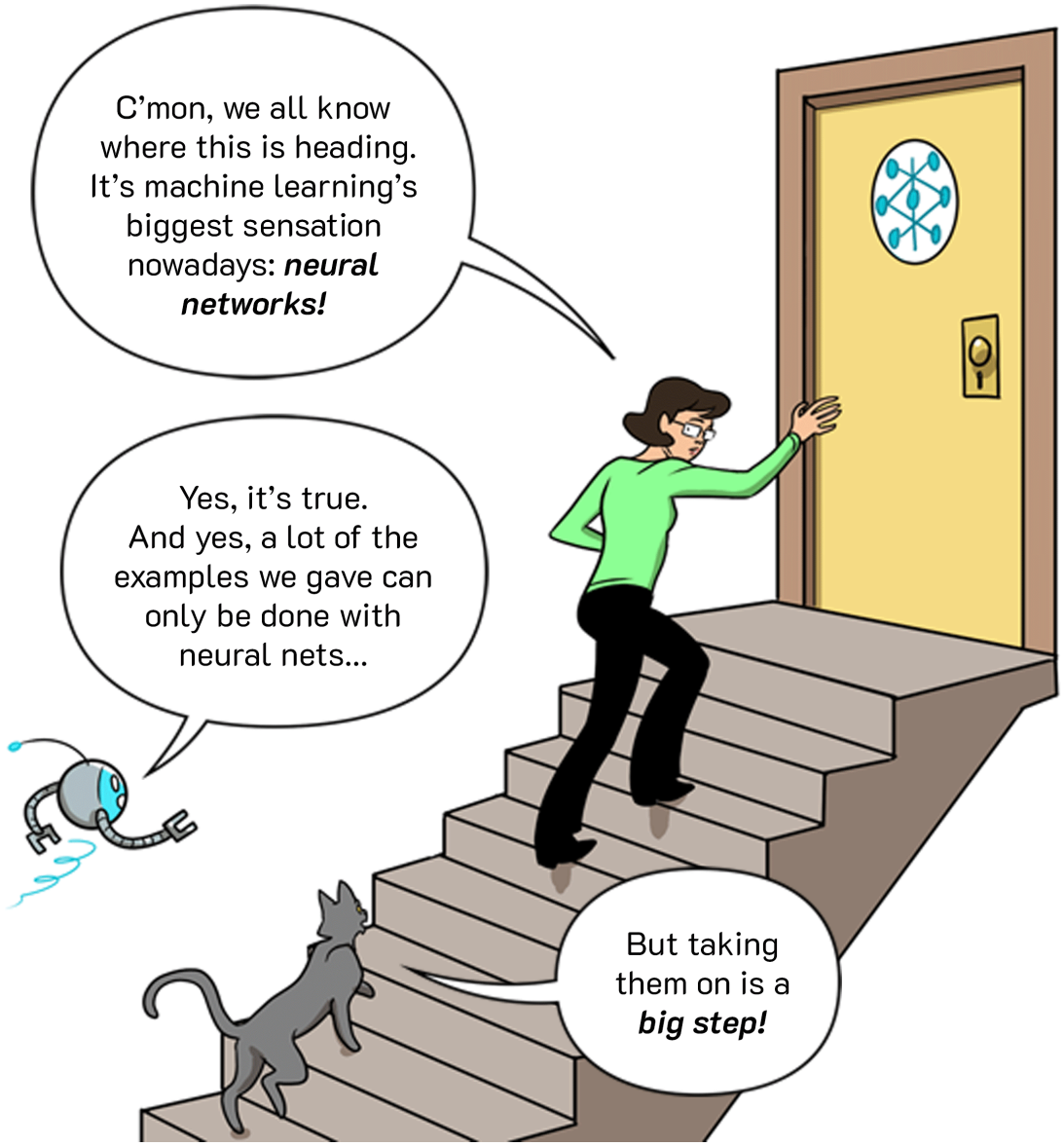
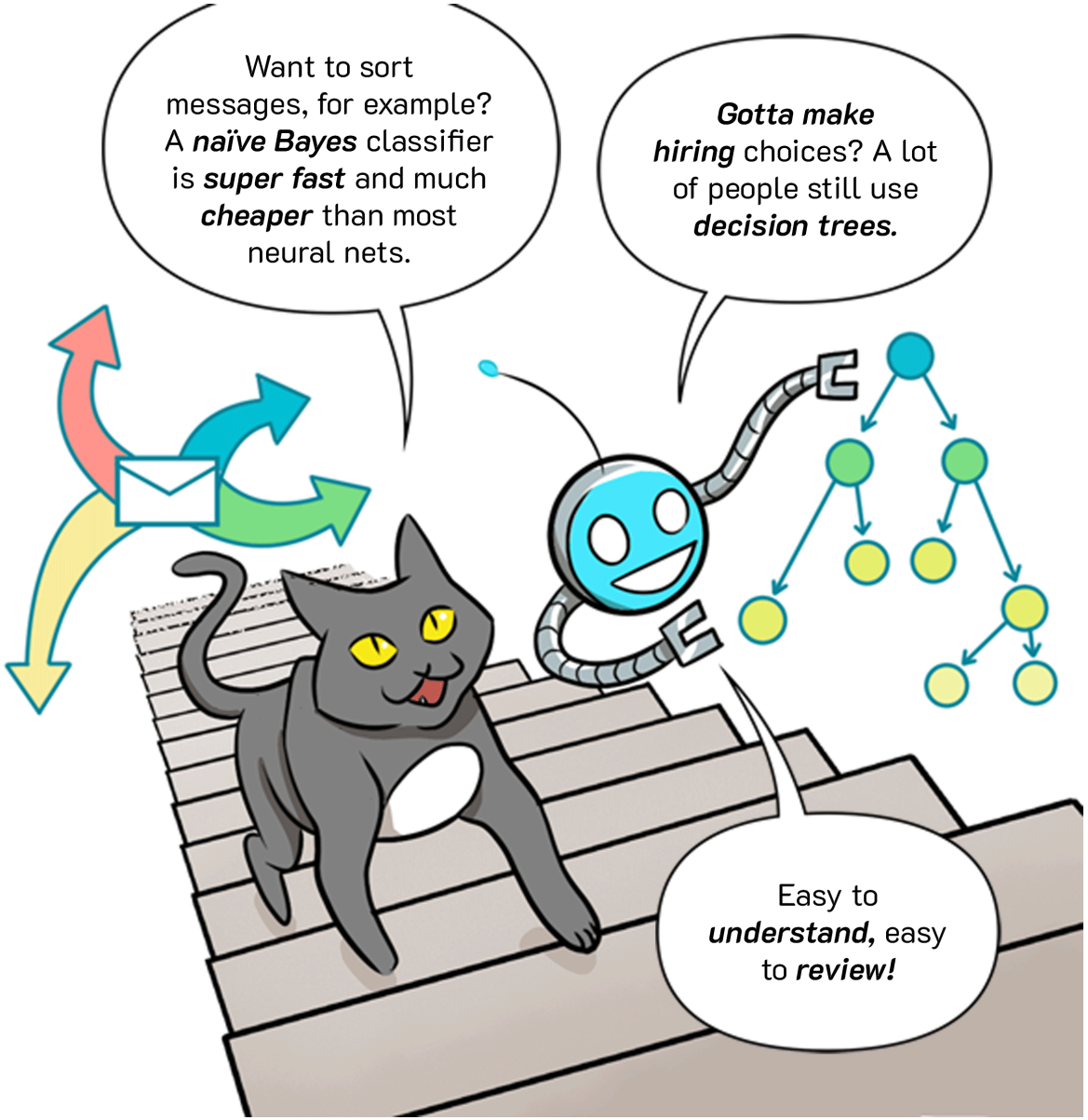

C'mon, we all know where this is heading. It's machine learning's biggest sensation nowadays: ***neural networks!***
Yes, it's true. And yes, a lot of the examples we gave can only be done with neural nets...
But taking them on is a ***big step!***



Comic source: https://cloud.google.com/products/ai/ml-comic-1

Comic source: https://cloud.google.com/products/ai/ml-comic-1

UNIVERSITY OF LIVERPOOL

Department of Physics

INTRODUCTION TO NEURAL NETWORKS I

Dr. Andrea Santamaria Garcia
Lecturer

CI lectures CI-ACC-226
Lecture on 23/02/2026

OUTLINE

- **General considerations about neural networks** (7 slides)
  - *What are NNs conceptually and mathematically, why use them, what can they actually do, and limitations*
- **Supervised and unsupervised learning** (5 slides)
- **Review of regression** (18 slides)
  - *Univariate linear regression, loss function and least squares fitting, the loss landscape, gradient descent*
- **Neural networks** (15 slides)
  - *The perceptron, forward propagation, backpropagation*

GENERAL CONSIDERATIONS FOR NEURAL NETWORKS

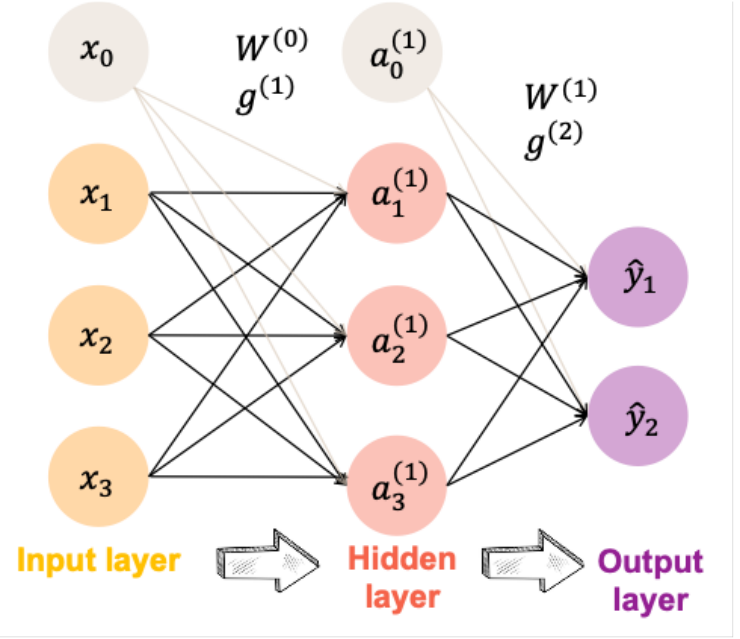
What are neural networks, conceptually?

They are combinations of mathematical functions that implement more complicated functions

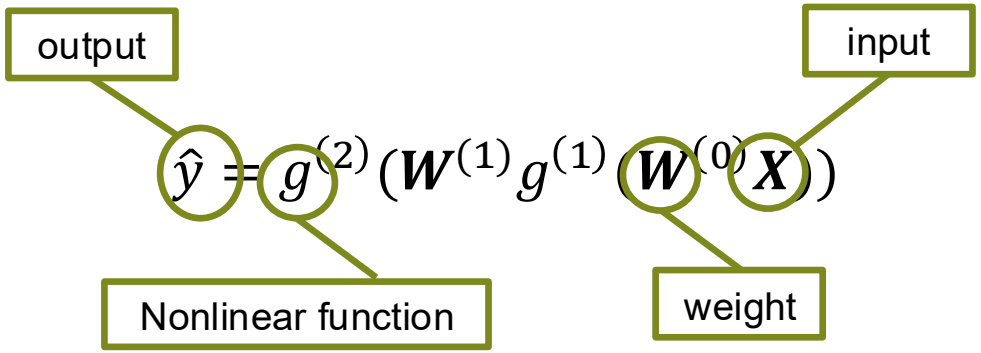
We have a dataset composed of data pairs (x, y).
We want to have an accurate prediction $\hat{y}$ of our target y given a certain input x

target input data
$y = f(x)$ the model of the data, mapping between x and y

$f(x)$= neural network



NN = **parameterised function approximator** built as a composition of affine transformations and nonlinear activations, whose parameters are learned from data by optimising a loss function



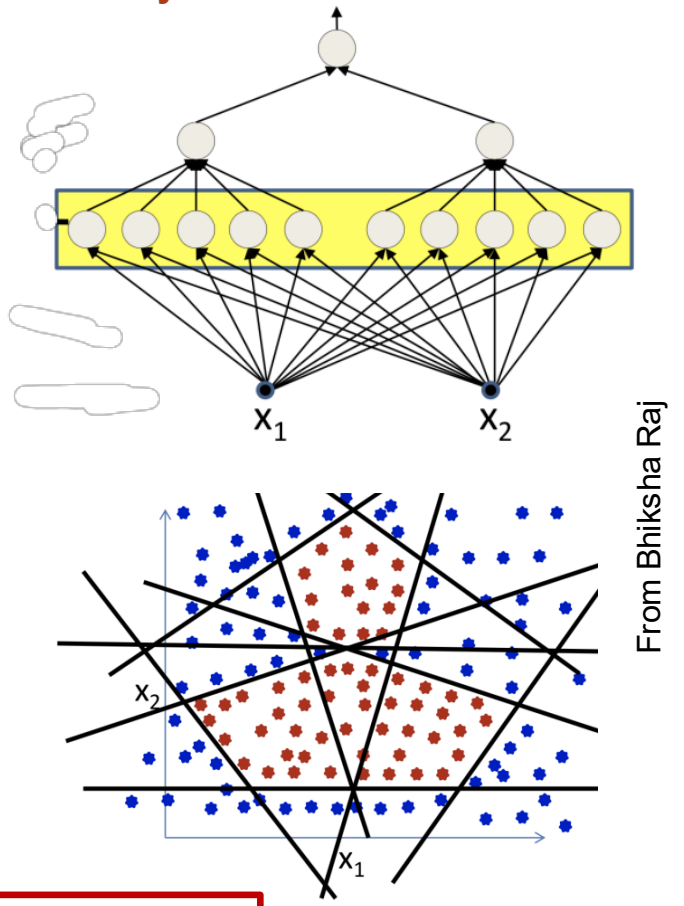
What are neural networks, mathematically?

Each layer computes: $\boldsymbol{g(Wx + b)}$

where $x \longmapsto Wx + b$ is an **affine transformation**

- **Linear maps mix coordinates and create new feature directions**
  Without $\boldsymbol{W}$, each dimension would be processed independently
- **The bias term allows translation of hyperplanes**
  Without $\boldsymbol{b}$, all decision boundaries would pass through the origin
- **The activation function breaks global linearity and enables nonlinear structure**
  Without the nonlinearity of $\boldsymbol{g}$, stacking layers would collapse to a single affine transformation
- **Stacking layers builds increasingly complex decision regions**
  Each layer increases the number of piecewise-linear regions. Without depth, complex functions would require dramatically more neurons

Decision boundary and corresponding two-layer classification network



From Bhiksha Raj

Expressivity arises from alternating **linear** mixing and **nonlinear transformations**

Why neural networks?

Universal Approximation Theorem

A neural network with just one hidden layer **can approximate any continuous function as closely as desired on a bounded region of the input space**, provided it has enough neurons

For the math people:

A single-hidden-layer network with a non-polynomial activation **can approximate any continuous function** on a compact set to arbitrary accuracy

Is this too good to be true 🧐 ?

of course, there are some caveats

Caveat 1: the network might need to be very large

→ The required number of neurons can grow rapidly with input dimension

Caveat 2: approximation is not the same as learning

→ The theorem assumes we can find the right parameters, but it does not guarantee that training will discover them

Caveat 3: fitting is not generalising

→ Fitting well on observed data does not ensure good predictions on new data

Thanks to machine-learning algorithms, the robot apocalypse was short-lived.

https://www.smbc-comics.com/comic/rise-of-the-machines

So, what can neural networks actually do?

Expressivity is not the bottleneck, data and scale are

The universal approximation theorem tells us neural networks *can* approximate almost anything.

The real question is: **under what conditions can they do so in practice?**

Network size determines representational capacity

Number of parameters for fully connected dense layers:

$$\sum_{l=0}^{L} (n_l n_{l+1} + n_{l+1})$$

n_l = hidden layer size (width)
L = number of layers (depth)

weights in layer l — biases in layer l

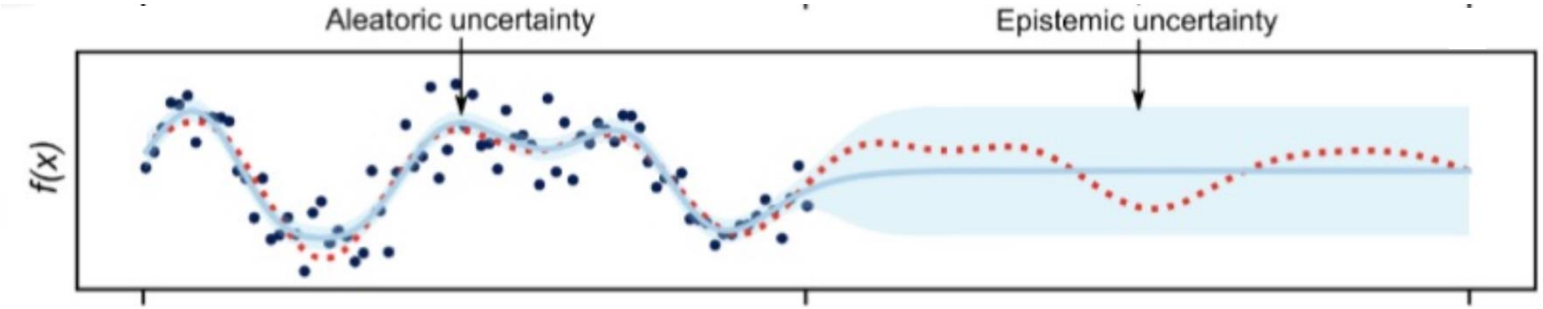
Learning depends critically on data

- Number of samples
- Coverage of input space
- Diversity of conditions
- Signal-to-noise ratio

Neural networks are powerful not because they are universal, but because large networks trained on **large, diverse datasets** can **approximate highly structured functions efficiently**

Limitations of neural networks: data

Types of uncertainty



Aleatoric uncertainty

Data uncertainty

- Represents irreducible noise in the observations
- High when measurements are noisy or labels are ambiguous
- Cannot be reduced by adding more data
- Can be modelled using probabilistic outputs

Epistemic uncertainty

Model uncertainty

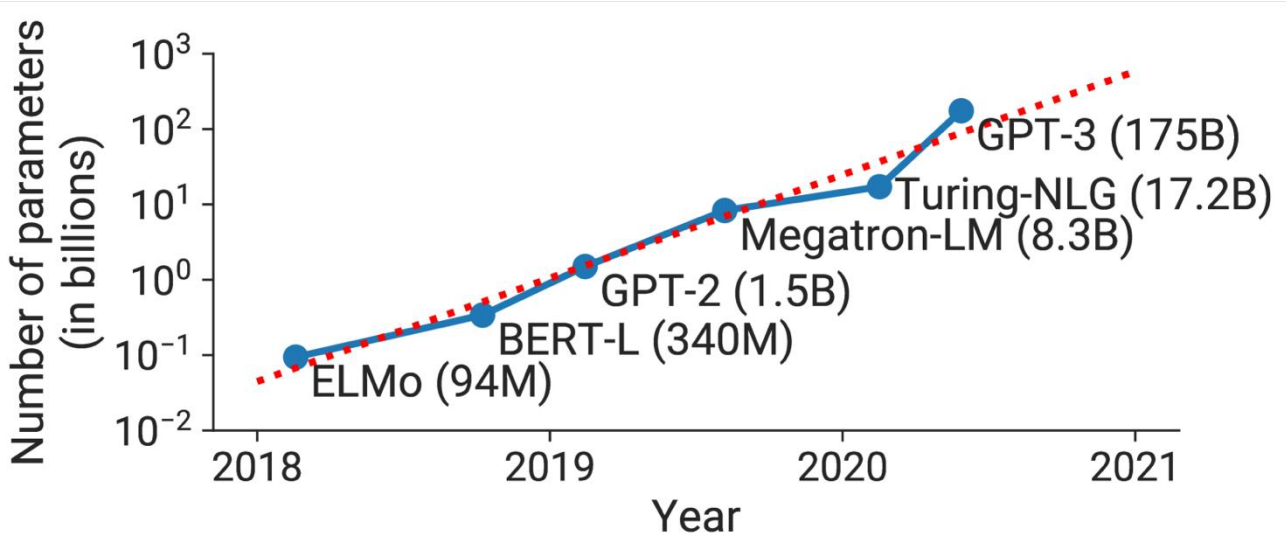
- Represents uncertainty in the model due to limited knowledge
- High in regions with little or no training data
- Can be reduced by adding more data
- Often estimated using Bayesian models or ensembles

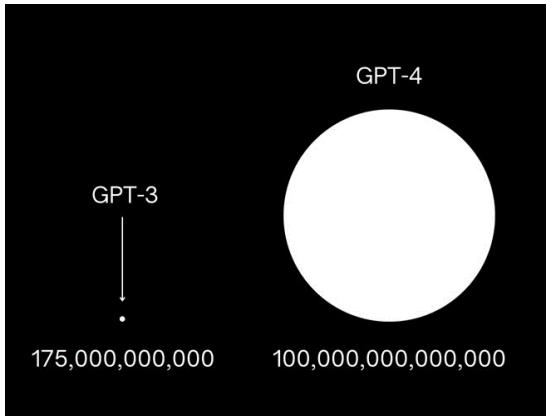
from “Introduction to Deep Learning” 6.S19 (MIT)

A data hungry, very large parameter example: LLMs

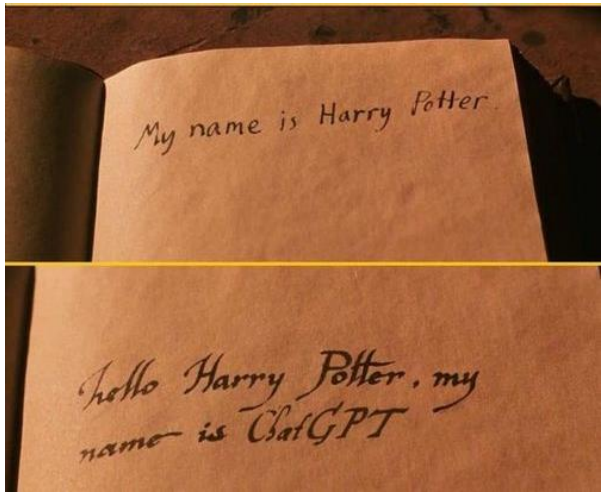
In deep networks, performance keeps increasing with data quantity and network size

https://ar5iv.labs.arxiv.org/html/2104.04473





Not sorcery, just a lot of resources!



Limitations of neural networks: narrow AI

The "No Free Lunch" theorem

Every model is a simplification of reality (encodes **assumptions** about the structure of the data)

These assumptions impose an **inductive bias**, which determines what functions are easier to learn

No **single algorithm** is best for **all possible problems**

No Free Lunch formalises why all practical AI systems are **narrow**:

→ **strong performance** requires **strong assumptions** about the structure of the world

Performance depends on how well the **algorithm's bias** matches the **structure of the data**

https://arxiv.org/pdf/2202.04513.pdf
https://arxiv.org/pdf/2007.10928.pdf

SUPERVISED AND UNSUPERVISED LEARNING

Supervised learning vs unsupervised learning

Supervised learning

Labeled data: we know x and y

Learns function $f: x \rightarrow y$

Optimises **predictive performance** on **unseen data**

- **Classification:** assign inputs to discrete categories (e.g., disease vs healthy)
- **Regression:** predict continuous numerical values (e.g., energy, temperature, price)
- **Forecasting:** predict future values of a time-dependent process based on past observations
- **Structured prediction:** predict outputs with internal structure (e.g. sequences, trees, images), rather than single scalar values

Unsupervised learning

Unlabeled data: we know x, we don't know y

Models the structure or distribution of the data x

Discovers **latent structure**, **clusters**, or **low-dimensional representations**

- **Clustering**: dataset divided into groups (similarity)
- **Dimensionality reduction**: selection of dimensions that best explain the variabilty in the data
- **Density estimation**: estimate the distribution of the data (anomaly detection, data generation)

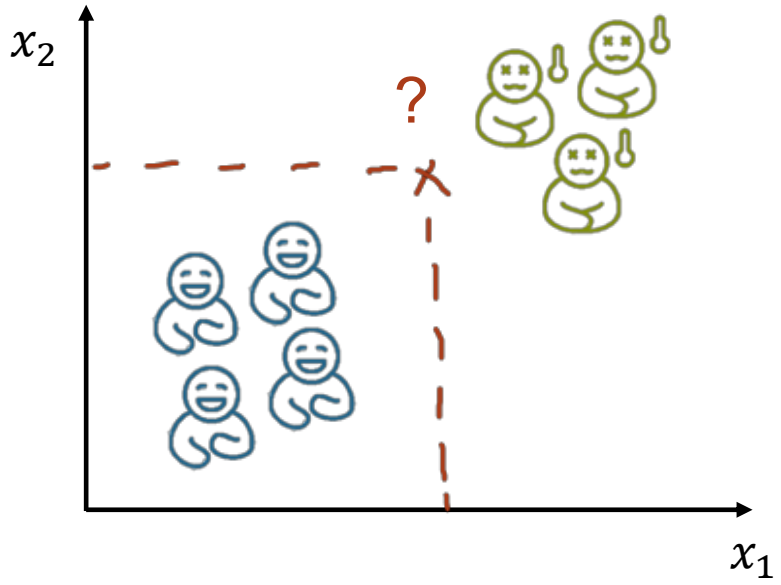
Supervised learning vs unsupervised learning

Supervised learning

Labeled data: we know x and y

Learns function $f: x \rightarrow y$

Optimises **predictive performance** on **unseen data**

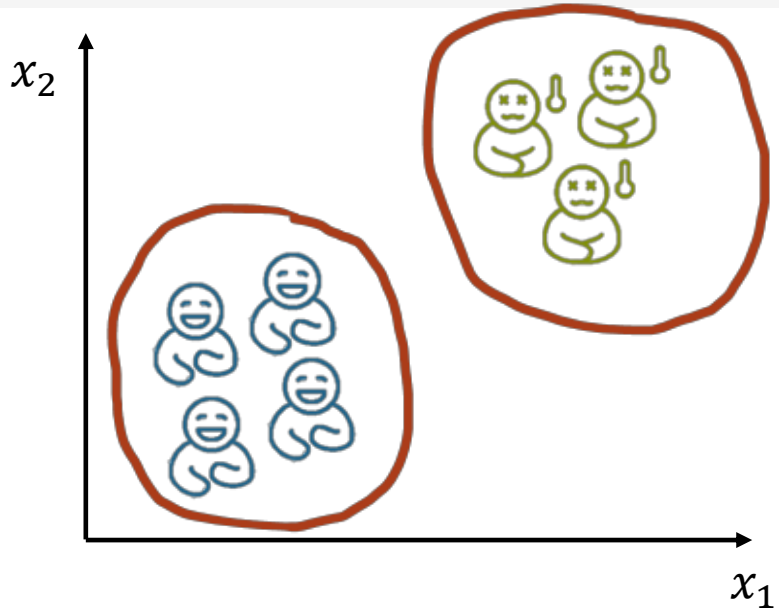


Unsupervised learning

Unlabeled data: we know x, we don't know y

Models the structure or distribution of the data x

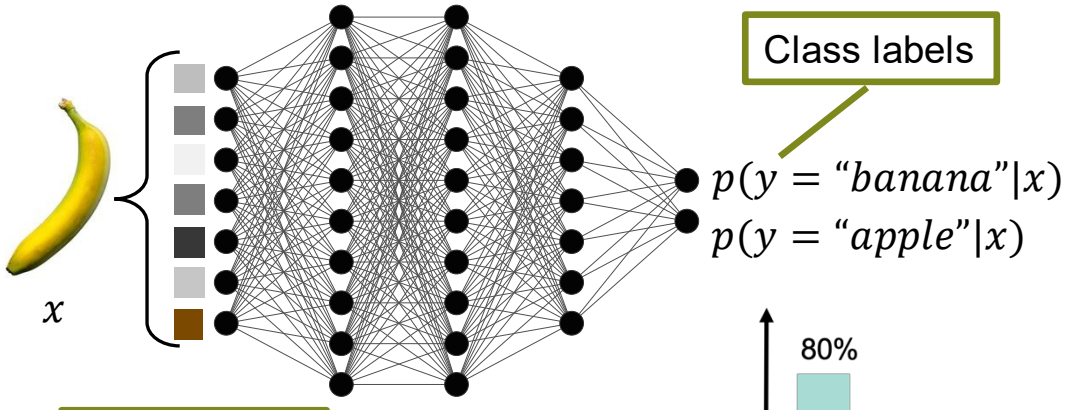
Discovers **latent structure**, **clusters**, or **low-dimensional representations**



Classification vs regression in supervised learning

Classification task

y is discrete (can take certain values only)



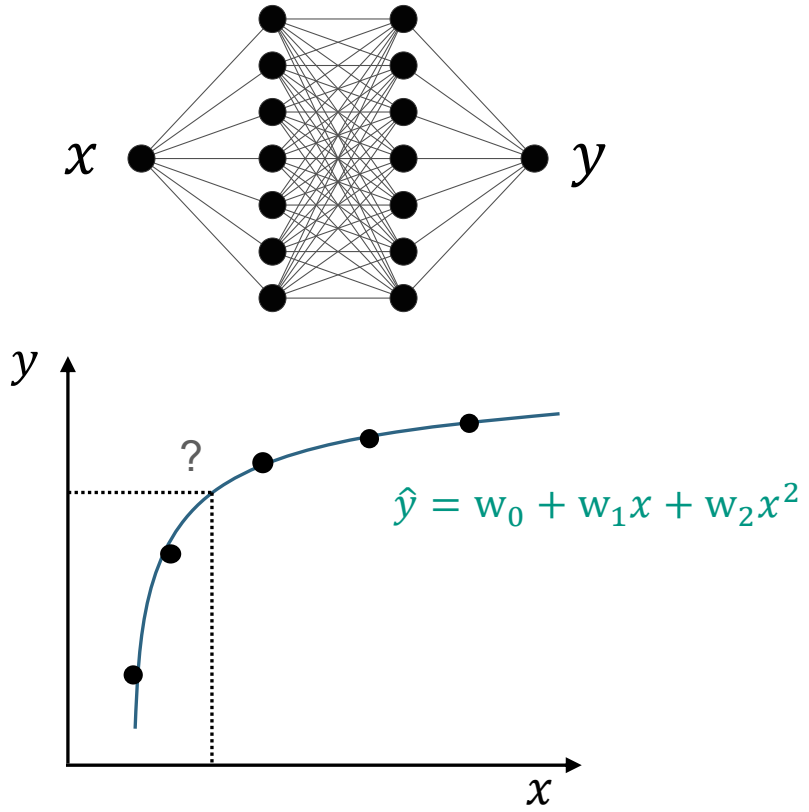
Likelihood function

$$L(w|x,y) = p_x(y)$$

Probability of a label y given the inputs x

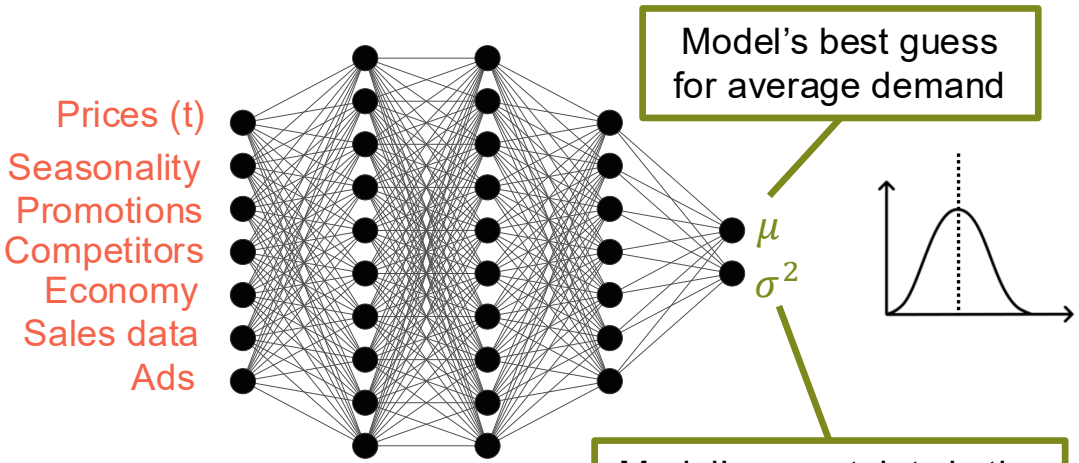
Regression task

y is continuous (can take any value)



Examples of continuous targets

Product demand forecasting

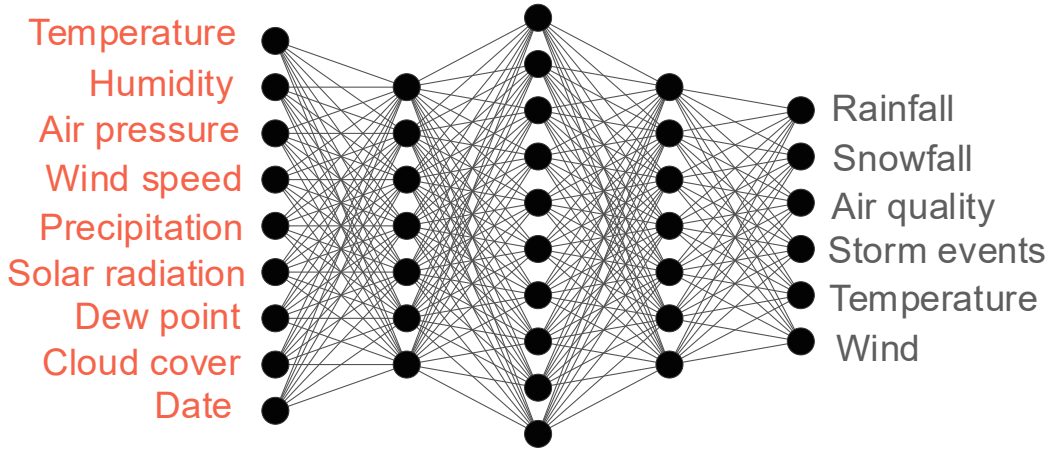


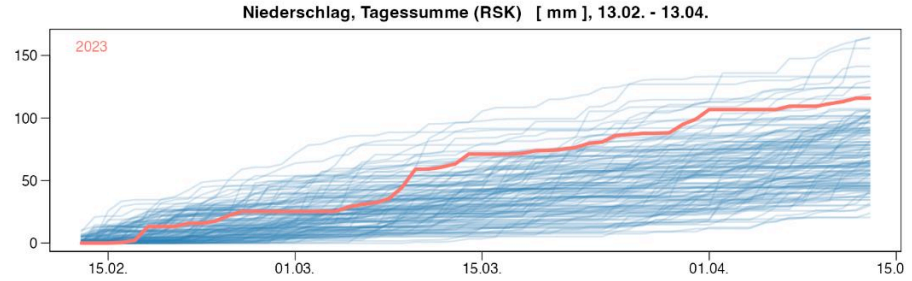
Likelihood function → gaussian

- Inventory management
- Risk assessment

$$L(w|x,y) = \text{normal}(\mu, \sigma^2)$$

Weather prediction





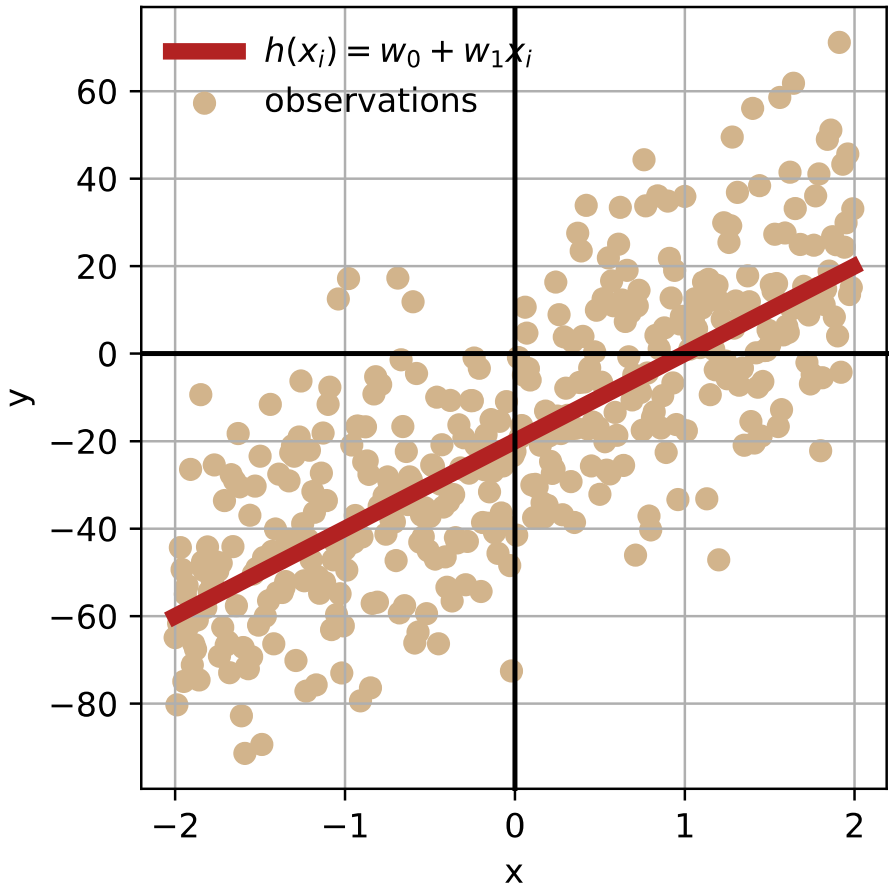
https://github.com/brry/rdwd

LET’S REVIEW REGRESSION

Univariate linear regression (one prediction)

Simple (one feature)

We want to fit linearly a set of points $(x_i,\ y_i)$



Hypothesis function

$$h: x \rightarrow \hat{y}$$

Space of input variables = **feature**

independent variable

Space of output variables = **estimated value** = **prediction**

dependent variable

$$h_w(x_i) = w_0 + w_1 x_i$$

- regression coefficients
- parameters of the model
- estimated from data

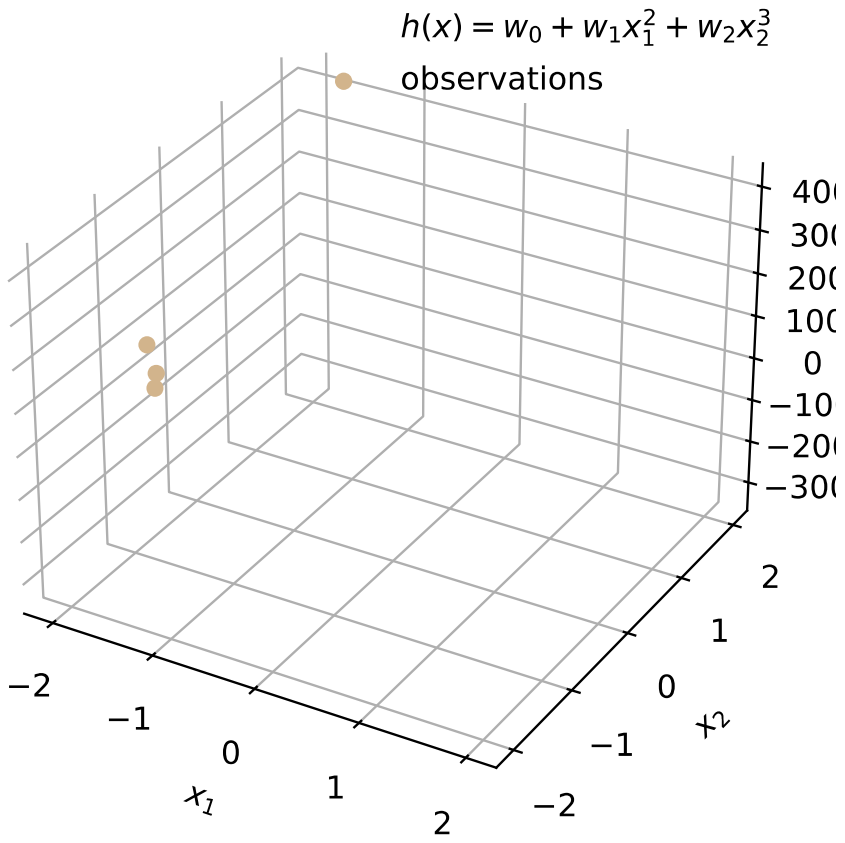
weights

$i = 1, \ldots, n$ data points

Univariate linear regression (one prediction)

Multiple (several features)

We want to fit a set of points $(x_i,\ y_i)$



Hypothesis function

$$h_w(x_1, x_2) = w_0 + w_1 x_1^2 + w_2 x_2^2$$

more generally

bias

$$h_w(x) = w_0 + \sum_{i,k=1}^{n,p} w_k \phi_k(x_i)$$

$= b$

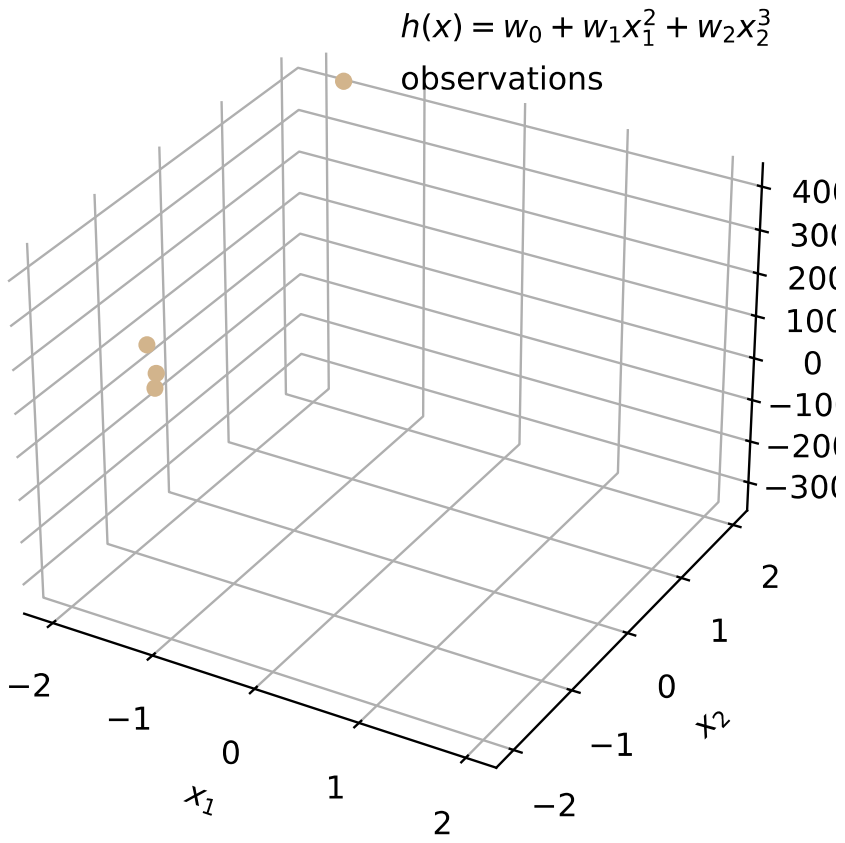
$i = 1, \ldots, \boldsymbol{n}$	data points
$k = 1, \ldots, \boldsymbol{p}$	features
$x_{0i} = 1$	pseudo-variable
$\phi_k(x_i) = x_{ki}$	basis function

Univariate linear regression (one prediction)

$i = 1, \ldots, n$ data points
$k = 1, \ldots, p$ features

Multiple (several features)

We want to fit a set of points $(x_i,\ y_i)$



Hypothesis function

$$h_w(x_1, x_2) = w_0 + w_1 x_1^2 + w_2 x_2^2$$

more generally

bias

$$h_w(x) = w_0 + \sum_{i,k=1}^{n,p} w_k \phi_k(x_i)$$

basis function

Allows to model nonlinearity in the data while keeping linearity in the parameters w

$\phi(x) = (x^0, x^1, x^2, \cdots, x^p)$

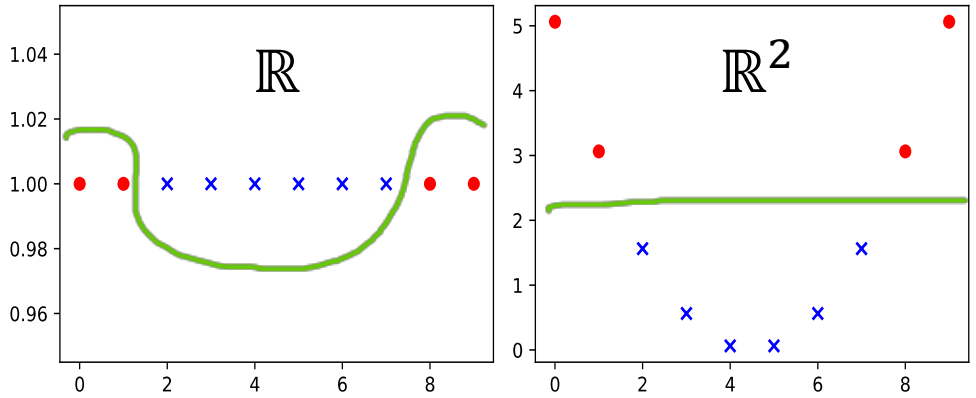
Polynomial basis function

Brief interlude

Deep learning can be viewed as adaptive basis function learning at scale

Basis functions $\phi(x)$

- Used for ***feature mapping*** $\phi : \mathbb{R} \rightarrow \mathbb{R}^L$
- They transform the input in a nonlinear way to a new space to capture more information about the data (e.g., make it linearly separable, easier to model):



They can be combined to fit complex nonlinear functions using ***linear regression***

Kernel functions $k(x_i, x')$

- Also used for feature mapping to higher dimensional spaces, where the number of basis functions would be too high and computationally expensive.
- Prediction for unseen data x' is done by measuring the similarity between x' and the training data x_i
  - *Similarity = inner product* $\begin{cases} x, x' \in \mathcal{X} \\ k: \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R} \end{cases}$

Example: prediction for x' in a binary classifier

$$\hat{y} = \text{sgn} \sum_{i=1}^{n} w_i y_i \, \boldsymbol{k(x_i, x')}$$

Often:

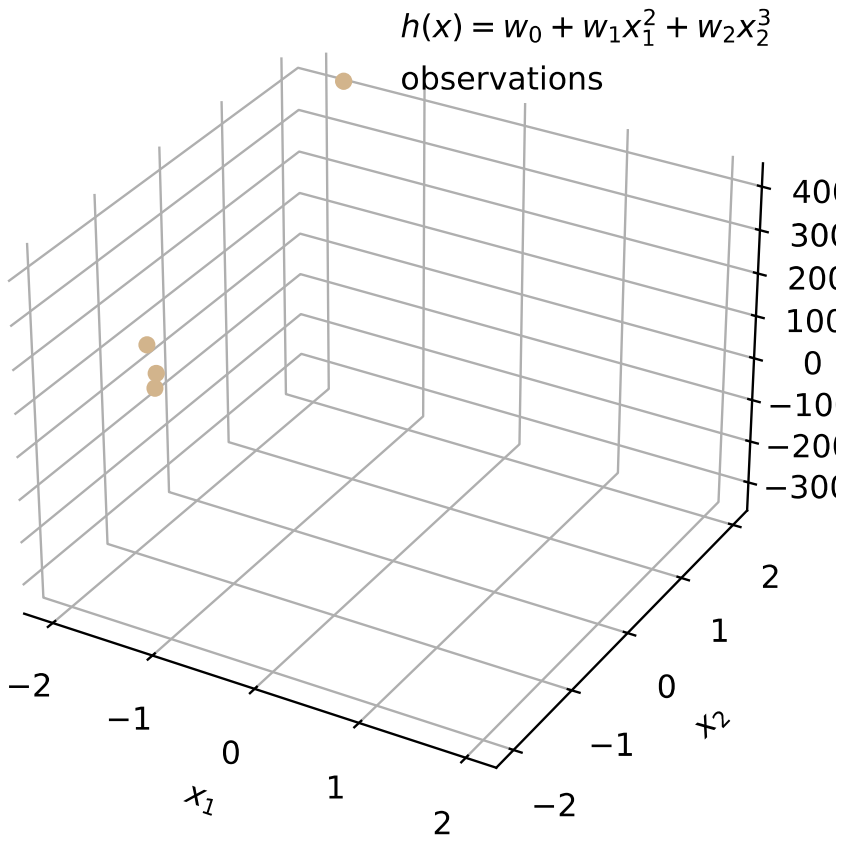
$$k(x, x') = \langle \phi(x), \phi(x') \rangle = \phi(x)^T \phi(x') \quad \phi : \mathcal{X} \rightarrow \mathcal{V}$$

Univariate regression (one prediction)

$i = 1, \dots, n$ data points
$k = 1, \dots, p$ features

Multiple (several features)

We want to fit a set of points $(x_i,\ y_i)$



Matrix notation

$$h_w(x) = W^T X$$

=1

$$\boldsymbol{X^T} = \begin{bmatrix} x_{01} & x_{11} & \cdots & x_{p1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{0n} & x_{1n} & \cdots & x_{pn} \end{bmatrix}$$

$[n \times (p+1)]$

$$\boldsymbol{W} = \begin{bmatrix} w_0 \\ \vdots \\ w_p \end{bmatrix}$$

bias included

$[(p+1) \times 1]$

How to iteratively find the correct fit?

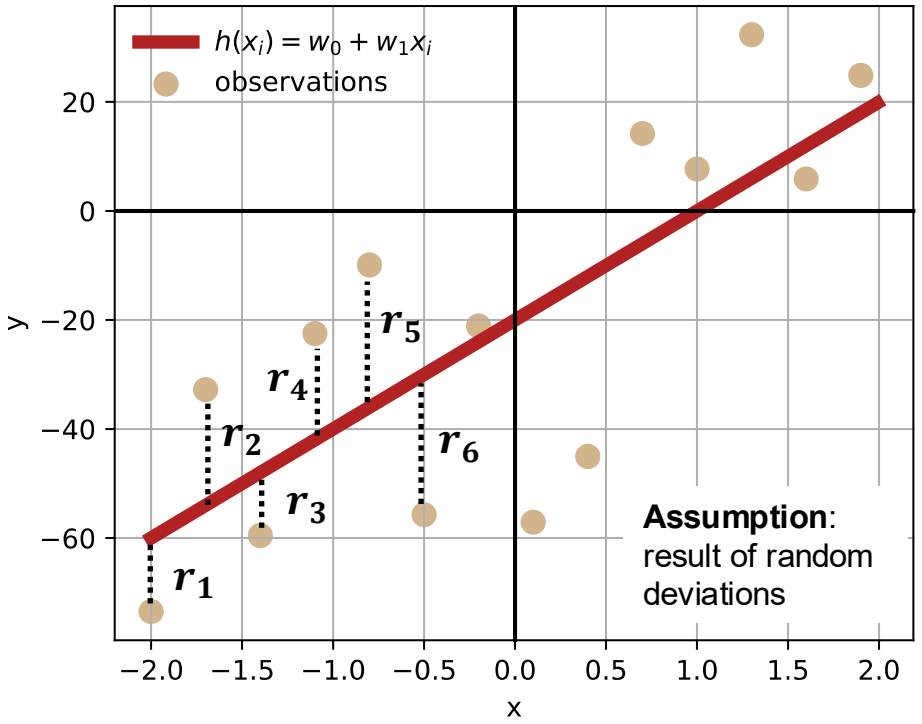
By minimizing the loss function

Goal: choose w_k such that $\boldsymbol{h_w(x_i)}$ is as close to y_i as possible

$$\boldsymbol{h_w(x_i)} - y_i = r_i$$

residual

error



Loss function $J_W(\hat{y}, \mathbf{y})$

Also called cost function or error function

→ Number representing how well your algorithm models your dataset

Quantifies the error between the predicted values $\hat{\mathbf{y}}$ by a model and the target values $\mathbf{y}$ of the data it is trying to fit or predict given your current weights $\mathbf{w}$

*The goal of training a model is to find the parameters w that **minimize the loss function***

The loss is monitored during the training of NNs

MAE (L1 loss)

MSE (L2 loss)

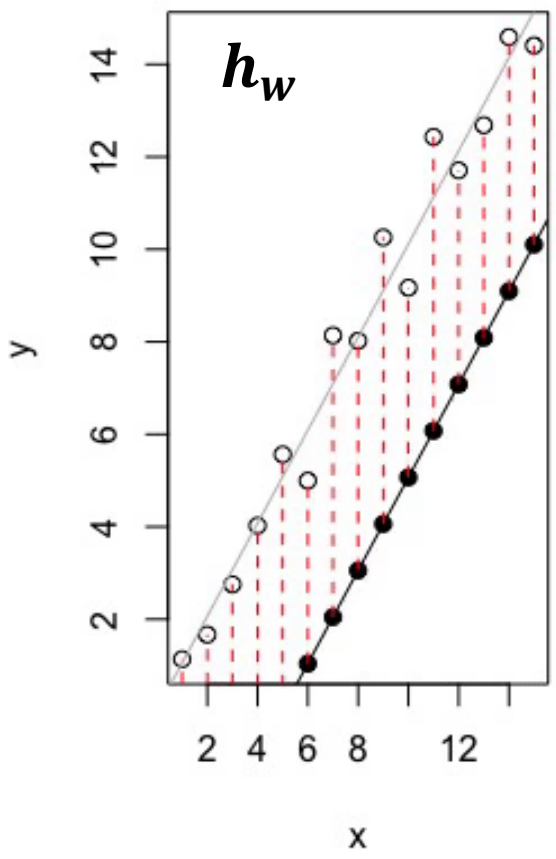
Negative log-likelihood function

Cross-entropy loss

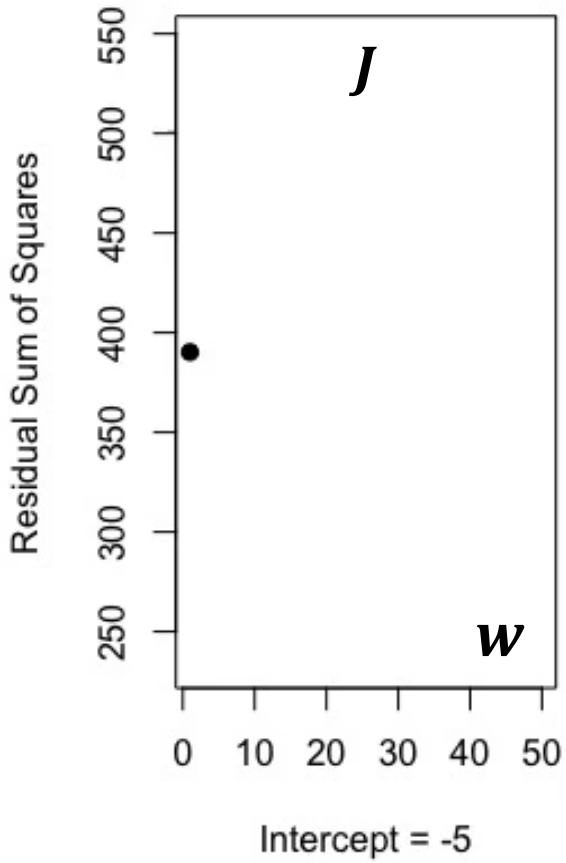
Example: least squares fitting

Common parameter estimation in regression analysis

Scan of hypothesis



Calculation of loss



Animation from: https://yihui.org/animation/example/least-squares/

Goal: find the weights that minimize $\boldsymbol{J}$

$$\sum_{i=1}^{n} \boldsymbol{r_i^2} = \sum_{i=1}^{n} (h_w(x_i) - y_i)^2 = |\boldsymbol{X}w - \boldsymbol{y}|^2 = \boldsymbol{J}(\boldsymbol{w})$$

$$\underset{\mathbf{w}}{\operatorname{argmin}}\, J(\boldsymbol{w})$$

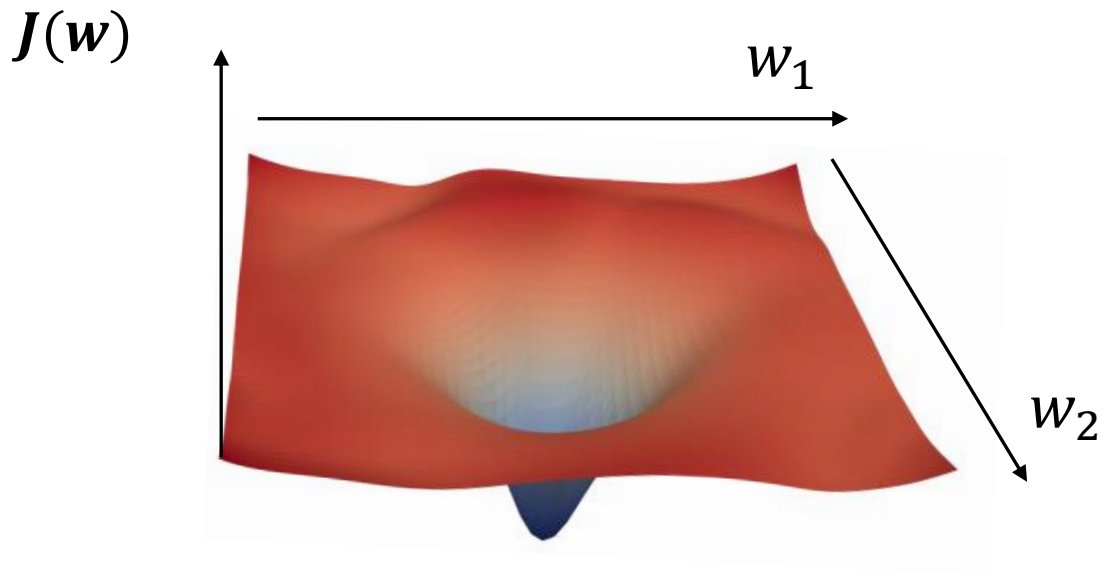
It depends on the weights (parameters) of your model

⚠ **Use least squares when:**

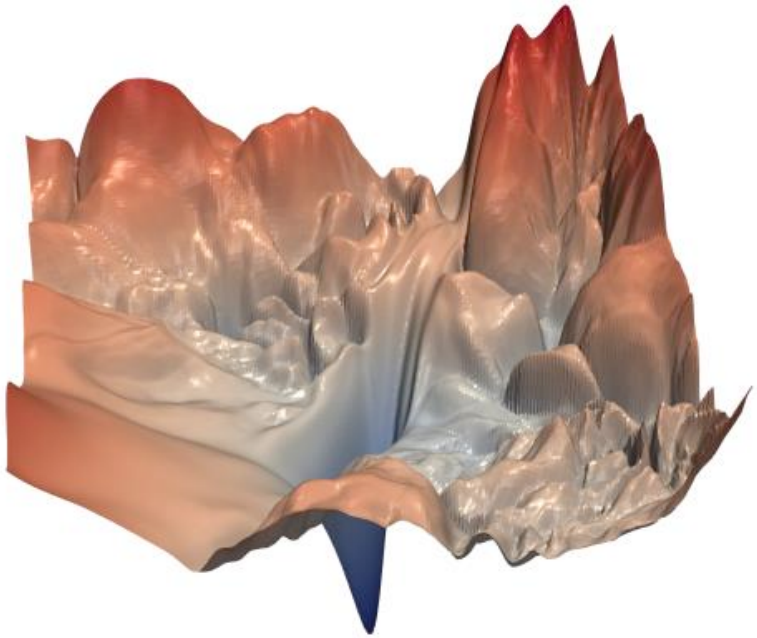
- System is overdetermined
  - Points > features $(n > p + 1)$
- Uncertainties in the data are "controlled" — Why?
  - ➢ Otherwise: maximum likelihood estimation, ridge regression, lasso regression, least absolute deviation, bayesian linear regression, etc.

The “loss landscape”

This is a simple loss landscape for two weights:



Convex problems have a single global minimum and are much easier to optimise



Deep networks typically produce highly non-convex loss landscapes with many saddle points and local minima

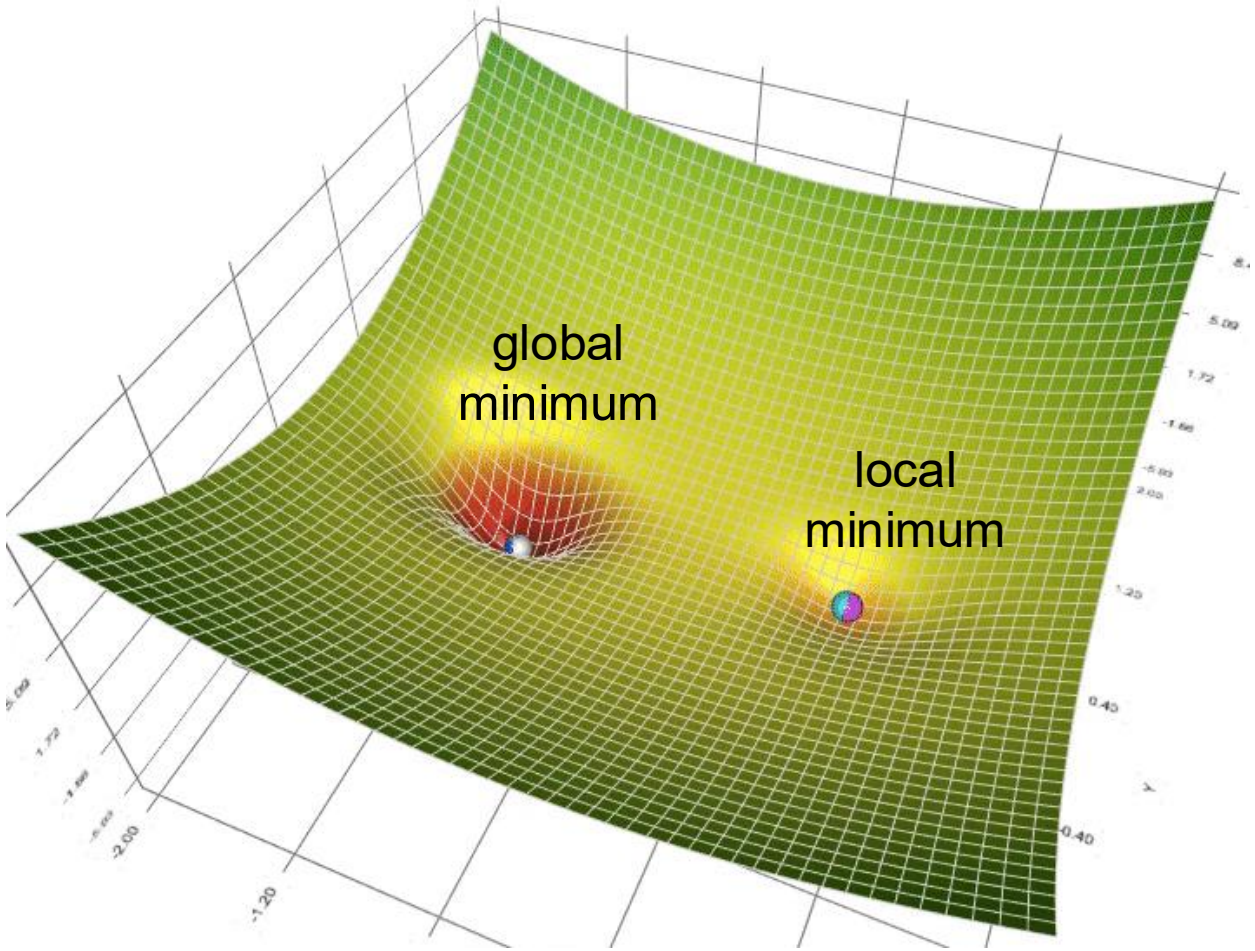
Remarks:

- We never see it like this, we only observe the loss along the optimisation trajectory
- In practice, the parameter space is extremely high dimensional (millions to billions of parameters)

https://arxiv.org/pdf/1712.09913.pdf

How do we find our way in the loss landscape?

With optimisers!



Animation from Lili Jiang

Gradient based methods (first order):

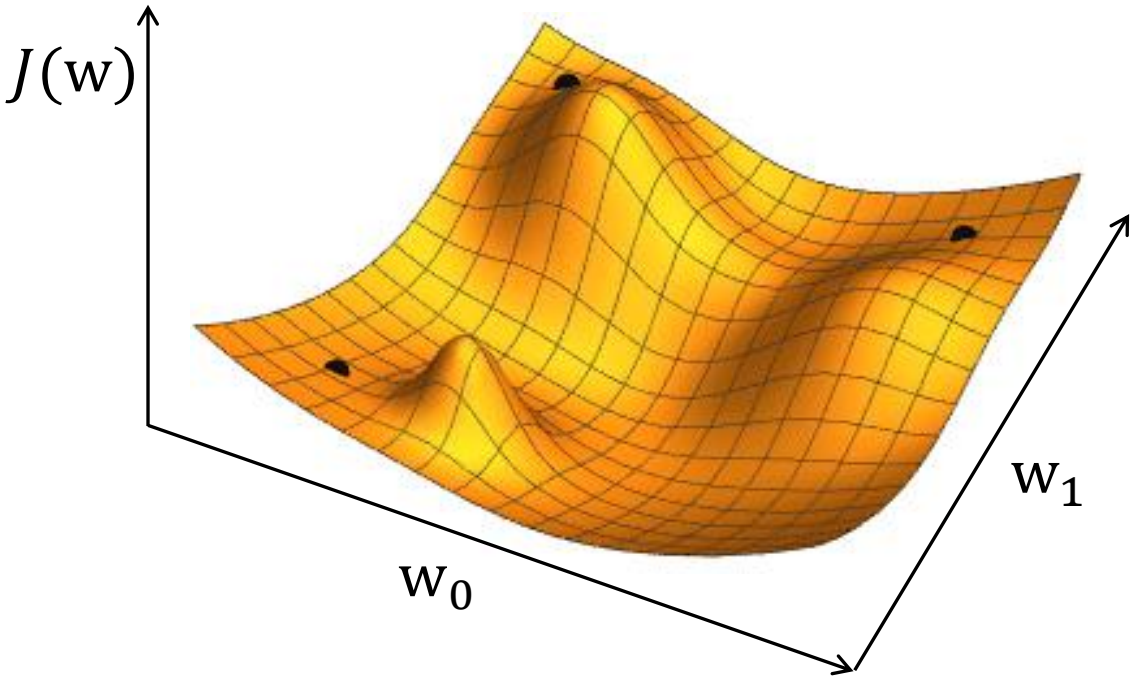
- Gradient descent
- momentum
- AdaGrad
- RMSProp
- Adam

dynamic adjustment of algorithm parameters

More later!

Gradient descent

Algorithm to iteratively solve $\underset{\boldsymbol{w}}{\operatorname{argmin}} J(\boldsymbol{w})$



Animation from Wikimedia commons

Update rule:

step size
learning rate

first order

$$\boldsymbol{w} := \boldsymbol{w} - \alpha \boldsymbol{\nabla} J(\mathrm{w}_k)$$

take repeated steps in the opposite direction of the gradient

$$J(\mathrm{w}_k) = \frac{1}{2n}\sum_{i=1}^{n}(h_{\mathrm{w}}(x_i) - y_i)^2$$

Loss function

$$h_{\mathrm{w}}(x) = \mathrm{w}_0 + \mathrm{w}_1 x$$

Hypothesis

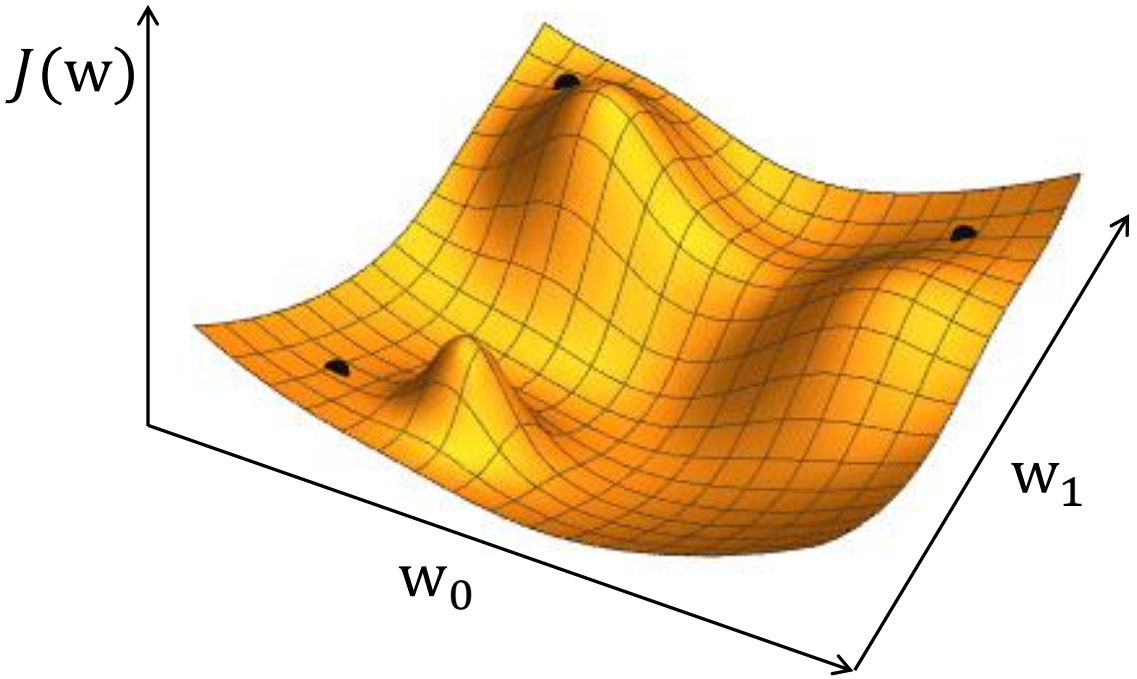
$$\vec{\nabla} = \left(\frac{\partial}{\partial w_1}, \ldots, \frac{\partial}{\partial w_k}\right)$$

$$\frac{\partial J(\mathrm{w}_0, \mathrm{w}_1)}{\partial \mathrm{w}_0} = \frac{1}{n}\sum_{i=1}^{n}(\mathrm{w}_0 + \mathrm{w}_1 x_i - y_i)$$

$$\frac{\partial J(\mathrm{w}_0, \mathrm{w}_1)}{\partial \mathrm{w}_1} = \frac{1}{n}\sum_{i=1}^{n} x_i(\mathrm{w}_0 + \mathrm{w}_1 x_i - y_i)$$

Gradient descent

Algorithm to iteratively solve $\underset{\boldsymbol{w}}{\operatorname{argmin}} J(\boldsymbol{w})$



Animation from Wikimedia commons

Update rule:

step size
learning rate

first order

$$\boldsymbol{w} := \boldsymbol{w} - \alpha \boldsymbol{\nabla} J(\mathrm{w}_k)$$

take repeated steps in the opposite direction of the gradient

$$\frac{\partial J(\mathrm{w}_0, \mathrm{w}_1)}{\partial \mathrm{w}_0} = \frac{1}{n}\sum_{i=1}^{n}(\mathrm{w}_0 + \mathrm{w}_1 x_i - y_i)$$

$$\frac{\partial J(\mathrm{w}_0, \mathrm{w}_1)}{\partial \mathrm{w}_1} = \frac{1}{n}\sum_{i=1}^{n} x_i(\mathrm{w}_0 + \mathrm{w}_1 x_i - y_i)$$

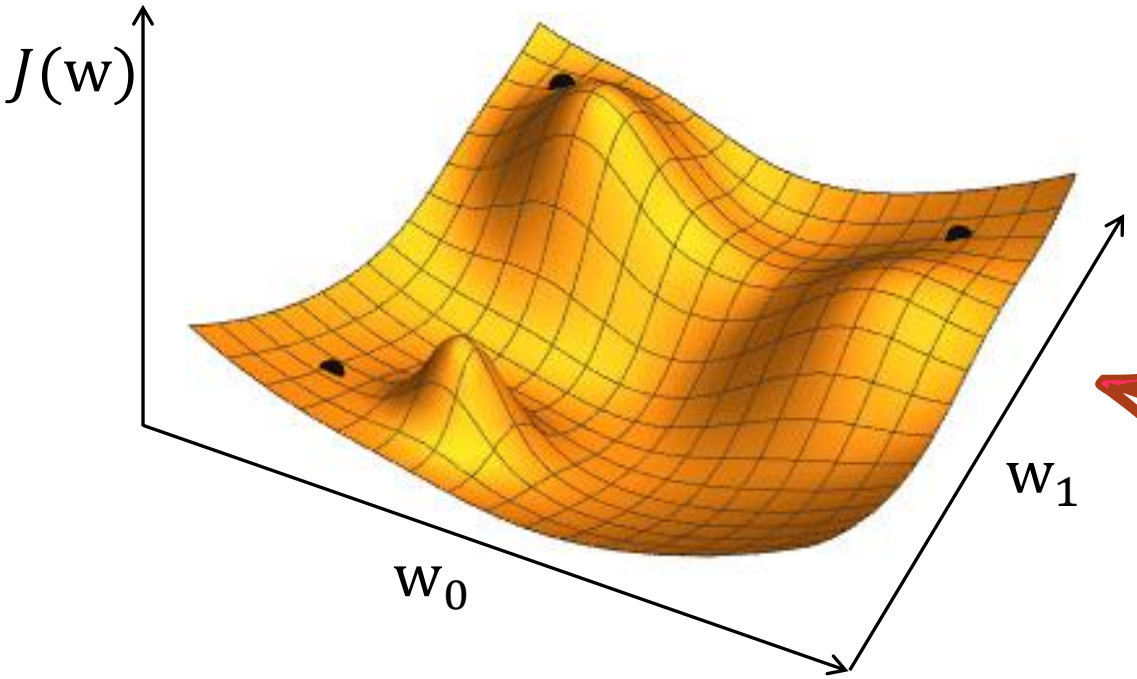
$$\mathrm{w}_0 := \mathrm{w}_0 - \alpha \frac{\partial J(\mathrm{w}_0, \mathrm{w}_1)}{\partial \mathrm{w}_0}$$

$$\mathrm{w}_1 := \mathrm{w}_1 - \alpha \frac{\partial J(\mathrm{w}_0, \mathrm{w}_1)}{\partial \mathrm{w}_1}$$

updated simultaneously

Gradient descent

Algorithm to <u>iteratively</u> solve $\underset{\boldsymbol{w}}{\operatorname{argmin}} J(\boldsymbol{w})$



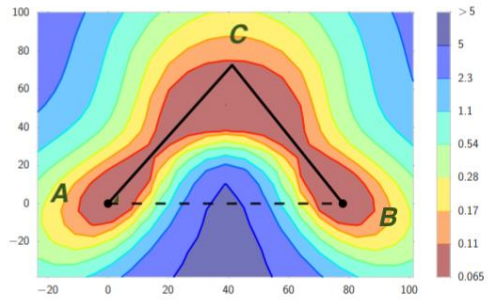
Animation from Wikimedia commons

General remarks

- Works in any number of dimensions.
- Depending on the initial $(\mathrm{w}_0, \mathrm{w}_1)$ optimization can end up at different points.
- It proceeds very slowly near saddle points but can eventually escape them if weights were randomly initialized.
- In highly nonconvex loss landscapes, gradient descent often finds near-optimal solutions.

Recent research shows that simple, low-cost paths connect different optima

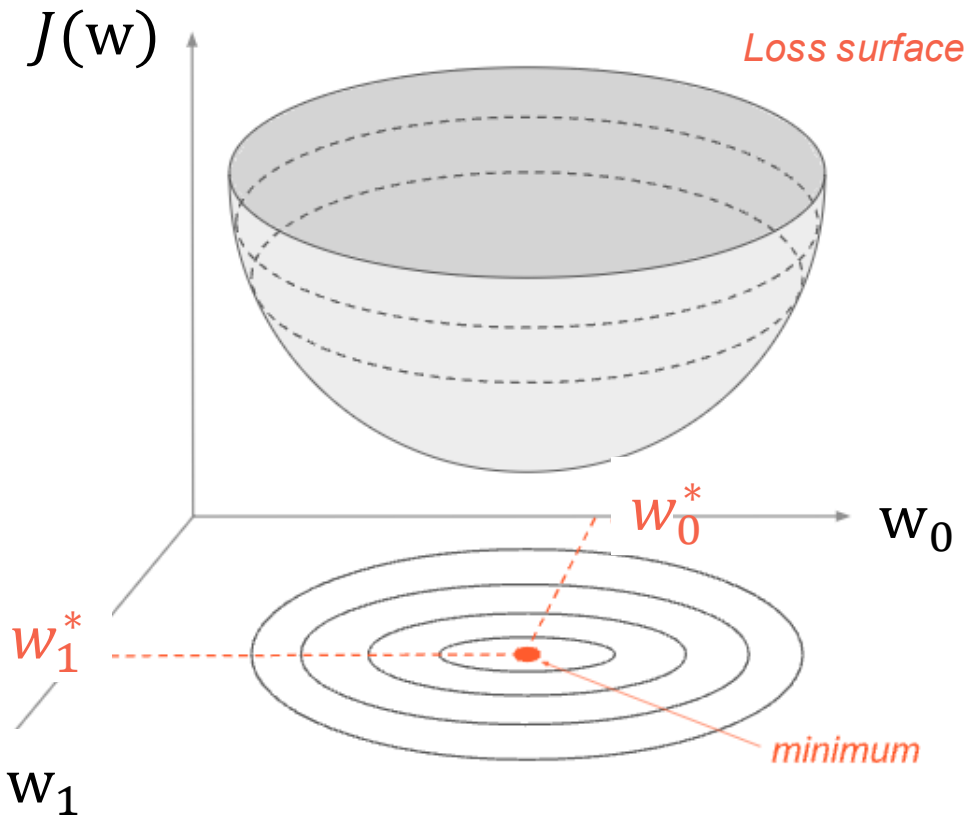
→ Low loss connected manifold



Garipov et al. 2018
Draxler et al. 2019

Gradient descent

Algorithm to iteratively solve $\underset{\boldsymbol{w}}{\operatorname{argmin}} J(\boldsymbol{w})$



Matrix notation

$$J(W) = \frac{1}{2}(XW - Y)^T(XW - Y)$$

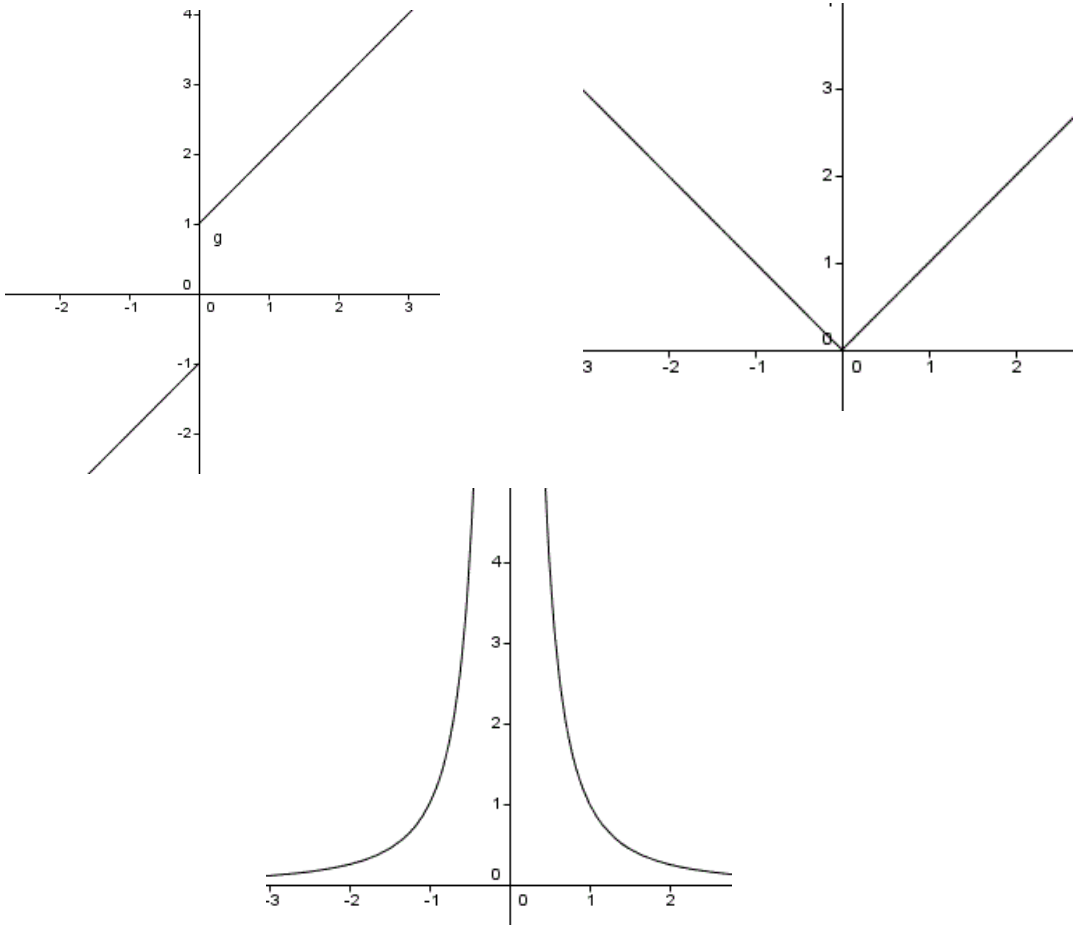
$$= \frac{1}{2}(\underbrace{W^T X^T XW}_{\text{cuadratric}} - \underbrace{2Y^T XW}_{\text{linear}} + \underbrace{Y^T Y}_{\text{constant}})$$

Analytically: $W = (X^T X)^{-1} X^T Y$

Iteratively: $\frac{\partial J}{\partial W} = X^T(X^T W - Y)$

Image from https://allmodelsarewrong.github.io/gradient.html

Gradient descent: some limitations



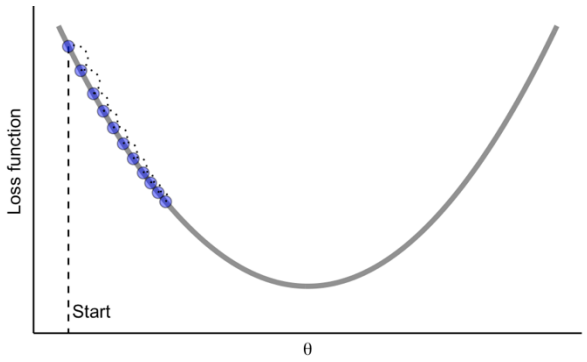
- Sensitivity to learning rate (next slide)
- Slow convergence in areas with small gradients (plateaus)
- Very dependent on weight initialisation
- Computationally intensive for large datasets
- Only works when the function is continuously differentiable (gradient needs to exist)

Gradient descent: remarks

Step size / learning rate $\boldsymbol{w} := \boldsymbol{w} - \boldsymbol{\alpha}\, \boldsymbol{\nabla} J(\mathrm{w}_k)$

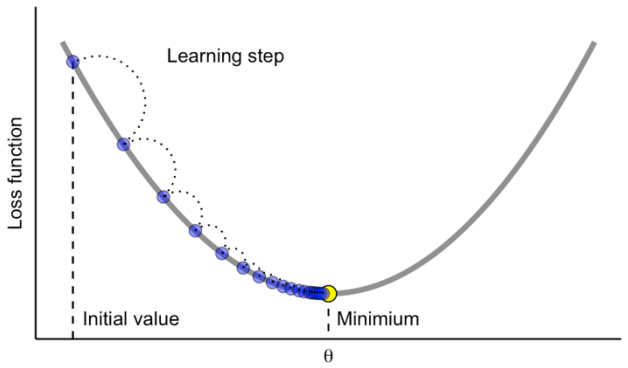
Too small

Converges slowly



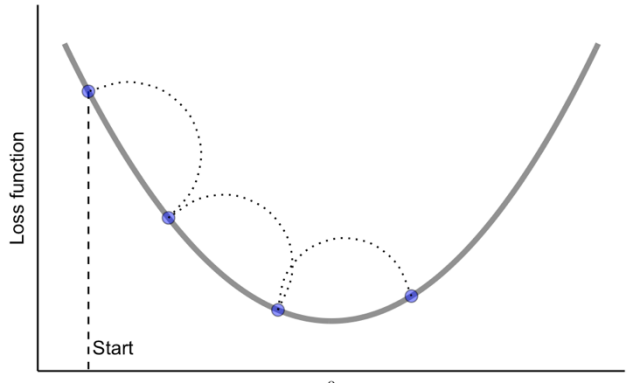
Just right

Converges quickly



Too large

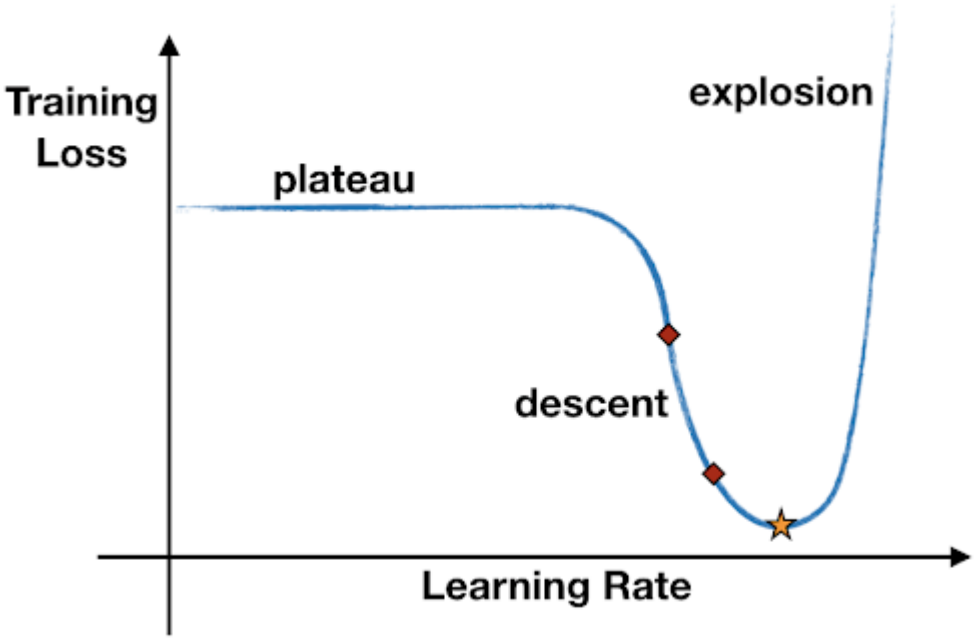
Diverges



Images from https://bradleyboehmke.github.io

Gradient descent: remarks

Step size / learning rate $\boldsymbol{w := w - \alpha \nabla J(w_k)}$



Modern algorithms have adaptive learning rates

- The optimizer is working correctly if the loss decreases after every iteration.
- The recommended minimum learning rate is the value where the loss decreases the fastest
- Usual to **declare convergence tests.** (e.g., declare convergence when $J(\theta)$ decreases by less than 10^{-3} in one iteration)

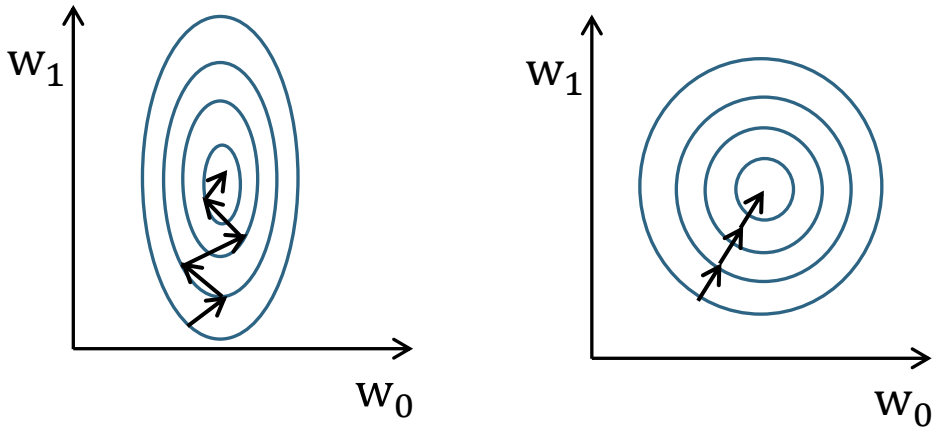
Images from https://blog.dataiku.com/

Gradient descent: remarks

Feature scaling

- Gradient-based optimizers converge faster when features have similar scale
- Becomes critical in polynomial regression due to large differences in feature magnitude:

$$w_0 + w_1 x + w_2 x^2 + w_3 x^3$$



Normalization

$$-1 \leq x_i \leq 1 \quad \text{or} \quad 0 \leq x_i \leq 1$$

To rescale between [a, b]:

$$x' = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)}$$

- Geometrically: will scale the n-dimensional data into an n-dimensional hypercube.
- Sensitive to outliers, works best when extreme values are not dominant

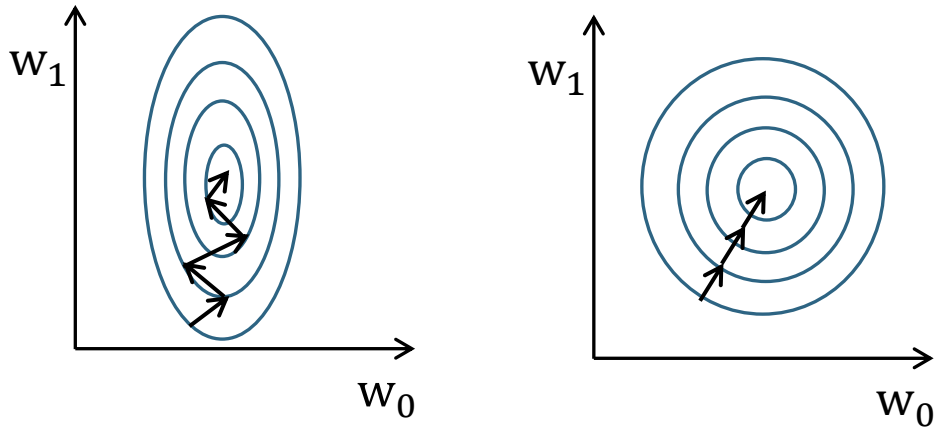
Scaling reduces anisotropy in the loss landscape

Gradient descent: remarks

Feature scaling

- Gradient-based optimizers converge faster when features have similar scale
- Becomes critical in polynomial regression due to large differences in feature magnitude:

$$w_0 + w_1 x + w_2 x^2 + w_3 x^3$$



Standardization

$$x' = \frac{x-\mu}{\sigma} \quad \text{so that} \quad \mu = 0,\ \sigma^2 = 1$$

- Geometrically: centers the data and rescales its variance
- Particularly appropriate when the data is approximately Gaussian
- Sensitive to outliers (mean and variance are affected)

BUT WHEN ARE WE GOING TO LEARN ABOUT NEURAL NETWORKS??

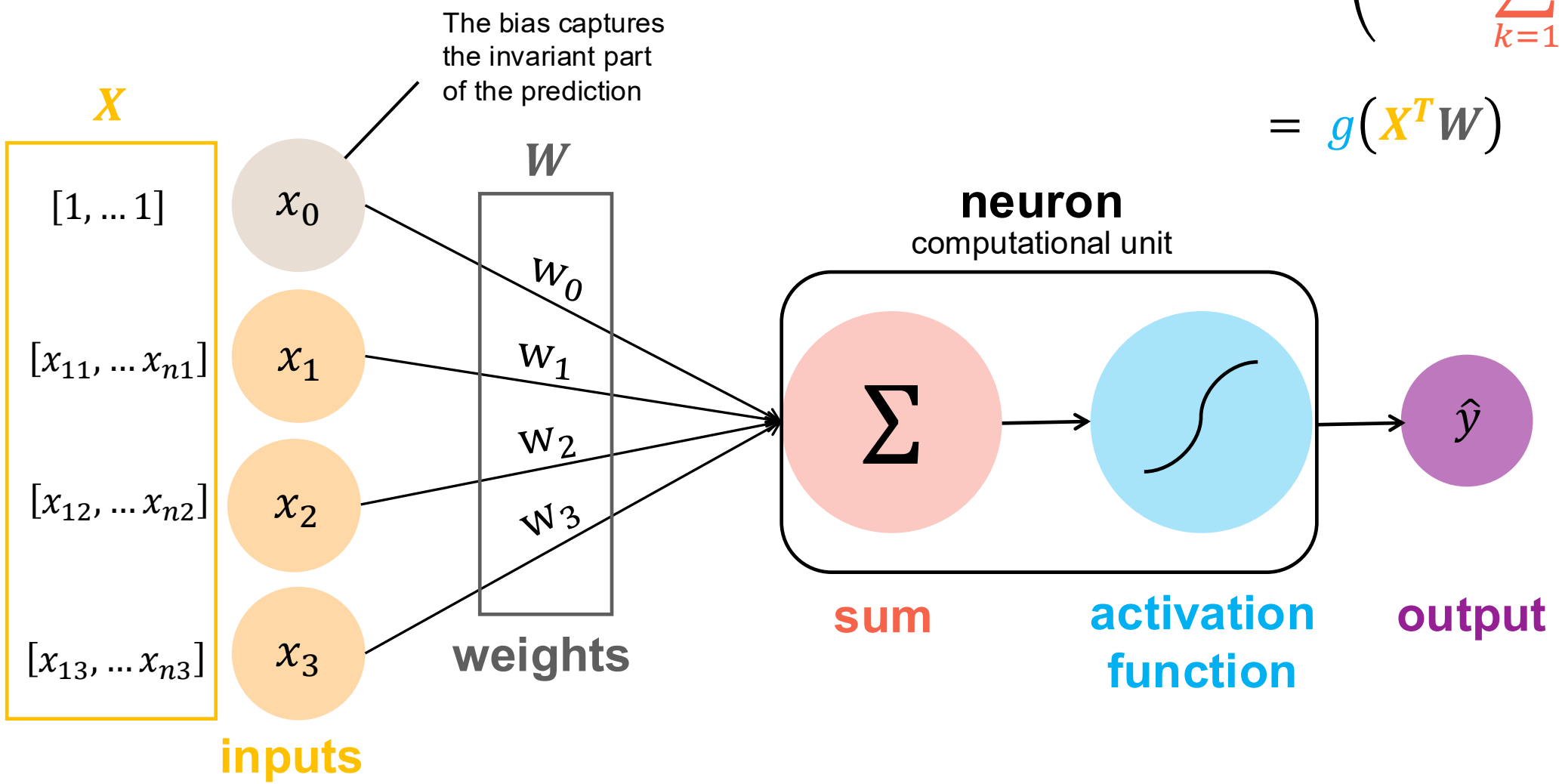
Neural networks

Linear algebra, multivariate calculus, probability & statistics, optimization theory

Neural networks

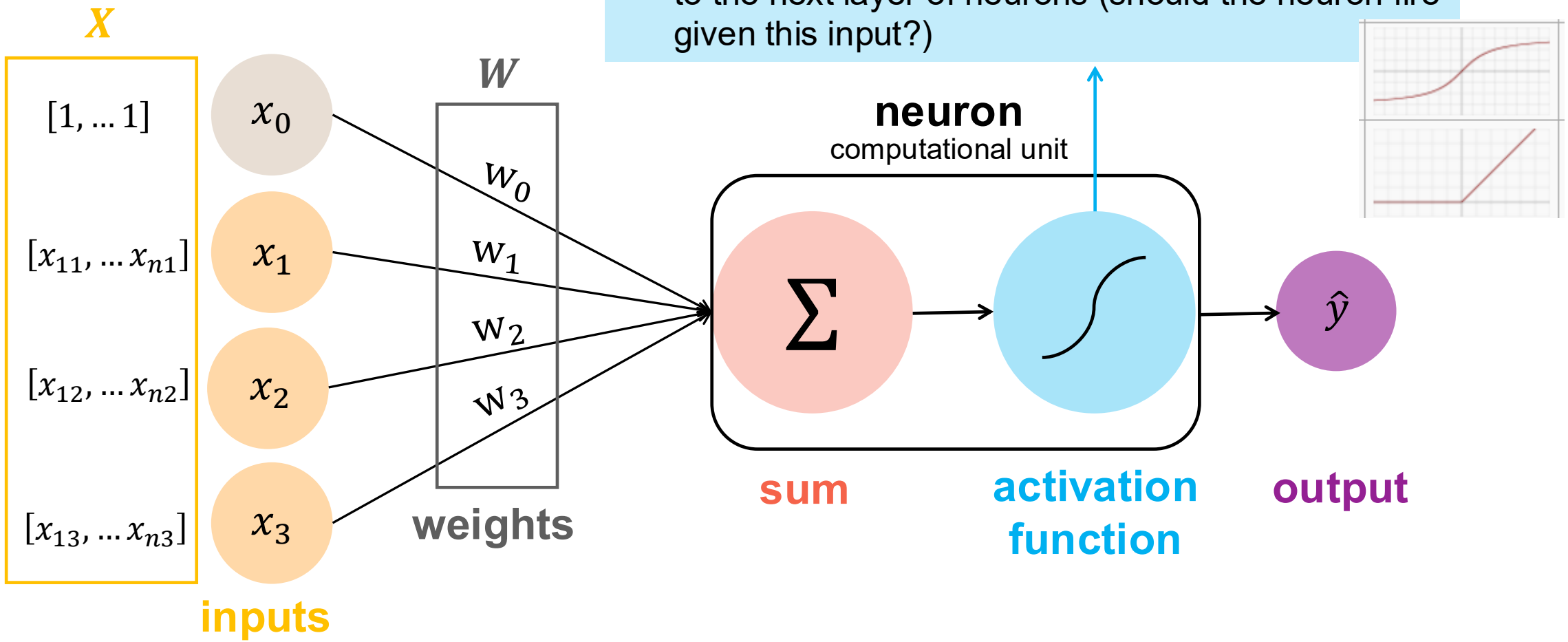
$$\hat{y} = g\left(w_0 + \sum_{k=1}^{p} x_i\, w_i\right)$$

$$= g\left(X^T W\right)$$



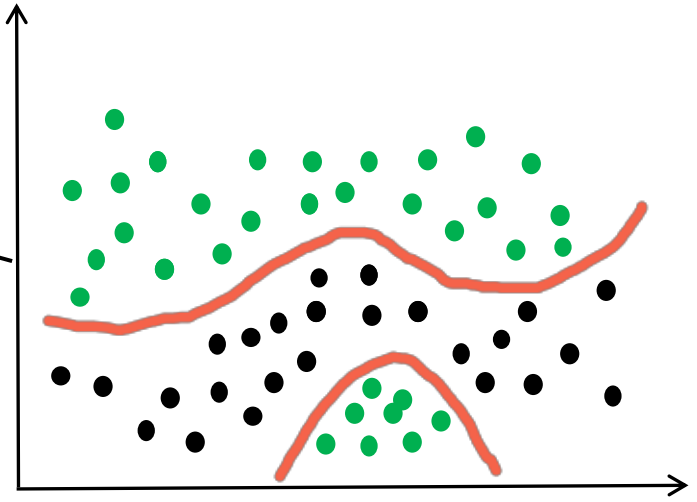
Neural networks

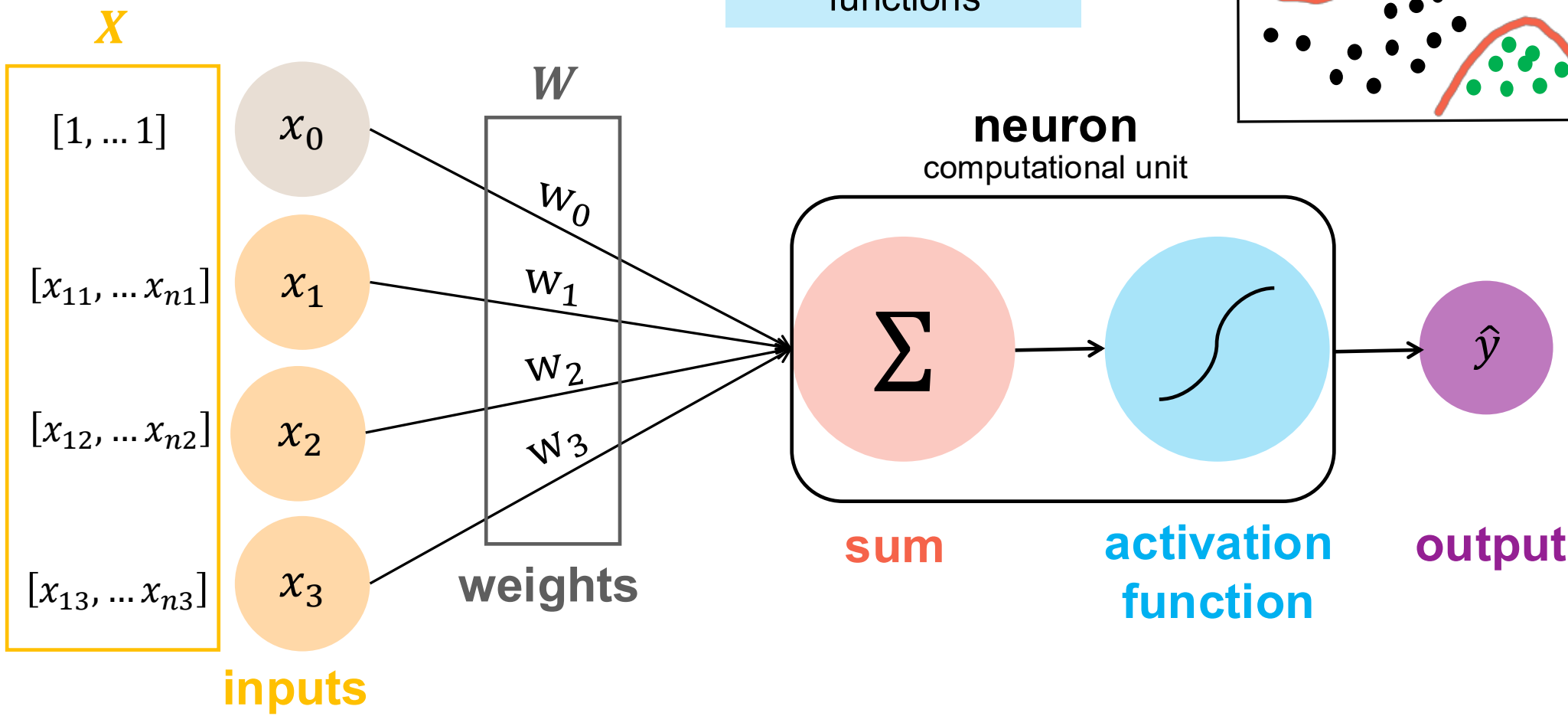
- What makes NNs different from linear regression.
- Defines the output of the neuron given an input.
- "**Mathematical gate**" that filters which data passes to the next layer of neurons (should the neuron fire given this input?)



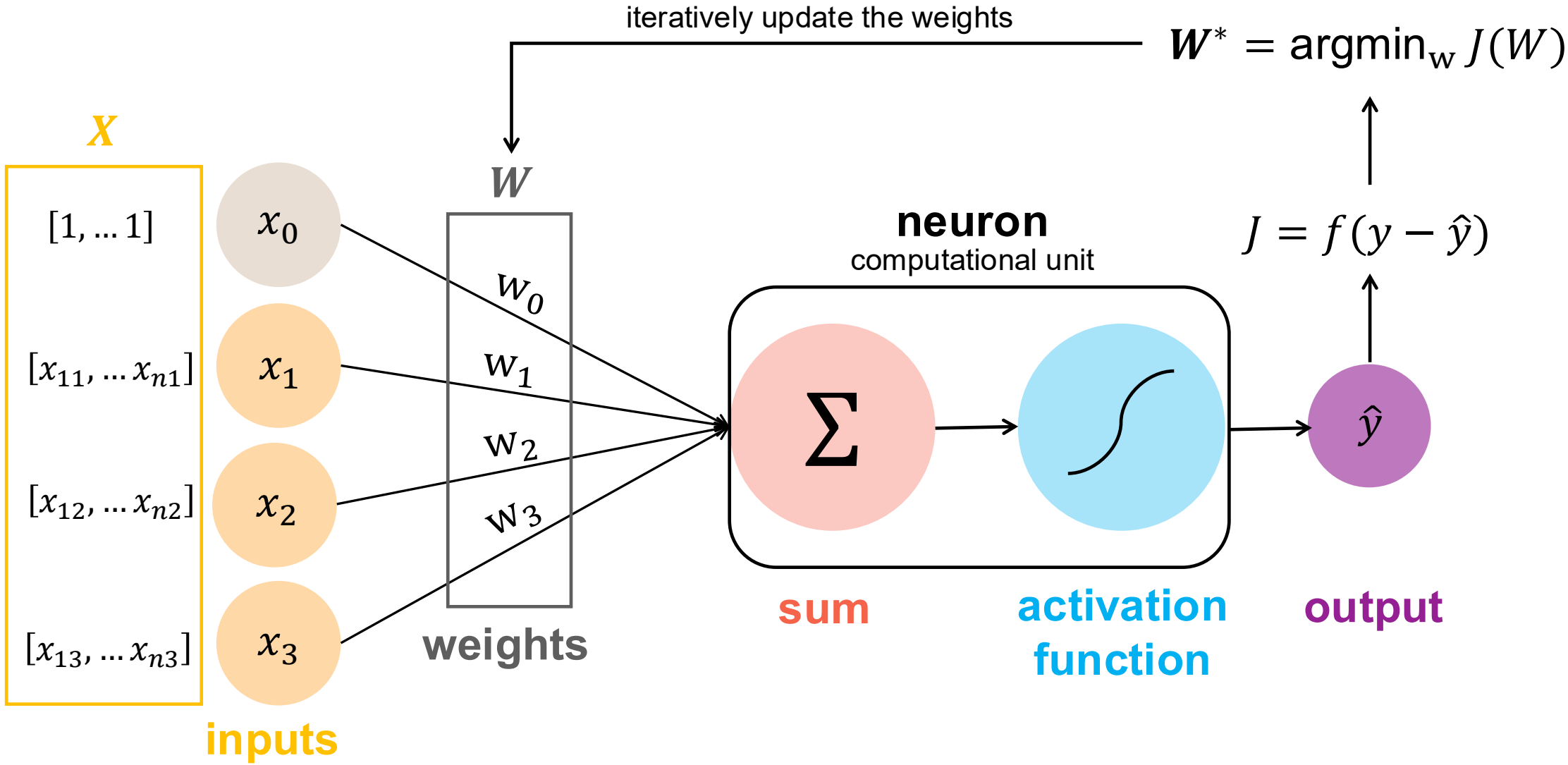
Neural networks

introduces nonlinearities to approximate arbitrarily complex functions



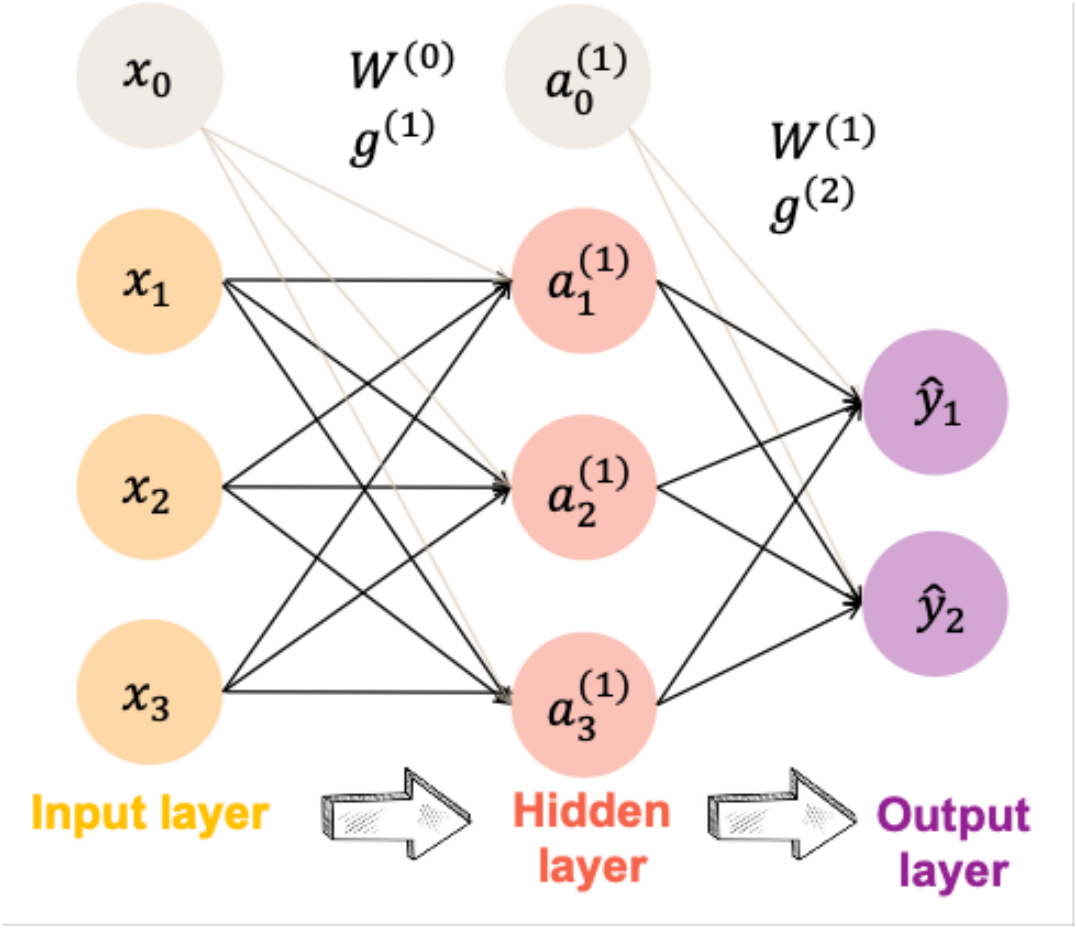


Neural networks



Forward propagation

Single layer neural network



New notation:

- **Superscript** = layer number
- **Subscript** = neuron number
- z = pre-activation vector
- a = activation vector
- g = activation function

$$\hat{y} = g^{(2)}(\boldsymbol{W}^{(1)}\overbrace{g^{(1)}(\underbrace{\boldsymbol{W}^{(0)}\boldsymbol{X}}_{\boldsymbol{z}^{(1)}})}^{\boldsymbol{a}^{(1)}})$$

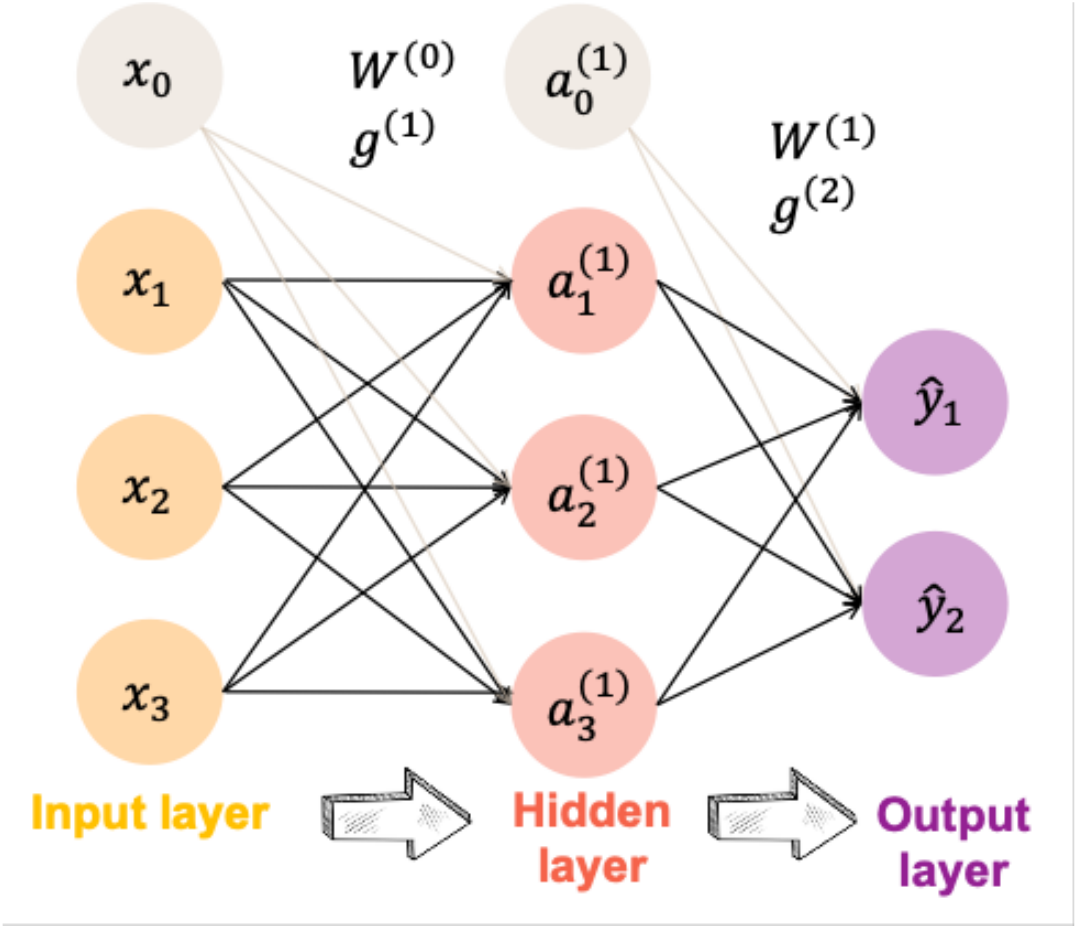
A multilayer network is just repeated affine transformation followed by elementwise nonlinearity. Mathematically, it is a composition of functions

Forward propagation

Single layer neural network

$$\hat{y} = g^{(2)}(\boldsymbol{W}^{(1)} \underbrace{g^{(1)}(\boldsymbol{W}^{(0)}\boldsymbol{X})}_{\boldsymbol{z}^{(1)}})$$

$\boldsymbol{a}^{(1)}$



Step 1

$$\boldsymbol{W}^{(0)} = \begin{bmatrix} w_{10}^{(0)} & w_{11}^{(0)} & w_{12}^{(0)} & w_{13}^{(0)} \\ w_{20}^{(0)} & w_{21}^{(0)} & w_{22}^{(0)} & w_{23}^{(0)} \\ w_{30}^{(0)} & w_{31}^{(0)} & w_{32}^{(0)} & w_{33}^{(0)} \end{bmatrix}$$

neurons in hidden layer

neurons in input layer + bias

$$\boldsymbol{X} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

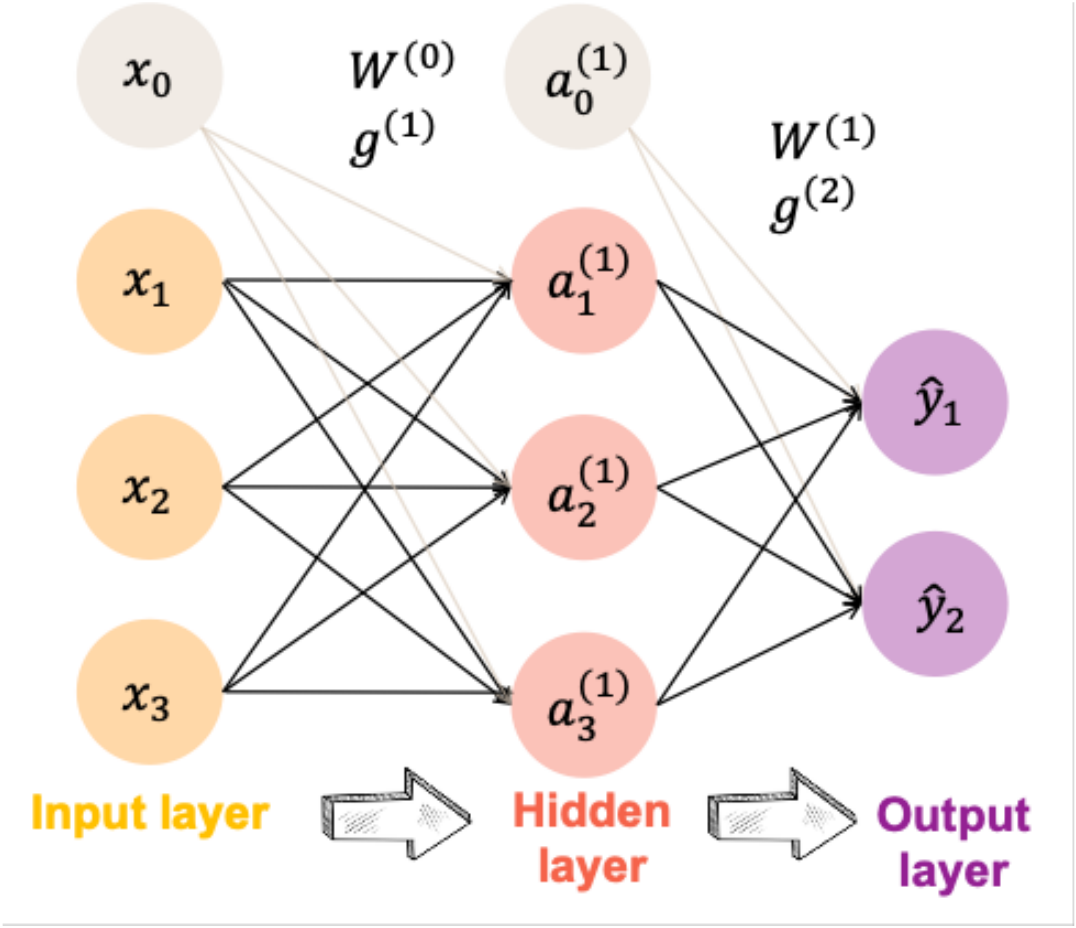
$x_1 = (x_{11}, \ldots, x_{1n})$

$$\boldsymbol{a}^{(1)} = \begin{bmatrix} a_0^{(1)} \\ g^{(1)}(w_{10}^{(0)} x_0 + w_{11}^{(0)} x_1 + w_{12}^{(0)} x_2 + w_{13}^{(0)} x_3) \\ g^{(1)}(w_{20}^{(0)} x_0 + w_{21}^{(0)} x_1 + w_{22}^{(0)} x_2 + w_{23}^{(0)} x_3) \\ g^{(1)}(w_{30}^{(0)} x_0 + w_{31}^{(0)} x_1 + w_{32}^{(0)} x_2 + w_{33}^{(0)} x_3) \end{bmatrix}$$

Forward propagation

Single layer neural network

$$\hat{y} = g^{(2)}(\boldsymbol{W}^{(1)}\overbrace{g^{(1)}(\boldsymbol{W}^{(0)}\boldsymbol{X})}^{\boldsymbol{a}^{(1)}})$$



Step 2

$$w^{(1)} = \underbrace{\begin{bmatrix} w_{10}^{(1)} & w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{20}^{(1)} & w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \end{bmatrix}}_{\text{neurons in hidden layer + bias}} \} \text{neurons in output layer} \qquad a^{(1)} = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix}$$

$$a^{(2)} = \hat{y} = \begin{bmatrix} g^{(2)}(w_{10}^{(1)} a_0^{(1)} + w_{11}^{(1)} a_1^{(1)} + w_{12}^{(1)} a_2^{(1)} + w_{13}^{(1)} a_3^{(1)}) \\ g^{(2)}(w_{20}^{(1)} a_0^{(1)} + w_{21}^{(1)} a_1^{(1)} + w_{22}^{(1)} a_2^{(1)} + w_{23}^{(1)} a_3^{(1)}) \end{bmatrix}$$

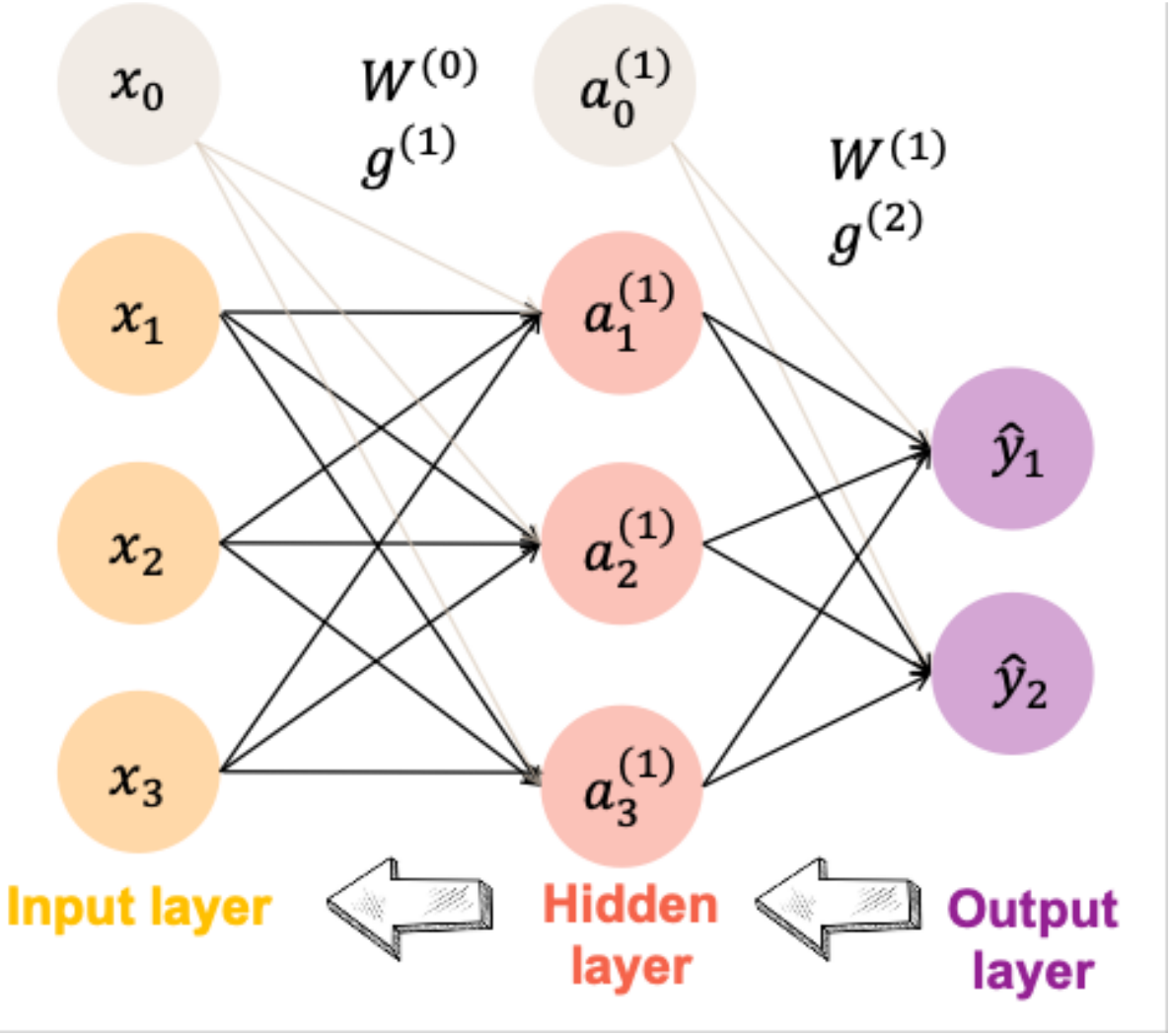
✅ Forward propagation completed

Backpropagation

Single layer neural network

We want to find the network weights that achieve the lowest loss (training)

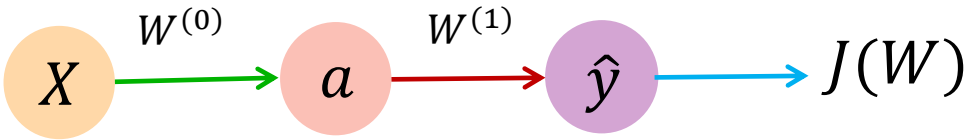
$\boldsymbol{W}^* = \text{argmin}_w J(\boldsymbol{W});\ \boldsymbol{W} = \{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}\}$



With gradient descent

$$W \leftarrow W - \alpha \frac{\partial J(W)}{\partial W}$$

Calculating gradients with the chain rule



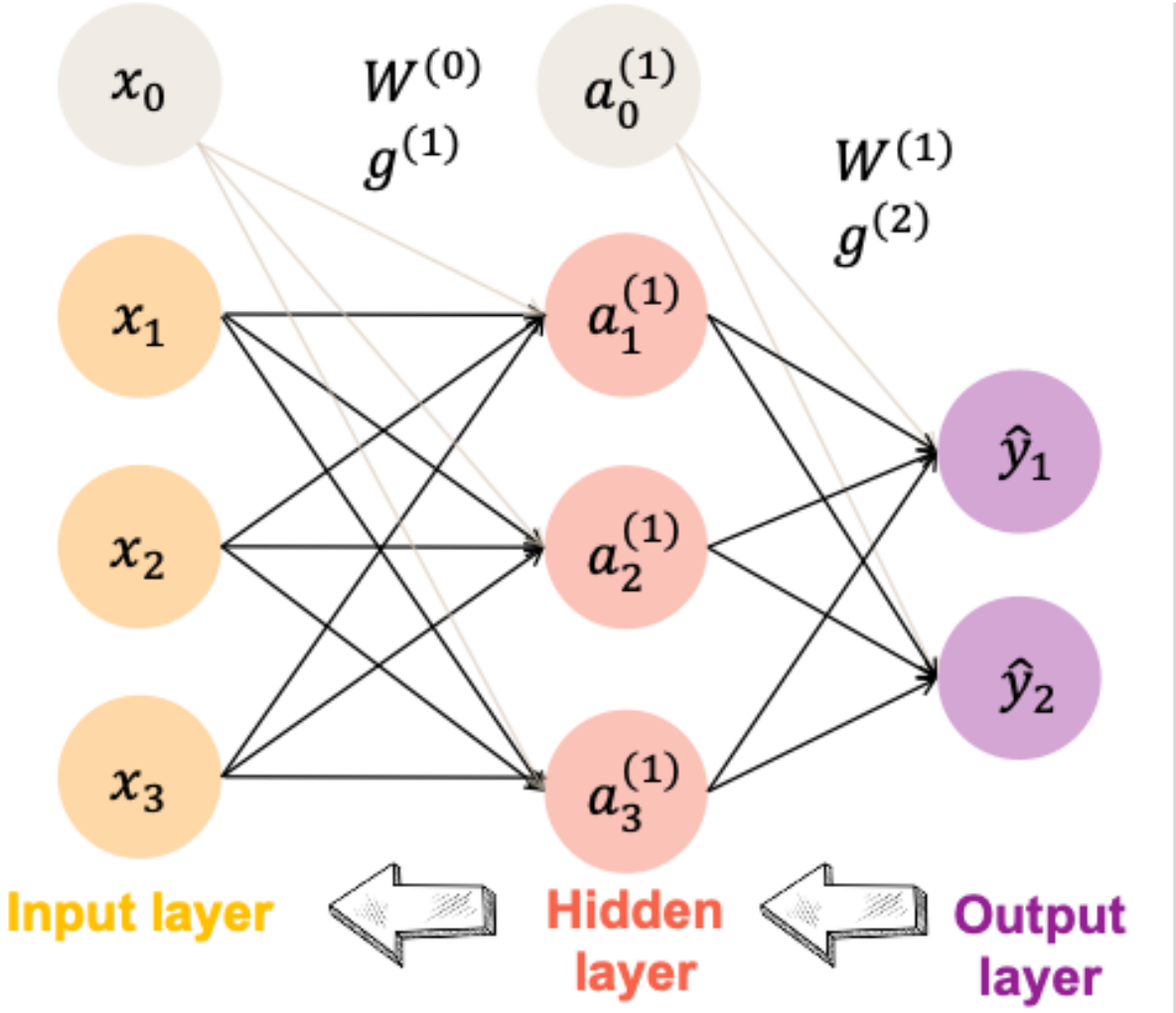
- We need to consider the sequential nature of the operations in a neural network (layer by layer) to calculate the new weights, as they depend on each other.
- We can do this by applying the chain rule to the gradients, since we have a composition of functions.

Backpropagation

Single layer neural network

We want to find the network weights that achieve the lowest loss

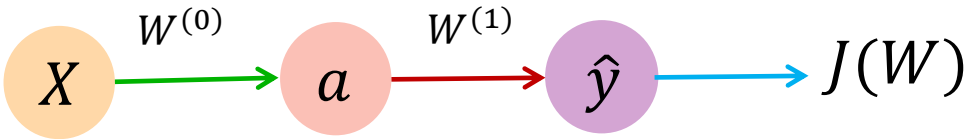
$$\boldsymbol{W}^* = argmin_{\mathrm{w}}\, J(\boldsymbol{W});\ \ \boldsymbol{W} = \{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}\}$$



With gradient descent

$$W \leftarrow W - \alpha \frac{\partial J(W)}{\partial W}$$

Calculating gradients with the chain rule



$$\frac{\partial J(W)}{\partial W^{(1)}} = \frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W^{(1)}}$$

$$J(W) = \frac{1}{2}(\hat{y} - y)^2 = \frac{1}{2}(\hat{y}^2 + y^2 - 2\hat{y}y)$$

$$\hat{y} = g^{(2)}(W^{(1)} g^{(1)}(W^{(0)} X)) = g^{(2)}\left(W^{(1)} a^{(1)}\right) = g^{(2)}\left(z^{(2)}\right)$$

$$\frac{\partial J(W)}{\partial \hat{y}} = \hat{y} - y$$

$$\frac{\partial \hat{y}}{\partial W^{(1)}} = \frac{\partial \hat{y}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(1)}}$$

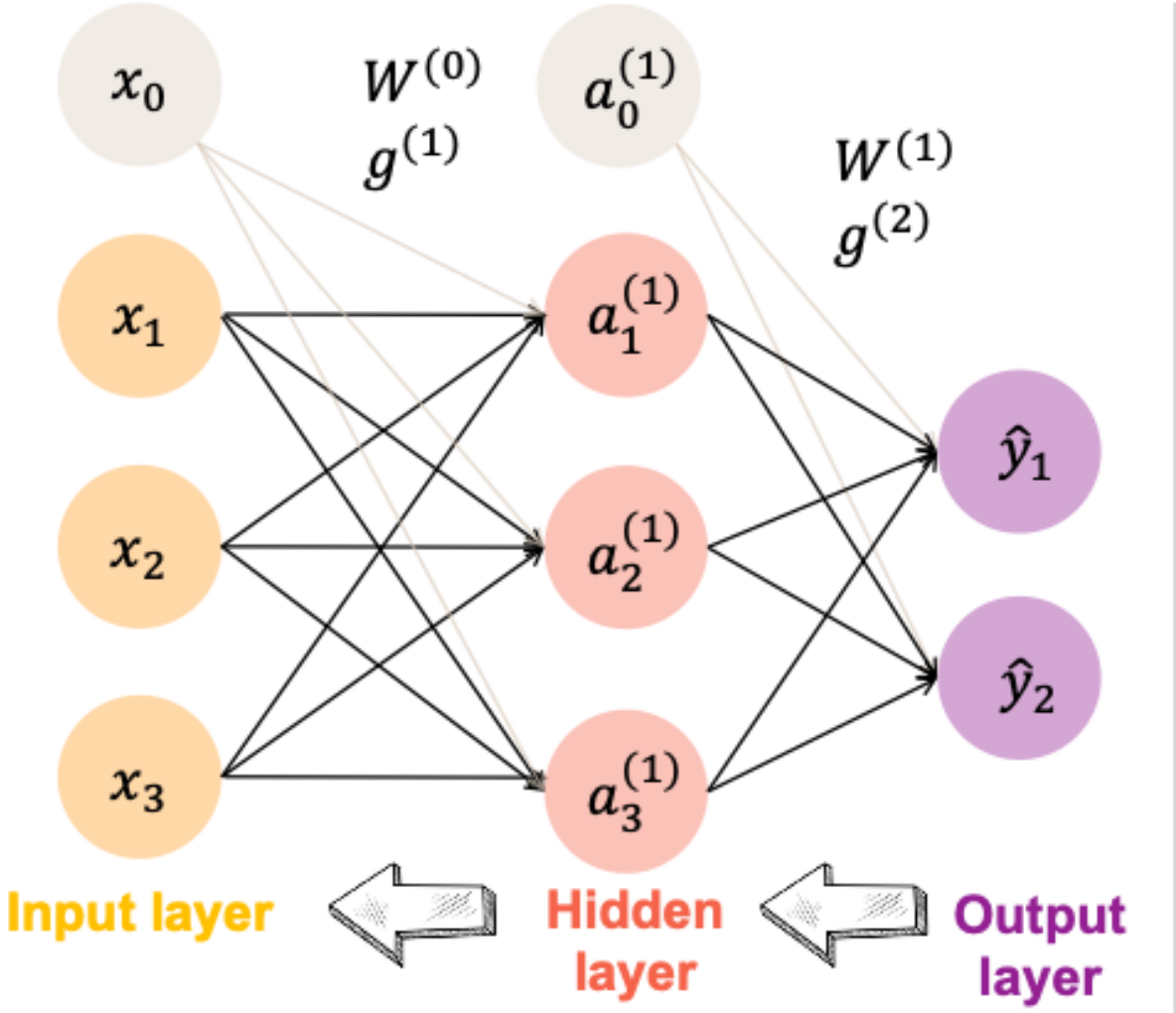
$$= g'^{(2)}\left(z^{(2)}\right) a^{(1)}$$

Backpropagation

Single layer neural network

We want to find the network weights that achieve the lowest loss

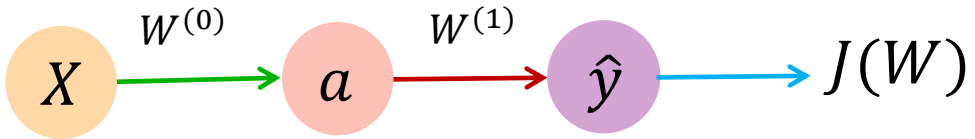
$$\boldsymbol{W}^* = argmin_{\mathrm{w}} J(\boldsymbol{W});\ \ \boldsymbol{W} = \{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}\}$$



With gradient descent

$$W \leftarrow W - \alpha \frac{\partial J(W)}{\partial W}$$

Calculating gradients with the chain rule



$$\frac{\partial J(W)}{\partial W^{(0)}} = \frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial W^{(0)}}$$

$$a^{(1)} = g^{(1)}(W^{(0)}X)$$

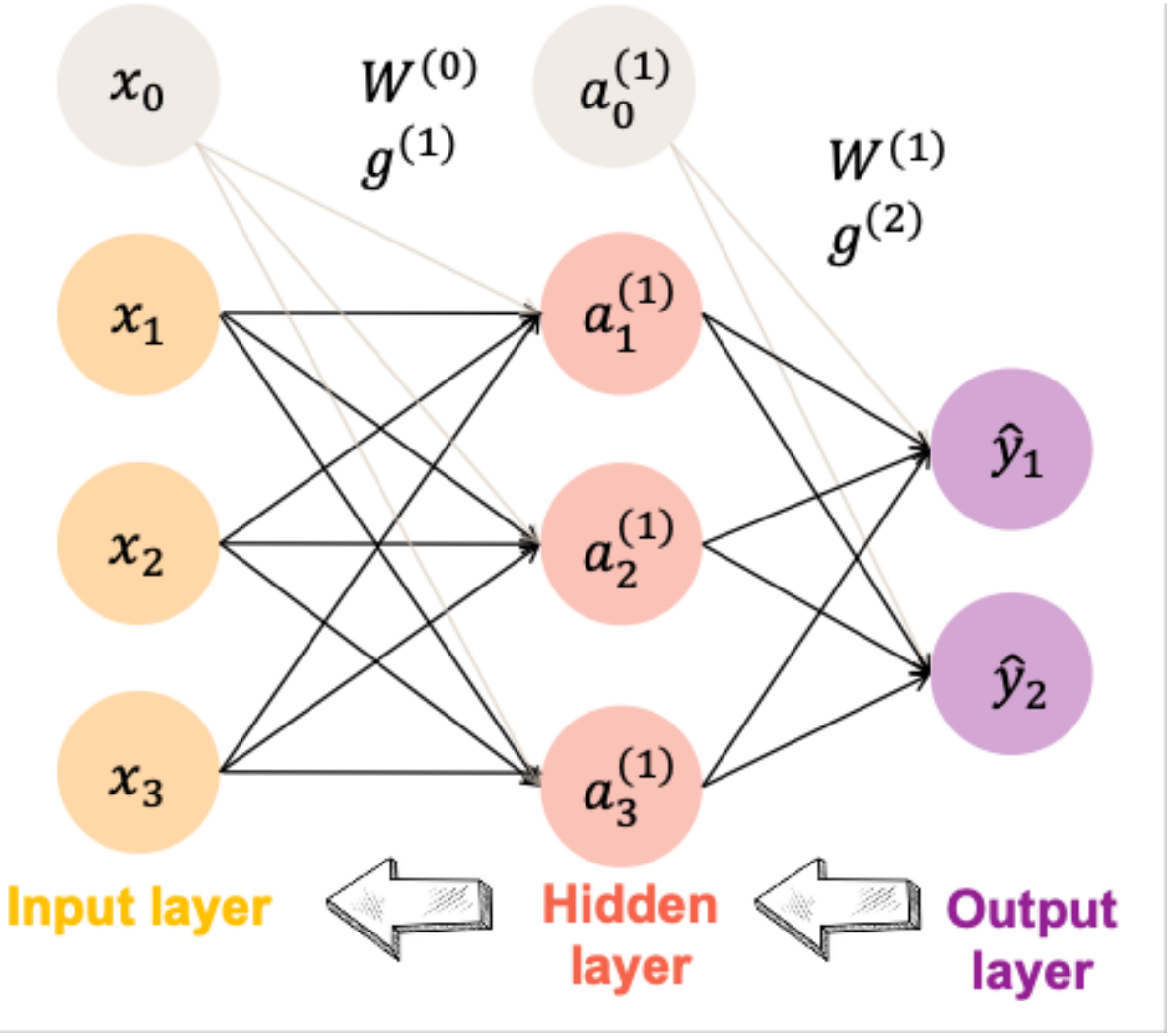
$\hat{y} - y$ $\quad g'^{(2)}(W^{(1)}a^{(1)})W^{(1)}$ $\quad g'^{(1)}(W^{(0)}X)X$

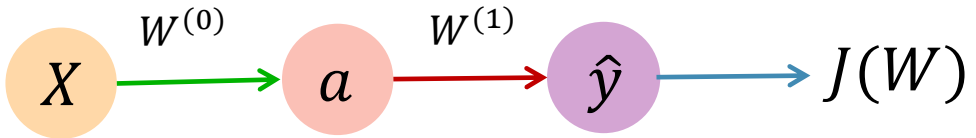
Backpropagation

Single layer neural network

We want to find the network weights that achieve the lowest loss

$$\boldsymbol{W}^* = argmin_{\mathrm{w}} J(\boldsymbol{W});\ \boldsymbol{W} = \{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}\}$$





$$\frac{\partial J(W)}{\partial W^{(1)}} = \frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W^{(1)}} \qquad \frac{\partial J(W)}{\partial W^{(0)}} = \frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial W^{(0)}}$$

- Efficient computation of gradients (by parts).
- **Automatic differentiation (autodiff)**:
  - Efficient $O(n)$ and numerically stable way of calculating gradients.
  - Decomposes the algorithm into primitive operations
  - Explicitly constructs a computation graph.
- **Autograd** = Python autodiff library, wraps numpy functions

Training neural networks: summary

- Neural networks are **compositions of functions**: $\hat{y} = g^{(L)}\left(W^{(L-1)} \dots g^{(1)}\left(W^{(0)}x\right)\right)$
- We minimize a loss that measures **prediction error**: $f(\hat{y} - y)$
- Because the network is a composition, **error signals propagate backwards** through layers

$$\frac{\partial J}{\partial \mathrm{W}^{(l)}} = \frac{\partial J}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial W^{(l)}}$$

Optimisation loop

- Forward pass computes $\hat{y}$ and J
- Backpropagation computes gradients
- Gradient descent (or other gradient-based method) uses them to update W

Neural networks are just function approximators trained by gradient-based optimisation

UNIVERSITY OF LIVERPOOL

THANK YOU FOR YOUR ATTENTION!

What questions do you have for me?

RESOURCES

- https://ml-cheatsheet.readthedocs.io/
- https://notesonai.com/
- https://buildmedia.readthedocs.org/media/pdf/ml-cheatsheet/latest/ml-cheatsheet.pdf
- http://introtodeeplearning.com/
- https://www.offconvex.org/
- https://developers.google.com/machine-learning

Dr. Andrea Santamaria Garcia
Lecturer at University of Liverpool
Cockcroft Institute

ansantam@liverpol.ac.uk
https://www.linkedin.com/in/ansantam/
https://github.com/ansantam
https://instagram.com/ansantam
https://www.liverpool.ac.uk/people/andrea-santamaria-garcia