Machine Learning for Physics and Astronomy

Juan Rojo

VU Amsterdam & Theory group, Nikhef

Natuur- en Sterrenkunde BSc (Joint Degree), Honours Track

Lecture 2, 07/09/2020

Today's lecture

- Optimisation strategies
- Stochastic Gradient Descent & Generic Algorithms
- (Deep) Neural Networks
- NN training: backpropagation

Supervised Learning:

Optimisation Strategies

Optimisation strategies

problems in **Supervised Machine Learning** are defined by the following ingredients:

(1) Input dataset: $\mathscr{D} = (\boldsymbol{X}, \boldsymbol{Y})$

(2) Model: $f\left(\boldsymbol{X}, \boldsymbol{\theta}\right)$

(3) Cost function: $C\left(\boldsymbol{Y}; f(\boldsymbol{X}; \boldsymbol{\theta}\right)$

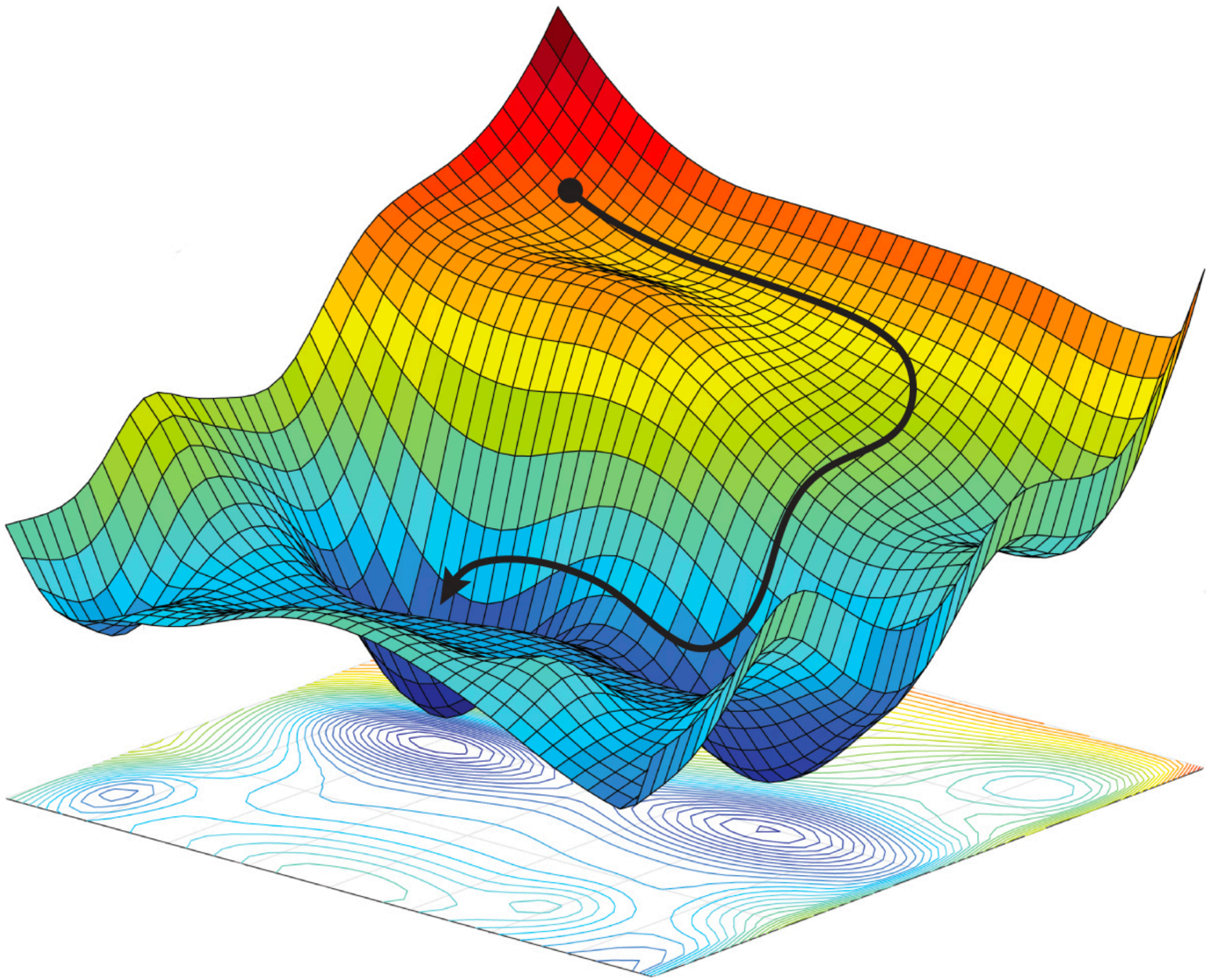
the model parameters are then found by **minimising the cost function**

for simple models (*e.g.* polynomial regression), the solution of this minimisation problem can be found analytically. This is not possible with complex models in realistic applications. In the context of ML studies, there exist two main classes of **minimisers (optimisers)** that are frequently deployed:

- **Gradient Descent** and its generalisations
- **Genetic Algorithms** and its generalisations

Gradient Descent

basic idea: iteratively adjust the model parameters in the direction where the **gradient of the cost function** is large and negative (**steepest descent direction**)



Gradient descent

basic idea: iteratively adjust the model parameters in the direction where the **gradient of the cost function** is large and negative (**steepest descent direction**)

Our goal is thus to **minimise an error (cost) function** that can usually be expressed as

$$E\left(\boldsymbol{\theta}\right) = \sum_{i=1}^{n} e_i(x_i;\boldsymbol{\theta})$$

e.g. in polynomial regression we had that

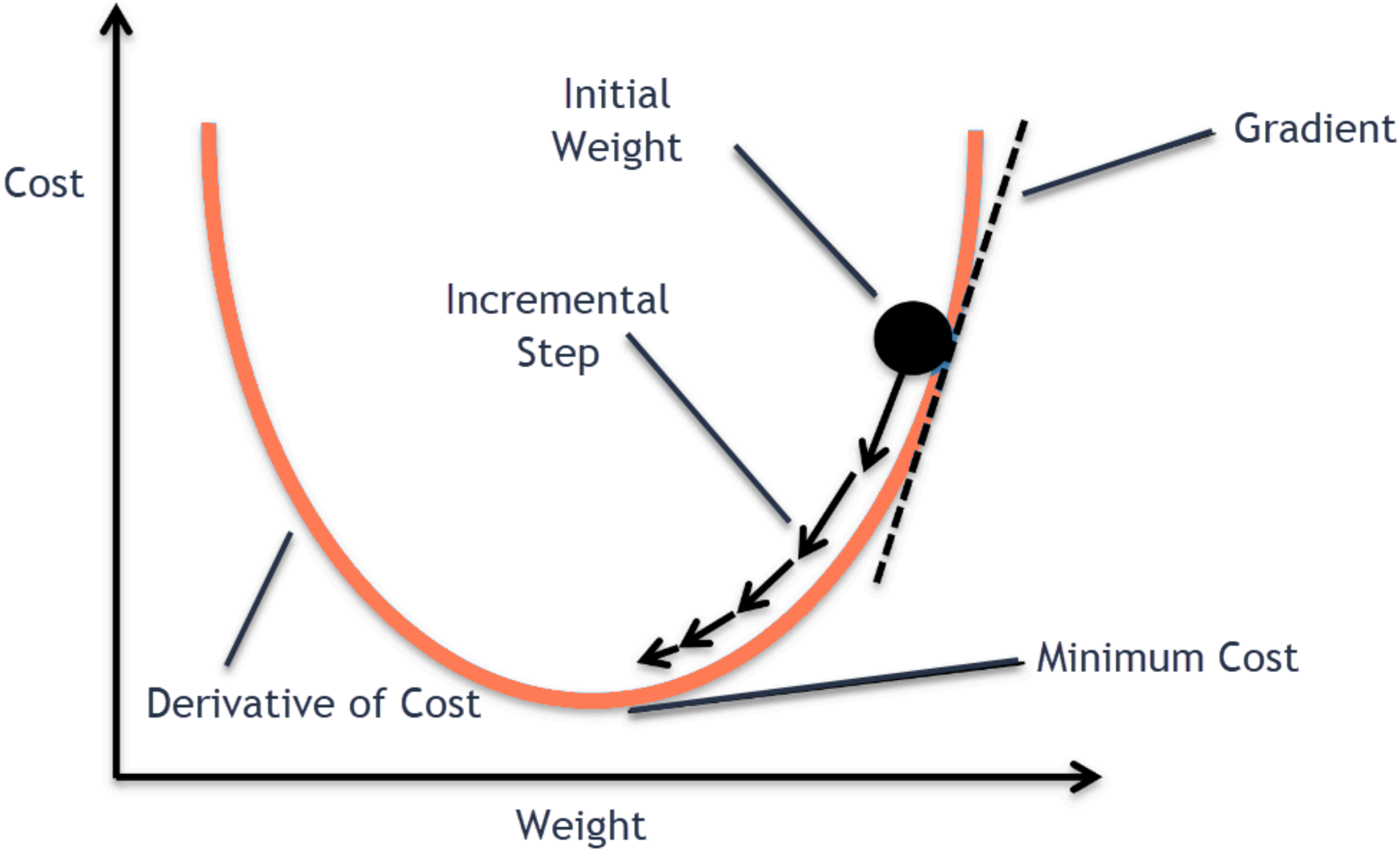
$$E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(y_i - f_\alpha(x_i;\boldsymbol{\theta}_\alpha)\right)^2 \quad \text{so} \quad e_i = \left(y_i - f_\alpha(x_i;\boldsymbol{\theta}_\alpha)\right)^2$$

and we will see that in **logistic regression** (classification problems)

$$E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(-y_i \ln \sigma(\boldsymbol{x}_i^T\boldsymbol{\theta}) - (1-y_i)\ln[1-\sigma(\boldsymbol{x}_i^T\boldsymbol{\theta}]\right)$$

aka the cross-entropy, relevant for categorisation

Gradient descent



Gradient descent

starting from a suitable initial condition, GD **iteratively updates** the model parameters:

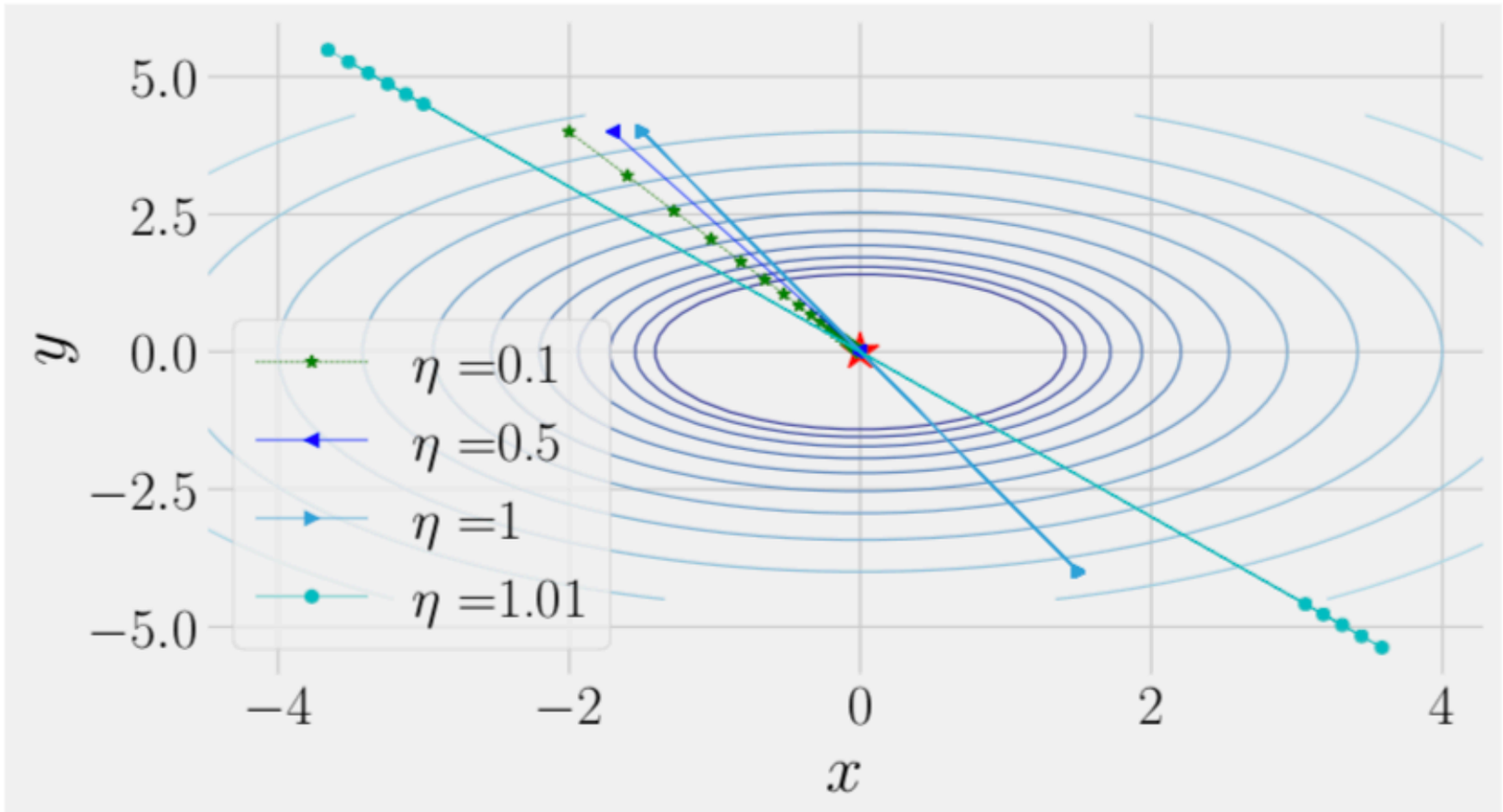
$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

learning rate

gradient of cost function

*update of model parameters between iterations **t** and **t+1***

the learning rate determines the size of the step in the direction of the gradient



small η: guaranteed to find local minimum, at price of large number of iterations

large η: can overshoot minimum, problems of convergence

nb evaluating gradient can be very cpu-intensive!

Gradient descent

starting from a suitable initial condition, GD **iteratively updates** the model parameters:

$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

learning rate

gradient of cost function

*update of model parameters between iterations **t** and **t+1***

compare GD with **Newton's method**: first Taylor-expand of cost function for a small change of the model parameters

$$E\left(\boldsymbol{\theta}+\mathbf{v}\right) \simeq E\left(\boldsymbol{\theta}\right) + \nabla_\theta E\left(\boldsymbol{\theta}\right)\cdot\mathbf{v} + \frac{1}{2}\mathbf{v}^T H(\boldsymbol{\theta})\mathbf{v}$$

$$H_{ij}(\boldsymbol{\theta}) = \left\{ \frac{\partial^2 E(\boldsymbol{\theta})}{\partial\theta_i \partial\theta_j} \right\}$$

Hessian matrix (2nd derivatives)

Gradient descent

starting from a suitable initial condition, GD **iteratively updates** the model parameters:

$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

learning rate

gradient of cost function

*update of model parameters between iterations **t** and **t+1***

compare GD with **Newton's method**: first Taylor-expand of cost function for a small change of the model parameters

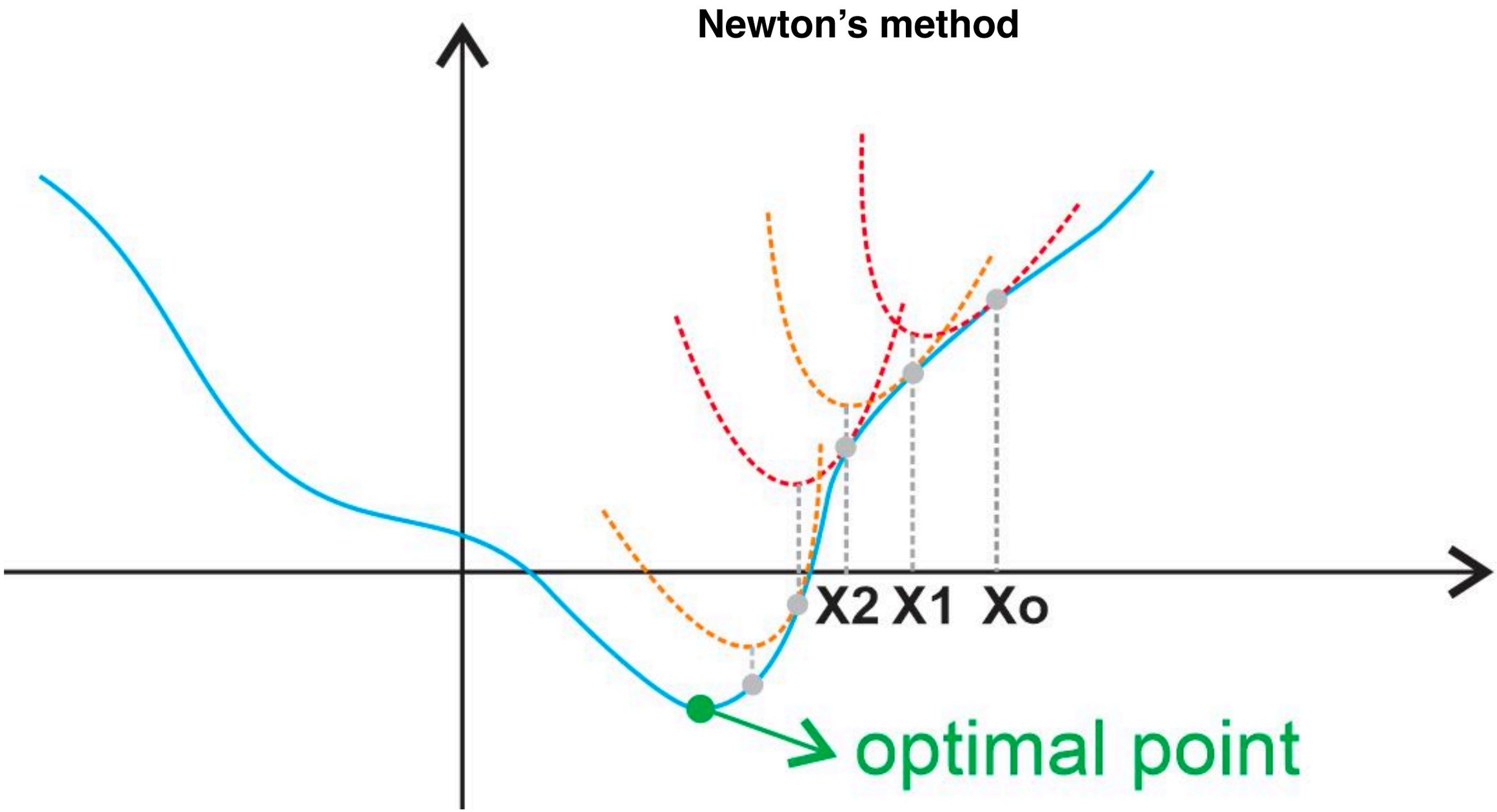
$$E\left(\boldsymbol{\theta} + \mathbf{v}\right) \simeq E\left(\boldsymbol{\theta}\right) + \nabla_\theta E\left(\boldsymbol{\theta}\right) \cdot \mathbf{v} + \frac{1}{2}\mathbf{v}^T H(\boldsymbol{\theta})\mathbf{v}$$

and then differentiate with respect to the step requiring that the **linear term vanishes**

$$\left.\frac{\partial E\left(\boldsymbol{\theta} + v\right)}{\partial \mathbf{v}}\right|_{\mathbf{v}_{\text{opt}}} = 0 \longrightarrow \nabla_\theta E\left(\boldsymbol{\theta}\right) + H(\boldsymbol{\theta})\mathbf{v}_{\text{opt}} = \mathbf{0}$$

$$\mathbf{v}_t = H^{-1}(\boldsymbol{\theta}_t) \cdot \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

Gradient descent



Gradient descent

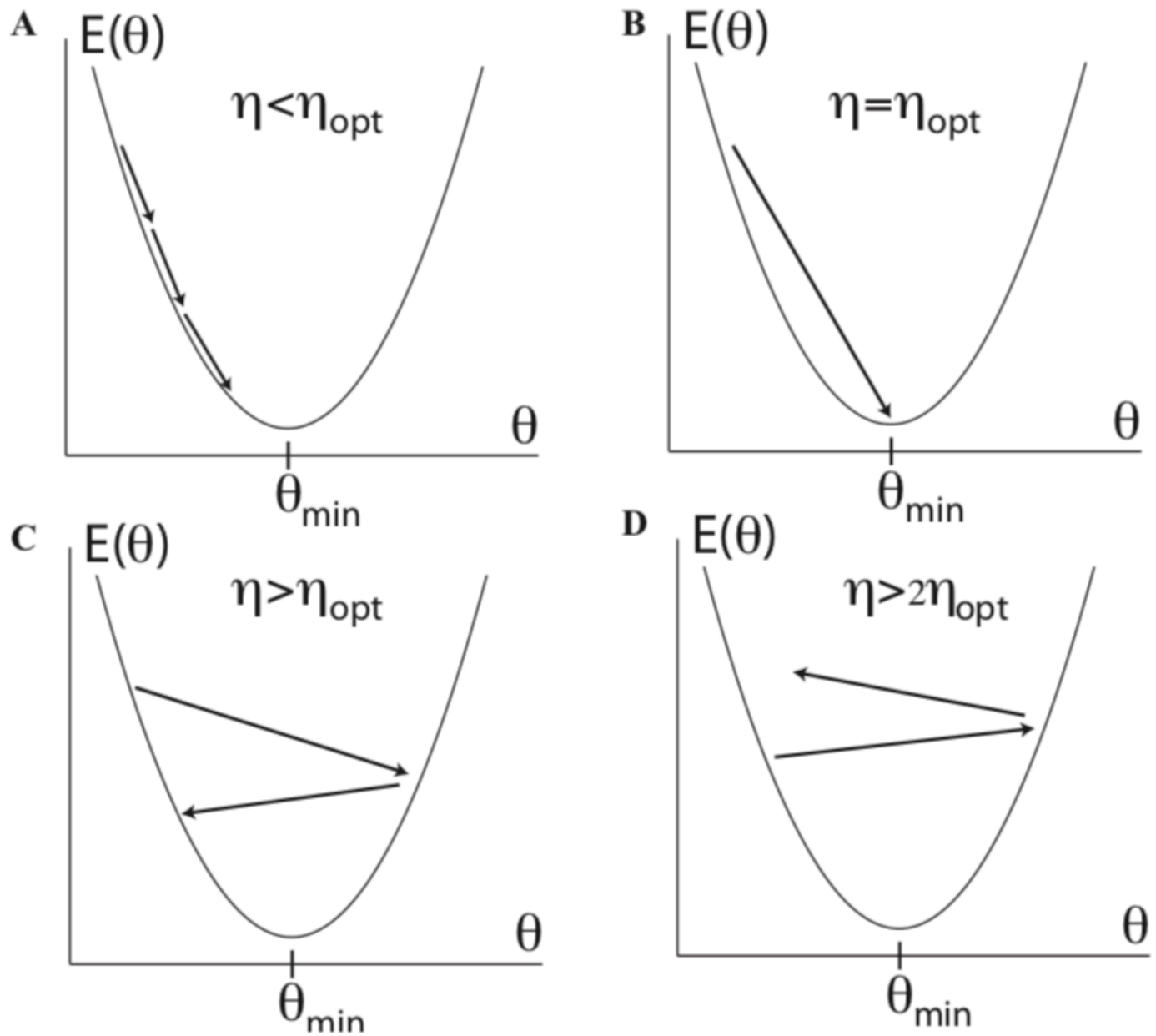
Gradient Descent ⟶ $\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$

Newton's method ⟶ $\mathbf{v}_t = H^{-1}(\theta_t) \cdot \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$

- Newton's method not practical for ML applications: it involves **evaluation and inversion of Hessian matrices with M^2 entries**, with M = number of model parameters

- But it provides useful ideas about how to improve GD, for example by **adapting the learning rate to the local curvature** of region of the parameter space

suggest improvements of GD!



Advanced gradient descent

the simplest implementation of GD, unsurprisingly, has several limitations

- It converges to **local, rather than global**, minima of the cost function

poor performance for realistic applications

Advanced gradient descent

the simplest implementation of GD, unsurprisingly, has several limitations

- It converges to **local, rather than global**, minima of the cost function

poor performance for realistic applications

- Evaluating gradients is **computationally expensive** for large datasets

$$E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(y_i - f_\alpha(x_i; \boldsymbol{\theta}_\alpha)\right)^2$$

involves sum over all n data points

Advanced gradient descent

the simplest implementation of GD, unsurprisingly, has several limitations

- It converges to **local, rather than global**, minima of the cost function

poor performance for realistic applications

- Evaluating gradients is **computationally expensive** for large datasets

$$E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(y_i - f_\alpha(x_i; \boldsymbol{\theta}_\alpha)\right)^2$$

involves sum over all n data points

- Very sensitive to choice of **learning rates**

Ideally one would like an adaptive learning rate

Advanced gradient descent

the simplest implementation of GD, unsurprisingly, has several limitations

- It converges to **local, rather than global**, minima of the cost function

poor performance for realistic applications

- Evaluating gradients is **computationally expensive** for large datasets

$$E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(y_i - f_\alpha(x_i; \boldsymbol{\theta}_\alpha)\right)^2$$

involves sum over all n data points

- Very sensitive to choice of **learning rates**

Ideally one would like an adaptive learning rate

- Treats **uniformly all directions** in the parameter space

we would like to use info also on curvature (as in Newton's method)

Generalised Gradient Descent methods have been developed to address these shortcomings and are at the basis of **modern deep learning methods**

Stochastic gradient descent

Stochasticity can be added to GD by approximating the gradient on a subset of the training data, called a **mini-batch**

$$\nabla_\theta E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \nabla_\theta e_i(\boldsymbol{x}_i;\boldsymbol{\theta}) \simeq \nabla_\theta E^{\mathrm{MB}}(\boldsymbol{\theta}) \equiv \sum_{i\in B_k} \nabla_\theta e_i(\boldsymbol{x}_i;\boldsymbol{\theta})$$

$k = 1,\ldots,n/K$ ← *# points per batches*

 # batches

We then **cycle over all mini-batches**, updating the model parameters at each step *k*

$$\mathbf{v}_t = \eta_t \nabla_\theta^{\mathrm{MB}} E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

A full iteration over all *n* data points (over the *n/K* batches) is called an **epoch**

benefits of SGD: stochasticity prevents getting stuck in local minima, the calculation of the gradient is speed up & stochasticity acts as natural regulariser

Adding momentum to SGD

SGD can be used with a **momentum term** that provides some **memory** on the direction in which one is moving in the parameter space

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta^{\mathrm{MB}} E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

momentum parameter (0 < ɣ < 1)

$$\Delta\boldsymbol{\theta}_{t+1} = \gamma \Delta\boldsymbol{\theta}_t - \eta_t \nabla_\theta E(\boldsymbol{\theta}_t), \qquad \Delta\boldsymbol{\theta}_t = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$$

proposed parameter change in iteration t+1

proposed parameter change in iteration t

momentum in SGD helps to **gain speed in directions with persistent but small gradients**, while suppressing oscillations in high-curvature directions.

Adaptative Gradient Descent

Adaptative GD varies the learning rate to reflect the **local curvature** of the parameter space without the need to evaluate the Hessian matrix

RMSprop: keep track also of the **second moment of gradient**, similar as how the momentum term is a running average of previous gradients

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\boldsymbol{g}_t}{\sqrt{\boldsymbol{s}_t + \epsilon}}$$

regulator to avoid numerical divergences

$$\boldsymbol{g}_t = \nabla_\theta E(\boldsymbol{\theta})$$

gradient at iteration t

$$\boldsymbol{s}_t = \beta \boldsymbol{s}_{t-1} + (1-\beta)\boldsymbol{g}_t^2$$

controls averaging time of 2nd moment of gradient

$$\boldsymbol{s}_t = \mathbb{E}[\boldsymbol{g}_t^2]$$

2nd moment of gradient (averaged over iterations)

$$\beta = 1 \rightarrow \boldsymbol{s}_t = \boldsymbol{s}_{t-1}$$

$$\beta = 0 \rightarrow \boldsymbol{s}_t = \boldsymbol{g}_t^2$$

effective learning rate reduced in directions where gradient is consistently large

Adaptative Gradient Descent

Adaptative GD varies the learning rate to reflect the **local curvature** of the parameter space without the need to evaluate the Hessian matrix

ADAM: adaptively change the parameters from information on **running averages** of first and second moments of the gradient

$$\boldsymbol{g}_t = \nabla_\theta E(\boldsymbol{\theta})$$

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} - (1-\beta_1)\boldsymbol{g}_t \qquad \boldsymbol{s}_t = \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2$$

$$\widehat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1-(\beta_1)^t} \qquad \widehat{\boldsymbol{s}}_t = \frac{\boldsymbol{s}_t}{1-(\beta_2)^t}$$

sets lifetime of first moment

sets lifetime of second moment

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\widehat{\boldsymbol{m}}_t}{\sqrt{\widehat{\boldsymbol{s}}_t + \epsilon}}$$

Adaptative Gradient Descent

Adaptative GD varies the learning rate to reflect the **local curvature** of the parameter space without the need to evaluate the Hessian matrix

ADAM has two main advantages: *(i)* adapting step size to cut off large gradient directions (prevent oscillations) and *(ii)* measuring gradients in a natural length scale

to see this, express the ADAM update rules in terms of the **variance in parameter space**

$$\sigma_t = \hat{s}_t - \left(\hat{m}_t\right)^2$$

and we can see that the update rule for one of the parameters reads now

$$\Delta\theta_{t+1} = -\eta_t \frac{\hat{m}_t}{\sqrt{\sigma_t^2 + \hat{m}_t^2 + \epsilon}}$$

small fluctuations of gradient

$$\Delta\theta_{t+1} \to -\eta_t$$

large fluctuations of gradient

$$\Delta\theta_{t+1} = -\eta_t \hat{m}_t / \sigma_t^2$$

learning rate promotional to signal-to-noise ratio

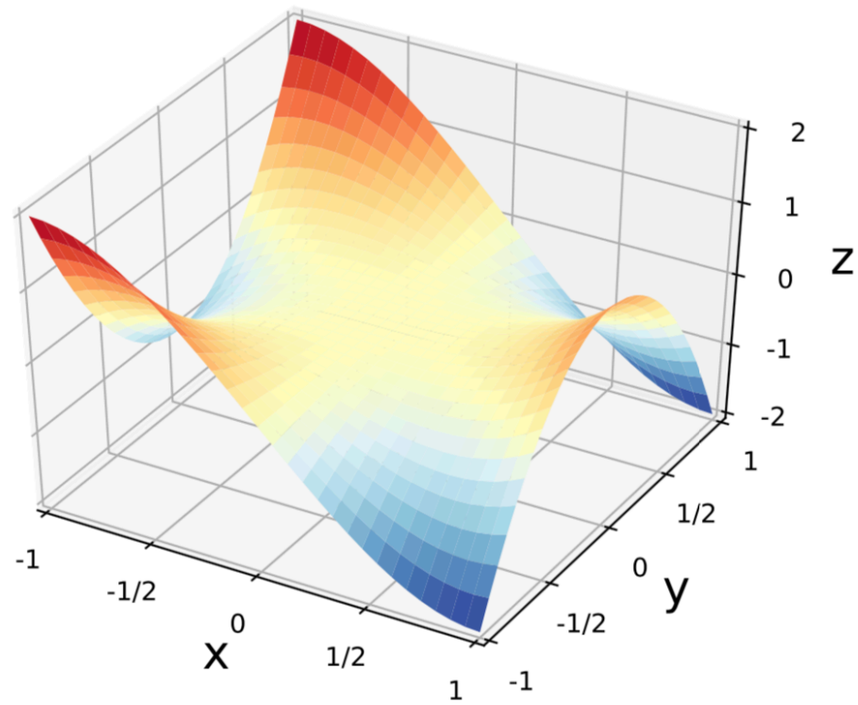
Tutorial 1 Exercise 1b

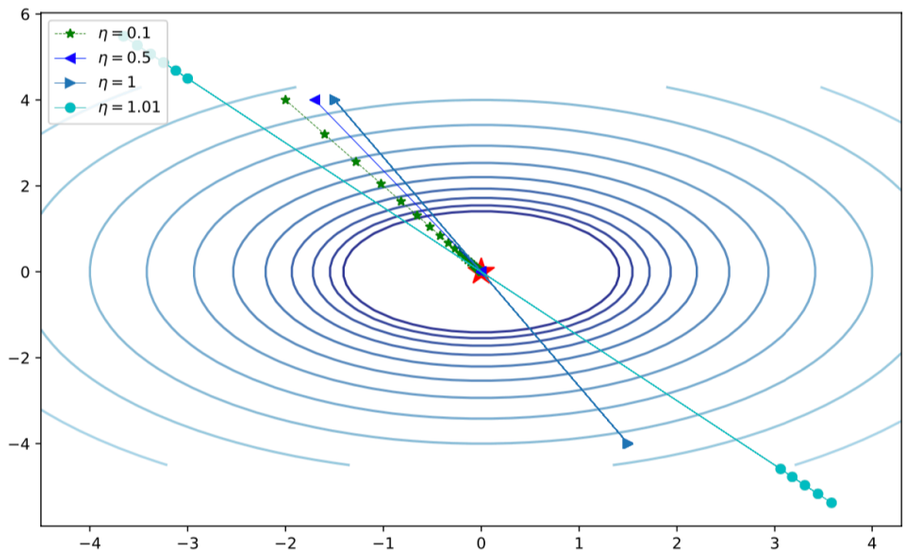
- Study different optimisation algorithms based on GD for simple 2D functions

- Visualise the **path of GD algorithms in parameter space**

- Determine which choices of the parameters allow reaching the minima quicker

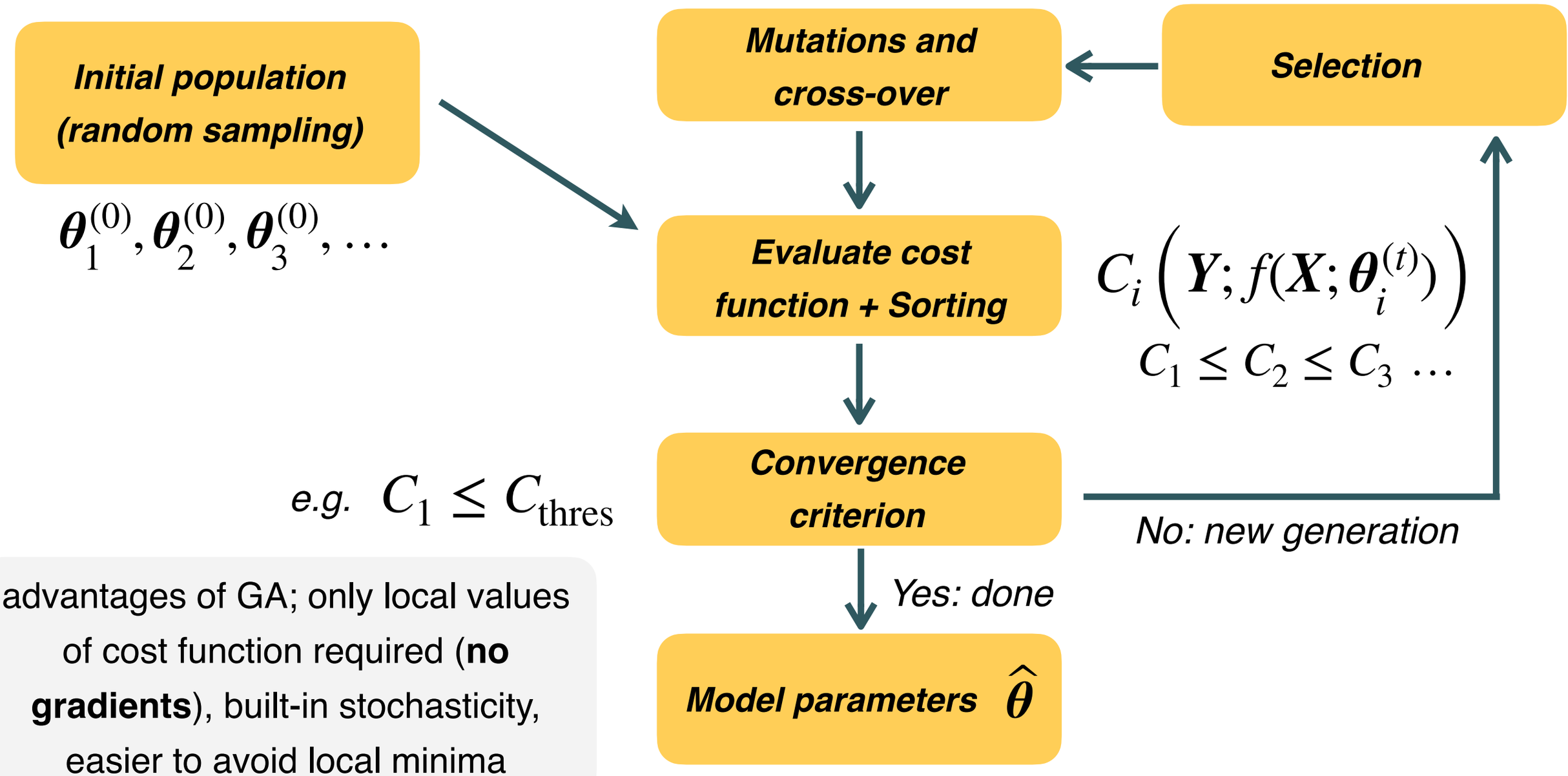
- Given new 2D functions, show that you can **tune GD to find all the minima**, and compare with the analytical results





Genetic Algorithms

GAs combine **stochastic and deterministic ingredients** to explore the model parameter space and minimise the cost function



e.g. $C_1 \leq C_{\text{thres}}$

advantages of GA; only local values of cost function required (**no gradients**), built-in stochasticity, easier to avoid local minima

Genetic Algorithms

GAs combine **stochastic and deterministic ingredients** to explore the model parameter space and minimise the cost function

mutations

$$\theta_{i,j}^{(t+1)} = \theta_{i,j}^{(t)} \left(1 + \eta_j^{(t)}\right) , \quad j = 1 \ \ldots, m, i = 1, \ldots, M$$

learning rates (depends in generation and parameter)

 # model parameters

 # mutants

crossover

$$\boldsymbol{\theta}_i^{(t)} = \left(\theta_{i,1}^{(t)}, \theta_{i,2}^{(t)}, \ldots, \theta_{i,m}^{(t)}\right) \rightarrow \left(\theta_{i,1}^{(t)}, \theta_{i,2}^{(t)}, \ldots, \theta_{i,p-1}^{(t)}, \theta_{k,p}^{(t)}, \ldots, \theta_{k,m}^{(t)}\right)$$

$$\boldsymbol{\theta}_k^{(t)} = \left(\theta_{k,1}^{(t)}, \theta_{k,2}^{(t)}, \ldots, \theta_{k,m}^{(t)}\right) \rightarrow \left(\theta_{k,1}^{(t)}, \theta_{k,2}^{(t)}, \ldots, \theta_{k,p-1}^{(t)}, \theta_{i,p}^{(t)}, \ldots, \theta_{i,m}^{(t)}\right)$$

children inherit part of the DNA of each parent

Genetic Algorithms

GAs combine **stochastic and deterministic ingredients** to explore the model parameter space and minimise the cost function

mutations

$$\theta_{i,j}^{(t+1)} = \theta_{i,j}^{(t)} \left(1 + \eta_j^{(t)}\right) , \quad j = 1 \ \ldots, m, i = 1, \ldots, M$$

learning rates (depends in generation and parameter)

 # model parameters

 # mutants

crossover

Chromosome1	11011|00100110110
Chromosome2	11011|11000011110
Offspring1	11011|11000011110
Offspring2	11011|00100110110

children inherit part of the DNA of each parent

Genetic Algorithms

we can illustrate how **GAs work in practice** in a graphical way with this example

The CMA-ES algorithm

Improve the efficiency of simple GAs by incorporating global (in addition to local) information on the parameter space

One example is the **Covariance matrix Adaptation - Evolutionary Strategy** (CMA-ES)

- Start by randomly initialising the model parameters using a Gaussian distribution

$$\boldsymbol{\theta}^{(0)} \sim \mathcal{N}\left(0, \mathbf{C}^{(0)}\right)$$

- At every generation, we generate ***M* mutants** according to the following distribution

$$\boldsymbol{a}_k^{(t)} \sim \boldsymbol{\theta}^{(t-1)} + \boldsymbol{\sigma}^{(t-1)} \mathcal{N}\left(0, \mathbf{C}^{(t-1)}\right), \quad k = 1, \ldots, M$$

step size

covariance matrix of search distribution

tuned during fit

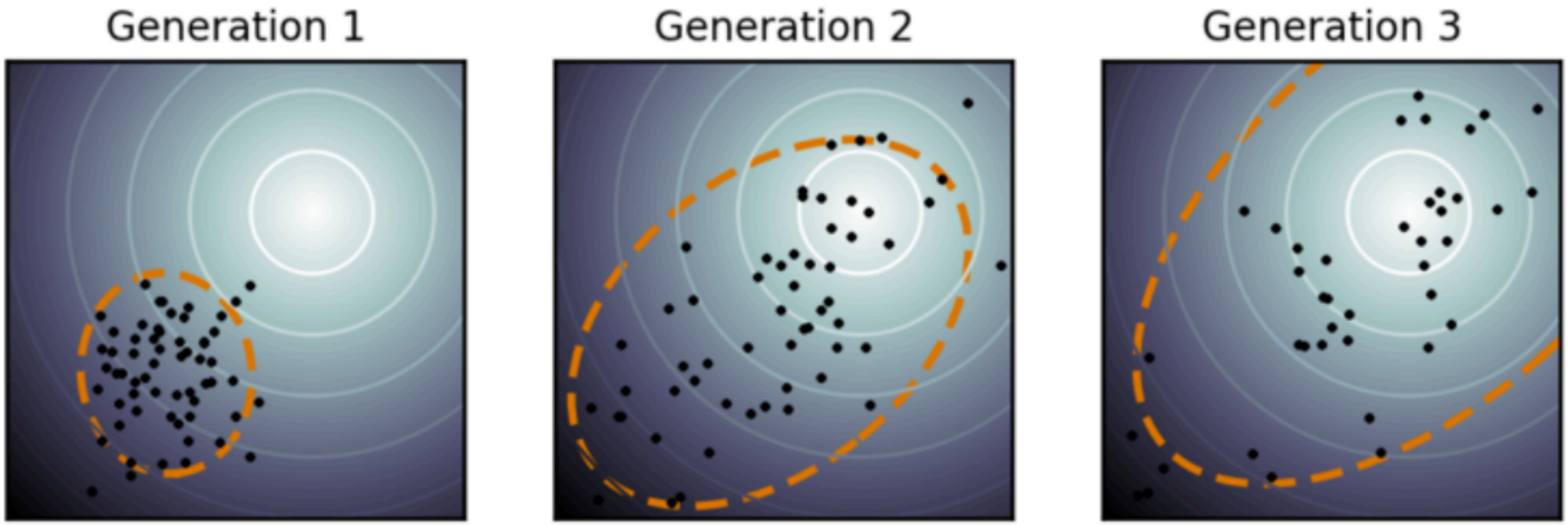
- The **new search centre** is computed as weighted average over fraction of best mutants

$$\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} + \sum_{k=1}^{\mu} W_k (\boldsymbol{a}_k - \boldsymbol{\theta}^{(t-1)})$$

Parameters of the algorithms

adaptation as the parameter space is explored is key feature of CMA-ES

The CMA-ES algorithm

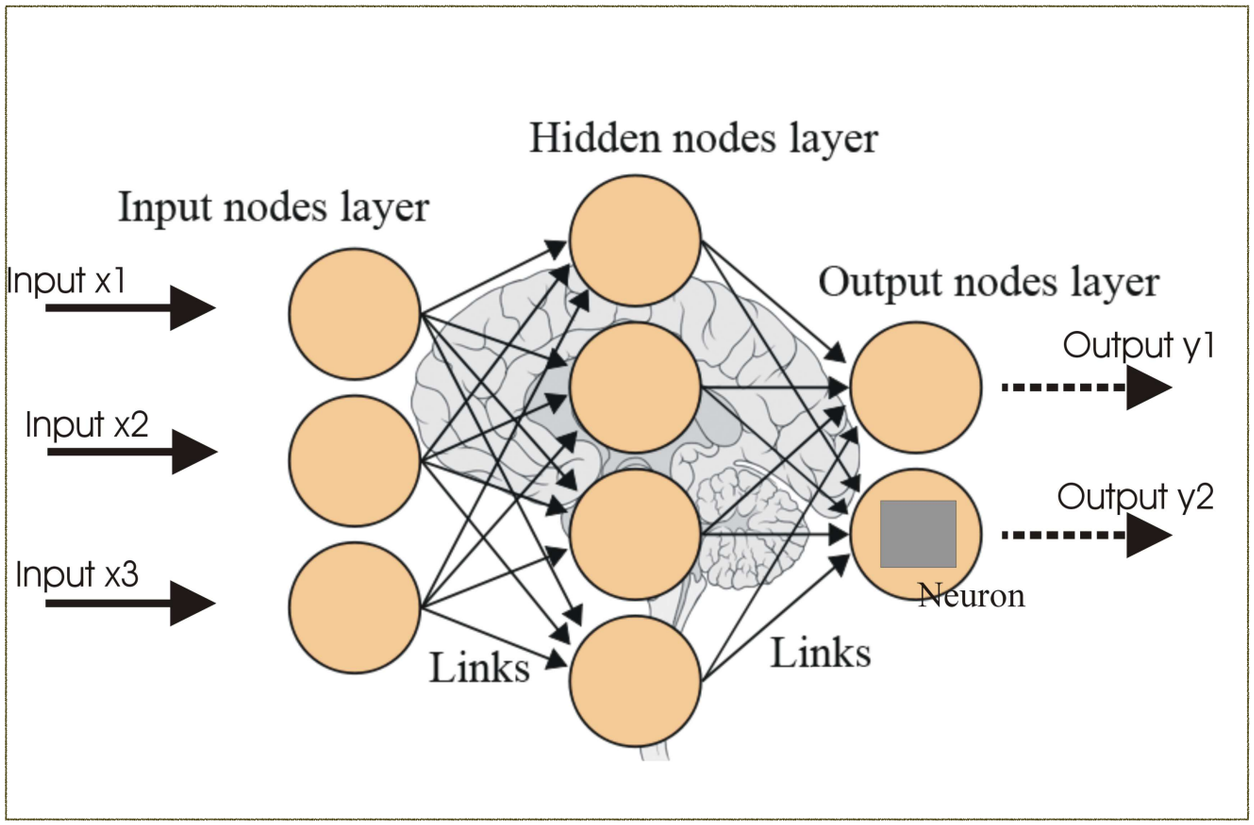


main advantages: (i) information on gradients never required, only local values of cost function, (ii) information on whole set of mutants (not only best ones) used to tune step size and search covariance matrix

(Deep) Neural Networks

Artificial Neural Networks

Inspired by **biological brain models**, **Artificial Neural Networks** (ANNs) are mathematical algorithms designed to excel where domains as their evolution-driven counterparts outperforms traditional algorithms in tasks such as **pattern recognition, forecasting, classification**, ...



in ML context, ANN provide a flexible, powerful non-linear model for many problems

Neural Networks

Neural Nets can be defined as **neural-inspired nonlinear models** for supervised learning

- The basic unit of a NN is the **neuron,** a transformation of a set of *d* input features into a scalar output

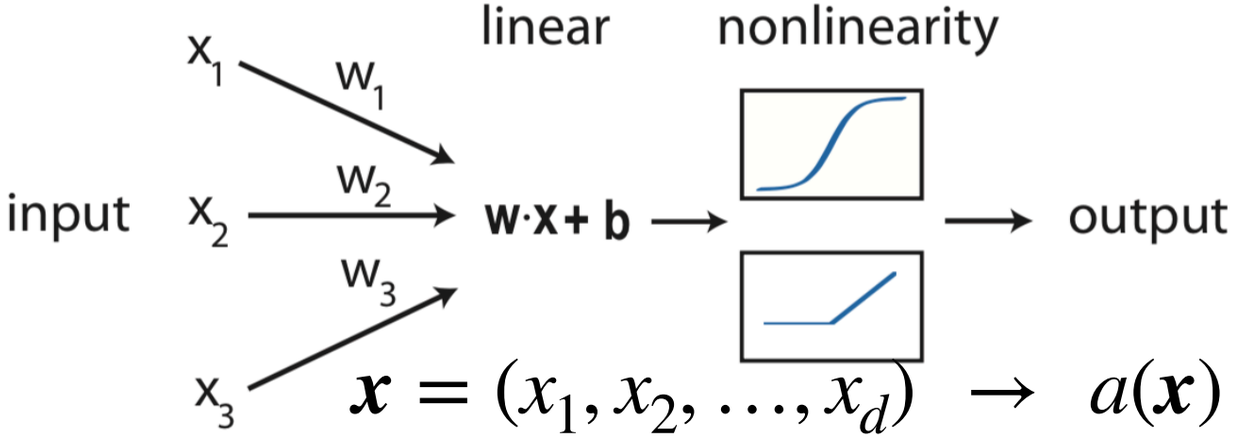
$$\boldsymbol{x} = (x_1, x_2, \ldots, x_d) \rightarrow a(\boldsymbol{x})$$

mapping

neuron inputs

neuron output (activation state)

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_d) \qquad a(\boldsymbol{x})$$



biology analog: input are electric pulses, output the activation state of the neuron

Neural Networks

Neural Nets can be defined as **neural-inspired nonlinear models** for supervised learning

- The basic unit of a NN is the **neuron,** a transformation of a set of *d* input features into a scalar output

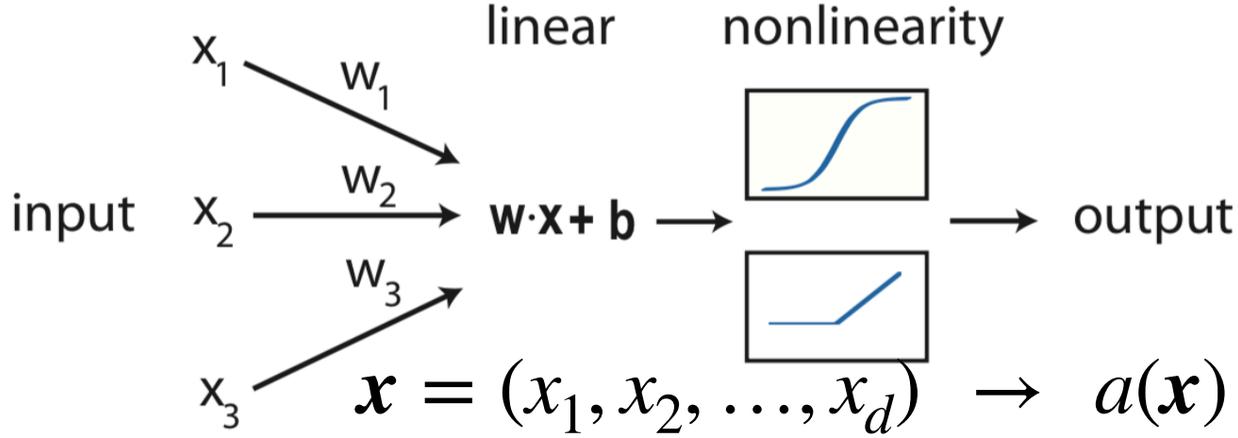
$$\boldsymbol{x} = (x_1, x_2, \ldots, x_d) \rightarrow a(\boldsymbol{x})$$

mapping

neuron inputs

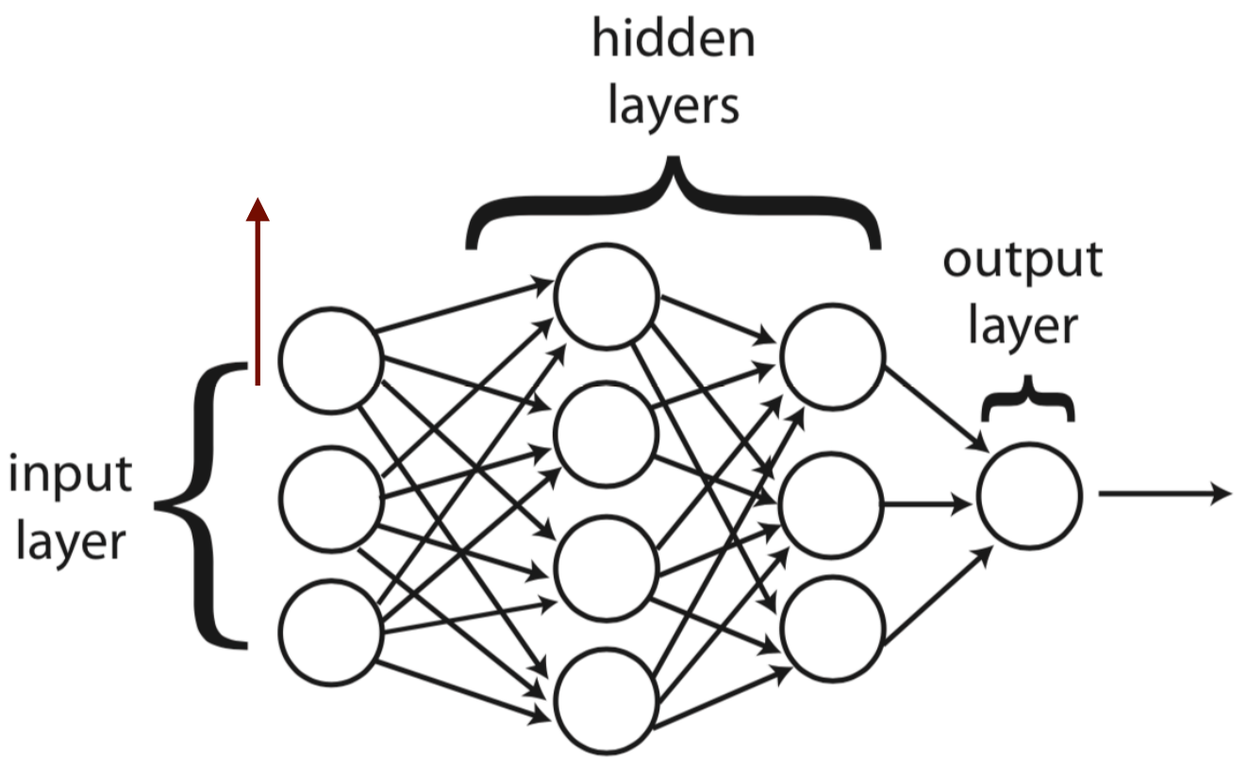
neuron output (activation state)

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_d) \qquad a(\boldsymbol{x})$$



- These neutrons are arranged in **layers**, which in turn are stacked on each other. The intermediate layers are called **hidden layers**

- Here we will focus on **feed-forward NNs,** where the output of the neurons of the previous layer becomes the input of the neurons in the subsequent layer



Neural Networks

Neural Nets can be defined as **neural-inspired nonlinear models** for supervised learning

the neural network output has two components

a **linear operation**, weighting the various inputs

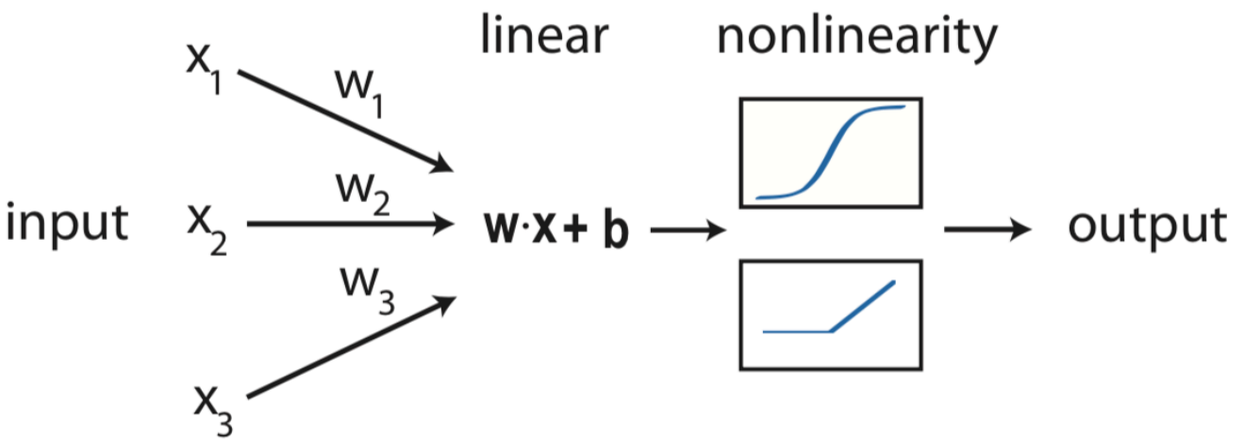
$$z^{(i)} = \boldsymbol{x}^T \cdot \boldsymbol{\theta}^{(i)} + \boldsymbol{\theta}_0^{(i)}$$

neuron inputs — *i-th neuron parameters* — *i-th neuron bias*

a **non-linear transformation**

$$a^{(i)}(\boldsymbol{x}) = \sigma_i(z^{(i)})$$

i-th neuron activation state — *i-th neuron activation function (non-linear)*

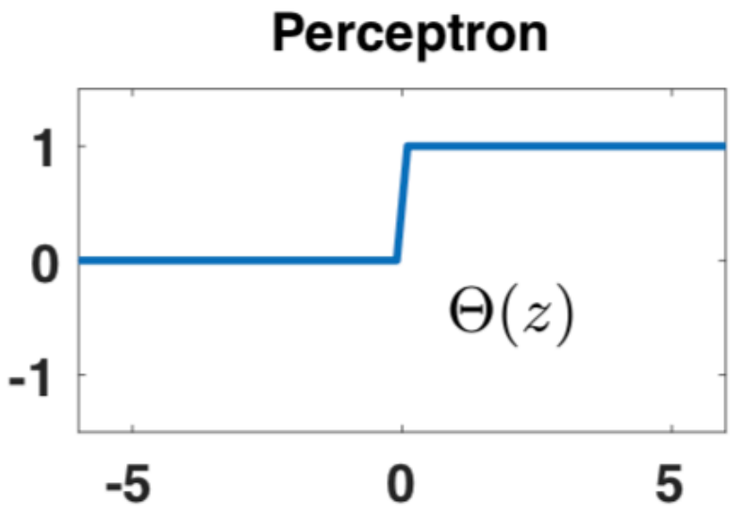


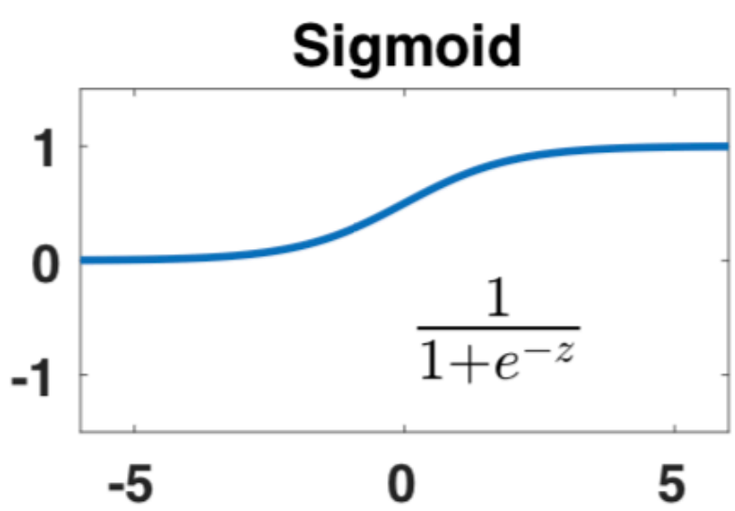
the parameters of NNs are often called ``weights" and ``thresholds"

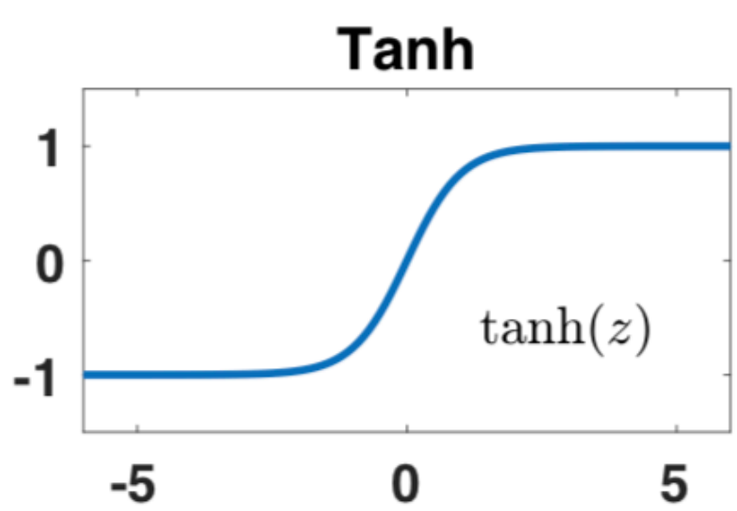
the choice of non-linear activation function affects the **computational and training properties** of the neural nets, since they modify the output gradients required for GD training

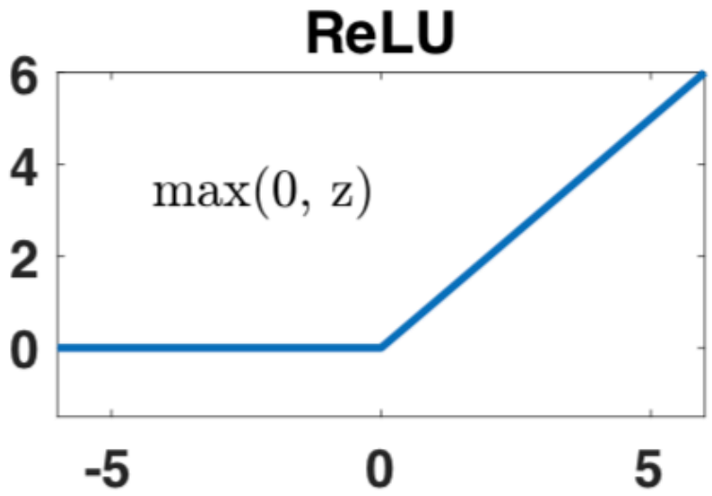
NN nonlinearly only via activation functions!

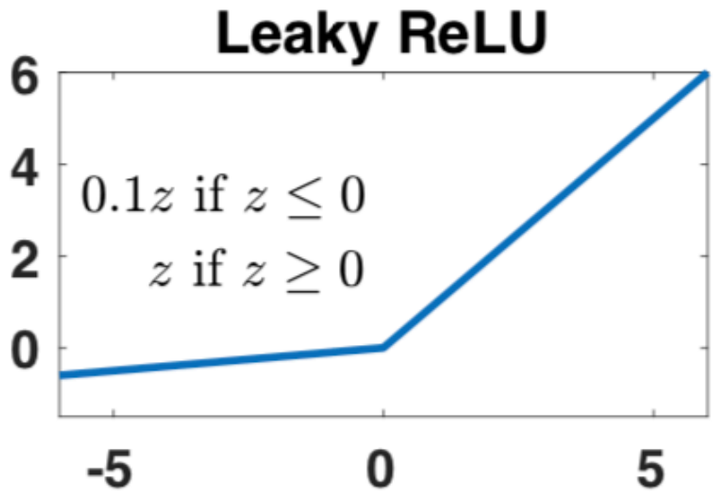
Activation functions

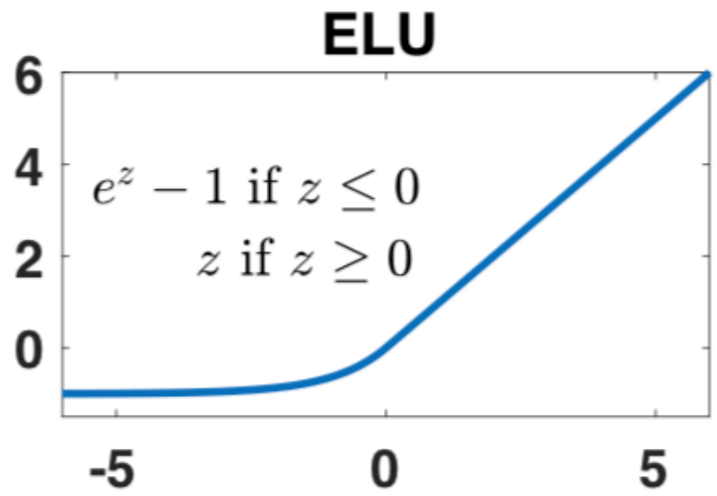






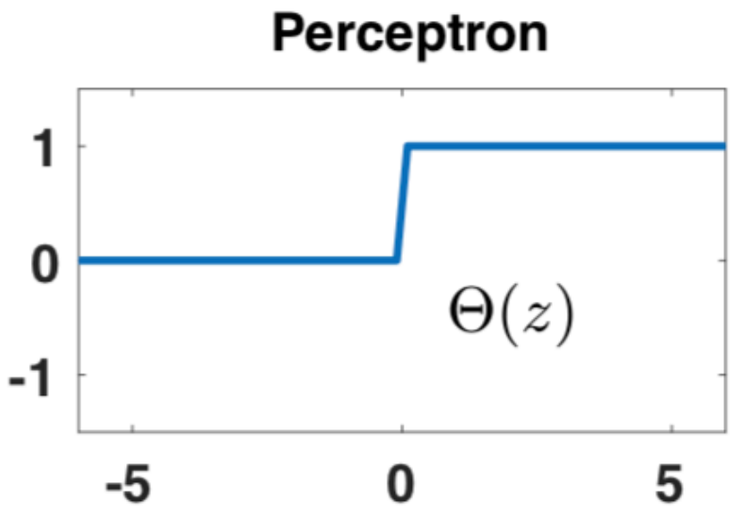


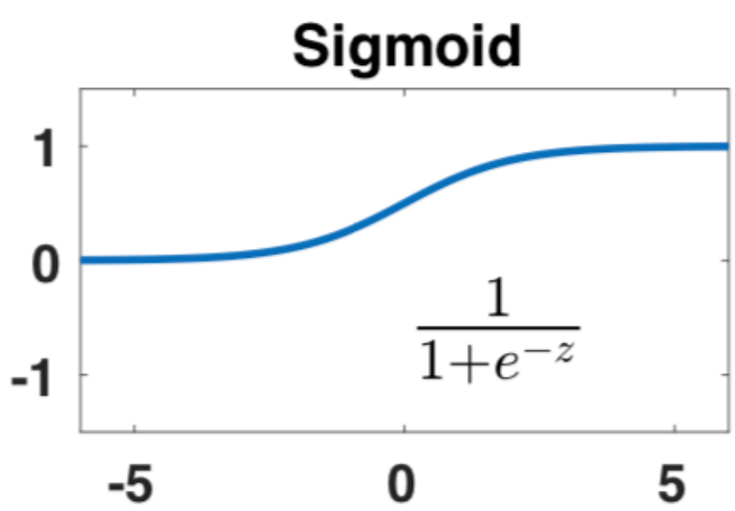


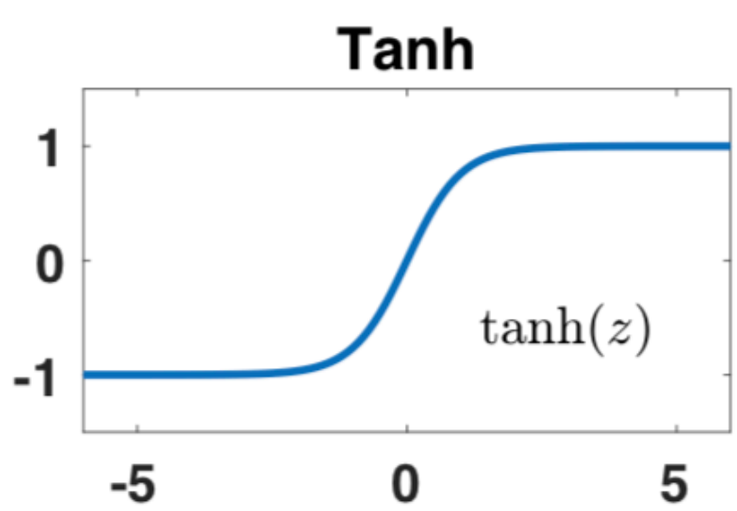


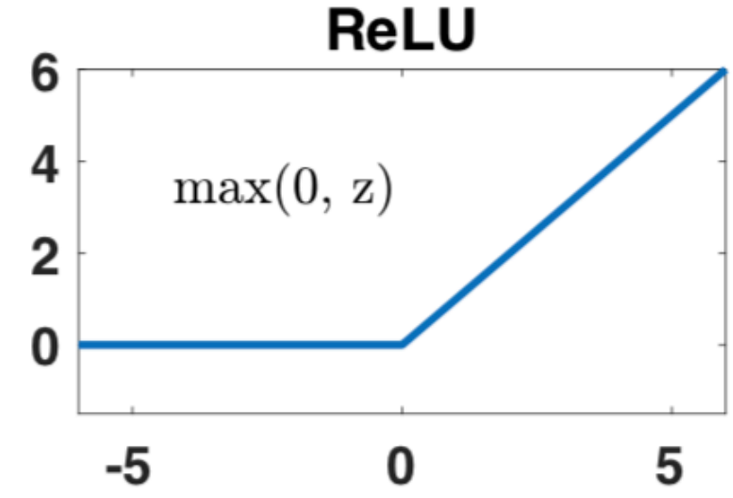
what is the main difference between these various activation functions?

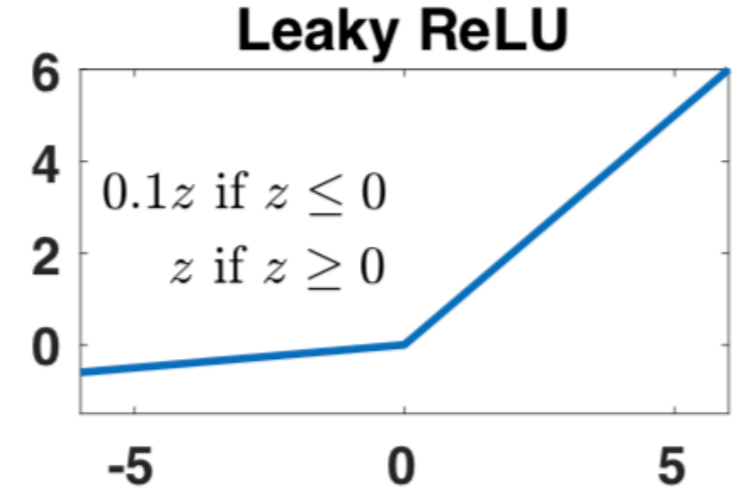
Activation functions

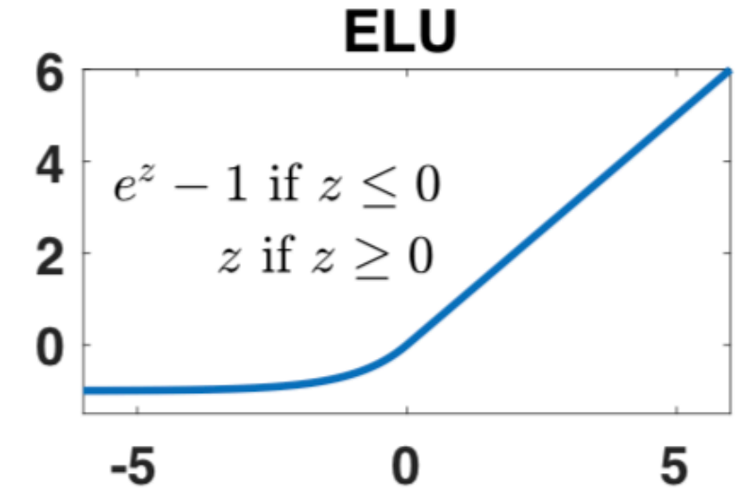












- Activation functions can be classified between those that **saturate at large inputs** (*e.g.* sigmoid) and those that **do not saturate at large inputs** (*e.g.* Rectified Linear Units ReLU)

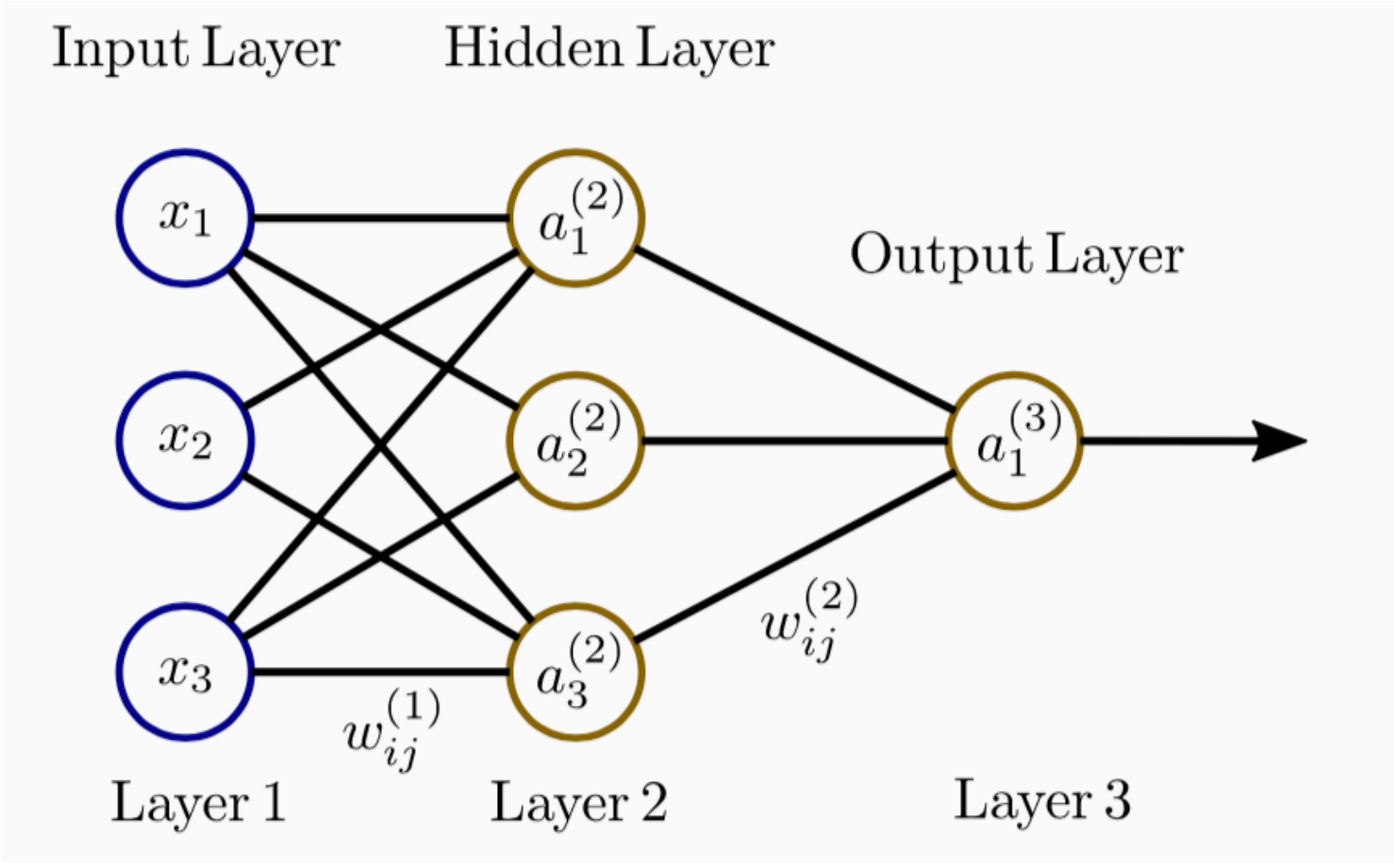
- The choice of non-linearities has important implications for **GD NN training methods:**

sigmoid → $\left.\frac{d\sigma}{dz}\right|_{z\gg 1} = 0$ ***ReLU*** → $\left.\frac{d\sigma}{dz}\right|_{z\gg 1} \neq 0$

vanishing gradients are problematic for deep networks: loss of sensitivity

A simple network

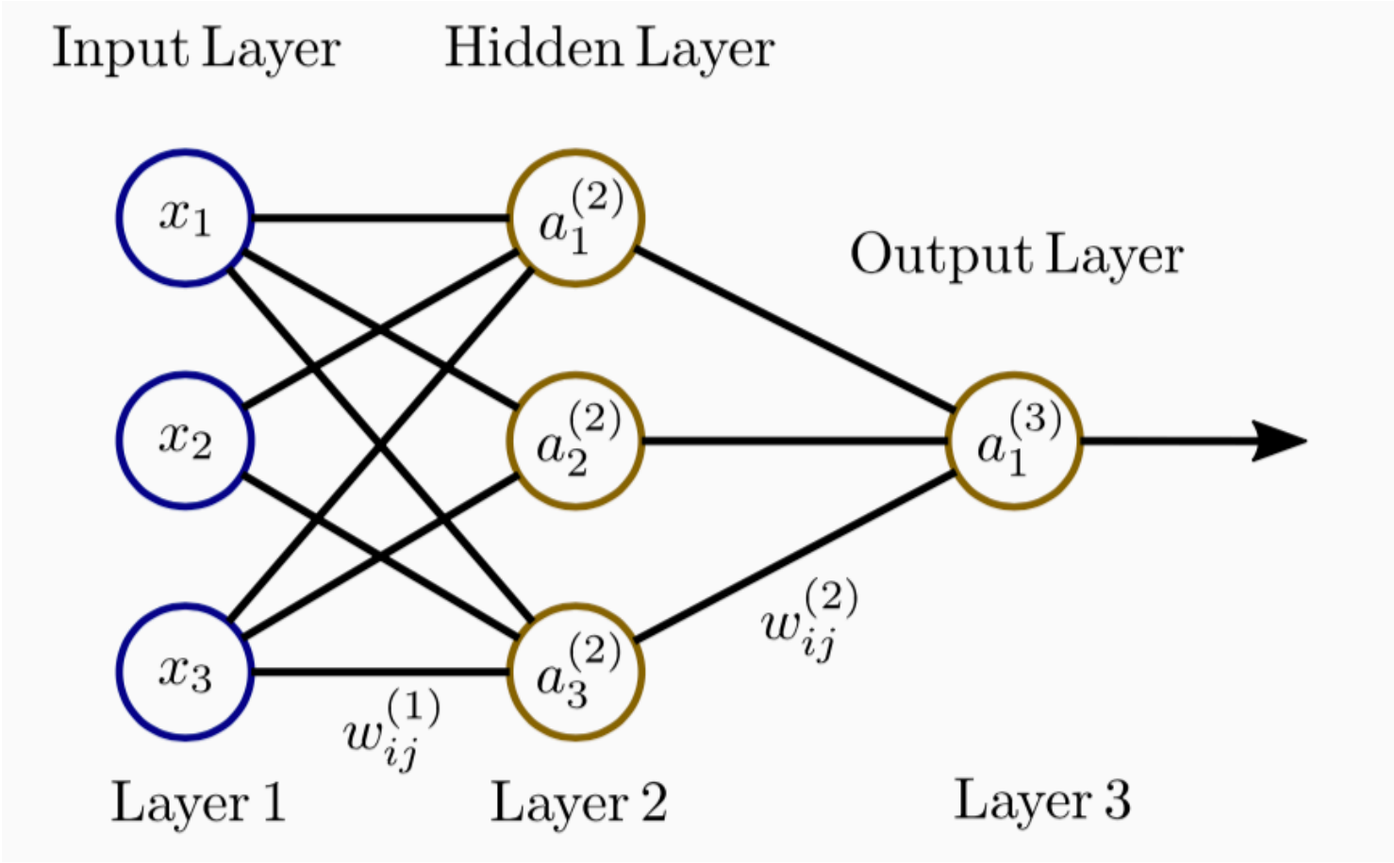
to realise that NNs are nothing mysterious but just a **complex non-linear mapping** between inputs and outputs, consider the explicit calculation of the output of a simple network



can we evaluate analytically $a_1^{(3)}$ as function of x_1, x_2, x_3?

A simple network

to realise that NNs are nothing mysterious but just a **complex non-linear mapping** between inputs and outputs, consider the explicit calculation of the output of a simple network



using the three inputs (Layer 1), compute activation states of neurons in Layer 2

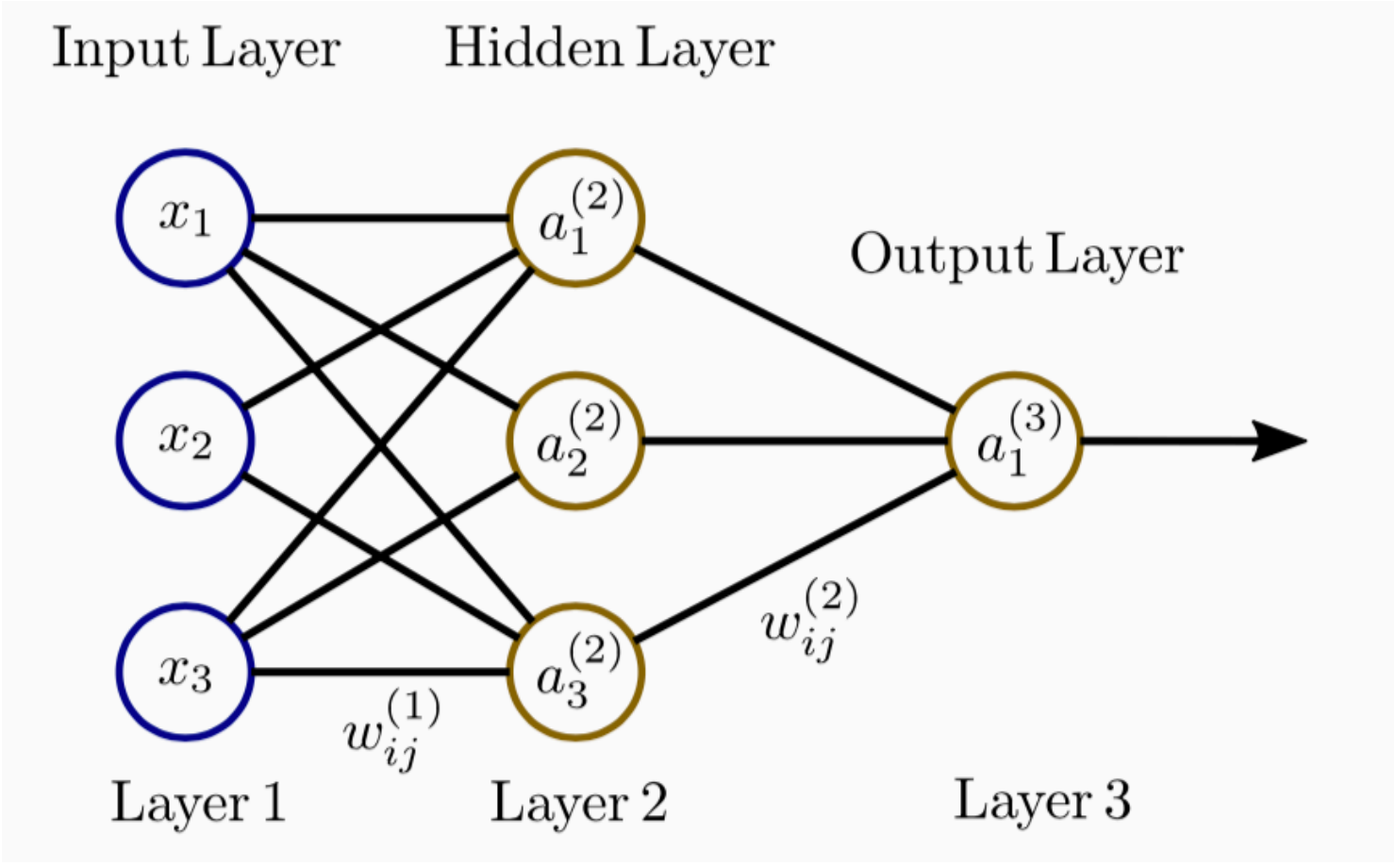
$$a_1^{(2)} = \sigma\left(x_1\theta_{11}^{(1)} + x_2\theta_{12}^{(1)} + x_3\theta_{13}^{(1)} + \theta_{10}^{(1)}\right)$$

$$a_2^{(2)} = \sigma\left(x_1\theta_{21}^{(1)} + x_2\theta_{22}^{(1)} + x_3\theta_{23}^{(1)} + \theta_{20}^{(1)}\right)$$

$$a_2^{(3)} = \sigma\left(x_1\theta_{31}^{(1)} + x_2\theta_{32}^{(1)} + x_3\theta_{33}^{(1)} + \theta_{30}^{(1)}\right)$$

A simple network

to realise that NNs are nothing mysterious but just a **complex non-linear mapping** between inputs and outputs, consider the explicit calculation of the output of a simple network



using the activation states of neurons in Layer 2, compute activation state of output neuron

$$a_1^{(3)} = \sigma\left(a_1^{(2)}\theta_{11}^{(2)} + a_2^{(2)}\theta_{12}^{(2)} + a_3^{(2)}\theta_{13}^{(2)} + \theta_{10}^{(2)}\right)$$

NN output is **analytical function** of the inputs and the model parameters

A simple network

to realise that NNs are nothing mysterious but just a **complex non-linear mapping** between inputs and outputs, consider the explicit calculation of the output of a simple network

$$a_1^{(3)} = \sigma\Bigg(\sigma\left(x_1\theta_{11}^{(1)} + x_2\theta_{12}^{(1)} + x_3\theta_{13}^{(1)} + \theta_{10}^{(1)}\right)\theta_{11}^{(2)} +$$

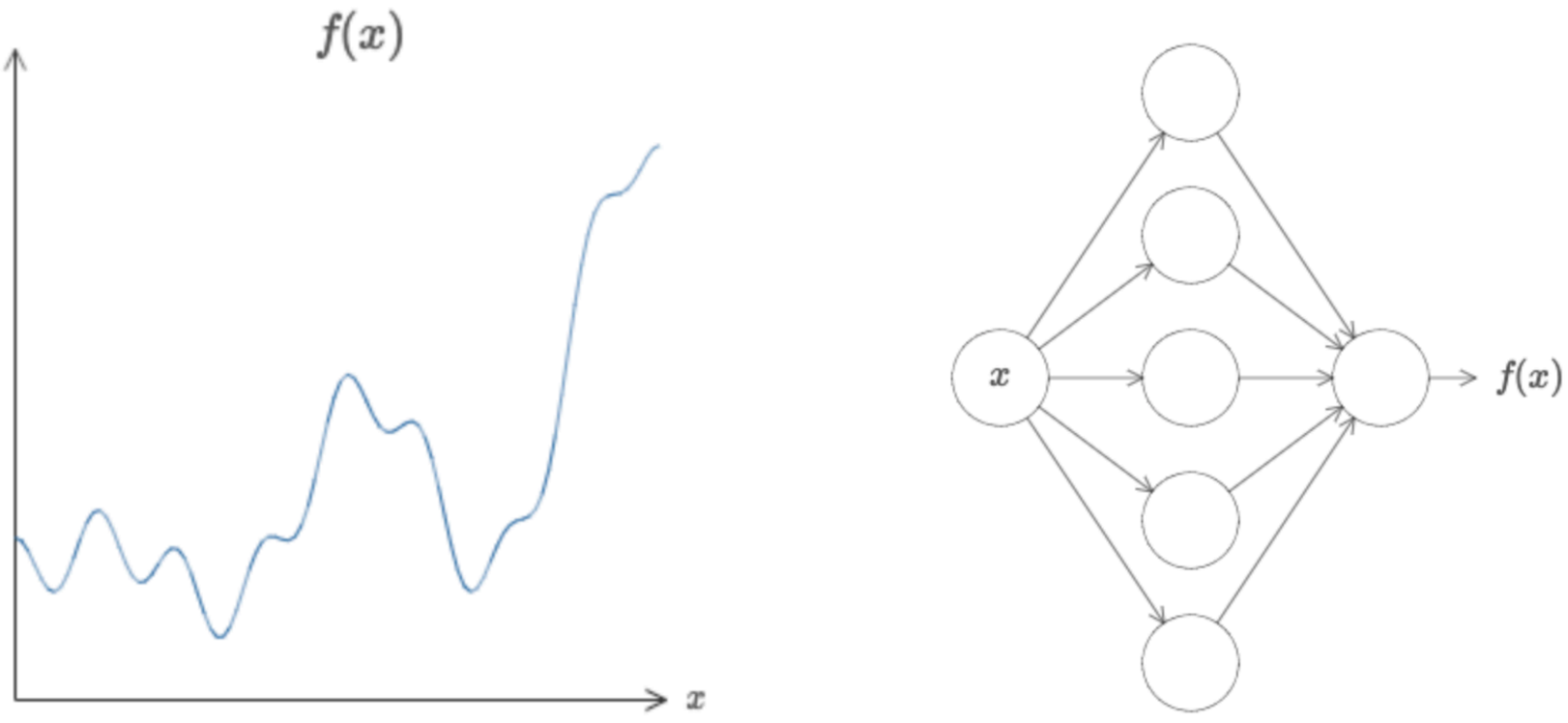
$$\sigma\left(x_1\theta_{21}^{(1)} + x_2\theta_{22}^{(1)} + x_3\theta_{23}^{(1)} + \theta_{20}^{(1)}\right)\theta_{12}^{(2)} +$$

$$\sigma\left(x_1\theta_{31}^{(1)} + x_2\theta_{32}^{(1)} + x_3\theta_{33}^{(1)} + \theta_{30}^{(1)}\right)\theta_{13}^{(2)} + \theta_{10}^{(2)}\Bigg)$$

substitute the activation function *e.g.* for the sigmoid and you have the analytic output of a simple NN

The Universal Approximation Theorem

theorem: a neural network with a single hidden layer and enough neurones can **approximate any continuous, multi-input/multi-output function** with arbitrary accuracy



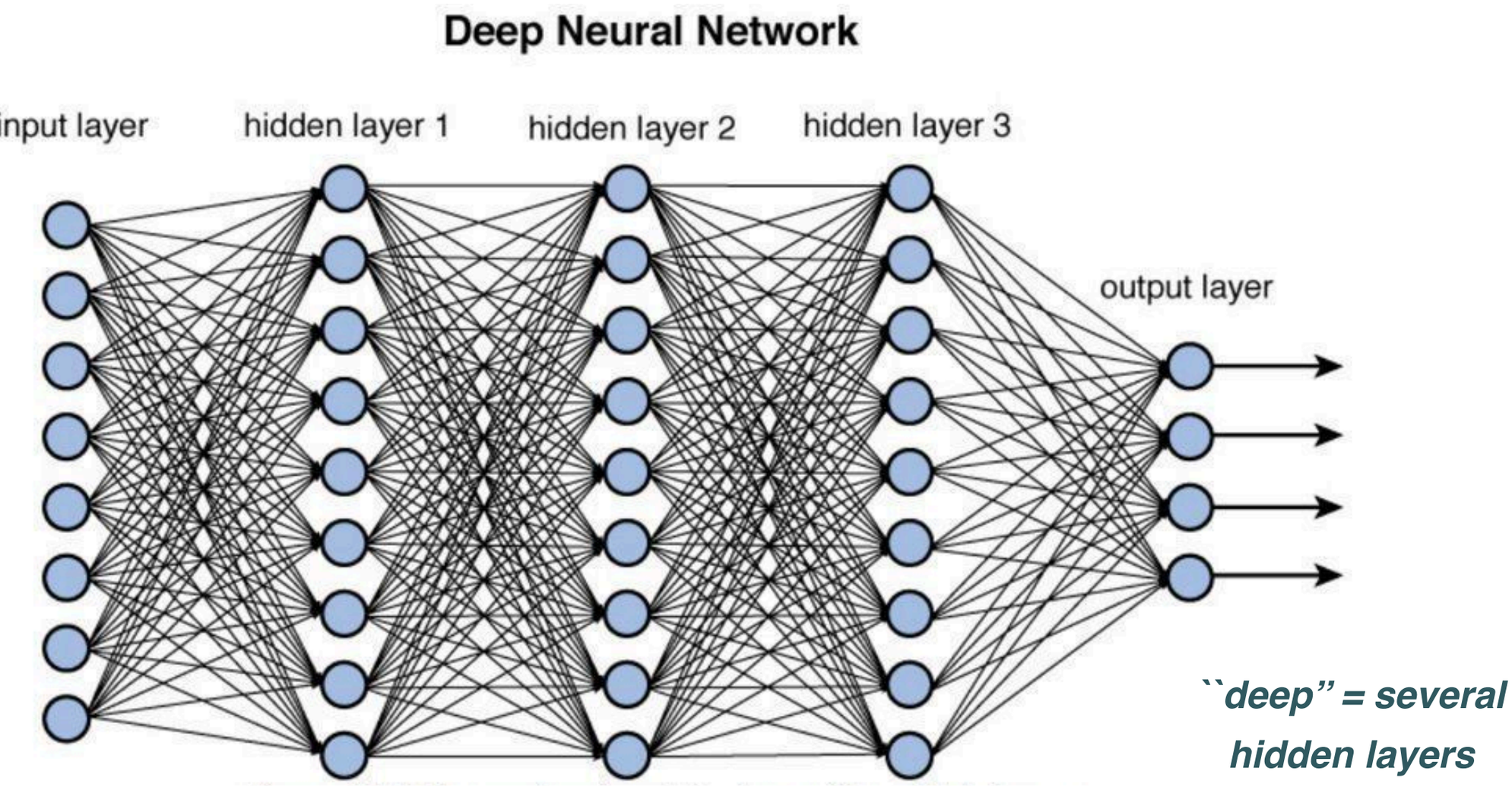
neural networks exhibit *universality properties:* no matter what function we want to compute, we know (theorem!) that there is a neural network which can carry out this task

See M. Nielsen, *Neural Networks and Deep Learning*: http://neuralnetworksanddeeplearning.com/chap4.html

Deep Networks

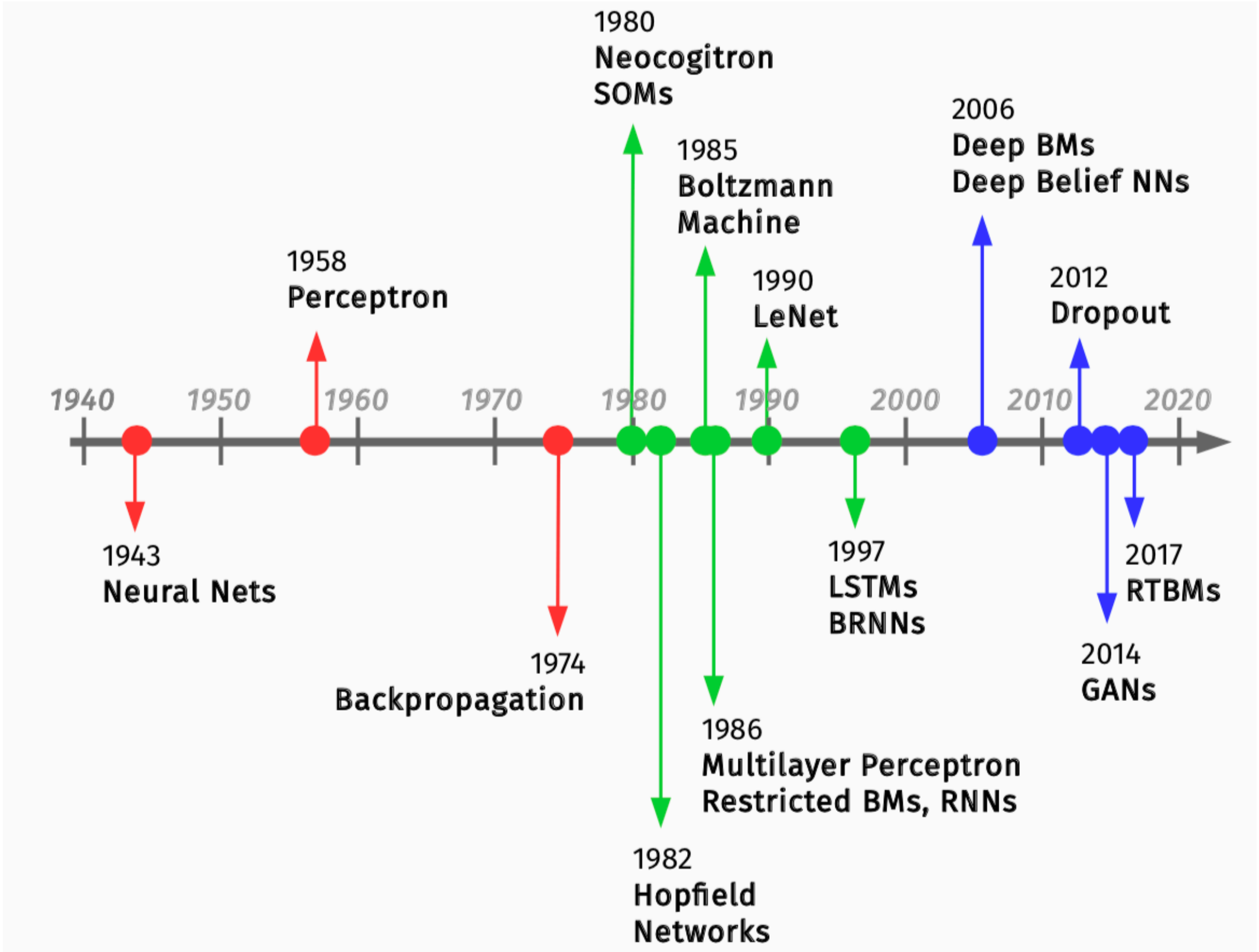
A neural network can thus be thought of a **complicated non-linear mapping** between the inputs and the outputs that depends on the parameters (weights and bias) of each neuron

We can make a NN **deep** by adding hidden layers, which greatly expands their **representational power**, also known as **expressivity**



``deep'' = several hidden layers

A timeline of neural networks



NN training

As standard in **Supervised Learning**, the first step to train a NN is to specify a **cost function**

$$\left(\boldsymbol{x}_i, y_i\right) \ , \quad i = 1,\ldots,n \longrightarrow \hat{y}_i\left(\boldsymbol{\theta}\right) \ , \quad i = 1,\ldots,n$$

for each of the n data points … *… the output of the NN provides the model prediction*

the loss function depends on whether the NN should provide **continuous or categorical** (discrete) predictions. For continuous data we can have the **mean square error**

$$E(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_i\left(\boldsymbol{\theta}\right)\right)^2$$

for categorical data we use the **cross-entropy**, which for binary (true/false) classification is

$$E(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i \ln \hat{y}_i(\boldsymbol{\theta}) + (1 - y_i)\ln(1 - \hat{y}_i(\boldsymbol{\theta}))\right)$$

where the true labels satisfy $y_i \in \{0,1\}$

NN training are based on a specific version of GD methods: **backpropagation**

Backpropagation

For deep NNs the brute-force evaluation of the **gradients of the cost function** is impractical

GD method:

$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

learning rate

gradient of cost function

*update of model parameters between iterations **t** and **t+1***

Why? What prevents us from using simple GD to train NNs?

Backpropagation

For deep NNs the brute-force evaluation of the **gradients of the cost function** is impractical

GD method:

$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

learning rate

gradient of cost function

*update of model parameters between iterations **t** and **t+1***

instead one uses **backpropagation**, which cleverly exploits the **layered structure** of NNs

we will use the following notation:

- A neural network with ***L* layers**, labelled as *l=1,…,L*
- $\omega_{jk}^{(l)}$**: weights** connecting *k*-th neuron in the *(l-1)*-th layer to *j*-th neutron in the *l*-th layer
- $b_j^{(l)}$ **: bias** of the *j*-th neuron in the *l*-th layer
- $a_j^{(l)}$**: activation state** of the *j*-th neuron in the *l*-th layer

Backpropagation

feed-forward NNs: the activation state of the *j*-th neuron in the *l*-th layer is a (nonlinear) function of the activation states of all the neutrons in the *(l-1)*-th layer

$$a_j^{(l)} = \sigma\left(\sum_{k=1}^{n_{l-1}} \omega_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}\right) \equiv \sigma(z_j^{(l)}), \quad j = 1,\ldots,n_l$$

the cost function of the NN therefore depends on:

- $a_j^{(L)}$: **activation states** of the neurons on the **output layer *L*** (directly)
- $a_j^{(l)}$: **activation states** of the neurons on the hidden layers ***l* <*L*** (indirectly)

we define the **error** associated to the *j*-th neuron in the output layer and hidden layers as

$$\Delta_j^{(L)} \equiv \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(L)}} \qquad \Delta_j^{(l)} \equiv \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(l)}} = \frac{\partial E(\boldsymbol{\theta})}{\partial a_j^{(l)}} \frac{da_j^{(l)}}{dz_j^{(l)}}$$

here assume that output layer is linear: ensures that model predictions are unbounded

Backpropagation

$$\Delta_j^{(L)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(L)}} \qquad \Delta_j^{(l)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(l)}} = \frac{\partial E(\boldsymbol{\theta})}{\partial a_j^{(l)}} \sigma'\left(z_j^{(l)}\right)$$

to derive the backpropagation equations, we will use the chain rule & NN layer structure

$$\Delta_j^{(l)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(l)}} = \sum_{k=1}^{n_{l+1}} \frac{\partial E(\boldsymbol{\theta})}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_{k=1}^{n_{l+1}} \Delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$$

where here we exploit the **NN layered structure:** activation states of neurons in *(l+1)*-th layer depends only on activation states of neurons in *l*-th layer

$$\Delta_j^{(l)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(l)}} = \sum_{k=1}^{n_{l+1}} \Delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \left(\sum_{k}^{n_{l+1}} \Delta_k^{(l+1)} \omega_{kj}^{(l+1)} \right) \sigma'\left(z_j^{(l)}\right)$$

Backpropagation

$$\Delta_j^{(L)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(L)}} \qquad \Delta_j^{(l)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(l)}} = \frac{\partial E(\boldsymbol{\theta})}{\partial a_j^{(l)}} \sigma'\left(z_j^{(l)}\right)$$

to derive the backpropagation equations, we will use the chain rule & NN layer structure

$$\Delta_j^{(l)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(l)}} = \sum_{k=1}^{n_{l+1}} \frac{\partial E(\boldsymbol{\theta})}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_{k=1}^{n_{l+1}} \Delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$$

where here we exploit the **NN layered structure:** activation states of neurons in *(l+1)*-th layer depends only on activation states of neurons in *l*-th layer

$$\Delta_j^{(l)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(l)}} = \sum_{k=1}^{n_{l+1}} \Delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \left(\sum_{k}^{n_{l+1}} \Delta_k^{(l+1)} \omega_{kj}^{(l+1)} \right) \sigma'\left(z_j^{(l)}\right)$$

$$z_k^{(l+1)} = \sum_{k'=1}^{n_l} \omega_{jk'}^{(l+1)} a_{k'}^{(l)} + b_j^{(l+1)} = \sum_{k'=1}^{n_l} \omega_{jk'} \sigma(z_{k'}^{l}) + b_j^{(l+1)}$$

Backpropagation

$$\Delta_j^{(L)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(L)}} \qquad \Delta_j^{(l)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(l)}} = \frac{\partial E(\boldsymbol{\theta})}{\partial a_j^{(l)}} \sigma'\left(z_j^{(l)}\right)$$

to derive the backpropagation equations, we will use the chain rule & NN layer structure

$$\Delta_j^{(l)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(l)}} = \sum_{k=1}^{n_{l+1}} \frac{\partial E(\boldsymbol{\theta})}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_{k=1}^{n_{l+1}} \Delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$$

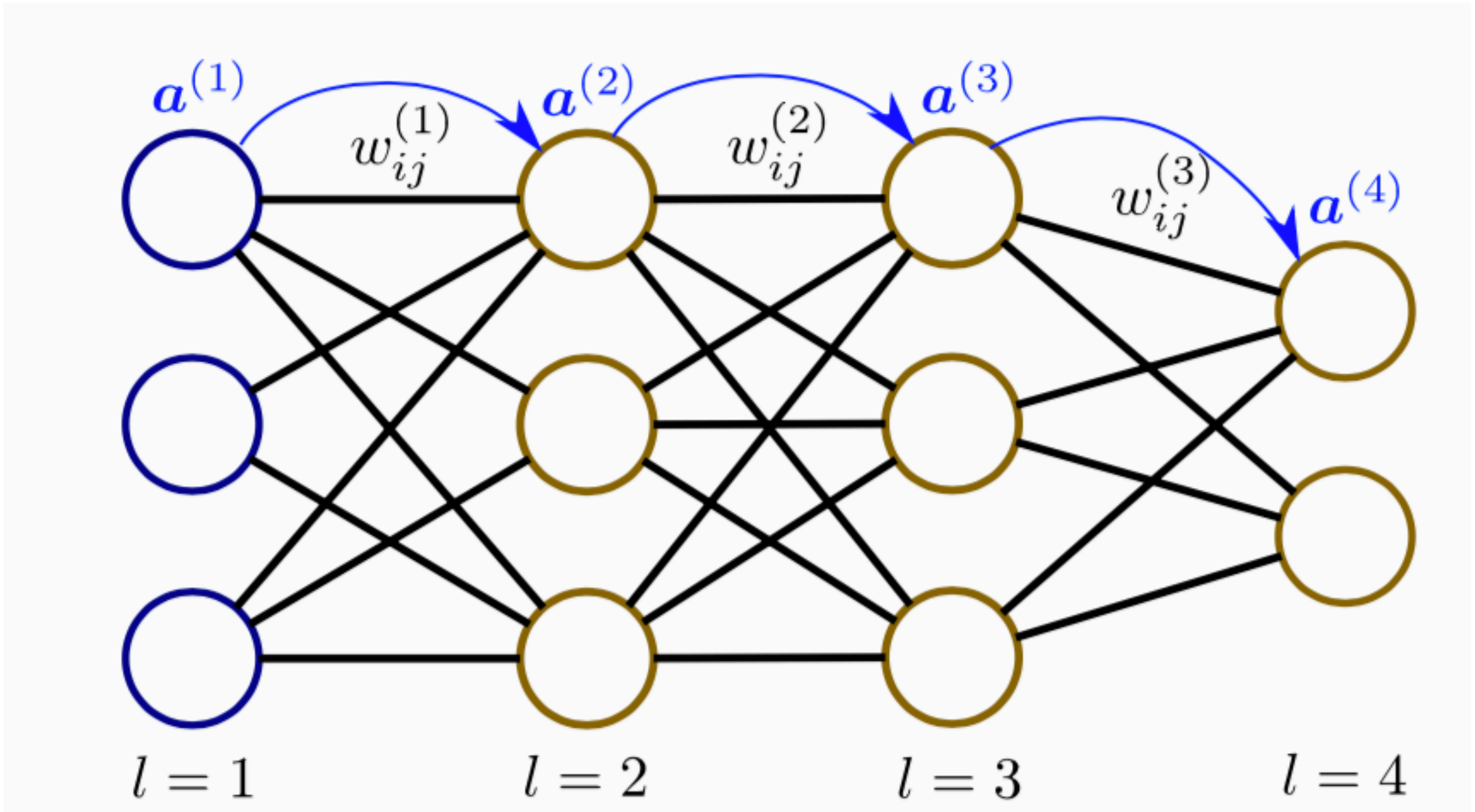
where here we exploit the **NN layered structure:** activation states of neurons in *(l+1)*-th layer depends only on activation states of neurons in *l*-th layer

$$\Delta_j^{(l)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(l)}} = \sum_{k=1}^{n_{l+1}} \Delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \left(\sum_{k}^{n_{l+1}} \Delta_k^{(l+1)} \omega_{kj}^{(l+1)} \right) \sigma'\left(z_j^{(l)}\right)$$

$$\frac{\partial E}{\partial \omega_{jk}^{(l)}} = \frac{\partial E}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial \omega_{jk}^{(l)}} = \Delta_j^{(l)} a_k^{(l-1)}$$

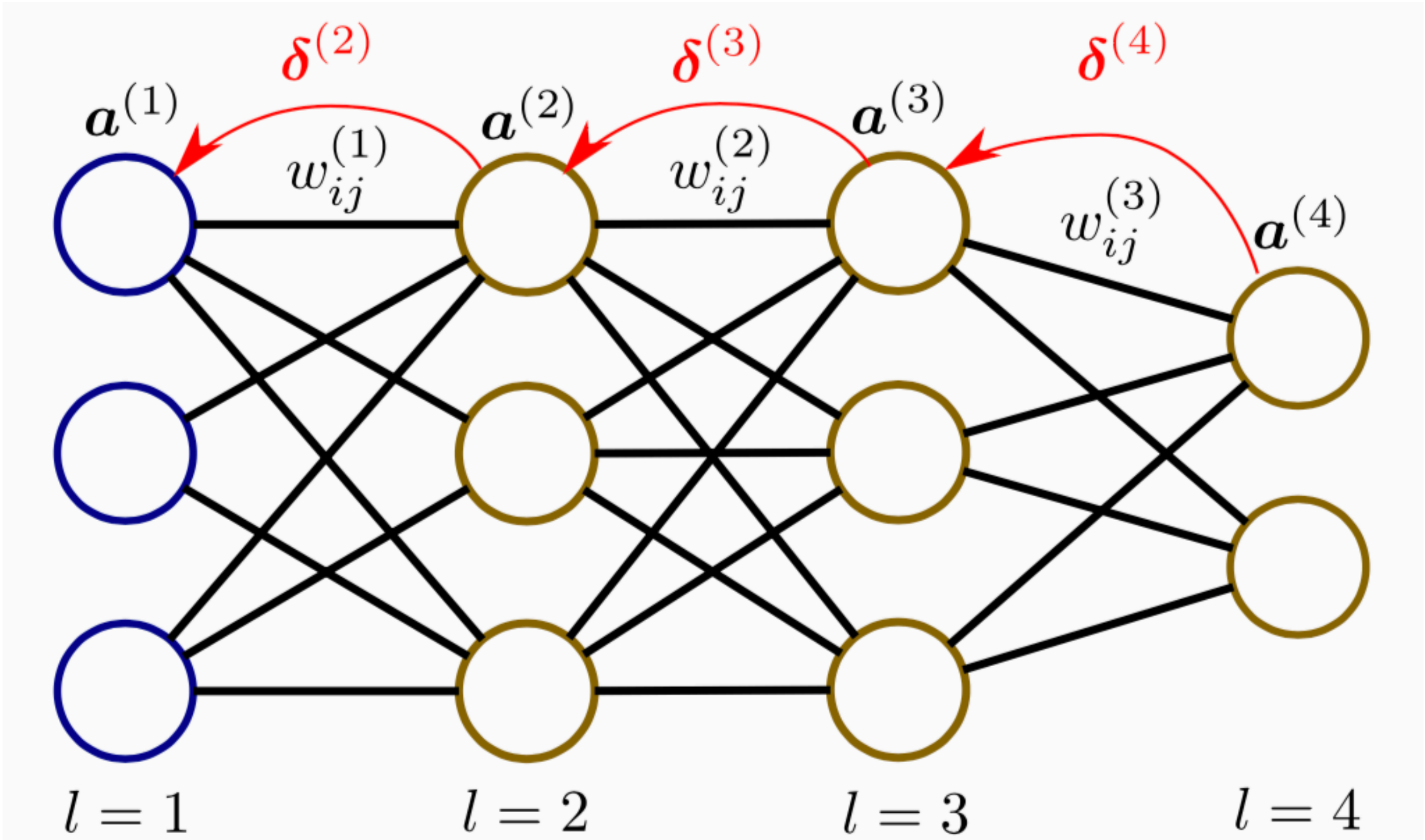
Backpropagation

first evaluate the activation states of all neurons using **forward propagation …**



Backpropagation

then **backpropagate** to evaluate the **errors Δ** and thus the gradients over model parameters



Backpropagation

we now have all the ingredients to deploy the **backpropagation algorithm for NN training**

- ***(1)*** Evaluate activation state of neurons in input layer $a_j^{(1)}, \quad j = 1,\ldots,n_1$

- ***(2)*** **Feed-forward:** evaluate activation states for each subsequent layer

$$a_j^{(l)} = \sigma\left(\sum_{k=1}^{n_l} \omega_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}\right) \equiv \sigma(z_j^{(l)}) \quad , l = 2,\ldots,L$$

- ***(3)*** With this information evaluate error on network's output layer

$$\Delta_j^{(L)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(L)}}$$

here enters dependence on cost function

- ***(4)*** **Backpropagate** this error to evaluate the errors on all hidden layers

$$\Delta_j^{(l)} = \left(\sum_{k}^{n_{l+1}} \Delta_k^{(l+1)} \omega_{kj}^{(l+1)}\right) \sigma'\left(z_j^{(l)}\right)$$

error in j-th neuron in (l)-th layer

error in k-th neuron in (l+1)-th layer

Backpropagation

we now have all the ingredients to deploy the **backpropagation algorithm for NN training**

- ***(5)*** Evaluate **gradients of the model parameters** associated to the gradient of cost function

$$\frac{\partial E}{\partial \omega_{jk}^{(l)}} = \Delta_j^{(l)} a_k^{(l-1)} \qquad \frac{\partial E}{\partial b_j^{(l)}} = \Delta_j^{(l)}$$

extremely efficient way of calculating the gradients of the model parameters, requiring only a **single forward and backward pass** of the neural network

$$\Delta_j^{(L)} = \frac{\partial E(\boldsymbol{\theta})}{\partial z_j^{(L)}}$$

at most j numerical derivatives need to be evaluated, even in NNs with millions of parameters

Backpropagation

we now have all the ingredients to deploy the **backpropagation algorithm for NN training**

- ***(5)*** Evaluate **gradients of the model parameters** associated to the gradient of cost function

$$\frac{\partial E}{\partial \omega_{jk}^{(l)}} = \Delta_j^{(l)} a_k^{(l-1)} \qquad \frac{\partial E}{\partial b_j^{(l)}} = \Delta_j^{(l)}$$

extremely efficient way of calculating the gradients of the model parameters, requiring only a **single forward and backward pass** of the neural network

with this info one can carry out with Gradient Descent and **update the model parameters**

$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)\,, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

note that even with BP training of large (deep) networks can be **computationally intensive!**

Further complication if cost function **depends non-trivially on NN output**

$$E(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \left(\mathscr{F}_i^{(\mathrm{dat})} - \mathscr{F}_i^{(\mathrm{model})}(\hat{\boldsymbol{y}}; \boldsymbol{\theta}) \right)^2$$

e.g. model requires integral of NN output

NN initialisation

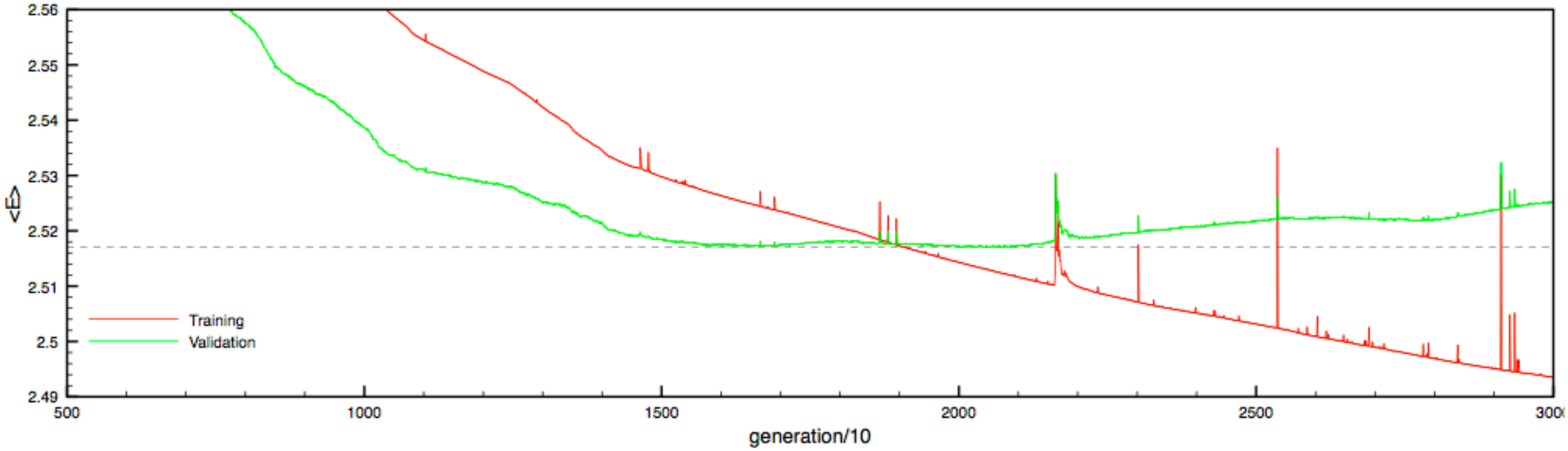
A further subtlety concerning NN training is that sometimes **a clever initialisation of the model parameters** helps in the learning process

- **zero:** all weights set to zero (initial complexity equivalent to single neuron)
- **random:** breaks parameter symmetry
- **glorot/xavier:** weights distributed randomly with Gaussian with variance based on in/out size of the neutron
- **he:** random initialisation avoiding the saturation region of the activation function

in many cases trial-and-error required to assess what works best

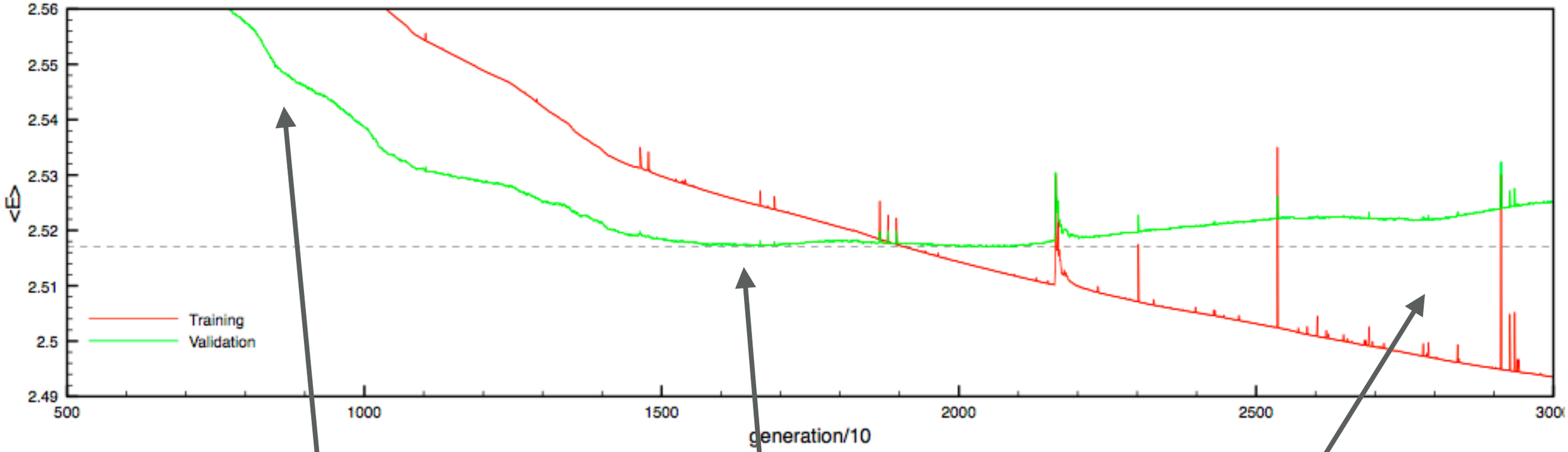
NN regularisation: cross-validation

Neural networks provide extremely flexible models to describe complex datasets, but one should **avoid overfitting**, else the model will be **unable to generalise**



NN regularisation: cross-validation

Neural networks provide extremely flexible models to describe complex datasets, but one should **avoid overfitting**, else the model will be **unable to generalise**



underlearning

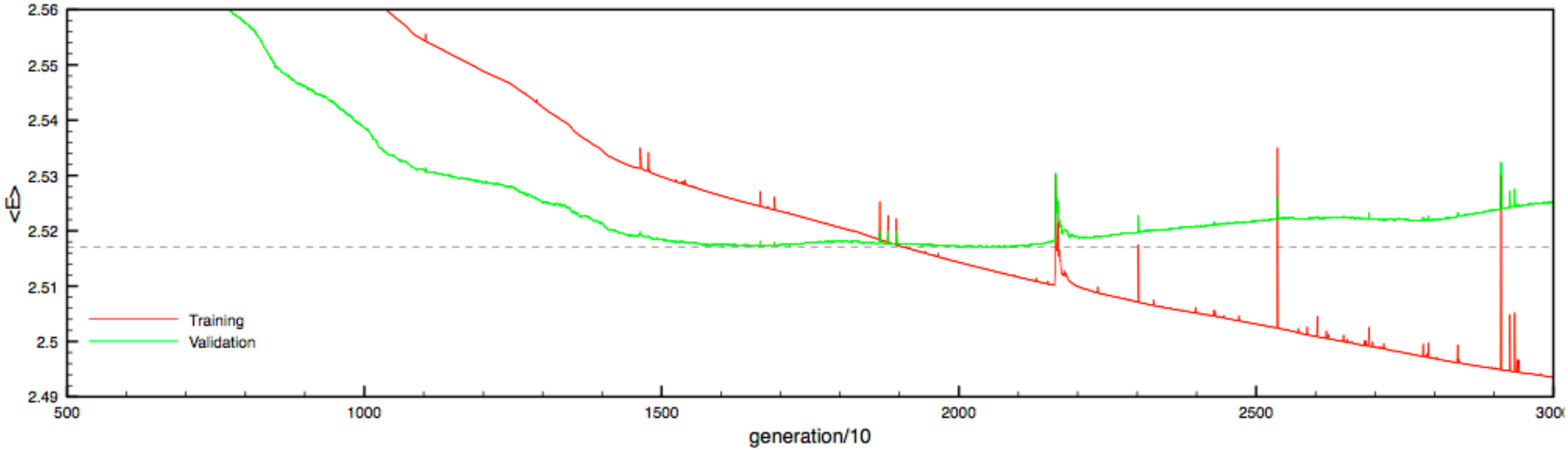
Optimal stopping

overfitting

recall the bias/variance tradeoff!

NN regularisation: cross-validation

Neural networks provide extremely flexible models to describe complex datasets, but one should **avoid overfitting**, else the model will be **unable to generalise**



Cross-validation look-back stopping: train the NN for a large fixed number of iterations. Then look back and determine the point at the training where the out-of-sample error was the smallest. Take as best-fit NN parameters those corresponding to that point of the training

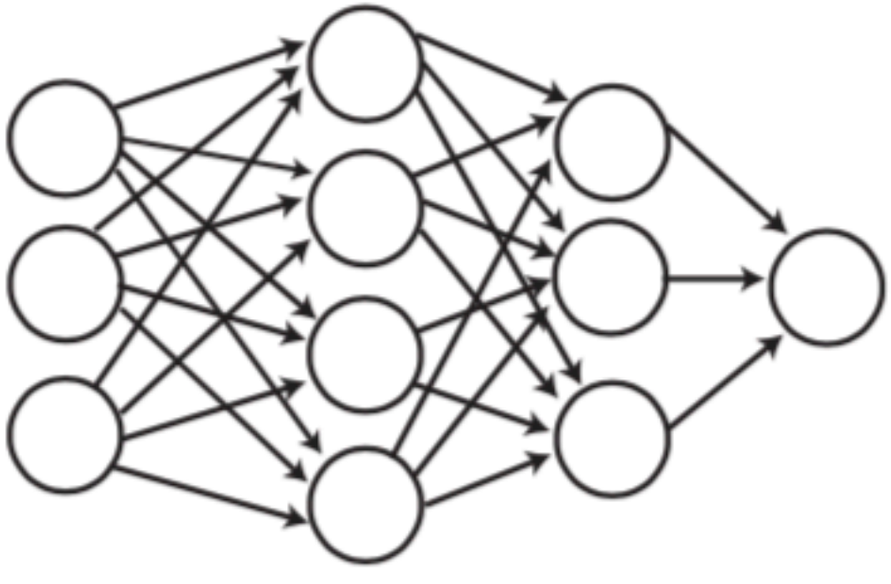
NN regularisation: Dropout

- Dropout is one of the standard **regularisation procedures** that aim to avoid overfitting when training deep NNs

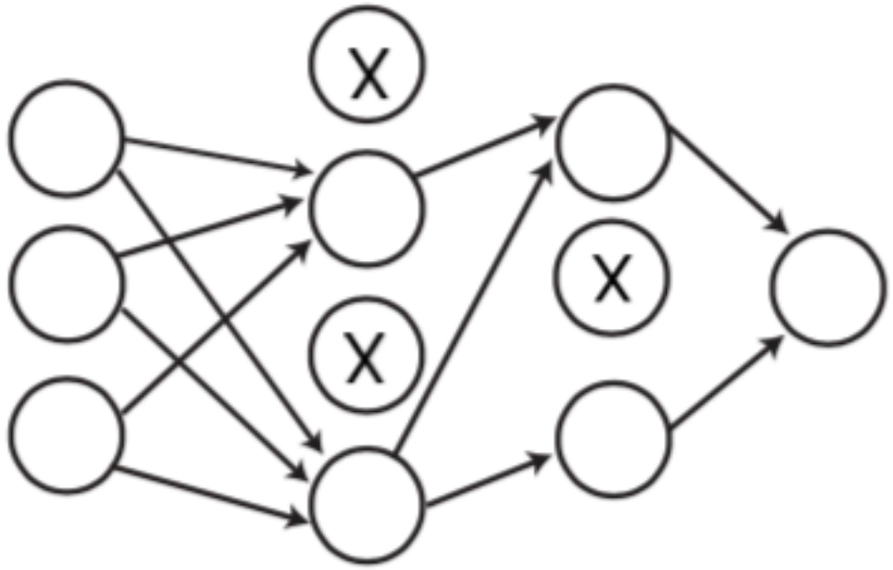
- During the training procedure, neurons are randomly "**dropped out**" of the neural network with some probability *p* giving rise to a thinned network, and the GD gradients are computed only there

- Dropout prevents overfitting by **reducing correlations among neurons** and reducing the variance

Standard Neural Net



After applying Dropout



effective reduction of model complexity

Hyperoptimisation

In most Machine Learning applications, the model has several parameters which are typically **adjusted by hand** (trial and error) rather than algorithmically:

- Network architecture: number of layers of neurons per layer, activation functions, …
- Choice of minimiser (which of the GD variants?)
- Learning rate, momentum, memory, size of mini-batches, ….
- Regulariation parameters, stopping, dropout rate, patience, …

one can avoid the need of subjective choice by means of **an hyperoptimisation procedure**, where all model and training/stopping parameters are determined algorithmically

Such hyperoptimisation requires introducing a **reward function** to grade the model. Note that this is different from the **cost function:** the latter is optimised separately model by model (e.g. for each NN architecture) while the former compares between all optimised models

e.g. cost function $C = E_{\text{tr}}$ $R = E_{\text{val}}$ ***reward function***

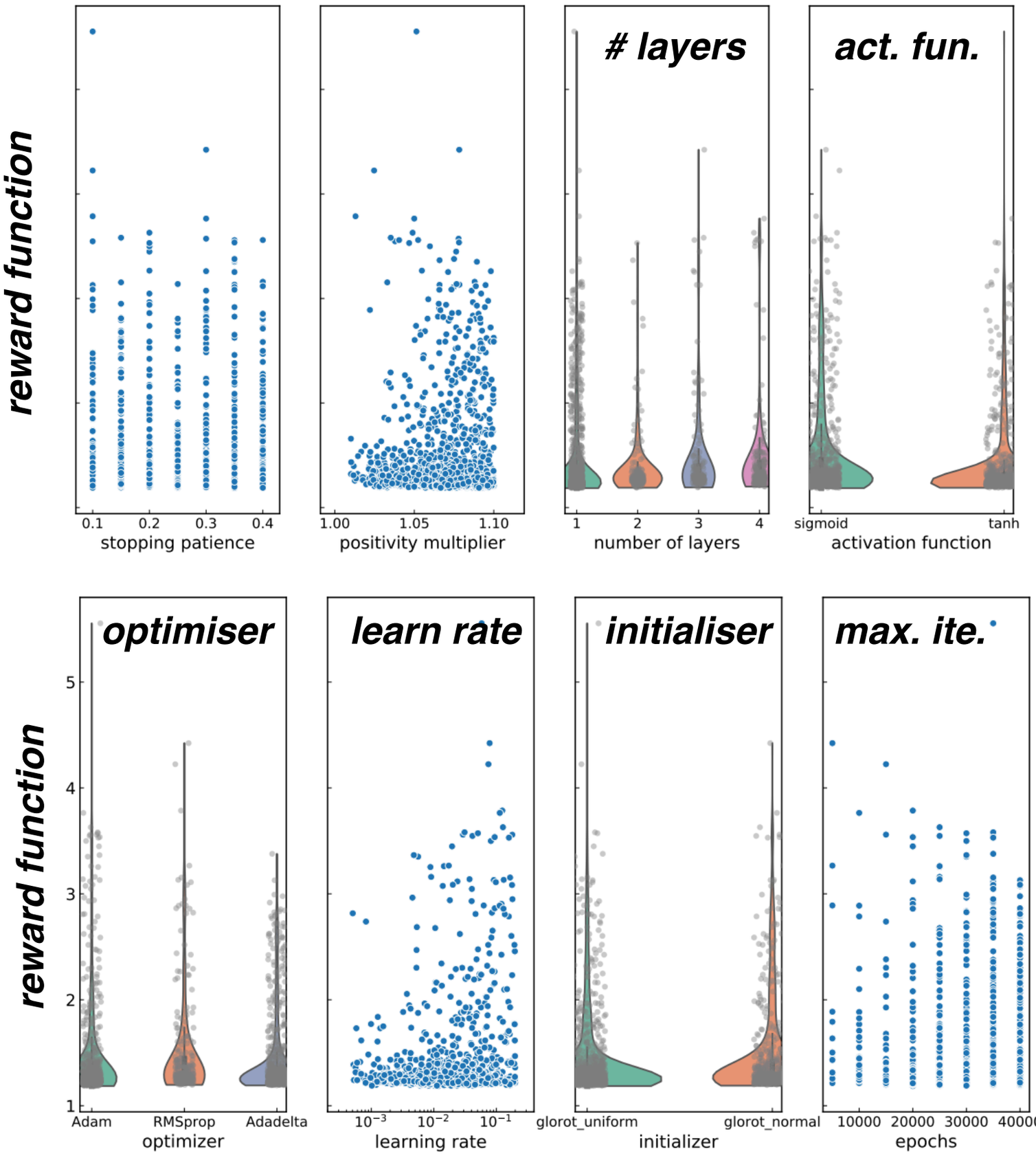
Hyperoptimisation

- In a hyperparameter scan one can compare the performance of **hundreds or thousands** of parameter combinations

- Some choices are **discrete** (type of minimiser, # of layers) others are **continuous** (learning rate)

- One can also **visualise** which choices are more crucial and which ones less important

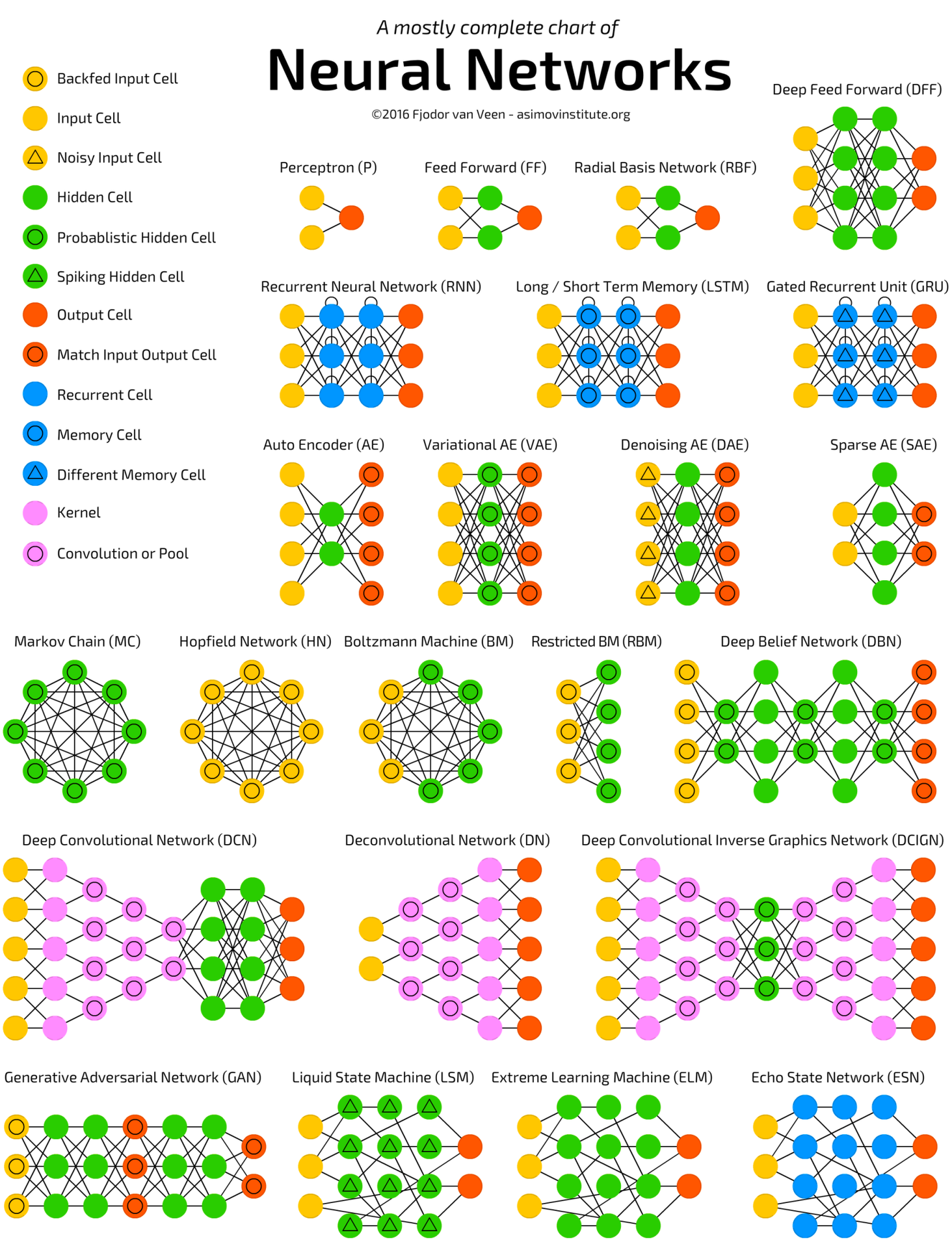
- The violin plots are the **KDE-reconstructed probability distributions** for the hyper-parameters



- A **large variety of neural network architectures** have been proposed: we will only study some of them in this course

- They differ in: number of layers and neurons, role of the neurons, connections between neurons and layers, ….

- Each architecture in general has associated **different training and regularisation strategies**: no fit-for-all methods available!



Tutorial 2: NN training with TensorFlow

☑ Introduction to **non-linear regression problems** using neural networks.

☑ You will learn how to **build and train neural networks** with TensorFlow, a powerful machine learning library.

☑ The goal will be to extract a certain physical property of protons, the distribution of momentum carried by their gluons, **from simulated pseudo-data** based on some known underlying truth

☑ The neural nets should then aim to l**earn this known underlying truth** from the data

☑ **Assess/avoid overfitting** and how to **estimate the model uncertainties**, e.g. What happens when different sets of NN parameters give an equally good description of the *experiment* data?