Neural Networks for Data Science Applications

Master's Degree in Data Science

Lecture 9: Transformer (attention-based) models

Lecturer: S. Scardapane

SAPIENZA
UNIVERSITÀ DI ROMA

Designing the transformer

Moving beyond convolutional layers

The problem of convolutions

The assumption of **locality** embedded in convolutional layers is not always optimal: in a text, for example, a subject can depend on an object quite far from its position. In text, audio, graphs, etc., dependencies can be **sparse**, **long-range**, and possibly **dynamic**.

For example, in '*the cat is on the table*' and '*the cat, which belonged to my mother, is on the table*', the relation between the two words is similar, but their relative positioning is quite different.

Recurrent neural networks

Until a few years ago, **recurrent neural networks** (RNNs) were a viable alternative. The idea is to process a sequence of tokens $\mathbf{x}_i$ with a *stateful* function $\mathbf{h}_i = f(\mathbf{x}_i; \mathbf{h}_{i-1})$, hoping that all important information will be embedded in the state $\mathbf{h}_i$.

Because of their structure, RNNs can be time-consuming to train, as they need to backpropagate through each iteration (**backpropagation through time**). **Transformer** models have become common instead, especially when trained from huge datasets. In fact, most pre-trained word embeddings are built on this architecture.

Transformers

The core of the transformer is a new layer called **multi-head attention** (MHA). It replaces the assumption of locality with a more general notion of (soft) **sparsity** of interactions.

The original Transformer (Vaswani et al., 2017), was an encoder-decoder model for NLP tasks. Today, similar models are gaining interest in audio, computer vision, biology, etc.

Transformers have better scaling lawys

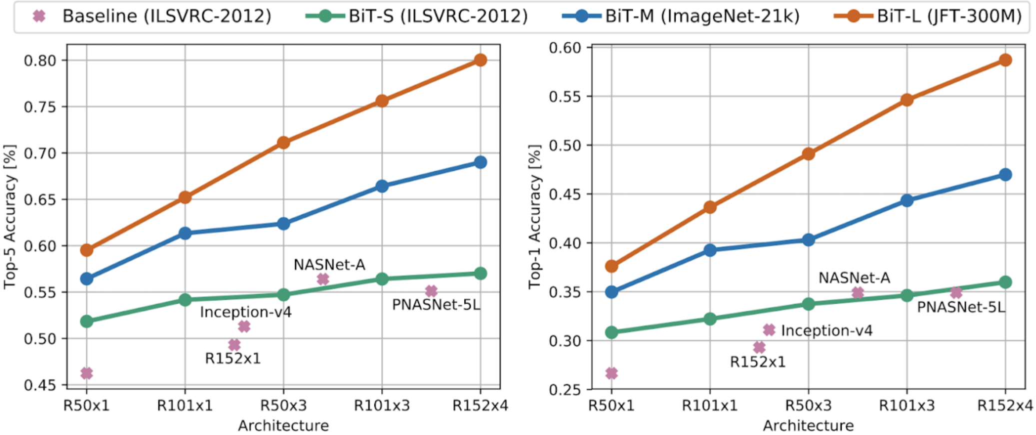


Figure 1: *Open-Sourcing BiT: Exploring Large-Scale Pre-training for Computer Vision* (Google AI Blog).

Designing the transformer

Self-attention

Moving beyond locality

Consider a 1D sequence $\mathbf{x}_1, \ldots, \mathbf{x}_n$, where $\mathbf{x}_i \in \mathbb{R}^d$. Because transformers originate from NLP, we call each element of the sequence a **token** and d the **embedding dimension**.

We can write a 1D convolutional layer (ignoring padding) of kernel size k as:

$$\mathbf{h}_i = \sum_{j=-k}^{k} \mathcal{W}_j \mathbf{x}_{i+j} \,, \tag{1}$$

where $\underset{(2k,d,d)}{\mathcal{W}}$ is the kernel tensor. We want to remove the assumption of locality while maintaining parameter efficiency.

Continuous convolutions

Increasing k increases the number of parameters linearly. As an alternative, we can *learn* the parameters of the kernel for each possible position i via a trainable block $g(i) : \mathbb{N} \to \mathbb{R}^{d \times d}$ taking as input the shift:

$$\mathbf{h}_i = \sum_{j=-i+1}^{n-i} g(i+j)\mathbf{x}_j \,. \tag{2}$$

These are called **continuous convolutions**, and they work well with image-like data with, e.g., variable sizes and resolutions.

Romero, D.W. et al., 2022. **Towards a General Purpose CNN for Long Range Dependencies in $\mathbb{N}$D**. *arXiv preprint arXiv:2206.03398*.

Non-local neural networks

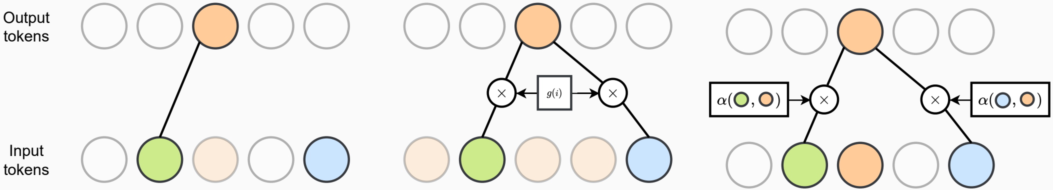
The previous model works well at handling non-locality, but it still assumes that dependencies are *regular*, i.e., they only depend on j. We can make it more general by letting them depend on the *values* of tokens instead:

$$\mathbf{h}_i = \sum_{j=1}^{n} \alpha(\mathbf{x}_i, \mathbf{x}_j)\mathbf{x}_j \,. \tag{3}$$

This is an example of a **non-local neural network** model. By a proper choice of the weighting function $\alpha(\cdot, \cdot)$ we can obtain the MHA layer.

Wang, X., Girshick, R., Gupta, A. and He, K., 2018. **Non-local neural networks**. In *IEEE/CVF CVPR*.

Schematic depiction



(a) Conv1D, kernel size 3 (b) Continuous convolution (c) Non-local model

Figure 2: Example of short-term (ST, blue) and long-term (LT, green) interaction. (a) Conv1D model: ST has a trainable weight, LT is removed; (b) both connections have weights given by $g(-1)$ and $g(2)$; both connections have weight that depend on the tokens' similarities.

Choosing the weighting function

We make a few assumptions to simplify the layer:

- The output of α is a scalar (not a matrix). We call α the **attention scoring function** (or attention function), and its outputs the **attention scores** for token *i*.
- For each token, its attention scores are normalized in the simplex (they are positive and they sum to one). With this formulation, each token will have a 'budget' of attention to allocate, i.e., increasing an attention score necessarily decreases the attention over the remaining tokens.

Attention layer

There are many choices for the attention function; commonly, we use the normalized dot product $\alpha(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{\sqrt{d}}\mathbf{x}_i^\top \mathbf{x}_j$ because it is fast and efficient to parallelize.

Putting everything together we obtain (always for a single token):

$$\mathbf{h}_i = \sum_{j=1}^{n} \text{softmax}_j \left(\frac{1}{\sqrt{d}} \mathbf{x}_i^\top \mathbf{x}_j \right) \mathbf{x}_j \,. \tag{4}$$

Why the extra factor $\sqrt{d}$? Suppose the elements of $\mathbf{x}_i$ are sampled according to $\mathcal{N}(0, \sigma^2)$. The variance of $\mathbf{x}_i^\top \mathbf{x}_j$ will be σ^4 (check!), which can easily saturate the softmax with a single large (positive or negative) value.

Adding some parameters

In its current formulation, the layer lacks trainable parameters. To this end, we first reproject the input three times using three trainable matrices:

$$\mathbf{q}_i = \mathbf{W}_q^\top \mathbf{x}_i, \quad \mathbf{k}_i = \mathbf{W}_k^\top \mathbf{x}_i, \quad \mathbf{v}_i = \mathbf{W}_v^\top \mathbf{x}_i \,.$$

We call these the **query**, **key**, and **value** (for reasons to be explained in detail later on). The **self-attention** (SA) layer can now be written as:

$$\mathbf{h}_i = \sum_j \text{softmax}\left(\frac{1}{\sqrt{d}}\mathbf{q}_i^\top \mathbf{k}_j\right) \mathbf{v}_j \,.$$

Vectorized version

Let us write the previous equation in a vectorized form, by stacking the n input vectors $\{\mathbf{x}_i\}$ into a matrix $\underset{(n,d)}{\mathbf{X}}$. SA can be rewritten as:

$$\underset{(n,q)}{\mathbf{Q}} = \mathbf{X}\mathbf{W}_q\,, \quad \underset{(n,q)}{\mathbf{K}} = \mathbf{X}\mathbf{W}_k\,, \quad \underset{(n,v)}{\mathbf{V}} = \mathbf{X}\mathbf{W}_v$$

$$\underset{(n,v)}{\mathbf{H}} = \operatorname{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{q}}\right)\mathbf{V}\,.$$

where the hyper-parameters are q and v. When the layer is applied to a batch of elements (e.g., sentences), it computes the attention function independently for every element of the batch (i.e., each token can *attend* only to tokens in the same sentence).

Visualizing the attention operation

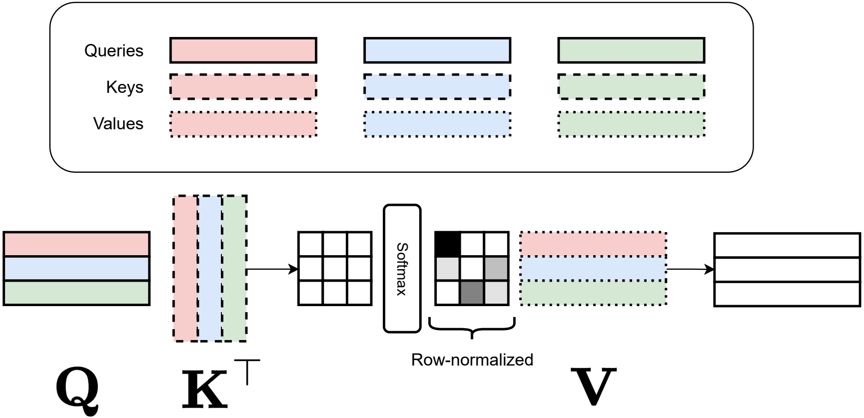


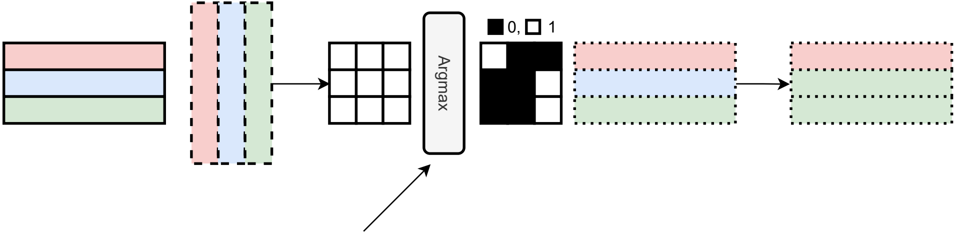
Figure 3: Visualization of the attention operation (ignoring the initial projections).

The dictionary analogy

To understand the terminology, consider a Python dictionary `d = dict(...)`. It is a collection of key/values `(k,v)` pairs, such that for a given query `d[q] = v` if `k` is stored inside. If the key does not exists, an error or default value is returned.

We can consider instead a ‘soft’ variant that always returns a value, by considering the value associated to the most similar key, even if a perfect match does not occur. If the keys, queries, and values are vectors and the distance is the dot product, this is equivalent to SA when replacing the softmax with an argmax over rows!

Hard attention as a dictionary



Difficult in practice
(gradient zero ≈ everywhere)

Figure 4: Hard attention is fundamentally equivalent to a dictionary with associative recall.

Multi-head attention

The SA layer can handle quasi-sparse dependencies (because of the softmax), and also dynamic ones (because of the attention function). However, what happens when the token can depend on multiple subsets of tokens?

A common generalization in this case is **multi-head attention** (MHA). It works by computing $i = 1, \ldots, h$ separate sets of keys, querys, and values:

$$\mathbf{Q}_t = \mathbf{X}\mathbf{W}_{q,t}\,, \quad \mathbf{K}_t = \mathbf{X}\mathbf{W}_{k,t}\,, \quad \mathbf{V}_t = \mathbf{X}\mathbf{W}_{v,t}$$

$$\mathbf{H}_t = \operatorname{softmax}\left(\frac{\mathbf{Q}_t\mathbf{K}_t^\top}{\sqrt{q}}\right)\mathbf{V}_t\,.$$

We now have $3h$ trainable matrices, or a $3 \times h \times q$ tensor assuming $q = v$.

Multi-head attention (2)

The previous operation has h separate outputs; we combine them by concatenation over the embedding dimension, and a final reprojection with a trainable output matrix W_o:

$$\mathsf{H} = \begin{bmatrix} \mathsf{H}_1 & \cdots & \mathsf{H}_h \end{bmatrix} \mathsf{W}_o \,.$$

As hyperparameters, we typically choose an embedding dimension m, an output size o, and a number of heads h, and we set $q = v = m /\!/ h$ for all heads.

Visualizing multi-head attention

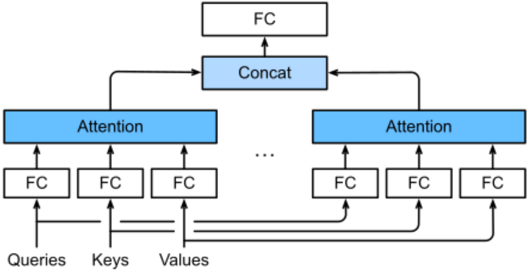


Figure 5: Visualization of the multi-head attention operation (D2L, Chapter 11.5).

Designing the transformer

The Transformer block

The transformer block

In transformers, the MHA layer is always used inside a more complex block, called the **transformer** block. Originally, this was composed of a MHA layer, two layer normalization operations, two residual connections, and a so-called **position-wise** network as follows:

1. Start with a MHA layer: $\mathbf{H} = \text{MHA}(\mathbf{X})$.
2. Add a residual connection and a layer normalization operation: $\mathbf{H} = \text{LayerNorm}(\mathbf{H} + \mathbf{X})$.
3. Apply a fully-connected model $g(\cdot)$ on each row: $\mathbf{F} = g(\mathbf{H})$.
4. Do again step 2: $\mathbf{H} = \text{LayerNorm}(\mathbf{F} + \mathbf{H})$.

Vaswani, A. et al., 2017. **Attention is all you need**. *NeurIPS*.

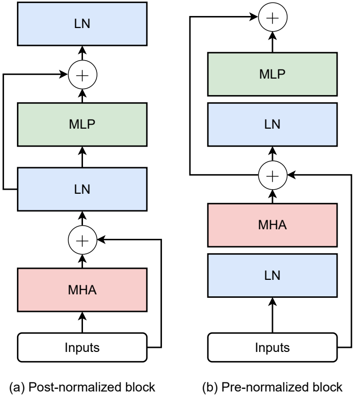
Design of the block

The design of the block was mostly based on empirical considerations. Roughly speaking, steps (1)-(2) correspond to a **token mixing** operation, while steps (3)-(4) are a per-token update which is akin to a 1x1 convolution. The block is similar in spirit to the depthwise separable convolution model.

The intermediate MLP is typically designed as a 2-layer MLP, with hidden dimension an integer multiple of the input dimension (e.g., 3x, 4x), and no biases.

Vaswani, A. et al., 2017. **Attention is all you need**. *NeurIPS*.

Pre- and post-normalized blocks



(a) Post-normalized block (b) Pre-normalized block

Figure 6: The original block is called **post-normalized**. A **pre-normalized** variant is also common due to it being simpler to train in most cases.[1]

---

[1] Xiong, R. et al., 2020. **On layer normalization in the transformer architecture**. *ICML*.

Variants of the transformer block

Many other variants are now common, depending on the application and computational considerations, e.g.:

- **Parallel** variants perform the MHA and MLP operations in parallel, i.e., $\mathbf{H} = \mathbf{H} + \mathrm{MLP}(\mathbf{H}) + \mathrm{MHA}(\mathbf{H})$. In this way, the initial and final projections of the MLP and MHA layers can be fused.[2]
- **Q/K normalized** variants add additional LN operations over the keys and queries (ibidem).
- **Multi-query** variants share the same keys and values over different heads to save computations.[3]

---

[2] Dehghani et al., 2023. **Scaling vision transformers to 22 billion parameters**. *ICML*.
[3] Shazeer, N., 2019. **Fast transformer decoding: One write-head is all you need**. *arXiv preprint arXiv:1911.02150*.

Designing the transformer

Positional embeddings

Permutation matrices

Consider the 3×3 matrix defined as:

$$\mathsf{P} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

It is easy to check that:

$$\mathsf{P} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_3 \\ x_2 \end{pmatrix}.$$

These are called **permutation matrices**.

The MHA layer is equivariant to permutations

In audio and text, the *i*th row of $\mathbf{X}$ represents a single time-step or a single text token (e.g., a word). In a MHA layer, their ordering is lost, because the layer is *equivariant* to the ordering (similar to the GAT layer for graphs).

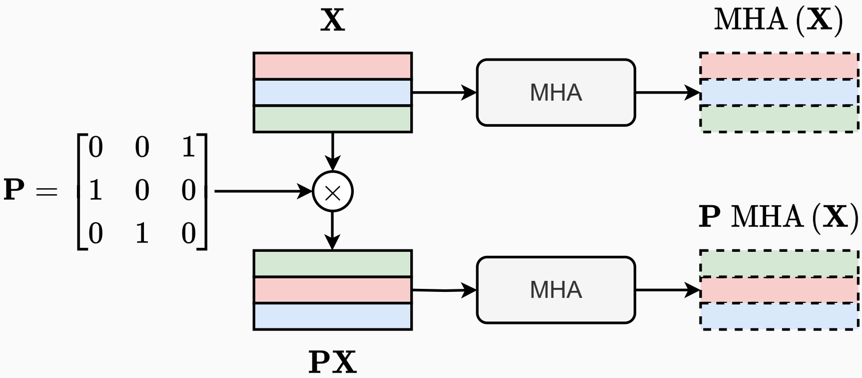
If we multiply $\mathbf{X}$ by a permutation matrix $\mathbf{P}$, then (the same holds trivially for the entire block):

$$\mathrm{MHA}(\mathbf{PX}) = \mathbf{P} \cdot \mathrm{MHA}(\mathbf{X})\,.$$

This is not a good property to have for sequences.

This is easy to show, since $(\mathbf{PK})^\top = \mathbf{K}^\top \mathbf{P}^\top$ and $\mathrm{softmax}(\mathbf{PQK}^\top \mathbf{P}^\top)\mathbf{PV} = \mathbf{P}\,\mathrm{softmax}(\mathbf{QK}^\top)\mathbf{V}$.

Visualizing permutation equivariance



Positional embeddings

Before the first MHA layer, we concatenate or sum to the input $\mathbf{X}$ a matrix of **positional embeddings** $\underset{(n,e)}{\mathbf{E}}$:

$$\mathbf{X}' = [\mathbf{X} \parallel \mathbf{E}] \quad \textit{or} \quad \mathbf{X}' = \mathbf{X} + \mathbf{E}\,,$$

where each row $[\mathbf{E}]_i$ should *uniquely* encode the position of every element of the sequence.

Using this strategy, we 'break' the equivariance:

$$\mathrm{MHA}(\mathbf{P}\mathbf{X} \parallel \mathbf{E}) \neq \mathbf{P} \cdot \mathrm{MHA}(\mathbf{X} \parallel \mathbf{E})\,.$$

Simple positional embeddings

We can encode the position for a sequence of maximum length p with a one-hot vector of dimension p, e.g.:

$$\mathbf{E}_0 = [1, 0, 0, \ldots] \ , \quad \mathbf{E}_1 = [0, 1, 0, \ldots] \ , \quad \mathbf{E}_2 = [0, 0, 1, \ldots] \ , \quad \cdots \ .$$

Or with a single increasing scalar:

$$\mathbf{E}_0 = [0/p] \ , \quad \mathbf{E}_1 = [1/p] \ , \quad \mathbf{E}_2 = [2/p] \ , \quad \cdots \ .$$

Both strategies are not particularly good empirically.

Trainable positional embeddings

We can *learn* the positional embeddings using the `tf.keras.layers.Embeddi` layer:

- To each position *i* we associate an embedding vector of fixed dimension.
- The embeddings are trained with the rest of the network.

Note that we need to fix the maximum length of the sentence. For longer sentences, we need to linearly interpolate the set of vectors up to a larger dimension (this is the strategy used in BERT and the Vision Transformer described below).

Sinusoidal embeddings

Consider a single sinusoidal function of frequency ω:

$$\mathsf{E}_i = [\sin(i\omega)] \ .$$

We can interpret this as a clock with frequency ω: for two points inside a single rotation, it will give us their relative distance. For other points, the distance will be precise modulo the frequency.

Multiple sinusoidal embeddings

To uniquely identify any possible position, we can consider multiple sinusoids, each with a frequency $\omega_j, j = 1, \ldots, e$:

$$\mathsf{E}_i = \left[\sin(i\omega_0)\,, \sin(i\omega_1)\,, \ldots\,, \sin(i\omega_e)\right]\,.$$

You can think of this as a clock with e different hands, each rotating at its own frequency. This is a nice representation because it can possibly generalize to any length, without the need to impose a maximum length *a priori*.

Choosing the frequency

An empirically good choice for the frequencies (popularized by (Vaswani et al., 2017)) is:

$$\omega_j = \frac{1}{10000^{j/e}} \,.$$

For $j = 0$, this has frequency 2π. For $j = e$, this has frequency $10000 \cdot 2\pi$. In the middle, the frequency are increasing at a geometric progression.

To reduce the number of parameters, it is also common to *sum* the positional encodings instead of concatenating (in which case the dimension e is equal to d):

$$\mathsf{X}' = \mathsf{X} + \mathsf{E} \,.$$

Final version

A popular extension is to alternate sines and cosines of the same frequency:

$$[\mathbf{E}]_{i,2j} = \sin\left(\frac{i}{10000^{2j/e}}\right) , \tag{5}$$

$$[\mathbf{E}]_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/e}}\right) . \tag{6}$$

One important property of this encoding is that it is possible to translate an encoding via matrix multiplication:

$$[\mathbf{E}]_{i+p} = [\mathbf{E}]_i \mathbf{T}(p) \quad \text{for some } \mathbf{T}(p) .$$

See https://kazemnejad.com/blog/transformer_architecture_positional_encoding/ and references therein.

Visualizing positional encodings

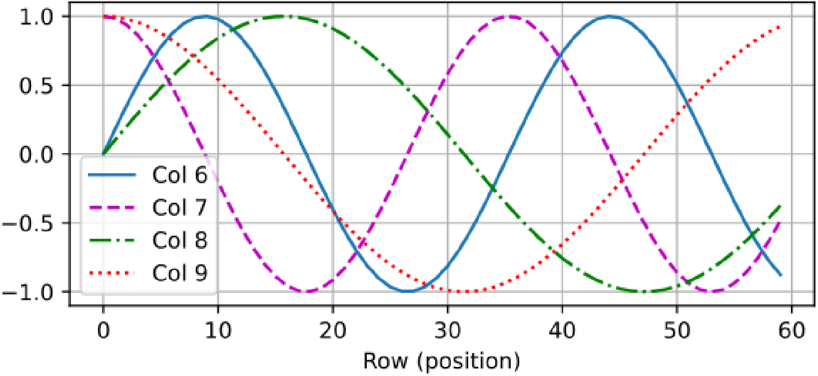


Figure 7: Visualization of the sinusoidal positional encodings (book, Chapter 10.6).

Relative positional embeddings

Another possibility is using **relative positional embeddings**. In this case, we modify the attention function to make it depend on the relative distance $i-j$ between tokens.

For example, **attention with linear biases**[4] (ALiBi) adds trainable biases b_{ij}:

$$\alpha(\mathbf{x}_i, \mathbf{x}_j, i-j) = \mathbf{x}_i^\top \mathbf{x}_j + b_{ij}\,. \tag{7}$$

Another common option are **rotary position embeddings**[5] (RoPE).

---

[4] Press, O., Smith, N.A. and Lewis, M., 2021. **Train short, test long: Attention with linear biases enables input length extrapolation**. *arXiv preprint arXiv:2108.12409*.

[5] Su, J. et al., 2021. **Roformer: Enhanced transformer with rotary position embedding**. *arXiv preprint arXiv:2104.09864*.

Visualization of ALiBi

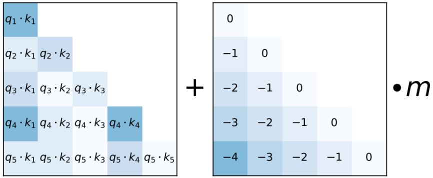


Figure 3: When computing attention scores for each head, our linearly biased attention method, ALiBi, adds a constant bias (right) to each attention score ($\mathbf{q}_i \cdot \mathbf{k}_j$, left). As in the unmodified attention sublayer, the softmax function is then applied to these scores, and the rest of the computation is unmodified. **m is a head-specific scalar** that is set and not learned throughout training. We show that our method for setting m values generalizes to multiple text domains, models and training compute budgets. When using ALiBi, we do *not* add positional embeddings at the bottom of the network.

Figure 8: Linear biases for attention (reproduced from Press, Smith, Lewis, 2021.).

Designing the transformer

The complete transformer model

Putting everything together

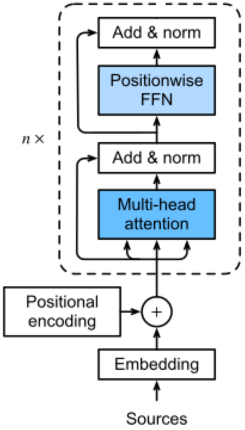


Figure 9: The final model is built with positional encodings and a stack of *n* transformer blocks (adapted from Chapter 10.7 of the book).

Classification token

To perform classification or regression, we can apply a final global pooling on the n tokens and one or more fully-connected layers.

An alternative that is empirically found to work well is the **class token**, which is an additional *trainable* token $\mathbf{c}$ added to the input matrix:

$$\underset{(n+1,d)}{\mathbf{X}'} = \begin{bmatrix} \mathbf{X} \\ \mathbf{c}^\top \end{bmatrix} \,.$$

The transformer model is applied to the matrix $\mathbf{X}'$ as input ($\mathbf{H} = \text{Transformer}(\mathbf{X}')$), and classification is performed on its last row:

$$\mathbf{y} = \text{softmax}(\mathbf{W}^\top [\mathbf{H}]_{n+1}) \,.$$

Designing the transformer

Causal models and encoder-decoder models

Encoder-decoder models

The original transformer model was a more general model defined for **sequence to sequence** (seq2seq) tasks, such as machine translation (variable number of tokens in inputs *and* in output).

It performed an **encoding** of the input sequence, which was then **decoded** by a second, masked transformer to generate the output sequence. In order to understand it, we need to introduce two further mechanisms: **masked attention** and **cross-attention**.

For this reason, the model described before is sometimes called an **encoder-only** Transformer.

Vaswani, A., et al., 2017. **Attention is all you need**. In *Advances in neural information processing systems* (pp. 5998-6008).

Building a causal transformer

In order to build a causal transformer variant, we can replace the SA layer with a masked variant:

$$\mathbf{H} = \operatorname{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top \odot \mathbf{M}}{\sqrt{q}}\right)\mathbf{V}\,.$$

where $\mathbf{M}$ has a triangular structure:

$$M_{ij} = \begin{cases} 1 & \text{if } j \leq i \\ -\infty & \text{otherwise} \end{cases}\,. \tag{8}$$

In practice we can use very small numbers, e.g., 10^{-10}.

Visualizing the masking operation

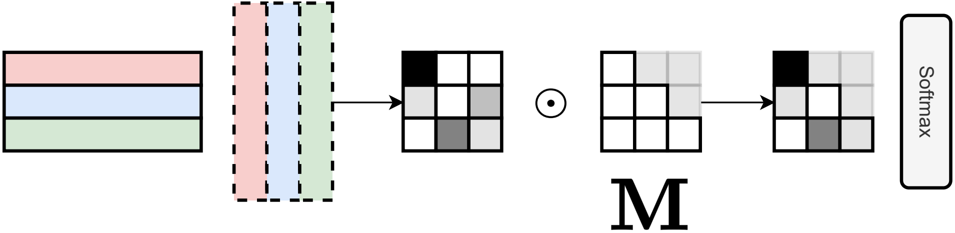


Figure 10: Note that masking with 0 is invalid because $\exp(0) = 1$, and masking after the softmax is invalid because of its denominator.

Cross-attention

Given *two* sets $\mathbf{X}$ and $\mathbf{Z}$, **cross attention** is defined as:

$$\mathrm{CA}(\mathbf{X}, \mathbf{Z}) = \mathrm{softmax}\left((\mathbf{Z}\mathbf{W}_q)\,(\mathbf{X}\mathbf{W}_k)^\top \right) \mathbf{X}\mathbf{W}_v \,. \tag{9}$$

This is a useful operation that can combine information coming from different streams of information (e.g., audio-visual datasets).

Full encoder-decoder architecture

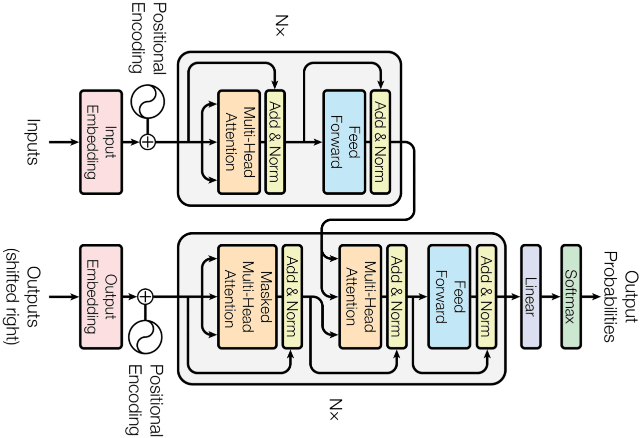


Figure 11: Reproduced from Vaswani et al., 2017.

Designing the transformer

Computational considerations

Comparing convolutive layers and MHA layers

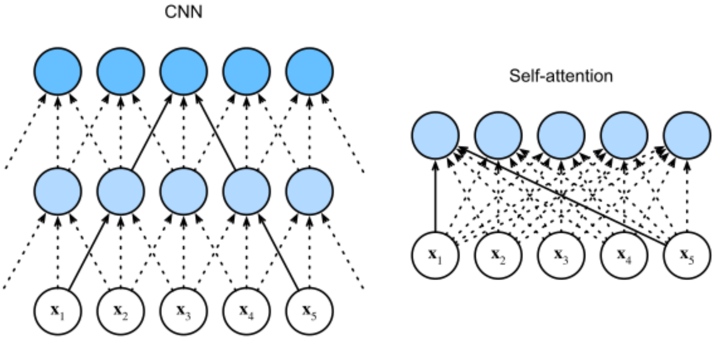


Figure 12: Adapted from Chapter 10.6 of the book.

Computational cost comparison

Consider a 1D convolutional operation $\underset{(n,d)}{\mathbf{H}} = \text{Conv1D}(\underset{(n,d)}{\mathbf{X}})$ with a filter size of k. Computing the output requires $\mathcal{O}(nkd^2)$ operations.

Self-attention (with one head) requires $\mathcal{O}(nd^2)$ for computing keys, queries, and values, and $\mathcal{O}(n^2d)$ for computing the output. The n^2 term limits the applicability to long sequences, unless more advanced models are used (e.g., $n < 512$ in many text models).

However, a single layer of MHA has a receptive field of n, while the convolutive layer has a receptive field of k.

There is a large interest in designing variants (or approximations) of SA that scale linearly in n. A popular class of methods relies on **kernel functions**. We first rewrite SA for a single token as:

$$\mathbf{h}_i = \frac{\sum_j \alpha(\mathbf{q}_i, \mathbf{k}_j)\mathbf{v}_j}{\sum_j \alpha(\mathbf{q}_i, \mathbf{k}_j)} .$$

where in our final model we had (ignoring scalar factors) $\alpha(\mathbf{x}, \mathbf{y}) = \exp(\mathbf{x}^\top \mathbf{y})$. Note that any positive-definite $\alpha(\cdot, \cdot)$ is a valid kernel function (as in, e.g., **support vector machines**).

A refresher on kernel functions

For some specific kernel functions, it is possible to rewrite them as a dot product in a different *finite-dimensional*, parameter-free feature space $\phi(\mathbf{x})$:

$$\alpha(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^\top \phi(\mathbf{y})\,. \tag{10}$$

(This is always possible if allowing for infinite-dimensional spaces, like in the exponential case.)

For example, $\alpha(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^\top \mathbf{y} + c)^p$ can be rewritten as (10) with $\phi(\mathbf{x})$ containing all terms $x_1^{i_1} x_2^{i_2} \dots x_d^{i_d} y_1^{j_1} y_2^{j_2} \dots y_d^{j_d}$ with $\sum_k i_k + j_k = p$.

Linearized transformers

In this case, we can rewrite the attention operation as:

$$\mathbf{h}_i = \frac{\sum_j \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)\mathbf{v}_j}{\sum_j \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)} = \frac{\phi(\mathbf{q}_i)^\top \overbrace{\sum_j \phi(\mathbf{k}_j)\mathbf{v}_j^\top}^{\mathbf{S}}}{\phi(\mathbf{q}_i)^\top \underbrace{\sum_j \phi(\mathbf{k}_j)}_{\mathbf{Z}}} \,. \tag{11}$$

where $\mathbf{S}$ and $\mathbf{Z}$ are now shared across all tokens and can be computed only once, making the operation linear in n.

Katharopoulos, A. et al., 2020. **Transformers are RNNs: Fast autoregressive transformers with linear attention**. *ICML*.

Recurrent formulation

In the autoregressive case, we replace $\sum_j$ with the masked version $\sum_{j=1}^{i}$. Define the partial sums $\mathsf{S}_i = \sum_{j=1}^{i} \phi(\mathsf{k}_j)\mathsf{v}_j^\top$ and $\mathsf{Z}_i = \sum_{j=1}^{i} \phi(\mathsf{k}_j)$. We can rewrite the previous equation as:

$$\mathsf{h}_i = \frac{\phi(\mathsf{q}_i)^\top \mathsf{S}_i}{\phi(\mathsf{q}_i)^\top \mathsf{Z}_i}\,. \tag{12}$$

Since $\mathsf{S}_i = \mathsf{S}_{i-1} + \phi(\mathsf{k}_i)\mathsf{v}_i^\top$ and $\mathsf{Z}_i = \mathsf{Z}_{i-1} + \phi(\mathsf{k}_i)$, each new generated token requires a *constant* amount of time, making this recurrent formulation attractive for the generation of long sequences.

Memory considerations

Materializing the $\mathbf{QK}^\top$ matrix also requires $\mathcal{O}(n^2)$ memory, which is the main bottleneck in existing GPUs and similar hardware.

Modern implementations (e.g., **FlashAttention**[6]) can avoid this by processing tokens in multiple chunks. This can be done by storing intermediate values on the denominator of the softmax, and only applying the normalization at the end (**lazy softmax**).

---

[6] `https://github.com/Dao-AILab/flash-attention`

Practical transformer models

Text Transformers

Transformers for contextual embeddings

The majority of pre-trained word embedding models we discussed above are standard transformer models trained on the sequence of text tokens.

- **BERT-like** models are pre-trained by masking one word in a sentence, and reconstructing the full sentence in output.
- **GPT-like** models are (causal) variants pre-trained to generate the sequence auto-regressively.

These models are called **contextual** embeddings because the same word in different sentences can be encoded to different vectorial representations.

Qiu, X., et al., 2020. **Pre-trained models for natural language processing: A survey**. *Science China Technological Sciences*, pp. 1-26.

Because these models are trained from the raw text alone (no specific targets) they are called **self-supervised** models (we will cover this more in-depth later).

Their strengths is that scaling laws for transformers are empirically better than for other models (i.e., they benefit more from increasing the dataset by order of magnitude).

In natural language processing, this is also shown by the emergence of paradigms like **text-prompting** and **zero-shot learning**.

Foundation models

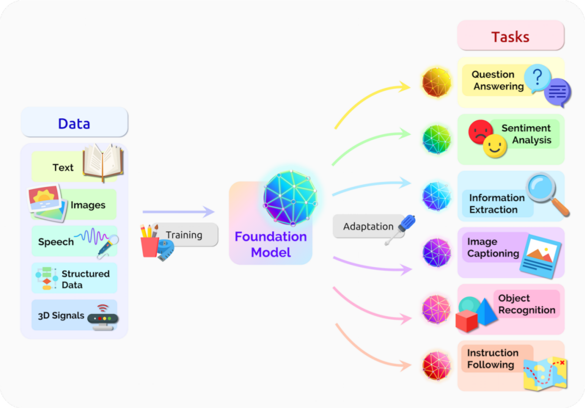


Fig. 2. A foundation model can centralize the information from all the data from various modalities. This one model can then be adapted to a wide range of downstream tasks.

Figure 13: An emerging name for these huge, pre-trained models is **foundation models**.

Practical transformer models

Vision, Audio, & Graph Transformers

Applying transformers on images

One important realization of the last two years is that transformers can also benefit computer vision, especially when trained on huge datasets (e.g., ImageNet21k).

However, this requires to convert the original image (a 2D grid) into a 1D sequence (actually, a set together with the positional embeddings). Because this would scale quadratically in the number of pixels, a common solution is to work on **patches** of the original image.

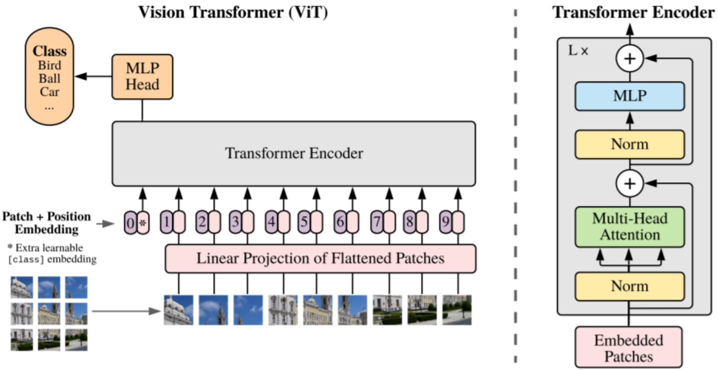


Figure 14: The Vision Transformer (ViT) is a standard transformer applied on top of image patches.

Dosovitskiy, A., et al., 2020. **An image is worth 16x16 words: Transformers for image recognition at scale.** *arXiv preprint arXiv:2010.11929.*

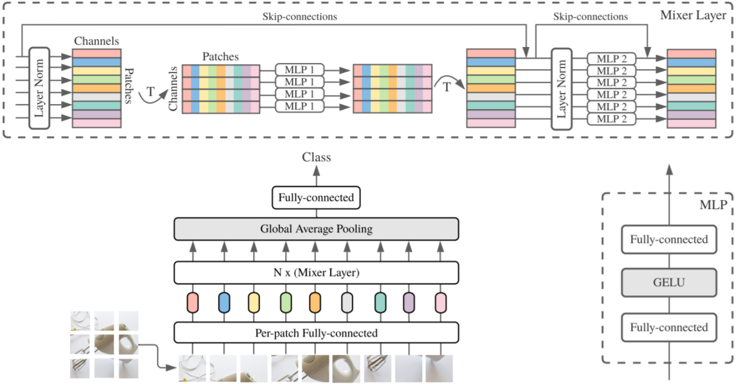


Figure 15: Mixer models are variants of the ViT, where the MHA is replaced by fully-connected layers.

Tolstikhin, I., et al., 2021. **MLP-Mixer: An all-MLP architecture for vision**. *arXiv preprint arXiv:2105.01601*.

Audio transformers

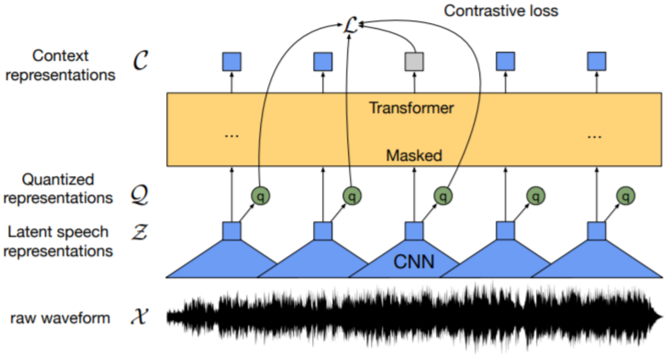


Figure 16: Architectures like Wav2Vec 2.0 are pre-trained audio models exploiting transformers However, this is harder because of the nature of the audio signal.

Baevski, A., Zhou, H., Mohamed, A. and Auli, M., 2020. **wav2vec 2.0: A framework for self-supervised learning of speech representations**. *arXiv preprint arXiv:2006.11477.*

Graph transformers

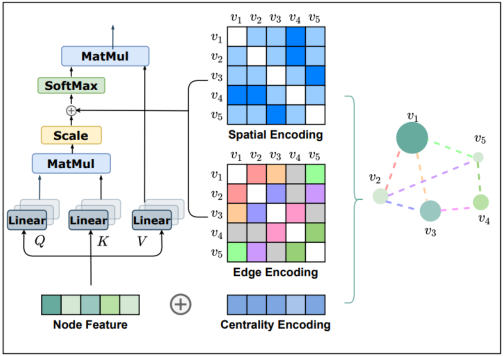


Figure 1: An illustration of our proposed centrality encoding, spatial encoding, and edge encoding in Graphormer.

Figure 17: We can also design graph transformers, where nodes become tokens and the connectivity is embedded inside the positional embeddings.

Further readings

- **D2L**: Chapter 11;[7] **UDL**: Chapter 12.
- https://jalammar.github.io/illustrated-transformer/.
- https://srush.github.io/raspy/ for intuitions about transformers can work.

---

[7] In addition, you can check out Chapter 15 on pre-training NLP models.