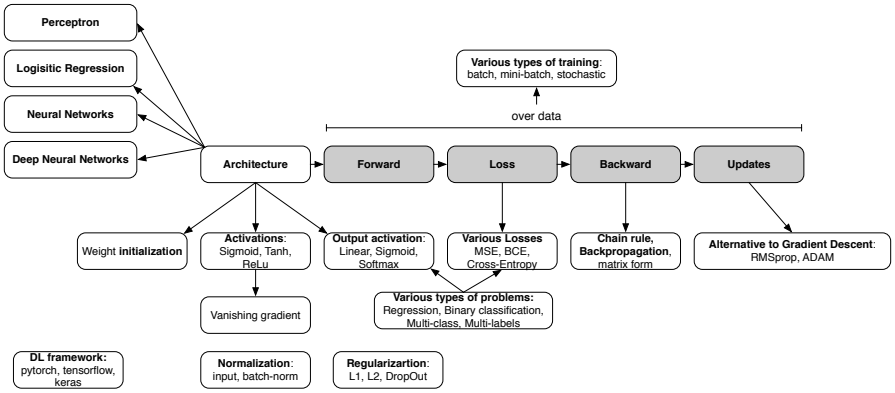
M2 Data Science - "Deep Learning 1" (IA-306)
Topics: Multi-Layers-Perceptron

Geoffroy Peeters

LTCI, Télécom Paris, IP Paris

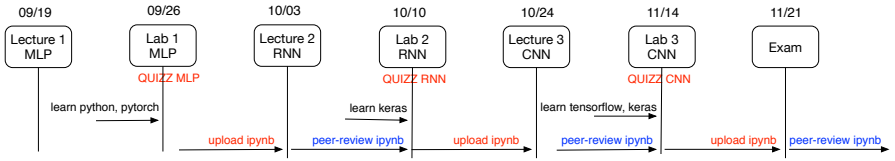
2019 - 2020

Overview



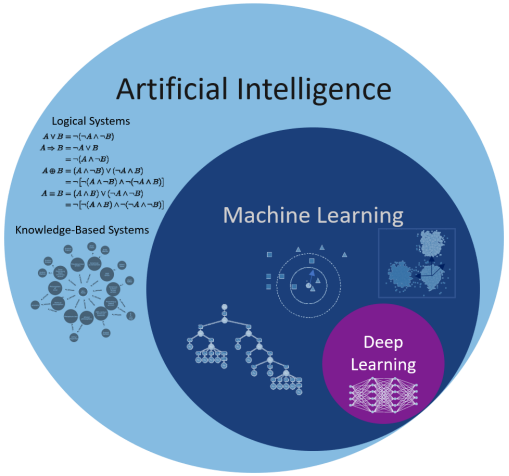
Timetable/ Organization

Date	Type	Conte	Teacher(s)	Location
September 19, 2019 PM	Lesson	MLP	G. Peeters	Télécom Paris (Amphi Estaunié)
September 26, 2019 PM	*Lab*	*MLP*	*G. Peeters, A. Newson + others*	*Télécom Paris (C126, C127, C129, C130, C45)*
October 3, 2019 PM	Lesson	RNN	G. Peeters	Télécom Paris (Amphi Estaunié)
October 10, 2019 PM	*Lab*	*RNN*	*G. Peeters, A. Newson + others*	*Télécom Paris (C126, C127, C129, C130, C45)*
October 24, 2019 PM	Lesson	CNN	A. Newson	Télécom Paris (Amphi Estaunié)
November 14, 2019	*Lab*	*CNN*	*A. Newson, G. Peeters + others*	*Télécom Paris/Saclay (1A201, 1A207, 1A226, 1A252, 1A260)*
November 21, 2019	Exam			Télécom Paris/Saclay (Amphi OB01)



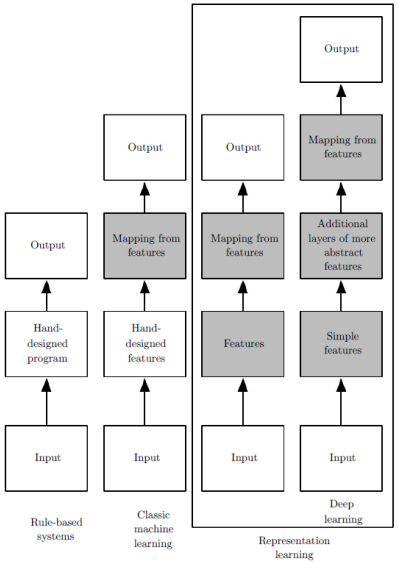
Deep Learning and Neural Networks : History

Deep learning (a subset of machine learning)



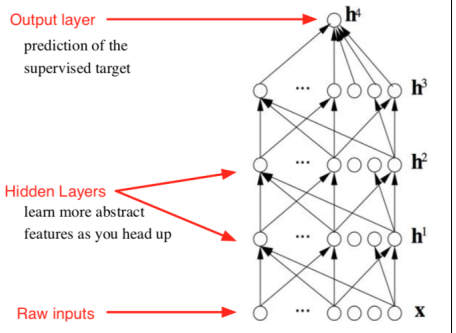
From http://canvart.club/atmose-11_21_13.html

Deep learning : learning hierachical representations

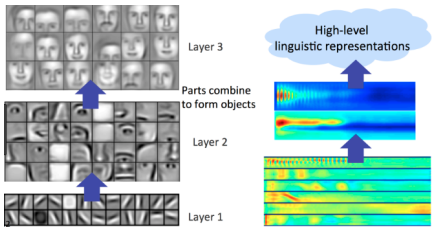


From Deep Learning by Ian Goodfellow and Yoshua Bengio and

Deep learning : learning hierachical representations

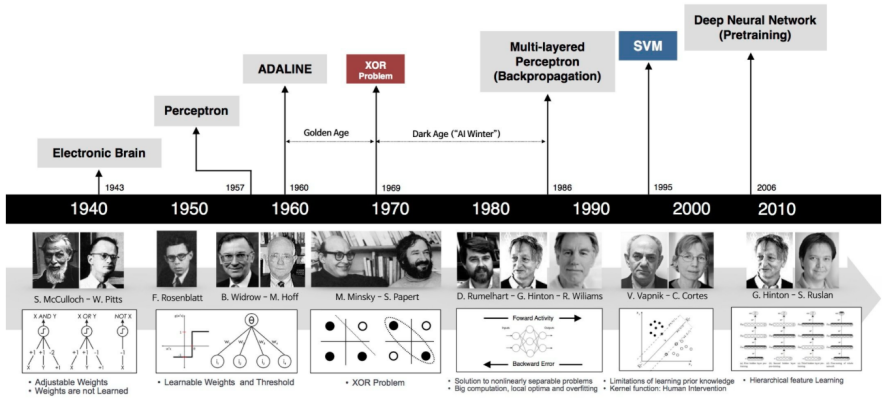


From Bengio 2015

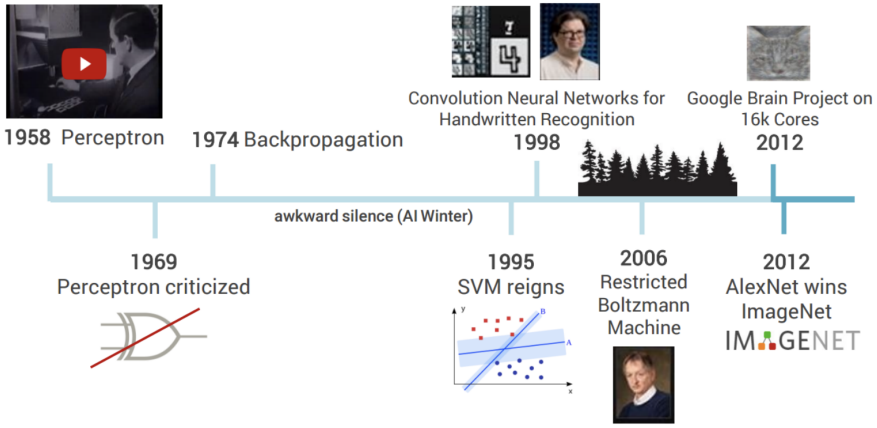


From Bengio 2015

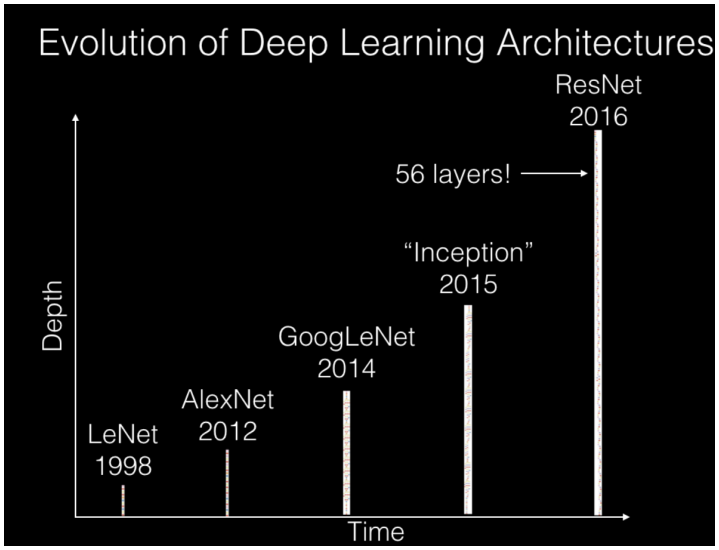
Deep Learning and Neural Networks : History



Deep Learning and Neural Networks : History

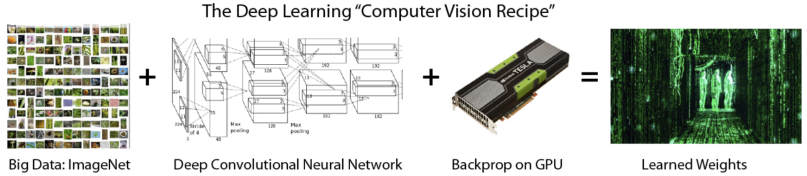


Deep Learning and Neural Networks : History

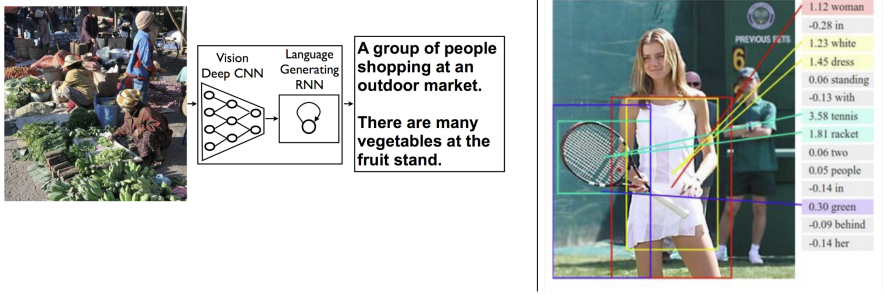


Deep Learning and Neural Networks : History

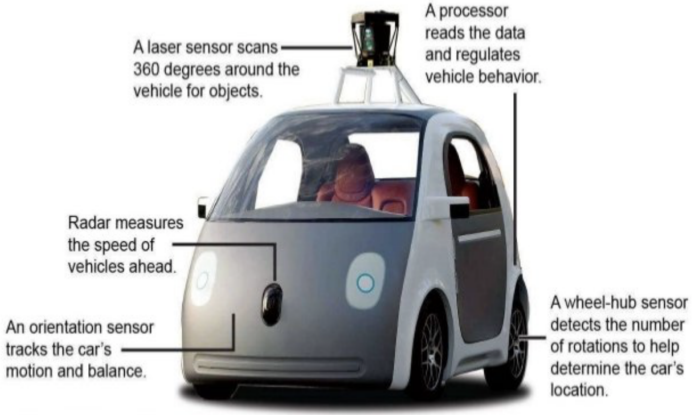
***Deep Learning* =**
Lots of training data* + *Parallel Computation* + *Scalable, smart algorithms



Automatic picture captioning



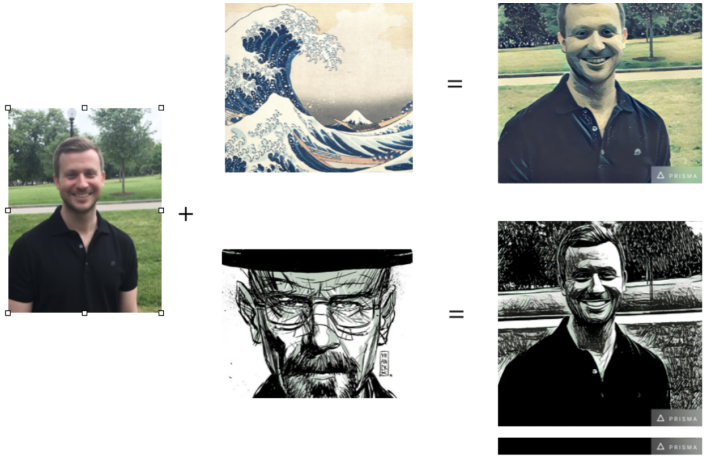
Autonomous car



Google Alpha Go

DEEP LEARNING HAS MASTERED GO

Style Transfer



Deep Learning and Neural Networks : History

- Yann LeCun (french)
  - Facebook AI (FAIR), University of New-York

- Geoffrey Hinton (canadian)
  - Google Brain, University of Toronto

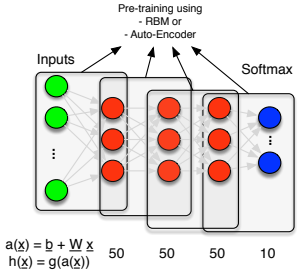
- Yoshua Bengio (french)
  - University of Montreal, MILA

Three main types of Nets

Three main types of Nets

Multi Layers Perceptron (MLP) or Fully-Connected (FC)

Multi Layers Perceptron (Fully Connected)

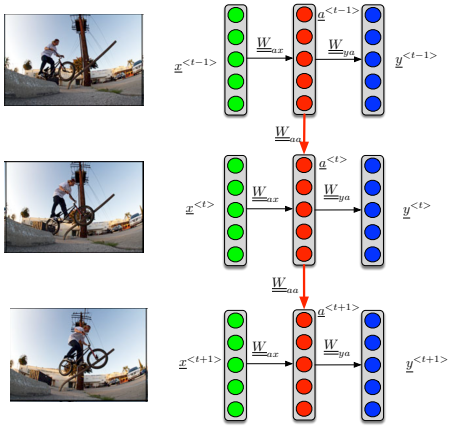


$$\underline{z}^{[l]} = \underline{\underline{W}}^{[l]}\underline{a}^{[l-1]} + \underline{b}^{[l]}$$
$$\underline{a}^{[l]} = g^{[l]}(\underline{z}^{[l]})$$

Three main types of Nets

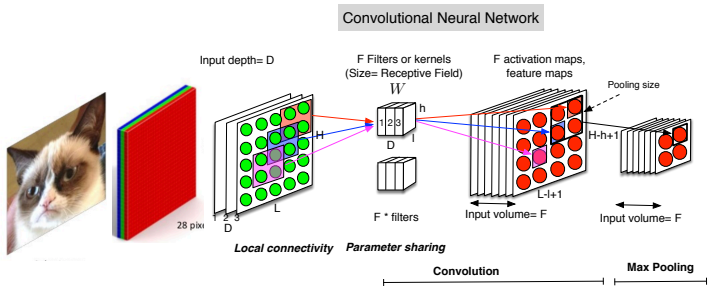
Recurrent Neural Networks (RNN)

Recurrent Neural Network



Three main types of Nets

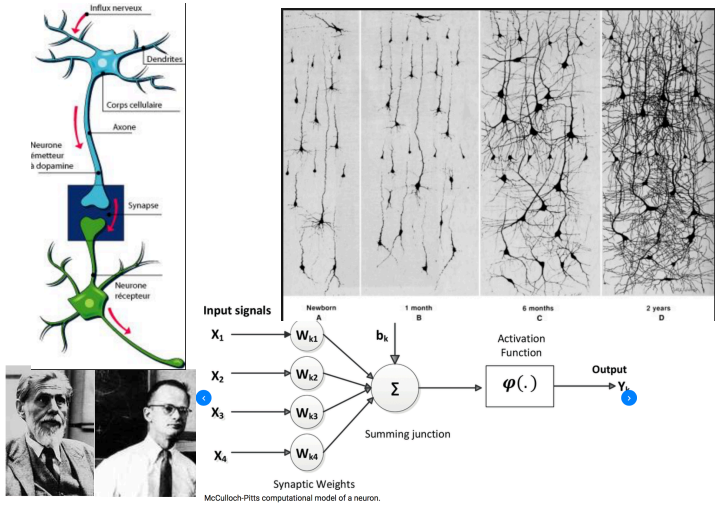
Convolutional Neural Networks (CNN)



Perceptron

Mc Culloch - Pitts model of a neuron

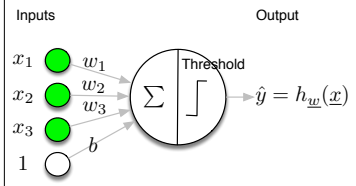
Warren S. McCulloch and Walter H. Pitts "A logical calculus of the ideas immanent in nervous activity", Bulletin of Mathematical Biophysisc, vol. 5, 1943, p. 115-133



Perceptron

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

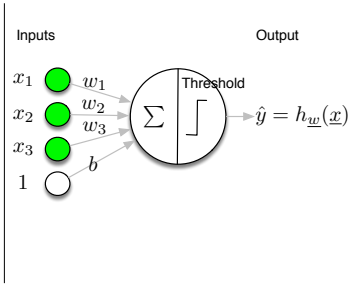
- **Perceptron ?**
  - A binary **classification algorithm** :
    - output is 0 or 1
  - Models the decision using a linear function, followed by a threshold
    $$\hat{y} = h_{\underline{w}}(\underline{x}) = \text{Threshold}(\underline{x}^T \cdot \underline{w}) \quad (1)$$
  - where
    - Threshold$(z) = 1$ if $z \geqslant 0$
    - Threshold$(z) = 0$ if $z < 0$
- **Parameters :**
  - weights $\underline{w}$ and bias b



Perceptron

Training

- **Training ?**
  - Adapt $\underline{w}$ and b so that $\hat{y} = h_{\underline{w}}(\underline{x})$ gives the correct answer y on training data
- **Algorithm :**
  - for each pair $(\underline{x}^{(i)}, y^{(i)})$
    - compute
      $\hat{y}^{(i)} = h_{\underline{w}}(\underline{x}^{(i)}) = \text{Threshold}(\underline{x}^{(i)\,T} \cdot \underline{w})$
    - if $\hat{y}^{(i)} \neq y(t)^{(i)}$ update
      - $w_d \leftarrow w_d + \alpha(y^{(i)} - \hat{y}^{(i)})x_d^{(i)} \quad \forall d$
      - where α is the learning rate



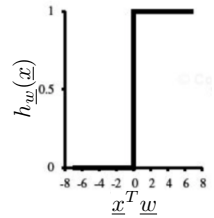
- **Three cases :**

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

Perceptron

Perceptron training as an optimization problem

- **Perceptron :**
  - $\hat{y}^{(i)} = h_{\underline{w}}(\underline{x}^{(i)}) = \text{Threshold}(\underline{x}^{(i)\,T} \cdot \underline{w})$
- Update rule :
  - $w_d \leftarrow w_d + \alpha(y^{(i)} - \hat{y}^{(i)})x_d^{(i)} \quad \forall d$
- It corresponds to **minimizing the loss**
  $\mathcal{L}(y^{(i)}, \hat{y}^{(i)}) = -(y^{(i)} - \hat{y}^{(i)})\underline{x}^{(i)\,T} \cdot \underline{w}$
  - if the predicition is good
    - cost = 0
  - if the prediction is wrong
    - cost = distance between $\underline{w} \cdot x^{(i)}$ and necessary threshold for the prediction to be good

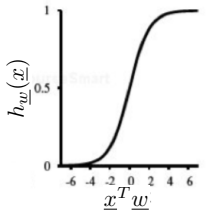


- Gradient descent ?
  $w_d \leftarrow w_d - \alpha\frac{\partial \mathcal{L}(y^{(i)}, \hat{y}^{(i)})}{\partial w_d} \quad \forall d$
- Gradient ?
  $\frac{\partial \mathcal{L}}{\partial w_d} = -\left(\frac{\partial\left(y^{(i)} - h_{\underline{w}}(x^{(i)})\right)}{\partial \underline{w}_d}\underline{x}^{(i)\,T} \cdot \underline{w} + \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_d^{(i)}\right) \cong -\left(0 + \left(y^{(i)} - h_{\underline{w}}(x^{(i)})\right) \cdot x_d^{(i)}\right)$
- Not exactly, since the derivative of $h_{\underline{w}}(\underline{x}^{(i)})$ does not exist at $\underline{w} \cdot \underline{x} = 0$
- $\Rightarrow$ Using the Threshold function can lead to instability during training

Perceptron

From Perceptron to Logistic Regression

- **Logisitic Regression**
  - instead of predicting a class, **predict a probability** to belong to this class
    $p(Y = 1|\underline{x}) = h_{\underline{w}}(\underline{x}) = \text{Logistic}(\underline{x}^T \cdot \underline{w}) = \frac{1}{1 + e^{-\underline{x}^T \cdot \underline{w}}}$
- Choose the class with the largest probability
  - If $h_{\underline{w}}(\underline{x}) \geqslant 0.5$ choose 1
  - If $h_{\underline{w}}(\underline{x}) < 0.5$ choose 0



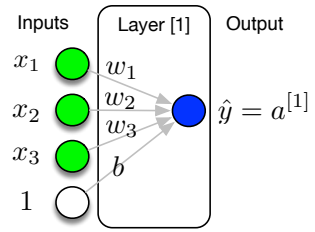
- We define the Loss as (binary cross-entropy)
  $\mathcal{L}(y^{(i)}, h_{\underline{w}}(x^{(i)})) = -y^{(i)} \log(h_{\underline{w}}(\underline{x}^{(i)})) - (1 - y^{(i)}) \log(1 - h_{\underline{w}}(\underline{x}^{(i)}))$
- Gradient descent ?
  $w_d \leftarrow w_d - \alpha\frac{\partial \mathcal{L}(y^{(i)}, h_{\underline{w}}(x^{(i)}))}{\partial w_d} \quad \forall d$
- Gradient ?
  $\frac{\partial \mathcal{L}(y^{(i)}, h_{\underline{w}}(x^{(i)}))}{\partial w_d} = -(y^{(i)} - h_{\underline{w}}(x^{(i)})) \cdot x_d \quad \forall d$
- The update rule is therefore the same as for the Perceptron but the definition of the Loss and of $h_{\underline{w}}(\underline{x})$ is different

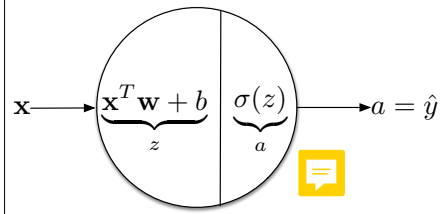
Logistic Regression (0 hidden layers)

Logistic Regression (0 hidden layers)

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

- Problem :
  - Given $\underline{x} \in \mathbb{R}^{n_x}$
  - Predict $\hat{y} = P(y = 1|\underline{x})$
    - with $0 \leqslant \hat{y} \leqslant 1$
- Model :
  - $\hat{y} = \sigma(\underline{x}^T \underline{w} + b)$
- Parameters :
  - $\underline{w} \in \mathbb{R}^{n_x}$ and $b \in \mathbb{R}$
- Sigmoid function : $\sigma(z) = \frac{1}{1+e^{-z}}$
  - If z is very large then $\sigma(z) \simeq \frac{1}{1+0} = 1$
  - If z is very (negative) small then $\sigma(z) = 0$





Logistic Regression (0 hidden layers)

Loss / Cost function (Empirical Risk Minimization)

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

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

- **Cost J function** : sum of the Loss for all training examples

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

  - In the case of the Binary Cross-Entropy

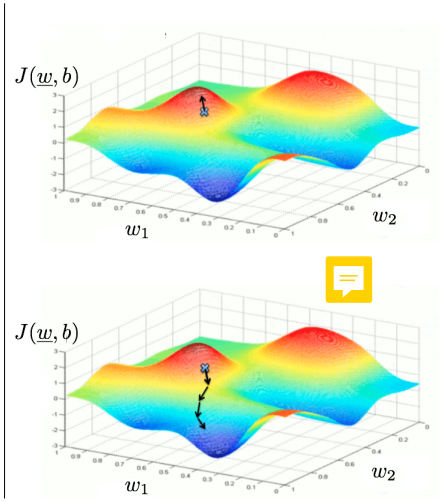
$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right)$$

- We want to find the parameters θ of the network that minimize $J(\theta)$

Logistic Regression (0 hidden layers)

Gradient Descent

- **How to minimize $J(\underline{w}, b)$?**
- The gradient $\frac{\partial J(\underline{w},b)}{\partial \underline{w}}$ points in the direction of the greatest rate of increase of the function
- We will go in the opposite direction : $-\frac{\partial J(\underline{w},b)}{\partial \underline{w}}$
  - We move down-hill in the steepest direction
- **Gradient descent :**
  - Repeat
    - $\underline{w} \leftarrow \underline{w} - \alpha \cdot \frac{\partial J(\underline{w},b)}{\partial \underline{w}}$
    - $b \leftarrow b - \alpha \cdot \frac{\partial J(\underline{w},b)}{\partial b}$
  - where α is the "learning rate"



Logistic Regression (0 hidden layers)

Gradient Descent

- **Parameters :** to update
  - $\underline{w}$, b
- **Gradient Descent :**
  - ⓘ **Initialize** the parameters
  - Repeat for # iterations
    - Repeat for all training examples $\forall i = 1, \ldots, m$
      - ⓕ **Forward propagation** : compute prediction $\hat{y}^{(i)}$
      - ⓛ **Compute the loss** $\mathcal{L}$
      - ⓑ **Backward propagation** : compute gradients $\frac{\partial \mathcal{L}}{\partial \underline{w}}$, $\frac{\partial \mathcal{L}}{\partial b}$
    - ⓤ **Update the parameters** using the learning rate α

Logistic Regression (0 hidden layers)

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

ⓕ Forward propagation

$$z^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + b$$
$$a^{(i)} = \sigma(z^{(i)}) \text{ with } \sigma(z) = \frac{1}{1+e^{-z}}$$

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

Logistic Regression (0 hidden layers)

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

ⓛ Loss

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

Logistic Regression (0 hidden layers)

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

ⓑ Backward propagation

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

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

$$dw_1 = \frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial w_1} = x_1 \cdot dz$$
$$dw_2 = \frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial w_2} = x_2 \cdot dz$$
$$db = \frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial b} = dz$$

- Notation : we will use $d\theta$ **for** $\frac{\partial \mathcal{L}}{\partial \theta}$
  - Examples :
    - $da = \frac{\partial \mathcal{L}}{\partial a}$
    - $dw = \frac{\partial \mathcal{L}}{\partial w}$
    - $db = \frac{\partial \mathcal{L}}{\partial b}$

Logistic Regression (0 hidden layers)

Gradient of the Cost / Gradient of the Loss

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

- **For all training examples**
  - Computing the Cost (sum of the Loss $\mathcal{L}$ over all training examples m)
    $$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

- **Minimizing the Cost**
  - We need to compute the gradient of the Cost w.r.t. the parameters θ
    $$\frac{\partial J(\theta)}{\partial \theta_1} = \frac{\partial}{\partial \theta_1}\left(\frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})\right) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}{\partial \theta_1}$$
  - $\Rightarrow$ the gradient of the Cost is the average (over the m training examples) of the gradient of the Loss

Logistic Regression (0 hidden layers)

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

ⓤ 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

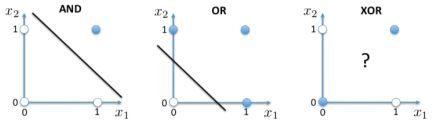
Neural Networks (1 hidden layer)

Logistic Regression (0 hidden layers)

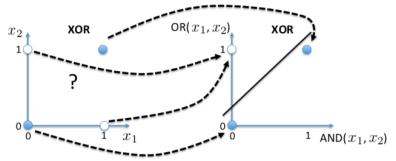
Limitation of linear classifiers

- Perceptron and Logistic Regression are linear classiers
  - What if classes are not linearly separable ?
  - What about the XOR function ?
    - Need 2 layers
      - $x_1^{[1]} = AND(x_1^{[0]}, x_2^{[0]})$
      - $x_2^{[1]} = OR(x_1^{[0]}, x_1^{[0]})$

- **1-Layer** :



- **2-Layers** :



Neural Networks (1 hidden layer)

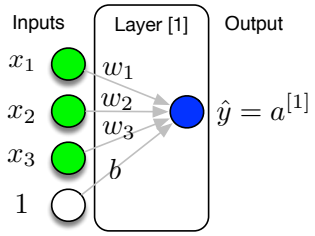
- **Logistic Regression** (1 layer, **0 hidden layer**)

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

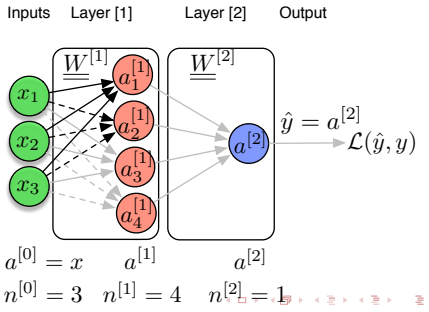
- **Neural Network** (2 layers, **1 hidden layer**)

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

Logistic Regression



Neural Network



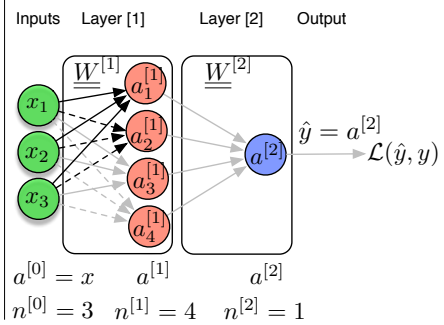
Neural Networks (1 hidden layer)

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

$$\underbrace{\qquad\qquad}_{Layer[1]} \qquad\qquad \underbrace{\qquad\qquad}_{Layer[2]}$$

- **Notations :**

$*^{[l]}$	variables of **layer** l
$*^{[l]}_d$	**dimension** d of variables
$*^{[l](i)}_d$	training **example** i
$n^{[l]}$	**number of neurons** of layer l
$g^{[l]}$	**activation function** of layer l (possibly a sigmoid σ function)



Neural Networks (1 hidden layer)

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

$$\underbrace{\qquad\qquad}_{Layer[1]} \qquad\qquad \underbrace{\qquad\qquad}_{Layer[2]}$$

Gradient Descent

- **Parameters :** to update
  - $\underbrace{\underline{\underline{W}}^{[1]}}_{(n^{[0]},n^{[1]})}$, $\underbrace{\underline{b}^{[1]}}_{(1,n^{[1]})}$, $\underbrace{\underline{\underline{W}}^{[2]}}_{(n^{[1]},n^{[2]})}$, $\underbrace{\underline{b}^{[2]}}_{(1,n^{[2]})}$
- **Gradient Descent :**
  - (i) **Initialize** the parameters
  - Repeat
    - (f) **Forward propagation** : compute prediction $\hat{y}^{(i)} \quad \forall i = 1, \ldots, m$
    - (l) **Compute the loss**
      - Cost function (binary classification) :
      - $J(w, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$
    - (b) **Backward propagation** : compute gradients $dW^{[l]}$, $db^{[l]}$
    - (u) **Update the parameters** using the learning rate α

Neural Networks (1 hidden layer)

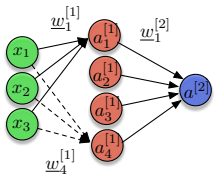
$$\underline{x} \xrightarrow{\underline{a}^{[0]}} \boxed{\underline{x}\,\underline{\underline{W}}^{[1]} + \underline{b}^{[1]}} \xrightarrow{\underline{z}^{[1]}} \boxed{g^{[1]}(\underline{z}^{[1]})} \xrightarrow{\underline{a}^{[1]}} \boxed{\underline{a}^{[1]}\underline{\underline{W}}^{[2]} + \underline{b}^{[2]}}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

$$\underbrace{\qquad\qquad}_{Layer[1]} \qquad\qquad \underbrace{\qquad\qquad}_{Layer[2]}$$

(f) Forward propagation (each dimension d, each training examples i)

`for i=1 to m`

- Input
  - $\underline{a}^{[0](i)} = \underline{x}^{(i)}$
- Output
  - $\hat{y}^{(i)} = \underline{a}^{[2](i)}$



- Layer 1
  - `for d=1 to 4`
    - $z_d^{[1](i)} = \underline{a}^{[0](i)}\,\underline{w}_d^{[1]} + b_d^{[1]}$
    - $a_d^{[1](i)} = g^{[1]}(z_d^{[1](i)})$
- Layer 2
  - `for d'=1 to 1`
    - $z_{d'}^{[2](i)} = \underline{a}^{[1](i)}\,\underline{w}_{d'}^{[2]} + b_{d'}^{[2]}$
    - $a_{d'}^{[2](i)} = g^{[2]}(z_{d'}^{[2](i)})$

Neural Networks (1 hidden layer)

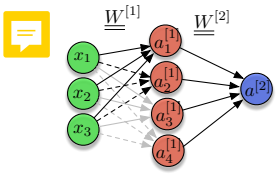
$$\underline{x} \xrightarrow{\underline{a}^{[0]}} \boxed{\underline{x}\,\underline{\underline{W}}^{[1]} + \underline{b}^{[1]}} \xrightarrow{\underline{z}^{[1]}} \boxed{g^{[1]}(\underline{z}^{[1]})} \xrightarrow{\underline{a}^{[1]}} \boxed{\underline{a}^{[1]}\underline{\underline{W}}^{[2]} + \underline{b}^{[2]}}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

$$\underbrace{\qquad\qquad}_{Layer[1]} \qquad\qquad \underbrace{\qquad\qquad}_{Layer[2]}$$

(f) Forward propagation (all dimensions, each i training examples)

`for i=1 to m`

- Input
  - $\underline{a}^{[0](i)} = \underline{x}^{(i)}$
- Output
  - $\hat{y}^{(i)} = \underline{a}^{[2](i)}$



- Layer 1

$$\underbrace{\underline{z}^{[1](i)}}_{(1,4)} = \underbrace{\underline{a}^{[0](i)}}_{(1,3)}\underbrace{\underline{\underline{W}}^{[1]}}_{(3,4)} + \underbrace{\underline{b}^{[1]}}_{(1,4)}$$

$$\underbrace{\underline{a}^{[1](i)}}_{(1,4)} = g^{[1]}(\underbrace{\underline{z}^{[1](i)}}_{(1,4)})$$

- Layer 2

$$\underbrace{\underline{z}^{[2](i)}}_{(1,1)} = \underbrace{\underline{a}^{[1](i)}}_{(1,4)}\underbrace{\underline{\underline{W}}^{[2]}}_{(4,1)} + \underbrace{\underline{b}^{[2]}}_{(1,1)}$$

$$\underbrace{\underline{a}^{[2](i)}}_{(1,1)} = g^{[2]}(\underbrace{\underline{z}^{[2](i)}}_{(1,1)})$$

Neural Networks (1 hidden layer)

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

$\underbrace{\qquad\qquad}_{Layer[1]}$ $\underbrace{\qquad\qquad\qquad\qquad}_{Layer[2]}$

ⓕ Forward propagation (all dimensions, all m training examples)

- Input
  - $\underline{\underline{A}}^{[0]} = \underline{\underline{X}}$
- Output
  - $\hat{\underline{\underline{Y}}} = \underline{\underline{A}}^{[2]}$

- Layer 1

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

$$\underbrace{\underline{\underline{A}}^{[1]}}_{(m,n^{[1]})} = g^{[1]}\left(\underline{\underline{Z}}^{[1]}\right)$$

- Layer 2

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

$$\underbrace{\underline{\underline{A}}^{[2]}}_{(m,n^{[2]})} = g^{[2]}\left(\underline{\underline{Z}}^{[2]}\right)$$

Neural Networks (1 hidden layer)

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

$\underbrace{\qquad\qquad}_{Layer[1]}$ $\underbrace{\qquad\qquad\qquad\qquad}_{Layer[2]}$

ⓑ Backward propagation (each training example)

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

- Layer 2 (input $da^{[2]}$)

$$dz^{[2]} = \frac{\partial \mathcal{L}}{\partial z^{[2]}} = \frac{\partial \mathcal{L}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}} = da^{[2]} \odot g^{[2]\prime}(z^{[2]})$$
$$= da^{[2]} \odot a^{[2]}(1-a^{[2]}) = a^{[2]} - y$$

$$\underbrace{dW^{[2]}}_{(n^{[1]},n^{[2]})} = \frac{\partial \mathcal{L}}{\partial W^{[2]}} = \frac{\partial \mathcal{L}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial W^{[2]}} = \underbrace{a^{[1]\,T}}_{(1,n^{[1]})}\ \underbrace{dz^{[2]}}_{(1,n^{[2]})}$$

$$db^{[2]} = \frac{\partial \mathcal{L}}{\partial b^{[2]}} = \frac{\partial \mathcal{L}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial b^{[2]}} = dz^{[2]}$$

– – –

$$da^{[1]} = \frac{\partial \mathcal{L}}{\partial a^{[1]}} = \frac{\partial \mathcal{L}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial a^{[1]}} = \underbrace{dz^{[2]}}_{(1,n^{[2]})}\ \underbrace{W^{[2]\,T}}_{(n^{[1]},n^{[2]})}$$

- Layer 1 (input $da^{[1]}$)

$$dz^{[1]} = \frac{\partial \mathcal{L}}{\partial z^{[1]}} = \frac{\partial \mathcal{L}}{\partial a^{[1]}}\frac{\partial a^{[1]}}{\partial z^{[1]}} = da^{[1]} \odot g^{[1]\prime}(z^{[1]})$$

$$\underbrace{dW^{[1]}}_{(n^{[0]},n^{[1]})} = \frac{\partial \mathcal{L}}{\partial W^{[1]}} = \frac{\partial \mathcal{L}}{\partial z^{[1]}}\frac{\partial z^{[1]}}{\partial W^{[1]}} = \underbrace{a^{[0]\,T}}_{(1,n^{[0]})}\ \underbrace{dz^{[1]}}_{(1,n^{[1]})}$$

$$db^{[1]} = \frac{\partial \mathcal{L}}{\partial b^{[1]}} = \frac{\partial \mathcal{L}}{\partial z^{[1]}}\frac{dz^{[1]}}{db^{[1]}} = dz^{[1]}$$

- where $\odot$ is the element-wise (also named Hadamard) product.

Neural Networks (1 hidden layer)

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

$\underbrace{\qquad\qquad}_{Layer[1]}$ $\underbrace{\qquad\qquad\qquad\qquad}_{Layer[2]}$

ⓑ Backward propagation (all m training examples)

- Layer 2

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

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

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

– – –

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

- Layer 1

$$\underbrace{d\underline{\underline{Z}}^{[1]}}_{(m,n^{[1]})} = \underbrace{d\underline{\underline{A}}^{[1]}}_{(m,n^{[1]})} \odot g^{[1]\prime}(\underbrace{\underline{\underline{Z}}^{[1]}}_{(m,n^{[1]})})$$

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

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

Neural Networks (1 hidden layer)

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

$\underbrace{\qquad\qquad}_{Layer[1]}$ $\underbrace{\qquad\qquad\qquad\qquad}_{Layer[2]}$

ⓤ Parameters update

- **Parameters update :**

$$\underline{\underline{W}}^{[l]} = \underline{\underline{W}}^{[l]} - \alpha \cdot dW^{[l]}$$
$$\underline{b}^{[l]} = \underline{b}^{[l]} - \alpha \cdot db^{[l]}$$

where α is the learning rate

Deep Neural Networks (> 2 hidden layers)

Deep Neural Networks (> 2 hidden layers)

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

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

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

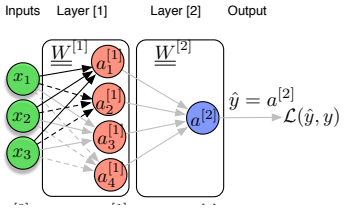
$$W^{[l]}, b^{[l]}$$

$$\downarrow$$

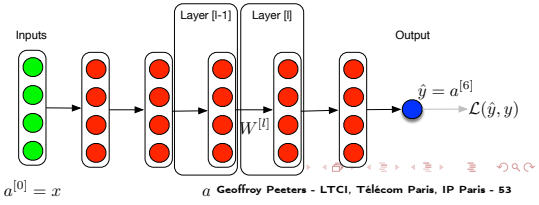
$$\boxed{a^{[0]} = x} \rightarrow \ldots \xrightarrow{a^{[l-1]}} \boxed{a^{[l-1]}\,W^{[l]} + b^{[l]}} \xrightarrow{z^{[l]}} \boxed{g^{[l]}(z^{[l]})} \xrightarrow{a^{[l]}} \ldots \rightarrow \boxed{\mathcal{L}(\hat{y} = a^{[L]}, y)}$$

$$\boxed{a^{[0]} = x} \leftarrow \ldots \xleftarrow[da^{[l-1]}]{} \underbrace{\boxed{a^{[l-1]}\,W^{[l]} + b^{[l]}} \xleftarrow[dz^{[l]}]{} \boxed{g^{[l]}(z^{[l]})}}_{Layer[l]} \xleftarrow[da^{[l]}]{} \ldots \leftarrow \boxed{\mathcal{L}(\hat{y} = a^{[L]}, y)}$$

Neural Network



Deep Neural Networks



Deep Neural Networks (> 2 hidden layers)

$$W^{[l]}, b^{[l]}$$

$$\downarrow$$

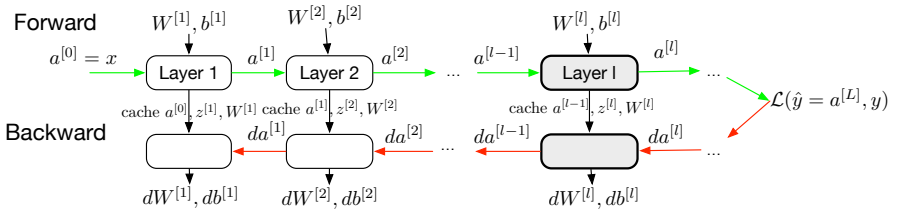
$$\boxed{a^{[0]} = x} \rightarrow \ldots \xrightarrow{a^{[l-1]}} \boxed{a^{[l-1]}\,W^{[l]} + b^{[l]}} \xrightarrow{z^{[l]}} \boxed{g^{[l]}(z^{[l]})} \xrightarrow{a^{[l]}} \ldots \rightarrow \boxed{\mathcal{L}(\hat{y} = a^{[L]}, y)}$$

$$\boxed{a^{[0]} = x} \leftarrow \ldots \xleftarrow[da^{[l-1]}]{} \underbrace{\boxed{a^{[l-1]}\,W^{[l]} + b^{[l]}} \xleftarrow[dz^{[l]}]{} \boxed{g^{[l]}(z^{[l]})}}_{Layer[l]} \xleftarrow[da^{[l]}]{} \ldots \leftarrow \boxed{\mathcal{L}(\hat{y} = a^{[L]}, y)}$$

ⓕ Forward (each layer, all m training examples)

$$\begin{aligned}
\underline{\underline{A}}^{[0]} &= \underline{\underline{X}} \\
\underline{\underline{Z}}^{[1]} &= \underline{\underline{A}}^{[0]}\,\underline{\underline{W}}^{[1]} + \underline{b}^{[1]} \\
\underline{\underline{A}}^{[1]} &= g^{[1]}(\underline{\underline{Z}}^{[1]}) \\
\underline{\underline{Z}}^{[2]} &= \underline{\underline{A}}^{[1]}\,\underline{\underline{W}}^{[2]} + \underline{b}^{[2]} \\
\underline{\underline{A}}^{[2]} &= g^{[2]}(\underline{\underline{Z}}^{[2]}) \\
&\ldots \\
\underline{\underline{\hat{Y}}} = \underline{\underline{A}}^{[L]} &= g^{[L]}(\underline{\underline{Z}}^{[L]})
\end{aligned}$$

Deep Neural Networks (> 2 hidden layers)



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

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

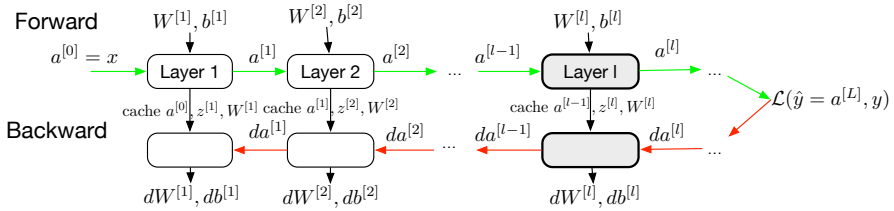
$$\underbrace{\underline{\underline{Z}}^{[l]}}_{(m,n^{[l]})} = \underbrace{\underline{\underline{A}}^{[l-1]}}_{(m,n^{[l-1]})}\,\underbrace{\underline{\underline{W}}^{[l]}}_{(n^{[l-1]},n^{[l]})} + \underbrace{\underline{b}^{[l]}}_{(1,n^{[l]})}$$

$$\underbrace{\underline{\underline{A}}^{[l]}}_{(m,n^{[l]})} = g^{[l]}(\underline{\underline{Z}}^{[l]})$$

Output : $\underline{\underline{A}}^{[l]}$

Cache for Backprop : $\underline{\underline{A}}^{[l-1]}$, $\underline{\underline{Z}}^{[l]}$, $\underline{\underline{W}}^{[l]}$

Deep Neural Networks (> 2 hidden layers)



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

Input : $d\underline{\underline{A}}^{[l]}$

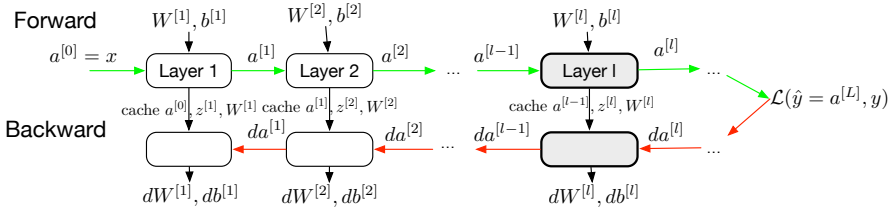
Cache (passed from Forward) : $\underline{\underline{A}}^{[l-1]}$ $\underline{\underline{Z}}^{[l]}$, $\underline{\underline{W}}^{[l]}$

$$
\begin{aligned}
d\underline{\underline{Z}}^{[l]} &= d\underline{\underline{A}}^{[l]} \odot g^{[l]\prime}(\underline{\underline{Z}}^{[l]}) \\
&= \left(d\underline{\underline{Z}}^{[l+1]}\ \underline{\underline{W}}^{[l+1]\,T}\right) \odot g^{[l]\prime}(\underline{\underline{Z}}^{[l]}) \\
d\underline{\underline{W}}^{[l]} &= \frac{1}{m}\underline{\underline{A}}^{[l-1]\,T}\ d\underline{\underline{Z}}^{[l]} \\
d\underline{b}^{[l]} &= \frac{1}{m}\sum_{i=1}^{m} d\underline{\underline{Z}}^{[l]}
\end{aligned}
\qquad\qquad
d\underline{\underline{A}}^{[l-1]} = d\underline{\underline{Z}}^{[l]}\ \underline{\underline{W}}^{[l]\,T}
$$

Output : $dA^{[l-1]}$

Output for parameters update : $dW^{[l]}$, $db^{[l]}$

Deep Neural Networks (> 2 hidden layers)



ⓤ Parameters update

$$
\begin{aligned}
\underline{\underline{W}}^{[l]} &= \underline{\underline{W}}^{[l]} - \alpha d\underline{\underline{W}}^{[l]} \\
\underline{b}^{[l]} &= \underline{b}^{[l]} - \alpha d\underline{b}^{[l]}
\end{aligned}
$$

where α is the learning rate

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

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

  - 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 :

$$
\begin{aligned}
h(x) &= f(x)g(x) \\
\frac{\partial h}{\partial x} &= \frac{\partial h}{\partial f}\frac{\partial f}{\partial x} + \frac{\partial h}{\partial g}\frac{\partial g}{\partial x} = f'g + fg'
\end{aligned}
$$

- **What is back-propagation ?**
  - an efficient algorithm to compute the chain-rule by storing intermediate (and re-use derivatives)

Ex 1 : Logisitic regression / least square (single output)

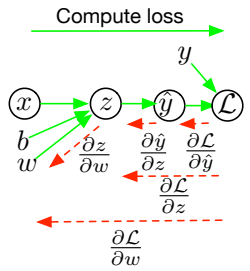
$$z = xw + b$$
$$\hat{y} = a = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(\hat{y} - y)^2$$

- **Computing derivative as in calculus class**

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

Ex 1 : Logisitic regression / least square (single output)

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



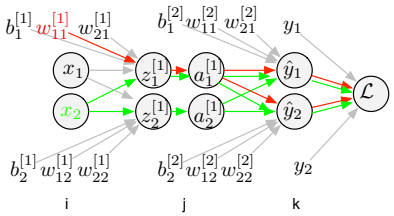
- **Computing derivative using back-propagation**

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

- 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

Ex 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

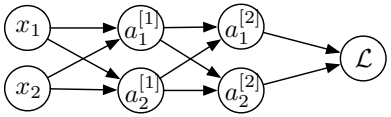
$$\frac{\partial \mathcal{L}}{\partial \hat{y}_k} = \hat{y}_k - y_k$$
$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{[2]}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_k} a_j^{[1]} \qquad \frac{\partial \mathcal{L}}{\partial b_k^{[2]}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_k}$$
$$\frac{\partial \mathcal{L}}{\partial a_j^{[1]}} = \sum_k \frac{\partial \mathcal{L}}{\partial \hat{y}_k} w_{jk}^{[2]}$$
$$\frac{\partial \mathcal{L}}{\partial z_j^{[1]}} = \frac{\partial \mathcal{L}}{\partial a_j^{[1]}}\sigma'(z_j)$$
$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{[1]}} = \frac{\partial \mathcal{L}}{\partial z_j^{[1]}} x_i \qquad \frac{\partial \mathcal{L}}{\partial b_j^{[1]}} = \frac{\partial \mathcal{L}}{\partial z_j^{[1]}}$$

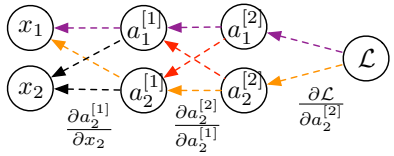
Ex 3 : MLP / least square (2 hidden layers, multiple outputs)

$$a_1^{[1]} = f_1^{[1]}(x_1, x_2)$$
$$a_2^{[1]} = f_2^{[1]}(x_1, x_2)$$
$$a_1^{[2]} = g_1^{[2]}(a_1^{[1]}, a_2^{[1]})$$
$$a_2^{[2]} = g_2^{[2]}(a_1^{[1]}, a_2^{[1]})$$
$$\mathcal{L} = h(a_1^{[2]}, a_2^{[2]})$$

$$\frac{\partial \mathcal{L}}{\partial x_1} = \frac{\partial \mathcal{L}}{\partial a_1^{[2]}}\underbrace{\frac{\partial a_1^{[2]}}{\partial x_1}}_{\frac{\partial a_1^{[2]}}{\partial a_1^{[1]}}\frac{\partial a_1^{[1]}}{\partial x_1} + \frac{\partial a_1^{[2]}}{\partial a_2^{[1]}}\frac{\partial a_2^{[1]}}{\partial x_1}} + \frac{\partial \mathcal{L}}{\partial a_2^{[2]}}\underbrace{\frac{\partial a_2^{[2]}}{\partial x_1}}_{\frac{\partial a_2^{[2]}}{\partial a_1^{[1]}}\frac{\partial a_1^{[1]}}{\partial x_1} + \frac{\partial a_2^{[2]}}{\partial a_2^{[1]}}\frac{\partial a_2^{[1]}}{\partial x_1}}$$

$$\frac{\partial \mathcal{L}}{\partial x_1} = \frac{\partial \mathcal{L}}{\partial a_1^{[2]}}\frac{\partial a_1^{[2]}}{\partial a_1^{[1]}}\frac{\partial a_1^{[1]}}{\partial x_1} + \frac{\partial p\mathcal{L}}{\partial a_1^{[2]}}\frac{\partial a_1^{[2]}}{\partial a_2^{[1]}}\frac{\partial a_2^{[1]}}{\partial x_1} + \frac{\partial \mathcal{L}}{\partial a_2^{[2]}}\frac{\partial a_2^{[2]}}{\partial a_1^{[1]}}\frac{\partial a_1^{[1]}}{\partial x_1} + \frac{\partial \mathcal{L}}{\partial a_2^{[2]}}\frac{\partial a_2^{[2]}}{\partial a_2^{[1]}}$$





- Each edge is assigned to derivative of origin w.r.t. destination

Computation Graph

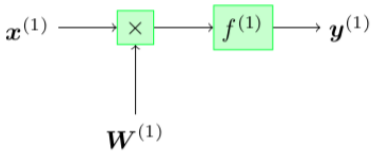
Computation Graph

(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

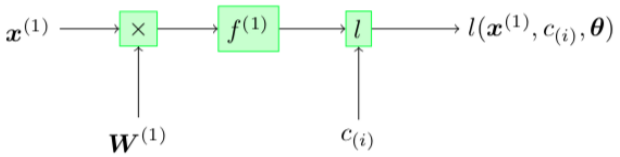
Example 1 : A single layer network



- 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

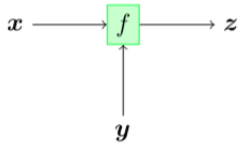


- A variable node encodes the label
- To compute the output for a given input
  $\rightarrow$ forward pass
- To compute the gradient of the loss *wrt* the parameters ($\boldsymbol{W}^{(1)}$)
  $\rightarrow$ backward pass

Computation Graph

A function node

Forward pass

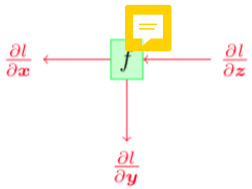


This node implements :

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

A function node - 2

Backward pass



A function node knows :

- the "local gradients" computation

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

- how to return the gradient to the inputs :

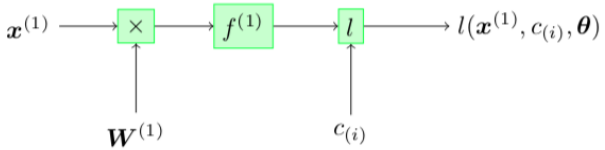
$$(\frac{\partial l}{\partial z}\frac{\partial z}{\partial x}), (\frac{\partial l}{\partial z}\frac{\partial z}{\partial y})$$

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

Example of a single layer network



Forward

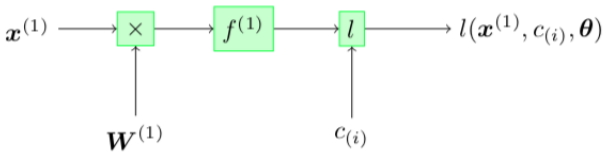
For each function node in topological order

- forward propagation

Which means :

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

Example of a single layer network



Backward

For each function node in reversed topological order

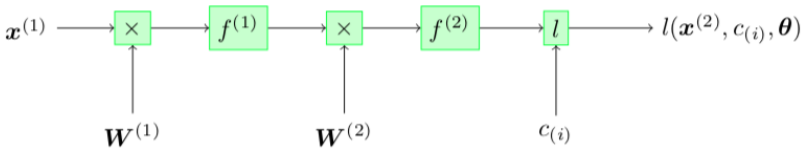
- backward propagation

Which means :

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

Computation Graph

Example of a two layers network

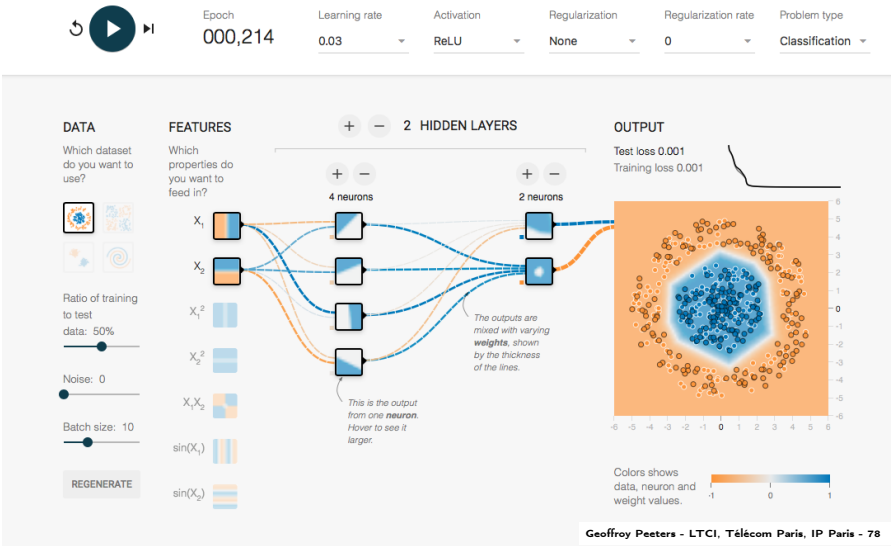


- The algorithms remain the same,
- even for more complex architectures
- Generalization by coding your own function node or by
- Wrapping a layer in a module

Deep Learning Frameworks

Deep Learning Frameworks

`https://playground.tensorflow.org/`

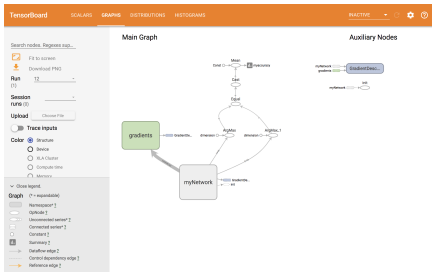


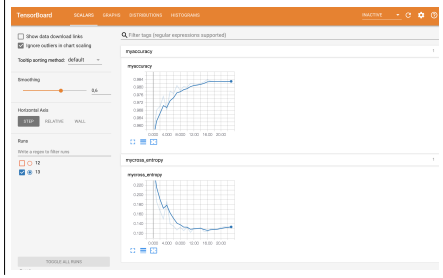
Deep Learning Frameworks

- **Python**
  - coding forward, loss, backward, upgrading
    - painfull, proned to coding errors
- **DL frameworks**
  - Advantages
    - CPU/GPU
    - Automatic differentiation based on computational graph
  - **(py)Torch**
    - The Facebook library with Lua/python API
    - https ://pytorch.org
  - **Tensorflow**
    - The Google library with python API
    - https ://www.tensorflow.org
  - **Keras**
    - A high-level API, in Python, running on top of TensorFlow
    - https ://keras.io

Deep Learning Frameworks

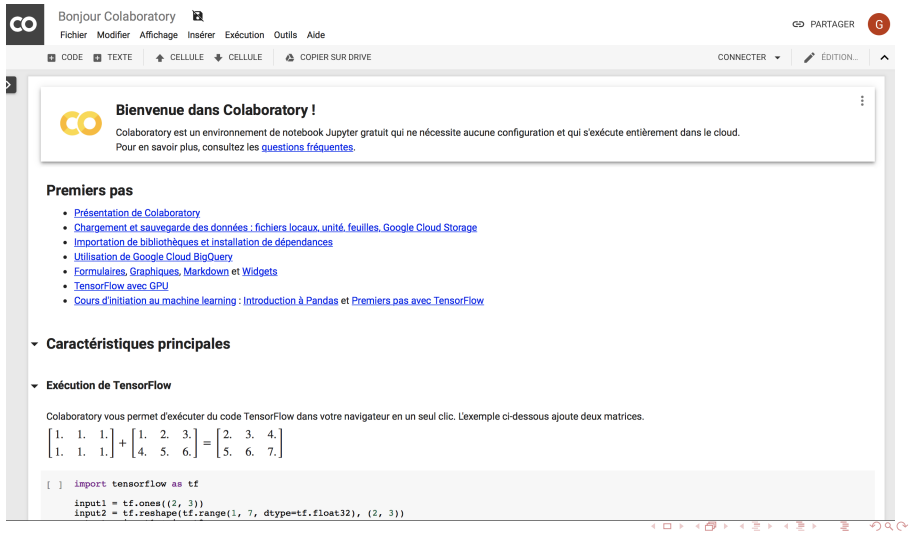
Tensorboard





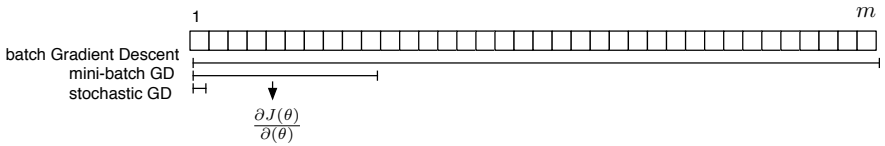
Deep Learning Frameworks

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



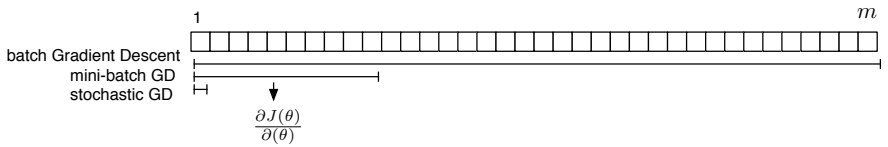
Various types of training

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)

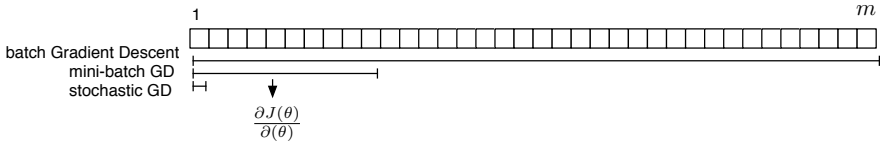
Various types of training



Batch Gradient Descent

Gradient computation $\frac{\partial J(\theta)}{\partial \theta}$	average of the gradients $\frac{\partial \mathcal{L}(\theta)^{(i)}}{\partial \theta}$ over **all** m training examples
Parameters update	after the m training examples went to forward
Pros	the gradient is accurate
Cons	can be very costly (if $m = 5.000.000$, we need to do m forward pass before doing any parameter update, can be very slow)

Various types of training



Mini Batch Gradient Descent

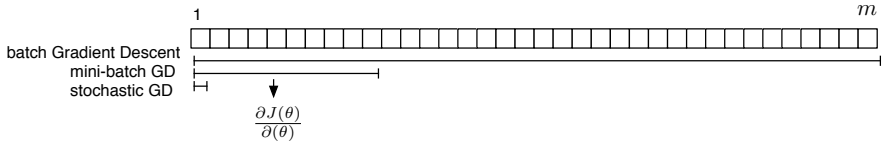
Gradient computation $\frac{\partial J(\theta)}{\partial \theta}$	average of the gradients $\frac{\partial \mathcal{L}(\theta)^{(i)}}{\partial \theta}$ over each **mini-batch**
Parameters update	after after each mini-batch
Pros	allows to do more steps of gradient-descent
Cons	gradient is still an approximation

- **Mini-batch ?**
  - Split training set into small training sets : $X^{\{t\}}, Y^{\{t\}}$

$$X = [\underbrace{|X^{(1)} \cdots X^{(1000)}|}_{(1,n_x)}, |X^{(2001)} \cdots X^{(3000)}| \cdots X^{(5.000.000)}] = [\underbrace{X^{\{1\}}}_{(1000,n_x)} \underbrace{X^{\{2\}}}_{(1000,n_x)} \cdots \underbrace{X^{\{5000\}}}_{(1000,n_x)}]$$

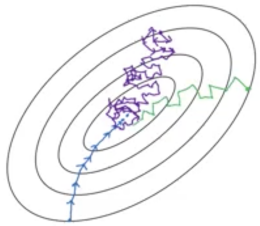
- **Pseudo-code**
  - for t=1, ..., 5000 :
    - ⓕ Forward propagation on $X^{\{t\}}$: $Z^{[1]} = W^{[1]}X^{\{t\}} + b^{[1]}$, $A^{[1]} = g^{[1]}(Z^{[1]})$, ...
    - ⓙ Compute cost : $J^{\{t\}} = \frac{1}{1000}\sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})) + \frac{\lambda}{2 \cdot 1000}\sum_{l} ||W^{[l]}||_F^2$
    - ⓑ Backpropagation to compute gradients of $J^{\{t\}}$ w.r.t the parameters
    - ⓤ Update : $W^{[l]} = W^{[l]} - \alpha \frac{\partial J^{\{t\}}}{\partial W^{[l]}}$, $b^{[l]} = b^{[l]} - \alpha \frac{\partial J^{\{t\}}}{\partial b^{[l]}}$
  - One pass over the whole training set= one **"epoch"**

Various types of training



Stochastic Gradient Descent (SGD)

Gradient computation $\frac{\partial J(\theta)}{\partial \theta}$	use directly the gradient **on a single training examples** i (same as mini-batch but with a mini-batch size of 1)
Parameters update	after each training examples
Pros	allows to do a lot of gradient-descent steps
Cons	can lead to a very noisy gradient descent + loses speed up from vectorization



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 $\underline{\underline{W}}$ (weight matrix) or $\underline{b}$ (bias)
  - gradient of the loss with respect to θ
    - $\frac{\partial \mathcal{L}(\theta,x,y)}{\partial \theta}$
- **Mini-batch Gradient Descent**

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

- **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

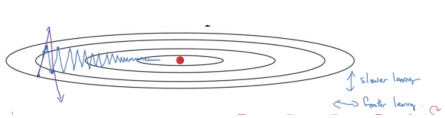
Momentum

- 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 $\frac{\partial \mathcal{L}(\theta(t-1),x,y)}{\partial \theta}$ on current mini-batch

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

  - 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 velocity
  - $\frac{\partial \mathcal{L}}{\partial \theta}$ plays the role of acceleration



Alternatives to gradient descent

Nesterov Accelerated Gradient (NAG)

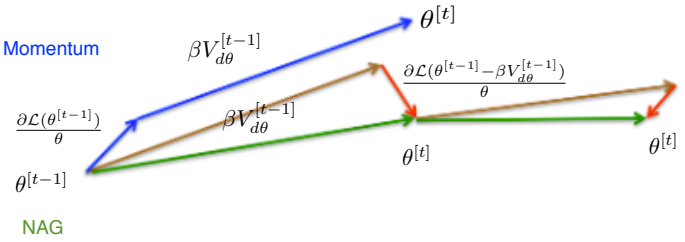
- Problem :
  - "A ball that rolls down a hill, blindly following the slope, is highly unsatisfactory."
- Solution :
  - "We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.""
  - We know that we will use our momentum term $\beta V_{d\theta}^{[t-1]}$ to move θ
    - $\theta^{[t]} = \theta^{[t-1]} - \alpha[\beta V_{d\theta}^{[t-1]} + (1-\beta)\frac{\partial \mathcal{L}}{\partial \theta}]$
  - 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 position of θ.
- **Nesterov Accelerated Gradient**
  - On 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 current gradient $\frac{\partial \mathcal{L}(\theta^{[t-1]})}{\partial \theta}$
  - 2) big jump in the direction of the previous accumulated gradient $\beta V_{d\theta}^{[t-1]}$
- **Nesterov Accelerated Gradient (NAG)**
  - 1) big jump in the direction of the previous accumulated gradient $\beta V_{d\theta}^{[t-1]}$
  - 2) measures the gradient $\frac{\partial \mathcal{L}(\theta^{[t-1]} - \alpha\beta V_{d\theta}^{[t-1]})}{\partial \theta}$
  - 3) makes a correction
- Prevents us from going too fast and results in increased responsiveness



AdaGrad

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

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

- **Adagrad :**

$$G_{i,i}^{[t]} = \sum_{\tau=0}^{t} d\theta_i^{[\tau]2}$$
$$\theta_i^{[t]} = \theta_i^{[t-1]} - \frac{\alpha}{\sqrt{G_{i,i}^{[t-1]} + \epsilon}} g_i^{[t-1]}$$

- Problem
  - the gradient are accumulated since the beginning
    - the learning rates will schrink

Adadelta

- Extension of Adagrad : instead of accumulating all past squared gradients (as in Adagrad) in Adadelta we restrict the window of accumulated past gradients to some fixed size
- **Adadelta**

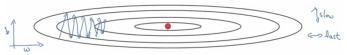
$$\mathbb{E}[d\theta^2]^{[t]} = \gamma \mathbb{E}[d\theta^2]^{[t-1]} + (1-\gamma) d\theta^{[t-1]2} \quad (2)$$
$$\theta^{[t]} = \theta^{[t-1]} - \frac{\alpha}{\sqrt{\mathbb{E}[g^2]^{[t]} + \epsilon}} d\theta^{[t-1]} \quad (3)$$

RMSprop (Root Mean Square prop)

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

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

- Want speed up in horizontal (W)
  - $S_{dW}^{[t]}$ small $\Rightarrow \frac{1}{\sqrt{S_{dW}^{[t]}+\epsilon}}$ large $\Rightarrow$ speed up
- Want slow down (damping) oscillation in vertical (b)
  - $S_{db}^{[t]}$ large $\Rightarrow \frac{1}{\sqrt{S_{db}^{[t]}+\epsilon}}$ small $\Rightarrow$ slow down



Adam (Adaptive moment estimation)

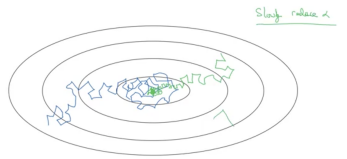
- Adam = Momentum with Adadelta/RMSprop
- **Adam**
  - On iteration t compute $d\theta$ on current mini-batch

$$V_{d\theta}^{[t]} = \beta_1 V_{d\theta}^{[t-1]} + (1-\beta_1) d\theta^{[t-1]}$$
$$S_{d\theta}^{[t]} = \beta_2 S_{d\theta}^{[t-1]} + (1-\beta_2) d\theta^{[t-1]2}$$
$$\theta^{[t]} = \theta^{[t-1]} - \alpha \frac{V_{d\theta}^{[t]}}{\sqrt{S_{d\theta}^{[t]} + \epsilon}}$$

- 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)

Learning rate

- Learning rate **decay**
  - Slowly reduce the learning rate α over time (simulated annealing)
    - Oscillating in a tighter region



- How to decay ?
  - $\alpha = \frac{1}{1+\text{decay\_rate}*\text{epoch\_num}} \alpha_0$
    - Example with $\alpha_0 = 0.2$ and decay_rate $= 1$

epoch	α
1	0.1
2	0.0666
3	0.05
4	0.04
5	0.0333

- Other strategies
  - exponential decay : $\alpha = 0.95^{\text{epoch\_num}} \alpha_0$
  - $\alpha = \frac{\text{some constant}}{\sqrt{\text{epoch\_num}}} \alpha_0$
  - $\alpha = \frac{\text{some constant}}{\sqrt{\text{minibatchnum}}} \alpha_0$
  - or decrease in discrete step (stair cases)
  - or manual decay (watch your model during training)

Loss functions

Loss functions

Loss function : Binary Cross-Entropy

- 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 logisitic regression, the output of the network $\hat{y}$ estimates p : $\hat{y} = P(Y=1|x,\theta)$
  $P(Y=y|x,\theta) = \hat{y}^y(1-\hat{y})^y \quad y \in \{0,1\}$
- We want to
  - find the θ that ... maximize the **likelihood** of the y given the input x
    $$\max_\theta P(Y=y|x,\theta) = \hat{y}^y(1-\hat{y})^y \quad y \in \{0,1\}$$
  - ... that maximize the **log-likelihood**
    $$\max_\theta \log p(y|x) = \log(\hat{y}^y\,(1-\hat{y})^{(1-y)}) = y\log(\hat{y}) + (1-y)\log(1-\hat{y}) = -\mathcal{L}(\hat{y},y)$$
  - ... that minimize the **binary cross-entropy** (minimize the loss)
    $$\min_\theta \boxed{\mathcal{L}(\hat{y},y) = -\left(y\log(\hat{y}) + (1-y)\log(1-\hat{y})\right)}$$
- ... maximizing **log-likelihood** on the whole training set
  $$p(\text{labels }) = \prod_{i=1}^{m} p(y^{(i)}|x^{(i)})$$
  $$\log p(\text{labels }) = \log\left(\prod_{i=1}^{m} p(y^{(i)}|x^{(i)})\right) = \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}) = \sum_{i=1}^{m} -\mathcal{L}(y^{(i)},y^{(i)})$$
  - ... is equivalent to minimizing the **Cost** $J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(y^{(i)}|x^{(i)})$

Activation functions $a = g(z)$

Activation functions $a = g(z)$

Output activation function : sigmoid/ logistic

- **Usage** : binary classification (0 or 1)
  - (logistic regression or deep neural network with binary output)
    $$\begin{cases} P(y=1|\underline{x}) = \sigma(\underline{x}^T\underline{w}) = \boxed{\dfrac{1}{1+e^{-(\underline{x}^T\underline{w})}}} \\ P(y=0|\underline{x}) = 1 - P(y=1|\underline{x}) \end{cases}$$
- **Models the log-odds** $\log\frac{p}{1-p}$ (posterior probability) of the value "1" using a linear model of the inputs $\underline{x}$
  $$P(y=1|\underline{x}) = \frac{1}{1+e^{-\underline{x}^T\underline{w}}}$$
  $$\frac{1}{P(y=1|\underline{x})} = 1 + e^{-\underline{x}^T\underline{w}}$$
  $$\frac{1-P(y=1|\underline{x})}{P(y=1|\underline{x})} = e^{-\underline{x}^T\underline{w}}$$
  $$\log\left(\frac{P(y=0|\underline{x})}{P(y=1|\underline{x})}\right) = -\underline{x}^T\underline{w}$$
  $$\log\left(\frac{P(y=1|\underline{x})}{P(y=0|\underline{x})}\right) = \underline{x}^T\underline{w}$$

Activation functions $a = g(z)$

Output activation function : softmax

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

$$P(y=1|\underline{x}) = \frac{e^{\underline{w_1}\cdot\underline{x}}}{\sum_{c=1}^{K} e^{\underline{w_c}\cdot\underline{x}}}$$

$$\cdots$$

$$P(y=o|\underline{x}) = \boxed{\frac{e^{\underline{w_o}\cdot\underline{x}}}{\sum_{c=1}^{K} e^{\underline{w_c}\cdot\underline{x}}}}$$

$$\cdots$$

$$P(y=K|\underline{x}) = \frac{e^{\underline{w_K}\cdot\underline{x}}}{\sum_{c=1}^{K} e^{\underline{w_c}\cdot\underline{x}}}$$

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

$$P(y=o|\underline{x}) = \frac{e^{(\underline{w_o}-\psi)\cdot\underline{x}}}{\sum_{c=1}^{K} e^{(\underline{w_c}-\psi)\cdot\underline{x}}} = \frac{e^{\underline{w_o}\cdot\underline{x}}\ e^{-\psi\cdot\underline{x}}}{\sum_{c=1}^{K} e^{\underline{w_c}\cdot\underline{x}}\ e^{-\psi\cdot\underline{x}}} = \frac{e^{\underline{w_o}\cdot\underline{x}}}{\sum_{c=1}^{K} e^{\underline{w_c}\cdot\underline{x}}}$$

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

$$P(y=o|\underline{x}) = \frac{e^{\underline{w_o}\underline{x}}}{1+\sum_{c=1}^{K-1} e^{\underline{w_c}\underline{x}}} \quad \forall o \in [1 \ldots K-1]$$

$$P(y=K|\underline{x}) = \frac{1}{1+\sum_{c=1}^{K-1} e^{\underline{w_c}\cdot\underline{x}}}$$

Activation functions $a = g(z)$

Output activation function : softmax

- We of course have
  $P(y=1|\underline{x}) + P(y=2|\underline{x}) + ... + P(y=K|\underline{x}) = 1$
- **Models the log-odds** $\log \frac{p_c}{p_K}$ (posterior probability) using linear models of the inputs $\underline{x}$

$$\log\left(\frac{P(y=1|\underline{x})}{P(y=K|\underline{x})}\right) = \underline{w}_1\underline{x}$$

$$\cdots$$

$$\log\left(\frac{P(y=o|\underline{x})}{P(y=K|\underline{x})}\right) = \underline{w}_o\underline{x}$$

$$\cdots$$

$$\log\left(\frac{P(y=K-1|\underline{x})}{P(y=K|\underline{x})}\right) = \underline{w}_{K-1}\underline{x}$$

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

$$P(y=1|\underline{x}) = \frac{e^{\underline{w_1}\underline{x}}}{e^{\underline{w_1}\underline{x}}+e^{\underline{w_2}\underline{x}}} = \frac{e^{(\underline{w_1}-\psi)\underline{x}}}{e^{(\underline{w_1}-\psi)\underline{x}}+e^{(\underline{w_2}-\psi)\underline{x}}} = \frac{1}{1+e^{(\underline{w_2}-\underline{w_1})\underline{x}}}$$

$$P(y=2|\underline{x}) = \frac{e^{\underline{w_2}\underline{x}}}{e^{\underline{w_1}\underline{x}}+e^{\underline{w_2}\underline{x}}} = \frac{e^{(\underline{w_2}-\psi)\underline{x}}}{e^{(\underline{w_1}-\psi)\underline{x}}+e^{(\underline{w_2}-\psi)\underline{x}}} = \frac{e^{(\underline{w_2}-\underline{w1})\underline{x}}}{1+e^{(\underline{w_2}-\underline{w1})\underline{x}}} = 1 - P(y=1|\underline{x})$$

Activation functions $a = g(z)$

Why non-linear activation functions ?

- Consider the following network

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

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

$$\underline{a}^{[1]} = \underline{z}^{[1]} = \underline{x}\ \underline{\underline{W}}^{[1]} + \underline{b}^{[1]}$$
$$\underline{a}^{[2]} = \underline{z}^{[2]} = \underline{a}^{[1]}\underline{\underline{W}}^{[2]} + \underline{b}^{[2]}$$
$$= \left(\underline{x}\ \underline{\underline{W}}^{[1]} + \underline{b}^{[1]}\right)\underline{\underline{W}}^{[2]} + \underline{b}^{[2]}$$
$$= \underline{x}\underline{\underline{W}}^{[1]}\underline{\underline{W}}^{[2]} + \underline{b}^{[1]}\underline{\underline{W}}^{[2]} + \underline{b}^{[2]}$$
$$= \underline{x}\ \underline{\underline{W}}' + b'$$

  - then the network **the network reduces to a simple linear function**

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

Activation functions $a = g(z)$

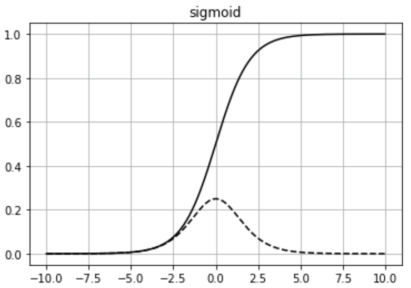
Sigmoid σ

- **Sigmoid function**

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

- **Derivative**

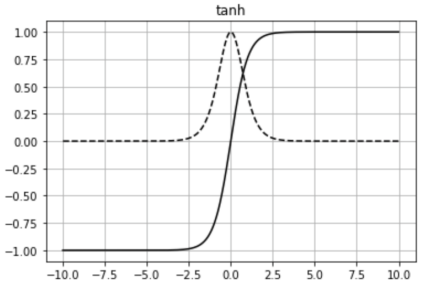
$$g'(z) = --e^{-z}\frac{1}{(1+e^{-z})^2}$$
$$= \frac{1+e^{-z}-1}{(1+e^{-z})^2}$$
$$= \frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right)$$
$$= \sigma(z)(1-\sigma(z))$$
$$g'(z) = a(1-a)$$



Activation functions $a = g(z)$

Hyperbolic tangent g

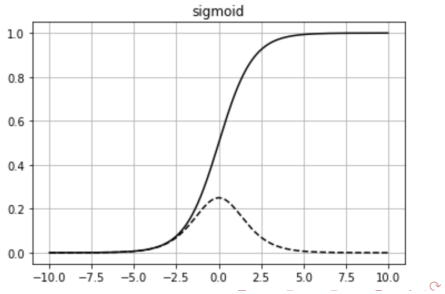
- **Hyperbolic tangent function**
  $a = g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
  - **Derivative**
    $g'(x) = 1 - (\tanh(z))^2$
    $g'(z) = 1 - a^2$
  - **Usage**
    - $\tanh(z)$ better than $\sigma(z)$ in middle hidden layers because its mean = zero ($a \in [-1, 1]$).
  - **Problem** 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 $a = g(z)$

Vanishing gradient

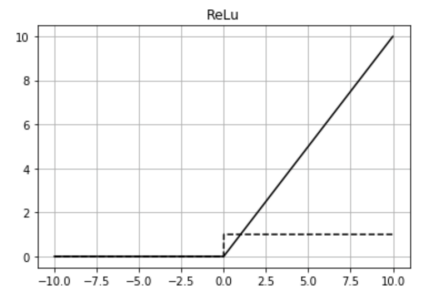
- Reminder :
  $dz^{[l]} = dz^{[l+1]} \underline{W^{[l+1]^T} \odot g^{[l]'}(z^{[l]})}$
  $db^{[l]} = \frac{1}{m} \sum_m dZ^{[l]}$
- Hence for a deep network (supposing $g^{[l]}(z) = \sigma(z)$)
  $\frac{\partial \mathcal{L}}{\partial b^{[1]}} = \frac{\partial \mathcal{L}}{\partial da^{[4]}} \sigma'(z^{[4]}) \underline{W^{[4]} \sigma'(z^{[3]})} \underline{W^{[3]} \sigma'(z^{[2]})} \underline{W^{[2]} \sigma'(z^{[1]})}$
- But $\max_z \sigma'(z) = \frac{1}{4}$!
  - Therefore, the deeper the network, the fastest the gradient diminishes/vanishes during backpropagation
    - Consequence ?
      - The network stop learning
- Solution ?
  - Use a ReLu activation



Activation functions $a = g(z)$

ReLU (Rectified Linear Unit)

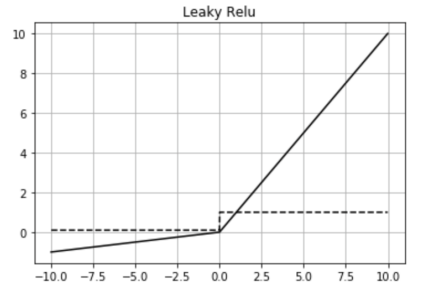
- **ReLU function**
  $a = g(z) = \max(0, z)$
  - **Derivative**
    $g'(x) = 1 \quad \text{if } z > 0$
    $= 0 \quad \text{if } z \leqslant 0$



Activation functions $a = g(z)$

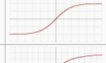
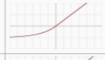
Variations of ReLU

- **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**
    $g'(x) = 1 \quad \text{if } z > 0$
    $= 0.01 \quad \text{if } z \leqslant 0$
- **PReLU function**
  $a = g(x) = \max(\alpha z, z)$
  - same as Leaky ReLU but α is a parameter to be learnt
  - **Derivative**
    $g'(x) = 1 \quad \text{if } z > 0$
    $= \alpha \quad \text{if } z \leqslant 0$
- **Softplus function**
  $g(x) = \log(1 + e^x)$
  - continuous appriximation 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.)



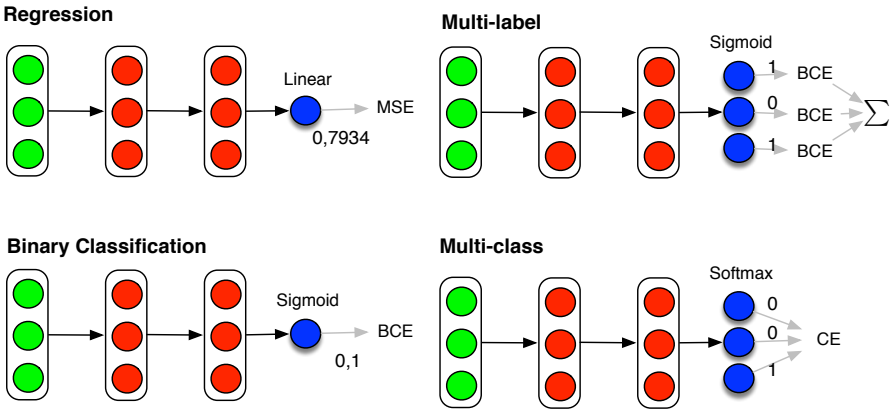
Activation functions $a = g(z)$

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		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{\cdot}$

Various types of problems

Various types of problems



Weight initialization

Weight initialization with zero ?

- **Initialize with zeros**

$$W^{[1]} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

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

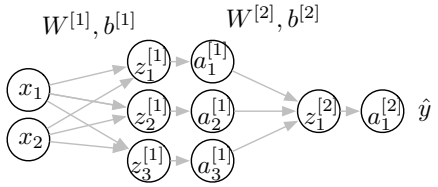
  - then for any example $a_1^{[1]} = a_2^{[1]} = a_3^{[1]}$
  - the two hidden units are computing exactly the same function
    - they are symetric

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

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

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

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



Weight initialization with zero ?

- Backpropagation

$$dz^{[2]} = a^{[2]} - y$$

$$dW^{[2]} = {a^{[1]}}^T dz^{[2]} = \begin{pmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{pmatrix} dz_1^{[2]}$$

$$da^{[1]} = dz^{[2]} {W^{[2]}}^T = dz^{[2]} \left(w_{11}^{[2]} w_{21}^{[2]} w_{31}^{[2]} \right)$$

$$dz^{[1]} = da^{[1]} \odot {g^{[1]}}'(z^{[1]}) = dz^{[2]} \left(w_{11}^{[2]} w_{21}^{[2]} w_{31}^{[2]} \right) \odot {g^{[1]}}' \left(z_1^{[1]} z_2^{[1]} z_3^{[1]} \right)$$

$$dW^{[1]} = {a^{[0]}}^T dz^{[1]} = \begin{pmatrix} a_1^{[0]} \\ a_2^{[0]} \end{pmatrix} \left(dz_1^{[1]} dz_2^{[1]} dz_3^{[1]} \right) = \begin{pmatrix} dz_1^{[1]} a_1^{[0]} & dz_2^{[1]} a_1^{[0]} & dz_3^{[1]} a_1^{[0]} \\ dz_1^{[1]} a_2^{[0]} & dz_2^{[1]} a_2^{[0]} & dz_3^{[1]} a_2^{[0]} \end{pmatrix} = \begin{pmatrix} u & u & u \\ v & v & v \end{pmatrix}$$

- When we compute backpropagation : $dz_1^{[1]} = dz_2^{[1]} = dz_3^{[1]}$
- Therefore dW is in the form

$$dW = \begin{pmatrix} u & u & u \\ v & v & v \end{pmatrix} \rightarrow W^{[1]} = W^{[1]} - \alpha dW$$

  - $\Rightarrow$ the update will keep the symetricity : $z_1^{[1]} = z_2^{[1]} = z_3^{[1]} = a_1^{[0]} u + a_2^{[0]} v$
- so this is not usefull since we want the different units to compute difference functions
  - $\Rightarrow$ initialize the parameters randomly

Weight initialization with random values

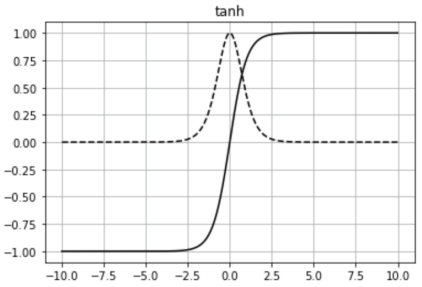
- **Random initialization**
  - $W^{[1]}$ = `np.random.randn((2,3))*0.01`
  - $b^{[1]}$ = `0`
  - $W^{[2]}$ = `np.random.randn((3,1))*0.01`
  - $b^{[1]}$ = `0`
- Remark : b doesn't have the symetry problem

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

$$Z^{[1]} = X\, W^{[1]} + b^{[1]} \quad (4)$$

$$A^{[1]} = g^{[1]}(Z^{[1]}) \quad (5)$$

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



Vanishing/ exploding gradients

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

$$y = \underbrace{\underbrace{X\, \underline{\underline{W}}^{[1]}}_{a^{[1]}=g(z^{[1]})=z^{[1]}} \underline{\underline{W}}^{[2]} \cdots \underline{\underline{W}}^{[L]}}_{a^{[2]}=g(z^{[2]})=z^{[2]}}$$

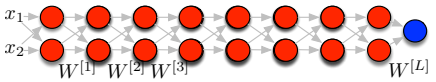
- **Exploding gradient**
  - suppose

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

  - then

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

  - 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

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

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

Weight initialization

Weight initialization 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/exploiding gradient) $\Rightarrow$
  - The larger n is $\Rightarrow$ the smallest w_i should be
- Solution ?
  - set $\mathrm{Var}(w_i) = \frac{1}{n}$
- In practice
  - $\underline{\underline{W}}^{[l]} =$ `np.random.randn(shape) * np.sqrt` $\left(\frac{1}{n^{[l-1]}}\right)$
- Other possibilities
  - For a ReLu
    - $\mathrm{Var}(w_i) = \frac{2}{n}$ works a bit better
  - For a tanh
    - $\mathrm{Var}(w_i) = \frac{1}{n^{[l-1]}}$: Xavier initialization
    - $\mathrm{Var}(w_i) = \frac{2}{n^{[l-1]} n^{[l]}}$: Bengio initialization

Regularization

Regularization

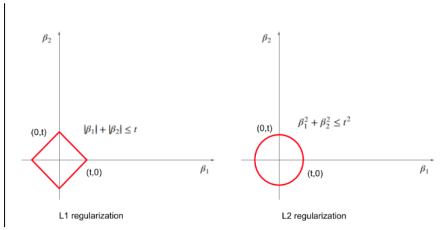
L1 and L2 regularization

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

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

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

- **L1 regularization** (Lasso) :
  - will end up with sparse $\underline{w}$ (many zero)
- **L2 regularization** (Ridge) :
  - will end up with small values of $\underline{w}$
- L1+L2 : ELastic Search
- λ is the regularization parameter (hyperparameter)



Regularization

L1 and L2 regularization

- In neural network

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

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

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

- In gradient descent ?

$$d'W^{[l]} = dW^{[l]} + \frac{\lambda}{m} W^{[l]}$$

Therefore

$$\begin{aligned} W^{[l]} &\leftarrow W^{[l]} - \alpha d'W^{[l]} \\ &\leftarrow W^{[l]} - \alpha(dW^{[l]} + \frac{\lambda}{m} W^{[l]}) \\ &\leftarrow W^{[l]} - \frac{\alpha\lambda}{m} W^{[l]} - \alpha dW^{[l]} \\ &\leftarrow W^{[l]} \underbrace{(1 - \frac{\alpha\lambda}{m})}_{\leq 1} - \alpha dW^{[l]} \end{aligned}$$

weight decay

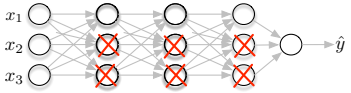
Regularization

Why regularization reduces overfitting ?

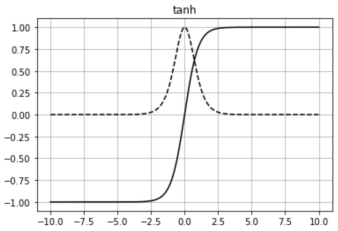
- **Intuition 1**

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

  - If we set λ very very big then $\underline{\underline{W}}^{[l]} \simeq 0$
    - $\Rightarrow$ many hidden units are not active
      - $\Rightarrow$ the network becomes much simpler $\Rightarrow$ avoid over-fitting



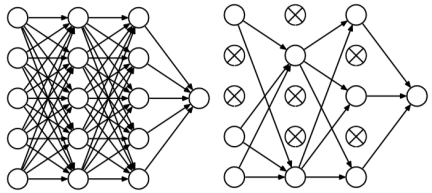
- **Intuition 2**
  - Suppose we are using a tanh
    - If z is small, the tanh is linear
  - If λ is large,
    - then $\underline{\underline{W}}^{[l]}$ is small,
    - then $z^{[l]}$ is small
    - then every layer is almost linear,
      - $\Rightarrow$ the whole network is linear



Regularization

DropOut regularization

- **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 $\underline{\underline{W}}$ $\Rightarrow$ p is higher
    - for small matrix $\underline{\underline{W}}$ $\Rightarrow$ 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 regularization scheme
  - since the network is smaller, it is easier to train (as regularization)
  - The network cannot rely on any feature, it has to spread out weights
    - Effect : shrinking the squared norm of the weights (similar to L2 regularization)
    - Can be shown to be an adaptive form of L2-regularization

Regularization

Data augmentation

Normalization

Normalization

Normalizing the inputs

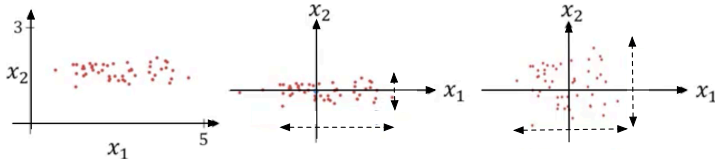
- **"Standardizing" :**
  - subtracting a measure of location and dividing by a measure of scale
  - subtract the mean and divide by the standard deviation

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

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

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

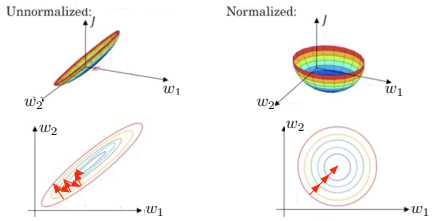
- We use also μ_{train} and σ^2_{train} to standardize the test set



Normalization

Normalizing the inputs

- Suppose : $x_1 \in [1 \dots 1000]$ and $x_2 \in [0 \dots 1]$
  - Then w_1 and w_2 will take very different value
- if no normalization
  - many oscillations
  - need to use a small learning rate
- **Why use input normalization ?**
  - 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



Normalization

Batch Normalization (BN)

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

- **Objective ?**
  - Apply the same normalization 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 !
    - **normalize the output of the previous layer** $a^{[l-1]}$
    - in practice normalize 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 $g^{[l]}$

Normalization

Batch Normalization (BN)

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

- **Normalization**

$$\mu^{[l]} = \frac{1}{m}\sum_i z^{[l](i)}$$

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

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

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

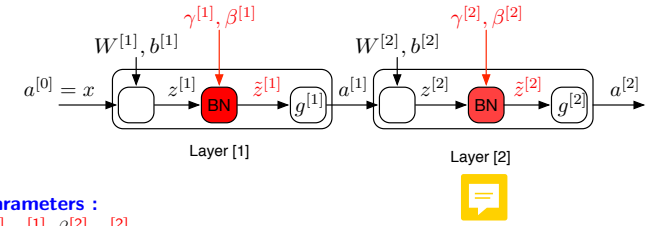
$$\tilde{z}^{[l](i)} = \gamma \cdot z^{[l](i)}_{norm} + \beta$$

- Non-linearity

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

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

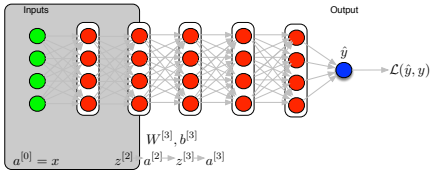
Batch Normalization (BN)

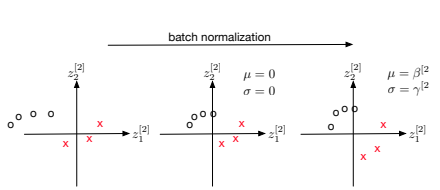


- **New parameters :**
  - $\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \ldots$
- How to estimate $\beta^{[l]}$, $\gamma^{[l]}$?
  - $\beta^{[l]}$, $\gamma^{[l]}$ are estimated using **gradient-descent**
    - **Gradient** : $\frac{\partial \mathcal{L}}{\partial \beta^{[l]}}$, $\frac{\partial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

Batch Normalization (BN)

- **Why does it work ?**





- We consider the left part as the input ($\underline{a}^{[2]}$) of the current layer $l = 3$
- **Problem :**
  - when training $W^{[3]}, b^{[3]}$, inputs $\underline{a}^{[2]}$ change over time (since previous layers weights also change)
- **Solution :** BN allows to reduce these changes
  - BN ensures that the mean and variance will remain the same (β and γ)
    - the values become more stable
    - it reduces the coupling between the layers
    - it speed up training
- **Regularization effect**
  - each mini-batch is scaled using μ and σ computed only on $X^{\{t\}}$
  - it adds some noise to the values $z^{[l]}$ within that mini-batch
    - // to drop-out which add noise to each hidden layer activation
  - to obtain regularization effect, using small mini-batch (64) is better than large ones (512)