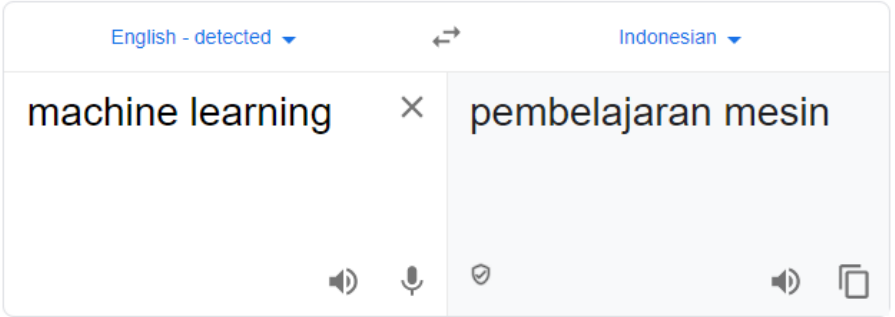
# Transformers

# Sequence Data

We will focus on applications where it is desirable to predict an output at each element of the sequence

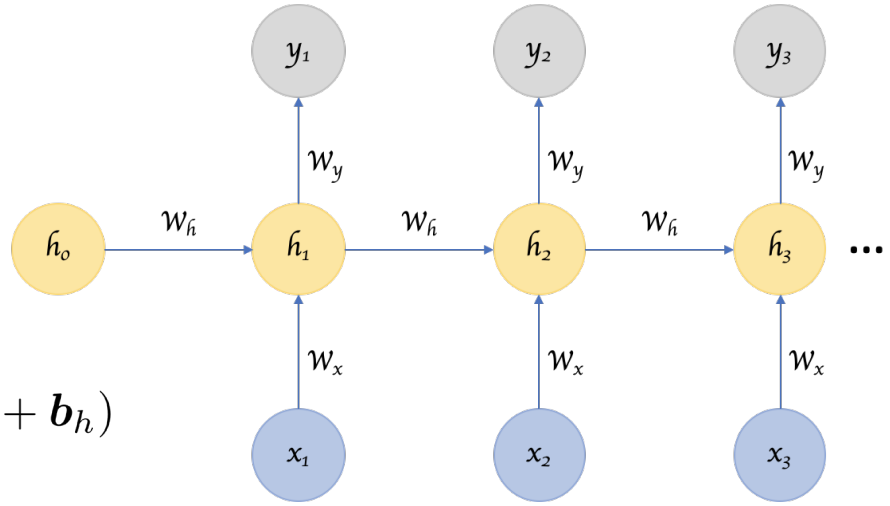
- Stock market data
- Heart rate monitoring (patient monitoring in general)
- Music (transcription, source separation)
- Text (Text-to-speech, auto-completion, machine translation)
- Speech (Speech-to-text)
- Video (tracking, tagging, captioning, object recognition)
- fMRI
- Weather forecasting
- Production management
- . . .



# Recurrent Neural Networks

- Input: $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T \in \mathbb{R}^{d_x}$
- Output: $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T \in \mathbb{R}^{d_y}$
- Hidden state: $\boldsymbol{h}_0, \boldsymbol{h}_1, \ldots, \boldsymbol{h}_T \in \mathbb{R}^{d_h}$
- Dynamical model:

$$\boldsymbol{h}_t = \sigma(\boldsymbol{W}_h \boldsymbol{h}_{t-1} + \boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{b}_h)$$
$$\boldsymbol{y}_t = \boldsymbol{W}_y \boldsymbol{h}_t + \boldsymbol{b}_y$$




- $\sigma$ is a scalar activation function applied elementwise
- Intuitively, $\boldsymbol{x}_t$ and $\boldsymbol{y}_t$ are Euclidean vectors that represent an element of a sequence, such a phoneme or word (text) or a vocal utterance (speech), while $\boldsymbol{h}_t$ is an intermediate variable that summarizes both the input $\boldsymbol{x}_t$ and history $\boldsymbol{h}_{t-1}$, and is used to predict $\boldsymbol{y}_t$.
- The weight matrices and biases are learned from training data using an algorithm called *backpropagation through time*

Figure: https://gotensor.com/2019/02/28/recurrent-neural-networks-remembering-whats-important/

# Recurrent Neural Networks

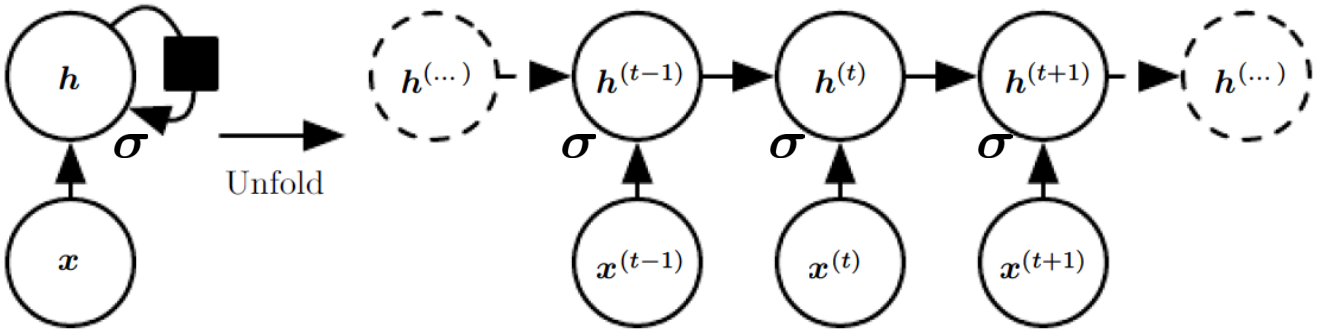
- Consider hidden state recursion (recurrent circuit)

$$\boldsymbol{h}(t) = \boldsymbol{\sigma}(\boldsymbol{W}_{hh}\boldsymbol{h}(t-1) + \boldsymbol{W}_{hx}\boldsymbol{x}(t) + \boldsymbol{b}_h), \quad t = 1, \ldots, T$$

- Recursion can be unfolded: $\boldsymbol{h}(T) = g(\boldsymbol{x}(1), \ldots, \boldsymbol{x}(T))$

  for a function **g** depending on **W, b.**

- This gives a representation of an RNN as a feed forward neural network (FFNN) with T layers



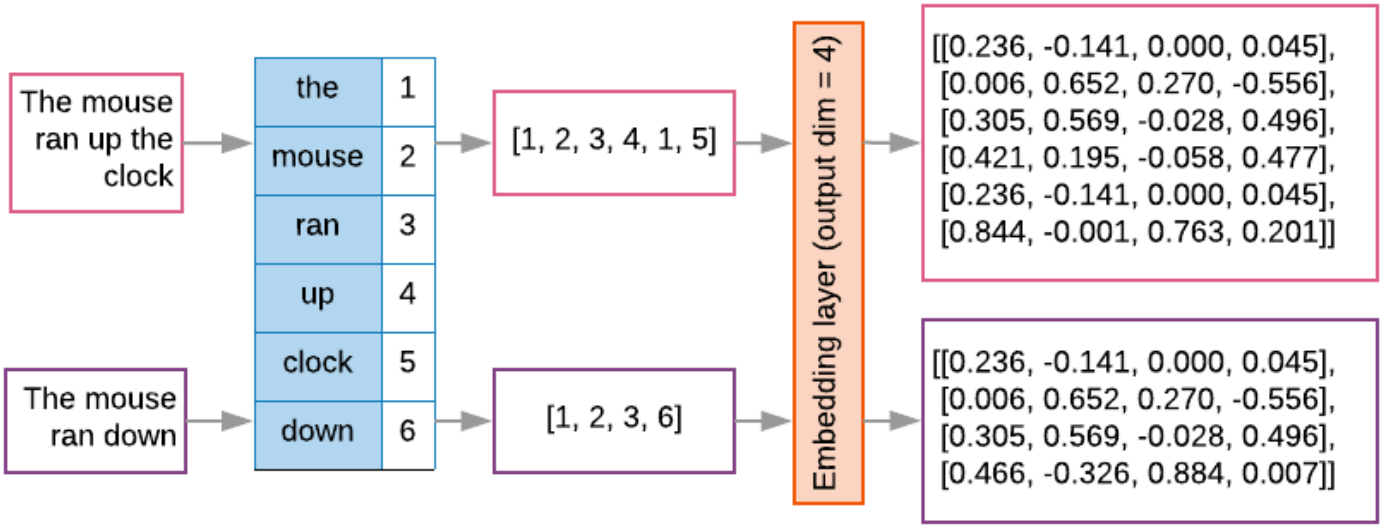

Recurrent circuit (RNN)  Unfolded circuit (FFNN with T layers)

# Backpropagation Through Time

- Main idea: unfolded RNN is simply a FFNN with T layers
- Hence, can apply standard backprogagation with constraints:
  - Matrices $\mathbf{W}_{hh}$, $\mathbf{W}_{hx}$, $\mathbf{W}_{yh}$ are *identical* over all T layers
  - Biases are *identical* over all T layers
  - These three sets of parameters are *shared* over all layers
- Variant of BP on unfolded RNN is called BPTT
  - Unfold RNN to T layers (T is maximum length sequence in database)
  - Decouple $\mathbf{W}_{hh}$, $\mathbf{W}_{hx}$, $\mathbf{W}_{yh}$ as $\mathbf{W}_{hh}^{(k)}$, $\mathbf{W}_{hx}^{(k)}$, $\mathbf{W}_{yh}^{(k)}$ at each layer, k=1:T
  - Run BP, updating gradients wrt $\{\mathbf{W}_{hh}^{(k)}, \mathbf{W}_{hx}^{(k)}, \mathbf{W}_{yh}^{(k)}\}$ as if they were not recurrent from layer to layer
  - At end of each full pass project onto linear subspace of recurrent matrices
  - Corresponds to averaging the T weight gradients over k.
- This results in a single set of learned RNN recurrent weights $\mathbf{W}_{hh}$, $\mathbf{W}_{hx}$, $\mathbf{W}_{yh}$

# Feature Vectors for Text

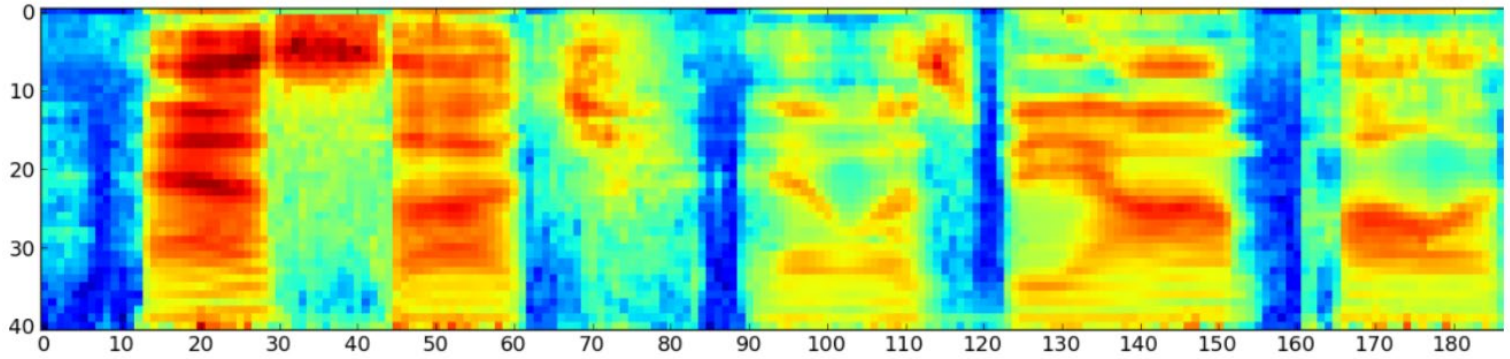
- Text is segmented into small units called *tokens*. Think of these as short strings of characters.
- Example: GPT Tokenizer
- Tokens are then mapped to Euclidean space with a *word embedding*. Every token gets mapped to a distinct vector. We'll have a lecture on Euclidean embedding later.




https://geoffrey-geofe.medium.com/tokenization-vs-embedding-understanding-the-differences-and-their-importance-in-nlp-b62718b5964a

# Feature Vectors for Speech

- A speech signal can be represented by a feature vector where each feature represents the instantaneous energy of the signal in a particular frequency range. Look up *spectrogram* for more details.



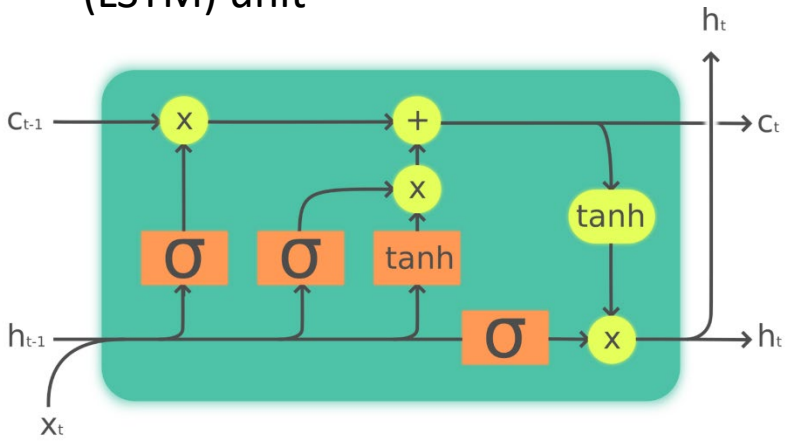

column $t$ is $x_t$

Graves, Mohamed, Hinton, ICASSP 2013

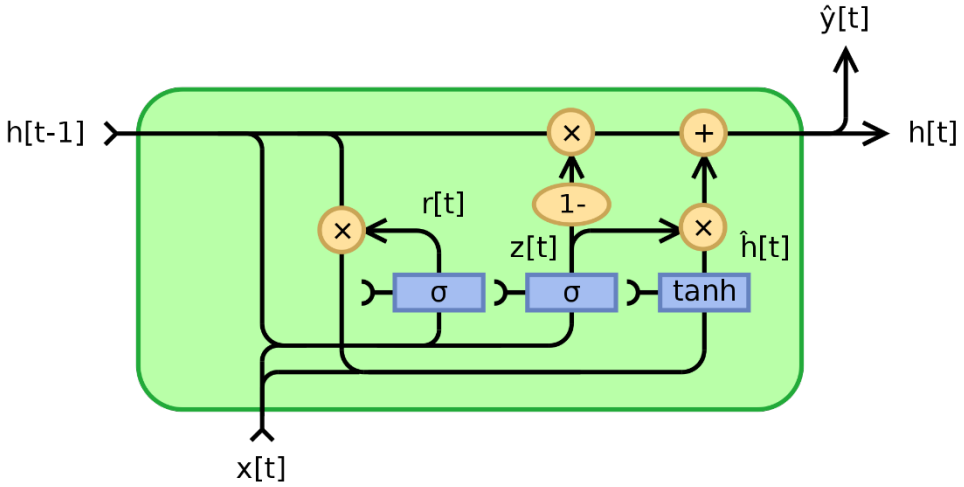
# Fancier Recurrent Units

- Address two shortcomings of basic RNNs
  - Vanishing and exploding gradients
  - Long term dependencies: “The guy with the really big hat on his head walked into the store”

- Long short-term memory (LSTM) unit




- Gated Recurrent Unit (GRU)



# Limitations of RNNs

- While RNNs (especally with LSTM recurrent units) perform extremely well at some tasks (and are incorporated into many commercial products), they are less adept at others.
- In some sequential prediction tasks (like translation), $\boldsymbol{y}_t$ depends on several inputs, including possibly from the distant past or the future.
- The precise inputs that each $\boldsymbol{y}_t$ depends on may vary with $t$

$i = 2$
$j = 3$
$a_{ij} = a_{ji}$ ?

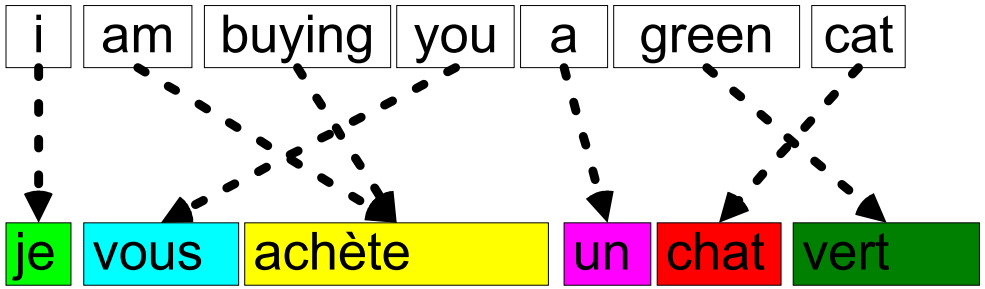



Figure credit: Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi, N., … & Dyer, C. (2007, June). Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the association for computational linguistics companion volume proceedings of the demo and poster sessions* (pp. 177–180).

# Self Attention Layer

- A self-attention layer is a layer in a feedforward neural network, just like fully connected and convolutional layers
- The input to a self-attention layer is $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N \in \mathbb{R}^{d_x}$
- The output of a self-attention layer is $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N \in \mathbb{R}^{d_y}$
- Time is ignored (for now): we predict the entire output sequence from the entire input sequence
- Self-attention layers are the key ingredient in transformers, which are the architectures used in large language models
- There is a lot of research prior to self-attention layers that attempted to improve upon the limitations of RNNs, including bidirectional RNNs, deep RNNs, sequence to sequence learning, and RNNs with attention. Self-attention is often motivated from that history. I will instead present a standalone motivation for self-attention layers.

# Self Attention Layer

- Layer input: $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N \in \mathbb{R}^{d_x}$. Layer output: $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N \in \mathbb{R}^{d_y}$
- There are still unobserved (hidden) variables $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N \in \mathbb{R}^{d_y}$ (note dimension) that are functions of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$, and used to predict $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N$
- The output is defined by

$$\boldsymbol{y}_j = \sum_{i=1}^{N} a_{ij} \boldsymbol{h}_i,$$

  where the weights $a_{ij}$ satisfy

$a_{ij} \geq 0$ , $\sum_{i=1}^{N} a_{ij} = 1$

  and are called the attention weights

- The feature vector $\boldsymbol{h}_i$ is derived from $\boldsymbol{x}_i$, and captures those aspects of $\boldsymbol{x}_i$ that are most important for predicting the outputs.
- The $a_{ij}$ are also functions of the inputs, which is where the term "self" come from. Previous approaches used additional information to compute these weights.

# Self Attention Layer

- Introduce the matrices $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{H}$, and $\boldsymbol{A}$.

$$X = \begin{bmatrix} x_1 & \cdots & x_N \end{bmatrix} \quad (d_x \times N) \qquad Y = \begin{bmatrix} y_1 & \cdots & y_N \end{bmatrix} \quad (d_y \times N)$$

$$H = \begin{bmatrix} h_1 & \cdots & h_N \end{bmatrix} \quad (d_y \times N) \qquad A = \begin{bmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \\ a_{N1} & & a_{NN} \end{bmatrix} \quad (N \times N)$$

- We can now express the output in matrix form

$$Y = HA$$

- Keep in mind that a self-attention layer is a layer in a larger network. Thus it should be as simple as possible while conforming to the self-attention model as expressed above. We gain complexity by composing layers as usual.

- As with fully connected and convolutional layers, a self attention layer is expressed in terms of (bi)linear operations and one nonlinearity

# Self Attention Layer

- Like in RNNs, each feature vector $\boldsymbol{h}_i$ is a *linear* function of the corresponding input $\boldsymbol{x}_i$

$$\forall i \quad h_i = W_h x_i$$

$$\Updownarrow$$

$$H = W_h X$$

# Attention Weights

- Let $\boldsymbol{a}_j$ denote the $j$-th column of $\boldsymbol{A}$. These are probability vectors.
- Recall the softmax function $v \in \mathbb{R}^N$

$$\psi(v) = \begin{bmatrix} \psi_1(v) & \cdots & \psi_N(v) \end{bmatrix}^T, \quad \psi_i(v) = \frac{\exp(v_i)}{\sum_k \exp(v_k)}$$

- Now define the function $\Psi : \mathbb{R}^{N \times N} \to \mathbb{R}^{N \times N}$ that applies softmax columnwise

If $Z \in \mathbb{R}^{N \times N}$, then $\Psi(Z) = \begin{bmatrix} \psi(z_1) & \cdots & \psi(z_N) \end{bmatrix}$

where $z_i$ = $i^{th}$ column of $Z$.

- The idea is that $a_{ij}$ should reflect the ~~similarity between $\boldsymbol{x}_i$ and $\boldsymbol{y}_j$~~. Thus we will let

influence of $x_i$ on $y_j$

$$A = \Psi(S)$$

where $\boldsymbol{S}$ is a ~~similarity~~ matrix whose $(i, j)$ element captures the ~~similarty between $\boldsymbol{x}_i$ and $\boldsymbol{y}_j$~~. influence

influence of $x_i$ on $y_j$

# Poll

influence

True or false: Ideally, the ~~similarity~~ matrix should be symmetric.

(A) True

(B) False

# Influence ~~Similarity~~ Matrix

- In the interest of simplicity, we take the most basic similarity measure, the dot product (scaled)

- In particular, assume that $\boldsymbol{S} = [s_{ij}]_{ij}$ where

$$s_{ij} = \langle \boldsymbol{b}_i, \boldsymbol{c}_j \rangle \Big/ \sqrt{d_s}$$

for some vectors $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_N \in \mathbb{R}^{d_s}$ and $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_N \in \mathbb{R}^{d_s}$

↙ encodes influence of $x_i$ on $y_j$

- Introduce the matrices

$$B = \begin{bmatrix} b_1 & \cdots & b_N \end{bmatrix}, \quad C = \begin{bmatrix} c_1 & \cdots & c_N \end{bmatrix}$$

- Now assume $\boldsymbol{B}$ and $\boldsymbol{C}$ are linear function of the inputs:

$$B = W_B X \qquad C = W_C X$$

# Self Attention Layer

- In summary, the self attention layer is

$$\begin{aligned}\boldsymbol{Y} &= \boldsymbol{H}\boldsymbol{A} \\ &= \boldsymbol{W}_h\boldsymbol{X}\Psi(\boldsymbol{S}/\sqrt{d_s}) \\ &= \boldsymbol{W}_h\boldsymbol{X}\Psi(\boldsymbol{B}^T\boldsymbol{C}/\sqrt{d_s}) \\ &= \boldsymbol{W}_h\boldsymbol{X}\Psi(\boldsymbol{X}^T\boldsymbol{W}_b^T\boldsymbol{W}_c\boldsymbol{X}/\sqrt{d_s})\end{aligned}$$

with learnable parameters $\boldsymbol{W}_h$, $\boldsymbol{W}_b$, and $\boldsymbol{W}_c$ and hyperparameter $d_s$.

# Poll

True or false: The self-attention layer takes into account that the input is a sequence. In other words, it accounts for the order in which the inputs are presented.

(A) True

(B) False

# Remarks

- All of the operations are differentiable: can backpropagate
- In the literature, the matrices $\boldsymbol{H}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ are usually denoted $\boldsymbol{V}$, $\boldsymbol{K}$, and $\boldsymbol{Q}$, and called the value, key, and query matrices. This terminology derives from information retrieval.
- As formulated, the self-attention layer does not account for the sequential nature of the data, and in fact is applicable to nonsequential data.
- "Positional encoding" is a technique to account for sequential data (modify the $\boldsymbol{x}_n$ vectors to encode sequence position). Another technique is to force certain entries of the ~~similarity~~ influence matrix to be $-\infty$.

# Acknowledgment

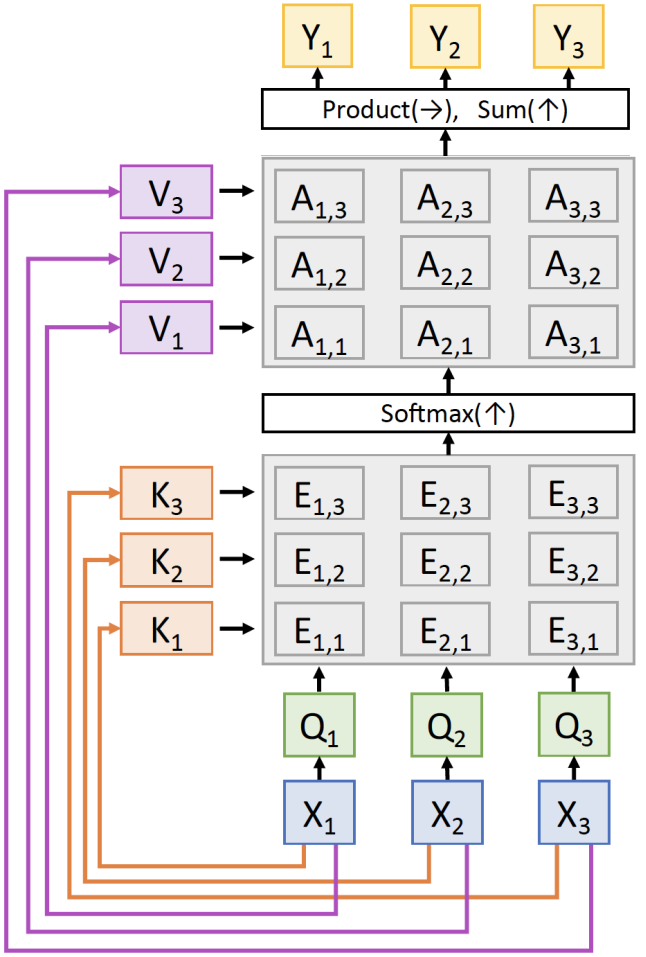

- The following slides are from Justin Johnson (shared with me by Samet Oymak who may have modified them slightly).

# Self-Attention Layer

One **query** per **input vector**

**Inputs**:
**Input vectors**: $X$ (Shape: $N_X \times D_X$)
**Key matrix**: $W_K$ (Shape: $D_X \times D_Q$)
**Value matrix**: $W_V$ (Shape: $D_X \times D_V$)
**Query matrix**: $W_Q$ (Shape: $D_X \times D_Q$)

**Computation**:
**Query vectors**: $Q = XW_Q$
**Key vectors**: $K = XW_K$ (Shape: $N_X \times D_Q$)
**Value Vectors**: $V = XW_V$ (Shape: $N_X \times D_V$)
**Similarities**: $E = QK^T / \sqrt{D_Q}$ (Shape: $N_X \times N_X$) $E_{i,j} = (Q_i \cdot K_j) / \sqrt{D_Q}$
**Attention weights**: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_X \times N_X$)
**Output vectors**: $Y = AV$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j} V_j$



# Masked Self-Attention Layer

Don't let vectors "look ahead" in the sequence

**Inputs**:

**Input vectors**: X (Shape: $N_X$ x $D_X$)

**Key matrix**: $W_K$ (Shape: $D_X$ x $D_Q$)

**Value matrix**: $W_V$ (Shape: $D_X$ x $D_V$)

**Query matrix**: $W_Q$ (Shape: $D_X$ x $D_Q$)

**Computation**:

**Query vectors**: $Q = XW_Q$

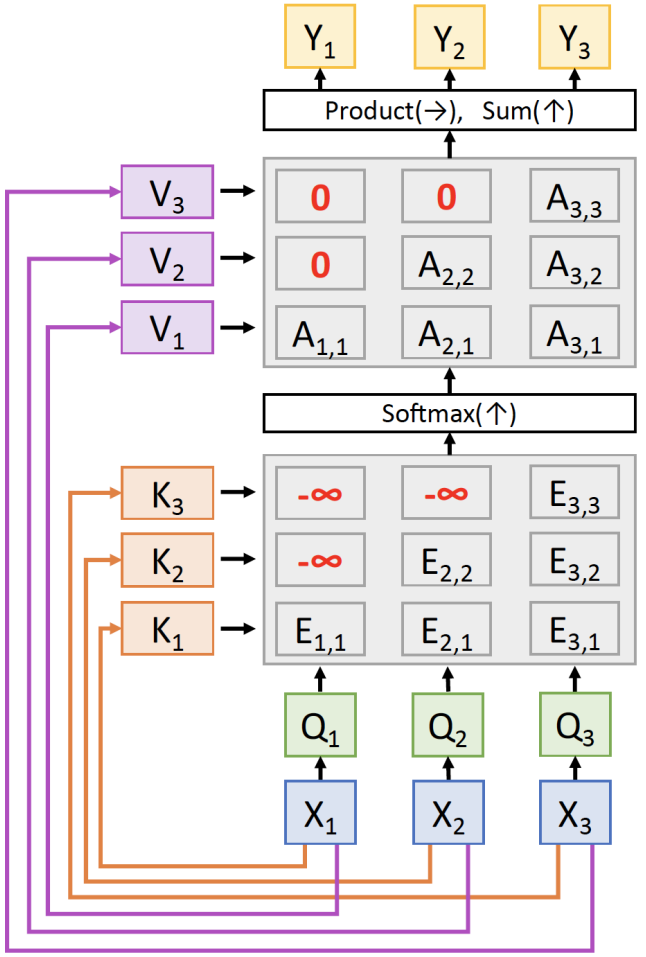
**Key vectors**: $K = XW_K$ (Shape: $N_X$ x $D_Q$)

**Value Vectors**: $V = XW_V$ (Shape: $N_X$ x $D_V$)

**Similarities**: $E = QK^T / \sqrt{D_Q}$ (Shape: $N_X$ x $N_X$) $E_{i,j} = (Q_i \cdot K_j) / \sqrt{D_Q}$

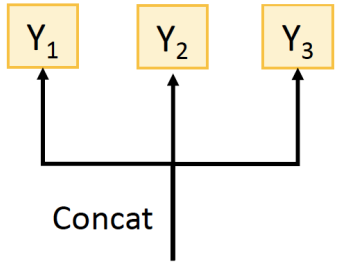
**Attention weights**: A = softmax(E, dim=1) (Shape: $N_X$ x $N_X$)

**Output vectors**: Y = AV (Shape: $N_X$ x $D_V$) $Y_i = \sum_j A_{i,j} V_j$



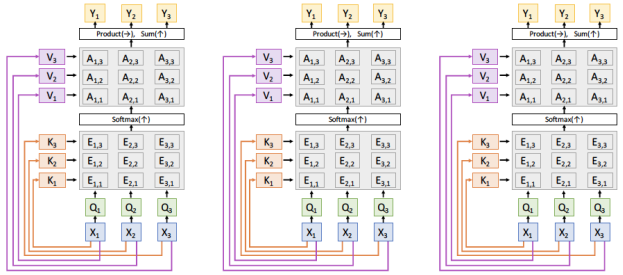
# Multihead Self-Attention Layer

Use H independent
"Attention Heads" in parallel




**Inputs**:
**Input vectors**: X (Shape: $N_X$ x $D_X$)
**Key matrix**: $W_K$ (Shape: $D_X$ x $D_Q$)
**Value matrix**: $W_V$ (Shape: $D_X$ x $D_V$)
**Query matrix**: $W_Q$ (Shape: $D_X$ x $D_Q$)

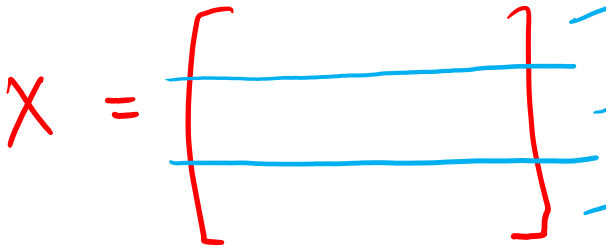
**Hyperparameters**:
Query dimension $D_Q$
Number of heads H



**Computation**:
**Query vectors**: $Q = XW_Q$
**Key vectors**: $K = XW_K$ (Shape: $N_X$ x $D_Q$)
**Value Vectors**: $V = XW_V$ (Shape: $N_X$ x $D_V$)
**Similarities**: $E = QK^T / \sqrt{D_Q}$ (Shape: $N_X$ x $N_X$) $E_{i,j} = (Q_i \cdot K_j) / \sqrt{D_Q}$
**Attention weights**: A = softmax(E, dim=1) (Shape: $N_X$ x $N_X$)
**Output vectors**: Y = AV (Shape: $N_X$ x $D_V$) $Y_i = \sum_j A_{i,j} V_j$

Split

$X_1$ $X_2$ $X_3$



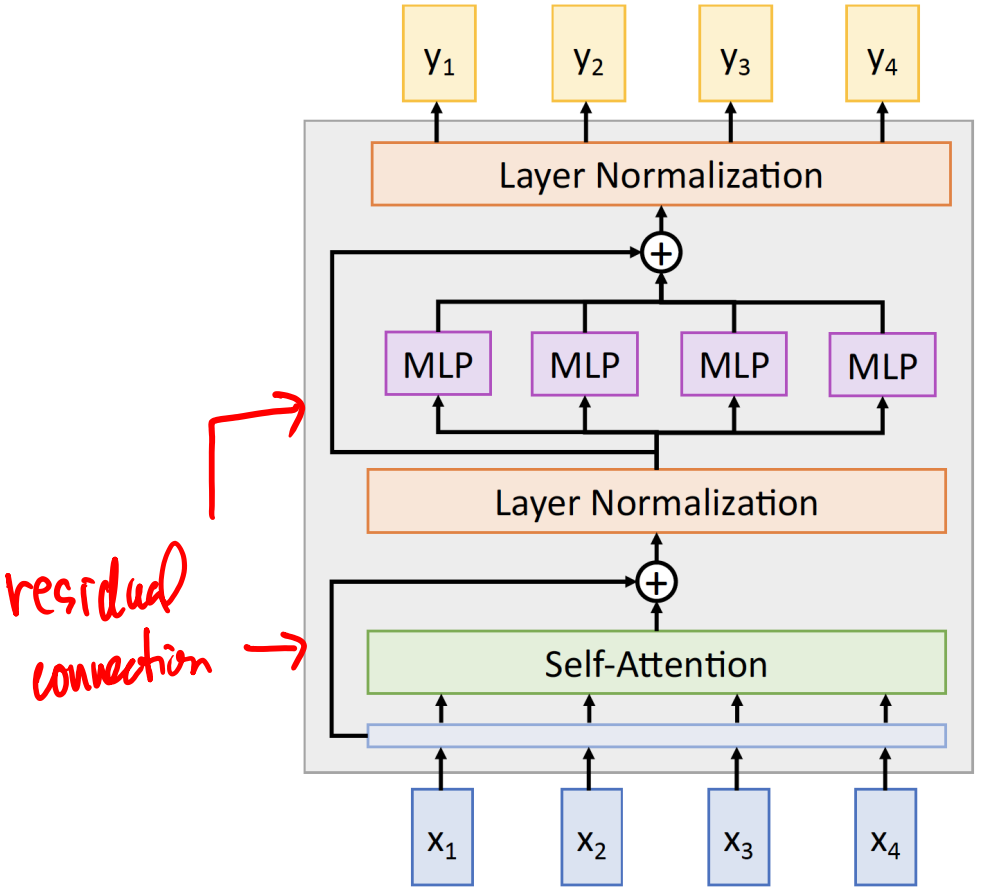
# The Transformer

**Transformer Block:**
**Input**: Set of vectors x
**Output**: Set of vectors y

Self-attention is the only interaction between vectors!

Layer norm and MLP work independently per vector

Highly scalable, highly parallelizable




Vaswani et al, "Attention is all you need", NeurIPS 2017

# The Transformer

**Transformer Block:**
**Input**: Set of vectors x
**Output**: Set of vectors y

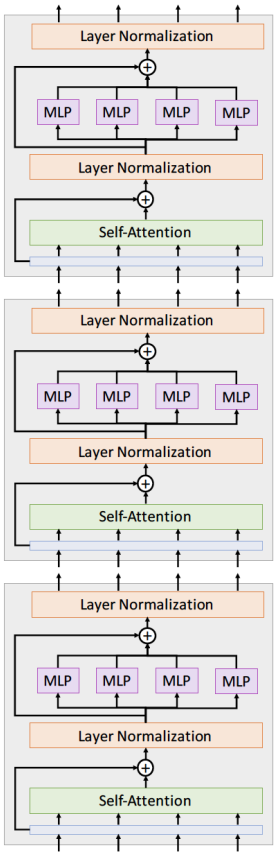
Self-attention is the only interaction between vectors!

Layer norm and MLP work independently per vector

Highly scalable, highly parallelizable

A **Transformer** is a sequence of transformer blocks

Vaswani et al:
12 blocks, $D_Q$=512, 6 heads




Vaswani et al, "Attention is all you need", NeurIPS 2017

# Scaling up Transformers

| Model | Layers | Width | Heads | Params | Data | Training |
|---|---|---|---|---|---|---|
| Transformer-Base | 12 | 512 | 8 | 65M | | 8x P100 (12 hours) |
| Transformer-Large | 12 | 1024 | 16 | 213M | | 8x P100 (3.5 days) |
| BERT-Base | 12 | 768 | 12 | 110M | 13 GB | |
| BERT-Large | 24 | 1024 | 16 | 340M | 13 GB | |
| XLNet-Large | 24 | 1024 | 16 | ~340M | 126 GB | 512x TPU-v3 (2.5 days) |
| RoBERTa | 24 | 1024 | 16 | 355M | 160 GB | 1024x V100 GPU (1 day) |
| GPT-2 | 48 | 1600 | ? | 1.5B | 40 GB | |
| Megatron-LM | 72 | 3072 | 32 | 8.3B | 174 GB | 512x V100 GPU (9 days) |
| Turing-NLG | 78 | 4256 | 28 | 17B | ? | 256x V100 GPU |
| GPT-3 | 96 | 12288 | 96 | 175B | 694GB | **$4.6 million to train!** |

LLM down scaling

LLM Alignment

Brown et al, "Language Models are Few-Shot Learners", arXiv 2020

# Key References


- Introduced transformers: Vaswani et al., “Attention is all you need,” NeurIPS 2017

- Introduced generative transformers: Liu et al., “Generating Wikipedia by Summarizing Long Sequences,” ICLR 2018

- Introduced generative pre-trained transformers (GPT): Radford et al., “Improving language understanding by generative pre-training,” 2018.

- Devlin et al., “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” 2018