Artificial Neural Networks and the Multi-Layer Perceptron

Alasdair Newson

Equipe IMAGES - Télécom Paris
alasdair.newson@telecom-paris.fr

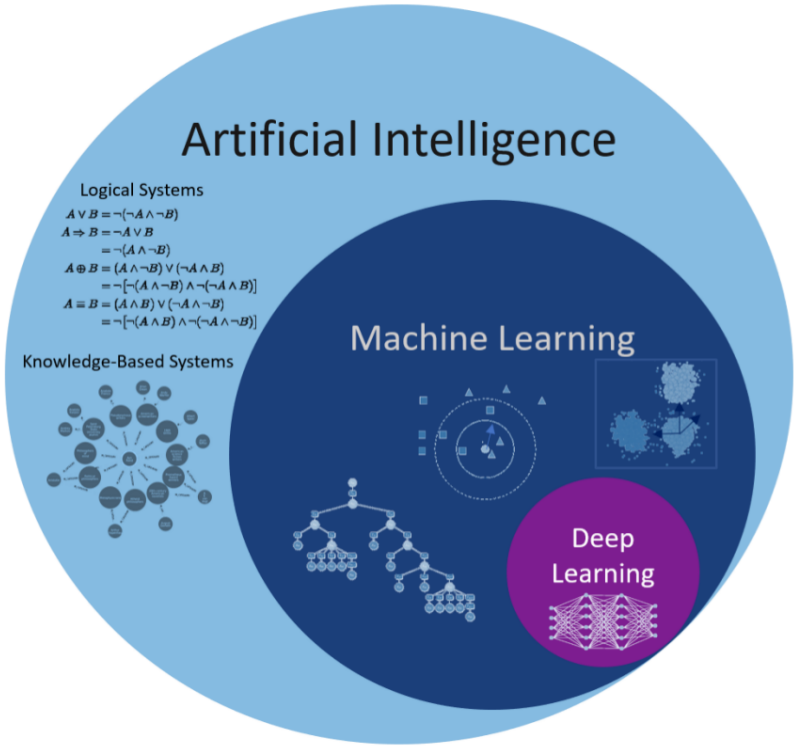
March 27, 2023

* *Thanks to Geoffroy Peeters for his kind permission to use many of the images in these slides*

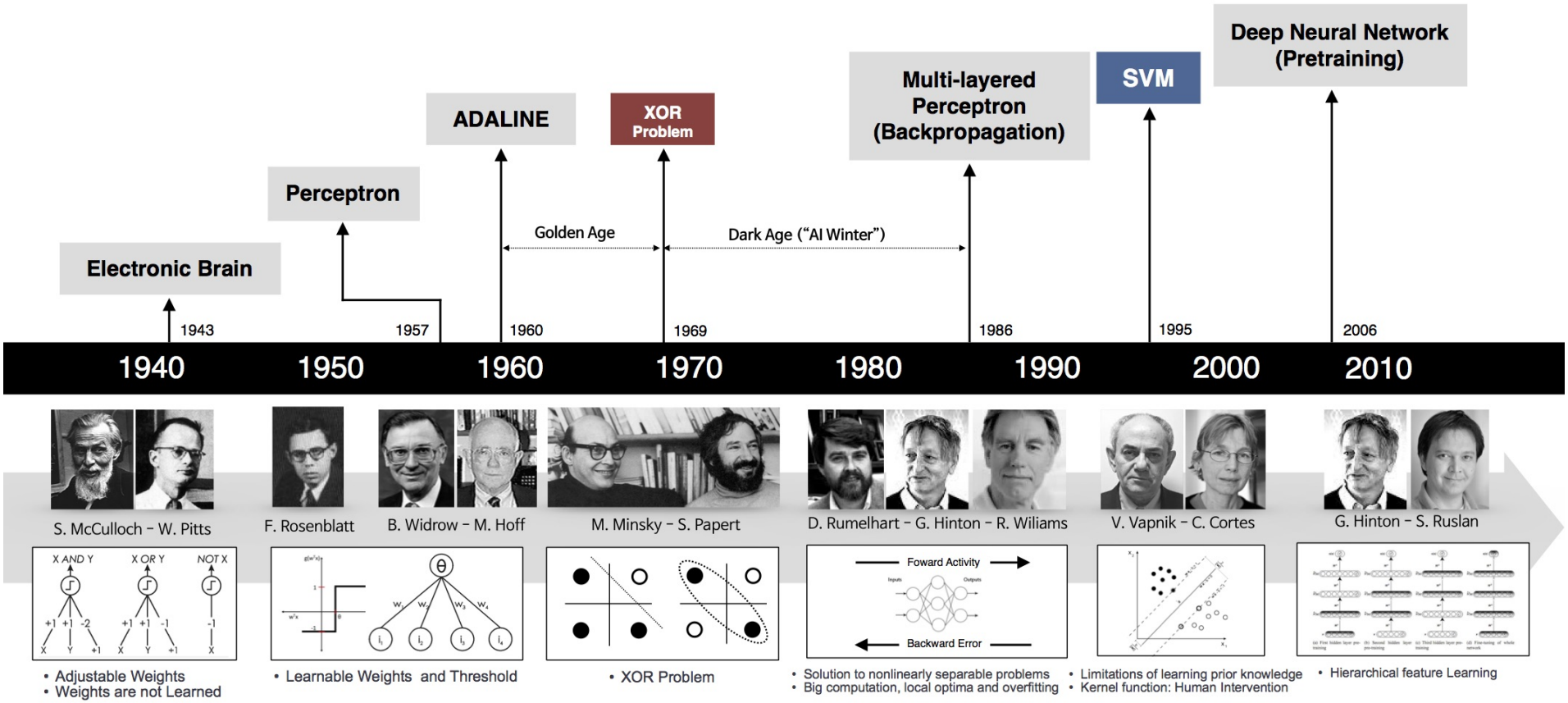
Summary

Introduction

- In this lesson, we will look at Artificial Neural Networks, and in particular **Multi-Layer Perceptrons**
  - Class of machine learning algorithms/models
- Wide range of applications, supervised or unsupervised learning
- More recently, **deep learning** has had a huge impact on many domains, and in particular, image processing and computer vision



Evolution of AI and deep learning

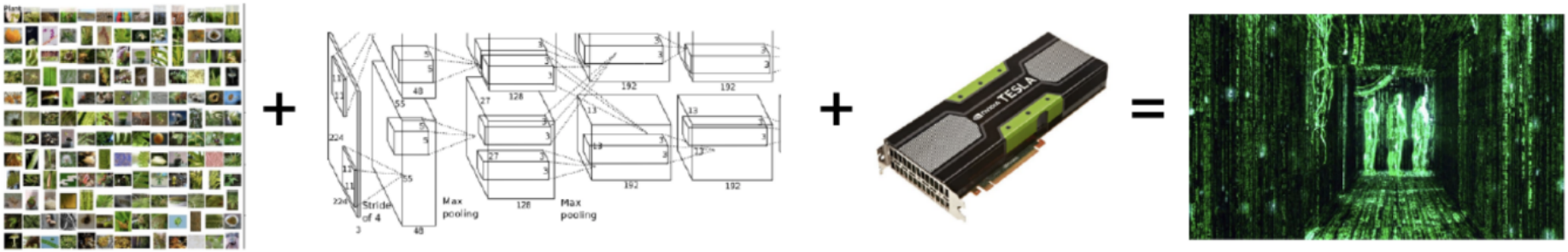


* From https://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html

Evolution of AI and deep learning

- "Deep learning" has been known by various names for a long time ("cybernetics", artifical neural networks etc)
- Why has there been a recent explosion of deep learning ?
  - Increase in number of large datasets (in terms of number of elements and dimensionality)
  - **Increase in computing power** (GPUs)

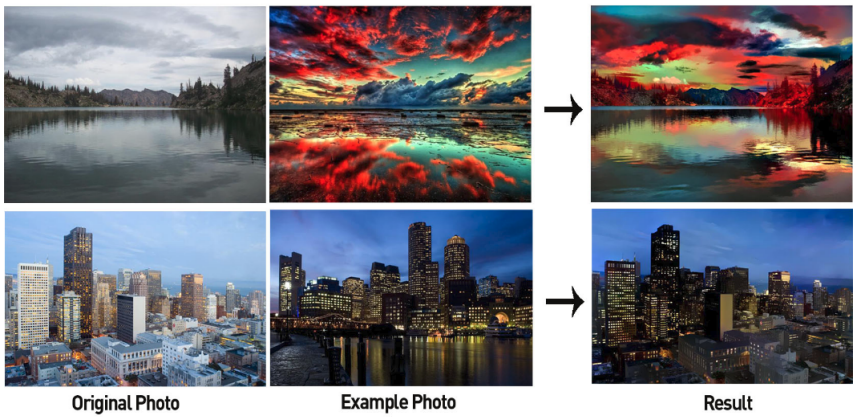
The Deep Learning "Computer Vision Recipe"



Big Data: ImageNet | Deep Convolutional Neural Network | Backprop on GPU | Learned Weights

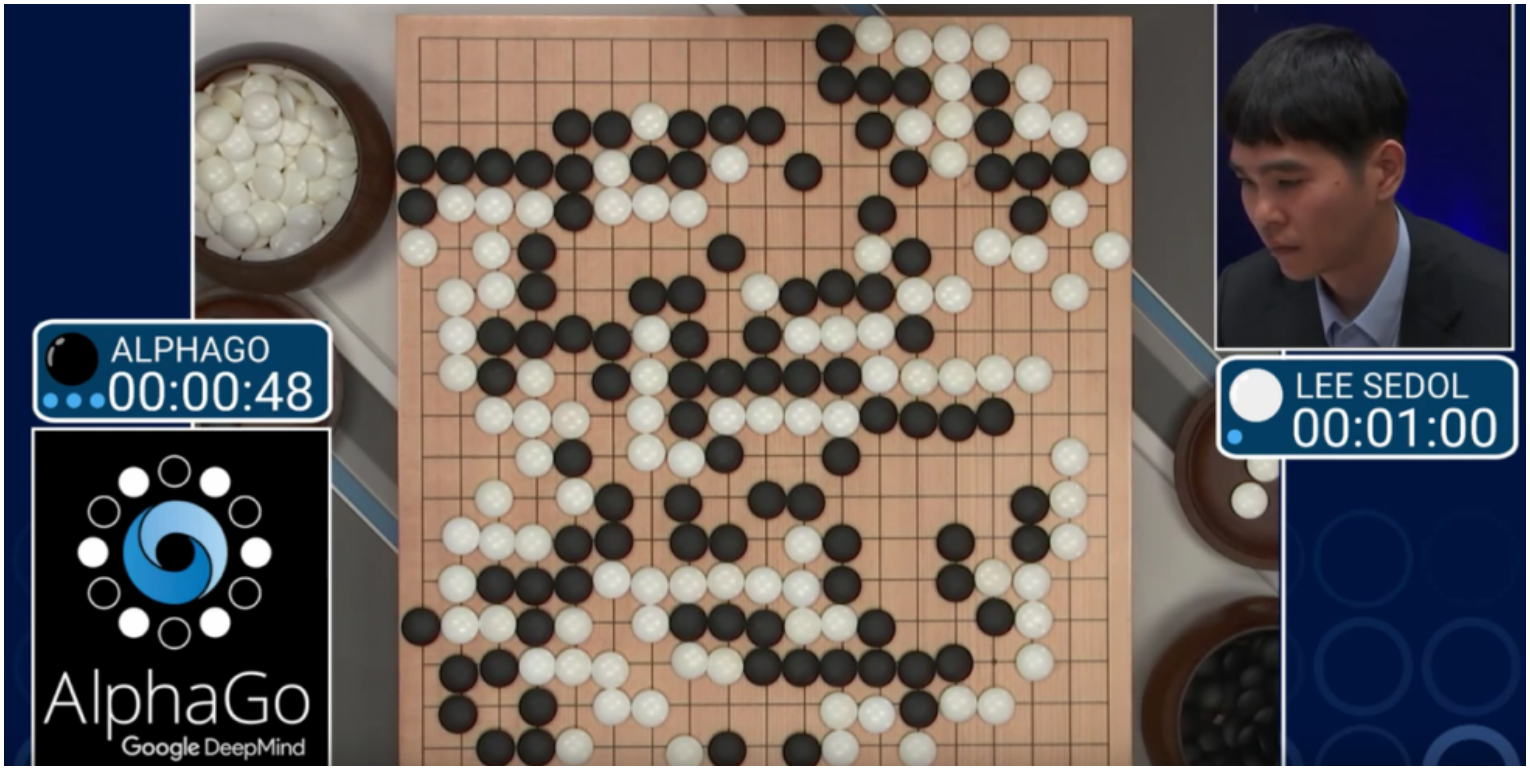
Introduction

- Modern deep learning has applications in practially all areas of image processing/computer vision, and in a multitude of other domains



Style transfer

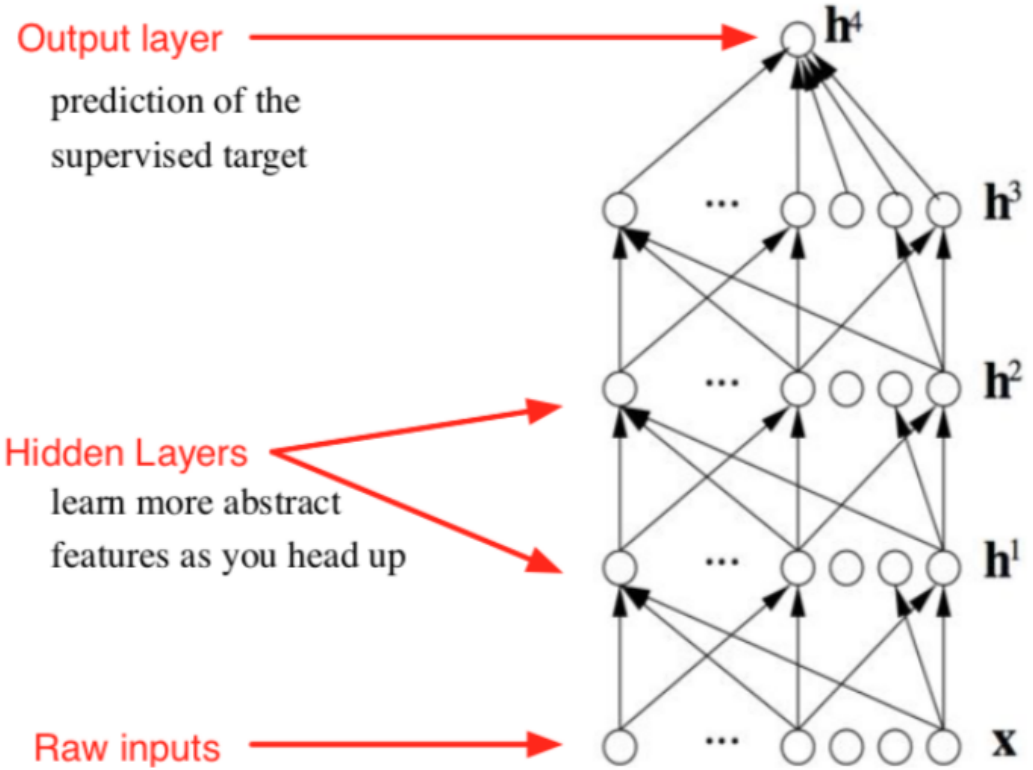
Evolution of AI and deep learning

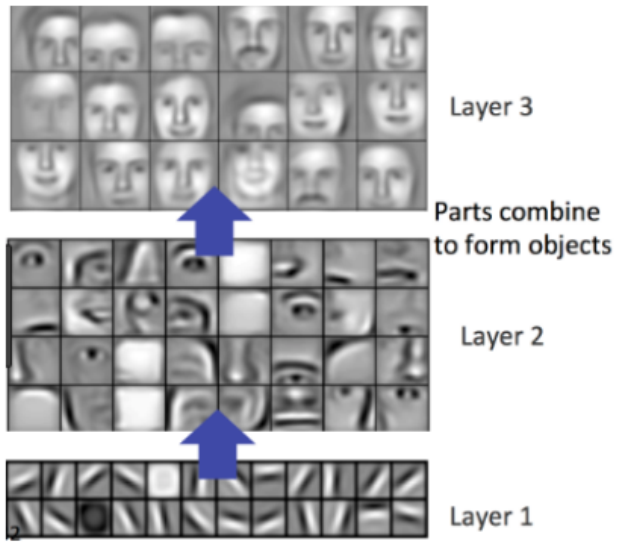


* From https://www.deeplearningitalia.com/a-gentle-overview-on-the-deep-learning-and-machine-learning-2/

Introduction

- Idea of a deep neural network : hierarchical representation of images/data
- Increased depth : increased descriptive power and complexity





Summary

Biological neuron

- The brain and its functioning inspired early artificial intelligence
- Neurons are fundamental components of the nervous system

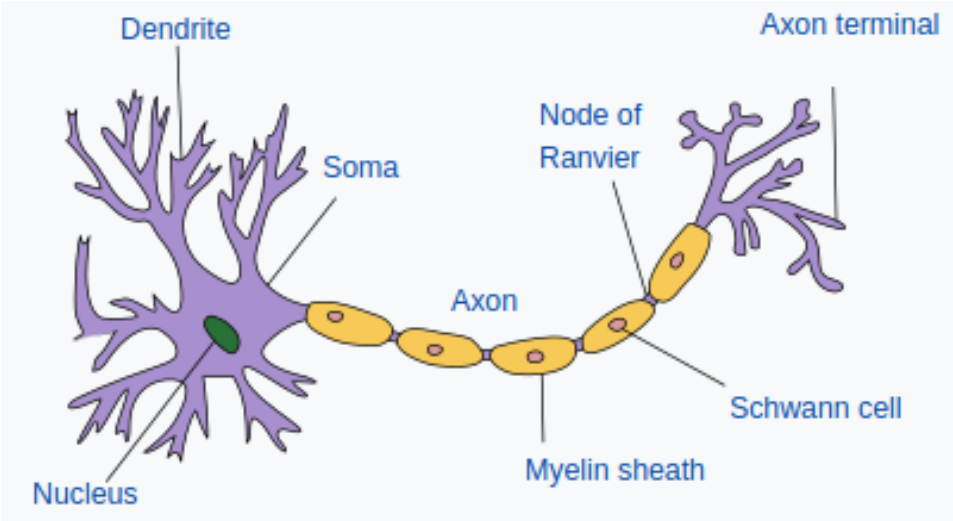


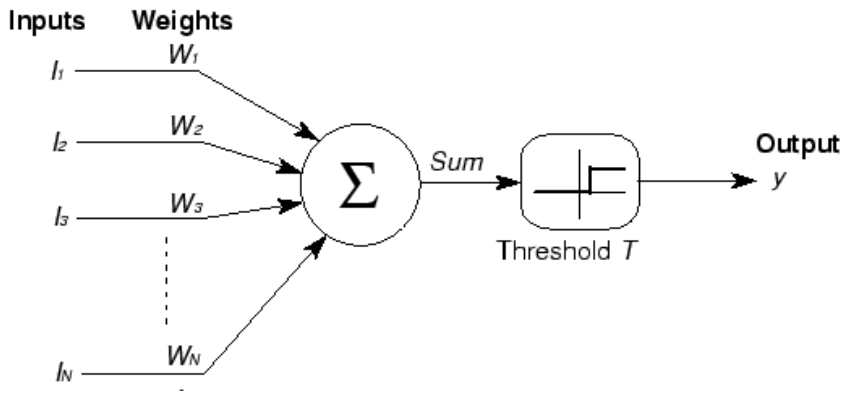
Illustration from https://en.wikipedia.org/wiki/Neuron

- Neurons recieve electrical signals via *dendrites*, process the signal in the *soma* and transmit the output via *synapses*
- Neurons have an "all or none" principle : under a threshold, no electrical signal is emitted, above the threshold, the same one is always emmitted

Biological neuron

- A mathematical model of the neuron was first proposed by Mcculloch and Pitts*
- The model is quite simple (in fact, the simplest possible ANN) :
  - A series of inputs $\mathbf{x}^{(j)} \in \mathbb{R}^n$
  - A list of weights $\mathbf{w}_i, i \in \{0, \ldots, n-1\}$
  - A threshold T (inspired by the "all or none" principle)

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_i \mathbf{w}_i \mathbf{x}_i > T \\ 0 & \text{otherwise} \end{cases}$$



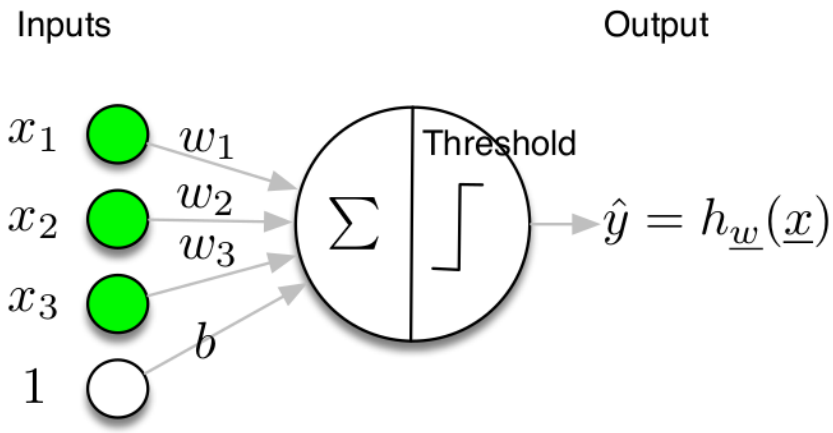
- The weights had to be set by hand; this is quite time-consuming

*, ***A logical calculus of the ideas immanent in nervous activity***, *W. McCulloch and W. Pitts, The bulletin of mathematical biophysics 5.4: 115-133, 1943*

Multi-Layer Perceptron

- Rosenblatt* proposed an algorithm to optimise these weights
- The **perceptron** is an algorithm for binary classification

$$h_w(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{wx} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$



- Notice the **additive** factor b
  - Known as the **bias**.

* *The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain, Psychological Review, 65 (6): 386–408, 1958*

Multi-Layer Perceptron

- Weights are updated by comparing the current prediction with the labels

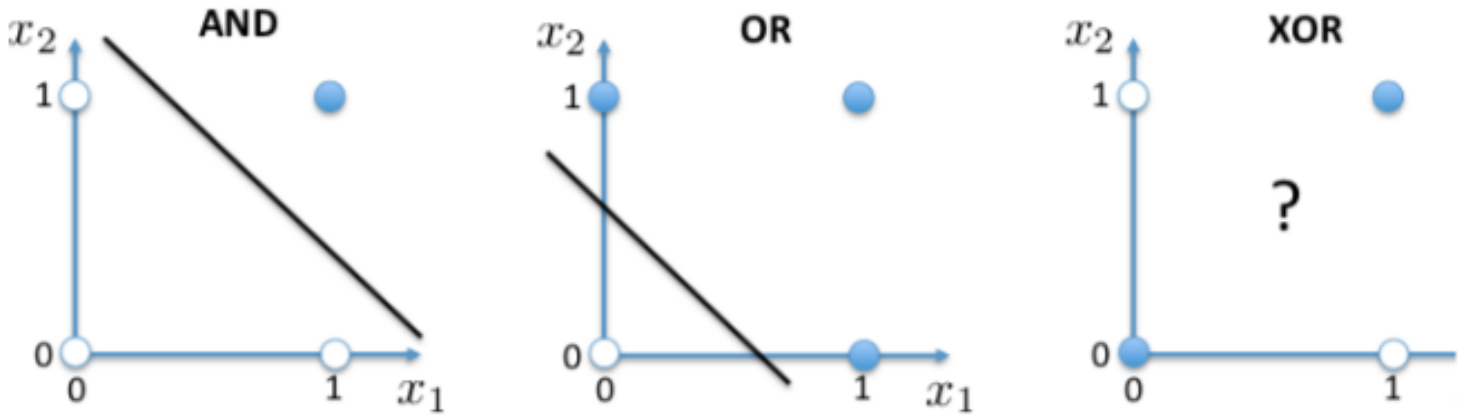
Perceptron algorithm

$\mathbf{X} = \left[\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\right]$: dataset (of size m), with $\mathbf{x}^{(i)} \in \mathbb{R}^d$
$\hat{\mathbf{Y}} = \left[\hat{\mathbf{y}}^{(1)}, \ldots, \hat{\mathbf{y}}^{(m)}\right]$: labels (0 or 1)
Initialise weights $\mathbf{w}$
for $i = 0$ to m **do**
  $y \leftarrow f(\mathbf{w}\mathbf{x}^{(i)})$
  for $j = 0$ to n **do**
    $\mathbf{w}_j \leftarrow \mathbf{w}_j + \alpha(\hat{\mathbf{y}}_j^{(i)} - \mathbf{y}_j^{(i)})\mathbf{x}_j^{(i)}$
  end for
end for

- α : learning parameter

Multi-Layer Perceptron

- Unfortunately, a significant problem was found in the case of the perceptron
- Perceptron is a linear classifier. Minsky and Papert showed that the **xor** function could not be approximated by a perceptron: the **xor is not linearly separable**
- Perceptron can only classifier linearly separable data



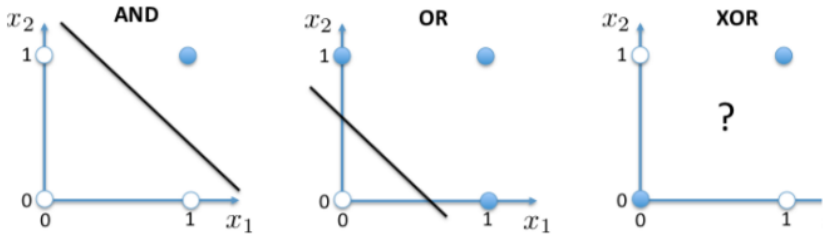
One layer

- This introduced a long period (about 10 or so years) when neural networks fell into disfavour. This was known as the "AI winter"

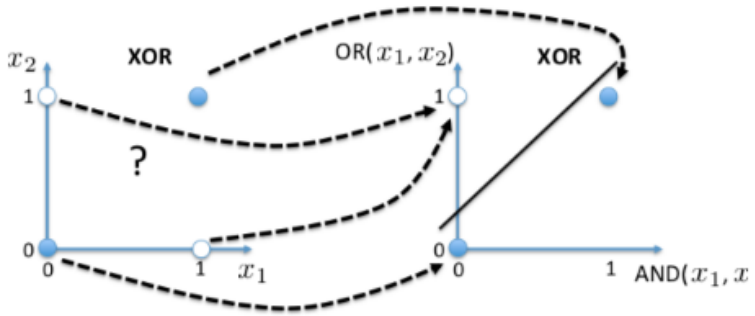
*, ***Perceptrons: An introduction to computational geometry***, *M. Minsky and S. A. Papert, Cambridge, MA: MIT Press, 1969*

Multi-Layer Perceptron

- However, a solution was found to this by introucing **several layers** ("hidden layers")
  - This lead to the **Multi-layer Perceptron** (MLP)
- Rumelhart et al.* showed that such a network could be trained to learn the xor function, using the **backpropagation algorithm**



One layer



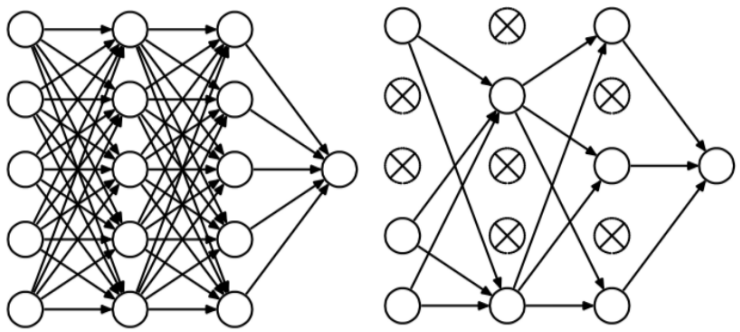
Two layers

- This ushered in a new interest in Artificial Neural Networks, which we look at now

*, ***Learning internal representations by error propagation***, *D. Rumelhart, G. Hinton and R. Williams, California University San Diego LA Jolla Inst. for Cognitive Science, 1985*

Artificial neural network

- An **Artificial Neural Network** (ANN) is simply a sequence of simple functions which are applied to some inputs in a cascade
  - Note : there can be **no loops** in the sequence of functions, this will be important further on
- The **Multi-Layer Perceptron** (MLP) is a special case of an ANN
  - Several layers (at least two : one input, one hidden)
  - Each neuron in one layer is **connected to each neuron in the next layer by a single, unique weight**
  - Also known as a **fully connected layer**



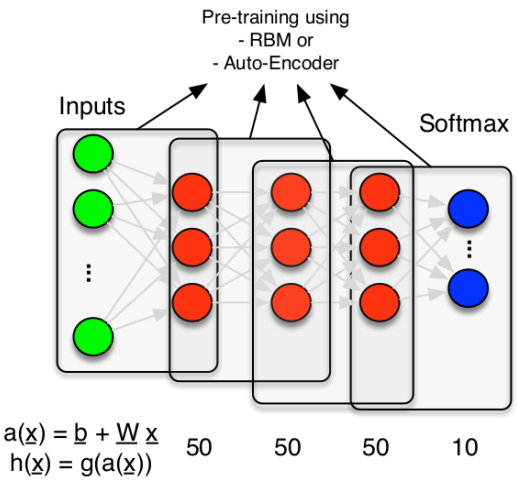
MLP Aribtrary ANN

- We will be focussing on MLPs in this lesson

Artificial neural network

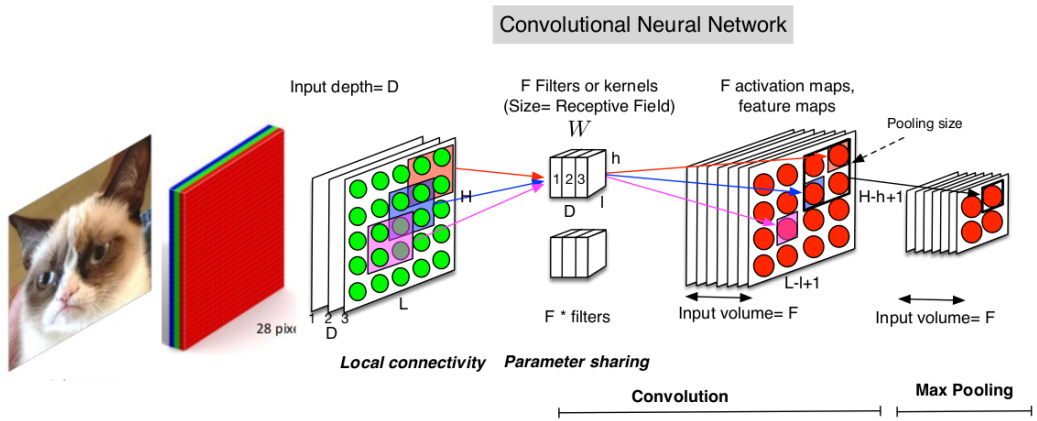
- Here are **three of the major types** of ANNs :
  1. MLP : **Multilayer Perceptron**
  2. CNN : **Convolutional Neural Network**
  3. RNN : Recurrent Neural Network

Multi Layers Perceptron (Fully Connected)



$$\underline{z}^{[l]} = \underline{\underline{W}}^{[l]}\underline{a}^{[l-1]} + \underline{b}^{[l]}$$
$$\underline{a}^{[l]} = g^{[l]}(\underline{z}^{[l]})$$

MLP



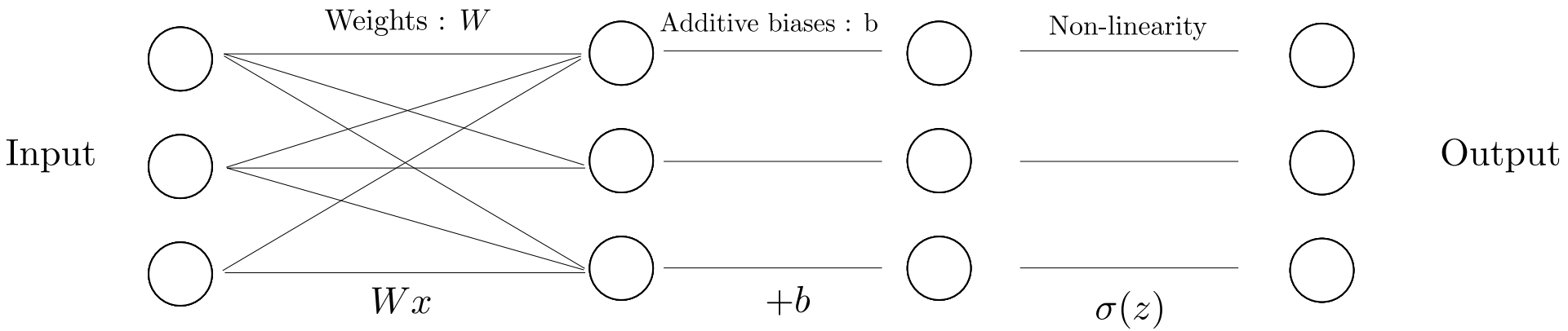
CNN

Artificial neural network

- Before continuing, let's define (or recall) some notation
  - $\mathbf{x} \in \mathcal{X}$: input
  - $\mathbf{y} \in \mathcal{Y}$: output
  - $\mathbf{x}_i$: ith element of the vector $\mathbf{x}$, and likewise for $\mathbf{y}_i$
  - $\mathbf{x}^{(i)}$: ith data sample in a database, and likewise for $\mathbf{y}^{(i)}$
  - $\hat{\mathbf{y}} \in \mathcal{Y}$: a label/true value of the output (we want to predict this)
  - $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$: the loss between the prediction $\mathbf{y}$ and the true value $\hat{\mathbf{y}}$
  - θ : the parameters of the network
- A **layer** often refers to weights $+$ bias $+$ non-linearity
- A **hidden layer** is an intermediate layer in the network

Artificial neural network

- An ANN contains the following main components :
  - Weights, $\mathbf{W}$, a linear transformation. This can be represented as a matrix/vector multiplication : $\mathbf{y} = \mathbf{W}\mathbf{x}$
  - Additive biases, $\mathbf{b}$: $\mathbf{y} = \mathbf{x} + \mathbf{b}$
  - Non-linearities, $\mathbf{y} = \sigma(\mathbf{x})$
  - Loss function $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$



- The ANN is just the cascade of these functions
- **Training** an ANN will consist in finding an algorithm to progressively modify/tune the parameters to minimise a **loss function**

Artificial neural network

- Why are ANNs so useful and widely used ? An important result known as the **universal approximation theorem***
- This states that any continuous function may be approximated with arbitrary precision by a large enough neural network with **one hidden layer**
  - One hidden layer, and many, many neurons
- This is important because it means that with enough capacity, we can approximate extremely complex and high-level functions
- Caveat : the network may be very, unfeasibly, large

* ***Approximations by superpositions of sigmoidal functions***, *G. Cybenko, Mathematics of Control, Signals, and Systems, 2(4), 303–314, 1989*

Artificial neural networks

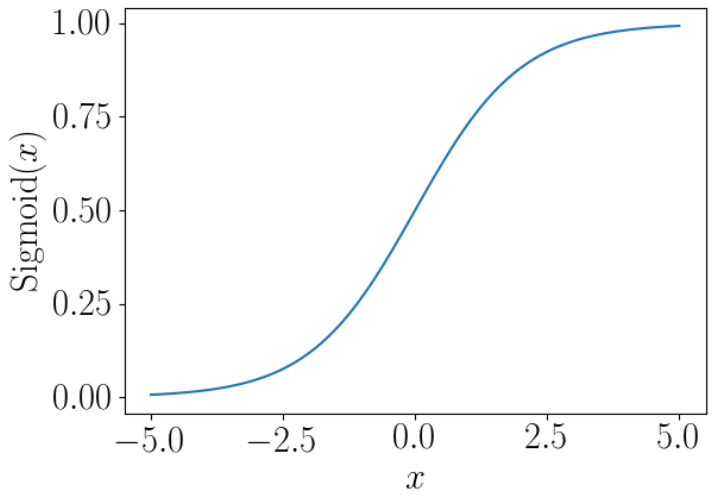
- **Non-linearities** are a crucial component in artificial neural networks
- Without these non-linearities, the ANN would just be a linear transformation
  - $\mathbf{W}_n \dots \mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = \mathbf{V}\mathbf{x}$, for some linear transform $\mathbf{V}$
- Several non-linearities exist, which are useful in different situations
  - Sigmoid (logistic function)
  - Tanh
  - Rectified Linear Unit (ReLU)
  - Leaky ReLU
- Linear activation only useful in last layer for regression problem, with $\mathbf{y}_i \in \mathbb{R}$

Artificial neural networks - non-linearities

- A commonly used non-linearity is the **sigmoid function**

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

$$\begin{aligned}
\sigma'(x) &= \frac{-(-e^{-x})}{(1+e^{-x})^2} \\
&= \frac{(1+e^{-x})-1}{(1+e^{-x})^2} \\
&= \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \sigma(x)(1-\sigma(x))
\end{aligned}$$



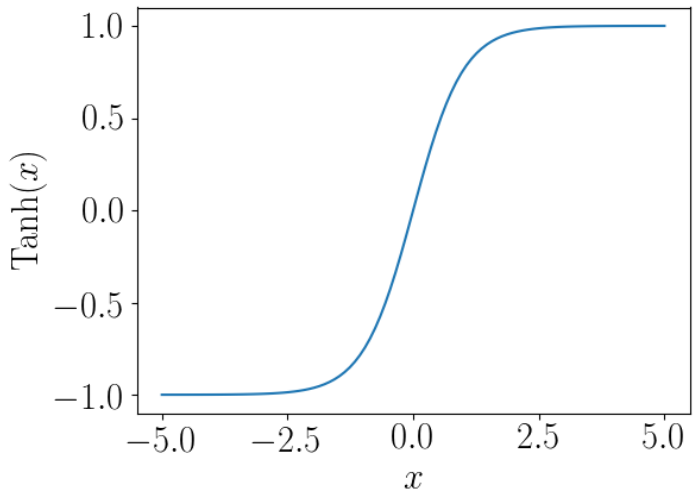
- Useful to squeeze the output into the interval $(0, 1)$
- Most often used as the output non-linearity in the case of **binary classification** problems (logistic regression)

Artificial neural networks - non-linearities

- **Hyperbolic tangent function** : Tanh

$$\text{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\text{Tanh}'(x) = 1 - (\text{Tanh}(x))^2$$



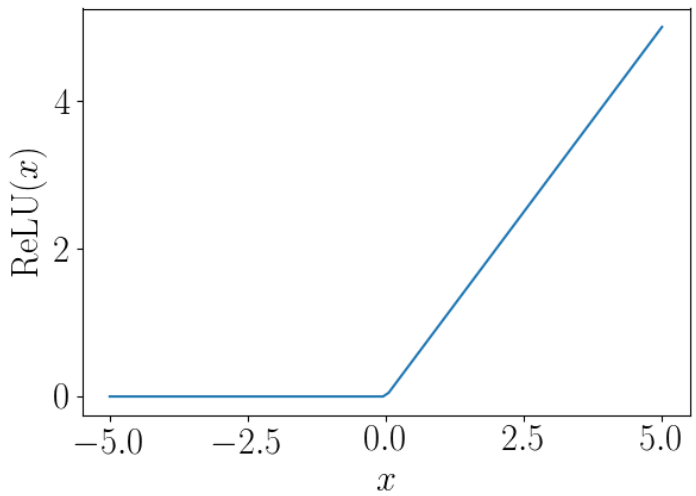
- Sets the output to the interval $(-1, 1)$
- Can be more useful in intermediate layers of networks
- A problem with sigmoid and tanh functions : gradients vanish at very small and large values of x

Artificial neural networks - non-linearities

- ReLU : **Rectified Linear Unit**
- Often used because of its implementation and computational simplicity

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$



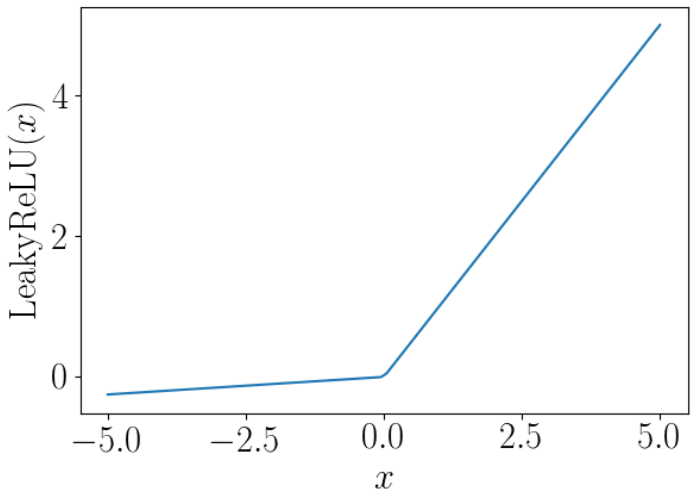
- This is not, strictly speaking, differentiable, but it is differentiable almost everywhere

Artificial neural networks - non-linearities

- A modification of the ReLU is the **leaky ReLU**
- Gradient is small, but non-zero below zero

$$\text{LeakyReLU}_\alpha(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{LeakyReLU}'_\alpha(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{otherwise} \end{cases}$$



- Avoids the problem of **dead neurons** : neurons which get deactivated during training due to zero gradient

Artificial neural networks - non-linearities

- Finally, the **softmax** function is used for **multi-class** classification problems
- Contrary to other functions (apart from sigmoid), this normalises the output over all the output nodes, so that they sum to 1

$$\text{Softmax}(\mathbf{x})_i = \frac{e^{\mathbf{x}_i}}{\sum_{i=1}^{K} e^{\mathbf{x}_i}}$$

- K is the number of classes
- The output is a **probability distribution** :
  - $\text{Softmax}(\mathbf{x})_i \in (0, 1)$
  - $\sum_i \text{Softmax}(\mathbf{x})_i = 1$

Artificial neural networks - loss functions

- Finally, the last component to chose when setting up an ANN is the **loss function**
- This obviously depends on the problem
- Regression problem : **mean squared error**
  - $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{d} \sum_i (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2$
- Binary classification (yes/no) : **binary cross-entropy**
  - $\mathcal{L}(y, \hat{y}) = -\hat{y} \log(y) - (1 - \hat{y}) \log(1 - y)$
- Multi-class classification (cat/dog/bird): **categorical cross-entropy**
  - $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{K} \hat{\mathbf{y}}_i \log(\mathbf{y}_i)$
- The final loss must be a *scalar*, take the **average** over the dataset

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)})$$

Artificial neural networks - loss functions

- Summary of MLP components to use for different problems

Problem type	Final non-linearity	Loss
Regression	Linear, ReLU	$\frac{1}{2}\sum_i(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2$ (MSE)
Binary classification	Sigmoid	$-\hat{y}\log y - (1-\hat{y})\log(1-y)$ (Binary cross-entropy)
Multi-class classification	Softmax	$-\sum_i^K \hat{\mathbf{y}}_i \log \mathbf{y}_i$ (Categorical cross-entropy)

Summary

Backpropagation

- Now that we are able to *define* a network, we need an algorithm to **train on a dataset**
- ANNs are trained using **gradient descent**, or variants thereof

$$\theta^{(n+1)} = \theta^{(n)} - \alpha \nabla_\theta \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}_i)$$

- However, for this, we need to be able to calculate the derivatives of the loss function with respect to the parameters
- Here, for ease of notation, we define $dz := \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial z}$

Backpropagation

- Let's take the case of a single layer, single variable, ANN, with
  - w : weight
  - b : bias
  - f : non-linearity
  - $\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$: loss function
- Let $z = wx + b$
- $y = f(z) = f(wx + b)$
- In order to update the network, we need following quantities : dw, db

Backpropagation

- dw, db can be calculated directly, using basic calculus ("analyse" in french)

$$
\begin{aligned}
dw &= \frac{\partial}{\partial w} \mathcal{L}(f(wx+b), \hat{y}) \\
&= \frac{\partial}{\partial w} \left(\frac{1}{2} \left(f(wx+b) - \hat{y} \right)^2 \right) \\
&= \left(f(wx+b) - \hat{y} \right) \; \frac{\partial}{\partial w} \left(f(wx+b) - \hat{y} \right) \\
&= \left(f(wx+b) - \hat{y} \right) \; \left(f'(wx+b) \right) \; \frac{\partial}{\partial w} \left(wx+b \right) \\
&= \left(f(wx+b) - \hat{y} \right) \; \left(f'(wx+b) \right) \; x
\end{aligned}
$$

- However, you can imagine that this would quickly become inefficient if we had many layers !
- How can we solve this ? The **backpropagation** algorithm

Backpropagation

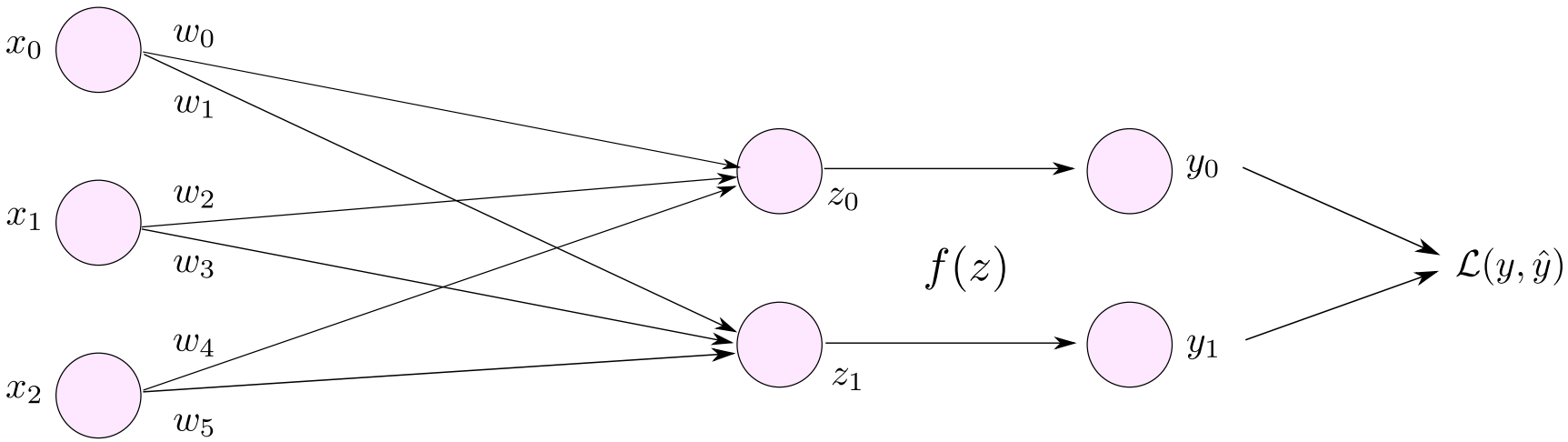
- The **backpropagation** algorithm is an efficient way of implementing the **chain rule** !!
- In the previous example, we developed the derivative using this rule (in the univariate case)

$$dw = \frac{\partial}{\partial y}\mathcal{L}(y, \hat{y}) \; \frac{\partial}{\partial z}(f(wx+b)) \; \frac{\partial}{\partial w}(wx+b)$$

- Note, $\frac{\partial}{\partial z}(f(wx+b))$ corresponds to the derivative of the single-valued function f, evaluated at $f(wx+b)$
- If we know how to calculate the partial derivatives for each computation, we can create an algorithm to determine any derivative
- This idea is the basis of the backpropagation algorithm

Backpropagation

- The previous case dealt with one neuron per layer
- In general, we have several neurons. In this case, we need the **multi-variate** chain rule



Backpropagation

- Again, let's start out with a simple example
- Let $x = f(t)$ and $y = g(t)$ be two single-variable differentiable functions
- Let $h(x, y)$ be a differentiable function of *two* variables

$$\begin{aligned}\frac{d}{dt}h(f(t), g(t)) &= \frac{\partial}{\partial x}h(x, y)\frac{\partial x}{\partial t} + \frac{\partial}{\partial y}h(x, y)\frac{\partial y}{\partial t} \\ &= \frac{\partial}{\partial x}h(f(t), g(t))\frac{\partial f(t)}{\partial t} + \frac{\partial}{\partial y}h(f(t), g(t))\frac{\partial g(t)}{\partial t}\end{aligned}$$

Backpropagation

- More generally, we can write the chain rule in the case of multiple variables and functions

Multi-variate chain rule

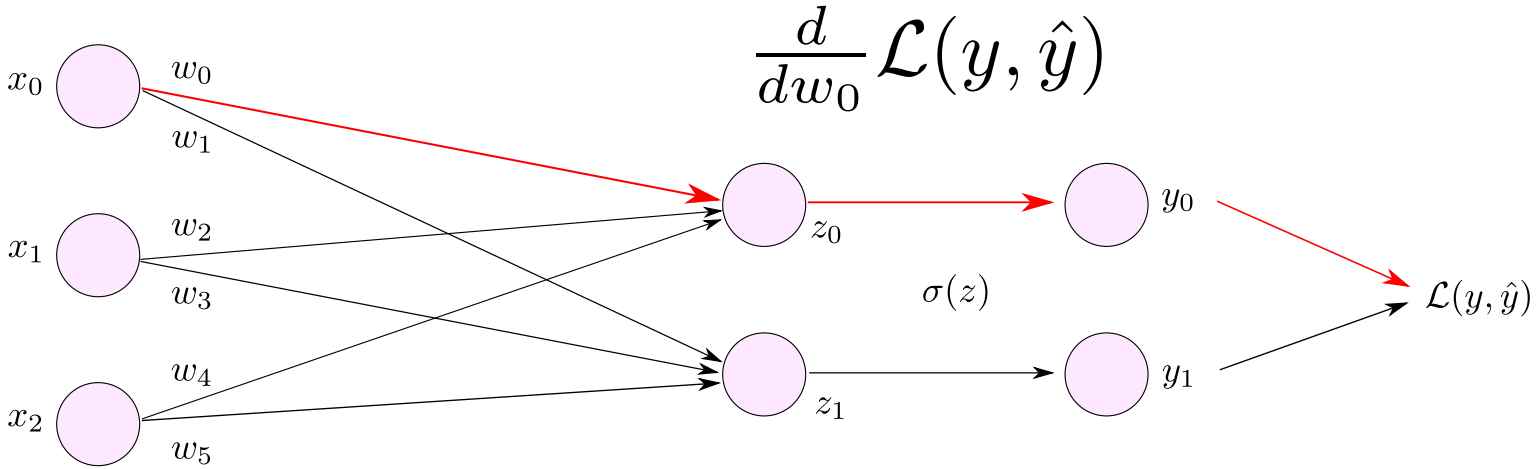
- Let $\mathbf{x} = (x_0, \ldots, x_{n-1})$ be a series of variables
- Let $f_0(\mathbf{x}), \ldots, f_{m-1}(\mathbf{x})$ be a series of functions
- Let $h(f_0, \ldots, f_{m-1})$ be another function

$$\frac{d}{dt_i} h(f_0(\mathbf{x}), \ldots, f_{m-1}(\mathbf{x})) = \sum_j \frac{\partial}{\partial f_j} h(f_0(\mathbf{x}), \ldots, f_{m-1}(\mathbf{x})) \frac{\partial f_j(\mathbf{x})}{\partial x_i}$$

- We have used an abuse of notation: in $h(f_0, \ldots, f_{m-1})$ the $f_0, \ldots, f_{m-1}$ are variables

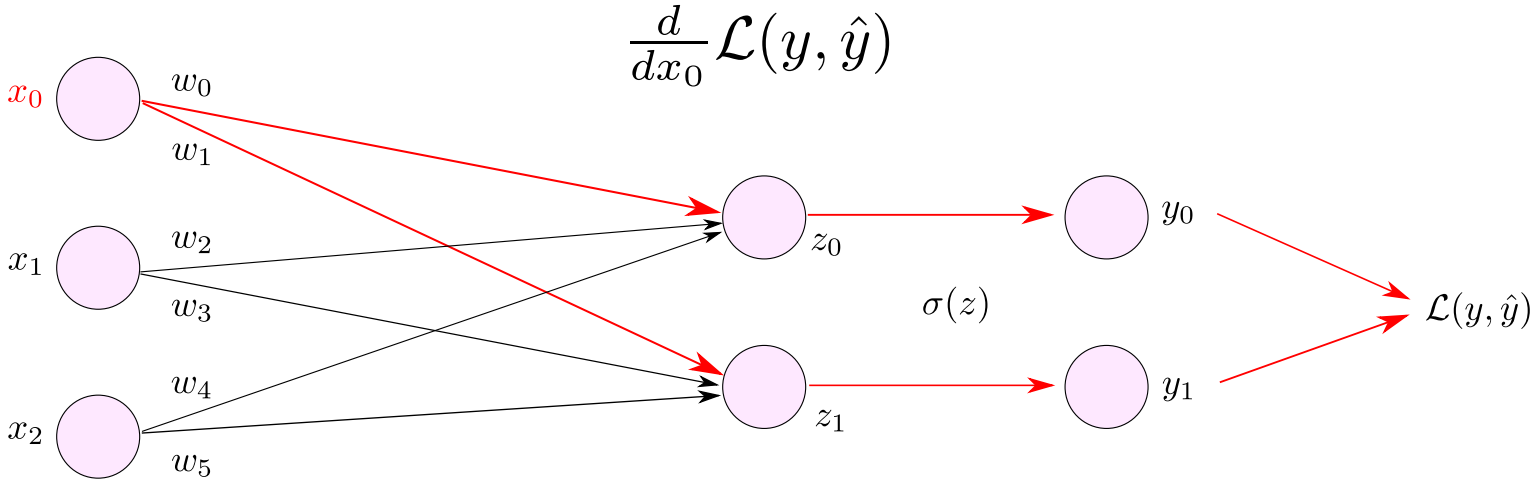
Backpropagation

- This means that we can determine **any derivative** we like



Backpropagation

- Here is a case with **several paths** of dependence



$$\frac{d}{dx_0}\mathcal{L}(y;\hat{y}) = \frac{\partial}{\partial y_0}\mathcal{L}(y;\hat{y})\ \frac{\partial}{\partial z_0}\sigma(z)\ \frac{\partial}{\partial x_0}w_0x_0 + \frac{\partial}{\partial y_1}\mathcal{L}(y;\hat{y})\ \frac{\partial}{\partial z_1}\sigma(z)\ \frac{\partial}{\partial x_0}w_1x_1$$

- We have taken the derivative wrt x_0 here

Backpropagation

- Let's take another example with two layers (one hidden layer)
- Furthermore, given a dataset, we can write the calculation in matrix format, to take the dataset into account

$$\mathbf{X} \longrightarrow \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \xrightarrow{\mathbf{z}^{[1]}} f\left(\mathbf{z}^{[1]}\right) \xrightarrow{\mathbf{a}^{[1]}} \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} \xrightarrow{\mathbf{z}^{[2]}} \sigma\left(\mathbf{z}^{[2]}\right) \xrightarrow{\mathbf{a}^{[2]}} \mathbf{y}$$

- Input dimension : n_0, number of data samples : m
- Number of neurons in first layer : n_1, second layer : n_2 (output dimension)
- Two weight matrices : $\mathbf{W}^{[1]} \in \mathbb{R}^{n_1 \times n_0}$, $\mathbf{W}^{[2]} \in \mathbb{R}^{n_2 \times n_1}$
- Biases : $\mathbf{b}^{[1]} \in \mathbb{R}^{n_1}$, $\mathbf{b}^{[2]} \in \mathbb{R}^{n_2}$
- Intermediate layer non-linearity : f
- Sigmoid output non-linearity : $\sigma(x) = \frac{1}{1+e^{-x}}$
- Output of layer ℓ : $\mathbf{a}^{[\ell]} := \sigma(W^{[\ell]}\mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]})$
- Binary cross-entropy loss : $\mathcal{L}(y, \hat{y}) = -\hat{y} \log y - (1 - \hat{y}) \log (1 - y)$

Backpropagation

- Forward pass

$$\underbrace{\mathbf{Z}^{[1]}}_{(n^{[1]},m)} = \underbrace{\mathbf{W}^{[1]}}_{(n^{[1]},n^{[0]})} \underbrace{\mathbf{X}}_{(n^{[0]},m)} + \underbrace{\mathbf{b}^{[1]}}_{n^{(1)}}$$

$$\underbrace{\mathbf{A}^{[1]}}_{(n^{[1]},m)} = f(\mathbf{Z}^{[1]})$$

$$\underbrace{\mathbf{Z}^{[2]}}_{(n^{[2]},m)} = \underbrace{\mathbf{W}^{[1]}}_{(n^{[2]},n^{[1]})} \underbrace{\mathbf{A}^{[1]}}_{(n^{[1]},m)} + \underbrace{\mathbf{b}^{[1]}}_{n^{(2)}}$$

$$\underbrace{\mathbf{A}^{[2]}}_{(n^{[2]},m)} = \sigma(\mathbf{Z}^{[2]}) =: \mathbf{Y}$$

$$\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\mathbf{Y}^{(i)}, \hat{\mathbf{Y}}^{(i)})$$

Backpropagation

- Now we carry out the backward pass
- Note, in the case of binary cross-entropy and a sigmoid function σ, we have

$$\begin{aligned} dz &= \frac{\partial \mathcal{L}(y, \hat{y})}{\partial y} \frac{\partial \sigma(z)}{\partial z} \\ &= \left(-\frac{\hat{y}}{y} + \frac{1-\hat{y}}{1-y} \right) \sigma(z)(1-\sigma(z)) \\ &= \left(\frac{-\hat{y}(1-y) + y(1-\hat{y})}{y(1-y)} \right) y(1-y) \\ &= -\hat{y} + \hat{y}y + y - y\hat{y} \\ &= y - \hat{y} \end{aligned}$$

- Therefore, we can cary out the backpropagation
- For each parameter, we take the **average** over the data samples

Backpropagation

- Backward pass

$$\underbrace{d\mathbf{Z}^{[2]}}_{(n^{[2]},m)} = \underbrace{\mathbf{A}^{[2]}}_{(n^{[2]},m)} - \underbrace{\hat{\mathbf{Y}}}_{(n^{[2]},m)}$$

$$\underbrace{d\mathbf{W}^{[2]}}_{(n^{[2]},n^{[1]})} = \frac{1}{m} \underbrace{d\mathbf{Z}^{[2]}}_{(n^{[2]},m)} \underbrace{\mathbf{A}^{[1]}}_{(n^{[1]},m)}^{T}$$

$$\underbrace{d\mathbf{b}^{[2]}}_{(n^{[2]})} = \frac{1}{m} \sum_{i=1}^{m} \underbrace{d\mathbf{Z}^{[2]}}_{(n^{[2]},m)}$$

$$\underbrace{d\mathbf{A}^{[1]}}_{(n^{[1]},m)} = \underbrace{\mathbf{W}^{[2]}}_{(n^{[2]},n^{[1]})}^{T} \underbrace{d\mathbf{Z}^{[2]}}_{(n^{[2]},m)}$$

$$\underbrace{d\mathbf{Z}^{[1]}}_{(n^{[1]},m)} = \underbrace{d\mathbf{A}^{[1]}}_{(n^{[1]},m)} \odot f'\Big(\underbrace{\mathbf{Z}^{[1]}}_{(n^{[1]},m)} \Big)$$

$$\underbrace{d\mathbf{W}^{[1]}}_{(n^{[1]},n^{[0]})} = \frac{1}{m} \underbrace{d\mathbf{Z}^{[1]}}_{(n^{[1]},m)} \underbrace{\mathbf{X}}_{(n^{[0]},m)}^{T}$$

$$\underbrace{\mathbf{b}^{[1]}}_{(n^{[1]})} = \frac{1}{m} \sum_{i=1}^{m} \underbrace{d\mathbf{Z}^{[1]}}_{(n^{[1]},m)}$$

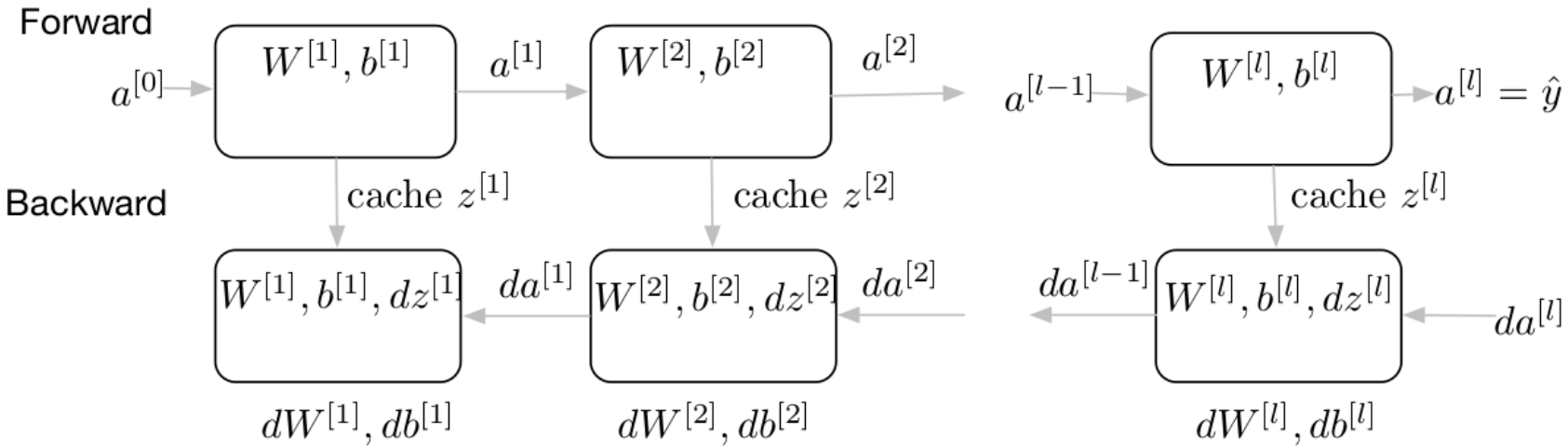
Backpropagation

- To update parameters, for every layer ℓ :
  - $\mathbf{W}^{[\ell]} = \mathbf{W}^{[\ell]} - \alpha \, d\mathbf{W}^{[\ell]}$
  - $\mathbf{b}^{[\ell]} = \mathbf{b}^{[\ell]} - \alpha \, d\mathbf{b}^{[\ell]}$

$$\mathbf{X} \longrightarrow \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \xrightarrow{\mathbf{z}^{[1]}} f\left(\mathbf{z}^{[1]}\right) \xrightarrow{\mathbf{a}^{[1]}} \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} \xrightarrow{\mathbf{z}^{[2]}} \sigma\left(\mathbf{z}^{[2]}\right) \xrightarrow{\mathbf{a}^{[2]}} \mathbf{y}$$

Backpropagation

- This approach generalises to ANNs with even greater numbers of layers (deep : > 2)



$$d\mathbf{Z}^{[\ell]} = d\mathbf{A}^{[\ell]} \odot f^{[\ell]'}(\mathbf{Z}^{(\ell)})$$

$$d\mathbf{W}^{[\ell]} = \frac{1}{m} d\mathbf{Z}^{[\ell]} \ \mathbf{A}^{[\ell-1]^T}$$

$$d\mathbf{b}^{[\ell]} = \frac{1}{m} \sum_{i=1}^{m} d\mathbf{Z}^{\ell}$$

$$d\mathbf{A}^{[\ell-1]} = \mathbf{W}^{[\ell]^T} \ d\mathbf{Z}^{[\ell]}$$

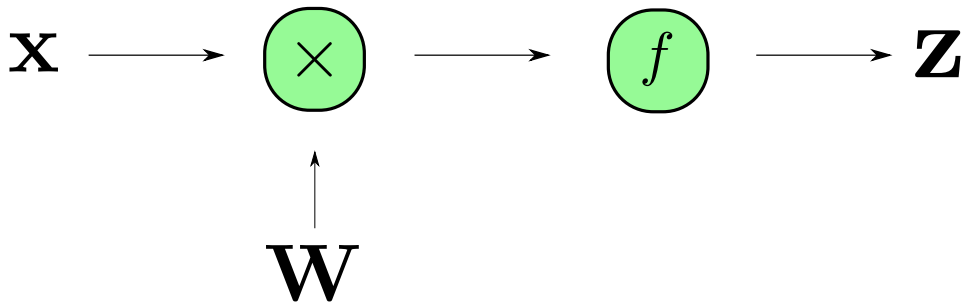
Backpropagation

- However, as such this approach is obviously sub-optimal, **a lot of work !**
- Implementing the foward and backward propagation steps manually for each new network is **prone to errors**
- We would like a way to do carry this out automatically. How is this done ? With a **computation graph**

Computation graph

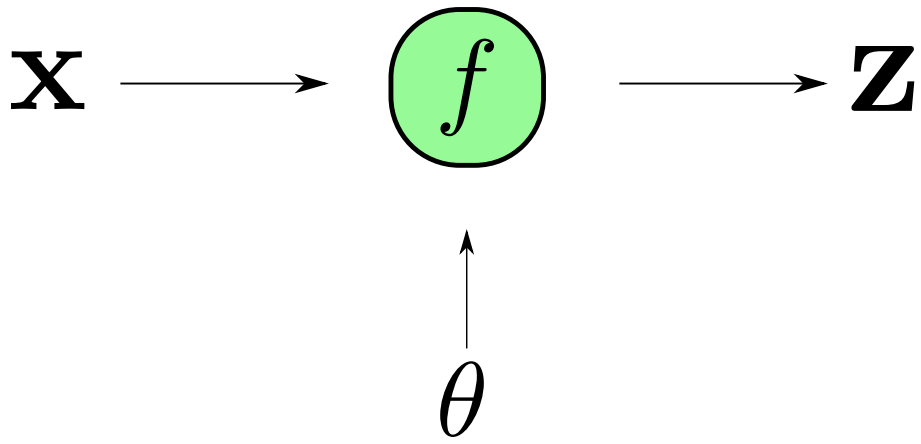
- A **computation graph** is a useful way to represent complex mathematical expressions
- Each node of the computation graph is a simple operation or a variable
- Example, single layer ANN

$$\mathbf{z} = f(\mathbf{W}\mathbf{x})$$



Backpropagation - computational graph

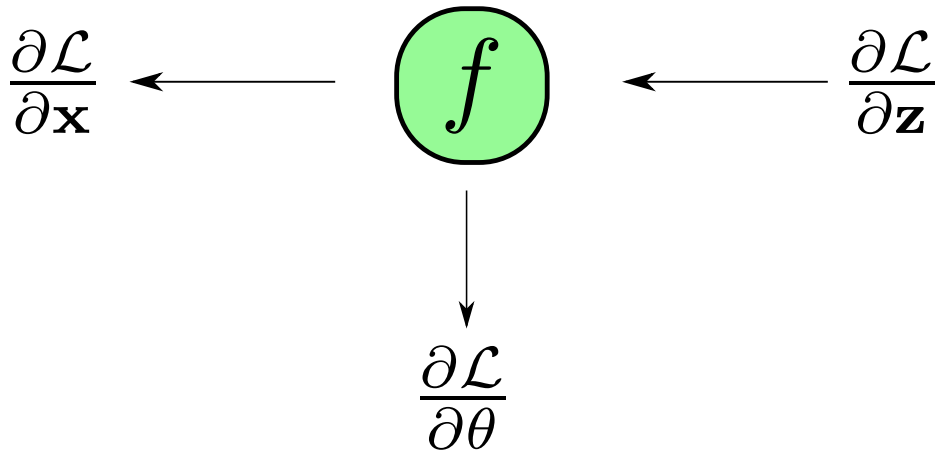
- Consider one elementary function $f(\mathbf{x};\theta)$ of the network, with input $\mathbf{x}$ and output $\mathbf{z}$, and with parameter θ (this can be some weights or biases, or anything else)



- In the **forward pass**, this node simply calculates $\mathbf{z} = f(\mathbf{x};\theta)$, and passes it on to the next computation node

Backpropagation - computational graph

- In the **backward pass**, the node receives the gradients from the **children nodes** in the computation graph : $d\mathbf{z}$
- The node then calculates two quantities :
  - Gradient of the loss with respect to the parameter : $d\theta = d\mathbf{z}\frac{\partial f(\mathbf{x};\theta)}{\partial \theta}$
  - Gradient of the loss with respect to the input : $d\mathbf{x} = d\mathbf{z}\frac{\partial f(\mathbf{x};\theta)}{\partial \mathbf{x}}$



- $d\mathbf{x}$ is passed back to the parent node(s) in the graph
  - This is needed for further gradient calculations;

Backpropagation - computational graph

- An ANN can be described as a **directed acyclic graph**
  - There is an input and an output, with **no loops**, and **no interdependencies** in the neurons of a layer
  - This is why they are also known as **feed-forward** neural networks
  - We can determine an **ordering** of the calculations
- All we need to know is how to calculate the derivatives of the elementary functions in the network
- Given this computation graph, and a correct definition of the nodes (forward/backward/gradients etc.), the backpropagation algorithm can be carried out **without having to define it by hand**

Backpropagation - computational graph

- To summarise, the backpropagation algorithm can be implemented as follows
- Let $v_1, \ldots, v_V$ be a **topological ordering** of the computations in the graph
- Let $\mathsf{Pa}(v_i)$ represent the **parents** of the node v_i
- Let $\mathsf{Ch}(v_i)$ represent the **children** of the node v_i

Backpropagation algorithm with computation graph

$\theta \leftarrow$ Initialise parameters
for $i = 1$ to V **do**
  Compute the v_j's as a function of $\mathsf{Ch}(v_i)$
  Forward pass
end for
for $i = V$ to 1 **do**
  $\theta_i = \theta_i - \alpha \nabla_{\theta_i} \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$ *Gradient of parameters*
  $dv_i = \sum_{j \in \mathsf{Ch}(v_i)} dv_j \frac{\partial v_j}{v_i}$ *Backward pass*
end for

Summary

ANN frameworks

- It is possible to create an ANN and carry out the backpropagation in Python (we will do this in the TP)
- However, it is quite time consuming and prone to error. The following Python packages allow for easier creation and training of ANNs
  - Tensorflow : open source, developed by Google
  - Pytorch : open source, developed by Facebook
  - Keras : open source, a wrapper to ease the creation of ANNs, can use either Tensorflow or Theano as a "backend" (the library with which the functions are actually implemented)
- The following packages were previously used, but are now either discontinued or rarely used
  - Theano : open source, developed by the Université de Montreal
  - Caffe : open source, University of California, Berkley (somewhat laborious to use)

ANN frameworks

- Example of an MLP in python

```
import numpy as np

# Generate dummy data
x_train = np.random.random((1000, 784))
Y_train = np.random.randint(10, size=(1000, 1))

x_test = np.random.random((50, 784))
Y_test = np.random.randint(10, size=(50, 1))

learning_rate = 0.5
nbEpochs = 10
batch_size = 100
D_in = 784
D_out = 10
H = 300
N = x_train.shape[0]
```

ANN frameworks

- Example of an MLP in python

```python
def nn_model(X, Y, n_h, num_iterations=10000):
    for i in range(0, num_iterations):
        A2, cache = forward_propagation(X, parameters)
        cost = compute_cost(A2, Y, parameters)
        grads = backward_propagation(parameters, cache, X, Y)
        parameters = update_parameters(parameters, grads)
    return parameters

def forward_propagation(X, parameters):
    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)
    return A2, cache

def compute_cost(A2, Y, parameters):
    logprobs = np.multiply(np.log(A2), Y) + np.multiply((1 - Y), np.log(1 - A2))
    cost = - np.sum(logprobs) / m
    return cost

def backward_propagation(parameters, cache, X, Y):
    dZ2 = A2 - Y
    dW2 = (1 / m) * np.dot(dZ2, A1.T)
    db2 = (1 / m) * np.sum(dZ2, axis=1, keepdims=True)
    dZ1 = np.multiply(np.dot(W2.T, dZ2), 1 - np.power(A1, 2))
    dW1 = (1 / m) * np.dot(dZ1, X.T)
    db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)
    return grads

def update_parameters(parameters, grads, learning_rate=1.2):
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    return parameters
```

ANN frameworks

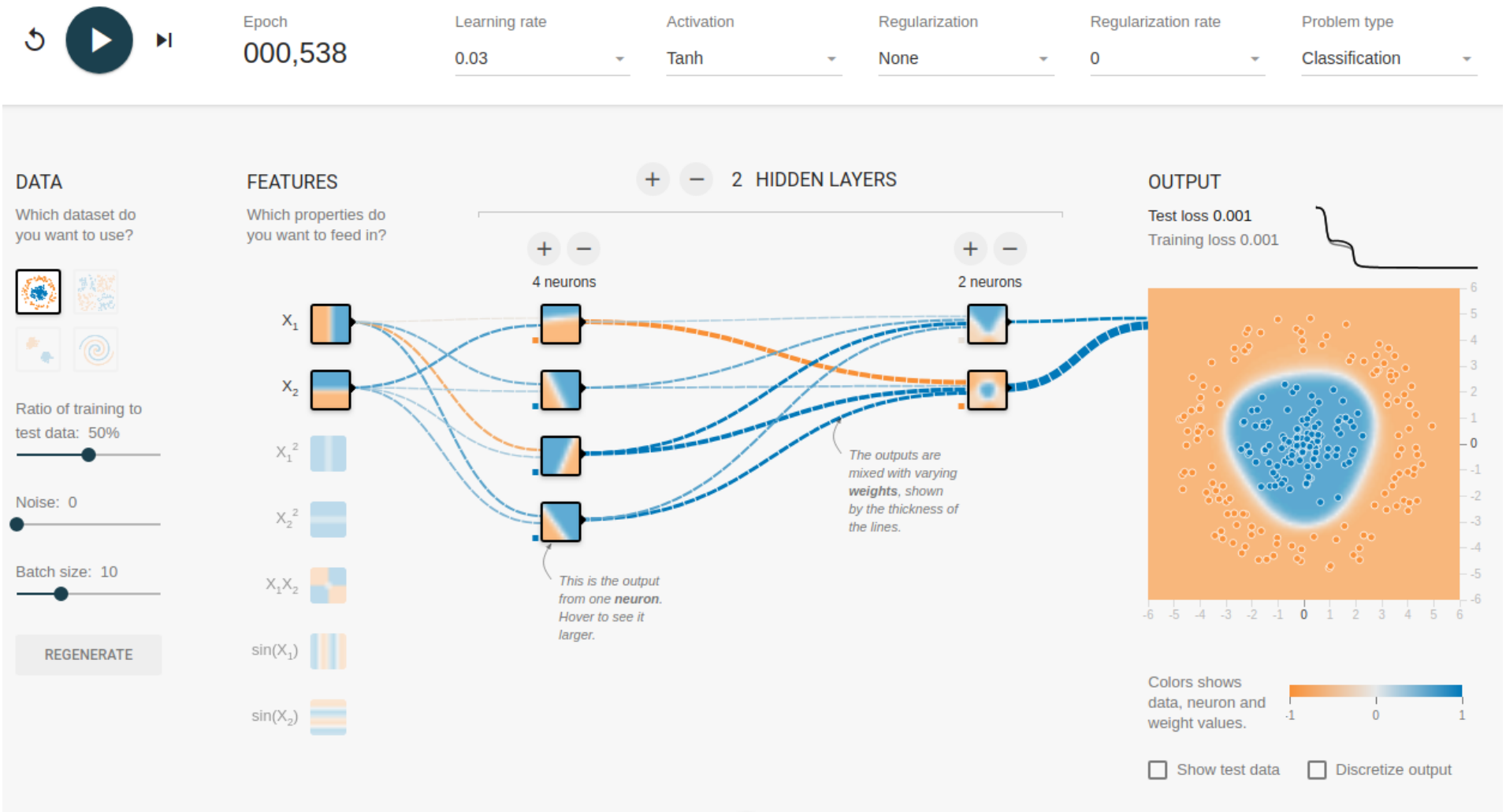
- Example of an MLP in pytorch

```
import torch
from torch.autograd import Variable
# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension. N, D_in, H, D_out = 64, 1000, 100, 10
# Create random Tensors to hold inputs and outputs,
# and wrap them in Variables.
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# Use the nn package to define our model # as a sequence of layers.
model = torch.nn.Sequential(
   torch.nn.Linear(D_in, H),
   torch.nn.ReLU(),
   torch.nn.Linear(H, D_out),
   )
loss_fn = torch.nn.MSELoss(size_average=False)
# Optimizer will update the weights of the model. lr0 = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
for t in range(10):
   # Forward pass: compute predicted y by passing x.
    y_pred = model(x)
    # Compute and print loss.
    loss = loss_fn(y_pred, y)
    print(t, loss.data[0])
    # Optim in two steps
    optimizer.zero_grad()
    # Backward pass: compute gradient of the loss wrt parameters
    loss.backward()
    # Calling the step function on an Optimizer makes an update
    optimizer.step()
```

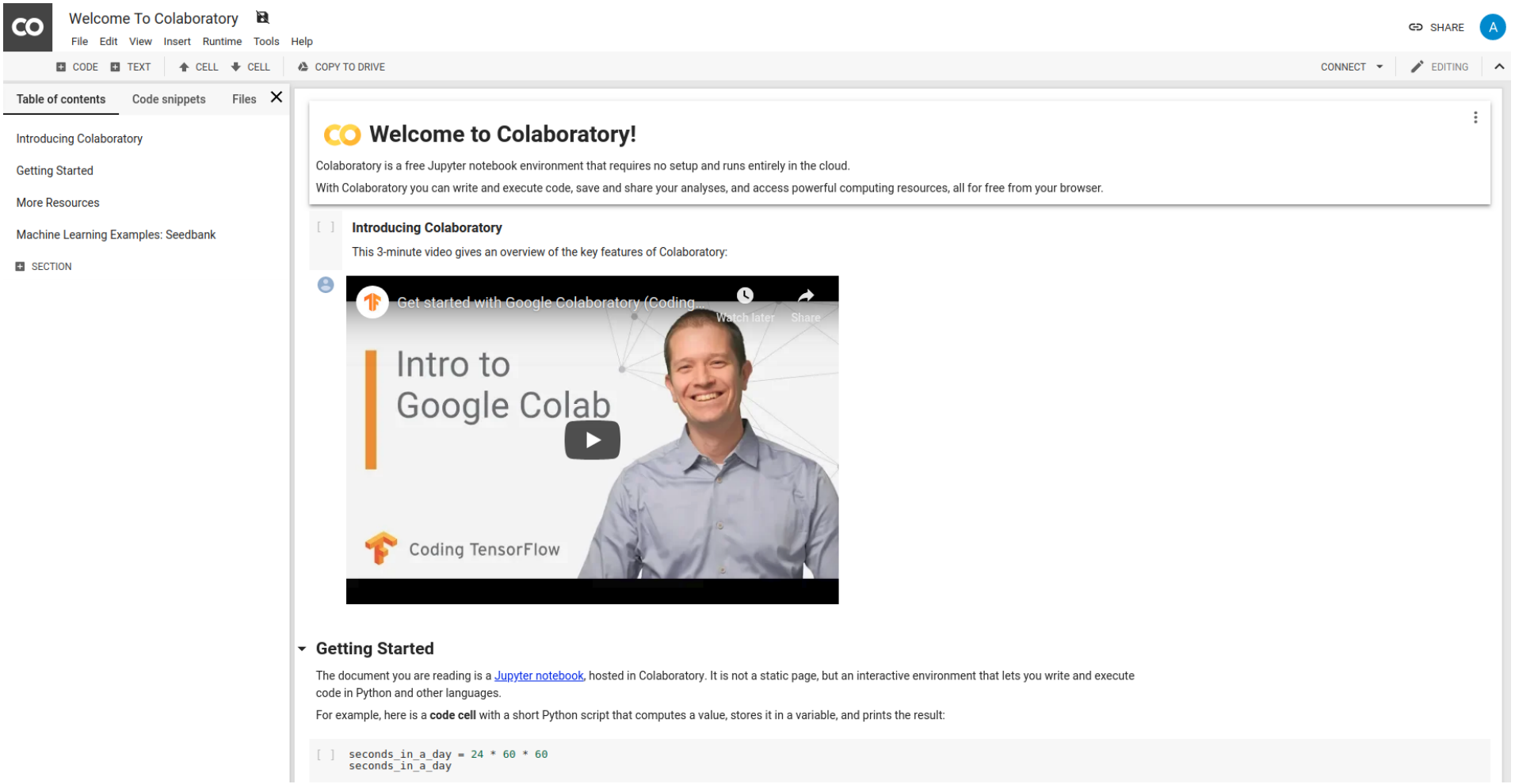
ANN frameworks

- Tensorflow playground : online graphical testing of an mlp



ANN frameworks

- Google colab : free online GPU ressources to train and test neural networks
- Caution : after a certain number of hours, the user is disconnected



Summary

Regularisation

- As you know, machine learning algorithms can be prone to **overfitting**
- A way to avoid this is to **reduce model complexity**
- How is this done ? **Regularisation** of the network via the loss function

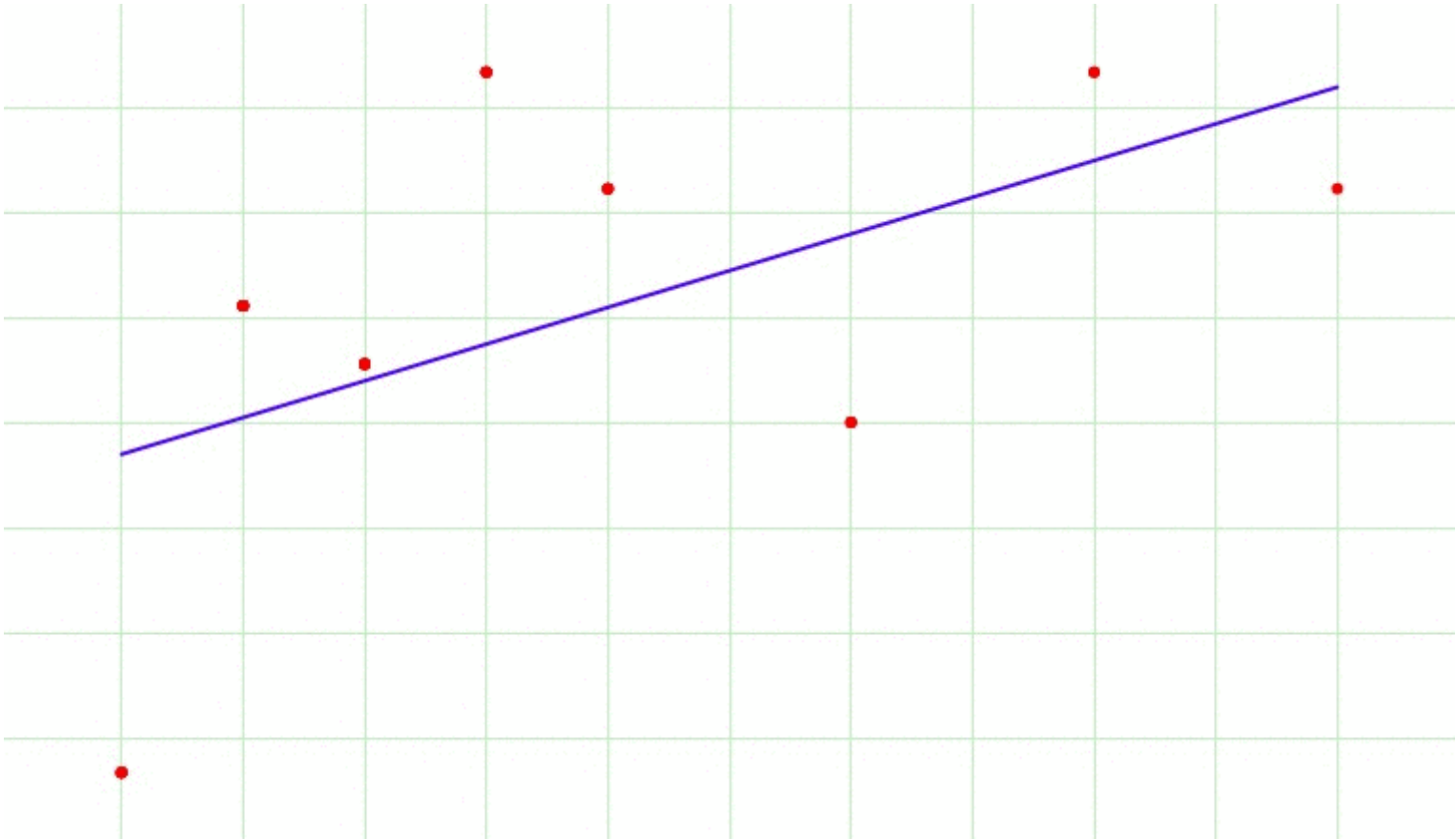
Regularisation

$$J(\theta, \mathbf{y}; \hat{\mathbf{y}}) = \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) + \mathcal{R}(\theta),$$

- Examples:
  - $\mathcal{R}(\theta) = \sum_i \|w_i\|_2^2$ (ℓ_2 regularisation)
  - $\mathcal{R}(\theta) = \sum_i \|w_i\|_1$ (ℓ_1 regularisation)

Regularisation

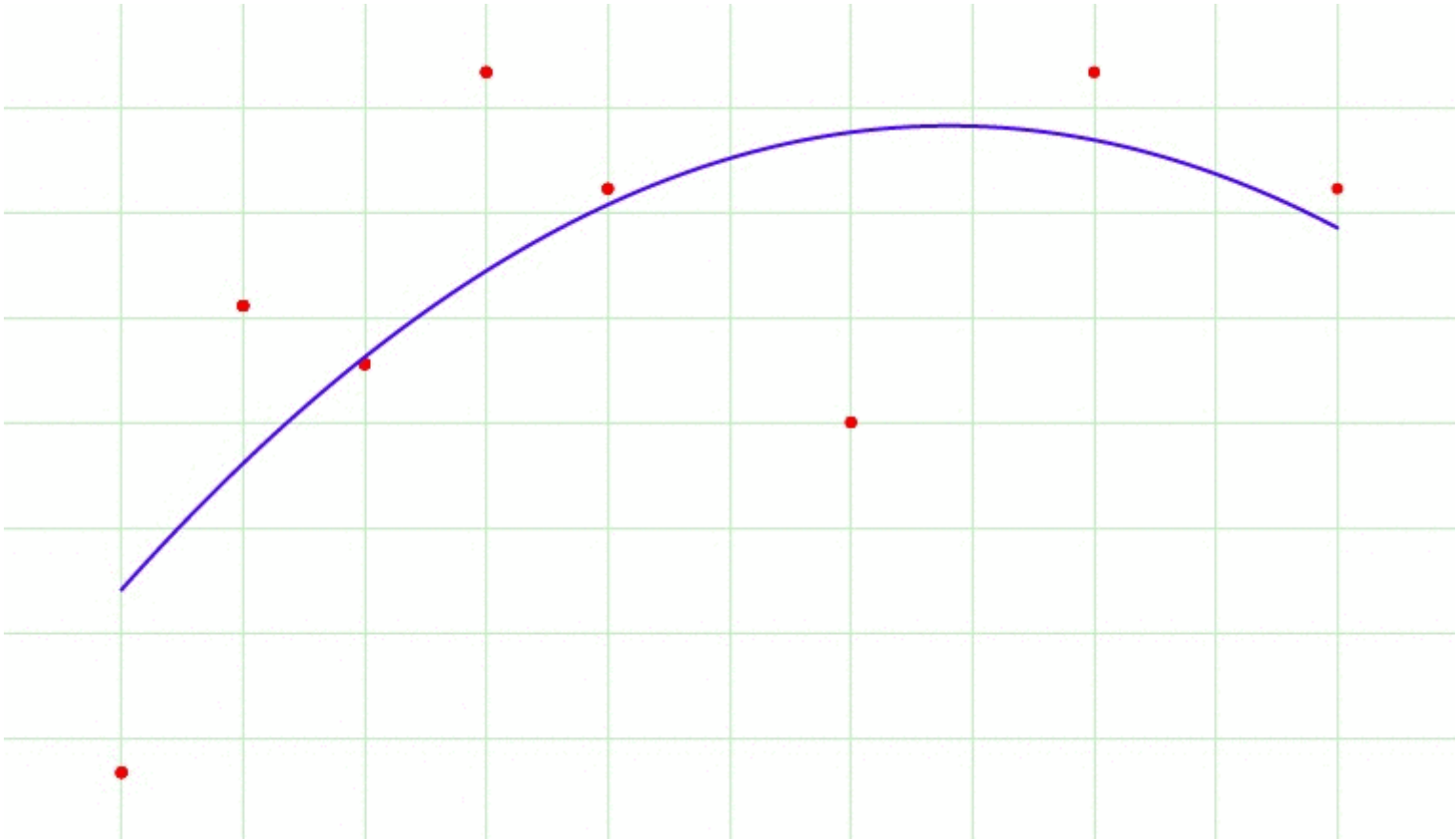
- Why does this reduce model complexity ?



1 parameter

Regularisation

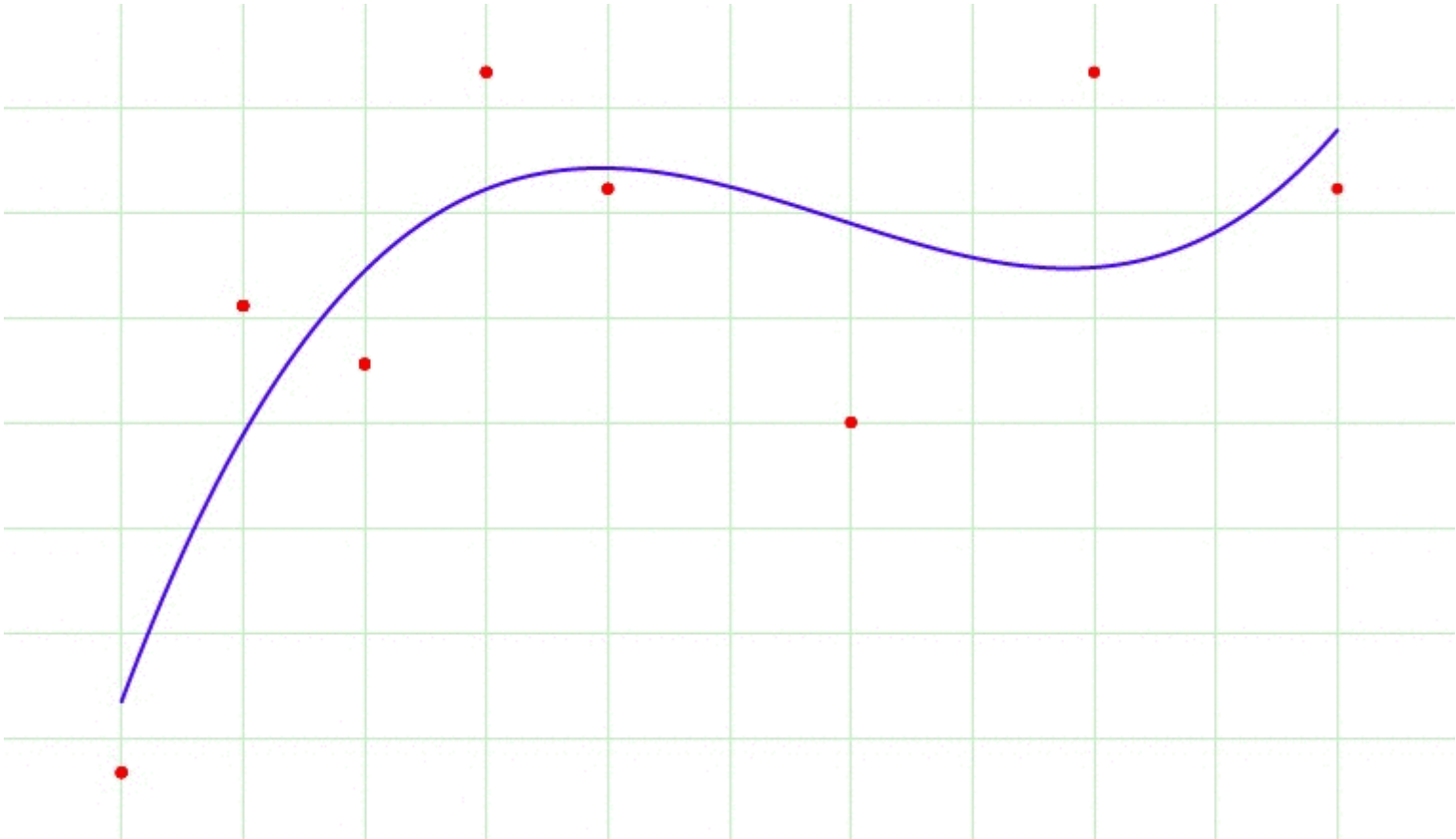
- Why does this reduce model complexity ?



2 parameters

Regularisation

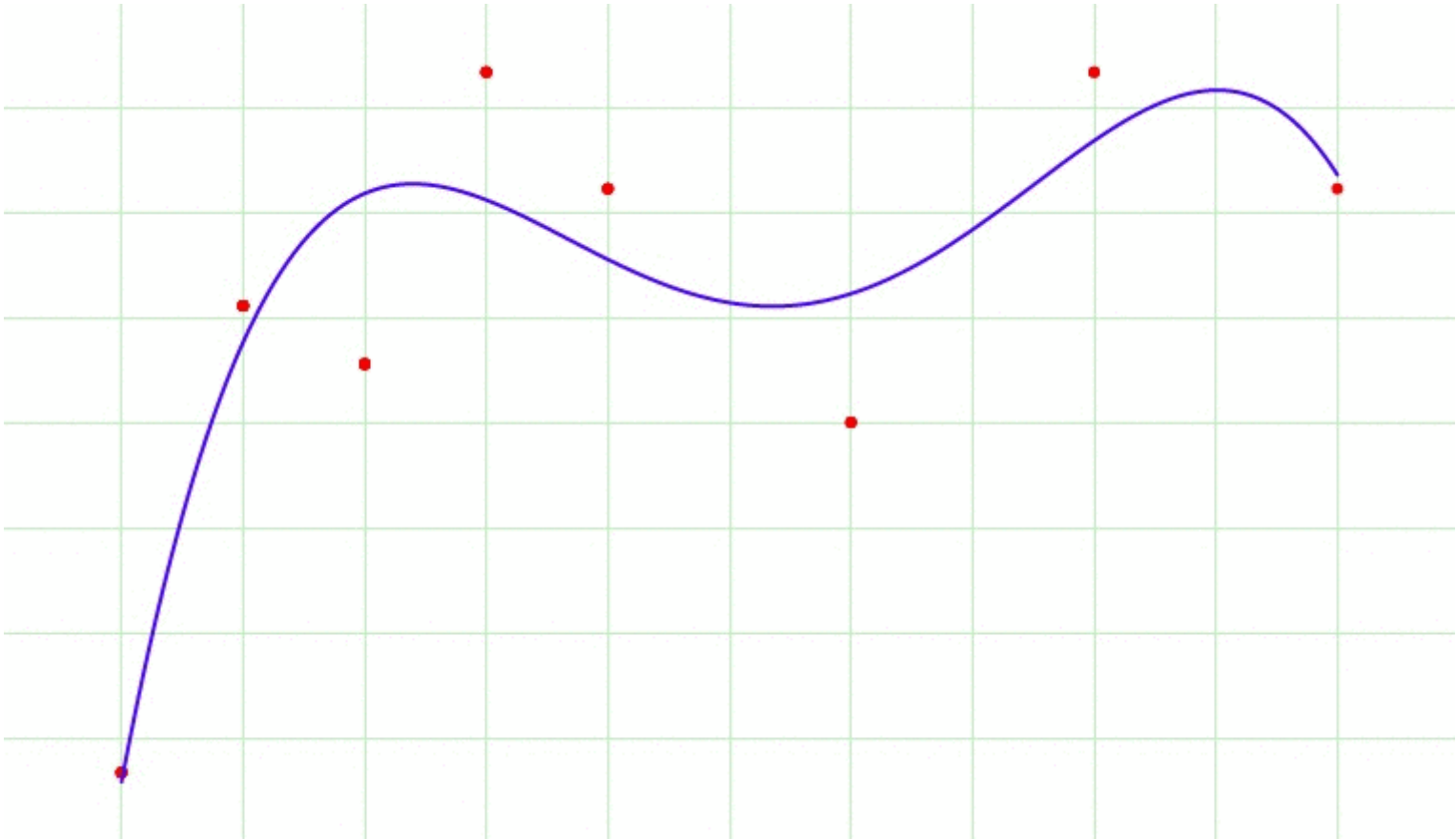
- Why does this reduce model complexity ?



3 parameters

Regularisation

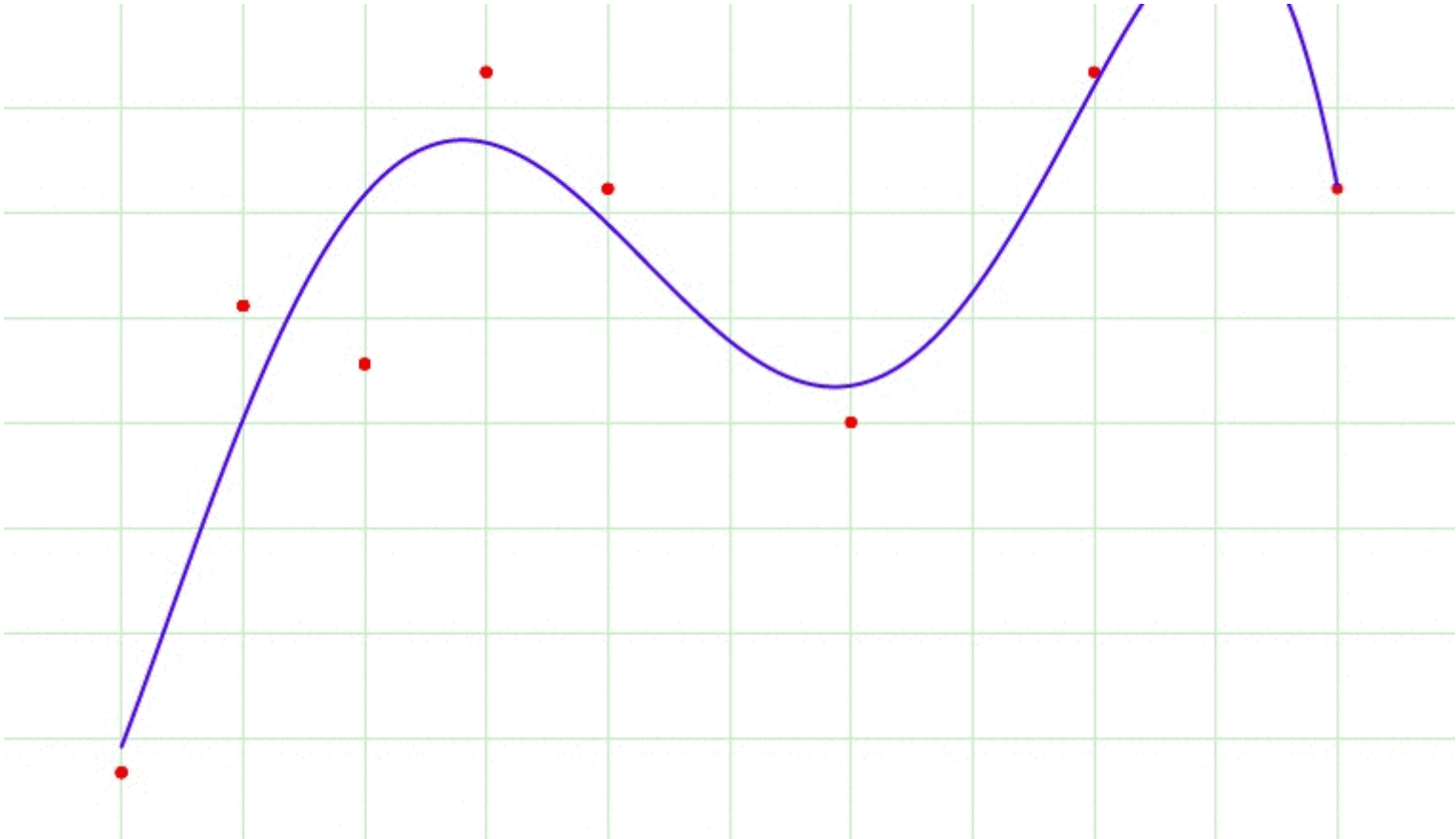
- Why does this reduce model complexity ?



4 parameters

Regularisation

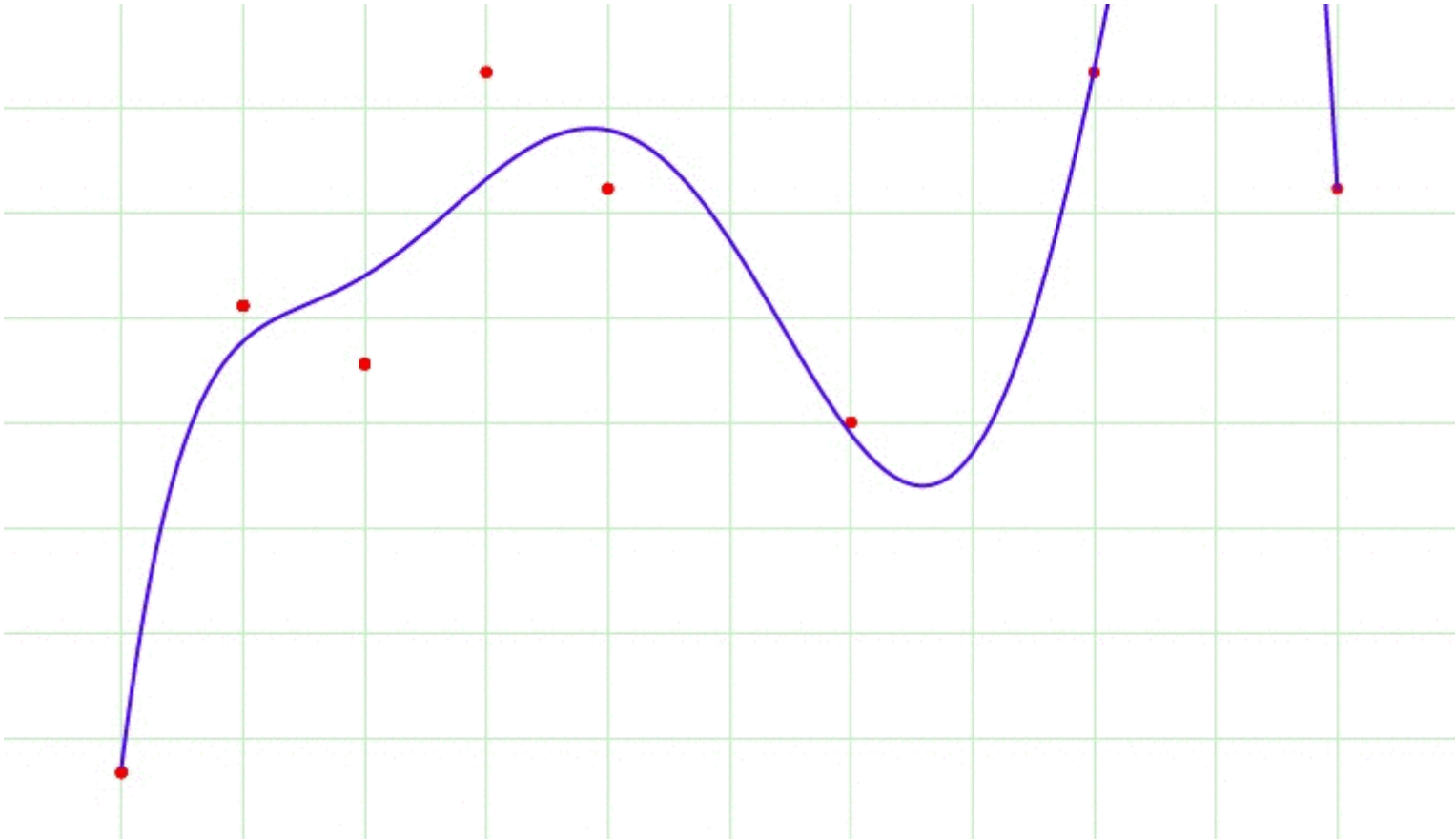
- Why does this reduce model complexity ?



5 parameters

Regularisation

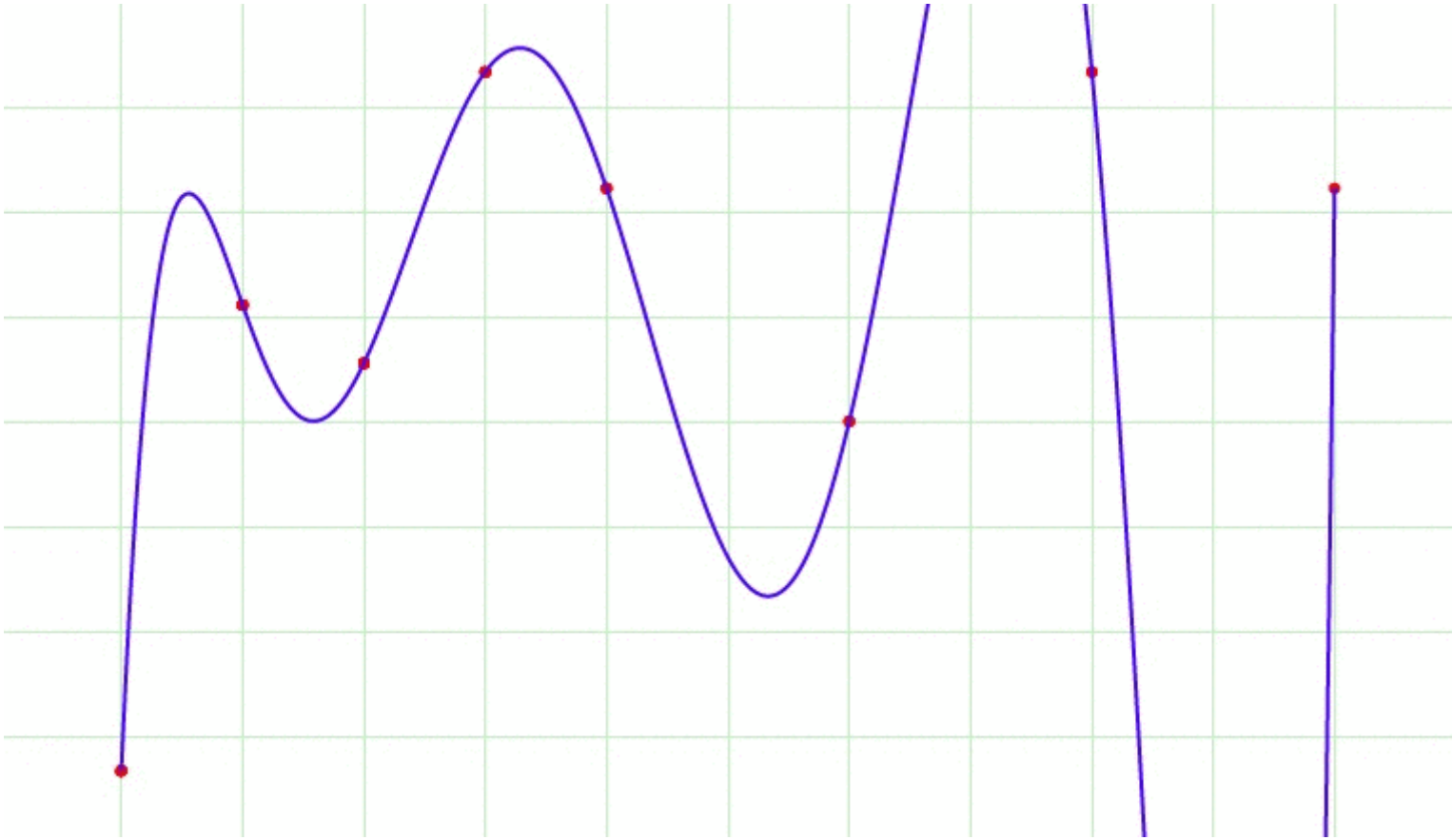
- Why does this reduce model complexity ?



6 parameters

Regularisation

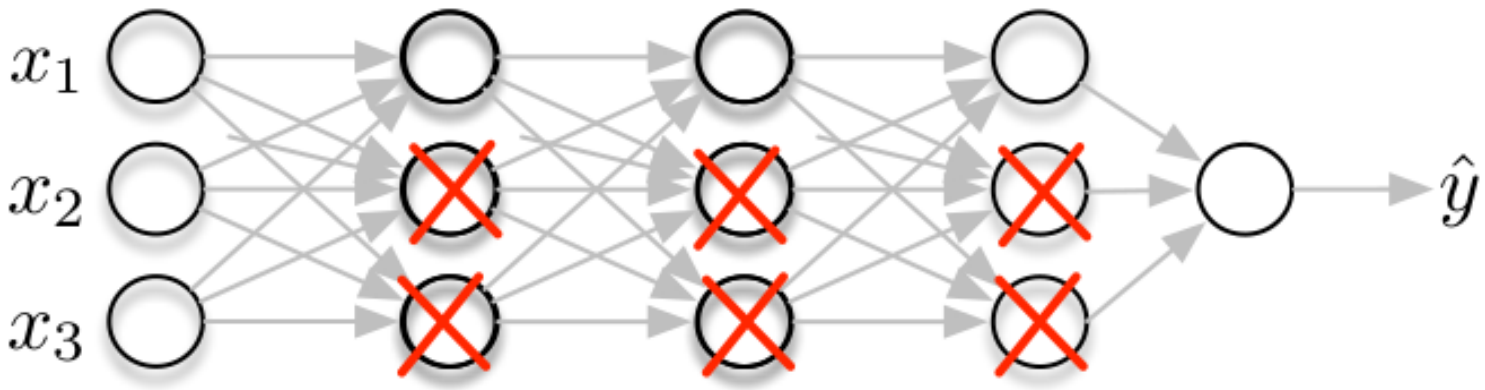
- Why does this reduce model complexity ?



7 parameters

Regularisation

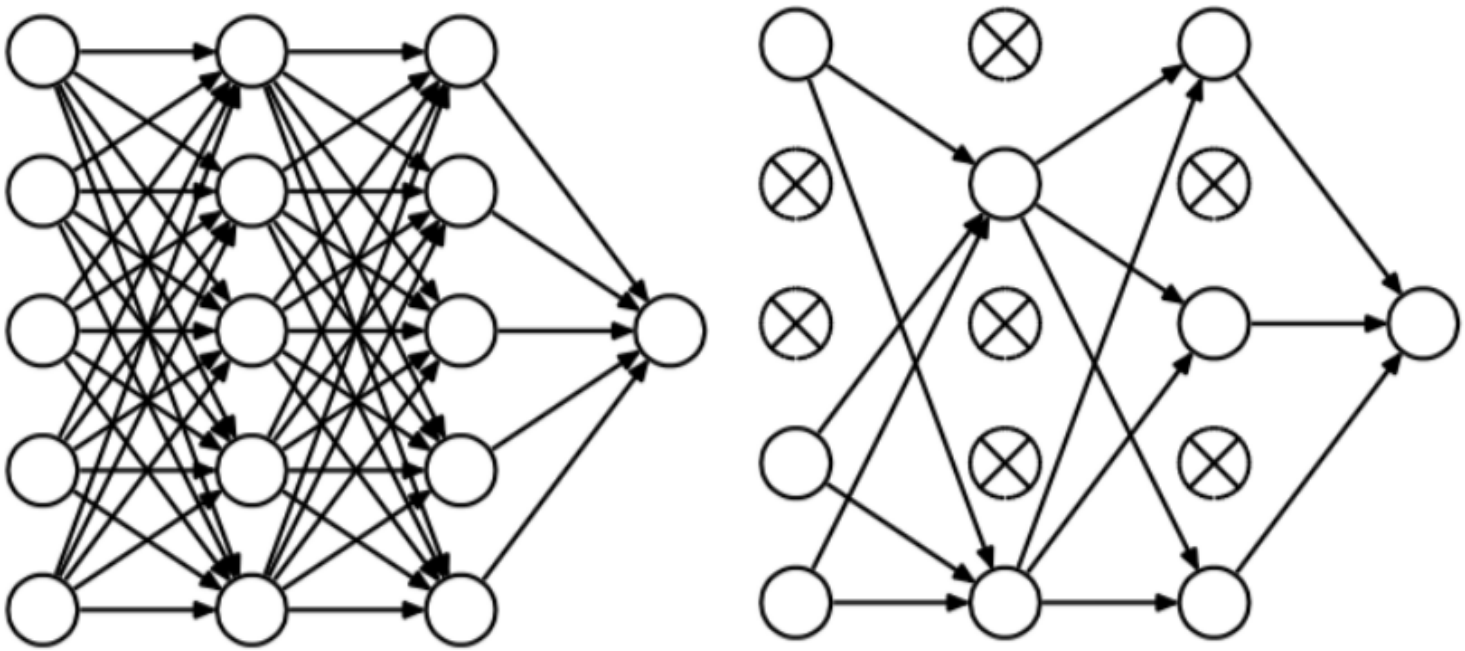
- $\ell - 1$ regularisation encourages *sparse* connections in the network
- $\ell - 2$ regularisation encourages *small* weight values



- Less parameters, less complexity

Regularisation - dropout

- Another approach to regularisation, based on the same principle, is known as **dropout**
- Dropout involves randomly **turning off** neurons. This removes different paths in the network
- The goal is to make the network find robust connections to represent the data, thus avoiding overfitting



Regularisation - dropout

- During **training**, randomly remove neurons with probability p
  - Network cannot rely on one single feature
- During **testing**, no dropout
- This can be added quite easily in various frameworks (Keras, Pytorch etc.)
  - The frameworks take care of storing the batch moments

Regularisation - data augmentation


Car_A_0_345


Car_A_0_1193


Car_A_0_1589


Car_A_0_2933


Car_A_0_3228


Car_A_0_3274


Car_A_0_3614


Car_A_0_3686


Car_A_0_3894


Car_A_0_5212


Car_A_0_5528

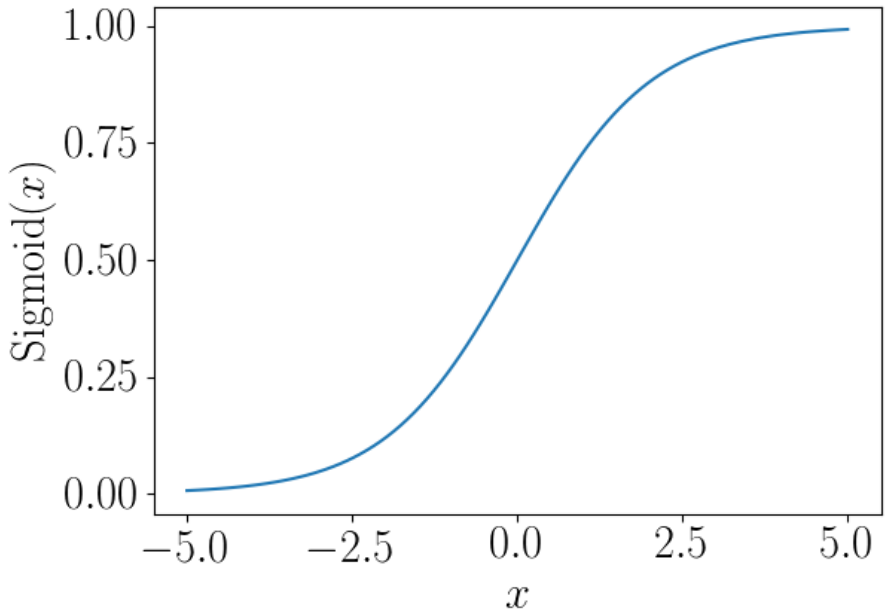

Car_A_0_5574

Weight initialisation

- We need a method to **initialise the weights and biases** of the network
- A straightforward method might be to simply **initialise the parameters to 0**
- However, doing this is in fact a bad idea. Why ?
  - If all the weights have the same value then the udpates will be the same
  - All the hidden units calculate the same value
- This is referred to as having **symmetric weights**
- Solution : make initialisation non-symmetric : **random initialisation**

Weight initialisation

- Another approach: initialise the **weights** randomly, to a **small value**
- Why set it them to a small value ?



- The sigmoid or tanh functions have largest gradients close to 0
- If $|z = wx + b|$ is very large, then the gradient of $\sigma(z = wx + b)$ is small
  - Very small gradient udpates : **slower learning**

Weight initialisation

- A final problem which appears here is known as the **vanishing gradient**
- Let us suppose that the non-linearities are all sigmoids. We have then :

$$\begin{aligned} d\mathbf{z}^{[\ell]} &= d\mathbf{a}^{[\ell+1]} \odot \sigma'(\mathbf{z}^{[\ell]}) \\ &= \mathbf{W}^{[\ell+1]} d\mathbf{z}^{[\ell+1]} \odot \sigma'(\mathbf{z}^{[\ell]}) \\ &= \mathbf{W}^{[\ell+1]} \left(\mathbf{W}^{[\ell+2]} d\mathbf{z}^{[\ell+2]} \odot \sigma'(\mathbf{z}^{[\ell+1]})\right) \odot \sigma'(\mathbf{z}^{[\ell]}) \end{aligned}$$

- This means that as we go further and further up the backpropagation, we multiply the derivative of the sigmoid many times
- But, $\max_{\mathbf{x}}(\sigma'(\mathbf{x})) = \frac{1}{4}$! Therefore, after a while, the gradient updates will become very small
- This is known as the **vanishing gradient** problem
- Solution ? Use **ReLU or Leaky ReLU** in the intermediate layers (they do not have this problem)

Summary

Training algorithms

- All ANN training algorithms are variants of the **gradient descent** algorithm

Gradient descent for ANN parameter updates

$$\theta = \theta - \alpha \nabla_\theta \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$$

- α is the learning rate

- There are many variants on this idea. The two main questions which they address are :
  1. Which samples to use from the (potentially very large) dataset ?
  2. What gradient update to use (simple, with momentum etc.)

Training algorithms - batch gradient descent

- Batch gradient descent only updates parameters after calcualting the gradient on the **entire database**
- Let N be the database size : $\mathbf{X} = \left[\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)} \right]$

Batch Gradient descent for ANN parameter updates

$\theta \leftarrow \text{Initialise}(\theta)$
for $i = 1$ to n **do**
$\theta \leftarrow \theta - \alpha \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \mathcal{L}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)})$
end for

- In this context, an update over the entire dataset is known as an **epoch**

- However, this can be **very costly**
  - One update loops over the **entire database**
  - Modern databases often contain *millions* of samples

Training algorithms - mini-batch gradient descent

- A simple solution is to calculate the gradient over smaller sub-samples, known as **mini-batches**, of the dataset

$$\mathbf{X} = \left[\left[\, \mathbf{x}^{(1)} \dots \mathbf{x}^{(M)}\right], \dots, \left[\, \mathbf{x}^{(N-M+1)} \dots \mathbf{x}^{(N)}\right]\right]$$

Mini-batch gradient descent for ANN parameter updates

$\theta \leftarrow$ Initialise parameters
for $i = 1$ to n **do**
  for $j = 0$ to $\frac{N}{M} - 1$ **do**
    $\theta \leftarrow \theta - \alpha \frac{1}{M} \sum_{k=j*M+1}^{M(j+1)} \nabla_\theta \mathcal{L}(\mathbf{y}^{(k)}, \hat{\mathbf{y}}^{(k)})$
  end for
end for

- Advantage : we can carry out a single update much faster
- Disadvantage : gradient is an approximation

Training algorithms - stochastic gradient descent

- Another approach consists in updating the gradient on each sample, in a randomly chosen order

Stochastic gradient descent for ANN parameter updates

$\theta \leftarrow$ Initialise parameters
for $i = 1$ to n **do**
  Shuffle examples in training dataset
  for $j = 1$ to N **do**
    $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\mathbf{y}^{(j)}, \hat{\mathbf{y}}^{(j)})$
  end for
end for

- Advantage : frequent updates
- Disadvantages : gradient is likely to be noisy

Training algorithms - momentum

- **Momentum** is a term added to the gradient update which helps stabilise training, making the gradient less erratic
- A fraction β of the gradient at the previous timestep : a momentum

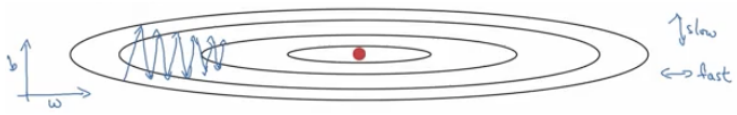
Gradient descent with momentum

$\theta \leftarrow$ Initialise parameters
$d\theta = 0$
for $i = 1$ to n **do**
  $d\theta \leftarrow \beta d\theta + (1 - \beta)\nabla_\theta \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$
  $\theta \leftarrow \theta - \alpha d\theta$
end for

- Common parameter choice $\beta = 0.9$
- Reduces oscillations of outlier gradients, reinforces contribution of samples which give a similar gradient direction

Training algorithms - RMSprop

- If the loss function surface is not isotropic (changes faster in one direction), then optimisation can oscillate
- Idea : **normalise** the learning rate by the recent magnitude of the gradient. This is known as the **RMSprop** algorithm



RMSprop

$\theta \leftarrow$ Initialise parameters
for $i = 1$ to n **do**
  $S_\theta \leftarrow \beta S_\theta + (1-\beta)\left(\nabla_\theta \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})\right)^2$
  $\theta \leftarrow \theta - \alpha \frac{\nabla_\theta \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\sqrt{S_\theta + \epsilon}}$
end for

- S_θ is an estimation of the squared magnitude of the recent gradient
- Adaptive learning rate dampens oscillations

Training algorithms - Adam

- Adam : Adaptive Moment Estimation
- Combination of **momentum and RMSprop** algorithms : both first and second moments of the recent gradients

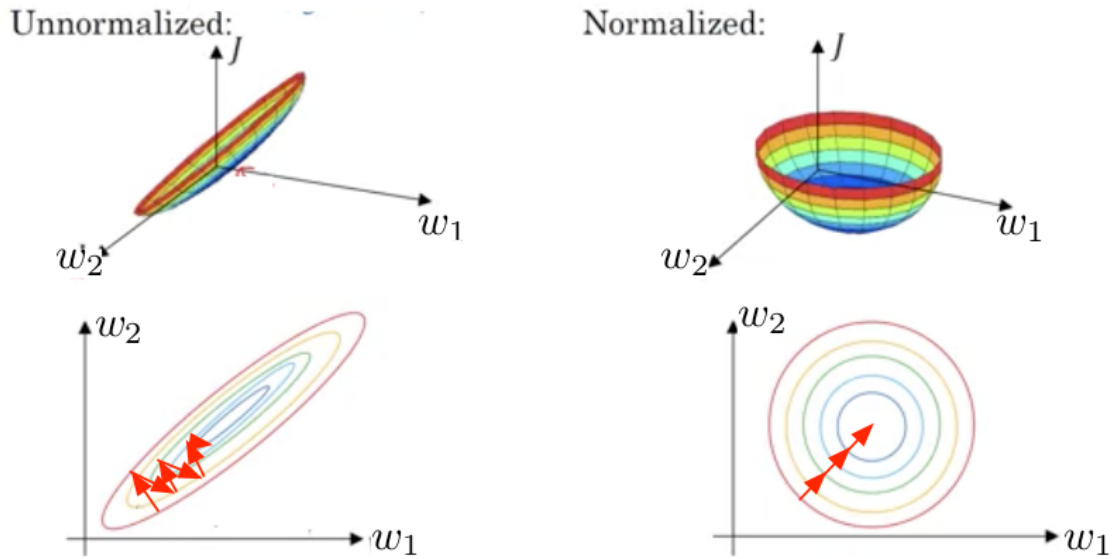
Adam

$\theta \leftarrow$ Initialise parameters
for $i = 1$ to n **do**
$d_\theta \leftarrow \beta_1 d_\theta + (1-\beta_1)\nabla_\theta \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$
$S_\theta \leftarrow \beta_2 S_\theta + (1-\beta_2)\left(\nabla_\theta \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})\right)^2$
$\widehat{d_\theta} \leftarrow \frac{d_\theta}{1-(\beta_1)^t}$ *Unbiased estimator of first moment*
$\widehat{S_\theta} \leftarrow \frac{S_\theta}{1-(\beta_2)^t}$ *Unbiased estimator of second moment*
$\theta \leftarrow \theta - \alpha \frac{\widehat{d_\theta}}{\sqrt{\widehat{S_\theta}+\epsilon}}$
end for

- Adam optimiser is quite robust, **one of the most commonly used**
- Common parameters : $\alpha = 0.0002$, $\beta_1 = 0.5$, $\beta_2 = 0.999$

Training algorithms - normalising the inputs

- Data normalisation is a technique to control the mean and standard deviation of the input data
- Why is this important ? Without normalisation, the magnitude of different parameters may need to be very different
  - For example, if $\mathbf{x}_1 \in [-0.001, 0.001]$ and $\mathbf{x}_2 \in [-100, 100]$, weights will need to be very different
  - This means that a very small learning parameter will need to be chosen : slow convergence

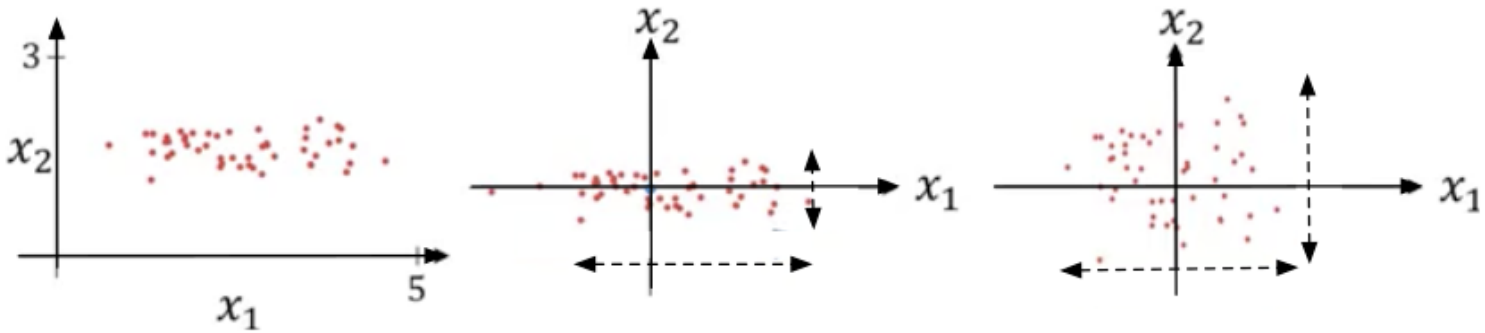


Training algorithms - normalising the inputs

- Data normalisation is quite simple, simply calculate mean and standard deviation of data, and impose the desired value (in general 0 and 1, respectively)

$$\mu = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}^{(i)} \qquad \sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}^{(i)})^2$$

$$\mathbf{x}^{(i)} \leftarrow \frac{1}{\sigma}(\mathbf{x}^{(i)} - \mu)$$



Training algorithms - batch normalisation

- Normalisation can be carried out on **intermediate network layers** as well. This is known as **batch normalisation**
- Objective of batch normalisation : normalise inputs to a (or all) layer(s), to reduce **internal covariate shift**
- In general, machine learning algorithms suppose a certain probability distribution
- However, during training, the distributions can change, due to changes in network parameters. This change is known as covariate shift

*

* From https://blog.bigml.com/2014/01/03/simple-machine-learning-to-detect-covariate-shift/

Training algorithms - batch normalisation

Batch normalisation

$$\mu \leftarrow \frac{1}{M}\sum_i \mathbf{z}^{(i)}$$
$$\sigma^2 \leftarrow \frac{1}{M}\sum_i (\mathbf{z}^{(i)})^2$$
$$\mathbf{z}^{(i)}_{norm} \leftarrow \frac{\mathbf{z}^{(i)}-\mu}{\sqrt{\sigma^2+\epsilon}}$$
$$\mathbf{z}^{(i)} \leftarrow \gamma \mathbf{z}^{(i)}_{norm} + \beta$$

- γ and β are new parameters to be learned
- The training can modify the average and standard deviation of the intermediate layers

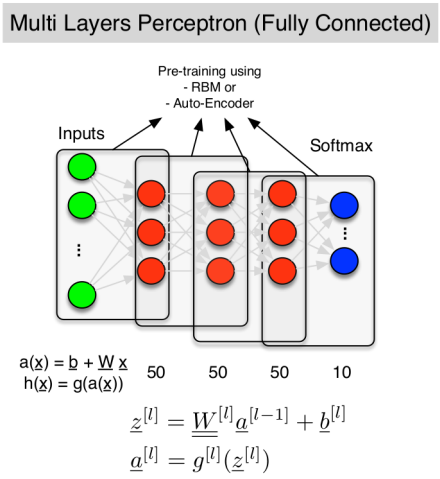
- Often, the batch normalisation is applied **before applying a non-linearity**
- However, this is subject to debate in the community

Summary

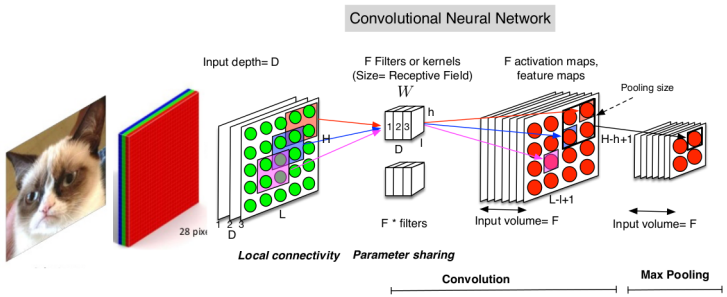
- We have seen the Multi-Layer Perceptron and in general ANNs
- The training of an ANN can be done using the **backpropagation** algorithm
- Several regularisation techniques exist which avoid **over-fitting** of the network
- Various optimisation algorithms are used, **all based on gradient descent**

Summary

- Next week : TP on MLPs (implementation in Python and Tensorflow)



MLP



CNN