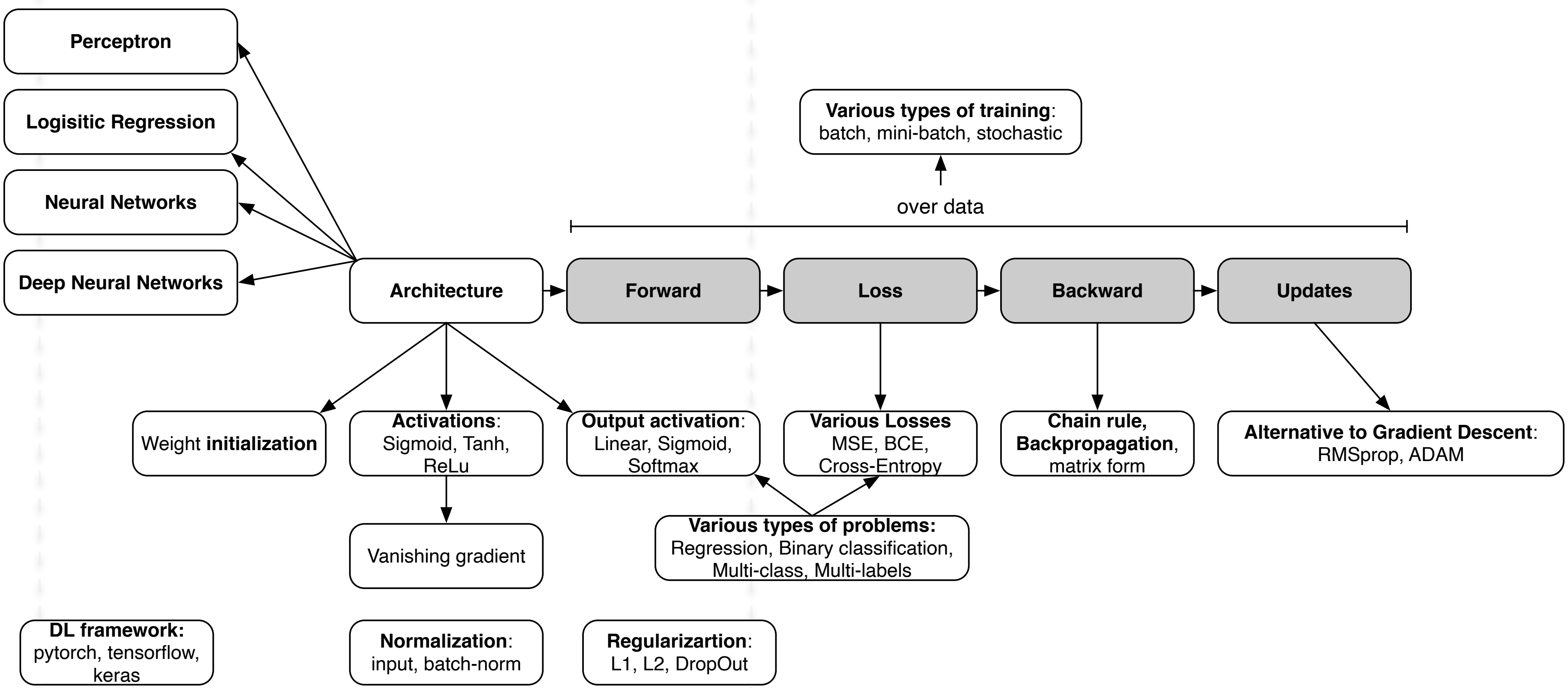
Deep Learning

Multi Layer Perceptron

Geoffroy Peeters

contact: geoffroy.peeters@telecom-paris.fr
Télécom-Paris, IP-Paris, France

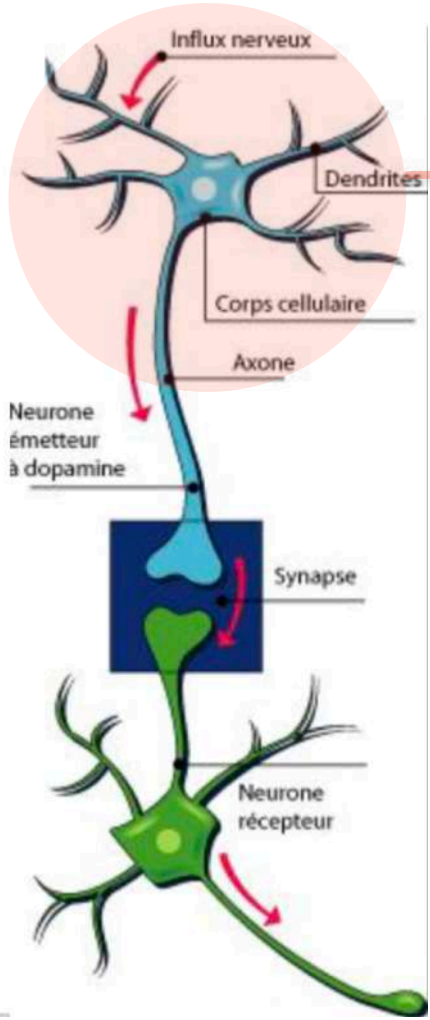
Overview

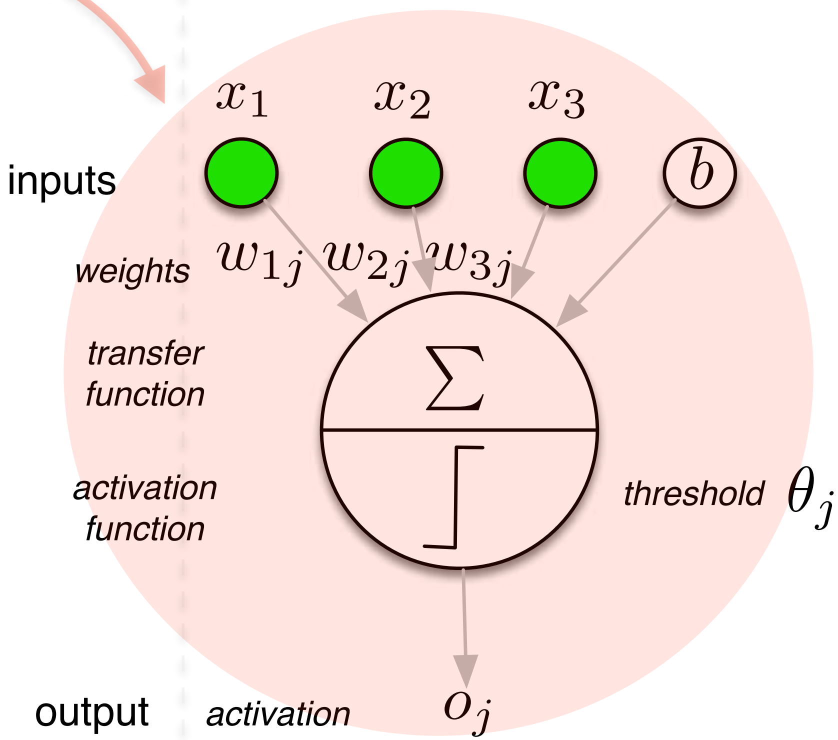


Perceptron
Logisitic Regression
Neural Networks
Deep Neural Networks
Various types of training: batch, mini-batch, stochastic
over data
Architecture
Forward
Loss
Backward
Updates
Weight initialization
Activations: Sigmoid, Tanh, ReLu
Output activation: Linear, Sigmoid, Softmax
Various Losses MSE, BCE, Cross-Entropy
Chain rule, Backpropagation, matrix form
Alternative to Gradient Descent: RMSprop, ADAM
Vanishing gradient
Various types of problems: Regression, Binary classification, Multi-class, Multi-labels
DL framework: pytorch, tensorflow, keras
Normalization: input, batch-norm
Regularizartion: L1, L2, DropOut

Perceptron

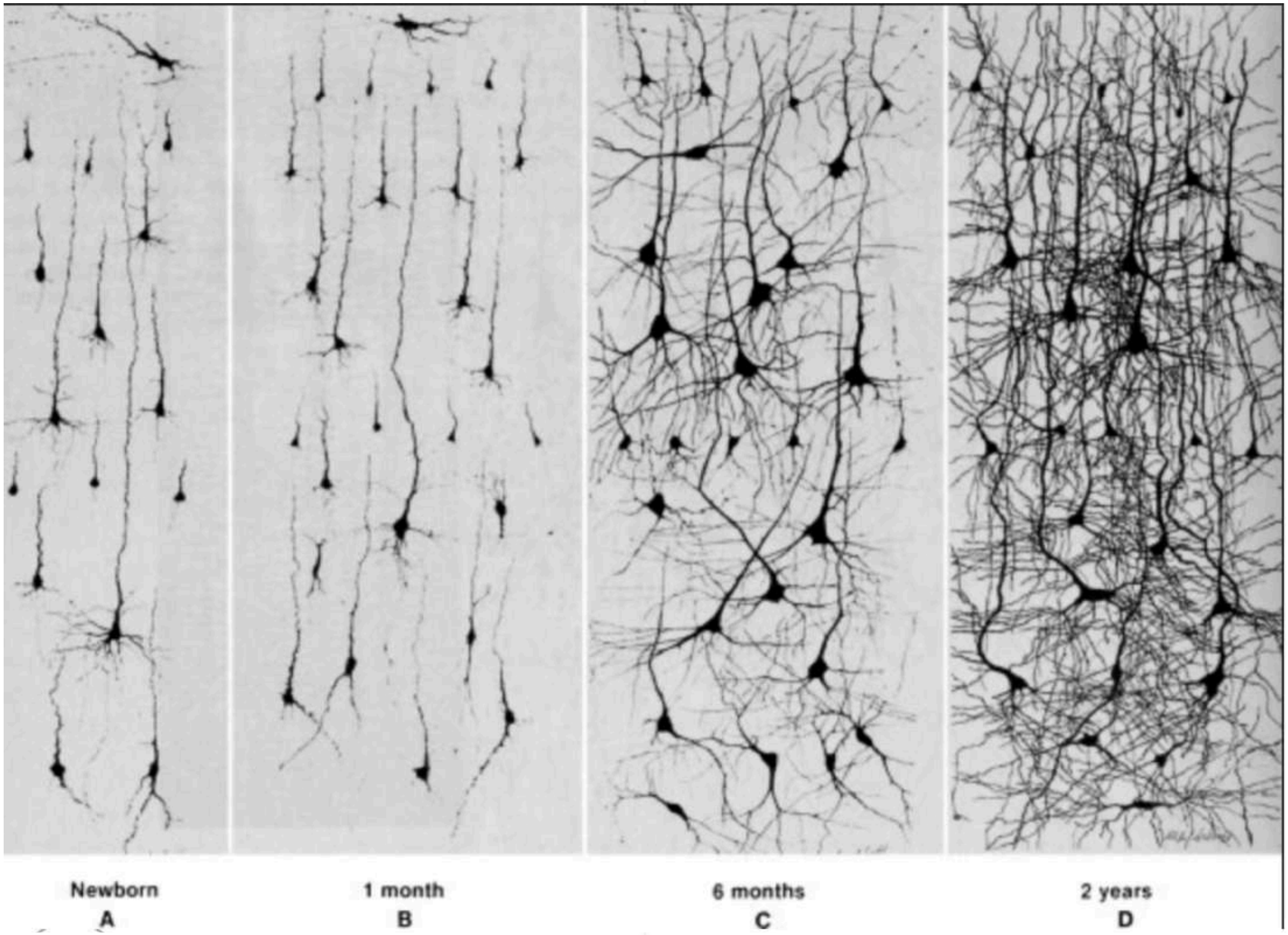
McCulloch and Pitts - mathematical model of a neuron





Warren S. McCulloch and Walter H. Pitts "A logical calculus of the ideas immanent in nervous activity", Bulletin of Mathematical Biophysics, vol. 5, 1943, p. 115-133

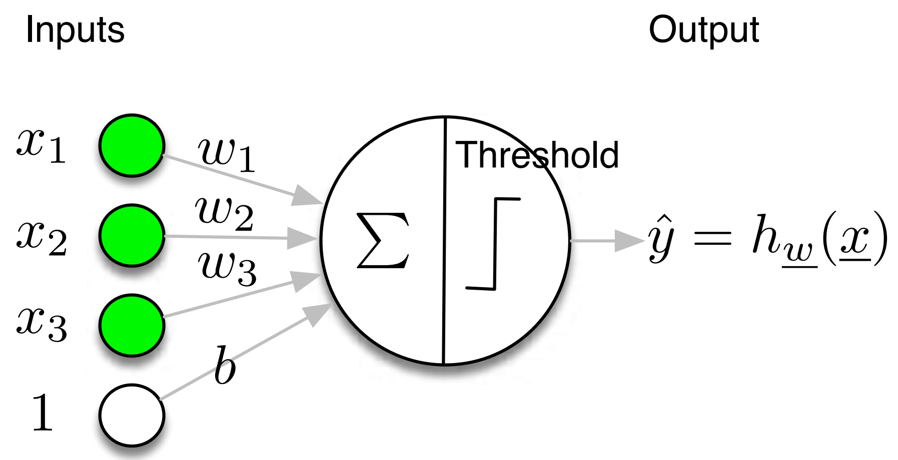
McCulloch - Pitts - mathematical model of a neuron



https://www.sciencedirect.com/science/article/pii/S0896627307007775

Perceptron model

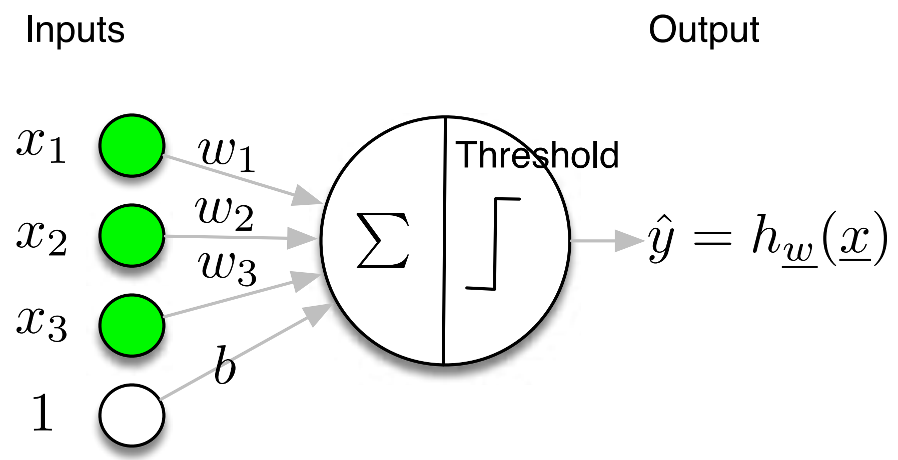
- **Problem:**
  - given $\mathbf{x} \in \mathbb{R}^{n_x}$,
  - predict $\hat{y} \in \{0,1\} \Rightarrow$ binary classification
- **Model:**
  - linear function followed by a threshold
    - $\hat{y} = h_{\mathbf{w}}(\mathbf{x})$
  - Formulation 1
    - $\hat{y} =$ Threshold$(\mathbf{x}^T \cdot \mathbf{w})$ with $\mathbf{x}_0 = 1, \mathbf{w}_0 = b$
  - Formulation 2
    - $\hat{y} =$ Threshold$(\mathbf{x}^T \cdot \mathbf{w} + b)$
  - where
    - Threshold$(z) = 1$ if $z \geq 0$
    - Threshold$(z) = 0$ if $z < 0$
- **Parameters:**
  - $\theta = \{\mathbf{w} \in \mathbb{R}^{n_x}, b \in \mathbb{R}\}$
    - $\mathbf{w}$: weights
    - b: bias



Frank Rosenblatt, "The perceptron : a perceiving and recognizing automation", Projet PARA, Cornell Aeronautical Laboratory, Inc. 1957

Perceptron training (empirical)

- **Training**:
  - Adapt $\mathbf{w}$ and b such that $\hat{y} = h_{\mathbf{w}}(\mathbf{x})$ equal y on training data



- **Algorithm:**
  - $\forall i \quad (\mathbf{x}^{(i)}, y^{(i)})$
    - compute $\hat{y}^{(i)} = h_{\mathbf{w}}(\mathbf{x}^{(\mathbf{i})}) =$ Threshold$(\mathbf{x}^{\mathbf{T}^{(i)}} \cdot \mathbf{w})$
    - if $\hat{y}^{(i)} \neq y^{(i)}$ update
      - $w_d \leftarrow w_d + \alpha(y^{(i)} - \hat{y}^{(i)})x_d^{(i)} \quad \forall d$
      - where α is the learning rate

- **Three cases:**
  - $(y^{(i)} - \hat{y}^{(i)}) = 0,$ no update
  - $(y^{(i)} - \hat{y}^{(i)}) = +1,$ update, the weights are too low, increase w_d for positive $x_d^{(i)}$
  - $(y^{(i)} - \hat{y}^{(i)}) = -1,$ update, the weights are too high, decrease w_d for positive $x_d^{(i)}$

Perceptron training (minimising a loss)

- **Perceptron**:
  - predict a **class**
  - model: $\hat{y}^{(i)} = h_{\mathbf{w}}(\mathbf{x}^{(\mathbf{i})}) = \text{Threshold}(\mathbf{x}^{\mathbf{T}^{(i)}} \cdot \mathbf{w})$
  - update rule: $w_d \leftarrow w_d + \alpha(y^{(i)} - \hat{y}^{(i)})x_d^{(i)} \quad \forall d$
- It corresponds to **minimising a loss function**

$$\mathscr{L}(y^{(i)}, \hat{y}^{(i)}) = -\,(y^{(i)} - \hat{y}^{(i)})\ \mathbf{x}^{\mathbf{T}^{(i)}} \cdot \mathbf{w}$$

- **Gradient descent ?**

$$w_d \leftarrow w_d - \alpha \frac{\partial \mathscr{L}(y^{(i)}, \hat{y}^{(i)})}{\partial w_d} \quad \forall d$$

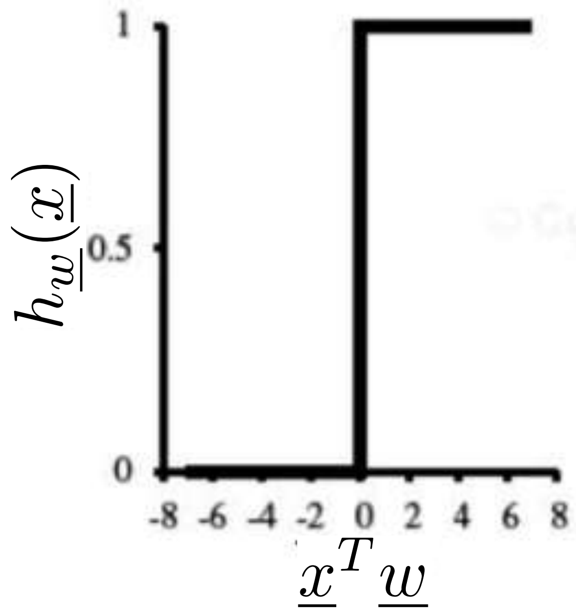
$$\leftarrow w_d + \alpha(y^{(i)} - \hat{y}^{(i)})x_d^{(i)} \quad \forall d$$

- **Gradient ?**

$$\frac{\partial \mathscr{L}}{\partial w_d} = -\left(\frac{\partial (y^{(i)} - \hat{y}^{(i)})}{\partial w_d}\mathbf{x}^{\mathbf{T}^{(i)}}\mathbf{w} + (y^{(i)} - \hat{y}^{(i)})x_d^{(i)}\right)$$

$$= -\left(0 + \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_d^{(i)}\right)$$

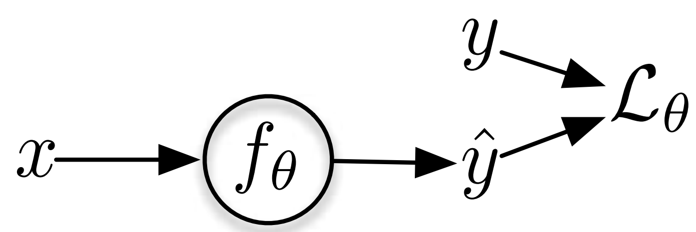
  - **Problem**: the derivative of $h_{\mathbf{w}}(\mathbf{x}^{(\mathbf{i})})$ does not exist at $\mathbf{x}^{\mathbf{T}}\mathbf{w} = 0$
    - $\Rightarrow$ Using the Threshold function lead to instability during training



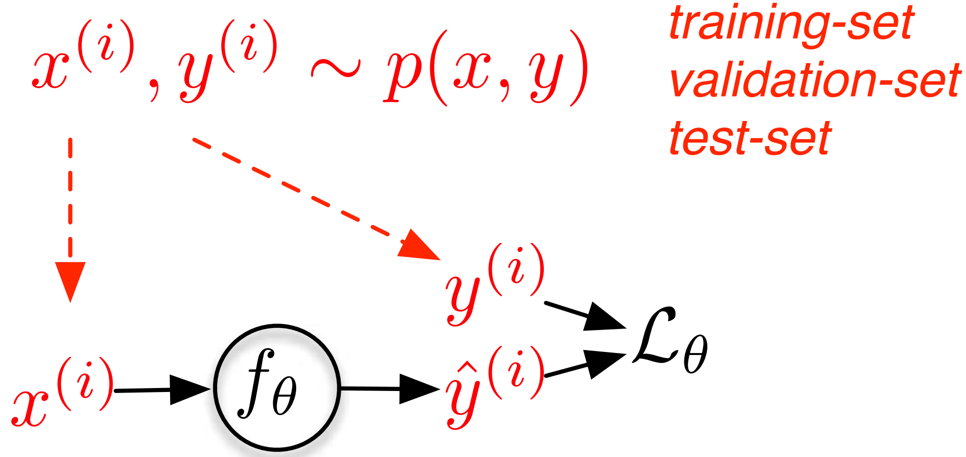
Empirical Risk Minimisation
Loss minimisation and Gradient Descent

Loss minimisation and Gradient Descent

- **Risk minimisation**
  - A neural network is a function $f_\theta(\mathbf{x})$ of parameters θ ($\mathbf{w}, b$ in the example) which gives a prediction $\hat{y}$ of a ground-truth value y
  - The goal is to find the parameters θ that minimises a **risk**, or a **loss** $\mathscr{L}_\theta(\hat{y}, y)$

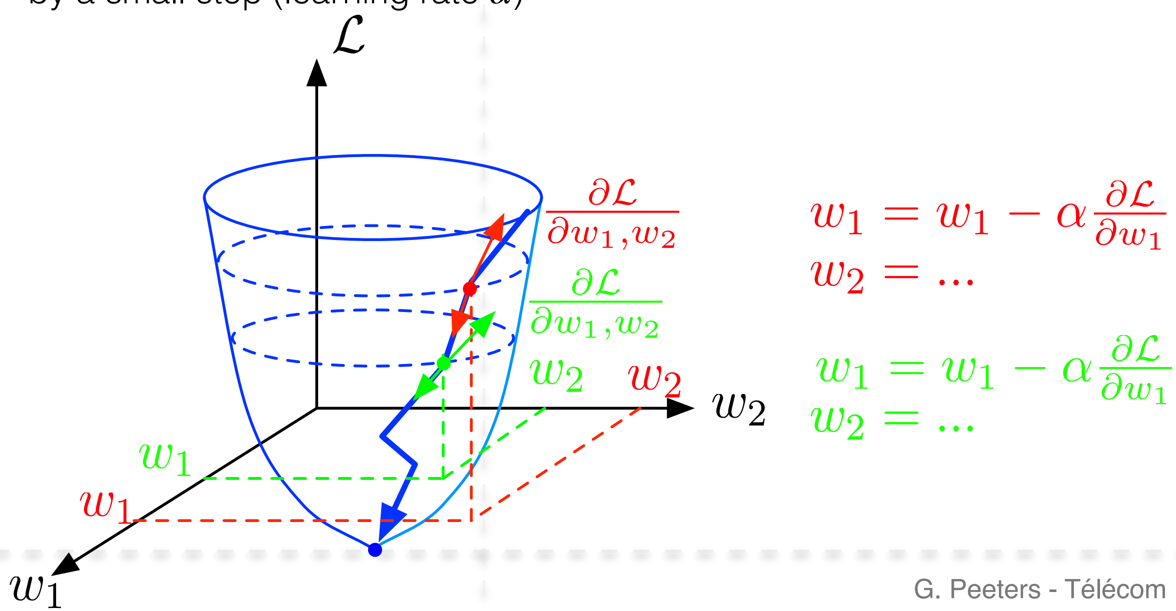


- **Empirical Risk minimisation**
  - we do not observe $p(x, y)$ but only samples from this distribution $x^{(i)}, y^{(i)} \sim p(x, y)$ (dataset: train/valid/test)
  - find the parameters θ that minimises empirically a **risk**, or a **loss** $\mathscr{L}_\theta(\hat{y}^{(i)}, y^{(i)})$ over the dataset (train/valid/test)



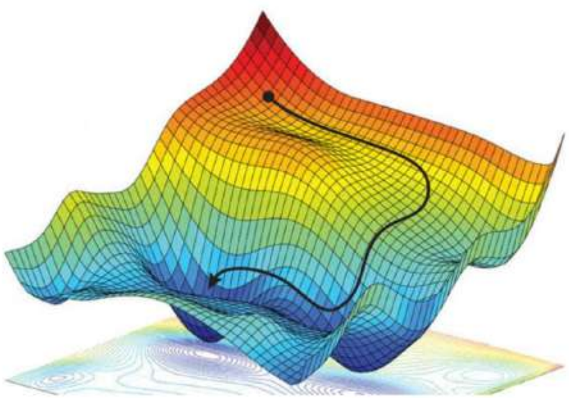
Loss minimisation and Gradient Descent

- **Gradient Descent algorithm**
  - We compute the dependency of the Loss $\mathscr{L}_\theta$ w.r.t. to the parameters θ
  - Dependency ?
    - partial derivatives $\frac{\partial \mathscr{L}}{\partial w_1}, \frac{\partial \mathscr{L}}{\partial w_2}, \frac{\partial \mathscr{L}}{\partial b} \Rightarrow$ the gradient vector $\frac{\partial \mathscr{L}}{\partial \underline{\theta}}$
  - $\frac{\partial \mathscr{L}}{\partial \underline{\theta}}$ indicates the direction of maximum increase of the loss
  - To reduce the loss, we progressively move down the hill in the opposite direction of $\frac{\partial \mathscr{L}}{\partial \underline{\theta}}$
    - progressively= by a small step (learning rate α)

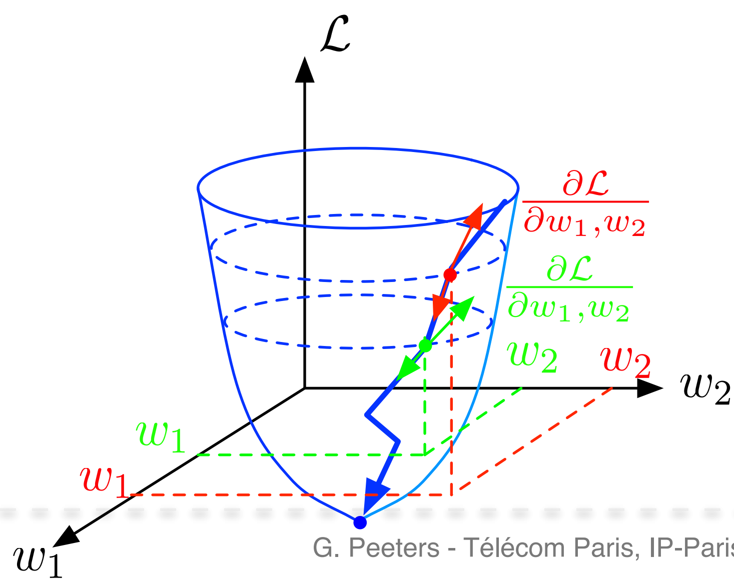


Deep Learning issues

- Need to define a **Loss** $\mathscr{L}$, **output activation**
  - depends on the problem: binary classification, muti-class, multi-label, regression, ...
- Need to be able to compute derivatives $\frac{\partial \mathscr{L}}{\partial \underline{\theta}}$
  - use **differentiable functions** (not the Threshold !)
- In deep learning $\mathscr{L}_\theta(\hat{y}, y)$ is a non-convex function
  - use more elaborate **optimisation algorithm** (Momentum, NAG, ADAM, ...)
- Need an efficient way to compute the derivatives $\frac{\partial \mathscr{L}}{\partial \underline{\theta}}$
  - the **back-propagation algorithm**
- How to sample $x^{(i)}, y^{(i)} \sim p(x, y)$
  - **batch GD, mini-batch GD, stochastic GD**

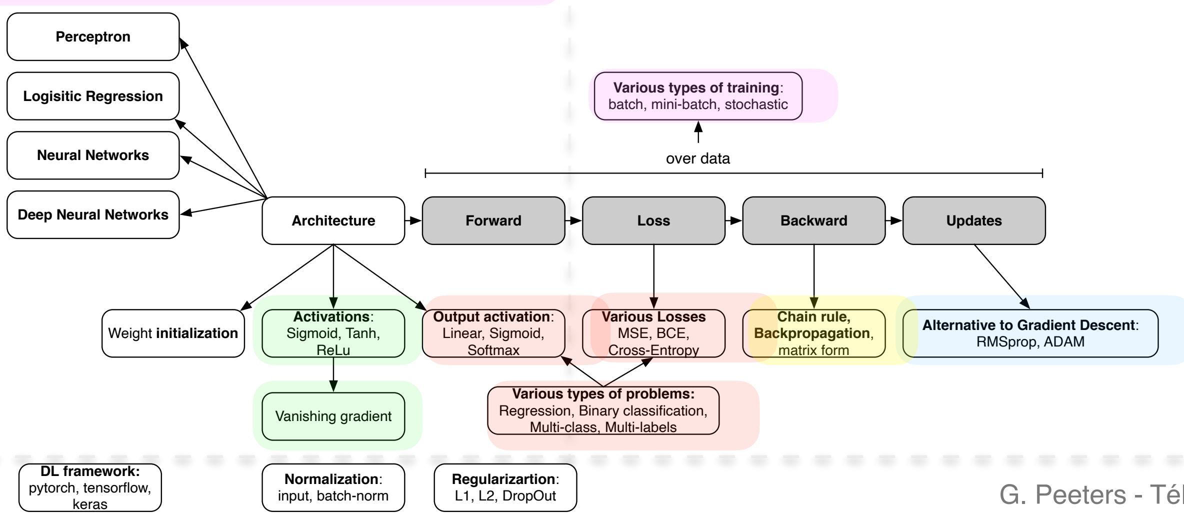


https://medium.com/swlh/non-convex-optimization-in-deep-learning-26fa30a2b2b3



Deep Learning issues

- Need to define a **Loss** $\mathscr{L}$, **output activation**
  - depends on the problem: binary classification, muti-class, multi-label, regression, ...
- Need to be able to compute derivatives $\dfrac{\partial \mathscr{L}}{\partial \underline{\theta}}$
  - use **differentiable functions** (not the Threshold !)
- In deep learning $\mathscr{L}_\theta(\hat{y}, y)$ is a non-convex function
  - use more elaborate **optimisation algorithm** (Momentum, NAG, ADAM, ...)
- Need an efficient way to compute the derivatives $\dfrac{\partial \mathscr{L}}{\partial \underline{\theta}}$
  - the **back-propagation algorithm**
- How to sample $x^{(i)}, y^{(i)} \sim p(x, y)$
  - **batch GD, mini-batch GD, stochastic GD**

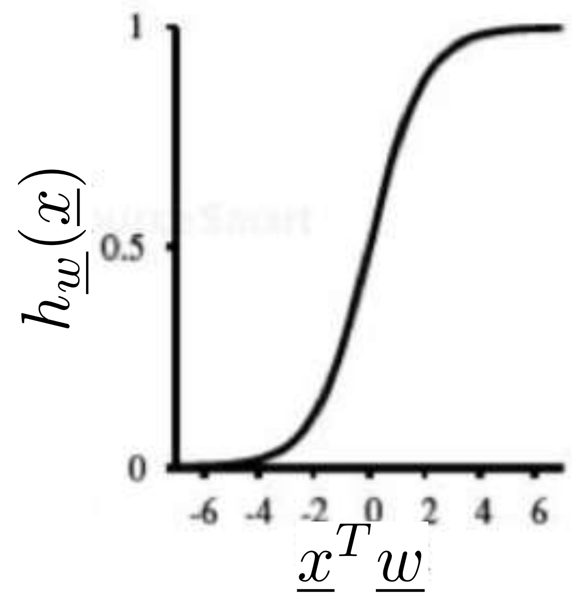


From Perceptron to Logistic Regression

- **Logistic regression**
  - predicts a **probability to belong to a class**
  - model

$$\hat{y} = p(Y=1\,|\,\mathbf{x}) = h_{\mathbf{w}}(\mathbf{x}) = \text{Logistic}(\mathbf{x}^{\mathbf{T}} \cdot \mathbf{w}) = \frac{1}{1+e^{-\mathbf{x}^{\mathbf{T}} \cdot \mathbf{w}}}$$

  - decision
    - if $h_{\mathbf{w}}(\mathbf{x}) \geq 0.5$ choose class 1
    - if $h_{\mathbf{w}}(\mathbf{x}) < 0.5$ choose class 0



- We define the **Loss** as (binary-cross-entropy)

$$\mathcal{L}(y^{(i)}, \hat{y}^{(i)}) = -\left(y^{(i)} \log(\hat{y}^{(i)}) + (1-y^{(i)}) \log(1-\hat{y}^{(i)})\right)$$

- **Gradient descent ?**

$$w_d \leftarrow w_d - \alpha \frac{\partial \mathcal{L}(y^{(i)}, \hat{y}^{(i)})}{\partial w_d} \quad \forall d$$

$$\leftarrow w_d + \alpha (y^{(i)} - \hat{y}^{(i)}) x_d^{(i)} \quad \forall d$$

- **Gradient ?**

$$\frac{\partial \mathcal{L}}{\partial w_d} = -(y^{(i)} - \hat{y}^{(i)}) \cdot x_d^{(i)} \quad \forall d$$

  - The update rule is therefore the same as for the Perceptron but the definitions of the Loss $\mathcal{L}$ and of $h_{\mathbf{w}}(\mathbf{x})$ are different

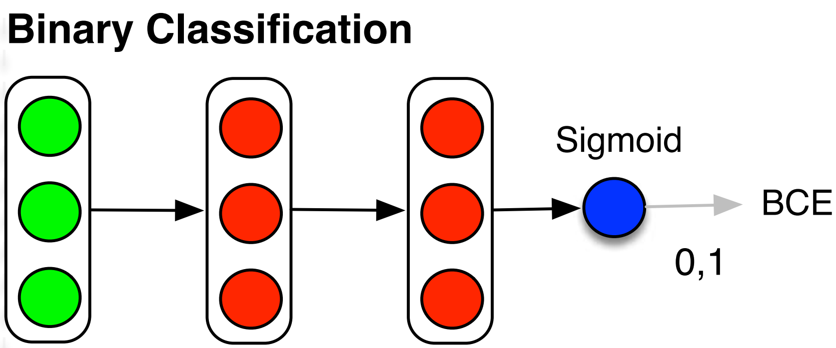
Binary classification

Binary classification

Model

– We have several a single output neurons $\hat{y}$
  - with a **sigmoid** activation function

– We minimise the **binary-cross-entropy**
$$\mathscr{L} = -\left(y\log(\hat{y}) + (1-y)\log(1-\hat{y})\right)$$



Binary classification

Output activation function: sigmoid/ logistic

- **Usage**: binary classification (0 or 1)
  - (logistic regression or deep neural network with binary output)

$$\begin{cases} P(y=1|\mathbf{x}) & = \sigma(\mathbf{x}^{\mathbf{T}}\mathbf{w}) = \boxed{\dfrac{1}{1+e^{-(\mathbf{x}^{\mathbf{T}}\mathbf{w})}}} \\ P(y=0|\mathbf{x}) & = 1 - P(y=1|\mathbf{x}) \end{cases}$$

- **Models the log-odds** $\log \dfrac{p}{1-p}$ (posterior probability) of the value "1" using a linear model of the inputs $\mathbf{x}$

$$P(y=1|\mathbf{x}) = \frac{1}{1+e^{-\mathbf{x}^{\mathbf{T}}\mathbf{w}}}$$

$$\frac{1}{P(y=1|\mathbf{x})} = 1 + e^{-\mathbf{x}^{\mathbf{T}}\mathbf{w}}$$

$$\frac{1-P(y=1|\mathbf{x})}{P(y=1|\mathbf{x})} = e^{-\mathbf{x}^{\mathbf{T}}\mathbf{w}}$$

$$\log\left(\frac{P(y=0|\mathbf{x})}{P(y=1|\mathbf{x})}\right) = -\,\mathbf{x}^{\mathbf{T}}\mathbf{w}$$

$$\log\left(\frac{P(y=1|\mathbf{x})}{P(y=0|\mathbf{x})}\right) = \mathbf{x}^{\mathbf{T}}\mathbf{w}$$

Binary Cross-Entropy (1)

- The ground-truth output y is a **binary** variable $\in \{0,1\}$
  - y follows a **Bernoulli** distribution: $P(Y=y) = p^y(1-p)^{1-y} \quad y \in \{0,1\}$
  - $\begin{cases} P(Y=1) & = p \\ P(Y=0) & = 1-p \end{cases}$

- In logistic regression, the output of the network $\hat{y}$ estimates p: $\hat{y} = P(Y=1 \mid x, \theta)$
  - $P(Y=y \mid x, \theta) = \hat{y}^y(1-\hat{y})^{1-y} \quad y \in \{0,1\}$

- We want to
  - find the θ that ... maximise the **likelihood** of the y given the input x
    - $\max_\theta P(Y=y \mid x, \theta) = \hat{y}^y(1-\hat{y})^{1-y} \quad y \in \{0,1\}$
  - ... that maximise the **log-likelihood**
    - $\max_\theta \log p(y \mid x) = \log(\hat{y}^y\,(1-\hat{y})^{(1-y)}) = y\log(\hat{y}) + (1-y)\log(1-\hat{y}) = -\mathscr{L}(\hat{y}, y)$
  - ... that minimise a loss named **Binary Cross-Entropy** (BCE)
    - $\min_\theta \mathscr{L}(\hat{y}, y) = -\left(y\log(\hat{y}) + (1-y)\log(1-\hat{y})\right)$

Binary Cross-Entropy (2)

- ... maximising **log-likelihood** on the whole training set

$$p(\text{labels}) = \prod_{i=1}^{m} p(y^{(i)} | x^{(i)})$$

$$\log p(\text{labels}) = \log\left(\prod_{i=1}^{m} p(y^{(i)} | x^{(i)})\right) = \sum_{i=1}^{m} \log p(y^{(i)} | x^{(i)}) = \sum_{i=1}^{m} - \mathscr{L} y^{(i)}, y^{(i)})$$

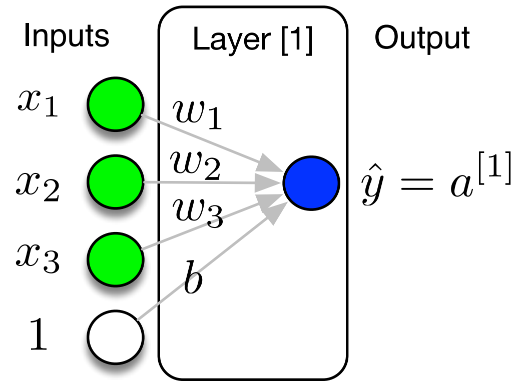
- ... is equivalent to minimising the **Cost** $J(\theta) = \dfrac{1}{m} \sum_{i=1}^{m} \mathscr{L} y^{(i)} | x^{(i)})$

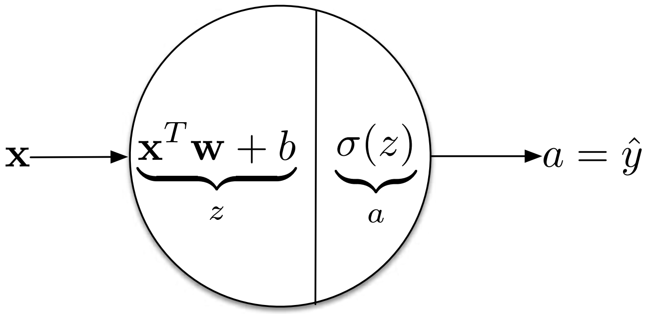
Logistic Regression (0 hidden layers)

Logistic Regression (0 hidden layers)

$$\underline{x} \rightarrow \boxed{w_1x_1 + w_2x_2 + b} \xrightarrow{z} \boxed{\sigma(z)} \xrightarrow{a} \boxed{\mathcal{L}(\hat{y} = a, y)}$$

- **Problem**:
  - Given $\mathbf{x} \in \mathbb{R}^{n_x}$, predict $\hat{y} = P(y = 1 \,|\, \mathbf{x})$ $\hat{y} \in [0,1]$
- **Model**
  - $\hat{y} = \sigma(\mathbf{x}^{\mathbf{T}} \cdot \mathbf{w})$
  - Sigmoid function: $\sigma(z) = \dfrac{1}{1 + e^{-z}}$
    - if z is very large then $\sigma(z) \simeq \dfrac{1}{1+0} = 1$
    - if z is very small (negative) then $\sigma(z) = 0$
- **Parameters**
  - $\theta = \{\mathbf{w} \in \mathbb{R}^{n_x}, b \in \mathbb{R}\}$





Loss/ Cost function (Empirical Risk Minimisation)

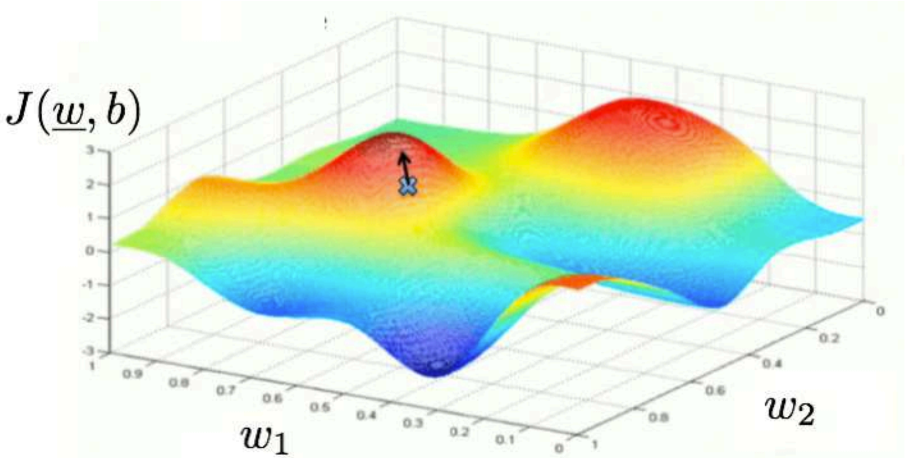
- **Training data**
  - Given $\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\}$
  - We want to find the parameters $\theta$of the network such that $\hat{y}^{(i)} \simeq y^{(i)}$

- How to measure ? Define a **Loss $\mathscr{L}$ (error) function** which needs to be minimised
  - if $y \in \mathbb{R}$: **Mean-Square-Error (MSE)**: $\mathscr{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$
  - if $y \in \{0,1\}$:**Binary-Cross-Entropy (BCE)**: $\mathscr{L}(\hat{y}, y) = -\left(y\log(\hat{y})) + (1-y)\log(1-\hat{y})\right)$
  - if $y \in \{1,\ldots,K\}$: **Cross-Entropy**: $\mathscr{L}(\hat{y}, y) = -\sum_{c=1}^{K}\left(y_c\log(\hat{y}_c))\right)$

- **Cost J function**= sum of the Loss for all training examples
  - $J_\theta = \frac{1}{m}\sum_{i=1}^{m}\mathscr{L}_\theta(\hat{y}^{(i)}, y^{(i)})$
    - In the case of the BCE: $J_\theta = -\sum_{i=1}^{m}\left(y^{(i)}\log(\hat{y}^{(i)})) + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right)$
  - We want to find the parameters θ of the network that minimise J_θ

Gradient Descent

- **How to minimise J_θ ?**
  - The gradient $\dfrac{\partial J_\theta}{\partial \mathbf{w}}$ points in the direction of the greatest rate of increase of the function
  - We will go in the opposite direction: $-\dfrac{\partial J_\theta}{\partial \mathbf{w}}$
    - We move down the hill in the steepest direction

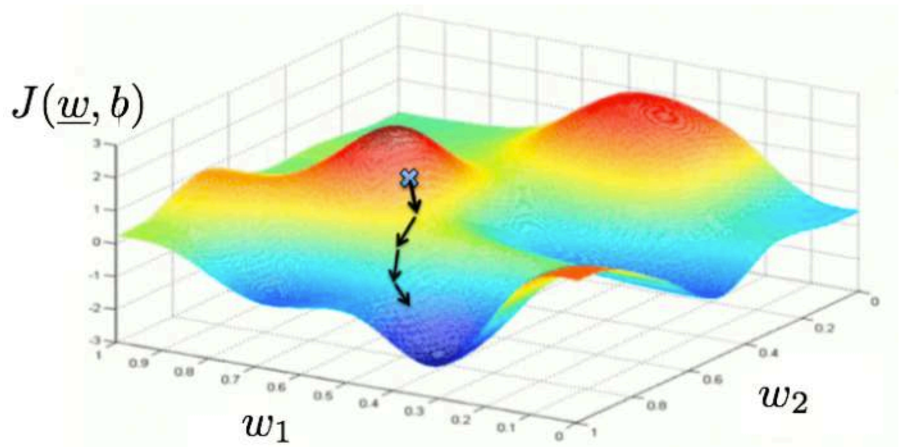


- **Gradient descent:**
  - Repeat

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial J_\theta}{\partial \mathbf{w}}$$

$$b \leftarrow b - \alpha \frac{\partial J_\theta}{\partial b}$$

- where α is the **"learning rate"**



Gradient Descent

- **Parameters to update**
  - $\theta = \{\mathbf{w}, b\}$

- **Gradient descent:**
  - **Initialise** the parameters θ
  - Repeat for # iterations
    - Repeat for all training examples $\forall i \in \{1, \ldots, m\}$
      - **Forward computation:** compute the prediction $\hat{y}^{(i)}$
      - **Compute the Loss** $\mathscr{L}_\theta(\hat{y}^{(i)}, y^{(i)})$
      - **Backward propagation:** compute the gradients: $\dfrac{\partial \mathscr{L}_\theta(\hat{y}^{(i)}, y^{(i)})}{\partial \mathbf{w}}, \dfrac{\partial \mathscr{L}_\theta(\hat{y}^{(i)}, y^{(i)})}{\partial b}$
    - Compute the Cost J_θ
    - **Update the parameters** θ using the learning rate α

Logistic Regression (0 hidden layers)

Forward propagation

$$\underline{x} \rightarrow \boxed{w_1 x_1 + w_2 x_2 + b} \xrightarrow{z} \boxed{\sigma(z)} \xrightarrow{a} \boxed{\mathcal{L}(\hat{y} = a, y)}$$

$$z^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(2)} + b$$

$$a^{(i)} = \sigma(z^{(i)}) \qquad \text{with } \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$-$$

$$\hat{y}^{(i)} = a^{(i)}$$

Logistic Regression (0 hidden layers)

Loss

$$\underline{x} \rightarrow \boxed{w_1 x_1 + w_2 x_2 + b} \xrightarrow{z} \boxed{\sigma(z)} \xrightarrow{a} \boxed{\mathcal{L}(\hat{y} = a, y)}$$

$$\hat{y}^{(i)} = a^{(i)}$$

$$\mathcal{L}_\theta(\hat{y}^{(i)}, y^{(i)}) = -\left(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})\right)$$

Logistic Regression (0 hidden layers)

Backward propagation

(we ommit $^{(i)}$ in the following)

$$\underline{x} \rightarrow \boxed{w_1x_1 + w_2x_2 + b} \xrightarrow{z} \boxed{\sigma(z)} \xrightarrow{a} \boxed{\mathcal{L}(\hat{y} = a, y)}$$

$$\frac{\partial \mathscr{L}}{\partial a} = -\left(\frac{y}{a} - \frac{1-y}{1-a}\right) = \frac{a-y}{a(1-a)}$$

$$\frac{\partial \mathscr{L}}{\partial z} = \frac{\partial \mathscr{L}}{\partial a}\frac{da}{dz} = \frac{a-y}{a(1-a)} \cdot a(1-a) = a - y$$

$$\frac{\partial \mathscr{L}}{\partial w_1} = \frac{\partial \mathscr{L}}{\partial z}\frac{\partial z}{\partial w_1} = x_1 \cdot \frac{\partial \mathscr{L}}{\partial z}$$

$$\frac{\partial \mathscr{L}}{\partial w_2} = \frac{\partial \mathscr{L}}{\partial z}\frac{\partial z}{\partial w_2} = x_2 \cdot \frac{\partial \mathscr{L}}{\partial z}$$

$$\frac{\partial \mathscr{L}}{\partial b} = \frac{\partial \mathscr{L}}{\partial z}\frac{\partial z}{\partial b} = \frac{\partial \mathscr{L}}{\partial z}$$

Chain rule and back-propagation

Chain rule and back-propagation

What is the chain rule ?

- formula for computing the derivative of the composition of two or more functions

$$\frac{\partial}{\partial t} f(x(t)) = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t}$$

$$\frac{\partial}{\partial t} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}$$

- Example 1:

$$\begin{aligned} z_1 &= z_1(x_1, x_2) \\ z_2 &= z_2(x_1, x_2) \\ p &= p(z_1, z_2) \\ \frac{\partial p}{\partial x_1} &= \frac{\partial p}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial z_2}\frac{\partial z_2}{\partial x_1} \end{aligned}$$

- Example 2:

$$h(x) = f(x)g(x)$$

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial f}\frac{\partial f}{\partial x} + \frac{\partial h}{\partial g}\frac{\partial g}{\partial x} = f'g + fg'$$

What is back-propagation ?

- an efficient algorithm to compute the chain-rule by storing intermediate (and re-use derivatives)

Example 1: Logistic regression / least square (single output)

$$z = xw + b$$
$$\hat{y} = a = \sigma(z)$$
$$\mathscr{L} = \frac{1}{2}(\hat{y} - y)^2$$

- **Computing derivative as in calculus class**

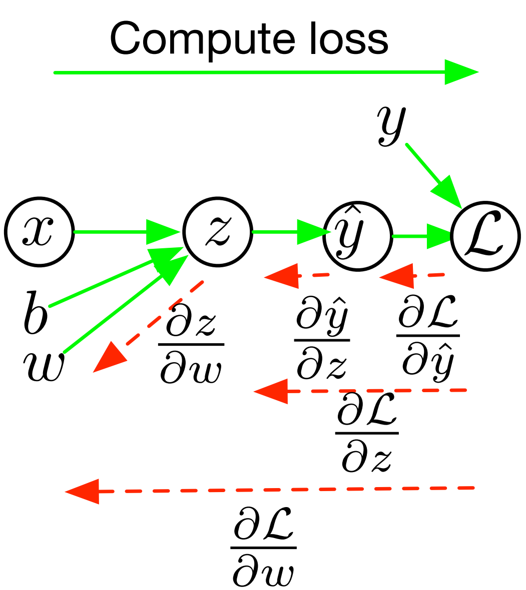
$$\mathscr{L} = \frac{1}{2}\left(\sigma(wx + b) - y\right)^2$$

$$\begin{aligned}
\frac{\partial \mathscr{L}}{\partial w} &= \frac{\partial}{\partial w}\left[\frac{1}{2}(\sigma(wx + b) - y)^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial w}\left(\sigma(wx + b) - y\right)^2 \\
&= \left(\sigma(wx + b\right) - y)\frac{\partial}{\partial w}\left(\sigma(wx + b) - y\right) \\
&= \left(\sigma(wx + b\right) - y)\sigma'(wx + b)\frac{\partial}{\partial w}(wx + b) \\
&= (\sigma(wx + b) - y)\sigma'(wx + b)x
\end{aligned}$$

Chain rule and back-propagation

Example 1: Logistic regression / least square (single output)

$$z = xw + b$$
$$\hat{y} = a = \sigma(z)$$
$$\mathscr{L} = \frac{1}{2}(\hat{y} - y)^2$$



- **Computing derivative using back-propagation**

$$\frac{\partial \mathscr{L}}{\partial \hat{y}} = \hat{y} - y$$
$$\frac{\partial \mathscr{L}}{\partial z} = \frac{\partial \mathscr{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} = \frac{\partial \mathscr{L}}{\partial \hat{y}} \sigma'(z)$$
$$\frac{\partial \mathscr{L}}{\partial w} = \frac{\partial \mathscr{L}}{\partial z} \frac{\partial z}{\partial w} = \frac{\partial \mathscr{L}}{\partial z} x$$
$$\frac{\partial \mathscr{L}}{\partial b} = \frac{\partial \mathscr{L}}{\partial z} \frac{\partial z}{\partial b} = \frac{\partial \mathscr{L}}{\partial z}$$

- We can diagram out the computations using a **computation graph**
  - nodes represent all the inputs and computed quantities,
  - edges represent which nodes are computed directly as a function of which other nodes

Chain rule and back-propagation

Example 2: MLP / least square (1 hidden layer, multiple outputs)

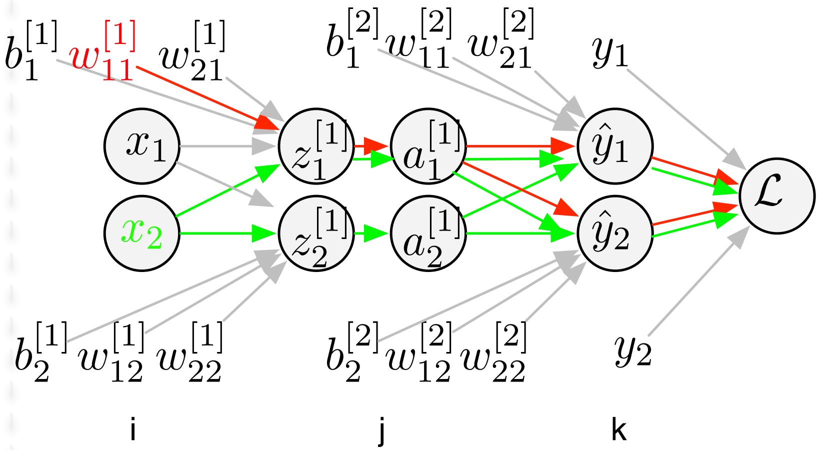
$$z_j^{[1]} = \sum_i x_i w_{ij}^{[1]} + b_j^{[1]}$$

$$a_j^{[1]} = \sigma(z_j^{[1]})$$

$$z_k^{[2]} = \sum_j a_j^{[1]} w_{jk}^{[2]} + b_k^{[2]}$$

$$\hat{y}_k = a_k^{[2]} = z_k^{[2]}$$

$$\mathcal{L} = \frac{1}{2} \sum_k (\hat{y}_k - y_k)^2$$



- **Computing derivative using back-propagation**
  - How much changing w_{11} or x_2 affect $\mathcal{L}$?
  - We need to take into account all possible paths

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_k} = \hat{y}_k - y_k$$

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{[2]}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_k} a_j^{[1]} \qquad \frac{\partial \mathcal{L}}{\partial b_k^{[2]}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_k}$$

$$\frac{\partial \mathcal{L}}{\partial a_j^{[1]}} = \sum_k \frac{\partial \mathcal{L}}{\partial \hat{y}_k} w_{jk}^{[2]}$$

$$\frac{\partial \mathcal{L}}{\partial z_j^{[1]}} = \frac{\partial \mathcal{L}}{\partial a_j^{[1]}} \sigma'(z_j)$$

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{[1]}} = \frac{\partial \mathcal{L}}{\partial z_j^{[1]}} x_i \qquad \frac{\partial \mathcal{L}}{\partial b_j^{[1]}} = \frac{\partial \mathcal{L}}{\partial z_j^{[1]}}$$

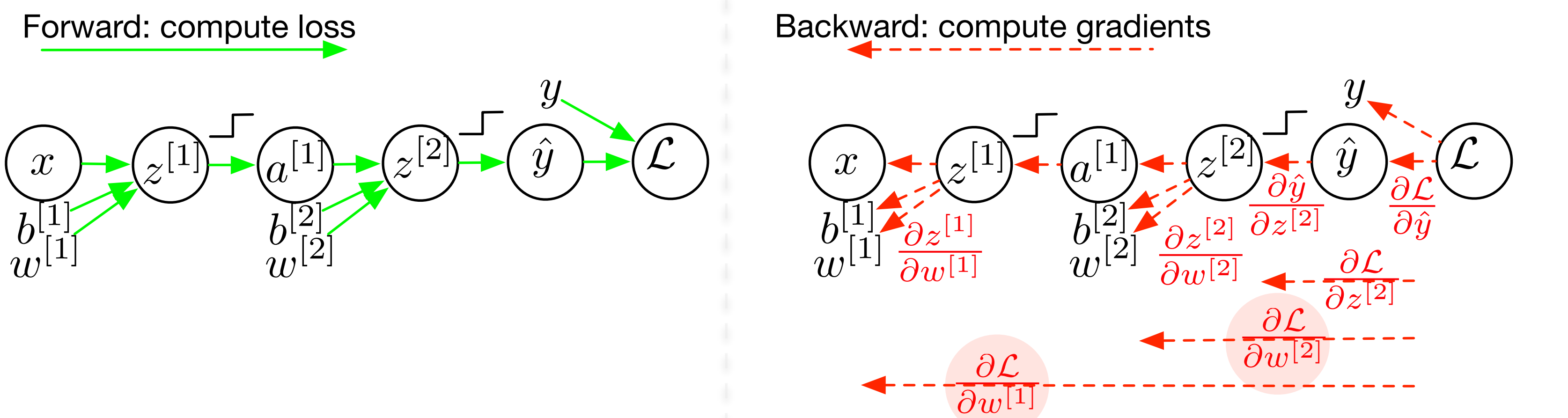
How to compute efficiently the gradients ? The back-propagation algorithm

- **Chain-rule**

$$\frac{d}{dt} f(x(t)) = \frac{df}{dx}\frac{dx}{dt}$$

$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

- **Back-propagation**
  - an efficient way to compute the chain-rule as a succession of partial chain-rules (minimum storage, maximum re-use)



$$w^{[1]} = w^{[1]} - \alpha \frac{\partial \mathcal{L}}{\partial w^{[1]}} \qquad w^{[2]} = w^{[2]} - \alpha \frac{\partial \mathcal{L}}{\partial w^{[2]}}$$

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. nature, 323(6088):533–536, 1986.

Gradient of Cost / Gradient of the Loss

- **For one training example i:**
  - Forward propagation:
    - $\hat{y}^{(i)} = a^{(i)} = \sigma(z^{(i)}) = \sigma(\mathbf{x}^{\mathbf{T}^{(i)}}\mathbf{w} + b)$
  - Computing the Loss:
    - $\mathscr{L}_\theta(\hat{y}^{(i)}, y^{(i)})$

- **For all training examples**
  - Computing the Cost (sum of the Loss $\mathscr{L}$ over all training examples m)

$$J_\theta = \frac{1}{m}\sum_{i=1}^{m} \mathscr{L}_\theta(\hat{y}^{(i)}, y^{(i)})$$

- **Minimising the Cost**
  - We need to compute the gradient of the Cost w.r.t. the parameters θ
    - $\dfrac{\partial J_\theta}{\partial \theta_1} = \dfrac{\partial}{\partial \theta_1}\left(\dfrac{1}{m}\sum_{i=1}^{m} \mathscr{L}_\theta(\hat{y}^{(i)}, y^{(i)})\right) = \dfrac{1}{m}\sum_{i=1}^{m} \dfrac{\partial \mathscr{L}_\theta(\hat{y}^{(i)}, y^{(i)})}{\partial \theta_1}$
    - The gradient of the Cost is the average (over the m training examples) of the gradient of the Loss

Logistic Regression (0 hidden layers)

Parameters update

- At iteration t

$$w_1^{[t]} = w_1^{[t-1]} - \alpha \frac{\partial J}{\partial w_1}$$

$$w_2^{[t]} = w_2^{[t-1]} - \alpha \frac{\partial J}{\partial w_2}$$

$$b^{[t]} = b^{[t-1]} - \alpha \frac{\partial J}{\partial b}$$

where α is the learning rate

Example code in python

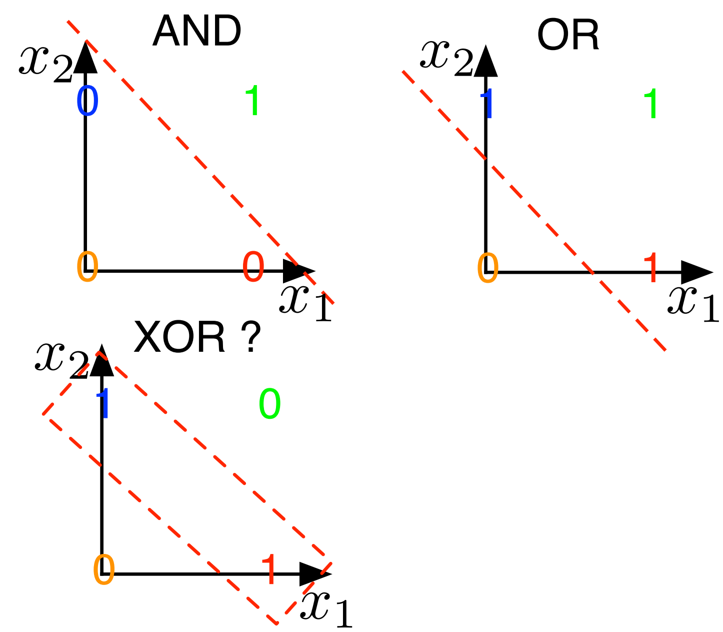
```
# --- initialisation
w1 =  np.random.randn(1)*0.01
w2 =  np.random.randn(1)*0.01
b = 0

# --- loop over epochs
for epoch in range(1000):
  J, dw1, dw2, db = 0, 0, 0, 0

  # --- loop over training examples
  for i in range(m):
     # --- forward
     z[i] = w1 * x[i,0] + w2 * x[i,1] + b
     a[i] = sigmoid(z[i])
     # --- cost
     J += -( y[i] * np.log(a[i]) + (1-y[i]) * np.log(1-a[i]) )
     # --- backward
     dz[i] = a[i] - y[i]
     dw1 += x[i,0] * dz[i]
     dw2 += x[i,1] * dz[i]
     db += dz[i]

  J /= m
  dw1 /= m
  dw2 /= m
  db /= m
  # --- parameters update
  w1 = w1 - alpha * dw1
  w2 = w2 - alpha * dw2
  b = b - alpha * dw2
```

- One **Epoch** $\Rightarrow$ one pass over all (m) training data
- **Batch Gradient descent** $\Rightarrow$ the gradient is estimated using all (m) training data

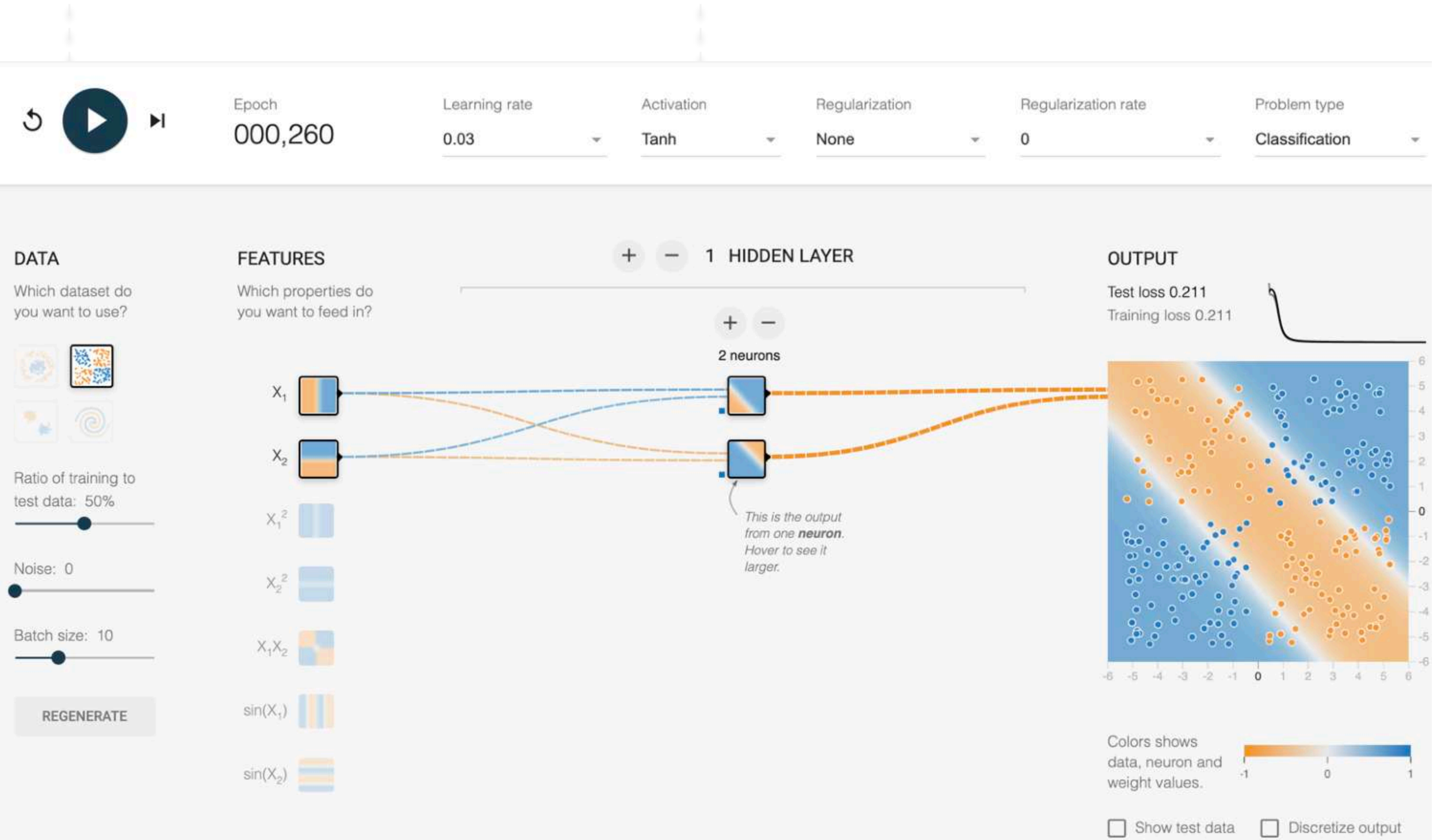
Logistic Regression (0 hidden layers)

Limitation of linear classifiers

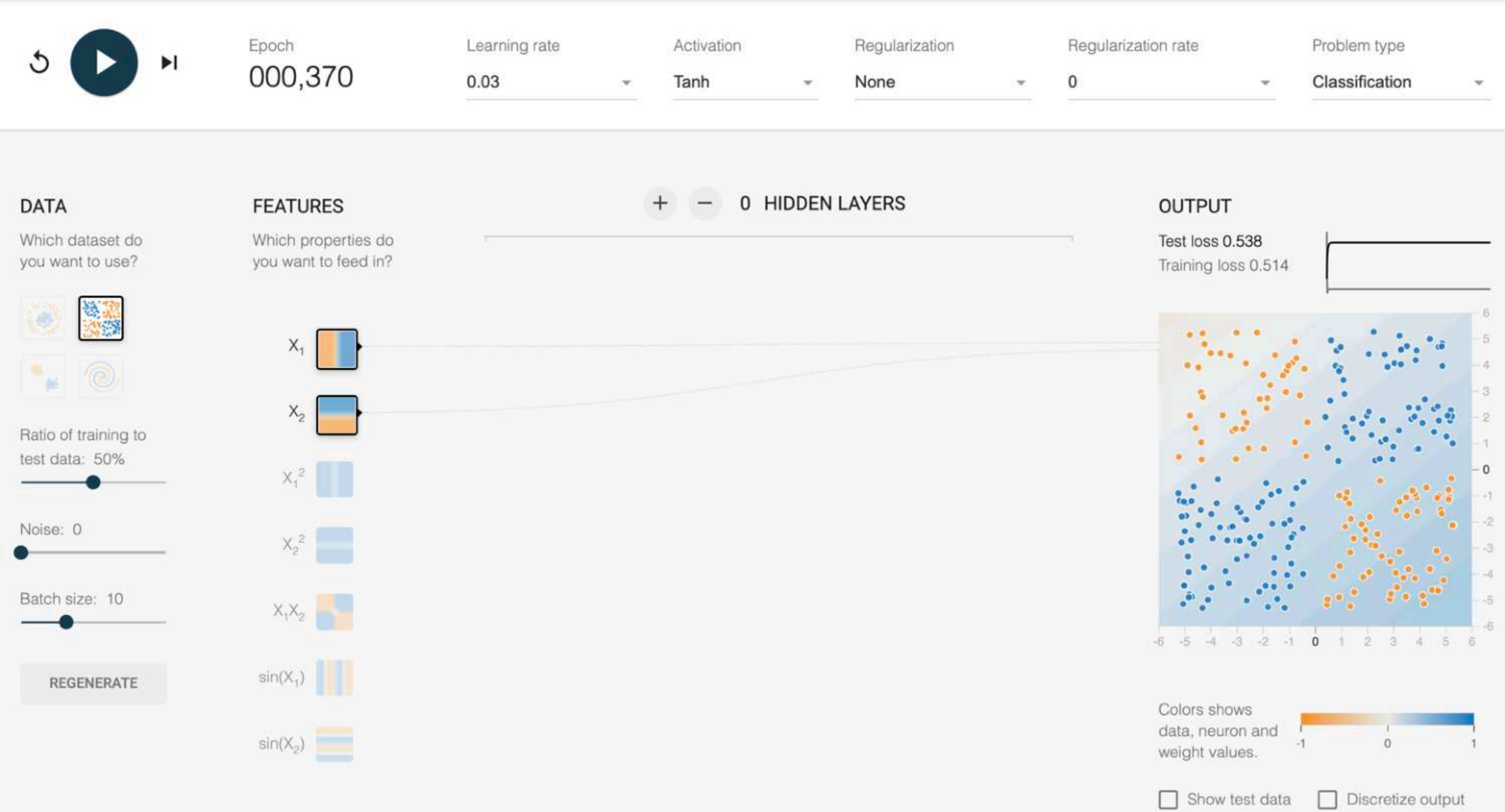
- Perceptron and Logistic Regression are linear classifiers
  - can model AND, OR operators
- What if classes are not linearly separable ?
  - cannot model XOR operator



Logistic Regression (0 hidden layers)
illustration in https://playground.tensorflow.org/



Logistic Regression (0 hidden layers) illustration in https://playground.tensorflow.org/



Logistic Regression (0 hidden layers)

Limitation of linear classifiers

– **Solution to the XOR problem**:

- project the input data $\mathbf{x}$ in a new space $\mathbf{x}'$

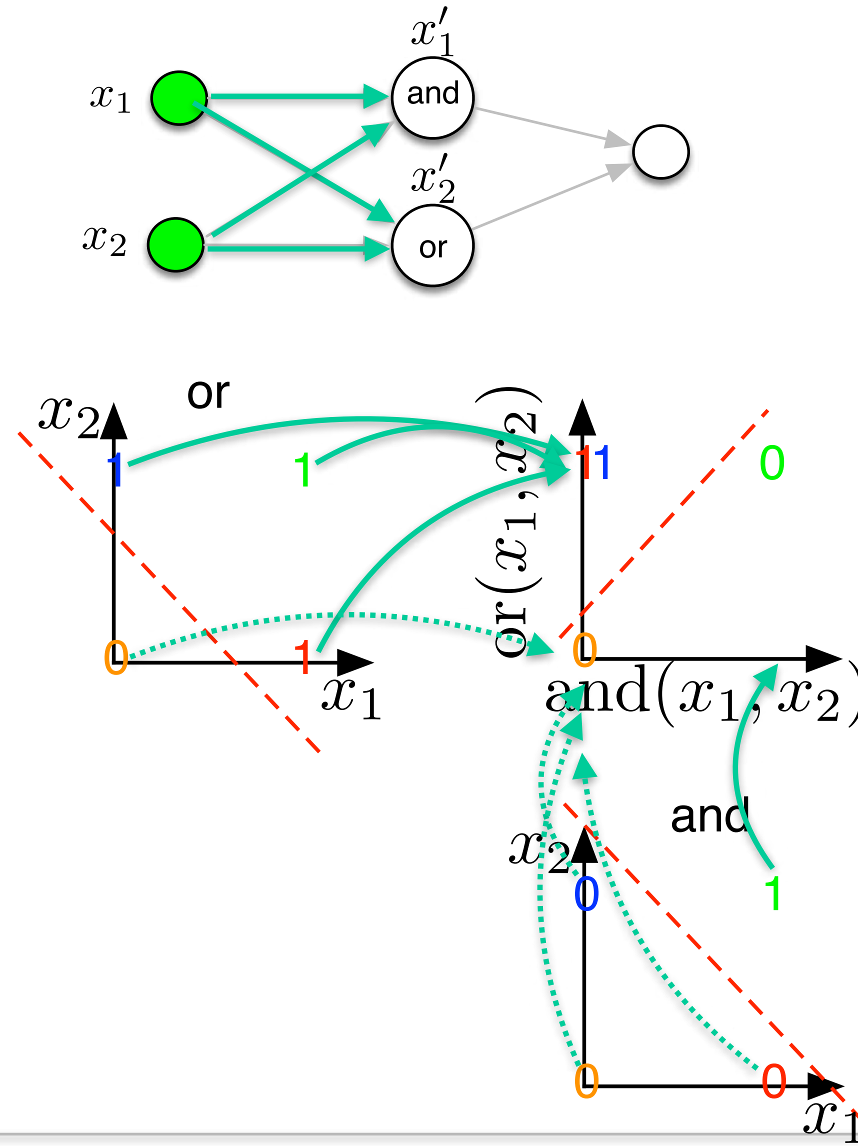
$$x_1^{[1]} = \textbf{AND}(x_1^{[0]}, x_2^{[0]})$$

$$- \; x_2^{[1]} = \textbf{OR}(x_1^{[0]}, x_2^{[0]})$$

– We therefore need one hidden layer of projection

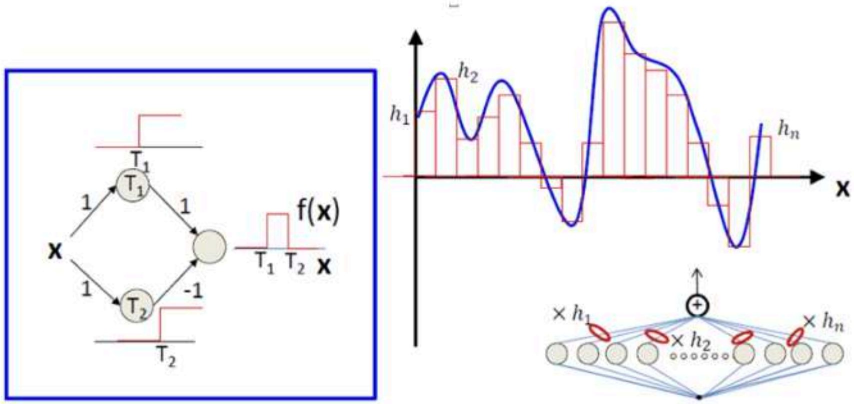
- ⇒ this is a 2 layers **Neural Network** ⇒1 hidden projection

– In a Neural Network, f_1 and f_2 and trainable projections; they can be many more of such projections



Neural Network as Universal Function Approximator

- A Neural Network with only one hidden layer can **approximate any function** (with enough hidden neurons)

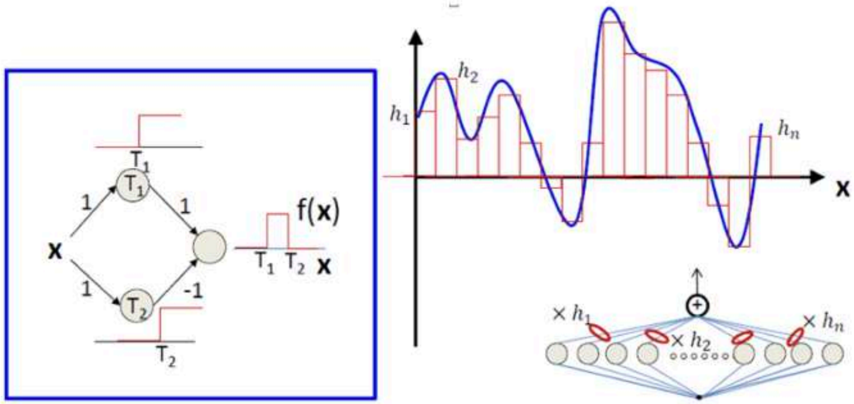


3 unit Multi Layer Perceptron using Step function to approximate a continuous function

https://medium.com/analytics-vidhya/neural-networks-and-the-universal-approximation-theorem-e5c387982eed

Neural Network as Universal Function Approximator

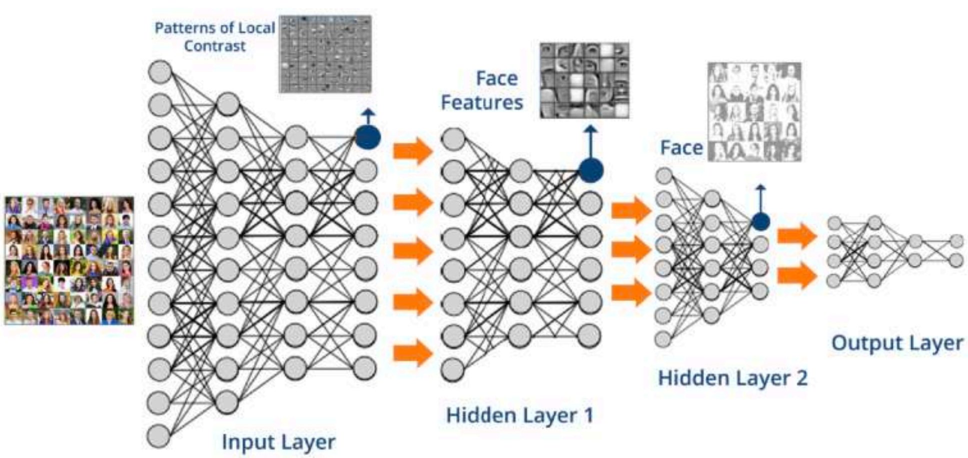
- A Neural Network with only one hidden layer can **approximate any function** (with enough hidden neurons)



3 unit Multi Layer Perceptron using Step function to approximate a continuous function

https://medium.com/analytics-vidhya/neural-networks-and-the-universal-approximation-theorem-e5c387982eed

- Deep Learning is about making the approximation of the function **progressively** (with several hidden layers of fewer neurons)



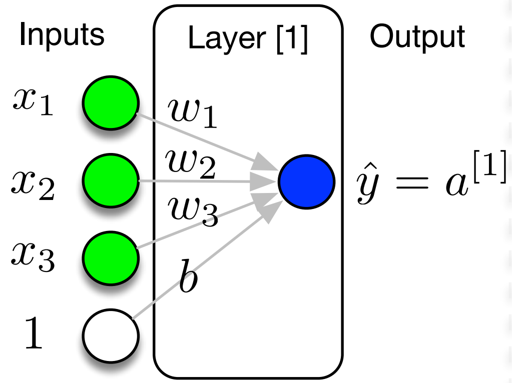
https://thedatascientist.com/what-deep-learning-is-and-isnt/

Neural Networks (1 hidden layer)

Neural Networks (1 hidden layer)

- **Logistic Regression (**1 layer, **0 hidden layer)**

$$\underline{x} \rightarrow \boxed{w_1 x_1 + w_2 x_2 + b} \xrightarrow{z} \boxed{\sigma(z)} \xrightarrow{a} \boxed{\mathcal{L}(\hat{y} = a, y)}$$

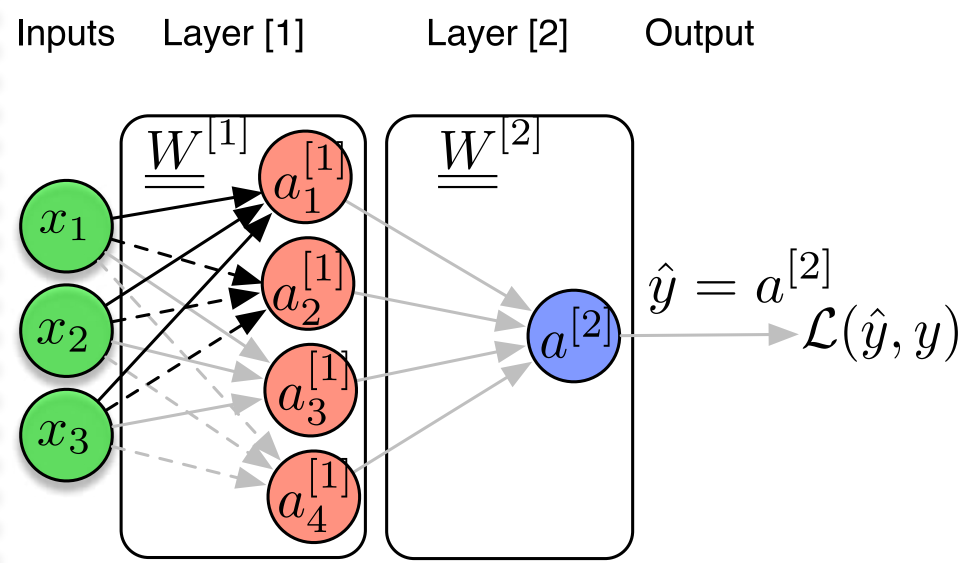


Neural Networks (1 hidden layer)

- **Neural Network (**2 layers, **1 hidden layer)**

$$\underline{x} \xrightarrow{a^{[0]}} \underbrace{\boxed{\underline{x}\,\underline{\underline{W}}^{[1]} + \underline{b}^{[1]}} \xrightarrow{\underline{z}^{[1]}} \boxed{g^{[1]}(\underline{z}^{[1]})}}_{Layer[1]} \xrightarrow{\underline{a}^{[1]}} \underbrace{\boxed{\underline{a}^{[1]}\,\underline{\underline{W}}^{[2]} + \underline{b}^{[2]}} \xrightarrow{z^{[2]}} \boxed{g^{[2]}(z^{[2]})}}_{Layer[2]} \xrightarrow{a^{[2]}} \boxed{\mathcal{L}(\hat{y} = a^{[2]}, y)}$$

- **Notations**
  - $\mathbf{a}^{[l]}$
    - activations at layer $[l]$
  - $\mathbf{n}^{[l]}$
    - number of neurons of layer $[l]$
  - $\mathbf{a}^{[0]} = \mathbf{x}$
    - input vector
  - $\hat{\mathbf{y}} = \mathbf{a}^{[L]}$
    - output vector
  - $\mathbf{W}^{[l]}$
    - weight matrix connecting layer $[l-1]$ to layer $[l]$
  - $\mathbf{b}^{[l]}$
    - bias of layer $[l]$



$a^{[0]} = x$ $\quad a^{[1]}$ $\quad a^{[2]}$

$n^{[0]} = 3$ $\quad n^{[1]} = 4$ $\quad n^{[2]} = 1$

$$\underline{x} \xrightarrow{a^{[0]}} \underbrace{\boxed{\underline{x}\,\underline{\underline{W}}^{[1]} + \underline{b}^{[1]}} \xrightarrow{z^{[1]}} \boxed{g^{[1]}(\underline{z}^{[1]})}}_{Layer[1]} \xrightarrow{a^{[1]}} \underbrace{\boxed{\underline{a}^{[1]}\underline{\underline{W}}^{[2]} + \underline{b}^{[2]}} \xrightarrow{z^{[2]}} \boxed{g^{[2]}(z^{[2]})}}_{Layer[2]} \xrightarrow{a^{[2]}} \boxed{\mathcal{L}(\hat{y} = a^{[2]}, y)}$$

- **Parameters to update**
  - $\theta = \{ \underset{(n^{[0]},n^{[1]})}{\mathbf{W}^{[1]}}, \underset{(1,n^{[1]})}{\mathbf{b}^{[1]}}, \underset{(n^{[1]},n^{[2]})}{\mathbf{W}^{[2]}}, \underset{(1,n^{[2]})}{\mathbf{b}^{[2]}} \}$

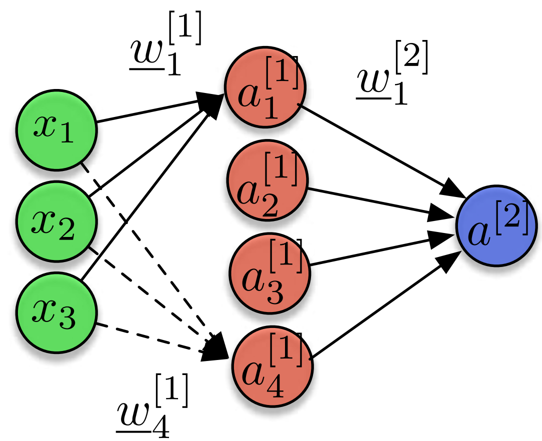
- **Gradient descent:**
  - **Initialise** the parameters θ
  - Repeat for # iterations
    - Repeat for all training examples $\forall i \in \{1,\ldots,m\}$
      - **Forward computation:** compute the prediction $\hat{y}^{(i)}$
      - **Compute the Loss** $\mathscr{L}_\theta(\hat{y}^{(i)}, y^{(i)})$
      - **Backward propagation:** compute the gradients: $\dfrac{\partial \mathscr{L}_\theta(\ldots)}{\partial \mathbf{W}^{[1]}}, \dfrac{\partial \mathscr{L}_\theta(\ldots)}{\partial \mathbf{b}^{[1]}}, \dfrac{\partial \mathscr{L}_\theta(\ldots)}{\partial \mathbf{W}^{[2]}}, \dfrac{\partial \mathscr{L}_\theta(\ldots)}{\partial \mathbf{b}^{[2]}}$
    - Compute the Cost J_θ
    - **Update the parameters** θ using the learning rate α

Neural Networks (1 hidden layer)

$$\underline{x} \xrightarrow{a^{[0]}} \underbrace{\boxed{\underline{x}\ \underline{\underline{W}}^{[1]} + \underline{b}^{[1]}} \xrightarrow{z^{[1]}} \boxed{g^{[1]}(\underline{z}^{[1]})}}_{Layer[1]} \xrightarrow{a^{[1]}} \underbrace{\boxed{\underline{a}^{[1]}\underline{\underline{W}}^{[2]} + \underline{b}^{[2]}} \xrightarrow{z^{[2]}} \boxed{g^{[2]}(z^{[2]})}}_{Layer[2]} \xrightarrow{a^{[2]}} \boxed{\mathcal{L}(\hat{y} = a^{[2]}, y)}$$

Version 1 (each dimension d, each training examples i) ⇒ Forward

- for i=1 to m
  - Input
    - $\mathbf{a}^{[0](i)} = \mathbf{x}^{(i)}$
  - Output
    - $\hat{y}^{(i)} = \mathbf{a}^{[2](i)}$



- Layer 1
  - for d=1 to $n^{[1]}$

$$z_d^{[1](i)} = \mathbf{a}^{[0](i)^T}\mathbf{w}_d^{[1]} + b_d^{[1]}$$

$$a_d^{[1](i)} = g^{[1]}(z_d^{[1](i)})$$

- Layer 2
  - for d'=1 to $n^{[2]}$

$$z_{d'}^{[2](i)} = \mathbf{a}^{[1](i)^T}\mathbf{w}_{d'}^{[2]} + b_{d'}^{[2]}$$

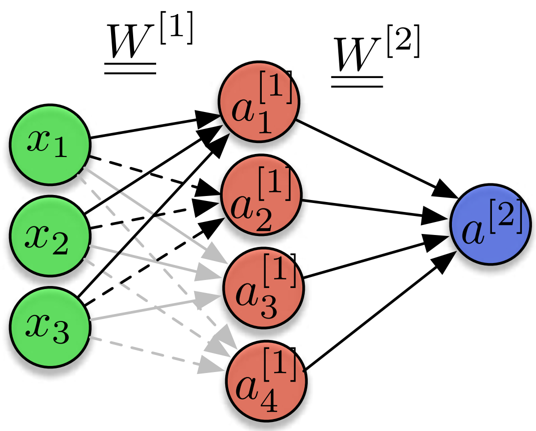
$$a_{d'}^{[2](i)} = g^{[2]}(z_{d'}^{[2](i)})$$

Neural Networks (1 hidden layer)

$$x \xrightarrow{a^{[0]}} \underbrace{\boxed{x\, W^{[1]} + b^{[1]}} \xrightarrow{z^{[1]}} \boxed{g^{[1]}(z^{[1]})}}_{Layer[1]} \xrightarrow{a^{[1]}} \underbrace{\boxed{a^{[1]} W^{[2]} + b^{[2]}} \xrightarrow{z^{[2]}} \boxed{g^{[2]}(z^{[2]})}}_{Layer[2]} \xrightarrow{a^{[2]}} \boxed{\mathcal{L}(\hat{y} = a^{[2]}, y)}$$

Version 2 (all dimensions d, each training examples i) $\Rightarrow$ Forward

- for i=1 to m
  - Input
    - $\mathbf{a}^{[0](i)} = \mathbf{x}^{(i)}$
  - Output
    - $\hat{y}^{(i)} = \mathbf{a}^{[2](i)}$



- Layer 1

$$\underset{(n^{[1]},1)^T}{\mathbf{z}^{[1](i)^T}} = \underset{(n^{[0]},1)^T}{\mathbf{a}^{[0](i)^T}} \underset{(n^{[0]},n^{[1]})}{\mathbf{W}^{[1]}} + \underset{(n^{[1]})}{\mathbf{b}^{[1]}}$$

$$\underset{(n^{[1]},1)^T}{\mathbf{a}^{[1](i)^T}} = g^{[1]}(\underset{(n^{[1]},1)^T}{\mathbf{z}^{[1](i)^T}})$$

- Layer 2

$$\underset{(n^{[2]},1)^T}{\mathbf{z}^{[2](i)^T}} = \underset{(n^{[1]},1)^T}{\mathbf{a}^{[1](i)^{T}T}} \underset{(n^{[1]},n^{[2]})}{\mathbf{W}^{[2]}} + \underset{(n^{[2]})}{\mathbf{b}^{[2]}}$$

$$\underset{(n^{[2]},1)^T}{\mathbf{a}^{[2](i)^T}} = g^{[2]}(\underset{(n^{[2]},1)^T}{\mathbf{z}^{[1](i)^T}})$$

Neural Networks (1 hidden layer)

$$\underline{x} \xrightarrow{\underline{a}^{[0]}} \underbrace{\boxed{\underline{x}\ \underline{\underline{W}}^{[1]} + \underline{b}^{[1]}} \xrightarrow{\underline{z}^{[1]}} \boxed{\underline{g}^{[1]}(\underline{z}^{[1]})}}_{Layer[1]} \xrightarrow{\underline{a}^{[1]}} \underbrace{\boxed{\underline{a}^{[1]}\ \underline{\underline{W}}^{[2]} + \underline{b}^{[2]}} \xrightarrow{\underline{z}^{[2]}} \boxed{\underline{g}^{[2]}(\underline{z}^{[2]})}}_{Layer[2]} \xrightarrow{\underline{a}^{[2]}} \boxed{\mathcal{L}(\hat{y} = \underline{a}^{[2]}, y)}$$

Version 2 (all dimensions d, each training examples i) $\Rightarrow$ Backward

- $\dfrac{\partial \mathscr{L}}{\partial a^{[2]}}$ = derivative of the loss $= -\left(\dfrac{y}{a^{[2]}} + \dfrac{(1-y)}{(1-a^{[2]})}\right) = \dfrac{a^{[2]} - y}{a^{[2]}(1-a^{[2]})}$
- **Layer 2** (input $\dfrac{\partial \mathscr{L}}{\partial a^{[2]}}$)

$$\frac{\partial \mathscr{L}}{\partial z^{[2]}} = \frac{\partial \mathscr{L}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} = \frac{\partial \mathscr{L}}{\partial a^{[2]}} \odot g^{[2]'}(\mathbf{z}^{[2]})$$

$$= \frac{\partial \mathscr{L}}{\partial a^{[2]}} \odot a^{[2]}(1-a^{[2]}) = a^{[2]} - y$$

$$\underset{(n^{[1]},n^{[2]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[2]}}} = \frac{\partial \mathscr{L}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial \mathbf{W}^{[2]}} = \underset{(n^{[1]},1)}{\mathbf{a}^{[1]}} \underset{(1,n^{[2]})}{\frac{\partial \mathscr{L}}{\partial z^{[2]}}}$$

$$\frac{\partial \mathscr{L}}{\partial \mathbf{b}^{[2]}} = \frac{\partial \mathscr{L}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial b^{[2]}} = \frac{\partial \mathscr{L}}{\partial z^{[2]}}$$

$$\frac{\partial \mathscr{L}}{\partial \mathbf{a}^{[1]}} = \frac{\partial \mathscr{L}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} = \underset{(1,n^{[2]})}{\frac{\partial \mathscr{L}}{\partial z^{[2]}}} \underset{(n^{[1]},n^{[2]})^T}{W^{[2]^T}}$$

Neural Networks (1 hidden layer)

$$\underline{x} \xrightarrow{\underline{a}^{[0]}} \underbrace{\boxed{\underline{x}\,\underline{\underline{W}}^{[1]} + \underline{b}^{[1]}} \xrightarrow{\underline{z}^{[1]}} \boxed{\underline{g}^{[1]}(\underline{z}^{[1]})}}_{Layer[1]} \xrightarrow{\underline{a}^{[1]}} \underbrace{\boxed{\underline{a}^{[1]}\,\underline{\underline{W}}^{[2]} + \underline{b}^{[2]}} \xrightarrow{\underline{z}^{[2]}} \boxed{\underline{g}^{[2]}(\underline{z}^{[2]})}}_{Layer[2]} \xrightarrow{\underline{a}^{[2]}} \boxed{\mathcal{L}(\hat{y} = \underline{a}^{[2]}, y)}$$

Version 2 (all dimensions d, each training examples i) ⇒ Backward

- $\frac{\partial \mathscr{L}}{\partial a^{[2]}}$ = derivative of the loss $= -\left(\frac{y}{a^{[2]}} + \frac{(1-y)}{(1-a^{[2]})}\right) = \frac{a^{[2]}-y}{a^{[2]}(1-a^{[2]})}$

- **Layer 2** (input $\frac{\partial \mathscr{L}}{\partial a^{[2]}}$)

$$\frac{\partial \mathscr{L}}{\partial z^{[2]}} = \frac{\partial \mathscr{L}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}} = \frac{\partial \mathscr{L}}{\partial a^{[2]}} \odot g^{[2]'}(\mathbf{z}^{[2]})$$

$$= \frac{\partial \mathscr{L}}{\partial a^{[2]}} \odot a^{[2]}(1-a^{[2]}) = a^{[2]} - y$$

$$\underset{(n^{[1]},n^{[2]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[2]}}} = \frac{\partial \mathscr{L}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial \mathbf{W}^{[2]}} = \underset{(n^{[1]},1)}{\mathbf{a}^{[1]}}\ \underset{(1,n^{[2]})}{\frac{\partial \mathscr{L}}{\partial z^{[2]}}}$$

$$\frac{\partial \mathscr{L}}{\partial \mathbf{b}^{[2]}} = \frac{\partial \mathscr{L}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial b^{[2]}} = \frac{\partial \mathscr{L}}{\partial z^{[2]}}$$

— — —

$$\boxed{\frac{\partial \mathscr{L}}{\partial \mathbf{a}^{[1]}}} = \frac{\partial \mathscr{L}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial a^{[1]}} = \underset{(1,n^{[2]})}{\frac{\partial \mathscr{L}}{\partial z^{[2]}}}\ \underset{(n^{[1]},n^{[2]})^T}{W^{[2]^T}}$$

- **Layer 1** (input $\boxed{\frac{\partial \mathscr{L}}{\partial a^{[1]}}}$)

$$\boxed{\frac{\partial \mathscr{L}}{\partial \mathbf{z}^{[1]}}} = \frac{\partial \mathscr{L}}{\partial \mathbf{a}^{[1]}}\mathbf{a}^{[1]}\mathbf{z}^{[1]} = \boxed{\frac{\partial \mathscr{L}}{\partial \mathbf{a}^{[1]}}} g^{[1]'}(\mathbf{z}^{[1]})$$

$$\boxed{\underset{(n^{[0]},n^{[1]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[1]}}}} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}^{[1]}}\frac{\partial \mathbf{z}^{[1]}}{\partial \mathbf{W}^{[1]}} = \underset{(n^{[0]},1)}{\mathbf{a}^{[0]}}\ \boxed{\underset{(1,n^{[1]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{z}^{[1]}}}}$$

$$\boxed{\frac{\partial \mathscr{L}}{\partial \mathbf{b}^{[1]}}} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}^{[1]}}\frac{\partial \mathbf{z}^{[1]}}{\partial \mathbf{b}^{[1]}} = \boxed{\frac{\partial \mathscr{L}}{\partial \mathbf{z}^{[1]}}}$$

Neural Networks (1 hidden layer)

$$\underline{x} \xrightarrow{a^{[0]}} \underbrace{\boxed{\underline{x}\,\underline{\underline{W}}^{[1]} + \underline{b}^{[1]}} \xrightarrow{z^{[1]}} \boxed{g^{[1]}(\underline{z}^{[1]})}}_{Layer[1]} \xrightarrow{a^{[1]}} \underbrace{\boxed{\underline{a}^{[1]}\,\underline{\underline{W}}^{[2]} + \underline{b}^{[2]}} \xrightarrow{z^{[2]}} \boxed{g^{[2]}(z^{[2]})}}_{Layer[2]} \xrightarrow{a^{[2]}} \boxed{\mathcal{L}(\hat{y} = a^{[2]}, y)}$$

Version 3 (all dimensions d, all training examples D) ⇒ Forward

- Input
  - $\mathbf{A}^{[0]} = \underset{(m,n^{[0]})}{\mathbf{X}}$
- Output
  - $\hat{Y} = \mathbf{A}^{[2]}$

sklearn order of dimensions: (n_samples, n_features)

- Layer 1

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

$$\underset{(m,n^{[1]})}{\mathbf{A}^{[1]}} = g^{[1]}\big(\underset{(m,n^{[1]})}{\mathbf{Z}^{[1]}}\big)$$

- Layer 2

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

$$\underset{(m,n^{[2]})}{\mathbf{A}^{[2]}} = g^{[2]}\big(\underset{(m,n^{[2]})}{\mathbf{Z}^{[2]}}\big)$$

$$\underline{x} \xrightarrow{a^{[0]}} \underbrace{\boxed{\underline{x}\,\underline{\underline{W}}^{[1]} + \underline{b}^{[1]}} \xrightarrow{z^{[1]}} \boxed{g^{[1]}(\underline{z}^{[1]})}}_{Layer[1]} \xrightarrow{a^{[1]}} \underbrace{\boxed{\underline{a}^{[1]}\,\underline{\underline{W}}^{[2]} + \underline{b}^{[2]}} \xrightarrow{z^{[2]}} \boxed{g^{[2]}(z^{[2]})}}_{Layer[2]} \xrightarrow{a^{[2]}} \boxed{\mathcal{L}(\hat{y} = a^{[2]}, y)}$$

Version 3 (all dimensions d, all training examples D) $\Rightarrow$ Backward

– **Layer 2**

$$\underset{(m,n^{[2]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{Z}^{[2]}}} = \underset{(m,n^{[2]})}{\mathbf{A}^{[2]}} - \underset{(m,n^{[2]})}{\mathbf{Y}}$$

$$\underset{(n^{[1]},n^{[2]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[2]}}} = \frac{1}{m} \underset{(m,n^{[1]})^T}{\mathbf{A}^{[1]^T}} \underset{(m,n^{[2]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{Z}^{[2]}}}$$

$$\underset{(1,n^{[2]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{b}^{[2]}}} = \frac{1}{m} \sum_{i=1}^{m} \underset{(m,n^{[2]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{Z}^{[2]}}}$$

– – –

$$\underset{(m,n^{[1]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{A}^{[1]}}} = \underset{(m,n^{[2]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{Z}^{[2]}}} \underset{(n^{[1]},n^{[2]})^T}{\mathbf{W}^{[2]^T}}$$

– **Layer 1**

$$\underset{(m,n^{[1]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{Z}^{[1]}}} = \underset{(m,n^{[1]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{A}^{[1]}}} \odot g^{[1]'}(\underset{(m,n^{[1]})}{\mathbf{Z}^{[1]}})$$

$$\underset{(n^{[0]},n^{[1]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[1]}}} = \frac{1}{m} \underset{(m,n^{[0]})^T}{\mathbf{A}^{[0]^T}} \underset{(m,n^{[1]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{Z}^{[1]}}}$$

$$\underset{(1,n^{[1]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{b}^{[1]}}} = \frac{1}{m} \sum_{i=1}^{m} \underset{(m,n^{[1]})}{\frac{\partial \mathscr{L}}{\partial \mathbf{Z}^{[1]}}}$$

Neural Networks (1 hidden layer)

$$\underline{x} \xrightarrow{\underline{a}^{[0]}} \underbrace{\boxed{\underline{x}\,\underline{\underline{W}}^{[1]} + \underline{b}^{[1]}} \xrightarrow{\underline{z}^{[1]}} \boxed{g^{[1]}(\underline{z}^{[1]})}}_{Layer[1]} \xrightarrow{\underline{a}^{[1]}} \underbrace{\boxed{\underline{a}^{[1]}\,\underline{\underline{W}}^{[2]} + \underline{b}^{[2]}} \xrightarrow{z^{[2]}} \boxed{g^{[2]}(z^{[2]})}}_{Layer[2]} \xrightarrow{a^{[2]}} \boxed{\mathcal{L}(\hat{y} = a^{[2]}, y)}$$

Parameters update

$$\mathbf{W}^{[l]} = \mathbf{W}^{[l]} - \alpha \frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[l]}}$$

$$\mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \alpha \frac{\partial \mathscr{L}}{\partial \mathbf{b}^{[l]}}$$

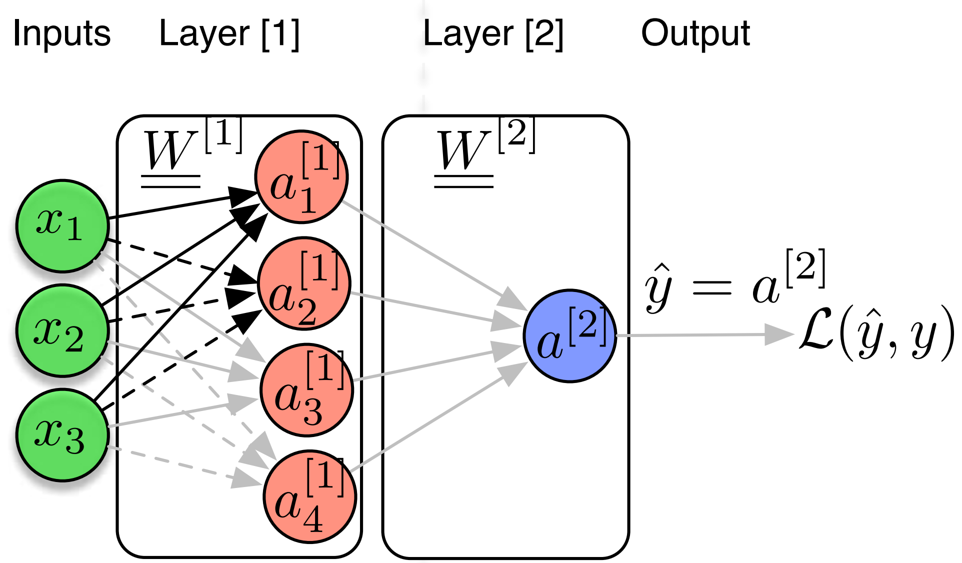
– where α is the learning rate

Deep Neural Networks (> 2 hidden layer)

Deep Neural Networks (> 2 hidden layer)

- **Neural Network (**2 layers, **1 hidden layer)**

$$\underline{x} \xrightarrow{a^{[0]}} \underbrace{\boxed{\underline{x}\,\underline{\underline{W}}^{[1]} + \underline{b}^{[1]}} \xrightarrow{\underline{z}^{[1]}} \boxed{g^{[1]}(\underline{z}^{[1]})}}_{Layer[1]} \xrightarrow{\underline{a}^{[1]}} \underbrace{\boxed{\underline{a}^{[1]}\,\underline{\underline{W}}^{[2]} + \underline{b}^{[2]}} \xrightarrow{z^{[2]}} \boxed{g^{[2]}(z^{[2]})}}_{Layer[2]} \xrightarrow{a^{[2]}} \boxed{\mathcal{L}(\hat{y} = a^{[2]}, y)}$$

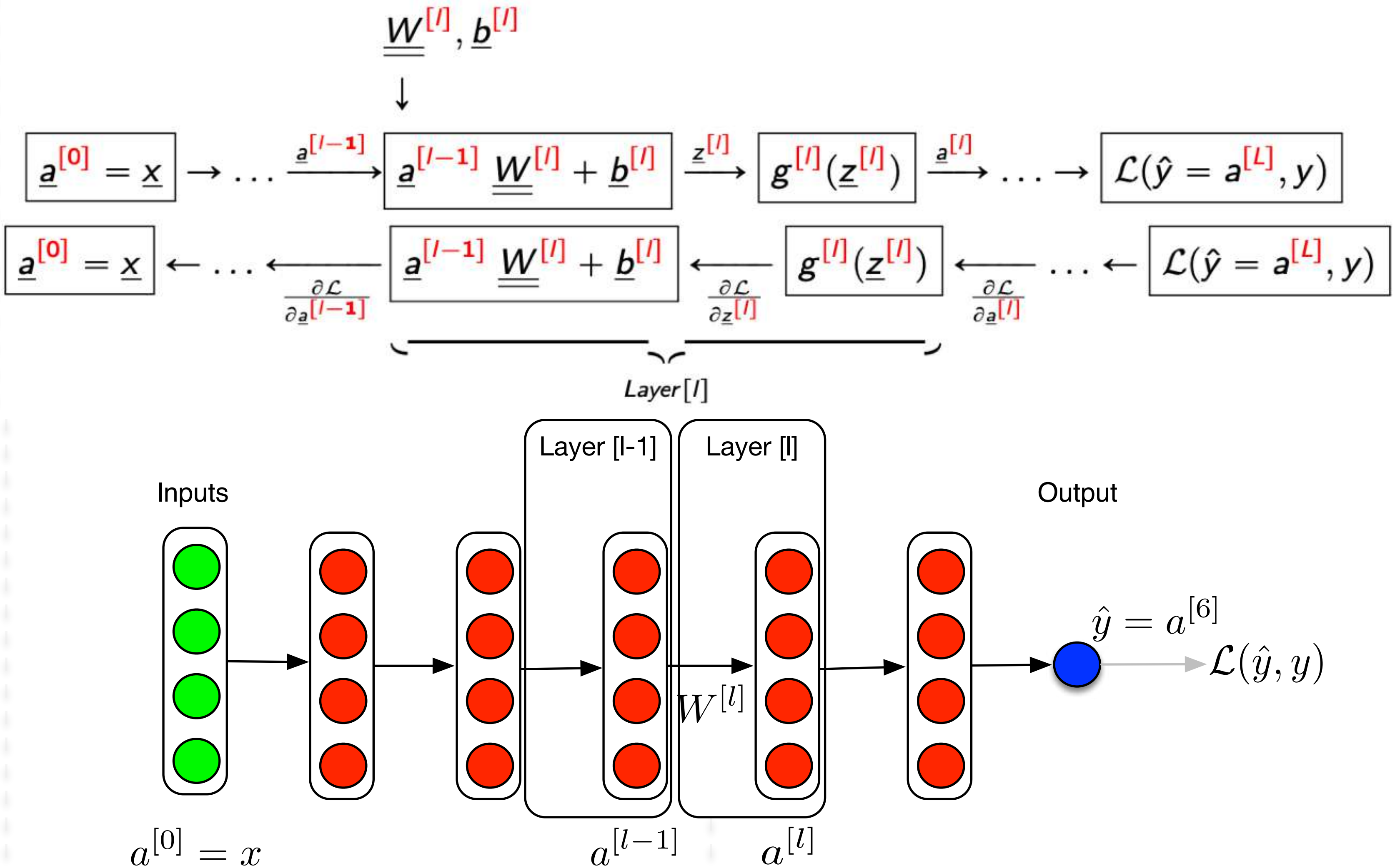


$a^{[0]} = x$ $\quad a^{[1]}$ $\quad a^{[2]}$

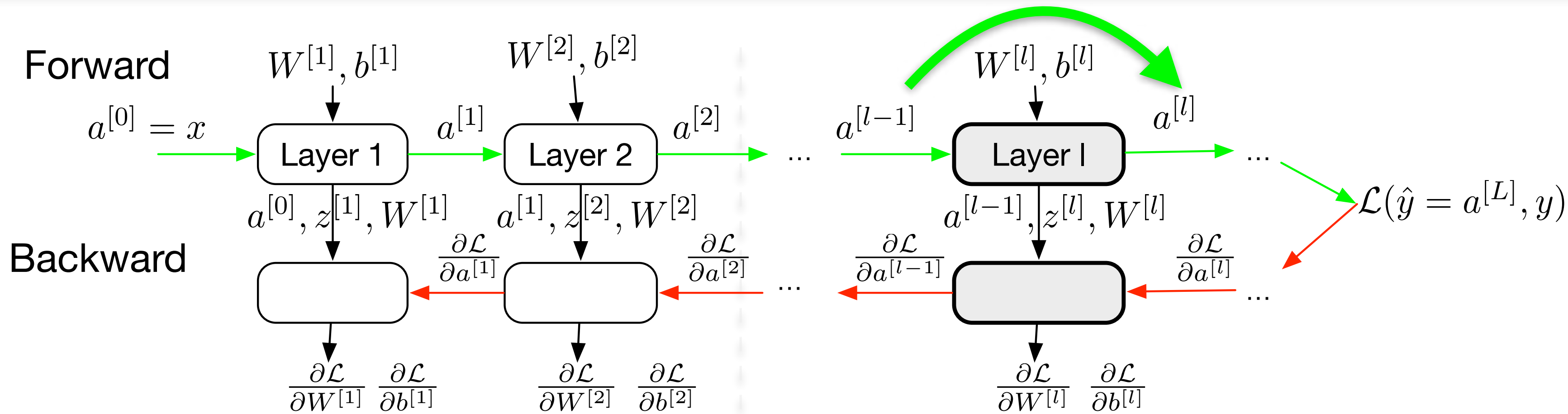
$n^{[0]} = 3$ $\quad n^{[1]} = 4$ $\quad n^{[2]} = 1$

Deep Neural Networks (> 2 hidden layer)

- **Deep Neural Network (**many layers, **> 2 hidden layer)**



Deep Neural Networks (> 2 hidden layer)



Forward (general formulation for layer l, all training examples)

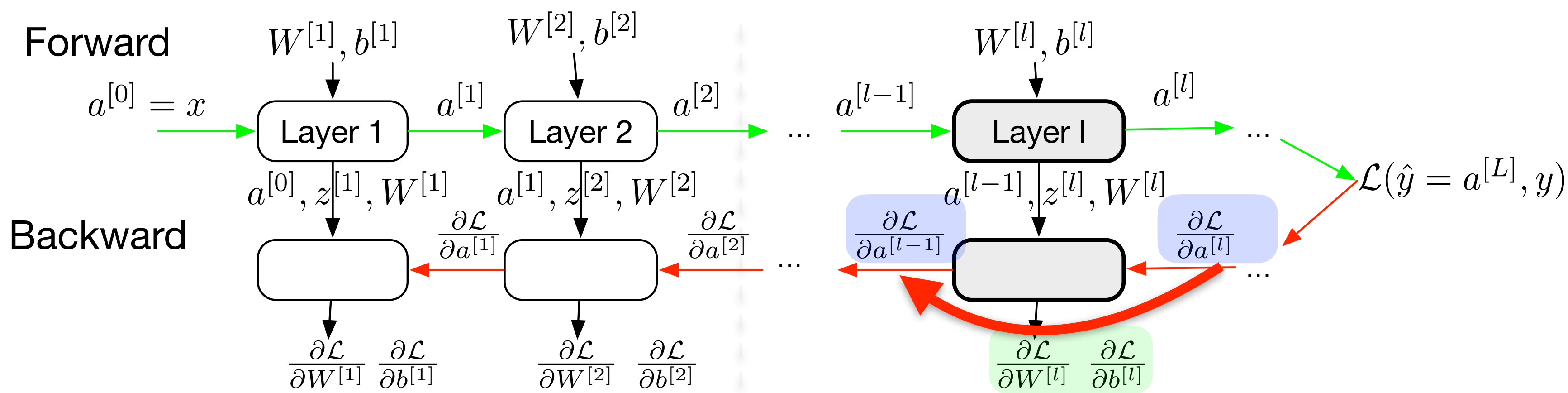
- **Input**: $\mathbf{A}^{[l-1]}$

$$\underset{(m,n^{[l]})}{\mathbf{Z}^{[l]}} = \underset{(m,n^{[l-1]})}{\mathbf{A}^{[l-1]}} \underset{(n^{[l-1]},n^{[l]})}{\mathbf{W}^{[l]}} + \underset{(1,n^{[l]})}{\mathbf{b}^{[l]}}$$

$$\underset{(m,n^{[l]})}{\mathbf{A}^{[l]}} = g^{[l]}(\mathbf{Z}^{[l]})$$

- **Output**: $\mathbf{A}^{[l]}$
- Storage for back-propagation: $\mathbf{A}^{[l-1]}, \mathbf{Z}^{[l]}, \mathbf{W}^{[l]}$

Deep Neural Networks (> 2 hidden layer)



Backward (general formulation for layer l, all training examples)

- Input: $\dfrac{\partial \mathscr{L}}{\partial \mathbf{A}^{[l]}}$ Input from storage: $\mathbf{A}^{[l-1]}, \mathbf{Z}^{[l]}, \mathbf{W}^{[l]}$

$$\frac{\partial \mathscr{L}}{\partial \mathbf{Z}^{[l]}} = \frac{\partial \mathscr{L}}{\partial \mathbf{A}^{[l]}} g^{[l]'}(\mathbf{Z}^{[l]}) = \left(\frac{\partial \mathscr{L}}{\partial \mathbf{Z}^{[l+1]}} \mathbf{W}^{[l+1]^T} \right) \odot g^{[l]'}(\mathbf{Z}^{[l]})$$

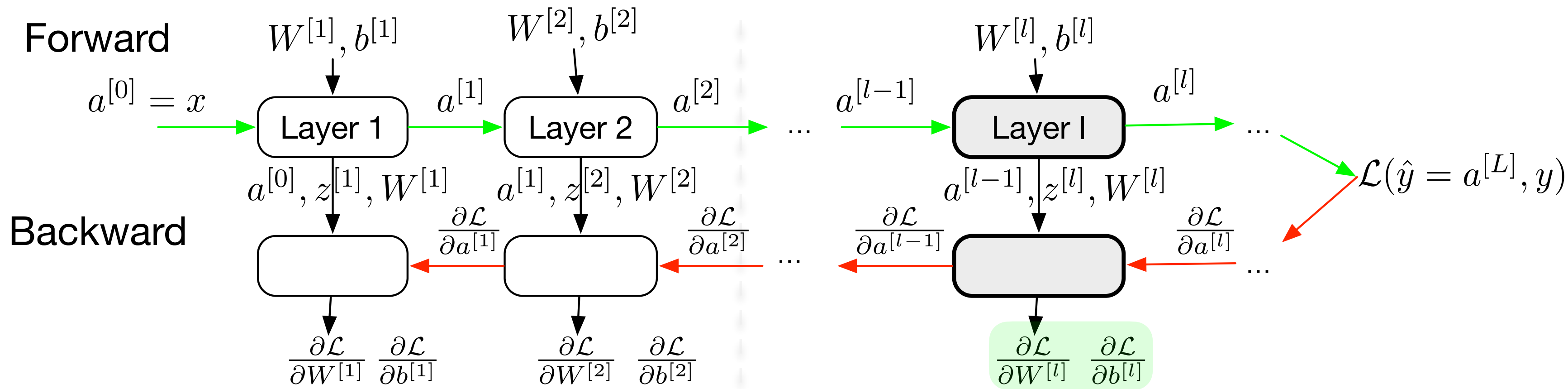
$$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[l]}} = \frac{1}{m} \mathbf{A}^{[l-1]^T} \frac{\partial \mathscr{L}}{\partial \mathbf{Z}^{[l]}}$$

$$\frac{\partial \mathscr{L}}{\partial \mathbf{b}^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathscr{L}}{\partial \mathbf{Z}^{[l]}}$$

$$\frac{\partial \mathscr{L}}{\partial \mathbf{A}^{[l-1]}} = \frac{\partial \mathscr{L}}{\partial \mathbf{Z}^{[l]}} \mathbf{W}^{[l]^T}$$

- Output for back-propagation: $\dfrac{\partial \mathscr{L}}{\partial \mathbf{A}^{[l-1]}}$ Output parameters update: $\dfrac{\partial \mathscr{L}}{\partial \mathbf{W}^{[l]}}, \dfrac{\partial \mathscr{L}}{\partial \mathbf{b}^{[l]}}$

Deep Neural Networks (> 2 hidden layer)



Parameters update

$$\mathbf{W}^{[l]} = \mathbf{W}^{[l]} - \alpha \frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[l]}}$$

$$\mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \alpha \frac{\partial \mathscr{L}}{\partial \mathbf{b}^{[l]}}$$

– where α is the learning rate

Activation functions

Why non-linear activation functions ?

– Consider the following network

$$\mathbf{z}^{[1]} = \mathbf{x}\ \mathbf{W}^{[1]} + \mathbf{b}^{[1]}$$
$$\mathbf{a}^{[1]} = g^{[1]}(\mathbf{z}^{[1]})$$
$$\mathbf{z}^{[2]} = \mathbf{a}^{[1]}\mathbf{W}^{[2]} + \mathbf{b}^{[2]}$$
$$\mathbf{a}^{[2]} = g^{[2]}(\mathbf{z}^{[2]})$$

– If $g^{[1]}$ and $g^{[2]}$ are **linear activation functions** (identity function)

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

– then the network the network reduces to a simple linear function

– Linear activation ? only interesting for the last layer $g^{[L]}$ of regression problem:$y \in \mathbb{R}$

Activation functions

Sigmoid σ .

- **Sigmoid function**

$$a = g(z) = \sigma(z) = \boxed{\frac{1}{1+e^{-z}}}$$

- **Derivative:**

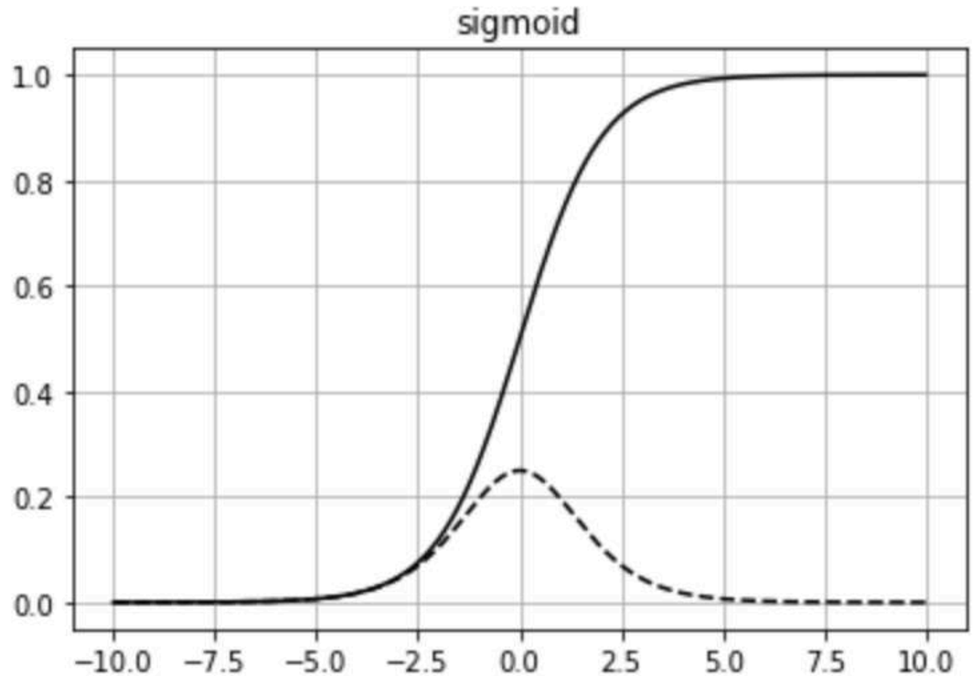
$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$
$$g'(z) = a(1-a)$$

- Proof:

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

- **Other properties:**

$$\sigma(-z) = 1 - \sigma(z)$$



Activation functions

Hyperbolic tangent function

- **Sigmoid function**

$$a = g(z) = \tanh(z) = \boxed{\frac{e^z - e^{-z}}{e^z + e^{-z}}}$$

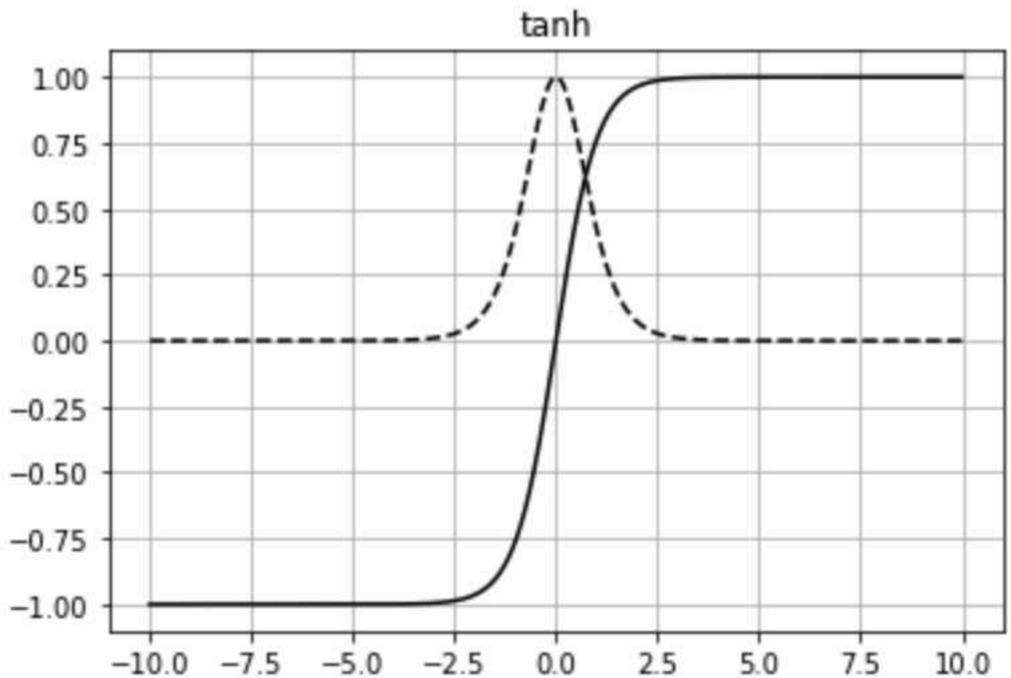
- **Derivative**

$$g'(x) = 1 - \left(\tanh(z)\right)^2$$

$$g'(z) = 1 - a^2$$

- **Other properties:**

$$\tanh(z) = 2\sigma(2z) - 1$$



- **Usage**
  - $\tanh(z)$ better than $\sigma(z)$ in middle hidden layers because its mean = zero ($a \in [-1,1]$).

- **Problem** with σ and $\tanh$:
  - if z is very small (negative) or very large (positive)
  - $\Rightarrow$ slope becomes zero
  - $\Rightarrow$ slow down Gradient Descent

Activation functions

Vanishing gradient

- Reminder:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{[l]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{[l+1]}} \underbrace{\mathbf{W}^{[l+1]^T} \odot g^{[l]'}(\mathbf{Z}^{[l]})}$$

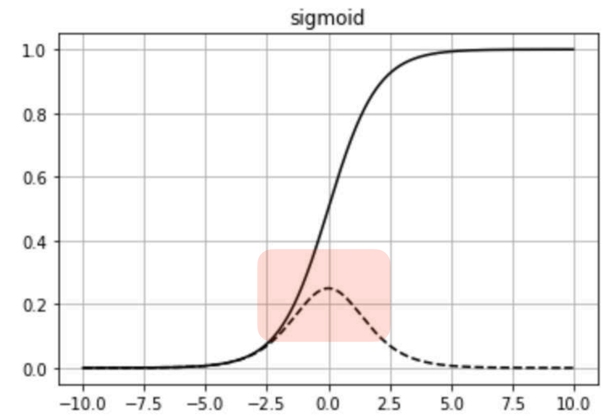
$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}} = \frac{1}{m} \sum_m \frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{[l]}}$$

- Hence for a deep network (supposing $g^{[l]}(z) = \sigma(z)$)

$$\frac{\partial \mathcal{L}}{\partial b^{[1]}} = \frac{\partial \mathcal{L}}{\partial a^{[4]}} \sigma'(z^{[4]}) \underbrace{W^{[4]}\sigma'(z^{[3]})} \; \underbrace{W^{[3]}\sigma'(z^{[2]})} \; \underbrace{W^{[2]}\sigma'(z^{[1]})}$$

- **Problem:** $\max_z \sigma'(z) = \frac{1}{4}!$
  - Therefore, the deeper the network, the fastest the gradient diminishes/vanishes during back-propagation
  - Consequence ? **The network stop learning !**

- **Solution ?**
  - Use a ReLu activation



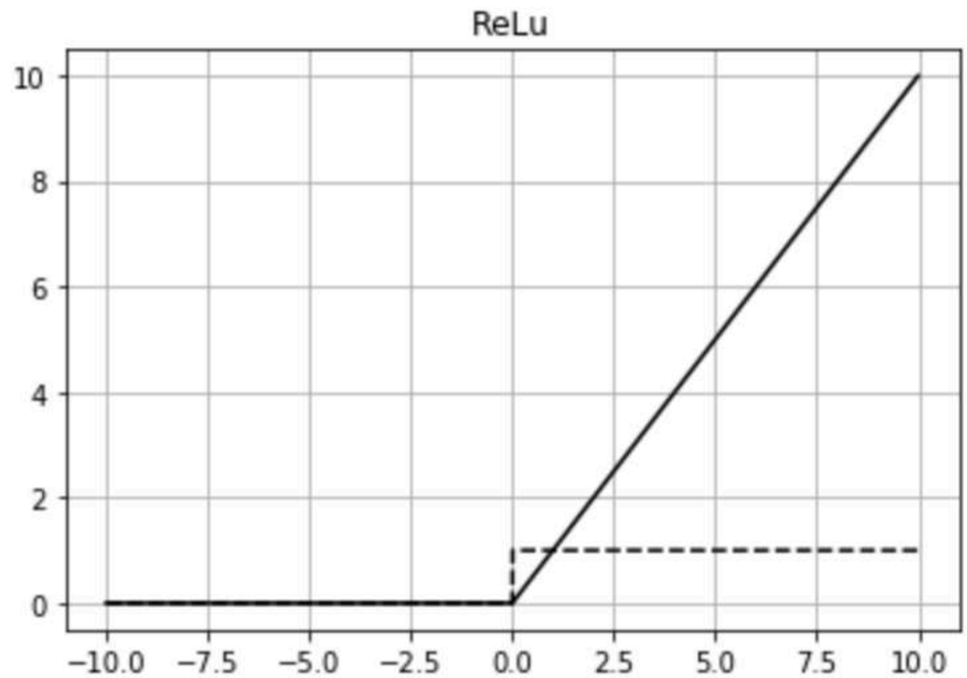
ReLU (Rectified Linear Units)

- **ReLU function**

$$a = g(z) = \max(0,z)$$

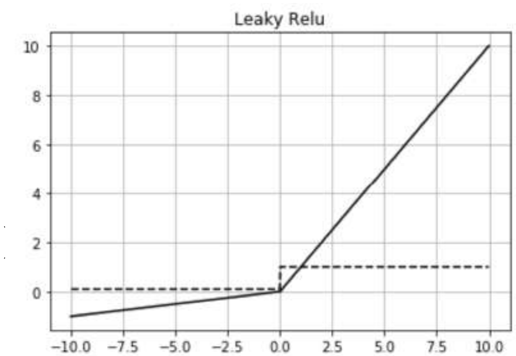
- **Derivative**

$$g'(x) = 1 \quad \text{if } z > 0$$
$$= 0 \quad \text{if } z \leq 0$$



Activation functions

Variations of ReLU



- **Leaky ReLU function**
  - $a = g(x) = \max(0.01z, z)$
  - allows to avoid the zero slope of the ReLU for $z < 0$ ("the neuron dies")
  - **Derivative**

$$g'(x) = 1 \quad \textbf{if } z > 0$$
$$= 0.01 \quad \textbf{if } z \leq 0$$

- **PReLU function**
  - $a = g(x) = \max(\alpha z, z)$
  - same as Leaky ReLU but α is a parameter to be learnt
  - **Derivative**

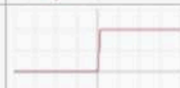
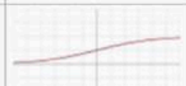
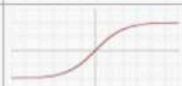
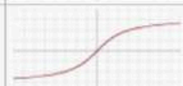
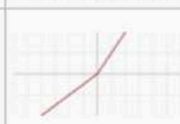
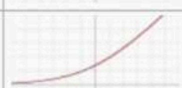
$$g'(x) = 1 \quad \textbf{if } z > 0$$
$$= \alpha \quad \textbf{if } z \leq 0$$

- **Softplus function**
  - $g(x) = \log\left(1 + e^x\right)$
  - **continuous approximation of ReLU**
  - **Derivative**

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

  - the derivative of the Softplus function is the Logistic function (smooth approximation of the derivative of the rectifier, the Heaviside step function.)

List of possible activation functions

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Various types of problems

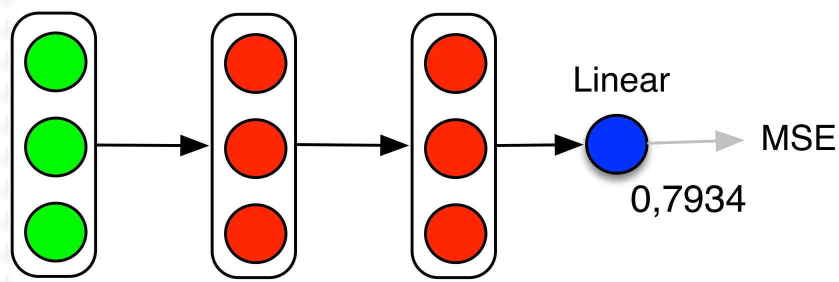
Regression

Model

- We (can) have several output neurons $\hat{y}_c$
- output value are **not mutually exclusive**
  - $\Rightarrow$ each neuron has a **linear (ReLU)** activation function,
  - $\Rightarrow$ for each neuron we minimise the **mean-square-error**
- We minimise the sum of the MSE

$$\mathscr{L} = - \sum_c (y_c - \hat{y}_c)^2$$

Regression



Mean Square Error

- The ground-truth output y is a continuous variable $\in \mathbb{R}$
  - $y^{(i)}$ is gaussian distributed with mean $\hat{y}^{(i)}$
  - $y^{(i)} \sim \mathcal{N}(\hat{y}^{(i)}, \sigma^2) = \hat{y}^{(i)} + \mathcal{N}(0,\sigma^2)$
- We want to find the θ
  - ... that maximise the **likelihood** of the $y^{(i)}$ given the $x^{(i)}$

$$p(y\,|\,X,\theta,\sigma) = \prod_{i=1}^{n} p(y^{(i)}\,|\,x^{(i)},\theta,\sigma)$$

$$= \prod_{i=1}^{n} \frac{1}{(2\pi\sigma)^{1/2}} e^{-\frac{1}{2\sigma^2}(y^{(i)}-\hat{y}^{(i)})^2}$$

$$= \frac{1}{(2\pi\sigma)^{n/2}} e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y^{(i)}-\hat{y}^{(i)})^2}$$

  - ... that minimise the Mean Square Error - MSE (minimise the cost)
    - $$J(\theta) = \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2$$

Multi-label classification

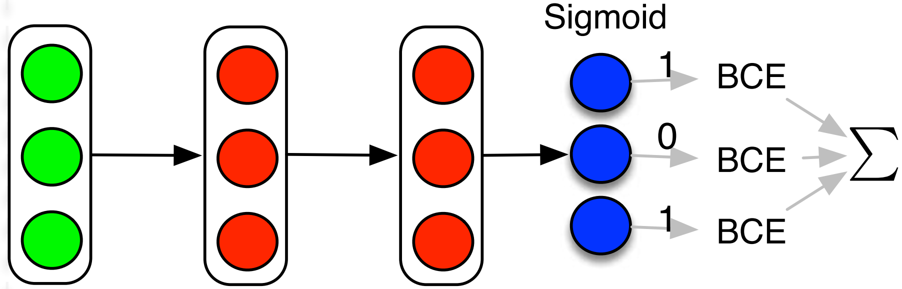
Multi-label classification

Model

- We have several output neurons $\hat{y}_o$
- output label are **not mutually exclusive**
  - $\Rightarrow$ each neuron has a **sigmoid** activation function,
  - $\Rightarrow$ for each neuron we minimise the **binary cross-entropy**
- We minimise the sum of the BCEs

$$\mathscr{L} = -\sum_o \left(y_o \log(\hat{y}_o) + (1 - y_o)\log(1 - \hat{y}_o) \right)$$

Multi-label



Multi-label classification

Loss

- **Loss**

$$\mathscr{L} - \sum_o \left(y_o \log(\hat{y}_o) + (1 - y_o)\log(1 - \hat{y}_o) \right)$$

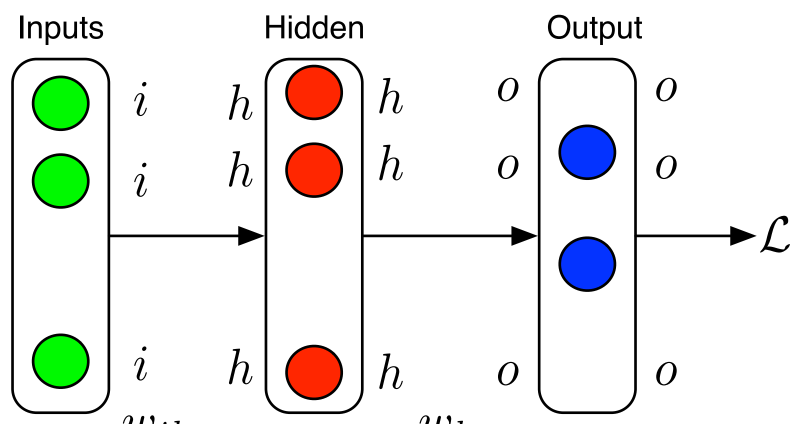
- **Forward**

$$z_h^{[1]} = \sum_i w_{ih}^{[1]} x_i$$

$$a_h^{[1]} = \frac{1}{1 + e^{-z_h^{[1]}}}$$

$$z_o^{[2]} = \sum_h w_{ho}^{[2]} a_h^{[1]}$$

$$\hat{y}_o = \frac{1}{1 + e^{-z_o^{[2]}}}$$



$$x_i \xrightarrow{w_{ih}} z_h^{[1]} \rightarrow a_h^{[1]} \xrightarrow{w_{ho}} z_o^{[2]} \rightarrow \hat{y}_o = a_o^{[2]}$$

Backward

$$\frac{\partial \mathscr{L}}{\partial w_{ho}^{[2]}} = \frac{\partial \mathscr{L}}{\partial \hat{y}_o} \frac{\partial \hat{y}_o}{\partial z_o^{[2]}} \frac{\partial z_o^{[2]}}{\partial w_{ho}^{[2]}}$$

$$d(Loss) \quad \frac{\partial \mathscr{L}}{\partial \hat{y}_o} = \left(-\frac{y_o}{\hat{y}_o} + \frac{1-y_o}{1-\hat{y}_o} \right) = \frac{\hat{y}_o - y_o}{\hat{y}_o(1-\hat{y}_o)}$$

$$d(Sigmoid) \quad \frac{\partial \hat{y}_o}{\partial z_o^{[2]}} = \hat{y}_o(1-\hat{y}_o)$$

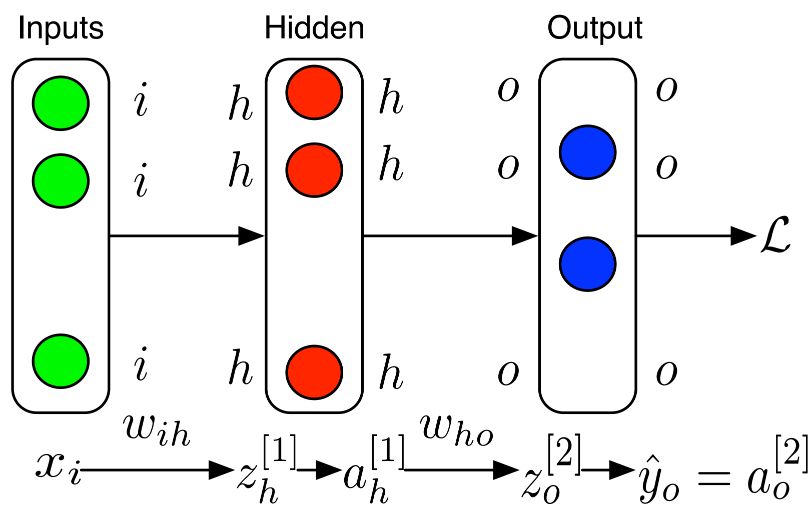
$$\frac{\partial \mathscr{L}}{\partial z_o^{[2]}} = \hat{y}_o - y_o$$

$$\frac{\partial z_o^{[2]}}{\partial w_{ho}^{[2]}} = a_h^{[1]}$$

$$\frac{\partial \mathscr{L}}{\partial w_{ho}^{[2]}} = (\hat{y}_o - y_o) a_h^{[1]}$$

$$\frac{\partial \mathscr{L}}{\partial w_{ih}^{[1]}} = \sum_o \frac{\partial \mathscr{L}}{\partial z_o^{[2]}} \frac{\partial z_o^{[2]}}{\partial a_h^{[1]}} \frac{\partial a_h^{[1]}}{\partial z_h^{[1]}} \frac{\partial z_h^{[1]}}{\partial w_{ih}^{[1]}}$$

$$= \sum_o (\hat{y}_o - y_o) w_{ho}^{[2]} a_h^{[1]} (1 - a_h^{[1]}) x_i$$



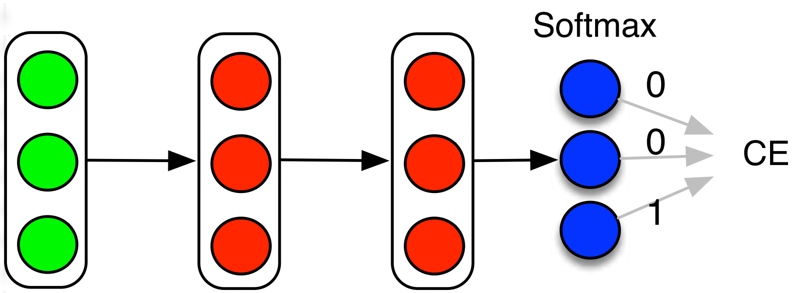
Multi-class classification

Model

- We have several output neurons $\hat{y}_c$
- output label are not **mutually exclusive**
  - $\Rightarrow$ each neuron has a **softmax** activation function,
- We minimise the **cross-entropy**

$$\mathscr{L} - \sum_{c=1}^{K} \left(y_c \log(\hat{y}_c)\right)$$

Multi-class



Multi-class classification

Output activation function: softmax (1)

- **Usage**: multi-class classification ($1 \cdots K$)
  - (softmax regression or deep neural network with several mutually exclusive outputs)

$$P(y=1\,|\,\mathbf{x}) = \frac{e^{\mathbf{w}_1 \cdot \mathbf{x}}}{\sum_{c=1}^{K} e^{\mathbf{w}_c \cdot \mathbf{x}}}$$

$$\cdots$$

$$P(y=o\,|\,\mathbf{x}) = \boxed{\frac{e^{\mathbf{w}_o \cdot \mathbf{x}}}{\sum_{c=1}^{K} e^{\mathbf{w}_c \cdot \mathbf{x}}}}$$

$$\cdots$$

$$P(y=K\,|\,\mathbf{x}) = \frac{e^{\mathbf{w}_K \cdot \mathbf{x}}}{\sum_{c=1}^{K} e^{\mathbf{w}_c \cdot \mathbf{x}}}$$

  - **Has a "redundant" set of parameters !**
    - if we subtract some fixed vector ψ from each $\mathbf{w}_o$, we get the same results

$$P(y=o\,|\,\mathbf{x}) = \frac{e^{(\mathbf{w}_o-\psi)\cdot\mathbf{x}}}{\sum_{c=1}^{K} e^{(\mathbf{w}_c-\psi)\cdot\mathbf{x}}} = \frac{e^{\mathbf{w}_o\cdot\mathbf{x}} \;\; e^{-\psi\cdot\mathbf{x}}}{\sum_{c=1}^{K} e^{\mathbf{w}_c\cdot\mathbf{x}} \;\; e^{-\psi\cdot\mathbf{x}}} = \frac{e^{\mathbf{w}_o\cdot\mathbf{x}}}{\sum_{c=1}^{K} e^{\mathbf{w}_c\cdot\mathbf{x}}}$$

    - Common choice: $\psi = \mathbf{w}_K$ hence $e^{(\mathbf{w}_K-\psi)\cdot\mathbf{x}} = 1$

$$P(y=o\,|\,\mathbf{x}) = \frac{e^{\mathbf{w}_o\mathbf{x}}}{1+\sum_{c=1}^{K-1} e^{\mathbf{w}_c\mathbf{x}}} \qquad \forall o \in [1 \ldots K-1]$$

$$P(y=K\,|\,\mathbf{x}) = \frac{1}{1+\sum_{c=1}^{K-1} e^{\mathbf{w}_c\cdot\mathbf{x}}}$$

Output activation function: softmax (2)

- We of course have
  $$P(y=1\,|\,\mathbf{x}) + P(y=2\,|\,\mathbf{x}) + \ldots + P(y=K\,|\,\mathbf{x}) = 1$$

- **Models the log-odds** $\log \frac{p_c}{p_K}$ (posterior probability) using linear models of the inputs $\mathbf{x}$

$$\log\left(\frac{P(y=1\,|\,\mathbf{x})}{P(y=K\,|\,\mathbf{x})}\right) = \mathbf{w}_1\mathbf{x}$$
$$\ldots$$
$$\log\left(\frac{P(y=o\,|\,\mathbf{x})}{P(y=K\,|\,\mathbf{x})}\right) = \mathbf{w}_o\mathbf{x}$$
$$\ldots$$
$$\log\left(\frac{P(y=K-1\,|\,\mathbf{x})}{P(y=K\,|\,\mathbf{x})}\right) = \mathbf{w}_{K-1}\mathbf{x}$$

- If $K=2$ their is an equivalence with Logistic Regression (binary classification)
  - we choose $\psi = \mathbf{w}_1$

$$P(y=1\,|\,\mathbf{x}) = \frac{e^{\mathbf{w}_1\mathbf{x}}}{e^{\mathbf{w}_1\mathbf{x}} + e^{\mathbf{w}_2\mathbf{x}}} = \frac{e^{(\mathbf{w}_1-\psi)\mathbf{x}}}{e^{(\mathbf{w}_1-\psi)\mathbf{x}} + e^{(\mathbf{w}_2-\psi)\mathbf{x}}} = \frac{1}{1+e^{(\mathbf{w}_2-\mathbf{w}_1)\mathbf{x}}}$$
$$P(y=2\,|\,\mathbf{x}) = \frac{e^{\mathbf{w}_2\mathbf{x}}}{e^{\mathbf{w}_1\mathbf{x}} + e^{\mathbf{w}_2\mathbf{x}}} = \frac{e^{(\mathbf{w}_2-\psi)\mathbf{x}}}{e^{(\mathbf{w}_1-\psi)\mathbf{x}} + e^{(\mathbf{w}_2-\psi)\mathbf{x}}} = \frac{e^{(\mathbf{w}_2-\mathbf{w}1)\mathbf{x}}}{1+e^{(\mathbf{w}_2-\mathbf{w}1)\mathbf{x}}} = 1 - P(y=1\,|\,\mathbf{x})$$

Loss

- **Loss**

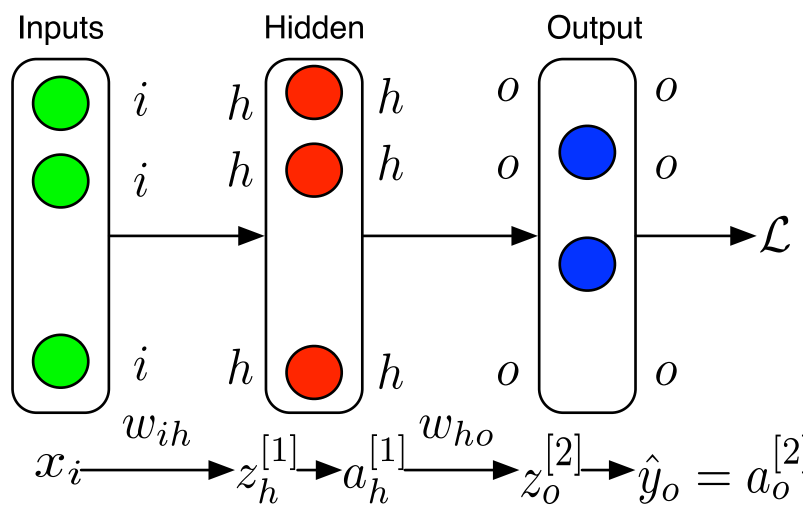
$$\mathscr{L} - \sum_{c=1}^{K} \left(y_c \log(\hat{y}_c) \right)$$

- **Forward**

$$\ldots$$

$$z_o^{[2]} = \sum_h w_{ho}^{[2]} a_h^{[1]}$$

$$\hat{y}_c = \frac{e^{z_c^{[2]}}}{\sum_o e^{z_o^{[2]}}}$$



Multi-class classification

Backward

$d(Loss)$ $\frac{\partial \mathscr{L}}{\partial \hat{y}_c} = -\frac{y_c}{\hat{y}_c}$

- reminder: $\hat{y}_c = \frac{e^{z_c^{[2]}}}{\sum_o e^{z_o^{[2]}}}$

$d(Softmax)$

$$\left(\frac{f}{g}\right)' = \frac{f'}{g} - \frac{fg'}{g^2}$$

if $c = o$ $\frac{\partial \hat{y}_c}{\partial z_o^{[2]}} = \frac{e^{z_c^{[2]}}}{\sum_o e^{z_o^{[2]}}} - \frac{e^{z_c^{[2]}} e^{z_c^{[2]}}}{\left(\sum_c e^{z_o^{[2]}}\right)^2} = \hat{y}_c(1 - \hat{y}_c)$

if $c \neq o$ $\frac{\partial \hat{y}_c}{\partial z_o^{[2]}} = 0 - \frac{e^{z_c^{[2]}} e^{z_o^{[2]}}}{(\sum_o e^{z_o^{[2]}})^2} = -\hat{y}_c \hat{y}_o$

$$
\begin{aligned}
\frac{\partial \mathscr{L}}{\partial z_o^{[2]}} &= \sum_c \frac{\partial \mathscr{L}}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial z_o^{[2]}} \\
&= \frac{\partial \mathscr{L}}{\partial \hat{y}_{c=o}} \frac{\partial \hat{y}_{c=o}}{\partial z_{c=o}^{[2]}} + \sum_{c \neq o} \frac{\partial \mathscr{L}}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial z_o^{[2]}} \\
&= -\frac{y_o}{\hat{y}_o} \hat{y}_o (1 - \hat{y}_o) + \sum_{c \neq o} \frac{y_c}{\hat{y}_c} \hat{y}_c \hat{y}_o \\
&= -y_o (1 - \hat{y}_o) + \sum_{c \neq o} y_c \hat{y}_o \\
&= -y_o + y_o \hat{y}_o + \sum_{c \neq o} y_c \hat{y}_o \\
&= -y_o + \hat{y}_o \sum_c y_c \\
&= -y_o + \hat{y}_o \cdot 1 \\
&= \hat{y}_o - y_o
\end{aligned}
$$

- $\Rightarrow$ same gradient $\frac{\partial \mathscr{L}}{\partial z_o^{[2]}}$ as for sigmoid binary

Computation Graph

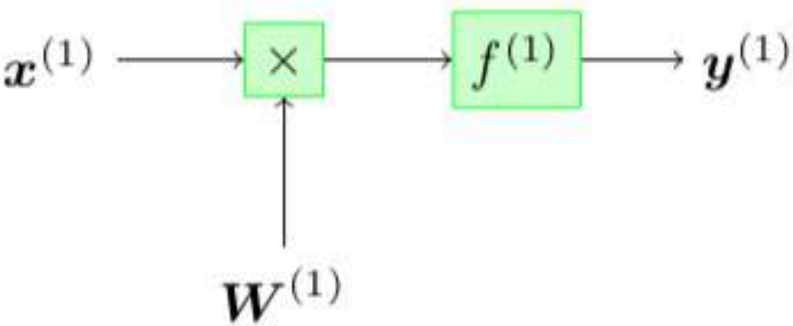
Computation Graph

(slide from from Alexandre Allauzen)

A convenient way to represent a complex mathematical expressions :

- each node is an operation or a variable
- an operation has some inputs / outputs made of variables

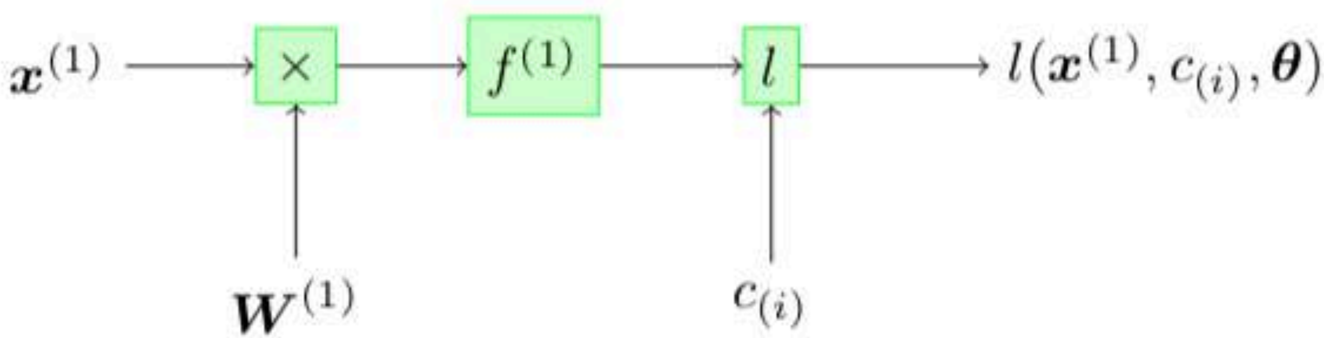
Example 1 : A single layer network



- Setting $\boldsymbol{x}^{(1)}$ and $\boldsymbol{W}^{(1)}$
- Forward pass $\rightarrow \boldsymbol{y}^{(1)}$

$$\boldsymbol{y}^{(1)} = f^{(1)}(\boldsymbol{W}^{(1)}\boldsymbol{x}^{(1)})$$

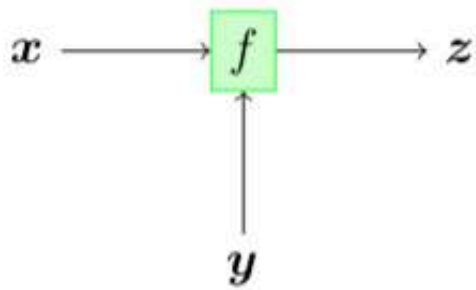
Computation Graph
(slide from from Alexandre Allauzen)



- A variable node encodes the label
- To compute the output for a given input
  - $\rightarrow$ forward pass
- To compute the gradient of the loss *wrt* the parameters ($\boldsymbol{W}^{(1)}$)
  - $\rightarrow$ backward pass

A function node

Forward pass

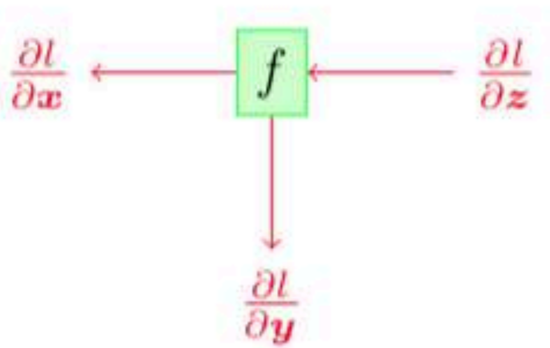


This node implements :

$$\boldsymbol{z} = f(\boldsymbol{x}, \boldsymbol{y})$$

A function node - 2

Backward pass



A function node knows :

- the "local gradients" computation

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}, \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}}$$

- how to return the gradient to the inputs :

$$\left(\frac{\partial l}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}\right), \left(\frac{\partial l}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}}\right)$$

Summary of a function node

$f:$

$\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ # store the values

$\boldsymbol{z} = f(\boldsymbol{x}, \boldsymbol{y})$ # forward

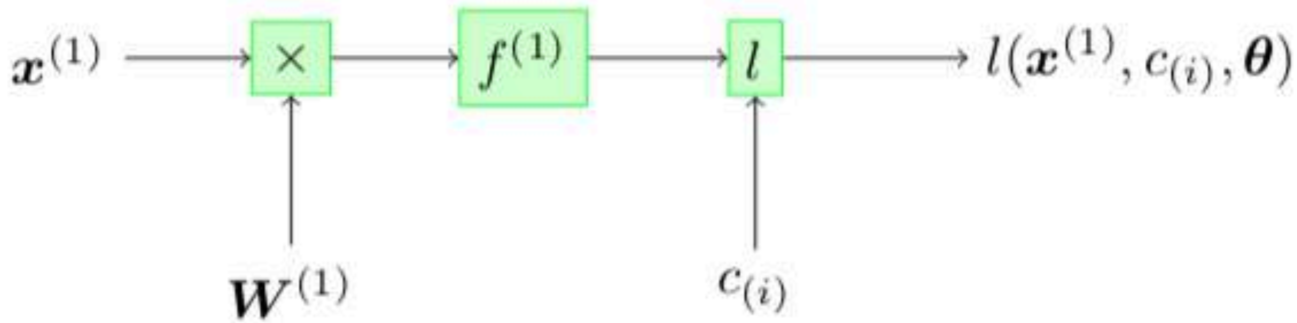
$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \rightarrow \frac{\partial f}{\partial \boldsymbol{x}}$ # local gradients

$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}} \rightarrow \frac{\partial f}{\partial \boldsymbol{y}}$

$(\frac{\partial l}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}), (\frac{\partial l}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}})$ # backward

Computation Graph
(slide from from Alexandre Allauzen)

Example of a single layer network



Forward

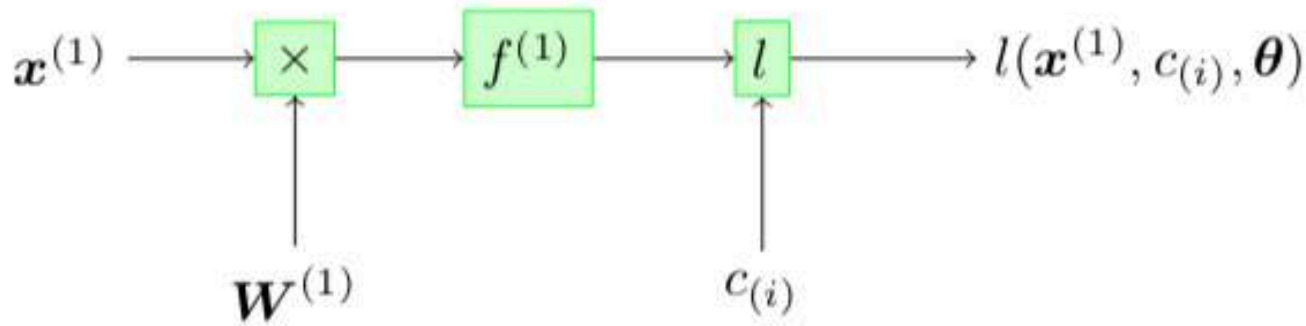
For each function node in topological order

- forward propagation

Which means :

1. $a^{(1)} = W^{(1)}x^{(1)}$
2. $y^{(1)} = f^{(1)}(a^{(1)})$
3. $l(y^{(1)}, c_{(i)})$

Example of a single layer network



Backward

For each function node in reversed topological order

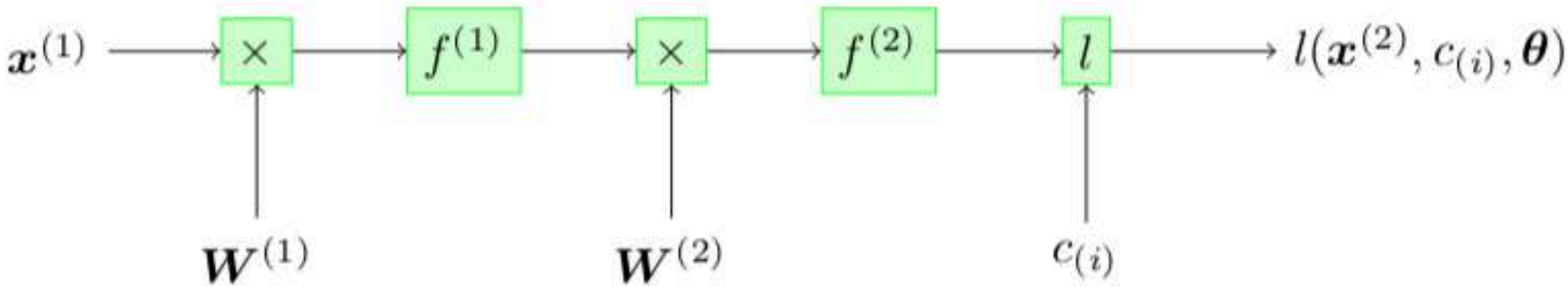
- backward propagation

Which means :

1. $\nabla_{y^{(1)}}$
2. $\nabla_{a^{(1)}}$
3. $\nabla_{W^{(1)}}$

Computation Graph
(slide from from Alexandre Allauzen)

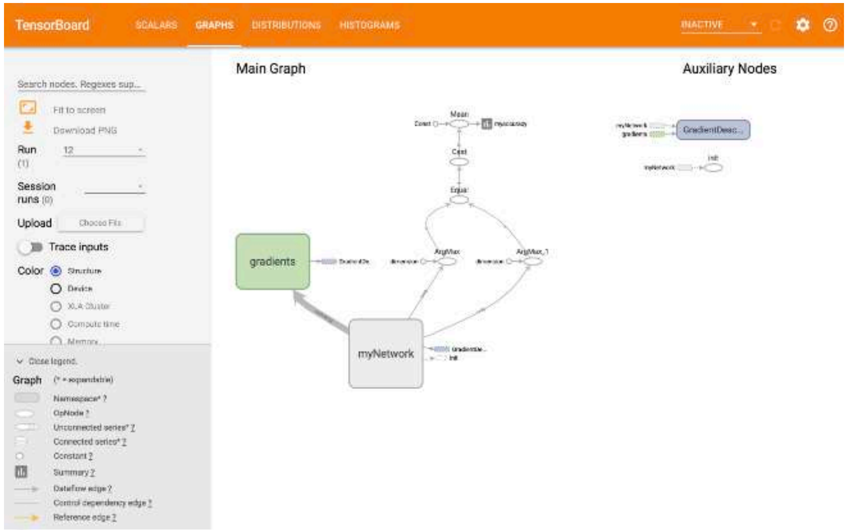
Example of a two layers network

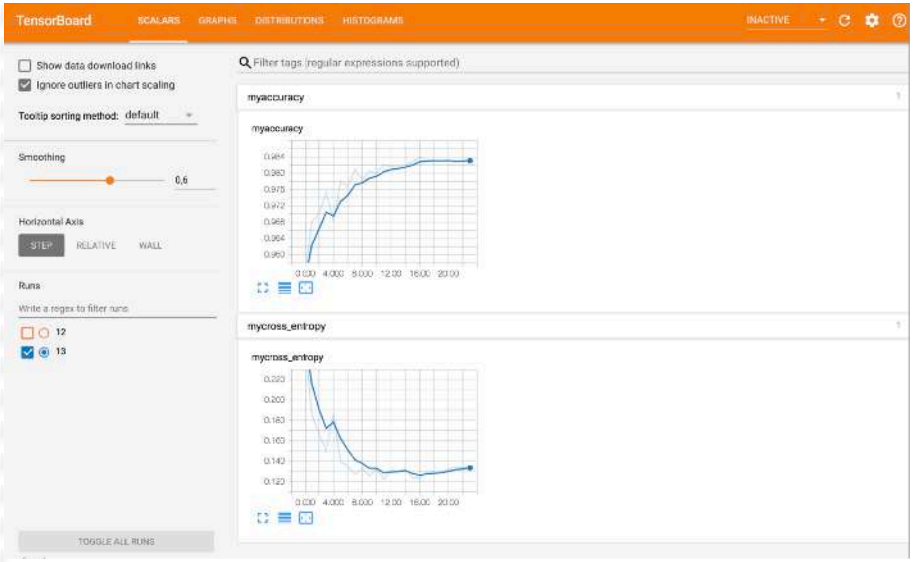


- The algorithms remain the same,
- even for more complex architectures
- Generalization by coding your own function node or by
- Wrapping a layer in a module

Deep Learning Frameworks

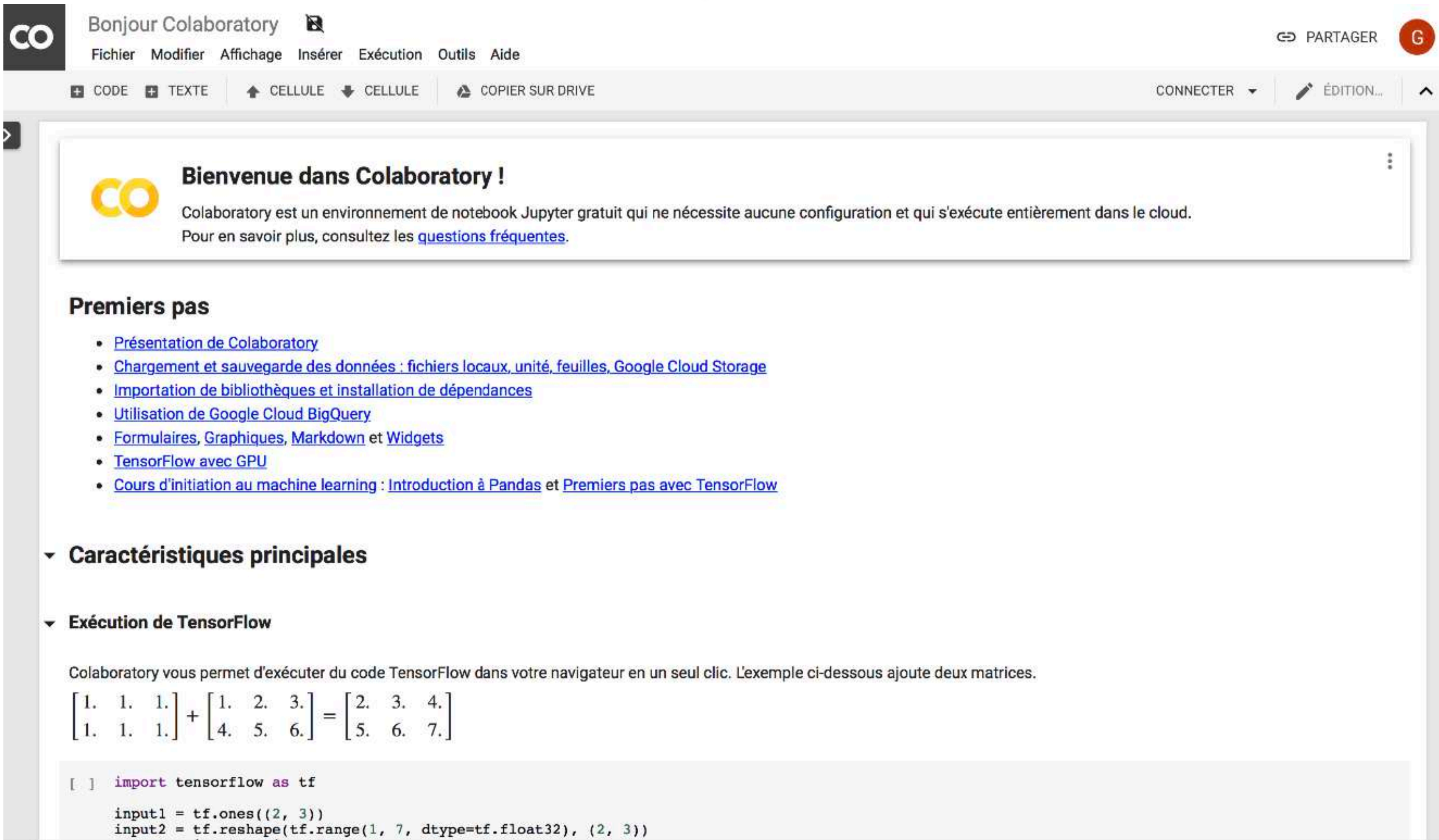
Tensorboard





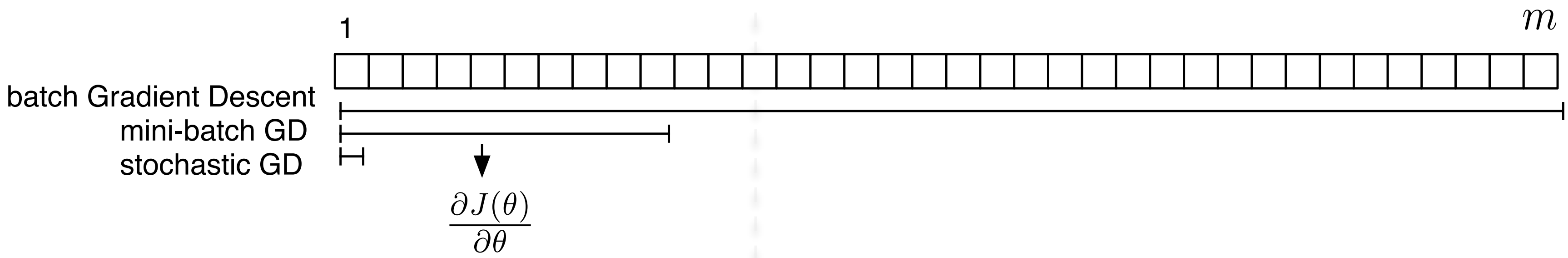
Deep Learning Frameworks

Colab https://colab.research.google.com/



Various types of training

Various types of training



- m training examples:
  - $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$
- Various types of training:
  - Batch Gradient Descent
  - Mini Batch Gradient Descent
  - Stochastic Gradient Descent (SGD)

Alternatives to gradient descent

Alternatives to gradient descent

(mini-batch) Gradient Descent

- **Notations**
  - iteration: t
  - parameter at iteration t: $\theta^{[t]}$, θ can be either $\mathbf{W}$ (weight matrix) or $\mathbf{b}$ (bias vector)
  - gradient of the loss w.r.t. θ: $\dfrac{\partial \mathscr{L}}{\partial \theta}$
- **Mini-batch Gradient Descent**

$$\theta^{[t]} = \theta^{[t-1]} - \alpha \frac{\partial \mathscr{L}(\theta^{[t-1]}, x^{(i:i+n)}, y^{(i:i+n)})}{\partial \theta}$$

- **Problems**
  - does not guarantee good convergence
  - neural network = highly non-convex error functions
    - avoid getting trapped in their numerous suboptimal local minima or saddle points (points where one dimension slopes up and another slopes down) which are usually surrounded by a plateau
  - choosing a proper learning rate can be difficult, need to adapt the learning rate to dataset's characteristics
  - each parameter may require a different learning rate (sparse data)
- **Alternatives to Gradient Descent**
  - first-order methods: Momentum, Nesterov (NAG), Adagrad, Adadelta/RMSprop, Adam
  - second-order methods: Newton

Alternatives to gradient descent

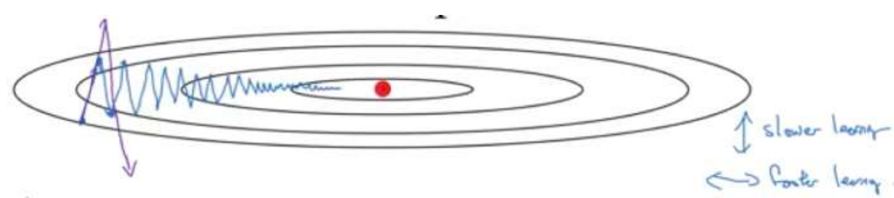
Momentum

- Goal ?
  - helps accelerating gradient descent in the relevant direction and dampens oscillations
- How ?
  - add a fraction β of the update vector of the past time step to the current step
- **Momentum**
  - On iteration t, compute $\dfrac{\partial \mathscr{L}(\theta^{[t-1]}, x, y)}{\partial \theta}$ on current mini-batch

$$V_{d\theta}^{[t]} = \beta V_{d\theta}^{[t-1]} + (1-\beta)\frac{\partial \mathscr{L}(\theta^{[t-1]}, x, y)}{\partial \theta}$$

$$\theta^{[t]} = \theta^{[t-1]} - \alpha V_{d\theta}^{[t]}$$

  - usual choice: $\beta = 0.9$
- Explanation: the momentum term
  - increases for dimensions whose gradients point in the same directions
  - reduces updates for dimensions whose gradients change directions
  - gain faster convergence and reduced oscillation
- β plays the role of a friction parameter
  - $V_{d\theta}$ plays the role of the velocity
  - $\dfrac{\partial \mathscr{L}}{\partial \theta}$ plays the role of acceleration



Alternatives to gradient descent

Nesterov Accelerated Gradient (NAG)

- Problem:
  - "A ball that rolls down a hill, blindly following the slope, is highly unsatisfactory."
- Solution:
  - "We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again."
  - We know that we will use our momentum term $\beta V_{d\theta}^{[t-1]}$ to move θ
    - $\theta^{[t]} = \theta^{[t-1]} - \alpha \left[\beta V_{d\theta}^{[t-1]} + (1-\beta) \frac{\partial \mathcal{L}}{\partial \theta} \right]$
    - We therefore compute the derivative of the loss at $\theta^{[t-1]} - \alpha\beta V_{d\theta}^{[t-1]}$ instead of $\theta^{[t-1]}$
    - This gives us an approximation of the next positon of θ
- Nesterov Accelerated Gradient
  - At iteration t

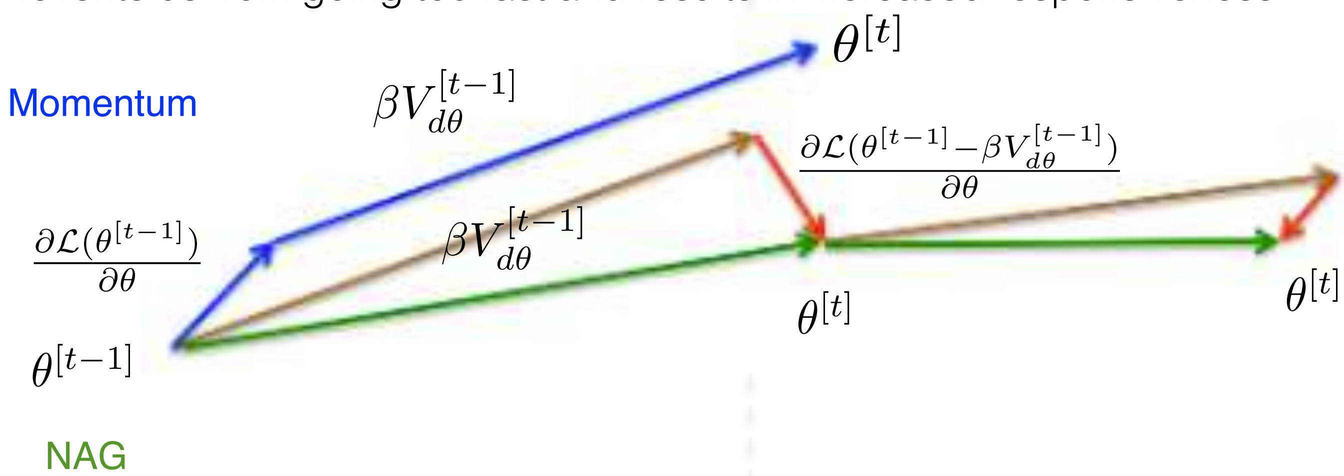
$$V_{d\theta}^{[t]} = \beta V_{d\theta}^{[t-1]} + (1-\beta) \frac{\partial \mathcal{L}(\theta^{[t-1]} - \alpha\beta V_{d\theta}^{[t-1]}, x, y)}{\partial \theta}$$

$$\theta^{[t]} = \theta^{[t-1]} - \alpha V_{d\theta}^{[t]}$$

Alternatives to gradient descent

Nesterov Accelerated Gradient (NAG)

- **Momentum**
  - 1) computes the gradient at the current ($\theta^{[t-1]}$) position: $\dfrac{\partial \mathscr{L}(\theta^{[t-1]})}{\partial \theta}$
  - 2) before using it, we do a big jump *in the direction of the previous accumulated gradient* $\alpha \beta V_{d\theta}^{[t-1]}$
- **Nesterov Accelerated Gradient (NAG)**
  - 1) big jump *in the direction of the previous accumulated gradient* $\alpha \beta V_{d\theta}^{[t-1]}$
  - 2) compute the gradient at the propagated ($\theta^{[t-1]} - \alpha \beta V_{d\theta}$) position: $\dfrac{\partial \mathscr{L}(\theta^{[t-1]} - \alpha \beta V_{d\theta}^{[t-1]})}{\partial \theta}$
  - 3) makes the correction
- Prevents us from going too fast and results in increased responsiveness



Alternatives to gradient descent

AdaGrad

- **Goal**: adapt the updates to each individual parameters
  - we want smaller update (lower learning rate) for frequently occurring features
  - we want larger update (higher learning rate) for infrequent features
- **Notation**: $d\theta_i^{[t]} = \dfrac{\partial \mathcal{L}(\theta^{[t]}, x, y)}{\partial \theta_i}$
- **SGD**: $\theta_i^{[t]} = \theta_i^{[t-1]} - \alpha d\theta_i^{[t-1]}$
- **AdaGrad**
  - Compute the past gradients that have been used for θ_i: $G_{i,i}^{[t]} = \sum_{\tau=0}^{t} d\theta_i^{[\tau]^2}$

$$G_{i,i}^{[t]} = \sum_{\tau=0}^{t} d\theta_i^{[\tau]^2}$$

$$\theta_i^{[t]} = \theta_i^{[t-1]} - \frac{\alpha}{\sqrt{G_{i,i}^{[t-1]} + \epsilon}} d\theta_i^{[t-1]}$$

- **Problem**
  - the gradient are accumulated since the beginning
  - the learning rates will shrink

Alternatives to gradient descent

AdaDelta

– Extension of AdaGrad: instead of accumulating all past squared gradients, we restrict the window of accumulated past gradients to some fixed size

– **AdaDelta:**

$$\mathbb{E}[d\theta^2]^{[t]} = \gamma \mathbb{E}[d\theta^2]^{[t-1]} + (1-\gamma) {d\theta^{[t]}}^2$$
$$\theta^{[t]} = \theta^{[t-1]} - \frac{\alpha}{\sqrt{\mathbb{E}[d\theta^2]^{[t-1]} + \epsilon}} d\theta^{[t-1]}$$

RMSprop (Root Mean Square prop)

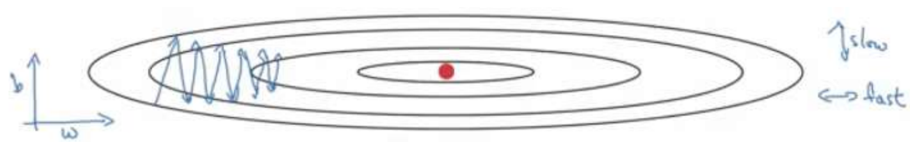
– On iteration t compute $d\theta$ on current mini-batch

– RMSprop:

$$S_{d\theta}^{[t]} = \gamma S_{d\theta}^{[t-1]} + (1-\gamma) {d\theta^{[t]}}^2$$
$$\theta^{[t]} = \theta^{[t-1]} - \frac{\alpha}{\sqrt{S_{d\theta}^{[t-1]} + \epsilon}} d\theta^{[t-1]}$$

– Want speed up in horizontal (W)

- $S_{dW}^{[t]}$ small $\Rightarrow 1/\sqrt{S_{dW}^{[t]} + \epsilon}$ large $\Rightarrow$ speed up



– Want slow down (damping) oscillation in vertical (b)

- $S_{dW}^{[t]}$ large $\Rightarrow 1/\sqrt{S_{dW}^{[t]} + \epsilon}$ small $\Rightarrow$ slow down

Adam (Adaptive moment estimation)

- Adam = **Mometum** + **AdaDelta/RMSprop**
  - On iteration t compute $d\theta$ on current mini-batch

$$V_{d\theta}^{[t]} = \beta_1 V_{d\theta}^{[t-1]} + (1-\beta_1) d\theta^{[t-1]}$$

$$S_{d\theta}^{[t]} = \beta_2 S_{d\theta}^{[t-1]} + (1-\beta_2) d\theta^{[t-1]^2}$$

$$\theta^{[t]} = \theta^{[t-1]} - \alpha \frac{V_{d\theta}^{[t]}}{\sqrt{S_{d\theta}^{[t]} + \epsilon}}$$

- **Hyperparameters**
  - α: learning rate (needs to be tuned)
  - $\beta_1 = 0.9$(momentum term, first moment)
  - $\beta_2 = 0.999$ (RMSprop term, second moment)
  - $\epsilon = 10^{-8}$ (avoid divide-by-zero)

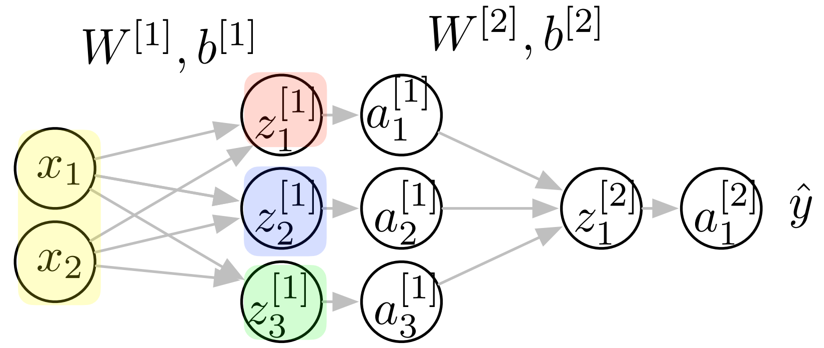
Weight initialisation

Weight initialisation

Initialise W with all zeros ?

$$\mathbf{W}^{[1]} = \begin{pmatrix} 000 \\ 000 \end{pmatrix}$$

$$\mathbf{W}^{[2]} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$



- then for any input $\Rightarrow a_1^{[1]} = a_2^{[1]} = a_3^{[1]}$
  - the three hidden units are computing exactly the same function
    - they are symmetric

$$\mathbf{z}^{[1]} = \mathbf{a}^{[0]}\mathbf{W}^{[1]}$$

$$\begin{pmatrix} z_1^{[1]} z_2^{[1]} z_3^{[1]} \end{pmatrix} = \begin{pmatrix} a_1^{[0]} a_2^{[0]} \end{pmatrix} \begin{pmatrix} w_{11}^{[1]} & w_{12}^{[1]} & w_{13}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & w_{23}^{[1]} \end{pmatrix}$$

$$\mathbf{z}^{[2]} = \mathbf{a}^{[1]}\mathbf{W}^{[2]}$$

$$z_1^{[2]} = \begin{pmatrix} a_1^{[1]} a_2^{[1]} a_3^{[1]} \end{pmatrix} \begin{pmatrix} w_{11}^{[2]} \\ w_{21}^{[2]} \\ w_{31}^{[2]} \end{pmatrix}$$

Weight initialisation

Initialise W with all zeros ?

– **Back-propagation ?**

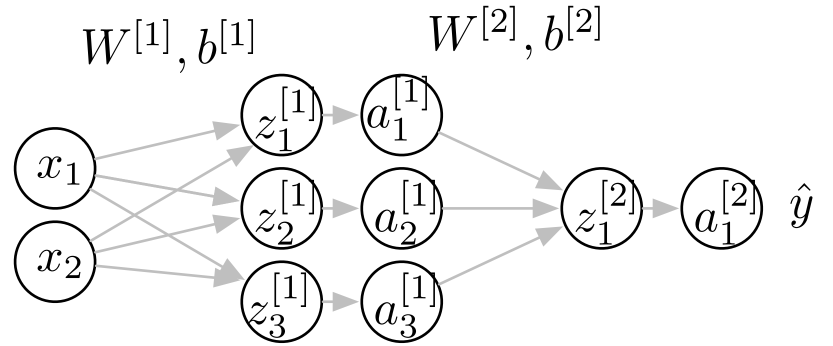
$$\frac{\partial \mathcal{L}}{\partial z^{[2]}} = a^{[2]} - y$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[2]}} = \mathbf{a}^{[1]} \frac{\partial \mathcal{L}}{\partial z^{[2]}} = \begin{pmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{pmatrix} \frac{\partial \mathcal{L}}{\partial z^{[2]}}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[1]}} = \frac{\partial \mathcal{L}}{\partial z^{[2]}} \mathbf{W}^{[2]T} = \frac{\partial \mathcal{L}}{\partial z^{[2]}} \left(w_{11}^{[2]} w_{21}^{[2]} w_{31}^{[2]} \right)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[1]}} \odot g^{[1]'}(z^{[1]}) = \frac{\partial \mathcal{L}}{\partial z^{[2]}} \left(w_{11}^{[2]} w_{21}^{[2]} w_{31}^{[2]} \right) \odot g^{[1]'} \left(z_1^{[1]} z_2^{[1]} z_3^{[1]} \right)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[1]}} = \mathbf{a}^{[0]} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}} = \begin{pmatrix} a_1^{[0]} \\ a_2^{[0]} \end{pmatrix} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}{}_1} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}{}_2} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}{}_3} \right) = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}{}_1} a_1^{[0]} & \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}{}_2} a_1^{[0]} & \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}{}_3} a_1^{[0]} \\ \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}{}_1} a_2^{[0]} & \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}{}_2} a_2^{[0]} & \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}{}_3} a_2^{[0]} \end{pmatrix}$$



– When we compute back-propagation, we have $\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}{}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}{}_2} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[1]}{}_3}$

- Therefore $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[1]}}$ is in a symmetric form $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[1]}} = \begin{pmatrix} u & u & u \\ v & v & v \end{pmatrix}$ and the update $\mathbf{W}^{[1]} = \mathbf{W}^{[1]} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[1]}}$ will keep the symmetricity: $z_1^{[1]} = z_2^{[1]} = z_3^{[1]} = a_1^{[0]} u + a_2^{[0]} v$

– so this is not use-full since we want the different units to compute difference functions

- ⇒initialise the parameters randomly

Weight initialisation with random values

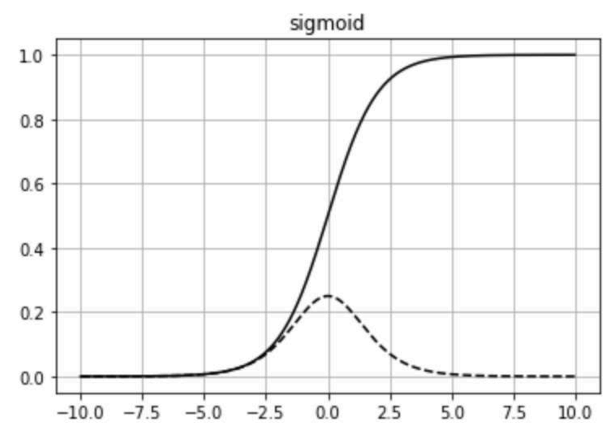
- **Random initialisation**
  - $\mathbf{W}^{[1]}$ = np.random.randn((2,3))*0.01
  - $\mathbf{b}^{[1]}$= 0
  - $\mathbf{W}^{[2]}$ = np.random.randn((3,1))*0.01
  - $\mathbf{b}^{[2]}$ = 0
- Remark: $\mathbf{b}$ doesn't have the symmetry problem

- **Why 0.01 ?**
  - If $\mathbf{W}$ is big $\Rightarrow Z$ is also big

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

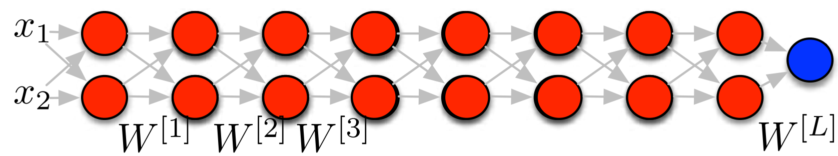
$$\mathbf{A}^{[1]} = g^{[1]}(\mathbf{Z}^{[1]})$$

  - $\Rightarrow$we are in the flat part of the sigmoid/tanh
    - $\Rightarrow$ slope is small
    - $\Rightarrow$ gradient descent slow
    - $\Rightarrow$ learning slow
  - Better to initialise to a very small value (valid for $\sigma(z)$ and $\tanh(z)$)



Weight initialisation

Vanishing/ exploding gradients



- **For a very deep neural network**
  - suppose $g^{[l]}(z) = z$ (linear activation) and $b^{[l]} = 0$
  - then

$$y = \underbrace{\underbrace{\mathbf{X}\,\mathbf{W}^{[1]}}_{\mathbf{A}^{[1]}=g(\mathbf{Z}^{[1]})=\mathbf{Z}^{[1]}} \quad \mathbf{W}^{[2]}}_{\mathbf{A}^{[2]}=g(\mathbf{Z}^{[2]})=\mathbf{Z}^{[2]}} \ldots \mathbf{W}^{[L]}$$

- **Exploding gradient**
  - suppose

$$\mathbf{W}^{[l]} = \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}$$

  - then

$$\hat{y} = \mathbf{X} \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}^{L-1} \mathbf{W}^{[L]}$$

  - if L is large $\Rightarrow (1.5)^{L-1}$is very large
    - $\Rightarrow$ the value of $\hat{y}$ will explode
  - Similar arguments can be used for the gradient

- **Vanishing gradient**
  - suppose

$$\mathbf{W}^{[l]} = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}$$

  - if L is large $\Rightarrow (0.5)^{L-1}$is very small
    - $\Rightarrow$ the value of $\hat{y}$ will vanish

Weight initialisation for Deep Neural Networks

- Suppose a single neuron network
  $$z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$
- In order to avoid vanishing/exploding gradient) \AR
  - The larger n is $\Rightarrow$the smallest w_d should be
- Solution ?
  - set $\text{Var}(w_i) = 1/n$
- In practice
  $\mathbf{W}^{[l]} =$ np.random.randn(shape) * np.sqrt $\left(1/n^{[l-1]}\right)$
- Other possibilities
  - For a ReLu: $\text{Var}(w_i) = \frac{2}{n}$works a bit better
  - For a $\tanh$
    - $\text{Var}(w_i) = \frac{1}{n^{[l-1]}}$: Xavier initialisation
    - $\text{Var}(w_i) = \frac{2}{n^{[l-1]} n^{[l]}}$: Bengio initialisation

Regularisation

L1 and L2 regularisation

- **Goal** ?
  - avoid over-overfitting (high variance)
- **How** ?
  - reduce model complexity
- In logistic regression

$$J(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^{m} \mathscr{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda_1}{2m} ||w||_1 + \frac{\lambda_2}{2m} ||w||_2^2 +$$

with $||w||_1 = \sum_{j=1}^{n_x} |w_j|$ and $||w||_2^2 = \sum_{j=1}^{n_x} w_j^2 = \mathbf{w}^T \mathbf{w}$

- **L1 regularisation** (Lasso):
  - will end up with sparse $\mathbf{w}$ (many zero)
- **L2 regularisation** (Ridge):
  - will end up with small values of $\mathbf{w}$
- L1+L2: Elastic Search
  - λ is the regularisation parameter (hyper-parameter)

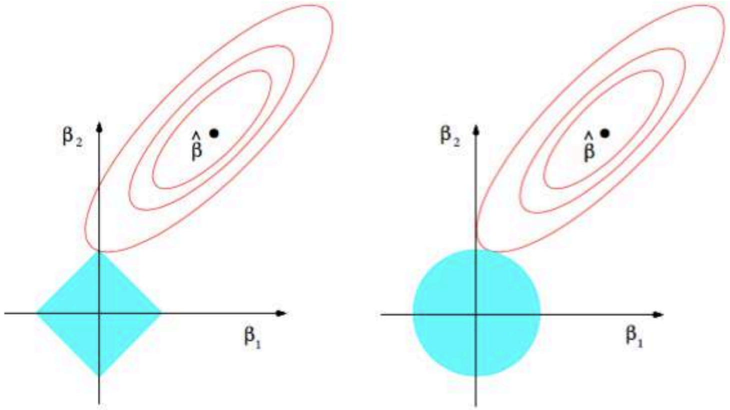


FIGURE 3.11. *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions $|\beta_1| + |\beta_2| \leq t$ and $\beta_1^2 + \beta_2^2 \leq t^2$, respectively, while the red ellipses are the contours of the least squares error function.*

L1 and L2 regularisation

- In neural network

$$J(\dots) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} ||\mathbf{W}^{[l]}||^2$$

- where $||\mathbf{W}||_2^2 = ||\mathbf{W}||_F$ is the "Frobenius norm"

$$||\mathbf{W}||^2 = \sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}} (\mathbf{W}_{i,j}^{[l]})^2$$

- In gradient descent ?

$$d'\mathbf{W}^{[l]} = \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} + \frac{\lambda}{m}\mathbf{W}^{[l]}$$

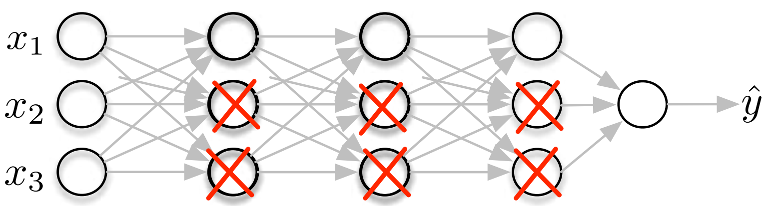
- Therefore

$$\begin{aligned}
\mathbf{W}^{[l]} &\leftarrow \mathbf{W}^{[l]} - \alpha d'\mathbf{W}^{[l]} \\
&\leftarrow \mathbf{W}^{[l]} - \alpha\left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} + \frac{\lambda}{m}\mathbf{W}^{[l]}\right) \\
&\leftarrow \mathbf{W}^{[l]} - \frac{\alpha\lambda}{m}\mathbf{W}^{[l]} - \alpha\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} \\
&\leftarrow \underbrace{\mathbf{W}^{[l]}\underbrace{\left(1 - \frac{\alpha\lambda}{m}\right)}_{\leq 1}}_{\text{weight decay}} - \alpha\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}}
\end{aligned}$$

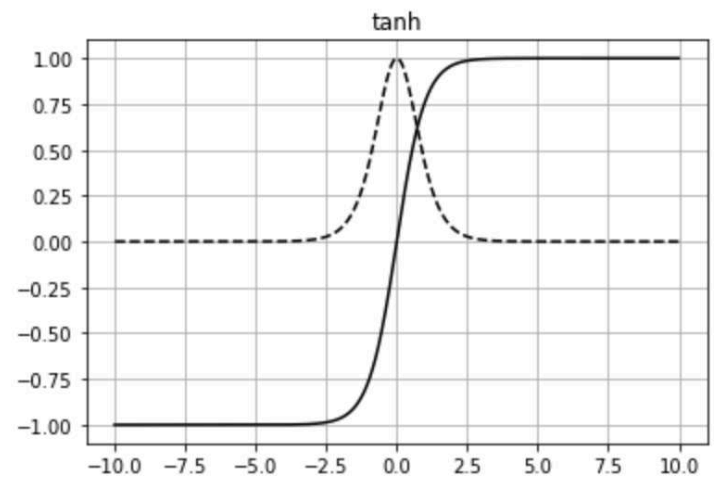
Regularisation

Why regularisation reduces over-fitting ?

- **Intuition 1**
  - $$J = \frac{1}{m}\sum_{i=1}^{m} \mathscr{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} ||\mathbf{W}^{[l]}||^2$$
  - If we set λ very very big then $\mathbf{W}^{[l]} \simeq 0$
    - $\Rightarrow$ many hidden units are not active
    - $\Rightarrow$ the network becomes much simpler $\Rightarrow$ avoid over-fitting



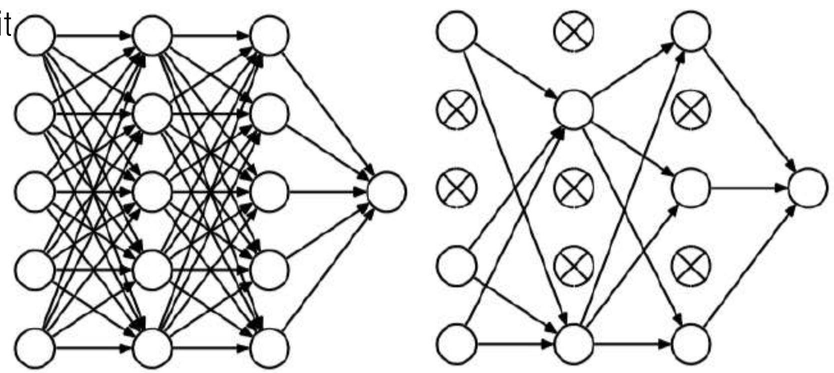
- **Intuition 2**
  - Suppose we are using a $\tanh$
    - If z is small, the $\tanh$ is linear
  - If λ is large,
    - then $\mathbf{W}^{[l]}$ is small,
    - then $z^{[l]}$ is small
    - then every layer is almost linear,
    - $\Rightarrow$ the whole network is linear



Regularisation

DropOut regularisation

- **During training**
  - for each training example **randomly turn-off** the neurons of hidden units (with $p = 0.5$)
    - this also removes the connections
  - for different training examples, turn-off different units
  - possible to vary the probability across layers
    - for large matrix $\mathbf{W} \Rightarrow p$ is higher
    - for small matrix $\mathbf{W} \Rightarrow p$ is lower



- **During testing**
  - no drop out
- **Dropout effects:**
  - prevents co-adaptation between units
  - can be seen as averaging different models that share parameters
  - acts as a powerful regularisation scheme
  - since the network is smaller, it is easier to train (as regularisation)
  - The network cannot rely on any feature, it has to spread out weights
    - Effect: shrinking the squared norm of the weights (similar to L2 regularisation)
    - Can be shown to be an adaptive form of L2-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

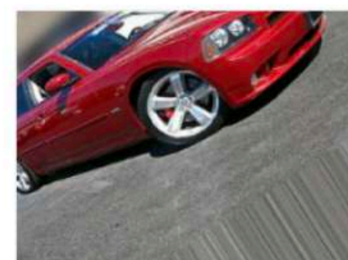

Car_A_0_3614


Car_A_0_3686


Car_A_0_3894


Car_A_0_5212


Car_A_0_5528


Car_A_0_5574

Normalisation

Normalisation

Normalising the inputs

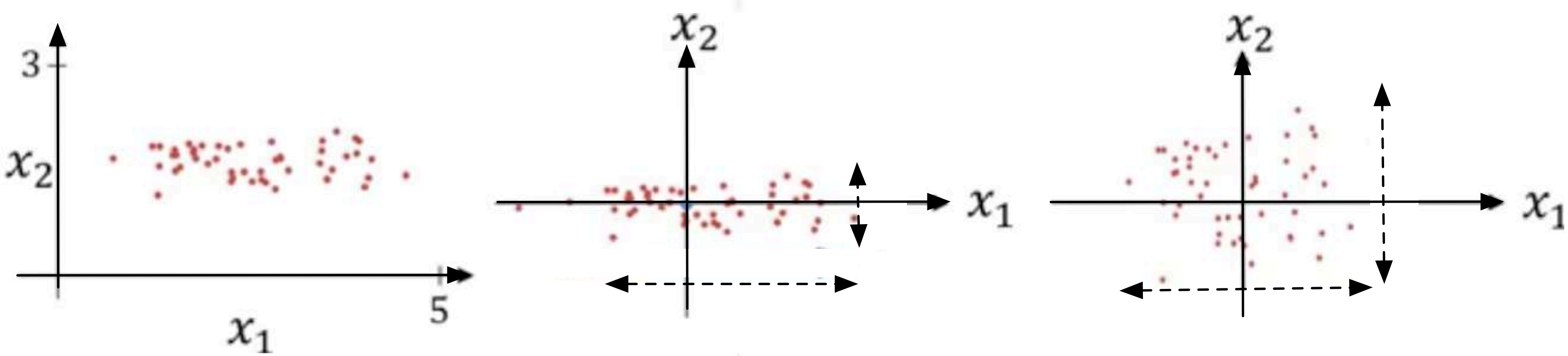
- "**Standardising**":
  - subtracting a measure of location and dividing by a measure of scale
  - subtract the mean and divide by the standard deviation

$$\mu_d = \frac{1}{m}\sum_{i=1}^{m} x_d^{(i)} \rightarrow x_d = x_d - \mu_d$$

$$\sigma_d^2 = \frac{1}{m}\sum_{i=1}^{m} (x_d^{(i)} - \mu_d)^2 \rightarrow x_d = \frac{x_d}{\sigma_d}$$

  - We use also μ_{train} and σ^2_{train} to standardise the test set

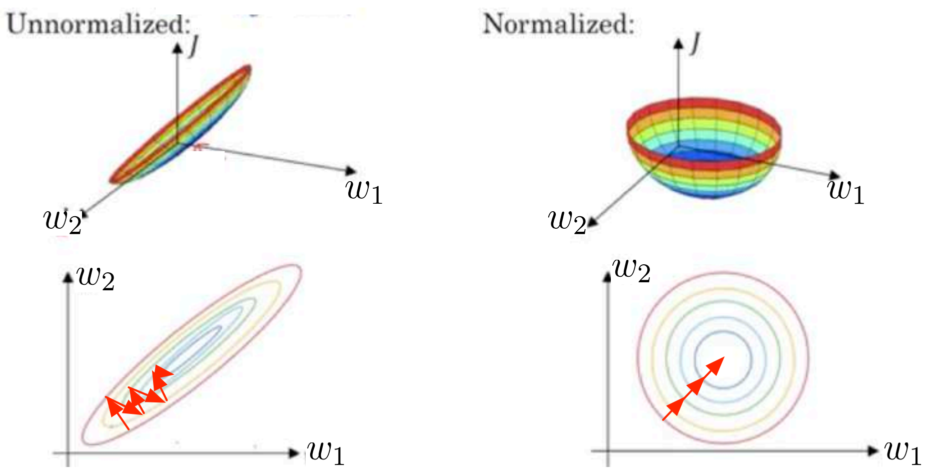
- "**Normalising**":
  - rescaling by the minimum and range of the vector
  - make all the elements lie between 0 and 1.



Normalisation

Normalising the inputs

- Suppose: $x_1 \in [1\ldots1000]$ and $x_2 \in [0\ldots1]$
    - Then w_1 and $w_2$will take very different value
- if no normalisation
    - many oscillations
    - need to use a small learning rate
- **Why use input normalisation ?**
    - get similar values for w_1 and w_2
        - gradient descent can go straight to the minimum
        - can use large learning rate
    - avoid each activation to be large (see sigmoid activation)
    - numerical instability



Batch Normalisation (BN)

- **Objective ?**
  - Apply the same normalisation for the input of each layer $[l]$
  - allows to learn faster

- Try to **reduce the "covariate shift"**
  - the inputs of a given layer $[l]$ is the outputs of the previous layer $[l-1]$
  - these outputs $[l-1]$ depends on the parameters of the previous layer which change over training !
  - **normalise the output of the previous layer** $a^{[l-1]}$
  - in practice normalise the pre-activation $z^{[l-1]}$

- Don't want all units to always have mean 0 and standard-deviation 1
  - Learn an appropriate bias β and scale γ to apply to $z^{[l-1]}$ before the non-linear function $g^{[l]}$

Sergey Ioffe and Christian Szegedy, "Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift" [ICML2015]

Normalisation

Batch Normalisation (BN)

- Given some intermediate values in the network: $z^{[l](1)}, z^{[l](2)}, \ldots, z^{[l](m)}$

- **Normalisation**

$$\mu^{[l]} = \frac{1}{m} \sum_i z^{[l](i)}, \qquad \sigma^{2^{[l]}} = \frac{1}{m} \sum_i (z^{[l](i)} - \mu^{[l]})^2$$

$$z_{norm}^{[l](i)} = \frac{z^{[l](i)} - \mu^{[l]}}{\sqrt{\sigma^{2^{[l]}} + \epsilon}}$$

- **Re-scaling:**
  - New parameters **to be trained**:β allows to set the **mean** of $\tilde{z}^{[l]}$, γ allows to set the **variance** of $\tilde{z}^{[l]}$,

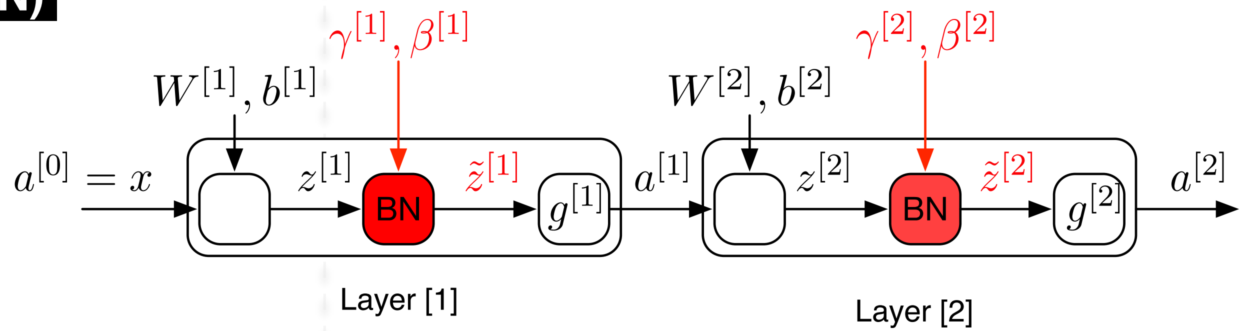
$$\tilde{z}^{[l](i)} = \gamma \cdot z_{norm}^{[l](i)} + \beta$$

- **Non-linearity**

$$a^{[l](i)} = g^{[l]}(\tilde{z}^{[l](i)})$$

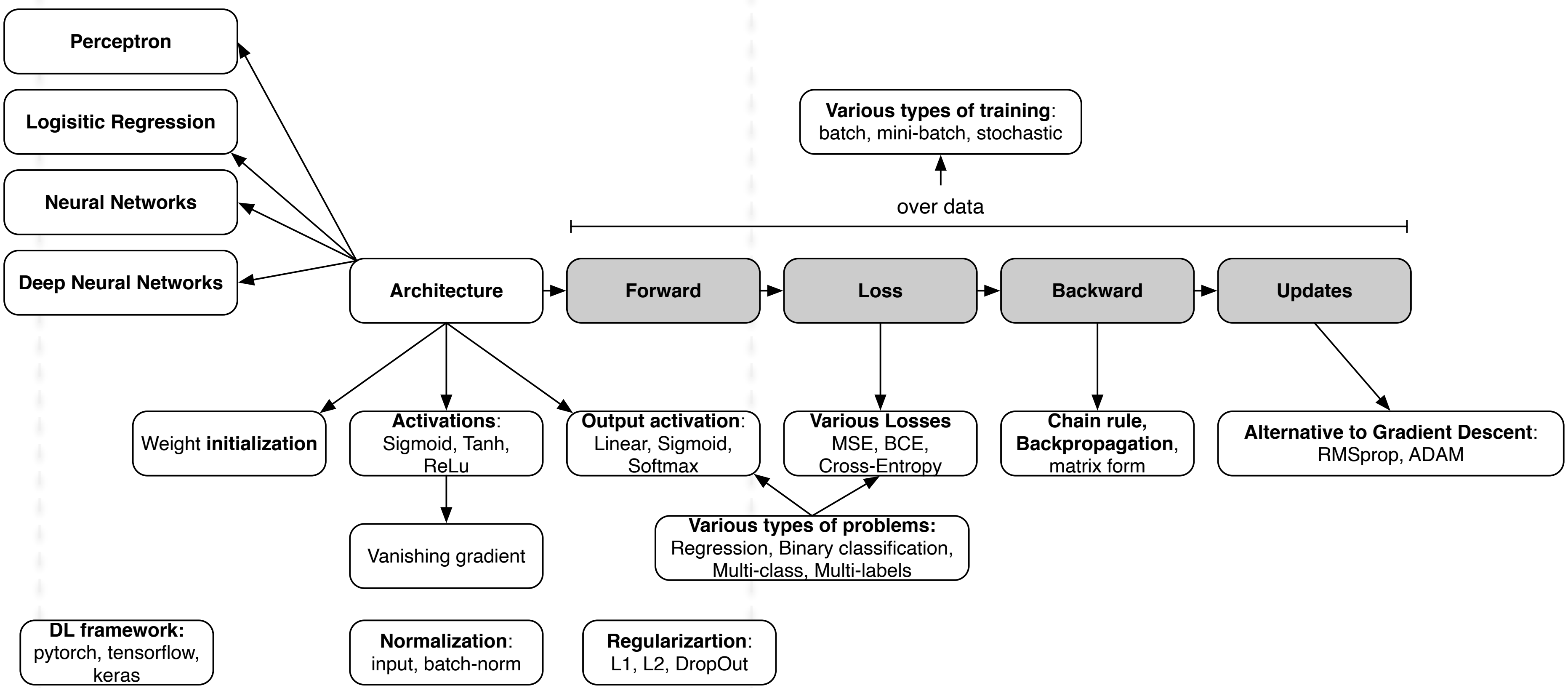
- **Note**: if $\gamma = \sqrt{\sigma^{2^{[l]}} + \epsilon}$ and $\beta = \mu$ then $\tilde{z}^{[l](i)} = z^{[l](i)}$

Batch Normalisation (BN)



- **New parameters:**
  - $\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \ldots$
- How to estimate $\beta^{[l]}, \gamma^{[l]}$?
  - $\beta^{[l]}, \gamma^{[l]}$ are estimated using **gradient-descent**
    - **Gradient**: $\dfrac{\partial \mathcal{L}}{\partial \beta^{[l]}}, \dfrac{\partial \mathcal{L}}{\partial \gamma^{[l]}}$
    - **Update**: $\beta^{[l]} = \beta^{[l]} - \alpha \dfrac{\partial \mathcal{L}}{\partial \beta^{[l]}}, \qquad \gamma^{[l]} = \gamma^{[l]} - \alpha \dfrac{\partial \mathcal{L}}{\partial \gamma^{[l]}}$
- **With mini-batches** $\{1\}, \{2\}, \ldots$
  - $a^{[0]\{1\}} \xrightarrow{W^{[1]},b^{[1]}} z^{[1]\{1\}} \xrightarrow{BN:\gamma^{[1]},\beta^{[1]}} \tilde{z}^{[1]\{1\}} \rightarrow a^{[1]\{1\}} = g^{[1]}(\tilde{z}^{[1]\{1\}})$
    - μ and σ are computed over the mini-batch$\{1\}$
  - $a^{[0]\{2\}} \xrightarrow{W^{[1]},b^{[1]}} z^{[1]\{2\}} \xrightarrow{BN:\gamma^{[1]},\beta^{[1]}} \tilde{z}^{[1]\{2\}} \rightarrow a^{[1]\{2\}} = g^{[1]}(\tilde{z}^{[1]\{2\}})$
    - μ and σ are computed over the mini-batch$\{2\}$

Overview



Perceptron
Logisitic Regression
Neural Networks
Deep Neural Networks
Architecture
Forward
Loss
Backward
Updates
Various types of training: batch, mini-batch, stochastic
over data
Weight initialization
Activations: Sigmoid, Tanh, ReLu
Output activation: Linear, Sigmoid, Softmax
Various Losses MSE, BCE, Cross-Entropy
Chain rule, Backpropagation, matrix form
Alternative to Gradient Descent: RMSprop, ADAM
Vanishing gradient
Various types of problems: Regression, Binary classification, Multi-class, Multi-labels
DL framework: pytorch, tensorflow, keras
Normalization: input, batch-norm
Regularizartion: L1, L2, DropOut