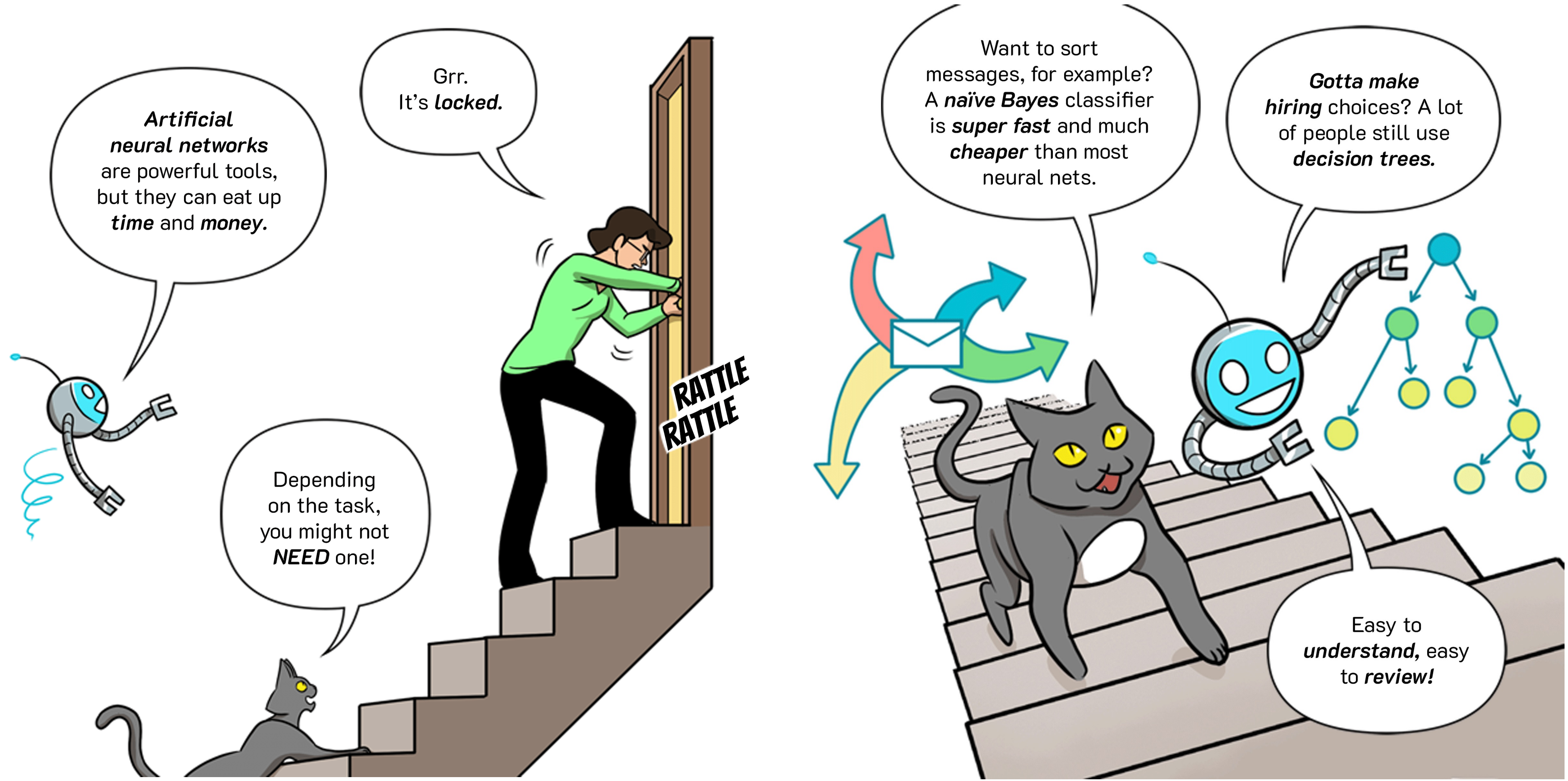
Introduction to **Artificial Neural Networks**

Andrea Santamaría García

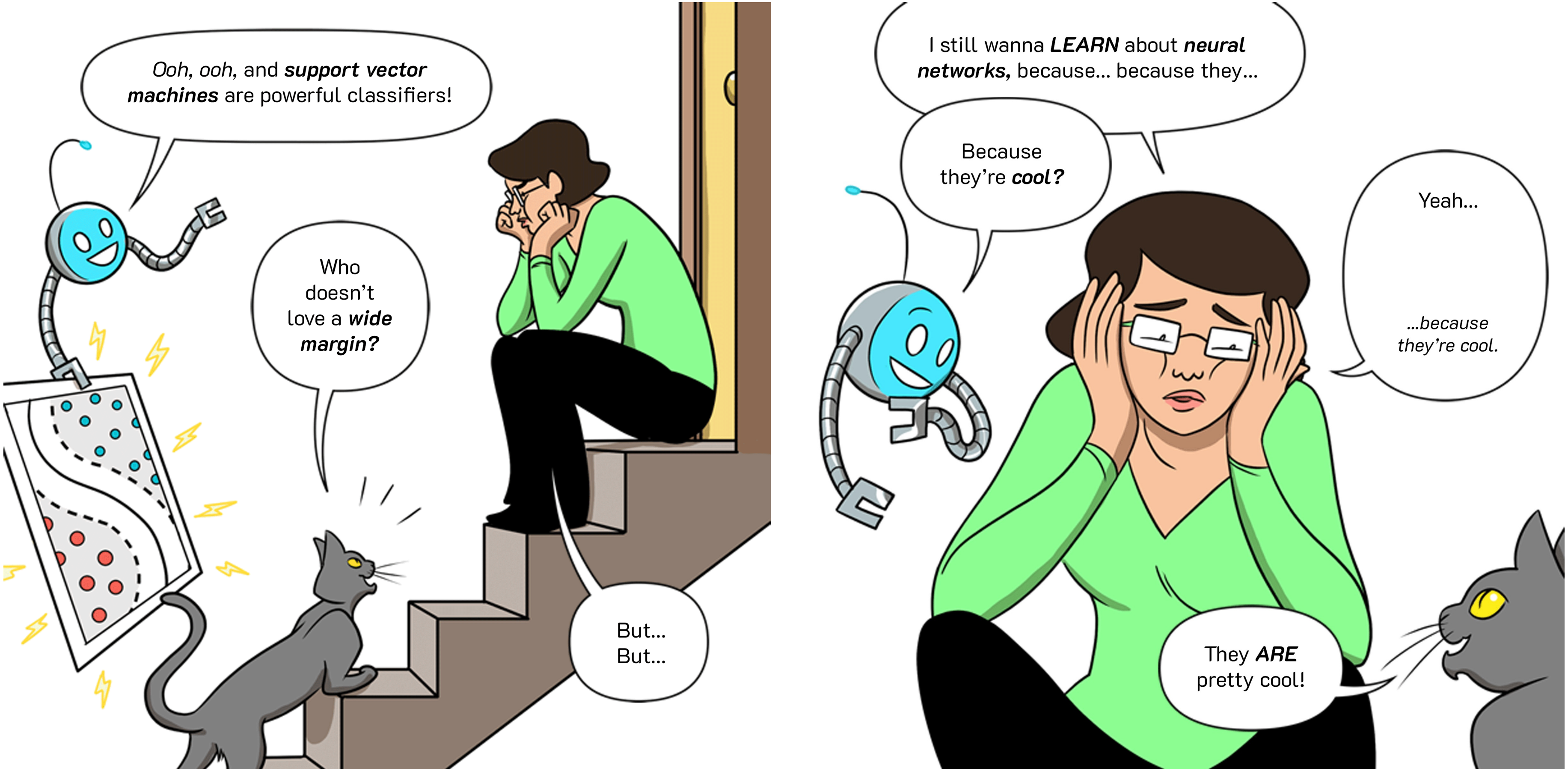
Laboratory for Applications of Synchrotron Radiation (KIT-LAS)

07/09/2022

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!*


Artificial neural networks are powerful tools, but they can eat up time and money.
Grr. It's locked.
RATTLE RATTLE
Depending on the task, you might not NEED one!
Want to sort messages, for example? A naïve Bayes classifier is super fast and much cheaper than most neural nets.
Gotta make hiring choices? A lot of people still use decision trees.
Easy to understand, easy to review!

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


Ooh, *ooh*, and ***support vector machines*** are powerful classifiers!
Who doesn't love a ***wide margin?***
But... But...
I still wanna ***LEARN*** about ***neural networks,*** because... because they...
Because they're ***cool?***
Yeah...
...because they're cool.
They ***ARE*** pretty cool!

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

UNIVARIATE LINEAR REGRESSION (one prediction)

Simple (one feature)

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

Hypothesis function: $h_\theta(x_i) = \theta_0 + \theta_1 x_i$

Estimated from data — **weights**

- regression coefficients
- parameters of the model

$h: x \to y$

Space of input variables = **feature**

independent variable

continuous: regression problem
discrete: classification problem

Space of output variables = **estimated value**

- dependent variable
- scalar response

Multiple (several features)

$$h_\theta(x) = \theta_0 + \sum_{i,\,k=1}^{n,\,p} \theta_k \phi_k(x_i) \quad \begin{cases} i = 1, \dots, \boldsymbol{n} & \text{data points} \\ k = 1, \dots, \boldsymbol{p} & \text{features} \\ x_{i0} = 1 & \text{pseudo-variable} \\ \phi_k(x_i) = x_{ik} & \text{basis function} \end{cases}$$

bias

Polynomial

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

⚠ Fits a nonlinear model to the data, but it's still linear in the parameters θ of the model

Matrix notation $h_\theta(x) = \theta^T x = x^T \theta = \boldsymbol{X\theta}$

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

UNIVARIATE LINEAR REGRESSION

Loss (cost) function

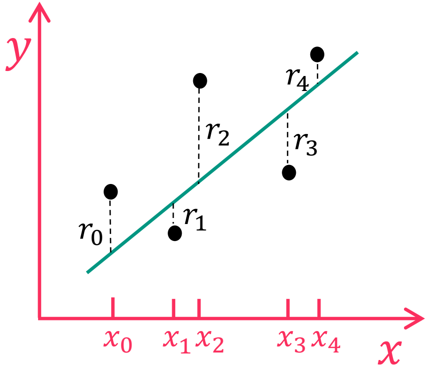
Goal: choose θ_k such that $h_\theta(x_i)$ is as close to y_i as possible

$$h_\theta(x_i) - y_i = r_i$$

as small as possible
= minimization problem

residual

error/disturbance/noise



Assumption:

observations $(x_i,\ y_i)$ = result of $\underbrace{\text{random}}_{\text{normal}}$ deviations

Least Squares

Reasonable choice that works well for many regression problems (desirable properties)

Find the analytical minimum of:

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

loss (cost) function

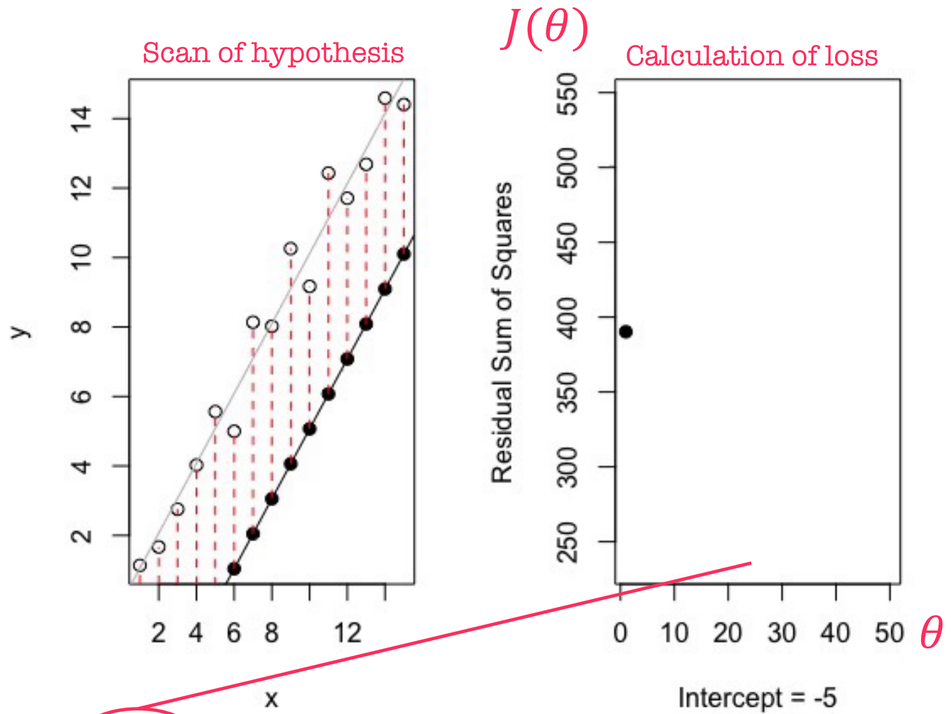
- Takes an average difference of the predictions of the hypothesis
- “Intuitive” number to represent deviation from target
- **Measures of performance of a model**

UNIVARIATE LINEAR REGRESSION

Loss functions - least squares

⚠ **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.



Least Squares

Find the analytical minimum of:

$$\sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n}\left(h_\theta(x_i) - y_i\right)^2 = |\boldsymbol{y} - \boldsymbol{X\theta}|^2 = \boxed{J(\theta_k)}$$

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

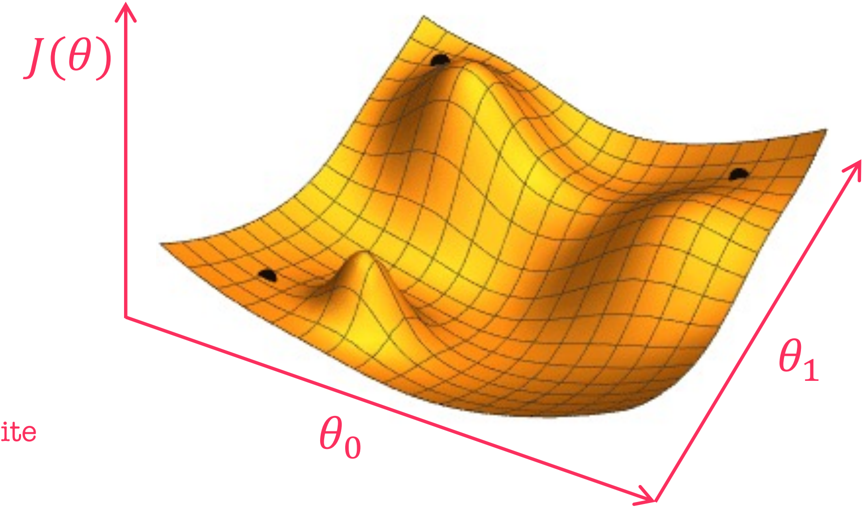
OPTIMIZERS

Example: gradient descent

Algorithm to iteratively solve $argmin_\theta \; J(\theta_k)$

$$\vec{\theta} := \vec{\theta} - \boxed{\alpha}\,\underbrace{\overbrace{\nabla}^{\text{first order}} J(\theta_k)}_{\text{take repeated steps in the opposite direction of the gradient}}$$

step size
learning rate



$$J(\theta_k) = \frac{1}{2n}\sum_{i=1}^{n}(h_\theta(x_i) - y_i)^2$$

$$h_\theta(x) = \theta_0 + \theta_1 x$$

$$\frac{\partial J(\theta_0,\theta_1)}{\partial\theta_0} = \frac{1}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i)$$

$$\frac{\partial J(\theta_0,\theta_1)}{\partial\theta_1} = \frac{1}{n}\sum_{i=1}^{n}x_i(\theta_0 + \theta_1 x_i - y_i)$$

$$\left.\begin{aligned}\theta_0 &:= \theta_0 - \alpha\frac{\partial J(\theta_0,\theta_1)}{\partial\theta_0}\\ \theta_1 &:= \theta_1 - \alpha\frac{\partial J(\theta_0,\theta_1)}{\partial\theta_1}\end{aligned}\right\}\text{updated simultaneously}$$

- Depending on the initial (θ_0, θ_1) optimization can end up at different points
- If the loss function is not strictly convex and saddle points exist finding the global minimum is not guaranteed
- Works in any number of dimensions

- Iteratively (this example): $\mathcal{O}(features\,.\,points^2)$
- Analitycally (normal equation): $\underbrace{\mathcal{O}(points^3)}_{\text{matrix inversion}}$

Animation from Wikimedia commons

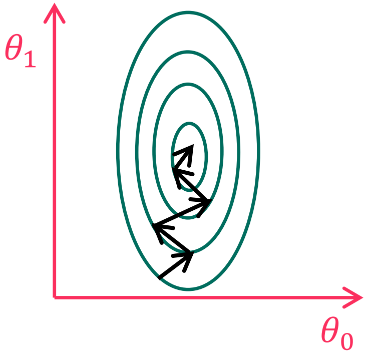
OPTIMIZERS

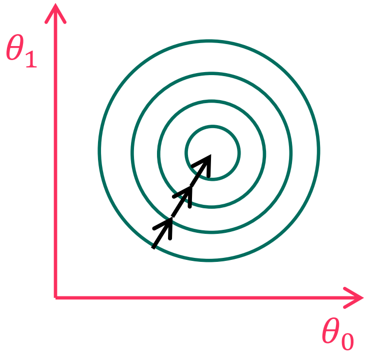
Feature scaling

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

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

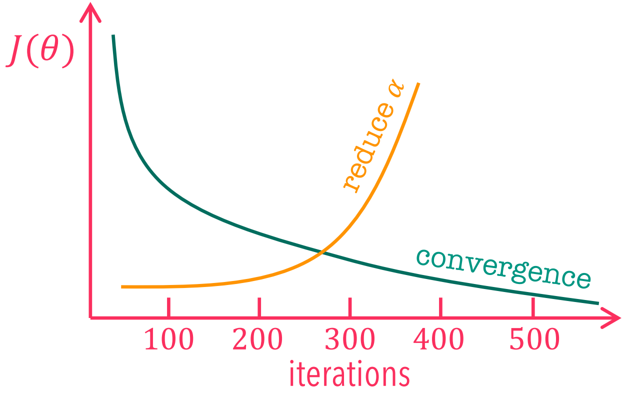
- Normalization: $-1 \leq x_i \leq 1$ or $0 \leq x_i \leq 1$
- Standardization: $\mu = 0, \ \sigma^2 = 1$





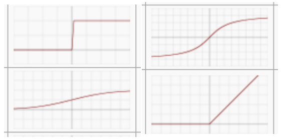
Learning rate

- The optimizer is working correctly if the loss decreases after every iteration
- Usual to **declare convergence tests** (e.g., declare convergence when $J(\theta)$ decreases by less than 10^{-3} in one iteration)
  - α too small: slow convergence
  - α too large: may not converge



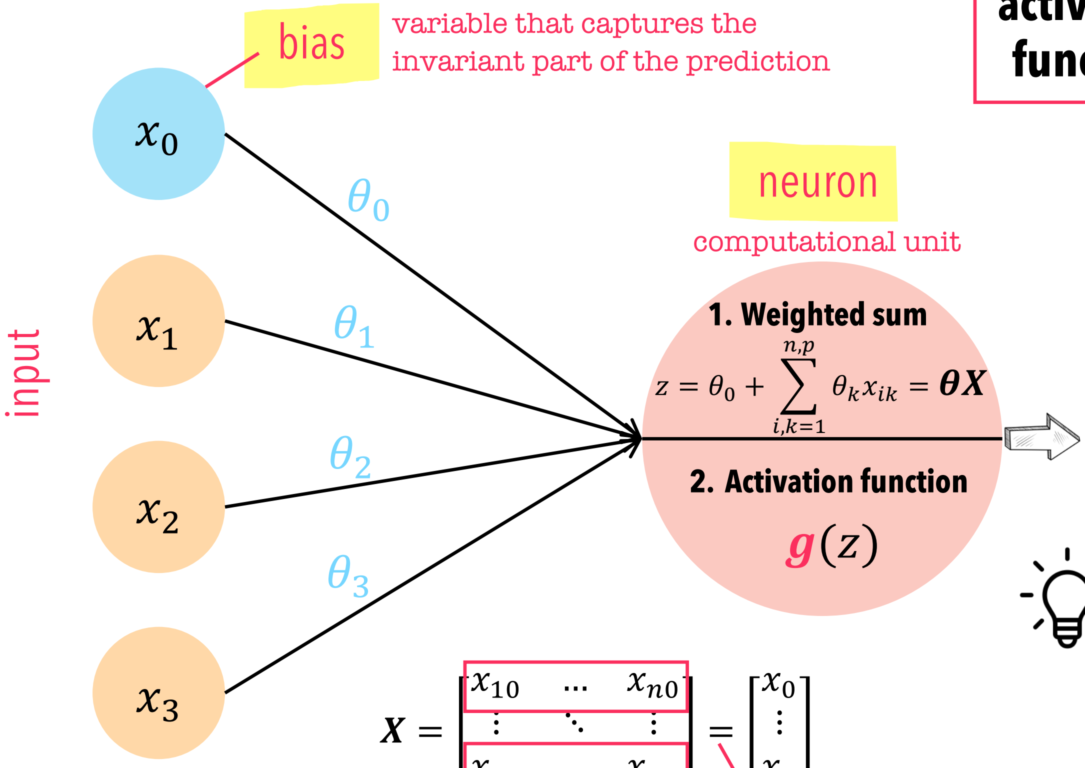
NEURAL NETWORKS (NNs)

Perceptron



activation function

- 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?)



bias: variable that captures the invariant part of the prediction

neuron: computational unit

1. Weighted sum

$$z = \theta_0 + \sum_{i,k=1}^{n,p} \theta_k x_{ik} = \boldsymbol{\theta X}$$

2. Activation function $g(z)$

$$g(z) = h_\theta(x) = \hat{y}$$

prediction

$$\boldsymbol{X} = \begin{bmatrix} x_{10} & \cdots & x_{n0} \\ \vdots & \ddots & \vdots \\ x_{1p} & \cdots & x_{np} \end{bmatrix} = \begin{bmatrix} x_0 \\ \vdots \\ x_p \end{bmatrix}$$

n points, p features

we don't know how the data points n will be fed at a time (batches)

Universal approximation theorem

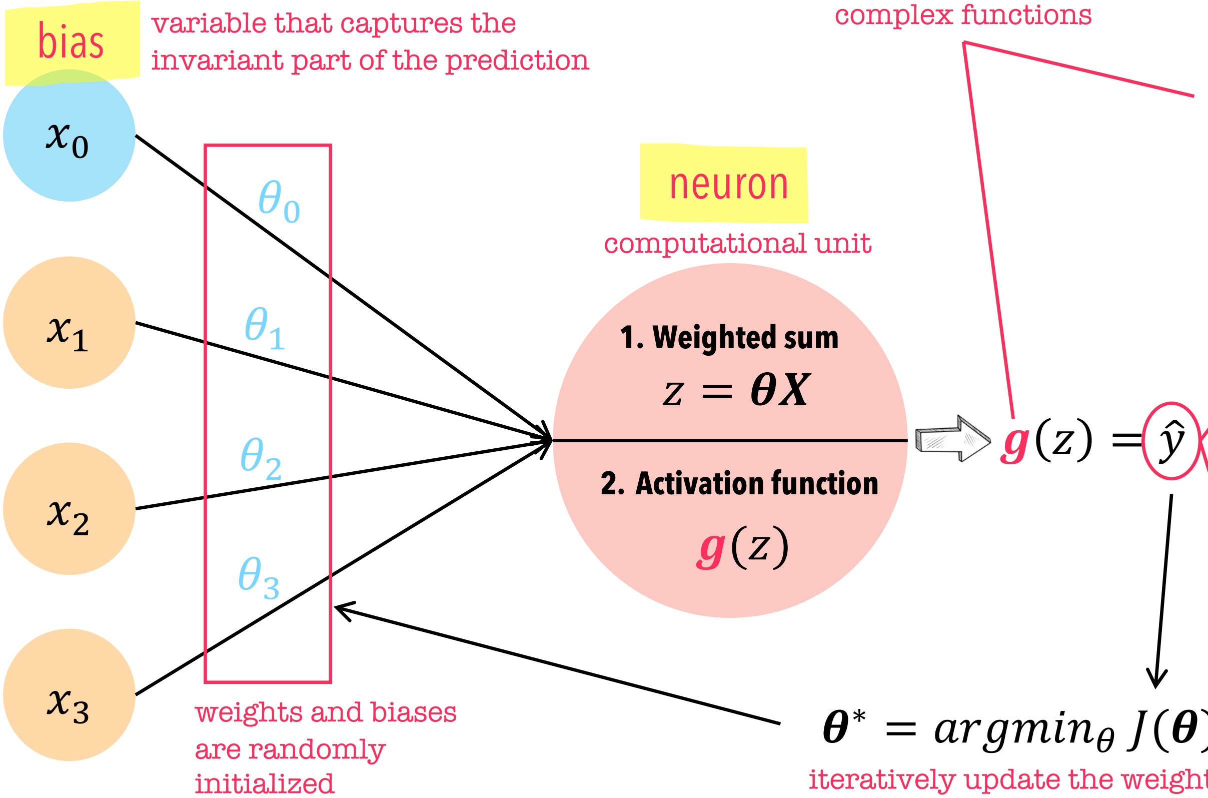
When the activation function is nonlinear, then a two layer NN can be proven to be a universal function approximator

NEURAL NETWORKS (NNs)

Perceptron

bias

variable that captures the invariant part of the prediction



weights and biases are randomly initialized

neuron

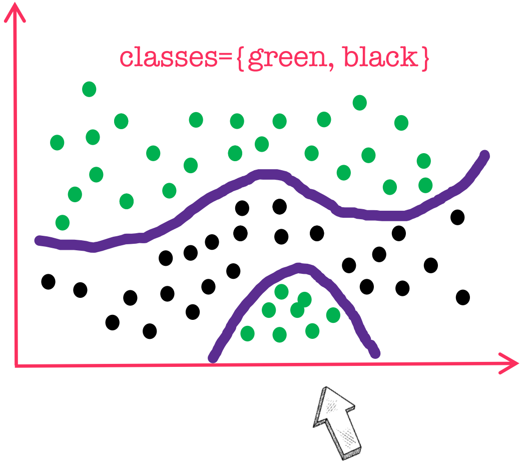
computational unit

1. Weighted sum
$$z = \boldsymbol{\theta X}$$

2. Activation function
$$\boldsymbol{g}(z)$$

$$\boldsymbol{g}(z) = \hat{y}$$

introduces nonlinearities to approximate arbitrarily complex functions



classes={green, black}

discrete: classification problem

probabilities between 0 and 1 of belonging to a specific class (logistic regression)

continuous: regression problem

$$\boldsymbol{\theta}^* = argmin_\theta\, J(\boldsymbol{\theta})$$

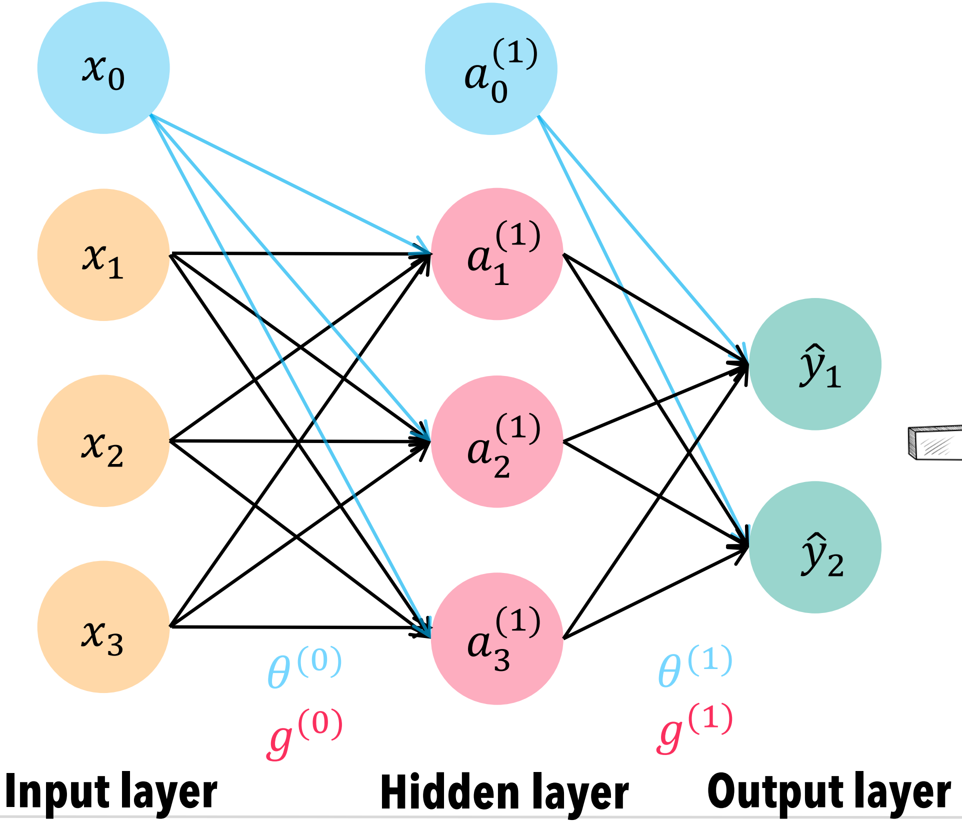
iteratively update the weights

FORWARD PROPAGATION

Single layer neural network

New notation:

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

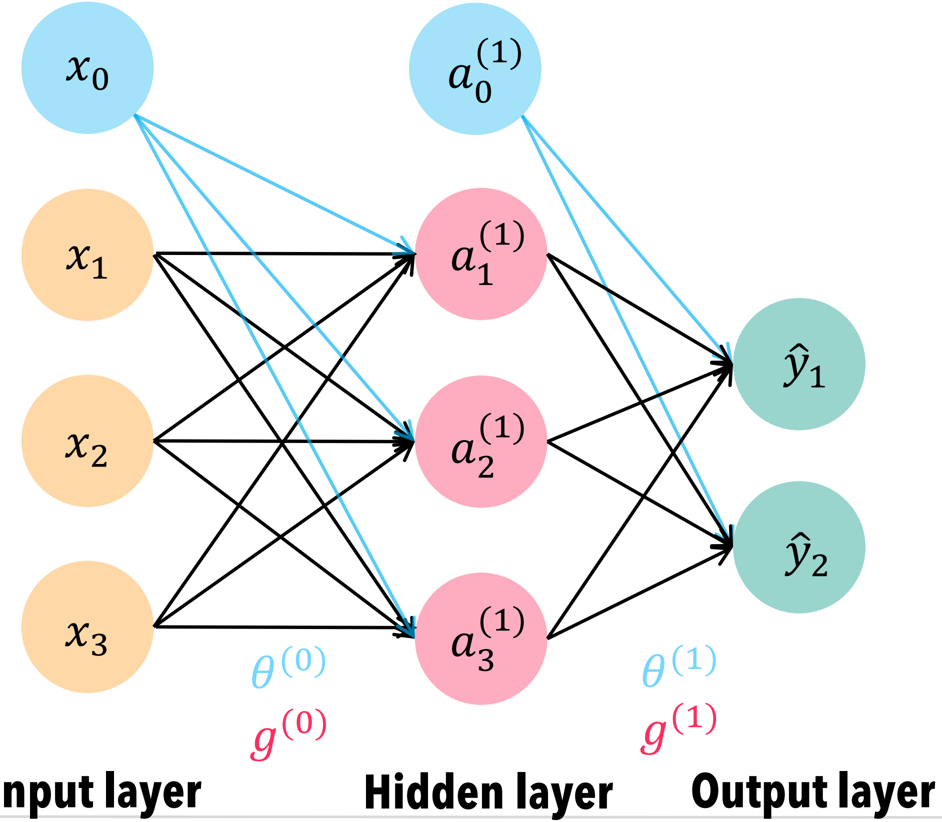


$$\boxed{\hat{y} = g^{(1)}(\overbrace{\boldsymbol{\theta}^{(1)}\overbrace{g^{(0)}(\underbrace{\boldsymbol{\theta}^{(0)}\boldsymbol{X}}_{z^{(1)}})}^{\boldsymbol{a}^{(1)}}}^{z^{(2)}})}$$

multilayer network evaluates compositions of functions computed at individual neurons

FORWARD PROPAGATION

Single layer neural network



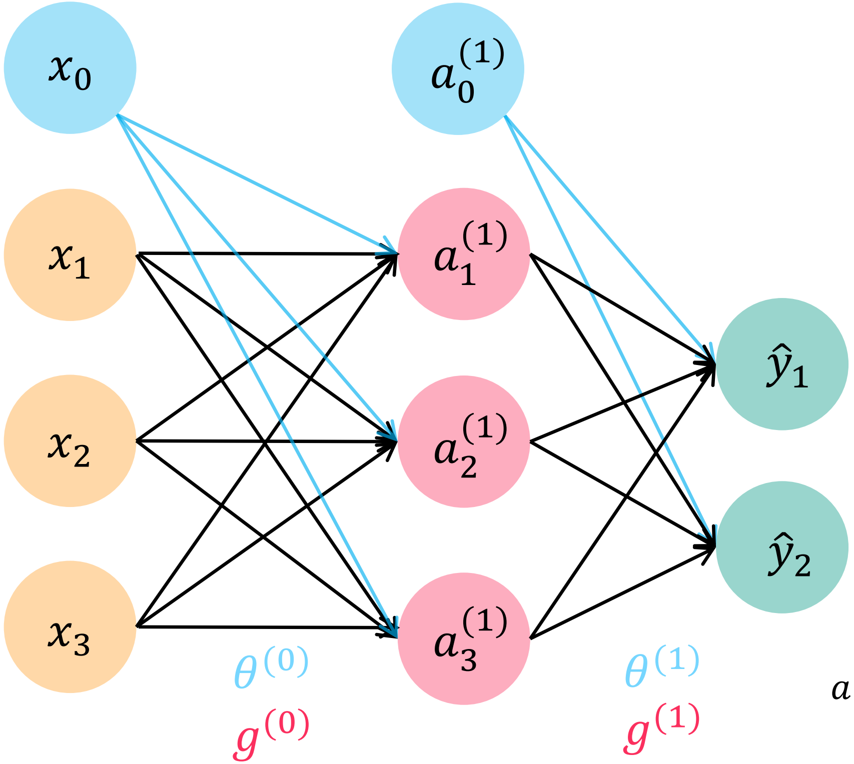
$$\hat{y} = g^{(1)}(\overbrace{\boldsymbol{\theta}^{(1)}\overbrace{g^{(0)}(\underbrace{\boldsymbol{\theta}^{(0)}\boldsymbol{X}}_{z^{(1)}})}^{\boldsymbol{a}^{(1)}}}^{z^{(2)}})$$

$$\boldsymbol{\theta}^{(0)} = \underbrace{\begin{bmatrix} \theta_{10}^{(0)} & \theta_{11}^{(0)} & \theta_{12}^{(0)} & \theta_{13}^{(0)} \\ \theta_{20}^{(0)} & \theta_{21}^{(0)} & \theta_{22}^{(0)} & \theta_{23}^{(0)} \\ \theta_{30}^{(0)} & \theta_{31}^{(0)} & \theta_{32}^{(0)} & \theta_{33}^{(0)} \end{bmatrix}}_{\text{neurons in input layer + bias}} \left.\vphantom{\begin{bmatrix} a \\ b \\ c \end{bmatrix}}\right\} \text{neurons in layer 2} \qquad \boldsymbol{X} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

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

FORWARD PROPAGATION

Single layer neural network



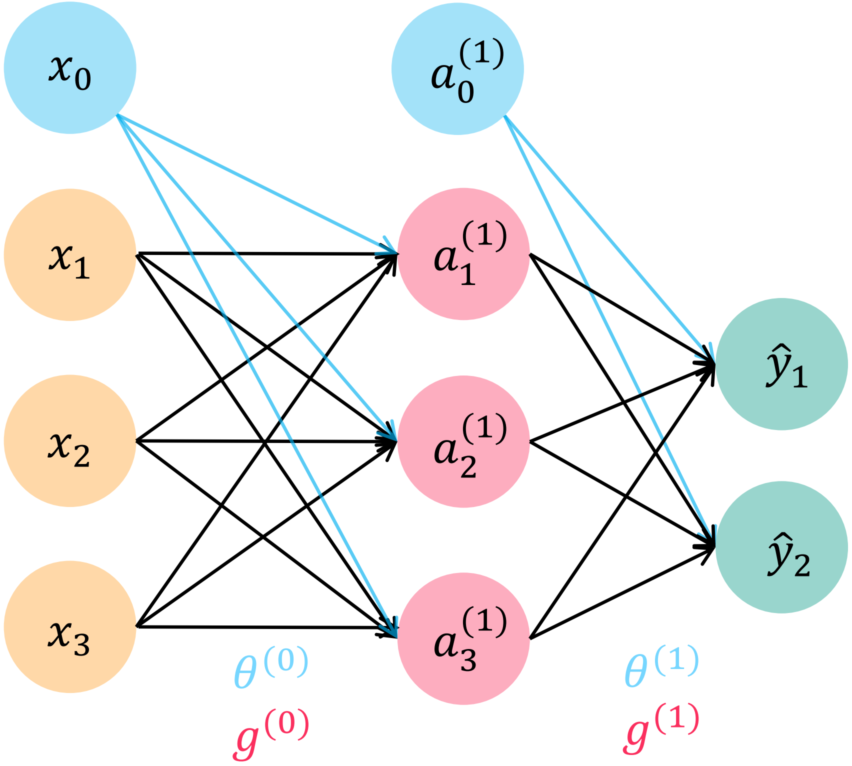
Input layer **Hidden layer** **Output layer**

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

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

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

BACKPROPAGATION
Single layer neural network



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

$$\boldsymbol{\theta}^* = argmin_\theta\, J(\boldsymbol{\theta}) \qquad \boldsymbol{\theta} = \{\theta^{(1)}, \theta^{(2)}\}$$

1. Gradient calculation:

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

$$= g'^{(1)}(\theta^{(1)} a^{(1)}) a^{(1)}$$

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

least squares

$$= \hat{y} - y$$

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

$$= g'^{(0)}(\theta^{(0)} X) X$$

$$= g'^{(1)}(\theta^{(1)} a^{(1)})\ \theta^{(1)}$$

2. Weight update:

$$\begin{cases} \theta^{(0)} := \theta^{(0)} - \alpha \dfrac{\partial J(\theta^{(0)}, \theta^{(1)})}{\partial \theta^{(0)}} \\ \theta^{(1)} := \theta^{(1)} - \alpha \dfrac{\partial J(\theta^{(0)}, \theta^{(1)})}{\partial \theta^{(1)}} \end{cases}$$

gradient descent

STOCHASTIC GRADIENT DESCENT

How much computational time does it take to calculate the gradients?

- Number of points n
- Number of features p

$$\underbrace{\sum_{i=1}^{n}(h_\theta(x_i)-y_i)^2}_{n \text{ terms}} = J(\theta_k)$$

$$\vec{\theta} := \vec{\theta} - \underbrace{\alpha\,\nabla J(\theta_k)}_{\text{Calculate gradient } p \text{ times}}$$

- ➢ 1e4 points
- ➢ 1e1 features

= 1e5 computations

In stochastic gradient descent (SGD) the gradient is approximated by a gradient at a single sample:

repeat until approx. minimum

```
Randomly shuffle samples in the data set
for $i = 1, \dots, n$ do:
    $\vec{\theta} := \vec{\theta} - \alpha\,\nabla J(\theta_k)$
```

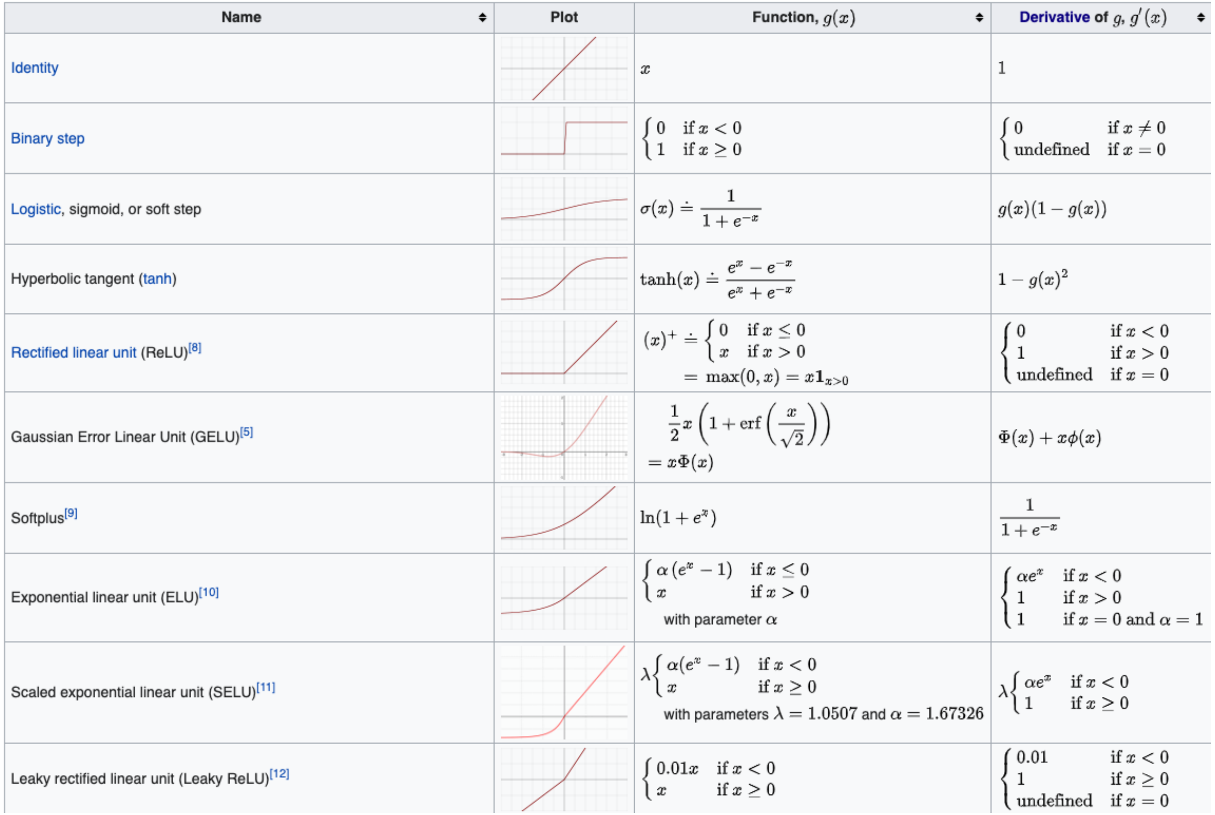
More than one training example at each step = **mini batch**

Image from wikipedia

NEURAL NETWORKS
Activation functions

Differentiable, quickly converging wrt the weights

* There are also radial basis functions (RBF) which are efficient as universal function approximators (Gaussian, multiquadratics)



Name	Plot	Function, $g(x)$	Derivative of g, $g'(x)$
Identity		x	1
Binary step		$\begin{cases}0 & \text{if } x<0\\ 1 & \text{if } x\geq 0\end{cases}$	$\begin{cases}0 & \text{if } x\neq 0\\ \text{undefined} & \text{if } x=0\end{cases}$
Logistic, sigmoid, or soft step		$\sigma(x) \doteq \frac{1}{1+e^{-x}}$	$g(x)(1-g(x))$
Hyperbolic tangent (tanh)		$\tanh(x) \doteq \frac{e^x-e^{-x}}{e^x+e^{-x}}$	$1-g(x)^2$
Rectified linear unit (ReLU)[8]		$(x)^+ \doteq \begin{cases}0 & \text{if } x\leq 0\\ x & \text{if } x>0\end{cases}$ $= \max(0,x) = x\mathbf{1}_{x>0}$	$\begin{cases}0 & \text{if } x<0\\ 1 & \text{if } x>0\\ \text{undefined} & \text{if } x=0\end{cases}$
Gaussian Error Linear Unit (GELU)[5]		$\frac{1}{2}x\left(1+\text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$ $= x\Phi(x)$	$\Phi(x)+x\phi(x)$
Softplus[9]		$\ln(1+e^x)$	$\frac{1}{1+e^{-x}}$
Exponential linear unit (ELU)[10]		$\begin{cases}\alpha\,(e^x-1) & \text{if } x\leq 0\\ x & \text{if } x>0\end{cases}$ with parameter α	$\begin{cases}\alpha e^x & \text{if } x<0\\ 1 & \text{if } x>0\\ 1 & \text{if } x=0 \text{ and } \alpha=1\end{cases}$
Scaled exponential linear unit (SELU)[11]		$\lambda\begin{cases}\alpha(e^x-1) & \text{if } x<0\\ x & \text{if } x\geq 0\end{cases}$ with parameters $\lambda=1.0507$ and $\alpha=1.67326$	$\lambda\begin{cases}\alpha e^x & \text{if } x<0\\ 1 & \text{if } x\geq 0\end{cases}$
Leaky rectified linear unit (Leaky ReLU)[12]		$\begin{cases}0.01x & \text{if } x<0\\ x & \text{if } x\geq 0\end{cases}$	$\begin{cases}0.01 & \text{if } x<0\\ 1 & \text{if } x\geq 0\\ \text{undefined} & \text{if } x=0\end{cases}$

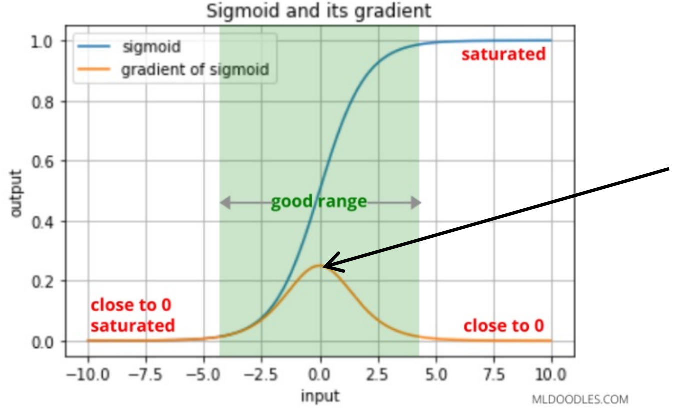
Image from wikipedia

NEURAL NETWORKS

Vanishing gradients

appear in backpropagation using gradient-based methods in deep networks

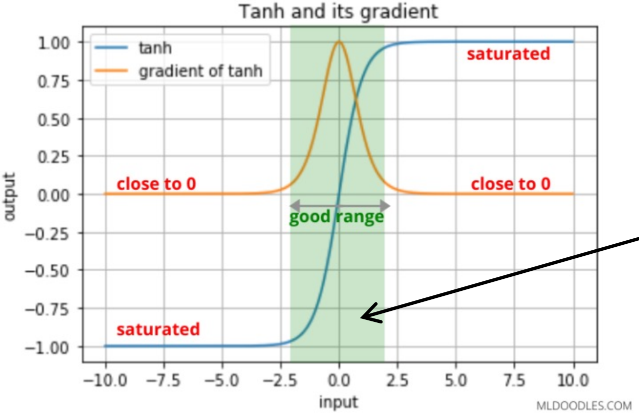
not goodin hidden layers



Maximum of gradient 0.25
With chain rule the gradient product can become very small

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta^{(0)}} = \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a^{(1)}} \cdots \frac{\partial a^{(l)}}{\partial \theta^{(0)}}$$

$$0.2 \times 0.15 \times 0.22 \times 0.09 \ldots$$

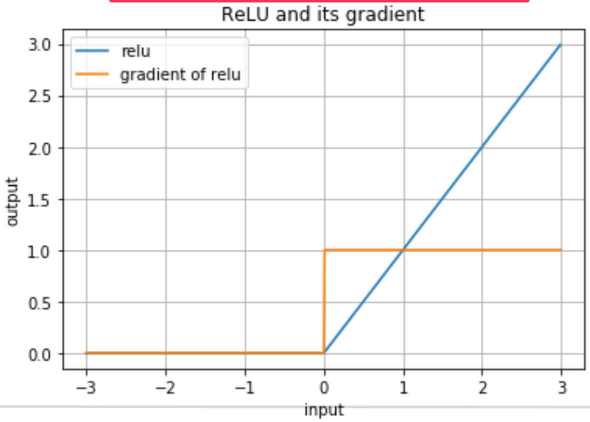


Very narrow range, small values

When the partial derivative vanishes the weights are not updated anymore

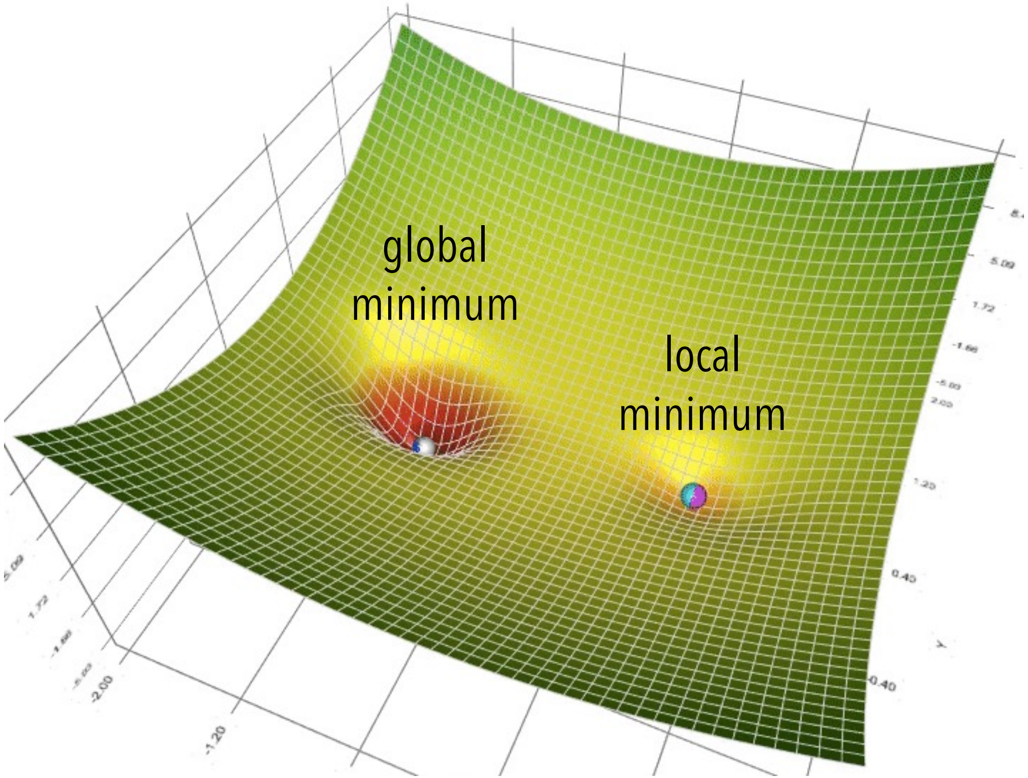
$$\vec{\theta} := \vec{\theta} - \alpha \nabla J(\theta_k)$$

good for hidden layers



Images from mldoodles

NOTE ON OPTIMIZERS IN NNs



Gradient based methods (first order):

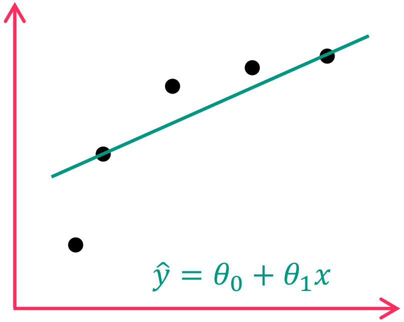
- **Gradient descent**
- **momentum**
- □ **AdaGrad**
- **RMSProp**
- **Adam**

dynamic adjustment of algorithm parameters

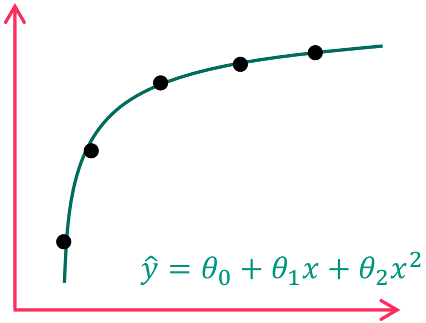
Animation from Lili Jiang

NOTE ON OVERFITTING

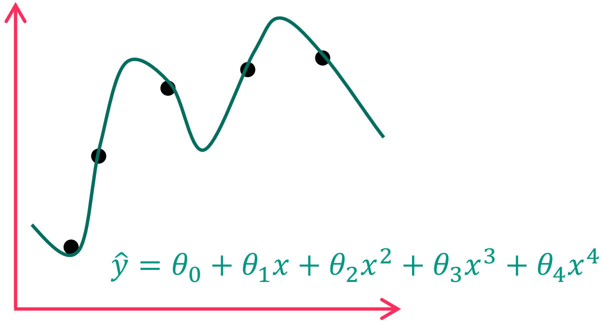
Underfit



Good fit



Overfit



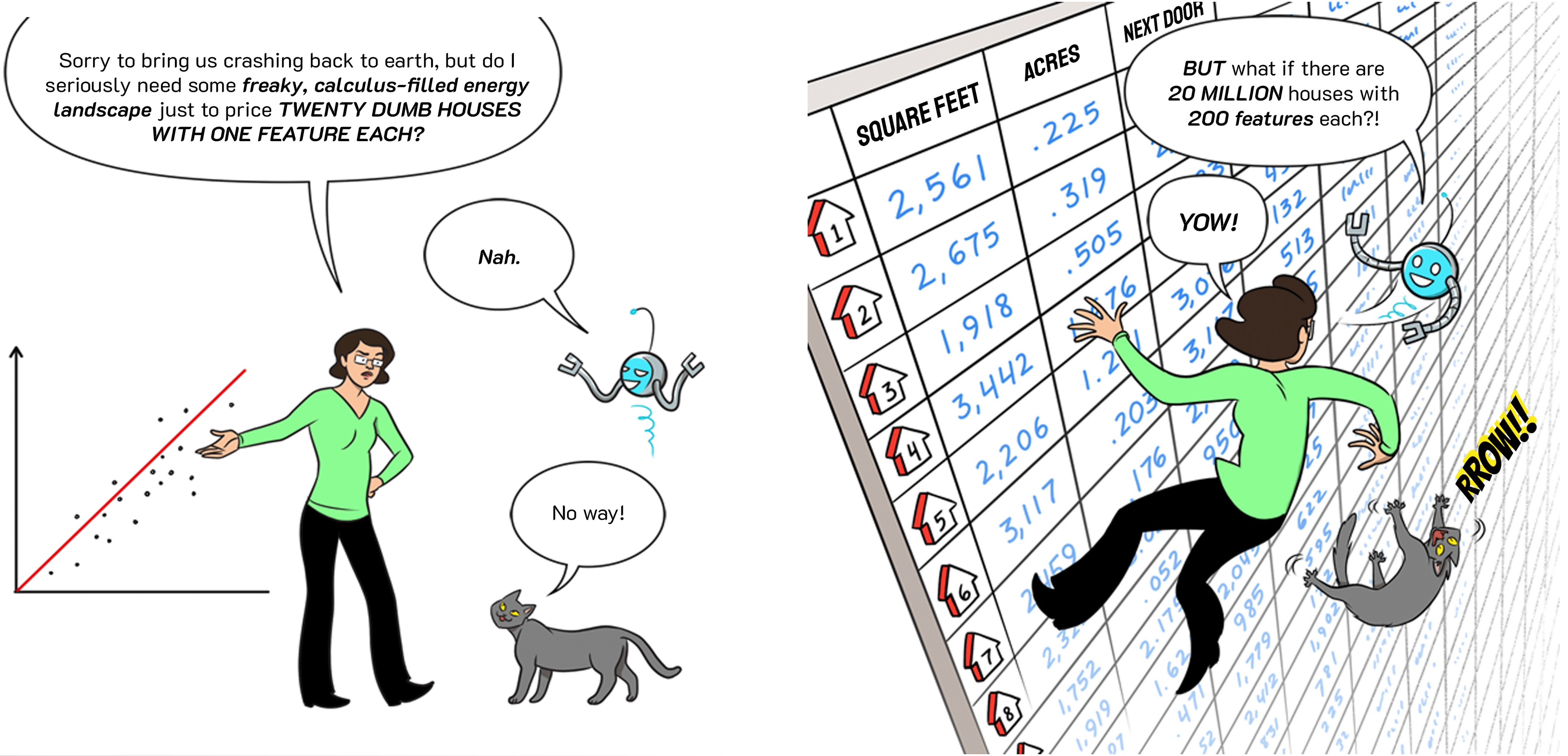
Test for overfitting with independent test dataset
→ Difference in loss = generalization error

Regularization

Improves generalization on useen data by constraining the optimization problem to discourage complex models

- Regularization term in **loss function** (penalty or shrinkage term, L1, L2)
- **Dropout**: not relying on certain nodes that only learn certain patterns (e.g., set 50% of activations to 0 during training)
- **Early stopping**: stop training before overfitting
- **Batch normalization**: normalizing each activation value using the batch (μ, σ)

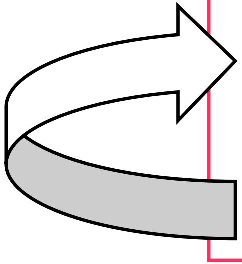
COMPUTATIONALLY EFFICIENT FOR BIG DATA!


Sorry to bring us crashing back to earth, but do I seriously need some *freaky, calculus-filled energy landscape* just to price *TWENTY DUMB HOUSES WITH ONE FEATURE EACH?*
Nah.
No way!
SQUARE FEET
ACRES
NEXT DOOR
BUT what if there are *20 MILLION* houses with *200 features* each?!
YOW!
RROW!!

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

SO WHAT'S THE RECIPE?

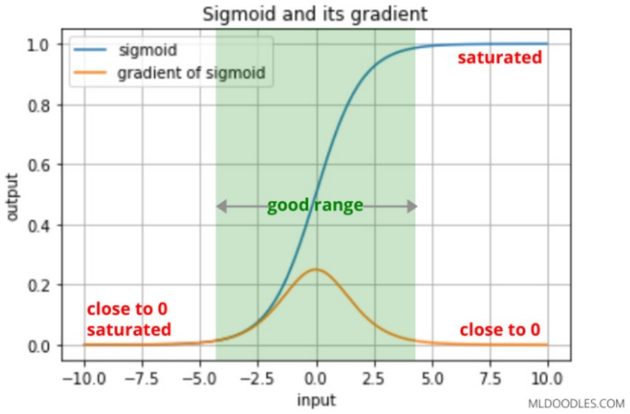
1. Select data features + perform data scaling
2. Choose network architecture
3. Choose activation function for each layer
4. Choose loss function & convergence criteria
5. Choose an optimizer & set its hyperparameters
6. Choose number of epochs to train
7. Decide on regularization techniques

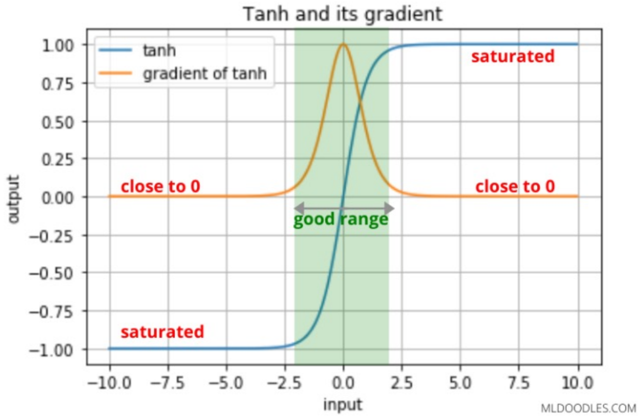


8. Perform forward propagation
9. Compute gradients with backwards propagation
10. Update weights & keep track of loss

LET'S TRY IT!

- Click here
- Why is the example not working?
- How can you solve that?
- Try different activation functions



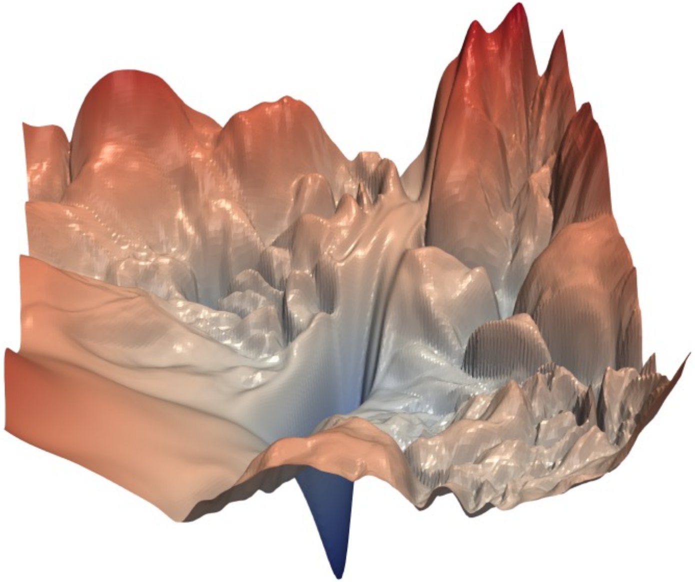


Thank you for your attention!

What questions do you have for me?

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

https://buildmedia.readthedocs.org/media/pdf/ml-cheatsheet/latest/ml-cheatsheet.pdf



Let's connect! andrea.santamaria@kit.edu / @ansantam

Highly non-convex loss of ResNet-56 without skip connections [arxiv:1712.09913]

All icons from this talk from TheNounProject

NOTE ON LOSS FUNCTION FOR CLASSIFICATION

Maximum likelihood estimation (MLE)

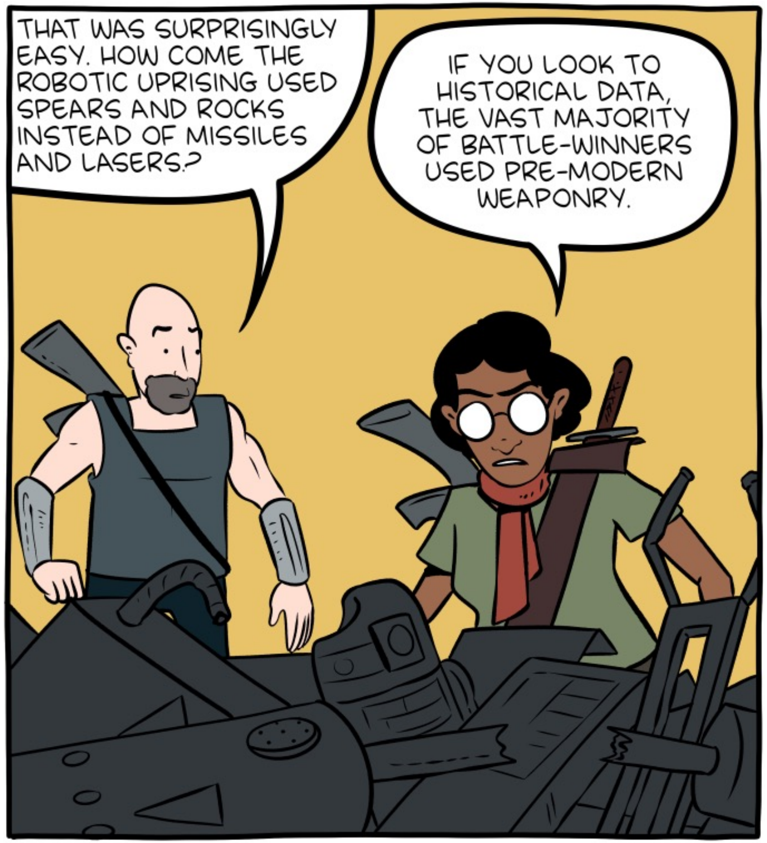
- $\vec{x}$ = random sample from unknown joint probability distribution $p(\vec{y}|\vec{x};\vec{\theta})$ —— probability of y given x and θ (conditional probability)
- $\vec{\theta}$ = parametrization of p
- Optimal $\vec{\theta}$ value can be estimated by maximizing a likelihood function:

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} p\,(y_i|x_i;\theta_i) \quad \rightarrow \quad argmax_\theta\, \mathcal{L}(\theta)$$

for independent observations

Most NNs use the negative log-likelihood as loss function: $\mathrm{J}(\theta) = -\ln(\mathcal{L}(\theta))$

cross-entropy between the training data and model distribution


THAT WAS SURPRISINGLY EASY. HOW COME THE ROBOTIC UPRISING USED SPEARS AND ROCKS INSTEAD OF MISSILES AND LASERS?
IF YOU LOOK TO HISTORICAL DATA, THE VAST MAJORITY OF BATTLE-WINNERS USED PRE-MODERN WEAPONRY.

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

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