Cette séance est consacrée aux réseaux de neurones, en en particulier aux Perceptrons. En python, les produits de matrices se font avec l'opérateur : @

1 Perceptron learning algorithm

Commençons dans un premier temps par examiner l'algorithme d'apprentissage du perceptron (*Perceptron Learning Algorithm*).

1. Créer deux groupes de points, suffisamment éloignés pour être séparables linéairement, par exemple en utilisant `make_blobs` de sklearn.

   ```
   X, y = make_blobs(n_samples=100, n_features=2, random_state=0,
   ↪  centers=2, cluster_std=0.6)
   ```

   Séparer X et y pour créer un ensemble d'entraînement et un ensemble de test.

2. Construire une classe `Neurone_maison` permettant de réaliser une classification binaire avec un seul neurone. Ce neurone reçoit des entrées x_i ($i = 1, \ldots,$ `input_length`) en entrée et attribue à chacune un poids w_i, par défaut 0,5.

   On complétera cette classe Python pour implémenter le Perceptron Learning Algorithm.

   ```
   class Neurone_maison:
       def __init__(self, input_length, weights=None):
           if weights is None:
               self.weights =          # Initialisation
           else:
               self.weights =          # Poids en arguments

       def activation_function(self,x):
           if x > 0.5:
               return [...]
           return [...]

       def output(self, in_data):
           weighted_sum =          # Somme pondérée des entrées
           activation =          # Valeur de la fonction d'activation
           return activation

       def adjust(self, y_target, y_pred, in_data):
                   # Ajustement des poids si mauvaise classification (si
                   ↪  y_target =/= y_pred
   ```

3. A partir de cette classe, réaliser l'apprentissage du modèle, et étudier après combien d'itérations il converge. N'oubliez pas de tenir compte du biais ($+b$ dans la combinaison

linéaire, ou w_0 dans la notation condensée). Calculer le score de ce classifieur sur l'ensemble de test.

Note : pour aller plus loin, vous pouvez également vous inspirer de ceci :

`http://neuralnetworksanddeeplearning.com/chap1.html`

2 Réseau de neurones

Cet exercice a pour but de construire notre propre classe de réseaux de neurones. L'ambition n'est pas qu'elle soit compétitive en termes d'efficacité par rapport à l'existant (e.g., `scikit-learn`), mais cela va nous permettre de décortiquer les algorithmes à l'oeuvre dans la constitution d'un réseau de neurones.

Nous allons donc construire le réseau de neurones illustré à la Fig.1, qui contient 3 neurones en entrée, 2 en sortie, et une couche intermédiaire ("cachée") de 4 neurones.

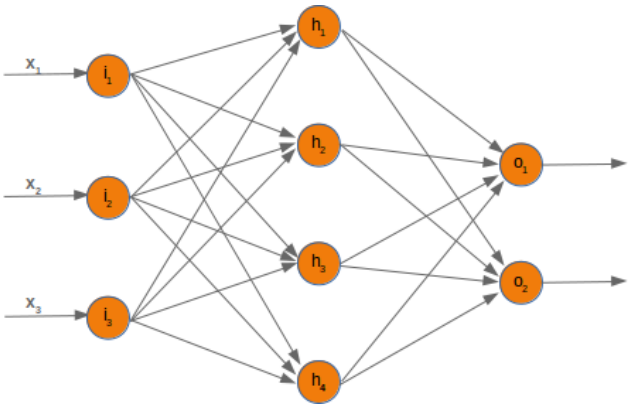


Figure 1: Réseau de neurones à une couche cachée

On ajoutera des poids sur chaque connection, comme illustré à la Fig.2 pour les connections entre la première couche et la couche cachée.

On a donc en conservant les notations de Figure 2 que $h_i = \sum_{j=1}^{3} w_{ij} x_j$. Et en notant W la matrice des poids telles que $Wx = h$, on obtient :

$$\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{pmatrix}$$

Les sorties h_i deviennent les entrées de la dernière couche, et en notant V la matrice des poids :

$$Vh = y \Leftrightarrow VWx = y$$

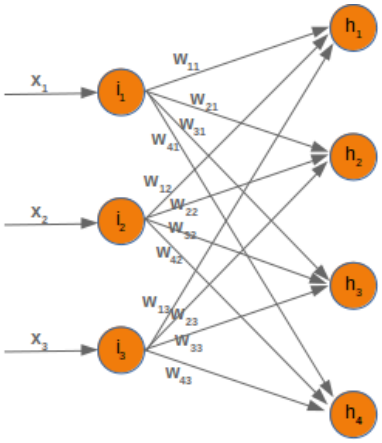
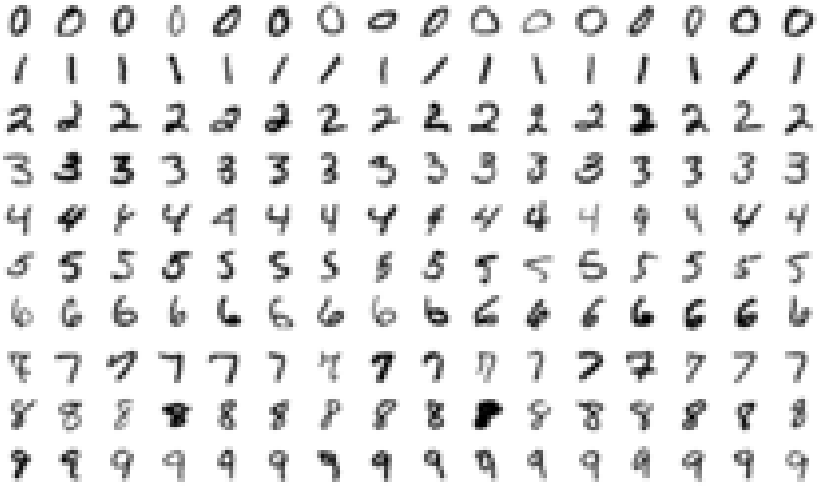


Figure 2: Poids entre la couche d'entrée et la couche cachée

Dans le fichier `ANN_TP.py` disponible sur Moodle, on trouve notamment l'ébauche de classe `NeuralNetwork`. On y trouve déjà implémentées les fonctions `__init__` et `create_weight_matrices`. La première fonction initialise l'object `NeuralNetwork`, c'est à dire qu'on crée tous ses attributs. En particulier on doit initialiser les matrices de poids grâce à la fonction membre de la classe `create_weight_matrices`. Il s'agit donc d'ajouter par vous-même :

1. l'évaluation d'un point : `run`
2. l'apprentissage du réseau : `train`. Pour une fonction d'activation sigmoïde $\sigma(x)$, on a $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

3 Reconnaissance de chiffres manuscrits



Le but de cet exercice est de trouver un algorithme permettant une reconnaissance automatique d'un chiffre manuscrit (entre 0 et 9). Le jeu de données ("MNIST"[1]) est composé d'images isolées comportant chacune un chiffre manuscrit. Des fichiers de données comportent un ensemble d'entraînement (*training data*) et un ensemble de test (*test data*).

On demande d'appliquer divers algorithmes de classification au choix, rapporter les résultats de chacun, et essayer de déterminer le meilleur. Cette question est inspirée d'une compétition[2] sur la reconnaissance automatique de chiffres manuscrits. A titre de comparaison, certains résultats sont disponibles sur le site

`http://yann.lecun.com/exdb/mnist/index.html`.

[1]Modified National Institute of Standards, où le "modified" indique que les images ont subit un prétraitement pour centrer les chiffres autour de l'origine

[2]`https://www.kaggle.com/c/digit-recognizer`