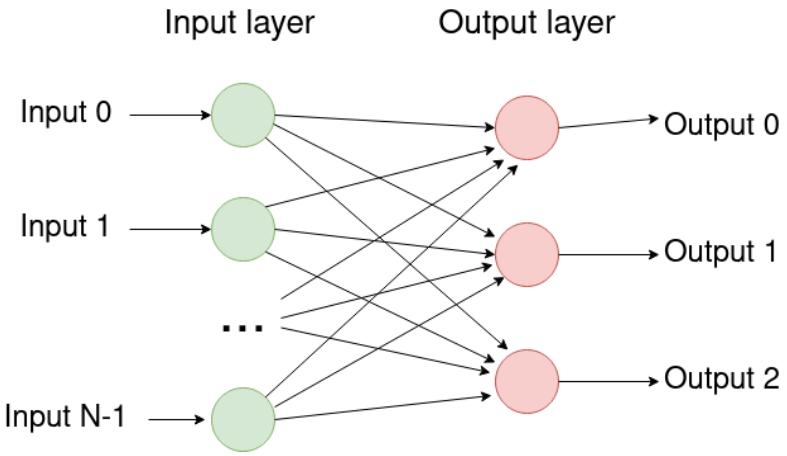
TP NN

## 1. Implémentation d'un réseau de neurones à une couche « from scratch »




Notations :

- Lettres minuscules : scalaires et vecteurs (exemples n, x, z)
- Lettres majuscules : matrices ou parfois scalaire représentant le nombre d'éléments d'un ensemble (exemples N, X, Z)
- "Échantillon" : chaque échantillon représente une entrée, ici une image représentée par un vecteur dimension 784.

### 1.0. Préparation des données

Pour ce TP, vous utiliserez les données (à télécharger sur moodle) suivantes :
- x.npy : données d'apprentissage
- y.npy : labels associés aux données d'apprentissage
- xt.npy : données de test
- yt.npy : labels associés aux données

Ces données représentent des images de chiffres manuscrits de dimensions 28x28 pixels. Les chiffres manuscrits sont de trois classes différentes : 1, 7, 8. Vous pouvez les charger via la fonction *np.load()*.

Les données issues de x.npy et xt.npy se présentent chacune sous la forme d'une matrice de dimensions *(nb_échantillons, 784)*. Chaque ligne correspond à une image. En utilisant la méthode *.reshape( (nb_échantillons, 28, 28)).* Si vous êtes curieux, vous pouvez les visionner en utilisant la librairie *matplotlib*.

Vous trouverez dans le code fourni la fonction *load_data(data_repository, data='digit_3_classes')* prenant en entrée le chemin d'accès vers le répertoire contenant les données, qui les charge, les reformate au besoin et retourne un tuple de quatre matrices *(x_train, y_train, x_test, y_test)*. Les données y_train et y_test sont de dimensions *(nb_échantillons, nb_classes)*. Les '1' manuscrits correspondent à la classe 0, les '7' à la classe 1 et les '8' à la classe 2. Les y_train et y_test sont sous forme 'one hot

encoding', i.e., chaque exemple de la matrice contient un 1 dans la colonne de la classe correspondante, exemple :

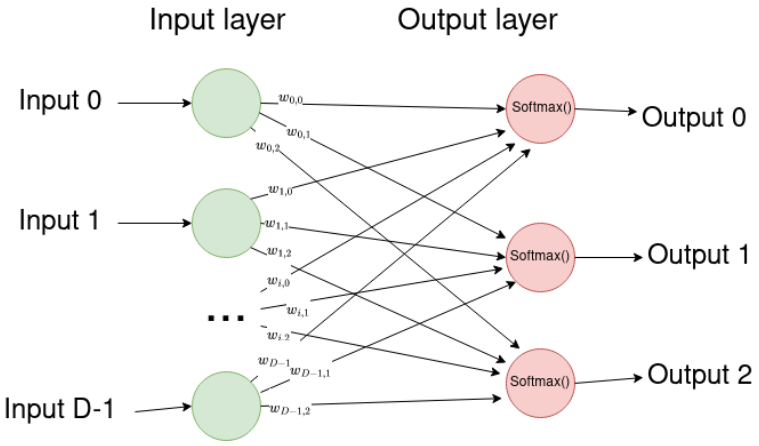
Premier échantillon '7'

ième échantillon : '0'

Dernier échantillon : '8'

$$\begin{bmatrix} 0 & 1 & 0 \\ \dots & \dots & \dots \\ 1 & 0 & 0 \\ \dots & \dots & \dots \\ 0 & 0 & 1 \end{bmatrix}$$

### 1.1. Le réseau de neurones




Nous allons construire un réseau de neurones à une seule couche (l'output layer). Pour ce type de modèle, le nombre de nœuds dépend de la dimension D des entrées.

#### *1.1.1. Fonction de prédiction*

Un réseau de neurones à une seule couche est très similaire à une régression logistique. On cherche à obtenir en sortie de notre réseau de neurones un vecteur de 'probabilités' d'appartenance de chacun des échantillons en entrée à chacune des classes en sortie.

Chaque nœud de la couche de sortie prend en entrée l'ensemble des valeurs des nœuds de la couche d'entrée et calcul une valeur *z* :

$$z_j = \sum_{i=0}^{D} w_{i,j} \times x_i + b_j$$

Avec $x^i$ la ième valeur du vecteur d'entrée, $z_j$, le $j^{ème}$ nœud de la couche de sortie, $w_{i,j}$ le poid associé au $i^{ème}$ nœud de la couche d'entrée par le $j^{ème}$ nœud de la couche de sortie, $b_j$ la valeur de biais associée au $j^{ème}$ nœud de la couche de sortie et D la dimension des échantillons en entrée.

Cependant, cette formule ne garantit pas que la valeur $z_j$ reste comprise entre 0 et 1 ni que la somme des $z_j$ soit égale 1 comme ce serait le cas d'une vraie probabilité. C'est pourquoi on applique au résultat la fonction suivante :

$$\hat{y} = softmax(z_j) = \frac{e^{z_j}}{\sum_{j'=0}^{K} e^{z_{j'}}}$$

Avec *K* le nombre de classes différentes en sortie.

On peut écrire ces équations sous forme matricielle :

$$z = x \cdot W + b \qquad \hat{y} = softmax(z)$$

Où, *z* est le vecteur de dimension *K* contenant chacune des valeurs de *z*, *x* est le vecteur de dimension *D* contenant l'entrée dont on veut déterminer la classe, *W* est la matrice de dimensions *(D, K)* contenant chacun des $w_{i,j}$, *b* est le vecteur des biais de dimension *K* et $\hat{y}$ est le vecteur des probabilités d'appartenance de *x* à chacune des classes.

On peut aussi écrire matriciellement le calcul des probabilités pour plusieurs échantillons simultanément :

$$Z = X \cdot W + b \qquad \hat{Y} = softmax(Z)$$

Avec *Z* la matrice de dimensions *(N, K)* contenant les vecteurs z de chacune des *N* entrées, *X* la matrice de dimensions *(N, D)* contenant chacune des *N* entrées dont on souhaite connaître les classes, $\hat{Y}$ la matrice de dimensions *(N, K)* contenant les probabilités d'appartenance à chacune des classes pour chacun des *N* échantillons.

Les paramètres de notre modèle, c.à.d. ce que l'on cherchera à optimiser afin d'obtenir des probabilités $\hat{Y}$ aussi proches que possible de la réalité sont les valeurs contenues dans *W* et *b*.

**Question 1 :** Dans le cas présent, quelles sont les valeurs de *N*, *K* et *D*, combien y-a-t'il au total de paramètres ?

**Consigne 1 :** initialisez aléatoirement *W* et *b* avec de petites valeurs. Vous utiliserez la librairie numpy, vous pouvez notamment employer la fonction *numpy.random.randn(...)* avec les paramètres appropriés et multiplier le résultat par 1e-3 (10^-3).

**Consigne 2 :** définissez une fonction *predict_one_layer_nn(X, parameters):* prenant entrée les échantillons dont on souhaite déterminer les probabilités d'appartenance à chaque classe et qui retoune $\hat{Y}$. Au besoin, codez la fonction Softmax.

**Question 2 :** que remarquez-vous lorsque vous invoquez la méthode *.sum(axis=1)* sur $\hat{Y}$ ? Pourquoi ?

*1.1.2. Fonction de perte (Loss function) et descente de gradient*

Apprendre les paramètres optimaux de ce réseau signifie trouver les paramètres *W*, *b* qui minimisent les erreurs de classification sur nos données d'entraînement. On utilise une fonction de perte mesurant l'écart entre les résultats obtenus et ceux que l'on aurait dû obtenir. Un choix courant lorsque l'on travaille avec une fonction softmax est d'utiliser à cet effet la fonction d'entropie croisée (cross-entropy) :

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{n \in N} \sum_{k \in K} y_{n,k} \times ln(\hat{y}_{n,k}) + (1 - y_{n,k}) \times ln(1 - \hat{y}_{n,k})$$

**Question 3 :** Vers quoi tend $y_{n,k} \times ln(\hat{y}_{n,k})$ quand :

| $y_{n,k}$ = | $\hat{y}_{n,k}$ tend vers | $y_{n,k} \times ln(\hat{y}_{n,k})$ tend vers |
|---|---|---|
| 0 | 0 | |
| 0 | 1 | |
| 1 | 0 | |
| 1 | 1 | |

**Question 4 :** Vers quoi tend $(1 - y_{n,k}) \times ln(1 - \hat{y}_{n,k})$ quand :

| $y_{n,k}$ = | $\hat{y}_{n,k}$ tend vers | $(1-y_{n,k}) \times ln(1 - \hat{y}_{n,k})$ tend vers |
|---|---|---|
| 0 | 0 | |
| 0 | 1 | |
| 1 | 0 | |
| 1 | 1 | |

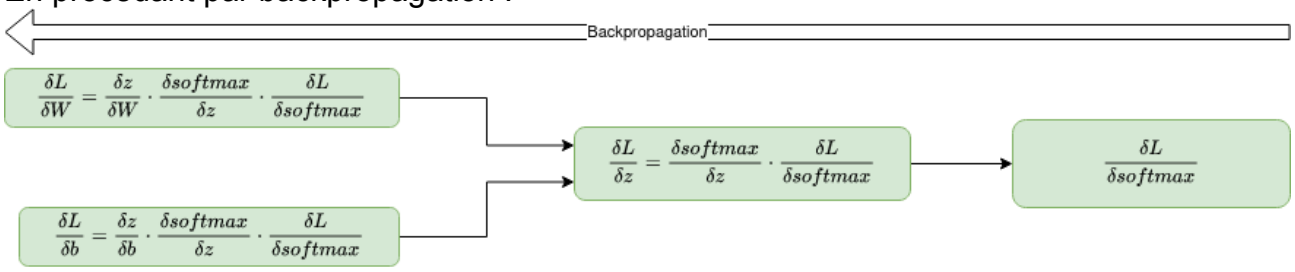
**Question 5 :** Dans quels cas la fonction $L(y, \hat{y})$ tend-t-elle vers 0 ? Qu'est-ce qui la fait décroitre ?

**Consigne 3 :** Définissez une fonction evaluate_loss(y, y_hat): qui calcule et retourne le résultat de la fonction d'entropie croisée. Pensez à ajouter une toute petite valeur (+ 1e-16) dans les log pour éviter les erreurs numériques.

Pour trouver le minimum de cette fonction $L(y, \hat{y})$, nous allons procéder par descente de gradient. Il existe plusieurs variations autour de cet algorithme, la version que nous allons implémenter est une des plus basiques, à savoir la descente de gradient par lot (batch gradient descent) avec un taux d'apprentissage fixe.

Une façon efficace de calculer les dérivées partielles d'une fonction composée est de la représenter sous la forme d'un graphe de calcul et d'appliquer l'algorithme dit de backpropagation [2]. Dans notre cas voici le graph:

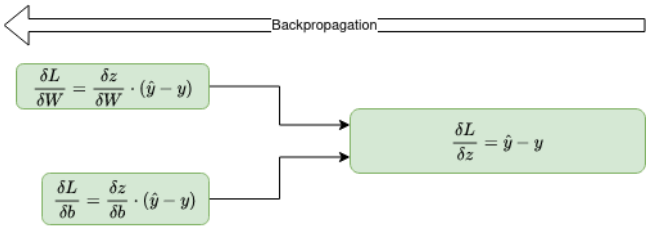
En procédant par backpropagation :




On utilise la fonction softmax pour plusieurs raisons, notamment :
- Elle sort un vecteur dont chaque valeur est comprise entre 0 et 1 et dont la somme vaut 1 ce qui permet de 'simuler' des probabilités d'appartenance à chacune des K classes.
- La fonction softmax utilisée conjointement avec la fonction d'entropie croisée permet cette simplification [1] :

$$\frac{\delta L}{\delta z} = \hat{y} - y$$

Une fois répercuté dans le graph :




(facultatif) **Question 6 :** Calculez *dz / dW* (la réponse est donnée à la fin de cet énoncé)
(facultatif) **Question 7 :** Calculez *dz / db* (la réponse est donnée à la fin de cet énoncé)
(facultatif) **Question 8 :** Calculez *dL / dW* (la réponse est donnée à la fin de cet énoncé)
(facultatif) **Question 9 :** Calculez *dL / db* (la réponse est donnée à la fin de cet énoncé)

**Consigne 4 :** Définissez une fonction *partial_derivative_one_layer_nn(x, y, y_hat, parameters):* dont le rôle est de calculer les dérivées partielles et de les retourner sous la forme d'un tuple *(dW, db)*.

Nous avons maintenant tout le nécessaire pour mettre en œuvre notre descente de gradient. L'algorithme consiste en une simple boucle au cours de laquelle les paramètres W et b sont mis à jour :

$$W = W - e \cdot dW$$
$$b = b - e \cdot db$$

Avec e, le taux d'apprentissage.

**Consigne 5 :** Définissez la fonction *gradient_descent_one_layer_nn(x_train, y_train, parameters, num_passes=200, epsilon=1e-5)*.

**Consigne 6 :** Finalisez votre programme. Chargez vos données, initialisez vos paramètres, optimisez -les avec votre descente de gradient en utilisant les données x_train / y_train, puis effectuez une prédiction sur vos données de test x_test et y_test.

**Consigne 7 :** Vous pouvez invoquer la méthode .argmax(axis=1) sur les données prédites afin d'obtenir pour chaque échantillon le numéro de la classe prédite (celle ayant obtenue la plus grosse probabilité). Quelle est la précision obtenue (ratio nombre_de_prédiction_justes / nombre_d'échantillons) ?

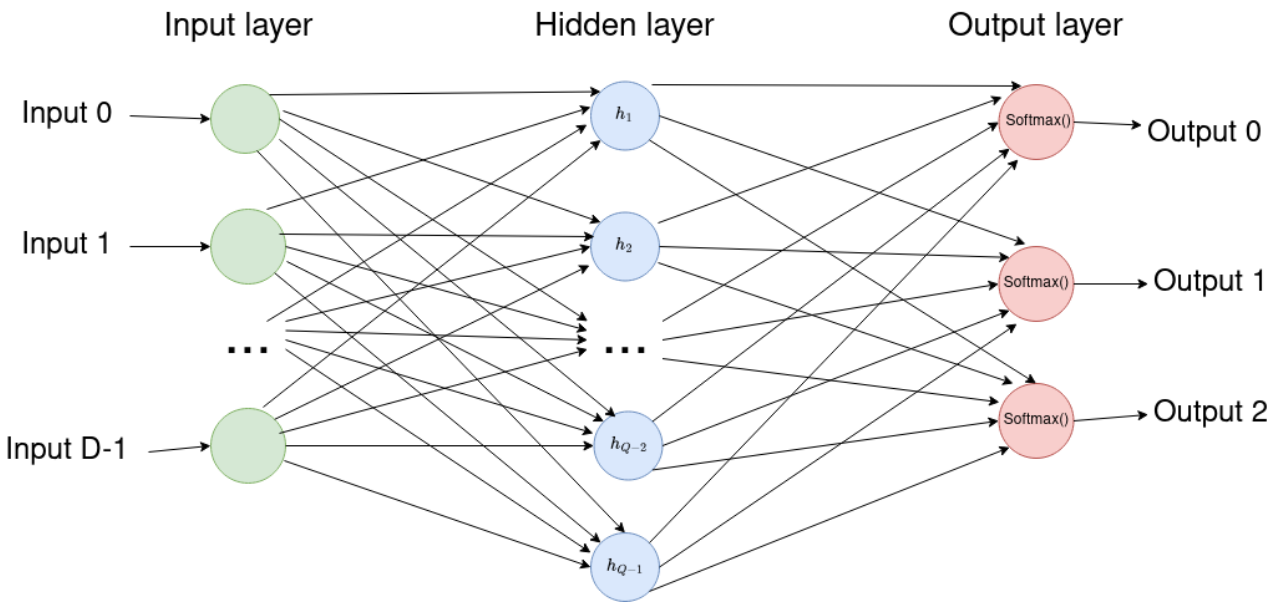
**Consigne 8 :** Enregistrez dans une liste à chaque itération de la décente de gradient la valeur de la fonction de perte calculée. Une fois la descente de gradient accomplie, affichez (matplotlib) la courbe des valeurs de la fonction de perte en fonction des itérations.

**Question 10 :** Comment évolue cette courbe ? Converge-t-elle ? Qu'en pensez vous ?

**Consigne 9** : Ajouter en paramètre de la descente de gradient les données *x_test* et *y_test*, à chaque itération calculer la prédiction y_hat_test pour les données de test puis calculer la valeur de la fonction de perte pour *(y_test, y_hat_test)*, sauvegardez cette valeur dans une liste. Une fois la descente de gradient accomplie, affichez simultanément les courbes de la fonction de perte calculée sur les données d'entraînement et celle calculée sur les données de test.

**Question 11 :** Que pensez-de ces courbes ? Si vous aviez constaté que l'allue de la courbe des données de test est différente de celle des données d'entraînement, qu'en auriez-vous pensé ? (par exemple si la courbe de test se met à croître tandis que celle de train décroît)

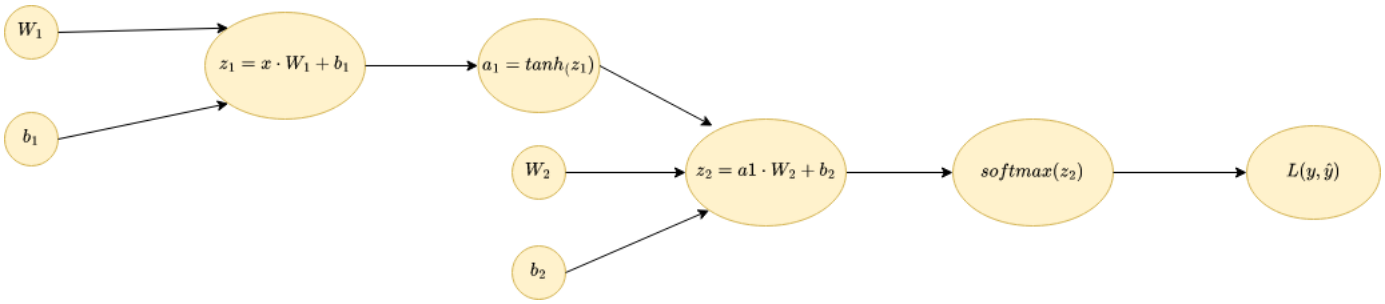
## 2. Implémentation d'un réseau de neurones à deux couches « from scratch »



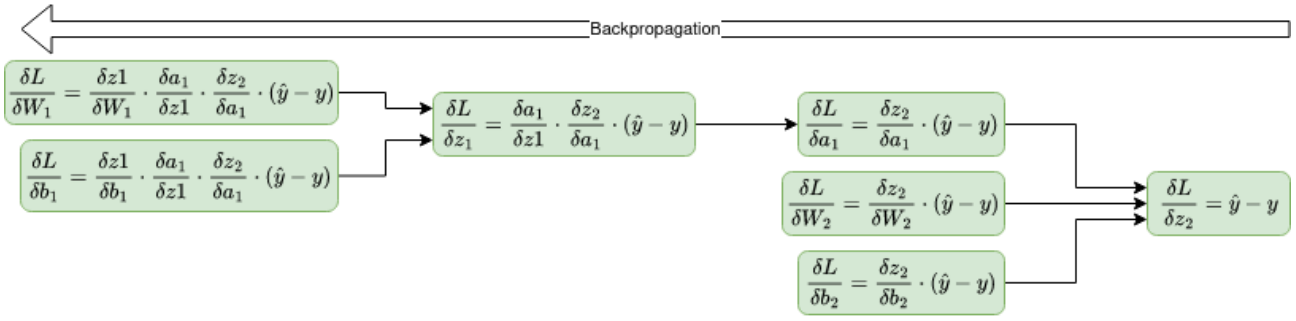

Nous allons maintenant implémenter un réseau à deux couches : une couche de sortie et une couche intermédiaire cachée.

Nous pouvons paramétrer le nombre Q de neurones de la couche cachée suivant la complexité que nous voulons donner à notre modèle. Il faut aussi choisir une fonction d'activation pour cette couche. Les choix les plus communs sont tanh, sigmoid ou relu [3, 4, 5]. Pour cette fois, nous utiliserons la fonction *tanh* dont la dérivée vaut *1-tanh²*.

Voici le graph de calcul de la fonction de perte :




Si nous réutilisons la simplification vue précédemment : $\frac{\delta L}{\delta z_2} = \hat{y} - y$



Déterminons maintenant les dérivées partielles :

$$\frac{\delta L}{\delta W_2} = \underbrace{a_1}_{\frac{\delta z_2}{\delta W_2}=\frac{\delta(a_1 \cdot W_2 + b_2)}{\delta W_2}} \cdot \underbrace{(\hat{y} - y)}_{\frac{\delta L}{\delta softmax}}$$

$$\frac{\delta L}{\delta b_2} = \underbrace{1}_{\frac{\delta z_2}{\delta b_2}=\frac{\delta(a_1 \cdot W_2 + b_2)}{\delta b_2}} \cdot \underbrace{(\hat{y} - y)}_{\frac{\delta L}{\delta softmax}}$$

$$\frac{\delta L}{\delta a_1} = \underbrace{W_2}_{\frac{\delta z_2}{\delta a_1}=\frac{\delta(a_1 \cdot W_2 + b_2)}{\delta a_1}} \cdot \underbrace{(\hat{y} - y)}_{\frac{\delta L}{\delta z_2}}$$

$$\frac{\delta L}{\delta z_1} = \underbrace{(1 - tanh(z_1)^2)}_{\frac{\delta a_1}{\delta z_1}} \cdot \underbrace{W_2}_{\frac{\delta z_2}{\delta a_1}} \cdot \underbrace{(\hat{y} - y)}_{\frac{\delta L}{\delta z_2}}$$

$$\frac{\delta L}{\delta W_1} = \underbrace{x}_{\frac{\delta z_1}{\delta W_1}} \cdot \underbrace{(1 - tanh(z_1)^2)}_{\frac{\delta a_1}{\delta z_1}} \cdot \underbrace{W_2}_{\frac{\delta z_2}{\delta a_1}} \cdot \underbrace{(\hat{y} - y)}_{\frac{\delta L}{\delta z_2}}$$

$$\frac{\delta L}{\delta b_1} = \underbrace{1}_{\frac{\delta z_1}{\delta b_1}} \cdot \underbrace{(1 - tanh(z_1)^2)}_{\frac{\delta a_1}{\delta z_1}} \cdot \underbrace{W_2}_{\frac{\delta z_2}{\delta a_1}} \cdot \underbrace{(\hat{y} - y)}_{\frac{\delta L}{\delta z_2}}$$

**Consigne 10 :** initialisez les paramètres $W_1$, $b_1$, $W_2$ et $b_2$, avec de petites valeurs aléatoires et une dimension de *Q=10* pour la couche cachée.

**Question 12 :** Si $W_1$ est de dimensions *(D, Q)*, $b_1$ de dimensions (1, Q), $W_2$ de dimension *(Q, K)* et $b_2$ de dimensions *(1,K)* avec, pour rappel, *D=784*, *Q=10*, *K=3*, combien cela fait-il de paramètres au total ? Que vaudrait N si les images ne mesuraient pas 28x28 pixels mais 256x256 ? Combien de paramètres seraient nécessaires si nous conservions la même structure de réseau de neurones ?

**Pour aller plus loin :** renseignez-vous sur les réseaux de neurones convolutionnels.

**Consigne 11** : définissez la fonction *predict_two_layers_nn(X, parameters)* prenant entrée les échantillons dont on souhaite déterminer les probabilités d'appartenance à chaque classe et qui retoune $\hat{Y}$.

**Consigne 12 :** Définissez une fonction *partial_derivative_two_layers_nn(x, y, y_hat, parameters):* dont le rôle est de calculer les dérivées partielles et de les retourner sous la forme d'un tuple *(dW1, db1, dW2, db2)*.

Pour optimiser les calculs pensez à factoriser :

$$\delta_3 = \hat{y} - y$$
$$\delta_2 = (1 - tanh(z1)^2) \cdot W_2 \cdot \delta_3$$
$$\frac{\delta L}{\delta W_2} = a_1 \cdot \delta_3$$
$$\frac{\delta L}{\delta b_2} = \delta_3$$
$$\frac{\delta L}{\delta W_1} = x \cdot \delta_2$$
$$\frac{\delta L}{\delta b_1} = \delta_2$$

**Consigne 13 :** Définissez la fonction *gradient_descent_two_layers_nn(x_train, y_train, parameters, hidden_dim=100, num_passes=2000, epsilon=1e-3)*.

**Consigne 14 :** Testez votre réseau de neurones à deux couches. Amusez vous un peu, faites varier les paramètres. Qu'observez-vous au niveau des courbes des fonctions de perte ? Comment évolue la précision ?

### 3. En utilisant torch

Maintenant que vous avez mis en place la partie théorique, nous allons voir au travers de la librairie Torch qu'il existe des outils qui permettent d'automatiser certains calculs fastidieux, en particulier les calculs de dérivées.

Nous allons ré-implémenter le réseau de neurones à deux couches.

**Consigne 15 :** Initialisez les paramètres comme pour le réseau à deux couches précédent puis convertissez les matrices en tenseur torch avec la méthode *torch.tensor*. N'oubliez pas d'indiquer en paramètre *require_grad=True*, c'est ce qui va permettre le calcul des gradients.

**Consigne 16 :** Définissez la fonction *predict_two_layers_nn_torch(x, parameters)*, calculant les 'probabilités' qu'a un échantillon d'appartenir à chacune des classes. Vous pouvez utiliser les fonctions *torch.mm* en lieu et place de *numpy.dot* et *torch.softmax(matrice, axe)*. Vérifiez que pour chaque ligne la somme des probas est bien égale à 1 (vous pouvez utiliser torch.sum).

**Consigne 17 :** Définissez la fonction de perte *evaluate_loss_torch(y, y_hat)*. Vous pouvez utiliser *torch.sum*, *torch.mul* (qui multiplie deux à deux chaque élément de deux matrices <=> *numpy.multiply*) et *torch.log*. Pensez à ajouter une toute petite valeur dans le log ($10^{-16}$) pour éviter les problèmes numériques.

**Consigne 18 :** Nous allons définir la fonction gradient_descent_torch(x_train, y_train, model=None, hidden_dim=100, num_passes=200, epsilon=1e-4). Nous allons procéder par étapes :

1. Comme précédemment, vous écrirez une boucle dans laquelle, vous effectuerez une prédiction (*predict_two_layers_nn_torch*) suivie d'une évaluation de la fonction de perte (*evaluate_loss_torch*) dont vous stockerez le résultat dans une variable *loss*.
2. Vérifier que la variable *loss* est bien de type *torch.Tensor*. Vérifiez aussi que vos différents paramètres ($W_1$, $W_2$, $b_1$, $b_2$) sont bien de type *torch.Tensor*.
3. Essayer de printer l'attribut *.grad* d'une de vos matrices de paramètres,, par exemple *print(W1.grad)*. Appelez sur la variable *loss* la méthode *.backward()*. Puis essayez à nouveau de printer *W1.grad*.
4. Les gradients ont été automatiquement calculés. Vous pouvez mettre à jour vos matrices de paramètres comme précédemment :
   Exemple : *W1 -= epsilon * W1.grad*
5. Puis remettez les gradients à 0, exemple : *W.grad.zero_()*
6. Tester cette nouvelle version de la descente de gradient
7. Tracez vos courbes pour la fonction de perte. Est-ce que 200 itérations suffisent à faire converger l'algorithme ? Quelle précision obtenez-vous sur vos données de test ?
8. Amusez-vous un peu avec vos différents paramètres. Faites varier le nombre d'itération, le taux d'apprentissage epsilon, le nombre de neurones de la couche cachée…
9. Essayez de changer la fonction d'activation de votre couche cachée, par exemple utilisez la fonction torch.relu, testez la avec 200 itérations, 100 neurones sur la couche cachée et un taux d'apprentissage de $10^{-4}$. Comparez les courbes des fonctions de pertes et la précision avec celles obtenues avec la fonction *torch.tanh* toutes choses égales par ailleurs.

Tout ceci est bel et bon mais nous pouvons l'automatiser encore davantage. Nous allons maintenant utiliser plusieurs outils fournis par la librairie torch :

- *th.optim.SGD(mlp.parameters(), lr=epsilon, momentum=0.9)*, qui nous permettra de rendre plus générique le code en optimisant un grand nombre de paramètres automatiquement. En particulier si l'on souhaite augmenter le nombre de couches.
- *torch.nn.Module* est une interface qui vous permettra de définir un réseau de neurones en torch.

**Consigne 19 :** Vous allez maintenant réécrire votre descente de gradient en tirant parti de l'optimiseur torch.optim.SGD.

1. Avant la boucle, initialisez le solver en lui passant les paramètres à optimiser ([W1, b1, W2, b2]) et le learning rate lr (epsilon).
2. Dans la boucle pour effectuer un tour d'optimisation vous pouvez utiliser la méthode .step() de l'optimiseur (il calculera les gradients et mettra à jour les paramètres).
3. Utilisez la méthode .zero_grad() pour remettre à 0 les gradients.
4. Testez.

```python
class MLP(th.nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        # TODO: création des couches et stockage dans self

    def forward(self, x):
        # TODO: calcule des probabilités pour chacune des classes
        return predictions
```

**Consigne 20 :** Écrivez votre classe *MLP* implémentant l'interface *torch.nn.Module*.

1. Modifiez la fonction forward pour qu'elle retourne *x*. Ce n'est pas très utile mais nous permettra d'effectuer un petit test préliminaire. Créez ensuite une instance *mlp = MLP()*. La classe *MLP* est callable car elle hérite de *th.nn.Module* qui l'est, cela signifie qu'elle implémente une méthode *__call__* et que ses instances peuvent être utilisées comme des fonction (c'est la méthode *__call__* qui est appelée dans ce cas). Essayez de l'appeler en lui passant *x_train* en paramètre *mlp(x_train)* et vérifiez qu'elle retourne bien les mêmes valeurs que celles entrées.
2. Dans le constructeur (*__init__*) vous définirez deux couches *self.c1* et *self.c2* de type *torch.nn.Linear*. Pensez à leur donner les bonnes dimensions en paramètre (celles de *W1* et *W2*). Convertissez ces deux couches en double lors de leur *initialisation .double()*.
3. Complétez maintenant la méthode *.forward()* pour qu'elle calcule les probabilités d'appartenance à chacune des classes en utilisant les deux couches *self.c1* et *self.c2* initialisées dans le constructeur ainsi que les fonctions *torch.relu* et *torch.softmax*. Créez une instance de votre classe et vérifiez que les probas somment bien à 1 lorsqu'elle est appelée.
4. Vous pouvez maintenant utiliser cet objet (*mlp*) en lieu dans la descente de gradient en lieu et place de la fonction de prédiction et passer et utiliser ses paramètres (méthode *.parameters()*) dans l'optimiseur *torch.optim.SGD*.
5. Testez.

**Réponses :**

Question 8 :

$$\frac{\delta z}{\delta W} = x$$

Question 9 :

$$\frac{\delta z}{\delta b} = 1$$

Question 10 :

$$\frac{\delta L}{\delta W} = x \cdot (\hat{y} - y)$$

Question 11 :

$$\frac{\delta L}{\delta b} = (\hat{y} - y)$$

**Bibliographie :**

[1] : https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1
[2] : http://colah.github.io/posts/2015-08-Backprop/

Activation functions :
[3] : https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/
[4] : https://www.analyticsvidhya.com/blog/2021/04/activation-functions-and-their-derivatives-a-quick-complete-guide/
[5] : https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

documentation diverse :
[6] : torch : https://pytorch.org/docs/stable/index.html
[7] : matplotlib : https://matplotlib.org/stable/index.html
[8] :