C++ Projet

Le projet consiste à implémenter un réseau de neurones capable d'apprendre à classifier des images en fonction de leurs contenus. Dans un premier temps, nous allons créer l'ensemble des classes destinées à contenir les images et à les charger à partir d'un fichier.

I) Charger les inputs

a) Classe Input

Dans ce projet, nous allons considérer deux types d'inputs : des images et des vecteurs décrivant des fleurs. Ces deux types de données ont en commun d'être représentables sous la forme d'un « tableau », dont la taille va dépendre du type de données contenues (28×28 pour les images et 4 pour les fleurs). Nous reviendrons en détails sur ces deux types d'inputs plus tard. Afin d'homogénéiser le traitement sur ces deux types d'inputs, vous allez créer une classe abstraite, nommée *Input*, qui aura pour rôle de décrire l'ensemble des méthodes nécessaires pour accéder aux informations d'un input.

Les méthodes de la classe sont :

- L'opérateur [ ] qui permet d'accéder à une valeur (*double*) du « tableau » en fonction de son indice (*int*)
- La fonction *get_label* qui renverra un *char* représentant le label de l'input.

Le label d'un input indique à quelle famille il appartient. C'est cette famille que devra reconnaitre plus tard le réseau de neurones. Cette classe abstraite possède un unique attribut (privé) qui stocke le label de l'input.

b) Classe Iris

La classe *Iris* a pour but de stocker la (courte) description de fleurs sous la forme d'un tableau de 4 réels. Il existe 3 types de fleurs : *Iris setosa* (label 0), *Iris virginica* (label 1) and *Iris versicolor* (label 2). Les descriptions de ces fleurs se trouvent dans *iris_training.tar.gz*. Il y a 150 descriptions de fleurs, chacune contenue dans un fichier dont le nom débute par « Iris » et finit par l'indice de la fleur (*Iris0*, …, *Iris149*). Chaque description contient 4 réels, séparés par des points-virgules, ainsi qu'une chaine de caractères indiquant le type de la fleur.

La classe *Iris* descend de la classe *Input*. Les méthodes de la classe *Iris* sont :

- Un constructeur prenant en paramètre un entier correspondant à l'indice de la fleur (entier entre 0 et 149)
- L'opérateur [ ] qui prend en paramètre un entier correspondant à l'indice de la case et qui renvoie le *double* correspondant
- Eventuellement la fonction *get_label* (si elle n'a pas déjà été définie dans la classe *Input*).

Le constructeur va charger la description de la fleur (ainsi que son label) à partir du fichier correspondant.

c) *Classe Image*

La classe *Image* a pour but de stocker des images en noir et blanc, comprenant 28×28 pixels. Chaque pixel est décrit par son niveau de gris, qui est une valeur entre 0 et 255 (qui pourra être stockée dans un *char*). Les images considérées proviennent de la base de données MNIST et représentent des chiffres dessinés à la main. Il y a 10 chiffres différents (0, 1, …, 9). Le label d'un chiffre correspond à sa valeur. Ces images sont stockées dans *MNIST_training.tar.gz* sous la forme de fichiers bmp. Il y a 60 000 images, chacune contenue dans un fichier d'extension bmp dont le nom débute par « training » et finit par l'indice de l'image (*training0*, …, *training59999*). Le fichier contenant les labels de ces images est *train-labels-idx1-ubyte.gz*. Les images, ainsi que les labels, sont contenus dans des fichiers binaires. Chaque valeur de ces fichiers est codée sous la forme d'un octet (dont la valeur pourra être stockée dans un *char*). Pour les images, les octets correspondant aux pixels de l'images se trouvent à partir de l'octet 1078 du fichier d'extension bmp correspondant (et donc jusqu'à l'octet $1078+28^2$). Pour les labels, les octets correspondant au tableau de labels se trouvent à partir de l'octet 8 dans le fichier de labels (le label de l'image d'indice *i* se trouve donc à l'octet *8+i*).

La classe *Image* descend de la classe *Input*. Les méthodes de la classe *Image* sont :

- Un constructeur prenant en paramètre un entier correspondant à l'indice de l'image (entier entre 0 et 59999).
- L'opérateur [ ] qui prend en paramètre un entier correspondant à l'indice du pixel et qui renvoie son niveau de gris sous la forme d'un *double.*
- Eventuellement la fonction *get_label* (si elle n'a pas déjà été définie dans la classe *Input*).

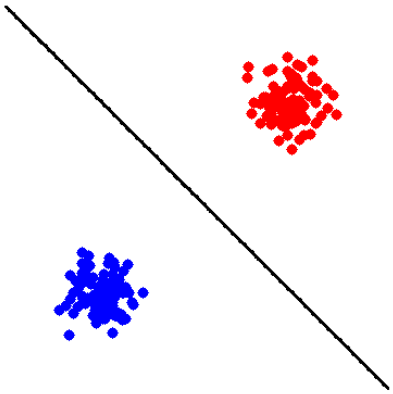
Le constructeur devra charger l'image à partir du fichier bmp correspondant, ainsi que charger le label de l'image dans le champ *label* hérité de la classe *Input*.

II) Perceptron

Maintenant que nous avons des inputs décrits par un vecteur de caractéristiques (le « tableau » de *double* qui correspond soit aux caractéristiques d'une fleur, soit aux pixels de l'image), ainsi que des labels décrivant leurs catégories (type de fleur ou chiffre), nous allons chercher à classifier les inputs en fonction de ces caractéristiques. L'algorithme de classification (notons le $\mathcal{A}: \mathbb{R}^n \to L$) devra prendre en entrée le vecteur de caractéristiques d'un input (que nous allons noter $\boldsymbol{x} = (x_1, \dots, x_n)$) et renvoyer un label correspondant à celui de l'input (que nous allons noter $y \in L$, où L est l'ensemble des labels possibles). L'objectif est donc d'avoir $\mathcal{A}(\boldsymbol{x}) = y$.

Le perceptron est un algorithme de classification linéaire qui s'applique au cas où $|L| = 2$ (c'est-à-dire, il y a deux labels possibles, et donc deux catégories d'inputs). Un perceptron est décrit par un ensemble de poids $\boldsymbol{w} = (w_0, w_1, \dots, w_n) \in \mathbb{R}^{n+1}$ qui définissent un hyperplan (d'équation $w_0 + \sum_{i=1}^{n} w_i x_i = 0$) dans l'espace des inputs. L'objectif est de trouver un hyperplan qui sépare les inputs

de catégories différentes. L'image suivante donne un exemple pour $n = 2$, où la couleur des points (autrement dit des inputs) définissent leurs catégories.



L'algorithme de classification sera donc décrit par l'équation suivante :

$$\mathcal{A}(\boldsymbol{x}) = \varphi\left(w_0 + \sum_{i=1}^{n} w_i x_i\right)$$

où $\varphi: \mathbb{R} \to [0,1]$ est la *fonction d'activation* qui renvoie une valeur entre 0 et 1 permettant de rapprocher l'input d'un des deux labels (si cette valeur est inférieur à 0,5 alors il sera classé 0, sinon il sera classé 1).

Le vecteur de poids $\boldsymbol{w}$ qui définit le perceptron devra être appris à partir des données. Pour apprendre les poids, on a à disposition en ensemble d'inputs avec labels $\{(\boldsymbol{x}^1, y^1), \dots, (\boldsymbol{x}^m, y^m)\}$. L'algorithme d'apprentissage consiste à minimiser la fonction d'erreur suivante :

$$E(\boldsymbol{w}, \{(\boldsymbol{x}^1, y^1), \dots, (\boldsymbol{x}^m, y^m)\}) = \frac{1}{2}\sum_{j=1}^{m}\left(y^j - \mathcal{A}(\boldsymbol{x}^j)\right)^2$$

Afin de minimiser cette fonction d'erreur, nous allons utiliser un algorithme de descente de gradient qui va itérativement mettre à jours les poids de $\boldsymbol{w}$. Voici la description en pseudo-code de cet algorithme :

Choisir aléatoirement $\boldsymbol{w}^0 \in \mathbb{R}^{n+1}$

Pour $k = 1..K$

$$\boldsymbol{w}^k = \boldsymbol{w}^{k-1} - \mu \frac{\partial E(\boldsymbol{w}^{k-1}, \{(\boldsymbol{x}^1, y^1), \dots, (\boldsymbol{x}^m, y^m)\})}{\partial \boldsymbol{w}}$$

où K correspond au nombre d'itérations, μ correspond au *pas de gradient* (appelé aussi *learning rate*), et où $\frac{\partial E(\boldsymbol{w}^{k-1}, \{(\boldsymbol{x}^1, y^1), \dots, (\boldsymbol{x}^m, y^m)\})}{\partial \boldsymbol{w}}$ est le gradient de la fonction d'erreur (c'est un vecteur à n+1 cordonnées). La $i^{ème}$ coordonnée du gradient de la fonction d'erreur correspond à la dérivée de la fonction erreur par rapport à la variable w_i (où tous les autres termes sont considérés comme constant).

Les bases d'apprentissage étant en générale très larges (par exemple, pour les images on a 60 000 inputs), la fonction d'erreur sur l'ensemble de la base peut être difficile à gérer. Une approche *online* (parfois appelée *gradient stochastique*), où l'apprentissage est fait input par input (en choisissant les inputs aléatoirement), est donc souvent privilégiée. Voici la description (plus détaillée) en pseudo-code de cet algorithme.

Choisir aléatoirement $\boldsymbol{w}^0 \in \mathbb{R}^{n+1}$

Pour $k = 1..K$

> *Choisir aléatoirement une image* $(\boldsymbol{x}^j, y^j)$ *dans la base*
>
> *Pour* $i = 0,1..n$

$$w_i^k = w_i^{k-1} - \mu \frac{\partial E(\boldsymbol{w}^{k-1}, \{(\boldsymbol{x}^j, y^j)\})}{\partial w_i^{k-1}}$$

Il est facile de voir que pour le cas du perceptron, le gradient de la fonction d'erreur est défini de la manière suivante :

$$\frac{\partial E(\boldsymbol{w}^{k-1}, \{(\boldsymbol{x}^j, y^j)\})}{\partial w_i^{k-1}} = \begin{cases} \varphi'\left(w_0^{k-1} + \sum_{i=1}^{n} w_i^{k-1} x_i^j\right) \times (\mathcal{A}(\boldsymbol{x}^j) - y^j) \; si \; i = 0 \\ x_i \times \varphi'\left(w_0^{k-1} + \sum_{i=1}^{n} w_i^{k-1} x_i^j\right) \times (\mathcal{A}(\boldsymbol{x}^j) - y^j) \; sinon \end{cases}$$

On va noter $\delta^{k-1} = \varphi'\left(w_0^{k-1} + \sum_{i=1}^{n} w_i^{k-1} x_i^j\right) \times (\mathcal{A}(\boldsymbol{x}^j) - y^j)$ le facteur commun à toutes les coordonnées du gradient.

a) *Classe Fonction_activation*

La classe *Fonction_activation* est une classe abstraite dont les classes héritées représenterons les différentes fonctions d'activation possibles. Cette classe possède les méthodes nécessaires à toute fonction d'activation pour être utilisée par l'algorithme du perceptron.

Les méthodes de la classe sont :

- L'opérateur () qui prend en paramètre une valeur réelle (*double*) et qui renvoie une valeur réelle (*double*) correspondant à l'application de la fonction d'activation au paramètre
- Une fonction membre *prim* qui prend en paramètre une valeur réelle (*double*) et qui renvoie une valeur réelle (*double*) correspondant à l'application de la dérivée de la fonction d'activation au paramètre.

b) *Classe Tanh*

La classe *Tanh* descend de la classe *Fonction_activation* et correspond tout simplement à la tangente hyperbolique (pour rappel, $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ et $\tanh'(x) = 1 - (\tanh(x))^2$). Cette classe surdéfinie naturellement l'opérateur () et la fonction membre *prim*.

c) *Classe Sigmoide*

La classe *Sigmoide* descend de la classe *Fonction_activation* et correspond tout simplement à la fonction sigmoïde (définie comme $\varphi(x) = \frac{1}{1+e^{-x}}$ et $\varphi'(x) = \varphi(x)(1 - \varphi(x))$. Cette classe surdéfinie naturellement l'opérateur () et la fonction membre *prim*.

d) *Classe Perceptron*

La classe *Perceptron* sert à représenter un perceptron, et devra donc stocker l'ensemble des poids ainsi que la fonction d'activation (cela correspondra à deux de ses champs). Un perceptron devra également posséder un champ réel (*double*) supplémentaire servant à stocker la valeur de δ^{k-1} décrite précédemment. Enfin, pour des raisons qui seront claires plus tard, le perceptron possède également un label qui indique pour quelle catégorie le perceptron devra renvoyer

une valeur proche de 1 (dans le cas de deux catégories où les labels sont 0 et 1, ce sera le label 1 par exemple).

Les méthodes de la classe sont :

- Un constructeur qui prend en paramètre un entier correspondant à la taille de l'input (par exemple, 4 pour les fleurs et 784 pour les images), une fonction d'activation (de préférence un pointeur pour avoir une même fonction d'activation commune à tous les perceptrons), ainsi qu'un *char* correspondant au label du perceptron. Le constructeur va initialiser les poids du perceptron de manière aléatoire (par exemple en choisissant des valeurs entières entre -1 et 1 aléatoirement).
- Une fonction membre *get_poids* qui prend en paramètre un indice (*int*) et renvoie une valeur réelle (*double*) correspondant à la valeur du poids correspondant à l'indice en paramètre.
- Une fonction membre *forward* qui prend en paramètre un *Input* (de préférence par référence), et qui renvoie une valeur réelle (*double*) correspondant à l'application de l'algorithme du perceptron à l'input en paramètre (correspond au calcul de $\mathcal{A}(\boldsymbol{x})$).
- Une fonction membre *calcul_delta* qui prend en paramètre un *Input* (de préférence par référence), et qui renvoie une valeur réelle (*double*) correspondant à l'évaluation de la valeur $\delta^{k-1} = \varphi'(w_0^{k-1} + \sum_{i=1}^{n} w_i^{k-1} x_i) \times (\mathcal{A}(\boldsymbol{x^j}) - y^j)$ pour l'input $(\boldsymbol{x}^j, y^j)$ donné en paramètre. Cette fonction va également stocker cette valeur dans le champ *delta.*
- Une fonction membre *get_delta* sans paramètre qui renvoie la valeur du champs *delta*.
- Une fonction membre *backprop* qui prend en paramètre un *Input* (de préférence par référence), ainsi qu'une valeur réelle (*double*) correspondant au pas de gradient μ, et qui ne renvoie rien. Cette fonction va apprendre les poids à partir de l'input en paramètre en utilisant la formule

$$w_i^k = \begin{cases} w_i^{k-1} - \mu\delta^{k-1} \; si \; i = 0 \\ w_i^{k-1} - \mu x_i \delta^{k-1} \; sinon \end{cases}$$

III) Perceptron multicouche sans couche cachée

L'algorithme du perceptron permet de distinguer deux catégories uniquement. Dans les problèmes de classification que nous allons considérer, plus de deux catégories sont nécessaires (3 pour le problème de fleurs, et 10 pour les images). Dans ce cas-là, une solution est d'employer un perceptron par catégorie, et de classer un input selon la classe du perceptron qui renvoie la valeur la plus haute. Plus formellement, lorsque le classifieur porte sur *r* différentes catégories, on considère r perceptrons $\mathcal{A}_0, \mathcal{A}_1, \dots, \mathcal{A}_{r-1}$. Pour un input $\boldsymbol{x} = (x_1, \dots, x_n)$, la catégorie qui sera attribuée à cette input par l'algorithme sera $\underset{l \in \{0, \dots r-1\}}{\operatorname{argmax}} \mathcal{A}_l(\boldsymbol{x})$.

L'algorithme d'apprentissage est essentiellement le même que dans le cas d'un seul perceptron. Le $i^{ème}$ perceptron devra apprendre la sortie correspondant à la $i^{ème}$ catégorie, qui devra être 1 lorsque l'input appartient à cette catégorie, et 0 sinon.

a) Classe NN1

La classe *NN1* sert à représenter un réseau de neurones. Cette classe a pour membre l'ensemble des perceptrons qui constituent le réseau de neurones.

Les méthodes de la classe sont :

- Un constructeur qui prend en paramètre un entier correspondant à la taille des inputs (par exemple 4 pour les fleurs et 784 pour les images), ainsi qu'un entier correspondant au nombre de catégories (et donc le nombre de perceptrons). Chaque perceptron correspondra à une catégorie, et le label associé au perceptron (celui donné à son constructeur) devra être celui de la catégorie associée.
- Une fonction *evaluation* qui prend en paramètre un *Input* (de préférence une référence), et qui renvoie son label (*char* dont la valeur est comprise entre 0 et $r-1$) évalué en recherchant la plus grande valeur retournée par un des perceptrons.
- Une fonction *apprentissage* qui prend en paramètre un *Input* et un *double* (correspondant au pas de gradient μ) et qui va appliquer l'algorithme d'apprentissage pour cet input. Chaque perceptron du réseau de neurones appliquera son propre algorithme d'apprentissage qui dépendra de la catégorie qui lui est associée.

b) *Patron de classe Apprentissage*

Le patron de classe *Apprentissage* sera en charge de l'apprentissage du réseau de neurones. Les paramètres de ce patron sont un paramètre de type qui indique le type d'*Input* que prendra le réseau de neurones (qui sera concrètement soit Iris, soit Image), ainsi qu'un paramètre d'expression de type *int* qui indiquera le nombre d'inputs existant pour le type d'inputs donné précédemment (qui sera concrètement 150 pour les Iris, et 60 000 pour les Images). L'unique membre de cette classe sera un pointeur vers un réseau de neurones de classe *NN1*.

Les méthodes de la classe sont :

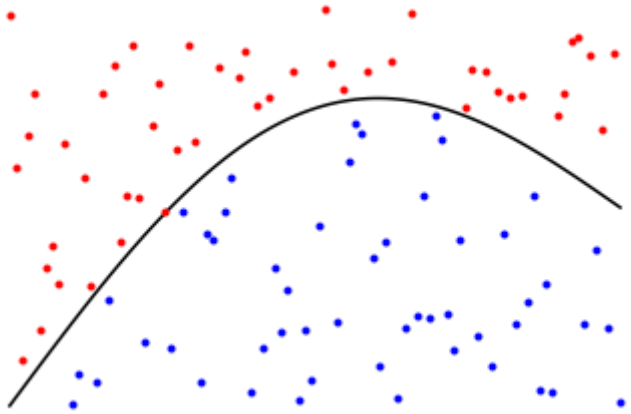
- Un constructeur qui prend en paramètre un pointeur vers un *NN1* qui pointera sur le réseau de neurones qui doit être entrainé.
- Une fonction membre *apprendre_base* qui prend en paramètre un entier correspondant au nombre d'itérations K qui devront être effectuées lors de l'apprentissage, et un *double* (correspondant au pas de gradient μ). Chaque itération de l'algorithme d'apprentissage consiste à choisir au hasard un input de la base et d'appliquer l'algorithme d'apprentissage du réseau de neurones sur cet input.
- Une fonction membre *evaluer* qui n'a aucun paramètre et qui retourne un entier correspondant au nombre d'inputs de la base pour lesquels le réseau de neurones a fait une bonne classification (c'est-à-dire, a retourné le label correspondant à l'input). Cette fonction devra évaluer le label de chaque input de la base (150 pour les fleurs et 60 000 pour les images) et comparer ce label avec celui de l'input.

c) *Tests*

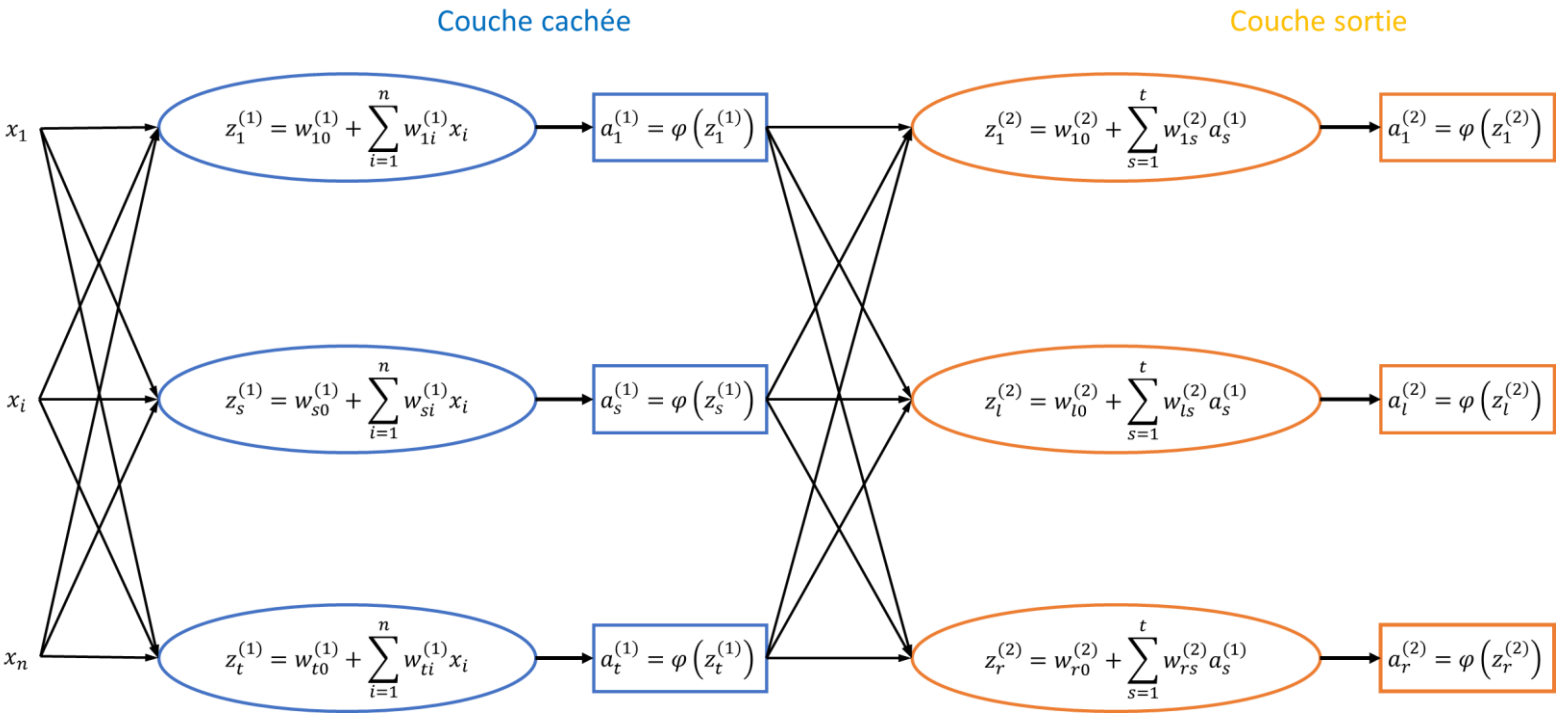
Testez vos classes dans la fonction *main* en faisant apprendre à votre réseau de neurones multicouche sans couche cachée à reconnaitre les fleurs et les images. Pour les fleurs, on pourra utiliser un pas de gradient de 0,1 et un nombre d'itérations de 15 000. Pour les images, on pourra utiliser un pas de gradient de 1 et un nombre d'itérations de 100 000.

IV) Perceptron multicouche avec couche cachée

Un réseau de neurones sans couche cachée (tel que décrit dans la partie III) ne permet de délimiter l'espace des inputs que par l'intermédiaire d'hyperplan séparateurs. Dans certains cas, cette délimitation linéaire de l'espace ne sera pas adaptée. Par exemple, dans l'exemple ci-dessous, il n'existe pas de droite qui permette de séparer les deux catégories d'inputs.



C'est pour cette raison qu'il faut dans certain cas utiliser des classifieurs non-linéaires. Un exemple simple de classifieur non linéaire est le perceptron multicouche avec une couche cachée. Ce modèle de classifieur généralise le perceptron multicouche sans couche cachée, en ajoutant une couche intermédiaire entre l'entrée et la sortie du réseau de neurones. La figure ci-dessous illustre cette construction :



La couche cachée (en bleu) contient t perceptrons (ovales) accompagnés de leurs fonctions d'activation (rectangles). La couche cachée prend comme entrée l'input $\boldsymbol{x} = (x_1, \dots, x_n)$, et renvoie en sortie un vecteur intermédiaire $\boldsymbol{a^{(1)}} = \left(a_1^{(1)}, \dots, a_t^{(1)}\right) \in \mathbb{R}^t$, dont les valeurs sont définies par l'équation :

$$a_s^{(1)} = \varphi\left(w_{s0}^{(1)} + \sum_{i=1}^{n} w_{si}^{(1)} x_i\right)$$

La couche de sortie (en orange) contient r perceptrons (ovales) accompagnés de leurs fonctions d'activation (rectangles). La couche cachée prend comme entrée le vecteur intermédiaire $\boldsymbol{a^{(1)}} = \left(a_1^{(1)}, \dots, a_t^{(1)}\right)$, et renvoie en sortie le vecteur $\boldsymbol{a^{(2)}} = \left(a_1^{(2)}, \dots, a_r^{(2)}\right) \in \mathbb{R}^r$, dont kes vakeurs sont définies par l'équation

$$a_l^{(2)} = \varphi\left(w_{l0}^{(2)} + \sum_{s=1}^{t} w_{ls}^{(2)} a_s^{(1)}\right)$$

Le vecteur $\boldsymbol{a^{(2)}} = \left(a_1^{(2)}, \dots, a_r^{(2)}\right)$ permet de classifier l'input $\boldsymbol{x}$ parmi une des r catégories : la règle consiste à classer l'input dans la catégorie $\underset{l \in \{1, \dots r\}}{\operatorname{argmax}} a_l^{(2)}$.

L'algorithme d'apprentissage est similaire au perceptron multicouche sans couche cachée. Il s'agit de minimiser la fonction d'erreur suivante, en utilisant l'algorithme de descente de gradient de façon online (c'est-à-dire input par input) :

$$E\left(\boldsymbol{w}, \{(\boldsymbol{x}^j, \boldsymbol{y}^j)\}\right) = \frac{1}{2}\sum_{l=1}^{r}\left(y_l^j - \mathcal{N}_l(\boldsymbol{x^j})\right)^2$$

où $\boldsymbol{y}^j = \left(y_1^j, \dots, y_r^j\right)$ est le label correspondant à l'input j (où y_l^j est égal à 0 pour tout l, excepté pour l correspondant à la catégorie de l'input où y_l^j est égale à 1), et $\mathcal{N}_l(\boldsymbol{x^j})$ correspond à la $l^{ème}$ sortie du réseau de neurones appliqué à l'input $\boldsymbol{x^j}$ (qui est noté $a_l^{(2)}$ dans la figure plus haut). Cette fonction d'erreur peut être réécrite :

$$E\left(\boldsymbol{w}, \{(\boldsymbol{x}^j, \boldsymbol{y}^j)\}\right) = \frac{1}{2}\sum_{l=1}^{r}\left[y_l^j - \varphi\left(w_{l0}^{(2)} + \sum_{s=1}^{t} w_{ls}^{(2)} \varphi\left(w_{s0}^{(1)} + \sum_{i=1}^{n} w_{si}^{(1)} x_i\right)\right)\right]^2$$

Le gradient de cette fonction d'erreur en fonction des poids de la couche de sortie est calculé de manière similaire au cas du perceptron :

$$\frac{\partial E\left(\boldsymbol{w}, \{(\boldsymbol{x}^j, \boldsymbol{y}^j)\}\right)}{\partial w_{ls}^{(2)}} = \begin{cases} \left(a_l^{(2)} - y_l^j\right)\varphi'\left(z_l^{(2)}\right) \; si \; s = 0 \\ a_s^{(1)}\left(a_l^{(2)} - y_l^j\right)\varphi'\left(z_l^{(2)}\right) \; sinon \end{cases}$$

où $a_s^{(1)}$, $a_l^{(2)}$ et $z_l^{(2)}$ sont définis plus haut dans la figure. Comme pour le cas du perceptron, nous allons noter $\delta_l^{(2)} = \left(a_l^{(2)} - y_l^j\right)\varphi'\left(z_l^{(2)}\right)$ le facteur commun pour le $l^{ème}$ perceptron de la couche de sortie.

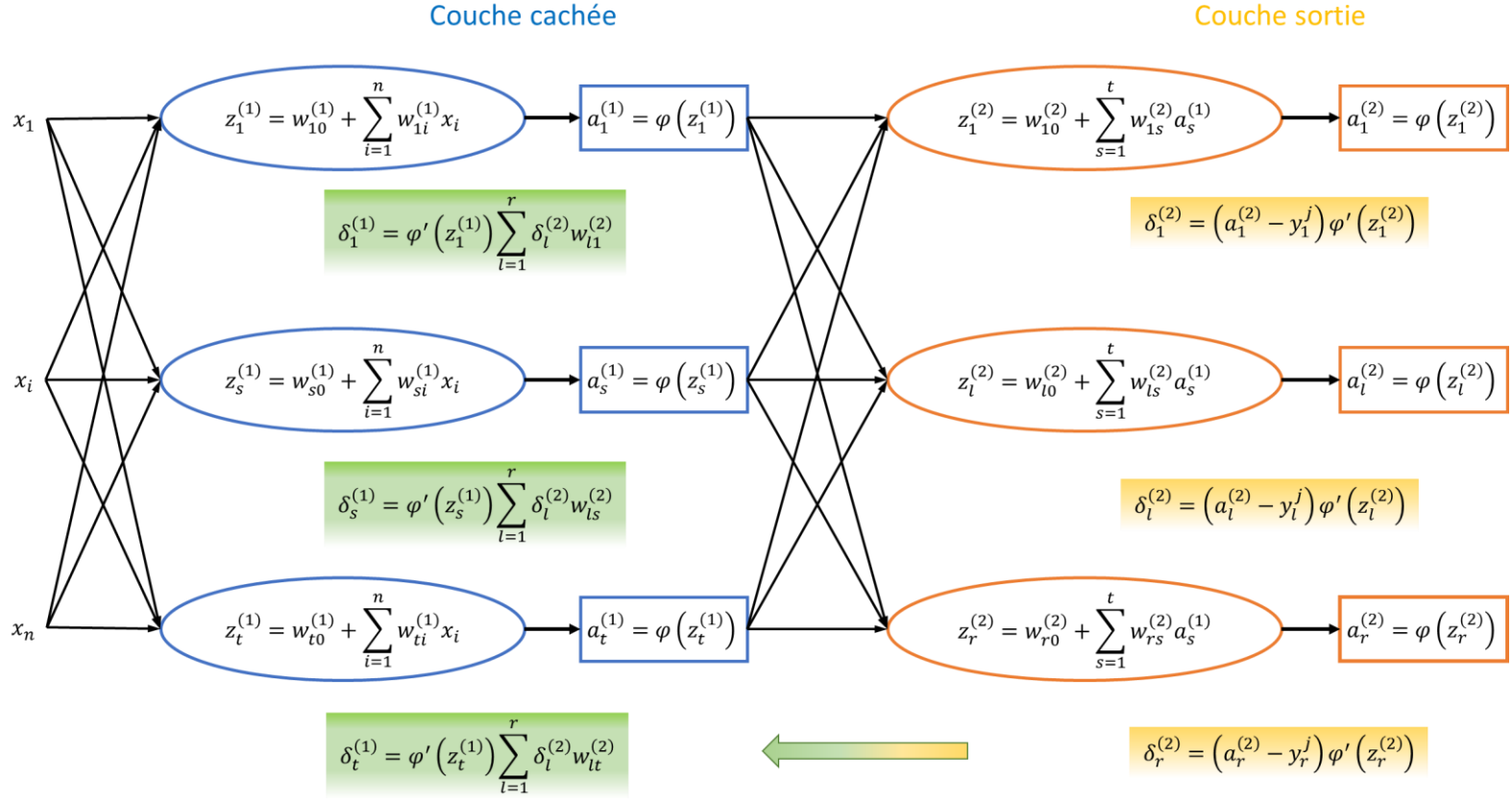
Le gradient de la fonction d'erreur en fonction des poids de la couche d'entrée est calculé de la manière suivante :

$$\frac{\partial E\left(\boldsymbol{w},\{(\boldsymbol{x}^j,\boldsymbol{y}^j)\}\right)}{\partial w_{si}^{(1)}} = \begin{cases} \sum_{l=1}^{r}\left(a_l^{(2)}-y_l^j\right)\varphi'\left(z_l^{(2)}\right)w_{ls}^{(2)}\varphi'\left(z_s^{(1)}\right) & si\ i=0 \\ \sum_{l=1}^{r}\left(a_l^{(2)}-y_l^j\right)\varphi'\left(z_l^{(2)}\right)w_{ls}^{(2)}\varphi'\left(z_s^{(1)}\right)x_i & sinon \end{cases}$$

où $a_l^{(2)}$, $z_l^{(2)}$ et $z_s^{(1)}$ sont définis plus haut dans la figure. Le facteur $\delta_l^{(2)} = \left(a_l^{(2)}-y_l^j\right)\varphi'\left(z_l^{(2)}\right)$ apparait dans les deux termes, et on peut donc le remplacer et obtenir l'expression suivante :

$$\frac{\partial E\left(\boldsymbol{w},\{(\boldsymbol{x}^j,\boldsymbol{y}^j)\}\right)}{\partial w_{si}^{(1)}} = \begin{cases} \varphi'\left(z_s^{(1)}\right)\sum_{l=1}^{r}\delta_l^{(2)}w_{ls}^{(2)} & si\ i=0 \\ x_i\varphi'\left(z_s^{(1)}\right)\sum_{l=1}^{r}\delta_l^{(2)}w_{ls}^{(2)} & sinon \end{cases}$$

On va noter $\delta_s^{(1)} = \varphi'\left(z_s^{(1)}\right)\sum_{l=1}^{r}\delta_l^{(2)}w_{ls}^{(2)}$ le facteur commun pour le $s^{ème}$ perceptron de la couche cachée. L'algorithme d'apprentissage, qui consiste à minimiser la fonction d'erreur en utilisant l'algorithme de descente de gradient, commence par une phase de calcul de la valeur $\delta_l^{(2)}$ pour chaque perceptron de la couche de sortie, suivi du calcul de la valeur $\delta_s^{(1)}$ pour chaque perceptron de



la couche cachée. Ce calcul, appelé backpropagation, s'effectue dans l'ordre inverse de l'algorithme pour calculer les sorties à partir des inputs (commence par les perceptrons de la couche de sortie, suivi des perceptrons de la couche cachée). La backpropagation est illustrée dans la figure suivante :

L'algorithme de rétropropagation du gradient peut être décrit de la manière suivante :

Pour $j = 1..K$

Choisir aléatoirement une image $(\boldsymbol{x}^j, y^j)$ *dans la base*

Pour $l = 1, 2..r$

$$\delta_l^{(2)} = \left(a_l^{(2)} - y_l^j\right) \varphi'\left(z_l^{(2)}\right)$$

Pour $s = 1,2..t$

$$\delta_s^{(1)} = \varphi'\left(z_s^{(1)}\right) \sum_{l=1}^{r} \delta_l^{(2)} w_{ls}^{(2)}$$

$$w_{s0}^{(1)} = w_{s0}^{(1)} - \mu \delta_s^{(1)}$$

Pour $i = 1,2..n$

$$w_{si}^{(1)} = w_{si}^{(1)} - \mu \delta_s^{(1)} x_i$$

Pour $l = 1, 2..r$

$$w_{l0}^{(2)} = w_{l0}^{(2)} - \mu \delta_l^{(2)}$$

Pour $s = 1, 2..r$

$$w_{ls}^{(2)} = w_{ls}^{(2)} - \mu \delta_l^{(2)} a_s^{(1)}$$

La classe *Perceptron* permet de modéliser les perceptrons de la couche de sortie. Il reste à fabriquer une classe pour les perceptrons de la couche cachée, ainsi qu'une classe pour le réseau de neurones avec couche cachée.

a) Classe Perceptron_cachee

La classe *Perceptron_cachee* sert à représenter un perceptron de la couche cachée et descend de la classe *Perceptron*. Il hérite donc des champs correspondant au vecteur de poids et au label de la classe *Perceptron.* La différence avec un perceptron classique se trouve essentiellement au niveau du calcul de $\delta_s^{(1)}$ qui dépend des valeurs $\delta_l^{(2)}$ des perceptrons de la couche cachée. Pour permettre ce calcul, un perceptron de la couche cachée possède un champ supplémentaire qui est un vecteur de pointeurs vers des *Perceptrons* (chaque pointeur pointe vers un des perceptrons de la couche de sortie). Le perceptron de la couche cachée aura accès aux valeurs $\delta_l^{(2)}$ et $w_{ls}^{(2)}$ par l'intermédiaire des fonctions membres *get_delta* via (on supposera que $\delta_l^{(2)}$ a déjà été calculé au préalable) et *get_poids* du perceptron de la couche de sortie correspondant.

Les méthodes de la classe sont :

- Un constructeur qui prend en paramètre un entier correspondant à la taille de l'input, une fonction d'activation (de préférence un pointeur), un *char* correspondant au label du perceptron, ainsi qu'un vecteur (de préférence une référence) de pointeurs vers des *Perceptrons* (les perceptrons de la couche de sortie).
- Une fonction membre *calcul_delta* qui prend en paramètre un *Input* (de préférence par référence), et qui renvoie une valeur réelle (*double*) correspondant à l'évaluation de la valeur $\delta_s^{(1)} = \varphi'\left(z_s^{(1)}\right)\sum_{l=1}^{r}\delta_l^{(2)}w_{ls}^{(2)} = \varphi'\left(w_{s0}^{(1)} + \sum_{i=1}^{n}w_{si}^{(1)}x_i\right)\sum_{l=1}^{r}\delta_l^{(2)}w_{ls}^{(2)}$ pour l'input $\left(\boldsymbol{x}^j, y^j\right)$ donné en paramètre. Cette fonction va également stocker cette valeur dans le champ *delta.*
- Une fonction membre *backprop* qui prend en paramètre un *Input* (de préférence par référence), ainsi qu'une valeur réelle (*double*) correspondant au pas de gradient μ, et qui ne renvoie rien. Cette fonction va apprendre les poids à partir de l'input en paramètre en utilisant la formule :

$$w_{si}^{(1)} = \begin{cases} w_{s0}^{(1)} - \mu\delta_s^{(1)} \; si \; i = 0 \\ w_{si}^{(1)} - \mu\delta_s^{(1)}x_i \quad sinon \end{cases}$$

b) *Classe Input_intermediaire*

La classe *Input_intermediaire* va servir à stocker les valeurs intermédiaires $\boldsymbol{a}^{(1)} = \left(a_1^{(1)}, \ldots, a_t^{(1)}\right)$ à la sortie des perceptrons de la couche cachée. Elle descend de la classe *Input*. Elle possède un champ qui va contenir des valeurs réelles (*double*), qui sera de préférence un vecteur (*vector*).

Les méthodes de la classe sont :

- Un constructeur avec un paramètre de type *char* correspondant au label qui va initialiser le champ label et construire le vecteur
- Une fonction membre *add* qui prend en paramètre un réel (*double*) dont la valeur est ajoutée au vecteur. Cette fonction ne renvoie rien.
- L'opérateur [ ] qui prend en paramètre un entier correspondant à l'indice d'une coordonnée du vecteur et qui renvoie sa valeur (*double*)
- Eventuellement la fonction *get_label* (si elle n'a pas déjà été définie dans la classe *Input*).

c) *Classe NN2*

La classe *NN2* sert à représenter un réseau de neurones avec une couche cachée. Cette classe a pour membre l'ensemble des perceptrons de la couche caché (chacun de la classe *Perceptron_cachee*), ainsi que l'ensemble des perceptrons de la couche de sortie (chacun de la classe *Perceptron*) qui constituent le réseau de neurones.

Les méthodes de la classe sont :

- Un constructeur qui prend en paramètre un entier correspondant à la taille des inputs (par exemple 4 pour les fleurs et 784 pour les images), un entier correspondant au nombre de catégories (et donc le nombre de perceptrons de la couche de sortie), ainsi qu'un entier correspondant au nombre de perceptrons dans la couche cachée. Chaque perceptron de la couche de sortie correspondra à une catégorie, et le label associé au perceptron (celui donné à son constructeur) devra être celui de la catégorie associée. Pour les perceptrons de la couche cachée, la label (qui n'aura pas vraiment le sens de catégorie) correspondra à l'indice du perceptron dans le tableau (ou vecteur).

- Une fonction *evaluation* qui prend en paramètre un *Input* (de préférence une référence), et qui renvoie son label (*char* dont la valeur est comprise entre 0 et $r-1$) évalué en recherchant la plus grande valeur retournée par un des perceptrons de la couche de sortie. L'évaluation des perceptrons de la couche cachée sera effectuée en premier, avec comme entrée l'input. Un *Input_intermédiaire* (correspondant à $\boldsymbol{a}^{(1)} = \left(a_1^{(1)}, \dots, a_t^{(1)}\right)$) sera créé en ajoutant la valeur de chaque perceptron de la couche cachée à cet input intermédiaire. Par la suite, l'input intermédiaire sera donné en entrée des perceptrons de la couche de sortie, et la catégorie retournée sera celle du perceptron de la couche de sortie qui a la plus grande valeur.
- Une fonction *apprentissage* qui prend en paramètre un *Input* (qui correspond à $\left(\boldsymbol{x}^j, y^j\right)$) et un *double* (correspondant au pas de gradient μ) et qui va appliquer l'algorithme d'apprentissage pour cet input. Il suffit d'appliquer l'algorithme décrit plus haut. Dans un premier temps, un input intermédiaire (correspondant à $\boldsymbol{a}^{(1)} = \left(a_1^{(1)}, \dots, a_t^{(1)}\right)$), dont le label est le même que l'input (y^j), est créé (de la même manière que dans la fonction membre *evaluation)*. Les valeurs $\delta_l^{(2)}$ et $\delta_s^{(1)}$ sont ensuite calculées (dans cet ordre) en faisant appel à la fonction membre *calcul_delta* des perceptrons de la couche de sortie (avec comme paramètre l'input intermédiaire correspondant à $\boldsymbol{a}^{(1)} = \left(a_1^{(1)}, \dots, a_t^{(1)}\right)$) et la fonction membre *calcul_delta* des perceptrons de la couche cachée (avec comme paramètre $\boldsymbol{x}^j$). Après cette étape, la fonction *backprop* de chaque perceptron de la couche cachée (avec comme paramètre $\boldsymbol{x}^j$) et la fonction *backprop* de chaque perceptron de la couche de sortie (avec comme paramètre l'input intermédiaire correspondant à $\boldsymbol{a}^{(1)} = \left(a_1^{(1)}, \dots, a_t^{(1)}\right)$) est appliquée pour mettre à jours les poids.

d) Classe Apprentissage

Modifier le patron de classe *Apprentissage* pour qu'il puisse être appelé sur des réseaux de neurones avec couche cachée en ajoutant un paramètre de type supplémentaire qui correspondra à la classe du réseau de neurones (qui pourra être *NN1* ou *NN2*).

e) Tests

Testez vos classes dans la fonction *main* en faisant apprendre à votre réseau de neurones multicouche avec couche cachée à reconnaitre les fleurs et les images. Pour les fleurs, on pourra utiliser un pas de gradient de 0,01 et un nombre d'itérations de 10 000. Pour les images, on pourra utiliser un pas de gradient de 0,1 et un nombre d'itérations de 100 000.