# Initiation à l'intelligence artificielle

## Rapport SAE

**Repo github :** https://github.com/SolareFlame/s5_sae_ia

# Partie I – MLP Discrimination

L'objectif ici était simple : construire les perceptrons multicouches de tailles minimales pour résoudre les quatre exemples donnés (le Gaussien, le Cercle, le XOR et la Spirale). Cependant, il existait deux contraintes. On pouvait utiliser uniquement Tanh & Sigmoïde et garder que les deux premiers neurones de la couche d'entrées ($X^1$ et $X^2$).

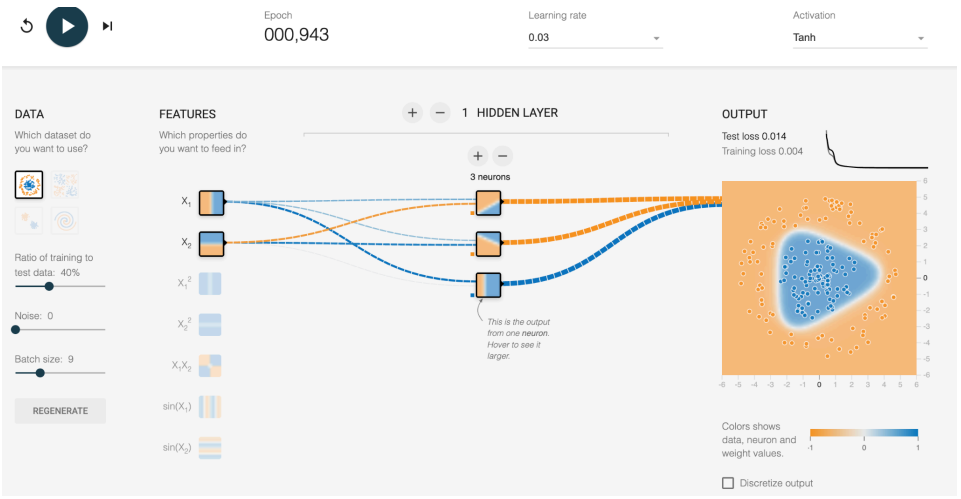
## Analyse des cas

### Le cercle

Première chose que l'on remarque dans ce cas, c'est qu'un groupe de points bleus se trouve au milieu et qu'il est entouré par des points oranges. Ici, une seule ligne droite ne suffit pas à encadrer le bleu.

**Configuration Minimale :**

- Couches cachées : 1 couche.
- Neurones : 3 neurones.
- Fonction choisie : Tanh.

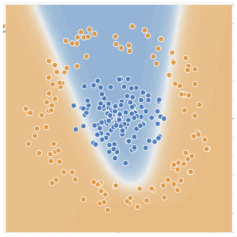



https://playground.tensorflow.org – *Circle Data exemple*

Notre démarche a été simple, géométriquement pour fermer une zone avec une ligne droite, la forme la plus simple, c'est un triangle (il a 3 côtés.). Ainsi, ici, chaque neurone va tracer une ligne qui va correspondre au bord du triangle et une fois que les 3 ont été tracés les points bleus sont isolés des points orange.

De plus, ici, on comprend rapidement le rôle des signes, les poids bleus sont positifs, ça va permettre de renforcer le fait de choisir le bleu pour le neurone suivant, et les poids oranges sont négatifs vu qu'ils s'opposent à l'activation.

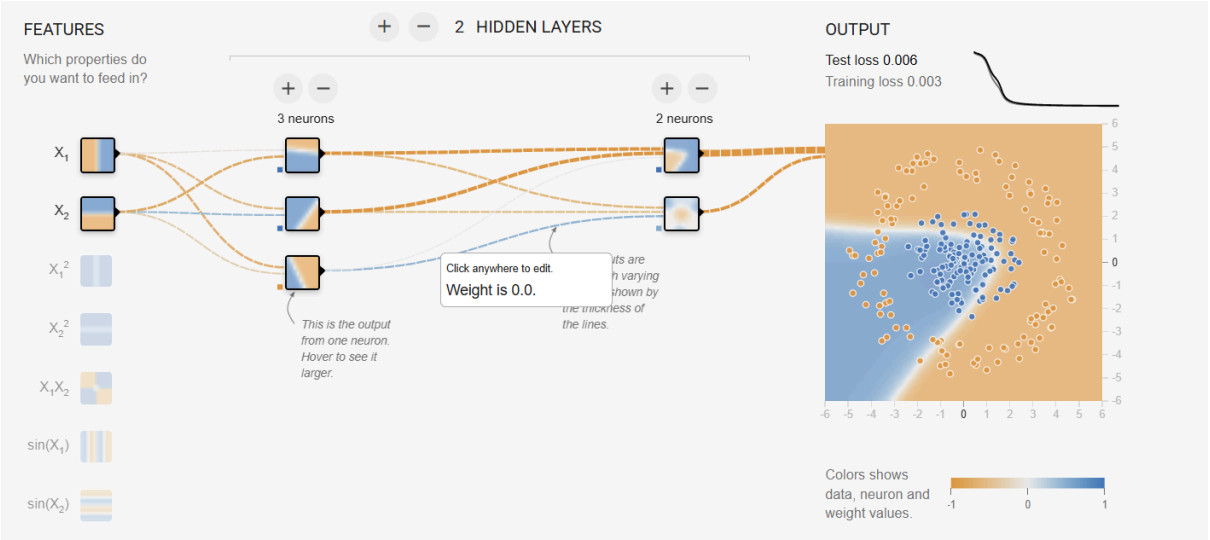
Avec deux neurones, impossible de “refermer” la zone supérieure du triangle de la configuration minimale.



https://playground.tensorflow.org

On remarque donc que des points oranges sont contenus dans la zone bleue.

Maintenant, si on clique sur un lien entre deux neurones pour le mettre, le poids à 0 (dans le cas où avec 3 neurones), le triangle perd un côté. La classification ne marche plus et des points oranges se retrouvent encore dans la zone bleue. Cela prouve que chaque neurone est indispensable.



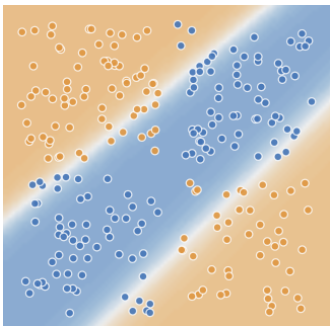

https://playground.tensorflow.org – *Circle Data exemple*

## XOR

Ce cas est spécial et ça ne nous a pas vraiment surpris étant donné qu'il a été vu en cours. On savait déjà qu'il ne pouvait pas être résolu par un simple perceptron. On ne peut pas avoir de solution linéaire dans ce cas.

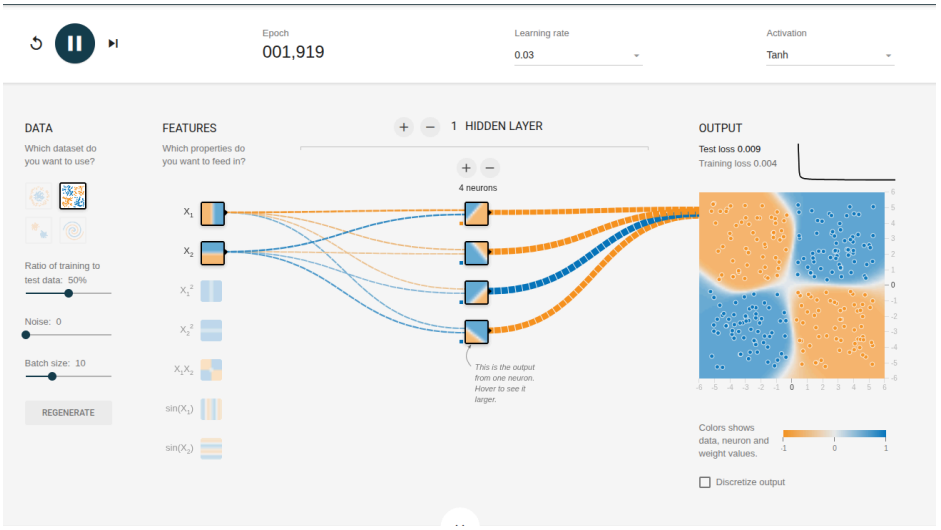
De plus, on peut théoriquement résoudre ce cas avec 2 neurones et une couche cachée.
L'un s'active pour $X_1 \wedge X_2$ (AND), l'autre pour $\neg X_1 \wedge \neg X_2$ (NOR).

Cependant, étant donné que *Tensor Flow* met des poids aléatoirement, la convergence aussi devient aléatoire, donc dans la grande majorité des cas, un minimum local est atteint et cela converge vers une seule ligne tracée.



https://playground.tensorflow.org

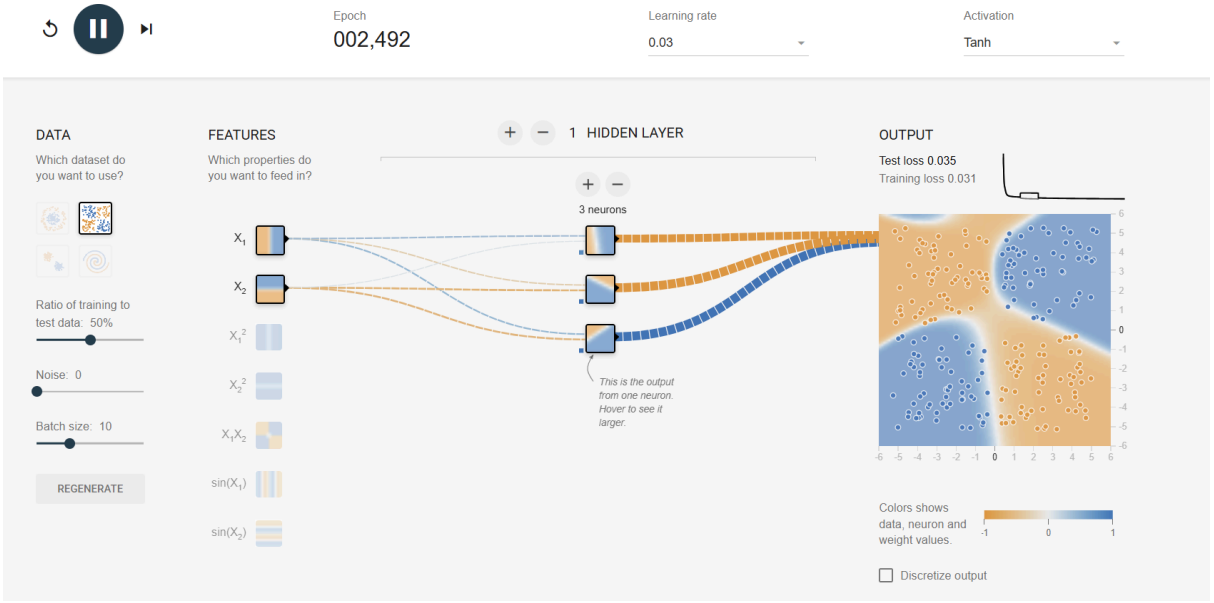
Ainsi, l'utilisation de 4 neurones permet une convergence rapide vers de bons résultats, car il y a 4 séparations, mais ce n'est pas la configuration minimale.




https://playground.tensorflow.org – *Circle Data exemple*

**Configuration Minimale :**

- Couches cachées : 1 couche.
- Neurones : 3 neurones.
- Fonction choisie : Tanh.




[https://playground.tensorflow.org](https://playground.tensorflow.org) – *Exclusive Or Data exemple*

Dans cet exemple, on peut utiliser 3 neurones et une seule couche pour résoudre le problème du OU exclusif, les résultats sont approximatifs mais ça a le mérite de plutôt bien marcher.

Avec la configuration qu'on a trouvée (3 neurones), on peut souvent observer la logique du XOR :
Le neurone 1 (Comportement OR) s'active si au moins une entrée est positive.
Le neurone 2 (Comportement NAND) s'active, sauf si les deux entrées sont positives.
Enfin, le neurone de sortie (AND) : Combine neurone 1 et neurone 2.
Le résultat final est $(A \vee B) \wedge \neg(A \wedge B)$, ce qui est la définition exacte de XOR ($A \oplus B$).

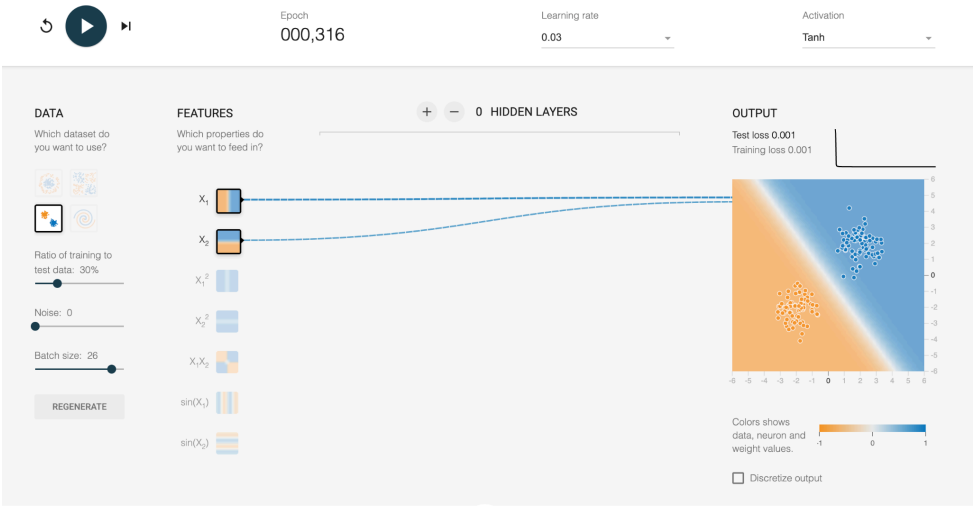
Les poids bleus et oranges nous démontrent bien cette logique : des connexions bleues épaisses indiquent une corrélation positive requise, tandis que des connexions oranges fortes indiquent une inhibition nécessaire, ce qui correspond au NOT dans NAND.

## Gaussien

C'est le cas le plus simple. On a des points bleus et des points oranges bien séparés.

**Configuration Minimale :**

- Couches cachées : 0 couche.
- Neurones : 0 neurones.
- Fonction choisie : Tanh.




*https://playground.tensorflow.org – Gaussian Data exemple*

Pour ce cas, une réflexion simple suffit, car on peut séparer les deux groupes par une simple ligne droite, aucune couche cachée n'est nécessaire.

Si $X_1$ est positif et w1 est positif, le produit est positif (**bleu**).

Si $X_1$ est négatif et w1 est positif, le produit est négatif (**orange**).

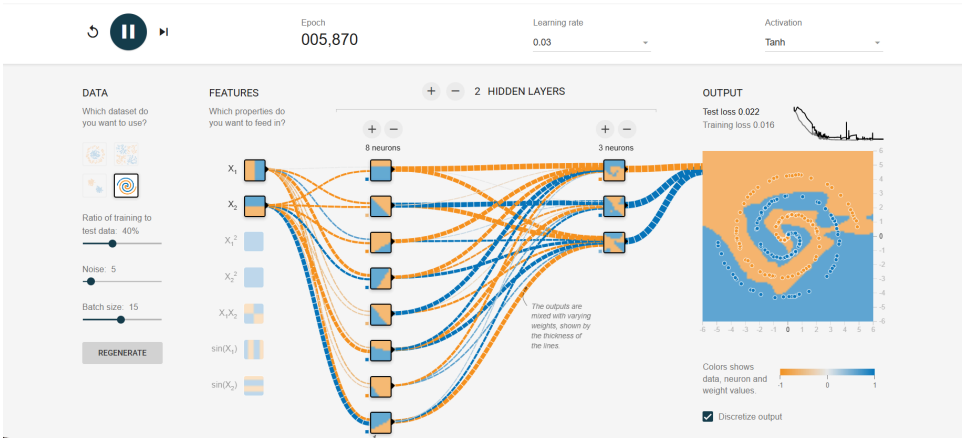
## La spirale

Ce cas a été de loin le plus dur pour nous, car on ne savait même pas comment aborder le problème mathématiquement parlant. La forme tourne est n'est donc pas "fermée", il est difficile de visualiser des contours simples et donc impossibles d'estimer une forme simple comme triangle utilisé dans le cas du cercle.

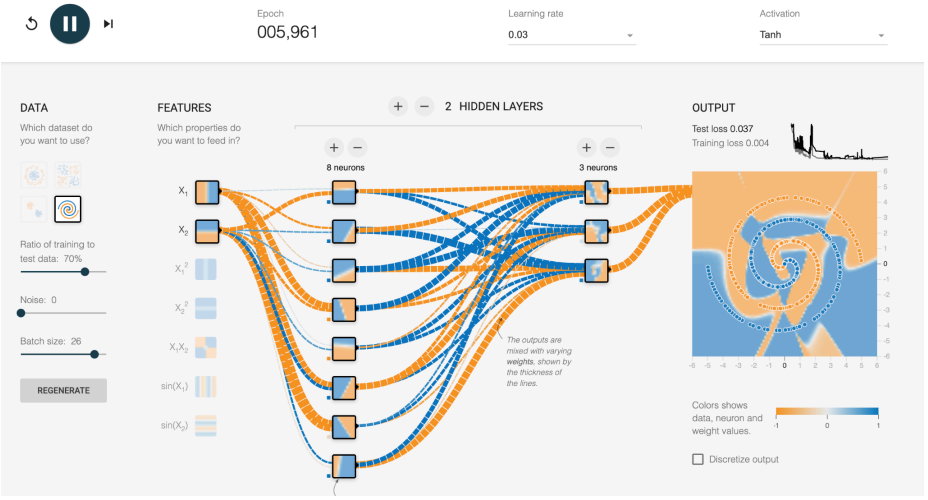
On a donc décidé d'y aller un peu à l'aveugle pour ce cas.
Avec 1 couche, c'est tout simplement impossible parce que le réseau essaie de faire des polygones, mais il n'arrive pas à suivre la forme de spirale. Donc, il faudrait une deuxième couche pour combiner les segments de la première et créer une forme enroulée. On a aussi testé des cas avec sigmoïdes, mais c'est impossible avec cette fonction, on a l'impression que cette fonction n'apprend pas du tout.

**Configuration Minimale :**

- Couches cachées : 2 couches.
- Neurones : 11 neurones.
- Fonction choisie : Tanh.




https://playground.tensorflow.org – *Gaussian Data exemple (11 neuronnes, Than, LR 0,03, Noise 5)*

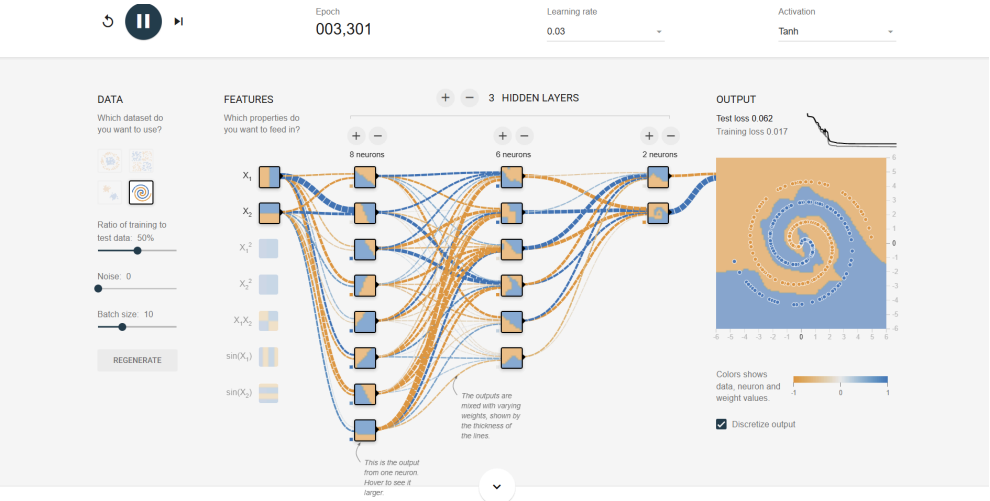



https://playground.tensorflow.org – *Gaussian Data exemple (11 neuronnes, Than, LR 0,03)*

Dans cet exemple, il est possible d'utiliser 2 couches, cumulant donc 11 neurones internes. Cependant, le nombre d'**époques (epoch)** nécessaire peut s'avérer élevé. Cela nous a demandé la modification de plus de paramètres comme le **taux d'apprentissage (learning rate)**, le **bruit (noise)** ou la **taille de lots (batch size)**.

Ci-dessus plusieurs captures d'écrans de plusieurs essais qui se sont avérés concluants sur plusieurs configurations de points en forme de Spirale.

On a eu aussi un résultat visuellement plus satisfaisant en ajoutant une nouvelle couche avec 16 neurones au total.




*https://playground.tensorflow.org – Gaussian Data exemple (16 neuronnes, Than, LR 0,03)*

## Conclusions

Après avoir réalisé les 4 formes disponibles, les conclusions sur les paramètres testés sont les suivantes :

- **Taux d'apprentissage**: À quel point le modèle corrige ses erreurs.

  Si le taux est trop élevé, le résultat ne se stabilise jamais, dans notre cas, cela correspond à de grosses variations de couleurs sans jamais converger vers une forme précise.

  Si le taux est trop bas, le résultat est extrêmement lent, dans notre cas, cela correspond à des variations extrêmement minimes qui demandent un nombre d'époques immense pour atteindre un résultat satisfaisant.

- **Activation**: Fonction qui calibre le signal de sortie du neurone.

  Une fois l'angle de la ligne défini par les poids, l'activation "plie" le résultat mathématique pour le transformer en une zone de couleur. Avec Tanh, elle compresse tout score entre **-1 (Orange)** et **1 (Bleu)**, permettant ainsi de créer des formes complexes au lieu de simples droites.

- **Ratio de données**: Définis la proportion de points utilisés pour l'apprentissage.

  Plus le taux est élevé, plus le modèle subit de contraintes. Il doit s'adapter à une densité de points importante, ce qui rend la forme finale plus précise.

  Plus le taux est faible, plus le modèle est libre. Il doit travailler avec moins de points de repère, il risque de créer des formes imprécises ou de laisser des zones vides, car il manque d'informations pour définir les frontières de couleur qui ressembleraient à un résultat "propre".

- **Bruit**: La propreté des données détermine si les points sont bien rangés ou s'ils sont mélangés. Si on augmente le bruit, certains points oranges et bleus vont se mélanger.
  Le bruit permet d'étudier si le modèle à la capacité d'apprendre à ignorer les points isolés pour ne garder que la forme générale, sans se soucier des points "mal placés".

- **Batch Size**: Le rythme de correction correspond au nombre de points étudiés avant d'appliquer une correction.

  Si on réduit la batch size, le modèle devient "nerveux". Il change d'avis après chaque point étudié et fait un effet de tremblement.

Si on augmente la batch size, le modèle applique moins de corrections par époques. Le nombre d'époques nécessaires pour converger vers une forme adaptée est alors plus élevé, mais cela rend le modèle bien plus stable.

# Partie II – MLP Un peu de programmation !

## Analyse du code existant

On a 4 classes (dont une interface):

- Neuron

```
class Neuron {
   public double    Value;
   public double[]  Weights;
   public double    Bias;
   public double    Delta;

   public Neuron(int prevLayerSize) {
       Weights = new double[prevLayerSize];
       Bias = Math.random();
       Delta = Math.random() / 10000000000000.0;
       Value = Math.random() / 10000000000000.0;

       for(int i = 0; i < Weights.length; i++)
           Weights[i] = Math.random()/ Weights.length;
   }
}
```

Un neurone possède 4 principales variables;

- *Value* : Sa valeur (initialisé à une valeur aléatoire)
- *Weights* : La liste des poids entre ce neurone et tous les neurones précédents
- *Bias* : Le biais du neurone
- *Delta* : Permet de stocker l'erreur (est utilisé pour l'apprentissage par la suite)

Le constructeur prend en paramètre la taille de la couche précédente afin de pouvoir initialiser correctement ses poids.

- Layer

```
class Layer {
   public Neuron Neurons[];
   public int    Length;

   /**
    * Couche de Neurones
    *
    * @param l     Taille de la couche
    * @param prev  Taille de la couche précédente
    */
   public Layer(int l, int prev) {
```

```
        Length = l;
        Neurons = new Neuron[l];

        for(int j = 0; j < Length; j++)
            Neurons[j] = new Neuron(prev);
    }
}
```

Une couche (*Layer*) possède 2 variables :

- *Neurons* : Une liste contenant les neurones que la couche
- *Length* : Le nombre de neurones de la couche

On a un constructeur qui permet d'initialiser une couche, on précise la taille de la couche ainsi que la taille de la couche précédente (nécessaire pour la construction des neurones, qui, prennent en attribut du constructeur la taille de la couche précédente afin d'initialiser les poids)

- Interface TransferFunction

```
interface TransferFunction  {
    /**
     * Function de transfert
     * @param value entrée
     * @return sortie de la fonction sur l'entrée
     */
    public double evaluate(double value);

    /**
     * Dérivée de la fonction de tranfert
     * @param value entrée
     * @return sortie de la fonction dérivée sur l'entrée
     */
    public double evaluateDer(double value);
}
```

Une fonction de transfert contient 2 méthodes :

- *evaluate* : correspond à l'évaluation de l'entré par la fonction de transfert
- *evaluateDer* : correspond à l'évaluation de l'entré par la dérivé de la fonction de transfert

L'interface transferFunction va permettre d'implémenter les fonctions de transfert Sigmoïde et Tangente Hyperbolique. C'est également ce qu'on pourrait appeler une fonction d'activation.

- MLP

La classe MLP possède 3 grosses méthodes : Son constructeur, execute et backPropagation.

*Constructeur* :

```
/**
* @param layers Nb neurones par couches
* @param learningRate tx d'apprentissage
* @param fun Function de transfert
*/

public MLP(int[] layers, double learningRate, TransferFunction fun)
{
   fLearningRate = learningRate;
   fTransferFunction = fun;

   fLayers = new Layer[layers.length];
   for(int i = 0; i < layers.length; i++) {
       if(i != 0) {
           fLayers[i] = new Layer(layers[i], layers[i - 1]);
       } else {
           fLayers[i] = new Layer(layers[i], 0);
       }
   }
}
```

Ce constructeur prend un paramètre; La liste de la taille des couches de neurones de notre réseau, le learningRate (*pas d'apprentissage*) ainsi que la fonction d'activation utilisée pour le réseau.

À l'appelle du constructeur on initialise les couches de neurones, si c'est la première couche on précise que l'on a 0 neurone dans la couche précédente. Et pour le reste on met bien le nombre de neurones dans la couche, ainsi que le nombre de neurones dans la couche précédente.

*execute* :

```
/**
* Réponse à une entrée
*
* @param input l'entrée testée
* @return résultat de l'exécution
*/
public double[] execute(double[] input) {
   int i, j, k;
```

```
    double new_value;

    double output[] = new double[fLayers[fLayers.length -
1].Length];

    // input en entrée du réseau
    for(i = 0; i < fLayers[0].Length; i++) {
        fLayers[0].Neurons[i].Value = input[i];
    }

    // calculs couches cachées et sortie
    for(k = 1; k < fLayers.length; k++) {
        for(i = 0; i < fLayers[k].Length; i++) {
            new_value = 0.0;
            for(j = 0; j < fLayers[k-1].Length; j++)
                new_value += fLayers[k].Neurons[i].Weights[j] *
fLayers[k - 1].Neurons[j].Value;

            new_value -= fLayers[k].Neurons[i].Bias;
            fLayers[k].Neurons[i].Value =
fTransferFunction.evaluate(new_value);
        }
    }

    // Renvoyer sortie
    for(i = 0; i < fLayers[fLayers.length-1].Length; i++) {
        output[i] = fLayers[fLayers.length-1].Neurons[i].Value;
    }
    return output;
}
```

On stock pour chaque entrée la valeur de l'entrée dans le neurone correspondant.

On calcule la nouvelle valeur :
Pour chaque couche
    Pour chaque neurone
        La nouvelle valeur = la somme de : poids du neurone courant * valeur de l'entrée du neurone de la couche précédente
        On retire le biais du neurone courant à la nouvelle valeur
        La valeur du neurone courant = dérivé de la fonction de transfert sur la nouvelle valeur

On fait une boucle pour construire le tableau d'output constitué de toute les valeurs des neurones de sortie (dernière couche)

*backPropagate* :

```
/**
* Rétropropagation
* @param input    L'entrée courante
* @param output   Sortie souhaitée (apprentissage supervisé !)
* @return Error différence entre la sortie calculée et la sortie
souhaitée
*/

public double backPropagate(double[] input, double[] output)
```

Cette méthode prend en argument : Les inputs et la sortie souhaitée.

```
double new_output[] = execute(input);
double error;
int i, j, k;
```

Elle commence par faire passer dans le réseau les inputs afin de donner la sortie réelle de notre réseau (pour pouvoir voir l'erreur entre la réponse voulue et la réponse réelle).

```
// Erreur de sortie
for(i = 0; i < fLayers[fLayers.length - 1].Length; i++) {
   error = output[i] - new_output[i];
   fLayers[fLayers.length-1].Neurons[i].Delta = error *
fTransferFunction.evaluateDer(new_output[i]);
}
```

Ensuite elle parcourt chaque neurones de la dernière couche et ajoute leurs delta. Pour calculer le delta elle doit d'abord calculer l'erreur (la différence entre la sortie attendu et la sortie réelle), puis elle doit multiplier l'erreur par la dérivé de la fonction d'activation avec la sortie réelle du neurone

```
for(k = fLayers.length - 2; k >= 0; k--) {
   // Calcul de l'erreur courante pour les couches cachées
   // et mise à jour des Delta de chaque neurone
   for(i = 0; i < fLayers[k].Length; i++) {
       error = 0.0;
       for(j = 0; j < fLayers[k+1].Length; j++)
           error += fLayers[k+1].Neurons[j].Delta *
fLayers[k+1].Neurons[j].Weights[i];
       fLayers[k].Neurons[i].Delta = error *
fTransferFunction.evaluateDer(fLayers[k].Neurons[i].Value);
   }
   // Mise à jour des poids de la couche suivante
   for(i = 0; i < fLayers[k+1].Length; i++) {
       for(j = 0; j < fLayers[k].Length; j++)
```

```
            fLayers[k+1].Neurons[i].Weights[j] += fLearningRate *
fLayers[k+1].Neurons[i].Delta *
                    fLayers[k].Neurons[j].Value;
        fLayers[k+1].Neurons[i].Bias -= fLearningRate *
fLayers[k+1].Neurons[i].Delta;
    }
}
```

Cette partie remonte toutes les couches du réseau en sens inverse (de l'avant-dernière jusqu'à la première). Pour chaque couche, on calcule d'abord le delta de chaque neurone en propageant les erreurs de la couche suivante, pondérées par les poids de connexion. Ensuite, on ajuste les poids et les biais de la couche suivante en fonction du taux d'apprentissage, du delta calculé et de la valeur d'activation du neurone. C'est le cœur de l'algorithme de descente de gradient qui permet au réseau d'apprendre.

```
// Calcul de l'erreur
error = 0.0;
for(i = 0; i < output.length; i++) {
   error += Math.abs(new_output[i] - output[i]);
}
error = error / output.length;
return error;
```

Enfin, on calcule l'erreur globale du réseau en faisant la somme des valeurs absolues des différences entre chaque sortie prédite et chaque sortie attendue. On divise ensuite cette somme par le nombre de sorties pour obtenir l'erreur moyenne absolue, qui est retournée comme indicateur de performance du réseau.

## Pseudo-code, un squelette

```
fonction Creation()
début
	//on prépare nos couches
	layers <- liste(2, 4, 4, 1) //4 couches avec; 2, 4, 4, 1 neurone(s)

	learningRate <- 0.1
	fonctionActivation <- Sigmoïd()

	//on creer un réseau
	mlp <- MLP(layers, learningRate, fonctionActivation)

	//on creer des donnees pour l'apprentissage:
	//on va faire une porte XOR
	inputs <- liste(liste(0, 0), liste(0, 1), liste(1, 0), liste (1, 1))
	outputs <- liste(liste(0), liste(1), liste(1), liste(0)) //il faut des listes car backpropagation prend des listes même si on n'a qu'une valeur dedans car un seul neurone de sortie

	//on va boucler et faire plusieurs fois backPropagation pour entraîner le modèle sur les données
	i <- 0
	tant que i < 10
		i <- i + 1
		//on parcours chaque input / output
		j <- 0
		tant que j < inputs.size()
			j <- j + 1
			error <- MLP.backPropagate(inputs[j], outputs[j])
			afficher ("erreur à l'appelle " + i + " et l'input " + j + " : " + error)
		fin tant que
	fin tant que

	//ensuite on va tester la qualité de l'apprentissage
	//il suffit de parcourir nos données d'apprentissage et de stocker quel résultat sont correctes afin d'avoir une métrique
	correct <- 0
	incorrect <- 0
	total <- 0

	j <- 0
	tant que j < inputs.size()
		j <- j + 1
		total <- total + 1
		output <- MLP.execute(inputs[j])
		si output = outputs[j]
			correct <- correct + 1
		sinon
			incorrect <- incorrect + 1
	fin tant que
	afficher ("il y a " + correct + " réponse(s) correctes et " + incorrect + " réponse(s) incorrectes sur un total de " + total + " réponses.")
```

```
fin
```

## Applications

Commençons par regarder les résultats de l'apprentissage avec un problème linéairement séparable. On va ici utiliser la porte logique **ET (AND)**. On va analyser comment les paramètres (fonction d'activation, dimension de sortie, architecture) influencent l'apprentissage.

Sigmoïde vs Tanh :
Pour cette comparaison, on a fixé les mêmes paramètres pour les deux fonctions :
On a donc 1 couche cachée de 3 neurones et un taux d'apprentissage de 0.5 pour voir qui apprend le plus vite.

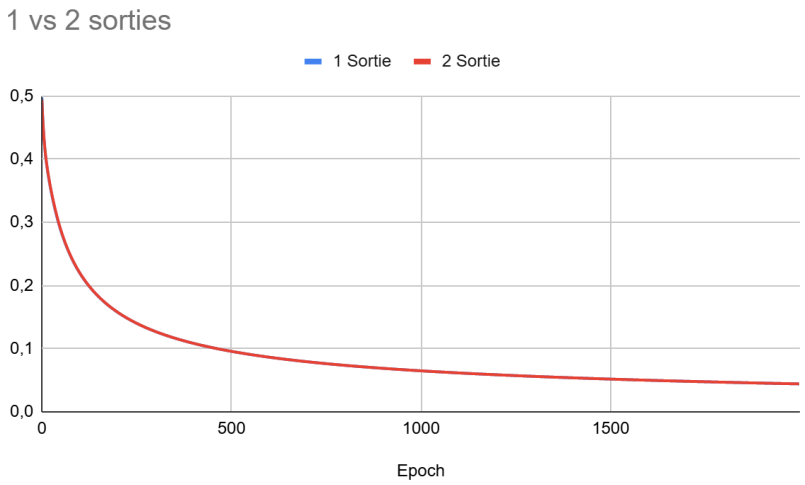
**Fonction :** Sigmoïde,Tanh **Learning Rate :** 0.5 **Nb Epochs :** 2000
**Nb Couches Cachées :** 0 **Sortie :** 1 dimension


On remarque que la fonction Tanh est beaucoup plus performante que la Sigmoïde. La courbe d'erreur de tanh tombe à 0 quasi instantanément (dès les 5 premières epoch), alors que la Sigmoïde a plus de temps pour stabiliser sa précision à 100%.

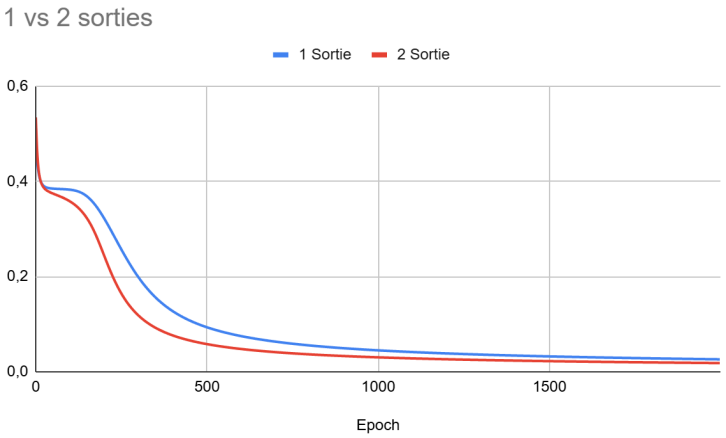
1 vs 2 Neurones
**Fonction :** Sigmoïde **Learning Rate :** 0.5 **Nb Couches Cachées :** 0
**Sortie :** 1 ou 2 dimension

Maintenant comparons les résultats lorsqu'on a une sortie et deux sorties :

On va utiliser Sigmoïde parce que tanh est trop rapide.

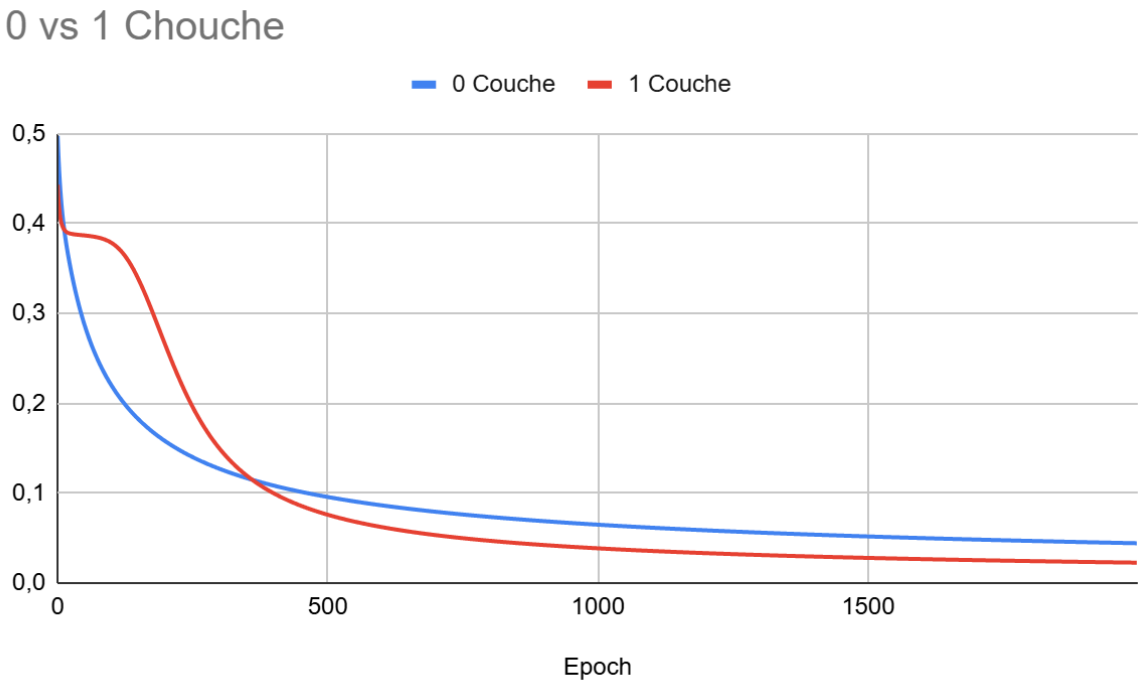



Les deux ont l'air d'avoir les mêmes résultats, cependant si on met 1 couche cachée avec 3 neurones, il y a une petite différence :



Influence du nombre de couches

**Fonction :** Sigmoïde **Nb Couches Cachées :** 0 et 1 (8 neurones)
**Sortie :** 1 dimension




Pour la porte ET, **0 couche cachée suffit**. Avec 0 couche, on converge très bien (Loss ~0.04). Quand on ajoute 8 neurones (dans une couche cachée) , ça n'apporte pas grand chose. Donc pour les problèmes linéairement séparables, le perceptron simple est le meilleur choix.

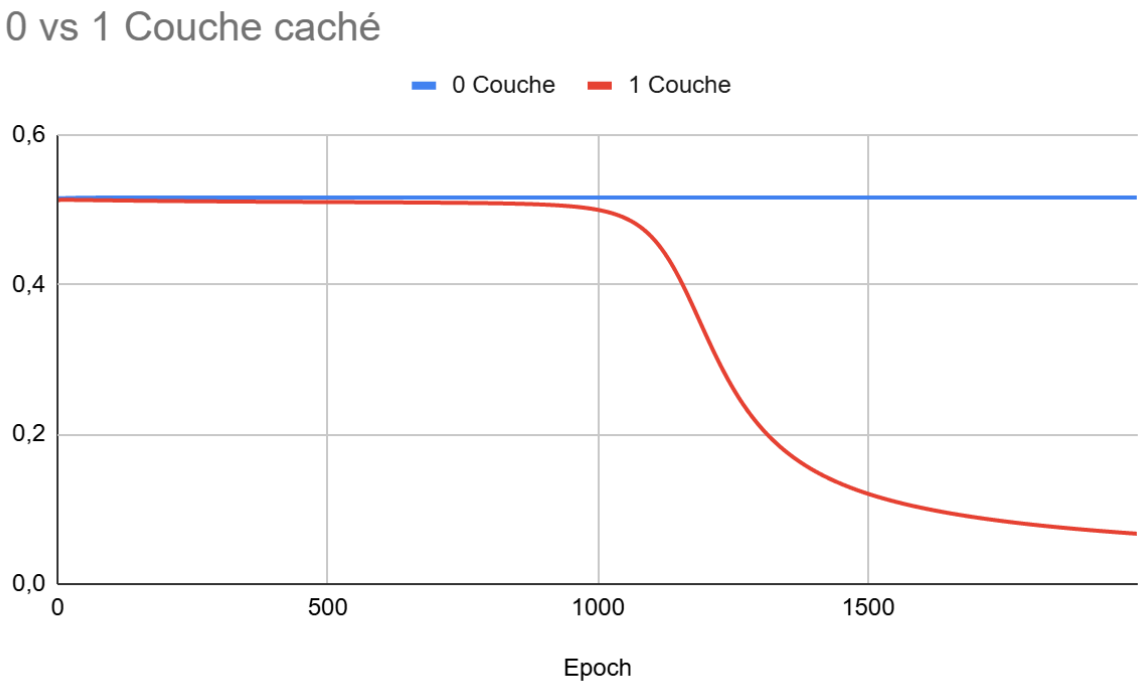
Maintenons, testons XOR

Contrairement à ET, **XOR** n'est **pas linéairement séparable**. Donc en fonction du nombre de couches qu'on choisit, on peut ou non résoudre le problème.

On va comparer un modèle linéaire (0 couche cachée) avec un modèle non linéaire (ici on va mettre 3 neurones cachés dans une couche).

**Fonction :** Sigmoïde **Learning Rate :** 0.5 **Nb Epochs :** 2000 **Sortie :** 1 dimension

**Nb Couches Cachées :** (0 vs 1 couche de 3 neurones)



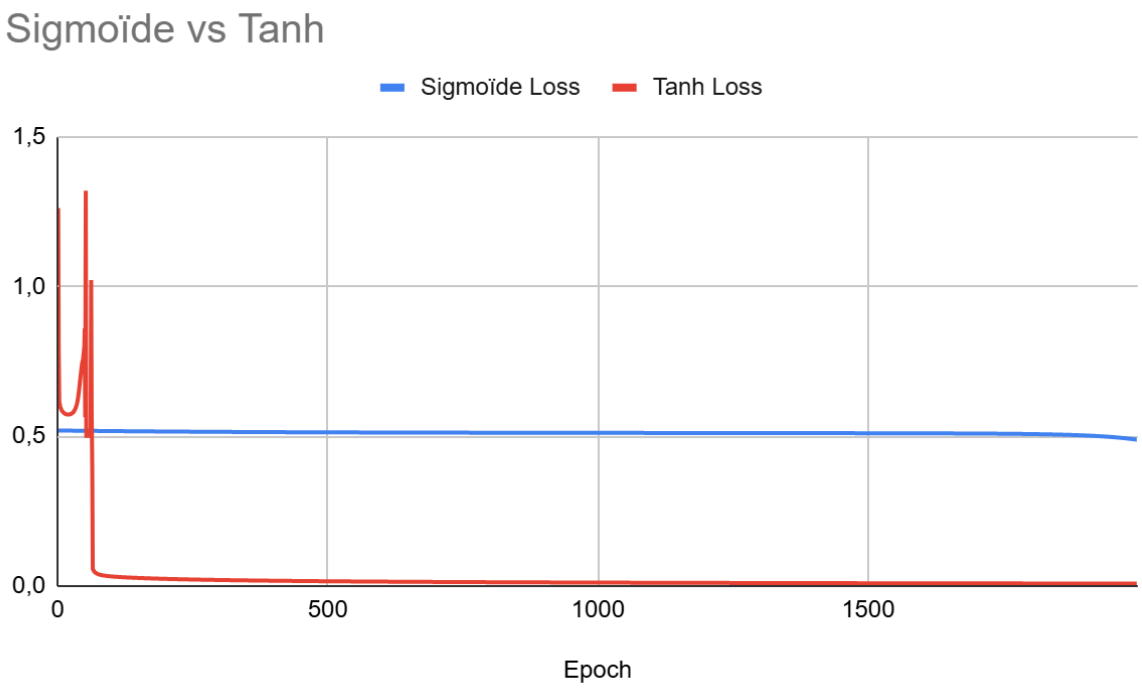

**Avec 0 Couche Cachée :** la courbe ne descend pas (environ 0.5). Le modèle est incapable de séparer les données.

**Avec 3 Neurones Cachés** la courbe descend progressivement jusqu'à 0. La couche cachée permet au réseau de déformer l'espace pour résoudre la non-linéarité du XOR.

Sigmoïde vs Tanh

**Architecture :** 1 couche cachée (8 neurones) **Learning Rate :** 0.5

**Nb Epochs :** 2000 **Sortie :** 1 dimension **Fonction d'activation :** Sigmoïde ou Tanh

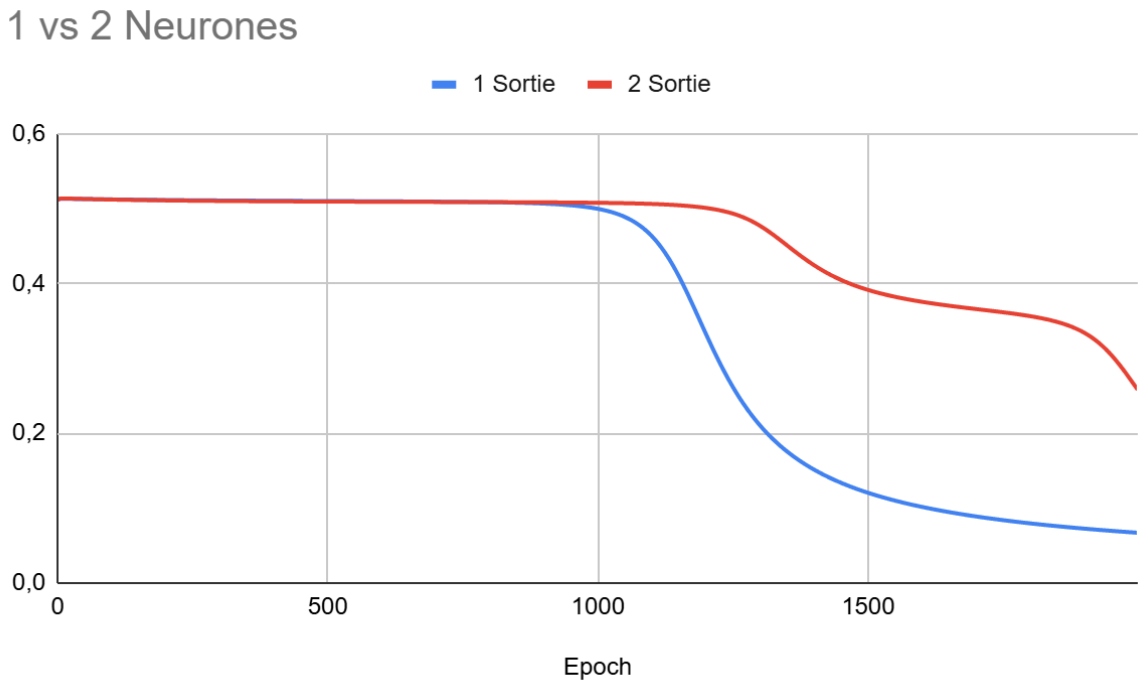



On voit encore que Tanh est plus rapide avec cette fonction, la convergence est très rapide(chute brutale de l'erreur en < 100 époques). Contrairement à Sigmoïde qui stagne stagne longtemps avant de commencer à descendre ( il faut plus que 2000 epoch pour résoudre)

3. Impact de la dimension de sortie (1 vs 2 Neurones)

**Fonction :** Sigmoïde **Learning Rate :** 0.5 **Nb Epochs :** 2000

**Nb Couches Cachées :** 1 couche (3 neurones) **Taille de la sortie** : 1 vs 2




Pour le problème XOR avec 3 neurones, la sortie en **1 dimension** est plus rapide que celle avec 2 dimensions

Cependant, on peut pas être sûr à 100% que les résultats sont vraiment bons, vu qu'on est face à un problème. En fait, le problème avec les portes logiques, c'est que les données d'entraînement et les données de test sont les mêmes, donc en soi il y a rien qui nous dit que MLP n'apprend juste pas par cœur les résultats et les recracher bêtement sans vraiment avoir appris. On va donc s'en assurer dans la partie suivante qui va utiliser MNIST qui a justement des données d'entraînement et des données de test.

# Partie III – MLP vs KNN

Intéressons-nous maintenant à la base MNIST qui contient un grand nombre de chiffres manuscrits. L'objectif ici est d'apprendre au MLP comment reconnaître ces nombres. Cependant, il faut faire attention, car en fonction de la fonction d'activation qu'on utilise, il faut adapter les données. De base, nous avons des valeurs comprises entre 0 et 255 que l'on doit convertir en valeurs entre 0 et 1 pour la fonction sigmoïde, et en valeurs entre -1 et 1 pour la tangente hyperbolique (c'était également le cas avec les tables de vérité).

Commençons par observer l'influence du nombre de couches sur l'apprentissage. Ici, on a fait le choix de tester MNIST avec 10 000 images et un nombre d'epoch fixé à 20, pour que l'apprentissage lancé par notre script qui teste de nombreux cas possibles ne prenne pas trop de temps. À noter qu'à chaque epoch, notre script lance le MLP avec les données de test afin de voir l'évolution du test loss. Cela nous permet de comparer facilement le train loss et le test loss.

La première question que l'on se pose, c'est : sur quelles bases de données doit-on se baser ?

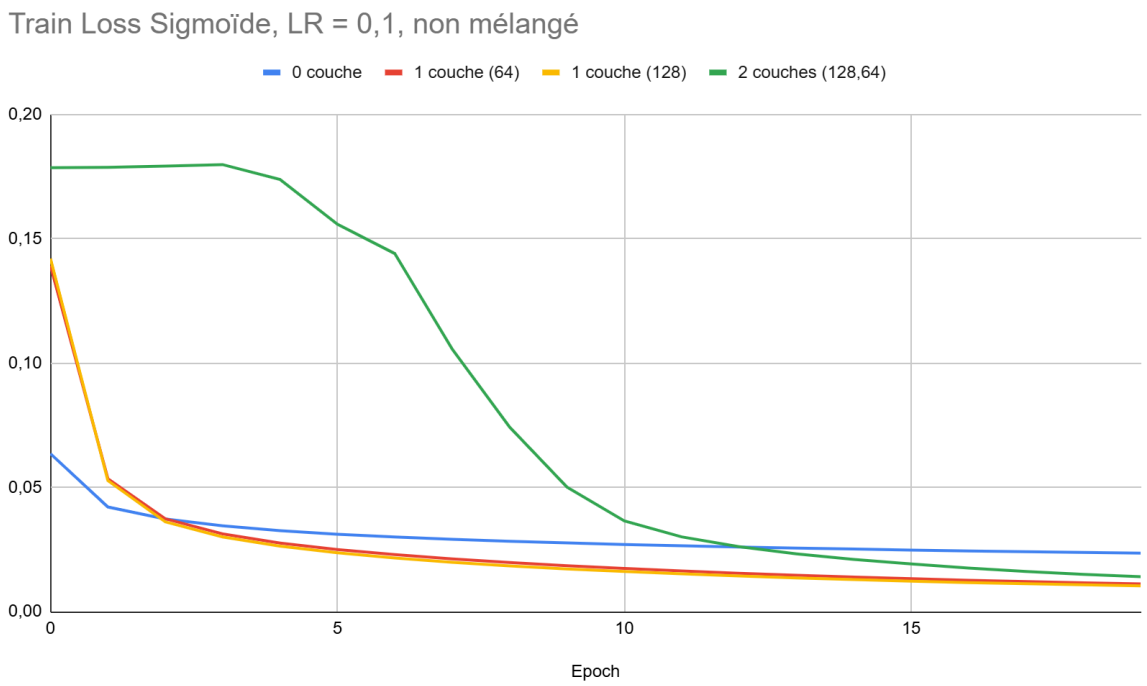
MNIST à deux types de base : une base d'entraînement et une base de test. En soit, a notre repose rien que dans le nom des bases, mais voyons cela plus en détail :

Comparons les différents résultats obtenus entre les données d'apprentissage et les données de test :

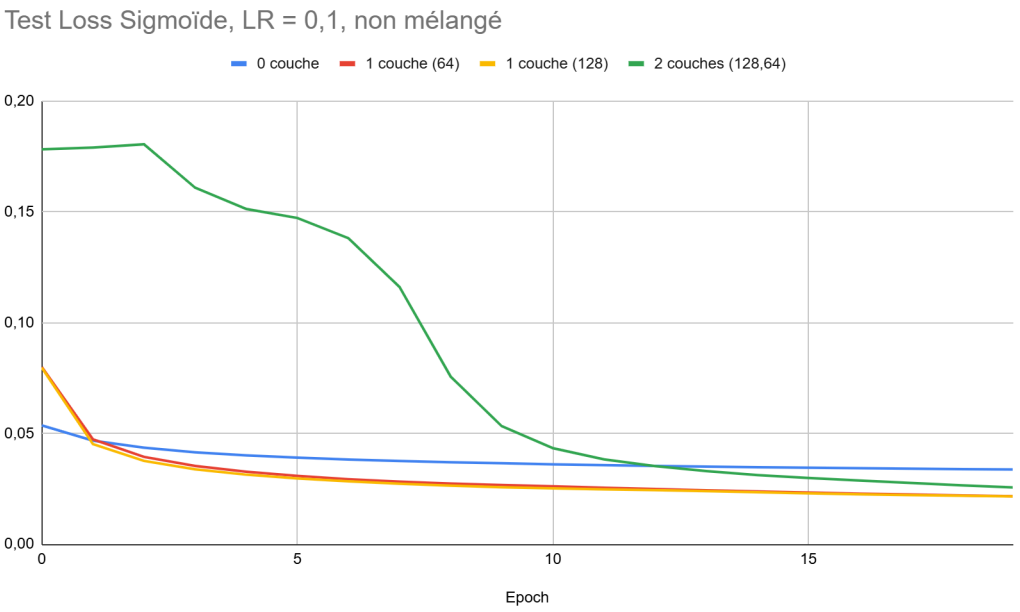
Paramètres utilisé :

- **Fonction :** Sigmoïde
- **Learning Rate :** 0.1
- **Nb Epochs :** 20
- **Nb Couches Cachées :** 0, 1(64-128), 2(128,64)

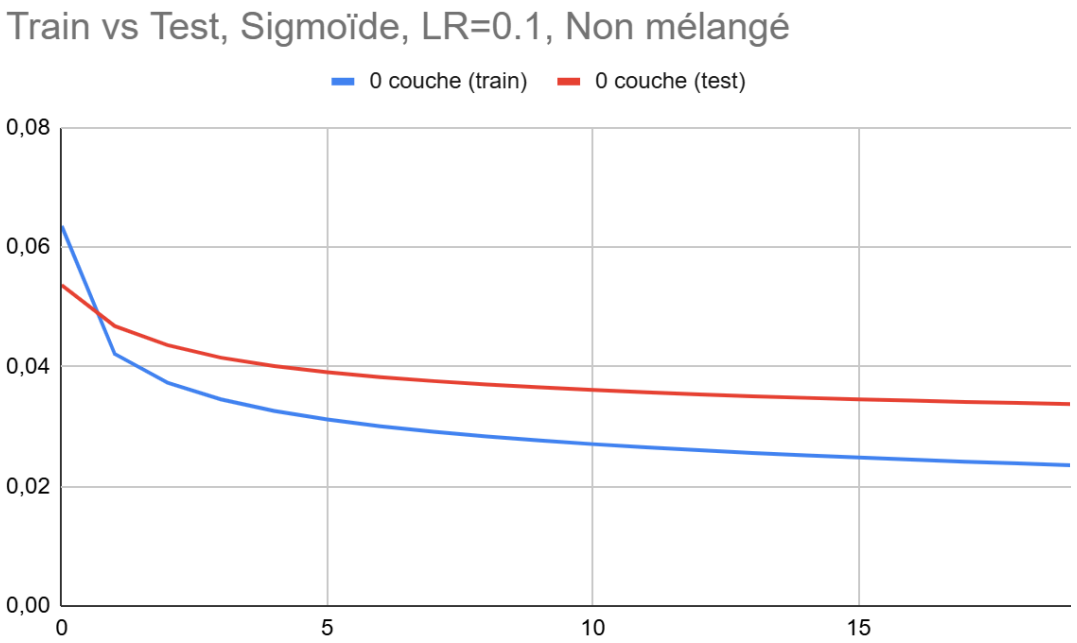
Commençons par regarder comment évolue l'apprentissage avec les données d'entraînement :




Puis faisons la même chose avec les données de test :



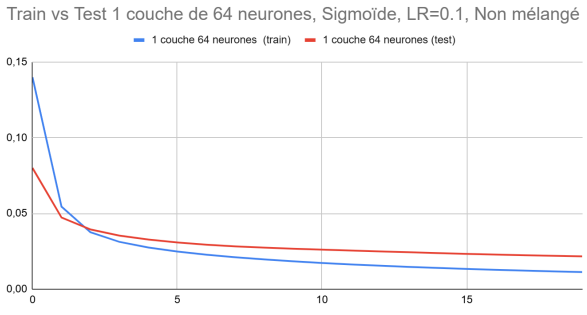

À première vue, les courbes ont l’air très similaires. Cependant, si l'on compare les résultats obtenus entre les données d’entraînement et les données de test, par exemple dans le cas où il n'y aurait pas de couche cachée, nous obtenons ceci :

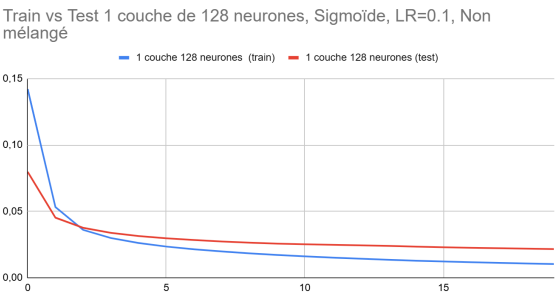


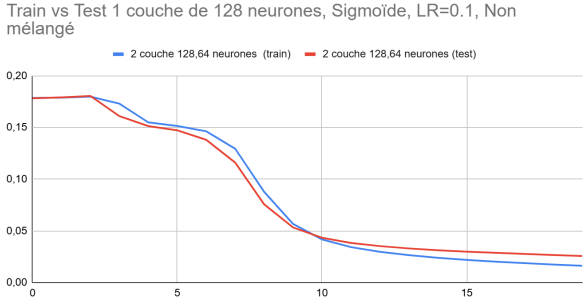

On remarque qu'il y a une légère différence entre les données d'entraînement et les données de test. Ce constat est logiquement normal, étant donné qu'il est logique que le MLP ait plus de mal à identifier des données sur lesquelles il ne s'est pas entraîné par rapport à celles qu'il a déjà vues.

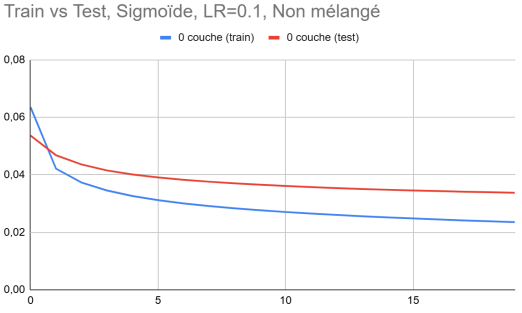
Cependant, on remarque aussi qu'au début (à l'epoch 0) les données de test ont de meilleurs résultats que les données d'entraînement, ce qui est normal, car MLP est moins entraîné quand il fournit les premiers résultats de la base de train que les résultats de la base de test. De toute façon, la tendance s'inverse rapidement.

Et ce n'est pas un cas isolé, on le remarque peu importe le nombre de couches caché :



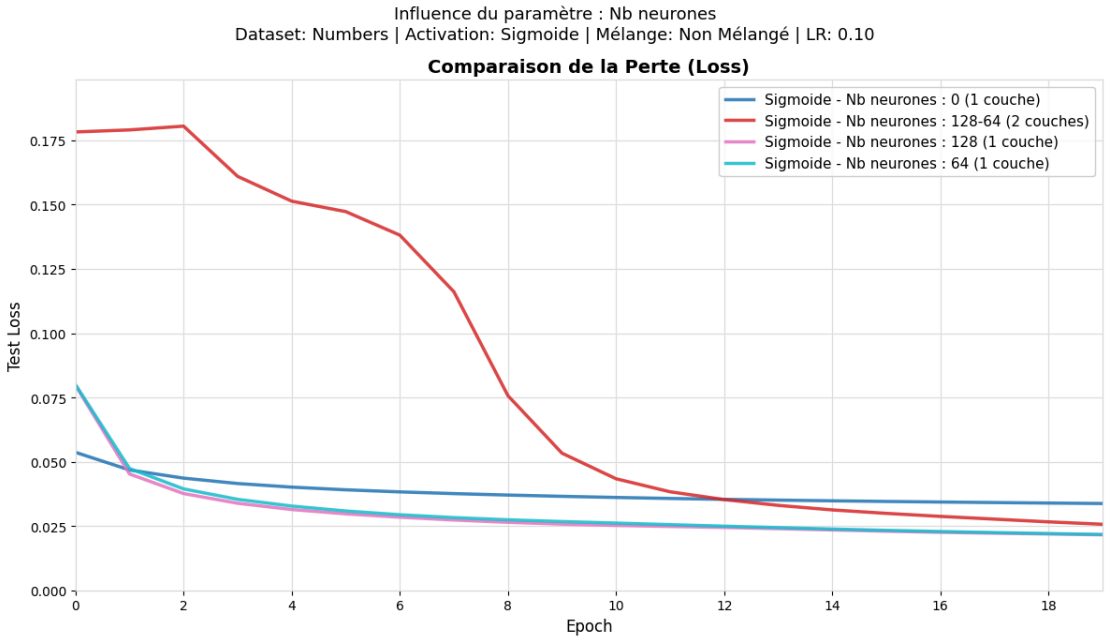
**Quelles données choisir ?**

On a vu sur le graphique que les données de test affichent des résultats légèrement moins bons que les données d'entraînement, ce qui est tout à fait normal. Ce sont bien les données de test qu'il faut privilégier (même si les résultats sont moins bons), car ce sont elles qui nous permettent d'estimer au mieux la qualité de l'apprentissage du MLP. Vu qu'il ne s'est pas entraîné sur ces données, elles valident sa capacité à s'adapter à des données inconnues.

Si l'on se basait uniquement sur les données d'apprentissage, les résultats ne seraient pas fiables car le MLP pourrait très bien apprendre les données par cœur. C'est pour cette raison qu'il est préférable de s'appuyer sur des données jamais vues pour vérifier s'il a réellement appris à reconnaître les formes.

Maintenant, que nous savons quelle base utiliser (la base test), regardons comment les différents paramètres influencent les résultats :

Nombre de neurones / Nombre de couche :




En regardant les courbes, on remarque immédiatement la différence entre le modèle sans couche cachée et ceux qui en possèdent. Le perceptron simple (0 couche) apprend vite au début, mais plafonne rapidement autour de 89.2% de précision. Ce résultat s'explique par la nature du modèle qui tente de séparer les chiffres de manière linéaire. Cependant, les chiffres écrits à la main ne peuvent pas vraiment être séparés par de simples droites, on pense qu'ici le modèle apprend un peu par cœur, c'est pour ça qu'il arrive pas à classer parfaitement.

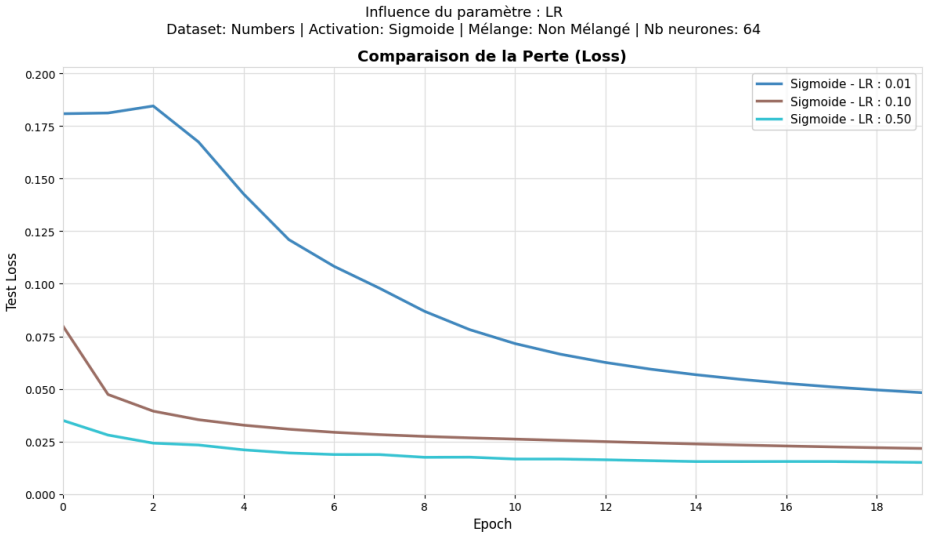
L'ajout d'une seule couche cachée de 64 neurones permet d'atteindre une précision de 93,6 %. Cela montre que le réseau a besoin d'une couche cachée pour bien marcher et comprendre des formes complexes. On a aussi remarqué qu'augmenter le nombre de neurones à 128 n'apporte quasiment aucun gain supplémentaire, la précision finale étant quasiment identique (93,7 %). On peut donc en conclure que 64 neurones suffisent pour comprendre MNIST et en ajouter davantage ne fait qu'augmenter le temps de calcul pour rien.

Enfin, le test avec deux couches cachées (128 puis 64) montre des résultats légèrement inférieurs, tombant à 92,5 %. Contrairement à ce qu'on pourrait penser, complexifier le réseau n'améliore pas forcément les résultats immédiats. Avec deux couches, l'apprentissage devient sûrement plus lent et 20 epochs ne suffisent pas. Donc la meilleure architecture ici est une couche cachée de 64 neurones.

**Learning rate :**

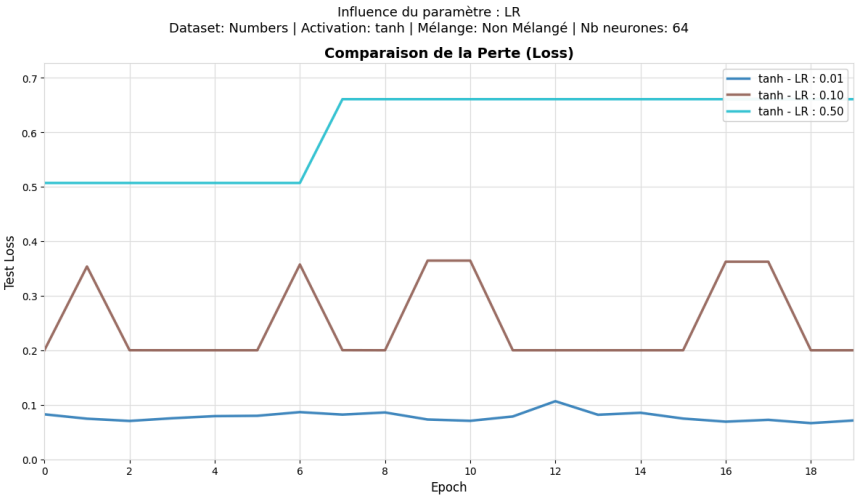
Maintenant analysons l'impact du Learning Rate.

Pour Sigmoïde :




En regardant les courbes, on remarque que Sigmoïde préfère des taux d'apprentissage élevés. Par exemple, avec un learning rate faible de 0.01, la courbe descend très lentement et le modèle stagne à environ 89% de précision. En augmentant le taux à 0.1, puis à 0.5, l'apprentissage est beaucoup plus rapide et atteint une précision de 95,2 % avec un learning rate de 0.5.

Tanh :




À l'inverse, la Tangente Hyperbolique (Tanh) est sensible. Le seul cas qui fonctionne est celui avec le taux le plus faible (0.01), qui permet d'atteindre 85.2% de précision. Dès que l'on augmente le taux à 0.1 ou 0.5, le modèle marche plus, il a une précision de  à 8.7% de précision, encore pire que du hasard.

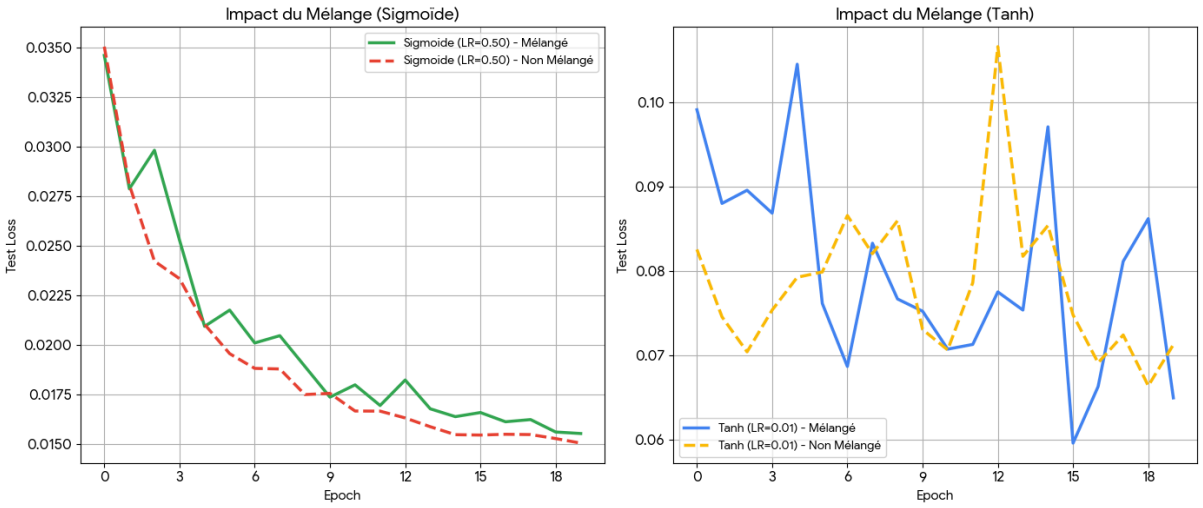
Mais le plus étonnant, c’est que Tanh c’est qu'on pensait qu’il allait être meilleur vu qu'il prend ses valeurs en -1 et 1 donc logiquement ça permettrait de mieux séparer les valeurs, donc on ne comprenait pas pk il avait plus de mal que sigmoïde même avec un learning rate de 0.01. Après avoir enquêté, on s’est dit que cet écart s'expliquerait par l'initialisation des poids.

La sigmoïde qui prend ses valeurs entre 0 et 1 avec des poids entre 0 et 1 qui sont choisis au hasard par la bibliothèque qu'on nous a fournie, alors que Tanh, au contraire, prend ses valeurs entre -1 et 1, alors qu'on l'initialisait avec un poids entre 0 et 1. Donc le réseau apprend donc très lentement durant les premières epochs (le temps de s’adapter), du coup, on pense qu'il a un retard qu'il ne parvient pas à combler sur 20 epochs. (on n'est pas sûr, sur ce coup.)

Donc on remarque finalement que Tanh a besoin d’un réglage beaucoup plus fin que sigmoïde pour bien marcher.

**Impact du mélange :**

Regardons enfin l'impact du fait de mélanger les données pendant l'entraînement. (Ne pas faire une simple boucle 1,2,3...12,13... Mais plutôt 13,2,1,12,3...)

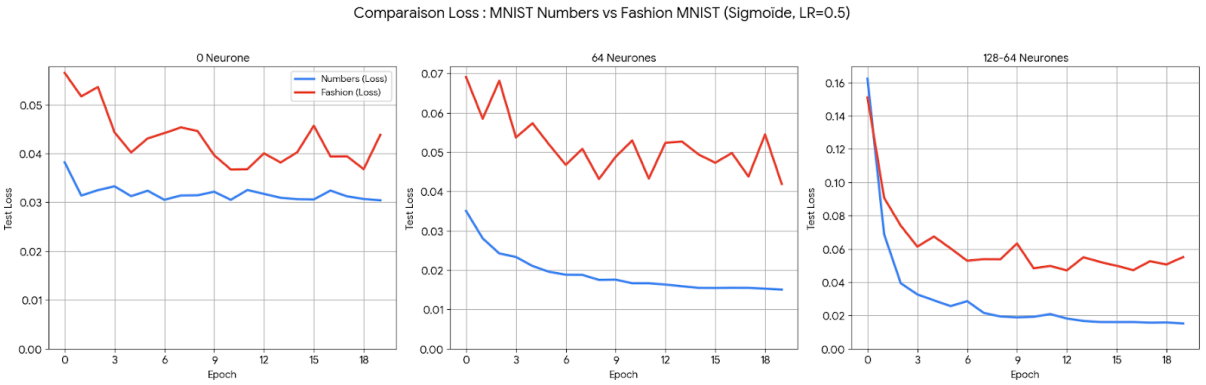



En comparant les courbes avec et sans mélange, on remarque que le mélange améliore la performance.

Pour la Sigmoïde, la différence est faible (95.2% avec mélange contre 94.8% sans). En revanche, pour Tanh, on sent plus la différence. Sans mélange, la précision chute à 83.8% et la courbe de loss montre une convergence moins stable.

Les résultats sont meilleurs dans les deux cas sûrement parce qu'ici, le modèle ne peut pas apprendre par cœur l'ordre dans lequel on lui donne les résultats, donc cela permet de mieux s'adapter quand il est face à des données qu'il ne connaît pas.

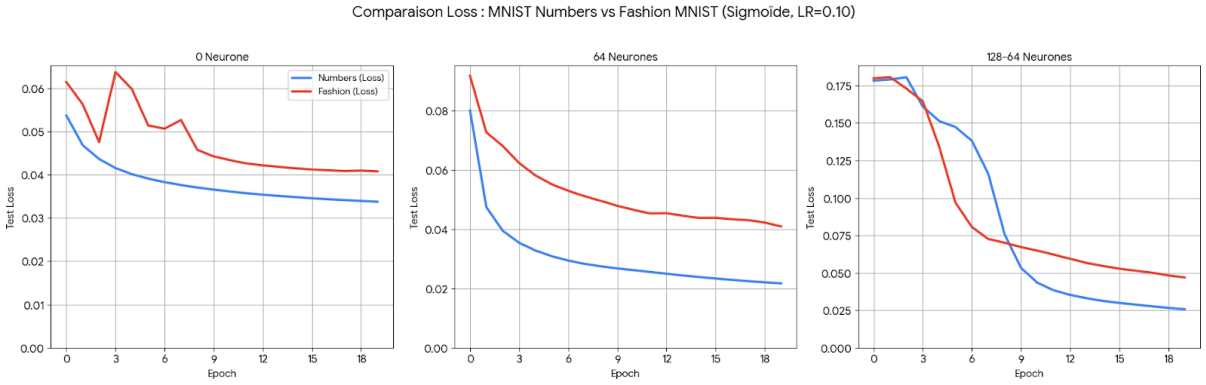
**Maintenant regardons ce que ça donne avec Fashion :**




On a refait l'expérience avec les trois cas principaux (0 neurone, 64 neurones, 128-64 neurones) en utilisant sigmoïde avec un taux d'apprentissage de 0.5.

On remarque que dans tous les cas, c'est un problème plus difficile quelle que soit l'architecture utilisée, la courbe rouge (Fashion) est systématiquement au-dessus de la courbe bleue (Les nombres). Le loss final pour Fashion est entre 0.042 et 0.055. Et le loss final pour les nombres descend jusqu'à 0.015. Cela confirme mathématiquement que la base Fashion MNIST est plus complexe.

Le réseau fait environ 3 fois plus d'erreurs sur les vêtements que sur les chiffres, car la variété des formes (un sac peut avoir plein de formes différentes.) est bien plus grande que celle des chiffres écrits à la main.




Pour être bien sûr, on a testé un taux d'apprentissage plus doux (0.10) afin de voir si cela aidait le réseau à mieux converger sur la base Fashion.

Avec LR=0.50, la Loss finale pour Fashion était entre 0.042 (64 neurones) et 0.055 (128-64 neurones). Et avec LR=0.10, la Loss finale est plus stable : 0.0409 (64 neurones) et 0.0469 (128-64 neurones). Donc baisser le taux d'apprentissage a permis de réduire le sur-apprentissage et les oscillations sur les architectures complexes (128-64). Le réseau apprend plus "calmement" et finit avec une meilleure précision globale sur les vêtements.

Cependant, même avec ce réglage, on retrouve ce plafond de verre autour de Loss = 0.04.

Le Perceptron Multicouche montre ici ses limites. Si pour des chiffres, il atteint l'excellence (Loss 0.015), il marche moins bien pour des images réelles (Loss > 0.04), même en ajoutant des couches.

## Comparaison avec l'algorithme KNN (K-Plus Proches Voisins)

Comparons maintenant de notre réseau de neurones, aux performances de l'algorithme des K-Plus Proches Voisins, configuré avec k=10.

| Base de Données | Algorithme KNN (k=10) | Notre meilleur MLP (64 neurones) | Écart de performance |
|---|---|---|---|
| **MNIST (Nombres)** | **84.48%** | **95.20%** | **+10.72%** (Victoire MLP) |
| **Fashion MNIST** | **77.80%** | **80.50%** | **+2.70%** (Léger avantage MLP) |

**Avantages et Inconvénients des deux méthodes :**

Cette comparaison révèle deux approches opposées au sein du Machine Learning :

**KNN :**

- **Avantages**

C'est extrêmement simple à mettre en place (pas d'entraînement, on stocke juste les données). C'est aussi très "explicable" (on peut montrer à l'utilisateur : "J'ai prédit que c'est une chaussure parce que ça ressemble à ces 10 images").

- **Inconvénients**

C'est très lourd et lent pour prédire. Pour chaque image on doit calculer la distance avec les 60 000 images d'entraînement. De plus, il faut garder toute la base en mémoire (RAM).

**MLP :**

- **Avantages**

Une fois entraîné, le modèle est très léger (quelques Ko pour stocker les poids) et la prédiction est instantanée (quelques multiplications). Surtout, il généralise mieux sur des données inconnues.

- **Inconvénients**

L'apprentissage est long et nécessite de régler finement des paramètres (Learning Rate, Architecture...).

## Conclusion

Ce projet nous permet de voir le fonctionnement d'un Perceptron Multicouche, avec un cas simple tel que les portes logiques, mais aussi dans des cas plus complexes tels que la la classification d'images complexes (MNIST)

Nos tests ont montré plusieurs choses:

- **L'importance de l'architecture**

Contrairement à l'intuition : plus le réseau est gros, mieux c'est, nous avons montré que pour MNIST, une seule couche cachée de **64 neurones** offre le meilleur rapport performance/complexité. Ajouter des couches ou des neurones supplémentaires (128) n'améliore pas les résultats et peut même ralentir la convergence sur des durées d'apprentissage courtes.

- **La sensibilité aux hyperparamètres**

Nous avons observé que la fonction Sigmoïde (avec un learning rate élevé de 0.5) surpasse la Tangente Hyperbolique dans notre cas, ce qui nous a surpris car tanh était plus performante sur la première partie de la Saé sur le site playground tensorflow. On s’est donc dit que c’est peut être parce que Tanh était pénalisée par une initialisation des poids (entre 0 et 1) inadaptée, provoquant un retard comparé à sigmoïde.

- **Les limites du modèle face à la complexité**

On a aussi remarqué que comparés aux chiffres, les vêtements sont moins performants. Alors que le MLP excelle sur les chiffres (Test Loss : 0.015), il marche moins bien sur les vêtements (Test Loss : 0.042). Cela est sûrement dû au fait que les vêtements ont des formes plus complexes que des chiffres.

En conclusion, si le MLP est supérieur à des méthodes naïves comme le KNN (gain de précision et de rapidité d'exécution), il reste limité par sa structure dense pour l'analyse d'images complexes.

# Partie IV – Approfondir les algorithmes

Dans le cadre de cette quatrième partie, nous sommes partis de l'algorithme AlphaBeta standard implémenté lors des précédents TP. Cet algorithme fonctionne correctement pour élaguer l'arbre de recherche, mais il souffrait d'un défaut majeur lorsqu'il atteignait sa profondeur limite (par exemple 4 coups), il ne savait pas comment départager les situations.

### Les limites de l’évaluation linéaire

Dans les versions précédentes, notre algorithme AlphaBeta utilisait une heuristique linéaire. Cette méthode consistait essentiellement à compter le nombre d'alignements potentiels ou le nombre de pions posés, en attribuant un poids fixe à chaque configuration.

Ce système souffrait d'un défaut majeur : il ne traduisait pas correctement l'urgence de la situation. Pour l'algorithme, avoir 4 lignes contenant chacune 1 pion (Score cumulé faible) pouvait sembler équivalent, voire supérieur, à avoir une seule ligne contenant 3 pions (menace immédiate). Arrivé en bout de profondeur de recherche, il était incapable de discerner si une configuration était "dangereuse" ou simplement "prometteuse", ce qui conduisait à des choix sous-optimaux où elle préférait occuper le terrain plutôt que de parer une défaite imminente ou de concrétiser une victoire.

### Exploration d’une nouvelle stratégie

Nous avons donc modifié la méthode d'arrêt de l'algorithme pour y intégrer la stratégie du "Danger" issue de l'article d'Interstices.

La fonction d'évaluation fonctionne de la même manière que celle native, on vérifie l'axe vertical, horizontal et les diagonales. Cependant, au lieu d’attribuer une valeur fixe, on utilise la formule du danger (théorème d’Erdös-Selfridge). L’implémentation du danger est donc la suivante :

```
private double getDanger(char[] board, int r, int c, int dr, int dc, int
k, char me, char opp, int width) {
    int count_me = 0;

    for (int i = 0; i < k; i++) {
        int idx = (r + i * dr) * width + (c + i * dc);
        char val = board[idx];
```

```
        if (val == opp) return 0; // Ligne bloquée
        if (val == me) count_me++;
    }

    return Math.pow(2, count_me);
}
```

Cette méthode évalue le potentiel de menace d'un segment spécifique de k cases (axes étudiés). Elle parcourt les cases du segment pour vérifier deux conditions : la **viabilité du segment** (si la victoire y est possible) et l'**urgence** (si la victoire y est proche).

Si le segment contient au moins un pion adverse, la fonction retourne immédiatement 0, car cette ligne est bloquée et ne peut plus mener à la victoire. En revanche, si la voie est libre, la fonction comptabilise les pions du joueur et applique une croissance exponentielle.

Cette formule permet de distinguer nettement une simple présence (score faible) d'une menace imminente (score élevé), l'incitant à prioriser la complétion ou le blocage des alignements les plus avancés.

### Exemple concret

Pour illustrer l'impact de cette nouvelle heuristique, prenons une situation simple dans une partie de MnK où il faut aligner 4 "pions" pour gagner.

- **Beta (β)** : Possède une ligne dangereuse de 3 pions alignés (victoire imminente), mais ne couvre qu'une seule ligne.

- **Alpha (α)** : Possédons 3 pions dispersés sur le plateau, sans alignement connecté (4 lignes de 1 pion par exemple), mais couvre plusieurs possibilités.

## 1. Heuristique linéaire (Liberté)

L'algorithme ne mesure pas la menace, mais seulement le nombre d'opportunités de victoire, les lignes non bloquées.

**Score α :** 3 lignes disponibles = **3 pts**
**Score β :** 2 lignes disponibles = **2 pts**

**Bilan** (α - β) : 3 - 2 = **1**

L'interprétation est fausse. L'heuristique linéaire de **α** considère qu'elle est en tête (score positif de **1**). Elle ne perçoit pas l'urgence absolue et risque de choisir une branche où elle continue de s'étendre ailleurs au lieu de bloquer, menant à une défaite immédiate au tour suivant.

## 2. Heuristique "Danger" (Exponentielle)

L'algorithme applique la formule de danger pour chaque alignement.

**Score α :** 2^1 + 2^1 + 2^1 = 2 + 2 + 2 = **6 pts**
**Score β :** 2^3 = **8 pts**

**Bilan** (α - β) : 6 - 8 = **-2**

Ici, le score montre que la partie en a l'avantage de **β**.
Plus important encore, si on simule le coup suivant sans bloquer, **β** passera à 4 pions. Le calcul basé sur la liberté augmentera le score de manière négligeable ; tandis que le score basé sur le danger passerait de **8 (2^3)** à **16 (2^4)**. Cette explosion exponentielle du score **β** agit comme une alerte rouge et force le choix sur la branche qui le bloque.