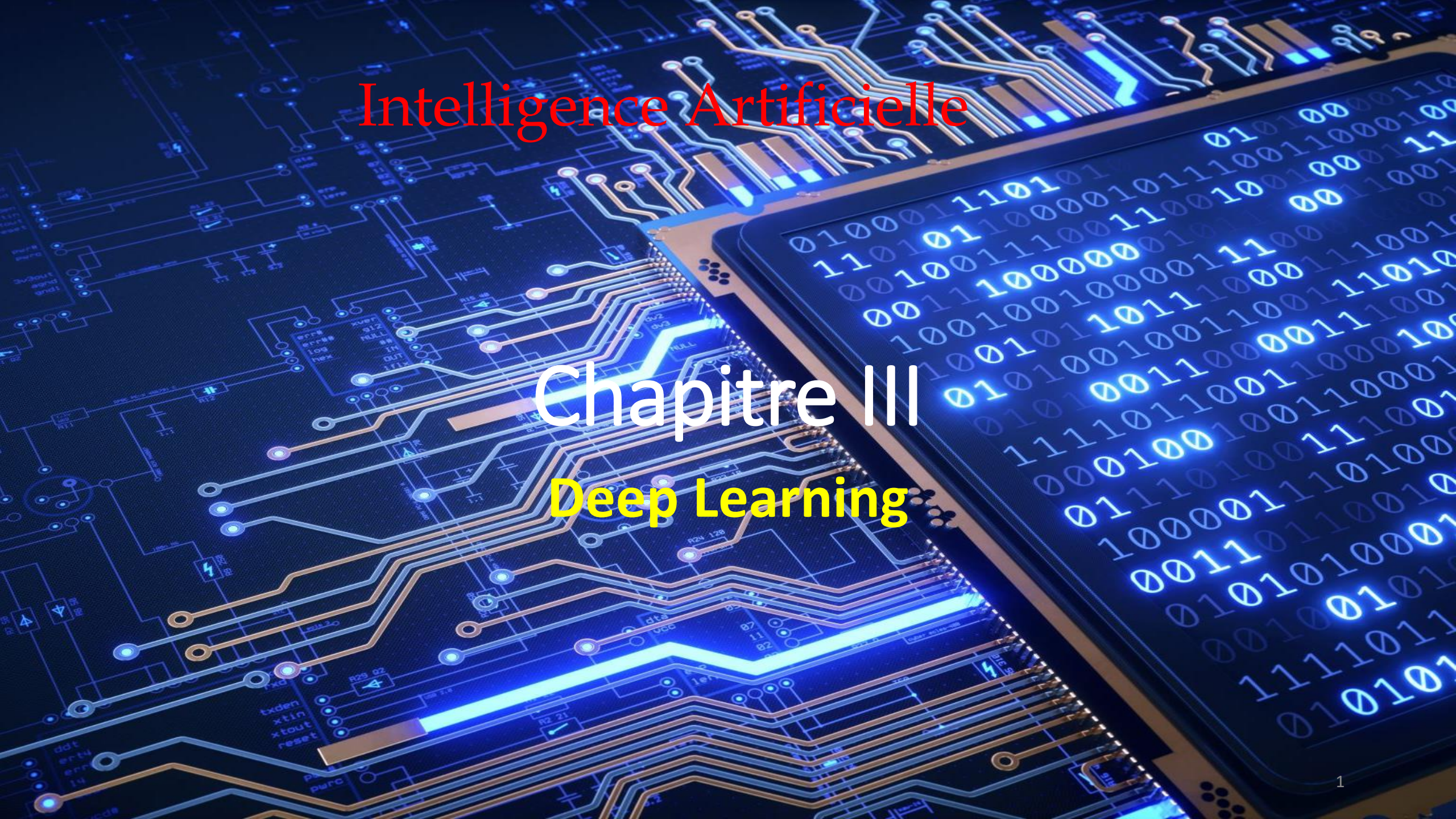
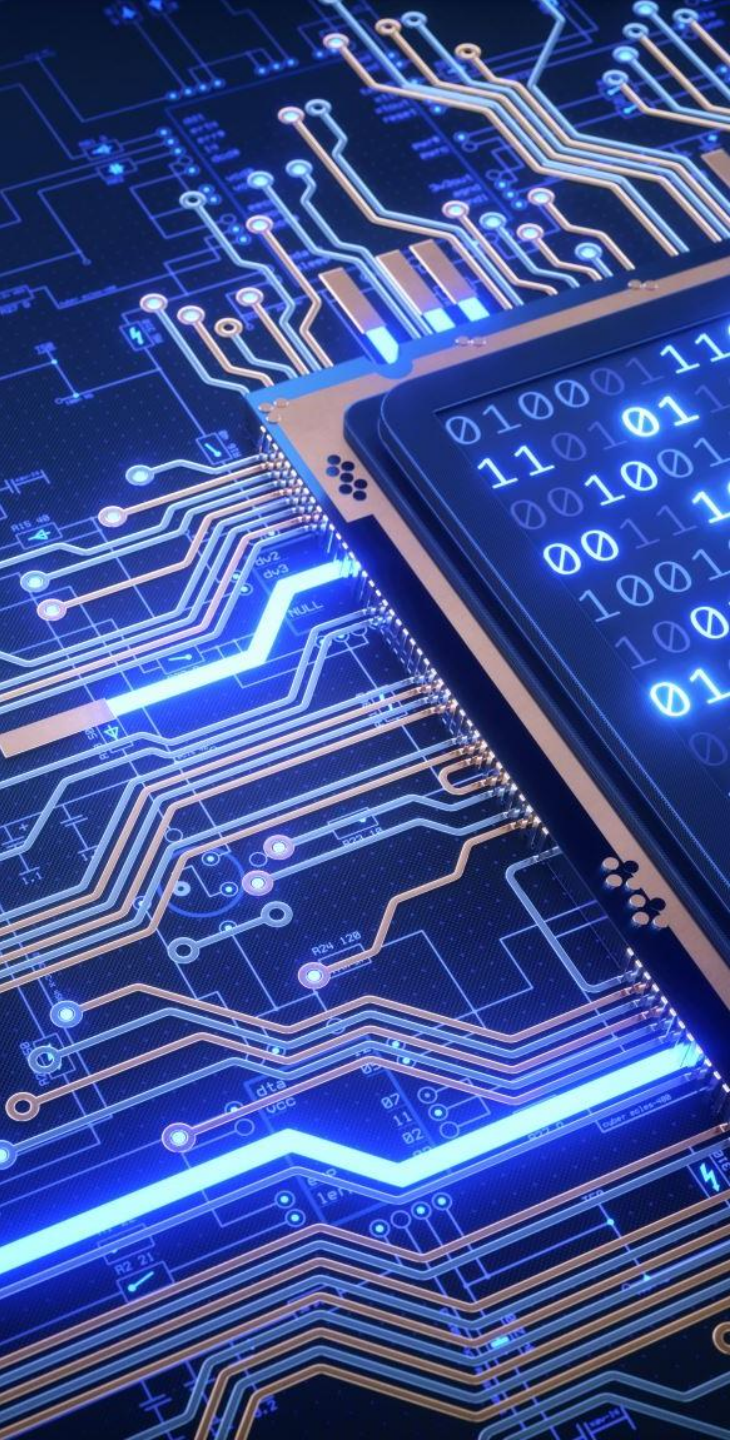

Intelligence Artificielle
Chapitre III
Deep Learning



Plan

- **Du modèle Perceptron aux réseaux de neurones**
- **Fonctions d'Activation**
- **Fonctions de Coût d'erreur (Cost)**
- **Rétropropagation (BackPropagation)**
- **Normalisation**
- **Optimisation**

Introduction

- Le chapitre décrit les connaissances de base de l'apprentissage en profondeur, y compris l'historique du développement de l'apprentissage en profondeur, les composants et les types de réseaux de neurones d'apprentissage en profondeur, et les problèmes courants dans les projets d'apprentissage en profondeur.
- Les objectifs :
    - Décrire la définition et le développement des réseaux de neurones.
    - Découvrez les composants importants des réseaux de neurones d'apprentissage en profondeur. Comprendre la formation et l'optimisation des réseaux de neurones.
    - Décrire les problèmes courants de l'apprentissage en profondeur.

Introduction

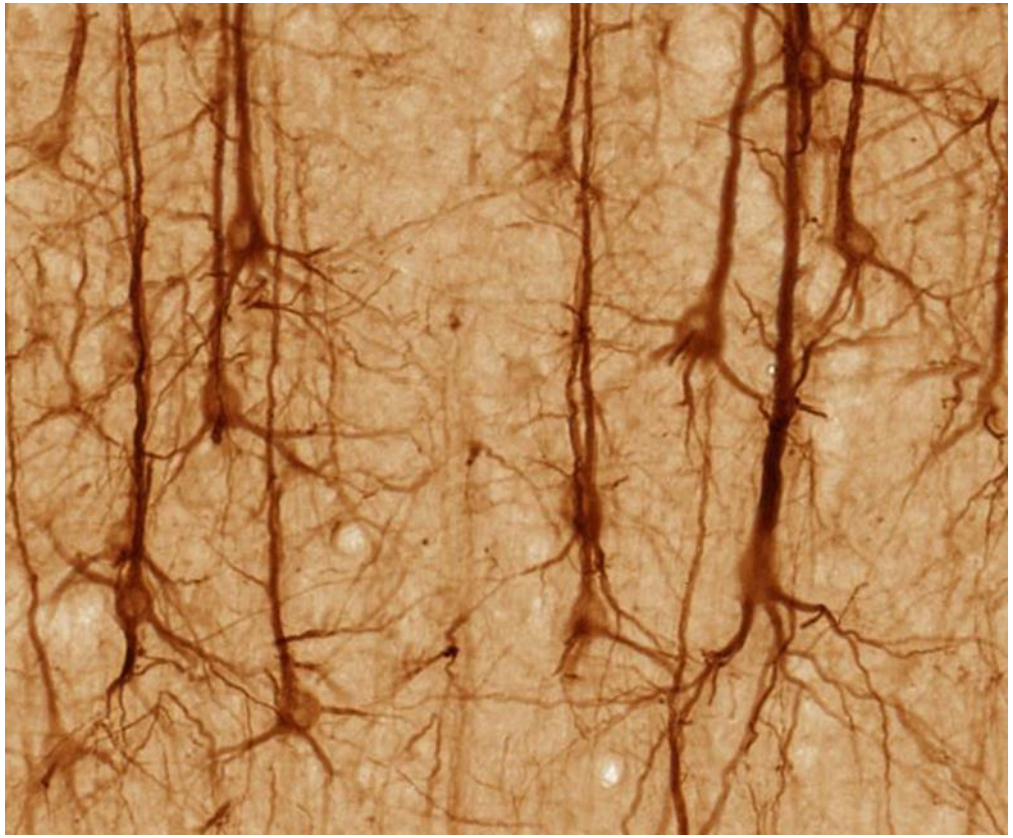
- Pour commencer à comprendre le Deep Learning, nous allons construire nos modèles d'abstraction :
  - Neurone biologique seul
  - Perceptron
  - Modèle de Perceptron Multicouche
  - Réseau de Neurones Deep Learning
- Au fur et à mesure que nous apprendrons des modèles plus complexes, nous introduirons également des concepts tels que :
  - Fonctions d'Activation
  - Gradient Descent - Descente de Gradient
  - BackPropagation - rétropropagation

Apprentissage automatique vs apprentissage profond

ML	DL
Faible configuration matérielle requise sur l'ordinateur : étant donné la quantité de calcul limitée, l'ordinateur n'a généralement pas besoin de GPU pour le calcul parallèle.	Exigences matérielles plus élevées sur l'ordinateur : pour exécuter des opérations matricielles sur des données massives, l'ordinateur a besoin d'un GPU pour effectuer un calcul parallèle.
Applicable à l'entraînement avec une petite quantité de données et dont les performances ne peuvent pas être améliorées en continu à mesure que la quantité de données augmente.	Les performances peuvent être élevées lorsque des paramètres de poids dimensionnels élevés et des données d'entraînement massives sont fournis.
Répartition des problèmes niveau par niveau	Apprentissage de bout en bout
Sélection manuelle des caractéristiques	Extraction automatique de caractéristiques basée sur un algorithme
Fonctionnalités faciles à expliquer	Fonctionnalités difficiles à expliquer

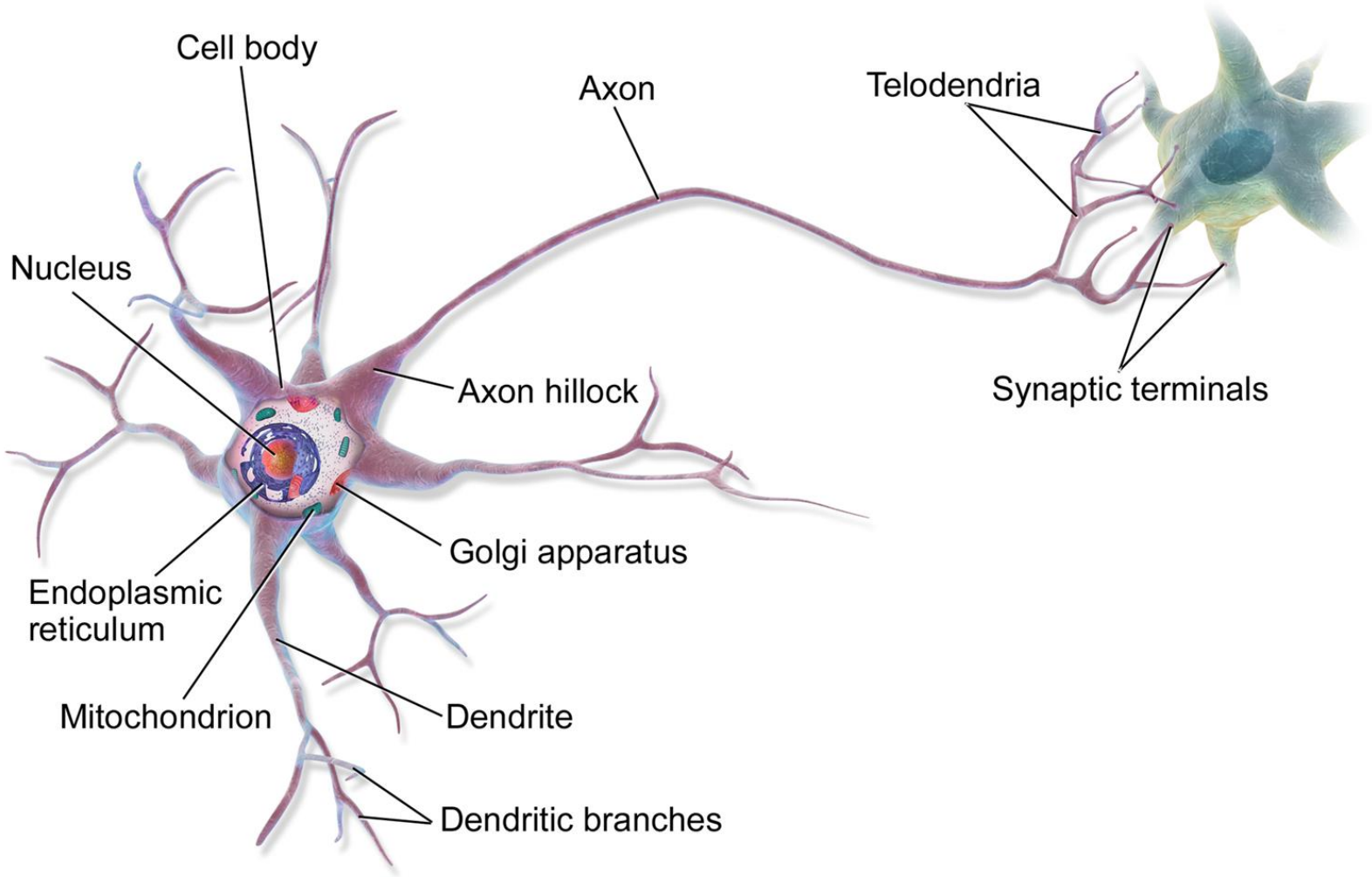
Modèle Perceptron

- Si l'idée générale du Deep Learning est de faire en sorte que les ordinateurs imitent artificiellement l'intelligence naturelle biologique, nous devrions probablement acquérir une compréhension générale du fonctionnement des neurones biologiques !

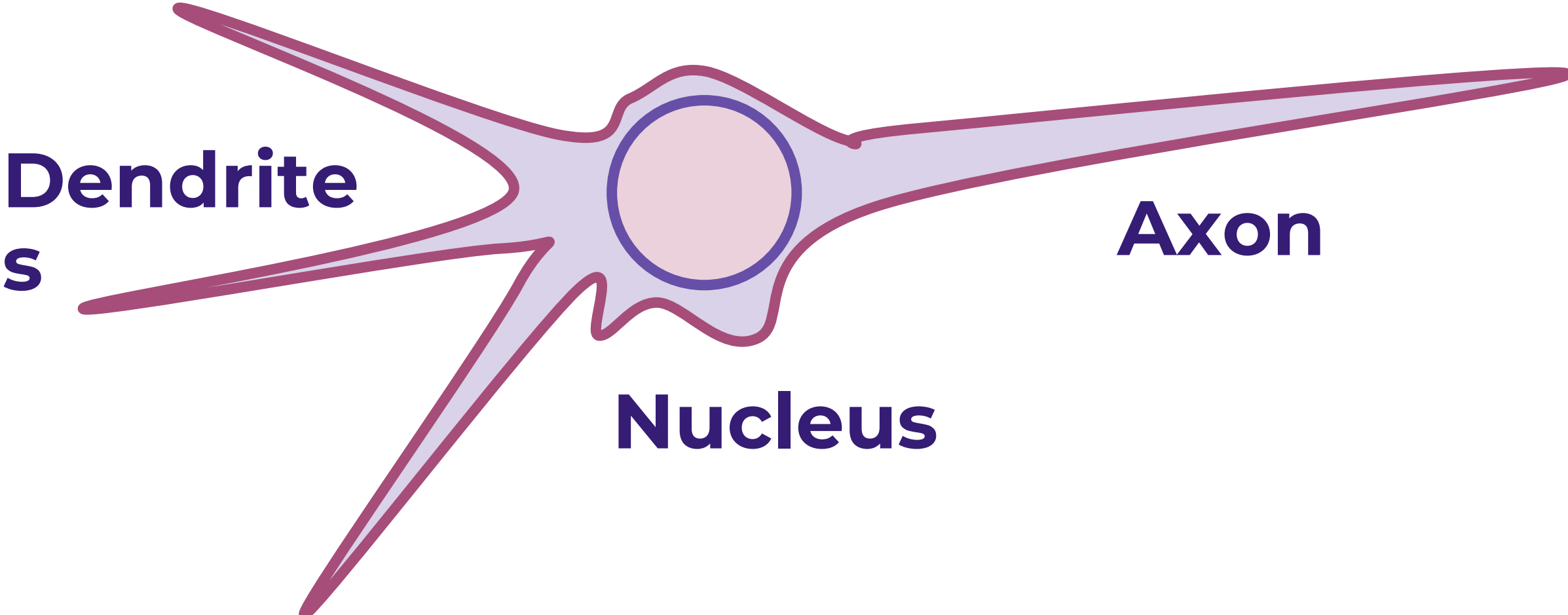


Neurones colorés dans le cortex cérébral

Illustration des neurones biologiques



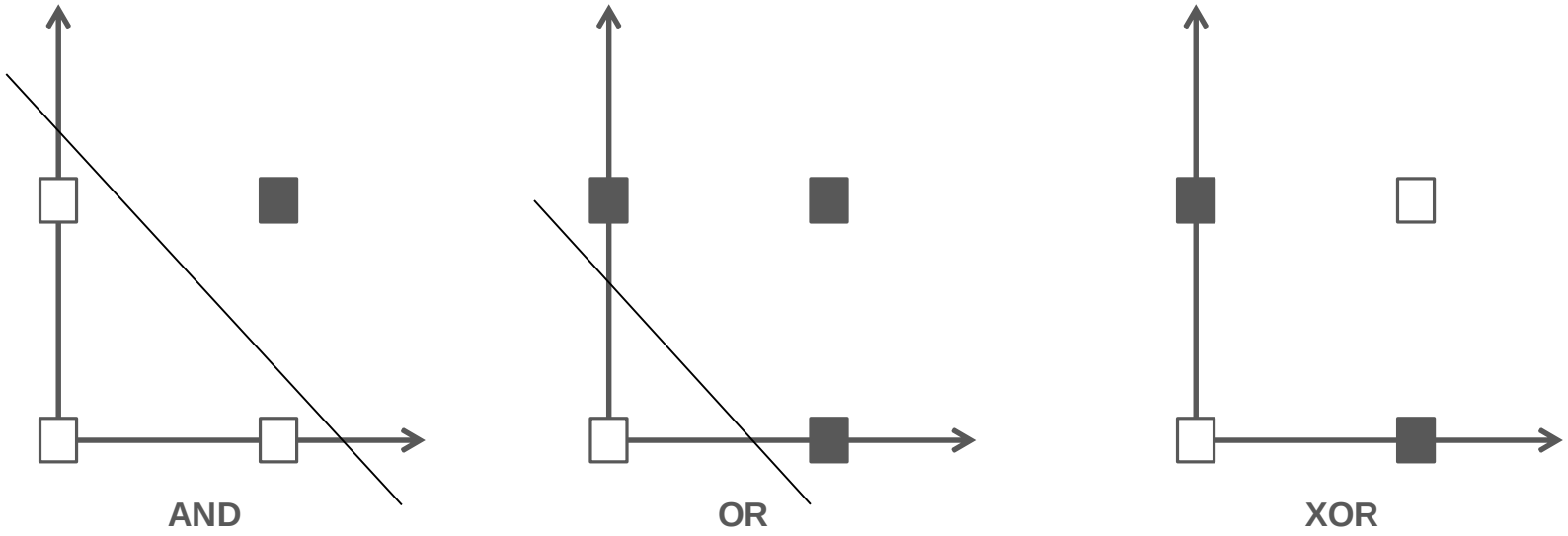
Modèle simplifié de neurone biologique



Modèle Perceptron

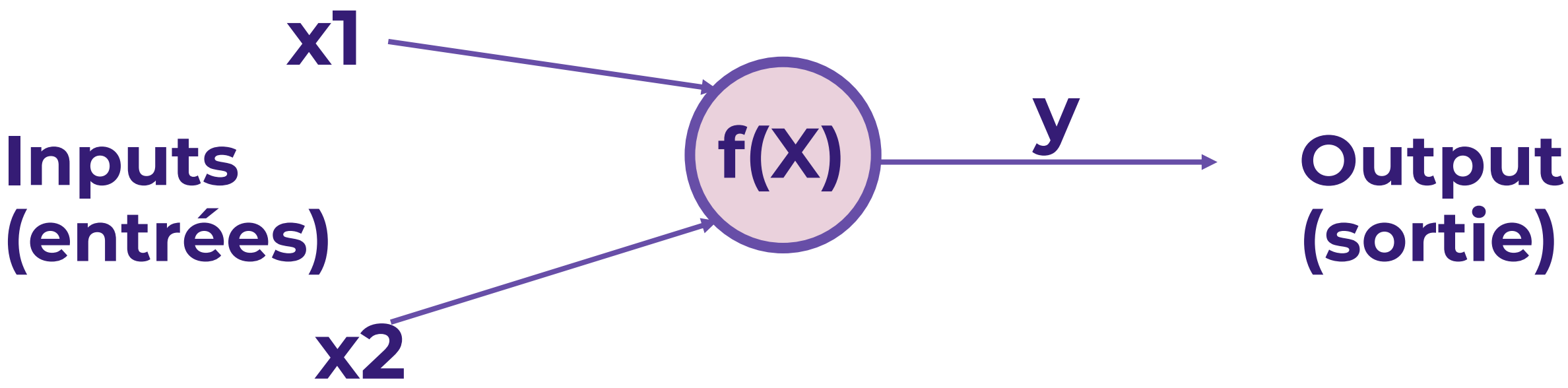
- Un perceptron est une forme de réseau de neurones introduite en 1958 par Frank Rosenblatt.
- Étonnamment, même à l'époque, il voyait un énorme potentiel : *"...perceptron pourrait éventuellement être capable d'apprendre, de prendre des décisions et de traduire des langues."*
- Cependant, en 1969, Marvin Minsky et Seymour Papert ont publié leur livre **Perceptrons**.
- Ce livre suggérait qu'il y avait de sérieuses limites à ce que les perceptrons pouvaient faire.
- Cela a marqué le début de ce que l'on appelle l'hiver de l'IA, avec peu de financement pour l'IA et les réseaux de neurones dans les années 1970.

XOR



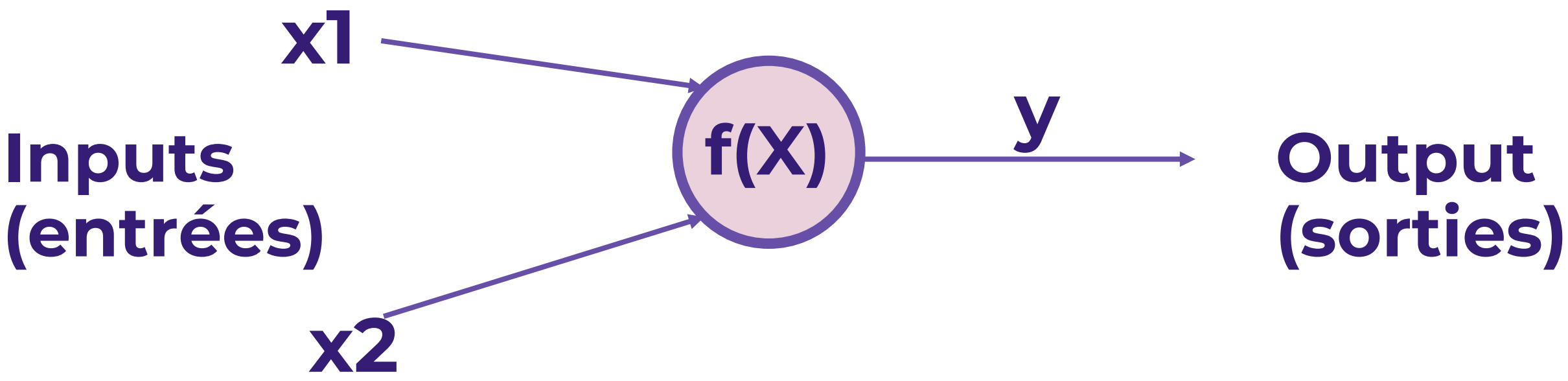
Modèle Perceptron

- Prenons un exemple simple



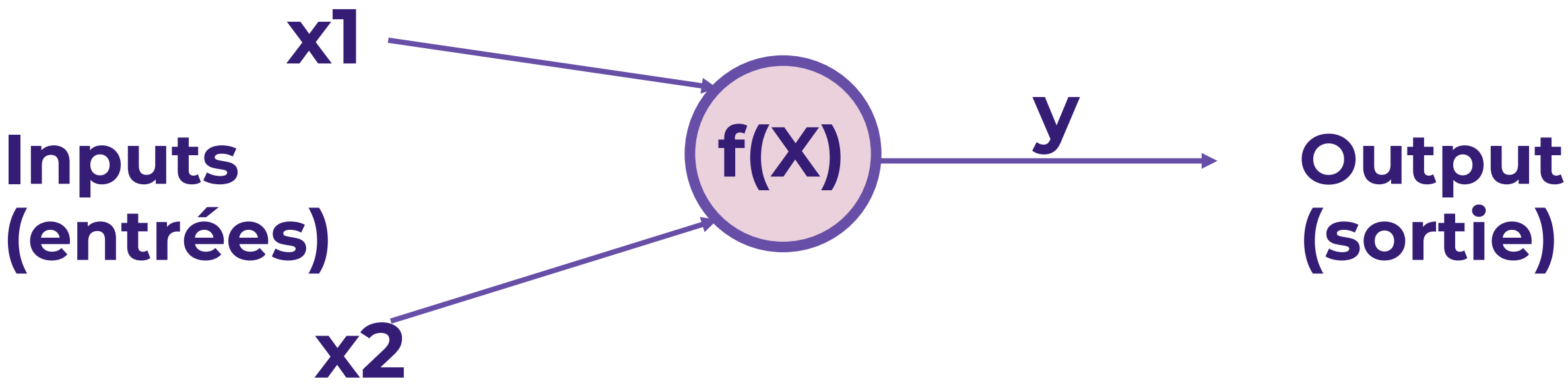
Modèle Perceptron

- Si f(X) est juste une somme, alors y=x1+x2



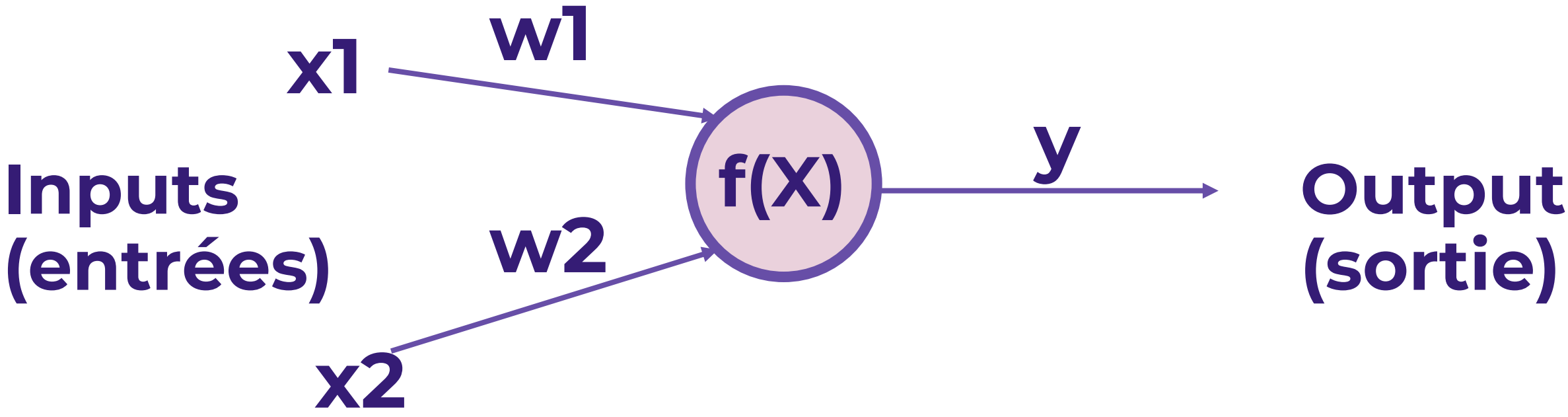
Modèle Perceptron

- De manière réaliste, nous voudrions pouvoir ajuster certains paramètres dans un but “d’apprentissage”



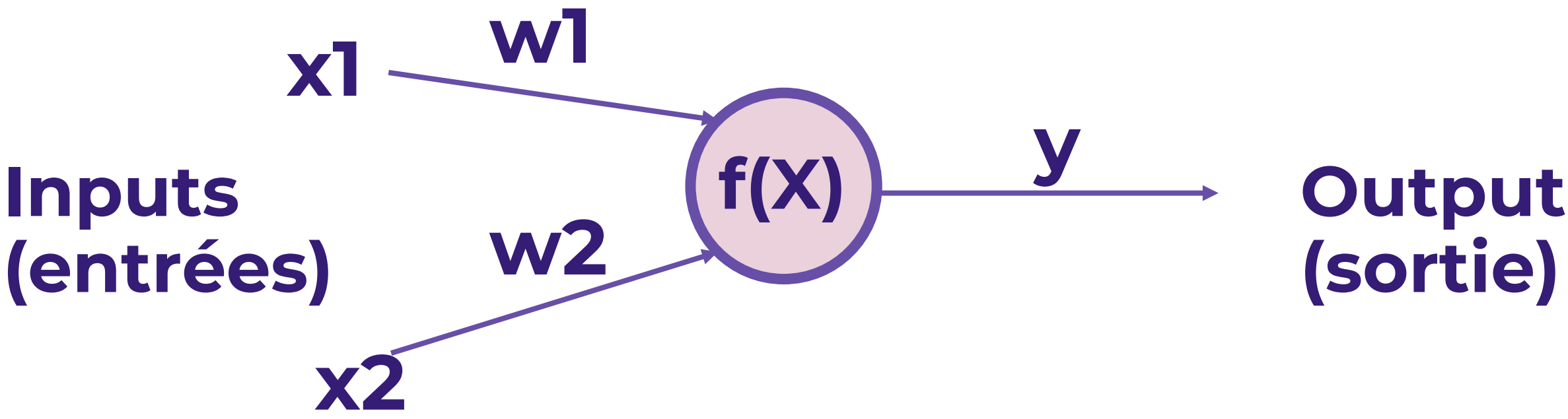
Modèle Perceptron

- Ajoutons un poids ajustable que nous multiplions par rapport à x



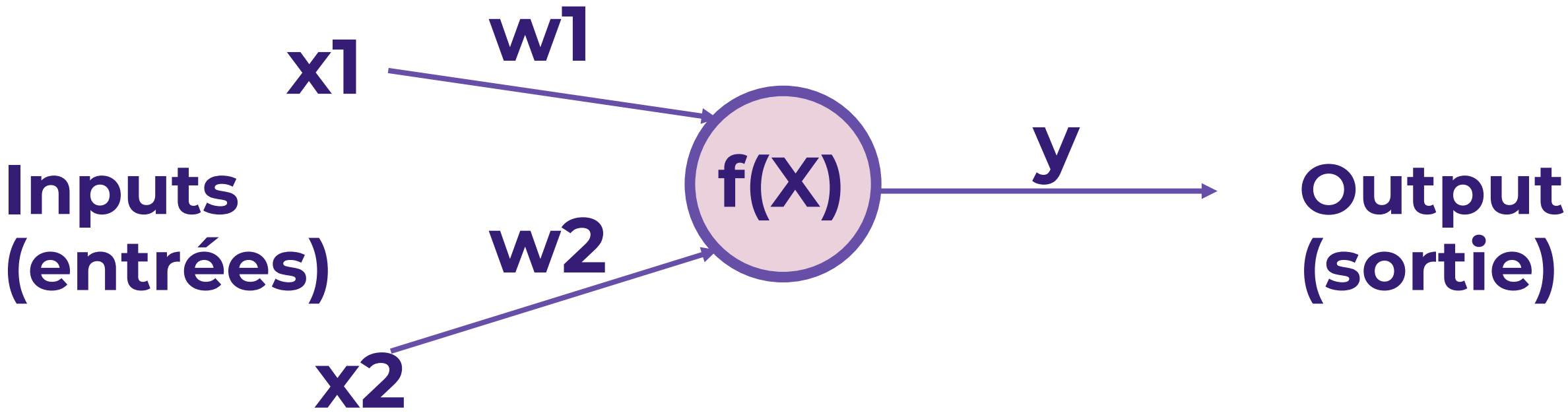
Modèle Perceptron

- Maintenant **y = x1w1 + x2w2**



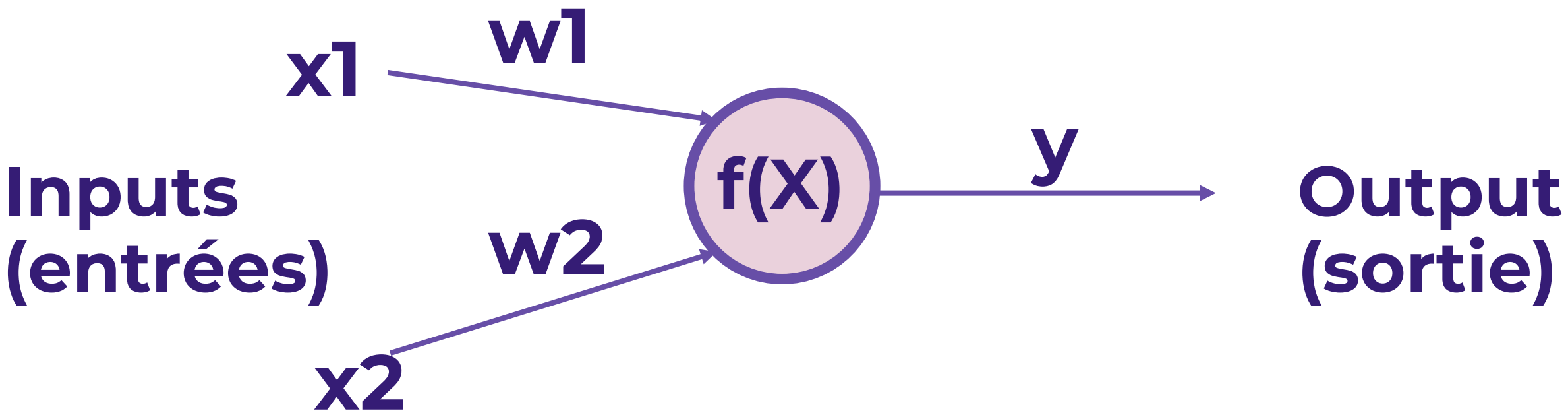
Modèle Perceptron

- Nous pourrions mettre à jour les **pondérations** pour affecter **y**



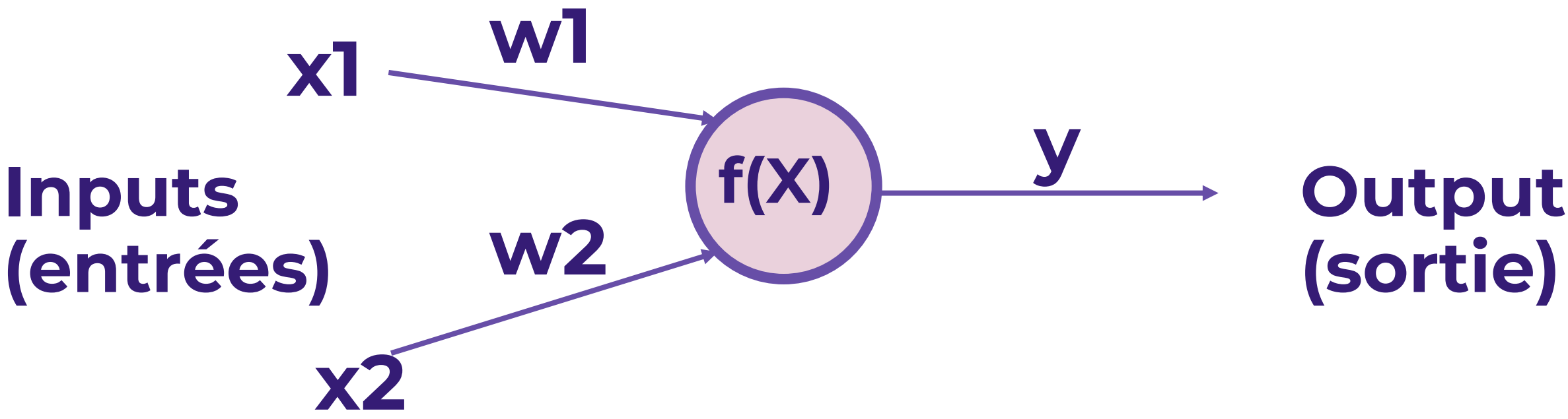
Modèle Perceptron

- Mais si un **x** est égal à zéro, **w** ne changera rien !



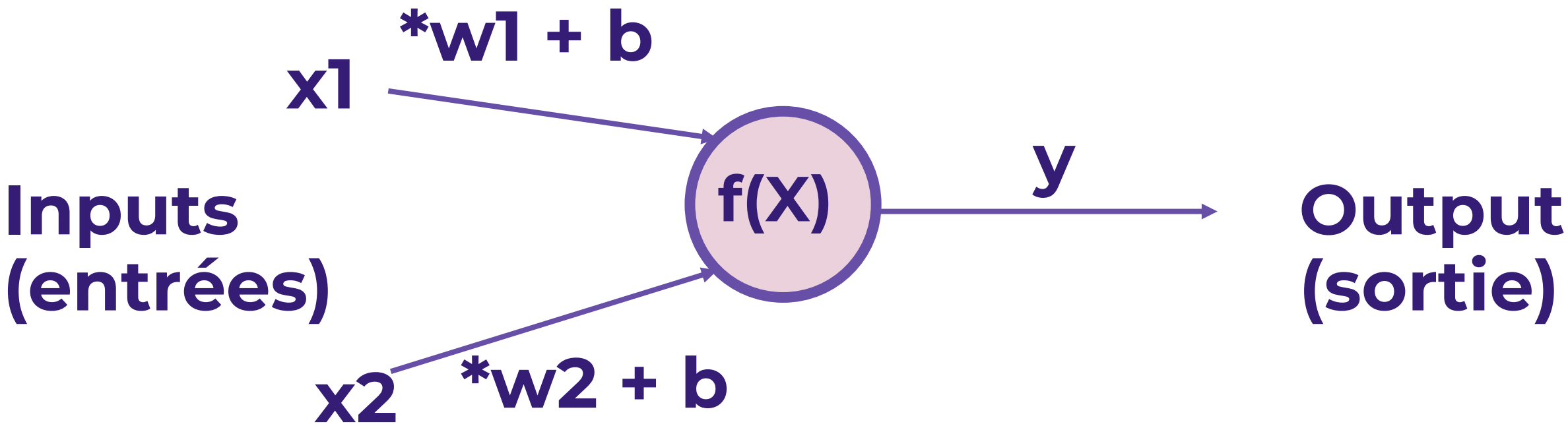
Modèle Perceptron

- Ajoutons un terme de **biais b** aux entrées.



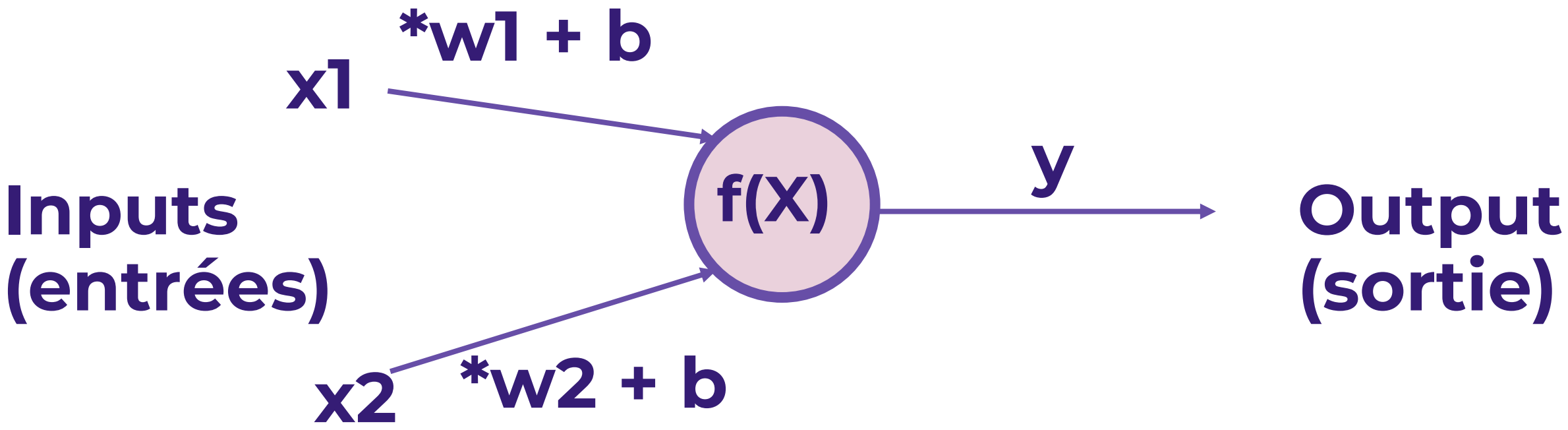
Modèle Perceptron

- Ajoutons un terme de **biais b** aux inputs.



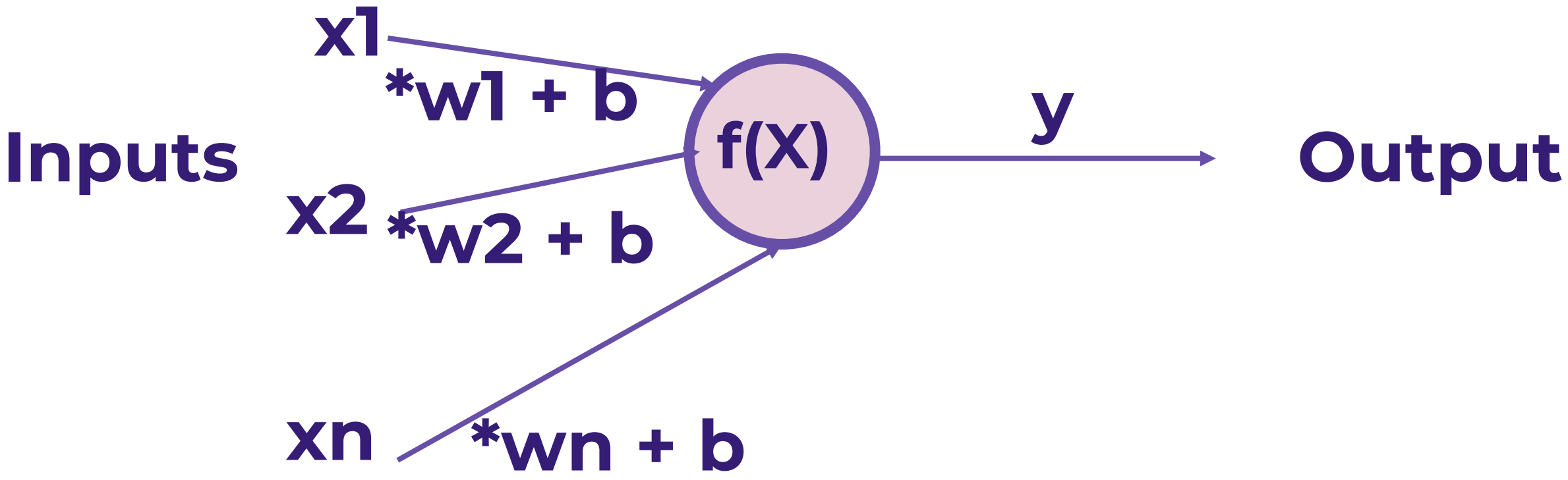
Modèle Perceptron

- y = (x1w1 + b) + (x2w2 + b)



Modèle Perceptron

- Nous pouvons étendre ce phénomène et le généraliser :



Modèle Perceptron

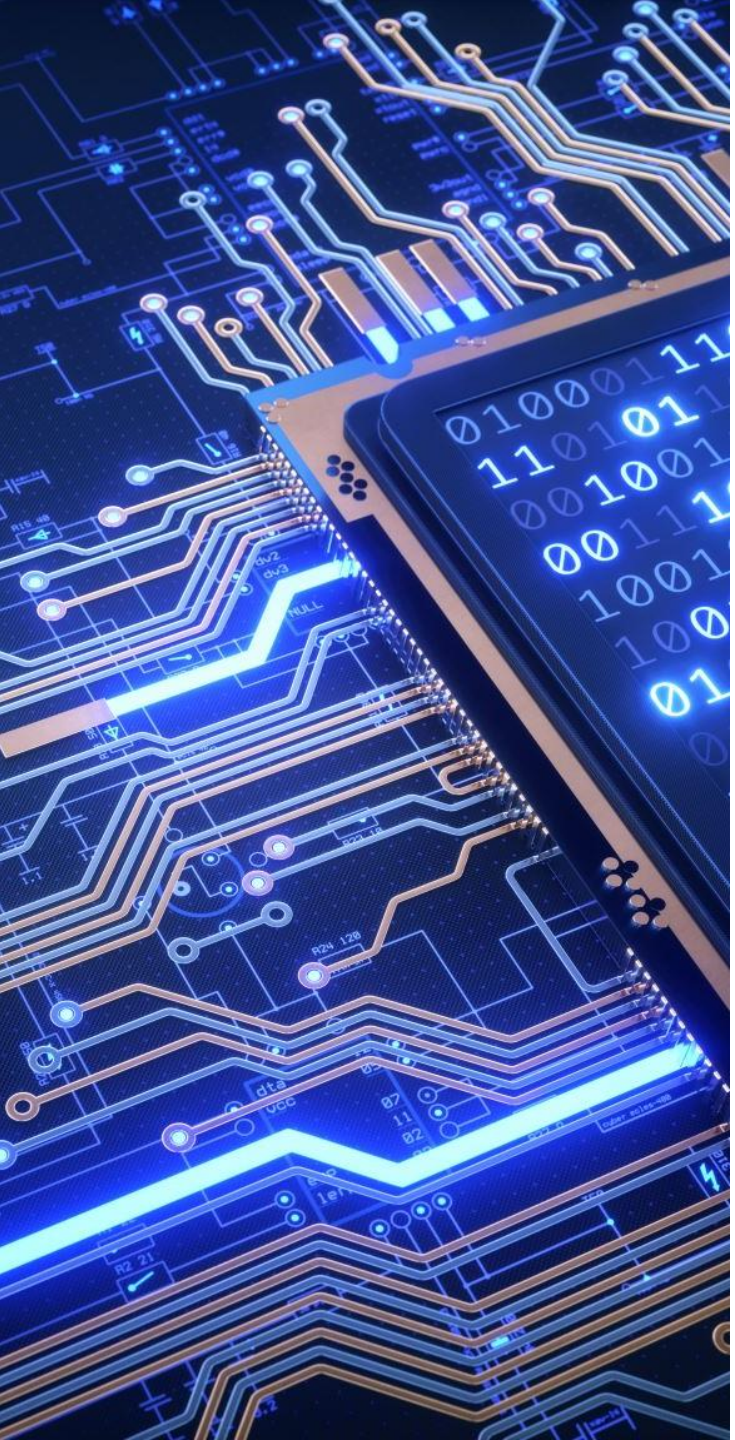
- Nous avons réussi à modéliser un neurone biologique comme un simple perceptron ! Mathématiquement, notre généralisation est :

$$\hat{y} = \sum_{i=1}^{n} x_i w_i + b_i$$

Modèle Perceptron

- Nous verrons plus tard comment nous pouvons étendre ce modèle pour que X soit un **tenseur** d'informations (une matrice à n dimensions).

$$\hat{y} = \sum_{i=1}^{n} x_i w_i + b_i$$



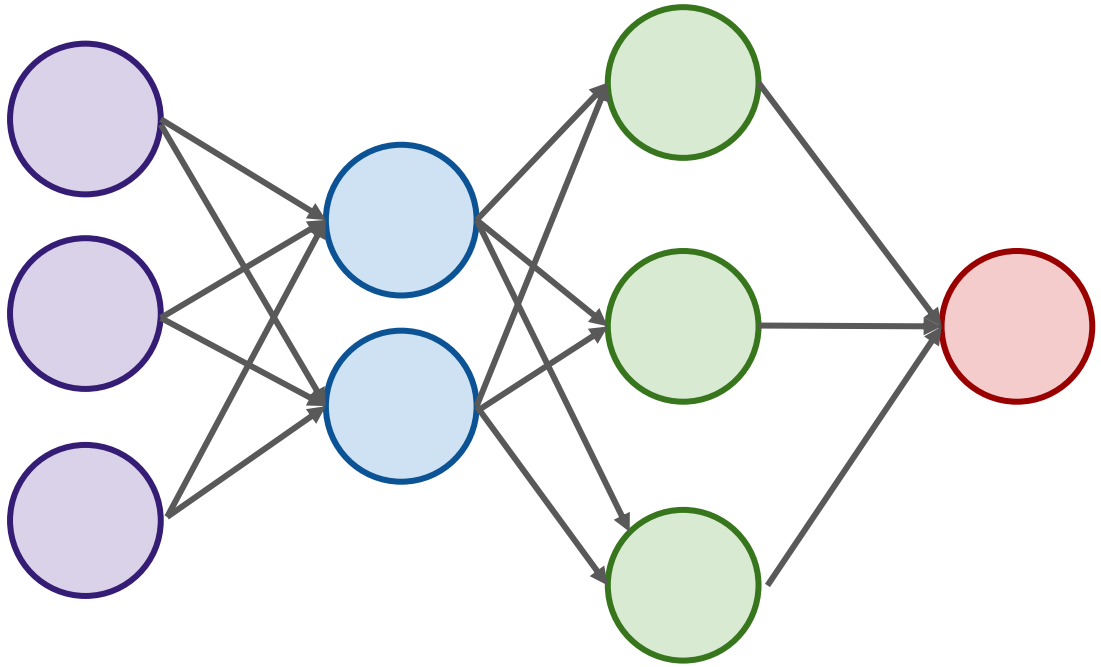
Réseaux de Neurones

Réseaux de Neurones

- Un seul perceptron ne suffira pas pour apprendre des systèmes complexes.
- Heureusement, nous pouvons développer l'idée d'un perceptron unique, pour créer un modèle de perceptron multicouche (multi-layer).
- Nous allons également introduire l'idée de fonctions d'activation.

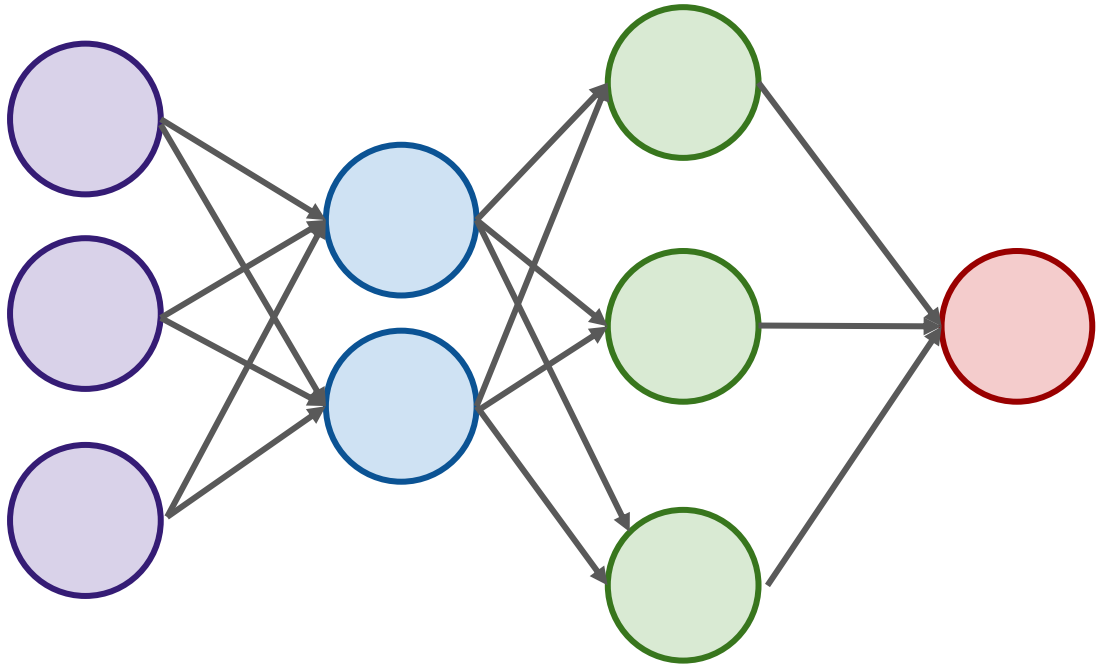
Réseaux de Neurones

- Pour construire un réseau de perceptrons, nous pouvons connecter des couches de perceptrons, en utilisant un **modèle de perceptron multicouche.**



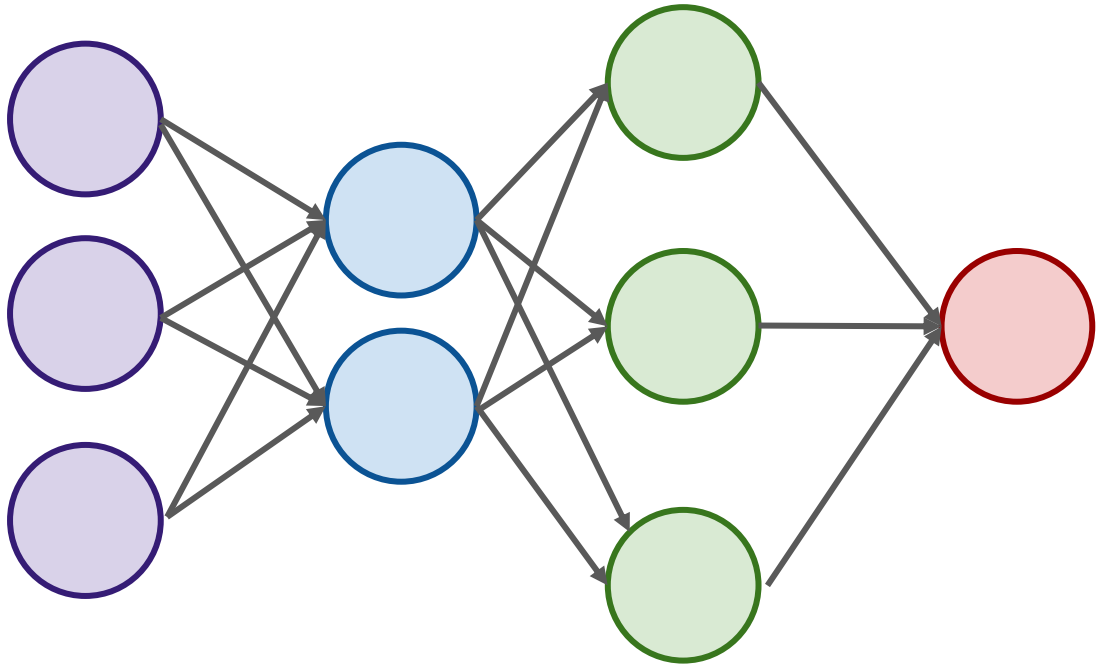
Réseaux de Neurones

- Les outputs d'un perceptron sont directement transmises comme inputs à un autre perceptron.



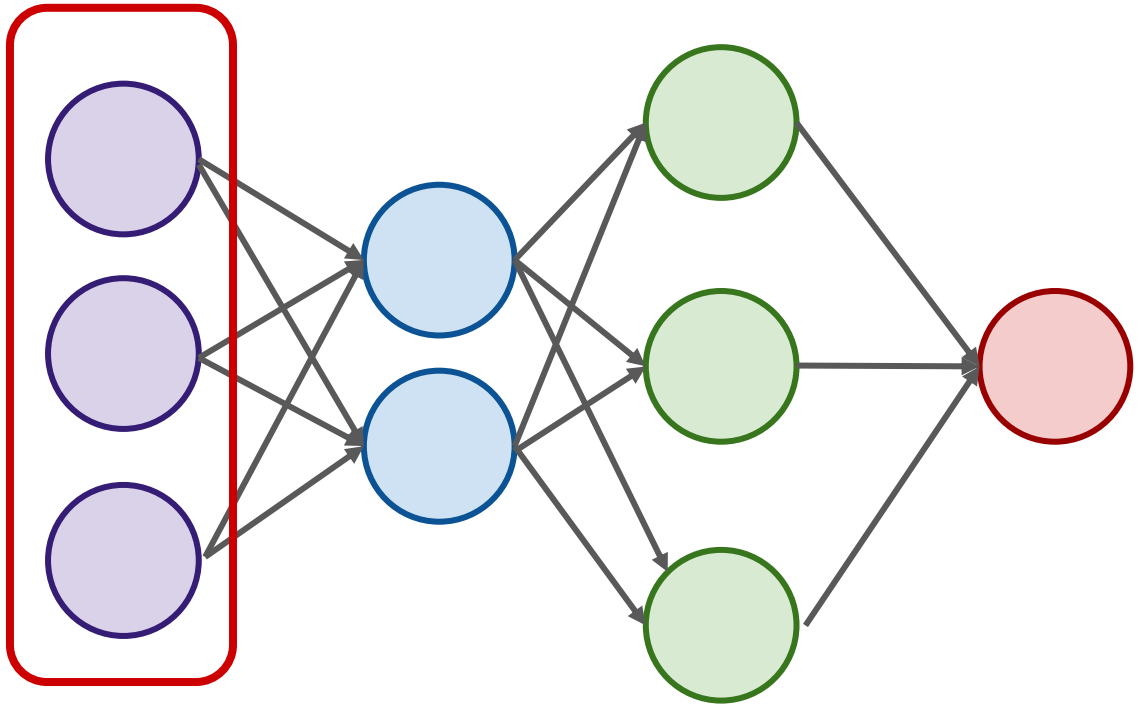
Réseaux de Neurones

- Cela permet au réseau dans son ensemble de connaître les interactions et les relations entre les features (caractéristiques).



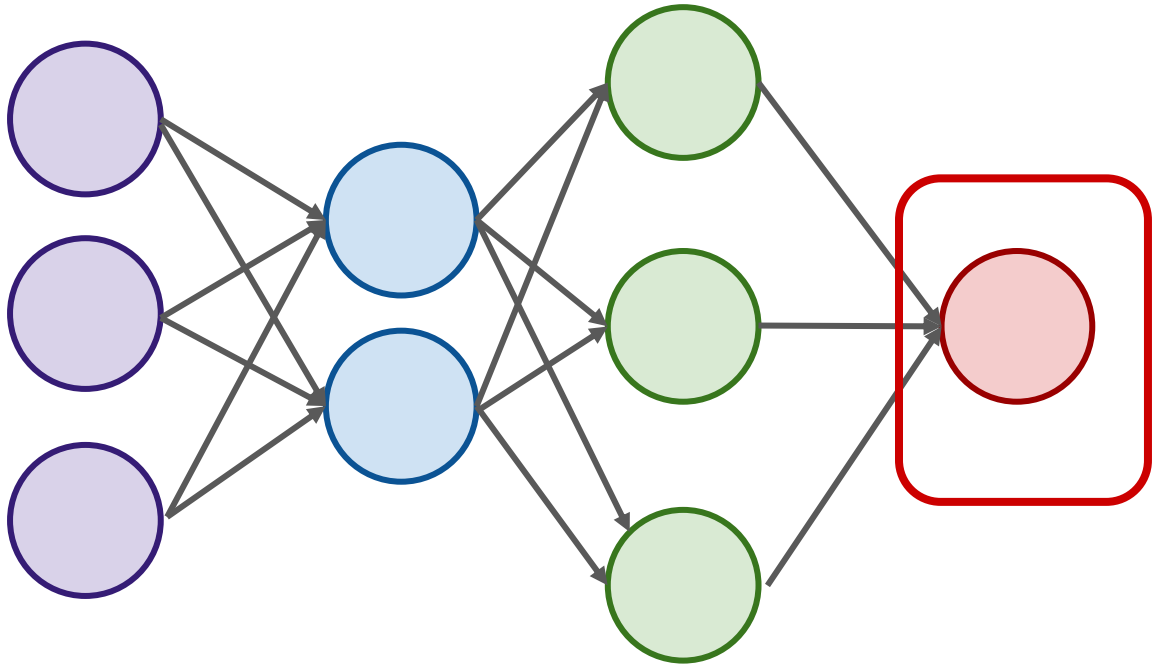
Réseaux de Neurones

- La première couche est la **input layer (couche d'entrée)**



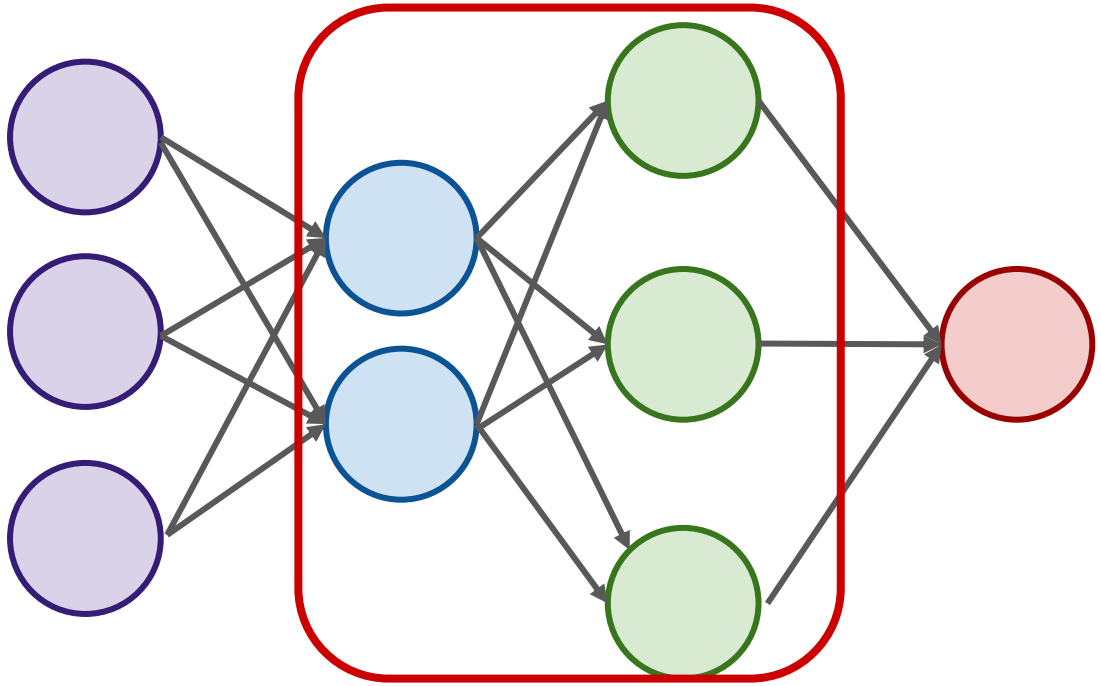
Réseaux de Neurones

- La dernière couche est la **output layer (couche de sortie).**
- Cette dernière couche peut être constituée de plus d'un neurone



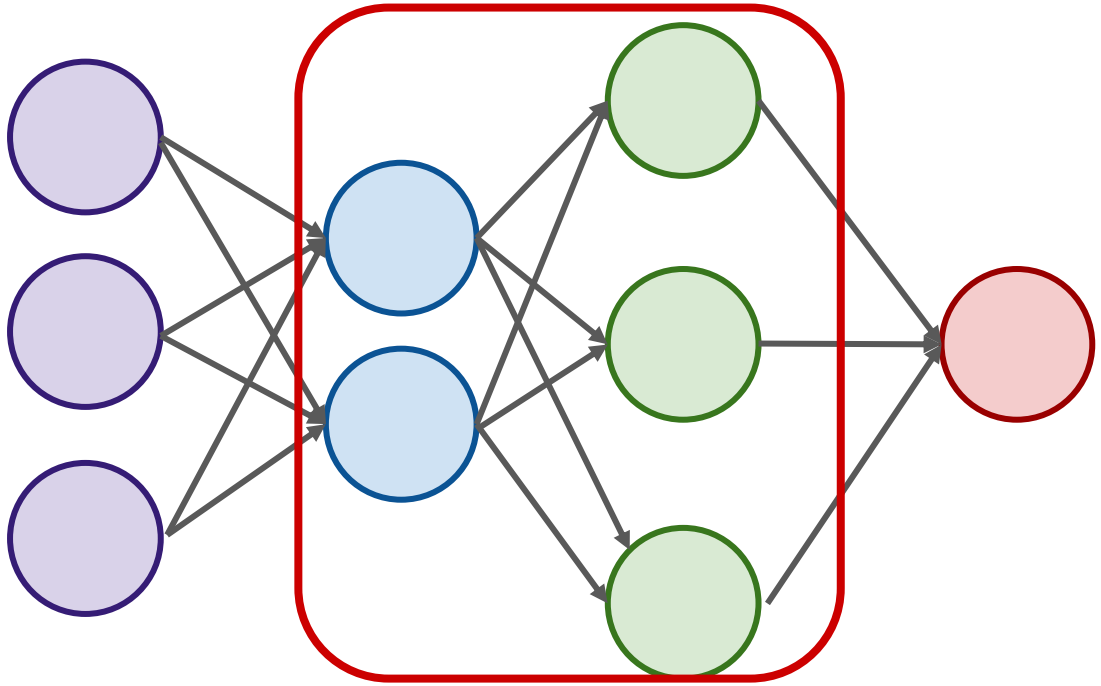
Réseaux de Neurones

- Les couches situées entre les couches d'entrée et de sortie sont les **hidden layers (couches cachées).**



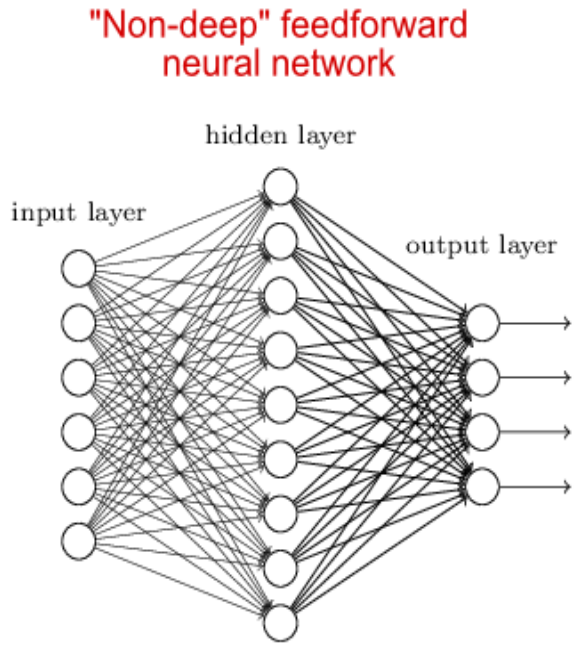
Réseaux de Neurones

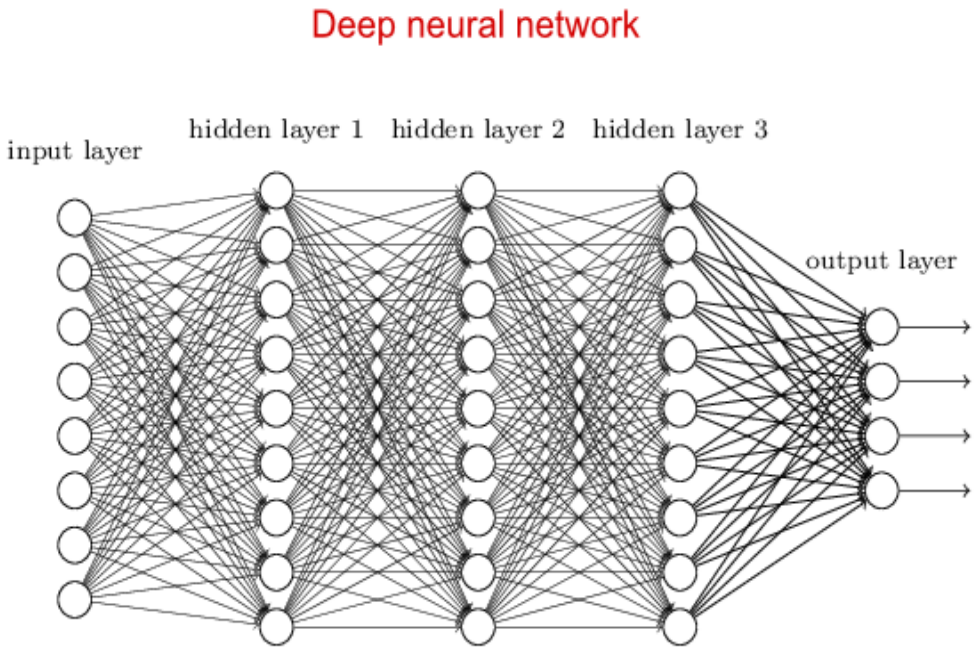
- Les couches cachées sont difficiles à interpréter, en raison de leur forte interconnectivité et de leur éloignement des valeurs d'entrée ou de sortie connues.



Réseaux de Neurones

- Les réseaux de neurones deviennent des **"réseaux neuronaux profonds"** s'ils contiennent au moins 2 couches cachées.





Réseaux de Neurones

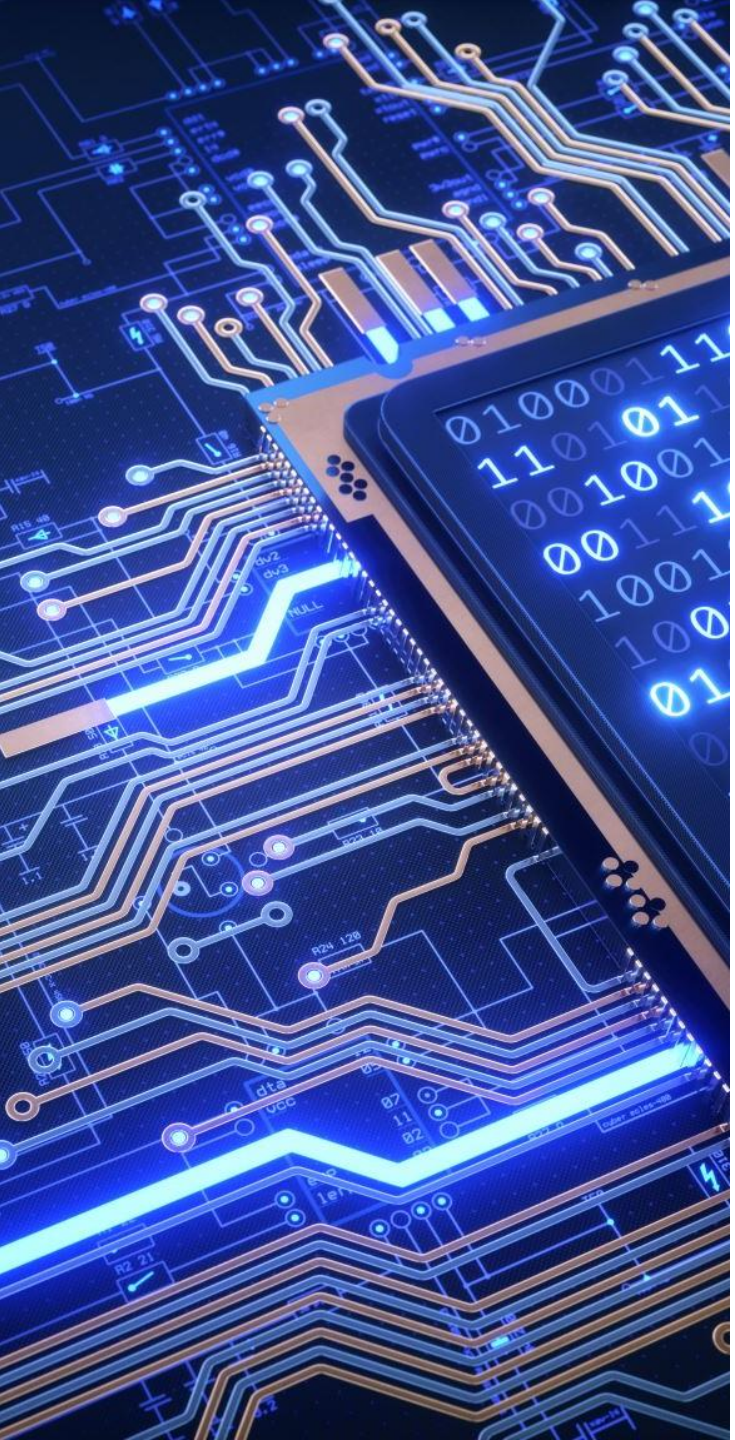
- Zhou Lu et plus tard Boris Hanin ont prouvé mathématiquement que les réseaux de neurones peuvent approximer n'importe quelle fonction continue convexe.

Réseaux de Neurones

- Auparavant, dans notre modèle simple, nous avons vu que le perceptron lui-même contenait une fonction de somme très simple f(x).
- Pour la plupart des cas d'utilisation, cependant, cela ne sera pas utile, nous voudrons être en mesure de fixer des contraintes à nos valeurs de sortie, en particulier dans les tâches de classification.

Réseaux de Neurones

- Dans une tâche de classification, il serait utile que tous les résultats se situent entre 0 et 1.
- Ces valeurs peuvent ensuite présenter des affectations de probabilité pour chaque classe.
- Dans la prochaine session, nous explorerons comment utiliser les **fonctions d'activation** pour fixer des limites aux valeurs de sortie du neurone.



Plan

- Du modèle Perceptron aux réseaux de neurones
- **Fonctions d'Activation**
- Fonctions de Coût d'erreur (Cost)
- Rétropropagation (BackPropagation)
- Normalisation
- Optimisation

Fonctions d'Activation

- Rappelons que les entrées **x** ont un poids **w** et un terme de biais **b** qui leur est attaché dans le modèle de perceptron.
- Ce qui signifie que nous avons
  - **x*w + b**
  - Il est clair que **w** implique le poids ou la force à donner à la valeur d'entrée.
  - On peut considérer **b** comme une valeur de décalage, ce qui fait que **x*w** doit atteindre un certain seuil avant d'avoir un effet.

Fonctions d'Activation

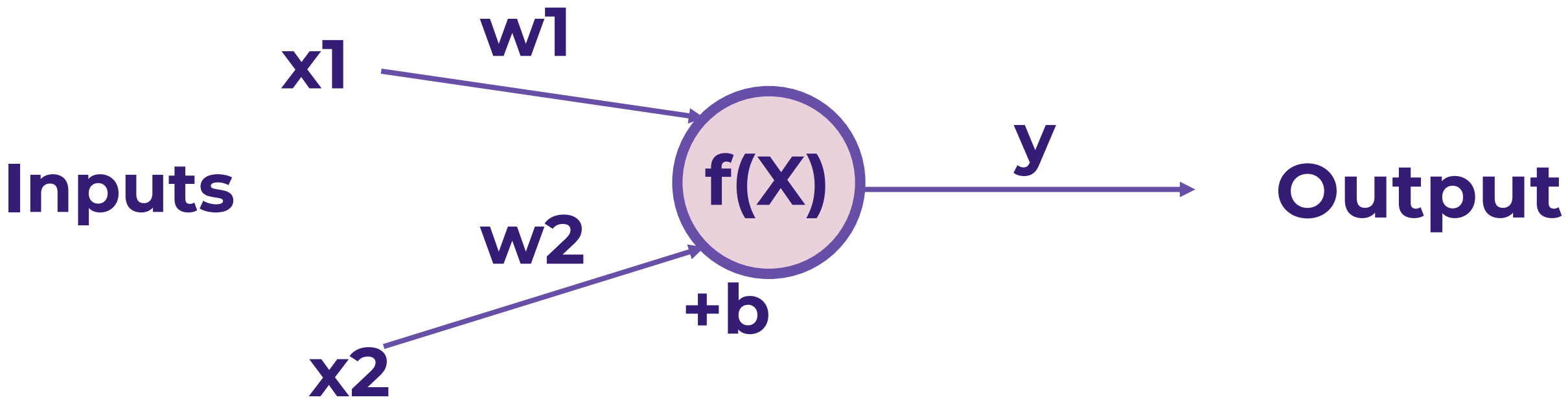
- Par exemple si **b= -10**
  - **x*w + b**
    - Ensuite, les effets de **x*w** ne commenceront vraiment à surmonter le biais que lorsque leur produit dépassera 10.
    - Après cela, l'effet est uniquement basé sur la valeur de **w**.
    - D'où le terme de "biais"

Fonctions d'Activation

- Ensuite, nous voulons fixer des limites pour la valeur globale de sortie de :
  - **x*w + b**
  - Nous pouvons déclarer :
    - **z = x*w + b**
  - Et ensuite faire passer **z** par une fonction d'activation pour limiter sa valeur.

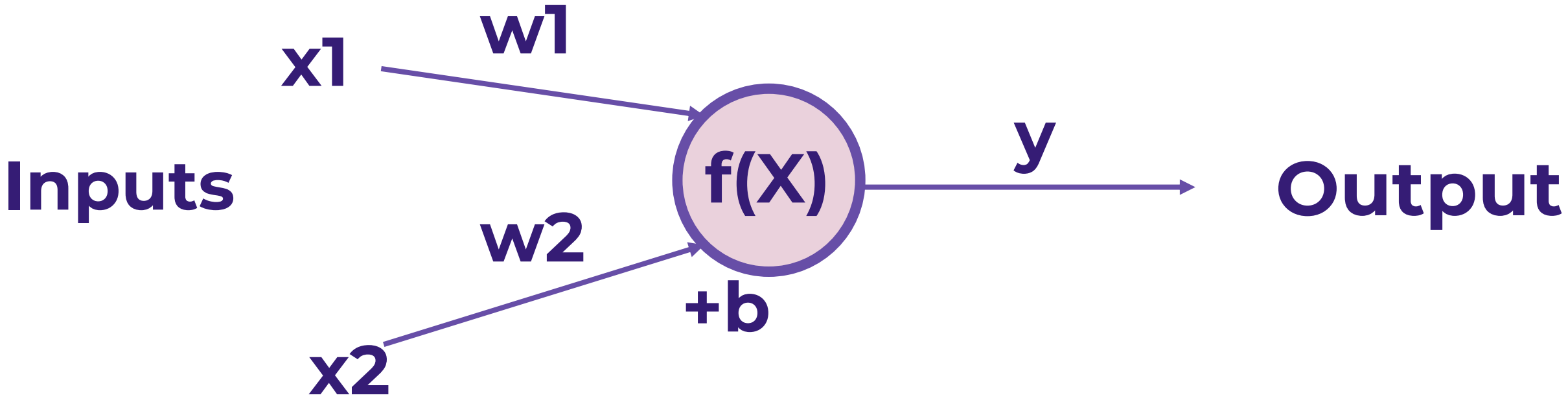
Fonctions d'Activation

- Rappelons que notre simple perceptron a un f(X)



Fonctions d'Activation

- Si nous avions un problème de classification binaire, nous voudrions une sortie de 0 ou de 1.

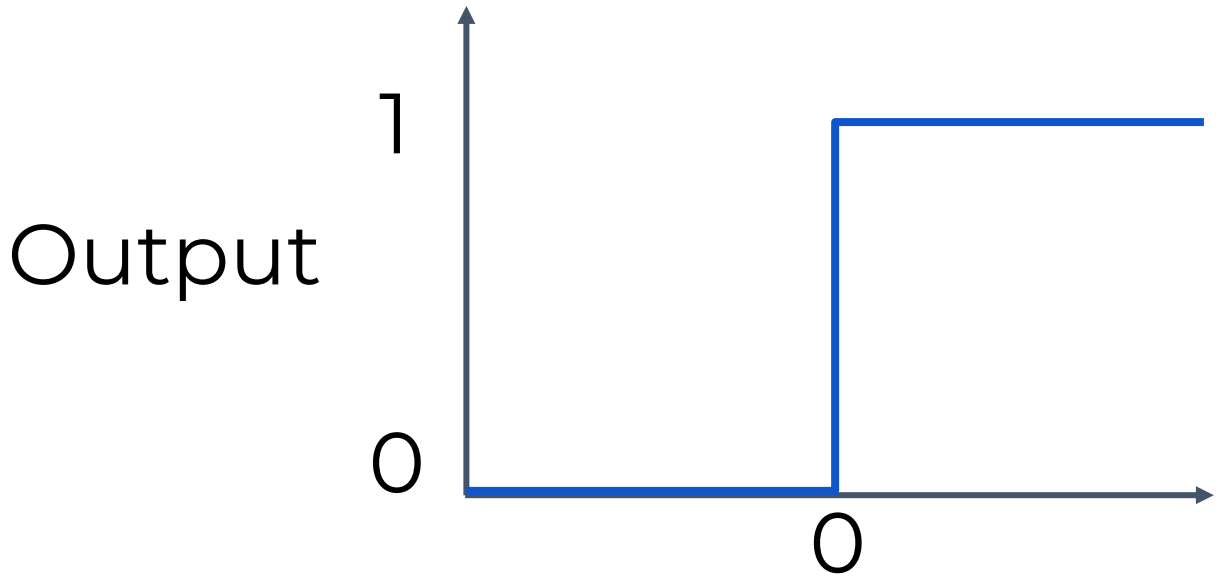


Fonctions d'Activation

- Pour éviter toute confusion, définissons les entrées totales comme une variable **z**.
- Où **z = wx + b**
- Dans ce contexte, nous appellerons les fonctions d'activation **f(z)**.
- N'oubliez pas que vous verrez souvent ces variables en majuscules **f(Z)** ou **X** pour indiquer une entrée de tenseur composée de valeurs multiples.

Fonctions d'Activation

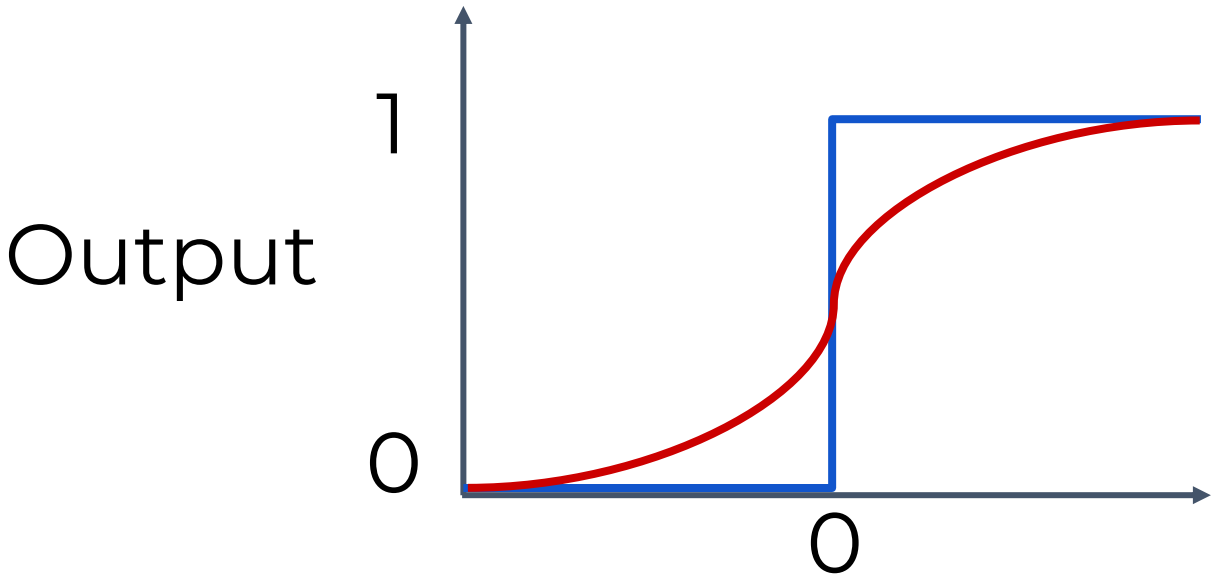
- Les réseaux les plus simples s'appuient sur une **fonction échelon** qui produit 0 ou 1.



$$z = wx + b$$

Fonctions d'Activation

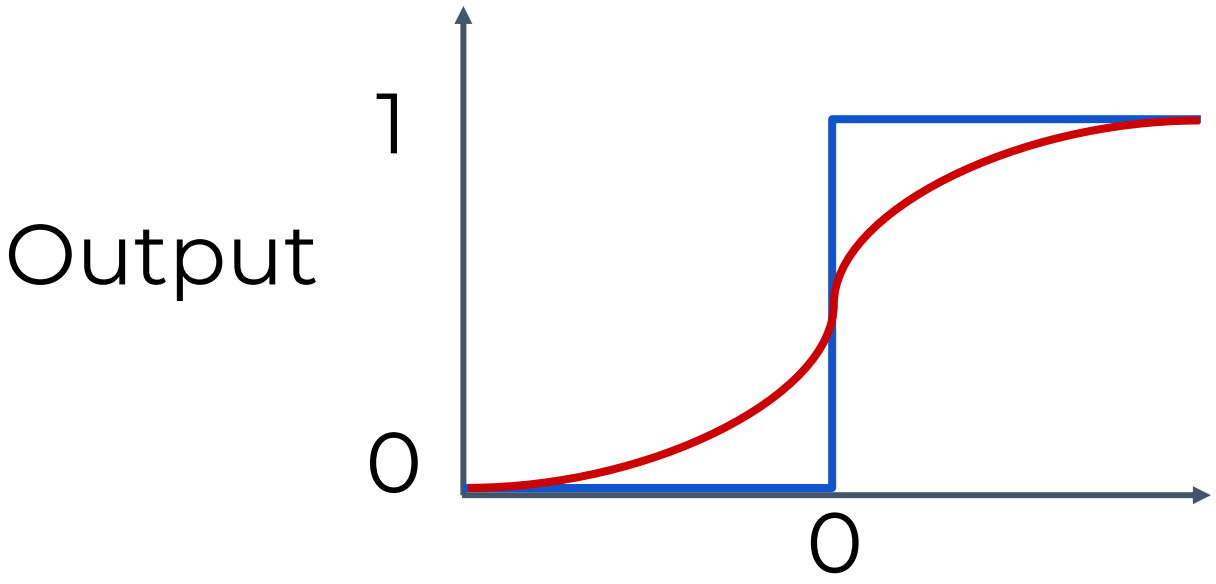
- Ce serait bien si nous pouvions avoir une fonction plus dynamique, par exemple la ligne rouge !



$$z = wx + b$$

Fonctions d'Activation

- Heureusement pour nous, c'est la fonction sigmoïde !

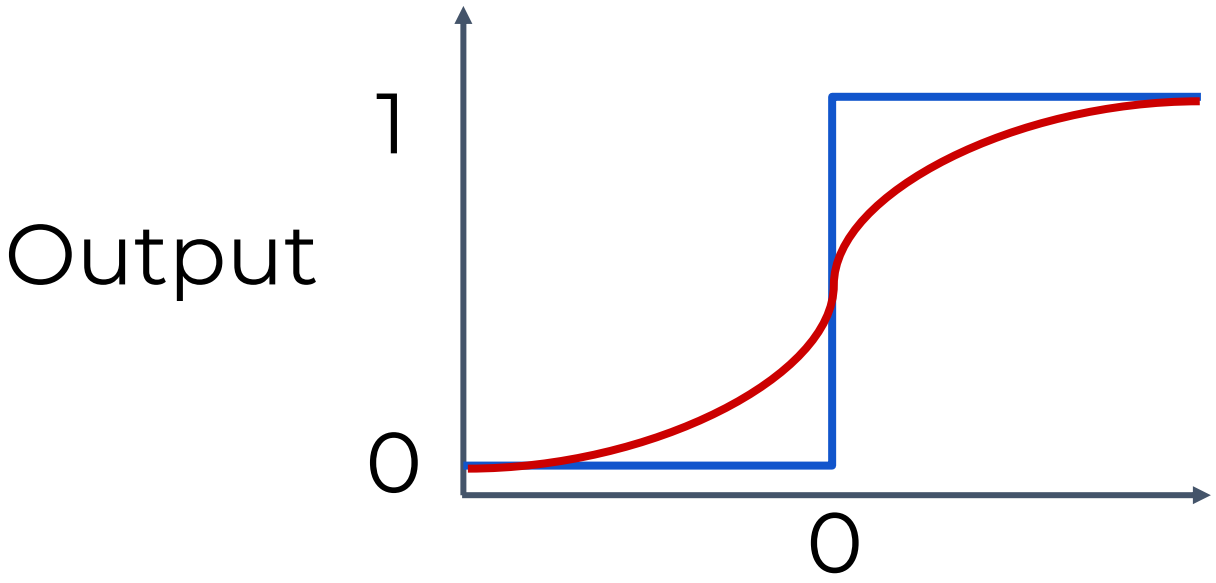


$$f(z) = \frac{1}{1 + e^{(-z)}}$$

z = wx + b

Fonctions d'Activation

- La modification de la fonction d'activation utilisée peut être bénéfique selon la tâche à accomplir !

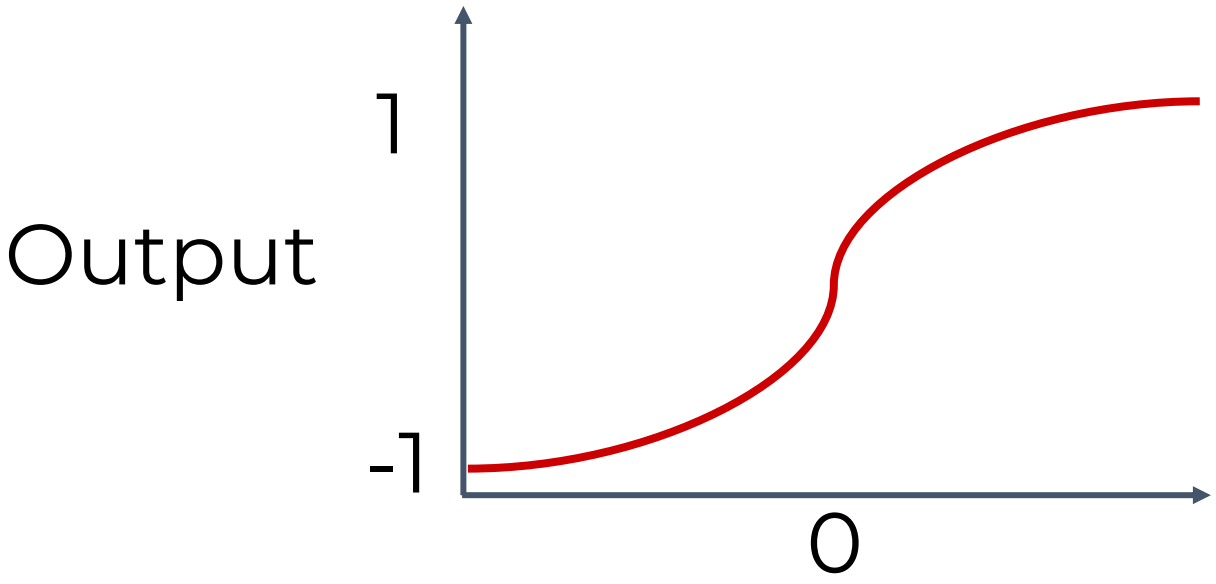


$$f(z) = \frac{1}{1 + e^{(-z)}}$$

z = wx + b

Fonctions d'Activation

- Tangente hyperbolique : tanh(z)
- Résultats de sortie entre -1 et 1 au lieu de 0 à 1



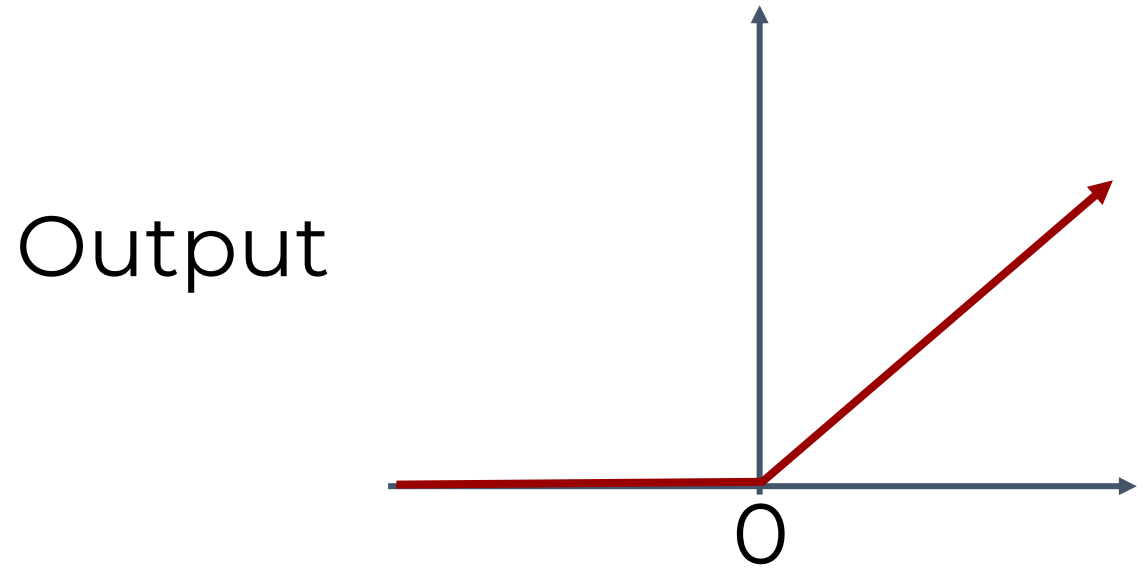
$$\cosh x = \frac{\mathbf{e}^{x} + \mathbf{e}^{-x}}{2}$$

$$\sinh x = \frac{\mathbf{e}^{x} - \mathbf{e}^{-x}}{2}$$

$$\tanh x = \frac{\sinh x}{\cosh x}$$

Fonctions d'Activation

- Unité linéaire rectifiée (ReLU) : Il s'agit en fait d'une fonction relativement simple : max(0,z)



Fonctions d'Activation

- On a constaté que ReLu avait de très bonnes performances, notamment en ce qui concerne la question de **vanishing gradient**.
- Nous choisissons souvent ReLu par défaut en raison de ses bonnes performances générales.

Fonctions d'Activation

- Pour une liste complète des fonctions d'activation possibles, consultez :
  - **en.wikipedia.org/wiki/Activation_function**

Fonctions d'Activation Multi-Classe

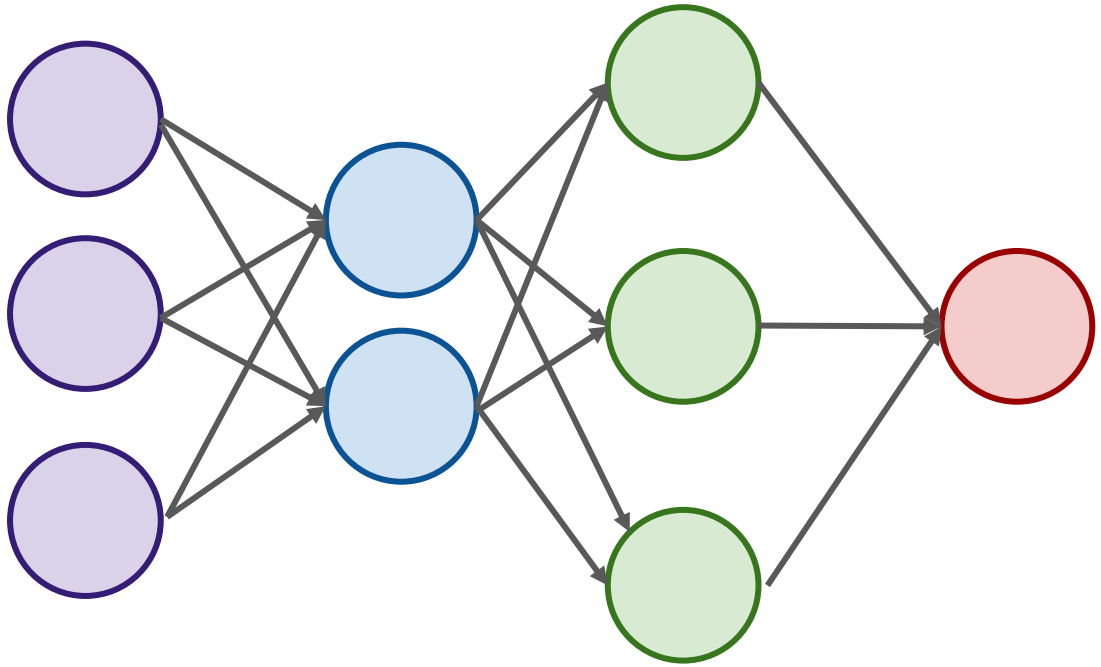
- Remarquez que toutes ces fonctions d'activation ont un sens pour une seule sortie, que ce soit une étiquette (label) continue ou une tentative de prédire une classification binaire (soit un 0 ou un 1).
- Mais que devons-nous faire si nous avons une situation multiclasse ?

Fonctions d'Activation Multi-Classe

- Il existe deux grands types de situations multiclasses
  - Classes non-exclusives
    - Un point de données peut se voir attribuer plusieurs classes/catégories
  - Classes mutuellement exclusives
    - Une seule classe par point de données.

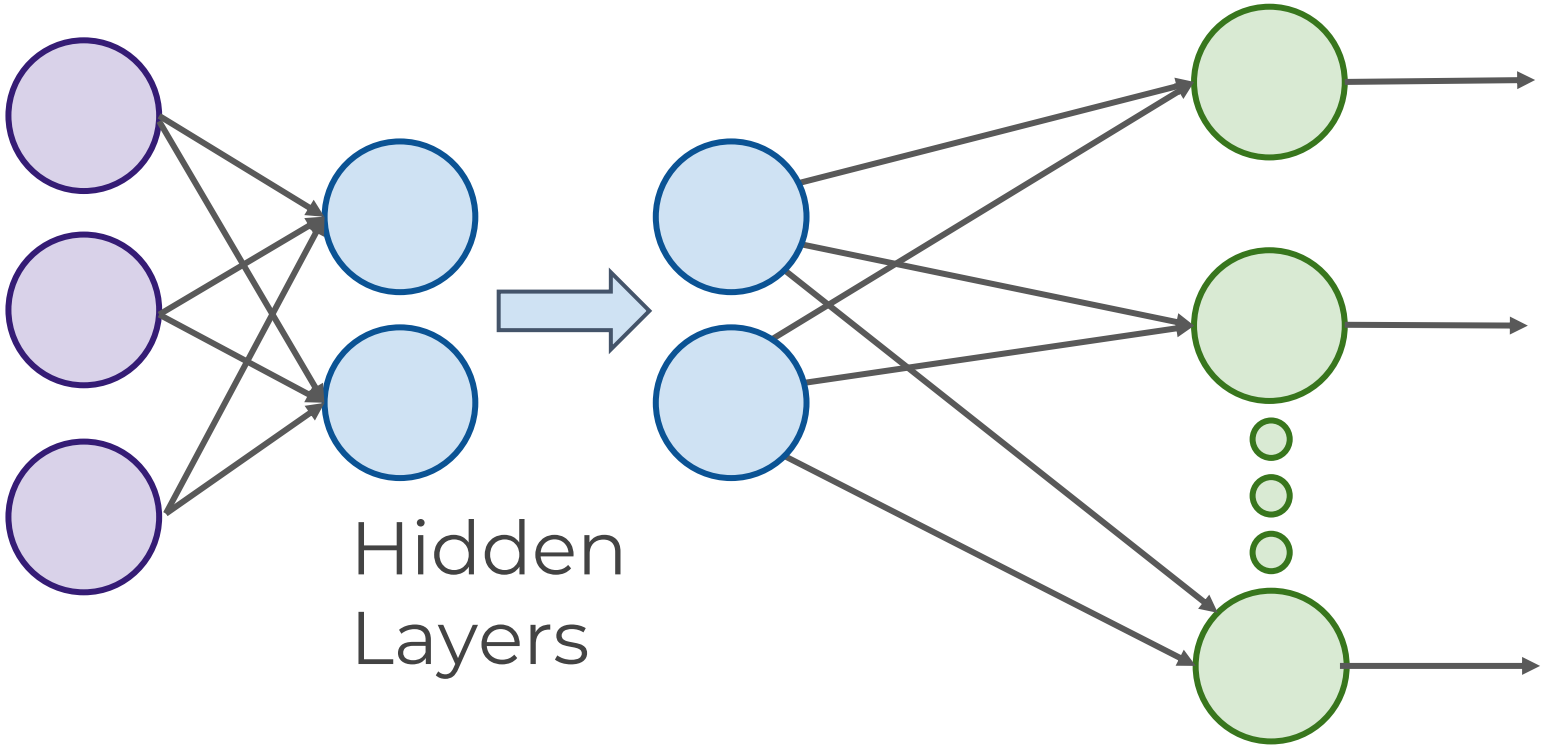
Réseaux de neurones

- Auparavant, nous considérions la dernière couche de sortie comme un seul nœud.
- Ce nœud unique pourrait produire une valeur de régression continue ou une classification binaire (0 ou 1).



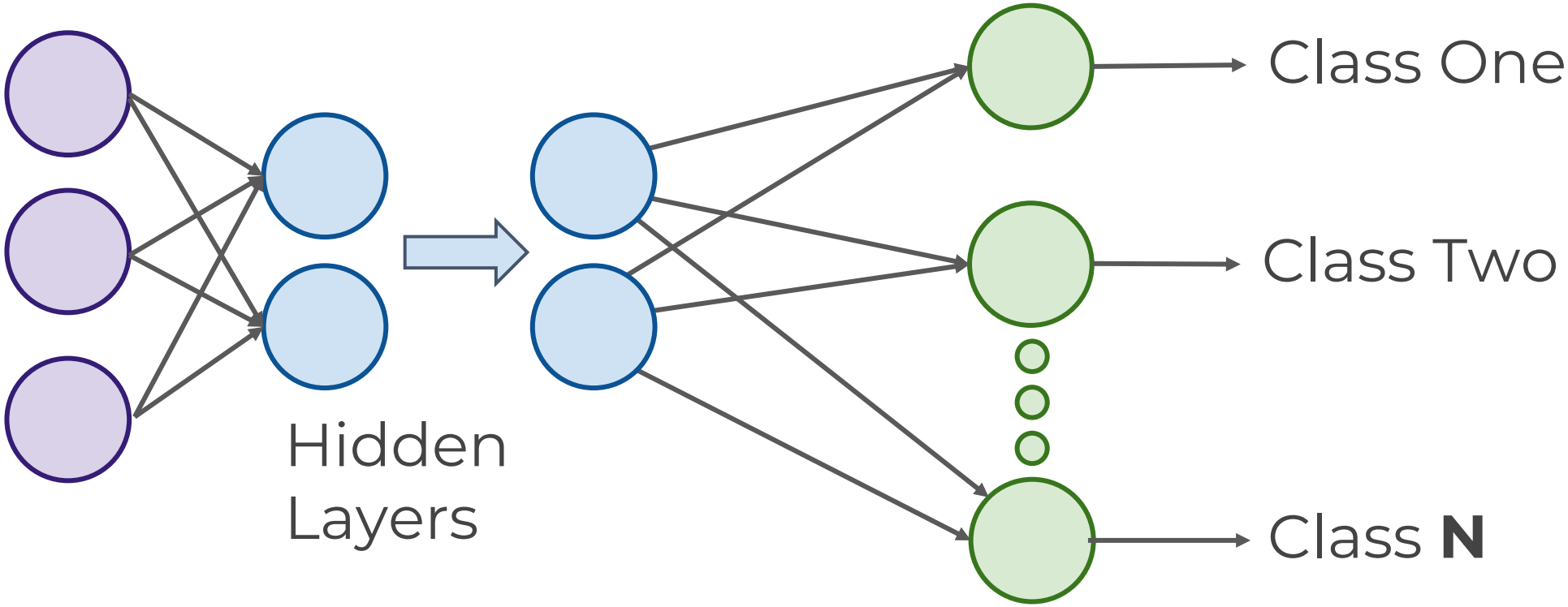
Classification Multiclasse

- Organisation pour plusieurs classes



Classification Multiclasse

- Organisation pour plusieurs classes



Deep Learning

- Organisation des classes multiples
  - Cela signifie que nous devrons organiser des catégories pour cette couche de sortie.
  - On ne peut pas avoir des catégories comme "rouge", "bleu", "vert", etc...

Deep Learning

- Organisation des classes multiples
  - Nous utilisons à la place le **one-hot encoding**
  - Voyons à quoi cela ressemble pour les classes mutuellement exclusives.

Deep Learning

- Classes mutuellement exclusives

Data Point 1	**RED**
Data Point 2	**GREEN**
Data Point 3	**BLUE**
...	**...**
Data Point N	**RED**

Deep Learning

- Classes mutuellement exclusives

Data Point 1	**RED**
Data Point 2	**GREEN**
Data Point 3	**BLUE**
...	**...**
Data Point N	**RED**

	RED	GREEN	BLUE
Data Point 1	**1**	**0**	**0**
Data Point 2	**0**	**1**	**0**
Data Point 3	**0**	**0**	**1**
...	**...**	**...**	**...**
Data Point N	**1**	**0**	**0**

Deep Learning

- Classes non-exclusives

Data Point 1	**A,B**
Data Point 2	**A**
Data Point 3	**C,B**
...	**...**
Data Point N	**B**

	A	B	C
Data Point 1	**1**	**1**	**0**
Data Point 2	**1**	**0**	**0**
Data Point 3	**0**	**1**	**1**
...	**...**	**...**	**...**
Data Point N	**0**	**1**	**0**

Deep Learning

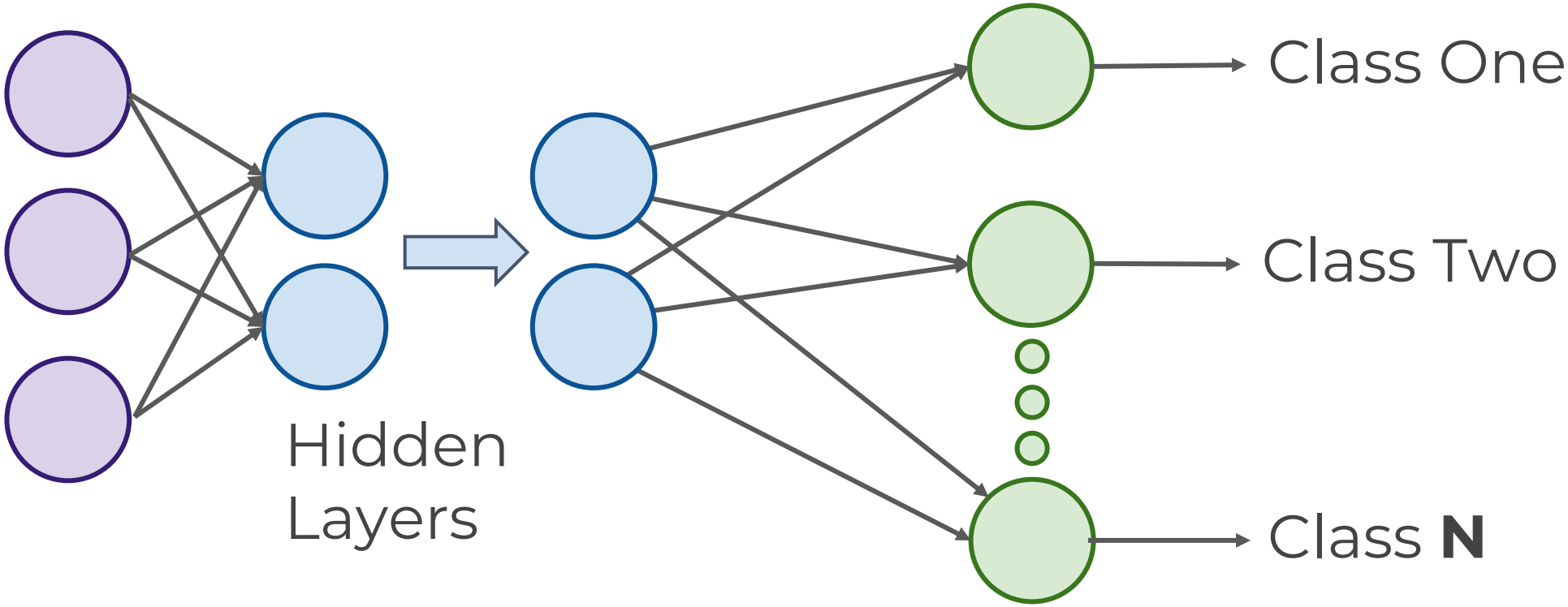
- Maintenant que nos données sont correctement organisées, il nous suffit de choisir la bonne fonction d'activation de classification que la dernière couche de sortie devrait avoir.

Deep Learning

- Non-exclusive
  - Fonction sigmoïde
    - Chaque neurone produira une valeur de sortie entre 0 et 1, indiquant la probabilité de se voir attribuer cette classe.

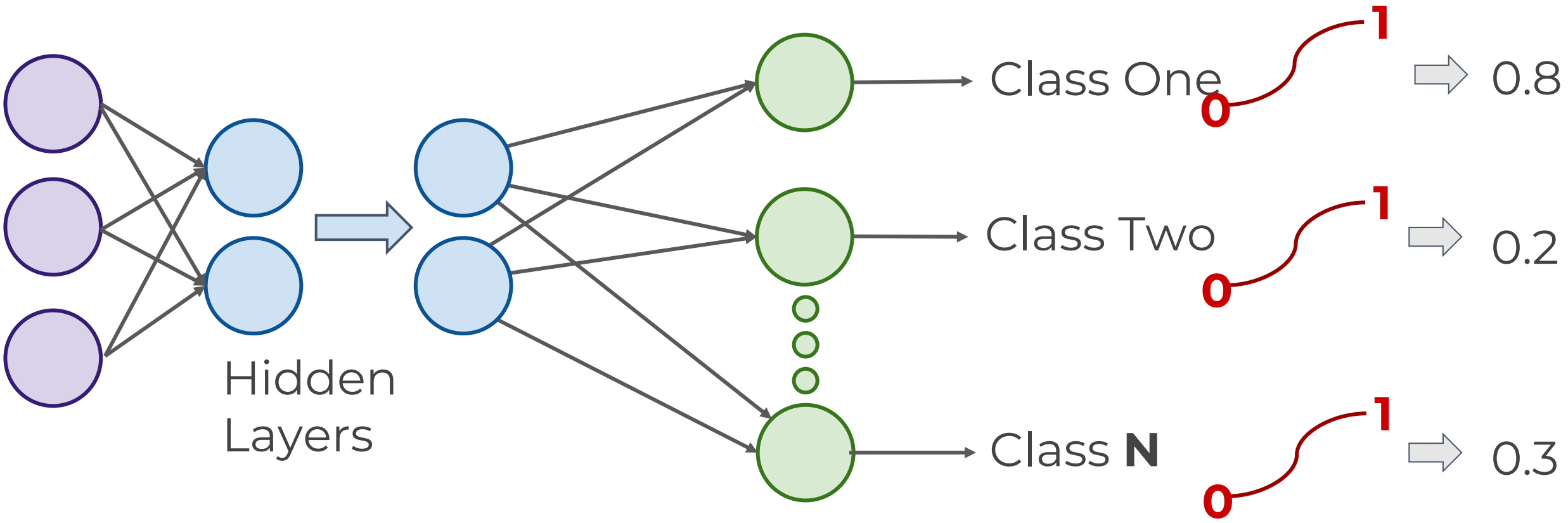
Classification Multiclasse

- Fonction Sigmoïde pour les classes non exclusives



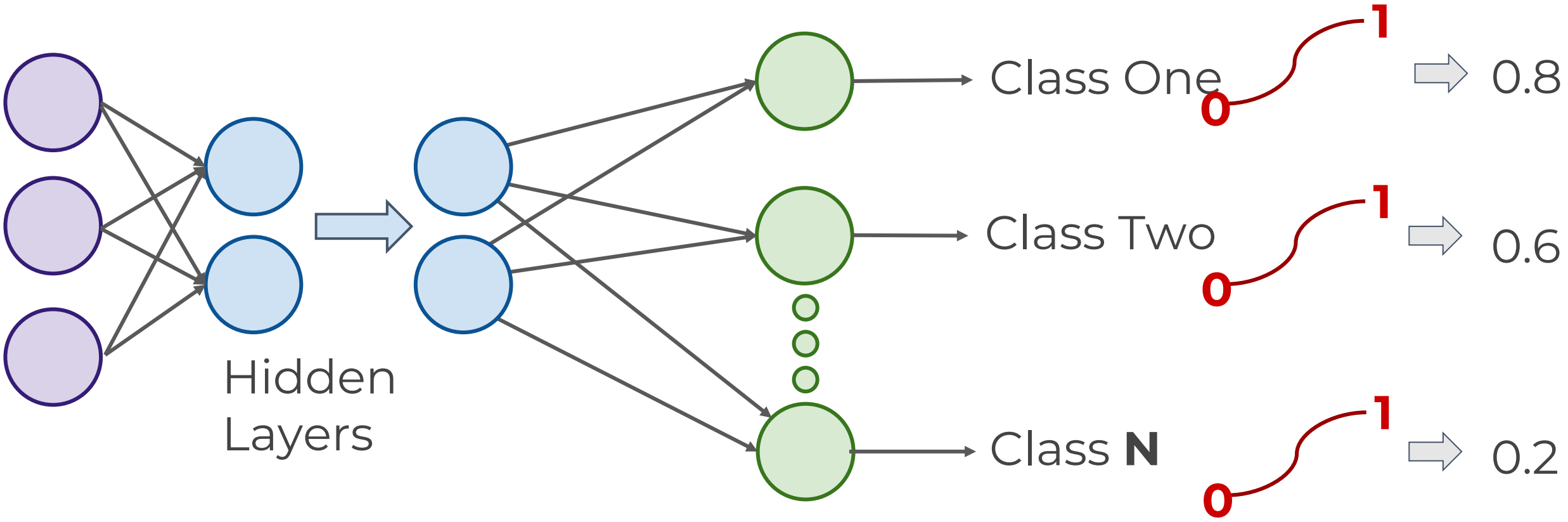
Classification Multiclasse

- Fonction Sigmoïde pour les classes non exclusives



Classification Multiclasse

- Fonction Sigmoïde pour les classes non exclusives



Deep Learning

- Non-exclusive
  - Fonction sigmoïde
    - Gardez à l'esprit que cela permet à chaque neurone de produire une sortie indépendamment des autres classes, ce qui permet à un seul point de données alimenté dans la fonction de se voir attribuer plusieurs classes.

Deep Learning

- Classes mutuellement exclusives
  - Mais que faire lorsque chaque point de données ne peut se voir attribuer qu'une seule classe ?
  - Nous pouvons utiliser la **fonction softmax** pour cela !

Deep Learning

- Fonction Softmax

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K$$

Deep Learning

- Classes mutuellement exclusives
  - La fonction Softmax calcule la distribution des probabilités de l'événement sur **K** événements différents.
  - Cette fonction calcule les probabilités de chaque classe de cible sur toutes les classes de cibles possibles.

Deep Learning

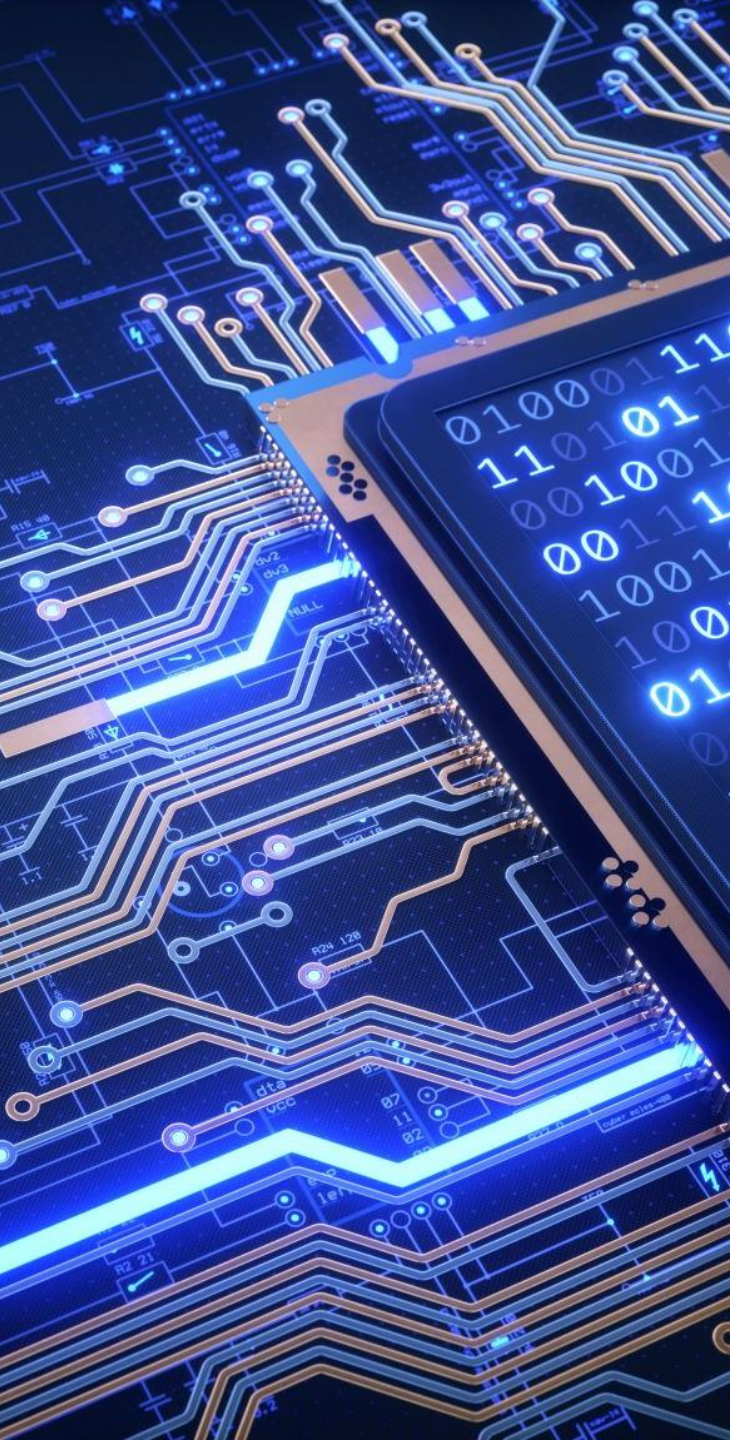
- Classes mutuellement exclusives
  - L'intervalle sera de 0 à 1, et **la somme de toutes les probabilités sera égale à un**.
  - Le modèle retourne les probabilités de chaque classe et la classe cible choisie aura la probabilité la plus élevée.

Deep Learning

- Classes mutuellement exclusives
  - La chose principale à garder à l'esprit est que si vous utilisez softmax pour des problèmes multi-classes, vous obtenez ce genre de résultat :
    - [Red , Green , Blue]
    - [0.1 , 0.6 , 0.3]

Deep Learning

- Classes mutuellement exclusives
  - Les probabilités pour chaque classe s'élèvent toutes à 1. Nous choisissons la probabilité la plus élevée pour notre mission.
    - [Red , Green , Blue]
    - [0.1 , 0.6 , 0.3]



Plan

- Du modèle Perceptron aux réseaux de neurones
- Fonctions d'Activation
- **Fonctions de Coût d'erreur (Cost)**
- Rétropropagation (BackPropagation)
- Normalisation
- Optimisation

Fonctions de Coût

- Nous comprenons maintenant que les réseaux de neurones absorbent des données, les multiplient par des poids et leur ajoutent des biais.
- Ce résultat passe ensuite par une fonction d'activation qui, à la fin de toutes les couches, mène à une sortie.

Fonctions de Coût

- Cette sortie $\hat{\mathbf{y}}$ est l'estimation du modèle de ce qu'il prédit pour l'étiquette (label).
- Donc, une fois que le réseau a créé sa prédiction, comment l'évaluer ?
- Et après l'évaluation, comment pouvons-nous actualiser les poids et les biais du réseau ?

Fonctions de Coût

- Nous devons prendre les résultats estimés du réseau et les comparer ensuite aux valeurs réelles du label.
- Gardez à l'esprit qu'il s'agit d'utiliser l'ensemble des données d’entraînement pendant l'ajustement/entraînement du modèle.

Fonctions de Coût

- La fonction de coût, souvent appelée fonction de perte (loss) doit être une moyenne afin de pouvoir produire une valeur unique.
- Nous pouvons suivre nos pertes/coûts pendant l'entraînement pour contrôler les performances du réseau.

Fonctions de Coût

- Nous utiliserons les variables suivantes :
  - **y** pour représenter la valeur réelle
  - **a** pour représenter la prédiction du neurone
  - En termes de poids et de biais :
    - $w*x + b = z$
    - Passez z en fonction d'activation $\sigma(z) = a$

Fonctions de Coût

- Une fonction de coût très courante est la fonction de coût quadratique :

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

Fonctions de Coût

- Remarquez comme le carré de tout ça fait 2 choses utiles pour nous, garde tout positif et **punit** les grosses erreurs !

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

Fonctions de Coût

- On peut considérer la fonction de coût comme :

$$C(W, B, S^r, E^r)$$

Fonctions de Coût

- **W** est le poids de notre réseau de neurones, **B** est le biais de notre réseau de neurones, $\mathbf{S^r}$ est l'apport d'un seul échantillon d'entraînement en entrée, et $\mathbf{E^r}$ est le résultat souhaité en sortie de cet échantillon d'entraînement.

$$C(W, B, S^r, E^r)$$

Fonctions de Coût

- Remarquez comment toutes ces informations ont été encodées dans notre notation simplifiée.
- Le **a(x)** contient des informations sur les poids et les biais.

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

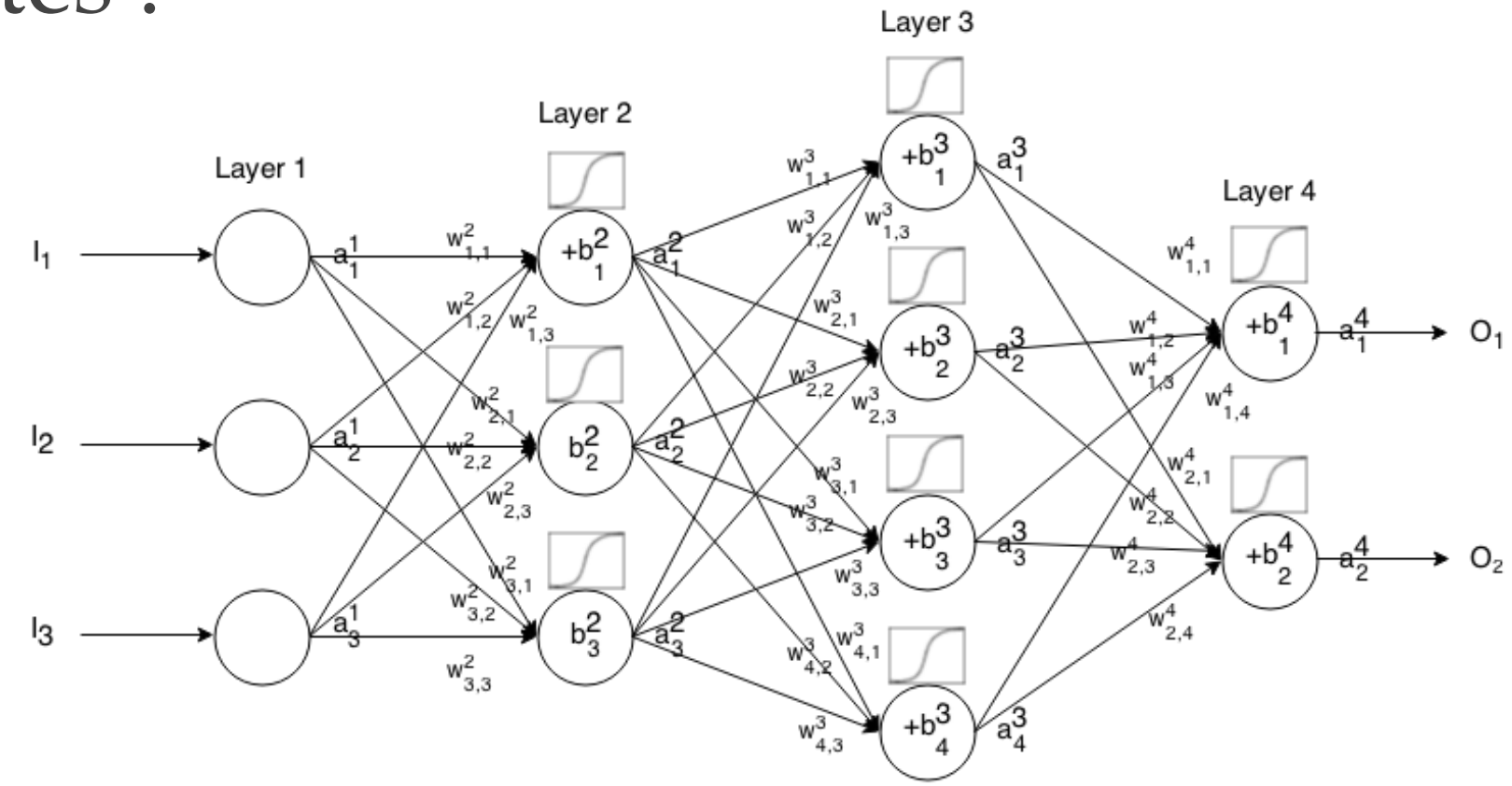
Fonctions de Coût

- Cela signifie que si nous disposons d'un vaste réseau, nous pouvons nous attendre à ce que **C** soit assez complexe, avec d'énormes vecteurs de poids et de biais.

$$C(W, B, S^r, E^r)$$

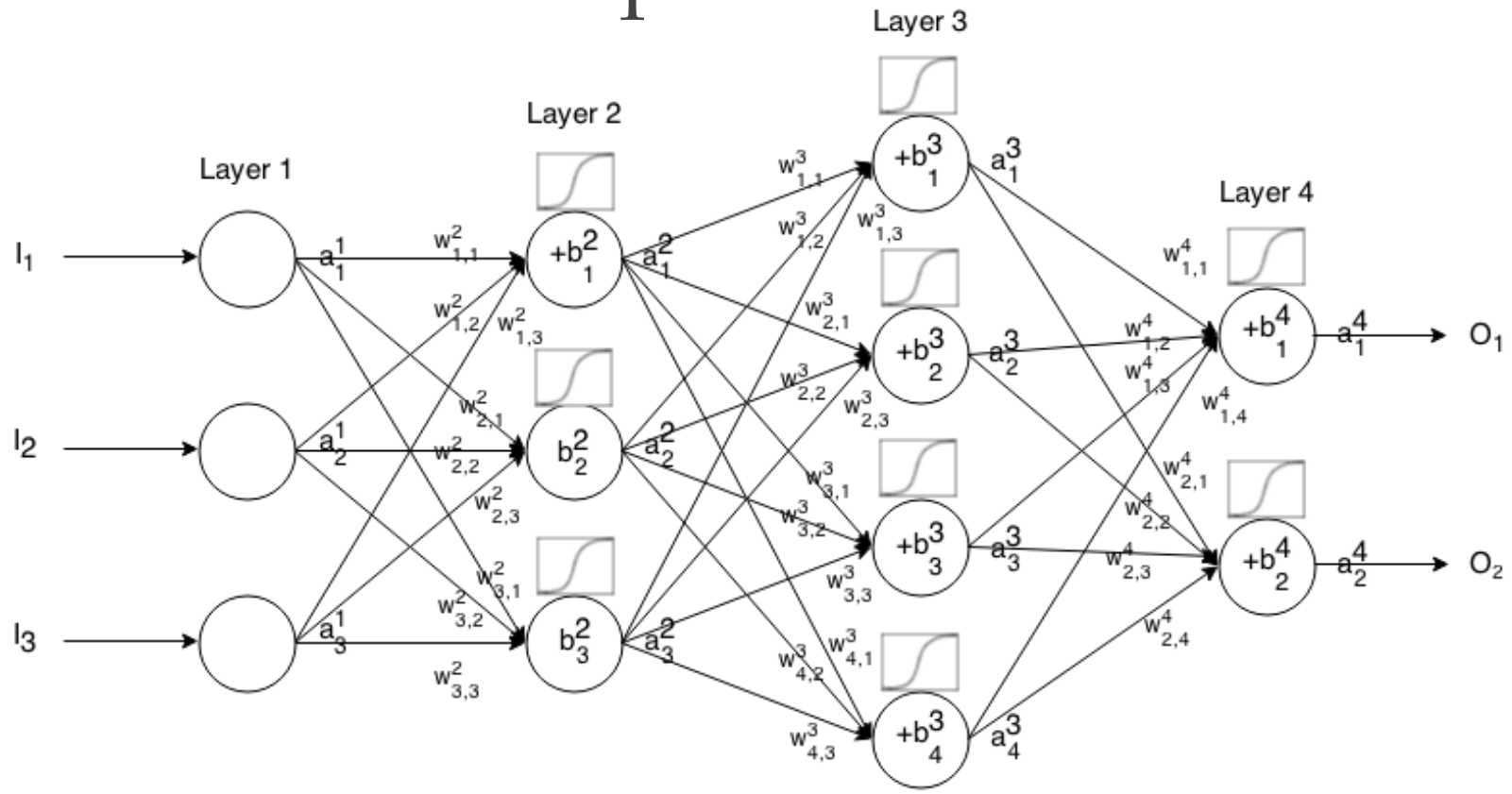
Fonctions de Coût

- Voici un petit réseau avec tous ses paramètres étiquetés :



Fonctions de Coût

- Ça fait beaucoup à calculer !
- Comment résoudre ce problème ?



Fonctions de Coût

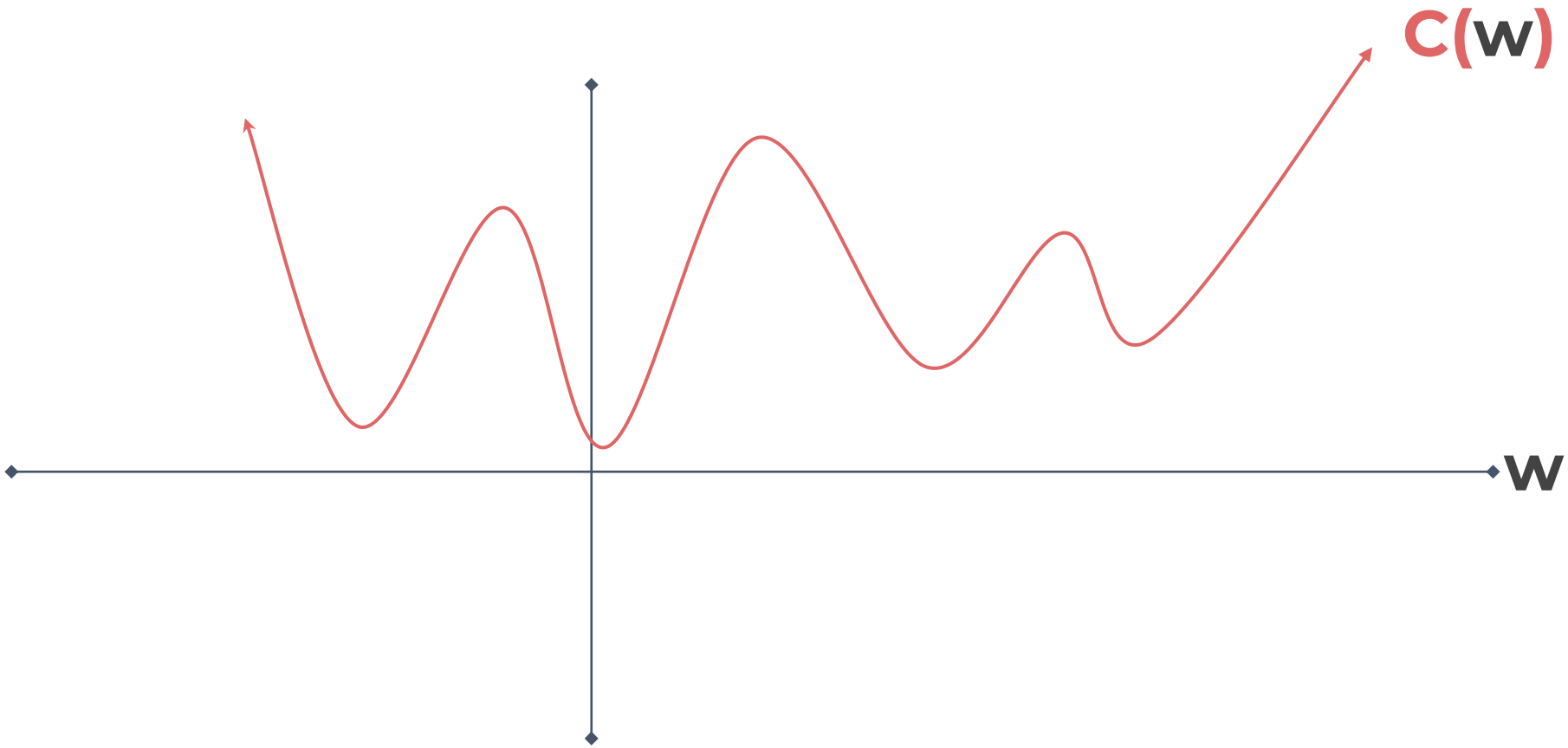
- Dans un cas réel, cela signifie que nous avons une fonction de coût **C** dépendant de nombreux poids !
    - **C(w1,w2,w3,....wn)**
    - Comment déterminer quels poids nous conduisent au coût le plus bas ?

Fonctions de Coût

- Pour simplifier, imaginons que nous n'ayons qu'un seul poids dans notre fonction de coût **w**.
- Nous voulons **minimiser** notre perte / coût (erreur globale).
- Ce qui signifie que nous devons déterminer quelle valeur de **w** donne le minimum de **C(w)**

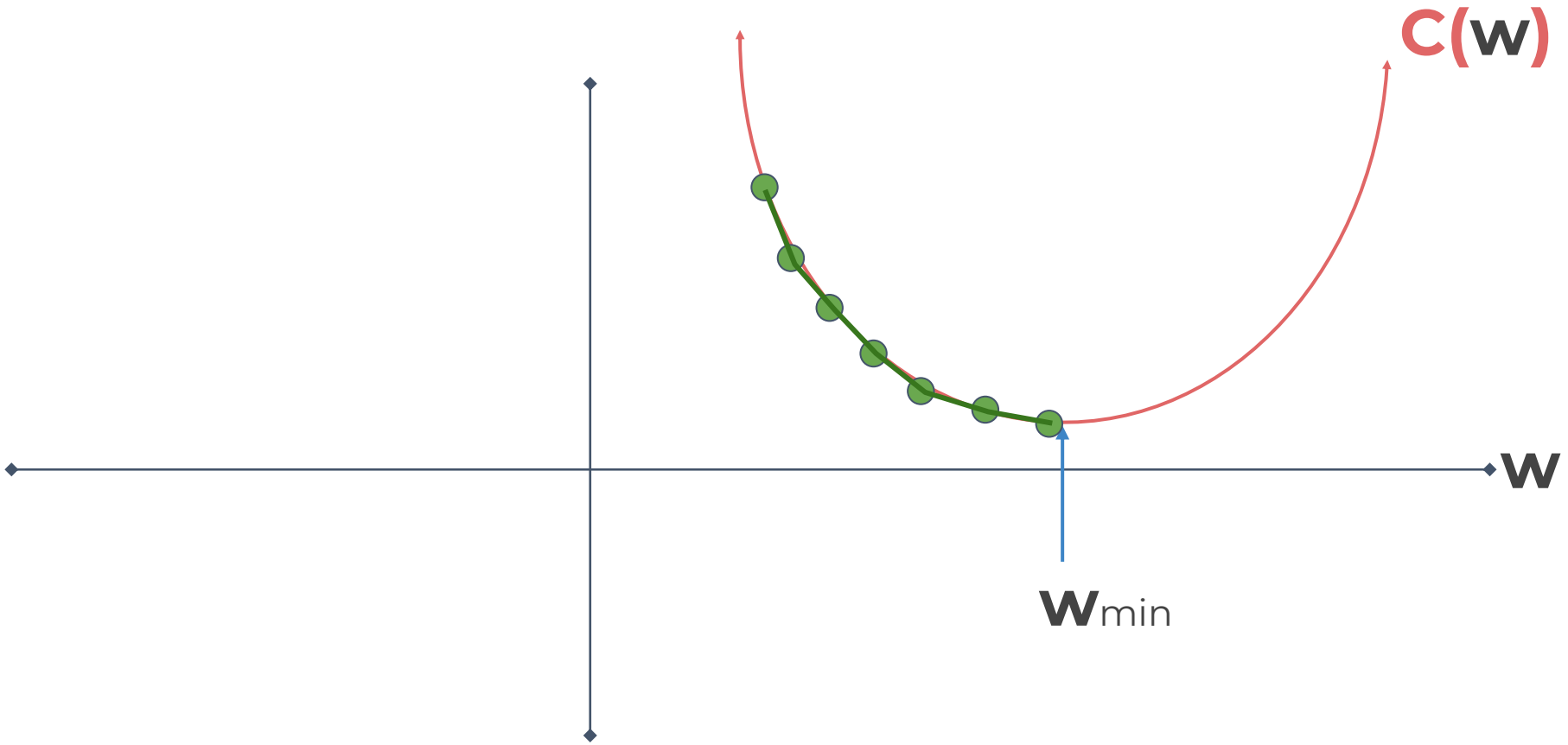
Fonctions de Coût

- Nous pouvons utiliser **la descente de gradient (gradient descent)** pour résoudre ce problème.



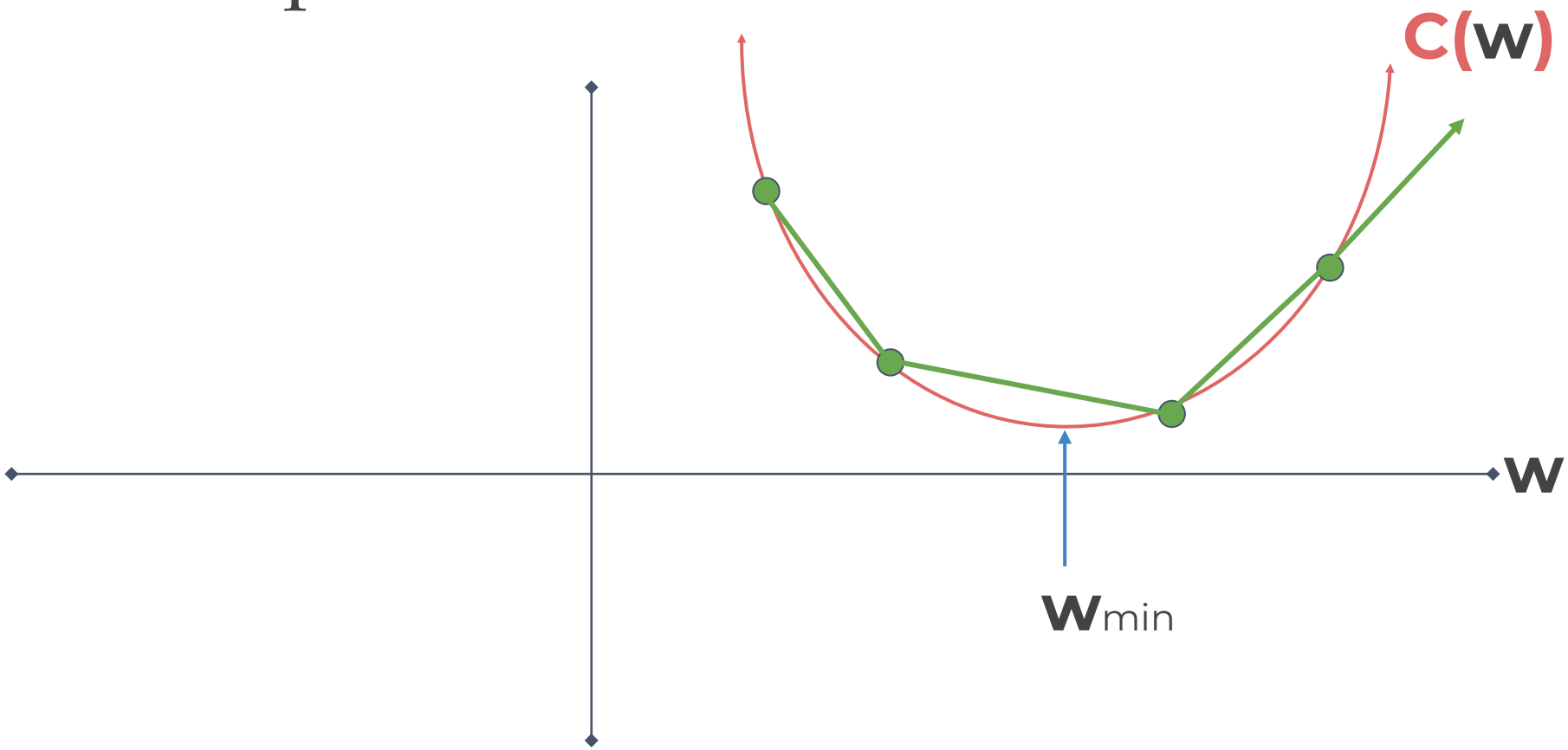
Fonctions de Coût

- Les petits pas prennent plus de temps pour trouver le minimum.



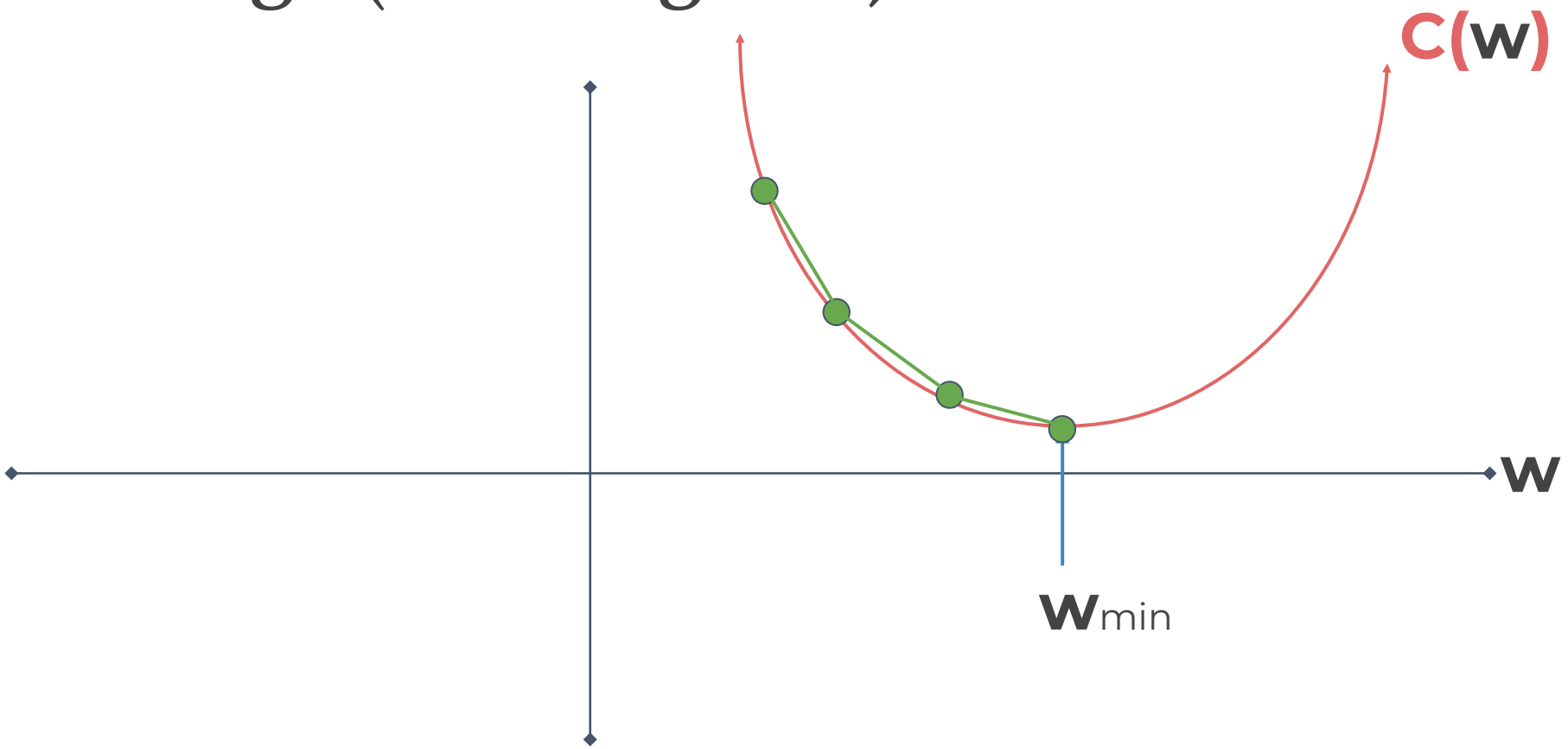
Fonctions de Coût

- Les plus grands pas sont plus rapides, mais nous risquons de dépasser le minimum !



Fonctions de Coût

- Cette taille de pas est connue sous le nom de **taux d'apprentissage (learning rate)**.



Fonctions de Coût

- Le taux d'apprentissage que nous avons montré dans nos illustrations était constant (chaque taille de pas était égale)
- Mais nous pouvons être plus malins et adapter notre taille de pas au fur et à mesure.

Fonctions de Coût

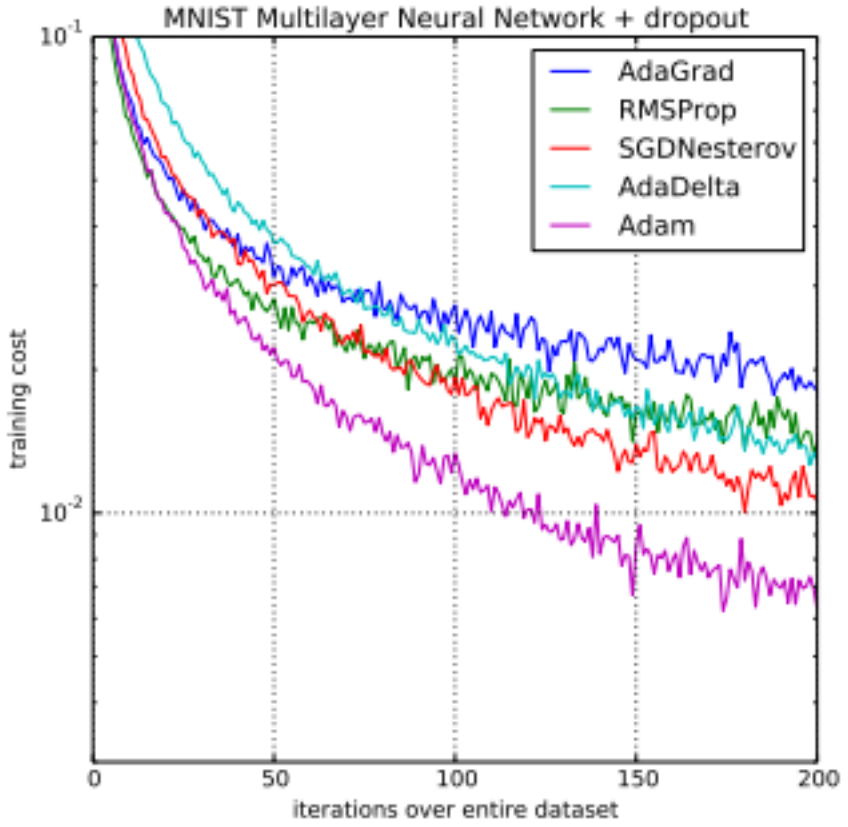
- Nous pourrions commencer par de plus grands pas, puis aller plus petit à mesure que nous nous rendons compte que la pente se rapproche de zéro.
- C'est ce qu'on appelle **la descente adaptative de gradient (adaptive gradient descent)**.

Fonctions de Coût

- En 2015, Kingma et Ba ont publié leur document : "Adam : une méthode d'optimisation stochastique".
- Adam est un moyen beaucoup plus efficace de rechercher ces minimums,

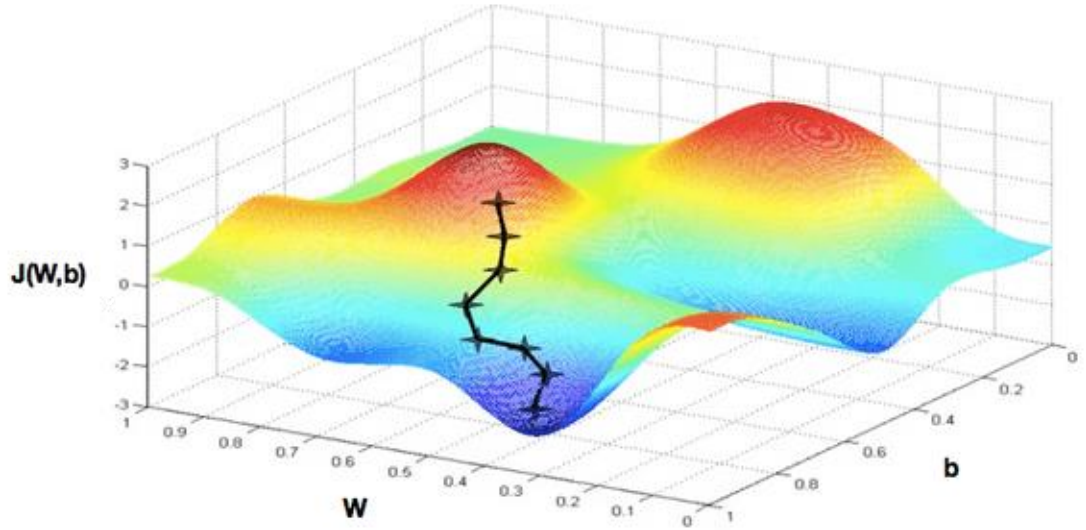
Fonctions de Coût

- Adam vs. d'autres algorithmes de descente de gradient :



Fonctions de Coût

- De manière réaliste, nous calculons cette descente dans un espace n-dimensionnel pour tous nos poids.



Fonctions de Coût

- Lorsqu'on traite ces vecteurs N-dimensionnels (tenseurs), la notation passe de la **dérivée** au **gradient**.
- Cela signifie que nous calculons **∇C(w1,w2,...wn)**

Fonctions de Coût

- Pour les problèmes de classification, nous utilisons souvent la fonction de perte d'**entropie croisée**.
- L'hypothèse est que votre modèle prédit une distribution de probabilité p(y=i) pour chaque classe i=1,2,…,C.

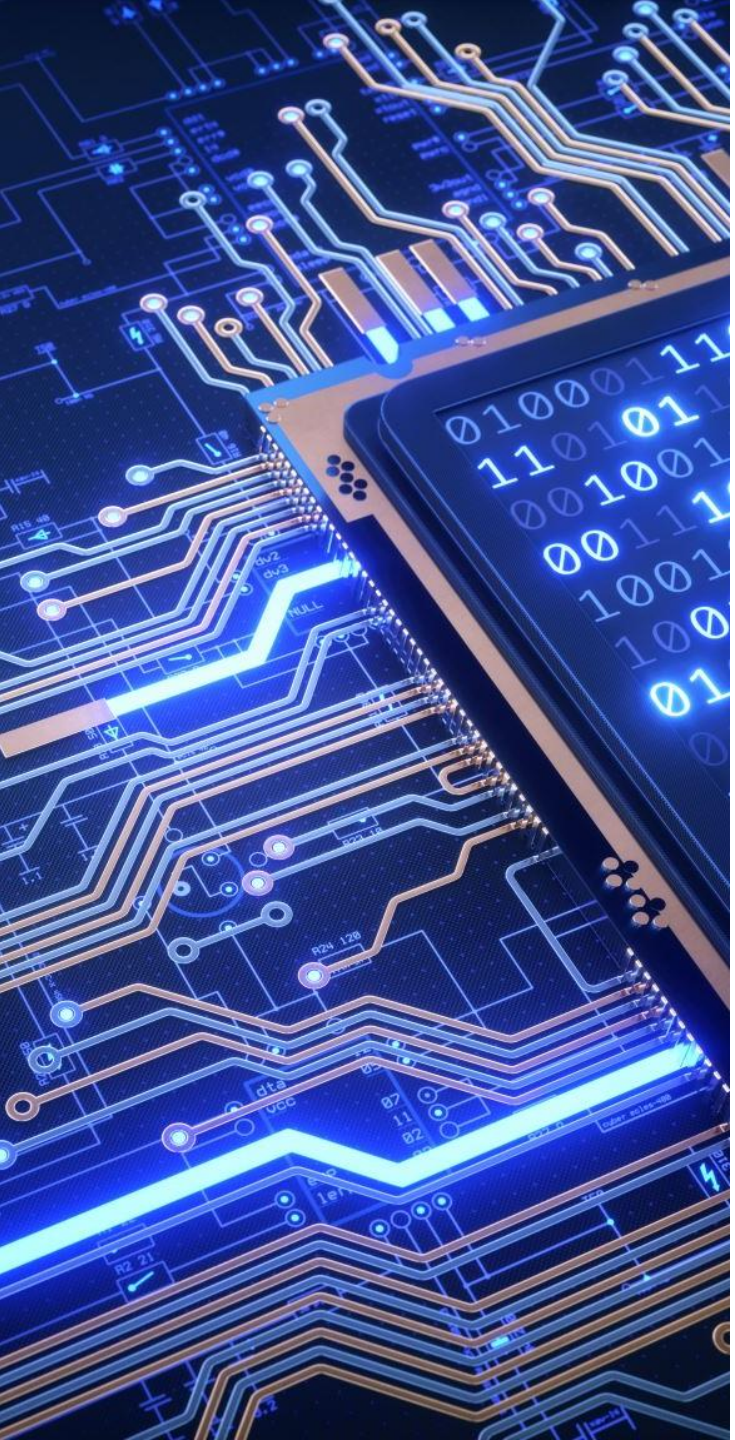
Fonctions de Coût

- Pour une classification binaire, il en résulte :

$$-(y\log(p)+(1-y)\log(1-p))$$

- Pour **M** nombre de classes > 2

$$-\sum_{c=1}^{M} y_{o,c}\log(p_{o,c})$$



Plan

- Du modèle Perceptron aux réseaux de neurones
- Fonctions d'Activation
- Fonctions de Coût d’erreur (Cost)
- **Rétropropagation (BackPropagation)**
- Normalisation
- Optimisation

Backpropagation

- Nous commencerons par construire une intuition derrière la rétropropagation (**backpropagation)**, puis nous nous plongerons dans le calcul et la notation de la rétropropagation.

Backpropagation

- Fondamentalement, nous voulons savoir comment les résultats de la fonction de coût changent par rapport aux pondérations dans le réseau, afin de pouvoir mettre à jour les pondérations pour minimiser la fonction de coût.

Backpropagation

- Commençons par un réseau très simple, où chaque couche ne possède qu'un seul neurone

Backpropagation

- Chaque entrée recevra un poids et un biais

Backpropagation

- Cela signifie que nous avons :
  - **C(w1,b1,w2,b2,w3,b3)**

Backpropagation

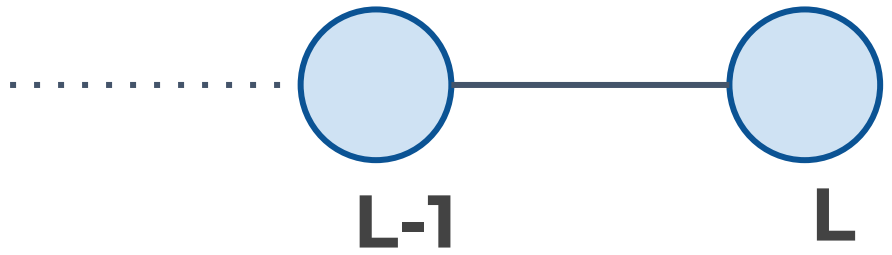
- Commençons par la fin pour voir la rétropropagation.

Backpropagation

- Supposons que nous ayons des couches (layers) **L**, alors notre notation devient :

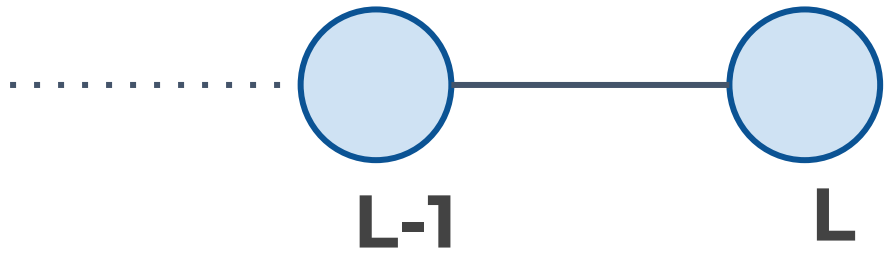
Backpropagation

- En nous concentrant sur ces deux dernières couches, nous allons définir **z=wx+b**
- Ensuite, nous appliquerons une fonction d'activation : **a = σ(z)**



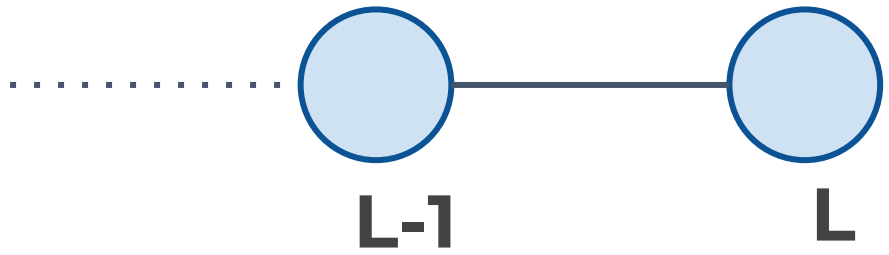
Backpropagation

- Cela signifie que nous avons :
  - $\mathbf{z^L = w^L a^{L-1} + b^L}$



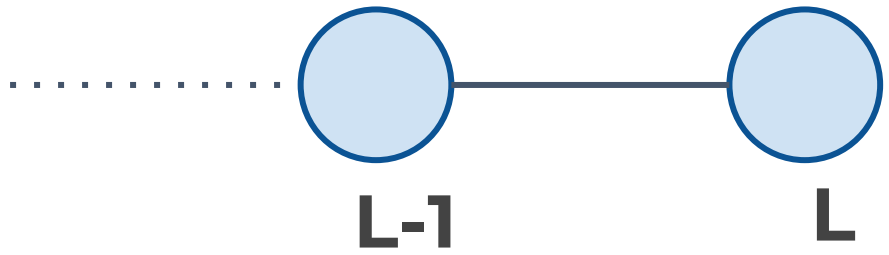
Backpropagation

- Cela signifie que nous avons :
  - $\mathbf{z^L = w^L a^{L-1} + b^L}$
  - $\mathbf{a^L = \sigma(z^L)}$



Backpropagation

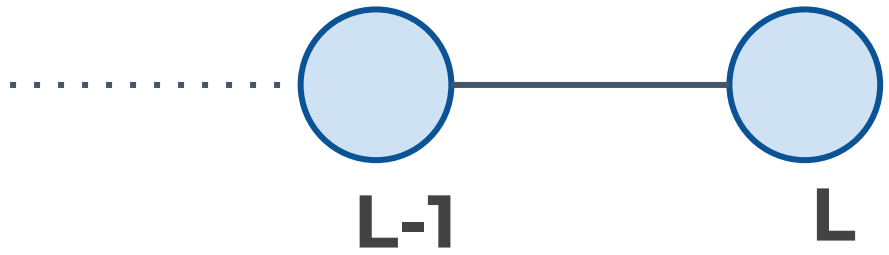
- Cela signifie que nous avons :
  - $z^L = w^L a^{L-1} + b^L$
  - $a^L = \sigma(z^L)$
  - $C_0(...) = (a^L - y)^2$



Backpropagation

- Nous voulons comprendre à quel point la fonction de coût est sensible aux variations de **w** :

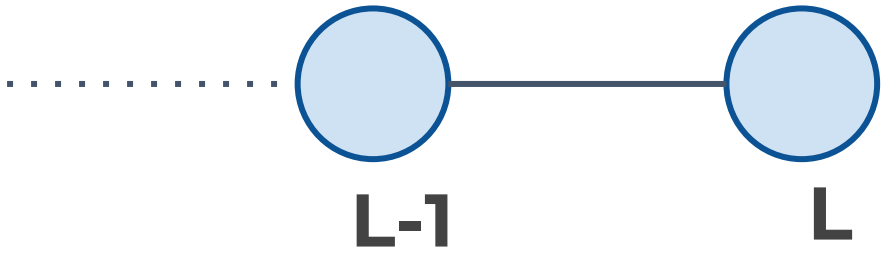
$$\frac{\partial C_0}{\partial w^L}$$



Backpropagation

- En utilisant les relations que nous connaissons déjà ainsi que la règle de dérivation en chaîne :

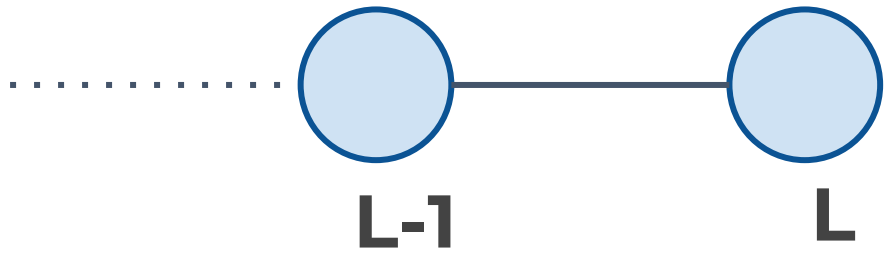
$$\frac{\partial C_0}{\partial w^L} = \frac{\partial z^L}{\partial w^L} \frac{\partial a^L}{\partial z^L} \frac{\partial C_0}{\partial a^L}$$



Backpropagation

- Nous pouvons faire le même calcul pour les termes de biais :

$$\frac{\partial C_0}{\partial b^L} = \frac{\partial z^L}{\partial b^L} \frac{\partial a^L}{\partial z^L} \frac{\partial C_0}{\partial a^L}$$



Backpropagation

- L'idée principale ici est que nous pouvons utiliser le gradient pour remonter dans le réseau et ajuster nos poids et biais afin de minimiser la sortie du vecteur d'erreur sur la dernière couche de sortie.

Backpropagation

- En utilisant une certaine notation de calcul, nous pouvons étendre cette idée aux réseaux comportant plusieurs neurones par couche.
- Produit Hadamard

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

Backpropagation

- Compte tenu de cette notation et de la rétropropagation, nous avons quelques étapes principales pour entraîner les réseaux de neurones.

Backpropagation

- Étape 1 : En utilisant l'entrée **x**, définissez la fonction d'activation **a** pour la couche d'entrée.
  - $\mathbf{z = w\,x + b}$
  - $\mathbf{a = \sigma(z)}$
- Le produit **a** ainsi obtenu alimente la couche suivante (et ainsi de suite).

Backpropagation

- Étape 2 : Pour chaque couche, calculez :
  - $\mathbf{z^L = w^L a^{L-1} + b^L}$
  - $\mathbf{a^L = \sigma(z^L)}$

Backpropagation

- Étape 3 : Nous calculons notre vecteur d'erreur :
  - $\delta^L = \nabla_a C \odot \sigma'(z^L)$

Backpropagation

- Étape 3 : Nous calculons notre vecteur d'erreur :
  - $\delta^L = \nabla_a C \odot \sigma'(z^L)$
    - $\nabla_a C = (a^L - y)$
    - **Exprimer le taux de variation de C par rapport aux activations de sortie**

Backpropagation

- Étape 3 : Nous calculons notre vecteur d'erreur :
  - $\delta^L = (a^L - y) \odot \sigma'(z^L)$
- Maintenant, écrivons notre terme d'erreur pour une couche en fonction de l'erreur de la couche suivante (puisque nous reculons dans le réseau).

Backpropagation

- Étape 4 : Rétropropagation de l'erreur :
  - Pour chaque couche : $L-1, L-2, \ldots$ nous calculons :
    - $\delta^{l}=(w^{l+1})^{T}\delta^{l+1}\odot\sigma'(z^{l})$
    - $(w^{l+1})^{T}$ est la transposition de la matrice de poids de la couche $l+1$

Backpropagation

- Étape 4 : Rétropropagation de l'erreur :
  - Il s'agit de l'erreur généralisée pour toute couche **l**:
    - $\delta^l=(w^{l+1})^T\delta^{l+1}\odot\sigma'(z^l)$
    - $(w^{l+1})^T$ est la transposée de la matrice de poids de la couche L+1

Backpropagation

- Étape 4 : Lorsque nous appliquons la transposée de la matrice de poids $\mathbf{(w^{l+1})^T}$, on peut penser intuitivement que cela revient à déplacer l'erreur vers l'arrière dans le réseau, ce qui nous donne une sorte de mesure de l'erreur à la sortie de la l-ème couche.

Backpropagation

- Étape 4 : Nous prenons ensuite le produit Hadamard $\odot \sigma'(z^l)$. Cela permet de déplacer l'erreur vers l'arrière à travers la fonction d'activation en couche $\mathbf{l}$, nous donnant l'erreur δ^l dans l'entrée pondérée de la couche $\mathbf{l}$.

Backpropagation

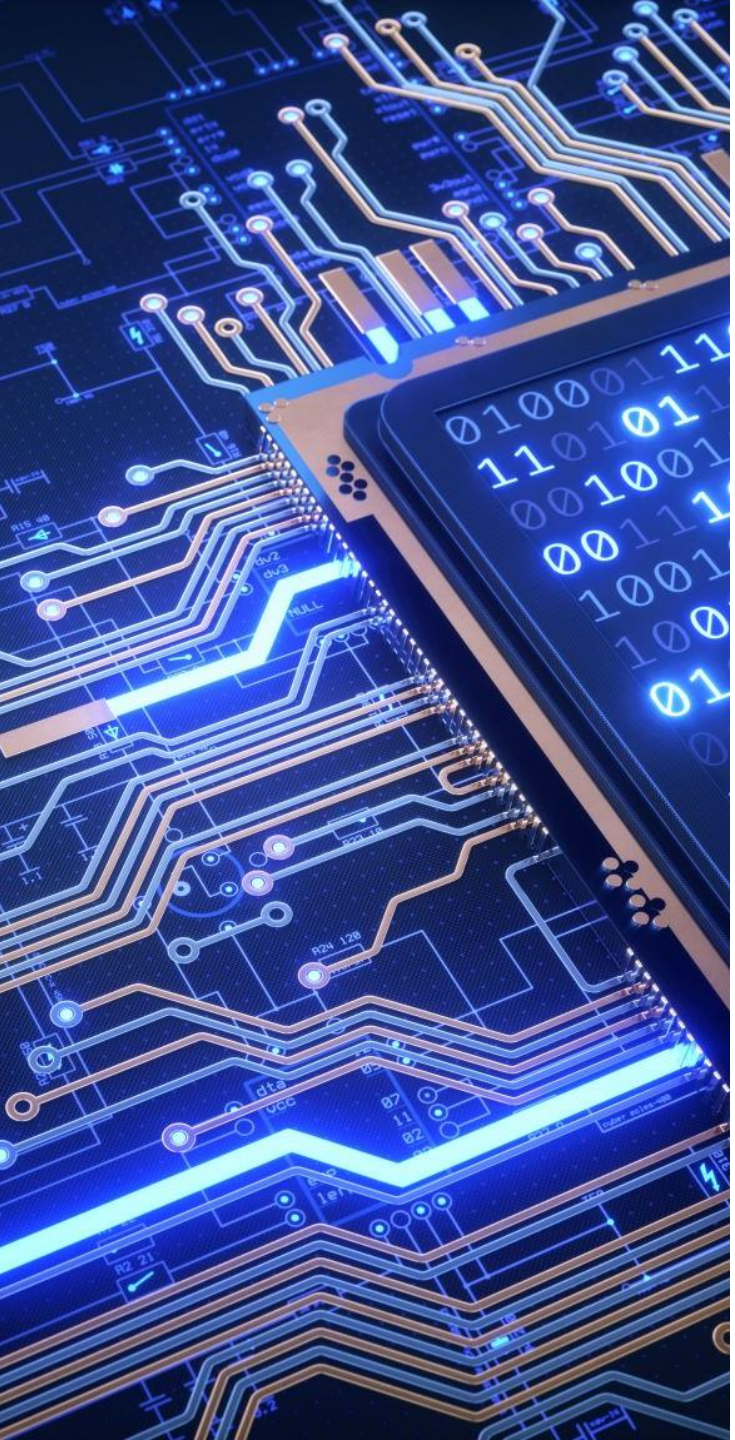
- Le gradient de la fonction de coût est donné par :
  - Pour chaque couche : L−1,L−2,… nous calculons

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \frac{\partial C}{\partial b^l_j} = \delta^l_j$$

Backpropagation

- Cela nous permet ensuite d'ajuster les poids et les biais pour aider à minimiser cette fonction de coût.

- Neural networks and deep learning



Plan

- Du modèle Perceptron aux réseaux de neurones
- Fonctions d'Activation
- Fonctions de Coût d’erreur (Cost)
- Rétropropagation (BackPropagation)
- Normalisation
- Optimisation

Normalisation

- La régularisation est une technologie importante et efficace pour réduire les erreurs de généralisation dans l'apprentissage automatique. Il est particulièrement utile pour les modèles d'apprentissage en profondeur qui ont tendance à être sur-ajuster en raison d'un grand nombre de paramètres. Par conséquent, les chercheurs ont proposé de nombreuses technologies efficaces pour éviter le sur-apprentissage, notamment :
  - Ajout de contraintes aux paramètres, telles que les normes $L1$ et $L2$
  - Élargissement de l'ensemble d'apprentissage, comme l'ajout de bruit et la transformation de données
  - Dropout
  - Arrêt précoce (Early stopping)

Normalisation

- De nombreuses méthodes de régularisation restreignent la capacité d'apprentissage des modèles en ajoutant un paramètre de pénalité $\Omega(\theta)$ à la fonction objectif J. Supposons que la fonction cible après régularisation soit :

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta),$$

- Où $\alpha\epsilon[0, \infty)$ est un hyperparamètre qui pondère la contribution relative du terme de pénalité de norme Ω et de la fonction objectif standard $J(X; \theta)$. Si il est mis à 0, aucune régularisation n'est effectuée. La pénalité en régularisation augmente avec α

L_1 Regularization

- Ajouter une contrainte de norme L_1 aux paramètres du modèle, c'est-à-dire,

$$\tilde{J}(w; X, y) = J(w; X, y) + \alpha \|w\|_1,$$

- Si une méthode de gradient est utilisée pour résoudre la valeur, le paramètre gradient est

$$\nabla \tilde{J}(w) = \propto sign(w) + \nabla J(w).$$

L_2 Regularization

- Ajoutez le terme de pénalité de norme L_2 pour éviter le surapprentissage.

$$\tilde{J}(w; X, y) = J(w; X, y) + \frac{1}{2}\alpha\|w\|_2^2,$$

- Une méthode d'optimisation des paramètres peut être déduite à l'aide d'une technologie d'optimisation (telle qu'une méthode de gradient) :

$$w = (1 - \varepsilon\alpha)\omega - \varepsilon\nabla J(w),$$

- Où ε est le taux d'apprentissage. Par rapport à une formule d'optimisation de gradient commune, cette formule multiplie le paramètre par un facteur de réduction.

L_1 v.s. L_2

- Les principales différences entre $L2$ et $L1$:
  - D'après l'analyse précédente, $L1$ peut générer un modèle plus pauvre que $L2$. Lorsque la valeur du paramètre w est petite, la régularisation $L1$ peut directement réduire la valeur du paramètre à 0, ce qui peut être utilisé pour la sélection des caractéristiques.
  - Dans la régularisation $L2$, la valeur du paramètre est conforme à la règle de distribution gaussienne. Dans la régularisation $L1$, la valeur du paramètre est conforme à la règle de distribution de Laplace.

Extension du jeu de données

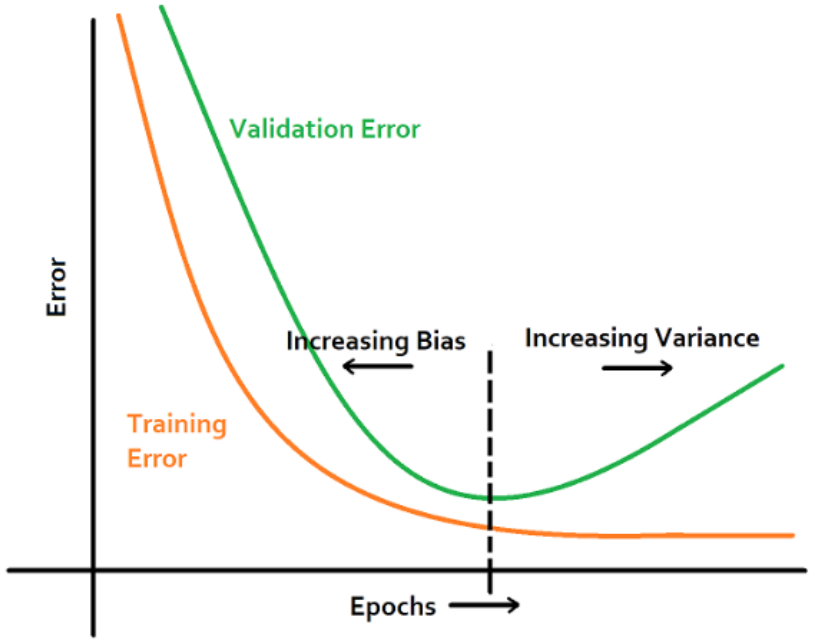
- Le moyen le plus efficace d'éviter le surapprentissage est d'ajouter un ensemble d'entraînement. Un ensemble d'entraînement plus grand a une probabilité de surapprentissage plus petite. L'expansion de l'ensemble de données est une méthode qui permet de gagner du temps, mais elle varie selon les domaines.
  - Une méthode courante dans le domaine de la reconnaissance d'objets consiste à faire pivoter ou à mettre à l'échelle les images. (La condition préalable à la transformation de l'image est que le type de l'image ne peut pas être modifié par la transformation. Par exemple, pour la reconnaissance des chiffres de l'écriture manuscrite, les catégories 6 et 9 peuvent être facilement modifiées après la rotation).
  - Un bruit aléatoire est ajouté aux données d'entrée dans la reconnaissance vocale.
  - Une pratique courante du traitement du langage naturel (TALN) consiste à remplacer les mots par leurs synonymes.

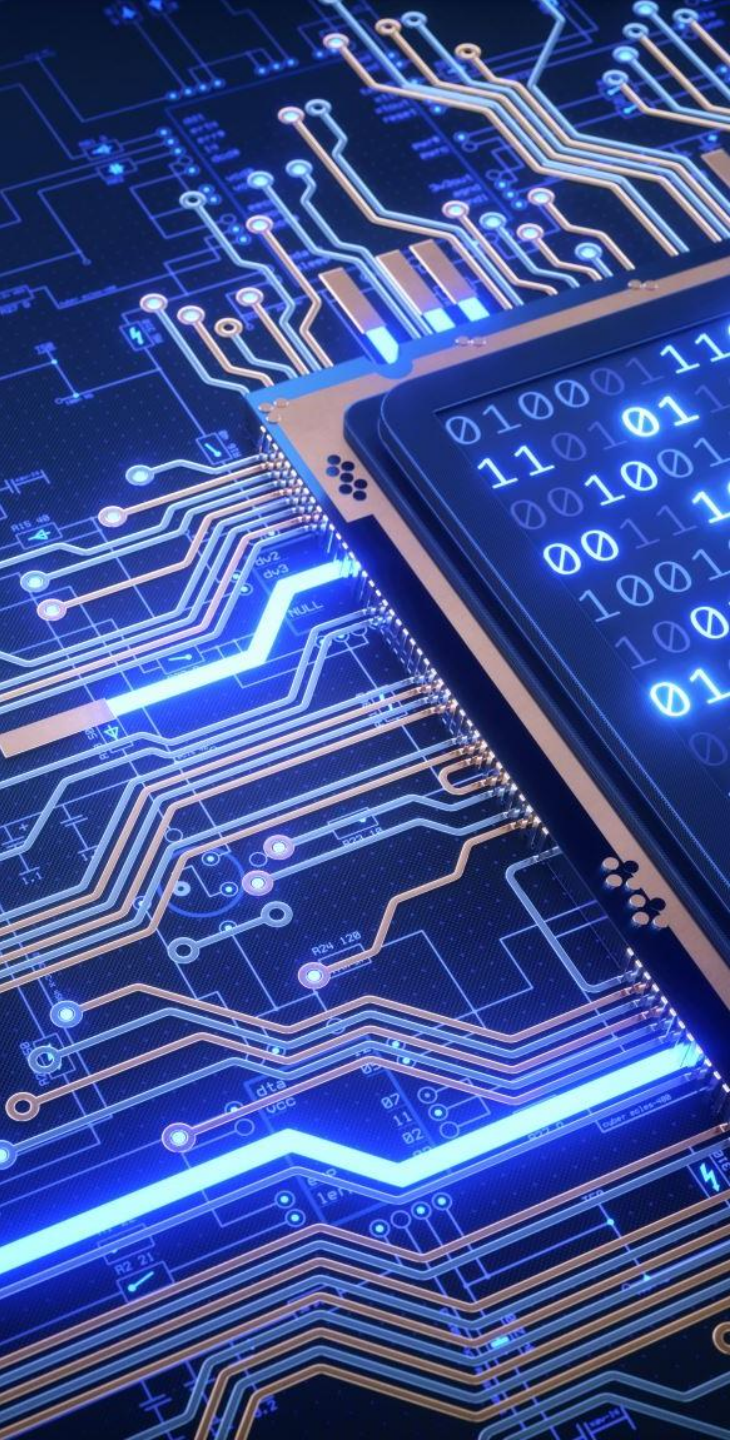
Dropout

- Dropout est une méthode de régularisation courante et simple, largement utilisée depuis 2014.
- En termes simples, Dropout supprime de manière aléatoire certaines entrées pendant le processus d'apprentissage. Dans ce cas, les paramètres correspondant aux entrées rejetées ne sont pas mis à jour.

Early Stopping

- Un test sur les données de l'ensemble de validation peut être inséré pendant la formation. Lorsque la perte de données de l'ensemble de vérification augmente, effectuez un arrêt anticipé.





Plan

- **Du modèle Perceptron aux réseaux de neurones**
- **Fonctions d'Activation**
- **Fonctions de Coût d'erreur (Cost)**
- **Rétropropagation (BackPropagation)**
- **Normalisation**
- **Optimisation**

Optimiseur

- Il existe différentes versions optimisées des algorithmes de descente de gradient. Dans la mise en œuvre du langage orienté objet, différents algorithmes de descente de gradient sont souvent encapsulés dans des objets appelés optimiseurs.
- Les objectifs de l'optimisation de l'algorithme incluent, sans s'y limiter :
  - Accélération de la convergence des algorithmes.
  - Empêcher ou sortir des valeurs extrêmes locales.
  - Simplification du paramétrage manuel, notamment du taux d'apprentissage (LR).
- Optimiseurs communs : optimiseur GD commun, momentum optimizer, Nesterov, AdaGrad, AdaDelta, RMSProp, Adam, AdaMax et Nadam.