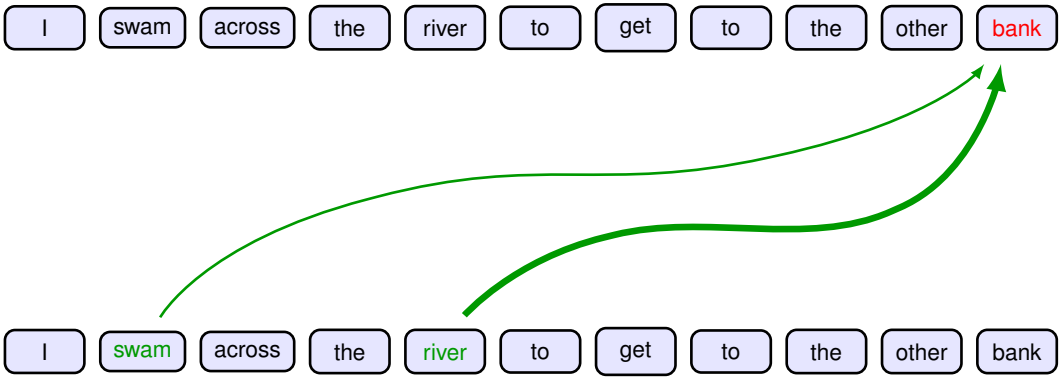
Transformers

Dr. Alejandro Veloz

Attention

The fundamental concept that underlies a transformer is **attention.**



Attention

I swam across the river to get to the other bank.

I walked across the road to get cash from the bank.


The
Law
will
never
be
perfect
,
but
its
application
should
be
just
-
this
is
what
we
are
missing
,
in
my
opinion
.
<EOS>
<pad>
The
Law
will
never
be
perfect
,
but
its
application
should
be
just
-
this
is
what
we
are
missing
,
in
my
opinion
.
<EOS>
<pad>

Transformer processing

The input data to a transformer is a set of vectors $\{\mathbf{x}_n\}$ of dimensionality D, where $n = 1, \ldots, N$.

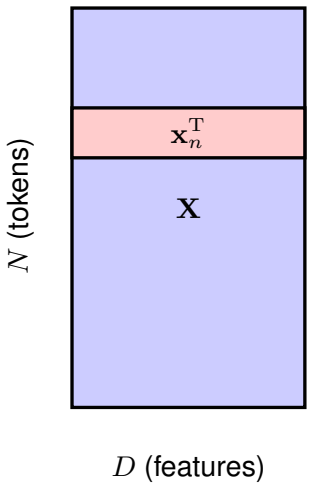
We refer to these data vectors as **tokens**:

- word within a sentence,
- a patch within an image, or
- an amino acid within a protein.

The elements x_{ni} of the tokens are called **features**.

Transformers handle heterogeneous data by merging variables into a unified token set, eliminating the need for specialized architectures.

Notation



Tokens are stored in $\mathbf{X}$ of dimensions $N \times D$.

The n-th row comprises the token vector $\mathbf{x}_n^{\mathrm{T}}$, and where $n = 1, \ldots, N$ labels the rows.

For most applications, we will require a data set containing many sets of tokens, such as independent passages of text where each word is represented as one token.

Transformer Layer

A transformer layer maps an input matrix $\mathbf{X}$ to an output matrix $\widetilde{\mathbf{X}}$ of the same size:

$$\widetilde{\mathbf{X}} = \mathsf{TransformerLayer}[\mathbf{X}].$$

Stacking multiple layers yields deep networks that learn complex representations through trainable parameters optimized by gradient descent.

Each layer consists of **two stages**:

- An attention mechanism that mixes information (features) across tokens.
- A feed-forward stage that transforms features within each token.

Attention

Let $\mathbf{x}_1, \dots, \mathbf{x}_N$ be input tokens in an embedding space.

We aim to map them to a modified representation $\mathbf{y}_1, \dots, \mathbf{y}_N$, a set of tokens in a new embedding space with richer semantic structure.

- One particular value $\mathbf{y}_n$ depends on all the inputs vectors $\mathbf{x}_1, \dots, \mathbf{x}_N$.
- This dependence is stronger for inputs $\mathbf{x}_m$ that are particularly important for determining $\mathbf{y}_n$.

Attention

A simple way to transform tokens $\mathbf{x}_1, \dots, \mathbf{x}_N$ into $\mathbf{y}_1, \dots, \mathbf{y}_N$ is the linear mapping:

$$\mathbf{y}_n = \sum_{m=1}^{N} a_{nm} \mathbf{x}_m$$

where a_{nm} are called **attention weights**.

Attention weights considerations

1. The coefficients should be **close to zero** for input tokens that have **little influence** on the output $\mathbf{y}_n$ and largest for inputs that have most influence.
2. We therefore constrain the coefficients to be **non-negative** to avoid situations in which one coefficient can become large and positive while another coefficient compensates by becoming large and negative.
3. We also want to ensure that if an output pays more attention to a particular input, this will be at the expense of paying less attention to the other inputs, and so we constrain the coefficients to sum to unity.

Attention weights contraints

The weighting coefficients must satisfy the following two constraints:

$$a_{nm} \geqslant 0$$

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

- Together these imply that each coefficient lies in the range $0 \leqslant a_{nm} \leqslant 1$ and so the coefficients define a **partition of unity**.
- If $a_{mm} = 1$, it follows that $a_{nm} = 0$ for $n \neq m$, and therefore $\mathbf{y}_m = \mathbf{x}_m$ (the input vector is unchanged).
- More generally, the output $\mathbf{y}_m$ is a blend of the input vectors with some inputs given more weight than others.

Self-attention

$$a_{nm} = \frac{\exp\left(\mathbf{x}_n^\mathsf{T}\mathbf{x}_m\right)}{\sum_{m'=1}^{N}\exp\left(\mathbf{x}_n^\mathsf{T}\mathbf{x}_{m'}\right)}$$

In matrix notation:

$$\mathbf{Y} = \text{Softmax}\left[\mathbf{X}\mathbf{X}^\mathsf{T}\right]\mathbf{X}$$

where $\text{Softmax}[\mathbf{L}]$ is an operator that takes the exponential of every element of a matrix $\mathbf{L}$ and then **normalizes each row independently** to sum to one.

Introducing network parameters

Self-attention – no learnable parameters.

- Features within a token $\mathbf{x}_n$ plays an equal role in determining the attention coefficients.
- We would like the network to have the flexibility to focus more on some features than others when determining token similarity.

We can address both issues if we define modified feature vectors given by the linear transformation:

$$\widetilde{\mathbf{X}} = \mathbf{X}\mathbf{U}$$

where $\mathbf{U}$ is a $D \times D$ matrix of **learnable weight parameters** (analogous to a `layer` in a standard neural network).

Introducing network parameters

This gives a modified transformation:

$$\mathbf{Y} = \text{Softmax}\left[\mathbf{X}\mathbf{U}\mathbf{U}^{\mathsf{T}}\mathbf{X}^{\mathsf{T}}\right]\mathbf{X}\mathbf{U}$$

Although this has much more flexibility, the matrix:

$$\mathbf{X}\mathbf{U}\mathbf{U}^{\mathsf{T}}\mathbf{X}^{\mathsf{T}}$$

is symmetric.

E.g. ‘cat’ is equally associated with ‘pet’, as ‘pet’ with ‘cat’.

The softmax function is the sole mechanism responsible for introducing asymmetry into the attention weight matrix.

Introducing network parameters

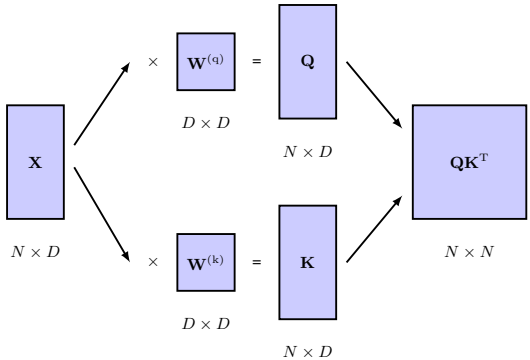
To support asymmetric attention (e.g. cat $\longrightarrow$ pet strong, pet $\longrightarrow$ cat less strong), we can allow the query, key, and value matrices each having their own independent linear transformations.

$$
\begin{aligned}
\mathbf{Y} &= \text{Softmax}\left[\mathbf{X}\mathbf{U}\ \mathbf{U}^{\mathsf{T}}\mathbf{X}^{\mathsf{T}}\right]\mathbf{X}\mathbf{U} \\
&= \text{Softmax}\left[\mathbf{X}\mathbf{W}^{(\mathsf{q})}\ \left(\mathbf{X}\mathbf{W}^{(\mathsf{k})}\right)^{\mathsf{T}}\right]\mathbf{X}\mathbf{W}^{(\mathsf{v})} \\
&= \text{Softmax}\left[\mathbf{Q}\ \mathbf{K}^{\mathsf{T}}\right]\mathbf{V}
\end{aligned}
$$

- The weight matrices $\mathbf{W}^{(\mathsf{q})}$, $\mathbf{W}^{(\mathsf{k})}$ and $\mathbf{W}^{(\mathsf{v})}$ represent learnable parameters.
- $\mathbf{W}^{(\mathsf{q})}$, $\mathbf{W}^{(\mathsf{k})}$ are of dimension $D \times D_{\mathsf{k}}$, whereas $\mathbf{W}^{(\mathsf{v})}$ is of dimension $D \times D_{\mathsf{v}}$.
- Multiple transformer layers can be stacked on top of each other if each layer has the same dimensionality.

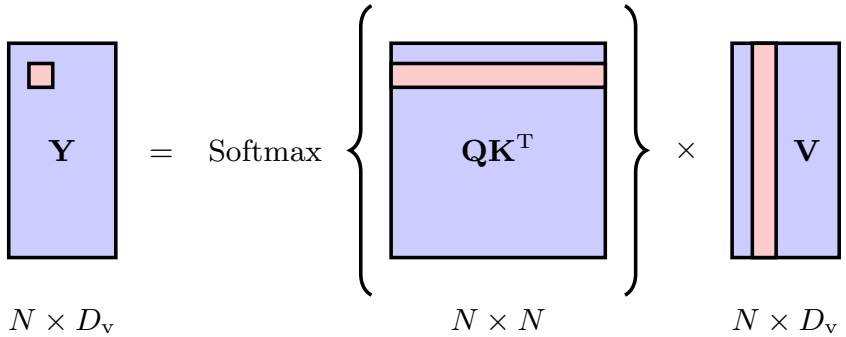
Attention layer

$$\mathbf{Y} = \text{Softmax}\left[\mathbf{X}\mathbf{W}^{(q)}\left(\mathbf{X}\mathbf{W}^{(k)}\right)^{T}\right]\mathbf{X}\mathbf{W}^{(v)} = \text{Softmax}\left[\mathbf{Q}\,\mathbf{K}^{T}\right]\mathbf{V}$$



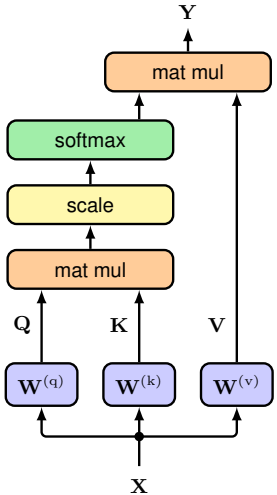
Attention layer

$$\mathbf{Y} = \text{Softmax}\left[\mathbf{X}\mathbf{W}^{(q)}\left(\mathbf{X}\mathbf{W}^{(k)}\right)^{T}\right]\mathbf{X}\mathbf{W}^{(v)} = \text{Softmax}\left[\mathbf{Q}\,\mathbf{K}^{T}\right]\mathbf{V}$$



Scaled dot-product self-attention layer

$$\mathbf{Y} = \text{Attention}\left(\mathbf{Q}, \mathbf{K}, \mathbf{V}\right)$$
$$= \text{Softmax}\left[\frac{\mathbf{Q}\,\mathbf{K}^{\mathsf{T}}}{\sqrt{D_{\text{k}}}}\right]\mathbf{V}$$



Scaled dot-product self-attention layer

Require: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$

Weight matrices $\left\{\mathbf{W}^{(\mathsf{q})}, \mathbf{W}^{(\mathsf{k})}\right\} \in \mathbb{R}^{D \times D_\mathsf{k}}$ and $\mathbf{W}^{(\mathsf{v})} \in \mathbb{R}^{D \times D_\mathsf{v}}$

Ensure: Attention $(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \in \mathbb{R}^{N \times D_\mathsf{v}} : \{\mathbf{y}_1, \ldots, \mathbf{y}_N\}$

1: $\mathbf{Q} = \mathbf{X}\mathbf{W}^{(\mathsf{q})}$ compute queries $\mathbf{Q} \in \mathbb{R}^{N \times D_\mathsf{k}}$

2: $\mathbf{K} = \mathbf{X}\mathbf{W}^{(\mathsf{k})}$ compute keys $\mathbf{K} \in \mathbb{R}^{N \times D_\mathsf{k}}$

3: $\mathbf{V} = \mathbf{X}\mathbf{W}^{(\mathsf{v})}$ compute values $\mathbf{V} \in \mathbb{R}^{N \times D}$

4: **return**

$$\text{Attention}\,(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left[\frac{\mathbf{Q}\mathbf{K}^\mathsf{T}}{\sqrt{D_\mathsf{k}}}\right]\mathbf{V}$$

Multi-Head Attention

- The attention layer allows the output vectors to attend to patterns of input vectors and is called an **attention head**.
- However, there might be multiple patterns of attention that are relevant at the same time.
- Using a single attention head **can lead to averaging over these effects**. Instead, we use multiple attention heads in parallel.
  - Identically structured copies of a single head, with independent learnable parameters for the query, key, and value matrices.
  - Analogous to using multiple different filters in each layer of a convolutional network.

Multi-Head Attention

Consider H heads indexed by $h = 1, \dots, H$:

$$\mathbf{H}_h = \text{Attention}\left(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h\right)$$

where $\text{Attention}(\cdot, \cdot, \cdot)$ is the scaled self-attention function, and each head has separate projection matrices:

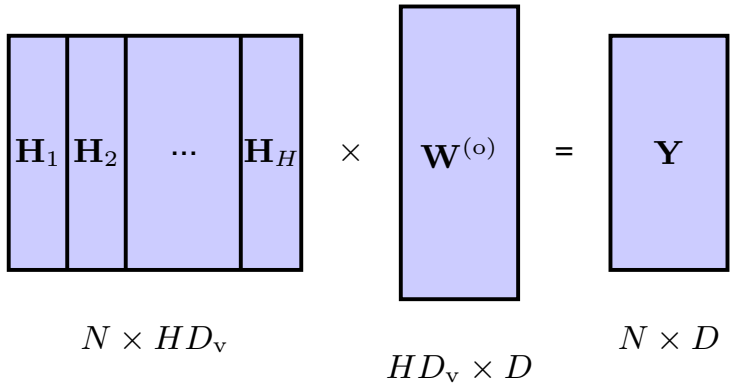
$$\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(\mathsf{q})}$$
$$\mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(\mathsf{k})}$$
$$\mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(\mathsf{v})}$$

Multi-Head Attention

The combined combined output is:

$$\mathbf{Y}(\mathbf{X}) = \text{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_H\right] \mathbf{W}^{(\text{o})}$$

The heads are first concatenated into a single matrix, and the result is then linearly transformed using a matrix $\mathbf{W}^{(\text{o})}$



Multi-Head Attention

Require: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$

Query weight matrices $\left\{\mathbf{W}_1^{(\mathsf{q})}, \ldots, \mathbf{W}_H^{(\mathsf{q})}\right\} \in \mathbb{R}^{D \times D}$

Key weight matrices $\left\{\mathbf{W}_1^{(\mathsf{k})}, \ldots, \mathbf{W}_H^{(\mathsf{k})}\right\} \in \mathbb{R}^{D \times D}$

Value weight matrices $\left\{\mathbf{W}_1^{(\mathsf{v})}, \ldots, \mathbf{W}_H^{(\mathsf{v})}\right\} \in \mathbb{R}^{D \times D_\mathsf{v}}$

Output weight matrix $\mathbf{W}^{(\mathsf{o})} \in \mathbb{R}^{H D_\mathsf{v} \times D}$

Ensure: $\mathbf{Y} \in \mathbb{R}^{N \times D} : \{\mathbf{y}_1, \ldots, \mathbf{x}_N\}$

// compute self-attention for each head

1: **for** $h = 1$ **to** H **do**

2: $\quad \mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(\mathsf{q})}, \quad \mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(\mathsf{k})}, \quad \mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(\mathsf{v})}$

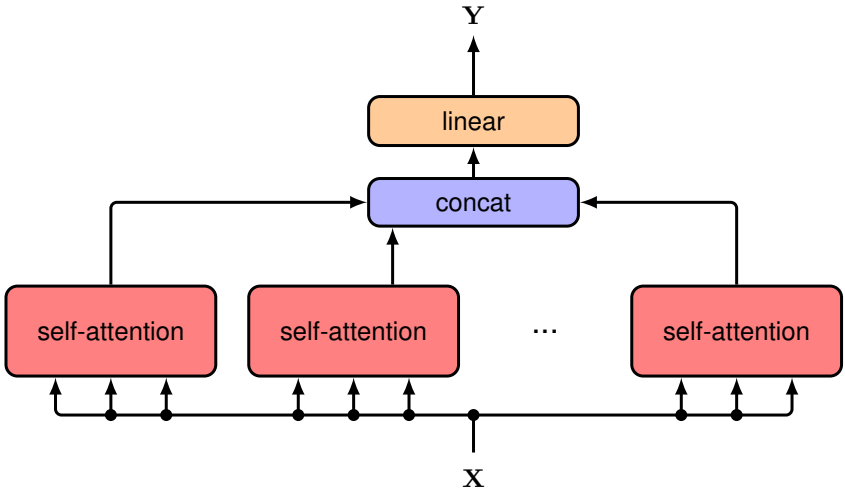
3: $\quad \mathbf{H}_h = \text{Attention}\left(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h\right)$ // $\mathbf{H}_h \in \mathbb{R}^{N \times D_\mathsf{v}}$

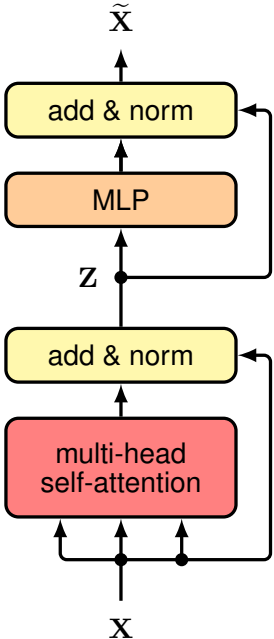
4: **end for**

5: $\mathbf{H} = \text{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_N\right]$ // concatenate heads

6: **return** $\mathbf{Y}(\mathbf{X}) = \mathbf{H}\mathbf{W}^{(\mathsf{o})}$

Multi-Head Attention





Transformer layer

$$\mathbf{Y}(\mathbf{X}) = \text{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_H\right] \mathbf{W}^{(o)}$$

$$\mathbf{Z} = \text{LayerNorm}\left[\mathbf{Y}(\mathbf{X}) + \mathbf{X}\right]$$

pre-norm

$$\mathbf{Z} = \mathbf{Y}\left(\mathbf{X}'\right) + \mathbf{X}, \quad \text{where} \quad \mathbf{X}' = \text{LayerNorm}\left[\mathbf{X}\right]$$

$$\widetilde{\mathbf{X}} = \text{LayerNorm}\left[\text{MLP}\left[\mathbf{Z}\right] + \mathbf{Z}\right]$$

pre-norm

$$\widetilde{\mathbf{X}} = \text{MLP}\left(\mathbf{Z}'\right) + \mathbf{Z}, \quad \text{where} \quad \mathbf{Z}' = \text{LayerNorm}\left[\mathbf{Z}\right]$$

In a typical transformer there are multiple such layers stacked on top of each other.

The layers generally have identical structures, although there is no sharing of weights and biases between different layers.

Transformer layer

Require: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$

Multi-head self-attention layer parameters

Feed-forward network parameters

Ensure: $\widetilde{\mathbf{X}} \in \mathbb{R}^{N \times D} : \{\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_N\}$

1: $\mathbf{Z} = \text{LayerNorm}[\mathbf{Y}(\mathbf{X}) + \mathbf{X}]$ // $\mathbf{Y}(\mathbf{X})$ from multi-head attention algorithm

2: $\widetilde{\mathbf{X}} = \text{LayerNorm}[\text{MLP}[\mathbf{Z}] + \mathbf{Z}]$ // shared neural network return $\tilde{\mathbf{X}}$

3: **return** $\widetilde{\mathbf{X}}$

Positional encoding

The matrices $\mathbf{W}_h^{(\mathsf{q})}$, $\mathbf{W}_h^{(\mathsf{k})}$, and $\mathbf{W}_h^{(\mathsf{v})}$ are shared across the input tokens.

A transformer is equivariant with respect to input permutations.

Permuting the order of the input tokens, i.e., the rows of $\mathbf{X}$, results in the same permutation of the rows of the output matrix $\tilde{\mathbf{X}}$.

The representation learned by a transformer will be independent of the input token ordering.

'The food was bad, not good at all' and 'The food was good, not bad at all' contain the same tokens but different meanings because of the different token ordering.

Positional encoding

The sharing of parameters in the network architecture facilitates the massively parallel processing of the transformer.

Also allows the network to learn long-range dependencies just as effectively as short-range dependencies.

However, the lack of dependence on token order becomes a major limitation when we consider sequential data (e.g. words in NLP).

Positional encoding

Strategy: to retain the powerful properties of the attention layers, the token order is encoded in the data itself instead of changing the network architecture.

Solution: Position encoding vector $\mathbf{r}_n$ associated with each input position n and then combine this with the associated input token embedding $\mathbf{x}_n$.

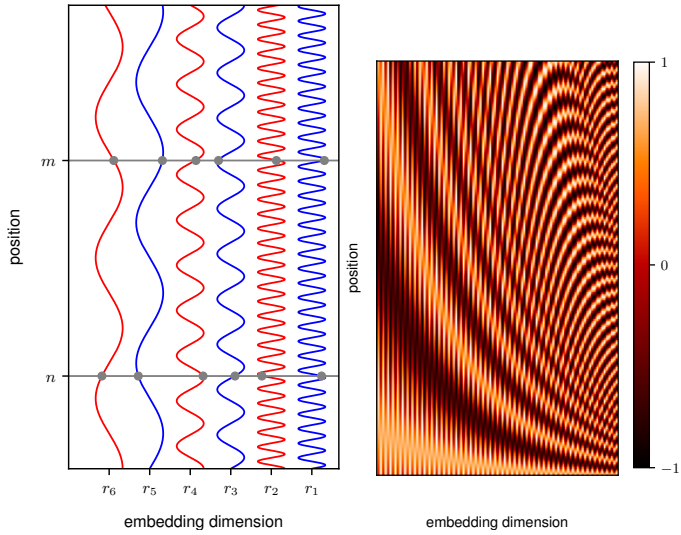
We can add the position vectors onto the token vectors to give:

$$\tilde{\mathbf{x}}_n = \mathbf{x}_n + \mathbf{r}_n$$

Positional encoding

For a given position n the associated position-encoding vector has components r_{ni} given by (Vaswani et al. 2017):

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



Natural Language Processing (NLP)

Word embedding

The first challenge is to convert the words into a numerical representation that is suitable for use as the input to a deep neural network.

One simple approach is to define a **fixed dictionary** of words and then introduce vectors of length equal to the size of the dictionary along with a **one hot** representation for each word.

The kth word in the dictionary is encoded with a vector having a 1 in position k and 0 in all other positions.

For example if `aardwolf` is the third word in our dictionary then its vector representation would be $(0, 0, 1, 0, \dots, 0)$.

Word embedding

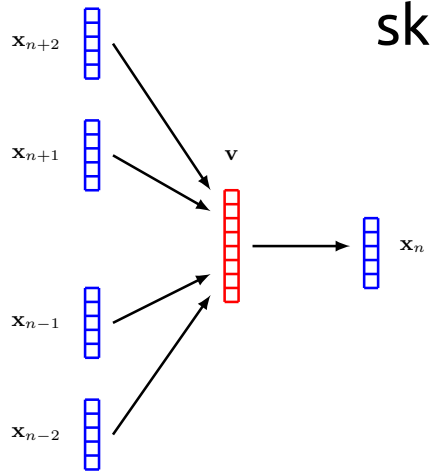
The embedding process can be defined by a matrix $\mathbf{E}$ of size $D \times K$ where D is the dimensionality of the embedding space and K is the dimensionality of the dictionary.

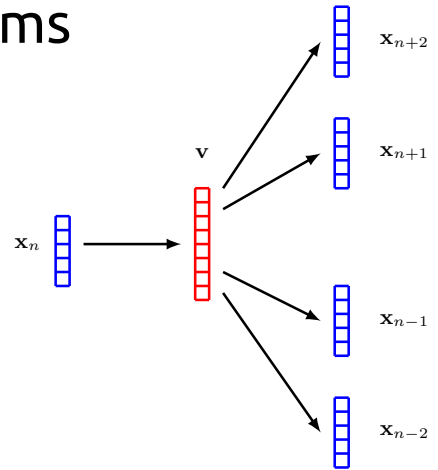
For each one-hot encoded input vector $\mathbf{x}_n$ we can then calculate the corresponding embedding vector using:

$$\mathbf{v}_n = \mathbf{E}\mathbf{x}_n.$$

Because $\mathbf{x}_n$ has a one-hot encoding, $\mathbf{v}_n$ is given by the corresponding column of the matrix $\mathbf{E}$.

Word2Vec: Continuous bag-of-words and skip-grams





Word2Vec: Continuous bag-of-words and skip-grams

E.g. $\mathbf{v}(\text{ Paris }) = [0.6, 53.0, 2.4, 0.2 \ldots]$

$$\mathbf{v}(\text{ Paris }) - \mathbf{v}(\text{ France }) + \mathbf{v}(\text{ Italy }) \simeq \mathbf{v}(\text{ Rome })$$

Tokenization

One limitation of using a fixed word dictionary is its **inability to handle out-of-vocabulary terms** or misspelled words, which restricts flexibility and robustness in natural language processing tasks.

An alternative approach is to operate at the **level of characters**.

- **The semantic structure of words is lost**, making it harder for the model to capture meaningful linguistic patterns.
- **Learning becomes more complex**, as neural networks must reconstruct words from individual characters, increasing the computational and representational burden.

Tokenization

Combining the benefits of character-level and word-level tokenization:

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

Transformer Language Models

Transformers can be used in three modes:

1. **Encoder:** Converts a sequence of tokens into a single vector, e.g., sentiment analysis where text is mapped to a sentiment label such as happy or sad.
2. **Decoder:** Generates a sequence of words from a single vector, e.g., producing a caption from an image input.
3. **Sequence-to-sequence (encoder-decoder):** Maps an input sequence to an output sequence, e.g., translating text between languages.

Decoder transformers

Decoder-only transformer models can be used as **generative models** that create output sequences of tokens.

An illustrative example is a class of models called GPT which stands for generative pretrained transformer (Radford et al., 2019; Brown et al., 2020; OpenAI, 2023).

The goal is to construct an **autoregressive model** in which the conditional distributions $p\left(\mathbf{x}_n \mid \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}\right)$ are expressed using a **transformer neural network** that is learned from data.

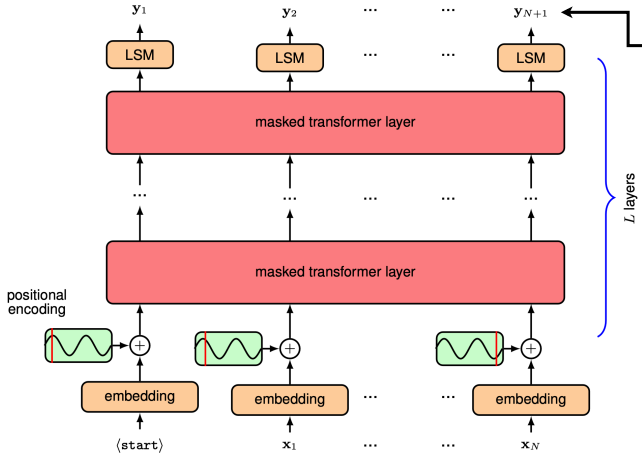
Decoder transformers

- At each step, the model receives the previous $n-1$ tokens as input and predicts the probability for the next token:

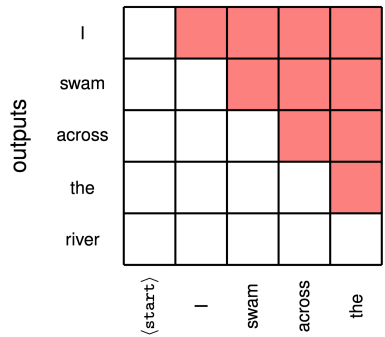
$$p(\mathbf{x}_n \mid \mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$$

- By sampling from this probability distribution, a new token is chosen and added to the sequence.
- The updated sequence, now with n tokens, is fed back into the model to predict the next token ($n+1$).
- This generation process repeats, producing one token at a time, until a specified maximum sequence length is reached.

Decoder transformers: GPT model



$$\mathbf{Y} = \text{Softmax}\left(\widetilde{\mathbf{X}}\mathbf{W}^{(\text{p})}\right)$$



Sampling strategies

The output of a decoder transformer is a **probability distribution over values for the next token in the sequence**, from which a particular value for that token must be chosen to extend the sequence.

To find the most probable sequence, we would need to maximize the joint distribution over all tokens, which is given by:

$$p\left(\mathbf{y}_1, \cdots, \mathbf{y}_N\right) = \prod_{n=1}^{N} p\left(\mathbf{y}_n \mid \mathbf{y}_1, \cdots, \mathbf{y}_{n-1}\right)$$

Complexity: $\mathcal{O}(K^N)$

Sampling strategies

Greedy Search – $\mathcal{O}(KN)$

- At each step, select the token with the highest probability.
- For step n, choose $y_n = \arg\max_i p(y_i|y_1, \dots, y_{n-1})$

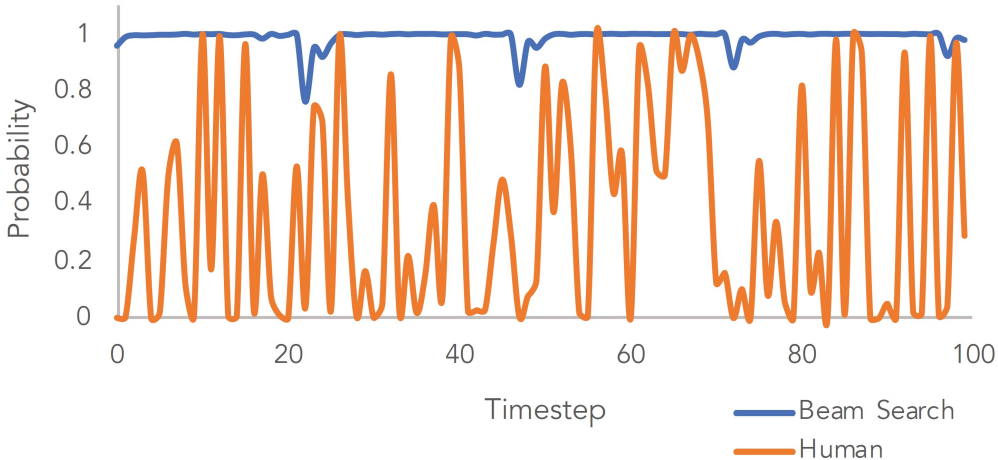
Beam Search – $\mathcal{O}(BKN)$

- Maintains top B candidate sequences, expanding each and keeping the best scoring ones at every step:

$$p(y_1, \dots, y_N) = \prod_{n=1}^{N} p(y_n|y_1, \dots, y_{n-1})$$

The algorithm keeps B sequences with the highest cumulative probability and normalizes by sequence length.

Sampling strategies



Sampling strategies

Random Sampling (Softmax Sampling) – $\mathcal{O}(KN)$

- The next token is selected stochastically using the softmax distribution over vocabulary probabilities.

$$p(y_i) = \frac{\exp(a_i)}{\sum_j \exp(a_j)}$$

where a_j is the logit for token j.

Sampling strategies

Top-K Sampling – $\mathcal{O}(KN)$

- Restricts candidates to the K most probable tokens. Samples from this truncated set.
- Similar to softmax but over top K tokens only:

$$p_{\text{Top-K}}(y_i) = \frac{\exp(a_i)}{\sum_{j \in K} \exp(a_j)}$$

for i in the set of top K.

Sampling strategies

Nucleus (Top-p) Sampling – $\mathcal{O}(NK \log K)$

- Dynamically chooses the smallest set of top tokens whose cumulative probability exceeds threshold p. Samples from this set.
- Define set S where $\sum_{i \in S} p(y_i) \geq p$, then sample among S.

Sampling strategies

Temperature Scaling – $\mathcal{O}(NK)$

- Controls randomness by scaling logits before applying softmax.

$$p_T(y_i) = \frac{\exp(a_i/T)}{\sum_j \exp(a_j/T)}$$

- For $T \to 0$ (lower temperature), is more deterministic.
- If $T = 1$, is equal to the standard softmax.
- For $T < 1$, high-probability tokens are favored.
- For $T \to \infty$, the distribution becomes uniform across states.

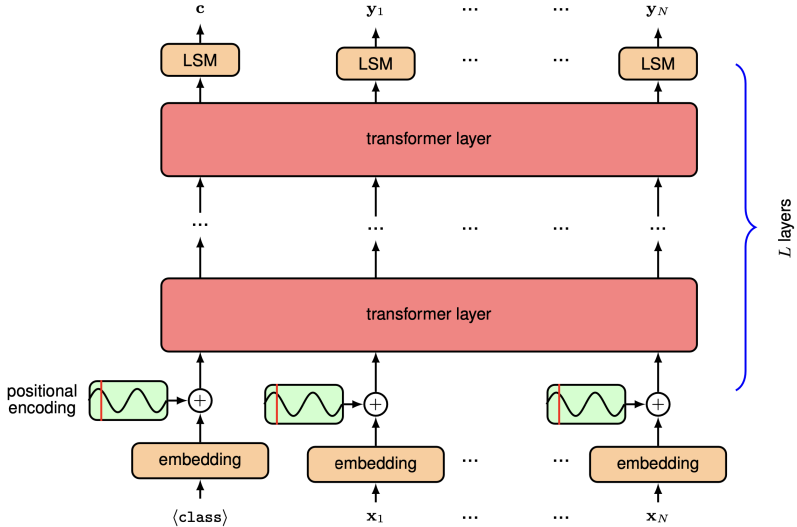
Encoder transformers

- Encoders takes sequences as input and produce fixed-length vectors, such as class labels, as output.
- An example of such a model is BERT (Bidirectional Encoder Representations from Transformers, Devlin et al., 2018).

Bidirectional – the network sees words both before and after the masked word and can use both sources of information to make a prediction.

- The goal is to pre-train a language model using a large corpus of text and then to fine-tune the model using transfer learning for a broad range of downstream tasks each of which requires a smaller application-specific training data set.

Encoder transformers



Encoder transformers

- An encoder model is unable to generate sequences.
- The first token of every input string is given by a special token $\langle \text{class} \rangle$, and the corresponding output of the model is ignored during pre-training.
- The model is pre-trained by presenting token sequences at the input.
- A randomly chosen subset of the tokens, say 15%, are replaced with a special token denoted $\langle \text{mask} \rangle$.
- The model is trained to predict the missing tokens at the corresponding output nodes.

Encoder transformers

- For example, an input sequence might be:

  I ⟨mask⟩ across the river to get to the ⟨mask⟩ bank.

  and the network should predict ‘swam’ at output node 2 and ‘other’ at output node 10.

- In this case only two of the outputs contribute to the error function and the other outputs are ignored.

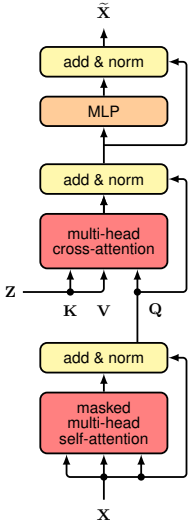
- The linear output transformation could alternatively be replaced with a more complex differentiable model such as an MLP (e.g. for token classification).

Sequence-to-sequence transformers

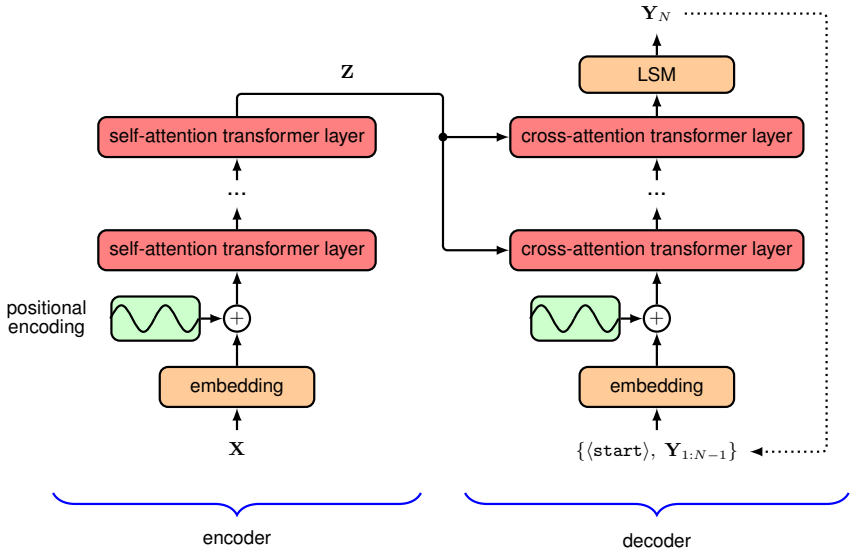
Combines an encoder and a decoder phases.

Uses a new type of attention called cross-attention in the decoder section.

$\mathbf{Z}$ denotes the output from the encoder section and determines the key and value vectors for the cross-attention layer, whereas the query vectors are determined within the decoder section.

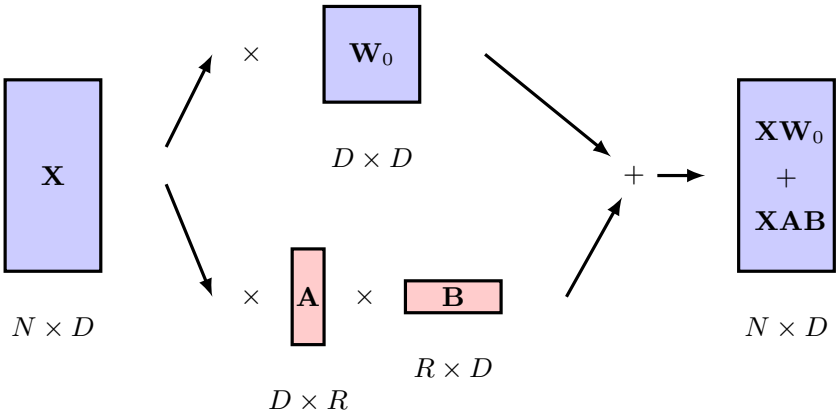


Sequence-to-sequence transformers



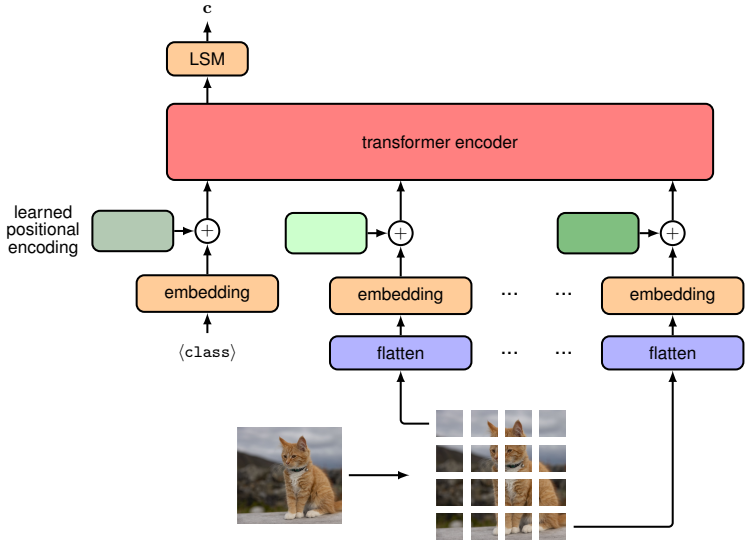
Large language models (LLMs)

[Low-rank adaptation - LoRa]



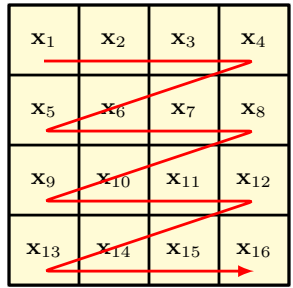
Multimodal Transformers

Vision transformers



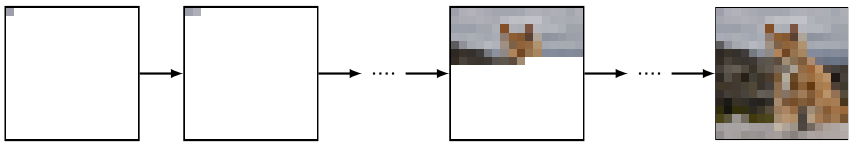
Generative image transformers

$$p\left(\mathbf{x}_1, \ldots, \mathbf{x}_N\right) = \prod_{n=1}^{N} p\left(\mathbf{x}_n \mid \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}\right)$$



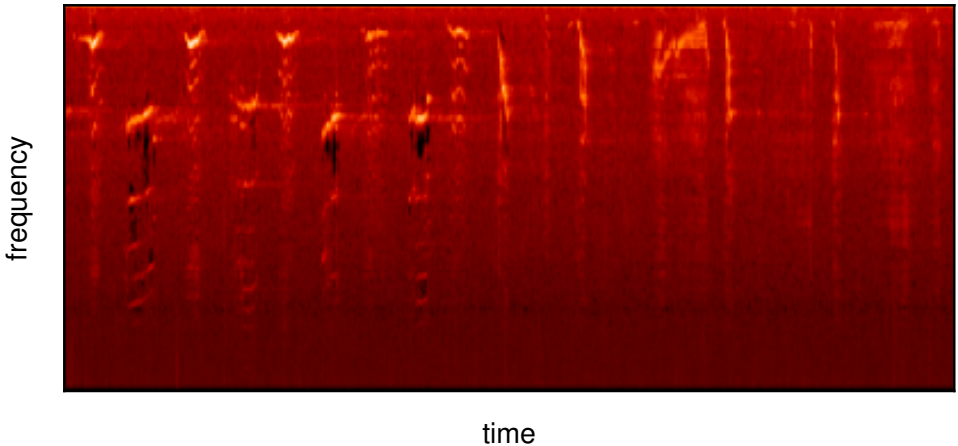
Generative image transformers

Image sampling from an autoregressive model.



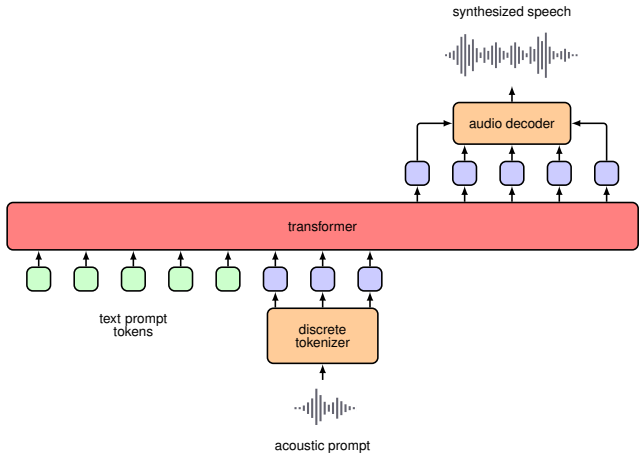
The first pixel is sampled from the marginal distribution $p(\mathbf{x}_{11})$, the second pixel from the conditional distribution $p(\mathbf{x}_{12} \mid \mathbf{x}_{11})$, and so on in raster scan order until we have a complete image.

Audio data



Text-to-speech

[Vall-E]



Vision and language transformers