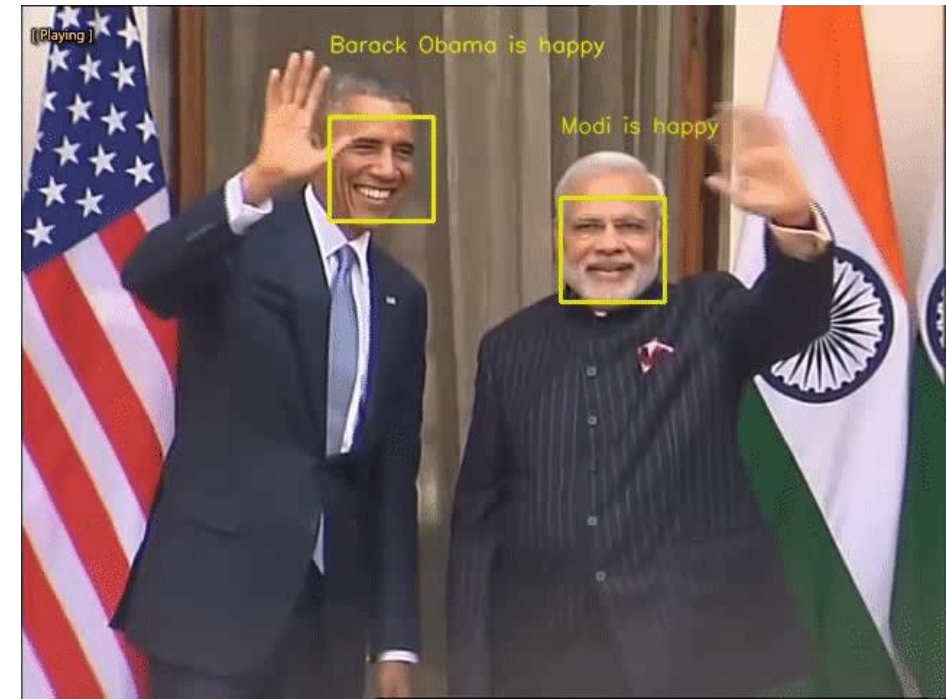


Les Réseaux de Neurones



Utilisations



RAPPEL

Régression Linéaire:

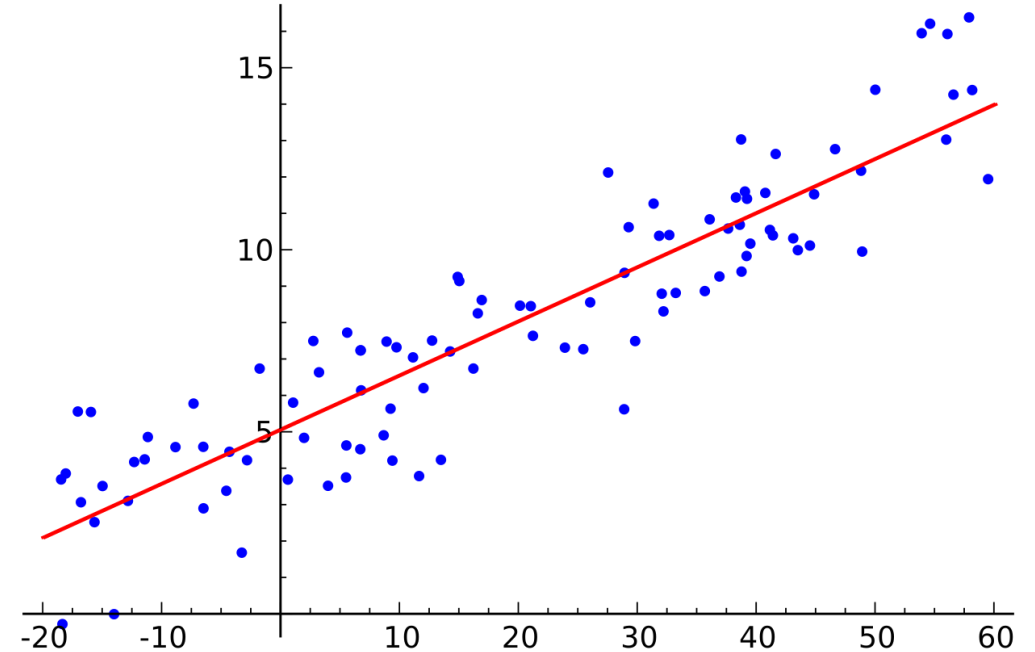
$$\hat{y} = x_1 w_1 + \dots + x_n w_n + b$$

\hat{y} : prédiction de y

x_i : valeur de x associé à i

w_i : poids associé à i

b : biais (ordonnée à l'origine)



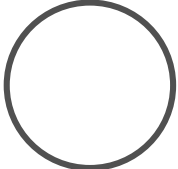
→ On cherche à avoir \hat{y} le plus proche possible de y en cherchant les valeurs optimales de w_i et b .

PERCEPTRON

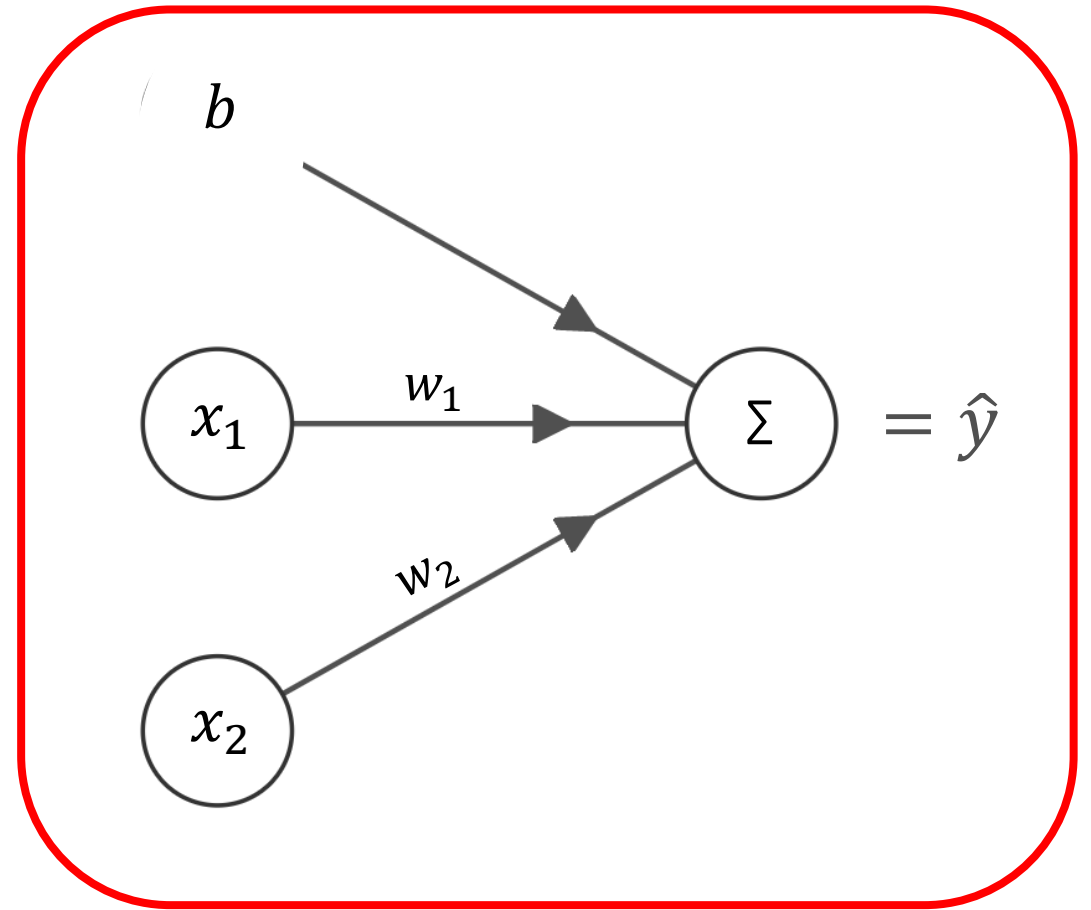
Régression Linéaire, *représentation graphique*:

$$\hat{y} = x_1 w_1 + x_2 w_2 + b$$

Σ : représente la fonction d'activation,
dans ce cas une régression linéaire simple

 représente les nœuds du réseau

Les poids sont initialisés aléatoirement



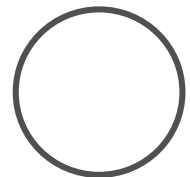
Perceptron

PERCEPTRON

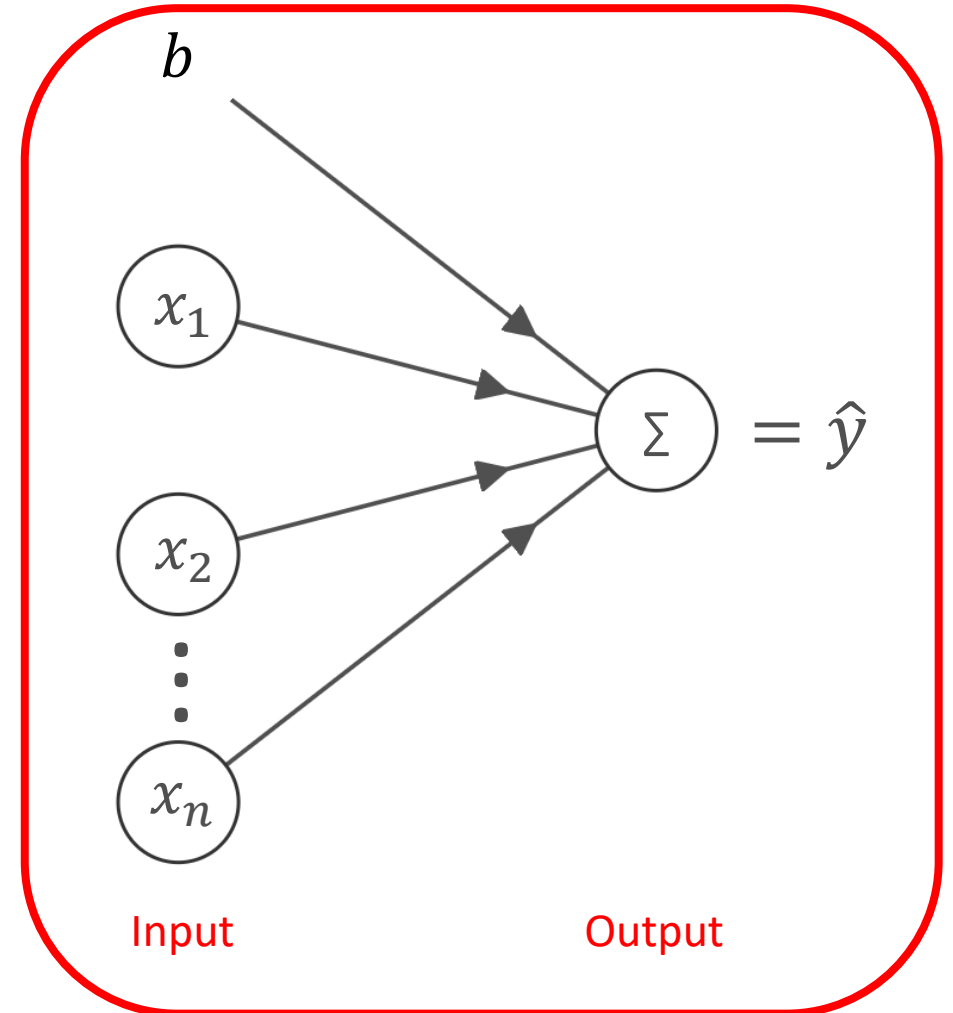
Réseau de neurone simple

$$\hat{y} = x_1 w_1 + \dots + x_n w_n + b$$

Σ : représente la fonction d'activation,
dans ce cas une régression linéaire simple

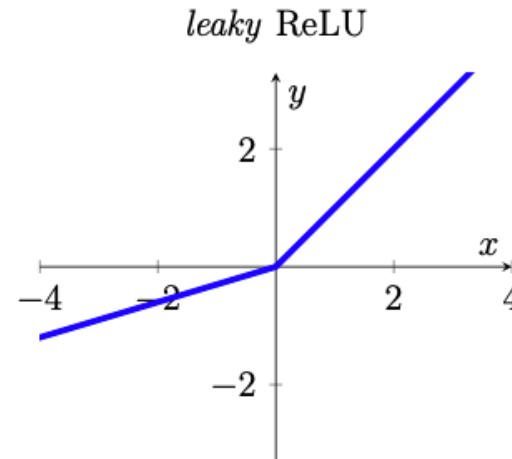
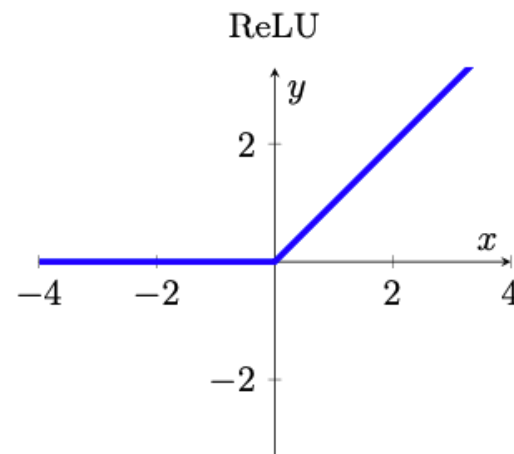
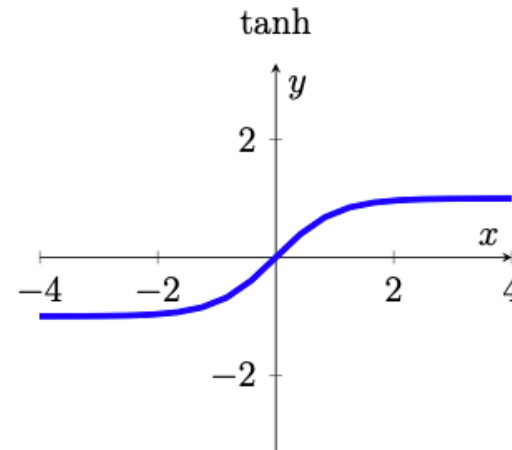
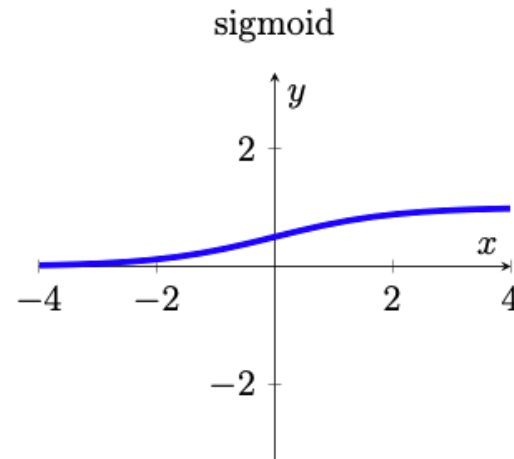
 représente les nœuds du réseau

Les poids sont initialisés aléatoirement



FONCTION D'ACTIVATION

Il existe plusieurs **fonctions d'activations** :



SIGMOÏDE

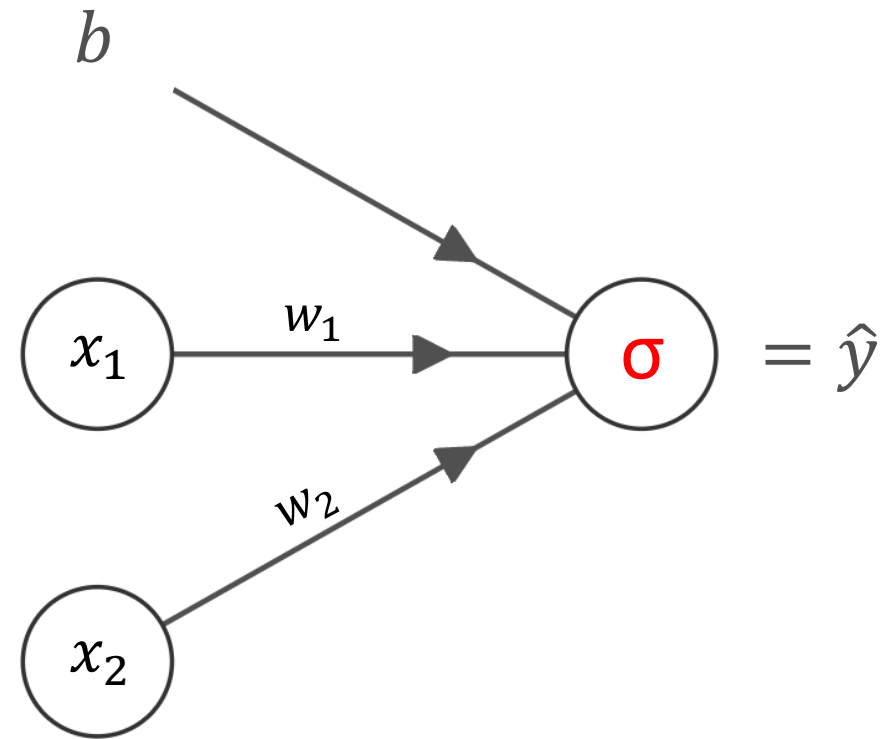
La fonction sigmoïdes peut être utiliser dans des cas de **classifications binaires** :

$$\hat{y} = \sigma(x_1 w_1 + x_2 w_2 + b)$$

$$\text{avec } \sigma(z) = \frac{1}{1 + e^{-z}}$$

On retrouve ici la régression logistique. On obtient alors une valeurs entre 0 et 1 qui permet de prédire à quel classe \hat{y} un individu x appartient.

 **Classification binaire**



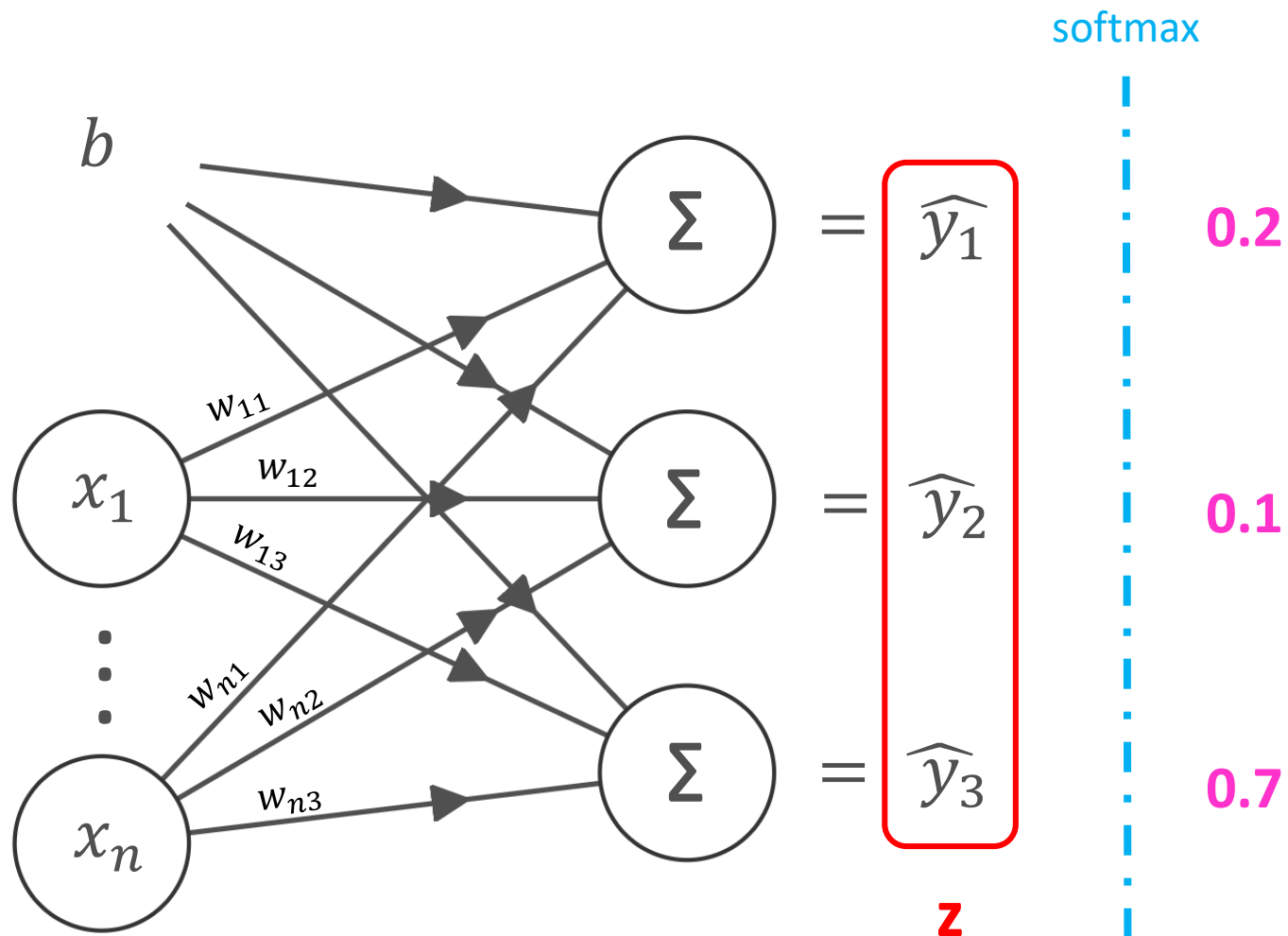
SOFTMAX

La fonction softmax peut être utilisée en fin de réseau pour la **classification multiple**:

Softmax:

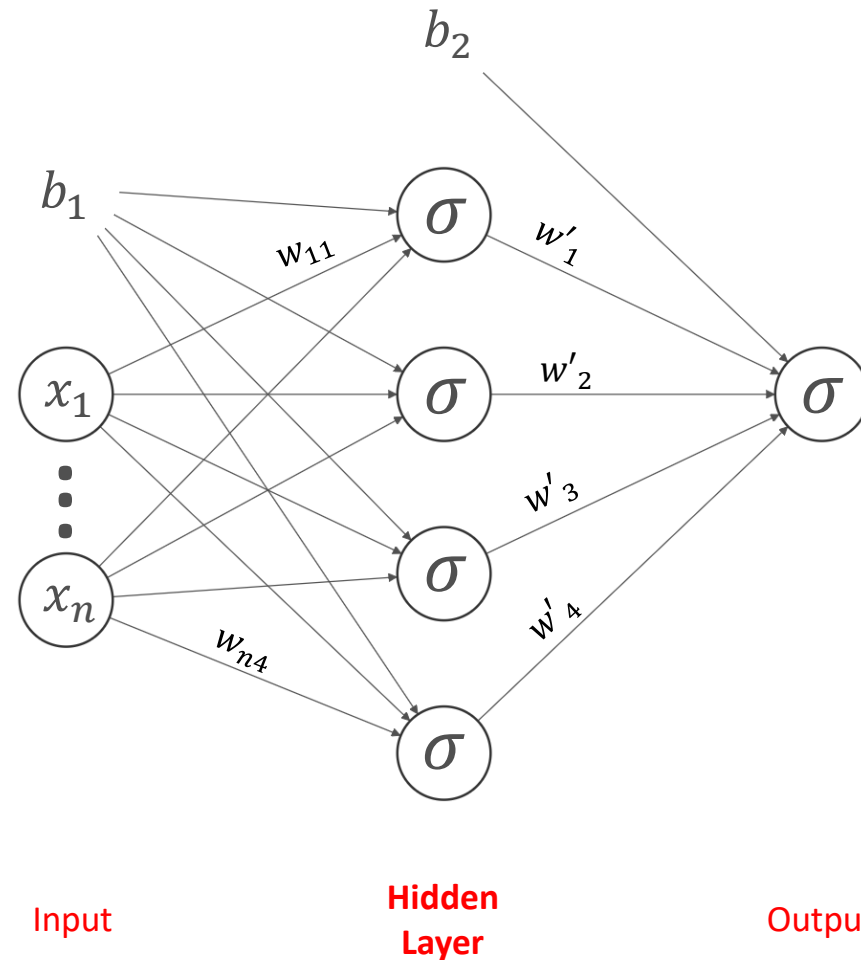
$$\sigma(z) = \frac{e^{z_i}}{\sum_{i=1}^n e^{z_i}}$$

Pour chaque nœud, on obtient la probabilité d'appartenance à la classe y_i . La somme des probabilités est égale à 1.



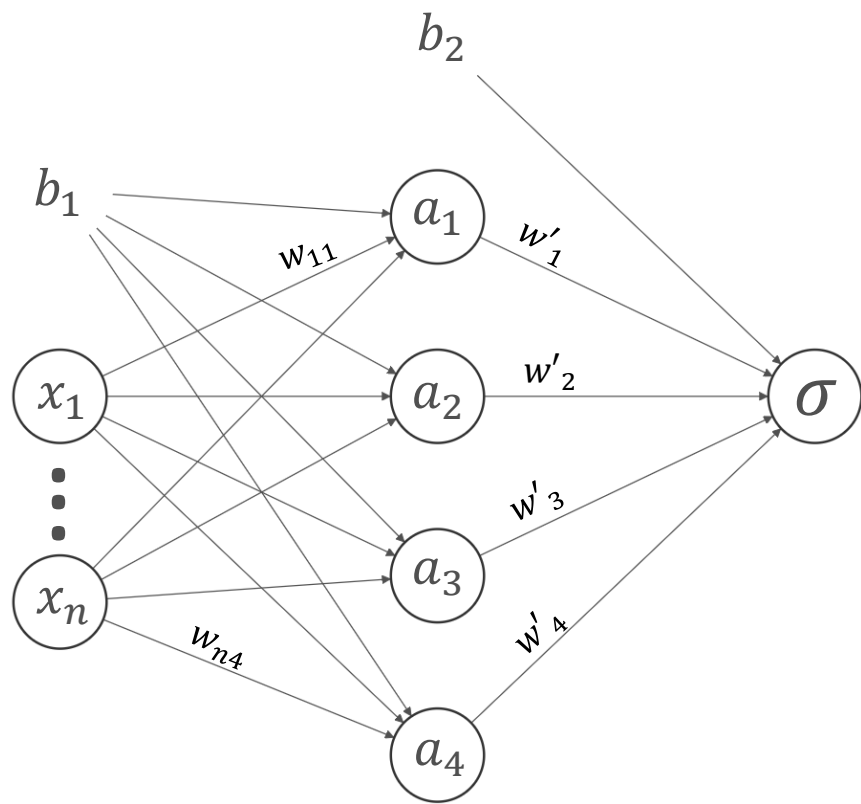
HIDDEN LAYERS

On peut complexifier le réseau en y ajoutant 1 couche cachée :



HIDDEN LAYERS

On peut complexifier le réseau en y ajoutant 1 couche cachée :



Input

Hidden
Layer

Output

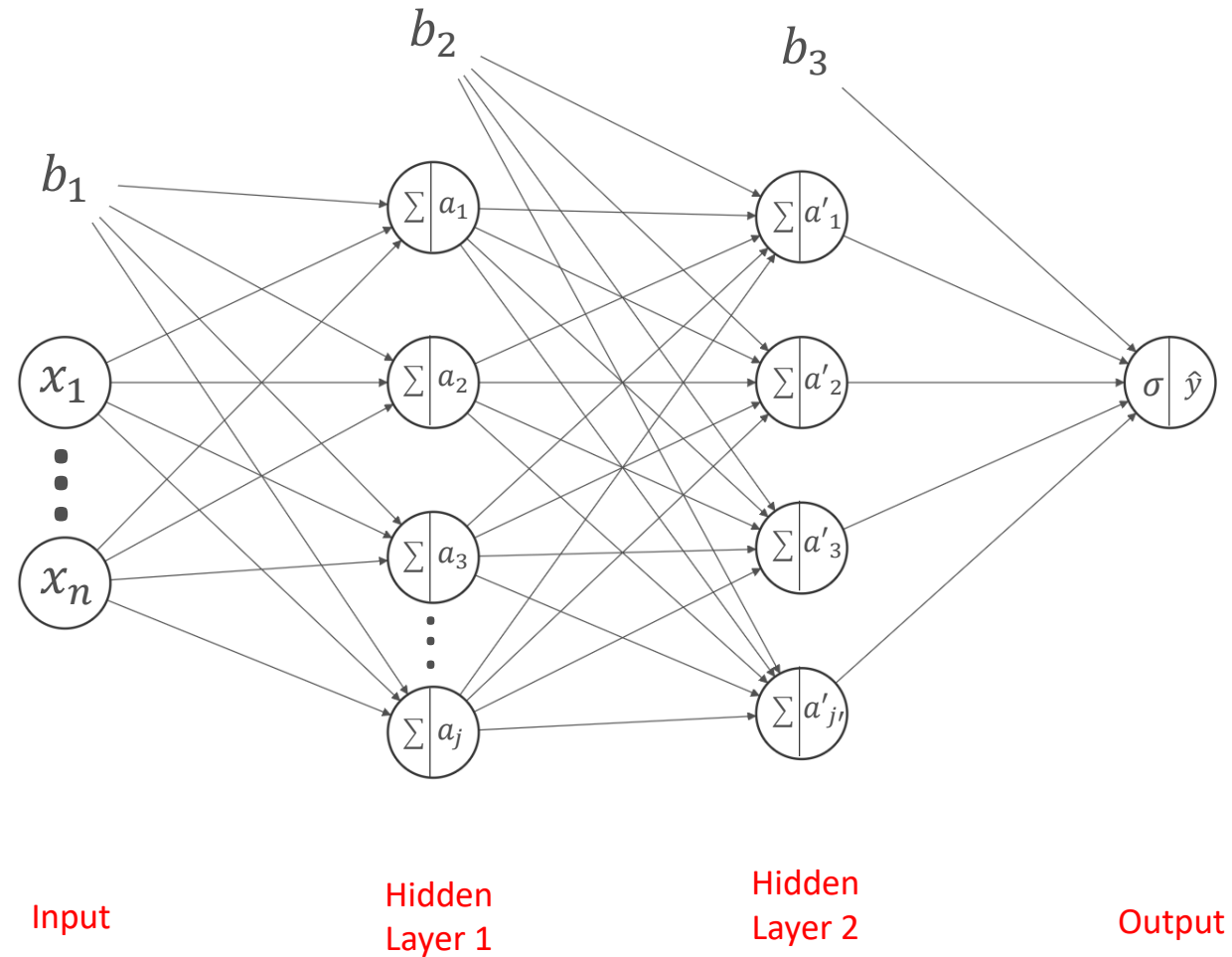
$$a_j = \sigma(x_1 w_{1j} + \dots + x_n w_{nj} + b_1) \quad \text{avec } j = 1, \dots, 4$$

$$\hat{y} = \sigma(a_1 w'_1 + \dots + a_4 w'_4 + b_2)$$

$$\text{avec } \sigma(z) = \frac{1}{1 + e^{-z}}$$

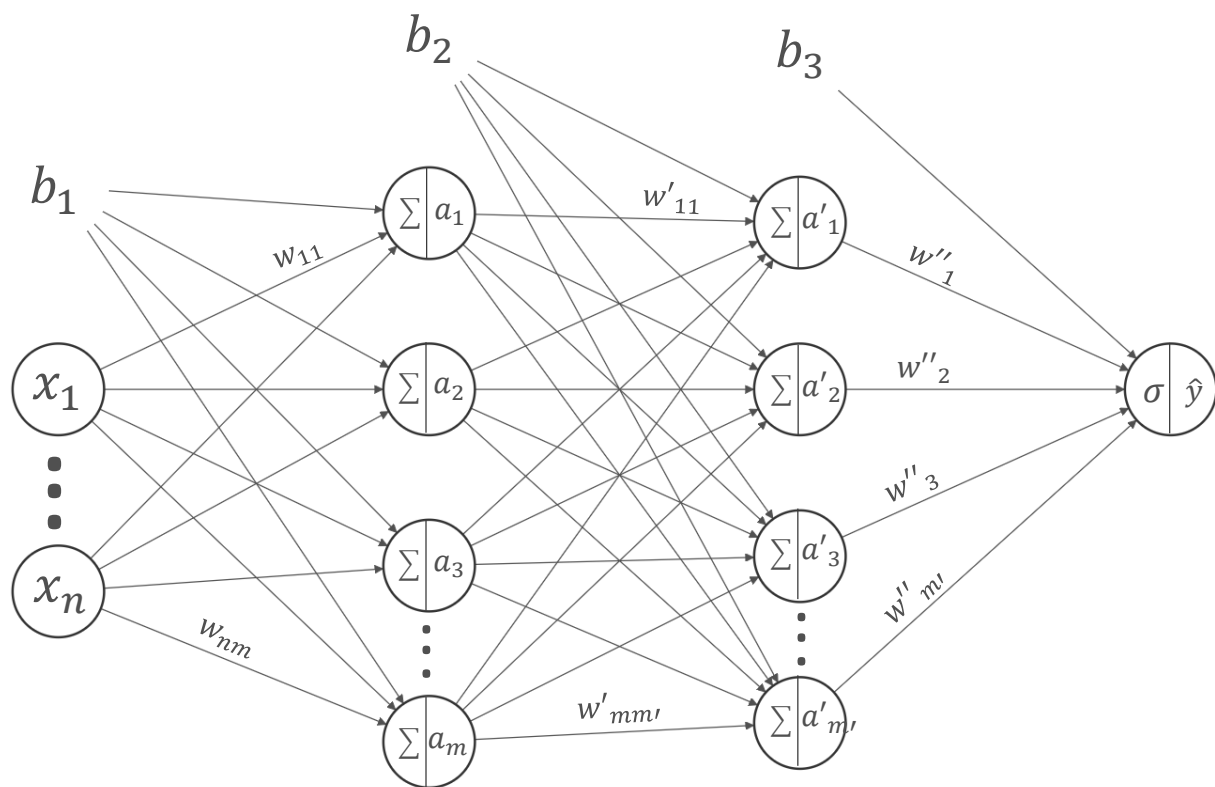
DEEP LEARNING

Ou en y ajoutant plusieurs couches cachées :



DEEP LEARNING

Ou en y ajoutant plusieurs couches cachées :



Input

Hidden
Layer 1

Hidden
Layer 2

Output

$$a_j = \sum (x_1 w_{1j} + \dots + x_n w_{nj} + b_1)$$

avec $j = 1, \dots, m$

$$a'_{j'} = \sum (a_1 w'_{1j'} + \dots + a_m w'_{mj'} + b_2)$$

avec $j' = 1, \dots, m'$

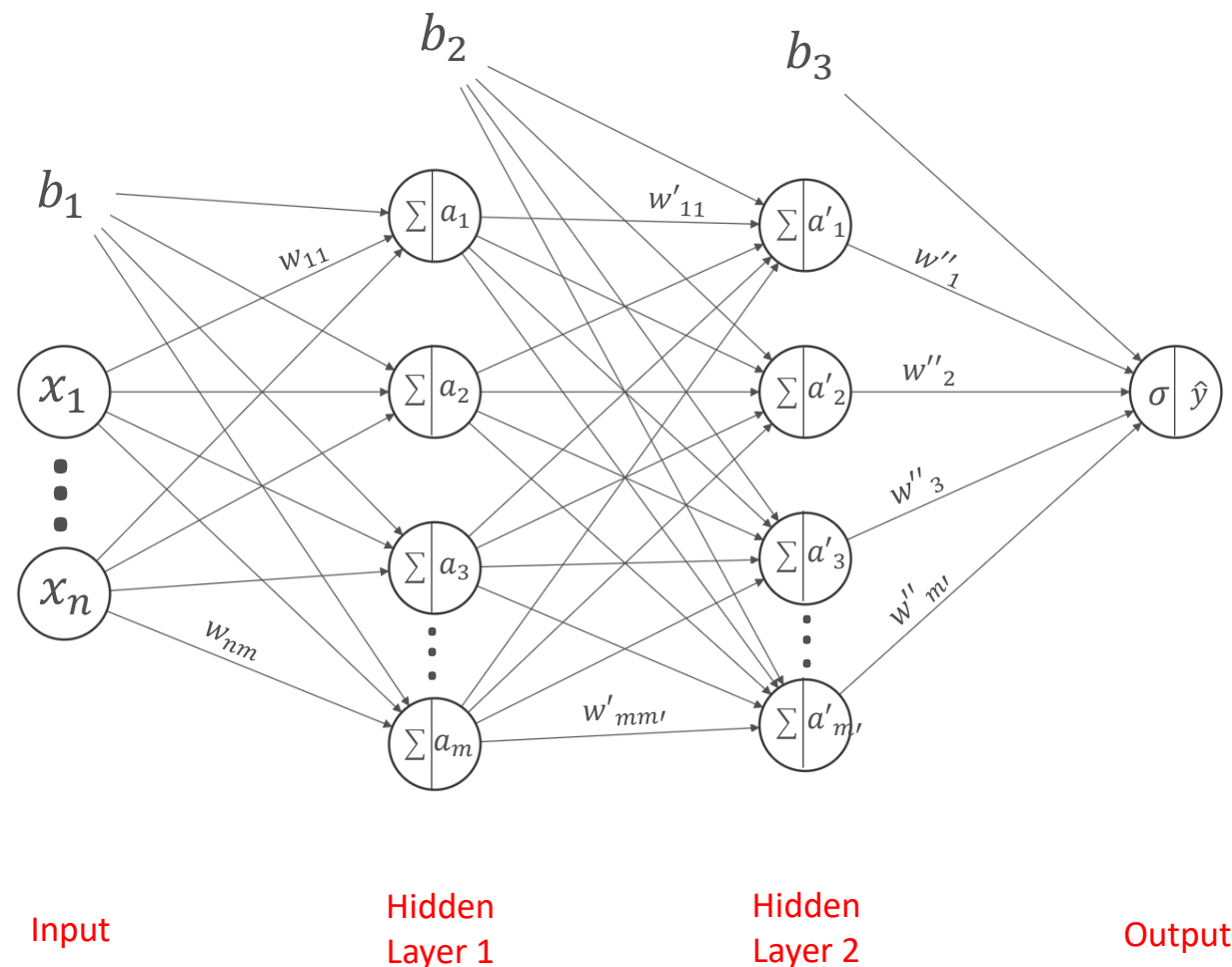
$$\hat{y} = \sigma(a'_1 w''_1 + \dots + a'_{m'} w''_{m'} + b_3)$$

$$\text{avec } \sigma(z) = \frac{1}{1 + e^{-z}}$$

VOCABULAIRE

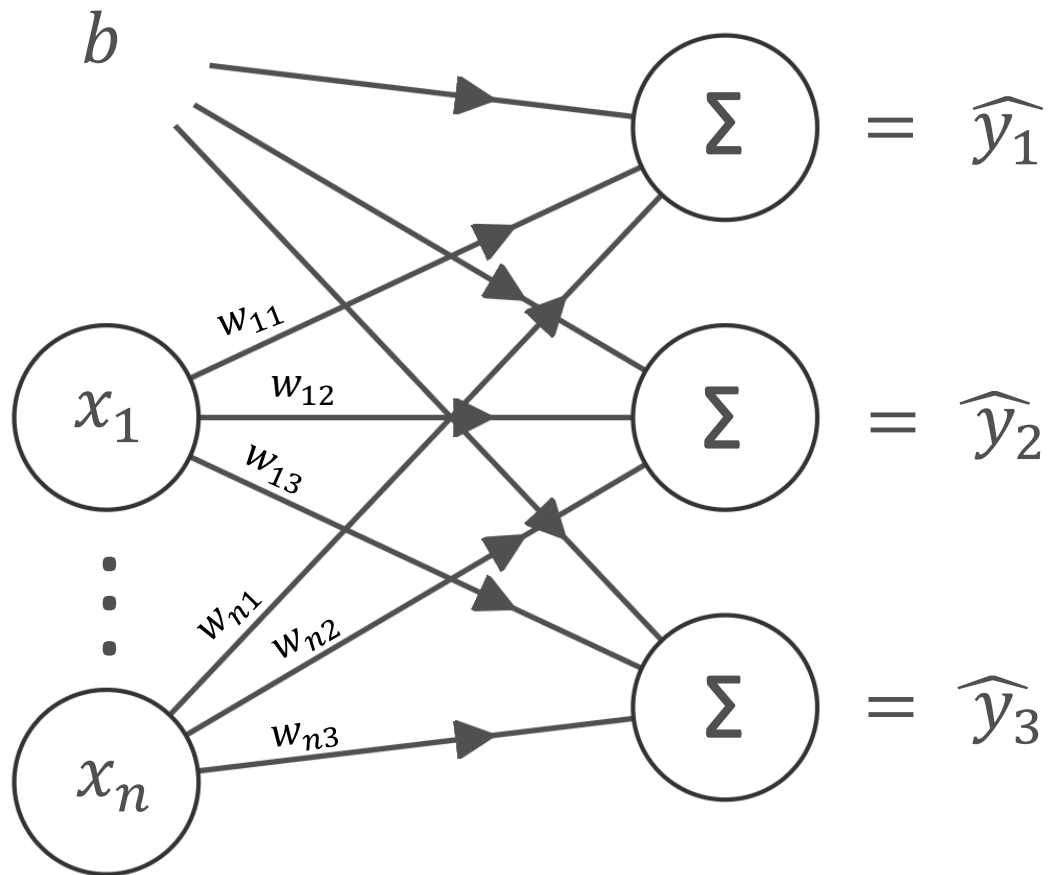
Pour ce type de réseau on peut parler de :

- **Deep Neural Network (DNN):** car il possède plusieurs couches cachées
- **Fully Connected NN:** car tous les nœuds d'une couche sont connectés avec ceux de la précédente
- **Feedforward NN (FNN):** car toutes les flèches vont dans le même sens (de la gauche vers la droite)



ERREUR

Comment fait le réseau pour **déduire la bonne valeur** ?



Grâce à un dataset d'entraînement, on **compare** la **valeur prédite \hat{y}** à la **vraie valeur y** que l'on doit trouver:

$$\text{error} = y_i - \hat{y}_i$$

$$\text{squared error} = (y_i - \hat{y}_i)^2$$

$$\text{mean squared error} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

APPRENTISSAGE

Nous savons donc faire une prédiction et calculer l'erreur associée.

Afin d'obtenir le modèle qui va donner **les meilleures prédictions**, nous devons avoir **l'erreur la plus faible** possible autrement que l'on obtienne une prédiction correcte: $\widehat{y}_i = y_i$

Dans ce cas là nous cherchons à avoir : $error = y_i - \widehat{y}_i = 0$

Comment fait on pour trouver le minimum d'une fonction ?



On cherche la valeur de sa dérivé égal à 0

Etant donné que nous pouvons jouer sur la valeur des poids, nous allons essayé de déterminer pour quelles valeurs de w l'erreur est la plus proche de 0.

FONCTION DE PERTE

On appelle fonction de perte (ou fonction de coût ou fonction d'erreur) la fonction E qui permet de déterminer la valeur de notre erreur en fonction de la valeur des poids w_i du réseau.

$$E(w_i) = y_i - \widehat{y}_i(w_i)$$

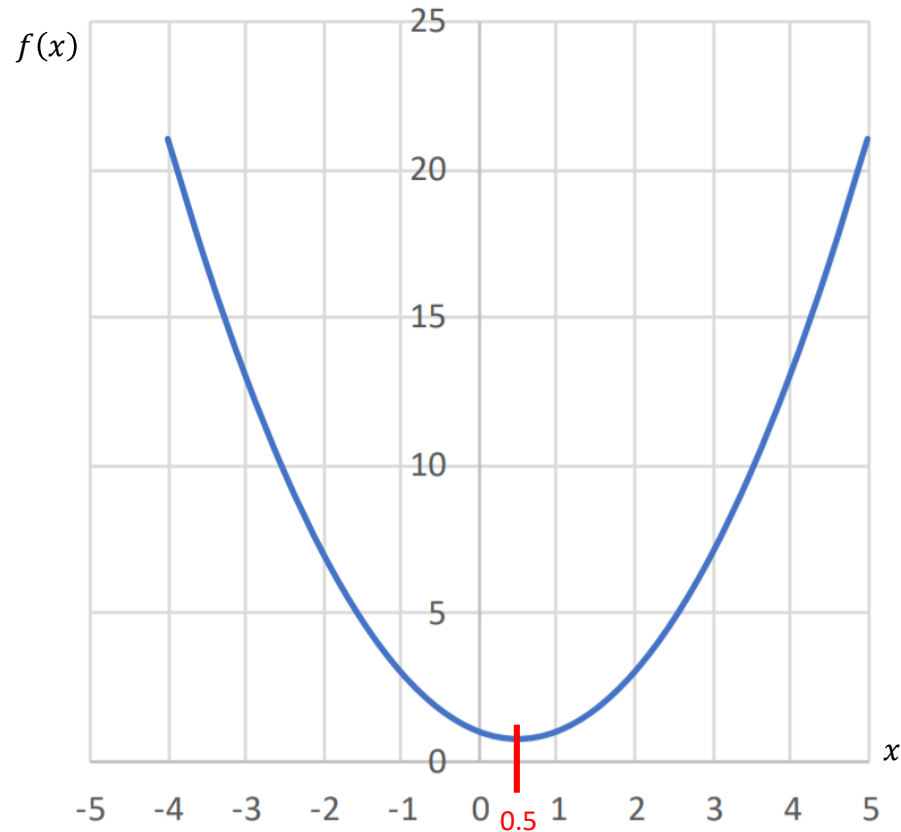
On cherche alors à minimiser cette fonction (quelle soit le plus proche de 0) afin de déterminer les valeurs optimales de tous les poids:

$$\frac{\partial E}{\partial w_i} = 0 \quad \Longleftrightarrow \quad \frac{\partial (y_i - \widehat{y}_i(w_i))}{\partial w_i} = 0$$

On peut tout à fait choisir de travailler avec une erreur égale à *mean squared error*, ou à d'autres valeurs plus exotiques.

GRADIENT DESCENDANT

Le gradient descendant est une technique **d'optimisation**. Elle permet de **minimiser** efficacement une fonction.



$$f(x) = x^2 - x + 1$$

$$\min f(x) = x^2 - x + 1 \implies f'(x) = 0$$

$$f'(x) = 2x - 1 = 0 \implies x = 1/2$$

GRADIENT DESCENDANT

Le **gradient descendant** va calculer les dérivées partielles de notre fonction de perte en par rapport aux valeurs de x :

eta 0.3

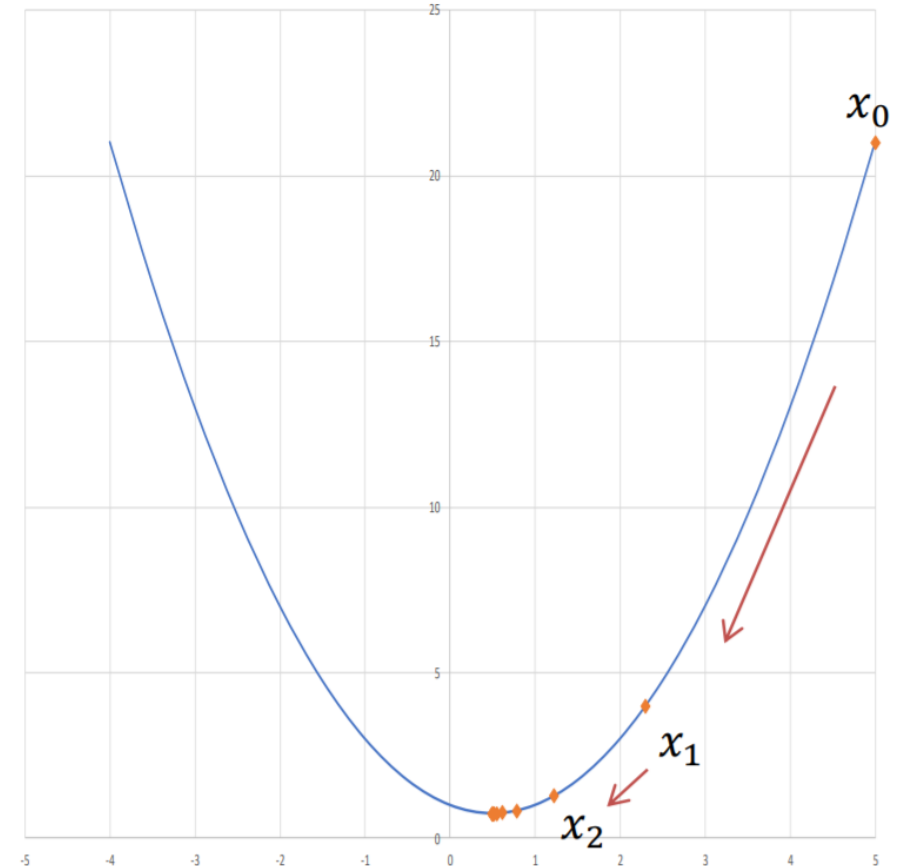
$x_0 = 5$	x	$f'(x_t)$	$f(x)$
	5.0000		21.0000
	2.3000	9.0000	3.9900
	1.2200	3.6000	1.2684
	0.7880	1.4400	0.8329
	0.6152	0.5760	0.7633
	0.5461	0.2304	0.7521
	0.5184	0.0922	0.7503
	0.5074	0.0369	0.7501
	0.5029	0.0147	0.7500
	0.5012	0.0059	0.7500
	0.5005	0.0024	0.7500
	0.5002	0.0009	0.7500
	0.5001	0.0004	0.7500
	0.5000	0.0002	0.7500

$$f(x) = x^2 - x + 1$$

$$\nabla f(x) = \frac{\partial f(x)}{\partial x} = f'(x) = 2x - 1$$

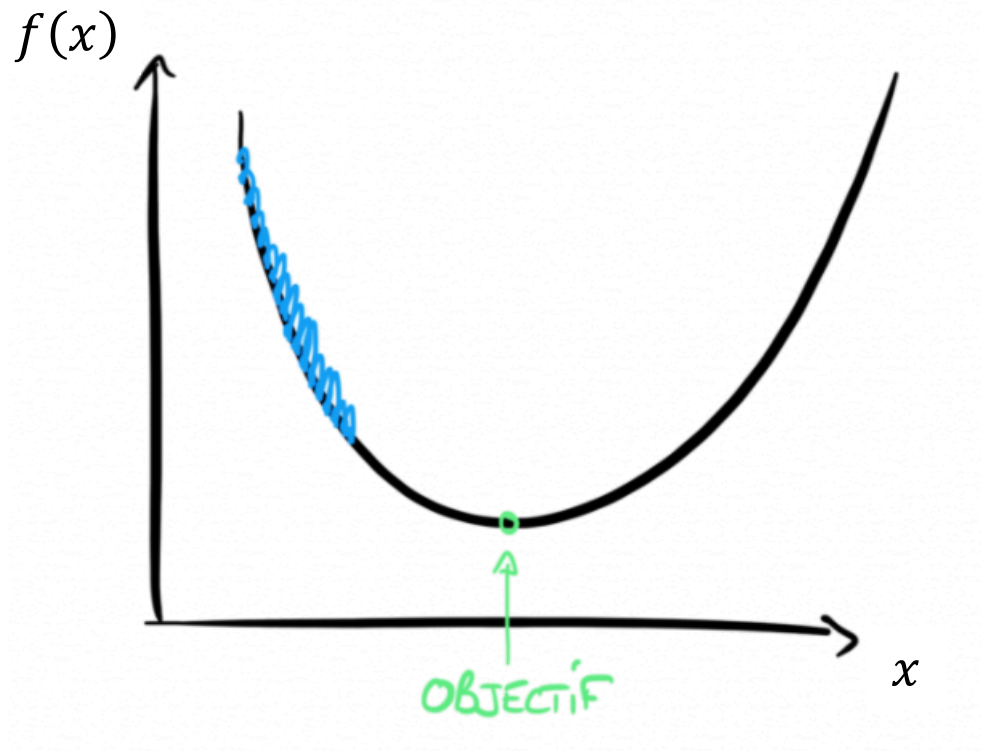
Gradient de f

$$x_{t+1} = x_t - \eta \times \nabla f(x_t)$$

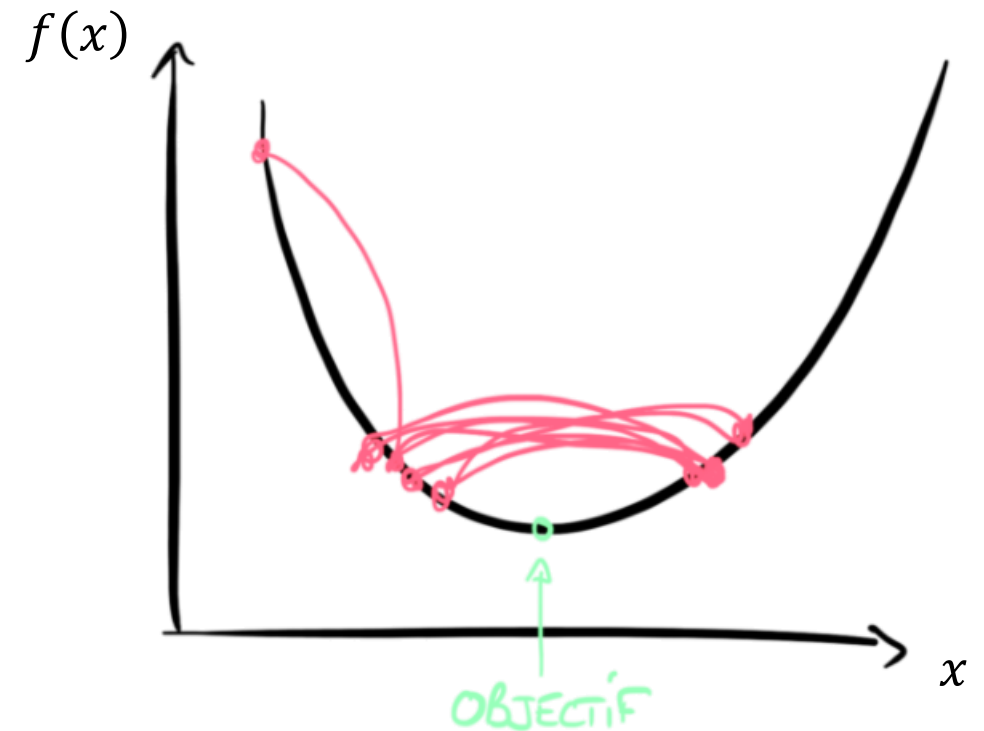


Learning Rate

- Un learning rate **trop faible** peut entraîner un temps de calcul très long



- Un learning rate **trop élevé** risque de ne pas réussir à trouver le minimum



BACKPROPAGATION

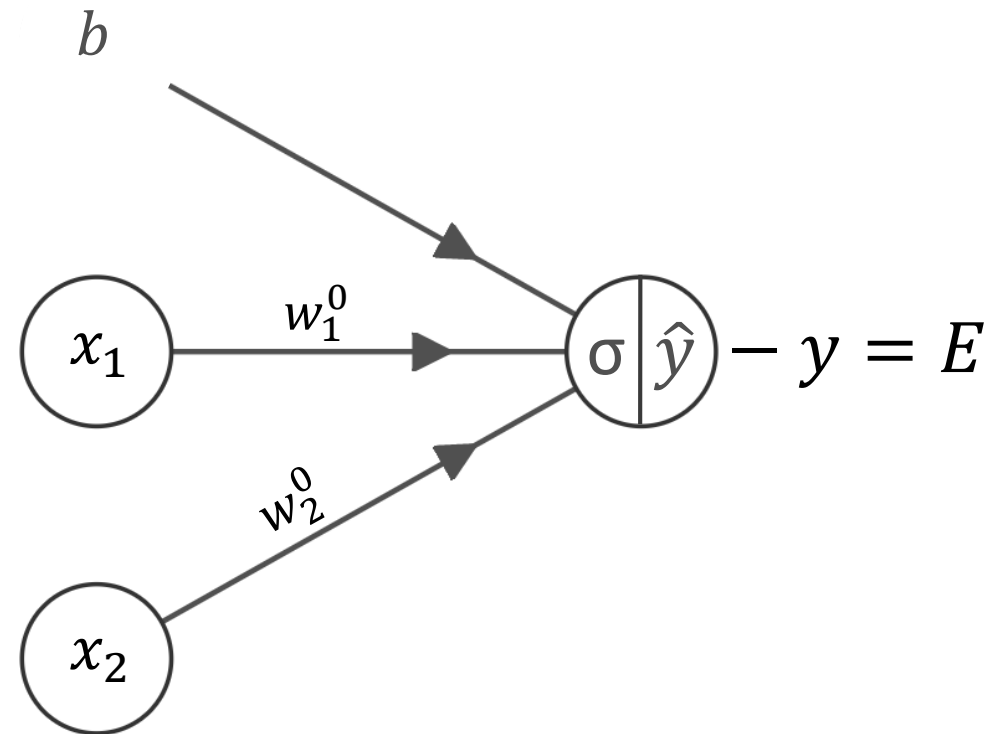
Avec le gradient descendant on peut trouver le minimum d'une fonction.
La **backpropagation** est un algorithme qui va permettre de **mettre à jour tous les poids du réseau** et permettre l'apprentissage du modèle.

$$\hat{y} = \sigma(x_1 w_1 + x_2 w_2 + b) = \sigma(z)$$

$$E = y - \hat{y} \quad \text{et} \quad \frac{\partial E}{\partial w_i} = 0$$

On calcul l'erreur pour chaque variation de poids:

$$\text{pour } w_1: \quad \frac{\partial E}{\partial w_1} = \underbrace{\frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_1}}_{\text{Chain rule}} \iff w_1^1 = w_1^0 - \alpha \frac{\partial E}{\partial z} \frac{\partial z}{\partial w_1^0} x_1$$



BACKPROPAGATION

Avec le gradient descendant on peut trouver le minimum d'une fonction. La **backpropagation** est un algorithme qui va permettre de **mettre à jour tous les poids du réseau** et permettre l'apprentissage du modèle.

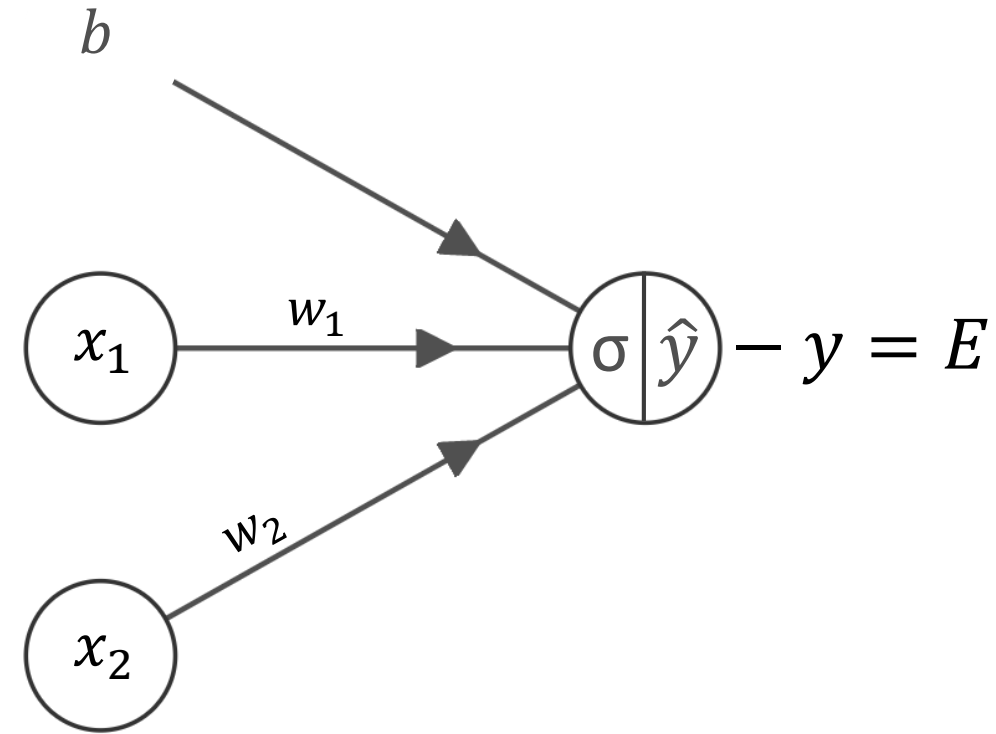
$$\hat{y} = \sigma(x_1 w_1 + x_2 w_2 + b) = \sigma(z)$$

$$E = y - \hat{y} \quad \text{et} \quad \frac{\partial E}{\partial w_i} = 0$$

On calcul l'erreur pour chaque variation de poids:

$$\text{pour } w_1: \quad \frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_1} \iff w_1^1 = w_1^0 - \alpha \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} x_1$$

$$\text{pour } w_2: \quad \frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_2} \iff w_2^1 = w_2^0 - \alpha \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} x_2$$



BACKPROPAGATION

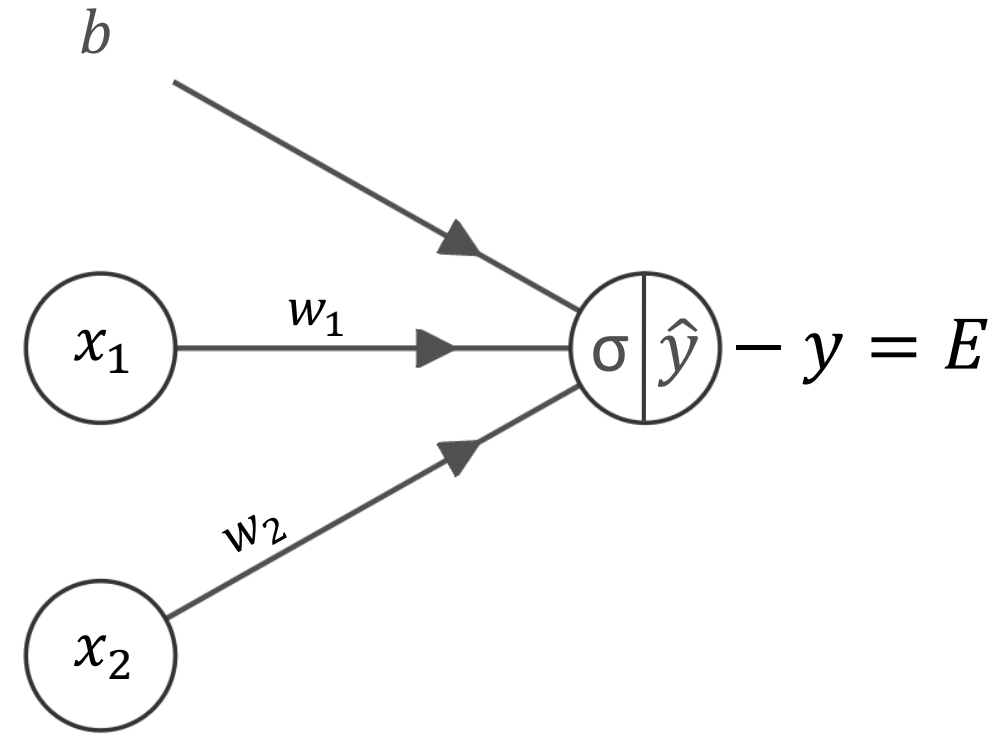
Avec le gradient descendant on peut trouver le minimum d'une fonction. La **backpropagation** est un algorithme qui va permettre de **mettre à jour tous les poids du réseau** et permettre l'apprentissage du modèle.

$$\hat{y} = \sigma(x_1 w_1 + x_2 w_2 + b) = \sigma(z)$$

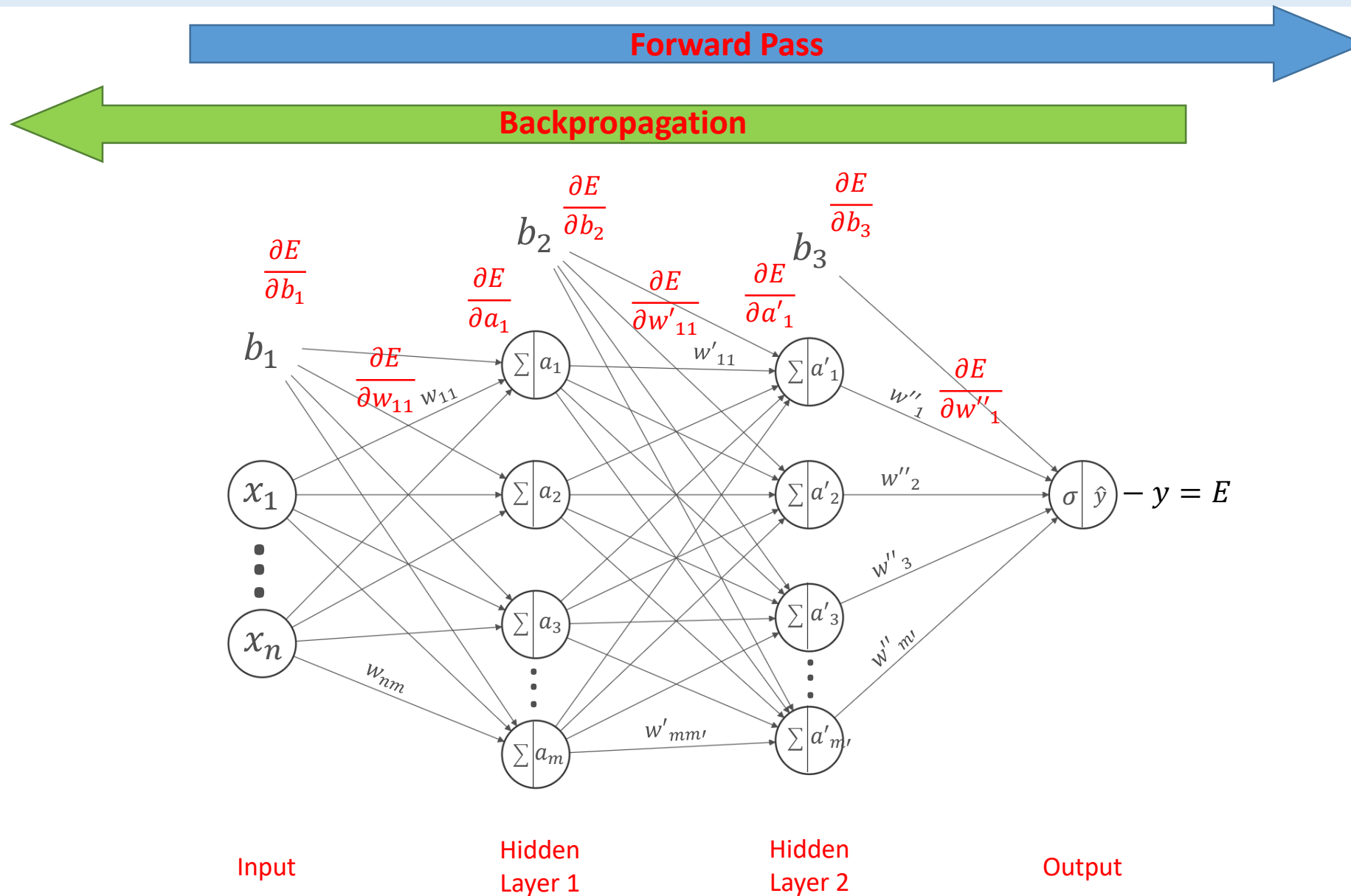
$$E = y - \hat{y} \quad \text{et} \quad \frac{\partial E}{\partial w_i} = 0$$

On calcul l'erreur pour chaque variation de poids **puis de biais**:

$$\text{pour } b: \quad \frac{\partial E}{\partial b} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \boxed{\frac{\partial z}{\partial b_1}} \quad \longleftrightarrow \quad b^1 = b^0 - \alpha \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} * \boxed{1}$$



BACKPROPAGATION



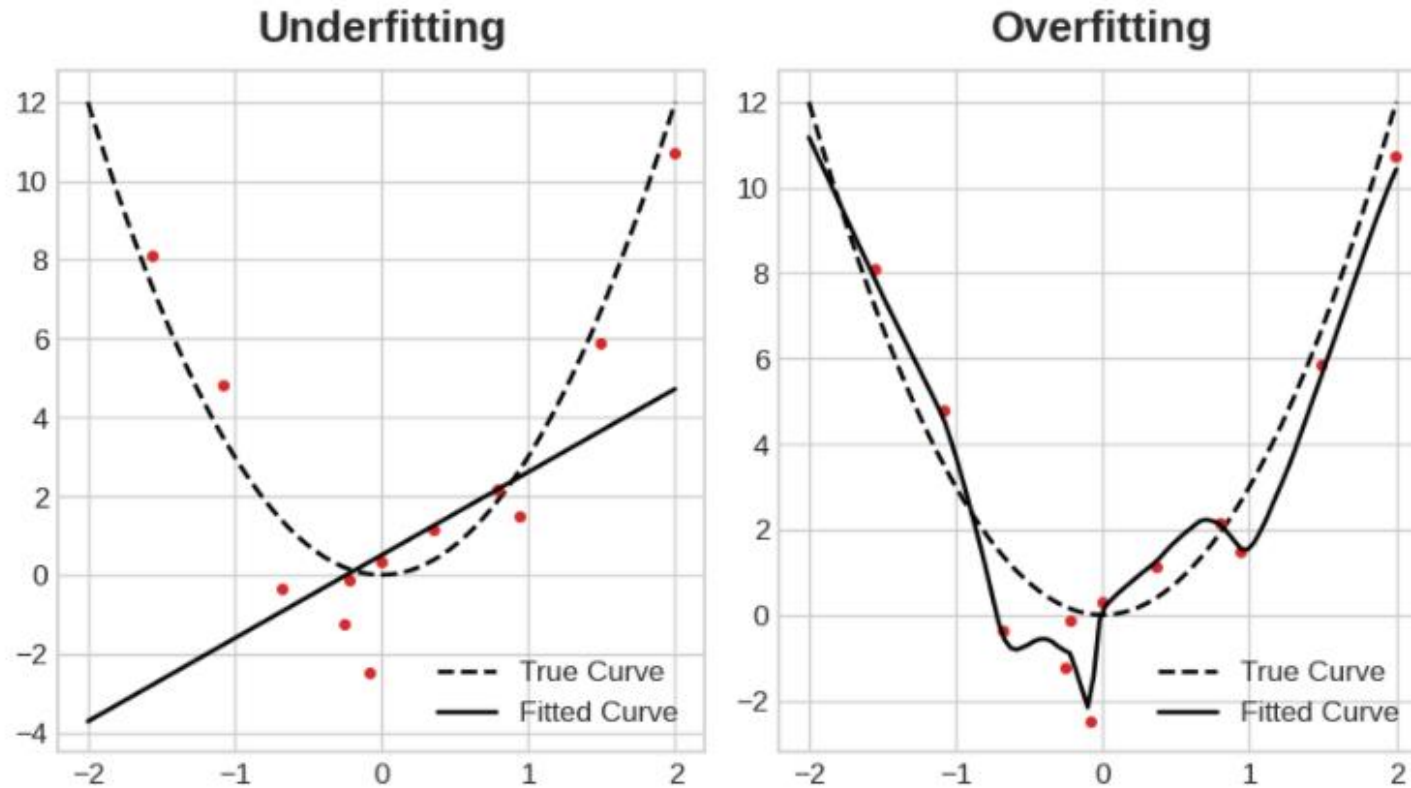
Hyperparamètres

Les **hyperparamètres** sont des paramètres que l'on doit choisir **manuellement** avant de pouvoir lancer l'entraînement de notre modèle. On retrouve:

- **Architecture**: il faut choisir l'architecture qui nous semble la plus pertinente pour résoudre notre problème (MLP, RNN, CNN)
- **Epoch**: le nombre de passage que fera chaque input dans le réseau. En effet, chaque valeur d'input (vecteur, image, texte) va passer dans le réseau afin d'optimiser au mieux les paramètres du réseau (poids, biais).
- **Batch size**: pour rendre l'entraînement plus rapide, il est souvent utile de découper le dataset d'entraînement en batch (ou mini-batch). Autrement dit, le réseau va voir les données par lots et mettre à jour les paramètres en fonction.

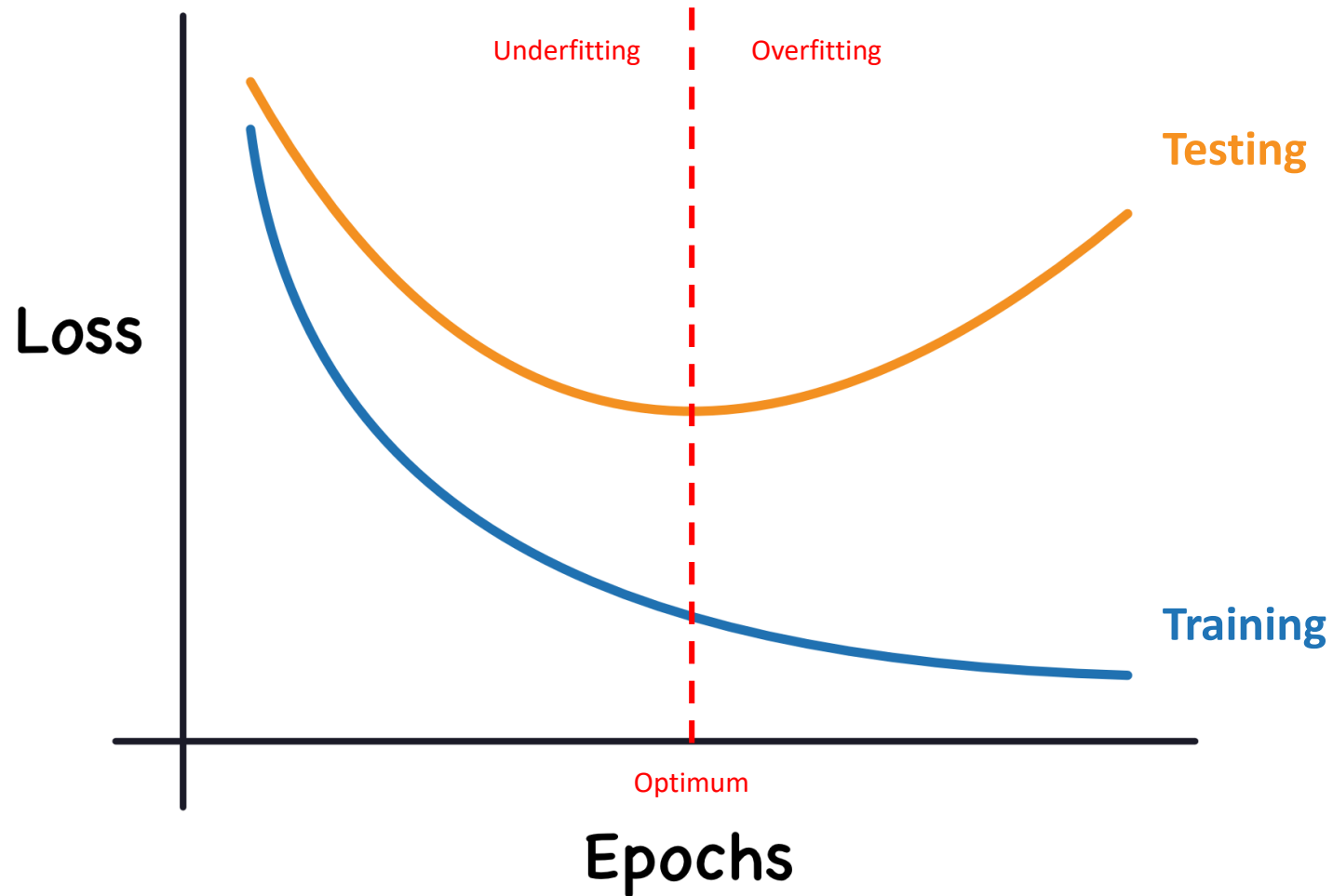
Si epoch=3 et batch size=10 pour 1000 observations alors on disposera de 100 batch. Les données du batch_1 vont d'abord passer dans le réseau 3 fois, les paramètres seront mis à jour. Puis les données du batch_2 passeront dans le réseau et ainsi de suite jusqu'au batch_100

Overfitting



Overfitting

The Learning Curves

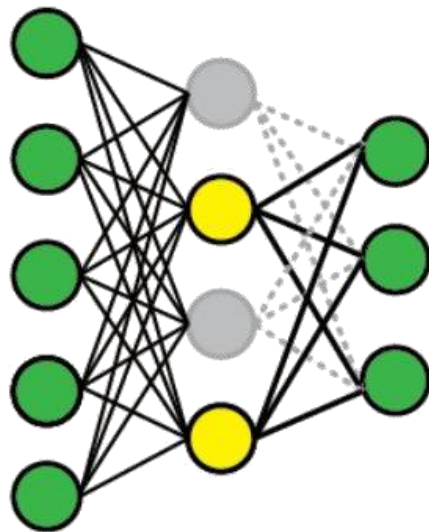


Régularisation

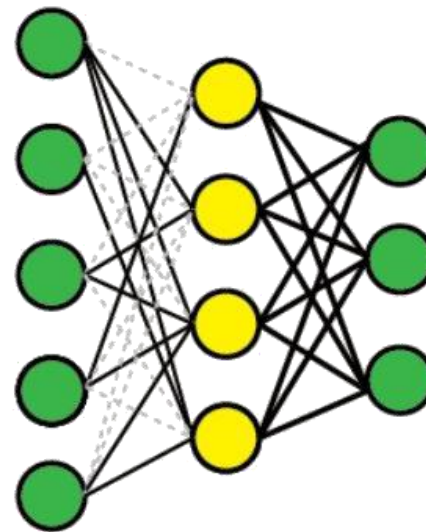
La **régularisation** a pour objectif de **prévenir du sur-apprentissage**. Différentes méthodes peuvent être utilisées :

- Méthode qui **modifie le réseau**:

Dropout



Dropconnect



Régularisation

La **régularisation** a pour objectif de **prévenir du sur-apprentissage**. Différentes méthodes peuvent être utilisées :

- Méthode qui applique une **pénalité à la fonction de perte**:

$$\sum_{i=1}^M (y_i - \hat{y}_i)^2 = \sum_{i=1}^M \left(y_i - \sum_{j=0}^p w_j \times x_{ij} \right)^2$$

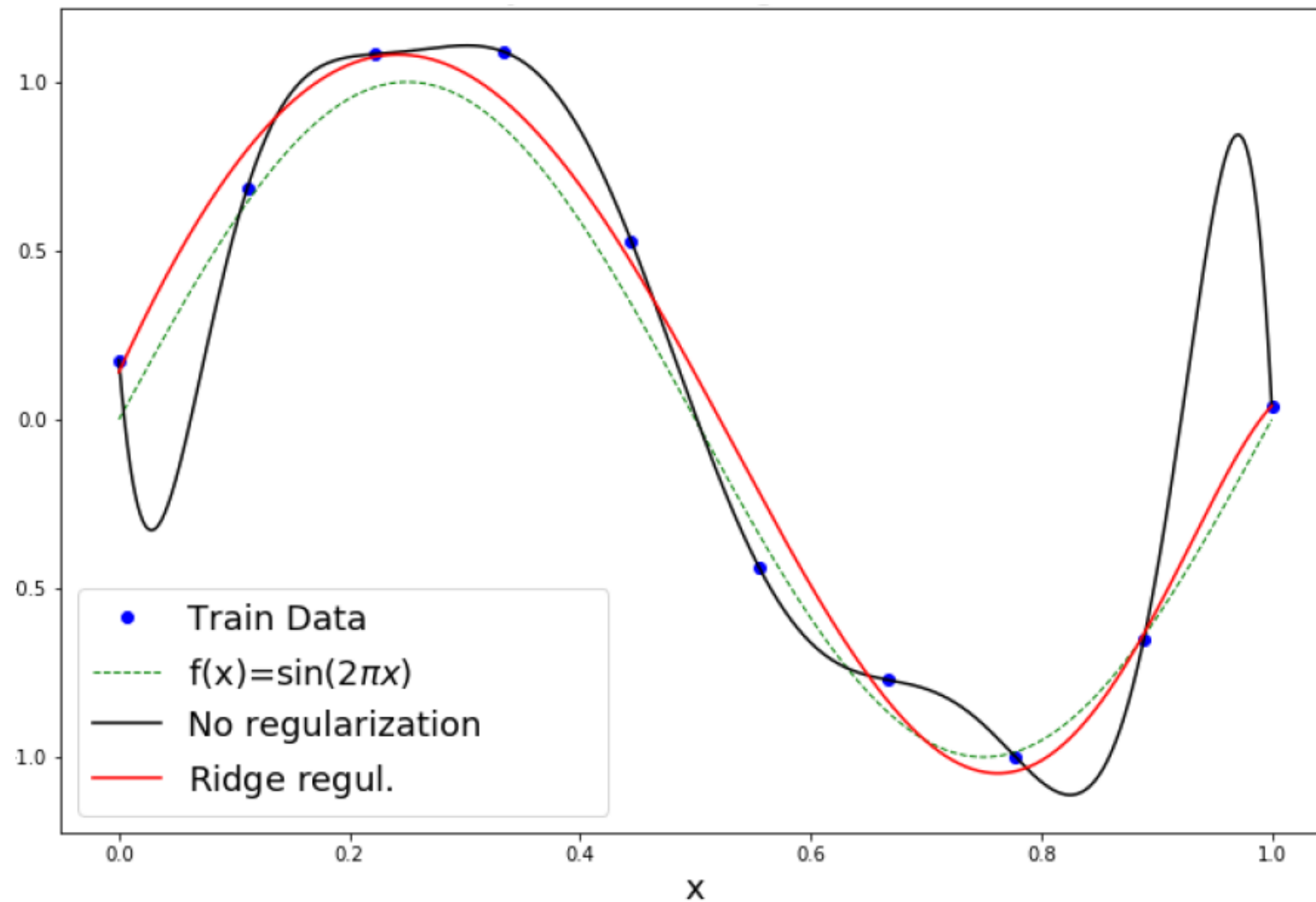
- L1 regularization (Lasso) : somme des valeurs absolues des poids

$$\sum_{i=1}^M (y_i - \hat{y}_i)^2 = \sum_{i=1}^M \left(y_i - \sum_{j=0}^p w_j \times x_{ij} \right)^2 + \lambda \sum_{j=0}^p |w_j|$$

- L2 regularization (Ridge) : somme des valeurs au carré des poids

$$\sum_{i=1}^M (y_i - \hat{y}_i)^2 = \sum_{i=1}^M \left(y_i - \sum_{j=0}^p w_j \times x_{ij} \right)^2 + \lambda \sum_{j=0}^p w_j^2$$

Régularisation



PyTorch

PyTorch est une librairie Python open source qui a été développée par Facebook. Elle permet d'effectuer des **calculs tensoriels** rapidement et de faciliter la **création** de réseaux de neurones.



Présentation rapide du TP:

1. Manipulation de tensors
2. Préparation des données
3. Développer votre réseau de neurones
4. A vous jouer !

1. Manipulation de tensors

Conversion d'une liste en tensor:

```
torch.tensor(list())
```

Conversion en tensor d'un array:

```
torch.from_numpy(np.array())
```

Tensor aléatoire:

```
torch.rand(X.shape)
```

Addition: `tensor1 + tensor2`

Multiplication: `tensor1 * tensor2`

Conversion en numpy:

```
tensor1.numpy()
```

Transforme un tensor de dimension n en dimension 1:

```
torch.squeeze()
```

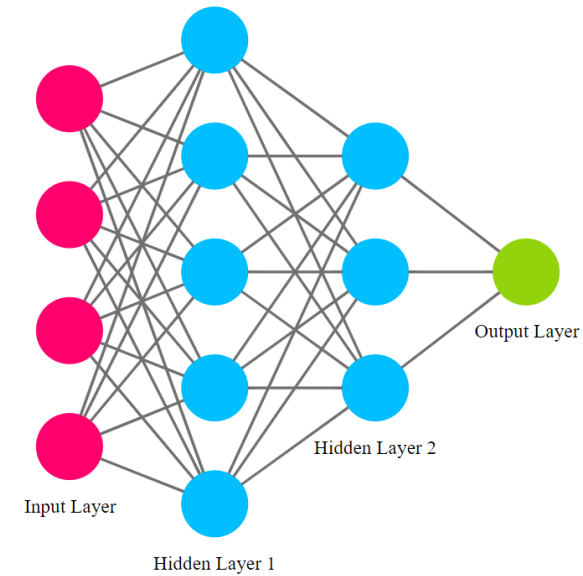

3. Développer votre réseau de neurones

```
# Définir la classe MLP
class MLP(nn.Module):

    def __init__(self, n_features):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(n_features, 5)
        self.fc2 = nn.Linear(5, 3)
        self.fc3 = nn.Linear(3, 1)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return torch.sigmoid(self.fc3(x))
```

PyTorch



3. Développer votre réseau de neurones

PyTorch

```
# Déterminer la loss function  
criterion = nn.BCELoss()
```

```
# Optimizer (stochastig gradient descendant)  
optimizer = optim.SGD()
```

```
# Prédiction avec le réseau sur trainset  
model(trainset)
```

```
# Calculer la perte entre prédiction et vraie valeur  
criterion()
```

```
# Gradient à zéro  
optimizer.zero_grad()
```

```
# Backproapagation  
train_loss.backward()
```

```
# Mise à jour des poids  
optimizer.step()
```

```
# Résultat de classification  
classification_report()
```

