rare, but not impossible. E.g. in first example of Figure 5 head 7 has 100% learned accuracy. In the second example, head 9 and 13 together also has near perfect accuracy, but they are very bad alone. In the third example, none of the heads neither alone nor together have good accuracy. We can conclude that usually there are only a few pairs of heads that achieve high learned accuracy. When we consider triplets of heads, we experience very similar results. Sometimes we find one triplet can achieve perfect accuracy, but each two of them has much lower separation accuracy. These results suggest that the model's output layer probably focuses only on a few heads, and the other heads may just help its decision, but the exact mechanism can be complex.

Let $w_{01} := \frac{A_{=0}}{A_{=1}}$, i.e. the attention weight ratio between '0' and '1' tokens. Similarly, let $w_{02} := \frac{A_{=0}}{A_{=2}}$.

Figure 6 shows two successful and two failed heads' with their value vectors for token '0', '1' and '2" and colored dots indicating head outputs for the test samples. Colors blue represents tokens of '4', while red represents '5'. We observe that the successful heads ignore the '2' tokens, i.e. w_{02} is large, and it gives equal weight to '0' and '1' tokens, i.e. $w_{01} \approx 1$. The two failed heads have different behavior: one of them gives more weight to '0' tokens, thus it collapses into a single point. The other one basically ignores the '1' tokens, and gives more weight to '2' tokens, therefore it cannot separate red and blue dots. Let us examine how the weight ratios w_{01} and w_{02} affect the separation accuracy.

Figure 7 shows the separation accuracy for each 16 heads of one run with respect to w_{01} and w_{02} values. We can see that the successful heads have $w_{01} \approx 1$ and $w_{02} \gg 1$, while the failed heads have different values. However, this does not imply that successful heads should necessarily have these ratios, it only implies that the model's successful heads have these ratios. For example Figure 8 shows the separation accuracy if we only modify the attention weights by hand. On the left

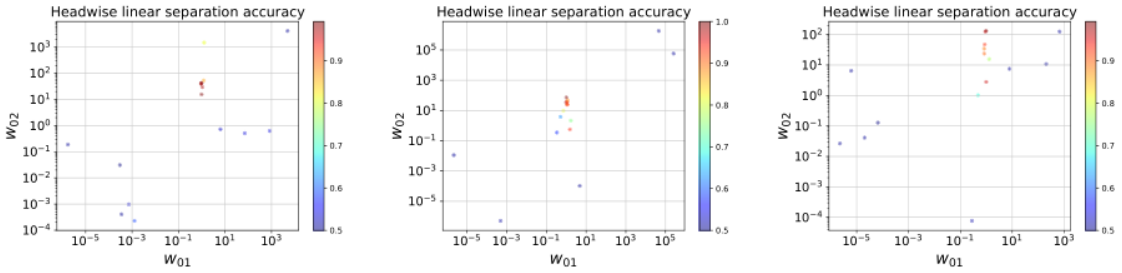

Figure 7: Comparing w_{01} and w_{02} values with separation accuracy on three different runs. Each dot represents one of the 16 trained heads, with color indicating its separation accuracy. The x-axis shows w_{01}, the weight ratio between '0' and '1' tokens, while the y-axis shows w_{02}, the weight ratio between '0' and '2' tokens.

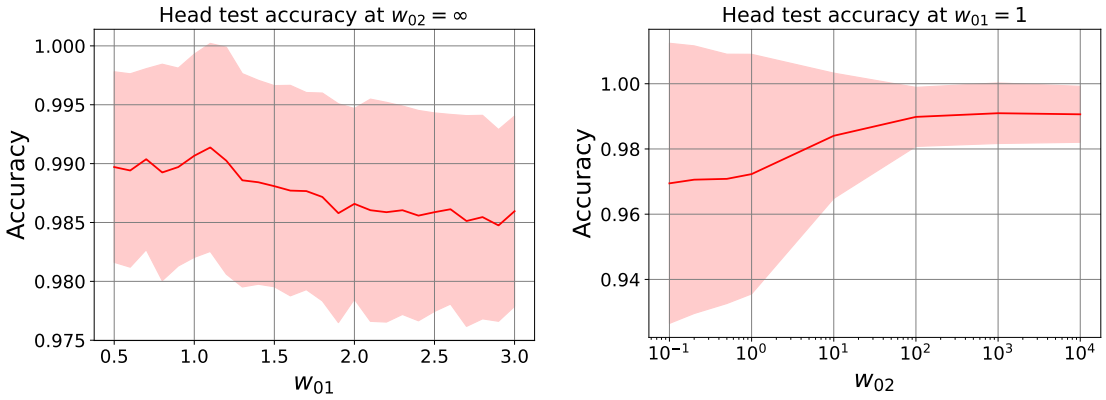

Figure 8: Average separation accuracy of heads with different w_{01} and w_{02} values. On the left plot $w_{02} = \infty$ is manually adjusted and different w_{01} values considered, while on the right plot $w_{01} = 1$ is manually adjusted and different w_{02} values considered.

plot we set $w_{02} = \infty$, i.e. we the model ignores the '2' tokens, and consider different w_{01} values. We can see that the model always achieves near perfect separation accuracy, which is not surprising. The attention heads output will always be the $\frac{A_{=0}n_0v_0+A_{=1}n_1v_1}{n_0+n_1}$, which is easily separable at the point of $\frac{A_{=0}v_0+A_{=1}v_1}{2}$. On the right plot we set $w_{01} = 1$ and considered different w_{02} values. Now, the model clearly has lower separation accuracy if the w_{02} is not enough large, though the difference is not huge.

5.5 Reducing the Number of Heads

As we saw in the previous section, while the model has many heads, not all of them are useful and the model often does not even use them all. Therefore, in this section,

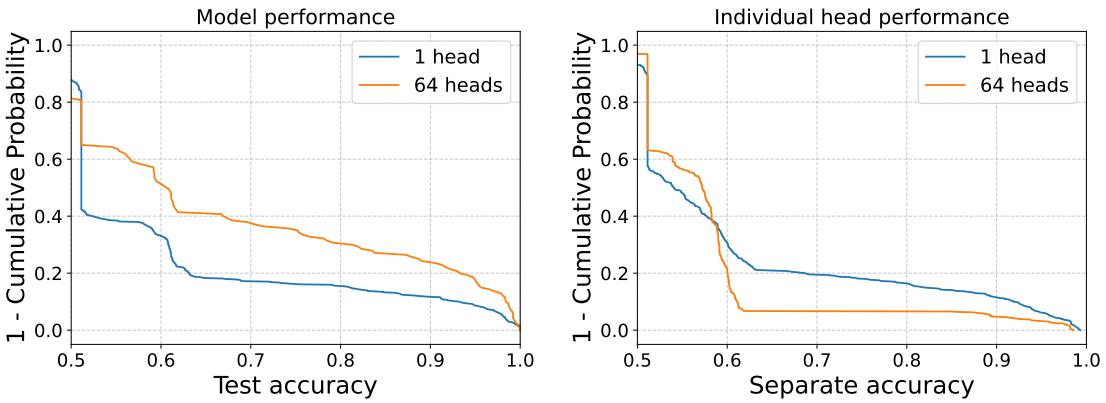

Figure 9: $1 -$ cumulative probability of accuracies for models with 1 and 64 heads. The left plot shows the test accuracy of the models, while the right plot shows the separation accuracy of the individual heads from the models.

I examine whether it is possible to achieve the same performance with fewer heads.

First, I compared two very small models with embedding dimension of 2, but with different numbers of heads. I investigated how each model performed and how individual heads performed according to separation accuracy. First model had 1 head, and the second model had 64 heads. In the latter model, I used an output matrix $W_O \in \mathbb{R}^{128\times 2}$ to convert back the 64 heads' 2-dimensional outputs to the 2-dimensional embedding space.

The results are shown in Figure 9. Both plots show the 1-cumulative probability of test accuracy, i.e. the proportion of runs out of 100 experiments where the accuracy is above a certain value. As we can see in the left plot, the 64-head model achieves high accuracy around two times more often, however, it needs much more compute to train them. Therefore, it can be a good comparison if we consider the individual heads' performance in the two models. The right plot shows again the 1-cumulative probability of the separation accuracy for individual heads. This means that for the 64 heads, I randomly selected 10 runs and calculated the separation accuracy of the individual heads. This gave me 640 data points for the 64-head model. I also trained 640 1-head models, and calculated here the separation accuracy of the individual heads, too. This also gave me 640 data points. As we can see, the 64-head model has much less high accuracy heads on average than the 1-head model. This suggests that it is not worth to keeping many heads in the model, since most of them are not useful. It is a better idea to train a model with many heads, and keep only the best ones. In the following, I present a method to do this.

Training a model with one head is possible, but it is very rare that it achieves

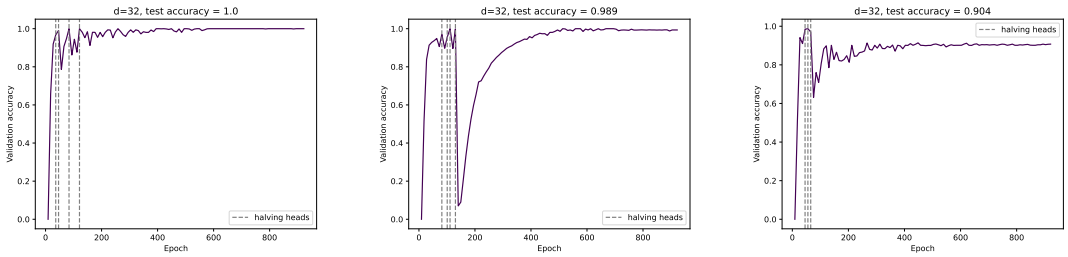

Figure 10: Three typical validation accuracy curves with hyperparameters of $d = 32, a = 16$. Dashed lines indicates the halving heads steps: it masks with zeros half of the active heads with least performance when the validation accuracy reaches 95%.

high accuracy. So, we have the idea of starting the model training with many heads. If the model has high validation accuracy (above 95%), we can drop half of the heads and continue training from there. If the validation accuracy becomes high again, we can drop half of the remaining heads, again. We repeat this process until we have only one head. I call this method as *"halving heads"*. There are multiple ways to choose heads to drop. One trivial way is is to choose them randomly, and we can use it as a baseline. Furthermore, as another baseline, we can drop all the heads except one right at the beginning. So I mask $a - 1$ heads and train the model with only one head. This training with masking method is different from just training a model with one head, because d still remains high, and will not be equal to d_h. Since I found that these two methods had nearly the same performance, I only attach the results of the masking $n - 1$ heads at the beginning.

Another way to choose the heads is using separation accuracy on the individual heads. This is denoted as "svc" in the figures referring to the support vector classification. We can also use the learned accuracy, but based on the previous results it doesn't give a good measure of the performance of the heads.

After running 100 trainings with hyperparameters of $d = 32, a = 16$, I found that the overwhelming majority of runs had 4 halving steps, thus the final models had only one head. Analyzing the validation accuracies of the models, I experienced three different behaviors as shown in Figure 10: 1) the model only had small drops in the accuracy, but quickly recovered, 2) the model had a large drop in the accuracy after the the fourth halving, but it recovered very well, 3) the model had a large drop in the accuracy, but could not recover at all or only partially. The first and second case was the most common, and the third case was rare, but not impossible.

Separation accuracy uses only individual heads, but as we see in the previous

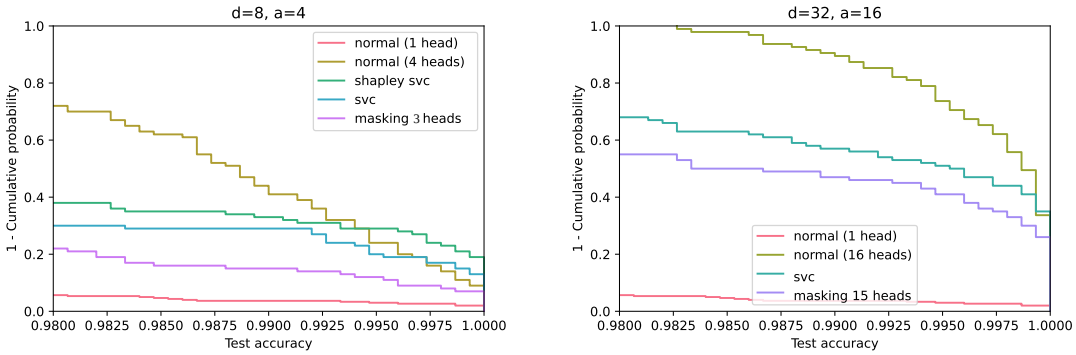

Figure 11: $1-$cumulative probability of accuracies for models with different halving strategies. The left plot shows the results for $d = 8, a = 4$, while the right plot shows for $d = 32, a = 16$.

section, some heads often cooperate with each other. Therefore, we may want to keep heads that can cooperate with each other effectively. Shapley values from game theory is designed to measure the importance of each individual. It provides a fair contribution of the head with respect to the model's performance. The formal definition is as follows:

Definition 5.3. *Given a set of heads H, let $v : 2^N \to \mathbb{R}$ be a characteristic function with $v(\emptyset) = 0$. If $S \subseteq H$ then $v(S)$ value specifies how well the model can perform with only the heads in S. The Shapley value $\varphi_i(v)$ gives a fair contribution of head i with respect to function v:*

$$\varphi_i(v) := \sum_{S \subseteq H \setminus \{i\}} \frac{|S|!(|H| - |S| - 1)!}{|H|!}(v(S \cup \{i\}) - v(S))$$

The formula takes into account all possible subsets of heads that do not contain head i. The term $v(S \cup \{i\}) - v(S)$ represents the marginal contribution of head i to the subset S. These marginal contributions are weighted by the number of possible orderings of the heads in S and $H \setminus S$.

There are multiple options to define the characteristic function v: we can use the learned accuracy or separation accuracy. I found that separation accuracy with SVC gave better results, so I omit the results of the learned accuracy. This method will be denoted as "shapley svc" in the figures.

Figure 11 shows the test accuracy frequencies of different halving algorithms with hyperparameters of $d = 8, a = 4$ and $d = 32, a = 16$ over 100-100 runs. (I did not run the Shapley method for $d = 32, a = 16$, because of the high computational time.) If the model had more than one head at the end (i.e. had less than $\log_2 a - 1$

Model Type	**d=32, a=16**	**d=8, a=4**	**d=2, a=1**
Normal (a heads)	34%	9%	2%
Shapley SVC	-	**19%**	-
SVC	**35%**	13%	-
Masking $a-1$ heads	26%	7%	-

Table 1: Number of models from 100 runs that achieves 100% test accuracy.

halving steps), then I considered it as a failure and marked it with 0% accuracy. I also plotted only the interval $[0.98, 1]$ to see the differences in more clearly, since we are only interested in high accuracy models. As a baseline, I also included the results when we do not mask any heads, and when we have only one head with an embedding dimension of $d = 2$.

We can see that the normal runs, i.e. when we do not mask any heads, are the most successful runs in both cases. They have 72 and 100 runs out of 100 that are above 98% test accuracies. The second best seems to be the Shapley SVC, then followed by the SVC and finally the masking $a - 1$ heads. However, this one is only true if we consider the 98% threshold. If we want to get as many models with 100% test accuracies as possible, then the Shapley SVC seems to be better. Table 1 shows the frequencies with which these models achieve perfect test accuracy. (These are the same values as in the Figure 11.) In both cases, the SVC algorithm more often achieves perfect test accuracy models, and the Shapley SVC even more often in the case of $d = 8$. Note, however, that the 19% with Shapley SVC is a significant improvement over the 9% with the normal run in the case of $d = 8$, but the 35% with SVC and 34% with the normal run cannot be considered as a difference over 100 runs. But it is still a good result, that the SVC can achieve as good results as the normal runs, and almost 10% more often than the masking $a-1$ heads method, which is also an improvement.

Furthermore, it is more relevant to compare the results of halving head strategies to the $d = 2, a = 1$ model. Halving head method succeeds much more frequently, even though the two models have the same number of heads. The small models only achieved two times the perfect test accuracy, while all of the halving head strategies achieved 13-35% perfect test accuracy. However, important to note that the embedding dimension is higher, therefore it is more relevant to compare the halving head method with the initial masking. This is 7-26% perfect test accuracy.

To summarize, normal training is powerful enough in high dimensions, but strug-

gles in lower dimensions. I have introduced a new technique called halving heads to improve it in lower dimensions. We can conclude that the halving head methods are better than the initial masking method, which is also better than 1-head 2-dimensional models.

6 Conclusion

In this thesis, I investigated Transformers from multiple aspects. First, I built up a mathematical framework for Transformers in Section 2. After that, I analyzed their complexity class in Section 3, and showed that any Boolean circuit can be simulated with a chain-of-thoughts. In Section 4, I summarized the results of modular addition from literature and our experiences. Two different papers analyzed how transformers learn to add two numbers modulo N, or two five-digit numbers.

Finally, in Section 5, I introduced the noisy majority problem, where the model has to determine whether zeros or ones form a majority in a sequence while ignoring noise symbols. This task was enough simple to understand fully how the model works and how to improve it. I showed that even a minimal setup with one-dimensional embedding and one attention head can theoretically solve this problem, but training a similar small model consistently is challenging.

I presented how individual heads and pairs of heads work together to accomplish this task. While most heads achieved high performance, some heads gave random outputs. Successful heads tended to give equal weight to '0' and '1' tokens while ignoring '2' tokens, even if in theory it is not needed. Additionally, unsuccessful individual heads sometimes could cooperate with each other and form a successful pair.

Instead of finding a model that is large enough to easily solve the noisy majority problem, the goal was to find a method that can consistently achieve high accuracy with a very small model. I introduced a method called "halving heads" that tries to achieve this. The idea is to train a model with many heads, and then iteratively drop half of the heads until only one head remains. I compared different strategies for selecting which heads to keep, including Shapley values. Results showed that if we focus on the perfect models only, then the halving heads method can achieve results as good as the normal runs, but it had only one head at the end.

As we saw, there are multiple approaches to understand and improve the performance of Transformers on formal language tasks. One way is to analyze the individual heads and attention patterns. Another way is to analyze their embed-

dings and interpret how they are used in the model, and understand how an ideal model should look like. Furthermore, there are numerous ways to approach the problem.

Future work could involve analyzing the learning dynamics in more detail, or starting to evaluate the second task (predicting the [EOS] token). Additionally, it is also relevant to understand more complex languages at this point, e.g. Dyck language [37].

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[2] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv:2004.05150*, 2020.

[3] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *ArXiv*, abs/1904.10509, 2019.

[4] Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J Colwell, and Adrian Weller. Rethinking attention with performers. In *International Conference on Learning Representations*, 2021.

[5] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research (JMLR)*, 2011.

[6] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM.

[7] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Re. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[9] Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez,

Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. https://transformer-circuits.pub/2021/framework/index.html.

[10] Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. Transformer feed-forward layers are key-value memories. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5484–5495, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[11] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[12] 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, 2022.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015.

[14] Xiao Shi Huang, Felipe Perez, Jimmy Ba, and Maksims Volkovs. Improving transformer optimization through better initialization. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 4475–4483. PMLR, 13–18 Jul 2020.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[16] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer, 2020.

[17] András Kornai. *Semantics*. Springer Verlag, 2019.

[18] András Kornai, Michael Bukatin, and Zsolt Zombori. Safety without alignment. *ArXiv*, 2303.00752, 2023.

[19] Michal Koucký. Circuit complexity of regular languages. In S. Barry Cooper, Benedikt Löwe, and Andrea Sorbi, editors, *Computation and Logic in the Real World*, pages 426–435, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[20] Zhiyuan Li, Hong Liu, Denny Zhou, and Tengyu Ma. Chain of thought empowers transformers to solve inherently serial problems. In *The Twelfth International Conference on Learning Representations*, 2024.

[21] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

[22] Jerry Ma and Denis Yarats. On the adequacy of untuned warmup for adaptive optimization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(10):8828–8836, May 2021.

[23] William Merrill and Ashish Sabharwal. A logic for expressing log-precision transformers. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[24] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[25] Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability. In *The Eleventh International Conference on Learning Representations*, 2023.

[26] Jorge Pérez, Javier Marinković, and Pablo Barceló. On the turing completeness of modern neural network architectures. In *International Conference on Learning Representations*, 2019.

[27] Philip Quirke and Fazl Barez. Understanding addition in transformers. In *The Twelfth International Conference on Learning Representations*, 2024.

[28] Philip Quirke, Clement Neo, and Fazl Barez. Increasing trust in language models through the reuse of verified circuits, 2024.

[29] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.

[30] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1), January 2020.

[31] Hinrich Schütze. Word space. In SJ Hanson, JD Cowan, and CL Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 895–902. Morgan Kaufmann, 1993.

[32] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[33] Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. What formal languages can transformers express? a survey. *Transactions of the Association for Computational Linguistics*, 12:543–561, 05 2024.

[34] Max Tegmark and Steve Omohundro. Provably safe systems: the only path to controllable AGI. *ArXiv*, 2309.01933, 2023.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[36] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[37] Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers, 2021.

[38] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences, 2021.

[39] Zsolt Zombori, Pál Zsámboki, Ádám Fraknói, Máté Gedeon, and Andras Kornai. Language models, mathematics, embeddings. In *9th Conference on Artificial Intelligence and Theorem Proving*, 2024.

[40] Ádám Fraknói, András Kornai, and Zsolt Zombori. Embedding mathematical formulas into vector space. In *8th Conference on Artificial Intelligence and Theorem Proving*, 2023.