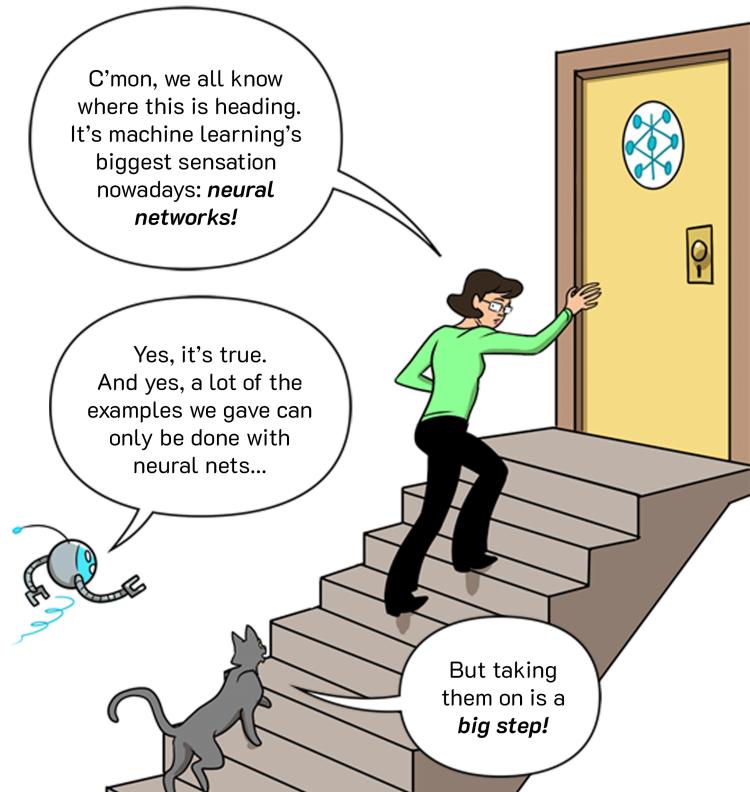


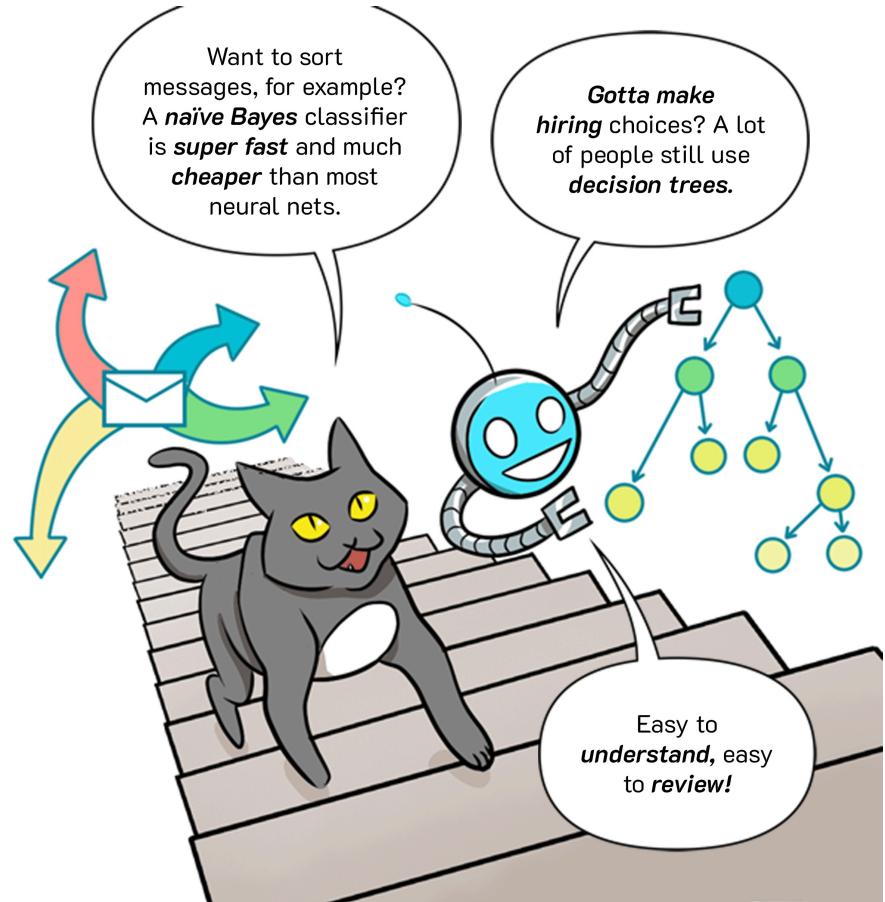
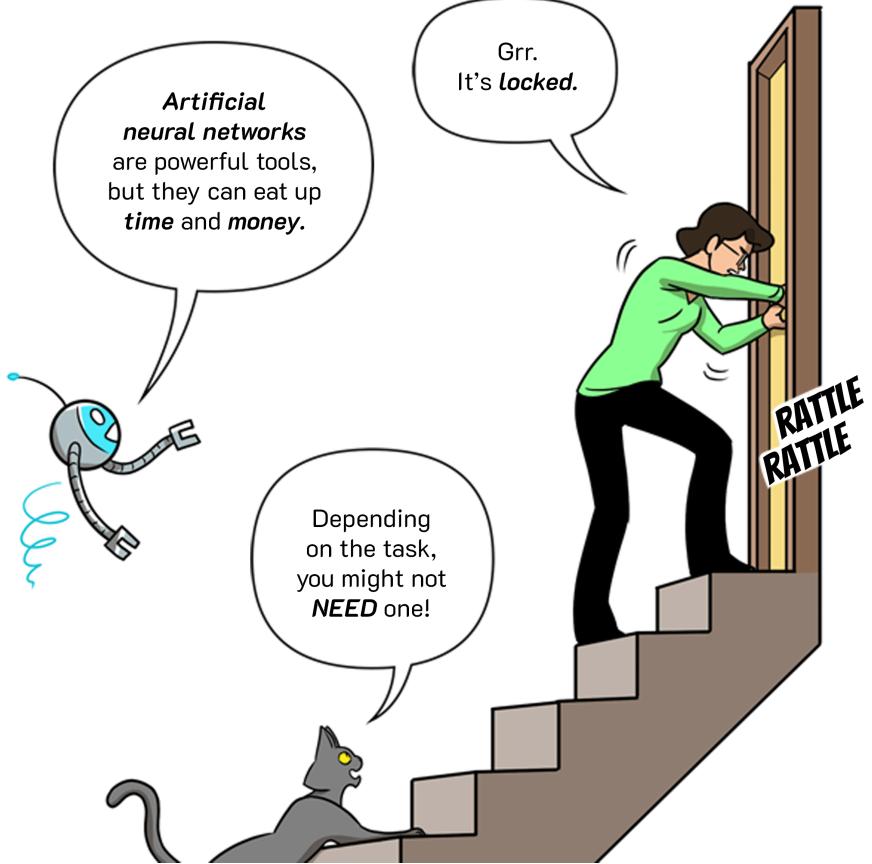
Introduction to Artificial Neural Networks

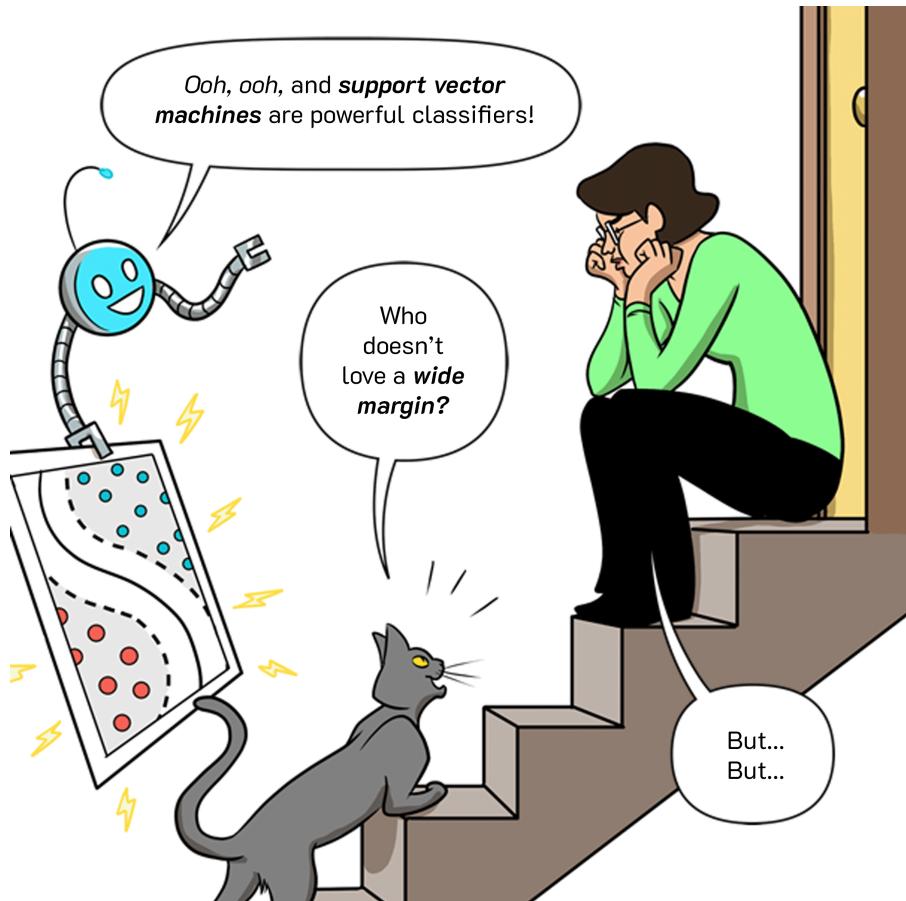
Part I

Dr. Andrea Santamaría García, Chenran Xu

Institute of Beam Physics and Technology (KIT)







AI4Accelerators Team at KIT

@ansantam



Dr. Andrea Santamaría García
andrea.santamaria@kit.edu

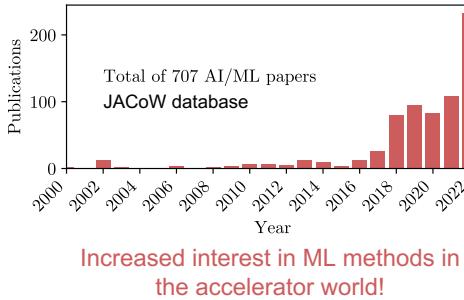
Team leader



Chenran Xu
chenran.xu@kit.edu
Doctoral student

We are accelerator
physicists researching
machine learning
applications for particle
accelerators

https://www.ibpt.kit.edu/team_AI.php



Increased interest in ML methods in
the accelerator world!

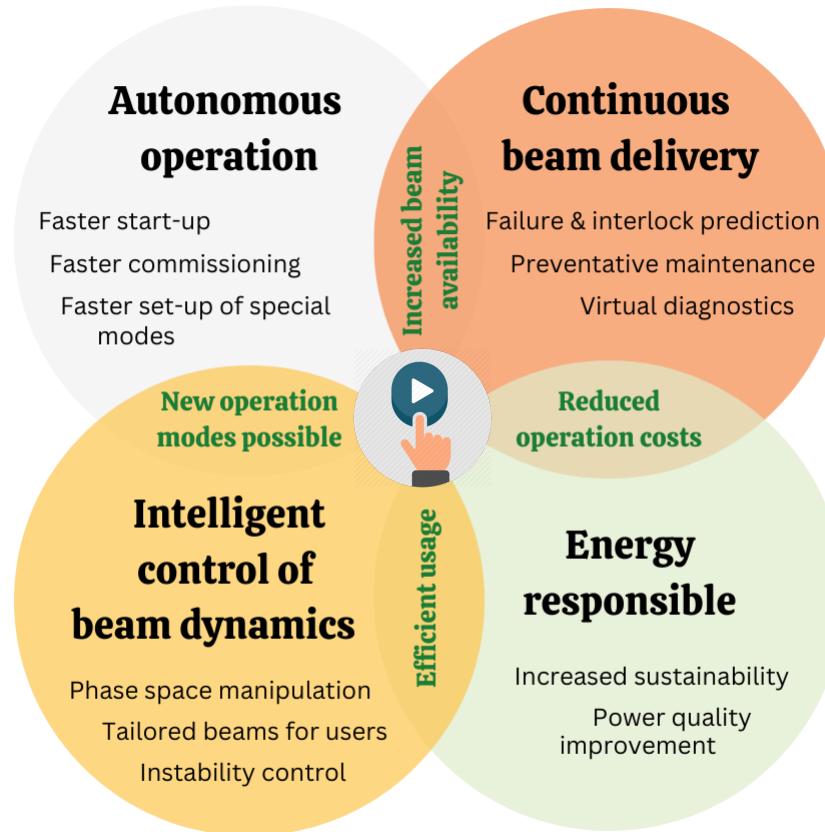
Task	Goal	Methods/Concepts	Examples ¹
Detection	Detect outliers and anomalies in accelerator signals for interlock prediction, data cleaning	<ul style="list-style-type: none">Anomaly detectionTime series forecastingClustering	<ul style="list-style-type: none">Collimator alignmentOptics correctionsSRF quench detection
Prediction	Predict the beam properties based on accelerator parameters	<ul style="list-style-type: none">Virtual diagnosticsSurrogate modelsActive learning	<ul style="list-style-type: none">Beam energy predictionAccelerator designPhase space reconstruction
Optimization	Achieve desired beam properties or states by tuning accelerator parameters	<ul style="list-style-type: none">Numerical optimizersBayesian optimizationGenetic algorithm	<ul style="list-style-type: none">Injection efficiencyRadiation intensity
Control	Control the state of the beam in real time in a dynamically changing environment	<ul style="list-style-type: none">Reinforcement learningBayesian optimizationExtremum Seeking	<ul style="list-style-type: none">Trajectory steeringInstability control

¹ non-exhaustive

A. Santamaría García et al, FLS23-TH3D3



A vision for future accelerators, driven by ML

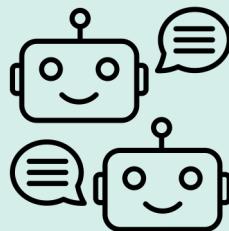


Introduction

ARTIFICIAL INTELLIGENCE (AI)

Computers mimic human behaviour

- First chatbots
- Robotics
- Expert systems
- Natural language processing
- Fuzzy logic
- Explainable AI



Narrow AI

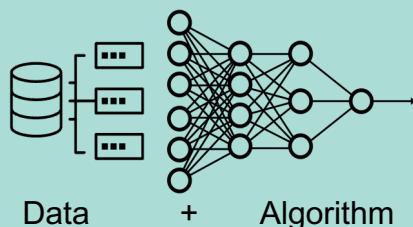
MACHINE LEARNING (ML)

Computers learn without being explicitly programmed to do so and improve with experience

Collection of **data-driven** methods / algorithms

Focused on **prediction / optimization / control** based on properties learned from data

Tries to **generalize** to unseen scenarios



DEEP LEARNING (DL)

Multi-layered neural networks perform certain tasks with high accuracy



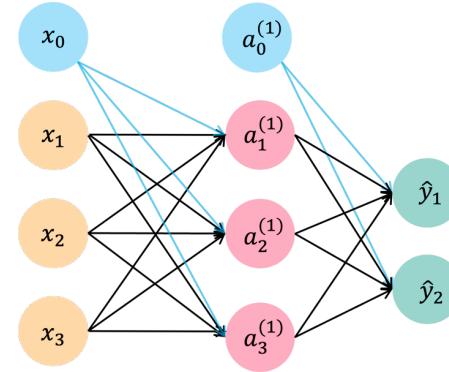
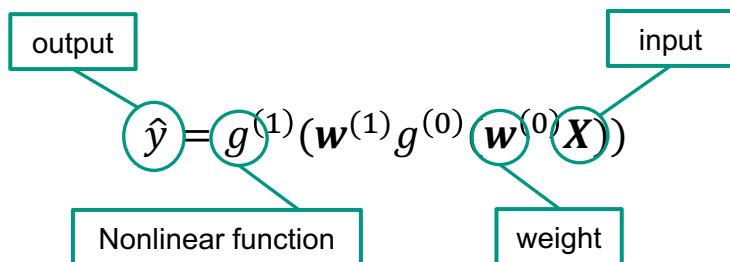
- Speech/handwriting recognition
- Language translation
- Recommendation engines
- Computer vision



What are neural networks, conceptually?

They are combinations of mathematical functions that implement more complicated functions

$$y = f(x) \quad \begin{matrix} \text{target} \\ \text{data} \end{matrix} \quad \text{the model of the data}$$



Input layer Hidden layer Output layer

The structure of the network, the functions it uses, and its multiplicative parameters define the spaces of multivariate functions that can be implemented by a neural network

NN = a nonlinear function of (usually) many variables that depends on (usually) many parameters

Why neural networks?

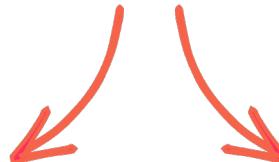
Universal Approximation Theorem

A feedforward network with a single hidden layer and an unbounded nonlinear activation function is sufficient to approximate, to an arbitrary precision, any continuous function

Is this too good to be true 🤔?

of course, there are some caveats

The number of neurons might be infeasibly large



The resulting model might not generalize

The approximated function does not extrapolate outside of the approximation region



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

Limitations of neural networks

NNs are excellent function approximators!

... when they have training data 😊

and a lot of it



- The universal approximation theorem does not tell you how to construct your network parameters.
- Neural networks are trained on data, but they might not converge on the correct parameters.

Machine learning \neq

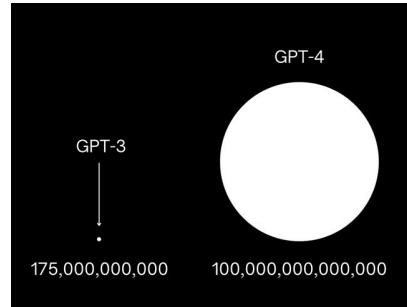
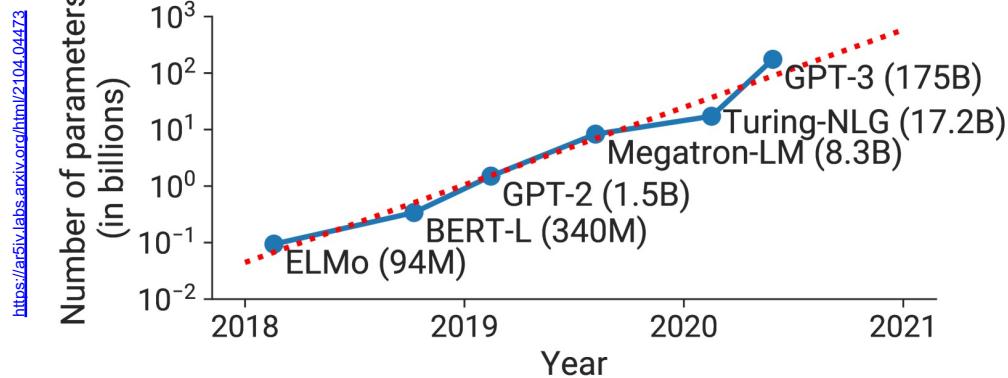


Although nowadays it starts looking like it!

[https://www.explainxkcd.com/wiki/index.php/1838: Machine Learning](https://www.explainxkcd.com/wiki/index.php/1838:Machine_Learning)

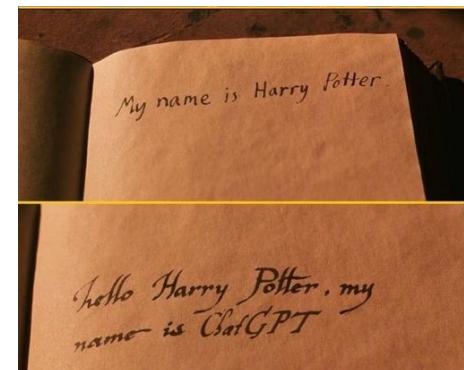
A data hungry, very large parameter example: LLMs

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



Not sorcery, just a lot of resources!

Still impressive



Limitations of neural networks

The No Free Lunch theorem

Our model is a simplification of reality



Every learning algorithm must possess a certain bias



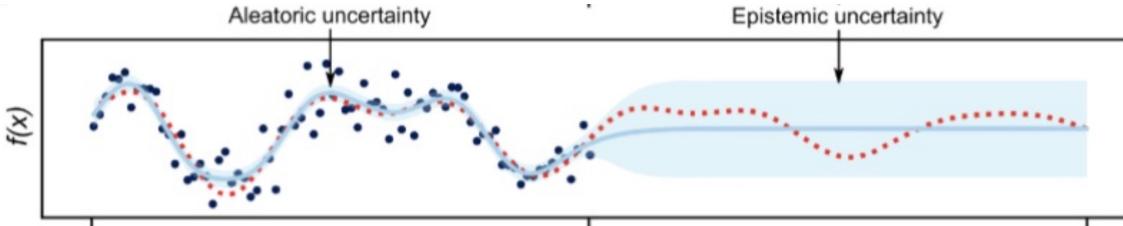
No single machine learning algorithm is universally the best-performing algorithm for all problems

+ Is there a scenario where linear regression outperforms neural networks?

- yes, when the linear regression assumptions are satisfied

Limitations of neural networks

Types of uncertainty



Aleatoric uncertainty

Data uncertainty

- Describes the confidence in the input data
- Is high when input data is noisy
- Cannot be reduced by adding more data
- Can be learned directly using neural networks
 - Probabilistic outputs, ensembling

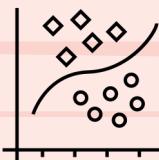
Epistemic uncertainty

Model uncertainty

- Describes the confidence of the prediction
- Is high when missing training data
- Can be reduced by adding more data
- Challenging to estimate
 - Bayesian NN, evidential deep learning

SUPERVISED LEARNING

Classification, prediction, forecasting
computer learns by example



- Spam detection
- Weather forecasting
- Housing prices prediction
- Stock market prediction

UNSUPERVISED LEARNING

Segmentation of data
computer learns without prior information about the data

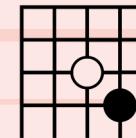


- Medical diagnosis
- Fraud (anomaly) detection
- Market segmentation
- Pattern recognition

MACHINE LEARNING

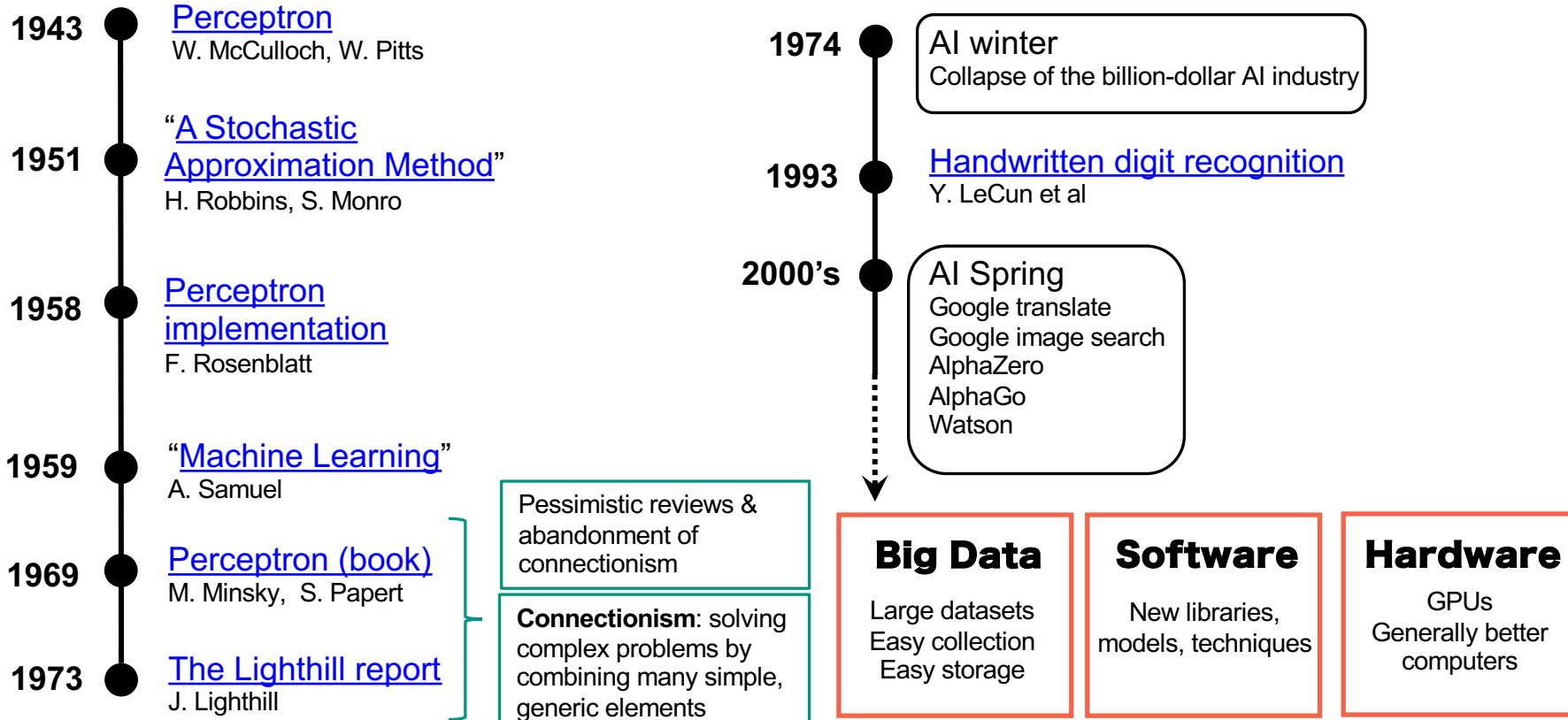
REINFORCEMENT LEARNING

Real-time decisions
computer learns through trial and error

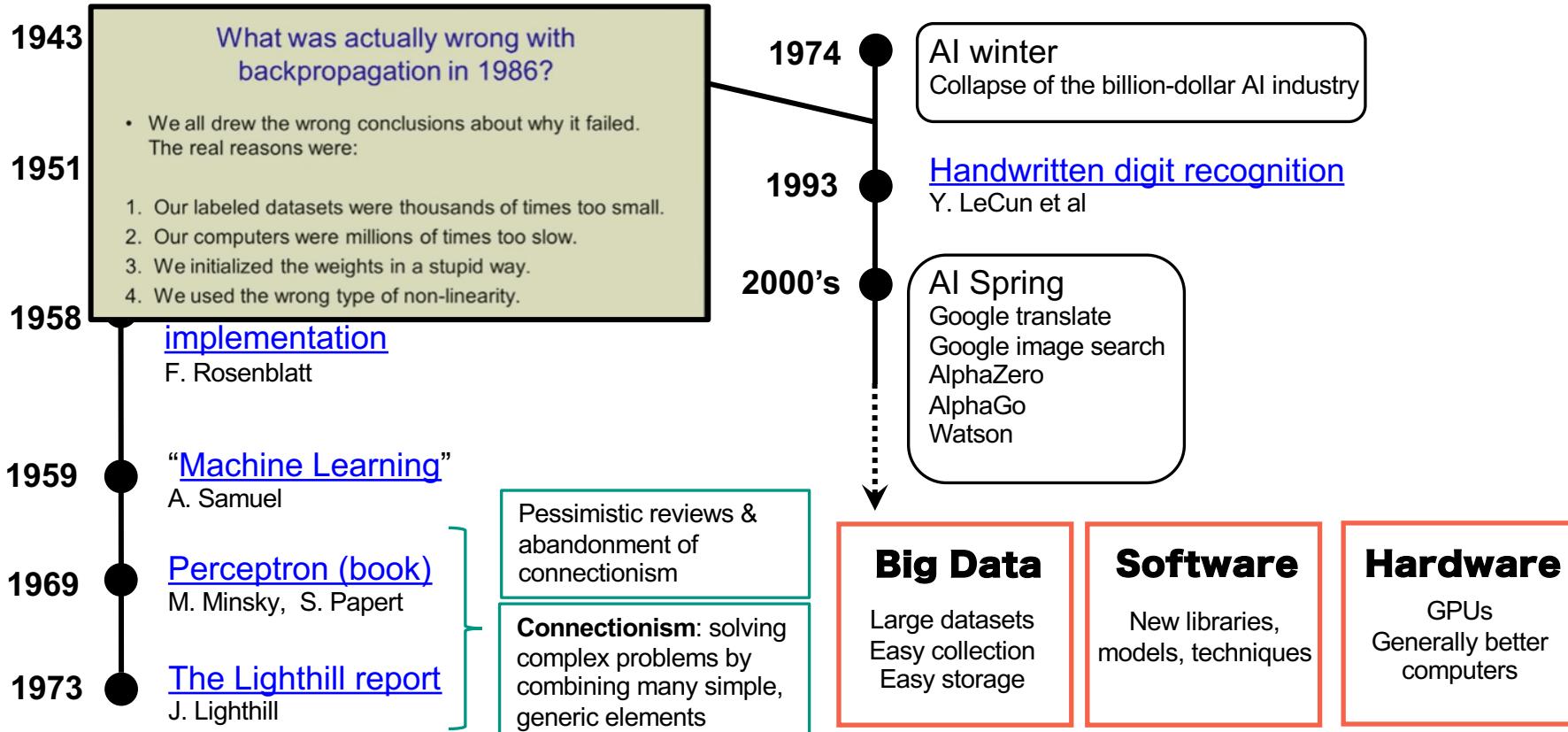


- Self-driving cars
- Make financial trades
- Gaming (AlphaGo)
- Robotics manipulation

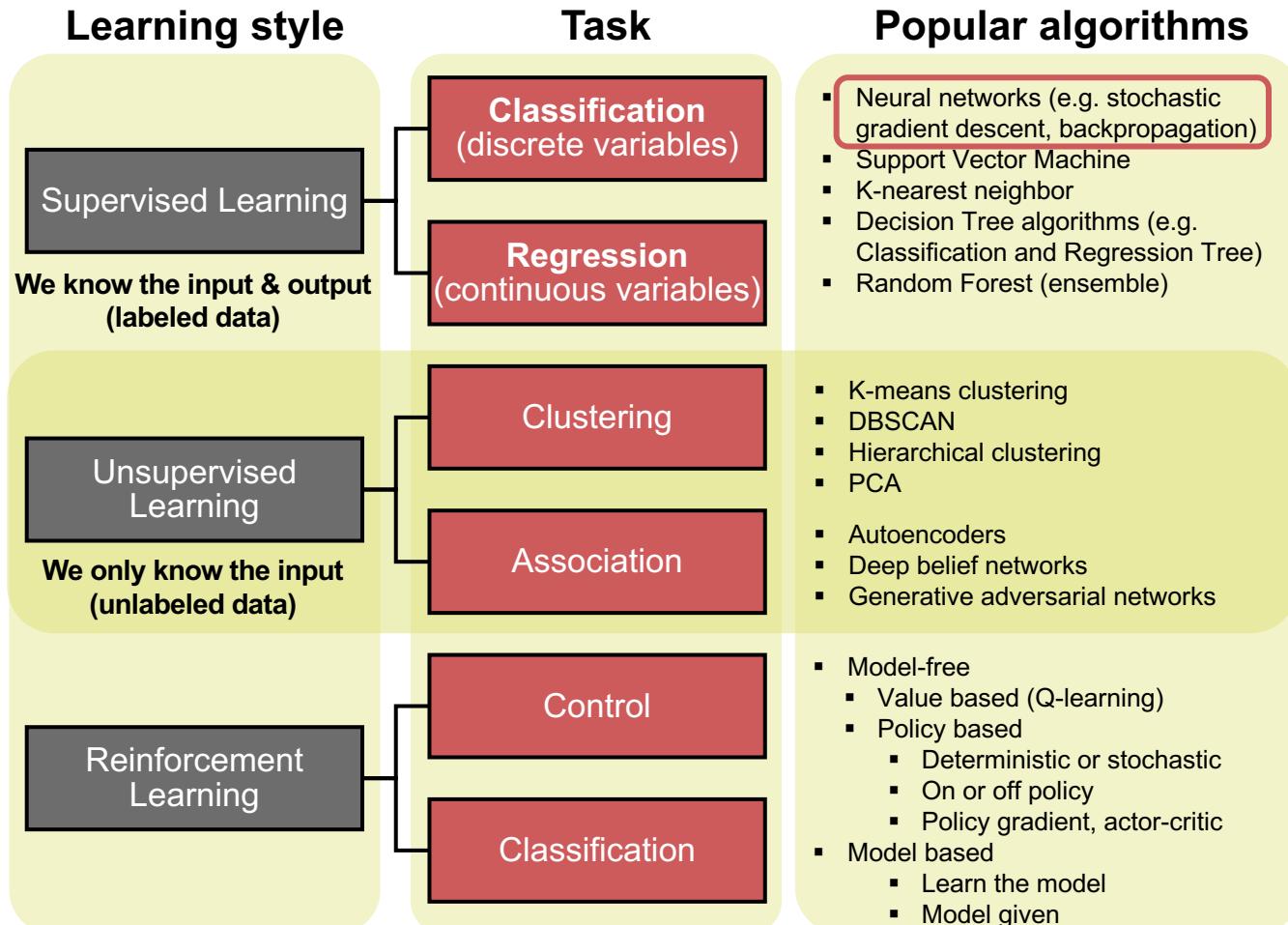
Why machine learning now?



Why machine learning now?



Machine Learning



Deep Learning Networks

- Convolutional Neural Networks
- Recurrent Neural Networks
- Long Short-Term Memory Networks
- Autoencoders
- Deep Boltzmann Machine
- Deep Belief Networks

Bayesian Algorithms

- Naive Bayes
- Gaussian Naive Bayes
- Bayesian Network
- Bayesian Belief Network
- Bayesian optimization

Regularization, dimensionality reduction, ensembling, evolutionary algorithms, computer vision, recommender systems, ...

Supervised learning **vs** unsupervised learning

Supervised learning

Labeled data: we know x and y

Learns the **mapping** between input and output

Focuses on **predicting** output for new data

- **Classification**
- **Regression**
- **Image tasks:** image classification, object detection, semantic segmentation.
- **Sequence tasks:** text classification, sentiment analysis, text translation, time-series forecasting, sequence generation.

Unsupervised learning

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

Models the **underlying structure** or distribution in the data to learn more about it

Focuses on **structure, distribution, and relationships** within the data

- **Clustering:** dataset divided into groups (similarity).
- **Dimensionality reduction:** selection of dimensions that best explain the variability in the data.
- **Association rule learning:** find relationships between variables in big databases.
- **Density estimation:** estimate the distribution of the data (anomaly detection, data generation).

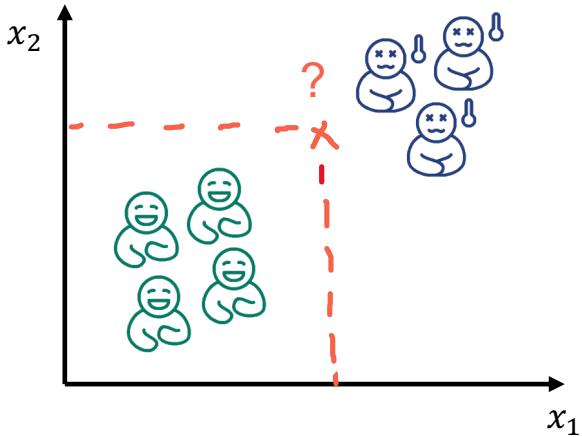
Supervised learning **vs** unsupervised learning

Supervised learning

Labeled data: we know x and y

Learns the **mapping** between input and output

Focuses on **predicting** output for new data

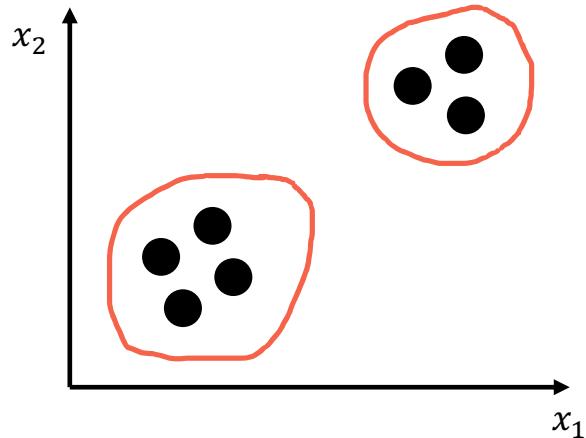


Unsupervised learning

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

Models the **underlying structure** or distribution in the data to learn more about it

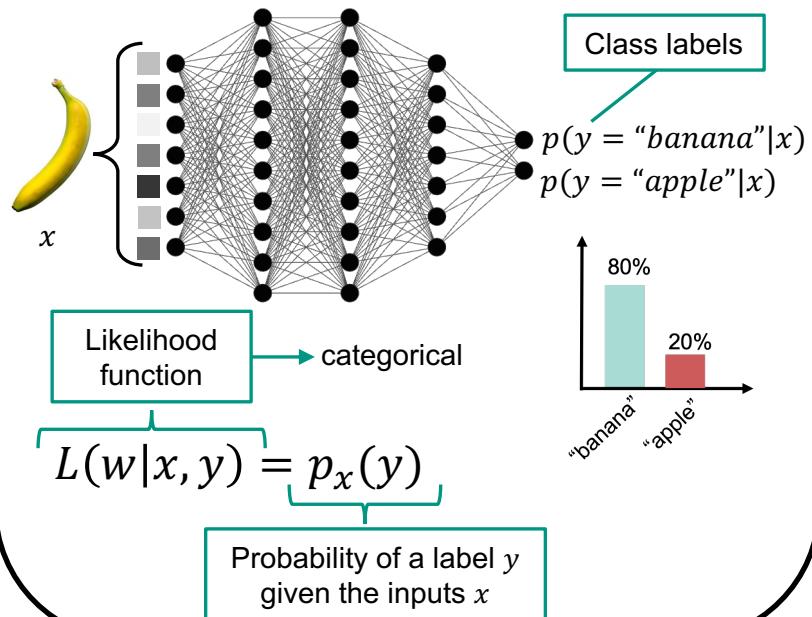
Focuses on **structure, distribution, and relationships** within the data



Classification vs regression in supervised learning

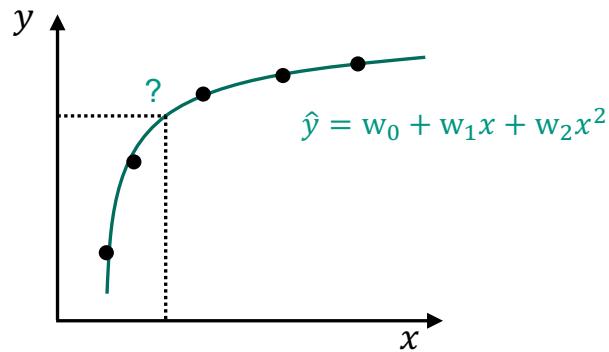
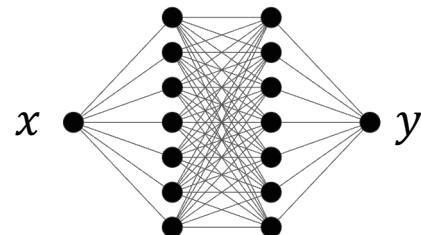
Classification task

y is discrete (can take certain values only)



Regression task

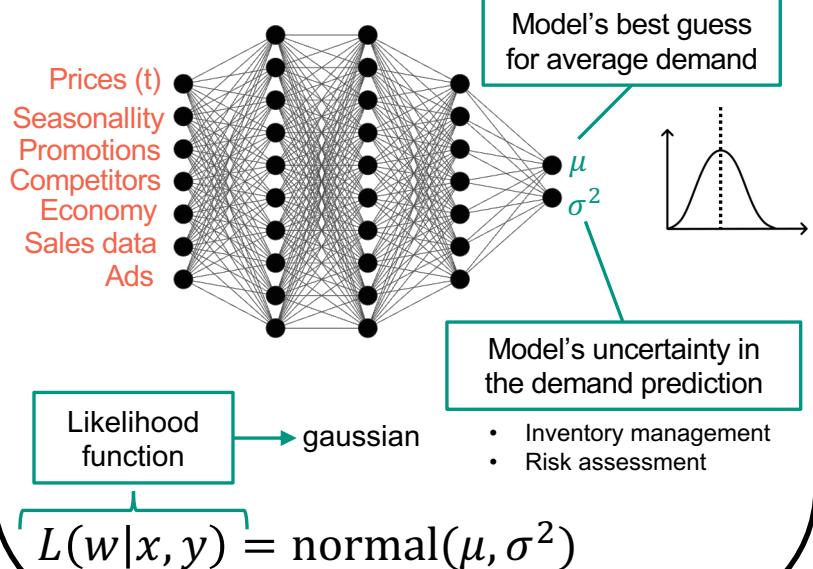
y is continuous (can take any value)



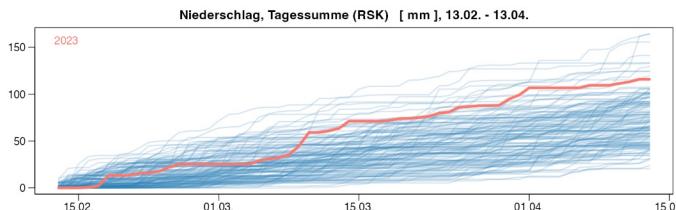
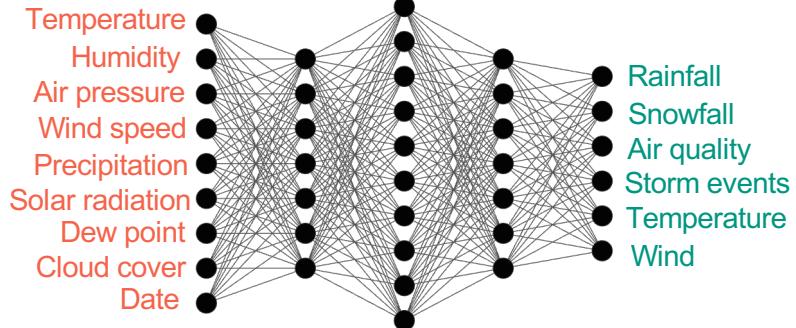
Examples of continuous targets



Product demand forecasting



Weather prediction



<https://github.com/brry/rdwd>

Performance metrics

Classification task

y is discrete

- **Accuracy:** proportion of correctly classified instances out of all instances.
- **Precision:** proportion of true positive results divided by the number of all positive results (including false positives).
- **Recall (Sensitivity):** proportion of true positive results divided by the number of positives that should have been identified.
- **F1 Score**
- **Confusion Matrix**
- **ROC Curve and AUC**

Regression task

y is continuous

- **Mean Absolute Error (MAE):** average of the absolute differences between the predicted values and the actual values.

$$MAE = \frac{\sum_{i=1}^n |\hat{y}_i - y_i|}{n}$$

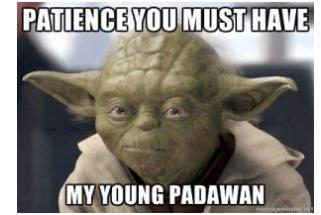
- **Mean Squared Error (MSE) or Root Mean Squared Error (RMSE):** average of the squared differences between the predicted values and the actual values.

$$MSE = \frac{\sum_{i=1}^n |\hat{y}_i - y_i|^2}{n} \quad RMSE = \sqrt{\frac{\sum_{i=1}^n |\hat{y}_i - y_i|^2}{n}}$$

- **R-squared:** A statistical measure of how close the data are to the fitted regression line.

But why
though?

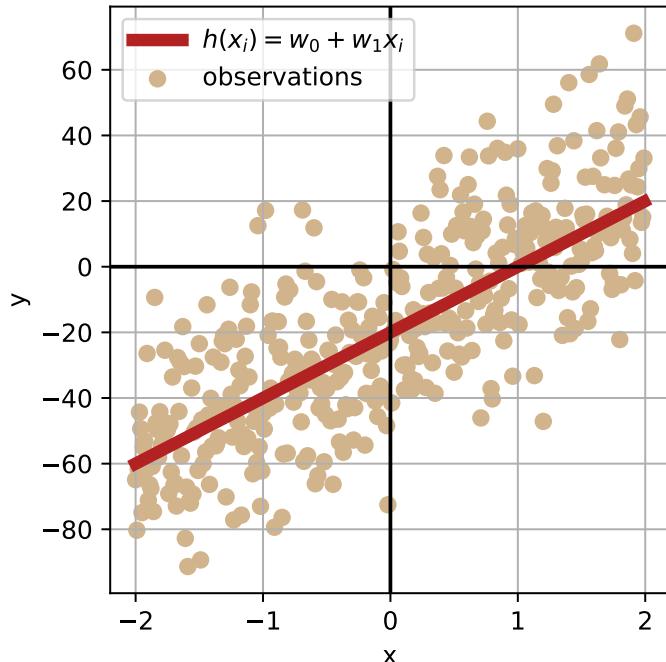
Let's review regression



Univariate linear regression (one prediction)

Simple (one feature)

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



Hypothesis function

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

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

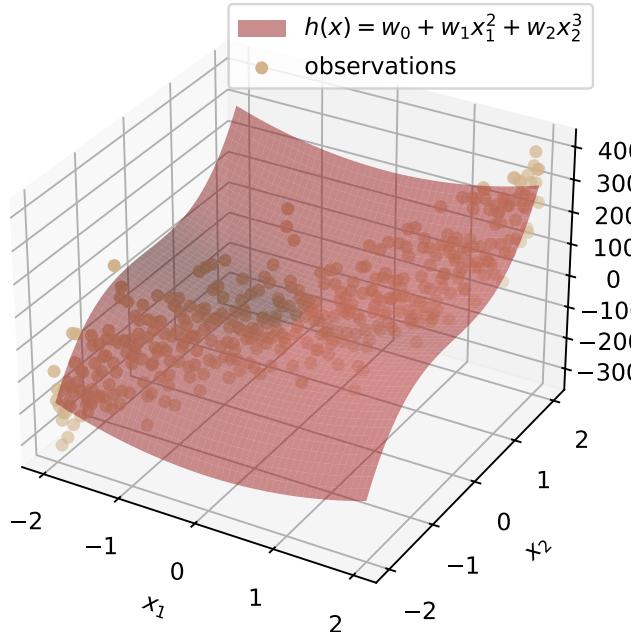
weights

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

Univariate regression (one prediction)

Multiple (several features)

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



Hypothesis function

$$h_w(x_1, x_2) = w_0 + w_1x_1^2 + w_2x_2^2$$

more generally

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

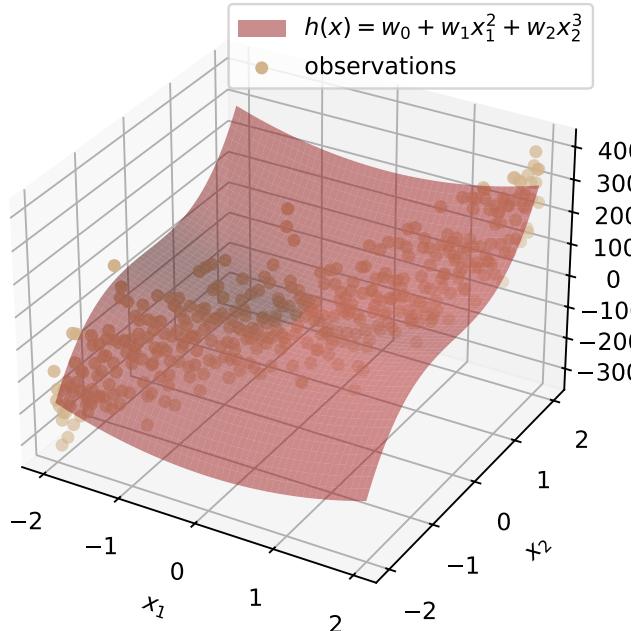
$i = 1, \dots, n$ data points
 $k = 1, \dots, p$ features
 $x_{0i} = 1$ pseudo-variable
 $\phi_k(x_i) = x_{ki}$ basis function

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)



Hypothesis function

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



more generally

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

basis function

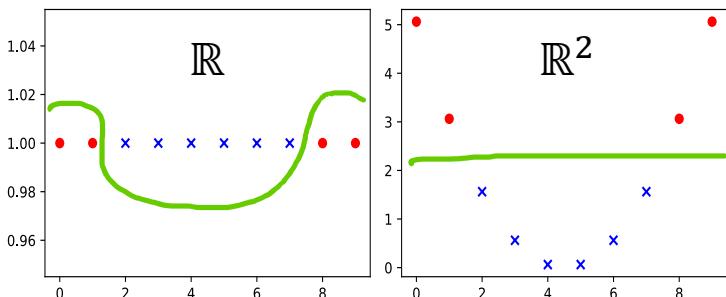
$\phi(x) = (x^0, x^1, x^2, \dots, x^p)$ —
Polynomial basis function

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

Brief interlude

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

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

Often:

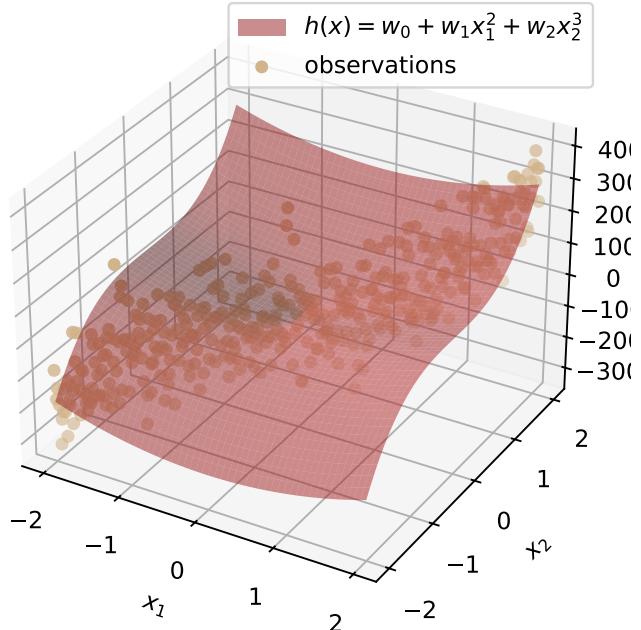
$$k(x, x') = \langle \phi(x), \phi(x') \rangle = \phi(x)^T \phi(x') \quad \phi: \mathcal{X} \rightarrow \mathcal{V}$$

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

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

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

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

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

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

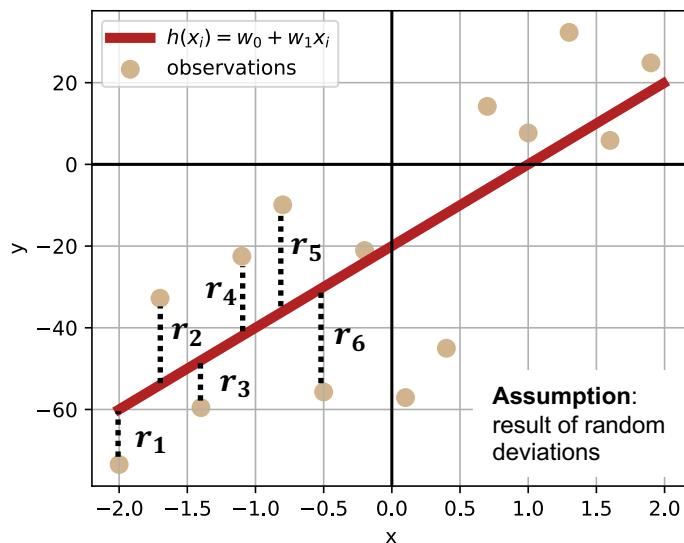
How to iteratively find the correct fit?

By minimizing the loss function

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

$$h_w(x_i) - y_i = r_i$$

residual
error



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

Also called cost function or error function

→ Number representing how well your algorithm models your dataset

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

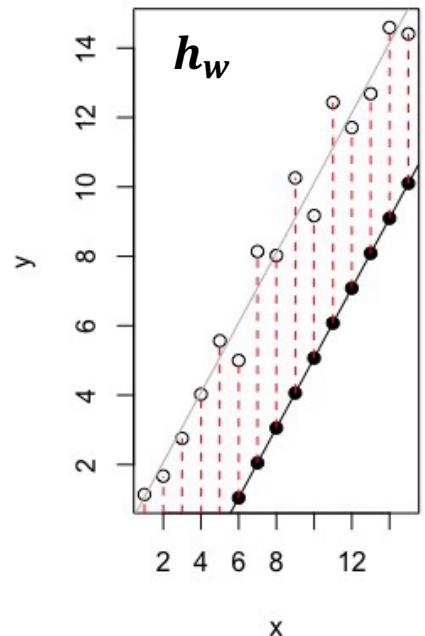
Negative log-likelihood function

Cross-entropy loss

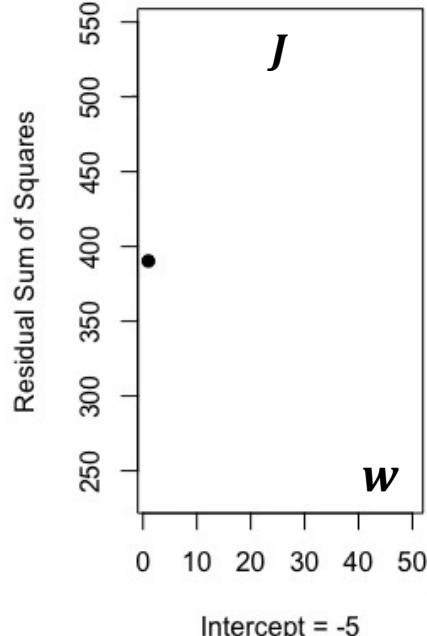
Example: least squares fitting

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 J

$$\sum_{i=1}^n \mathbf{r}_i^2 = \sum_{i=1}^n (h_w(x_i) - y_i)^2 = |\mathbf{Xw} - \mathbf{y}|^2 = J(\mathbf{w})$$

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

It depends on the weights (parameters) of your model



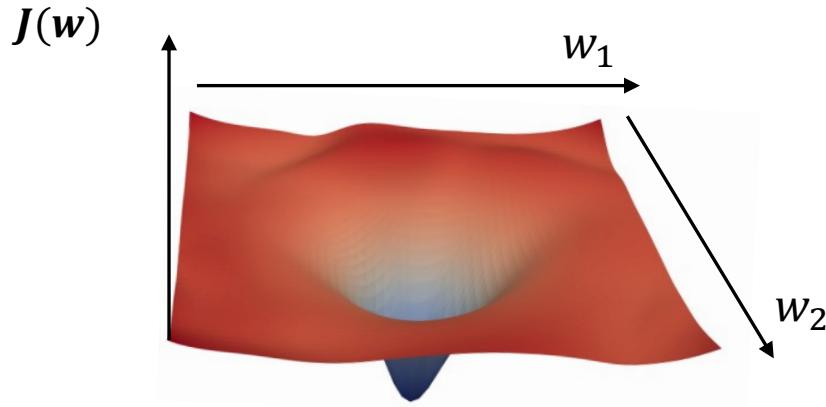
Use least squares when:

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

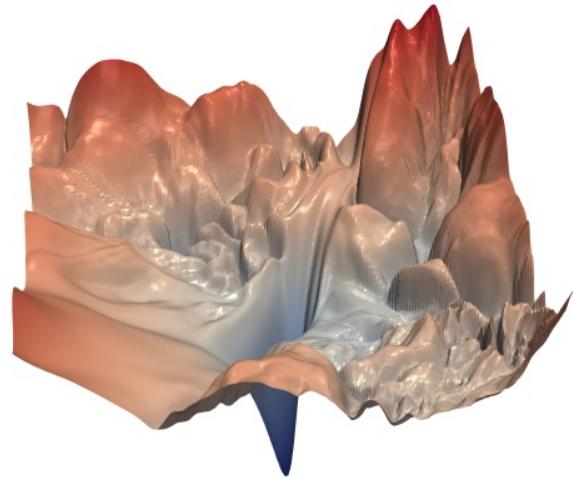
Why?

The “loss landscape”

This is a simple loss landscape for two weights:



*Nice and simply convex
Easy to minimize, convergence guaranteed*



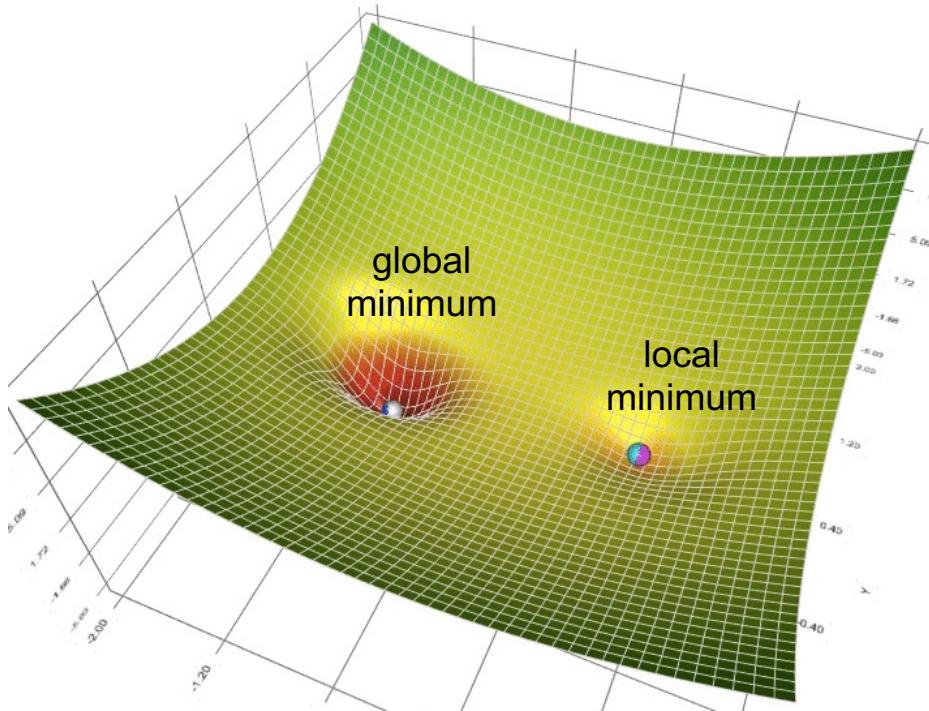
*This is more how it can be with deep NNs
(with no regularization)*

Remarks:

- We never see it like this, we only get one loss function value per algorithm iteration
- This is usually much higher dimensional

How do we find our way in the loss landscape?

With optimizers!



Animation from [Lili Jiang](#)

Gradient based methods (first order):

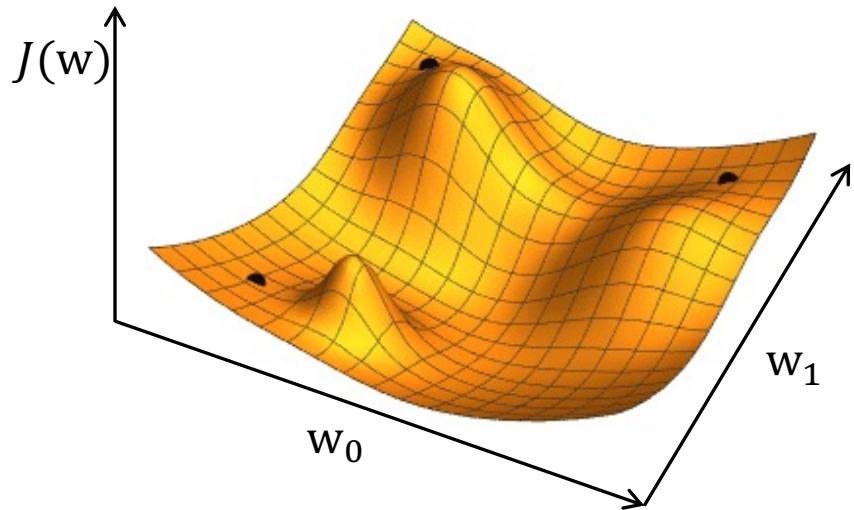
- Gradient descent
- momentum
- AdaGrad
- RMSProp
- Adam

} dynamic adjustment of algorithm parameters

More tomorrow!

Gradient descent

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



Update rule:

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

step size
learning rate

first order

take repeated steps in
the opposite direction
of the gradient

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

Loss function

$$h_{\mathbf{w}}(x) = w_0 + w_1 x$$

Hypothesis

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

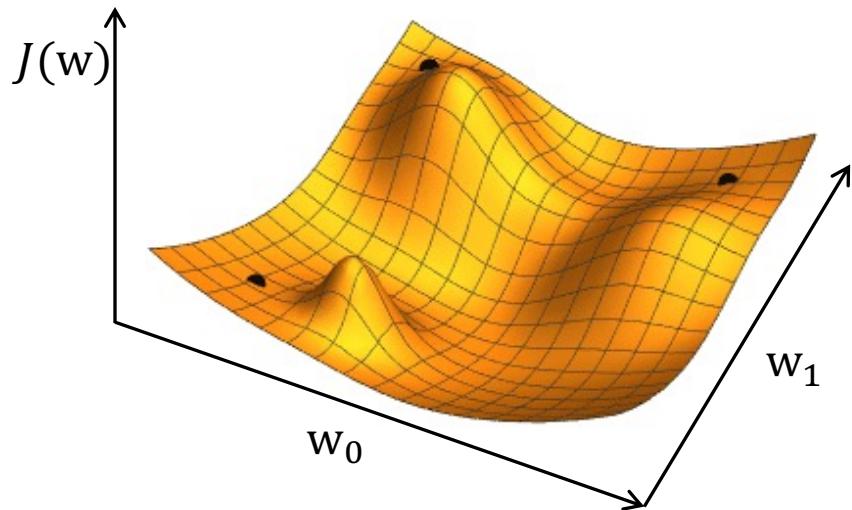
$$\frac{\partial J(w_0, w_1)}{\partial w_0} = \frac{1}{n} \sum_{i=1}^n (w_0 + w_1 x_i - y_i)$$

$$\frac{\partial J(w_0, w_1)}{\partial w_1} = \frac{1}{n} \sum_{i=1}^n x_i (w_0 + w_1 x_i - y_i)$$

Animation from [Wikimedia commons](#)

Gradient descent

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



Update rule:

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

step size
learning rate first order

take repeated steps in
the opposite direction
of the gradient

$$\frac{\partial J(\mathbf{w}_0, \mathbf{w}_1)}{\partial \mathbf{w}_0} = \frac{1}{n} \sum_{i=1}^n (\mathbf{w}_0 + \mathbf{w}_1 \mathbf{x}_i - \mathbf{y}_i)$$

$$\frac{\partial J(\mathbf{w}_0, \mathbf{w}_1)}{\partial \mathbf{w}_1} = \frac{1}{n} \sum_{i=1}^n \mathbf{x}_i (\mathbf{w}_0 + \mathbf{w}_1 \mathbf{x}_i - \mathbf{y}_i)$$

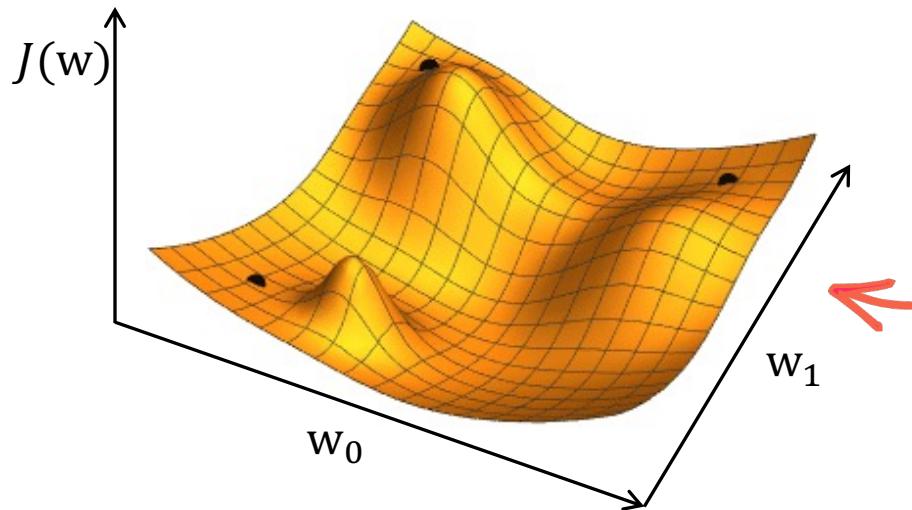
$$\left. \begin{aligned} \mathbf{w}_0 &:= \mathbf{w}_0 - \alpha \frac{\partial J(\mathbf{w}_0, \mathbf{w}_1)}{\partial \mathbf{w}_0} \\ \mathbf{w}_1 &:= \mathbf{w}_1 - \alpha \frac{\partial J(\mathbf{w}_0, \mathbf{w}_1)}{\partial \mathbf{w}_1} \end{aligned} \right\}$$

updated simultaneously

Animation from [Wikimedia commons](#)

Gradient descent

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

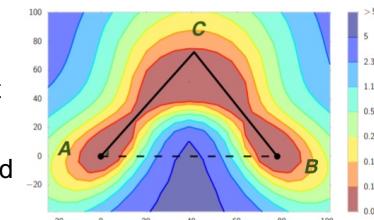


General remarks

- Works in any number of dimensions.
- Depending on the initial (w_0, 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

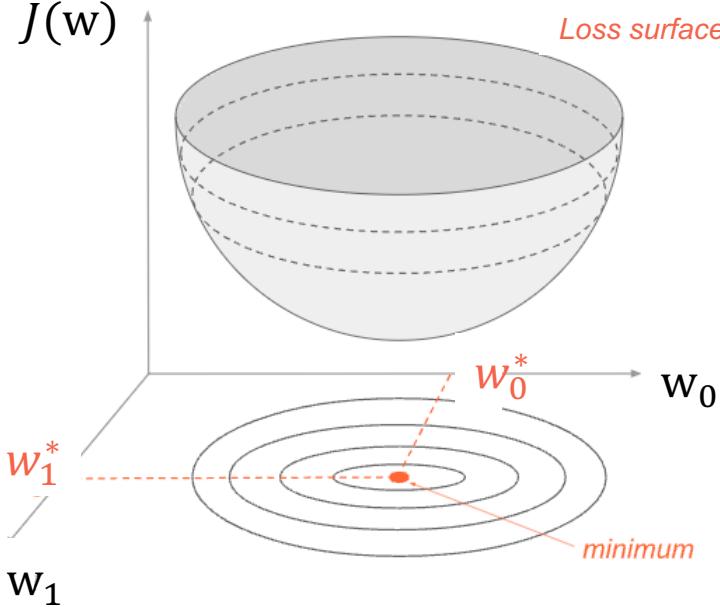


Garipov et al. 2018
Draxler et al. 2019

Animation from [Wikimedia commons](#)

Gradient descent

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



Matrix notation

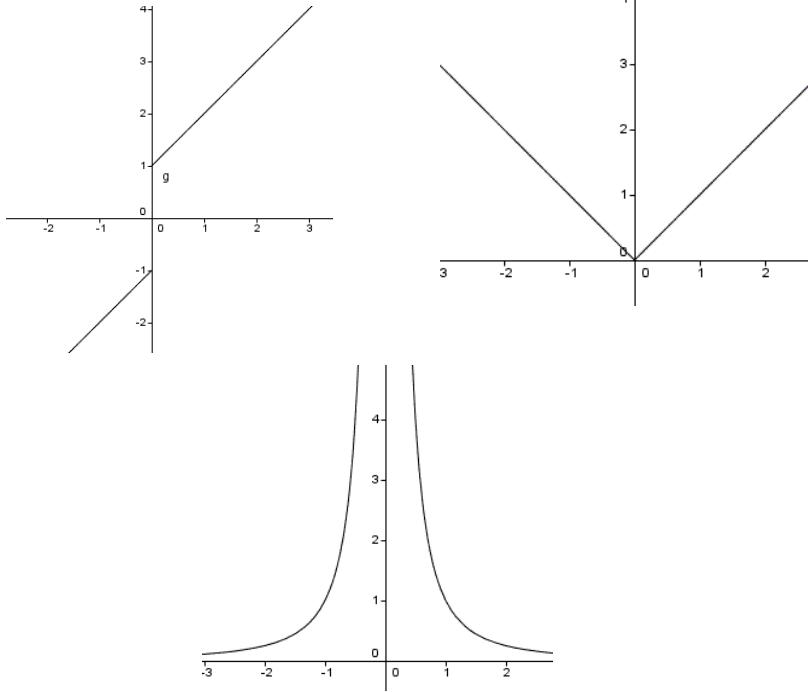
$$\begin{aligned} J(W) &= \frac{1}{2}(XW - Y)^T(XW - Y) \\ &= \frac{1}{2} \underbrace{(W^T X^T X W)}_{\text{quadratic}} - \underbrace{2Y^T X W}_{\text{linear}} + \underbrace{Y^T Y}_{\text{constant}} \end{aligned}$$

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



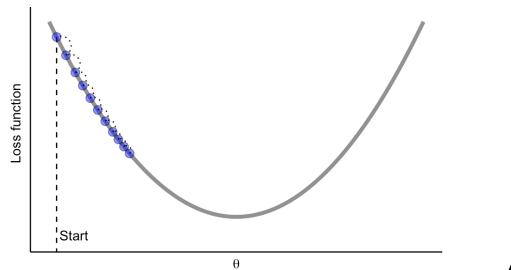
- Sensitivity to learning rate (next slide).
- Slow convergence in areas with small gradients (plateaus).
- Very dependent on weight initialization.
- Computationally intensive for large datasets.
- Only works when the function is continuously differentiable (gradient needs to exist).

Gradient descent: remarks

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

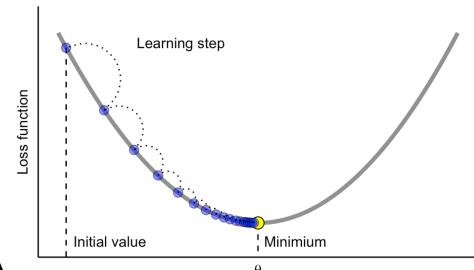
Too small

Converges slowly



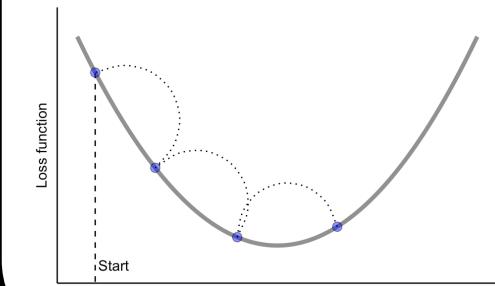
Just right

Converges quickly



Too large

Diverges



Gradient descent: remarks

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

Let's try it out!

<https://developers.google.com/machine-learning/crash-course/fitter/graph>

Try the following “start learning rate”:

- 0.02
- 0.1
- 0.4
- 1

Reducing Loss: Optimizing Learning Rate

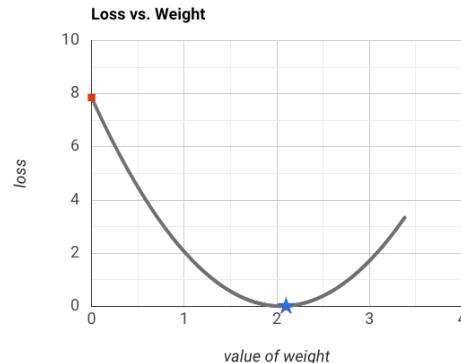
Estimated Time: 15 minutes

Experiment with different learning rates and see how they affect the number of steps required to reach the loss curve. Try the exercises below the graph.

Set learning rate:

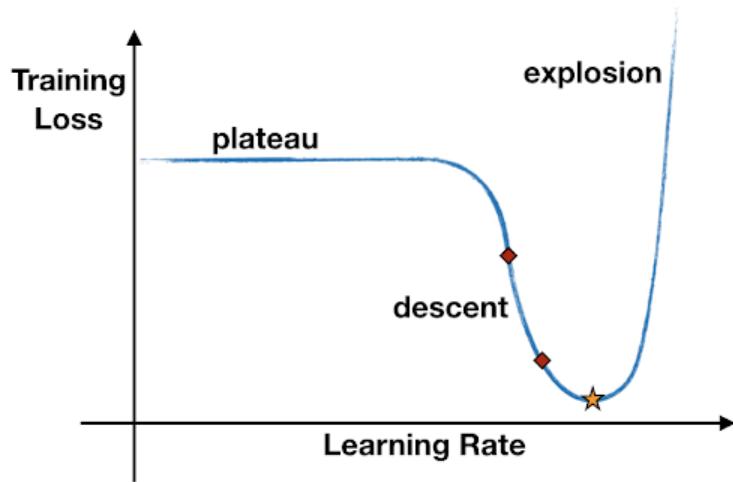
Execute single step: 0

Reset the graph:



Gradient descent: remarks

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



Modern algorithms have adaptive learning rates

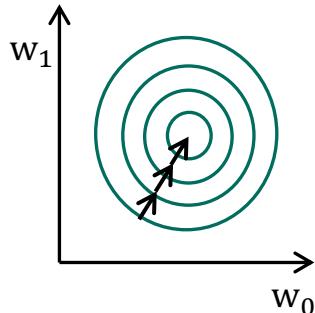
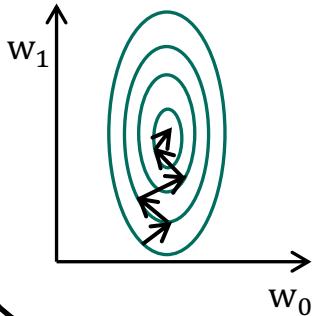
- The optimizer is working correctly if the loss decreases after every iteration.
- The recommended minimum learning rate is the value where the loss decreases the fastest
- Usual to **declare convergence tests**. (e.g., declare convergence when $J(\theta)$ decreases by less than 10^{-3} in one iteration)

Gradient descent: remarks

Feature scaling

- Optimizers can converge faster if the features are on a similar scale
- Becomes very important in polynomial regression:

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



Normalization

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

To rescale between [a, b]:

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

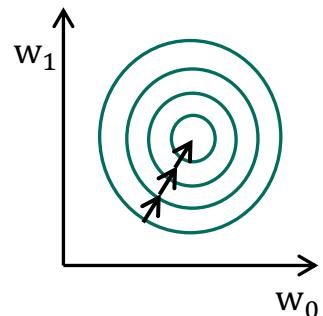
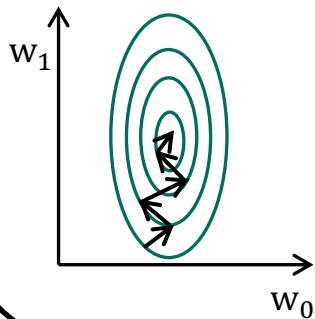
- Geometrically: will scale the n-dimensional data into an n-dimensional hypercube.
- Useful when there are no outliers and distribution is uniform.

Gradient descent: remarks

Feature scaling

- Optimizers can converge faster if the features are on a similar scale
- Becomes very important in polynomial regression:

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



Standardization

$$\mu = 0, \sigma^2 = 1$$

$$x' = \frac{x - \mu}{\sigma}$$

- Geometrically: translates the mean of the data to the origin and scales its spread.
- Useful when the data follows a Gaussian distribution.
- Deals well with outliers.

**But when are we going
to learn about neural
networks??**



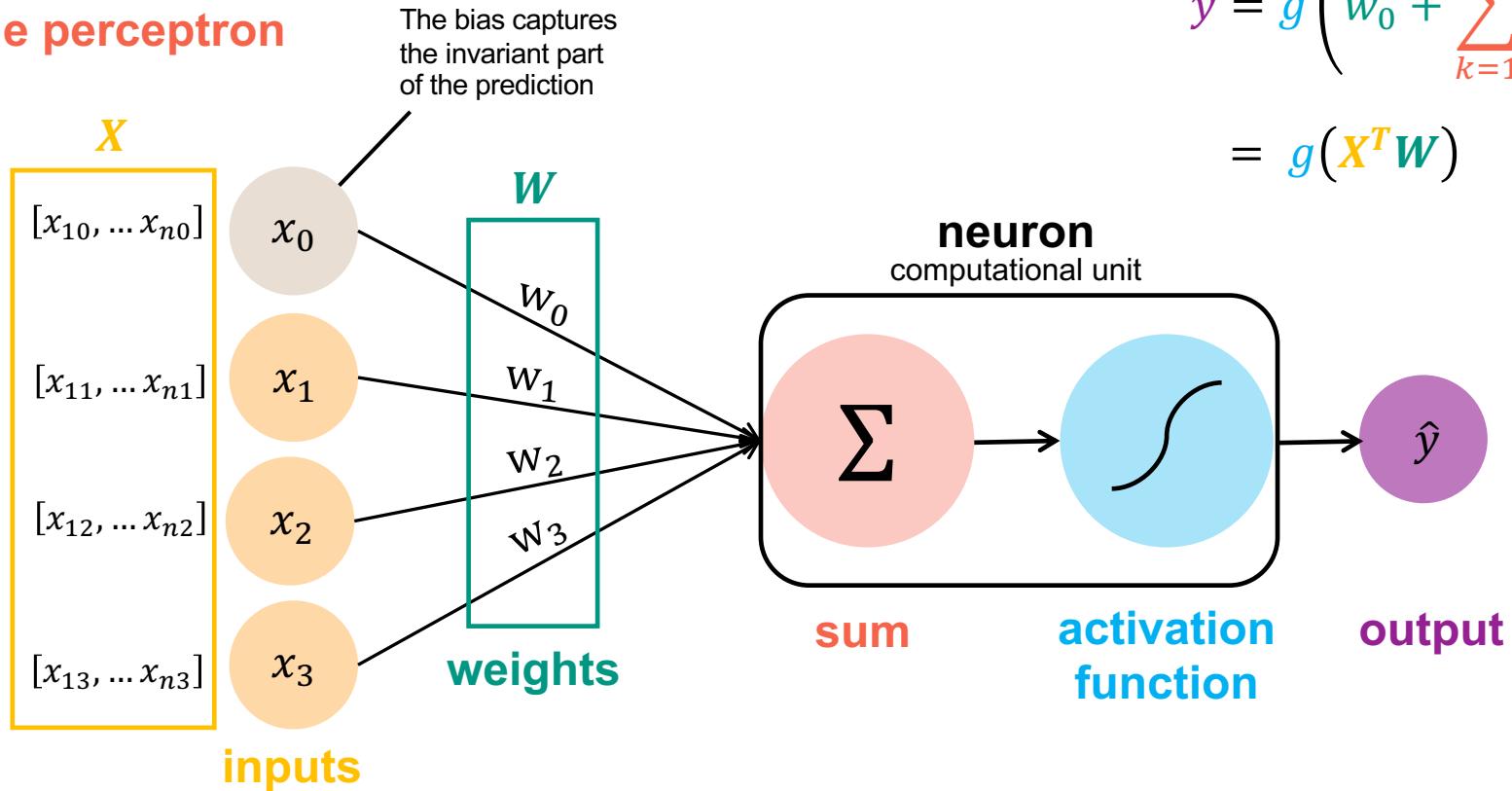


**Linear algebra,
multivariate calculus,
statistics**



Neural networks

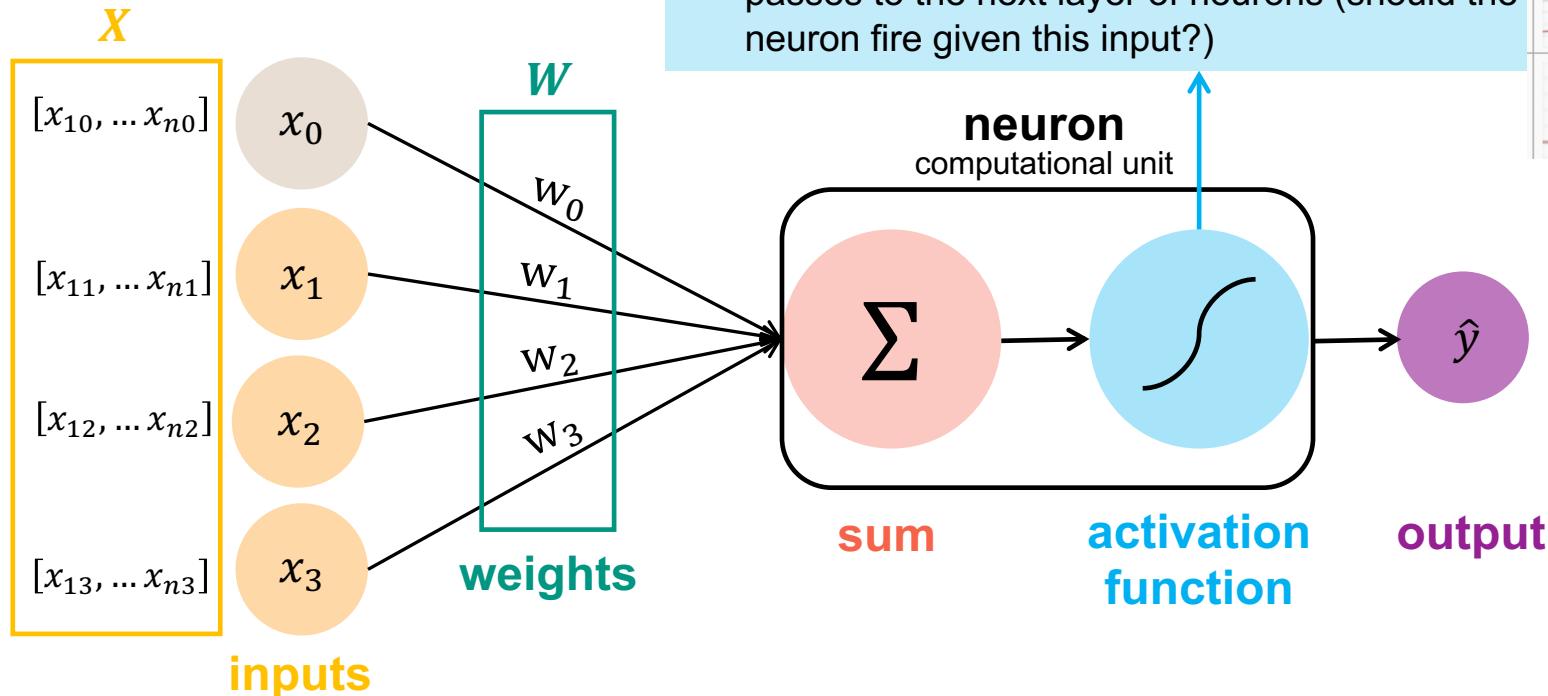
The perceptron



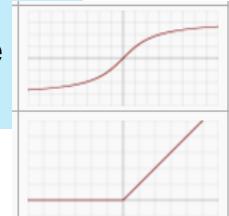
$$\begin{aligned}\hat{y} &= g\left(w_0 + \sum_{k=1}^p x_i w_i\right) \\ &= g(X^T W)\end{aligned}$$

Neural networks

The perceptron

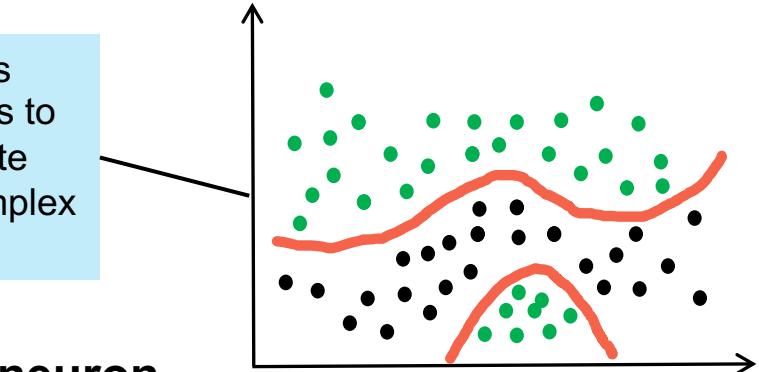
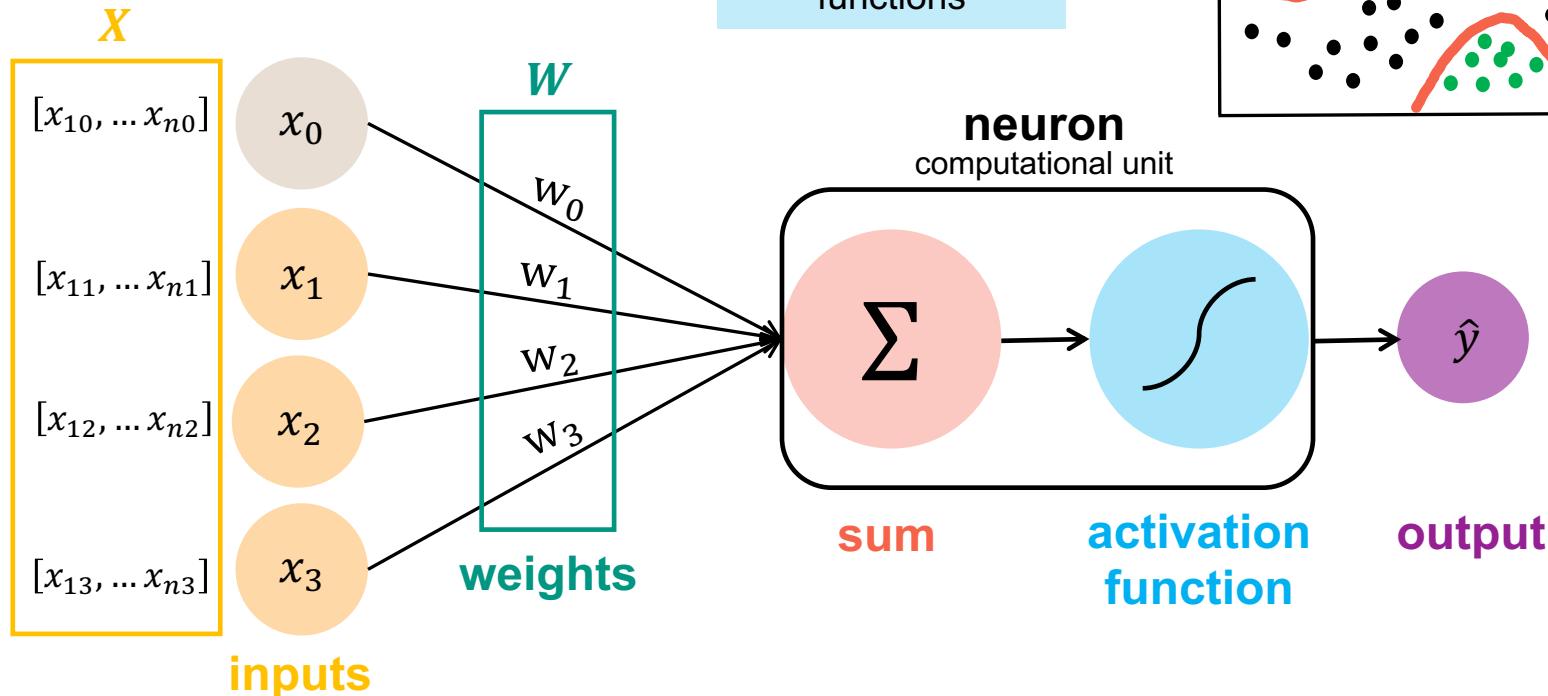


- What makes NNs different from linear regression.
- Defines the output of the neuron given an input.
- “**Mathematical gate**” that filters which data passes to the next layer of neurons (should the neuron fire given this input?)



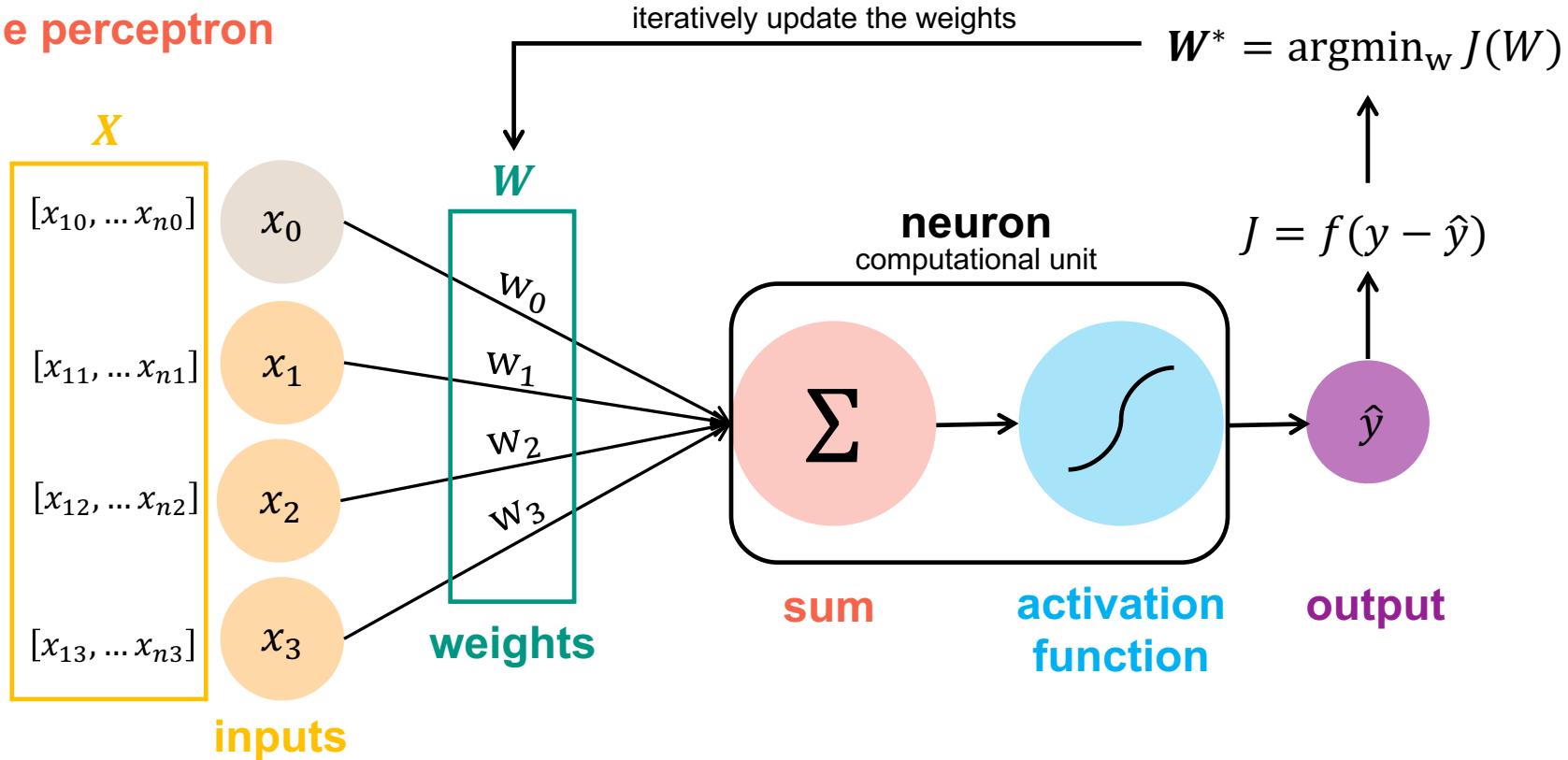
Neural networks

The perceptron



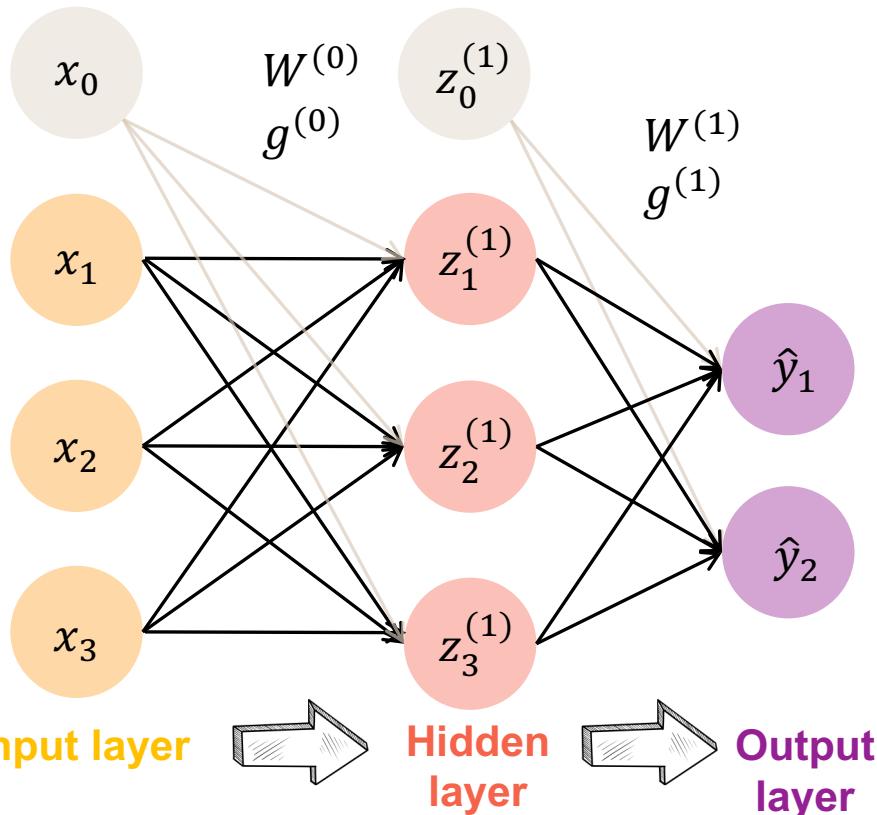
Neural networks

The perceptron



Forward propagation

Single layer neural network



New notation:

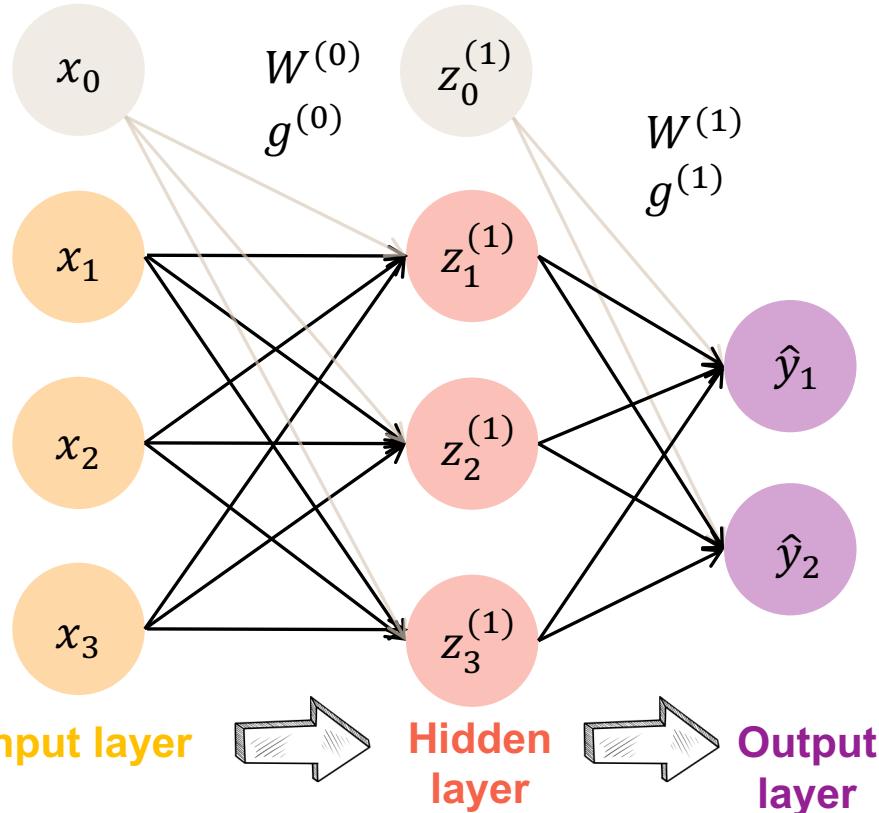
- **Superscript** = layer number
- **Subscript** = neuron number
- z = activation vector or unit
- g = activation function

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

multilayer network evaluates compositions of functions computed at individual neurons

Forward propagation

Single layer neural network



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

Step 1

$$\mathbf{W}^{(0)} = \left[\begin{array}{cccc} w_{10}^{(0)} & w_{11}^{(0)} & w_{12}^{(0)} & w_{13}^{(0)} \\ w_{20}^{(0)} & w_{21}^{(0)} & w_{22}^{(0)} & w_{23}^{(0)} \\ w_{30}^{(0)} & w_{31}^{(0)} & w_{32}^{(0)} & w_{33}^{(0)} \end{array} \right] \quad \begin{matrix} \text{neurons in} \\ \text{hidden layer} \end{matrix}$$

neurons in input layer + bias

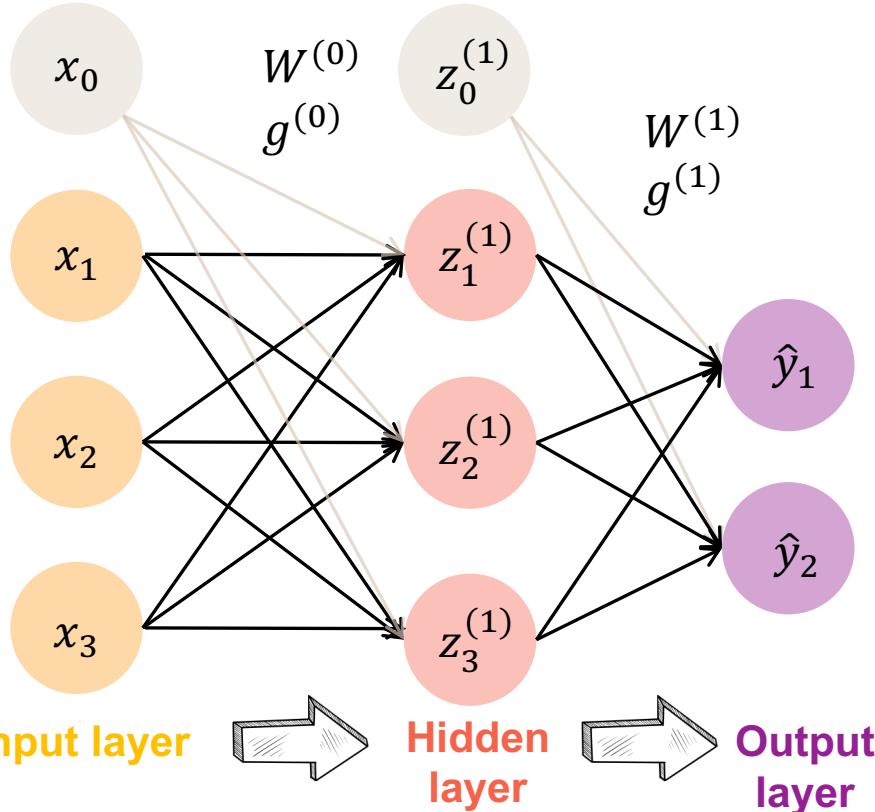
$$\mathbf{X} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

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

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

Forward propagation

Single layer neural network



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

Step 2

$$\mathbf{w}^{(1)} = \begin{bmatrix} w_{10}^{(1)} & w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{20}^{(1)} & w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \end{bmatrix}$$

neurons in hidden layer + bias

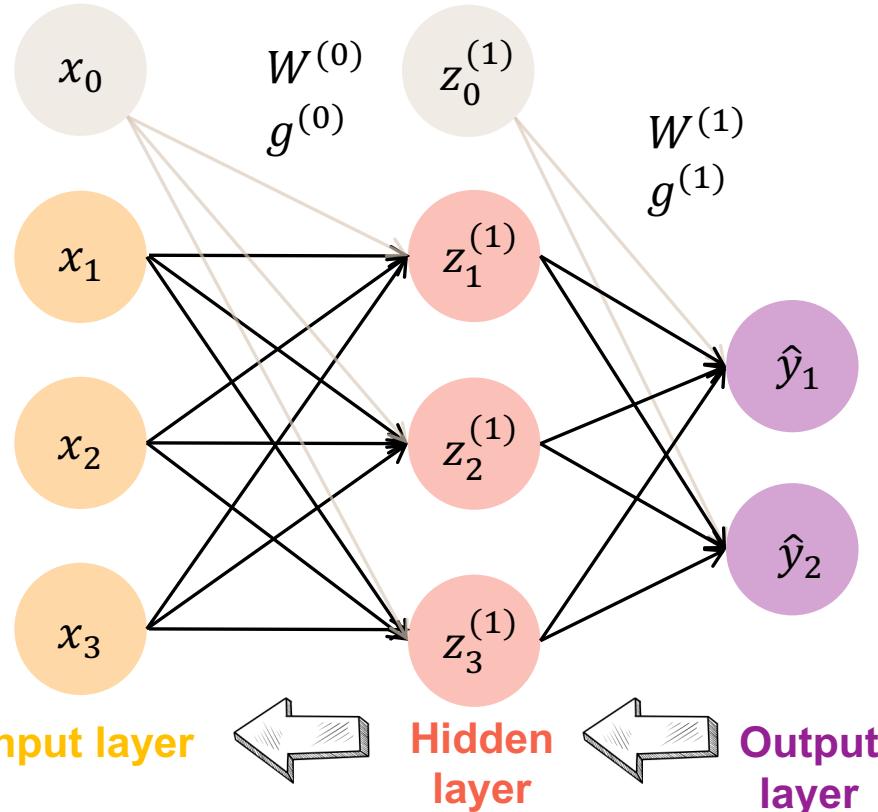
$$\mathbf{z}^{(1)} = \begin{bmatrix} z_0^{(1)} \\ z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \end{bmatrix}$$

$$\mathbf{z}^{(2)} = \hat{y} = \begin{bmatrix} g^{(1)}(w_{10}^{(1)} z_0^{(1)} + w_{11}^{(1)} z_1^{(1)} + w_{12}^{(1)} z_2^{(1)} + w_{13}^{(1)} z_3^{(1)}) \\ g^{(1)}(w_{20}^{(1)} z_0^{(1)} + w_{21}^{(1)} z_1^{(1)} + w_{22}^{(1)} z_2^{(1)} + w_{23}^{(1)} z_3^{(1)}) \end{bmatrix}$$

✓ Forward propagation completed

Backpropagation

Single layer neural network



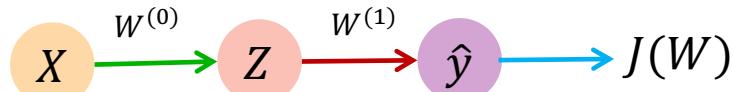
We want to find the network weights that achieve the lowest loss

$$W^* = \operatorname{argmin}_W J(W); \quad W = \{W^{(0)}, W^{(1)}\}$$

With gradient descent

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

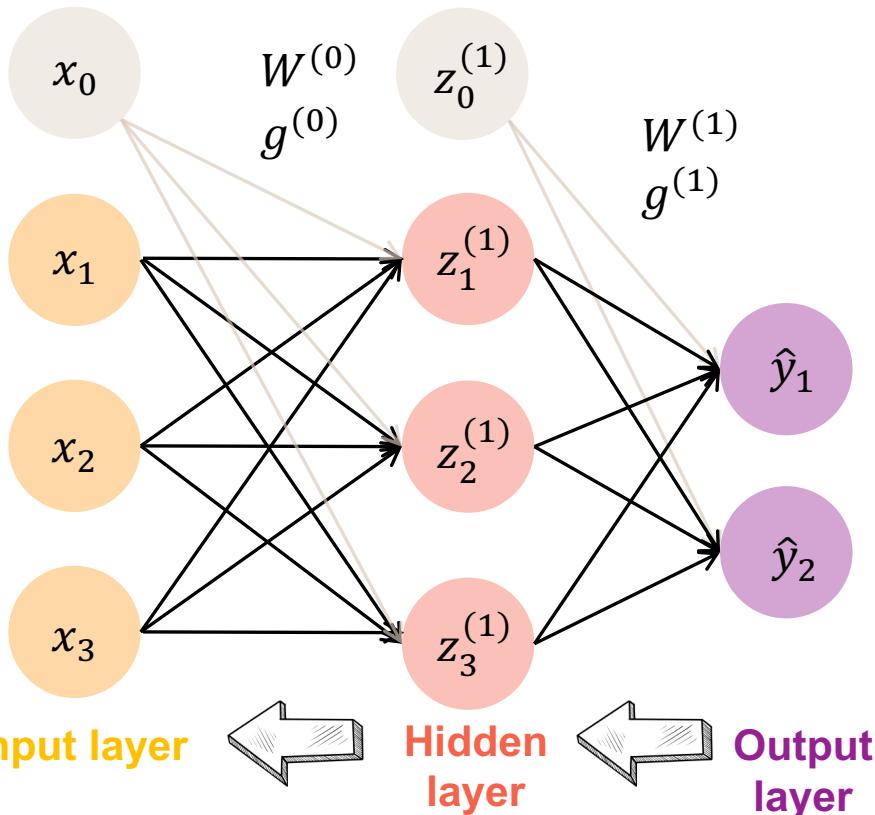
Calculating gradients with the chain rule



- 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

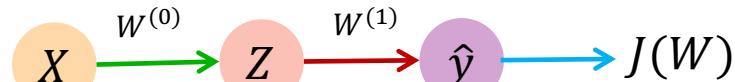


We want to find the network weights that achieve the lowest loss
 $\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W}); \mathbf{W} = \{\mathbf{W}^{(0)}, \mathbf{W}^{(1)}\}$

With gradient descent

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

Calculating gradients with the chain rule



$$\frac{\partial J(W)}{\partial W^{(1)}} = \frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W^{(1)}} \quad \left. \begin{array}{l} \frac{\partial J(W)}{\partial \hat{y}} = \hat{y} - y \\ \frac{\partial \hat{y}}{\partial W^{(1)}} = \end{array} \right\}$$

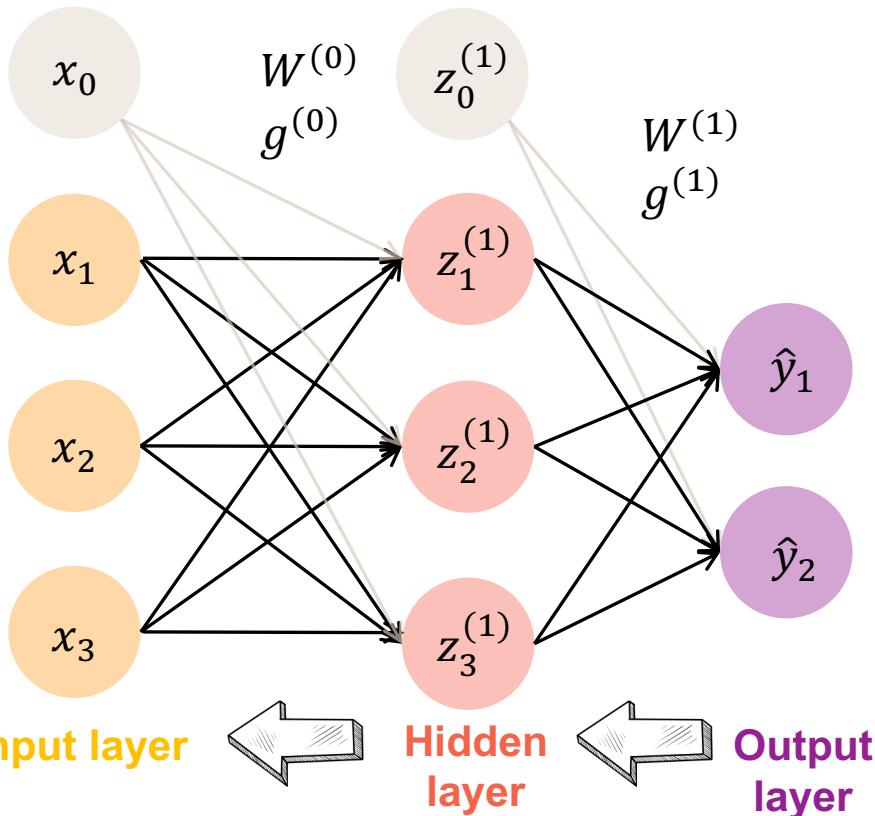
$$J(W) = \frac{1}{2}(\hat{y} - y)^2 = \frac{1}{2}(\hat{y}^2 + y^2 - 2\hat{y}y)$$

$$\begin{aligned} \hat{y} &= g^{(1)}(W^{(1)}g^{(0)}(W^{(0)}X)) \\ &= g^{(1)}(W^{(1)}Z^{(1)}) \end{aligned}$$

$$\left. \begin{array}{l} \frac{\partial J(W)}{\partial \hat{y}} \\ \frac{\partial \hat{y}}{\partial W^{(1)}} = g'^{(1)}(W^{(1)}Z^{(1)})Z^{(1)} \end{array} \right\}$$

Backpropagation

Single layer neural network



We want to find the network weights that achieve the lowest loss
 $W^* = \operatorname{argmin}_W J(W); W = \{W^{(0)}, W^{(1)}\}$

With gradient descent

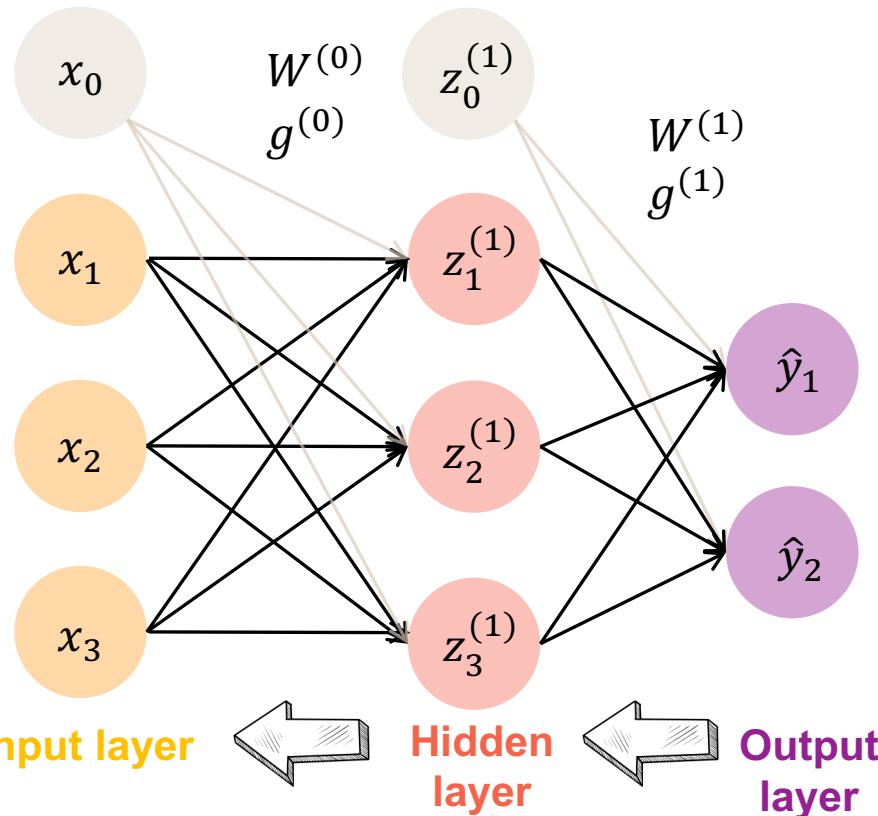
$$W \leftarrow W - \alpha \frac{\partial J(W)}{\partial W}$$

Calculating gradients with the chain rule

$$\begin{aligned} X &\xrightarrow{W^{(0)}} Z & Z &\xrightarrow{W^{(1)}} \hat{y} & \hat{y} &\rightarrow J(W) \\ \frac{\partial J(W)}{\partial W^{(0)}} &= \frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Z^{(1)}} \frac{\partial Z^{(1)}}{\partial W^{(0)}} & Z^{(1)} &= g^{(0)}(W^{(0)}X) \\ && \hat{y} - y && g'^{(1)}(W^{(1)}Z^{(1)})Z^{(1)} \\ && && g'^{(0)}(W^{(0)}X)X \end{aligned}$$

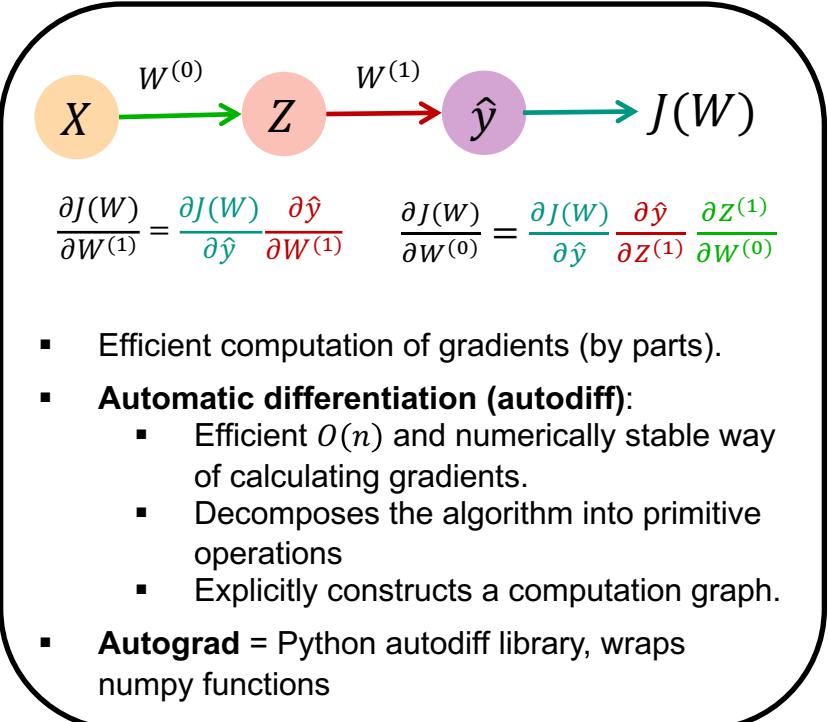
Backpropagation

Single layer neural network

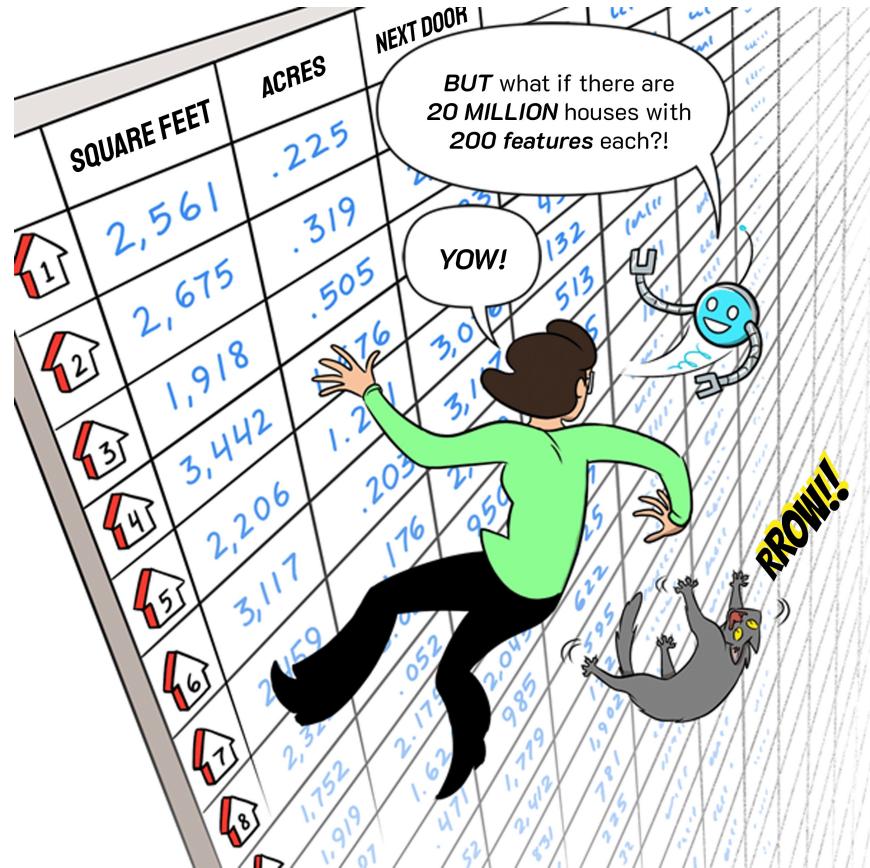
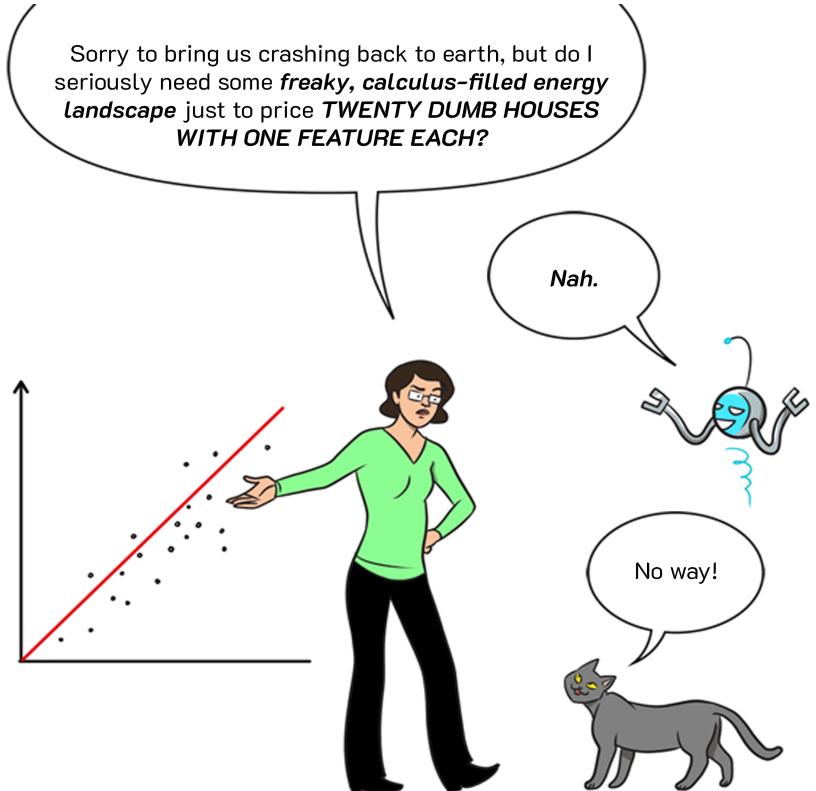


We want to find the network weights that achieve the lowest loss

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W}); \quad \mathbf{W} = \{\mathbf{W}^{(0)}, \mathbf{W}^{(1)}\}$$



NNs: computationally efficient for big data!



Thank you for your attention!

What questions do you have?

Resources:

<https://ml-cheatsheet.readthedocs.io/>

<https://notesonai.com/>

[https://buildmedia.readthedocs.org/media/
pdf/ml-cheatsheet/latest/ml-cheatsheet.pdf](https://buildmedia.readthedocs.org/media/pdf/ml-cheatsheet/latest/ml-cheatsheet.pdf)

<http://introtodeeplearning.com/>

<https://www.offconvex.org/>