Introduction

Most generally, a machine learning algorithm can be though of as a black box. It takes inputs and gives outputs.

The purpose of this course is to show you how to create this 'black box' and tailor it to your needs.

For example, we may create a model that predicts the weather tomorrow, based on meteorological data about the past few days.

The "black box" in fact is a mathematical model. The machine learning algorithm will follow a kind of trial-and-error method to determine the model that estimates the outputs, given inputs.

Once we have a model, we must **train** it. **Training** is the process through which, the model **learns** how to make sense of input data.

Types of machine learning

Supervised

It is called *supervised* as we provide the algorithm not only with the inputs, but also with the targets (desired outputs). This course focuses on supervised machine learning.

Based on that information the algorithm learns how to produce outputs as close to the **targets** as possible.

The objective function in supervised learning is called **loss function** (also cost or error). We are trying to minimize the loss as the lower the loss function, the higher the accuracy of the model.

Common methods:
- Regression
- Classification

Unsupervised

In *unsupervised* machine learning, the researcher feeds the model with inputs, but **not** with targets. Instead she asks it to find some sort of dependence or underlying logic in the data provided.

For example, you may have the financial data for 100 countries. The model manages to divide (cluster) them into 5 groups. You then examine the 5 clusters and reach the conclusion that the groups are: "Developed", "Developing but overachieving", "Developing but underachieving", "Stagnating", and "Worsening".

The algorithm divided them into 5 groups based on **similarities**, but you didn't know what similarities. It could have divided them by location instead.

Common methods:
- Clustering

Reinforcement

In reinforcement ML, the goal of the algorithm is to maximize its reward. It is inspired by human behavior and the way people change their actions according to incentives, such as getting a reward or avoiding punishment.

The objective function is called a **reward function.** We are trying to maximize the reward function.

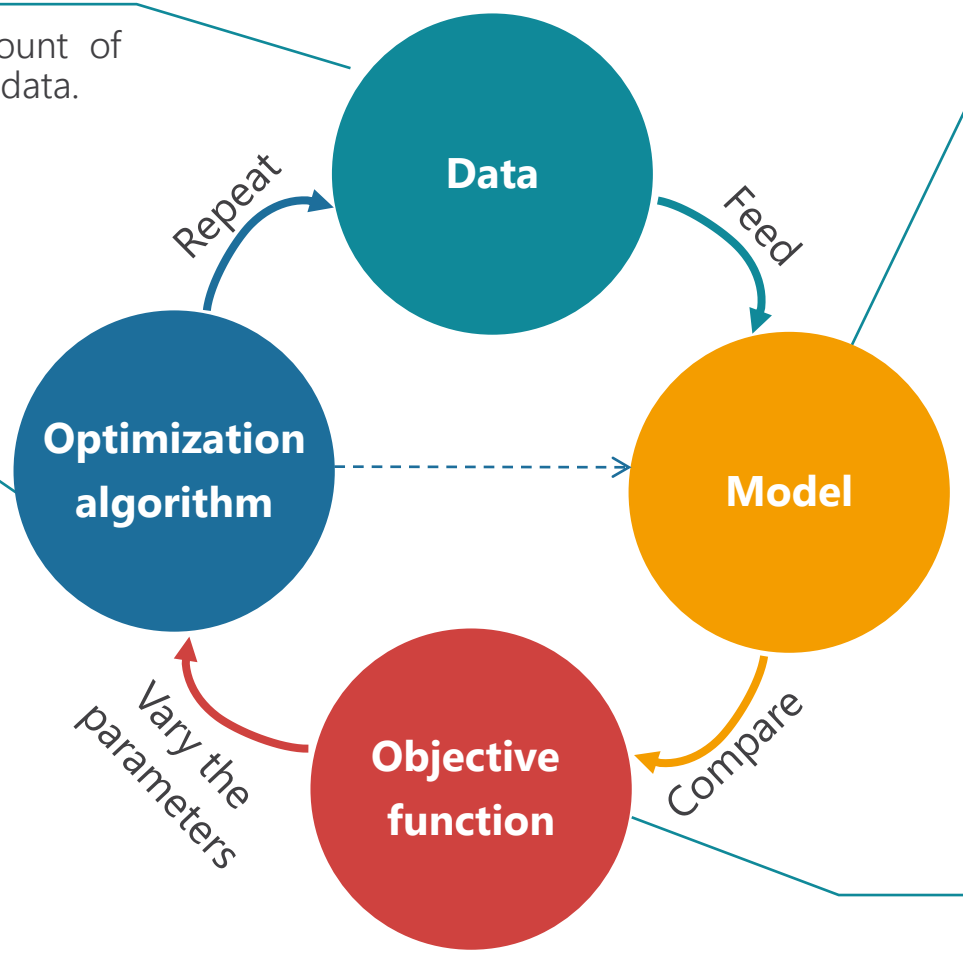
An example is a computer playing Super Mario. The higher the score it achieves, the better it is performing. The score in this case is the objective function.

Common methods:
- Decision process
- Reward system

Building blocks of a machine learning algorithm

The basic logic behind training an algorithm involves four ingredients: data, model, objective function, and optimization algorithm. They are **ingredients**, instead of steps, as the process is iterative.

First, we need to prepare a certain amount of **data** to train on. Usually, we take historical data.



We choose the type of **model**. Roughly speaking, this is some function, which is defined by the *weights* and the *biases*. We feed the input data into the model. Essentially, the idea of the machine learning algorithm is to find the *parameters* for which the model has the highest predictive power.

We achieve the optimization using an **optimization algorithm.** Using the value of the objective function, the optimization algorithm *varies the parameters* of the model. This operation is repeated until we find the values of the parameters, for which the objective function is optimal.

The **objective function** measures the predictive power of our model. Mathematically, the machine learning problem boils down to *optimizing* this function. For example, in the case of loss, we are trying to *minimize* it.

Types of supervised learning

Supervised learning could be split into two subtypes – **regression** and **classification**.

Regression

Regression outputs are continuous numbers.

Examples:

Predicting the EUR/USD exchange rate tomorrow. The output is a number, such as 1.02, 1.53, etc.

Predicting the price of a house, e.g. $234,200 or $512,414.

One of the main properties of the regression outputs is that they are ordered. As 1.02 < 1.53, and 243200 < 512414, we can surely say that one output is bigger than the other.

This distinction proves to be crucial in machine learning as the algorithm somewhat *gets additional information* from the outputs.

Classification

Classification outputs are labels from some sort of class.

Examples:

Classifying photos of animals. The classes are "cats", "dogs", "dolphins", etc.

Predicting conversions in a business context. You can have two classes of customers, e.g."will buy again" and "won't buy again".

In the case of classification, the labels are not ordered and cannot be compared at all. A dog photo is not "more" or "better" than a cat photo (objectively speaking) in the way a house worth $512,414 is "more" (more expensive) than a house worth $234,200.

This distinction proves to be crucial in machine learning, as the different classes are treated on **an equal footing.**

Model

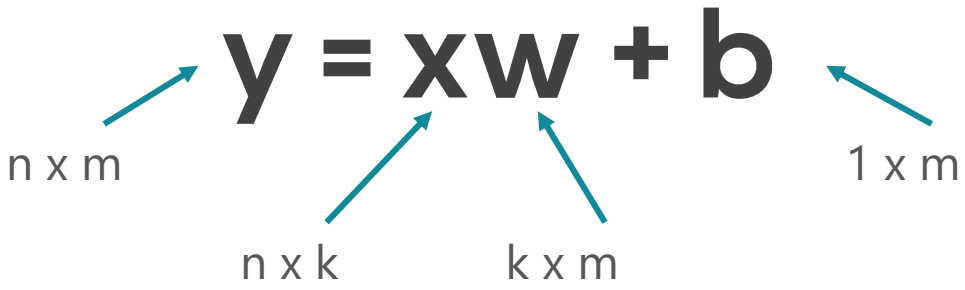
The simplest possible model is **a linear model**. Despite appearing unrealistically simple, in the deep learning context, it is the basis of more complicated models.

There are four elements.

- The input(s), x. That's basically the data that we feed to the model.
- The weight(s), w. Together with the biases, they are called **parameters.** The optimization algorithm will vary the weights and the biases, in order to produce the model that fits the data best.
- The bias(es), b. See weight(s).
- The output(s), y. y is a function of x, determined by w and b.

Each model is determined solely by its parameters (the weights and the biases). That is why we are interested in varying them using the (kind of) trial-and-error method, until we find a model that explains the data sufficiently well.

Model - Continued



Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

A linear model can represent multidimensional relationships. The shapes of y, x, w, and b are given above (notation is arbitrary).

The simplest linear model, where n = m = k = 1.

y	=	x	w	+	b

The simplest linear model, where m = k = 1, n > 1

y_1	=	$x_1w + b$	=	x_1	w	+	b
y_2		$x_2w + b$		x_2			
...		...		...			
y_n		$x_nw + b$		x_n			

Examples:

27	=	4	6	+	3

Since the weights and biases alone define a model, this example shows **the same model** as above but for many data points.

27	=	4	6	+	3
33		5			
...		...			
-3		-1			

Note that we add the bias to each row, essentially simulating an n x 1 matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Model multiple inputs

$$y = xw + b$$

n x m

n x k

k x m

1 x m

Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

We can extend the model to multiple inputs where n, k > 1, m = 1.

$$\begin{bmatrix} y_1 \\ y_2 \\ \dots \\ \dots \\ y_n \end{bmatrix} = \begin{bmatrix} x_{11}w_1 + x_{12}w_2 + \dots + x_{1k}w_k + b \\ x_{21}w_1 + x_{22}w_2 + \dots + x_{2k}w_k + b \\ \dots \\ \dots \\ x_{n1}w_1 + x_{n2}w_2 + \dots + x_{nk}w_k + b \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1k} \\ x_{21} & x_{22} & \dots & x_{2k} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ x_{n1} & x_{n2} & \dots & x_{nk} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_k \end{bmatrix} + \boxed{b}$$

n x 1

n x k

k x 1

1 x 1

Note that we add the bias to each row, essentially simulating an n x 1 matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Example on the next page.

Model multiple inputs

$$y = xw + b$$

n x m — n x k — k x m — 1 x m

Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

We can extend the model to multiple inputs where n, k > 1 m = 1. An example.

y_1	=	9 x (-2) + 12 x 5 +…+ 13 x (-1) + 4	=	9	12	…	13	-2	+	4
y_2		10 x (-2) + 6 x 5 +…+ 2 x (-1) + 4		10	6	…	2	5		
…		…		…	…	…	…	…		
…		…		…	…	…	…	-1		
y_n		7 x (-2) + 7 x 5 +…+ 1 x (-1) + 4		7	7	…	1			
n x 1				n x k				k x 1		1 x 1

Note that we add the bias to each row, essentially simulating an n x 1 matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Model multiple inputs and multiple outputs

$$y = xw + b$$

n x m — n x m

n x k — k x m

1 x m

Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

We can extend the model to multiple inputs where n, k, m > 1.

$$\begin{bmatrix} y_{11} & \cdots & y_{1m} \\ y_{21} & \cdots & y_{2m} \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ y_{n1} & \cdots & y_{nm} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ x_{n1} & x_{n2} & \cdots & x_{nk} \end{bmatrix} \begin{bmatrix} w_{11} & \cdots & w_{1m} \\ w_{21} & \cdots & w_{2m} \\ \cdots & \cdots & \cdots \\ w_{k1} & \cdots & w_{km} \end{bmatrix} + \begin{bmatrix} b_1 & \cdots & b_m \end{bmatrix}$$

n x m — n x k — k x m — 1 x m

Note that we add the bias to each row, essentially simulating an n x m matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Example on the next page.

Model multiple inputs and multiple outputs

$$y = xw + b$$

n x m — n x k — k x m — 1 x m

Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

We can extend the model to multiple inputs where n, k, m > 1.

$$\begin{bmatrix} y_{11} & \cdots & y_{1m} \\ y_{21} & \cdots & y_{2m} \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ y_{n1} & \cdots & y_{nm} \end{bmatrix} = \begin{bmatrix} x_{11}w_{11} + x_{12}w_{21} + \ldots + x_{1k}w_{k1} + b_1 & \cdots & x_{11}w_{1m} + x_{12}w_{2m} + \ldots + x_{1k}w_{km} + b_m \\ x_{21}w_{11} + x_{22}w_{21} + \ldots + x_{2k}w_{k1} + b_1 & \cdots & x_{21}w_{1m} + x_{22}w_{2m} + \ldots + x_{2k}w_{km} + b_m \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ x_{n1}w_{11} + x_{n2}w_{21} + \ldots + x_{nk}w_{k1} + b_1 & \cdots & x_{n1}w_{1m} + x_{n2}w_{2m} + \ldots + x_{nk}w_{km} + b_m \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ x_{n1} & x_{n2} & \cdots & x_{nk} \end{bmatrix} \begin{bmatrix} w_{11} & \cdots & w_{1m} \\ w_{21} & \cdots & w_{2m} \\ \cdots & \cdots & \cdots \\ w_{k1} & \cdots & w_{km} \end{bmatrix} + \begin{bmatrix} b_1 & \cdots & b_m \end{bmatrix}$$

n x m — n x k — k x m — 1 x m

Note that we add the bias to each row, essentially simulating an n x m matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Objective function

The objective function is a measure of how well our model's outputs match the targets.

The **targets** are the "correct values" which we aim at. In the cats and dogs example, the targets were the "labels" we assigned to each photo (either "cat" or "dog").

Objective functions can be split into two types: **loss** (supervised learning) and **reward** (reinforcement learning). Our focus is supervised learning.

Common loss functions

L2-norm loss
/regression/

$$\sum_i (y_i - t_i)^2$$

The L2-norm of a vector, **a**, (Euclidean length) is given by

$$\|a\| = \sqrt{a^T \cdot a} = \sqrt{a_1^2 + \cdots + a_n^2}$$

The main rationale is that the L2-norm loss is basically the distance from the origin (0). So, the closer to the origin is the difference of the outputs and the targets, the lower the loss, and the better the prediction.

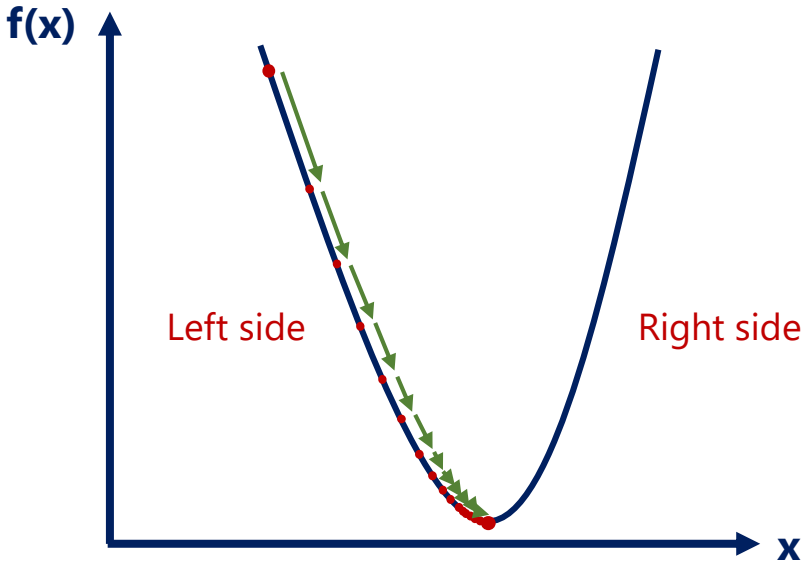
Cross-entropy loss
/classification/

$$-\sum_i \boldsymbol{t}_i \ln \boldsymbol{y}_i$$

The cross-entropy loss is mainly used for classification. Entropy comes from information theory, and measures how much information one is missing to answer a question. Cross-entropy (used in ML) works with probabilities – one is our opinion, the other – the true probability (the probability of a target to be correct is 1 by definition). If the cross-entropy is 0, then we are **not missing any information** and have a perfect model.

Gradient descent

The last ingredient is the optimization algorithm. The most commonly used one is the gradient descent. The main point is that we can find the minimum of a function by applying the rule: $x_{i+1} = x_i - \eta f'(x_i)$, where η is a small enough positive number. In machine learning, η, is called the learning rate. The rationale is that the first derivative at x_i , $f'(x_i)$ shows the slope of the function at x_i.



If the first derivative of f(x) at x_i , $f'(x_i)$, is negative, then we are on the left side of the parabola (as shown in the figure). Subtracting a negative number from x_i (as η is positive), will result in x_{i+1}, that is bigger than x_i. This would cause our next *trial* to be on the right; thus, closer to the sought minimum.

Alternatively, if the first derivative is positive, then we are on the right side of the parabola. Subtracting a positive number from x_i will result in a lower number, so our next trial will be to the left (again closer to the minimum).

So, either way, using this rule, we are approaching the minimum. When the first derivative is 0, we have reached the minimum. Of course, our update rule won't update anymore ($x_{i+1} = x_i - 0$).

The learning rate η, must be low enough so we don't oscillate (bounce around without reaching the minimum) and big enough, so we reach it in rational time.

In machine learning, f(x) is the **loss function**, which we are trying to minimize.

The variables that we are varying until we find the minimum are the **weights and the biases.** The proper update rules are:

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \eta \nabla_{\mathbf{w}} L(\boldsymbol{w}_i) = \boldsymbol{w}_i - \eta \sum_i \boldsymbol{x}_i \delta_i \quad \text{and} \quad b_{i+1} = b_i - \eta \nabla_b L(b_i) = b_i - \eta \sum_i \delta_i$$

Gradient descent. Multivariate derivation

The multivariate generalization of the gradient descent concept: $x_{i+1} = x_i - \eta f'(x_i)$ is given by: $\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \eta \nabla_{\mathbf{w}} L(\boldsymbol{w}_i)$ (using this example, as we will need it). In this new equation, w is a matrix and we are interested in the gradient of w, w.r.t. the loss function. As promised in the lecture, we will show the derivation of the gradient descent formulas for the L2-norm loss divided by 2.

Model: $y = xw + b$

Loss: $L = \frac{1}{2}\sum_i (y_i - t_i)^2$

Update rule: $\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \eta \nabla_{\mathbf{w}} L(\boldsymbol{w}_i)$

(opt. algorithm) $b_{i+1} = b_i - \eta \nabla_b L(b_i)$

Analogically, we find the update rule for the biases.

Please note that the division by 2 that we performed does not change the nature of the loss.

ANY function that holds the basic property of being higher for worse results and lower for better results can be a loss function.

$$\nabla_{\mathbf{w}} L = \nabla_{\mathbf{w}} \frac{1}{2} \sum_i (y_i - t_i)^2 =$$

$$= \nabla_{\mathbf{w}} \frac{1}{2} \sum_i \big((\boldsymbol{x}_i \boldsymbol{w} + b) - t_i\big)^2 =$$

$$= \sum_i \nabla_{\mathbf{w}} \frac{1}{2} (\boldsymbol{x}_i \boldsymbol{w} + b - t_i)^2 =$$

$$= \sum_i \boldsymbol{x}_i (\boldsymbol{x}_i \boldsymbol{w} + b - t_i) =$$

$$= \sum_i \boldsymbol{x}_i (y_i - t_i) \equiv$$

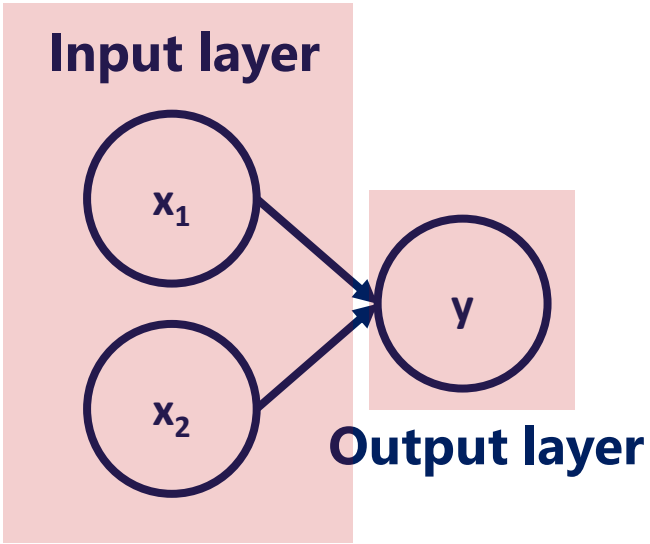
$$\equiv \sum_i \boldsymbol{x}_i \delta_i$$

IT’S TIME TO DIG DEEPER

Layers

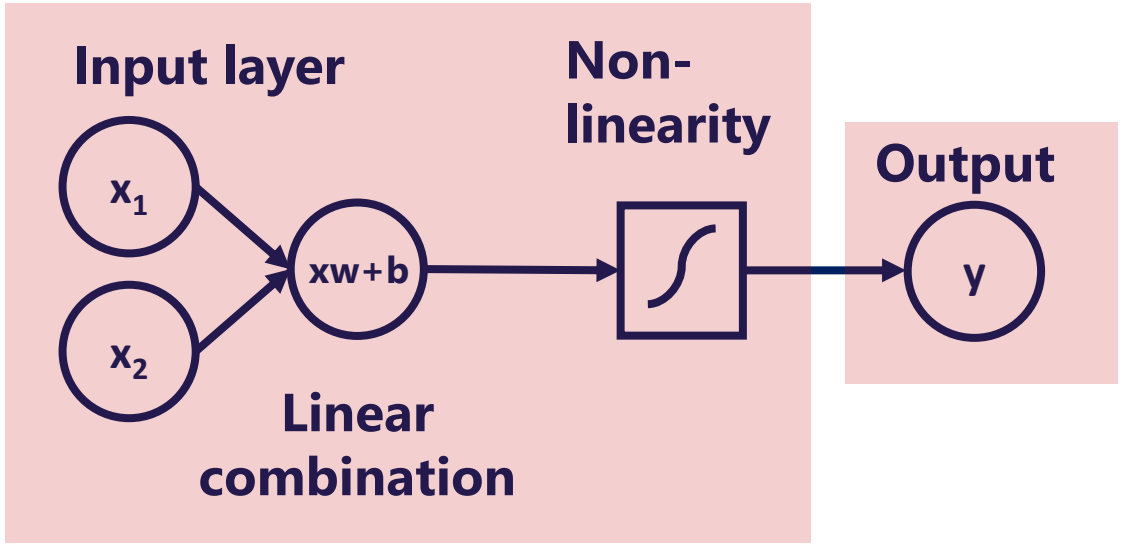
An initial linear combination and the added non-linearity form a **layer**. The layer is the building block of neural networks.

Minimal example (a simple neural network)



In the minimal example we trained a *neural network* which had no depth. There were solely an input layer and an output layer. Moreover, the output was simply **a linear combination** of the input.

Neural networks

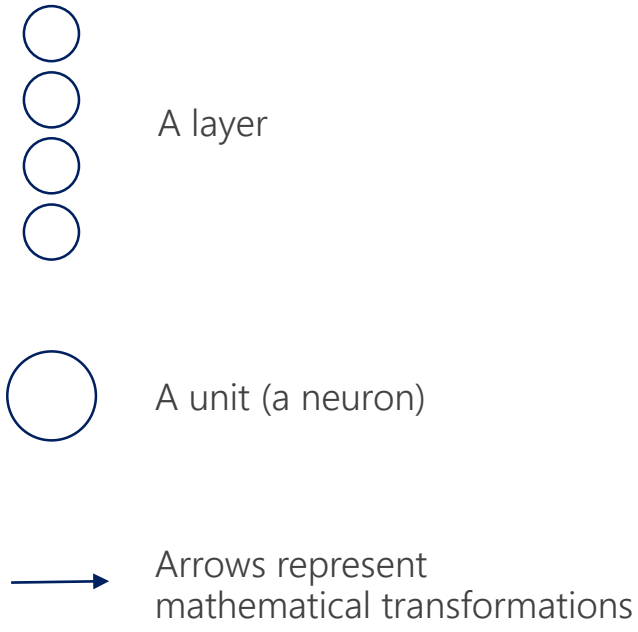


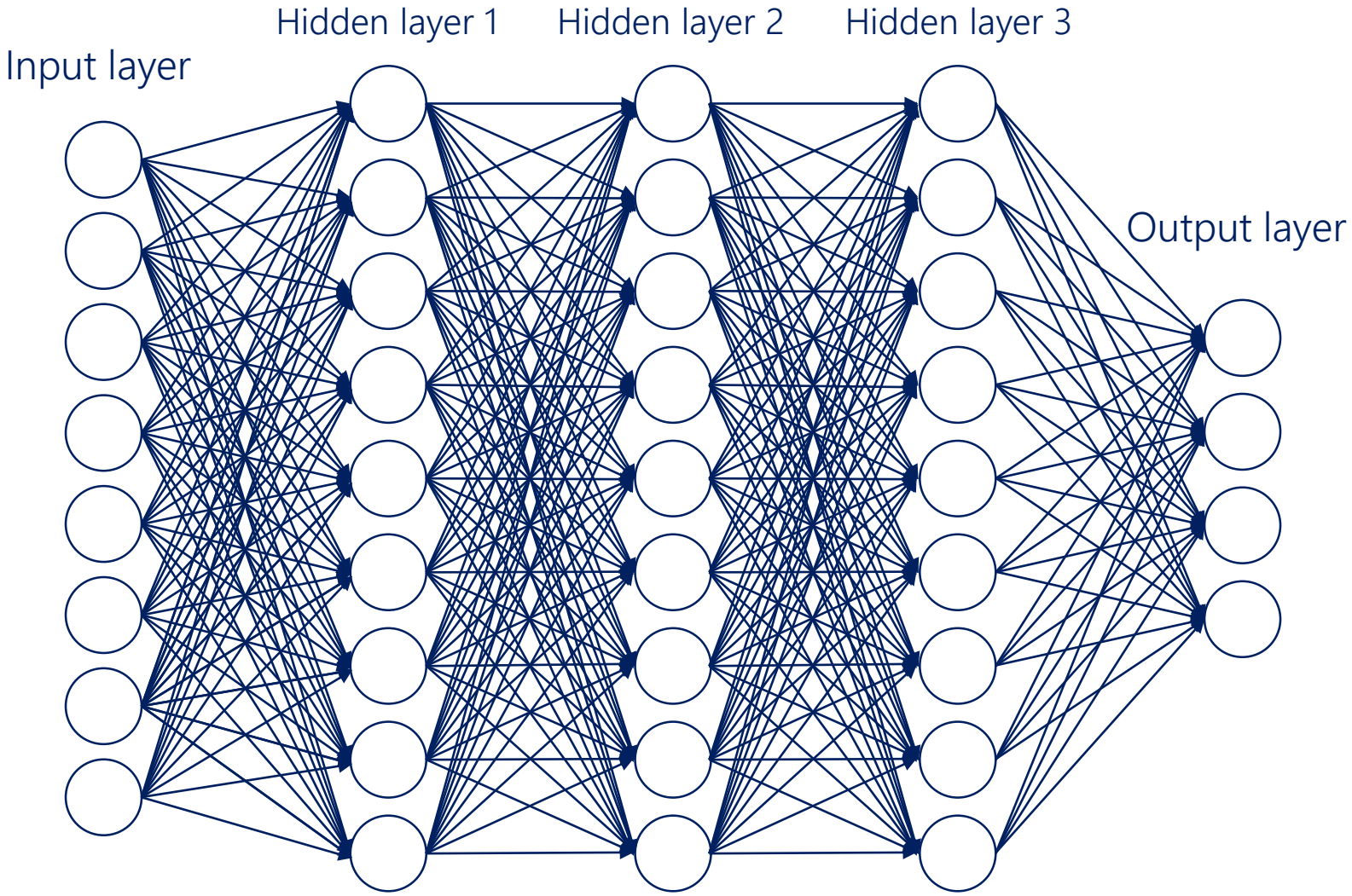
Neural networks step on linear combinations, but add a non-linearity to each one of them. Mixing linear combinations and non-linearities allows us to model arbitrary functions.

A deep net

This is a deep neural network (deep net) with 5 layers.

How to read this diagram:





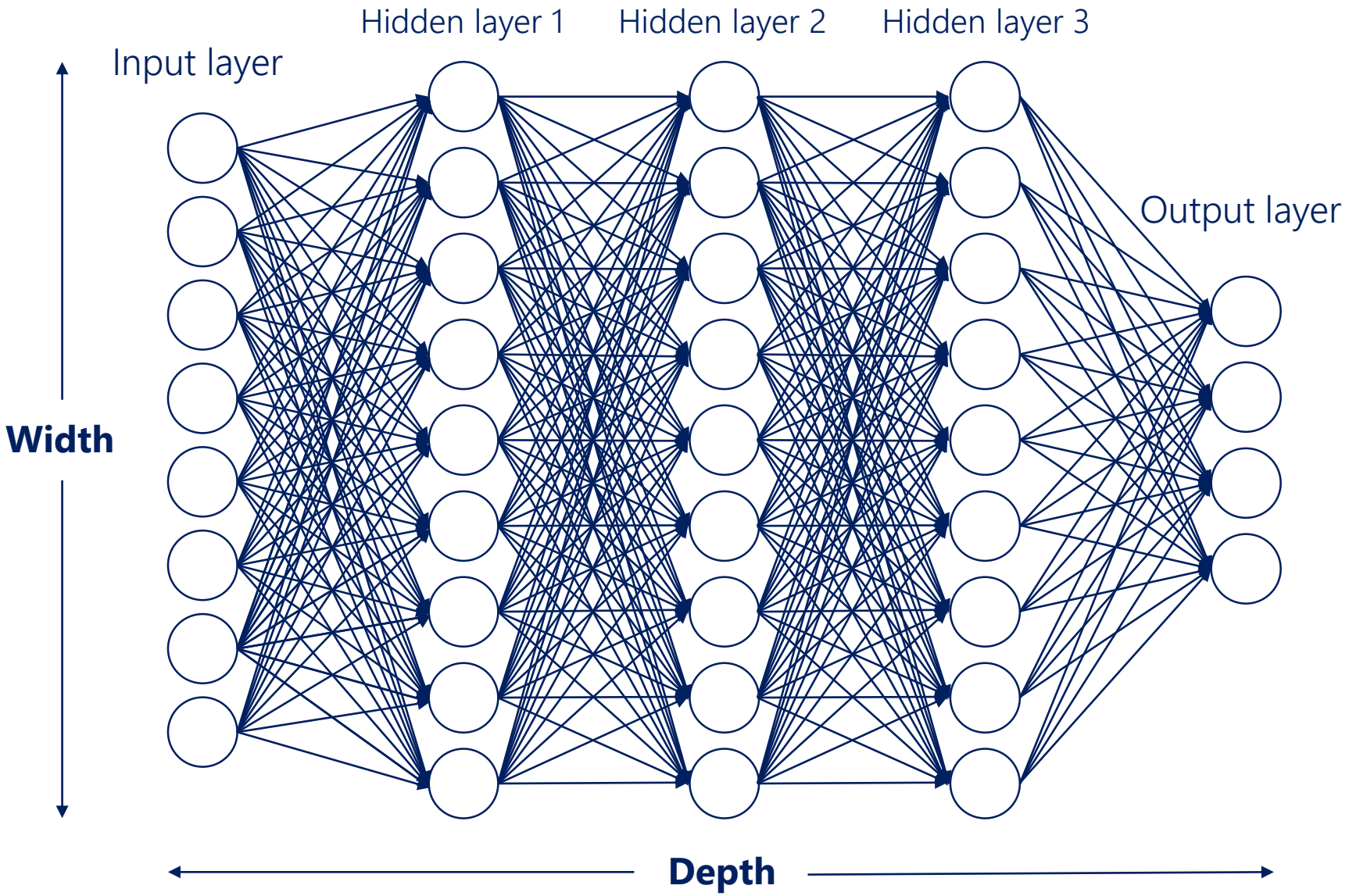
A deep net

The **width** of a layer is the number of units in that layer

The **width** of the net is the number of units of the biggest layer

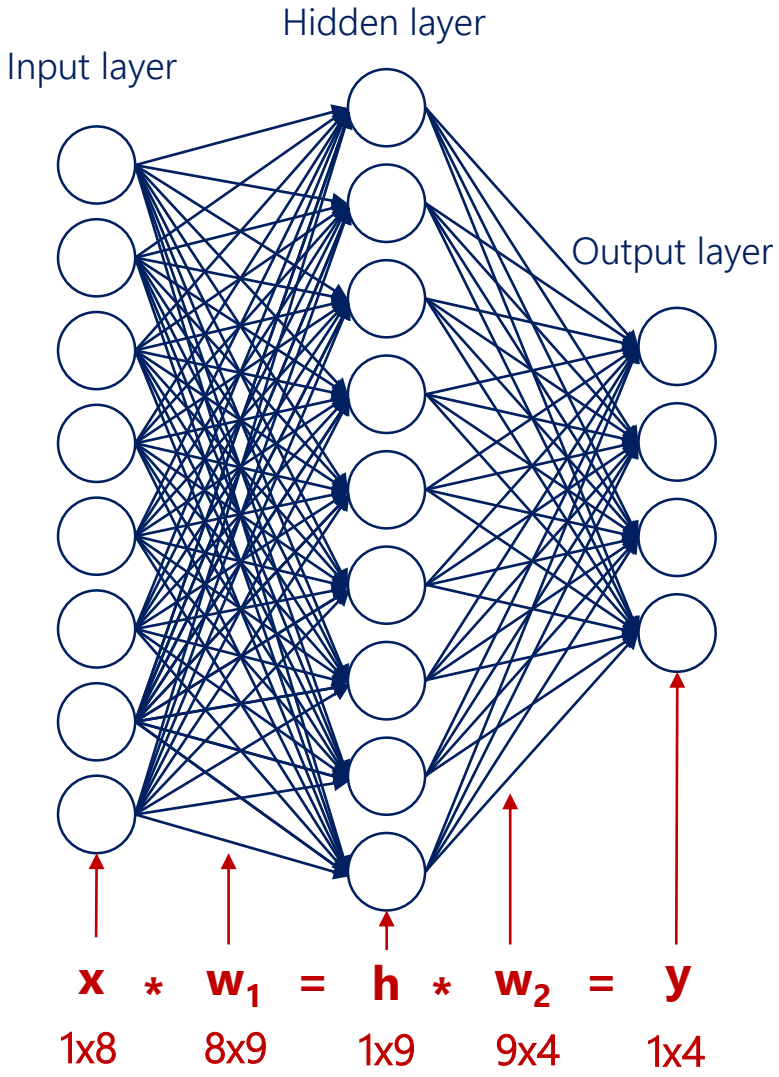
The **depth** of the net is equal to the number of layers or the number of hidden layers. The term has different definitions. More often than not, we are interested in the number of hidden layers (as there are always input and output layers).

The width and the depth of the net are called **hyperparameters.** They are values we manually chose when creating the net.



Why we need non-linearities to stack layers

You can see a net with no non-linearities: just linear combinations.



$$h = x * w_1$$

$$y = h * w_2$$

$$y = x * w_1 * w_2$$

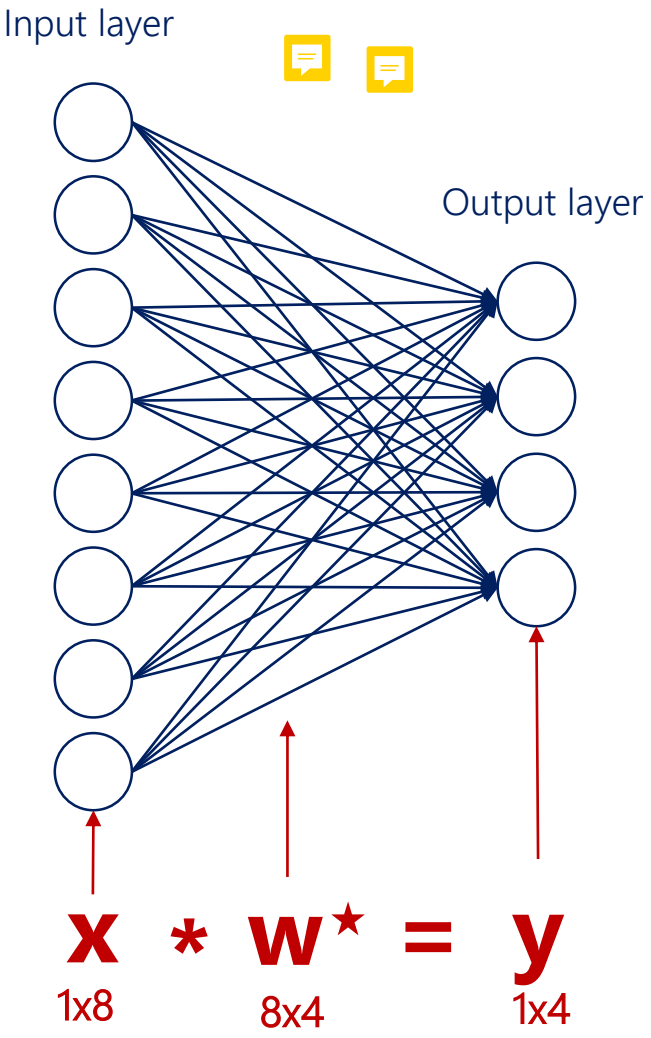
(w_1: 8x9, w_2: 9x4)

$$y = x * w^\star$$

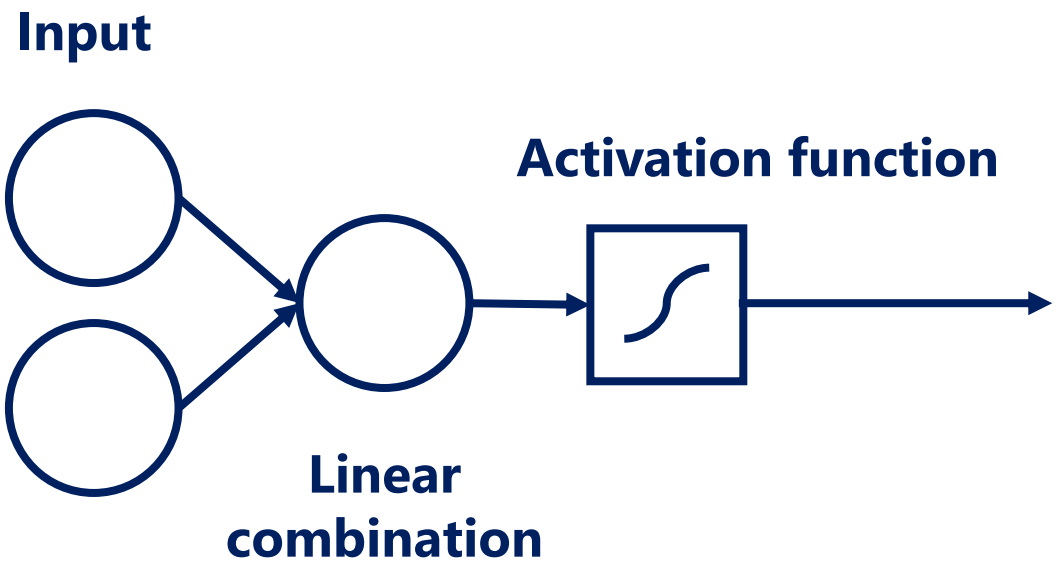
($w^\star$: 8x4)

Two consecutive linear transformations are equivalent to a single one.

Two consecutive linear transformations are equivalent to a single one.



Activation functions



In the respective lesson, we gave an example of temperature change. The temperature starts decreasing (which is a numerical change). Our brain is a kind of an 'activation function'. It tells us whether it is **cold enough** for us to put on a jacket.

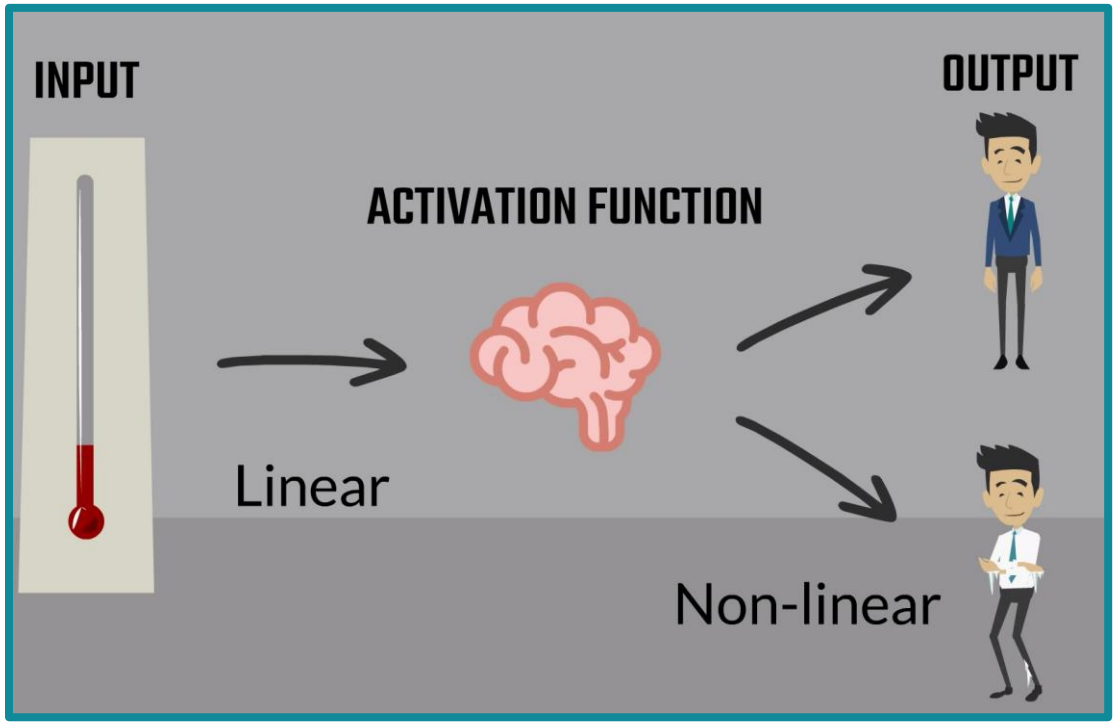
Putting on a jacket is a binary action: 0 (no jacket) or 1 (jacket).

This is a very intuitive and visual (yet not so practical) example of how activation functions work.

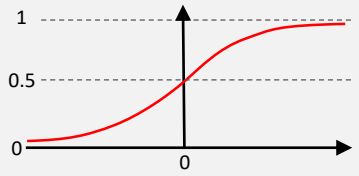
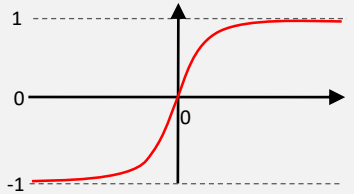
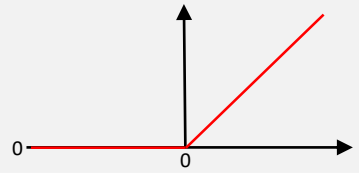
Activation functions (non-linearities) are needed so we can break the linearity and represent more complicated relationships.

Moreover, activation functions are required in order to **stack layers.**

Activation functions transform inputs into outputs of a different kind.

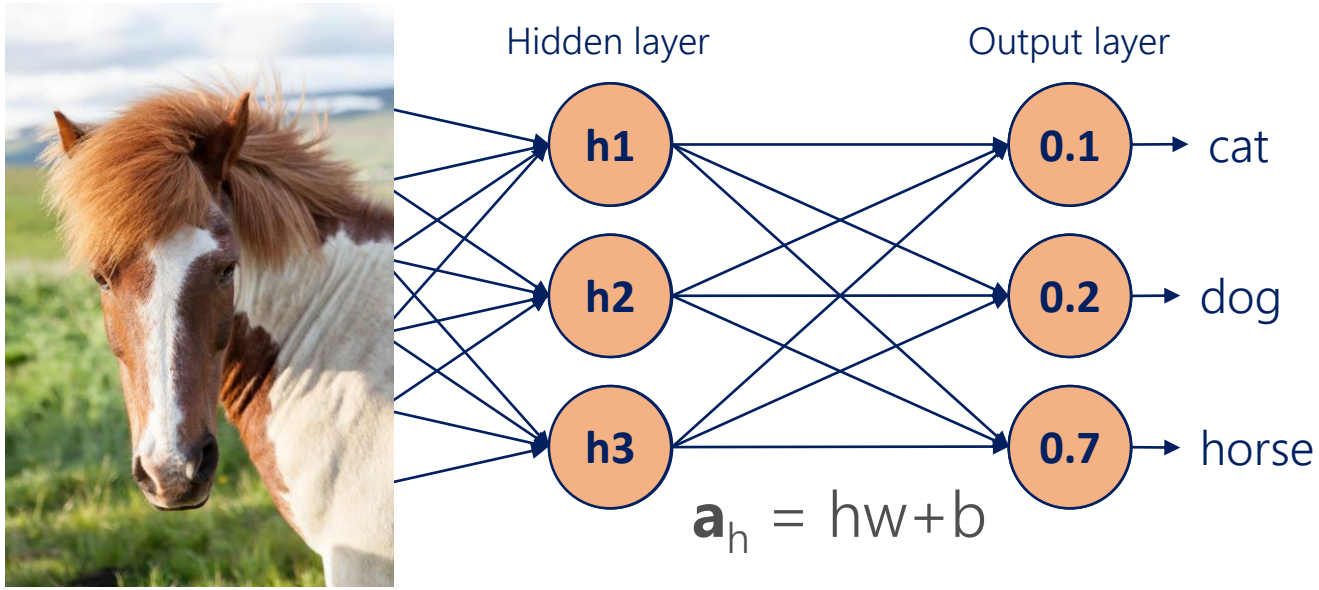


Common activation functions

Name	Formula	Derivative	Graph	Range
sigmoid (logistic function)	$\sigma(a) = \frac{1}{1+e^{-a}}$	$\frac{\partial\sigma(a)}{\partial a} = \sigma(a)(1-\sigma(a))$		(0,1)
TanH (hyperbolic tangent)	$\tanh(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$	$\frac{\partial \tanh(a)}{\partial a} = \frac{4}{(e^{a}+e^{-a})^2}$		(-1,1)
ReLu (rectified linear unit)	relu(a) = max(0,a)	$\frac{\partial \text{ relu}(a)}{\partial a} = \begin{cases} 0, if\ a \le 0 \\ 1, if\ a > 0 \end{cases}$		(0,∞)
softmax	$\sigma_i(\boldsymbol{a}) = \frac{e^{a_i}}{\sum_j e^{a_j}}$	$\frac{\partial\sigma_i(\boldsymbol{a})}{\partial a_j} = \sigma_i(\boldsymbol{a})\left(\delta_{ij} - \sigma_j(\boldsymbol{a})\right)$ Where δ_{ij} is 1 if i=j, 0 otherwise		(0,1)

All common activation functions are: **monotonic, continuous,** and **differentiable**. These are important properties needed for the optimization.

Softmax activation



The softmax activation transforms a bunch of arbitrarily large or small numbers into a valid probability distribution.

While other activation functions get an input value and transform it, regardless of the other elements, the softmax considers the information about the **whole set of numbers** we have.

The values that softmax outputs are in the range from 0 to 1 and their sum is exactly 1 (like probabilities).

Example:

$\mathbf{a}$ = [-0.21,0.47,1.72]

$$\text{softmax}(\mathbf{a}) = \frac{e^{a_i}}{\sum_j e^{a_j}}$$

$$\sum_j e^{a_j} = e^{-0.21} + e^{0.47} + e^{1.72} = 8$$

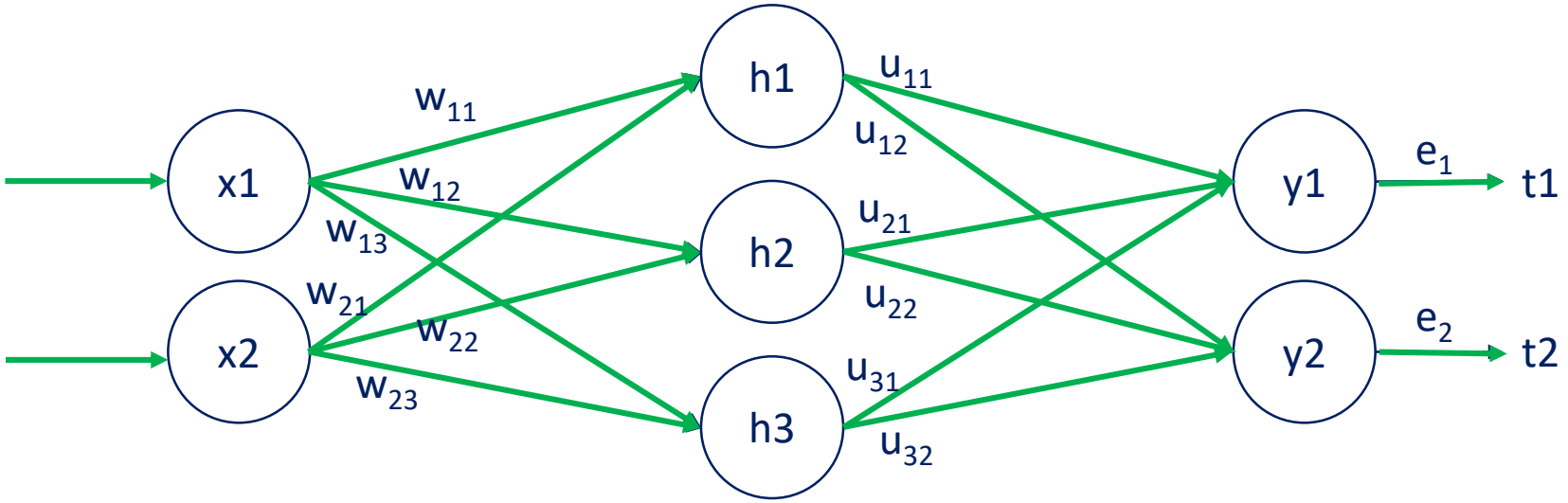
$$\text{softmax}(\mathbf{a}) = \left[\frac{e^{-0.21}}{8}, \frac{e^{0.47}}{8}, \frac{e^{1.72}}{8}\right]$$

y = [0.1,0.2,0.7] → probability distribution

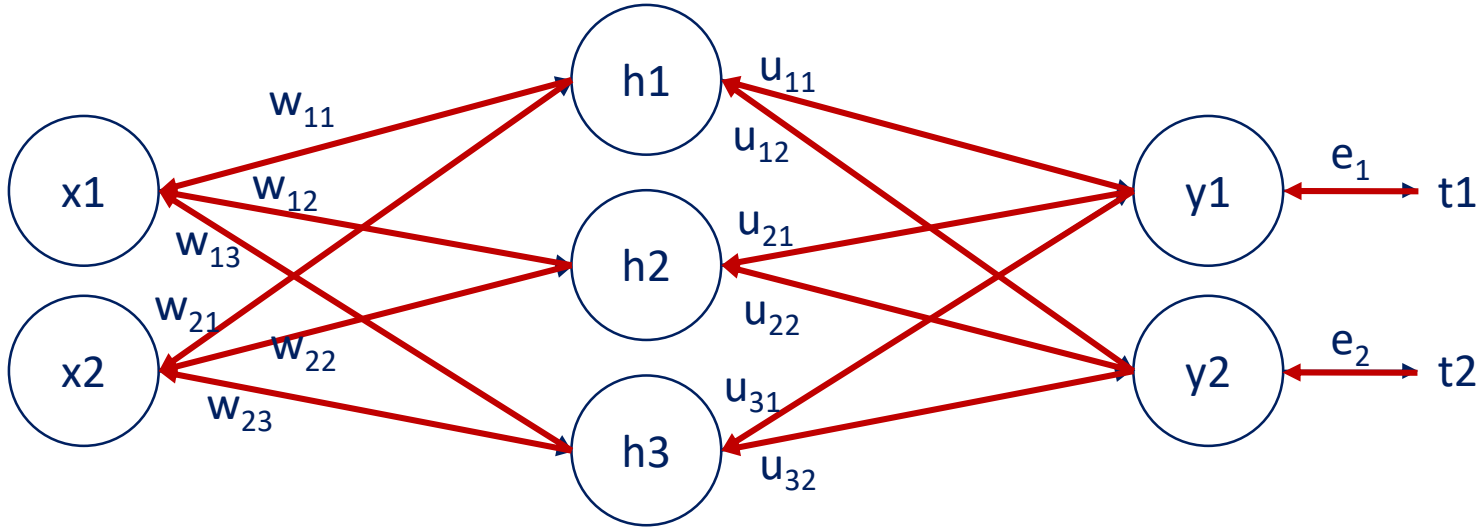
The property of the softmax to output probabilities is so useful and intuitive that it is often used as the activation function for the **final (output) layer.**

However, when the softmax is used prior to that (as the activation of a hidden layer), the results are not as satisfactory. That's because a lot of the information about the variability of the data is lost.

Backpropagation

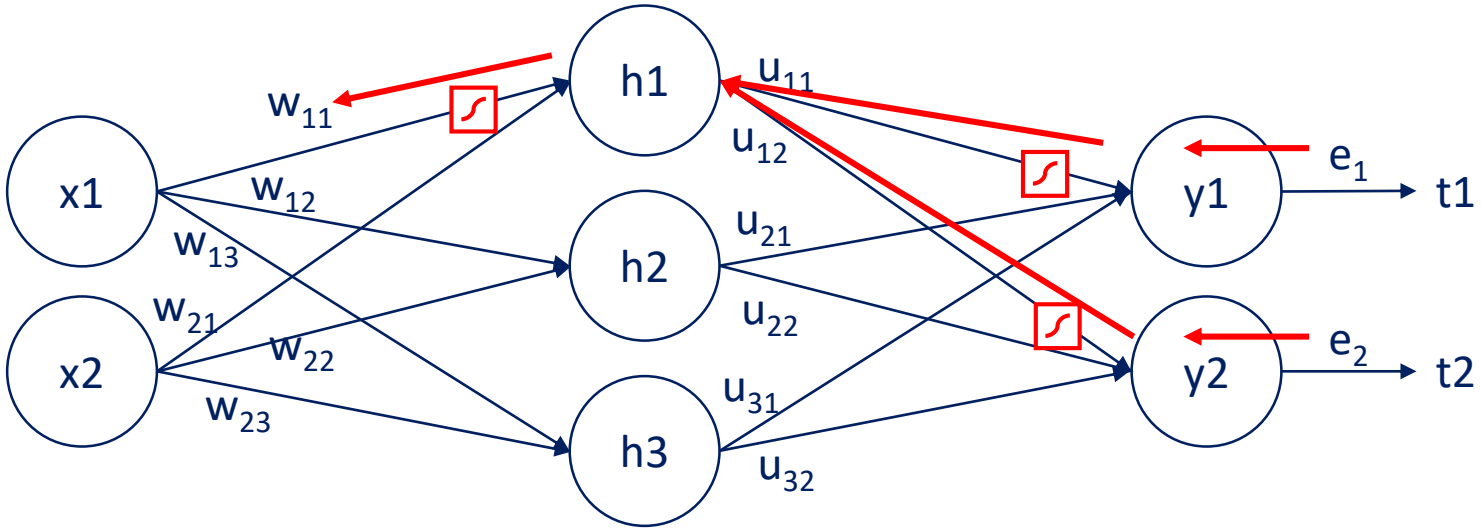


Forward propagation is the process of pushing inputs through the net. At the end of each epoch, the obtained outputs are compared to targets to form the errors.



Backpropagation of errors is an **algorithm** for neural networks using gradient descent. It consists of calculating the contribution of each **parameter** to the errors. We backpropagate the **errors** through the net and **update** the parameters (weights and biases) accordingly.

Backpropagation formula



$$\frac{\partial L}{\partial w_{ij}} = \delta_j x_i \text{ , where } \delta_j = \sum_k \delta_k w_{jk}\, y_j(1 - y_j)$$

If you want to examine the full derivation, please make use of the PDF we made available in the section: **Backpropagation. A peek into the Mathematics of Optimization.**

Backpropagation. A Peek into the Mathematics of Optimization

1 Motivation

In order to get a truly deep understanding of deep neural networks, one must look at the mathematics of it. As backpropagation is at the core of the optimization process, we wanted to introduce you to it. This is definitely not a necessary part of the course, as in TensorFlow, sk-learn, or any other machine learning package (as opposed to simply NumPy), will have backpropagation methods incorporated.

2 The specific net and notation we will examine

Here's our simple network:

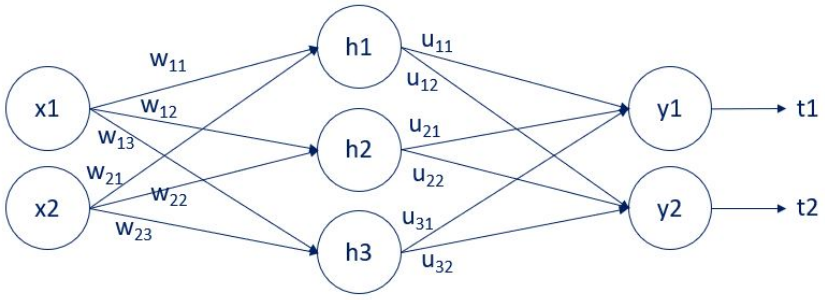


Figure 1: Backpropagation

We have two inputs: x_1 and x_2. There is a single hidden layer with 3 units (nodes): h_1, h_2, and h_3. Finally, there are two outputs: y_1 and y_2. The arrows that connect them are the weights. There are two weights matrices: $\mathbf{w}$, and $\mathbf{u}$. The $\mathbf{w}$ weights connect the input layer and the hidden layer. The $\mathbf{u}$ weights connect the hidden layer and the output layer. We have employed the letters $\mathbf{w}$, and $\mathbf{u}$, so it is easier to follow the computation to follow.
You can also see that we compare the outputs y_1 and y_2 with the targets t_1 and t_2.

There is one last letter we need to introduce before we can get to the computations. Let a be the linear combination prior to activation. Thus, we have: $\mathbf{a}^{(1)} = \mathbf{xw} + \mathbf{b}^{(1)}$ and $\mathbf{a}^{(2)} = \mathbf{hu} + \mathbf{b}^{(2)}$.
Since we cannot exhaust all activation functions and all loss functions, we will focus on two of the most common. A **sigmoid** activation and an **L2-norm loss**.
With this new information and the new notation, the output y is equal to the activated linear combination. Therefore, for the output layer, we have $\mathbf{y} = \sigma(\mathbf{a}^{(2)})$, while for the hidden layer: $\mathbf{h} = \sigma(\mathbf{a}^{(1)})$.
We will examine backpropagation for the output layer and the hidden layer separately, as the methodologies differ.

3 Useful formulas

I would like to remind you that:

$$\text{L2-norm loss: } L = \frac{1}{2}\sum_i (y_i - t_i)^2$$

The sigmoid function is:

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

and its derivative is:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

4 Backpropagation for the output layer

In order to obtain the update rule:

$$\mathbf{u} \leftarrow \mathbf{u} - \eta \nabla_{\mathbf{u}} L(\mathbf{u})$$

we must calculate

$$\nabla_{\mathbf{u}} L(\mathbf{u})$$

Let's take a single weight u_{ij}. The partial derivative of the loss w.r.t. u_{ij} equals:

$$\frac{\partial L}{\partial u_{ij}} = \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial a_j^{(2)}} \frac{\partial a_j^{(2)}}{\partial u_{ij}}$$

where i corresponds to the previous layer (input layer for this transformation) and j corresponds to the next layer (output layer of the transformation). The partial derivatives were computed simply following the chain rule.

$$\frac{\partial L}{\partial y_j} = (y_j - t_j)$$

following the L2-norm loss derivative.

$$\frac{\partial y_j}{\partial a_j^{(2)}} = \sigma(a_j^{(2)})(1 - \sigma(a_j^{(2)})) = y_j(1 - y_j)$$

following the sigmoid derivative.
Finally, the third partial derivative is simply the derivative of $\mathbf{a}^{(2)} = \mathbf{hu} + \mathbf{b}^{(2)}$. So,

$$\frac{\partial a_j^{(2)}}{\partial u_{ij}} = h_i$$

Replacing the partial derivatives in the expression above, we get:

$$\frac{\partial L}{\partial u_{ij}} = \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial a_j^{(2)}}\frac{\partial a_j^{(2)}}{\partial u_{ij}} = (y_j - t_j)y_j(1 - y_j)h_i = \delta_j h_i$$

Therefore, the update rule for a single weight for the output layer is given by:

$$u_{ij} \leftarrow u_{ij} - \eta\delta_j h_i$$

5 Backpropagation of a hidden layer

Similarly to the backpropagation of the output layer, the update rule for a single weight, w_{ij} would depend on:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial h_j}\frac{\partial h_j}{\partial a_j^{(1)}}\frac{\partial a_j^{(1)}}{\partial w_{ij}}$$

following the chain rule.
Taking advantage of the results we have so far for transformation using the sigmoid activation and the linear model, we get:

$$\frac{\partial h_j}{\partial a_j^{(1)}} = \sigma(a_j^{(1)})(1 - \sigma(a_j^{(1)})) = h_j(1 - h_j)$$

and

$$\frac{\partial a_j^{(1)}}{\partial w_{ij}} = x_i$$

The actual problem for backpropagation comes from the term $\frac{\partial L}{\partial h_j}$. That's due to the fact that there is no "hidden" target. You can follow the solution for weight w_{11} below. It is advisable to also check Figure 1, while going through the computations.

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial a_1^{(2)}}\frac{\partial a_1^{(2)}}{\partial h_1} + \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial a_2^{(2)}}\frac{\partial a_2^{(2)}}{\partial h_1} =$$
$$= (y_1 - t_1)y_1(1 - y_1)u_{11} + (y_2 - t_2)y_2(1 - y_2)u_{12}$$

From here, we can calculate $\frac{\partial L}{\partial w_{11}}$, which was what we wanted. The final expression is:

$$\frac{\partial L}{\partial w_{11}} = \left[(y_1 - t_1)y_1(1 - y_1)u_{11} + (y_2 - t_2)y_2(1 - y_2)u_{12}\right] h_1(1 - h_1)x_1$$

The generalized form of this equation is:

$$\frac{\partial L}{\partial w_{ij}} = \sum_k (y_k - t_k)y_k(1 - y_k)u_{jk}h_j(1 - h_j)x_i$$

6 Backpropagation generalization

Using the results for backpropagation for the output layer and the hidden layer, we can put them together in one formula, summarizing backpropagation, in the presence of L2-norm loss and sigmoid activations.

$$\frac{\partial L}{\partial w_{ij}} = \delta_j x_i$$

where for a hidden layer

$$\delta_j = \sum_k \delta_k w_{jk} y_j(1 - y_j)$$

Kudos to those of you who got to the end.

Thanks for reading.

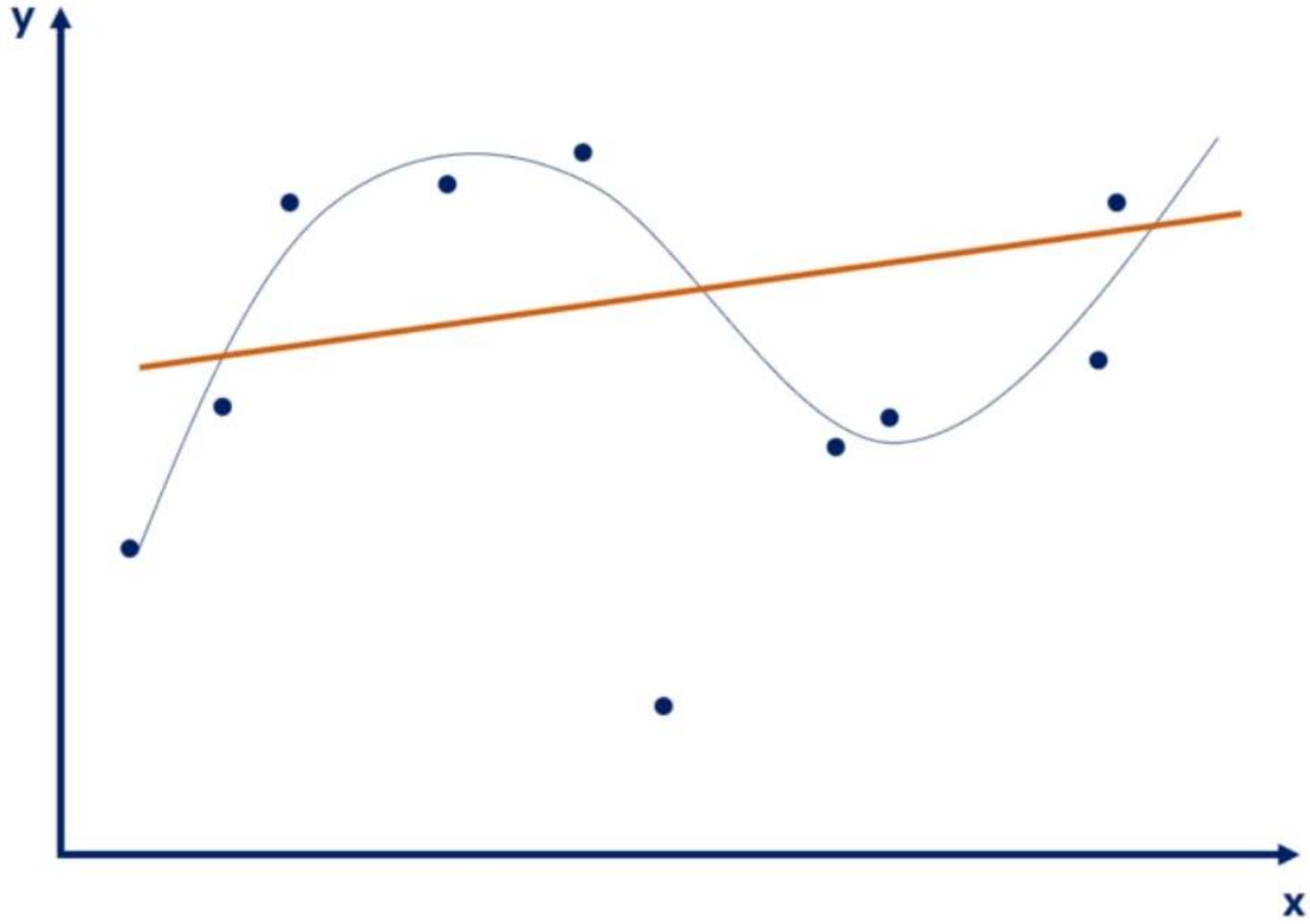
UNDERFITTING

The model has not captured the underlying logic of the data

OVERFITTING

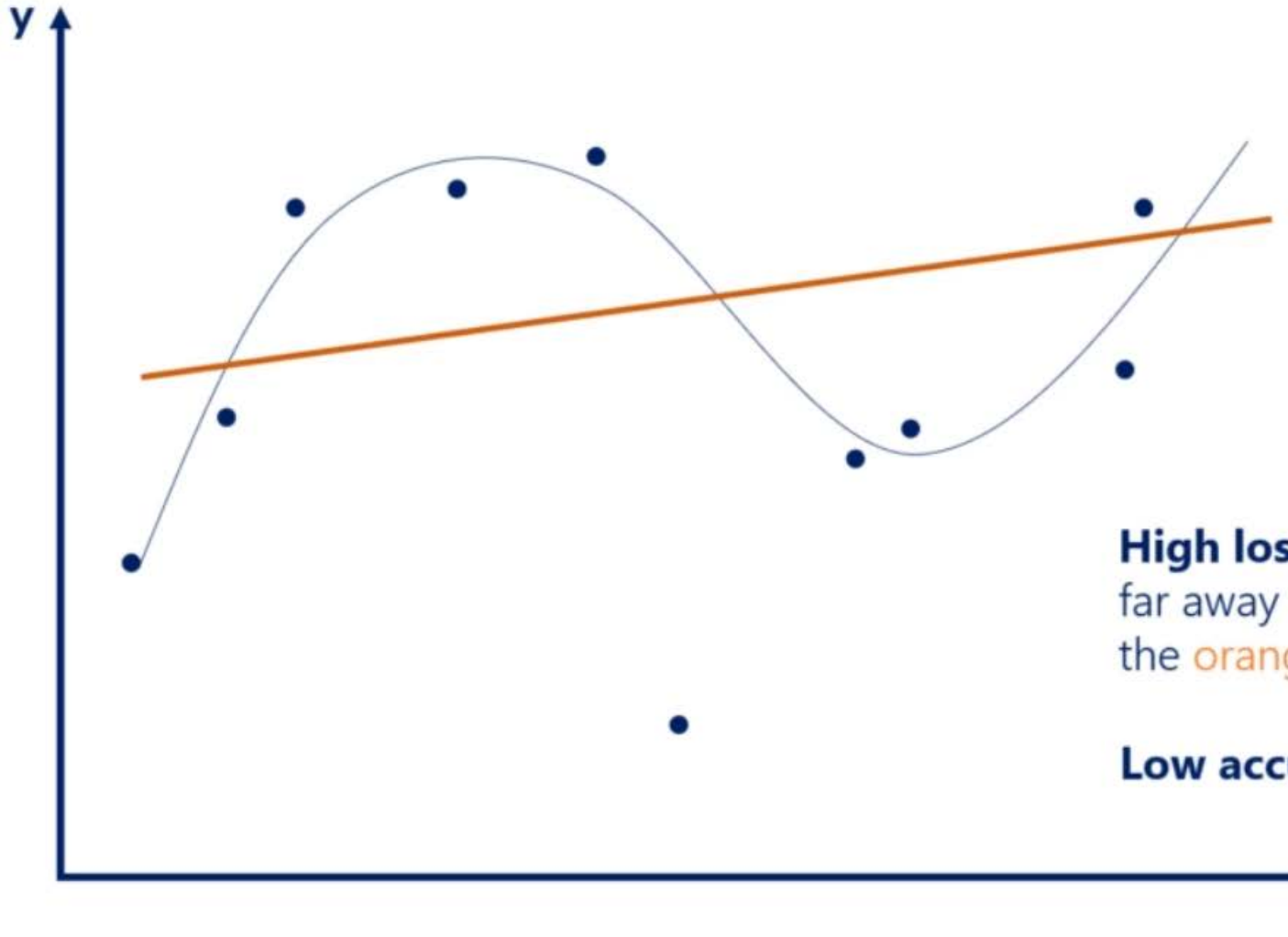
Our training has focused on the particular training set so much, it has "missed the point"

Underfitting and overfitting



Underfitted model: provides an answer, but does not capture the underlying logic

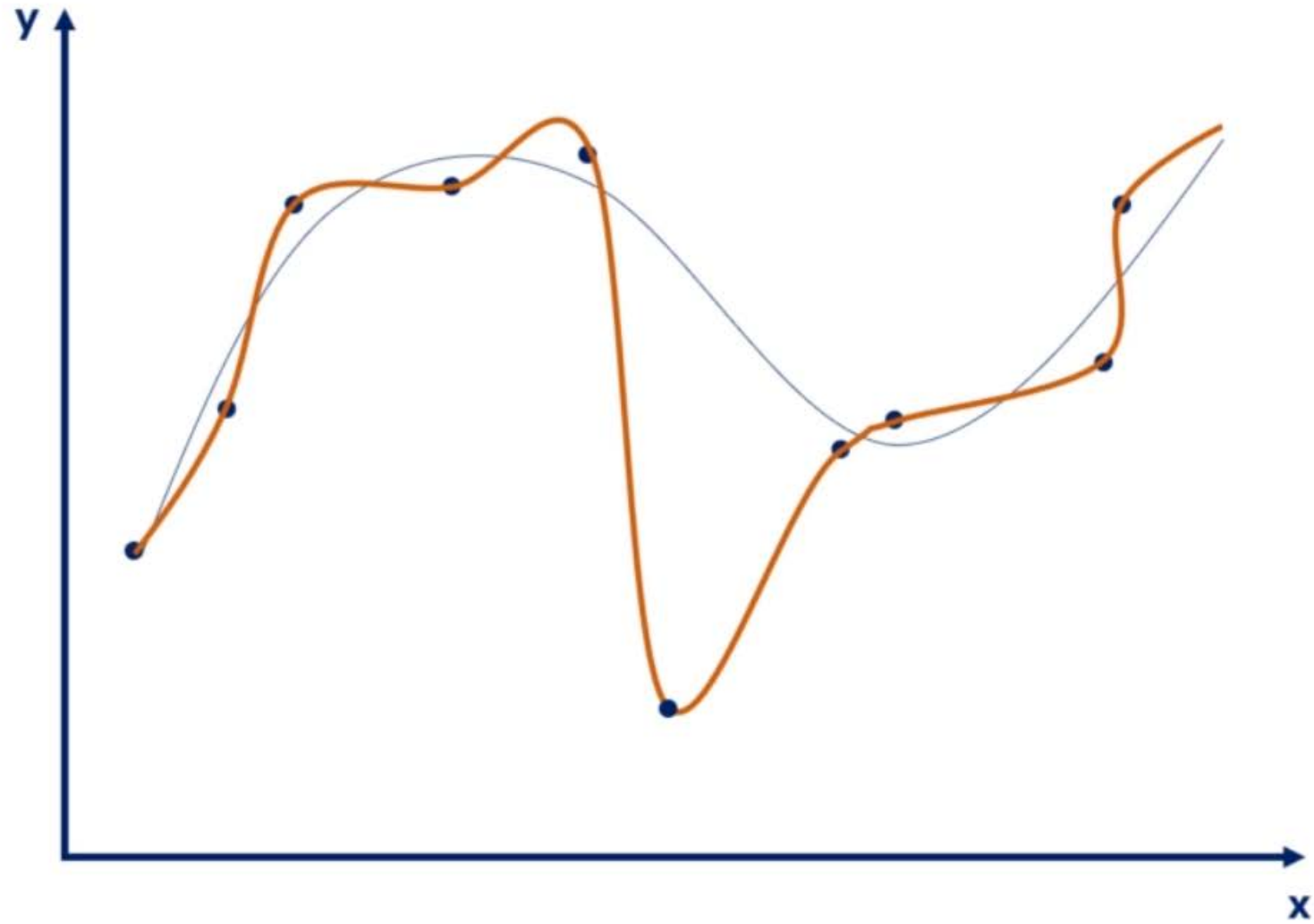
Underfitting and overfitting



High loss (targets, or points, are far away from the predictions, y, or the orange line)

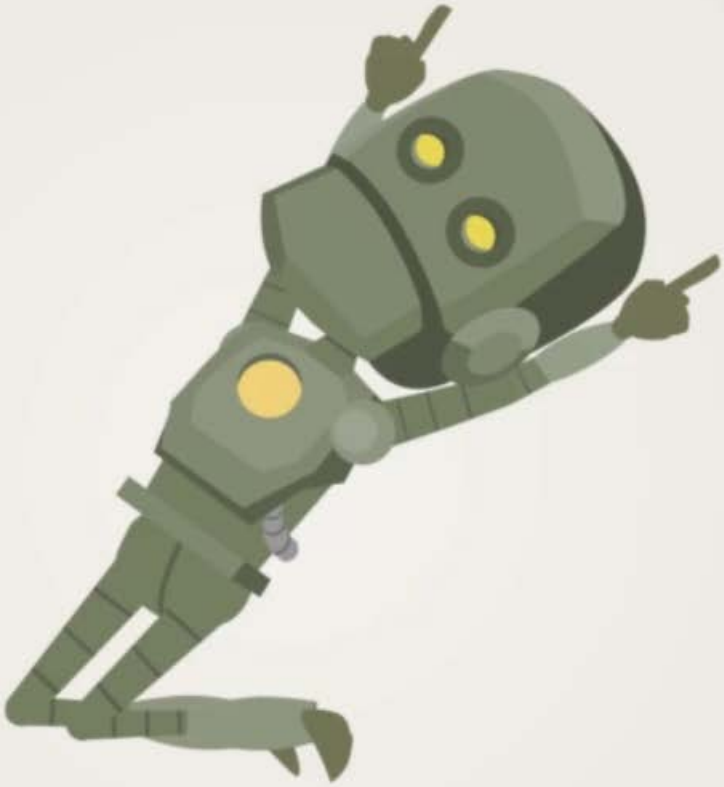
Low accuracy

Underfitting and overfitting



Overfitted model: so super good at modeling the training data that it "misses the point"

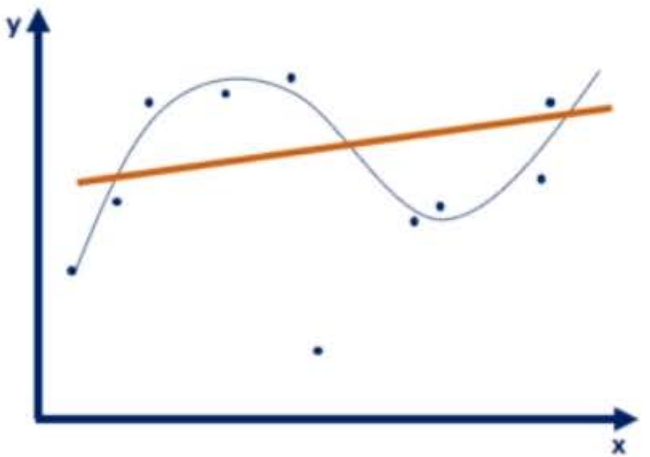
FIRST RULE OF PROGRAMMING



THE COMPUTER IS NEVER WRONG.
IT IS US, WHO MADE A MISTAKE

Underfitting and overfitting

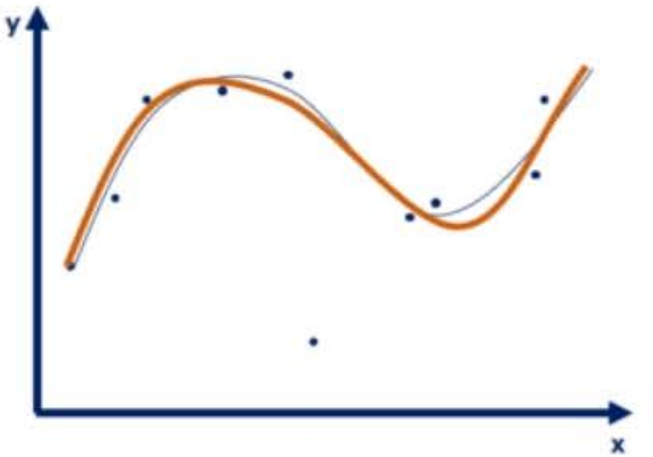
An **underfitted** model



Doesn't capture any logic

- High loss
- Low accuracy

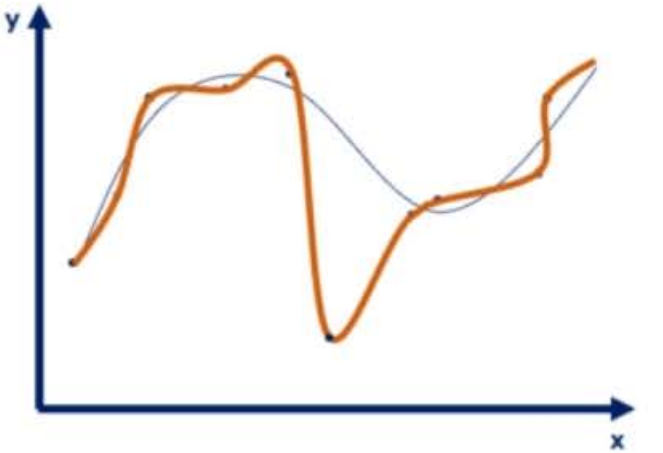
A **good** model



Captures the underlying logic of the dataset

- Low loss
- High accuracy

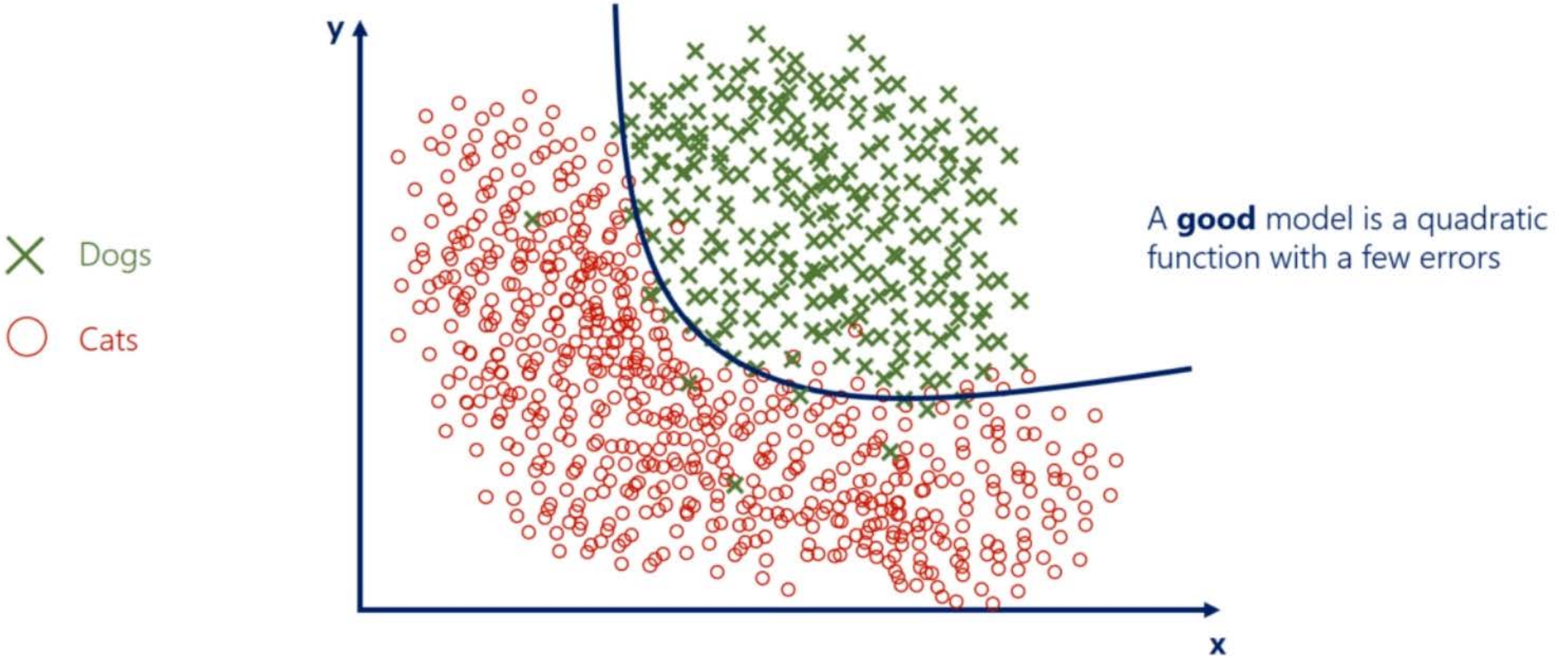
An **overfitted** model



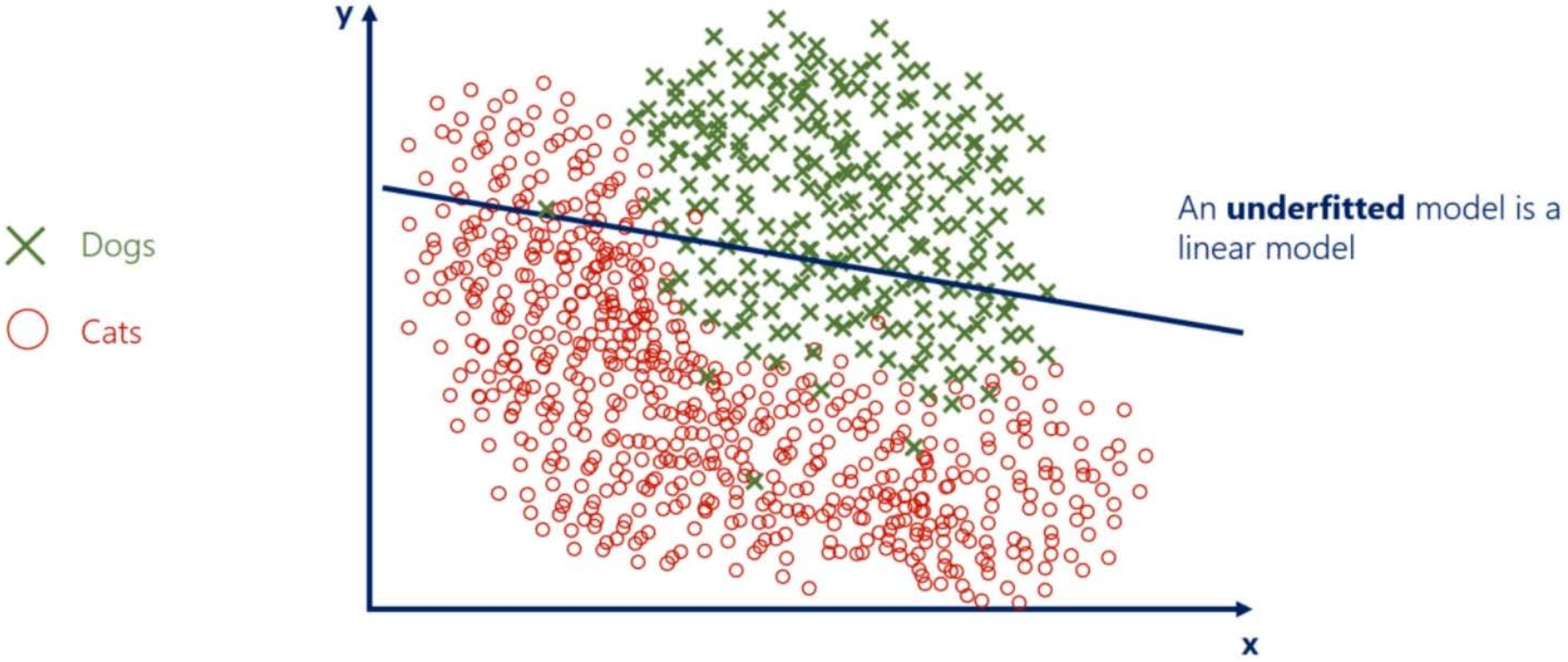
Captures all the noise, thus "missed the point"

- Low loss
- Low accuracy

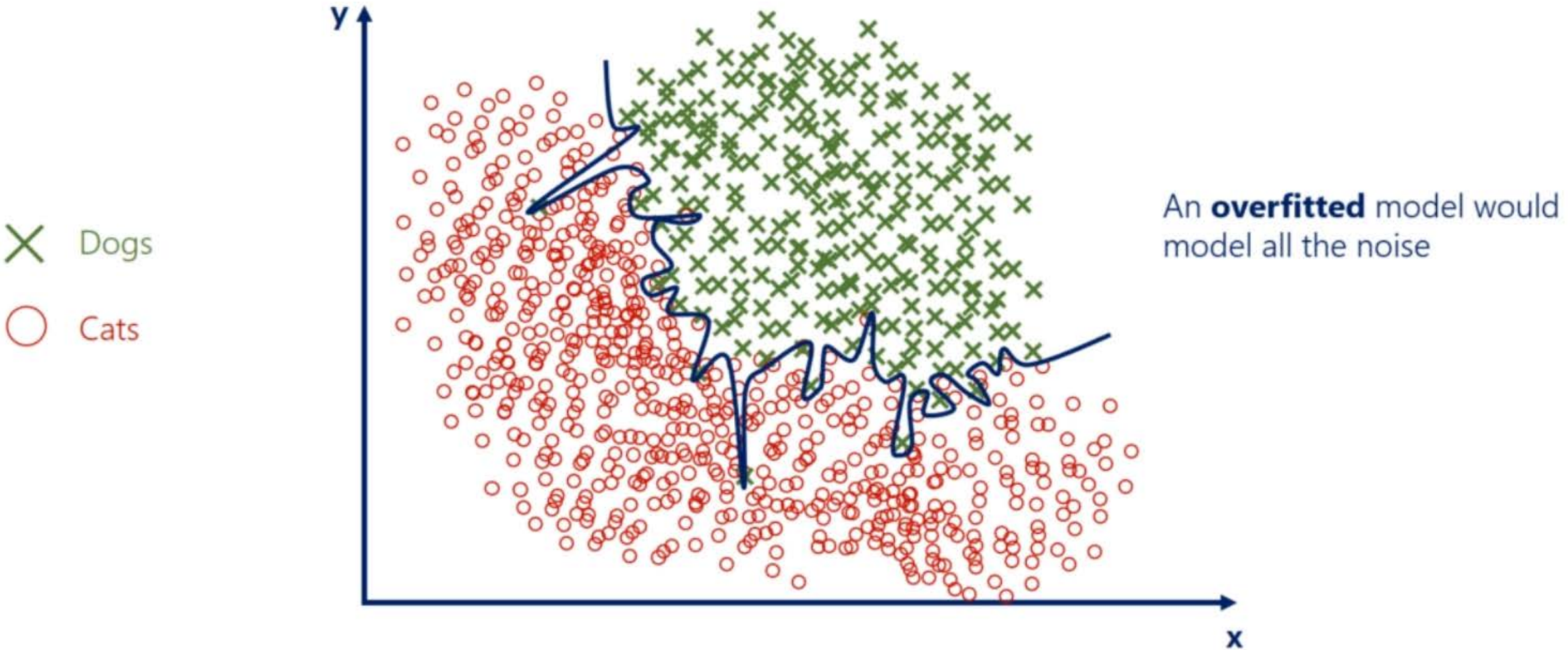
Underfitting and overfitting. A classification example



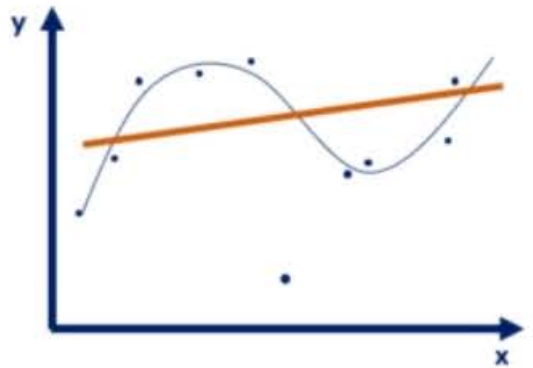
Underfitting and overfitting. A classification example

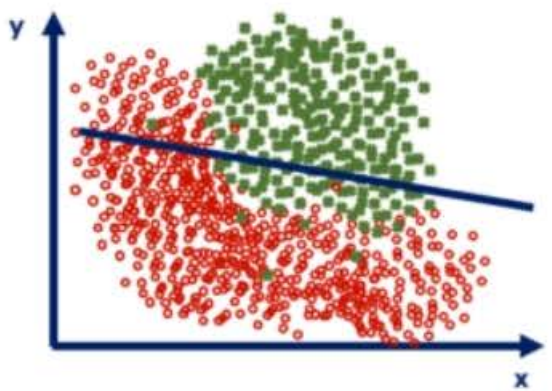


Underfitting and overfitting. A classification example



An **underfitted** model

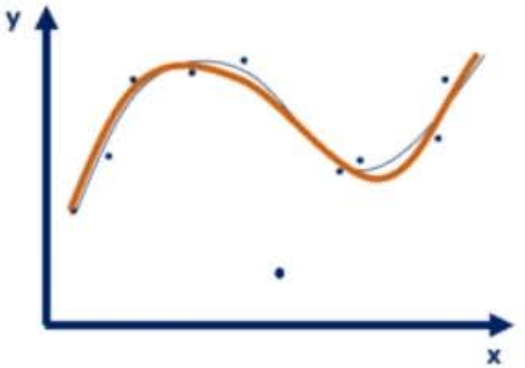


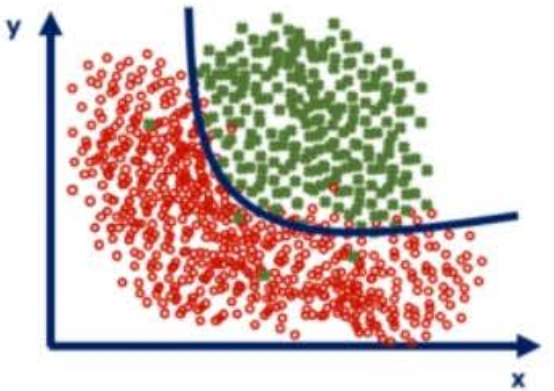


Doesn't capture any logic

- High loss
- Low accuracy

A **good** model

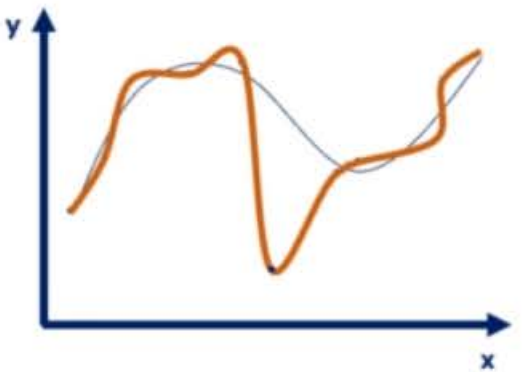


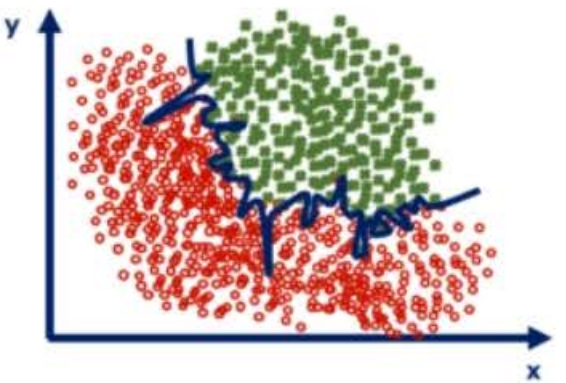


Captures the underlying logic of the dataset

- Low loss
- High accuracy

An **overfitted** model

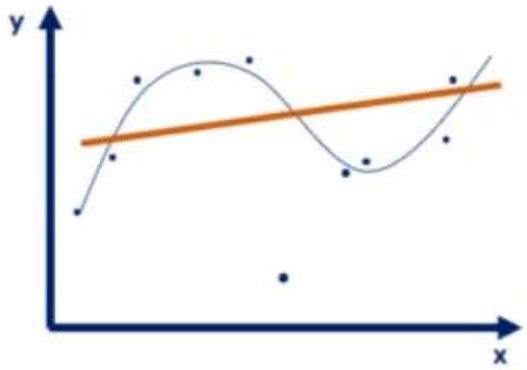


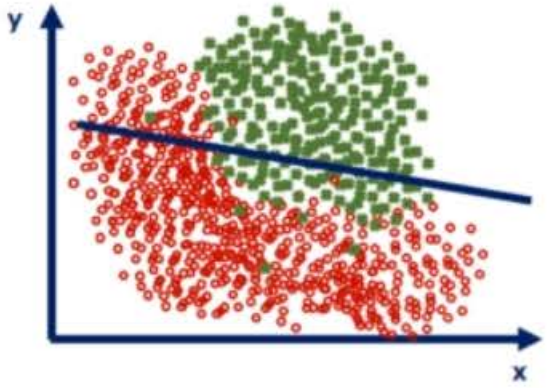


Captures all the noise, thus "missed the point"

- Low loss
- Low accuracy

An **underfitted** model

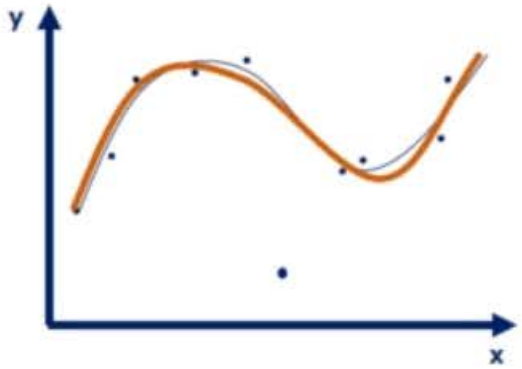


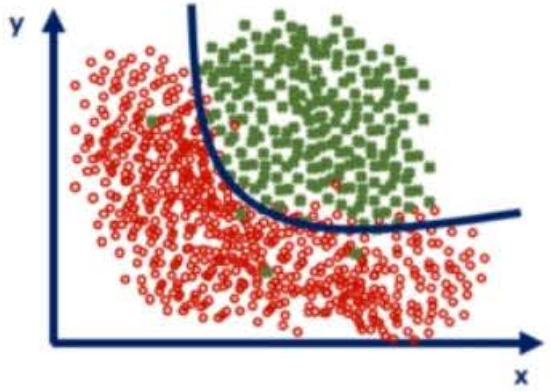


Doesn't capture any logic

- High loss
- Low accuracy

A **good** model

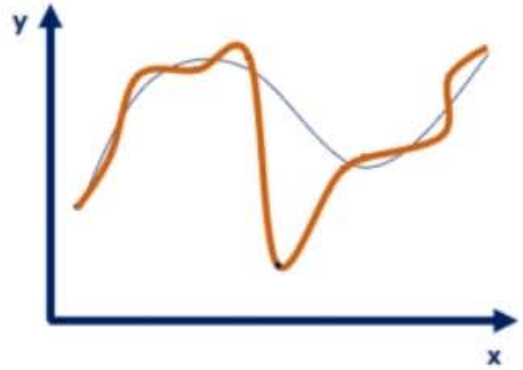


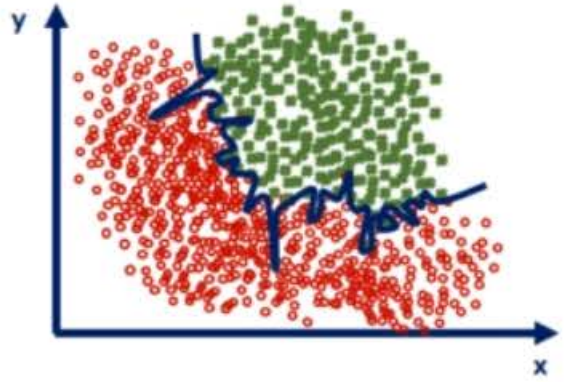


Captures the underlying logic of the dataset

- Low loss
- High accuracy

An **overfitted** model



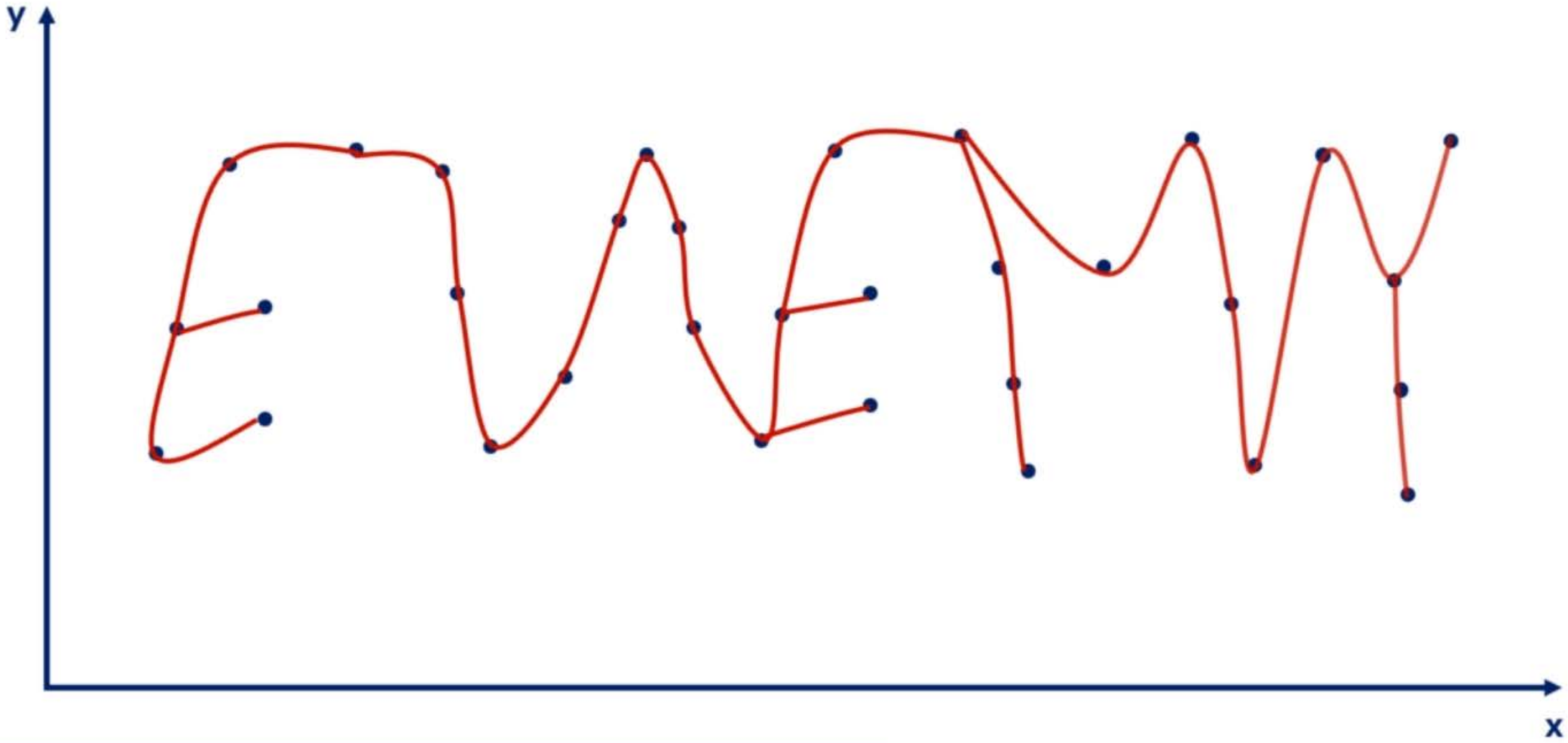


Captures all the noise, thus "missed the point"

- Low loss
- Low accuracy

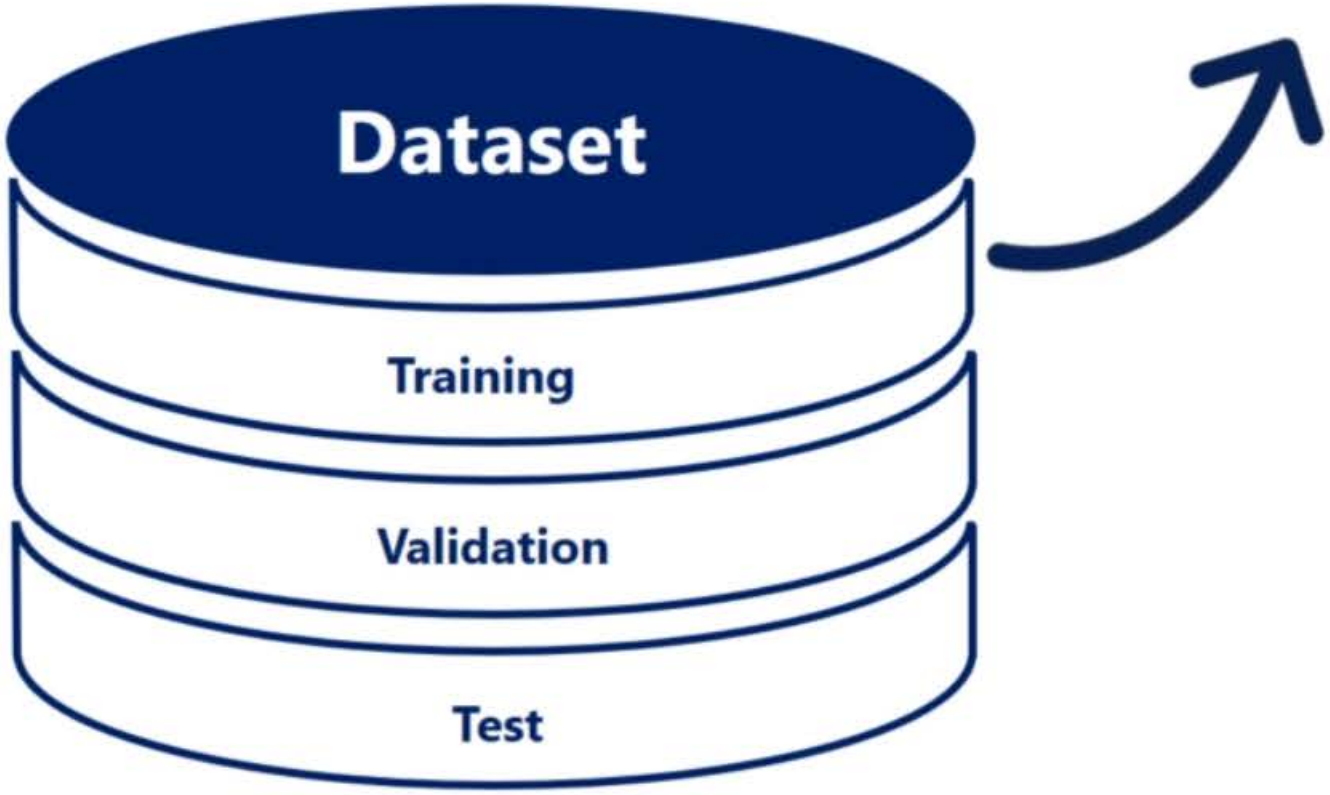
Bias-variance tradeoff: The balance between underfitting and overfitting

Overfitted model



Overfitting is the real enemy when it comes to machine learning

Training, validation, and test

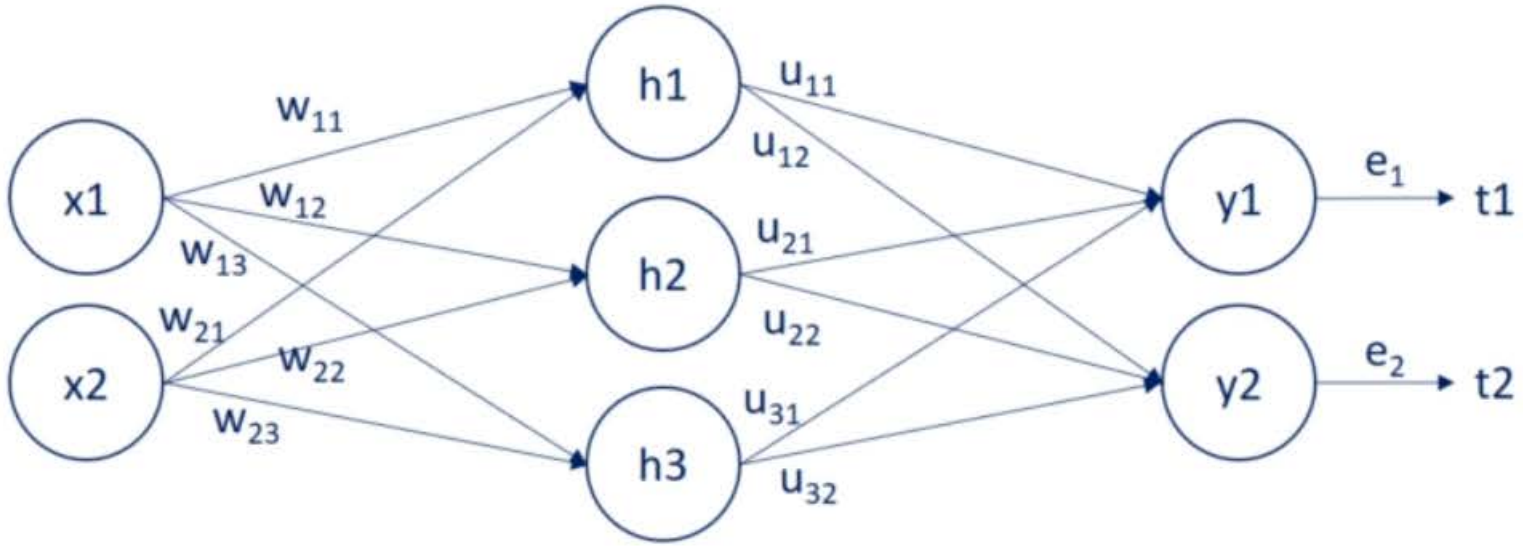


That's where the training happens

```
training_data = np.load('TF_intro.npz')
```

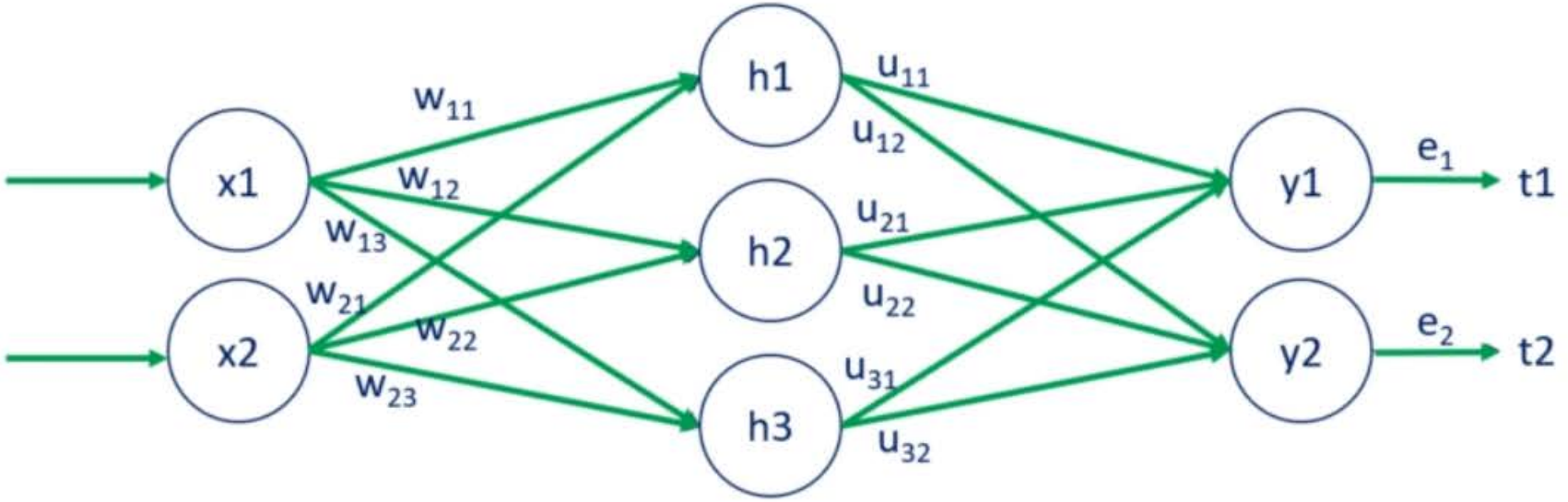
Nothing is new here, since so far we thought all data is training data

Training



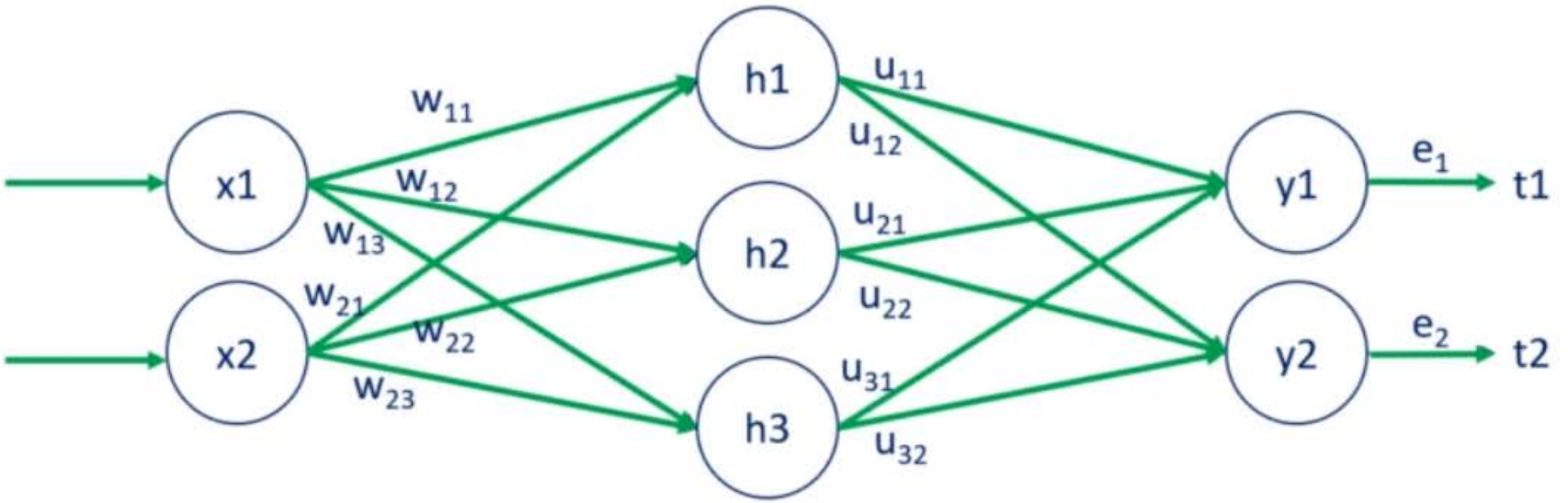
We update the weights and biases for the training set **only**

Validation



We run the model on the **validation dataset,** we only propagate **forward**

Validation



We just calculate the **validation loss.** On average it should equal the **training loss**

Training and validation

training_loss

validation_loss

```
Epoch 1. Training loss: 0.341. Validation loss: 0.192
Epoch 2. Training loss: 0.164. Validation loss: 0.143
Epoch 3. Training loss: 0.115. Validation loss: 0.106
Epoch 4. Training loss: 0.086. Validation loss: 0.093
Epoch 5. Training loss: 0.068. Validation loss: 0.087
Epoch 6. Training loss: 0.054. Validation loss: 0.084
Epoch 7. Training loss: 0.042. Validation loss: 0.074
Epoch 8. Training loss: 0.033. Validation loss: 0.077
Epoch 9. Training loss: 0.029. Validation loss: 0.090
Epoch 10. Training loss: 0.022. Validation loss: 0.072
Epoch 11. Training loss: 0.018. Validation loss: 0.074
Epoch 12. Training loss: 0.014. Validation loss: 0.075
Epoch 13. Training loss: 0.011. Validation loss: 0.081
Epoch 14. Training loss: 0.009. Validation loss: 0.085
Epoch 15. Training loss: 0.007. Validation loss: 0.085
Epoch 16. Training loss: 0.009. Validation loss: 0.091
Epoch 17. Training loss: 0.008. Validation loss: 0.099
Epoch 18. Training loss: 0.006. Validation loss: 0.100
Epoch 19. Training loss: 0.005. Validation loss: 0.092
Epoch 20. Training loss: 0.002. Validation loss: 0.084
Epoch 21. Training loss: 0.002. Validation loss: 0.092
Epoch 22. Training loss: 0.016. Validation loss: 0.085
Epoch 23. Training loss: 0.005. Validation loss: 0.094
Epoch 24. Training loss: 0.001. Validation loss: 0.085
Epoch 25. Training loss: 0.001. Validation loss: 0.088
Epoch 26. Training loss: 0.000. Validation loss: 0.087
Epoch 27. Training loss: 0.000. Validation loss: 0.089
Epoch 28. Training loss: 0.000. Validation loss: 0.091
Epoch 29. Training loss: 0.000. Validation loss: 0.092
Training loss: 0.000. Validation loss: 0.092
```

Each subsequent training loss will be lower or equal to the previous one

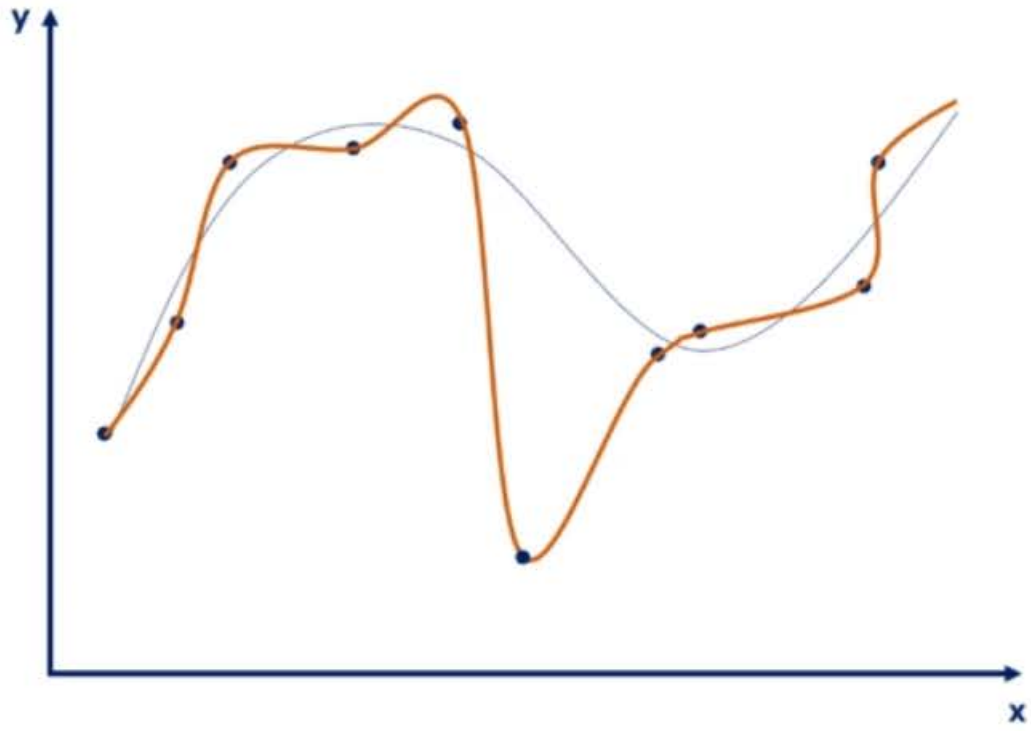
Training and validation

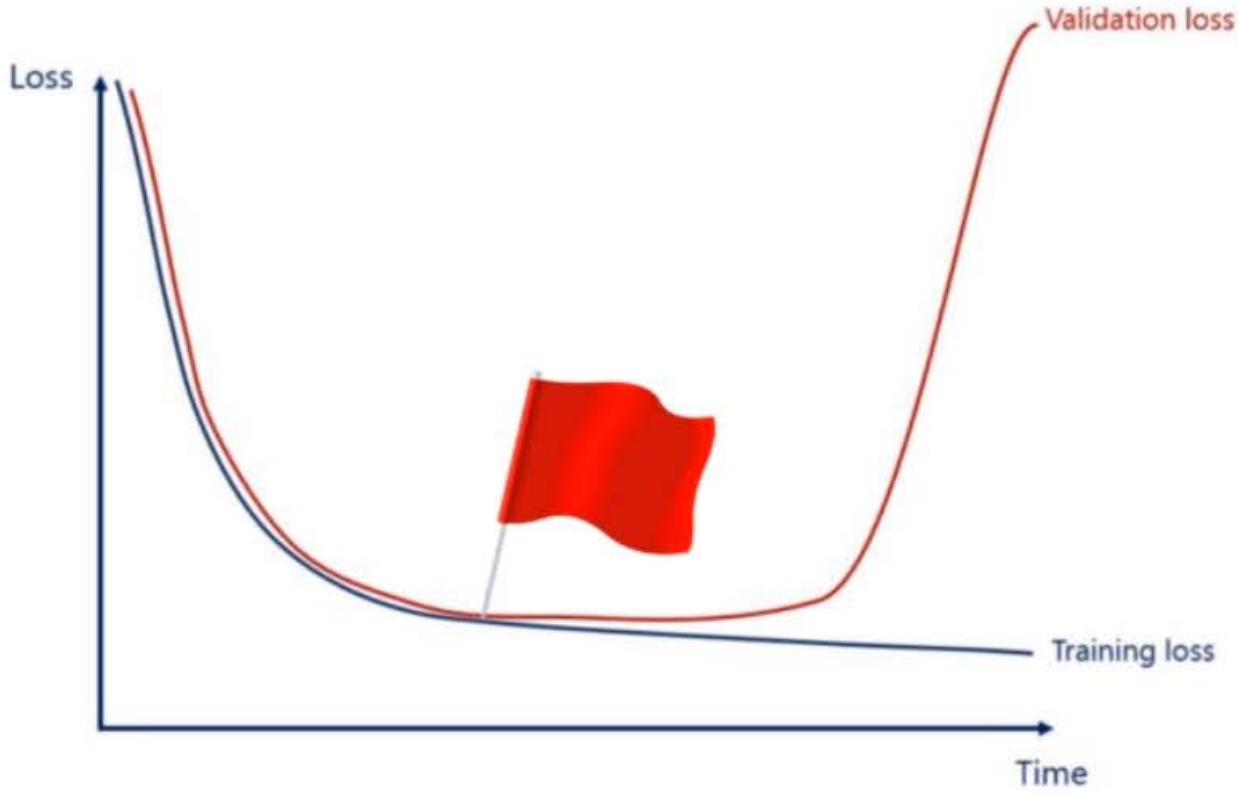
training_loss

validation_loss


```
Epoch 1. Training loss: 0.341. Validation loss: 0.192
Epoch 2. Training loss: 0.164. Validation loss: 0.143
Epoch 3. Training loss: 0.115. Validation loss: 0.106
Epoch 4. Training loss: 0.086. Validation loss: 0.093
Epoch 5. Training loss: 0.068. Validation loss: 0.087
Epoch 6. Training loss: 0.054. Validation loss: 0.084
Epoch 7. Training loss: 0.042. Validation loss: 0.074
Epoch 8. Training loss: 0.033. Validation loss: 0.077
Epoch 9. Training loss: 0.029. Validation loss: 0.090
Epoch 10. Training loss: 0.022. Validation loss: 0.072
Epoch 11. Training loss: 0.018. Validation loss: 0.074
Epoch 12. Training loss: 0.014. Validation loss: 0.075
Epoch 13. Training loss: 0.011. Validation loss: 0.081
Epoch 14. Training loss: 0.009. Validation loss: 0.085
Epoch 15. Training loss: 0.007. Validation loss: 0.085
Epoch 16. Training loss: 0.009. Validation loss: 0.091
Epoch 17. Training loss: 0.008. Validation loss: 0.099
Epoch 18. Training loss: 0.006. Validation loss: 0.100
Epoch 19. Training loss: 0.005. Validation loss: 0.092
Epoch 20. Training loss: 0.002. Validation loss: 0.084
Epoch 21. Training loss: 0.002. Validation loss: 0.092
Epoch 22. Training loss: 0.016. Validation loss: 0.085
Epoch 23. Training loss: 0.005. Validation loss: 0.094
Epoch 24. Training loss: 0.001. Validation loss: 0.085
Epoch 25. Training loss: 0.001. Validation loss: 0.088
Epoch 26. Training loss: 0.000. Validation loss: 0.087
Epoch 27. Training loss: 0.000. Validation loss: 0.089
Epoch 28. Training loss: 0.000. Validation loss: 0.091
Epoch 29. Training loss: 0.000. Validation loss: 0.092
Epoch 30. Training loss: 0.000. Validation loss: 0.092
```

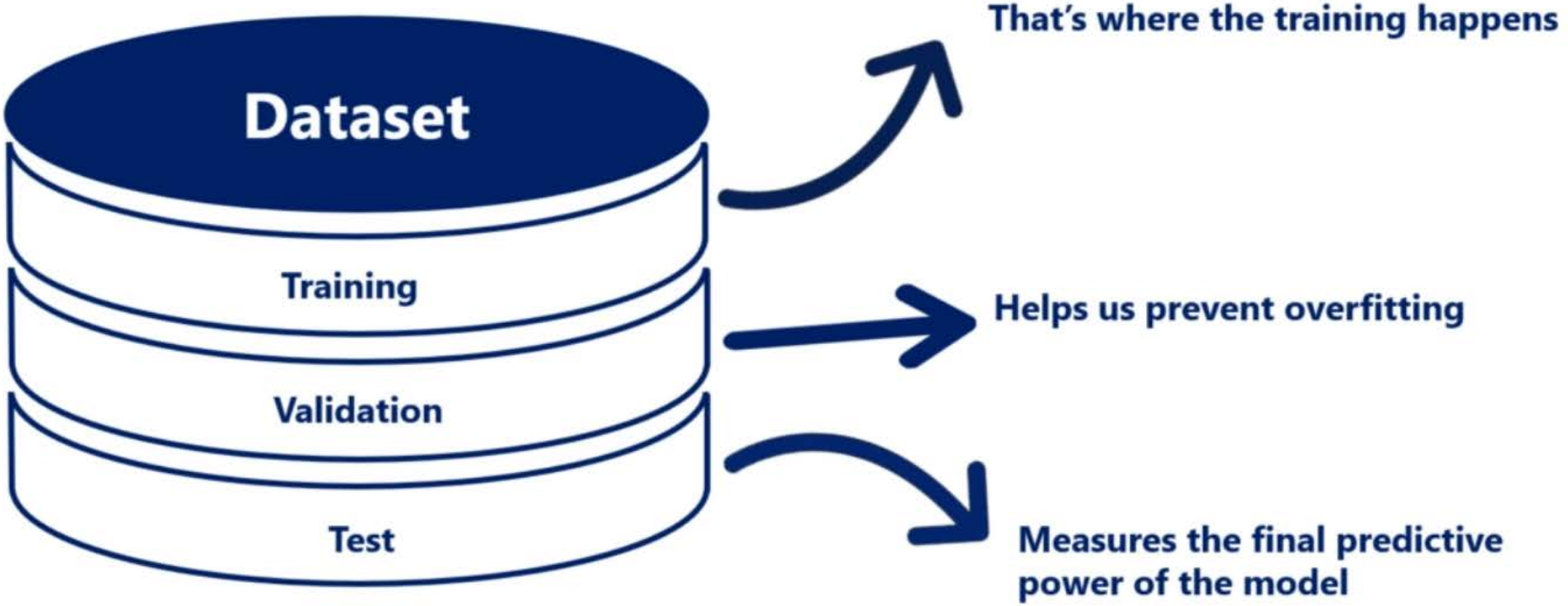
WE ARE OVERFITTING!!!



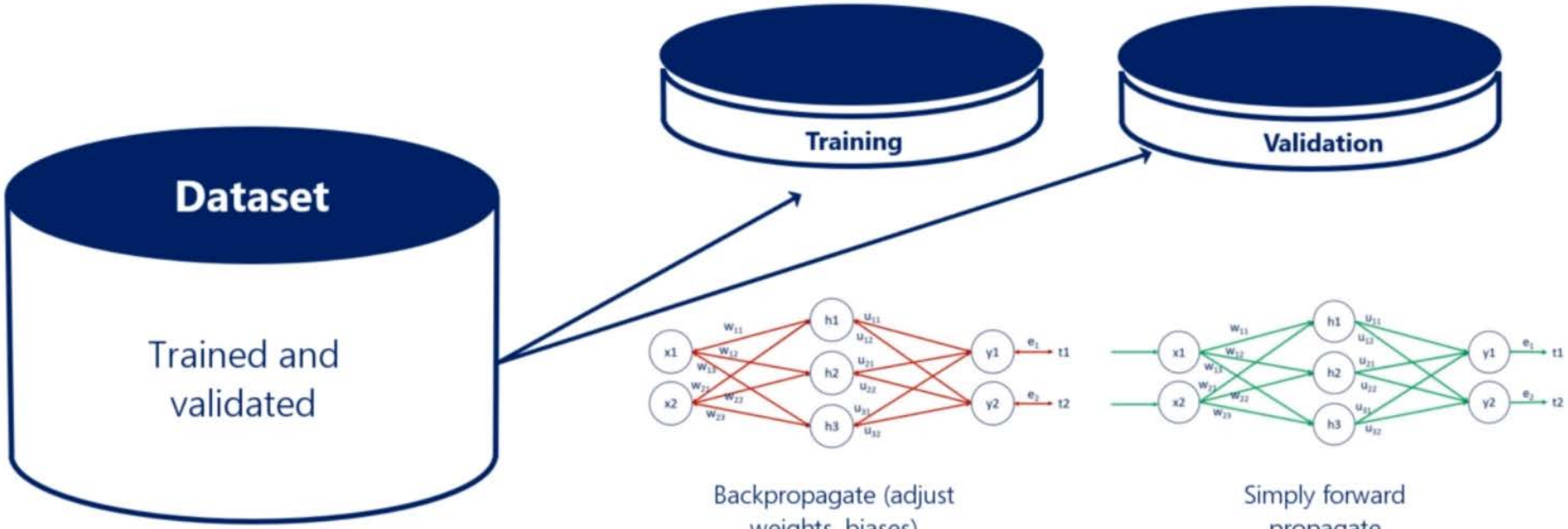


WE ARE OVERFITTING!!!

Training, validation, and test

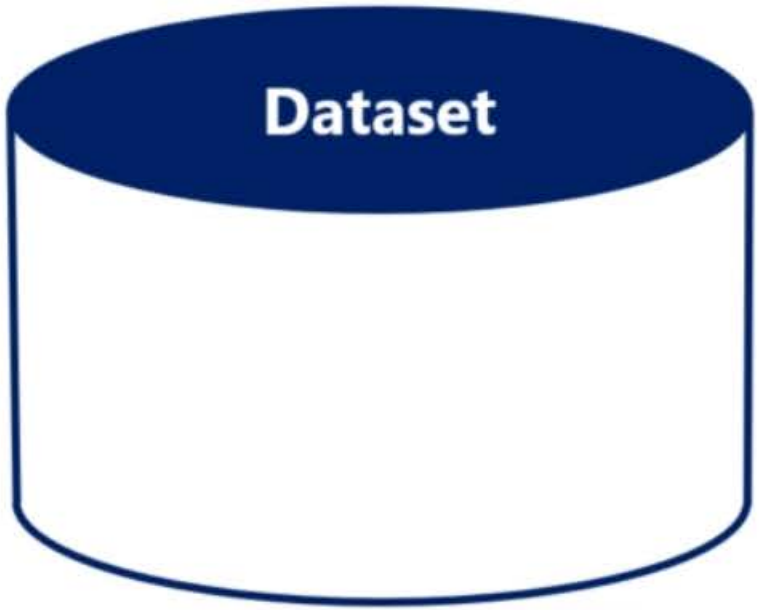


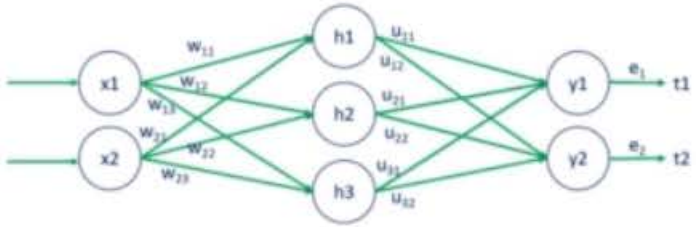
Training, validation, and test



We have the final version of the machine learning "black box"

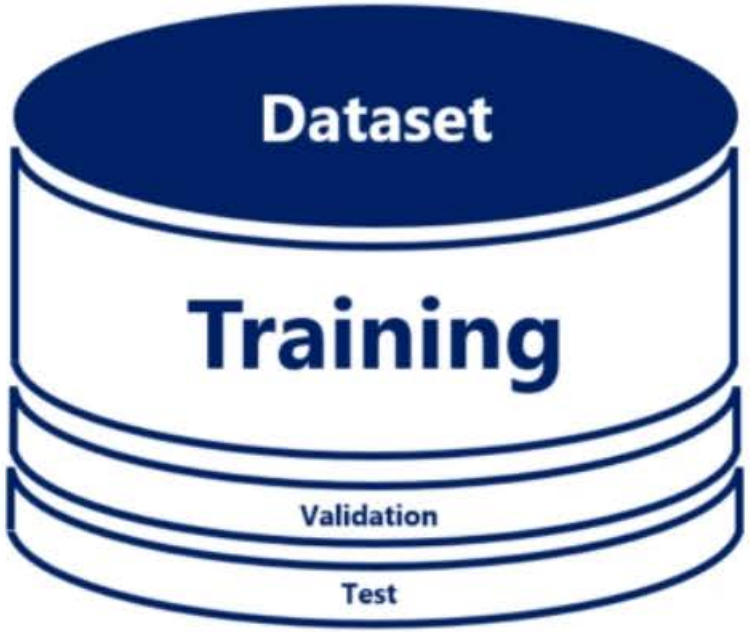
Training, validation, and test





The accuracy that we get by forward propagating the test dataset, is the accuracy we expect the model to have if we deploy it in real life

Summary


Dataset
Training
Validation
Test


80%
Training


10%
Validation

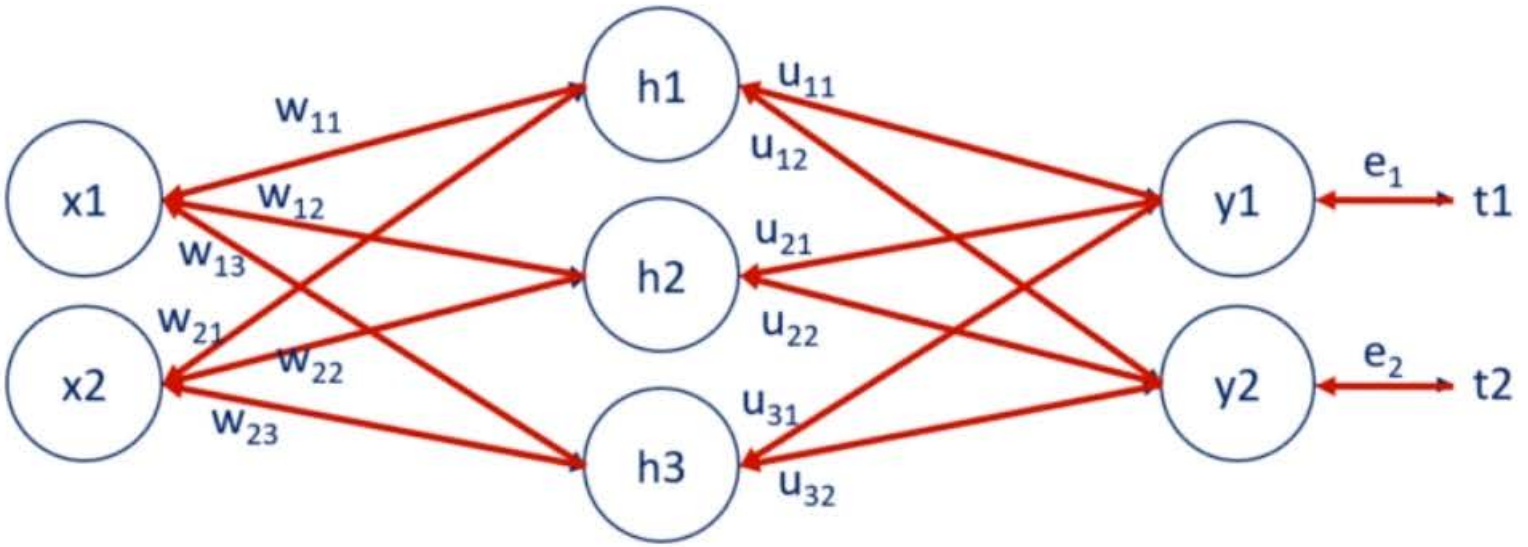

10%
Test


70%
Training


20%
Validation


10%
Test

Summary

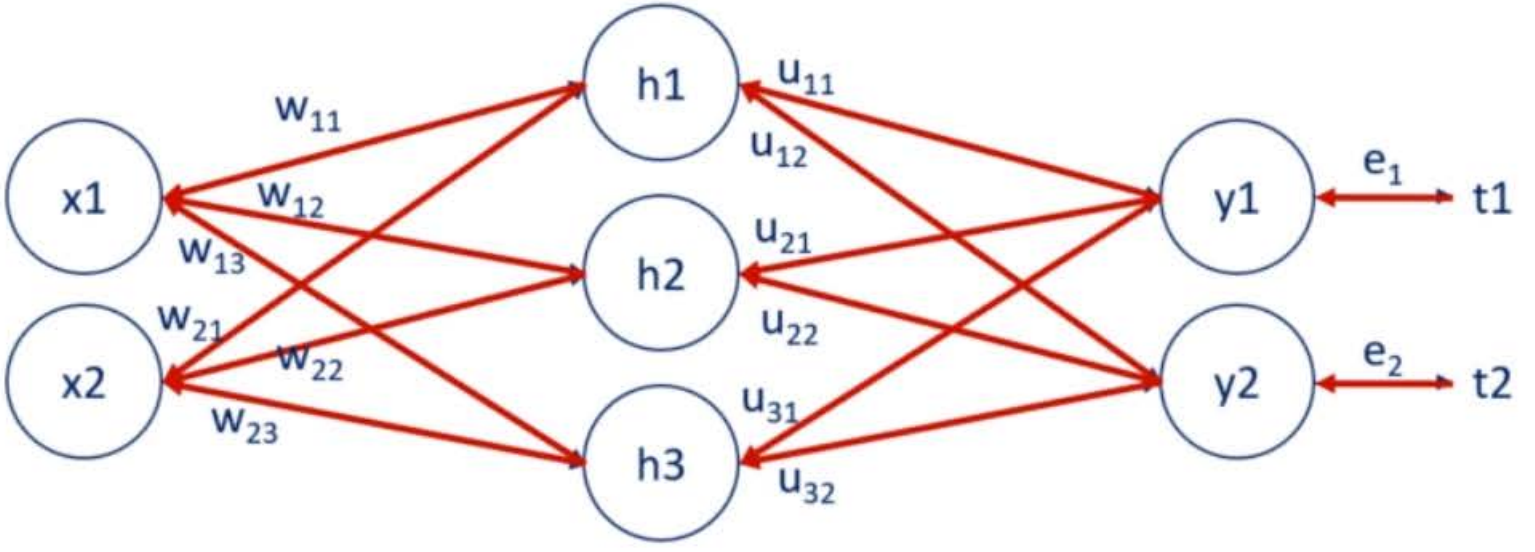


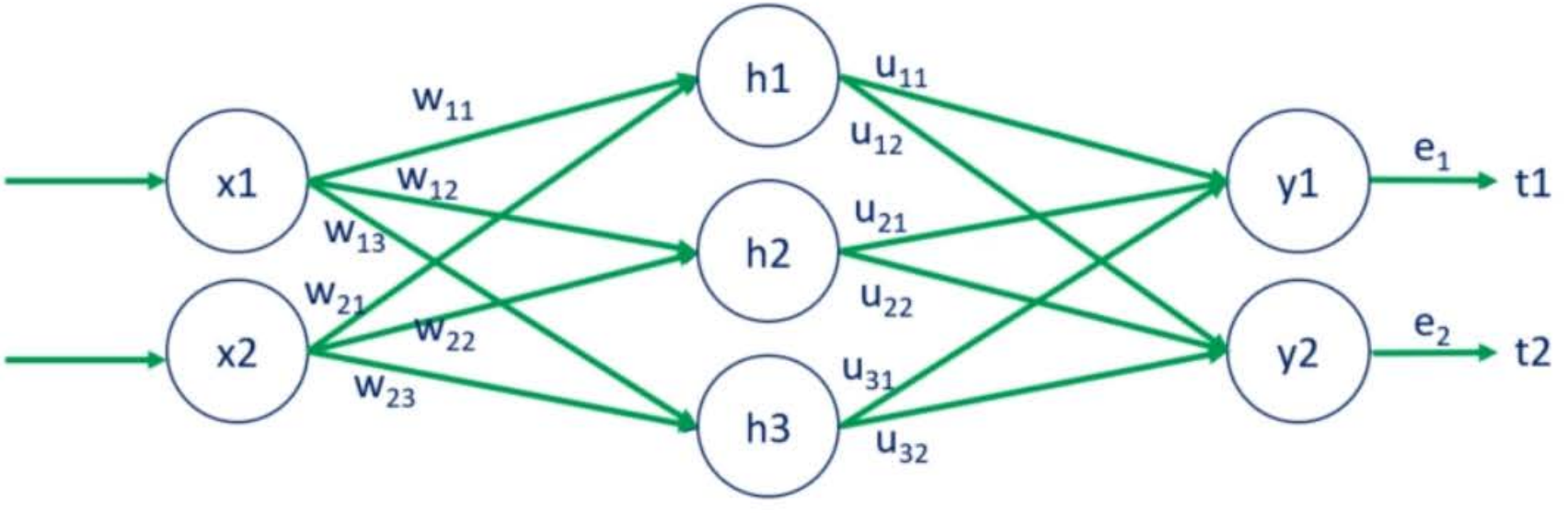
We train ONLY on the training set


80%
Training

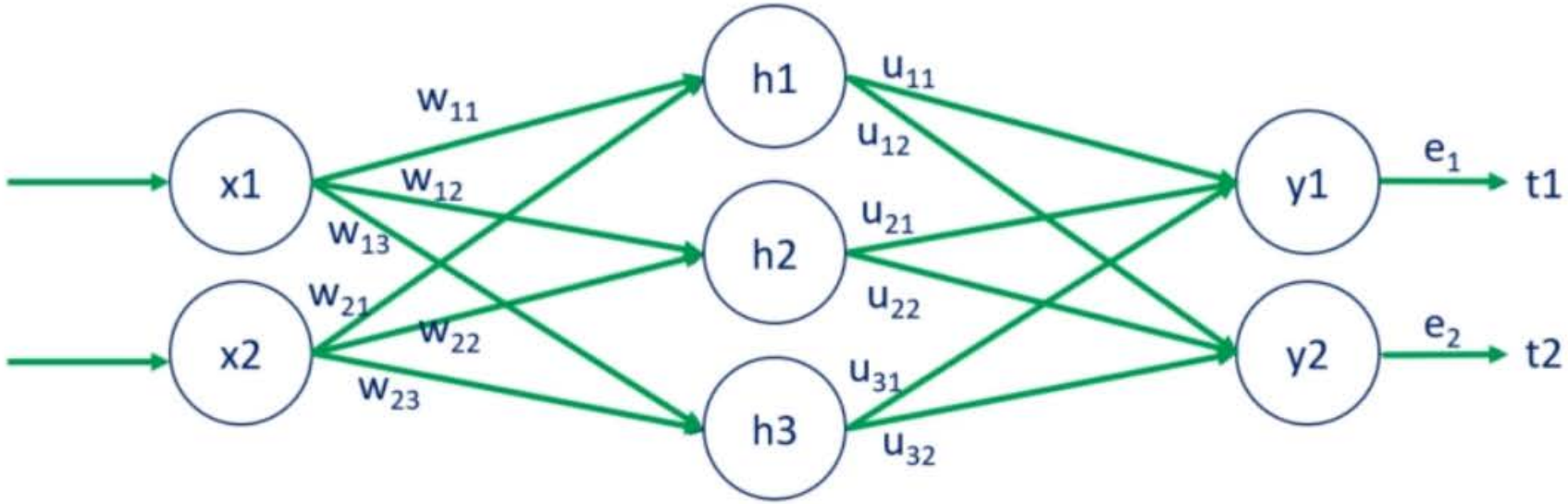

4


10%
Validation


x1
x2
h1
h2
h3
y1
y2
w11
w12
w13
w21
w22
w23
u11
u12
u21
u22
u31
u32
e1
e2
t1
t2

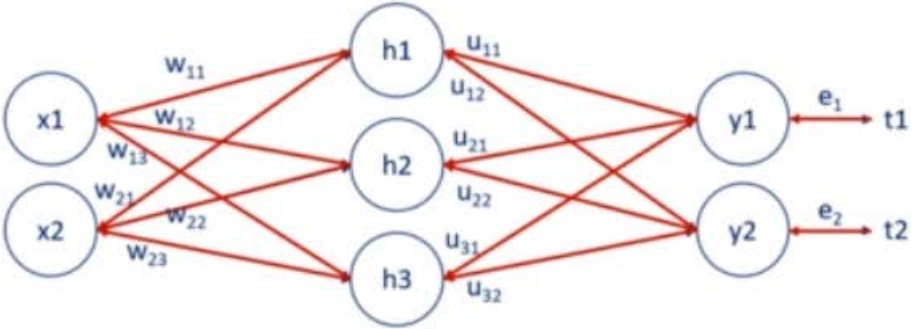

x1
x2
h1
h2
h3
y1
y2
w11
w12
w13
w21
w22
w23
u11
u12
u21
u22
u31
u32
e1
e2
t1
t2

What does "every now and then" mean?



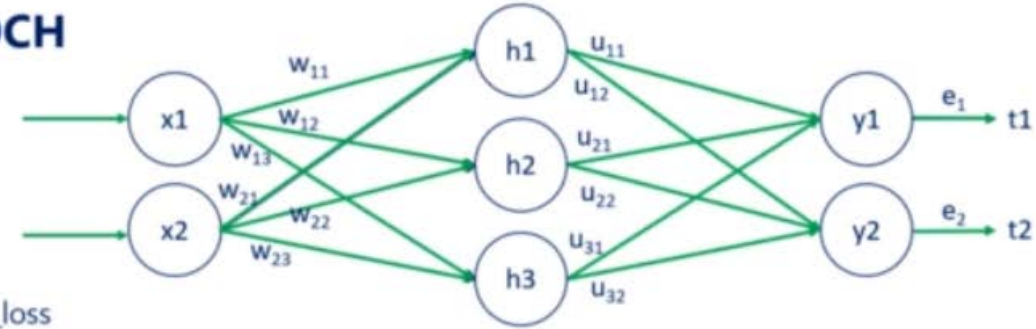
Usually, we validate on **every epoch**

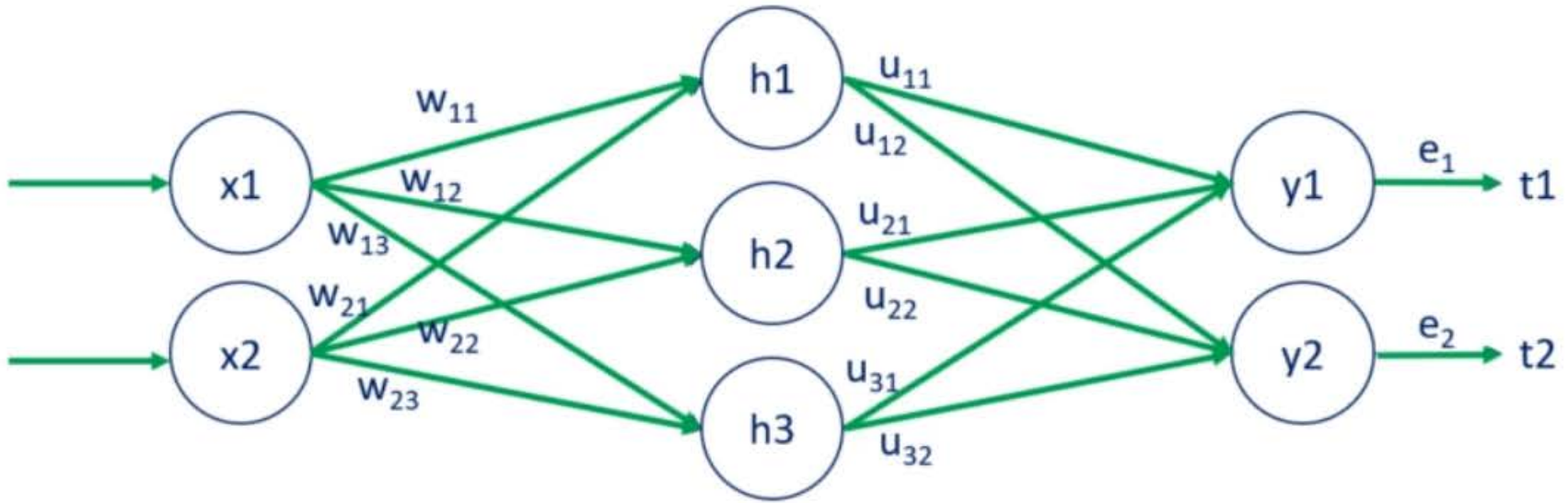
Calculate training_loss



END OF EPOCH

Calculate validation_loss





The accuracy we obtain at this stage is the accuracy of the algorithm

Training, validation, and test

We can't afford to split it in three...

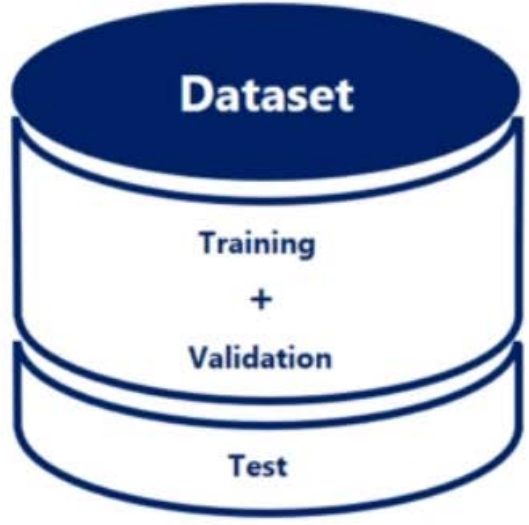
Because the algorithm may not **learn anything**

Training, validation, and test

N-FOLD CROSS-VALIDATION

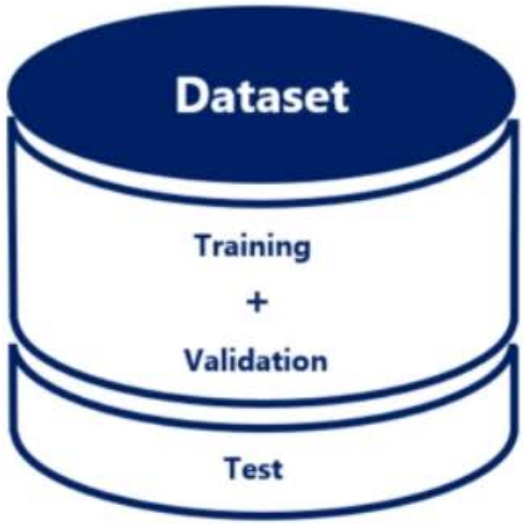
/ sometimes k-fold cross-validation /

N-FOLD CROSS-VALIDATION



N-FOLD CROSS-VALIDATION

11,000



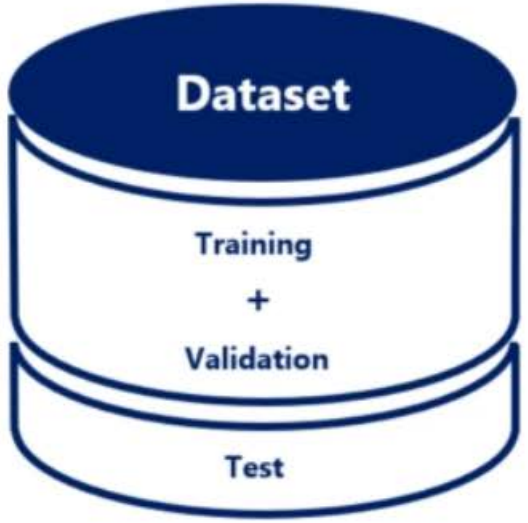
10,000

1,000

In data science we often deal with ginormous datasets

N-FOLD CROSS-VALIDATION

11,000



10,000

1,000

Side note:
"Ginormous" datasets have their own problems - being so large, they often have a lot of missing values. We refer to them as being: **sparse**

10-FOLD CROSS-VALIDATION

9,000
+
1,000

We fold it 10 times, so this is a **10-fold cross-validation**

10-FOLD CROSS-VALIDATION

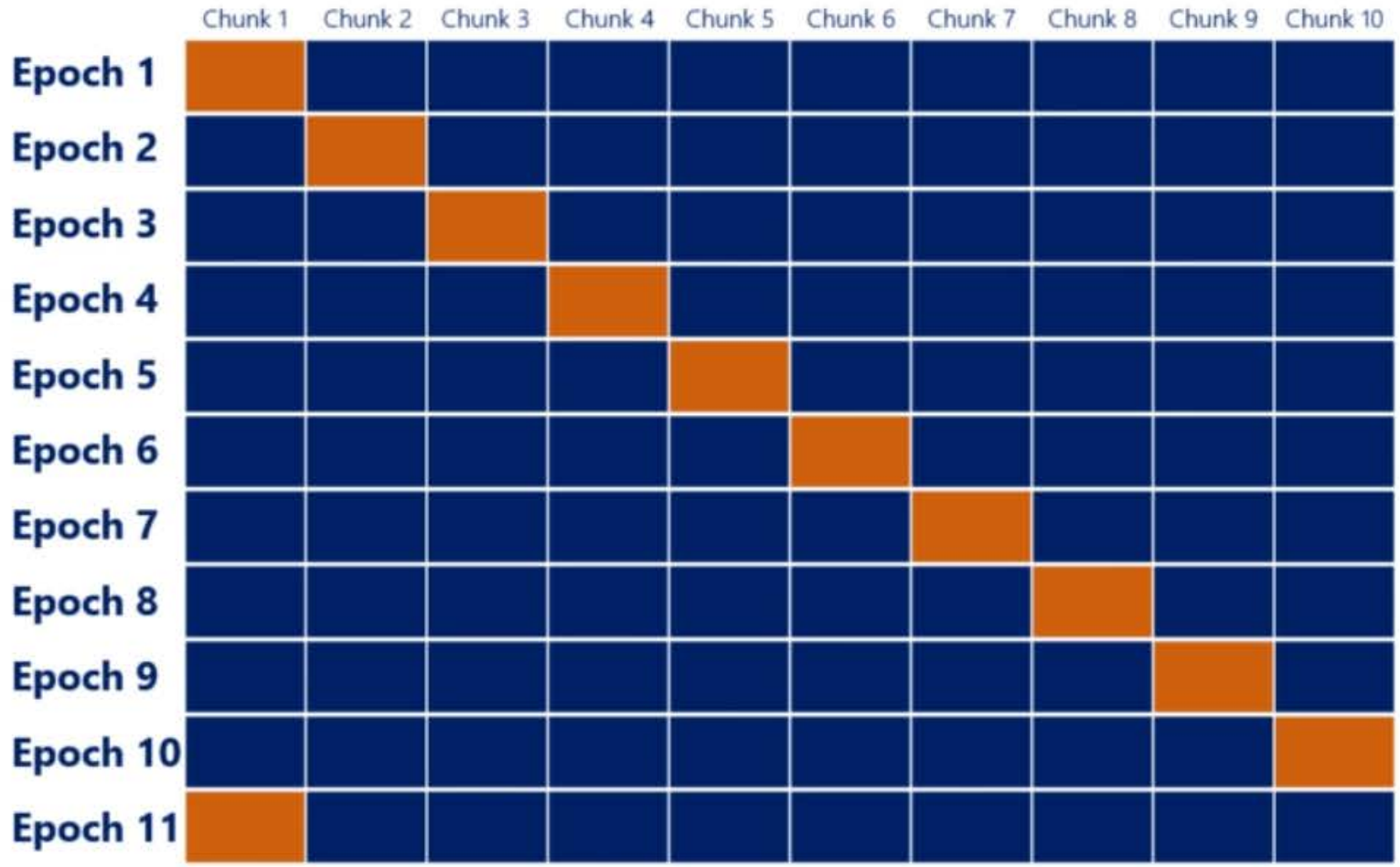
9,000
+
1,000

Training

Validation

10-FOLD CROSS-VALIDATION

9,000
+
1,000



Training

Validation

For each epoch, we don't overlap training and validation

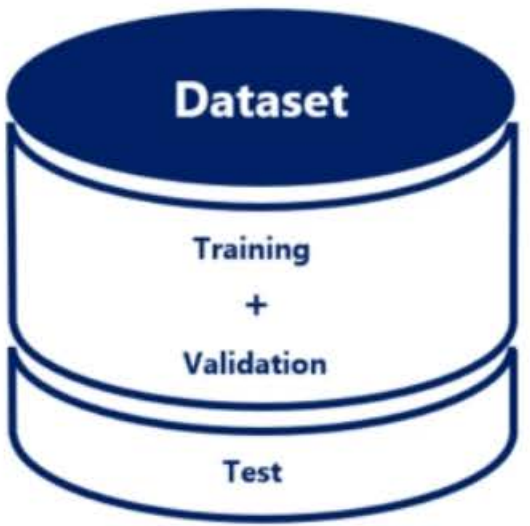
365 DataScience

N-FOLD CROSS-VALIDATION

Pros

Utilized more data

We have a model



Cons

Possibly overfitted a bit

Early stopping

Definition:

We want to...

... stop training early...

... i.e. before we overfit.

Early stopping

1 Preset number of epochs

Learning

In [9]:
```
for e in range(100):

    _, curr_loss = sess.run([optimize, mean_loss],
                            feed_dict = {inputs: training_data['inputs'], targets: training_data['targets']})

    print (curr_loss)
```

Pros:

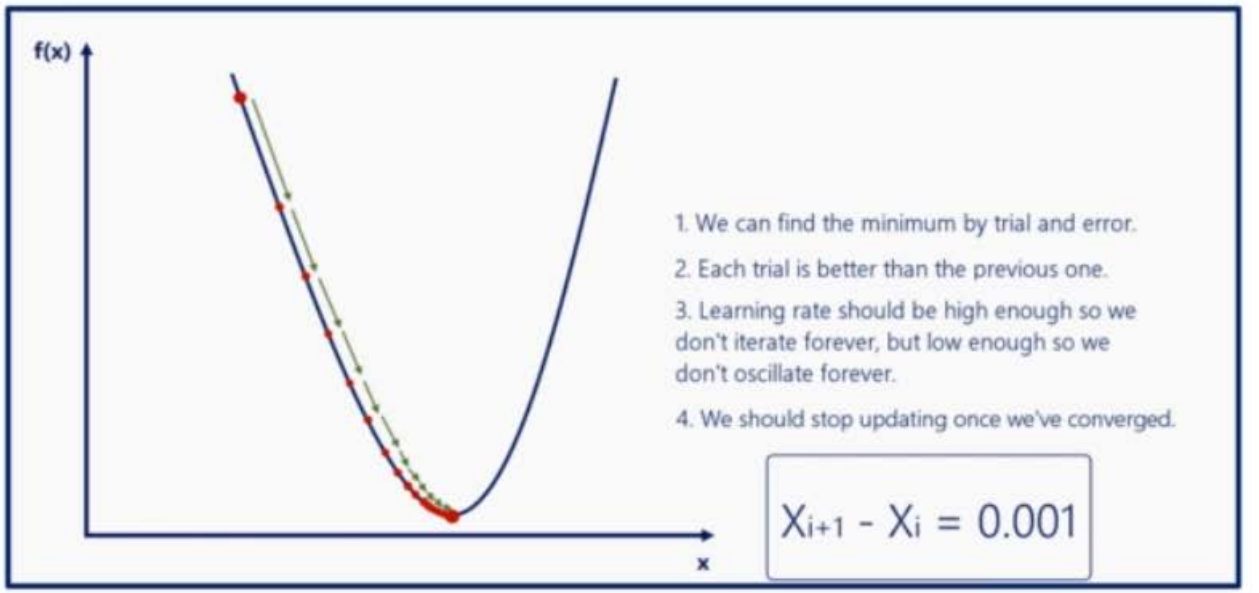
1. Eventually, solves the problem

Cons:

1. No guarantee that the min is reached
2. Maybe doesn't minimize at all

Early stopping

2 **Stop when updates become too small**



Pros:

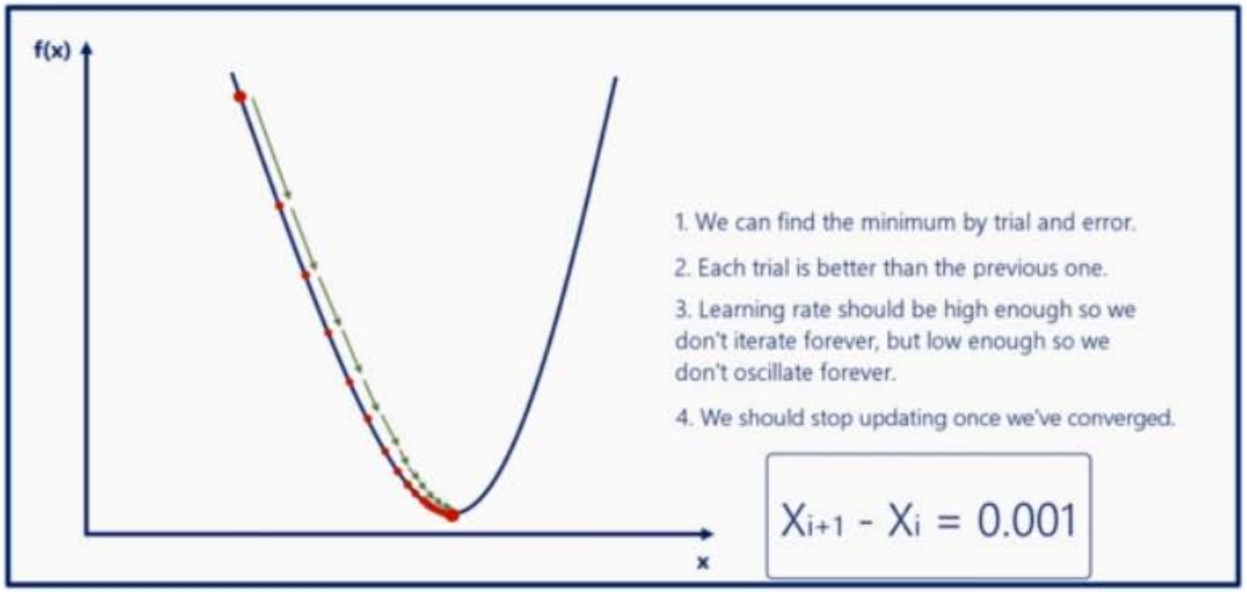
1. We are sure the loss is minimized

Cons:

The loss function stops changing, making the update rule yield same **weights**

Early stopping

2 Stop when updates become too small



Pros:

1. We are sure the loss is minimized
2. Saves computing power

Cons:

We save computing power by using as few iterations as possible; we don't iterate uselessly

Early stopping

	Preset epochs	Updates too small	
Solves the problem			
Certain that loss is minimized			
Doesn't iterate uselessly			

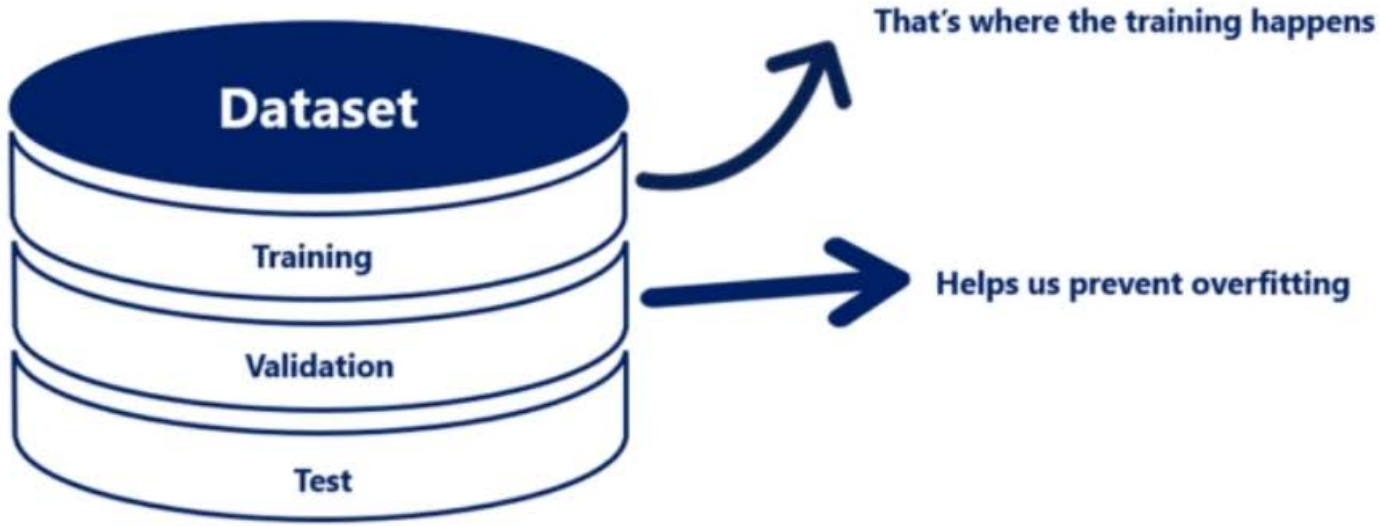
We can't guess the number of required epochs

Early stopping

	Preset epochs	Updates too small	
Solves the problem	✓	✓	
Certain that loss is minimized	✗	✓	
Doesn't iterate uselessly	✗	✓	
Prevents overfitting	✗	✗	

Early stopping

3 Validation set strategy

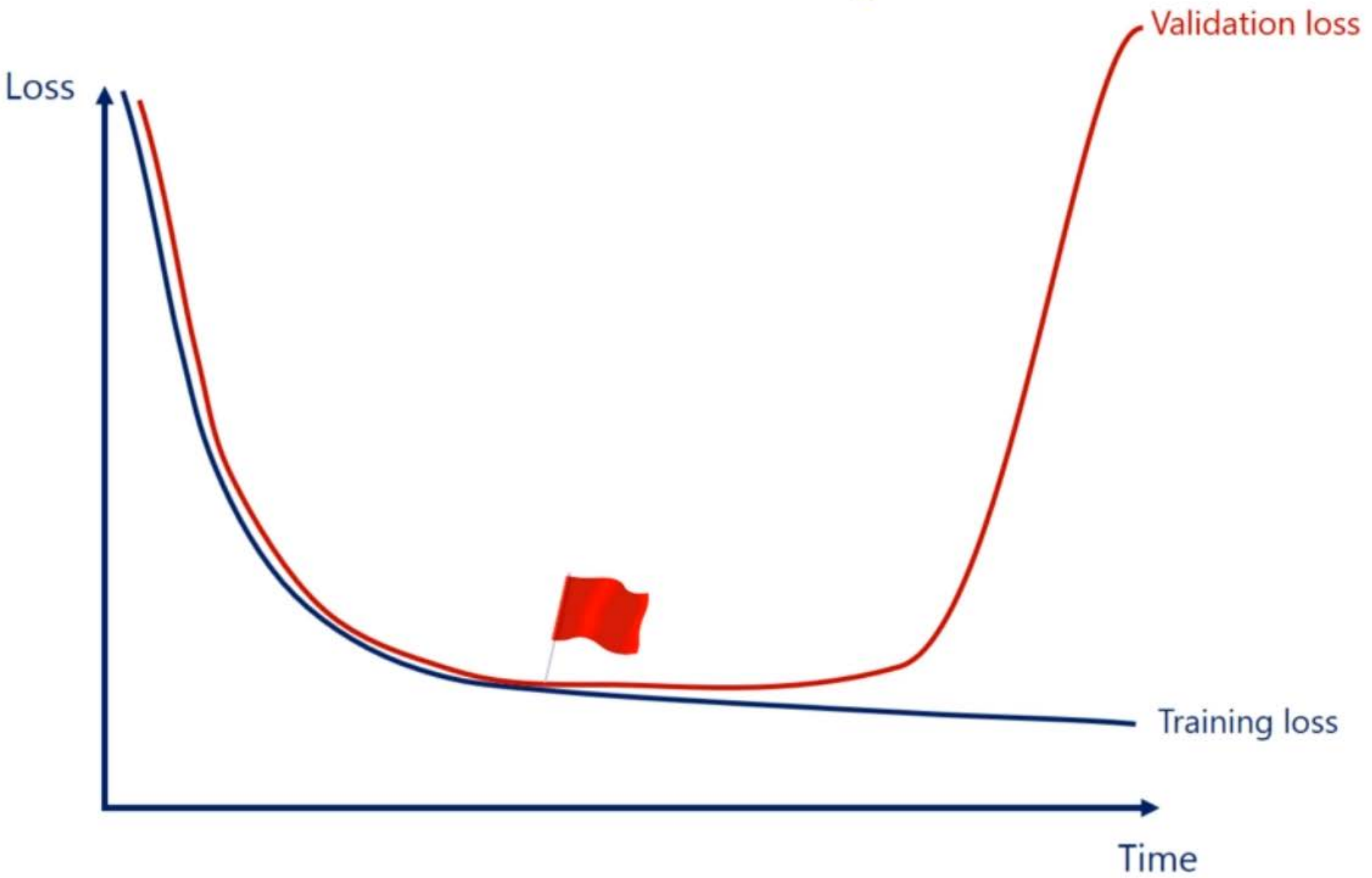


Pros:

1. We are sure the validation loss is minimized
2. Saves computing power
3. Prevents overfitting

Cons:

Typical training



Early stopping

	Preset epochs	Updates too small	Validation set
Solves the problem	✓	✓	✓
Certain that loss is minimized	✗	✓	✓
Doesn't iterate uselessly	✗	✓	✗
Prevents overfitting	✗	✗	✓

Early stopping

	Preset epochs	Updates too small	Validation set
Solves the problem	✓	✓	✓
Certain that loss is minimized	✗	✓	✓
Doesn't iterate uselessly	✗	✓	✗
Prevents overfitting	✗	✗	✓

Minimal example

Validation set strategy iterates **until we start overfitting**

Early stopping

	Preset epochs	Updates too small	Validation set
Solves the problem	✓	✓	✓
Certain that loss is minimized	✗	✓	✓
Doesn't iterate uselessly	✗	✓	✗
Prevents overfitting	✗	✗	✓

Minimal example

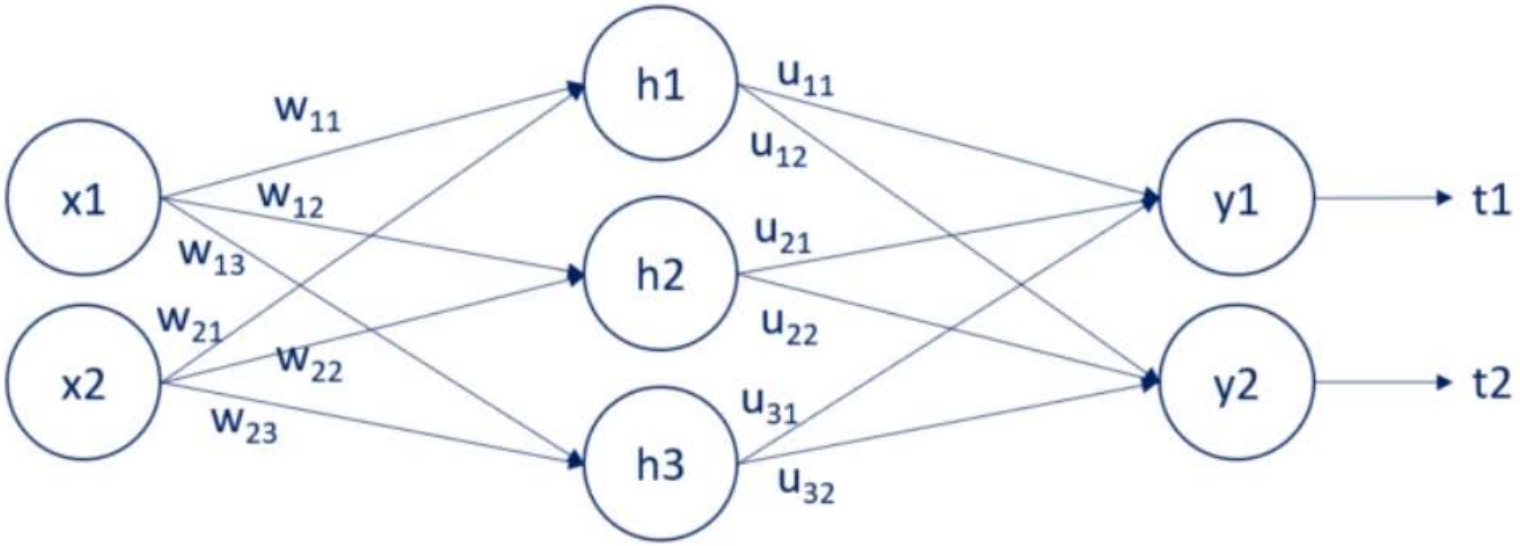
Best practice

INITIALIZATION

Initialization is the process in which we set the initial values of weights.

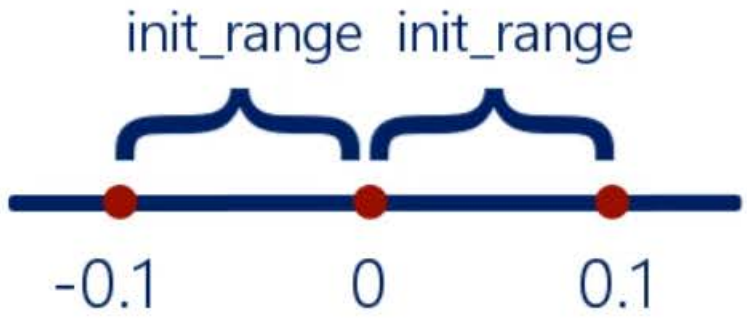
#veryimportant

Initialization



Let's initialize our weights and biases in such a way that they **are all equal to a constant**

Our initial weights and biases will be picked randomly from the interval [-0.1,0.1]



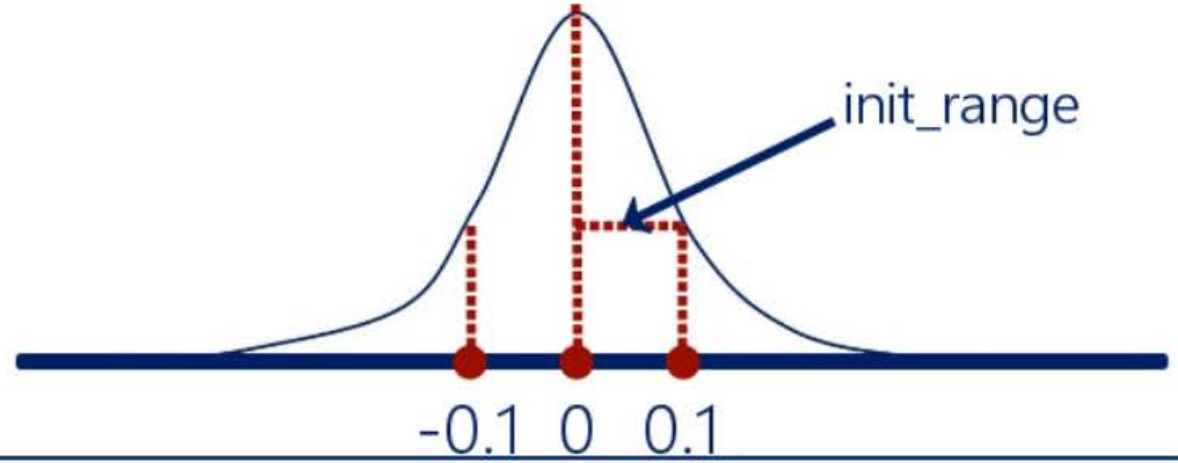
Initialize variables

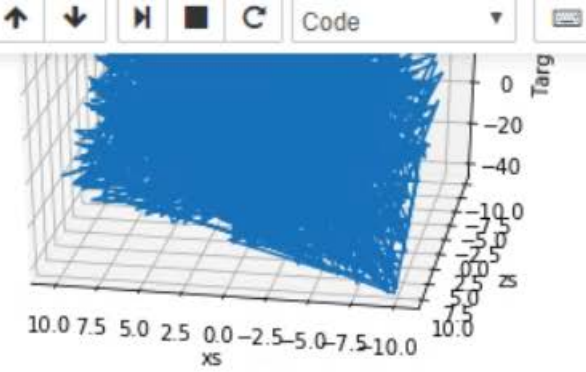
In [10]:
```
init_range = 0.1

weights = np.random.uniform(-init_range,init_range, size=(2,1))

biases = np.random.uniform(-init_range,init_range,size=1)

print(weights)
print(biases)
```

W
```
[[-0.02758852]
 [-0.03556078]]
```
b
```
[-0.05488636]
```

In [ ]:

Our initial weights and biases will be picked randomly from the interval [-0.1,0.1] in a random normal manner, where the mean is 0 and the standard deviation is 0.1 (variance 0.01).





In [ ]:

Initialize variables

In [10]:
```
init_range = 0.1

weights = np.random.uniform(-init_range,init_range, size=(2,1))

biases = np.random.uniform(-init_range,init_range,size=1)

print(weights)
print(biases)
```

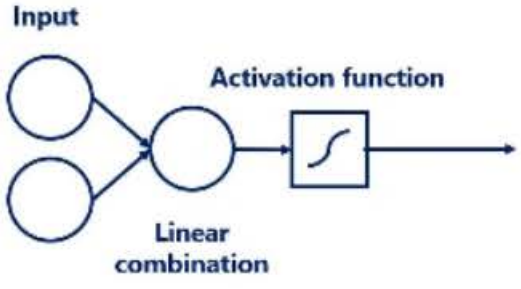
```
[[-0.02758852]
 [-0.03556078]]
[-0.05488636]
```

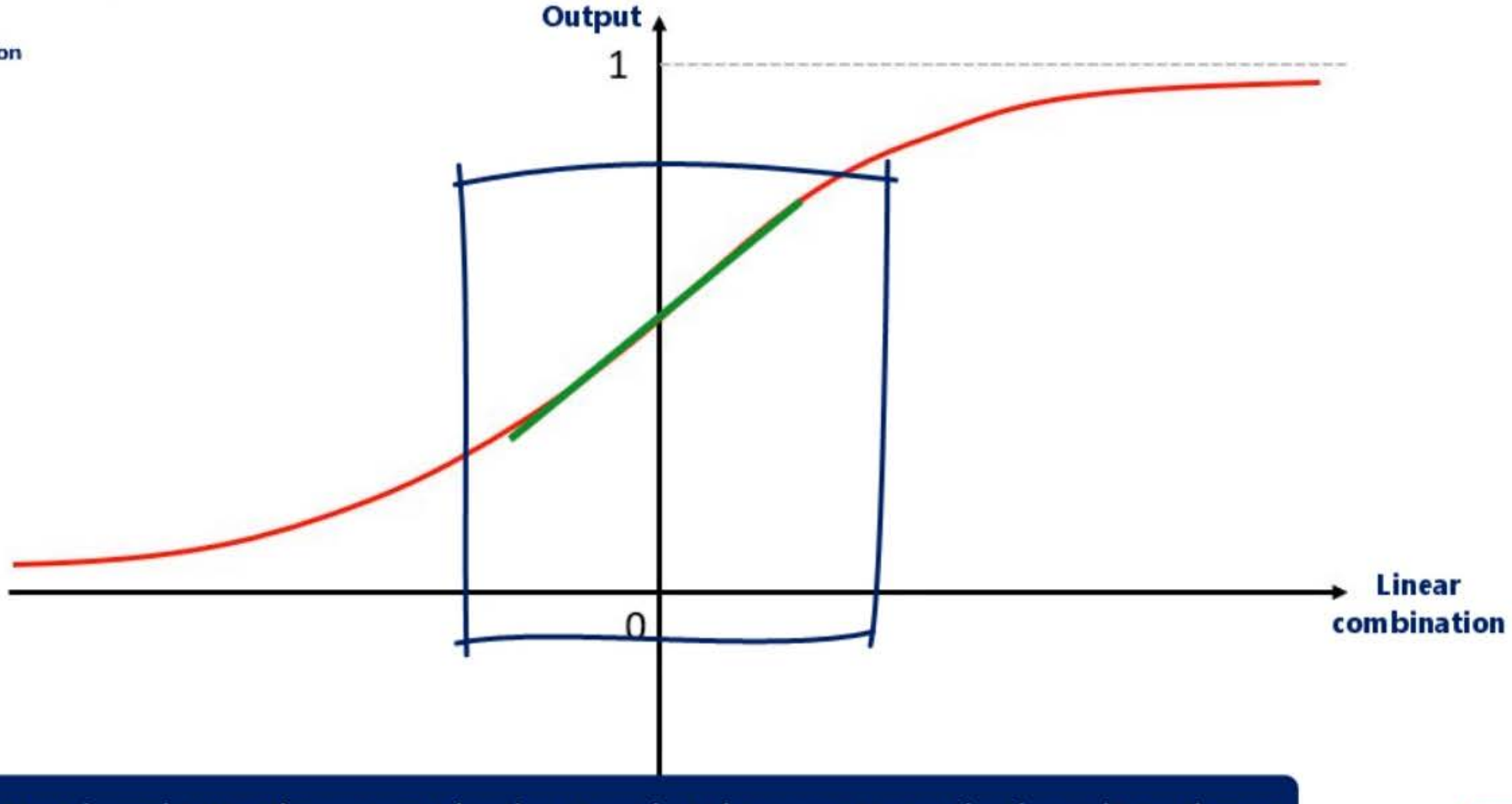
In [ ]:

In [ ]:

Both methods are somewhat **problematic**

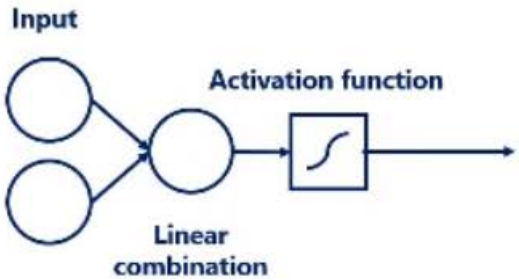
Sigmoid activation function

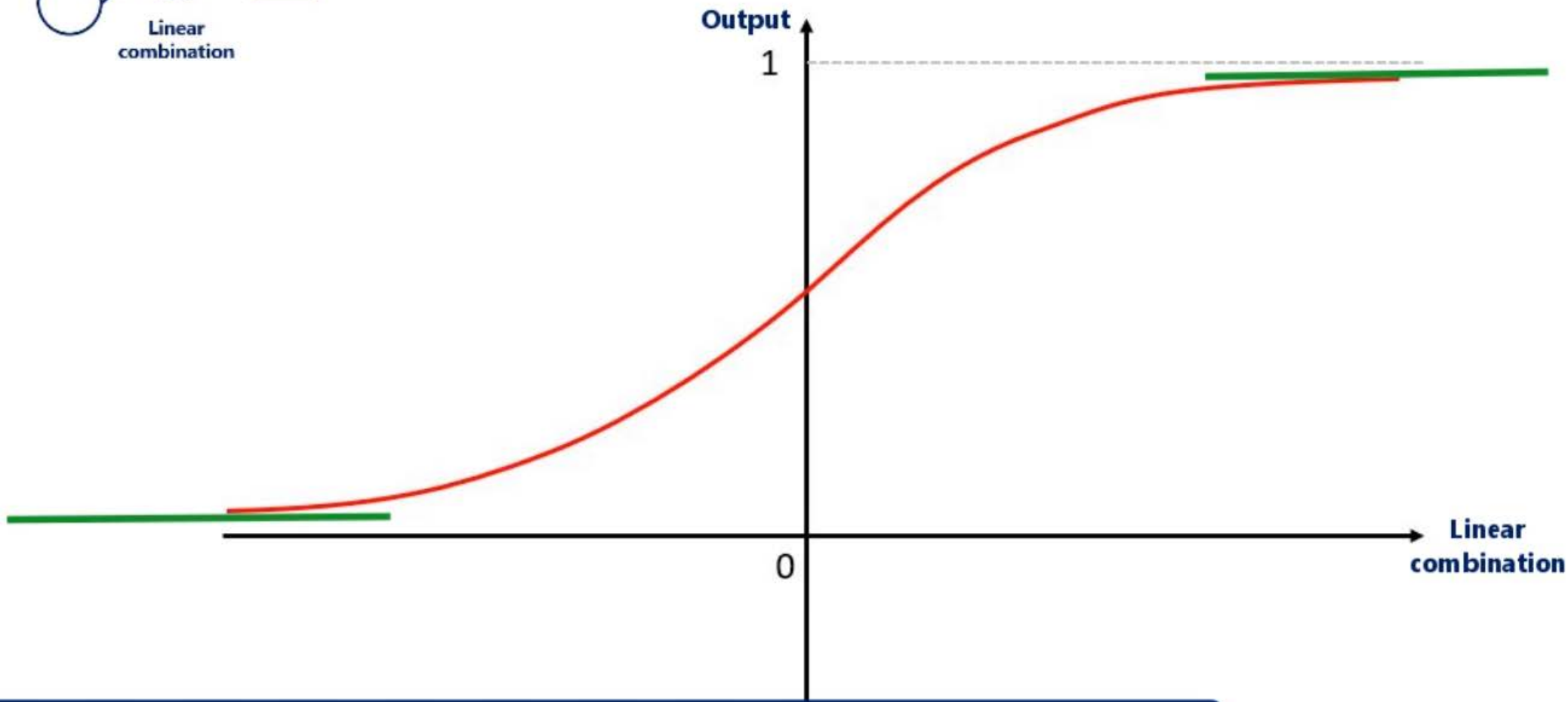




If all the inputs for the sigmoid (the linear combinations) are in this range, the activation will be linear

Sigmoid activation function





A static output (only ~0s or ~1s) is also problematic, as the algorithm doesn't learn

Xavier initialization

Glorot initialization

Xavier Glorot

initialization

Xavier initialization

Original paper

Understanding the difficulty of training deep feedforward neural networks

Xavier Glorot, Yoshua Bengio ;
Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, PMLR 9:249-256, 2010.

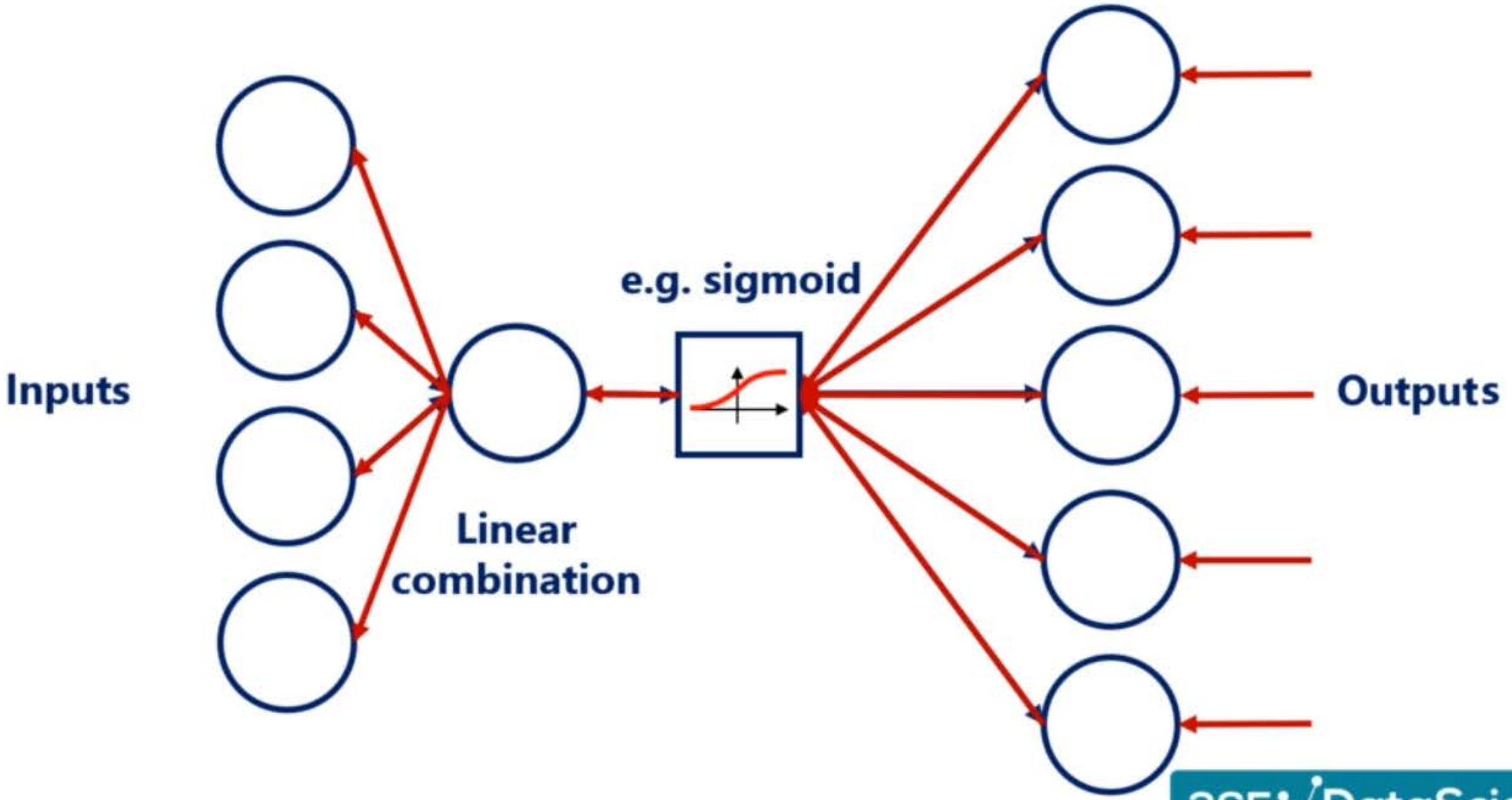
Xavier initialization

Uniform Xavier initialization: draw each weight, w, from a random uniform distribution in [-x,x] for $x = \sqrt{\frac{6}{inputs + outputs}}$

Normal Xavier initialization: draw each weight, w, from a normal distribution with a mean of 0, and a standard deviation $\sigma = \sqrt{\frac{2}{inputs + outputs}}$

Main idea: Method is not so important. The number of **inputs and outputs** is

Why do inputs and outputs matter?



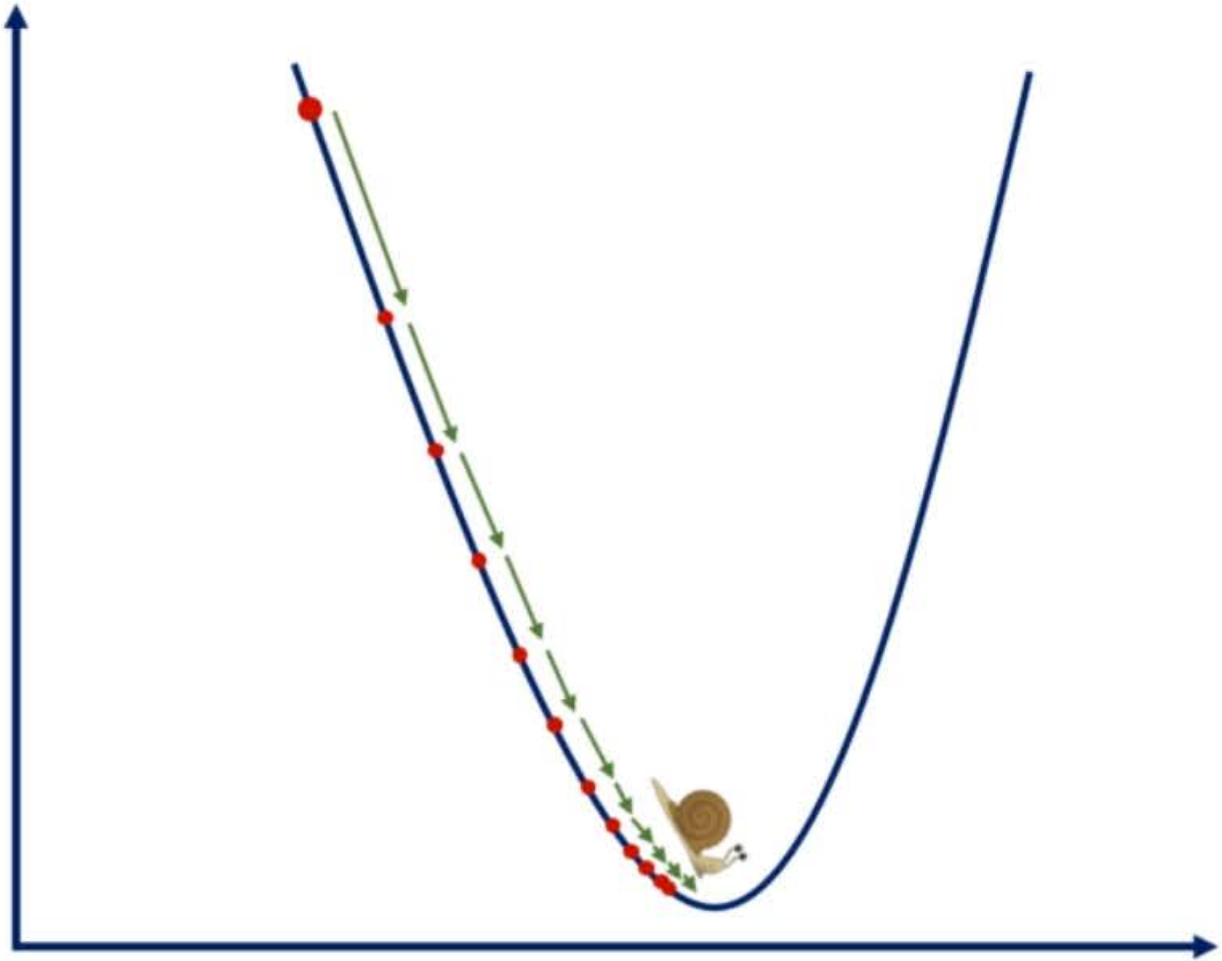
```
v1 = foo()  # Creates v.
v2 = foo()  # Gets the same, existing v.
assert v1 == v2
```

If initializer is `None` (the default), the default initializer passed in the variable scope will be used. If that one is `None` too, a `glorot_uniform_initializer` will be used. The initializer can also be a Tensor, in which case the variable is initialized to this value and shape.

OPTIMIZATION

GRADIENT DESCENT

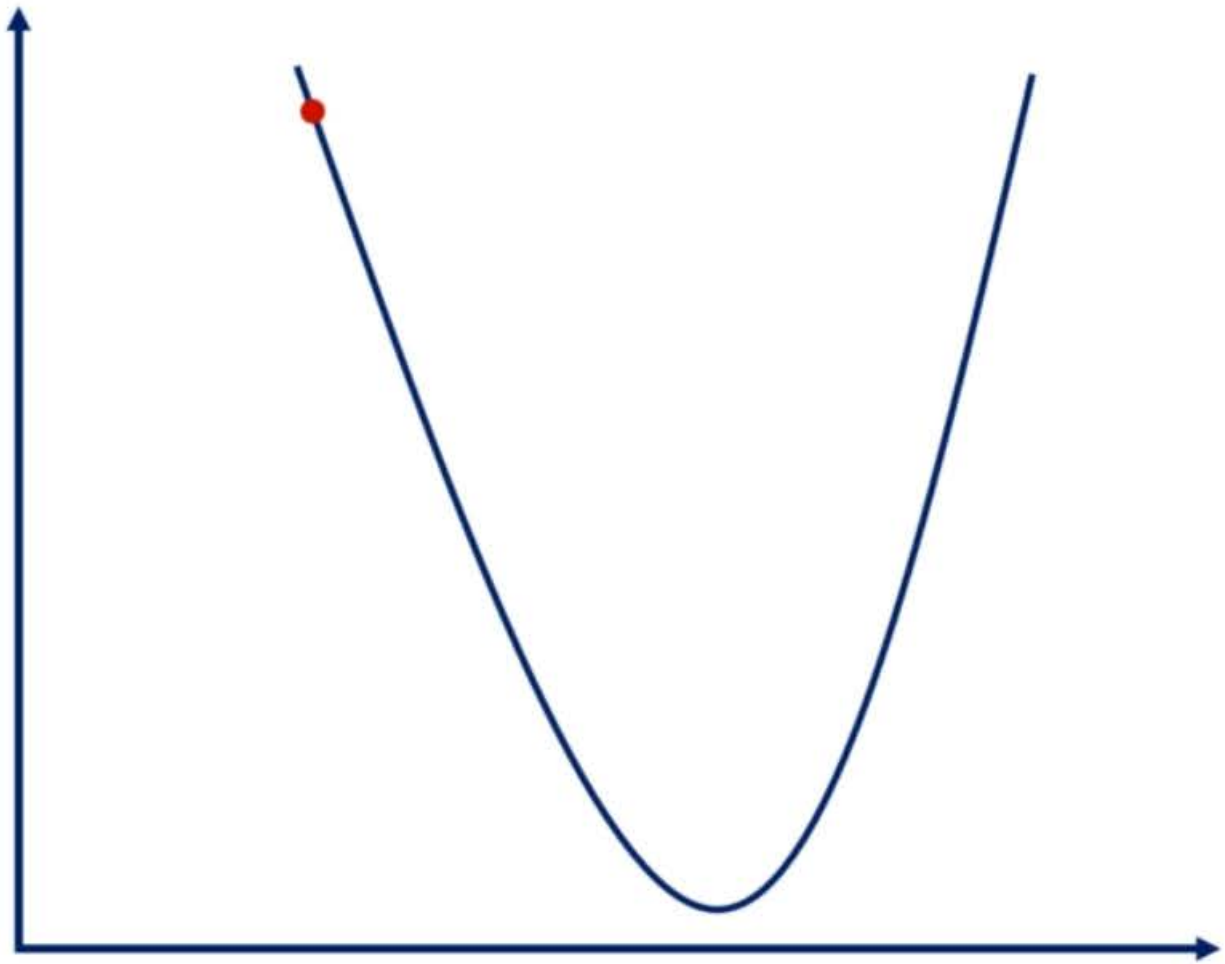
Gradient descent

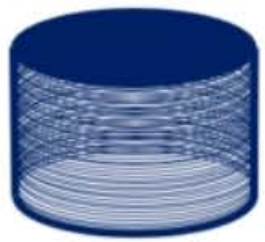


x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}
x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}	x_{28}
x_{31}	x_{32}	x_{33}	x_{34}	x_{35}	x_{36}	x_{37}	x_{38}
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
x_{n1}	x_{n2}	x_{n3}	x_{n4}	x_{n5}	x_{n6}	x_{n7}	x_{n8}

n x k

Stochastic gradient descent





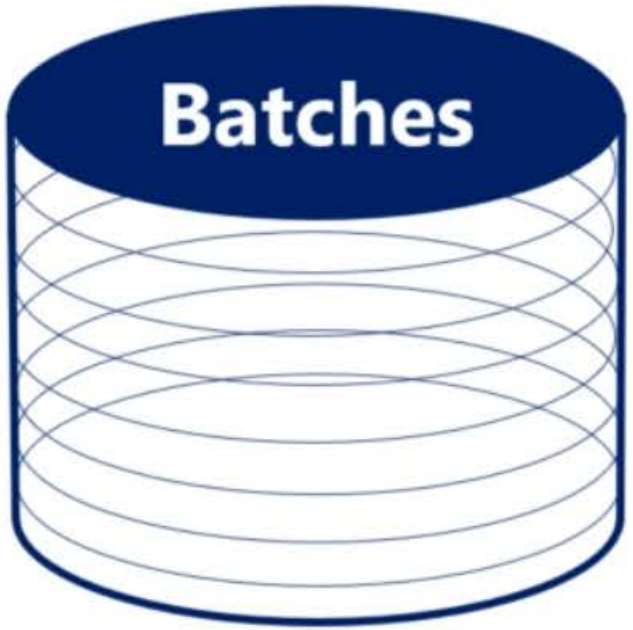
x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}
x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}	x_{28}
x_{31}	x_{32}	x_{33}	x_{34}	x_{35}	x_{36}	x_{37}	x_{38}
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
x_{n1}	x_{n2}	x_{n3}	x_{n4}	x_{n5}	x_{n6}	x_{n7}	x_{n8}

n x k

It works in the same way, but updates the weights many times inside a single epoch

Batching

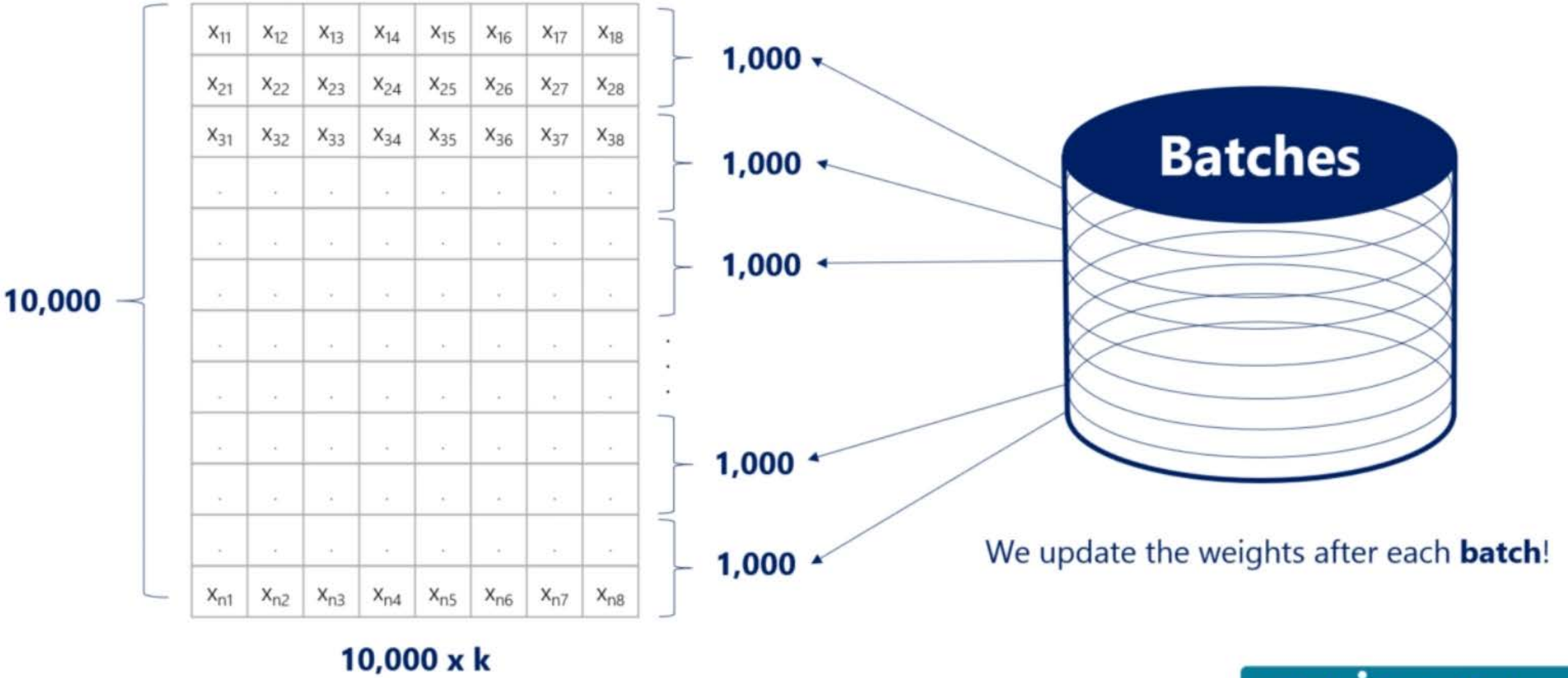
x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}
x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}	x_{28}
x_{31}	x_{32}	x_{33}	x_{34}	x_{35}	x_{36}	x_{37}	x_{38}
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
x_{n1}	x_{n2}	x_{n3}	x_{n4}	x_{n5}	x_{n6}	x_{n7}	x_{n8}



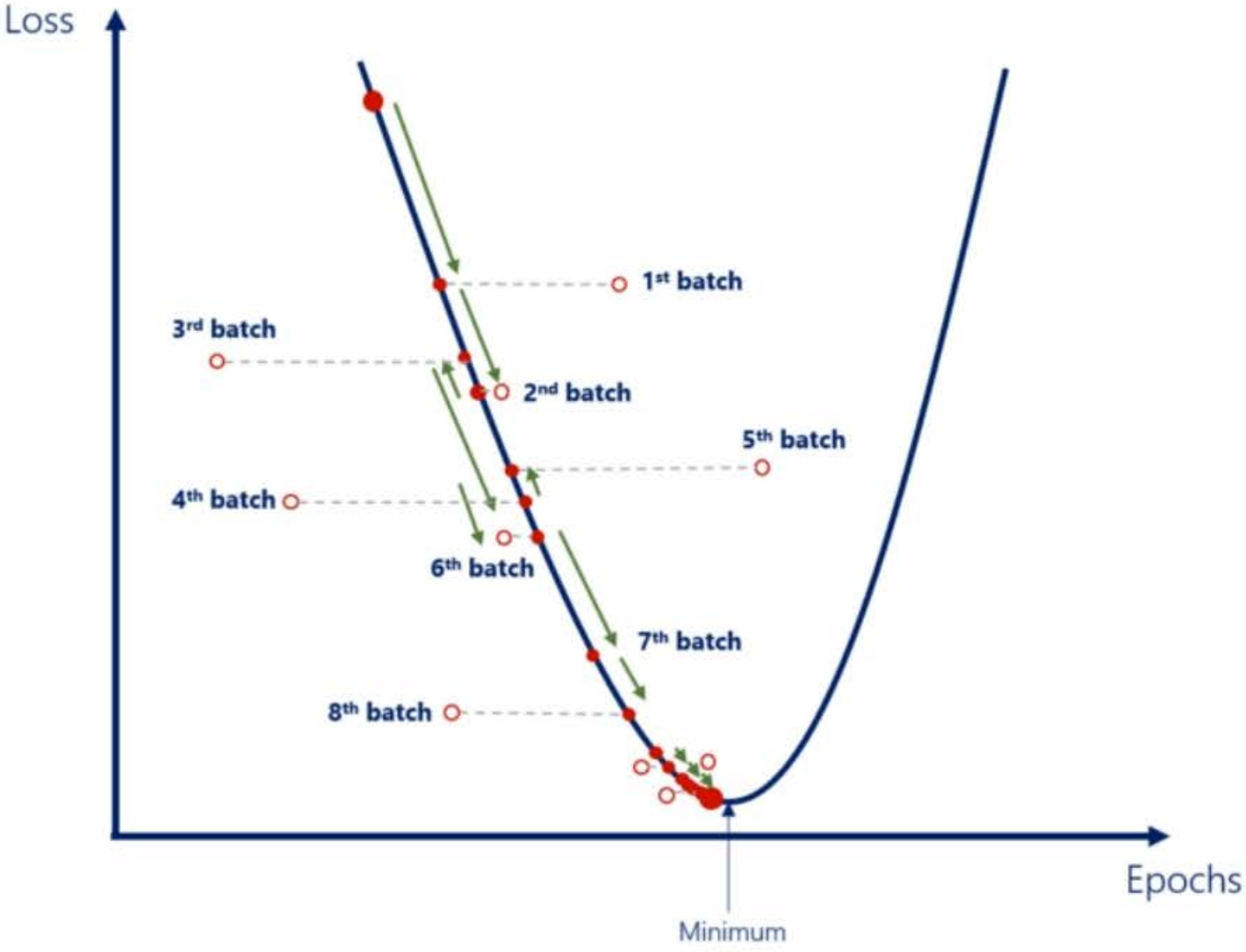
We update the weights after each **batch**!

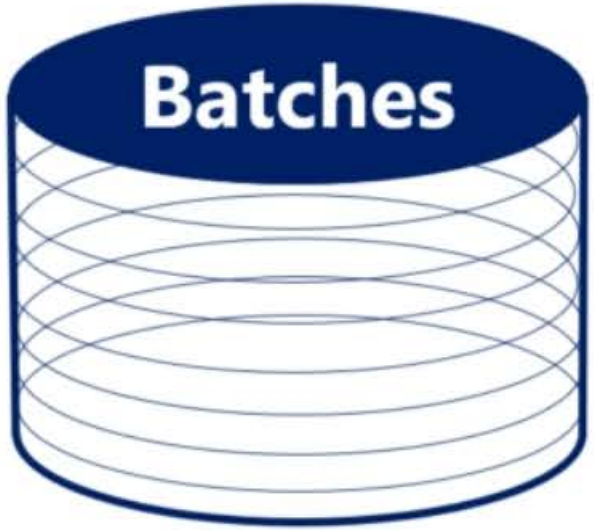
Batching: the process of splitting the dataset in **n** batches (mini-batches)

Batching

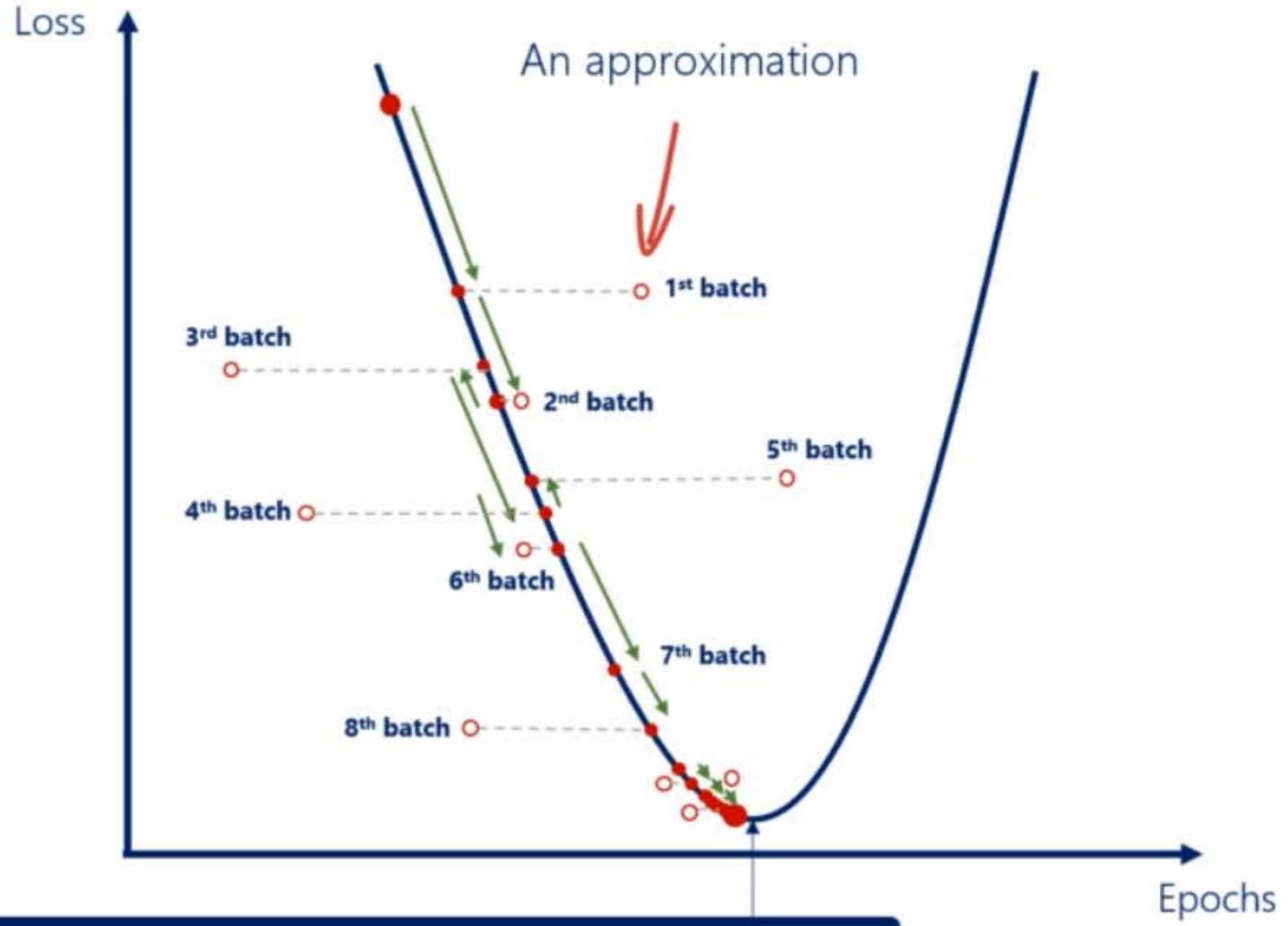

x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17} x_{18}
x_{21} x_{22} x_{23} x_{24} x_{25} x_{26} x_{27} x_{28}
x_{31} x_{32} x_{33} x_{34} x_{35} x_{36} x_{37} x_{38}
x_{n1} x_{n2} x_{n3} x_{n4} x_{n5} x_{n6} x_{n7} x_{n8}
10,000
10,000 x k
1,000
1,000
1,000
1,000
1,000
Batches
We update the weights after each **batch**!

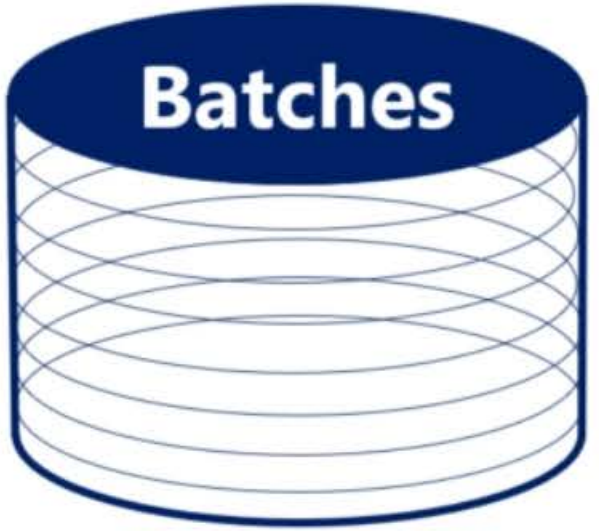
Batching


Loss
1st batch
3rd batch
2nd batch
5th batch
4th batch
6th batch
7th batch
8th batch
Epochs
Minimum


Batches

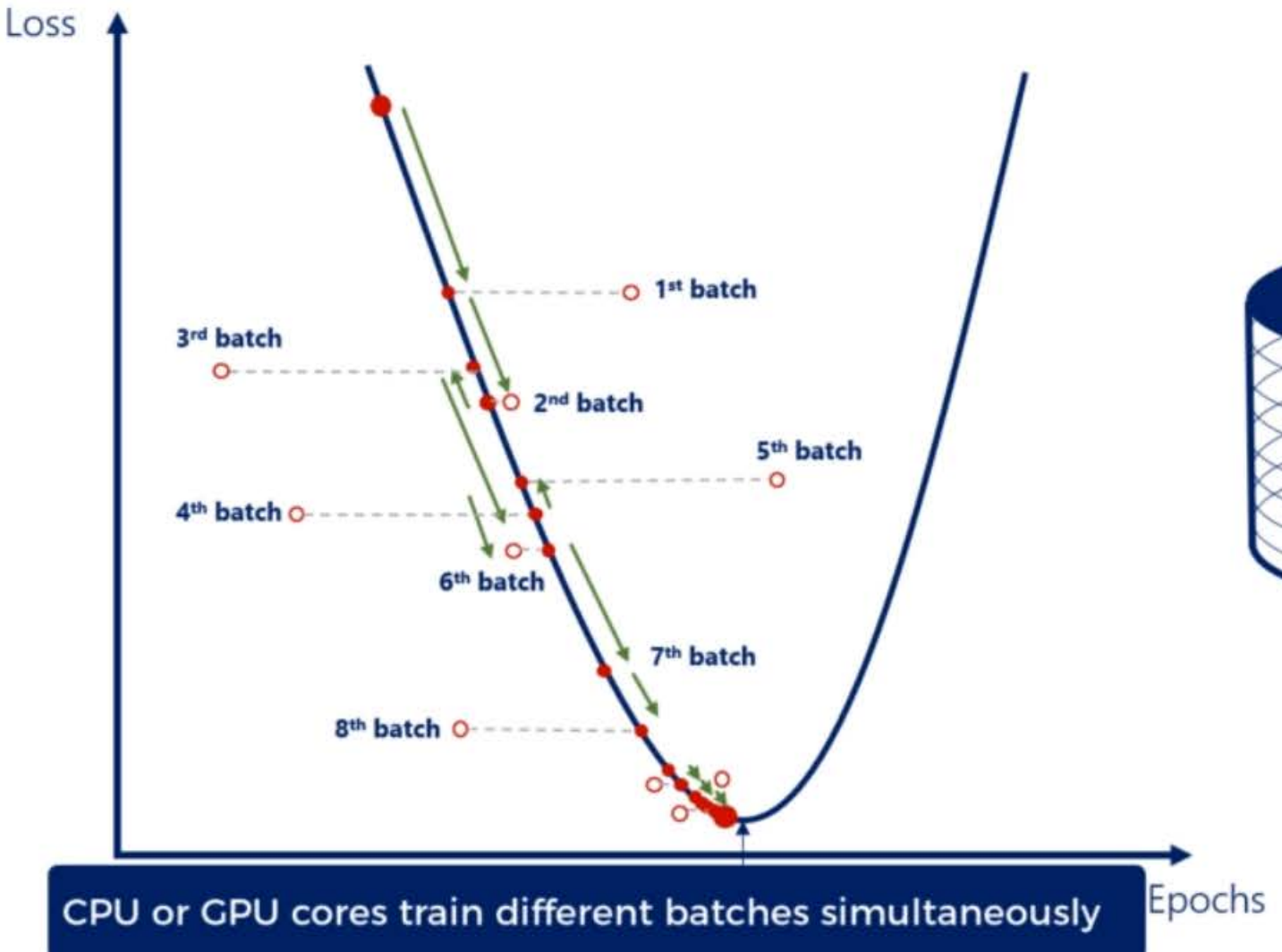
Batching

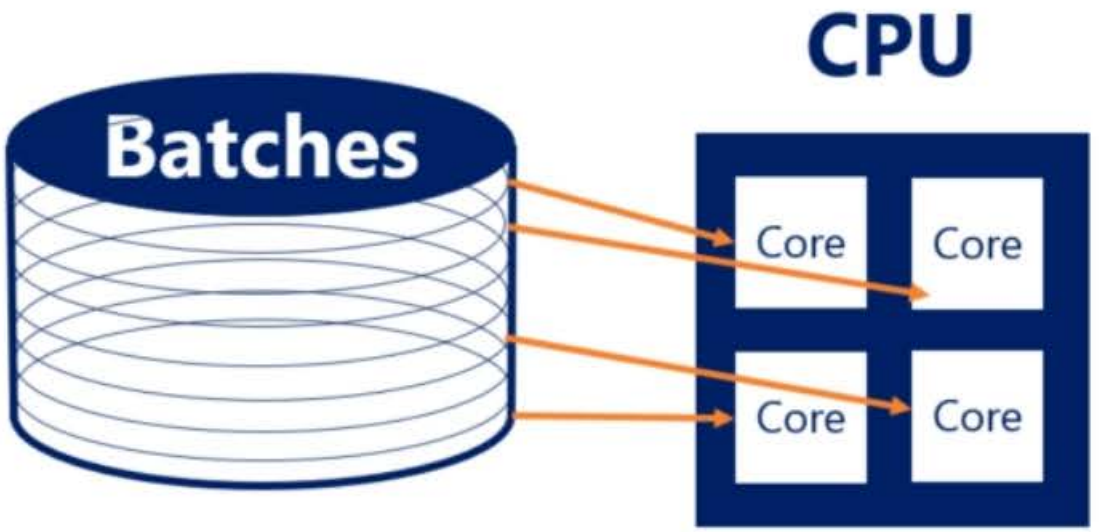




The SGD comes at a cost: it approximates things a bit

Batching





CPU or GPU cores train different batches simultaneously

Mini-batch GD

x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	
x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}	x_{28}
x_{31}	x_{32}	x_{33}	x_{34}	x_{35}	x_{36}	x_{37}	x_{38}
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
x_{n1}	x_{n2}	x_{n3}	x_{n4}	x_{n5}	x_{n6}	x_{n7}	x_{n8}

10,000

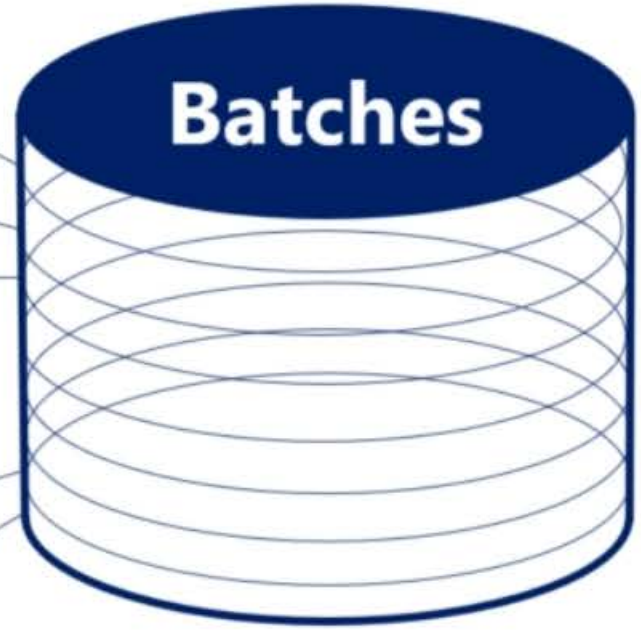
1,000

1,000

1,000

1,000

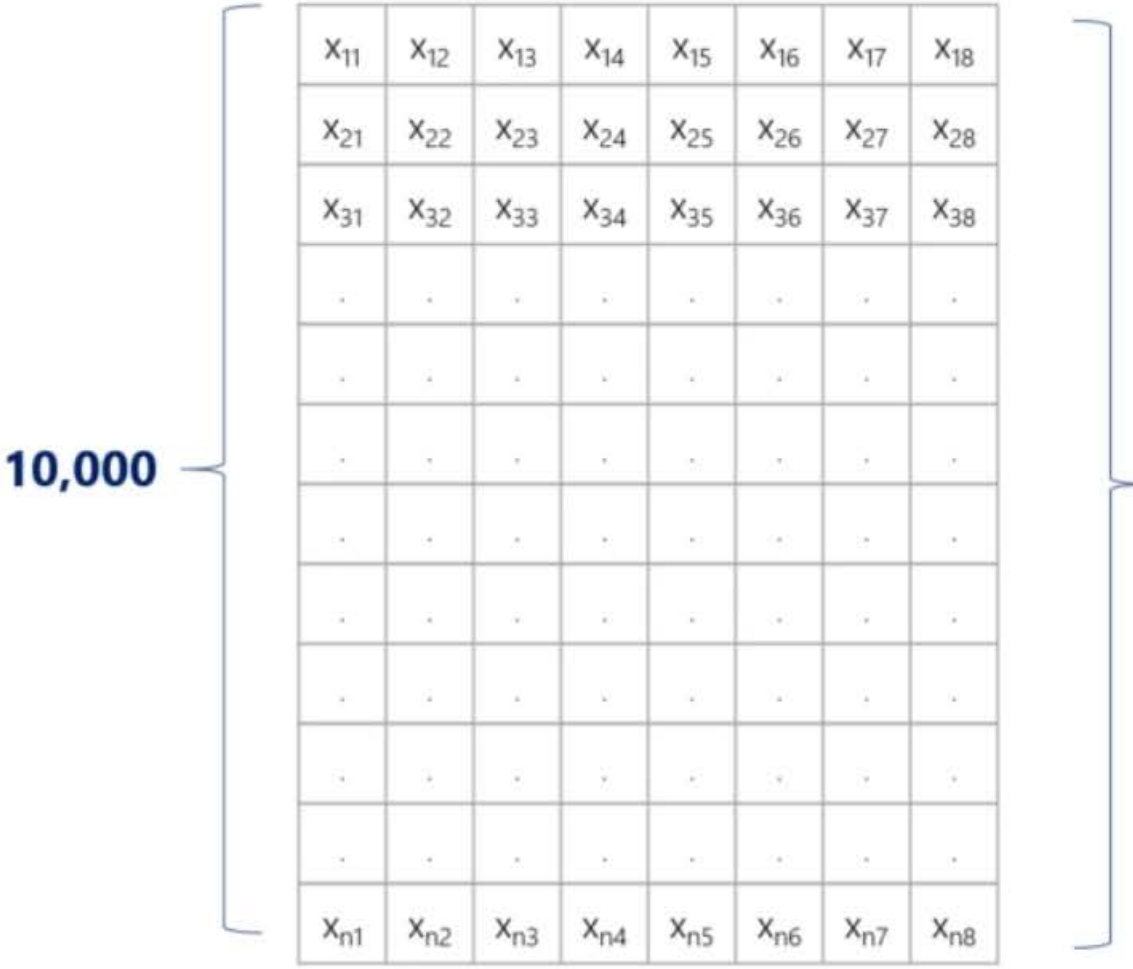
1,000

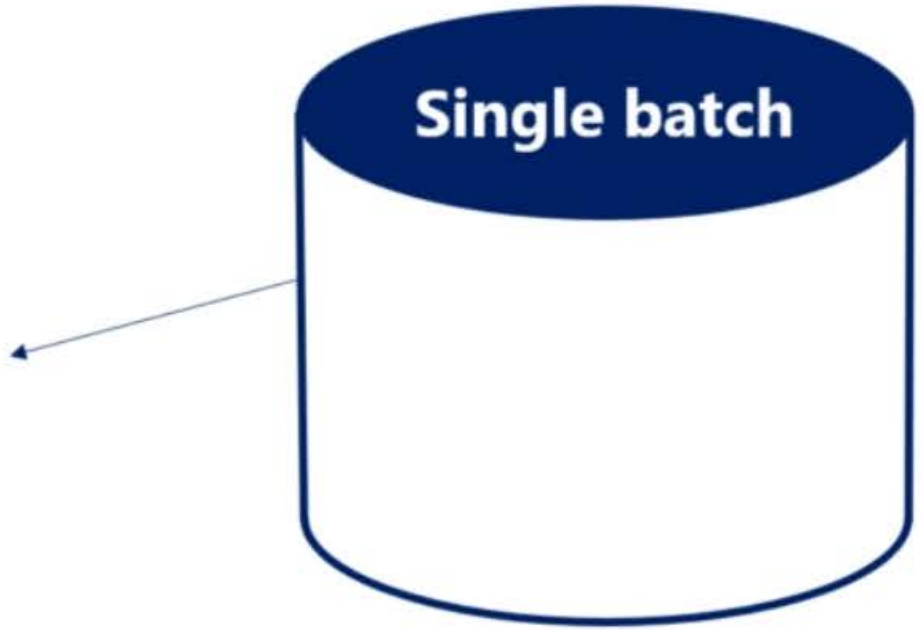


We update the weights after each **batch**!

Practitioners refer to the mini-batch GD as SGD

Batch GD



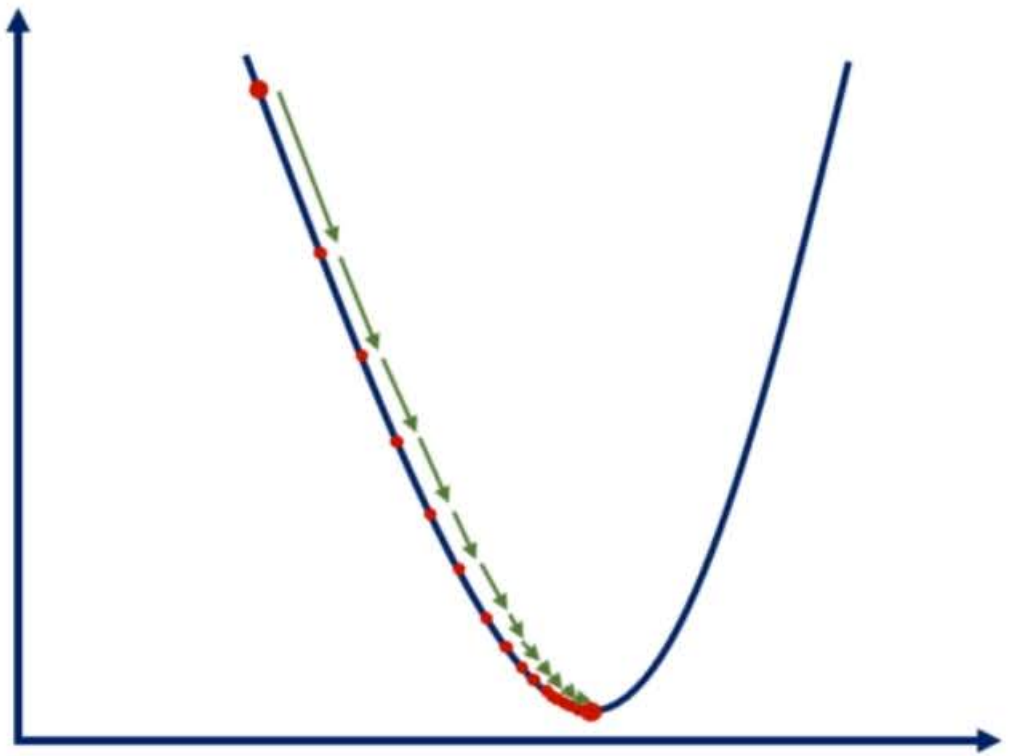


We update the weights once per epoch!

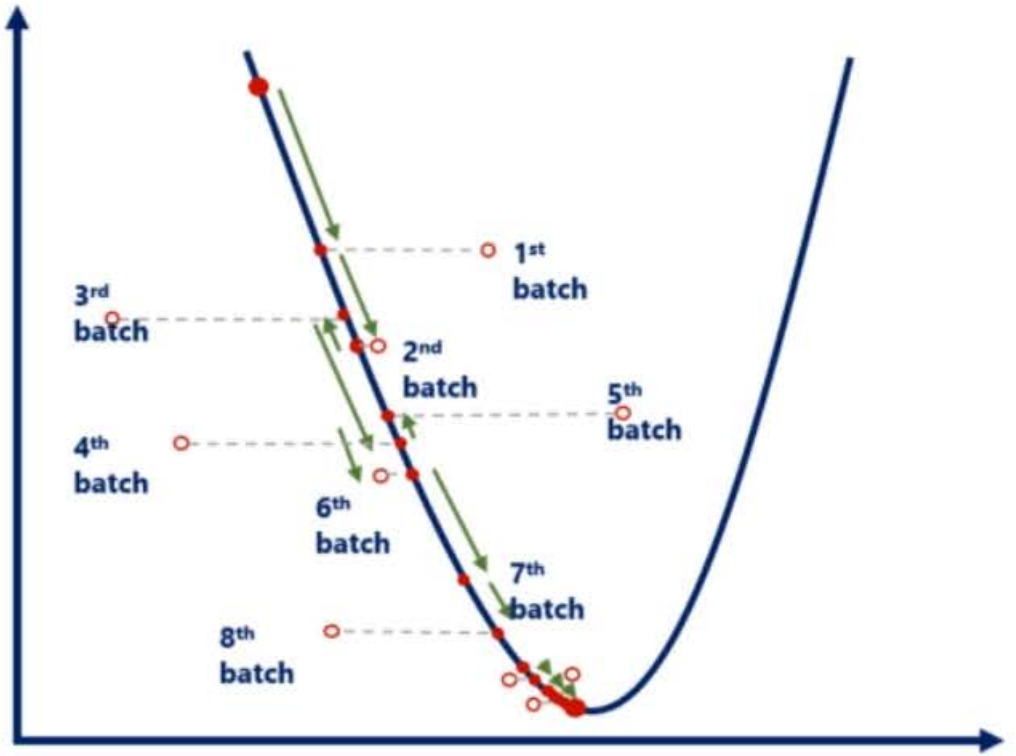
Practitioners refer to the mini-batch GD as SGD

Gradient descent pitfalls

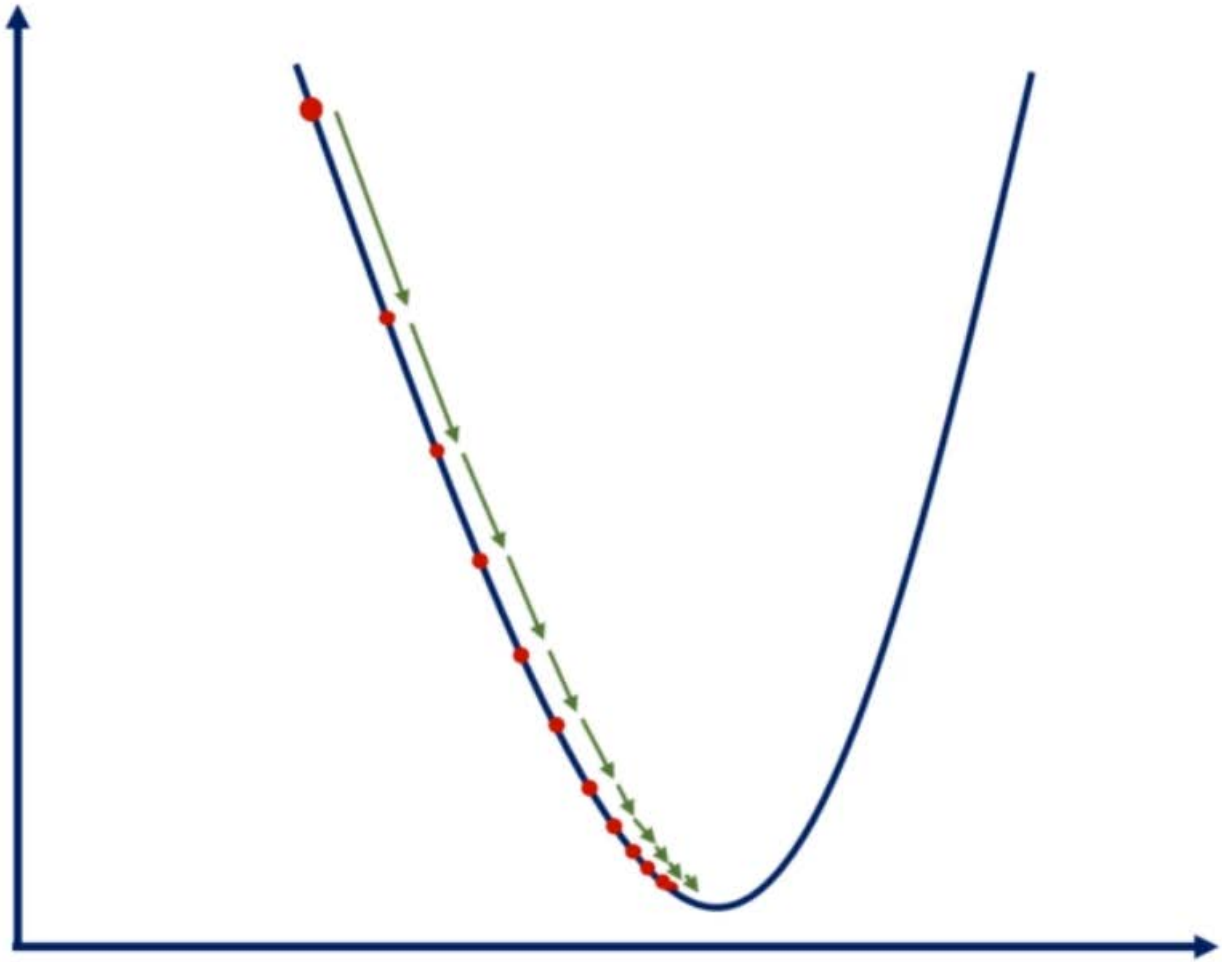
GD



SGD

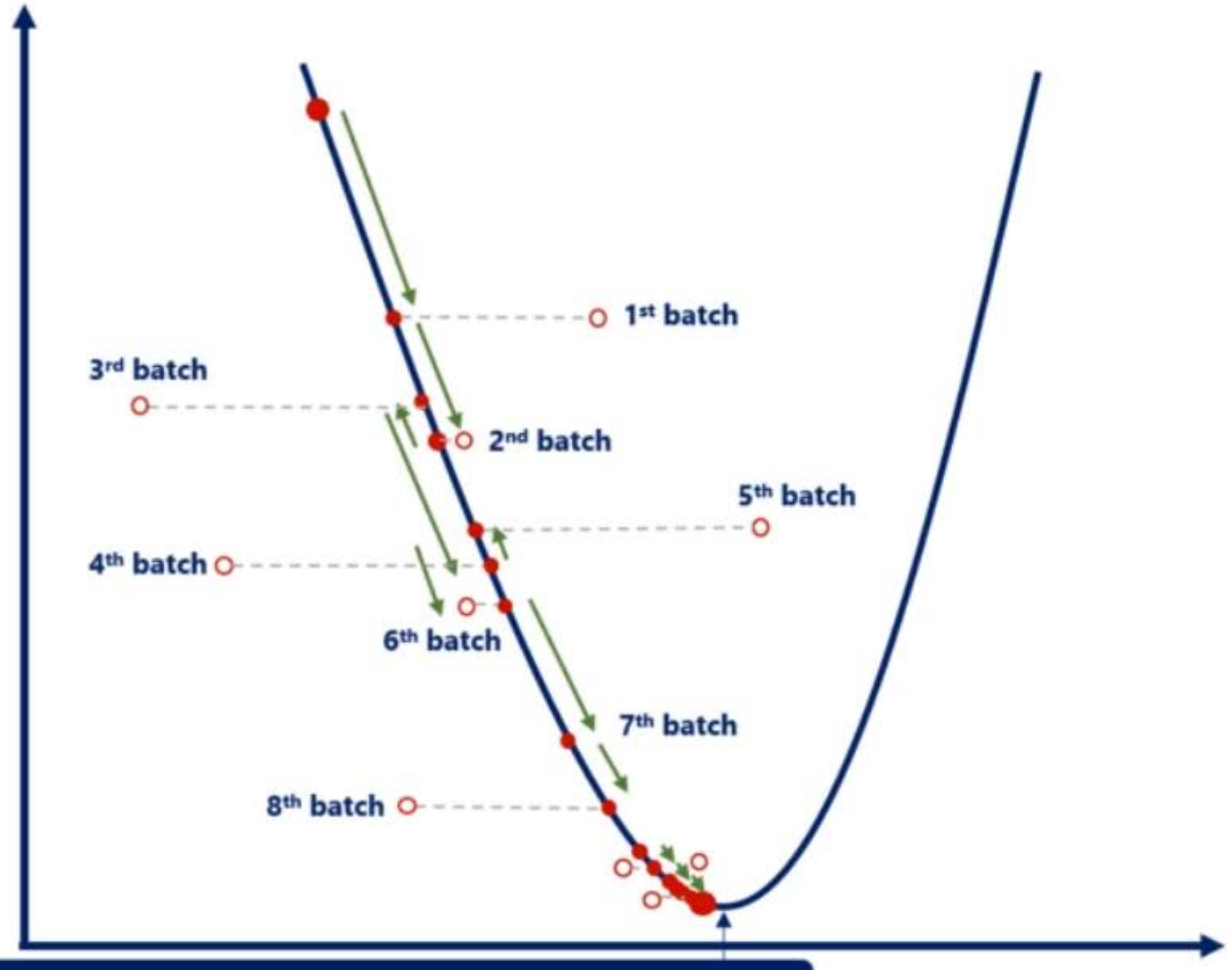


Gradient descent pitfalls



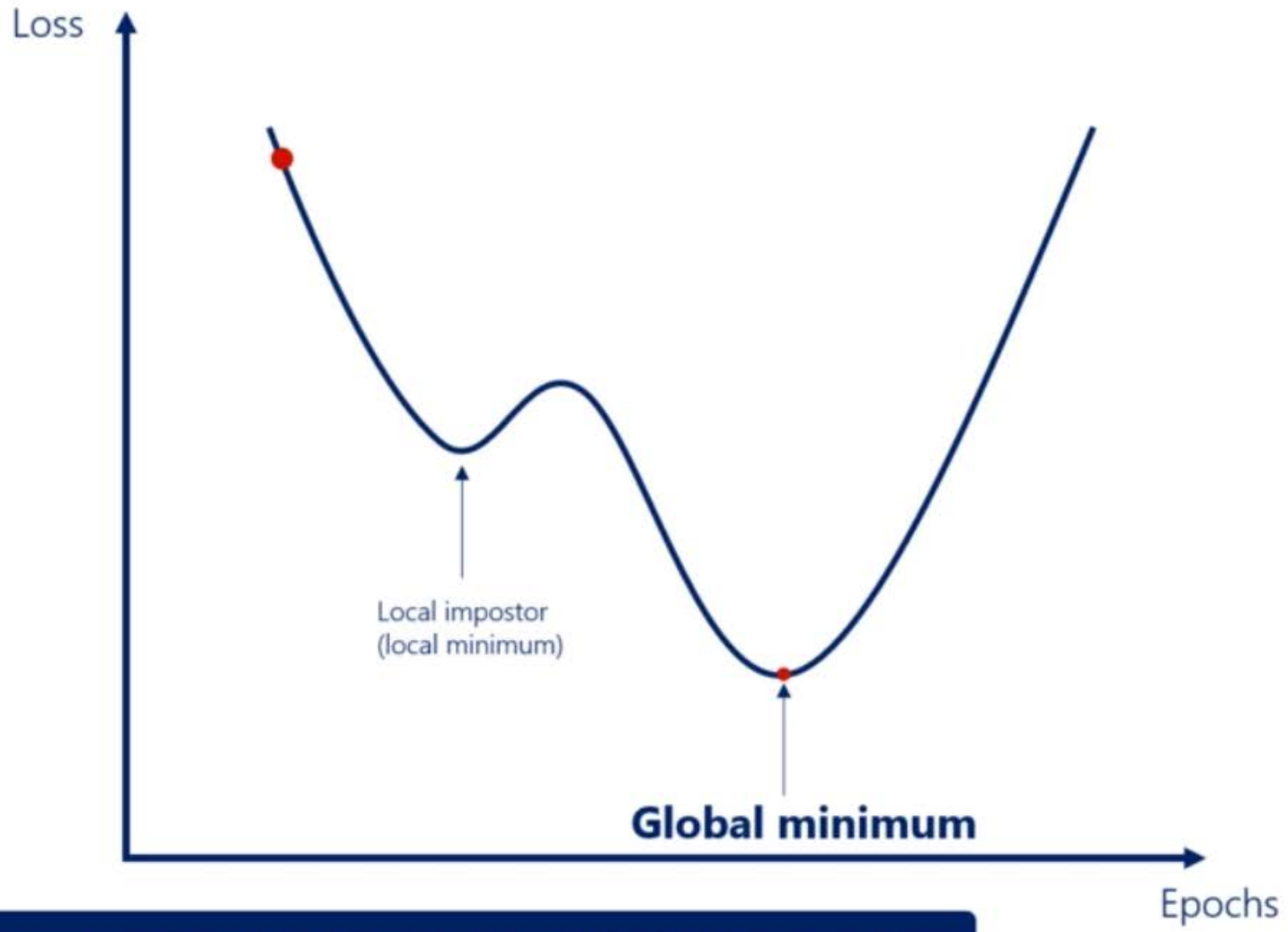
single batch GD would be slow, but will eventually reach the minimum

Gradient descent pitfalls



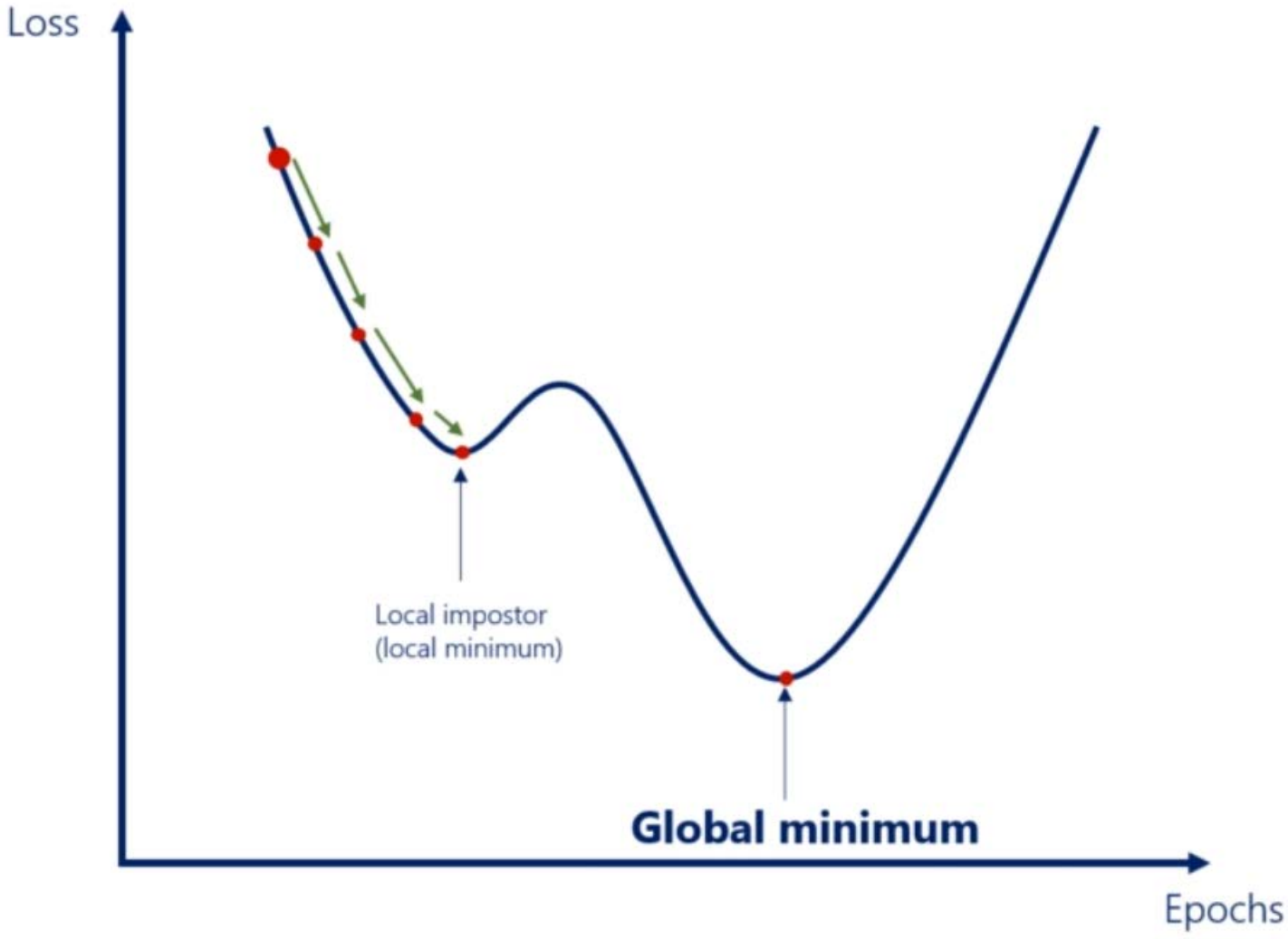
SGD would be much faster, but will give us an approximate answer

Gradient descent pitfalls

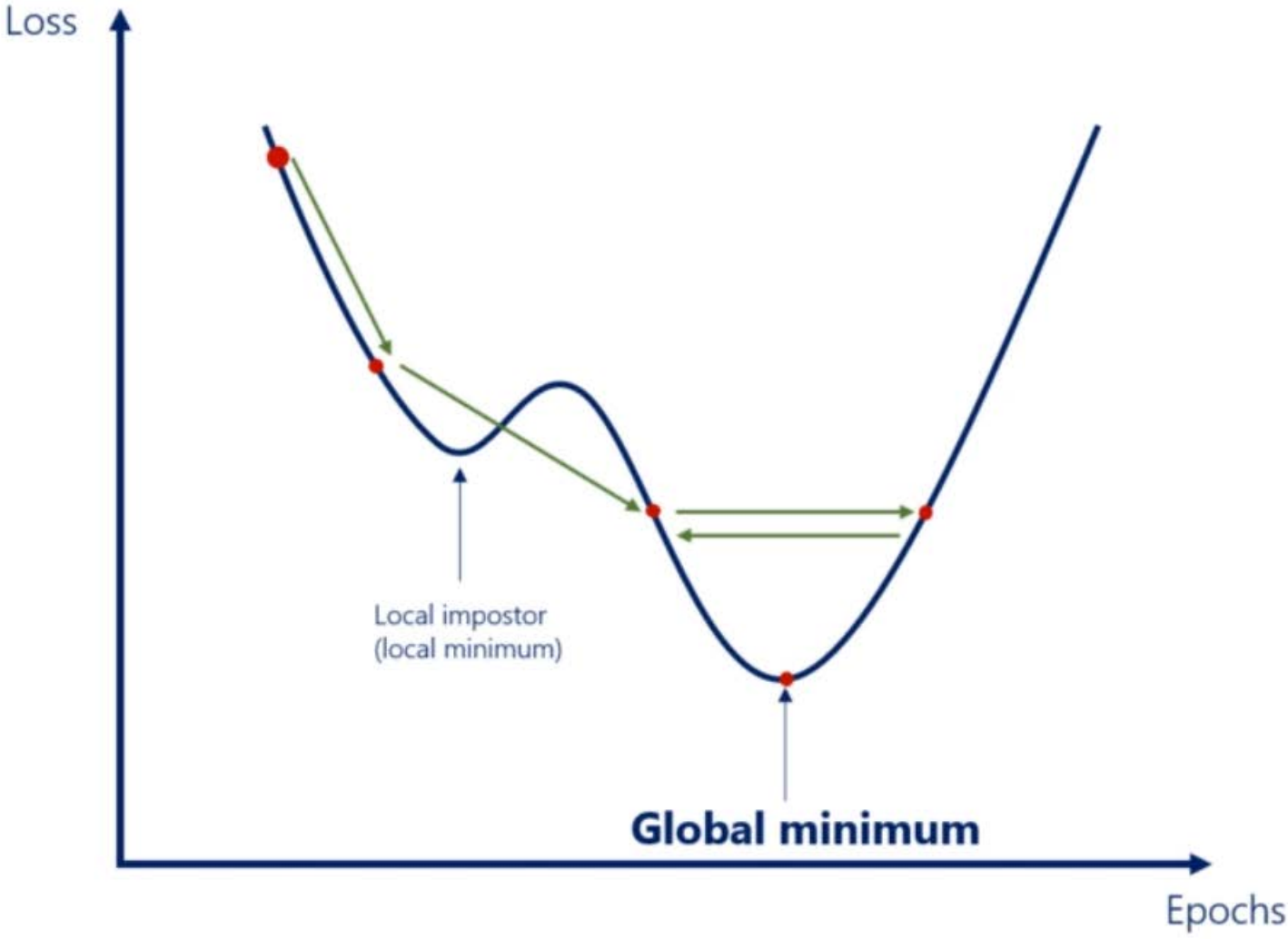


Each local minimum is a suboptimal solution to the optimization problem

Gradient descent pitfalls



Gradient descent pitfalls



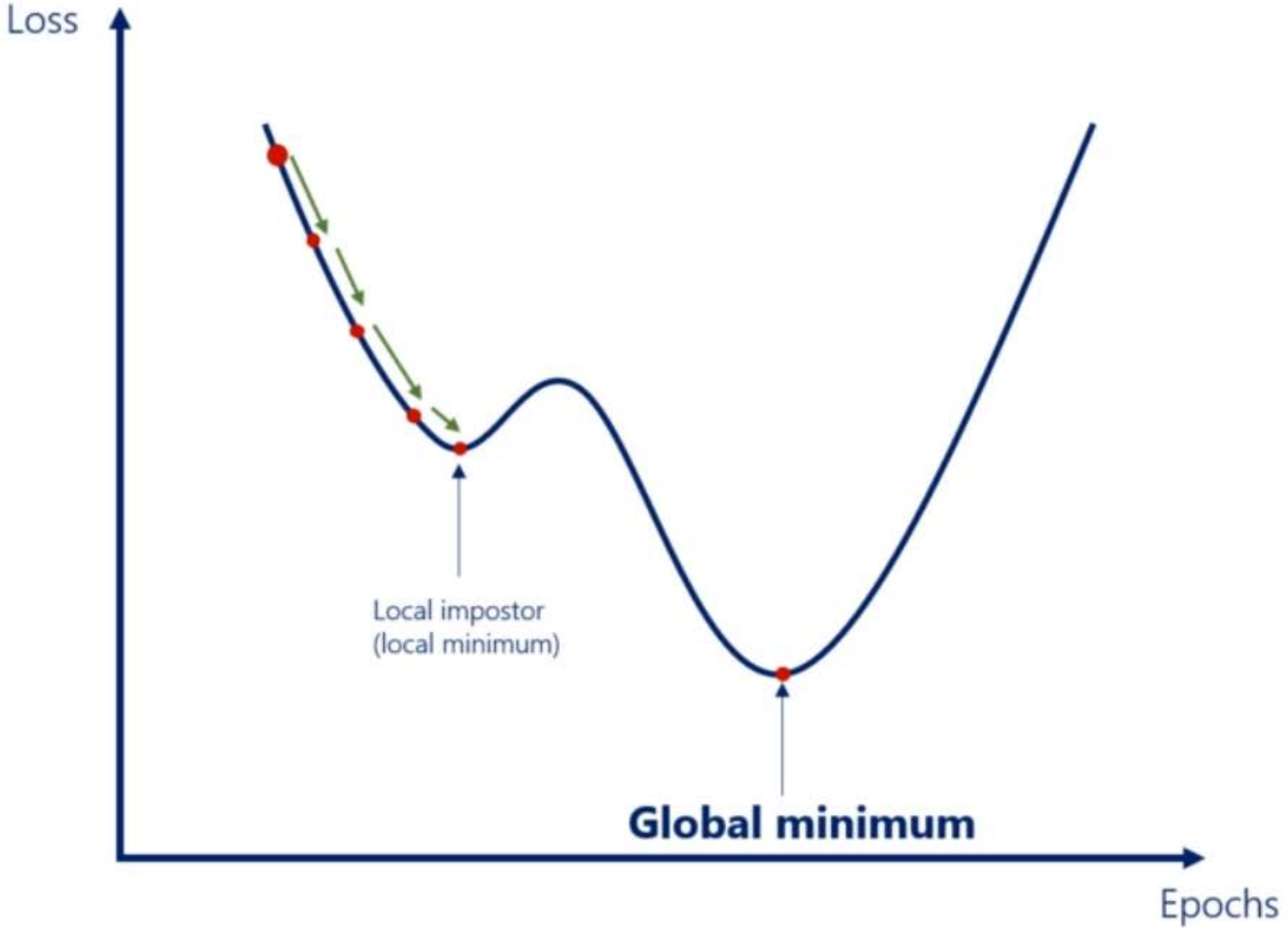
MOMENTUM

MOMENTUM

MOMENTUM

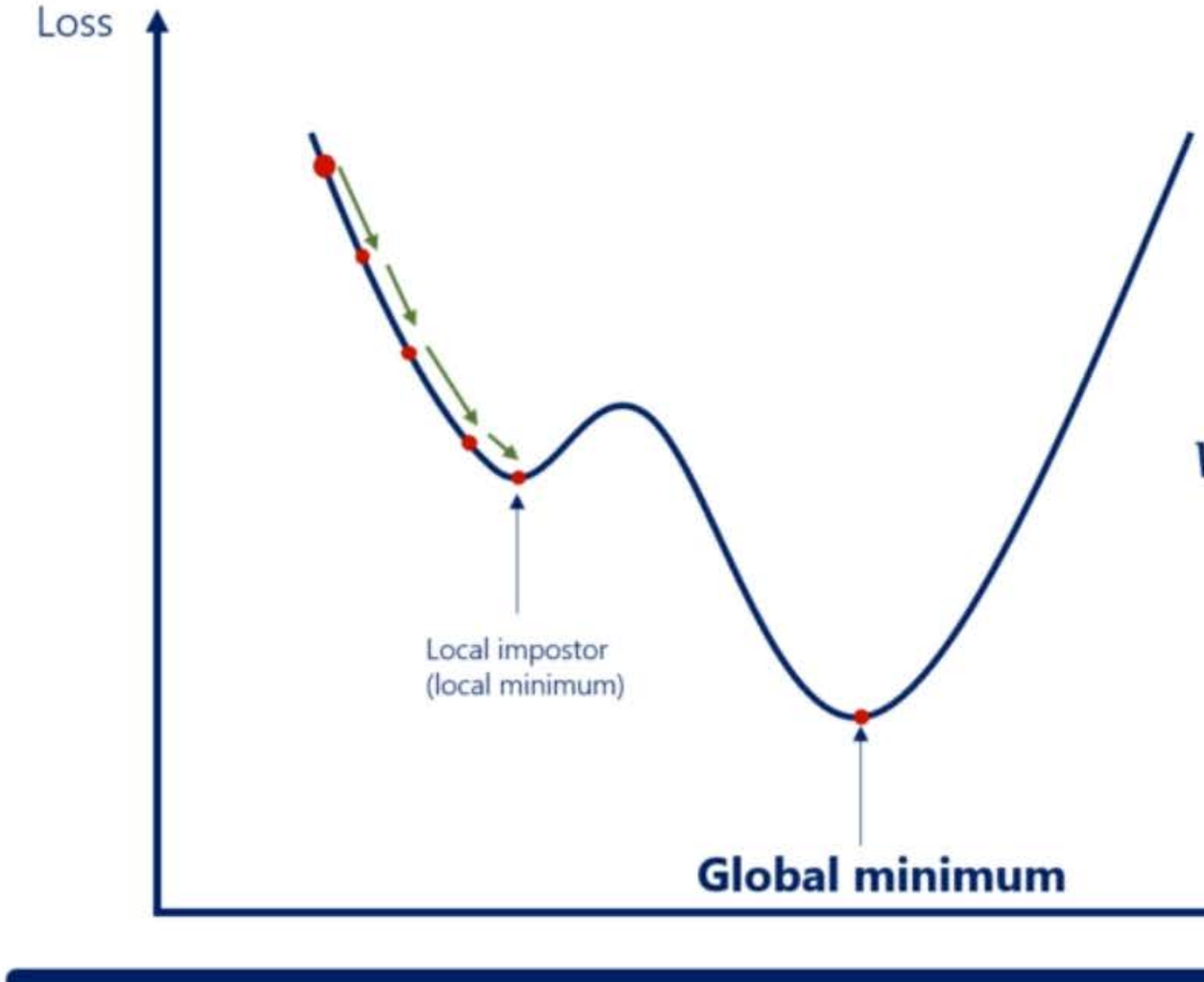
The momentum accounts for the fact that the ball actually going downhill

Momentum



$$w \leftarrow w - \eta \frac{\partial L}{\partial w}$$

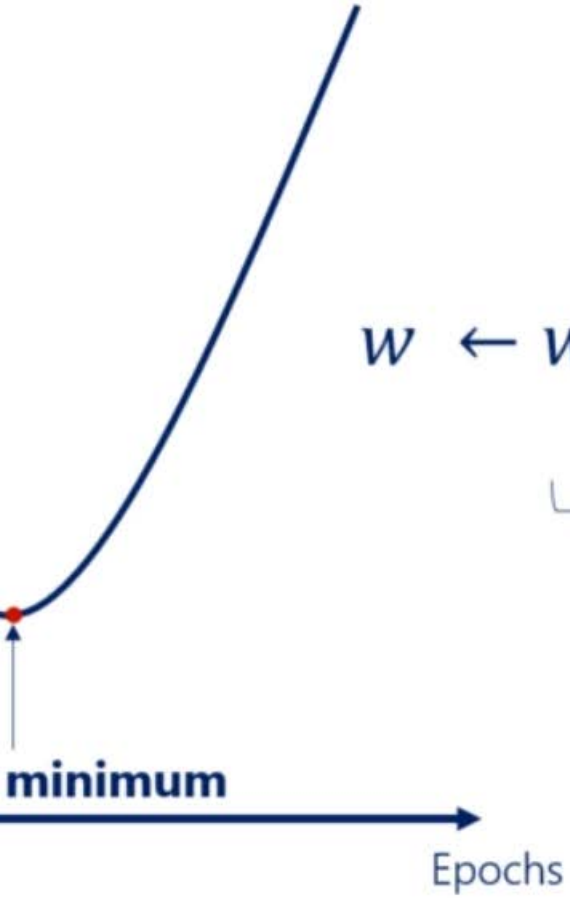
Momentum



$$w \leftarrow w(t) - \eta \frac{\partial L}{\partial w}(t) - \quad \eta \frac{\partial L}{\partial w}(t-1)$$

The best way to check how fast the ball rolls, is to check how fast it rolled a **moment** ago

Momentum

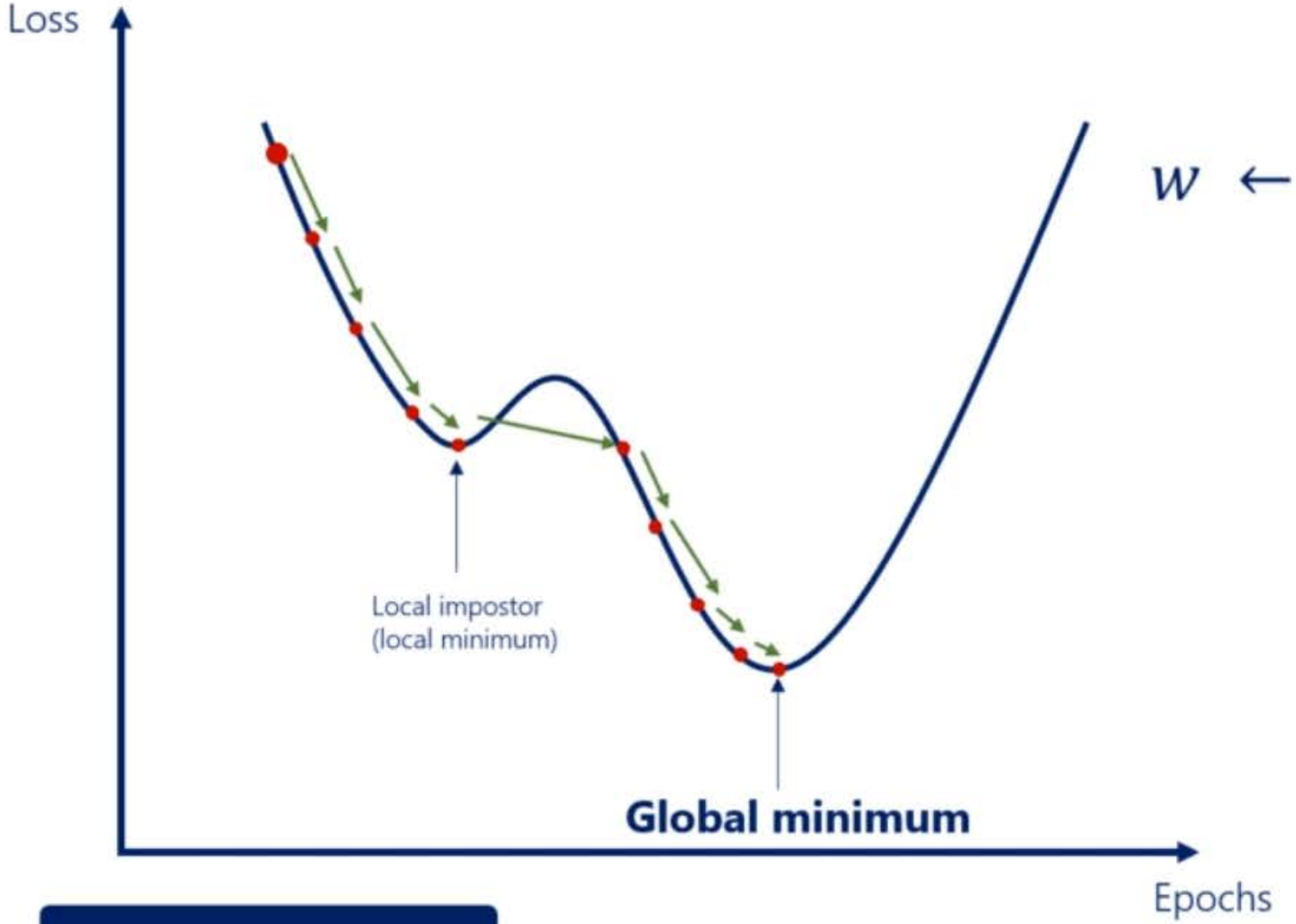


$$w \leftarrow w(t) - \eta \frac{\partial L}{\partial w}(t) - \eta \frac{\partial L}{\partial w}(t-1)$$

Current update

Update a moment ago

Momentum



$$w \leftarrow w(t) - \eta \frac{\partial L}{\partial w}(t) - \alpha \eta \frac{\partial L}{\partial w}(t-1)$$

Current update

Update a moment ago

$\alpha = 0.9$ is conventional

α is a hyperparameter

Hyperparameters vs Parameters

pre-set by us | **found by optimizing**

Width

Depth

Learning rate (η)

Weights (w)

Biases (b)

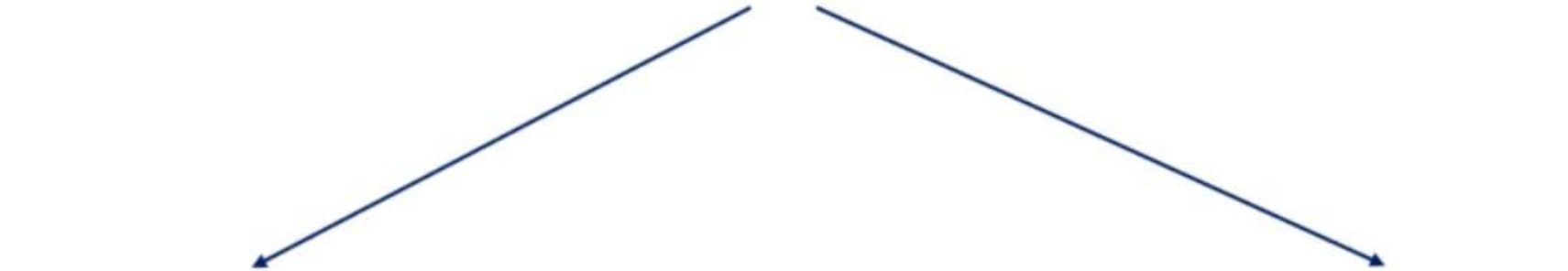
Learning rate (η)

Small enough so we gently descend, instead of oscillating or diverging

Big enough so we reach it in a rational amount of time

#datascience

Learning rate schedules



1. We start from a high initial learning rate

2. At some point we lower the rate to avoid oscillation

3. Around the end we pick a very small rate to get a precise answer

Learning rate schedules

1. We start from a high initial learning rate	First 5 epochs $\eta = 0.1$
2. At some point we lower the rate to avoid oscillation	Next 5 epochs $\eta = 0.01$
3. Around the end we pick a very small rate to get a precise answer	Until the end $\eta = 0.001$

A learning rate schedule causes the loss to converge much faster

Learning rate schedules

1. We start from a high initial learning rate

2. At some point we lower the rate to avoid oscillation

3. Around the end we pick a very small rate to get a precise answer

Exponential schedule. Still simple, but much better as it smoothly **decays** the learning rate

Learning rate schedules

1. We start from a high initial learning rate

2. At some point we lower the rate to avoid oscillation

3. Around the end we pick a very small rate to get a precise answer

$\eta_0 = 0.1$

current epoch

$$\eta = \eta_0 e^{-n/c}$$

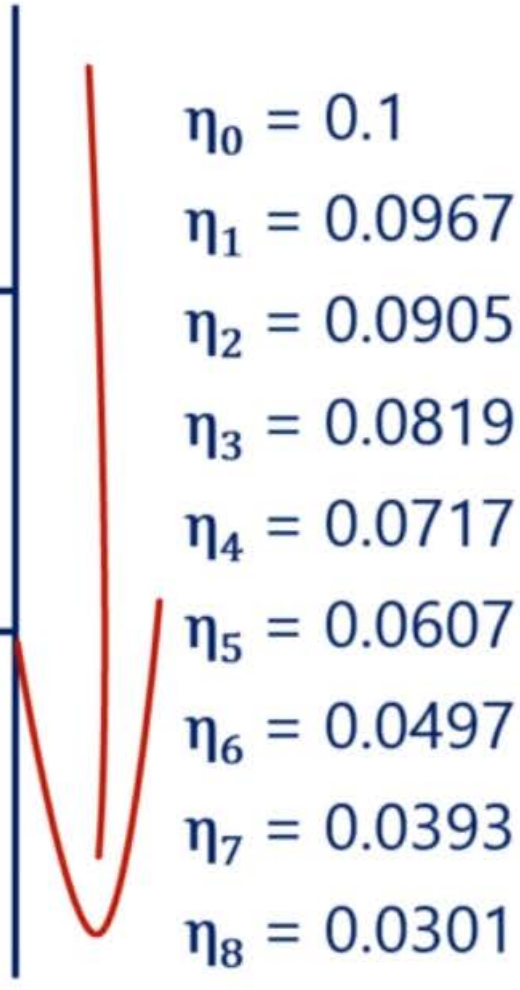
some constant

Learning rate schedules

1. We start from a high initial learning rate

2. At some point we lower the rate to avoid oscillation

3. Around the end we pick a very small rate to get a precise answer



$\eta_0 = 0.1$

$\eta_1 = 0.0967$

$\eta_2 = 0.0905$

$\eta_3 = 0.0819$

$\eta_4 = 0.0717$

$\eta_5 = 0.0607$

$\eta_6 = 0.0497$

$\eta_7 = 0.0393$

$\eta_8 = 0.0301$

current epoch

$$\eta = \eta_0 e^{-n/20}$$

some constant

Learning rate schedules

$\eta_0 = 0.1$

$\eta_1 = 0.0967$

$\eta_2 = 0.0905$

$\eta_3 = 0.0819$

$\eta_4 = 0.0717$

$\eta_5 = 0.0607$

$\eta_6 = 0.0497$

$\eta_7 = 0.0393$

$\eta_8 = 0.0301$

current epoch

$$\eta = \eta_0 e^{-n/20}$$

some constant

No set rule, but same order of magnitude

e.g. if we need:

100 epochs, 50 < c < 500
1000 epochs, 500 < c < 5000

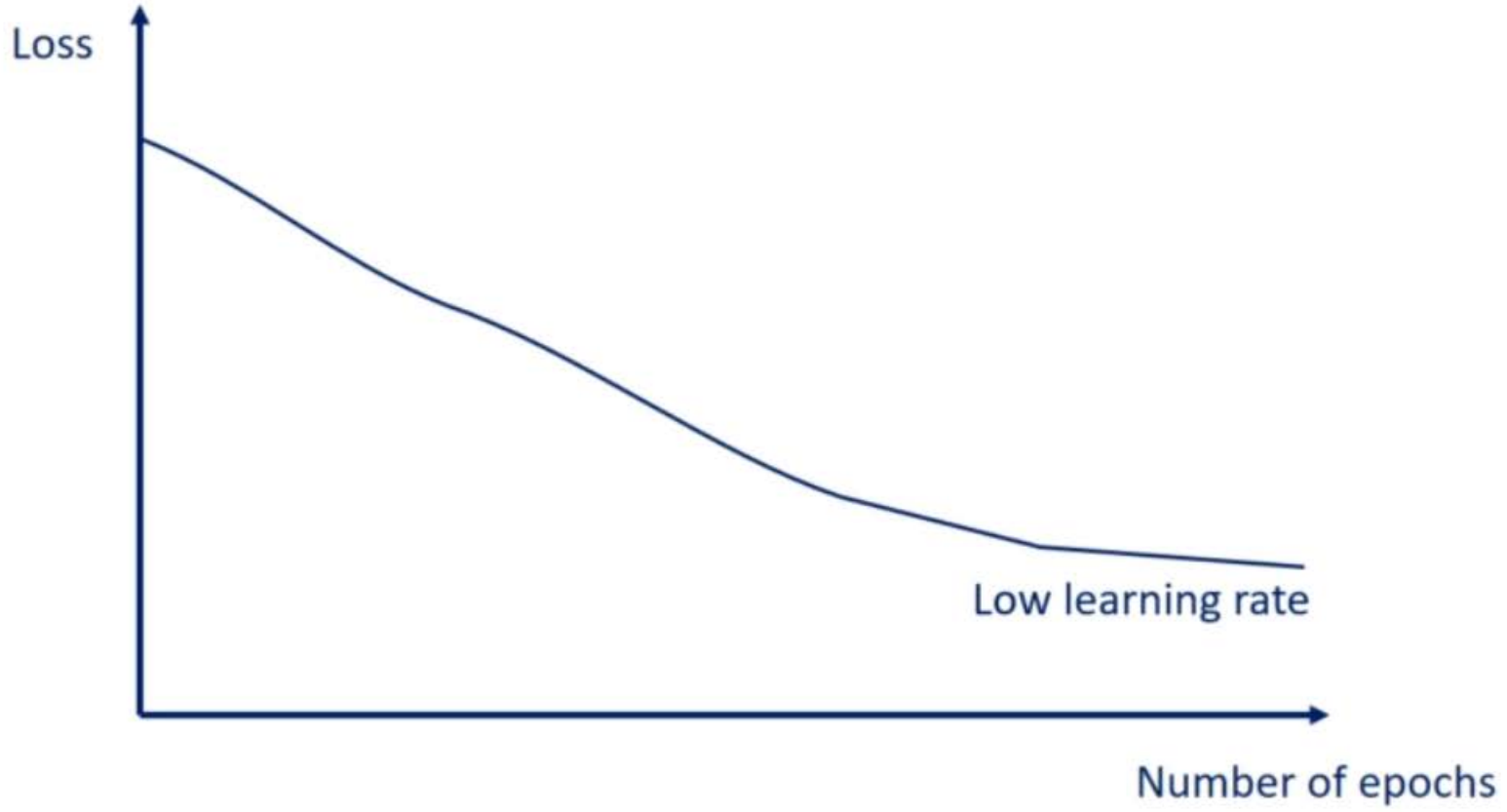
c ~ 20 is ok

The exact value is not important. **The presence of the learning schedule** does

Hyperparameters	vs	Parameters
pre-set by us		found by optimizing
Width		
Depth		Weights (w)
Learning rate (η)		
Batch size		Biases (b)
Momentum coefficient (α)		
Decay coefficient (c)		

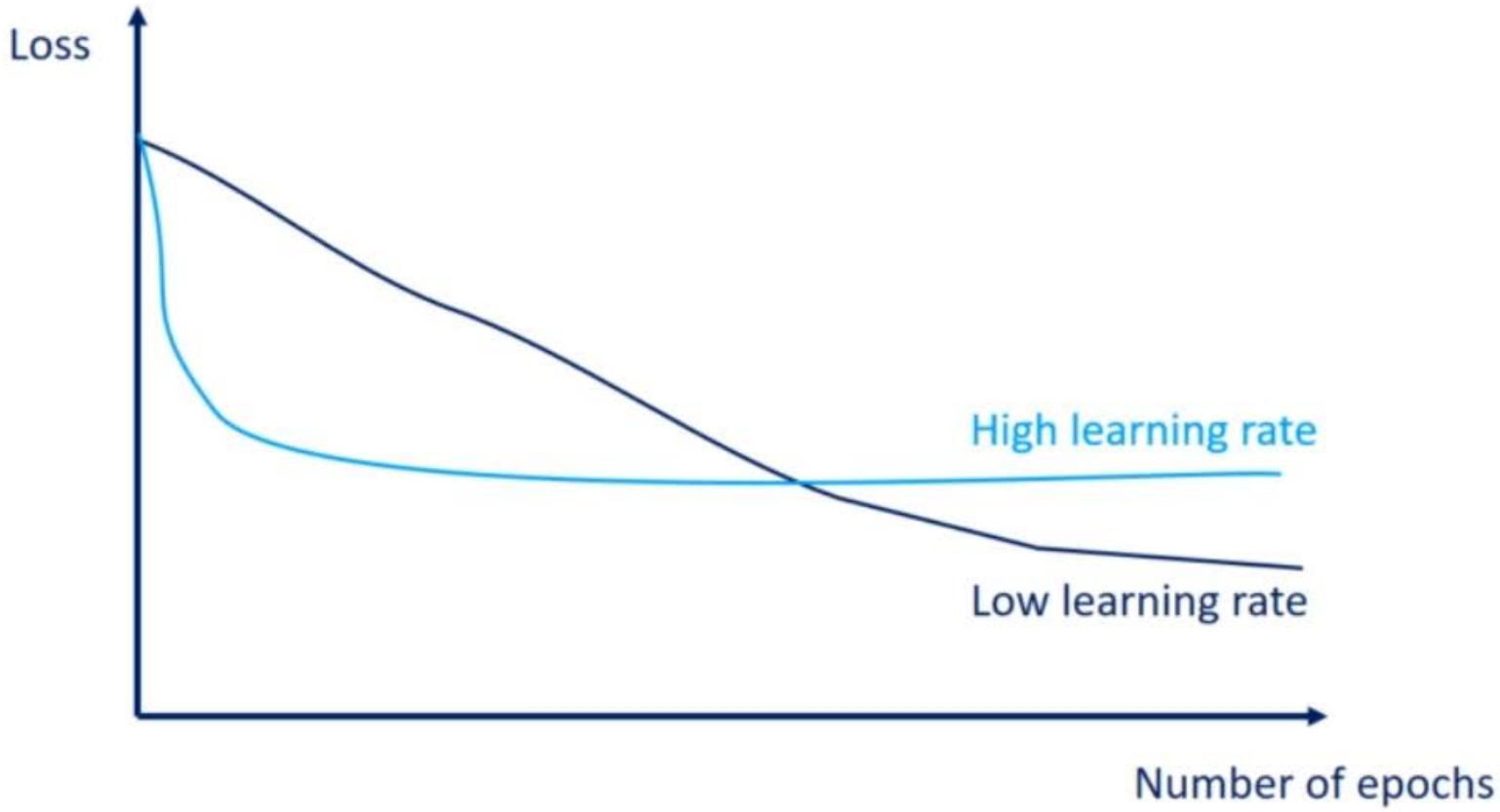
We increase the number of hyperparameters we must pick values for

Learning rate. A picture



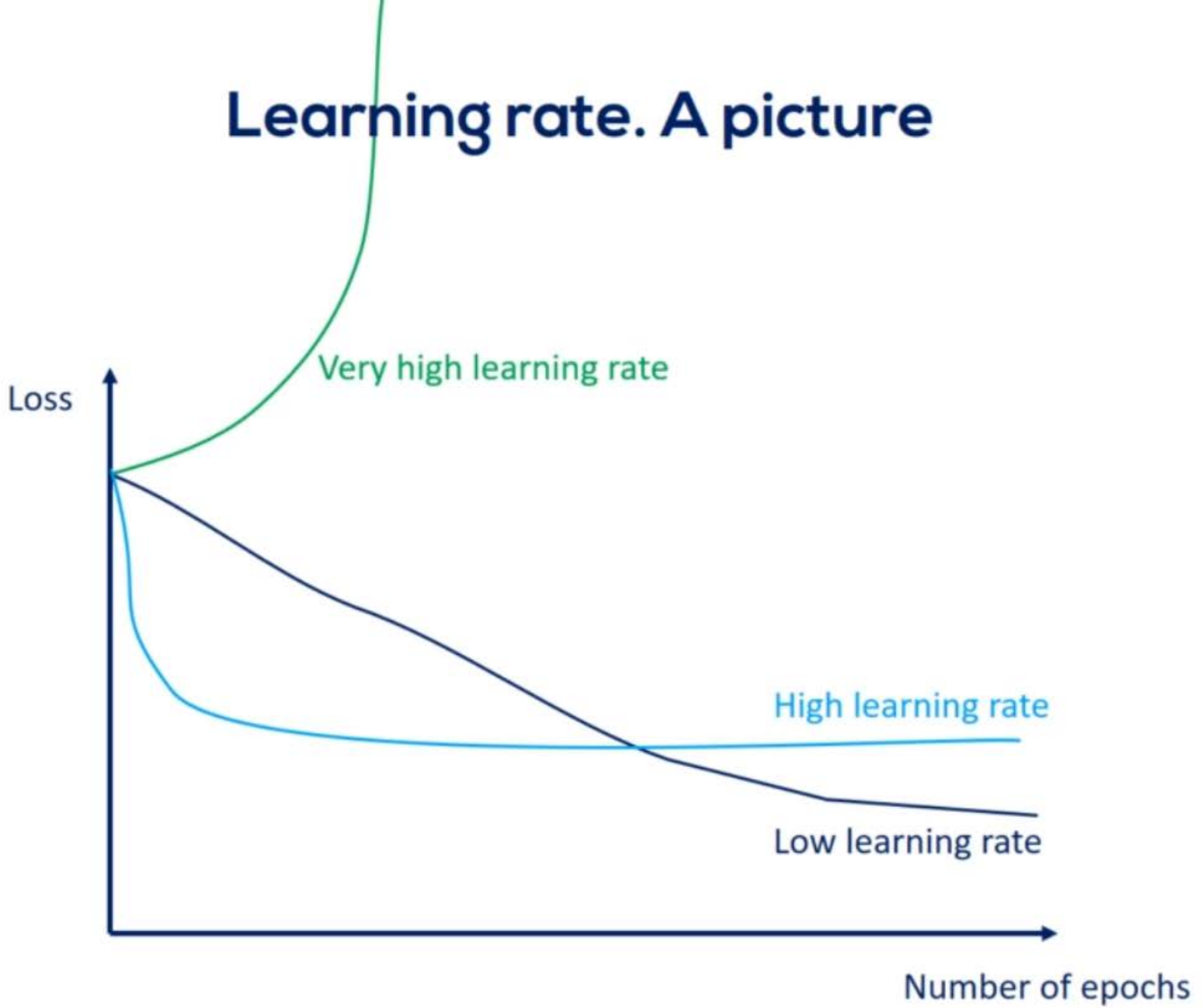
A low learning rate would minimize the loss but quite slowly

Learning rate. A picture



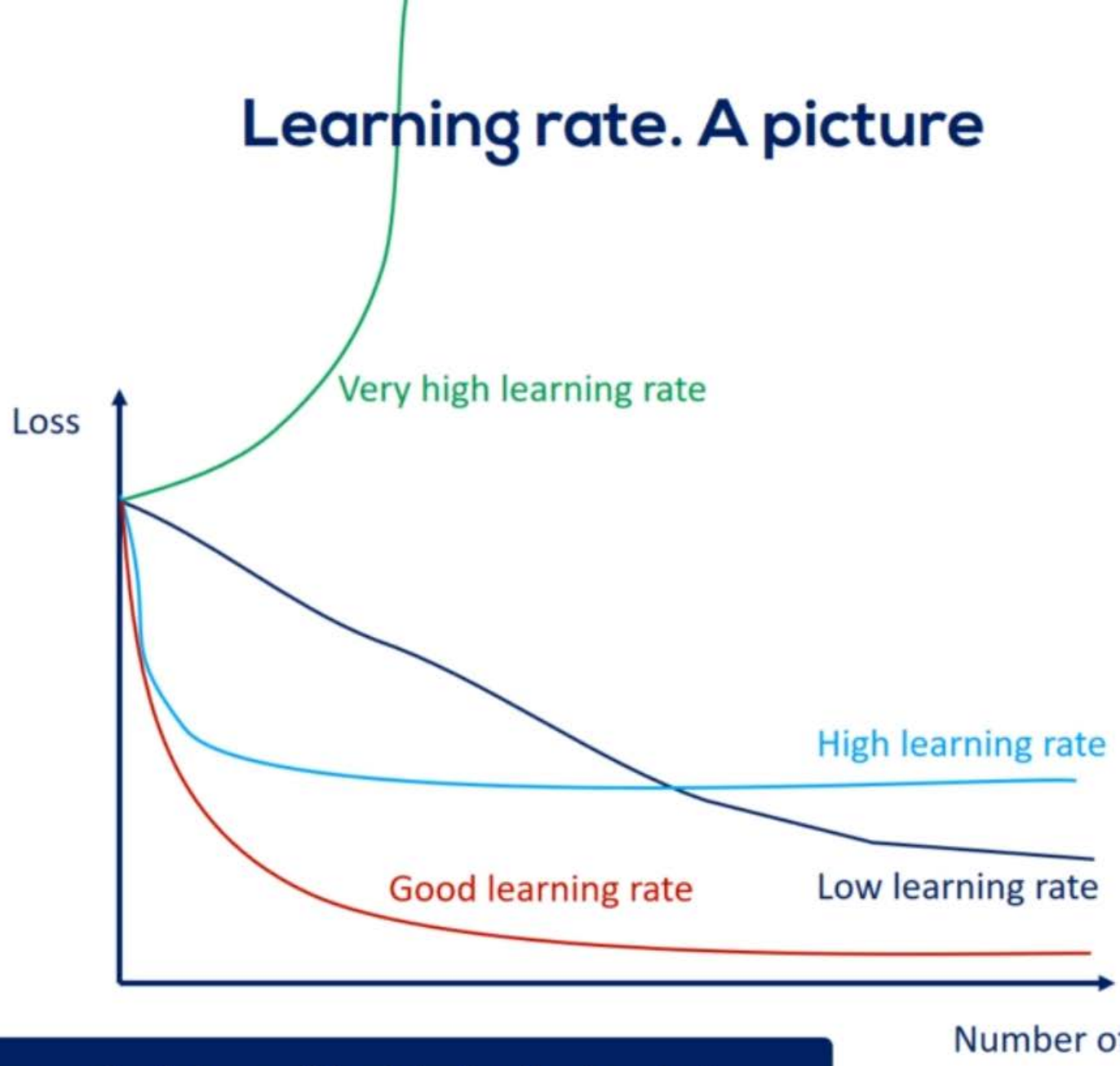
With a high learning rate, the loss is minimized faster but only to a certain extent

Learning rate. A picture



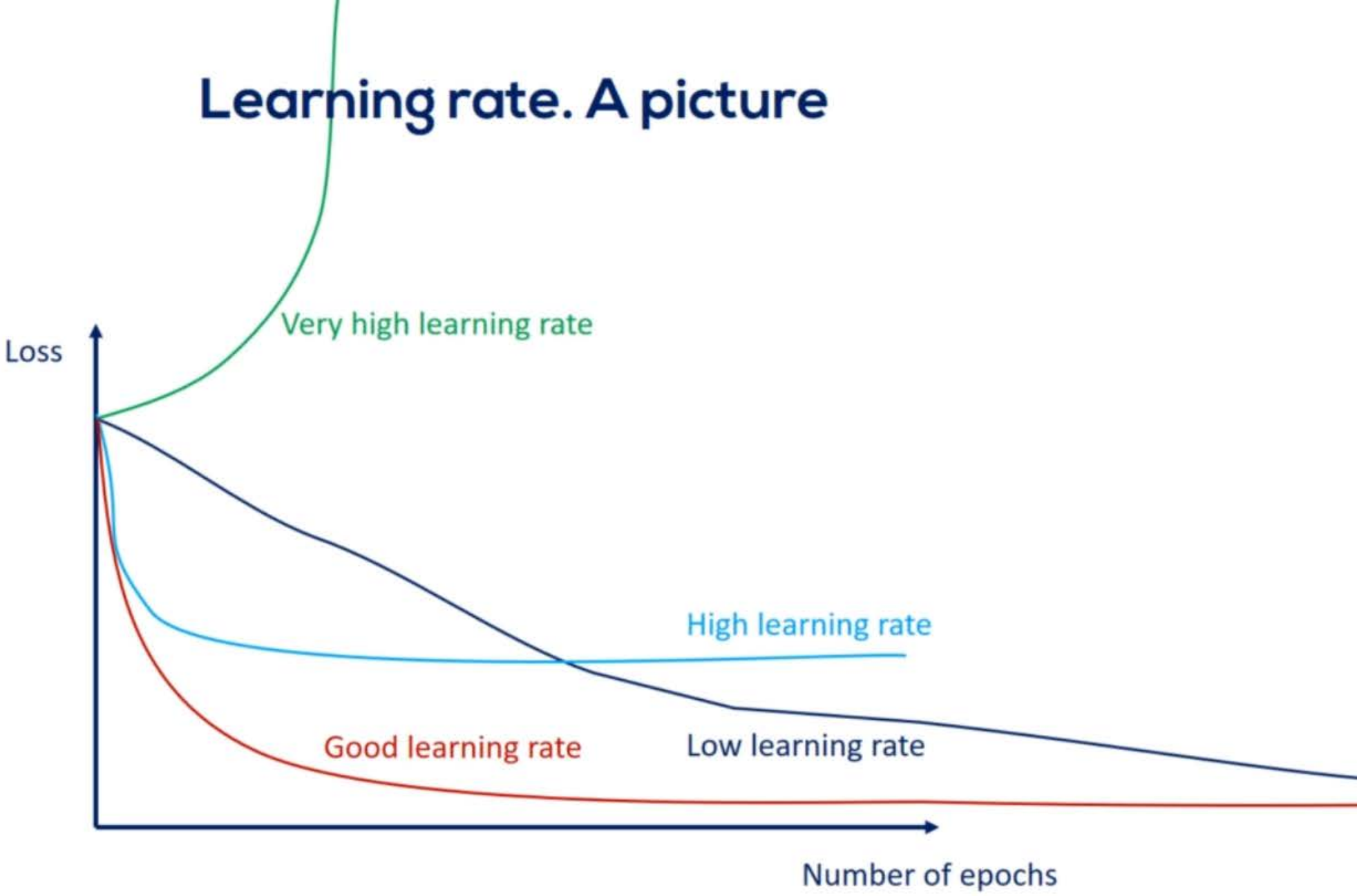
A **very** high learning rate would not even minimize the loss!

Learning rate. A picture



A learning rate following a schedule would minimize the loss faster than a low learning rate, and more accurately than a high learning rate

Learning rate. A picture



A low learning rate eventually converges with the good learning rate

Learning rate schedules

AdaGrad

RMSProp

TensorFlow

Just use
AdaGrad plz

AdaGrad

/adaptive gradient algorithm/

2011

It dynamically varies the learning rate at each update and for each weight **individually**

AdaGrad

/adaptive gradient algorithm/

$$w(t+1) = w(t) - \eta \frac{\partial L}{\partial w}(t)$$

The next weight (at time t+1)

The previous weight (at time t)

Follow the update rule

AdaGrad

/adaptive gradient algorithm/

$$w(t+1) = w(t) - \eta \frac{\partial L}{\partial w}(t)$$

$$w(t+1) - w(t) = -\eta \frac{\partial L}{\partial w}(t)$$

$$\Delta w = -\eta \frac{\partial L}{\partial w}(\mathrm{t})$$

AdaGrad

/adaptive gradient algorithm/

$$\Delta w = -\eta \frac{\partial L}{\partial w}(\mathrm{t})$$

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t) + \epsilon}} \frac{\partial L}{\partial w_i}(t)$$

AdaGrad

/adaptive gradient algorithm/

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t) + \epsilon}} \frac{\partial L}{\partial w_i}(t)$$

Weight index. The update rule is individual for each weight.

Iteration (time) at which we are updating

t dynamically varies the learning rate at each update and for each weight **individually**

AdaGrad

/adaptive gradient algorithm/

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t)} + \epsilon} \frac{\partial L}{\partial w_i}(t)$$

$$G_i(t) = G_i(t-1) + \left(\frac{\partial L}{\partial w_i}(t)\right)^2$$

with beginning point $G_i(0) = 0$

$G_i(0) = 0$

$G_i(1) = 0 + \text{non-neg}$

$G_i(2) = [0 + \text{non-neg}] + \text{non-neg}$

.
.
.

G (t) is monotonously increasing function (each consequent G is bigger or equal to the previous one)

AdaGrad

/adaptive gradient algorithm/

effective learning rate

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t) + \epsilon}} \frac{\partial L}{\partial w_i}(t)$$

$$G_i(t) = G_i(t-1) + \left(\frac{\partial L}{\partial w_i}(t)\right)^2$$

with beginning point $G_i(0) = 0$

$$G_i(0) = 0$$

$$G_i(1) = 0 + \text{non-neg}$$

$$G_i(2) = [0 + \text{non-neg}] + \text{non-neg}$$

.
.
.

Eta / mon. increasing function = mon. decreasing function

AdaGrad

/adaptive gradient algorithm/

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t)} + \epsilon} \frac{\partial L}{\partial w_i}(t)$$

$$G_i(t) = G_i(t-1) + \left(\frac{\partial L}{\partial w_i}(t) \right)^2$$

with beginning point $G_i(0) = 0$

- Smart
- Adaptive learning rate schedule
- Based on the training itself
- **Per weight**

AdaGrad

/adaptive gradient algorithm/

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t)} + \epsilon} \frac{\partial L}{\partial w_i}(t)$$

$$G_i(t) = G_i(t-1) + \left(\frac{\partial L}{\partial w_i}(t)\right)^2$$

with beginning point $G_i(0) = 0$

- Smart
- Adaptive learning rate schedule
- Based on the training itself
- **Per weight**

Different weights do not reach their optimal values simultaneously

AdaGrad

/adaptive gradient algorithm/

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t)} + \epsilon}\frac{\partial L}{\partial w_i}(t)$$

$$G_i(t) = G_i(t-1) + \left(\frac{\partial L}{\partial w_i}(t)\right)^2$$

with beginning point $G_i(0) = 0$

RMSprop

/root mean square propagation/

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t)} + \epsilon}\frac{\partial L}{\partial w_i}(t)$$

$$G_i(t) = \beta G_i(t-1) + (1-\beta)\left(\frac{\partial L}{\partial w_i}(t)\right)^2$$

with beginning point $G_i(0) = 0$

β – yet another hyperparameter
usually, around ~0.9

AdaGrad

/adaptive gradient algorithm/

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t)} + \epsilon}\frac{\partial L}{\partial w_i}(t)$$

$$G_i(t) = G_i(t-1) + \left(\frac{\partial L}{\partial w_i}(t)\right)^2$$

with beginning point $G_i(0) = 0$

RMSprop

/root mean square propagation/

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t)} + \epsilon}\frac{\partial L}{\partial w_i}(t)$$

$$G_i(t) = \beta G_i(t-1) + (1-\beta)\left(\frac{\partial L}{\partial w_i}(t)\right)^2$$

with beginning point $G_i(0) = 0$

β – yet another hyperparameter
usually, around ~0.9

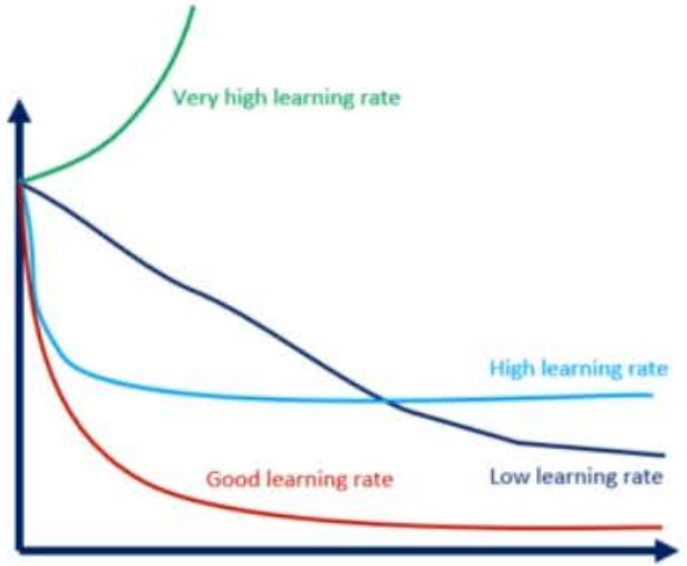
No longer monotonous, so it can adapt **upwards and downwards**

AdaGrad, RMSprop

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t) + \epsilon}} \frac{\partial L}{\partial w_i}(t)$$

- Smart
- Adaptive learning rate schedule
- Based on the training itself
- **Per weight**

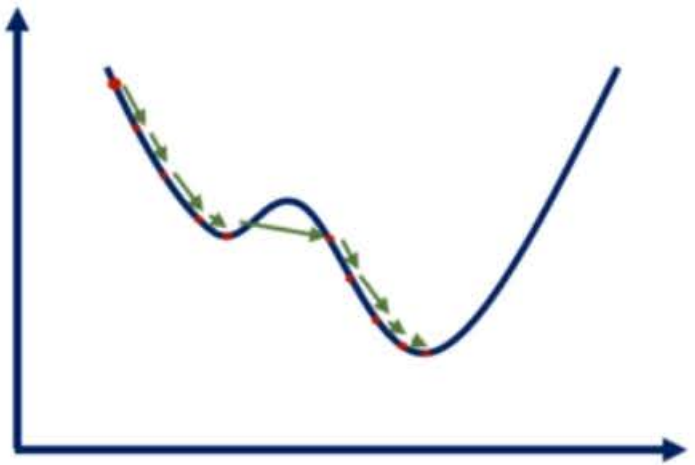
Learning rate schedules



$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t) + \epsilon}} \frac{\partial L}{\partial w_i}(t)$$

AdaGrad, RMSprop

Momentum



$$w \leftarrow w(t) - \eta \frac{\partial L}{\partial w}(t) - \alpha \eta \frac{\partial L}{\partial w}(t-1)$$

Current update

Update a moment ago

Adam

/adaptive moment estimation/

The most advanced optimizer (very fast and efficient)

Adam

/adaptive moment estimation/

RMSprop

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t)} + \epsilon}\frac{\partial L}{\partial w_i}(t)$$

Momentum

$$w \leftarrow w(t) - \eta\frac{\partial L}{\partial w}(t) - \alpha\eta\frac{\partial L}{\partial w}(t-1)$$

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t)} + \epsilon} M_i(t)$$

$$M_i(t) = \alpha M_i(t-1) + (1-\alpha)\frac{\partial L}{\partial w_i}(t)$$

$$M_i(0) = 0$$

PREPROCESSING

Any manipulation of the dataset before running it through the model

PREPROCESSING

motivation

- Compatibility

TensorFlow

PREPROCESSING

motivation

- Compatibility

Exchange rate	Volume
~1	~100,000

- Orders of magnitude

Mathematically 1 is negligible w.r.t. 100,000

(~100,000) + (~1) = ~100,000

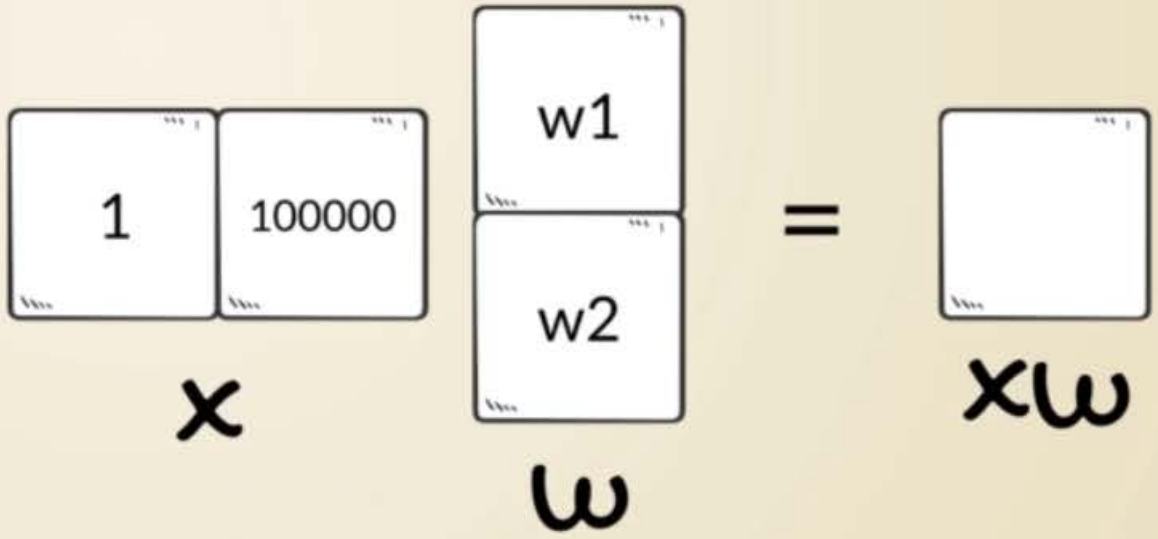
PREPROCESSING

motivation

- Compatibility

Exchange rate	Volume
~1	~100,000

- Orders of magnitude



100,000 affects the result much more than the 1

PREPROCESSING

motivation

- Compatibility
- Orders of magnitude
- Generalization

Same model,
different issue

Google

apple stock price

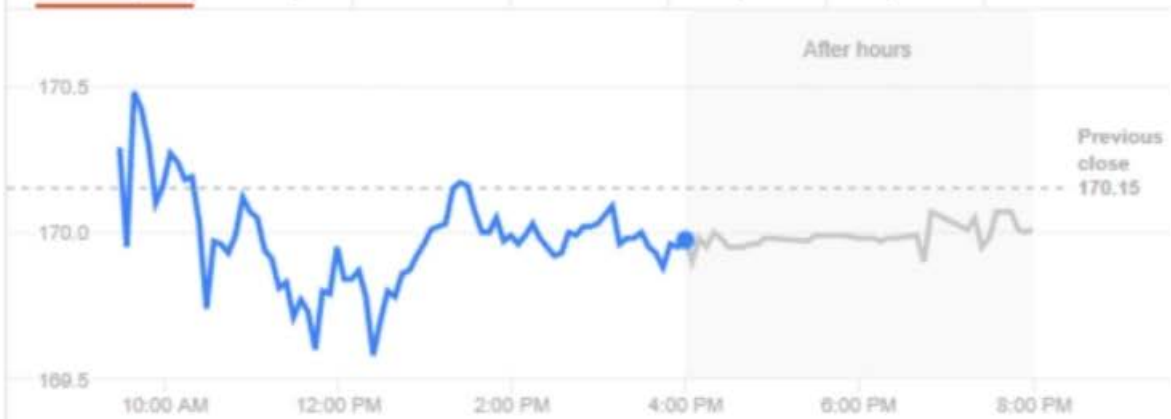
All News Images Videos Maps More Settings Tools

About 14,000,000 results (0.69 seconds)

Apple Inc.
NASDAQ: AAPL

169.98 USD ↓0.17 (0.10%)
After-hours: 170.59 ↑0.61 (0.36%)

1 day | 5 day | 1 month | 3 months | 1 year | 5 years | max



Open	170.29	Mkt cap	872.73B
High	170.56	P/E ratio	18.5
Low	169.56	Div yield	1.48%

Google Finance - Yahoo Finance - MSN Money Disclaimer

Top stories

Apple
Technology company

Apple Inc. is an American multinational techn
headquartered in Cupertino, California that de
consumer electronics, computer software, an

Customer service: 80060027753
Headquarters: Cupertino, California, United
Founded: April 1, 1976, Cupertino, California
Founders: Steve Jobs, Steve Wozniak, Rona
Subsidiaries: Beats Electronics, Apple Store
Did you know: Apple is the world's ninth-larg
wikipedia.org

Profiles

YouTube LinkedIn Twitter

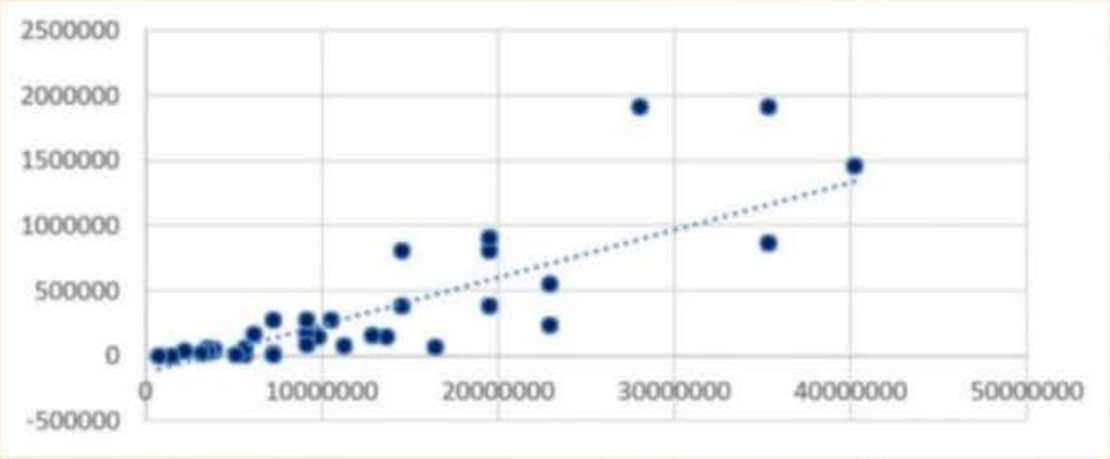
People also search for

TESLA NOKIA SAMSUNG

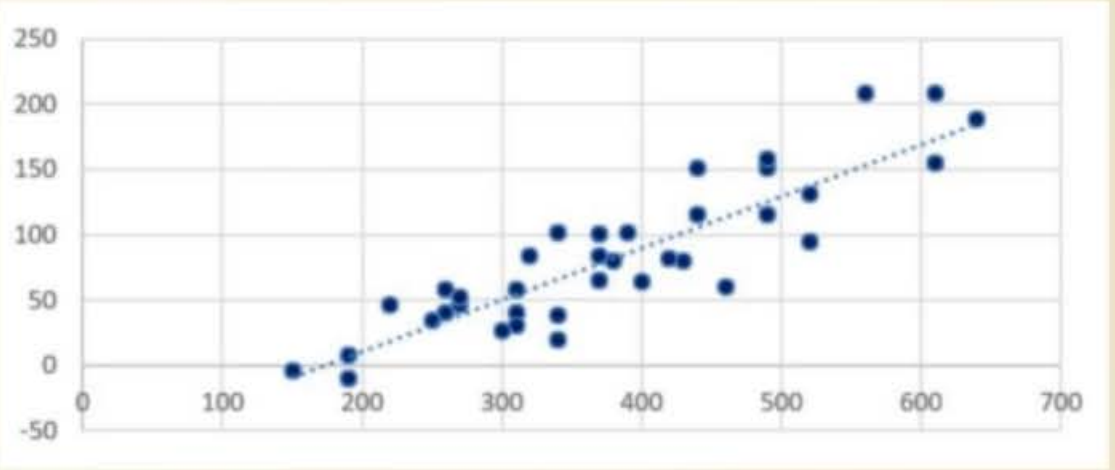
Relative metrics are especially useful when we have time-series data

LOGARITHMS

Financial data



Log transformed financial data



Advantages:

Faster computation

Lower order of magnitude

Clearer relationships

Homogeneous variance

STANDARDIZATION

FEATURE SCALING

NORMALIZATION

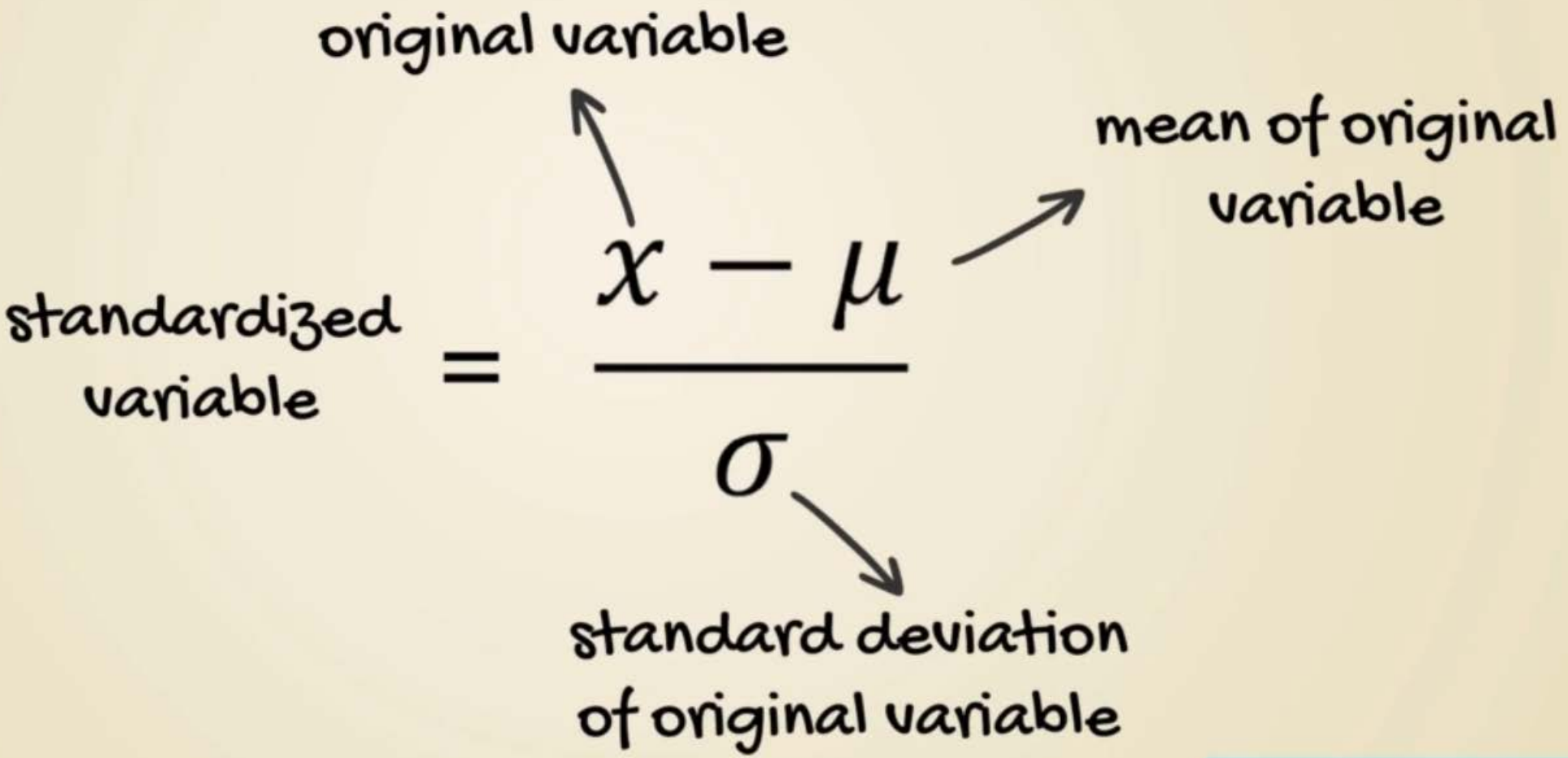
STANDARDIZATION

FEATURE SCALING

The process of transforming data into a standard scale

STANDARDIZATION

FEATURE SCALING



STANDARDIZATION

FEATURE SCALING

day	Exchange rate	Daily trading volume
1	1.3	110,000
2	1.34	98,700
3	1.25	135,000

mean: 1.3

std: 0.045

STANDARDIZATION

FEATURE SCALING

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

day	Exchange rate		Daily trading volume
1	0.07	(1.3)	110,000
2	0.96	(1.34)	98,700
3	-1.03	(1.25)	135,000

STANDARDIZATION

FEATURE SCALING

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

day	Exchange rate		Daily trading volume	
1	0.07	(1.3)	-0.25	(110,000)
2	0.96	(1.34)	-0.85	(98,700)
3	-1.03	(1.25)	1.1	(135,000)

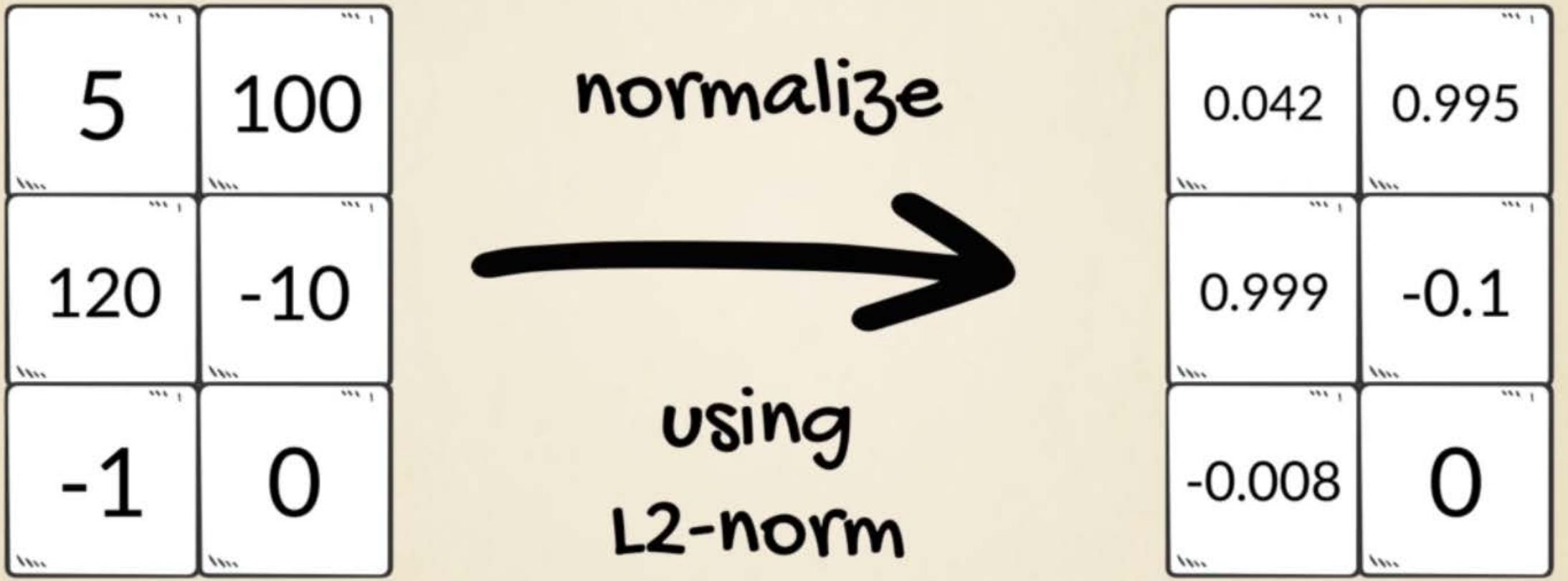
STANDARDIZATION

FEATURE SCALING

day	Exchange rate		Daily trading volume	
1	0.07	(1.3)	-0.25	(110,000)
2	0.96	(1.34)	-0.85	(98,700)
3	-1.03	(1.25)	1.1	(135,000)

We have forced the features to appear similar

NORMALIZATION



PCA

/principal components analysis/

Dimension reduction technique used to combine several variables into a bigger (latent) variable

PCA

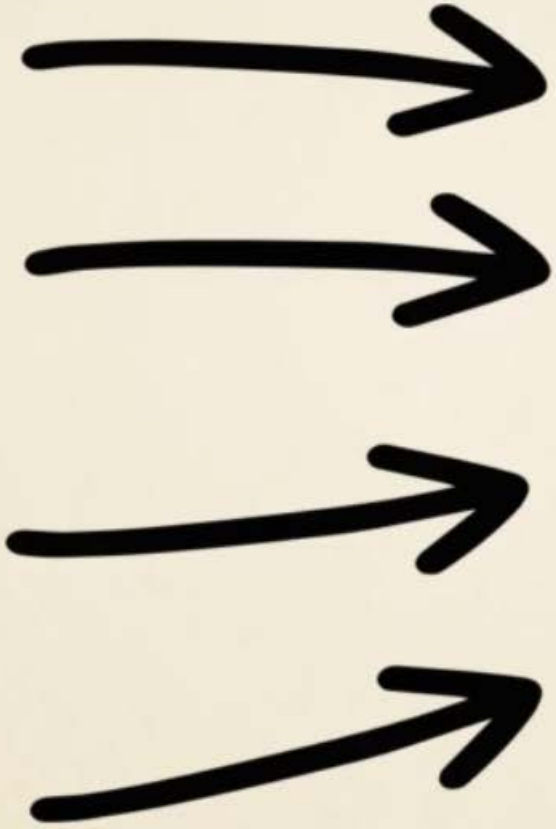
/principal components analysis/

Religion

voting history

participation in associations

upbringing



attitude towards immigration

mean = 0

std = 1

STANDARDIZATION

NORMALIZATION

PCA

WHITENING

...

CATEGORICAL DATA

Categories or groups (non-numerical data)

numbers →  → numbers

CATEGORICAL DATA

How to encode categories in a way useful for ML

One-hot encoding

Binary encoding

ONE-HOT ENCODING

	bread	yogurt	muffins
	1	0	0
	0	1	0
	0	0	1

BINARY ENCODING

	ordinal	binary
	1	01
	2	10
	3	11

BINARY ENCODING

	ordinal	var 1	var 2
	1	0	1
	2	1	0
	3	1	1

BINARY ENCODING

	var 1	var 2
	0	1
	1	0
	1	1

By splitting the numbers in two variables we have removed the order

...but...

there are some implied correlation between them

BINARY ENCODING

	var 1	var 2
	0	1
	1	0
	1	1

Bread is the opposite of yogurt

Whatever is bread is not yogurt and vice versa

12,000
products

	ONE-HOT	BINARY
NEW COLUMNS	12,000	16

12,000 IN BINARY: 10111011100000

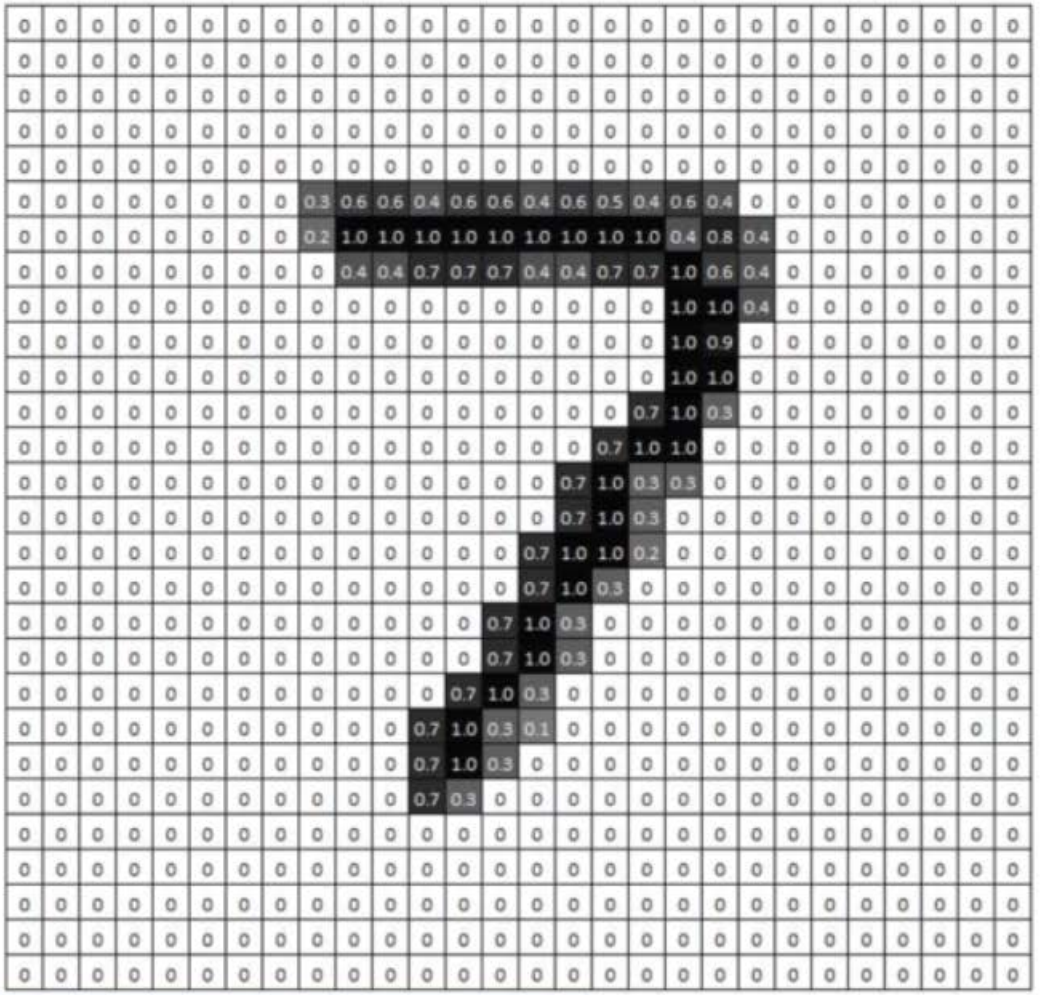
ONE-HOT

FEW
CATEGORIES

BINARY

MANY
CATEGORIES

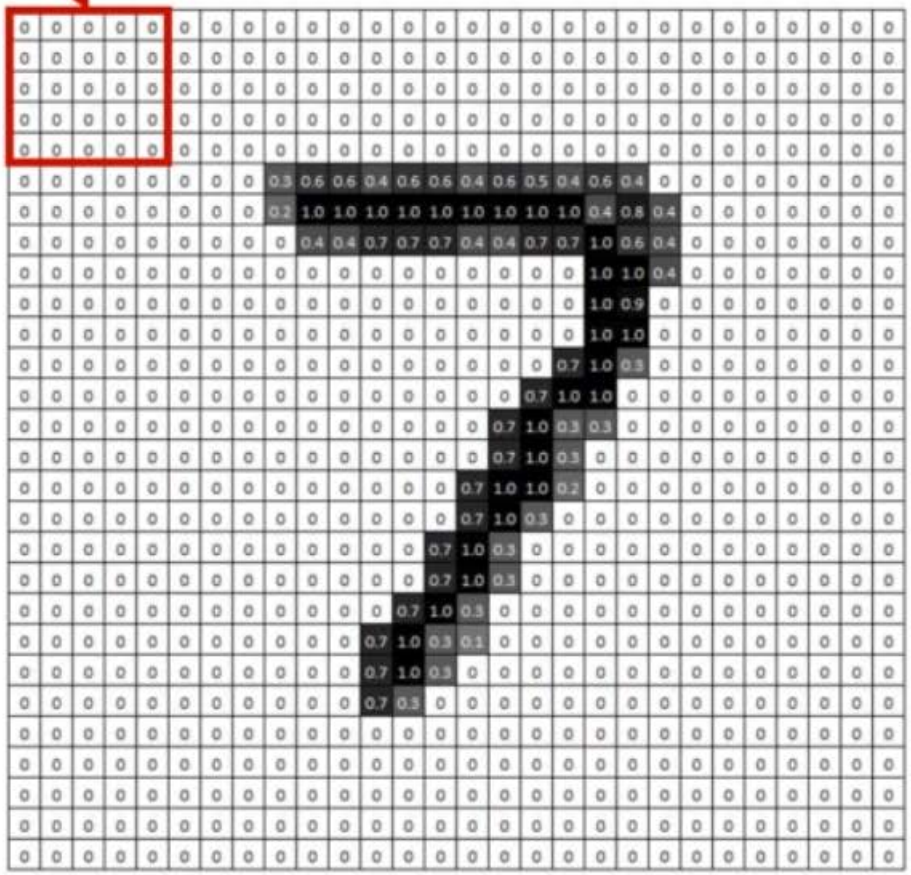
Convolutional Neural Networks (CNNs)



A CNN deals with the problem without **flattening** the images

Convolutional Neural Networks (CNNs)

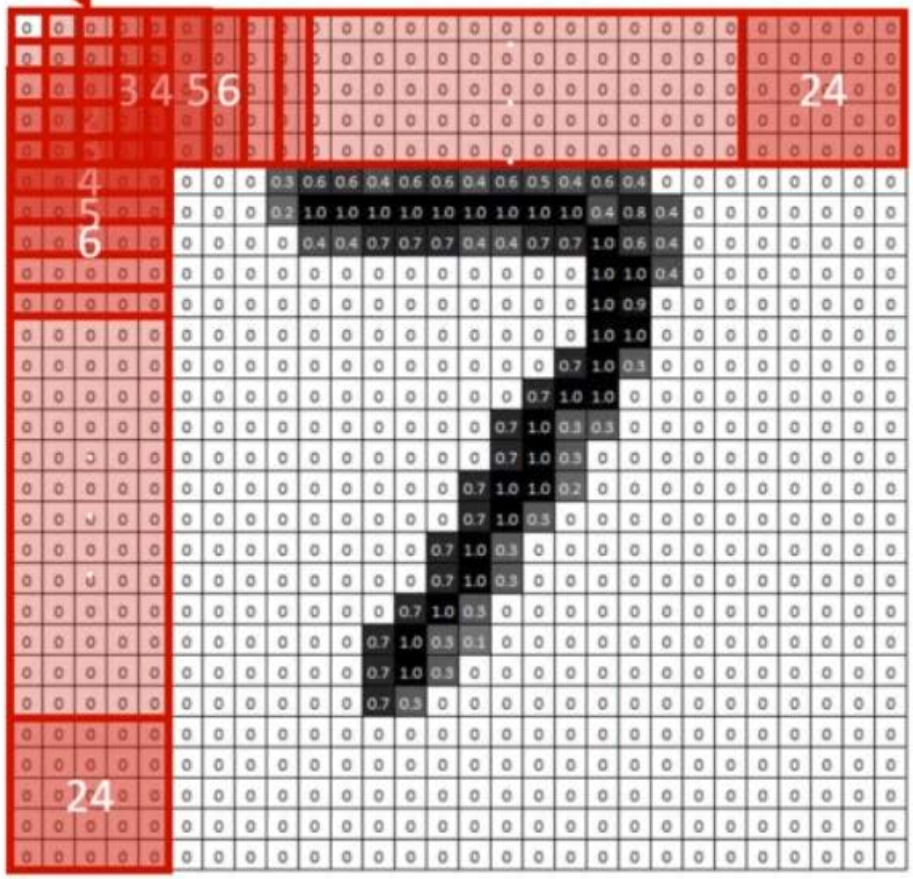
A 5x5 *kernel*



Kernels are like weights

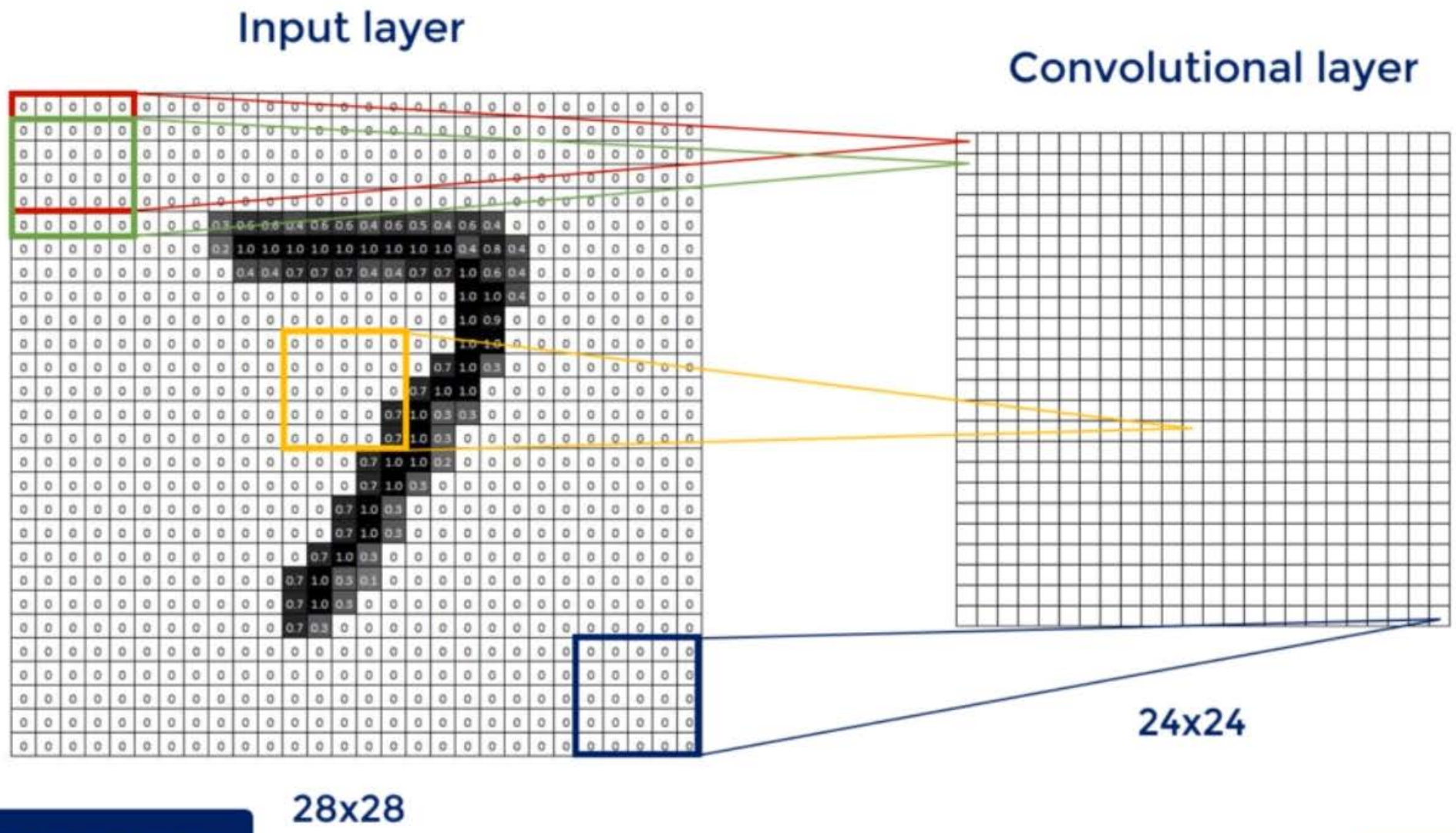
Convolutional Neural Networks (CNNs)

A 5x5 *kernel*



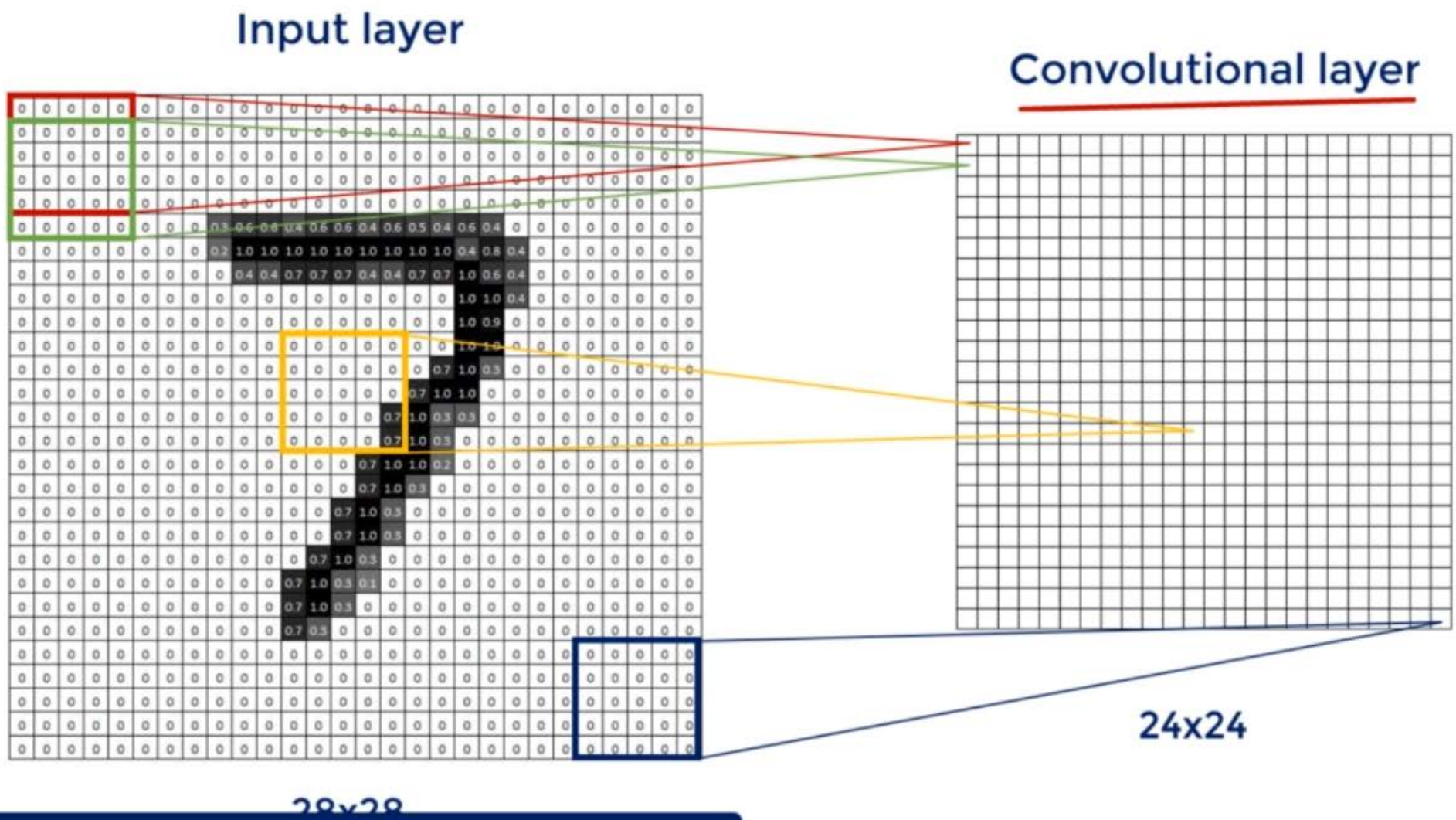
28x28

Convolutional Neural Networks (CNNs)



This is called **convolution**

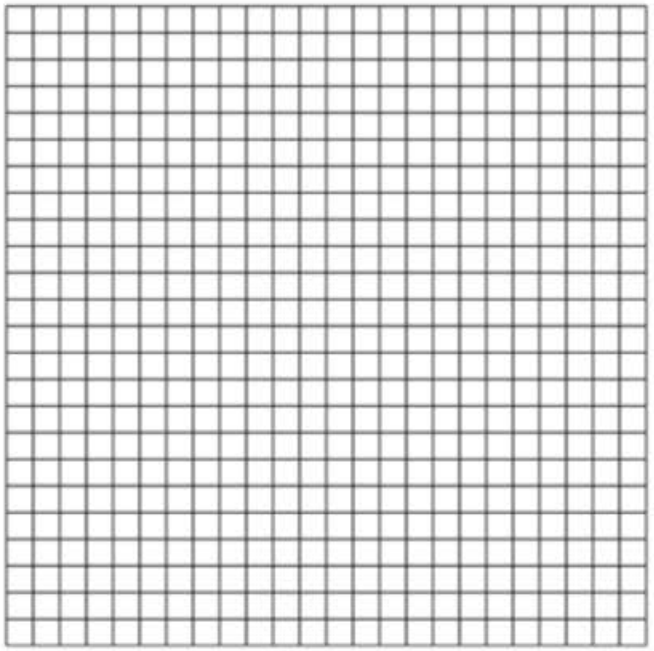
Convolutional Neural Networks (CNNs)



The number of kernels you choose is a hyperparameter

Convolutional Neural Networks (CNNs)

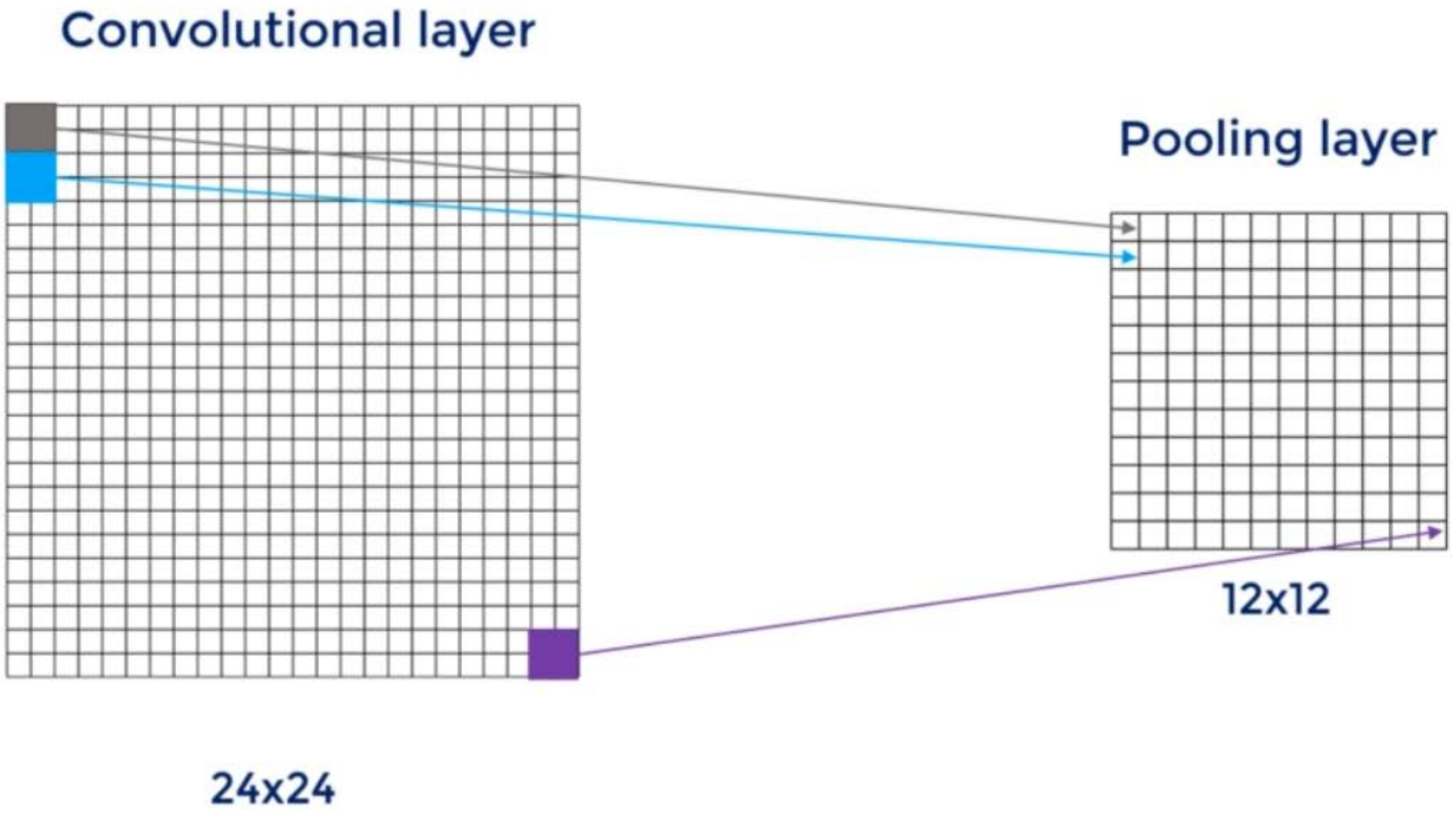
Convolutional layer



24x24

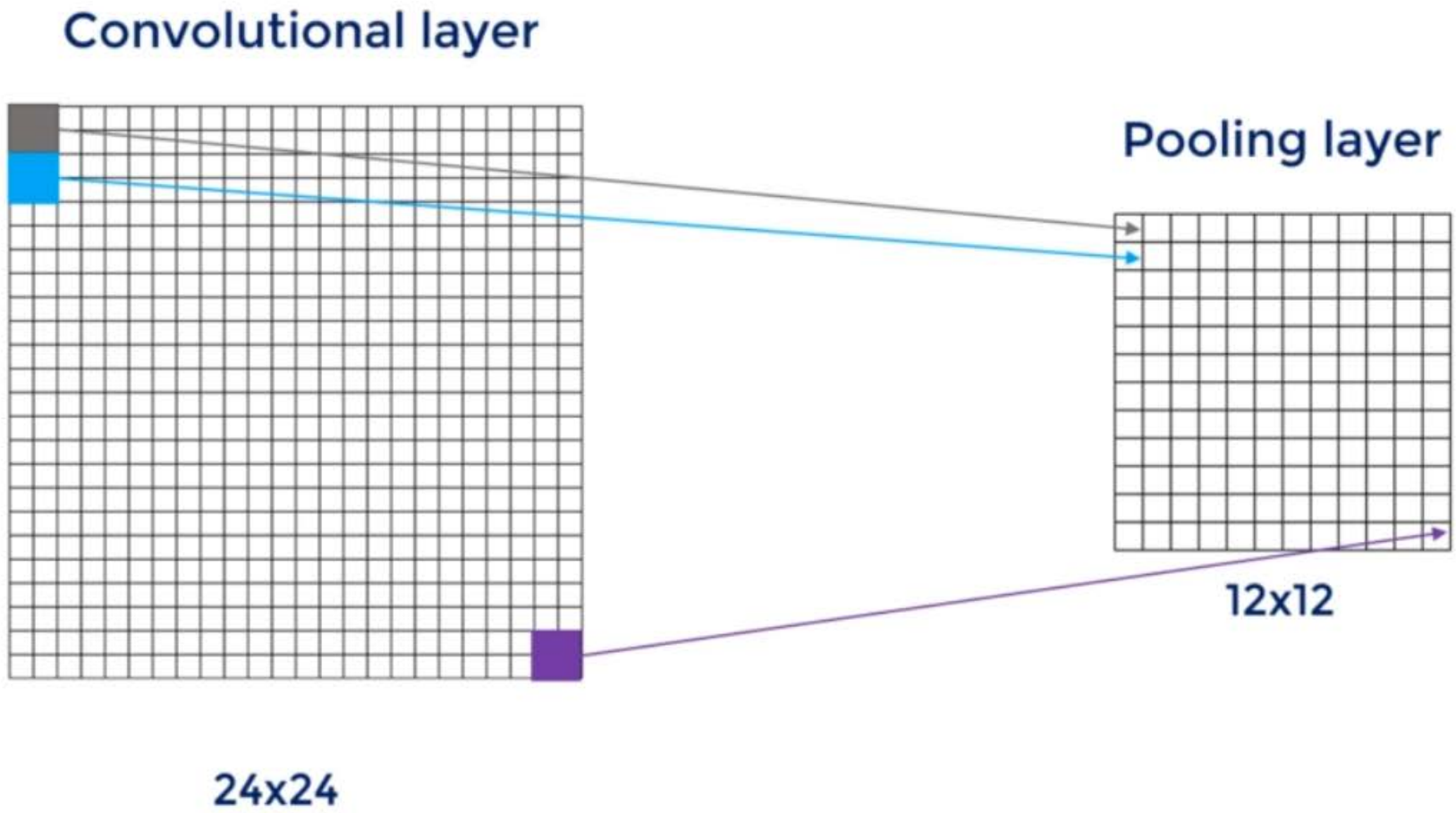
Apart from convolution there is another main step - **pooling**

Convolutional Neural Networks (CNNs)



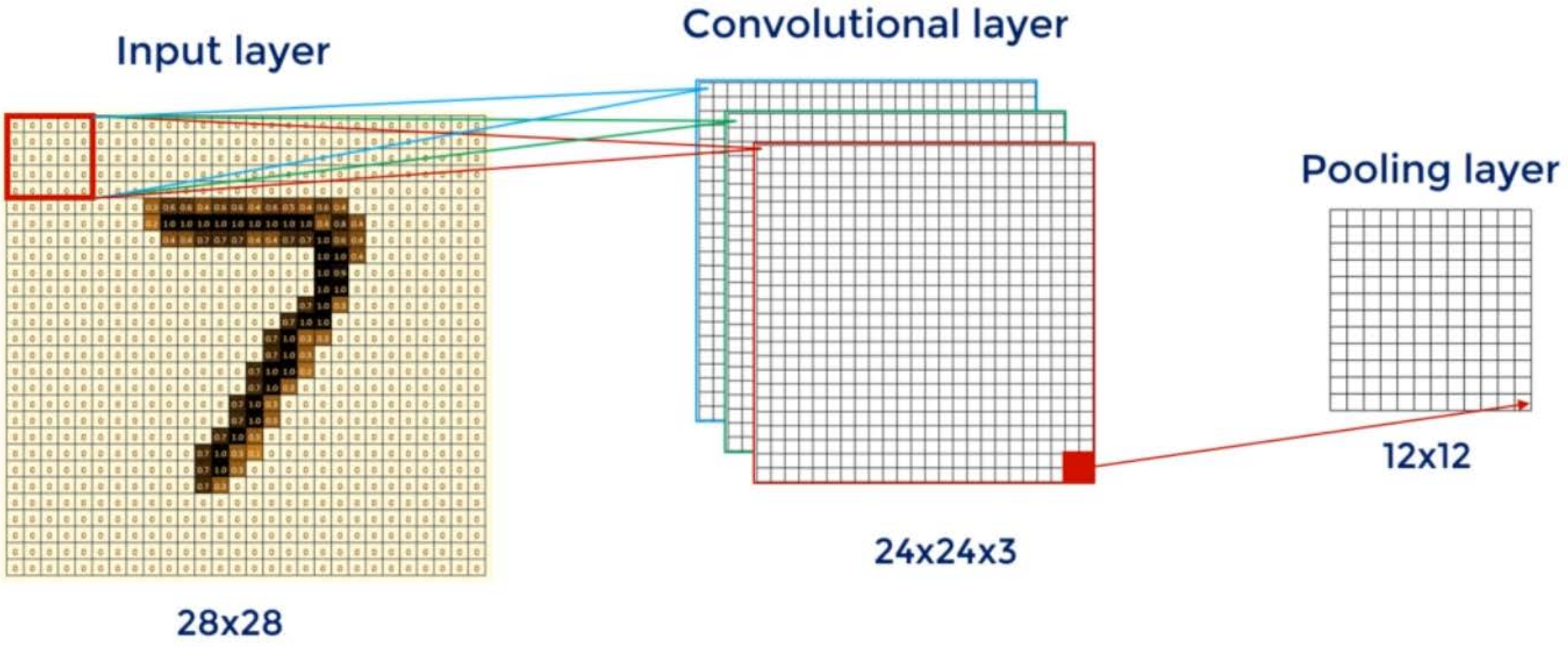
We would divide the convolutional layer into small squares **without overlapping**

Convolutional Neural Networks (CNNs)



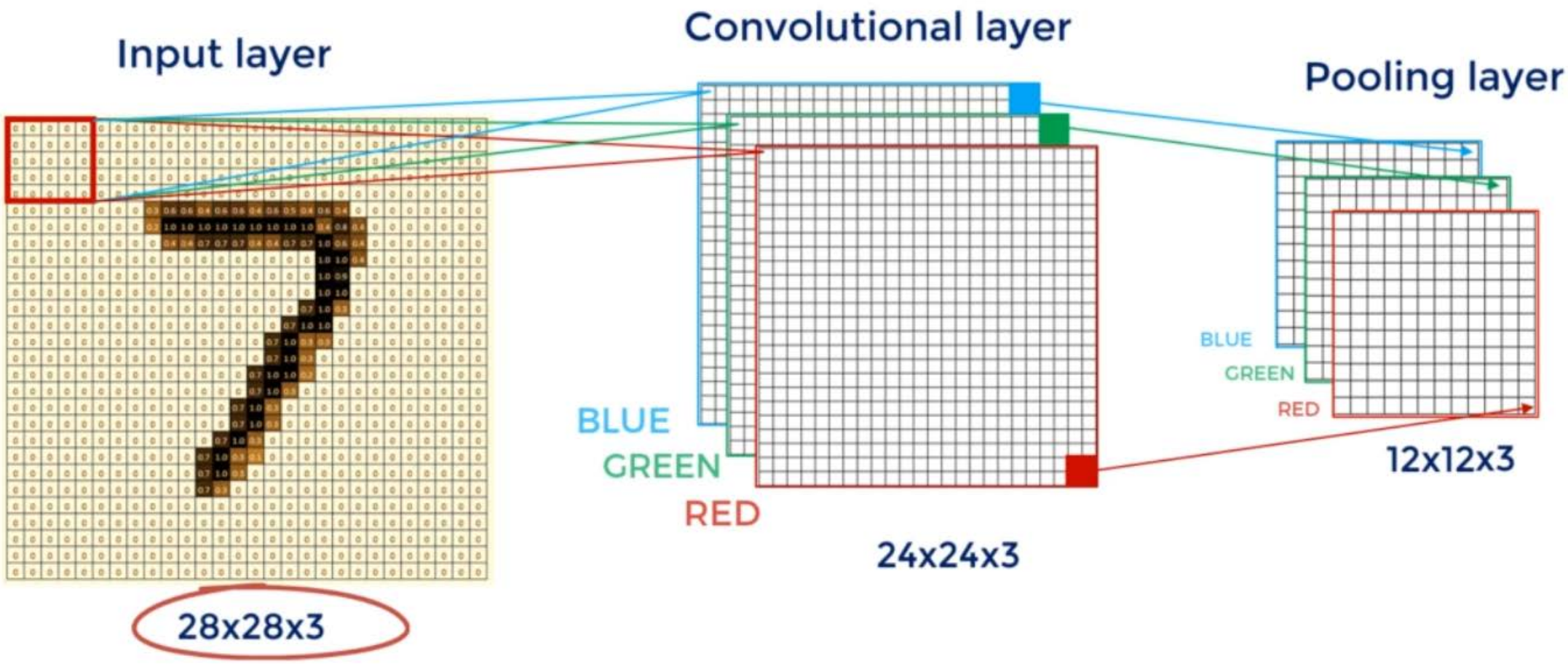
Normally we will take (and keep) the largest number from the square as it is the **strongest detail**

Convolutional Neural Networks (CNNs)



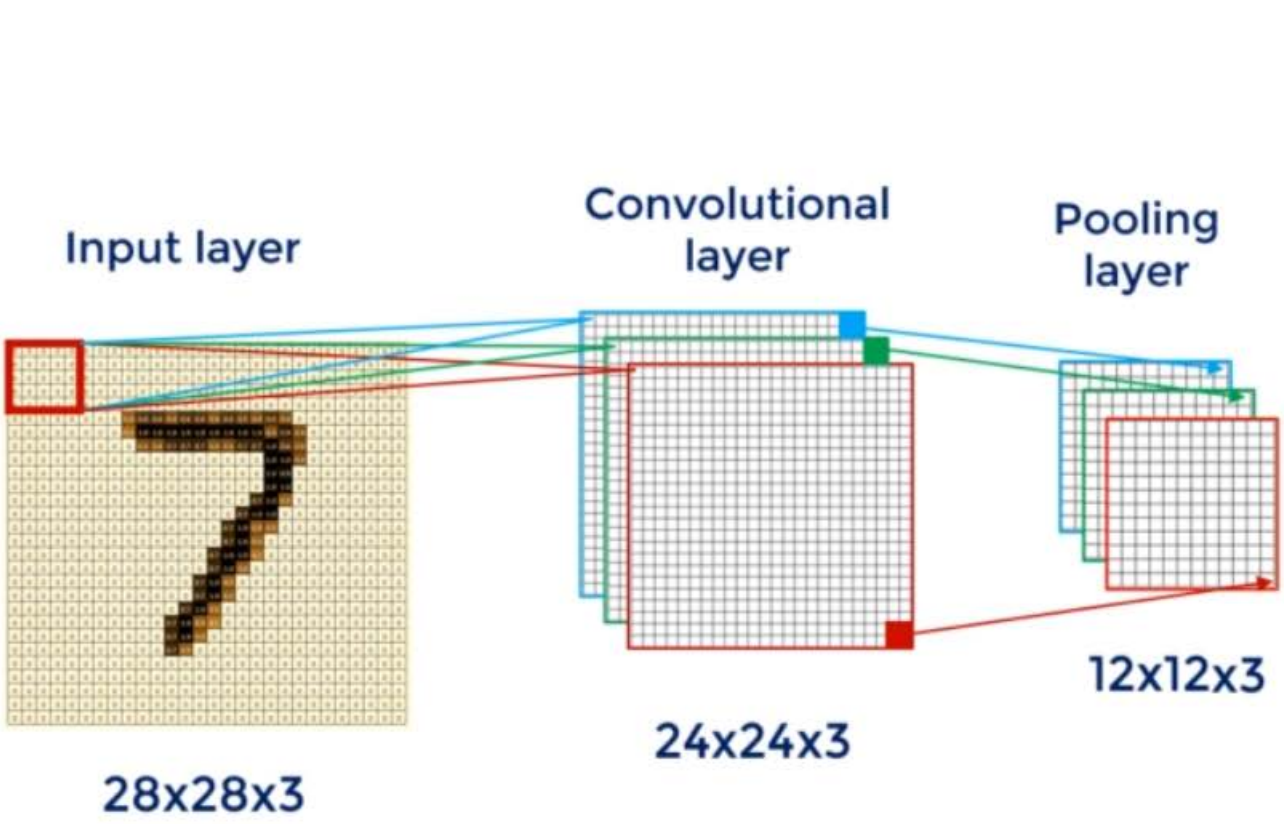
Images have height and width, but also... color **(depth)**

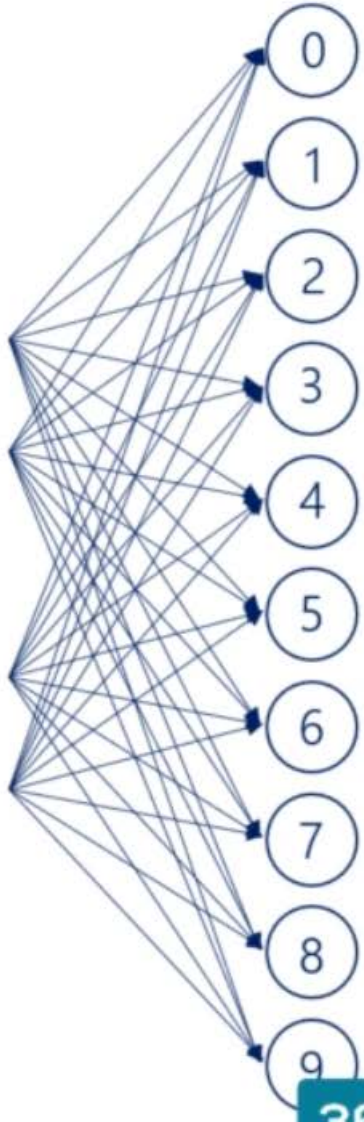
Convolutional Neural Networks (CNNs)



That's where the whole **tensor approach** fits perfectly

Convolutional Neural Networks (CNNs)





In order to truly get CNNs we need more time than 5 minutes :(

CNNs

- spatial proximities are preserved

- a detail is looked for everywhere in a photo 👁️

CNNs

Image recognition

RNNs

sequential data

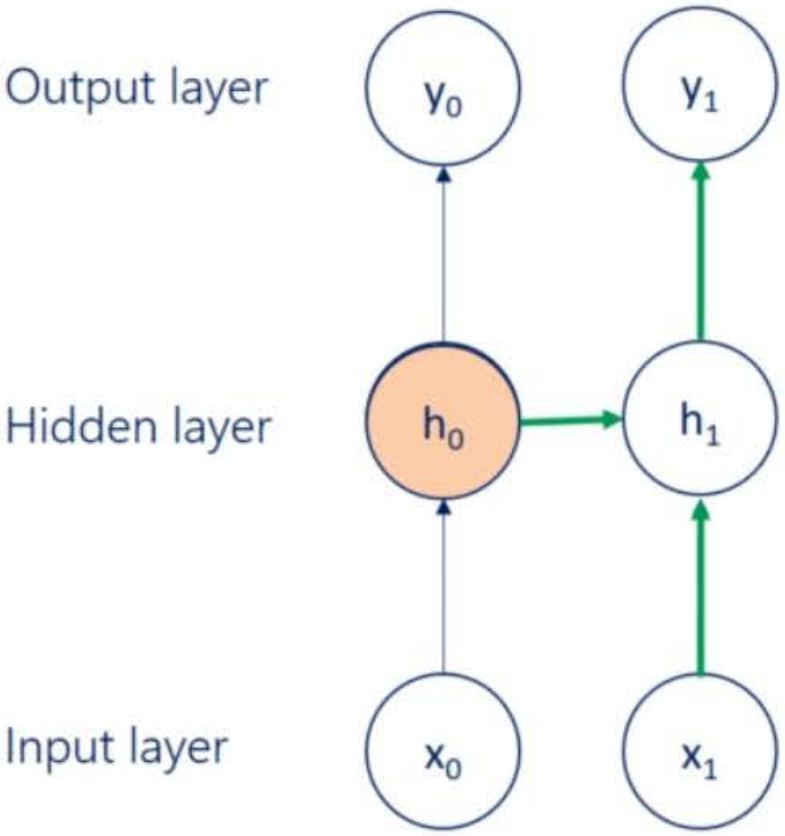
trading

music

NLP

depends on what you've said before

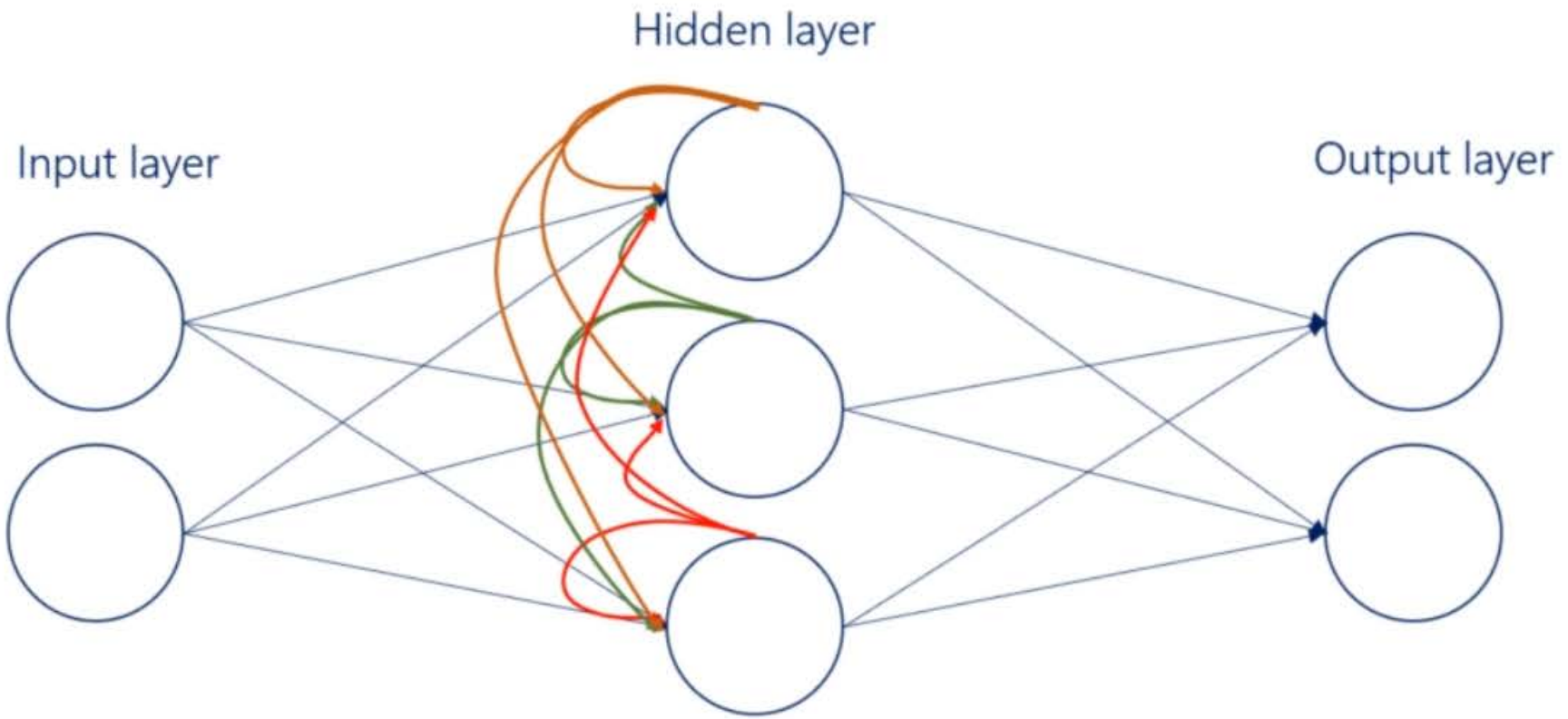
Recurrent neural networks (RNNs)



The hidden layer for the second sample is a function of the new inputs, the updated weights, **and the hidden layer** from the previous iteration.

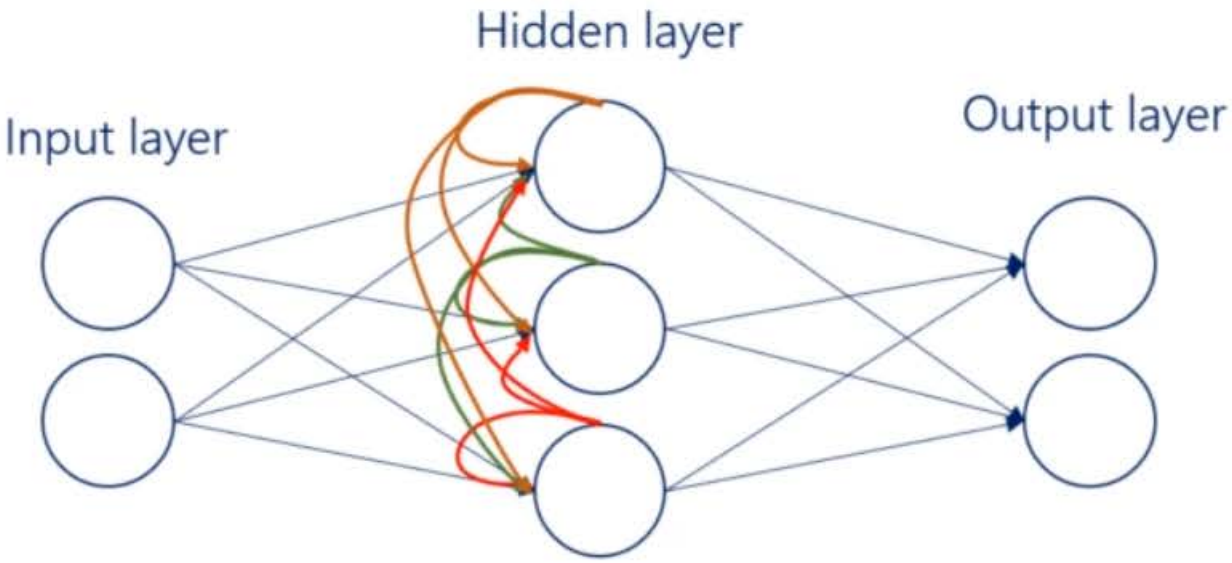
*Indices here indicate time, rather than a specific unit. An index of 0, implies that layer at time 0.

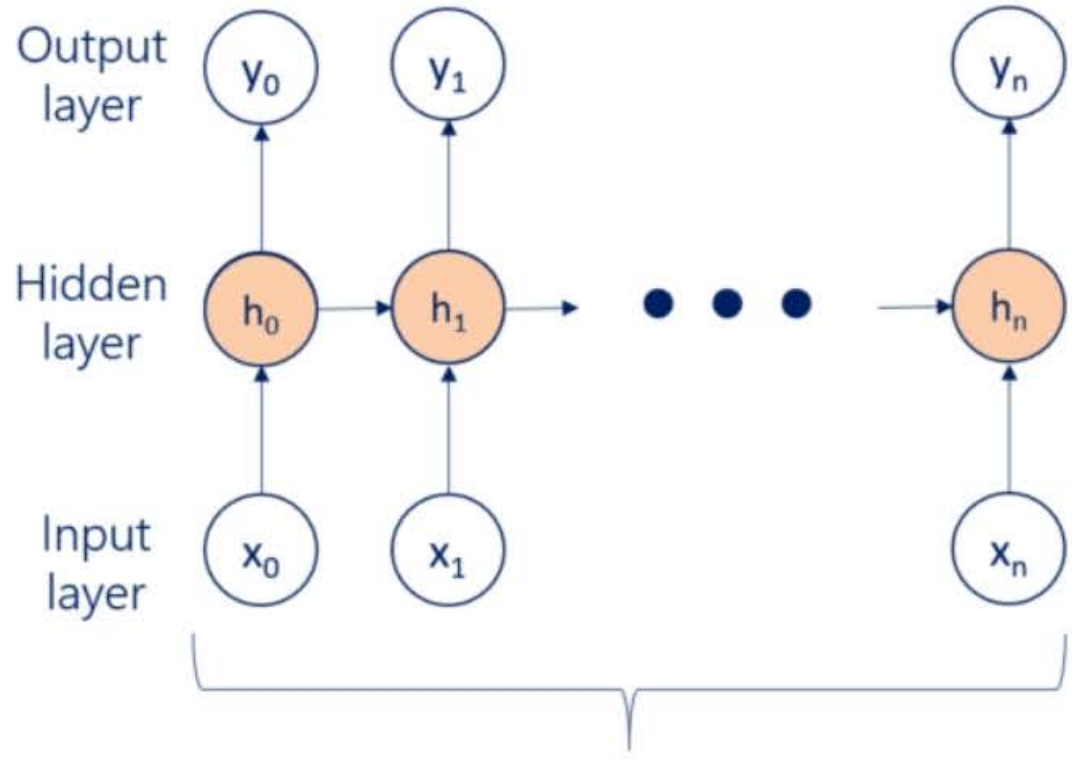
Recurrent neural networks (RNNs)



There are additional weights that lead from the hidden units to **themselves**

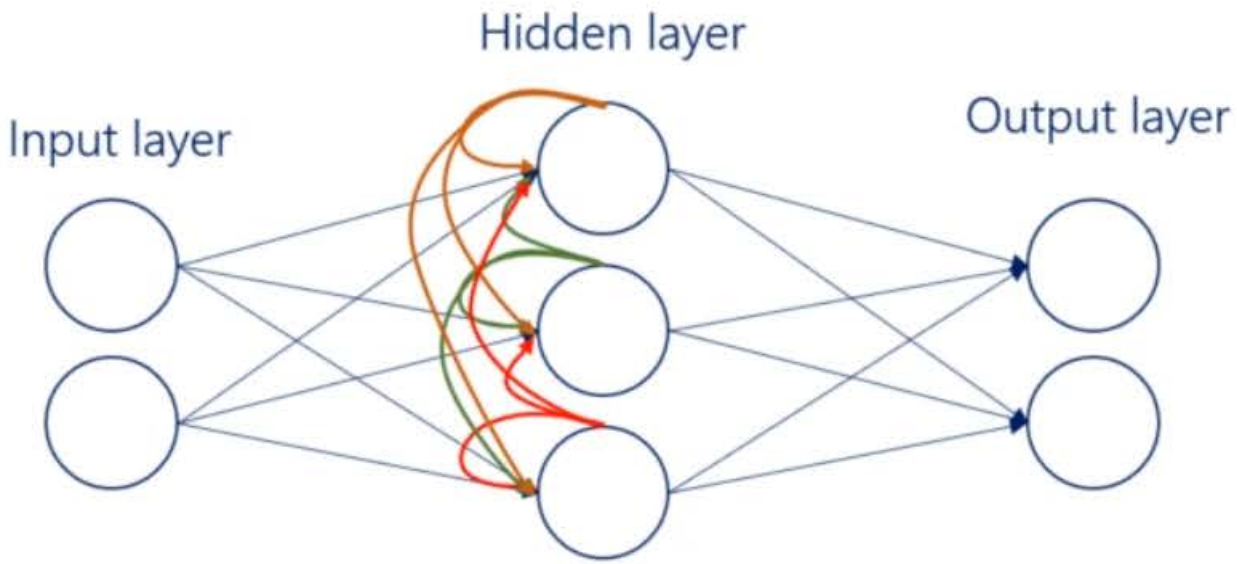
Recurrent neural networks (RNNs)



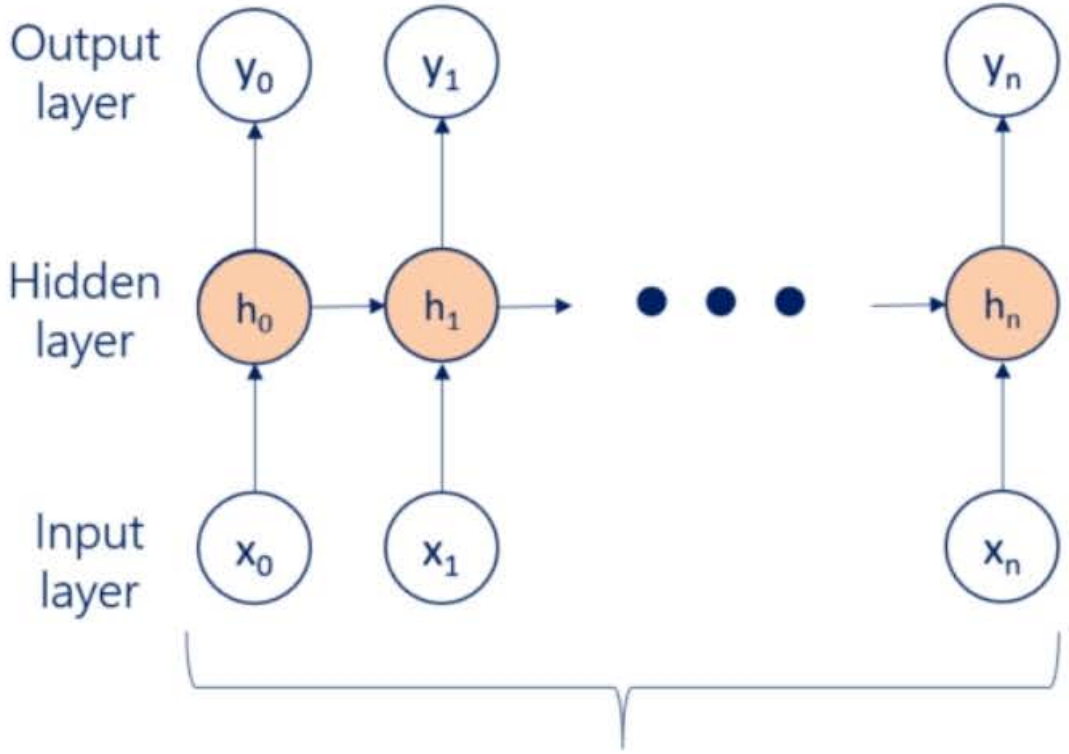


An extremely deep feedforward NN

Recurrent neural networks (RNNs)



"unrolling" the RNN



An extremely deep feedforward NN

RNNs are very computationally expensive

365 DataScience

MACHINE LEARNING

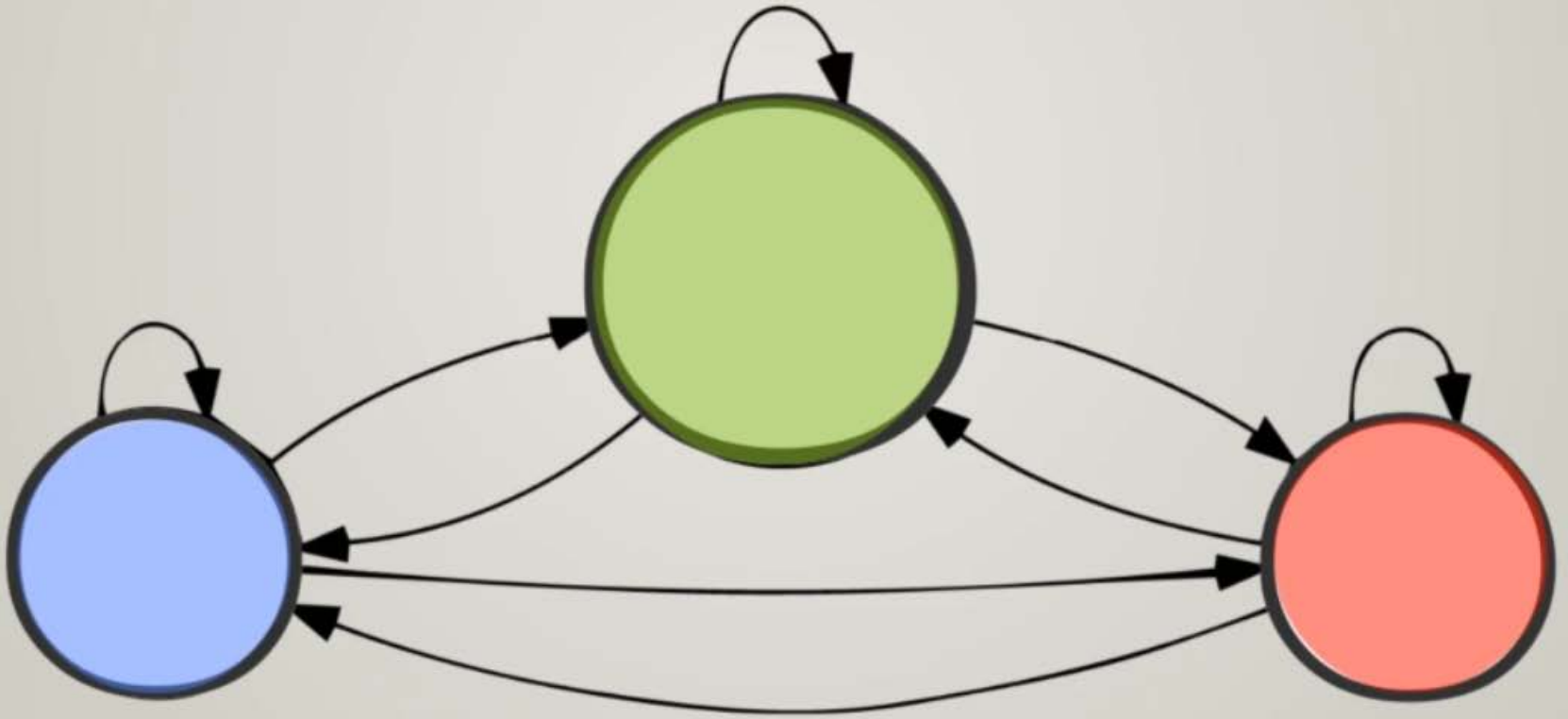
NON-NEURAL NETWORK APPROACHES

Generative models

ENGLISH  MANDARIN → TARGET JOINT PROBABILITY

Generative models

Hidden Markov models



Generative models

Bayesian networks

Likelihood of the evidence 'E' if the Hypothesis 'H' is true

Prior Probability

$$P(H|E) = \frac{P(E|H) * P(H)}{P(E)}$$

Posterior Probability of 'H' given the evidence

Prior probability that the evidence itself is true

Generative models

Bayesian networks

inputs are..

PROBABILITY ⇄  ⇄ OUTPUT

More information than in NN

Each problem, no matter if an image, a business problem, or a Shakespeare sonnet, tells a story. With the right tools you can reveal it and take advantage of it.

A matrix is a collection of numbers ordered in rows and columns. Here is one.

A is a 2-by-3 matrix

$$\underset{2\times 3}{A} = \begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix}$$

2 rows

3 columns

Matrices are main characters in mathematical operations

- addition
- subtraction
- multiplication

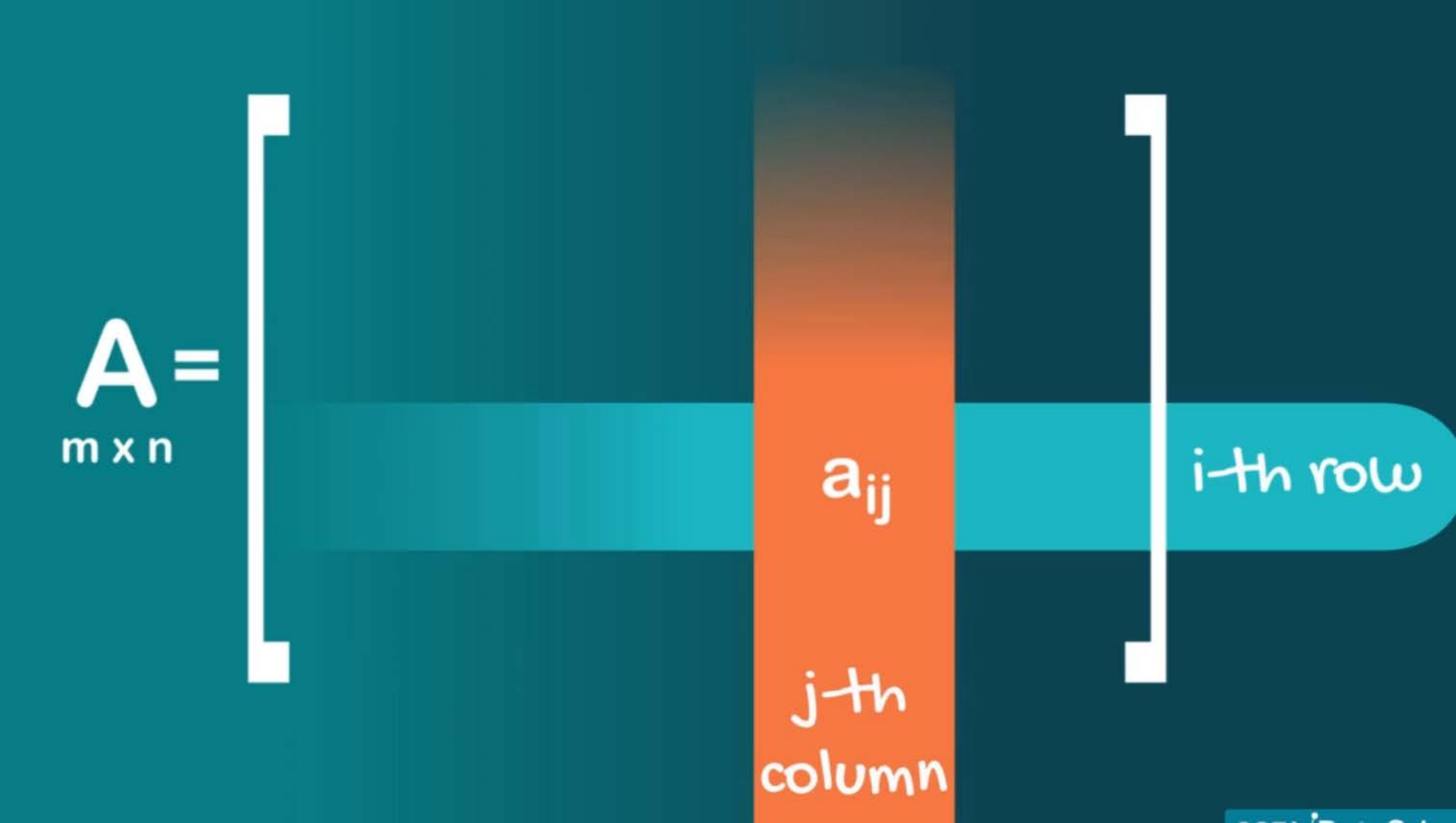
$$\begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix}$$

A matrix can only contain numbers, symbols, or expressions

$$A_{2x3} = \begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix}$$

$$B_{2x3} = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

$$C_{2x2} = \begin{bmatrix} a-x & b+5 \\ d+e & e \end{bmatrix}$$


A =
m x n
a_{ij}
i-th row
j-th
column

MATHEMATICS

$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & a_{ij} & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

PROGRAMING

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1(n-1)} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & a_{(m-1)2} & \cdots & a_{(m-1)(n-1)} \end{bmatrix}$$

MATRICES

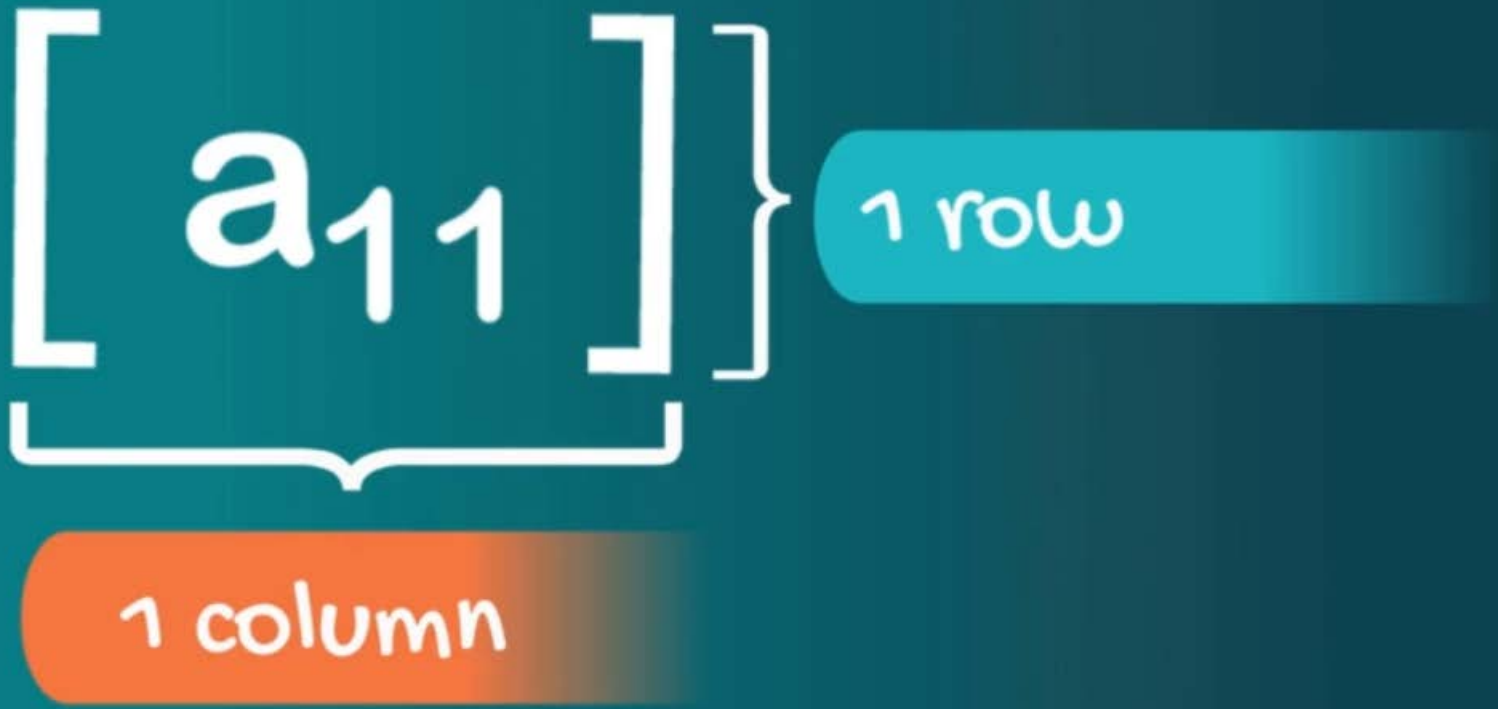
$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & a_{ij} & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

rows

columns

2D

SCALARS



SCALARS

All numbers we know from algebra are referred to as scalars in linear algebra

[15]; [1]; [2]; [-5]; []

SCALARS HAVE 0 DIMENSIONS

VECTORS

$$U_{3 \times 1} = \begin{bmatrix} 5 \\ -2 \\ 4 \end{bmatrix}$$

3 rows

1 column

VECTORS

A vector is practically the simplest linear algebraic object

$$\begin{bmatrix} 5 \\ -2 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix}$$

vect. 1 | vect. 2 | vect. 3

TYPES OF VECTORS

column

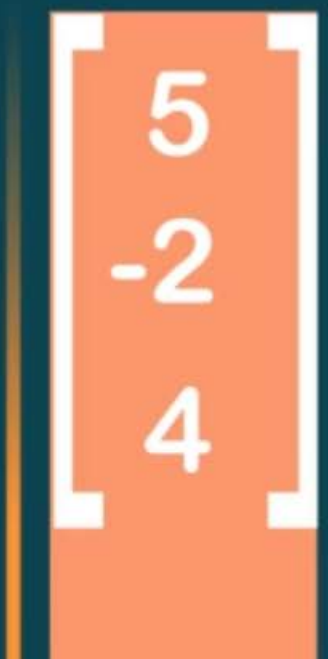
$$\begin{bmatrix} 5 \\ -2 \\ 4 \end{bmatrix}$$

row

$$\begin{bmatrix} 3 & 4 & 5 & 8 \end{bmatrix}$$

LENGTH

Df: The number of elements in a vector



Length = 3

Length = 4

[3 4 5 8]

SUMMARY

$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & a_{ij} & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

$$\begin{bmatrix} x \end{bmatrix}$$

2D Matrix
m x n

1D Vector
m x 1

0D Scalar
1 x 1

LINEAR ALGEBRA AND GEOMETRY

0D Scalar
1 x 1

0D Point

LINEAR ALGEBRA AND GEOMETRY

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

1D Vector $m \times 1$

1D Line

LINEAR ALGEBRA AND GEOMETRY

$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_m & y_m \end{bmatrix}$$

2D

Matrix
m x 2

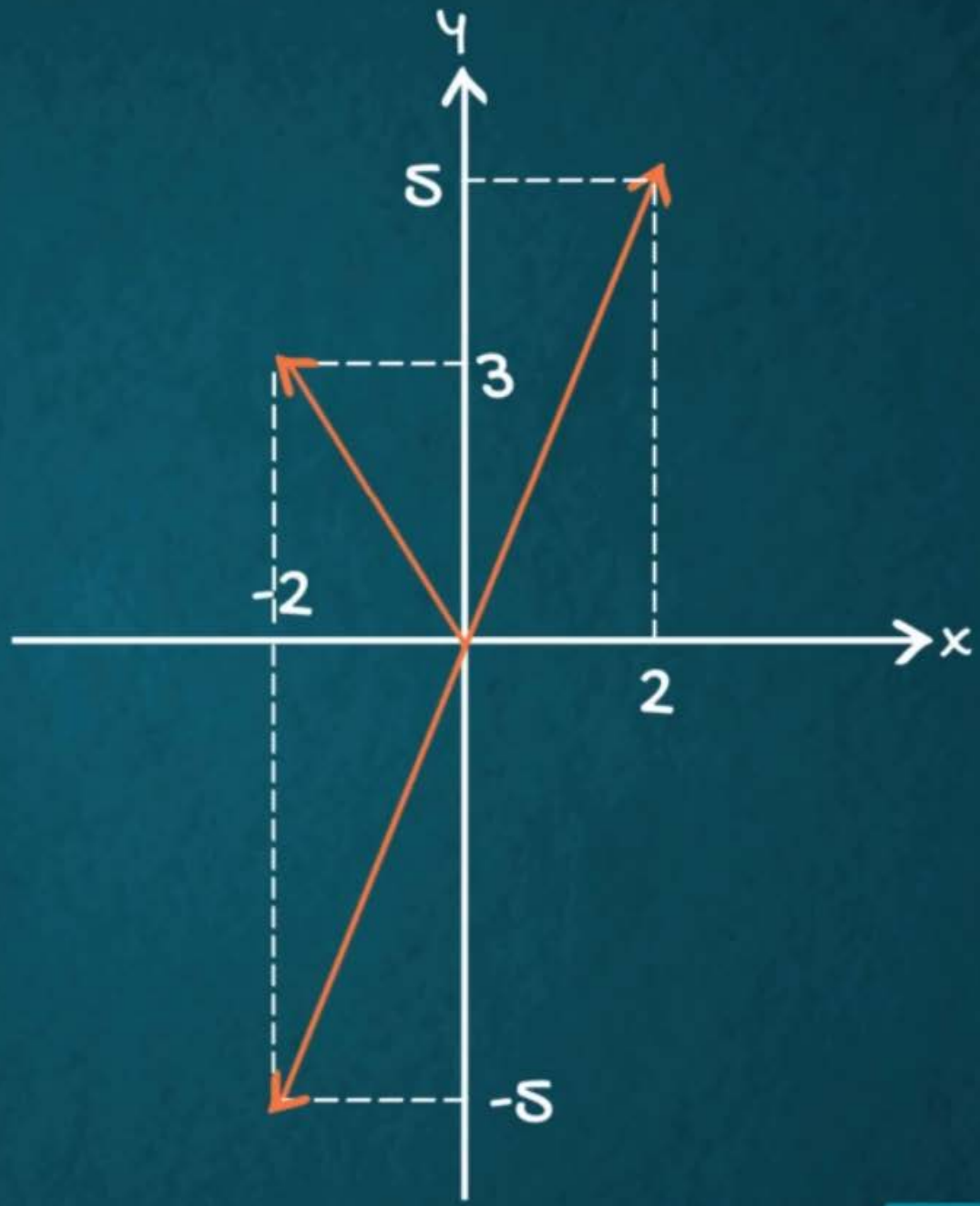
2D

Line

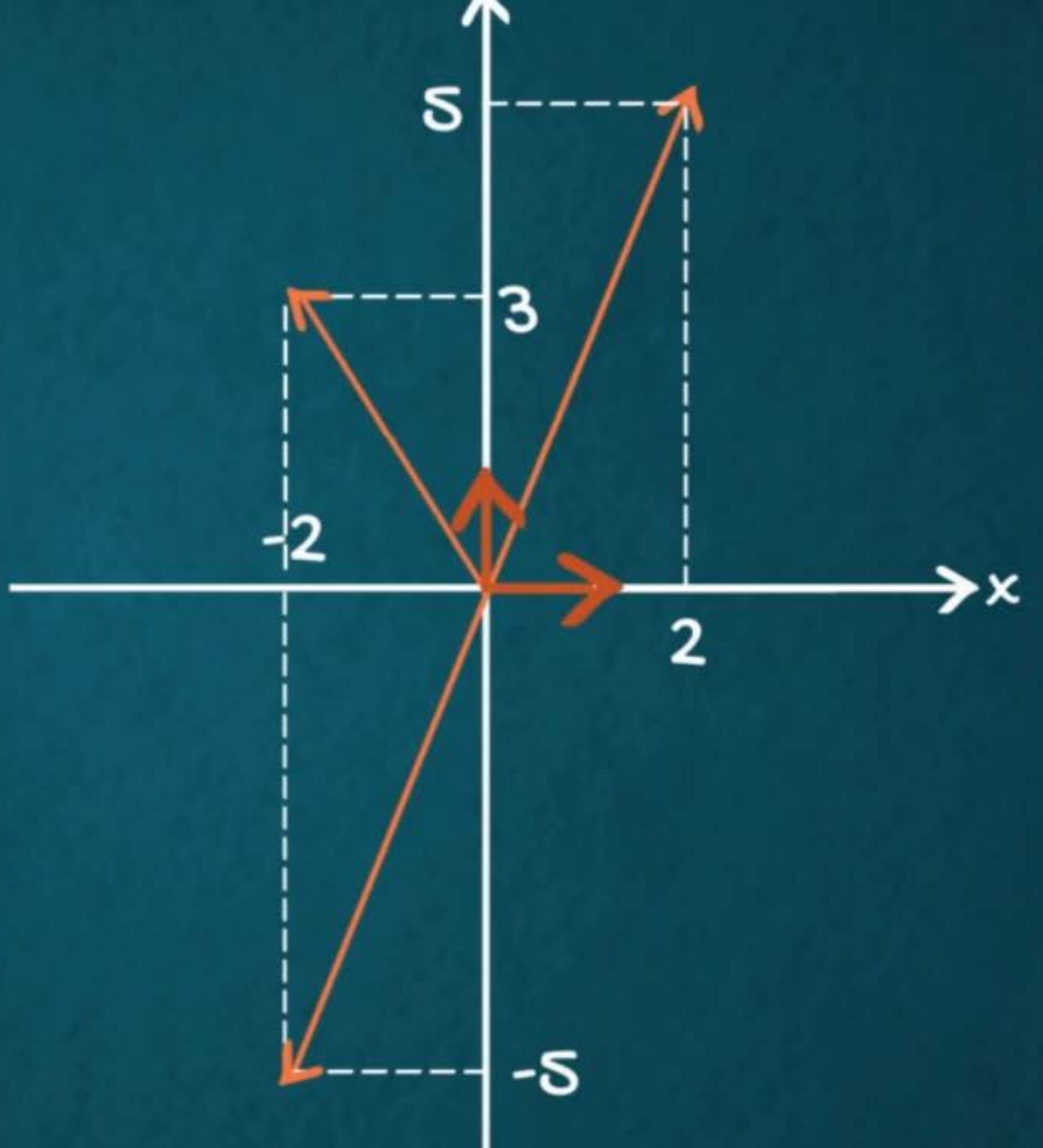
$$\begin{bmatrix} 2 \\ 5 \end{bmatrix}$$

$$\begin{bmatrix} -2 \\ -5 \end{bmatrix}$$

$$\begin{bmatrix} -2 \\ 3 \end{bmatrix}$$



$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$



TRANSPOSING VECTORS

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}^T$$

x

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

x^T

TRANSPOSING VECTORS

- The values are not changing or transforming; only their position is
- Transposing the same vector (object) twice yields the initial vector (object)
- A 3x1 matrix transposed is a 1x3 matrix

TRANSPOSING MATRICES

$$A_{2\times 3} = \begin{bmatrix} 5 & 12 & 6 \\ -3 & 0 & 14 \end{bmatrix}$$

$$A^{T}_{3\times 2} = \begin{bmatrix} 5 & -3 \\ 12 & 0 \\ 6 & 14 \end{bmatrix}$$

SCALAR MULTIPLICATION

$$[6] . [5] = [30]$$

$$[10] . [-2] = [-20]$$

DOT PRODUCT

$$\begin{bmatrix} 2 \\ 8 \\ -4 \end{bmatrix} \times \begin{bmatrix} 1 \\ -7 \\ 3 \end{bmatrix}$$

(inner product)

DOT PRODUCT

$$\begin{bmatrix} 2 \\ 8 \\ -4 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -7 \\ 3 \end{bmatrix} = \begin{bmatrix} 2 \times 1 + 8 \times (-7) + (-4) \times 3 \end{bmatrix}$$

DOT PRODUCT

$$\begin{bmatrix} 2 \\ 8 \\ -4 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -7 \\ 3 \end{bmatrix} = \begin{bmatrix} -66 \end{bmatrix}$$

Vector Vector Scalar

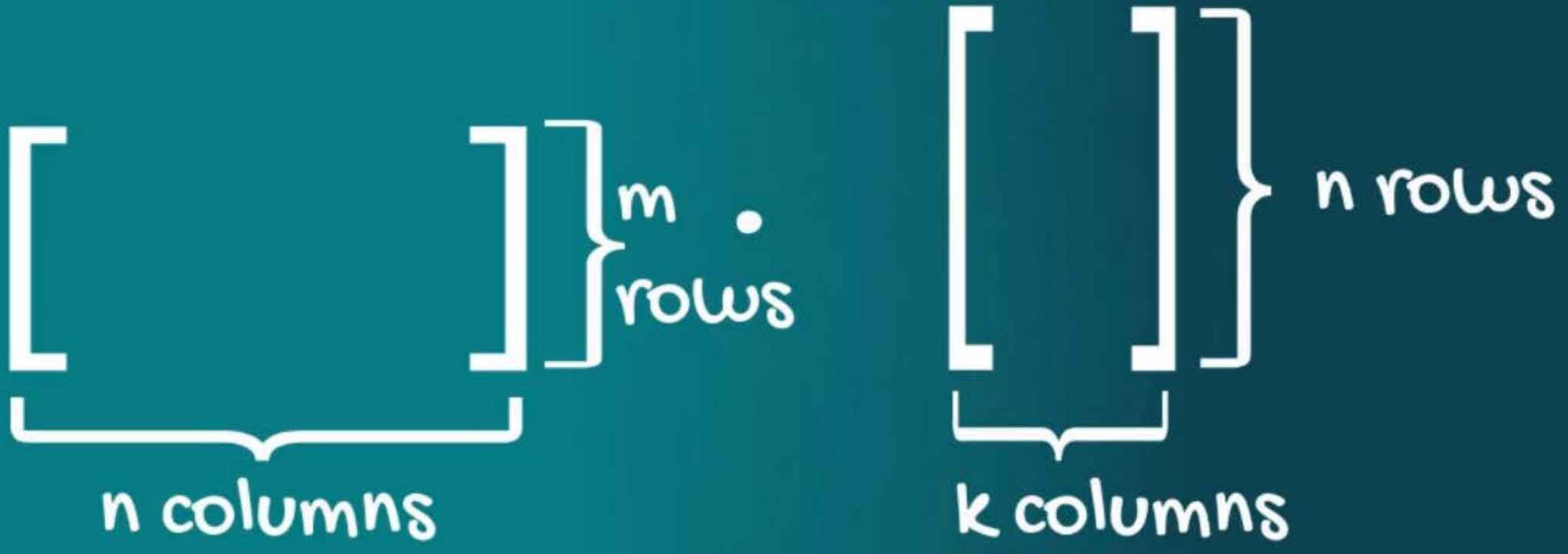
DOT PRODUCT

$$\begin{bmatrix} 2 \\ 8 \\ -4 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -7 \\ 3 \end{bmatrix} = \begin{bmatrix} -66 \end{bmatrix}$$

the 'sum of the products of the corresponding elements'

Vector Vector Scalar

DOT PRODUCT



Condition: We can only multiply an m x n with an n x k matrix

DOT PRODUCT

2 X 3

3 X k

Condition: We can only multiply an m x n with an n x k matrix

DOT PRODUCT

$$A_{2\times3} \cdot B_{3\times6} = C_{2\times6}$$

WHY IS LINEAR ALGEBRA ACTUALLY USEFUL?

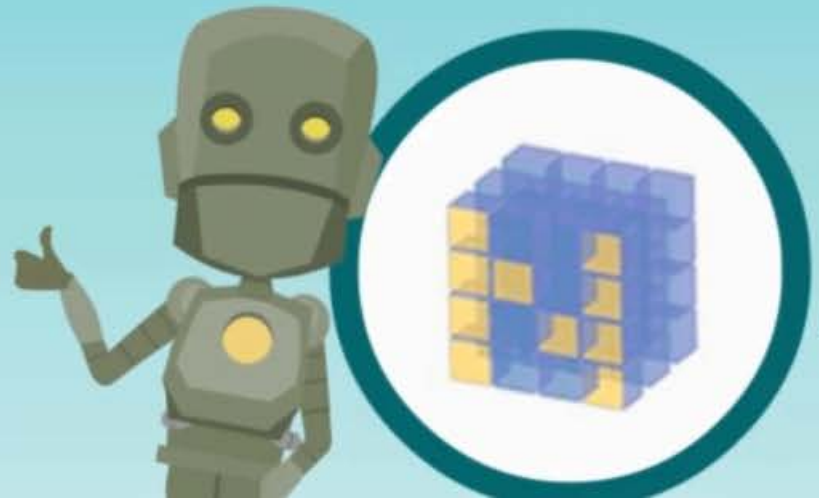
Applications in data science

- Vectorized code
- Image recognition
- Dimensionality reduction

VECTORIZED CODE

whenever we are using linear algebra
to compute many values simultaneously

much faster

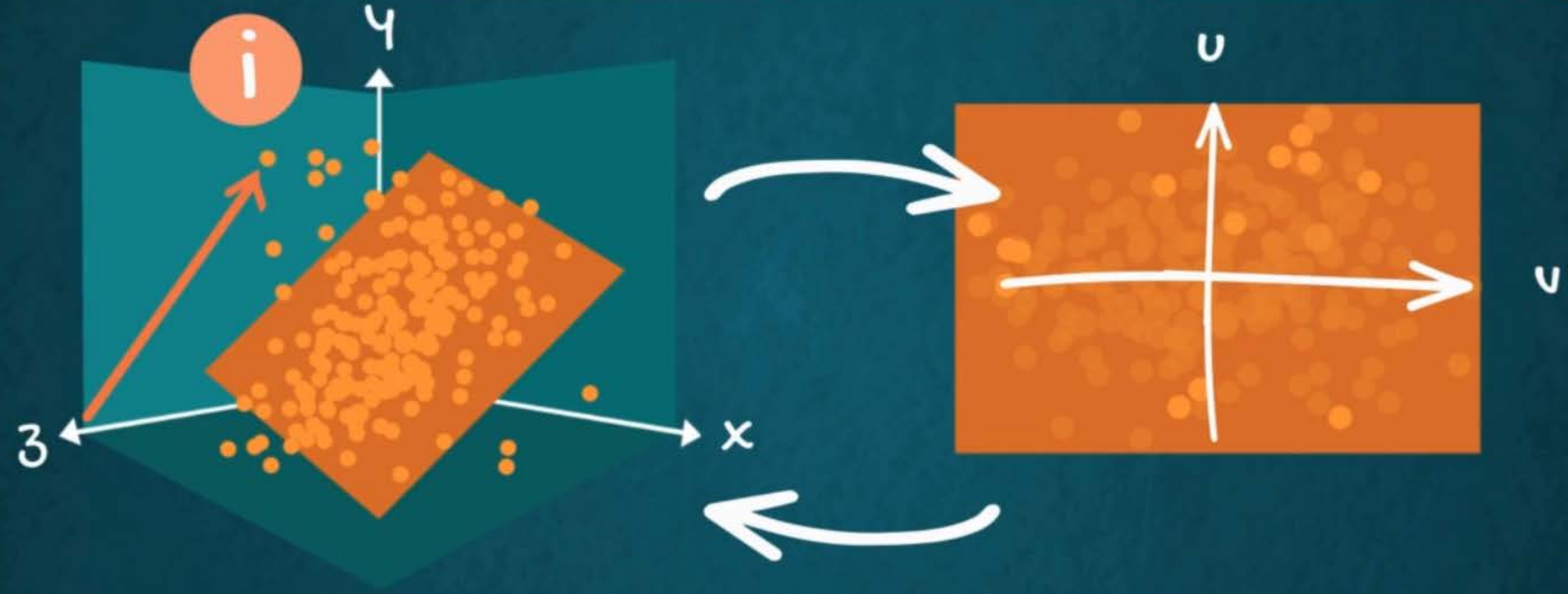


NUMPY

DIMENSIONALITY REDUCTION

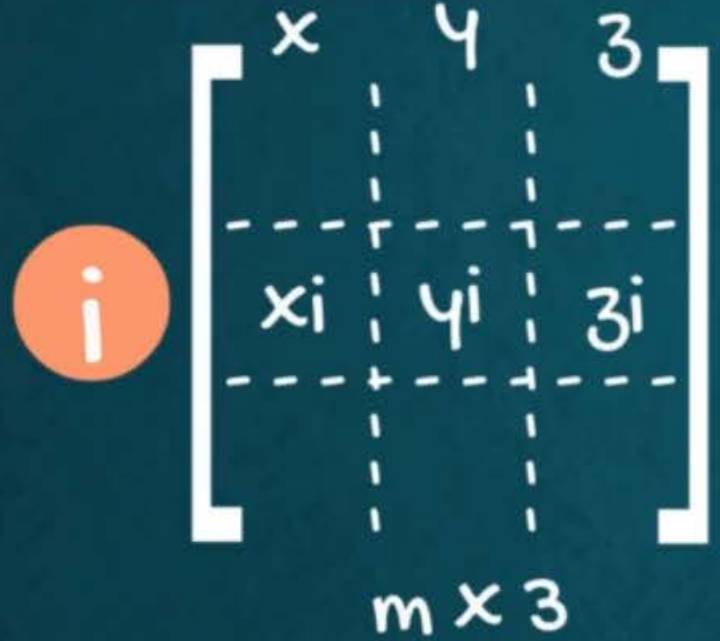
- Eigenvalues
- Eigenvector

DIMENSIONALITY REDUCTION



DIMENSIONALITY REDUCTION

INITIAL



TRANSFORMED