



# An Introduction to Neural Networks

Satchit Chatterji  
MSc Artificial Intelligence  
University of Amsterdam

*satchit.chatterji@gmail.com*



# Why you should consider Neural Networks

Satchit Chatterji  
BSc Artificial Intelligence  
University of Groningen

*satchit.chatterji@gmail.com*



2022



# An Introduction to Neural Networks

Satchit Chatterji  
MSc Artificial Intelligence  
University of Amsterdam

*satchit.chatterji@gmail.com*



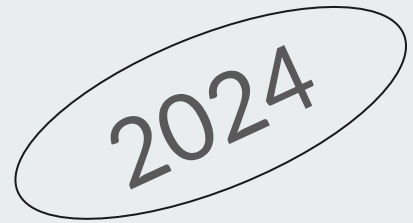
2023



# I don't need to convince you to use Neural Networks

Satchit Chatterji  
MSc Artificial Intelligence  
University of Amsterdam

*satchit.chatterji@gmail.com*





## The short answer

- They're useful!



## The short answer

- They're useful!
- They're fast!



## The short answer

- They're useful!
- They're fast!
- They're (now) easy to implement!

---

## The short answer

- They're useful!
- They're fast!
- They're (now) easy to implement!
- They're cute!



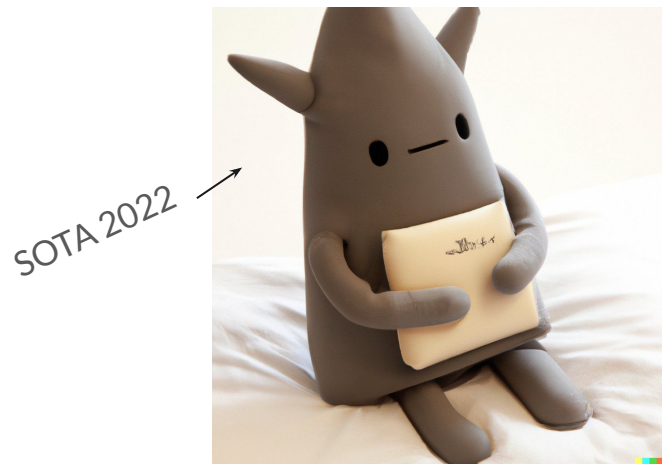
<https://twitter.com/gdb/status/1512521912064229377>



---

## The short answer

- They're useful!
- They're fast!
- They're (now) easy to implement!
- They're cute!



<https://twitter.com/gdb/status/1512521912064229377>



## The short answer

- They're useful!
- They're fast!
- They're (now) easy to implement!
- They're cute!
- They're *definitely* not going to take over the world!



<https://twitter.com/gdb/status/1512521912064229377>



OpenAI



## The short answer

- They're useful!
- They're fast!
- They're (now) easy to implement!
- They're cute!
- They're *definitely* not going to take over the world!

## The long answer

*It's a bit more complicated than that...*



<https://twitter.com/gdb/status/1512521912064229377>



OpenAI



# Introduction to the introduction

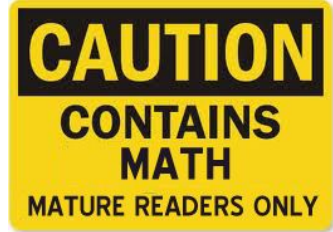
## Goals of this lecture:

The whats, hows, whys, whichs and wheres

- Teach you what a neural network is and how it works
- Why you should use them, and why not
- Which neural networks are used today
- Where neural networks are headed next

## Along with:

- A demo in a simulated environment
- A few tips on building and training your own networks



# Introduction to the introduction

## Goals of this lecture:

The whats, hows, whys, whichs and wheres

- Teach you what a neural network is and how it works
- Why you should use them, and why not
- Which neural networks are used today
- Where neural networks are headed next

## Along with:

- A demo in a simulated environment
- A few tips on building and training your own networks



# Introduction to Supervised Machine Learning

Given: Input-output examples of the form:

$$S = (\mathbf{x}_i, \mathbf{y}_i)_{i=1, \dots, T} \quad \mathbf{x}_i \in \mathbb{R}^N, \mathbf{y}_i \in \mathbb{R}^M$$



# Introduction to *Supervised* Machine Learning

Given: Input-output examples of the form:

$$S = (\mathbf{x}_i, \mathbf{y}_i)_{i=1, \dots, T} \quad \mathbf{x}_i \in \mathbb{R}^N, \mathbf{y}_i \in \mathbb{R}^M$$



# Introduction to *Supervised* Machine Learning

Given: Input-output examples of the form:

$$S = (\mathbf{x}_i, \mathbf{y}_i)_{i=1, \dots, T} \quad \mathbf{x}_i \in \mathbb{R}^N, \mathbf{y}_i \in \mathbb{R}^M$$

Assumption: Data is generated by a “true ” function, with some added noise:

$$\mathbf{y}_i = f(\mathbf{x}_i) + v_i$$





# Introduction to *Supervised* Machine Learning

Given: Input-output examples of the form:

$$S = (\mathbf{x}_i, \mathbf{y}_i)_{i=1, \dots, T} \quad \mathbf{x}_i \in \mathbb{R}^N, \mathbf{y}_i \in \mathbb{R}^M$$

Assumption: Data is generated by a “true” function, with some added noise:

$$\mathbf{y}_i = f(\mathbf{x}_i) + v_i$$

Goal: Learn an approximation  $\hat{f}(\mathbf{x})$  of the generator function to use on new data:

$$\hat{f}(\mathbf{x}) \approx f(\mathbf{x})$$



# Introduction to *Supervised* Machine Learning

Given: Input-output examples of the form:

$$S = (\mathbf{x}_i, \mathbf{y}_i)_{i=1, \dots, T} \quad \mathbf{x}_i \in \mathbb{R}^N, \mathbf{y}_i \in \mathbb{R}^M$$

Assumption: Data is generated by a “true” function, with some added noise:

$$\mathbf{y}_i = f(\mathbf{x}_i) + v_i$$

Goal: Learn an approximation  $\hat{f}(\mathbf{x})$  of the generator function to use on new data:

$$\hat{f}(\mathbf{x}) \approx f(\mathbf{x})$$

Loss function: A distance between  $\hat{f}(\mathbf{x})$  and  $f(\mathbf{x})$  such that we can say  $\hat{f}(\mathbf{x})$  is “good” if  $L$  is low across many given instances of  $S$ .

$$L : \mathbb{R}^M \times \mathbb{R}^M \rightarrow \mathbb{R}^{\geq 0}$$



## Aim: Learn a function with low “risk”

Risk: What we want to minimize

$$R(\hat{f}) = E[L(\hat{f}(X), Y)]$$



## Aim: Learn a function with low “risk”

Risk: What we want to minimize

$$R(\hat{f}) = E[L(\hat{f}(X), Y)]$$

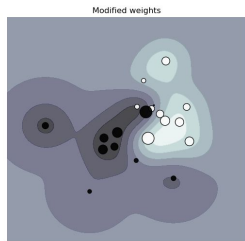
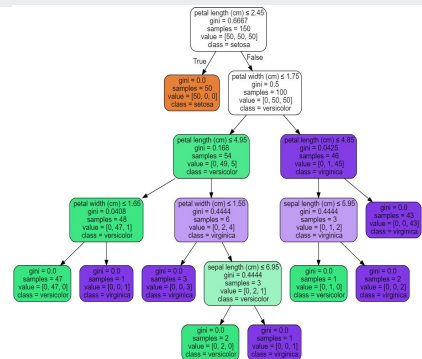
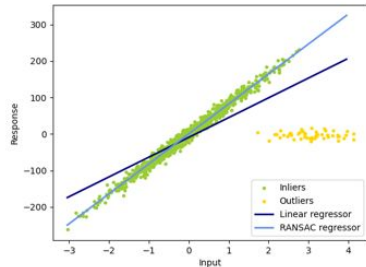
Empirical Risk: What we can actually calculate

(for a “candidate” model  $h$ , averaged over  $N$  training examples)

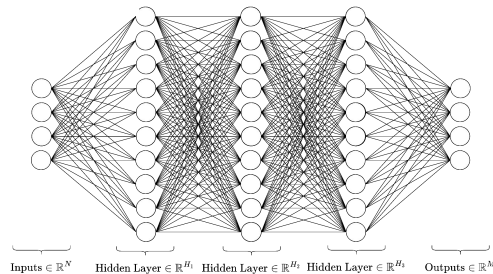
$$R^{\text{emp}}(h) = 1/N \sum_{i=1}^N L(h(\mathbf{x}_i), \mathbf{y}_i)$$

# Common Approaches

- Linear/Polynomial/Logistic Regression
- (Boosted) Decision trees
- Support Vector Machines
- Naive Bayes
- Neural Networks
- ...



$$P(A | B) = \frac{P(B | A) \cdot P(A)}{P(B)}$$



# 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

**McCulloch & Pitts (1943)**

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

**Rosenblatt (1958)**

Description of the *perceptron*

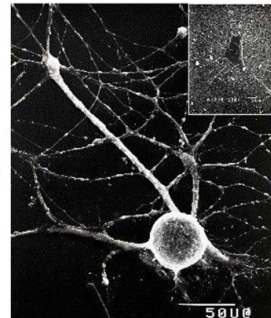
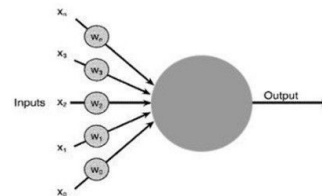
**Rumelhart, Hinton & Williams (1986)**

*Multi-layer perceptrons and error backpropagation (learning principle)*

**Modern:**

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

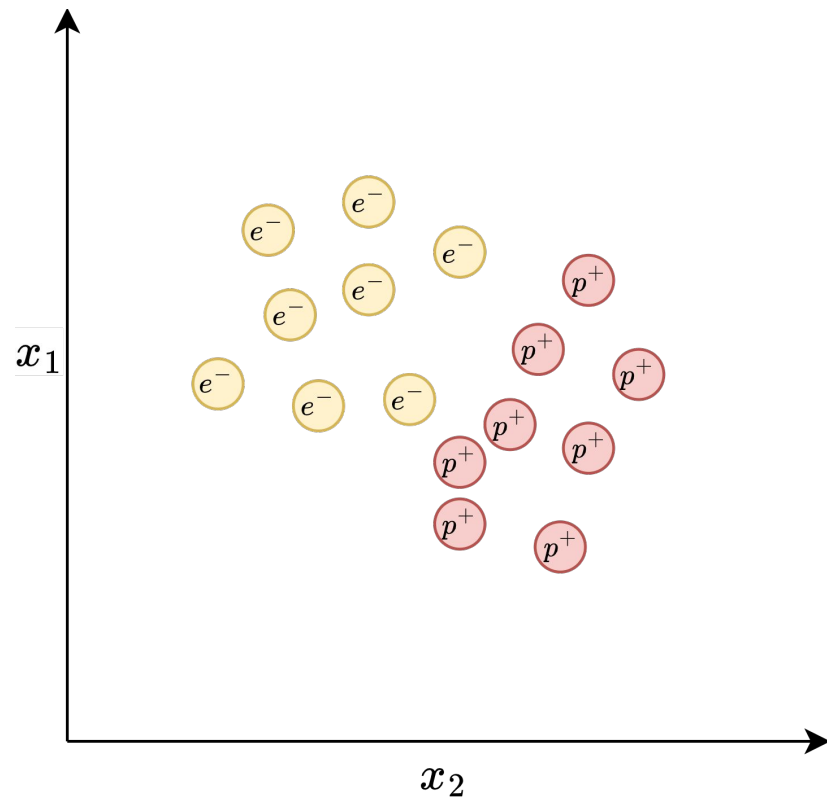
you vs the guy she told you not to worry about:



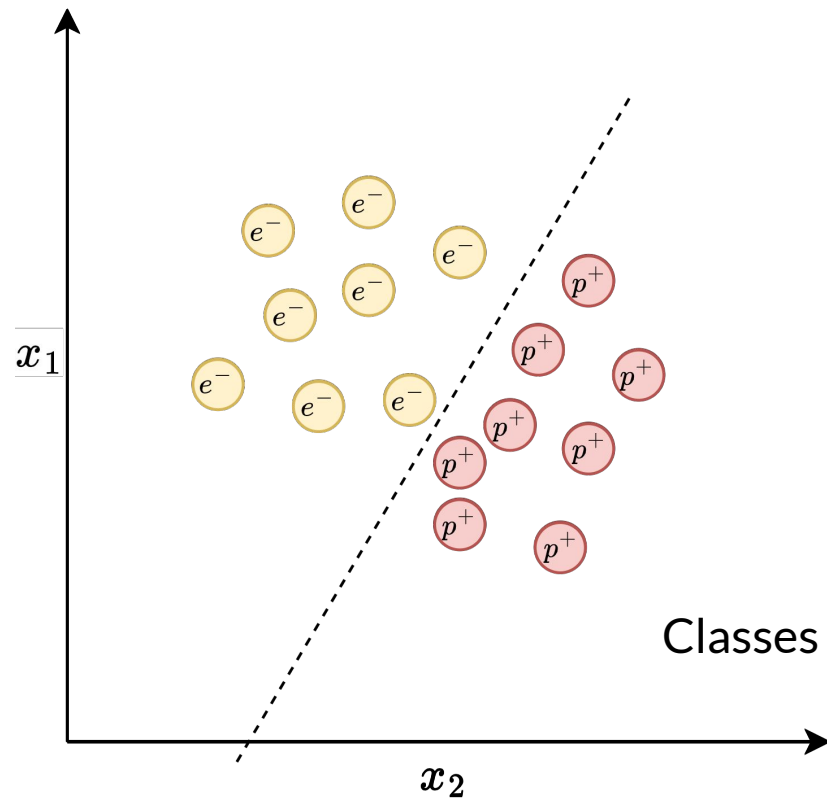
Source: [linkedin.com/company/deeplearningai](https://www.linkedin.com/company/deeplearningai)

---

# Building a Neural Network From Scratch (mathematically)







Classes are "linearly separable"

$$\hat{y} = w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n$$

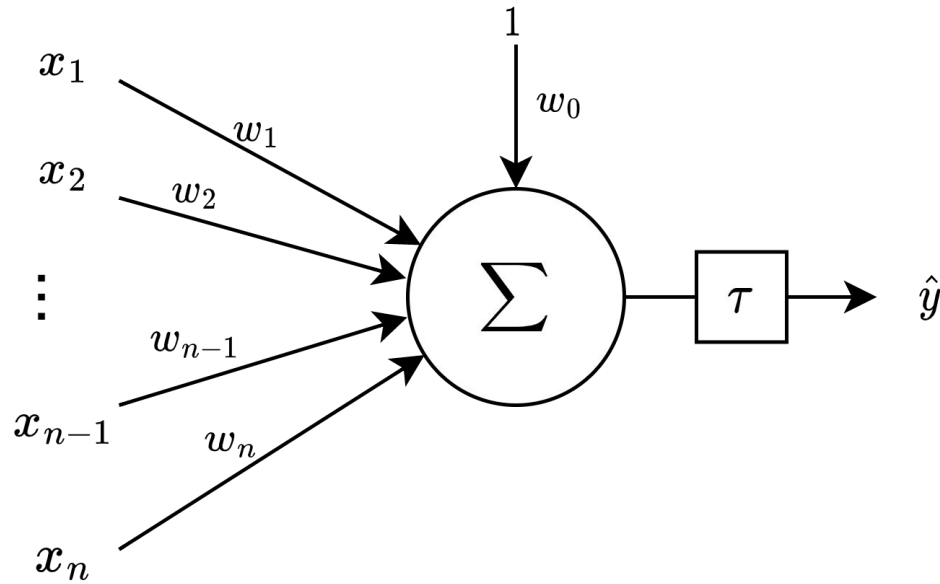
$w_i \leftarrow$  Coefficients

$x_i \leftarrow$  Variables

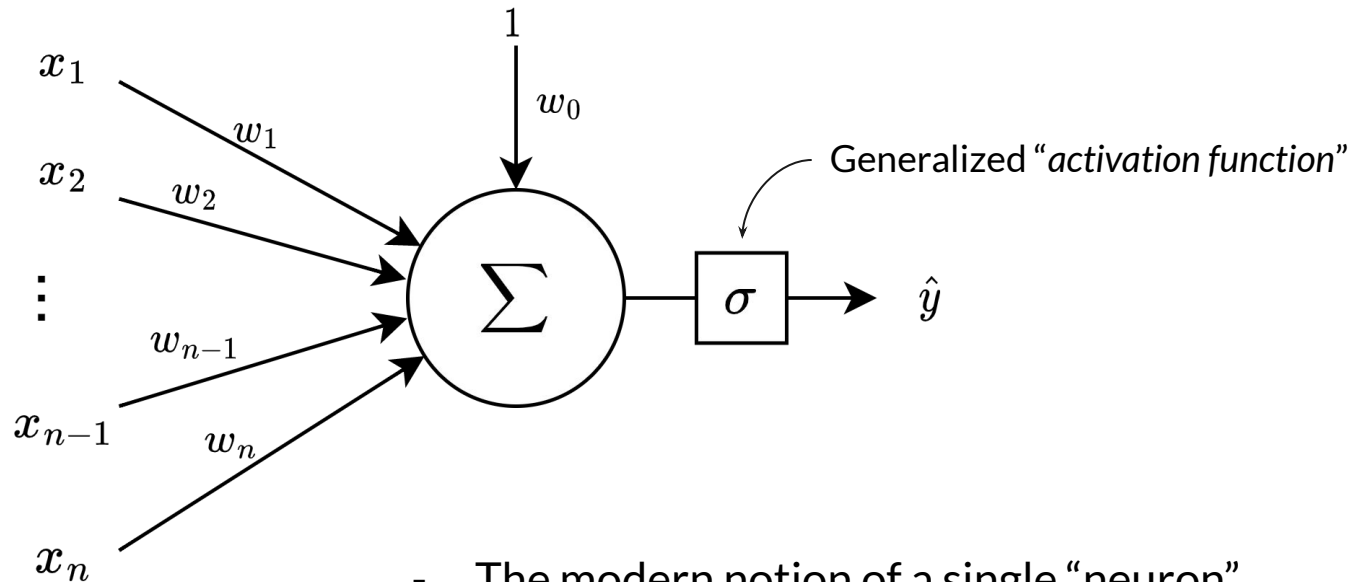
$$\hat{y} = \tau(w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n)$$

$$\tau(x) = \begin{cases} 1 & \text{if } x \geq 0 \quad e^- \\ 0 & \text{if } x < 0 \quad p^+ \end{cases}$$

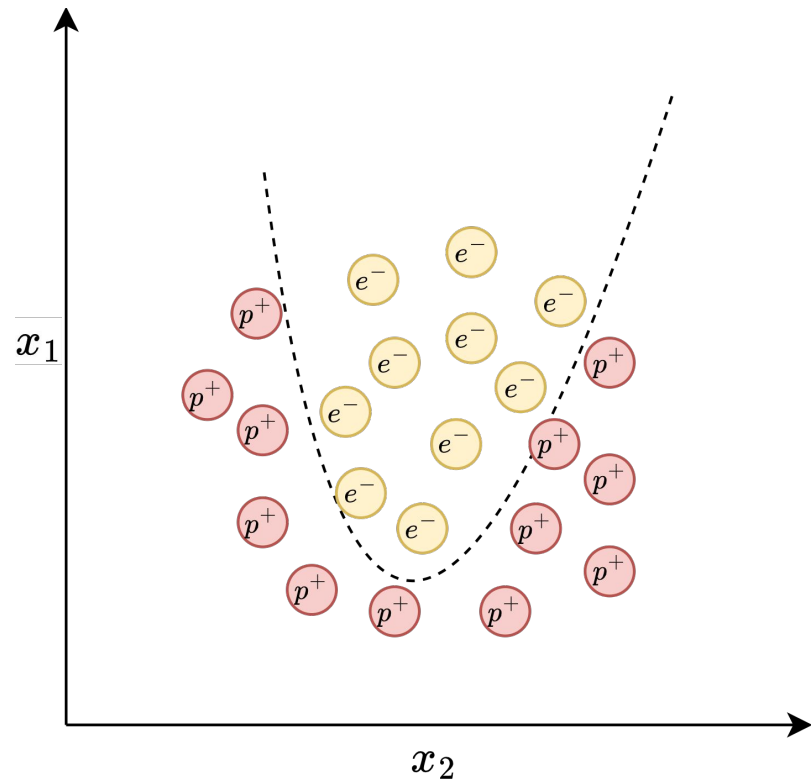
$$\hat{y} = \tau(w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n)$$

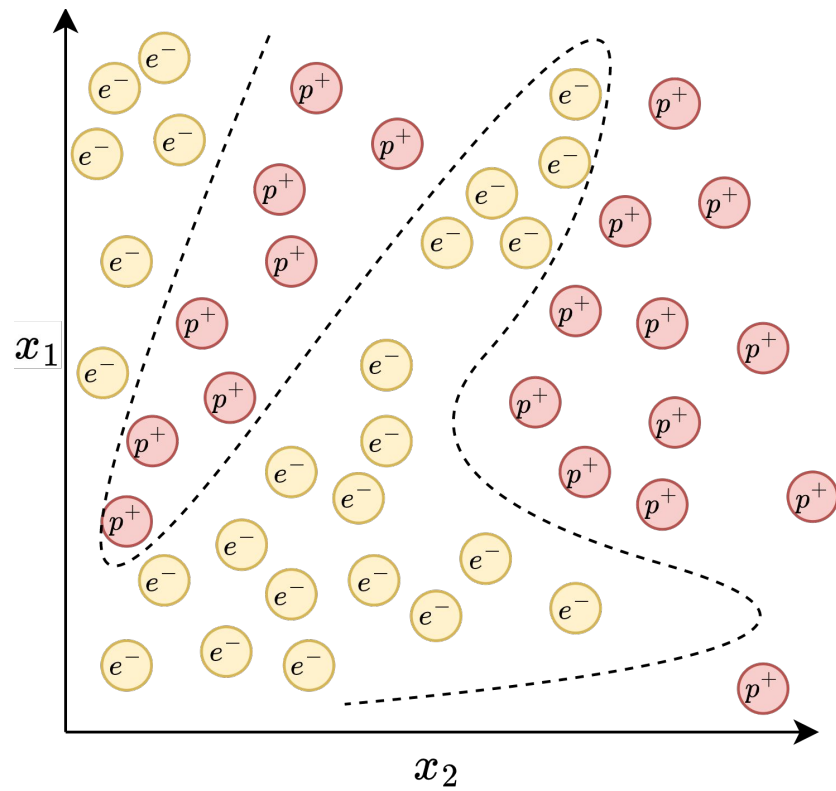


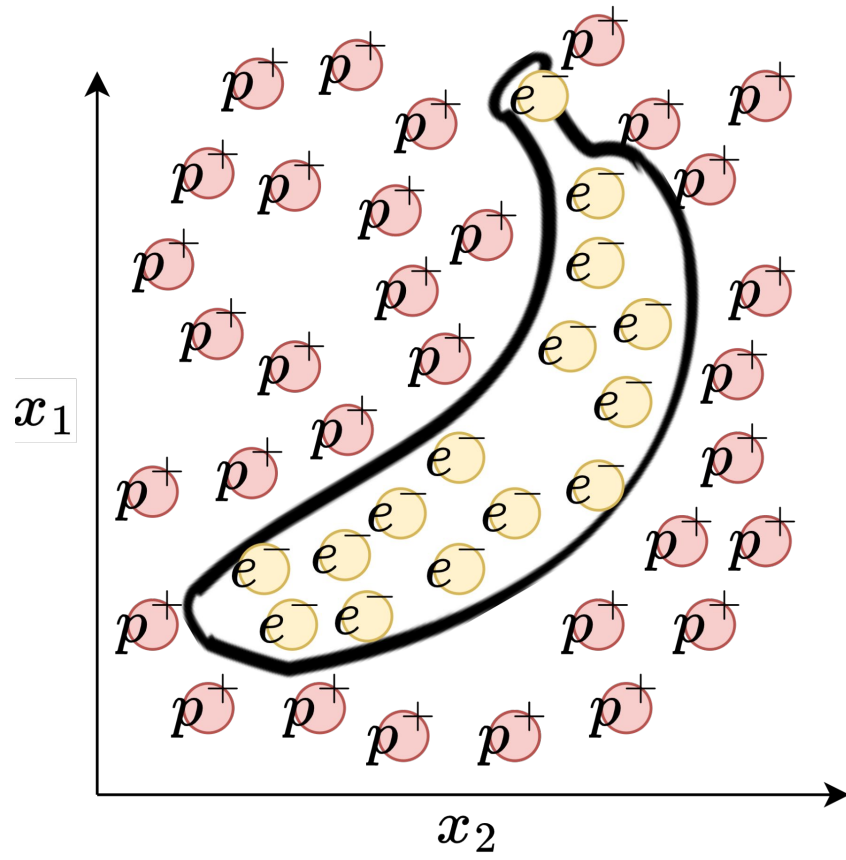
# The “Perceptron”



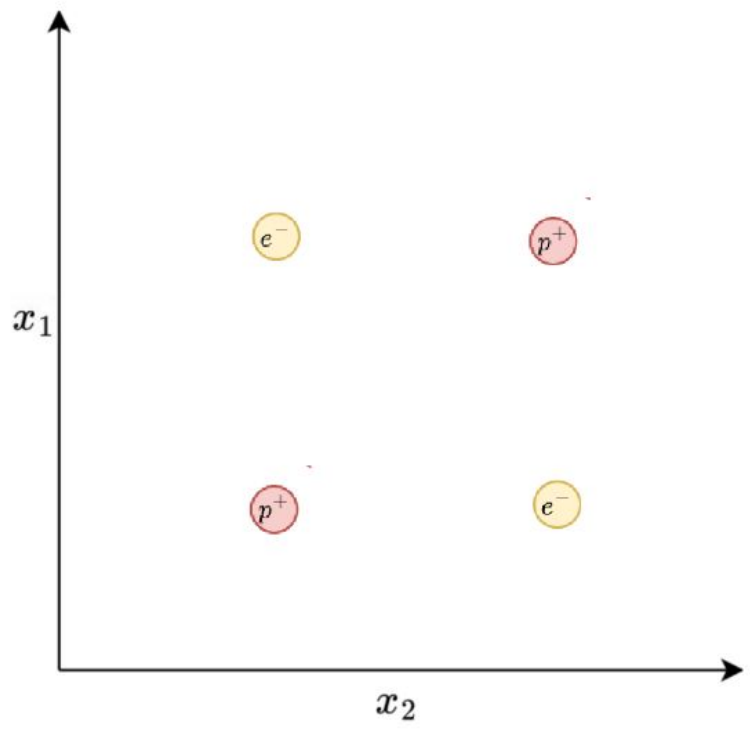
- The modern notion of a single “neuron”
- BUT: Only works on linearly separable classes

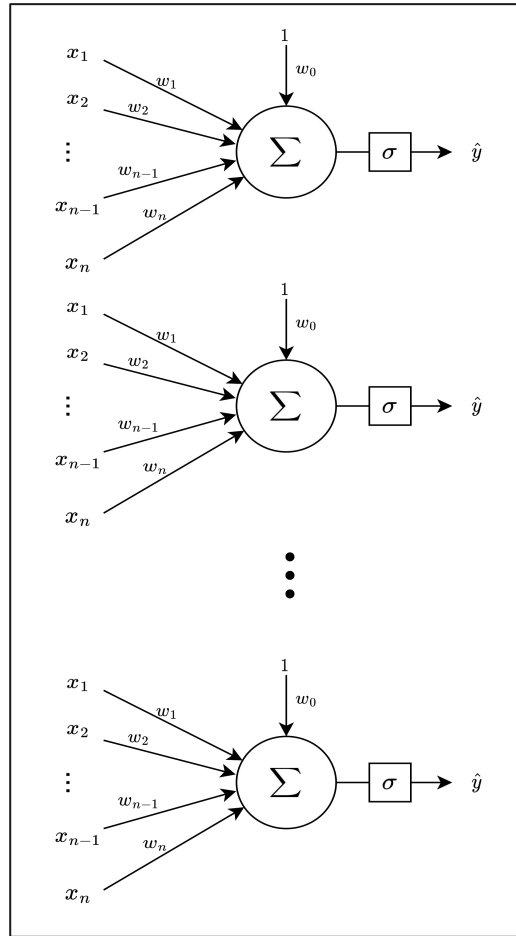


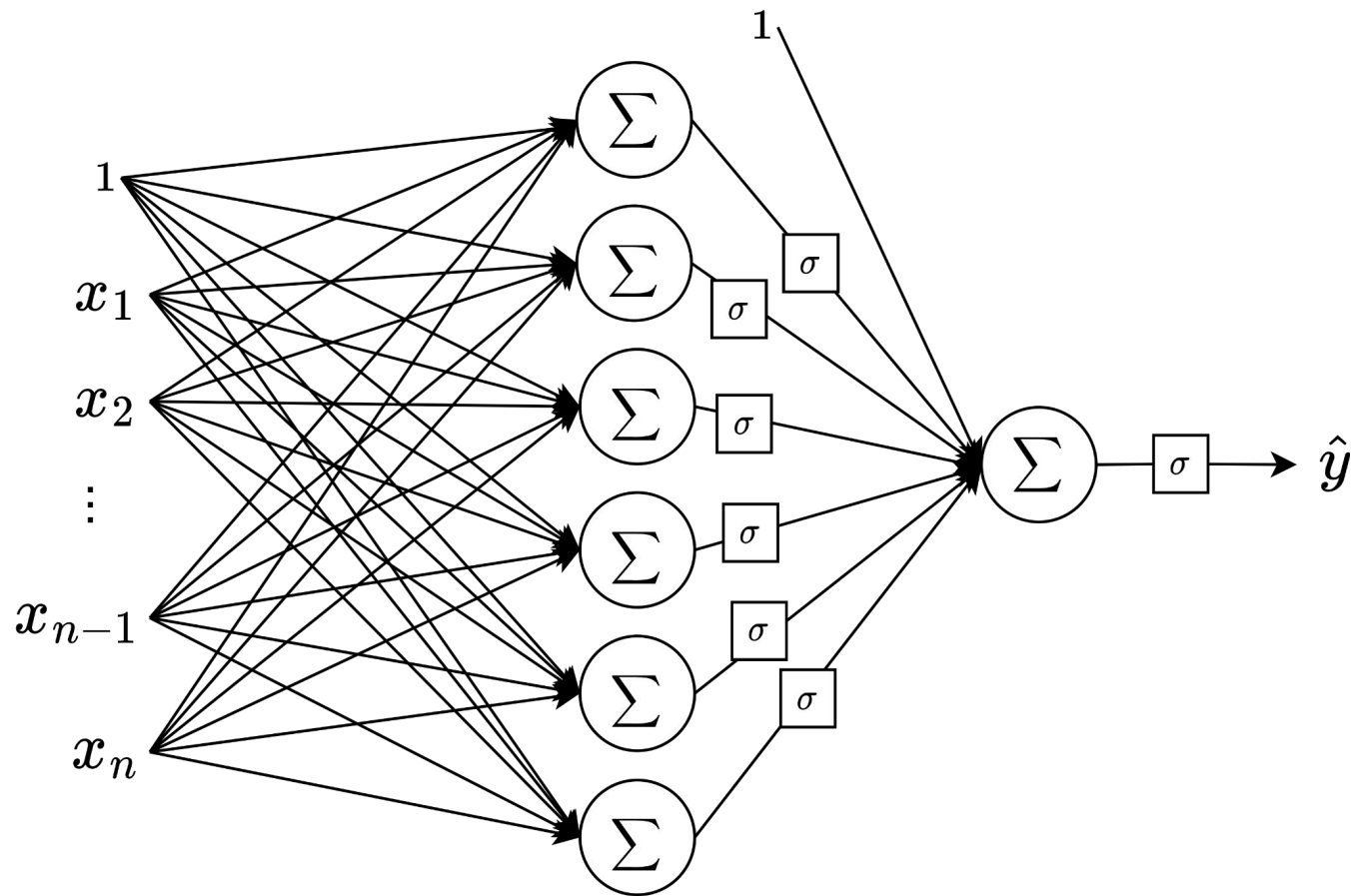




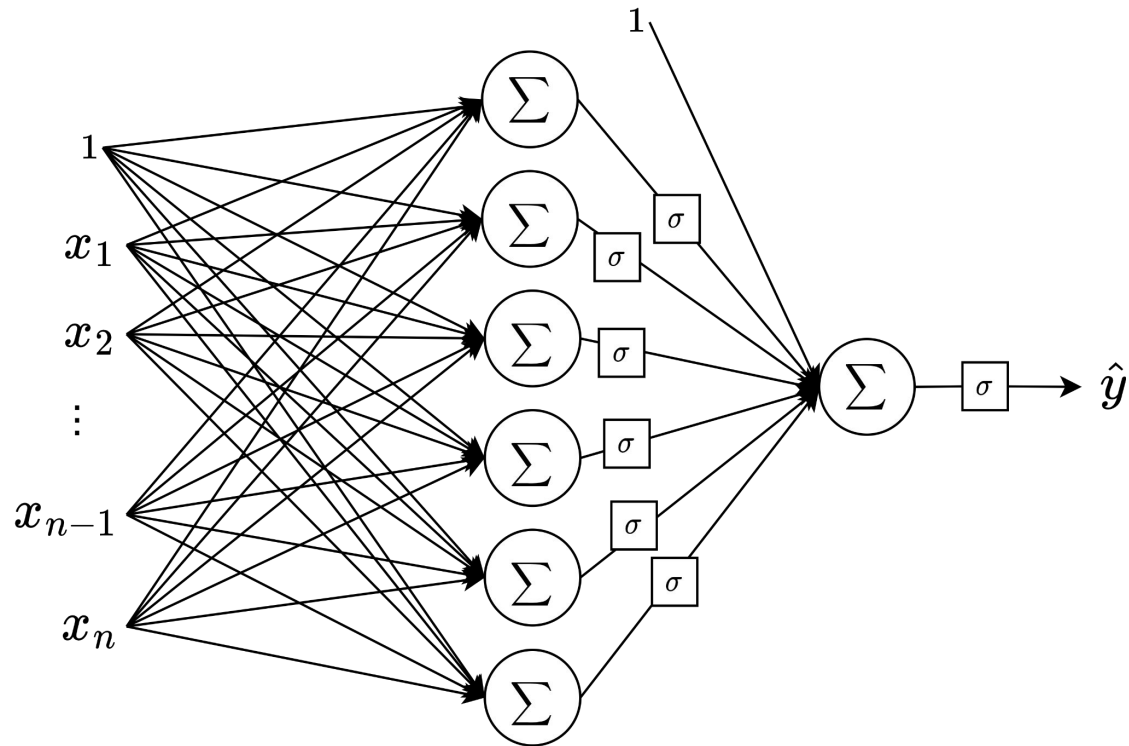








# Multi-layer Perceptrons (MLPs)



$$\hat{y} = \sigma(w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n)$$

$$\begin{aligned}\hat{y} &= \sigma(w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n) \\ &= \sigma(\vec{w}^\top \vec{x})\end{aligned}$$

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

“Activation”



“Bias”



$$\hat{y} = \sigma(w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n)$$
$$= \sigma(\vec{w}^T \vec{x})$$

“Activation function”



$$\vec{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

“Weight vector”



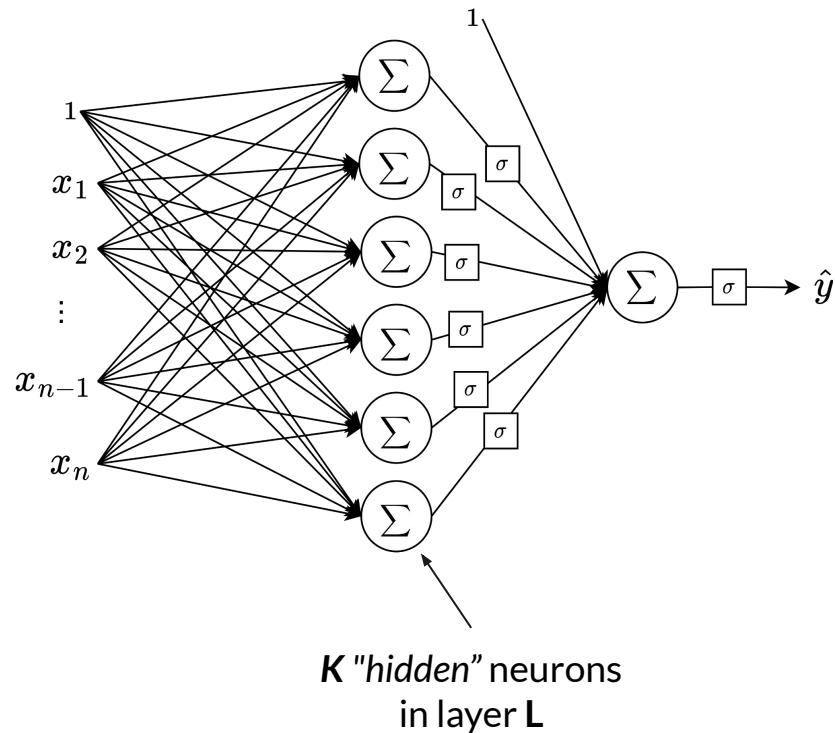
$$\vec{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

“Feature vector”



Activation/  
output  
of neuron  $k$

$$o^k = [w_0^k \quad w_1^k \quad \dots \quad w_n^k] \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$





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

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

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

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

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

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

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

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

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

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

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

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

$$\hat{y} = \sigma(w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n)$$



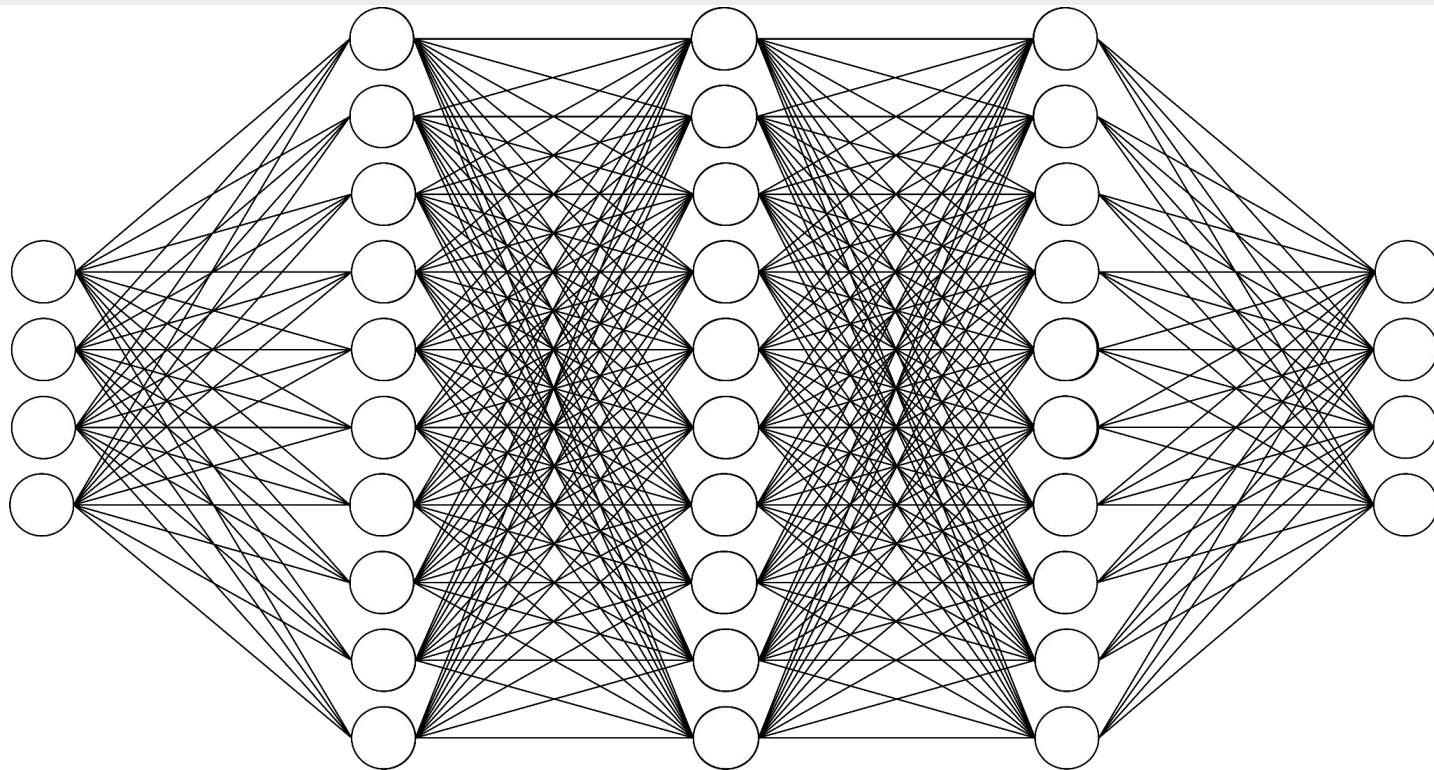
$$o^L = W\vec{x} + \vec{b}$$

$$\hat{y} = \sigma(w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n)$$

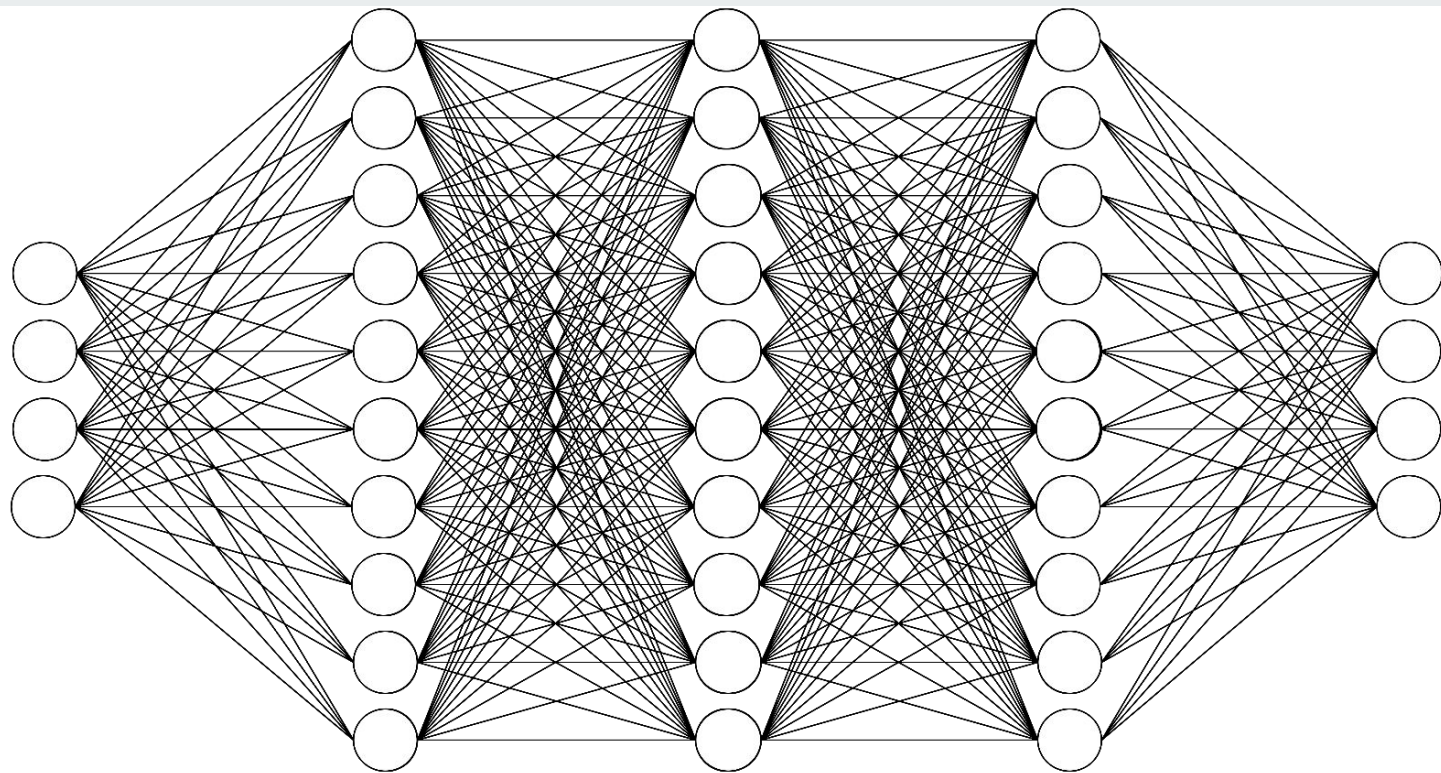
$$o = \sigma(Wx^{in} + b)$$

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



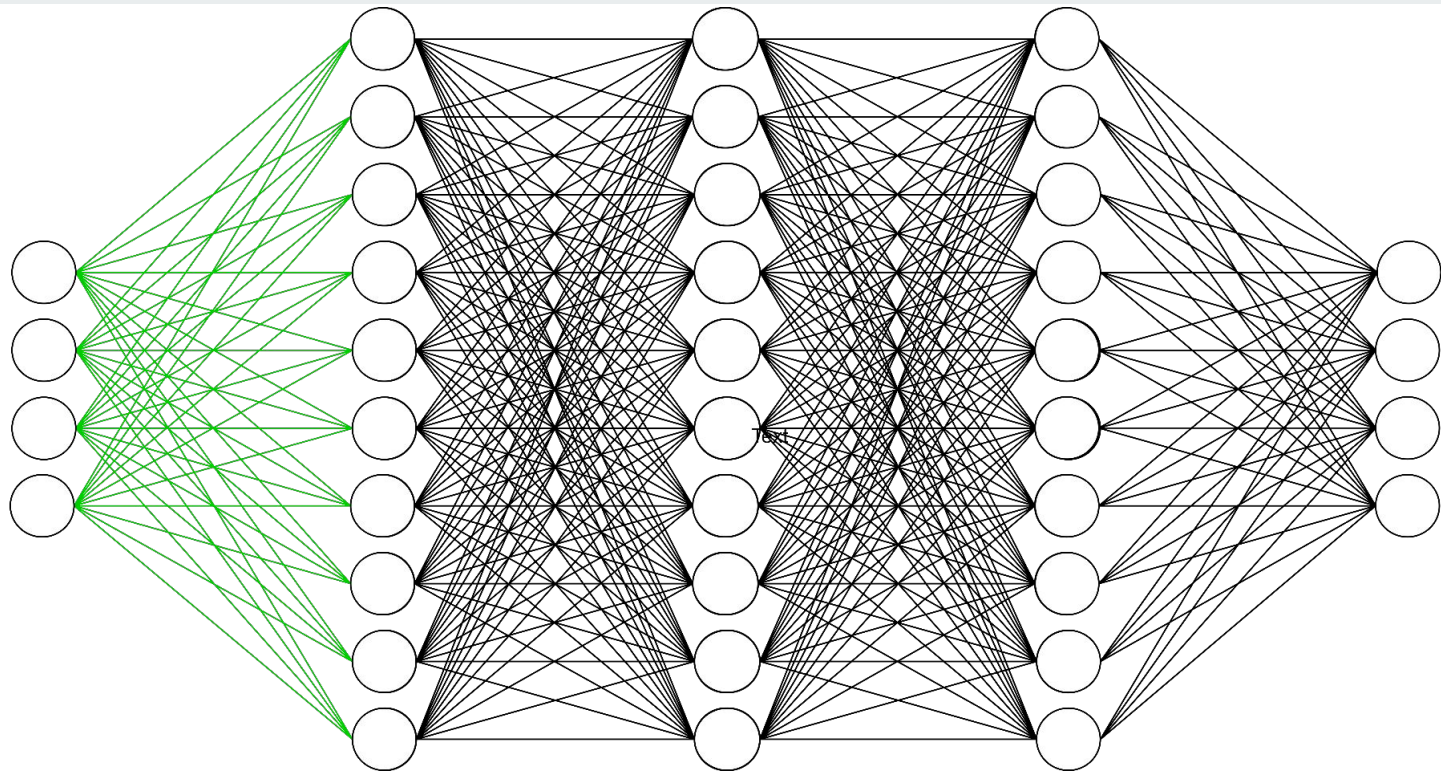
Inputs  $\in \mathbb{R}^N$

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

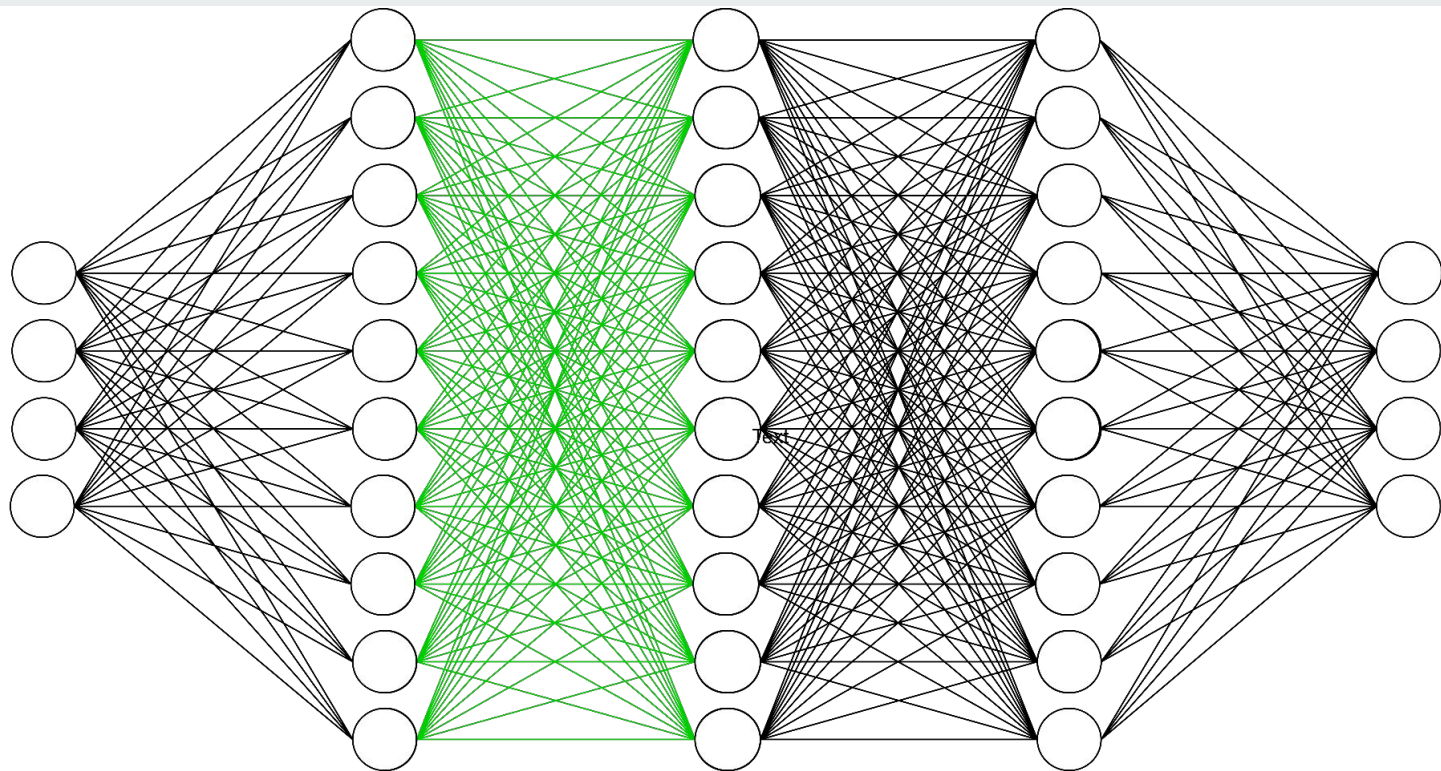
Hidden Layer  $\in \mathbb{R}^{H_2}$

Hidden Layer  $\in \mathbb{R}^{H_3}$

Outputs  $\in \mathbb{R}^M$

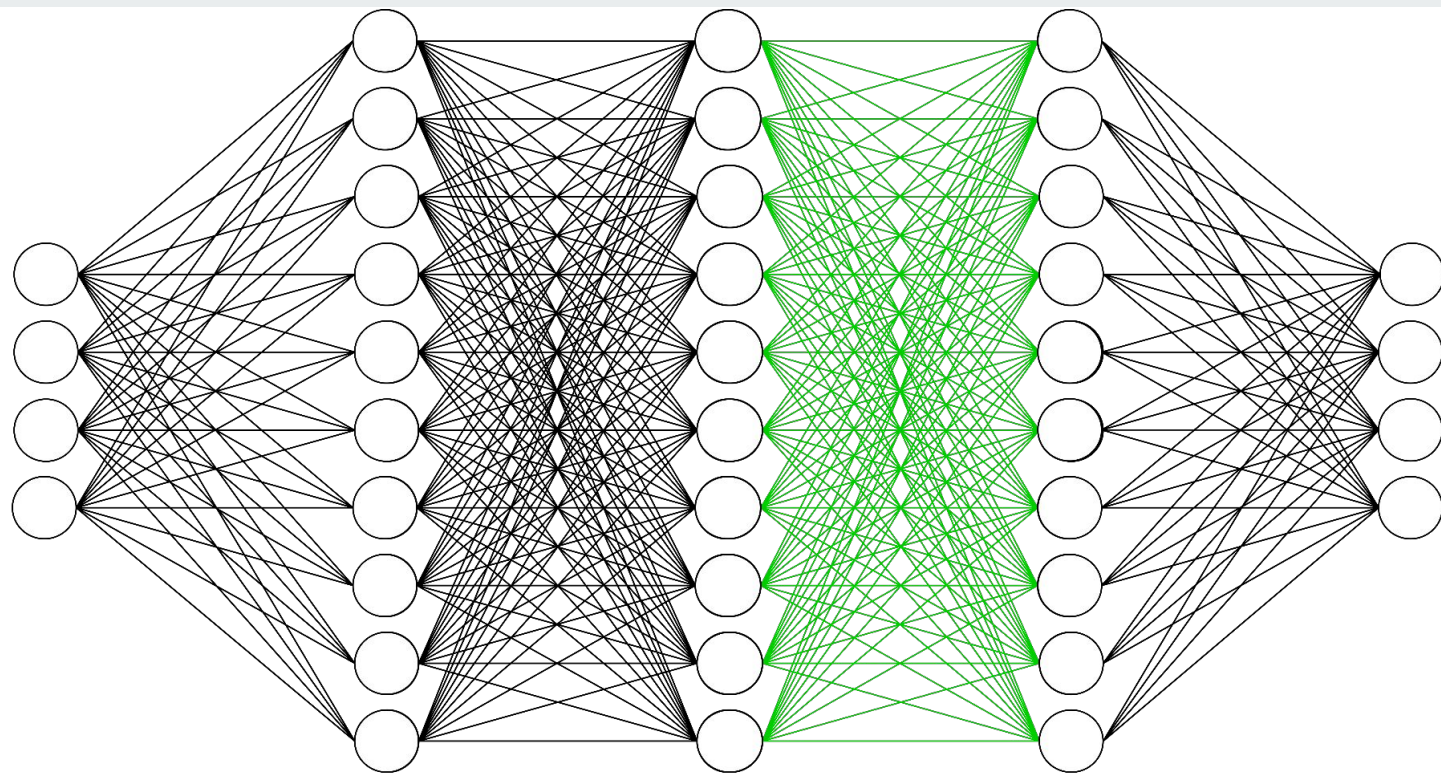


$\underbrace{\hspace{10em}}$   
Inputs  $\in \mathbb{R}^N$



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

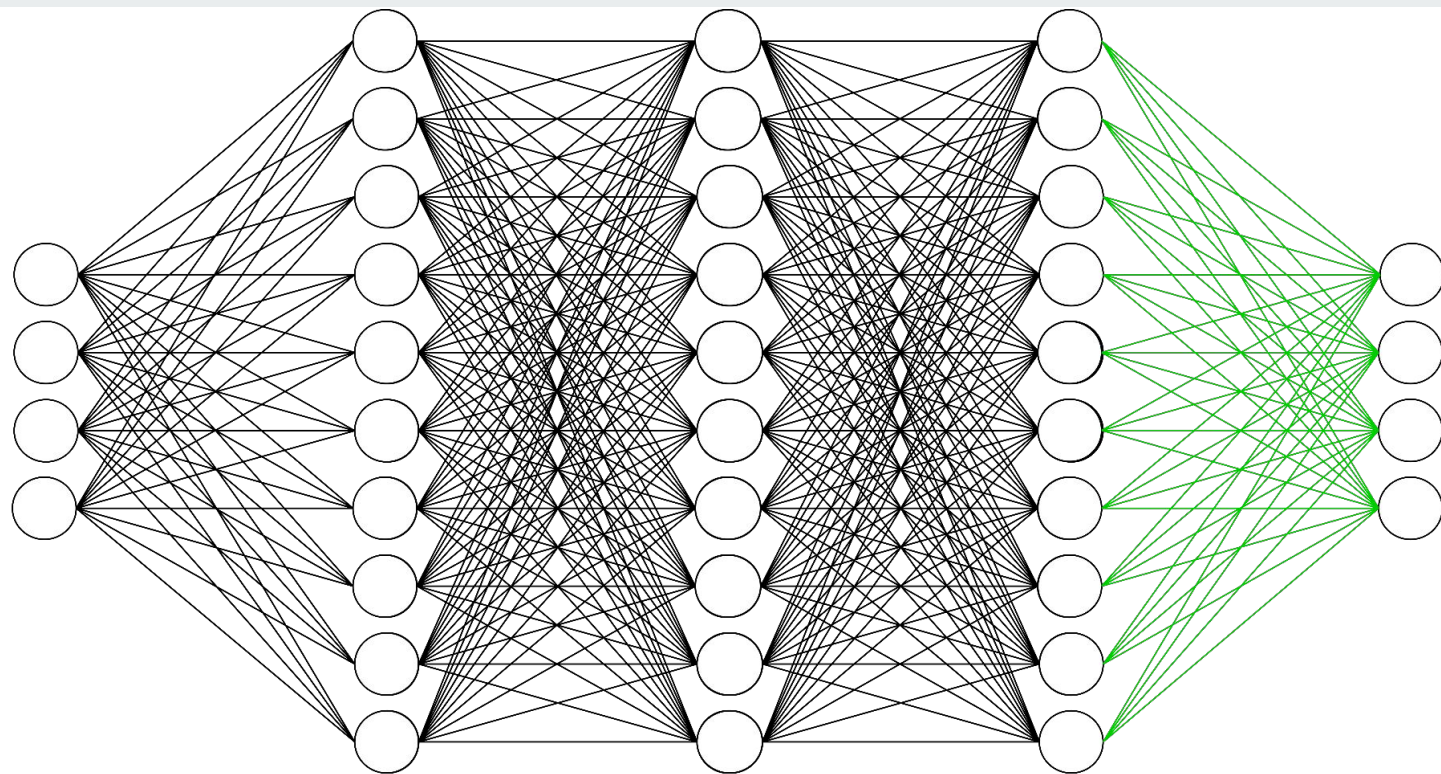
Hidden Layer  $\in \mathbb{R}^{H_1}$



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

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

Hidden Layer  $\in \mathbb{R}^{H_2}$



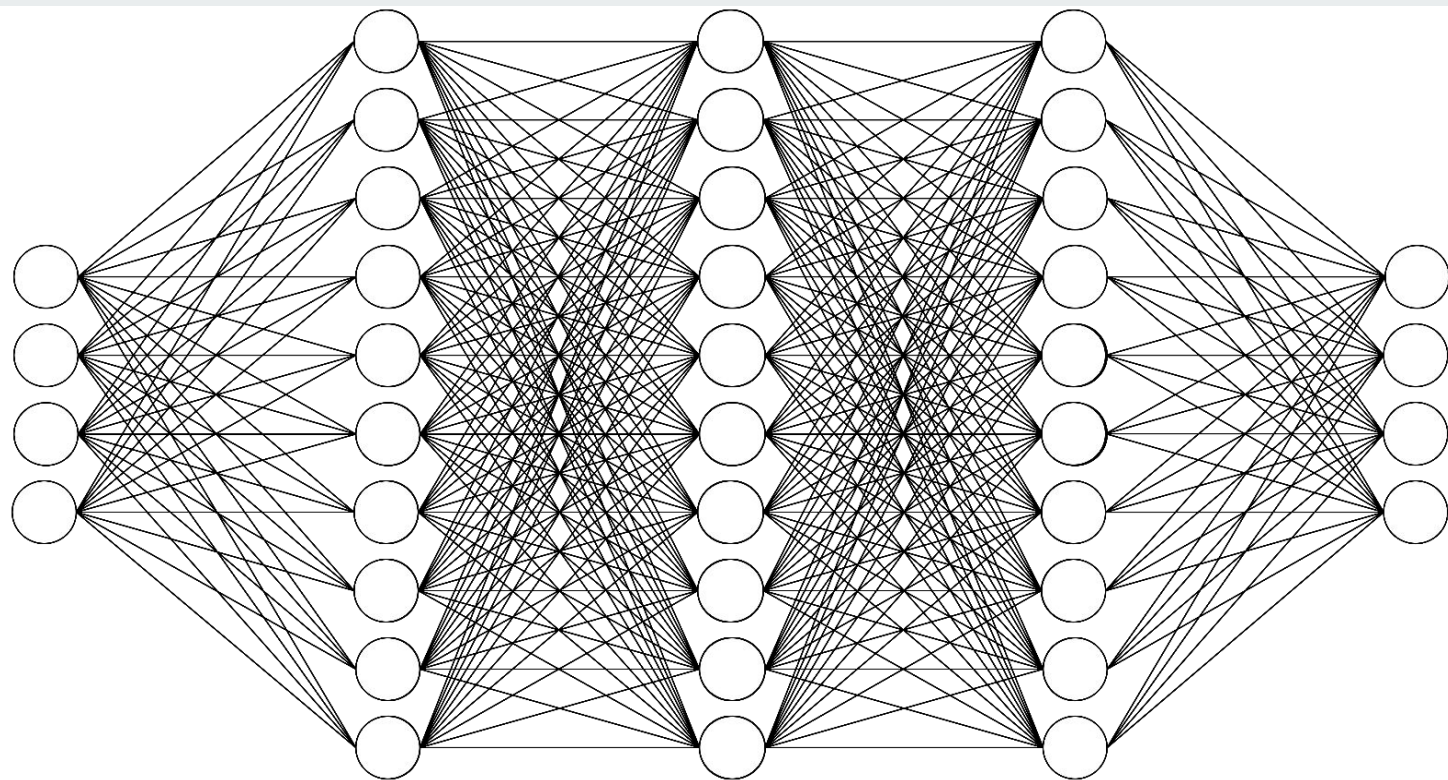
Inputs  $\in \mathbb{R}^N$

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

Hidden Layer  $\in \mathbb{R}^{H_2}$

Hidden Layer  $\in \mathbb{R}^{H_3}$





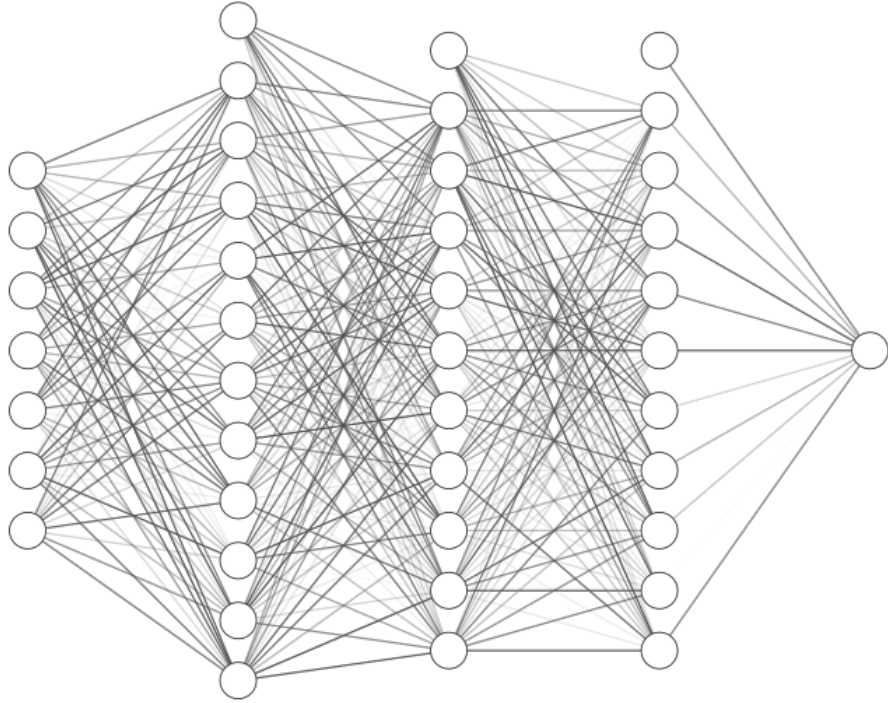
Inputs  $\in \mathbb{R}^N$

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.

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

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



## 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!



# Loss functions

*Depends on the task!*

## Mean Squared Error

Used for e.g. regression tasks

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

## Cross Entropy

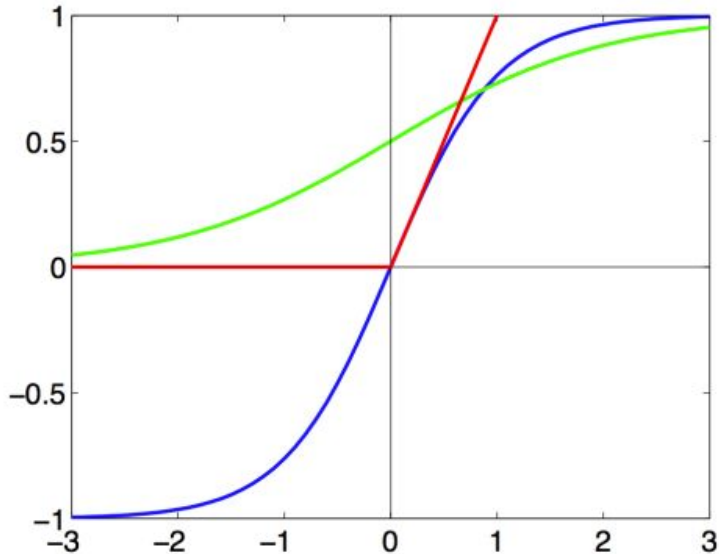
Used for e.g. classification tasks

$$\mathcal{L}_{CCE} = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^C y_{i,c} \log(\hat{y}_{i,c})$$

## Define your own!

Note: Must be differentiable for gradient descent based methods

## Common nonlinear functions



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

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

Rectified Linear Unit:

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

Sigmoid



$$y = \frac{1}{1+e^{-x}}$$

Tanh



$$y = \tanh(x)$$

Step Function



$$y = \begin{cases} 0, & x < n \\ 1, & x \geq n \end{cases}$$

Softplus



$$y = \ln(1+e^x)$$

ReLU



$$y = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$$

Softsign



$$y = \frac{x}{(1+|x|)}$$

ELU



$$y = \begin{cases} \alpha(e^x-1), & x < 0 \\ x, & x \geq 0 \end{cases}$$

Log of Sigmoid



$$y = \ln\left(\frac{1}{1+e^{-x}}\right)$$

Swish



$$y = \frac{x}{1+e^{-x}}$$

Sinc



$$y = \frac{\sin(x)}{x}$$

Leaky ReLU



$$y = \max(0.01x, x)$$

Mish



$$y = x(\tanh(\text{softplus}(x)))$$



## Implementing learning: Gradient Descent

Given:

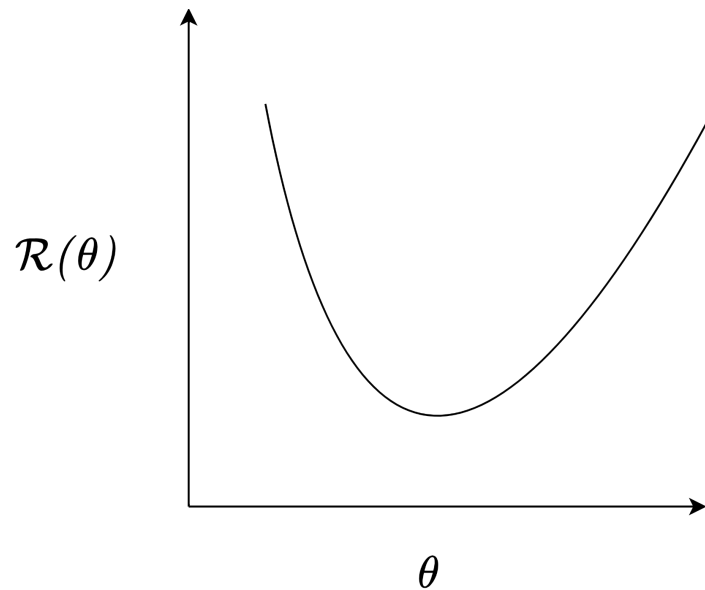
- Family of parameters  $\Theta$  (e.g. possible weights of a NN)
- Differentiable risk function  $\mathcal{R}(\theta)$

Goal:

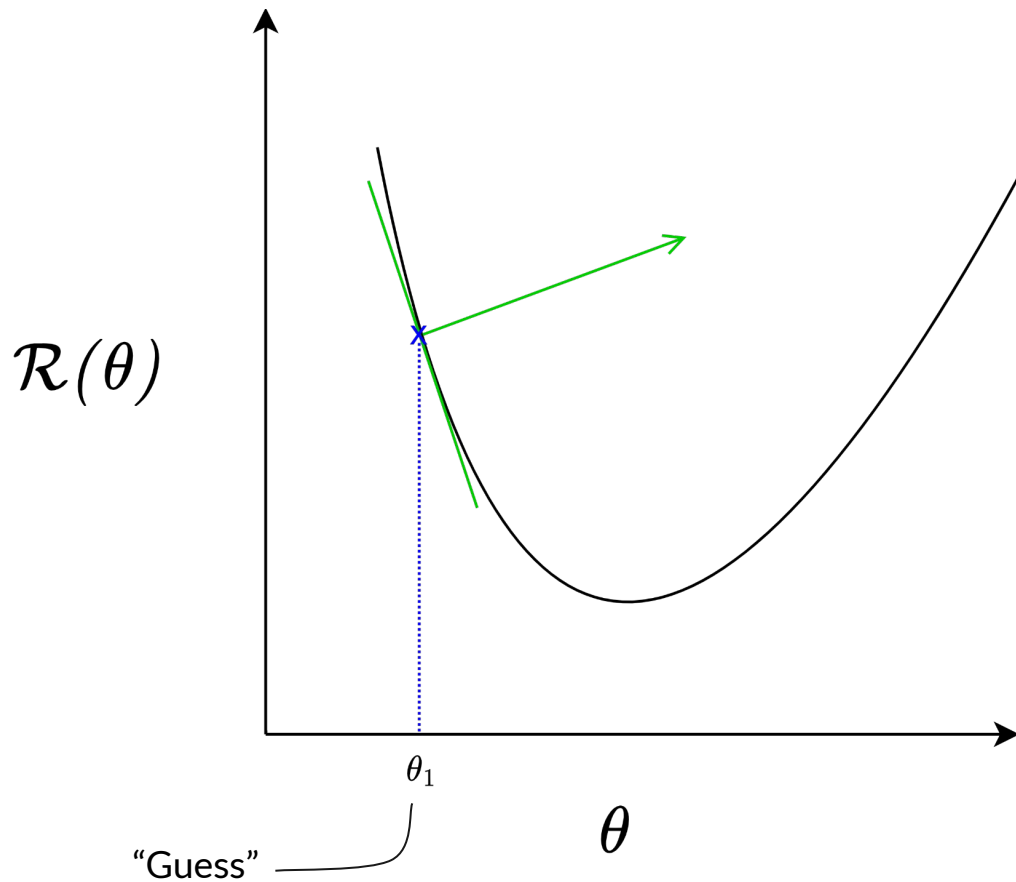
$$\theta_{opt} = \operatorname{argmin}_{\theta \in \Theta} \mathcal{R}(\theta)$$



## Backprop: Gradient descent

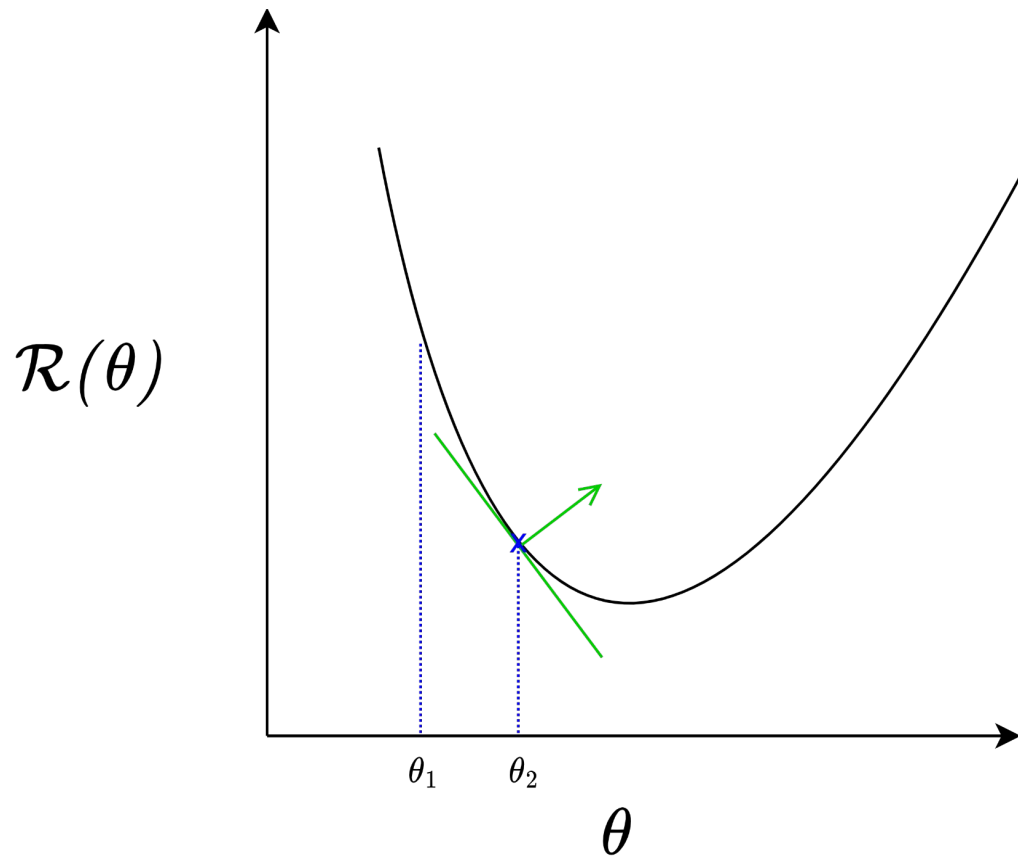




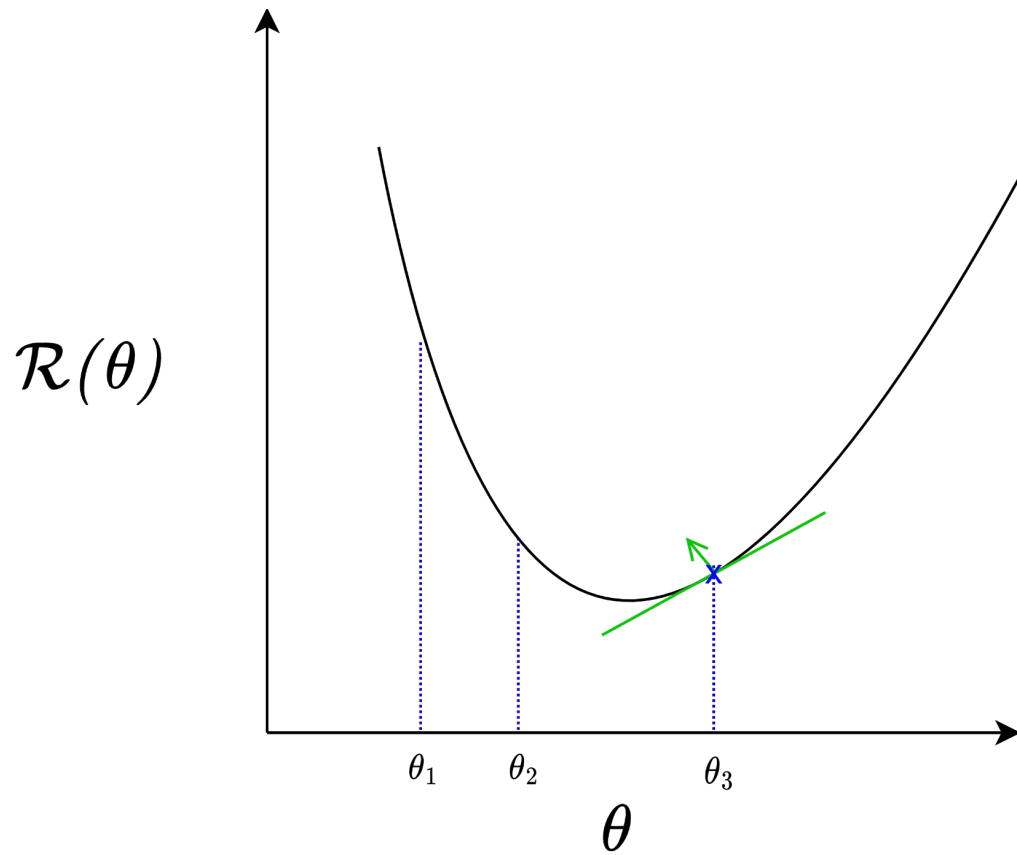


"Learning rate"

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

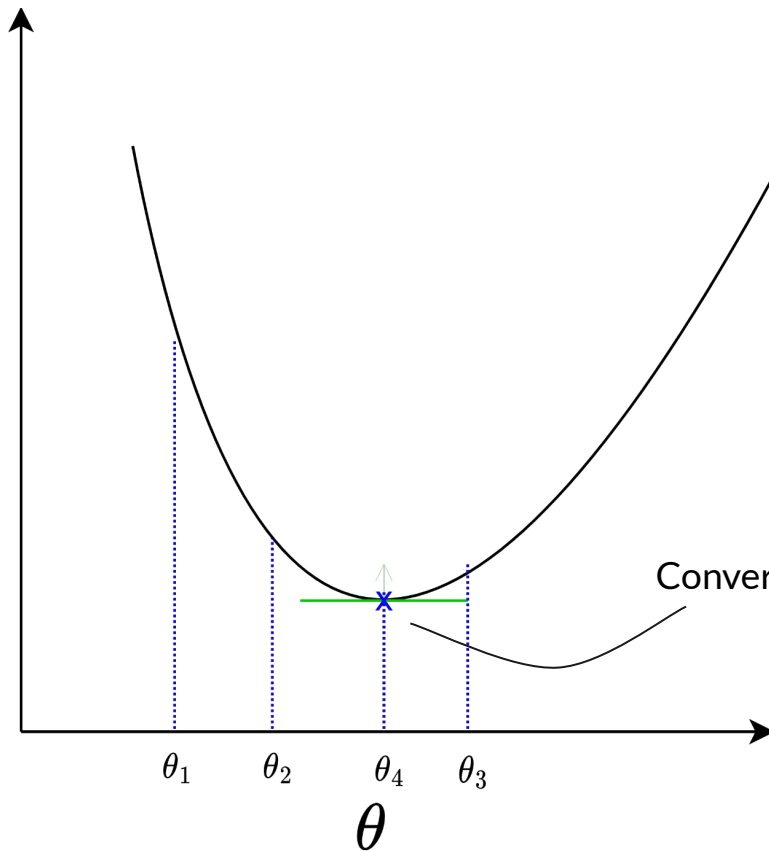


$$\theta_i = \theta_i - \alpha \frac{\partial}{\partial \theta_i} \mathcal{R}(\theta_i)$$



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

$\mathcal{R}(\theta)$



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

Converges to local minima

## Backprop: Efficient NN GD

- Goal: change optimization from  $\mathcal{O}(|\theta|^2)$  to  $\mathcal{O}(|\theta|)$
- Recall an MLP:

$$\mathcal{N}(x) = a^\ell \circ h^\ell \circ a^{\ell-1} \circ h^{\ell-1} \circ \dots \circ a^1 \circ h^1 x$$

$\mathcal{L}(\mathcal{N}(x), y) \longrightarrow \frac{\partial \mathcal{L}(\mathcal{N}(x), y)}{\partial w_i} \longrightarrow w_i \leftarrow w_i - \eta \frac{\partial \mathcal{L}(\mathcal{N}(x), y)}{\partial w_i}$

- Use the chain rule to compute the derivatives from output to input  
“Backpropagation of errors”

$$\frac{d\mathcal{L}}{dw_l} = \frac{d\mathcal{L}}{dh_L} \cdot \frac{dh_L}{dh_{L-1}} \cdot \dots \cdot \frac{dh_l}{dw_l} \Rightarrow \frac{d\mathcal{L}}{dw_l} = \frac{d\mathcal{L}}{dh_l} \cdot \frac{dh_l}{dw_l}$$

Gradient of loss w.r.t. the module output
Gradient of a module w.r.t. its parameters

## Backprop: Efficient NN GD

*Apologies for different notation*

- Goal: change optimization from  $\mathcal{O}(|\theta|^2)$  to  $\mathcal{O}(|\theta|)$
- Recall an MLP:

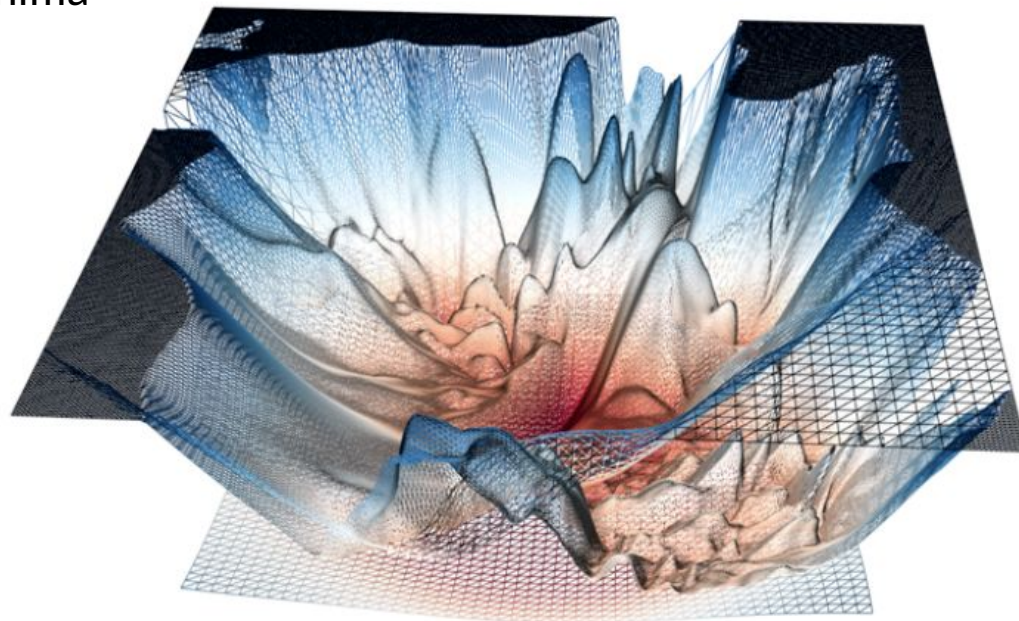
$$\mathcal{N}(x) = a^l \circ h^l \circ a^{l-1} \circ h^{l-1} \circ \dots \circ a^1 \circ h^1 x$$

$$\mathcal{L}(\mathcal{N}(x), y) \longrightarrow \frac{\partial \mathcal{L}(\mathcal{N}(x), y)}{\partial w_i} \longrightarrow w_i \leftarrow w_i - \eta \frac{\partial \mathcal{L}(\mathcal{N}(x), y)}{\partial w_i}$$

- Use the chain rule to compute the derivatives from output to input  
“Backpropagation of errors”

A “real” loss landscape:

- Many (many many) local minima
- Saddle points



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



# Optimizers

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

- Perform backprop on errors of *batches* of training samples instead of all at once
- Reduces the number of expensive backward passes
- Helps with getting out of local minima (due to higher gradient noise)

Optimizers determine exactly how backpropagation is implemented

- Stochastic Gradient Descent (most common)
- RMSProp
- Adaptive Momentum Estimation (“Adam”)





# Optimizers

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

- Perform backprop on errors of *batches* of training samples instead of all at once
- Reduces the number of expensive backward passes
- Helps with getting out of local minima (due to higher gradient noise)

Optimizers determine exactly how backpropagation is implemented

- Stochastic Gradient Descent (most common)
- RMSProp
- Adaptive Momentum Estimation (“A’dam”)





## ML Training paradigms (a selection)

- **Supervised**
  - Train a model with explicit input-output pairs
- **Unsupervised**
  - Learns “patterns” from unlabelled data
- **Semi-supervised learning**
  - Learn a few things with input-output pairs, relate them to patterns learnt unsupervised
- **Reinforcement Learning**
  - Learn an optimal “policy” that gives you the best action to take at any given state space by taking random actions and learning through positive or negative reinforcement.
- **Evolution**
  - Optimize parameters through (Darwinian) evolution; e.g. genetic algorithms.

---

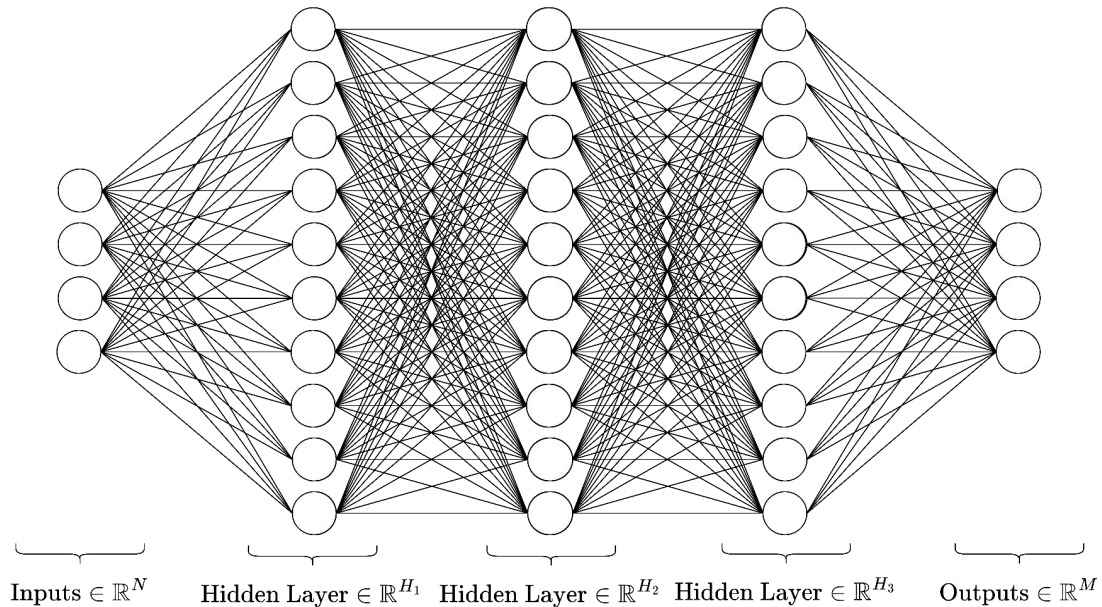
# Types of Neural Networks

# Multi-layer Perceptrons

Useful for **static** input-output relations

**More hidden layers ~ better approximation** of more complicated functions

**Quick** to design and implement

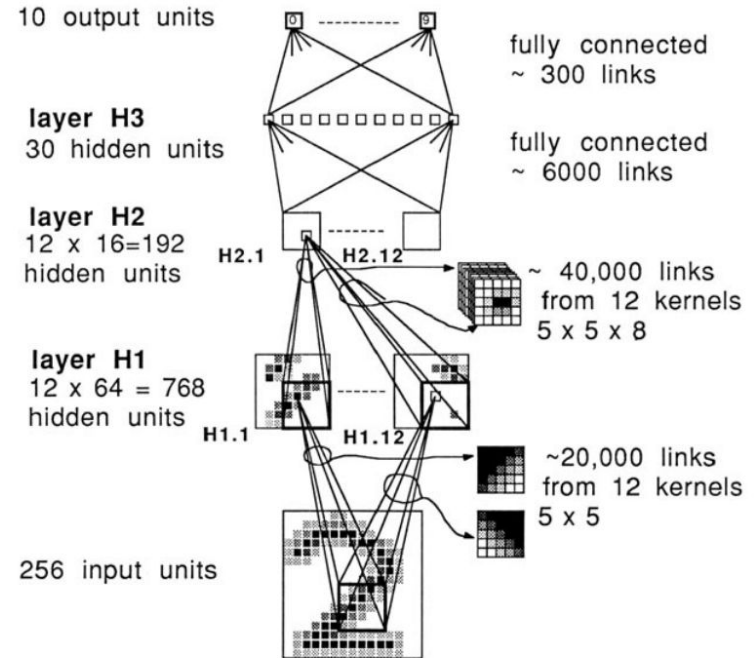


# Convolutional Neural Networks

Learn “kernels”, i.e. ‘tensors’ (multi-dimensional arrays) 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



# Recurrent Neural Networks

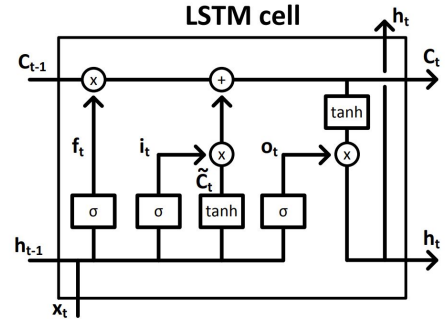
Outputs go back and forth between neurons (loops exist in the graphs)

Approximates dynamical systems

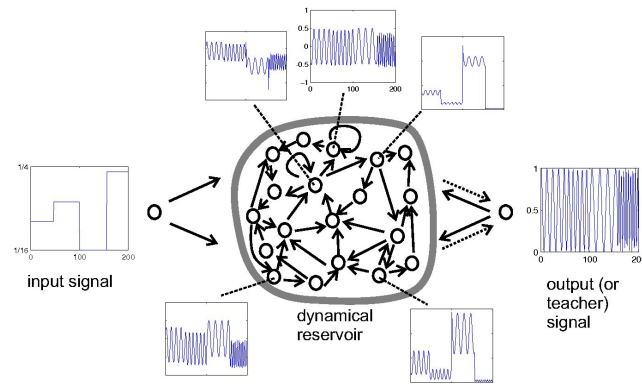
- Any time-based function
- Any data that can be modelled as being “ordered”

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

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



An LSTM cell schematic. Adapted from: [doi.org/10.4233/uuid:dc73e1ff-0496-459a-986f-de37f7f250c9](https://doi.org/10.4233/uuid:dc73e1ff-0496-459a-986f-de37f7f250c9)



Echo state network schematic. Adapted from [www.scholarpedia.org/article/Echo\\_state\\_network](http://www.scholarpedia.org/article/Echo_state_network)

# Graph Neural Networks

Models *any* system that can be modelled as a graph

Learns relations between nodes, edges, global properties

Accounts for **relational inductive bias**, **node invariance**, others

Used in e.g. image segmentation, chemistry and pharmacy models, NLP, hierarchically-related data

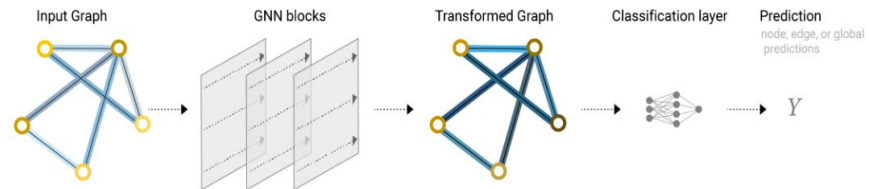


Image adapted from this excellent intro to GNNs: <https://distill.pub/2021/gnn-intro/>



## Geometric DL Models

Models or uses 'manifold embeddings' on non-Euclidean domains like graphs, meshes and manifolds - *geometric* inductive bias

Captures **various forms of invariance and equivariance** (e.g. Grids, Groups, Graphs, Geodesics, and Gauges)

Used in e.g. 3D object recognition, protein structures, medical image, etc.



---

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



# Universal Approximation Theorem

***Theorem (schematic).** Let  $\mathcal{F}$  be a certain class of functions  $f : \mathbb{R}^K \rightarrow \mathbb{R}^M$ . 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

## 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

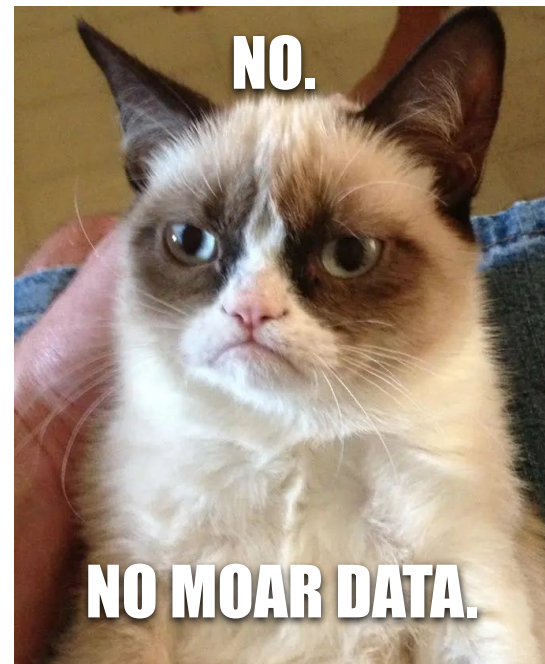


<https://lasp.colorado.edu/home/minxss/2016/07/12/minimum-mission-success-criteria-met/>

## Limits of NNs

- > No causal relations possible (yet)
- > Very data hungry - “Garbage in, garbage out”
- > Often expensive to train (depending on size)
- > **Nonextensible** and **specialized** to a range and task
  - Add one more neuron → needs fine-tuning
  - Undefined behaviour on out-of-domain test examples

*Note on specialization: rf. ‘Foundation Models’*



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

---

# Training tips



# Overfitting & Underfitting

The real troublemakers in ML in general!

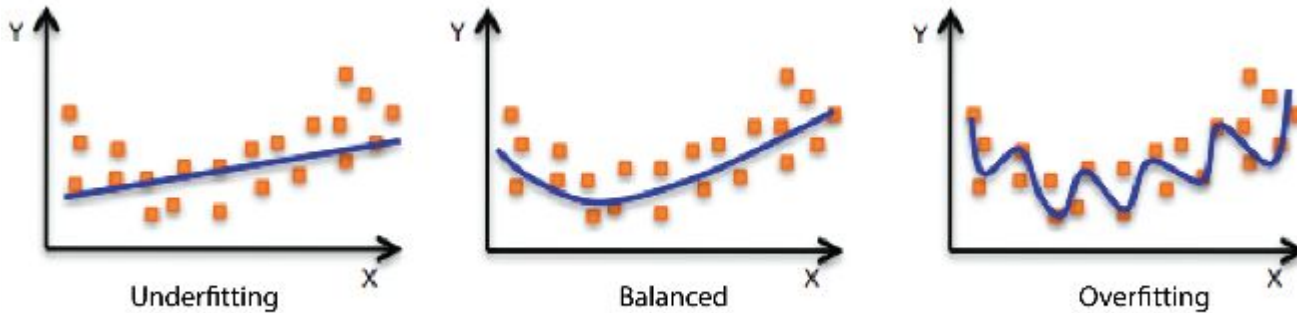
**Underfitting:** When the model fits the training data not well enough

- Empirical risk is high, actual risk is high
- Training loss is high, testing loss is not optimal

**Overfitting:** When the model fits the training data *too* closely (incl. noise)

- Empirical risk is low, actual risk is high
- Training loss is low, testing loss is not optimal
- e.g. An  $D$ -degree polynomial can fit  $D-1$  training points with zero error

# Overfitting & Underfitting



More complex models (e.g. more layers, neurons per layer) -> higher likelihood of overfitting



# Validation

Split your training set into two!

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

Train Set

Test Set (unseen)

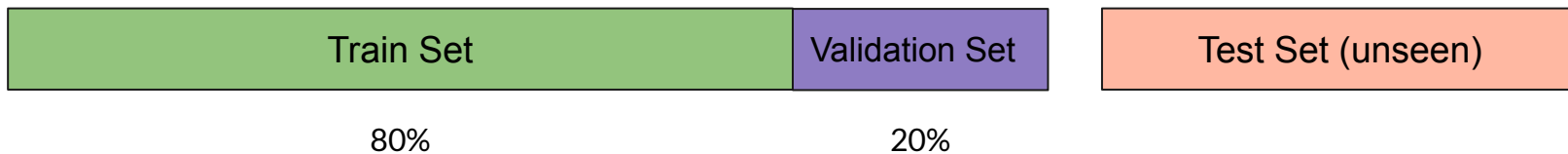




# Validation

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



Train Set

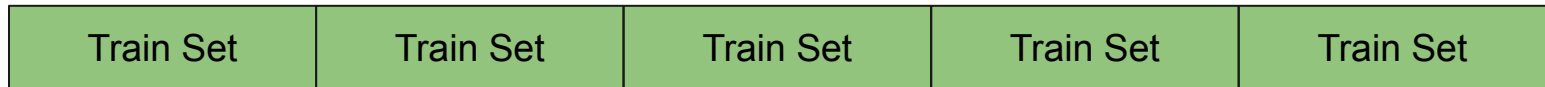


## ***k*-fold Cross-validation**

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)



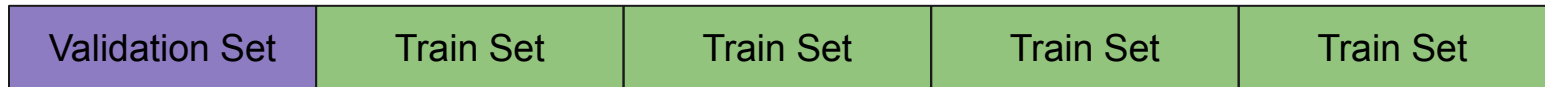


## ***k*-fold Cross-validation**

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)



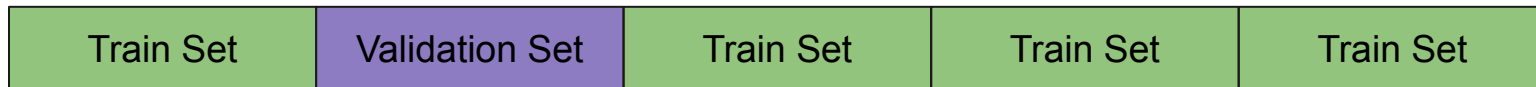


## ***k*-fold Cross-validation**

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)



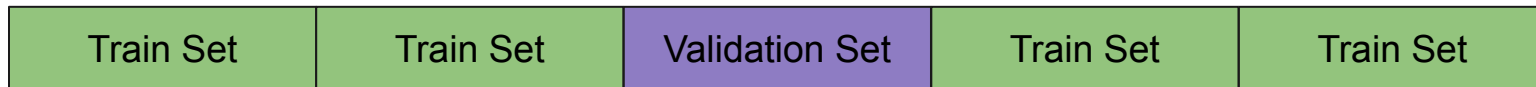


## ***k*-fold Cross-validation**

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)



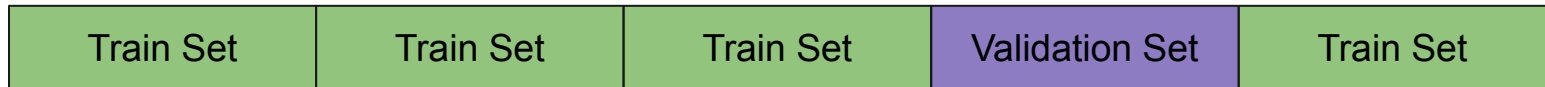


## ***k*-fold Cross-validation**

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)



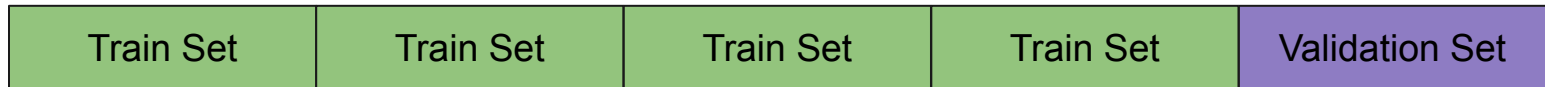


## ***k*-fold Cross-validation**

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)



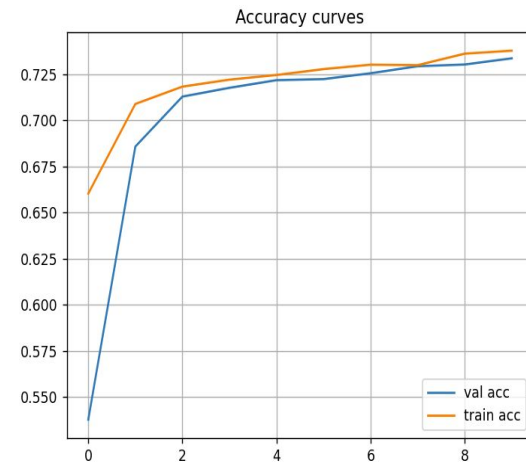
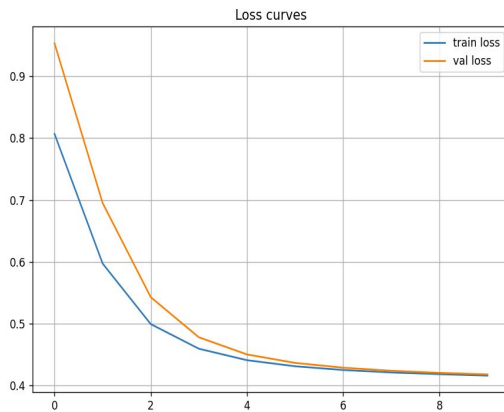
Result = average over all validation passes





# Training curves

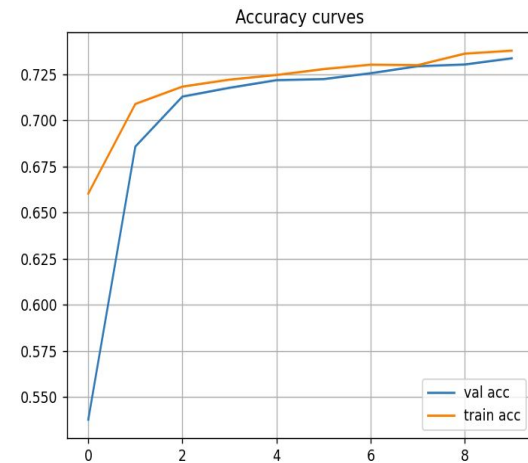
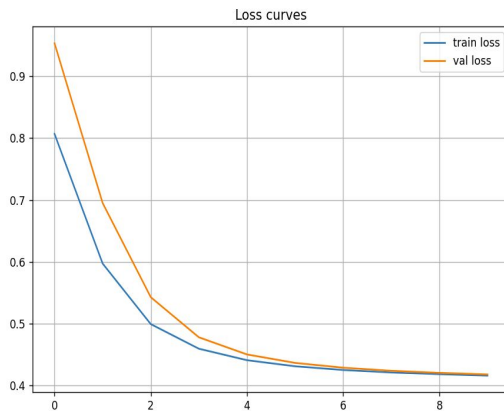
Important to plot!





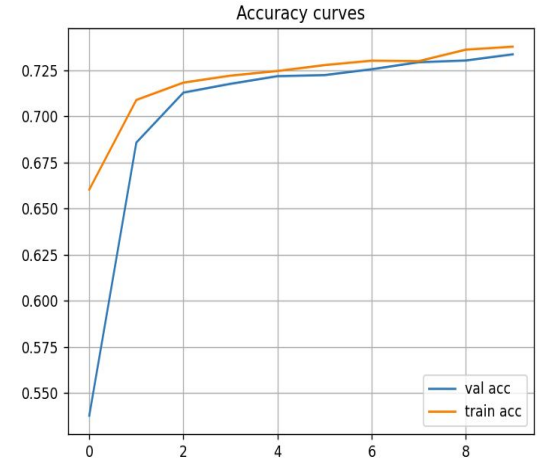
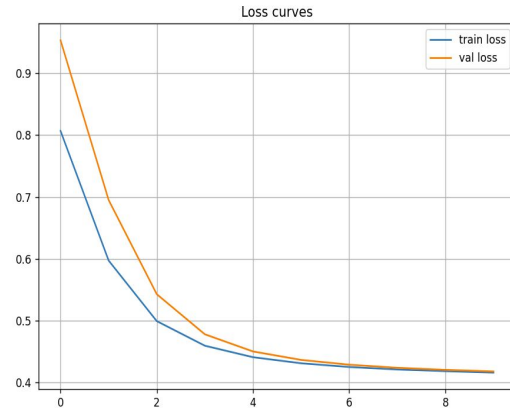
# Training curves

Important to plot!!!!



# Training curves

Important to plot!!!!

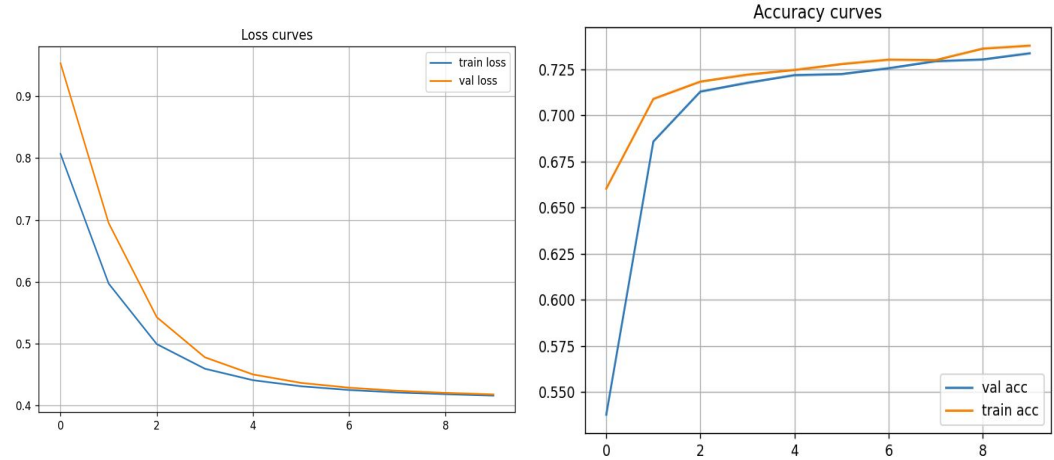


Shows if and how fast your model is learning on task-relevant metrics

- 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

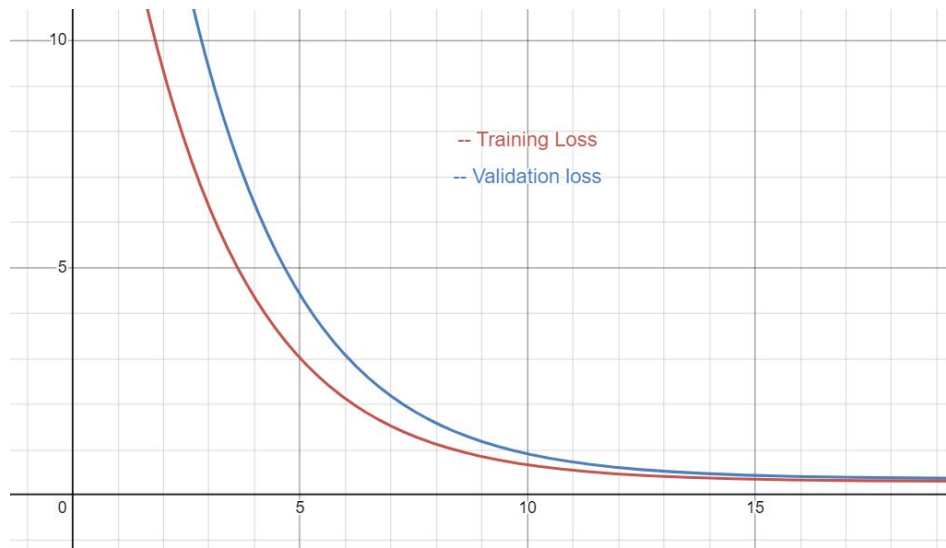
- e.g. loss, accuracy, AUC, F1 score
- Plot scores over training epochs

May indicate potential over and underfitting



## Reading training curves

*If*



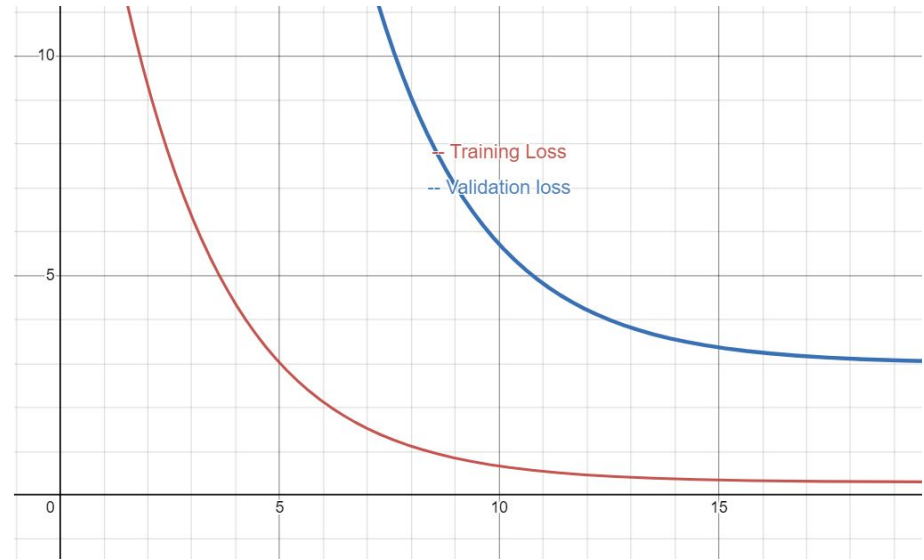
validation loss > training loss

then often the model is *good!*

Low loss == Better

## Reading training curves

*If*



validation loss >> training loss

then often the model is *overfitting*

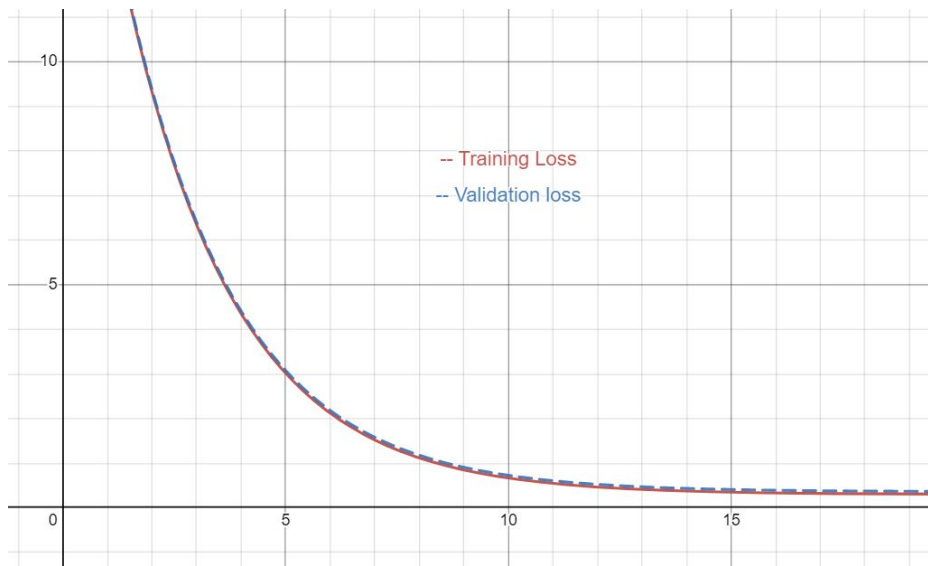
Low loss == Better

## Reading training curves

*If*

validation loss  $\sim$  training loss

Low loss == Better

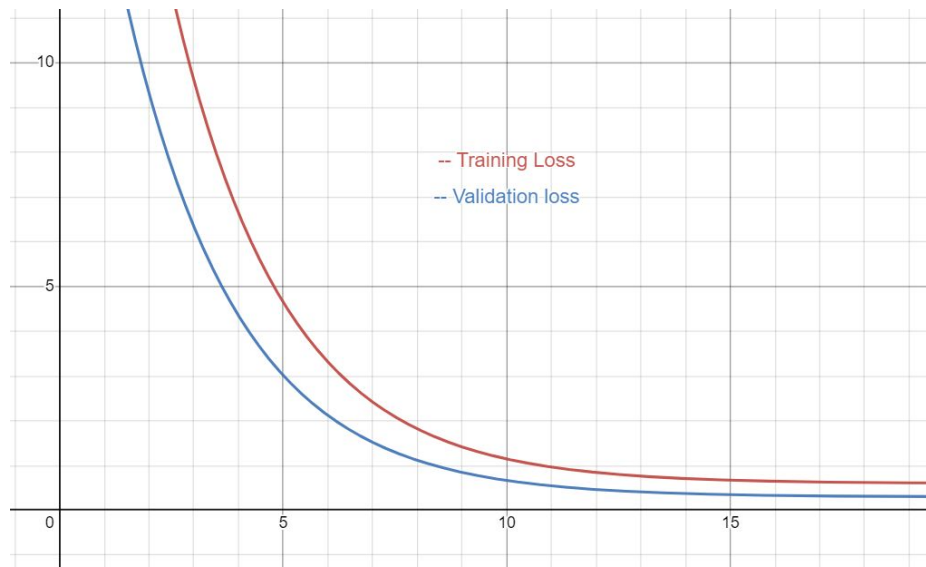


then often the model is *underfitting*



## Reading training curves

*If*



validation loss < training loss

then something is very wrong, or *totally expected!*

Low loss == Better





# Regularization

- L1/L2 Regularization

Added losses:

$$\lambda \cdot \sum_{i=1}^d |w_i| \qquad \lambda \cdot \sum_{i=1}^d w_i^2$$

- Dropout (on when training, off when testing/deploying)
- Early stopping

# 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

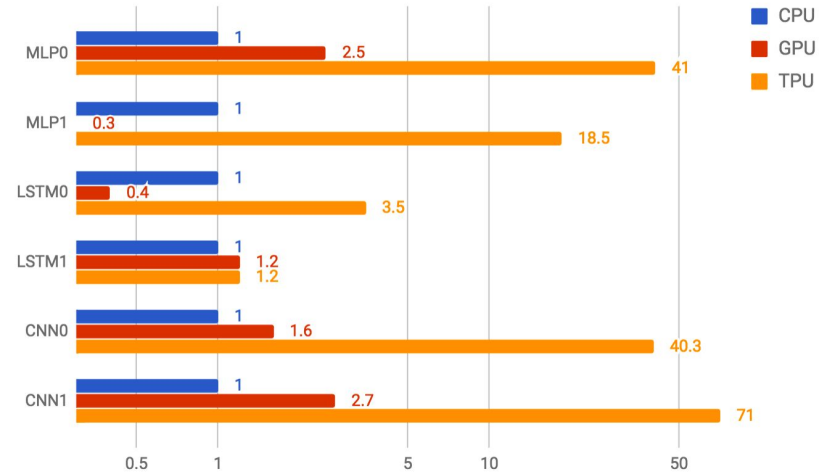


Image from

<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
- Foundation models, self-supervised learning
- AutoMLs: a tool to automatically generate good ML models for a task

## Explainable AI (XAI)

- Explainable+interpretable+controllable models
- Human-like and human-understandable reasoning
- Neurosymbolic:  
Semantic losses, logic tensors, symbolic regression, conceptors...

## Others: Physics Informed NNs

Neural ODEs, PDEs

Quantum, Geometric DL



# Frontiers

## *Deep learning*

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

## *Explainable AI (XAI)*

- Explainable+interpretable+controllable models
- Human-like and human-understandable reasoning
- Neurosymbolic:  
Semantic losses, logic tensors, symbolic regression, conceptors...

## *Others: Physics Informed NNs*

Neural ODEs, PDEs

Quantum, Geometric DL

*Me: Neurosymbolic-constrained  
multi-agent reinforcement learning*



# Reference Material

- Hastie, T., Tibshirani, R., Friedman, J. H., & Friedman, J. H. (2009). **The elements of statistical learning**
- Bishop, C. M. (2006). **Pattern recognition and machine learning**
- LeCun, Y., Bengio, Y., & Hinton, G. (2015). **Deep learning**
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). **Deep learning**
- Chollet, F. (2021). **Deep learning with Python**
- Géron, A. (2022). **Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow**
- Erdmann, M., Glombitza, J., Kasiieczka, G., & Klemradt, U. (2021). **Deep learning for physics research.**
- Bronstein, M. M., Bruna, J., Cohen, T., & Veličković, P. (2021). **Geometric deep learning**
- Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). **Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations**
- Kipf, T. N., & Welling, M. (2016). **Semi-supervised classification with graph convolutional networks**
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). **Attention is all you need**
- Kingma, D. P., & Welling, M. (2013). **Auto-encoding variational bayes**. *arXiv preprint arXiv:1312.6114*.
- UvA **Deep Learning 1 (Graduate) Tutorials** <https://uvadlc-notebooks.readthedocs.io/en/latest/> (see also the DL2 lectures)
- University of Groningen **ML and NN lecture notes** <https://www.ai.rug.nl/minds/teaching/ln/>
- ...many many more