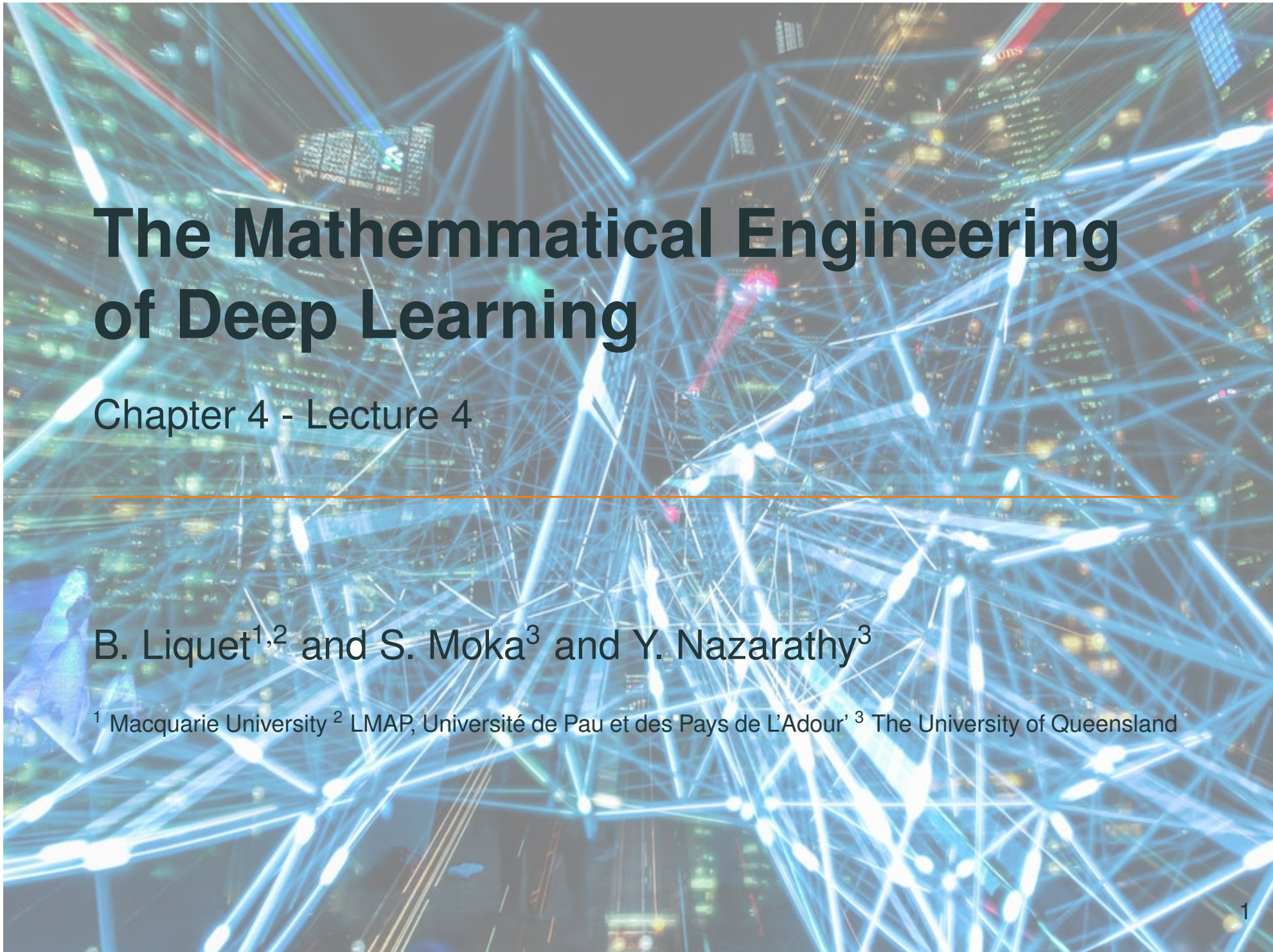


The Mathemmatical Engineering of Deep Learning

Chapter 4 - Lecture 4

B. Liquet[1,2] and S. Moka[3] and Y. Nazarathy[3]

[1] Macquarie University [2] LMAP, Université de Pau et des Pays de L'Adour' [3] The University of Queensland

Outline of Lecture 1/3

- Review: shallow Neural Network
- Full Neural Network
- Matrix Representation
- Activation Function and Derivative
- Backpropagation on a simple example

Outline of Lecture 2/3

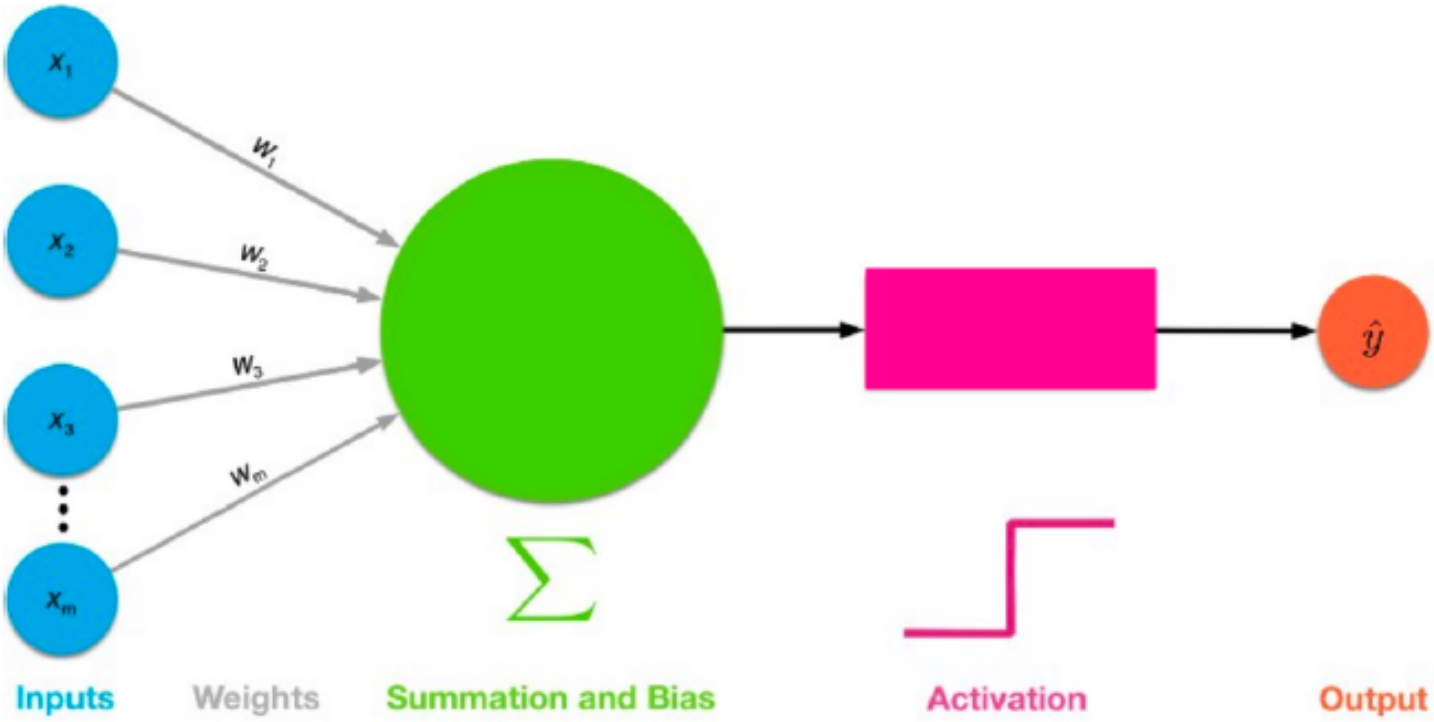
- Backpropagation on a DNN
- How to compute derivative?: Automatic differentiation
- Approximation Properties of Multilayer Perceptrons
- Weight initialization

Outline of Lecture 3/3

- Regularization
- Dropout
- Batch-Normalization
- About vanishing gradients

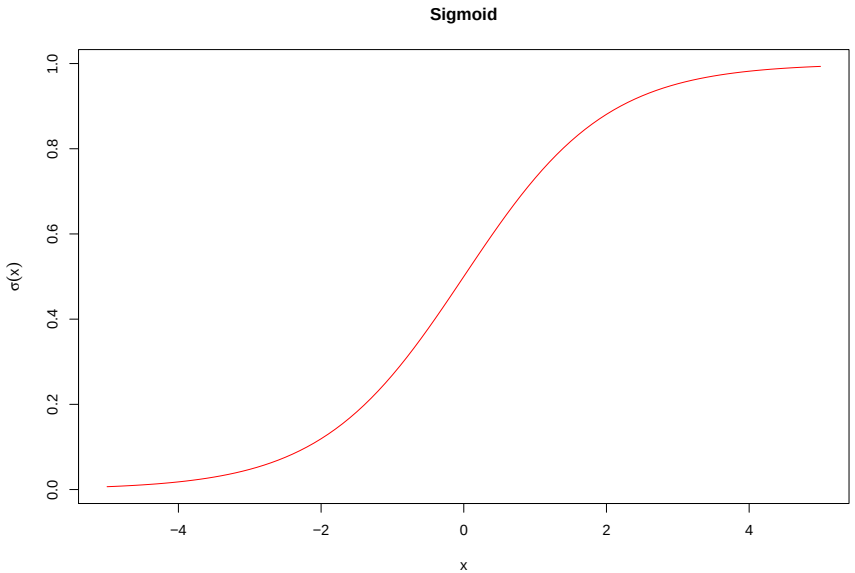
Review: The formal Neuron (1943) [1]

- Mapping from input x to output y
  - Linear (affine) mapping: $z = w^T x + b$ (linear model)
  - Non-linear activation function $\sigma(\cdot) \rightarrow \widehat{y} = \sigma(z)$



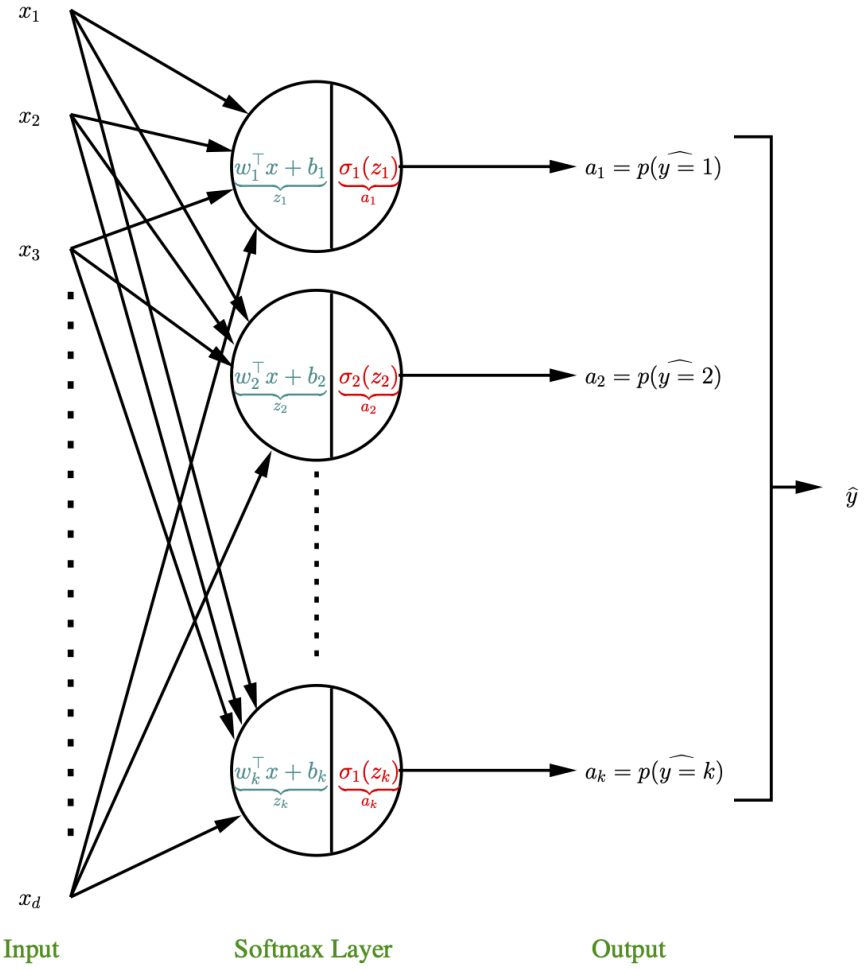
Formal Neuron: classical models

- **Regression Task** ($y \in \mathfrak{R}$).
  - Identity function, $\sigma(x) = x$, apply on the linear predictor $x^T w$
  - Equivalent to the Linear model: $E[Y|x] = x^T w$
- **Binary Classification Task** ($y \in \{0, 1\}$):
  - Sigmoid function, $\sigma(x) = \frac{1}{1+e^{-z}}$, apply on the linear predictor $x^T w$.
  - Equivalent to the logistic model: $P[Y = 1|x] = \frac{x^T w}{1+exp(x^T w)}$



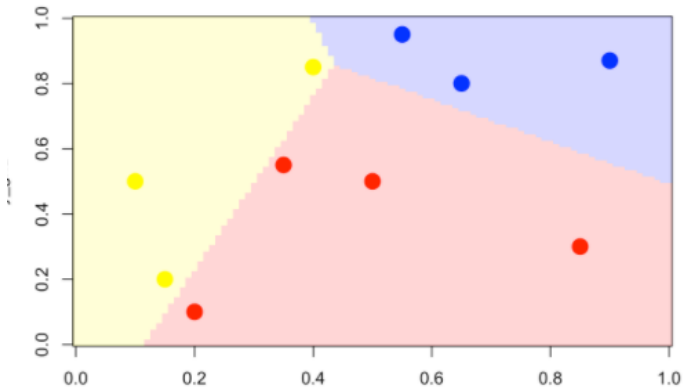
Formal Neuron: extension

- **Multi-class Classification** ($y \in \{1, \ldots, K\}$)
  - concatenation of K formal neurons
  - activation function called “softmax”

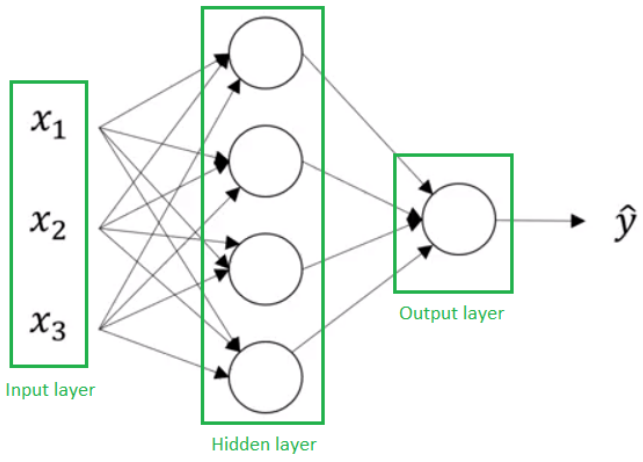


Beyond Linear Classification

- Logistic and Softmax models produce linear boundaries

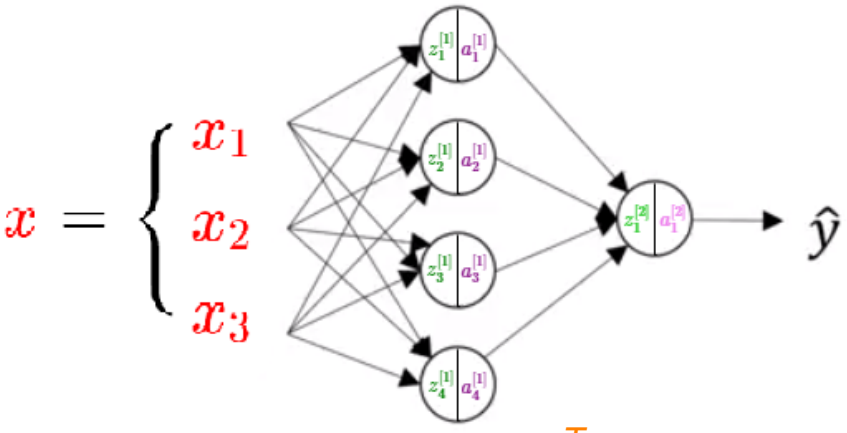


- Solution: add a layer



A short Demo

One hidden Layer Neural Network



$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]} \qquad a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]} \qquad a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]} \qquad a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]} \qquad a_4^{[1]} = \sigma(z_4^{[1]})$$

- output layer is defined by:

$$z_1^{[2]} = w_1^{[2]T} a^{[1]} + b_1^{[2]} \qquad a_1^{[2]} = \sigma(z_1^{[2]})$$

Matrix Notation

- The superscript number $^{[i]}$ for denoting the **layer number** and the subscript number $_j$ denotes the **neuron number** in a particular layer
- x is the input vector consisting of 3 features.
- $w_j^{[i]}$ is the **weight vector** associated with neuron j present in the layer i
- $b_j^{[i]}$ is the **bias scalar** associated with neuron j present in the layer i.
- $z_j^{[i]}$ is the **intermediate output** associated with neuron j present in the layer i.
- $a_j^{[i]}$ is the **final output** associated with neuron j present in the layer i.
- As an example $\sigma(\cdot)$ is the **sigmoid activation function**

Forward-propagation equations

$$\begin{cases} z^{[1]} = & \mathbf{W}^{[1]} x + b^{[1]} \\ a^{[1]} = & \sigma(z^{[1]}) \\ z^{[2]} = & \mathbf{W}^{[2]} a^{[1]} + b^{[2]} \\ \widehat{y} = & a^{[2]} = \sigma(z^{[2]}) \end{cases}$$

where

$$\mathbf{W}^{[1]} = \begin{bmatrix} - & w_1^{[1]T} & - \\ - & w_2^{[1]T} & - \\ - & w_3^{[1]T} & - \\ - & w_4^{[1]T} & - \end{bmatrix} \qquad b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} \qquad z^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} \qquad a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$

Why non-linear Activation is important

Consider this neural network without activation functions:

$$\begin{cases} z^{[1]} = W^{[1]}x + b^{[1]} \\ \widehat{y} = \qquad\qquad\qquad z^{[2]} = W^{[2]}z^{[1]} + b^{[2]} \end{cases}$$

Then, it follows

$$\begin{cases} z^{[1]} = W^{[1]T}x + b^{[1]} \\ \widehat{y} = z^{[2]} = W^{[2]}W^{[1]}x + W^{[2]}b^{[1]} + b^{[2]} \\ \widehat{y} = z^{[2]} = W_{new}x + b_{new} \end{cases}$$

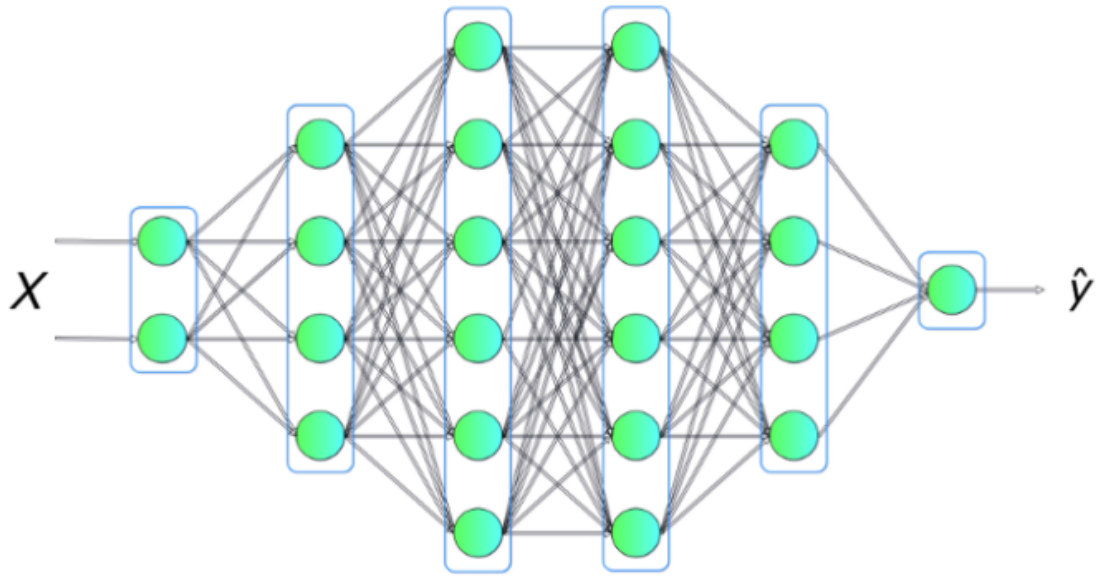
- The output is then a linear combination of a new weight matrix, input and a new bias.
- **Identity activation function**: NN will output linear output of the input.
- Composition of two linear functions is a linear function.
- Linear activation function is generally used for the **output layer** in case of **regression**.

Deep Learning: N layers Neural Network

- The elementary bricks of **deep learning** are the neural networks, that are combined to form the deep neural networks
- Deep learning architectures are based on **deep cascade of layers**.
- Several types of architectures:
  - The **multilayer perceptrons**, that are the oldest and simplest ones
  - The **Convolutional Neural Networks (CNN)**, particularly adapted for image processing
  - The **recurrent neural networks**, used for sequential data such as text or times series.

Multi-layer fully-connected; Multi-Layer Perceptron; Feed-forward Neural Networks

- **Multilayer network**: Cascade of multiple layers, each of which is a nonlinear transformation.
- A multilayer network consisting of fully connected layers is called a **multi-layer perceptron**
- The units are connected together into a **directed acyclic graph** which gives a **feed-forward neural network**



Multi-layer fully-connected neural networks

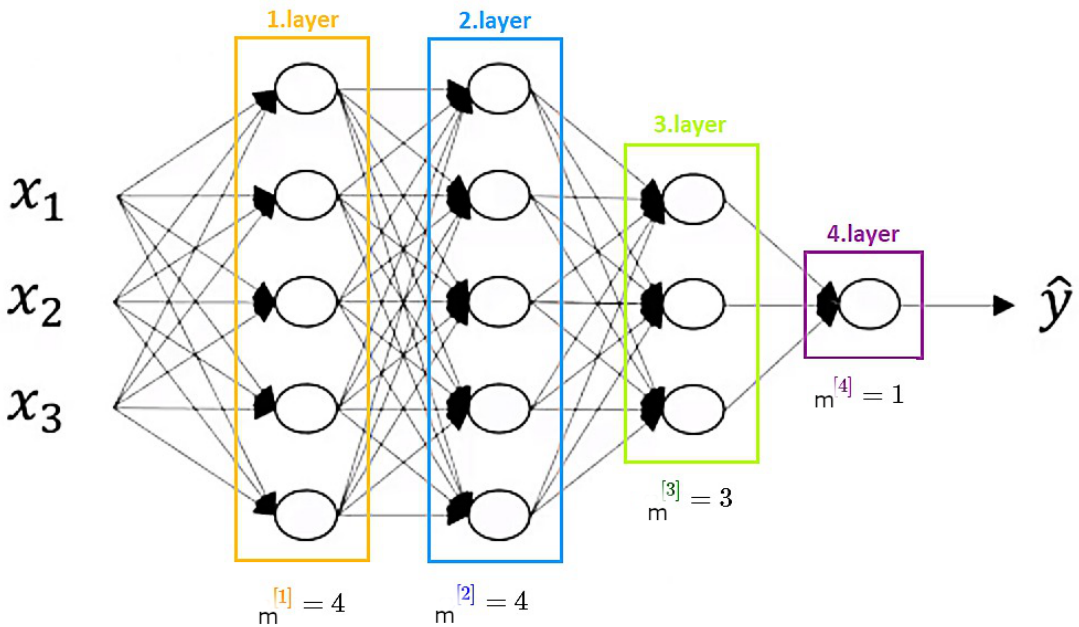
Forward pass equation

$$\begin{cases} a^{[1]} = & g^{[1]}(W^{[1]}x + b^{[1]}) \\ a^{[2]} = & g^{[2]}(W^{[2]}a^{[1]} + b^{[2]}) \\ \dots = & \dots \\ a^{[r-1]} = & g^{[r-1]}(W^{[r-1]}a^{[r-2]} + b^{[r-1]}) \\ \widehat{y} = a^{[r]} = & g^{[r]}(W^{[r]}a^{[r-1]} + b^{[r]}) \end{cases}$$

- r layers based on r weight matrices $W^{[1]}, \dots, W^{[r]}$
- r bias vectors $b^{[1]}, \dots, b^{[r]}$.
- r activation functions noted $g^{[r]}$ which might **be different for each layer** r.
- The number of neurons in each layer could be also be not equal (noted m_r)

How do we count layers in a Deep Neural Network?

When counting layers in a neural network we count hidden layers as well as the output layer, but we don't count an input layer.



It is a ?? layer neural network with ?? hidden layers.

Vectorizing Across Multiple Training Examples

- Consider m training samples $x^{[1]}, \ldots, x^{[m]}$
- Thus m predictions $x^{(i)} \longrightarrow a^{[2](i)} = \widehat{y} \quad i = 1, \ldots m$
- Define the matrices $\mathbf{X}$, $\mathbf{Z}^{[1]}$ and $\mathbf{A}^{[1]}$:

$$\mathbf{X} = \begin{bmatrix} | & | & \ldots & | \\ x^{(1)} & x^{(2)} & \ldots & x^{(m)} \\ | & | & \ldots & | \end{bmatrix}, \qquad \mathbf{Z}^{[1]} = \begin{bmatrix} | & | & \ldots & | \\ z^{[1](1)} & z^{[1](2)} & \ldots & z^{[1](m)} \\ | & | & \ldots & | \end{bmatrix}$$

$$\mathbf{A}^{[1]} = \begin{bmatrix} | & | & \ldots & | \\ a^{[1](1)} & a^{[1](2)} & \ldots & a^{[1](m)} \\ | & | & \ldots & | \end{bmatrix}$$

$$A^{[1]} = \begin{bmatrix} 1^{st}\,unit\ \ of\ \ 1.tr.example & \ldots & 1^{st}\,unit\ \ of\ \ m^{th}.tr.example \\ 2^{nd}\,unit\ \ of\ \ 1^{st}\,tr.example & \ldots & 2^{nd}\,unit\ \ of\ \ m^{th}\,tr.example \\ the\ \ last\ \ unit\ \ of\ \ 1^{st}\,tr.example & \ldots & the\ \ last\ \ unit\ \ ofm^{th}\,tr.example \end{bmatrix}$$

Forward equation using matrix notation

Based on this matrix representation we get:

$$\begin{cases} Z^{[1]} = & W^{[1]}\mathbf{X} + b^{[1]} \\ A^{[1]} = & \sigma(Z^{[2]}) \\ Z^{[2]} = & W^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} = & \sigma(Z^{[2]}) \end{cases}$$

Added $b^{[1]} \in \Re^{4\times 1}$ to $W^{[1]}\mathbf{X} \in \Re^{4\times m}$ is strictly not allowed following the rules of linear algebra. By defining

$$\widetilde{b}^{[1]} = \begin{bmatrix} | & | & \dots & | \\ b^{[1]} & b^{[1]} & \dots & b^{[1]} \\ | & | & \dots & | \end{bmatrix}.$$

we can compute:

$$Z^{[1]} = W^{[1]}\mathbf{X} + \widetilde{b}^{[1]}$$

Dimension Summary of the components

- Layer 1
  - dim of $\mathbf{W}^{[1]}$
  - dim of $b^{[1]}$
  - dim of $Z^{[1]}$
  - dim of $A^{[1]}$
- Layer r
  - dim of $\mathbf{W}^{[r]}$
  - dim of $b^{[r]}$
  - dim of $Z^{[r]}$
  - dim of $A^{[r]}$
- Layer R
  - dim of $\mathbf{W}^{[R]}$
  - dim of $b^{[R]}$
  - dim of $Z^{[R]}$
  - dim of $A^{[R]}$

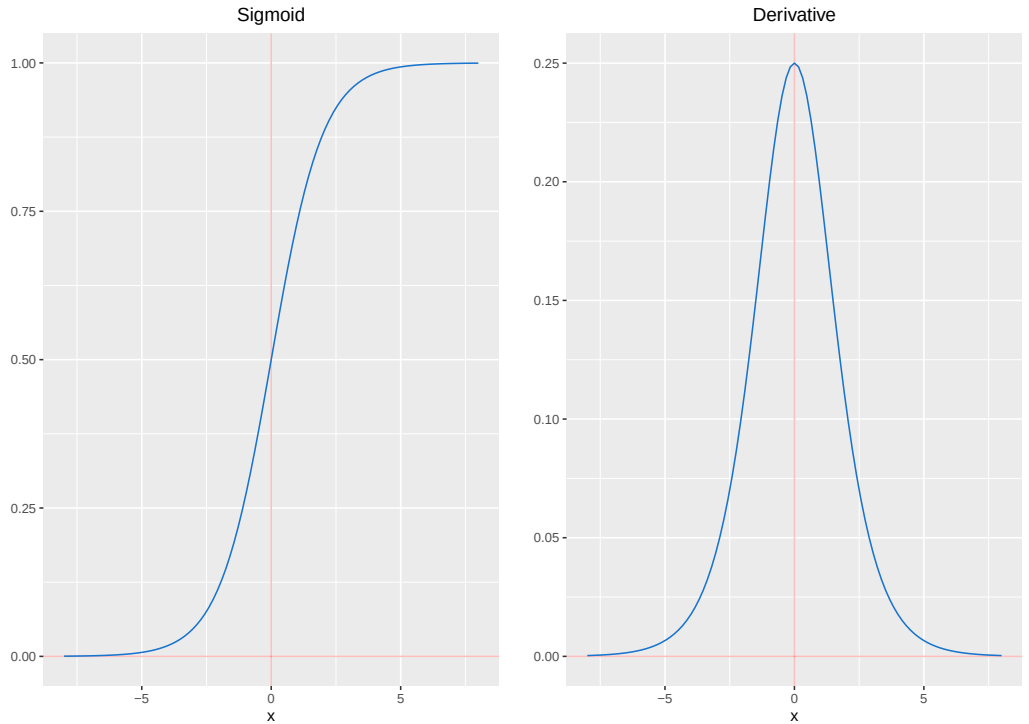
Need Activation Functions

- **Linear activation function** does not help to represent a nonlinear mapping between input and output. **Linear activation function** is mainly used for regression task in the **output Layer**
- **Non-linear Activation function** are non-linear differential functions. Nonlinear activation functions are mainly used in the hidden layers and in output layer depending the task.
- The choice of activation function is determined by the **nature of the data** and the assumed distribution of target variables.
    - For binary classification: **sigmoid** activation for the output layer
    - Multiclass classification task: **softmax** activation for the output layer
- Popular activations functions are: **sigmoid**, **ReLU**, **Soft ReLU**, **Hard Threshold**, **Hyperbolic Tangent**

Sigmoid function

- Historically, the **sigmoid** was the mostly used activation function
- The **sigmoid activation** saturates at either tail with a value of 0 or 1.
- Thus gradient is almost zero → make the gradient *vanish* and no signal will flow through the corresponding neuron.

$$\sigma(z) = g(z) = \frac{1}{1 + e^{-z}} \qquad \frac{d}{dz}\sigma(z) = \sigma(z)(1 - \sigma(z))$$

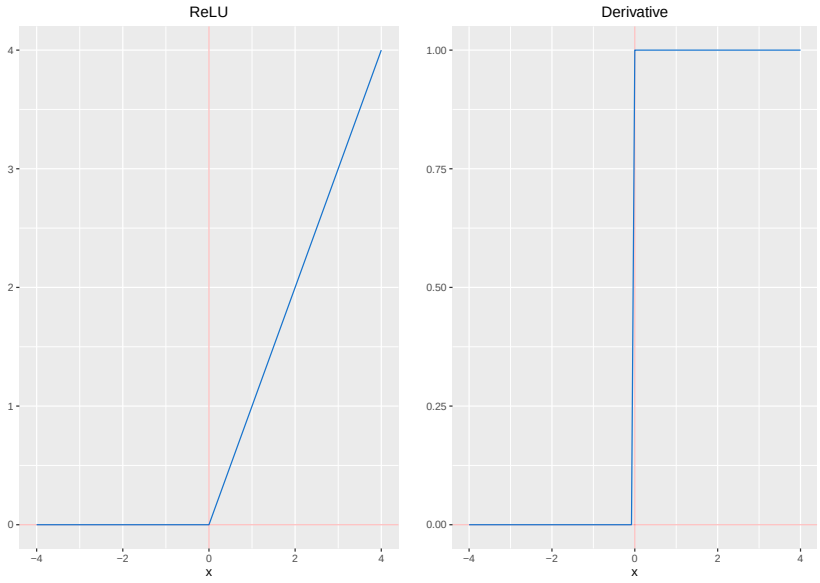


ReLU function

Rectified Linear Units is very popular. It is not linear and provides the same benefits as Sigmoid but with better performance.

$$ReLU(z) = \max(0, z)$$

$$\frac{d}{dz} ReLU(z) = \begin{cases} 1 \ \ if \ \ z > 0 \\ 0 \ \ if \ \ z < 0 \\ undefined \ \ if \ \ z = 0 \end{cases}$$



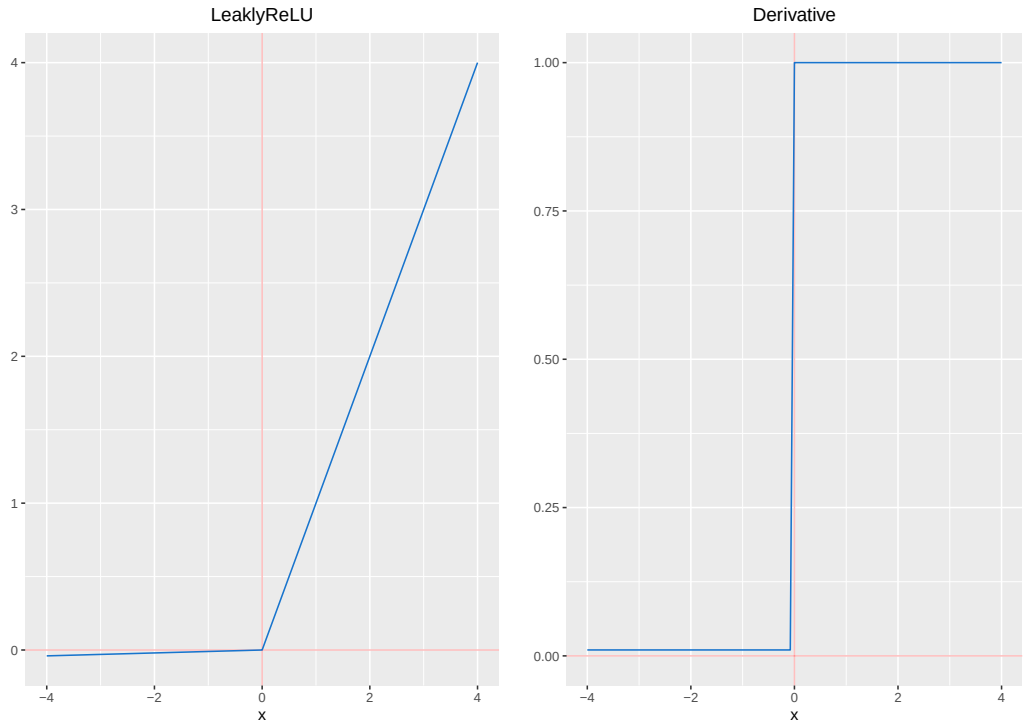
Drawback: **dead** ReLU means can **die** with an output of zero for a negative value input → cause problems in backpropagation → the gradients will be zero for one negative value input

LeakyRelu function

Leaky Relu is a variant of ReLU. Instead of being 0 when $z < 0$, a leaky ReLU allows a small, non-zero, constant gradient α (usually, $\alpha = 0.01$).

$$LeaklyReLU(z) = \max(\alpha z, z)$$

$$\frac{d}{dz} LeaklyReLU(z) = \begin{cases} \alpha & if \quad z < 0 \\ 1 & if \quad z \geqslant 0 \end{cases}$$

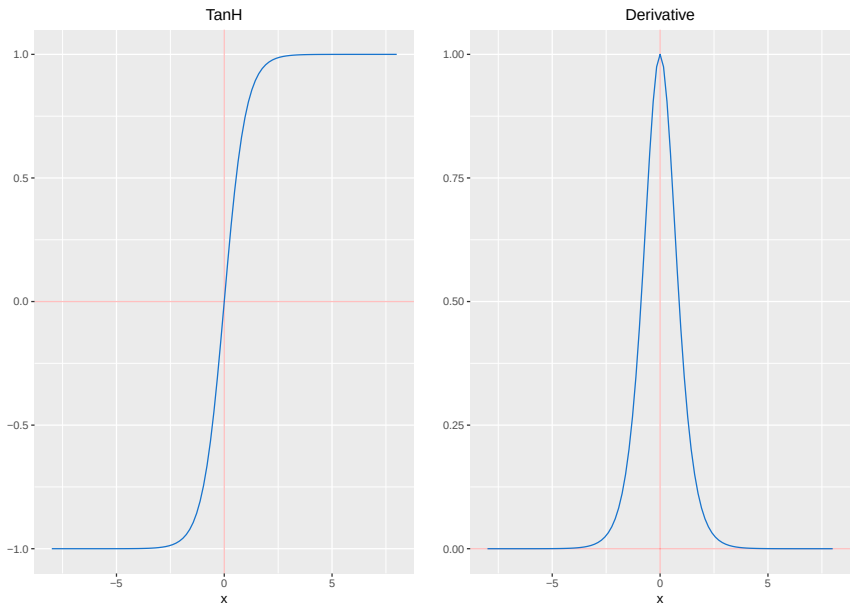


Tanh function

Tanh squashes a real-valued number to the range [−1, 1] (with "S"-shaped). But unlike **Sigmoid**, its output is zero-centered. The gradient of tanh is stronger than sigmoid.

In practice the **tanh** non-linearity is always preferred to the sigmoid nonlinearity.

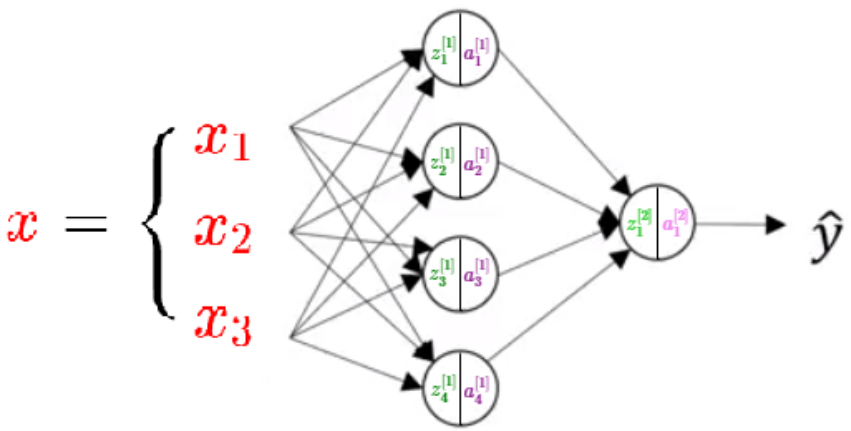
$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \qquad \frac{d}{dz} tanh(z) = 1 - tanh(z)^2$$



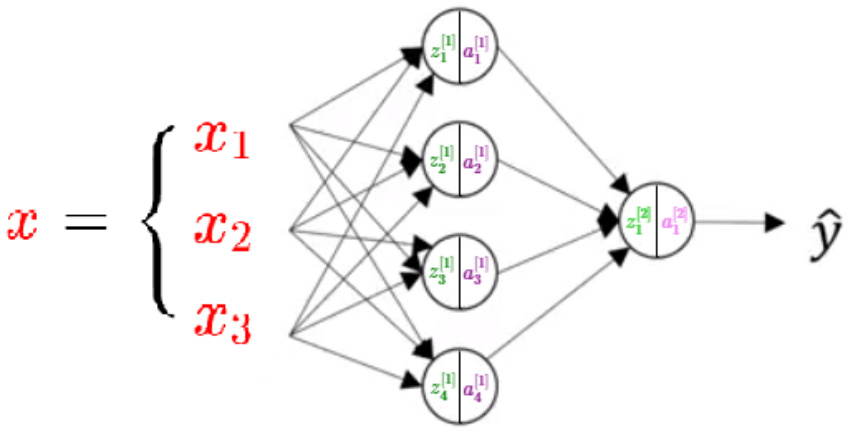
Simple Example in action

Task: Derive the backpropagation algorithm this neural network:

- 3 inputs, 2 Layers
- ReLu activations function for the first Layer
- identity function for the output layer



Forward equations



$$\begin{cases} z^{[1]} = W^{[1]}x + b^{[1]} \\ a^{[1]} = ReLu(Z^{[1]}) \\ z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \\ \widehat{y} = a^{[2]} = g(z^{[2]}) \end{cases}$$

- Question 1: which cost function to used? J =??
- Question 2: Dimension of our objects. replace the ??
  - d is the number of features and $x \in$??
  - m_1 number of neurons in layer 1 and so $W^{[1]} \in$??
  - $m_2 = 1$ number of neurons in output layer and so $W^{[2]} \in$??

Task: Computing derivatives using Chain Rule using Backward strategy:

-(1) Compute $\frac{\partial J}{\partial W_i^{[2]}}$ then get vectorize version $\frac{\partial J}{\partial W^{[2]}}$

-(2) Compute $\frac{\partial J}{\partial W_{ij}^{[1]}}$ then get vectorize version $\frac{\partial J}{\partial W^{[1]}}$

-(3) Compute $\frac{\partial J}{\partial Z_i^{[1]}}$ then get vectorize version $\frac{\partial J}{\partial Z^{[1]}}$

-(4) Compute $\frac{\partial J}{\partial a_i^{[1]}}$ then get vectorize version $\frac{\partial J}{\partial a^{[1]}}$

- Step 1

$$\begin{aligned}\frac{\partial J}{\partial W_i^{[2]}} &= \frac{\partial J}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial W_i^{[2]}} \\ &= (\widehat{y} - y) \frac{\partial \widehat{y}}{\partial W_i^{[2]}} \\ &= (\widehat{y} - y) a_i^{[1]}\end{aligned}$$

where $\widehat{y} = \sum_{i=1}^{m_1} W_i^{[2]} a_i^{[1]} + b^{[2]}$

$$\boxed{\frac{\partial J}{\partial W^{[2]}} = (\widehat{y} - y) a^{[1]T} \in \Re^{1 \times m_1}}$$

Prove the following one $\boxed{\frac{\partial J}{\partial b^{[2]}} = (\widehat{y} - y) \in \Re}$

- Step 2

$$\frac{\partial J}{\partial W_{ij}^{[1]}} = \frac{\partial J}{\partial z_i^{[1]}} \frac{\partial z_i^{[1]}}{\partial W_{ij}^{[1]}}$$
$$= \frac{\partial J}{\partial z_i^{[1]}} x_j$$

where $z_i^{[1]} = \sum_{k=d}^{m_1} W_{ik}^{[1]} x_k + b_i^{[1]}$

$$\boxed{\frac{\partial J}{\partial W^{[1]}} = \frac{\partial J}{\partial z^{[1]}} x^T \in \Re^{m_1 \times d}}$$

Indicate the dimension of each object (above)

- Step 3

$$\begin{aligned}\frac{\partial J}{\partial z_i^{[1]}} &= \frac{\partial J}{\partial a_i^{[1]}} \frac{\partial a_i^{[1]}}{\partial z_i^{[1]}} \\ &= \frac{\partial J}{\partial a_i^{[1]}} 1_{\{z_i^{[1]} \geqslant 0\}}\end{aligned}$$

$$\boxed{\frac{\partial J}{\partial z^{[1]}} = \frac{\partial J}{\partial a^{[1]}} \odot \sigma^{'}(z)}$$

where $\sigma^{'}(\cdot)$ is the element-wise derivative of the activation function σ (here *ReLU* function}) and $\odot$ denotes the element-wise product of two vectors of the same dimensionality.

Indicate the dimension of each object (above)

- Step 4

$$\begin{aligned}\frac{\partial J}{\partial a_i^{[1]}} &= \frac{\partial J}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial a_i^{[1]}} \\ &= (\widehat{y} - y) w_i^{[2]}\end{aligned}$$

where $\widehat{y} = \sum_{i=1}^{m_1} W_i^{[2]} a_i^{[1]} + b^{[2]}$

$$\boxed{\frac{\partial J}{\partial a^{[1]}} = (\widehat{y} - y) W^{[2]T}}$$

Algorithm : Back-propagation for two-layer neural netwoks

1. Compute the values of $z^{[1]}$, $a^{[1]}$ and $\widehat{y}$ using forward pass
2. Compute

$$\begin{aligned}
\delta^{[2]} &= \frac{\partial J}{\partial \widehat{y}} = (\widehat{y} - y) \\
\delta^{[1]} &= \frac{\partial J}{\partial Z^{[1]}} = (W^{[2]T}(\widehat{y} - y)) \odot 1_{\{z^{[1]} \geqslant 0\}}
\end{aligned}$$

3. Compute

$$\begin{aligned}
\frac{\partial J}{\partial W^{[2]}} &= \delta^{[2]} a^{[1]T} \\
\frac{\partial J}{\partial b^{[2]}} &= \delta^{[2]} \\
\frac{\partial J}{\partial W^{[1]}} &= \delta^{[1]} x^T \\
\frac{\partial J}{\partial b^{[1]}} &= \delta^{[1]}
\end{aligned}$$

Take Home Message

- Perceptron
- Multi-layer Perceptron
- Activation function
- Back propagation
- "Dead Neuron"

General Case with r layers

Consider the general case of a fully-connected Multi-layer networks defining by the following equations:

$$
\begin{cases}
a^{[0]} = x \\
z^{[1]} = W^{[1]} a^{[0]} + b^{[1]} \\
a^{[1]} = ReLu(Z^{[1]}) \\
z^{[2]} = W^{[2]} a^{[1]} + b^{[2]} \\
a^{[2]} = ReLu(Z^{[2]}) \\
\dots = \qquad\qquad\qquad\qquad\qquad\qquad \dots \\
z^{[r-1]} = W^{[r-1]} a^{[r-2]} + b^{[r-1]} \\
a^{[r-1]} = ReLu(Z^{[r-1]}) \\
z^{[r]} = W^{[r]} a^{[r-1]} + b^{[r]} \\
\widehat{y} = a^{[r]} = z^{[r]} \\
J = \qquad\qquad\qquad\qquad\qquad\qquad \frac{1}{2}(y - \widehat{y})^2
\end{cases}
$$

Back-propagation multi-layer

- Weights and bias depend of **intermediate** following intermediate variables:

$$z^{[k]} = W^{[k]} a^{[k-1]} + b^{[k]}, \qquad k \in \{1, \dots, r\}$$

- Cost function depends of weights and bias via the intermediate variables $z^{[k]}$.
- Using chain rule we get

$$\begin{cases} \dfrac{\partial J}{\partial W^{[k]}} = & \dfrac{\partial J}{\partial z^{[k]}} a^{[k-1]T} \\ \dfrac{\partial J}{\partial b^{[k]}} = & \dfrac{\partial J}{\partial z^{[k]}} \end{cases}$$

Using similar notation as last lecture, we define $\delta^{[k]} = \frac{\partial J}{\partial z^{[k]}}$ and compute it in a backward manner from $k = r$ to 1.

- **k=r**:

$$\boxed{\delta^{[r]} = \frac{\partial J}{\partial z^{[r]}}} = (z^{[r]} - y)$$

- **k<r**:

$$\boxed{\delta^{[k]} = \frac{\partial J}{\partial z^{[k]}} = \frac{\partial J}{\partial a^{[k]}} \odot \mathrm{ReLU}'(z^{[k]})}$$

By noting that $z^{[k+1]} = W^{[k+1]} a^{[k]} + b^{[k+1]}$ and assuming we have computed $\delta^{[k+1]}$ then we try to compute $\delta^{[k]}$. First note that

$$\frac{\partial J}{\partial a^{[k]}} = W^{[k+1]T} \frac{\partial J}{\partial z^{[k+1]}}$$

then we get

$$\boxed{\begin{aligned} \delta^{[k]} &= \left(W^{[k+1]T} \frac{\partial J}{\partial z^{[k+1]}} \right) \odot \mathrm{ReLU}'(z^{[k]}) \\ &= \left(W^{[k+1]T} \delta^{[k+1]} \right) \odot \mathrm{ReLU}'(z^{[k]}) \end{aligned}}$$

Algorithm : Back-propagation for multi-layer

1. Compute the values of $z^{[k]}$, $a^{[k]}$ for $k = 1, \ldots, r$ and J using forward pass
2. for $k = r$ to 1 do
   - if $k = r$ then compute $\delta^{[r]} = \frac{\partial J}{\partial z^{[r]}}$
   - if $k \neq r$ then compute $\delta^{[k]} = \frac{\partial J}{\partial z^{[k]}} = (W^{[k+1]T}\delta^{[k+1]}) \odot \mathrm{ReLU}'(z^{[k]})$
   - Compute

$$\begin{aligned} \frac{\partial J}{\partial W^{[k]}} &= \delta^{[k]} a^{[k-1]T} \\ \frac{\partial J}{\partial b^{[k]}} &= \delta^{[k]} \end{aligned}$$

How to compute derivatives ?

- **Manuel** using rules of differentiation.
  - Analytical derivatives: **unncessary** when we just need numerical derivatives for optimization
- **Symbolic derivatives**: Symbolic computation with Mathematica, Maple, Theano (for deep learning).
  - Main issue: expression swell

n	l_n	$\frac{d}{dx} l_n$
1	x	1
2	$4x(1-x)$	$4(1-x)-4x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2-16x(1-2x)^2-64x(1-x)(1-2x)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$

From tables from *Automatic Differntiation in Machine Learning: a Survey (2018)*

How to compute derivatives ?

- Numerical differentiation

Let $f : \Re^n \to \Re$ approximate the gradient $\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)$ using

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x + he_i) - f(x)}{h}$$

- need to choose a small h and face to approximation errors

- Can do better with higher-order finite differences:

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x + he_i) - f(x - he_i)}{2h}$$

But increase in complexity and never eliminate the error

Automatic differentiation

Examples and tables from *Automatic Differntiation in Machine Learning: a Survey (2018)*

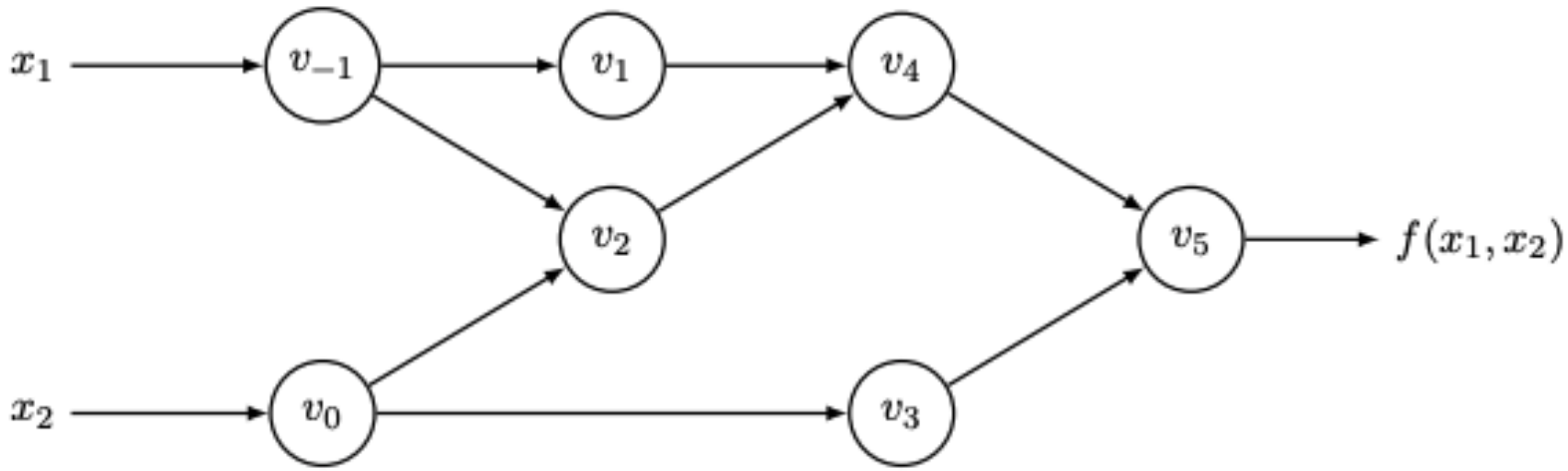
- Automatic differentiation: *Techniques to numerically evaluate the derivative of a function specified by a computer program* by exploiting the chain rule associated to a computational graph.
- Based on the decomposition of the target function to elementary operations of simple function
- AD: *"Refers to a general way of taking a program which computes a value, and automatically constructing a procedure for computing derivatives of that value."* from Roger Grosse.

Automatic differentiation

- AD shared roots with **backpropagation** algorithm for NN but more general
  - **reverse mode accumulation**:
  - **forward mode accumulation**
- Main principles:
  - build an augmented algorithm and keep for each value a primal and a derivative component
  - Algorithms are compositions of a finite set of elementary operations (with known derivatives)

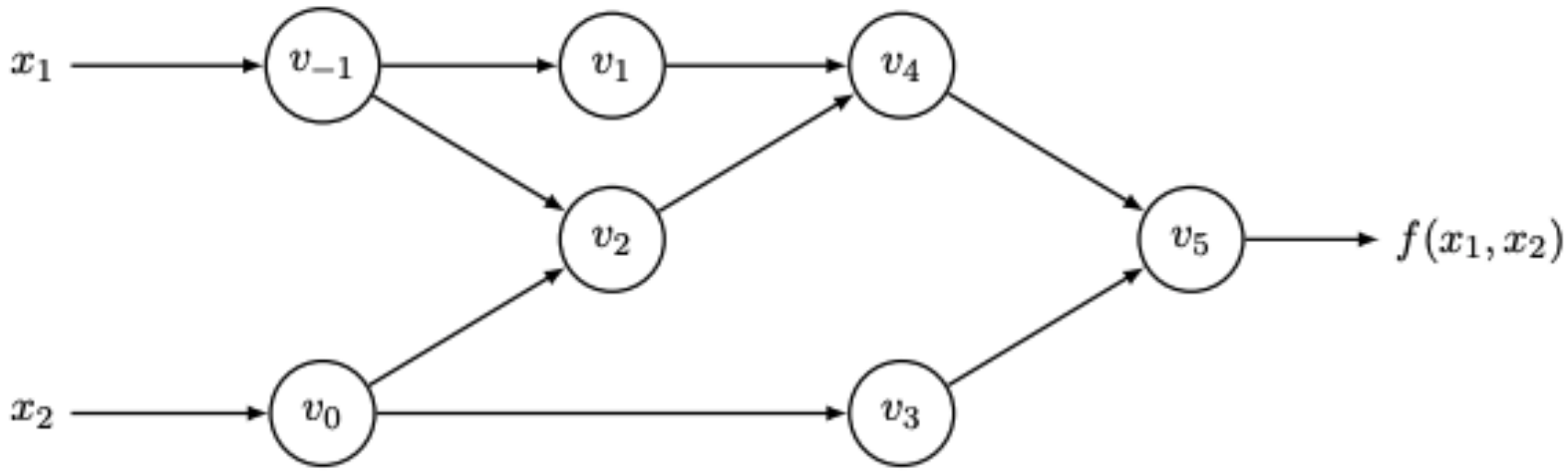
Foward mode

- Consider $y = f(x_1, x_2) = ln(x_1) + x_1 x_2 - sin(x2)$.
- Computational graph (elementary operations)



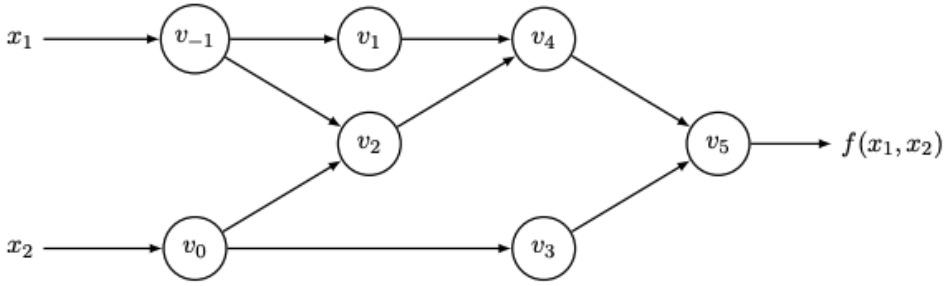
- variables $v_{i-n} = x_i$, $i = 1, \ldots, n$ are the inputs variables
- v_i, $i = 1, \ldots, n$ are the working variables
- $y_{m-i} = v_l$, $i = m - 1, \ldots, 0$ output variables

Foward mode



- Select a variable of differentiation x_i (we choose he x_1)
- augment each working variable value v_j with $\dot{v}_j = \frac{\partial v_j}{\partial x_i}$
- set $\dot{x}_i = 1$ and run a forward pass

Foward mode



Forward Primal Trace			Forward Tangent (Derivative) Trace		
v_{-1}	$= x_1$	$= 2$	$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
v_0	$= x_2$	$= 5$	$\dot{v}_0$	$= \dot{x}_2$	$= 0$
v_1	$= \ln v_{-1}$	$= \ln 2$	$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
v_3	$= \sin v_0$	$= \sin 5$	$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$	$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$	$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
y	$= v_5$	$= 11.652$	$\dot{y}$	$= \dot{v}_5$	$= \mathbf{5.5}$

Could you check we get the good derivatives ?

Foward mode

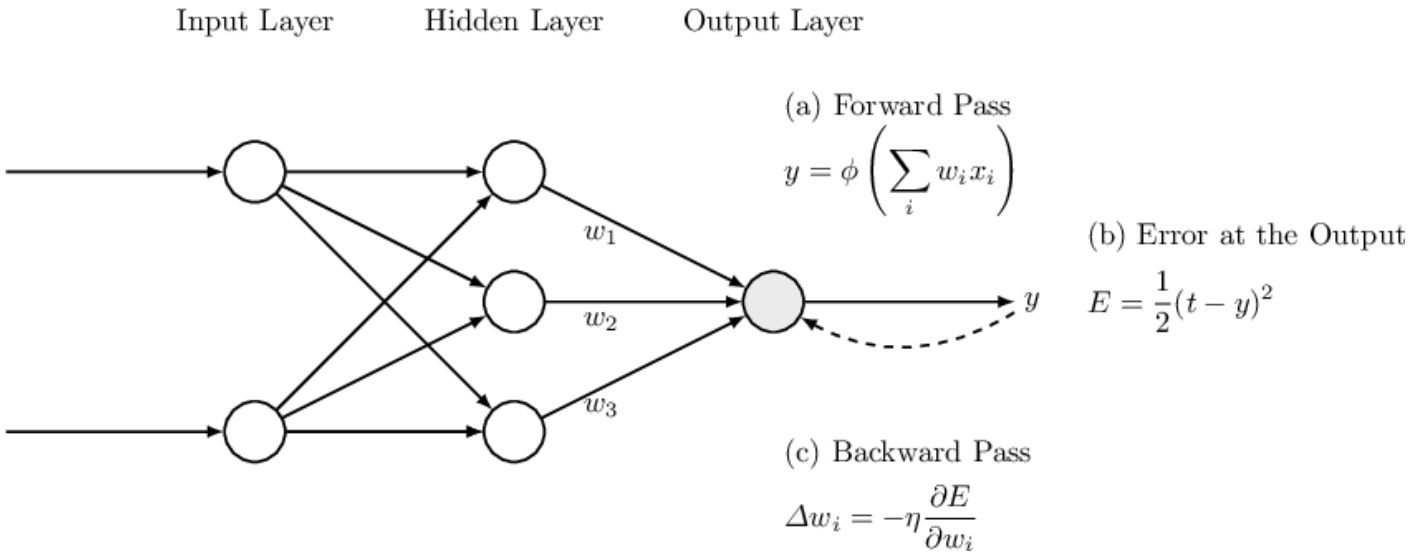
- Forward mode AD efficient and straightforward for function $f : \Re \rightarrow \Re^m$
- Derivatives $\frac{dy_i}{dx}$ computed with just one forward pass
- BUT for function $f : \Re^n \rightarrow \Re$ forward pass requires n evaluations to compute the gradient

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

- In the case of $f : \Re^n \rightarrow \Re^m$ where $n >> m$ better to use the reverse mode

Reverse mode

- Backpropagation is just a special case of reverse mode AD
- Origins in the same papers (Bryson and Ho, 1969, Werbos, 1974)
- Backpropagation brought to fame by Rumelhart et al. (Nature, 1986)



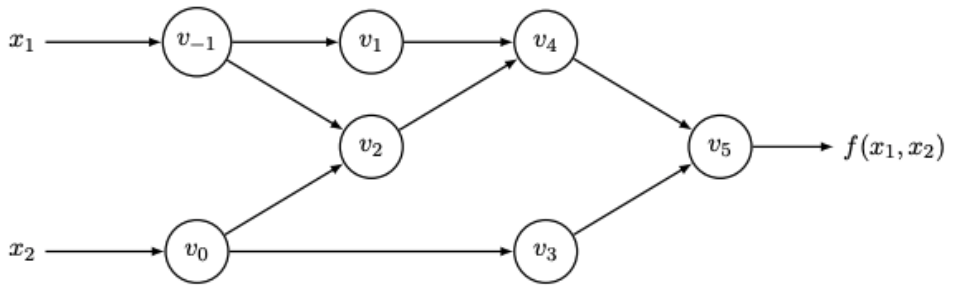
Reverse mode in action

- run a forward pass
- select a dependent variable y_j
- augment each intermediate value v_i with an *adjoint* $\bar{v}_i = \frac{\partial y_j}{\partial v_i}$

$$\bar{v}_i = \frac{\partial y_j}{\partial v_i} = \sum_{j:\text{child of } i} \bar{v}_j \frac{\partial v_j}{\partial v_i}$$

- set $\bar{y}_j = 1$ and run backward

Reverse mode in action



Forward Primal Trace

$v_{-1} = x_1$		$= 2$
$v_0 = x_2$		$= 5$
$v_1 = \ln v_{-1}$		$= \ln 2$
$v_2 = v_{-1} \times v_0$		$= 2 \times 5$
$v_3 = \sin v_0$		$= \sin 5$
$v_4 = v_1 + v_2$		$= 0.693 + 10$
$v_5 = v_4 - v_3$		$= 10.693 + 0.959$
$y = v_5$		$= 11.652$

Reverse Adjoint (Derivative) Trace

$\bar{x}_1 = \bar{v}_{-1}$		$= \mathbf{5.5}$
$\bar{x}_2 = \bar{v}_0$		$= \mathbf{1.716}$
$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5 = \bar{y}$		$= 1$

Reverse mode

- Significantly less costly to evaluate for functions with a large number of inputs
- for $f : \mathfrak{R}^n \to \mathfrak{R}$ only one application of the reverse mode to get the full gradient

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

- **TensorFlow AutoDiff** allows to compute and manipulate gradients
- There are many autodiff libraries (e.g., PyTorch, Tensorflow, Jax, etc.)

Want to play ?

$$y(x_0, x_1) = (1 + e^{x_0\ x_1 + sin(x_0)})^{-1}$$

- Do you recognize a known function ?
- Task: compute $\frac{\partial y}{x_0}$ and $\frac{\partial y}{x_1}$
- Use the two modes of AD
- Solution here

Weight initialization

- Important Step for optimization

$$\begin{cases} b^{[l]} := & b^{[l]} - \alpha \dfrac{\partial J}{\partial b^{[l]}} \\ W^{[l]} := & W^{[l]} - \alpha \dfrac{\partial J}{\partial W^{[l]}} \end{cases}$$

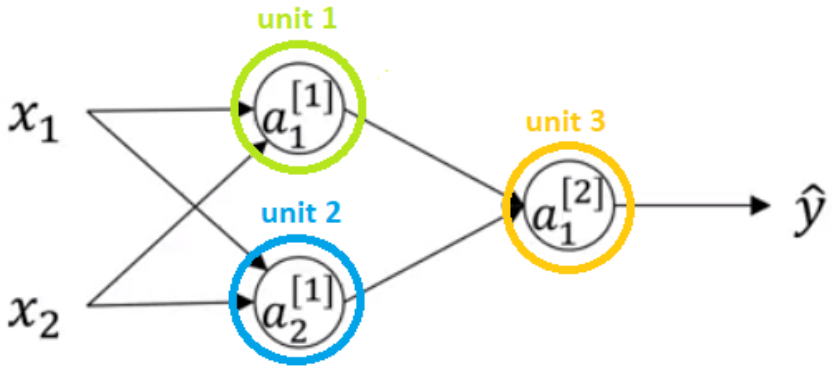
- $W^{[l]}$ weight matrix of dimension $m_l \times m_{l-1}$ (m_l is the size of the layer l)
- $b^{[l]}$ bias vectors odf dimension $m_l \times 1$

General practice

The biases are initialized with 0 and weights are initialized with random numbers.

What if weights are initialized with 0? or even same constant value

- Consider a neural network with two hidden units
- Initialize the biases to 0 and all the weights to a constant value γ.



- The output of both hidden units will be the same: $ReLU(\gamma x_1 + \gamma x_2)$.
- Identical influence on the cost function → identical gradients.
- Makes hidden units symmetrict → DNN will perform very poorly. Let plays Initializing neural networks

Random initialization

- **Random initialization**: break the symmetry.
- Initializing much **high** or **low value** can result in slower optimization.
- **General practice**: randomly gerenated from standard normal distribution.
- However, while working with a (deep) network can potentially lead to 2 issues:
  - **vanishing gradients**
  - **exploding gradients**.

Vanishing gradients

- For any **activation function**, $|\frac{\partial J}{\partial W^{[l]}}|$ will get smaller and smaller as we go backwards with every layer during back propagation.
- The **earlier layers** are the slowest to train in such a case.
- Thus, the update is **minor** and results in slower convergence. This makes the optimization of the loss function slow.
- In the worst case, this may completely **stop** the neural network from training further.
- For $sigmoid(z)$ and $tanh(z)$, if your weights are **large**, then the gradient will be vanishingly small, effectively preventing the weights from changing their value.
- With $ReLU(z)$ vanishing gradients are generally not a problem as the gradient is 0 for negative (and zero) inputs and 1 for positive inputs_

Exploding gradients

- This is the exact opposite of vanishing gradients.
- Consider you have non-negative and large weights and small activations.
- When these weights are multiplied along the layers, they cause a large change in the cost. Thus, the gradients are also going to be large.
- Thus the changes in $W^{[l]}$ will be in huge steps.
- Might result in oscillating around the minima or even overshooting the optimum again and again and the model will never learn!
- Another impact **huge values** of the gradients may cause **number overflow** resulting in incorrect computations or introductions of **NaN's**.

Solution

- For networks **not too deep**: *ReLU* or *leaky RELU* activation functions are relatively robust to the vanishing/exploding gradient issue.
- *leaky RELU* never has 0 gradient $\rightarrow$ never die, training continues.
- For DNN, heuristic to initialize the weights are generally used:
  - The most common practice is to draw the element of the matrix $W^{[l]}$ from normal distribution with variance k/m_{l-1}, where k depends on the activation function.
  - for *ReLU* activation: $k = 2$
  - for *tanh* activation: $k = 1$. The heuristic is called **Xavier initialization**. It is similar to the previous one, except that k is 1 instead of 2.
  - Another commonly used heuristic is to draw from normal distribution with variance $2/(m_{l-1} + m_l)$
- The bias terms can be safely initialized to 0 as the gradients with respect to bias depend only on the linear activation of that layer, and not on the gradients of the deeper layers.

Approximation Properties of Multilayer Perceptrons

Universal Approximation Theorems

Universal approximators

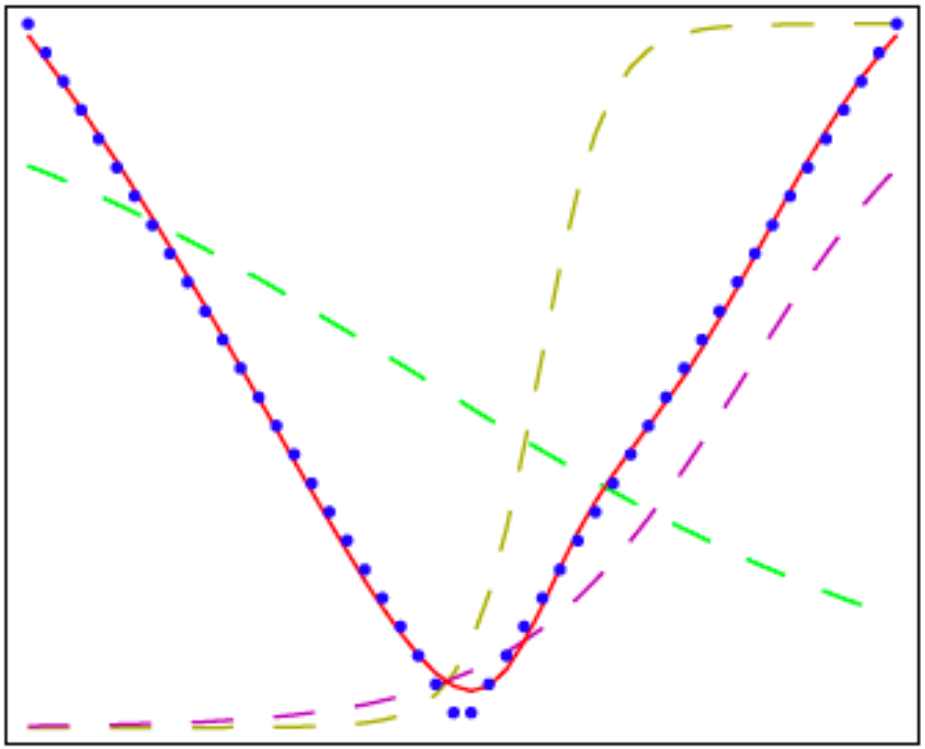
A two-layer network with linear outputs can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy.

- This result holds if the activation function is not a polynomial (i.e. tanh, logistic, and ReLU all works as do sin,cos, exp, etc.)
- See M. Leshno, et al (1991). *Multilayer feedforward networks with non-polynomial activation function can approximate any function*,Neural Networks, vol. 6, pp. 861–867, 1993.

Example: Approximation Ability

$f(x) = |x|$:

- 50 data points in $[-1, 1]$
- two layers (1 hidden Layer), linear activation (output layer)
- 3 hidden units; **tanh** activation functions

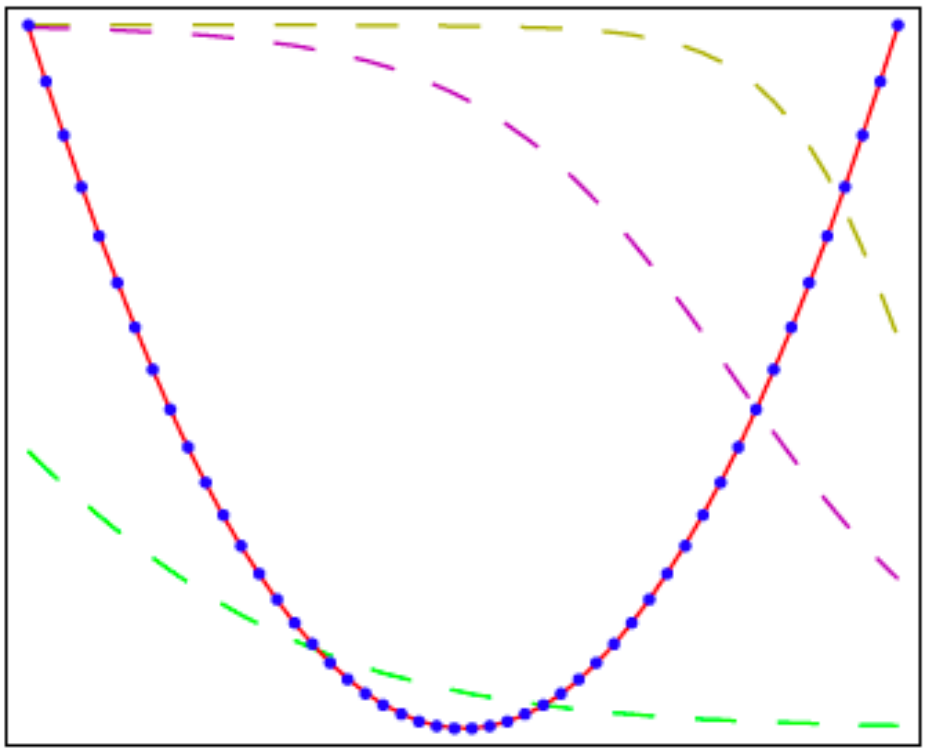


From Bishop's Pattern Recognition and Machine Learning, Fig 5.3

Example: Approximation Ability

$f(x) = x^2$:

- 50 data points in $[-1, 1]$
- two layers (1 hidden Layer), linear activation (output layer)
- 3 hidden units; **tanh** activation functions

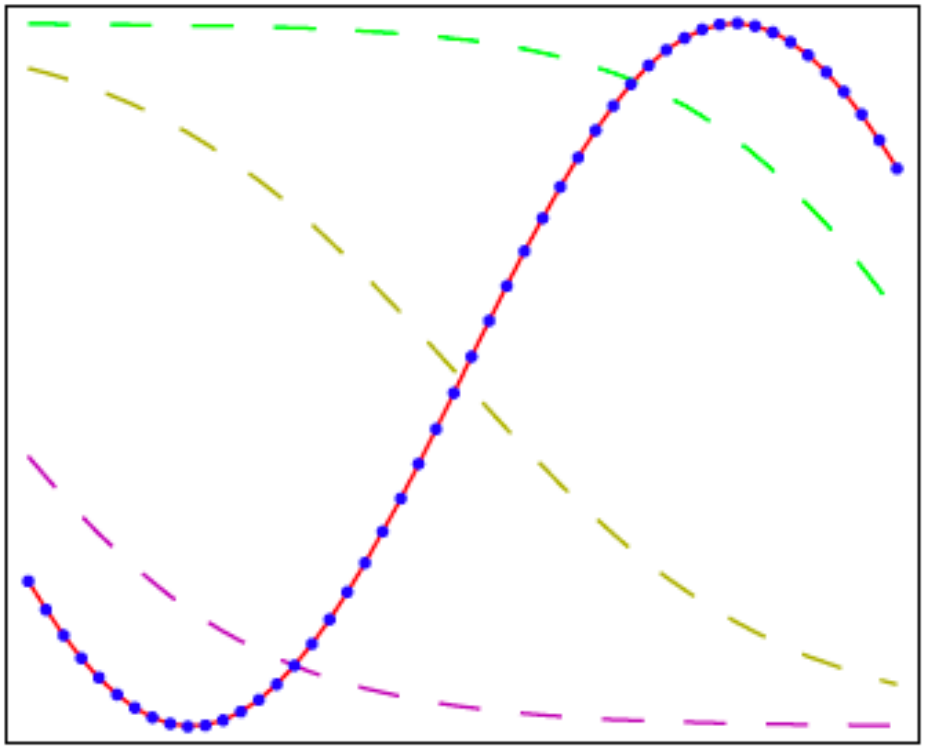


From Bishop's Pattern Recognition and Machine Learning, Fig 5.3

Example: Approximation Ability

$f(x) = sin(x)$:

- 50 data points in $[-1, 1]$
- two layers (1 hidden Layer), linear activation (output layer)
- 3 hidden units; **tanh** activation functions

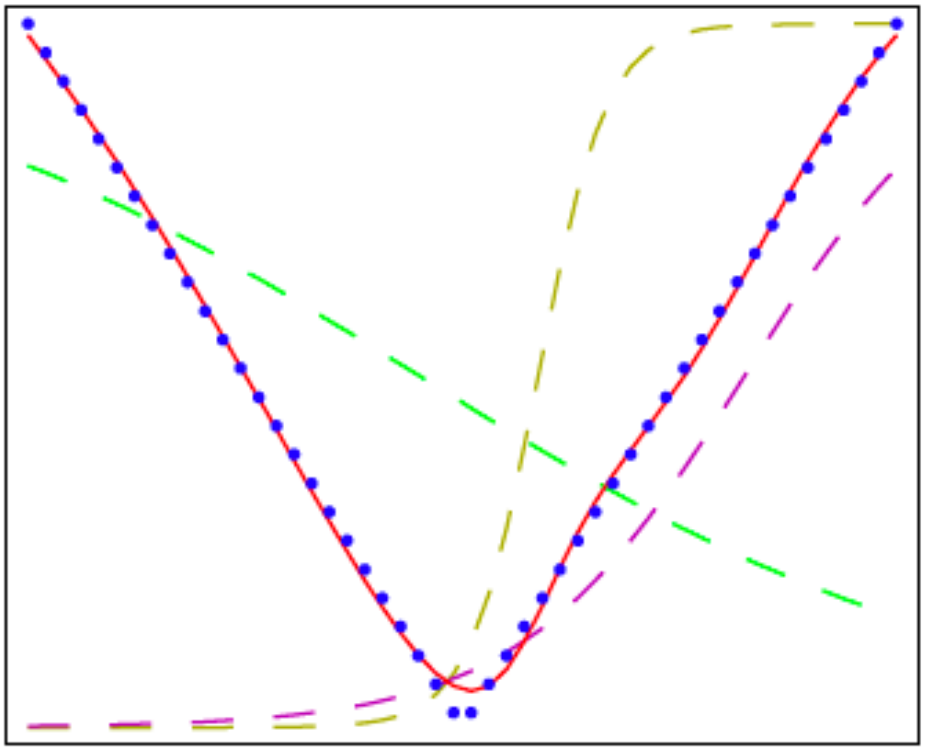


From Bishop's Pattern Recognition and Machine Learning, Fig 5.3

Example: Approximation Ability

$f(x) = |x|$:

- 50 data points in $[-1, 1]$
- two layers (1 hidden Layer), linear activation (output layer)
- 3 hidden units; **tanh** activation functions

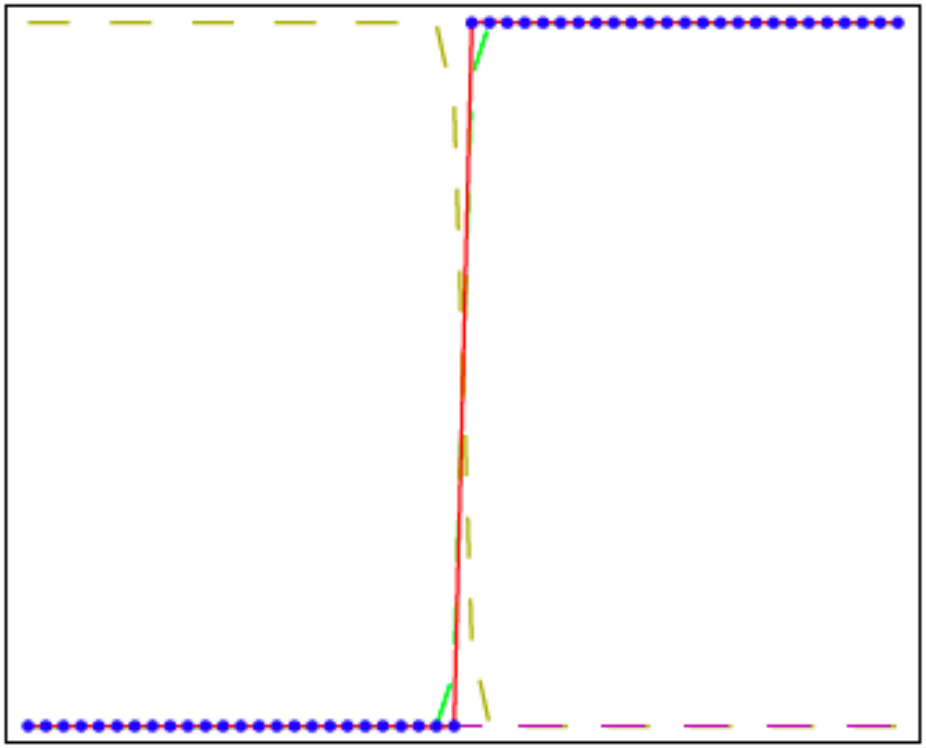


From Bishop's Pattern Recognition and Machine Learning, Fig 5.3

Example: Approximation Ability

$f(x) = 1_{\{x>0\}}$:

- 50 data points in $[-1, 1]$
- two layers (1 hidden Layer), linear activation (output layer)
- 3 hidden units; **tanh** activation functions



From Bishop's Pattern Recognition and Machine Learning, Fig 5.3

Universal Approximation

Leshno and Schocken (1993) showed:

- Let $\psi(\cdot)$ be any non-polynomial function (an activation function).
- Let define $f : K \longrightarrow \Re$ be any continuous function on a compact set $K \subset \Re^m$
- $\forall \varepsilon > 0$, there exists an integer N (the number of hidden units), and parameters v_i, $b_i \in \Re$ such that the function

$$F(x) = \sum_{i=1}^{N} v_i \psi(w_i^T x + b_i)$$

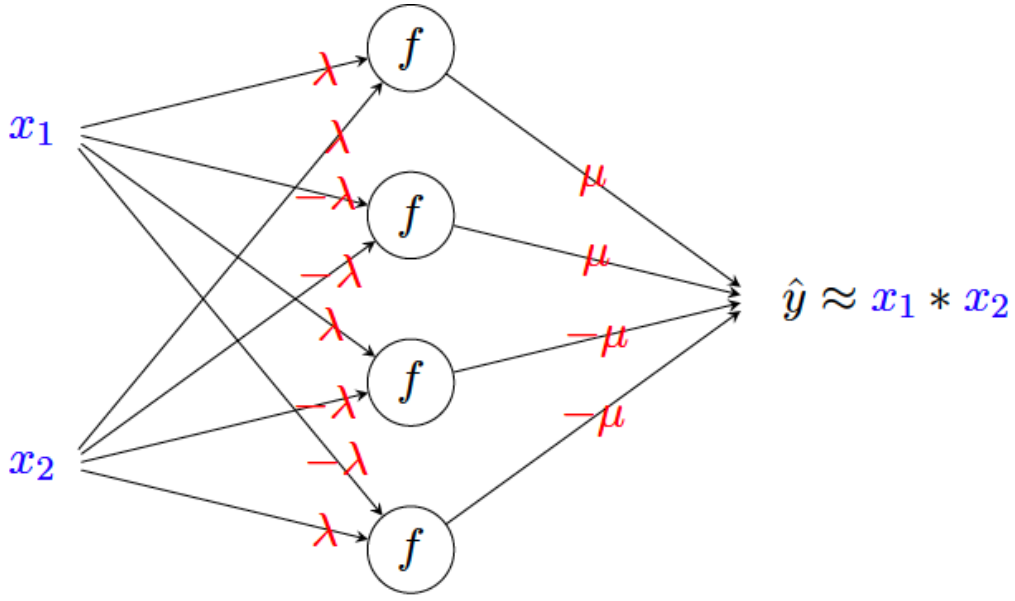
satisfies $|F(x) - f(x)| > \varepsilon$ for all $x \in K$.

Why deep Neural network ?

Continuous multiplication gate Henry W. Lin and Max Tegmark. (2016) *Why does deep and cheap learning work so well?*:

Continuous multiplication gate

A neural network with only four hidden units can model multiplication of two numbers arbitrarily well.



- With $\mu = \frac{1}{4\lambda^2 f''(0)}$ then $\widehat{y} \to x_1 \times x_2$ when $\lambda \to 0$

Regression example

- input: $x \in \mathbb{R}^{1000}$
- Output: $y \in \mathfrak{R}$

Aim build a model for representing a quadratic relation between x and y

$$\widehat{y} = w_{1,1}x_1x_1 + w_{1,2}x_1x_2 + \ldots + w_{1000,1000}x_{1000}x_{1000} = w^T\widetilde{x}$$

where

$$\widetilde{x} = (x_1x_1, x_1x_2, \ldots, x_{1000}x_{1000})^T$$

and

$$w = (w_{1,1}, w_{1,2}, \ldots, w_{1000,1000})^T$$

which requires $\approx \frac{1{,}000 \times 1{,}000}{2} = 500,000$ parameters.

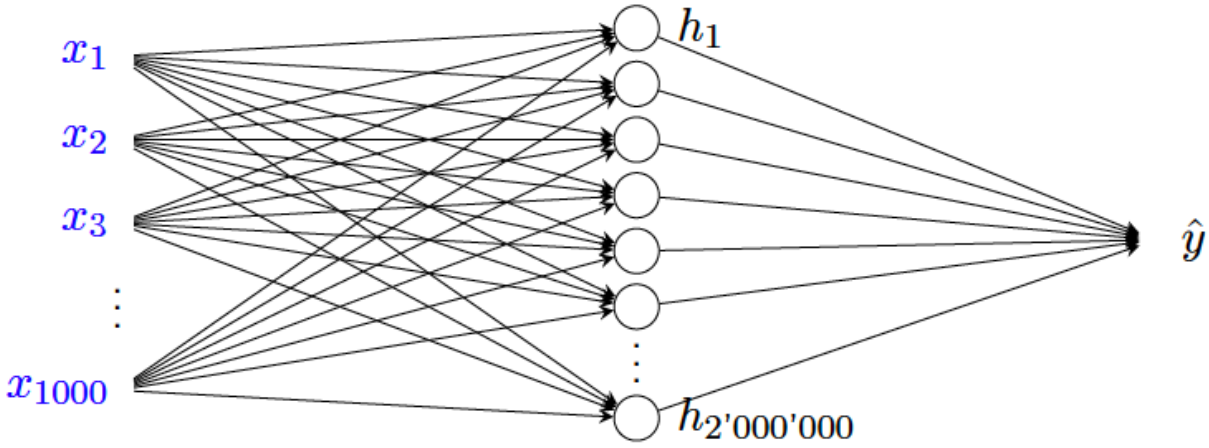
Quadratic model

- input: $x \in \mathbb{R}^{1000}$
- Output: $y \in \mathfrak{R}$

Aim build a model for representing a quadratic relation between x and y

Neural network

All products (interaction) with a neural network requires:
$4 \times 500,000 = 2 \times 10^6$ hidden units and so **2 billion parameters**



$$1000 \times (2 \times 10^6) + 2 \times 10^6 \text{ parameters}$$

Quadratic model with regression

- input: $x \in \mathbb{R}^{1000}$
- Output: $y \in \mathfrak{R}$

Aim build a model for representing a quadratic relation between x and y

- Consider that only 10 of the regressors $x_i x_j$ are of importance

$$\widehat{y} = w_{1,1} x_1 x_1 + w_{1,2} x_1 x_2 + \ldots + w_{1000,1000} x_{1000} x_{1000} = w^T \widetilde{x}$$

where

$$\widetilde{x} = (x_1 x_1, x_1 x_2, \ldots, x_{1000} x_{1000})^T$$

and

$$w = (w_{1,1}, w_{1,2}, \ldots, w_{1000,1000})^T$$

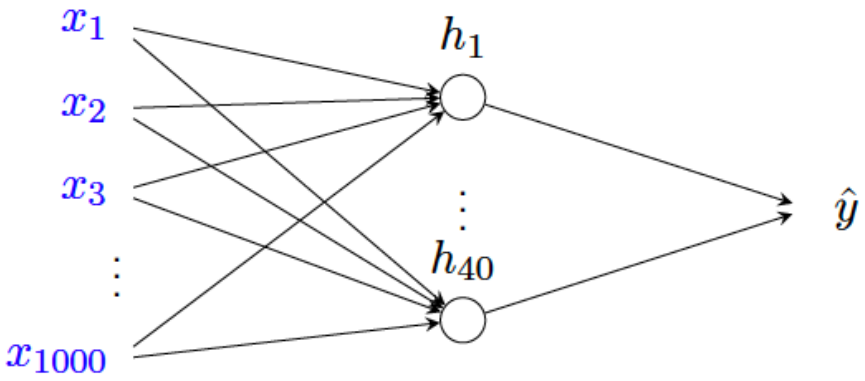
which still requires $\approx \frac{1{,}000 \times 1{,}000}{2} = 500,000$ parameters.

Why Neural network ?

- input: $x \in \mathbb{R}^{1000}$
- Output: $y \in \mathfrak{R}$

Aim build a model for representing a quadratic relation between x and y

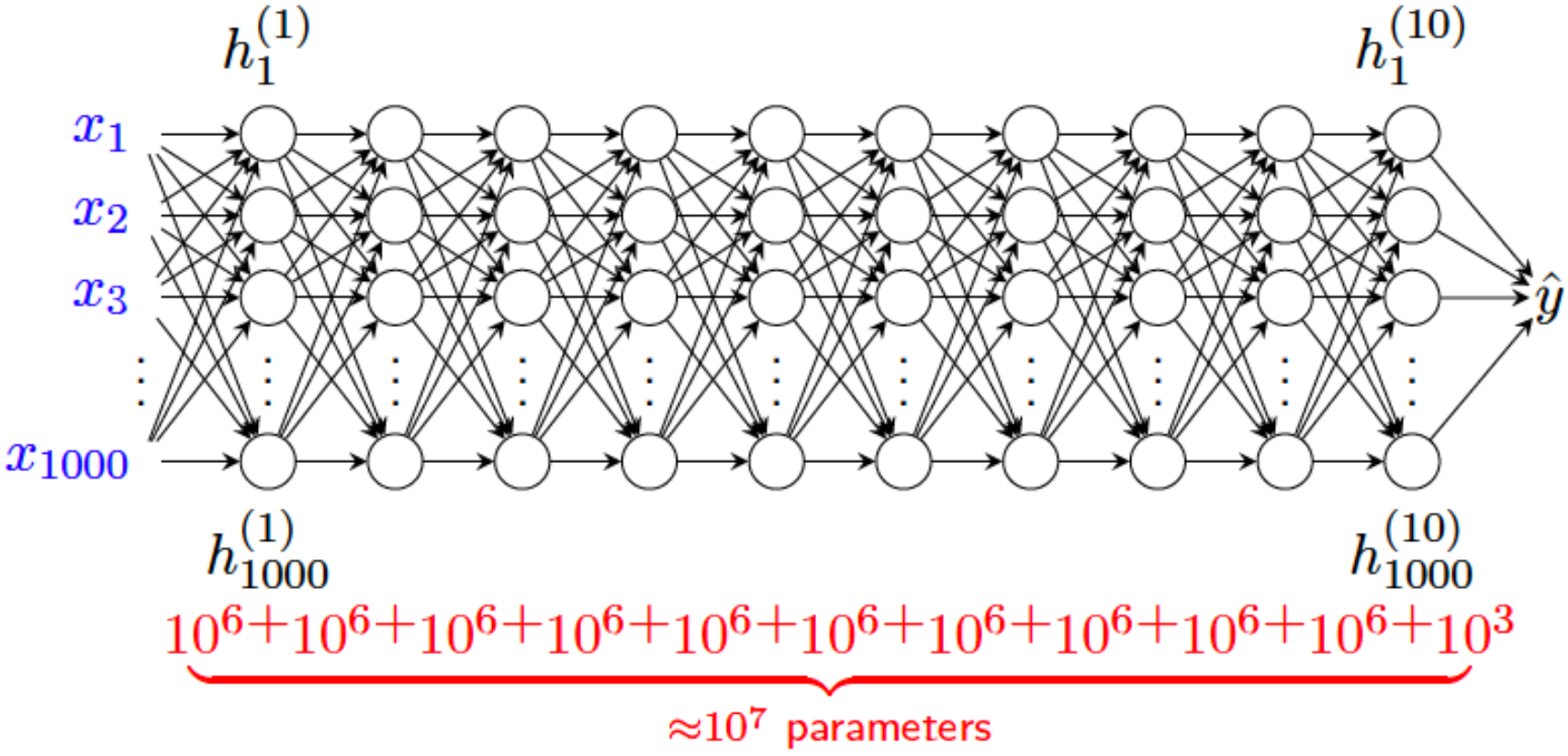
- Consider that only 10 of the regressors $x_i x_j$ are of importance
- Neural Network: for 10 products with NN $\longrightarrow 4 \times 10$ hidden units $\longrightarrow 40,000$ parameters



$$1000 \times 40 + 40 = 40,040 \text{ parameters}$$

Why deep Neural network ?

- Higher complexity model: polynomials of degree 1000
- Keep **250** products in each layer $\rightarrow 250 \times 4 = 1,000$ hidden units.



Linear regression would require $\approx \frac{1000^{1000}}{1000!}$

Take Home Message

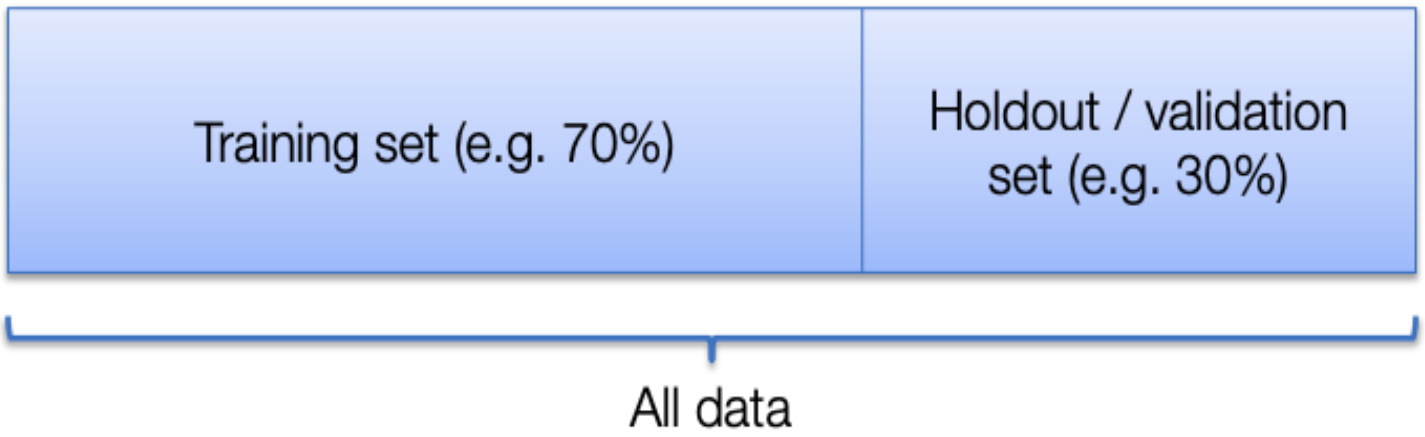
- Automatic Differentation
- Weight Initialization
- Forward mode AD
- Universal approximation theorem

Overfitting

How our model will generalize to new samples that we didn't use to train

Solution to quantify the true **generalization error** is to split the data:

- First version: **holdout cross-validation**



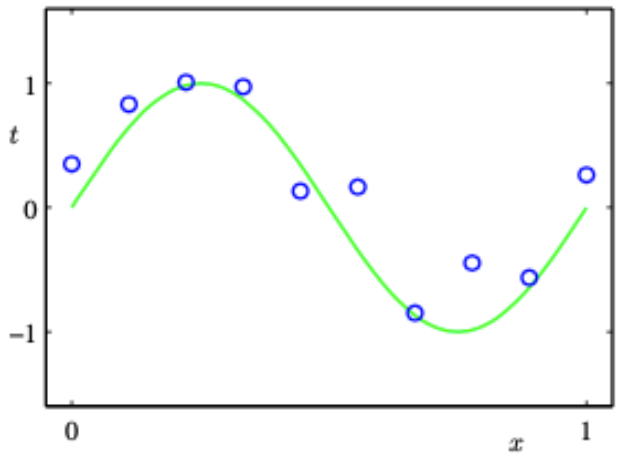
- Second version: **K-fold cross-validation**

Overfitting for Neural network

- Might need a very large network to represent a function
- Neural Network: can learn any function !!
- OVERFITTING is a serious concern
- Solution: Training/validation/test, k-fold cross-validation, dropout, regularisation

Example: curve fitting

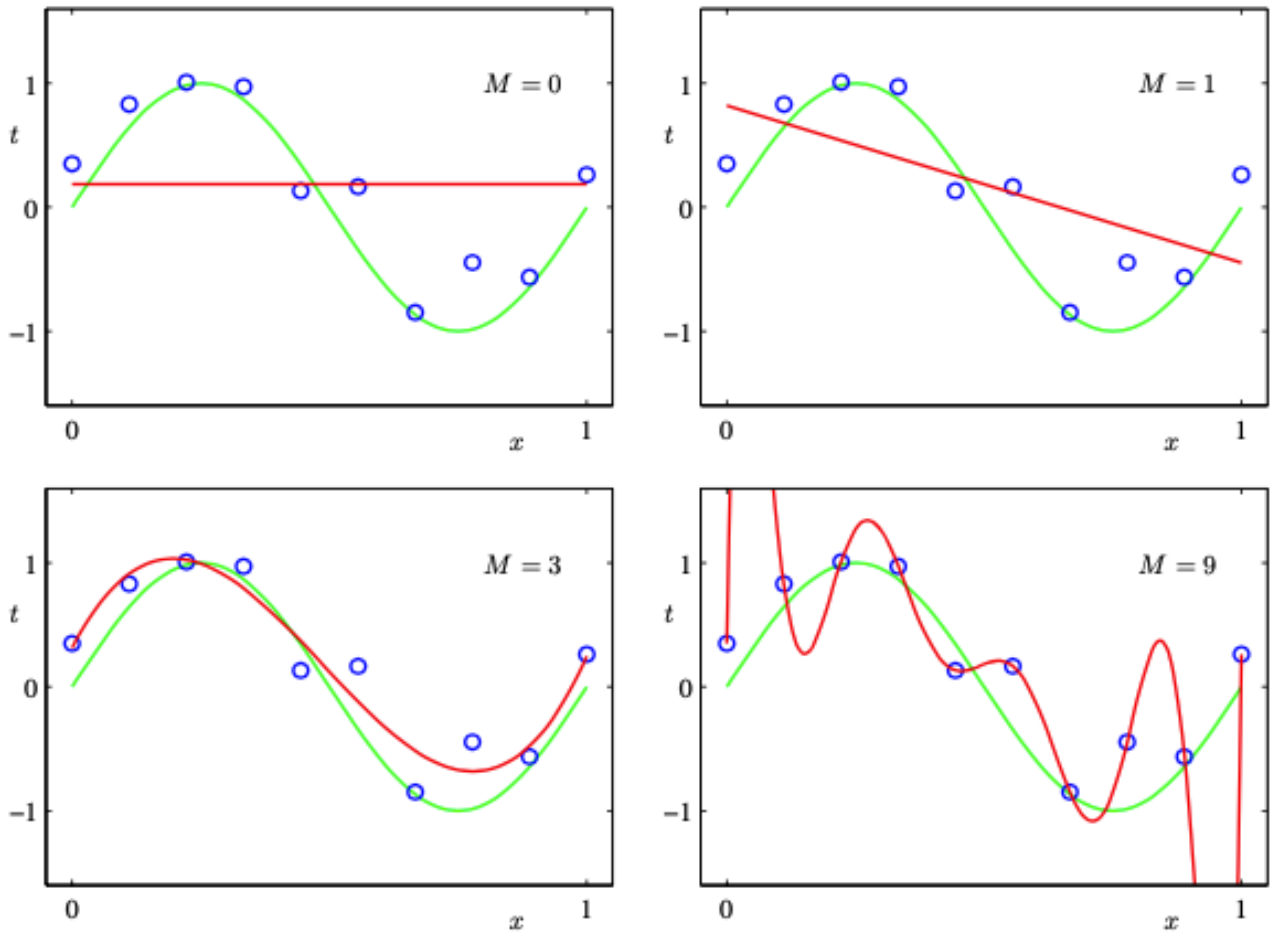
- $N = 10$ points from true function $f(x) = sin(2\pi x)$
- added noise
- Task: fit a polynomial model of degree $M \in \{0, \ldots, 9\}$
- Evaluate RMSE on test data



From Bishop's Pattern Recognition and Machine Learning

Example: curve fitting

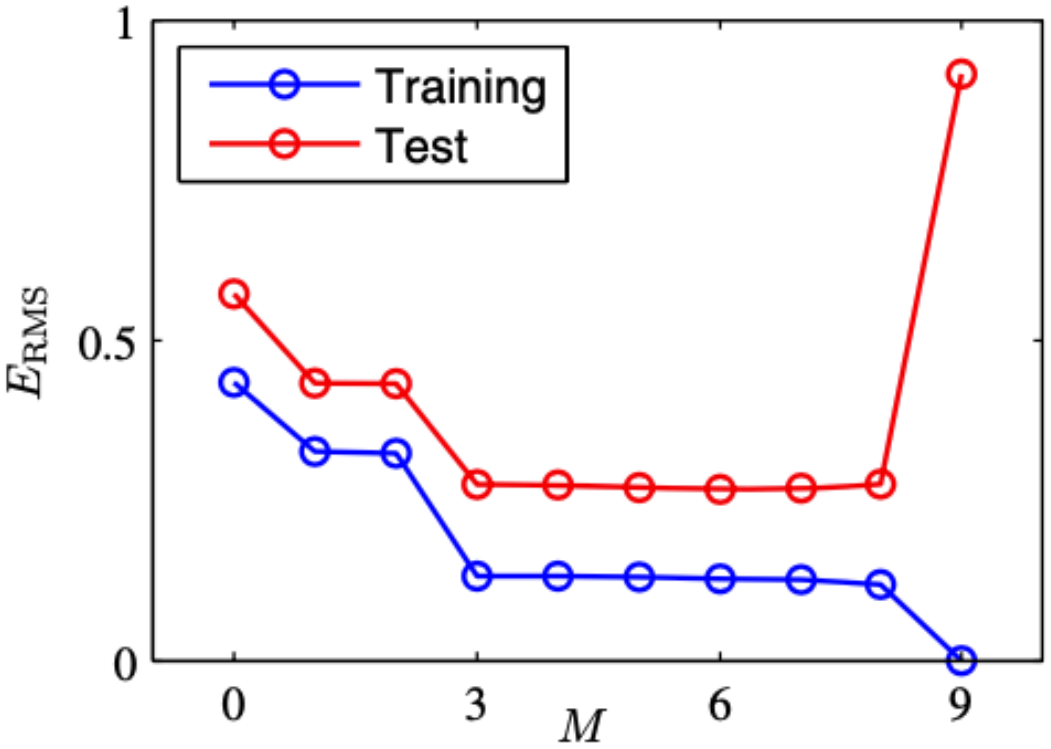
- $M = 3$ looks good
- $M = 9$ overfit ?



From Bishop's Pattern Recognition and Machine Learning

Example: curve fitting

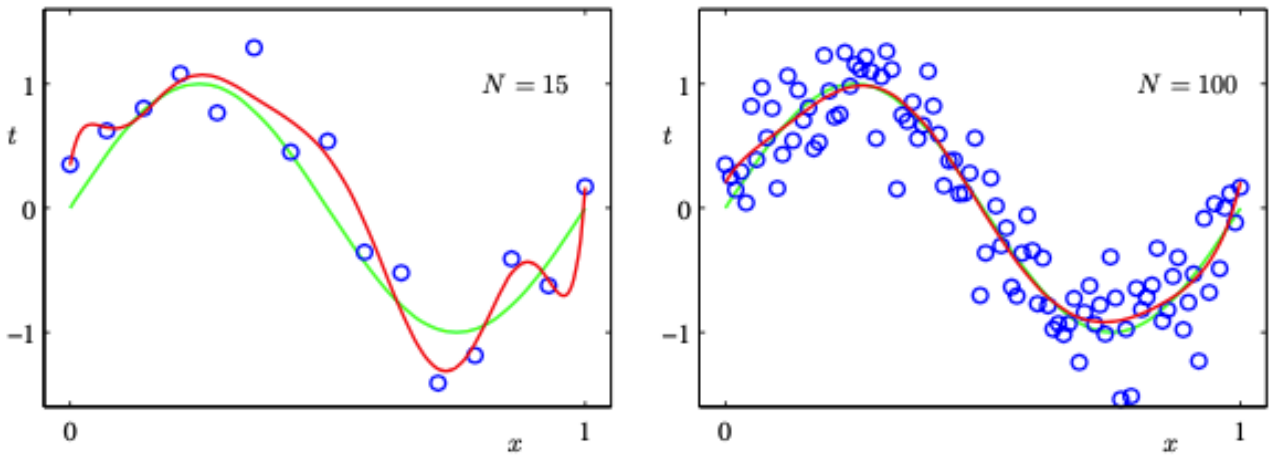
- RMSE on train
- RMSE on test (100 points generated from same process)



From Bishop's Pattern Recognition and Machine Learning

Example: curve fitting

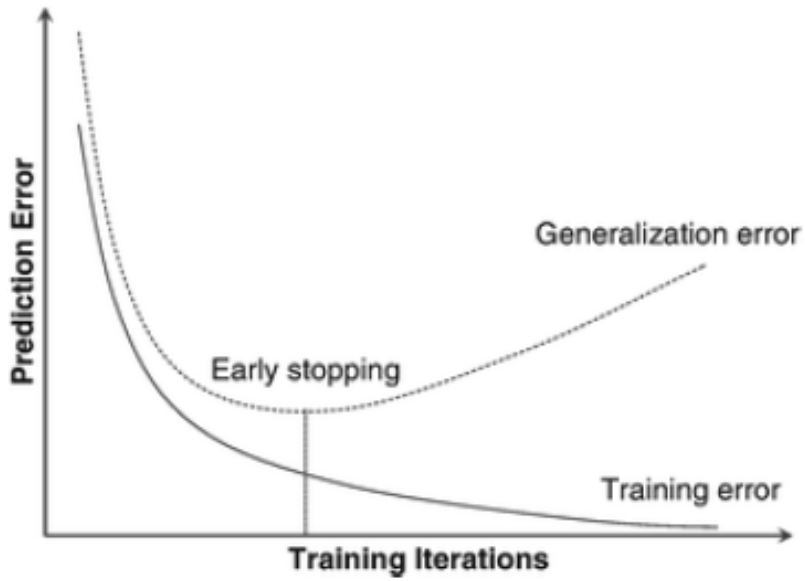
- over-fitting problem become less severe as the size of the data set increases.



From Bishop's Pattern Recognition and Machine Learning

Early stopping

- Early stopping: stopping training early since overfitting typically increases as training progresses.



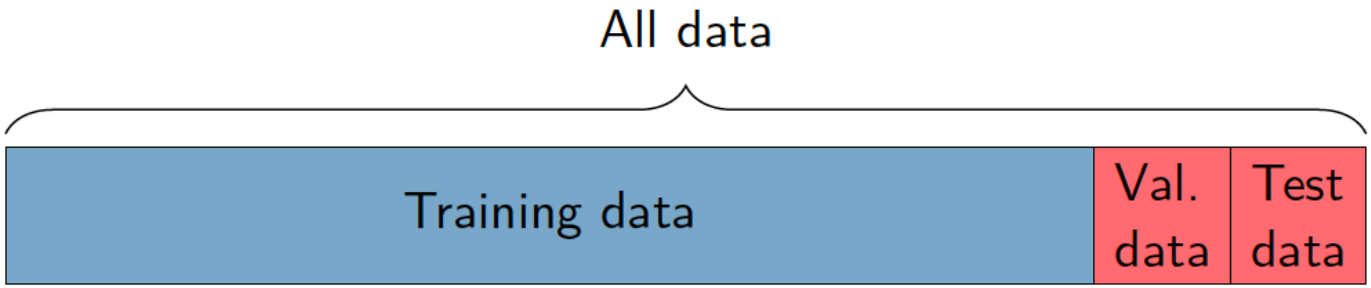
- **keras** offers patience parameter. Interrupts training when accuracy has stopped improving for more than k epochs.

Example in Action

- Demo on housing dataset
- More during tutorial

Split Data for Neural Network

How the data are split for Neural Network ?



- **Training Data**: used for Training models
- **Validation Data**: used for **optimizing hyperparameters**, choosing between models
- **Test Data**: for evaluating the performance of the final model

Overfitting: Regularization

- **Regularization**: process to improve generalization, restrict the flexibility of the model to prevent overfitting
- **Example**: Ridge Regression
- **Principle**: add **Prior** $R(\theta)$ in training objective: $J(\theta) + \lambda R(\theta)$
- Gradient descent update to minimize $J(\theta)$: $\theta \leftarrow \theta - \alpha \frac{\partial J}{\partial \theta}$
- The gradient descent update to minimize the L^2 regularized cost $J(\theta) + \lambda R(\theta)$ results in weight decay:

$$\begin{aligned} \theta &\leftarrow \theta - \alpha \frac{\partial (J + \lambda R)}{\partial \theta} \\ &= \theta - \alpha \left(\frac{\partial J}{\partial \theta} + \lambda \frac{\partial R}{\partial \theta} \right) \\ &= \theta - \alpha \left(\frac{\partial J}{\partial \theta} + \lambda \theta \right) \\ &= (1 - \alpha\lambda)\theta - \alpha \frac{\partial J}{\partial \theta} \end{aligned}$$

Overfitting: Regularization

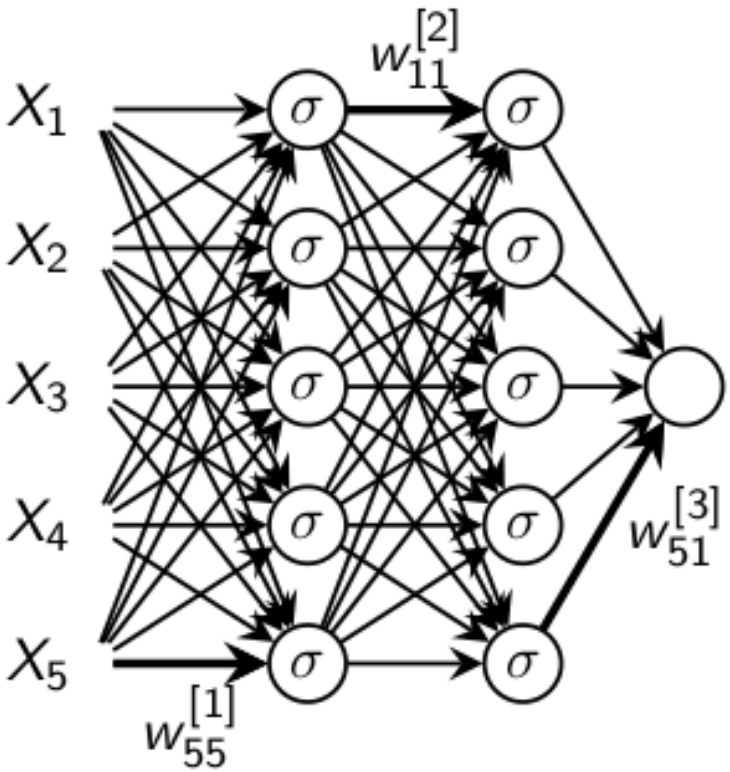
- **Lasso** penalty: L_1 norm, $\lambda_1 \sum_i^d |w_i|$
- **Ridge** penalty: L_2 norm, $\lambda_2 \sum_i^d w_i^2$
- **elastic net** penalty: combine L_1 and L_2 norms
- Different *weight regularization* could be added to different layers

Regularization: Dropout

Dropout is a popular and efficient regularization technique.

- *Srivastava, Nitish et al. (2014).* “Dropout: A simple way to prevent neural networks from overfitting”. In: The Journal of Machine Learning Research 15.1, pp. 1929-1958.

Consider the following network to be trained



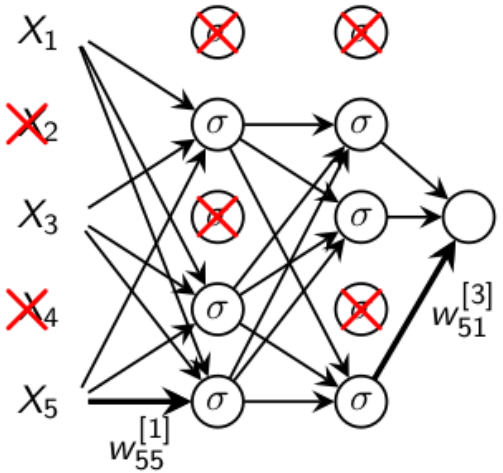
Regularization: Dropout

- Dropout is a regularization technique where we **during training** randomly drop units.

- The term **dropout** refers to dropping out units (hidden and visible) in a neural network.

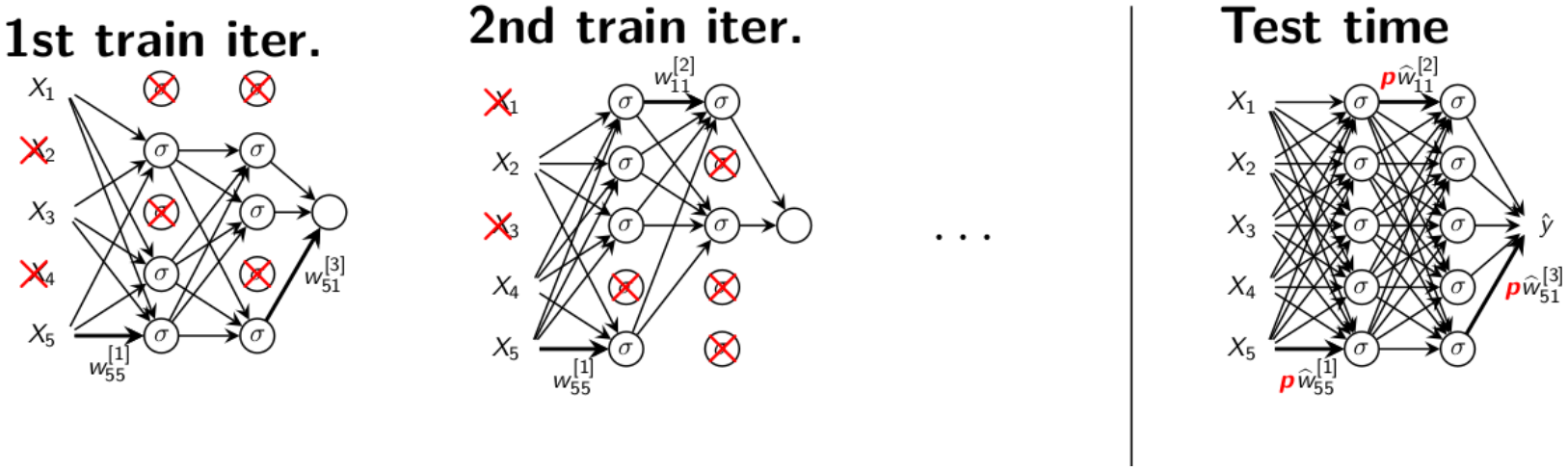
- By dropping a unit out, meaning temporarily removed it from the network, along with all its incoming and outgoing connections.

- The choice of which units to drop is random.



Dropout: principle

- 1st iteration: Keep and update each unit with probability p, drop remanding ones
- 2st iteration: Keep and update another random selection of units, drop remanding units.
- t**th iter** Continue in the same manner.
- **Test time** Use all units. Weight multiplied by p.

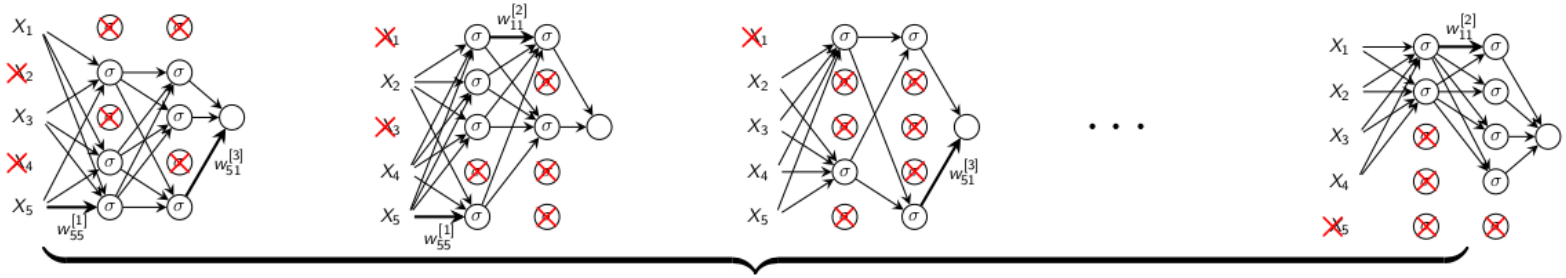


Why does this avoid overfitting?

Dropout can be viewed as an ensemble member with two clever approximations.

Ensemble methods principle: exploit multiple learning models to obtain better predictive performance than could be obtained from any of the contributing models.

1) For a neural network with M units there are 2^M possible thinned neural networks. Consider this as our **ensamble**.



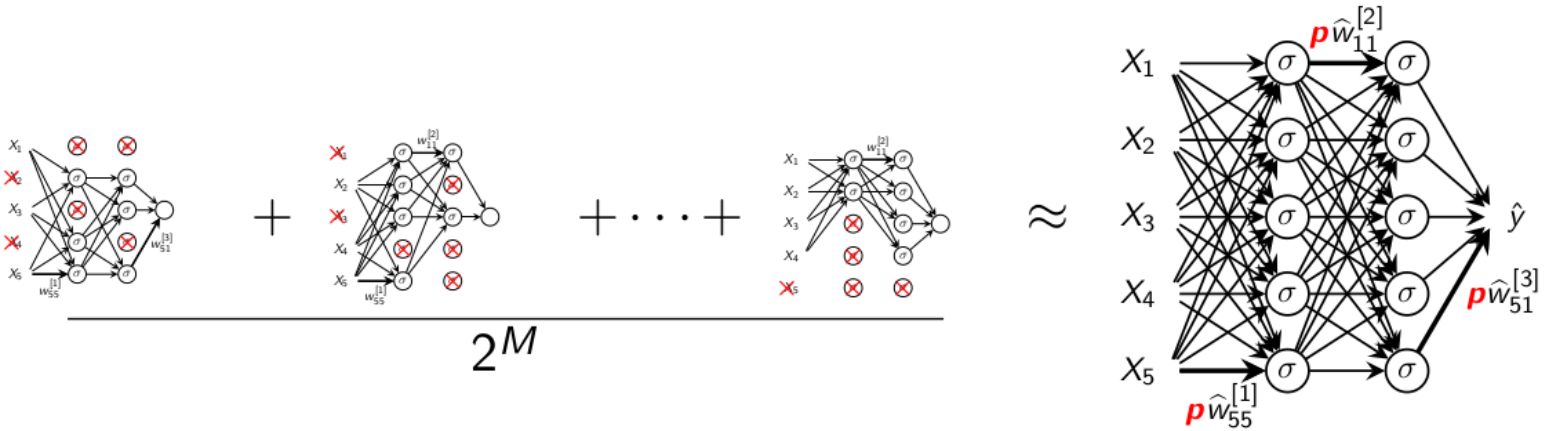
Our ensamble: 2^n"thinned" networks. They all share weights.

- **Approximation 1:** At each iteration we sample one ensemble member and update it. Most of the networks will never be updated since $2^M >>$ the number of iterations.

Why does this avoid overfitting?

2) At test time we would need to average over all 2^M which is not feasible when 2^M is huge.

- **Approximation 2:** Instead, at test time we evaluate the full neural network where the weight are multiplied by p.

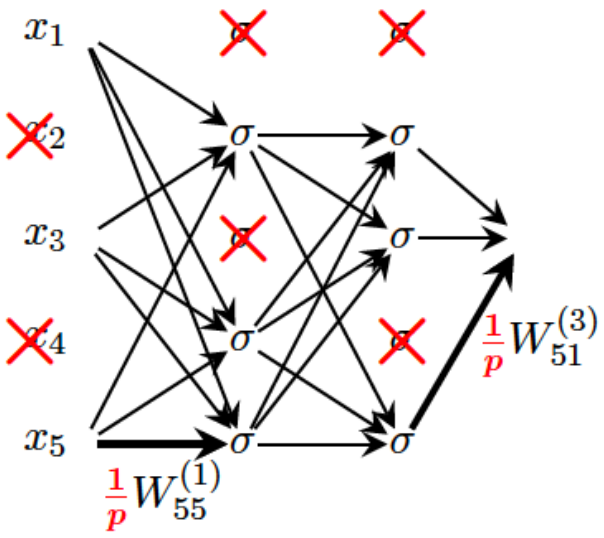


It has been empirically shown that this is a good approximation of the average of all ensemble members.

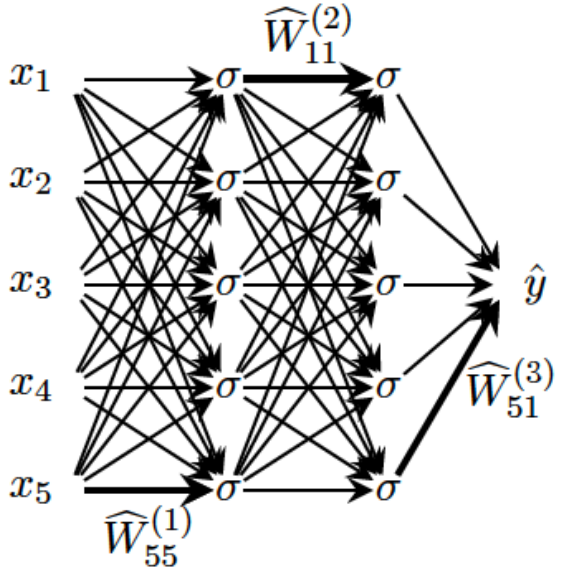
Dropout: Implementation

- **Current implementation:** scale factor p during training and testing
- **Advantage**: no need to remember at test time which p we used for training

During training



During testing



Reminder of Biais-variance decomposition

- Consider the true model $y = f(x) + \varepsilon$
- Let $\widehat{y} = \widehat{y}(x_*; \mathcal{D}_T)$ the prediction for the input sample x_* using the train dataset $\mathcal{D}_T$
- Task: Decomposition of expected mean square error of $\widehat{y}$

$$
\begin{aligned}
E_{\mathcal{D}_T}[(\widehat{y} - y)^2] &= \\
&= \\
&= \\
&= \underbrace{E_{\mathcal{D}_T}[(\widehat{y} - E_{\mathcal{D}_T}[\widehat{y}])^2]}_{\textit{variance}} + \underbrace{(E_{\mathcal{D}_T}[\widehat{y}] - f)^2}_{\textit{Bias}^2} + \underbrace{E_{\mathcal{D}_T}[\varepsilon^2]}_{\text{Irreducible error}}
\end{aligned}
$$

Decomposition for a specific input sample x_*.

Reminder of Biais-variance decomposition

Need average over all test samples

$$\underbrace{E_*[E_{\mathcal{D}_T}[(\widehat{y} - y)^2]]}_{\text{Expected MSE}} = \underbrace{E_*[E_{\mathcal{D}_T}[(\widehat{y} - E_{\mathcal{D}_T}[\widehat{y}])^2]]}_{\text{Variance}} + \underbrace{E_*[(E_{\mathcal{D}_T}([\widehat{y}] - f)^2]}_{\text{Bias}^2} + \underbrace{\sigma^2}_{\text{Irreducible error}}$$

$$\text{Expected MSE} \quad \text{average over test sample}$$

$$\text{Variance} \quad \text{average over test sample}$$

$$\text{Bias}^2 \quad \text{average over test sample}$$

- **Biais**: your model cannot **represent** the true model $f \longrightarrow \text{red}\{\text{Low model complexity}\}$
- **Variance**: Part of the MSE due to the variance in the training set, sensitivity of your model to the training data $\text{red}\{\text{High model complexity}\}$

Practice to track Biais and Variance

- Compare **Training data error** and **test data error**
- Bias is related to training error
- variance is related to the difference betwen test error and training error
- Short practice during our tutorial on MNIST data

Enesemble Methods: principle

- Let $\widehat{y}_1, \ldots, \widehat{y}_B$ be predictions from B different models
- $\widehat{y}_1, \ldots, \widehat{y}_B$ are identically distributed (might be not independent)
- $E[\widehat{y}_i] = \mu, \quad Var[\widehat{y}_i] = \sigma^2, \quad cor[\widehat{y}_i, \widehat{y}_j] = \varrho$

$$E\left[\frac{1}{B}\sum_{i=1}^{B}\widehat{y}_i\right] = \mu, \quad \text{and} \quad Var\left[\frac{1}{B}\sum_{i=1}^{B}\widehat{y}_i\right] = \frac{1-\varrho}{B}\sigma^2 + \varrho\sigma^2$$

-

Conclusion Model averaging does not affect **bias** but reduces **variance**

Bagging, Bootstrap aggregating

Aim: produces different prediction from models trained on different training dataset

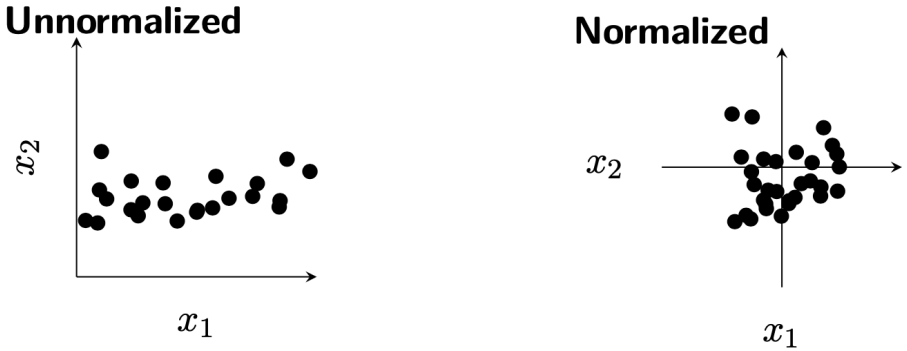
- IMPOSSIBLE: only one training dataset
- SOLUTION: Bootstrap your training data to mimic different dataset
- **sample data with replacement**
- Train a model on each of the rseampled data sets.
- Average their predictions

Difference between Bagging and Dropout

- **Bagging** all models are their own parameters while in **dropout** the different models (the sub-networks) share parameters.
- **Bagging** all models trained until convergence while **dropout** each sub-network is only trained for a singe gradient step.
- **Bagging** are trained on bootstrapped version of the whole data set while **dropout** sub-model is trained on randomly mini-batach of the data

Both Bagging and dropout are used to avoid overfitting and reduce the variance of the model

Batch Nomalization Normalizing inputs to speed up learning



- Compute mean and variance of training data

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{[i]}, \quad \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{[i]} - \mu_j)^2$$

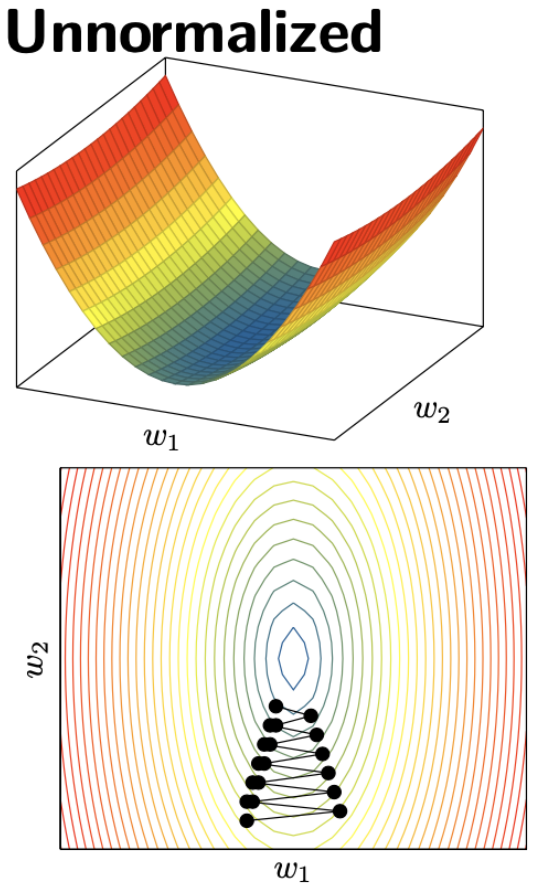
- Normalize

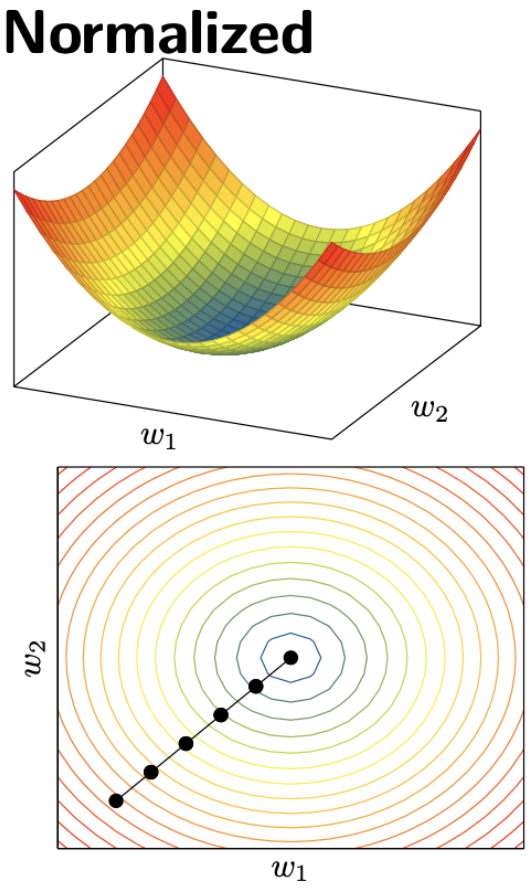
$$\widetilde{x}_j^{[i]} = \frac{x_j^{[i]} - \mu_j}{\sigma_j}$$

- μ_j and σ_j^2 are used to normalize validation/test data.

Why normalized the inputs data

-If inputs x_1 and x_2 are not normalized → cost function considered as *unormalized* → slower convergence.





Batch Normalization on each mini-batch

- Nomalized the inputs of each layer: **Batch Normalization (BN)**.
- Introduced in 2015 and it is one of the most efficient techniques for training deep neural networks.
- BN: enables to use higher learning rate without getting issues with vanishing or exploding gradients.
- BN: slight *regularization effect*.

Batch Normalization on each mini-batch

- Compute mean and variance for every unit j in all layers l

$$\mu_j^{(l)} = \frac{1}{m_{batch}} \sum_{i=1}^{m_{batch}} z_j^{(l)[i]}, \quad (\sigma_j^{(l)})^2 = \frac{1}{m} \sum_{i=1}^{m} (z_j^{(l)[i]} - \mu_j^{(l)})^2$$

where $z_j^{(l)[i]}$ is the hidden unit before the activation

- Normalize every unit j in all layers l

$$\bar{z}_j^{[i]} = \frac{z_j^{(l)[i]} - \mu_j^{(l)}}{\sqrt{(\sigma_j^{(l)})^2 + \varepsilon}}$$

- Scale and shift every unit

$$\tilde{z}_j^{[i]} = \gamma_j^{(l)} \bar{z}_j^{[i]} + \beta_j^{(l)}$$

where $\gamma_l^{(l}$ and $\beta_j^{(l)}$ are learned parameters (*called batch normalization layer*) that allow the new variable to have any mean and standard deviation.

Reading to understand: Why batch normalization ?

- The motivation of the prinsep paper is based on **internal covariante shift**: *Ioffe, Sergey, and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by reducing Internal Covariate Shift." International Conference on Machine Learning. 2015.*

- It has been recently shown that it makes the **loss landscape more smooth** and easier to optimize: *Santurkar, Shibani, et al. "How does batch normalization help optimization?." Advances in Neural Information Processing Systems. 2018*.

Take Home Message

- Overfitting
- Dropout
- Regularization
- Batch Normalization
- Bagging

 Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.