Projet - Réseau de neurones : DIY

L'objectif de ce projet est d'implémenter un réseau de neurones. L'implémentation est inspirée des anciennes versions de `pytorch` (en Lua, avant l'autograd que vous verrez l'année prochaine) et des implémentations analogues qui permettent d'avoir des réseaux génériques très modulaires. Chaque couche du réseau est vu comme un module et un réseau est constitué ainsi d'un ensemble de modules. En particulier, les fonctions d'activation sont aussi considérées comme des modules (cf Figure 1).

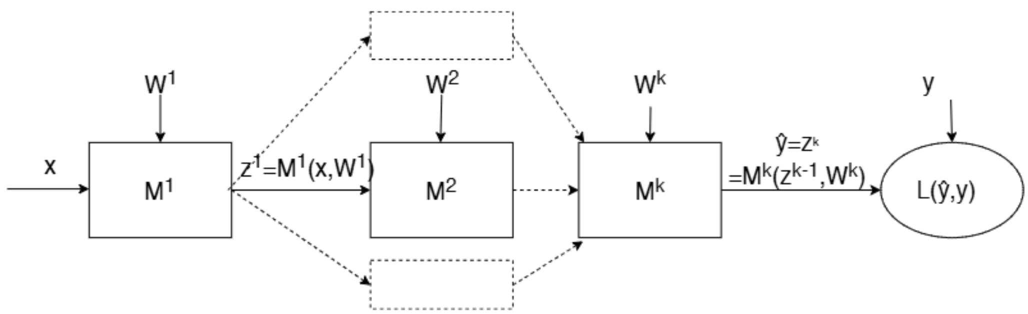

FIGURE 1 – Architecture module d'un réseau

Notons $M^h(\mathbf{z}, \mathbf{W})$ le module de la couche h de paramètre $\mathbf{W}$, $\mathbf{z}^h = M^h(\mathbf{z}^{h-1}, \mathbf{W}^h)$ l'entrée de la couche $h+1$ (ou la sortie de la couche h) et $L(\mathbf{y}, \hat{\mathbf{y}})$ la fonction de coût. Pour pouvoir calculer la mise-à-jour des paramètres de chaque module h, on a besoin de calculer $\nabla_{\mathbf{W}^h} L$. Ce gradient est calculé par rétro-propagation et en utilisant la dérivation en chaîne. Il dépend de deux gradients :

- celui du module par rapport aux paramètres $\nabla_{\mathbf{W}^h} M^h$; ce gradient est calculable sans connaître le reste du réseau, uniquement selon les caractéristiques du module ;
- celui de l'erreur par rapport aux sorties du module $\nabla_{\mathbf{z}^h} L$; ce gradient est assimilable à l'erreur à corriger en rétro-propagation à la sortie du module, et est fourni par l'aval du réseau par induction (les modules $M^{h+1}, M^{h+2}, \ldots$). On note généralement les éléments de ce gradient $\delta_j^h = \frac{\partial L}{\partial z_j^h}$

Ainsi pour un module M^h dont on "aplatit" les poids $\mathbf{W}^h$ en une dimension $(w_1^h, w_2^h, \ldots, w_d^h)$, on obtient les équations :

$$\frac{\partial L}{\partial w_i^h} = \sum_k \frac{\partial L}{\partial z_k^h} \frac{\partial z_k^h}{\partial w_i^h} = \sum_k \delta_k^h \frac{\partial z_k^h}{\partial w_i^h}, \text{ soit } \nabla_{\mathbf{w}^h} L = \begin{pmatrix} \frac{\partial z_1^h}{\partial w_1^h} & \frac{\partial z_2^h}{\partial w_1^h} & \cdots \\ \frac{\partial z_1^h}{\partial w_2^h} & \ddots & \\ \vdots & & \end{pmatrix} \nabla_{\mathbf{z}^h} L \tag{1}$$

$$\delta_j^{h-1} = \frac{\partial L}{\partial z_j^{h-1}} = \sum_k \frac{\partial L}{\partial z_k^h} \frac{\partial z_k^h}{\partial z_j^{h-1}}, \text{ soit } \nabla_{\mathbf{z}^{h-1}} L = \begin{pmatrix} \frac{\partial z_1^h}{z_1^{h-1}} & \frac{\partial z_2^h}{z_1^{h-1}} & \cdots \\ \frac{\partial z_2^h}{z_2^{h-1}} & \ddots & \cdots \\ \vdots & & \end{pmatrix} \nabla_{\mathbf{z}^h} L \tag{2}$$

avec $\frac{\partial z_k^h}{\partial z_i^{h-1}} = \frac{\partial M^h(\mathbf{z}^{h-1}, \mathbf{W}^h)_k}{\partial z_i^{h-1}}$, la dérivée partielle de la k-ème sortie du module par rapport à la i-ème entrée.

Ainsi, pour pouvoir utiliser la back-propagation, il suffit que chaque module puisse calculer sa dérivée par rapport à ses paramètres (utilisée dans l'équation 1) et sa dérivée par rapport à ses entrées (utilisée dans l'équation 2).

Classe (abstraite) Module

Cette section introduit le fonctionnement général de la librairie que vous allez développer. Elle est centrée autour de la classe abstraite `Module` qui représente un module générique du réseau de neurones. Le squelette vous est fourni dans le code source.
La classe module contient :

- une variable `_parameters` qui stocke les paramètres du module lorsqu'il en a (la matrice de poids par exemple pour un module linéaire) ;
- une méthode `forward(data)` qui permet de calculer les sorties du module pour les entrées passées en paramètre ;
- une variable `gradient` qui permet d'accumuler le gradient calculé ;
- une méthode `zero_grad()` qui permet de réinitialiser à 0 le gradient ;
- une méthode `backward_update_gradient(input,delta)` qui permet de calculer le gradient du coût par rapport aux paramètres et l'additionner à la variable `_gradient` - en fonction de l'entrée `input` et des δ de la couche suivante `delta` ;
- une méthode `backward_delta(input,delta)` qui permet de calculer le gradient du coût par rapport aux entrées en fonction de l'entrée `input` et des deltas de la couche suivante `delta` ;
- une méthode `update_parameters(gradient_step)` qui met à jour les paramètres du module selon le gradient accumulé jusqu'à son appel avec un pas de `gradient_step`.

Lorsque plusieurs modules sont mis en série, il suffit ainsi pour la passe forward d'appeler successivement les fonctions `forward` de chaque module avec comme entrée la sortie du précédent. Pour la passe backward, le dernier module calcule le gradient par rapport à ses paramètres et les deltas qu'il doit rétro-propager (à partir des deltas du loss) ; puis en parcourant en sens inverse le réseau, chaque module répète la même opération : le calcul de la mise à jour de son gradient (`backward_update_gradient`) et le delta qu'il doit transmettre à la couche précédente (`backward_delta`).
Remarquez que les paramètres ne sont pas mis tout de suite à jour en fonction du gradient : celui-ci est d'abord accumulé dans la variable `_gradient` et c'est uniquement lors de l'appel explicite à `backward_update_gradient` qui provoque la mise-à-jour des paramètres. Cela rend plus flexible l'utilisation des modules (plusieurs passes de backward peuvent être calculées avant de mettre à jour les paramètres du fait de l'additivité du gradient).
La classe `Loss` est plus simple : elle ne contient que deux méthodes :

- une fonction `forward(y,yhat)` qui permet de calculer le coût en fonction des deux entrées
- une fonction `backward(y,yhat)` qui permet de calculer le gradient du coût par rapport `yhat`.

Tout au long de votre implémentation, vous veillerez à réfléchir précisément à la taille des entrées et des sorties de chaque méthode. Il est conseillé d'utiliser l'instruction `assert condition` pour vous assurer que les paramètres que vous passez en entrée sont de la bonne taille. Par ailleurs, votre implémentation devra pouvoir traiter à chaque fois un `batch` d'exemples et non pas un seul exemple à la fois : ainsi la méthode `forward` d'une module linéaire devra pouvoir prendre en entrée une matrice de taille $batch \times d$ (d la dimension des entrées).

Mon premier est … linéaire !

Pour cette première étape, vous allez coder les deux classes dont vous avez besoin pour réaliser une régression linéaire :

- une fonction de coût `MSELoss` dont la méthode `forward(y,yhat)` doit rendre $\|y-\hat{y}\|^2$; Attention à la généricité de votre implémentation : la supervision `y` et la prédiction `yhat` sont des matrices de taille $batch \times d$ (chaque supervision peut être un vecteur de taille d, pas seulement un scalaire comme dans le cas de la régression univariée). La fonction doit rendre un vecteur de dimension *batch* (le nombre d'exemples).

- un module `Linear(input,output)` qui représente une couche linéaire avec `input` entrées et `output` sorties. La méthode `forward` prend donc une matrice de taille *batch* $\times$ *input* et produit une sortie *batch* $\times$ *output*.

N'oubliez pas de coder toutes les fonctions de ces deux modules ! Une fois l'implémentation réalisée, testez-la sur des données quelconques en réalisant une boucle d'apprentissage par descente de gradient pour optimiser votre premier réseau.

Mon second est ... non-linéaire !

Implémentez le module `TanH` qui permet d'appliquer une tangente hyperbolique aux entrées et le module `Sigmoide` qui permet d'appliquer une sigmoïde aux entrées. N'oubliez pas que les modules de transformation héritent de la classe `Module` et donc doivent implémenter les fonctions `backward_update_gradient`, `backward_delta` et `update_parameters` même si le module n'a pas de paramètre !
Testez votre implémentation en réalisant un réseau à deux couches linéaires avec une activation tangente entre les deux couches et une activation sigmoïde à la sortie. Vous utiliserez des données d'un problème de classification binaire en considérant 1 et 0 comme classes positive et négative.

Mon troisième est un encapsulage

En réalisant le réseau à deux couches précédents, vous remarquez que les opérations de chaînage entre modules sont répétitives lors de la descente de gradient - que ce soit pour la passe forward ou backward - et qu'il sera fastidieux de les écrire pour un grand nombre de modules. Implémenter une classe `Sequentiel` qui permet d'ajouter des modules en série et qui automatise les procédures de forward et backward quel que soit le nombre de modules mis à la suite.
Après l'avoir testé, vous pouvez implémenter une classe `Optim(net,loss,eps)` pour condenser une itération de gradient : elle prend dans son constructeur un réseau `net`, une fonction de coût `loss` et un pas `eps`. Elle contient une seule méthode `step(batch_x,batch_y)` qui calcule la sortie du réseau sur `batch_x`, calcule le coût par rapport aux labels `batch_y`, exécute la passe backward et met à jour les paramètres du réseau.
Vous pouvez également implémenter une fonction `SGD` qui prend en entrée entre autre un réseau, un jeu de données, une taille de batch et un nombre d'itération et s'occupe du découpage en batch du jeu de données et de l'apprentissage du réseau pendant le nombre d'itérations spécifié.

Mon quatrième est multi-classe

Le multi-classe utilise en sortie du réseau une dimension par classe pour dénoter la probabilité de chaque classe. Le vecteur de supervision est un encodage one-hot : un vecteur rempli de 0 sauf à l'index de la bonne classe qui prend la valeur 1. On utilise un Softmax que l'on introduit à la dernière couche pour transformer les entrées en distribution de probabilités grâce à la normalisation effectuée. On peut utiliser la MSE comme coût, cependant elle donne des résultats décevants car elle a tendance à trop "moyenner" les erreurs et ne pas pousser les sorties vers 0 ou 1. Il est mieux d'utiliser des coûts adaptés aux distributions de probabilités comme la cross entropie.
Pour pouvoir faire du multi-classe, nous avons donc besoin :

- d'une transformation `Softmax` qui permet d'appliquer un soft-max aux entrées : $\mathrm{softmax}(\mathbf{z}) = (\frac{e^{z_1}}{\sum_k e^{z_k}}, \frac{e^{z_2}}{\sum_k e^{z_k}}, \ldots)$
- d'un coût cross-entropique : pour y l'indice de la classe à prédire et $\hat{\mathbf{y}}$ le vecteur de prédiction, le coût cross-entropique (équivalent à un maximum de vraisemblance) est $\mathrm{CE}(y, \hat{\mathbf{y}}) = -\hat{\mathbf{y}}_y$;
- il est habituel de combiner les deux ensembles afin d'éviter des instabilités numériques. Dans ce cas, on enchaîne un `Softmax` passé au logarithme (`logSoftMax`) et un coût cross entropique. Le

coût s'écrit alors : $\mathrm{CE}(y, \hat{\mathbf{y}}) = -log \frac{e^{\hat{\mathbf{y}}_y}}{\sum_{i=1}^{K} e^{\hat{\mathbf{y}}_i}} = -\hat{\mathbf{y}}_y + log \sum_{i=1}^{K} e^{\hat{\mathbf{y}}_i}$.
Testez sur le jeu de données des chiffres manuscrits par exemple.

Quelques astuces implémentatoires

- pour transformer un vecteur d'indices `y` de classes en one-hot : `onehot = np.zeros((y.size,10)); onehot[np.arange(y.size),y]=1`
- pour la partie convolutionnelle, vous aurez à utiliser `np.newaxis` qui permet d'ajouter une dimension à un tableau (équivalent à un `.reshape(1, ...)`).
- une initialisation aléatoire des poids entre -1 et 1 des couches convolutionnelles ménera potentiellement à des nombres trop grands en sortie (et donc à un NaN au passage de l'exponentiel). Si cela se produit, modifiez l'initialisation par un facteur 10^{-1} ou plus.
- pour le MaxPool, vous aurez besoin pour le gradient d'affecter des valeurs que aux endroits où le max est atteint (et des zéros partout ailleurs). Une façon de faire est d'utiliser la même astuce que pour le one hot : si `idx` contient les indices de l'argmax sous forme aplatit, soit `res` la matrice résultat de taille `nb_batch,length,chan` :
  `res[np.repeat(range(nb_batch),chan),idx,list(range(chan))*nb_batch]` permet de toucher les indices voulues.
- de manière générale, n'oubliez pas que le gradien est de la même taille que les paramètres et que `backward_delta` prend un delta de même taille que la sortie du `forward` du module.
- essayez de faire le moins de boucles possibles ...