

---

# Deep Learning

---

## 6. Sequential Models

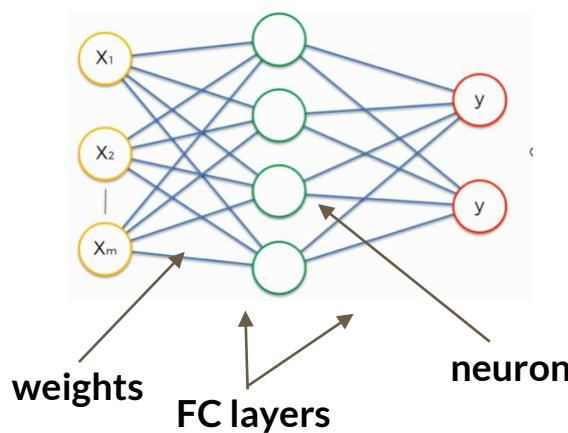
---

fbrad@bitdefender.com

---

# RECAP

## Fully connected network

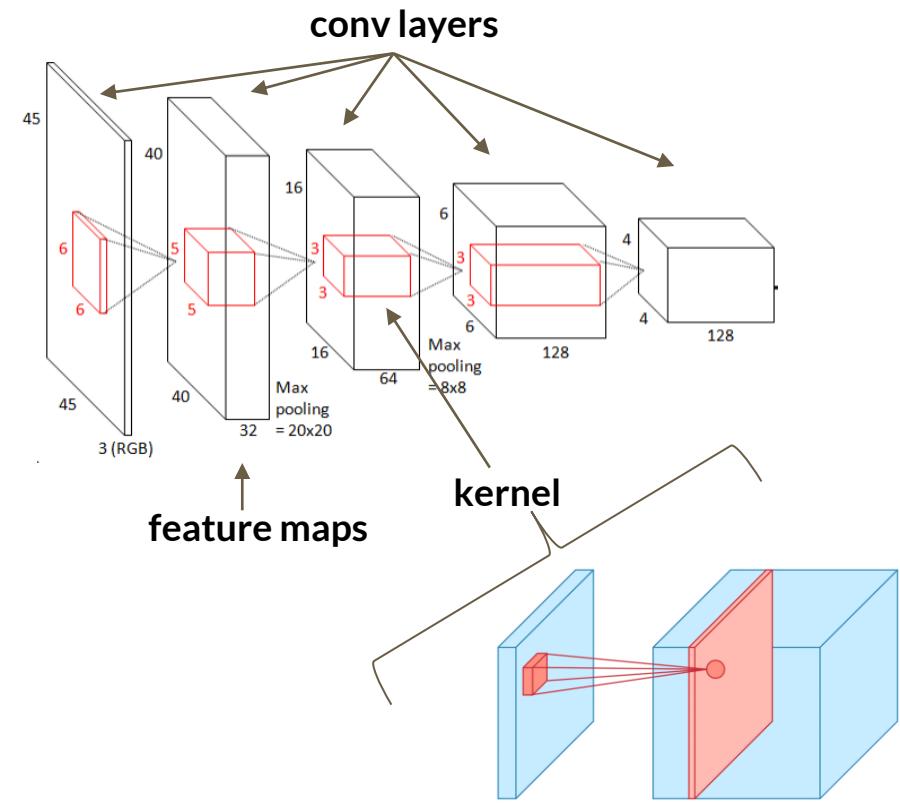


$$a = \sigma(W_1 x + b_1)$$

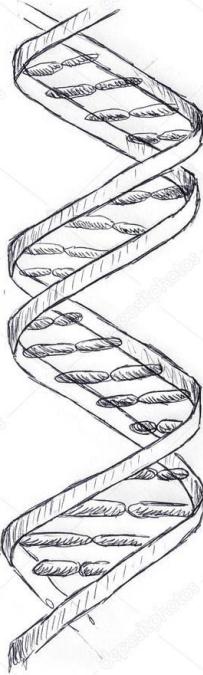
$$p = \sigma(W_2 a + b_2)$$

non-linearity

## ConvNet



# Motivation



- Until now we have:
  - FCN: unstructured data (l)
  - CNN: localized data (right)
- BUT sometimes we have **sequential data**



# Sequences

I love taking photos of nature. Can you come up with a 3-day itinerary visiting the most photogenic places in Iceland?



**ChatGPT**

Absolutely! Iceland is renowned for its breathtaking natural landscapes, making it a paradise for photographers. Here's a 3-day itinerary that will take you to some of the most photogenic spots in Iceland:

## **Day 1: The Golden Circle**

Morning:

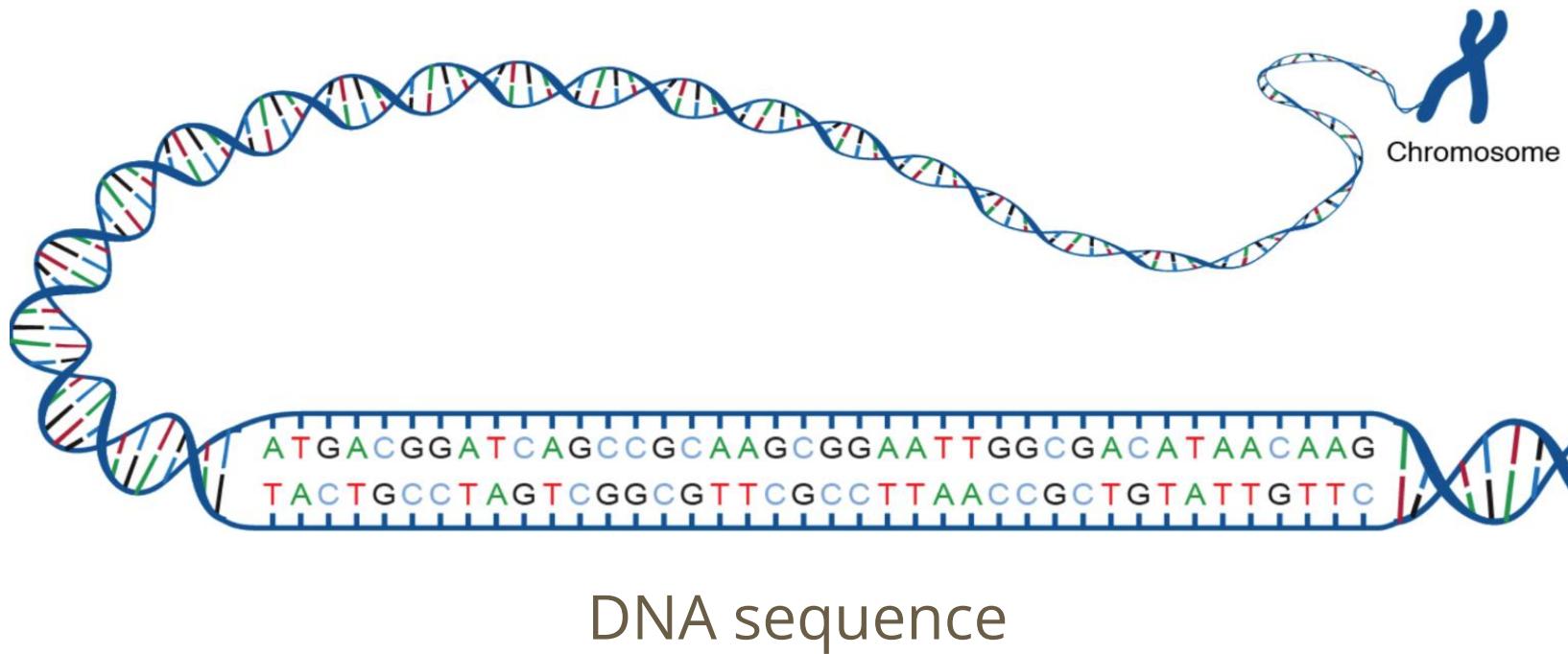
- Start your day early and head to Thingvellir National Park. Capture the stunning scenery of the rift valley formed by the drifting apart of the North American and Eurasian tectonic plates.
- Don't miss the chance to photograph the picturesque Oxararfoss waterfall within the park.

text

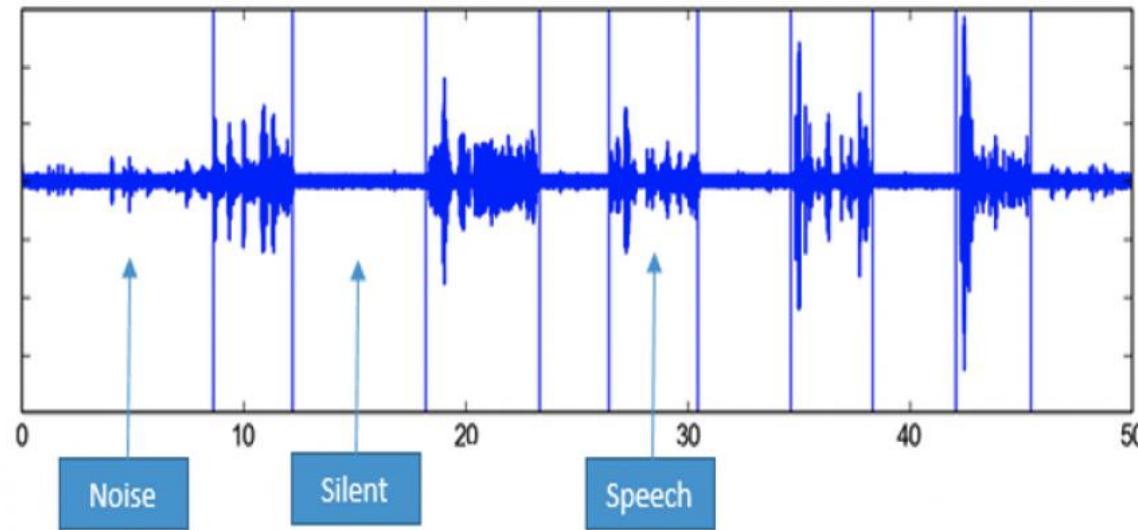
# Sequences



# Sequences



# Sequences



audio

# Sequences



video frames

# Plan for today

- RNNs: a classical approach to sequential data
  - vanilla RNNs
  - learning issues in RNNs
  - gated architectures: Gated Recurrent Units (GRU)
- Transformers: backbone of modern AI models
  - self-attention mechanism
  - other Transformer block components: FFN, Residuals, LayerNorm
- pros and cons of RNNs and Transformers

# Video classification:

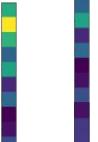
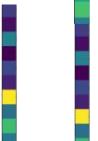
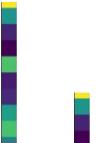


→ EAT

→ SCORE

→ LONG JUMP

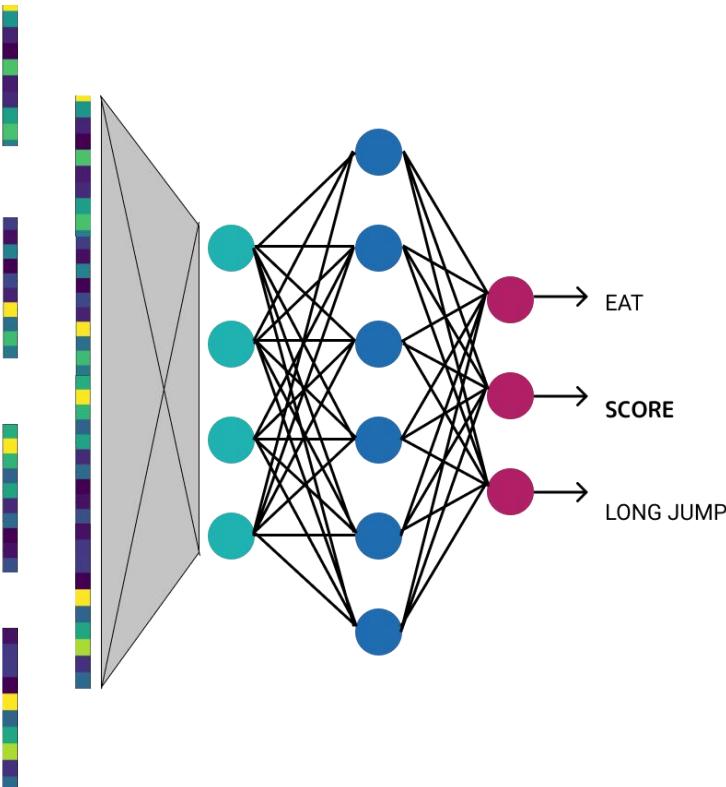
# Video classification:



- A pink circle with an arrow points to the text "EAT".
- A pink circle with an arrow points to the text "SCORE".
- A pink circle with an arrow points to the text "LONG JUMP".

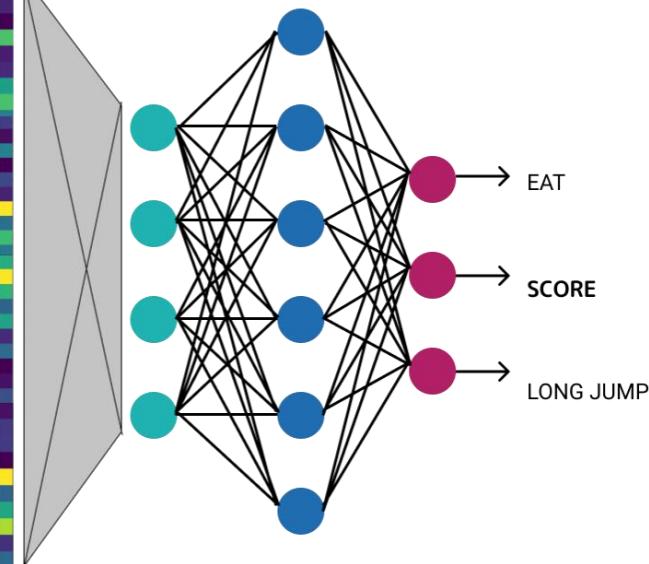
- for each frame extract features using **CNN**
- concatenate all the features in a **huge** video representation vector

# Video classification:



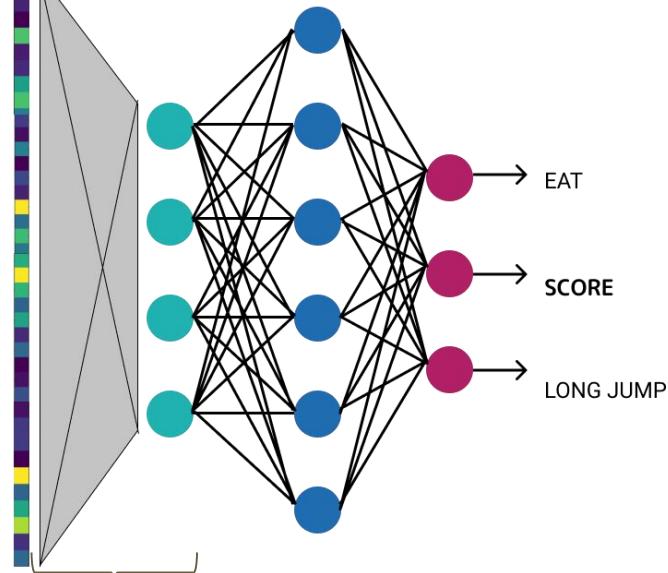
- for each frame extract features using **CNN**
- concatenate all the features in a **huge** video representation vector
- process the video representation using **fully connected layers**

# Video classification:



Why not?

# Video classification:



Why not?

# Video classification:

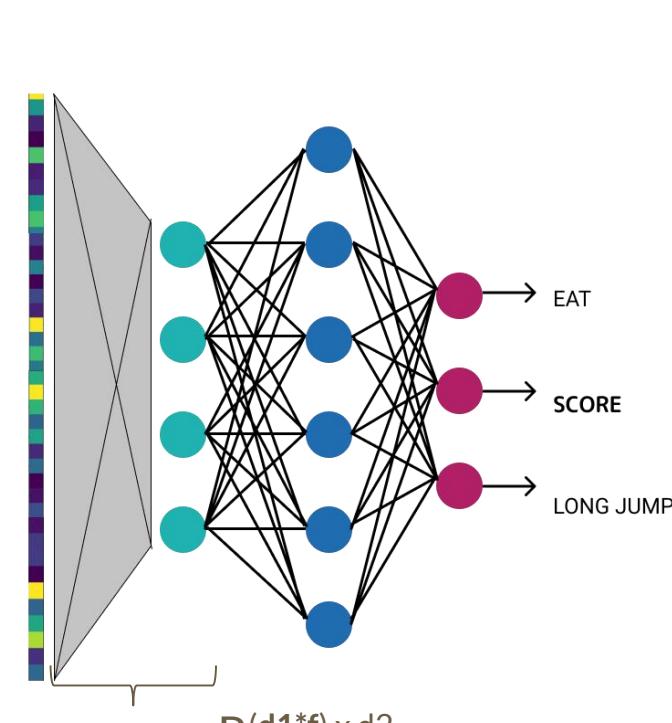


?

- → EAT
- → SCORE
- → LONG JUMP



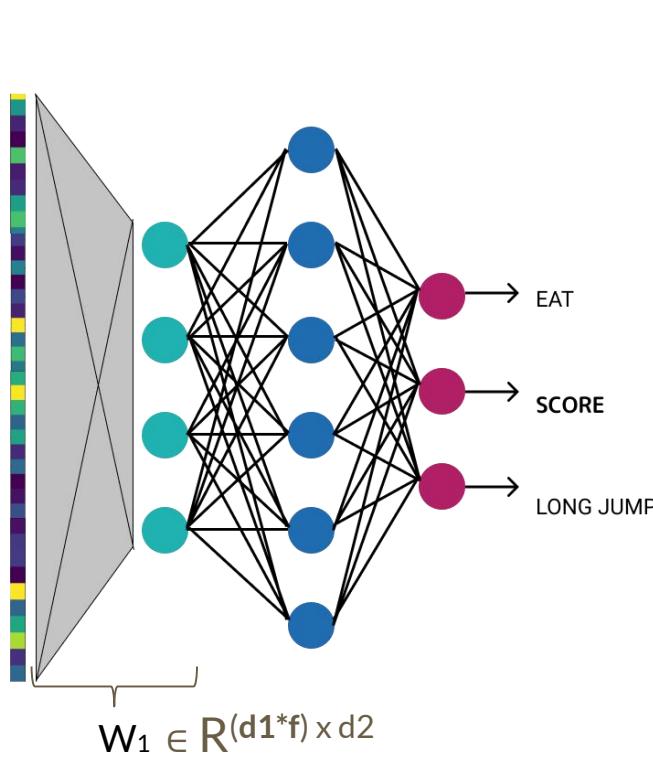
# Video classification:



Why not?

- **huge** number of parameters

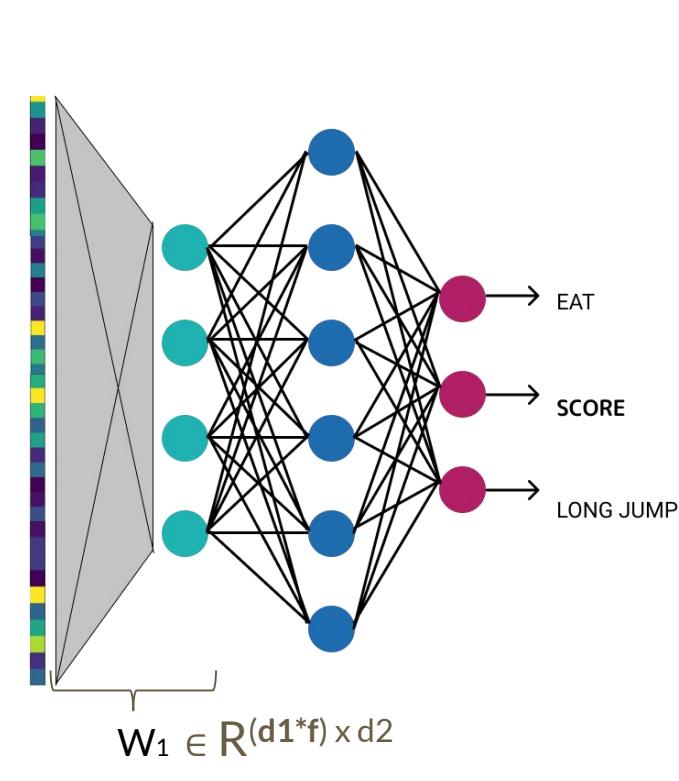
# Video classification:



Why not?

- **huge** number of parameters
- **different** parameters for similar frames/objects

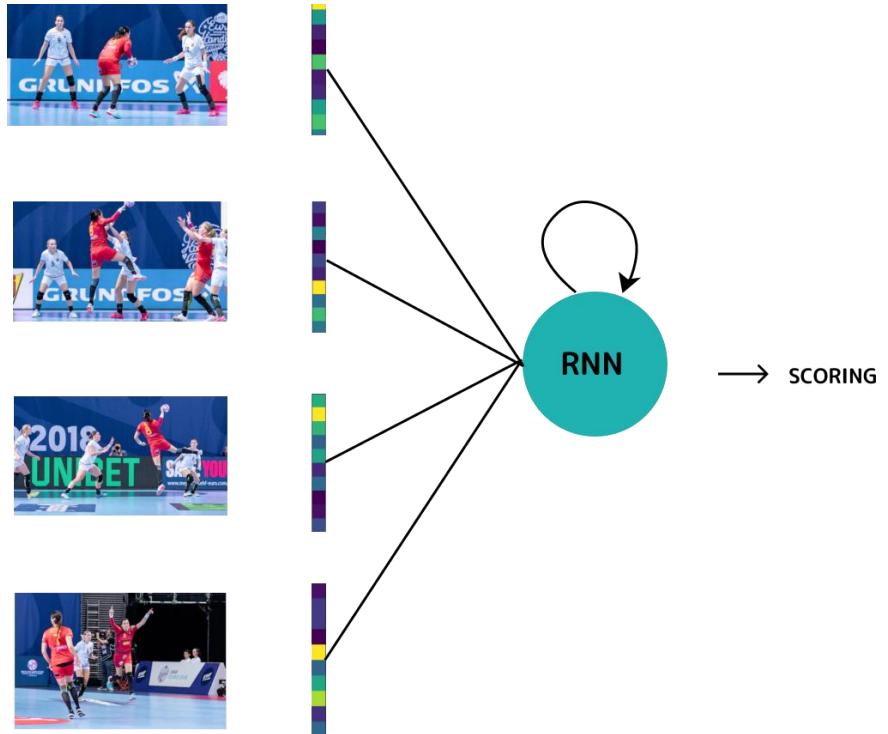
# Video classification:



## Why not?

- **huge** number of parameters
- **different** parameters for similar frames/objects
- **fixed** input length

# Video classification:



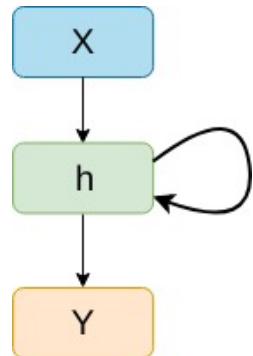
Solution:

**Recurrent Neural Network**

# Recurrent Neural Network

## key ideas:

- process sequence by repeatedly applying the same **recurrence** formula
- [optional] compute output for one or more timesteps



$$x_1, x_2 \dots x_{T_1} \rightarrow y_1, y_2 \dots y_{T_2}$$

$$\begin{aligned} h_t &= f_W(h_{t-1}, x_t) \\ y_t &= f_O(h_t) \end{aligned}$$

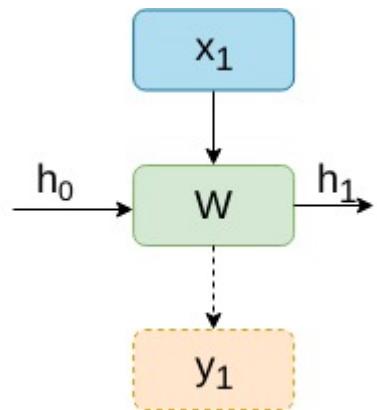
$\mathbf{x}_t$  - *input at step t*

$\mathbf{h}_t$  - *hidden state at step t*

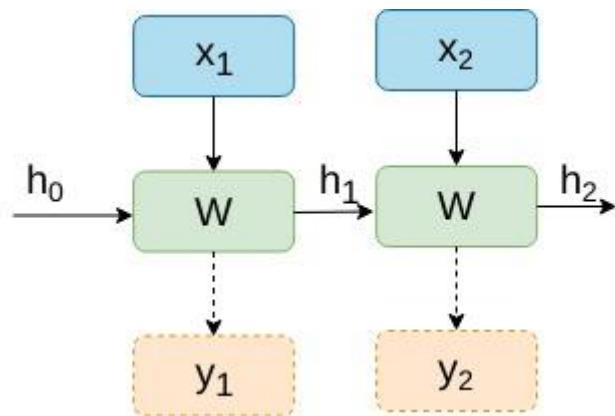
$\mathbf{y}_t$  - [optional] *output at step t*

# Forward step

$$h_1 = f_W(h_0, x_1)$$
$$[y_1 = f_O(h_1)]$$



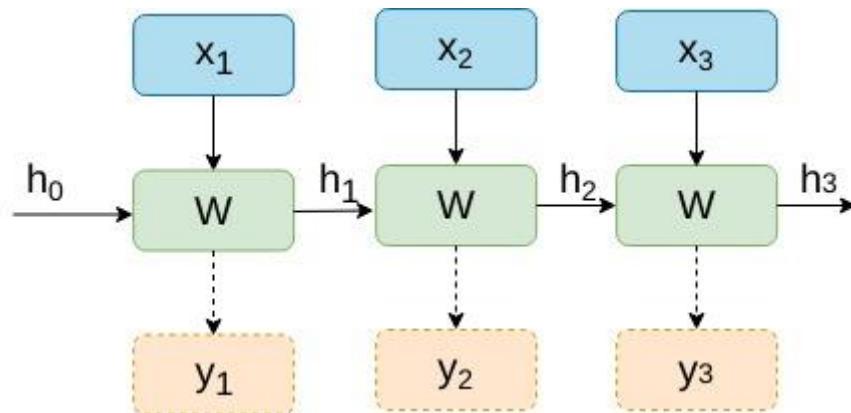
# Forward step



$$h_1 = f_W(h_0, x_1)$$
$$[y_1 = f_O(h_1)]$$

$$h_2 = f_W(h_1, x_2)$$
$$[y_2 = f_O(h_2)]$$

# Forward step

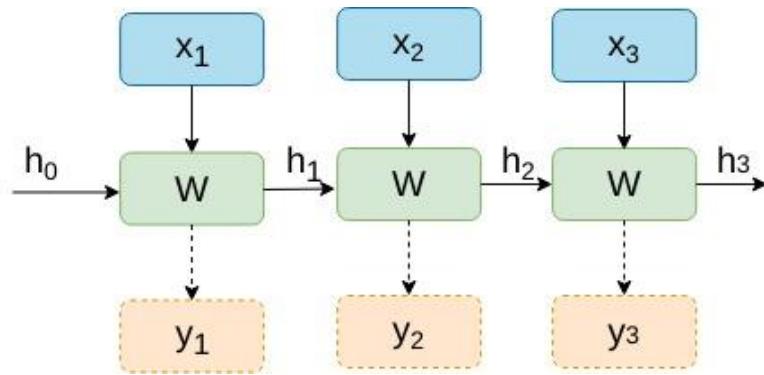


$$h_1 = f_W(h_0, x_1)$$
$$[y_1 = f_O(h_1)]$$

$$h_2 = f_W(h_1, x_2)$$
$$[y_2 = f_O(h_2)]$$

$$h_3 = f_W(h_2, x_3)$$
$$[y_3 = f_O(h_3)]$$

# Forward step



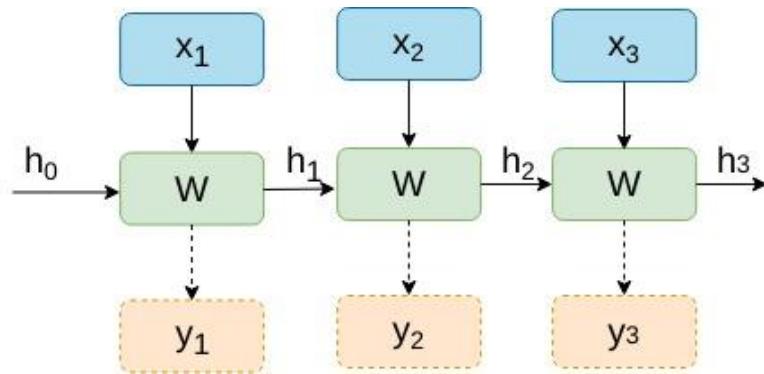
same W / same O at each time step

$$h_1 = f_W(h_0, x_1)$$
$$[y_1 = f_O(h_1)]$$

$$h_2 = f_W(h_1, x_2)$$
$$[y_2 = f_O(h_2)]$$

$$h_3 = f_W(h_2, x_3)$$
$$[y_3 = f_O(h_3)]$$

# Forward step



$$h_1 = f_W(h_0, x_1)$$
$$[y_1 = f_O(h_1)]$$

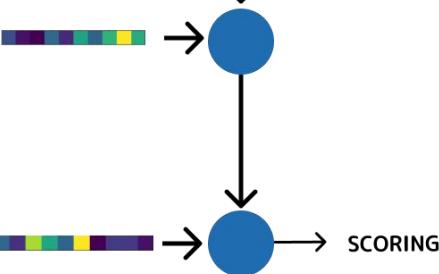
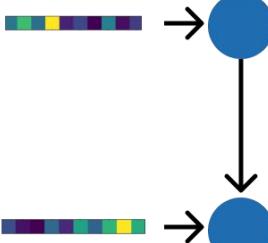
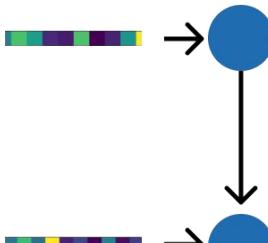
$$h_2 = f_W(h_1, x_2)$$
$$[y_2 = f_O(h_2)]$$

same W / same O at each time step

- allow input sequence of **variable lengths**
- given a context and an input, all timesteps should be **processed in the same way**

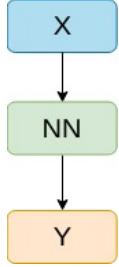
$$h_3 = f_W(h_2, x_3)$$
$$[y_3 = f_O(h_3)]$$

# Video classification: RNN



- one **CNN** to extract features
- one **RNN** to process the frames features
- output from the final step (many to one)

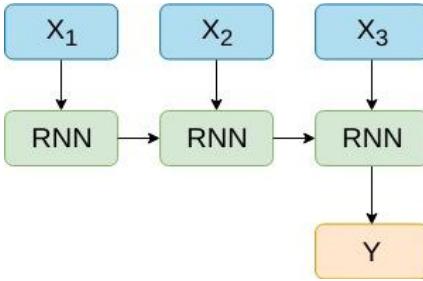
# Sequence modeling applications



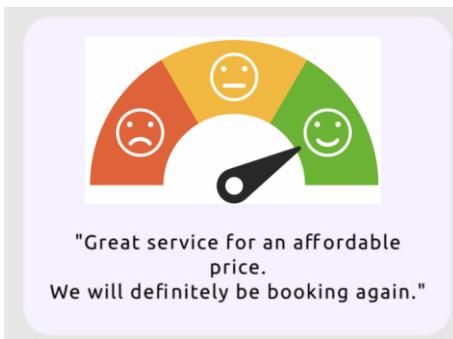
ONE - TO - ONE  
Binary Classification



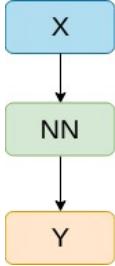
or



MANY - TO - ONE  
Sentiment Analysis  
Video Classification



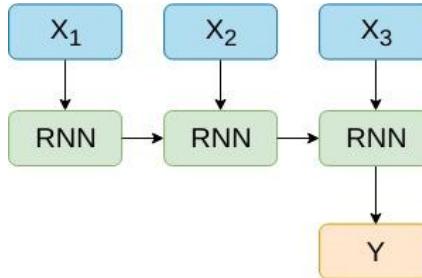
# Sequence modeling applications



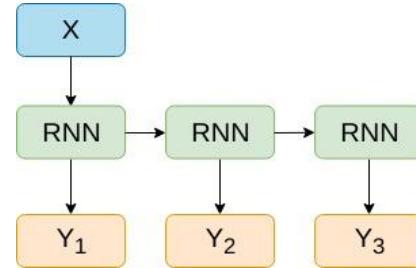
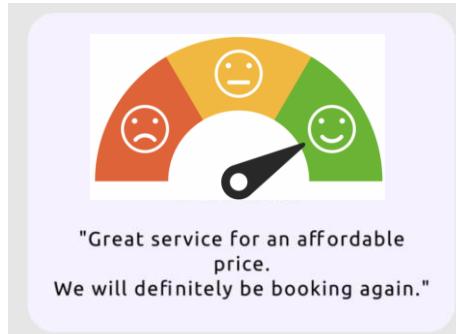
ONE - TO - ONE  
Binary Classification



or



MANY - TO - ONE  
Sentiment Analysis  
Video Classification

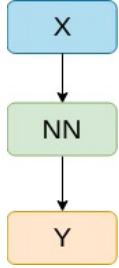


ONE - TO - MANY  
Image Captioning

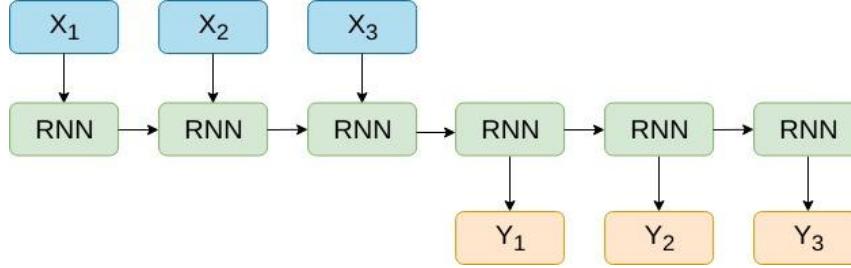


long dog is happy

# Sequence modeling applications



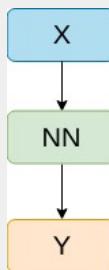
ONE - TO - ONE  
Binary Classification



MANY - TO - MANY  
Machine Translation



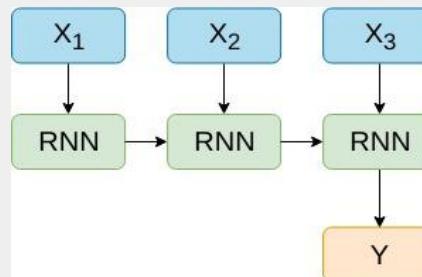
# Feed-forward



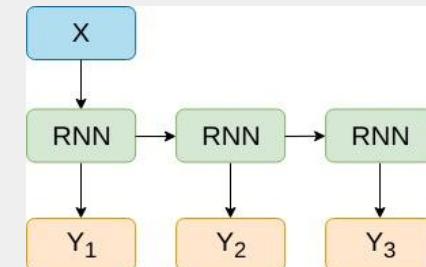
ONE - TO - ONE

vs

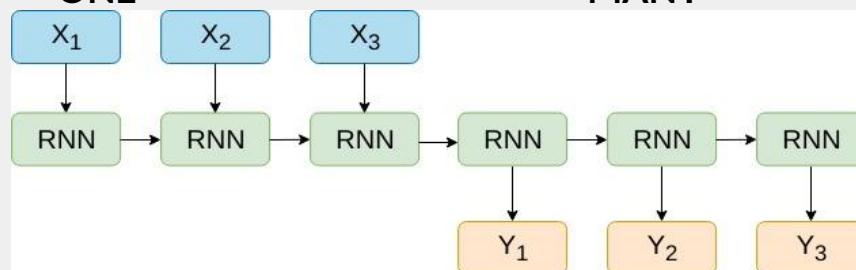
# RNN



MANY - TO -  
ONE

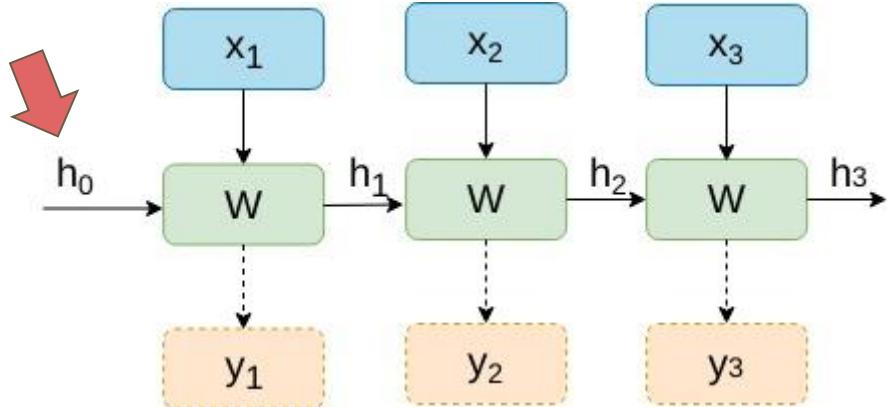


ONE - TO -  
MANY



MANY - TO - MANY

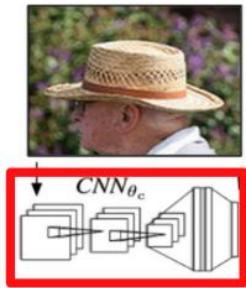
# Initial state



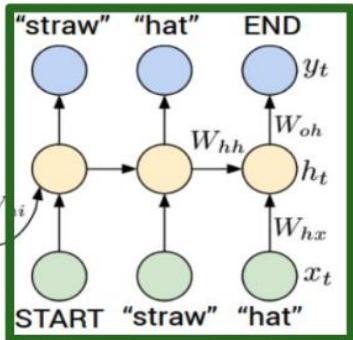
**What about  $h_0$  ?**

1. zeros vector
1. representation of data used as conditional information for current model

# Initial state



Recurrent Neural Network

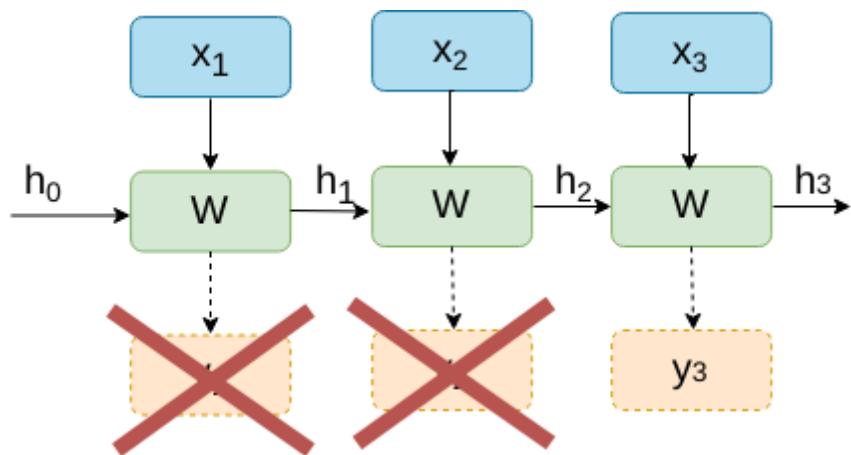


Convolutional Neural Network

What about  $h_0$  ?

- 1. zeros vector
- 1. representation of data used as conditional information for current model

# Forward: single output



$$h_1 = f_W(h_0, x_1)$$

~~$$[y_1 = f_O(h_1)]$$~~

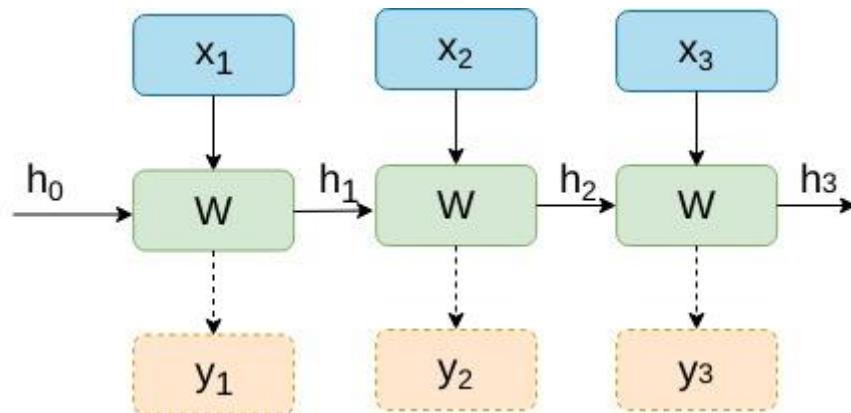
$$h_2 = f_W(h_1, x_2)$$

~~$$[y_2 = f_O(h_2)]$$~~

$$h_3 = f_W(h_2, x_3)$$

$$[y_3 = f_O(h_3)]$$

# Forward: multiple outputs (output labels != input labels)

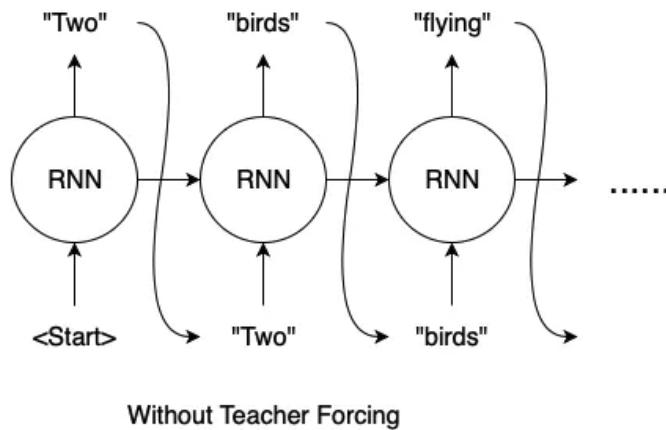


$$h_1 = f_W(h_0, x_1)$$
$$[y_1 = f_O(h_1)]$$

$$h_2 = f_W(h_1, x_2)$$
$$[y_2 = f_O(h_2)]$$

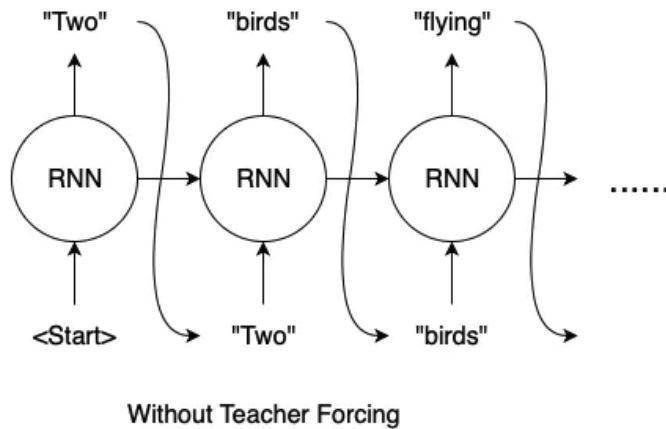
$$h_3 = f_W(h_2, x_3)$$
$$[y_3 = f_O(h_3)]$$

# Forward: multiple outputs (output labels = input labels)



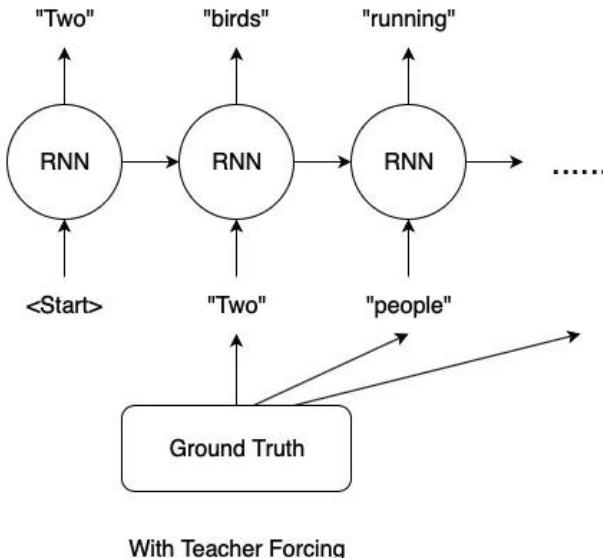
- Training: Why not feed output at time  $t$  as input at  $t+1$ ?

# Forward: multiple outputs (output labels = input labels)



- Training: Why not feed output at time  $t$  as input at  $t+1$ ?
- Early mistakes propagates in later steps:
  - Slow convergence
  - Instability of model

# Teacher forcing



## Teacher forcing:

- during train phase the model receive ground truth instead of model output
  - o later steps receive correct input even in the beginning of training
- during test phase use the model's output, because we don't have access to the ground truth

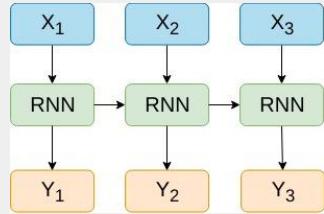
# Vanilla RNN

$$\boxed{\begin{aligned} h_t &= f_W(h_{t-1}, x_t) \\ [y_t &= f_O(h_t)] \end{aligned}}$$

$$\begin{aligned} h_t &= \sigma(W_{hh} \textcolor{green}{h}_{t-1} + W_{xh} \textcolor{blue}{x}_t) &\equiv h_t &= \sigma((W_{hh}; W_{xh}) \begin{pmatrix} \textcolor{green}{h}_{t-1} \\ \textcolor{blue}{x}_t \end{pmatrix}) \\ [y_t &= W_{hy} h_t] && [y_t = W_{hy} h_t] \end{aligned}$$

# Vanilla RNN vs Multi-modal RNN

One-modal input



$$\begin{aligned} h_t &= f_W(h_{t-1}, x_t) \\ [y_t &= f_O(h_t)] \end{aligned}$$

$$h_t = \sigma(W_{hh} \textcolor{green}{h}_{t-1} + W_{xh} \textcolor{blue}{x}_t)$$

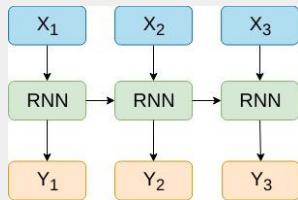
$$[y_t = W_{hy} h_t]$$

$$h_t = \sigma((W_{hh}; W_{xh}) \begin{pmatrix} \textcolor{green}{h}_{t-1} \\ \textcolor{blue}{x}_t \end{pmatrix})$$

$$[y_t = W_{hy} h_t]$$

# Vanilla RNN vs Multi-modal RNN

One-modal input



$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \sigma(W_{hh} \textcolor{green}{h}_{t-1} + W_{xh} \textcolor{blue}{x}_t)$$

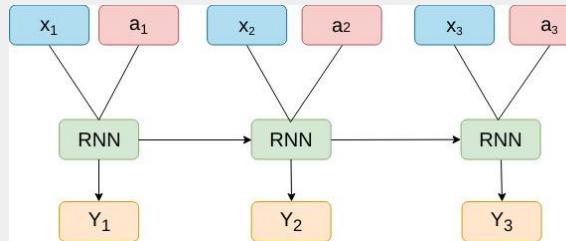
$$[y_t = W_{hy} h_t]$$

$\equiv$

$$h_t = \sigma((W_{hh}; W_{xh}) \begin{pmatrix} \textcolor{green}{h}_{t-1} \\ \textcolor{blue}{x}_t \end{pmatrix})$$

$$[y_t = W_{hy} h_t]$$

Multi-modal input



$$h_t = f_W(h_{t-1}, x_t, a_t \dots)$$

$$h_t = \sigma(W_{hh} \textcolor{green}{h}_{t-1} + W_{xh} \textcolor{blue}{x}_t + W_{ah} \textcolor{red}{a}_t + \dots)$$

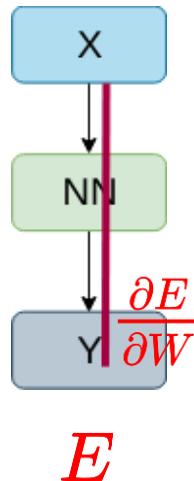
$$[y_t = W_{hy} h_t]$$

$\equiv$

$$h_t = \sigma((W_{hh}; W_{xh}; W_{ah}) \begin{pmatrix} \textcolor{green}{h}_{t-1} \\ \textcolor{blue}{x}_t \\ \textcolor{red}{a}_t \\ \dots \end{pmatrix})$$

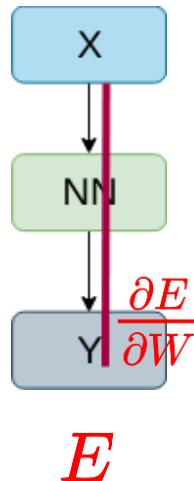
$$[y_t = W_{hy} h_t]$$

# Backpropagation



- Loss function ( $E$ ) measures how close we are to the optimal solution
- At train time, update parameters to minimize the loss
- The update is based on **gradient** descent

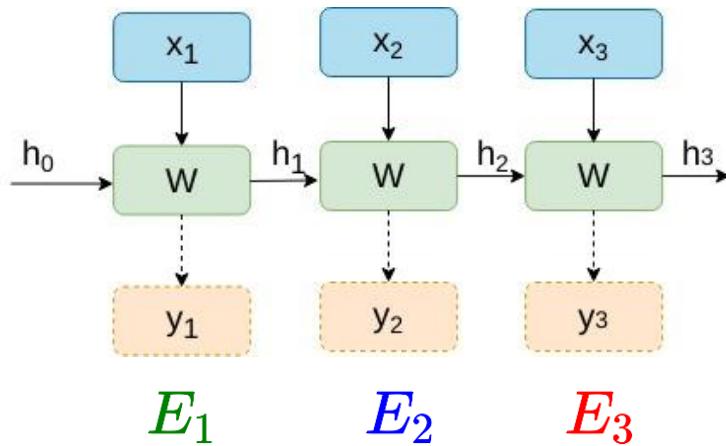
# Backpropagation



- Loss function ( $E$ ) measures how close we are to the optimal solution
- At train time, update parameters to minimize the loss
- The update is based on **gradient** descent

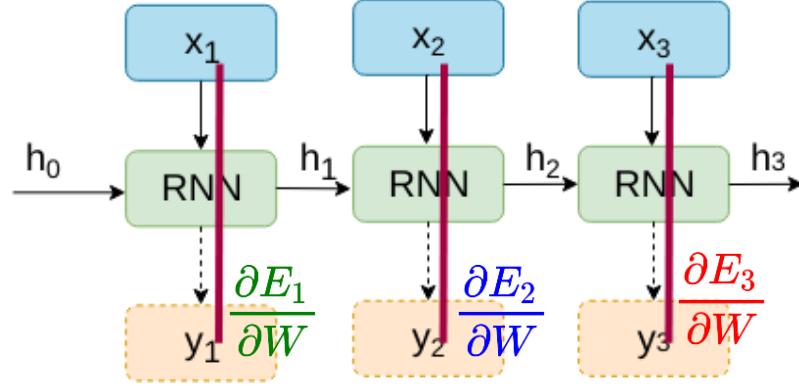
We need to compute  $\frac{\partial E}{\partial W}$

# Backpropagation through time (BPTT)



Fancy name for regular  
backpropagation on an unrolled RNN

# Backpropagation through time (BPTT)

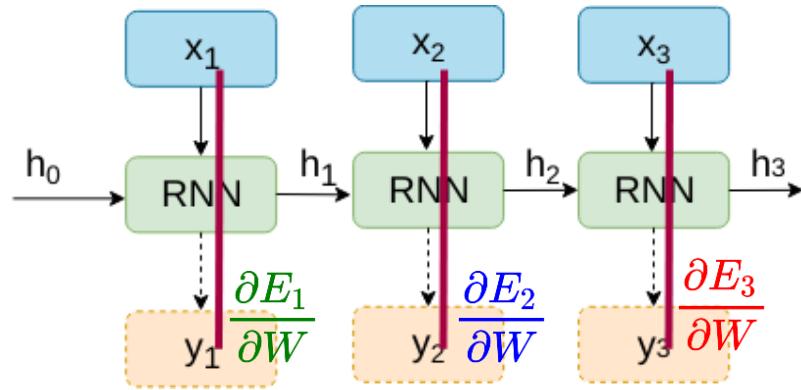


$$h_t = (W_{hh}\sigma(h_{t-1}) + W_{xh}x_t)^*$$
$$[y_t = W_{hy}\sigma(h_t)]$$

$$E = \sum_{t=1}^T E_t$$
$$E_t(y, y^*) = f_{loss}(y_t, y_t^*)$$

\* Slightly change the way we define this, for an easier computation. The intuition remains the same

# Backpropagation through time (BPTT)



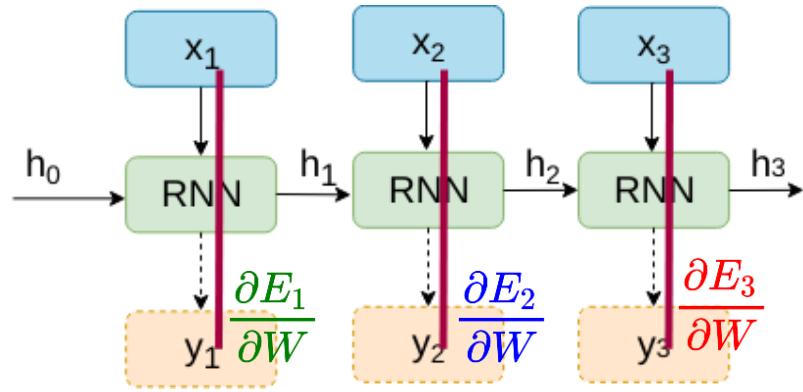
$$h_t = (\mathbf{W}_{hh}\sigma(h_{t-1}) + \mathbf{W}_{xh}x_t)$$
$$[y_t = \mathbf{W}_{hy}\sigma(h_t)]$$

$$E = \sum_{t=1}^T E_t$$

$$E_t(y, y^*) = f_{loss}(y_t, y_t^*)$$

$$\frac{\partial E}{\partial \mathbf{W}} = \frac{\partial}{\partial \mathbf{W}} \left( \sum_{t=1}^T E_t \right) = \sum_{t=1}^T \frac{\partial E_t}{\partial \mathbf{W}}$$

# Backpropagation through time (BPTT)



$$h_t = (\mathbf{W}_{hh} \sigma(h_{t-1}) + \mathbf{W}_{xh} x_t) = f(\mathbf{W}, h_{t-1}, x_t)$$

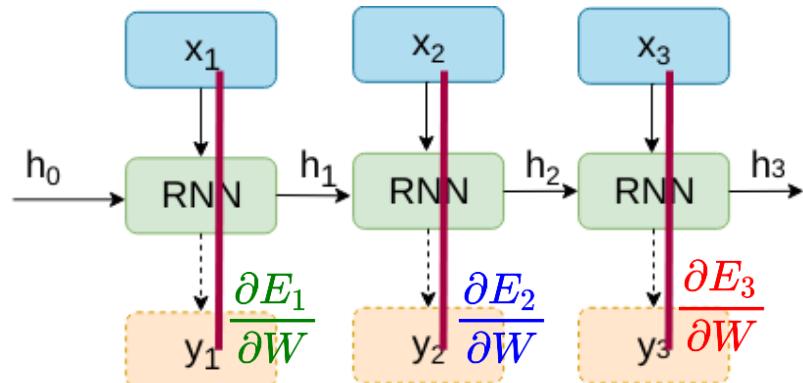
$$y_t = W_{hy} \sigma(h_t) = f(h_t)$$

$$\mathcal{E}_t(y, y^*) = f_{loss}(y_t, y_t^*)$$

$$\frac{\partial \mathcal{E}_t}{\partial \mathbf{W}} = \frac{\partial \mathcal{E}_t}{\partial y_t} \frac{\partial y_t}{\partial \mathbf{W}}$$

$$\frac{\partial \mathcal{E}_t}{\partial \mathbf{W}} = \frac{\partial \mathcal{E}_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial \mathbf{W}}$$

# Backpropagation through time (BPTT)



$$h_t = (\mathbf{W}_{hh} \sigma(h_{t-1}) + \mathbf{W}_{xh} x_t) = f(\mathbf{W}, h_{t-1}, x_t)$$

$$y_t = W_{hy} \sigma(h_t) = f(h_t)$$

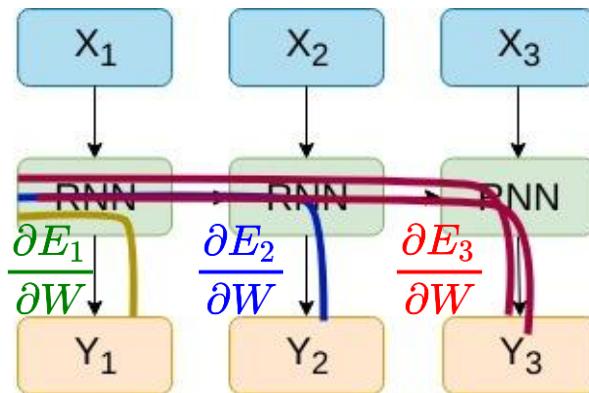
$$\mathcal{E}_t(y, y^*) = f_{loss}(y_t, y_t^*)$$

$$\frac{\partial \mathcal{E}_t}{\partial \mathbf{W}} = \frac{\partial \mathcal{E}_t}{\partial y_t} \frac{\partial y_t}{\partial \mathbf{W}}$$

$$\frac{\partial \mathcal{E}_t}{\partial \mathbf{W}} = \frac{\partial \mathcal{E}_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial \mathbf{W}}$$

No!

# Backpropagation through time (BPTT)



YES!

$$h_{t-1} = (W_{hh}\sigma(h_{t-2}) + W_{xh}x_{t-1})$$

$$h_t = (W_{hh}\sigma(h_{t-1}) + W_{xh}x_t) = f(W, h_{t-1}, x_t)$$

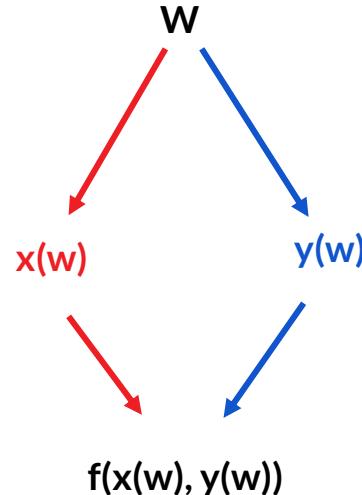
$$y_t = W_{hy}\sigma(h_t) = f(h_t)$$

$$E_t(y, y^*) = f_{loss}(y_t, y^*)$$

$$\cancel{\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial W}}$$

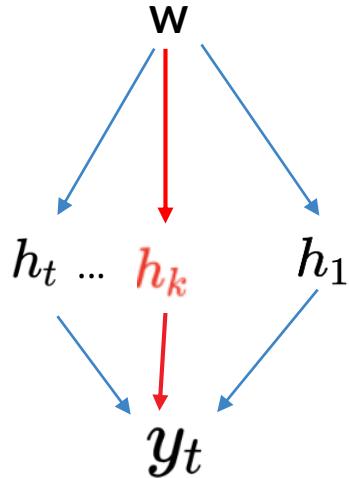
$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^t \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

# Multivariable Chain Rule



$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial x} \frac{\partial x}{\partial w} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial w}$$

# Multivariable Chain Rule

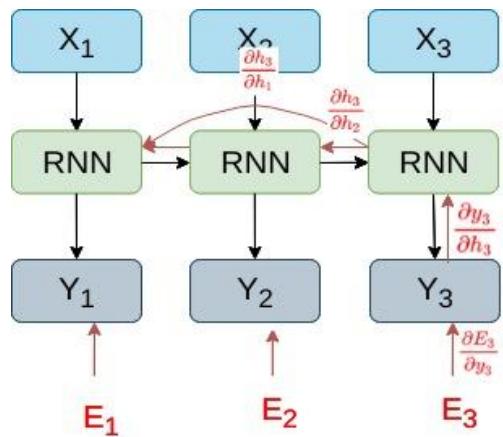


$$h_t = (W_{hh}\sigma(h_{t-1}) + W_{xh}x_t) = f(W, h_{t-1}(W), x_t)$$

$$y_t = W_{hy}\sigma(h_t) = f(h_t)$$

$$\frac{\partial y_t}{\partial W} = \sum_{k=1}^t \frac{\partial y_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

# Backpropagation through time (BPTT)



$$h_t = (W_{hh} \sigma(h_{t-1}) + W_{xh} x_t)$$

$$[y_t = W_{hy} \sigma(h_t)]$$

$$E_t(y^*, y) = f_{loss}(y_t^*, y_t)$$

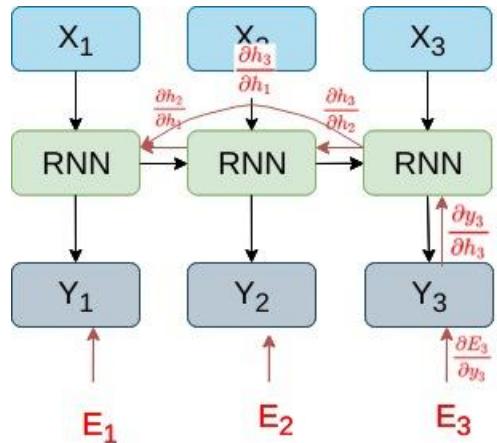
$$E = \sum_{t=1}^T E_t$$

$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial W}$$

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^t \frac{\partial E_t}{\partial y_t} \boxed{\frac{\partial y_t}{\partial h_k}} \frac{\partial h_k}{\partial W}$$

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^t \frac{\partial E_t}{\partial y_t} \boxed{\frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k}} \frac{\partial h_k}{\partial W}$$

# Backpropagation through time (BPTT)



$$h_t = (W_{hh} \sigma(h_{t-1}) + W_{xh} x_t)$$

$$[y_t = W_{hy} \sigma(h_t)]$$

$$E_t(y^*, y) = f_{loss}(y_t^*, y_t)$$

$$E = \sum_{t=1}^T E_t$$

$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial W}$$

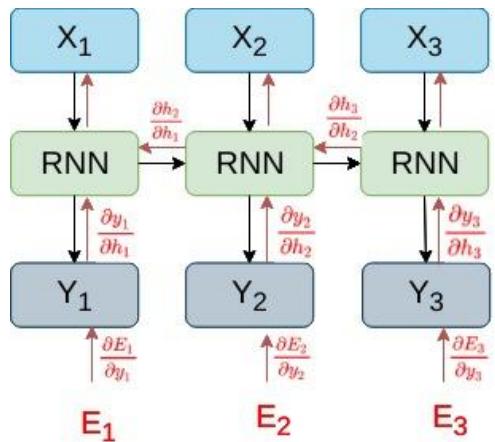
$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^t \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^t \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \boxed{\frac{\partial h_t}{\partial h_k}} \frac{\partial h_k}{\partial W}$$

$$= \sum_{k=1}^t \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \left( \prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial W}$$

$$= \sum_{k=1}^t \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \left( \prod_{j=k+1}^t (W^T \text{diag}[\sigma'(h_{j-1})]) \right) \frac{\partial h_k}{\partial W}$$

# Backpropagation through time



$$\begin{aligned} \frac{\partial E}{\partial W} &= \sum_{k=1}^t \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \times \\ &\times \left( \prod_{j=k+1}^t (W^T \text{diag}[\sigma'(h_{j-1})]) \right) \frac{\partial h_k}{\partial W} \end{aligned}$$

$$\begin{aligned} \frac{\partial h_t}{\partial h_k} &= \prod_{j=k+1}^t [W^T \text{diag}[\sigma'(h_{j-1})]] \\ \left| \frac{\partial h_t}{\partial h_k} \right| &\leq \prod_{j=k+1}^t |W^T| |\text{diag}[\sigma'(h_{j-1})]| \end{aligned}$$

# Backpropagation through time

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^t [W^T \text{diag}[\sigma'(h_{j-1})]]$$

$$|\frac{\partial h_t}{\partial h_k}| \leq \prod_{j=k+1}^t |W^T| |\text{diag}[\sigma'(h_{j-1})]|$$

For large number of timesteps (t):

if  $|W^T| < 1 \rightarrow 0$  vanishing gradients

# Backpropagation through time

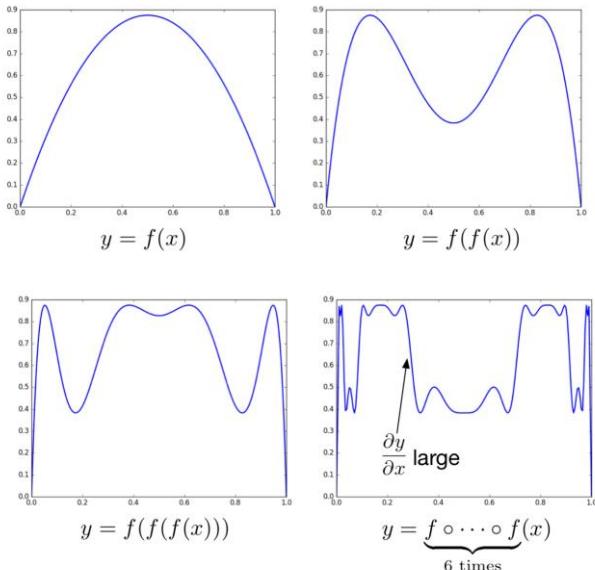


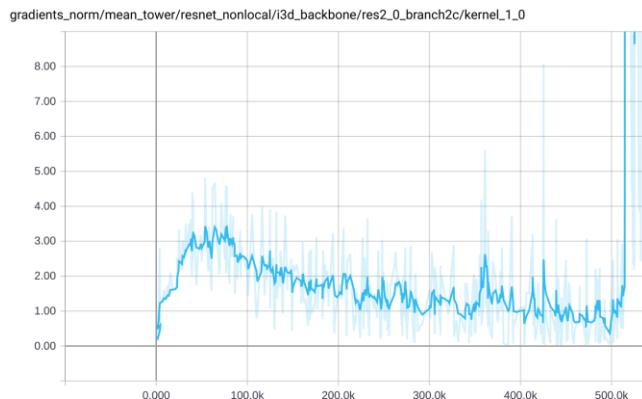
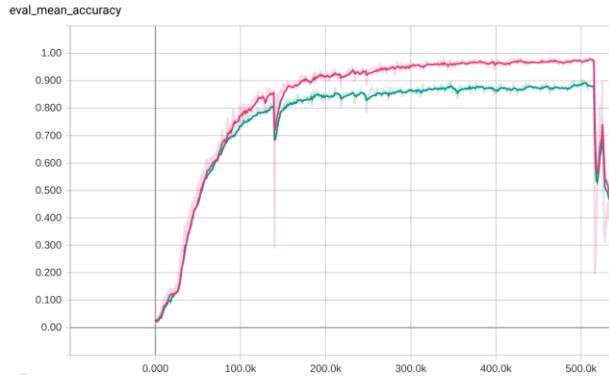
Figure 2: Iterations of the function  $f(x) = 3.5x(1-x)$ .

$$\left| \frac{\partial h_t}{\partial h_k} \right| \leq \prod_{j=k+1}^t |W^T| |diag[\sigma'(h_{j-1})]|$$

For large number of timesteps (t):

$\begin{cases} \text{if } |W^T| < 1 \rightarrow 0 & \text{vanishing gradients} \\ \text{if } |W^T| > 1 \rightarrow \infty & \text{exploding gradients} \end{cases}$

# Backpropagation through time



$$\left| \frac{\partial h_t}{\partial h_k} \right| \leq \prod_{j=k+1}^t |W^T| |diag[\sigma'(h_{j-1})]|$$

For large number of timesteps (t):

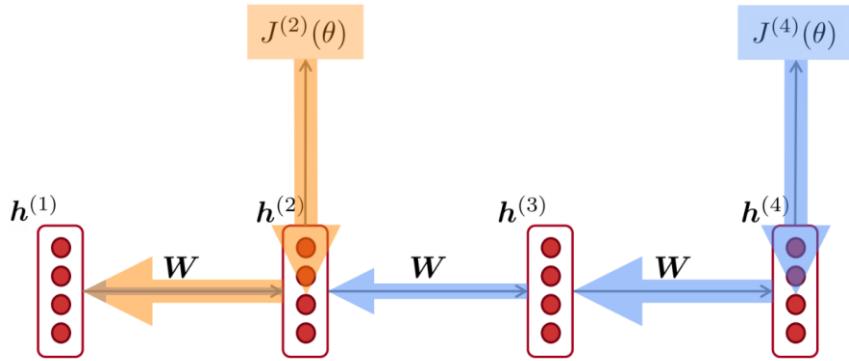
$\begin{cases} \text{if } |W^T| < 1 \rightarrow 0 & \text{vanishing gradients} \\ \text{if } |W^T| > 1 \rightarrow \infty & \text{exploding gradients} \end{cases}$

**HARD TO TRAIN**

**WHY?**



# Vanishing gradients

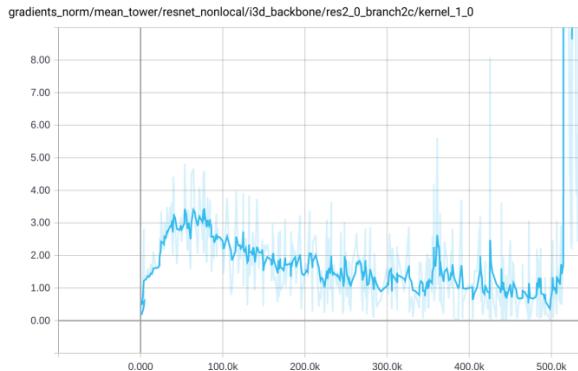
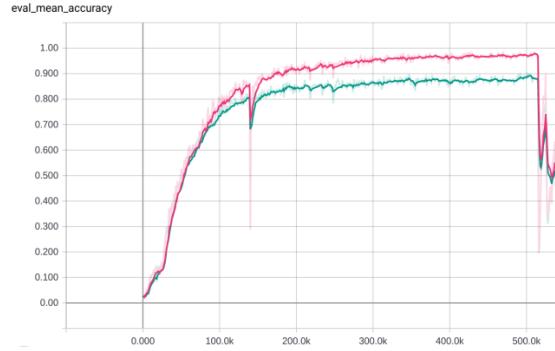


- unable to capture long-distance dependencies
- induce a bias towards recent events

Gradient can be seen as a measure for the impact of the past state for a future state



# Exploding gradients



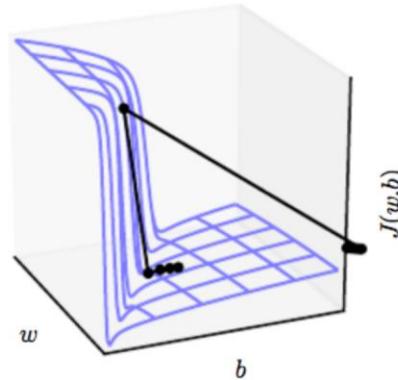
- Bad updates
- Huge step reaching a bad configuration
- Could result in NaN parameters

# Vanishing / exploding gradients

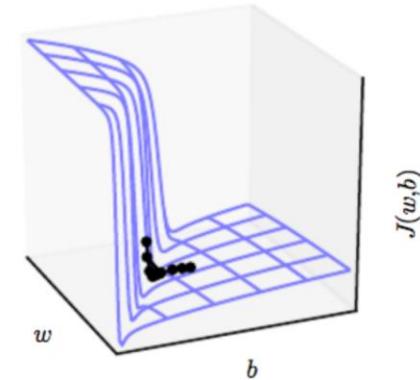
## Solution 1: Gradient clipping

$$g \leftarrow \begin{cases} \frac{\eta g}{|g|} & \text{if } |g| > \eta \\ g & \text{otherwise} \end{cases}$$

Without clipping



With clipping

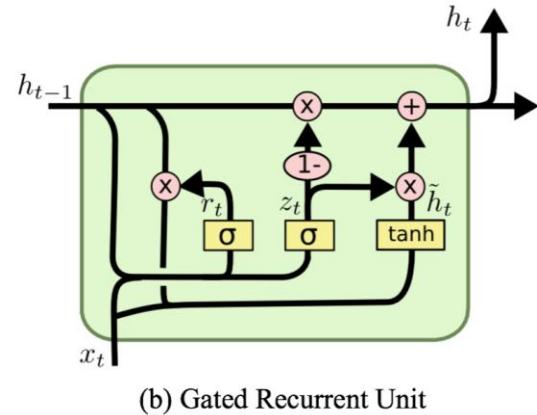
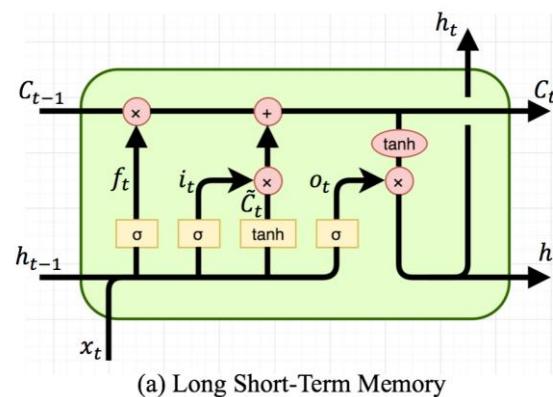


- avoid exploding gradients
- preserve direction, but a smaller step

# Vanishing / exploding gradients

## Solution 2: Architectures:

- LSTM
- GRU



- avoid vanishing gradients

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (November 1997)

Cho, Kyunghyun et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation." *EMNLP* (2014)

# Gates

- **gate:** vector  $\mathbf{g} \in [0,1]^N$  that *filters information* for input  $\mathbf{x}$
- **gated output:**  $\mathbf{y} = \mathbf{g} \odot \mathbf{x}$
- examples of fixed gates:
  - identity:  $\mathbf{g} = [1, 1, 1, \dots, 1]$
  - reset even positions:  $\mathbf{g} = ?$

# Gates

- **gate:** vector  $\mathbf{g} \in [0,1]^N$  that *filters information* for input  $\mathbf{x}$
- **gated output:**  $\mathbf{y} = \mathbf{g} \odot \mathbf{x}$
- examples of fixed gates:
  - identity:  $\mathbf{g} = [1, 1, 1, \dots, 1]$
  - reset even positions:  $\mathbf{g} = [1, 0, 1, 0 \dots, 1, 0]$
  - halve values:  $\mathbf{g} = [0.5, 0.5, \dots, 0.5]$

# Gates

- **gate:** vector  $\mathbf{g} \in [0,1]^N$  that *filters information* for input  $\mathbf{x}$
- **gated output:**  $\mathbf{y} = \mathbf{g} \odot \mathbf{x}$
- examples of fixed gates:
  - identity:  $\mathbf{g} = [1, 1, 1, \dots, 1]$
  - reset even positions:  $\mathbf{g} = [1, 0, 1, 0 \dots, 1, 0]$
  - halve values:  $\mathbf{g} = [0.5, 0.5, \dots, 0.5]$
- we want to **learn gates** to induce certain flow in RNNs
  - $\mathbf{g} = \sigma(f_w(\mathbf{x}))$

# Gating mechanism

## Vanilla RNN

$$h_t = \sigma(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$[y_t = W_{hy}h_t]$$

## Simple Gate

$$g_t = \sigma(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$c_t = g_t \odot c_{t-1} + (1 - g_t) \odot x_t$$
$$h_t = \tanh(c_t)$$
$$y_t = W_{hy}h_t$$

- **Gates** control the flow of information into the cell, allowing the model to *forget/remember* information adaptively:
  - $g_t = [1, 1, \dots, 1] \Rightarrow ?$

# Gating mechanism

## Vanilla RNN

$$h_t = \sigma(W_{hh} h_{t-1} + W_{xh} x_t)$$
$$[y_t = W_{hy} h_t]$$

## Simple Gate

$$g_t = \sigma(W_{hh} h_{t-1} + W_{xh} x_t)$$
$$c_t = g_t \odot c_{t-1} + (1 - g_t) \odot x_t$$
$$h_t = \tanh(c_t)$$
$$y_t = W_{hy} h_t$$

- **Gates** control the flow of information into the cell, allowing the model to *forget/remember* information adaptively:
  - **g<sub>t</sub>=[1,1,...,1]** => **retain previous state** and discard current step
  - **g<sub>t</sub>=[0,0,...,0]** => ?

# Gating mechanism

## Vanilla RNN

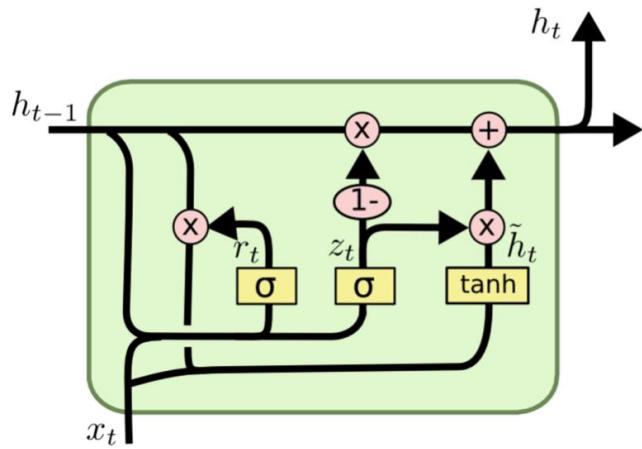
$$h_t = \sigma(W_{hh} h_{t-1} + W_{xh} x_t)$$
$$[y_t = W_{hy} h_t]$$

## Simple Gate

$$g_t = \sigma(W_{hh} h_{t-1} + W_{xh} x_t)$$
$$c_t = \underbrace{g_t}_{\textcircled{g_t}} \odot c_{t-1} + \underbrace{(1 - g_t)}_{\textcircled{1-g_t}} \odot x_t$$
$$h_t = \tanh(c_t)$$
$$y_t = W_{hy} h_t$$

- **Gates** control the flow of information into the cell, allowing the model to *forget/remember* information adaptively:
  - **$g_t=[1,1,\dots,1]$**  => **retain previous state** and discard current step
  - **$g_t=[0,0,\dots,0]$**  => **discard previous state** and retain current step

# GRU (Gated Recurrent Unit, Chung et al. 2014)



GRU use a gating mechanism with 2 gates:

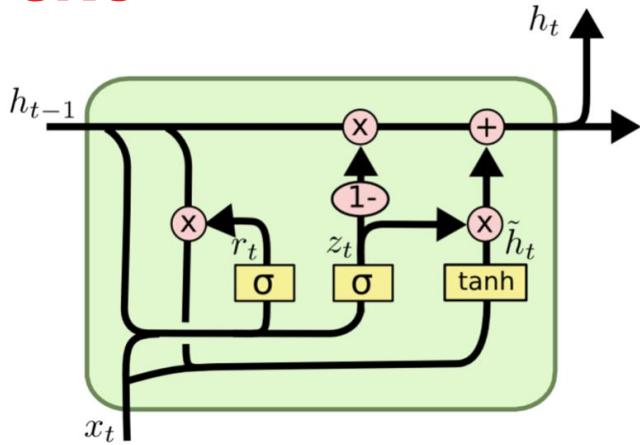
- a. **update gate**: determine how much of the past information needs to be passed along to the future

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

- a. **reset gate**: determine what parts of previous hidden state are used to compute new content

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

# GRU



GRU use a gating mechanism with 2 gates:

- update gate:** determine how much of the past information needs to be passed along to the future

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

- reset gate:** decide how to combine new input with previous state

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

selects useful parts of prev hidden state to compute crt

integrate crt info into the previous one

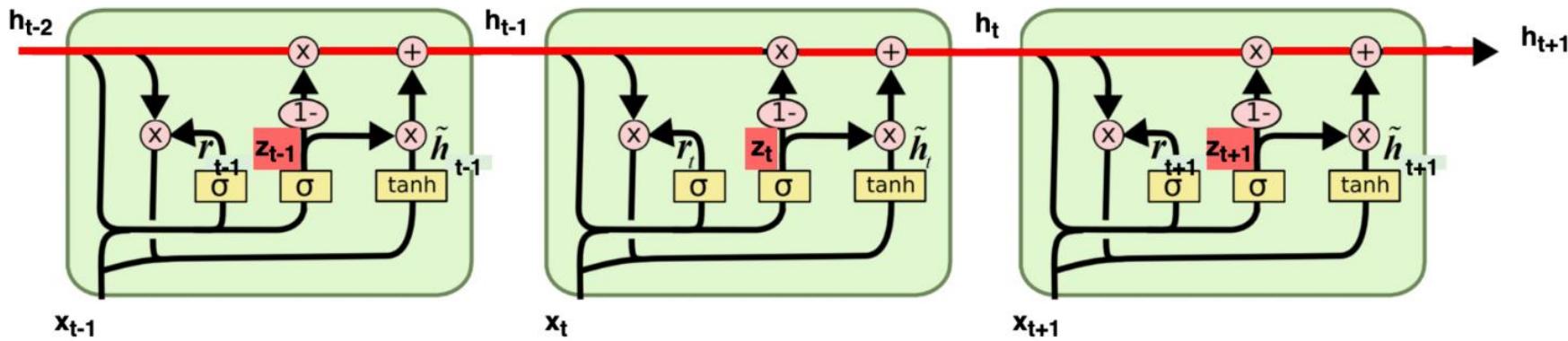
$$\tilde{h}_t = \tanh(Wx_t + r_t \odot Uh_{t-1})$$

keep/reset previous info for current target state

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$

copy/ignore previous info

# Why do gates prevent vanishing?



$$\frac{\partial h_t}{\partial h_j} \approx \prod_{i=j+1}^t (\text{diag}(z_i) + \text{other terms})$$

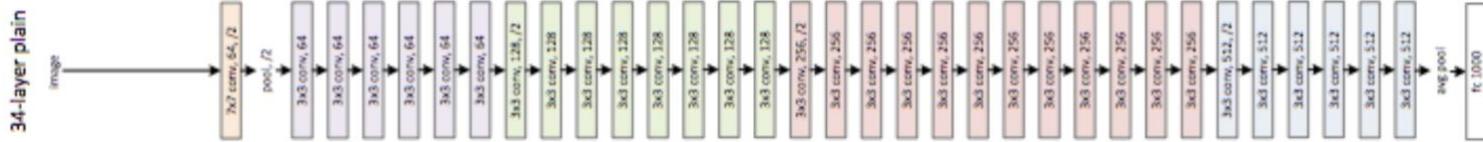
if gates  $z_i$  close to 1  $\Rightarrow dh_i/dh_{i-1} \sim 1$   
 $\Rightarrow dh_t/dh_j$  larger  $\Rightarrow$  better gradient propagation with distance

$$\tilde{h}_t = \tanh(Wx_t + r_t \odot Uh_{t-1})$$

$$h_t = \boxed{z_t} \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$

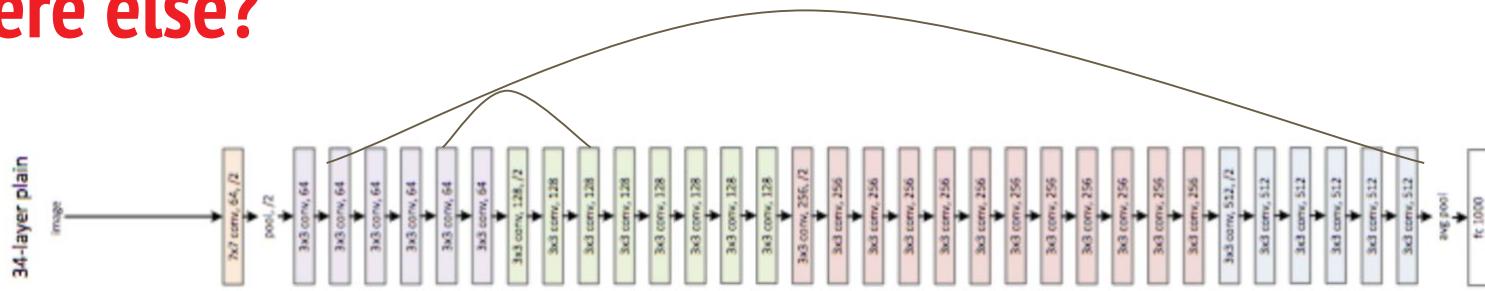
# Where else?

# Where else?



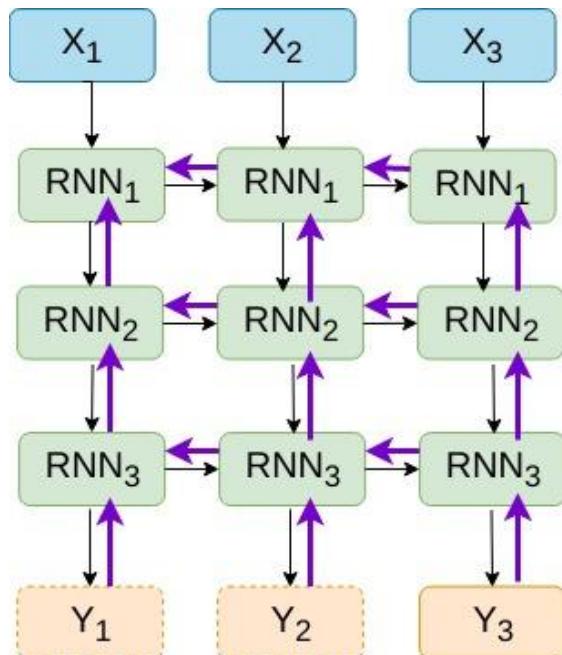
- can be a problem for any neural network
  - use skip-connection to preserve identity
  - Several architectures: *ResidualNet*; *DenseNet*; *HighwayNet*

# Where else?



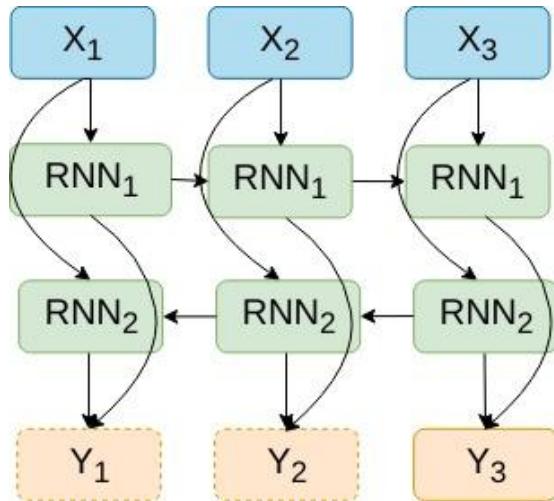
- can be a problem for any neural network
  - use skip-connection to preserve identity
  - Several architecture: *ResidualNet*; *DenseNet*; *HighwayNet*

# Adding depth => Multilayer RNN



- better capture the structure of the input sequence
- at most 2-6 layers: harder to optimize for higher depths (longer paths)

# Bidirectional RNN



$$\bar{h}_t = \sigma(\bar{W}_{hh}\bar{h}_{t-1} + \bar{W}_{xh}x_t)$$

$$\tilde{h}_t = \sigma(\tilde{W}_{hh}\tilde{h}_{t+1} + \tilde{W}_{xh}x_t)$$

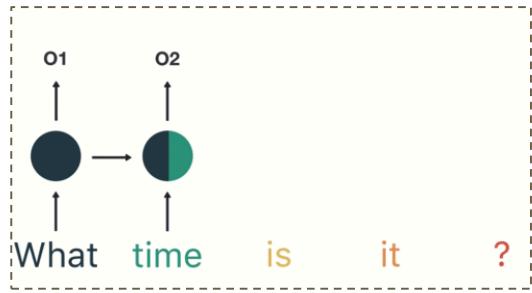
$$[y_t = W_{hy}[\bar{h}_t; \tilde{h}_t]]$$

- **key idea:** train 2 RNNs in both directions
- at each timestep  $t$ , combine hidden states from both **forward** RNN and **backward** RNN

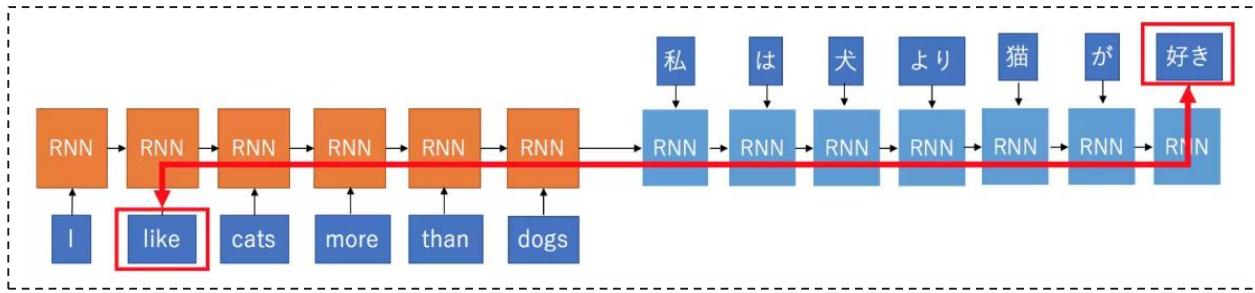
# RNNs pros

- can handle input of variable length
- each step can depend on context from many steps back
- symmetric, intuitive processing

# Why not RNNs?



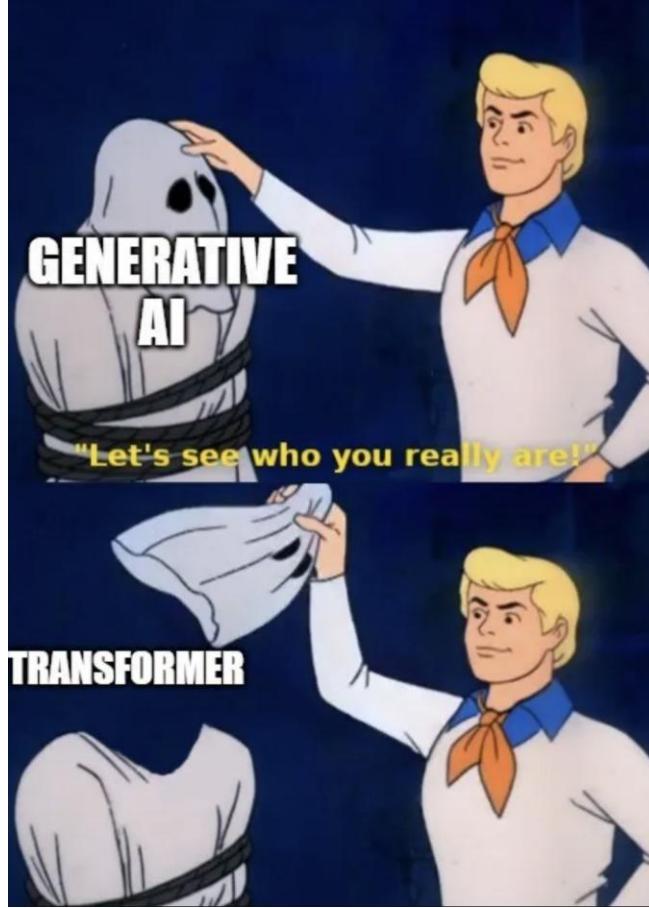
RNNs are not parallelizable across sequence



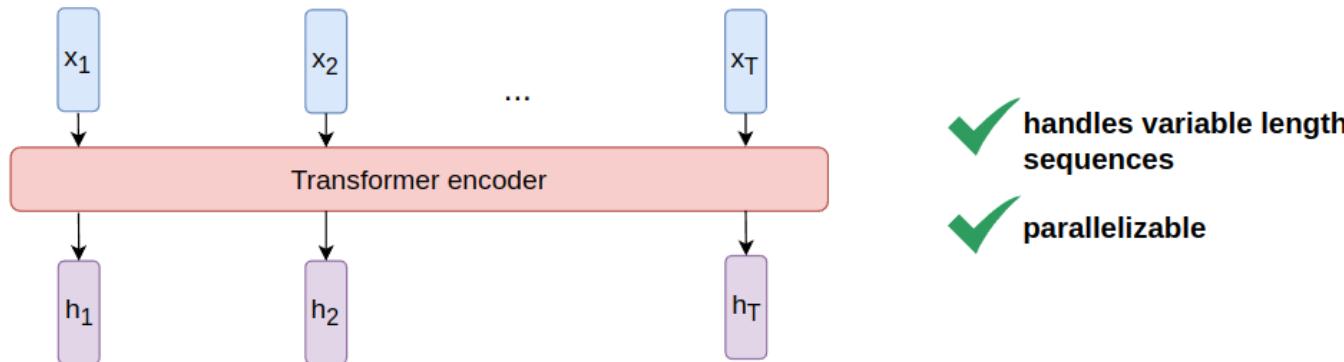
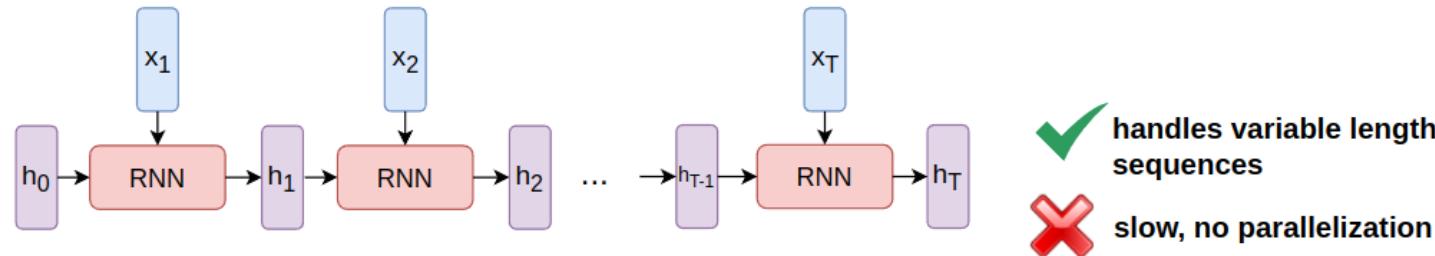
linear interaction distance:

- maximum path length between distant elements is  $O(n)$
- *long term dependencies* still tricky to learn, even with gating mechanisms (LSTM, GRU etc.)

# Generative AI

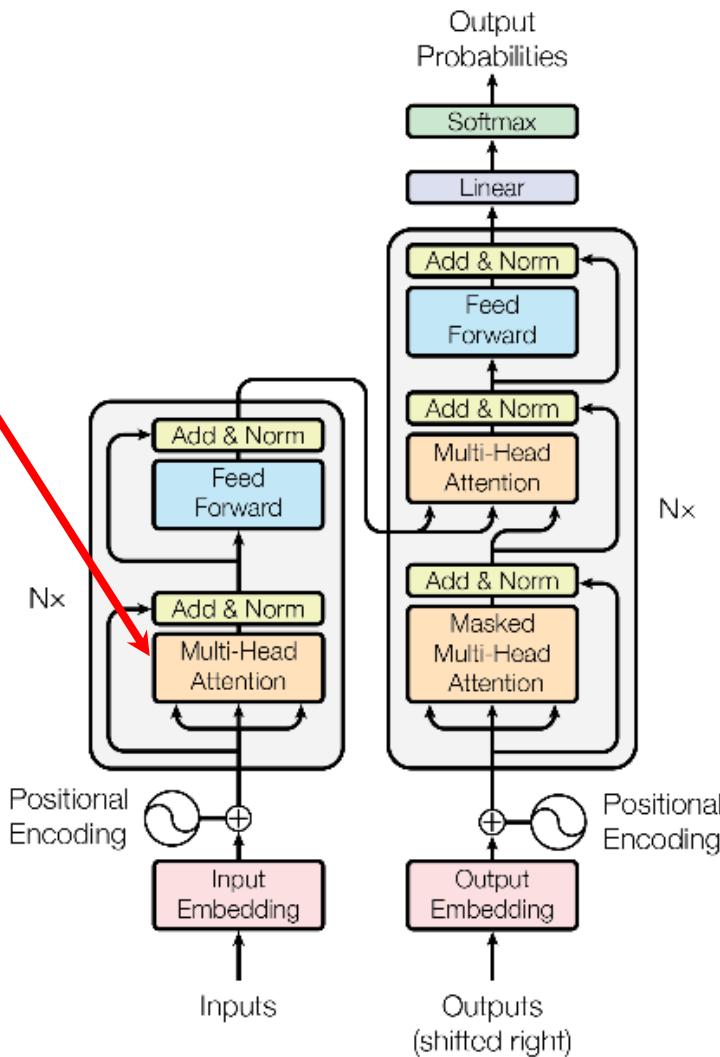


# Alternative approaches for Sequential Data



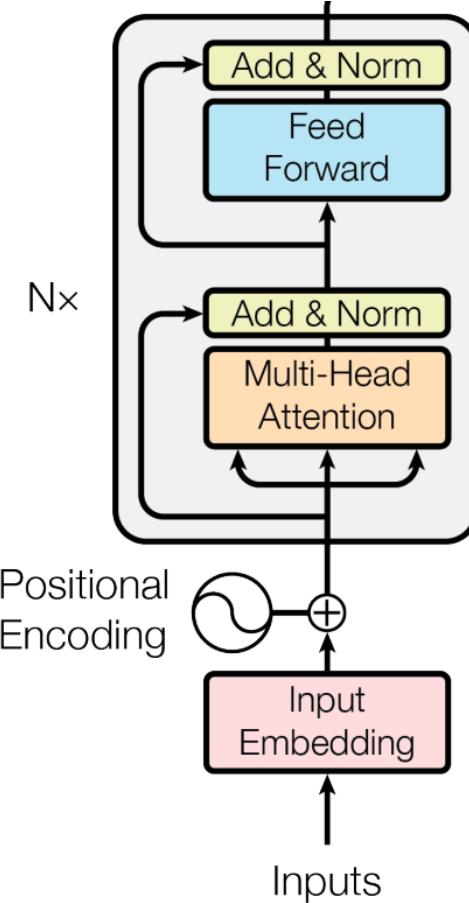
# Transformer (2017)

- no RNNs: introduces multi-head self-attention
- SOTA on Machine Translation (EN-DE, EN-FR)

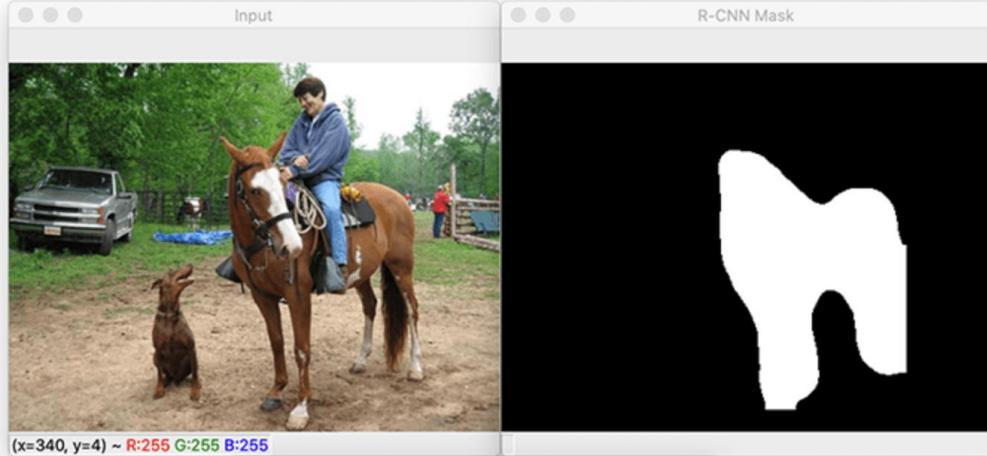


# Transformer today

- **backbone of chatbot revolution:**
  - used in GPT4, Gemini, Claude etc.
- **multimodal capabilities:**
  - SoTA results in other fields such as computer vision, audio and more
- **scalability:**
  - has been shown to improve with scale:  
**more data & bigger model => better performance (scaling laws)**

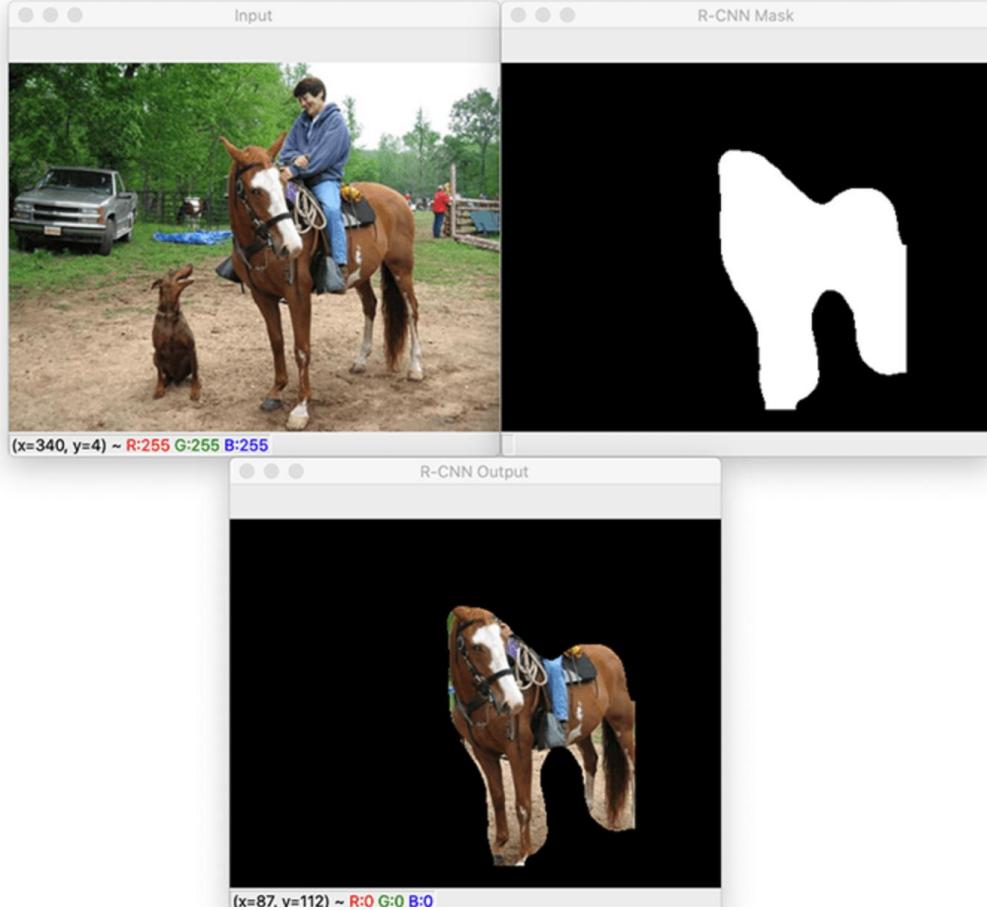


# Self-Attention intuition



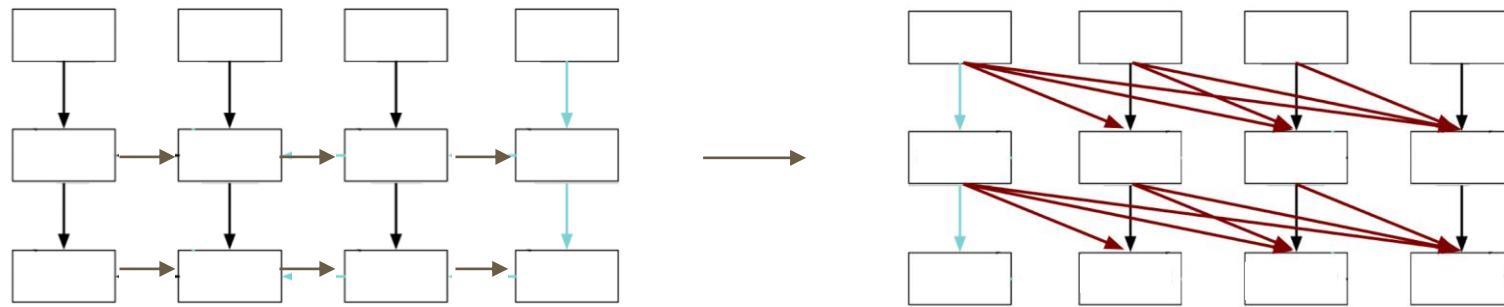
1. Identify which elements to attend to (search problem)
2. Extract features with high attention

# Self-Attention intuition



1. Identify which elements to attend to (search problem)
2. Extract features with high attention

# Self-Attention

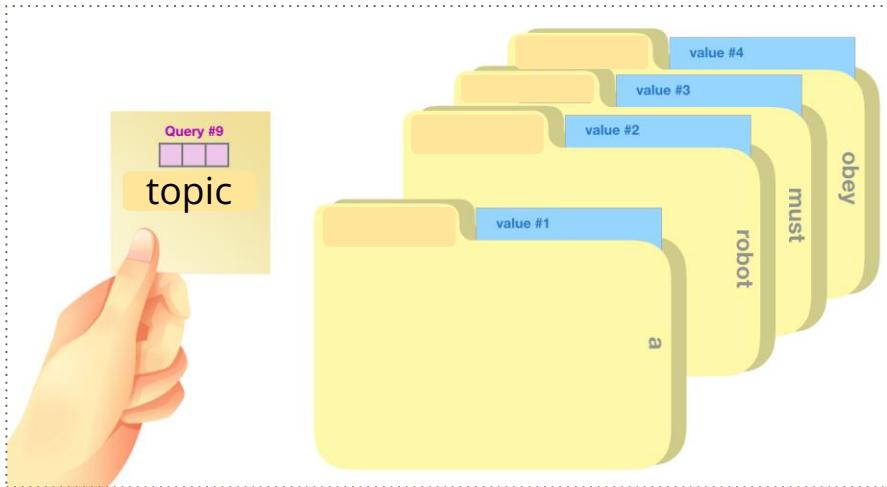


causal self-attention

- no more recurrence
- each state ***depends on (almost) all the others:***
  - *causal self-attention:* state **t** depends on *all previous states*
  - *self-attention:* state **t** depends on *all the other states*

# Attention as soft dictionary look-up

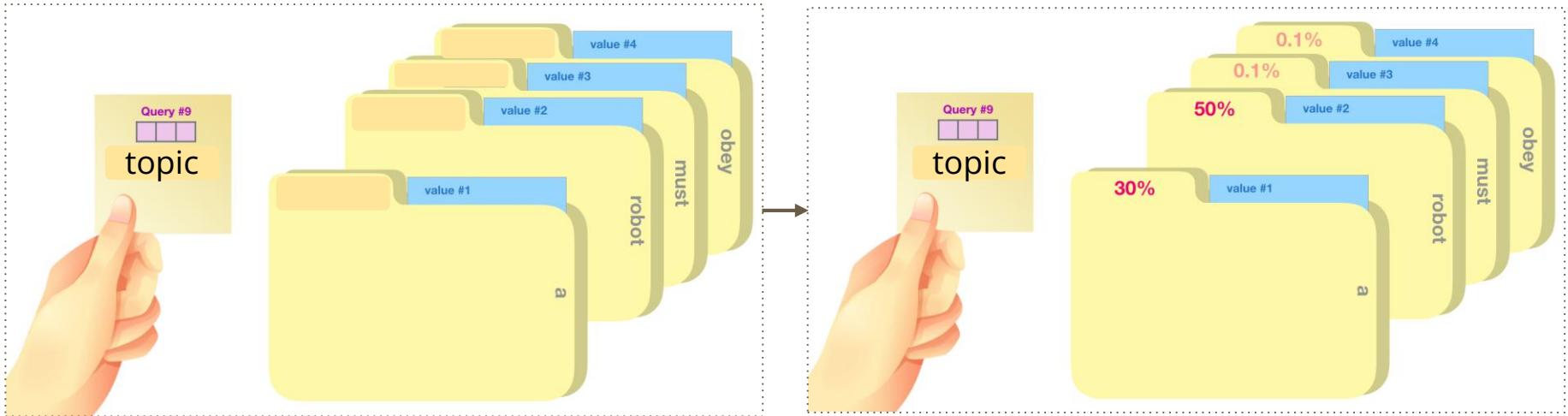
- **Scenario:** I want to research a topic => retrieve most relevant content
  - **query** - vector for research topic, **value<sub>i</sub>** - vector for folder **i**



figures from <http://jalammar.github.io/illustrated-gpt2/>

# Attention as soft dictionary look-up

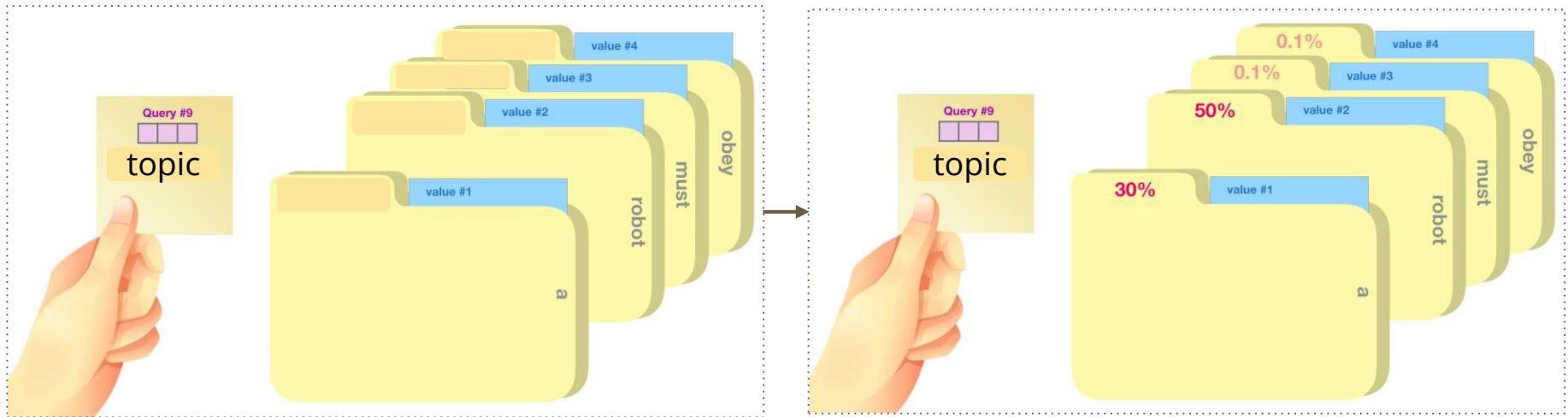
- **Scenario:** I want to research a topic => retrieve most relevant content
  - **query** - vector for research topic, **value<sub>i</sub>** - vector for folder **i**
  - **output** =  $\sum_i a_i \text{value}_i$ , where  $a_i = \text{similarity}(\text{query}, \text{value}_i)$



figures from <http://jalammar.github.io/illustrated-gpt2/>

# Attention as soft dictionary look-up

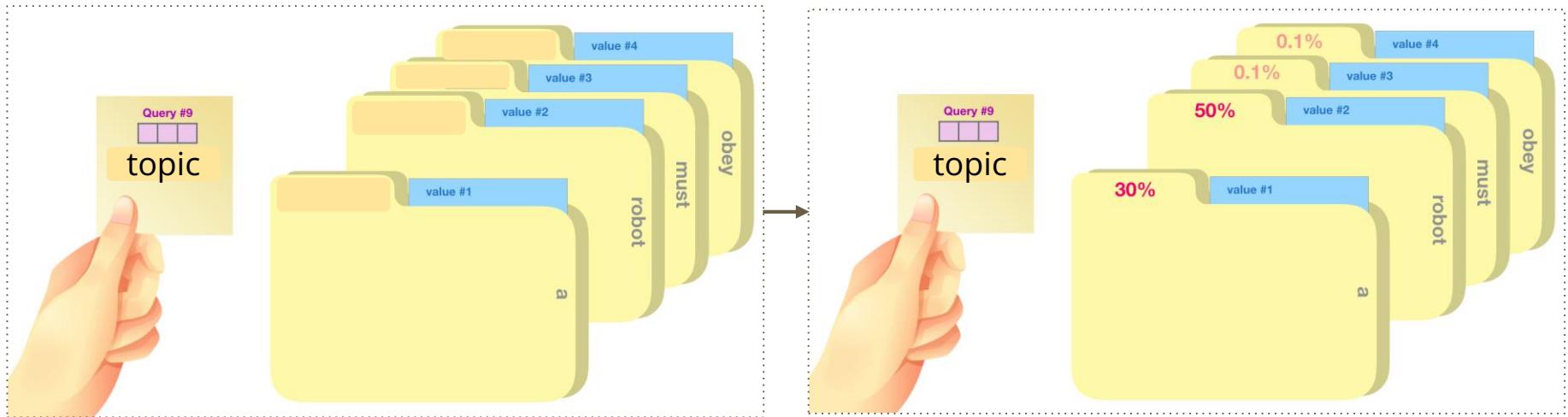
- step 1: match **query** (topic) with **values** => attention scores  $a_1, a_2, \dots, a_N$



figures from <http://jalammar.github.io/illustrated-gpt2/>

# Attention as soft dictionary look-up

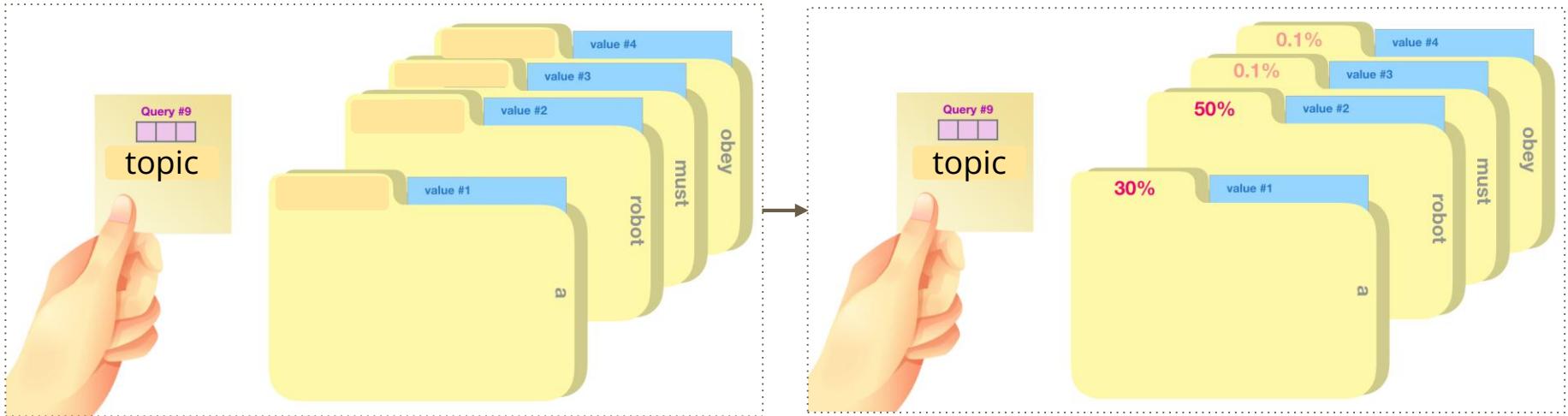
- step 1: match **query** (topic) with **values** => attention scores  $a_1, a_2, \dots, a_N$ 
  - take dot product between **query** and each **value<sub>i</sub>**, then normalize them



figures from <http://jalammar.github.io/illustrated-gpt2/>

# Attention as soft dictionary look-up

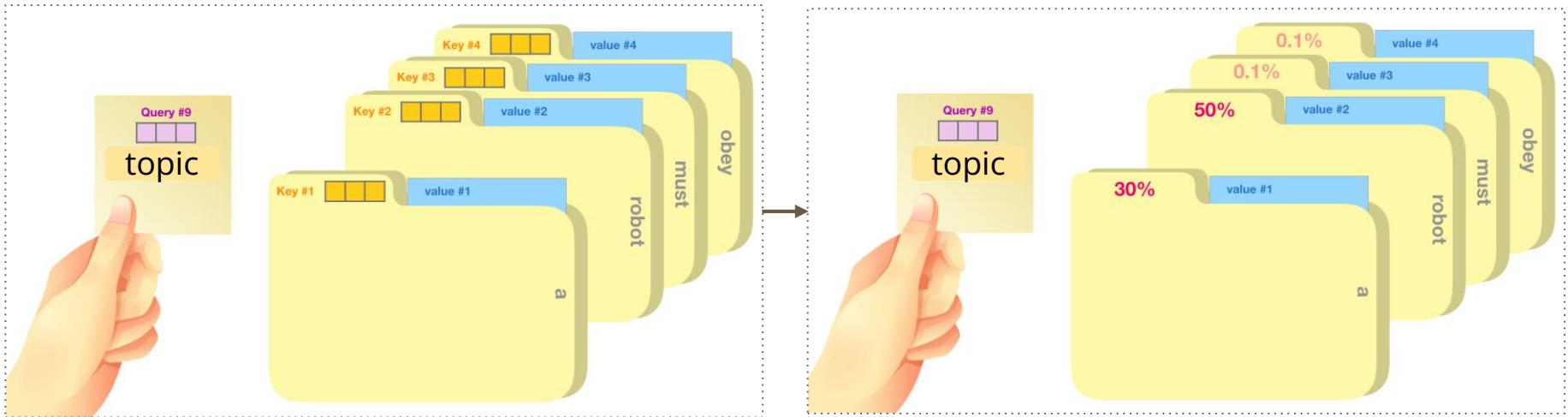
- **step 1:** match **query** (topic) with **values** => attention scores  $a_1, a_2, \dots, a_N$ 
  - take dot product between **query** and each **value<sub>i</sub>**, then normalize them
- **step 2:** weighted sum of **values** and **scores**: output =  $\sum_i a_i \text{ value}_i$



figures from <http://jalammar.github.io/illustrated-gpt2/>

# Attention with Query, Keys and Values

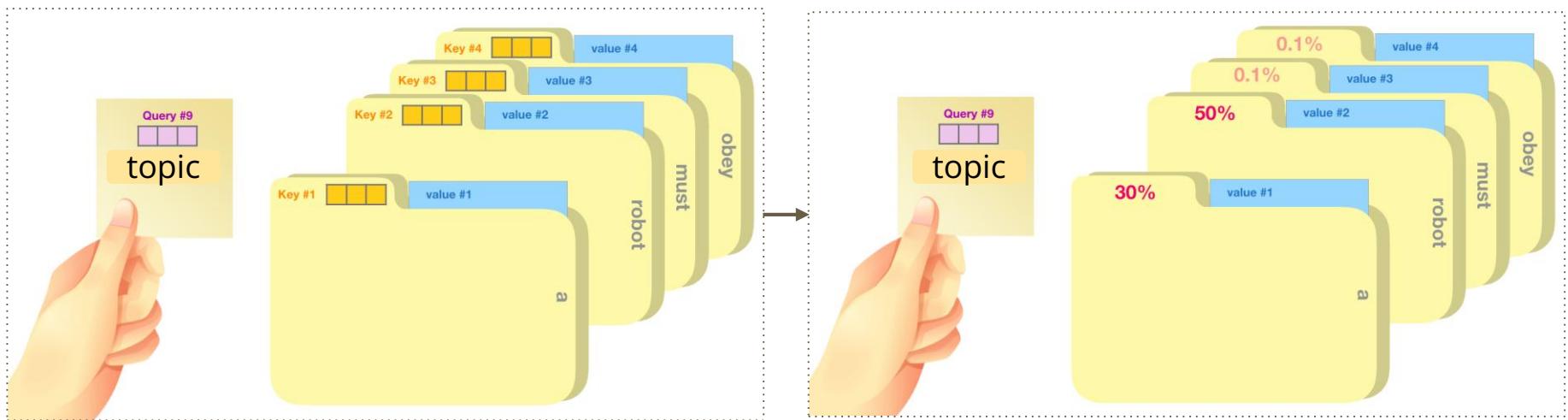
- **query** - research topic
- **key<sub>i</sub>** - name of folder **i**
- **value<sub>i</sub>** - content of folder **i**



figures from <http://jalammar.github.io/illustrated-gpt2/>

# Key, Query and Value analogy

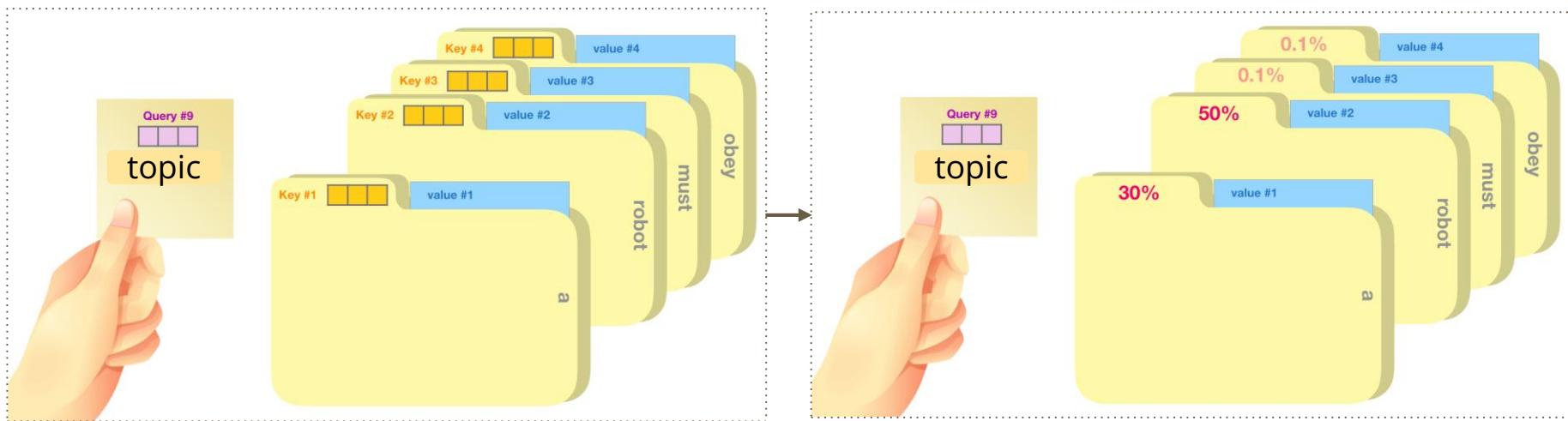
- Step 1: match **query** (topic) with **keys** (folder names) => attention scores



figures from <http://jalammar.github.io/illustrated-gpt2/>

# Key, Query and Value analogy

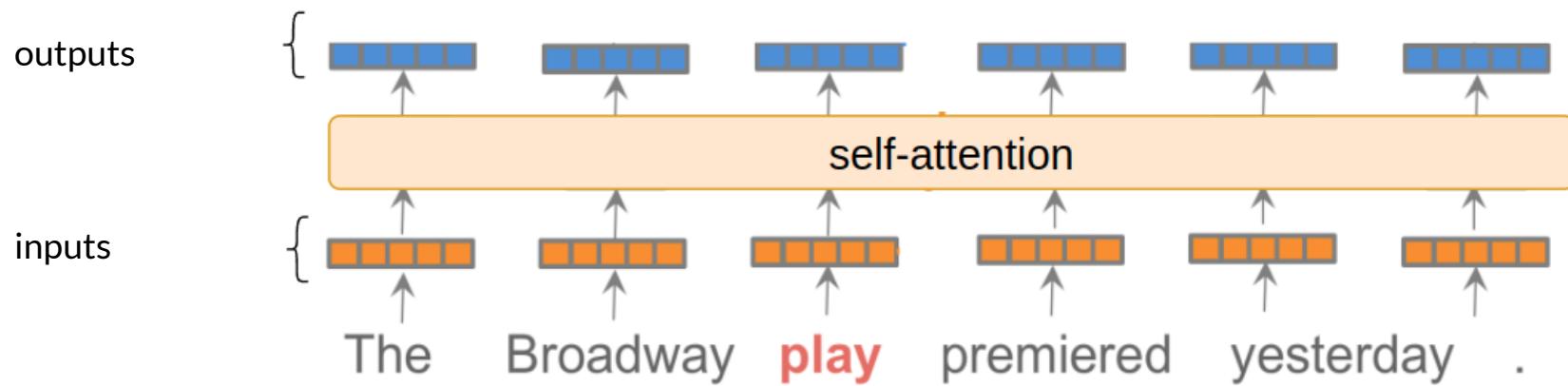
- Step 1: match **query** (topic) with **keys** (folder names) => attention scores
- Step 2: retrieve content from each folder (**value**) according to scores  
**output** =  $\sum_i a_i \text{ value}_i$ , where  $a_i$  = similarity(**query**, **key<sub>i</sub>**)



figures from <http://jalammar.github.io/illustrated-gpt2/>

# Self-attention blackbox

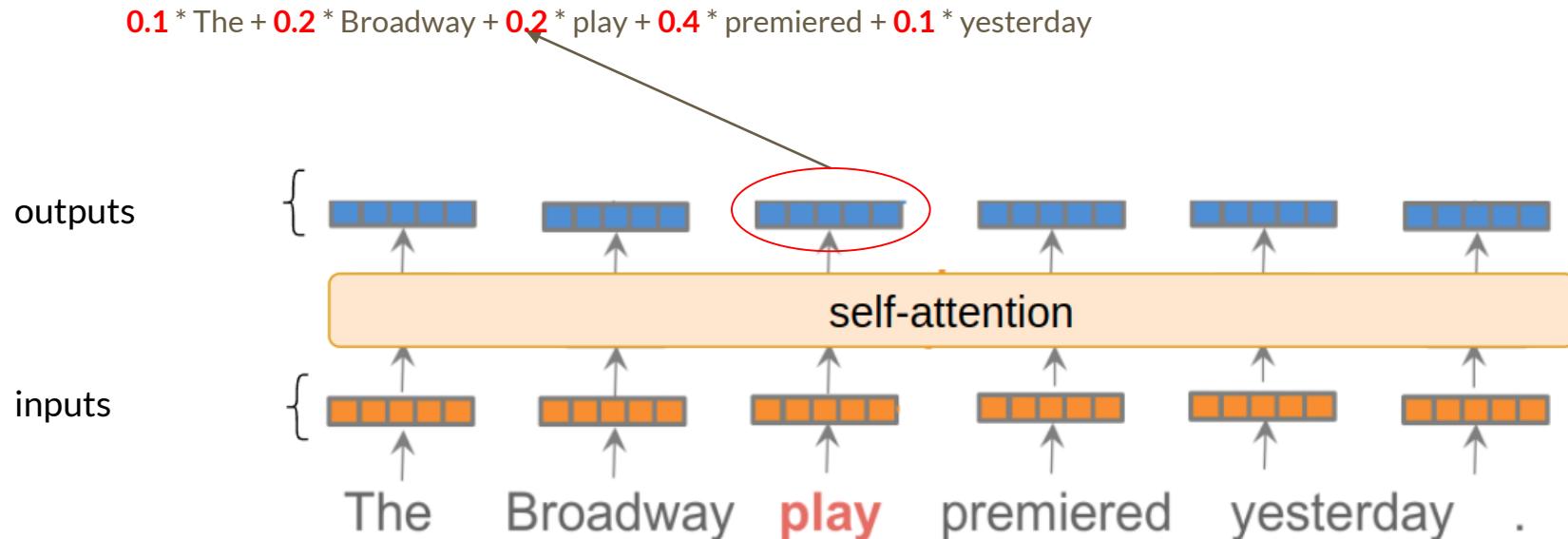
N inputs => N outputs



# Self-attention blackbox

Two steps for output i:

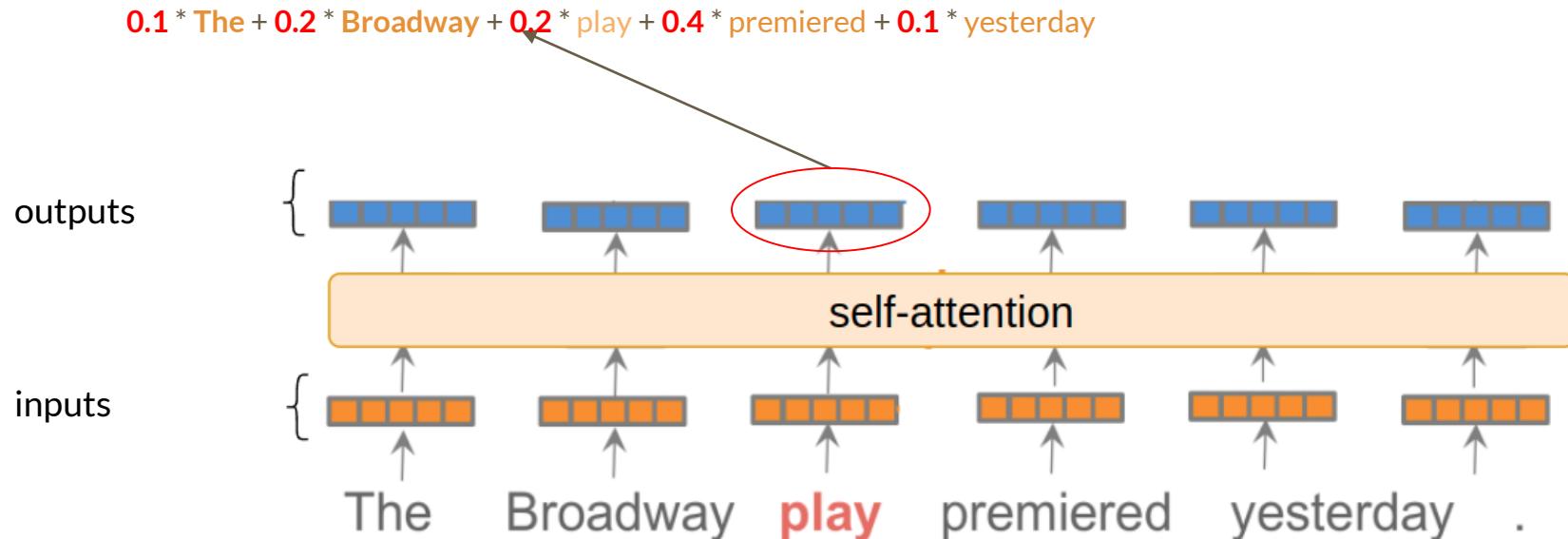
1. identify the most 'relevant' features for input i, by 'looking' at all the other input words => **attention scores**



# Self-attention blackbox

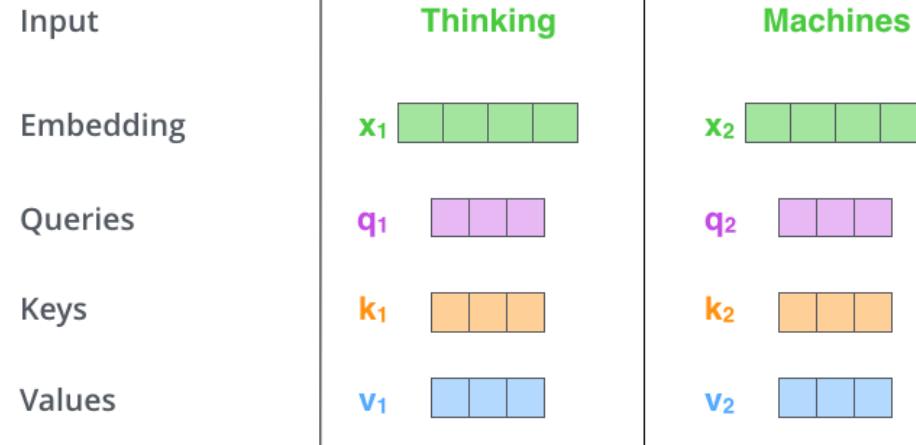
Two steps for output i:

1. identify the most 'relevant' features for input i, by 'looking' at all the other input words => **attention scores**
2. combine the most relevant features from **inputs** with **attention scores**



# Self-attention Q,K,V

- compute **query**, **keys** and **values** for each embedding
- why 3 subspaces?



# Self-attention Q,K,V

- compute **query**, **keys** and **values** for each embedding
- why 3 subspaces?
  - 1 subspace (V): large dot product between **word<sub>i</sub>**->**word<sub>i</sub>** => words 'attend' to themselves

Input

Embedding

Queries

Keys

Values

Thinking

**x<sub>1</sub>** 

**q<sub>1</sub>** 

**k<sub>1</sub>** 

**v<sub>1</sub>** 

Machines

**x<sub>2</sub>** 

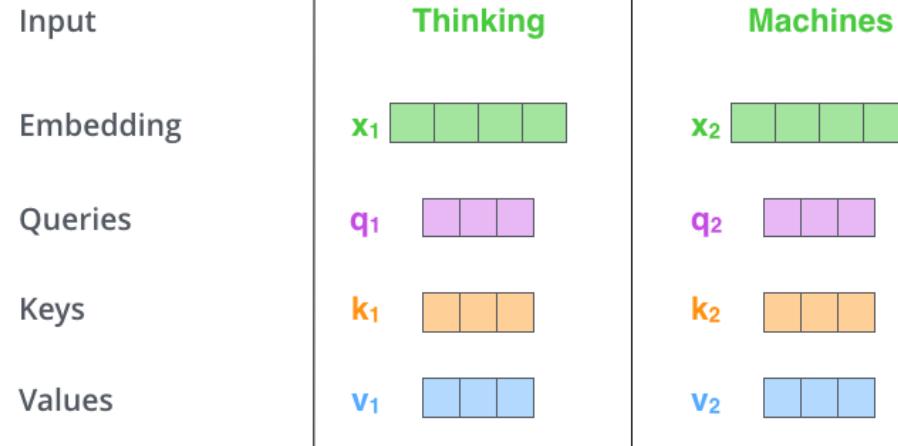
**q<sub>2</sub>** 

**k<sub>2</sub>** 

**v<sub>2</sub>** 

# Self-attention Q,K,V

- compute **query**, **keys** and **values** for each embedding
- why 3 subspaces?
  - 1 subspace (V): large dot product between **word<sub>i</sub>**<->**word<sub>i</sub>** => words 'attend' to themselves
  - 2 subspaces (Q, V):
    - some features in V may be **good for matching** with Q, others for **storing content**
    - sharing them in V may be suboptimal



# Compute queries, keys, values

- 3 linear layers:  $W^Q$ ,  $W^K$  and  $W^V$ 
  - usually without bias (position already encoded w/ positional embeddings)

$$X \times W^Q = Q$$

A diagram illustrating the computation of queries. It shows a green 4x4 matrix labeled 'X' multiplied by a purple 4x4 matrix labeled 'W<sup>Q</sup>' resulting in a purple 4x4 matrix labeled 'Q'. The matrices are represented as grids of colored squares.

$$X \times W^K = K$$

A diagram illustrating the computation of keys. It shows a green 4x4 matrix labeled 'X' multiplied by an orange 4x4 matrix labeled 'W<sup>K</sup>' resulting in an orange 4x4 matrix labeled 'K'. The matrices are represented as grids of colored squares.

$$X \times W^V = V$$

A diagram illustrating the computation of values. It shows a green 4x4 matrix labeled 'X' multiplied by a blue 4x4 matrix labeled 'W<sup>V</sup>' resulting in a blue 4x4 matrix labeled 'V'. The matrices are represented as grids of colored squares.

# Compute queries, keys, values

- 3 linear layers:  $W^Q$ ,  $W^K$  and  $W^V$ 
  - usually without bias (position already encoded w/ positional embeddings)
- map embedding space of size  $d_{\text{model}}$  to lower-dimensional space of size  $d_q/d_k/d_v$ 
  - e.g.:  $d_{\text{model}} = 512 \Rightarrow d_q = 64$

$$X \times W^Q = Q$$

A diagram illustrating the computation of queries (Q). It shows a green 4x4 matrix labeled 'X' multiplied by a purple 4x4 matrix labeled 'W<sup>Q</sup>' resulting in a purple 4x4 matrix labeled 'Q'. The matrices are represented as grids of colored squares.

$$X \times W^K = K$$

A diagram illustrating the computation of keys (K). It shows a green 4x4 matrix labeled 'X' multiplied by an orange 4x4 matrix labeled 'W<sup>K</sup>' resulting in an orange 4x4 matrix labeled 'K'. The matrices are represented as grids of colored squares.

$$X \times W^V = V$$

A diagram illustrating the computation of values (V). It shows a green 4x4 matrix labeled 'X' multiplied by a blue 4x4 matrix labeled 'W<sup>V</sup>' resulting in a blue 4x4 matrix labeled 'V'. The matrices are represented as grids of colored squares.

# Compute queries, keys, values

- 3 linear layers:  $W^Q$ ,  $W^K$  and  $W^V$ 
  - usually without bias (position already encoded w/ positional embeddings)
- map embedding space of size  $d_{\text{model}}$  to lower-dimensional space of size  $d_q/d_k/d_v$ 
  - e.g.:  $d_{\text{model}}=512 \Rightarrow d_q=64$
- why downscale?
  - efficiency
  - enables learning multiple subspaces of Q,K,V:
    - instead of  $1 \times (512 \rightarrow 512)$  mapping, learn  $8 \times (512 \rightarrow 64)$  mappings with the same cost

$$X \times W^Q = Q$$

$$X \times W^K = K$$

$$X \times W^V = V$$

# Self-attention steps ( $x_1$ )

- compute **attention scores** by taking dot products between **query  $q_1$**  and all the **keys**
- divide scores by  $\sqrt{d_k}$ , why?

Input

Embedding

Queries

Keys

Values

Score

Divide by 8 (  $\sqrt{d_k}$  )

Softmax

Thinking

$x_1$  

$q_1$  

$k_1$  

$v_1$  

$$q_1 \cdot k_1 = 112$$

14

0.88

Machines

$x_2$  

$q_2$  

$k_2$  

$v_2$  

$$q_1 \cdot k_2 = 96$$

12

0.12

# Self-attention steps ( $x_1$ )

- compute **attention scores** by taking dot products between **query  $q_1$**  and all the **keys**
- divide scores by  $\sqrt{d_k}$ , why?
  - dot products can get quite large  
=> softmax gets very 'peaky'

Input

Embedding

Queries

Keys

Values

Score

Divide by 8 (  $\sqrt{d_k}$  )

Softmax

Thinking

$x_1$  

$q_1$  

$k_1$  

$v_1$  

$q_1 \cdot k_1 = 112$

14

0.88

Machines

$x_2$  

$q_2$  

$k_2$  

$v_2$  

$q_1 \cdot k_2 = 96$

12

0.12

# Self-attention steps ( $x_1$ )

- compute **attention scores** by taking dot products between **query  $q_1$**  and all the **keys**
- divide scores by  $\sqrt{d_k}$ , why?
  - dot products can get quite large  
=> softmax gets very 'peaky'
  - variance is dependent on  $d_k$ 
    - if each feature in  $q_i, k_i \sim N(0, 1)$   
=>  $\text{var}(q^T k) = d_k$

Input

Embedding

Queries

Keys

Values

Score

Divide by 8 ( $\sqrt{d_k}$ )

Softmax

Thinking

$x_1$  

$q_1$  

$k_1$  

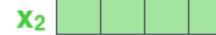
$v_1$  

$$q_1 \cdot k_1 = 112$$

$$14$$

$$0.88$$

Machines

$x_2$  

$q_2$  

$k_2$  

$v_2$  

$$q_1 \cdot k_2 = 96$$

$$12$$

$$0.12$$

# Self-attention steps ( $x_1$ )

- compute **attention scores** by taking dot products between **query  $q_1$**  and all the **keys**
- divide scores by  $\sqrt{d_k}$ , why?
  - dot products can get quite large  
=> softmax gets very 'peaky'
  - variance is dependent on  $d_k$ 
    - if each feature in  $q_i, k_j \sim N(0,1)$   
=>  $\text{var}(q_i^T k_j) = d_k$
  - dividing by  $\sqrt{d_k}$  keeps variance constant:
    - $\text{var}((q_i^T k_j) / \sqrt{d_k}) = 1$

Input

Embedding

Queries

Keys

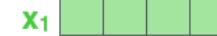
Values

Score

Divide by 8 ( $\sqrt{d_k}$ )

Softmax

Thinking



$$q_1 \cdot k_1 = 112$$

14

0.88

Machines



$$q_1 \cdot k_2 = 96$$

12

0.12

# Self-attention steps ( $x_1$ )

- compute **attention scores** by taking dot products between **query  $q_1$**  and all the **keys**
- scale and normalize **attention scores**

Input

Embedding

Queries

Keys

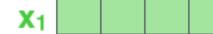
Values

Score

Divide by 8 (  $\sqrt{d_k}$  )

Softmax

Thinking

$x_1$  

$q_1$  

$k_1$  

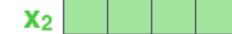
$v_1$  

$$q_1 \cdot k_1 = 112$$

14

0.88

Machines

$x_2$  

$q_2$  

$k_2$  

$v_2$  

$$q_1 \cdot k_2 = 96$$

12

0.12

# Self-attention steps ( $x_1$ )

- compute **attention scores** by taking dot products between **query  $q_1$**  and all the **keys**
- scale and normalize **attention scores**
- **output  $z_1$**  is the weighted sum of **values** and their **attention scores**

Input

Embedding

Queries

Keys

Values

Score

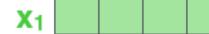
Divide by 8 ( $\sqrt{d_k}$ )

Softmax

Softmax  
X  
Value

Sum

Thinking

$x_1$  

$q_1$  

$k_1$  

$v_1$  

$$q_1 \cdot k_1 = 112$$

14

0.88

$v_1$  

$z_1$  

Machines

$x_2$  

$q_2$  

$k_2$  

$v_2$  

$$q_1 \cdot k_2 = 96$$

12

0.12

$v_2$  

$z_2$  

# Self-attention (all inputs)

repeat for all inputs  $x_i$ :

- compute **attention scores** between **query  $q_i$**  and all the **keys**
- scale and normalize **attention scores**
- **output  $z_i$**  is the weighted sum of **values** and their **attention scores**

Input

Embedding

Queries

Keys

Values

Score

Divide by 8 ( $\sqrt{d_k}$ )

Softmax

Softmax  
X  
Value

Sum

Thinking

$x_1$  

$q_1$  

$k_1$  

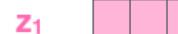
$v_1$  

$$q_1 \cdot k_1 = 112$$

14

0.88

$v_1$  

$z_1$  

Machines

$x_2$  

$q_2$  

$k_2$  

$v_2$  

$$q_1 \cdot k_2 = 96$$

12

0.12

$v_2$  

$z_2$  

## Step 1: Compute queries, keys, values for all inputs

layer 0:

input X = initial token embeddings

$$\begin{matrix} X & \times & W^Q & = & Q \end{matrix}$$

The diagram shows three matrices:  $X$  (green, 2x5),  $W^Q$  (purple, 5x5), and  $Q$  (purple, 2x3). The multiplication  $X \times W^Q$  results in matrix  $Q$ .

layer l:

input X = previous outputs from  
former block

$$\begin{array}{c} \text{X} \\ \times \\ \boxed{\begin{array}{|c|c|c|c|} \hline & & & \\ \hline \end{array}} \\ = \\ \boxed{\begin{array}{|c|c|} \hline & \\ \hline & \\ \hline \end{array}} \end{array}$$

$$\begin{matrix} X & \times & W^V & = & V \end{matrix}$$

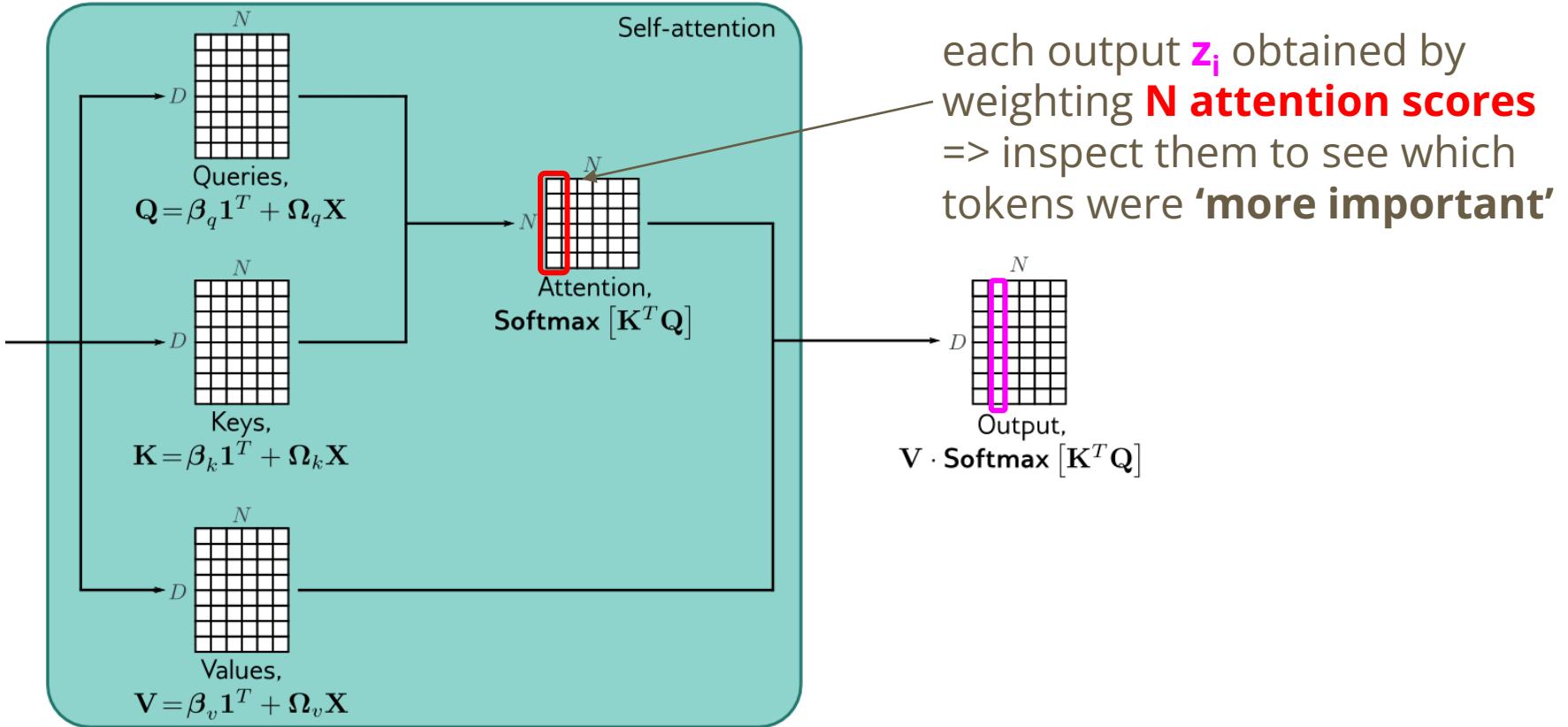
figure from <http://jalammar.github.io/illustrated-transformer/>

## Step 2: Compute self-attention in parallel

- **output vectors  $z_i$**  are computed for **all tokens at once** (unlike RNNs)
- complexity:  $O(N^2d)$

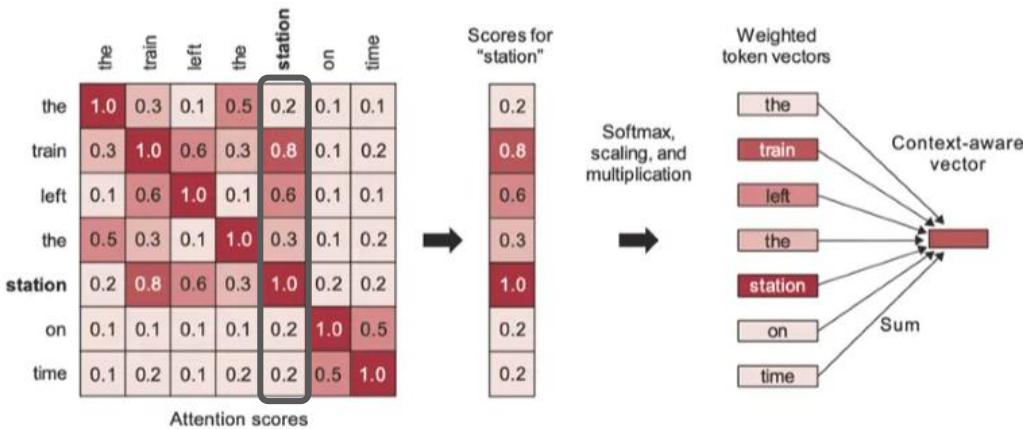
$$\text{softmax} \left( \frac{\begin{array}{c} \text{Q} \quad \text{K}^\top \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \quad \times \quad \begin{array}{|c|c|} \hline & \\ \hline & \\ \hline \end{array} \end{array}}{\sqrt{d_k}} \right) \text{V}$$
$$= \begin{array}{c} \text{Z} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \end{array}$$

# Attention scores are interpretable



# Attention scores are interpretable

when computing output for the word '**station**', attention places importance on itself, but also on '**train**' and '**left**'



The attention score is computed through cosine similarity. I.e., the dot

# Masked/Causal self-attention

- inputs should not 'peak' into the **future**:
  - each input can only 'attend' to *itself and previous* inputs

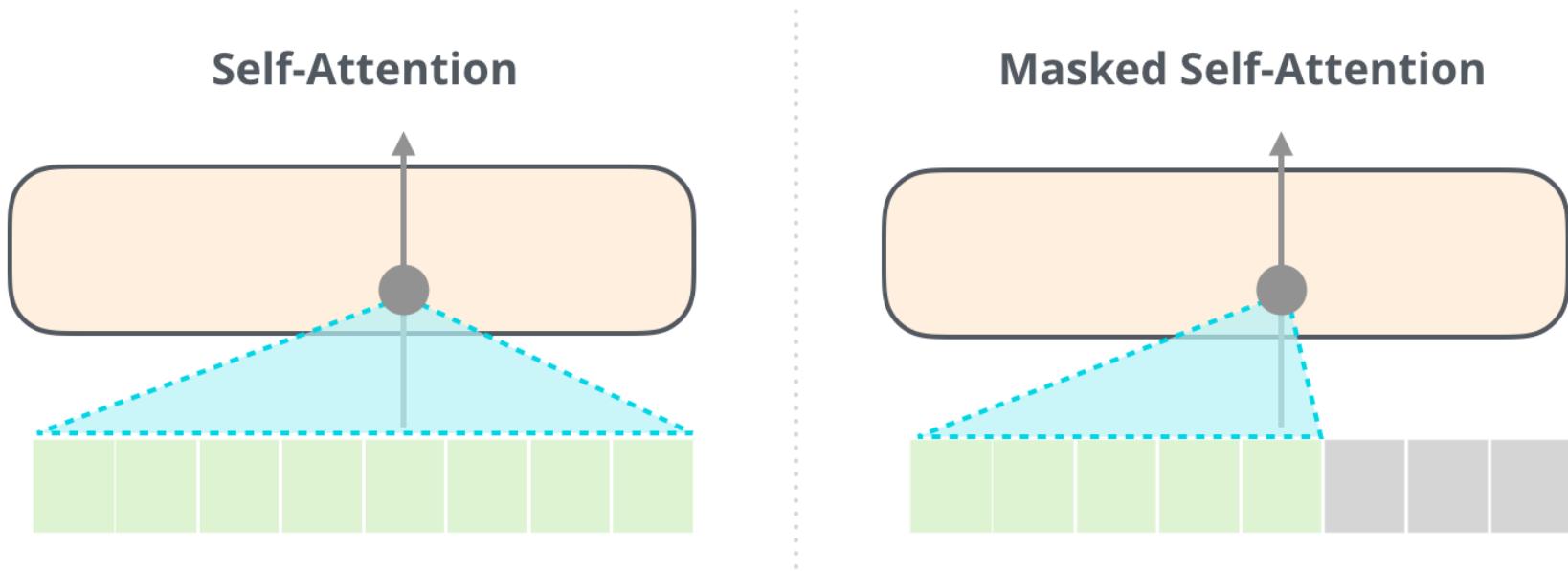


figure from figure from <http://jalammar.github.io/illustrated-gpt2/>

# Masked/Causal self-attention

- mask unnormalized attention scores above diagonal with -inf
- given token t, this results in attention scores = 0 for tokens > t

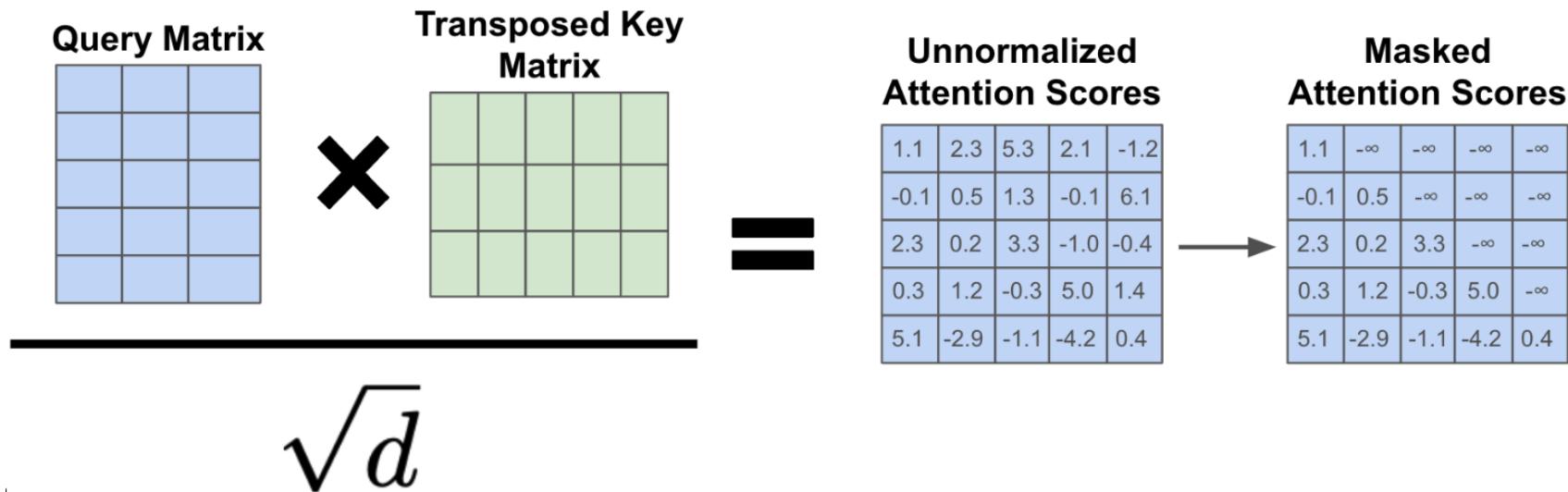
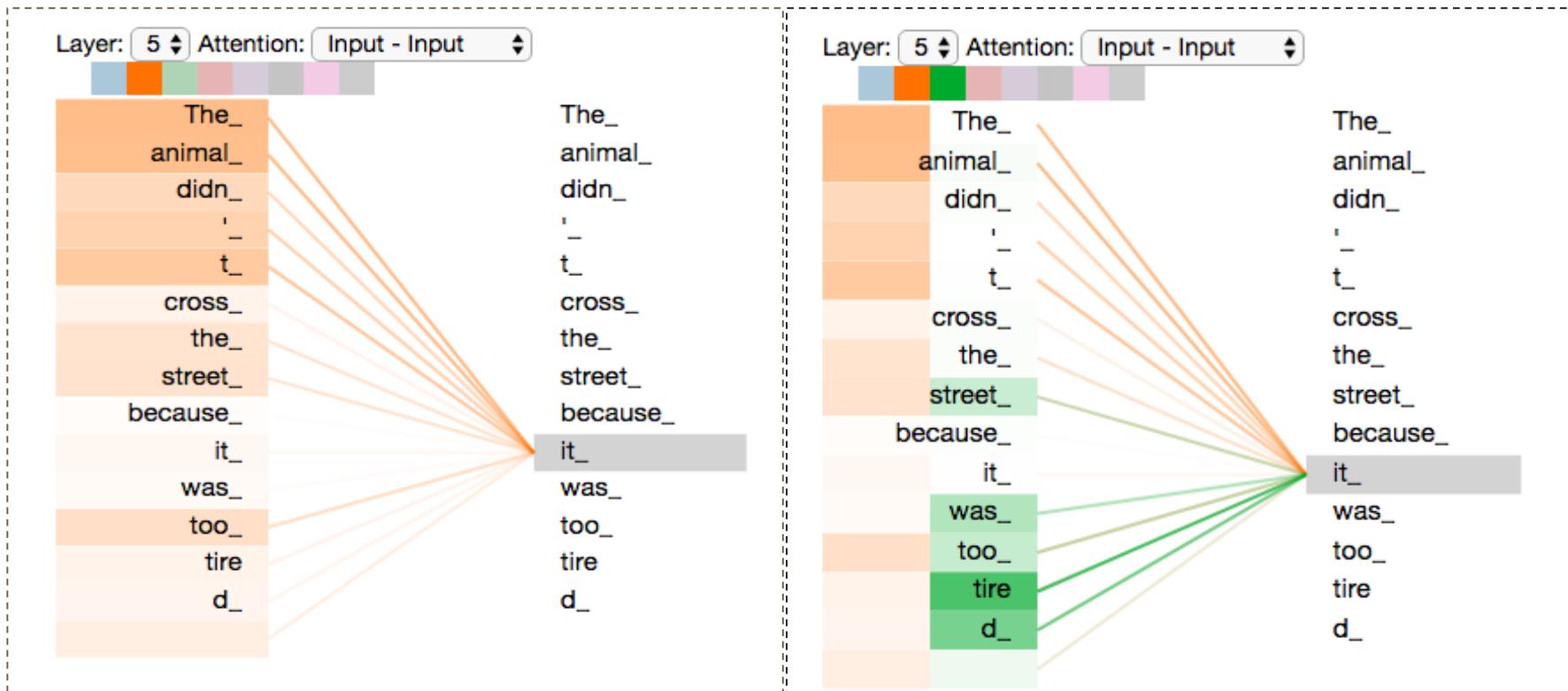


figure from <https://cameronrwolfe.substack.com/p/decoder-only-transformers-the-workhorse>

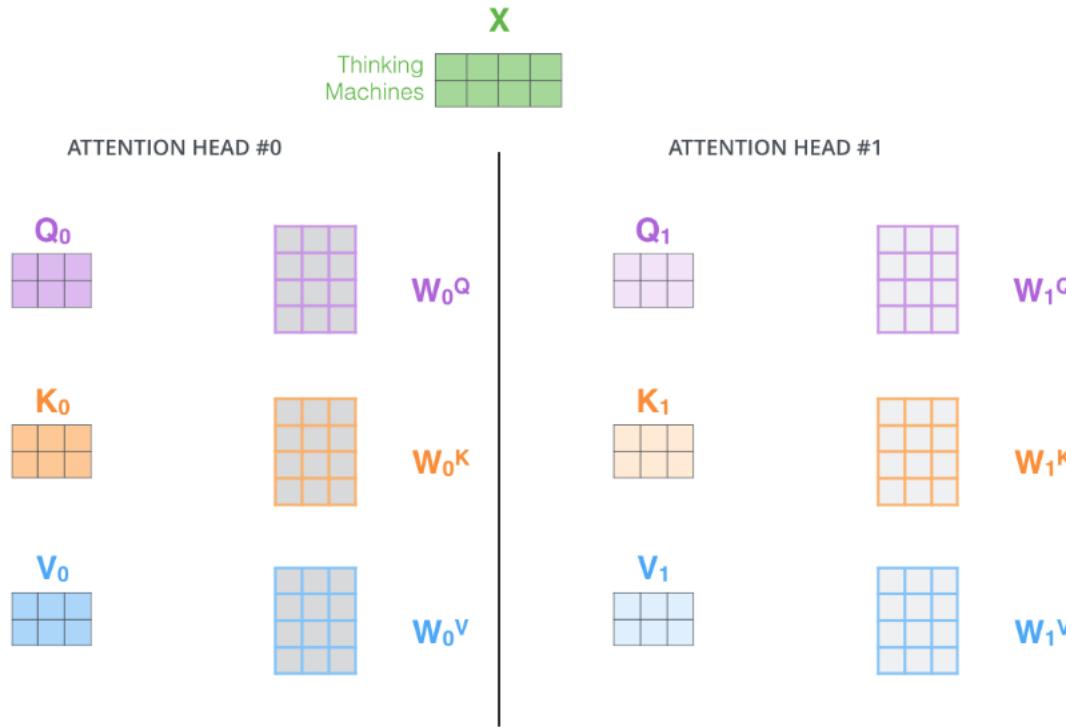
# Why only learn a single set of connections?

multiple self-attention heads can learn different relationships



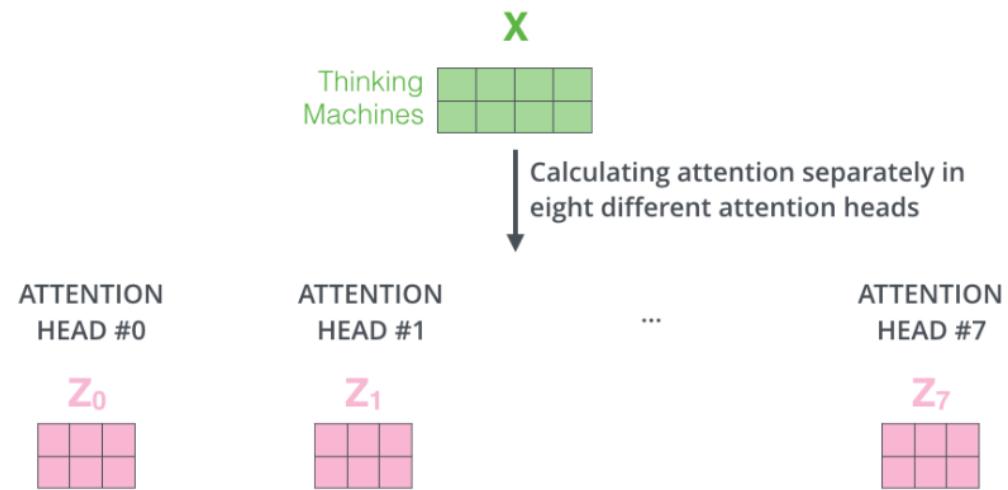
# Multi-head self attention

- use K heads instead of 1
  - $K=d_{\text{model}}/d_q$
- each head has its own  $W^Q$ ,  $W^K$  and  $W^V$  layer
  - each head learns a different subspace



# Multi-head self attention

- the K heads compute K sets of outputs for each token  
=>  $N \times (K \times d_v)$  tensor
- these outputs were computed independently:
  - how to mix them?



# Multi-head self attention: mix heads

1. concatenate outputs on the feature dimension  
=>  $N \times d_{\text{model}}$  tensor

1) Concatenate all the attention heads



# Multi-head self attention: mix heads

1. concatenate outputs on the feature dimension  
=>  $N \times d_{\text{model}}$  tensor

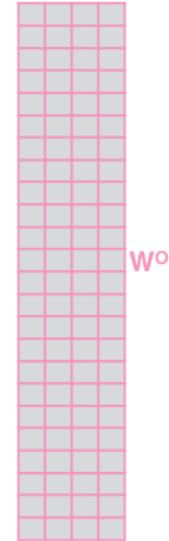
2. apply projection matrix  $W^o$  of size  $d_{\text{model}} \times d_{\text{model}}$ 
  - o this transform learns to **mix information** from *different attention heads*

1) Concatenate all the attention heads



2) Multiply with a weight matrix  $W^o$  that was trained jointly with the model

$X$

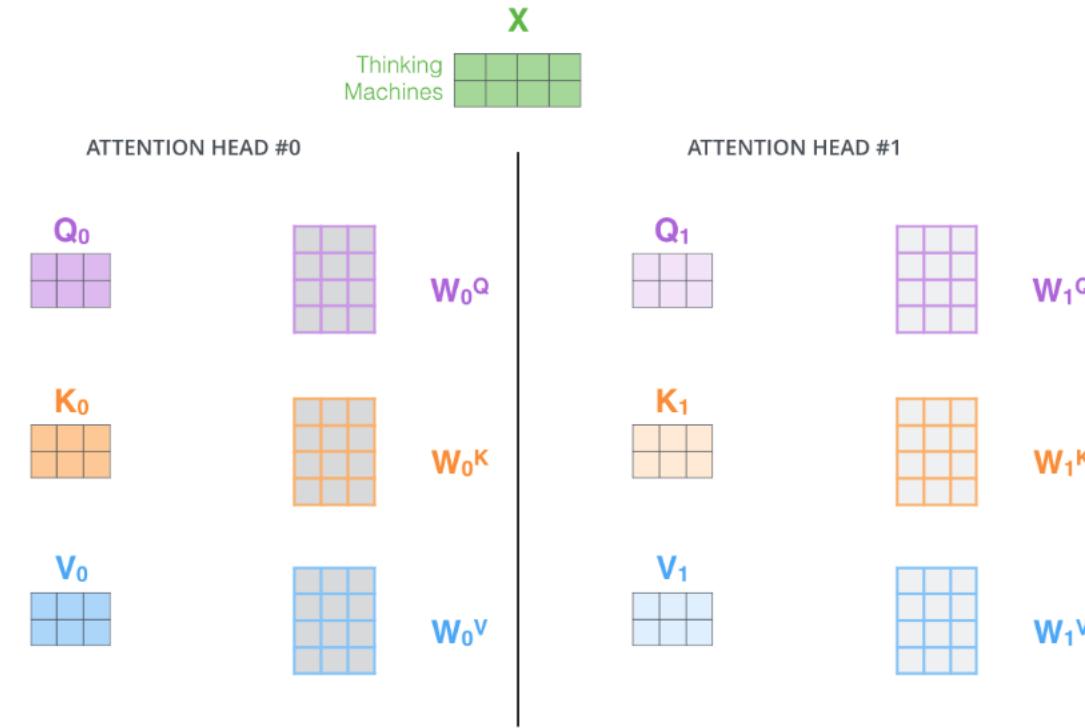


3) The result would be the  $Z$  matrix that captures information from all the attention heads. We can send this forward to the FFNN

$$= \begin{matrix} Z \\ \hline \end{matrix}$$

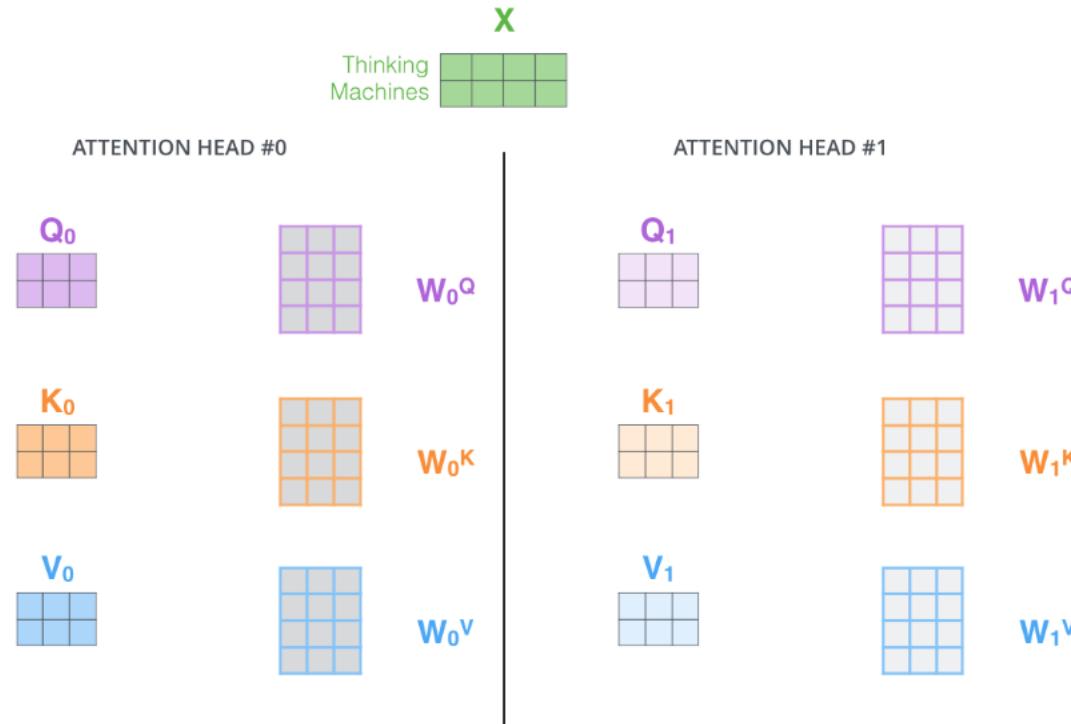
# Multi-head self attention: how many heads?

- increasing K usually helps:
  - 8 heads with 64 features each performed best in orig. paper



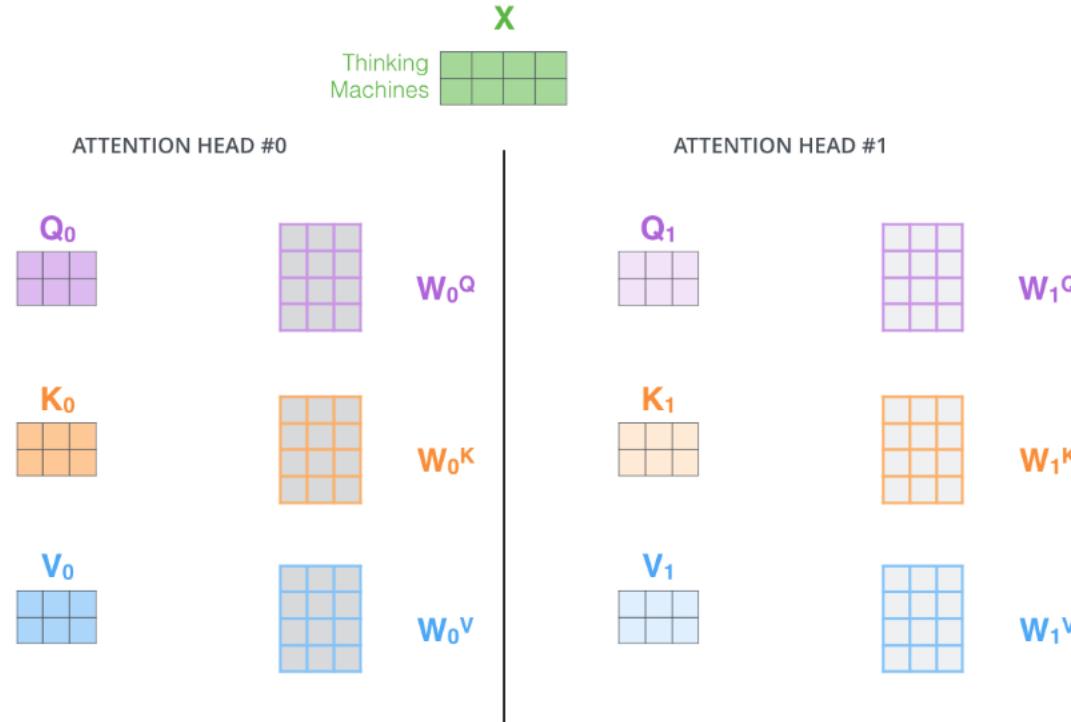
# Multi-head self attention: how many heads?

- increasing K usually helps:
  - 8 heads with 64 features each performed best in orig. paper
- diminishing returns:
  - in the original paper, **32** heads with **16** features performed **worse** than **16** heads with **32** features
  - larger K means  $d_k$  gets smaller => not enough features to encode useful relationships



# Multi-head self attention: how many heads?

- increasing K usually helps:
  - 8 heads with 64 features each performed best in orig. paper
- diminishing returns:
  - in the original paper, **32** heads with **16** features performed **worse** than **16** heads with **32** features
  - large K means  $d_k$  gets smaller  
=> not enough features to encode useful relationships
- modern models use  $d_k=64/128$  and increase K to get desired  $d_{\text{model}}$



# Positional Awareness

- Self-attention has no temporal order information:
  - ‘sat’ attends to ‘mat’ and ‘cat’ in the same way, *regardless of their order* in the sentence
  - outputs for the 2 ‘the’ are indistinguishable, instead of having one of them more linked to ‘cat’ and the other more linked to ‘mat’

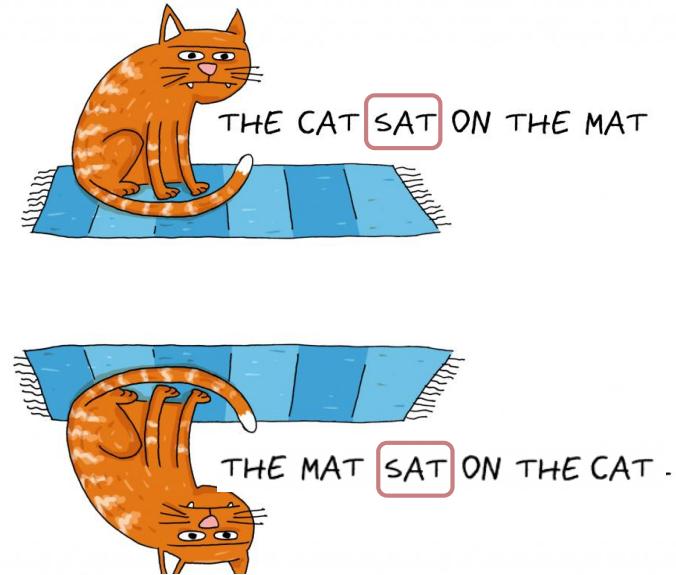
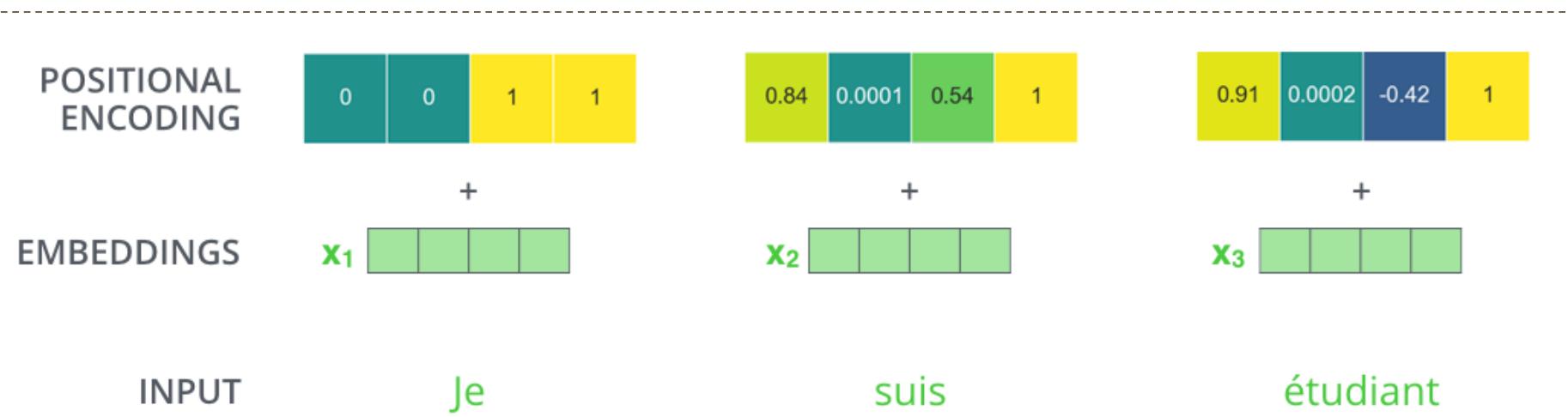


figure from <http://www.poetrypicture.club/the-cat-sat-on-the-mat/>

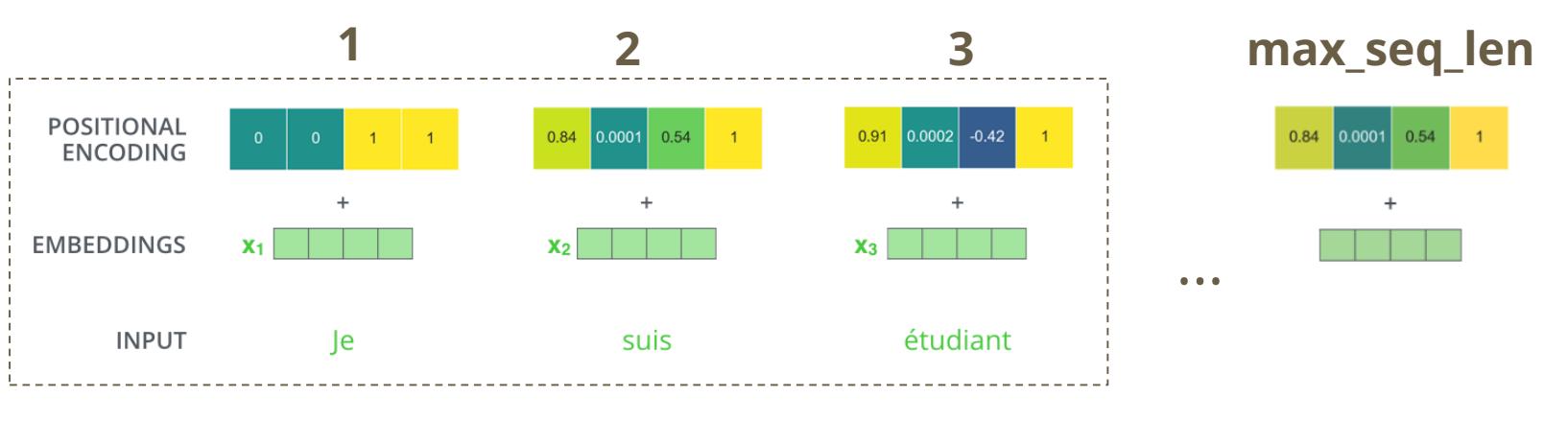
# Positional Encoding

- add position-aware encodings to the word embeddings so that *order matters*
- these encodings are either:
  - fixed and precomputed (no learned parameters)
  - learned embeddings



# Absolute Positional Encoding

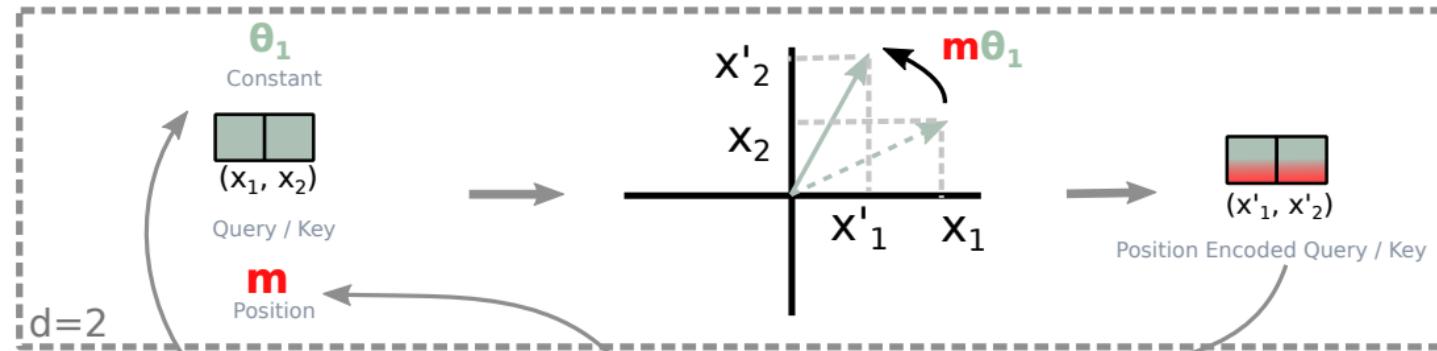
- initialize random vectors for positions [1.. **max\_seq\_len**] then learn them
- pros:
  - model learns how to represent position information
- cons:
  - cannot extrapolate outside maximum sequence length



# Relative Positional Encoding (RoPE)

each token embedding is rotated by an angle  $m\theta_i$ :

- $m$  is the **position** in the sequence
- $\theta_i$  controls the **frequency** (how slow or fast to rotate)



# Relative Positional Encoding (RoPE)

$m=1$ : rotate  $\mathbf{x}$  with  $\theta_i$

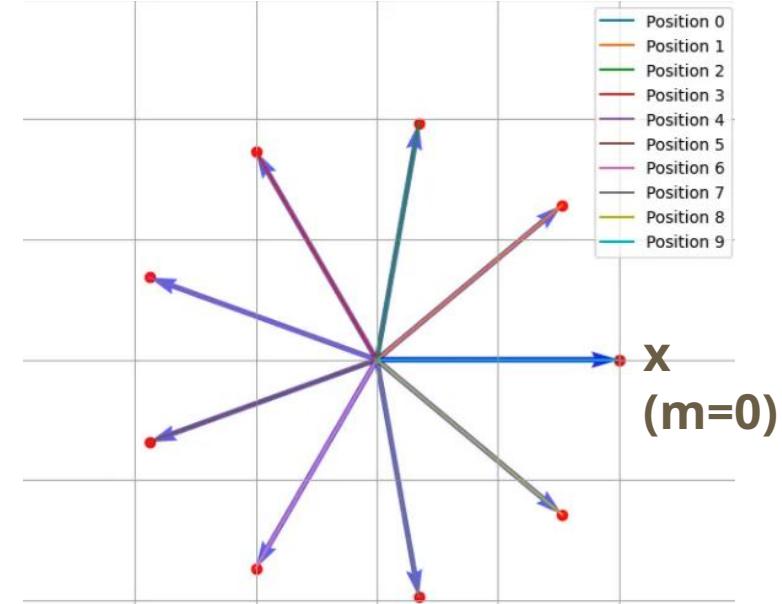
$m=2$ : rotate  $\mathbf{x}$  with  $2\theta_i$

...

## pros:

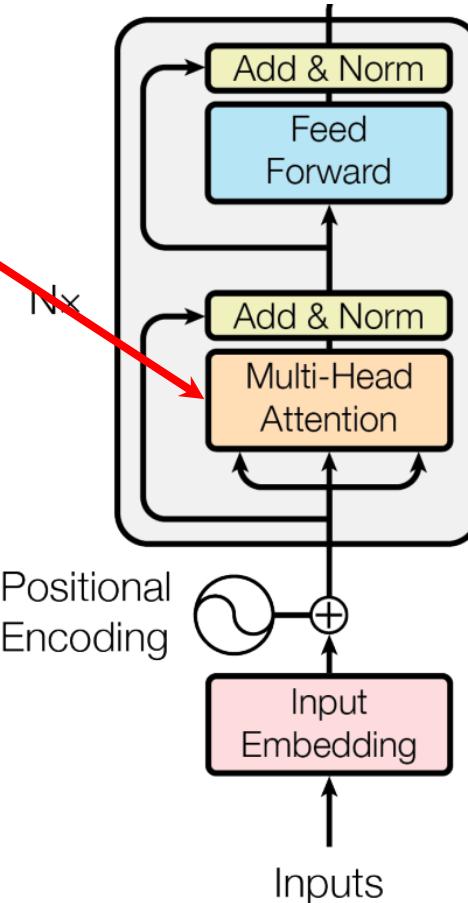
no need to learn parameters,  $\theta_i$  is fixed

**position extrapolation:** we can encode positions  $m$  that are past the maximum sequence length (just rotate with  $m\theta$ )



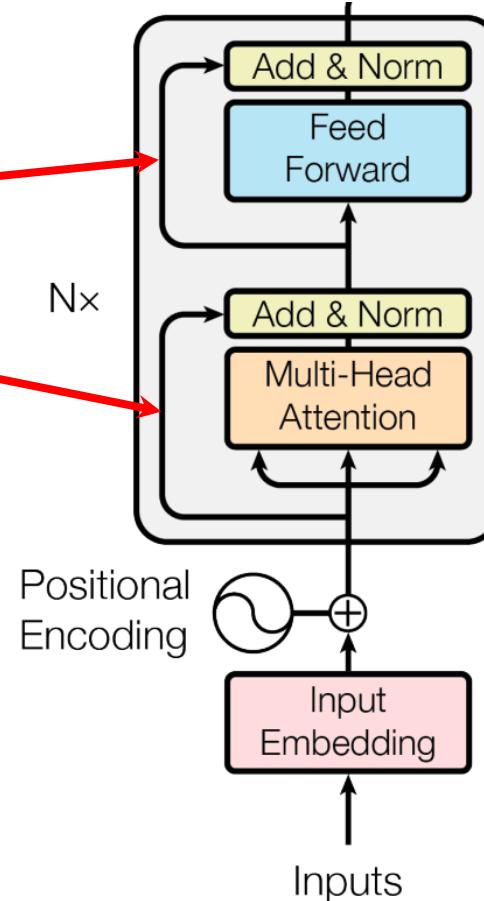
# Transformer recap

- multi-head self-attention:
  - several self-attention heads
  - each self-attention head *mixes inputs* based on the similarities between them
- positional encoding:
  - injects positional information into the sequence



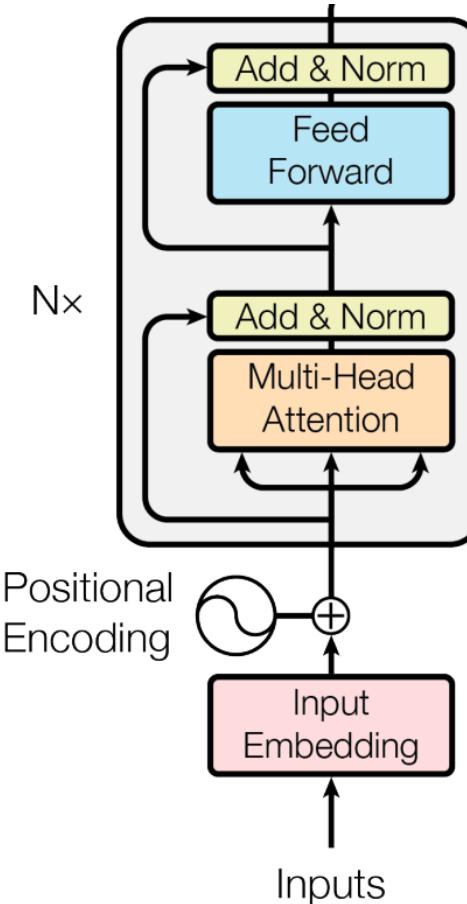
# Transformer

Transformer block has other ingredients as well  
what are these?



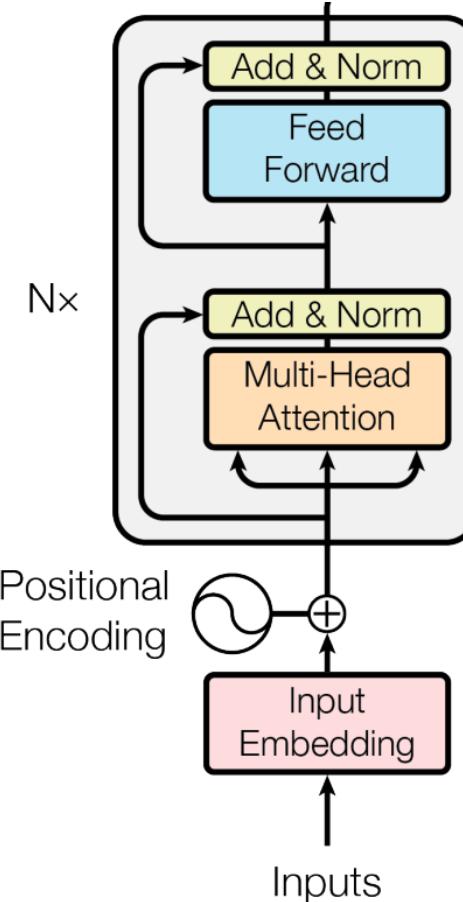
# Residual connections

- help with vanishing gradients:
  - not just a problem for RNNs or Deep Convolutional models
  - Transformer stacks several blocks on top of each other (N=32 for Llama3 8B)
- helps preserve information:
  - easier access to positional information or syntactic features from lower layers



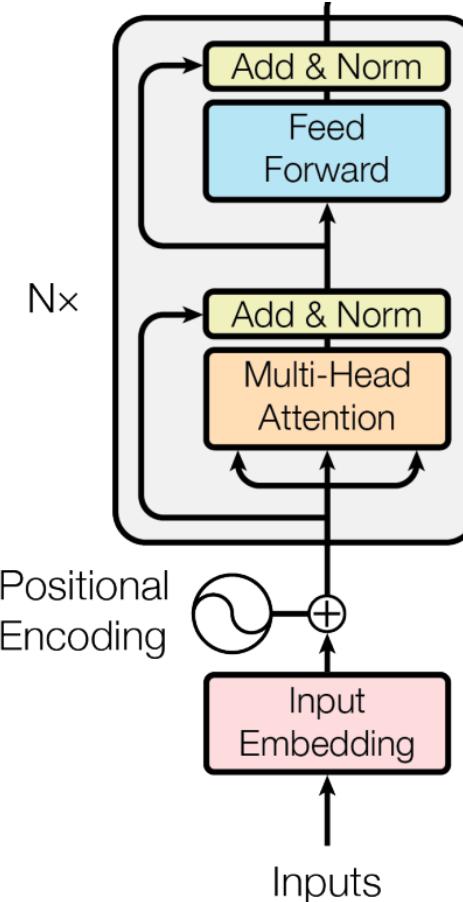
# Transformer

- residual connections make learning easier at depth, by preventing vanishing gradient
- what happens to output when adding updates over  $N$  layers?



# Transformer

- residual connections make learning easier at depth, by preventing vanishing gradient
- what happens to output when adding updates over N layers?
  - the magnitude grows
- solution: normalize features



# LayerNorm

1. normalizes across features (not examples)

- o make input have mean=0, std=1

$\mathbf{z}$  before: [2.0, 4.0, 6.0, 8.0, 10.0]

$\mathbf{z}$  after: [-1.41, -0.71, **0.0**, 0.71, 1.41 ]

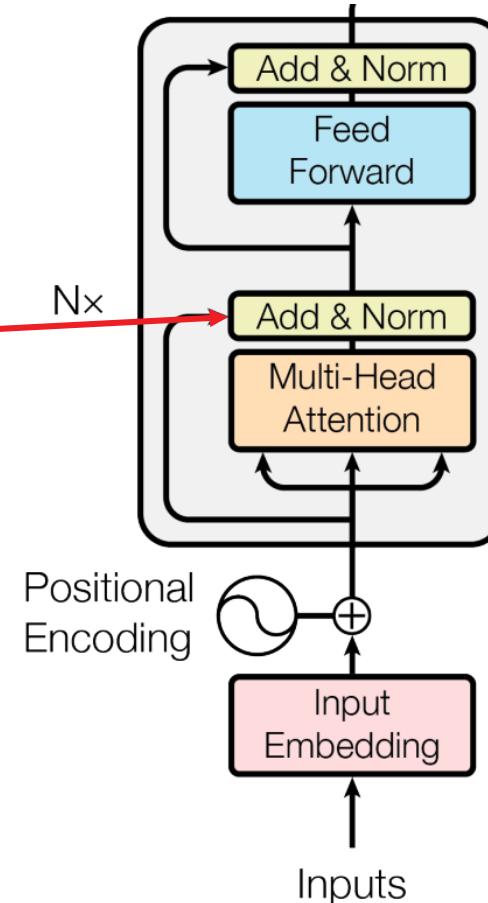
$$LN(\mathbf{z}; \alpha, \beta) = \frac{(\mathbf{z} - \mu)}{\sigma} \odot \alpha + \beta,$$

$$\mu = \frac{1}{D} \sum_{i=1}^D z_i, \quad \sigma = \sqrt{\frac{1}{D} \sum_{i=1}^D (z_i - \mu)^2},$$

1. recenter and rescale input with  $\alpha$  and  $\beta$

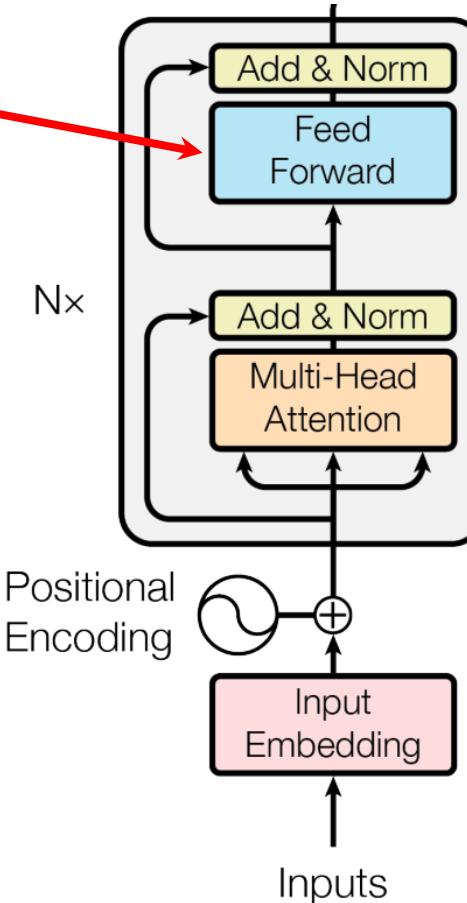
# LayerNorm

- applied after Multi-Head Self Attention (MHSA) and Feedforward layer
- original model: **add then normalize**
  - output = **LayerNorm**( $x + \text{Attention}(x)$ )
  - makes output well behaved before next layer
- nowadays: **normalize then add**
  - output =  $x + \text{LayerNorm}(\text{Attention}(x))$
  - more stable in deep models: during backprop, **gradient flows directly through  $x$**  without being modified by **LayerNorm**



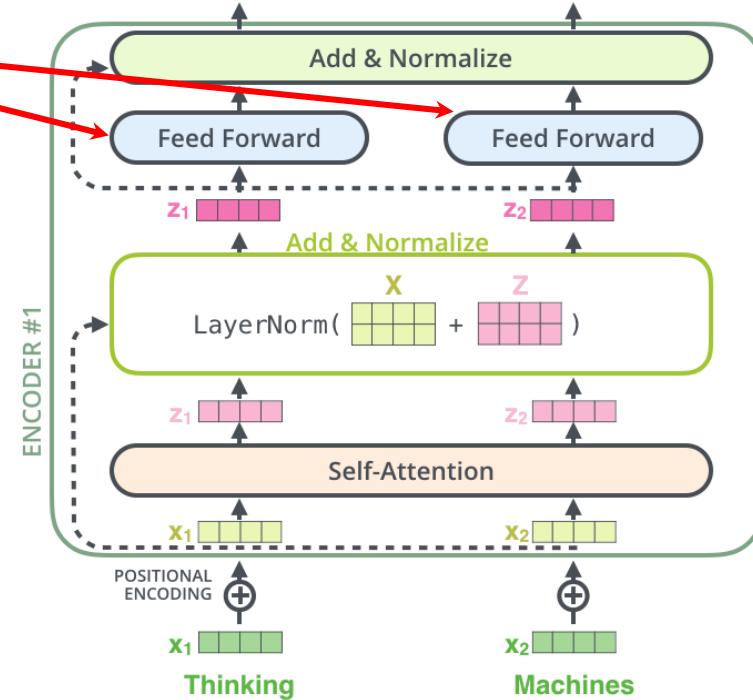
# Transformer FFN

- feed-forward network (FFN) processes outputs resulted from Multi-Head Attention + Norm
- unlike Attention, which mixes all tokens, FFN is a **per-token operation**
  - it enriches the features of each token output



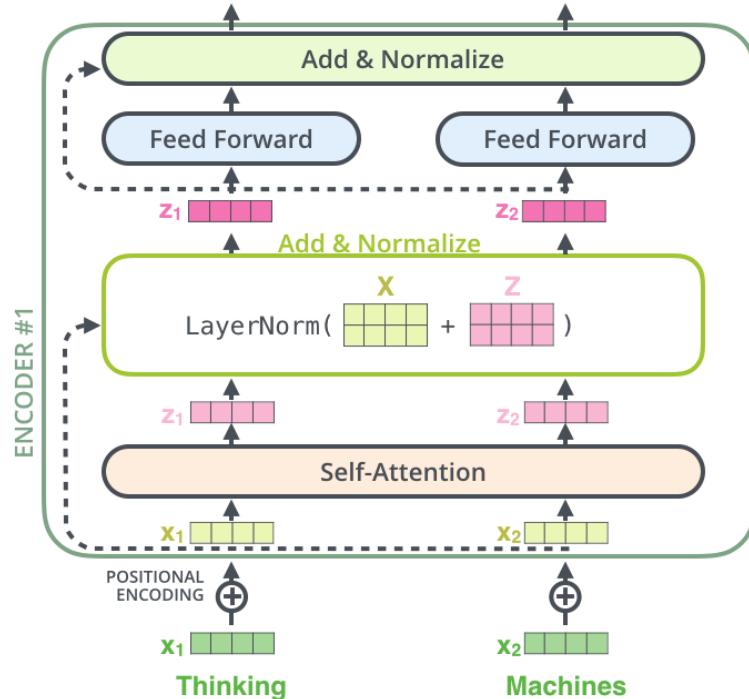
# Transformer FFN

- FFN applied on each output  $x_i$ :
  - $\text{FFN}(x_i) = W_2 \cdot \text{ReLU}(W_1 \cdot x_i + b_1) + b_2$
  - $W_1$  projects output from  $d_{\text{model}}$  to **larger hidden size** (usually 4x)
- larger hidden size:
  - allows increased expressivity



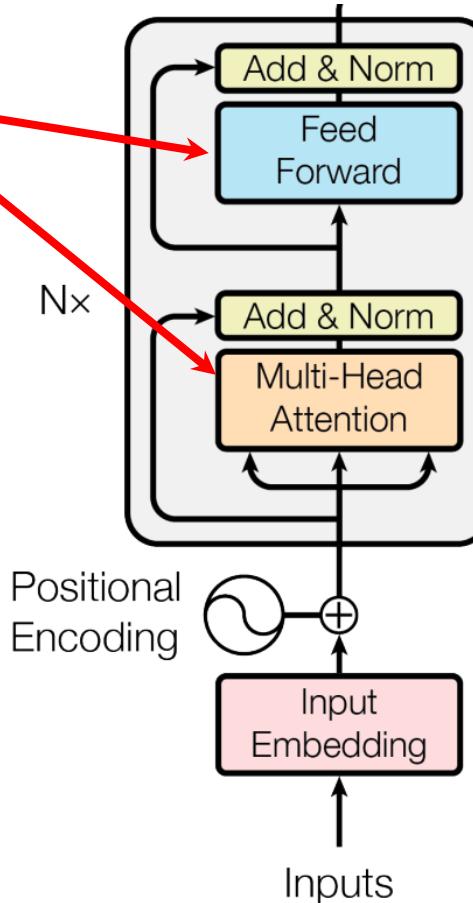
# SwiGLU

- modern Transformer models replace FFN with a gated linear transform:
  - $\text{SwiGLU}(\mathbf{x}_i) = \text{Swish}(\mathbf{W}\mathbf{x}_i + \mathbf{b}) \odot \mathbf{V}\mathbf{x}_i + \mathbf{c}$
- dynamic gating mechanism to filter irrelevant features or focus on important ones
- better empirical performance than vanilla FFN



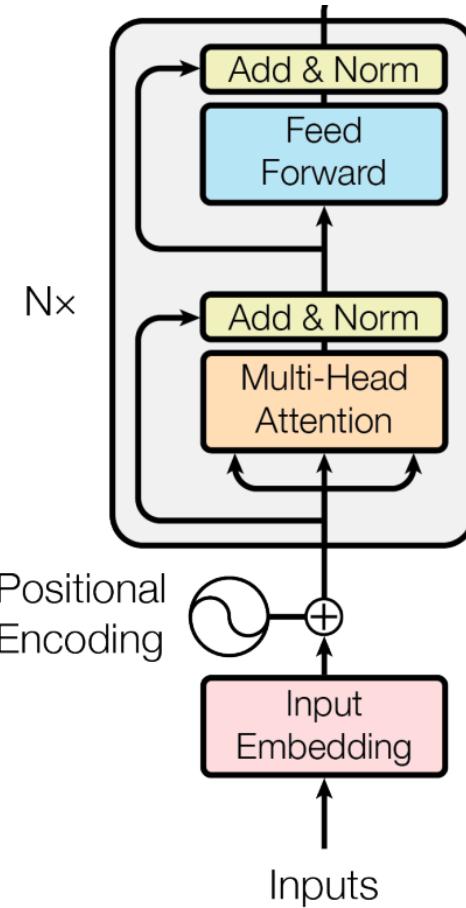
# Attention vs FFN layers

- FFN layers dominate parameters count:
  - takes up  $\sim \frac{2}{3}$  of params in a Transformer block
- evidence that FFN is a core component:
  - acts as a [memory](#)
  - learned associations (Berlin->Germany) reside in the FFN



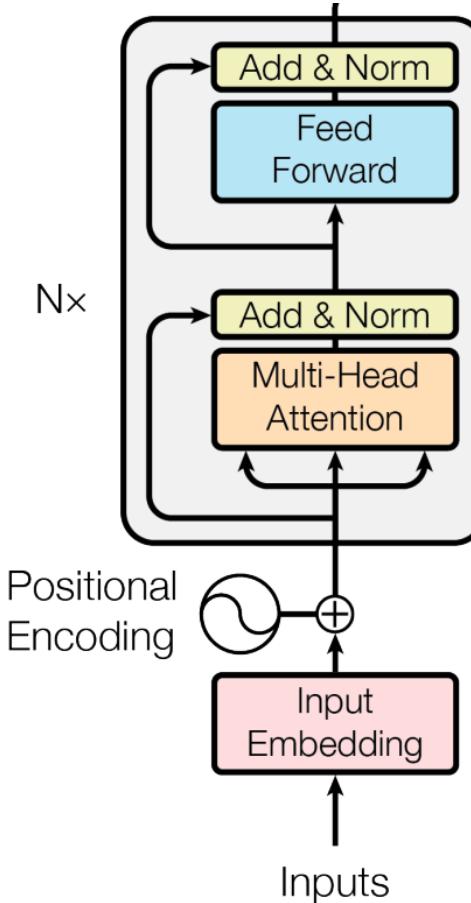
# Transformer diff: 2017 vs today

- self-attention as a concept has remained largely the same
- some notable improvements:
  - *self-attention variants*: global attention => **window attention/linear attention**
  - *hardware efficiency*: **FlashAttention**
  - *positional encoding*: sine/learned => **Rotary (RoPe)**
  - *norm layers*: LayerNorm => **RmsNorm**
  - *norm layers position*: after residual => **before**
  - *activation function in FFN*: Relu => **SwiGLU**



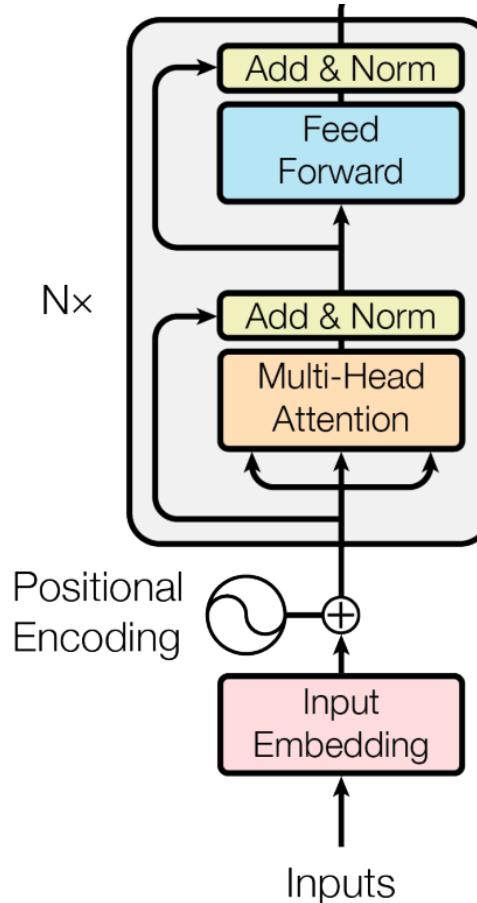
# Transformer block - recap

- recurrent-free, parallelizable alternative for sequential data
- constant gradient path (every element **directly interacts** with all the other elements)
- building block for SOTA architectures in NLP:
  - encoders: [BERT](#)
  - decoders: Large Language Models (GPT-series)

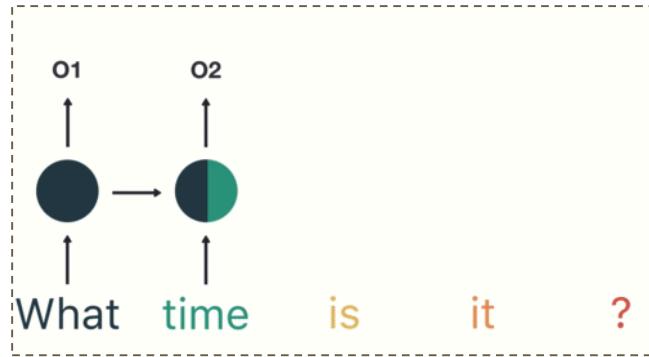


# Transformers: beyond sequential data

- versatile across domains:
  - applied to images (Visual Transformers), graphs and sets
  - **not limited to sequential data**
- it models relationships between elements rather than strict order:
  - positional information can be injected or not depending on the task



# RNN vs self-attention



in RNNs, sequence elements are processed **step by step**

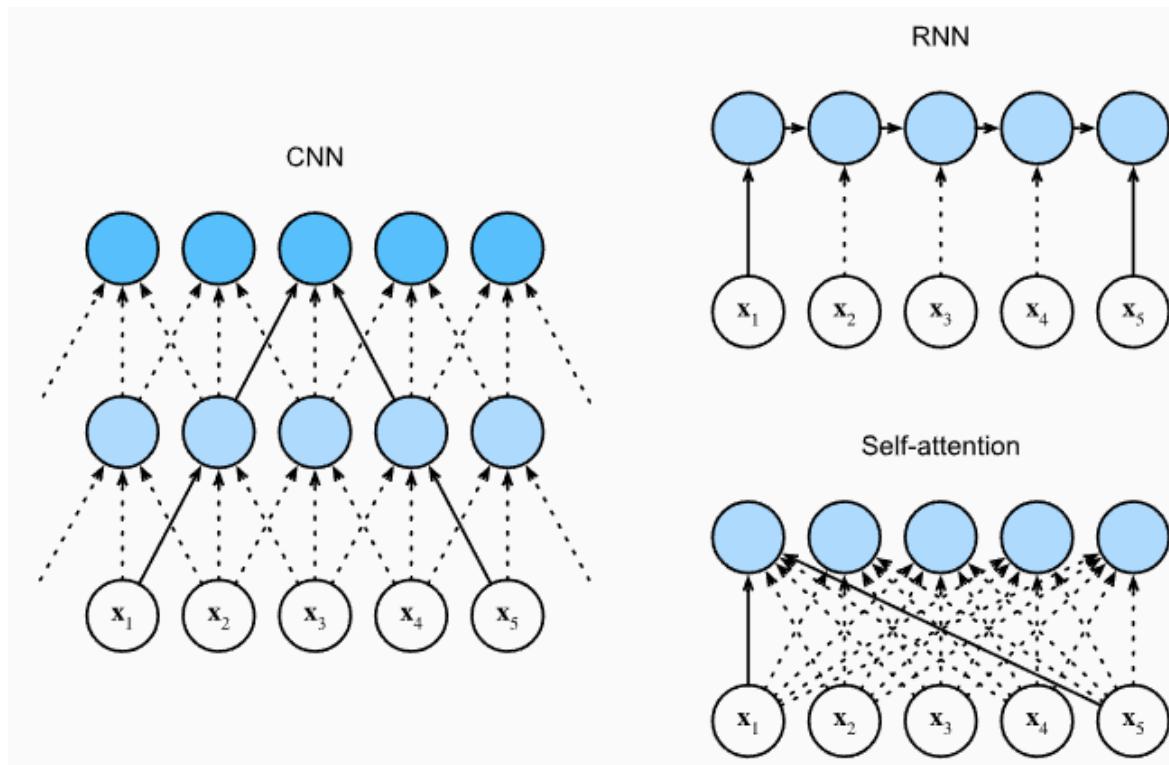


in self-attention, sequence elements are processed **all at once**

# RNN vs CNN vs self-attention

RNN ( $O(Nd^2)$ ):

- cannot be parallelized across sequence



[source](#)

# RNN vs CNN vs self-attention

RNN ( $O(Nd^2)$ ):

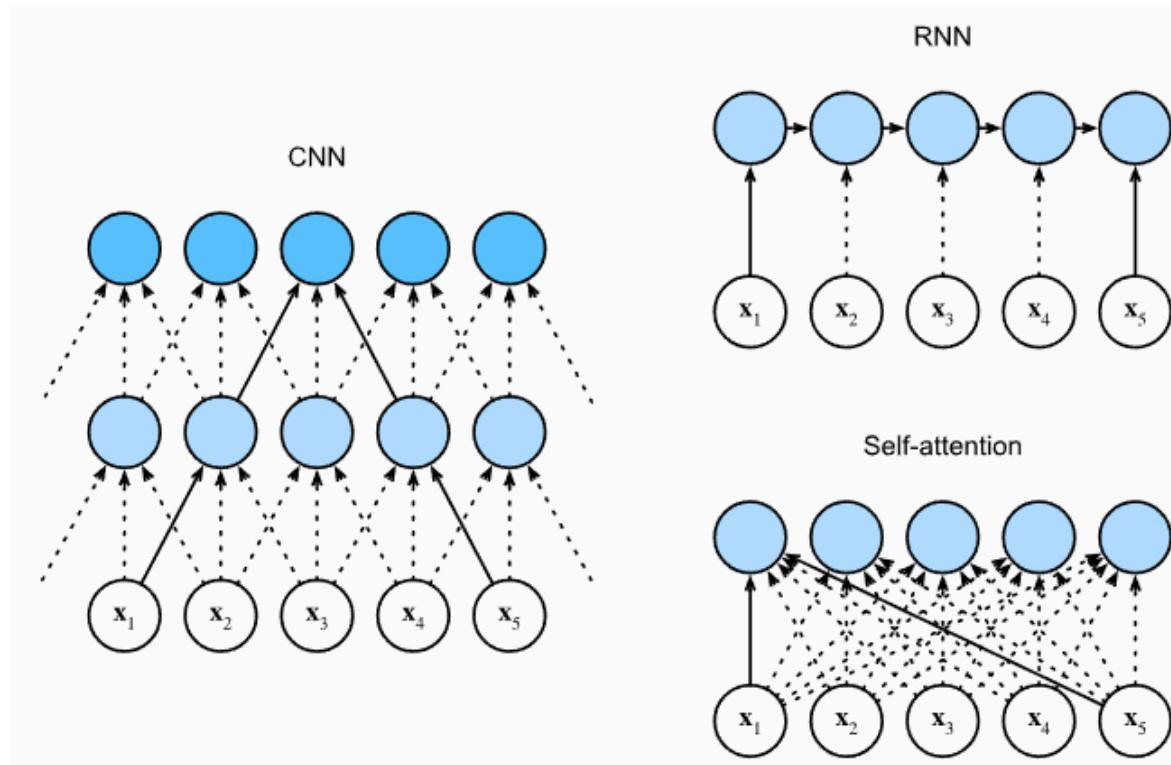
- cannot be parallelized across sequence
- **maximum path length  $O(N)$**
- bias towards recent past (due to sequential nature)

CNN ( $O(kNd^2)$ ):

- parallelizable across sequence
- **maximum path length  $O(N/k)$**

self-attention ( $O(N^2d)$ ):

- parallelizable across sequence
- **maximum path length  $O(1)$**  => equal preference for distant or recent past



[source](#)

# RNN vs CNN vs self-attention

RNN ( $O(Nd^2)$ ):

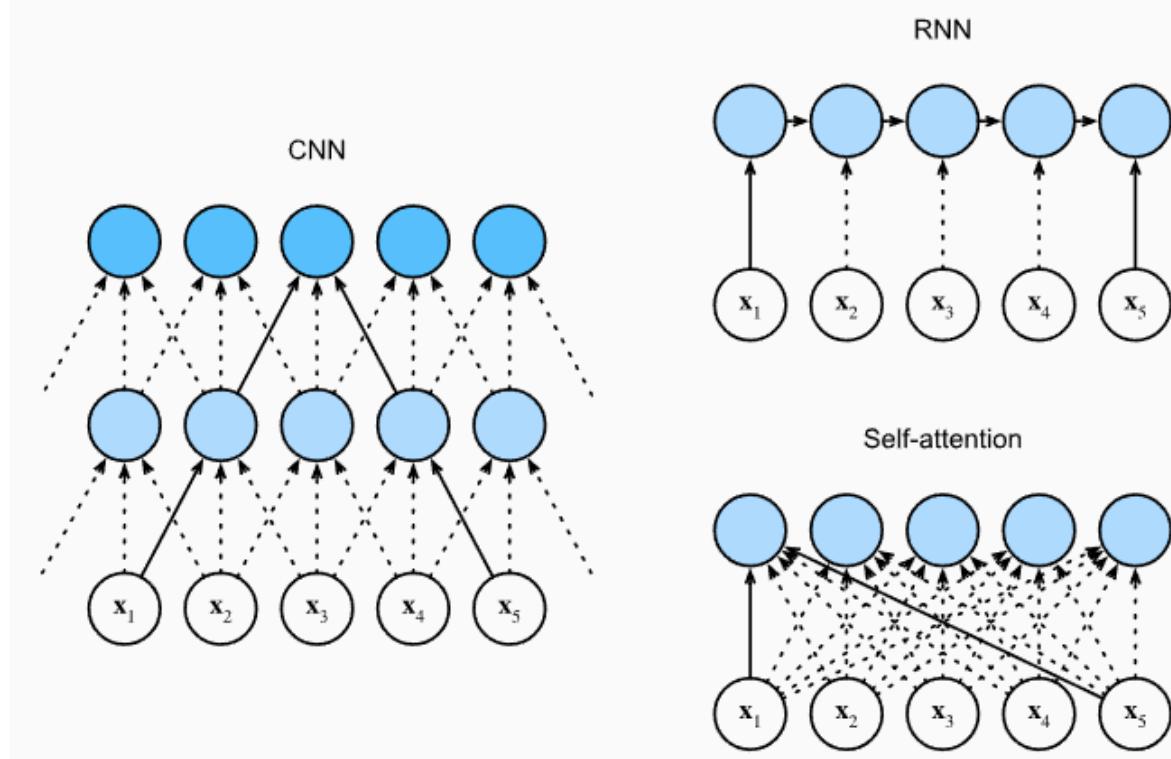
- cannot be parallelized across sequence
- maximum path length  $O(N)$
- bias towards recent past (due to sequential nature)
- **fast generation:** no need to recompute sequence for each new token (history already stored in previous state)

CNN ( $O(kNd^2)$ ):

- parallelizable across sequence
- maximum path length  $O(N/k)$
- **slow generation:** recompute sequence for each new token

self-attention ( $O(N^2d)$ ):

- parallelizable across sequence
- maximum path length  $O(1) \Rightarrow$  equal preference for distant or recent past
- **slow generation:** recompute sequence for each new token



[source](#)

# Transformers vs RNNs

	Training	Inference
Transformers	<b>Fast!</b> (parallelizable)	<b>Slow...</b> (scales <b>quadratically</b> with sequence length)
RNNs	<b>Slow...</b> (not parallelizable)	<b>Fast!</b> (scales <b>linearly</b> with sequence length)

# Combining best aspects of RNNs and Transformers

- linear attention:
  - reduces attention complexity from  $O(N^2d)$  to  $O(Nd^2)$  just like RNNs
  - scales linearly with sequence length => **faster training and inference than standard attention**

# Combining best aspects of RNNs and Transformers

- linear attention:
  - reduces attention complexity from  $O(N^2d)$  to  $O(Nd^2)$  just like RNNs
  - scales linearly with sequence length => **faster training and inference than standard attention**
- state space models (Mamba):
  - an RNN-like architecture without non-linearities
  - **parallelizable => faster training than traditional RNNs**
  - linear scaling in sequence length (just like RNNs) => **fast inference**

# Combining best aspects of RNNs and Transformers

- linear attention:
  - reduces attention complexity from  $O(N^2d)$  to  $O(Nd^2)$  just like RNNs
  - scales linearly with sequence length => **faster training and inference than standard attention**
- state space models (Mamba):
  - an RNN-like architecture without non-linearities
  - **parallelizable => faster training than traditional RNNs**
  - linear scaling in sequence length (just like RNNs) => **fast inference**
- no free lunch:
  - both approaches trade expressivity and performance for better efficiency

- RNNs handle **sequential data** (but suffer from **vanishing / exploding gradients**)
- *clipping gradients and gated architectures* (**LSTM/GRU**) address gradients or
- **Transformers** is a parallelizable alternative to recurrent networks
- recent efforts try to improve train and inference efficiency while also retaining expressivity



or



# TODAY

---

# Thank you!

(Next: Natural Language Processing Applications)

# Resources

## RNNs:

[http://www.cs.toronto.edu/~rgrosse/courses/csc321\\_2018/](http://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/) - Lectures 15-16

<https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/> - Lecture 12-13

[http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf) - Lecture 10

<http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

GRU: <https://arxiv.org/abs/1406.1078>

LSTM: <https://www.bioinf.jku.at/publications/older/2604.pdf>

## Transformers:

Understanding Deep Learning, Simon Prince, chapter 12

Illustrated Transformer (blog): <https://jalammar.github.io/illustrated-transformer/>

Annotated Transformer: <https://nlp.seas.harvard.edu/annotated-transformer/>

Linear Attention (blog): <https://haileyschoelkopf.github.io/blog/2024/linear-attn/>

**State Space Models:** <https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-mamba-and-state>

**NLP course:** <http://cs224d.stanford.edu/syllabus.html>