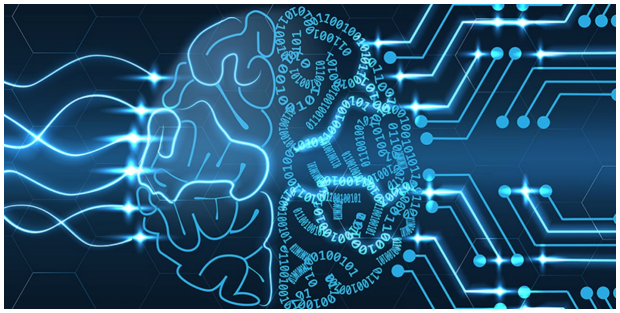
AI6130
Large Language Models

Lecture 2: Neural Network and Optimization Basics

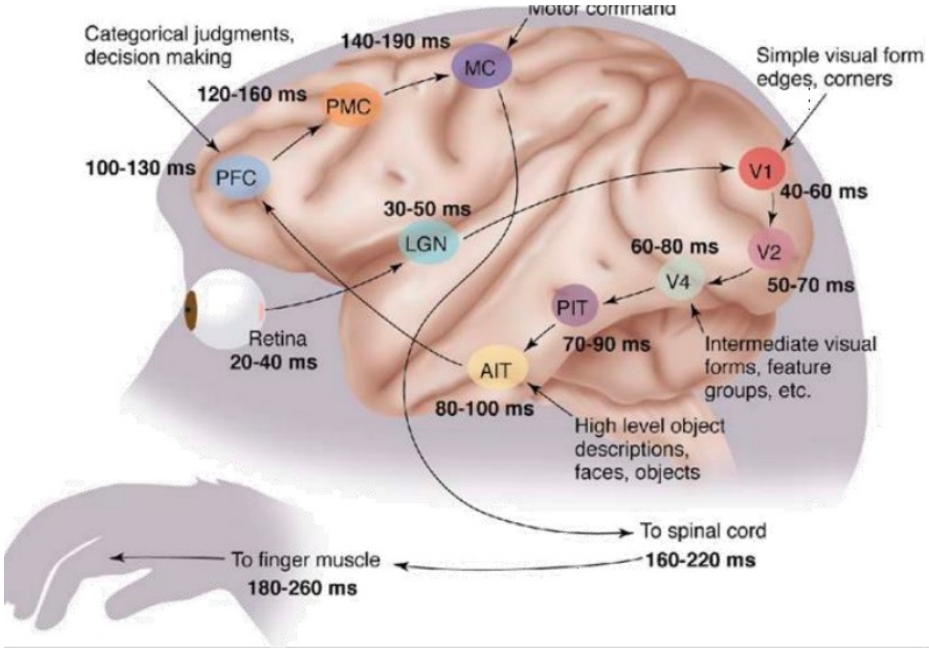


Instructor: Luu Anh Tuan

Email: anhtuan.luu@ntu.edu.sg

Office: #N4-02c-86

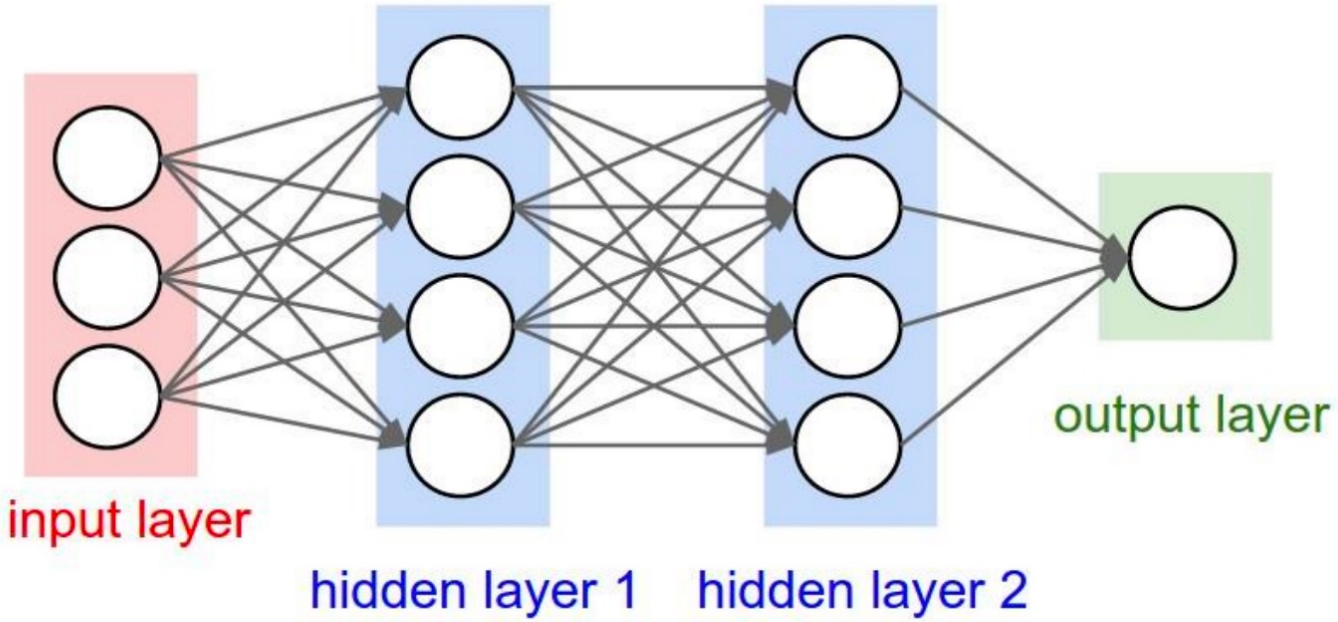
NEURAL NETWORKS: INSPIRED BY THE BRAIN



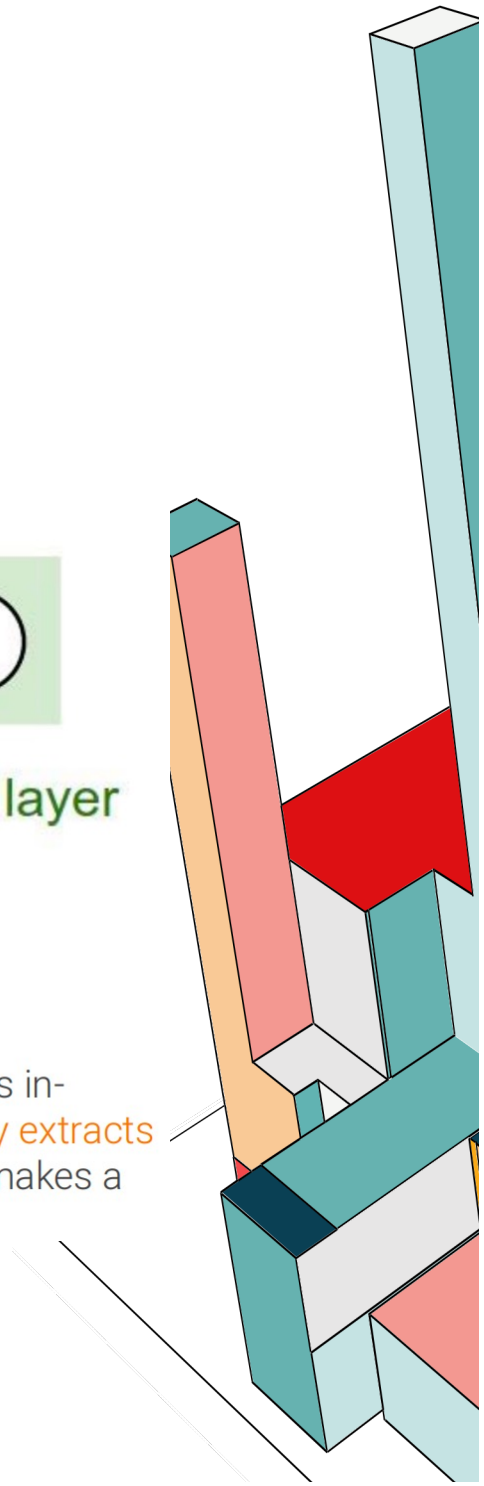
The first hierarchy of neurons that receives information in the visual cortex are sensitive to specific edges while brain regions further down the visual pipeline are sensitive to more complex structures such as faces.

Our brain has lots of neurons connected together and the strength of the connections between neurons represents long term knowledge.

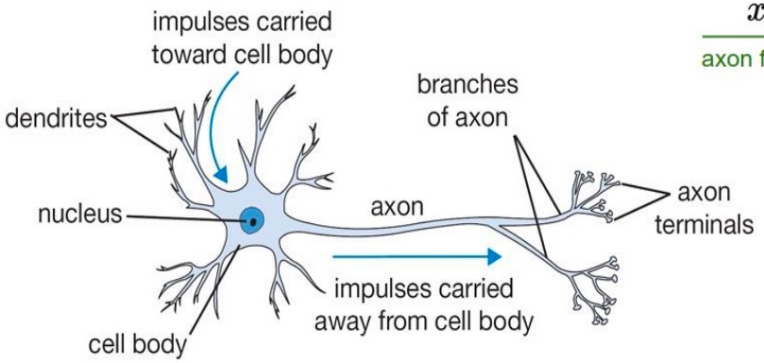
DEEP LEARNING - BASICS
ARTIFICIAL NEURAL NETWORKS

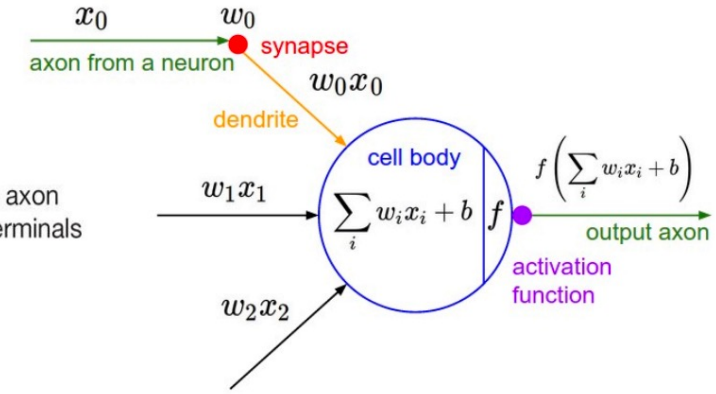


Consists of one input, one output and multiple fully-connected hidden layers in-between. Each layer is represented as a series of neurons and progressively extracts higher and higher-level features of the input until the final layer essentially makes a decision about what the input shows.



DEEP LEARNING - BASICS
THE NEURON





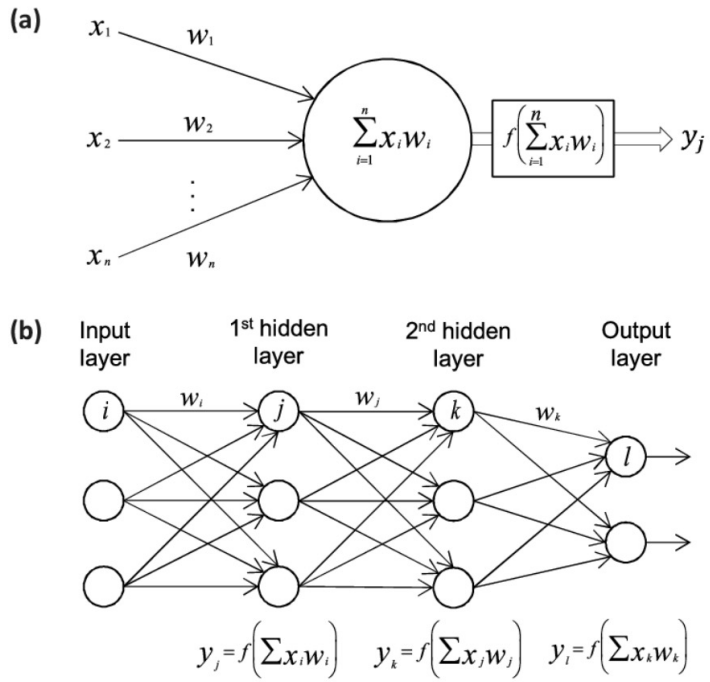
An artificial neuron contains a nonlinear activation function and has several incoming and outgoing weighted connections.

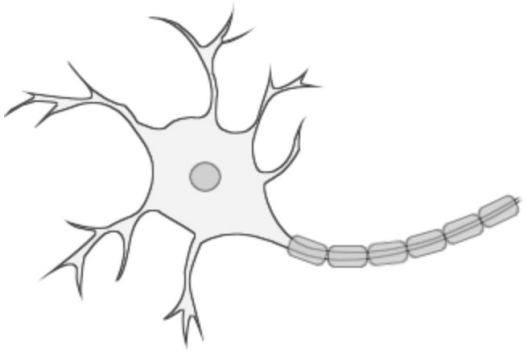
Neurons are trained to filter and detect specific features or patterns (e.g. edge, nose) by receiving weighted input, transforming it with the activation function und passing it to the outgoing connections.

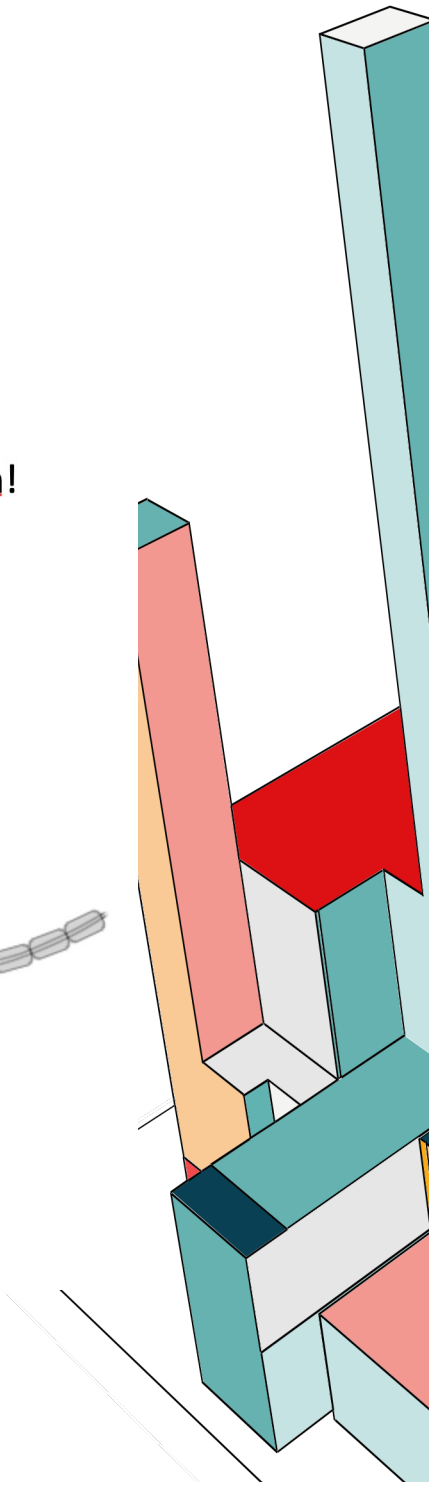
DEEP LEARNING - BASICS
THE NEURON

If you understand how logistic regression works

Then **you already understand** the operation of a basic neural network neuron!





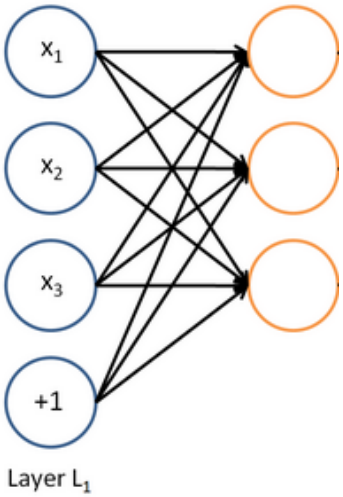


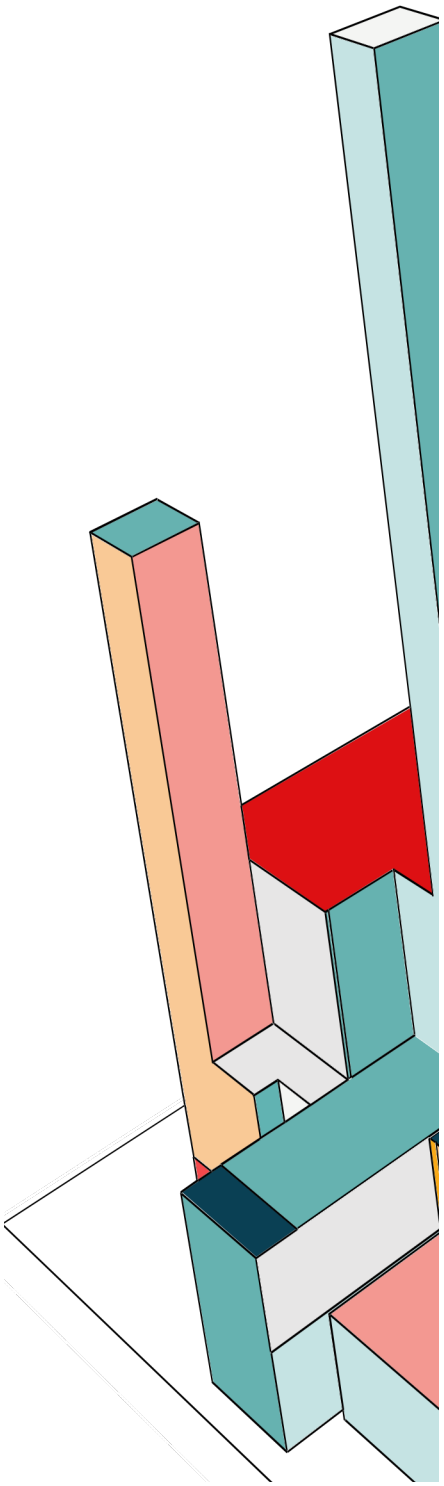
From LR to NN

A neural network = running several logistic regressions at the same time

If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs ...

But these are not random variables





From LR to NN

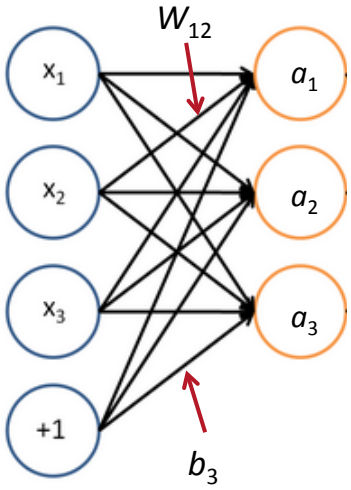
$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$
$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$
.....

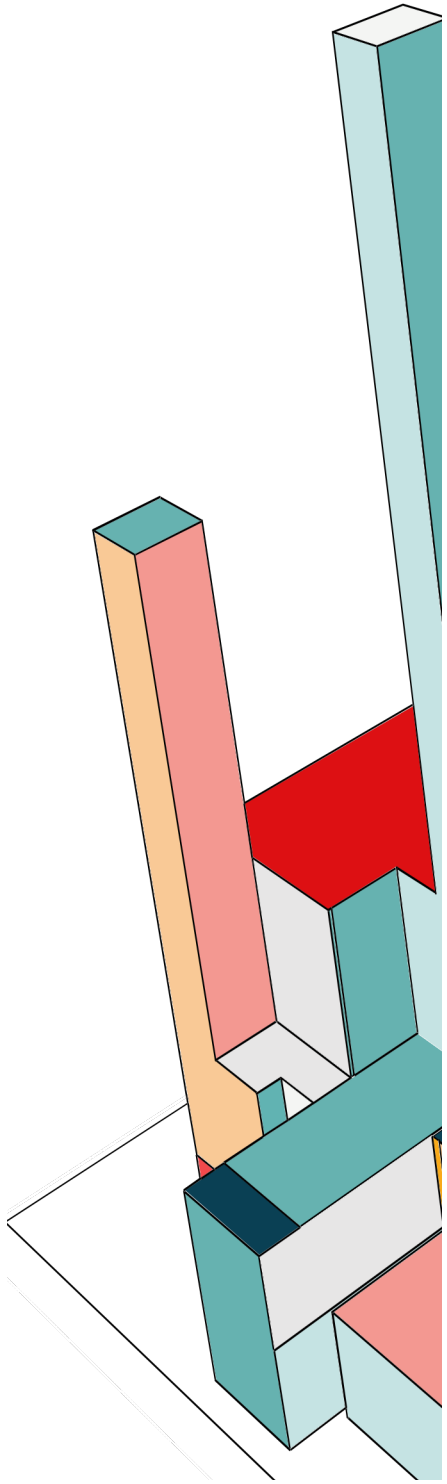
In Matrix Notation

$$z = Wx + b$$
$$a = f(z)$$

Where f() is applied element-wise

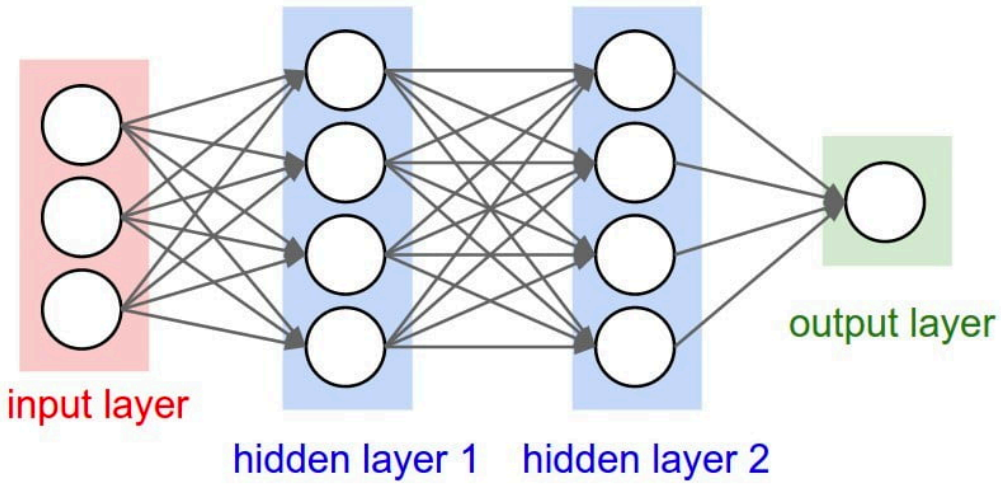
$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$



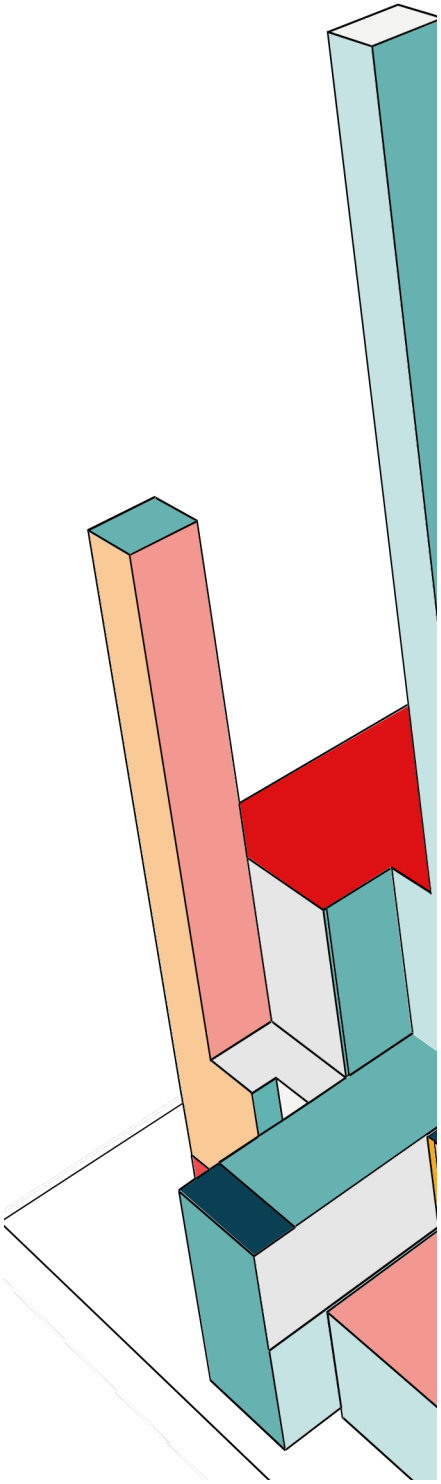


From LR to NN

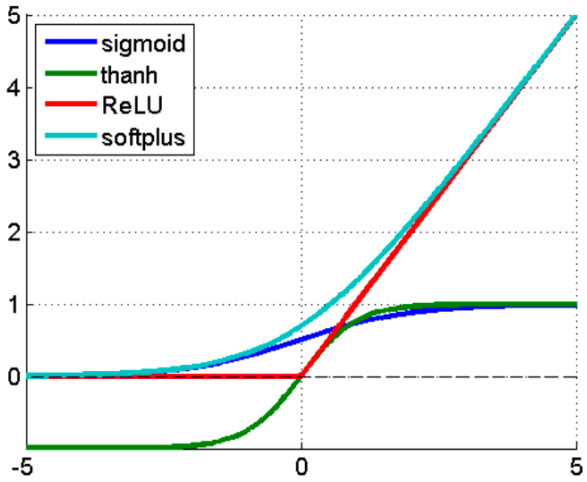
... which we can feed into another logistic regression function



Output layer can be a classification layer or a regression layer

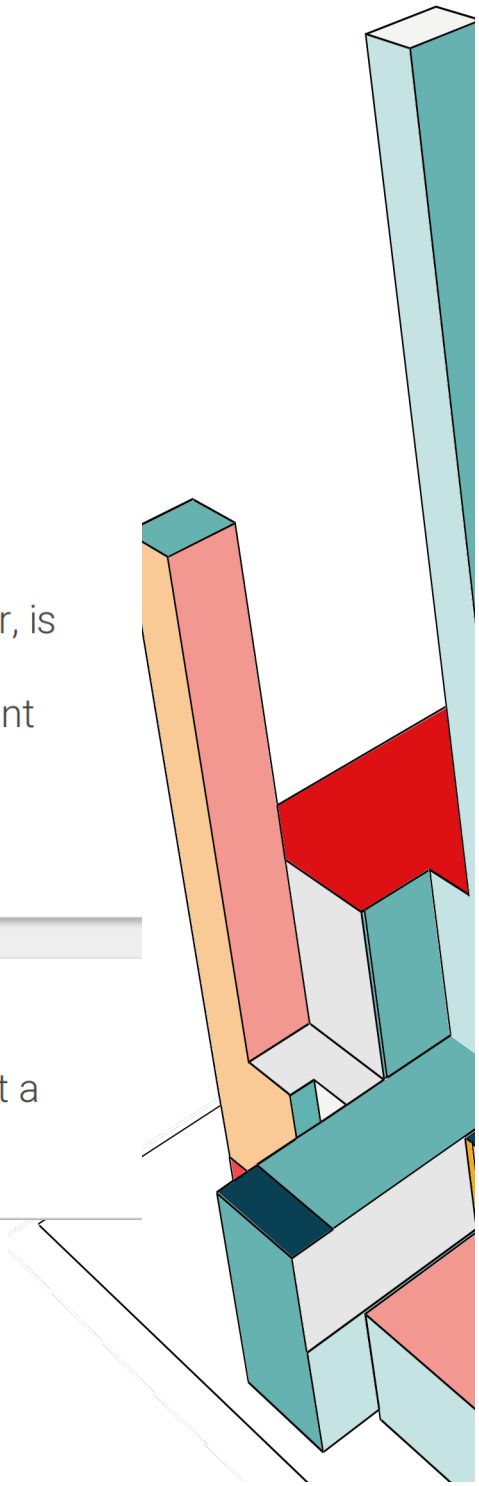


DEEP LEARNING - BASICS
NON-LINEAR ACTIVATION FUNCTION



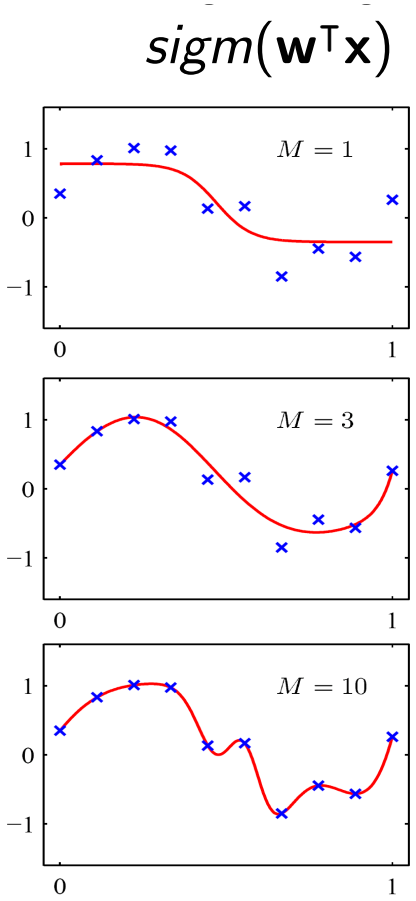
Most deep networks use ReLU - max(0,x) - nowadays for hidden layers, since it trains much faster, is more expressive than logistic function and prevents the gradient vanishing problem.

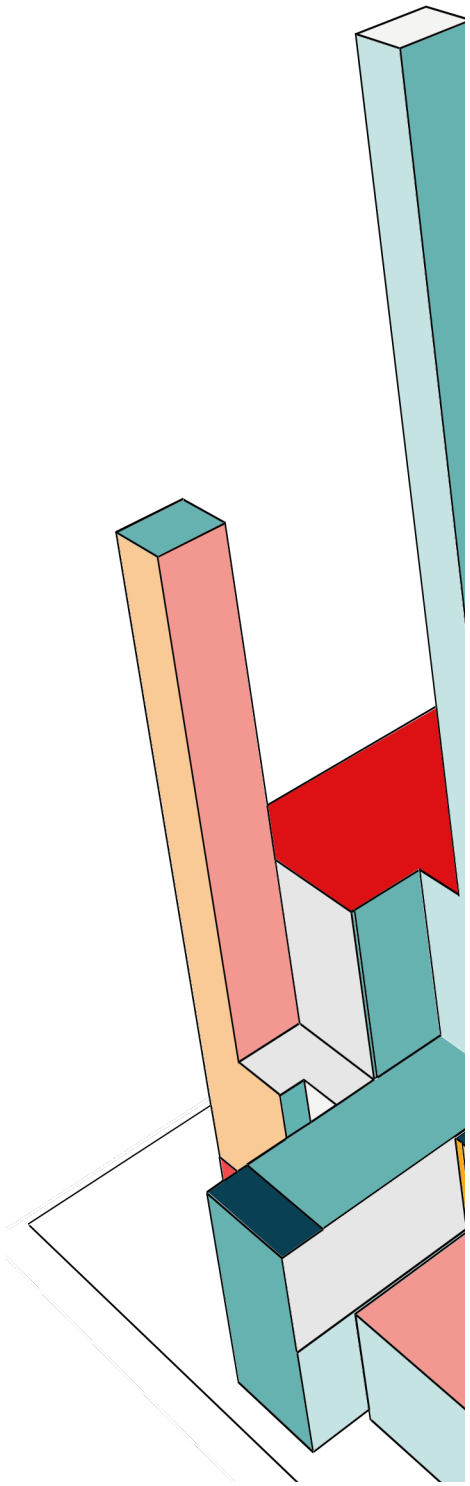
Non-linearity is needed to learn complex (non-linear) representations of data, otherwise the NN would be just a linear function.



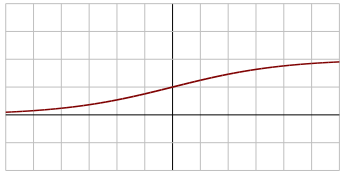
Why Non-linearity?

- For logistic regression: map to probabilities
- For NN: function approximation, e.g., regression or classification
  - Without non-linearities, NNs can't do anything more than a linear transform
  - Extra layers could just be compiled down into a single linear transform **W(Wx) = Vx**

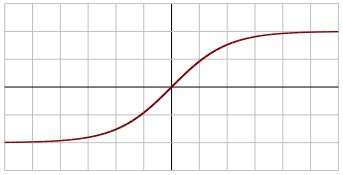




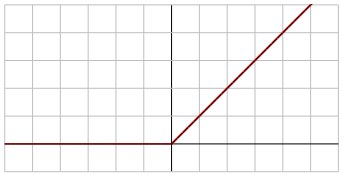
Activation Functions



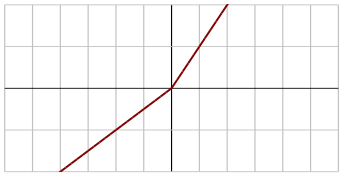
(a) Sigmoid



(b) tanh



(c) ReLU



(d) Leaky ReLU

- Sigmoid

$$\sigma(x) = \frac{1}{1 + \exp^{-x}}$$

- Hyperbolic Tangent:

$$\tanh(x) = \frac{\exp^{x} - \exp^{-x}}{\exp^{x} + \exp^{-x}}$$

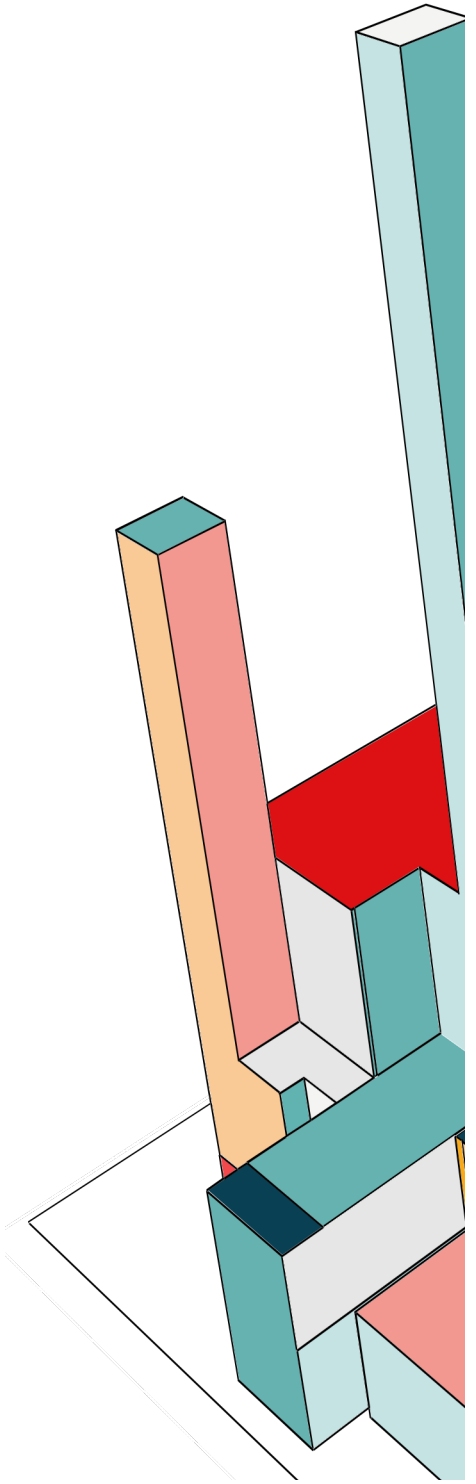
- Rectified Linear Unit (ReLU):

$$f(x) = \max(0, x)$$

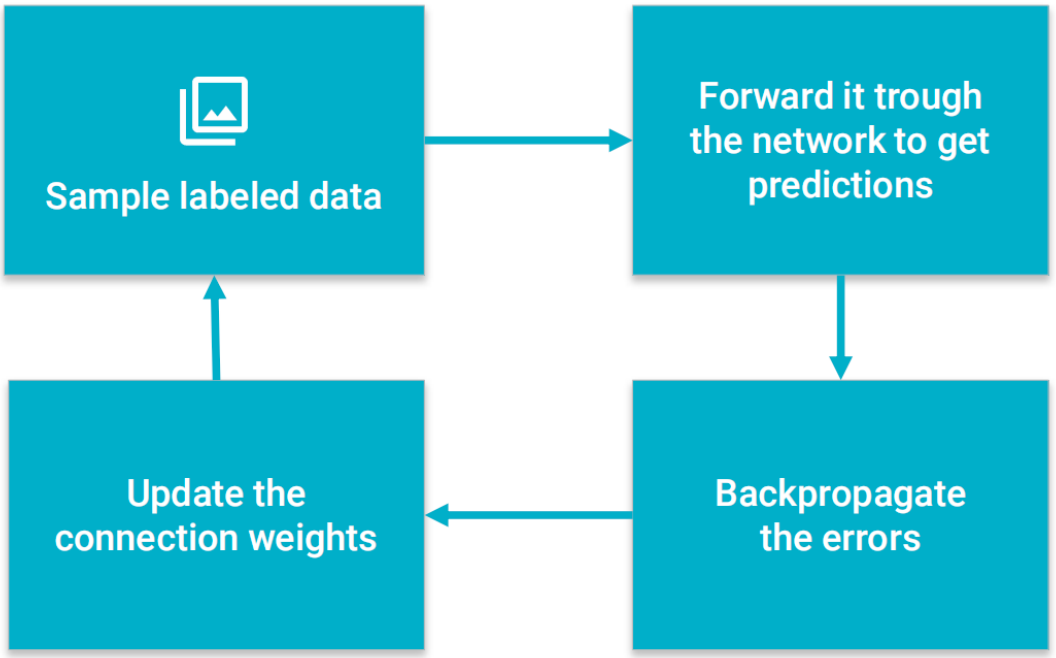
- Leaky ReLU:

$$f(x) = \begin{cases} \alpha x & x < 0 \\ x & x \geq 0 \end{cases}$$

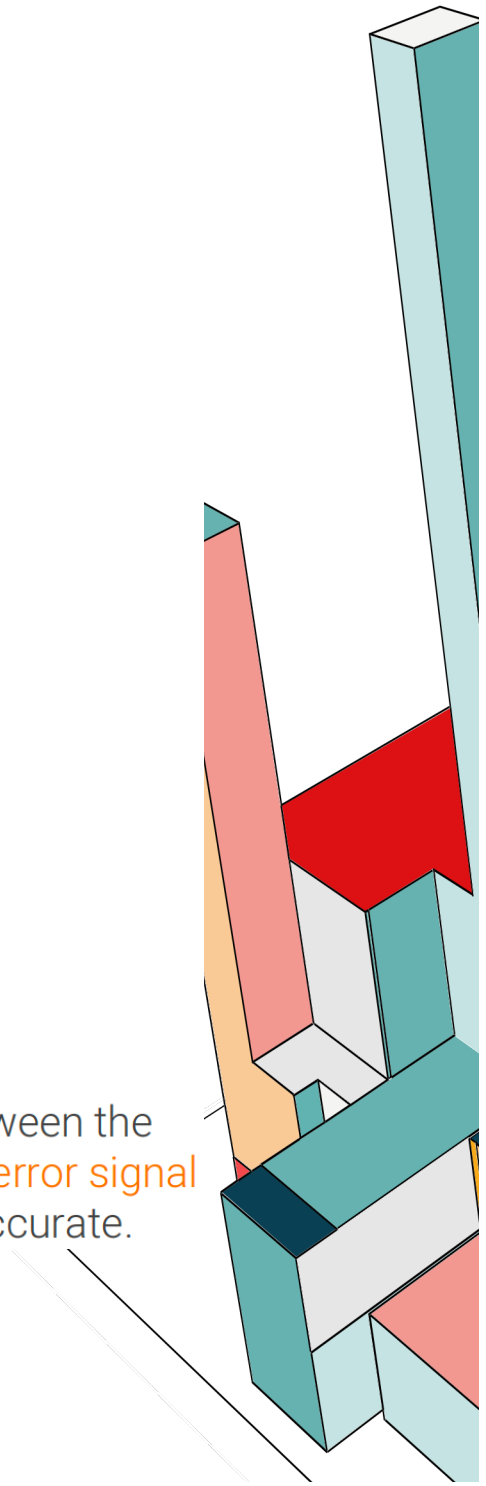
, where α is small constant



DEEP LEARNING - BASICS
THE TRAINING PROCESS



Learns by generating an error signal that measures the difference between the predictions of the network and the desired values and then using this error signal to change the weights (or parameters) so that predictions get more accurate.



Forward- & Back-propagation

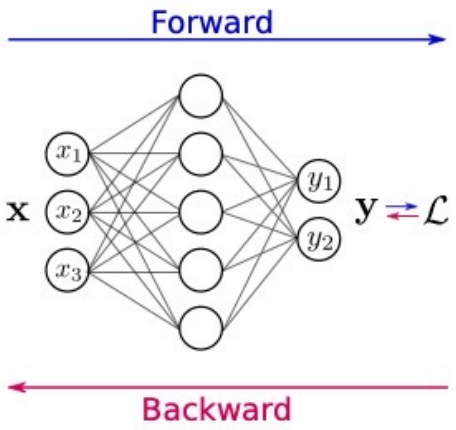
Two information flow directions:

Forward propagation

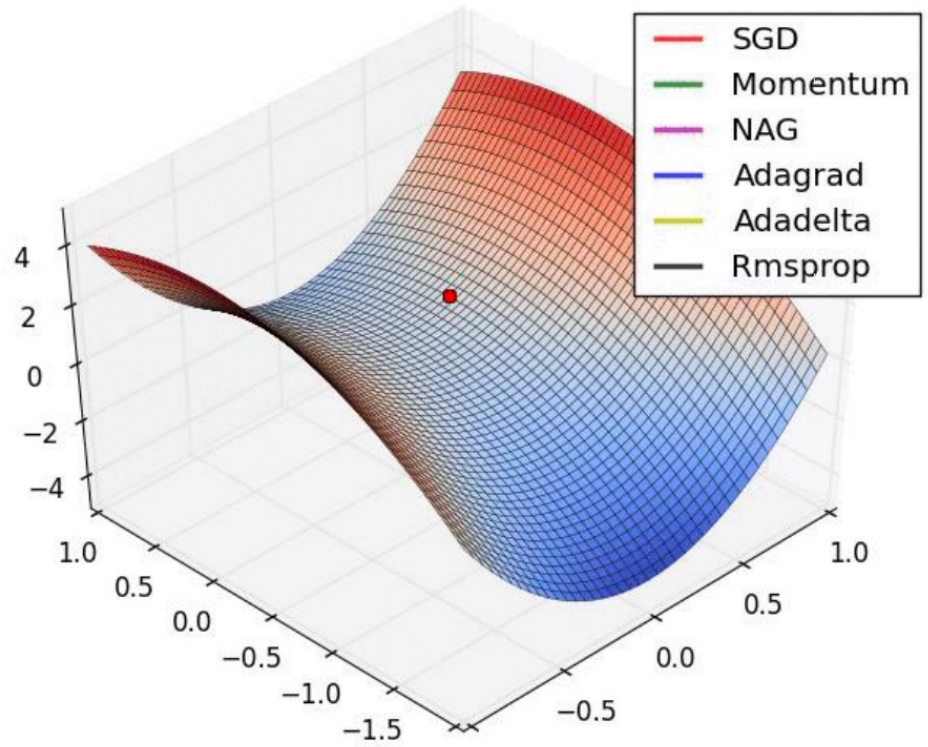
- NN accepts an input **x** and produces an output **y**
- During training, it continues onward until it produces a scalar cost **L**

Back-propagation

- Information (**gradients** with respect to the parameters) from the cost flows backward through the network



DEEP LEARNING - BASICS
GRADIENT DESCENT



Gradient Descent finds the (local) the minimum of the cost function (used to calculate the output error) and is used to adjust the weights.

Gradient Descent (Recall)

The gradient tells us how to change x (parameters) in order to make a small improvement in our goal.

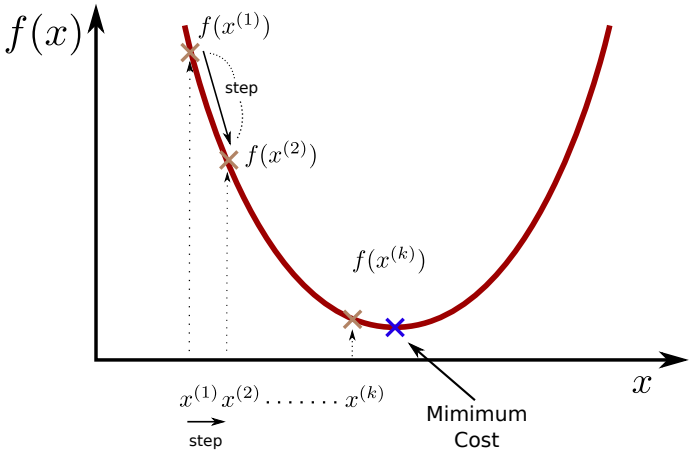
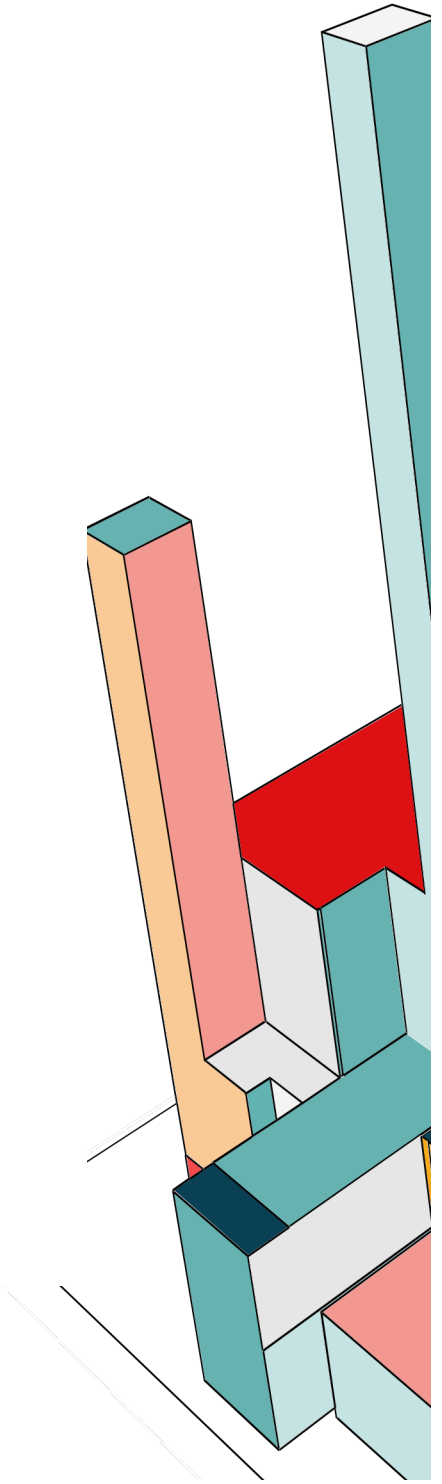


Figure: Illustration of gradient descent

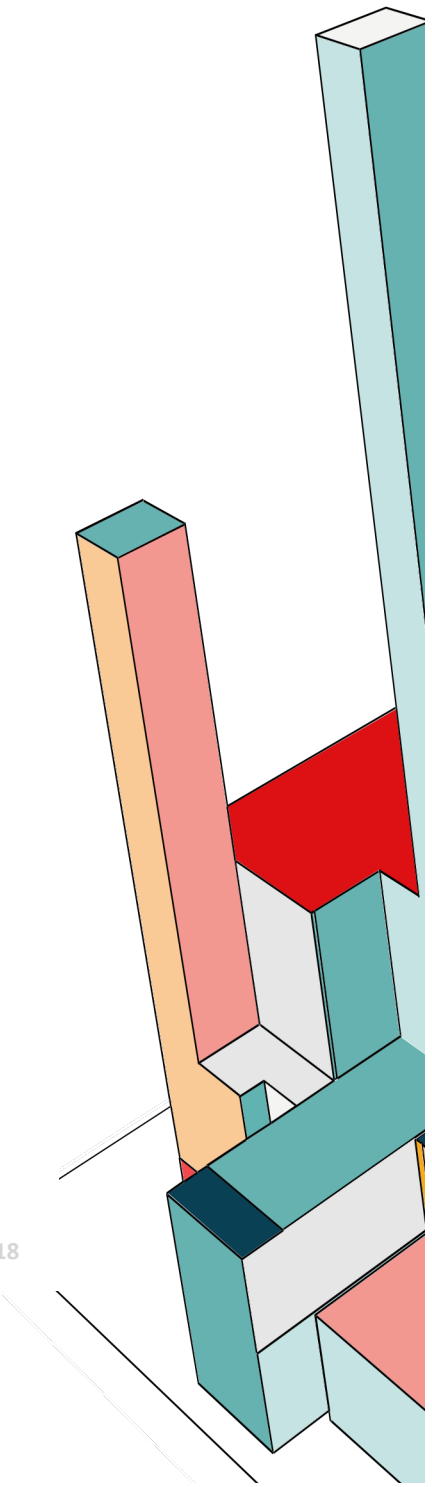


Gradient descent variants

$$\theta_{k+1} = \theta_k - \eta_k \mathrm{g}_k$$

1. Batch gradient descent
2. Stochastic gradient descent
3. Mini-batch gradient descent

Difference: Amount of data used per update

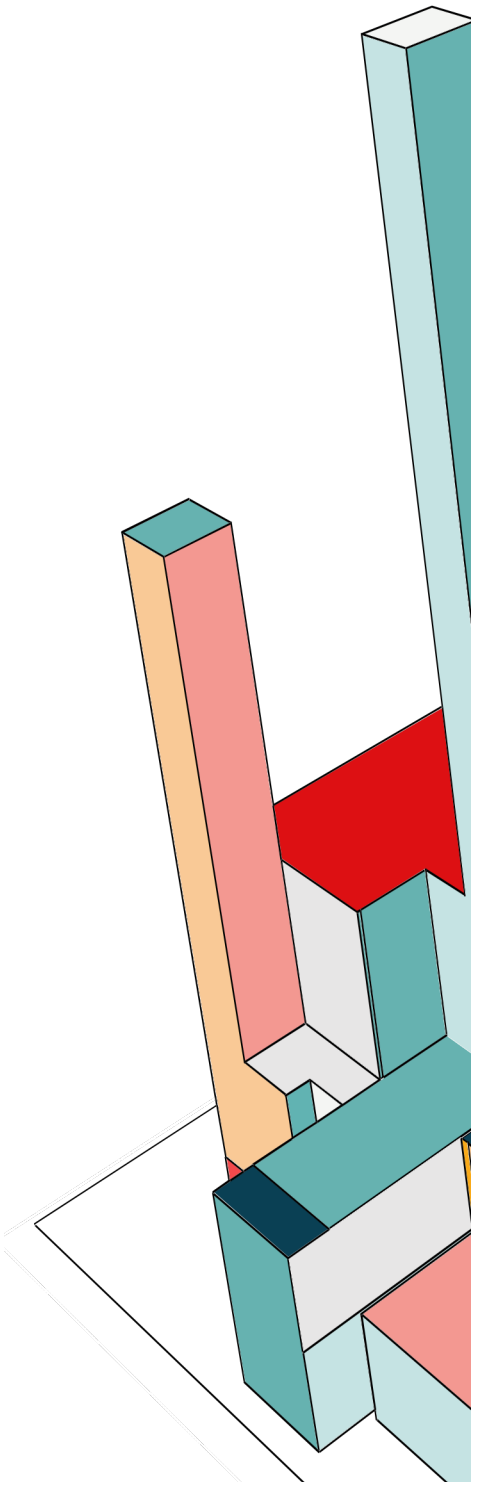


Source: Ruder, 2018

Comparison

Method	Accuracy	Update Speed	Memory Usage	Online Learning
Batch gradient descent	Good	Slow	High	No
Stochastic gradient descent	Good (with annealing)	High	Low	Yes
Mini-batch gradient descent	Good	Medium	Medium	Yes

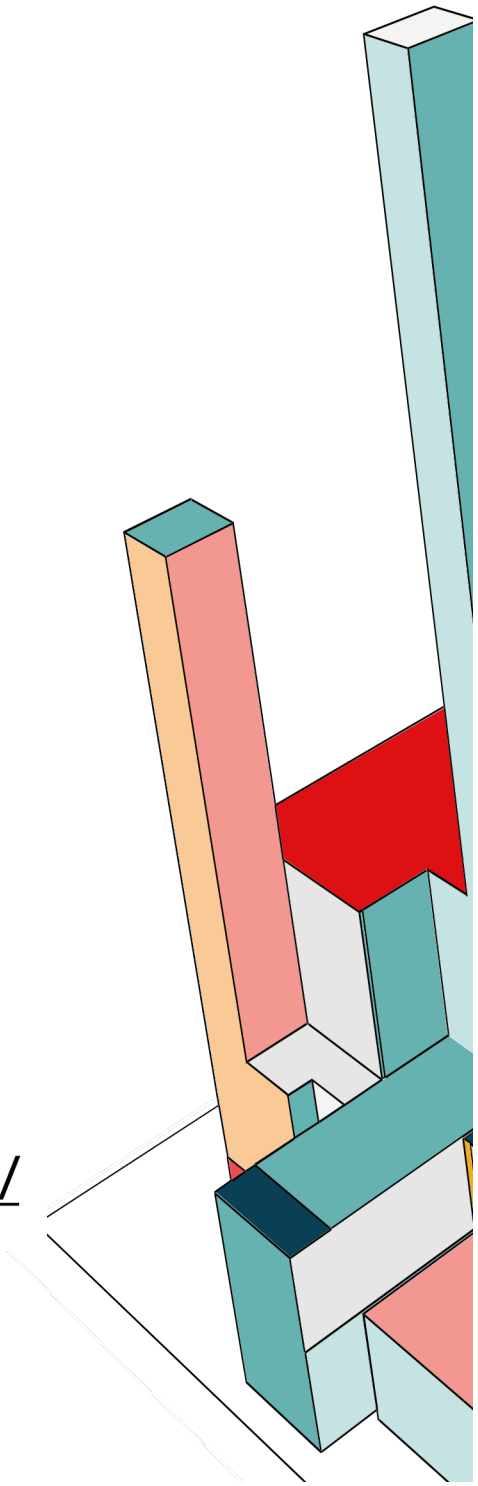
Table: Comparison of trade-offs of gradient descent variants



SGD Variants

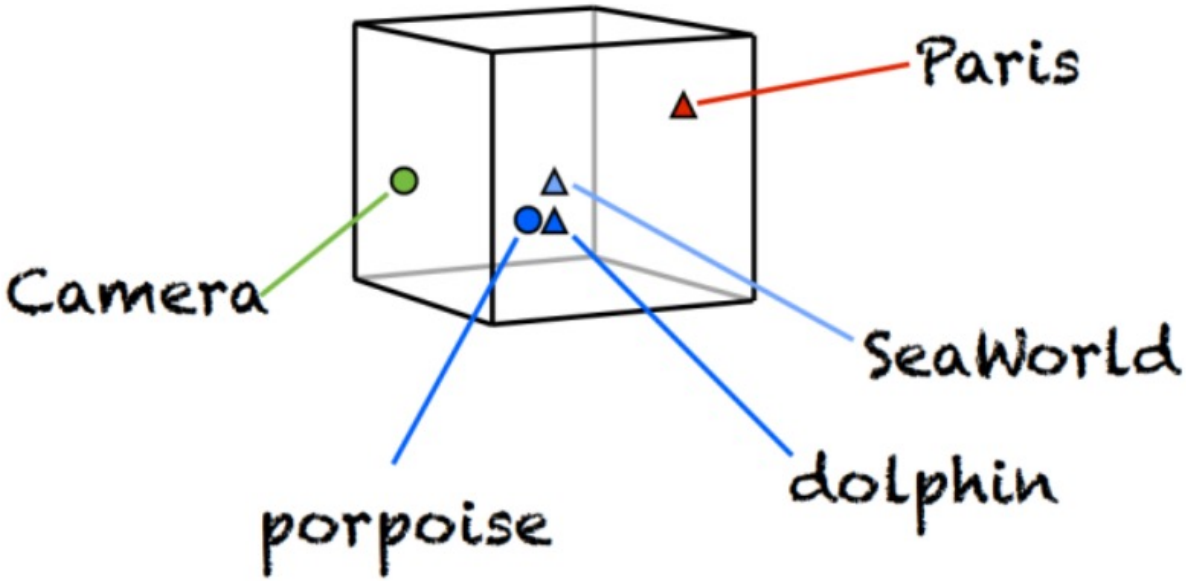
1. Momentum
2. Nesterov accelerated gradient
3. Adagrad
4. Adadelta
5. RMSprop
6. Adam
7. Adam extensions

See https://ruder.io/optimizing-gradient-descent/

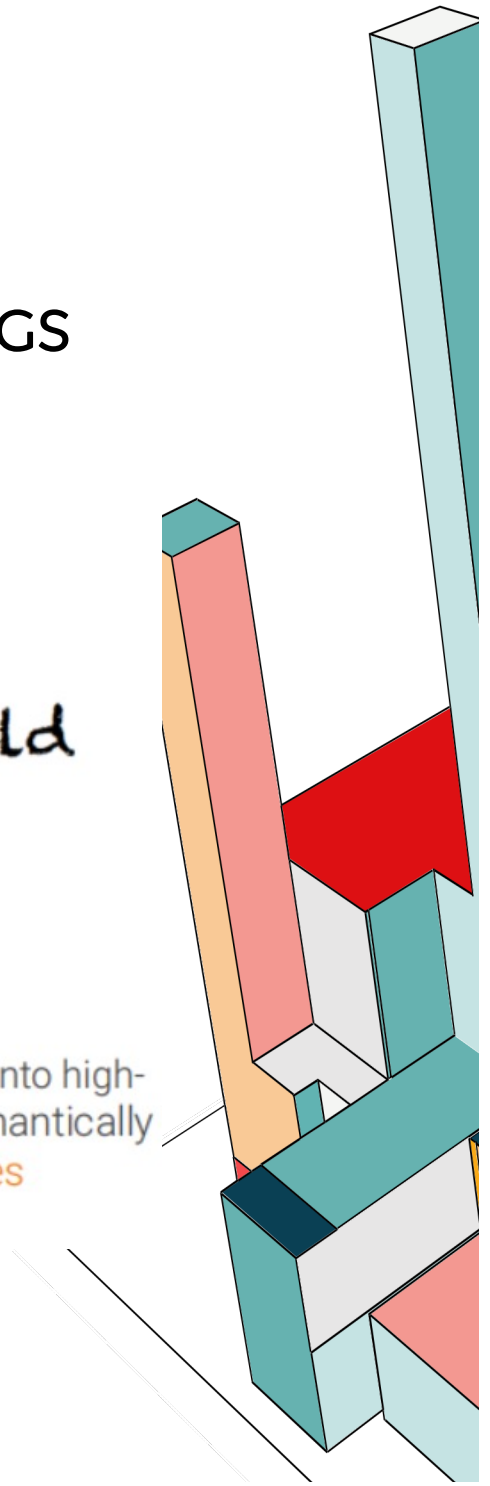


DEEP LEARNING - BASICS

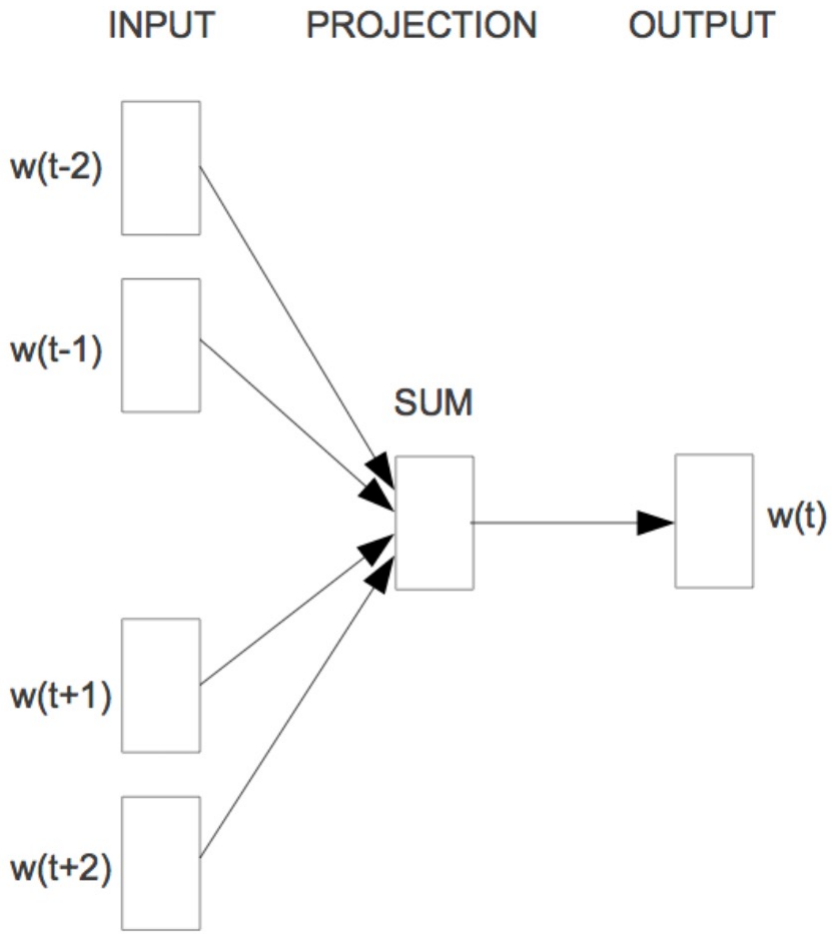
NATURAL LANGUAGE PROCESSING – EMBEDDINGS

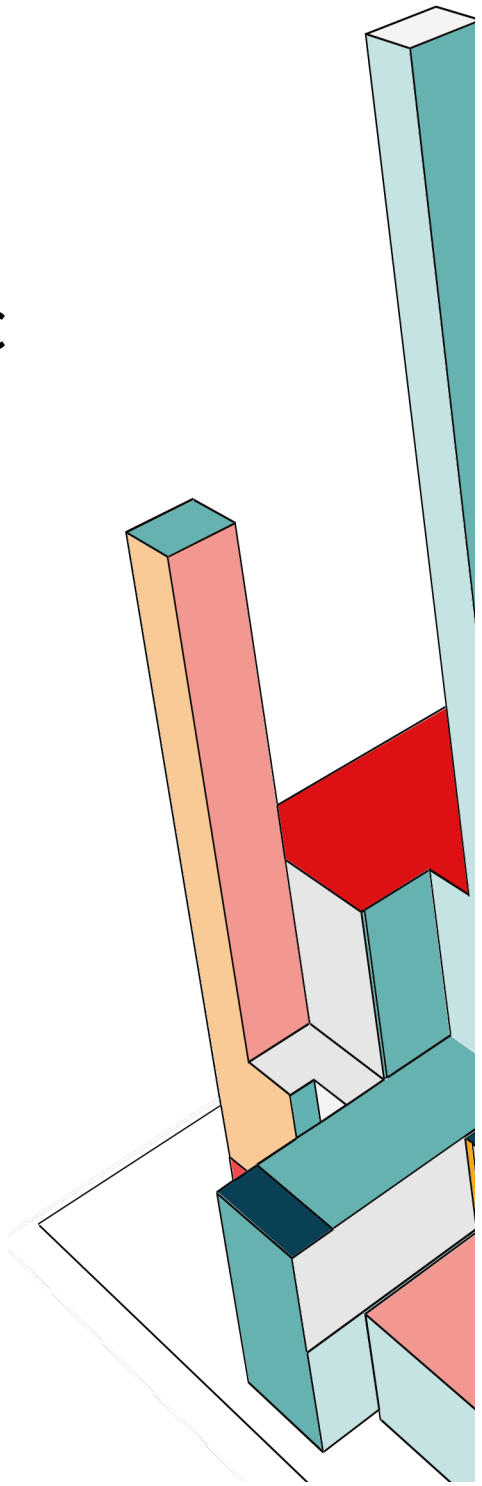


Embeddings are used to turn textual data (words, sentences, paragraphs) into high-dimensional vector representations and group them together with semantically similar data in a vectorspace. Thereby, computer can detect similarities mathematically.



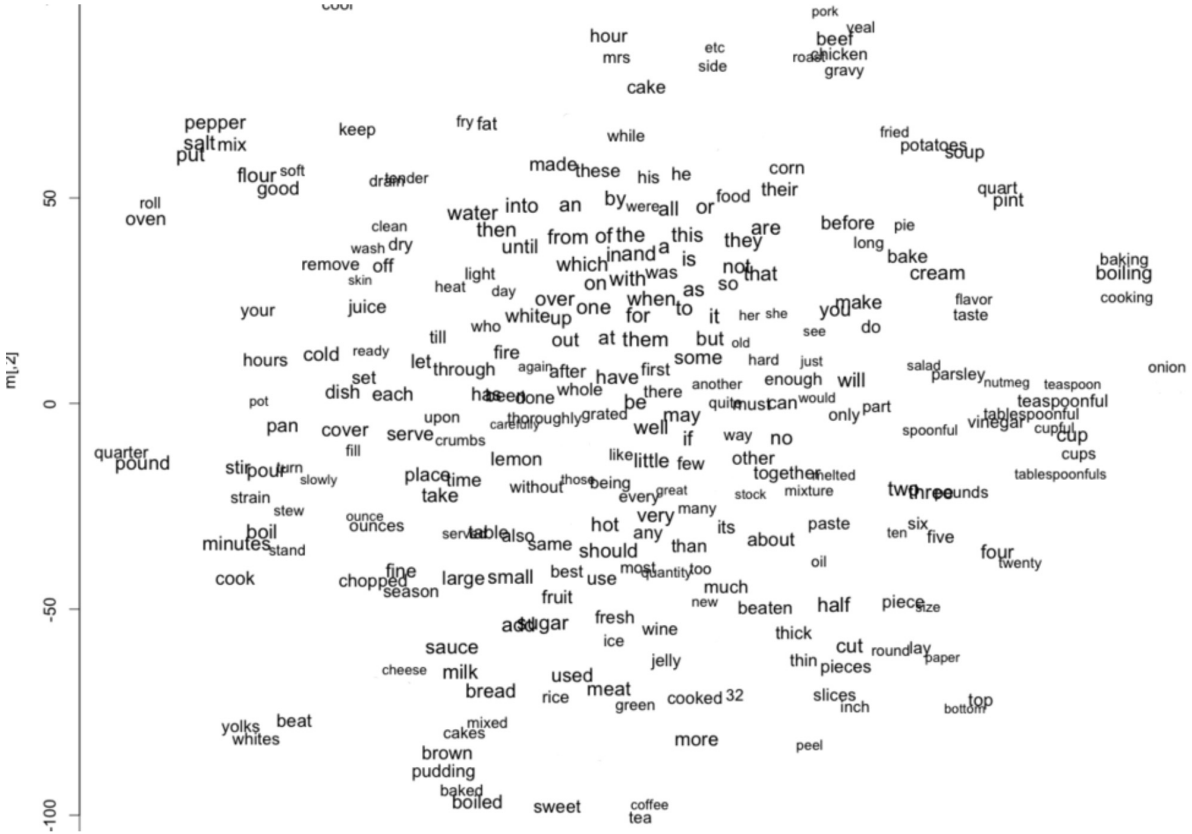
DEEP LEARNING - BASICS
NATURAL LANGUAGE PROCESSING – WORD2VEC

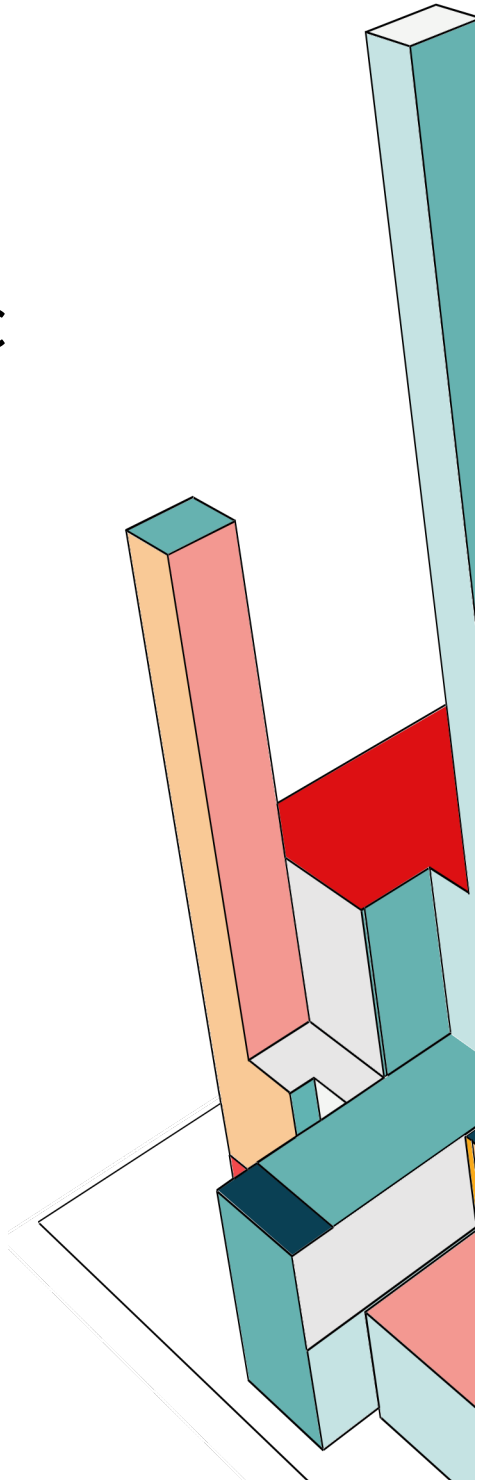




DEEP LEARNING - BASICS

NATURAL LANGUAGE PROCESSING – WORD2VEC

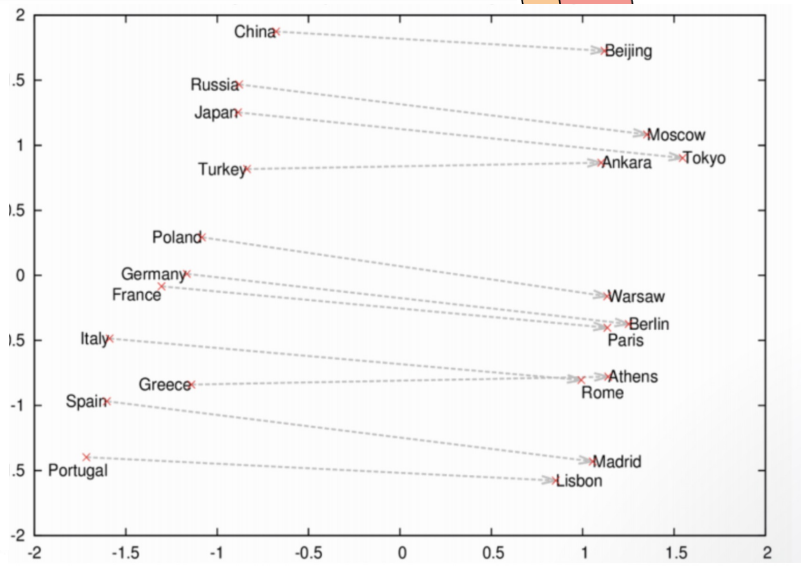




DEEP LEARNING – BASICS
NATURAL LANGUAGE PROCESSING – WORD2VEC

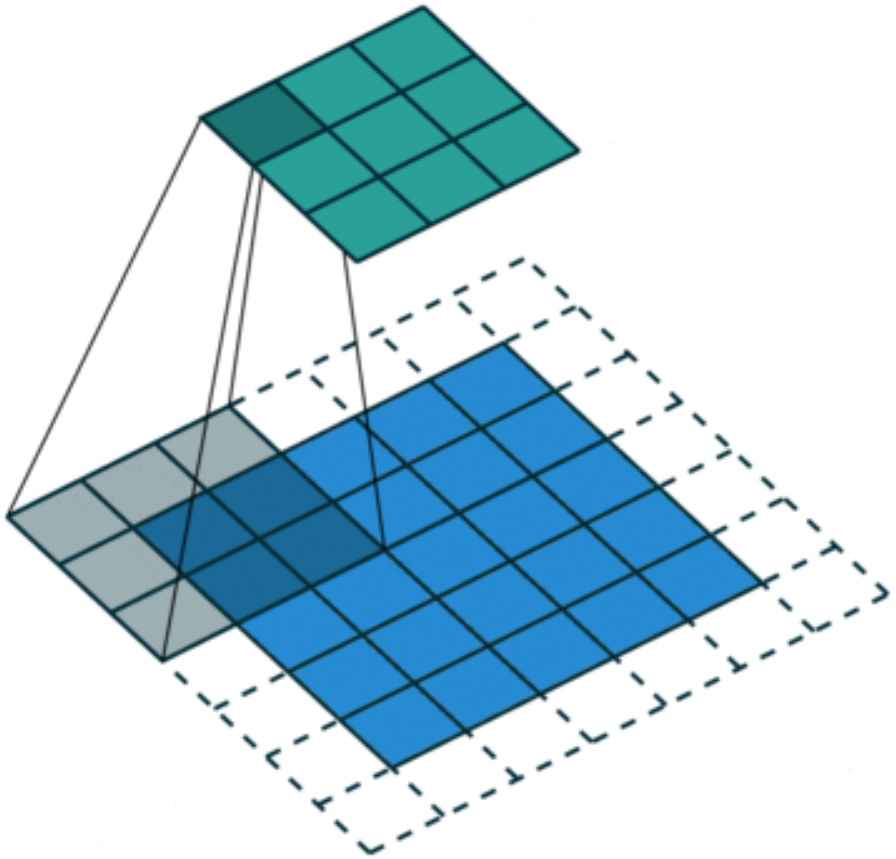
Word2Vec is an unsupervised learning algorithm for obtaining vector representations for words. These vectors were trained for a specific domain on a very large textual data set. **GloVe** is a better performing alternative.

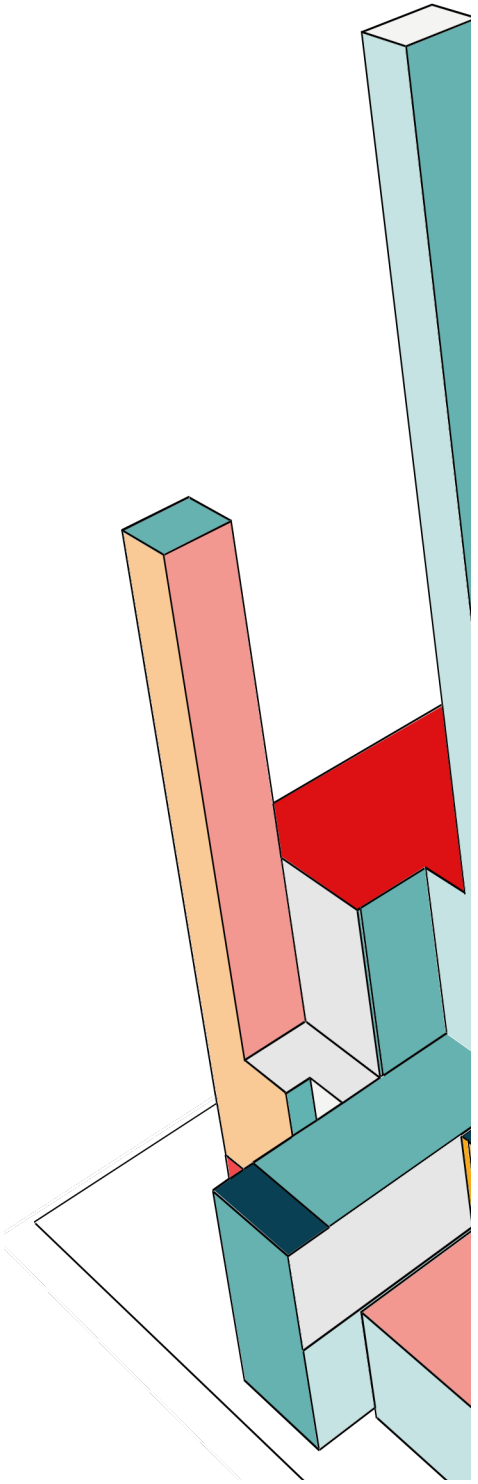
Expression	*Nearest token*
Paris - France + Italy	Rome
bigger - big + cold	colder
sushi - Japan + Germany	bratwurst
Cu - copper + gold	Au
Windows - Microsoft + Google	Android
Montreal Canadiens - Montreal + Toronto	Toronto Maple Leafs



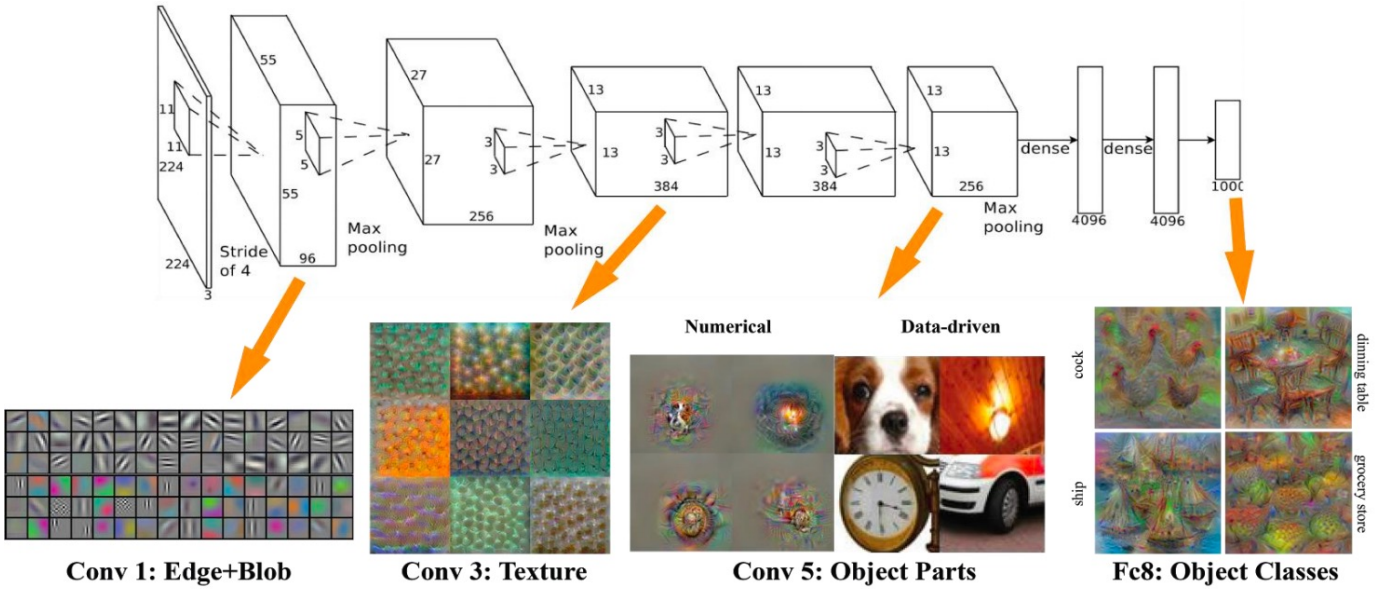
It detects similarities mathematically by grouping the vectors of similar words together.
All it needs is words co-occurance in the given corpus.

DEEP LEARNING - BASICS
CONVOLUTIONAL NEURAL NETS (CNN)



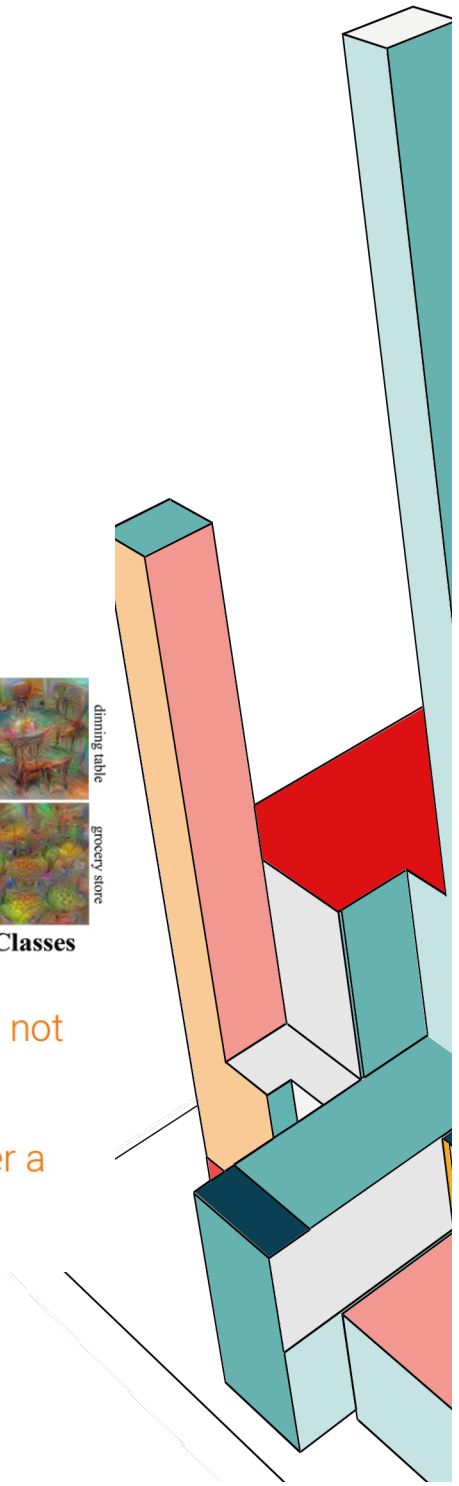


DEEP LEARNING - BASICS
CONVOLUTIONAL NEURAL NETS (CNN)

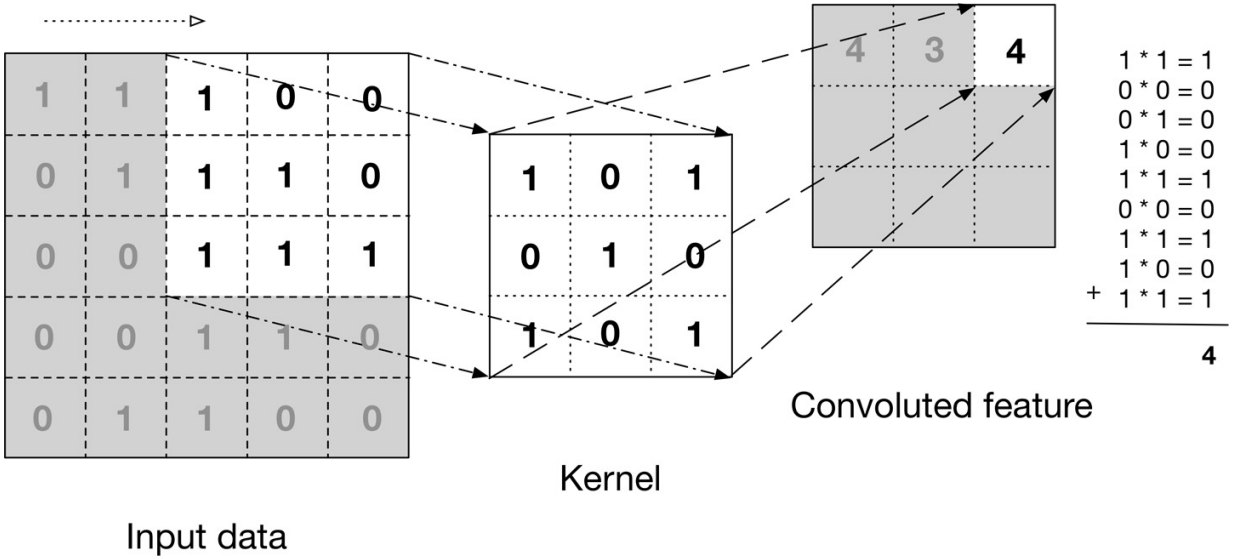


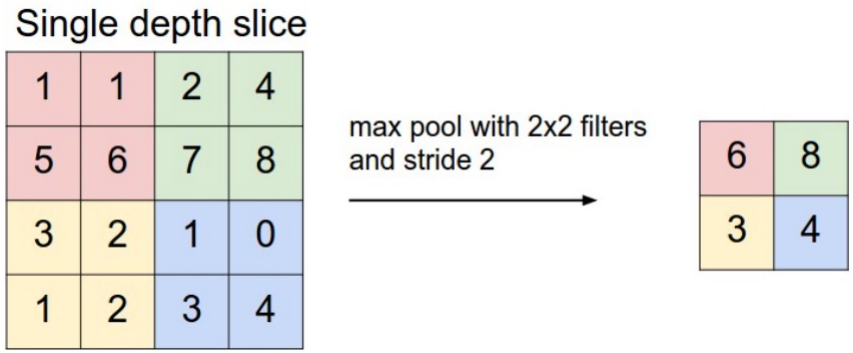
Convolution layer is a feature detector that automagically learns to filter out not needed information from an input by using convolution kernel.

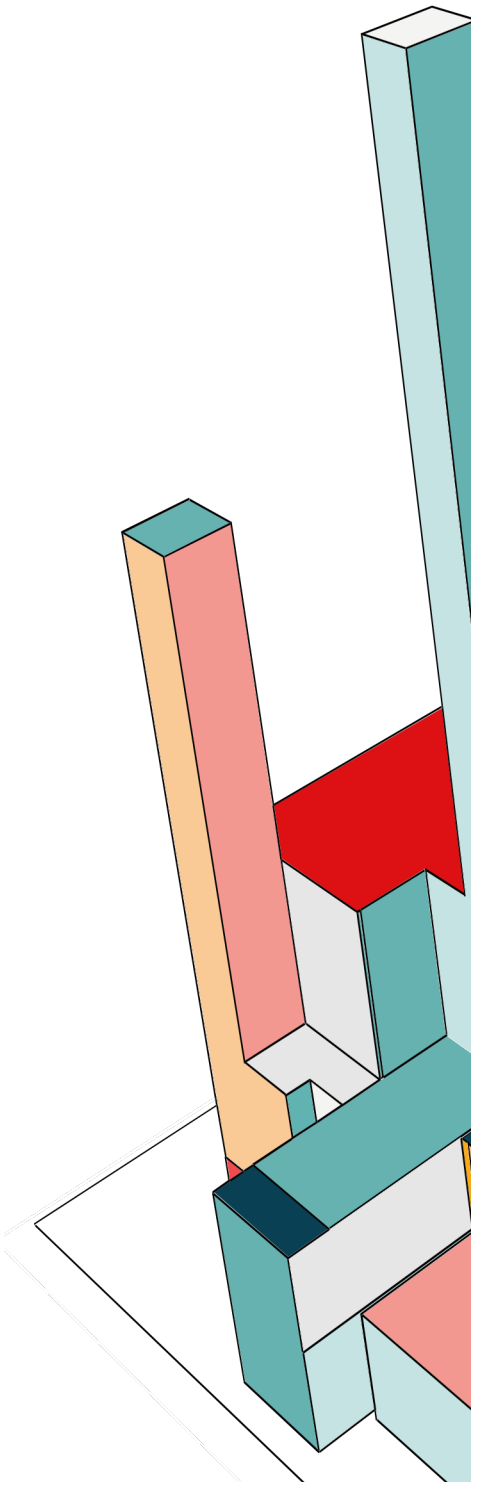
Pooling layers compute the max or average value of a particular feature over a region of the input data *(downsizing of input images)*. Also helps to detect objects in some unusual places and reduces memory size.



DEEP LEARNING - BASICS
CONVOLUTIONAL NEURAL NETS (CNN)

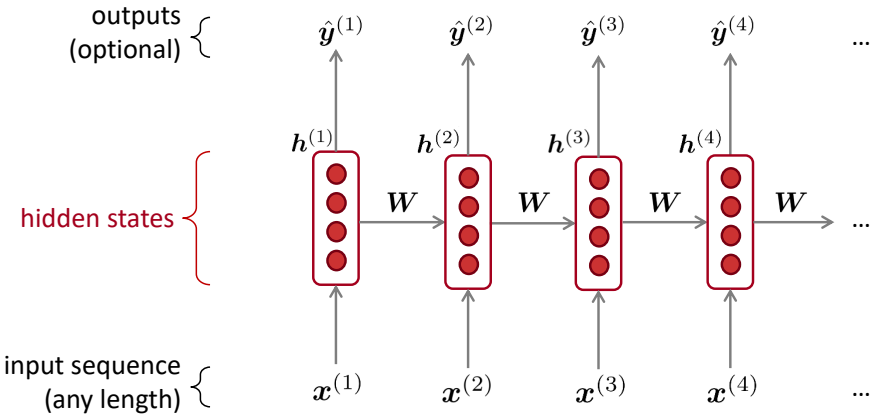






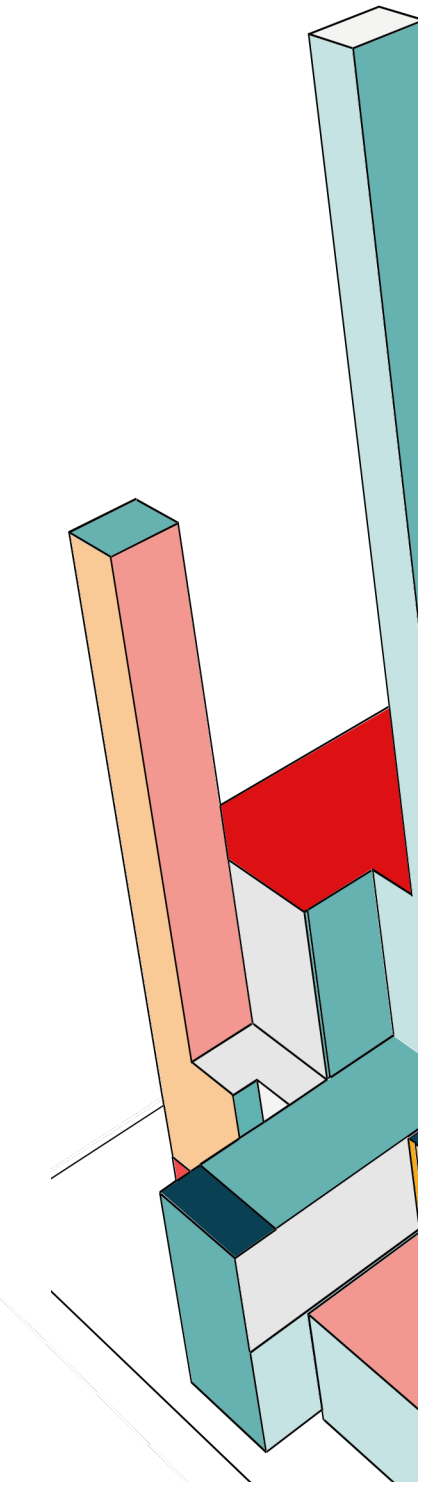
Recurrent Neural Networks

Main idea: Apply the same weights *repeatedly (recurrently)*



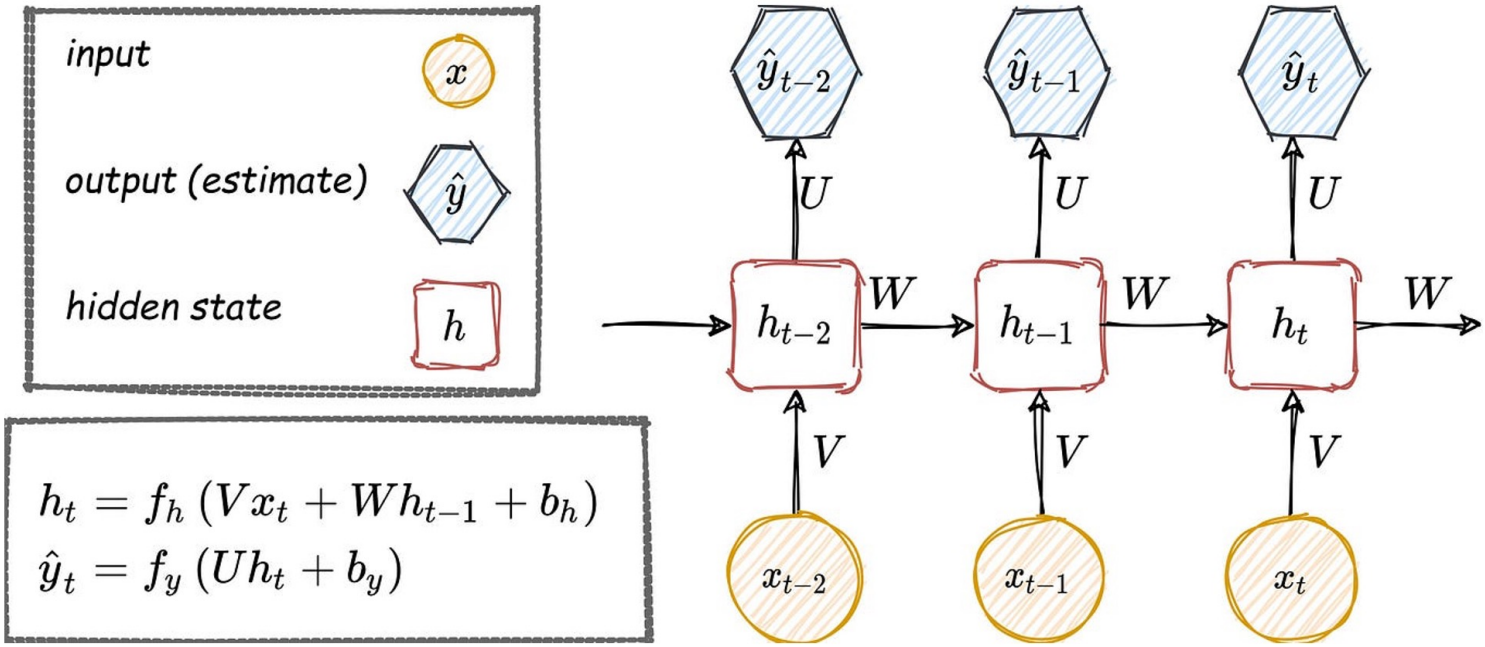
Notice:

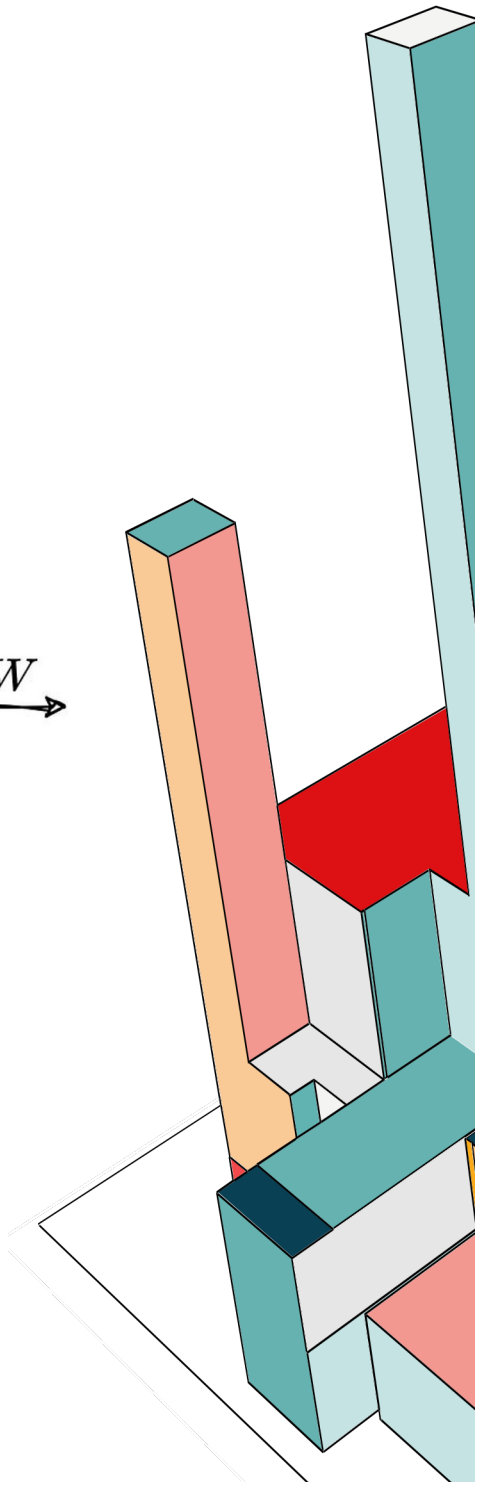
- Output at each step is optional
- Recurrence is done with hidden states



Source: Stanford 224n

Vanilla RNN


input
output (estimate)
hidden state
x
$\hat{y}$
h
$\hat{y}_{t-2}$
$\hat{y}_{t-1}$
$\hat{y}_t$
U
W
V
h_{t-2}
h_{t-1}
h_t
x_{t-2}
x_{t-1}
x_t
$h_t = f_h\left(Vx_t + Wh_{t-1} + b_h\right)$
$\hat{y}_t = f_y\left(Uh_t + b_y\right)$



Backward Propagation Through Time

Loss Function $\mathcal{L}$

output (estimate) $\hat{y}$

true label y

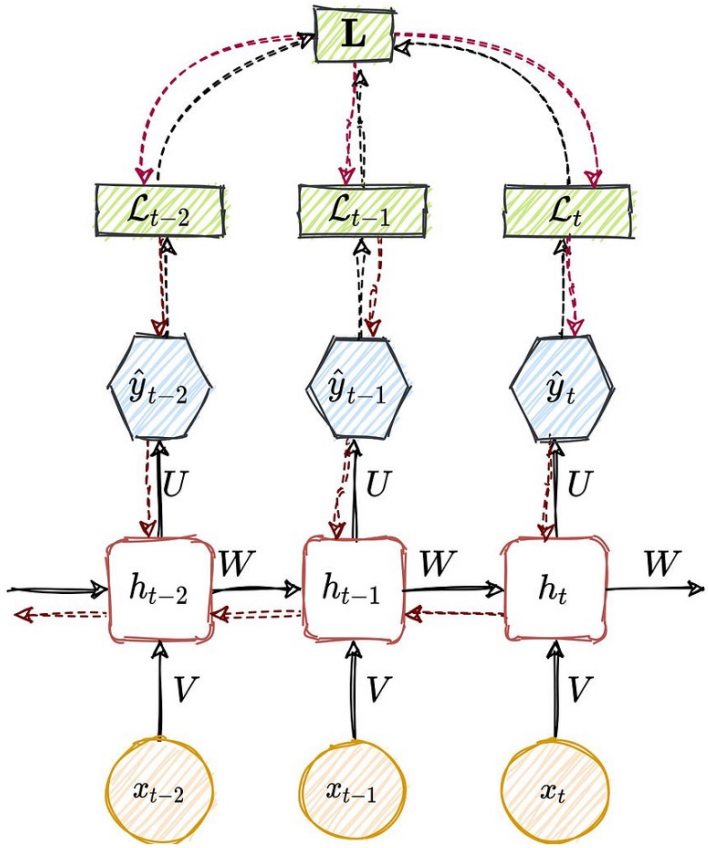
$$\mathbf{L} = \sum_i \mathcal{L}_i \left(\hat{y}_t, y_t\right)$$

Forward Pass:

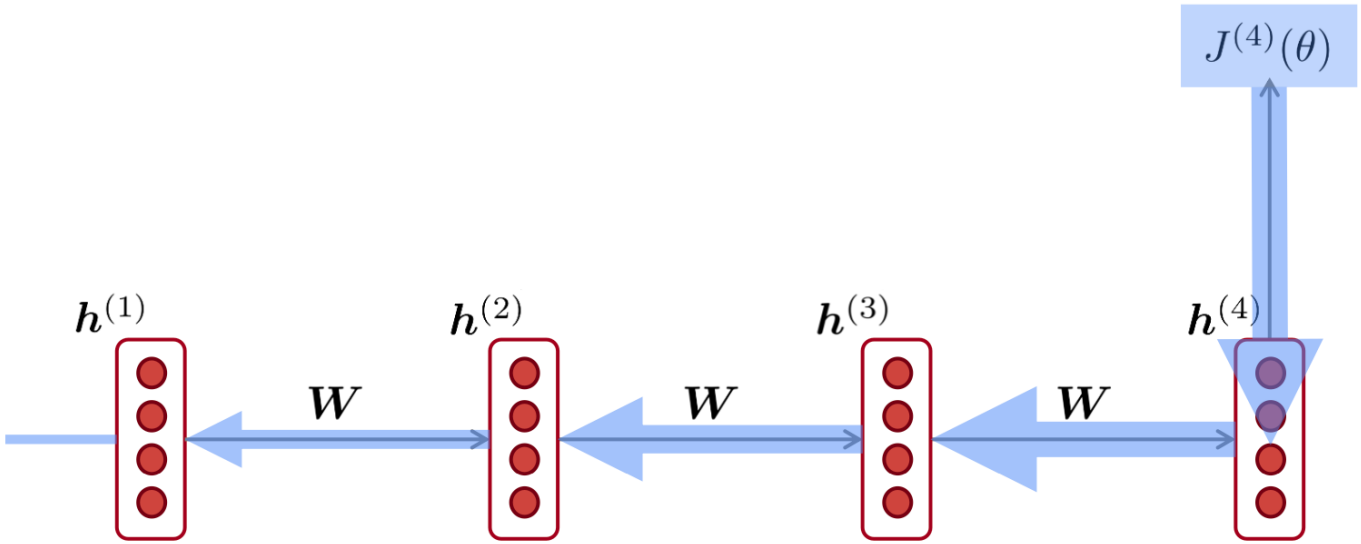
$$h_t, \hat{y}_t, \mathcal{L}_t, \mathbf{L}$$

Backward Pass:

$$\frac{\partial \mathbf{L}}{\partial U}, \frac{\partial \mathbf{L}}{\partial V}, \frac{\partial \mathbf{L}}{\partial W}, \frac{\partial \mathbf{L}}{\partial b_h}, \frac{\partial \mathbf{L}}{\partial b_y}$$



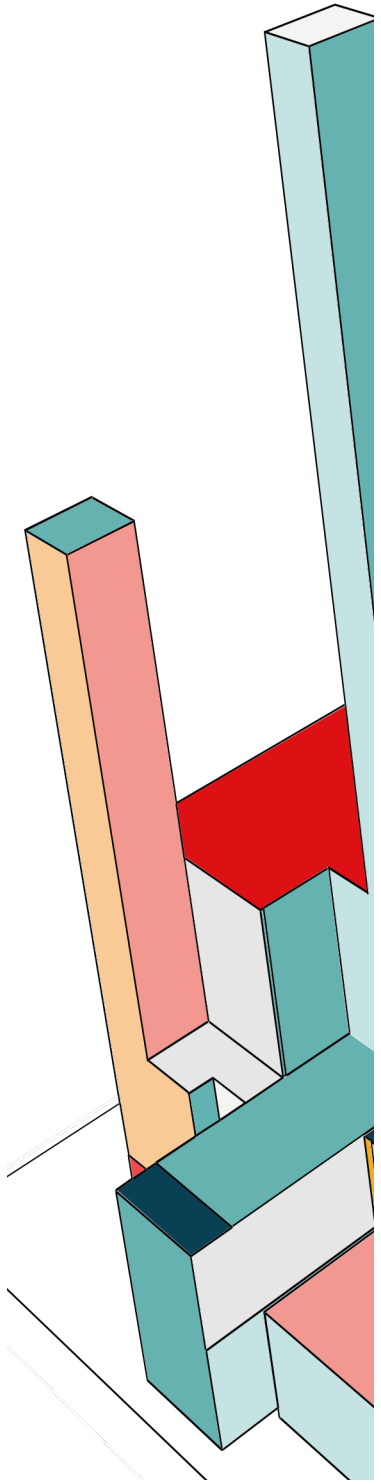
Gradient Vanishing/Explosion Problem



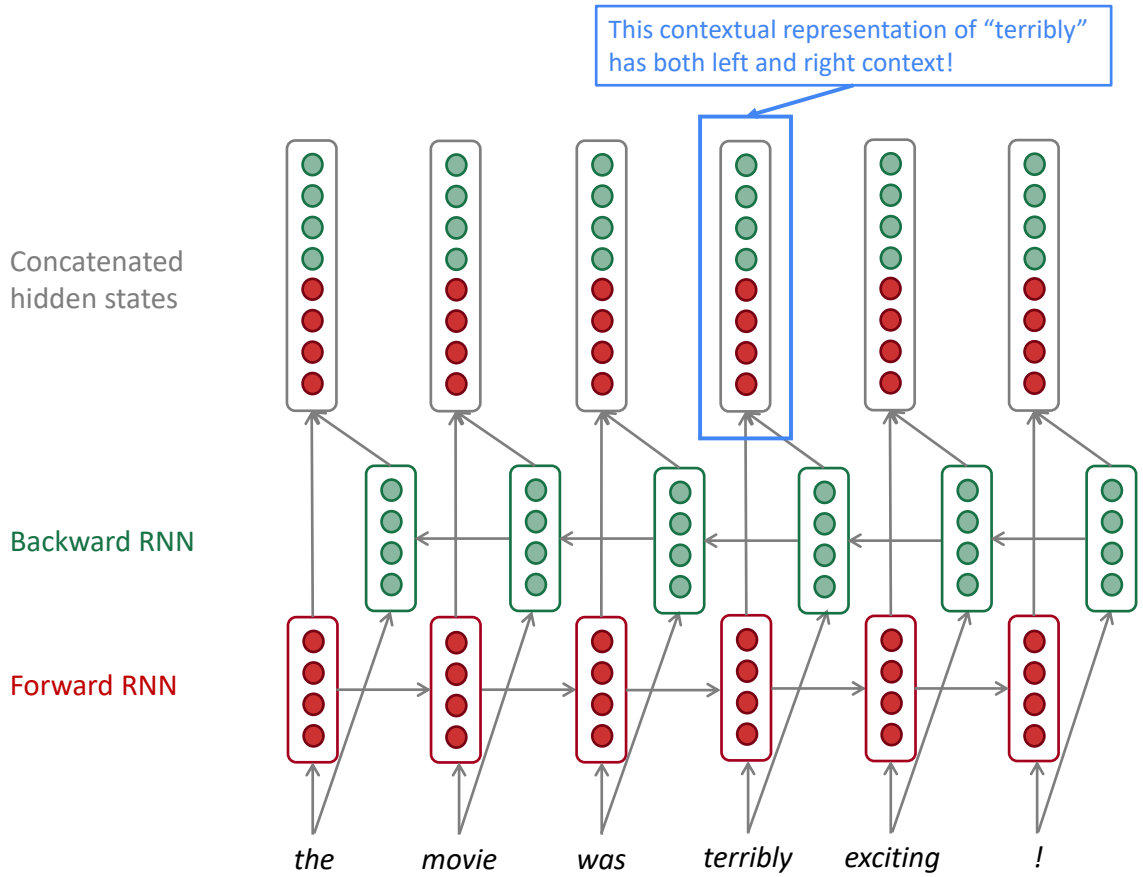
$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$$

What happens if these are small?

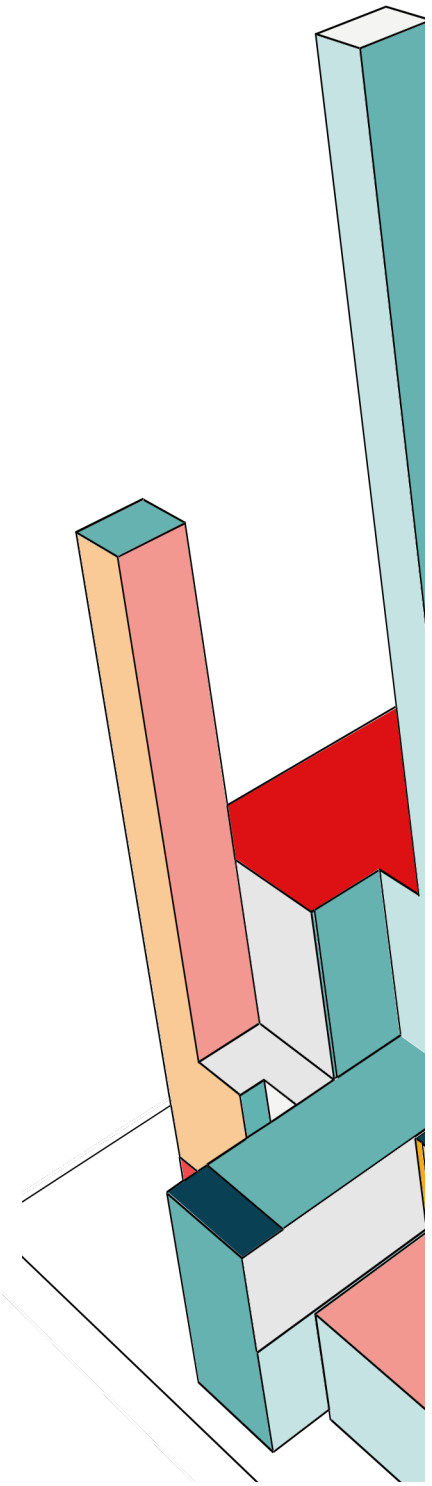
Vanishing gradient problem:
When these are small, the gradient signal gets smaller and smaller as it backpropagates further



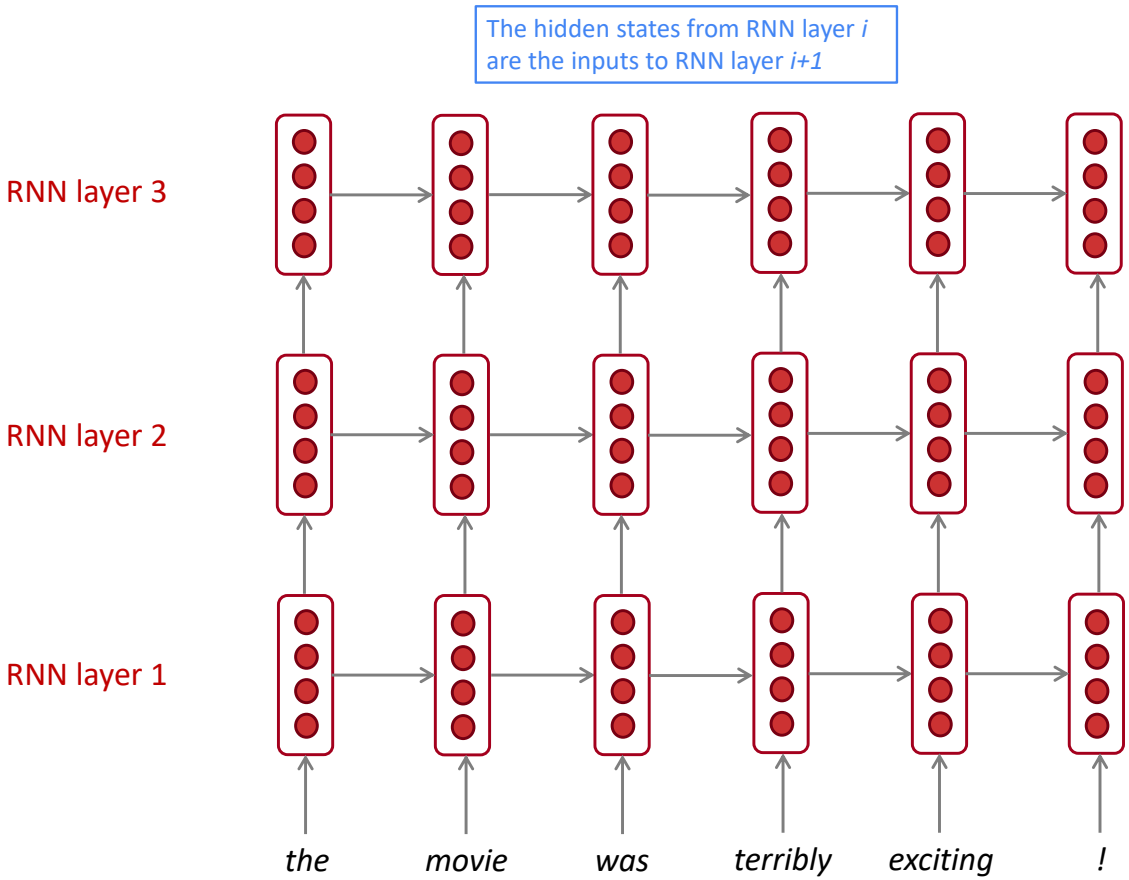
Bi-directional RNN



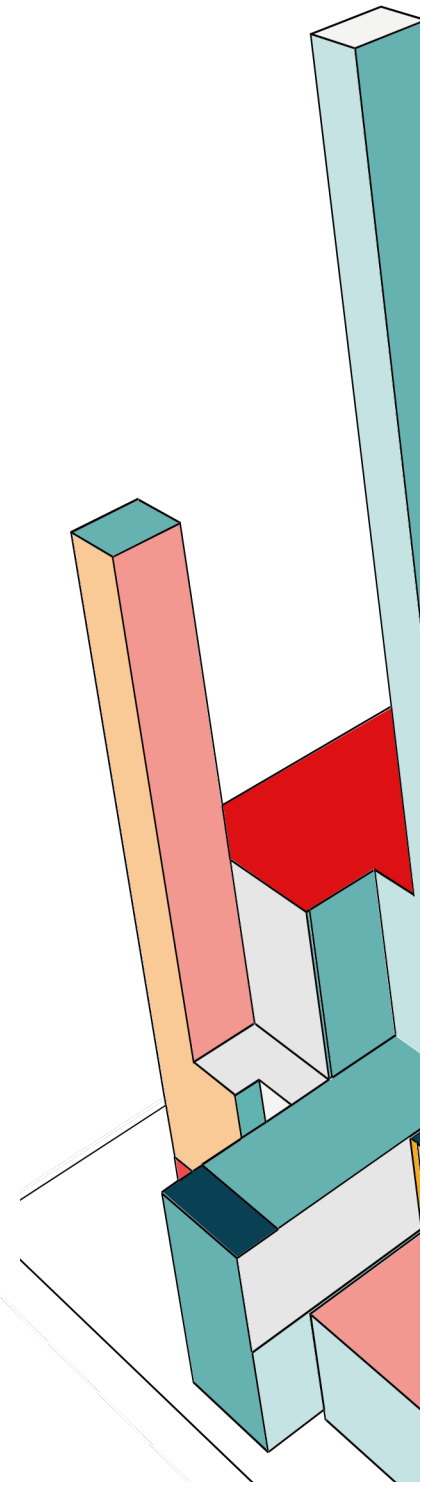
Source: Stanford 224n



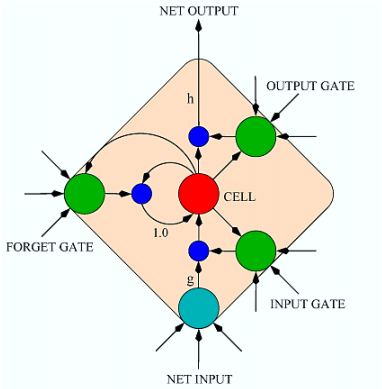
Stacked RNNs



Source: Stanford 224n



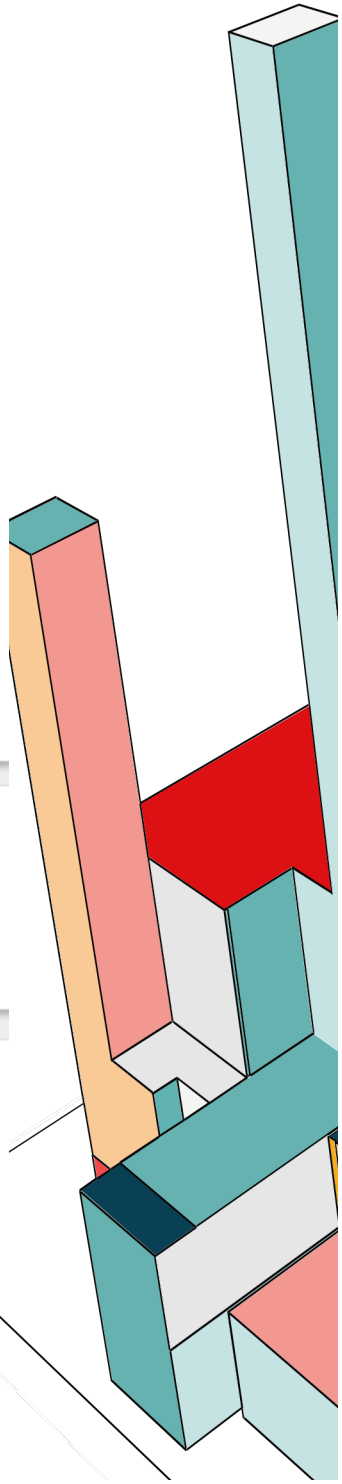
DEEP LEARNING - BASICS
LONG SHORT-TERM MEMORY RNN (LSTM)



A Long Short-Term Memory (LSTM) network is a particular type of recurrent network that works slightly better in practice, owing to its more powerful update equation and some appealing back propagation dynamics.

The **LSTM units** give the network memory cells with read, write and reset operations. During training, the network can learn when it should remember data and when it should throw it away.

Well-suited to learn from experience to classify, process and predict time series when there are very long time lags of unknown size between important events.



LSTM GATES

- Given current input and previous hidden state, we compute the gates

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

Sigmoid function: all gate values are between 0 and 1

$$\boldsymbol{f}^{(t)} = \sigma\left(\boldsymbol{W}_f \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_f \boldsymbol{x}^{(t)} + \boldsymbol{b}_f\right)$$
$$\boldsymbol{i}^{(t)} = \sigma\left(\boldsymbol{W}_i \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_i \boldsymbol{x}^{(t)} + \boldsymbol{b}_i\right)$$
$$\boldsymbol{o}^{(t)} = \sigma\left(\boldsymbol{W}_o \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_o \boldsymbol{x}^{(t)} + \boldsymbol{b}_o\right)$$

- Now we compute how much to forget, update and output

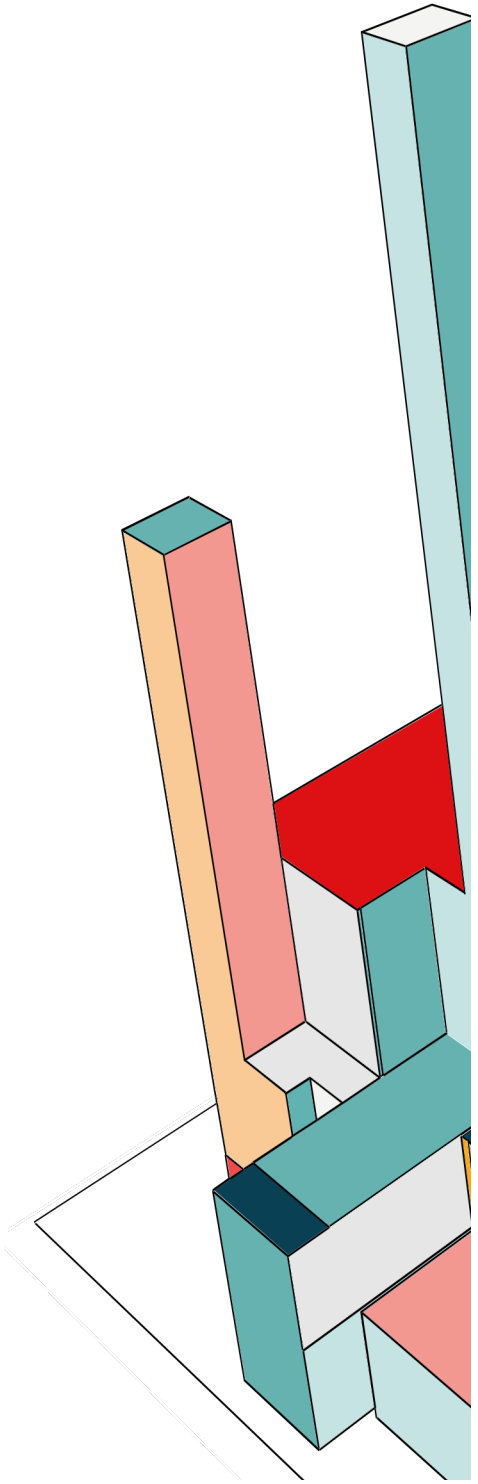
New cell content: this is the new content to be written to the cell

Cell state: erase ("forget") some content from last cell state, and write ("input") some new cell content

Hidden state: read ("output") some content from the cell

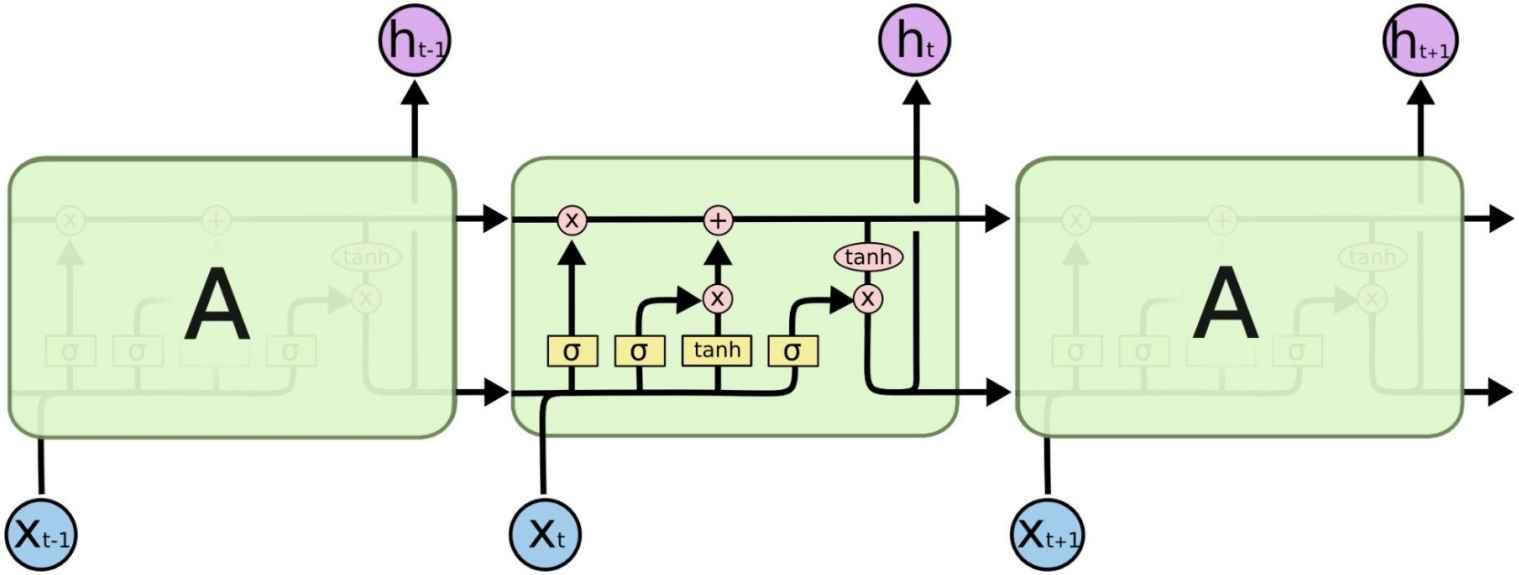
$$\tilde{\boldsymbol{c}}^{(t)} = \tanh\left(\boldsymbol{W}_c \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_c \boldsymbol{x}^{(t)} + \boldsymbol{b}_c\right)$$
$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \tilde{\boldsymbol{c}}^{(t)}$$
$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \circ \tanh \boldsymbol{c}^{(t)}$$

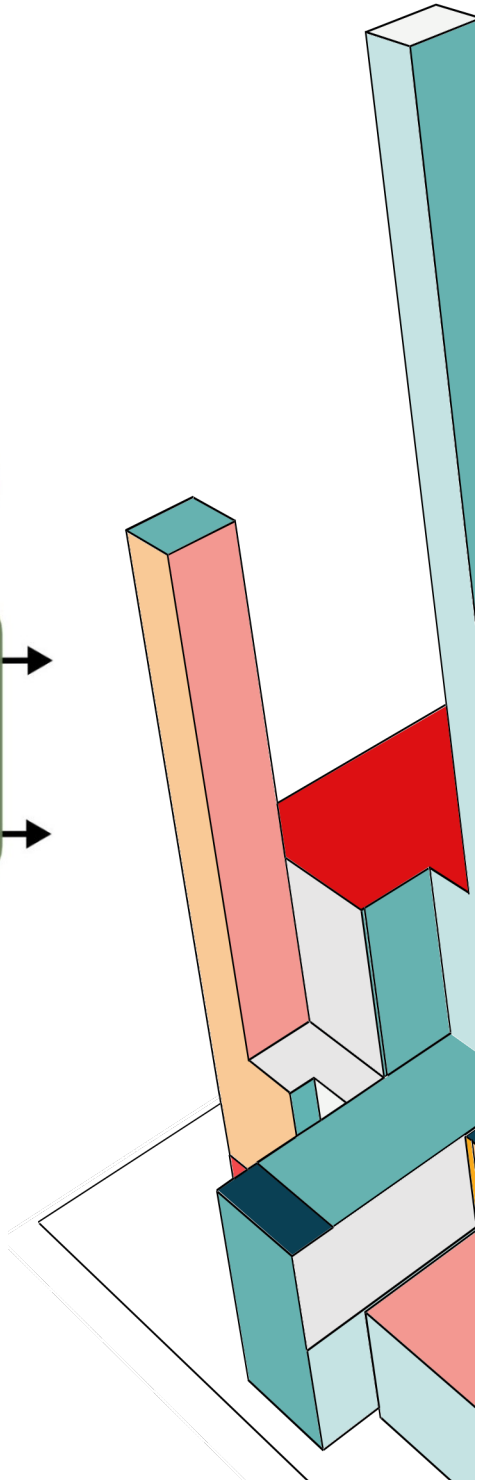
Gates are applied using element-wise product



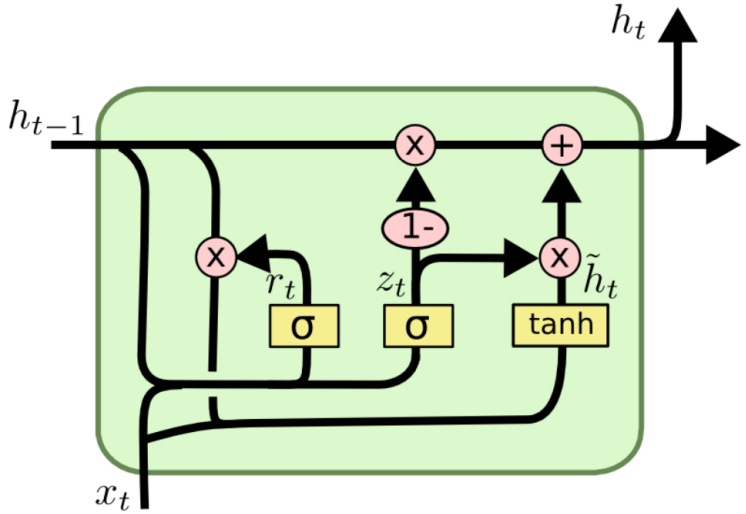
Source: Stanford 224n

ARCHITECTURE





Gated Recurrent Unit (GRU)

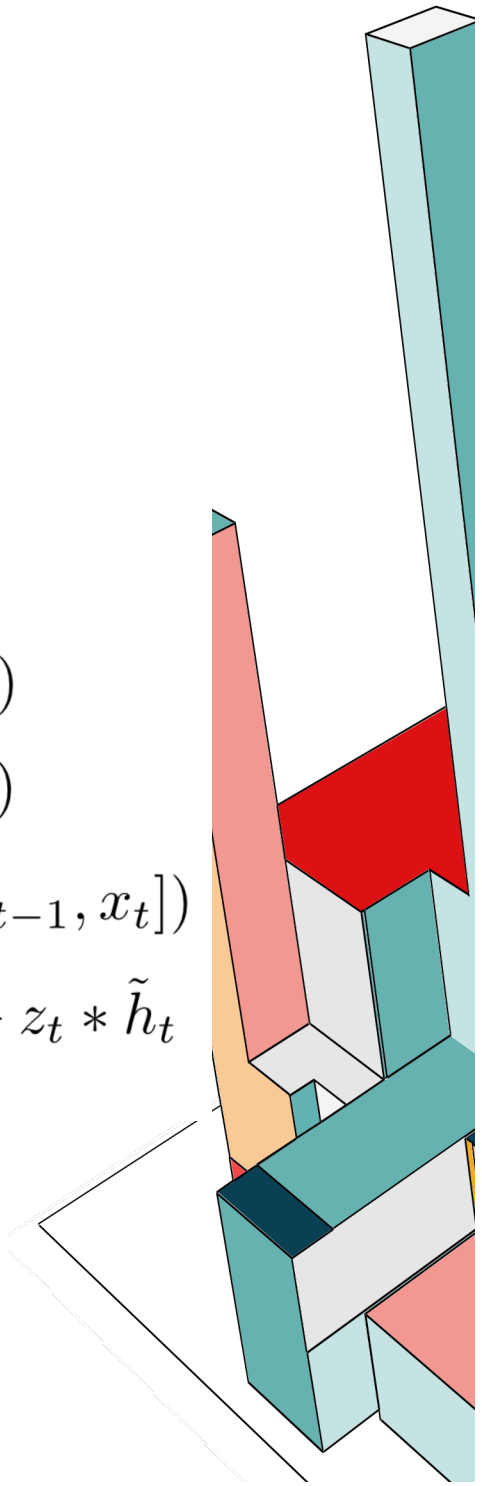


$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

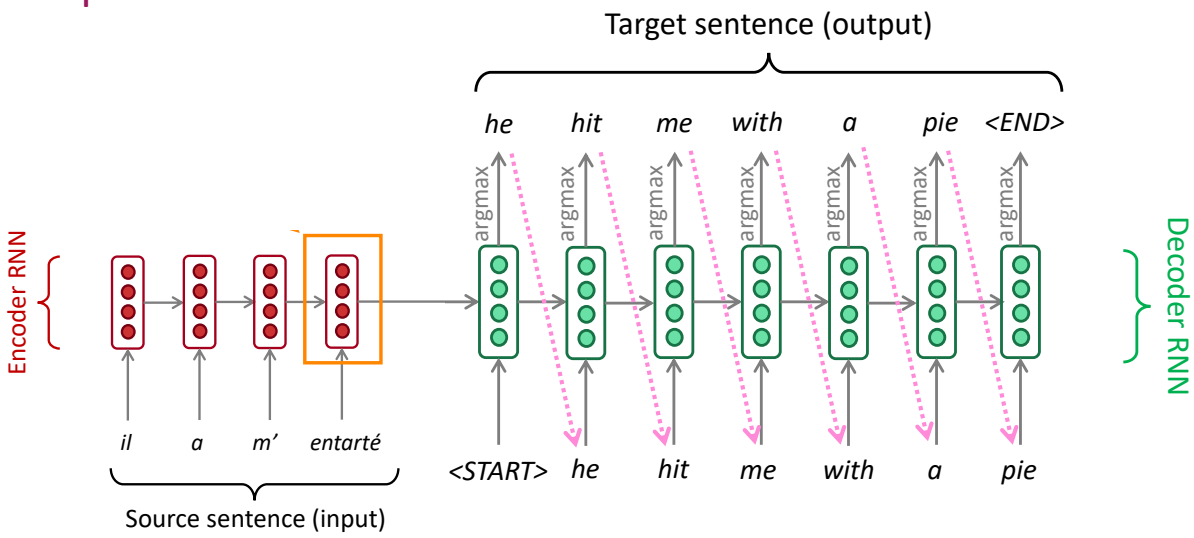
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

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



Sequence-to-sequence Model

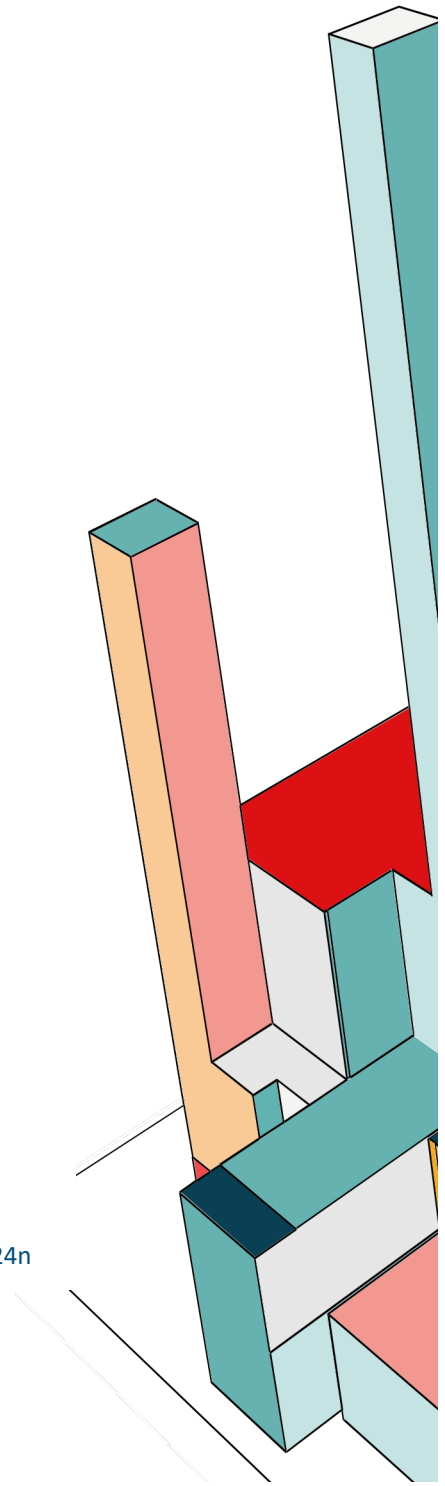
- Seq2Seq with Two RNNs



Encoder produces an encoding of the source sentence.

Provides Initial hidden state for Decoder

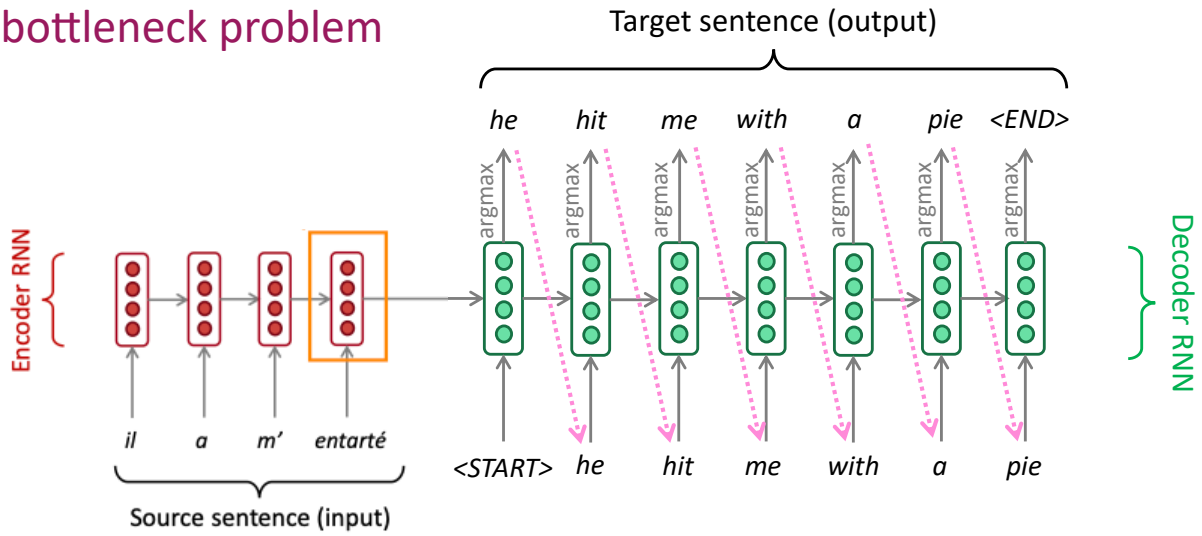
Decoder RNN is a **Conditional Language Model** that generates target sentence, *conditioned on encoding*.



Source: Stanford 224n

Sequence-to-sequence Model

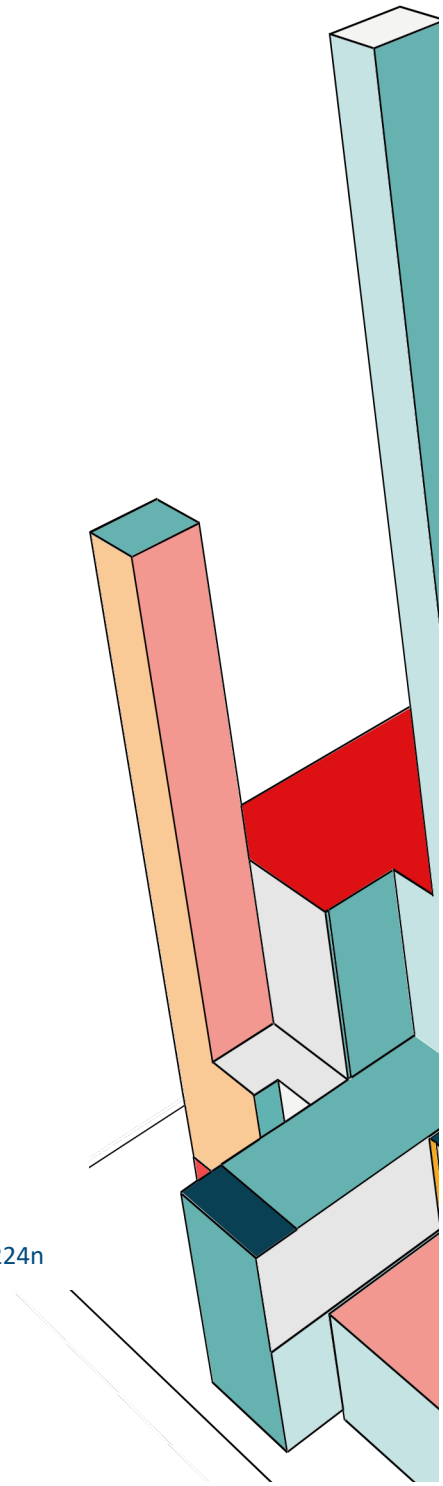
- The bottleneck problem



Encoding of the source sentence. This needs to capture *all information* about the source sentence. Information bottleneck!

X

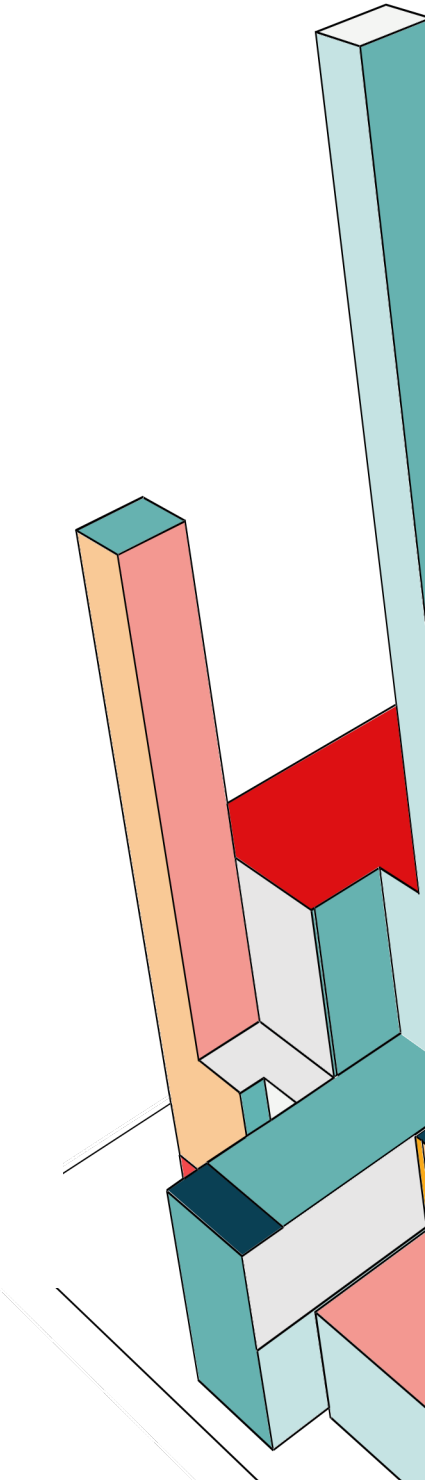
Source: Stanford 224n



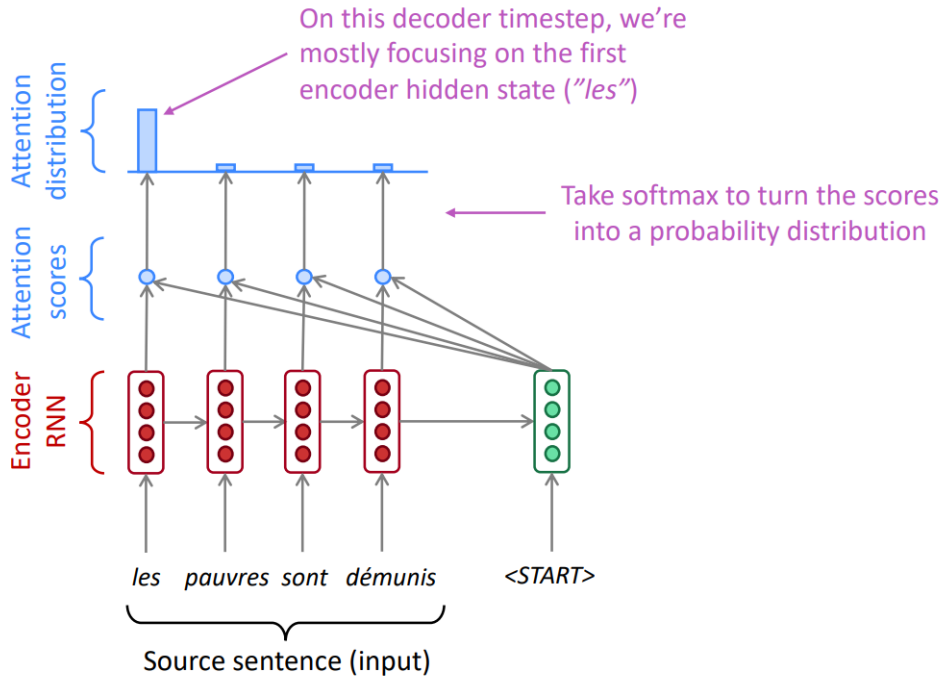
Attention Mechanism

- **Attention** provides a solution to the bottleneck problem.
- **Core idea:** on each step of the decoder, use *direct connection to the encoder* to *focus on the relevant part* of the source sequence

X

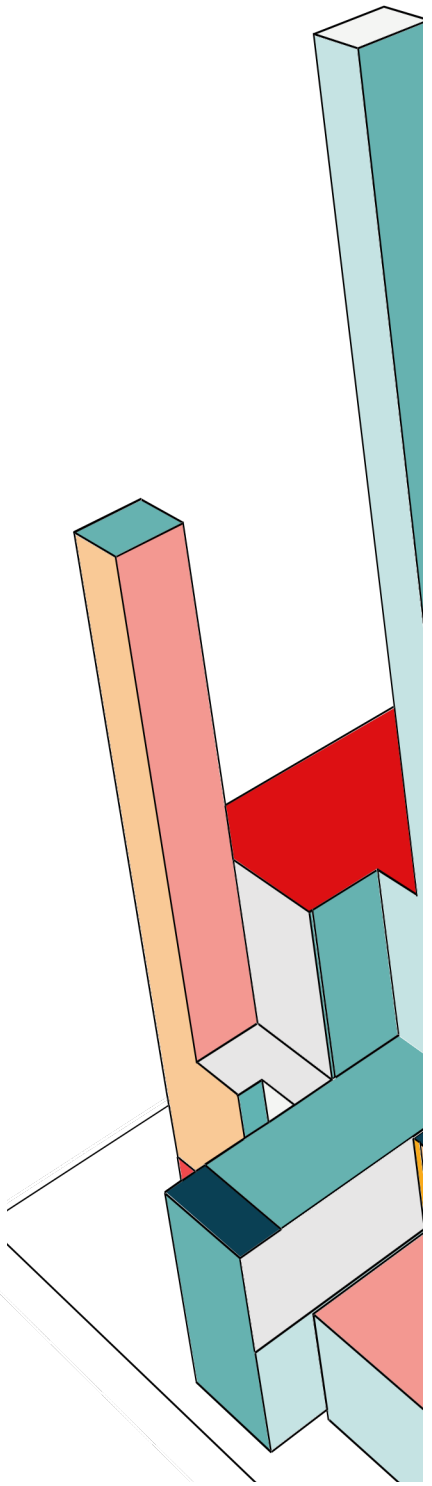


Attention Mechanism

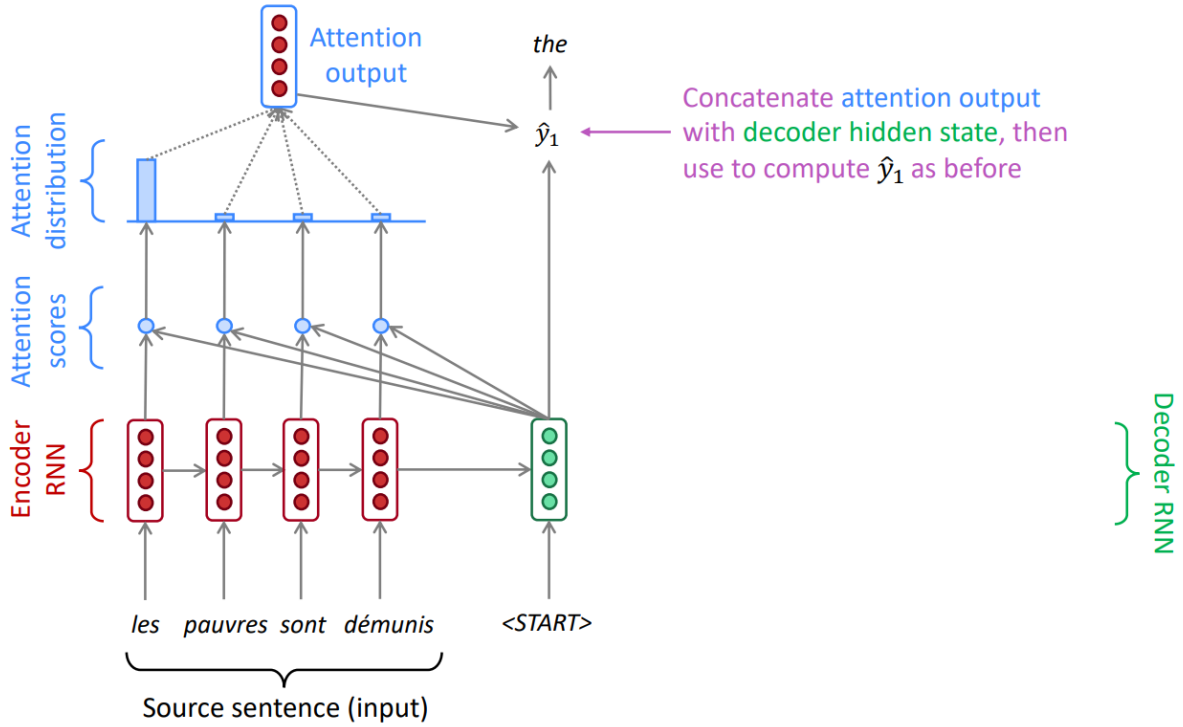


Decoder RNN

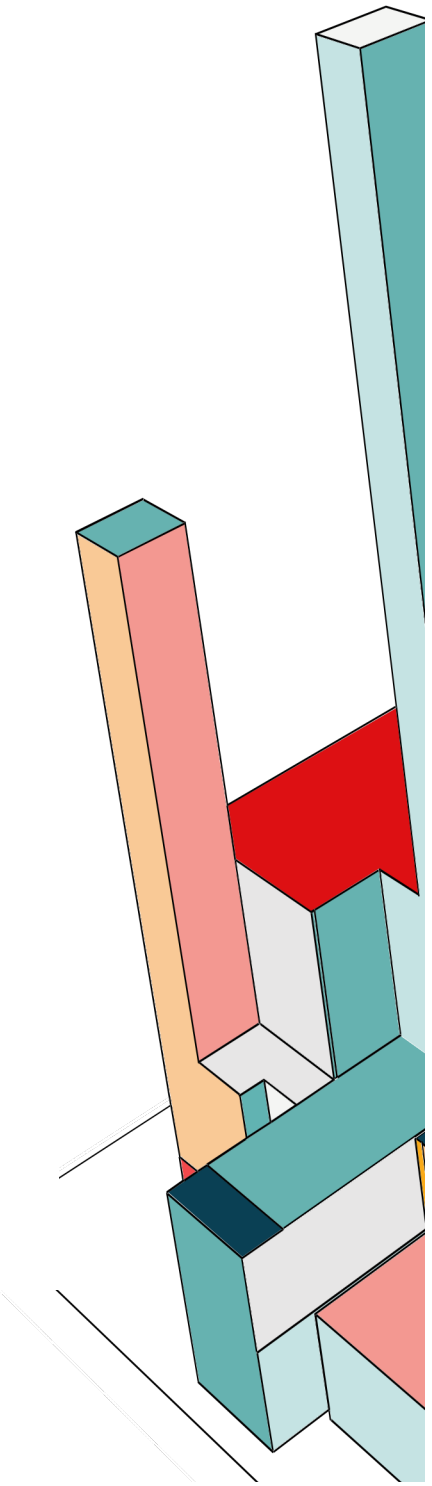
X



Attention Mechanism



X



Attention Mechanism (Formally)

- We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$
- On timestep *t*, we have decoder hidden state $s_t \in \mathbb{R}^h$
- We get the attention scores $\boldsymbol{e}^t$ for this step:

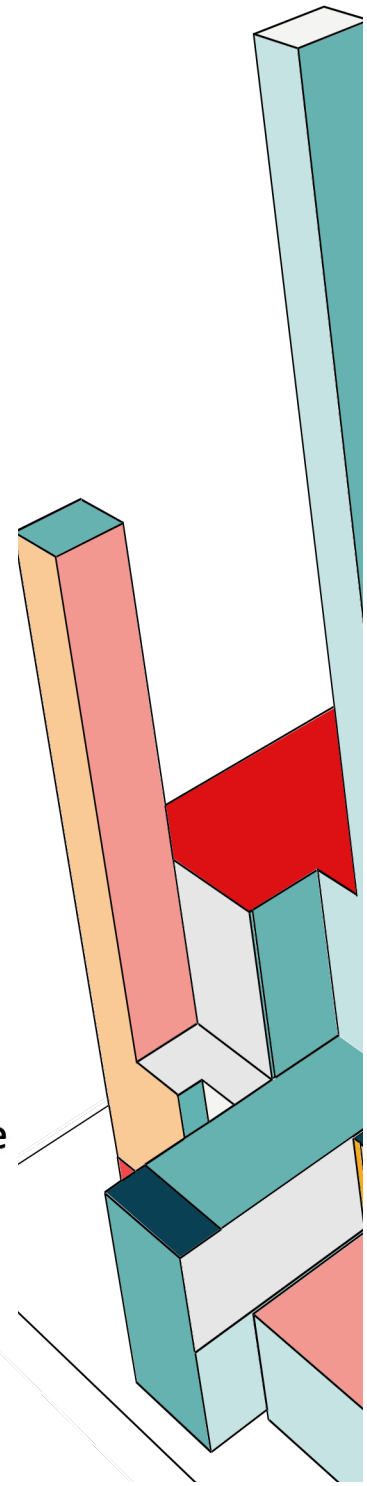
$$\boldsymbol{e}^t = [\boldsymbol{s}_t^T \boldsymbol{h}_1, \ldots, \boldsymbol{s}_t^T \boldsymbol{h}_N] \in \mathbb{R}^N$$

- We take softmax to get the attention distribution α^t for this step (this is a probability distribution and sums to 1)

$$\alpha^t = \text{softmax}(\boldsymbol{e}^t) \in \mathbb{R}^N$$

- We use α^t to take a weighted sum of the encoder hidden states to get the attention output $\boldsymbol{a}_t$

$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$



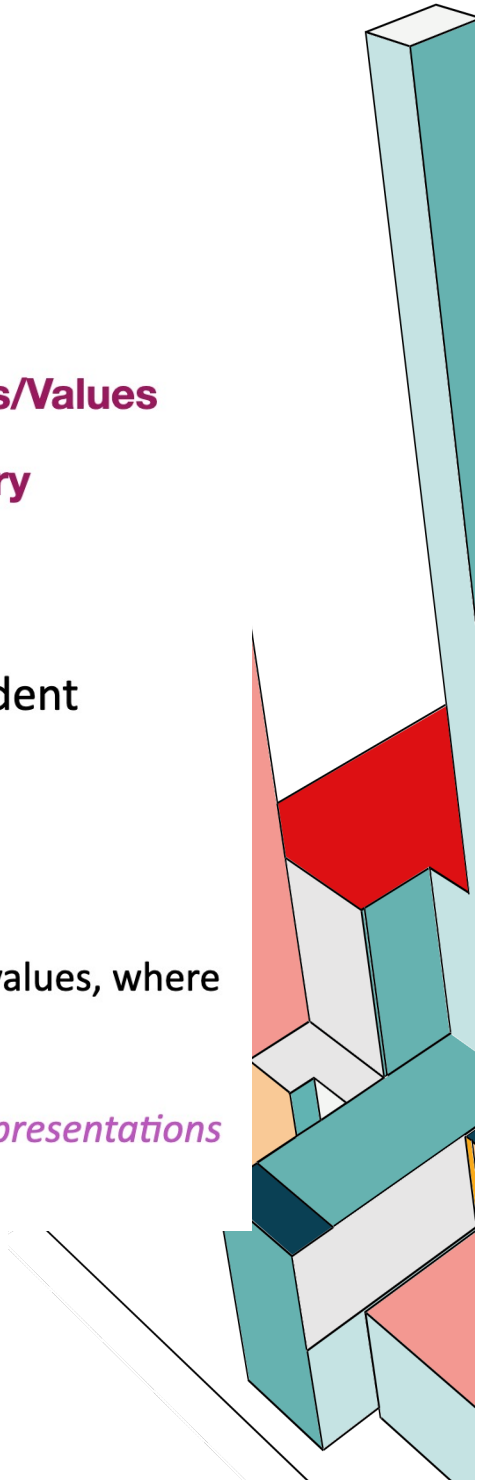
Attention is a *general* Deep Learning technique

- We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$ **Keys/Values**
- On timestep *t*, we have decoder hidden state $s_t \in \mathbb{R}^h$ **Query**

- Given a set of key-value vectors, and a *query vector*, **attention** is a technique to compute a weighted sum of the value vectors, dependent on the query-key relevance.

Intuition:

- The weighted sum is a *selective summary* of the information contained in the values, where the query-key determines which values to focus on.
- Attention is a way to obtain a *fixed-size representation of an arbitrary set of representations* (values), dependent on some other representation (query-key).



Attention in Matrix Notation

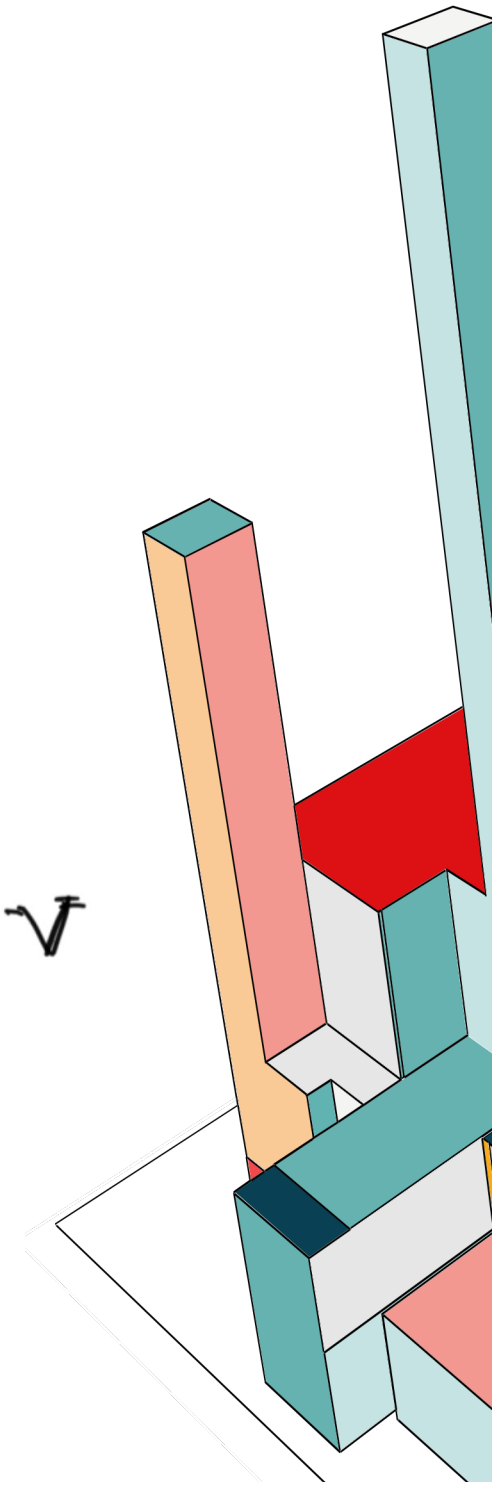
- We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$
- On timestep *t*, we have decoder hidden state $s_t \in \mathbb{R}^h$

- Dot-product attention in matrix notation

$$\begin{bmatrix} - s_1 - \\ - s_2 - \\ \vdots \\ - s_T - \end{bmatrix} \quad \begin{bmatrix} - h_1 - \\ - h_2 - \\ \vdots \\ - h_N - \end{bmatrix}$$

Q $\quad$ $K = V$

$$A = \mathcal{S}(Q K^T) V$$



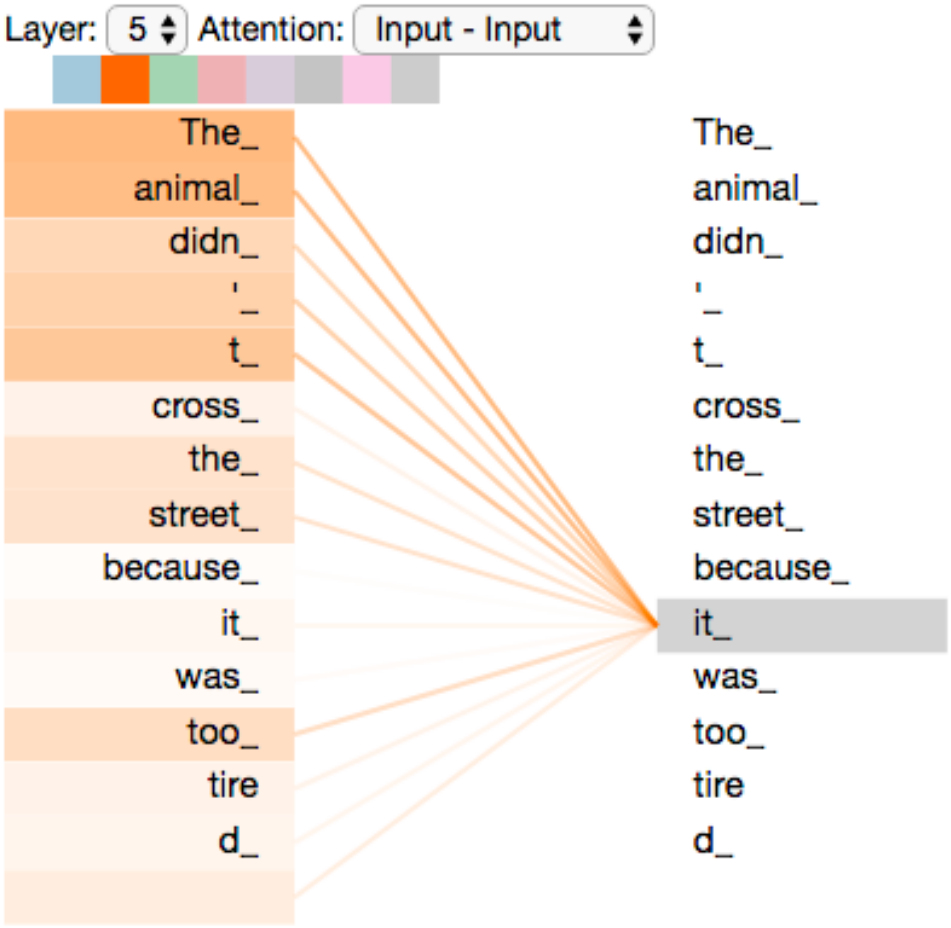
SELF-ATTENTION

ATTENTIONS?

- Attention between encoder and decoder is crucial in NMT

Why not use attention for representations?

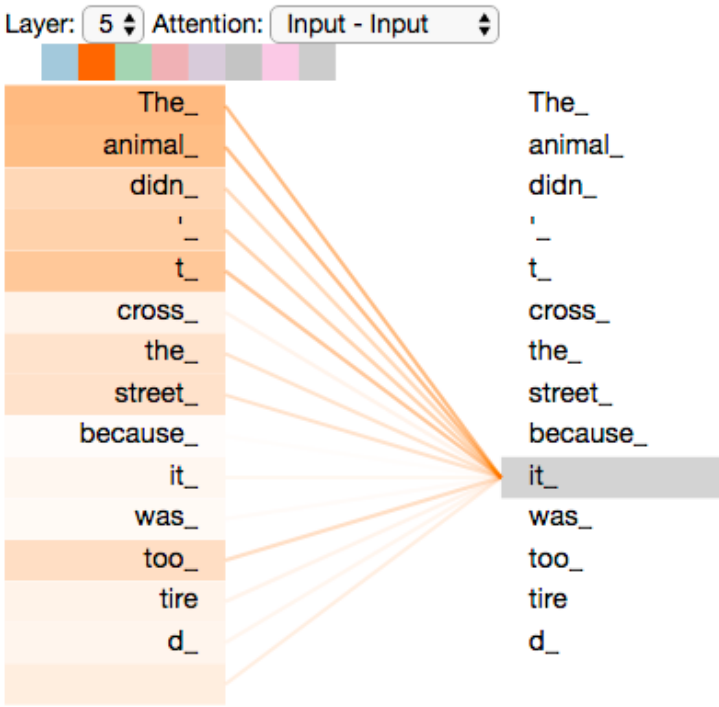
SELF ATTENTIONS



source: Vaswani et al.

SELF ATTENTIONS

- Constant ‘path length’ between any two positions.
- (Soft) Gating & multiplicative interactions.
- Easy to parallelize (per layer).
- Can replace sequential computation entirely



source: Vaswani et al.

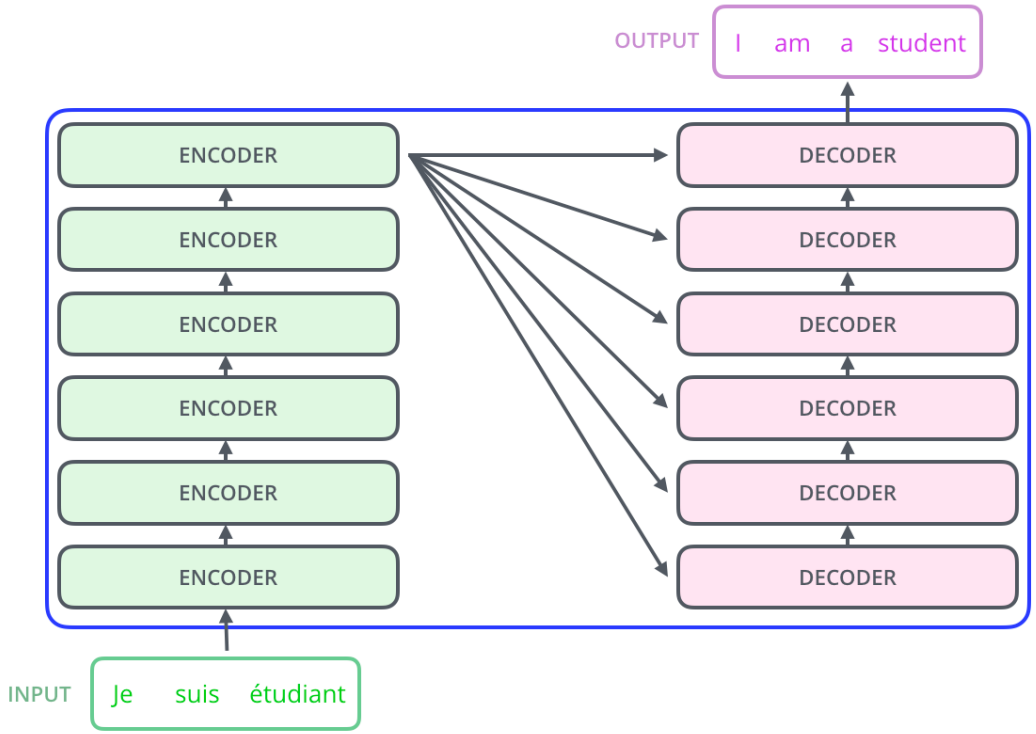
ATTENTION IS CHEAP

	FLOPs
Self-Attention	$O(\text{length}^2 \cdot \text{dim})$
RNN (LSTM)	$O(\text{length} \cdot \text{dim}^2)$
Convolution	$O(\text{length} \cdot \text{dim}^2 \cdot \text{kernel\_width})$

source: Vaswani et al.

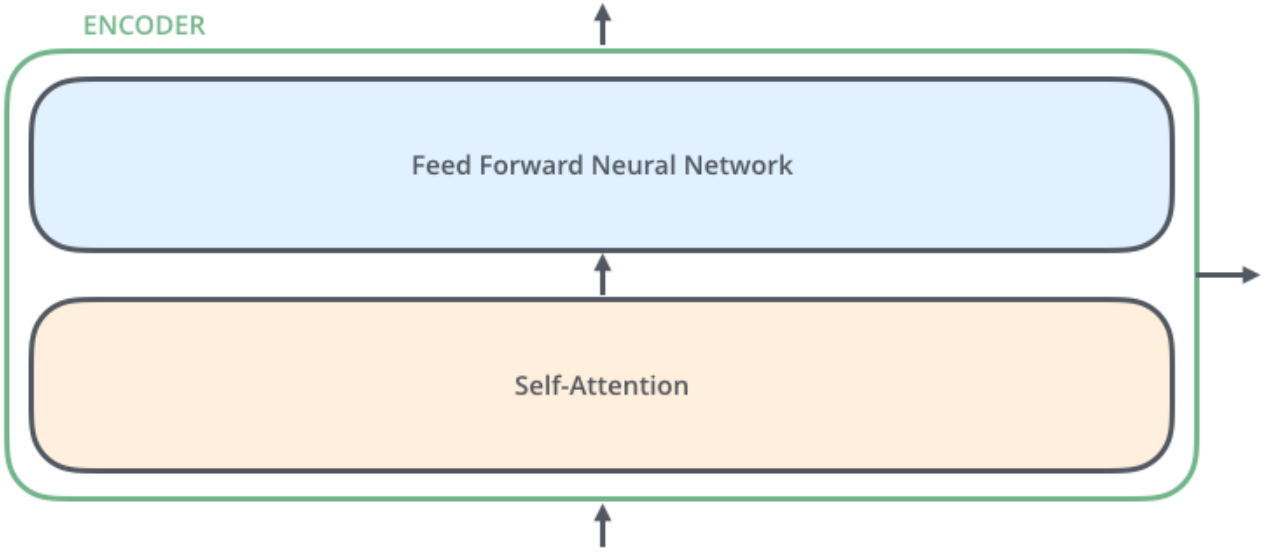
TRANSFORMER ARCHITECTURE

ENCODER-DECODER NETWORK



source: https://jalammar.github.io/

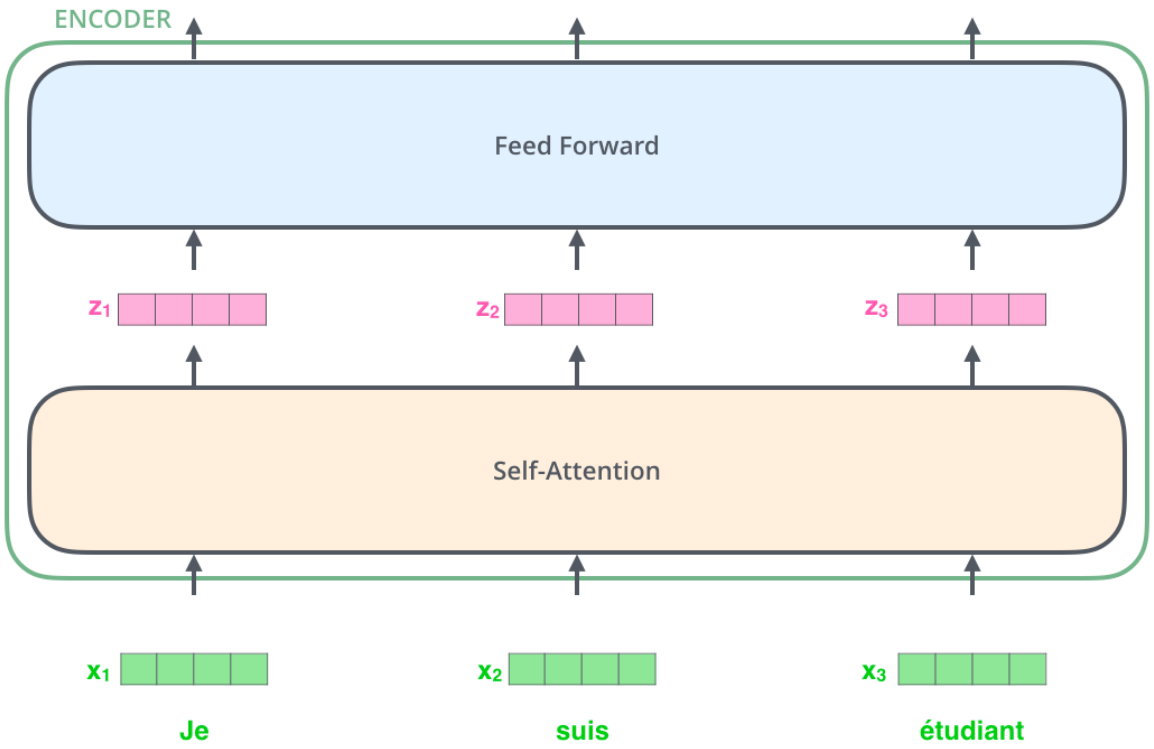
ONE ENCODER LAYER



- The encoder's inputs first flow through a self-attention layer
- The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

source: https://jalammar.github.io/

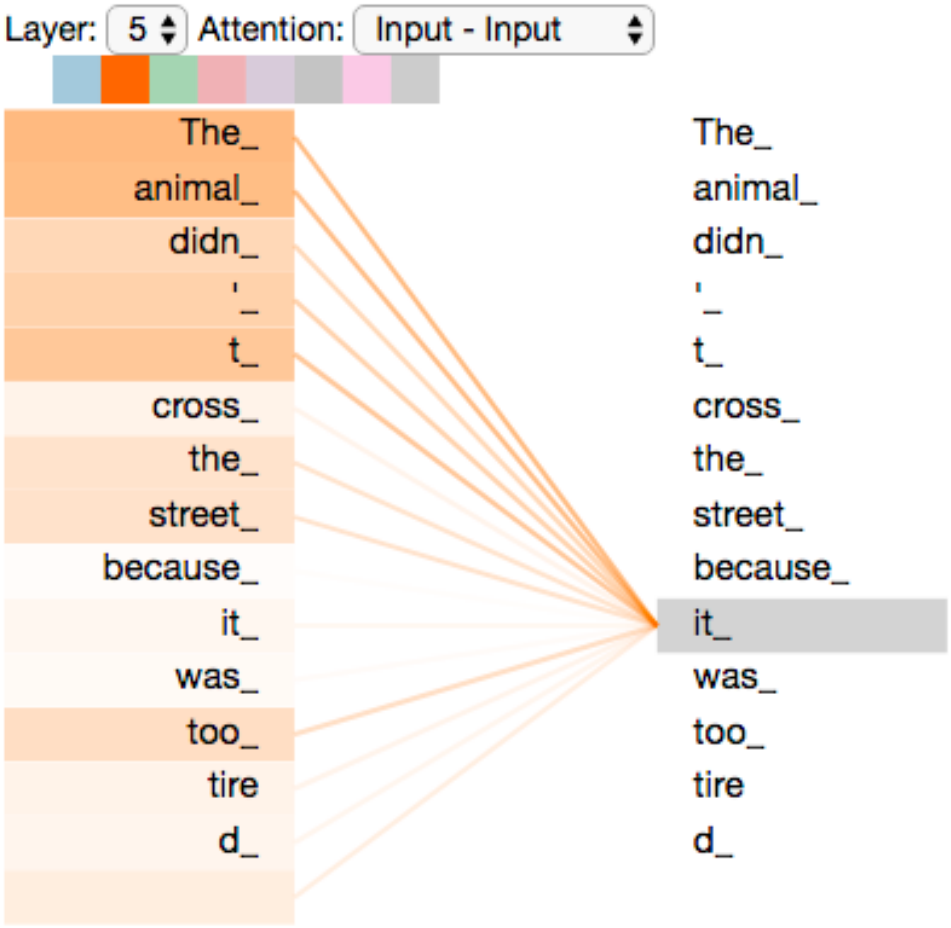
THE FIRST ENCODER LAYER



- The dependencies between word vectors are modelled by the self-attention layer.

source: https://jalammar.github.io/

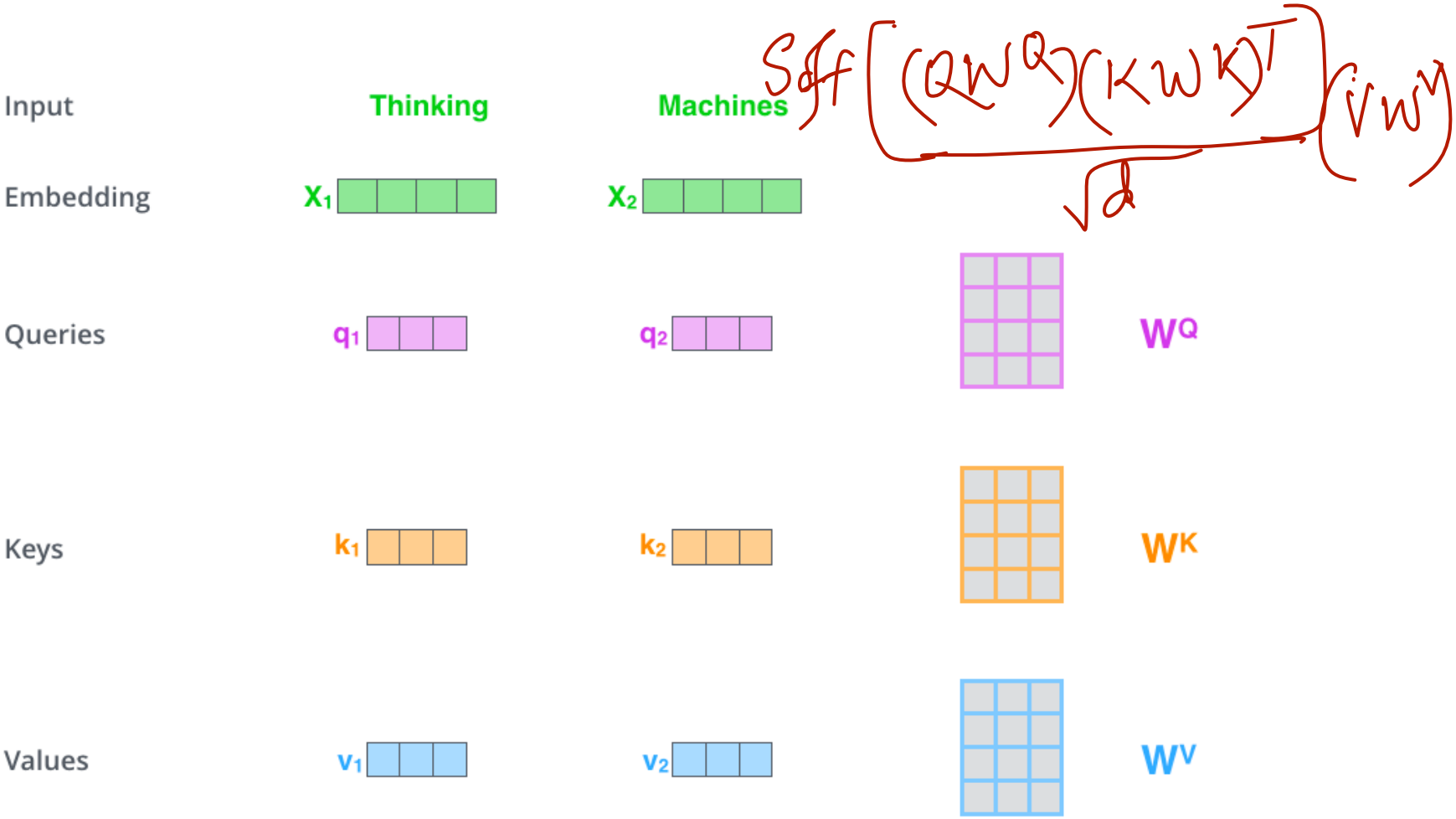
ONE SELF-ATTENTION LAYER



- The dependencies between word vectors are modelled by the self-attention layer.

source: https://jalammar.github.io/

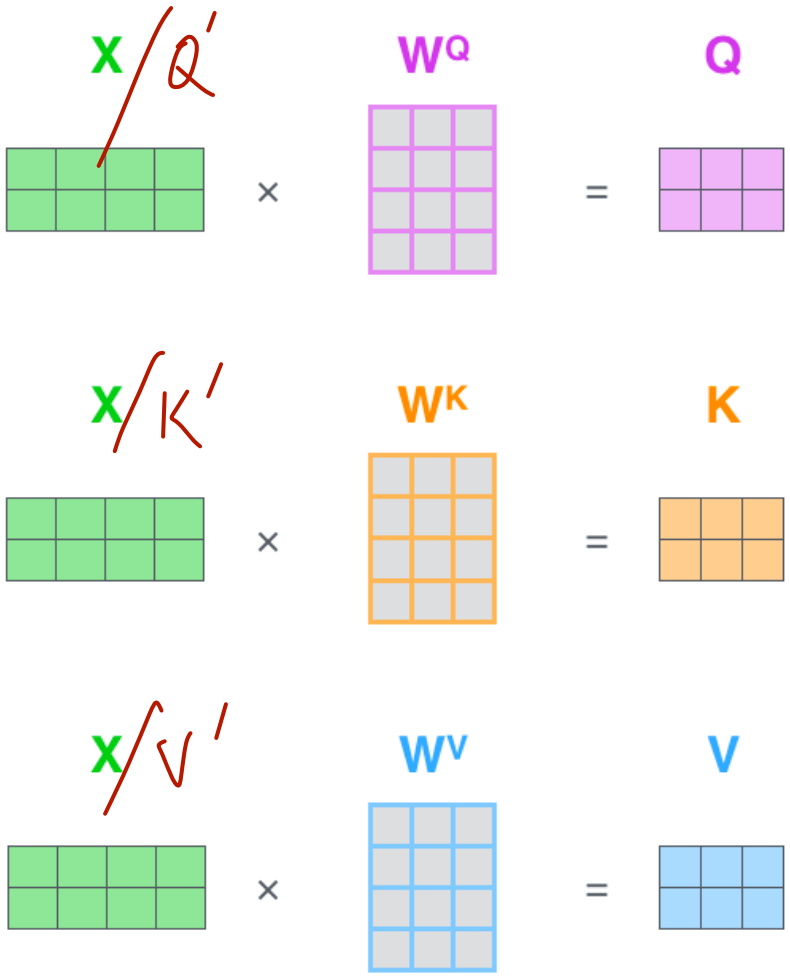
ONE SELF-ATTENTION LAYER



- The dependencies between word vectors are modelled by the self-attention layer.

source: https://jalammar.github.io/

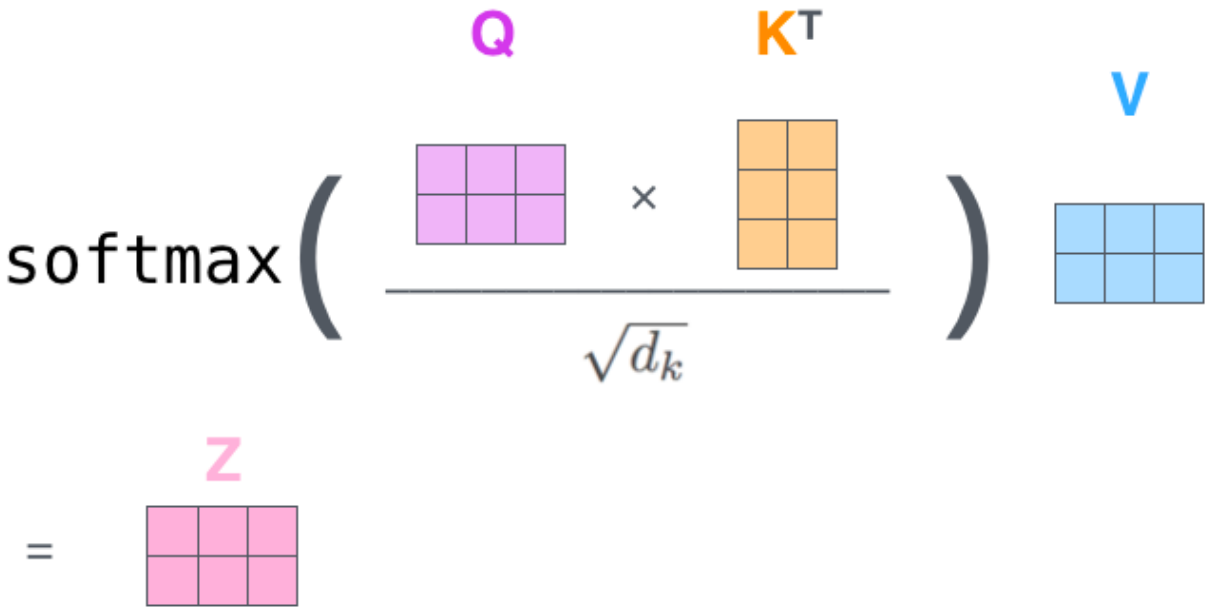
ONE SELF-ATTENTION LAYER



- The dependencies between word vectors are modelled by the self-attention layer.

source: https://jalammar.github.io/

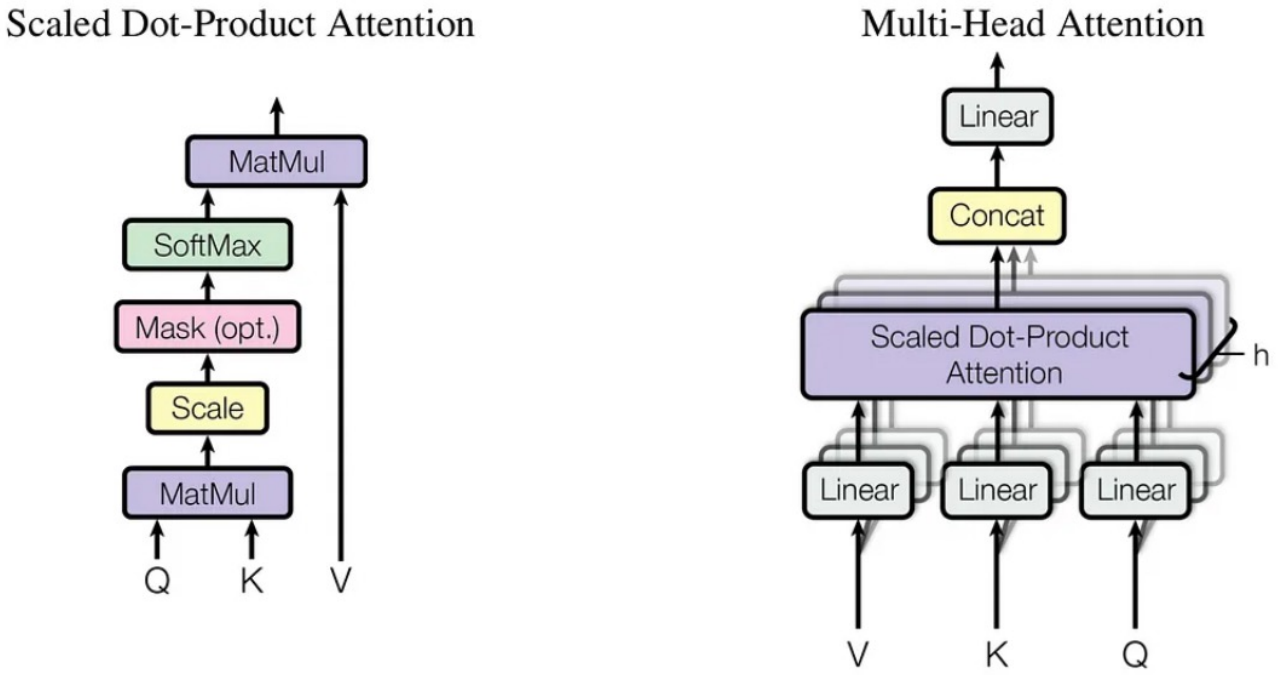
ONE SELF-ATTENTION LAYER



- The dependencies between word vectors are modelled by the self-attention layer.

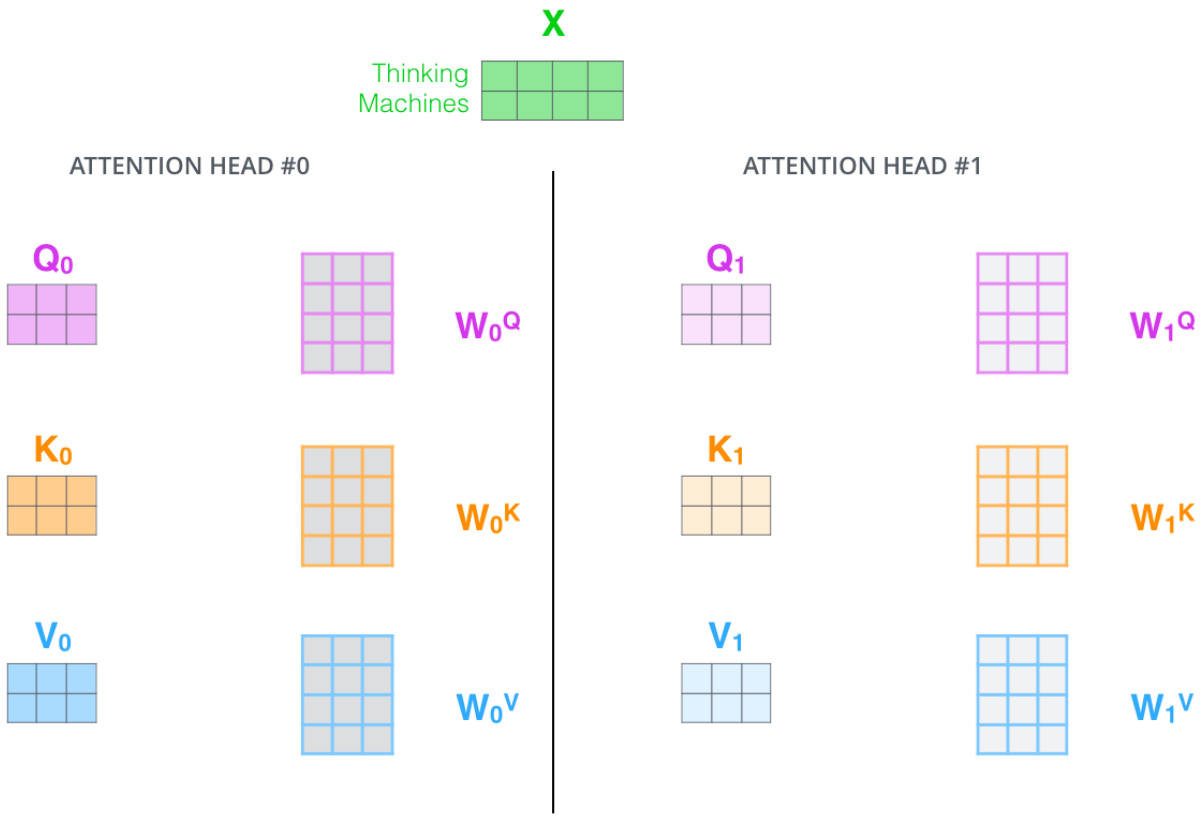
source: https://jalammar.github.io/

SELF-ATTENTION - MULTIHEAD



source: Vaswani et al.

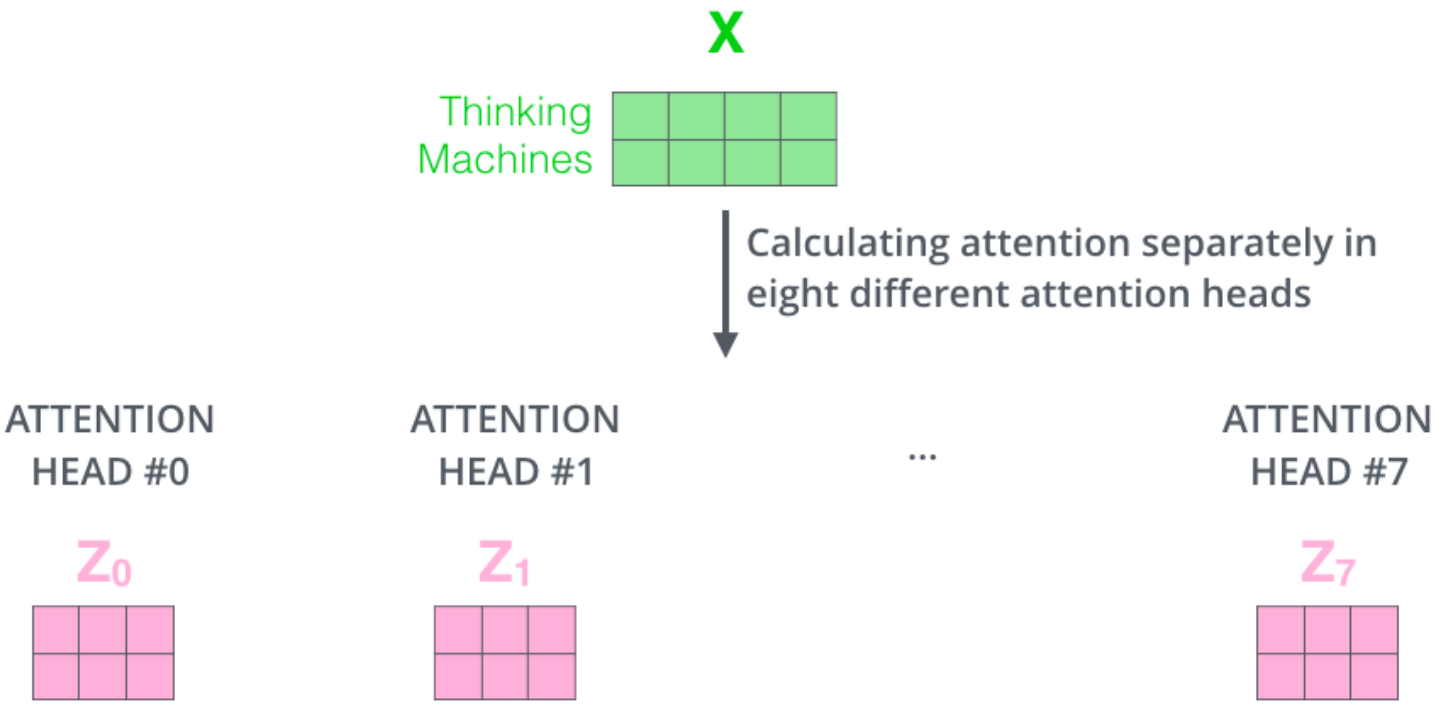
USING MULTIPLE HEADS



- Multiple heads expand the model's ability to focus on different positions.
- Each head gives the attention layer multiple representation subspaces

source: https://jalammar.github.io/

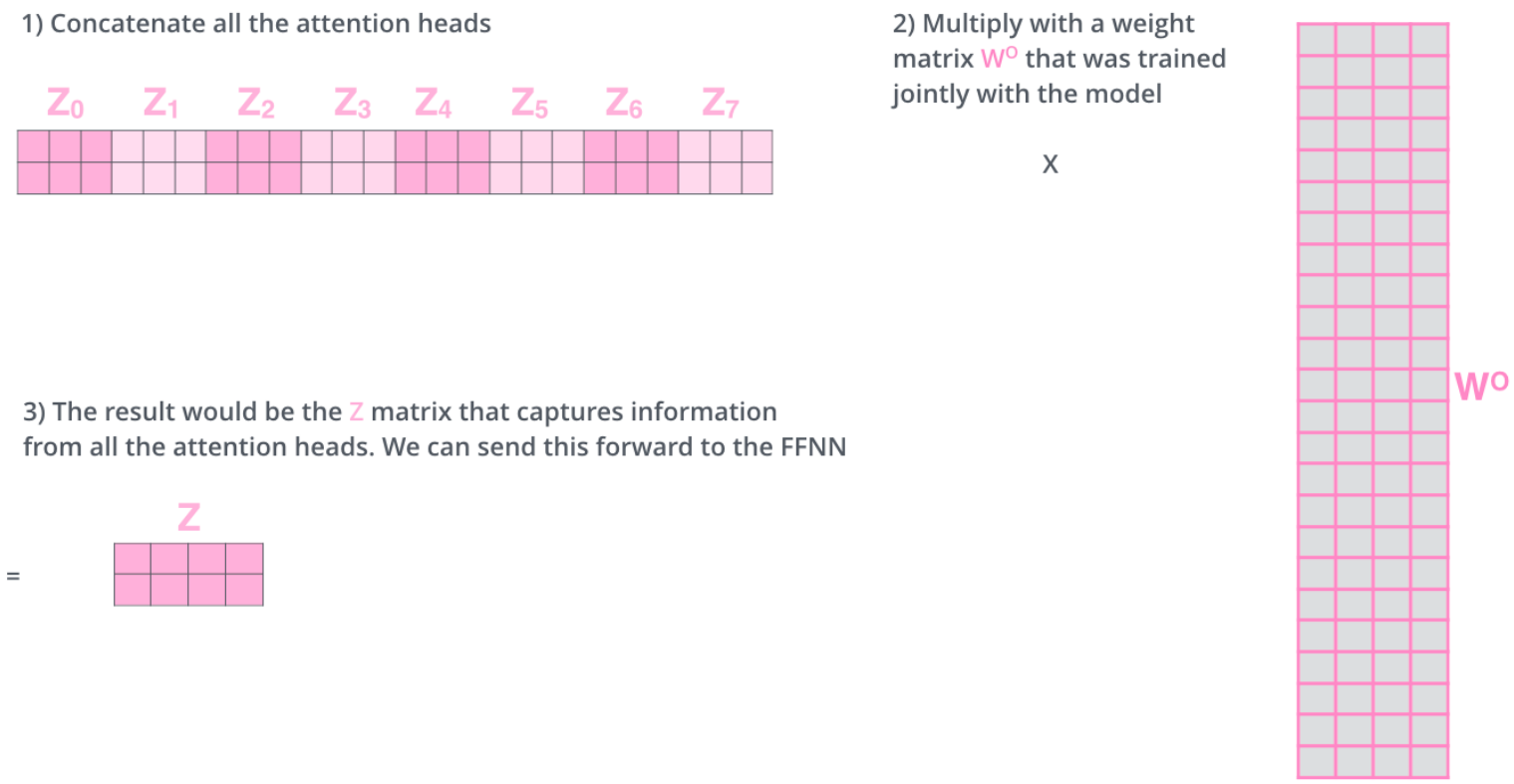
USING MULTIPLE HEADS



- Transformer uses 8 different heads

source: https://jalammar.github.io/

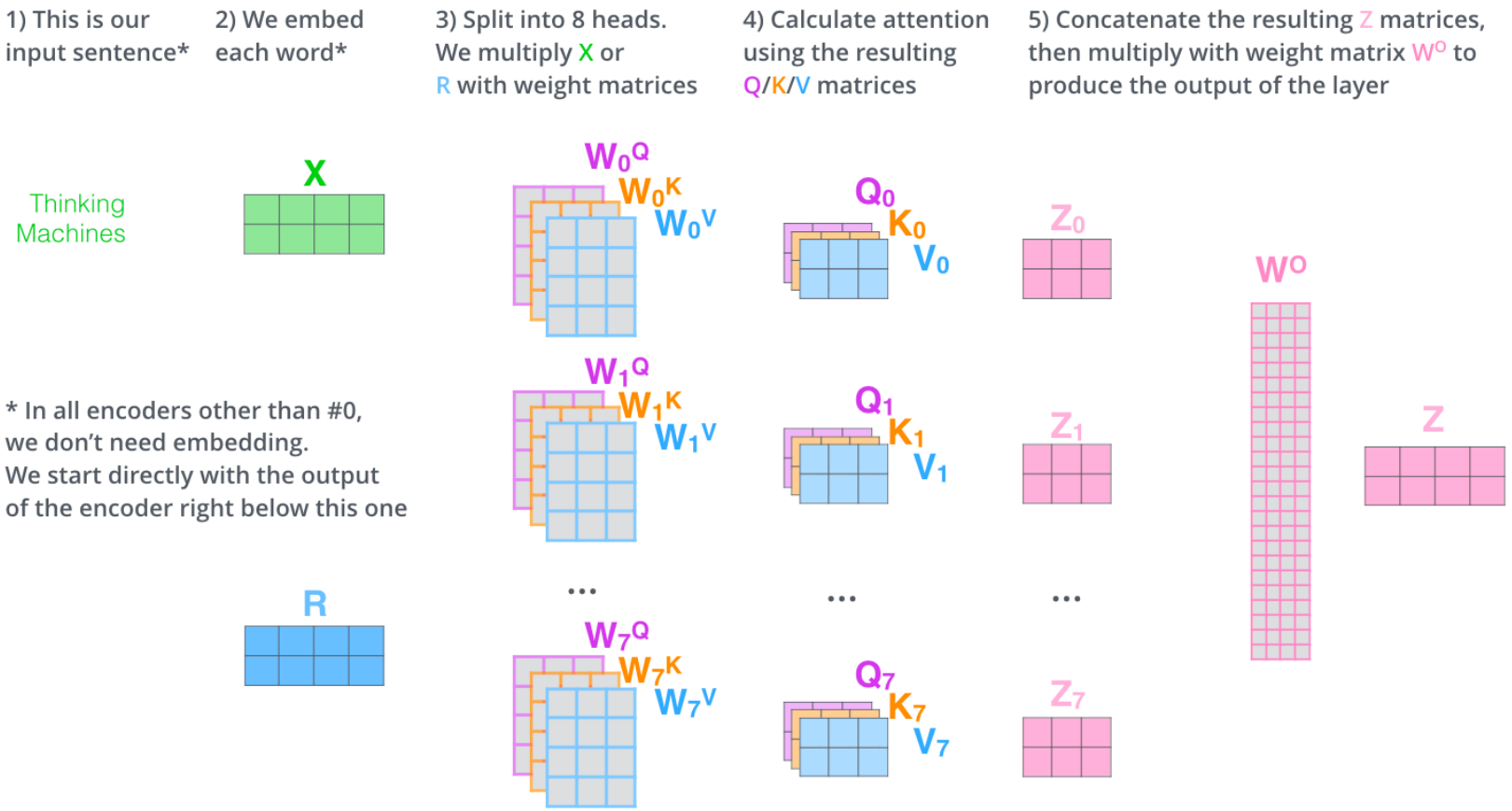
USING MULTIPLE HEADS



- Concatenate the 8-head's outputs and pass it the FFN

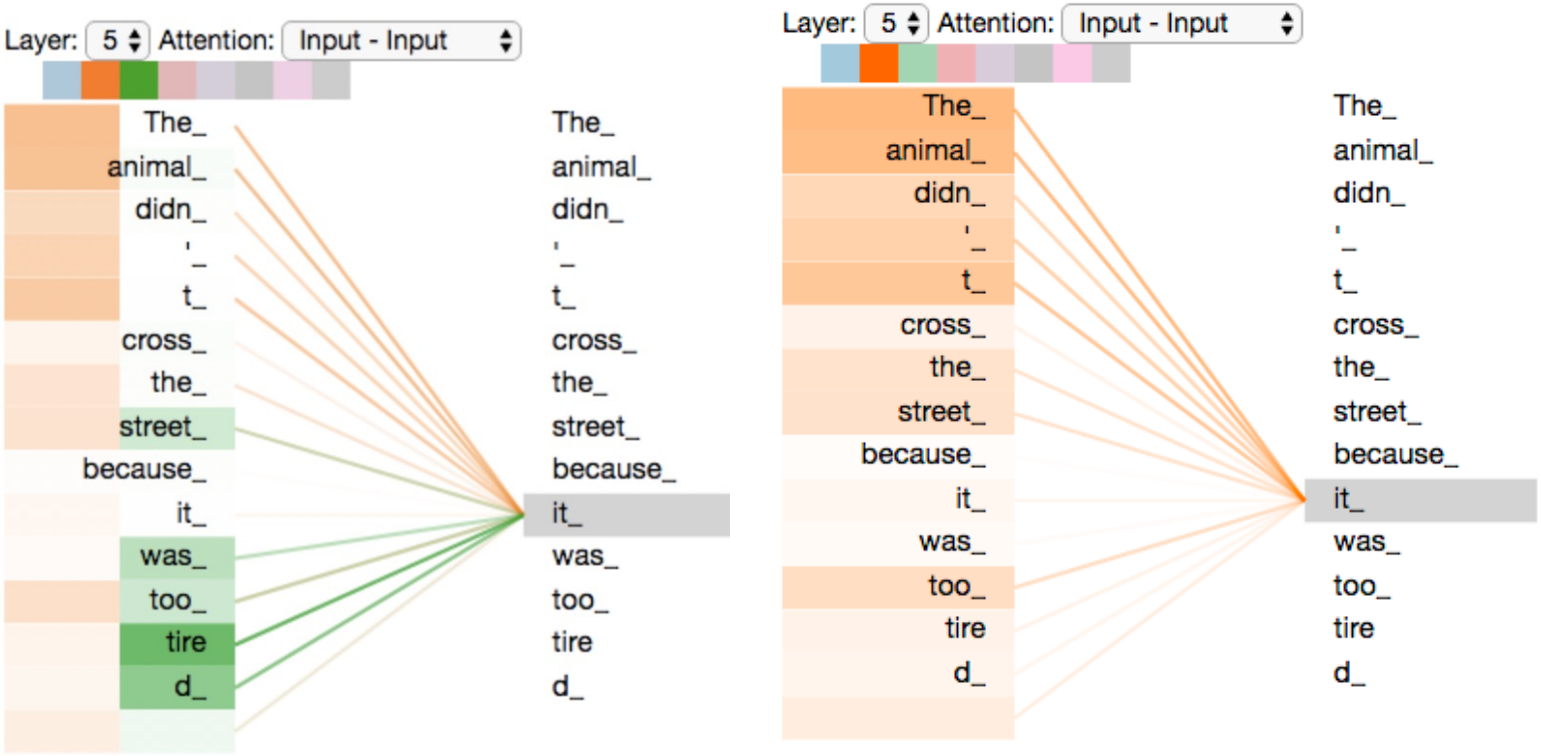
source: https://jalammar.github.io/

USING MULTIPLE HEADS



source: https://jalammar.github.io/

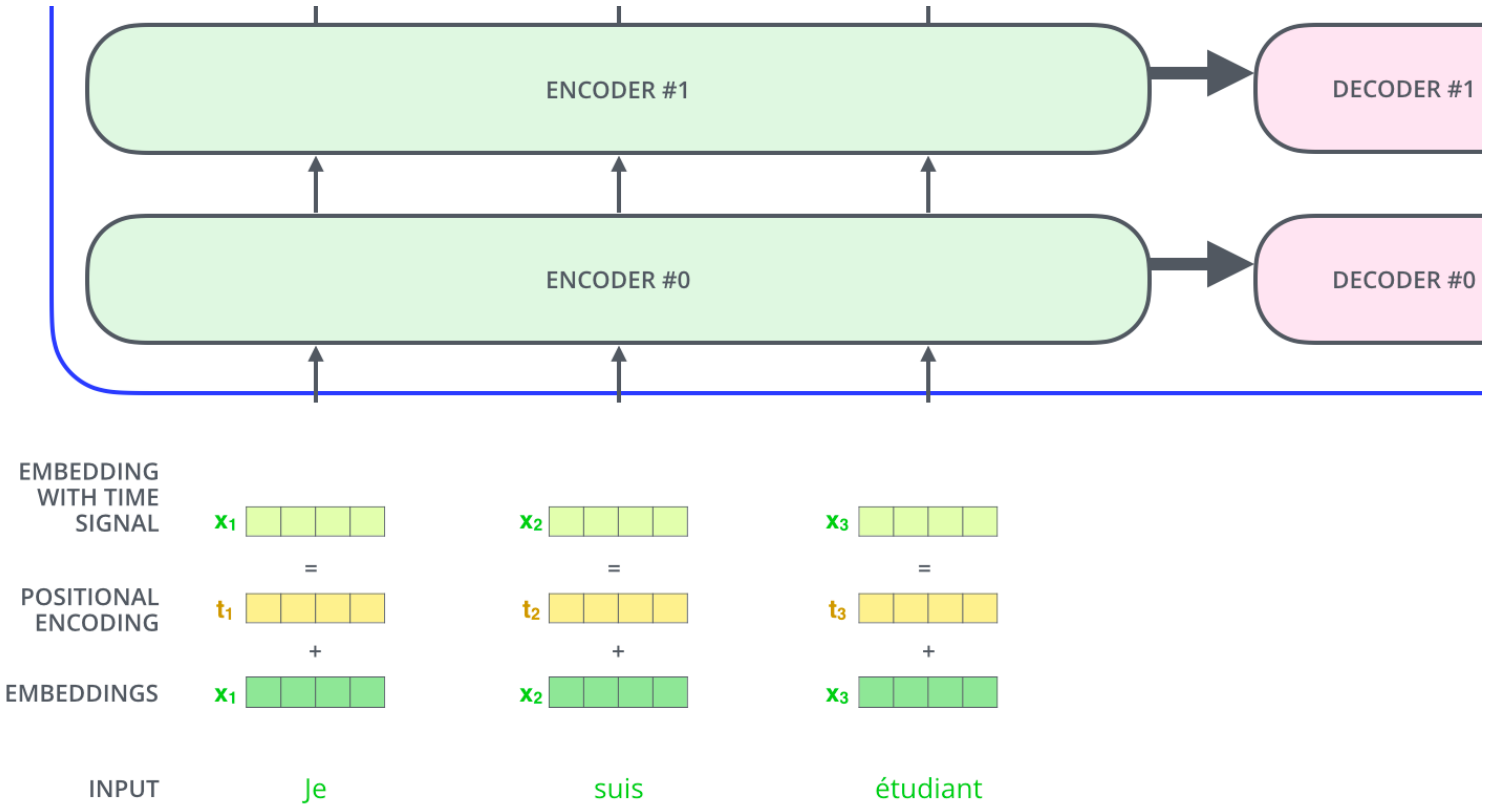
USING MULTIPLE HEADS



As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

source: https://jalammar.github.io/

INJECTING ORDER INTO EMBEDDINGS

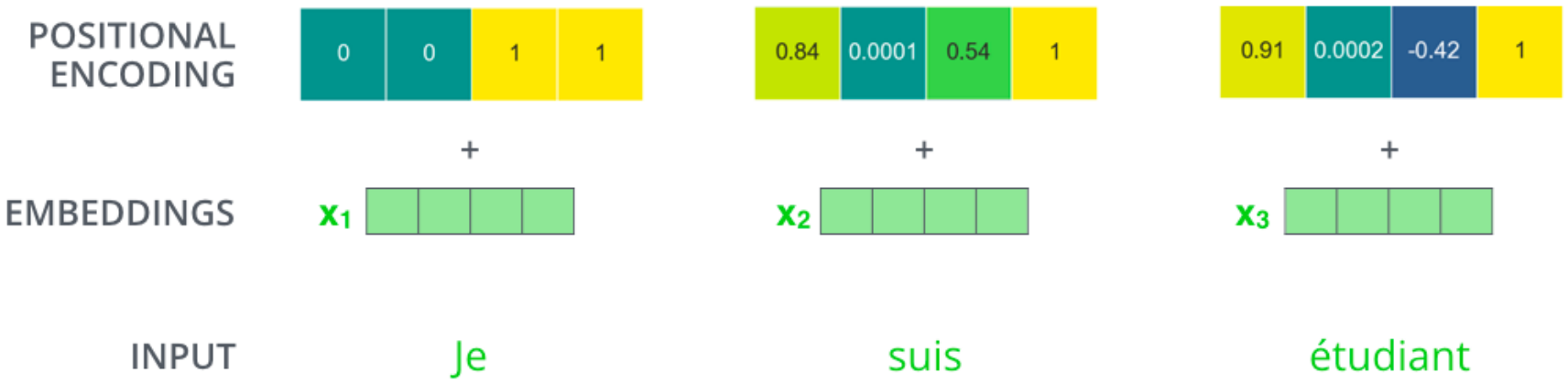


To give the model a sense of the order of the words, we add positional encoding vectors – the values of which follow a specific pattern (sine/cosine).

source: https://jalammar.github.io/

INJECTING ORDER INTO EMBEDDINGS

- Actual word representations are byte-pair encodings
  - As seen in last lecture

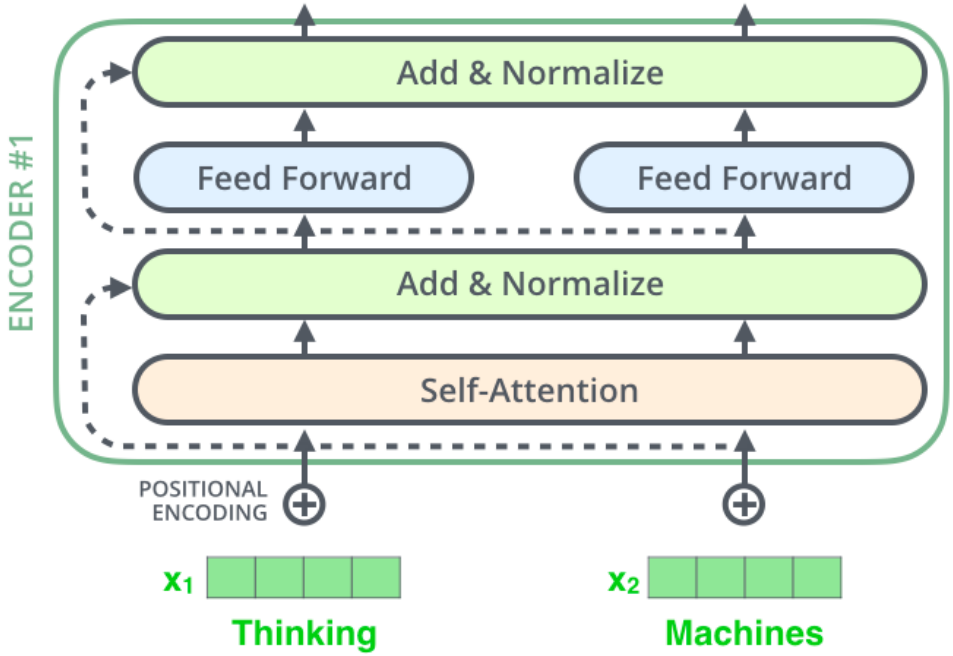


Example of positional encoding with a toy embedding size of 4

source: https://jalammar.github.io/

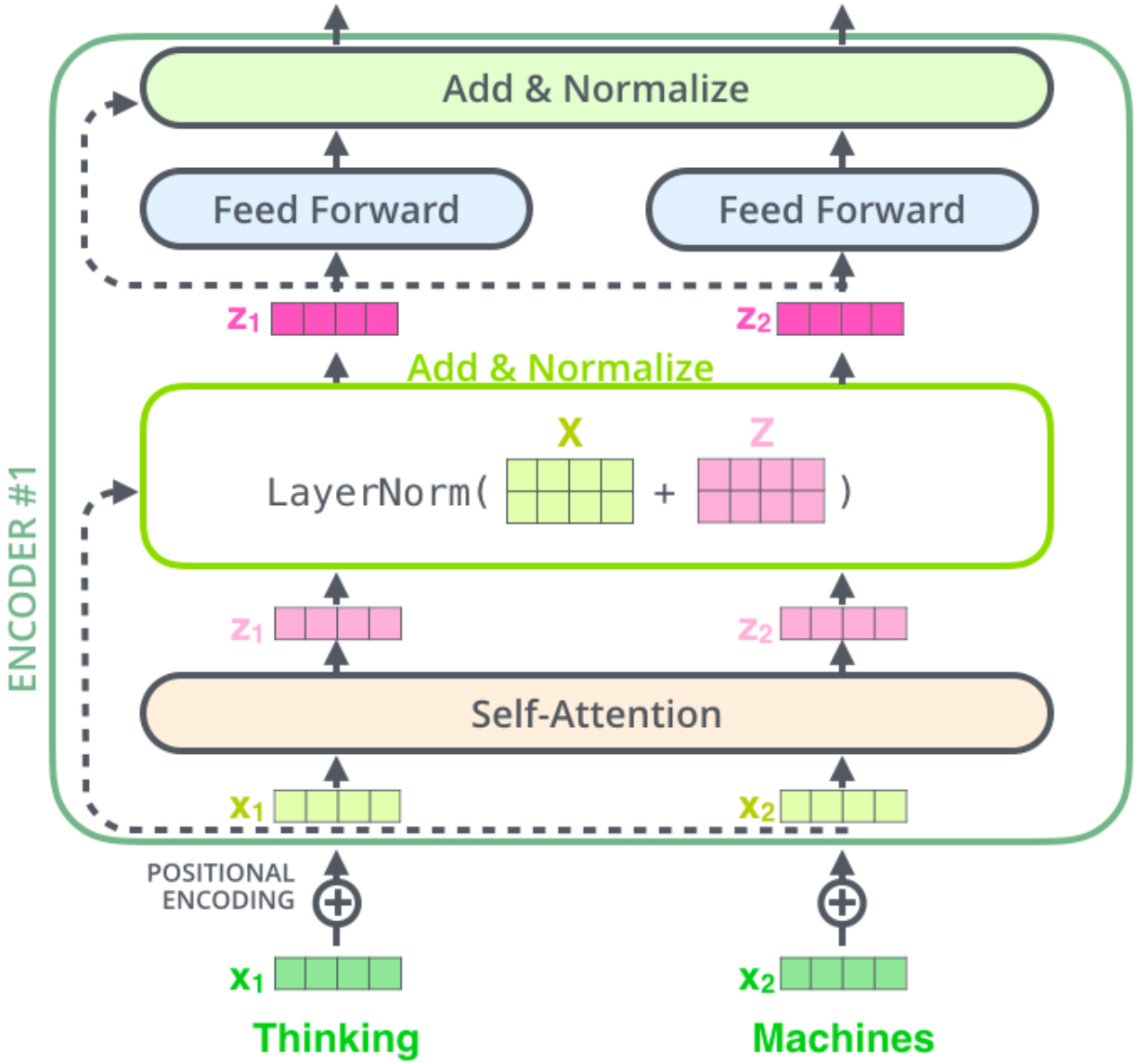
THE RESIDUALS

- Each sub-layer (self-attention, FFNN) in each encoder has a residual connection around it, and is followed by a layer-normalization step
- Layer-norm changes input to have mean 0 and variance 1, per layer and per training point



source: https://jalammar.github.io/

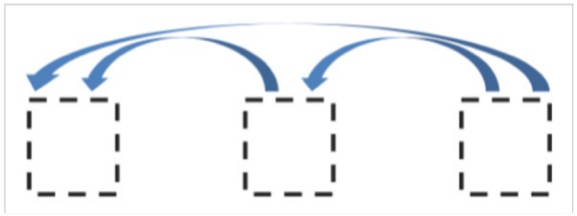
THE RESIDUALS



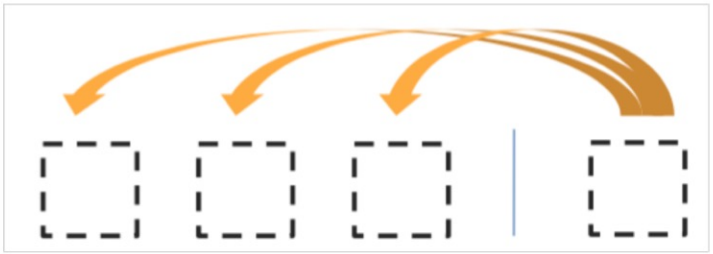
source: https://jalammar.github.io/

DECODER LAYERS

- 2 sublayer changes in decoder
- Masked decoder self-attention on previously generated outputs:

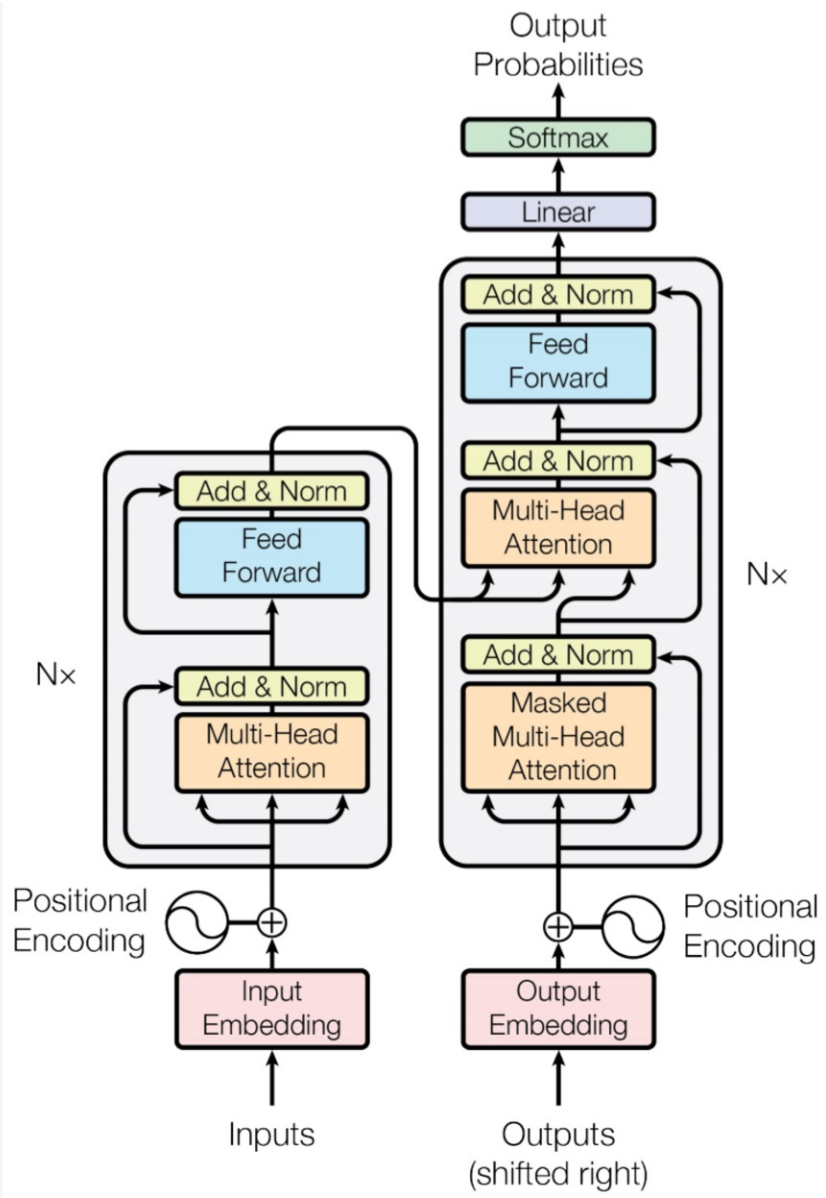


- Encoder-Decoder Attention, where queries come from previous decoder layer and keys and values come from output of encoder



- Blocks repeated 6 times also

WHOLE ENCODER-DECODER MODEL




Q&A