MathLearn: Attention and transformers

Joshua Maglione

June 3, 2026

Contents

1 Introduction

"*I learned very early the difference between knowing the name of something and knowing something.*" Richard Feynman

"*But before the road of Excellence the immortal gods have placed sweat. And the way to it is long and steep and rough at first. But when one arrives at the summit, then it is easy, even though remaining difficult.*" Also known as: "*No pain, no gain.*" Hesiod

These are lecture notes for the MathLearn workshop held at the University of Galway in June 2026. The purpose is to provide early-career researchers in pure mathematics with a working understanding of modern artificial intelligence (AI) tools. This is not meant to be exhaustive—topics not covered may still be relevant. These notes specifically are focused on the attention mechanism, transformers, and the PatternBoost method. I have included some extra, related topics that I find interesting; these section are distinguished from the rest with an asterisk.

Recall from previous lectures the multi-layer perceptron (MLP), which is a (fully connected) feedforward neural network (NN). Here, we will learn about *transformers*, an architecture that uses *attention* together with MLPs. Transformers were first introduced in 2017 by researchers at Google [21], and have since become one of the foundational architectures of modern AI. Transformers enable highly parallelizable training and more effective modelling of long-range dependencies, helping to drive major advances in natural language processing, large language models, computer vision, speech, code generation, and scientific machine learning.

2 Natural language processing

We will not cover the whole history of course, but it is worth noting what problem transformers solved in 2017 as additional motivation. Let us consider the problem of natural language processing (NLP). Suppose we want to translate a sentence from one language to another. (Later we will replace language with mathematical objects. Hang in there!)

We know that MLPs can be used to approximate functions $\mathbb{R}^m \to \mathbb{R}^n$, and they are fixed in size. Maybe we have a way to map a sentence into some $\mathbb{R}^m$ by padding or truncating it. It is also unclear what the output dimension n should be: translated sentences need not have the same length as the original, and there is no obvious fixed choice of n. Many more aspects of language are not captured by a plain MLP:

- sentences are variable-length sequences rather than fixed-size vectors;
- the order of symbols matters;
- both local and long-range dependencies are important;
- the meaning of a word or symbol depends strongly on context.

It is cumbersome to adapt a plain MLP to process language; in principle, it can be done, but this is not practical. Instead different architectures were proposed, and one such family is the recurrent neural network (RNN), which attempted to address the contextual and sequential problems that a plain MLP cannot. Before transformers, sequence modelling was dominated by RNNs, including bidirectional RNNs [17] and long short-term memory networks [7]. These architectures addressed sequential and contextual structure more naturally than plain MLPs, but they are harder to parallelize during training [21] and often struggle with long-range dependencies [1].

2.1 Tokenization

Maybe we map English and Japanese characters to elements in $\mathbb{R}$, and maybe we allow only a fixed number of characters, say, $m = 140$. This first-approach could work, but notice there's an inherent ordering: if

$$a \mapsto 1, \qquad b \mapsto 2, \qquad \text{etc.},$$

then a is closer to b than to c, but there is no reason for this. Instead we could map each character to a standard basis vector such as

$$a \mapsto e_1, \qquad b \mapsto e_2, \qquad \text{etc.},$$

Remark 2.1. "Standard basis vector" is common usage in mathematics for a vector of zeros except for exactly one entry equal to one. In machine learning, these are called "one-hot vectors". We will switch over now.

Let's work a bit more formally. Suppose Σ is a finite alphabet, for example, the set of ASCII characters, and let V be a finite set, called the ***vocabulary***. Elements of V are called ***token types***. Let Σ^* and V^* denote the sets of finite sequences with entries in Σ and V, respectively. An element of Σ^* is a ***token string***, and a term of a sequence in V^* is called a ***token instance***. A function $T : \Sigma^* \to V^*$ is called a ***tokenizer***. Often Σ is the set of characters of a language and V is identified with a finite set of indices $\{0, 1, \dots N-1\}$ for some $N \in \mathbb{N}_0$; in this encoding, the elements of V are called ***token IDs***. Note that the tokenizer does not output real vectors: it produces a finite sequence of symbols or indices. These are then mapped into a real vector space as we will see in Section 2.2.

Example 2.2. Suppose Σ is the set of Unicode characters and $V = \{0, \dots, 4\}$. Let $x =$ "naive". Different tokenizers might segment x in different ways, for example

$$(\mathrm{n}, \mathrm{a}, \mathrm{i}, \mathrm{v}, \mathrm{e}) \mapsto (0, 1, 2, 3, 4), \qquad (\mathrm{naive}) \mapsto (1), \qquad (\mathrm{nai}, \mathrm{ve}) \mapsto (2, 4)$$

We note that a lack of injectivity may be desirable: for example, one might choose a tokenizer that treats "naive" and "naïve" identically.

Remark 2.3. It might also be desirable to drop the function condition of the tokenizer and make it stochastic. In Example 2.2, one could instead segment the string completely down to characters and then merge from left to right, say. Each merge could be done with probability p (and not merging with probability $1 - p$). This adds some robustness to the training.

Remark 2.4. The word "token" is ambiguous without context, and this is often a feature not a bug.

2.2 Embeddings and positional information

After tokenization, each token is mapped to a vector. A function $\phi : V \to \mathbb{R}^d$ is called the ***embedding map***, and this is learned during training. Note that d is fixed and is a hyperparameter of the specific transformer model. The idea of learning continuous vector representations of words predates the transformer and was already central in earlier NLP models such as word2vec [13].

To me, it is mathematically nicer to think of ϕ as the composition of two functions although in practice one uses a lookup rather than explicit multiplication. Suppose $|V| = N$, and let $i : V \to \{1, \dots, N\}$ be some bijection fixed throughout. Thus, ϕ is the composition of the one-hot encoding $V \to \mathbb{R}^N$ via $t \mapsto e_{i(t)}$ with a linear map $W_E : \mathbb{R}^N \to \mathbb{R}^d$, also written $W_E \in \mathbb{R}^{N \times d}$, such that

$$\phi(t) = e_{i(t)} W_E.$$

(Note that the default for me is a row vector.) Each of the entries in W_E is learned during training.

Remark 2.5. It is understood that ϕ is injective, otherwise there would be no point in distinguishing some of the tokens. This however does not imply that the linear map W_E is injective; only the weaker condition should hold: W_E has distinct rows. Usually $d < N$, which precludes injectivity.

Although the tokenizer produces sequences, there is no recurrence or convolution, so we include positional information to make use of the order of the sequence. In the original transformer paper ([21]), positional information was added to the input embeddings via fixed sinusoidal functions of different frequencies. Modern large language models often use different positional mechanisms—most commonly rotary positional embeddings (RoPE) [20]—which encode position directly in the attention computation. For simplicity, we follow the original approach in [21].

Suppose we have a tokenized sequence $(t_1, t_2, \dots, t_n) \in V^*$. We construct n ***positional vectors*** $p_1, \dots, p_n \in \mathbb{R}^d$ as follows: for each $i \in \{1, \dots, n\}$ and each $j \in \{1, \dots, d\}$, define the jth entry of p_i as

$$p_{ij} = \begin{cases} \cos\left(i \cdot 10^{-4(j-2)/d}\right) & \text{if } j \text{ is even,} \\ \sin\left(i \cdot 10^{-4(j-1)/d}\right) & \text{if } j \text{ is odd.} \end{cases}$$

(We usually take d to be even so that the entries of the positional vectors come in sine-cosine pairs of the same frequency.) For example, if $n = 6$ and $d = 8$, we could put the six positional vectors as rows of a 6×8 matrix P as follows (with

only four decimal places showing):

$$P = \begin{pmatrix} 0.8415 & 0.5403 & 0.0998 & 0.9950 & 0.0100 & 1.0000 & 0.0010 & 1.0000 \\ 0.9093 & -0.4161 & 0.1987 & 0.9801 & 0.0200 & 0.9998 & 0.0020 & 1.0000 \\ 0.1411 & -0.9900 & 0.2955 & 0.9553 & 0.0300 & 0.9996 & 0.0030 & 1.0000 \\ -0.7568 & -0.6536 & 0.3894 & 0.9211 & 0.0400 & 0.9992 & 0.0040 & 1.0000 \\ -0.9589 & 0.2837 & 0.4794 & 0.8776 & 0.0500 & 0.9988 & 0.0050 & 1.0000 \\ -0.2794 & 0.9602 & 0.5646 & 0.8253 & 0.0600 & 0.9982 & 0.0060 & 1.0000 \end{pmatrix}.$$

The band structure is easier to appreciate at a larger scale; see Figure 2.1. The columns on the right vary slowly with i, while those on the left oscillate rapidly: position is encoded across a spectrum of frequencies. This structure is what makes the next exercise possible.

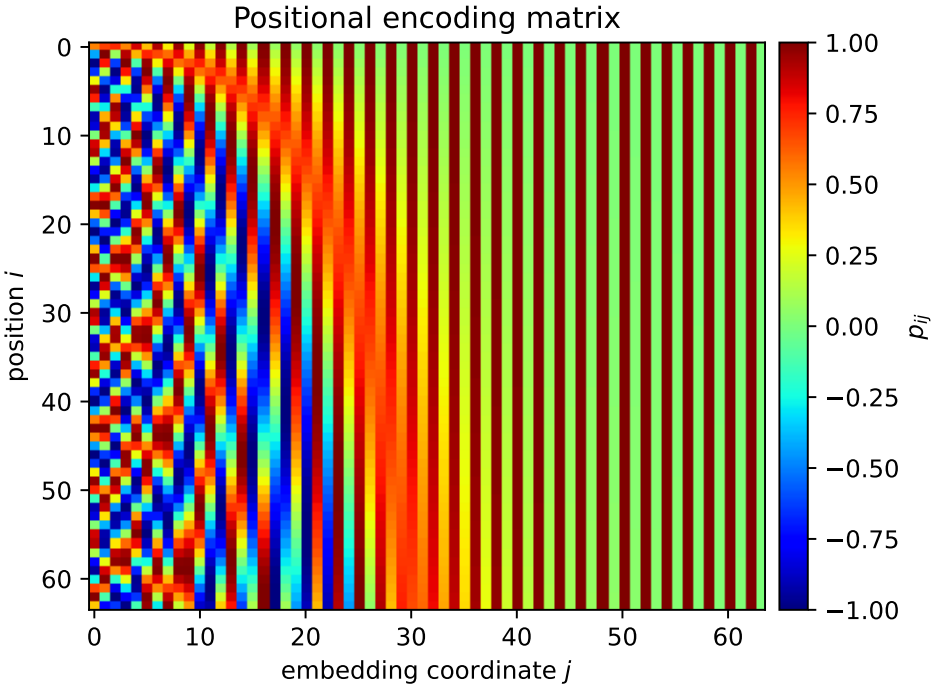


Figure 2.1: The positional encoding matrix P for $n = 64$ and $d = 64$

Exercise 1. Show that for each k, there exists $R_k \in \mathbb{R}^{d\times d}$ such that $p_{i+k} = p_i R_k$ for all i.

Now we sum these two vectors together to get the input into the transformer: for each $i \in \{1, \ldots, n\}$ we define

$$u_i^{(0)} = \phi(t_i) + p_i \in \mathbb{R}^d.$$

The role of self-attention is then to update each $u_i^{(0)}$ using information from the other positions in the sequence.

2.3 Self-attention

Self-attention is the most important part of a transformer—it is the single component that separates it from other architectures. We intuitively know that the meaning of words (and thus tokens) depends on context. For example:

- We fish for fish.
- When Alice met Bob, he tripped and fell, and she chuckled.
- The apprentice closely watches the watchmaker close the watches.

Currently the representations $u_1^{(0)}, \dots, u_n^{(0)}$ miss much of the subtlety of these sentences. In context, the meanings are clarified. Self-attention is a mechanism that addresses this problem: it captures both short-range and long-range dependencies efficiently in parallel. (Older architectures could capture the dependencies in principle, but they generally lack the efficiency of transformers; see [21, Section 4].)

Given a sequence of vectors, say, $u_1, \dots, u_n \in \mathbb{R}^d$, self-attention constructs new vectors $z_1, \dots, z_n \in \mathbb{R}^{d_v}$ for some $d_v \in \mathbb{N}$. To postpone discussion of the hyperparameter, we assert $d_v = d$. For each $i \in \{1, \dots, n\}$, we write

$$z_i = \sum_{j=1}^{n} \alpha_{ij} v_j, \tag{2.1}$$

where $v_j \in \mathbb{R}^{d_v}$ and $\alpha_{ij} \in \mathbb{R}$. Let's first understand *what* is going on in (2.1) before understanding how we build the pieces. Ideally, we want

- α_{ij} to score how much t_i should pay attention to t_j and
- v_j to tell us information about t_j.

By score we mean that the larger the value α_{ij}, the more attention should be paid; and conversely the smaller the value, the less attention should be paid. Thus, z_i is a more informed representation of u_i, incorporating information from $u_1, \dots, u_n$. The α_{ij} tell us which positions matter, and the v_j determine what those positions contribute.

Remark 2.6. Already we can start to see a difference between a plain MLP and how self-attention works. The weights and biases in each of the layers of an MLP update the input's representation—and this is the same for *all* inputs. With self-attention, the update process is dependent on the input.

Now we need to determine how we can build such objects α_{ij} and v_j. Let $U \in \mathbb{R}^{n \times d}$ be the matrix whose ith row is u_i. We construct three learned linear maps

$$W_Q \in \mathbb{R}^{d \times d_k}, \qquad W_K \in \mathbb{R}^{d \times d_k}, \qquad W_V \in \mathbb{R}^{d \times d_v}. \tag{2.2}$$

Again, $d_k \in \mathbb{N}$ is some hyperparameter; we assert $d_k = d$ for now. From the three linear maps in (2.2), we get three matrices:

$$Q = UW_Q \in \mathbb{R}^{n \times d_k}, \qquad K = UW_K \in \mathbb{R}^{n \times d_k}, \qquad V = UW_V \in \mathbb{R}^{n \times d_v}. \tag{2.3}$$

The vectors $v_j = u_j W_V$ from (2.1) come from the ***value matrix*** V. We call Q the ***query matrix*** and K the ***key matrix***. The score is given by

$$\alpha_{ij} = \frac{\exp\left(\frac{q_i \cdot k_j}{\sqrt{d_k}}\right)}{\sum_{\ell=1}^{n} \exp\left(\frac{q_i \cdot k_\ell}{\sqrt{d_k}}\right)} \tag{2.4}$$

where $q_i = u_i W_Q$ and $k_j = u_j W_K$. Put this all in a matrix, and we have the ***attention matrix***:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \in \mathbb{R}^{n \times d_v}. \tag{2.5}$$

Here, the softmax of a matrix is done row-wise, and it returns a row vector. Recall that for $u \in \mathbb{R}^d$, we define $\text{softmax}(u) \in \mathbb{R}^d$ such that for each $i \in \{1, \ldots, d\}$,

$$\text{softmax}_i(u) = \frac{\exp(u_i)}{\sum_{j=1}^{d} \exp(u_j)} \in [0, 1].$$

As usual, (2.5) need not demonstrate exactly what the actual code does; see [4] for an improved version. (At the time of writing they are on Flashattention-4.)

To make (2.4) less abstract, Figure 2.2 shows a real attention pattern from GPT-2 small on the sentence "The apprentice closely watches the watchmaker close the watches." Each row is the probability distribution $(\alpha_{i,1}, \ldots, \alpha_{i,n})$ for one query position. Real attention patterns are noticeably messier than one might expect—no single head carries the entire semantic story, and different patterns in different layers learn very different structures. The figure shown is one pattern chosen for clarity. In Figure 2.5, we look at all attention patterns.

Remark 2.7. We give some meaning to the score in (2.4). By taking the usual dot-product, we are concerned with the angle between the two vectors q_i and k_j. As we have seen before, softmax "smooths" vectors out. The factor $1/\sqrt{d_k}$ seems to address the problem of minuscule gradients [21, Section 3.2.1].

Before moving on to the next section, it might be helpful to consider some elementary exercises.

Exercise 2. What is the attention matrix when $QK^T = 0$? What does this mean in terms of attention?

The following exercise motivates why position must be encoded: self-attention is permutation equivariant. And the next one (Exercise 4) actually motivates why some of the later components to the transformer exist; see Section 2.5.

Exercise 3. Let $P \in \mathbb{R}^{n \times n}$ be a permutation matrix. Show that

$$\text{Attention}(PQ, PK, PV) = P \cdot \text{Attention}(Q, K, V).$$

Exercise 4. Prove that the rows of the attention matrix $\text{Attention}(Q, K, V)$ each lie in the convex hull of the rows of V. What does this imply about what attention can and cannot do (geometrically)?

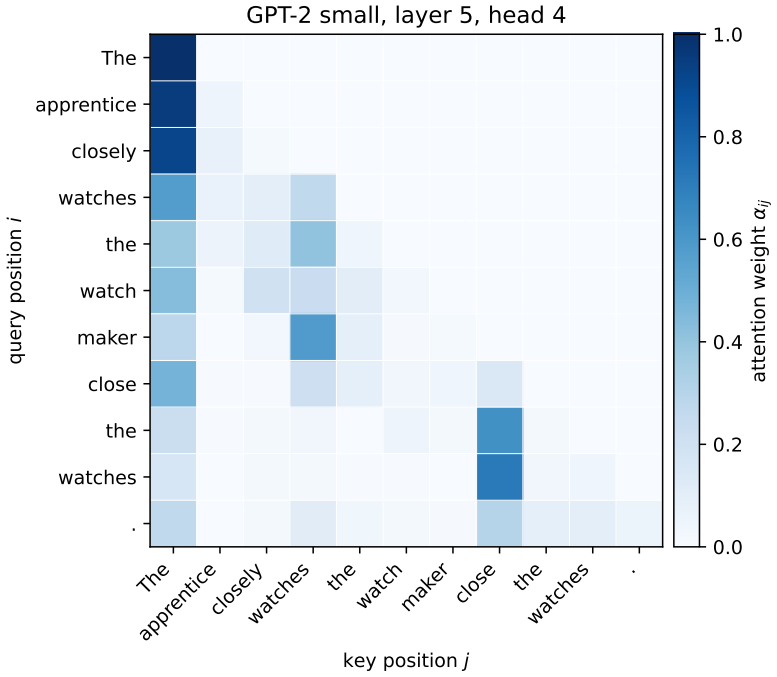


Figure 2.2: A real attention pattern from GPT-2 small

2.4 Multi-head attention

Now we take the basic ideas of self-attention and do this multiple times in parallel. Each head will do exactly what we covered in Section 2.3—usually with the exception that d_k and d_v be less than d, that is, each head will project the representations onto some subspace of $\mathbb{R}^d$. The idea is that different heads might learn or focus on different aspects of the relationship of attention. Computationally, we can perform the matrix multiplications for the attention matrix in parallel, so the division attention yields computational speed-up.

Suppose we have h heads. For each $i \in \{1, \ldots, h\}$, we have linear maps $W_Q^{(i)}, W_K^{(i)}, W_V^{(i)}$, and therefore we have h query, key, and value matrices $Q^{(i)}$, $K^{(i)}$, and $V^{(i)}$, respectively. One example of merging the attention matrices from the various heads is to simply concatenate the vectors: for a learned linear map $W_O \in \mathbb{R}^{hd_v \times d}$, define

$$\text{MultiHead}(U) = \text{Concat}\left(\text{head}_1(U), \ldots, \text{head}_h(U)\right) W_O,$$

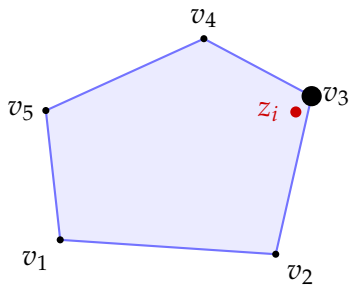
where

$$\text{head}_i(U) = \text{Attention}\left(UW_Q^{(i)}, UW_K^{(i)}, UW_V^{(i)}\right)$$

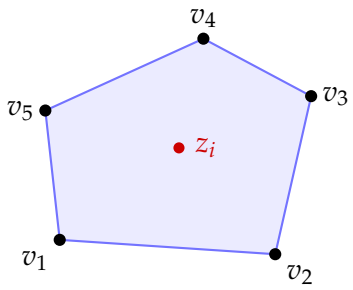
for all $i \in \{1, \ldots, h\}$.

2.5 The transformer block

Now that we have covered multi-head attention, we build a transformer block that incorporates this together with additional processing (and learning). We will



(a) Peaked attention. One α_{ij} dominates, so z_i is near a single vertex.



(b) Diffuse attention. The α_{ij} are close to uniform, so z_i is near the centroid.

Figure 2.3: Two extreme scenarios related to the conclusion from Exercise 4 in $\mathbb{R}^2$

discuss a transformer block at a high-level; the purpose is only to the steps involved. Suppose we have $U^{(\ell)} \in \mathbb{R}^{n\times d}$ viewed as n tokens embedded in $\mathbb{R}^d$ potentially with some additional information. The purpose of one transformer block will be to construct $U^{(\ell+1)} \in \mathbb{R}^{n\times d}$. Thus we can chain any number (a hyperparameter!) of transformer blocks together.

We break down the construction of $U^{(\ell+1)}$ into two key steps. Both steps finish by normalizing matrices; it does this essentially by centering and scaling the matrices to keep values stable while preserving (as best it can) the relative geometry. We have enough to describe the first step, which is all about attention:

$$\widehat{U}^{(\ell)} = \text{Normalize}\left(U^{(\ell)} + \text{MultiHead}\left(U^{(\ell)}\right)\right).$$

We note that attention is not replacing what we started with, but it is shifting or (ideally) correcting our embedded vectors.

In the second step, we act on the output of the first step, and simply use one MLP to manipulate each of the individual vectors. For a matrix X, we write $\text{FFN}(X)$ for the matrix obtained by feeding the rows of X into some MLP. In [21, Section 3.3],

$$\text{FFN}(x) = \sigma(xW_1 + b_1)W_2 + b_2,$$

where $W_1, W_2^T \in \mathbb{R}^{d\times d_{\text{ff}}}$ and σ is some activation function (e.g. ReLU). Thus, we define the output of the transformer block to be

$$U^{(\ell+1)} = \text{Normalize}\left(\widehat{U}^{(\ell)} + \text{FFN}\left(\widehat{U}^{(\ell)}\right)\right).$$

This step is motivated by the limitation encountered in Exercise 4: escaping the convex hull of the rows of the value matrix. Many modern implementations tweak this design, but we are staying mostly aligned to the original paper ([21]). The complete data flow of one block is shown in Figure 2.4; the blue arrows are the two residual connections.

Once blocks are stacked, the attention matrix in Figure 2.2 is no longer one matrix but many: one for each (layer, head)-pair. To get a sense of the diversity, Figure 2.5 shows all $12 \times 12 = 144$ attention matrices produced by GPT-2 small

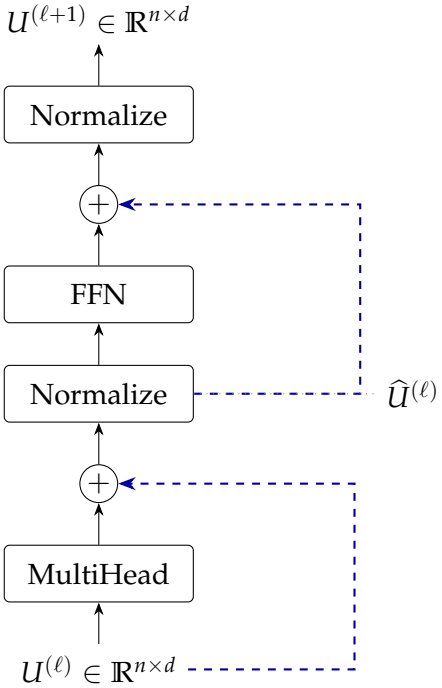


Figure 2.4: One transformer block; dashed arrows indicate residual connections

on the watchmaker sentence. Rows are indexed by layers and columns by heads. Several patterns are worth pointing out to students. Many heads have a dark first column: this is the "attention sink" phenomenon [24], where heads dump probability mass onto the first token when they have nothing in particular to look at. Early layers (say, layers 0–2) often contain heads that attend to the previous token or to a small local window—structurally similar to a convolution. Later layers tend to spread attention more broadly. The takeaway for the rest of this section is not to read off semantics from any single matrix, but to appreciate that the transformer produces 144 different views of the same sentence, and the rest of the transformer block is what stitches these views back together into a single contextualized representation.

2.6 Next token prediction

What we really mean is next token instance prediction. Suppose we have a sequence of token instances $(t_1, \ldots t_n) \in V^*$. The goal is simply to predict the next token instance t_{n+1}. Thus, a natural follow-up question is how do we create a probability distribution on V from $U^{(\ell)}$?

We have not yet taken our contextualized representation $U^{(\ell)} \in \mathbb{R}^{n\times d}$ and produced tokens. We introduce another learned linear map $W_{\text{out}} \in \mathbb{R}^{d\times|V|}$. We view

$$\ell_i = u_i^{(\ell)} W_{\text{out}} \in \mathbb{R}^{|V|}$$

as a vector of scores—the larger the jth coordinate (called the ***jth logit***), the more plausible the token ID for the next token after the prefix $t_1, \ldots, t_i$ is j. So with a

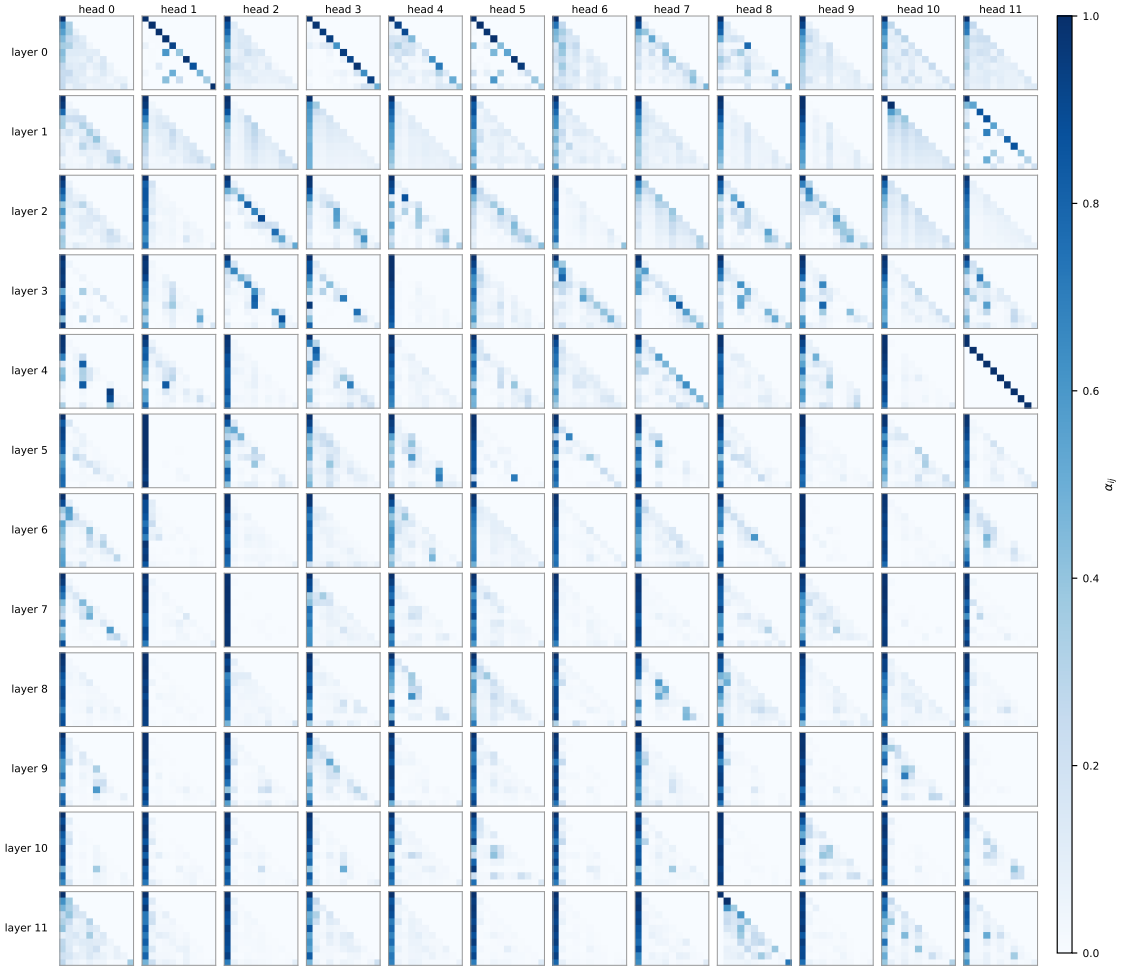


Figure 2.5: All $12 \times 12 = 144$ attention matrices produced by GPT-2 small on the sentence. Tokens are omitted; see Figure 2.2 for a single panel.

vector of scores, we have already seen that we can apply softmax to get an honest probability distribution on the tokens, which addresses our question. More specifically, for $v \in V$,

$$\mathbf{P}(t_{n+1} = v \mid t_1, \ldots, t_n) = \frac{\exp(\ell_n(v))}{\sum_{u \in V} \exp(\ell_n(u))},$$

where $\ell_n(u)$ is the logit associated to the token type u.

Now that we have a probability distribution on V, we can train and compare predictions with reality. In other words, next token prediction can be viewed as a classification problem, and as Tobias told us in the first lecture, we can expect to use the *cross-entropy loss function*. We briefly recap what he covered and align the notation to use our parameters.

Let $K = |V|$ be the cardinality of our vocabulary, $I = \{1, \ldots, K\}$ the set of token IDs, and Δ_K° the interior of the standard K-simplex. The cross-entropy loss function $L : \Delta_K^\circ \times I \to \mathbb{R}$ is given by

$$L(p, i) = -\log(p_i) = \log\left(\sum_{j=1}^{K} \exp(\ell_{nj})\right) - \ell_{ni}.$$

All of the weights are then updated to minimize the (empirical) loss function. Stochastic gradient descent can be used here, but it can pose problems due to the depth of the chain of transformer blocks. Another optimizer, called Adam [21, Section 5.3] is typically used. Adam [11] is a gradient-based optimization method which keeps running averages of the gradient and its square, and uses these to adapt the step size for each parameter separately.

2.7 Closing the loop

With our chain of transformer blocks, we have too much interaction between the tokens. The token in position i receives context from *all* the tokens—even those ahead of it. For natural language processing, this would have causal implications: one could receive the context before the context was given for example. The fix is very simple; we apply a mask (called a ***causal mask***) M to prevent the token in position i from attending to all the tokens in front of it (in position $i + j$ for all $j \in \mathbb{N}$). More specifically, we apply the mask M to the matrix $QK^T/\sqrt{d_k}$ so that the masked entries occur above the diagonal and are equal to $-\infty$. Thus after applying softmax, the matrix is lower triangular; see Figure 2.6.

Now we have a way to use to the transformer as an autoregressive model. Given some initial sequence of tokens (say, a prompt), we can use the transformer to predict the next token, feed that sequence of tokens into the transformer to get the next token, and continue this until some termination condition is met. As a final technical note, throughout we have only discussed a *decoder-only transformer*. Our main source ([21]) builds an encoder-decoder transformer.

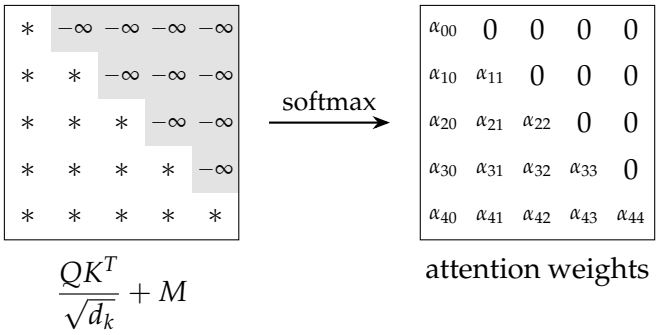


Figure 2.6: The causal mask

3 Transformers for discrete objects

As we learned in Section 2, the transformer architecture is well suited for natural language processing. They can, informally speaking, provide a distribution on language. However, transformers are not intrinsically tied to natural language itself but rather token sequences. In this section we depart the application towards natural language and look to applying transformers to discrete mathematical objects. But before we can do any of that, we need to tokenize our objects.

3.1 Some encodings of mathematical objects

We will use transformers to get distributions on mathematical objects. To do this, we need to tokenize our objects. We will look at a few examples encodings for some simple objects. We will look at finite subsets of positive integers, permutations of $\{1, \ldots, n\}$, and finite simple graphs.

Example 3.1. Suppose $S \subseteq \mathbb{N}$ is finite. We could encode S via its indicator vector; for example,

$$\{3, 5, 8\} \longmapsto (0, 0, 1, 0, 1, 0, 0, 1).$$

Another could simply be encoding S via its elements, e.g.

$$\{3, 5, 8\} \longmapsto (3, 5, 8).$$

Notice that both encodings are using the order on $\mathbb{N}$—this may or may not be desirable. Another example is to provide a program or description. For example,

$$\{2, 8, 14, 20\} \longmapsto \text{“positive integers less than 24 equal to 2 modulo 6”}.$$

Example 3.2. Suppose $\pi \in S_n$ is a permutation of $\{1, \ldots, n\}$. One example is its one-line encoding: for $n = 5$ and $\pi = 45231$,

$$\pi \longmapsto (4, 5, 2, 3, 1).$$

Another is its matrix encoding (but flattened)

$$\pi \longmapsto \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Our last example is its cycle encoding as a string:

$$\pi \longmapsto \text{“}(1, 4, 3, 2, 5)\text{”},$$

but people familiar with Coxeter theory might have another way.

Example 3.3. Now we consider a finite simple graph $G = (V, E)$ with $V = \{1, \ldots, 5\}$. One natural example is the adjacency matrix (but flattened):

$$G \longmapsto \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Another could just extract the strictly upper triangular part. Another could be an edge list: for example

$$G \longmapsto \text{“[(1,2), (1,5), (2,4), (3,4), (3,5)]”}.$$

Remark 3.4. I think graphs illustrate best out of the examples we have considered that it might be worthwhile to consider canonical encodings, if possible. Take the adjacency matrix encoding. Already for graphs on 10 vertices, there are $2^{\binom{10}{2}} = 35,184,372,088,832$ many adjacency matrices (or labelled graphs) but only $12,005,168$ graphs up to isomorphism [18]. One could also view this as having multiple ways to convey the same meaning (in language), so this is not necessarily a problem.

The previous examples illustrate an important point: a model usually does not see a mathematical object directly. It sees an encoding of that object. Thus, an encoding is a choice of coordinates on a class of mathematical objects. As usual, a good choice of coordinates can reveal structure, while a poor choice can obscure it. Let $\mathcal{X}$ be a class of mathematical objects and let

$$E : \mathcal{X} \to V^*$$

be an encoding. If we train a transformer on the token sequences $E(x)$, then the model is learning from the representations $E(x)$, not from the objects x themselves in some representation-independent sense. Consequently, artifacts of the encoding may become statistically meaningful even when they are mathematically irrelevant.

So what makes a good encoding? This will inevitably require a bit of trial and error, but here are three criteria worth thinking about:

1. validity,
2. uniqueness, and
3. locality.

Validity. The transformer will be generating next tokens based on a probability distribution. If one has encoded permutations in one-line notation, then there's an important and uncompromising rule: each integer must appear exactly once for a valid permutation. However, in the cycle notation, the rule is more flexible since duplicates are allowed in different cycles. An example of a very flexible encoding is a Coxeter-theoretic encoding. This does not preclude more rigid encodings; it just means that the transformer will need to "learn" the rules, which likely means it will need to train on more data. For example, the token strings for a typical modern-day transformer would not necessarily be a word in a language [9], yet they do not produce garbled up output text.

Uniqueness. As illustrated in Remark 3.4, different token sequences might represent isomorphic mathematical objects. With graphs given by an adjacency matrix, the transformer might be learning about labelings since that is part of the data. This might be avoided by choosing a canonical representative from each isomorphism class, for example by using a canonical labelling algorithm so that two graphs are isomorphic if and only if their canonical adjacency matrices are equal.

Locality. Informally, locality asks whether pieces of information that are mathematically related are nearby, or otherwise simply related, in the chosen token string. This is not a strict requirement: self-attention allows every token to attend to every other token. However, global access is not the same as effortless learning. If a simple mathematical relation is encoded as a complicated relation among distant or inconsistently placed tokens, then the model may need more data and more parameters to learn it. For permutations, one-line notation

$$\pi \longmapsto (\pi(1), \ldots, \pi(n))$$

makes some features local. For instance, descents are visible from adjacent entries:

$$\pi(i) > \pi(i+1).$$

Other features are less local. An inversion involves a pair

$$i < j, \qquad \pi(i) > \pi(j),$$

where i and j may be far apart.

There is no universally best encoding. A good encoding is one whose artificial structure aligns with the mathematical structure relevant to the task. This will inevitably take some practice.

3.2 What to learn and how to tell

After choosing an encoding, the next question is **what** we want the model to do with the encoded objects. Sometimes we want to

- predict a function,
- generate new examples,
- complete partial objects, or
- guide a search procedure.

While these are not necessarily mutually exclusive, these are different learning problems, even if the same transformer architecture can sometimes be adapted to all of them. In fact, the encoding might be informed by the problem at hand, so it is important to try many different options.

Now let us suppose we have established how to encode and what to learn. We should be thinking about how to tell if our eventual model is successful. There is really no point to building a model if we do not have a way of measuring whether it is successful. I think we have a few inter-related components to consider: data, train-test splits, and baselines. This list is not exhaustive!

Data. One of the biggest ways we can incorporate an unintended bias is by the way we generate our datasets. For example if we want to predict a function on the class of simple graphs whose output is only seen on graphs with a reasonably large automorphism group, then a massive dataset of random graphs (say, the Erdős–Rényi model) will have trivial automorphism groups and will therefore be unhelpful. The data may be abundant, but it is not representative of the mathematical situation we care about.

One way to address this is to have some "positive" and "negative" data. Determining a good proportion can be tricky and require a bit of trial and error. Nevertheless, if we have two algorithms to construct these two classes, we should still be careful as the machine might learn only artifacts of the algorithms. It may be impossible to eliminate all biases, but considering the various ways examples can arise will yield a richer dataset.

Train-test splits. What is there to say about how we split our data? Clearly a random split is the best! Actually, not always. A random split is a common default, and this will test how the model interpolates, but there are other situations where different kinds of splits might be interesting.

If we enumerate all graphs on at most n vertices and randomly split them, the test set will contain graphs of the same sizes and from the same distribution as the training set. High accuracy then tells us that the model can perform well on more examples drawn from the same pool, but not necessarily that it has learned a principle that extends to larger graphs.

If, instead, we train on graphs on at most n vertices and test on graphs on at least $n+1$ vertices, then we could see significantly different results. Indeed,

high accuracy here suggests that the model has learned a size-robust pattern. Of course this situation is much harder to attain than in the random split example.

There are many more kinds of splits. For example, we could train on certain families of graphs and then test on completely different families. Similarly we could train on "canonical" copies of graphs, and then test on isomorphic graphs. As above, high test accuracy suggests something interesting about what was learned—and all of these situations are rarer than high test accuracy in the random split.

Put simply, the train-test split is also part of the experiment. A performance number is hard to interpret without knowing what kind of split produced it.

Baselines. We need to know about how we can compare our eventual model against other (usually more trivial) models. Suppose we are trying to a predict a function that amounts to binary classification. If our transformer gets 66% accuracy on our dataset, that alone is meaningless. If two thirds of our dataset is in one class, then we built a complicated model to just simply return the majority or just randomly guess. In this example, a baseline for accuracy is 66%, and we would want to see higher accuracy for a more complicated model.

It might be more complicated than simply random guessing. For example, if regression makes sense, we should understand the accuracy of regression on our data, or decision trees, or nearest neighbors. All of these are simpler models. If a transformer out-performs these, then this suggests that the transformer is learning a novel pattern from our data.

3.3 PatternBoost

The team that designed the PatternBoost method [3] designed a simple and recursive method to construct mathematical objects. They apply this method to several unrelated problems—to name some of them: maximizing edges of a graph on n vertices without triangles; maximizing the matrix permanent on binary $n \times n$ matrix avoiding the 312 pattern; and maximizing the number of edges that can be deleted from the d-dimensional hypercube without decreasing its diameter.

Let's describe PatternBoost now, and we will use their example of maximizing the edges in a graph without having any triangles. We need two functions: a score and local search functions. Our score function will be

$$G \longmapsto \#E(G) - 2 \cdot \#\{\{u, v, w\} \subseteq V(G) \mid u \sim v \sim w \sim u\}.$$

Note that a graph can still score highly even with triangles. But the scalars are tuned in such a way that if G has exactly k triangles and if G' is obtained from G by removing exactly one of the edges of one of the triangles, then

$$\text{score}(G) < \text{score}(G').$$

This gets at our local search function: given a graph G, remove edges until it no longer has triangles, and then add edges until no more edges can be added (without creating a triangle). Call the output $\text{local}(G)$. Therefore,

$$\text{score}(G) \leqslant \text{score}(\text{local}(G)).$$

To get started, we need data: we build a reasonably large dataset of graphs. Ideally, we start with a dataset with as large a score as possible, but any dataset will do. We can easily generate a dataset by running the local search on the empty graph N times, where N is the number of examples we want to build.

Once we have our dataset, we train our transformer—first we build the transformer. The PatternBoost team used as comparatively small transformer (Makemore by Andrej Karpathy [10]), which may have caused some minor problems, but it is a good idea to start small and scale up, than to start too large. After the transformer has been trained on the dataset, use it to generate many examples—it could be that a large proportion of transformer outputs do not decode properly do graphs. While this might be mitigated with different hyperparameters or more training, it can be worked around by just simply generating many more examples. Once you have a sufficient number of generated examples, run the local search function on all of the examples. After that, repeat the process as much as you want.

The PatternBoost team establishes two baselines in this context: a local-only approach and a global-only approach. Both do not do as well as the local-global search; see the Discussion subsection in [3, Section 2].

3.4 Program search and FunSearch*

A slightly older and similar approach to PatternBoost is FunSearch [16]. The crucial conceptual shift is that we do not ask the transformer to produce mathematical objects directly. Instead, we ask it to produce a short computer program, such as a Python program, whose job is to construct or score the objects we care about. The search happens in the space of programs rather than in the space of potential solutions.

Why insert this layer of indirection? A program is usually far shorter than the object it describes, so the model has a much easier task. A program is interpretable, in the sense that a human can read it, simplify it, even prove things about it. And the same program may be evaluated on many instances of a problem, so a single discovery scales beyond the regime in which it was found.

Concretely, the user supplies three ingredients:

- a problem skeleton (containing boilerplate code and previous knowledge of the problem in the form of a program structure)
- an initial function, and
- an evaluator that scores candidate solutions.

We describe what these three ingredients should do, but it can also be useful to look at the example provided by the FunSearch team: See Figure 3.1 for an example regarding the cap set problem. The problem skeleton is an executable program with a single function left as a placeholder. In Figure 3.1, this is `main`. The initial function is exactly the placeholder function that the LLM should evolve; see `solve` in the figure. Finally the evaluator that takes a candidate program, runs it, and returns a real number (the score); this is `evaluate` in the figure.

The system then maintains a database of candidate programs together with their scores. Each step samples several high-scoring programs from the database, assembles them into a prompt, and asks a pretrained LLM to produce a new variant in the same style. The new program is executed in a sandbox and scored. To avoid collapse onto a single locally optimal idea, the database is split into "islands", each with its own pool, with occasional resetting [16, p. 470].

Compare this with PatternBoost. There, the transformer outputs the mathematical objects themselves and a hand-written local search polishes each. In FunSearch the transformer outputs programs and a hand-written evaluator scores them; the local search, if any, is whatever the program decides to do. In both cases the human-written code provides the deterministic, verifiable component, and the LLM provides creative variation.

The headline result of [16] is for the cap set problem: how large can a subset $A \subseteq \mathbb{F}_3^n$ be if no three of its elements sum to zero? The skeleton was a priority function $\mathbb{F}_3^n \to \mathbb{R}$, and a greedy procedure added vectors in order of decreasing priority while avoiding three-term progressions. FunSearch produced priority functions giving cap sets larger than the previously best-known ones in dimension $n = 8$, together with the largest improvement in two decades to the asymptotic lower bound. The team applied the same pipeline to online bin packing and recovered heuristics outperforming the textbook first-fit and best-fit rules on standard benchmarks. The resulting Python code is short enough to print on one page; see [16, Figures 4 & 6].

Remark 3.5. Most candidates proposed by the LLM are uninteresting: they fail to compile, crash on the evaluator, or score below their parent. The whole point of having the evaluator in the loop is that we do not need the LLM to be correct, only to be occasionally lucky.

3.5 AlphaEvolve*

AlphaEvolve [14] can be viewed as a substantially scaled-up version of FunSearch. The outer evolutionary loop is the same, but several restrictions are lifted. The LLM may now modify a whole codebase, not just a single function. For instance, it may change the model architecture, loss function, hyperparameters, and the priority function simultaneously. Rather than rewriting code from scratch, the LLM proposes a *diff*, in the format used by version-control software like git. The evaluator is permitted to return a *vector* of scores (correctness, runtime, memory usage, etc.) rather than a single scalar. And several LLMs of different sizes work together: a small, cheap model proposes many candidate edits, while a larger, slower model is reserved for the difficult steps [14, Section 2].

The applications reported in [14] span pure mathematics, hardware verification, and large-scale infrastructure at Google. We highlight one of mathematical interest.

Faster matrix multiplication. Strassen's 1969 algorithm multiplies two 2×2 matrices using 7 scalar multiplications rather than the obvious 8 [19]. This clever

```python
"""Finds large cap sets."""
import numpy as np
import utils_capset

# Function to be executed by FunSearch.
def main(n):
  """Runs `solve` on `n`-dimensional cap set and
  ↪  evaluates the output."""
  solution = solve(n)
  return evaluate(solution, n)

def evaluate(candidate_set, n):
  """Returns size of candidate_set if it is a cap
  ↪  set, None otherwise."""
  if utils_capset.is_capset(candidate_set, n):
    return len(candidate_set)
  else:
    return None

def solve(n):
  """Builds a cap set of dimension `n` using
  ↪  `priority` function."""
  # Precompute all priority scores.
  elements = utils_capset.get_all_elements(n)
  scores = [priority(el, n) for el in elements]
  # Sort elements according to the scores.
  elements = elements[np.argsort(scores,
  ↪  kind='stable')[::-1]]

  # Build `capset` greedily, using scores for
  ↪  prioritization.
  capset = []
  for element in elements:
    if utils_capset.can_be_added(element, capset):
      capset.append(element)
  return capset

# Function to be evolved by FunSearch.
def priority(element, n):
  """Returns the priority with which we want to add
  ↪  `element` to the cap set."""
  return 0.0
```

Figure 3.1: FunSearch example pulled directly from [16, Figure 2]

observation can be used to recursively speed up matrix multiplication. For two $n \times n$ matrices, the one we do by hand is $O(n^3)$, but using Strassen's algorithm it is $O(n^{\log_2 7}) = O(n^{2.81})$. Applying it recursively to a 4×4 matrix viewed as a 2×2 matrix of 2×2 blocks gives 49 scalar multiplications—in effect this shows that a specific tensor has *rank* at most 49.

AlphaEvolve searched not for a 4×4 algorithm directly, but for a Python program implementing a gradient-based optimizer of tensor decompositions, and let *that program* solve the optimization. The system arrived at a decomposition of the 4×4 matrix multiplication tensor over $\mathbb{C}$ [14, Section 3] that proves that its rank is at most 48: the first such improvement over Strassen's recursion that works over a non-commutative ring. This also improves on the earlier AlphaTensor system [6], a specialized reinforcement-learning approach which had found new 4×4 algorithms in finite-field settings but not over the complex numbers.

Remark 3.6. A version of this story is sometimes told as "the first improvement on Strassen's 4×4 algorithm in 56 years", which is not quite right. Winograd already gave a 48-multiplication algorithm in the 1960s, but only in the commutative setting, where it does not lift to recursion on larger matrices. The genuinely new content of AlphaEvolve's construction is that it works over arbitrary rings. See [5] for the full history and a refinement valid over $\mathbb{Q}$.

For the working mathematician, the lesson of AlphaEvolve is methodological rather than (yet) revolutionary. Over 50 open problems were treated in [14, Section 3], drawn from mathematical analysis, geometry, combinatorics, and number theory. In most cases AlphaEvolve matched the best known construction; in a smaller subset it improved on it, sometimes by small but verifiable amounts. The bottleneck is in the human design choices we have discussed at length: the encoding of the problem, the evaluator, the choice of program skeleton. The transformer is doing the recombination; the mathematics is in the framing.

3.6 Graph attention networks*

So far the only structure on our inputs has been a linear order: token 1, token 2, and so on. The self-attention mechanism does not use that order intrinsically—recall the permutation-equivariance exercise in Section 2.3—it is the positional encoding that puts it back. This raises the question of whether the same mechanism applies to inputs with very different connectivity. The cleanest such example is when the input is a graph.

Let $G = (V, E)$ be a (finite simple) graph with $n = |V|$ vertices, and suppose each vertex $i \in V$ carries a feature vector $h_i \in \mathbb{R}^f$. Write $\mathcal{N}(i)$ for the set of neighbours of i together with i itself. The ***graph attention network*** (GAT) [22] updates these features by letting each vertex attend only (!) to its neighbors.

Remark 3.7. There are two popular activation functions: Rectified linear unit (ReLU) and LeakyReLU [12]. The former was first used in 1941 [8]. They are

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

and for some negative parameter α,

$$\mathrm{LeakyReLU}(x) = \max(0, x) + \alpha \min(0, x).$$

LeakyReLU was constructed to avoid the problem of a having a region of values with zero gradient. Thus, α is usually some small negative value such as -0.1.

Fix a learned linear map $W \in \mathbb{R}^{f \times f'}$ and a learned vector $a \in \mathbb{R}^{2f'}$, and define

$$e_{ij} = \mathrm{LeakyReLU}\left(a^T \left[Wh_i \,\|\, Wh_j\right]\right) \qquad \text{for } j \in \mathcal{N}(i), \tag{3.1}$$

where $\|$ denotes concatenation. The normalized attention weights are

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik})}.$$

And the updated feature at vertex i is

$$h_i' = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} \, Wh_j\right), \tag{3.2}$$

where σ is some non-linear activation function. As in Section 2.3, multiple heads are run in parallel and their outputs concatenated.

Remark 3.8. Two differences with sequence self-attention are worth noting. First, the sum in (3.2) runs over neighbors only: vertex i does not attend to its non-neighbors. Equivalently, the matrix (e_{ij}) is masked by the adjacency matrix of G before softmax, in direct analogy with the causal mask of Section 2.7: entries outside the support of the adjacency are set to $-\infty$. Second, the scoring function (3.1) is additive rather than the scaled dot-product of (2.4). This is mostly a historical choice; modern graph transformers (see below) use the dot-product form.

A small but important caveat about (3.1) was raised some years later by Brody, Alon, and Yahav [2]. Writing $a = (a_1, a_2) \in \mathbb{R}^{f'} \times \mathbb{R}^{f'}$, the score becomes

$$e_{ij} \;=\; \mathrm{LeakyReLU}\big(a_1^T W h_i + a_2^T W h_j\big).$$

Because LeakyReLU is monotonic and the only j-dependent term is $a_2^T W h_j$, the ranking $j \mapsto e_{ij}$ for a fixed query vertex i is *independent of* i: every query produces the same ordering of the keys. They call this ***static attention***, and they exhibit synthetic graph problems that no GAT of any width can fit [2, Section 3]. The remedy, called ***GATv2***, is to swap the order of the linear projection and the nonlinearity:

$$e_{ij}^{\mathrm{v2}} = a^T \,\mathrm{LeakyReLU}\big(W[h_i \,\|\, h_j]\big). \tag{2.6$'$}$$

This costs no additional parameters and makes the attention function universal in the sense that any ranking of the neighbours is realisable by some choice of weights. GATv2 is a strict generalization of GAT, and the authors recommend (2.6′) as a drop-in replacement of (3.1).

Beyond GAT: graph transformers. GAT restricts attention to the neighbours of each vertex. A more recent and now dominant line of work drops this restriction and lets every vertex attend to every other vertex, exactly as in the sequence case, with the graph structure injected as bias terms in the dot-product attention rather than as a hard mask. Graphormer [25] encodes vertex degree, shortest-path distance, and edge features as additive biases, and won the OGB-LSC molecular property prediction competition in 2021. GraphGPS [15] combines local message passing with global transformer-style attention in a single block, and has become the basis of many subsequent architectures. NodeFormer [23] addresses the $O(n^2)$ cost of all-pairs attention by using a kernelized approximation, making the approach feasible for graphs with millions of vertices. At the time of writing, these graph-transformer architectures rather than GAT itself are the state of the art on most node- and graph-level benchmarks. The natural mathematical takeaway is that for problems on graphs, attention is still the right primitive; it is the choice of *what to attend to* that has shifted.

References

[1] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[2] S. Brody, U. Alon, and E. Yahav. How attentive are graph attention networks? In *International Conference on Learning Representations (ICLR)*, 2022.

[3] F. Charton, J. S. Ellenberg, A. Z. Wagner, and G. Williamson. PatternBoost: Constructions in mathematics with a little help from AI, 2024. arXiv:2411.00566.

[4] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359. Curran Associates, Inc., 2022.

[5] J.-G. Dumas, C. Pernet, and A. Sedoglavic. A non-commutative algorithm for multiplying 4×4 matrices using 48 non-complex multiplications, 2025. arXiv:2506.13242.

[6] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R. Ruiz, J. Schrittwieser, G. Swirszcz, D. Silver, D. Hassabis, and P. Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[7] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[8] A. S. Householder. A theory of steady-state activity in nerve-fiber networks: I. definitions and preliminary lemmas. *The bulletin of mathematical biophysics*, 3(2):63–69, 1941.

[9] Hugging Face. Tokenization algorithms. https://huggingface.co/docs/transformers/tokenizer_summary. Accessed: 2026-05-27.

[10] A. Karpathy. Makemore: An autoregressive character-level language model. https://github.com/karpathy/makemore, 2022. GitHub repository, accessed 2026-05-31.

[11] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *ICLR (Poster)*, 2015.

[12] A. L. Maas, A. Y. Hannun, A. Y. Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Atlanta, GA, 2013.

[13] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. 2013. arXiv:1301.3781.

[14] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. R. Ruiz, A. Mehrabian, M. P. Kumar, A. See, S. Chaudhuri, G. Holland, A. Davies, S. Nowozin, P. Kohli, and M. Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. arXiv:2506.13131.

[15] L. Rampášek, M. Galkin, V. P. Dwivedi, A. T. Luu, G. Wolf, and D. Beaini. Recipe for a general, powerful, scalable graph transformer. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[16] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. R. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, P. Kohli, and A. Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[17] M. Schuster and K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[18] N. J. A. Sloane. OEIS Sequence A000088, 2021. Accessed: 2026-05-27.

[19] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[20] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[22] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[23] Q. Wu, W. Zhao, Z. Li, D. Wipf, and J. Yan. NodeFormer: A scalable graph structure learning transformer for node classification. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[24] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations (ICLR)*, 2024.

[25] C. Ying, T. Cai, S. Luo, S. Zheng, G. Ke, D. He, Y. Shen, and T.-Y. Liu. Do transformers really perform badly for graph representation? In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.