Practical AI

Emanuele Fabbiani

What is Deep Learning?

nature

Explore content ▾ About the journal ▾ Publish with us ▾

nature › articles › article

Article | Open Access | Published: 15 July 2021

Highly accurate protein structure p… AlphaFold

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

Nature **596**, 583–589 (2021) | Cite this article

438k Accesses | **279** Citations | **2796** Altmetric | Metrics

Abstract

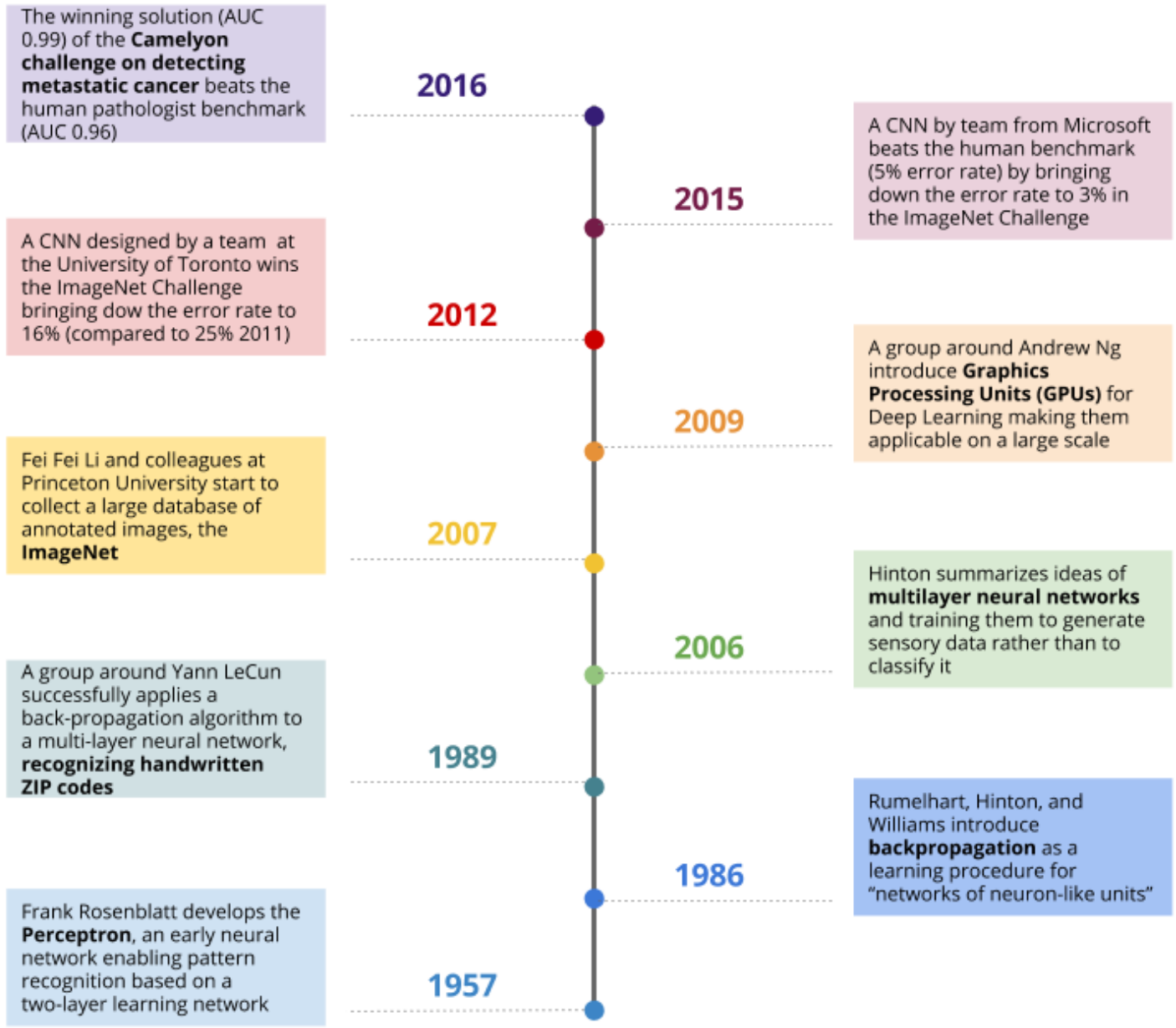
Proteins are essential to life, and understanding their structu… understanding of their function. Through an enormous expe… structures of around 100,000 unique proteins have been det… small fraction of the billions of known protein sequences[6,7]. … bottlenecked by the months to years of painstaking effort re… protein structure. Accurate computational approaches are n… to enable large-scale structural bioinformatics. Predicting th… that a protein will adopt based solely on its amino acid seque… component of the 'protein folding problem'[8]—has been an in…

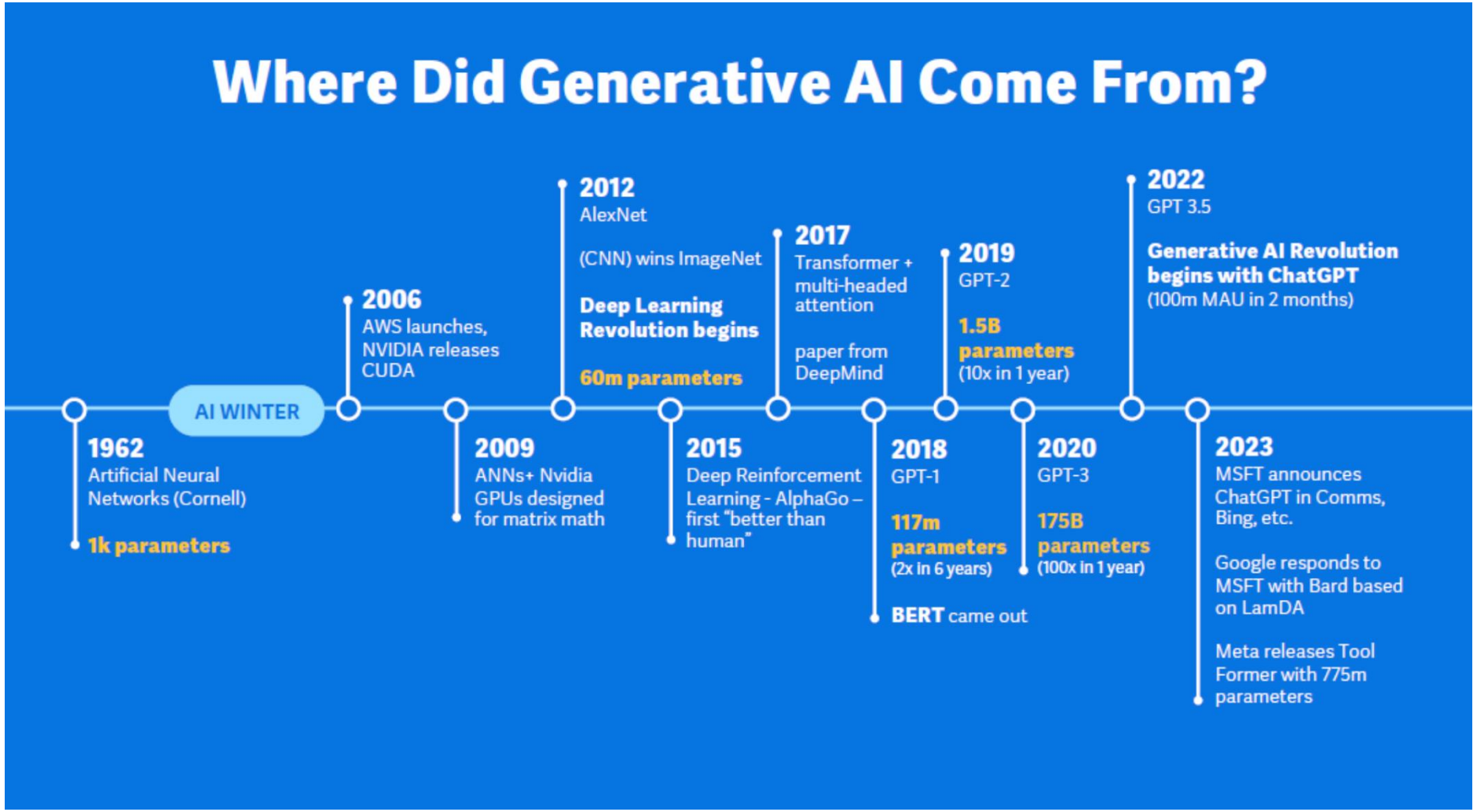
GPT-4 Technical Report

OpenAI*

Abstract

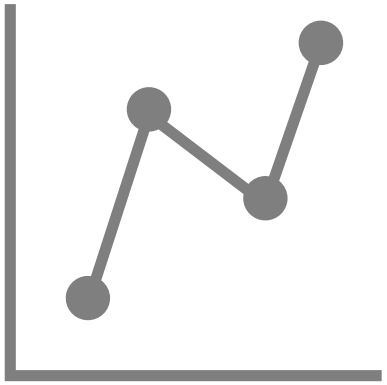
…development of GPT-4, a large-scale, multimodal model which can …and text inputs and produce text outputs. While less capable than …ny real-world scenarios, GPT-4 exhibits human-level performance …fessional and academic benchmarks, including passing a simulated … a score around the top 10% of test takers. GPT-4 is a Transformer-…re-trained to predict the next token in a document. The post-training …cess results in improved performance on measures of factuality and …desired behavior. A core component of this project was developing … and optimization methods that behave predictably across a wide …s. This allowed us to accurately predict some aspects of GPT-4's …ased on models trained with no more than 1/1,000th the compute of …


The winning solution (AUC 0.99) of the **Camelyon challenge on detecting metastatic cancer** beats the human pathologist benchmark (AUC 0.96)
2016
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
2015
A CNN designed by a team at the University of Toronto wins the ImageNet Challenge bringing dow the error rate to 16% (compared to 25% 2011)
2012
A group around Andrew Ng introduce **Graphics Processing Units (GPUs)** for Deep Learning making them applicable on a large scale
2009
Fei Fei Li and colleagues at Princeton University start to collect a large database of annotated images, the **ImageNet**
2007
Hinton summarizes ideas of **multilayer neural networks** and training them to generate sensory data rather than to classify it
2006
A group around Yann LeCun successfully applies a back-propagation algorithm to a multi-layer neural network, **recognizing handwritten ZIP codes**
1989
Rumelhart, Hinton, and Williams introduce **backpropagation** as a learning procedure for "networks of neuron-like units"
1986
Frank Rosenblatt develops the **Perceptron**, an early neural network enabling pattern recognition based on a two-layer learning network
1957

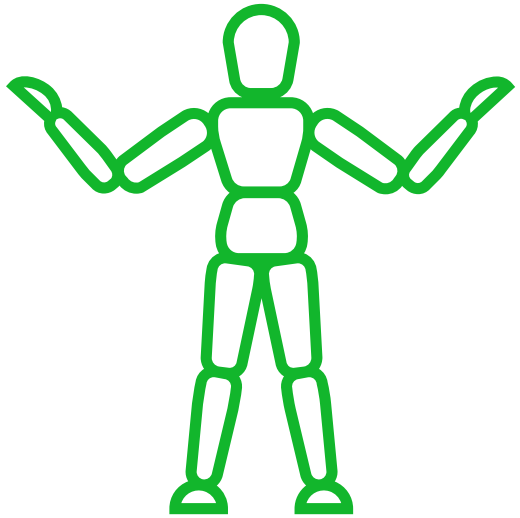

Where Did Generative AI Come From?
1962
Artificial Neural Networks (Cornell)
1k parameters
AI WINTER
2006
AWS launches, NVIDIA releases CUDA
2009
ANNs+ Nvidia GPUs designed for matrix math
2012
AlexNet
(CNN) wins ImageNet
Deep Learning Revolution begins
60m parameters
2015
Deep Reinforcement Learning - AlphaGo – first "better than human"
2017
Transformer + multi-headed attention
paper from DeepMind
2018
GPT-1
117m parameters
(2x in 6 years)
BERT came out
2019
GPT-2
1.5B parameters
(10x in 1 year)
2020
GPT-3
175B parameters
(100x in 1 year)
2022
GPT 3.5
Generative AI Revolution begins with ChatGPT
(100m MAU in 2 months)
2023
MSFT announces ChatGPT in Comms, Bing, etc.
Google responds to MSFT with Bard based on LamDA
Meta releases Tool Former with 775m parameters

Contents

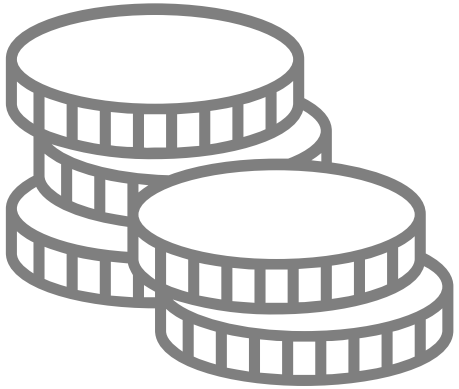
Data



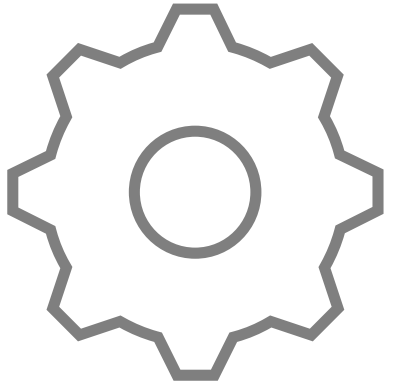
Model



Loss



Learning algorithm



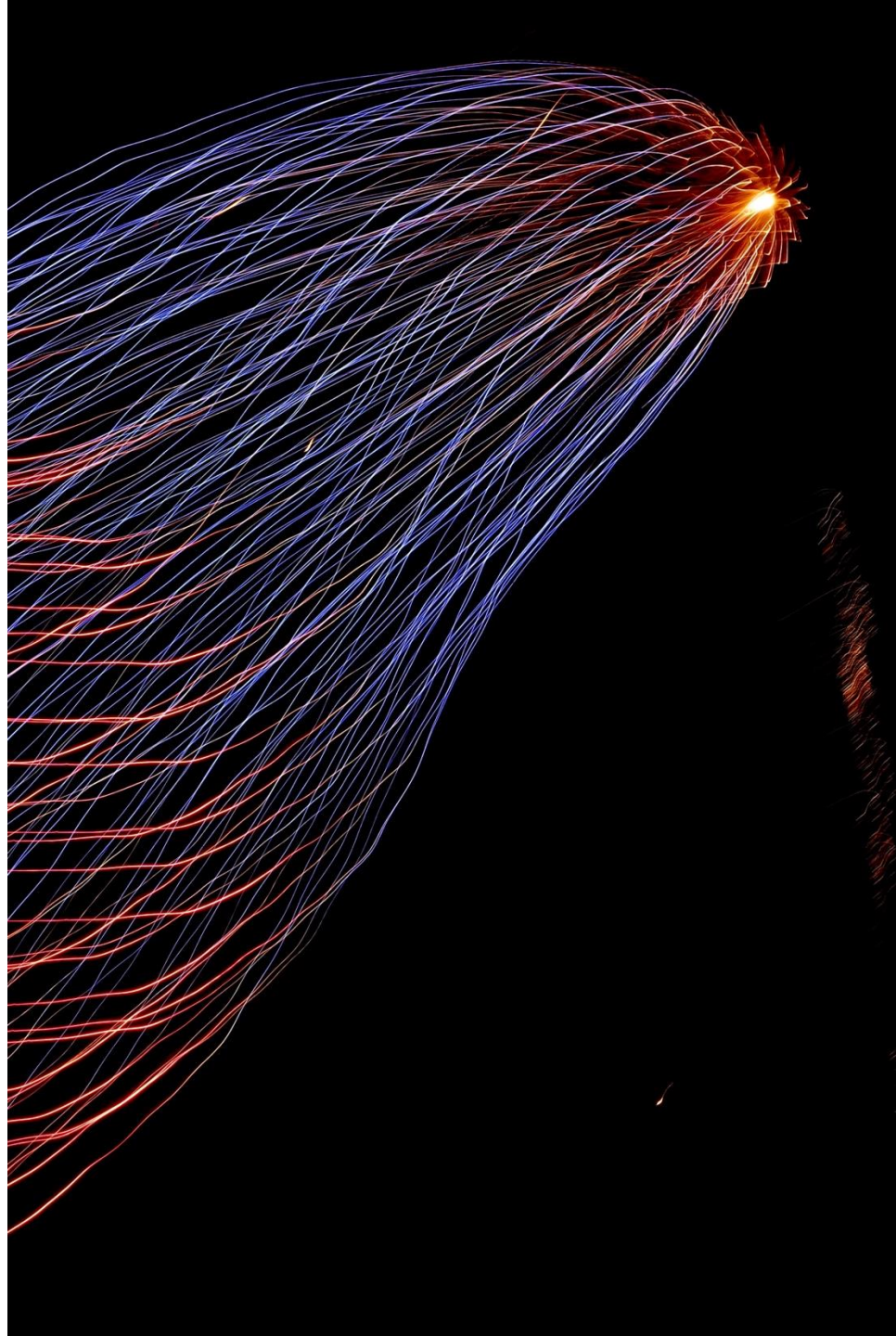
Data

We have a bunch of *numeric* data.

Each sample is made of:

- input real vector $\boldsymbol{x}_i$
- 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

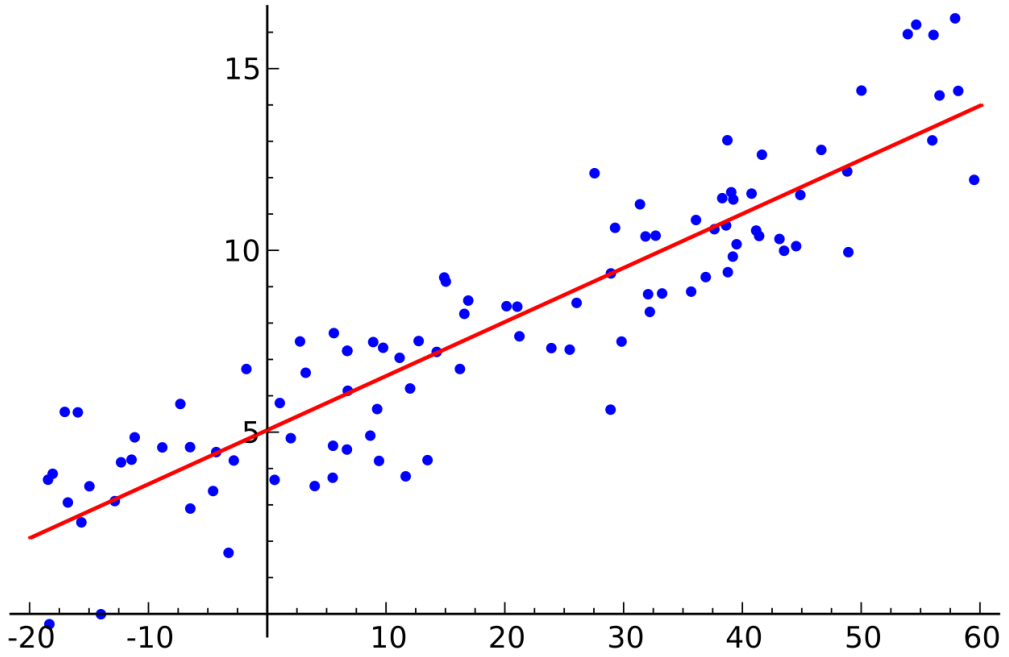
A completely legitimate model:

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

where:

$$w, x \in R^p$$
$$y, b \in R$$

Yet, this model has a problem…



Model

A completely legitimate model:

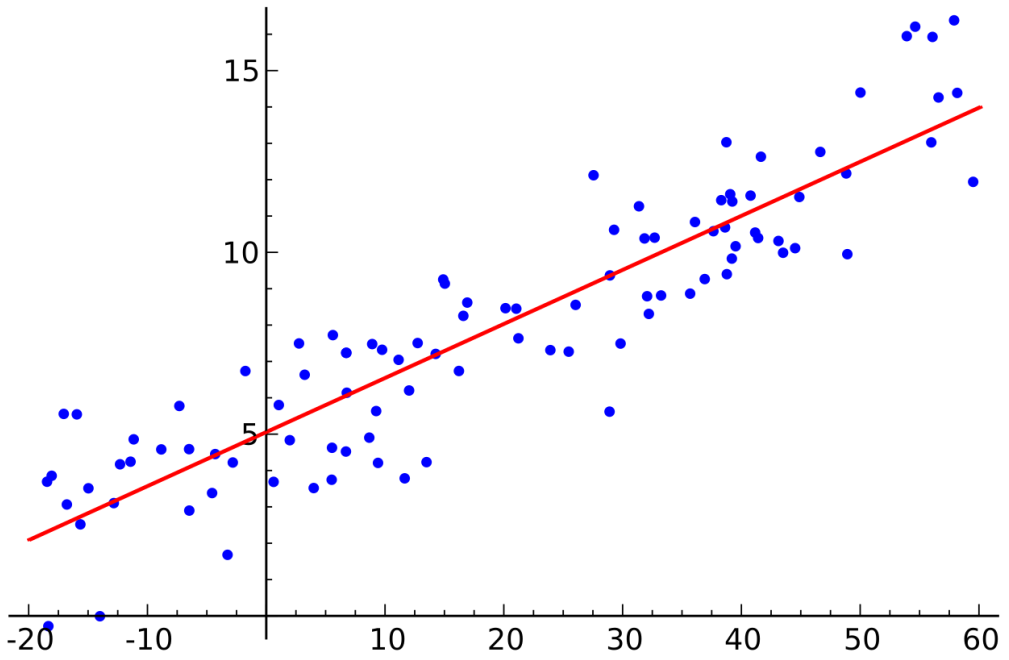
$$y = \boldsymbol{w}^T \boldsymbol{x} + b$$

where:

$$w, x \in R^p$$
$$y, b \in R$$

Yet, this model has a problem.

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



* unless we explicitly make up some elements in x.

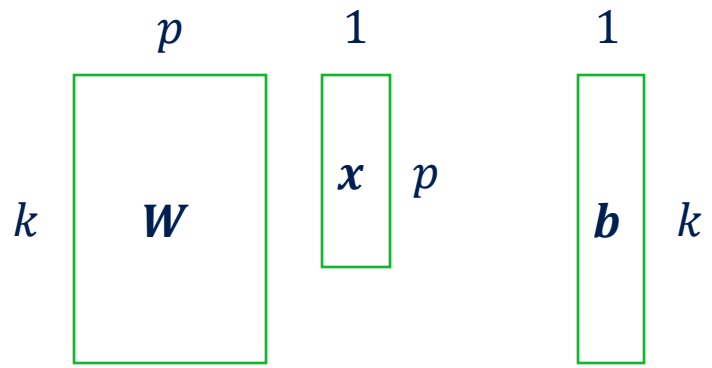
My First Network

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

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

where:

$$\boldsymbol{W} \in R^{k \times p}$$
$$\boldsymbol{w}, \boldsymbol{b} \in R^k$$
$$\boldsymbol{x} \in R^p$$
$$y, b \in R$$

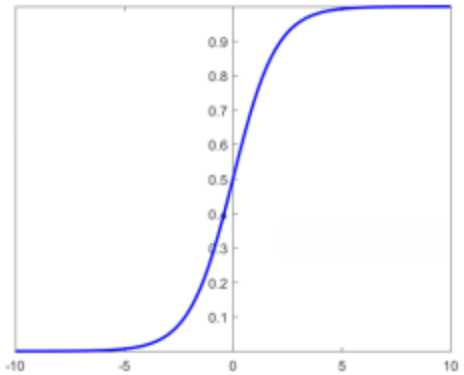


My First Network

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

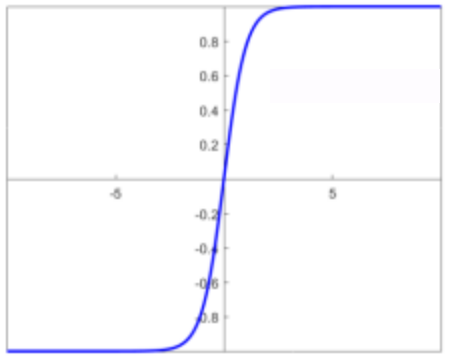
where g is a non-linear function, applied element-wise.

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



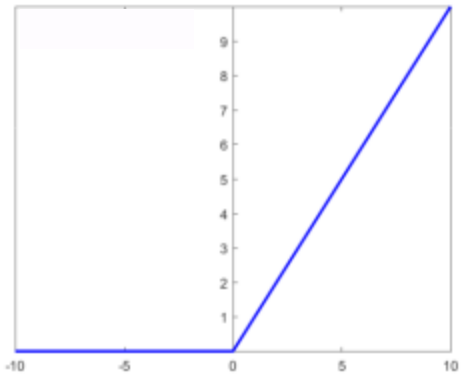
Sigmoid

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



Hyperbolic Tangent

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

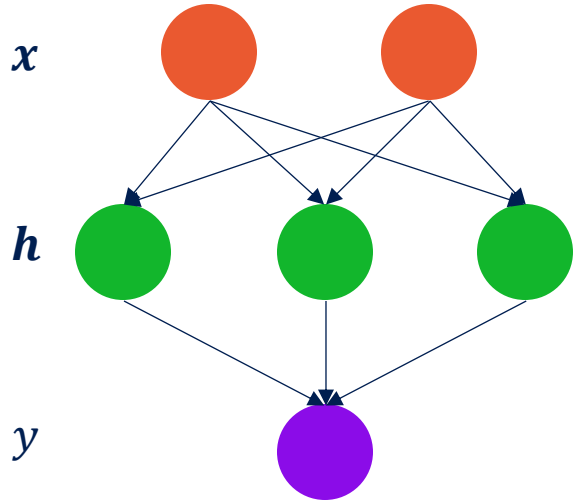


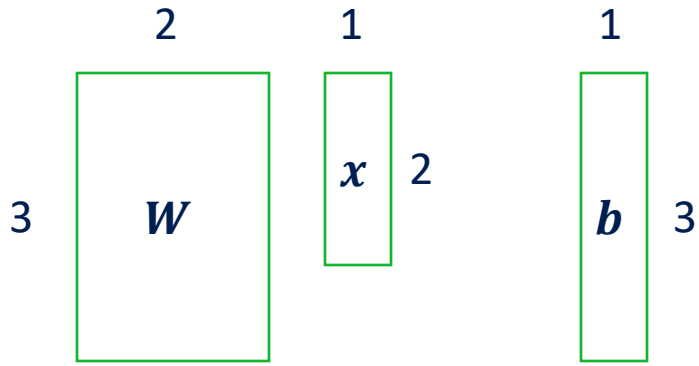
ReLU

Shallow Networks

We build a feed-forward shallow network.

$$y = \boldsymbol{w}^T \underbrace{g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})}_{\boldsymbol{h}} + b$$





Shallow Networks

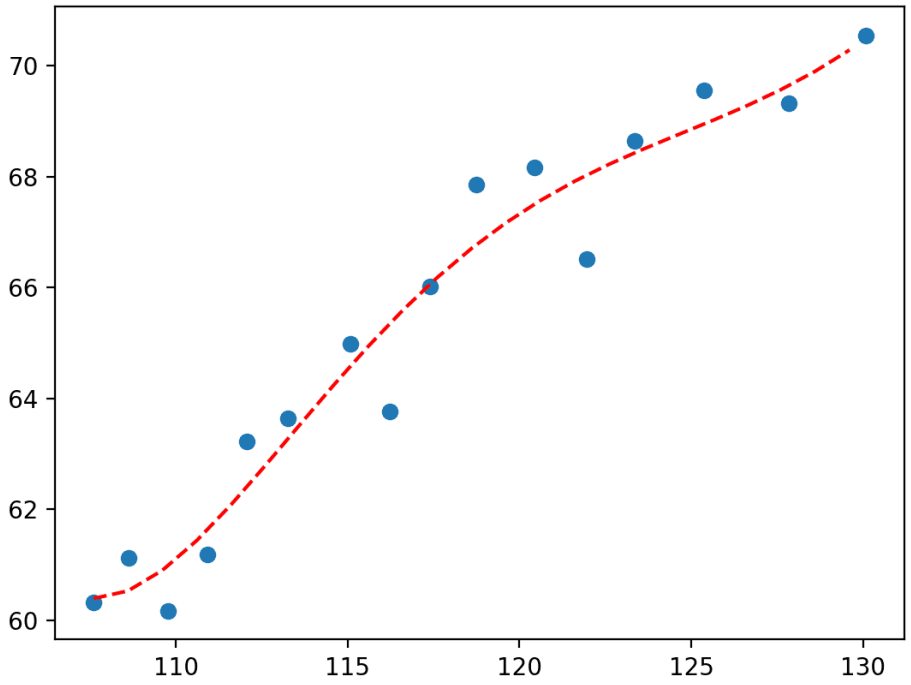
We build a feed-forward shallow network.

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

What did we gain?

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

Which functions?

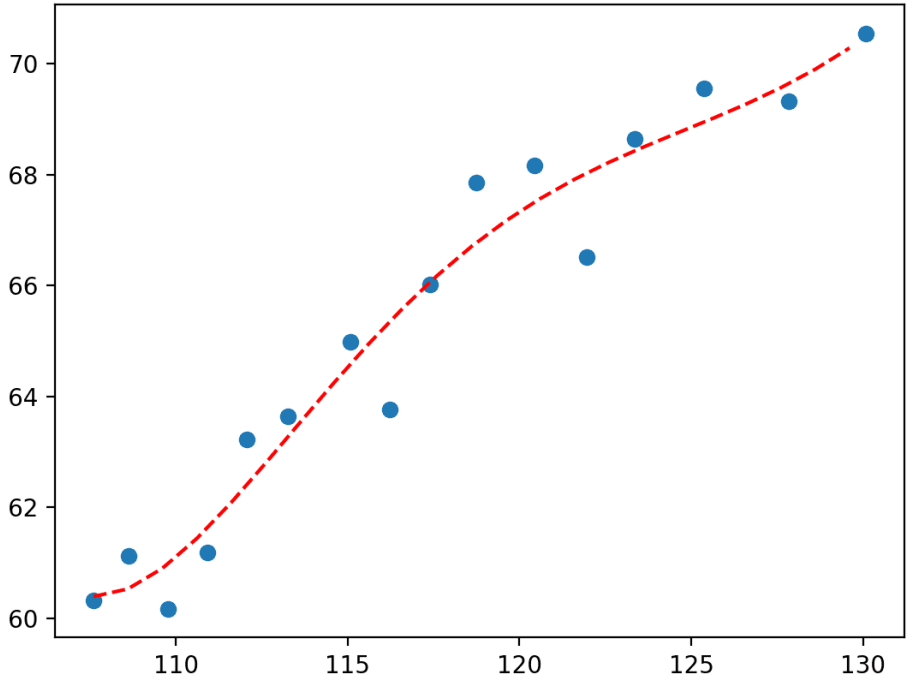



ALL OF THEM.
memegenerator.ne

The Universal Approximation Theorem(s)

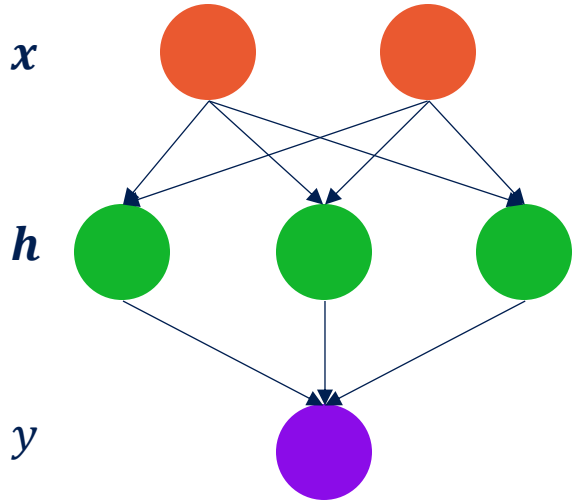
Universal Approximation Theorems are proven mathematical statements showing how networks with certain properties can approximate *to an arbitrary level of precision* **any regular function**.

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.



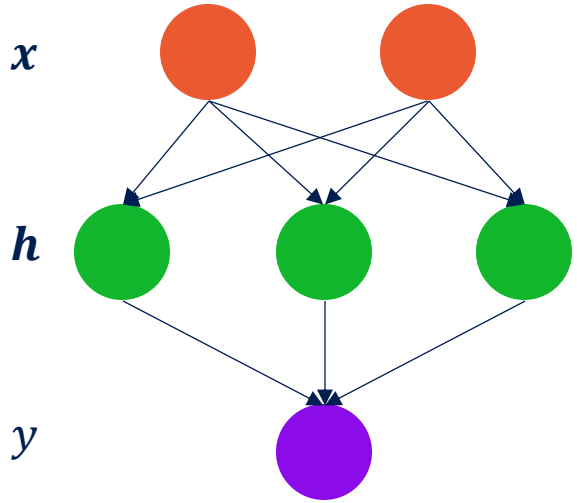
$$y = \underbrace{\boldsymbol{w}^T g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})}_{\boldsymbol{h}} + b$$

Adapted from: https://www.deeplearningbook.org/


REALLY?
ARE YOU KIDDING ME
memegenerator.net

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.

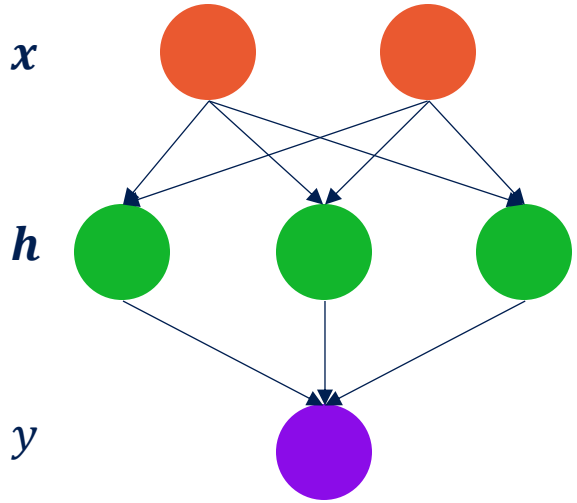


$$y = \underbrace{\boldsymbol{w}^T g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})}_{\boldsymbol{h}} + b$$

Adapted from: https://www.deeplearningbook.org/
See also: https://vision.unipv.it/AI/AIRG/DeepLearningAndTensorFlow2020-Episode1.pdf

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.

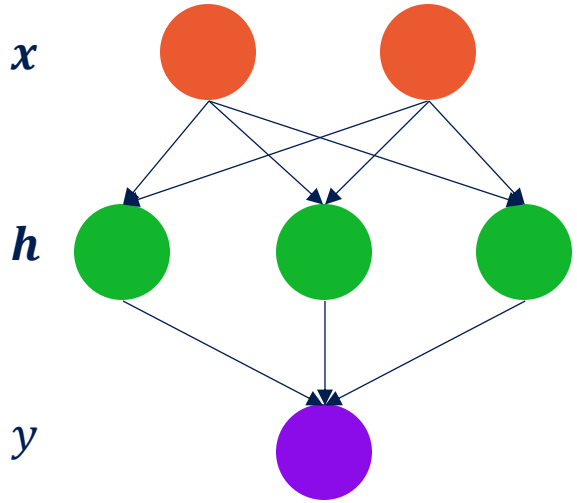


$$y = \underbrace{\boldsymbol{w}^T g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})}_{\boldsymbol{h}} + b$$

Adapted from: https://www.deeplearningbook.org/
See also: https://vision.unipv.it/AI/AIRG/DeepLearningAndTensorFlow2020-Episode1.pdf

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.

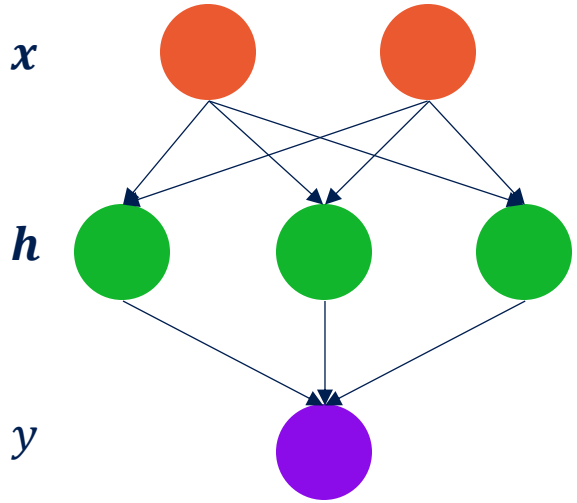


$$y = \underbrace{\boldsymbol{w}^T g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})}_{\boldsymbol{h}} + b$$

Adapted from: https://www.deeplearningbook.org/
See also: https://vision.unipv.it/AI/AIRG/DeepLearningAndTensorFlow2020-Episode1.pdf

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.



$$y = \underbrace{\boldsymbol{w}^T g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})}_{\boldsymbol{h}} + b$$

Adapted from: https://www.deeplearningbook.org/
See also: https://vision.unipv.it/AI/AIRG/DeepLearningAndTensorFlow2020-Episode1.pdf

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.

[1] Barron, A. E. (1993). Universal approximation bounds for superpositions of a sigmoidal function.

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?

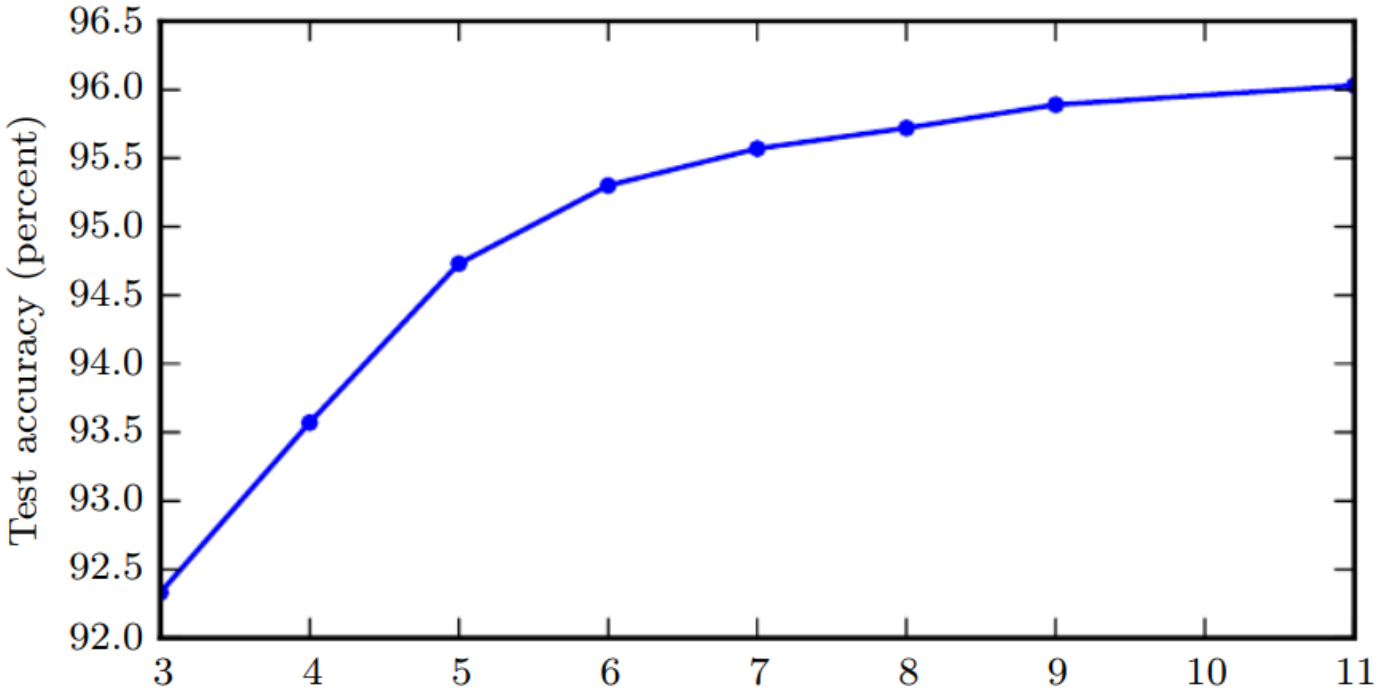


Figure 6.6: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating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

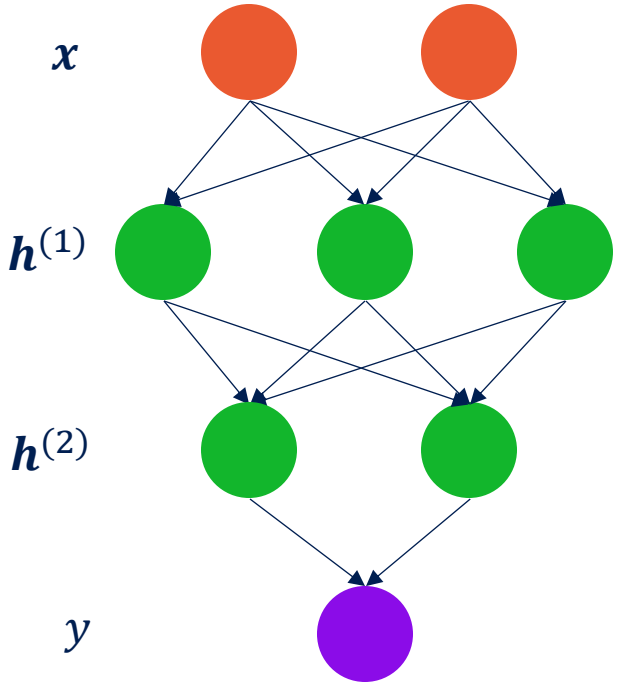
From: https://www.deeplearningbook.org/

My First Deep Network

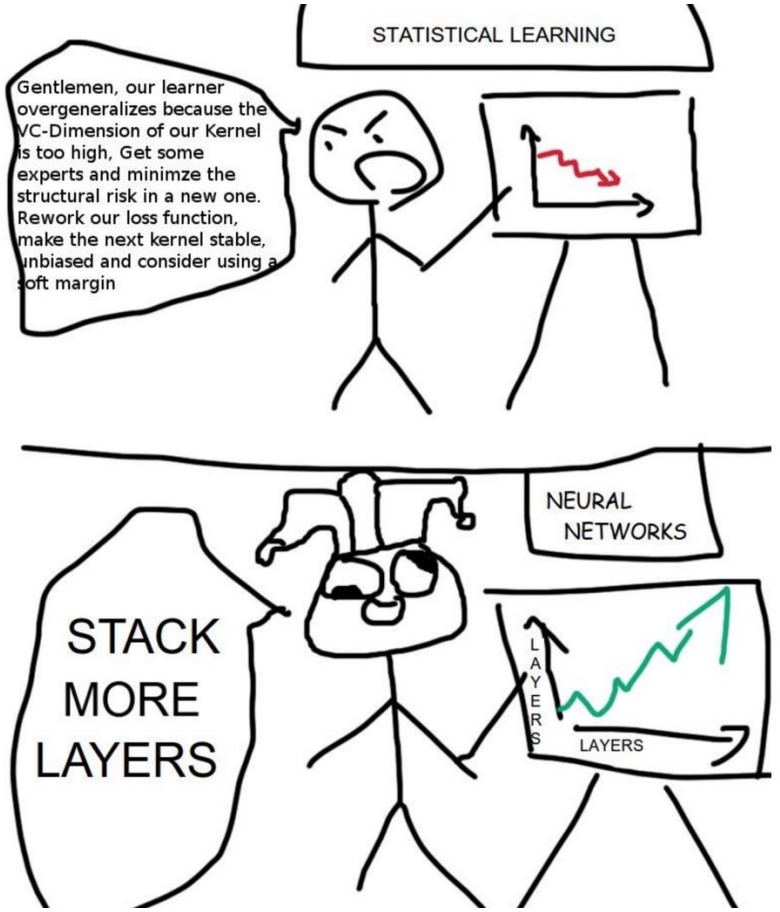
A network with two layers:

$$y = \boldsymbol{w}^T g\Big(\boldsymbol{W}^{(2)} \underbrace{g\big(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\big)}_{\boldsymbol{h}^{(1)}} + \boldsymbol{b}^{(2)}\Big) + b$$

$\boldsymbol{h}^{(2)}$

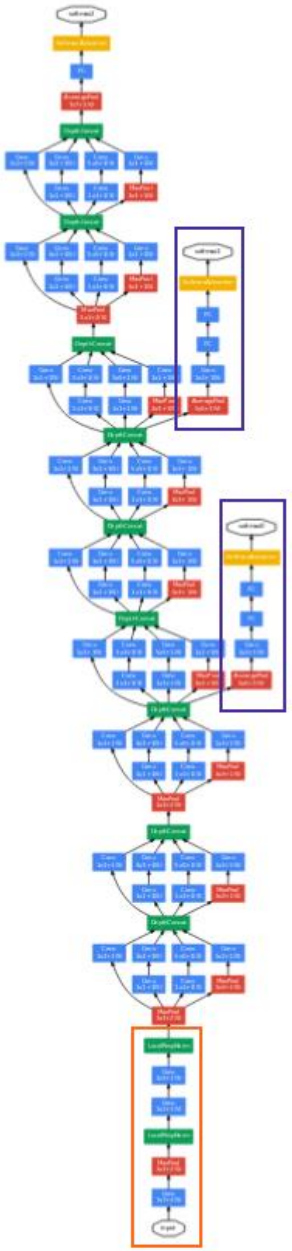


How Deep?

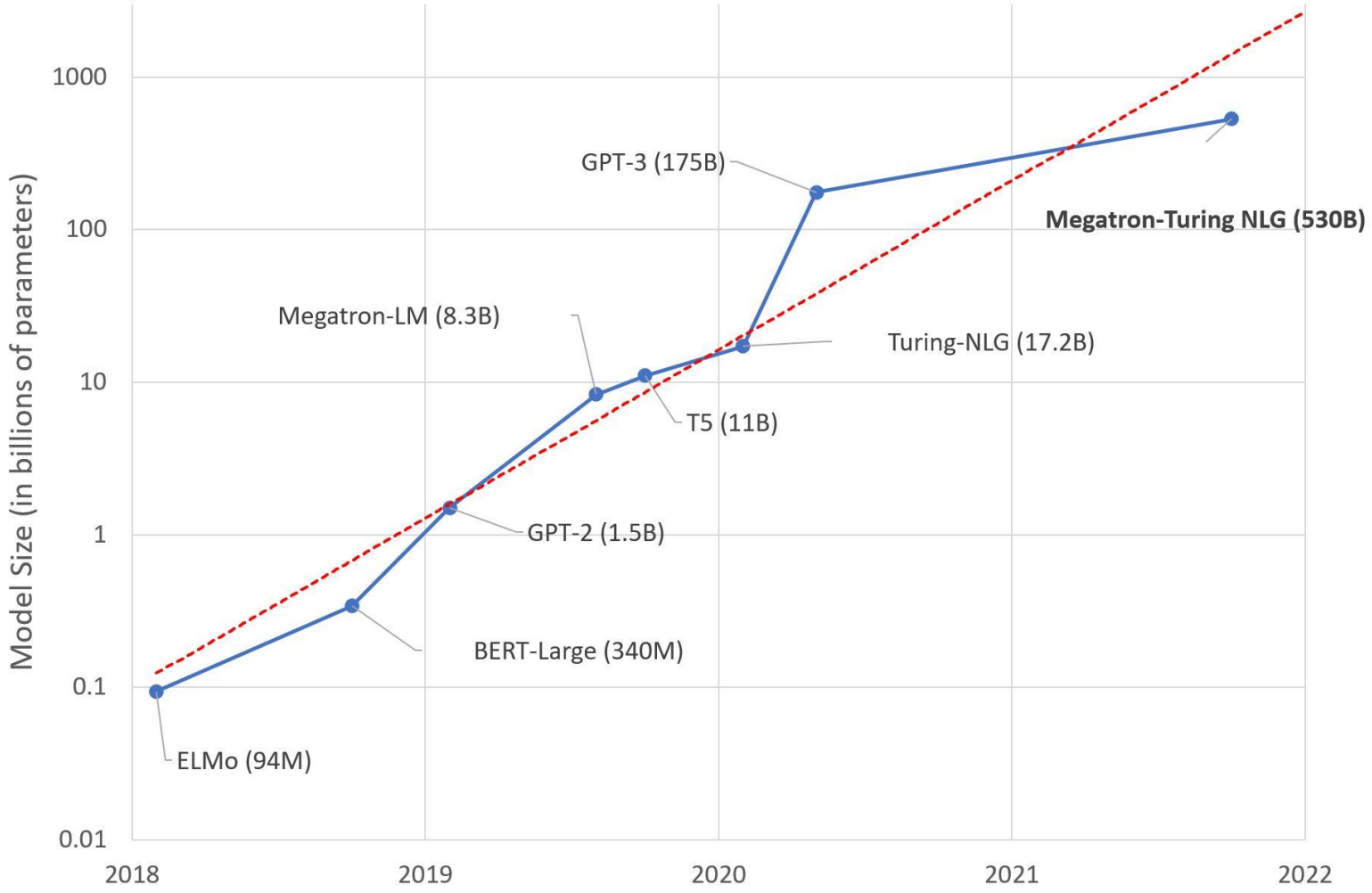


From: https://www.reddit.com/r/ProgrammerHumor/comments/8c1i45/stack_more_layers/

How Deep?

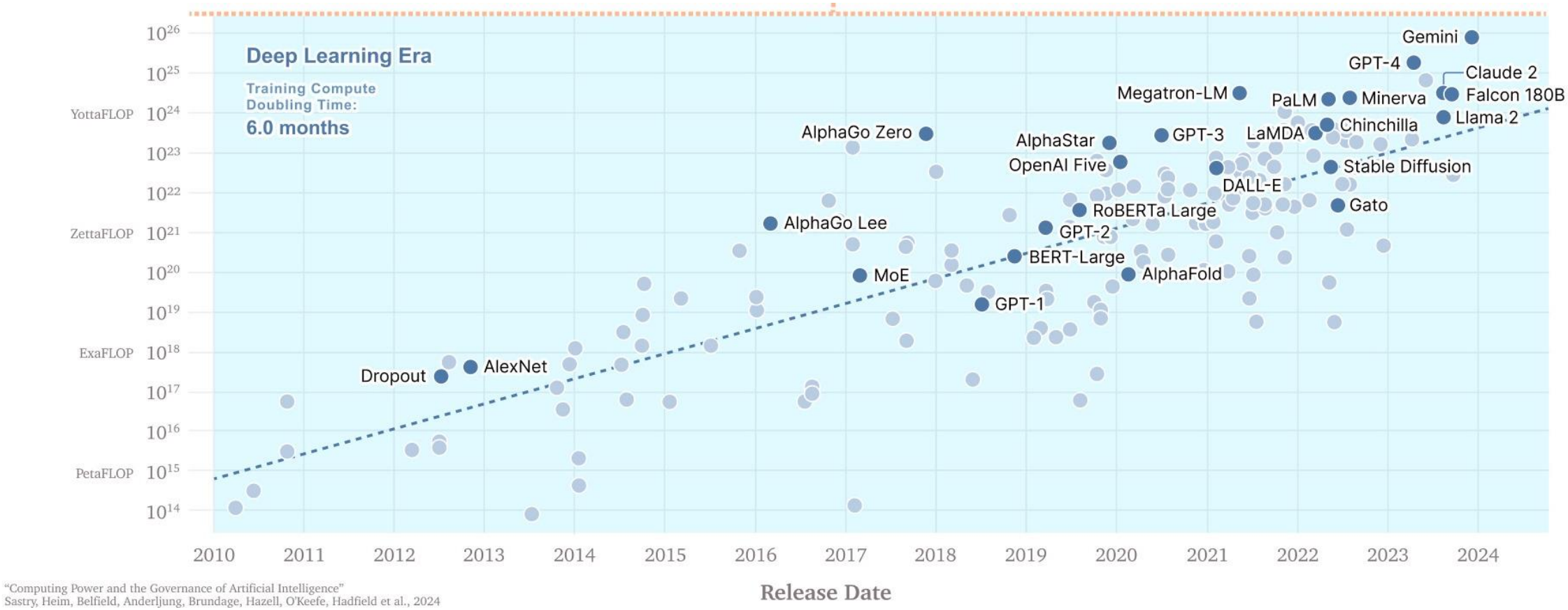


How Deep?



From: https://huggingface.co/blog/large-language-models

Scaling Compute



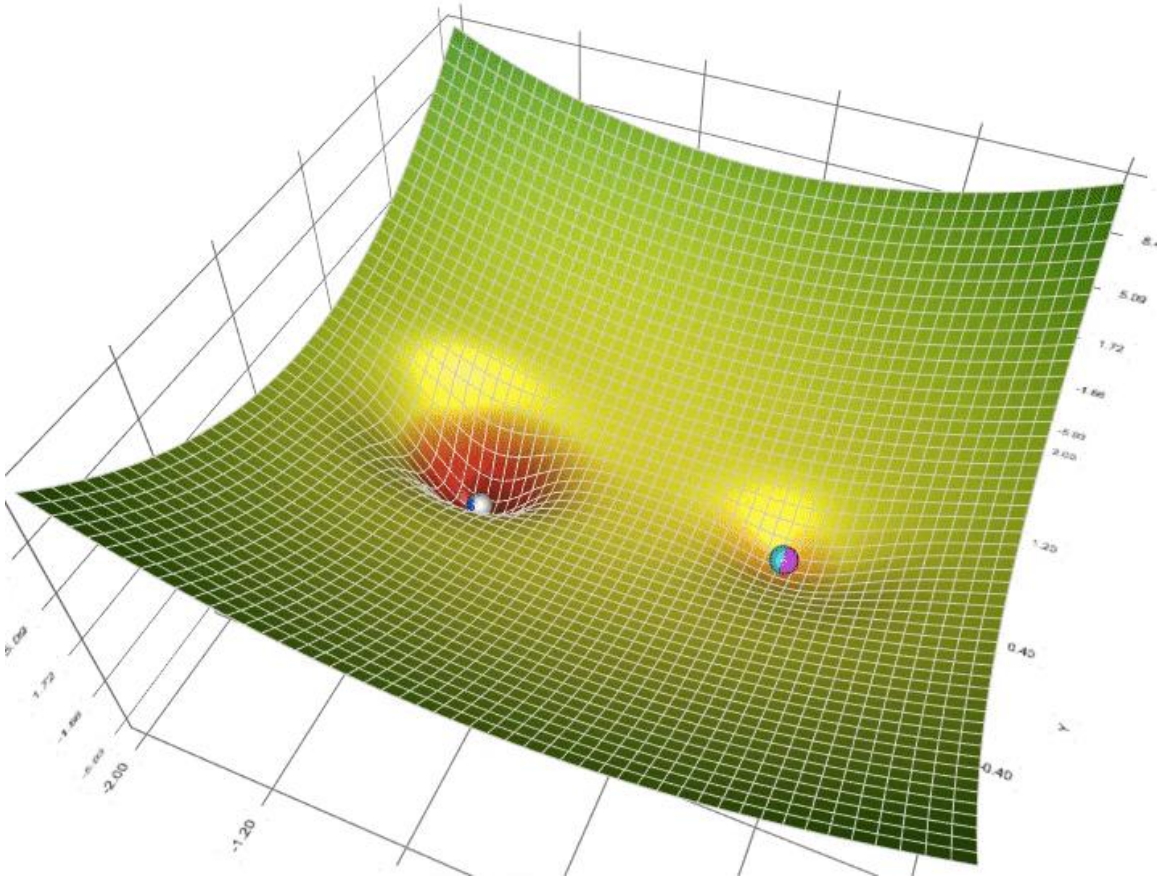
"Computing Power and the Governance of Artificial Intelligence"
Sastry, Heim, Belfield, Anderljung, Brundage, Hazell, O'Keefe, Hadfield et al., 2024

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 $\boldsymbol{x}_i$
- output real scalar y_i

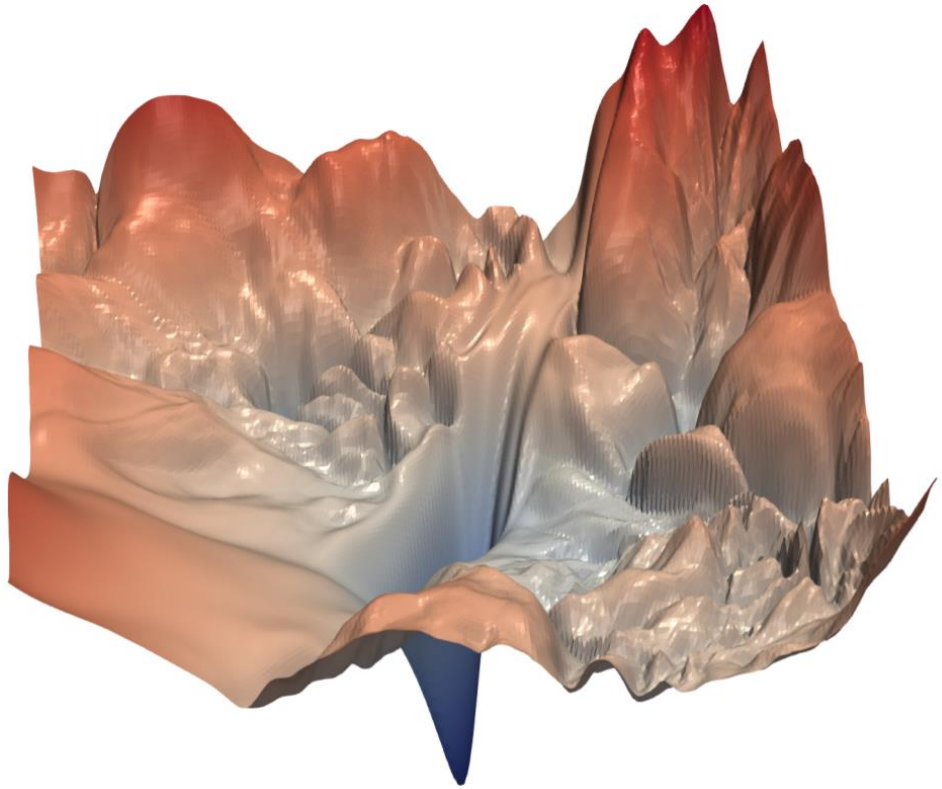
Our model is:

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

Our loss is:

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

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



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

A Hard Problem Statement

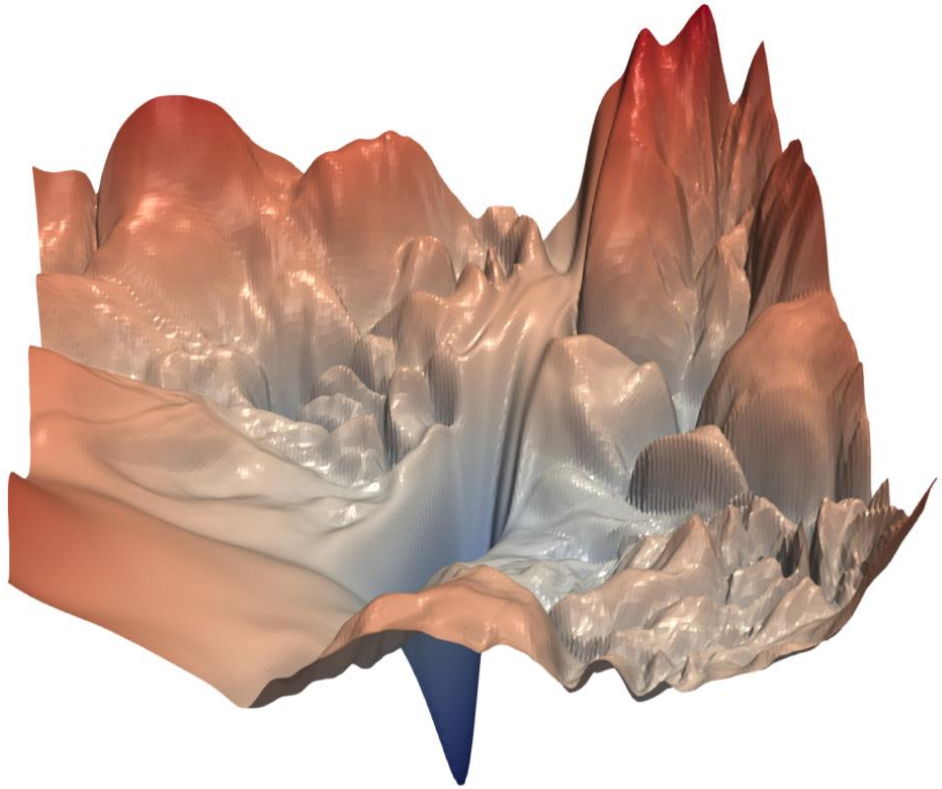
Our loss is:

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

The learning algorithm should find $\boldsymbol{w}, \boldsymbol{W}, \boldsymbol{b}, b$ 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.



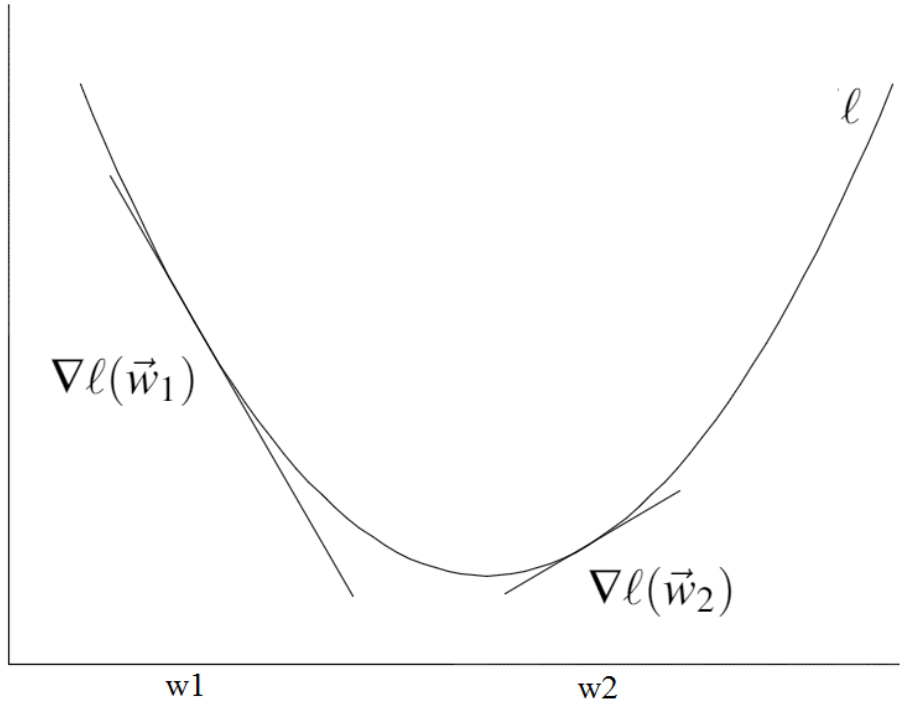
https://arxiv.org/pdf/1712.09913.pdf

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:

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

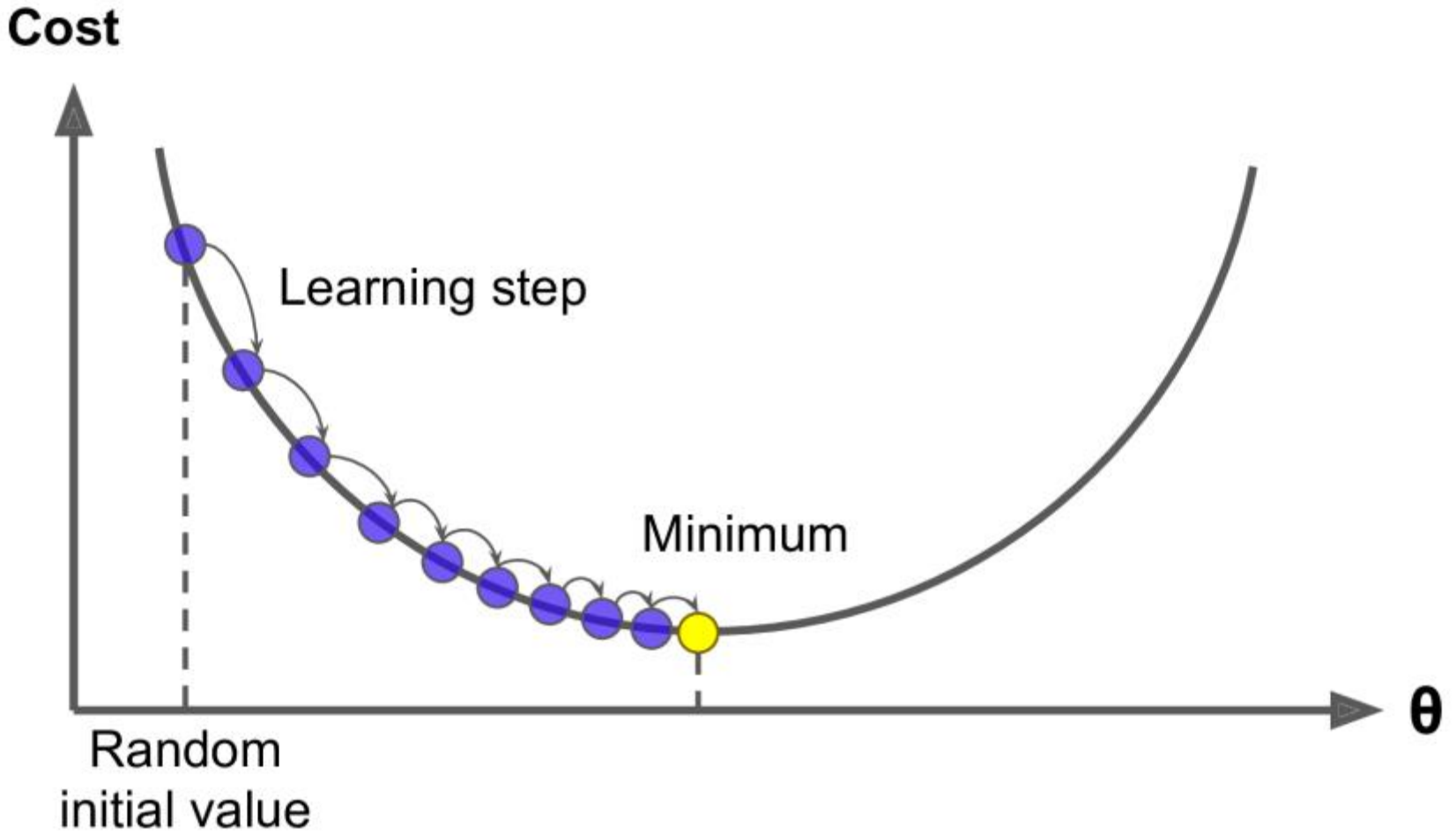
The learning algorithm should find $\boldsymbol{\vartheta} = (\boldsymbol{w}, \boldsymbol{W}, \boldsymbol{b}, b)$ to minimize L.

Gradient descent

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

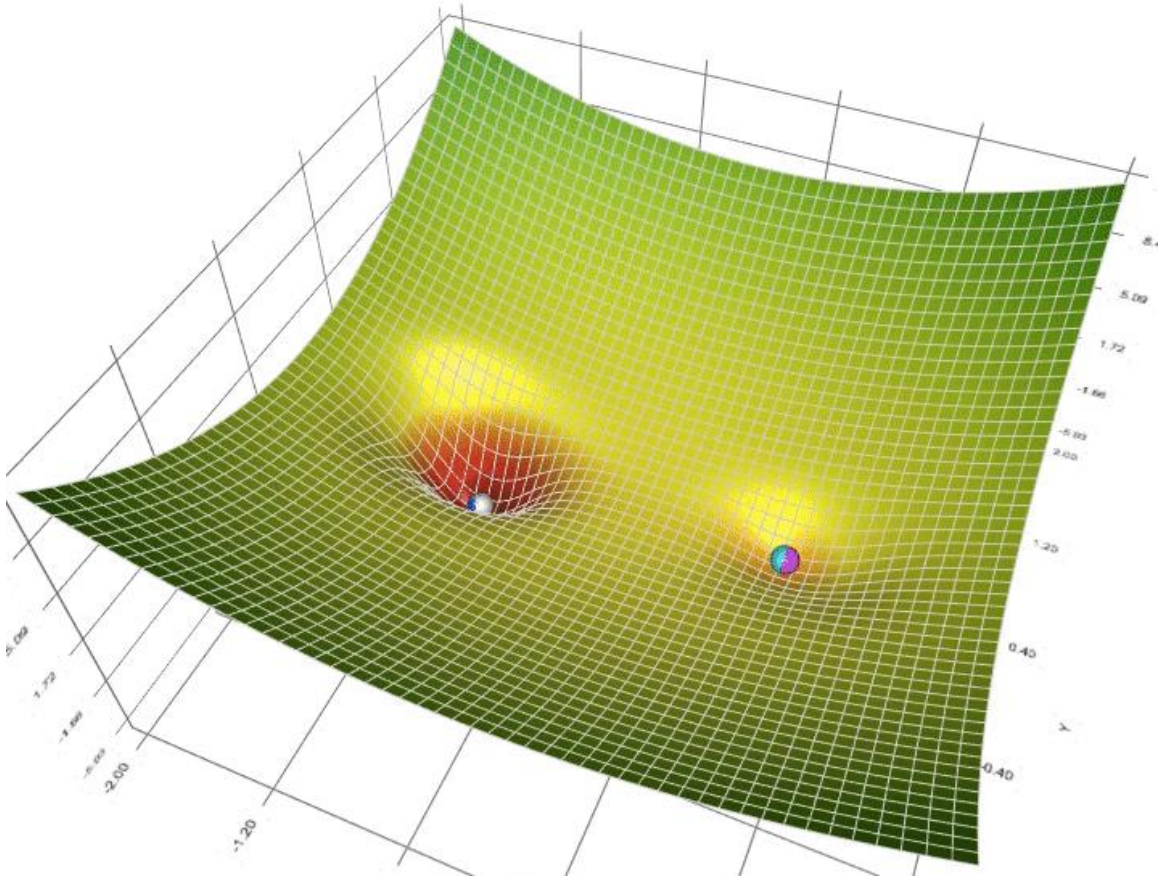
α is the **learning rate** and is very important…

How does it Work?

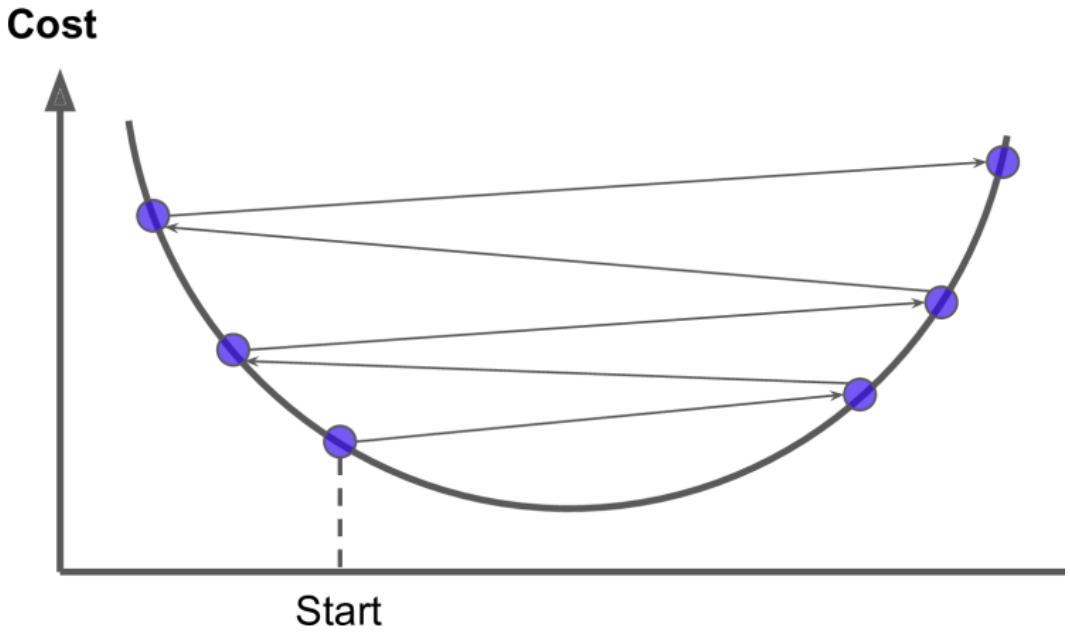


Adapted from: https://www.deeplearningbook.org/
See also: https://vision.unipv.it/AI/AIRG/DeepLearningAndTensorFlow2020-Episode1.pdf

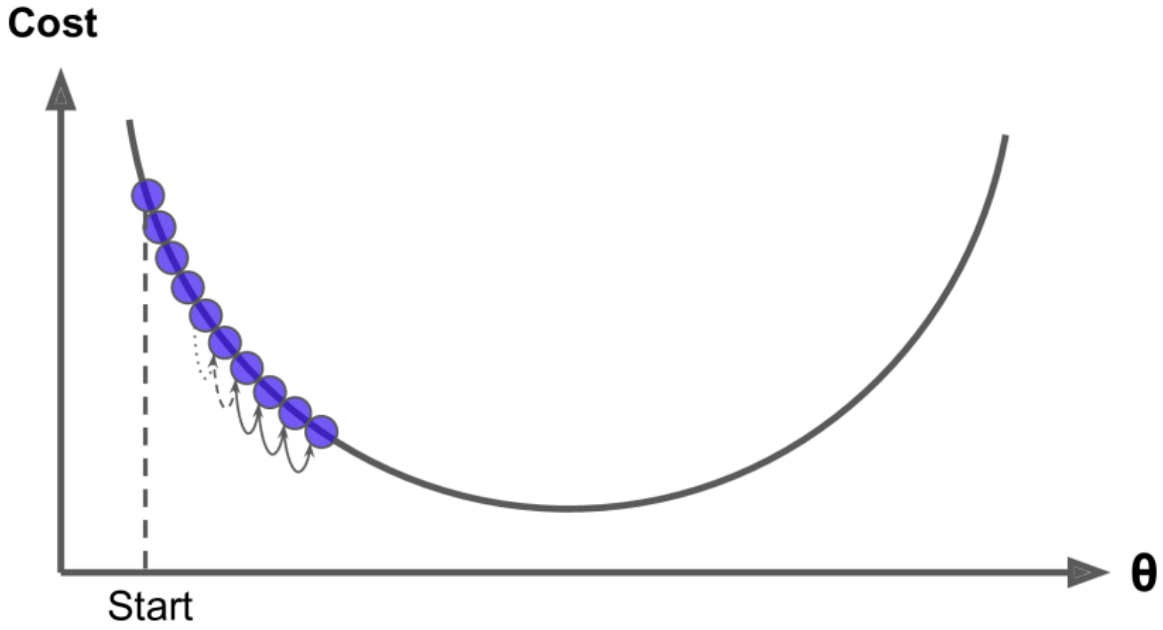
How does it Work?



Does it Work?

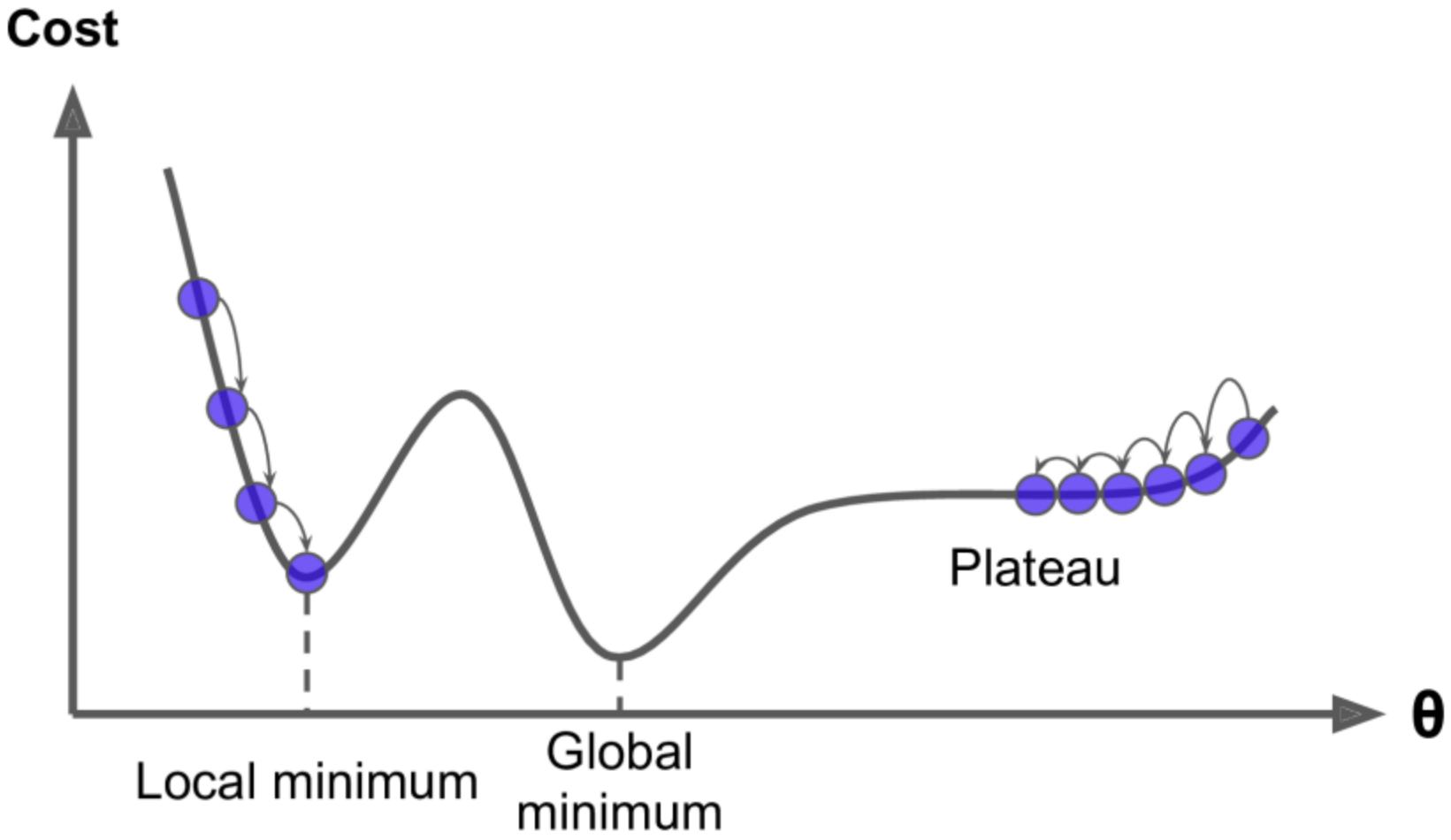


Learning rate too large ->
no convergence



Learning rate too small ->
takes forever

Does it Work?



Does it Work?

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

Then, if the learning rate α 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

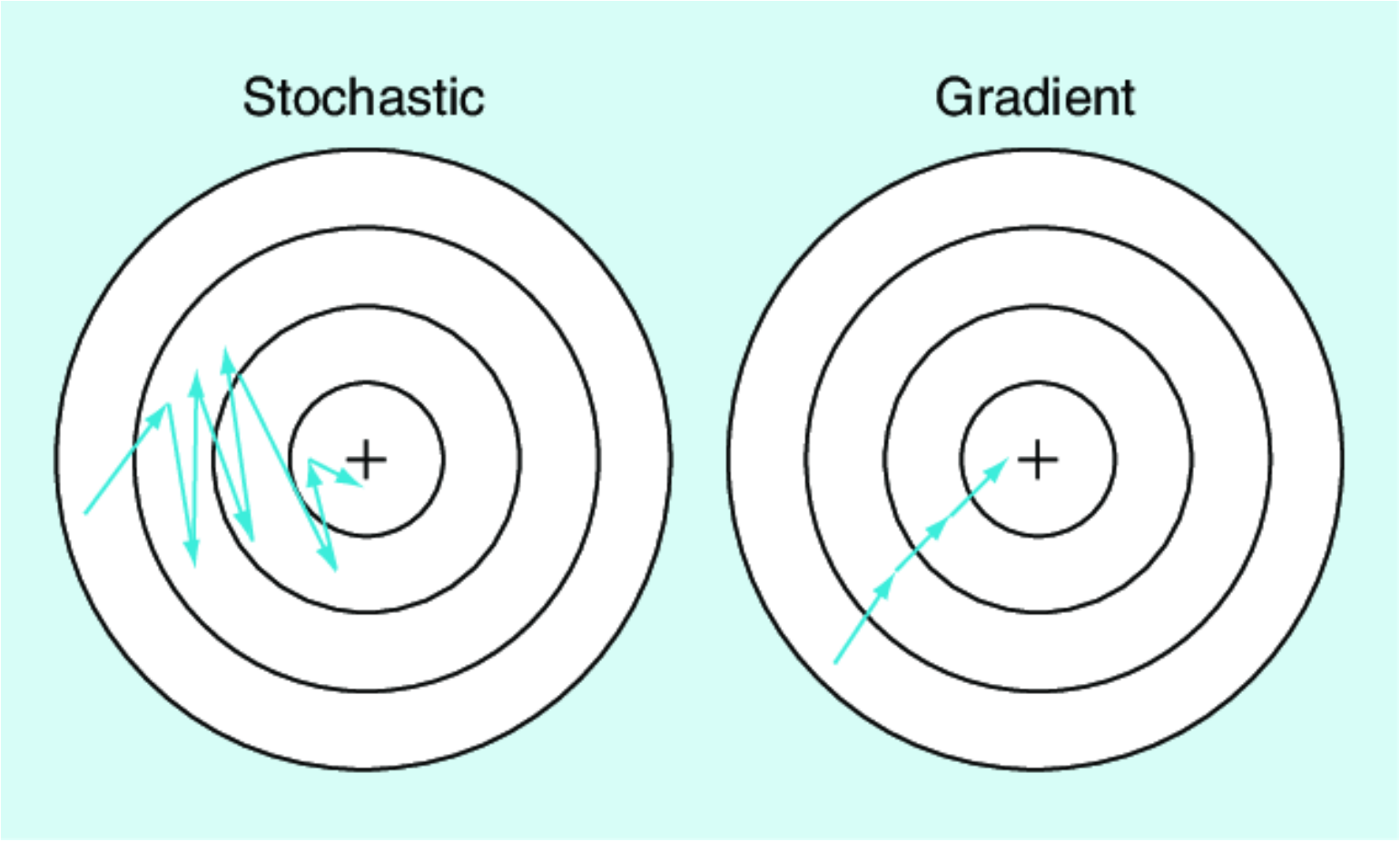
Most of the time, the loss is made of a sum over all the samples: $L(\boldsymbol{\vartheta}) = \sum_i L_i(\boldsymbol{\vartheta})$.

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

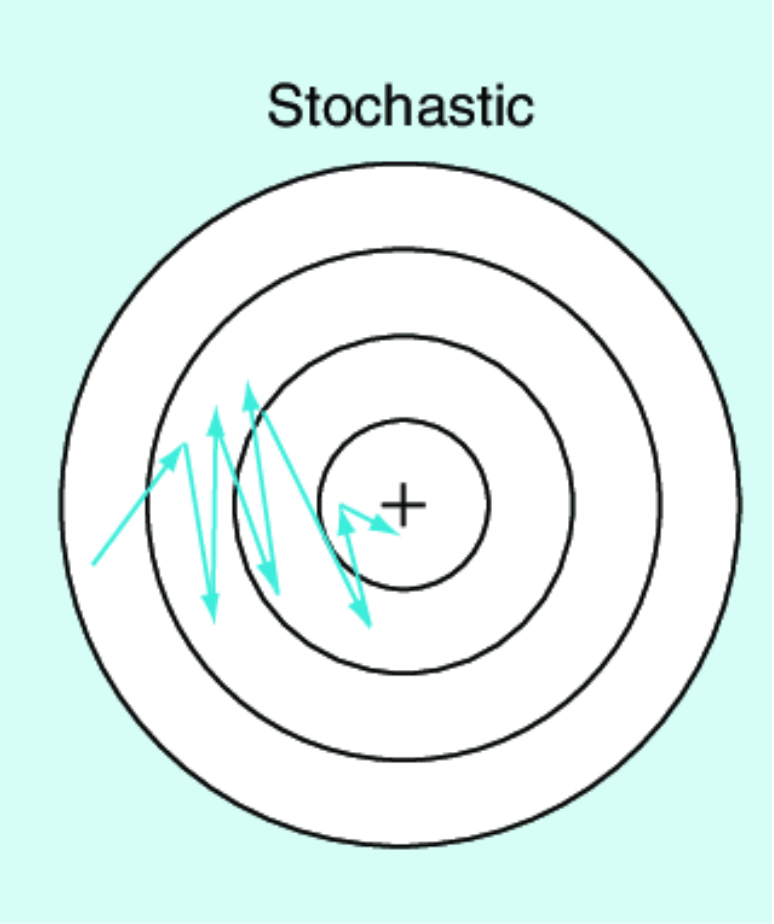


Stochastic Gradient Descent

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

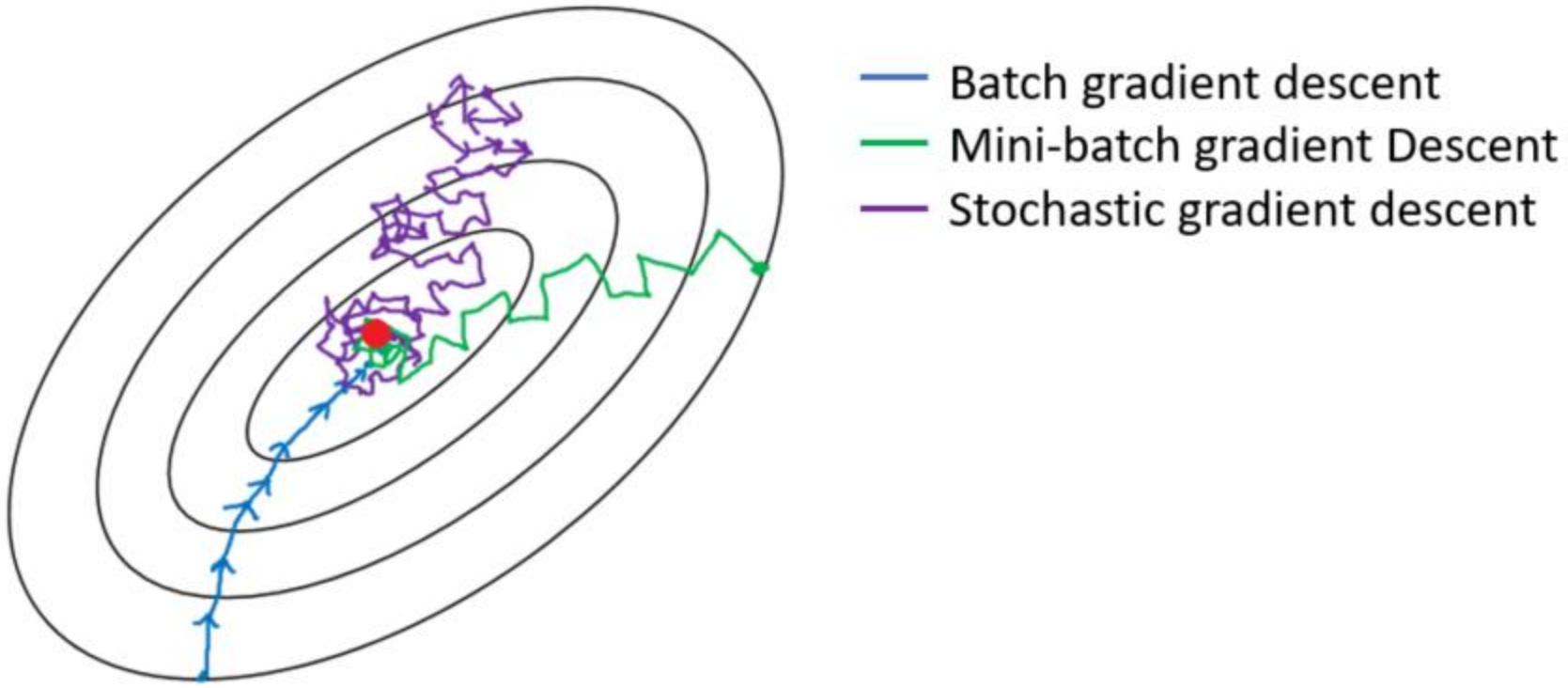
Instead of using all the samples or using only one, **we use b samples**, where b is chosen by the designer.

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 for a deeper discussion.

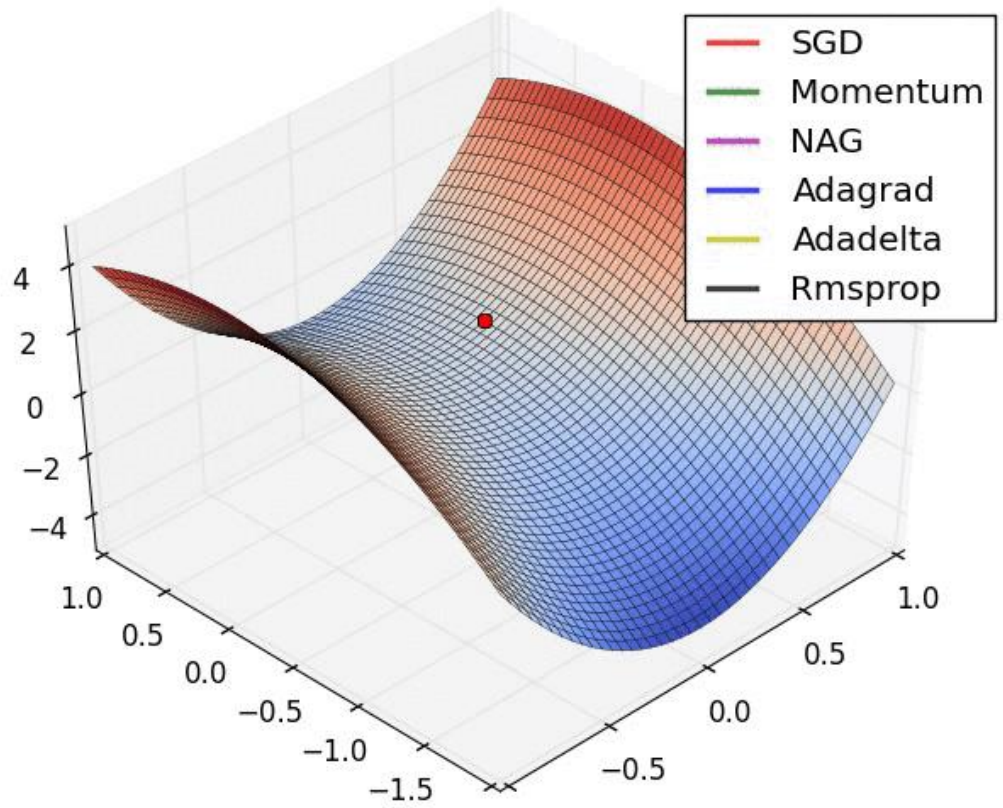
Moreover, do you need to **ever compute gradients**?

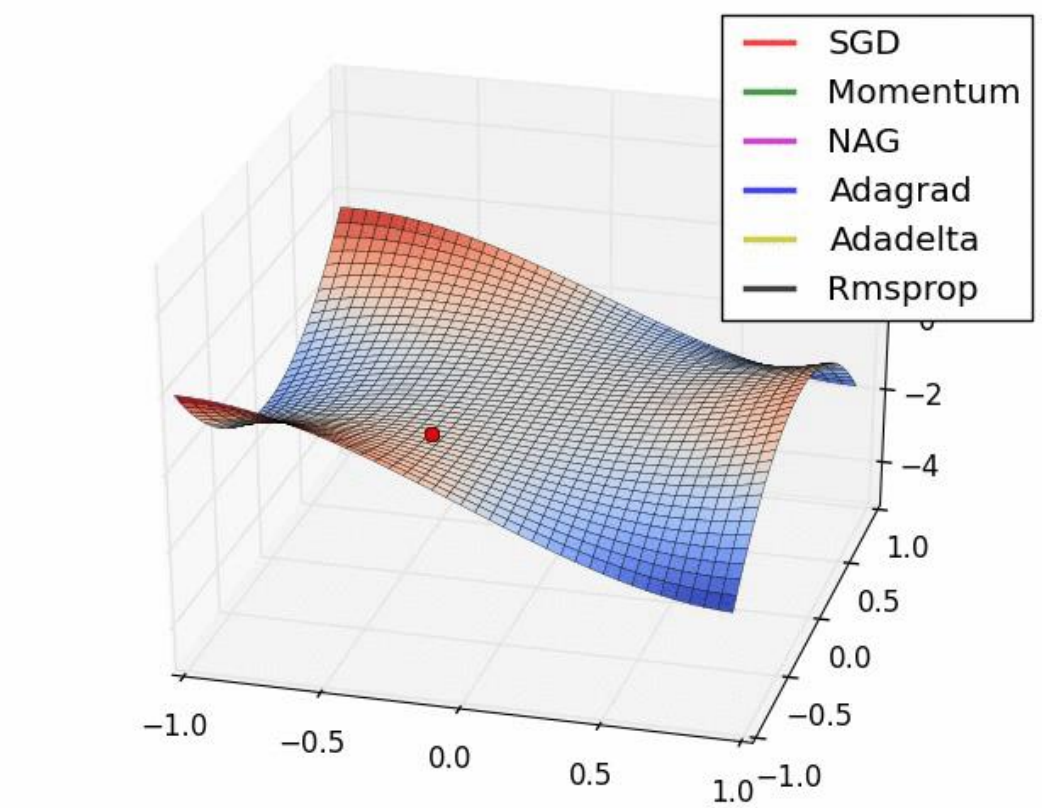
Hell no!

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 $f: R \to R$ and $g: R \to R$

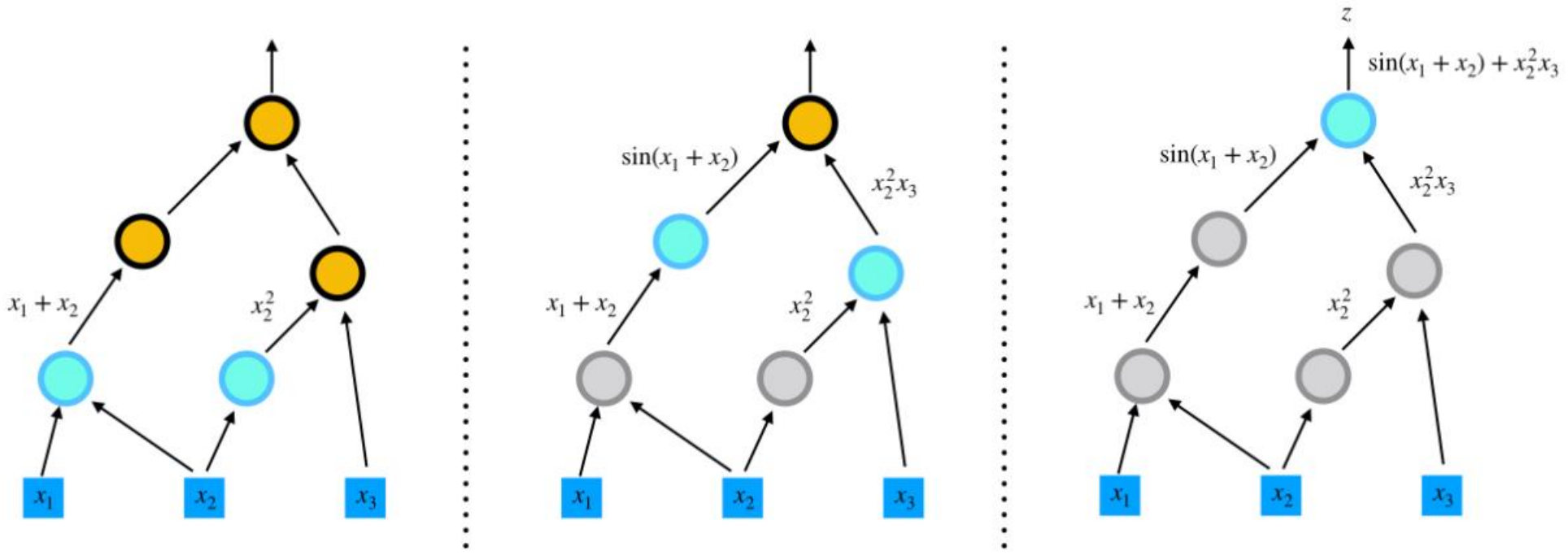
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}$$

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

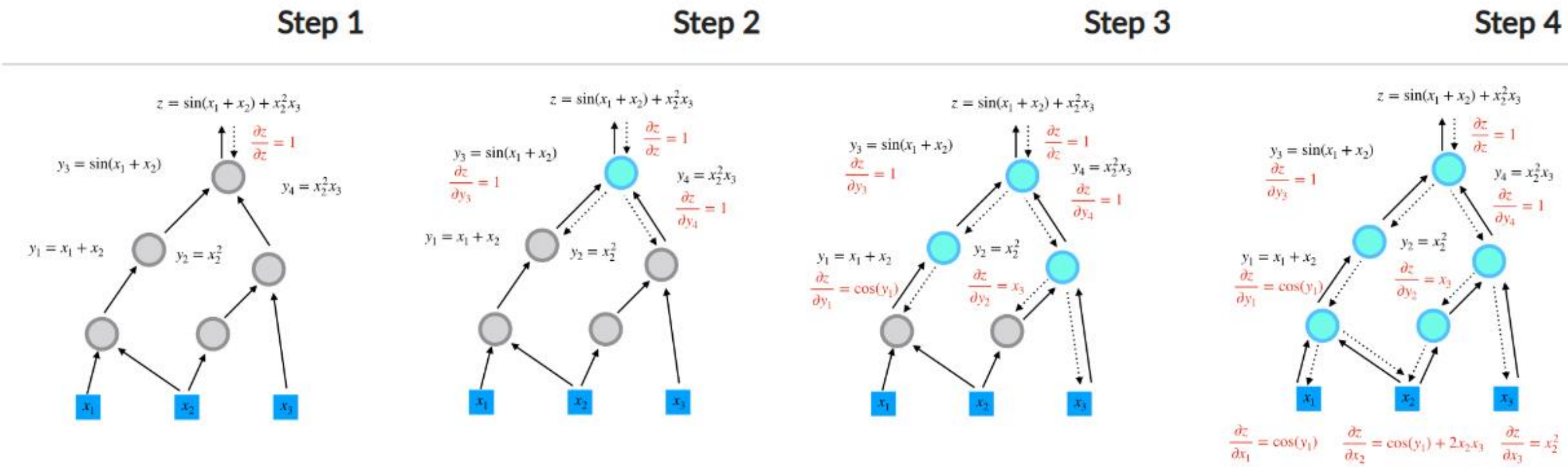
So, the gradient of every function can be **built** from the gradients of its basic blocs.

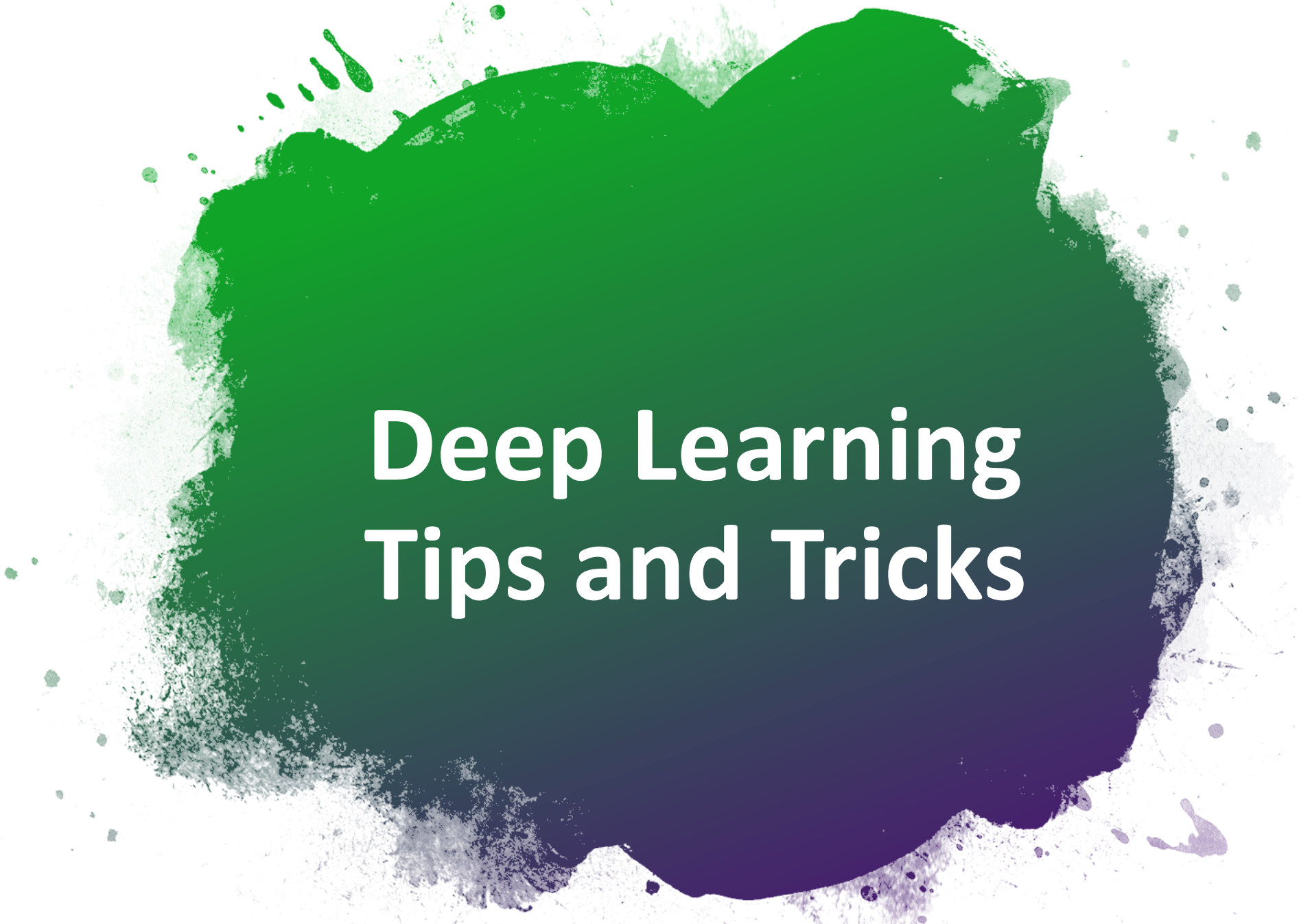
Automatic differentiation

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



Automatic differentiation



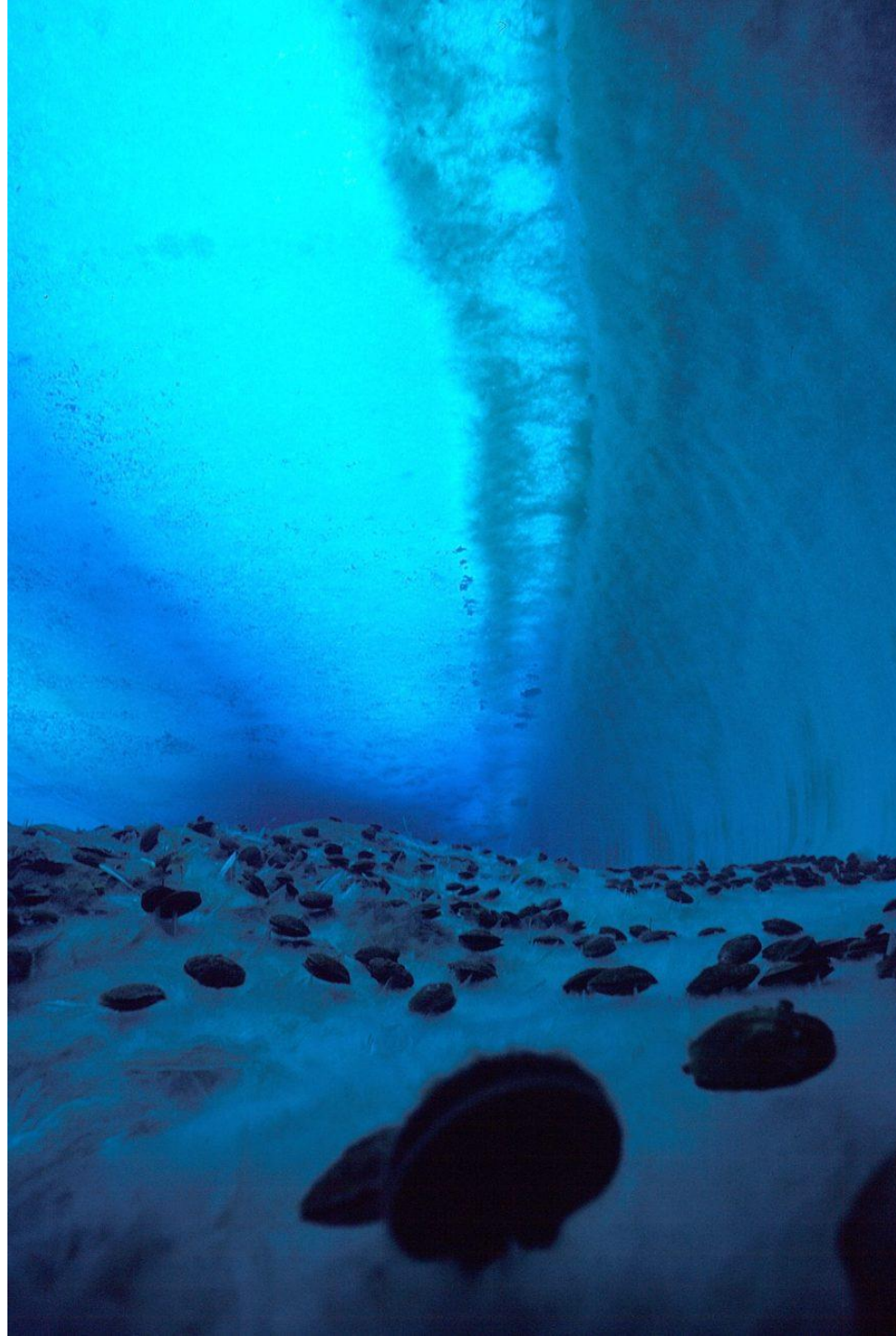

Deep Learning
Tips and Tricks

Disclaimer

Here we will discuss *some* common issues in Deep Learning.

Dy no means this is a comprehensive list, so feel free to **dive deeper**!

A good place to start is Dive into Deep Learning.



Vanishing and exploding gradient

A network with many layers is just a way to represent a composition of many functions.

By the chain rule, this means **many multiplications**.

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

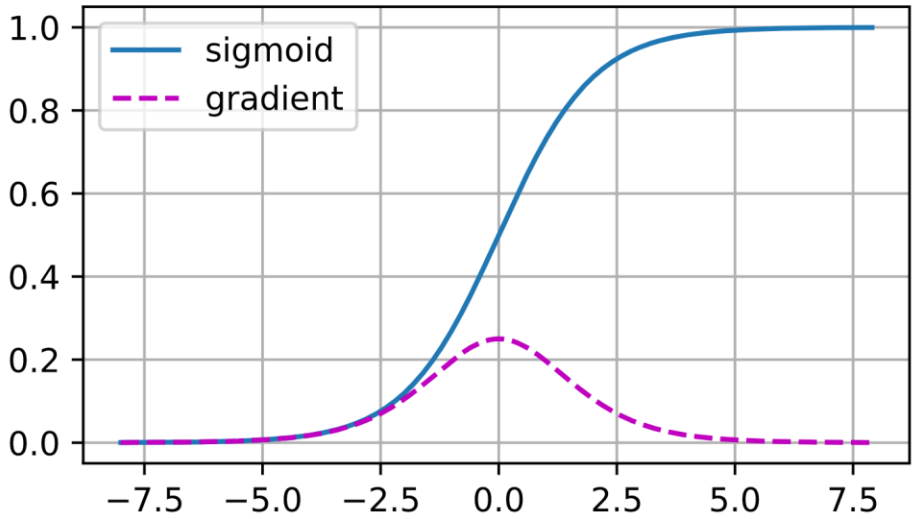
Due to this multiplications, the final gradient **can explode or vanish**.

Think about multiplying many large or small numbers…

How to prevent vanishing or exploding gradients?

All the parameters must be **initialized** properly – usually with small random numbers respecting some heuristics.

The non-linear activation function must be chosen carefully. Usually, we **prefer ReLU** or its variants **over sigmoid**, as the derivative of a sigmoid is very close to zero when its input is either large or small.

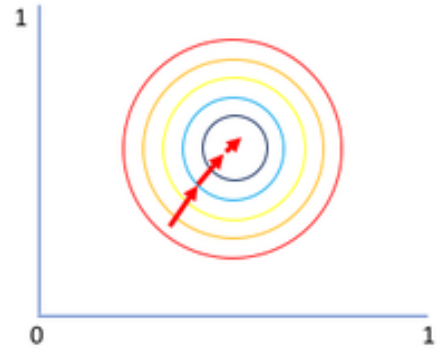


Always normalize the features

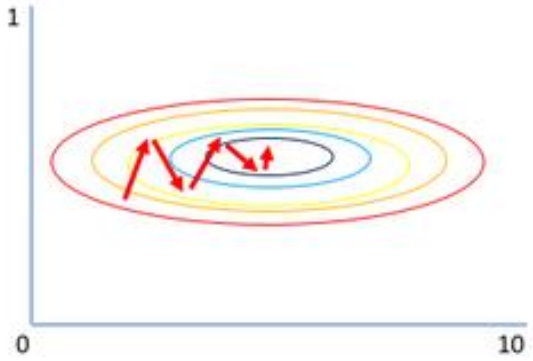
It is always good practice to normalize the data before training a Deep Learning model.

Unnormalized data **skew the gradient** and make the convergence of gradient descent slower and more difficult.

Nowadays, it is often preferred to normalize each mini-batch (**batch normalization**) than the whole dataset.



Both parameters can be updated in equal proportions



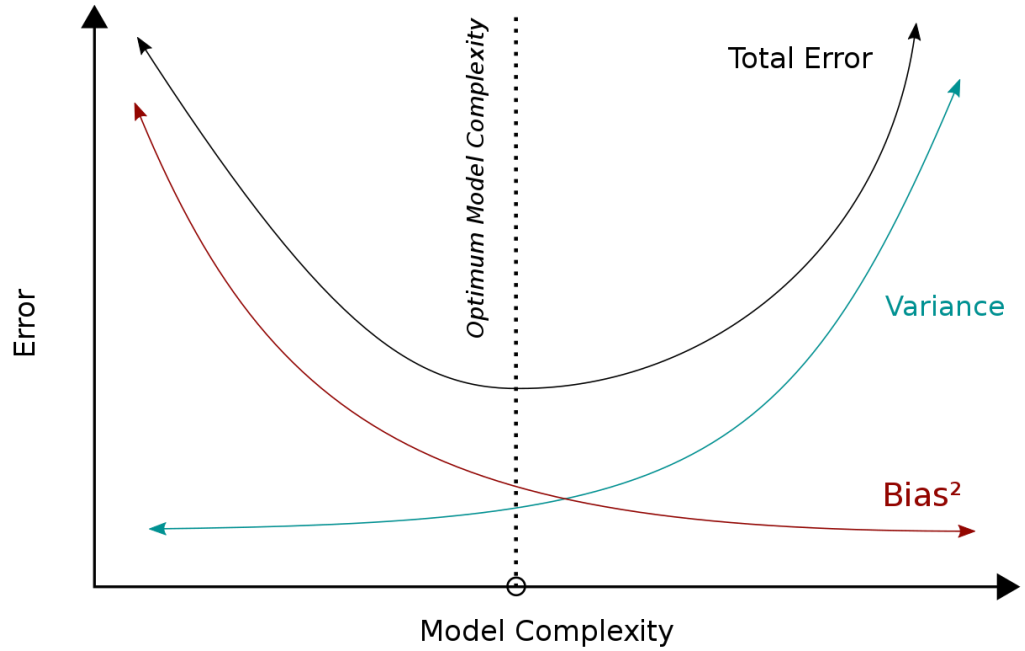
Gradient of larger parameter dominates the update

Generalization in Deep Learning

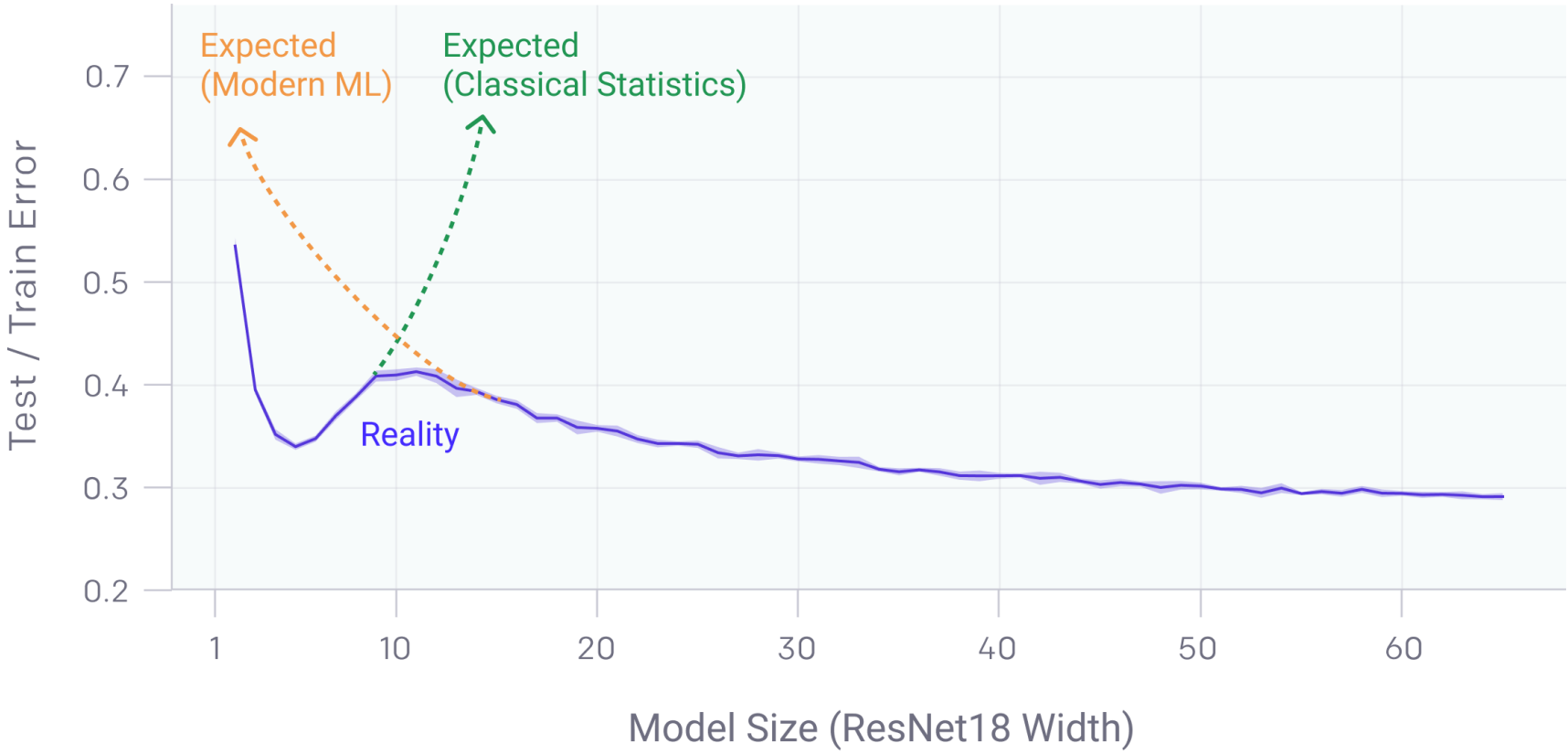
In classical machine learning overparameterization and excessive complexity are deemed dangerous, in view of the **bias-variance trade-off**.

However, in Deep Learning **different evidence emerges**.

Recent studies have shown that overparameterization results in smoother interpolation of the data and that additional complexity may hurt at first but help if we keep increasing it.



Generalization in Deep Learning



See https://arxiv.org/abs/2105.12806
See also: https://openai.com/blog/deep-double-descent/

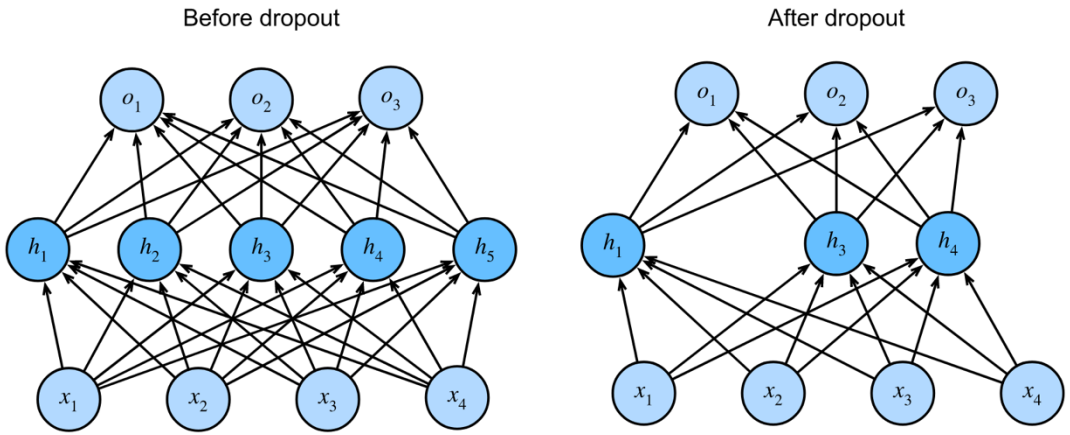
Dropout

In order to support a huge number of parameter while **controlling overfitting**, several regularization methods have been proposed.

Dropout is one of the most widely used.

When we train the network, at each epoch we **zero some inputs to each layer**. Each input is zeroed at random with probability p (an hyperparameter).

When we perform prediction, dropout is not used.




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".

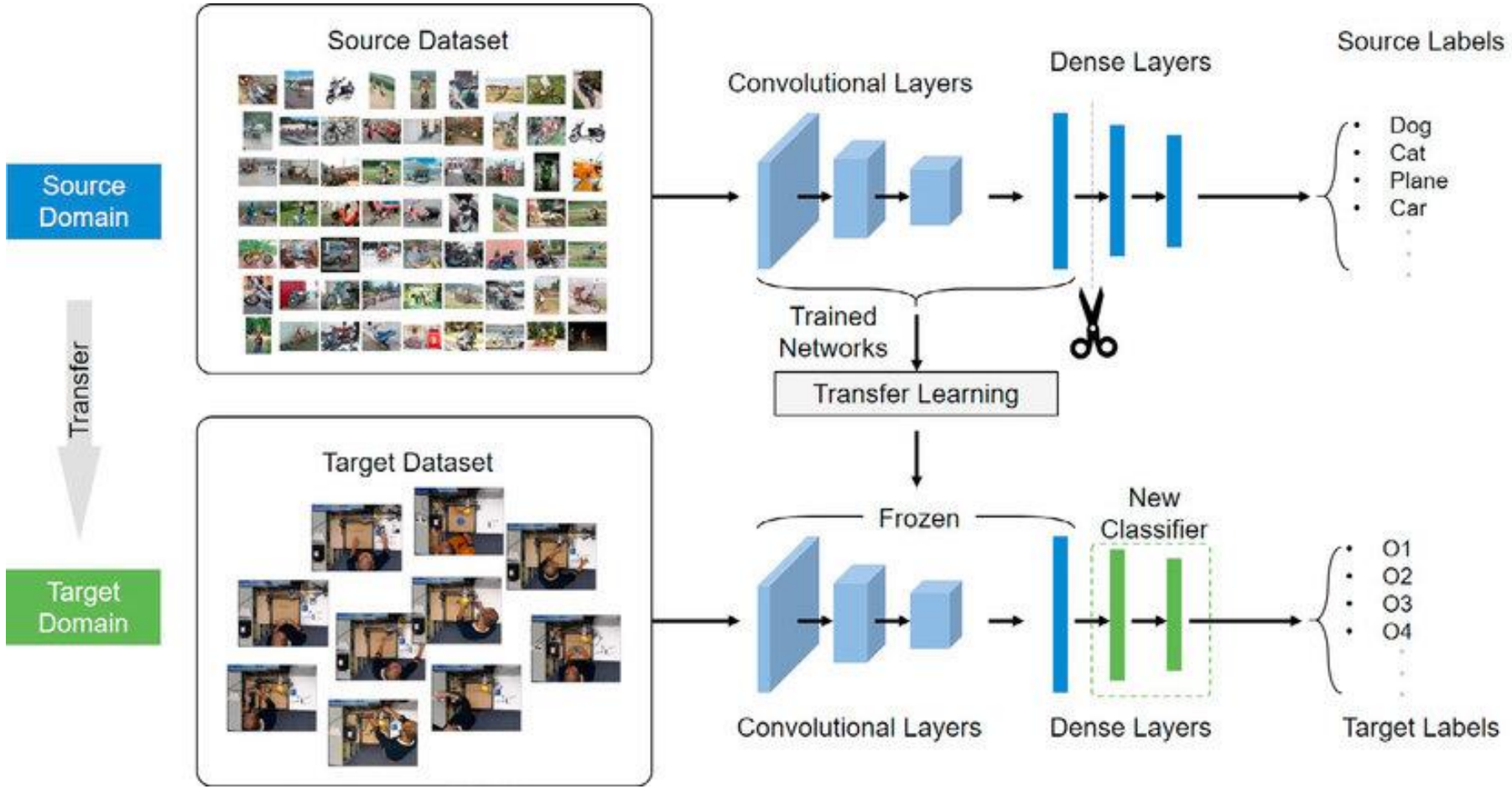
Instead of rolling your own architecture for a problem, you should look at whatever architecture currently works best on your problem, **download a pretrained model** and, at most, finetune it on your data.

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