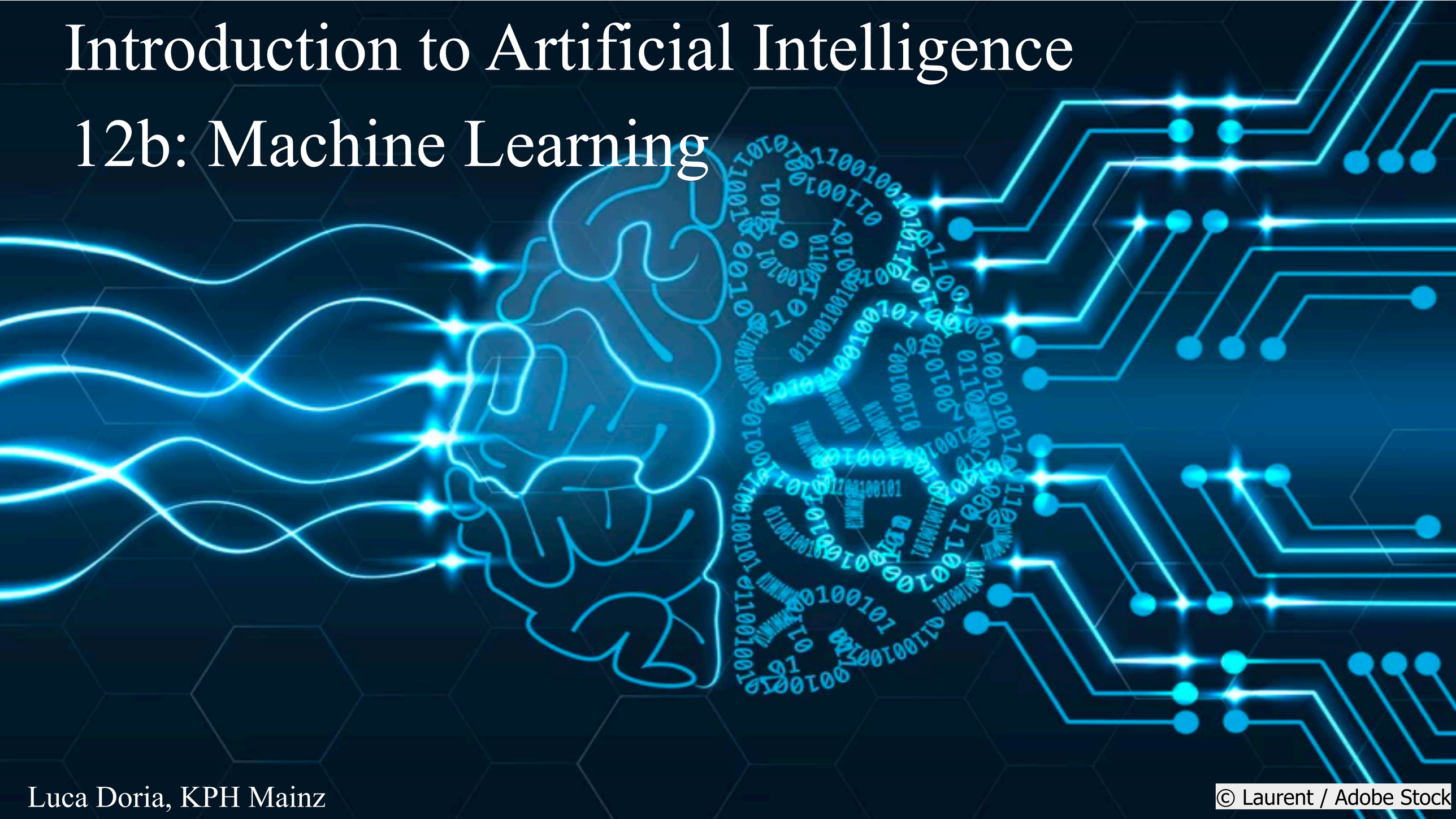
Introduction to Artificial Intelligence

12b: Machine Learning



Luca Doria, KPH Mainz

Neural Networks

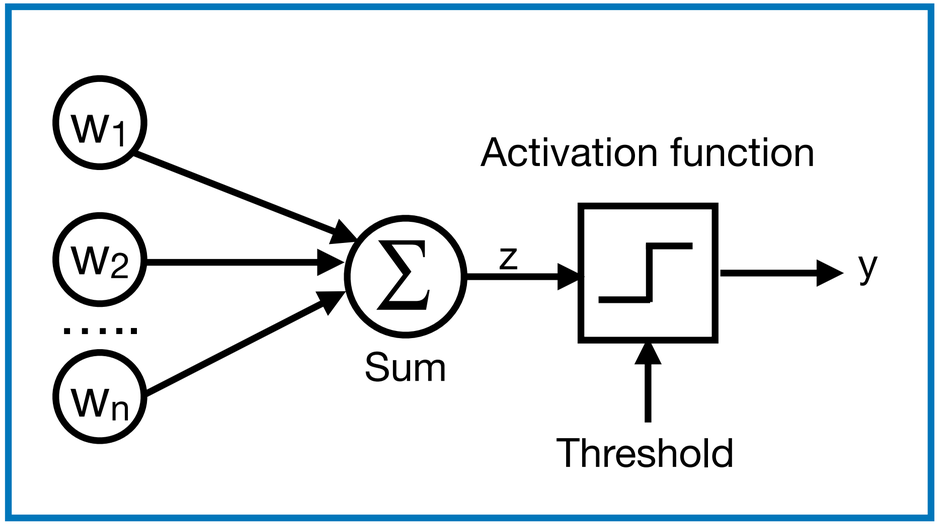
- Long history, with “ups” & “downs”:
- N. Rashevsky (1030s): likely the first works on neural learning
- McCullochs and Pitts (1943)
- M. Minsky and Papert : first working neural hardware (1951)
- The perceptron (F. Rosenblatt, 1957)
- First multi-layer networks: 1960s
- Book: Perceptrons by Minsky&Papert: 1 layer can only do linear separations!
- Starting from the 1960s: many “discoveries” of back propagation
- 1980s: multilayer networks are universal approximations (G. Cybenko’s theorem)
- 2000s: SVNs and Bayes networks shadowed NNs
- >2010s: deep learning revolution (+big data, +hardware)

The Perceptron

The Rosenblatt's perceptron:

$$z = \mathbf{w} \cdot \mathbf{x} + b = \sum_{1}^{n} w_i x_i + b$$

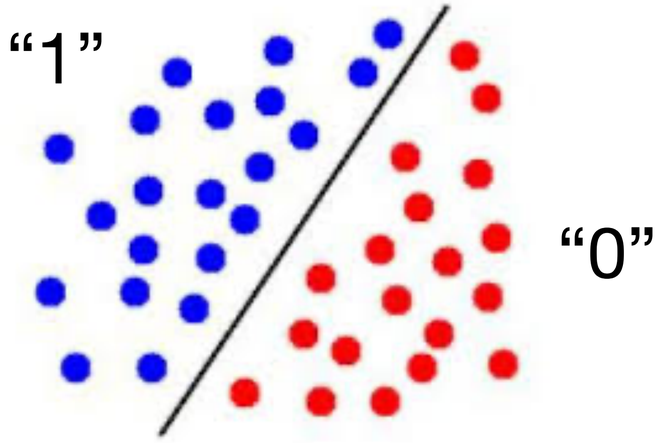
$$y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$



Learning (iterative algorithm):

$$w_i \leftarrow w_i + \Delta w_i \quad \text{for all } i$$
$$\Delta w_i = \eta(y_{\text{true}} - y_{\text{pred}})x_i$$
$$b \leftarrow b + \eta(y_{\text{true}} - y_{\text{pred}})$$

η: learning rate



Example: XOR, a linearly non-separable problem

AND	0	1	OR	0	1	XOR	0	1
0	0	0	0	0	1	0	0	1
1	0	1	1	1	1	1	1	0

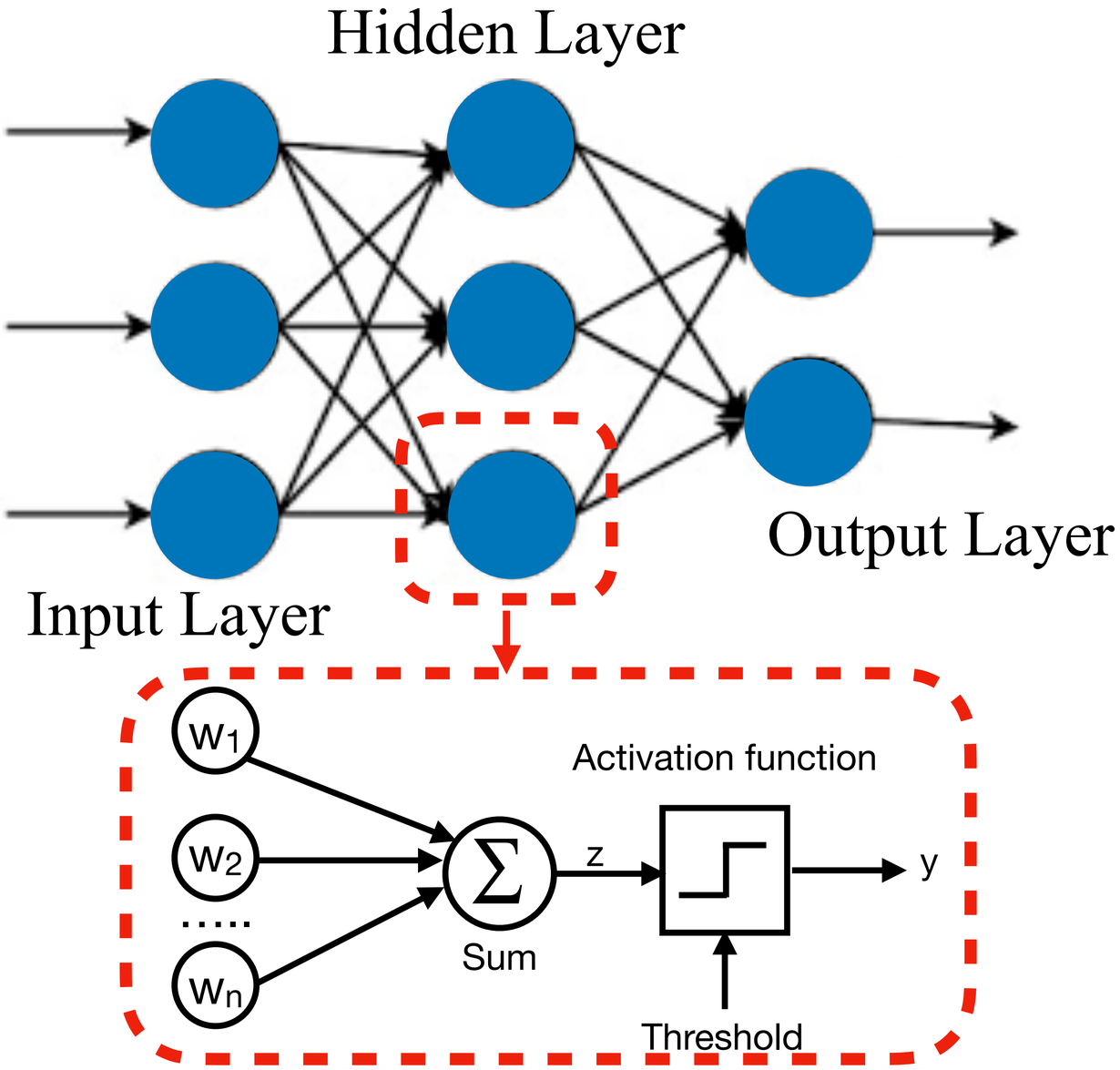
Example: XOR, a linearly non-separable problem

AND	0	1	OR	0	1	XOR	0	1
0	0	0	0	0	1	0	0	1
1	0	1	1	1	1	1	1	0

Example: XOR, a linearly non-separable problem

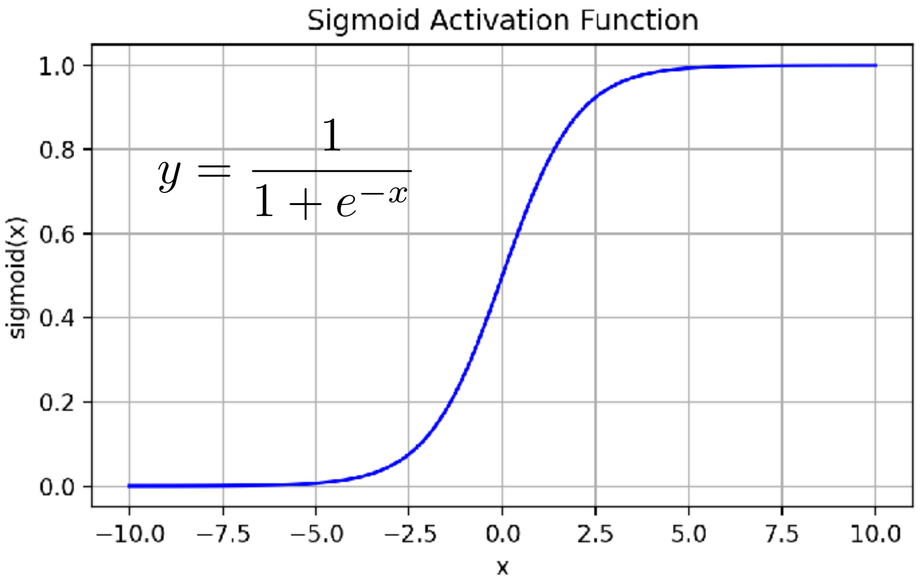
AND	0	1	OR	0	1	XOR	0	1
0	0	0	0	0	1	0	0	1
1	0	1	1	1	1	1	1	0

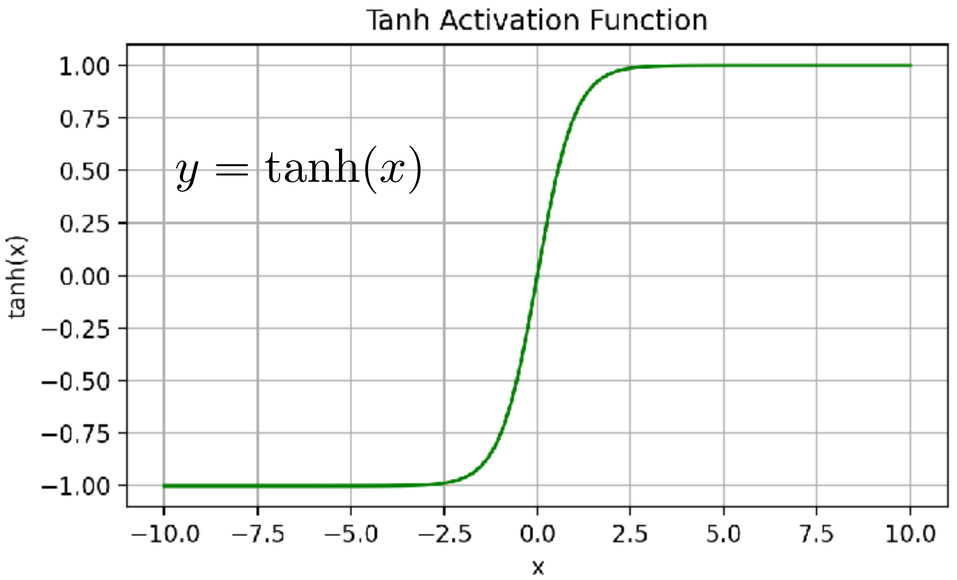
The Solution: Multi-Layer Perceptrons

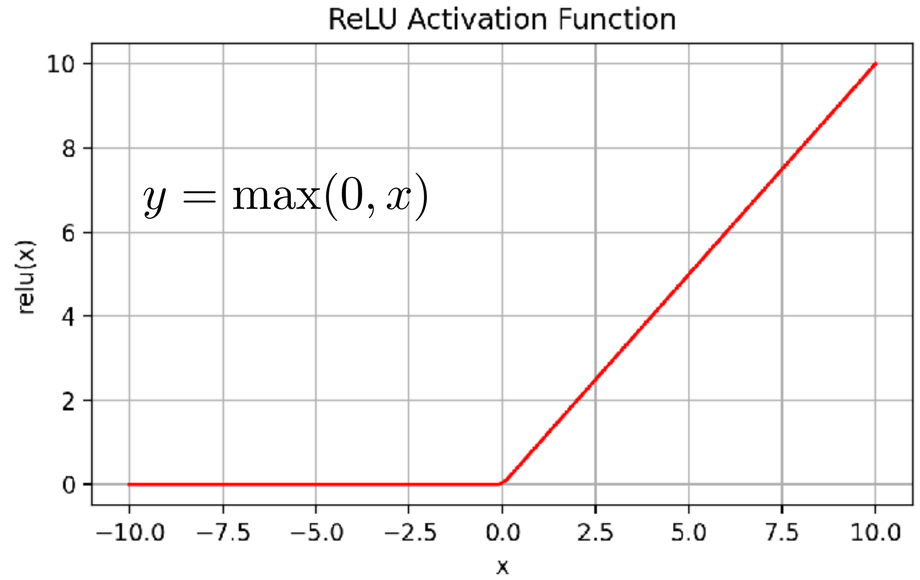


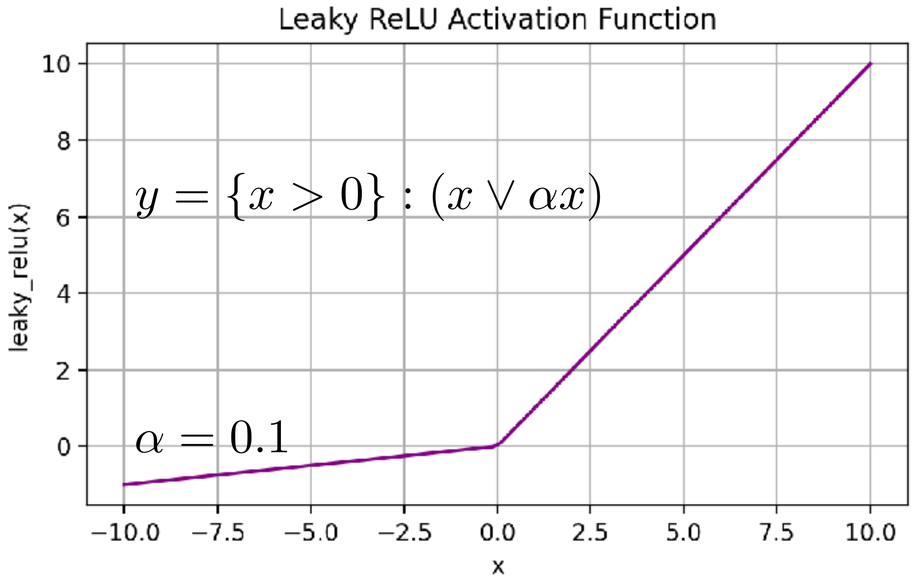
Combine Perceptrons in a layered structure: a Neural Network.

Activation Functions



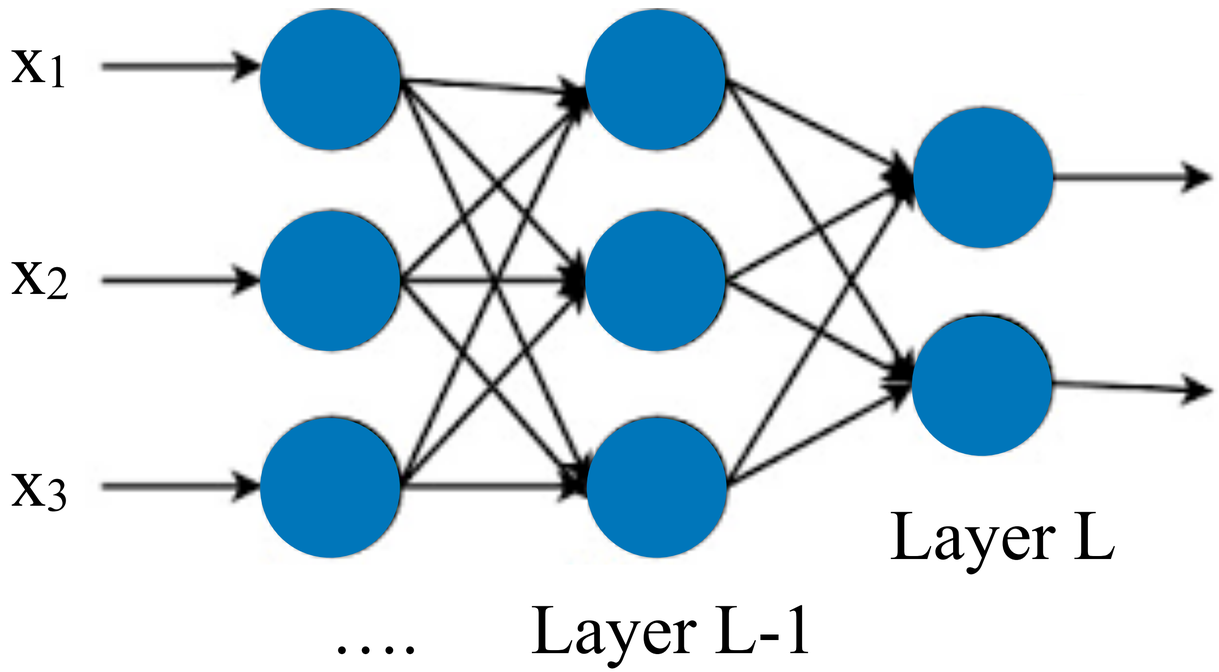






Training a Multi-Layer Perceptron

Idea: reduce the prediction error of the NN



$$E \propto \sum_x \|y(x) - a^L(x)\|$$

We can calculate the error on the output layer L and then adjust the NN weights such that the error is minimised. How?

Backpropagation

- This algorithm was re-"discovered" several times in the past:
- P. Werbos (1974): maybe the first application to NNS
- D. Rumerlhart, G. Hinton, R. Williams (1986): Nature paper, heuristic presentation
- G. Cybenko (1988,1989): mathematical proof that FFNNs are universal function approximators.: *"any continuous function on the unit cube can be uniformly approximated by a one layer network with an arbitrary continuous sigmoidal nonlinearity."*
- From the 1980s to the 1990s NN research grew by a factor > 100.

Backpropagation

- It is essentially based on the mathematical operation of derivative of composite functions.
- The idea is to calculate the error on layer L and then proceed backwards modifying the weights of layer L-1, then L-2, and so on for minimising the error.
- Two phases:
  - Feed forward: propagate the input data forward calculate the NN output.
  - Backpropagation: propagate the error backwards and minimise it.

Definitions

Commonly used error (root mean square): $C = \frac{1}{N} \sum_x \|y(x) - a^L(x)\|^2$

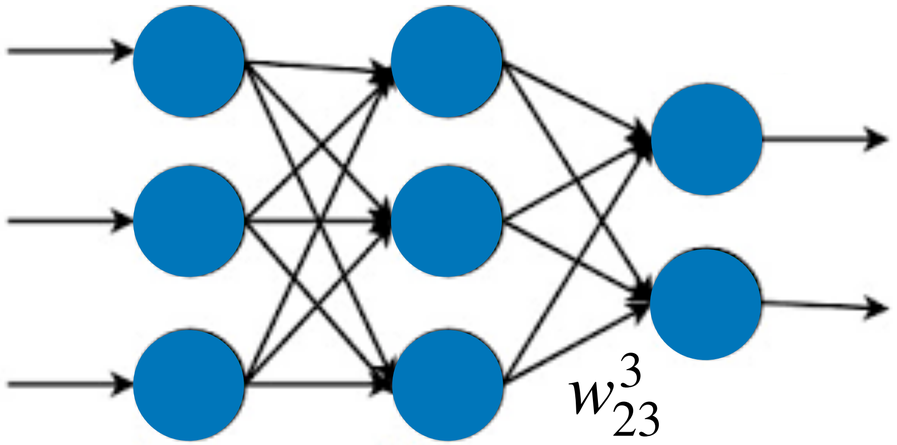
Weights: w_{jk}^l

- l: Layer
- k: k-th neuron connecting with layer j

Neuron output: $a_j^l = f\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right)$

argument of f: z_j^l

Vectorization: $a^l = f\left(w^l a^{l-1} + b^l\right)$



Backpropagation Equations (1)

We have to calculate the derivative of the error with respect to the variables of the NN. Ideally, the should start evaluating the change of C with respect to the last neurons' outputs a, but it is algebraically easier to show the change with respect to z.
The change in error of the last layer L is:

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \sum_k \frac{\partial C}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} f'(z_j^L) \qquad \text{(Eq. 1)}$$

- Einstein convention
- if k≠j this is zero, so put k=j

C depends from a which depends from z

Derivative of the activation function.

Backpropagation Equations (2)

The error of the last layer can be directly calculated from the NN output. Now we have to “backpropagate”, i.e. find the error for the other layers:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_k^{l+1}} \frac{\partial a_k^{l+1}}{\partial z_j^l} = \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1} = \frac{\partial}{\partial z_j^l} \left[\sum_i w_{kj}^{l+1} a_j^l + b_k^{l+1} \right] \delta_k^{l+1} \Rightarrow$$

$$\Rightarrow \delta_j^l = w_{kj}^{l+1} \delta_k^{l+1} f'(z_j^l) \quad \text{(Eq. 2)}$$

Backpropagation Equations (3)

With an analogous calculation, we can calculate the change of C with respect to the bias factors b:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad \text{(Eq. 3)}$$

Interestingly, this results equal to a quantity which we calculated already.
The last result, also obtained with a similar chain-derivatives calculation is

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \quad \text{(Eq. 4)}$$

which is also dependent from known quantities.

Learning with Backpropagation

$\odot$: Hadamard "element-wise" product

1) Feed-forward:

$$\vec{x} \rightarrow a^1$$

$$\forall l \quad z^l = w^l a^{l-1} + b^l$$

2) Backpropagation:

$$\delta^L = \nabla_a C \odot f'(z^L) \qquad \forall l \quad \frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \text{(Eq. 1)} \quad \text{(Eq. 2)}$$

$$\forall l \quad \delta^l = (w^{l+1})^T \delta^{l+1} \odot f'(z^l) \qquad \forall l \quad \frac{\partial C}{\partial b^l_j} = \delta^l_j \qquad \text{(Eq. 3)} \quad \text{(Eq. 4)}$$

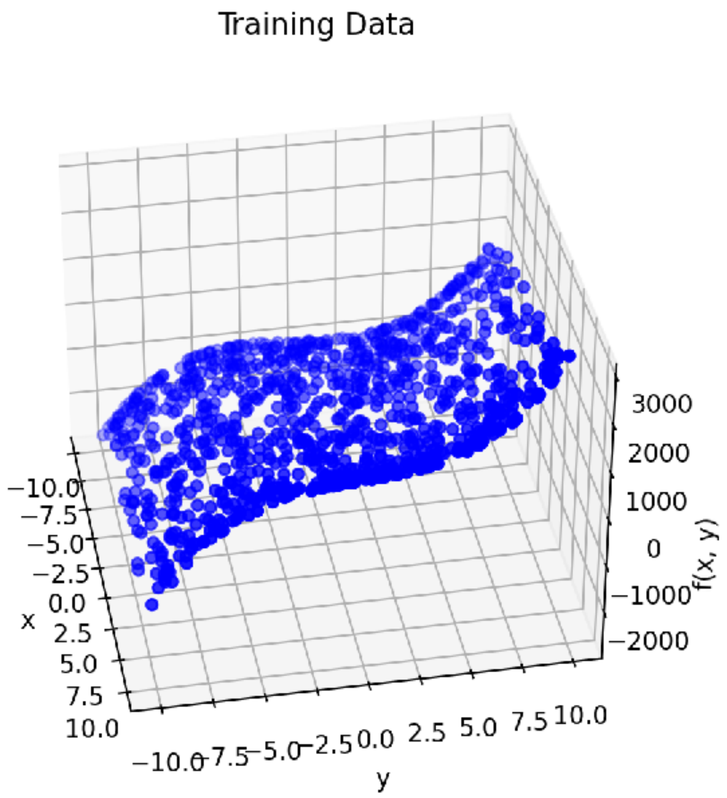
3) Gradient descent (or other opt. alg.)

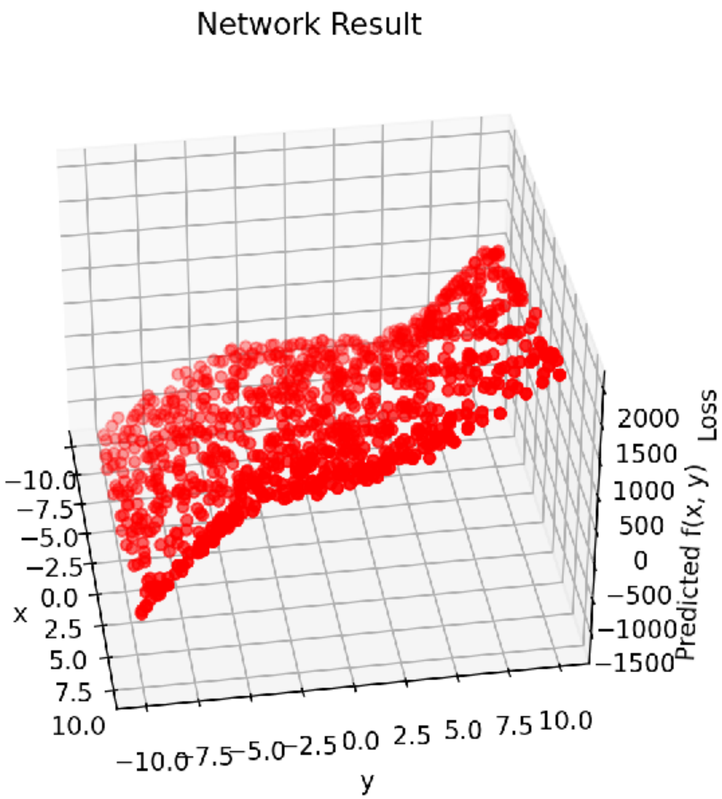
$\forall l$

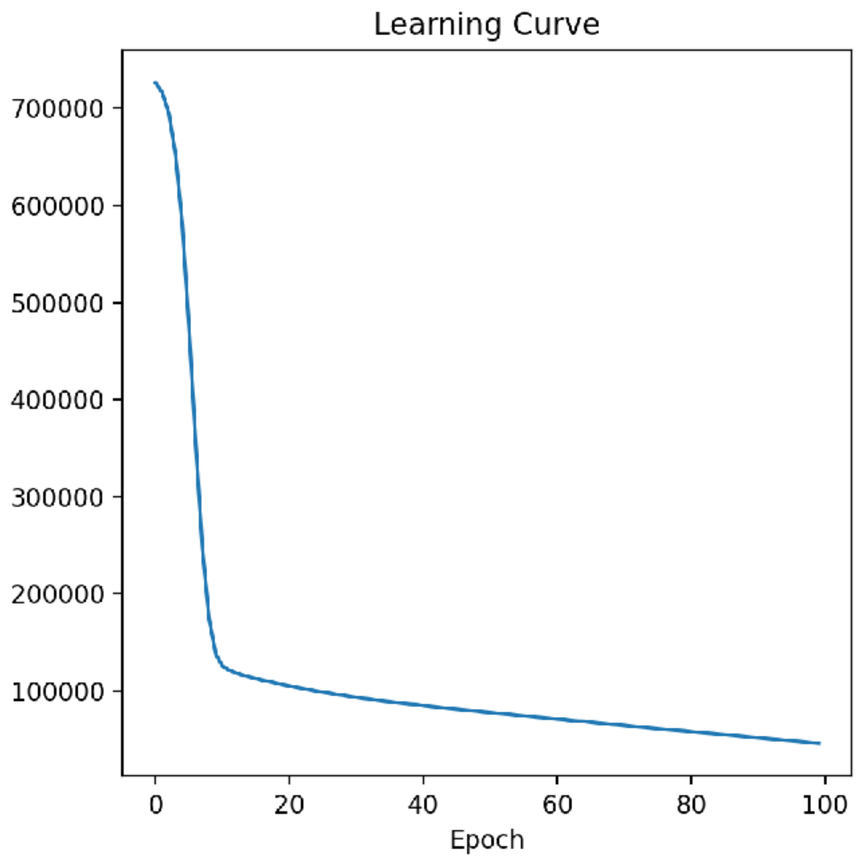
$$w^l \rightarrow w^l - \frac{\eta}{N} \sum_x \delta^{x,l} \left(a^{x,l-1}\right)^T$$

$$b^l \rightarrow b^l - \frac{\eta}{N} \sum_x \delta^{x,l}$$

Example







A FFNN learning the function $f(x,y) = ax^3 + by^3 + cxy$

NN architecture: 2-64-64-1, Training points: 1000, epochs = 100, ReLu tr. functions.

Language Models

Main phases:

- Tokenization
- Embedding
- Attention mechanism
- (Recurrent, FF, …) Neural network
- Prediction

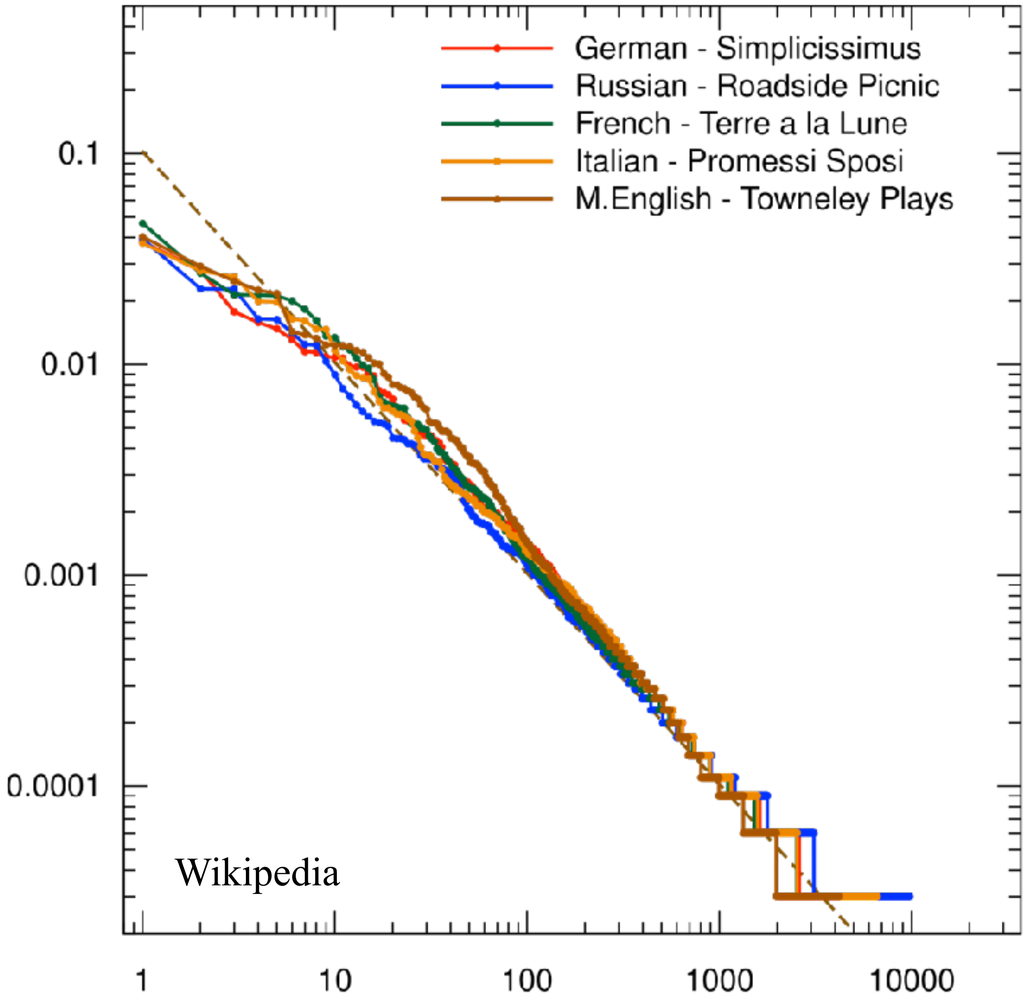
Different uses: e.g. translation, generation of content, …

Zipf's Law, or why LMs are a difficult problem

Zipf's Law (1935): the frequency of the n-th most frequent word is inversely proportional to n:

$$\text{Frequency} \propto \frac{1}{\text{Rank}}$$

The law has statistical nature and points to a "data sparsity" problem: since NNs require large training sets, some words will always appear with low statistics.



Tokenization

Tokenization is a fundamental preprocessing step in LLMs: the text is broken down into smaller units called tokens. Tokens can be words, subwords, characters, or other meaningful units. The tokenization method can significantly impact the performance of LLMs.
Example from the Natural Language Toolkit (https://www.nltk.org/):

```
import nltk

sentence = "Please tell me how to tokenize this sentence."
tokens = nltk.word_tokenize(sentence)
print(tokens)
```

OUTPUT:
['Please', 'tell', 'me', 'how', 'to', 'tokenize', 'this', 'sentence', '.']

Another interesting resource: https://github.com/huggingface/transformers

Embedding

There are many methods for associating a number/vector to a word/token.
The simplest way is just to associate an integer to each word of a vocabulary.
Better algorithms (like Word2Vec for example) use (shallow) neural networks for realising a very convenient mapping. An example from the gensim library:

```
from gensim.models import Word2Vec

sentences = [["the", "cat", "climbed", "on", "the", "roof"]]
model = Word2Vec(sentences, vector_size=10, min_count=1)
vector = model.wv['cat']  # vector representation of "cat"
print(vector)
```

The mapping is about linear in related words (“close” words have “close” vectors), so you can realise code like:

vec(“New England Patriots”)-vec(“New England”)+vec(“Seattle”)=~vec(“Seattle Seahawks”)

Attention Mechanism

The attention mechanism allows the model to focus on different parts of the input sequence when producing each element of the output sequence.

1) Assume we have an input sequence with n elements, each represented as a vector. Let's denote these vectors as x_1, x_2, …

2) For each input vector x_i the model computes three vectors:

- **Query (Q)**
- **Key (K)**
- **Value (V)**

These vectors are computed by multiplying the input vector x_i by three weight matrices: $Qi = W_Q x_i$, $K_i = W_K x_i$, $V_i = W_V x_i$.

Attention Mechanism

Meaning of the vectors:

You can look at something you would like to understand from a book: this is the query. You can look at "tags" in each paragraph of the book which could be relevant: these are the keys. You can assign an attention score (see next slide) to each paragraph for ranking how useful they were for you in understating the concept, or: how well the key matches the question. You then read the content (the "value") of the most relevant paragraphs, paying more attention to the ones with higher scores.

More concretely: the query is a word at hand, the keys are the other words and the value is assigned to each key. The value helps you in choosing the best next word.

Attention Mechanism

4) **Attention Scores**: For each query vector Q_i, the model computes a score against each key vector Kj in the input sequence. This score determines how much attention the model should pay to the corresponding value vector Vj when processing xi

The scores are computed using a similarity measure, often the dot product:

$$\text{score}(Q_i, K_j) = Q_i \cdot K_j$$

5) **Softmax**: The raw attention scores are then normalized using the `softmax` function to convert them into probabilities:

$$\alpha_{ij} = \frac{\exp(\text{score}(Q_i, K_j))}{\sum_{k=1}^{n} \exp(\text{score}(Q_i, K_k))}$$

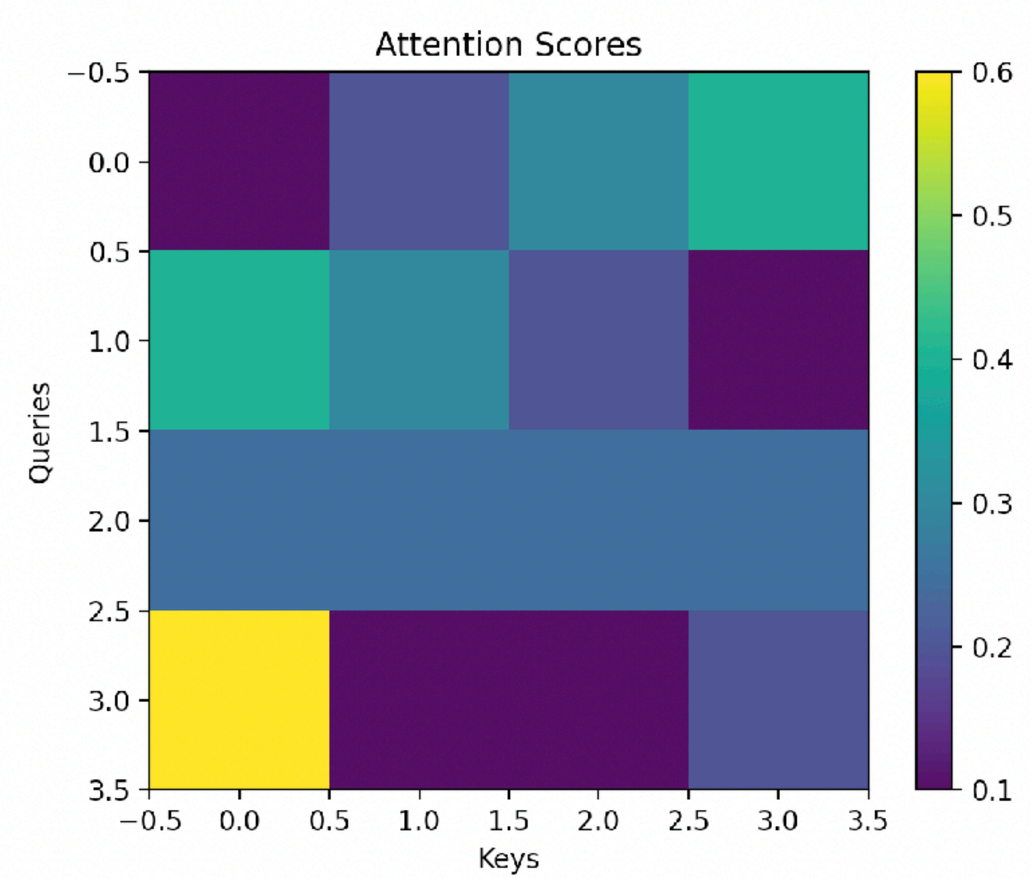
Represents the attention weight that the model assigns to the value vector V_j when producing the output for the input x_i .

Attention Mechanism

6) **Weighted Sum**: The output for each input x_i is computed as a weighted sum of the value vectors, where the weights are the attention scores:

$$\text{Attention}(Q_i, K, V) = \sum_{j=1}^{n} \alpha_{ij} V_j$$

This is the output of the attention mechanism algorithm.



Attention Mechanism: Example

Consider the sentence: “I love AI”.
Let’s take the simple encoding: I=(1,0,0) ; “love”=(0,1,0) ; “AI” = (0,0,1) , or:

$$x = \begin{pmatrix} I \\ love \\ AI \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Assume some starting value for the weight matrices (to learn later):

$$W_Q = W_K = W_V = \begin{bmatrix} 0.5 & 0.1 & 0.4 \\ 0.2 & 0.3 & 0.7 \\ 0.6 & 0.9 & 0.1 \end{bmatrix}$$

Attention Mechanism: Example

Calculate the Q,K,V vectors: $Q = XW_Q, \quad K = XW_K, \quad V = XW_V$

Calculate the score for the word “I” $\text{score}(Q_\text{I}, K_\text{I}) = \frac{Q_\text{I} \cdot K_\text{I}}{\sqrt{d_k}}$

Do the same for “love” and “AI”.

Softmax the scores (renormalize) and calculate the weighted sum for each word.

Attention Mechanism: another viewpoint

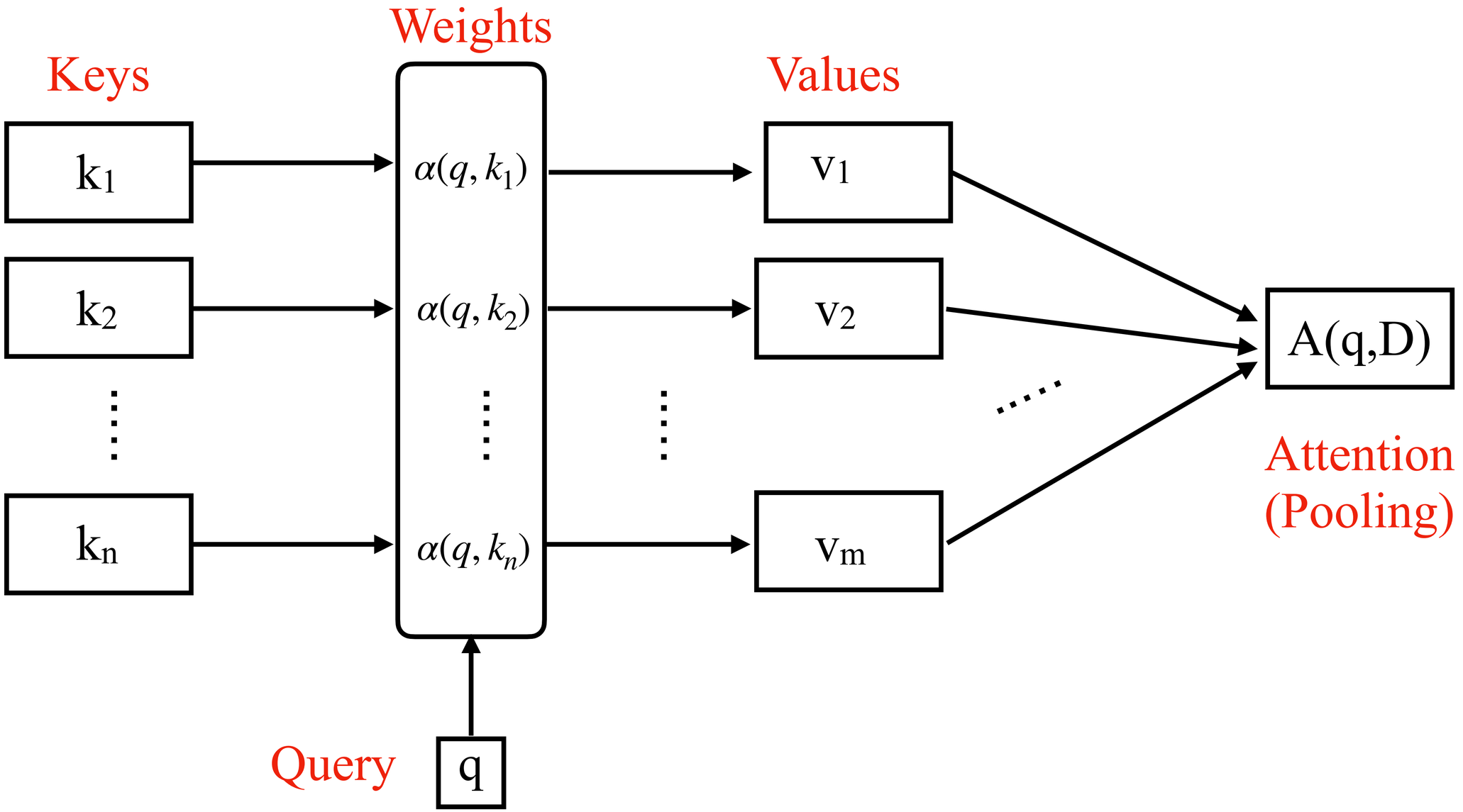
We have a database D made of couples $\{k_i, v_i\}$ = {key, value}.

Given a query q (a word/token), we can consult D and calculate:

$$\text{Attention}(q, D) = \sum_i \alpha(q, k_i) v_i$$

Sometimes this operation is called pooling.
The weights α are calculated with the softmax operation, which is always differentiable and with a non-vanishing gradient, thus well adapted for learning algorithms.
Note that if all the weights are zero but one, the attention operation is similar to a DB query.

Attention Mechanism visualised



Attention Mechanism: Another Example

Attention matrix for a NN translator

Attention scores for the translation

Query

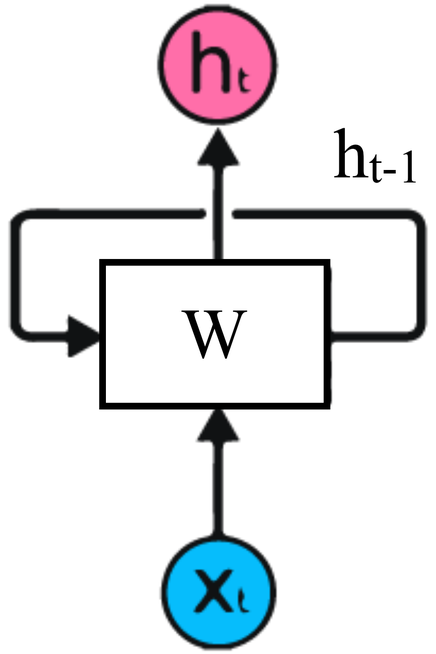
	Oggi	io	mangio	pizza
Heute				
esse				
ich				
pizza				

Digression: Recurrent Neural Networks

RNNs are a type of NNs well suited for processing sequential data. Unlike FFNNs, RNNs have connections forming directed cycles, allowing the persistence of a hidden state that captures information about previous elements in the sequence. RNNs are used in tasks like language modelling, automatic translation, and time series prediction.

The output of an RNN layer depends from previous data, e.g.:

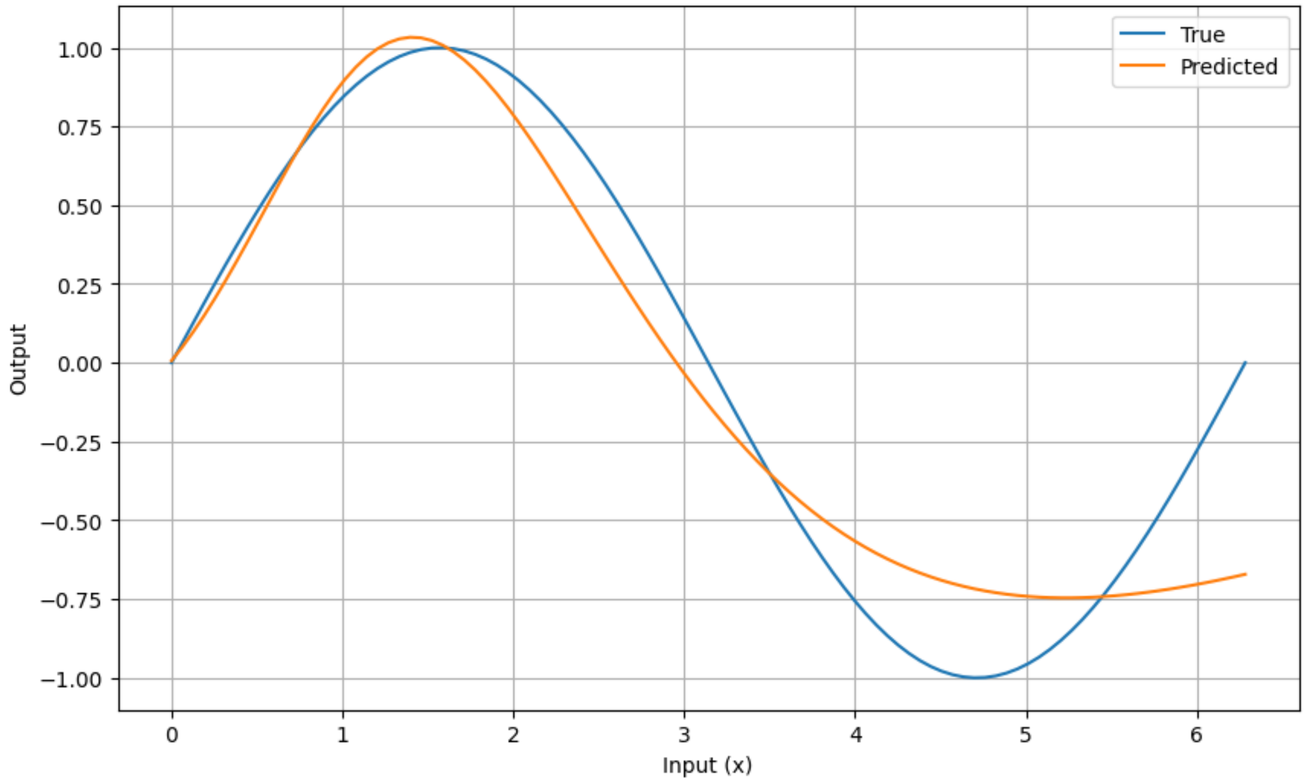
$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$



Recurrent Neural Networks: Simple Example

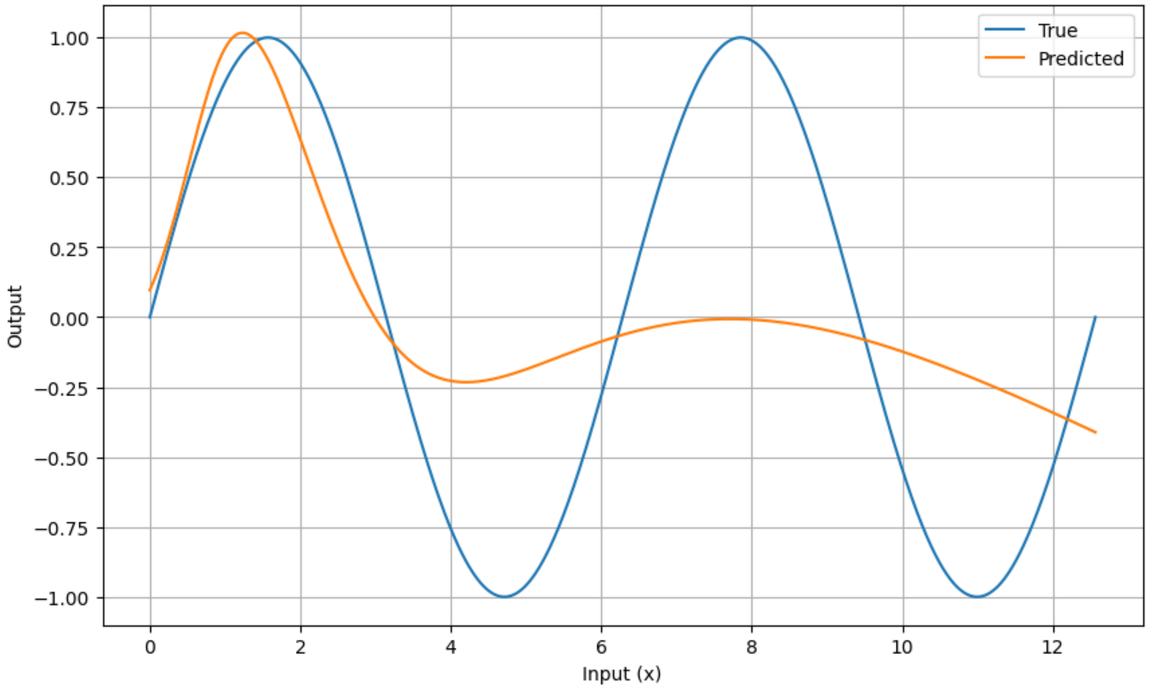
Time sequence: sinus function.
1000 epochs training, 50 neurons.

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# Generate data: sinus function
data = np.linspace(0, 2 * np.pi, 100)
target = np.sin(data)

# Reshape data for RNN input
data = data[:, np.newaxis]
target = target[:, np.newaxis]

# Define the RNN model
model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(50, input_shape=(1, 1)),
    tf.keras.layers.Dense(1)
])

model.compile(optimizer='adam', loss='mse')
history = model.fit(data, target, epochs=1000, verbose=1)
predicted = model.predict(data)
```

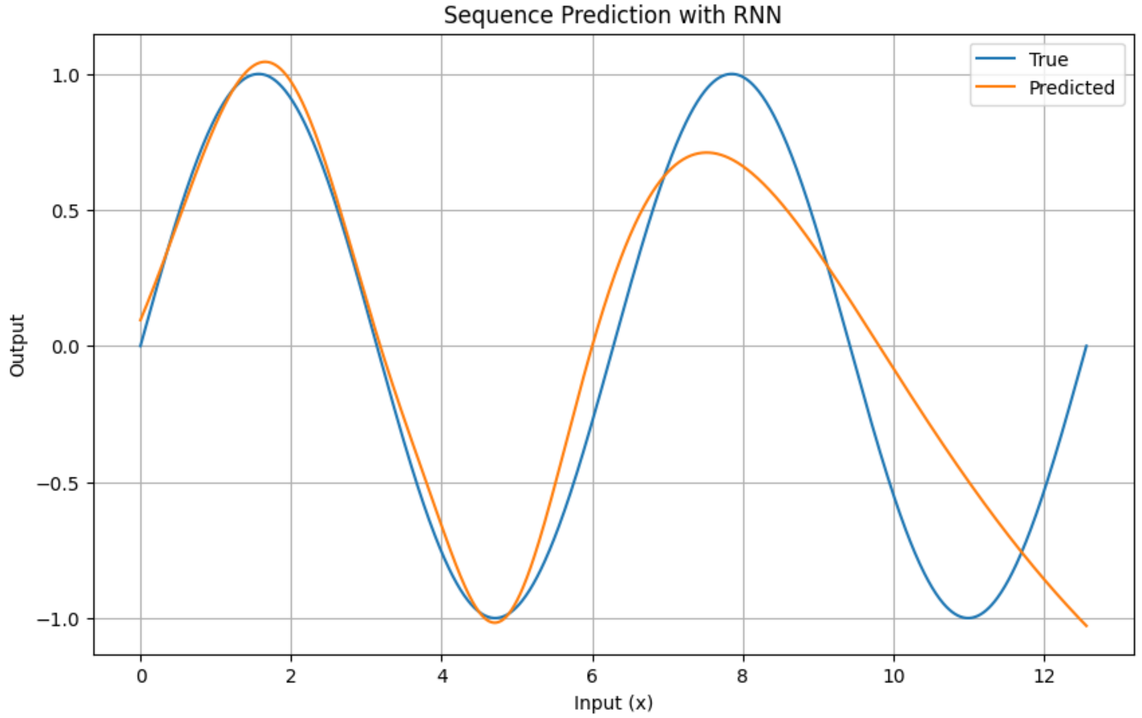


Recurrent Neural Networks: Simple Example

50 Epochs



1000 Epochs



RNNs converge slowly in general and suffer the "vanishing gradient" problem.

The Transformer Architecture

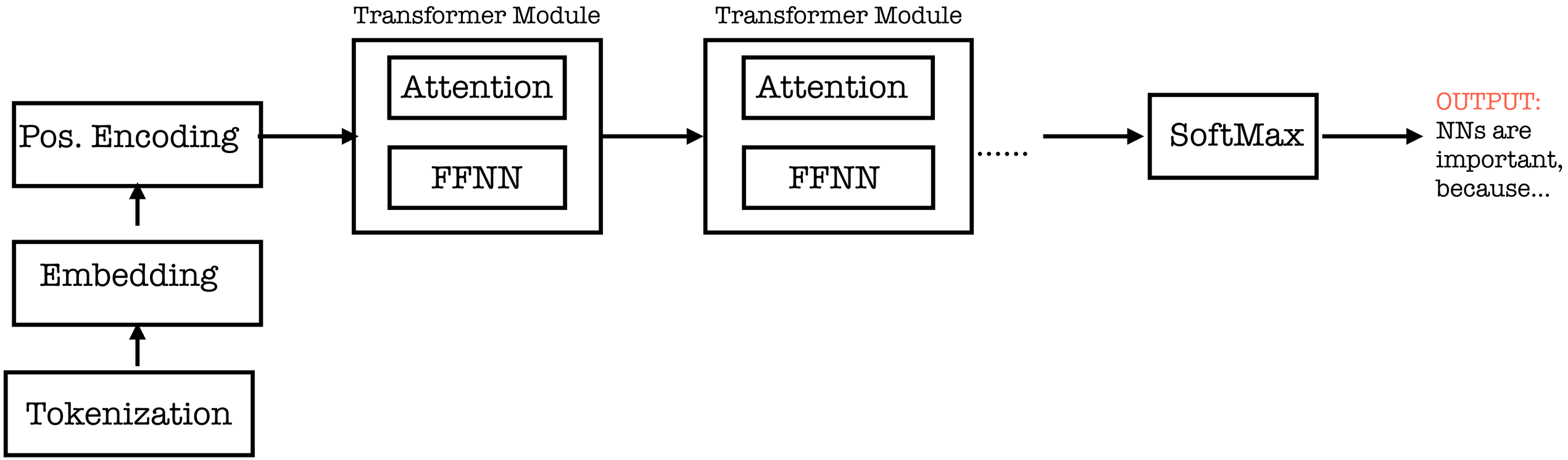
Self-attention mechanism: the attention is applied within a sentence. For each word in the input sequence, self-attention computes attention scores based on how much each word contributes to the representation of the current word.

Attention can "average out" some important information. This problem is alleviated with multi-headed attention: the sentence is divided in M pieces and attention is applied to each one. The results are concatenated instead of summed.

Positional encoding is added for imitating what a RNN can do (Transformers do not use RNNs).

A FFNN is used after attention, mostly with ReLu layers and residual connections for avoiding the vanishing gradient problem.

The Transformer Architecture



INPUT:
Explain me how a NN works

LLMs Summary (1)

Language models require many building blocks: Tokenization, Embedding, Attention, (recurrent) neural networks.

The exact architectures of popular LLMs are not disclosed (also training set and fine-tuning methods).
A simple RNN is not enough: RNNs tend to give more weight to the present data and forget about previous (likely important) information: more complex architectures and attention mechanisms are required: Transformers (FFNN-based).

Modern LLMs use variants of the Transformer model.

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

LLMs Summary (2)

Note how from the RNN paradigm, LLMs shifted to FFNNs+Attention.
RNNs constituted the first attempt at solving language-based problems, but they ran into different issues:

- Process only 1 data at a time: hard to implement parallelism,
- Difficult to remember initial words,
- Difficult to relate distant words
- Last two problems only partially solved by LSTM networks.

Transformers solved the attention and speed problems at the same time.