An Introduction to Neural Networks

Satchit Chatterji
BSc Artificial Intelligence
University of Groningen

satchit.chatterji@gmail.com

university of groningen

Why you should consider Neural Networks

Satchit Chatterji
BSc Artificial Intelligence
University of Groningen

satchit.chatterji@gmail.com

The short answer

- They’re useful!

The short answer

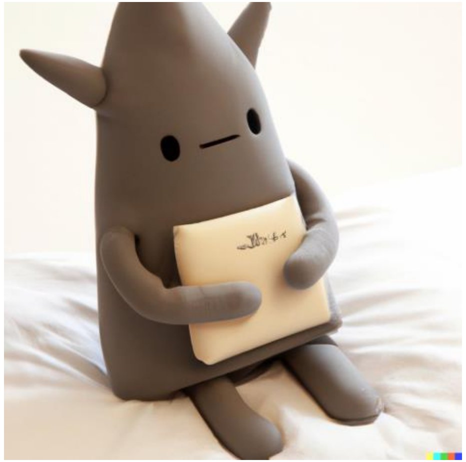
- They’re useful!
- They’re fast!

The short answer

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

The short answer

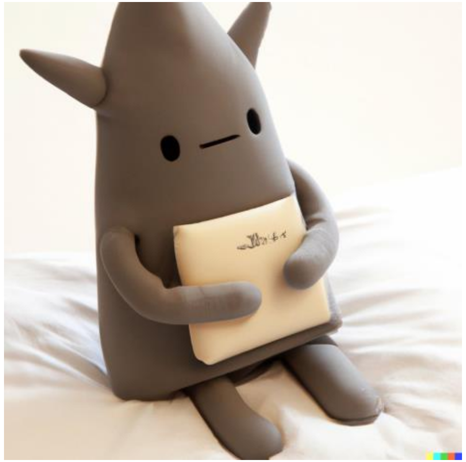
- They’re useful!
- They’re fast!
- They’re (now) easy to implement!
- They’re cute!



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

The short answer

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



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

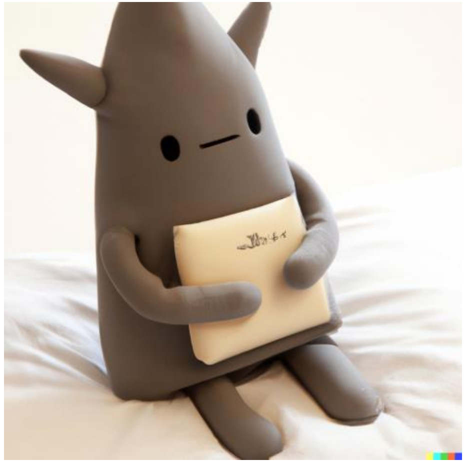
bostondynamics.com

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

bostondynamics.com

Introduction to the introduction

Goals of this lecture:

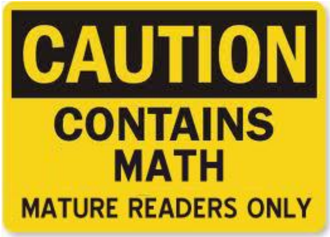
The whats, hows, whys, whichs and wheres

- 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 live 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 live 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:

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

Introduction to *Supervised* Machine Learning

Given: Input-output examples of the form:

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

Introduction to *Supervised* Machine Learning

Given: Input-output examples of the form:

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

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

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

Introduction to *Supervised* Machine Learning

Given: Input-output examples of the form:

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

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

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

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

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

Introduction to *Supervised* Machine Learning

Given: Input-output examples of the form:

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

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

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

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

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

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*.

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

Aim: Learn a function with low “risk”

Risk: What we want to minimize

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

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

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

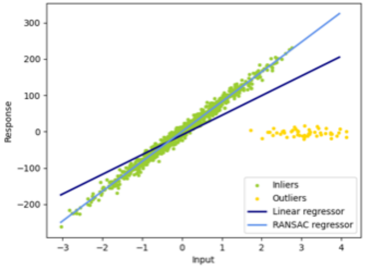
(for a “candidate” model ***h***, averaged over ***N*** training examples)

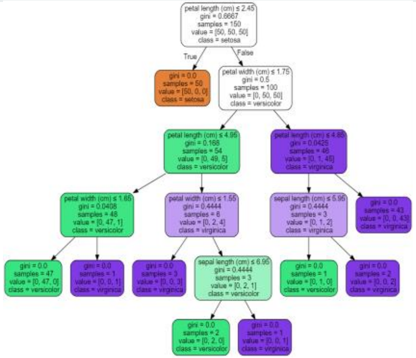
$$R^{\mathrm{emp}}(h) = 1/N \sum_{i=1}^{N} L(h(\mathbf{x}_i), \mathbf{y}_i)$$

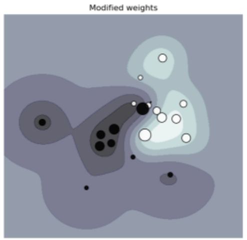
Slide adapted from Jaeger, H. (2022) *Neural Networks* Lecture Notes, https://www.ai.rug.nl/minds/uploads/LN_NN_RUG.pdf

Common Approaches

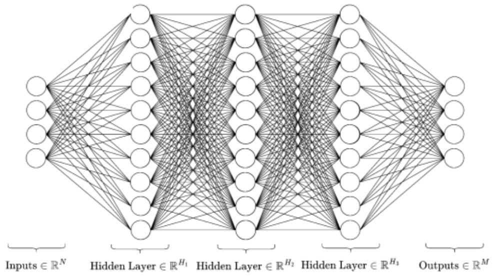
- Linear/Polynomial/Logistic Regression
- (Boosted) Decision trees
- Support Vector Machines
- Naive Bayes
- **Neural Networks**!
- …



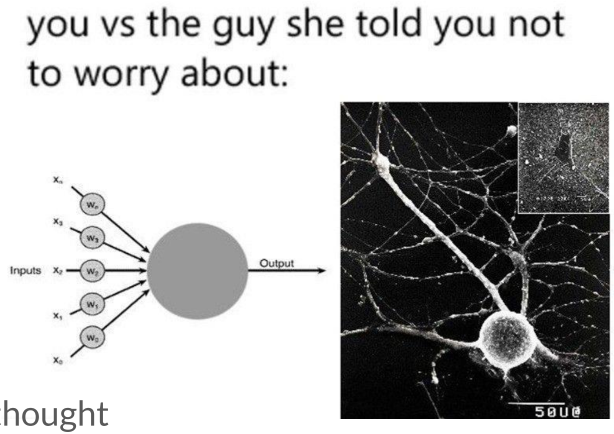




$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)}$$



Images of regressions, decision tree, and SVM from scikit-learn.org



Source: linkedin.com/company/deeplearningai

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*

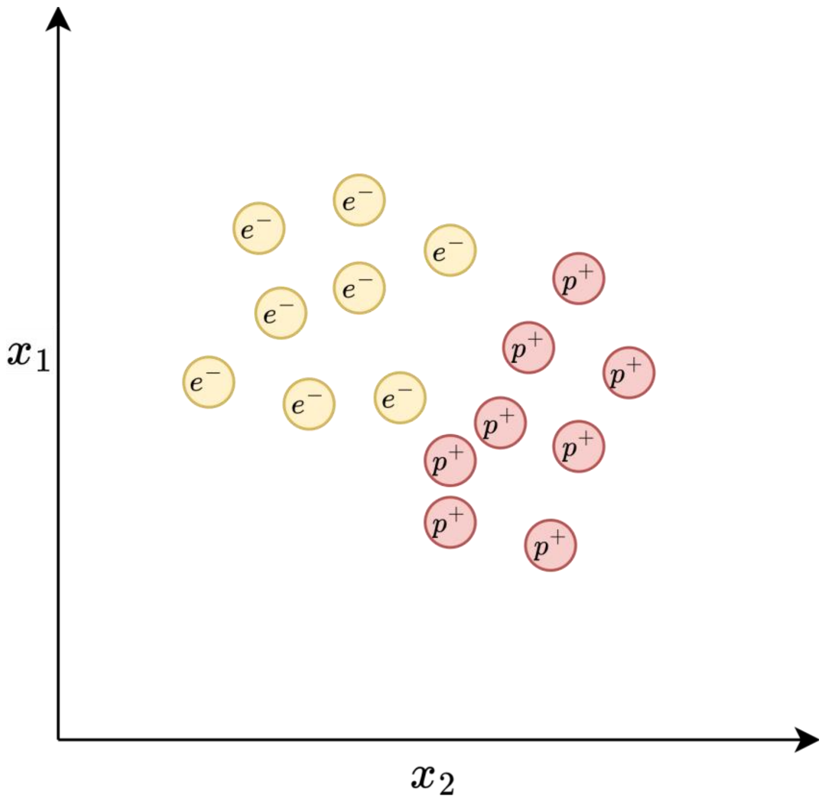
Rumelhart, Hinton & Williams (1986)

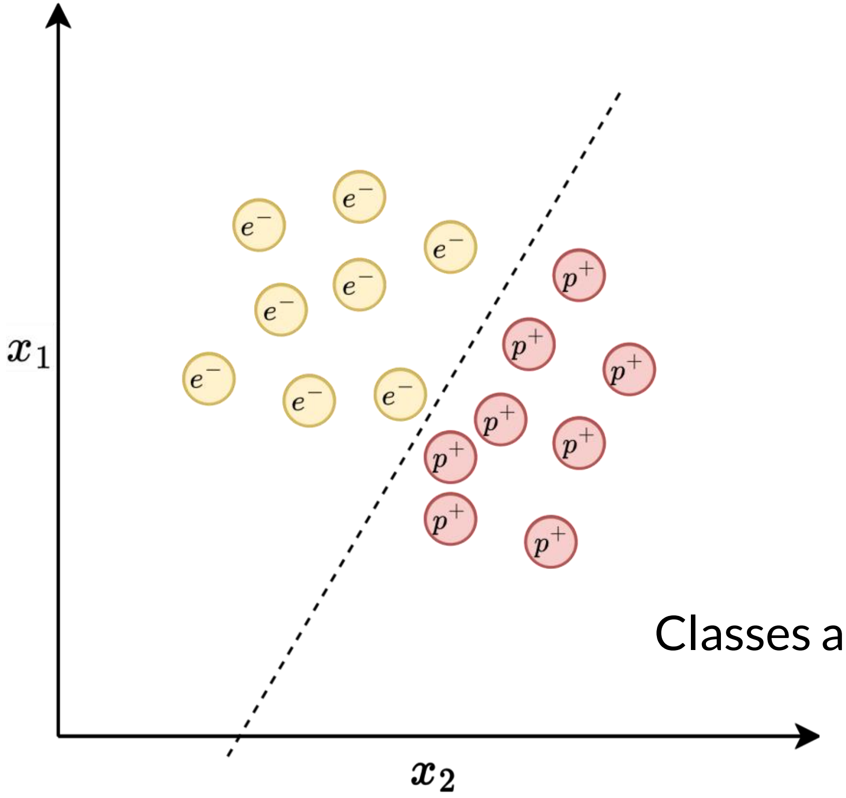
Multi-layer perceptrons and error backpropagation (learning principle)

Modern:

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

Building a Neural Network From Scratch (mathematically)


x_1
x_2
e^-
p^+



Classes are *"linearly separable"*

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

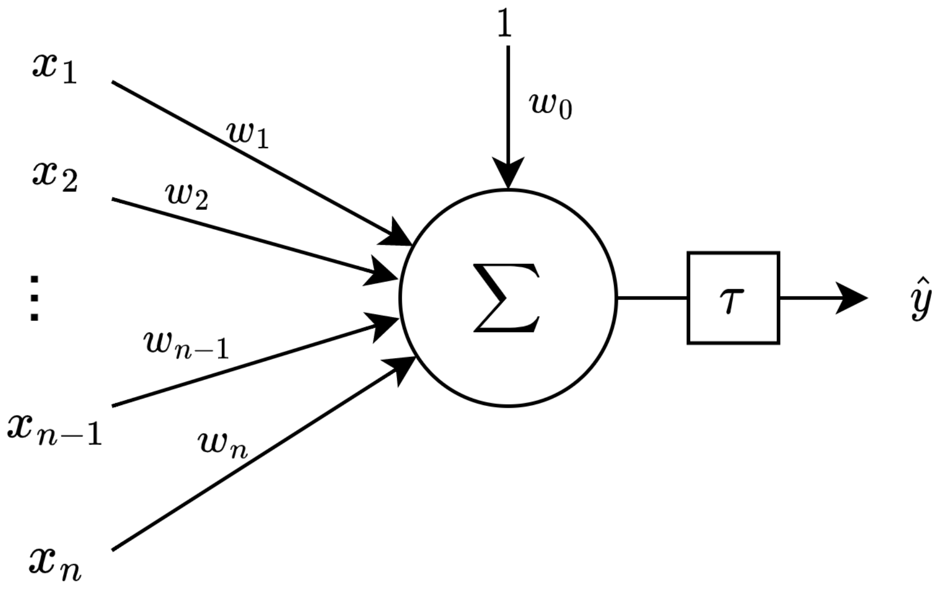
$w_i \leftarrow$ Coefficients

$x_i \leftarrow$ Variables

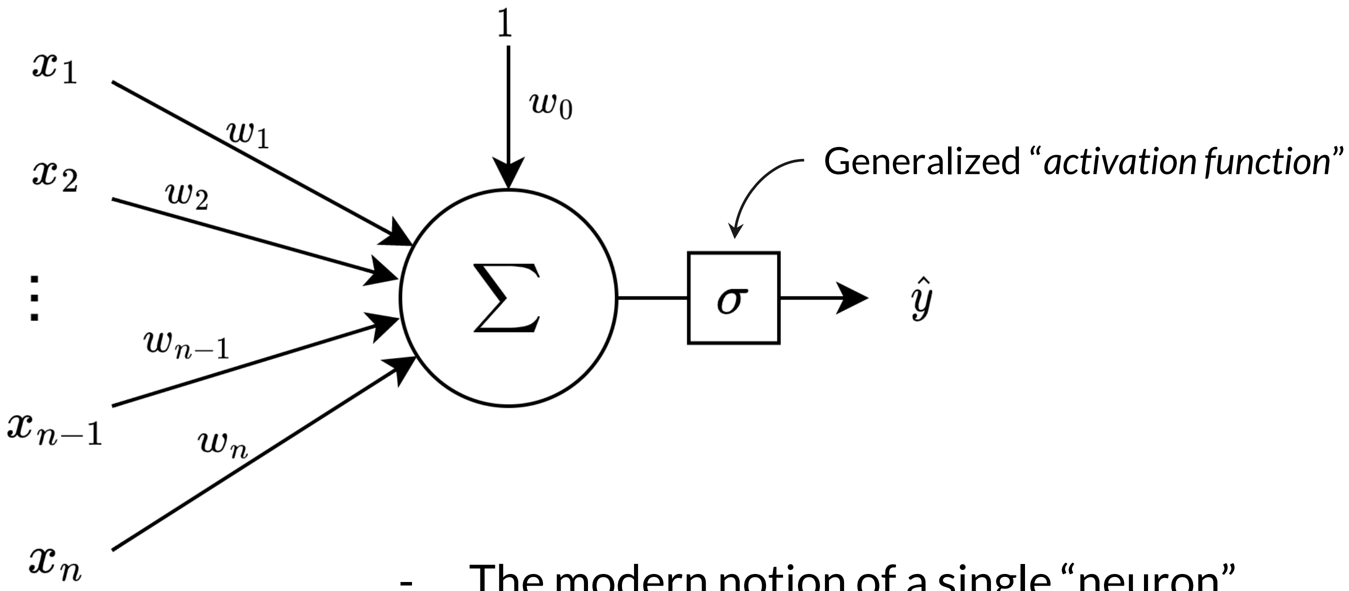
$$\hat{y} = \tau(w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n)$$

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

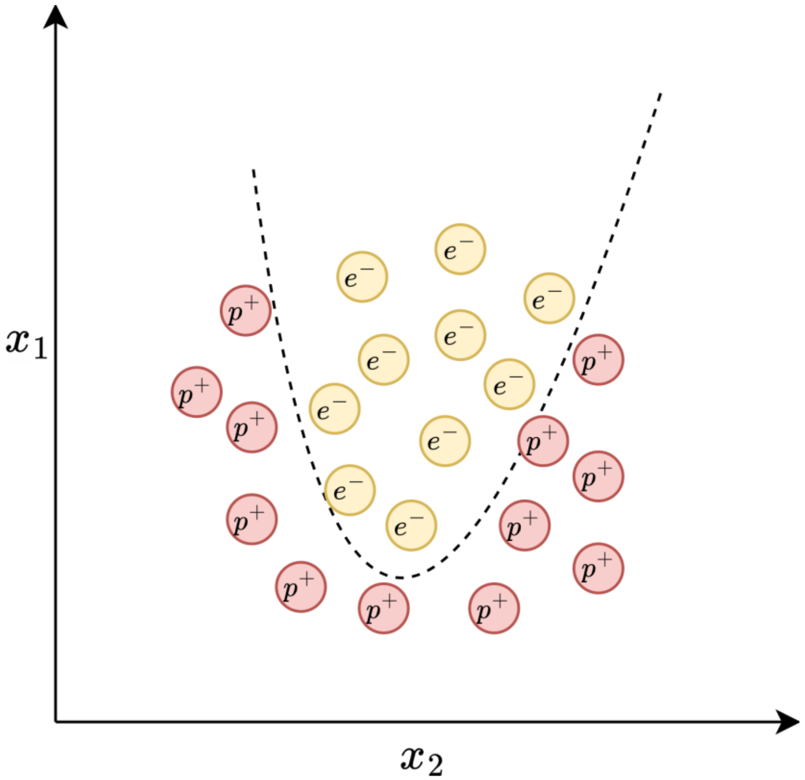
$$\hat{y} = \tau(w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n)$$



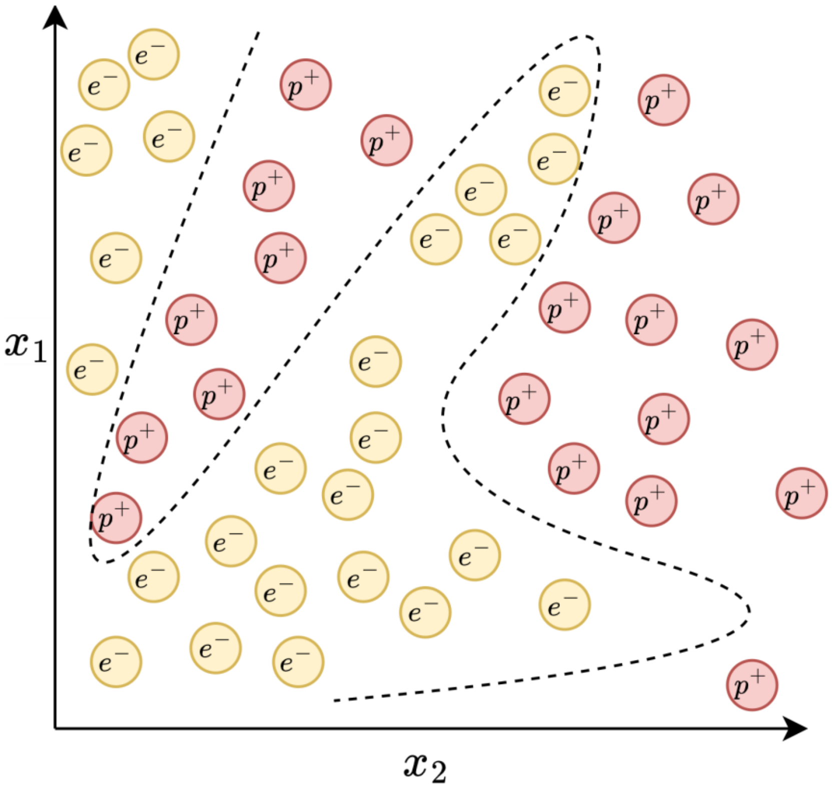
The “Perceptron”

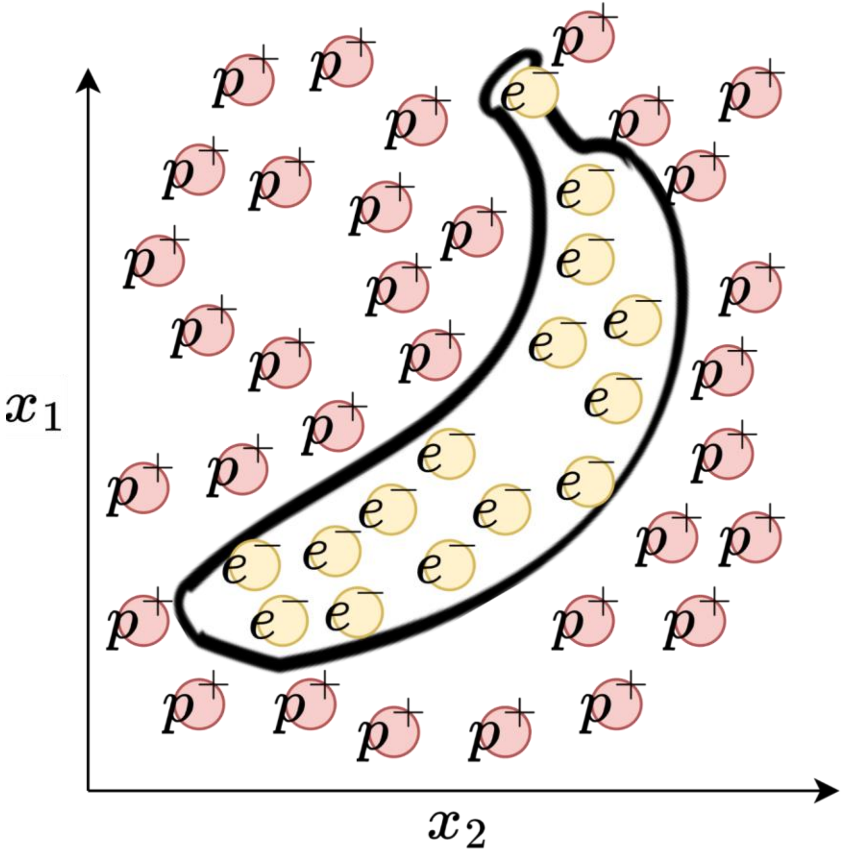


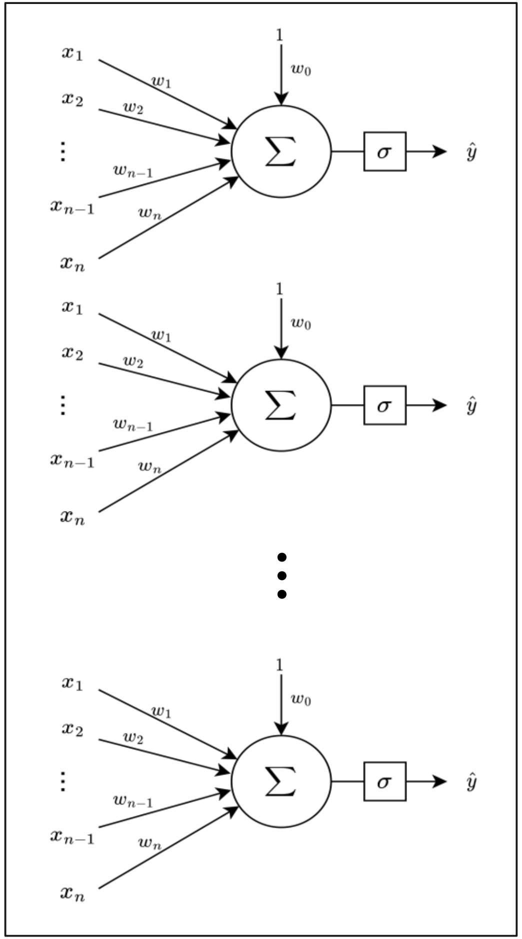
- The modern notion of a single “neuron”
- BUT: Only works on linearly separable classes



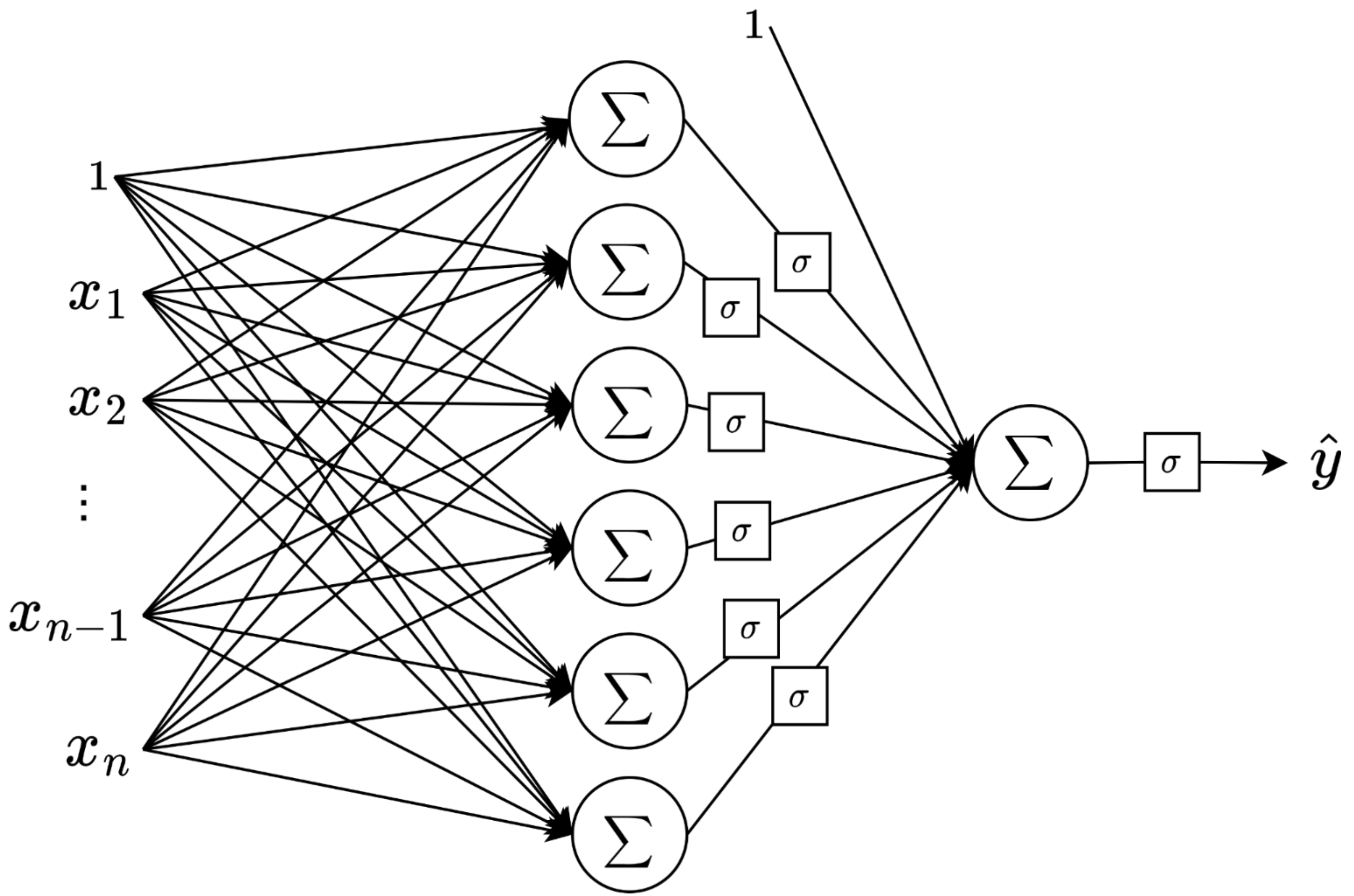
x_1
x_2
p^+
e^-


x_1
x_2
e^-
p^+

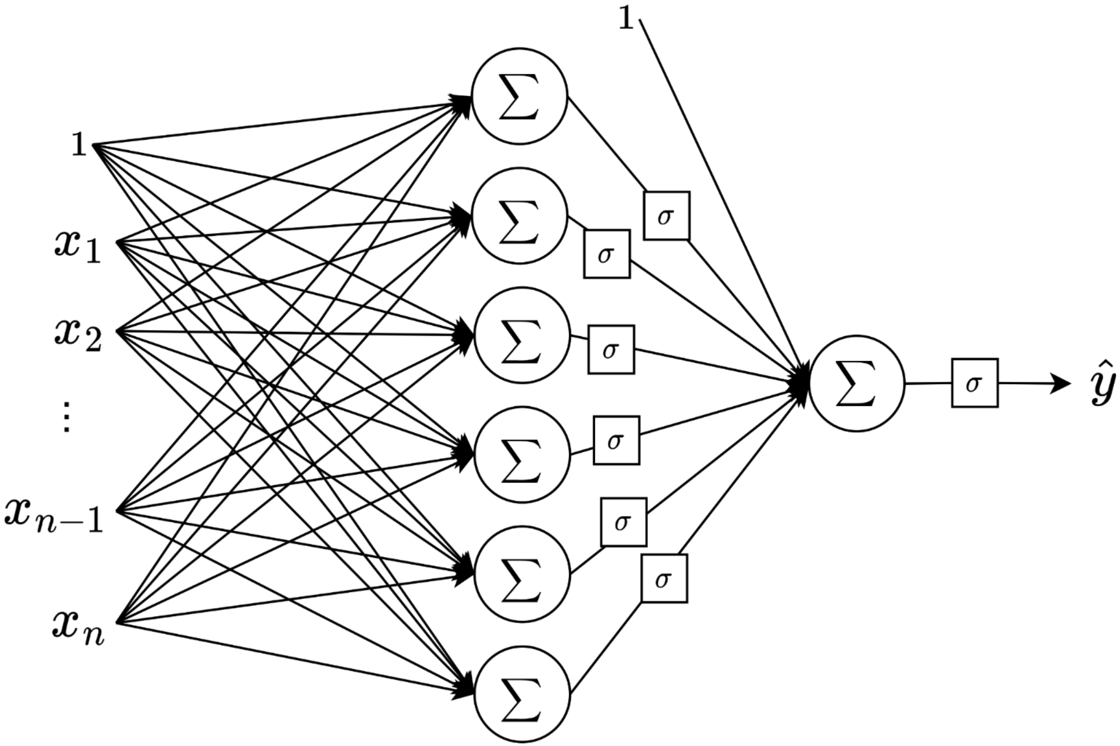

x_1
x_2
p^+
e^-



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


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

Multi-layer Perceptrons (MLPs)



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

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

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

“Activation” *“Bias”*

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

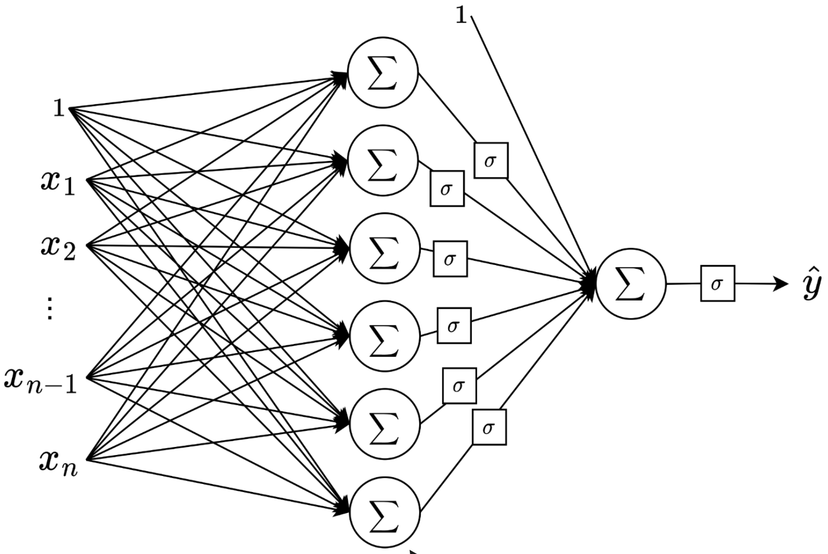
“Activation function”

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

“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**

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

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

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

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

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

$$\vdots$$

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

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

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

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

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

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

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

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

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

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

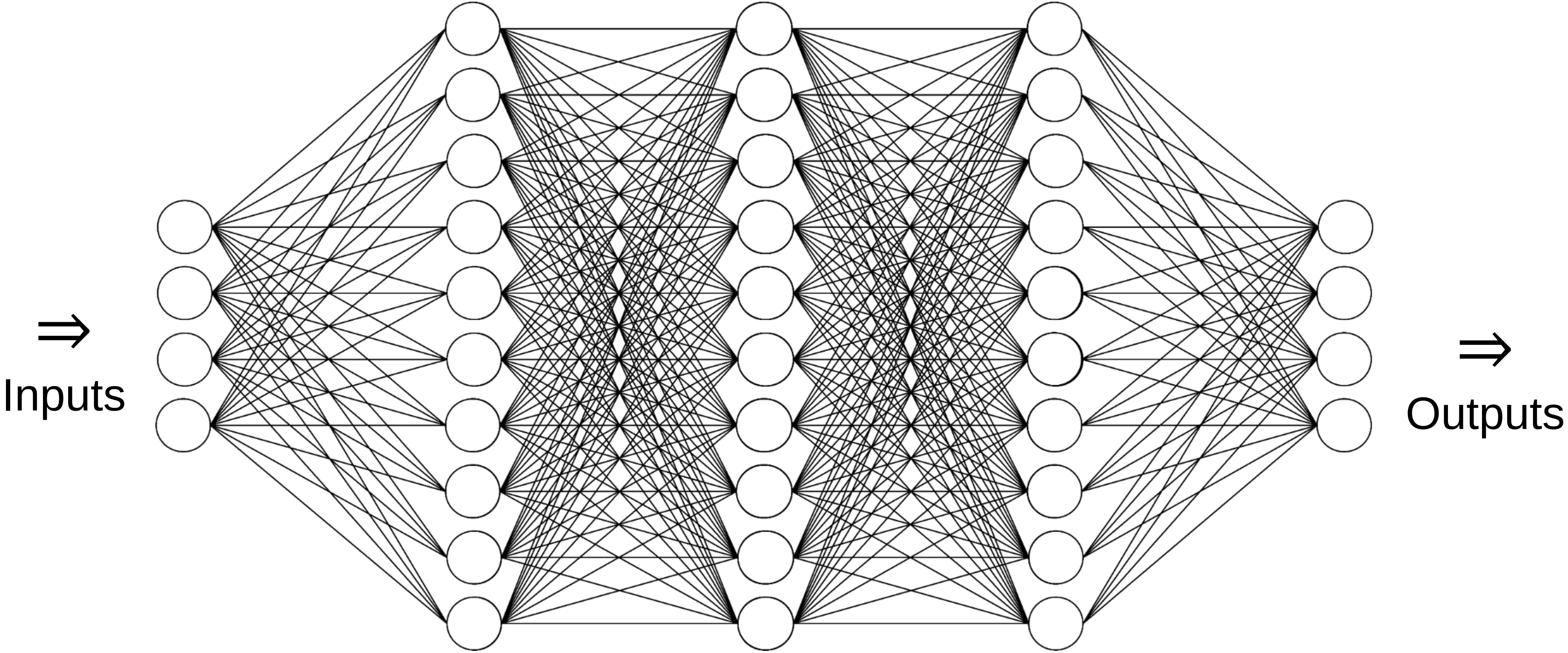
$$\hat{y} = \sigma(w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n)$$

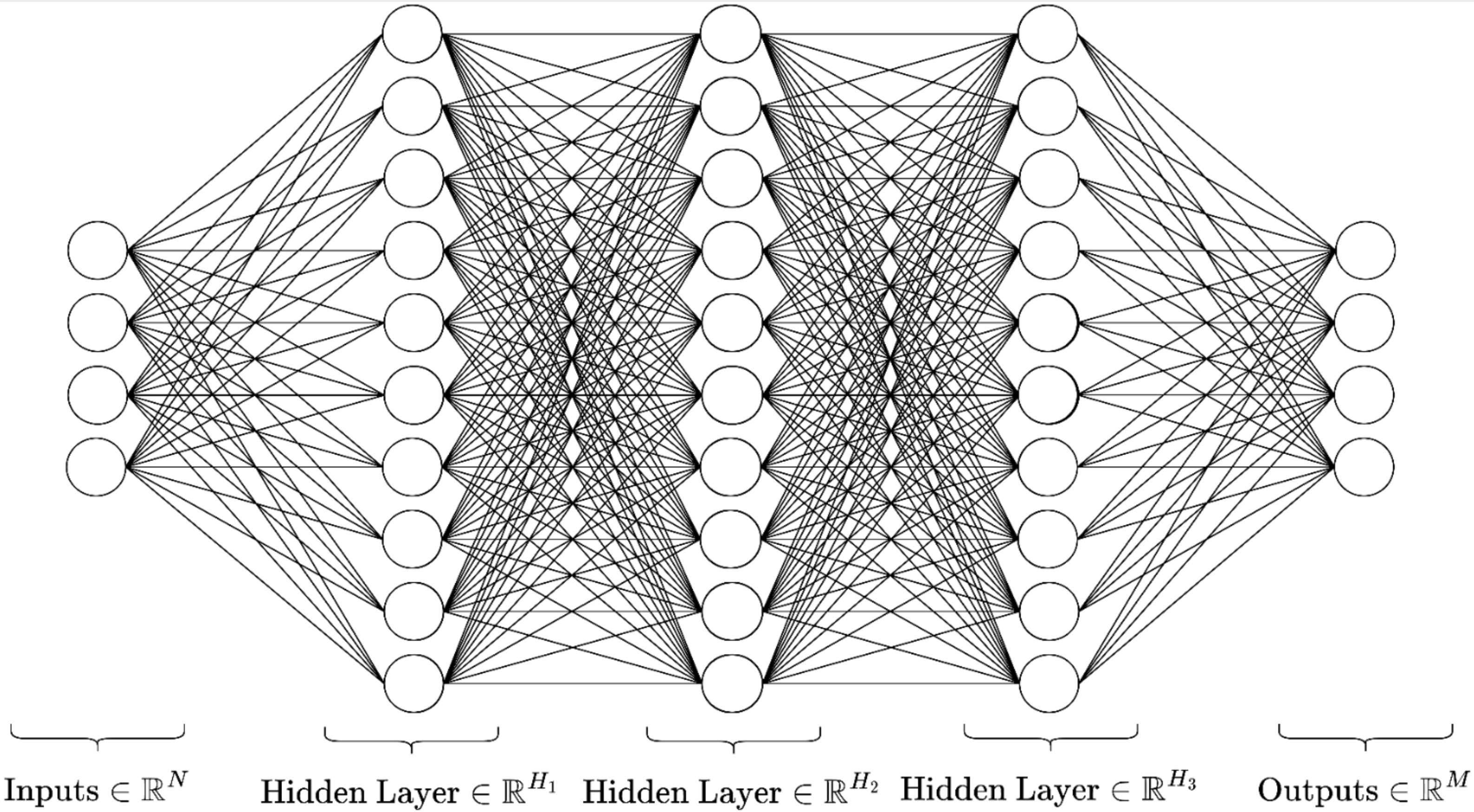
$$o^L = W\vec{x} + \vec{b}$$

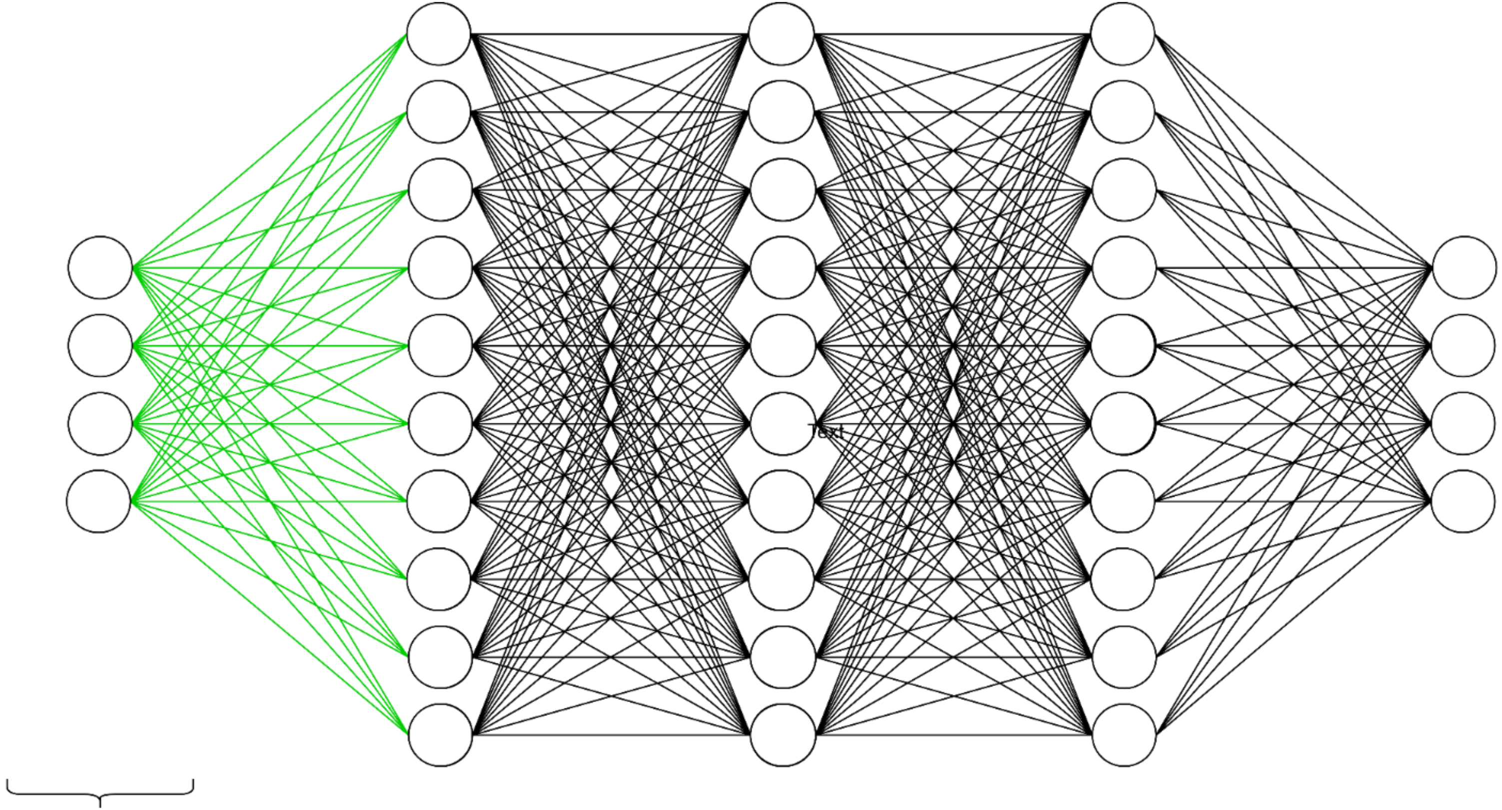
$$\hat{y} = \sigma(w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n)$$

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

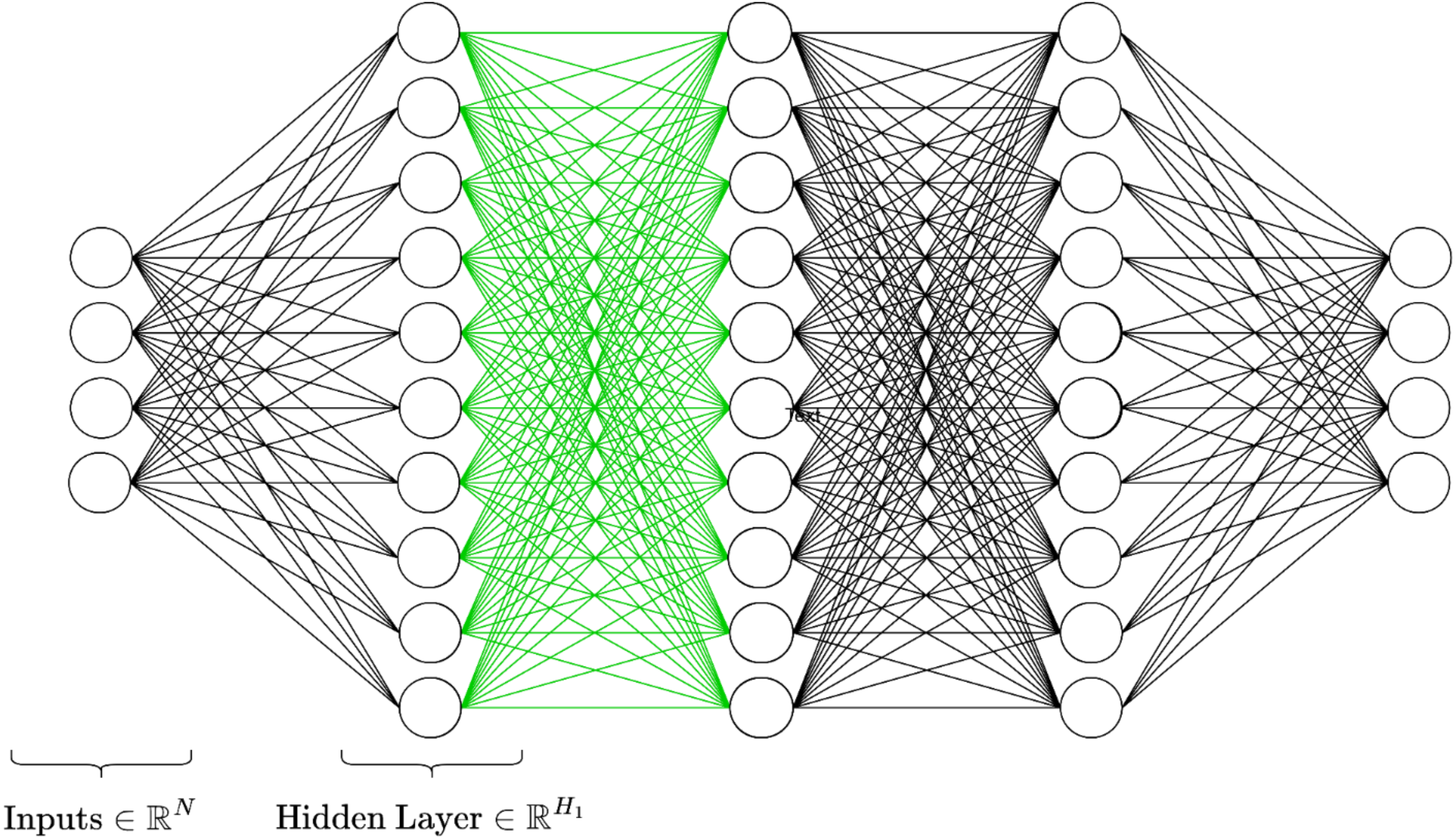
The **output** of each layer is the product of its **weight matrix** and the **input vector** plus its **bias vector**, all wrapped in a **non-linear activation function**.

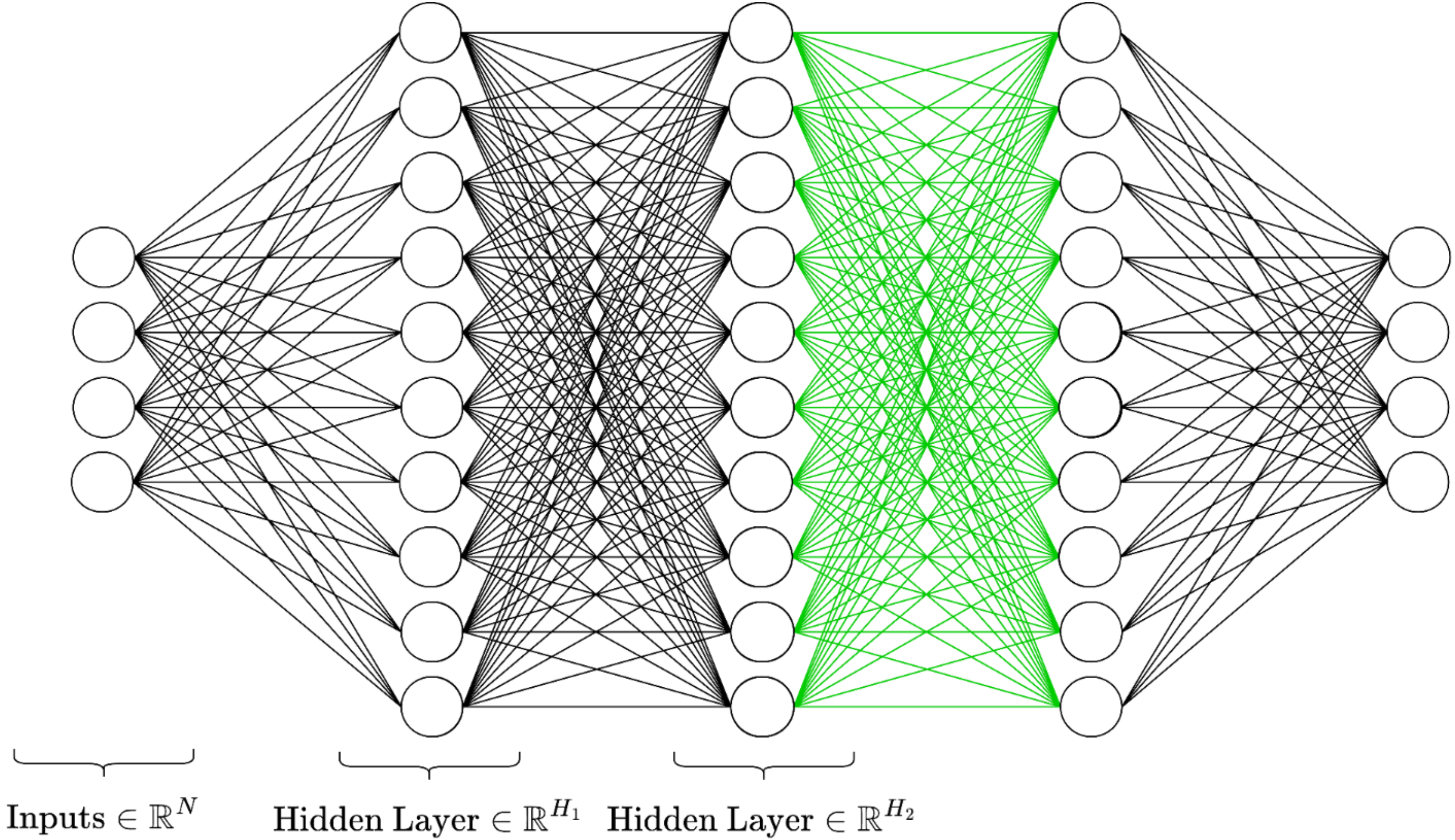

⇒
Inputs
⇒
Outputs

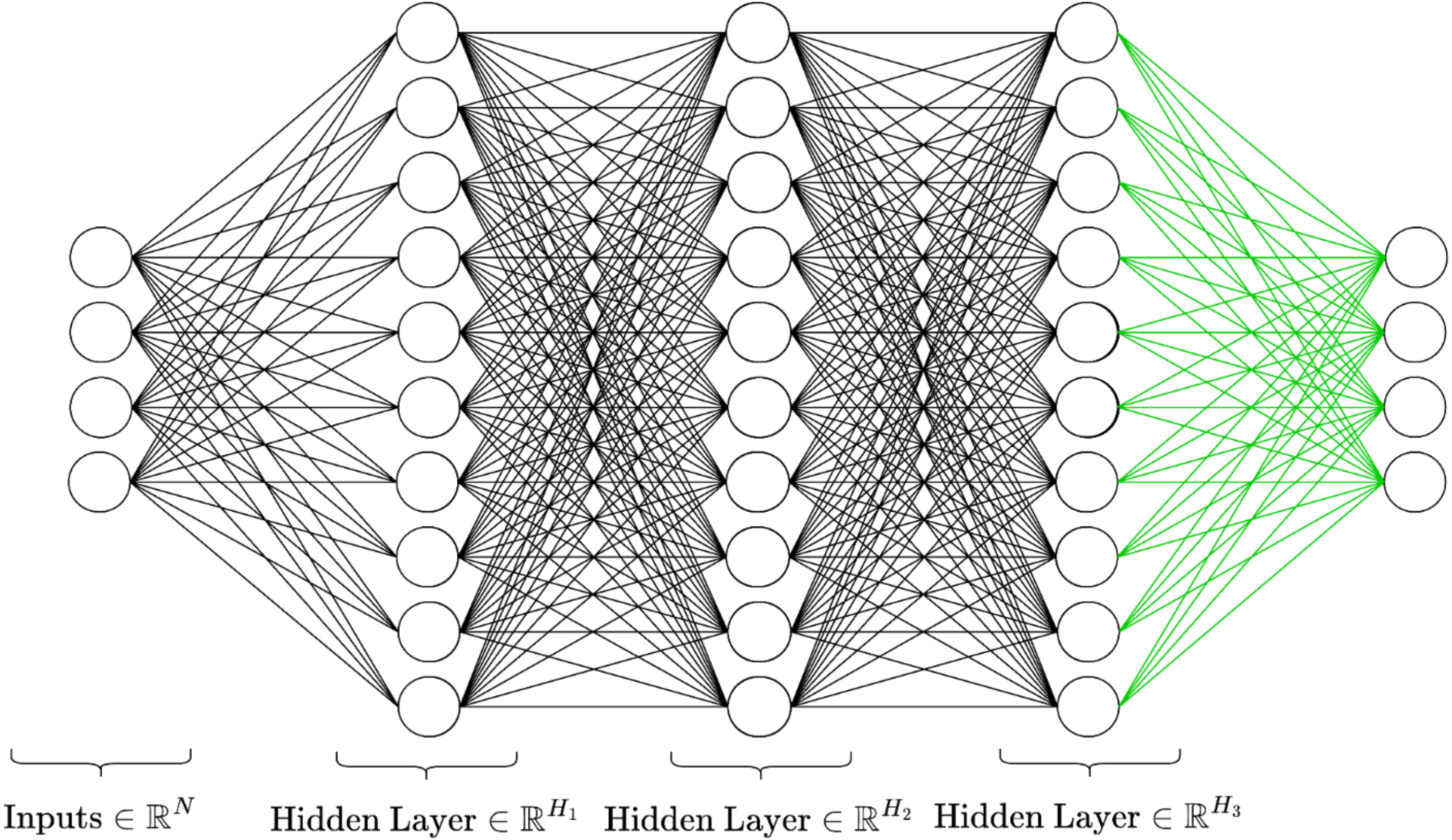

Inputs $\in \mathbb{R}^N$
Hidden Layer $\in \mathbb{R}^{H_1}$
Hidden Layer $\in \mathbb{R}^{H_2}$
Hidden Layer $\in \mathbb{R}^{H_3}$
Outputs $\in \mathbb{R}^M$

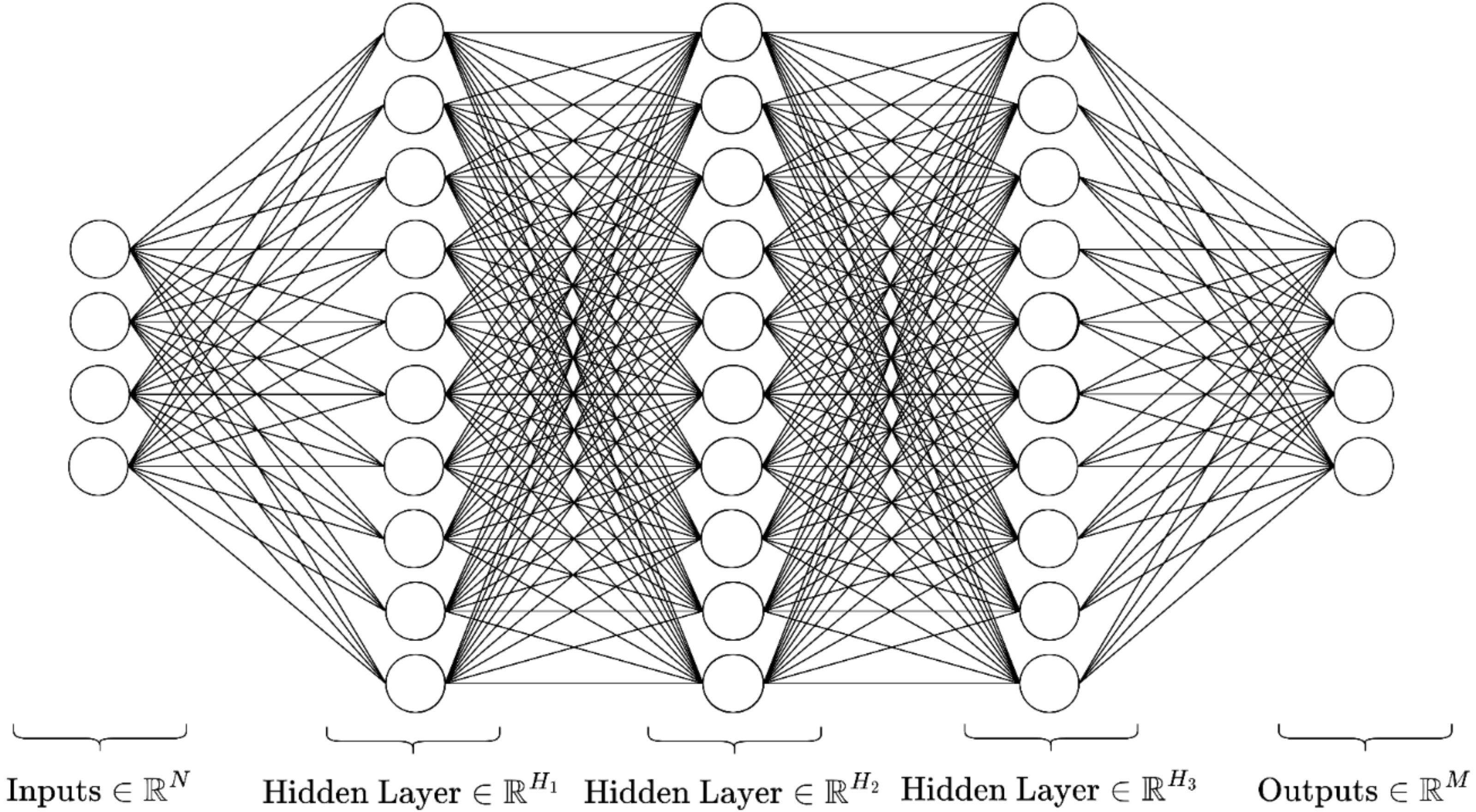


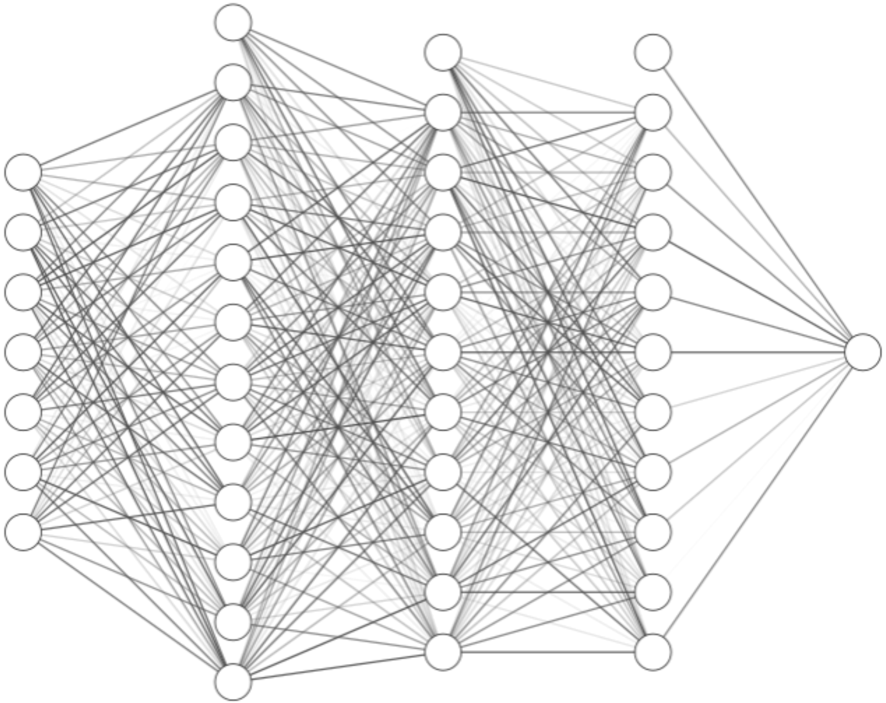
Inputs $\in \mathbb{R}^N$


Text
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.

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

$$N, M \in \mathbb{N}$$

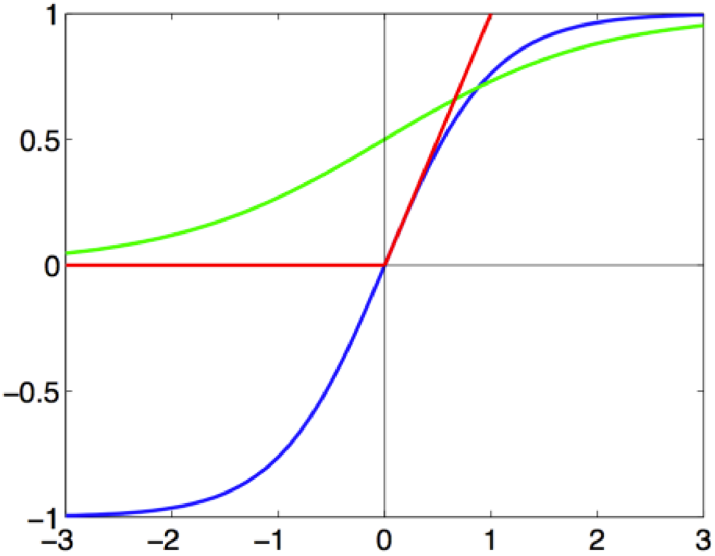
(*Translation:* <u>an MLP is a fancy function</u>)

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



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

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

Rectified Linear Unit:

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

Implementing learning: Backpropagation

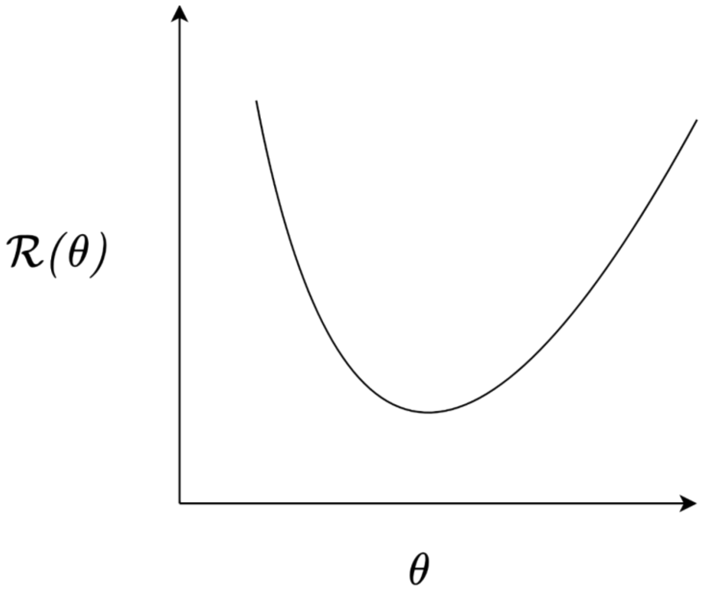
Given:

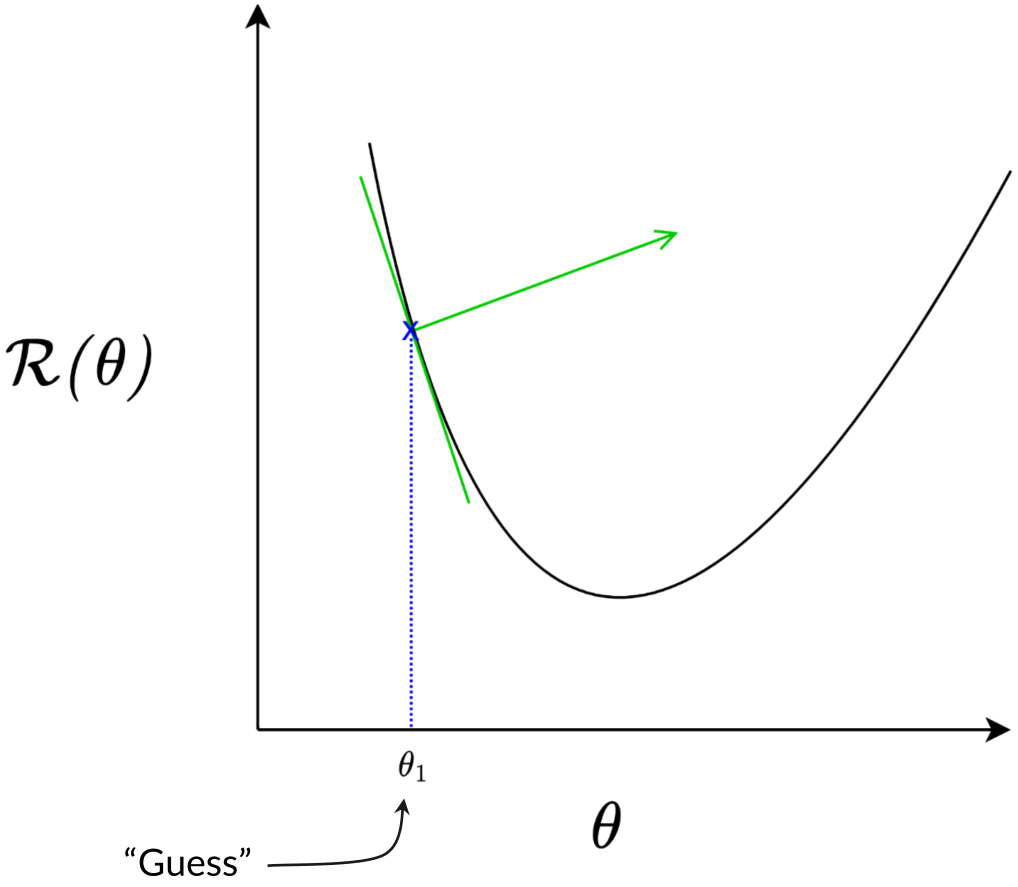
- Family of parameters Θ (e.g. possible weights of a NN)
- Differentiable risk function $\mathcal{R}(\theta)$

Goal:

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

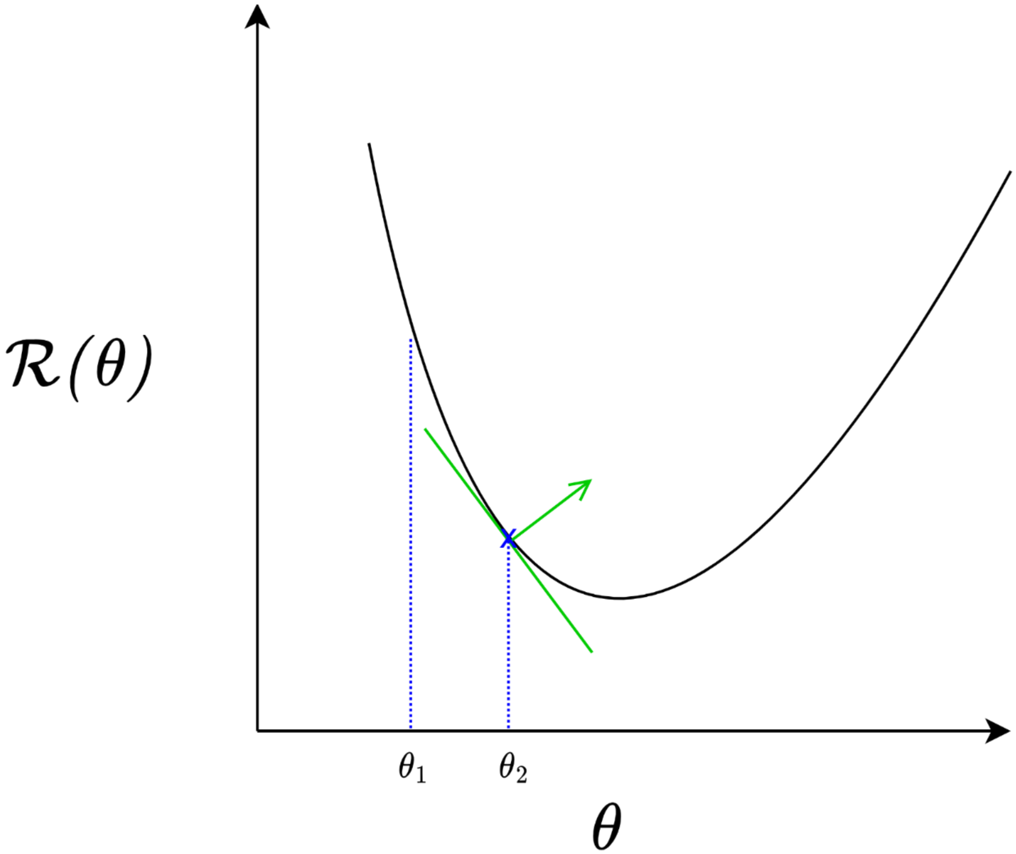
Backprop: Gradient descent



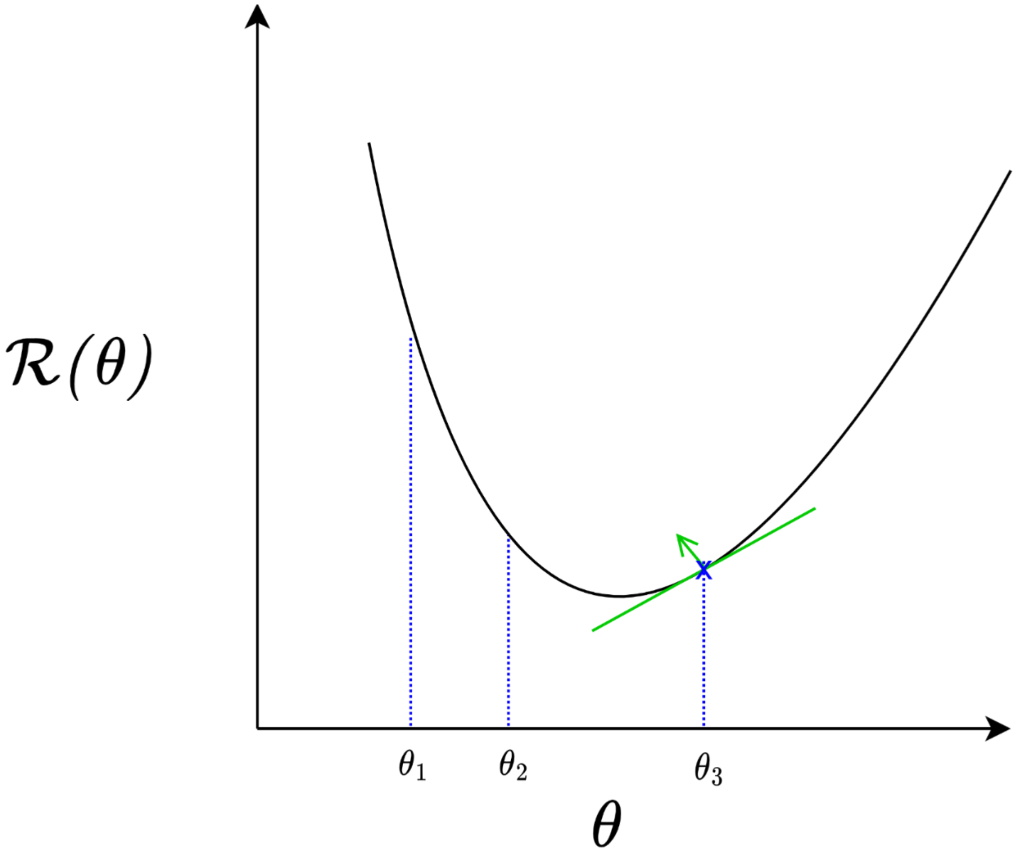


“Learning rate”

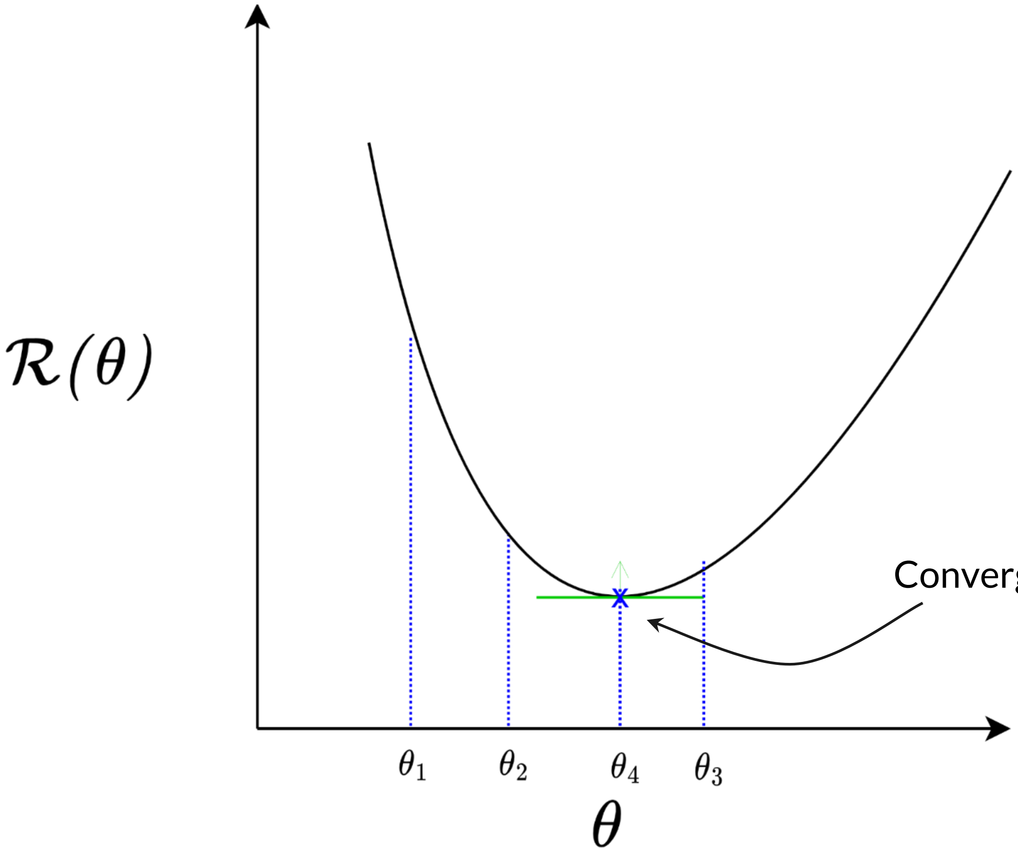
$$\theta_i = \theta_i - \alpha \frac{\partial}{\partial \theta_i} \mathcal{R}(\theta_i)$$



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



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



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

Converges to local minima

Optimizers

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

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

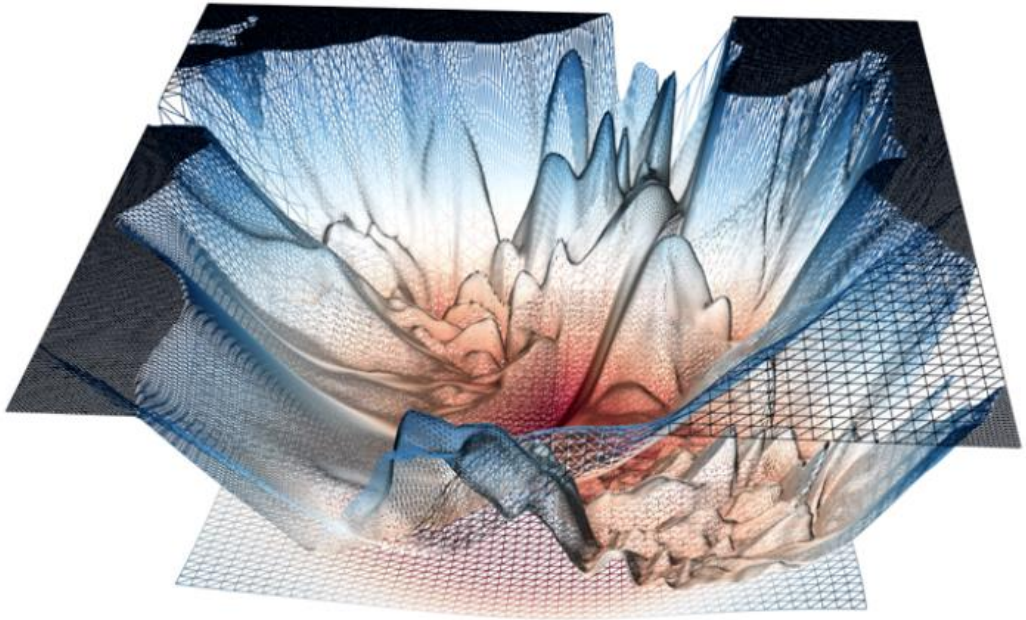
- Reduces the number of expensive backward passes

Optimizers determine exactly how backpropagation 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

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

Depends on the task!

Mean Squared Error

Used for e.g. regression tasks

Cross Entropy

Used for e.g. classification tasks

Define your own!

Note: Must be differentiable for gradient descent based methods

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.

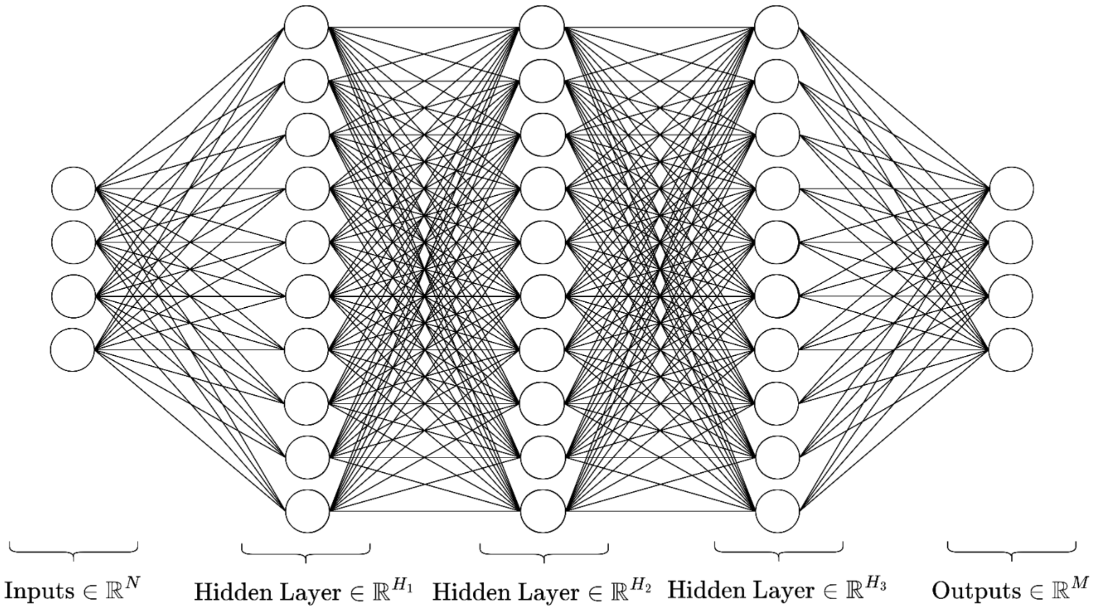
Types of Neural Networks

Multi-layer Perceptrons

Useful for **static** input-output relations

More hidden layers ~ better approximation of more complicated functions

Quick to design and implement

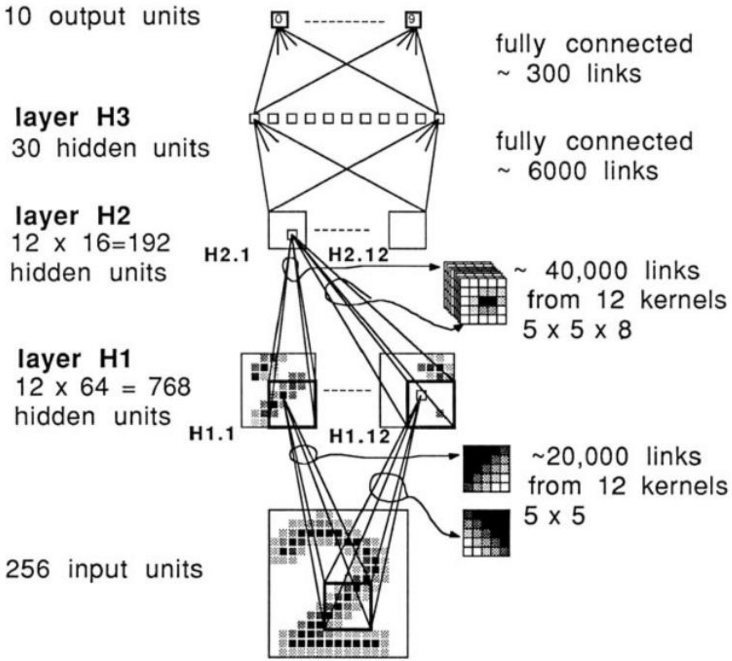


Convolutional Neural Networks

Learn “**kernels**”, i.e. 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

Recurrent Neural Networks

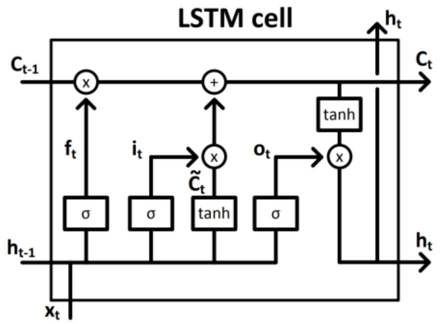
Outputs go back and forth between neurons (loops exist in the graphs)

Approximates dynamical systems

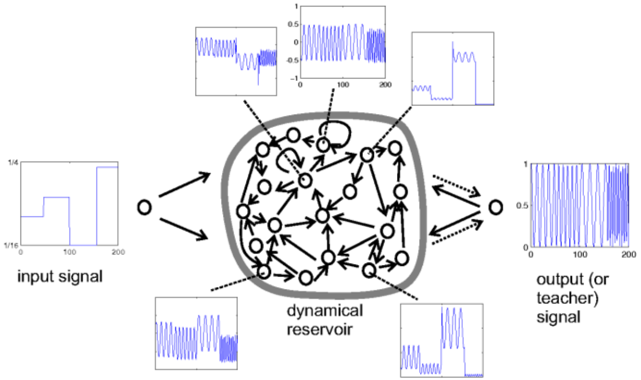
- Any time-based function
- Any data that can be modelled as being "ordered"

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*



Echo state network schematic. Adapted from *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

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

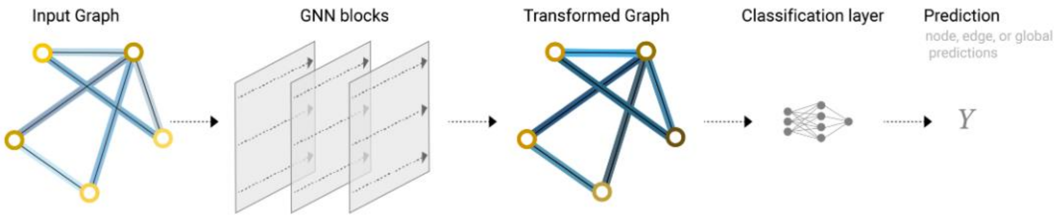


Image adapted from this excellent intro to GNNs: https://distill.pub/2021/gnn-intro/

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

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/

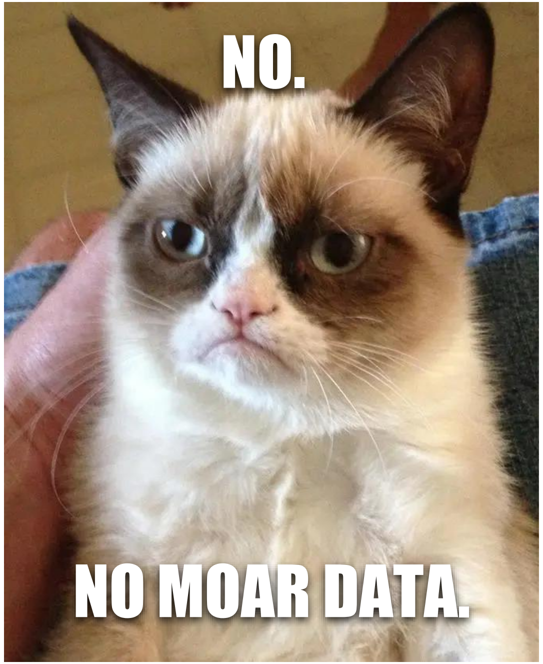
Limits of NNs

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

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

> **Often expensive to train**

> **Nonextensible** and **specialized** to a range and task
- Add one more neuron → retrain the entire network
- Undefined behaviour on out-of-domain test examples



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

In practice

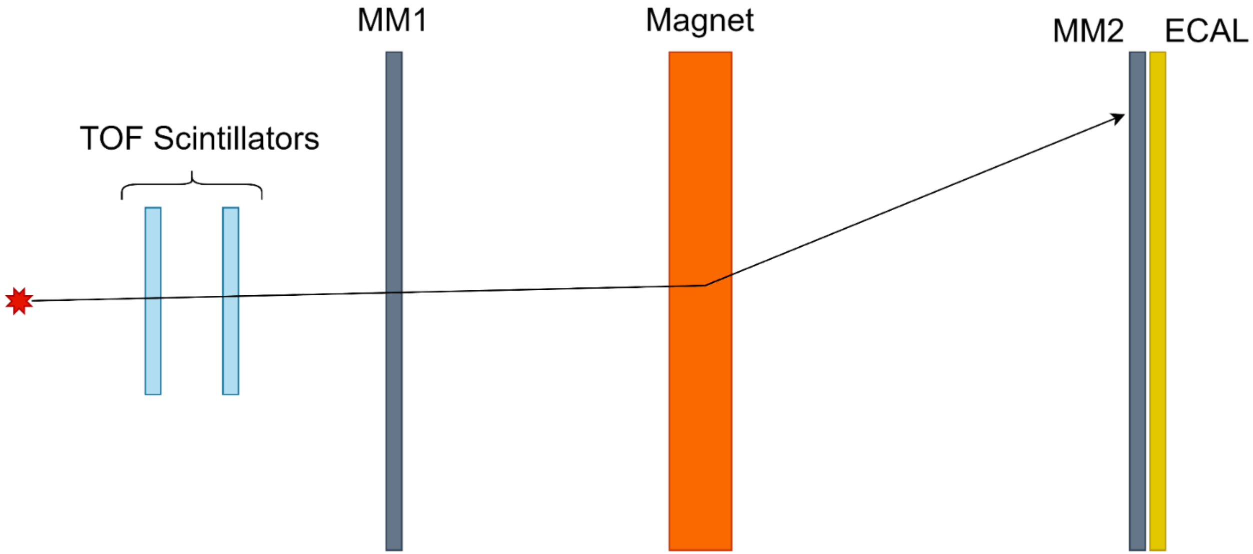
Frameworks

You don’t need any maths or programming skills (but hopefully you do!)

Use other people’s code! libraries, frameworks, modules

Image of logos adapted from S. Summers, ISOTDAQ Lecture on Machine Learning (2020) – Slide 8

A live demonstration in TensorFlow



Training tips

Overfitting & Underfitting

The real troublemakers in ML in general!

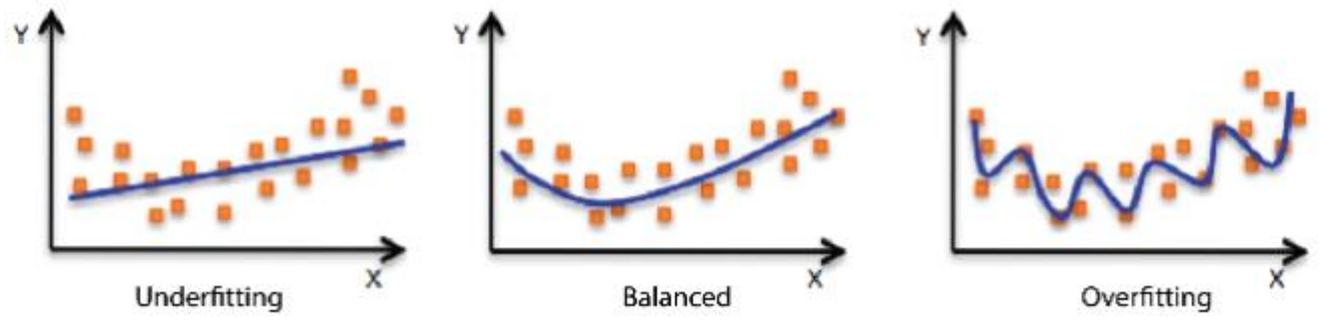
Underfitting: When the model fits the training data not well enough

- 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

Image source: https://docs.aws.amazon.com/machine-learning/latest/dg/model-fit-underfitting-vs-overfitting.html

Validation

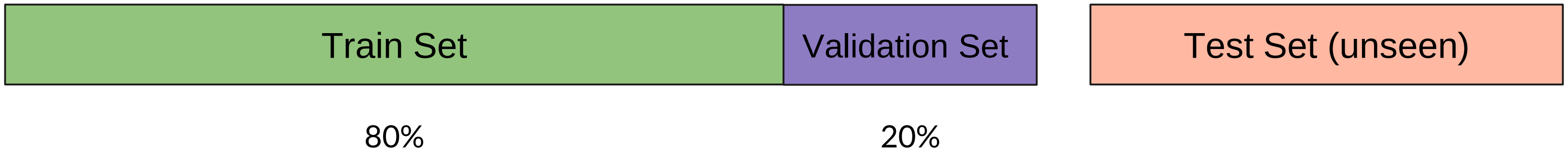
Split your training set into two!

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

Validation

Split your training set into two!

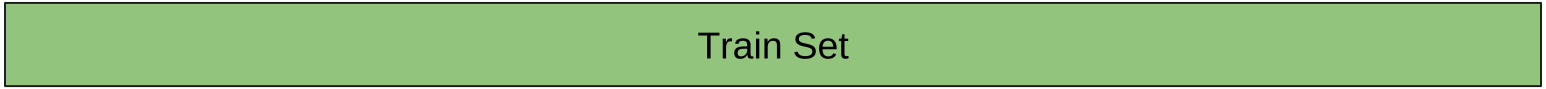
- New train set
- Unseen-by-the-model “validation”set
- e.g. 80-20 split (Note: split ratio depends on the model, task and data)



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



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)

Train Set	Train Set	Train Set	Train Set	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)

Validation Set	Train Set	Train Set	Train Set	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)

Train Set	Validation Set	Train Set	Train Set	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)

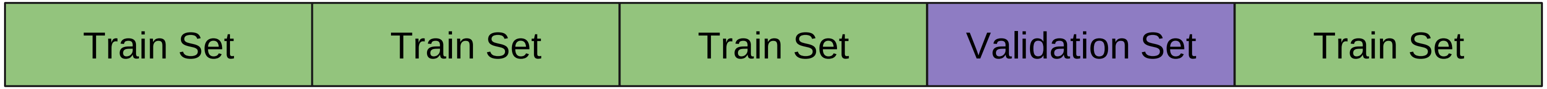
Train Set	Train Set	Validation Set	Train Set	Train Set

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

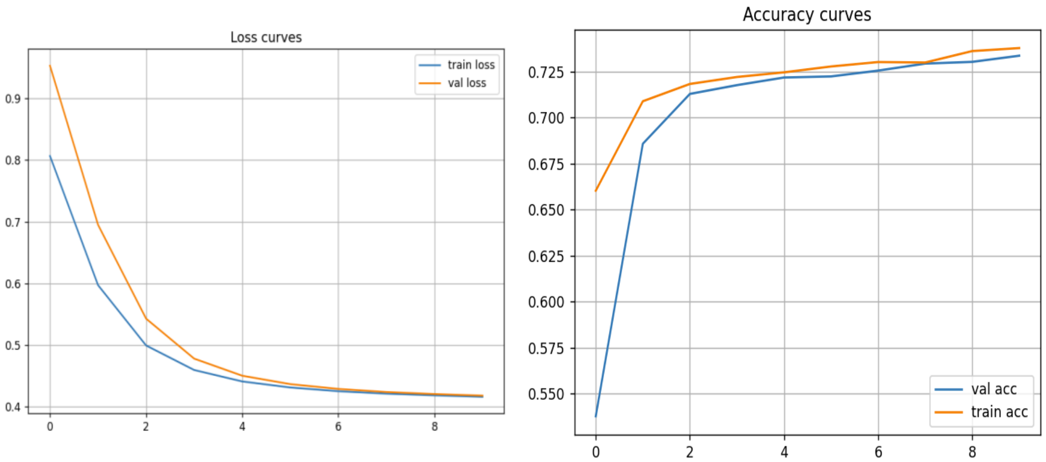
e.g. k=5 (5-fold cross-validation)

Train Set	Train Set	Train Set	Train Set	Validation Set

Result = average over all validation passes

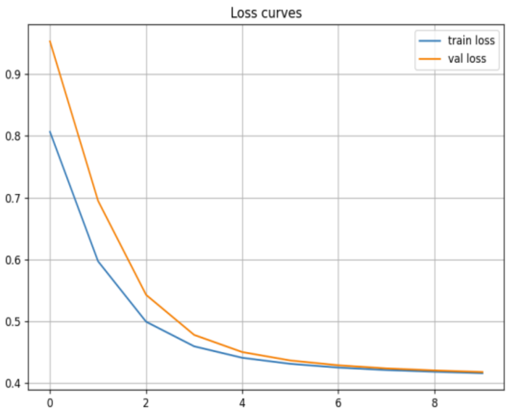
Training curves

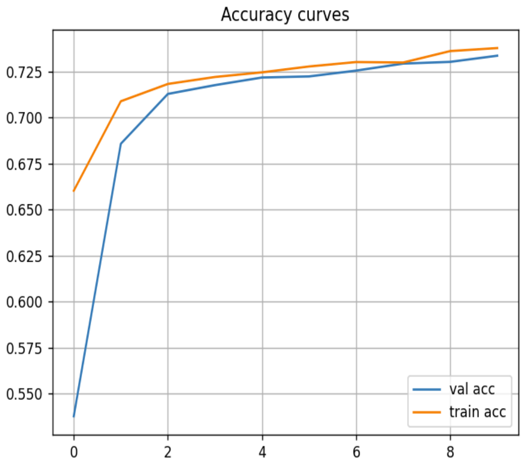
Important to plot!



Training curves

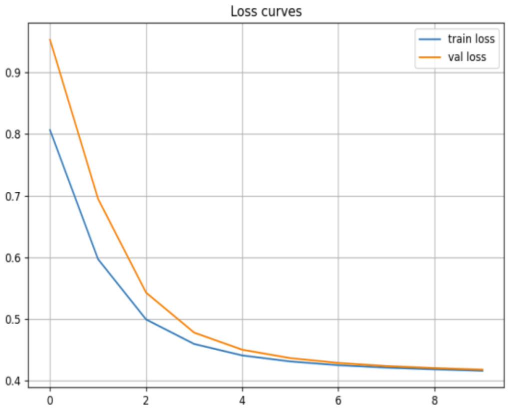
Important to plot!(!!!!)

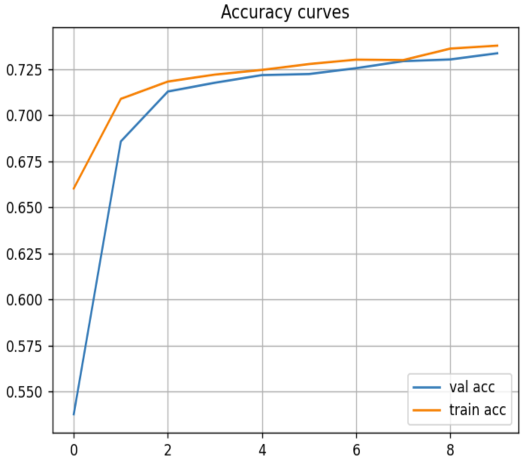




Training curves

Important to plot!(!!!!)



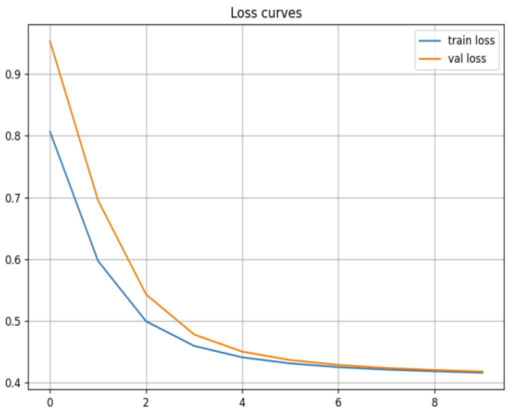


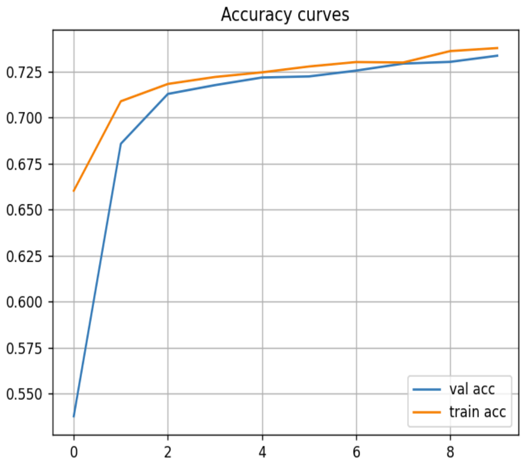
Shows if and how fast your model is learning on task-relevant metrics

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

Training curves

Important to plot!(!!!!)



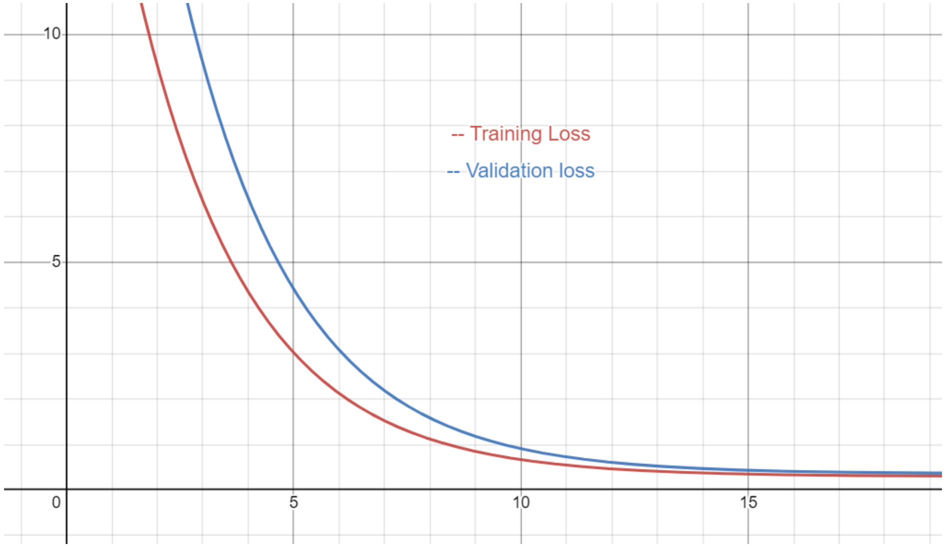


Shows if and how fast your model is learning on task-relevant metrics

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

May indicate potential over and underfitting

Reading training curves



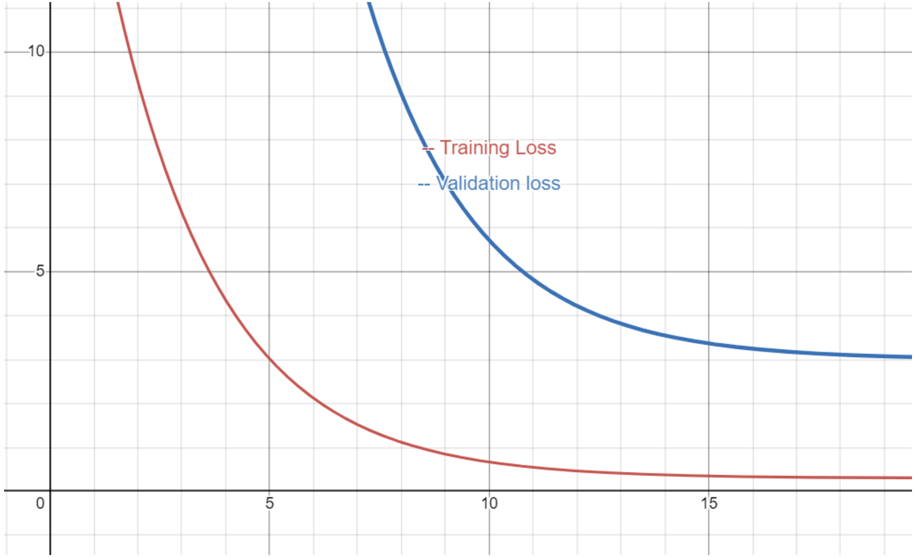
If

validation loss > training loss

then often the model is *good!*

Low loss == Better

Reading training curves



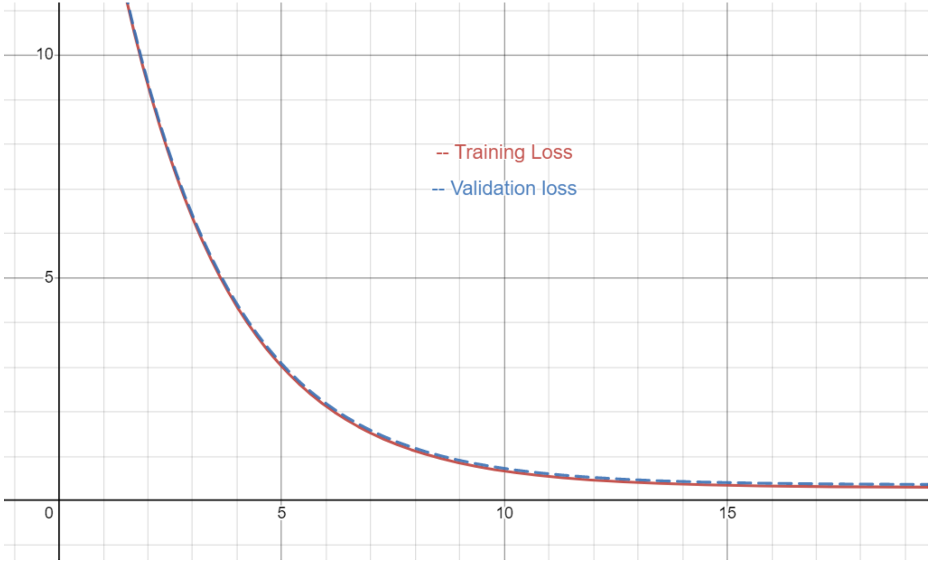
If

validation loss >> training loss

then often the model is *overfitting*

Low loss == Better

Reading training curves



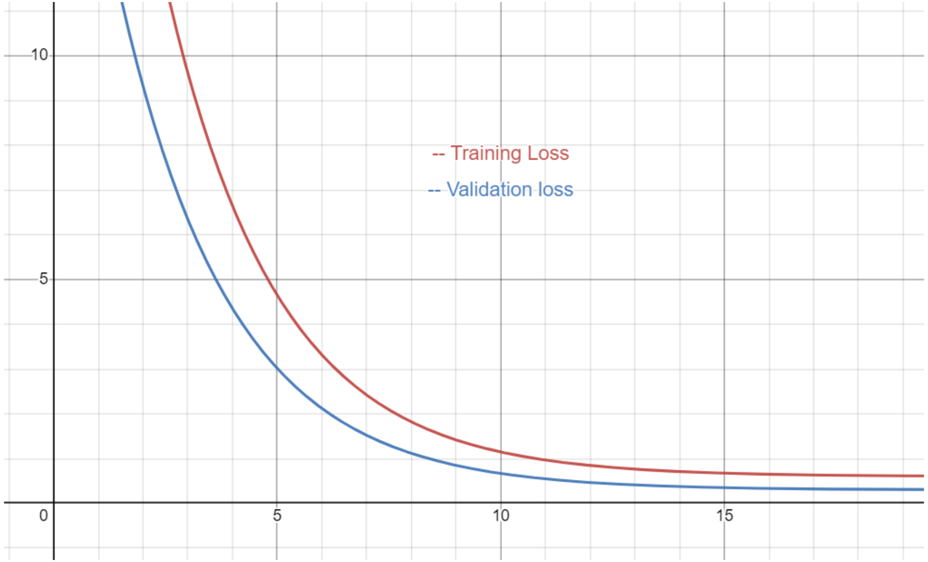
If

validation loss ~ training loss

then often the model is *underfitting*

Low loss == Better

Reading training curves



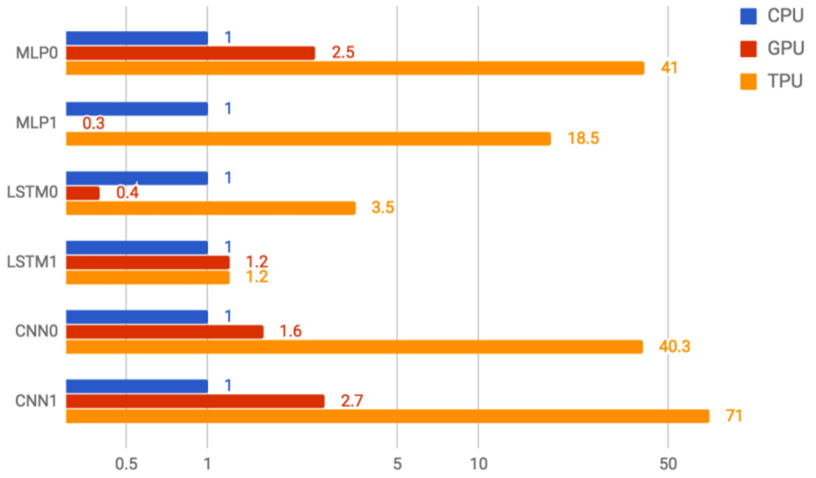
If

validation loss < training loss

then something is *very* wrong, or *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