# Deep Learning - Foundations and Concepts

## Chapter 12. Transformers

nonlineark@github

March 20, 2025

# Outline

1. Attention

2. Natural Language

3. Transformer Language Models

# Attention

The fundamental concept that underpins a transformer is attention:

- This was originally developed as an enhancement to RNNs for machine translation (Bahdanau, Cho and Bengio, 2014)
- Later, it was found that significantly improved performance could be obtained by eliminating the recurrence structure and instead focusing exclusively on the attention mechanism (Vaswani et al., 2017).

# Attention

Consider the following two sentences:

- I swam across the river to get to the other bank.
- I walked across the road to get cash from the bank.

Here the word “bank” has different meanings in the two sentences:

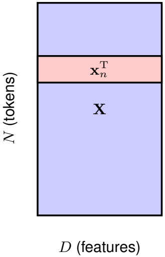
- In the first sentence, the words “swam” and “river” most strongly indicate that “bank” refers to the side of a river.
- In the second sentence, the word “cash” is a strong indicator that “bank” refers to a financial institution.

To determine the appropriate interpretation of “bank”, a neural network processing such a sentence should:

- Attend to specific words from the rest of the sequence.
- The particular locations that should receive more attention depend on the input sequence itself.

# Transformer processing

- The input data to a transformer is a set of vectors $\{x_n\}$ of dimensionality $D$, where $n = 1, \ldots, N$.
- We refer to these data vectors as tokens, and the elements of the tokens are called features.
- We will combine the tokens into a matrix $X$ of dimension $N \times D$ in which the $n$th row comprises the token $x_n^T$.



# Transformer processing

The fundamental building block of a transformer is a function that takes a data matrix $X$ as input and creates a transformed matrix $\tilde{X}$ of the same dimensionality as the output:

$$\tilde{X} = \mathrm{TransformerLayer}(X)$$

A single transformer layer itself comprises two stages:

- The first stage, which implements the attention mechanism, mixes together the corresponding features from different tokens across the columns of the data matrix.
- The second stage acts on each row independently and transforms the features within each token.

# Attention coefficients

Suppose we have a set of input tokens $x_1, \dots, x_N$ and we want to map this set to another set $y_1, \dots, y_N$:

- $y_n$ should depend on all the tokens $x_1, \dots, x_N$.
- This dependence should be stronger for those tokens $x_m$ that are particularly important for determining the modified representation of $y_n$.

A simple way to achieve this is to define each output token $y_n$ to be a linear combination of the input tokens:

$$y_n = \sum_{m=1}^{N} a_{nm} x_m$$

$$a_{nm} \geq 0 \qquad \sum_{m=1}^{N} a_{nm} = 1$$

# Self-attention

The problem of determining the attention coefficients can be viewed from an information retrieval perspective:

- We could view the vector $x_n$ as:
  - The key for input token $n$.
  - The value for input token $n$.
  - The query for output token $n$.
- To measure the similarity between the query $x_n$ and the key $x_m$, we could use their dot product: $x_n^T x_m$.
- To make sure the attention coefficients define a partition of unity, we could use the $\mathrm{softmax}$ function to transform the dot products.

# Self-attention

Dot-product self-attention:

$$y_n = \sum_{m=1}^{N} a_{nm} x_m$$

$$a_{nm} = \frac{\exp(x_n^T x_m)}{\sum_{m'=1}^{N} \exp(x_n^T x_{m'})}$$

Or write in matrix notation:

$$Y = \mathrm{softmax}(X X^T) X$$

where $\mathrm{softmax}(L)$ means to apply $\mathrm{softmax}$ to each row of the matrix $L$.

# Network parameters

The current transformation from input tokens $\{x_n\}$ to output tokens $\{y_n\}$ has major limitations:

- The transformation is fixed and has no capacity to learn from data because it has no adjustable parameters.
- Each of the feature values within a token $x_n$ plays an equal role in determining the attention coefficients.

# Network parameters

We can overcome these limitations by defining separate query, key and value matrices each having their own independent linear transformations:

- $Q = XW^{(q)}$, where $W^{(q)}$ has dimensionality $N \times D_k$.
- $K = XW^{(k)}$, where $W^{(k)}$ has dimensionality $N \times D_k$.
- $V = XW^{(v)}$, where $W^{(v)}$ has dimensionality $N \times D_v$.
- A typical choice is $D_k = D$.
- If we set $D_v = D$:
  - This will facilitate the inclusion of residual connections.
  - Multiple transformer layers can be stacked on top of each other.

The dot-product self-attention now takes the form:

$$Y = \mathrm{softmax}(QK^T)V$$

# Scaled self-attention

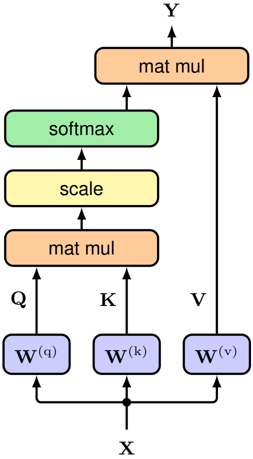
- When logits are too large, the $\mathrm{softmax}$ function will produce extremely small gradients, which is not desirable.
- We need to scale the logits before applying the $\mathrm{softmax}$ function.
- Notice that if $q, k \in \mathbb{R}^{D_k}$ and the elements of $q$ and $k$ are all independent random numbers with zero mean and unit variance, then $\mathrm{var}(q^T k) = D_k$. Thus it would be appropriate to scale the logits by the standard deviation $\sqrt{D_k}$.

The scaled dot-product self-attention now takes the form:

$$Y = \mathrm{Attention}(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{D_k}})V$$

# Scaled self-attention

Figure: Information flow in a scaled dot-product self-attention neural network layer



# Scaled self-attention

**Algorithm 1:** Scaled dot-product self-attention

```
$Q \leftarrow XW^{(q)}$;
$K \leftarrow XW^{(k)}$;
$V \leftarrow XW^{(v)}$;
return $\mathrm{Attention}(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{D_k}})V$;
```

# Multi-head attention

We can use multiple attention heads in parallel to attend to multiple data-dependent patterns at the same time. Suppose we have $C$ heads:

$$\begin{aligned} H_c &= \mathrm{Attention}(Q_c, K_c, V_c) \\ &= \mathrm{Attention}(XW_c^{(q)}, XW_c^{(k)}, XW_c^{(v)}) \end{aligned}$$

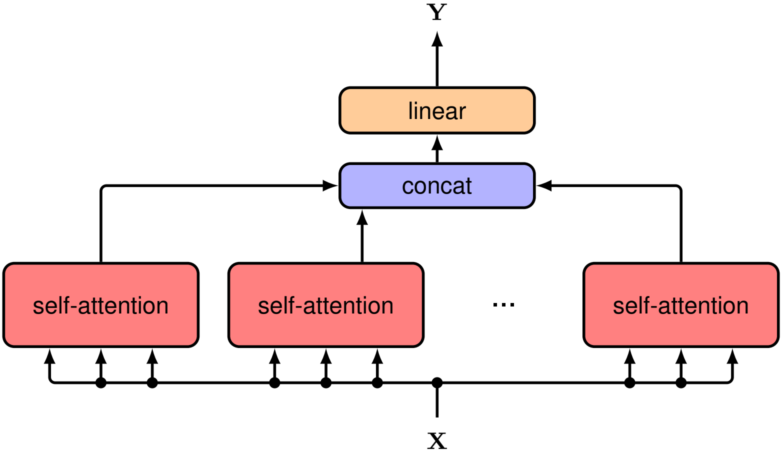
The heads are first concatenated into a single matrix, and the result is then linearly transformed using a matrix $W^{(o)}$ to give a combined output:

$$Y = (H_1, \ldots, H_C)W^{(o)}$$

The matrix $W^{(o)}$ has dimension $HD_v \times D$, so that the final output matrix $Y$ has dimension $N \times D$.

# Multi-head attention

Figure: Information flow in a multi-head attention layer



# Multi-head attention

**Algorithm 2:** Multi-head attention

```
for $c \leftarrow 1$ to $C$ do
    $Q_c = XW_c^{(q)}$;
    $K_c = XW_c^{(k)}$;
    $V_c = XW_c^{(v)}$;
    $H_c = \mathrm{Attention}(Q_c, K_c, V_c)$;
end
$H = (H_1, \ldots, H_C)$;
return $Y = HW^{(o)}$;
```

# Transformer layers

To improve training efficiency, we can introduce residual connections and layer normalization:

$$Z = \mathrm{LayerNorm}(Y(X) + X)$$

Or using pre-norm:

$$Z = Y(\mathrm{LayerNorm}(X)) + X$$

Until now, the output data matrix $Y$ is still a linear transformation of the input data matrix $X$, and this limits the expressive capabilities of the attention layer. We can enhance the flexibility by post-processing the outputs using a standard nonlinear neural network denoted $\mathrm{MLP}$:

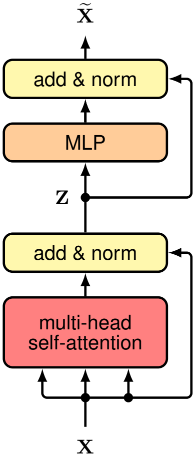
$$\tilde{X} = \mathrm{LayerNorm}(\mathrm{MLP}(Z) + Z)$$

Again, we can use a pre-norm instead:

$$\tilde{X} = \mathrm{MLP}(\mathrm{LayerNorm}(Z)) + Z$$

# Transformer layers

Figure: One layer of the transformer architecture



## Computational complexity

- In the attention layer:
  - Calculate the matrices $Q$, $K$ and $V$: $\mathcal{O}(ND^2)$.
  - Calculate the dot products $QK^T$: $\mathcal{O}(N^2D)$.
  - Calculate the matrix $Y$: $\mathcal{O}(N^2D)$.
- In the neural network layer:
  - Calculate the matrix $\tilde{X}$: $\mathcal{O}(ND^2)$.

# Positional encoding

- A transformer is equivariant with respect to input permutations due to the shared matrices $W_c^{(q)}$, $W_c^{(k)}$, $W_c^{(v)}$ and the shared subsequent neural network.
- The lack of dependence on token order becomes a major limitation when we consider sequential data, so we need to find a way to inject token order information into the network.

# Positional encoding

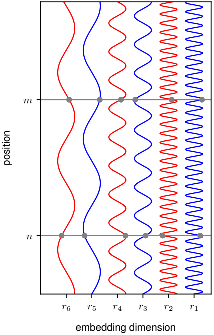
The requirements for a positional encoding:

- The token order should be encoded in the data itself instead of having to be represented in the network architecture.
- We will construct a position encoding vector $r_n$ associated with each input position $n$ and then combine this with the associated input token $x_n$:
  - [No] $\tilde{x}_n = (x_n, r_n)$.
  - [Yes] $\tilde{x}_n = x_n + r_n$.
- $r_n$ should be bounded.
- $r_n$ should generalize well to new input sequences that are longer than those used in training.
- $r_n$ should be unique for a given position.
- $r_n$ should have a consistent way to express the number of steps between any two input tokens irrespective of their absolute position.

# Positional encoding

Positional encoding based on sinusoidal functions:

$$r_{ni} = \begin{cases} \sin(\frac{n}{L^{\frac{i}{D}}}), \text{if } i \text{ is even} \\ \cos(\frac{n}{L^{\frac{i-1}{D}}}), \text{if } i \text{ is odd} \end{cases}$$



# Word embedding

To convert the words into a numerical representation that is suitable for use as the input to a deep neural network:

- [No] One simple approach is to define a fixed dictionary of words, and use a one hot representation for each word.
- The embedding process can be defined by a matrix $E$ of size $D \times K$ where $D$ is the dimensionality of the embedding space and $K$ is the dimensionality of the dictionary. Each column of $E$ represents the embedding of a word.

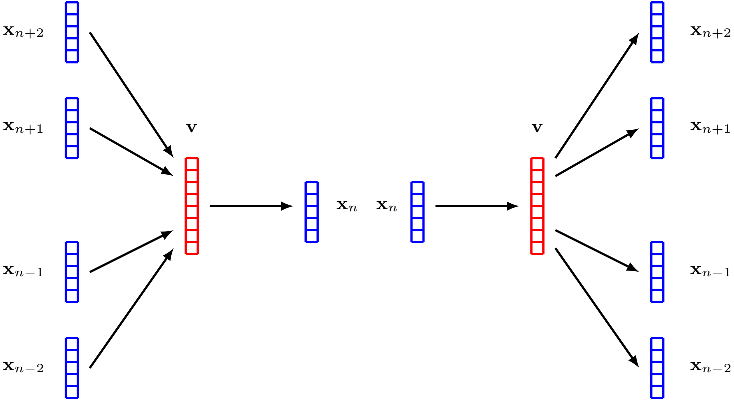
# Word embedding

The word2vec technique:

- A training set is constructed in which each sample is obtained by considering a window of $M$ adjacent words in the text, where a typical value might be $M = 5$.
- The error function is defined as the sum of the error functions for each sample.
- In continuous bag of words, the target variable for network training is the middle word, and the remaining context words form the inputs.
  - Once trained, $E$ is given by the transpose of the second-layer weight matrix.
- In skip-grams, the center word is presented as the input and the target values are the context words.
  - Once trained, $E$ is given by the first-layer weight matrix.

# Word embedding

Figure: Two-layer neural networks used to learn word embeddings



# Word embedding

Words that are semantically related are mapped to nearby positions in the embedding space.

The learned embedding space often has an even richer semantic structure than just the proximity of related words, and that this allows for simple vector arithmetic:

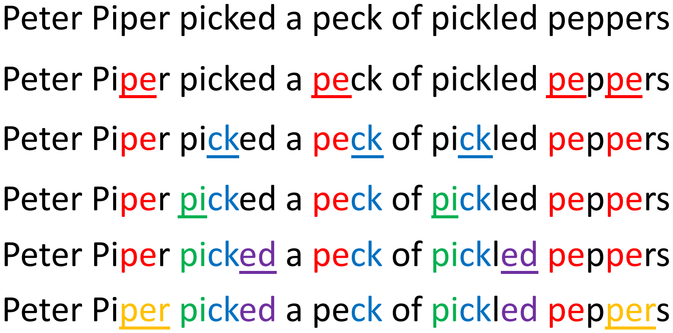
$$v(\text{Paris}) - v(\text{France}) \approx v(\text{Rome}) - v(\text{Italy})$$

# Tokenization

- Limitations in word-level representation:
  - Words not in the dictionary.
  - Misspelled words.
  - Punctuation symbols or other character sequences such as computer code.
- Limitations in character-level representation:
  - The semantically important word structure of language is discarded.
  - Requires a much larger number of sequential steps for a given body of text, thereby increasing the computational cost of processing the sequence.
- We can combine the benefits of character-level and word-level representations by using a pre-processing step that converts a string of words and punctuation symbols into a string of tokens.

# Tokenization

Figure: Tokenizing natural language by analogy with byte pair encoding



Peter Piper picked a peck of pickled peppers

Peter Piper picked a peck of pickled peppers

Peter Piper picked a peck of pickled peppers

Peter Piper picked a peck of pickled peppers

Peter Piper picked a peck of pickled peppers

Peter Piper picked a peck of pickled peppers

# Bag of words

The bag-of-words model assumes that the words are drawn independently from the same distribution and hence that the joint distribution is fully factorized in the form:

$$p(x_1, \dots, x_N) = \prod_{n=1}^{N} p(x_n)$$

The bag-or-words model completely ignores the ordering of the words.

# Autoregressive models

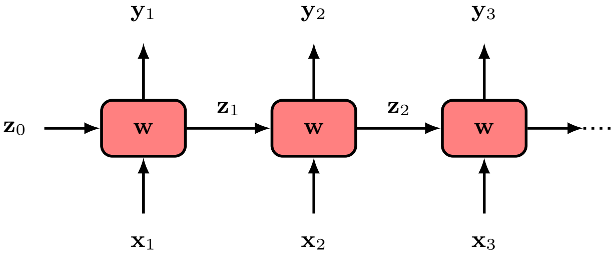
The autoregressive approach assumes that each of the conditional distributions is independent of all previous observations except the $L$ most recent words. For example, for $L = 2$:

$$p(x_1, \ldots, x_N) = p(x_1)p(x_2|x_1)\prod_{n=3}^{N} p(x_n|x_{n-1}, x_{n-2})$$

The case with $L = 1$ is known as a bi-gram model; when $L = 2$, it is called a tri-gram model; and in general these are called n-gram models.

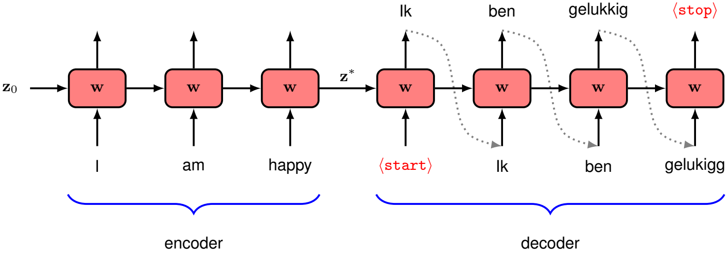
# Recurrent neural networks

Figure: A general RNN with parameters $w$



# Recurrent neural networks

Figure: An example of a recurrent neural network used for language translation



## Backpropagation through time

- RNNs can be trained by stochastic gradient descent. The error function consists of a sum over all output units of the error for each unit, in which each output unit has a $\mathrm{softmax}$ activation function along with an associated cross-entropy error function.
  - In practice, for very long sequences, training can be difficult due to the problems of vanishing gradients or exploding gradients.
- Bottleneck problem: RNNs deal poorly with long-range dependencies. For the translation task, the entire concept of the English sentence must be captured in the single hidden vector $z^*$ of fixed length, which becomes increasingly problematic with longer sequences.
- RNNs do not support parallel computation within a single training example due to the sequential nature of the processing.

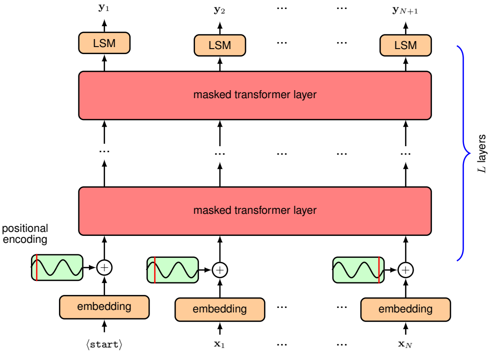
# Transformer language models

Transformers can be applied to many different kinds of language processing task, and can be grouped into three categories:

- Encoder. For example, sentiment analysis, in which we take a sequence of words as input and provide a single variable representing the sentiment of the text.
- Decoder. For example, image caption generation, in which we take a single image as input and generate a word sequence as output.
- Sequence-to-sequence. For example, machine translation, in which both the input and output comprise a sequence of words.

# Decoder transformers

Figure: Architecture of a GPT decoder transformer network



# Decoder transformers

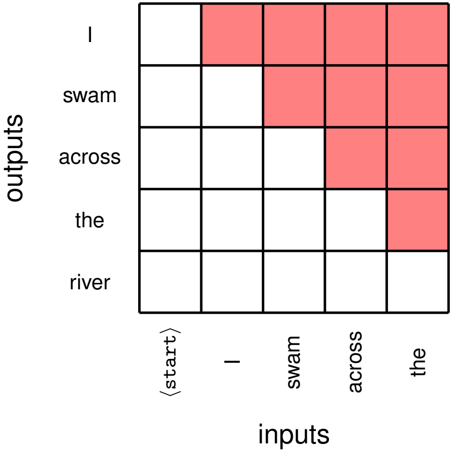
- The stack of transformer layers take a sequence $x_1, \ldots, x_N$ of tokens, each of dimensionality $D$, as input and produce a sequence $\tilde{x}_1, \ldots, \tilde{x}_N$ of tokens, again of dimensionality $D$, as output.
- We make a linear transformation of each output token using a matrix $W^{(p)}$ of dimensionality $D \times K$ followed by a $\mathrm{softmax}$ activation function so that each output represents a probability distribution over the dictionary of tokens: $Y = \mathrm{softmax}(\tilde{X} W^{(p)})$.
- Each $\mathrm{softmax}$ output unit has an associated cross-entropy error function, and the error function used for training is the sum of the cross-entropy error values summed over the training set.

# Decoder transformers

- We can achieve much greater efficiency by processing an entire sequence at once so that each token acts both as a target value for the sequence of previous tokens and as an input value for subsequent tokens.
- However, we have to ensure that the network is not able to cheat by looking ahead in the sequence:
  - We shift the input sequence to the right by one step.
  - We introduce masked attention into each of the attention layers, in which we set to zero all of the attention weights that correspond to a token attending to any later token in the sequence.

# Decoder transformers

Figure: An illustration of the mask matrix for masked self-attention



| outputs \ inputs | ⟨start⟩ | I | swam | across | the |
|---|---|---|---|---|---|
| I | | 🟥 | 🟥 | 🟥 | 🟥 |
| swam | | | 🟥 | 🟥 | 🟥 |
| across | | | | 🟥 | 🟥 |
| the | | | | | 🟥 |
| river | | | | | |

## Decoder transformers

- To make efficient use of the massive parallelism of GPUs, multiple sequences may be stacked together into an input tensor for parallel processing in a single batch.
- The sequences are padded to the same length by introducing a specific token `<pad>`.
- An additional mask is then used in the attention weights to ensure that the outputs do not pay attention to any inputs occupied by the `<pad>` token.

# Sampling strategies

There are several options for selecting the value of the token based on the computed probabilities:

- One extreme is greedy search, which simply selects the token with the highest probability.
  - Deterministic, $\mathcal{O}(KN)$ cost, far from optimal.
- The other extreme is exhaustive search, which enumerates all the possible output sequences and then outputs the one that scores the highest predicted probability.
  - Infeasible given the total number of sequences is $K^N$.

# Sampling strategies

Beam search:

- At each step, we maintain a set of $B$ hypotheses, where $B$ is called the beam width.
- We then feed all these sequences through the network, and for each sequence we find the $B$ most probable token values, thereby creating $B^2$ possible hypotheses for the extended sequence.
- This list is then pruned by selecting the most probable $B$ hypotheses.
- The sequence probabilities are generally normalized by the corresponding lengths of the sequence before making comparisons to prevent bias towards short sequences.
- Beam search has cost $\mathcal{O}(BKN)$.

# Sampling strategies

- One problem with approaches such as greedy search and beam search is that they limit the diversity of potential outputs.
- Top-$K$ sampling: Consider only the states having the top $K$ probabilities, for some choice of $K$, and then sample from these according to their renormalized probabilities.
- Top-$p$ sampling: Calculates the cumulative probability of the top outputs until a threshold is reached and then samples from this restricted set of token states.

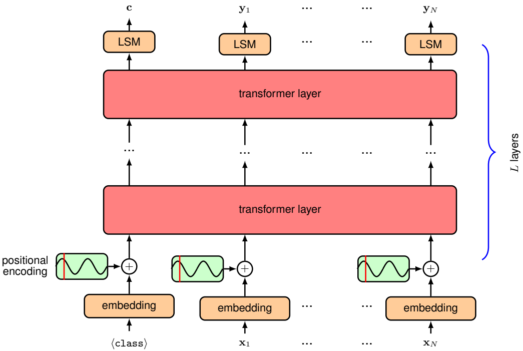
# Sampling strategies

A softer version of top-$K$ sampling is to introduce a parameter $T$ called temperature into the definition of the $\mathrm{softmax}$ function: $y_i = \frac{\exp(\frac{a_i}{T})}{\sum_j \exp(\frac{a_j}{T})}$.

- When $T \to 0$, the probability mass is concentrated on the most probable state.
- For $T = 1$, we recover the unmodified $\mathrm{softmax}$ distribution.
- For $T \to \infty$, the distribution becomes uniform across all states.
- By choosing a value in the range $0 < T < 1$, the probability is concentrated towards the higher values.

# Encoder transformers

Figure: Architecture of an encoder transformer model



# Encoder transformers

- The first token of every input string is given by a special token `<class>`, and the corresponding output of the model is ignored during pre-training.
- A randomly chosen subset of the input tokens, say $15\%$, are replaced with a special token denoted `<mask>`. The model is trained to predict the missing tokens at the corresponding output nodes.

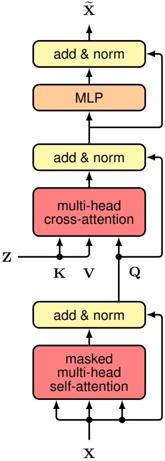
# Encoder transformers

Once the encoder model is trained it can then be fine-tuned for a variety of different tasks:

- For a text classification task, only the first output position is used, which corresponds to the `<class>` token that always appears in the first position of the input sequence.
- If the goal is to classify each token of the input string, then the first output is ignored and the subsequent outputs have a shared linear-plus-softmax layer.
- Alternatively the output of a pre-trained model might feed into a sophisticated generative deep learning model for applications such as text-to-image synthesis.

# Sequence-to-sequence transformers

Figure: Schematic illustration of one cross-attention layer



# Sequence-to-sequence transformers

- An encoder transformer can be used to map the input token sequence into a suitable internal representation $Z$.
- Cross attention: The query vectors come from the sequence being generated, the key and value vectors come from the sequence represented by $Z$.

# Sequence-to-sequence transformers

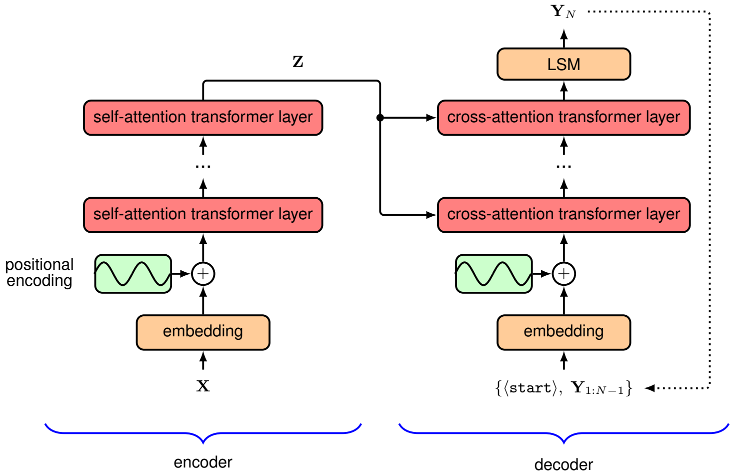

$\mathbf{Y}_N$
LSM
$\mathbf{Z}$
self-attention transformer layer
cross-attention transformer layer
...
...
self-attention transformer layer
cross-attention transformer layer
positional encoding
embedding
embedding
$\mathbf{X}$
$\{\langle\texttt{start}\rangle,\ \mathbf{Y}_{1:N-1}\}$
encoder
decoder

Figure: Schematic illustration of a sequence-to-sequence transformer

# Large language models

- The use of self-supervised learning led to a paradigm shift in which a large model is first pre-trained using unlabelled data and then subsequently fine-tuned using supervised learning based on a much smaller set of labelled data.
- Fine-tuning can be done by adding extra layers to the outputs of the network or by replacing the last few layers with fresh parameters and then using the labelled data to train these final layers.

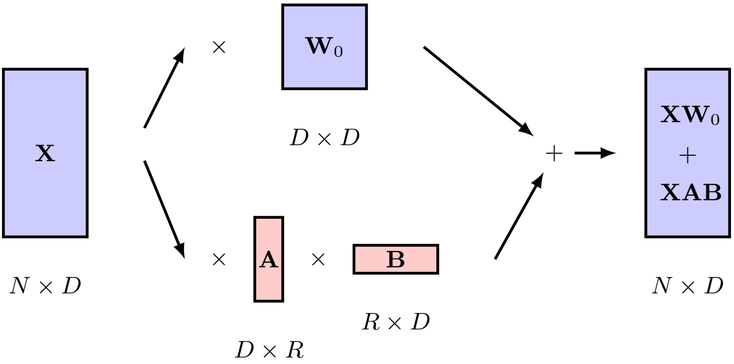
# Large language models

Low-rank adaptation (LoRA):

- A trained over-parameterized model has a low intrinsic dimensionality with respect to fine-tuning.
- LoRA freezes the weights of the original model and adds additional learnable weight matrices into each layer of the transformer in the form of low-rank products.
- Typically only attention-layer weights are modified, whereas MLP-layer weights are kept fixed.

# Large language models

Figure: Schematic illustration of low-rank adaptation



## Large language models

- As language models have become larger and more powerful, the need for fine-tuning has diminished, with generative language models now able to solve a broad range of tasks simply through text-based interaction.
- Reinforcement learning through human feedback (RLHF): Fune-tuning large language models through human evaluation of generated output.
- The performance of the model now depends on the form of the prompt, leading to a new field called prompt engineering.
- Few-shot learning: By providing some examples within the prompt, the model is able to solve new tasks.