Université de Rennes

istic Informatique Électronique

UMR IRISA

Neural Networks basics

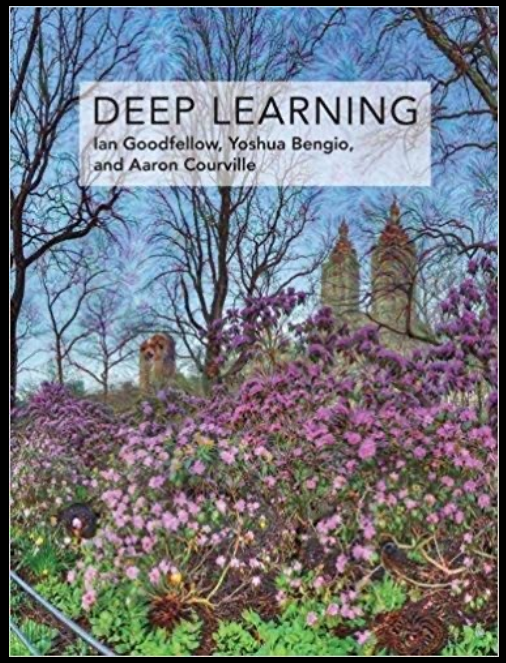
(P1_2)

Elisa Fromont

M2 SIF DLV

http://people.irisa.fr/Elisa.Fromont/Cours/DLV/DLV.html

RESOURCES



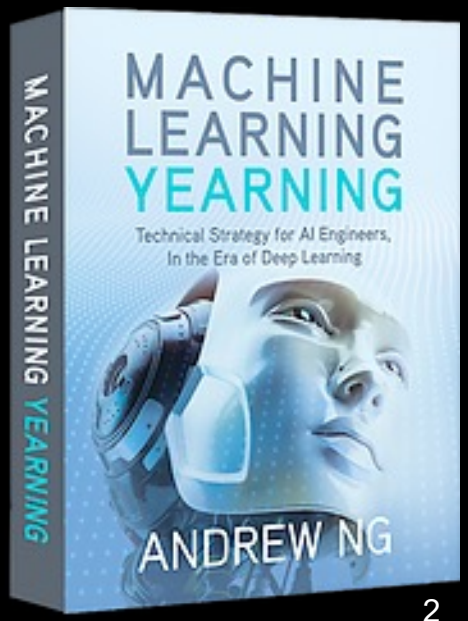
Some slides are borrowed from:

Damien Fourure, Kenrick Mock (University of Alaska, Anchorage), Tony R. Martinez (Brigham Young University), Hugo Larochelle, Yoshua Bengio, Jerome Louradour, Pascal Lamblin, Geoffrey Hinton, Andrew Ng., Andrew L. NelsonR. Salskhutdino, Su-A Kim.

For RNN:

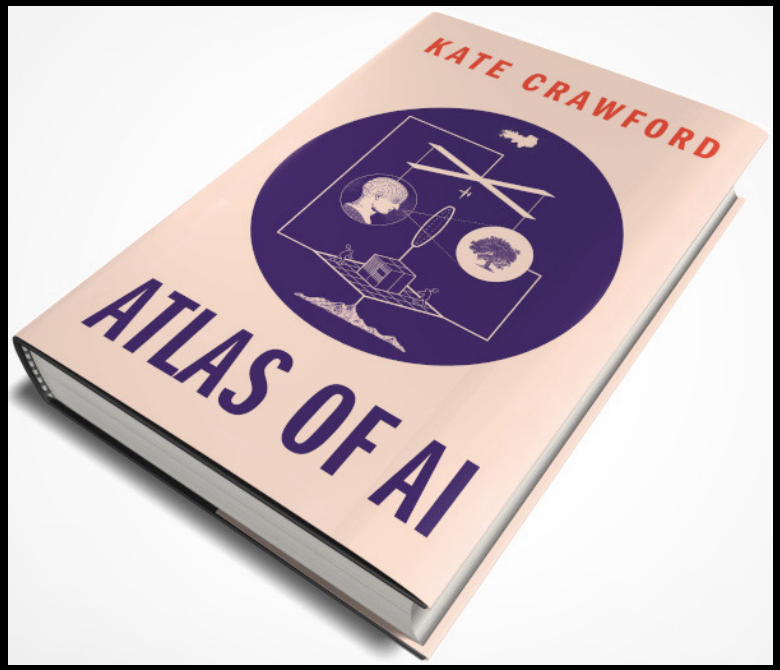
- http://cs231n.stanford.edu/slides/winter1516_lecture10.pdf
- http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

- More to learn here:
  - **Deep Learning** (Adaptive Computation and Machine Learning series) by Ian Goodfellow, Yoshua Bengio Aaron Courville
  - **Machine Learning Yearning** (Andrew Ng)



About the impact of AI on earth

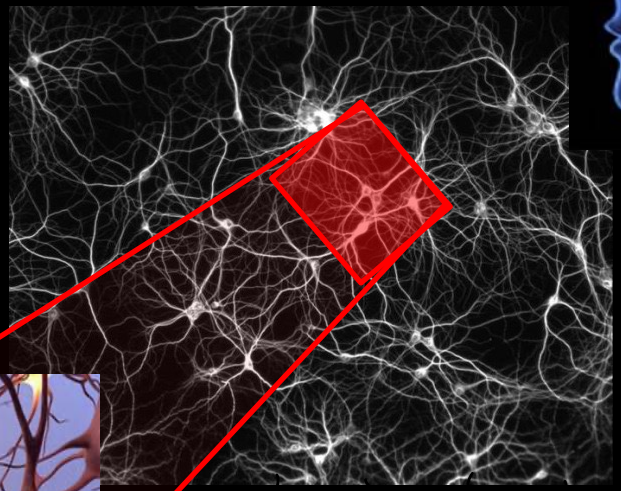
Atlas of Ai: Power, Politics, and the Planetary Costs of Artificial Intelligence

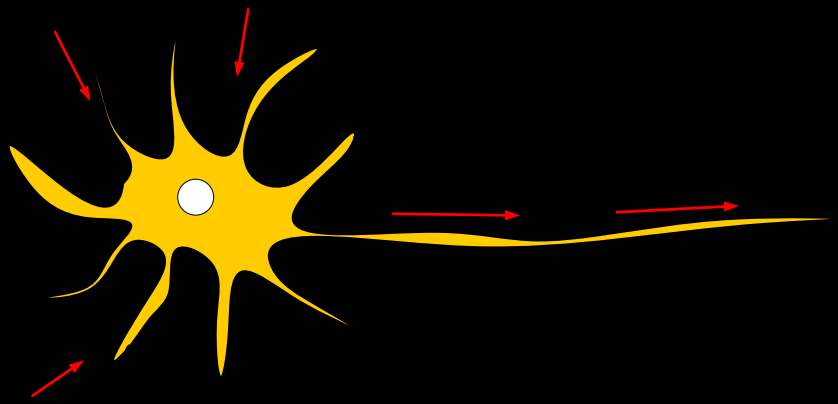


2021 (pre Chat GPT)

- Chapter 1: mineral extraction needed to power contemporary computation
- Chapter 2: how artificial intelligence is made of human labor
- Chapter 3: the role of data (not people's personal anymore but "infrastructure")
- Chapter 4: how automatic classification can be really offensive
- Chapter 5: about recognizing affect
- …

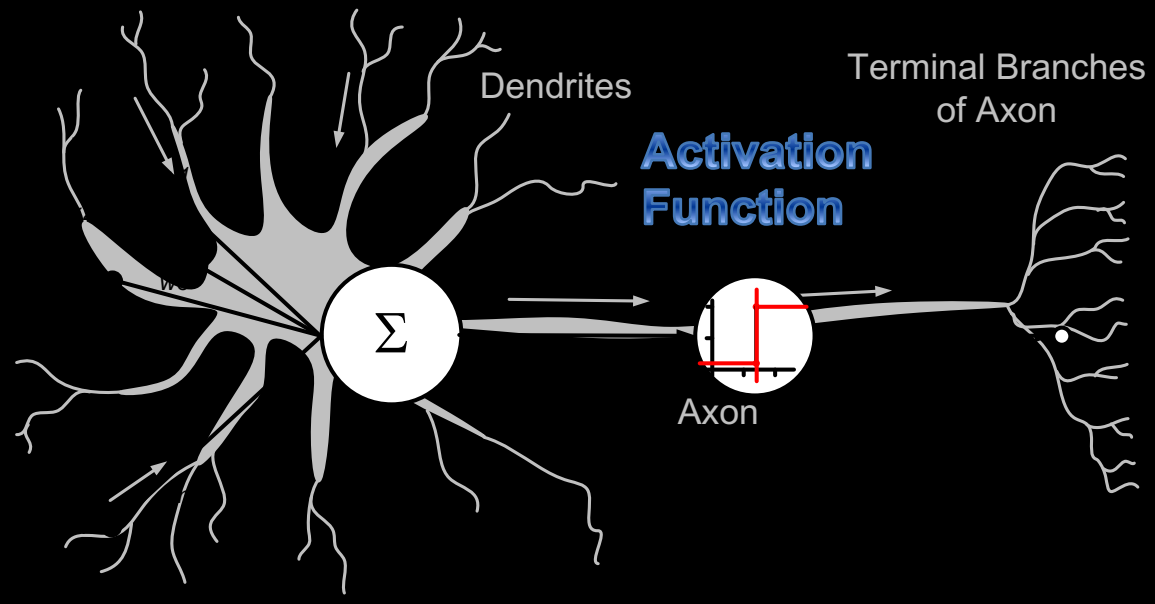
Biological Neurons





Neurons in the Brain

- Although heterogeneous, at a low level the brain is composed of neurons
  - A neuron receives input from other neurons (generally thousands) from its synapses
  - Inputs are approximately summed
  - When the input exceeds a threshold the neuron sends an electrical spike that travels from the body, down the axon, to the next neuron(s)



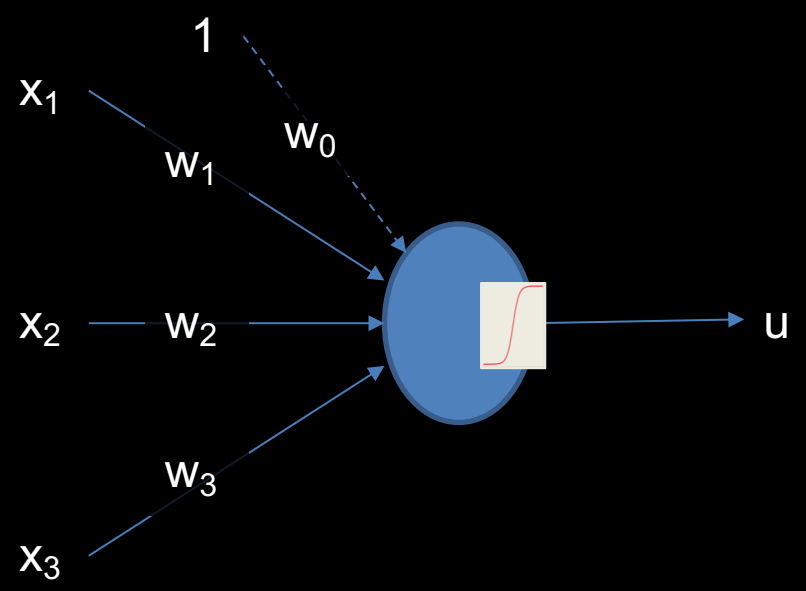
The McCullogh-Pitts model [1943]

- spikes are interpreted as spike rates;
- synaptic strength are translated as synaptic weights;
- excitation means positive product between the incoming spike rate and the corresponding synaptic weight;
- inhibition means negative product between the incoming spike rate and the corresponding synaptic weight;

Neural Network History

- History traces back to the 50’s but became popular in the 80’s with work by Rumelhart, Hinton, and Mclelland
  - A General Framework for Parallel Distributed Processing : explorations in the microstructure of cognition
- Peaked in the 90’ then “desert crossing”.
- Today
  - confusion btw machine learning and deep learning!
  - Hundreds of variants (thousands of research papers)
  - Less a model of the actual brain than a useful tool, but still some debate
- Numerous applications
  - Handwriting, face, speech recognition
  - Autonomous vehicles
  - Models of reading, sentence production, dreaming
- Debate for philosophers and cognitive scientists
  - Can human consciousness or cognitive abilities be explained by a connectionist model or does it require the manipulation of symbols?

Perceptron

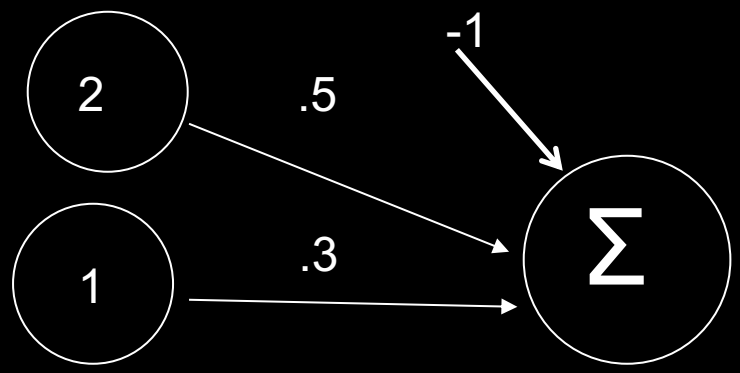


- Initial proposal of connectionist networks
- Rosenblatt (learning rule), late 50's
- Essentially a linear discriminant composed of nodes, weights
- h is an activation function (in the original formulation, step function)

If h(x) is an activation function, then a perceptron (*in this example* with 3 inputs) is defined as:

$$u = F(\mathbf{x},\mathbf{w}) = h(w_0 + w_1x_1 + w_2x_2 + w_3 x_3)$$
$$= h(\sum_{i=0}^{d} w_i * xi)$$
$$= h(\mathbf{w}\mathbf{x}^T)$$

Learning the Perceptron's weights (with h = step function)



$$F(X,W) = u = \begin{cases} 1 : \left(\sum_i w_i x_i\right) + w0 > 0 \\ 0 : otherwise \end{cases}$$

$\Sigma_{i=0,2} w_i x_i$ = 2(0.5) + 1(0.3) + -1 = 0.3 , F(X,W) =1 (because 0.3 >0)

Learning Procedure:

1. Randomly assign weights (e.g. between [-1,1])
2. Present inputs from training data (sequentially)
3. Get output F(x,W), **change weights** (with the perceptron learning rule) to gives results toward our desired output y
4. Repeat from 2; stop when no errors, or enough epochs completed

Perceptron Learning Rule

$$W_i^{t+1} = W_i^t + \Delta W_i^t$$
$$\Delta W_i^t = c*(y-u)*x_i$$

$$F(X,W) = u = \left\{ \begin{array}{c} 1 : \left(\sum_i w_i x_i \right) + w0 > 0 \\ 0 : otherwise \end{array} \right\}$$

Example **y**: desired output, actual output **u** = F(**x**,W),

2 inputs (x_1, x_2), x_0 = 1, c = 1 (learning rate)

At step t : y=0, u=1, w_1=0.5, w_2=0.3, x_1=2, x_2=1, w_0= -1

$$W_0^{t+1} = -1 + (0-1) * 1 = -2$$
$$W_1^{t+1} = 0{,}5 + (0-1) * 2 = -1{,}5$$
$$W_2^{t+1} = 0{,}3 + (0-1) * 1 = -0{,}7$$

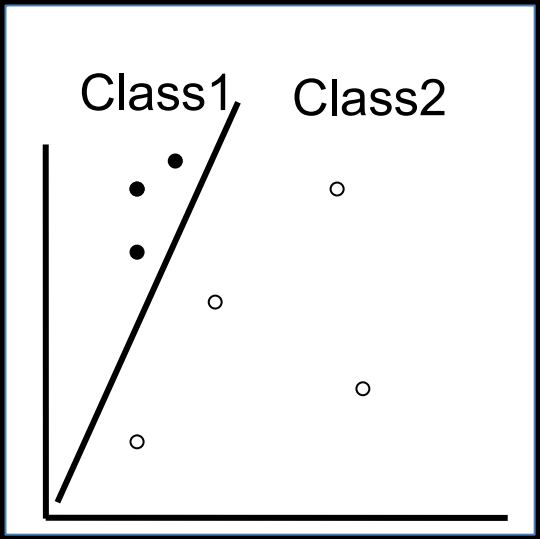
Note that if we present this input again, we'd output 0 instead

How might you use a perceptron?

- They (and other networks) are generally used to learn how to make predictions (classification or regression)
- Say you have collected some data regarding the diagnosis of patients with heart disease
  - Age, Sex, Chest Pain Type, Resting BPS, Cholesterol, …, Diagnosis (<50% diameter narrowing, >50% diameter narrowing)
  - 67,1,4,120,229,…, 1
  - 37,1,3,130,250,… ,0
  - 41,0,2,130,204,… ,0
- Train network to predict heart disease of new patient

Remark on (original) perceptrons

- Can add *learning rate c* to speed up the learning process; just multiply in with delta computation
- Essentially a linear discriminant
- Perceptron theorem [Rosenblatt et al. 1958]: If a linear discriminant exists that can separate the classes without error, the training procedure is guaranteed to find that line or plane.



Ex: learning the logical OR

Examples in $\{0,1\}^2$, Perceptron inputs in $\{0,1\}^3$, first component (bias) $x_0 = 1$, two binary inputs: x_1 and x_2 . Weights initialisation : $w_0=0$ (corresponds to w_Θ) ; $w_1 = 1$ and $w_2 = -1$.

Example are always given in the same order (binary). c = 1.

Step	w_0	w_1	w_2	Input $(x_0, x_1, x_2)^T$	$\sum_0^2 w_i x_i$	Output u	True label y	w_0	w_1	w_2
init								0	1	-1
1	0	1	-1	100	0	0	0	0+0x1	1+0x0	-1+0x0
2	0	1	-1	101	-1	0	1	0+1x1	1+1x 0	-1+1x1
3	1	1	0	110	2	1	1	1	1	0
4	1	1	0	111	2	1	1	1	1	0
5	1	1	0	100	1	1	0	1+(-1)x1	1+(-1)x0	0+(-1)x0
6	0	1	0	101	0	0	1	0+1x1	1+1x0	0+1x1
7	1	1	1	110	2	1	1	1	1	1
8	1	1	1	111	3	1	1	1	1	1
9	1	1	1	100	1	1	0	1+(-1)x1	1+(-1)x0	1 +(-1)x0
10	0	1	1	101	1	1	1	0	1	1

No more changes from here… (so we can stop at the end of the epoch, 4 steps later)

Exclusive Or (XOR) Problem

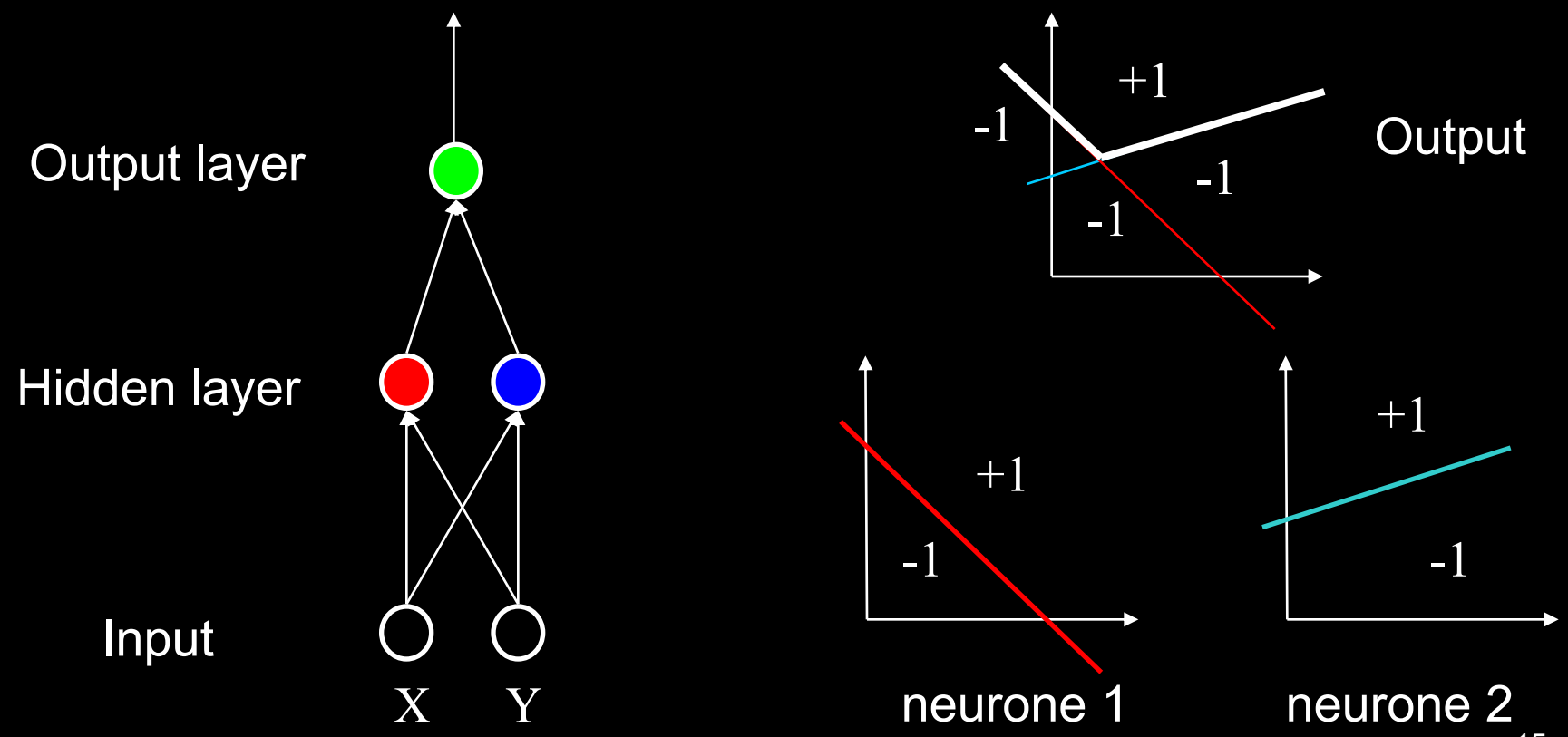
Input: 0,0 Output: 0
Input: 0,1 Output: 1
Input: 1,0 Output: 1
Input: 1,1 Output: 0

XOR Problem: Not Linearly Separable!
(cannot be learned by a perceptron)

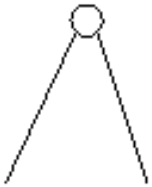
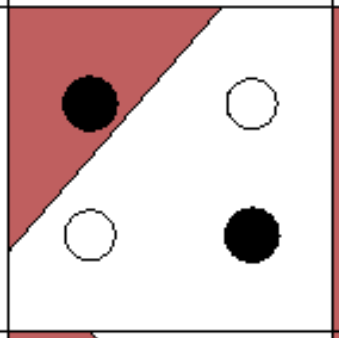
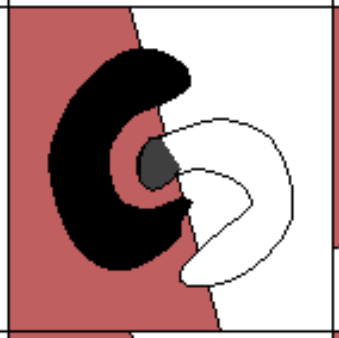
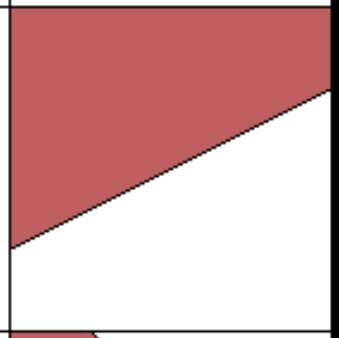
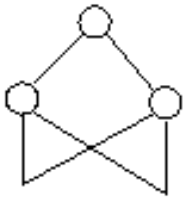
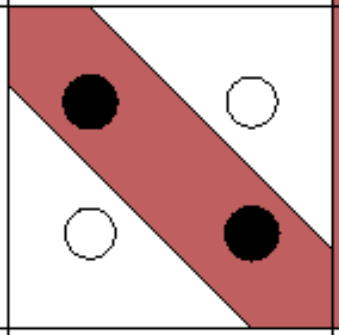
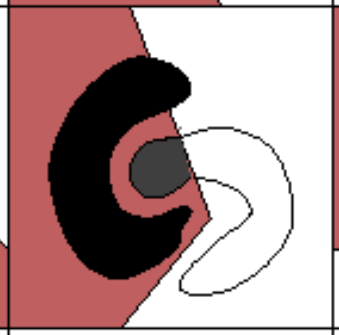
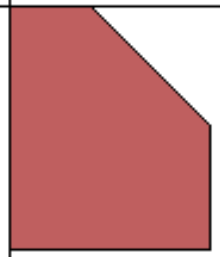
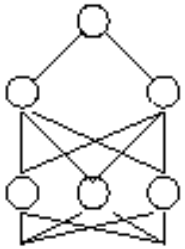
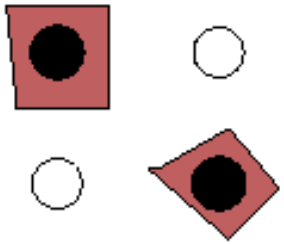
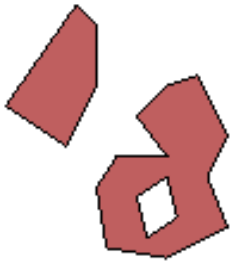
We could however construct multiple layers of perceptrons to get around this problem.

Multiple layers network

Can approximate any function f : $\Re^d \rightarrow \Re^C$



1 layer vs multiple layers

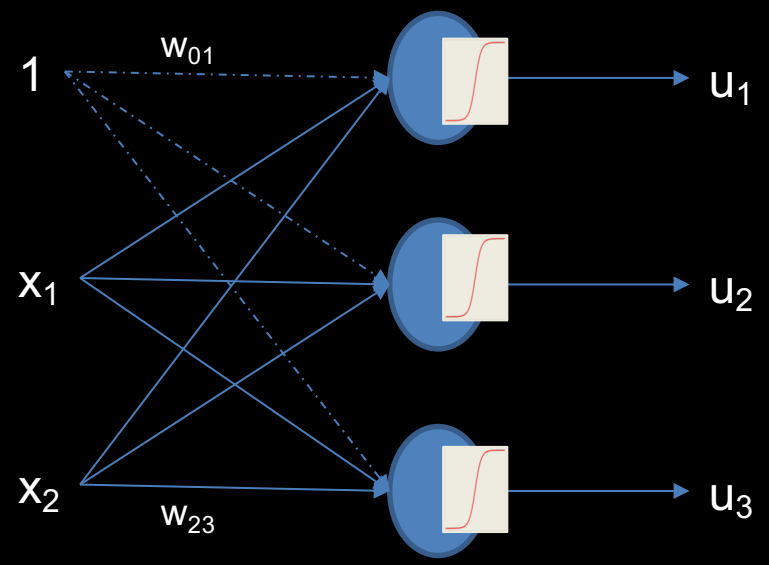
Structure	Régions décision-nelles	Pb du XOR	Régions pénét-rantes	Forme générale
 1 couche	Demi Plan			
 2 couches	Arbitraire dépend du nombre de couches cachées			
 3 couches	Arbitraire dépend du nombre de couches cachées			

Exercise
(« by hand / no learning »)

1. Draw (by hand, do not learn) a perceptron with 2 inputs which encodes the Boolean function: $A \wedge \neg B$ (give some possible weights w_A, w_B and w_0)

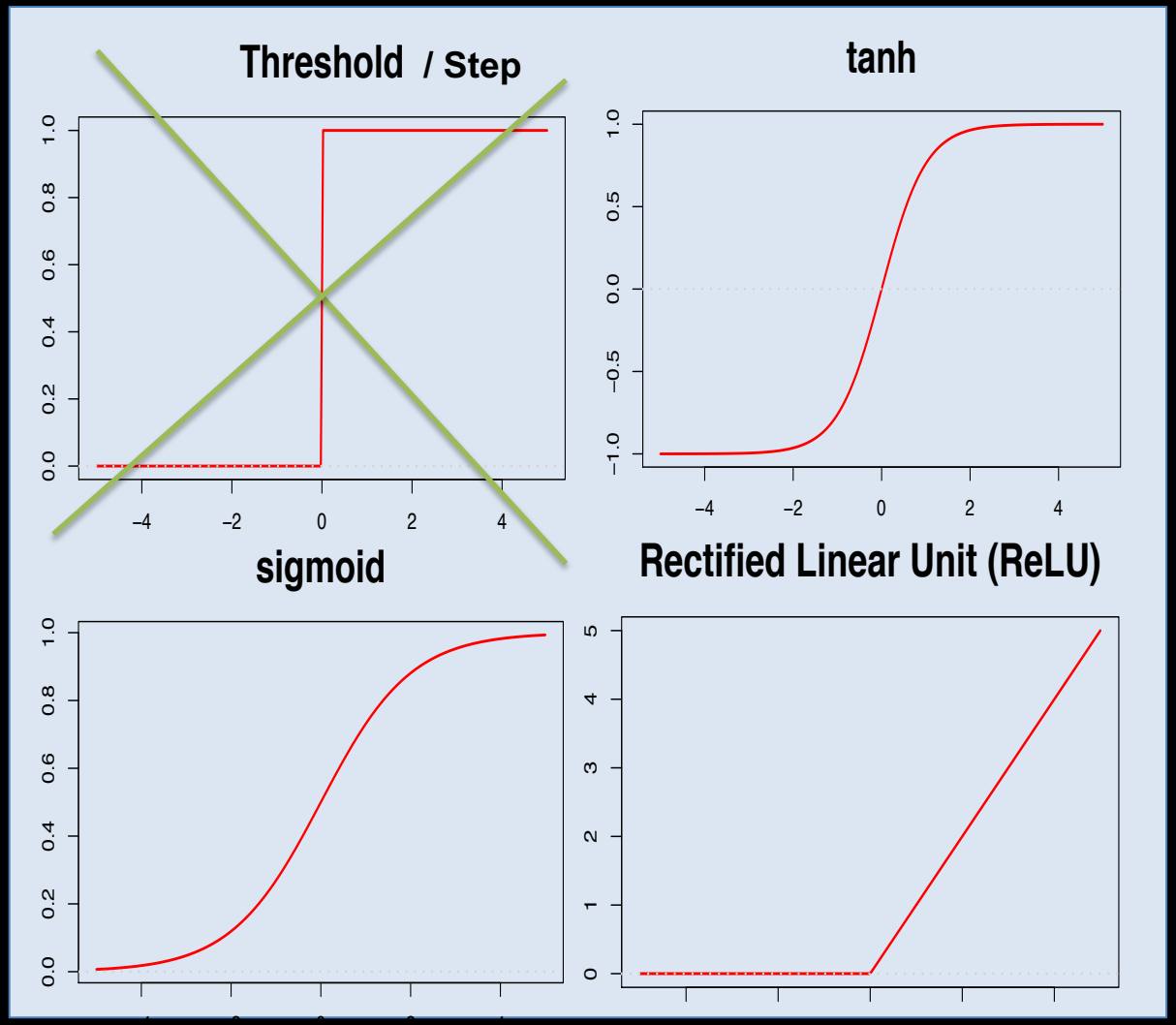
2. Draw a 2-layers perceptron to encode the Boolean function A XOR B (also give the weights in this case)

First layer of neurones



$$F(\mathbf{x},\mathbf{W}) = h(\mathbf{x}^t \times \mathbf{W}) = h\left(\begin{bmatrix}1\\x_1\\x_2\end{bmatrix}^t \times \begin{bmatrix}w_{01} & w_{02} & w_{03}\\ w_{11} & w_{12} & w_{13}\\ w_{21} & w_{22} & w_{23}\end{bmatrix}\right) = h\left(\begin{bmatrix}w_{01}+x_1w_{11}+x_2w_{21}\\ w_{02}+x_1w_{12}+x_2w_{22}\\ w_{03}+x_1w_{13}+x_2w_{23}\end{bmatrix}^t\right) = h\left(\begin{bmatrix}y_1\\y_2\\y_3\end{bmatrix}^t\right) = \begin{bmatrix}h(y_1)\\h(y_2)\\h(y_3)\end{bmatrix}$$

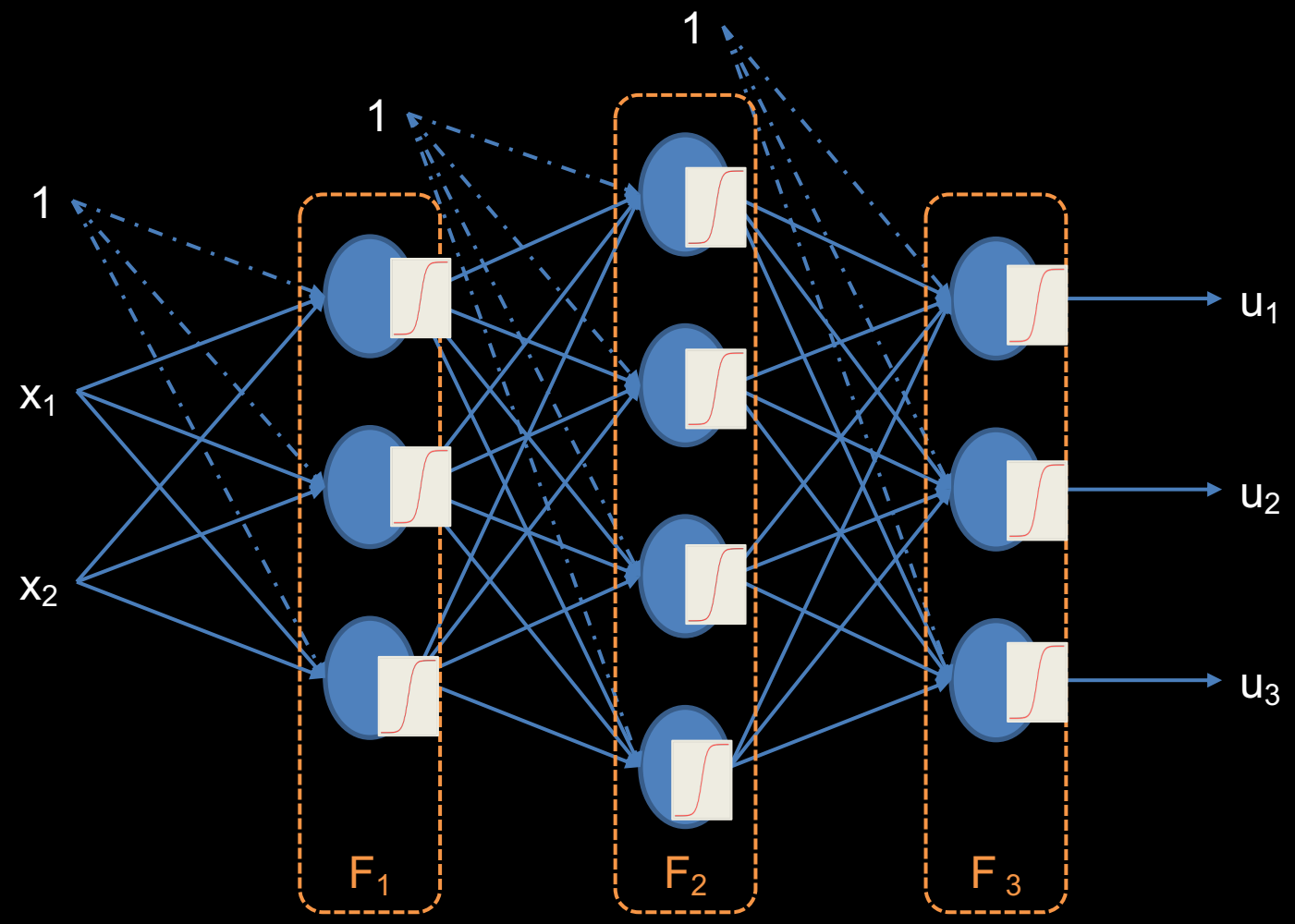
Activation function



- Threshold
- Tanh
- Sigmoïd
- Recitified Linear Unit (ReLU)= max(0,x)

- Leaky ReLU
- PReLU
- Etc…

MLP: multi layer perceptron



$$F_3(F_2(F_1(\mathbf{x},\mathbf{w}_1), \mathbf{w}_2),\mathbf{w}_3) = F_3(F_2(F_1(\mathbf{x})))) = F_3 \circ F_2 \circ F_1 (\mathbf{x})$$

What about changing the weights when there are many layers?

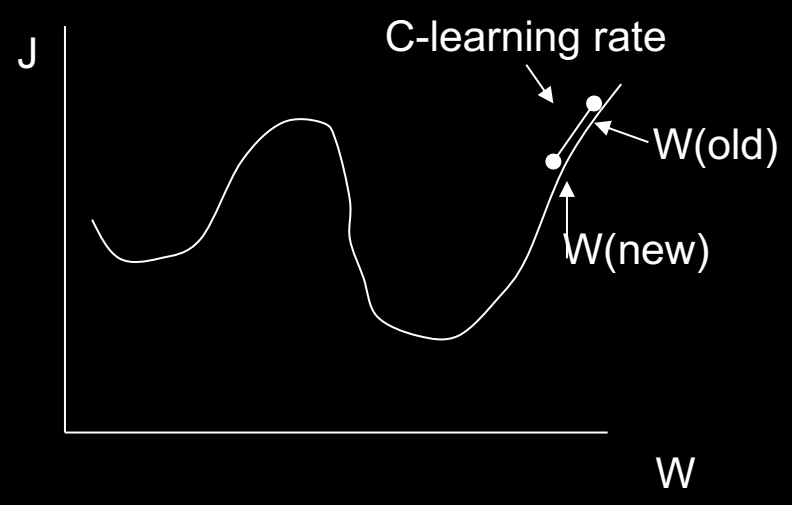
- Ex: MLP
- Loss function non convex (composition of functions with non linear components)
  - Gradient descent can deal with that (to find a local minimum)
- Need to propagate the error at the network's output to all the neurones
  - Backpropagation (attributed to Rumelhart and McClelland, late 70's)

General learning rule

We want to minimize the loss!

$$\Delta w_i^t = -c \cdot \frac{\partial J(W, y)}{\partial w_i^t}$$

$$w_i^{t+1} = w_i^t + \Delta w_i^t$$

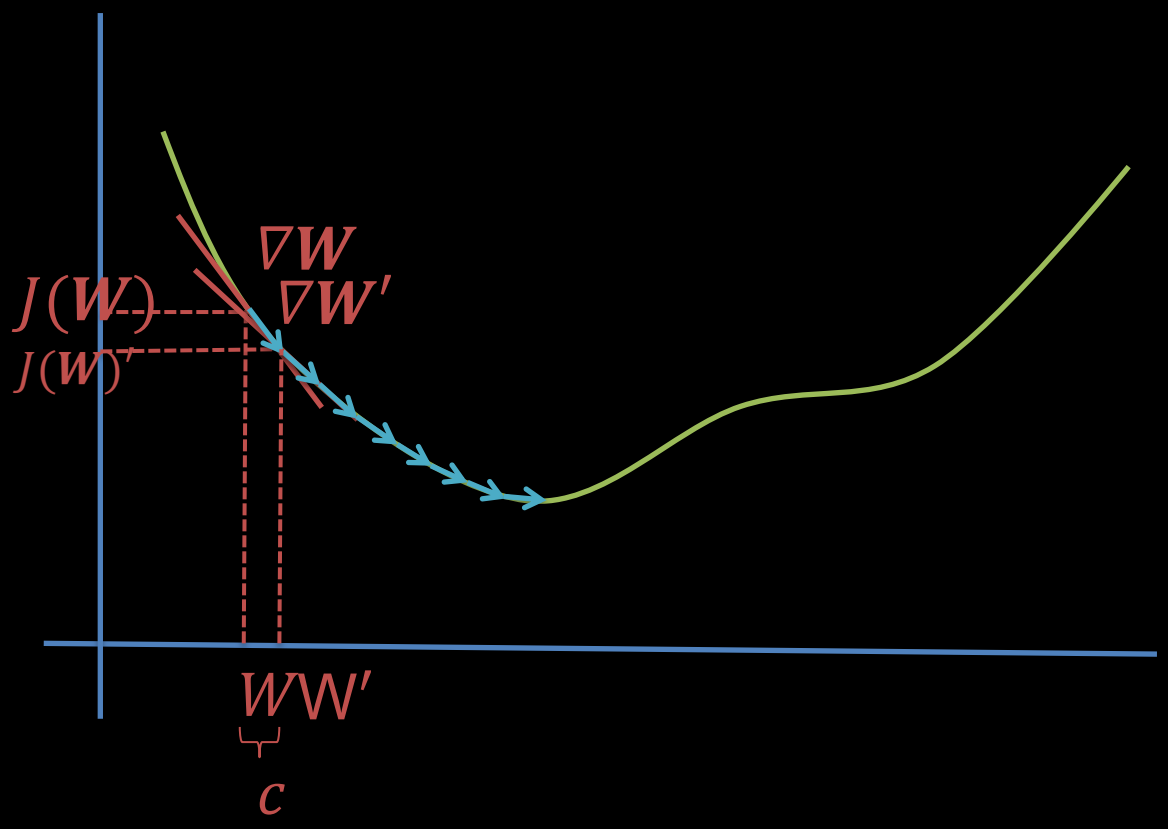


c (or η) is the learning rate parameter (can be a constant, set by the optimizer)

$$\Delta w_i^t = -c \cdot \frac{\partial J(W, y)}{\partial w_i^t}$$

$$w_i^{t+1} = w_i^t + \Delta w_i^t$$

Gradient descent



Objective: minimizing an objective (loss) function

Gradient gives the slope of the function

Updating the parameters in the opposite direction of the gradient according to a learning rate

Repeat until convergence

Which loss? …in multiclass classification
multiclass cross entropy (or neg log likelihood)

(x,**y**) an instance, y: one-hot vector of classes, n: nb of examples, C: nb of classes; **u**: output of the network for x; **p**: vector of predicted class probabilities **t**: index of the « 1 » in y

SOFTMAX (layer):
convert the network outputs into probabilities

$$p = soft\max(u) = \left[\frac{e^{u_j}}{\sum_{k=1}^{c} \left[e^{u_k} \right]} \right]_{j=1..C}$$

Multiclass Cross entropy loss
(or neg log likelihood):

$$CLoss(p,y) = -\sum_{j=1}^{c} \left[y_j \ln p_j \right]$$

Combinaison of both:

$$J(\mathbf{W},\mathbf{y}) = J(u,t) = -u_t + \ln\left(\sum_{j=1}^{c} \left[e^{u_j} \right]\right)$$

http://peterroelants.github.io/posts/neural_network_implementation_intermezzo02/

Example

Network output u:

-2	3	1

Class probabilities:

0.0058	0.87	0.118

y = [0.0,1.0,0.0]

CLoss(p,y) = 0.1328

J(u,1) = 0.1328

```
import numpy as np

def softmax(x):
        return np.exp(x) /
np.sum(np.exp(x))

u = [-2.0,3.0, 1.0]
print(softmax(scores))
[ 0.00589975  0.8756006   0.11849965]

def crossEnt(p,y):
        return
np.sum(y*np.log(p))

def mycrossEnt(u,t):
        return  -u[t]+
np.log(np.sum(np.exp(u)))
```

Which loss? … in regression

A set of N training examples N

P outputs. u_p is the raw output (for output p), as calculated by the network.

$$E = \frac{1}{2}\sum_{P}\left(u_P - y_P\right)^2, MSE = \frac{1}{2N}\sum_{N}\sum_{P}\left(u_P - y_P\right)^2$$

E.g. if we have one example (n=1) and

$u = (u_1,u_2)^T = (1,0)$ and $y = (y_1,y_2)^T = (0.8,0.5)$

then

$E= (0.5)* [(1-0.8)^2+(0-0.5)^2] = 0.145$

NB: this loss can also be used for classification but this is less common

Train complex (multilayer) feedforward networks

Learning Procedure:

1. Randomly assign weights (within a “reasonable” range)
2. Present inputs from training data, propagate to outputs (= feedforward pass)
3. Compute outputs u, adjust weights according to the delta rule, backpropagating the errors. The weights will be nudged closer so that the network learns to give the desired output.
4. Repeat from 2; stop when no errors, or enough epochs completed

Reminder: Chain rule

Composition function:

$$F(x) = (\mathrm{f} \circ g)(x) = f(g(x))$$

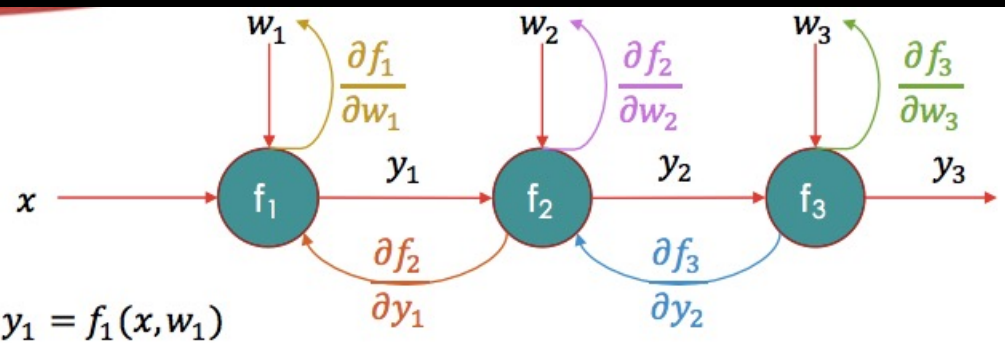
Derivative of a composition function:

$$F'(x) = (f' \circ g)(x) \times g'(x) = f'\big(g(x)\big) \times g'(x)$$

Using Leibniz's notation:

$$F'(x) = \frac{\partial F(x)}{\partial x} = \frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \times \frac{\partial g(x)}{\partial x}$$

Back-propagation



$$y_1 = f_1(x, w_1)$$

$$y_2 = f_2(y_1, w_2) = f_2(f_1(x, w_1), w_2)$$

$$y_3 = f_3(y_2, w_3) = f_3(f_2(y_1, w_2), w_3) = f_3(f_2(f_1(x, w_1), w_2), w_3)$$

Objective: $\nabla_{w_1, w_2, w_3} y_3 = \nabla_{w_1} y_3; \nabla_{w_2} y_3; \nabla_{w_3} y$

$$\nabla_{w_3} y_3 = \frac{\partial f_3(y_2, w_3)}{\partial w_3}$$

$$\nabla_{w_2} y_3 = \frac{\partial f_3(y_2, w_3)}{\partial y_2} \times \frac{\partial f_2(y_1, w_2)}{\partial w_2}$$

$$\nabla_{w_1} y_3 = \frac{\partial f_3(y_2, w_3)}{\partial y_2} \times \frac{\partial f_2(y_1, w_2)}{\partial y_1} \times \frac{\partial f_1(x, w_1)}{\partial w_1}$$

$$\nabla_{w_3} y_3 = \frac{\partial y_3}{\partial w_3} = \frac{\partial f_3(y_2, w_3)}{\delta w_3}$$

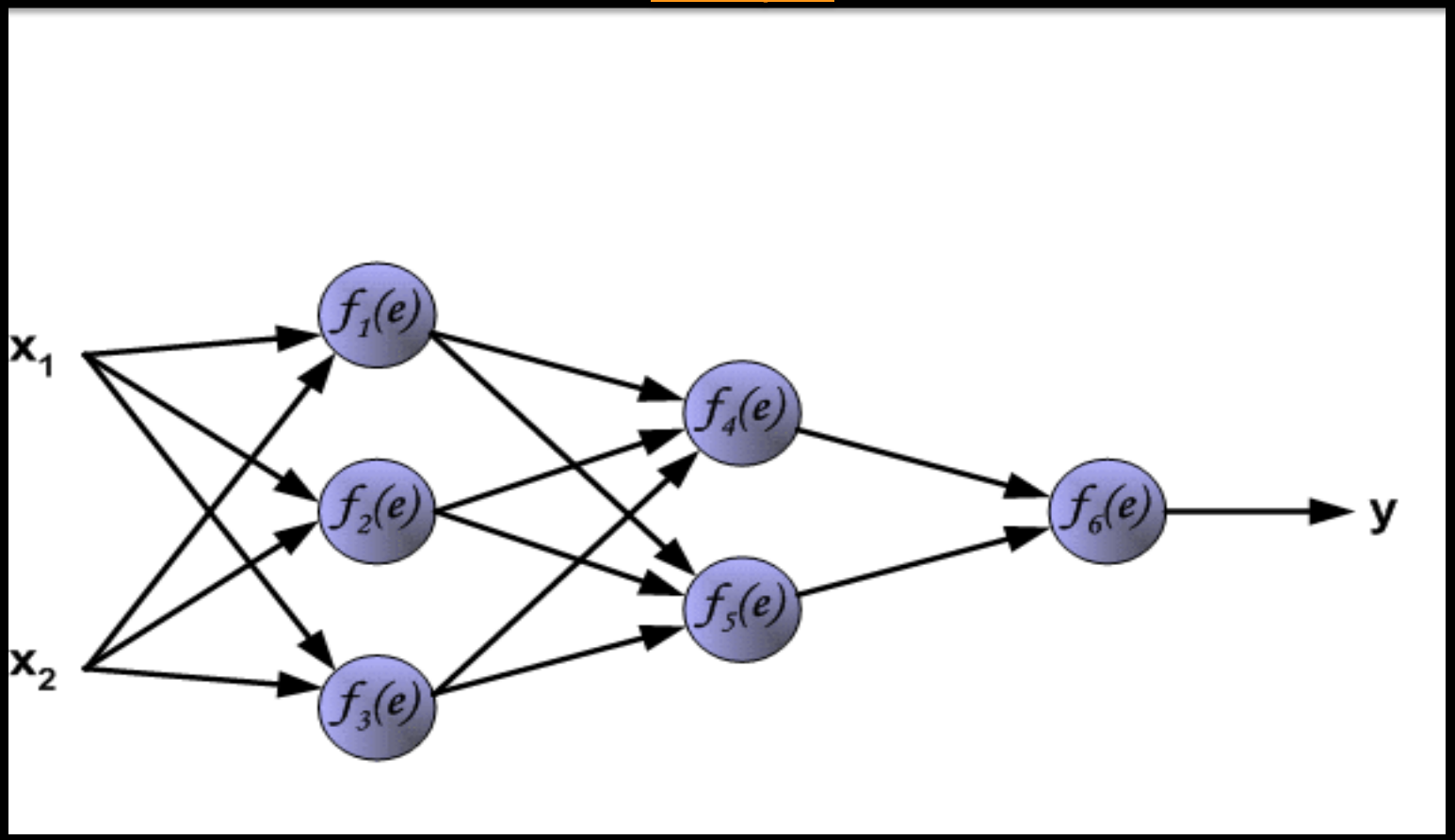
$$\nabla_{w_2} y_3 = \frac{\partial y_3}{\partial w_2} = \frac{\partial f_3(y_2, w_3)}{\partial w_2} = \frac{\partial f_3(y_2, w_3)}{\partial y_2} \times \frac{\partial y_2}{\partial w_2} = \frac{\partial f_3(y_2, w_3)}{\partial y_2} \times \frac{\partial f_2(y_1, w_2)}{\partial w_2}$$

$$\nabla_{w_1} y_3 = \frac{\partial y_3}{\delta w_1} = \frac{\partial f_3(y_2, w_3)}{\partial w_1} = \frac{\partial f_3(y_2, w_3)}{\partial y_2} \times \frac{\partial y_2}{\partial w_1} = \frac{\partial f_3(y_2, w_3)}{\partial y_2} \times \frac{\partial f_2(y_1, w_2)}{\partial w_1} = \frac{\partial f_3(y_2, w_3)}{\partial y_2} \times \frac{\partial f_2(y_1, w_2)}{\partial y_1} \times \frac{\partial y_1}{\partial w_1}$$

A lot more complex than for the simple perceptron !

Backprop: Computation flow

Try it: http://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/



Ex: Gradient computation if Loss = MSE

To compute how much to change weight for **link k**:

$$\Delta w_k = -c\frac{\delta Error}{\delta w_k}$$

$$u_j = f(X^T W)$$

$$\frac{\delta u_j}{\delta w_k} = x_k f'\left(X^T W\right)$$

Chain rule:

$$\frac{\delta Error}{\delta w_k} = \frac{\delta Error}{\delta u_j} * \frac{\delta u_j}{\delta w_k}$$

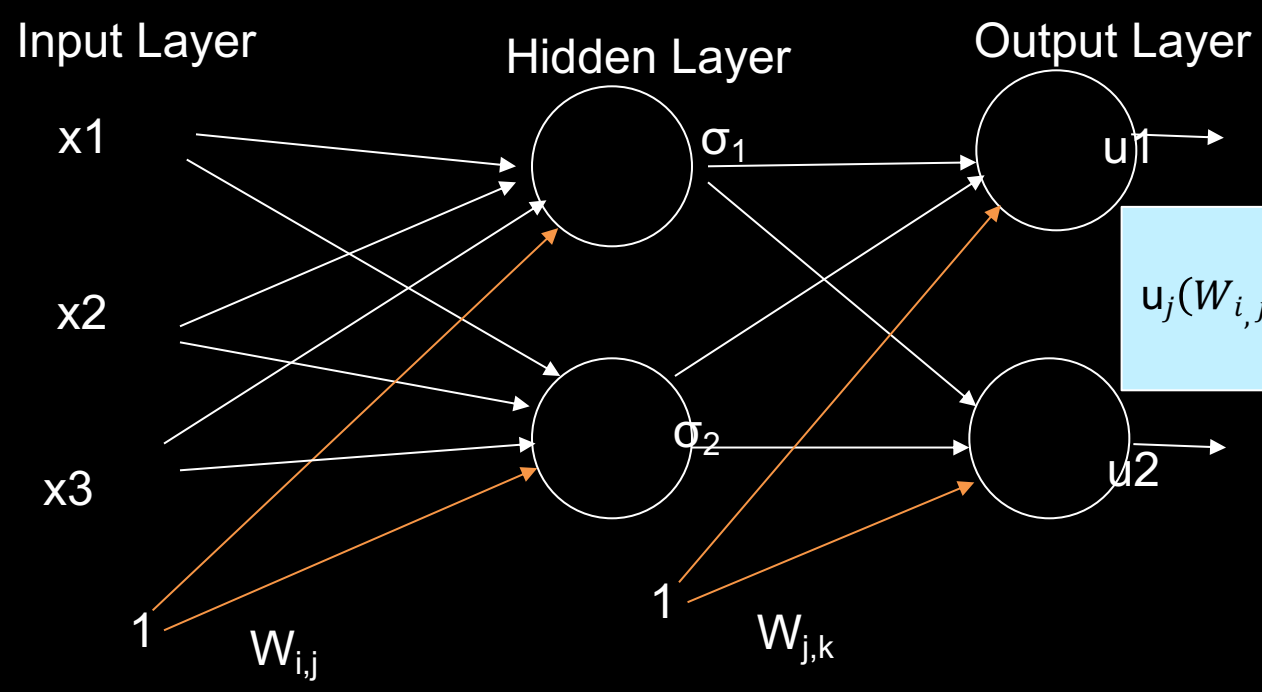
$$\Delta w_k = -c\left(-(y_j - u_j)\right)x_k f'\left(X^T W\right)$$

$$= c\left(y_j - u_j\right)x_k f'\left(X^T W\right)$$

$$\frac{\delta Error}{\delta u_j} = \frac{\delta \frac{1}{2}\sum_P (y_P - u_P)^2}{\delta u_j} = \frac{\delta \frac{1}{2}(y_j - u_j)^2}{\delta u_j} = -(y_j - u_j)$$

We can remove the sum since we are taking the partial derivative w.r.t u_j

Ex: backpropagation
(with activation = sigmoid and loss = MSE)



$$u_j(W_{i,j}, \sigma i) = \frac{1}{1+e^{-(\sum_i W_{i,j} * \sigma_i)}}$$

1's - bias

$$\sigma_j(W_{i,j}, xi) = \frac{1}{1+e^{-(\sum_i W_{i,j} * x_i)}}$$

$$f(x) = \left(\frac{1}{1+e^{-x}}\right)$$

Les maths qui manquent…

https://ia802306.us.archive.org/30/items/c-72_20211011/C72.pdf (lire l'Annexe 9 page 778/834)

Pour plus de clarté, nous pouvons omettre l'indice x dans les équations qui viennent. Nous définissons, pour chaque neurone j, la quantité :

$$\delta_j = \frac{\partial E}{\partial \sigma_j}$$

Le cas des neurones de sortie

Comme E dépend de $w(i,j)$ seulement par l'intermédiaire de σ_j, la formule des dérivations partielles chaînée permet d'écrire[3] :

$$\frac{\partial E}{\partial w(i,j)} = \frac{\partial E}{\partial \sigma_j} \cdot \frac{\partial \sigma_j}{\partial w(i,j)} = \delta_j \cdot y_i$$

Lorsque j est un neurone de sortie, avec f définie comme dans l'équation (22.21) (et λ égal à 1), on a, en remarquant que y_j ne dépend que de σ_j :

$$\delta_j = \frac{\partial E}{\partial \sigma_j} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial \sigma_j}$$

D'après les équations 22.20 et 22.22 :

$$\frac{\partial y_j}{\partial \sigma_j} = y_j \cdot (1 - y_j)$$

Puisque j est un neurone de sortie, d'après l'équation 22.23 :

$$\frac{\partial E}{\partial y_j} = y_j - u_j$$

Donc, pour un neurone de sortie :

$$\delta_j = y_j \cdot (1 - y_j) \cdot (y_j - u_j)$$

et finalement :

$$\frac{\partial E}{\partial w(i,j)} = y_j \cdot (1 - y_j) \cdot (y_j - u_j) \cdot y_i$$

Le cas des neurones cachés

Lorsque j est un neurone caché, on calcule δ_j par *rétropropagation*. La remarque fondamentale est que la contribution de chaque neurone formel j sur la sortie est propagée vers la sortie à travers les éléments de la couche $dest(j)$.

Comme E dépend de $w(i,j)$ seulement par l'intermédiaire de σ_j, la formule des dérivations partielles chaînée permet d'écrire :

$$\frac{\partial E}{\partial w(i,j)} = \frac{\partial E}{\partial \sigma_j} \cdot \frac{\partial \sigma_j}{\partial w(i,j)} = \delta_j \cdot y_i$$

Comme E dépend de tous les états σ_k, pour $k \in dest(j)$, et que chaque σ_k dépend (en ce qui concerne les variables qui nous intéressent) seulement de y_j (et donc seulement de σ_j), on peut aussi écrire :

$$\frac{\partial E}{\partial w(i,j)} = \sum \frac{\partial E}{\partial \sigma_k} \cdot \frac{\partial \sigma_k}{\partial \sigma_j} \cdot \frac{\partial \sigma_j}{\partial w(i,j)}$$

En sortant le terme non dépendant de k de la somme dans la seconde expression de $\frac{\partial E}{\partial w(i,j)}$, en égalant les deux expressions, et en utilisant la définition de δ_j et de δ_k, on obtient :

$$\delta_j = \sum_{k \in dest(j)} \delta_k \cdot \frac{\partial \sigma_k}{\partial \sigma_j}$$

Cela peut s'interpréter par le fait que la « responsabilité » δ_j de la cellule j dans l'erreur est propagée à ses successeurs δ_k avec un poids $w(j,k)$.

On peut maintenant terminer les calculs :

$$\begin{aligned}
\delta_j &= \sum_{k \in dest(j)} \delta_k \cdot \frac{\partial \sigma_k}{\partial \sigma_j} \\
&= \sum_{k \in dest(j)} \delta_k \cdot \frac{\partial \sigma_k}{\partial y_j} \cdot \frac{\partial y_j}{\partial \sigma_j} \\
&= \sum_{k \in dest(j)} \delta_k \cdot w(j,k) \cdot y_j \cdot (1 - y_j) \\
&= y_j \cdot (1 - y_j) \cdot \sum_{k \in dest(j)} \delta_k \cdot w(j,k)
\end{aligned}$$

et finalement :

$$\frac{\partial E}{\partial w(i,j)} = y_i \cdot y_j \cdot (1 - y_j) \cdot \sum_{k \in dest(j)} \delta_k \cdot w(j,k)$$

Conclusion

En résumant, on a donc les valeurs :

- Pour les neurones de sortie :

$$\frac{\partial E}{\partial w(i,j)} = y_i \cdot \delta_j = y_i \cdot y_j \cdot (1 - y_j) \cdot (y_j - u_j)$$

- Pour les neurones cachés :

$$\frac{\partial E}{\partial w(i,j)} = y_i \cdot \delta_j = y_i \cdot y_j \cdot (1 - y_j) \cdot \sum_{k \in dest(j)} \delta_k \cdot w(j,k)$$

Finalement, pour modifier $w(i,j)$, il faut lui additionner une quantité dans la direction opposée au gradient :

$$\Delta w(i,j) = -\alpha \frac{\partial E}{\partial w(i,j)} = -\alpha \cdot y_i \cdot \delta_j$$

où α, compris entre 0 et 1 est le pas de déplacement, aussi appelé le taux d'apprentissage.

10. L'analyse de l'induction de Vapnik

10.1 Cas où $|\mathcal{H}| = \infty$ et $\mathcal{F} \subseteq \mathcal{H}$

Dans le cas où le nombre d'hypothèses est fini, on comprend que l'on puisse borner la pro-

Backprop - Modifying Weights (for sigmoid activation)

We had computed:

$$\Delta w_k = c * x_k \left(y_j - u_j\right) f'\left(X^T W\right)$$

$$\Delta w_k = c * x_k \left(y_j - u_j\right)\left(f(X^T W)(1 - f(X^T W)\right)$$

$$f(x) = \left(\frac{1}{1+e^{-x}}\right)$$

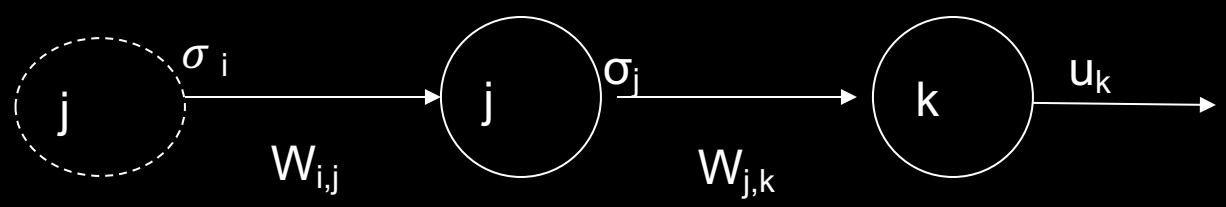
$$f'(x) = \left(\frac{-(-e^{-x})}{(1+e^{-x})\wedge 2}\right) = f(x) * (1 - f(x))$$

For the output units k, $f(X^T W) = u_k$. So, for the output units, the learning rule is:

$$\Delta w_{j,k} = c * \sigma_j * \overbrace{(y_k - uk)u_k(1 - uk)}^{\delta_k}$$

For the Hidden units (skipping some math), this is:

$$\Delta w_{i,j} = c * \sigma_i * \overbrace{\sigma_j(1 - \sigma_j) * \textstyle\sum_k (y_k - u_k)u_k(1 - u_k)w_{j,k}}^{\delta_j}$$



Recap (for the previous setting)

- To change the weights going from neuron i to neuron j (c =learning rate): $\Delta w_{i,j} = c * \delta_j * x_i$

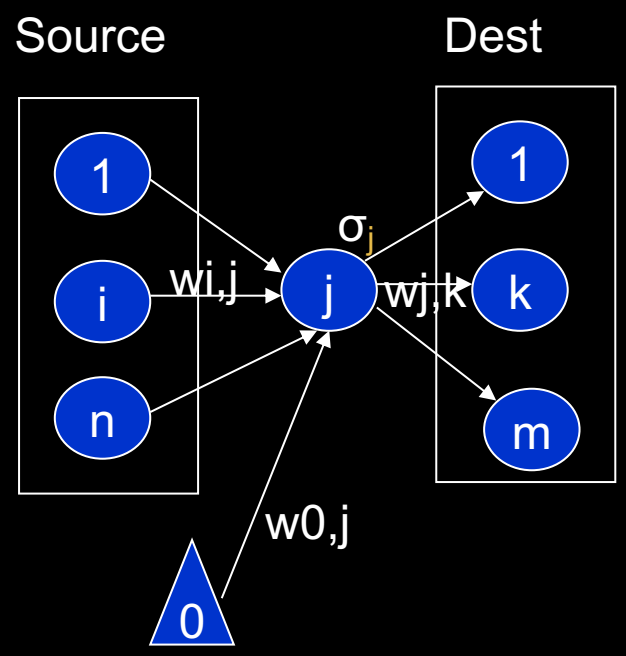
= change proportional to the error δj measured at neuron j and to the input value x_i

- Error at each neurone u_j of the output layer:

$$\delta_j = (y_j - u_j)\, u_j\, (1-u_j)$$

- Error in the output of each neuron in a hidden layers Hj which compute σ_j computed recursively using gradient descent:

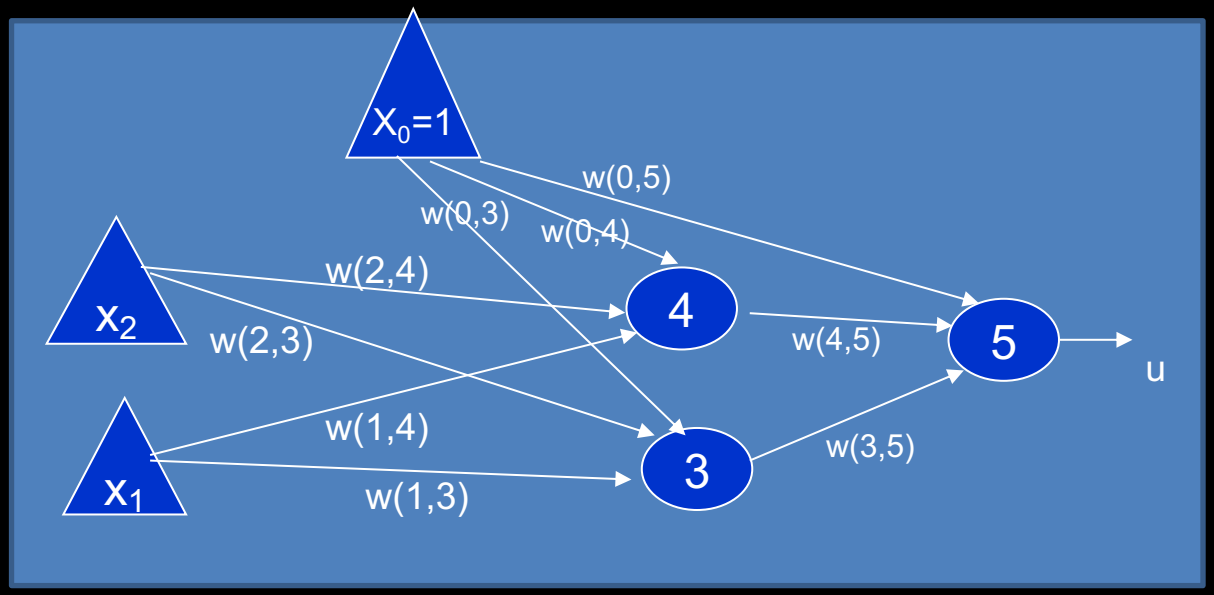
$$\delta_j = \sigma_j\, (1-\sigma_j) \sum_{k \in dest(j)} \delta_k\, w_{j,k}$$



Exercise 2 (use previous slide)

For an input vector X^T =(x1, x2) = (1,1) , **c= 1**, y = 0 and using a sigmoid transfer/activation function and a MSE loss:

- Compute the output of each neuron
- Compute the new weights and the new output after ONE back-propagation step (using the formula of the previous slide)



w(0,3) = 0,2

w(0,4) = -0,3

w(0,5) = 0,4

w(1,3) = 0,1

w(1,4) = -0,2

w(3,5) = 0,5

w(2,3) = 0,3

w(2,4) = 0,4

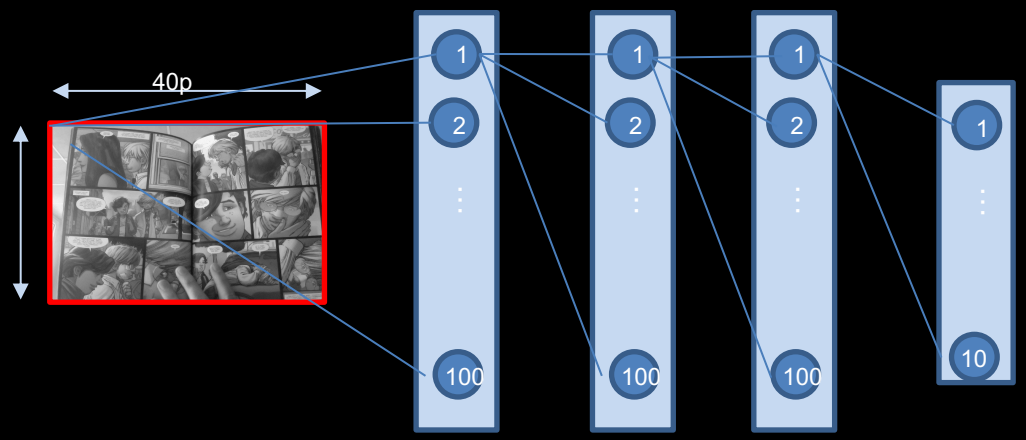
w(4,5) = -0,4

Exercise: count # parameters

- Suppose that you have built a MLP architecture (by default fully connected) with 3 hidden layers that contain each 100 nodes. Your inputs are images of size 40*40 pixels (black and white) and you want to predict 10 classes.

40p

- How many parameters should your network learn?



Batch vs Stochastic vs Mini-Batch Gradient Descent

Batch: compute the gradient of the cost function for the entire dataset:

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$$

SGD: parameter update for *each* training example ($x^{(i)}$, $y^{(i)}$):

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta, x^{(i)}, y^{(i)})$$

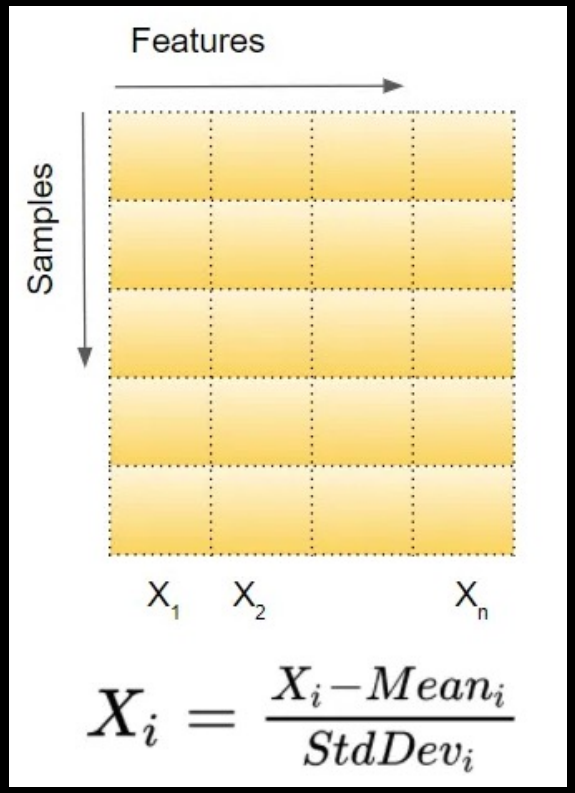
Mini batch: performs an update for every mini-batch of n training examples:

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta, x^{(i:i+n)}, y^{(i:i+n)})$$

Batch normalization (1/4)

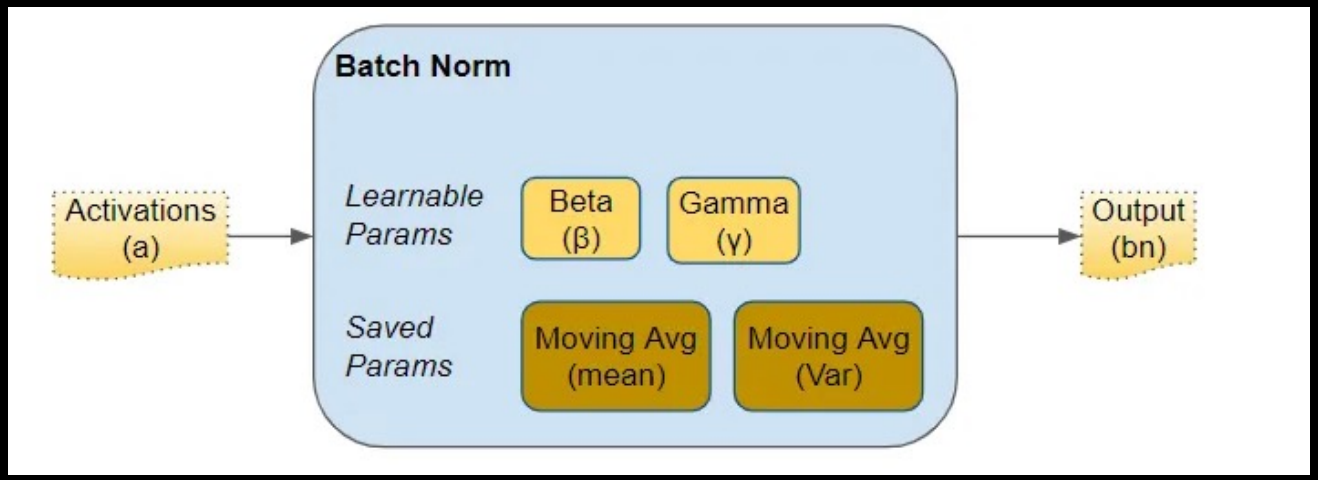
- Standard practice to normalize the data to **zero mean and unit variance**
- The same logic that requires us to normalize the input for the first layer will also apply to each of these hidden layers ➔ batch norm!

Normalize the activations from each previous layer so that the gradient descent will converge better during training.



https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

Batch normalization (2/4)

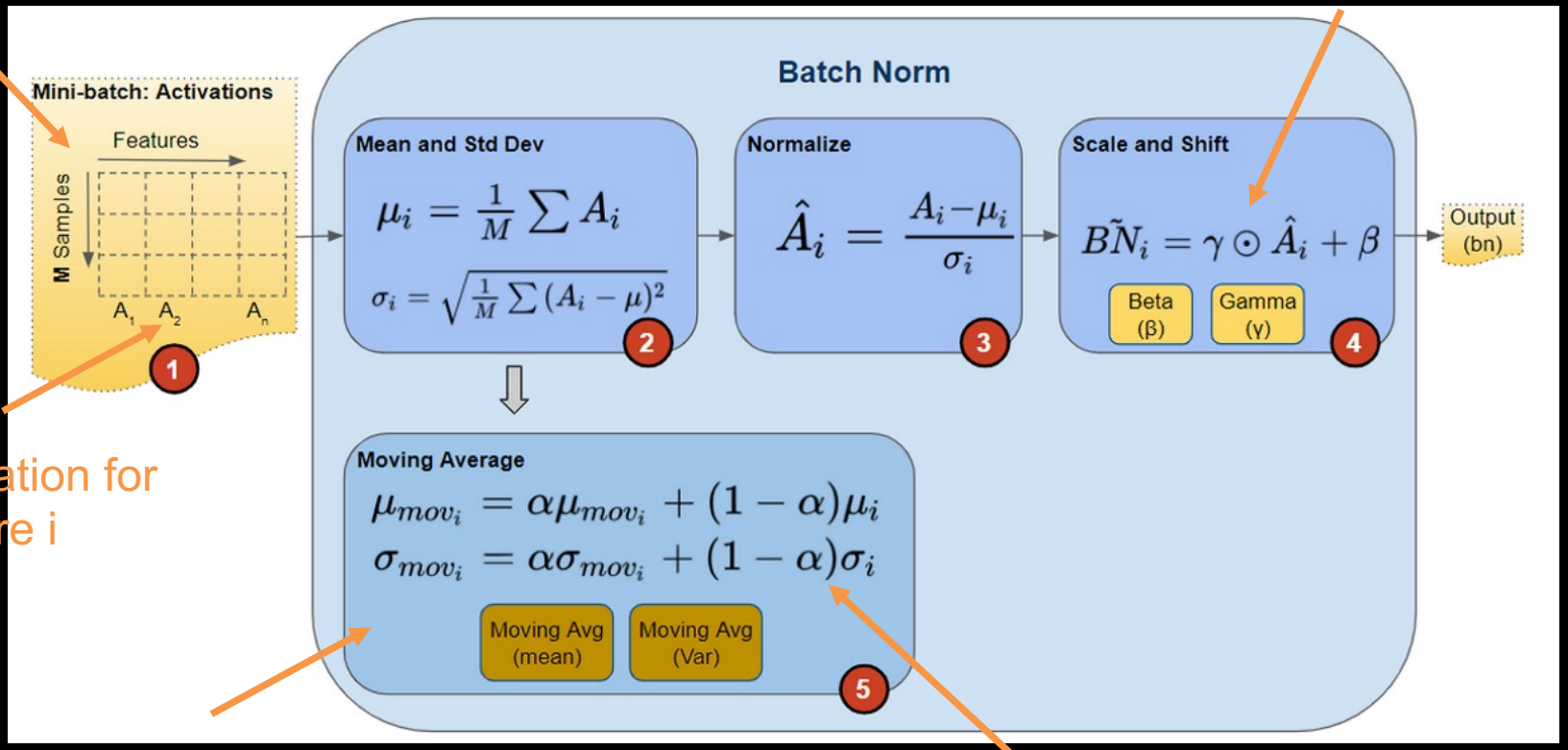


- Another network layer that gets inserted between a hidden layer and the next hidden layer
- Just like the parameters (eg. weights, bias) of any network layer, a Batch Norm layer also has parameters of its own:
  - Two learnable parameters called beta and gamma (per Batch Norm layer), not hyperparameters
  - Two non-learnable parameters (Mean Moving Average and Variance Moving Average) are saved as part of the 'state' of the Batch Norm layer.
- Can be put before or after the activation layers (it depends…)

Batch normalization (3/4) Training

M samples in the minibatch

Element-wise multiply, not a matrix multiply

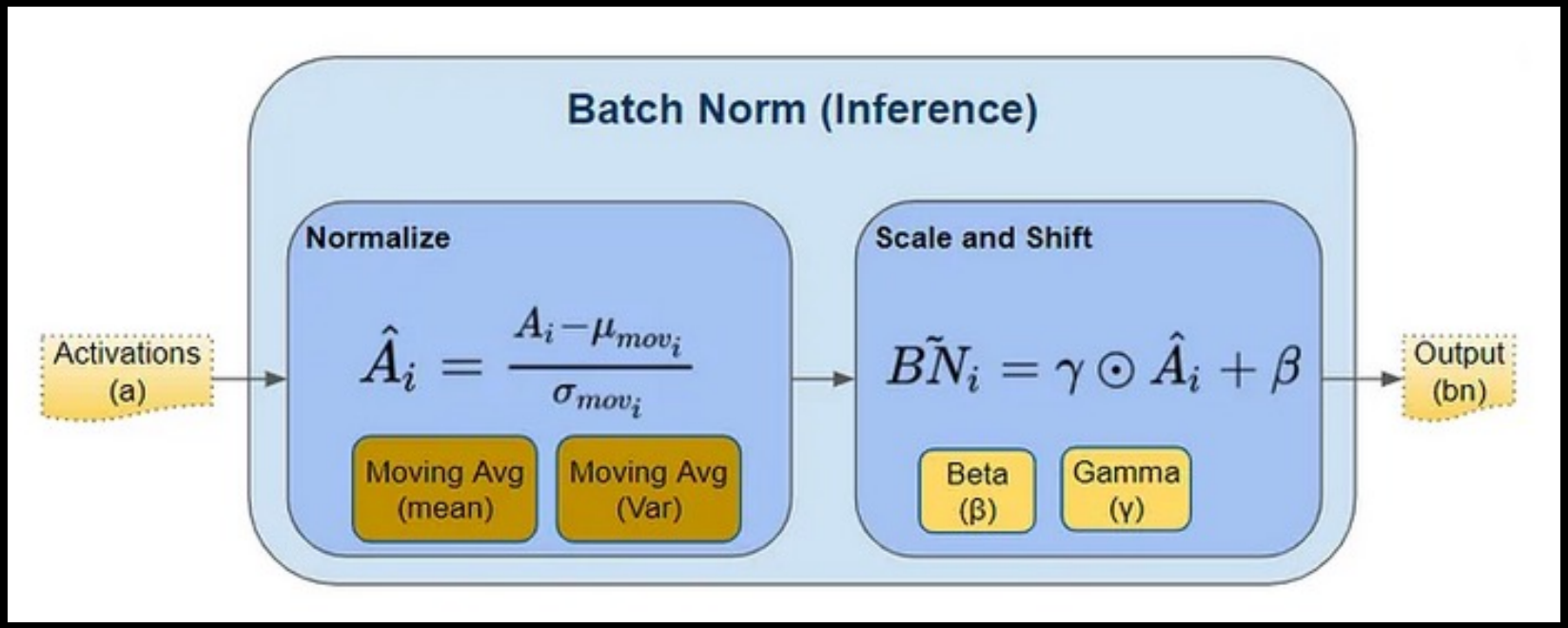


Activation for feature i

Used only at Inference (not at training time)

Scalar ‘momentum’ hyperpameter ≠ momentum of the optimizer

Batch normalization (4/4) Inference



- Moving Average acts as a good proxy for the mean and variance of the data (but with incremental computation)

Dropout

During **training**, for each forward pass, randomly set units to 0.

13	9	7	4
4	16	2	5
0	6	8	19
4	7	7	5

Input

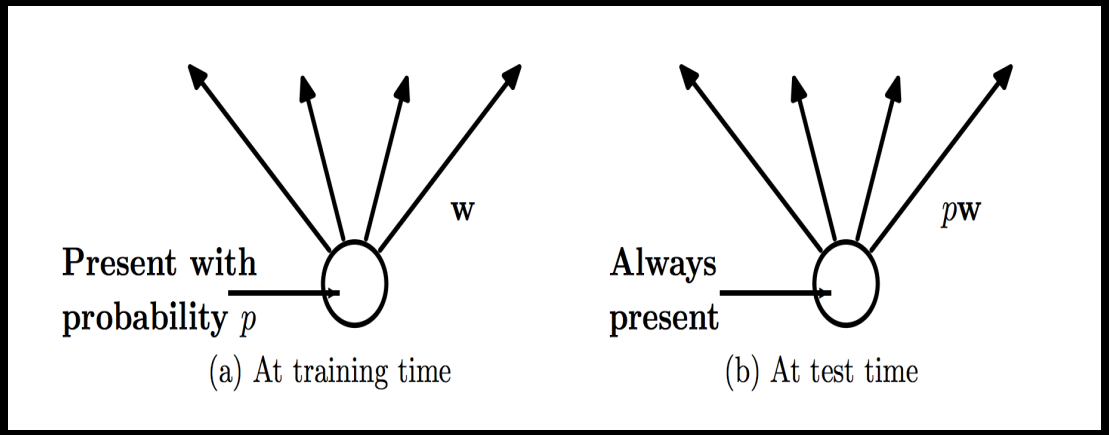
Dropout

Drop factor =0.3

13	9	7	4
4	0	2	0
0	6	8	0
0	0	7	5

Output

At **test** time, keep the same « energy » into the network



Equivalent to train all possible networks at the same time in training, and averaging them out in testing.

Dropout vs BatchNorm

- Dropout: strong regularizer
- BatchNorm: less regularization, more popular but not usable in sequence models (RNN)
- Effect a bit redundant: should not be used at the same layer

- "Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift": https://openaccess.thecvf.com/content_CVPR_2019/papers/Li_Understanding_the_Disharmony_Between_Dropout_and_Batch_Normalization_by_Variance_CVPR_2019_paper.pdf

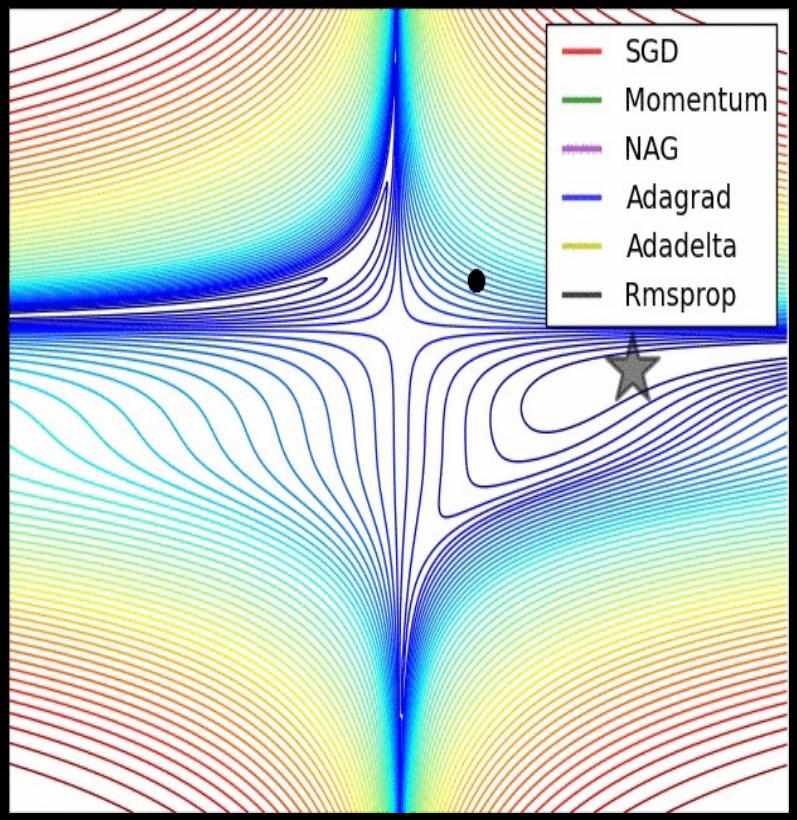
Different SGD optimization algorithms

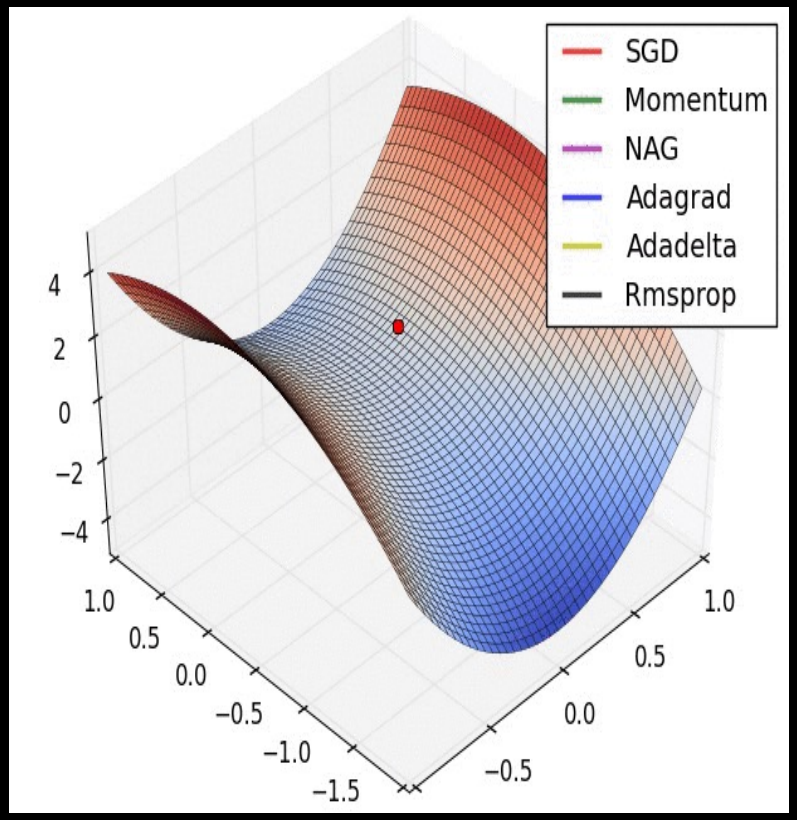
Momentum: adds a fraction of the previously computed gradient (gives inertia to the gradient)

$$v_t \leftarrow \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta \leftarrow \theta - v_t$$

NAG: extension of momentum
Adagrad: adapts the learning rate to each parameters individually
Adadelta: extension of Adagrad
RMSprop: another extension of Adagrad
Adam: takes into account the mean and variance of gradients
Etc…

Gradient descent illustration





See:
http://sebastianruder.com/optimizing-gradient-descent/index.html#whichoptimizertouse

Main drawbacks of NN

- The structure of the networks is not learned, it is usually set by test and trial
  - In practice a small amount of layers is enough (too many is also harmful)
- The initialization of the weights has a great impact on the results
  - nitialize the weights on unlabeled data using autoencoders or RBM ?
  - Random between [-1;1]
- Too many other hyper parameters
  - SGD or not, optimizers, ….