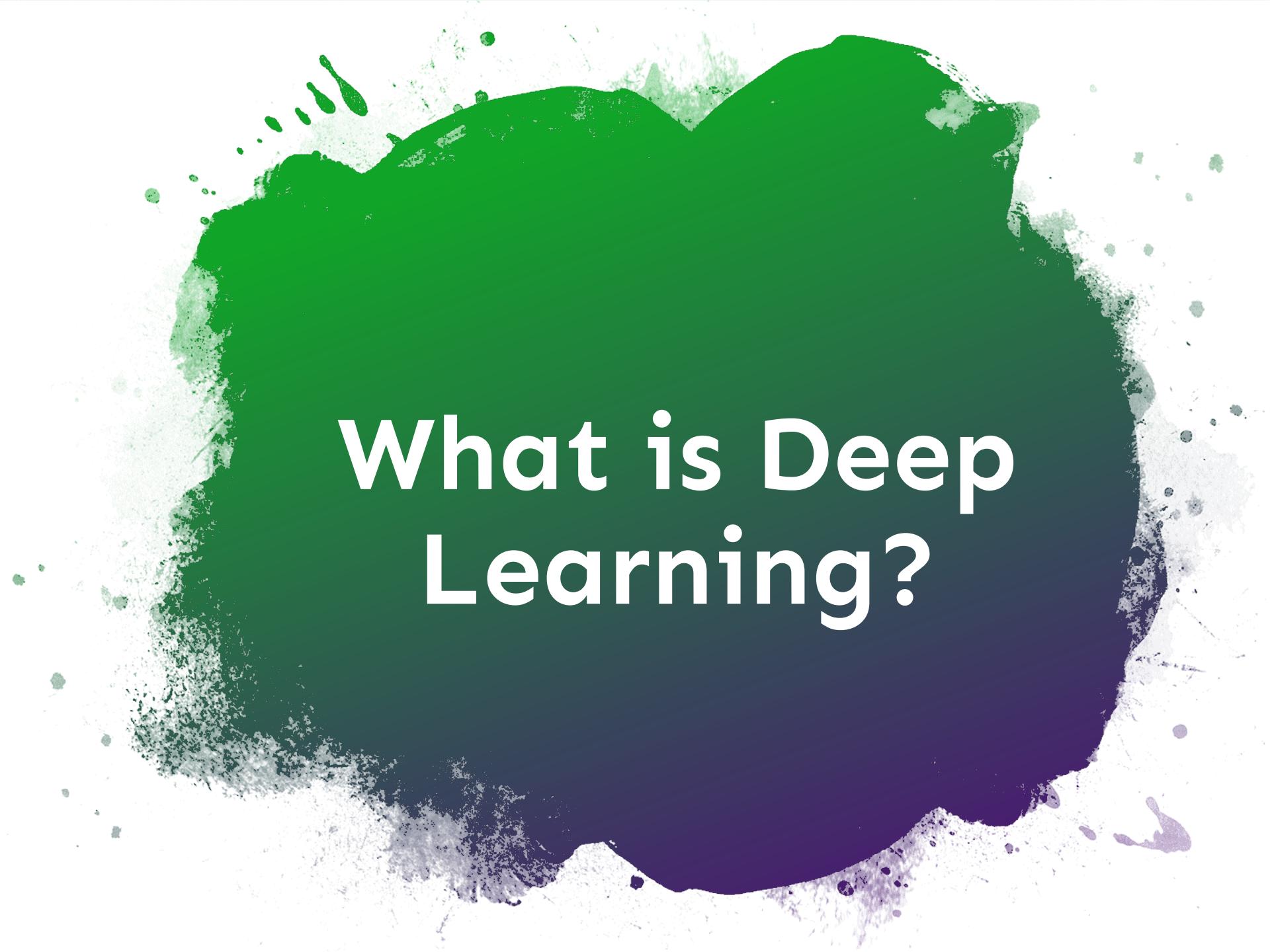




# Practical AI

## 04 – Deep Learning

Emanuele Fabbiani



# What is Deep Learning?

Article | Open Access | Published: 15 July 2021

## Highly accurate protein structure prediction by AlphaFold

John Jumper✉, Richard Evans, [...] Demis Hassabis✉

[Nature](#) 596, 583–589 (2021) | [Cite this article](#)438k Accesses | 279 Citations | 2796 Altmetric | [Metrics](#)

### Abstract

Proteins are essential to life, and understanding their structures is key to understanding their function. Through an enormous experimental effort, the three-dimensional structures of around 100,000 unique proteins have been determined, but this represents only a small fraction of the billions of known protein sequences<sup>6,7</sup>. This bottlenecked the months to years of painstaking effort required to predict protein structure. Accurate computational approaches are now enabling large-scale structural bioinformatics. Predicting the three-dimensional structure that a protein will adopt based solely on its amino acid sequence—the final component of the ‘protein folding problem’<sup>8</sup>—has been an impor-



## GPT-4 Technical Report

OpenAI\*

### Abstract

development of GPT-4, a large-scale, multimodal model which can process images and text inputs and produce text outputs. While less capable than humans in many real-world scenarios, GPT-4 exhibits human-level performance on professional and academic benchmarks, including passing a simulated bar exam with a score around the top 10% of test takers. GPT-4 is a Transformer model that was pre-trained to predict the next token in a document. The post-training process results in improved performance on measures of factuality and other metrics that are desired behavior. A core component of this project was developing training and optimization methods that behave predictably across a wide range of tasks. This allowed us to accurately predict some aspects of GPT-4’s behavior, such as its ability to generate text that is based on models trained with no more than 1/1,000th the compute of the previous version.



The winning solution (AUC 0.99) of the **Camelyon challenge on detecting metastatic cancer** beats the human pathologist benchmark (AUC 0.96)

A CNN designed by a team at the University of Toronto wins the ImageNet Challenge bringing down the error rate to 16% (compared to 25% 2011)

Fei Fei Li and colleagues at Princeton University start to collect a large database of annotated images, the **ImageNet**

A group around Yann LeCun successfully applies a back-propagation algorithm to a multi-layer neural network, **recognizing handwritten ZIP codes**

Frank Rosenblatt develops the **Perceptron**, an early neural network enabling pattern recognition based on a two-layer learning network

2016

2015

2012

2009

2007

2006

1989

1986

1957

A CNN by team from Microsoft beats the human benchmark (5% error rate) by bringing down the error rate to 3% in the ImageNet Challenge

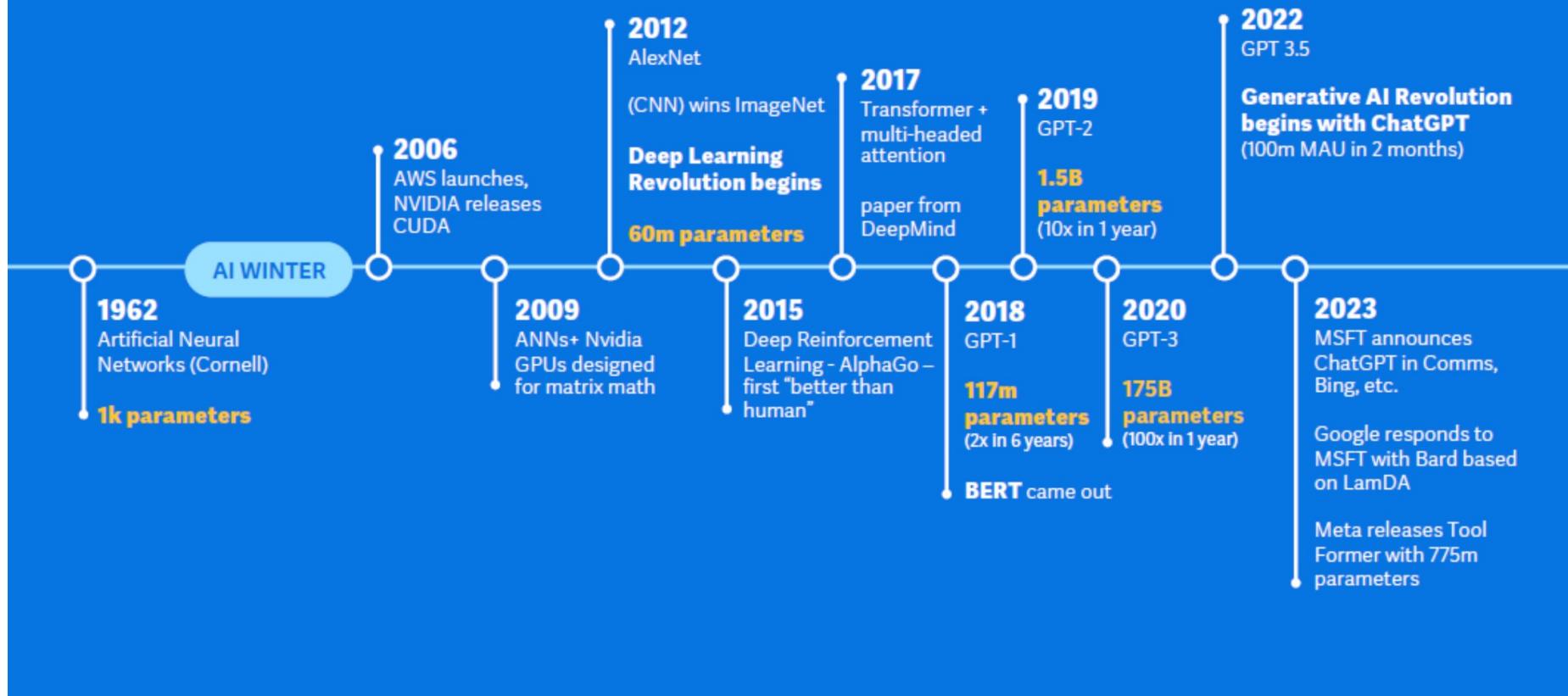
A group around Andrew Ng introduce **Graphics Processing Units (GPUs)** for Deep Learning making them applicable on a large scale

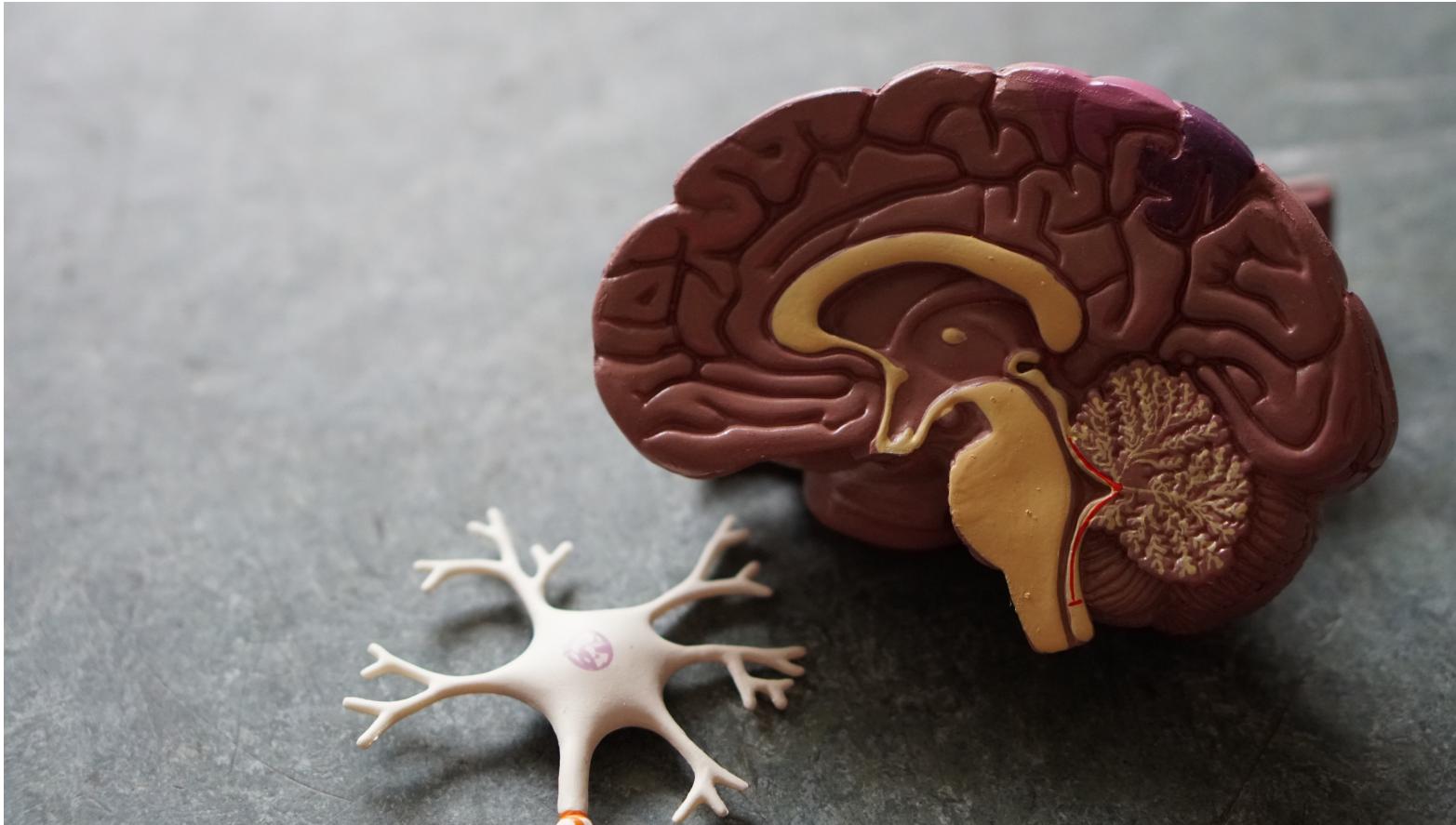
Hinton summarizes ideas of **multilayer neural networks** and training them to generate sensory data rather than to classify it

Rumelhart, Hinton, and Williams introduce **backpropagation** as a learning procedure for "networks of neuron-like units"



# Where Did Generative AI Come From?









# Contents

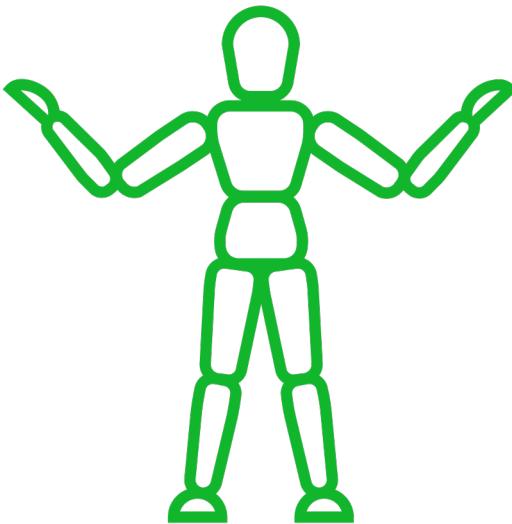
<b>Website</b>	vii	3.2 Random Variables . . . . .	56
<b>Acknowledgments</b>	viii	3.3 Probability Distributions . . . . .	56
<b>Notation</b>	xi	3.4 Marginal Probability . . . . .	58
<b>1 Introduction</b>	1	3.5 Conditional Probability . . . . .	59
1.1 Who Should Read This Book? . . . . .	8	3.6 The Chain Rule of Conditional Probabilities . . . . .	59
1.2 Historical Trends in Deep Learning . . . . .	11	3.7 Independence and Conditional Independence . . . . .	60
<b>I Applied Math and Machine Learning Basics</b>	29	3.8 Expectation, Variance and Covariance . . . . .	60
<b>2 Linear Algebra</b>	31	3.9 Common Probability Distributions . . . . .	62
2.1 Scalars, Vectors, Matrices and Tensors . . . . .	31	3.10 Useful Properties of Common Functions . . . . .	67
2.2 Multiplying Matrices and Vectors . . . . .	34	3.11 Bayes' Rule . . . . .	70
2.3 Identity and Inverse Matrices . . . . .	36	3.12 Technical Details of Continuous Variables . . . . .	71
2.4 Linear Dependence and Span . . . . .	37	3.13 Information Theory . . . . .	73
2.5 Norms . . . . .	39	3.14 Structured Probabilistic Models . . . . .	75
2.6 Special Kinds of Matrices and Vectors . . . . .	40		
2.7 Eigendecomposition . . . . .	42		
2.8 Singular Value Decomposition . . . . .	44		
2.9 The Moore-Penrose Pseudoinverse . . . . .	45		
2.10 The Trace Operator . . . . .	46		
2.11 The Determinant . . . . .	47		
2.12 Example: Principal Components Analysis . . . . .	48		
<b>3 Probability and Information Theory</b>	53		
3.1 Why Probability? . . . . .	54		
		<b>4 Numerical Computation</b> . . . . .	80
		4.1 Overflow and Underflow . . . . .	80
		4.2 Poor Conditioning . . . . .	82
		4.3 Gradient-Based Optimization . . . . .	82
		4.4 Constrained Optimization . . . . .	93
		4.5 Example: Linear Least Squares . . . . .	96
		<b>5 Machine Learning Basics</b> . . . . .	98
		5.1 Learning Algorithms . . . . .	99
		5.2 Capacity, Overfitting and Underfitting . . . . .	110
		5.3 Hyperparameters and Validation Sets . . . . .	120
		5.4 Estimators, Bias and Variance . . . . .	122
		5.5 Maximum Likelihood Estimation . . . . .	131
		5.6 Bayesian Statistics . . . . .	135
		5.7 Supervised Learning Algorithms . . . . .	140
		5.8 Unsupervised Learning Algorithms . . . . .	146
		5.9 Stochastic Gradient Descent . . . . .	151
		5.10 Building a Machine Learning Algorithm . . . . .	153
		5.11 Challenges Motivating Deep Learning . . . . .	155
		<b>II Deep Networks: Modern Practices</b> . . . . .	166



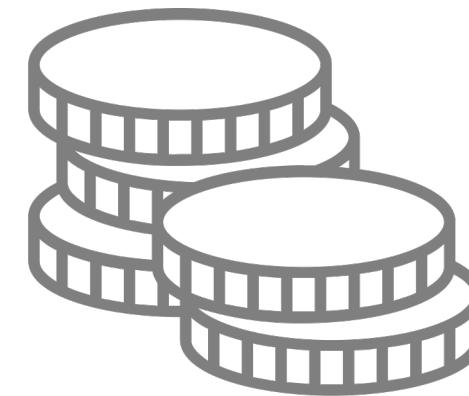
Data



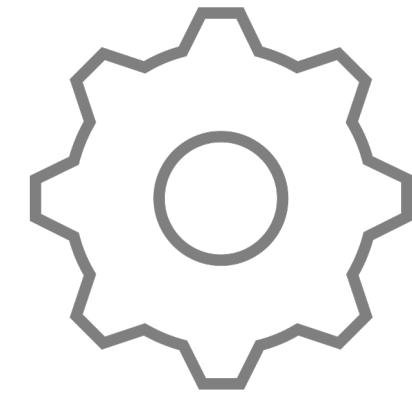
Model



Loss



Learning algorithm



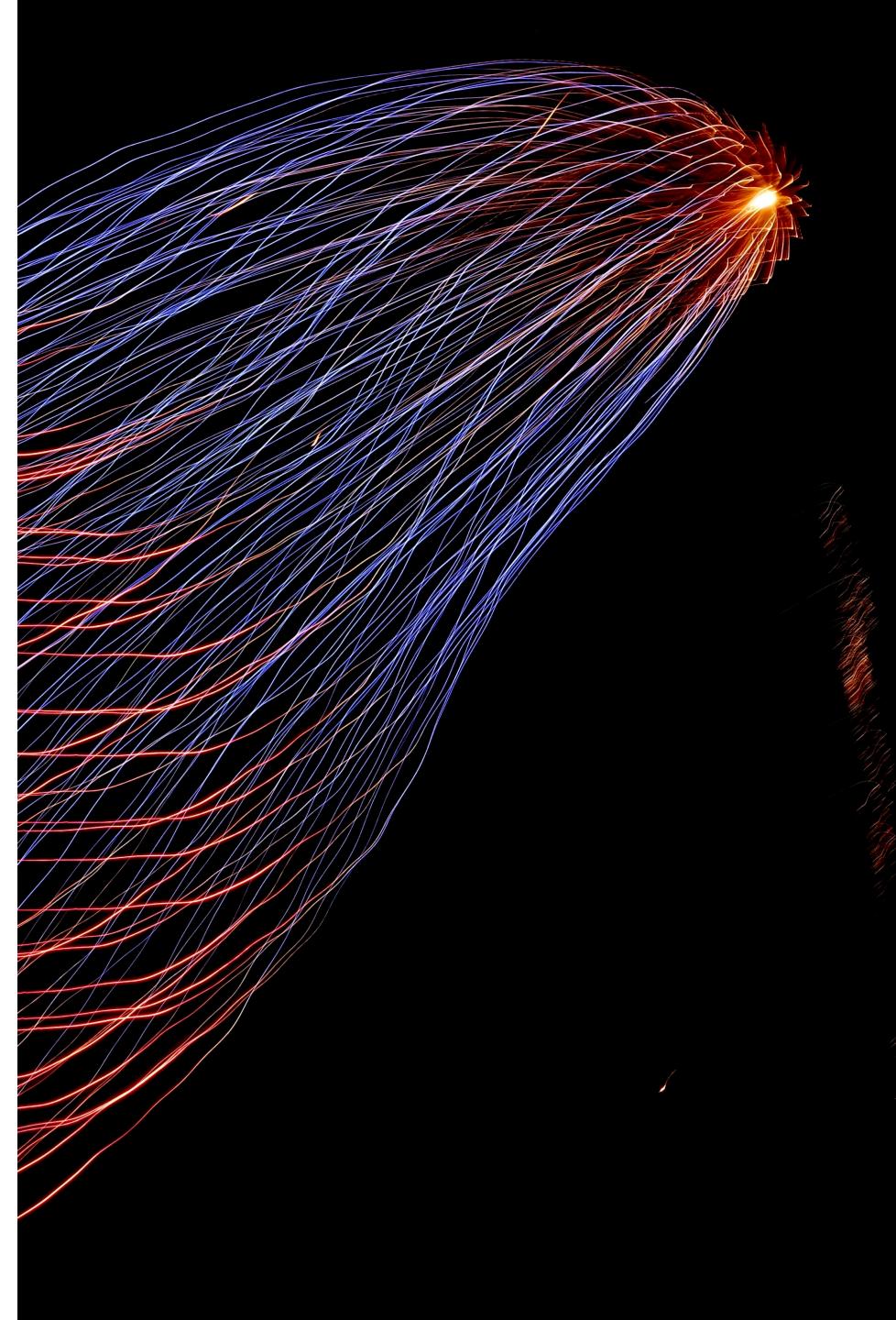
# Data

We have a bunch of *numeric* data.

Each sample is made of:

- input real vector  $x_i$
- output real scalar  $y_i$

In fact, input and outputs can be vectors, matrices, or even tensors (remember the numeric representation of a picture?), but we will keep it simple.





# Model

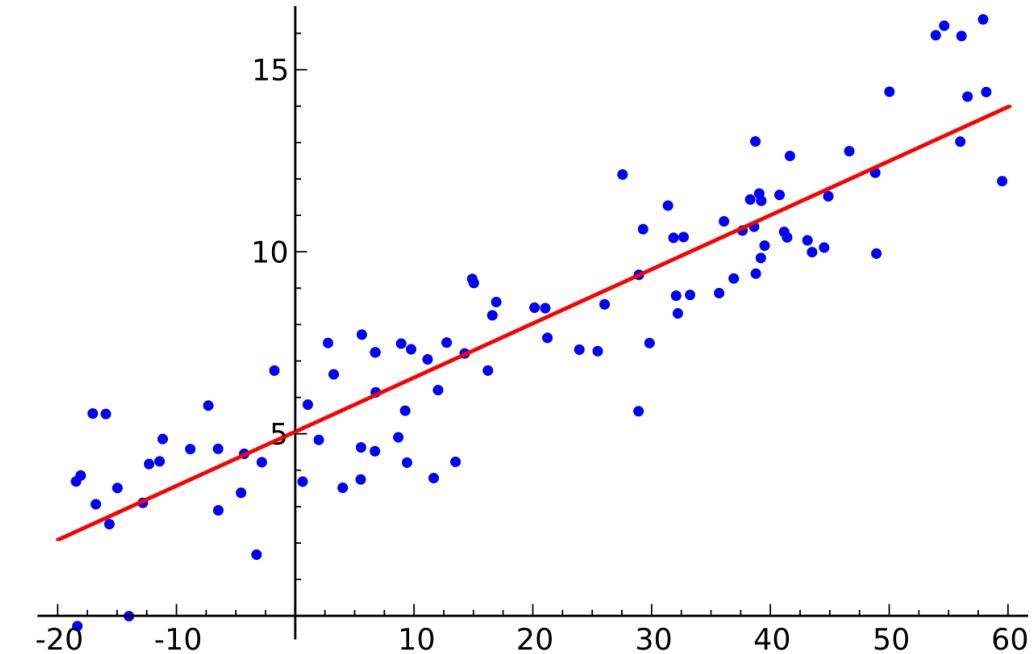
A completely legitimate model:

$$y = \mathbf{w}^T \mathbf{x} + b$$

where:

$$\begin{aligned}\mathbf{w}, \mathbf{x} &\in R^p \\ y, b &\in R\end{aligned}$$

Yet, this model has a problem...





# Model

A completely legitimate model:

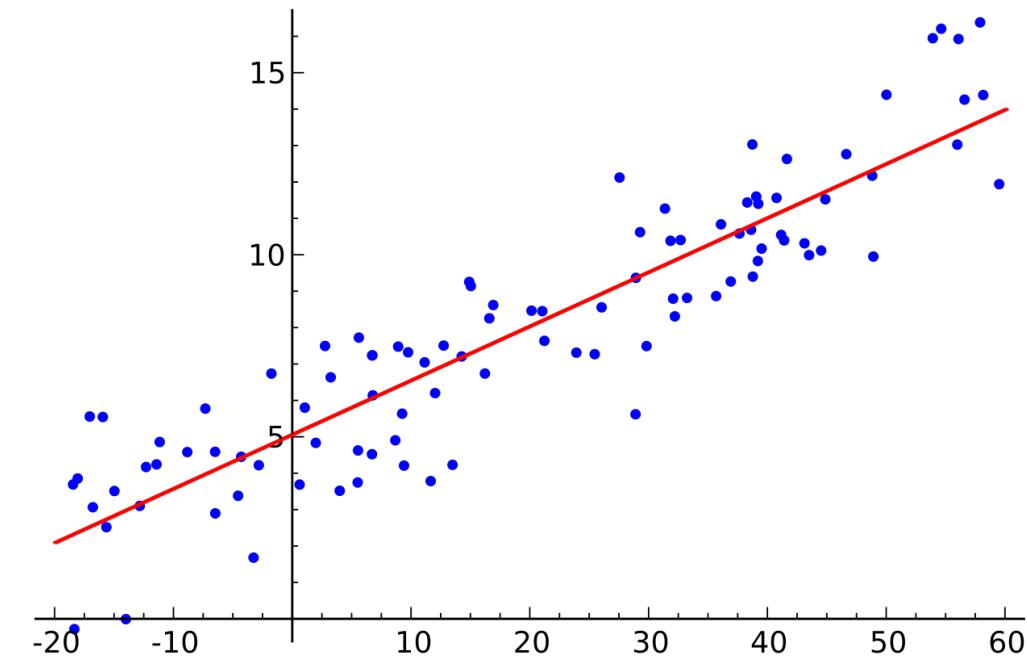
$$y = \mathbf{w}^T \mathbf{x} + b$$

where:

$$\begin{aligned}\mathbf{w}, \mathbf{x} &\in R^p \\ y, b &\in R\end{aligned}$$

Yet, this model has a problem.

It is linear! So, it can only form straight lines in 2D, planes in 3D, ... \*





# My First Network

Let's add a non-linear function and call it **activation function**:

$$y = \mathbf{w}^T g(\mathbf{W}\mathbf{x} + \mathbf{b}) + b$$

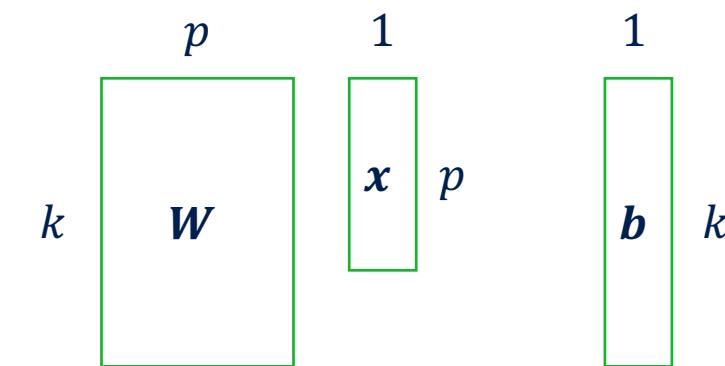
where:

$$\mathbf{W} \in R^{k \times p}$$

$$\mathbf{w}, \mathbf{b} \in R^k$$

$$\mathbf{x} \in R^p$$

$$y, b \in R$$



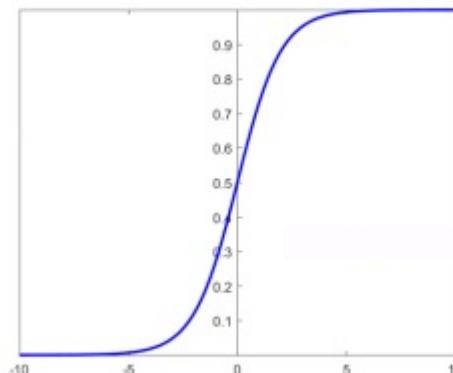


# My First Network

$$y = \mathbf{w}^T g(\mathbf{Wx} + \mathbf{b}) + b$$

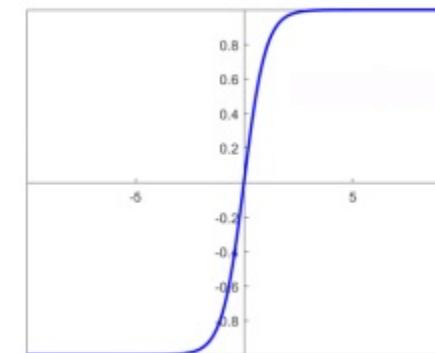
where  $g$  is a non-linear function, applied element-wise.

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



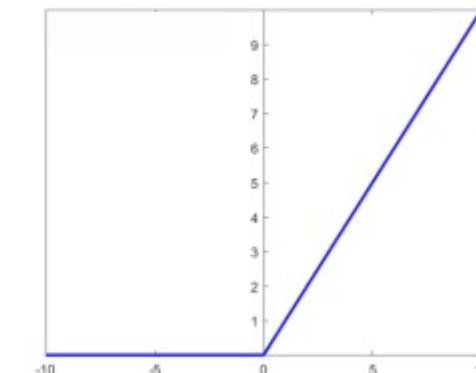
Sigmoid

$$g(x) = \tanh(x)$$



Hyperbolic Tangent

$$g(x) = \max(0, x)$$



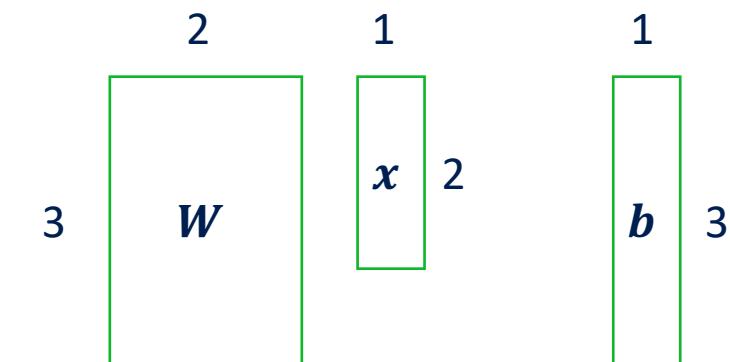
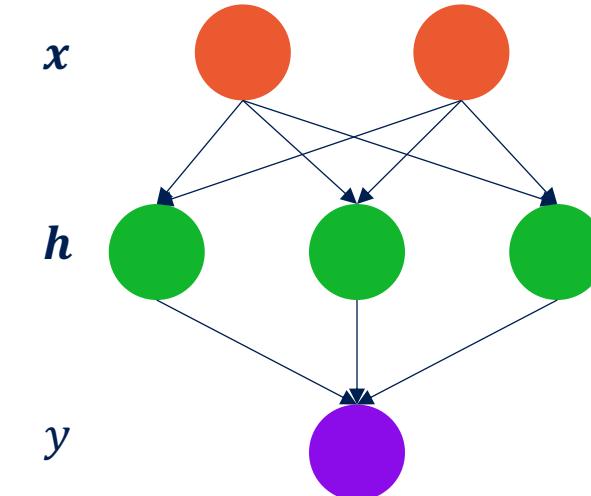
ReLU



# Shallow Networks

We build a feed-forward shallow network.

$$y = \underbrace{\mathbf{w}^T g(\mathbf{W}\mathbf{x} + \mathbf{b})}_h + b$$





# Shallow Networks

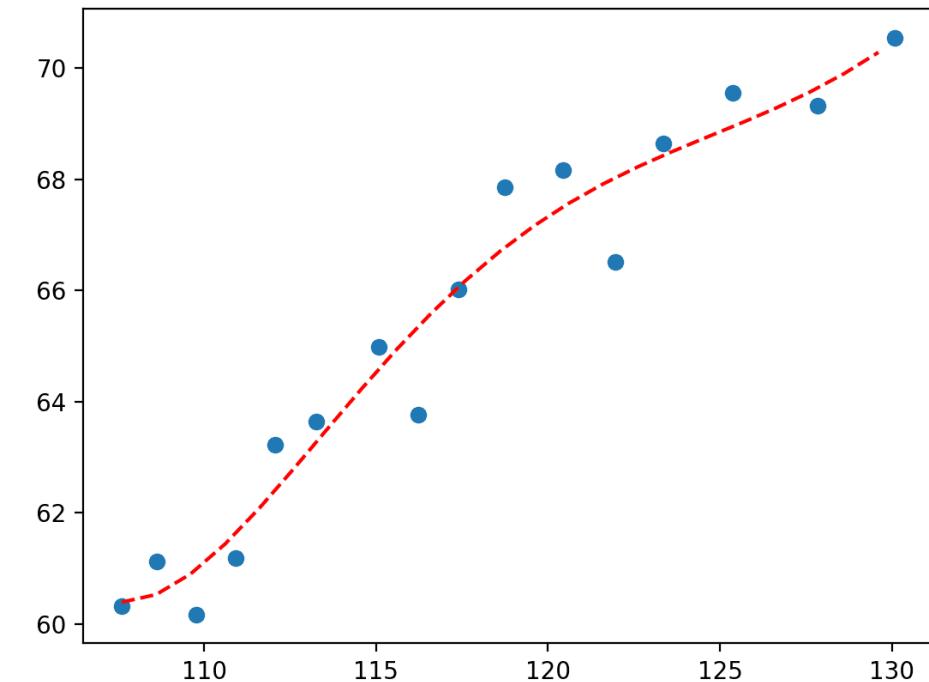
We build a feed-forward shallow network.

$$y = \mathbf{w}^T g(\mathbf{W}\mathbf{x} + \mathbf{b}) + b$$

What did we gain?

The possibility to represent and learn **non-linear functions**.

Which functions?





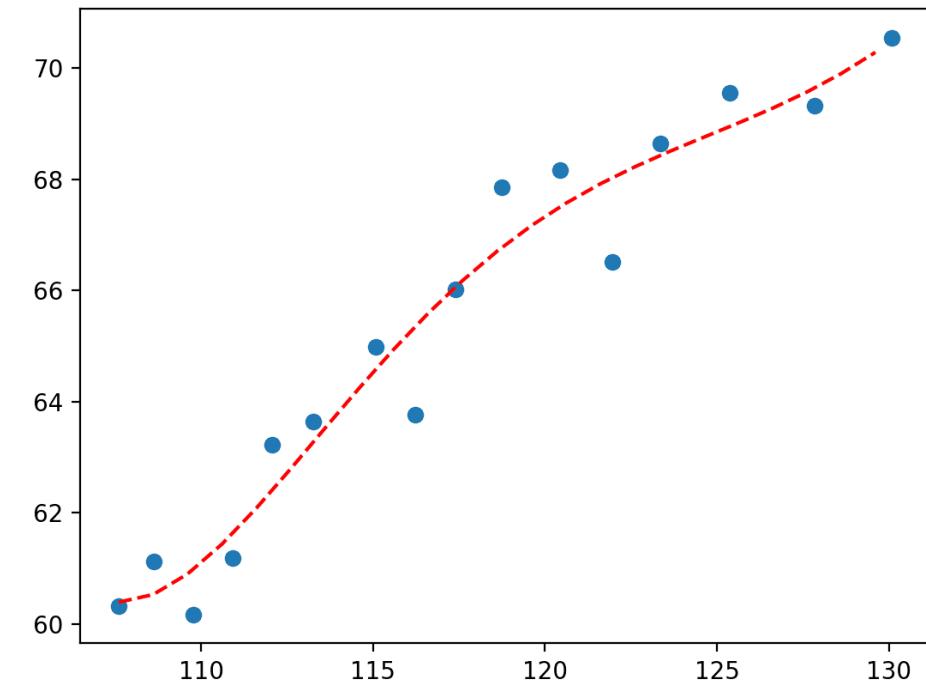
**ALL OF THEM.**



# The Universal Approximation Theorem(s)

Universal Approximation Theorems are proven mathematical statements showing how networks with certain properties can approximate *to an arbitrary level of precision any regular function*.

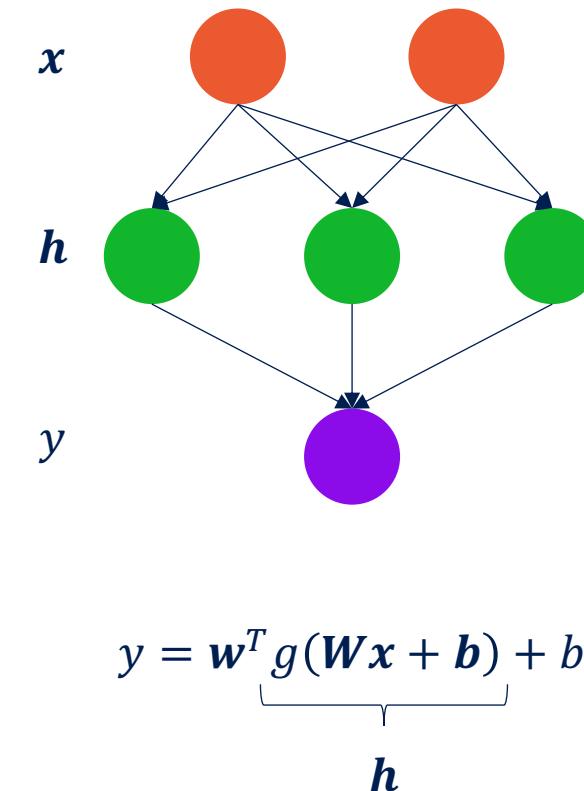
Since 1989, several universal approximation theorems have been proposed: we will discuss just the original one by Hornik & Cybenko.





# The Universal Approximation Theorem

A feedforward network with a linear output layer and at least one hidden layer with a suitable activation function – such as the ones shown before – can approximate any continuous function from one finite-dimensional space to another, defined on a compact set, with any desired non-zero amount of error, provided that the network is given enough hidden units.





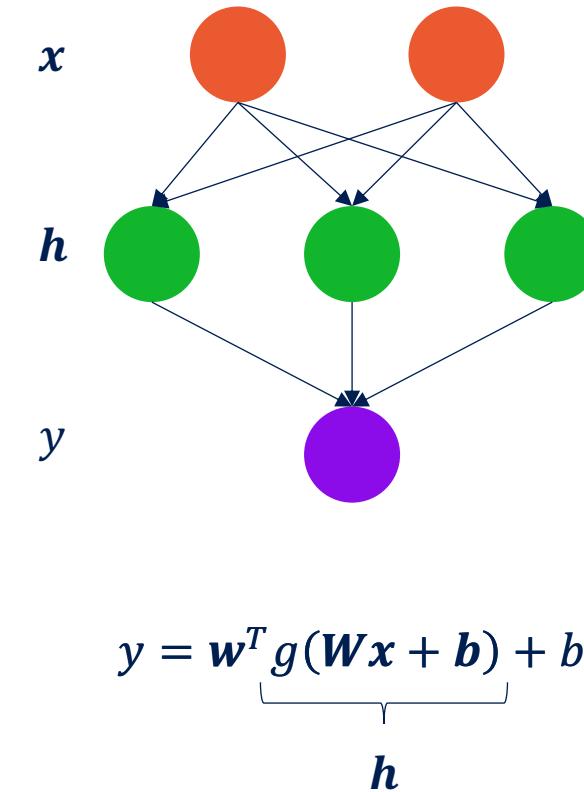
**REALLY?**

**ARE YOU KIDDING ME**



# The Universal Approximation Theorem

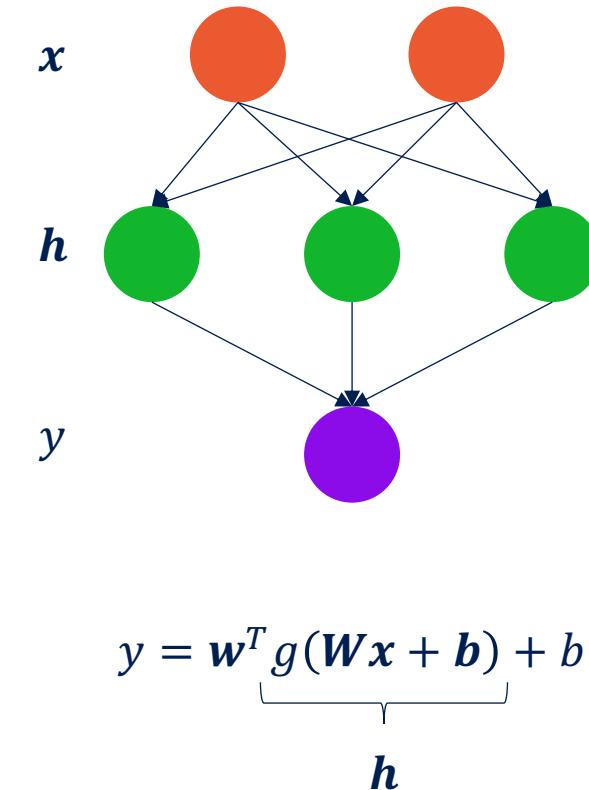
A feedforward network with a linear output layer and at least one hidden layer with a suitable activation function – such as the ones shown before – can approximate any continuous function from one finite-dimensional space to another, defined on a compact set, with any desired non-zero amount of error, provided that the network is given enough hidden units.





# The Universal Approximation Theorem

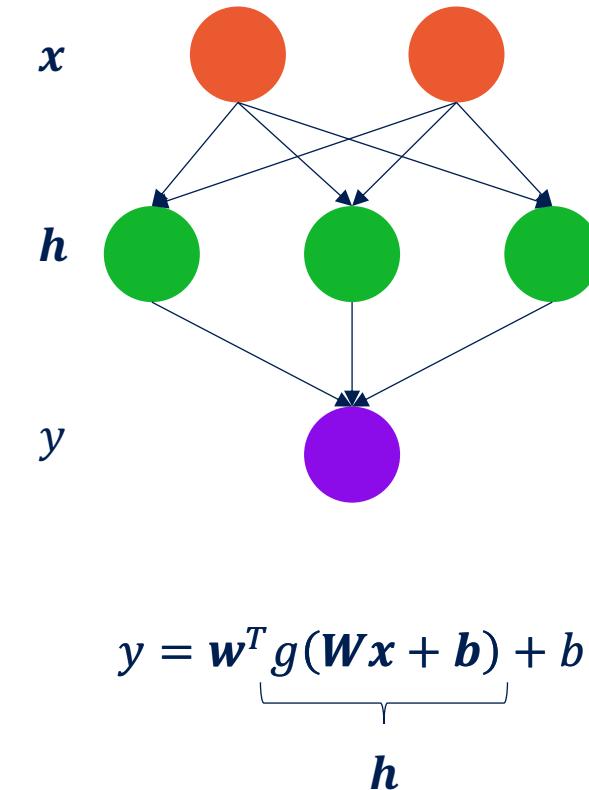
A feedforward network with a linear output layer and at least one hidden layer with a suitable activation function – such as the ones shown before – can approximate any continuous function from one finite-dimensional space to another, defined on a compact set, with any desired non-zero amount of error, provided that the network is given enough hidden units.





# The Universal Approximation Theorem

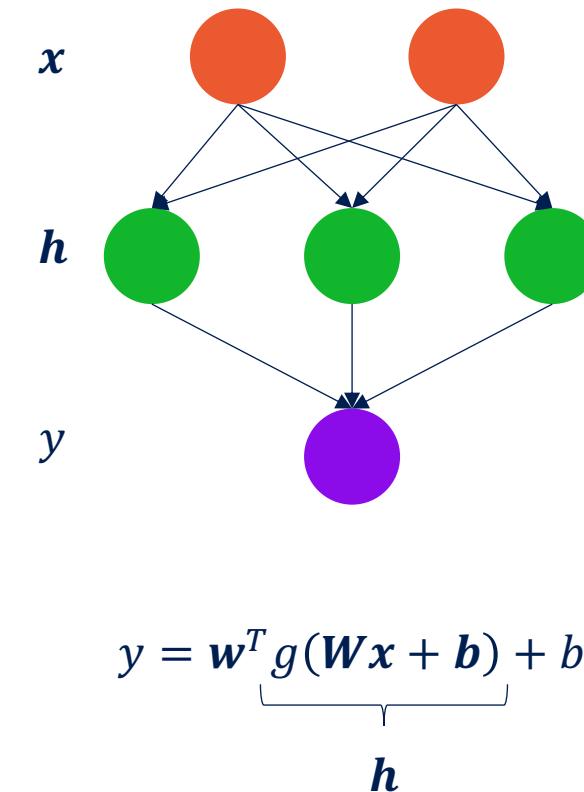
A feedforward network with a linear output layer and at least one hidden layer with a suitable activation function – such as the ones shown before – can approximate any continuous function from one finite-dimensional space to another, defined on a compact set, with any desired non-zero amount of error, provided that the network is given enough hidden units.





# The Universal Approximation Theorem

A feedforward network with a linear output layer and at least one hidden layer with a suitable activation function – such as the ones shown before – can approximate any continuous function from one finite-dimensional space to another, defined on a compact set, with any desired non-zero amount of error, provided that the network is given enough hidden units.



# Discussion

What are the issues with this theorem?

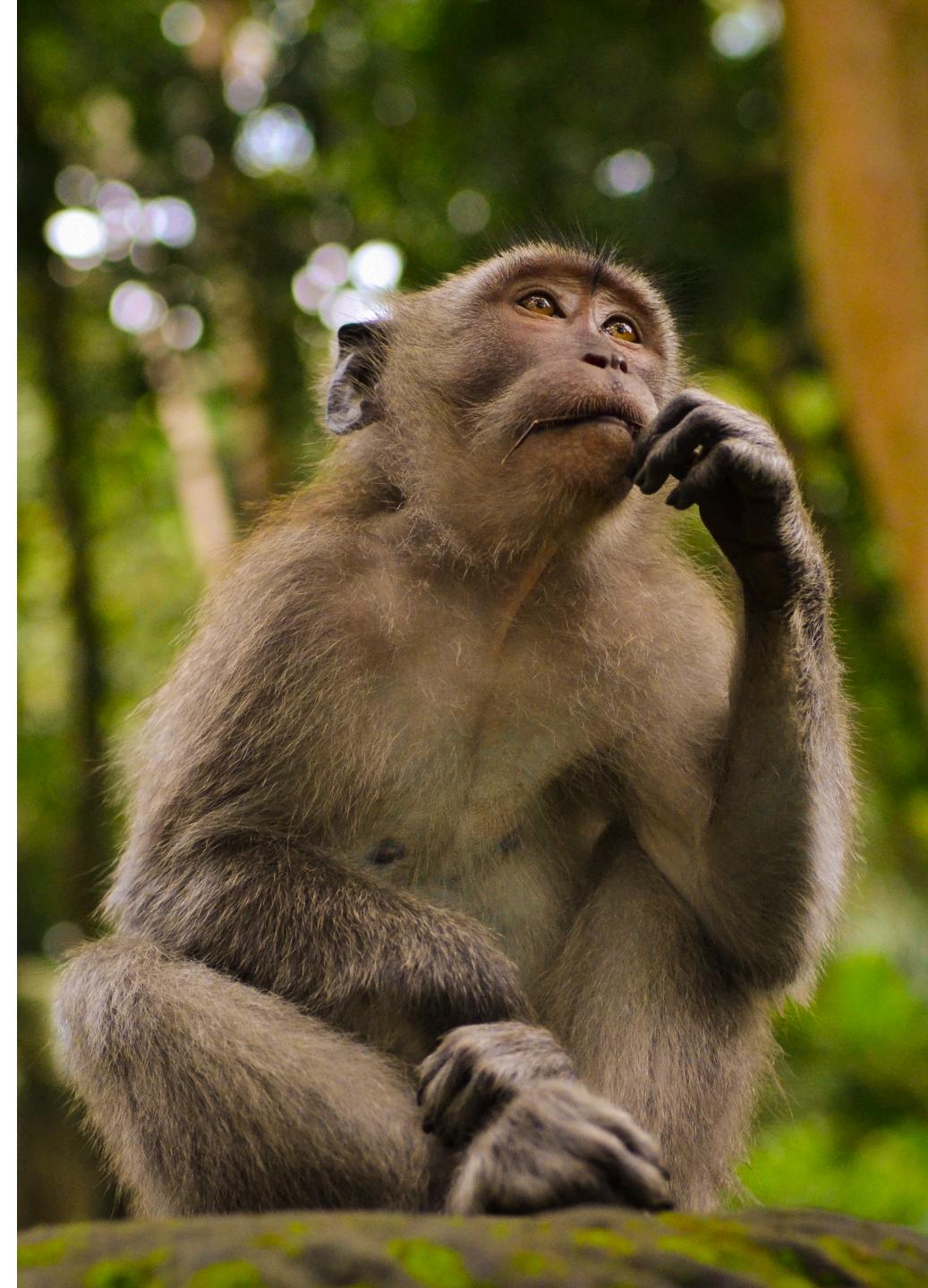
Why aren't feed-forward networks the only AI model?



# The Universal Approximation Theorem

There are two annoying parts of the theorem:

1. “A specific network” (i.e., a specific choice of parameters) *can* achieve specific level of error. Yeah, **but which network?**
2. “Provided that the network is given enough hidden units”. OK, **but how many is enough?**



# The issue with the Universal Approximation Theorem

“Provided that the network is given enough hidden units”. OK, **but how many is enough?**

Bounds on the size of the hidden layer were computed: the bounds are **exponential** in the input size.  
[1]

Which is very bad.



# The issue with the Universal Approximation Theorem

"A specific network" (i.e., a specific choice of parameters) *can* achieve specific level of error. Yeah, **but which network?**

It's your problem to find it!

You need a **learning algorithm**. Yet, very large networks, which are required to capture very complex relationships, are **very hard to train**.





# Going Deep

# Why Deep?

Both theoretical [1] and practical [2] results suggest that deep networks are more powerful than shallow ones.

That is, they need less parameters to approximate complex functions.



[1] Montufar, G. F. et al, (2014). On the number of linear regions of deep neural networks

[2] Goodfellow, I. J. et al. (2014). Multi-digit number recognition from Street View imagery using deep convolutional neural networks



# Why Deep?

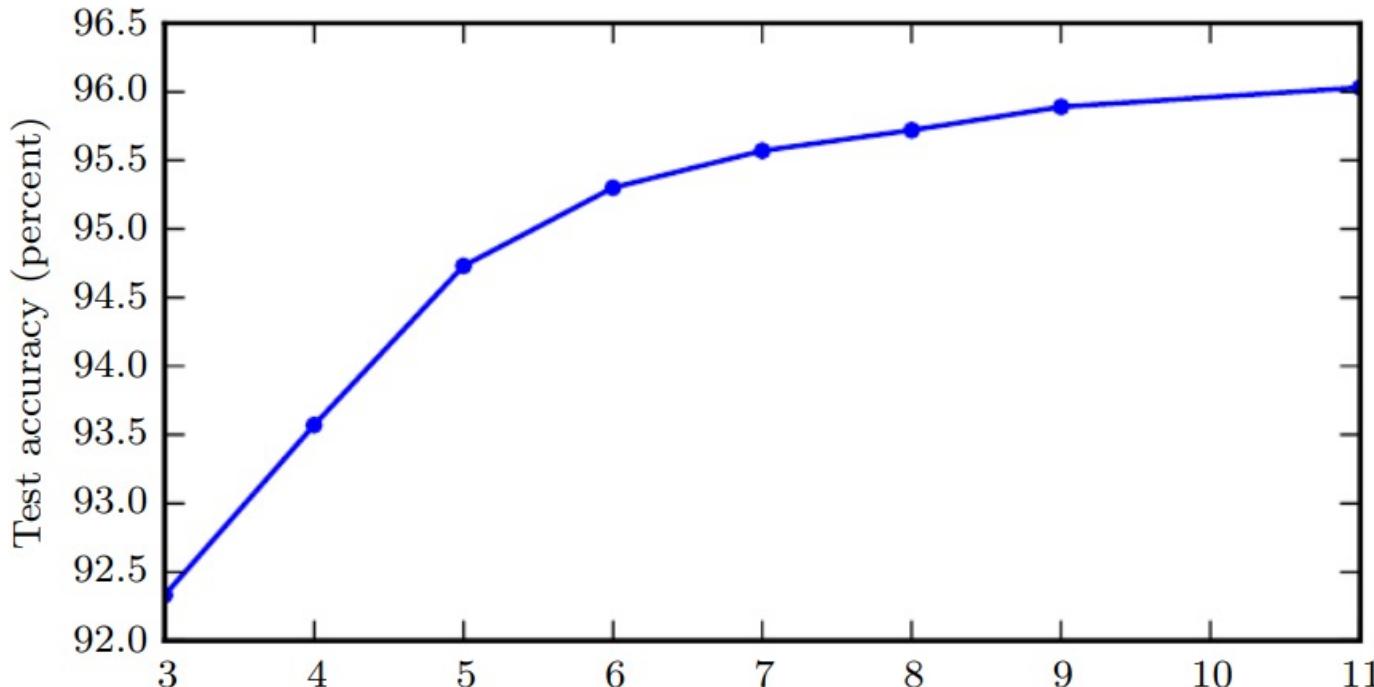


Figure 6.6: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Data from [Goodfellow et al. \(2014d\)](#). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.



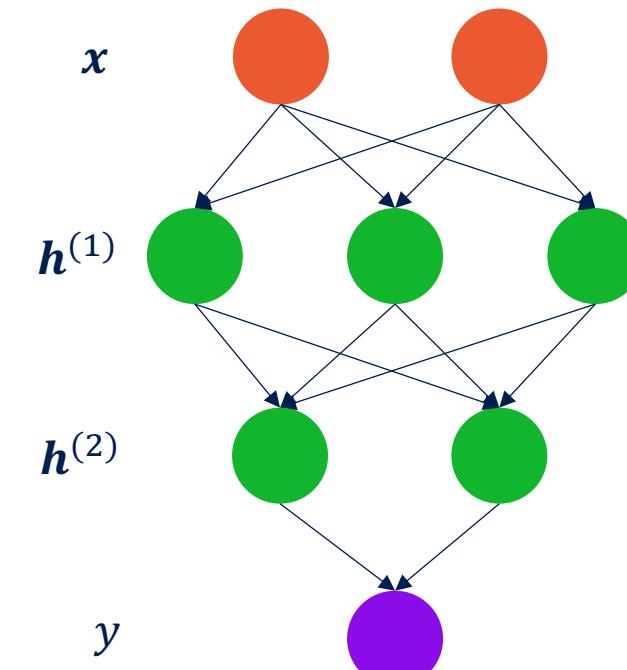
# My First Deep Network

A network with two layers:

$$y = \mathbf{w}^T g\left(\mathbf{W}^{(2)} g\left(\mathbf{W}^{(1)} \mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right) + b$$

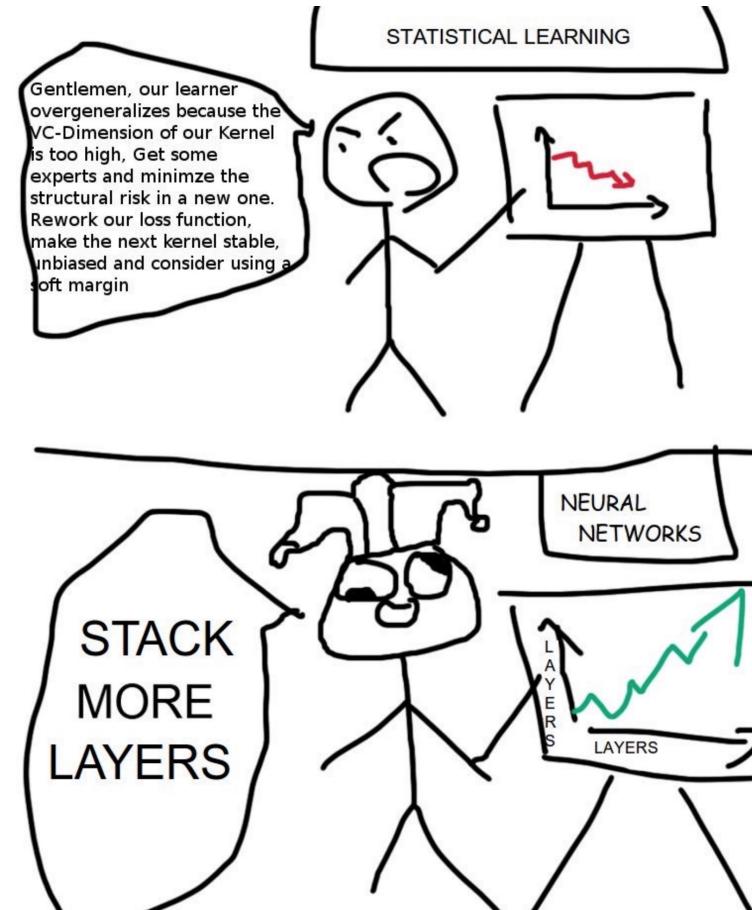
$\underbrace{\hspace{10em}}_{\mathbf{h}^{(1)}}$

$\underbrace{\hspace{10em}}_{\mathbf{h}^{(2)}}$



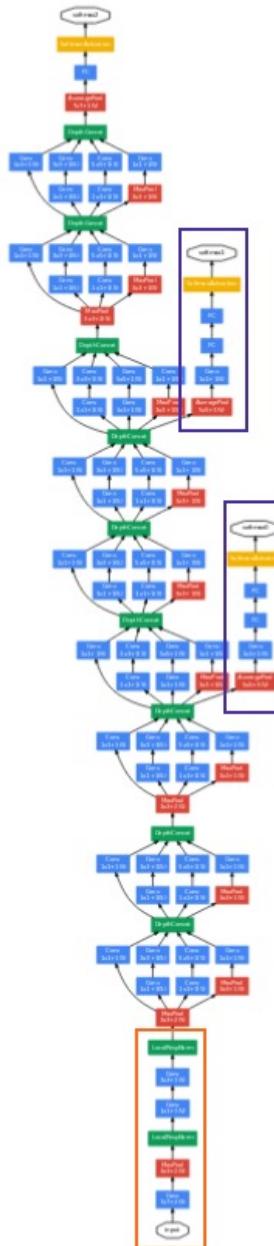


# How Deep?



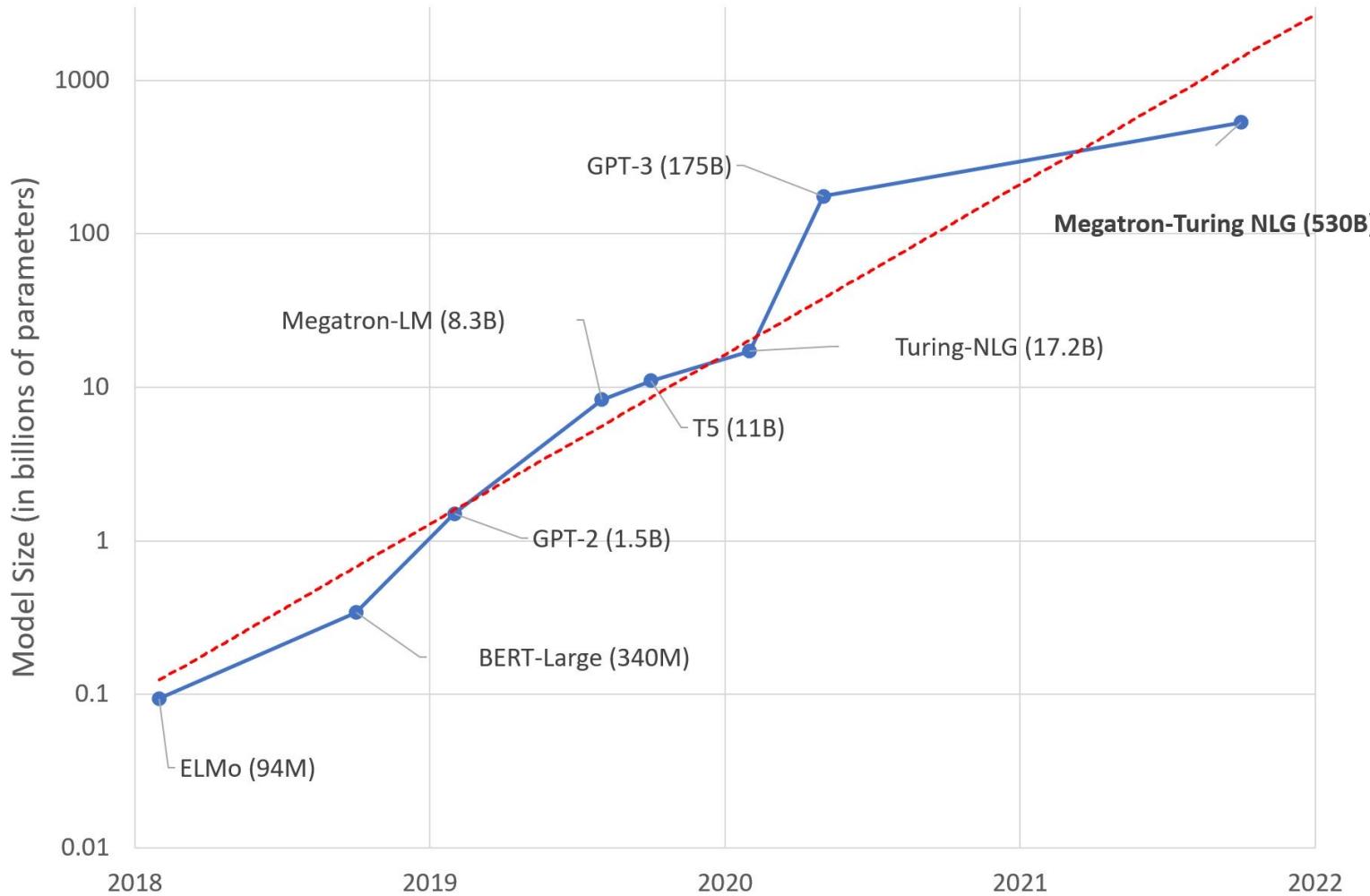


# How Deep?



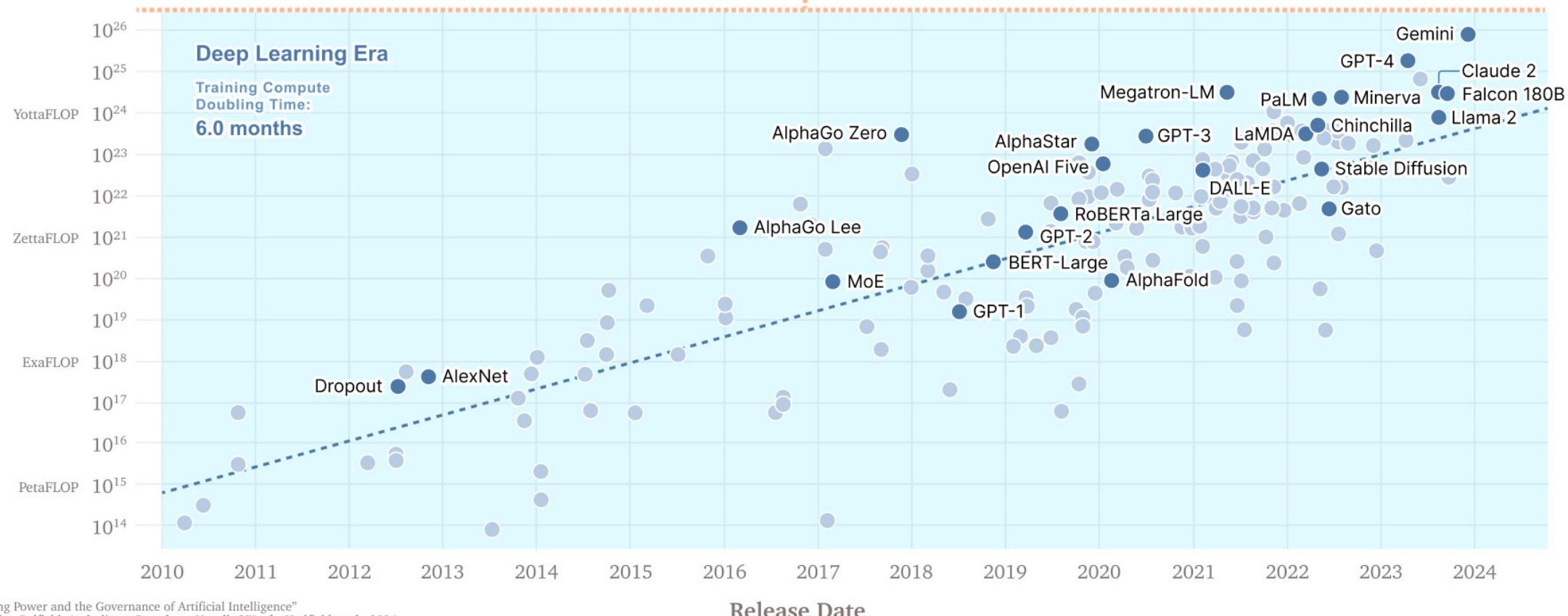


# How Deep?





# Scaling Compute



"Computing Power and the Governance of Artificial Intelligence"  
Sastry, Heim, Belfield, Anderljung, Brundage, Hazell, O'Keefe, Hadfield et al., 2024

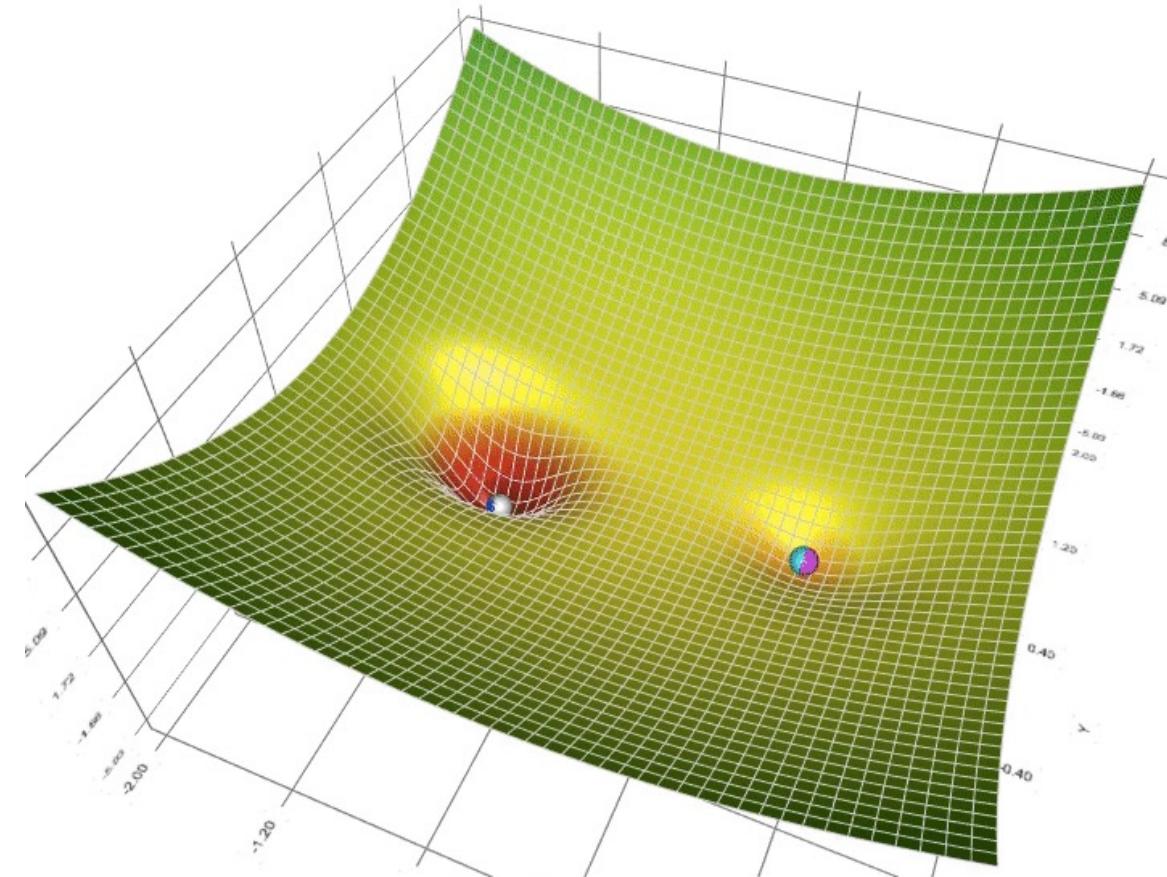


# How to Train your Network

# Remember: what is a learning algorithm?

An algorithm is a **set of instructions** to achieve a task.

A learning algorithm finds the model parameters which **minimize the loss** given a cost function **dependent on some data**.





# Problem Statement

We have *numeric* data. Each of the  $n$  samples is made of:

- input real vector  $\mathbf{x}_i$
- output real scalar  $y_i$

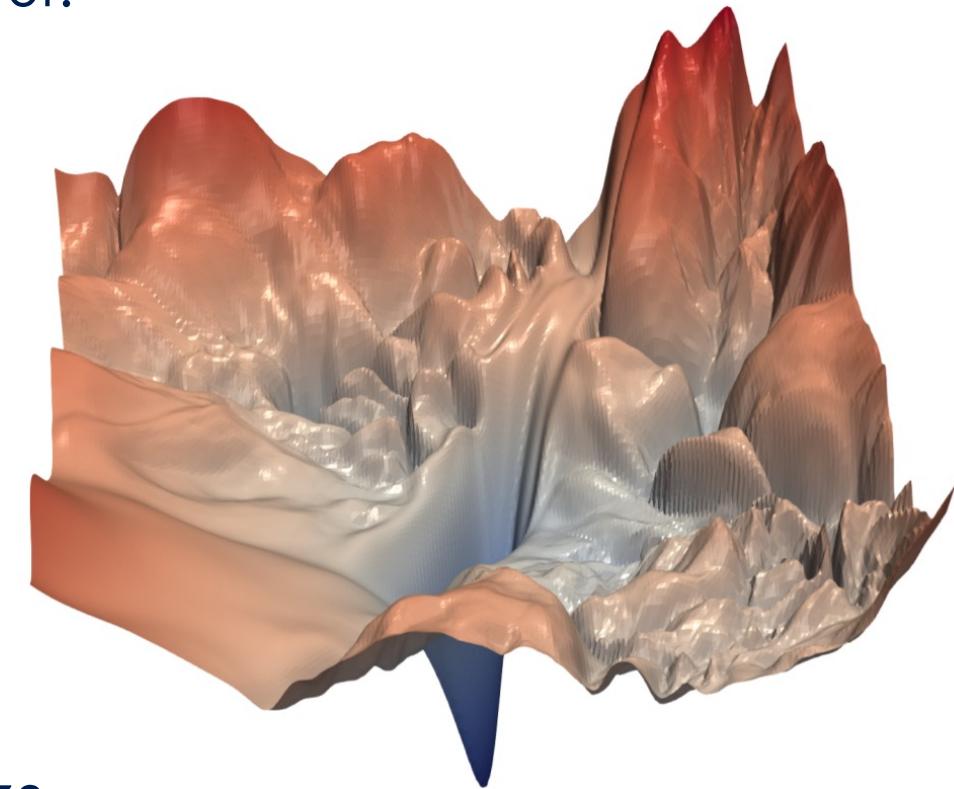
Our model is:

$$y = \mathbf{w}^T g(\mathbf{W}\mathbf{x} + \mathbf{b}) + b$$

Our loss is:

$$L(\mathbf{w}, \mathbf{W}, \mathbf{b}, b) = \sum_{i=1}^n (y_i - \mathbf{w}^T g(\mathbf{W}\mathbf{x}_i + \mathbf{b}) + b)^2$$

The learning algorithm should find  $\mathbf{w}, \mathbf{W}, \mathbf{b}, b$  to minimize  
 $L$





# A Hard Problem Statement

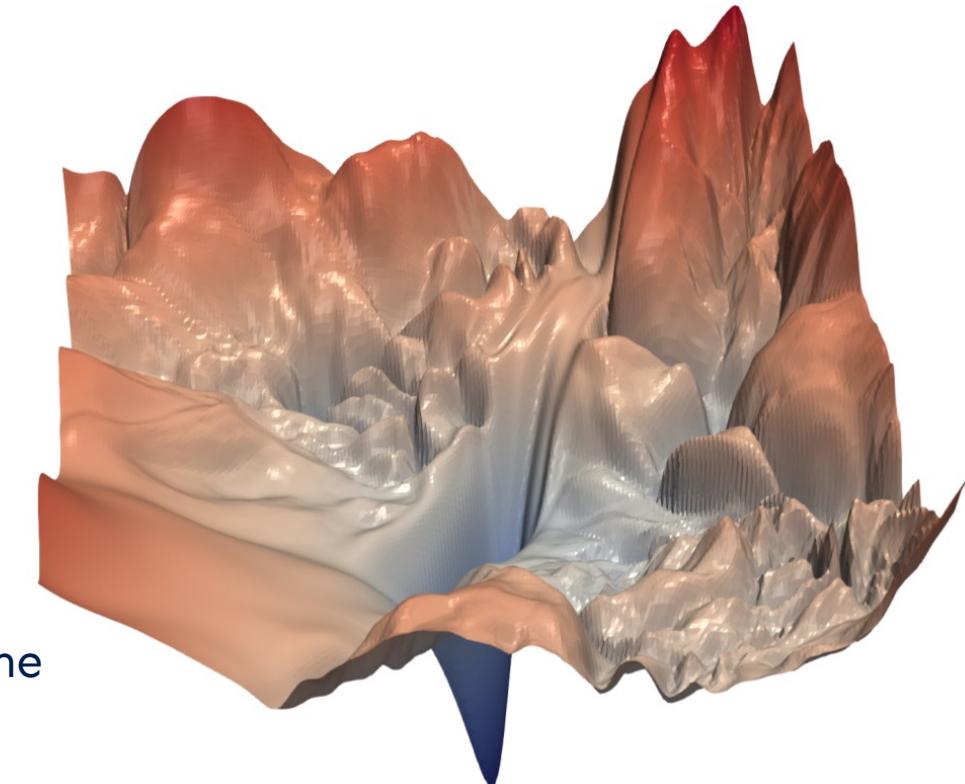
Our loss is:

$$L(\mathbf{w}, \mathbf{W}, \mathbf{b}, b) = \sum_{i=1}^n (y_i - \mathbf{w}^T g(\mathbf{W}\mathbf{x}_i + \mathbf{b}) + b)^2$$

The learning algorithm should find  $\mathbf{w}, \mathbf{W}, \mathbf{b}, b$  to minimize  $L$ .

This is very difficult because of the **non-linear function**  $g$ .

In general,  $L$  is not convex and there is **no guarantee** that the absolute minimum can be found.



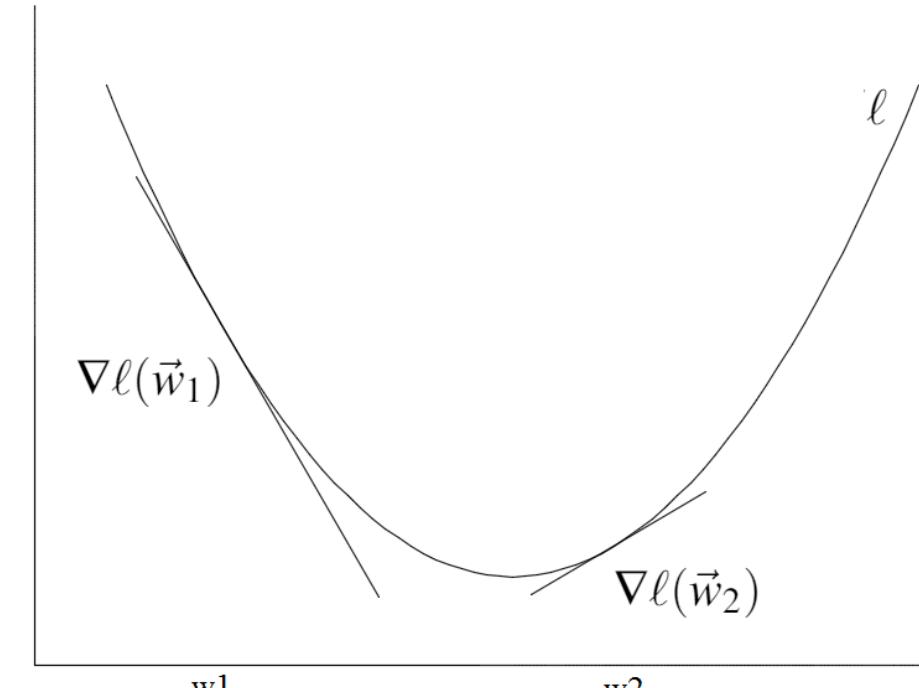


# The Gradient

In 2D, the derivative of a function in a point is the angular coefficient of the **tangent** in that point.

The **gradient** of a function in a point is a vector pointing to the **direction of steepest increment** of the function in that point.

So, the opposite of the gradient points to the direction of **steepest decrease**.





# Gradient Descent

Our loss is:

$$L(\mathbf{w}, \mathbf{W}, \mathbf{b}, b) = \sum_{i=1}^n (y_i - \mathbf{w}^T g(\mathbf{W}\mathbf{x}_i + \mathbf{b}) + b)^2$$

The learning algorithm should find  $\boldsymbol{\vartheta} = (\mathbf{w}, \mathbf{W}, \mathbf{b}, b)$  to minimize  $L$ .

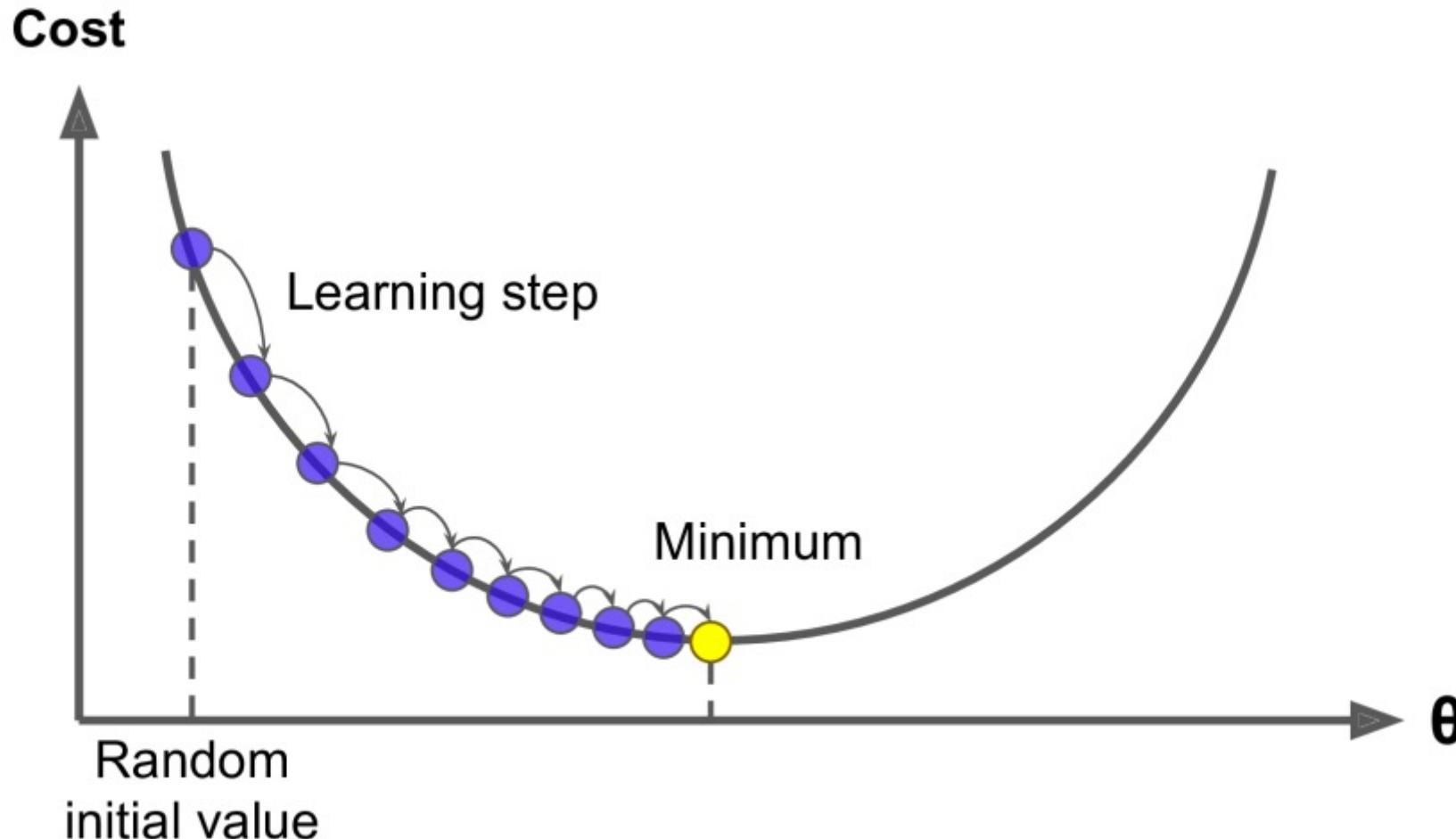
## Gradient descent

1. Initialize  $\boldsymbol{\vartheta}_0$  at random
2. Set  $\boldsymbol{\vartheta}_i = \boldsymbol{\vartheta}_{i-1} - \alpha \frac{\partial L(\boldsymbol{\vartheta})}{\partial \boldsymbol{\vartheta}}|_{\boldsymbol{\vartheta}=\boldsymbol{\vartheta}_{i-1}}$
3. Check if some terminal condition is met. If yes, output  $\boldsymbol{\vartheta}_i$ , if no, go to 2

$\alpha$  is the **learning rate** and is very important...

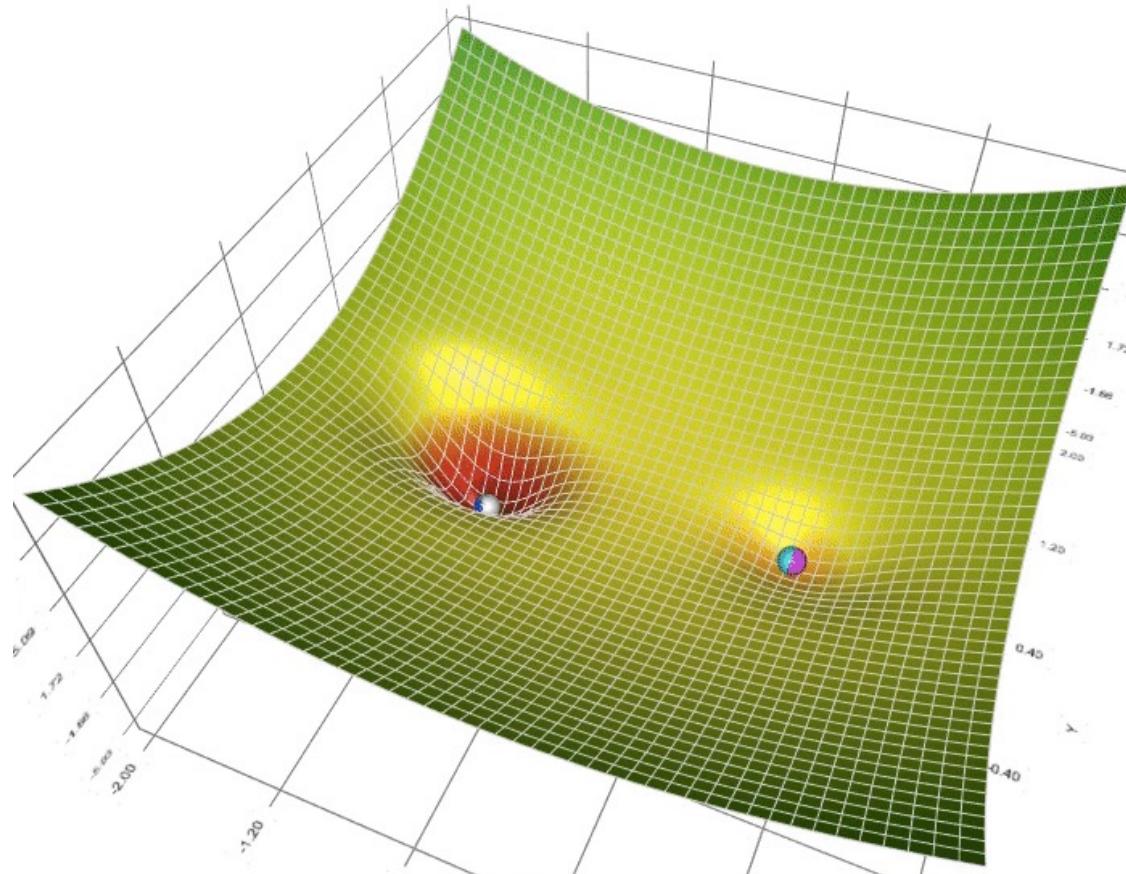


# How does it Work?



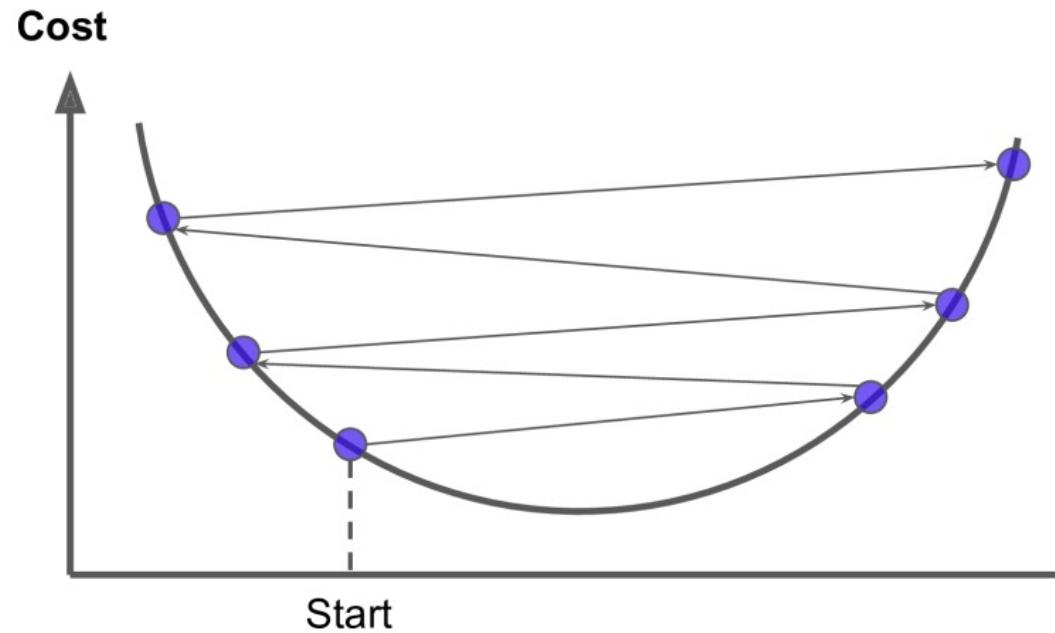


# How does it Work?

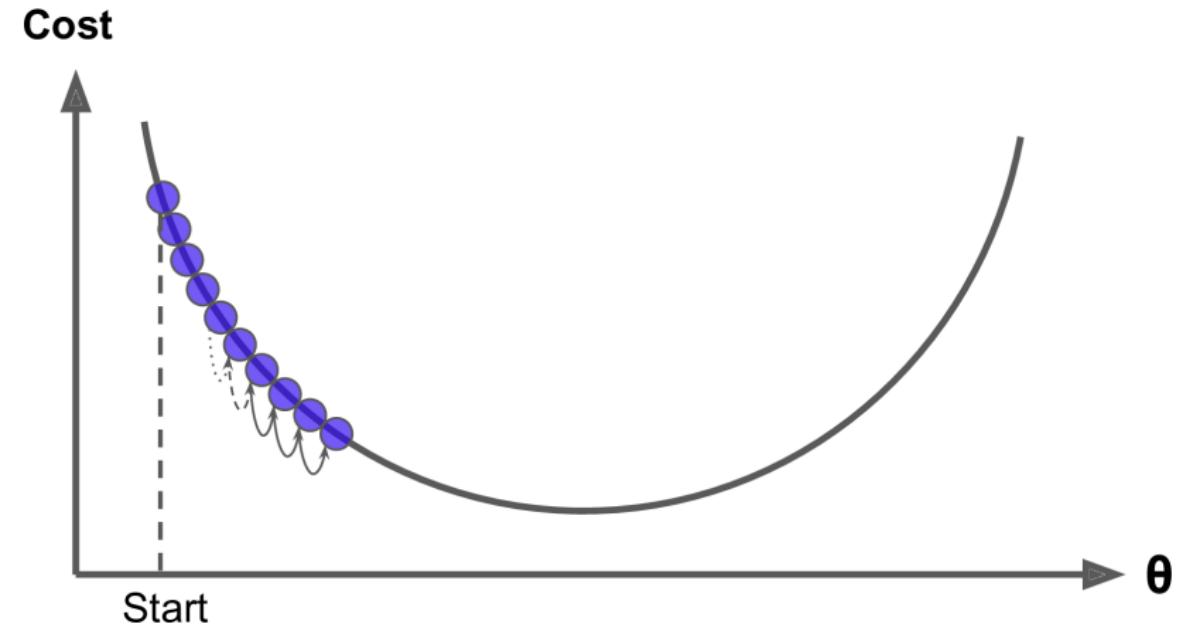




# Does it Work?



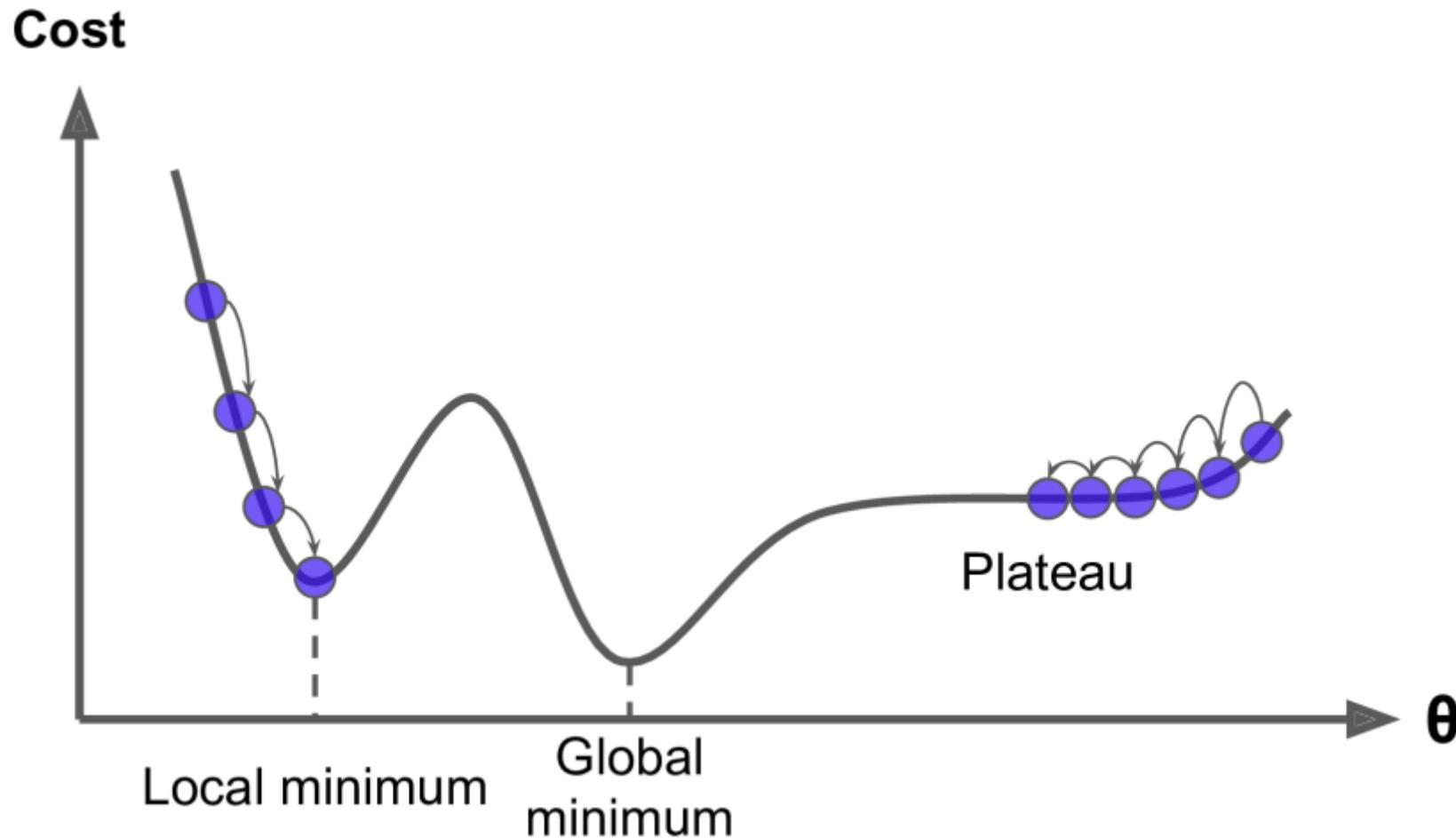
Learning rate too large ->  
no convergence



Learning rate too small ->  
takes forever



# Does it Work?





# Does it Work?

Let the loss function  $L$  be differentiable and let its gradient  $\frac{\partial L(\theta)}{\partial \theta}$  be Lipschitz-continuous (note: we do not require convexity!).

Then, if the learning rate  $\alpha$  is lower than a suitable constant depending on  $L$ , the gradient descent algorithm converges to a **local minimum**.

If  $L$  is convex, that will be also your global minimum.

Usually, you do not know the value of the magic constant – and you cannot compute it either!



# Stochastic Gradient Descent

Most of the time, the loss is made of a sum over all the samples:  $L(\boldsymbol{\vartheta}) = \sum_i L_i(\boldsymbol{\vartheta})$ .

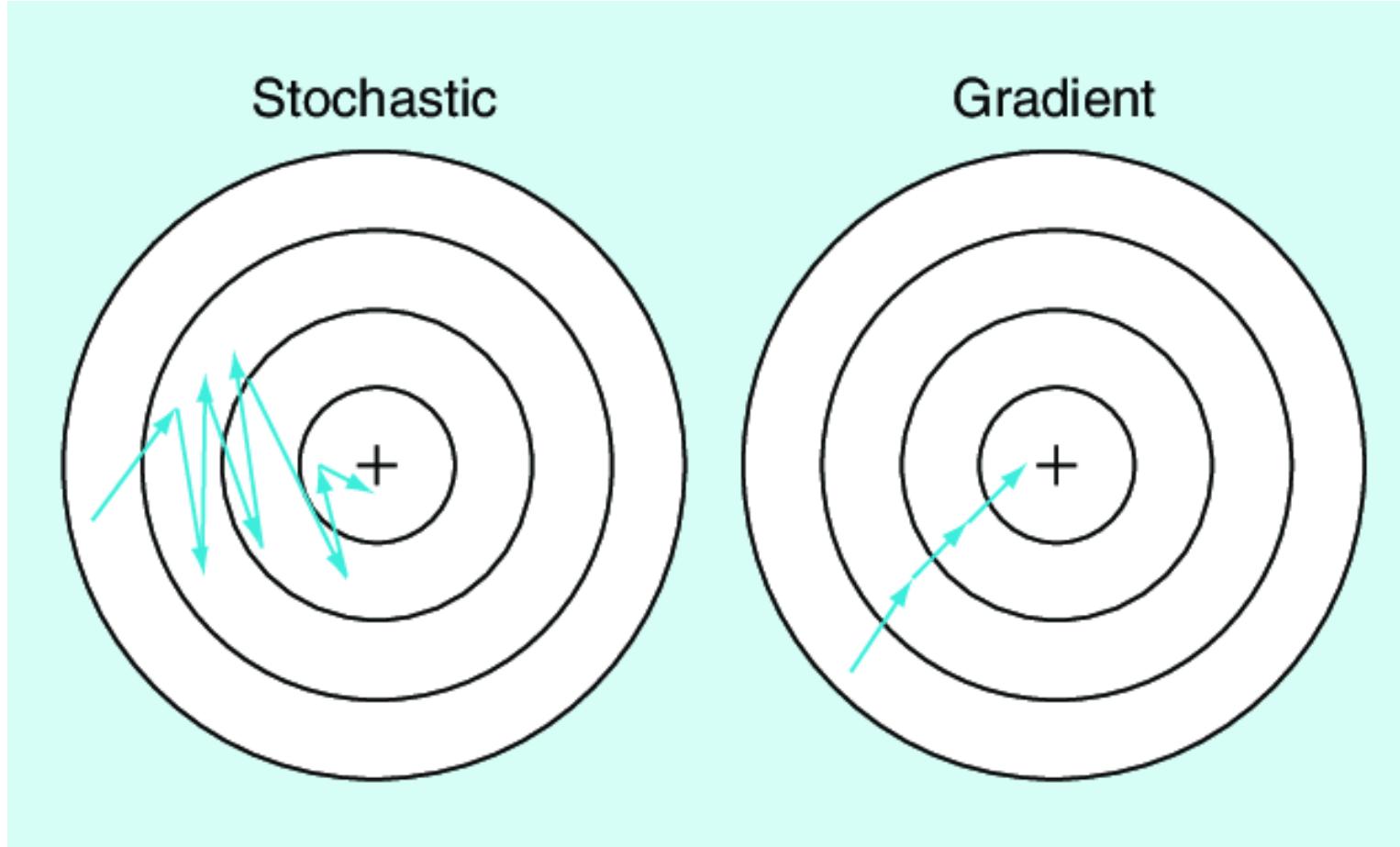
Why should we use all the samples? Let's just pick one!

The gradient on a single sample is **much faster** to compute, and, on average,

$$E_i[\nabla L_i(\boldsymbol{\vartheta})] = \frac{1}{n} \sum_i \nabla L_i(\boldsymbol{\vartheta}) = \nabla \left( \frac{1}{n} \sum_i L_i(\boldsymbol{\vartheta}) \right) = \nabla L(\boldsymbol{\vartheta})$$



# Stochastic Gradient Descent



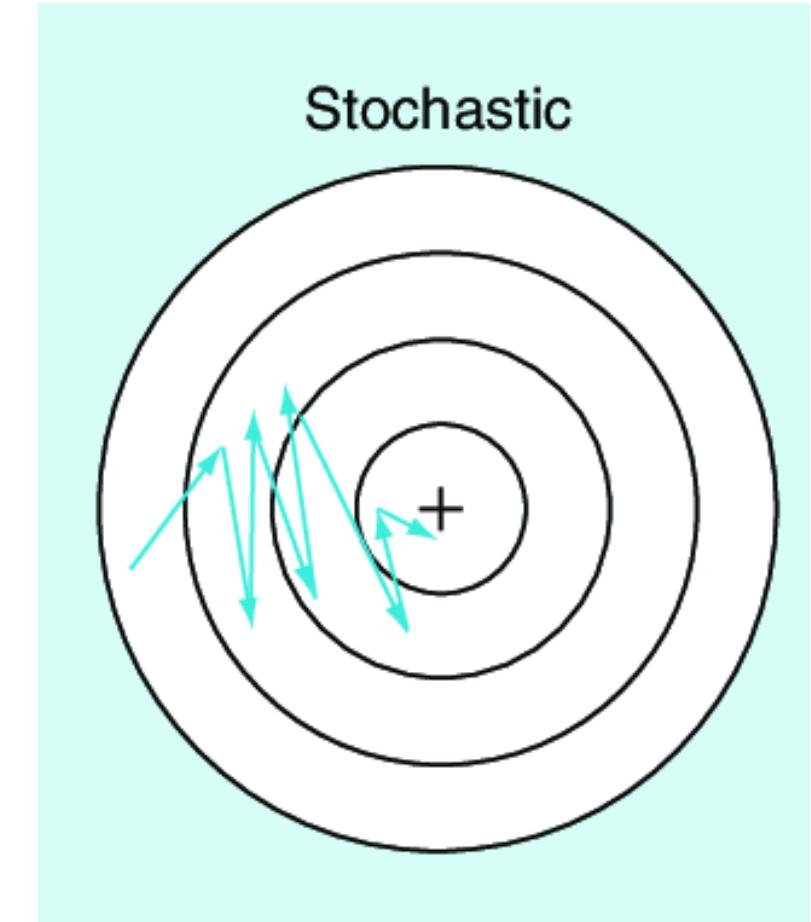


# Stochastic Gradient Descent

Stochastic Gradient Descent is very **noisy** and erratic.

We can try and improve the situation by using a dynamic learning rate, which decreases with iterations.

This is done in practice, but we can also try and find a compromise between gradient descent and stochastic gradient descent.





# Minibatch Gradient Descent

Instead of using all the samples or using only one, **we use  $b$  samples**, where  $b$  is chosen by the designer.

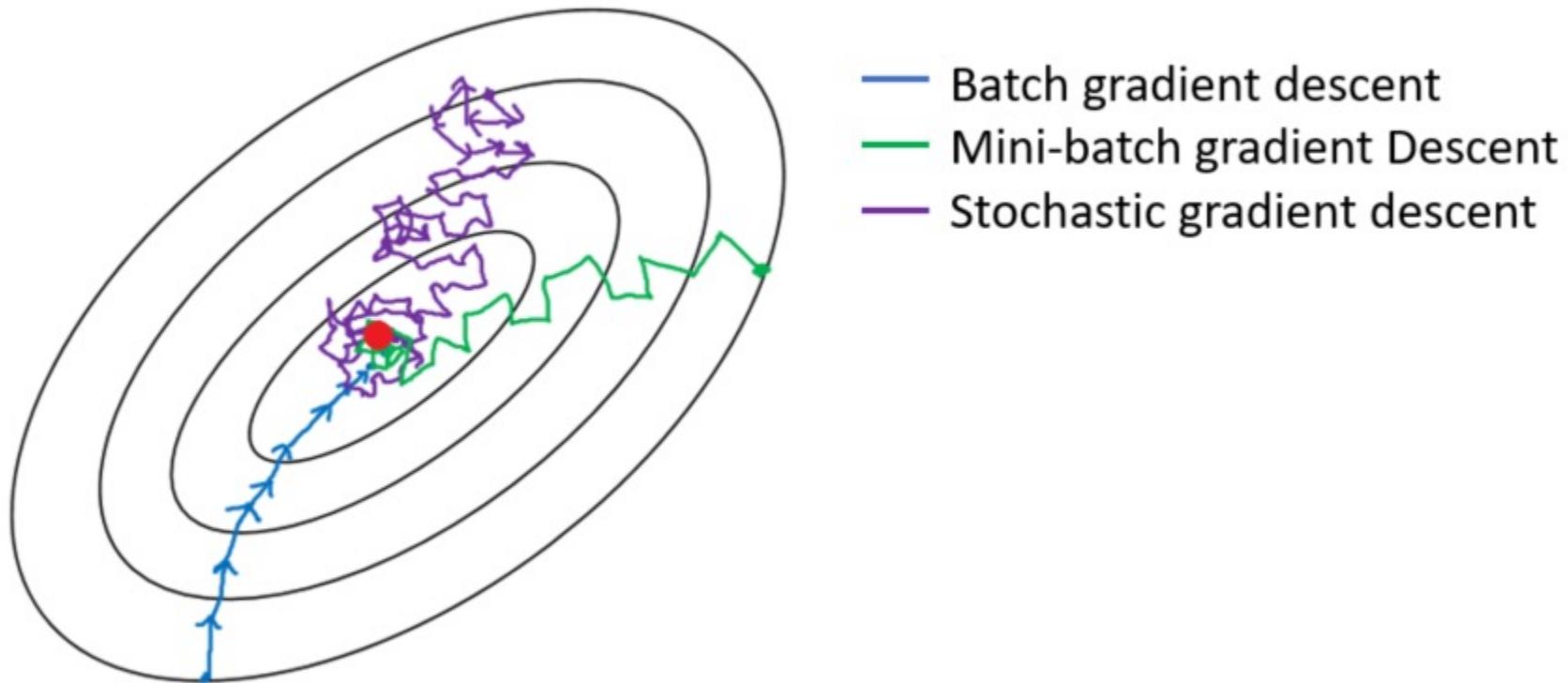
At each step, we compute the gradient:

$$\nabla \left( \frac{1}{b} \sum_{i=1}^b L_i(\theta) \right)$$

This way, we get a much smoother progress, and we can fully exploit hardware caching.



# Minibatch Gradient Descent





# Minibatch Gradient Descent

Instead of using all the samples or using only one, **we use  $b$  samples**, where  $b$  is chosen by the designer.

At each step, we compute the gradient:

$$\nabla \left( \frac{1}{b} \sum_{i=1}^b L_i(\theta) \right)$$

This way, we get a much smoother progress, and we can fully exploit hardware caching.



# Practical Gradient Descent

Nowadays, **nobody uses simple gradient descent anymore.**

More effective algorithms were developed and many tricks to improve their speed of convergence are known.

See [https://d2l.ai/chapter\\_optimization/index.html](https://d2l.ai/chapter_optimization/index.html) for a deeper discussion.

Moreover, do you need to **ever compute gradients?**

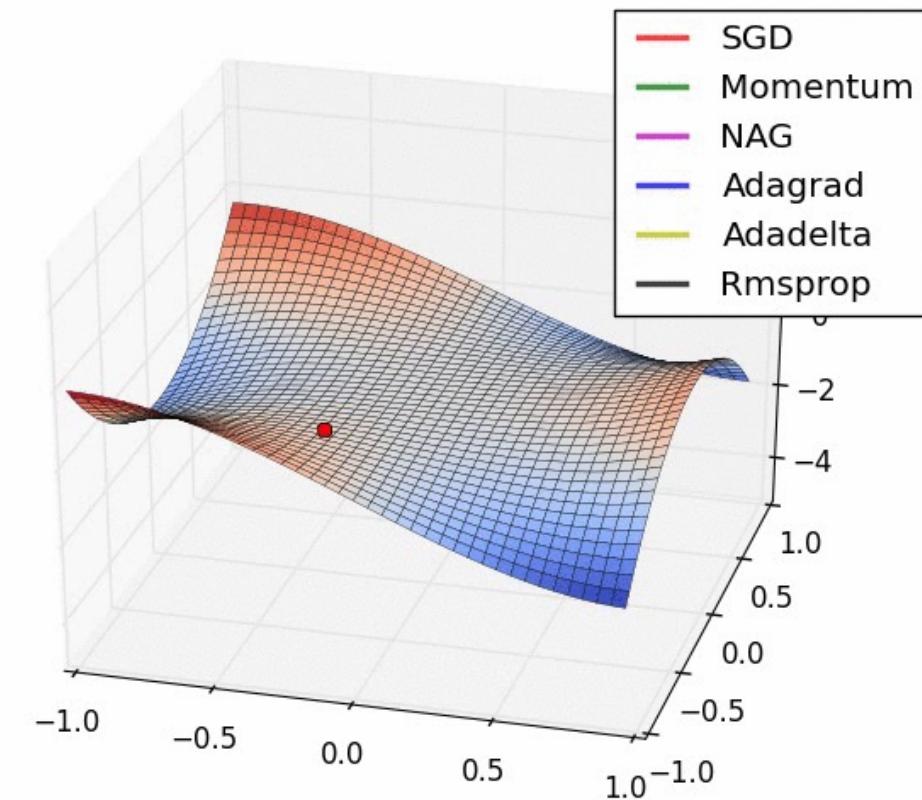
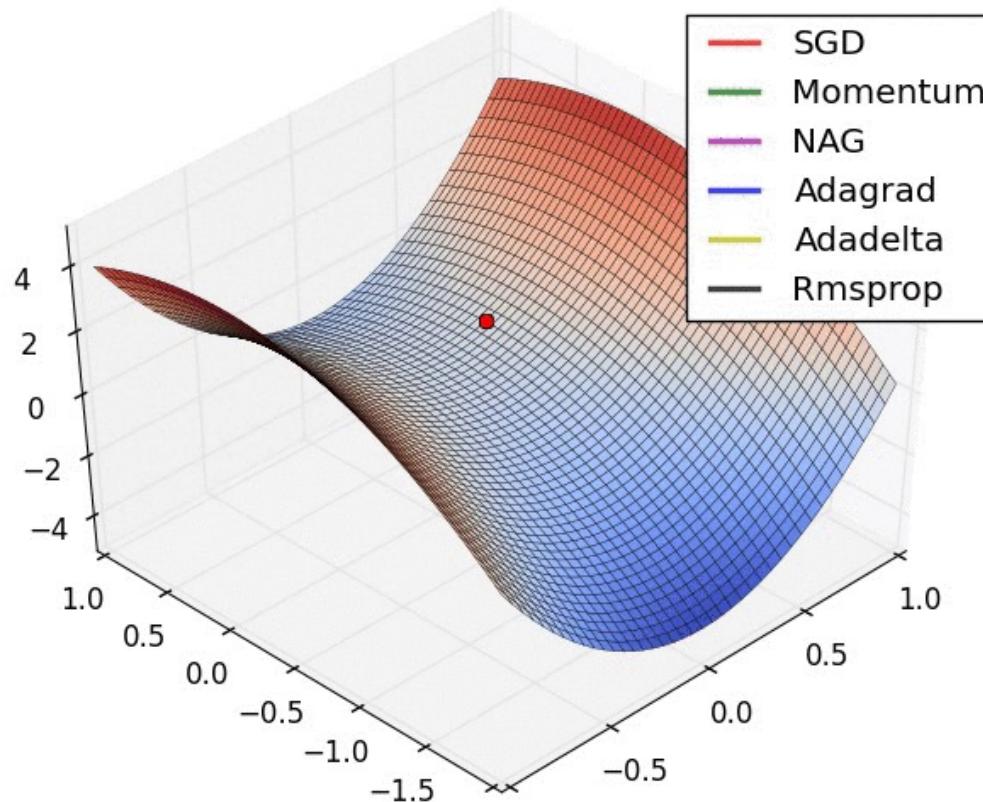
Hell no!

Use the **automatic differentiation** tools implemented by many libraries.

In Python: TensorFlow, PyTorch, JAX.



# Practical Gradient Descent





# Automatic differentiation

How can Deep Learning frameworks compute gradients **automatically**?

The derivatives of the basic building blocks of networks are **known**. So, the only question is how to link them together.

Enter the **chain rule**. Given a composite function  $f(g(x))$ , with  $f: R \rightarrow R$  and  $g: R \rightarrow R$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}$$

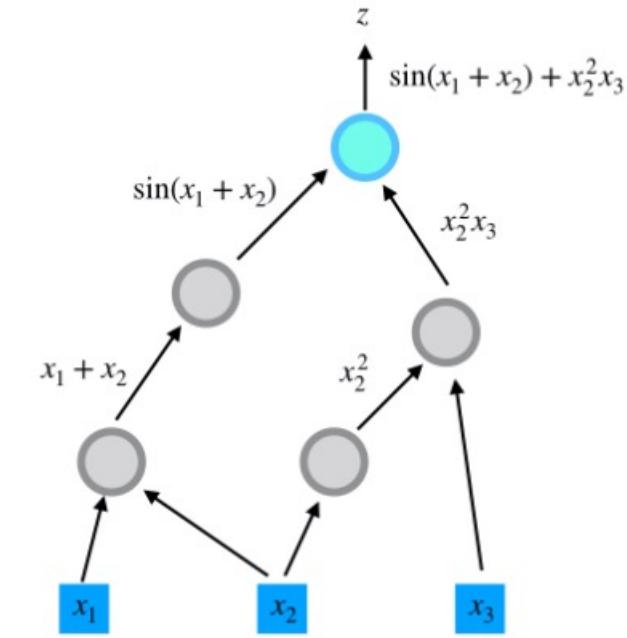
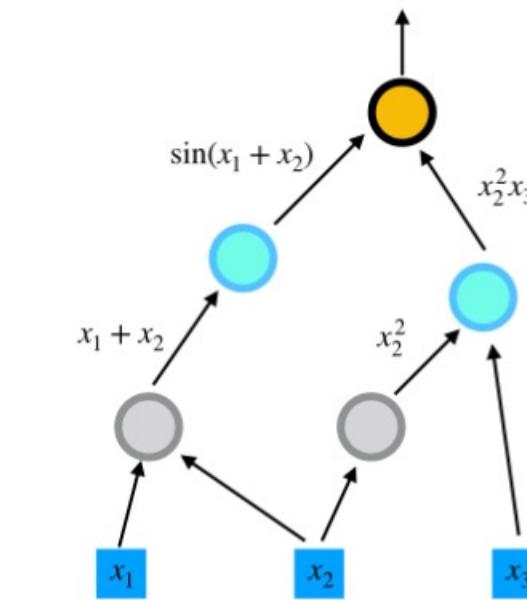
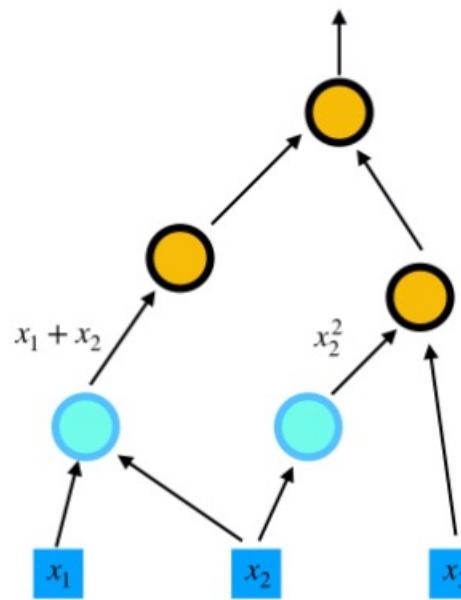
Similar rules apply in higher dimensions, with vector, matrices, ...

So, the gradient of every function can be **built** from the gradients of its basic blocs.



# Automatic differentiation

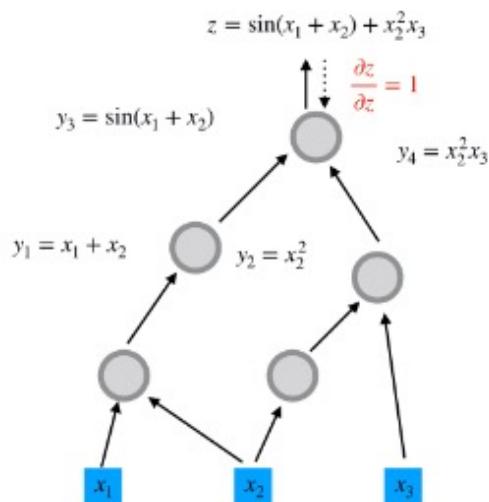
$$z = \sin(x_1 + x_2) + x_2^2 x_3$$



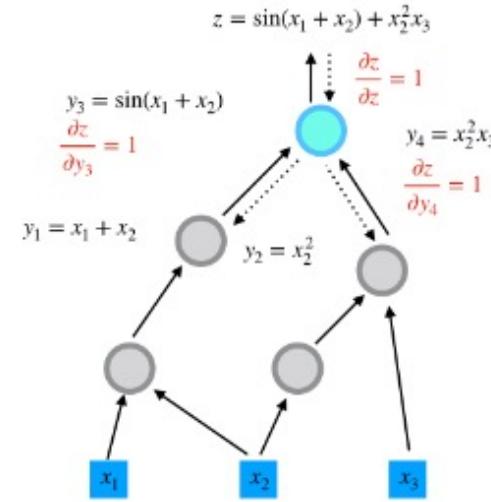


# Automatic differentiation

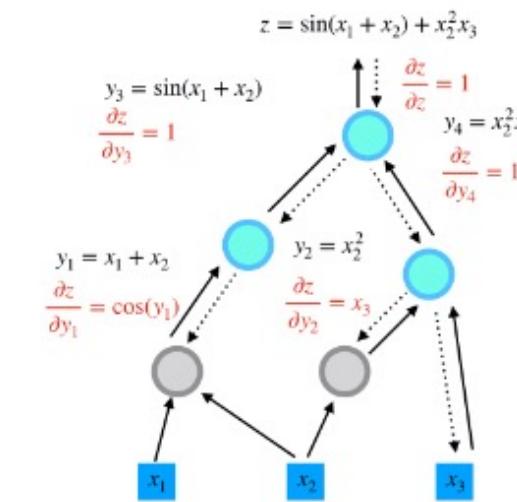
Step 1



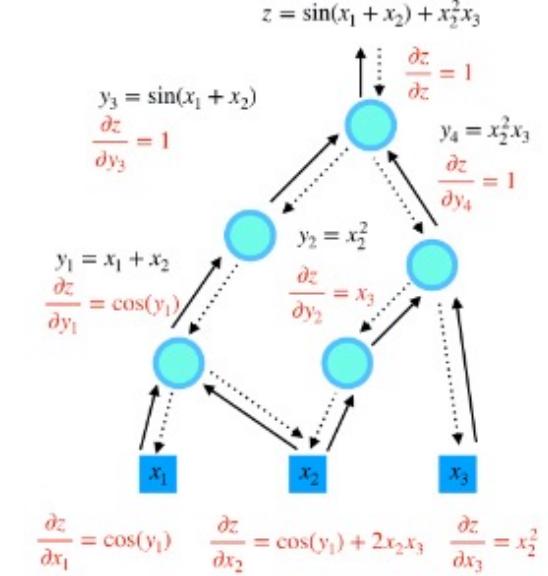
Step 2



Step 3



Step 4





# Getting Practical



# How you Do It

If you're feeling a bit of a fatigue in thinking about the architectural decisions, you'll be pleased to know that in 90% or more of applications you should not have to worry about these.

I like to summarize this point as "don't be a hero".

Instead of rolling your own architecture for a problem, you should look at whatever architecture currently works best on your problem, **download a pretrained model** and, at most, finetune it on your data.

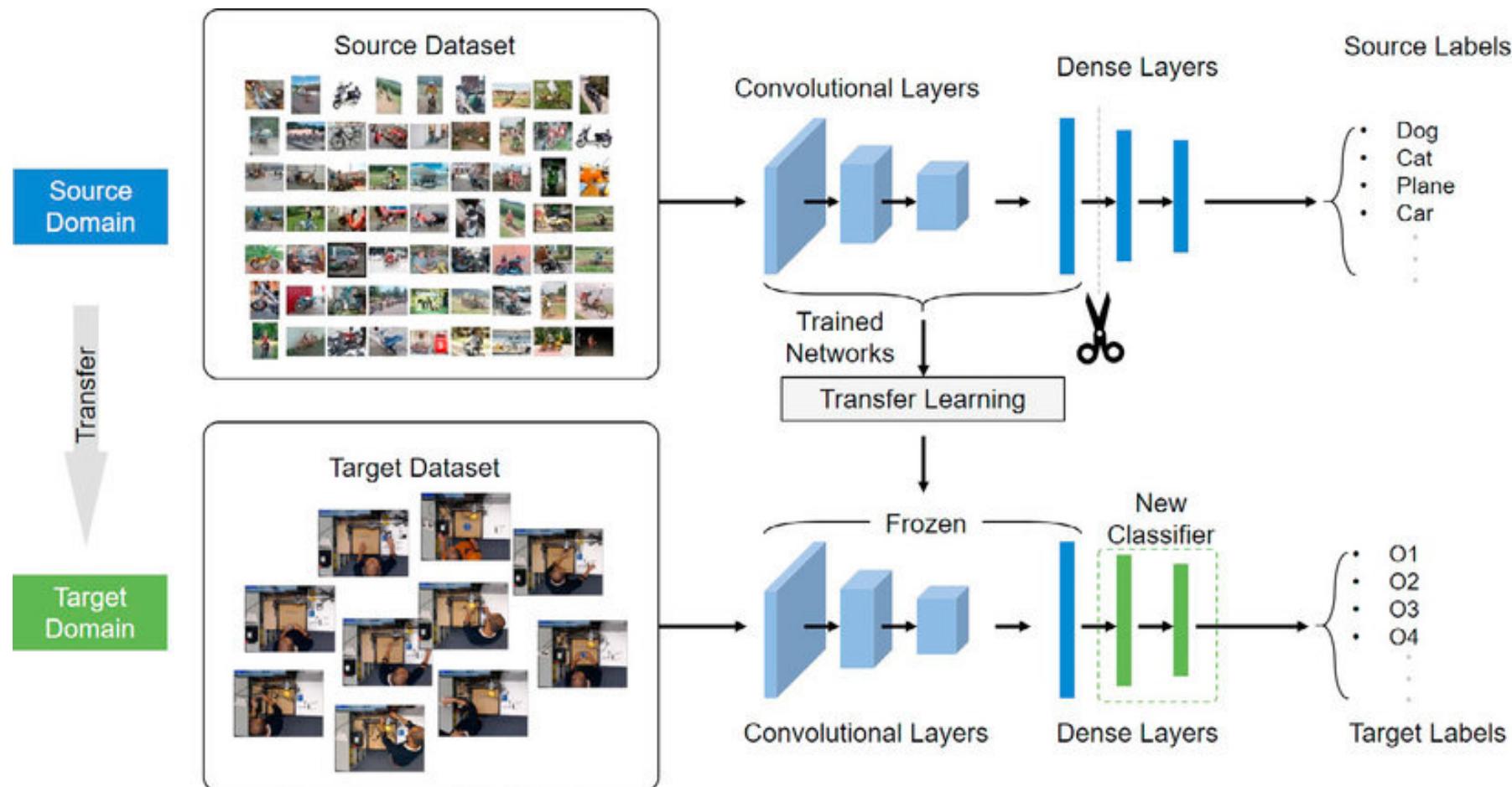
With **GenAI**, most of the times you do not even need to fine-tune your models.

You should rarely ever have to train or design a deep learning model from scratch.

From <https://cs231n.github.io/convolutional-networks/>



# Transfer Learning





The End