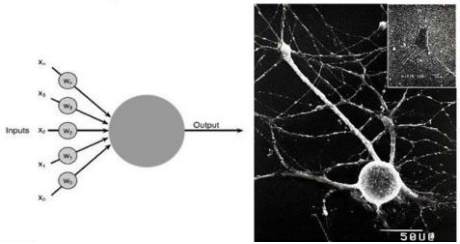
An Introduction to Neural Networks

Satchit Chatterji
MSc Artificial Intelligence
University of Amsterdam

satchit.chatterji@gmail.com

Adapted from
ISOTDAQ 2022

you vs the guy she told you not to worry about:



Artificial vs Biological NNs

ANNs initially inspired by the brain:

Alexander Bain (1873), William James (1890)

Electrical connections/flow of neurons result in thought and movement

McColloch & Pitts (1943)

Modern mathematical “artificial” NN models (not the only neural network model!)

Rosenblatt (1958)

Description of the *perceptron*

Rumelhart, Hinton & Williams (1986)

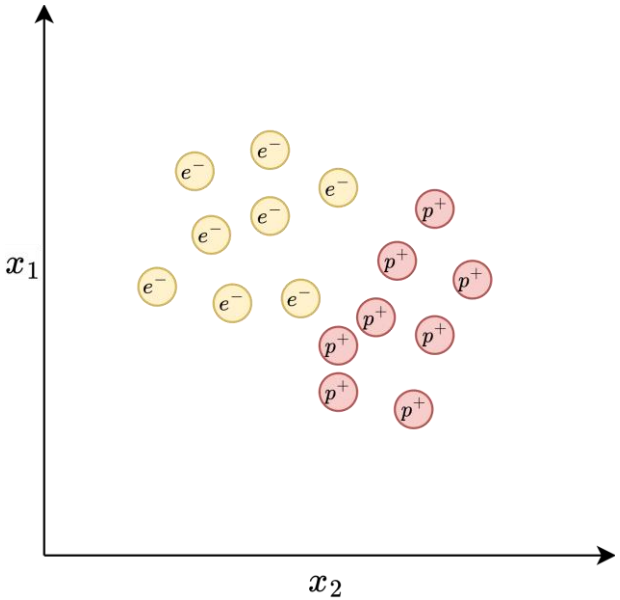
Multi-layer perceptrons and error backpropagation (learning principle)

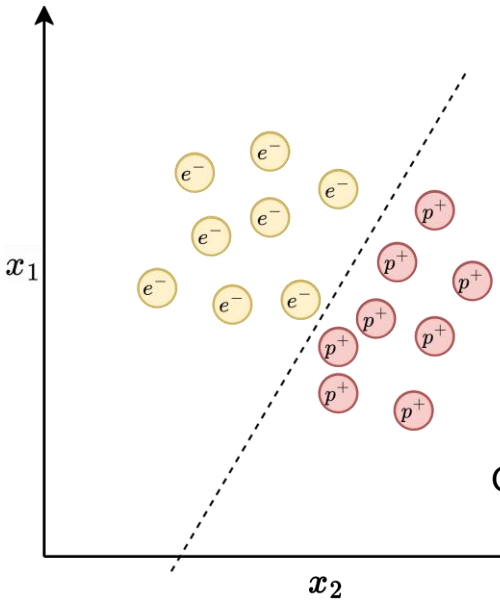
Modern day:

- ANNs used *everywhere* for *everything!*
- Simplified, abstracted version of “synaptically”-connected “neurons”
- Biologically implausible

Image source:
linkedin.com/company/deeplearningai

Building a Neural Network From Scratch (intuitively)


x_1
x_2
e^-
e^-
e^-
e^-
e^-
e^-
e^-
e^-
p^+
p^+
p^+
p^+
p^+
p^+
p^+



Classes are *"linearly separable"*

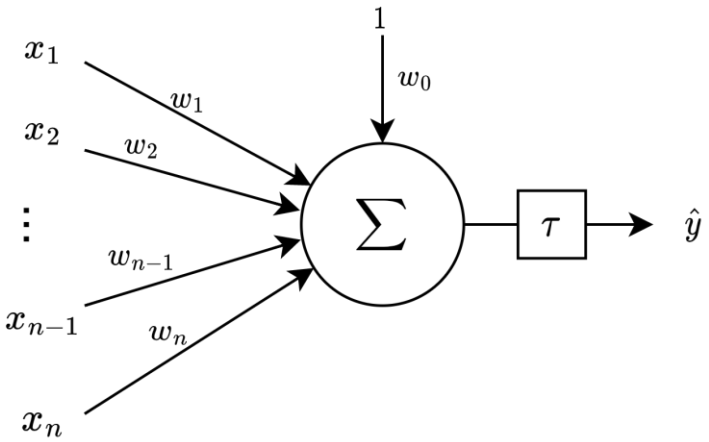
$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n$$

$w_i \leftarrow$ Coefficients
$x_i \leftarrow$ Variables

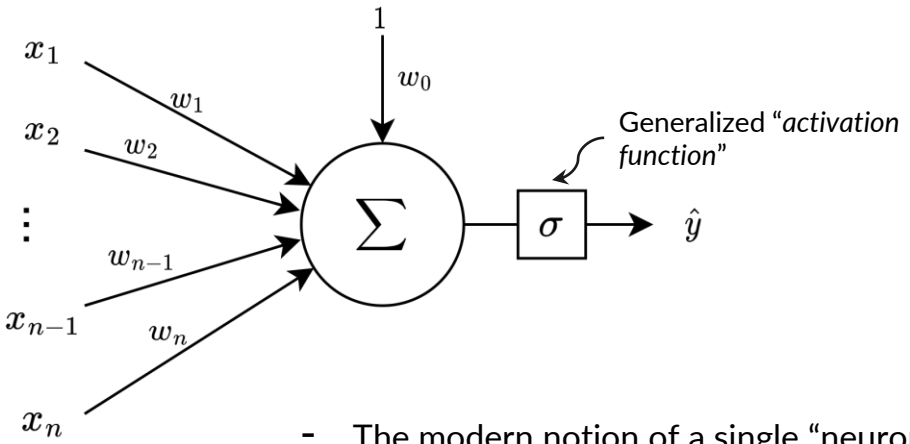
$$\hat{y} = \tau(w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n)$$

$$\tau(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

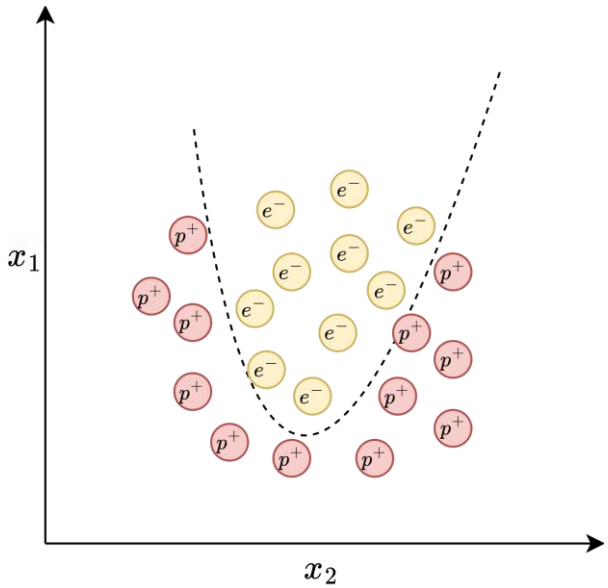
$$\hat{y} = \tau(w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n)$$

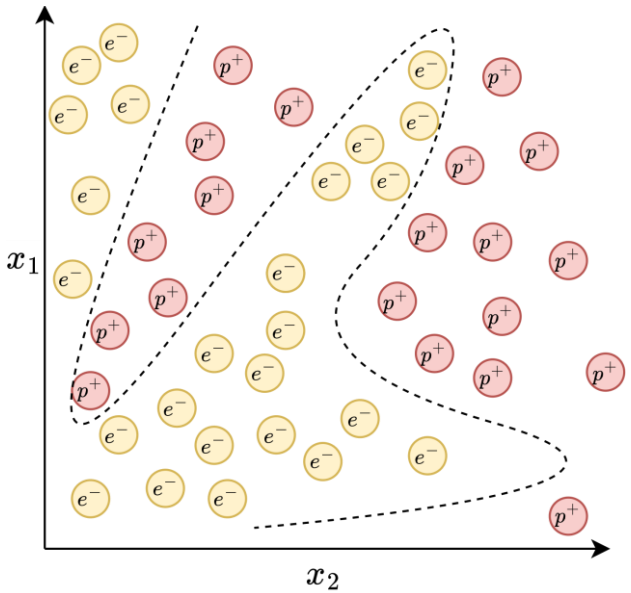


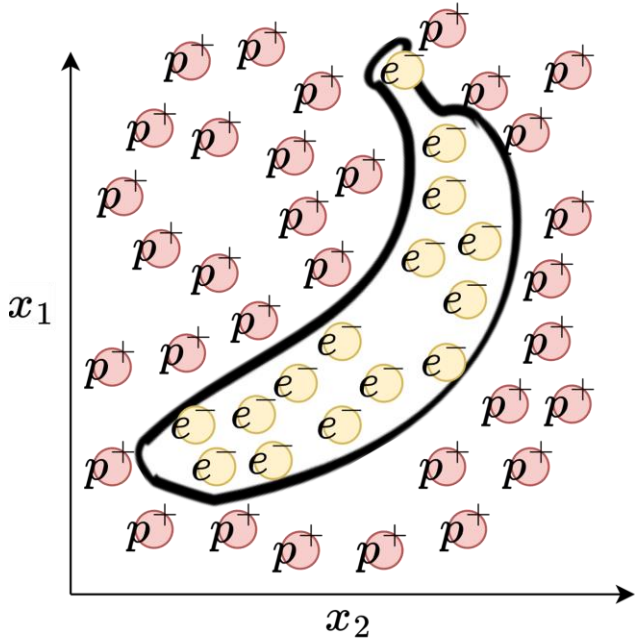
The “Perceptron”

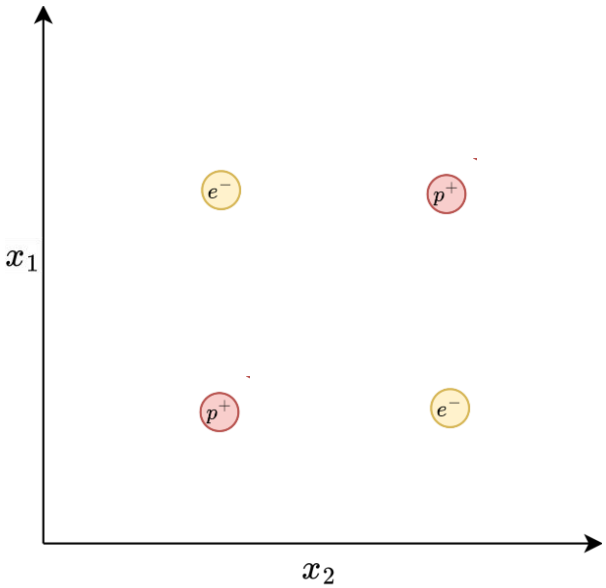


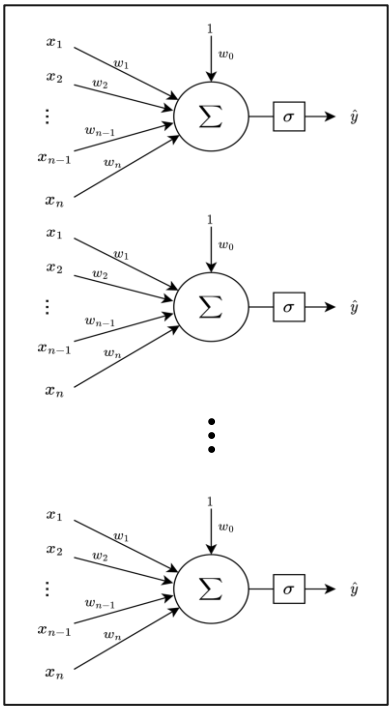
- The modern notion of a single “neuron”
- BUT: Only works on linearly separable classes

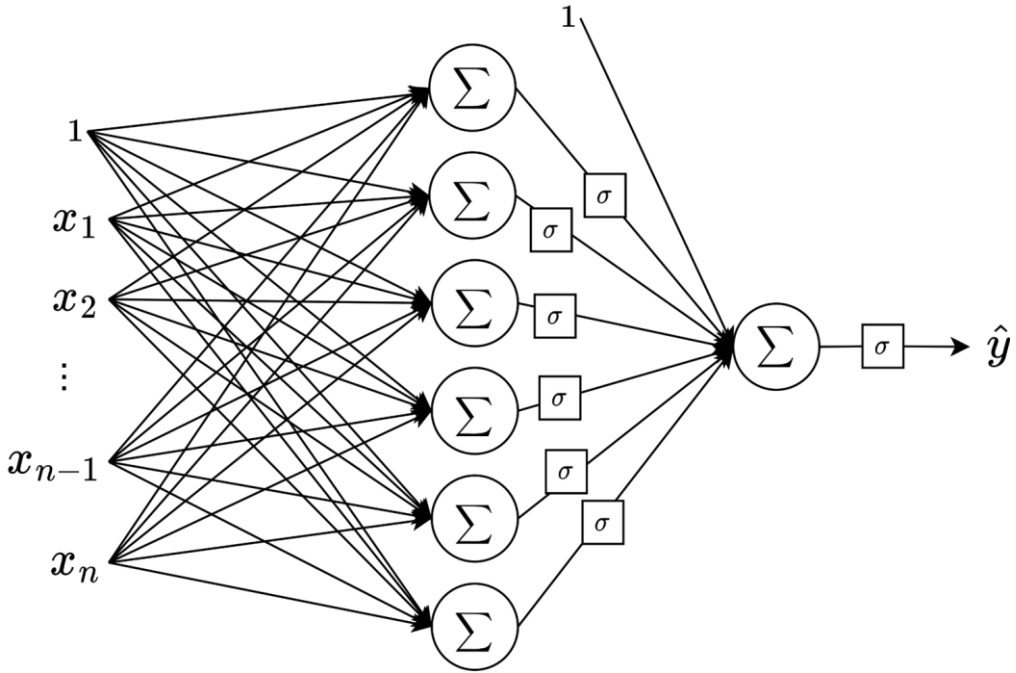

x_1
x_2
e^-
p^+


x_1
x_2

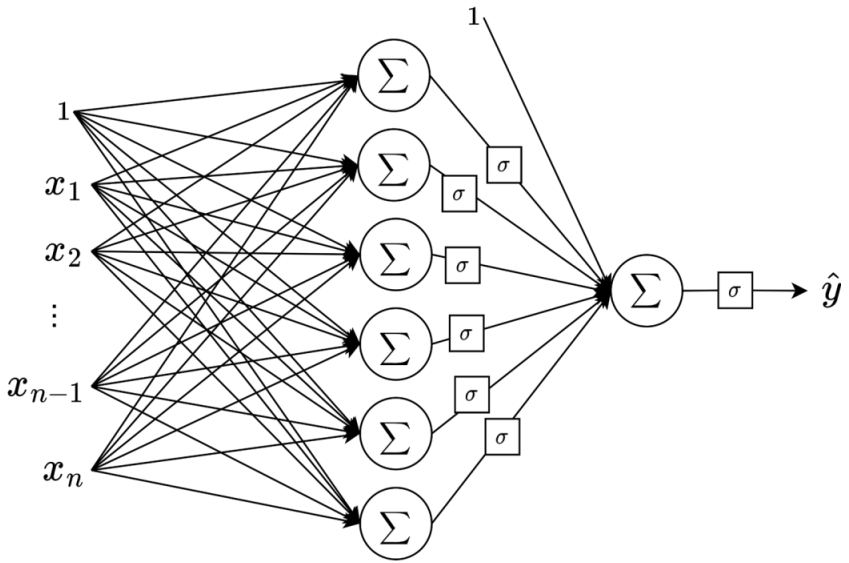

x_1
x_2


e^-
p^+
x_1
p^+
e^-
x_2


x_1
x_2
x_{n-1}
x_n
w_1
w_2
w_{n-1}
w_n
1
w_0
Σ
σ
$\hat{y}$


1
x_1
x_2
$\vdots$
x_{n-1}
x_n
1
σ
$\hat{y}$

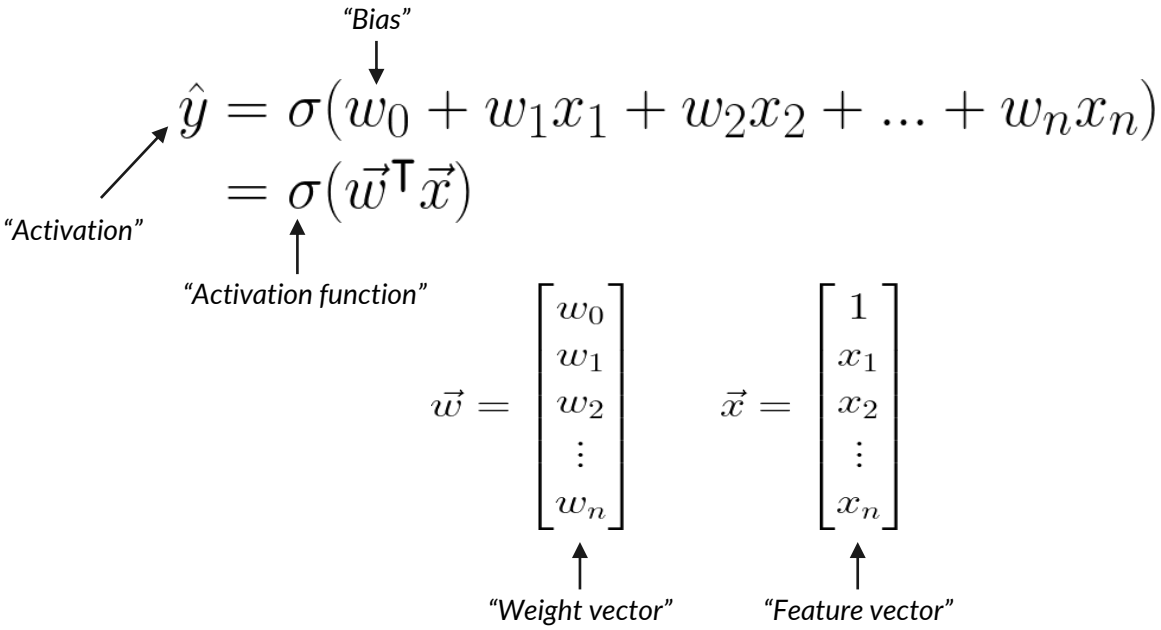
Multi-layer Perceptrons (MLPs)



$$\hat{y} = \sigma(w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n)$$

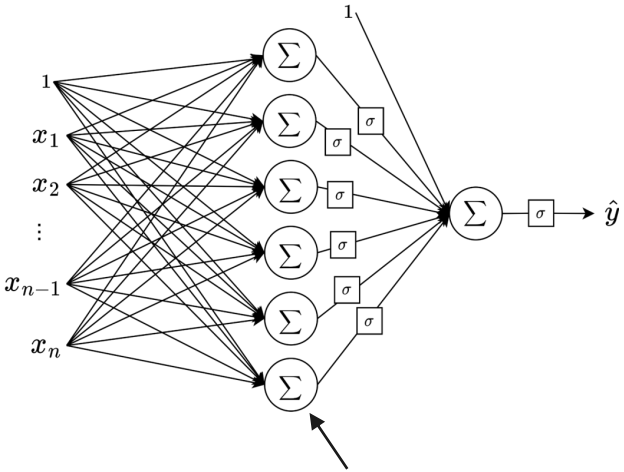
$$\begin{aligned}\hat{y} &= \sigma(w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n) \\ &= \sigma(\vec{w}^\intercal \vec{x})\end{aligned}$$

$$\vec{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \qquad \vec{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$


"Bias"
$\hat{y} = \sigma(w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n)$
$= \sigma(\vec{w}^\intercal \vec{x})$
"Activation"
"Activation function"
$\vec{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$
$\vec{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$
"Weight vector"
"Feature vector"

Activation/
output
of neuron *k*

$$o^k = \begin{bmatrix} w_0^k & w_1^k & \cdots & w_n^k \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$



***K** "hidden"* neurons
in layer **L**

$$o^1 = \begin{bmatrix} w_0^1 & w_1^1 & \cdots & w_n^1 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

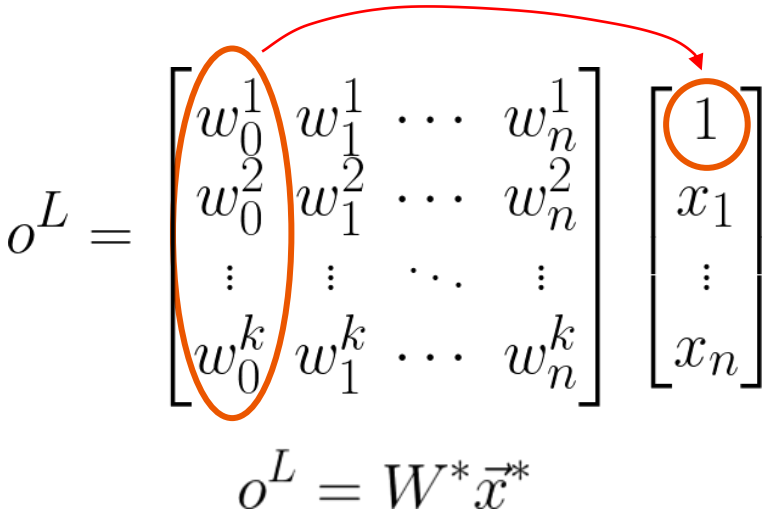
$$o^1 = \begin{bmatrix} w_0^1 & w_1^1 & \cdots & w_n^1 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$$o^2 = \begin{bmatrix} w_0^2 & w_1^2 & \cdots & w_n^2 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$$
\begin{aligned}
o^1 &= \begin{bmatrix} w_0^1 & w_1^1 & \cdots & w_n^1 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \\
o^2 &= \begin{bmatrix} w_0^2 & w_1^2 & \cdots & w_n^2 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \\
&\vdots \\
o^k &= \begin{bmatrix} w_0^k & w_1^k & \cdots & w_n^k \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}
\end{aligned}
$$

$$o^L = \begin{bmatrix} w_0^1 & w_1^1 & \cdots & w_n^1 \\ w_0^2 & w_1^2 & \cdots & w_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ w_0^k & w_1^k & \cdots & w_n^k \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$$o^L = W^* \vec{x}^*$$


$o^L = \begin{bmatrix} w_0^1 & w_1^1 & \cdots & w_n^1 \\ w_0^2 & w_1^2 & \cdots & w_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ w_0^k & w_1^k & \cdots & w_n^k \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$
$o^L = W^* \vec{x}^*$

$$o^L = \begin{bmatrix} w_1^1 & w_2^1 & \cdots & w_n^1 \\ w_1^2 & w_2^2 & \cdots & w_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ w_1^k & w_2^k & \cdots & w_n^k \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} w_0^1 \\ w_0^2 \\ \vdots \\ w_0^k \end{bmatrix}$$

$$o^L = W\vec{x} + \vec{b}$$

Most common way of writing out the activation of a layer of an MLP

$$o^L = W\vec{x} + \vec{b}$$

$$o^L = W\vec{x} + \vec{b}$$

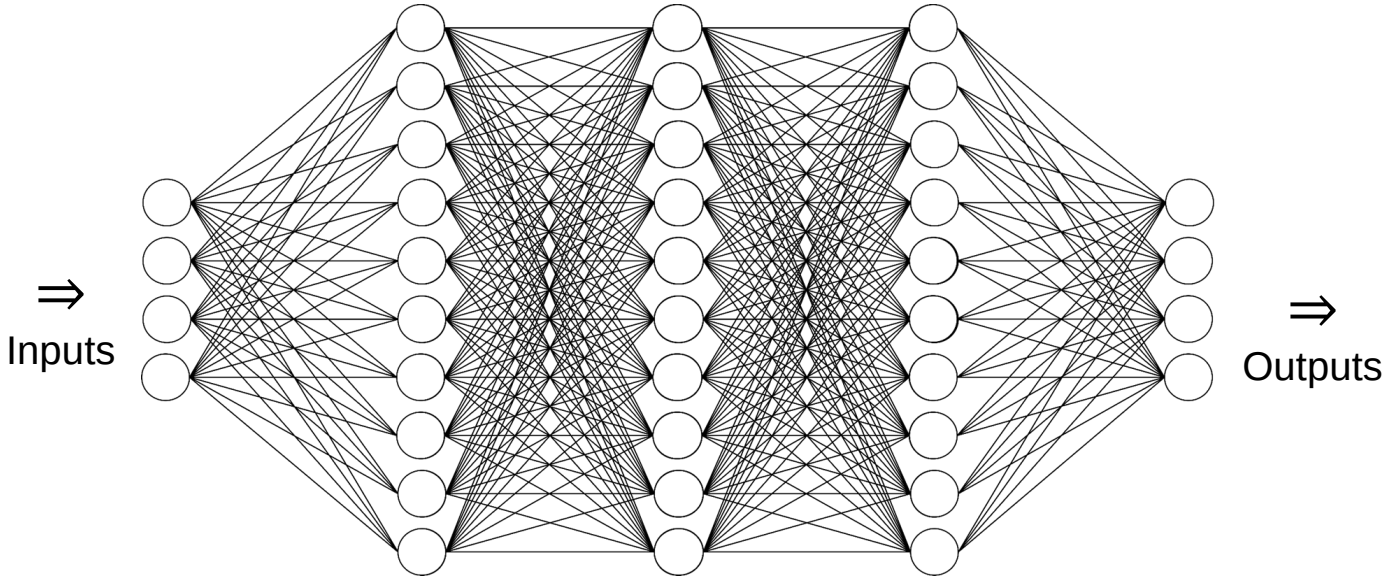
$$\hat{y} = \sigma(w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n)$$

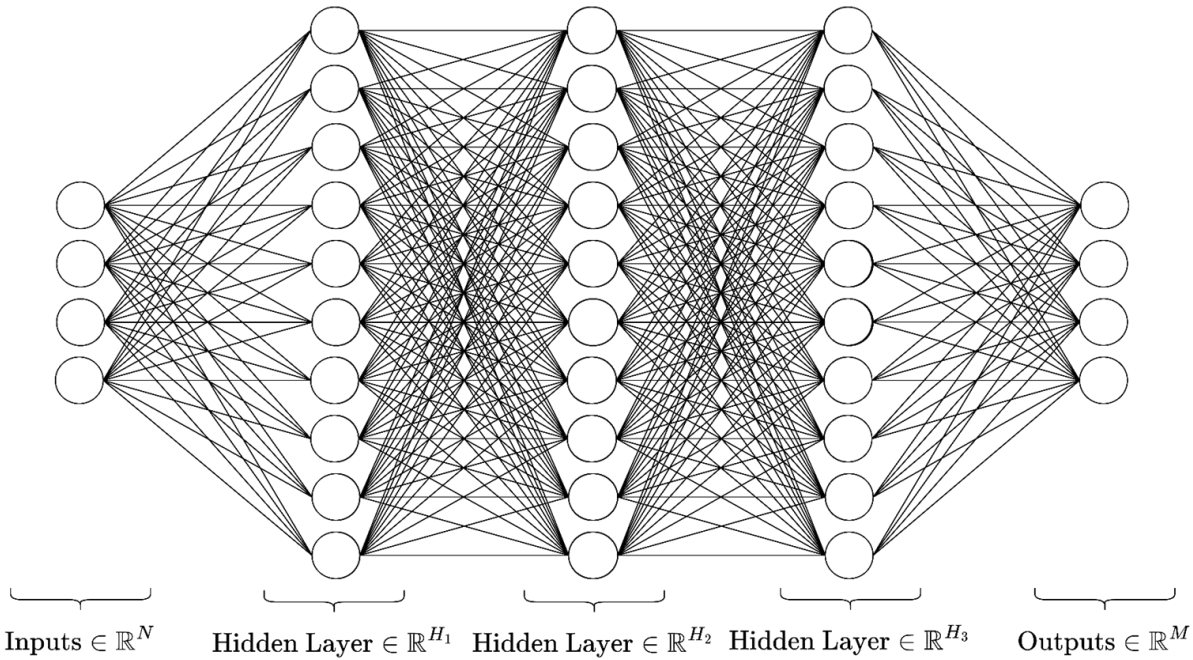
$$o^L = W\vec{x} + \vec{b}$$

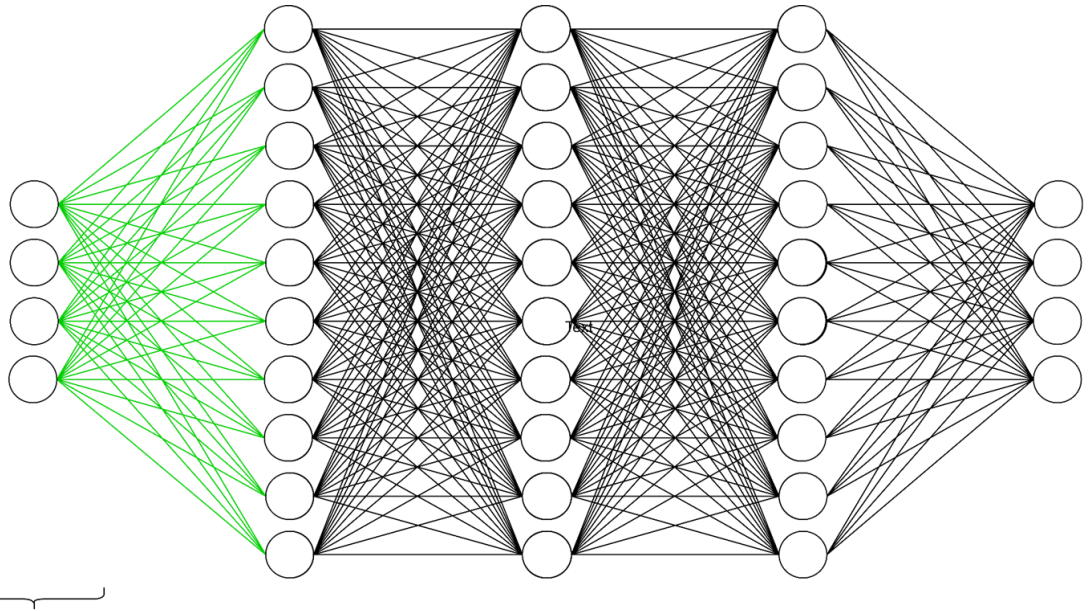
$$\hat{y} = \sigma(w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n)$$

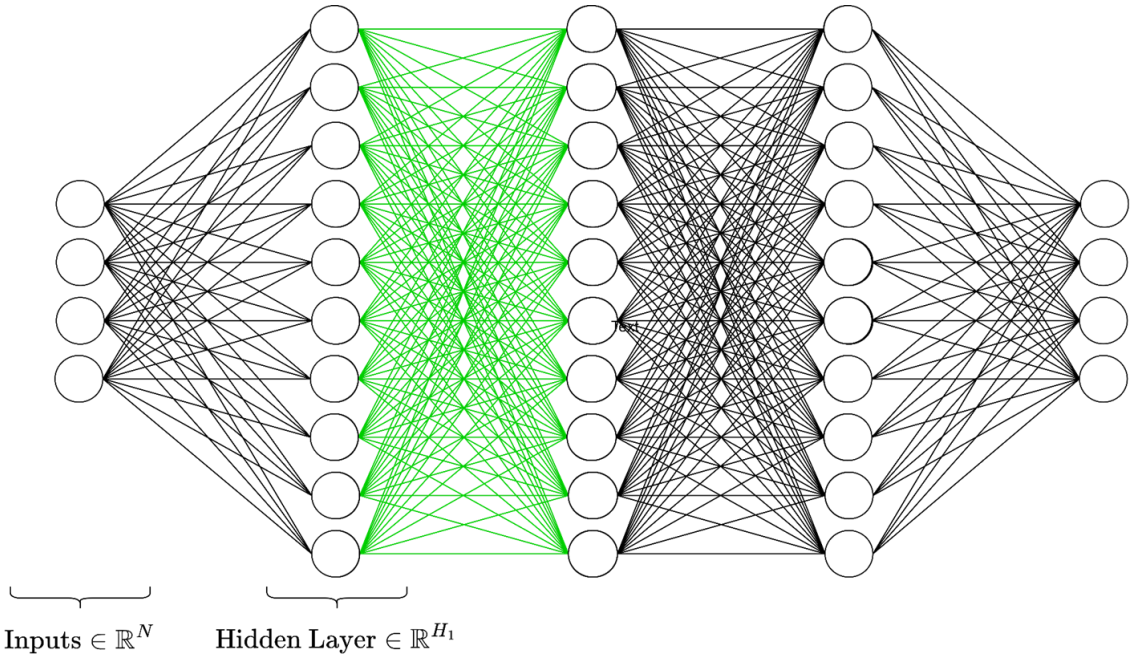
$$o = \sigma(W x^{in} + b)$$

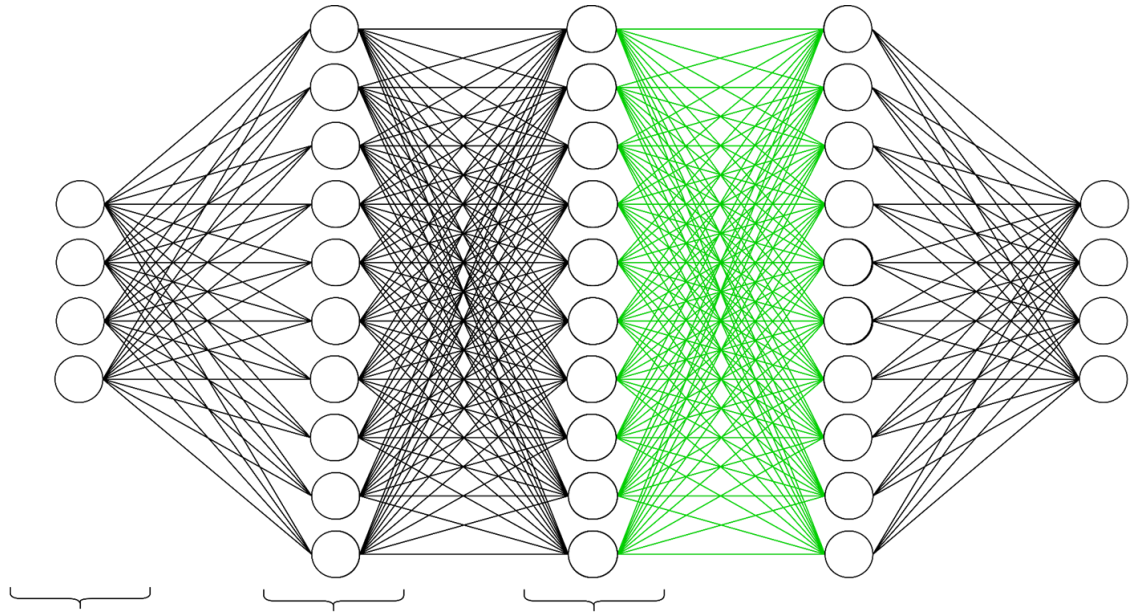
The **output** of each layer is the product of its **weight matrix** and the **input vector** plus its **bias vector**, all wrapped in a **non-linear activation function**.

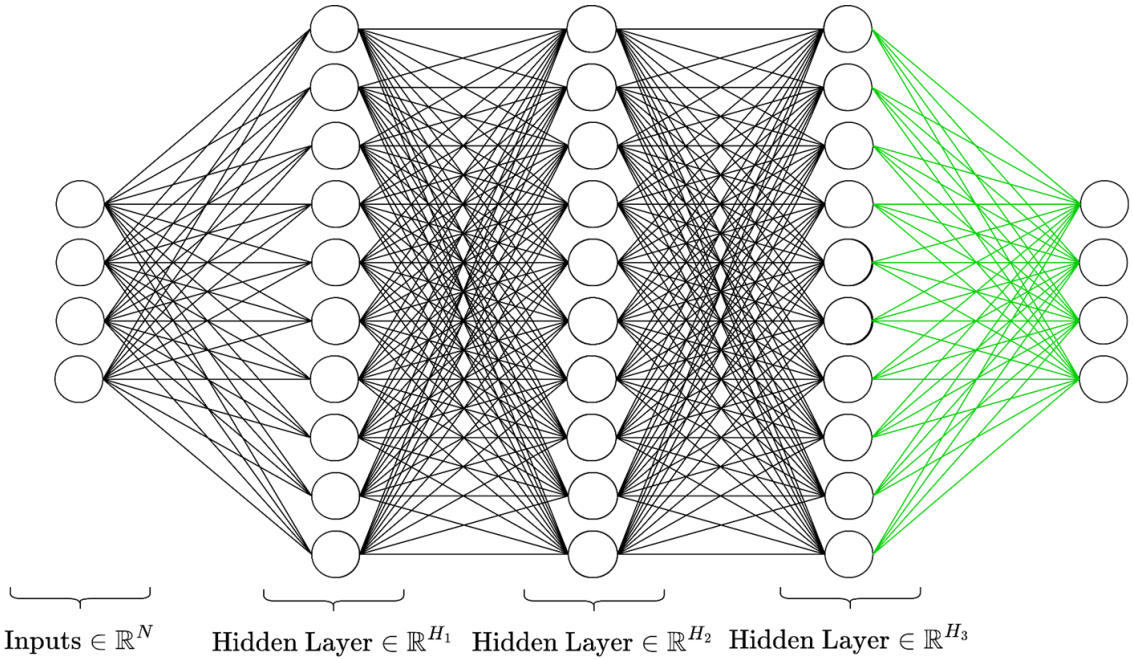

⇒
Inputs
⇒
Outputs

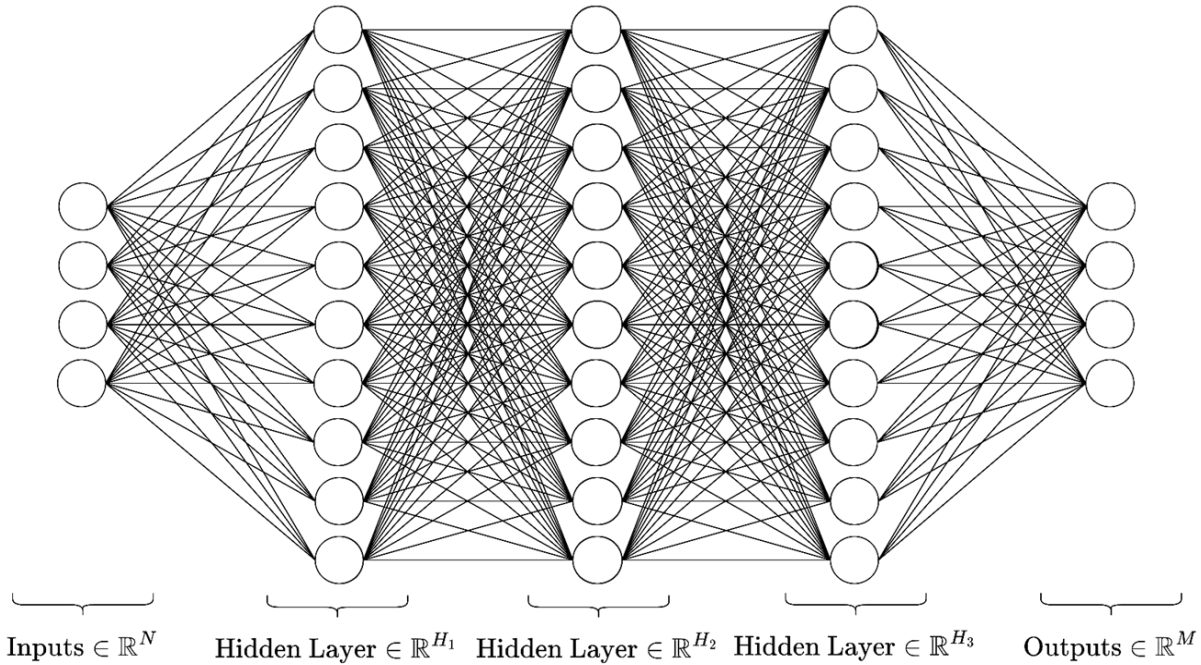

Inputs $\in \mathbb{R}^N$
Hidden Layer $\in \mathbb{R}^{H_1}$
Hidden Layer $\in \mathbb{R}^{H_2}$
Hidden Layer $\in \mathbb{R}^{H_3}$
Outputs $\in \mathbb{R}^M$

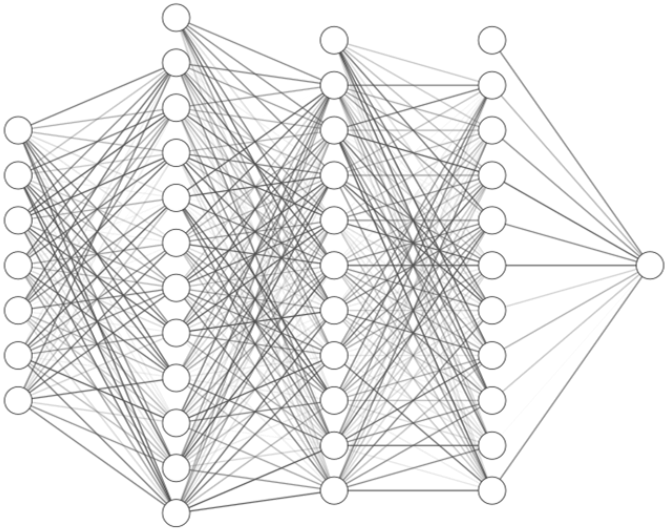

tanh
Inputs $\in \mathbb{R}^N$


Inputs $\in \mathbb{R}^N$
Hidden Layer $\in \mathbb{R}^{H_1}$


Inputs $\in \mathbb{R}^N$
Hidden Layer $\in \mathbb{R}^{H_1}$
Hidden Layer $\in \mathbb{R}^{H_2}$


Inputs $\in \mathbb{R}^N$
Hidden Layer $\in \mathbb{R}^{H_1}$
Hidden Layer $\in \mathbb{R}^{H_2}$
Hidden Layer $\in \mathbb{R}^{H_3}$


Inputs $\in \mathbb{R}^N$
Hidden Layer $\in \mathbb{R}^{H_1}$
Hidden Layer $\in \mathbb{R}^{H_2}$
Hidden Layer $\in \mathbb{R}^{H_3}$
Outputs $\in \mathbb{R}^M$



A multi-layer perceptron is a series of affine transformations of an input vector, each of which is wrapped in a non-linear activation function.

$$\mathcal{N} : \mathbb{R}^N \to \mathbb{R}^M$$
$$N, M \in \mathbb{N}$$

(*Translation:* an MLP is a fancy function)

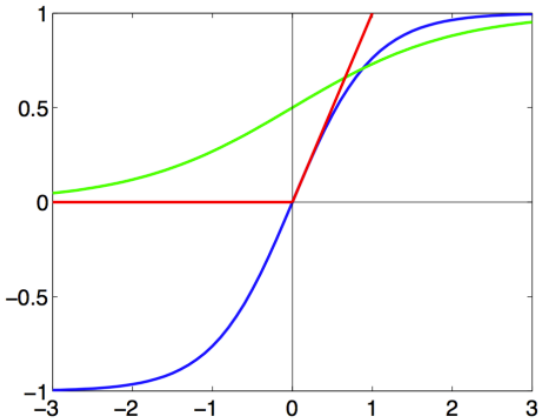
A Note on Nonlinearity

A Note on Nonlinearity

Without a non-linear activation function, a series of linear transformations would result in just a linear transformation of the input to the output.

We would still be stuck in the land of linear separability!

Common nonlinear functions



Sigmoid: $\sigma(x) = \dfrac{1}{1+e^{-x}}$

Hyperbolic tangent: $\tanh(x) = \dfrac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

Rectified Linear Unit:

$\mathrm{ReLU}(x) = max(0, x)$

Backpropagation

Implementing learning: Gradient descent

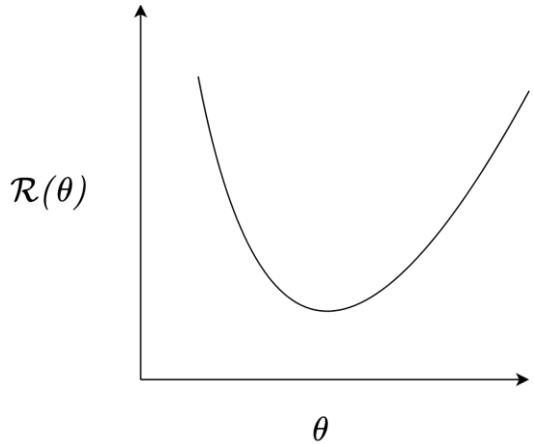
Given:

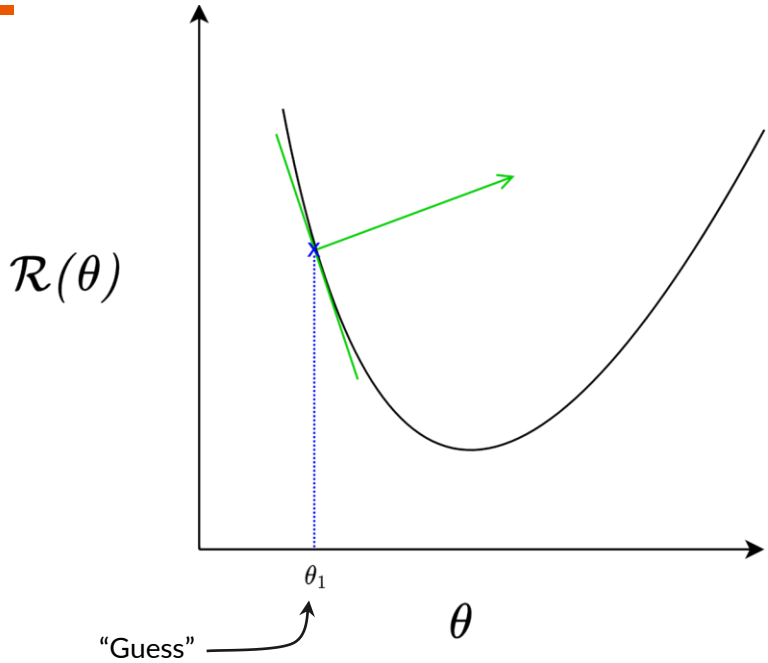
- Family of parameters Θ (e.g. possible weights of a NN)
- Differentiable risk function $\mathcal{R}(\theta)$

Goal: $$\theta_{opt} = \underset{\theta \in \Theta}{\operatorname{argmin}}\ \mathcal{R}(\theta)$$

Backprop: A specific implementation of GD, where errors are passed backwards through layers

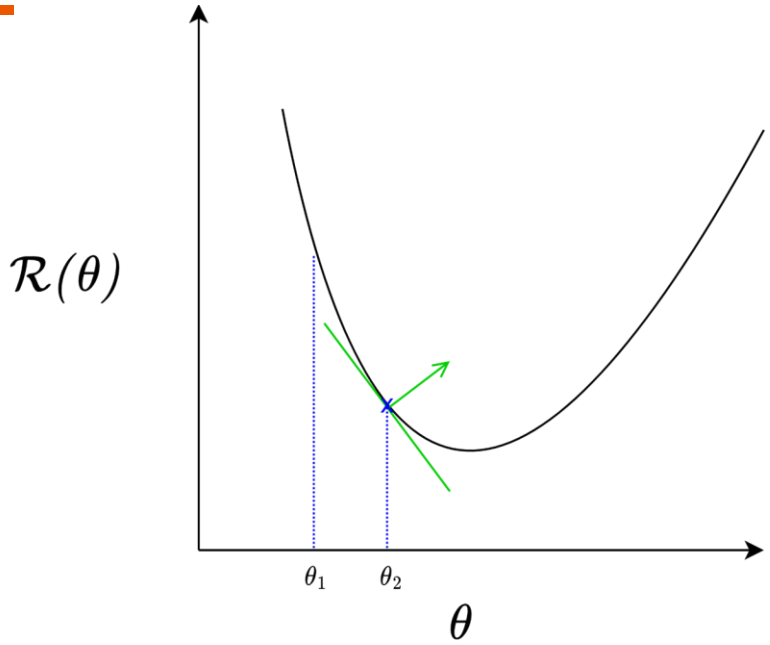
Gradient descent



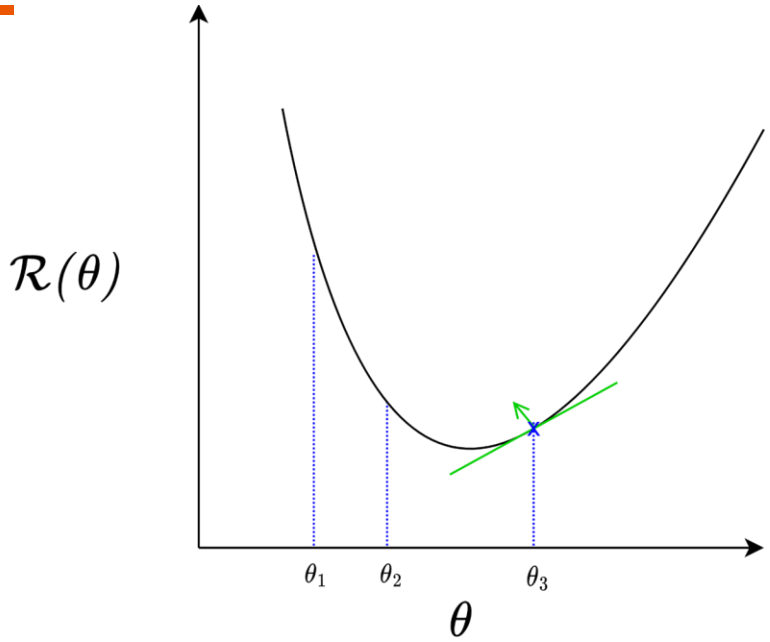


"Learning rate"

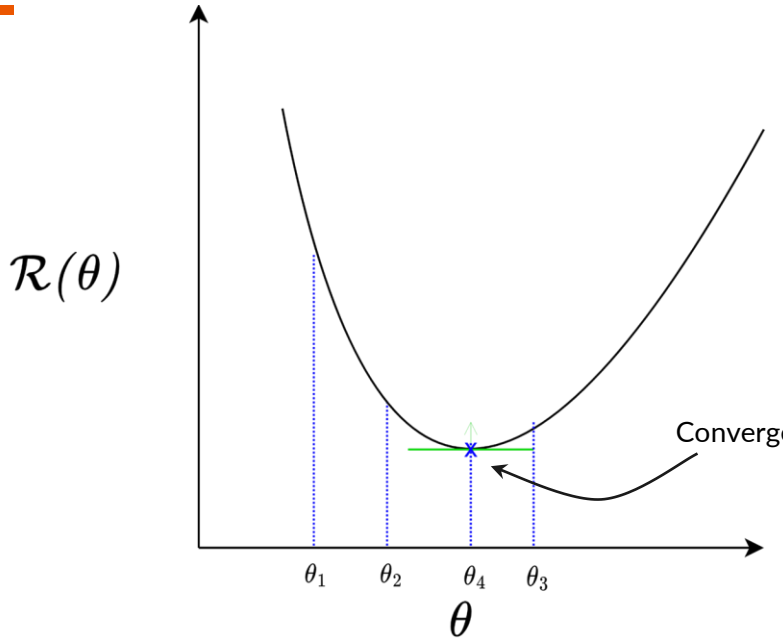
$$\theta_i = \theta_i - \alpha \frac{\partial}{\partial \theta_i} \mathcal{R}(\theta_i)$$



$$\theta_i = \theta_i - \alpha \frac{\partial}{\partial \theta_i} \mathcal{R}(\theta_i)$$



$$\theta_i = \theta_i - \alpha \frac{\partial}{\partial \theta_i} \mathcal{R}(\theta_i)$$



$$\theta_i = \theta_i - \alpha \frac{\partial}{\partial \theta_i} \mathcal{R}(\theta_i)$$

Converges to local minima

Optimizers

Stochastic/Mini-batch GD: *Speed improvement*!

Perform backprop on *batches* of training samples instead of all at once

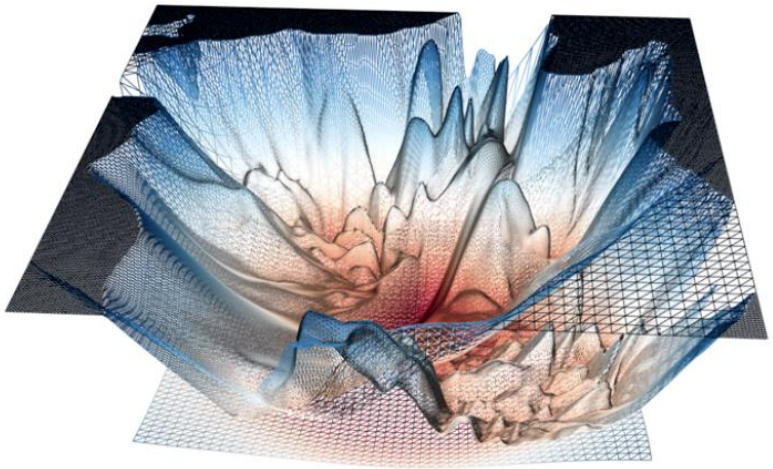
- Reduces the number of expensive backward passes

Optimizers determine exactly how gradient descent is implemented

- Stochastic Gradient Descent (most common)
- Adam
- RMSProp

A “real” loss landscape:

- Many (many many) local minima
- Saddle points



http://www.telesens.co/2019/01/16/neural-network-loss-visualization/

Loss functions

Depends on the task!

Mean Squared Error

Used for e.g. regression tasks

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Cross Entropy

Used for e.g. classification tasks

$$CE = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i)$$

Define your own!

Note: Must be differentiable for gradient descent based methods

What NNs *can* and *can't* do

Universal Approximation Theorem

Theorem (schematic). *Let $\mathcal{F}$ be a certain class of functions $f : \mathbb{R}^K \to \mathbb{R}^M$. Then for any $f \in \mathcal{F}$ and any $\varepsilon > 0$ there exists an multilayer perceptron $\mathcal{N}$ with one hidden layer such that $\|f - \mathcal{N}\| < \varepsilon$.*

⇒ We can approximate *any* function we want with a *one*-layer MLP!
More effective with more layers than just one ("deeper" networks)
Easier said than done in practice

Schematic borrowed from Jaeger, H. (2022) *Neural Networks* Lecture Notes, https://www.ai.rug.nl/minds/uploads/LN_NN_RUG.pdf

Collection of proofs:
https://ai.stackexchange.com/questions/13317/where-can-i-find-the-proof-of-the-universal-approximation-theorem

Why function approximation?

- Pattern recognition
- Time series prediction
- Denoising, restoration
- Pattern completion
- Data compression
- Process control
- Classification and regression
- Image and speech recognition
- Natural language processing
- Recommender systems
- Anomaly detection
- Robotics and control systems
- Generative modeling
- Reinforcement learning
- Drug discovery and bioinformatics
- ...

Where NNs thrive

> **Statistical/correlation inference** needed

> There exists **a lot of good quality (labelled) training data**

> **Parallelizable** training and deployment

> Tasks **without expansion** (input-output fixed)

> **Specialized tasks**

> Good **in-range performance** IRL

https://lasp.colorado.edu/home/minxss/2016/07/12/minimum-mission-success-criteria-met/

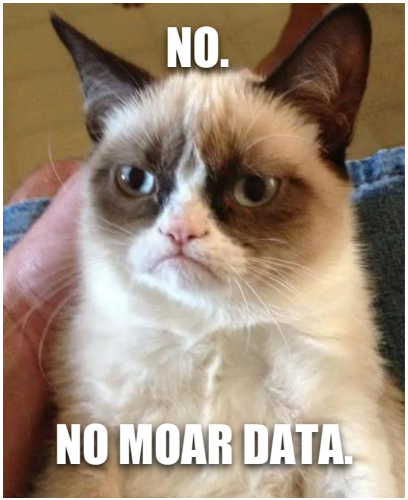
Limits of NNs

> **No causal relations** possible (yet)

> **Very data hungry** - "*Garbage in, garbage out*"

> **Often expensive to train** (money & time)

> **Nonextensible** and **specialized** to a range and task
  - One more ouptut → finetune/retrain the entire network
  - Undefined behaviour on out-of-domain test examples



https://knowyourmeme.com/memes/grumpy-cat

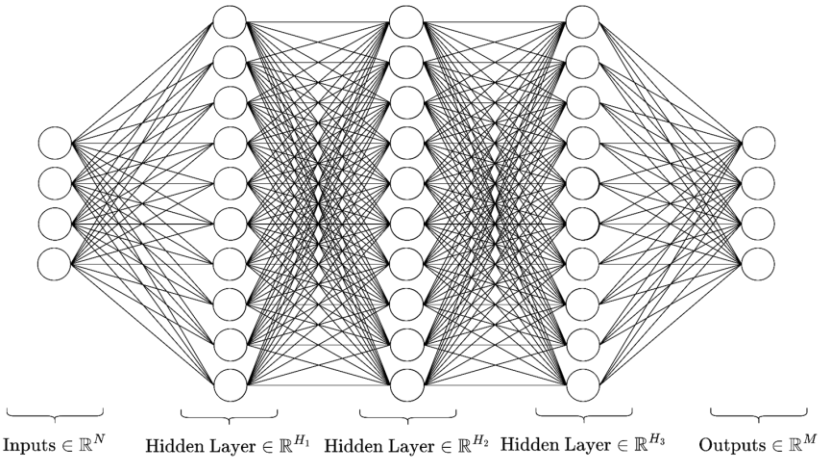
Families of Neural Networks

Multi-layer Perceptrons

Useful for **static** input-output relations

More hidden layers ~ better approximation of more complicated functions

Quick to design and implement

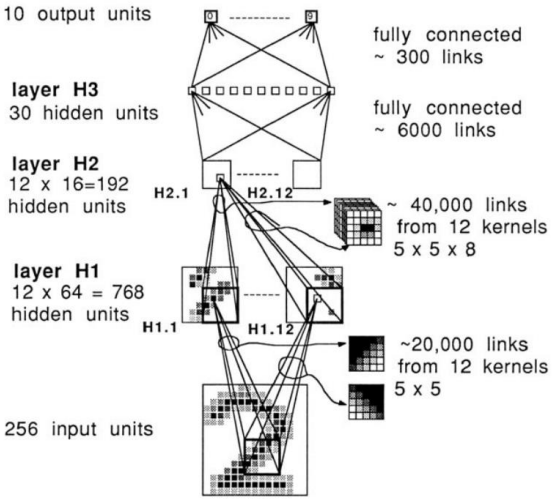


Convolutional Neural Networks

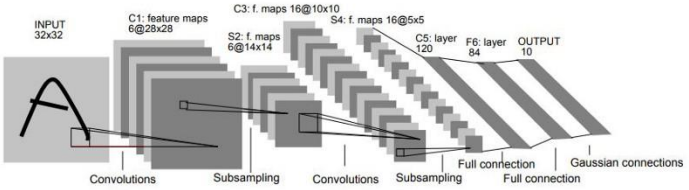
Learn “**kernels**”, i.e. matrices that convolve over *n*-dimensional data to extract abstract, lower-dimensional features.

Used often in **image and signal processing tasks** such as object detection and segmentation.

Accounts for translational variance: the object can be anywhere in the image and still be found



LeNet’s architecture: One of the first CNNs
https://doi.org/10.1162/neco.1989.1.4.541



LeNet-5: one of the most influential CNNs

Convolutions

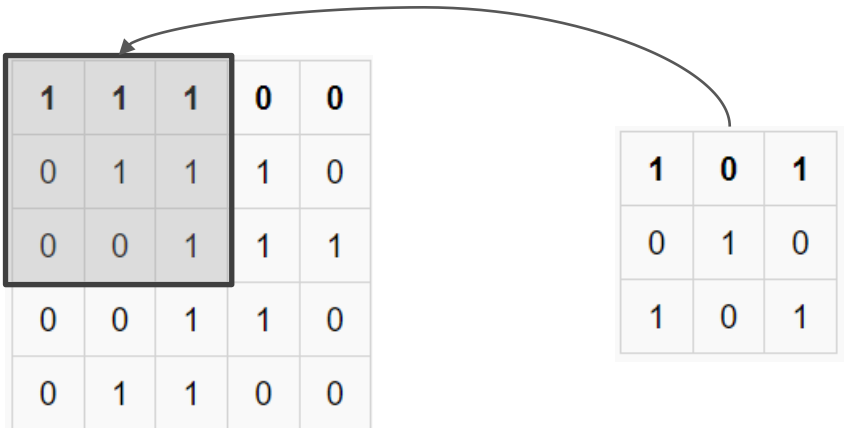
1	**1**	**1**	**0**	**0**
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

image

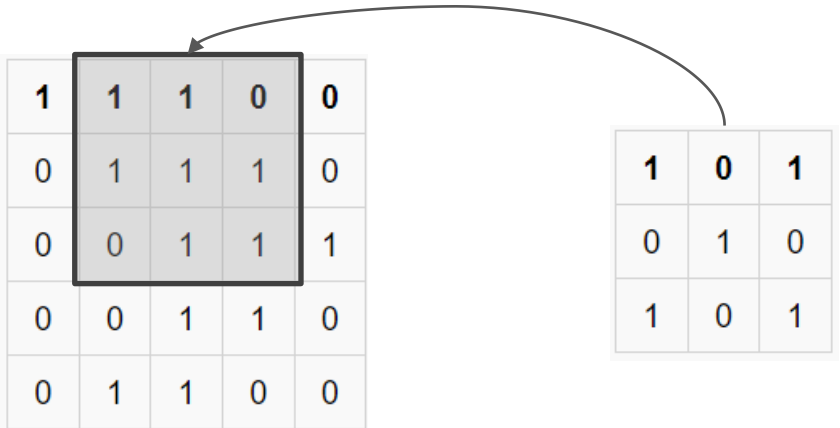
1	**0**	**1**
0	1	0
1	0	1

kernel

Convolutions



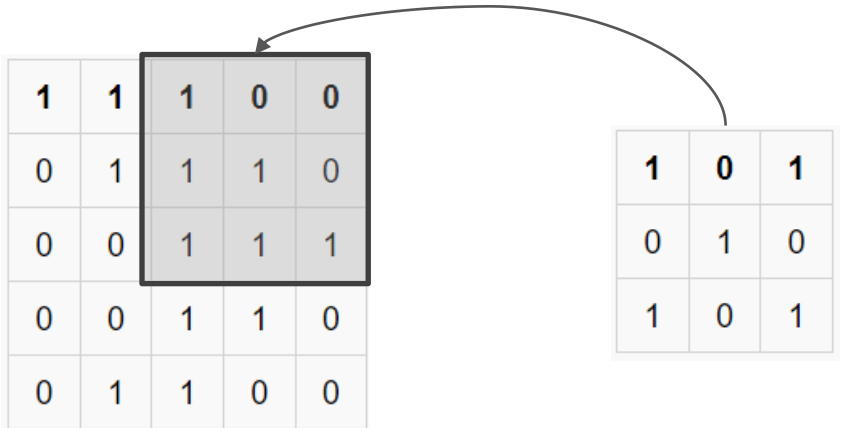
Convolutions



image

kernel

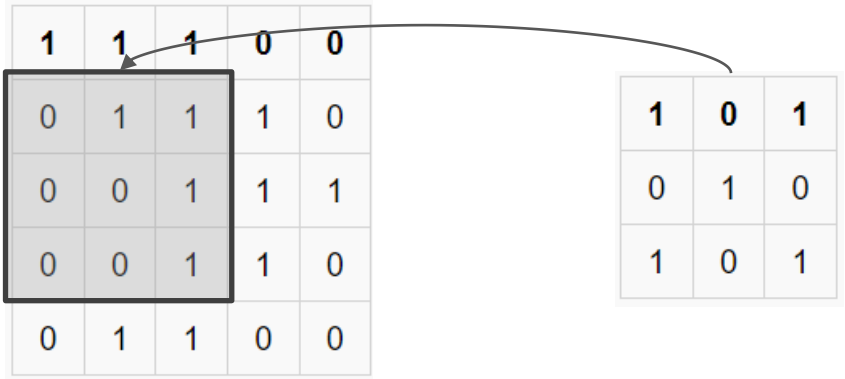
Convolutions



image

kernel

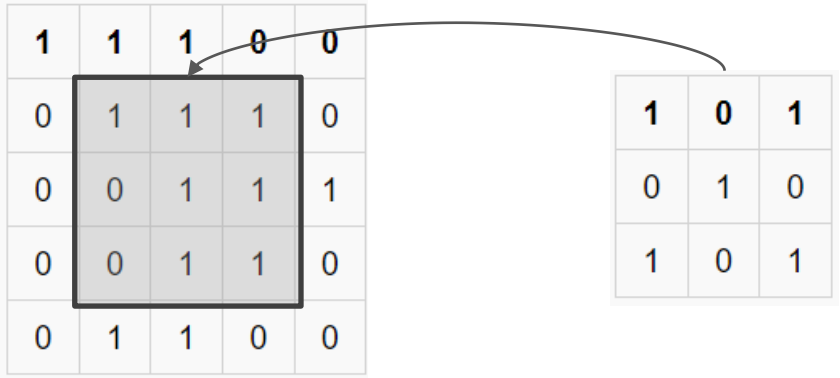
Convolutions



image

kernel

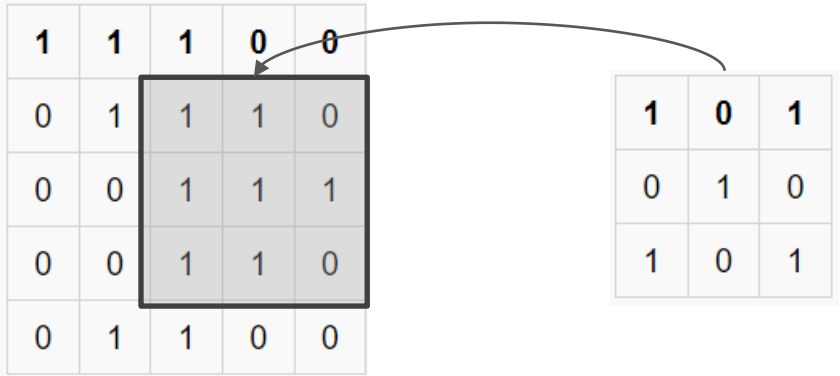
Convolutions



image

kernel

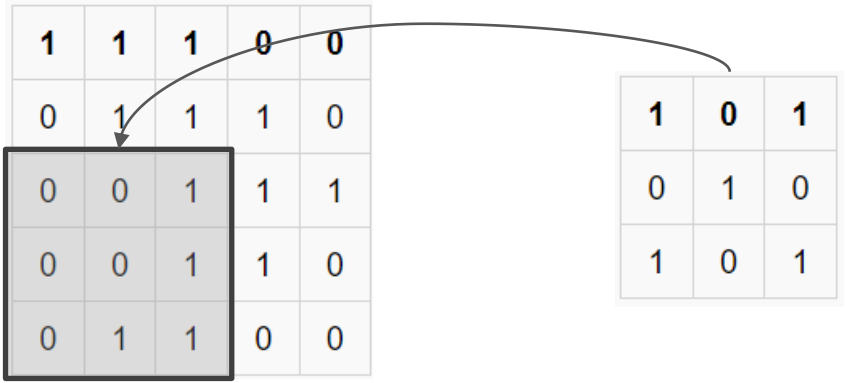
Convolutions



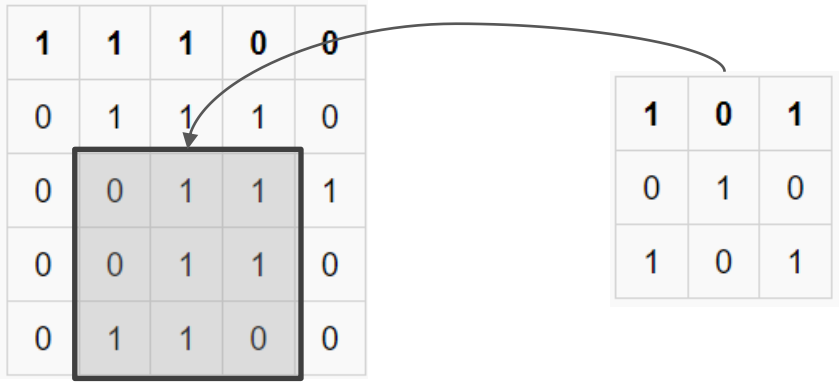
image

kernel

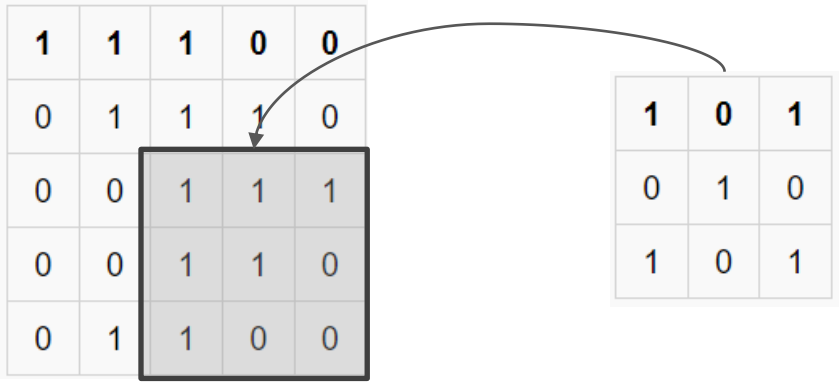
Convolutions



Convolutions



Convolutions



Convolutional Neural Networks

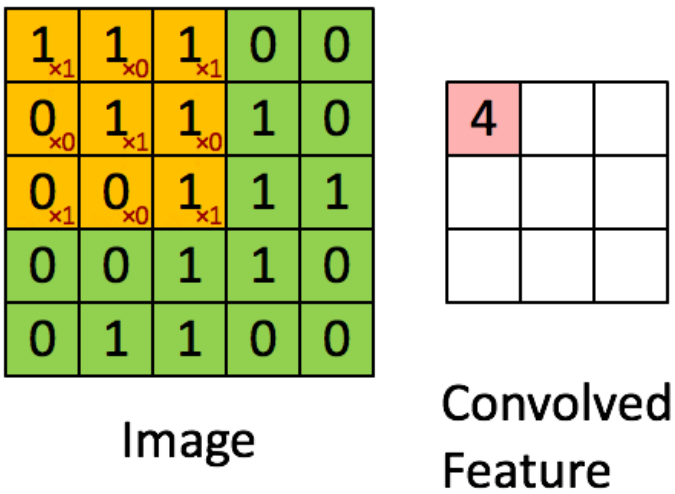
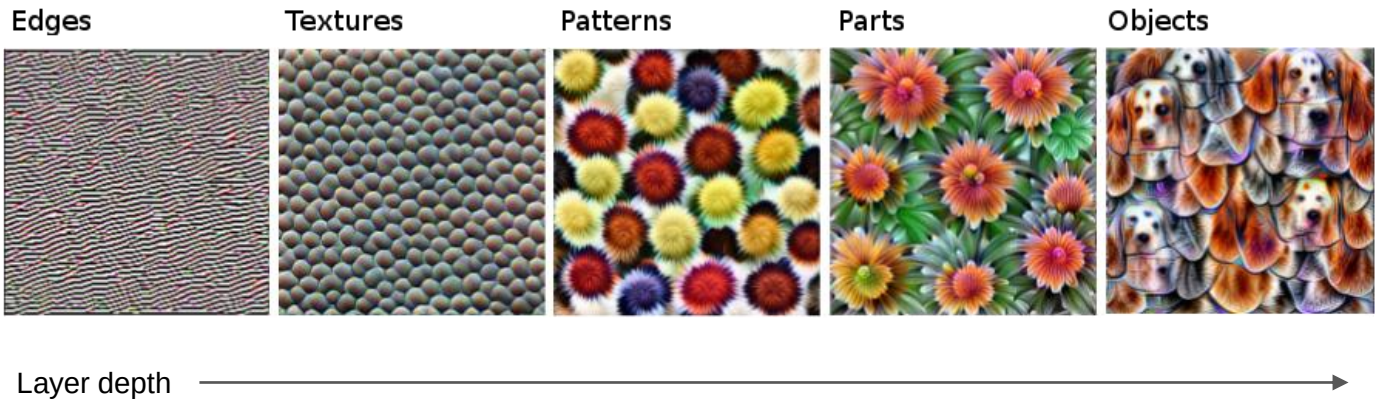


Image source: https://giphy.com/explore/convolutional-neural-network

What do these kernels learn?



Source: https://distill.pub/2017/feature-visualization/appendix/

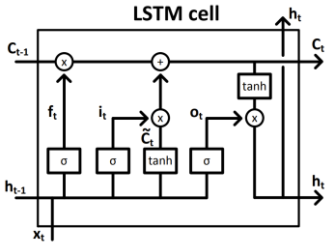
Recurrent Neural Networks

Outputs go back and forth between neurons (loops exist in the graphs)

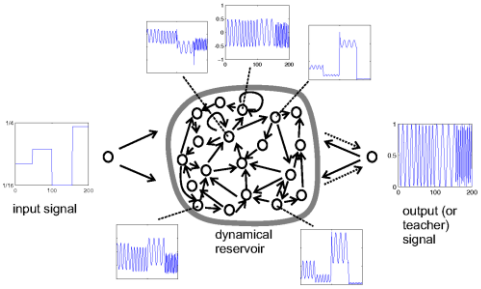
Approximates dynamical systems
- Any time-based function
- Any data that can be modelled as being “ordered”

Used often in **time-series tasks** like signal processing, natural language processing

Several types: Fully-connected RNNs, LSTMs, GRUs, reservoirs



An LSTM cell schematic. Adapted from: *doi.org/10.4233/uuid:dc73e1ff-0496-459a-986f-de37f7f250c9*



Echo state network schematic. Adapted from *www.scholarpedia.org/article/Echo_state_network*

Practical tips

How to train a NN – A pipeline

(Jaeger, 2022)

1. A clear idea of the nature of the task. *Inputs? Outputs? Underlying function?*
2. Decide on a loss function. *Categorization? CE? Regression? Quadratic?*
3. Decide on a regularization method. *L2 norm? Dropout? Early stopping?*
4. Vector-encode inputs. *Dimensionality reduction? Ratio of dimension/data points?*
5. Decide on architecture. *Width/breadth? Activation functions? Output function?*
6. Set up cross-validation routine. *Simple cross-val? Early stopping? k-fold?*
7. Train and test your model. *Epochs? Learning rate? Optimizer?*
8. Enjoy the power of neural networks!

Validation (Recap)

Split your training set into two!

- New train set
- Unseen-by-the-model “validation”set

Validation (Recap)

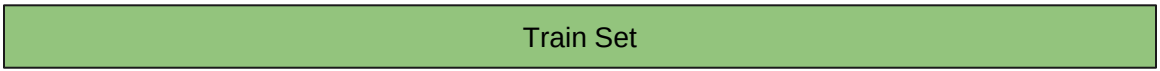
Split your training set into two!

- New train set
- Unseen-by-the-model “validation”set
- e.g. 80-20 split (Note: split ratio depends on the model, task and data)

k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

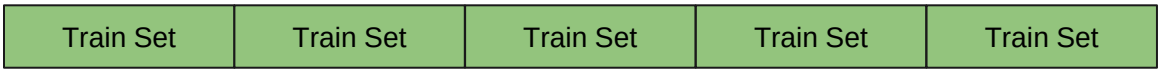


k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

e.g. *k*=5 (5-fold cross-validation)

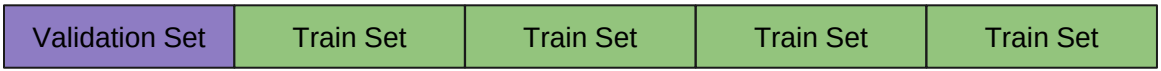


k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

e.g. *k*=5 (5-fold cross-validation)

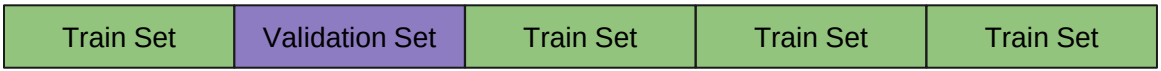


k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

e.g. *k*=5 (5-fold cross-validation)



k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

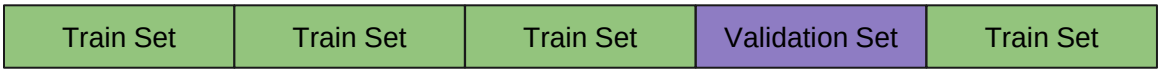
e.g. *k*=5 (5-fold cross-validation)

k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

e.g. *k*=5 (5-fold cross-validation)

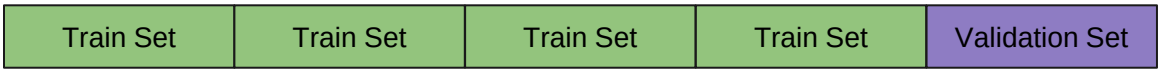


k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

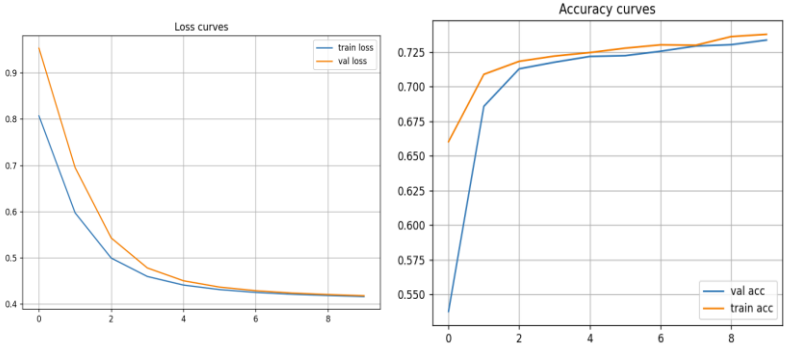
e.g. *k*=5 (5-fold cross-validation)



Result = average over all validation passes

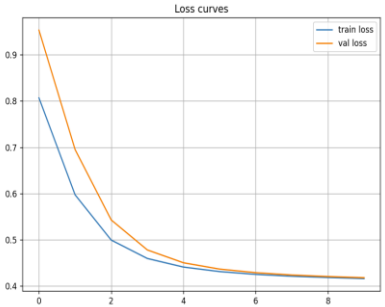
Training curves

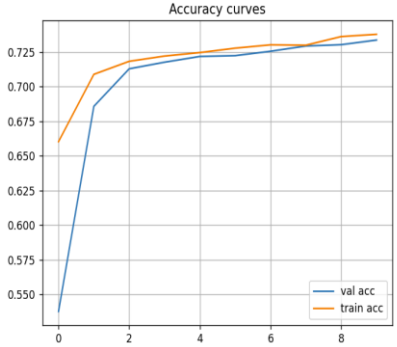
Important to plot!



Training curves

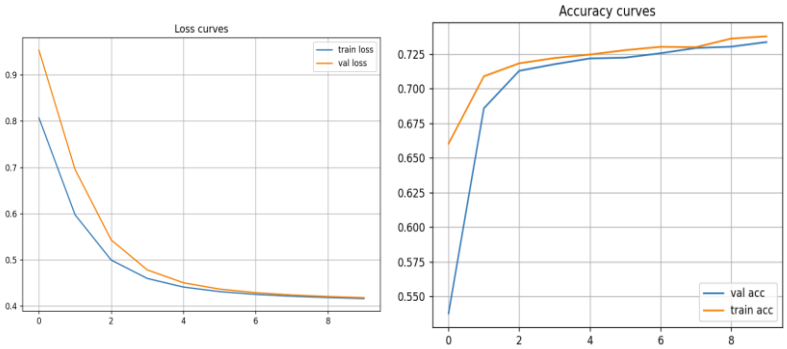
Important to plot!(!!!!)





Training curves

Important to plot!(!!!!)

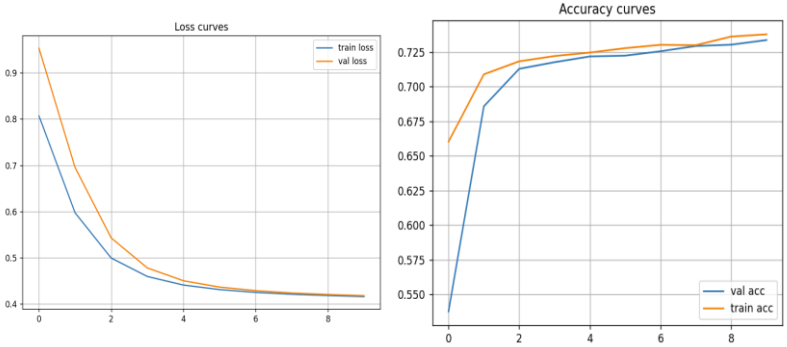


Shows if and how fast your model is learning on task-relevant metrics

- e.g. loss, accuracy, AUC, F1 score
- Plot scores over training epochs

Training curves

Important to plot!(!!!!)

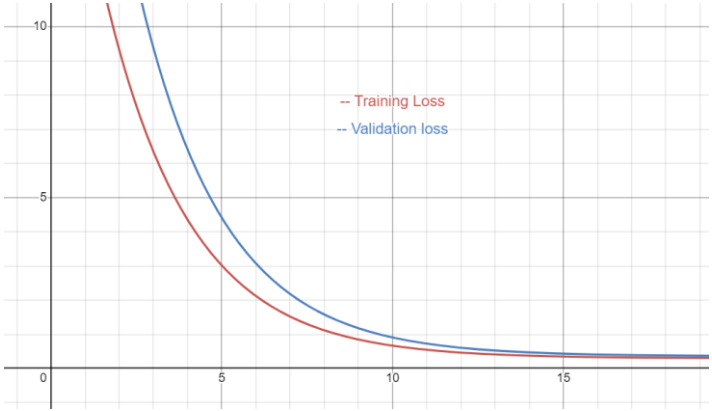


Shows if and how fast your model is learning on task-relevant metrics

- e.g. loss, accuracy, AUC, F1 score
- Plot scores over training epochs

May indicate potential over and underfitting

Reading training curves



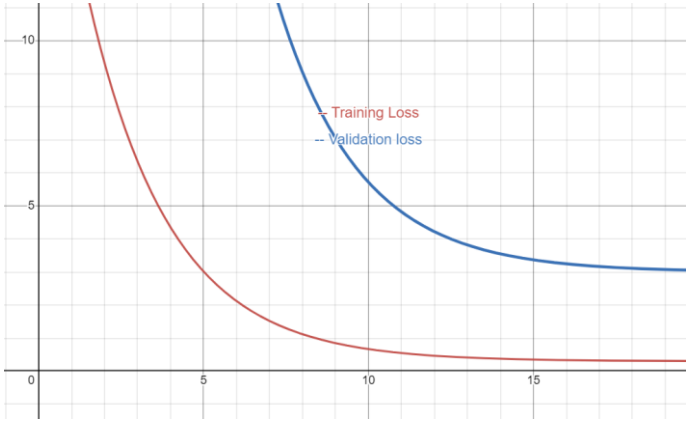
If

validation loss > training loss

then often the model is *good!*

Low loss == Better

Reading training curves



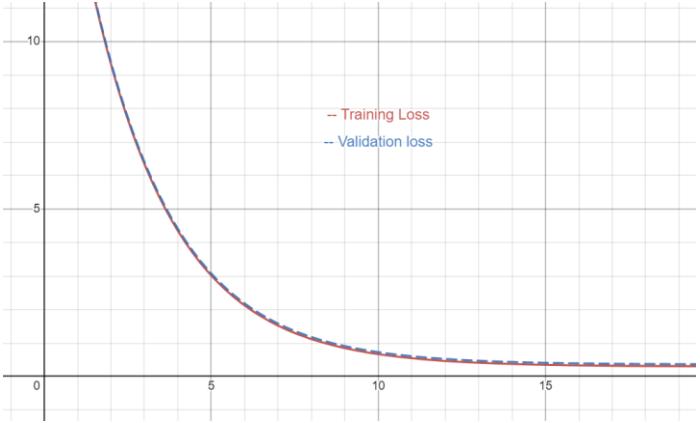
If

validation loss >> training loss

then often the model is *overfitting*

Low loss == Better

Reading training curves



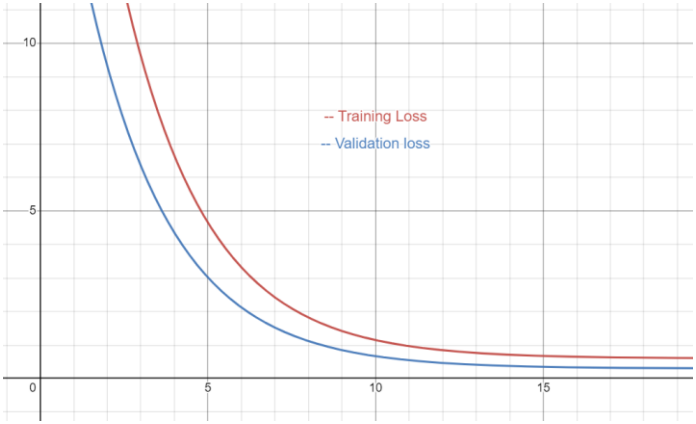
If

validation loss ~ training loss

then often the model is *underfitting*

Low loss == Better

Reading training curves



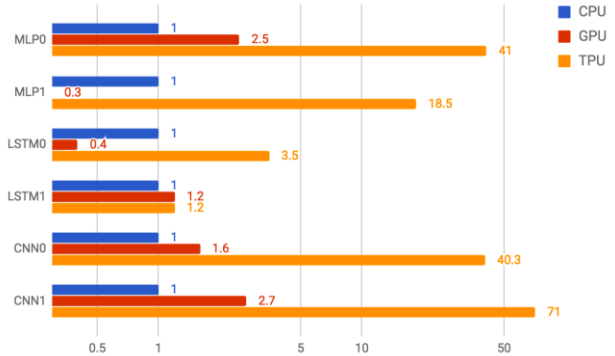
If

validation loss < training loss

something is *very* wrong/*totally expected*!

Low loss == Better

Parallelization: Speeding up NNs



Main math operation in NNs:

- Matrix-vector multiplications
- Element-wise nonlinear activation functions

Parallelization can be used to **massively speed up** learning and deployment!

- Multi-core CPUs
- Graphics processing units (GPUs)
- Tensor processing units (TPUs)
- FPGAs

Image from https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

Frontiers

Deep learning

- Models with hundreds of layers, billions of weights
- Transformers, generative adversarial networks, autoencoders
- AutoMLs: a tool to automatically generate good ML models for a task

Explainable AI (XAI)

- Explainable+interpretable models
- Human-like and human-understandable reasoning

Reservoir computing

- Echo state networks
- Conceptors