Transformers:

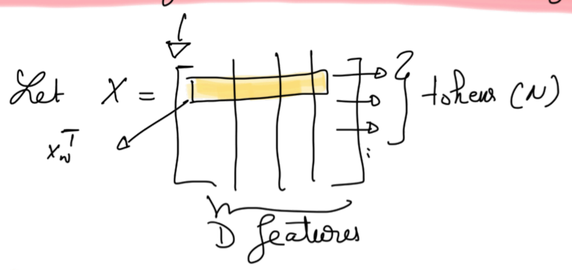
- Based on a processing called 'attention'
- Inductive Bias is what a model believes before seeing the data
  - CNNs: Local Patterns matter (pixels next to each other)
  - RNNs: things happen in order and recent inputs matter more
  - Transformers: Any input can relate to any other input: no built-in locality of order.

- These models are known as transformers because they transform a set of vectors in some representation space into a corresponding set of vectors, having the same dimensionality in a new space. This new space has a richer internal representation.

(1) Attention: It is a mechanism that allows the model to assign different levels of importance to different parts of the input, enabling it to selectively focus on the most relevant information when computing a representation or making predictions.

1- Transformer Processing:

- the input data to a transformer is a set of vectors $\{x_n\}$ of dimensionality D. $n \in \{1 \dots N\}$
- Those data vectors are called "tokens". A token might correspond to a word within a sentence or a patch in an image.
- The elements x_{ni} of each token are called features.



$\tilde{X} = \text{Transformer Layer}(X)$
↳ a function that takes the data matrix X as input and creates a transformed matrix $\tilde{X}$ of the same dimensionality as output

- Multiple transformer layers are applied in succession to create a deep network capable of learning rich internal representations.
  - Each transformer layer contains its own weights and biases. (learned using gradient descent using an appropriate cost function)
  - Each transformer layer is composed by 2 stages:
    - (1) Attention Mechanism: mixes information between different tokens (rows) → for each token (word), it looks at other tokens and decides which ones are important and combines their information.
    - (2) Feed Forward Transformation: works on each token seperately. → updates the token's internal representation without looking at other tokens.

2- Attention Coefficients:

we have our set of input tokens $x_1 \dots x_N$ in an embedding space and we want to map this to another set $y_1 \dots y_N$ having the same number of tokens but in a new embedding space that captures a richer semantic structure.

- The value of y_N should depend not just on the corresponding input vector x_N but on all the vectors $x_1 \dots x_N$ in the set.
- A simple way to achieve this is to define each output vector y_N as a linear combination of the input vectors $x_1 \dots x_N$ with weighting coefficients.

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

(a_{nm}: attention weights)

Constraints:
- $a_{mn} \geq 0$
- $\sum_{m=1}^{N} a_{mn} = 1$

3- Self Attention:

given a sequence, we want each token to look at all the other tokens (including itself) and decides how much attention to pay to each one to compute a new representation.

- This is done using:
  - Queries (Q): what this token is asking for?
  - Keys (K): what this token offers?
  - Values (V): the actual value/info this token carries.

(1) Linear Projections: we project X into Query, Key and Value matrices using learnable weights.

$$Q = X \cdot W_Q \quad ; \quad K = X \cdot W_K \quad ; \quad V = X \cdot W_V$$

where: $W_Q, W_K \in \mathbb{R}^{d \times d_k}$; $W_V \in \mathbb{R}^{d \times d_v}$

- d_k is usually set to $d_k = \frac{d_{model}}{num\_heads}$

(2) Attention Scores: we compute the pairwise product between all queries and keys

$$scores = \frac{Q \cdot K^T}{\sqrt{d_k}} \in \mathbb{R}^{N \times N}$$

(3) Softmax to normalize:

$$\alpha_{nm} = \frac{\exp[score_{nm}]}{\sum_{m=1}^{N} \exp[score_{nm}]}$$

(4) Weighted Sum of Values:

$$y_n = \sum_{m=1}^{N} \alpha_{nm} V_m \quad \equiv \quad Attention(Q,K,V) = Softmax\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right) \cdot V$$

(2) Example:

- Input: 3 tokens with 2 features → $X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \in \mathbb{R}^{3 \times 2}$
- For simplicity now, we take $d_{model} = 2 = d_k$ and set $W_Q = W_V = W_K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

1) Compute $Q \cdot K^T$:

$$Q = X \cdot W_Q = \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}}_{3 \times 2} \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}}_{2 \times 2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 2 \end{bmatrix}$$

$$K = Q \rightarrow Q \cdot K^T = \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}}_{3 \times 2} \underbrace{\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}}_{2 \times 3} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

2) Scores? → Scale by d_k

$$\frac{Q \cdot K^T}{\sqrt{2}} = \begin{bmatrix} 0{,}707 & 0 & 0{,}707 \\ 0 & 0{,}707 & 0{,}707 \\ 0 & 0{,}707 & 1{,}414 \end{bmatrix}$$

3) Apply Softmax rows wise:

row 1: $softmax[0{,}707 \quad 0 \quad 0{,}707] = [0{,}401 \quad 0{,}198 \quad 0{,}401]$

4) Multiply by V:

$$y_1 = 0{,}401 \underbrace{[0 \; 1]}_{\text{token 1}} + 0{,}198 [0 \; 1] + 0{,}401 [1 \; 1]$$

↳ $y_1 = [1{,}203 \; , \; 0{,}599]$

1. For each output yn, the corresponding input xn is considered the query while all inputs xm (including itself) are considered keys and values.
2. This process is calles Self Attention because we use the same sequence to computer the queries, keys ans values.
3. This is also knows as dot product Self Attention because the similarity between key and query is computed by a dot product.

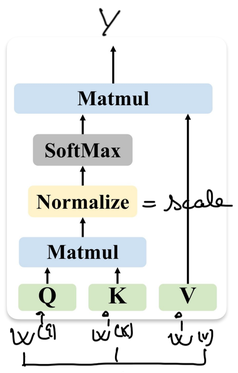
There is nof probabilistic interpretation to the Softmax in this case and it is only uses to normalize the attention weights (on each row independently) to make them sum to 1.

③ Attention Parameters:

Attention parameters are $W^{(q)}$, $W^{(K)}$ and $W^{(V)}$.

- $W^{(K)}$ has dimensionality $d \times dk$. } so we can do the dot product
- $W^{(q)}$ " " "
- $W^{(V)}$ " " $d \times dv$. if $d = dv$, the output representation has the same dimensionality as the input.

$W^{(K)}$, $W^{(q)}$ and $W^{(V)}$ are learnable linear transformations (fully connected layers without activation) applied to the input tokens to produce the query, key and value. In practice, bias parameters are also included.



④ Multihead Attention:

Why? Single head attention computes one similarity space. but that might not be enough to capture all types of relationships (position, semantics etc).

↳ MHA lets the model focus on different aspects of the input at the same time, capturing richer and more diverse relationships.

↳ Idea:

- we split the input into multiple heads
- Each head has its own W_Q, W_K and W_V.
- Each head computes its own self attention.
- Then we concatenate the results and project them back to the model dimension.

↳ Let's

① $X \in \mathbb{R}^{N \times d}$: input tokens
h : nb of heads
$dk = dv = d/h$: dimension per head

② for each head $i = 1 .. h$ compute:

$Q_i = X \cdot W_q^i \in \mathbb{R}^{N \cdot dk}$
$N \cdot d \quad d \times dk$
$K_i = X \times W_K^i \in \mathbb{R}^{N \times dk}$
$V_i = X \, W_V^i \in \mathbb{R}^{N \times dk}$

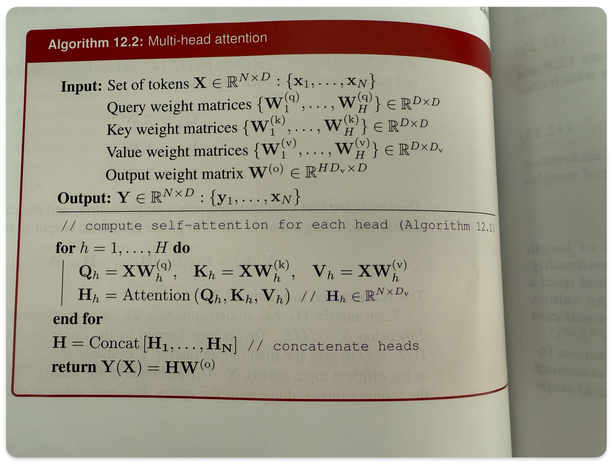
③ then apply scaled dot product attention per head

$$\text{head}_i = \text{Softmax}\left[\frac{Q_i \, k_i^T}{\sqrt{dk}}\right] \cdot V_i \in \mathbb{R}^{N \times dk}$$

④ Concatenate and project

$$\Pi\text{HA}(X) = \text{Concat}(\text{head}_1 \dots \text{head}_h) \cdot W_o$$
$\in \mathbb{R}^{N \cdot d}$; $\in \mathbb{R}^{N \times (h \cdot dk)}$; $\in \mathbb{R}^{(h \cdot dk) \cdot d}$

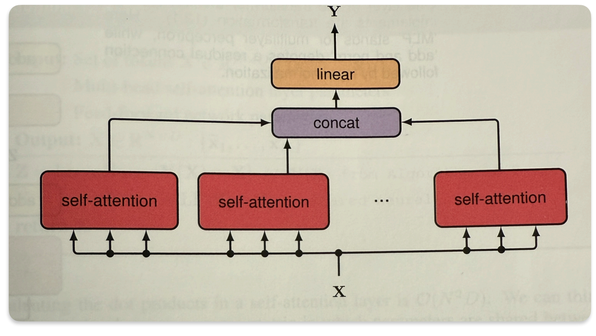
Even tho the concatenated output already has the same shape as the input, the final projection with W_o allows to learn how to combine the information from all the heads into a meaningful representation. because the concatenation allows to just put the heads side by side → disjoint information



Algorithm 12.2: Multi-head attention

Input: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$
Query weight matrices $\{\mathbf{W}_1^{(q)}, \dots, \mathbf{W}_H^{(q)}\} \in \mathbb{R}^{D \times D}$
Key weight matrices $\{\mathbf{W}_1^{(k)}, \dots, \mathbf{W}_H^{(k)}\} \in \mathbb{R}^{D \times D}$
Value weight matrices $\{\mathbf{W}_1^{(v)}, \dots, \mathbf{W}_H^{(v)}\} \in \mathbb{R}^{D \times D_v}$
Output weight matrix $\mathbf{W}^{(o)} \in \mathbb{R}^{HD_v \times D}$
Output: $\mathbf{Y} \in \mathbb{R}^{N \times D} : \{\mathbf{y}_1, \dots, \mathbf{x}_N\}$

// compute self-attention for each head (Algorithm 12.1)
for $h = 1, \dots, H$ **do**
  $\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(q)}, \quad \mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(k)}, \quad \mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(v)}$
  $\mathbf{H}_h = \text{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h)$ // $\mathbf{H}_h \in \mathbb{R}^{N \times D_v}$
end for
$\mathbf{H} = \text{Concat}[\mathbf{H}_1, \dots, \mathbf{H}_N]$ // concatenate heads
return $\mathbf{Y}(\mathbf{X}) = \mathbf{H}\mathbf{W}^{(o)}$



⑤ Transformer Layers:

- Multihead Self Attention (TIHSA) is the core architecture of the transformer network.
- Neural Networks benefit greatly from depth → we want to stack multiple SA layers on top of each other.
- To improve training efficiency, residual connection (adding the input to the output) are introduced → we need the dimensionality to be equal to the input one → NxD.

↳ why residual connections?

- Stabilize gradients because when we stack many layers, gradients can explode or vanish
- Allow the model to learn adjustments
  ↳ it gives the model flexibility to learn when to change and when to keep information.
- Enable deeper architectures → without residuals, deep transformers (with many layers) become hard to train.

- Then, we apply layer normalization to improve training efficacy

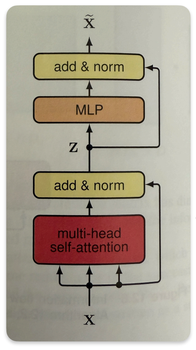
$$Z = \text{LayerNorm}[X(X) + X]$$ ≡ $Z = \text{LayerNorm}(X) + X$ ⚠
↳ the output of the TIHSA ↳ more effective

Even tho non linearity enters through attention weights (Softmax) but the outputs are still constrained to lie in the subspace spanned by the input vectors → limit the expressiveness of the attention layer → we use a standard non-linear neural network with D inputs and D outputs as a post processing of the output of each layer.

$$\tilde{X} = \text{LayerNorm}[\text{TILP}(Z) + Z]$$
↳ the final output

≡ $\tilde{X} = \text{TILP}[\text{LayerNorm}(Z)] + Z$

↳ one transformer layer



⑥ Computational Complexity:

it's a way to estimate how much work an algorithm needs depending on the input.

1- Linear proj: Calculate Q, K and V.

$Q = X \, W_Q$ → shape = (N, D)
$N \cdot D \quad D \cdot D$

token 1 → $N \times D$; $D \times D$; for token 1, on a [5 multiplications + 4 additions] x 5 → $\mathcal{O}(5^2)$ → $\mathcal{O}(D^2)$

↳ for N tokens → $\mathcal{O}(ND^2)$ ①

2- Attention Scores calculation:

$\text{scores} = Q \, @ \, K^T$
$N \times D \quad D \times N$

token 1, token 2 → $N \times D$ (2 x 3) ; $D \times N$

for token 1, on a [3 multi + 2 additions] x N x 2
↳ $\mathcal{O}(6 + 4) = \mathcal{O}(6) = \mathcal{O}(N \cdot D)$
↳ for N tokens → $\mathcal{O}(N^2 D)$ ②

③ Apply Attention weights to

$\text{output} = \text{softmax}(\text{scores}) \cdot V$
$N \cdot N \quad N \cdot D$

token 1, token 2 → 2x2 ; for token 1, on a (2 multi + 1 addi) x 3
↳ 6 multi + 3 add
↳ $\mathcal{O}(6) = \mathcal{O}(N \cdot D)$
↳ for N tokens → $\mathcal{O}(N^2 D)$ ③

↳ total cost = $\mathcal{O}(ND^2 + N^2 D + N^2 D)$
$= \mathcal{O}(ND^2 + 2N^2 D)$
$= \mathcal{O}(ND^2 + N^2 D)$
≡ generally $N >> D$ → $\mathcal{O}(N^2 D)$ → SA is quadratic in sequence length ⚠