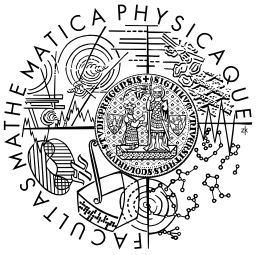

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Tymofii Reizin

Fast Algorithms for Attention Mechanism

Department of Applied Mathematics

Supervisor of the bachelor thesis: doc. Mgr. Petr Kolman, Ph.D.
Advisor of the bachelor thesis: Timothy Chu, Ph.D.
Study programme: Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I want to thank my supervisor doc. Mgr. Petr Kolman, Ph.D. and my advisor Timothy Chu, Ph.D. for their help. I also want to thank Timothy Chu, Ph.D. for giving me interest in this topic and suggesting directions.

Title: Fast Algorithms for Attention Mechanism

Author: Tymofii Reizin

Department: Department of Applied Mathematics

Supervisor: doc. Mgr. Petr Kolman, Ph.D., Department of Applied Mathematics

Advisor: Timothy Chu, Ph.D., Google

Abstract: In this thesis we explore transformers, which are neural networks that excel in tasks such as natural language processing, computer vision and audio processing. Examples of transformer-based models include ChatGPT, BERT, GPT-4, Gemini, Claude3, and more. We explain the basic building blocks and structure of such neural networks. Our main focus is on the attention mechanism that is used in transformers for understanding the context of individual words. We explore a hardness result stating that it is impossible to approximate standard attention in sub-quadratic time with respect to the length of input, assuming a famous conjecture in fine-grained complexity (SETH). We then explore alternative models for attention that can be compute in linear time. Finally we conduct experiments, comparing accuracy of various attention implementations.

Keywords: Machine learning, Large language models, Transformers, Linear algebra, Polynomials

Contents

Introduction **6**

1 General Structure of Transformers **8**
1.1 Components of transformer 8
1.1.1 Input Tokenization 8
1.1.2 Token embedding 9
1.1.3 Positional embedding 10
1.1.4 Attention 10
1.1.5 Multi-head attention 13
1.1.6 Feed-forward layer 14
1.1.7 Layer normalization 15
1.1.8 Unembedding 15
1.1.9 Residual connections 16
1.2 Types of Transformers 16
1.2.1 Encoder-only transformer 16
1.2.2 Decoder-only transformer 17
1.2.3 Encoder-decoder transformer 18

2 Limits of Softmax Attention **21**
2.1 Prerequisite statements 21
2.2 Hardness Result 23

3 Linearized Attention **28**
3.1 Generalized formula for attention 28
3.2 Linearized attention 28
3.3 Masked attention 30

4 Polynomial Attention **31**
4.1 Exploring polynomial kernel 31
4.2 Approximating polynomial attention 32
4.3 More general polynomials 34

5 Experiments **36**
5.1 Single term polynomials of even degree 37
5.2 Squares of general polynomials 38
5.3 Learning squares of general polynomials 40
5.4 Comparing best from different groups 41

Conclusion **44**

Bibliography **45**

Introduction

Transformers are neural networks that excel in natural language processing, computer vision, audio processing. Since they were introduced in 2017 in a paper titled 'Attention Is All You Need' [Vas+17] they have been gaining more and more popularity. Nearly everyone nowadays knows about ChatGPT which is a transformer neural network.

Transformers are usually trained by certain implementations of gradient descent. Thus, during training the model first performs a, so called, forward pass to compute its prediction based on given input. Then its prediction is compared against a 'correct' answer and a loss values is calculated. Then a, so called, backward pass is performed, computing gradients of functions that were used in the computation to descent in the direction of minima. During prediction stage only the forward pass is performed. In this thesis we will only focus on the forward pass. There is no consensus what run time is more important, during training or during prediction. There are arguments for both sides. Nonetheless, it is important to try to improve both of them, since they may be useful for different things. We think that prediction run time is more important, since usually the model is trained once, and then used repeatedly. For example Chat-GPT has been trained once and is now being used hundreds of millions times per day.

The core operations behind transformers is so called attention. It is the mechanism that allows transformers to understand context, that is for individual words to 'attend' to words around them to infer their meaning. The most popular and widely-used mechanism for computing attention, softmax, has a crucial drawback. It has been show by Alman and Song [AS23] that in general it is impossible to compute softmax attention in subquadratic time (with respect to length of the sequence of words), assuming a certain hypothesis from complexity theory holds (The Strong Exponential Time Hypothesis). Thus, people have been looking for other ways to implement attention to achieve more efficient run times.

Researchers have been looking into using various functions for attention that can be decomposed in a specific way that allows for the computation to be performed in linear time with respect to the number of words. Some such functions include certain types of polynomials.

In the first chapter we will explore building blocks and general structure of transformer models. In the second chapter we will look into the result about hardness of attention computation. In the third chapter we will build a framework for more general attention functions. In the fourth chapter we will explore various kinds of polynomial functions that may be used for attention. In the fifth chapter we will implement a transformer model and compare accuracy of different variants of attention.

Notation

Unlike in math, in artificial intelligence and machine learning standard notation is using row vectors. We will thus conform to this notation.

When functions are used row-wise or entry-wise it is usually noted; if it is not, then the reader may assume the function is used entry-wise. For example, division

of a matrix by a number is just division of each its entry by the same number.

We may use either M_i or $M[i,:]$ for the i-th row of matrix M.

By $[n]$ for a positive integer n, we denote the set $\{1, 2, \ldots, n\}$.

We define $[expression]$ as 0 if $expression$ is false and 1 otherwise. For example $[1 \leq 2] = 1$ and $[5 = 8] = 0$.

1 General Structure of Transformers

Transformers are deep feed-forward neural networks. They have been first introduced in 2017 by Vaswani et al in [Vas+17] in a paper titled 'Attention Is All You Need'. The paper's title hints at the core technique transformers use as means of communication between tokens. In this section we will explore the general architecture of transformers.

Transformers have been shown to excel in many fields of artificial intelligence, such as natural language translation, music generation, language modeling, text-to-speech, spam filtering. Perhaps most well-known transformer model, especially by general public, is ChatGPT, that in last years has become very popular. This model, created by OpenAI, acts as a chat-bot that provides answers to user's prompts.

A transformer model would take some input data and produce some output data. For example, a model tasked with answering questions could take 'What is 2 + 2?' as input and produce '2 + 2 is equal to 4.' as output.

Generally, all transformers can be divided into 3 groups: Encoder-only, Decode-only and Encoder-Decoder transformers. We will first describe the building parts of the architecture common to all 3 of these groups, and then proceed to explain the differences in structure. In this section we will describe the structure of transformers, following Phuong and Hutter [PH22].

1.1 Components of transformer

1.1.1 Input Tokenization

The input to the transformer has to be represented in some machine-friendly way, that is, with numbers. Each model has some bijective mapping from all tokens, e.g. words or characters, that can appear in the input to positive consecutive integers, called token IDs. Each input would then be converted to a list of token IDs, and after the model produces a result, this sequence will be converted to an output, since each token ID corresponds to a unique token. We will denotes the alphabet of all the tokens by A and the set of respective token IDs by T, and the bijective mapping between them by $t : A \to T$.

Some examples of ways to perform tokenization for the natural language tasks are:

1. Character-level tokenization. Here the alphabet A is the set of all characters that can appear in the input. For example, it can be the set of all English characters, digits and punctuation signs. This method produces very long token sequences, which is undesirable, since it both impacts computation speed and ability of model to take into account longer 'history' of text to predict next token.

2. Word-level tokenization. Here the alphabet A is the set of all words that can appear in the input, that is sequences of characters separated by spaces.

This method would produce much shorter sequence than character-level tokenization but it would require a very big alphabet and wouldn't be able to deal with new words after end of training, both of which are also undesirable.

3. Subword tokenization. This is the method used in practice nowadays. The alphabet consists of some short common words, such as 'and', 'me', 'are', and word segments that occur often in words, such as 'ing', 'ism', 'less'. All the characters are also included to ensure that it is possible to express all words. This method balances between the first two, having average sequence length and average alphabet size.

The alphabet A is then extended by some special tokens. It could be a mask token that is used to mask some of the words in the sequence and then ask the model to predict masked words. Another example are beginning and end of sequence tokens that are used for representing beginning and end of the sequence respectively. They can be useful, for example, in GPT-like models whose purpose is to answer prompts, and then end of sequence token would indicate that the model should stop generating and output the answer.

The set T is usually taken to be $\{1, 2, \ldots, |A|\}$. A piece of text then would be represented as a list of integers from T, with first and last usually corresponding to beginning and end of sequence tokens, respectively.

1.1.2 Token embedding

After converting tokens to token IDs we once more convert them to a 'better' representation. We represent each token as a row vector in $\mathbb{R}^{d_e}$ where d_e is the embedding dimension. This representation is not fixed but the model changes and improves it during training.

Intuitively, we want to represent each token with a vector such that closely related words are represented by similar vectors, and regions in the space correspond to some concepts from the language. For example the word 'big' could be represented by the vector $[1, 5]$, the word 'huge' by the vector $[2, 9]$, the word 'small' by the vector $[-2, -5]$, and the word 'yellow' by a vector $[-4, 7]$. We can feel the word 'huge' being further in the direction of 'bigness' than 'big', word 'small' being opposite to 'big', and 'yellow' being relatively independent of those.

Let's denote the embedding of a word by the function e. Another example of such intuition, described in [Mik+13], is that $e(\text{'king'}) - e(\text{'queen'})$ could be approximately equal to $e(\text{'boy'}) - e(\text{'girl'})$. Which would mean that 'king' is different from 'queen' in a similar way to 'boy' being different from a 'girl. Since our embedding has multiple dimensions, each of them can represent some different features. For example, let $m = e(\text{'cars'}) - e(\text{'car'})$. Then, intuitively, adding m to a word should move it from singular to the plural form. For example, we would have $e(\text{'person'}) + m = e(\text{'people'})$.

Of course, this intuitive understanding may have no relation to what actually happens in a trained model. In general in machine learning it is very hard to predict how the model would represent the data. Nevertheless, intuitive understanding is very helpful since it provides motivation for the design decisions, and makes

it easier to think about the algorithm. In this case, though, there is research confirming this intuition [Mik+13] is indeed close to truth.

The embedding can be represented by a matrix $W_e \in \mathbb{R}^{|T| \times d_e}$. Then the token with id i is embedded as $W_e[i, :]$ - the i-th row of the matrix W_e. This matrix is a learnable parameter which means that the model learns it during training.

1.1.3 Positional embedding

As we will see later, attention, the only mechanism that makes possible interaction between different tokens, doesn't distinguish tokens at different positions in the input sequence. This means the model, for example, would have no way to tell apart 'A dog attacked a cat' from 'A cat attacked a dog'. This means that we need to somehow encode the position of token into its representation.

This will be accomplished by a positional embedding. Each position in the input sequence of tokens will be associated with a row vector from $\mathbb{R}^{d_e}$. We will represent it with a matrix $W_p \in \mathbb{R}^{l_{max} \times d_e}$ where l_{max} is the maximum allowed length of the input sequence. The positional embedding of position i will then be equal to $W_p[i, :]$ - the i-th row of matrix W_p.

For a token x_i in the input sequence $[x_1, x_2, \ldots, x_n]$, its full embedding is then equal to sum of its embedding and embedding of its position. We will denote it by $F(x_i) = W_e[x_i, ;] + W_p[i, :]$. Intuitively, adding the position embedding moves the vector representation in the direction of 'being i-th token in the sequence'.

The matrix W_p isn't always a learnable parameter. In fact, a lot of models use a fixed positional encoding. The original transformer uses

$$W_p[t, 2i - 1] = \sin \frac{t}{l_{max}^{\frac{2i}{d_e}}}$$
$$W_p[t, 2i] = \cos \frac{t}{l_{max}^{\frac{2i}{d_e}}}$$

for $0 < i \leq \frac{d_e}{2}$. The original paper mentions that they chose this encoding because they hypothesized it would allow the model to easily learn to infer relative positions since for any fixed k, $W_p[t + k, :]$ can be represented as a linear function of $W_p[t]$. They also mention that they have experimented with using learned positional embeddings, and that both methods produced nearly identical results.

The advantage of the model using fixed positional encoding instead of a learnable encoding is the ability to extrapolate to sequences of length not encountered during training.

1.1.4 Attention

We've now reached the core part of the transformers. Transformers use attention to exchange information between different tokens in the sequence. Moreover, in transformers, attention is the only part where different tokens interact, thus, it is important for it to capture as much as possible. The same word can have very different meanings in different contexts, and the purpose of attention is to capture that. For example in the sentence 'The dog didn't cross the river because it was too tired', does 'it' refer to the 'dog' or to the 'river'?

Let x be the embedding of a token we want to predict (that is our model needs to output a value for this token). And let $[c_1, c_2, \ldots, c_n]$ be the embeddings of tokens in the context. Context, similarly to usual meaning of this word, is just the set of tokens influencing meaning of the token we want to predict.

We assign to a token x a query vector $q \in \mathbb{R}^{d_{attn}}$, where d_{attn} is the dimension of the attention space. The meaning of this query vector is, intuitively, like a call to other tokens in the context, saying 'I am a token that wants to learn these pieces of extra information about me'. For example, the token corresponding to the word 'eye' may want to learn its color.

Each token c_i in the context would be assigned two vectors. First is the key vector $k_i \in \mathbb{R}^{d_{attn}}$. The purpose of these values is to respond to query vectors, with key vector being more similar to the query vector meaning it should influence the value of the prediction more. We formalize this notion of similarity with the dot product. The degree to which a context token c_i is important to the token x is determined by $qk^\intercal$, we call this value score. Intuitively the score corresponds to similarity fot the following reasons:

- If two vectors u and v point in a very similar direction, that is, the angle between them is small, the cosine is positive, and thus, their score is very close to the product of their norms $|u||v|$ - a big number.
- If two vectors u and v are almost orthogonal, that is, the angle between them is close to 90 degrees, the cosine is almost 0, and thus, their score is be very close 0.
- If two vectors u and v point in nearly opposite direction, that is, the angle between them is big and obtuse, the cosine is negative, and thus, their score is very close to the negative product of their norms $-|u||v|$ - a large (in magnitude) negative number.

The second vector assigned to the token c_i is the value vector $v_i \in \mathbb{R}^{d_{value}}$, where d_{value} is the dimension of value space. It would seem that it just should be equal to the dimension of the token embedding but having these dimension different would prove useful later when we will work with multi-head attention. Intuitive meaning of this is 'how should c_i influence x'. For example, as we've seen earlier, the word 'big' may have a value vector that moves words in direction of 'bigness'.

The scores are used to derive a distribution over the context tokens. We want to assign to each token c_i from the context a part of the influence on x that belongs to c_i. We want all the parts to be non-negative and sum up to 1. A common choice, and the approach described in the original transformer paper[Vas+17], is the softmax function.

Let $z \in \mathbb{R}^n$ be a row vector. We, for each $i \in n$ define

$$\operatorname{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}.$$

The exponential function of a real number is always positive, thus, there can never occur division by 0. The exponential function is also strictly increasing on $\mathbb{R}$, and

thus doesn't change the order of the scores. We can also write down the softmax function in matrix form. We have

$$\begin{aligned} &\text{softmax}(M) = D^{-1}\exp(M), \\ &\text{where } D = \text{diag}(\exp(M)1^\intercal), \end{aligned} \tag{1.1}$$

where exp is applied element-wise and diag means a diagonal matrix obtained by placing the i-th entry of the input vector as the i-th element on the diagonal. To see why this is equivalent, we can first notice that inverse of a diagonal matrix is obtained just by writing for each element on the diagonal its reciprocal. Matrix D is obtained from a vector that contains the sum of the row i of $\exp(M)$ as its i-th coordinate. Then multiplying D^{-1} by $\exp(M)$ is just dividing each row by the element on diagonal in the corresponding row of D^{-1}. Thus, it's just dividing each element of the row by sum of the row which exactly matches the definition.

Usually we don't just apply the softmax function but first divide all the scores by the square root of the attention dimension. This is done because if we assume that components of q and k are independent random variable, with mean 0 and variance 1, then after the dot product, the mean of $qk^\intercal$ stays 0 but the variance increase to d_{attn}. Thus, via dividing by $\sqrt{d_{attn}}$, we bring the variance back to 1. If this is not done the dot product grows very large, decreasing numerical stability, as mentioned in [Vas+17].

We then sum up the value vectors using the numbers from the distribution obtained from softmax as coefficients. That is, the representation of a token is set to be $\sum_{i=1}^{n} \text{softmax}\left(\frac{qk_i^\intercal}{\sqrt{d_{attn}}}\right) v_i$.

Finally, in reality, this is implemented via matrix multiplications, to make use of parallel computation, in which modern GPUs excel. We compute attention for all tokens in parallel. Let $X \in \mathbb{R}^{l_x \times d_x}$ be the matrix of tokens we want to predict, each row representing a single token. Let $Z \in \mathbb{R}^{l_z \times d_z}$ be the matrix of context tokens, each row, again, representing a single token. The dimensions of the embedding space are denoted by two different values d_x and d_z since we may have different embeddings for tokens we want to predict and context tokens.

The transformations to obtain query, key and value vectors are linear predictor functions. They are represented by matrices $W_q \in \mathbb{R}^{d_x \times d_{attn}}$, $W_k \in \mathbb{R}^{d_z \times d_{attn}}$, $W_v \in \mathbb{R}^{d_z \times d_{value}}$ and biases $b_q \in \mathbb{R}^{d_{attn}}$, $b_k \in \mathbb{R}^{d_{attn}}$, $b_v \in \mathbb{R}^{d_{value}}$. All of these are learnable parameters. We then obtain:

- The query matrix $Q = XW_q + 1^\intercal b_q$. We have $Q \in \mathbb{R}^{l_x \times d_{attn}}$. Each row represents the query vector corresponding to a single token from the original matrix X.
- The key matrix $K = ZW_k + 1^\intercal b_k$. We have $K \in \mathbb{R}^{l_z \times d_{attn}}$. Each row represents the key vector corresponding to a single token from the original context matrix Z.
- The value matrix $V = ZW_v + 1^\intercal b_v$. We have $V \in \mathbb{R}^{l_z \times d_{value}}$. Each row represents the value vector corresponding to a single token from the original context matrix Z.

We then obtain a score matrix $S = QK^\intercal$. We have $S \in \mathbb{R}^{l_x \times l_z}$. Finally the updated representations of the vectors from X are computed as $\tilde{V} = \text{softmax}\left(\frac{S}{\sqrt{d_{attn}}}\right) V$, where the softmax is applied row-wise. We have $\tilde{V} \in \mathbb{R}^{l_x \times d_{value}}$.

The reader may have noticed that by doing attention for a lot of tokens at once we can only use the same context tokens for all of them. This may be undesirable in some use cases. Thus, we introduce an extra step in the middle of computation called masking. We may mask some of the entries of the score matrix by setting them to negative infinity. This way their influences on the predicted value is zero. In particular, if we want the j-th context token to not influence the i-th token, we will set $S_{i,j}$ to negative infinity, before applying the softmax function. Thus, the mask can be represented as a 0/1 matrix of the same dimension as S where 0 represents that we need to mask the corresponding entry in S.

Most common usages of attention can be divided into 3 groups. We will see how all of them fit in a transformer later. First, we have self-attention. In this case, the context is actually the sequence itself, in other words, we have $X = Z$. This can be further split into two groups:

1. Bidirectional or unmasked self-attention. Here, we don't apply any mask, every token attends to all tokens in the sequence. The name 'bidirectional' comes from tokens taking context from both tokens before and after it in the sequence.

2. Unidirectional or masked self-attention. Here we mask from a token all tokens that come after it. This means that we use a mask matrix M such that $M_{i,j} = [j \leq i]$, that is, an upper-unitriangular matrix.

The third group is cross-attention. Here, the context sequence is different from the primary token sequence, and each token from the primary sequence attends to all tokens from the context. One example of a use-case of this are sequence to sequence task, for example, translation from one language to another. Then the context is the text in the initial language while the primary sequence is the translated text.

1.1.5 Multi-head attention

What we have seen in the previous section was a single attention head. Transformers usually use multiple attention heads. If it useful because each head can learn to attend over different properties of words, and thus, the model is able to grasp more context. To achieve that, each attention head has a separate set of learnable parameters. Additionally, the operations on all heads are implemented to perform in parallel, to speed up the computation.

Let h be the number of heads. We will apply the attention procedure described in the previous section h times. We will have d_{value} equal for all heads, that is, each head produces an equal part of the embedding vector. We will also pass the same mask to all heads. We will obtain h predicted value matrices $\tilde{V}_1, \tilde{V}_2, \ldots, \tilde{V}_h$. We concatenate them into a single matrix $\tilde{V} = [\tilde{V}_1, \tilde{V}_2, \ldots, \tilde{V}_h]$. We have $\tilde{V} \in \mathbb{R}^{l_x \times d_{value} H}$. We then apply an output mapping to $\tilde{V}$ to obtain the predicted value vectors of the multi-head attention. This is done so that the results of different heads can communicate between them, to improve the encoding. This mapping is a linear predictor function represented by a matrix $W_o \in \mathbb{R}^{d_{value} H \times d_e}$ and a bias $b_o \in \mathbb{R}^{d_e}$. We then obtain the matrix $\tilde{X} = \tilde{V} W_o + 1^\intercal b_o$, i-th row of which corresponds to the encoding of the i-th token after attending to the context.

We denote the result of multi-head attention for primary sequence X, context sequence Z (they don't have to be different) and a mask matrix M by $\mathrm{MHAttention}(X, Z, M)$ where the last argument may be omitted, and in that cask attention will be unmasked.

A block-diagram of attention and multi-head attention can be seen in Figure 1.1.

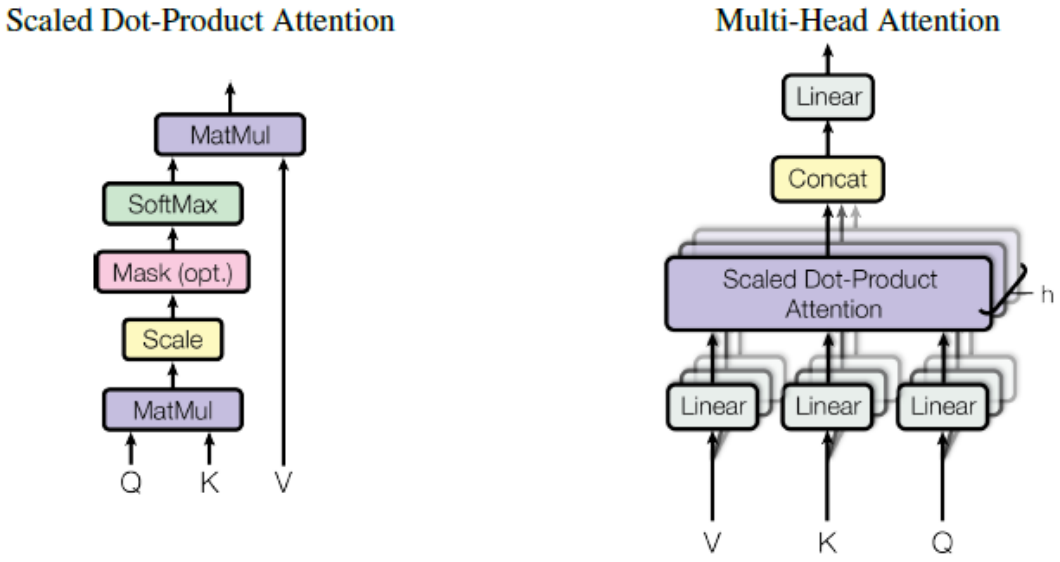

Figure 1.1 On the left is a scheme of a single head of attention. On the right is a scheme of a multi-head attention. Image was taken from the original paper [Vas+17].

1.1.6 Feed-forward layer

Transformers also include position-wise feed-forward layers to improve the model's processing abilities. All of them operate separately and identically on each position in the sequence. They employ multilayer perceptrons with a single hidden layer. This simply means that a linear predictor function is applied to the data, then a non-linear activation function, and then again a linear predictor function. Activation functions introduce non-linearity and make the model capable of learning more convoluted concepts. The transformation can be represented by two matrices $W_{mlp1} \in \mathbb{R}^{d_e \times d_{mlp}}$ and $W_{mlp2} \in \mathbb{R}^{d_{mlp} \times d_e}$, two biases $b_{mlp1} \in \mathbb{R}^{d_{mlp}}$ and $b_{mlp2} \in \mathbb{R}^{d_e}$, and a non-linear activation function f. All of these, except for the function f, are learnable parameters. Common choices for the function f are:

1. ReLU - rectified linear unit. This is a simple activation function defined as

$$\mathrm{ReLU}(x) = \max(0, x).$$

   It was first introduced by Kunihiko Fukushima in 1969 [Fuk69] for visual feature extraction in neural networks.

2. GELU - Gaussian error linear unit. It is defined as

$$\mathrm{GELU}(x) = x\mathbb{P}(X \leq x) = x\Phi(x)$$

where $X \sim \mathcal{N}(0,1)$ is a standard normal random variable, and $\Phi(x)$ is the cumulative distribution function of a standard normal random variable. GELU weights inputs by percentile while ReLU does just by sign, thus, we can think of GELU as a smoothed version of ReLU.

Let X be the matrix of the token sequence that the model is trying to predict, with each row corresponding to a token in the sequence. Then the transformation resulting from application of this layer is

$$\tilde{X} = f(XW_{mlp1} + 1^\intercal b_{mlp1})W_{mlp2} + 1^\intercal b_{mlp2}$$

where the function f is applied element-wise.

Intuitively, we can understand the purpose of this layer as individual tokens 'thinking' on the information they obtain from the context during attention stage.

We denote the result of applying a feed-forward layer to a matrix X by FeedForward(X).

1.1.7 Layer normalization

Layer normalization [BKH16] is introduced to solve the problem of changes in output of one layer causing highly correlated changes in the summed inputs of the next layer. This problem can be fixed by fixing the mean and variance within each layer. The layer norm is represented by two learnable parameters $\gamma, \beta \in \mathbb{R}^{d_e}$, representing the new mean and variance respectively. The parameters are learnable instead of being fixed beforehand to provide the network with the possibility to learn optimal scaling and shifting.

For a given representation of a context token by a vector $x \in \mathbb{R}^{d_e}$ we then compute its mean $m = \sum_{i=1} \frac{x_i}{d_e}$, and variance $v = \sum_{i=1} \frac{(x_i - m)^2}{d_e}$. We then compute the normalized representation

$$\tilde{x} = \frac{x - m}{\sqrt{v}} \odot \gamma + \beta$$

where $\odot$ denotes element-wise multiplication.

We denote the result of applying layer normalization to a matrix X by LayerNorm(X).

1.1.8 Unembedding

We need a procedure to go back from the vectors in the embedding space to actual tokens. This will be done by producing a probability distribution over all the tokens in the alphabet A. The model then samples from this distribution to produce a token that is supplied to the output. Let x be a vector in the embedding space. We will use a learnable matrix $W_u \in \mathbb{R}^{d_e \times |A|}$ to produce a vector of values xW_u, i-th coordinate of which will determine how much the i-th token corresponds to the embedding vector x. We will then use the softmax function introduced earlier to obtain the probability distribution $p = \text{softmax}(xW_u)$. We can again make use of parallelism here and do this for all embeddings at once by calculating softmax(XW_u), where softmax is applied row-wise and $X \in \mathbb{R}^{l_x \times d_e}$ is the matrix, with each row corresponding to a token encoding.

1.1.9 Residual connections

The transformer employs residual connections all over it's structure. Residual connections were introduced in [He+15]. A residual connection is an identity mapping with a short-cut to a layer in the neural network. The identity is simply added to the value at the current layer. For example instead of sending output of an attention computation to the next layer, we first add the initial data to it and only then continue the computation. These have been introduced to help with the vanishing gradient problem during back-propagation in training, because with using a non-linear activation function in a lot of cases the gradient is 0, residual connections help to mitigate that effect. Another reason is that deep networks tend to 'forget' features of the original data, and residual connections help to 'remind' deeper layers of the original state.

1.2 Types of Transformers

We are now ready to explain the actual structure of a transformer model. We will again base our descriptions on [PH22], and the original papers about particular transformers. The core difference between these types of transformers is in attention. The encoder-only transformer uses bidirectional attention while the decoder-only uses unidirectional attention. Finally, the encoder-decoder transformer consists of both an encoder and a decoder (same as in the 'only' versions) but also uses cross-attention to unite their outputs. We will now proceed with the models in order of simplicity. For each we will describe the structure of a particular transformer from a given family, since all of them are the same in important for us parts. But they may differ, for example in the activation function, or order of application of layer-norms.

1.2.1 Encoder-only transformer

We will follow the structure of an encoder-only transformer BERT - Bidirectional Encoder Representation from Transformers [Dev+19]. It has been trained on the task of masked language modeling. Given a piece of text, some tokens in it are masked using a mask token describe earlier in the tokenization section. The task of the model is to predict the masked token. After the training, the model can be fine-tuned for various natural language processing tasks.

The model first performs conversion of the input, which is a single sequence of characters, into token IDs. It uses subword tokenization with a 30000 token vocabulary. The tokens are then converted to vector representations as described in subsection 1.1.2. The positional encodings are then obtained as described in subsection 1.1.3. The two values are added up for each token to obtain initial embeddings. We will denotes them by matrix $E \in \mathbb{R}^{l_x \times d_e}$ where l_x is the lengths of the sequence, and each row corresponds to an embedding of a token. The matrix E is supplied to input of the first encoder block.

The model has L layers of encoder blocks. Let X be the input to an encoder block. An encoder block is comprised of 4 parts.

1. The first is a multi-head bidirectional attention block with H heads as described in subsection 1.1.5. The output of the attention block applied to X

is then added to the initial embeddings to implement the residual connections described in subsection 1.1.9. We thus get $X_1 = X + \text{MHAttention}(X)$.

2. We apply layer normalization as described in subsection 1.1.7 to obtain $X_2 = LayerNorm(X_1)$.

3. The next step is the application of a feed-forward layer as described in subsection 1.1.6; in this model, we use an activation function $f = \text{GELU}$. We again implement residual connections, thus, we have $X_3 = X_2 + FeedForward(X_2)$.

4. We then again normalize the embeddings to obtain output of an encoder block $X_4 = LayerNorm(X_3)$.

Encoder blocks are chained, with the output of the block i supplied to the input of the block $i + 1$.

After applying the encoder blocks the model applies a final linear projection and layer normalization to output of last encoder block X', to obtain

$$\tilde{X} = LayerNorm(GELU(X'W_f + 1^\intercal b_f))$$

where $W_f \in \mathbb{R}^{d_e \times d_f}$ and $b_f \in \mathbb{R}^{d_f}$ are learnable parameters, and d_f is the dimension of final embeddings. We then obtain the probability distribution P on tokens as described in subsection 1.1.8.

The encoder-only transformers are used in various classification tasks. The use of bidirectional attention means that each token infers context from the entire sequence, and thus, it is able to provide general predictions about the entire sequence. It isn't suited well for text generation because it can look into the 'future' during training due to bidirectional attention. This motivates the decoder-only architecture.

1.2.2 Decoder-only transformer

The core difference of a decoder-only transformer from an encoder-only transformer is the usage of unidirectional attention. We will look at the architecture of the most popular decoder-only transformer GPT - Generative Pre-trained Transformer, in particular GPT-2 [Rad+19]. It has a few other differences from BERT, different order of layer normalization and absence of the final linear projection.

Similarly to BERT, GPT also uses subword tokenization, and has a vocabulary of 50257 tokens. It also takes a single sequence as input. It performs input encoding in the same way as BERT, by adding up token and positional embeddings; thus, we won't describe it in detail.

The main difference between the models is the type of the used blocks. The model has L layers of decoder blocks. Let X be the input to a decoder block. A decoder block in GPT also consists of 4 parts, as an encoder block in BERT, but it performs operations in different order.

1. It starts by applying layer normalization to the input to obtain $X_1 = \text{LayerNorm}(X)$, as described in subsection 1.1.7.

2. Then it performs multi-head unidirectional attention with H heads on X_1, as described in subsection 1.1.5, and adds that to the input to implement the residual connection, as described in subsection 1.1.9. We have $X_2 = X + \mathrm{MHAttention}(\mathrm{X}_1, \mathrm{M})$ where M is the upper-unitriangular (all 1s on and above the main diagonal, and 0s below) mask matrix, since we are using unidirectional attention.

3. We then again normalize rows of the matrix to obtain $X_3 = LayerNorm(X_2)$.

4. The model then applies supplies X_3 as input to a feed-forward layer, as described in subsection 1.1.6, and adds its output to X_2, to implement the residual connection. Thus, the output of an entire decoder block is $X_4 = X_2 + \mathrm{FeedForward}(X_3)$.

The decoder blocks are chained, with output of the block i supplied to the input of the block $i + 1$.

GPT then doesn't apply any more linear projections, but applies a layer normalization one last time to the output of last decoder block X', to obtain $\tilde{X} = \mathrm{LayerNorm}(X')$. We obtain the matrix of probability distributions P from $\tilde{X}$ as described in subsection 1.1.8.

Decoder-only transformers excel in sequence modeling tasks. Given beginning of a sequence its task is to predict the next token. They often feed their output to themselves as input, to produce one token after another. Usage of unidirectional attention is motivated by model being able to only use tokens coming before the currently predicted token in the sequence to produce it.

1.2.3 Encoder-decoder transformer

This is the architecture of the original transformer introduced in the 2017 paper by Vaswani et al [Vas+17]. We will follow its structure. It looks like an encoder-only and decoder-only transformers united together by cross-attention. It also takes not a single sequence as the other two, simpler, variants, but two sequences. One is the context sequence supplies as input to the encoder part, and the other is the primary sequence, supplied to the decoder part.

The encoder part of the encoder-decoder transformer is almost identical to a prefix of the encoder-only transformer described in subsection 1.2.1, we thus won't describe it in detail. The only difference lies in using ReLU as activation function in the feed-forward layers, instead of GELU. By prefix we mean that it doesn't perform any operations after the last encoder block since we don't need to generate any distributions for the context sequence but only process it to supply as context for the primary sequence.

The decoder part contains one crucial difference: it contains an extra cross-attention module that is used so that the primary sequence can attend to tokens from the context sequence. It also differs from decoder-only transformer in order of application of layer normalization, thus we describe the structure here in detail. The process of obtaining initial embeddings has the same structure as in GPT, and thus we won't describe it in detail, you can refer to subsection 1.2.2. Then the primary sequence encoded in a matrix X is supplied as input to the first decoder block. We will have L_d layers of decoder blocks. It is worth noting that the number of encoder block layers may be a different number L_e.

Each decoder block will consist of 6 parts. Let matrix X be the input to a block, and let Z be the matrix of embeddings of the context tokens that have been already processed by the encoder part.

1. The model first computes unidirectional multi-head self-attention (subsection 1.1.5) while also implementing a residual connection (subsection 1.1.9). We obtain $X_1 = X + \text{MHAttention(X, M)}$ where M is the upper-unitriangular (all 1s on and above the main diagonal, and 0s below) mask matrix.

2. We then apply layer normalization to get $X_2 = LayerNorm(X_2)$.

3. The next step is applying cross-attention. Here instead of treating X as both primary sequence and context, we use Z for the context. The attention is unmasked since the tokens should be able to attend to entire context. The cross-attention procedure was also described in subsection 1.1.5, since we didn't constraint ourselves to primary sequence and context being the same there. We also, again, add a residual connection. We have $X_3 = X_2 + \text{MHAttention}(X_2, Z)$.

4. We then apply a layer norm to get $X_4 = \text{LayerNorm}(X_3)$.

5. Finally, the data is passed trough a feed-forward layer (subsection 1.1.6), with activation function $f = \text{ReLU}$. Once again we add a residual connection. We get $X_5 = X_4 + \text{FeedForward}(X_4)$.

6. We apply a final layer norm to get output of a decoder block $X_6 = \text{LayerNorm}(X_5)$.

The decoder blocks are chained, with output of block i supplied to the input of block $i + 1$.

We then derive the probability distribution from the output of the last decoder block as described in subsection 1.1.8.

A block-diagram of an encoder-decoder transformer is depicted in the Figure 1.2. Note that the left side represent the encoder part and the right side represent the decoder part.

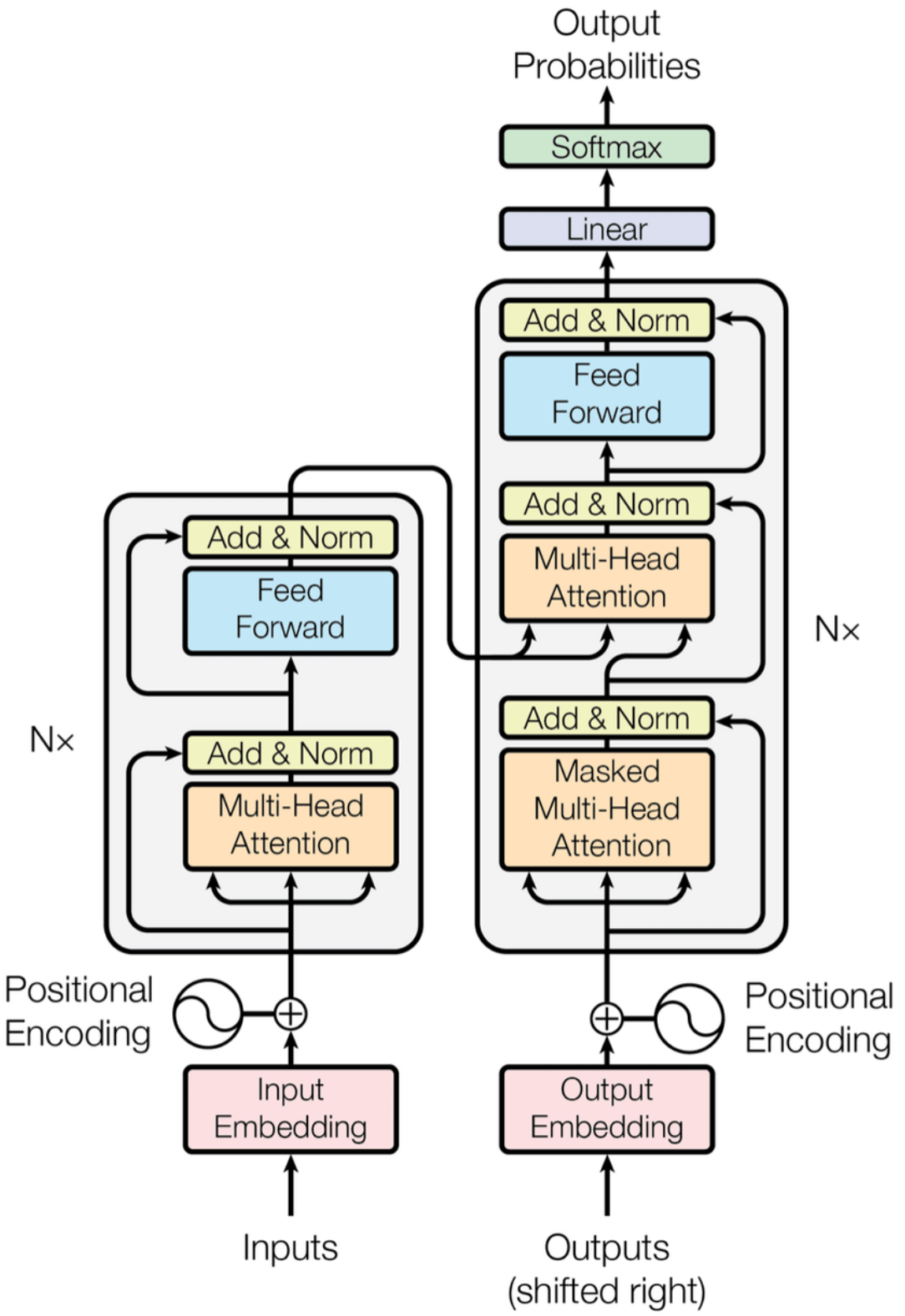

Figure 1.2 Model architecture of an encoder-decoder transformer. Image was taken from the original paper [Vas+17].

2 Limits of Softmax Attention

In this chapter we will see that, in general, it is impossible to calculate approximate softmax attention in subquadratic time, assuming one unproven conjecture. This will motivate search for alternative methods which will make faster attention computation possible. This section is based on a paper by Alman and Song [AS23].

It is pretty clear that attention computation is the runtime bottleneck of transformers since it is the only place where we need to realize a matrix of size $\mathcal{O}(n^2)$. Thus, it is an important goal to try to improve this time.

2.1 Prerequisite statements

In this chapter we will assume that primary and context sequences have the same length n and dimension of attention and values are also the same, denoted by d. We will thus have a query matrix $Q \in \mathbb{R}^{n\times d}$, a key matrix $K \in \mathbb{R}^{n\times d}$ and a value matrix $V \in \mathbb{R}^{n\times d}$. We will work in real-RAM model [Sha78], this model assumes ability to store and perform operations on exact real numbers. The allowed operations include addition, multiplication, subtraction, division and comparison, each of these operations is assumed to take constant time.

We will now state a formal definition of approximate softmax attention, we will use the same idea as in equation (1.1) for softmax.

Definition 1 (Approximate attention computation $AAttc(n, d, B, \varepsilon_a)$)**.**
Let $\varepsilon_a, B > 0$ be parameters. Given three matrices $Q, K, V \in \mathbb{R}^{n\times d}$, such that $||Q||_\infty, ||K||_\infty, ||V||_\infty \leq B$ we need to calculate a matrix T such that

$$||T - D^{-1}AV||_\infty \leq \varepsilon_a$$

where

$$A = \exp\left(\frac{QK^\intercal}{d}\right), \text{ and } D = \operatorname{diag}(A1^\intercal). \tag{2.1}$$

For a matrix M by $||M||_\infty$ we mean $\max_{i,j} |M_{i,j}|$, and exp is applies element-wise.

Remark. Notice that in equation (2.1) we divide by d instead of $\sqrt{d}$ inside of the exponential function, unlike in attention described in previous chapter. This is will be more convenient in the proof, and doesn't make a difference for the definition since result of attention is invariant for different divisor inside the exponential.

We will focus on the setting where $d = \mathcal{O}(\log n)$ and $\varepsilon_a = \frac{1}{poly(n)}$, since we need to model long sequences and want to have small enough error to be able to combine multiple attention computations.

We now define the Strong Exponential Time Hypothesis (SETH) which was introduced by Impagliazzo and Paturi [IP01]. It is a hypothesis from fine-grained complexity theory which studies exact upper bounds for problems solvable in polynomial time. SETH is based on assumption that current SAT algorithms are roughly optimal.

Hypothesis 1 (Strong Exponential Time Hypothesis)**.** For every $\varepsilon > 0$ there is an integer $k \geq 3$ such that k-SAT on formulas with n variables cannot be solve in $\mathcal{O}(2^{(1-\varepsilon)n}$ time, even by a randomized algorithm.

It is a popular conjecture, which have been used to prove a lot of lower-bounds [Wil19].

We will now state a problem which we will reduce to computation of softmax attention and its relation to SETH.

Definition 2 (Approximate hamming nearest neighbour search (ANN))**.** Let $\varepsilon > 0$ be a parameter. We are given two sets $A, B \subset \{0,1\}^n$ with $|A| = |B| = n$ and we need to find $a^\star \in A$ and $b^\star \in B$ such that

$$||a^\star - b^\star||_0 \leq (1+\varepsilon) \min_{a \in A, b \in B} ||a - b||_0. \tag{2.2}$$

Here, for a vector a, $||a||_0$ denotes the L_0 norm - number of non-zero entries. Note that for a difference of two 0/1 vectors a and b it is equal to the number of coordinates where a differs from b.

It was shown by Rubinstein [Rub18] that in general it is impossible to solve ANN in subquadratic time, assuming SETH.

Theorem 1. *Assuming SETH, for every every constant $\delta > 0$, there exist constants $\varepsilon \in (0,1)$ and $C > 0$ such that solving ANN with accuracy parameter ε in dimension $d = C \log n$ requires $\Omega(n^{2-\delta})$ time.*

Remark. Notice that in Theorem 1 we can assume that every vector in both A and B in has half entries equal to 0 and the other half to 1. Given an instance where that doesn't hold we can convert each input vector $v \in \{0,1\}^d$ into $v' = \begin{bmatrix} v & \overline{v} \end{bmatrix} \{0,1\}^{2d}$, where $\overline{v}_i = 1 - v_i$. Obviously v' has half zeroes and half ones, since if v had k ones, $\overline{v}$ would have $d - k$ ones, for a total of $d - k + k = d$ ones in the vector v'. For two vectors $a \in A$ and $b \in B$ we have that $||a' - b'||_0 = 2||a - b||_0$, thus we can just solve the version of the problem with transformed vectors.

We will now state a different version of ANN, for convenience of further analysis.

Definition 3 (Gap approximate nearest neighbour search (Gap-ANN(n, d, t, ε)))**.** Let n and d be two positive integers. Let $t > 0$ be a threshold parameter and $\varepsilon > 0$ be an accuracy parameter. Given two sets of points $A = \{a_1, \ldots, a_n\} \subset \{0,1\}^d$ and $B = \{b_1, , \ldots, b_n\} \subset \{0,1\}^d$, we need to distinguish the following two cases for each $i \in [n]$:

1. There exists $j \in [n]$ such that $||a_i - b_j||_0 \leq t$.
2. For all $j \in [n]$, we have $||a_i - b_j||_0 \geq (1+\varepsilon)t$.

If neither of these cases happens we may output any of the two.

Intuitively, for small t we will be in the case 2, then with increase in t we will enter a gray area where neither of the cases holds, and thus we may receive as output both case 1 and case 2, and then increasing t even further we will be in the case 1. Other way to look at this is that if we got case 1 as output then case 2 definitely does not happen and vice-versa.

We will now prove that Theorem 1 hols for Gap-ANN as well.

Lemma 1. *Assuming SETH, for every every constant $\delta \in (0,1)$, there exist constants $\varepsilon \in (0,1)$ and $C > 0$ such that solving* **Gap-ANN**(n, d, t, ε) *with accuracy parameter ε in dimension $d = C \log n$ requires $\Omega(n^{2-\delta})$ time.*

Proof. We will reduce ANN to Gap-ANN(n, d, t, ε). Let $A, B \in \{0,1\}^d$, $|A| = |B| = n$ be the input to ANN, and ε the accuracy parameter. First, we will iterate over all $k \in \{0, d\}$ (since 0 and d are the smallest and the biggest possible distances, respectively) and call Gap-ANN(n, d, k, ε) on A, B. Let $t^\star$ be the smallest k such that for at least one $i \in [n]$ we received that we are in case 1, and let $i^\star$ be one of those i's. Getting case 1 as output means that we are definitely not in case 2. That means, there exists $j \in [n]$ such that $||a_{i^\star} - b_j||_0 < (1+\varepsilon)t$. On the other hand, we received case 2 for $t^\star - 1$ (notice that we are always in case 2 for $k = 0$, so this is valid), since $t^\star$ is the smallest k for which we received case 1. That implies that we are definitely not in case 1 for $t^\star - 1$. Thus, for all $j \in [n]$ we have $||a_{i^\star} - b_j||_0 > t^\star - 1 \Rightarrow ||a_{i^\star} - b_j||_0 \geq t^\star$. Thus, if we find $j \in [n]$ such that $||a_{i^\star} - b_j||_0 < (1+\varepsilon)t^\star$ we will be done since $t^\star \leq \min_{a \in A, b \in B} ||a - b||_0$. Such a j exists by the choice of $t^\star$. Thus, we can just check all vectors in B to find it. Thus, we have solved ANN.

We will now analyze the complexity. Let ε and C be from Theorem 1 for $\frac{\delta}{2}$. Suppose it is possible to solve Gap-ANN$(n, C \log n, t, \varepsilon)$ in $o(n^{2-\delta})$ time. Calling Gap-ANN for $d+1$ values of t will then take $o(n^{2-\delta} d) = o(n^{2-\delta} \log n) = o(n^{2-\frac{\delta}{2}})$, since $\log n = o(n^c)$ for any $c > 0$. Checking all vectors in B will take $\mathcal{O}(nd) = \mathcal{O}(n \log n) = o(n^{2-\delta})$, since $\delta < 1$. Thus total complexity to solve ANN will be $o(n^{2-\frac{\delta}{2}})$, contradicting Theorem 1. □

2.2 Hardness Result

We are now ready to prove the result about hardness of approximating attention. We will first prove a reduction of gap approximate nearest neighbour search to approximate attention computation.

Lemma 2. *For all constant $C_\gamma \in (0, 0.1)$, $\varepsilon > 0$ and $C > 0$ there exist C_a and C_b such that, if Attcc$(2n, 2C \log n, B = C_b \sqrt{\log n}, n^{-C_a})$ can be solved in time T, then Gap-ANN$(n, C \log n, t, \varepsilon)$ can be solved in time $\mathcal{O}(T + n^{2-C_\gamma})$.*

Proof. We will show an algorithm for Gap-ANN$(n, C \log n, t, \varepsilon)$ with the claimed runtime. Let $c > 0$ be a parameter that will be chosen later. Our algorithm will consider two cases. For $t < c \log n$ we will use a bruteforce algorithm which works in $\setminus^{\in -\mathcal{C}_\gamma}$. In the other case, for $t \geq c \log n$, we will use a reduction to Attcc$(2n, 2C \log n, B = C_b \sqrt{\log n}, n^{-C_a})$ resulting in an $\mathcal{O}(T)$ runtime.

Let $d = C \log n$. Let $a_1, \ldots, a_n, b_1, \ldots, b_n$ be the input vectors to Gap-ANN.

Case 1: $t < c \log n$. In this case we will bruteforce for the answer. First, we will store vectors from the set B in a lookup table (we are not concerned about memory constraints here). Then for each $i \in [n]$ we will iterate over all vectors from B at distance (distance here is the number of coordinates where two vectors differ) at most t from a_i and check if they are present in the lookup table. If there is such vector we output case 1, otherwise we output case 2.

For each $i \in [n]$ there are $\sum_{j=0}^{t} \binom{d}{t}$ vectors at distance at most t from a_i. Thus the total running time will be $\mathcal{O}\left(n \sum_{j=0}^{t} \binom{d}{t}\right)$. Recall that $t < c \log n$ and $d = C \log n$, we will assume $c < \frac{C}{2}$. Using $\binom{a}{b} \leq \left(\frac{ea}{b}\right)^b$ we obtain

$$\begin{aligned} n \sum_{j=0}^{t} \binom{d}{t} &\leq n(t+1) \binom{C \log n}{c \log n} \\ &\leq cn \log n \left(\frac{eC}{c}\right)^{c \log n} \\ &= cn^{1+c} \log n \left(\frac{C}{c}\right)^{c \log n} \\ &= cn^{1+c+c \log \frac{C}{c}} \log n. \end{aligned}$$

Now since, for a fixed C, $\lim_{c \to 0} 1 + c + \log \frac{C}{c} = 0$ and $\log n = O(n^a)$ for any $a > 0$, we can pick c sufficiently small so that $1 + c + \log \frac{C}{c} + 0.1 \leq 2 - C_\gamma$ and $c < \frac{C}{2}$.

Case 2: $t \geq c \log n$. Let C_0 be such that $t = C_0 \log n$. Let $C_b = \sqrt{\frac{40C}{C_0 \varepsilon}}$ and $C_a = 2 + C_b^2 \left(1 + \frac{C_0}{C}\right)$. We will construct input to $\text{Attcc}(\tilde{n}, \tilde{d}, B, \varepsilon_a)$ where $\tilde{n} = 2n$, $\tilde{d} = 2d = 2C \log n$, $B = C_b \sqrt{\log n}$ and $\varepsilon_a = n^{-C_a}$.

We now define the input matrices $Q, K \in \mathbb{R}^{\tilde{n} \times \tilde{d}}$ as

$$Q = B \begin{bmatrix} a_1 & 1_d \\ a_2 & 1_d \\ \vdots & \vdots \\ a_n & 1_d \\ 0_d & 1_d \\ 0_d & 1_d \\ \vdots & \vdots \\ 0_d & 1_d \end{bmatrix} \text{ and } K = B \begin{bmatrix} b_1 & 0_d \\ b_2 & 0_d \\ \vdots & \vdots \\ b_n & 0_d \\ 0_d & 2_d \\ 0_d & 2_d \\ \vdots & \vdots \\ 0_d & 2_d \end{bmatrix}.$$

Each entry of both Q and K is either 0 or B, since input vectors are 0/1. We then have that $||Q||_\infty, ||K||_\infty \leq B$ where $|| \cdot ||_\infty$ denotes the L_∞ norm, that is maximum absolute value among all entries.

Let $\beta = B^2$ and $\tau = e^\beta$. We then have the matrix $A = \exp\left(\frac{QK^\intercal}{\tilde{d}}\right) \in \mathbb{R}^{\tilde{n} \times \tilde{n}}$ as in Definition 1 which looks like

$$A = \begin{bmatrix} \exp\left(\frac{\beta a_1 b_1^\intercal}{\tilde{d}}\right) & \cdots & \exp\left(\frac{\beta a_1 b_n^\intercal}{\tilde{d}}\right) & \tau & \cdots & \tau \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \exp\left(\frac{\beta a_n b_1^\intercal}{\tilde{d}}\right) & \cdots & \exp\left(\frac{\beta a_n b_n^\intercal}{\tilde{d}}\right) & \tau & \cdots & \tau \\ 0 & \cdots & 0 & \tau & \cdots & \tau \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \tau & \cdots & \tau \end{bmatrix}.$$

Note that we do not compute matrix A in our algorithm.

For all pairs of $i, j \in [n]$ we have

$$\begin{aligned} A_{i,j} &= \exp\left(\frac{\beta a_i b_j^\intercal}{\tilde{d}}\right) \\ &\leq \exp\left(\frac{\beta d}{\tilde{d}}\right) \\ &= \exp\left(\frac{\beta}{2}\right) \\ &\leq \exp\left(\beta\right) \\ &= \tau \end{aligned}$$

where the first inequality follows from a_i and b_j being 0/1 vectors. On the other hand we also have $A_{i,j}$ for all pairs $i, j \in [n]$ since they are values of the exponential function. We thus have $n\tau \leq (A1^\intercal)_i \leq 2n\tau$ for all $i \in [n]$ from the structure of matrix A (each row has n entries equal to τ on the right, and the entries on the left are upper-bounded by τ). Thus, we also have $n\tau \leq D_{i,i} \leq 2n\tau$ for all $i \in [n]$, where D is from the Definition 1.

Let

$$\tilde{t} = \frac{\exp\left(\frac{1}{4}\beta\left(1 - \frac{t}{d}\right)\right)}{6n\tau}.$$

We then have

$$\begin{aligned} \tilde{t} &= \frac{1}{6n\tau}\exp\left(\frac{1}{4}\beta\left(1 - \frac{t}{d}\right)\right) \\ &= \frac{1}{6}\exp\left(\frac{1}{4}\beta\left(1 - \frac{t}{d}\right) - \beta - \log n\right)) \\ &= \frac{1}{6}\exp\left(-0.75\beta - 0.25\frac{\beta t}{d} - \log n\right) \\ &= \frac{1}{6}\exp\left(-0.75C_b^2 \log n - 0.255C_b^2 \log n \frac{C_0}{C} - \log n\right) \\ &= \frac{1}{6}\exp\left(-0.75C_b^2 \log n - 0.25C_b^2 \log n \frac{C_0}{C} - \log n\right) \\ &\geq n^{-C_a} \\ &= \varepsilon_a \end{aligned}$$

where the penultimate step succeeds for big enough n (due to division by 6 it may fail for $n < 6$).

We will now define $V \in \mathbb{R}^{\tilde{n} \times \tilde{d}}$ - the last input matrix for Attcc. Let $v = \begin{bmatrix} 1_n \\ 0_n \end{bmatrix}$, then $V = \begin{bmatrix} v & 0_{\tilde{n}} & \cdots & 0_{\tilde{n}} \end{bmatrix}$. Then the output of Attcc will be a matrix U such that

$$||U - D^{-1}AV||_{L_\infty} < \varepsilon < \tilde{t}.$$

Let u be the first row of U. We then have

$$||u - D^{-1}Av||_{L_\infty} < \tilde{t}$$

and, in particular,

$$||u - (D^{-1}Av)_{||}|_{L_\infty} < \tilde{t} \tag{2.3}$$

for all $i \in [n]$. We will use u_i to determine the answer for i.

Using Remark 2.1 we have $||a_i||_0 = ||b_j||_0 = \frac{d}{2}$ for all $i, j \in [n]$. Thus, for any pair $i, j \in [n]$ we have

$$\begin{aligned}\frac{a_i b_j^\intercal}{d} &= \frac{1}{2d}\left(||a_i||_0 + ||b_j||_0 - ||a_i - b_j||_0\right) \\ &= \frac{1}{2d}\left(\frac{d}{2} + \frac{d}{2}||a_i - b_j||_0\right) \\ &= \frac{1}{2} - \frac{||a_i - b_j||_0}{2},\end{aligned}$$

where the first equality is true because dot product for 0/1 vectors is the number of coordinates where they both have a 1.

Recall that the goal is to distinguish two cases for each $i \in [n]$:

1. exists $j \in [n]$ such that $||a_i - b_j||_0 \leq t$;
2. for all $j \in [n]$ we have $||a_i - b_j||_0 \geq (1+\varepsilon)t$.

We will decide that by checking if u_i is less or greater than $\tilde{t}_0 = 2\tilde{t}$. We now consider these 2 cases separately for a fixed $i \in [n]$.

Case 2.i: There exists a $j \in [n]$ such that $||a_i - b_j||_0 \leq t$. We then have

$$\begin{aligned}\frac{\beta a_i b_j^\intercal}{\tilde{d}} &= \frac{\beta a_i b_j^\intercal}{2d} \\ &= \frac{\beta\left(\frac{1}{2} - \frac{||a_i - b_j||_0}{2d}\right)}{4} \\ &\geq \frac{\beta\left(1 - \frac{t}{d}\right)}{4}.\end{aligned} \tag{2.4}$$

Which implies that

$$\begin{aligned}u_i &\geq \frac{\exp\left(\frac{\beta\left(1-\frac{t}{d}\right)}{4}\right)}{2n\tau} - \tilde{t} \\ &= 3\tilde{t} - \tilde{t} \\ &= \tilde{t}_0,\end{aligned}$$

where the first step follows from the inequality (2.4), inequality (2.3) and the fact that $D_{i,i} \leq 2n\tau$, since u_i has to be at least as big as sum of first n entries of row i.

Thus, if $u_i \geq \tilde{t}_0$ we output case 1.

Case 2.ii: For all j we have $||a_i - b_j||_0 \geq (1+\varepsilon)t$. We then have

$$\frac{\beta a_i b_j^\intercal}{\tilde{d}} \geq \frac{\beta\left(1 - \frac{t(1+\varepsilon)}{d}\right)}{4}. \tag{2.5}$$

We then have

$$\begin{aligned} u_i &\leq \frac{n \cdot \exp\left(\frac{1}{4}\beta\left(1-\frac{t(1+\varepsilon)}{d}\right)\right)}{n\tau} + \tilde{t} \\ &= \frac{\exp\left(\frac{1}{4}\beta\left(1-\frac{t}{d}\right)\right)}{2n\tau} + \frac{2n}{\exp\left(\frac{\beta\varepsilon t}{4d}\right)} + \tilde{t} \\ &< 3\tilde{t} \cdot \frac{2n}{6n} + \tilde{t} \\ &= 2\tilde{t} \\ &= \tilde{t}_0, \end{aligned}$$

where the first step follows from the inequality (2.5), inequality (2.3) and the fact that $D_{i,i} \geq n\tau$, since u_i cannot be greater than sum of first n entries of row i. And the third step follows from inequality (2.6).

$$\begin{aligned} \exp\left(\frac{\beta\varepsilon t}{4d}\right) &= \exp\left(\frac{\beta\varepsilon C_0}{4C}\right) \\ &= \exp\left(\frac{\beta\varepsilon C_0}{4C}\right) \\ &= \exp\left(\frac{C_b^2 \log n \cdot \varepsilon C_0}{4C}\right) \\ &= \exp\left(\frac{\frac{40C}{C_0\varepsilon}\log n \cdot \varepsilon C_0}{4C}\right) \\ &= \exp\left(\frac{\frac{40C}{C_0\varepsilon}\log n \cdot \varepsilon C_0}{4C}\right) \\ &= \exp(10\log n) \\ &= n^{10} \\ &> 6n, \end{aligned} \tag{2.6}$$

where the last step holds for big enough n.

Note that if case 1 happens, we have $u_i \geq \tilde{t}_0$ and thus will never output case 2, and vice-versa.

□

We are now ready to prove the main result of this chapter

Theorem 2. *Assuming SETH, for every sufficiently small $\delta > 0$, there are constants $C > 0$, $C_\alpha > 0$ and $C_\beta > 1$ such that AAttC($n, C\log n, C_\beta\sqrt{\log n}, n^{-C_\alpha}$) (definition 1) requires $\Omega(n^{2-\delta})$ time.*

Proof. The proof trivially follows from Lemma 1 and Lemma 2. □

3 Linearized Attention

The limitations of softmax attention motivate seeking other ways to implement attention. The lofwer bound of Alman and Song [AS23], as described in chapter 2, shows that it is, in general, impossible to approximate softmax attention in subquadratic time, assuming the strong exponential time hypothesis. We may try to use other, perhaps 'friendlier' functions, to reduce the time it takes to compute attention. This was explored by Katharopoulos et al. [Kat+20], and we base this chapter on materials from this paper.

3.1 Generalized formula for attention

Given a query matrix $Q \in \mathbb{R}^{l_x \times d_{attn}}$, a key matrix $K \in \mathbb{R}^{l_z \times d_{attn}}$ and a value matrix $V \in \mathbb{R}^{l_z \times d_{value}}$, we can write down updated representations of tokens after attention as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\intercal}{\sqrt{d_{attn}}}\right)V \tag{3.1}$$

as we've seen in subsection 1.1.4. Here we only consider unmasked attention for now. If we expand equation (3.1) to see how the softmax function works, we obtain

$$\text{Attention}(Q, K, V)_i = \frac{\sum_{j=1}^{l_z} \exp(Q_i K_j^\intercal) V_j}{\sum_{j=1}^{l_z} \exp(Q_i K_j^\intercal)} \tag{3.2}$$

where for a matrix M we denote by M_i its i-th row. Writing the equation (3.1) in this form motivates considering a more general function in place of exponent of the dot product in equation (3.2):

$$\text{Attention}(Q, K, V)_i = \frac{\sum_{j=1}^{l_z} \text{sim}(Q_i, K_j) V_j}{\sum_{j=1}^{l_z} \text{sim}(Q_i, K_j)} \tag{3.3}$$

, where $\text{sim}(A, B)$ is a similarity function between two vectors. For example, if we set $\text{sim}(A, B) = \frac{\exp(AB^\intercal)}{\sqrt{d_{attn}}}$ in equation (3.3) we obtain equation (3.2). One restriction we want to impose on $\text{sim}(A, B)$ is non-negativity. We want for the output of a similarity function to always be non-negative, so that we are always able to normalize it into a distribution. Thus, we only consider functions $\text{sim} : \mathbb{R}^{d_{attn}} \times \mathbb{R}^{d_{attn}} \to \mathbb{R}^+$ where $\mathbb{R}^+$ denotes non-negative reals.

3.2 Linearized attention

The paper [Kat+20] considers usage of kernel functions for similarity function in equation (3.3). We now define kernel functions.

Definition 4. A function $k : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is said to be a kernel function if there exists a function $\phi : \mathbb{R}^n \to \mathbb{R}^m$ such that $\forall x, y \in \mathbb{R}^n : k(x, y) = \phi(x)\phi(y)^\intercal$. The function ϕ is called a feature representation or mapping.

If we consider a non-negative kernel function $k : \mathbb{R}^{d_{attn}} \times \mathbb{R}^{d_{attn}} \to \mathbb{R}$ with feature mapping $\phi : \mathbb{R}^{d_{attn}} \to \mathbb{R}^m$, we can rewrite equation (3.3) as

$$\text{Attention}(Q, K, V)_i = \frac{\sum_{j=1}^{l_z} k(Q_i, K_j)V_j}{\sum_{j=1}^{l_z} k(Q_i, K_j)} = \frac{\sum_{j=1}^{l_z} \phi(Q_i)\phi(K_j)^\intercal V_j}{\sum_{j=1}^{l_z} \phi(Q_i)\phi(K_j)^\intercal} \tag{3.4}$$

which can be further simplified due to distributive property of matrix multiplication over addition to obtain

$$\text{Attention}(Q, K, V)_i = \frac{\phi(Q_i)\sum_{j=1}^{l_z} \phi(K_j)^\intercal V_j}{\phi(Q_i)\sum_{j=1}^{l_z} \phi(K_j)^\intercal}. \tag{3.5}$$

We cannot divide numerator and denominator by $\phi(Q_i)$ since those are vectors, not numbers; the denominator becomes a number only after multiplication of $\phi(Q_i)$ with $\sum_{j=1}^{l_z} \phi(K_j)^\intercal$. Using the same tricks as for obtaining matrix form of softmax in equation (1.1), the equation (3.5) can be rewritten in matrix form to obtain

$$\begin{aligned} \text{Attention}(Q, K, V) &= D^{-1}\phi(Q)\phi(K)^\intercal V, \\ \text{where } D &= \text{diag}(\phi(Q)\phi(K)^\intercal 1^\intercal) \end{aligned} \tag{3.6}$$

where ϕ is applied row-wise.

For simplicity of calculation, let $n = \max(l_x, l_z)$. Also let T be the time it takes to compute function ϕ for a single vector. We now can make use of associativity of matrix multiplication and first compute $A = \phi(K)^\intercal V$ in $\mathcal{O}(nT + nd_{attn}d_{value})$ which is linear in n if we treat T, and dimensions of attention and values as constants, which makes sense since they are fixed for a given model. We have that $A \in \mathbb{R}^{d_{attn} \times d_{value}}$. We can then compute $\phi(Q)A$ in $\mathcal{O}(nT + nd_{attn}d_{value}) = \mathcal{O}(n)$ time. By identical analysis the calculation of $\phi(Q)\phi(K)^\intercal 1^\intercal$ can also be done in $\mathcal{O}(n)$ time. Notice that in real implementation we don't have to realize this large matrix D^{-1} (it has size of $\mathcal{O}(n^2)$), but instead can just divide the entries of the matrix in the denominator by values stored in the vector $\phi(Q)\phi(K)^\intercal 1^\intercal$. This achieves $\mathcal{O}(n)$ time and memory complexity for the entire attention computation. It's a significant speedup in comparison with $\mathcal{O}(n^2)$ time it takes to compute softmax attention. We have to be careful though to use ϕ such that T is smaller than realistic n, since if it is very big, this becomes impractical.

Remark. Notice that the described approach corresponds to just computing the sums in equation (3.5) once and then using them for all queries.

In [Kat+20] a simple feature mapping $\phi(v)_i = \text{ELU}(v)_i + 1$ is considered, where ELU is the exponential linear uni activation function [CUH16] defined as

$$\text{ELU}(x) = \begin{cases} x, x > 0 \\ \alpha(e^x - 1), x \leq 0 \end{cases}$$

where $\alpha > 0$ is a parameter which is taken to be 1 in this case. They use ELU instead of ReLU to avoid getting 0 gradient when v_i is negative. Using this feature representation results in having $T = d_{attn}$ which is a very practical value. The usage of this function instead of just taking raw dot products is necessary for non-negativity of the kernel function defined by this feature map. The paper [Kat+20] claims that attention implemented with this feature map has performance close to that of softmax attention while obviously being significantly faster. We will check those claims in our experiments.

3.3 Masked attention

In previous parts of this chapter we assumed usage of unmasked attention for simplicity of description. We will now see how masked attention can be computed with the described methods. We won't describe how to do this for general mask, since, in general, it is impossible to do that faster than $\mathcal{O}(n^2)$ since we may need to mask any subset of values of the big $\mathcal{O}(n^2)$ sized matrix $QK^\intercal$. Thus, we will work with unidirectional attention used in decoders (subsection 1.2.2). That is, for a given token we will only attend to tokens before it in the sequence. Considering this, the equation (3.3) changes into

$$\text{Attention}(Q, K, V)_i = \frac{\sum_{j=1}^{i} \text{sim}(Q_i, K_j)V_j}{\sum_{j=1}^{i} \text{sim}(Q_i, K_j)}$$

and for a given kernel, following the equation (3.5) it becomes

$$\text{Attention}(Q, K, V)_i = \frac{\phi(Q_i)\sum_{j=1}^{l_z} \phi(K_j)^\intercal V_j}{\phi(Q_i)\sum_{j=1}^{l_z} \phi(K_j)^\intercal}. \tag{3.7}$$

We can use prefix sums to perform this calculation in $\mathcal{O}(n)$ time as well using remark 3.2. We will define $N_i = \sum_{j=1}^{l_z} \phi(K_j)^\intercal V_j$ and $D_i = \sum_{j=1}^{l_z} \phi(K_j)^\intercal$, then the equation (3.7) becomes

$$\text{Attention}(Q, K, V)_i = \frac{\phi(Q_i)N_i}{\phi(Q_i)D_i}. \tag{3.8}$$

Notice that both N_i and D_i can be computed once for all i and then reused for all queries. Finally, we can compute N_{i+1} from N_i in constant time from $N_{i+1} = N_i + \phi(K_{i+1})^\intercal V_{i+1}$, and likewise for D_{i+1}. Since there are only $\mathcal{O}(n)$ times we need to perform this, the total calculation still takes $\mathcal{O}(n)$ time.

4 Polynomial Attention

In this chapter, we will explore replacing softmax attention with a polynomial of dot product, building up on the generalized formulas for attention from Chapter 3. In the first two sections we explore usage of polynomial kernels suggested in [KMZ24]. In the third section we explore usage of a more general class of polynomials. We won't describe handling of masked attention since it can be implemented as described in subsection 3.3.

4.1 Exploring polynomial kernel

In this section we will explore usage of kernels which are polynomials of dot product. Recalling our requirement on the similarity function to be non-negative (subsection 3.1), we only consider kernels of form

$$k(x, y) = \left(\frac{xy^\intercal + \alpha}{\beta}\right)^p$$

where p is an even positive integer, $x, y \in \mathbb{R}^{d_{attn}}$ are vectors and α, β are real numbers. It is obvious that changing value of β will not change attention values (we can take out $\frac{1}{\beta^p}$ from the sum and divide both numerator and denominator by it) but selecting an appropriate β will help with numerical stability, similarly to softmax attention.

We observe that in the case when $x1^\intercal = 0$ and $y1^\intercal = 0$, that is, the entries of both vectors have mean 0, we have

$$\frac{xy^\intercal + \alpha}{\beta} = x'y'^\intercal$$

for

$$x' = \frac{x}{\sqrt{\beta}} + \sqrt{\frac{\alpha}{\beta d_{attn}}} 1_{d_{attn}},$$
$$y' = \frac{y}{\sqrt{\beta}} + \sqrt{\frac{\alpha}{\beta d_{attn}}} 1_{d_{attn}}.$$

In other words, x' and y' are obtained from x and y by applying the same rescaling and bias. We thus add an additional layer normalization (subsection 1.1.7) to obtain Q' and K' from query and key matrices and use a kernel of the form

$$k(x, y) = (xy^\intercal)^p$$

for even positive integers p. For simplicity, we will abuse notation and use Q and K in place of Q' and K'.

The attention formula now looks like

$$\text{Attention}(Q, K, V)_i = \frac{\sum_{j=1}^{l_z} (Q_i K_j^\intercal)^p V_j}{\sum_{j=1}^{l_z} (Q_i K_j^\intercal)^p}.$$

Unlike softmax attention, here the denominator may be very close to 0. To avoid that we add 1 to it to obtain

$$\text{Attention}(Q,K,V)_i = \frac{\sum_{j=1}^{l_z}(Q_iK_j^\intercal)^pV_j}{1+\sum_{j=1}^{l_z}(Q_iK_j^\intercal)^p}.$$

Notice that in this case the weights applied to values for a given token will not sum up to 1, and thus we won't have a distribution. That is not a problem because the model will just learn to work with that, since it is just simple scaling. For example the model can learn the value matrix to be scaled by an appropriate parameter, and then the model will be identical. Notice that the values are still non-negative. Thus, this measure is applied just for numerical stability.

We have introduced this class of functions as kernels but have not yet provided a corresponding feature map. We first define the Kronecker product for two vectors.

Definition 5 (Kronecker product)**.** Let $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ be vectors. We say that the vector $(x_1y_1, x_1y_2, \ldots, x_1y_m, x_2y_1, x_2y_2, \ldots, x_2y_m, \ldots, x_ny_m)$ is the Kronecker product of x and y. We denote it by $x \otimes y$.

We also define a Kronecker power $x^{\otimes p}$ recursively as $x \otimes x^{\otimes(p-1)}$ and $x^{\otimes 1} = x$. We are now ready to introduce a proposition that will help us to obtain the required feature map.

Proposition 1. *For all vectors $x, y \in \mathbb{R}^n$ we have $(xy^\intercal)^p = x^{\otimes p}\,(y^{\otimes p})^\intercal$.*

Proof. We have

$$(xy^\intercal)^p = \sum_{i_1,i_2,\ldots,i_p}\prod_{j=1}^{p} x_{i_j}y_{i_j}$$

by the binomial theorem. We can look at $x^{\otimes p}$ as a flattening into a vector of a p-dimensional array, where at position $(i_1, \ldots, i_p)$ we have $\prod_{j=1}^{p} x_{i_j}$. The same holds for $y^{\otimes p}$. Thus, by tacking their dot product we obtain exactly the right-hand side of equation 4.1. □

Thus, for a kernel $k(x,y) = (xy^\intercal)^p$ we can use a feature representation $\phi(x) = x^{\otimes p}$. We may write down polynomial attention in matrix form using equation (3.6).

The paper [KMZ24] claims that polynomial attention with $p \geq 4$ performs with similar accuracy compared to softmax attention.

For our attention computation with attention dimension d_{attn}, computing the feature map $\phi(x)$ would take $\mathcal{O}(d_{attn}^p)$ time. This is constant in n (where n is defined as in subsection 3.2) but for big p this constant is going to be huge which makes it impractical, and only worth it if $d_{attn}^p < n$ for realistic values of n. This motivates the next section.

4.2 Approximating polynomial attention

In this section we will denote d_{attn} by h to not clutter the notation. Since h^p can be very big, we may try to approximate polynomial attention without

computing feature maps explicitly, to speed up the algorithm. In this section we will only consider primary and context sequences having the same length n as in subsection 3.2 for simplicity of analysis. We will make use of sketching techniques. We first define the approximate matrix multiplication (AMM) property [Woo14].

Definition 6 (Approximate matrix multiplication)**.** Let n, h and p be integer parameters and $\varepsilon > 0$ a real parameter. A randomized sketching matrix $S \in \mathbb{R}^{h^p \times r}$ has the (ε, p)-AMM property if for any two matrices $A, B \in \mathbb{R}^{n \times h}$ we have that

$$||(A^{\otimes p}S)(B^{\otimes p}S)^\intercal - A^{\otimes p}(B^{\otimes p}S)^\intercal||_F \leq \varepsilon ||(A^{\otimes p}||_F ||(B^{\otimes p}||_F$$

with probability at least $\frac{9}{10}$ over the randomized sketching matrix S.

In the above definition the Kronecker product is applied to a matrix row-wise and $||A||_F$ denotes the Frobenius norm of a matrix, that is $\sqrt{\sum_{i,j} A_{i,j}^2}$. The parameter r in the definition above is called the sketch size, it is a function of the accuracy parameter ε. We would want both the sketch size to be small and to be able to compute $A^{\otimes p}S$ fast for an arbitrary matrix A.

We use a result from sketching theory given by Ahle et. al [Ahl+20].

Theorem 3. *Let $p = 2^k$ and $\varepsilon > 0$ be parameters. There is a randomized sketching matrix S with $r = \Theta(\frac{p}{\varepsilon^2})$ columns such that S satisfies the (ε, p)-AMM property. For an arbitrary matrix $A \in R^{n \times h}$, computing $A^{\otimes p}S$ requires $\mathcal{O}(nphr + npr^2)$ time.*

They also provide a construction of the sketch S. We will first describe how to implement the sketch for $p = 2$. We sample two random matrices $G_1, G_2 \in \mathbb{R}^{h \times r}$ such that each entry is drawn independently from a standard normal distribution. Then the result of applying the sketch S to matrix A is

$$\text{Sketch}(A, r, p) = A^{\otimes 2}S = \sqrt{\frac{1}{r}}((AG_1) \star (AG_2)),$$

where by $\star$ we denote the entry-wise product. We can extend this construction to all powers of 2 via recursion. If we want to compute $\text{Sketch}(A, r, p)$ we will start by computing $M_1 = \text{Sketch}(A, r, \frac{p}{2})$ and $M_2 = \text{Sketch}(A, r, \frac{p}{2})$ and then setting

$$\text{Sketch}(A, r, p) = \sqrt{\frac{1}{r}}((M_1G_1) \star (M_2G_2)).$$

Note that here $G_1, G_2 \in \mathbb{R}^{r \times r}$ because $M_1, M_2 \in \mathbb{R}^{n \times r}$ come from computing the sketch. It is obvious that the described procedure achieves claimed complexity, since we perform one matrix multiplication of time complexity $\mathcal{O}(nhr)$, one martrix multipllication of time complexity $\mathcal{O}(nr^2)$ and one entry-wise product of time complexity $\mathcal{O}(nr)$ - $\mathcal{O}(p)$ times each.

Thus, we can approximate the matrix $\phi(Q)\phi(K)^\intercal = Q^{\otimes p}(K^{\otimes p})^\intercal$ with $(Q^{\otimes p}S)(K^{\otimes p}S)^\intercal$ to compute attention as in equation (3.6).

One issue we have is that $(Q^{\otimes p}S)(K^{\otimes p}S)^\intercal$ may contain negative values, which would not work, since we require the kernel function to be non-negative. To fix this we may notice that for two arbitrary vectors x, y we have $x^{\otimes 2}(y^{\otimes 2})^\intercal = (xy^\intercal)^{\circledcirc} \geq 0$. We thus may use an approximate feature map

$$\phi(X) = (X^{\otimes \frac{p}{2}}S)^{\otimes 2} \tag{4.1}$$

instead, at the cost of squaring the sketch size r.

Does this still represent a good sketch? Yes, the paper [KMZ24] proves the following theorem.

Theorem 4. *Let $\varepsilon, \delta > 0$ and $p = 2^k$ be constants. Let $S \in \mathbb{R}^{h^{\frac{p}{2}} \times r}$ be a randomized sketching matrix for $r = \Omega\left(\frac{p \log \frac{1}{\delta}}{\varepsilon^2}\right)$ and let ϕ be as in equation (4.1). Then for any two matrices $Q, K \in \mathbb{R}^{n \times h}$ we have*

$$||\phi(Q)(\phi(K))^\intercal - (QK^\intercal)^{\otimes p}||_F \leq \varepsilon ||Q^{\otimes p}||_F ||K^{\otimes p}||_F$$

with probability at least $1 - \delta$ over the randomized sketching matrix S.

Authors of the paper [KMZ24] call this mechanism for computing attention 'Polysketch Attention'.

Note that since we can compute $A^{\oplus \frac{p}{2}} S$ in $\mathcal{O}(nphr + npr^2)$ time by the construction for Theorem 3, we can computer $\phi(X)$ in $\mathcal{O}(nphr + npr^2)$ time since we just need to bring n vectors of dimension r to second Kronecker power which requires $\mathcal{O}(nr^2)$ time.

Remark. It is worth noting that instead of generating random matrices G_1, G_2 during computation of the sketch we may instead try to replace them with learnable parameters. The authors of the paper [KMZ24] claim that using a non-linear transformation introduced by a dense neural network (a network similar to the one described in subsection 1.1.6) with size comparable to size of matrices G_1, G_2 leads to a model with better quality.

4.3 More general polynomials

In previous sections we've only considered polynomials with a single term. In this section we will look at a more general class of polynomials. Due to the necessity of non-negativity of kernel function we cannot just use an arbitrary polynomial. On the other hand, we can use any polynomial that produces only non-negative values. We will consider a class of all polynomials that have the form of $(poly(n))^2$. Obviously, they only produce non-negative values. We thus will consider a functions of the form

$$f(x, y) = \left(\sum_{i=0}^{p} a_i (xy^\intercal)^i \right)^2$$

for arbitrary parameters $p \in \mathbb{Z}^+$ and $a_0, \ldots, a_p \in \mathbb{R}$. It may not always be a kernel function, or, at least, have a non-trivial feature representation but we will show how it is still possible to use it for attention. We can first expand the square to get

$$f(x, y) = \sum_{i=0}^{2p} b_i (xy^\intercal)^i$$

where $b_0, \ldots, b_{2p}$ are not arbitrary anymore, and depend on $a_0, \ldots, a_p$. We can now make use of Proposition 1 to get

$$f(x, y) = \sum_{i=0}^{2p} b_i x^{\otimes i} \left(y^{\otimes i} \right)^\intercal.$$

We now modify equation (3.3) with $sim = f$ to get

$$\begin{aligned}\text{Attention}(\mathrm{Q},\mathrm{K},\mathrm{V})_\mathrm{i} &= \frac{\sum_{j=1}^{l_z} f(Q_i, K_j)V_j}{\sum_{j=1}^{l_z} f(Q_i, K_j)} \\ &= \frac{\sum_{j=1}^{l_z}\sum_{r=0}^{2p} b_r Q_i^{\otimes r}\left(K_j^{\otimes r}\right)^\intercal V_j}{\sum_{j=1}^{l_z}\sum_{r=0}^{2p} b_r Q_i^{\otimes r}\left(K_j^{\otimes r}\right)^\intercal} \\ &= \frac{\sum_{r=0}^{2p} b_r Q_i^{\otimes r}\left(\sum_{j=1}^{l_z}\left(K_j^{\otimes r}\right)^\intercal V_j\right)}{\sum_{r=0}^{2p} b_r Q_i^{\otimes r}\left(\sum_{j=1}^{l_z}\left(K_j^{\otimes r}\right)^\intercal\right)}.\end{aligned}$$

We see that for a fixed r we obtain the same expressions in numerator and denominator (considered separately) as for attention with a kernel function, thus we can use the same methods. That is, we can calculate the inner sums once and reuse them for each query. The complexity of this is dominated by computation for the largest power (due to properties of geometric series) and thus is the same $\mathcal{O}(nd_{attn}^{p+1})$ as if we were computing for just a single term. Moreover, it is linear in n.

We may try to either hardcode such a polynomial or try to learn it. That means we can fix the power p and then let the model learn best coefficients $a_0, a_1, \ldots, a_p$. This was suggested to me by Dr. Timothy Chu [Chu24].

Remark. Note that implementing unidirectional attention can again be done in the same way as in subsection 3.3.

5 Experiments

In this chapter we will test accuracy of a an encoder-only transformer with various attention variants described in previous chapters. We will focus only on accuracy because properly measuring speed isn't possible due to standard libraries not having optimizations for necessary functions.

We use Python with PyTorch library [Pas+17] to implement the model. During implementation we used tutorials [Lip; Kar] and official documentation [Fou]. We used the Matplotlib library [Hun07] to produce plots.

The code we used during experiments can be found in attachments to this thesis. We implement an encoder-only transformer described in subsection 1.2.1, except we use different variants of attention. We implemented the non-standard variants of attention in quadratic time because the library isn't optimized for operations necessary to perform attention in linear time as described in subsection 3.1. Thus, using linear implementations is very impractical. Models using linear implementation of attention take more than 20 times the time it takes for models with quadratic implementation.

During training we use the AdamW optimizer [LH19], which uses gradient descent, and is known to perform better for transformers than the stochastic gradient descent.

The task on which we test our models is reversing an integer sequence. Obviously, this task can be trivially solved without any usage of neural networks but it is not such an easy task for neural networks. Still, it is not extremely complicated and thus would allows us to use a model of a small size to be able to train it in relatively short time. We use a single encoder layer and a single attention head. We use an embedding dimension of 32. We use batch size of 128. Batch size is just the number of requests to the model that we do at the same time, to make use of parallelism. The sequences we use consist of integers from 0 to 9 (inclusive) and have length 50.

To measure accuracy of a model we use a loss function. The smaller values it outputs, the better a model is. We use cross entropy loss function, which is standard. If p is a probability distribution produced by a model and q is the true probability distribution (which, in our case, is just a probability 1 for some element and 0 for all other) then the cross entropy loss is defined as

$$-\sum_{c=1}^{C} \log\left(\frac{e^{x_c} y_c}{\sum_{i=1}^{C} e^{x_i}}\right)$$

where C is the number of tokens. Loss for multiple batches is simple the mean of their respective losses.

We train each model for 10000 iterations, with calculation of average loss of 10 samples each 100 iterations. Additionally, during the first 1000 iterations we calculate average loss each 10 iterations. Each sequence will be randomly generated.

To not clutter the legends, we will write polynomial kernels as polynomials in variable x instead of $xy^\intercal$ in our plots.

For each group we plot two plots:

1. Based on first 1000 iterations with linear scale of y-axis.

2. Based on last 9000 iterations with logarithmic scale of y-axis.

The implementation may be found in attachments to this thesis in the file **main.py**.

5.1 Single term polynomials of even degree

In this section, we test polynomials of even degree from Section 4.1, in particular, we test attention with the following kernel functions:

1. $k(x, y) = (xy^\intercal)^2$;
2. $k(x, y) = (xy^\intercal)^4$;
3. $k(x, y) = (xy^\intercal)^8$.

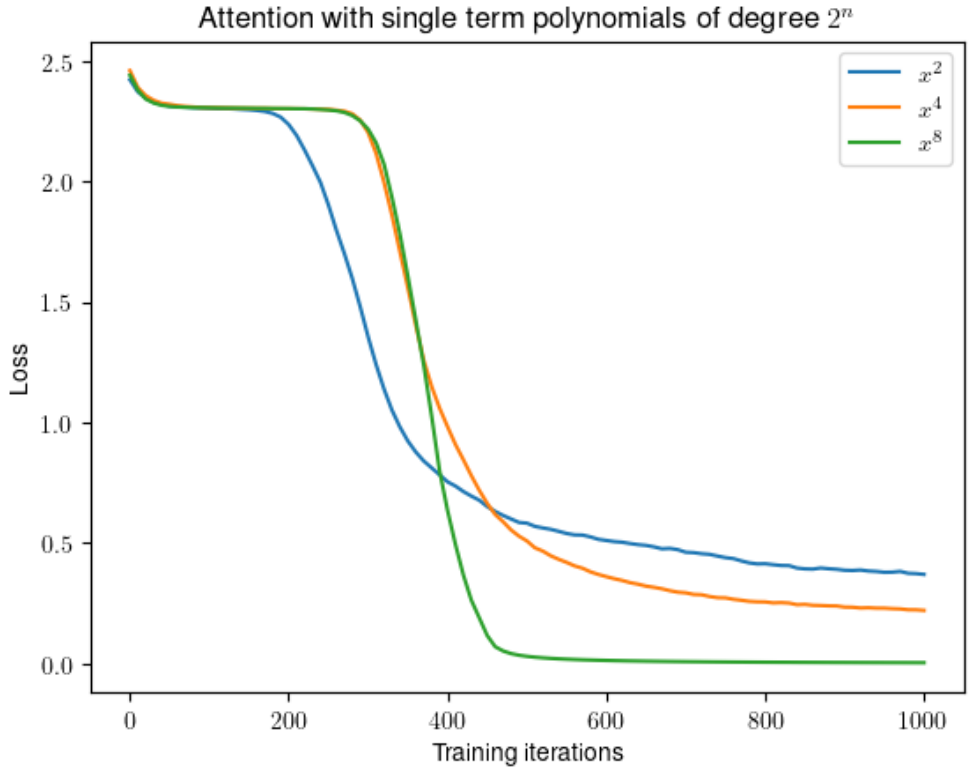

Figure 5.1 First 1000 training iterations with points every 10 iterations

In Figure 5.1 we can see that polynomials with higher degree perform better than those with lower degree. Interestingly, the polynomial with the lowest degree x^2 achieves better loss until around 400-th iteration, where it is overtaken by x^8, and, a little later, by x^4 as well. We can see that x^8 reaches very small values of loss relative to the other two polynomials around 600-th iteration, while both x^2 and x^4 continue decreasing throughout the entire training process at a similar rate, as can been seen from Figure 5.2.

The final values of losses the models achieve after 10000 training steps are approximately $0.19621022, 0.01990822$ and 0.00000408, with x^8 having an extremely better result than the other two polynomials.

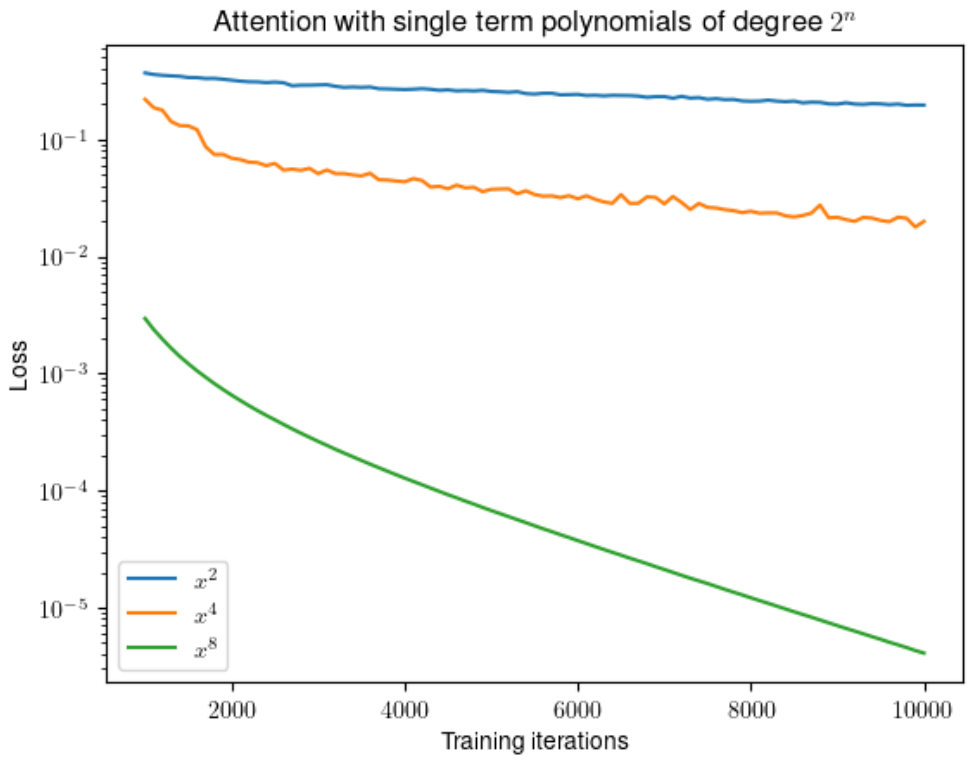

Figure 5.2 Last 9000 training iterations with points every 100 iterations

5.2 Squares of general polynomials

In this section, we test squares of general polynomials from Section 4.3. By trying different polynomials by hand we noted that polynomials with all coefficients being equal to 1 seem to have a consistently better performance than other polynomials. Thus, we will test such polynomials of different degrees. In particular we will test attention with all polynomials of form $(1 + x + \cdots + x^d)^2$ for d from 1 to 6 (inclusive).

In the Figure 5.3 we can see that here it is not the case that polynomials of higher degree perform better. In first 1000 iterations, all polynomials except $(1 + x)^2$ have similar learning curves. The only exception, $(1 + x)^2$, almost immediately gets stuck seeing tiny to no improvement. This trend continues on for the next 9000 iterations, with $(1 + x)^2$ never reaching below 2.2. Thus, we exclude it from the Figure 5.4 to be able to see plots for other polynomials much better. We also exclude it from further discussion.

In the Figure 5.4 we see that polynomials of degree 5 and 3 perform significantly better than others. The degree 5 polynomial is consistently improving, while the degree 3 polynomial is jumping up and down, but still improves overall. The jumps don't reach neither degree 5 polynomial nor the other polynomials. Polynomials of degrees 2, 4 and 6 all perform on a similar level, but significantly worse the polynomials of odd degree (excluding 1).

From the Figure 5.4 we notice that polynomials of odd degree perform significantly better than polynomials of even degree. We also notice that among polynomials with degree of same parity polynomials of higher degree perform better.

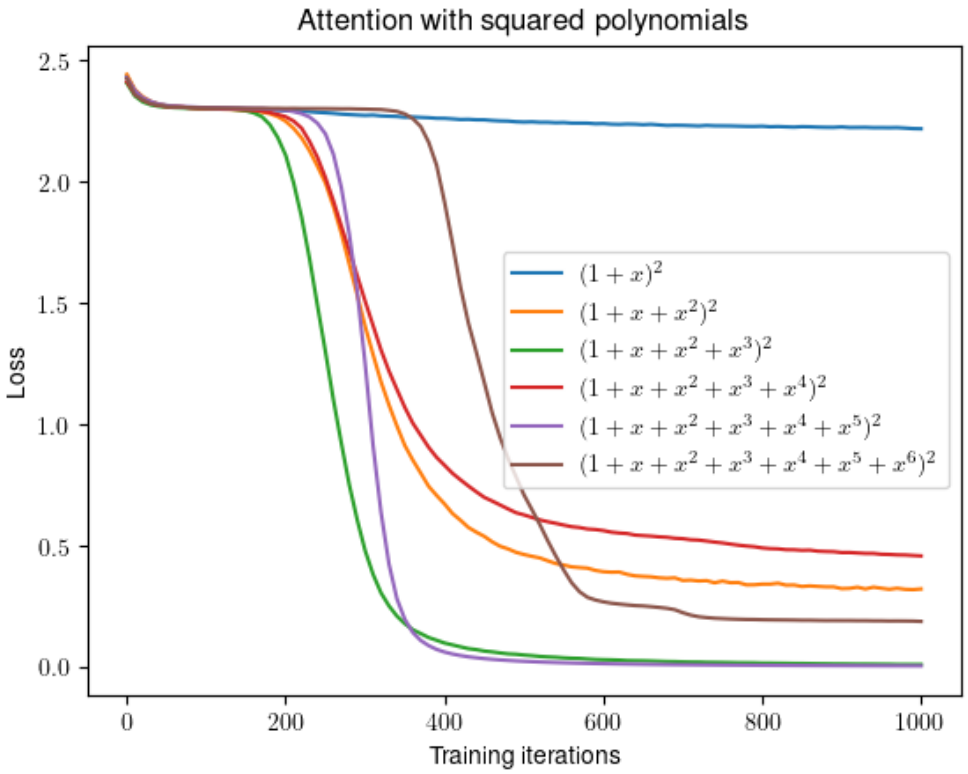

Figure 5.3 First 1000 training iterations with points every 10 iterations

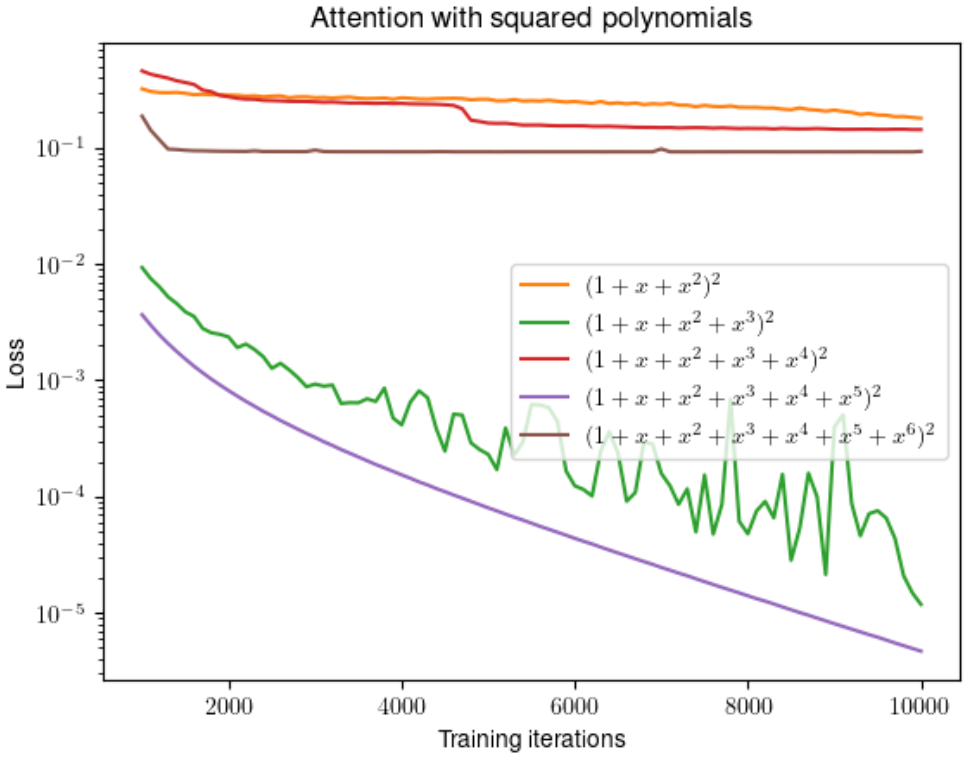

Figure 5.4 Last 9000 training iterations with points every 100 iterations

The final values of losses the models achieve after 10000 training steps are approximately $2.20814011, 0.17952636, 0.00001183, 0.14318735, 0.00000469$ and

0.09295710 for the degree from 1 to 6, respectively.

5.3 Learning squares of general polynomials

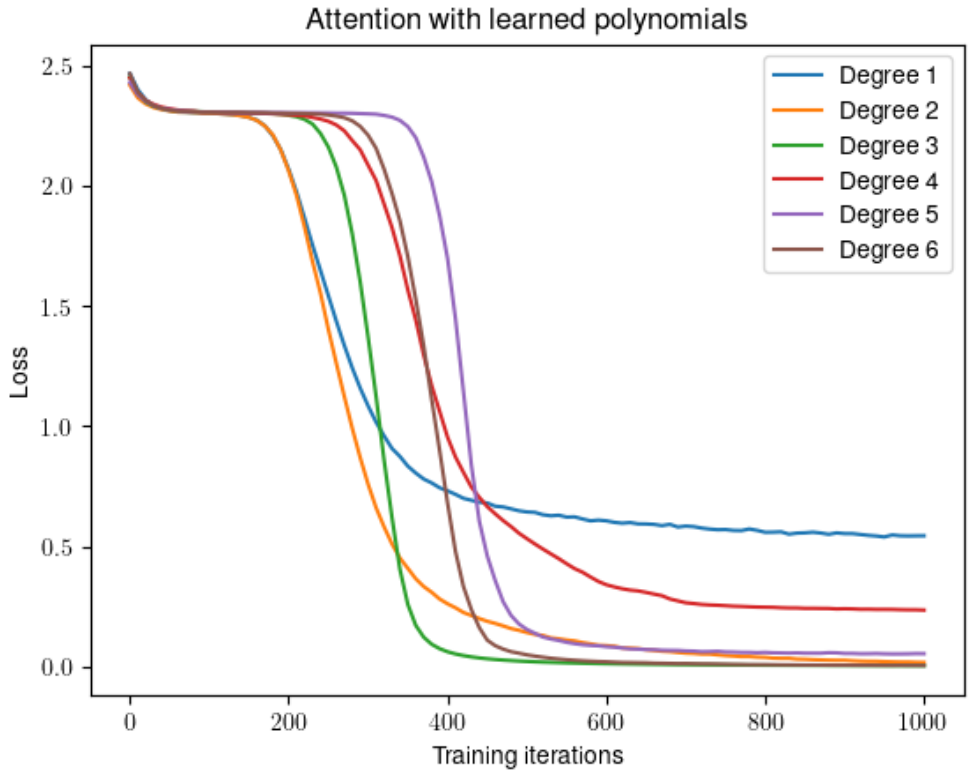

Figure 5.5 First 1000 training iterations with points every 10 iterations

In this section, we try learning a polynomial as suggested in the end of Section 4.3. We will initialize polynomials by sampling each coefficient from a $\mathcal{U}(-50, 50)$ (uniform real distribution) independently. We've tried sampling from other uniform distribution, such as $\mathcal{U}(-1, 1)$ or $\mathcal{U}(-500, 500)$ but they seemed to perform significantly worse.

As can be seen in the Figure 5.5, all of the models have a very steep drop at around 400 iterations, with much less steep improvement after that. From the Figure 5.6 we see that polynomials of degree 1 and 4 don't improve much. Polynomials of degree 2 and 3 are a very different story, with both seeing continuous improvement until the very end. Polynomials of degree 5 and 6 have a more curious behaviour. Degree 6 polynomial decrease similarly to polynomials of degree 2 and 3 but at some points has high spikes, with decrease in performance, quickly returning to better performances. Degree 5 polynomial stagnates until around 7000 iterations, much like polynomials of degree 1 and 4, but then very steeply drops to the level of polynomial of degree 6.

We see that here there is no obvious relationship between degree of polynomial and performance. In general, though, this group has the best performance out of 3 groups we have considered in first 3 sections of this chapter.

The final values of losses the models achieve after 10000 training steps are approximately 0.21729853, 0.00000564, 0.00000451, 0.18778882, 0.00001671, and

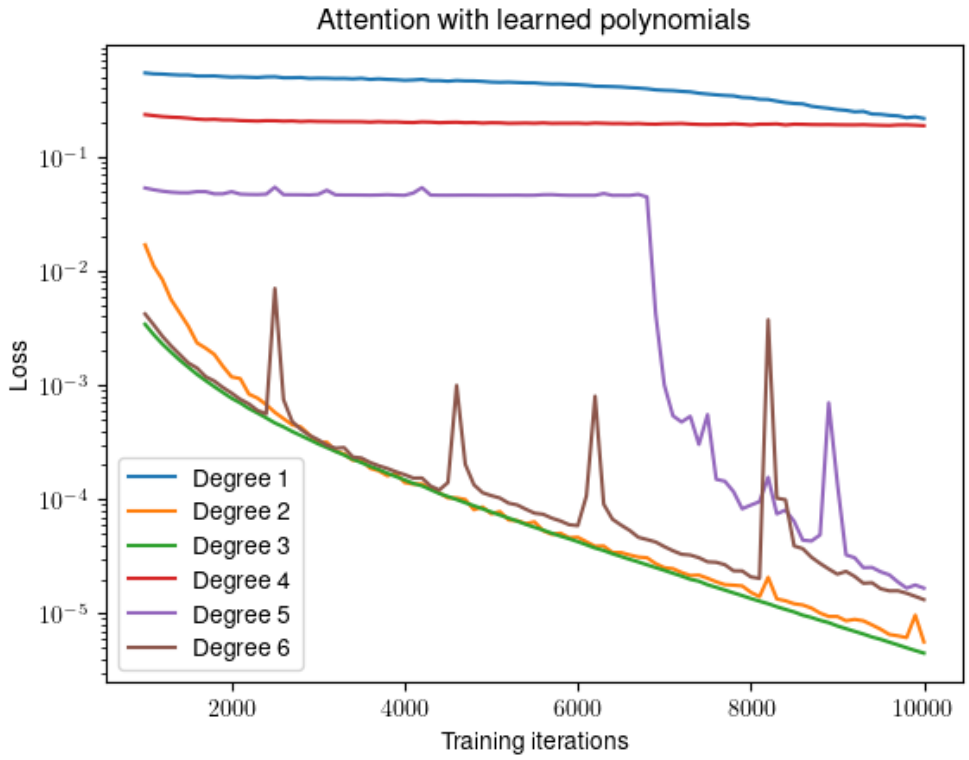

Figure 5.6 Last 9000 training iterations with points every 100 iterations

0.00001326 for degrees 1 to 6 respectively. Learned coefficients of polynomials may be seen in attachment **learned_polynomials.txt**. Coefficient of a polynomial are ordered from the coefficient by the lowest exponent to the one by the highest.

5.4 Comparing best from different groups

In this section we will compare best performers from the first 3 sections and also 3 new models:

1. Softmax attention, as a benchmark.
2. Attention using entry-wise ELU as the feature map from the section 3.2
3. Taking the best learned polynomial (the degree 3 polynomial from the previous section) and using it as a polynomial. This way the model will be able to learn other parameters without need to improve kernel function of attention. We will call this a fixed learned polynomial.

As best representatives of each of the three previous sections we will chose x^8, $1 + x + x^2 + x^3 + x^4 + x^5$ and the polynomial of degree 3 respectively from Section 5.3.

In the Figure 5.7 we see a similar learning curve for most of the functions, starting with initial stagnation and then a very steep drop at around 400 iterations. The only outlier is ELU, it also stagnates up to around 400 and after that start improving but the improvement is much less steep. From the Figure 5.8 we can

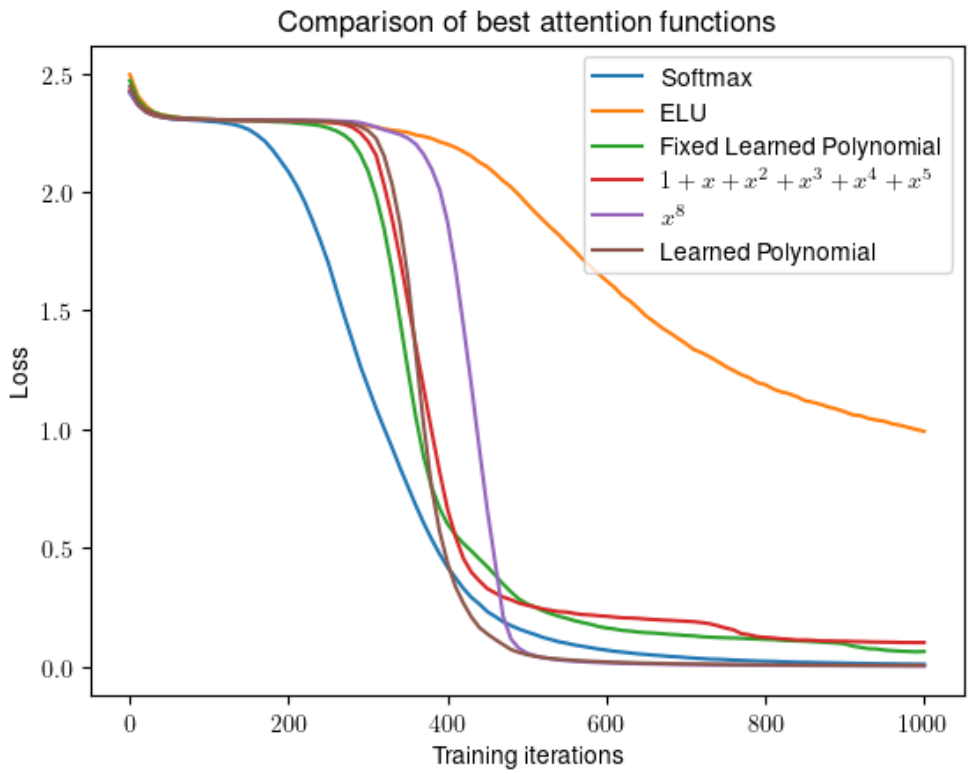

Figure 5.7 First 1000 training iterations with points every 10 iterations

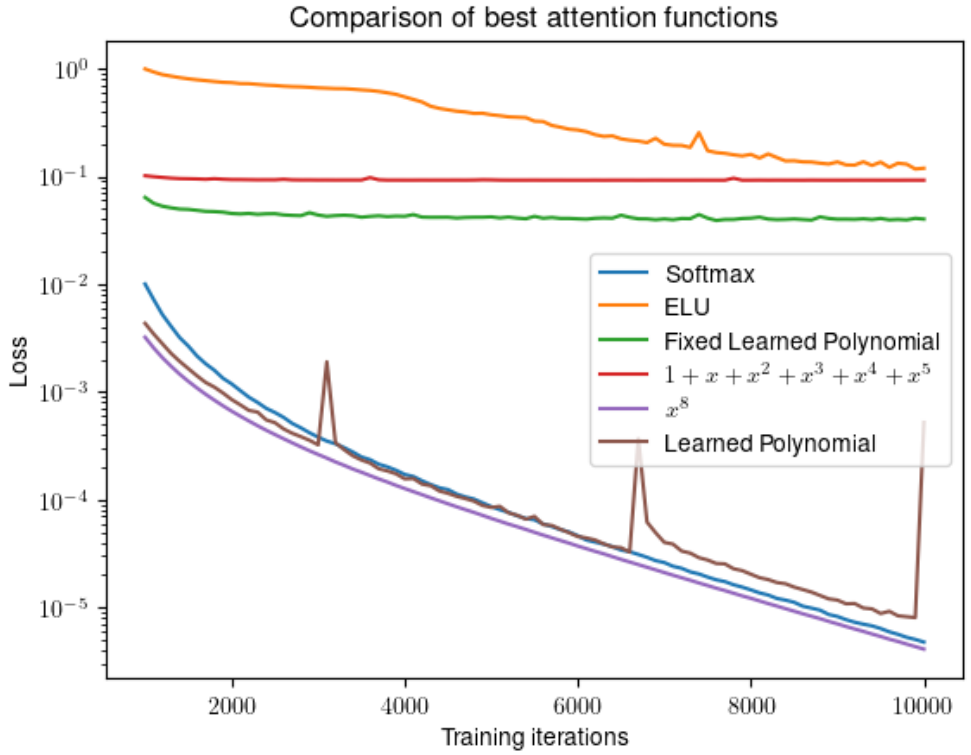

Figure 5.8 Last 9000 training iterations with points every 100 iterations

see that this trend for ELU continues in the next 9000 iterations. It has a slow but consistent improvement.

Fixed learned polynomial and $1+x+x^2+x^3+x^4+x^5$, on the other hand, don't see any improvement over last 9000 iterations, and end up with a much worse result then they had in previous sections. From our tests we noticed that all of these functions except softmax and ELU are highly susceptible to initial conditions, sometimes performing very bad and sometimes very well. This suggests that they are unstable and it is not clear if this is an inherent flaw of these attention models.

Plots suggest that in the best scenario one of the alternative attention functions manages to beat softmax, as in this case x^8 ends up with a smaller loss. But it is not clear how to reach those cases.

Finally, learned polynomial has a weird graph, with occasional spikes increasing loss by a lot in a span of few iterations and then decreasing back. It is not clear why this happens.

The final values of losses the models achieve after 10000 training steps are approximately 0.00000477, 0.11890406, 0.04012463, 0.09217419, 0.00000410, and 0.00052050 for the functions in the same order as in the legend of Figure 5.7 (top to bottom).

Conclusion

In this thesis we have explored the inner working of transformers. In particular, we have focused on the attention mechanism. We have looked at alternatives of softmax attention with potential to be much more efficient. All of them run in linear time with respect to context length, which is better than the theoretical limit of quadratic time for softmax attention, as we've seen in Chapter 2 (assuming SETH).

We've implemented an encoder-only transformer and used it to conduct multiple experiments, comparing softmax attention with its alternatives. We've found some alternatives that, on our dataset, performed on par with softmax attention. For example, the square of polynomial with all coefficients equal to 1 of degree 3 (albeit only in the first experiment). The function x^8 managed to beat softmax in the final experiment, and this function can be approximated very fast with polysketch.

Further directions of research may include further investigations of approximation of polynomial attention, so that it becomes possible to approximate fast attention defined by square of general polynomial, since those seem to perform much better than single-term polynomials with degree equal to a power of 2. Another direction of research may be handcrafting polynomials that perform better, and looking into theoretical reasons for their superiority. This is motivated by our finding that a square of polynomial with all coefficients equal to 1 seems to perform much better than other polynomials. Finally, another direction of development is creation of optimized libraries that can support operations necessary for implementation of polynomial attention.

Bibliography

[Ahl+20] Thomas D. Ahle et al.
"Oblivious Sketching of High-Degree Polynomial Kernels".
In: *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*. Ed. by Shuchi Chawla. SIAM, 2020, pp. 141–160.
DOI: 10.1137/1.9781611975994.9.
URL: https://doi.org/10.1137/1.9781611975994.9.

[AS23] Josh Alman and Zhao Song.
"Fast Attention Requires Bounded Entries".
In: *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.
Ed. by Alice Oh et al. 2023.
URL: http://papers.nips.cc/paper%5C_files/paper/2023/hash/c72861451d6fa9dfa64831102b9bb71a-Abstract-Conference.html.

[BKH16] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton.
"Layer Normalization". In: *CoRR* abs/1607.06450 (2016).
arXiv: 1607.06450. URL: http://arxiv.org/abs/1607.06450.

[Chu24] Timothy Chu. Personal communication. Apr. 2024.

[CUH16] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter.
"Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.
Ed. by Yoshua Bengio and Yann LeCun. 2016.
URL: http://arxiv.org/abs/1511.07289.

[Dev+19] Jacob Devlin et al. "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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*.
Ed. by Jill Burstein, Christy Doran, and Thamar Solorio.
Association for Computational Linguistics, 2019, pp. 4171–4186.
DOI: 10.18653/V1/N19-1423.
URL: https://doi.org/10.18653/v1/n19-1423.

[Fou] The Pytorch Foundation. *PyTorch documentation*.
URL: https://pytorch.org/docs/stable/index.html (visited on 05/09/2024).

[Fuk69] Kunihiko Fukushima. “Visual Feature Extraction by a Multilayered Network of Analog Threshold Elements”.
In: *IEEE Trans. Syst. Sci. Cybern.* 5.4 (1969), pp. 322–333.
DOI: 10.1109/TSSC.1969.300225.
URL: https://doi.org/10.1109/TSSC.1969.300225.

[He+15] Kaiming He et al. “Deep Residual Learning for Image Recognition”.
In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385.
URL: http://arxiv.org/abs/1512.03385.

[Hun07] J. D. Hunter. “Matplotlib: A 2D graphics environment”.
In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95.
DOI: 10.1109/MCSE.2007.55.

[IP01] Russell Impagliazzo and Ramamohan Paturi.
“On the Complexity of k-SAT”. In: *Journal of Computer and System Sciences* 62.2 (2001), pp. 367–375. ISSN: 0022-0000.
DOI: https://doi.org/10.1006/jcss.2000.1727.
URL: https://www.sciencedirect.com/science/article/pii/S0022000000917276.

[Kar] Andrej Karpathy. *NanoGPT video lecture*. URL:
https://github.com/karpathy/ng-video-lecture/tree/master (visited on 05/09/2024).

[Kat+20] Angelos Katharopoulos et al. “Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention”.
In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 5156–5165. URL:
http://proceedings.mlr.press/v119/katharopoulos20a.html.

[KMZ24] Praneeth Kacham, Vahab Mirrokni, and Peilin Zhong.
PolySketchFormer: Fast Transformers via Sketching Polynomial Kernels. 2024. arXiv: 2310.01655 [cs.LG].

[LH19] Ilya Loshchilov and Frank Hutter.
“Decoupled Weight Decay Regularization”.
In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.
URL: https://openreview.net/forum?id=Bkg6RiCqY7.

[Lip] Phillip Lippe. *Tutorial 6: Transformers and Multi-Head Attention*.
URL: https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html (visited on 05/09/2024).

[Mik+13] Tomás Mikolov et al.
“Efficient Estimation of Word Representations in Vector Space”.
In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2013.
URL: http://arxiv.org/abs/1301.3781.

[Pas+17] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: *NIPS-W*. 2017.

[PH22] Mary Phuong and Marcus Hutter. “Formal Algorithms for Transformers”. In: *CoRR* abs/2207.09238 (2022). DOI: 10.48550/ARXIV.2207.09238. arXiv: 2207.09238. URL: https://doi.org/10.48550/arXiv.2207.09238.

[Rad+19] Alec Radford et al. “Language Models are Unsupervised Multitask Learners”. In: 2019. URL: https://api.semanticscholar.org/CorpusID:160025533.

[Rub18] Aviad Rubinstein. “Hardness of approximate nearest neighbor search”. In: *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*. Ed. by Ilias Diakonikolas, David Kempe, and Monika Henzinger. ACM, 2018, pp. 1260–1268. DOI: 10.1145/3188745.3188916. URL: https://doi.org/10.1145/3188745.3188916.

[Sha78] M.I. Shamos. *Computational Geometry*. THESIS (Ph.D) - YALE UNIVERSITY, 1978. Yale University, 1978. URL: https://books.google.cz/books?id=Mf50SQAACAAJ.

[Vas+17] Ashish Vaswani et al. “Attention Is All You Need”. In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[Wil19] Virginia Vassilevska Williams. “ON SOME FINE-GRAINED QUESTIONS IN ALGORITHMS AND COMPLEXITY”. In: *Proceedings of the International Congress of Mathematicians (ICM 2018)* (2019). URL: https://api.semanticscholar.org/CorpusID:19282287.

[Woo14] David P. Woodruff. “Sketching as a Tool for Numerical Linear Algebra”. In: *Found. Trends Theor. Comput. Sci.* 10.1-2 (2014), pp. 1–157. DOI: 10.1561/0400000060. URL: https://doi.org/10.1561/0400000060.