MSBigData
Part: Neural Networks

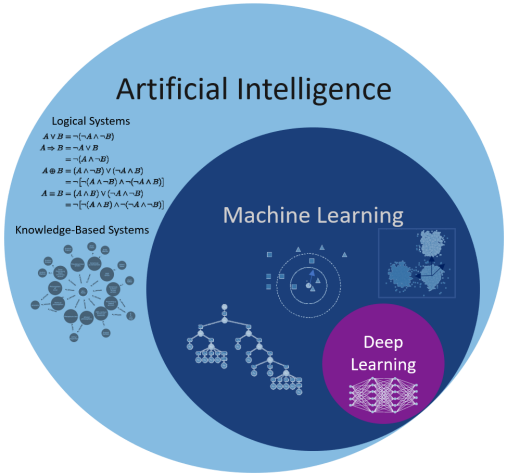
Geoffroy Peeters

LTCI - Télécom ParisTech

2018

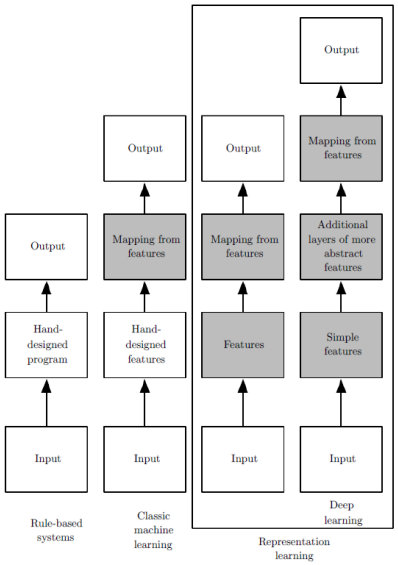
Deep Learning and Neural Networks : History

Deep learning (a subset of machine learning)



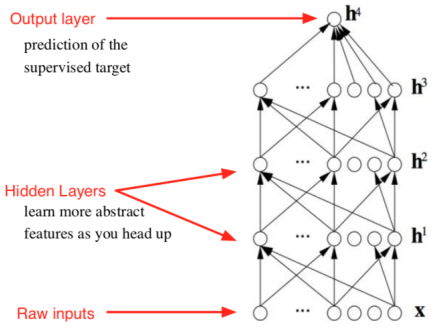
From http://canvart.club/atmose-11_21_13.html

Deep learning : learning hierachical representations

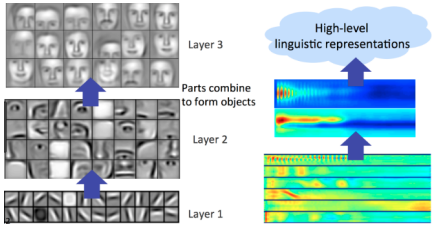


From Deep Learning by Ian Goodfellow and Yoshua Bengio and Aaron Courville

Deep learning : learning hierachical representations

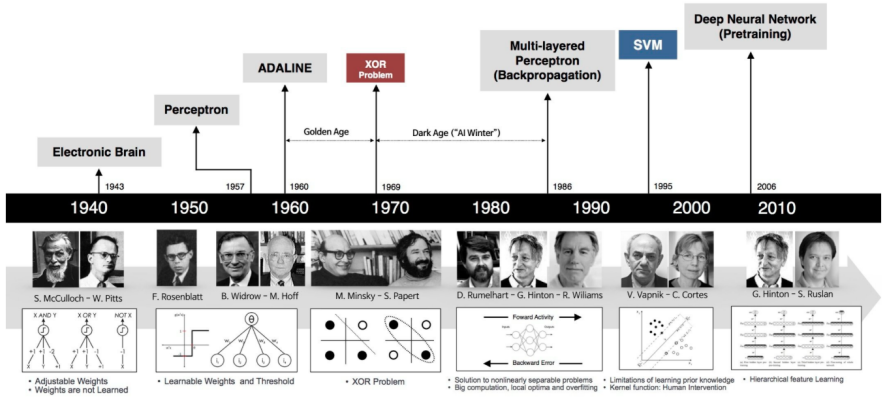


From Bengio 2015



From Bengio 2015

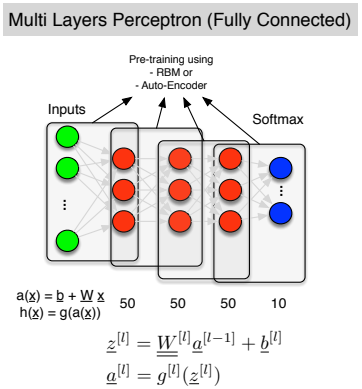
Deep Learning and Neural Networks : History



Three main types of Nets

Three main types of Nets

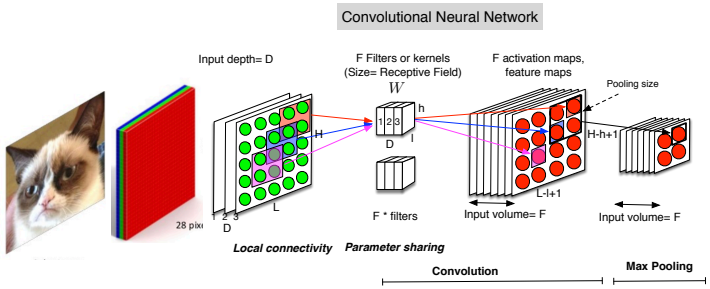
Multi Layers Perceptron (MLP) or Fully-Connected (FC)



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

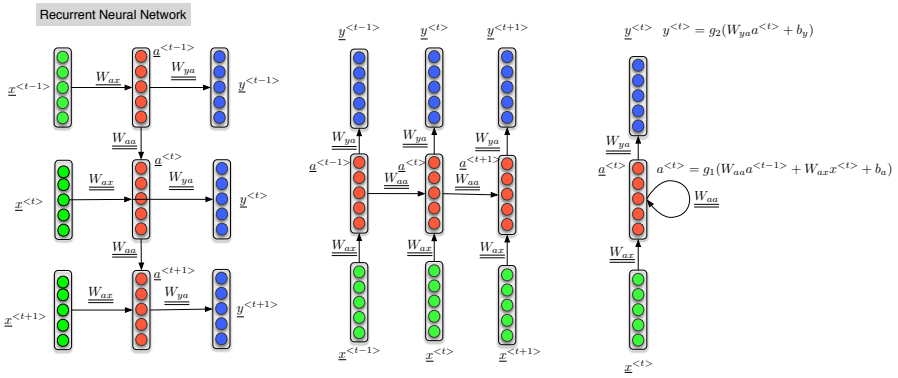
Three main types of Nets

Convolutional Neural Networks (CNN)



Three main types of Nets

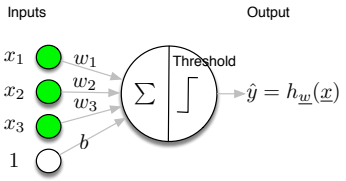
Recurrent Neural Networks (RNN)



Perceptron

Perceptron

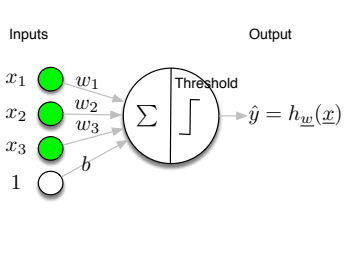
- **Perceptron ?**
  - A binary **classification algorithm** :
    - output is 0 or 1
  - Models the decision using a linear function, followed by a threshold
    $$\hat{y} = h_{\underline{w}}(\underline{x}) = \text{Threshold}(\underline{w} \cdot \underline{x}) \quad (1)$$
  - where
    - $\text{Threshold}(z)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



Perceptron

Training

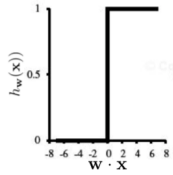
- **Training ?**
  - Adapt $\underline{w}$ and b so that $\hat{y} = h_{\underline{w}}(\underline{x})$ gives the correct answer y on training data
- **Algorithm :**
  - for each pair $(\underline{x}^{(i)}, y^{(i)})$
    - compute $h_{\underline{w}}(\underline{x}^{(i)}) = \text{Threshold}(\underline{w} \cdot \underline{x}^{(i)})$
    - 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

Formulating the training problem as an optimization problem

- In the case of the **Perceptron**, we want to **minimize the loss**
  $$\mathcal{L}(y^{(i)}, h_{\underline{w}}(x^{(i)})) = -(y^{(i)} - h_{\underline{w}}(\underline{x}^{(i)}))\underline{w} \cdot \underline{x}^{(i)}$$
  - if the predicition is good
    - $\Rightarrow$ the cost is 0
  - if the prediction is wrong
    - $\Rightarrow$ the cost is the distance between $\underline{w} \cdot x^{(i)}$ and the necessary threshold for the prediction to be good
- 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

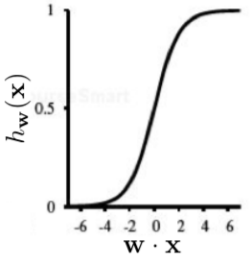


- Gradient descent ?
  $$w_d \leftarrow w_d - \alpha \frac{\partial \mathcal{L}(y^{(i)}, h_{\underline{w}}(x^{(i)}))}{\partial w_d} \quad \forall d$$
- Gradient ?
  $$\frac{\partial \mathcal{L}(y^{(i)}, h_{\underline{w}}(x^{(i)}))}{\partial w_d} = -\left(\frac{\partial \left(y^{(i)} - h_{\underline{w}}(x^{(i)})\right)}{\partial \underline{w}_d} \underline{w} \cdot \underline{x}^{(i)} + \left(y^{(i)} - h_{\underline{w}}(x^{(i)})\right) \cdot x_d^{(i)}\right)$$
  $$\cong -\left(0 + \left(y^{(i)} - h_{\underline{w}}(x^{(i)})\right) \cdot x_d^{(i)}\right)$$
  $$= -(y^{(i)} - h_{\underline{w}}(x^{(i)})) \cdot x_d^{(i)}$$

Perceptron

Making things correct

- In **Logisitic Regression** instead of predicting a class, we will **predict a probability** to belong to this class
  $$p(y = 1|\underline{x}) = h_{\underline{w}}(\underline{x}) = \text{Logistic}(\underline{w} \cdot \underline{x}) = \frac{1}{1 + e^{-\underline{w} \cdot \underline{x}}}$$
- We choose the class (0 or 1) 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

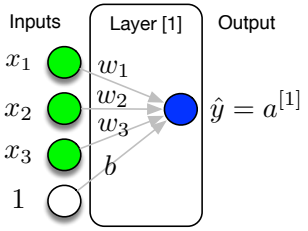


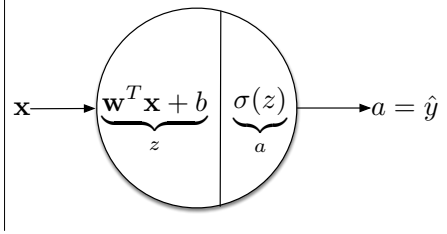
- We define the Loss as (binary cross-entropy)
  $$\mathcal{L}(y^{(i)}, h_{\underline{w}}(x^{(i)})) = -y^{(i)} \log(h_{\underline{w}}(\underline{x}^{(i)})) - (1 - y^{(i)}) \log(1 - h_{\underline{w}}(\underline{x}^{(i)}))$$
- Gradient descent ?
  $$w_d \leftarrow w_d - \alpha \frac{\partial \mathcal{L}(y^{(i)}, h_{\underline{w}}(x^{(i)}))}{\partial w_d} \quad \forall d$$
- 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)

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

- Problem :
  - Given $\underline{x} \in \mathbb{R}^{n_x}$, we want to predict $\hat{y} = P(y=1|\underline{x})$ with $0 \leqslant y \leqslant 1$
- Model :
  - $\hat{y} = \sigma(\underline{w}^T\underline{x} + b)$
  - 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 $\sigma(z) = 0$





Cost Function (Empirical risk minimization (ERM))

- **Training :**
  - Given $\{(\underline{x}^{(1)}, y^{(1)}), (\underline{x}^{(2)}, y^{(2)}), \ldots, (\underline{x}^{(m)}, y^{(m)})\}$,
  - We want $\hat{y}^{(i)} \simeq y^{(i)}$
- How to measure ? **Loss $\mathcal{L}$ (error) function**
  - Mean Square Error (MSE)
    - $\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ $\Rightarrow$ not convex, many local minima
  - Binary Cross-Entropy
    - $\mathcal{L}(\hat{y}, y) = -\left(y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right)$
      - If $y = 1$ $\Rightarrow \mathcal{L}(\hat{y}, y) = -\log(\hat{y})$ $\Rightarrow$ want $\log(\hat{y})$ large $\Rightarrow$ want $\hat{y}$ large
      - If $y = 0$ $\Rightarrow \mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$ $\Rightarrow$ want $\log(1-\hat{y})$ large $\Rightarrow$ want $\hat{y}$ small
- For all training example : **Cost J function**

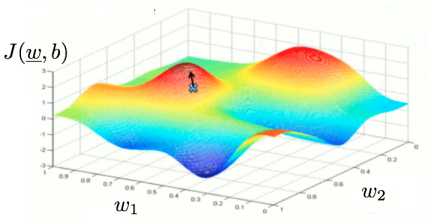
$$J(\underline{w}, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

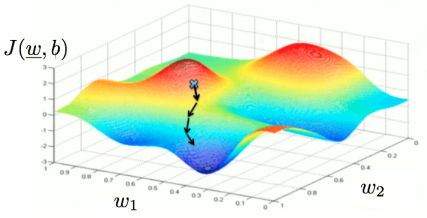
$$= -\frac{1}{m}\sum_{i=1}^{m} \left(y^{(i)} \log(\hat{y}^{(i)}) + (1-y^{(i)}) \log(1-\hat{y}^{(i)})\right)$$

- We want to find the values $\underline{w}$ and b that minimize $J(\underline{w}, b)$

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
    - $\underline{w} \leftarrow \underline{w} - \alpha \cdot d\underline{w}$
    - $b \leftarrow b - \alpha \cdot db$
  - where α is the "learning rate"
- Notation :
  - we will use $d\underline{w}$ for $\frac{\partial J(\underline{w},b)}{\partial \underline{w}}$
  - we will use db for $\frac{\partial J(\underline{w},b)}{\partial b}$





Logistic Regression (0 hidden layers)

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

Forward propagation

$$z = \underline{w}^T \underline{x} + b$$
$$\hat{y} = a = \sigma(z)$$

with $\sigma(z) = \frac{1}{1+e^{-z}}$.

Logistic Regression (0 hidden layers)

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

Backward propagation

$$da = \frac{\partial \mathcal{L}}{\partial a} = \frac{-y}{a} + \frac{(1-y)}{(1-a)}$$
$$dz = \frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a}\frac{da}{dz} = \cdots = a - y$$
$$dw_1 = \frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial w_1} = x_1 \cdot dz$$
$$dw_2 = \frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial w_2} = x_2 \cdot dz$$
$$db = \frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial b} = dz$$

Logistic Regression (0 hidden layers)

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

Parameters update

$$w_1 := w_1 - \alpha dw_1$$
$$w_2 := w_2 - \alpha dw_2$$
$$b := b - \alpha db$$

where α is the learning rate

Logistic Regression (0 hidden layers)

Computing the gradient descent on m training examples

- For one training example i :
  $\hat{y}^{(i)} = a^{(i)} = \sigma(z^{(i)}) = \sigma(\underline{w}^T \underline{x}^{(i)} + b)$
- Cost (sum of the Loss $\mathcal{L}$ over all training examples m)
  $$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$
- Minimizing the Cost $\Rightarrow$ derivative of the Cost w.r.t. the parameters
  $$\frac{\partial J(w,b)}{\partial w_1} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}{\partial w_1}$$
  $$= \frac{1}{m}\sum_{i=1}^{m} dw_1^{(i)}$$
  - $\frac{\partial J(w,b)}{\partial w_1}$ is the average over the m training examples of the gradients $dw_1^{(i)}$

Logistic Regression (0 hidden layers)

Equivalence between minimizing the Cost and and maximum likelihood

- We interpret $\hat{y}$ as $p(y=1|x)$

$$\begin{aligned}
&\text{If } y=1 \rightarrow p(y|x) = \hat{y}\\
&\text{If } y=0 \rightarrow p(y|x) = 1-\hat{y}\\
&p(y|x) = \hat{y}^y \cdot (1-\hat{y})^{(1-y)}\\
\text{If } y=1 \quad &= \hat{y}^1 \cdot (1-\hat{y})^0 = \hat{y}\\
\text{If } y=0 \quad &= \hat{y}^0 \cdot (1-\hat{y})^1 = (1-\hat{y})
\end{aligned}$$

- We want to maximize

$$\begin{aligned}
\log p(y|x) &= \log(\hat{y}^y \cdot (1-\hat{y})^{(1-y)})\\
&= y\log(\hat{y}) + (1-y)\log(1-\hat{y})\\
&= -\mathcal{L}(\hat{y}, y)
\end{aligned}$$

  - $\Rightarrow$ Maximizing $\log p(y|x)$ is equivalent to mimizing $\mathcal{L}(\hat{y}, y)$

- On the whole training set

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

$$\begin{aligned}
\log p(\text{labels }) &= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)})\\
&= \sum_{i=1}^{m} -\mathcal{L}(y^{(i)}, y^{(i)})
\end{aligned}$$

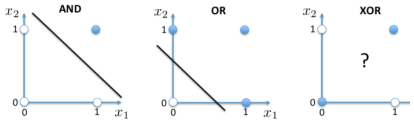
- Since the Cost is defined as $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(y^{(i)}|x^{(i)})$.
  - $\Rightarrow$ Maximizing the Likelihood is equivalent to minimizing the Cost

Logistic Regression (0 hidden layers)

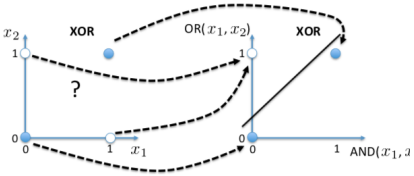
Limitation of linear classifiers

- 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 :



Logistic regression, Softmax regression, Multi-label, Multi-Class

Logistic regression (binary classification)

- **Logistic Regression** does binary classification (not regression)

$$P(y=1|\underline{x}) = h_w(x) = Logistic(\underline{w}\cdot\underline{x}) = \underbrace{\frac{1}{1+e^{-\underline{w}\cdot\underline{x}}}}_{\text{sigmoid function}} \tag{2}$$

$$P(y=0|\underline{x}) = 1 - P(y=1|\underline{x}) \tag{3}$$

- We can show that it models the posterior probability of the class $y=1$ using linear functions of the inputs $\underline{x}$
  - Proof :

$$P(y=1|\underline{x}) = \frac{1}{1+e^{-f(\underline{x})}} = \frac{e^{f(\underline{x})}}{1+e^{f(\underline{x})}} \tag{4}$$

$$\frac{1}{P(y=1|\underline{x})} = 1 + e^{-f(\underline{x})} \tag{5}$$

$$\frac{1-P(y=1|\underline{x})}{P(y=1|\underline{x})} = e^{-f(\underline{x})} \tag{6}$$

$$\log\left(\frac{P(y=1|\underline{x})}{P(y=0|\underline{x})}\right) = f(\underline{x}) \tag{7}$$

$$\log\left(\frac{P(y=1|\underline{x})}{P(y=0|\underline{x})}\right) = \underline{w}\cdot\underline{x} \tag{8}$$

Softmax regression (multi-class classification)

- **Softmax Regression** :
  - Model the posterior probabilities of the K classes using linear functions of the inputs $\underline{x}$, according to :

$$\log\left(\frac{P(y=c_1|\underline{x})}{P(y=c_k|\underline{x})}\right) = \underline{w}_1\underline{x} + w_{10} \quad (9)$$

$$\log\left(\frac{P(y=c_2|\underline{x})}{P(y=c_k|\underline{x})}\right) = \underline{w}_2\underline{x} + w_{20} \quad (10)$$

$$..... \quad (11)$$

$$\log\left(\frac{P(y=c_k-1|\underline{x})}{P(y=c_k|\underline{x})}\right) = \underline{w}_{k-1} + w_{(k-1)0}\underline{x} \quad (12)$$

$$(13)$$

  - It is easy to deduce that

$$P(c_k|\underline{x}) = \frac{e^{\underline{w}_k\underline{x}+w_{k0}}}{1+\sum_{l=1}^{K-1} e^{\underline{w}_l\underline{x}+w_{l0}}} \quad k = 1\dots K-1 \quad (14)$$

$$P(c_K|\underline{x}) = \frac{1}{1+\sum_{l=1}^{K-1} e^{\underline{w}_l\underline{x}+w_{l0}}} \quad (15)$$

  - We of course have

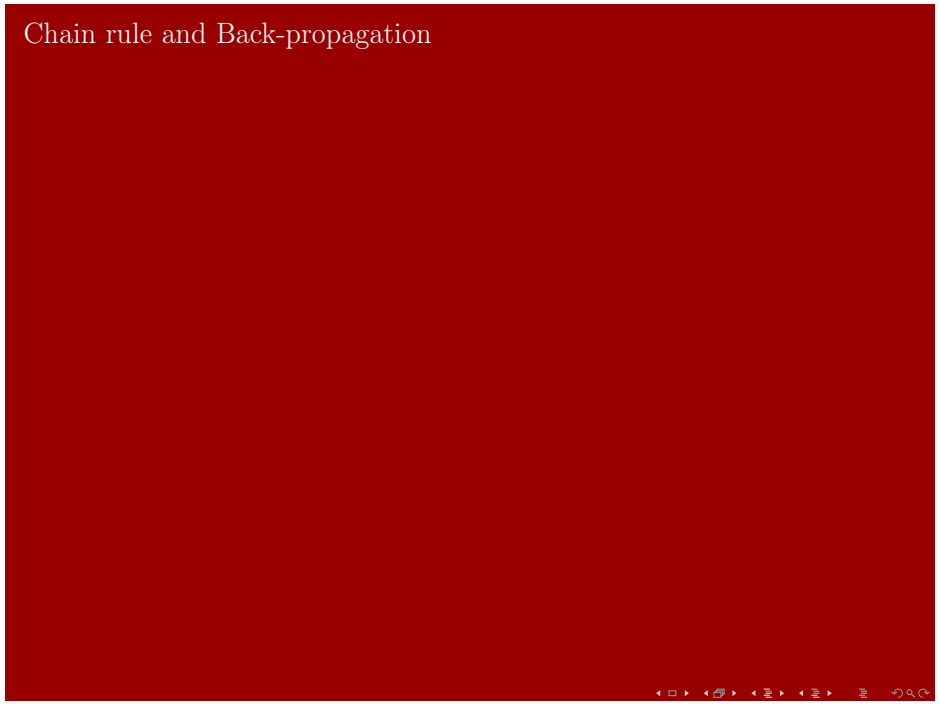
$$P(c_1|\underline{x}) + P(c_2|\underline{x}) + ... + P(c_K|\underline{x}) = 1 \quad (16)$$

- **Softmax function** (generalization of logistic function)

$$P(y=j|\underline{x}) = \frac{e^{\underline{w}_j\cdot\underline{x}}}{\sum_{k=1}^{K} e^{\underline{w}_k\cdot\underline{x}}} \quad (17)$$

using $\underline{w}_K = 0$

Chain rule and Back-propagation



Chain rule and Back-propagation

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

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

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

  - Example

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

  - Example :

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

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

Chain rule and Back-propagation

- **Univariate logistic least squares model**

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

- **Computing derivative as in calculus class**

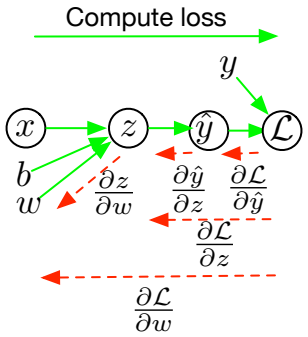
$$\mathcal{L} = \frac{1}{2}(\sigma(wx+b)-y)^2$$

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

Chain rule and Back-propagation

- **Univariate logistic least squares model**

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



- **Computing derivative using back-propagation**

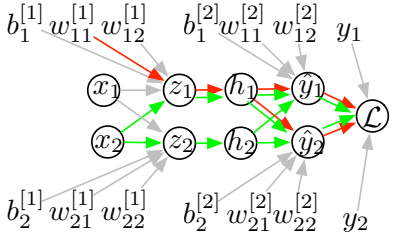
$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y$$
$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial \hat{y}}\sigma'(z)$$
$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial z}x$$
$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial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

- 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

Chain rule and Back-propagation

- **Multilayer Perceptron (multiple outputs)**

$$z_i = \sum_j w_{ij}^{[1]} x_j + b_i^{[1]}$$
$$h_i = \sigma(z_i)$$
$$\hat{y}_k = \sum_i w_{ki}^{[2]} h_i + b_l^{[2]}$$
$$\mathcal{L} = \frac{1}{2}\sum_k (\hat{y}_k - y_k)^2$$



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

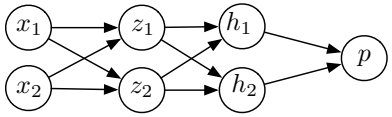
$$\frac{\partial \mathcal{L}}{\partial \hat{y}_k} = \hat{y}_k - y_k$$
$$\frac{\partial \mathcal{L}}{\partial w_{ki}^{[2]}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_k} h_i \qquad \frac{\partial \mathcal{L}}{\partial b_k^{[2]}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_k}$$
$$\frac{\partial \mathcal{L}}{\partial h_i} = \sum_k \frac{\partial \mathcal{L}}{\partial \hat{y}_k} w_{ki}^{[2]}$$
$$\frac{\partial \mathcal{L}}{\partial z_i} = \frac{\partial \mathcal{L}}{\partial h_i}\sigma'(z_i)$$
$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{[1]}} = \frac{\partial \mathcal{L}}{\partial z_i} x_j \qquad \frac{\partial \mathcal{L}}{\partial b_i^{[1]}} = \frac{\partial \mathcal{L}}{\partial z_i}$$

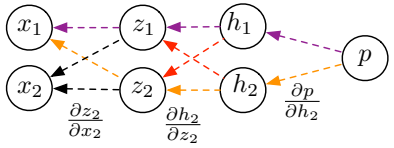
Chain rule and Back-propagation

$$z_1 = z_1(x_1, x_2)$$
$$z_2 = z_2(x_1, x_2)$$
$$h_1 = z_1(z_1, z_2)$$
$$h_2 = z_2(z_1, z_2)$$
$$p = p(h_1, h_2)$$

$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1}\underbrace{\frac{\partial h_1}{\partial x_1}}_{\frac{\partial h_1}{\partial z_1}\frac{\partial z_1}{\partial x_1}+\frac{\partial h_1}{\partial z_2}\frac{\partial z_2}{\partial x_1}} + \frac{\partial p}{\partial h_2}\underbrace{\frac{\partial h_2}{\partial x_1}}_{\frac{\partial h_2}{\partial z_1}\frac{\partial z_1}{\partial x_1}+\frac{\partial h_2}{\partial z_2}\frac{\partial z_2}{\partial x_1}}$$

$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1}\frac{\partial h_1}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1}\frac{\partial h_1}{\partial z_2}\frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2}\frac{\partial h_2}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_2}\frac{\partial h_2}{\partial z_2}\frac{\partial z_2}{\partial x_1}$$





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

Neural Networks (1 hidden layer)

Neural Networks (1 hidden layer)

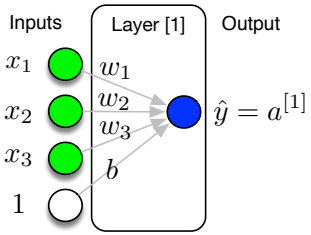
- **Logistic Regression** (1 layer, **0 hidden layer**)

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

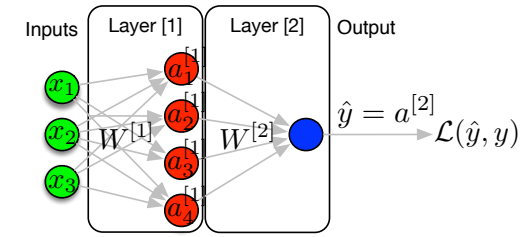
- **Neural Network** (2 layers, **1 hidden layer**)

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

Logistic Regression



Neural Network



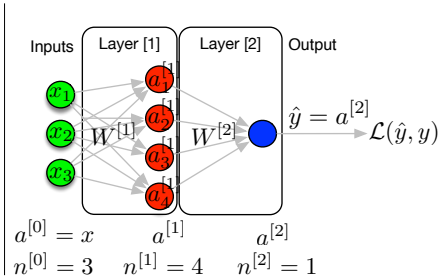
$a^{[0]} = x \quad a^{[1]} \quad a^{[2]}$

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

Neural Networks (1 hidden layer)

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

- **Notations :**
  - variables of **layer** $l \Rightarrow *^{[l]}$
  - **dimension** d of variables $\Rightarrow *_d^{[l]}$
  - training **example** $i \Rightarrow *_d^{[l](i)}$
  - input $x \Rightarrow a^{[0]}$
  - dimensions of layer $l \Rightarrow n^{[l]}$
  - $g^{[l]}$: activation function
    - (possibly a sigmoid σ function)



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

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

Neural Networks (1 hidden layer)

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

Gradient Descent

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

Neural Networks (1 hidden layer)

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

Forward propagation (all dimensions, all m training examples)

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

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

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

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

Neural Networks (1 hidden layer)

Backward propagation (each training example)

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

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

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

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

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

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

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

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

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

Neural Networks (1 hidden layer)

Backward propagation (all m training examples)

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

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

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

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

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

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

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

Neural Networks (1 hidden layer)

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

Parameters update

- **Parameters update :**

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

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

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**)

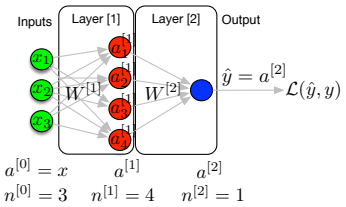
$$\underline{x} \xrightarrow{\underline{a}^{[0]}} \boxed{\underline{\underline{W}}^{[1]}\underline{x} + \underline{b}^{[1]}} \xrightarrow{\underline{z}^{[1]}} \boxed{g^{[1]}(\underline{z}^{[1]})} \xrightarrow{\underline{a}^{[1]}} \boxed{\underline{\underline{W}}^{[2]}\underline{a}^{[1]} + \underline{b}^{[2]}}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

$Layer[1]$ $Layer[2]$

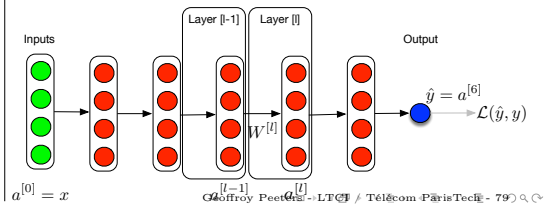
- **Deep Neural Networks** (many layers, **>2 hidden layer**)

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

Neural Network



Deep Neural Networks



Deep Neural Networks (> 2 hidden layers)

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

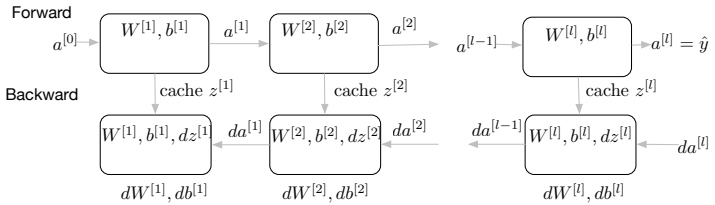
Forward (general formulation for layer l, all m training examples)

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

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

Output : $\underline{\underline{A}}^{[l]}$, cache($\underline{\underline{Z}}^{[l]}$, $\underline{\underline{W}}^{[l]}$, $\underline{b}^{[l]}$)

Deep Neural Networks (> 2 hidden layers)



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

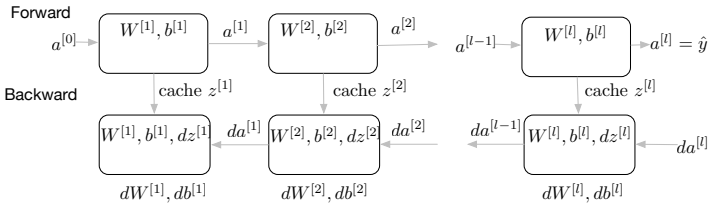
Input : $da^{[l]}$

Cache (passed from Forward) : $z^{[l]}$

$$d\underline{\underline{Z}}^{[l]} = d\underline{\underline{A}}^{[l]} \odot g^{[l]\prime}(\underline{\underline{Z}}^{[l]})$$
$$d\underline{\underline{W}}^{[l]} = \frac{1}{m} d\underline{\underline{Z}}^{[l]} \cdot \underline{\underline{A}}^{[l-1]T}$$
$$d\underline{b}^{[l]} = \frac{1}{m} \sum_{i=1}^{m} d\underline{\underline{Z}}^{[l]}$$
$$d\underline{\underline{A}}^{[l-1]} = \underline{\underline{W}}^{[l]T} \cdot d\underline{\underline{Z}}^{[l]}$$

Output : $da^{[l-1]}$, $dW^{[l]}$, $db^{[l]}$

Deep Neural Networks (> 2 hidden layers)



Parameters update

$$\underline{\underline{W}}^{[l]} = \underline{\underline{W}}^{[l]} - \alpha d\underline{\underline{W}}^{[l]}$$
$$\underline{b}^{[l]} = \underline{b}^{[l]} - \alpha d\underline{b}^{[l]}$$

where α is the learning rate

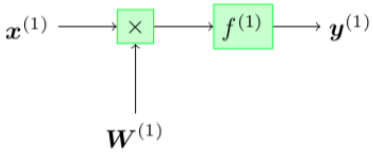
Computation Graph (from Alexandre Allauzen)

Computation graph

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

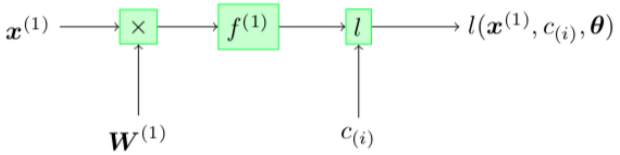
Example 1 : A single layer network



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

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

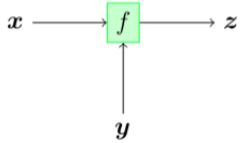
Computation graph for training



- 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 ($W^{(1)}$)
  $\rightarrow$ backward pass

A function node

Forward pass

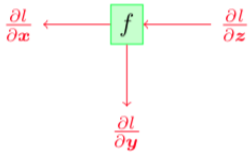


This node implements :

$$z = f(x, y)$$

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 :

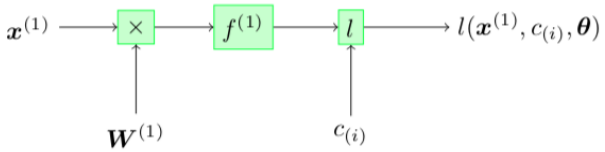
$$(\frac{\partial l}{\partial z}\frac{\partial z}{\partial x}), (\frac{\partial l}{\partial z}\frac{\partial z}{\partial y})$$

Summary of a function node

f :

x, y, z	# store the values
$z = f(x, y)$	# forward
$\frac{\partial z}{\partial x} \rightarrow \frac{\partial f}{\partial x}$	# local gradients
$\frac{\partial z}{\partial y} \rightarrow \frac{\partial f}{\partial y}$	
$(\frac{\partial l}{\partial z}\frac{\partial z}{\partial x}),(\frac{\partial l}{\partial z}\frac{\partial z}{\partial y})$	# backward

Example of a single layer network



Forward

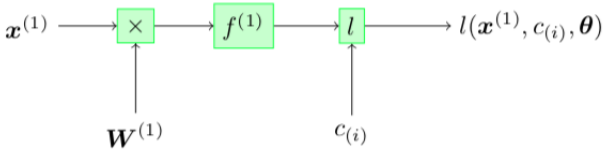
For each function node in topological order

- forward propagation

Which means :

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

Example of a single layer network



Backward

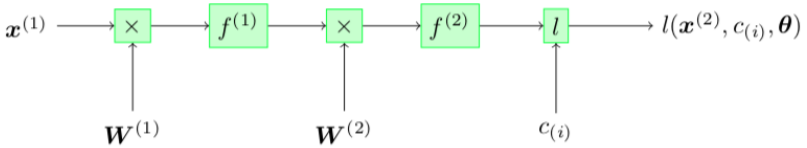
For each function node in reversed topological order

- backward propagation

Which means :

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

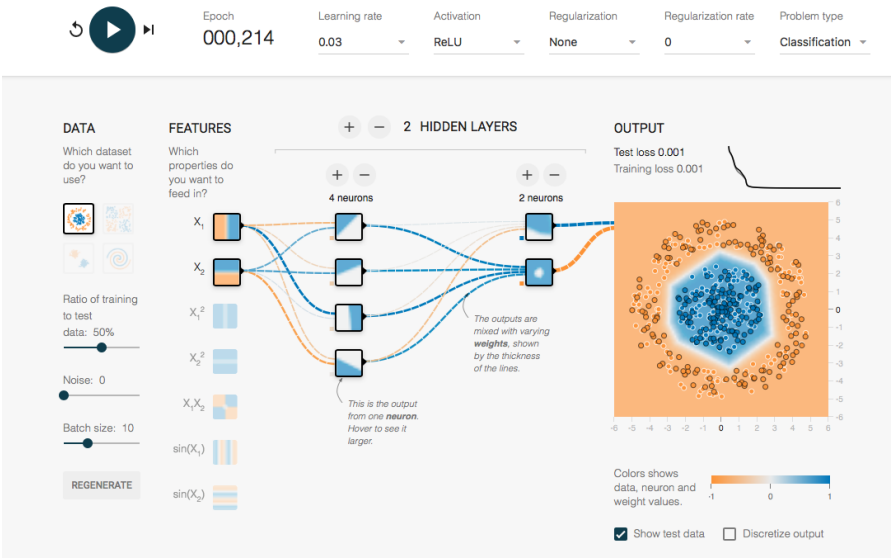
Example of a two layers network



- 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

Playground



Deep Learning Frameworks

- **Python**
  - coding forward, loss, backward, upgrading
    - painfull, proned to coding errors
- **(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
- CPU/GPU
- Automatic differentiation based on computational graph

Example of an MLP in python

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

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

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

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

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

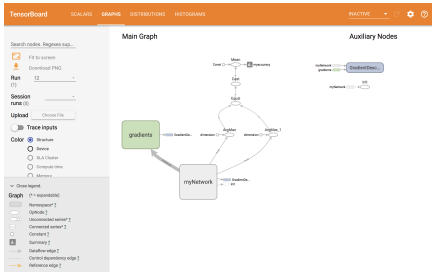

Example of an MLP in pytorch

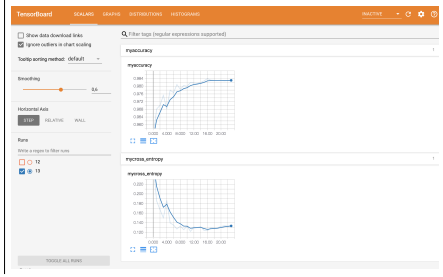
```python
import torch
from torch.autograd import Variable
# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension. N, D_in, H, D_out = 64, 1000, 100, 10
# Create random Tensors to hold inputs and outputs,
# and wrap them in Variables.
x = torch.randn(N, D_in)
y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

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


Example of an MLP in tensorflow

```python
import tensorflow as tf

# declare the training data placeholders
# input x - for 28 x 28 pixels = 784
x = tf.placeholder(tf.float32, [None, D_in])
# now declare the output data placeholder - 10 digits
y = tf.placeholder(tf.float32, [None, D_out])
# now declare the weights connecting the input to the hidden layer
W1 = tf.Variable(tf.random_normal([D_in, H], stddev=0.03), name='W1')
b1 = tf.Variable(tf.random_normal([H]), name='b1')
# and the weights connecting the hidden layer to the output layer
W2 = tf.Variable(tf.random_normal([H, D_out], stddev=0.03), name='W2')
b2 = tf.Variable(tf.random_normal([D_out]), name='b2')
# calculate the output of the hidden layer
hidden_out = tf.add(tf.matmul(x, W1), b1)
hidden_out = tf.nn.relu(hidden_out)
# output layer: now calculate the hidden layer output - in this case, let's use a softmax activated
y_ = tf.nn.softmax(tf.add(tf.matmul(hidden_out, W2), b2))
y_clipped = tf.clip_by_value(y_, 1e-10, 0.9999999)
cross_entropy = -tf.reduce_mean(tf.reduce_sum(y * tf.log(y_clipped) + (1 - y) * tf.log(1 - y_clipped), axis=1))
# add an optimiser
optimiser = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(cross_entropy)
# define an accuracy assessment operation
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
# finally setup the initialisation operator
init_op = tf.global_variables_initializer()
# start the session
with tf.Session() as sess:
    # initialise the variables
    sess.run(init_op)
    total_batch = int(N / batch_size)
    for epoch in range(nbEpochs):
        avg_cost = 0
        for i in range(total_batch):
            batch_x, batch_y = x_train[i*batch_size:(i+1)*batch_size], y_train_ohv[i*batch_size:(i+1)*batch_size]
            _, c = sess.run([optimiser, cross_entropy], feed_dict={x: batch_x, y: batch_y})
            avg_cost += c / total_batch
        test_acc = sess.run(accuracy, feed_dict={x: x_test, y: y_test_ohv})
        print("Epoch:", (epoch + 1), "cost =", "{:.3f}".format(avg_cost), " test accuracy: {:.3f}".format(test_acc))
    print("\nTraining complete!")
    print(sess.run(accuracy, feed_dict={x: x_test, y: y_test_ohv}))
```


Tensorboard

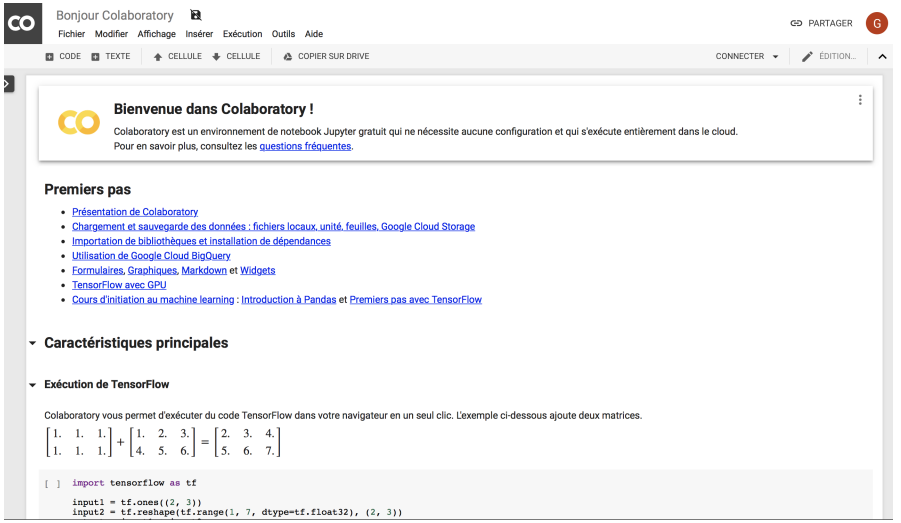




Example of an MLP in Keras

```
import keras
from keras.models import Sequential
from keras.layers import Dense

# Convert labels to categorical one-hot encoding
y_train_ohv = keras.utils.to_categorical(Y_train, num_classes=D_out)
y_test_ohv = keras.utils.to_categorical(Y_test, num_classes=D_out)

model = Sequential()
model.add(Dense(H, activation='relu', input_dim=D_in))
model.add(Dense(D_out, activation='softmax'))
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
# Train the model, iterating on the data in batches of 32 samples
model.fit(x_train, y_train_ohv, epochs=nbEpochs, batch_size=batch_size, validation_data=(x_test, y_test_ohv))
score = model.evaluate(x_test, y_test_ohv, verbose=0)
print(score)
```


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



Various types of training

Various types of training

Batch Gradient Descent

- **Gradient computation** $\frac{\partial J(w,b)}{\partial w_d}$
  - average of the gradient $w_d^{(i)}$ over **all** m training examples
- **Parameters update**
  - after the m training examples went to forward
- **Pros and Cons**
  - (+) The gradient is accurate
  - (-) 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

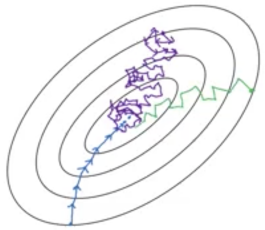
Mini Batch Gradient Descent

- **Gradient computation** $\frac{\partial J(w,b)}{\partial w_d}$
  - average of the gradient $w_d^{(i)}$ over each **mini-batch**
- **Mini-batch ?**
  - Split training set into small training sets : $X^{\{t\}}, Y^{\{t\}}$
  - Suppose mini-batch are of size 1000 :
    $$X = [\underbrace{X^{(1)}}_{(n_x,1)} \cdots X^{(1000)} | \cdots | X^{(2001)} \cdots X^{(5.000.000)}] = [\underbrace{X^{\{1\}}}_{(n_x,1000)} \underbrace{X^{\{2\}}}_{(n_x,1000)} \cdots \underbrace{X^{\{5000\}}}_{(n_x,1000)}]$$
- **Parameters update**
  - after each mini-batch
- **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 : $W^{[l]} = W^{[l]} - \alpha dW^{[l]}$, $b^{[l]} = b^{[l]} - \alpha db^{[l]}$
  - One pass to the training set= one **"epoch"**
- **Pros and Cons**
  - (+) Allows to do 5000 gradient-descent steps
  - (-) Gradient is an approximation

Various types of training

Stochastic Gradient Descent (SGD)

- **Gradient computation** $\frac{\partial J(w,b)}{\partial w_d}$
  - directly the gradient **on a single training examples** i (as mini-batch but with a mini-batch size of 1)
- **Parameters update**
  - after each training examples
- **Pros and Cons**
  - (+) Allows to do 5.000.000 gradient-descent steps
  - (-) Can lead to a very noisy gradient descent (see Figure)
  - (-) Lose speed up from vectorization



Activation functions $a = g(z)$

Activation functions $a = g(z)$

Why non-linear activation functions ?

- Consider the following network
  $$\underline{z}^{[1]} = \underline{\underline{W}}^{[1]}\underline{x} + \underline{b}^{[1]}$$
  $$\underline{a}^{[1]} = g^{[1]}(\underline{z}^{[1]})$$
  $$\underline{z}^{[2]} = \underline{\underline{W}}^{[2]}\underline{a}^{[1]} + \underline{b}^{[2]}$$
  $$\underline{a}^{[2]} = g^{[2]}(\underline{z}^{[2]})$$
- If $g^{[1]}$ and $g^{[2]}$ are **linear activation functions** (identity function),
  $$\underline{a}^{[1]} = \underline{z}^{[1]} = \underline{\underline{W}}^{[1]}\underline{x} + \underline{b}^{[1]}$$
  $$\underline{a}^{[2]} = \underline{z}^{[2]} = \underline{\underline{W}}^{[2]}\underline{a}^{[1]} + \underline{b}^{[2]}$$
  $$= \underline{\underline{W}}^{[2]}\left(\underline{\underline{W}}^{[1]}\underline{x} + \underline{b}^{[1]}\right) + \underline{b}^{[2]}$$
  $$= \underline{\underline{W}}^{[2]}\underline{\underline{W}}^{[1]}\underline{x} + \underline{\underline{W}}^{[2]}\underline{b}^{[1]} + \underline{b}^{[2]}$$
  $$= \underline{\underline{W}}'\underline{x} + b'$$
  - 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 $a = g(z)$

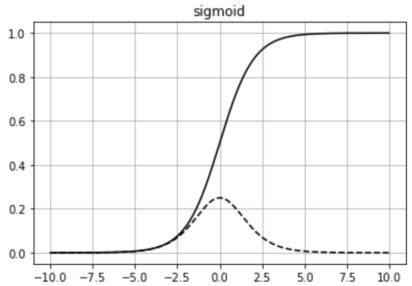
Sigmoid σ

- **Sigmoid function**

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

- **Derivative**

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



Activation functions $a = g(z)$

Hyperbolic tangent g

- **Hyperbolic tangent function**

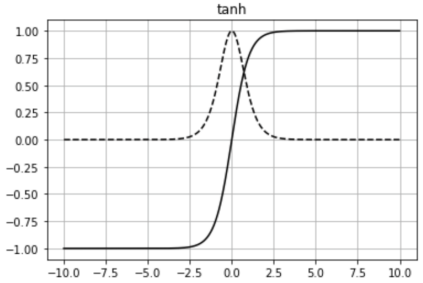
$$a = g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

- **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)
    - ⇒ slope becomes zero
    - ⇒ slow down Gradient Descent



Activation functions $a = g(z)$

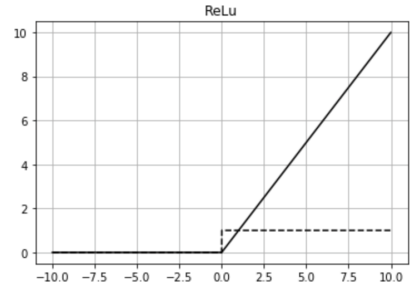
ReLU (Rectified Linear Unit)

- **ReLU function**

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

- **Derivative**

$$\begin{aligned} g'(x) &= 1 && \text{if } z > 0 \\ &= 0 && \text{if } z \leqslant 0 \end{aligned}$$



Activation functions $a = g(z)$

Leaky ReLU / Parametric ReLU

- **Leaky ReLU function**

$$a = g(x) = \max(0.01z, z)$$

  - allows avoiding the zero slope ("the neuron dies") for $z < 0$
  - **Derivative**

$$\begin{aligned} g'(x) &= 1 && \text{if } z > 0 \\ &= 0.01 && \text{if } z \leqslant 0 \end{aligned}$$

- **PReLU function**

$$a = g(x) = \max(\alpha z, z)$$

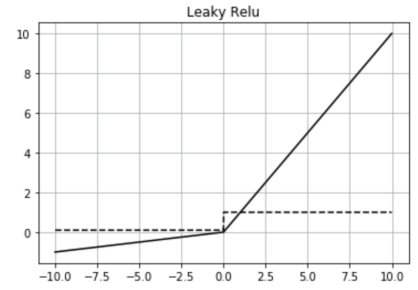
  - same as Leaky ReLU but α is a parameter to be learnt
  - **Derivative**

$$\begin{aligned} g'(x) &= 1 && \text{if } z > 0 \\ &= \alpha && \text{if } z \leqslant 0 \end{aligned}$$

- **Softplus function**

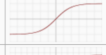
$$g(x) = \log(1 + e^x)$$

  - continuous appriximation of ReLU
  - **Derivative**



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

Weight initialization

Weight initialization

Weight initialization with random values

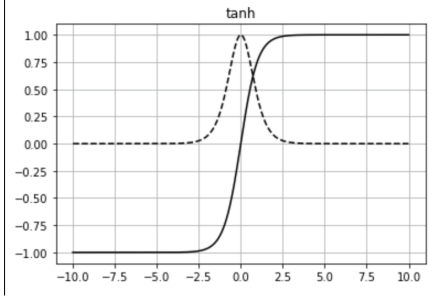
- **Random initialization**
  - $W^{[1]} =$ `np.random.randn((2,2))*0.01`
  - $b^{[1]} =$ `0`
  - $W^{[2]} =$ `np.random.randn((1,2))*0.01`
  - $b^{[1]} =$ `0`
- Remark : b doesn't have the symetry problem

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

$$Z^{[1]} = W^{[1]}X + b^{[1]} \quad (19)$$

$$A^{[1]} = g^{[1]}(Z^{[1]}) \quad (20)$$

    - $\Rightarrow$ 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)



Weight initialization

Vanishing gradient (part 2)

- Reminder :

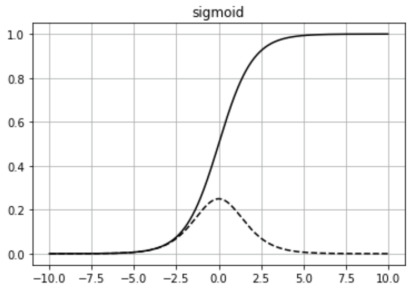
$$dz^{[l]} = W^{[l+1]}dz^{[l+1]} \odot g^{[l]\prime}(z^{[l]})$$

$$db^{[l]} = \frac{1}{m}\sum_m dZ^{[l]}$$

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

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

- 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



Regularization

Regularization

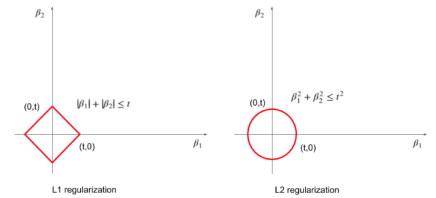
L1 and L2 regularization

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

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

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

- **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

$$W^{[l]} \leftarrow W^{[l]} - \alpha d'W^{[l]}$$

$$\leftarrow W^{[l]} - \alpha(dW^{[l]} + \frac{\lambda}{m}W|ll)$$

$$\leftarrow W^{[l]} - \frac{\alpha\lambda}{m}W^{[l]} - \alpha dW^{[l]}$$

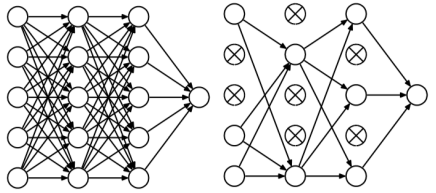
$$\leftarrow W^{[l]}\underbrace{(1 - \frac{\alpha\lambda}{m})}_{\leqslant 1} - \alpha dW^{[l]}$$

$$\underbrace{\qquad\qquad}_{\text{weight decay}}$$

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 $\underline{W}$ $\Rightarrow p$ 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

Alternative gradient descent algorithms

Alternative gradient descent algorithms

Momentum

- What ?
  - 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
- On iteration t
  - Compute $\frac{\partial \mathcal{L}(\cdots W(t-1)\cdots)}{\partial W}$, $\frac{\partial \mathcal{L}(\cdots b(t-1)\cdots)}{\partial b}$ on current mini-batch

$$V_{dW}(t) = \beta V_{dW}(t-1) + (1-\beta)\frac{\partial \mathcal{L}(\cdots W(t-1)\cdots)}{\partial W}$$
$$W(t) = W(t-1) - \alpha V_{dW}(t)$$

$$V_{db}(t) = \beta V_{db}(t-1) + (1-\beta)\frac{\partial \mathcal{L}(\cdots b(t-1)\cdots)}{\partial b}$$
$$b(t) = b(t-1) - \alpha V_{db}(t)$$

  - 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.

Normalization inputs

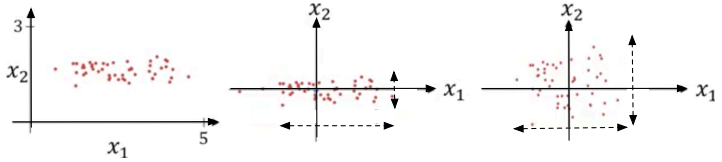
- **"Standardizing" :**
  - subtracting a measure of location and dividing by a measure of scale
  - subtract the mean and divide by the standard deviation

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

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

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

- We will use the same μ and σ^2 to normalize the test set



Batch Normalization (BN)

- **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 depends on the current 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 $z^{[l-1]}$
  - We don't want all units to always mean 0 and standard-deviation 1
    - We still allows to learn an appropriate bias β and scale γ to the input to the non-linear function $g^{[l]}$

Batch Normalization (BN)

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

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

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

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

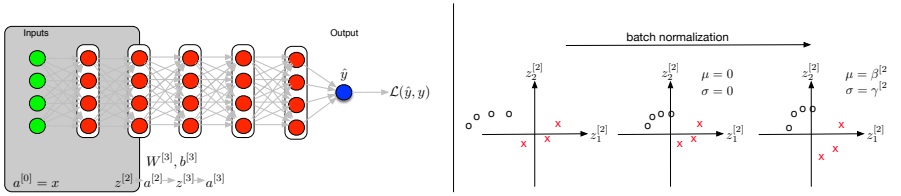
$$\tilde{z}^{[l](i)} = \gamma \cdot z^{[l](i)}_{norm} + \beta$$

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

- **New parameters to be trained : γ and β**
  - γ allows to set the variance of $\tilde{z}^{[l]}$
  - β allows to set the mean of $\tilde{z}^{[l]}$
  - If $\gamma = \sqrt{\sigma^{2[l]} + \epsilon}$ and $\beta = \mu$ then $\tilde{z}^{[l](i)} = z^{[l](i)}$

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 we train $W^{[3]}, b^{[3]}$, the input values $\underline{a}^{[2]}$ will change over time (since previous layers weight also change during training)
- **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 $X^{\{t\}}$
  - add 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)