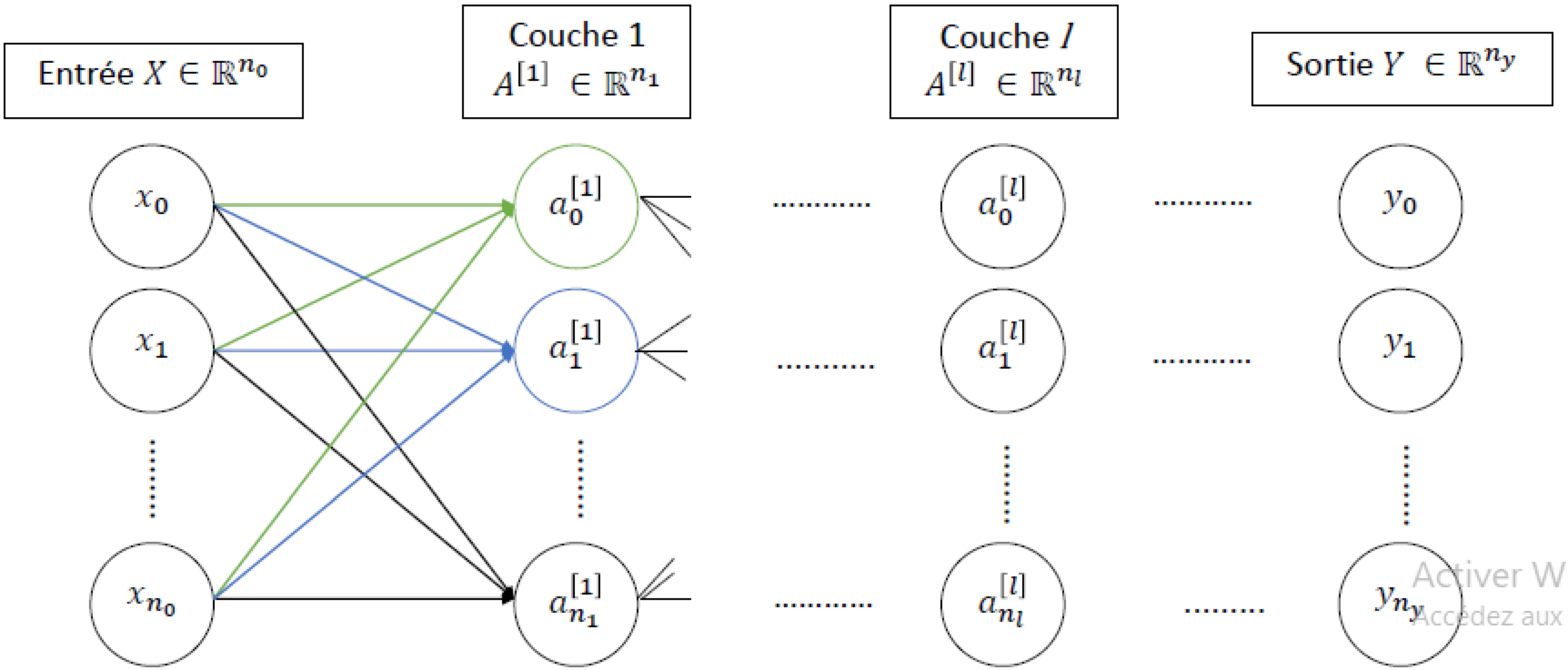
Année Universitaire 2021-2022

Apprentissage de réseaux : Perceptron Multi-Couches

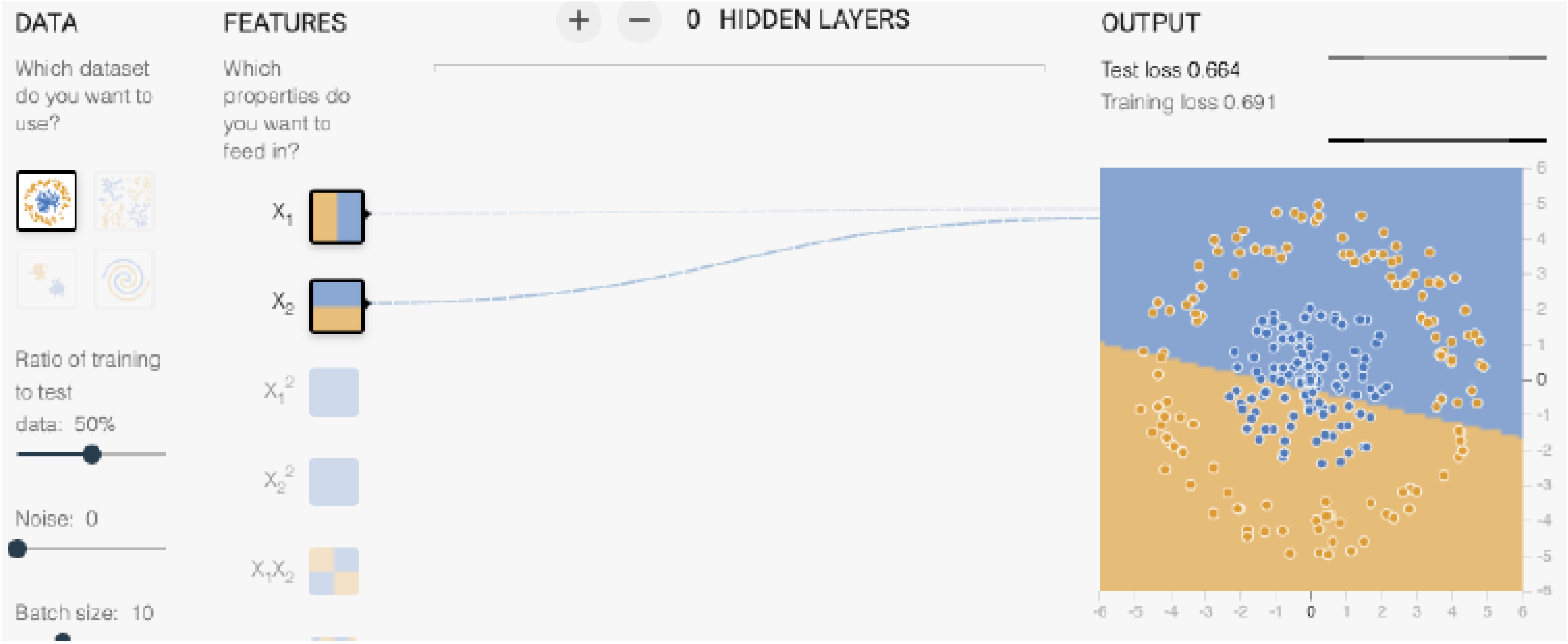
Mhamdi Farouk

Architecture et notations :



Exemple :Séparation 2-D (1/3)

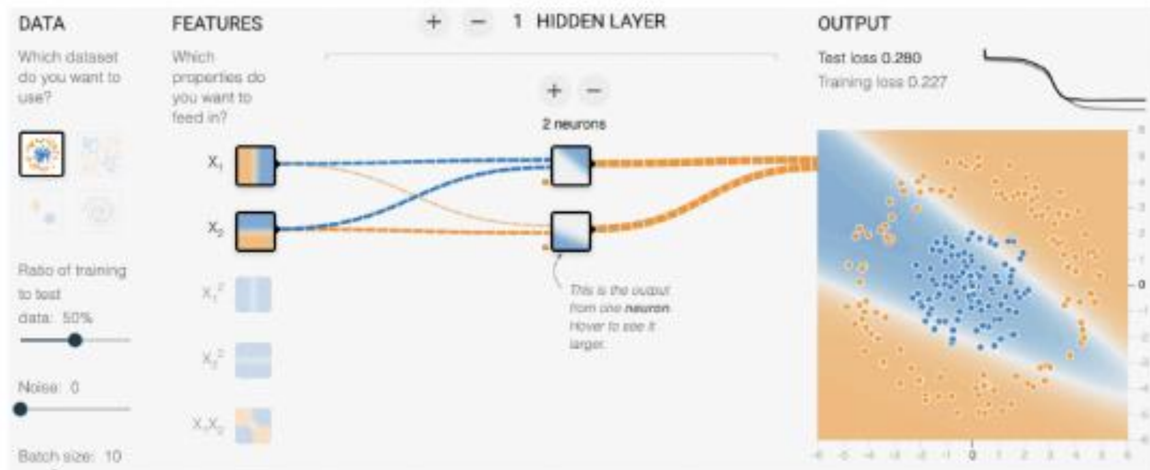
Pas de couche cachée



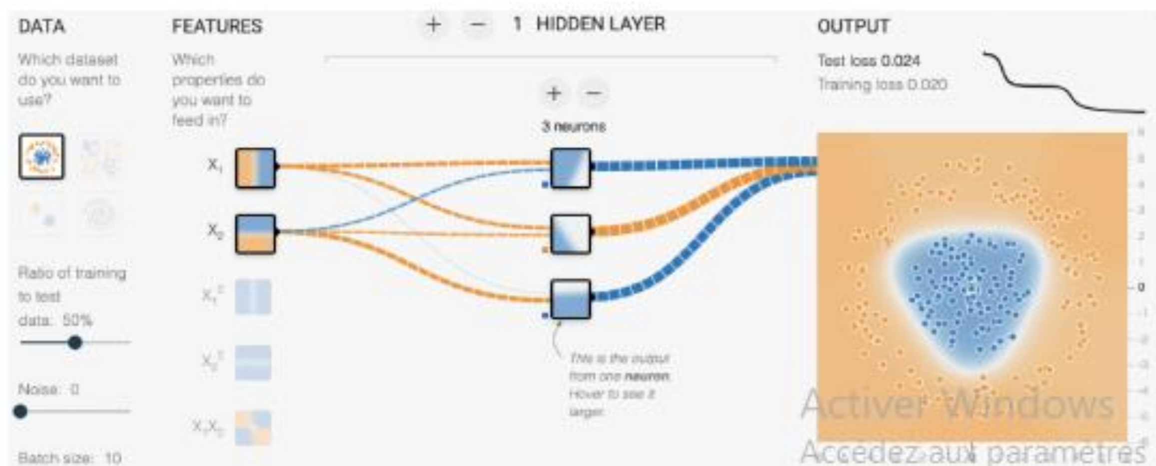
http://playground.tensorflow.org/

Exemple :Séparation 2-D (2/3)

One hidden layer
2 neurons

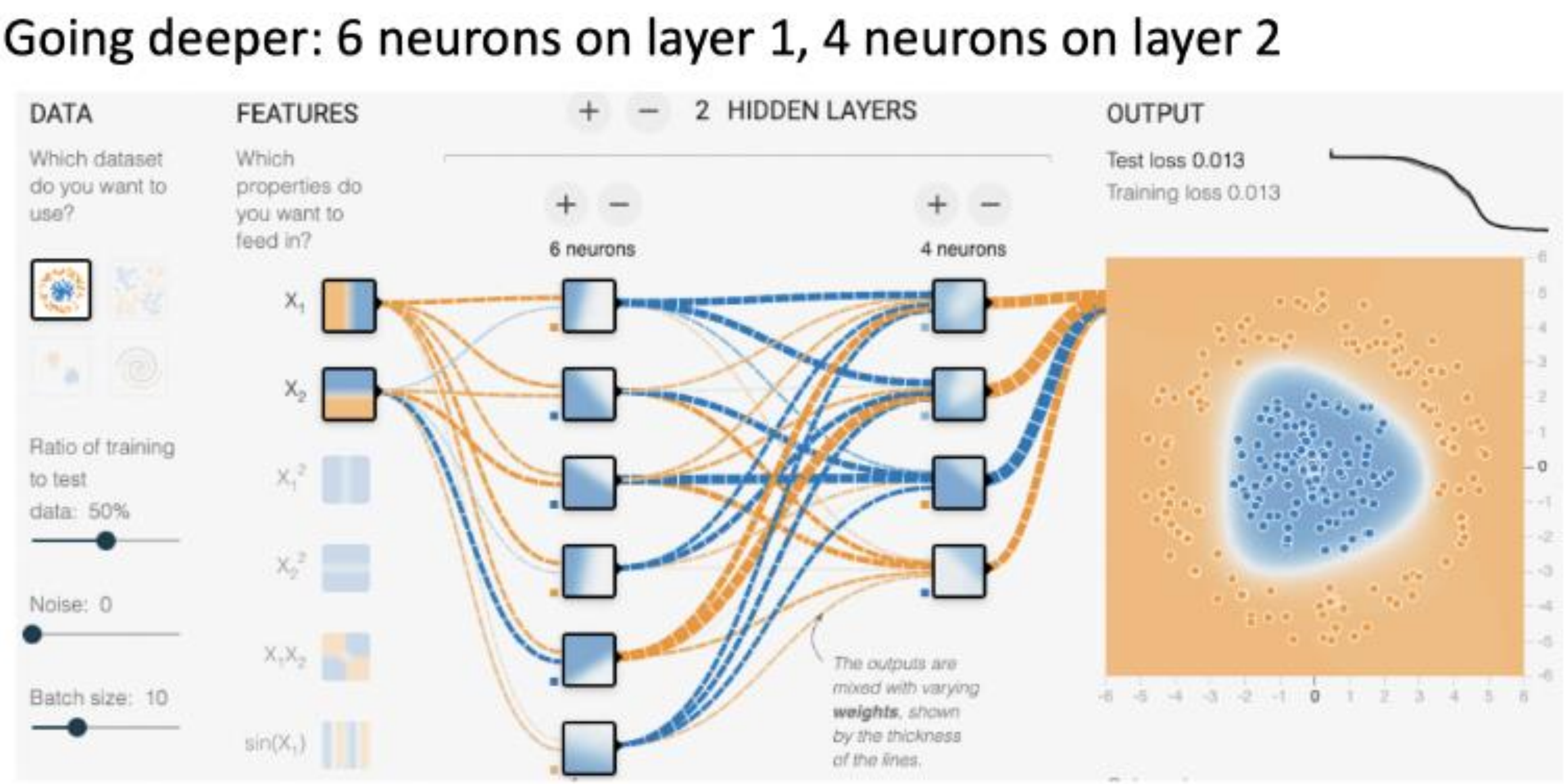


One hidden layer
3 neurons



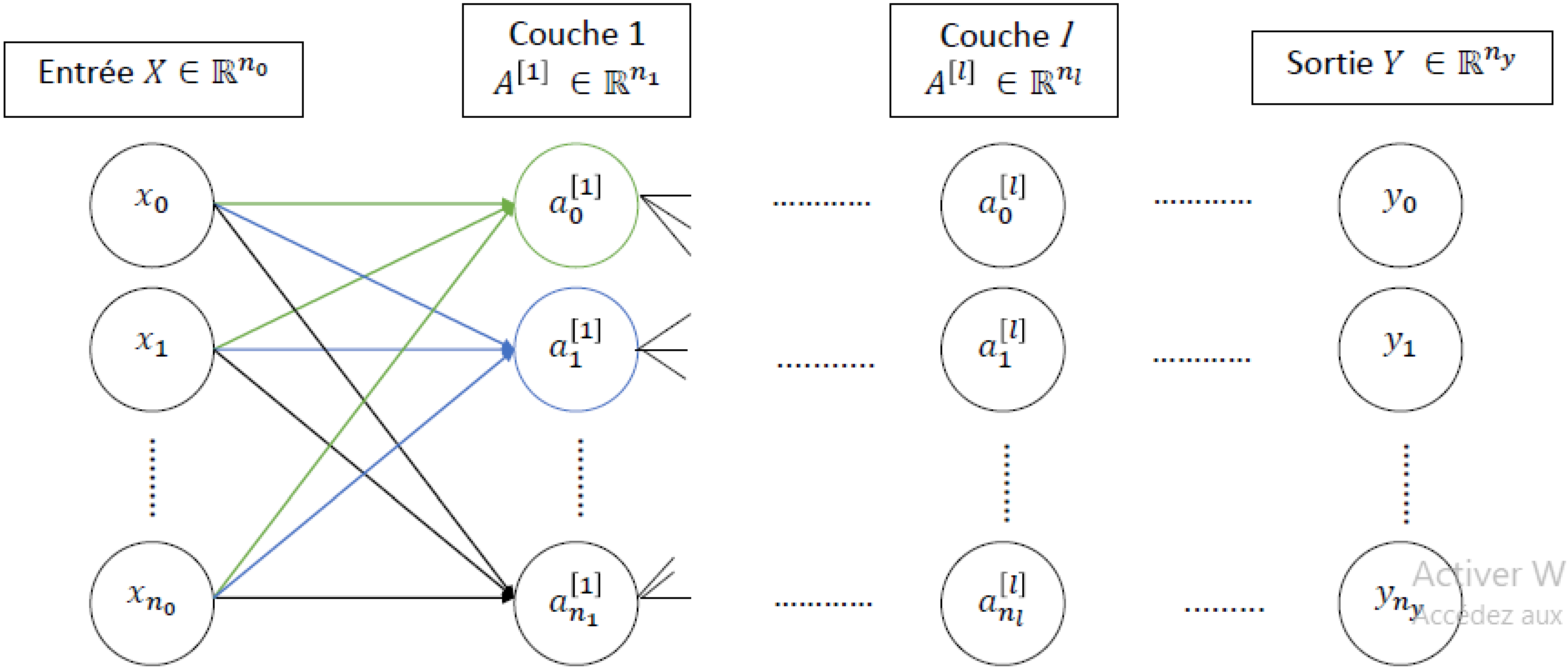
http://playground.tensorflow.org/

Exemple :Séparation 2-D (3/3)



http://playground.tensorflow.org/

Architecture et notations :



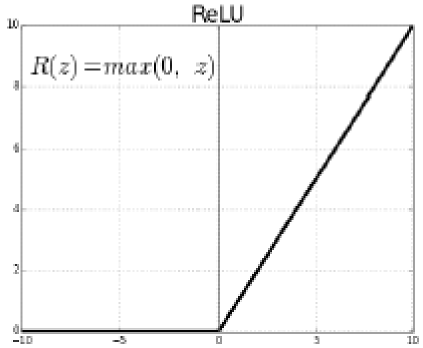
Neurone et fonction de transfert

Chaque neurone **calcule son état** :

$$y = f(W.X) = f(\sum_i w_i x_i)$$

Exemple de fonction de transfert : RELU

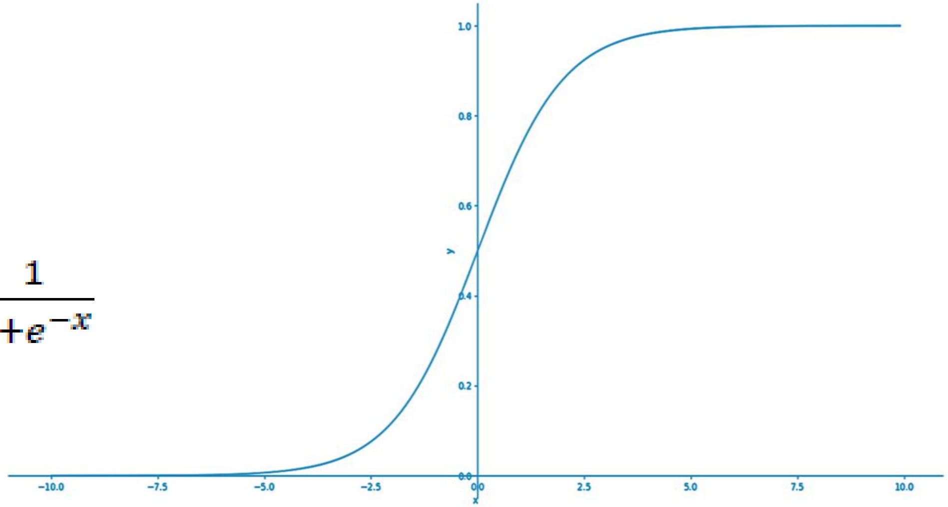
$$f(s) = \begin{cases} s & \text{si } s > 0 \\ 0 & \text{sinon} \end{cases}$$



- La plus utilisée
- Simplicité de calcul,
- gradient non évanescent

Autres fonctions de transfert :

fonction de transfert : Sigmoid



$$Sigmoïde(x) = \frac{1}{1+e^{-x}}$$

- Pour la classification entre 0 et 1 en output

fonction de transfert : Softmax

$$softmax(x) = \frac{e^{-z}}{\sum_{z_0 \in output} e^{-z_0}}$$

- Pour la classification « Multi-classes »

La fonction coût :

- **Évaluer** la qualité de la prédiction sur un jeu de données connues
- Notation : θ, paramètres du réseau (poids + biais) ; $\hat{Y}(\theta) = \left(\hat{y}_{\mathrm{ij}}(\theta)\right)_{\mathrm{ij}}$, output du réseau j pour l'exemple i ; $Y = \left(y_{ij}\right)_{ij}$, données réelles pour la valeur j du vecteur de sortie associé à l'exemple i
- Loss function :

$$L(\theta) = f\left(\hat{Y}(\theta), Y\right)$$

- Exemples de fonctions de coût :
  - Distance euclidienne (au carré) : $f\left(\hat{Y}(\theta), Y\right) = \|\hat{Y}(\theta) - Y\|_2^2 = \sum_{ij}\left(\hat{y}_{ij}(\theta) - y_{ij}(\theta)\right)^2$
  - Binary cross-entropy, pour la classification 0 ou 1 : $f\left(\hat{Y}(\theta), Y\right) = \sum_{ij}\left(y_{ij}\ln\left(\hat{y}_{ij}(\theta)\right) + \left(1 - y_{ij}\right)\ln\left(1 - \hat{y}_{ij}(\theta)\right)\right)$
  - Distance en norme 1 : $f\left(\hat{Y}(\theta), Y\right) = \|\hat{Y}(\theta) - Y\|_1 = \sum_{ij}\left|\hat{y}_{ij}(\theta) - y_{ij}(\theta)\right|$
  - Et d'autres…

Apprentissage et Optimisation:

- Fonction non convexe !!! Minima locaux possibles, mais :
  - Plusieurs minima locaux aussi « bons » vis-à-vis de la fonction de coût (Yann Le Cun)
  - On peut tomber dans un mauvais minimum, mais ceci est rare : différentes techniques permettent d'éviter cela (dropout, régularisation…)

- Supposons que nous avons un jeu de données d'entrées X_i et de sorties Y_i et un réseau de neurones avec les paramètres $\theta = (W, B)$ qui prédit les sorties $\hat{Y}_i$ à partir des données X_i

- Minimisation de la fonction de coût L par descente de gradient (itérations) :

$$\theta := \theta - \lambda \nabla \mathrm{L}(\theta)$$

λ est le taux d'apprentissage : valeur définie ou adaptée à chaque itération pour assurer la convergence

- Intérêt des réseaux de neurones : le gradient de la fonction de coût L se calcule « facilement », par succession de calculs élémentaires appelé backpropagation

Calcul du gradient : backpropagation (1/5)

- Pour chaque poids $w_k^{[l]}$ et biais $b_k^{[l]}$ du neurone k de la couche l, on veut calculer pour chque exemple i :

$$\frac{\partial L_i}{\partial w_k^{[l]}} ; \frac{\partial L_i}{\partial b_k^{[l]}}$$

- Pour la dernière couche n :

$$\frac{\partial L_i}{\partial w_k^{[n]}} = \frac{\partial L_i}{\partial \widehat{Y_{i_k}}} \frac{\partial \widehat{Y_{i_k}}}{\partial w_k^{[n]}} ; \frac{\partial L_i}{\partial b_k^{[n]}} = \frac{\partial L}{\partial \widehat{Y_{i_k}}} \frac{\partial \widehat{Y_{i_k}}}{\partial b_k^{[n]}}$$

$$\widehat{Y_{i_k}} = \sigma\left(z_k^{[n]} = w_k^{[n]^T} a^{[n-1]} + b_k^{[n]}\right)$$

$$\frac{\partial \widehat{Y_{i_k}}}{\partial w_k^{[n]}} = \frac{\partial \widehat{Y_{i_k}}}{\partial z_k^{[n]}} \frac{\partial z_k^{[n]}}{\partial w_k^{[n]}} = \sigma'\left(z_k^{[n]}\right) a^{[n-1]} \in \mathbb{R}^{[m_{n-1}]} ; \frac{\partial \widehat{Y_{i_k}}}{\partial b_k^{[n]}} = \frac{\partial \widehat{Y_{i_k}}}{\partial z_k^{[n]}} \frac{\partial z_k^{[n]}}{\partial b_k^{[n]}} = \sigma'\left(z_k^{[n]}\right) \in \mathbb{R}$$

$$\frac{\partial L_i}{\partial w_k^{[n]}} = \frac{\partial L_i}{\partial \widehat{Y_{i_k}}} \sigma'\left(z_k^{[n]}\right) a^{[n-1]} ; \frac{\partial L_i}{\partial b_k^{[n]}} = \frac{\partial L_i}{\partial \widehat{Y_{i_k}}} \sigma'\left(z_k^{[n]}\right)$$

Calcul du gradient : backpropagation (2/4)

- Exemple avec :

$$L_i = \left(\hat{Y}_i(\theta) - Y_i\right)^2$$
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

- On obtient :

$$\sigma'\left(z_k^{[n]}\right) = \sigma\left(z_k^{[n]}\right)\left(1 - \sigma\left(z_k^{[n]}\right)\right) = \widehat{Y_{i_k}}\left(1 - \widehat{Y_{i_k}}\right)$$
$$\frac{\partial L}{\partial \widehat{Y_{i_k}}} = 2\left(\widehat{Y_{i_k}} - Y_{i_k}\right)$$

Calcul du gradient : backpropagation (3/4)

- Pour les autres couches $l \neq n$: de manière récursive

$$\frac{\partial L_i}{\partial w_k^{[l]}} = \frac{\partial L_i}{\partial a_k^{[l]}} \frac{\partial a_k^{[l]}}{\partial w_k^{[l]}} = \frac{\partial L_i}{\partial a_k^{[l]}} \sigma' \left(z_k^{[l]} \right) a^{[l-1]}$$

En rouge : par forward pass

En bleu : par récursivité

$$\frac{\partial L_i}{\partial b_k^{[l]}} = \frac{\partial L_i}{\partial a_k^{[l]}} \frac{\partial a_k^{[l]}}{\partial b_k^{[l]}} = \frac{\partial L_i}{\partial a_k^{[l]}} \sigma' \left(z_k^{[l]} \right) ; \ (\text{pour } l = 1, a^{[0]} = X)$$

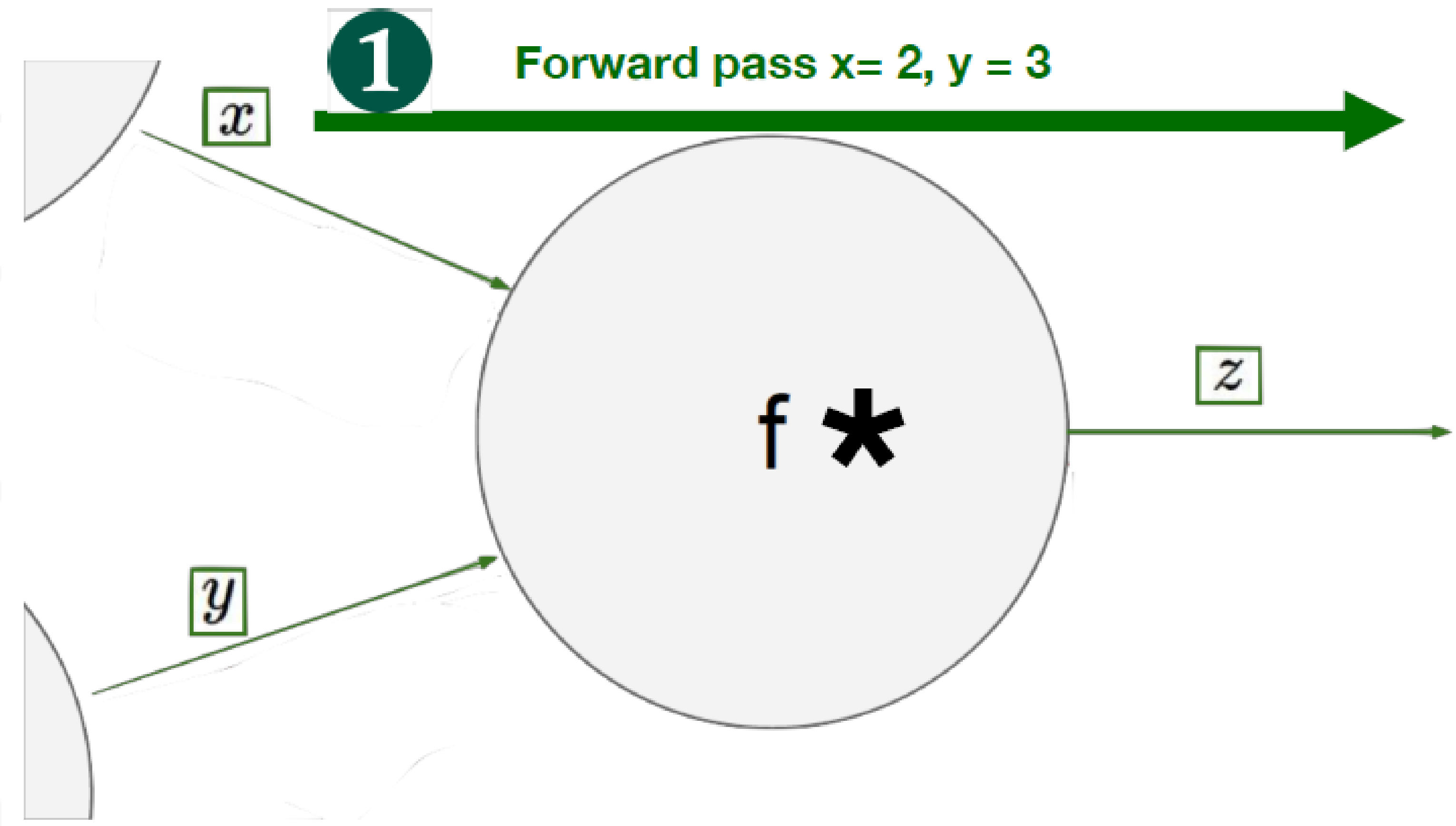
$$\frac{\partial L_i}{\partial a_k^{[l]}} = \sum_j \frac{\partial L_i}{\partial a_j^{[l+1]}} \frac{\partial a_j^{[l+1]}}{\partial a_k^{[l]}}$$

$$a_j^{[l+1]} = \sigma \left(z_j^{[l+1]} = \sum_{k'} \left(w_j^{[l+1]} \right)_{k'} a_{k'}^{[l]} + b_j^{[l+1]} \right)$$

$$\Rightarrow \frac{\partial a_j^{[l+1]}}{\partial a_k^{[l]}} = \left(w_j^{[l+1]} \right)_k \sigma' \left(z_j^{[l+1]} \right)$$

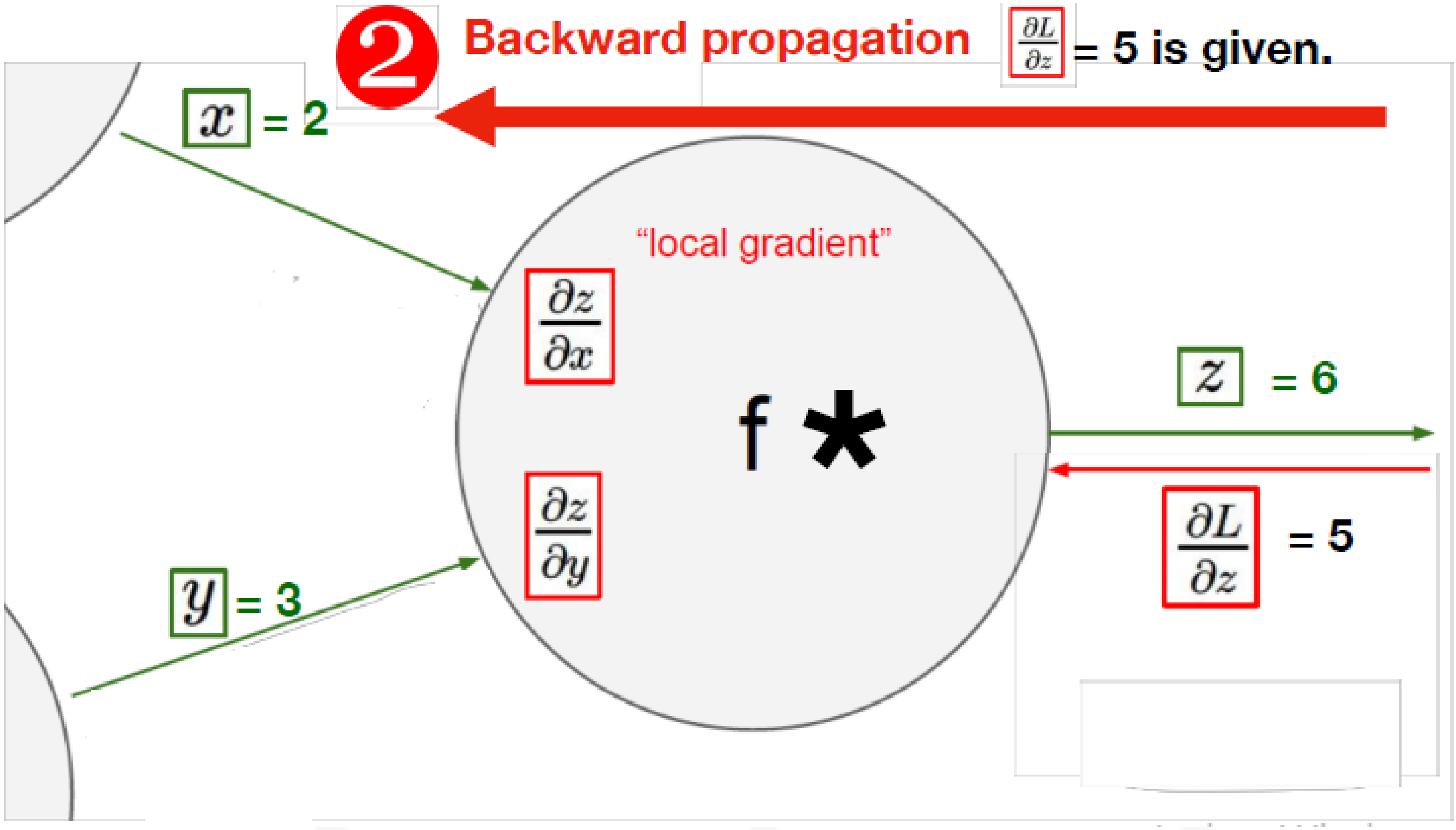
Calcul du gradient : backpropagation

Illustration :



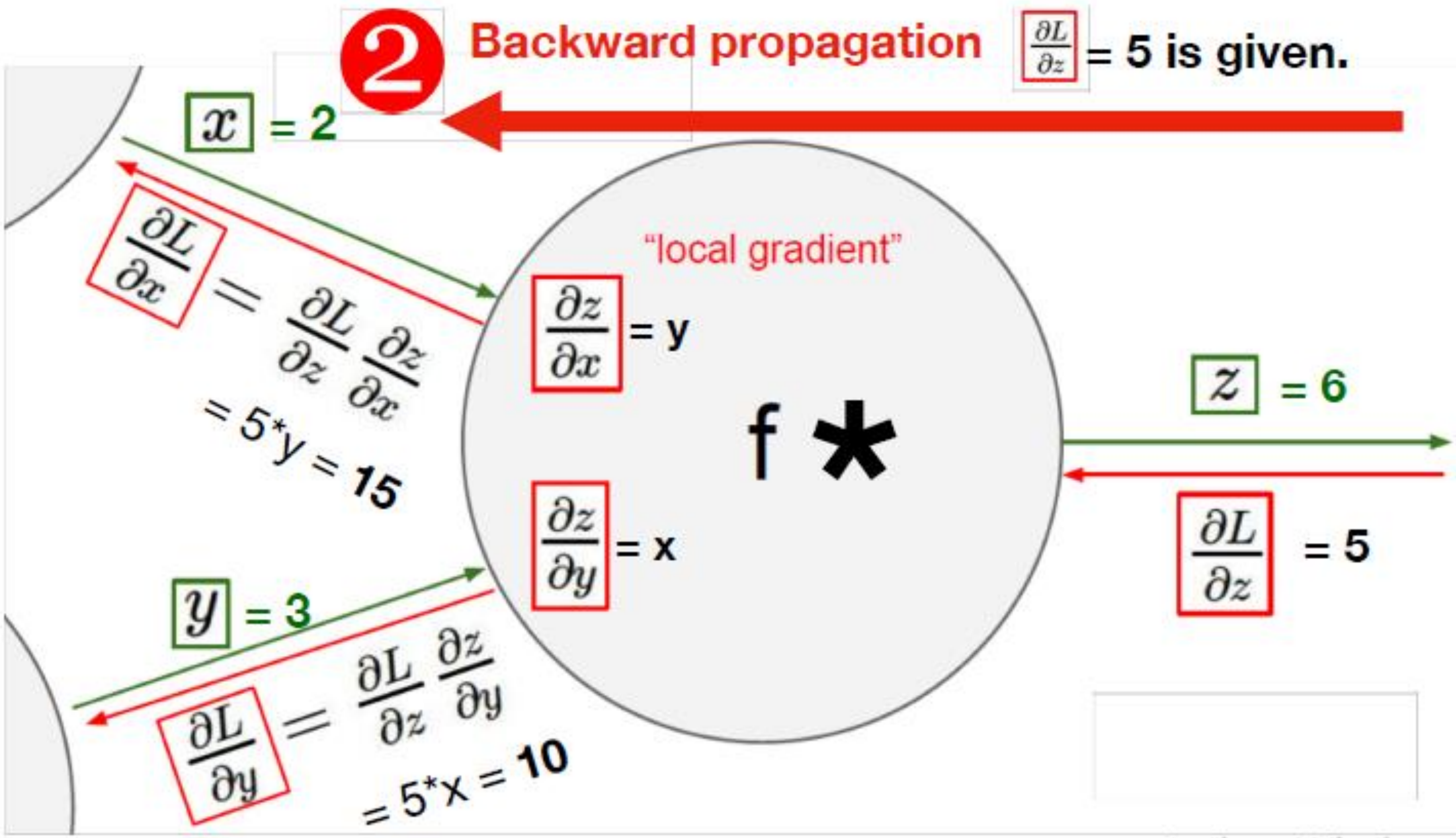
Calcul du gradient : backpropagation

Illustration :



Calcul du gradient : backpropagation (1/5)

Illustration :



Calcul du gradient : backpropagation (4/4)

- On connaît $\frac{\partial L_i}{\partial w_k^{[l]}}$ et $\frac{\partial L_i}{\partial b_k^{[l]}}$ pour chaque exemple i

- Finalement :

$$w_k^{[l]} := w_k^{[l]} - \lambda \frac{1}{N_{exemples}} \sum_i \frac{\partial L_i}{\partial w_k^{[l]}}$$
$$b_k^{[l]} := b_k^{[l]} - \lambda \frac{1}{N_{exemples}} \sum_i \frac{\partial L_i}{\partial b_k^{[l]}}$$

- On peut ne travailler simultanément que sur des sous-ensembles de la base de données (mini-batch), cela peut accélérer les calculs

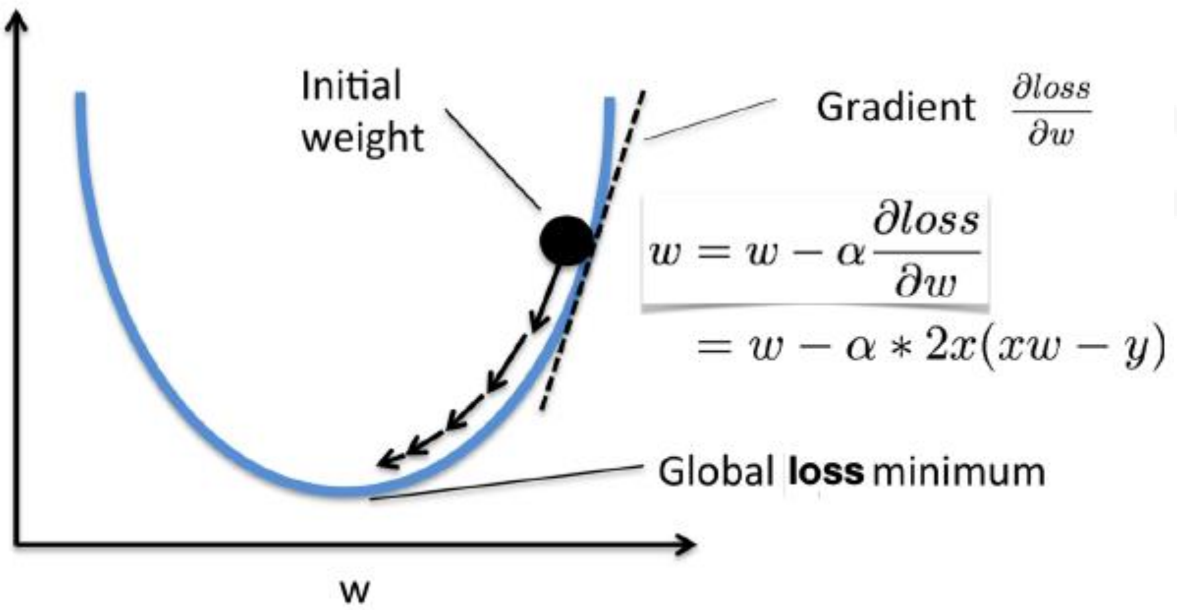
Exemple : optimisation MSE modèle linéaire 1 neuron

$$\hat{y} = w.x$$

$$\text{coût} = (\hat{y} - y)^2$$

Quadratique (MSE) :

$$L = \text{MSE} = \frac{1}{m}\sum_i (\hat{y}_i - y_i)^2$$



Exemple : optimisation MSE modèle linéaire Code python : (1 neuron)

```
# Données: entrées x (valeurs scalaires), sorties désirées y
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]
```

```
w = 1.0  # poids (paramètre) initial, au hasard

# modèle: calcul de la sortie ("forward pass")
def forward(x):
    return x * w


# Fonction de coût
def cout(x, y):
    y_pred = forward(x)
    return (y_pred - y) * (y_pred - y)


# calcul du gradient du cout par rapport au poids
def gradient(x, y):  # d_loss/d_w
    return 2 * x * (x * w - y)
```

```
# Before training
print("prévision avant apprentissage:",  4, forward(4))

# Boucle d'apprentissage
for epoch in range(100):
    for x_val, y_val in zip(x_data, y_data):
        grad = gradient(x_val, y_val)
        w = w - 0.01 * grad
        print("\tgrad: ", x_val, y_val, grad)
        l = cout(x_val, y_val)

    print("progress:", epoch, "w=", w, "loss=", l)

# After training
print("predict (after training)",  "4 hours", forward(4))
```