Réseaux de neurones (Neural Network)

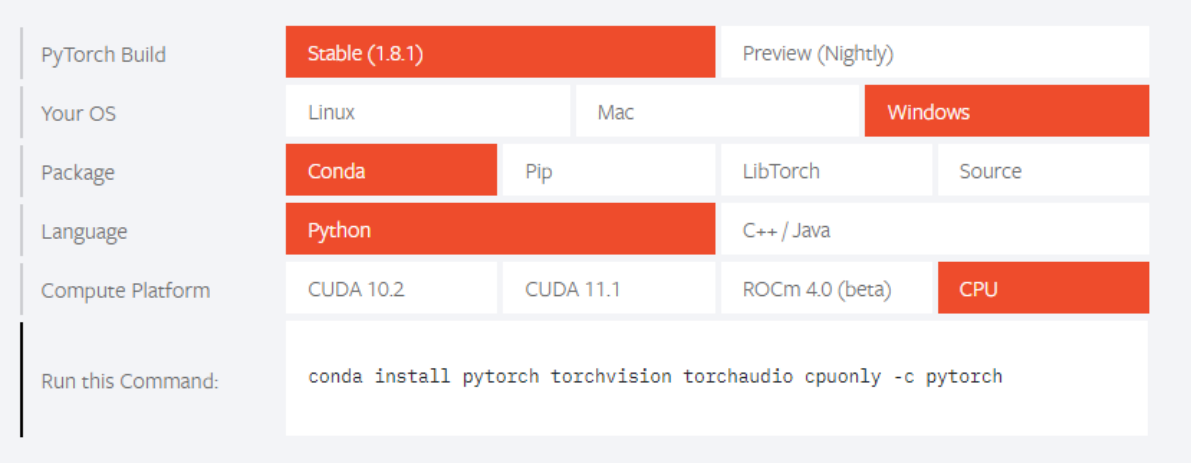
Introduction avec PyTorch

1 Découverte de Pytorch

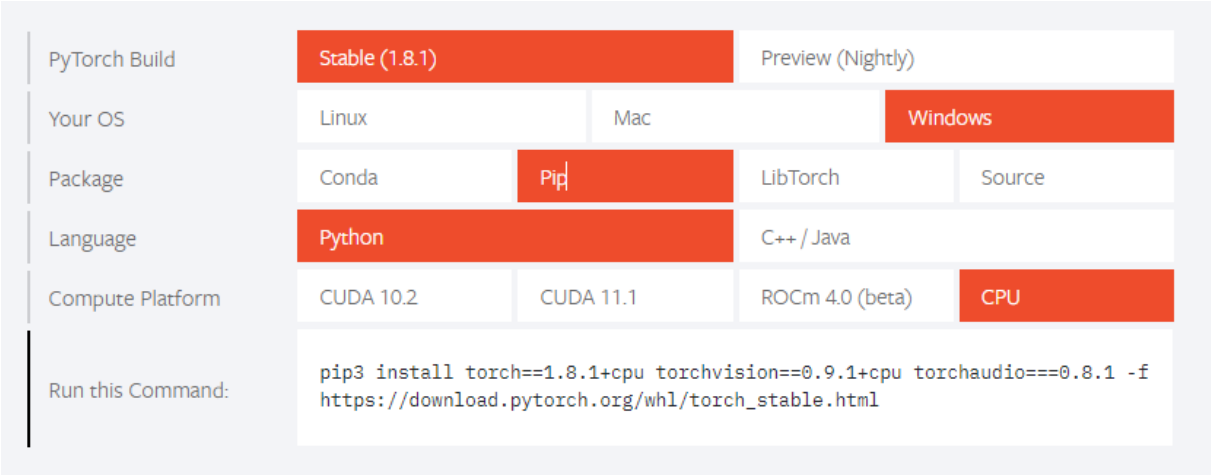
1.1 Installer Pytorch

Pytorch est une librairie maintenue par Facebook et utilisée pour la gestion des réseaux de neurones. Cette librairie offre une grande simplicité d'utilisation, une documentation complète et elle peut gérer CPU & GPU. Pour son installation, on ne peut pas faire un simple pip install. Pour faciliter la vie des programmeurs, une page web interactive a été mise en place pour expliquer la marche à suivre. Rendez-vous donc sur www.pytorch.org. Descendez dans la page pour arriver dans la zone contenant le tableau avec les cases orange. Par défaut vous travaillerez avec la version Stable et non le Night Build. Ensuite vous choisirez votre OS et le langage Python. Nous vous conseillons si vous êtes débutants de n'installer que la version CPU. Les tenseurs sur GPU demandent une gestion plus fine, notamment, en devant indiquer dans Pytorch que tel ou tel tenseur devra être traité sur GPU. Il faudra avoir aussi tous vos driver à jour et compatible avec la version de Pytorch que vous installée.

- Si vous avez installé Anaconda, vous devez cliquer sur la case CONDA. Ci-dessous, vous trouvez les cases cochées pour une install Python/Windows/CPU. Ensuite, ouvrez la fenêtre « Conda Prompt » et copier-coller la ligne en face de « Run this Command » : conda install…



- Si vous avez installé Python, vous devez cliquer sur la case PIP. Ensuite, vous devez ouvrir la fenêtre de commande Windows (cmd.exe) et aller dans le répertoire contenant votre version de Python. Vous taperez alors : python -m pip install… en copiant sa suite du texte en face de « Run this Command ». Cette procédure est nécessaire pour installer Pytorch pour votre installation de Python. Si vous ne faites pas ainsi, l'installation s'effectuera avec la version de Python désignée par votre PATH Windows par défaut, isntallation qui ne correspond pas forcément à celle que vous utilisez pour le projet.



1.2 Conversion entre liste et tableau

Par convention, le terme liste dans ce document désigne une liste Python que l'on obtient par exemple en écrivant : L = [1, 2, 3, 4, 5, 6]. Le terme tableau dans ce document désigne un objet Numpy array que l'on construit par exemple en écrivant : A = np.ones((3,2)). Voici comment convertir chaque objet :

```
A = np.array(L)    # convertit une liste Python en Numpy Array
L = A.tolist()     # convertit un Numpy array en liste Python
```

Pour créer un tableau voici les différentes options :

```
A = np.zeros ((3,2))      # crée un tableau de taille (3,2) rempli de 0
A = np.ones((3,2))        # crée un tableau de taille (3,2) rempli de 1
A = np.empty((3,2))       # crée un tableau sans initialiser les valeurs
```

Ce document n'a pas pour but de présenter les manipulations de listes ou de tableaux. Nous allons principalement nous concentrer sur les tenseurs PyTorch.

1.3 Création et conversion de tenseurs

Pour créer un tenseur, nous utilisons la fonction FloatTensor qui crée par défaut un tenseur en flottant 32 bits stocké dans la RAM de l'ordinateur, pas dans le GPU. On peut initialiser un tenseur indifféremment à partir d'une liste Python ou d'un tableau Numpy :

```
import torch, numpy
ListePython = [[1,2,3], [4,5,6]]
A = torch.FloatTensor(ListePython)
print(A)
ArrayNumpy = numpy.array(ListePython)
B = torch.FloatTensor(ArrayNumpy)
print(B)
```

Ce qui donne, dans les deux cas, le même résultat :

```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

On peut convertir un tenseur vers une liste Python ou un tableau Numpy en utilisant les fonctions tolist() et numpy(). Voici un exemple :

```
import torch
A = torch.FloatTensor([[1,2,3],[4,5,6]])
print(A.tolist())         # => liste python
print(A.numpy())          # => numpy array
```

Dans le cas particulier où le tenseur ne contient qu'une seule valeur, on peut utiliser la fonction item() pour récupérer cette unique valeur. Généralement, la sortie d'un réseau de neurones est un tenseur contenant une seule valeur correspondant au score courant ou à l'erreur courante. Pour voir l'évolution de notre réseau de neurones, à chaque itération, nous affichons cette valeur pour monitorer l'évolution du réseau. La fonction item() peut s'avérer utile dans ce cas :

```
A = torch.FloatTensor([2])
print(A.item()) # => numérique python
=> 2.0
```

1.4 Taille d'un tenseur et convention

Les conventions issues de l'algèbre linéaire, s'appliquent aux tenseurs. Ainsi un tenseur avec 2 lignes et 4 colonnes a une taille égale à (2,4). La propriété shape d'un objet tenseur nous permet de connaître sa taille :

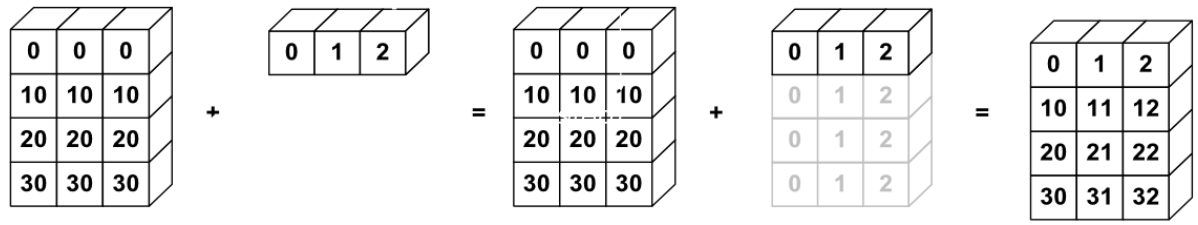
```
import torch
A = torch.FloatTensor([[0,0,0],[10,10,10]])
print(A.shape)
=> torch.Size([2, 3])
```

Si l'on veut nommer nos indices x et y en faisant une correspondance avec les lignes et les colonnes, les indices doivent être écris dans l'ordre inverse :

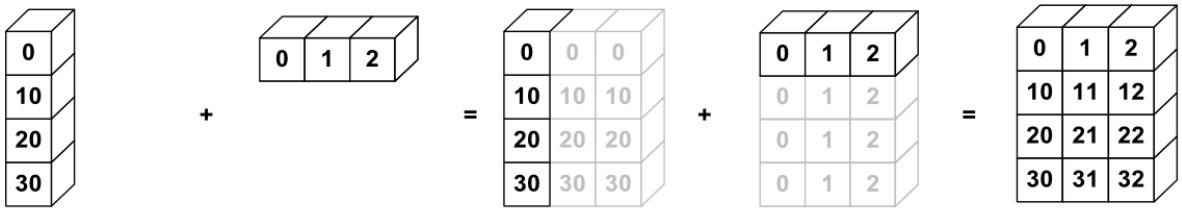
```
A[y][x]
```

1.5 Prise en main des tenseurs

Le terme tenseur (tensor en anglais) n'a aucun rapport avec celui que l'on trouve en physique. Pourquoi avoir choisi ce terme plutôt que celui de matrice ? En fait, les tenseurs sont plutôt des containers de données, ils ne sont pas reliés aux applications linéaires. Ainsi, avec un tenseur on n'effectuera pas une recherche de valeurs propres ou un changement de base. Cependant, les tenseurs héritent des opérations matricielles de base comme l'addition. Mais pas seulement, car les tenseurs permettent d'étendre ces opérations grâce à la méthode du broadcasting. Par exemple, ajouter une matrice à un vecteur ligne est en math impossible. Par contre, pour un tenseur cela a un sens, voici comment on procède :



Ou encore, ce qui est plus perturbant :



1.5.1 Logique du broadcasting

Si intuitivement on comprend comment les opérations se déroulent, voici une manière plus rigoureuse de les expliquer. Si vous ajoutez un tenseur de taille (1,4) avec un tenseur de taille (3,1), tout se comporte comme si chaque tenseur était dupliqué autant de fois que nécessaire pour obtenir deux matrices de taille (3,4). Le résultat obtenu est ainsi un tenseur de taille (3,4).

Voici comment calculer la taille du tenseur résultat. Notons (a_0, a_1, a_2) et (b_0, b_1, b_2) les tailles des deux tenseurs en entrée. Si leurs dimensions sont différentes, on doit compléter par des 1 sur la gauche la taille du tenseur ayant la dimension la plus faible. Finalement, la taille (s_0, s_1, s_2) du tenseur de sortie est donnée par la formule :

$$s_i = max\ (a_i\ ,\ b_i\ )$$

Si les deux valeurs a_i et b_i sont supérieures strictes à 1, alors elles doivent être égales. Dans le cas contraire, nous sommes face à une erreur. Voici quelques exemples pour vous entraîner :

- (3,4) + (1) → (3,4) + (1,1) → (3,4)
- (3,4) + (1,1) → (3,4)
- (3,4) + (4) → (3,4) + (1,4) → (3,4)
- (3,4) + (3) → (3,4) + (1,3) → erreur
- (3,4) + (1,4) → (3,4)
- (4) + (3) → erreur
- (1,1,4) + (1,3) → (1,1,4) + (1,1,3) → erreur
- (1,1,4) + (1,3,1) → (1,3,4)
- (1,1,4) + (3,2,1) → (3,2,4)

Pour tester vos réponses, vous pouvez utiliser la fonction torch.ones((4,3)) pour créer un tenseur de la taille correspondante. La propriété shape vous permet de connaître la taille du tenseur résultat.

Vous pouvez regarder cette page en complément d'information :

https://minitorch.github.io/broadcasting.html

Nous allons maintenant créer des tenseurs et appliquer des opérations de broadcasting.

1.5.2 Cas 1 : tenseur 2D + tenseur 2D

Dans ce cas, les deux tenseurs doivent avoir exactement la même taille. Voici un exemple :

```
import torch
A = torch.FloatTensor([[0,0,0],[10,10,10],[20,20,20],[30,30,30]])
B = torch.FloatTensor([[0,1,2],[0,1,2],[0,1,2],[0,1,2]])
print(A+B)
```

Ce code donne le résultat suivant :

```
tensor([[ 0., 1., 2.],
[10., 11., 12.],
[20., 21., 22.],
[30., 31., 32.]])
```

1.5.3 Cas 2 : tenseur 2D + tenseur 2D colonne

Voici un nouvel exemple :

```
import torch
A = torch.FloatTensor([[0,0,0],[10,10,10],[20,20,20],[30,30,30]])
B = torch.FloatTensor([[1],[2],[3],[4]])
R = A+B
print(R)
```

A ce niveau, Pytorch va créer une boucle implicite pour initialiser le tenseur R :

```
Pour i allant de 0 à nb_colonnes :
      colonne_R[i] = colonne_A[i] + B
```

Le tenseur A a une taille (4,3) et le tenseur B une taille (4,1). Le résultat a donc une taille de (4,3). Voici donc le résultat affiché :

```
tensor([[ 1., 1., 1.],
[12., 12., 12.],
[23., 23., 23.],
[34., 34., 34.]])
```

1.5.4 Cas 3 : tenseur 2D + tenseur 2D ligne

```
import torch
A = torch.FloatTensor([[0,0,0],[10,10,10],[20,20,20],[30,30,30]])
B = torch.FloatTensor([[0,1,2]])
R = A+B
```

```
print(R)
```

A ce niveau, Pytorch va créer une boucle implicite pour initialiser le tenseur R :

```
Pour i allant de 0 à nb_lignes :
      ligne_R[i] = ligne_A[i] + B
```

Le tenseur A a une taille (4,3) et le tenseur B (1,3), le résultat a donc une taille (4,3) :

```
tensor([[ 0., 1., 2.],
[10., 11., 12.],
[20., 21., 22.],
[30., 31., 32.]])
```

Si nous avions écrit :

```
B = torch.FloatTensor([0,1,2])
```

Nous aurions ajouté un tenseur 2D de taille (4,3) à un tenseur 1D de taille (3) ce qui produit un tenseur de taille (4,3). Le résultat est donc équivalent : même taille, mêmes valeurs.

1.5.5 Cas 3 : tenseur 2D + 1 valeur

```
import torch
A = torch.FloatTensor([[0,0,0],[10,10,10],[20,20,20],[30,30,30]])
B = torch.FloatTensor([[7]])  # ou ([7])
R = A + B
print(R)
```

Ici, Pytorch met en place une double boucle pour construire le tenseur R : R[i][j]= A[i][j] + B

Voici le résultat obtenu :

```
tensor([[ 7., 7., 7.],
[17., 17., 17.],
[27., 27., 27.],
[37., 37., 37.]])
```

1.5.6 Cas 4 : tenseur colonne + tenseur ligne

```
import torch
A = torch.FloatTensor([[10],[10],[10],[10]])
B = torch.FloatTensor([[0,1,2]])   # ou FloatTensor([0,1,2])
R = A+B
print(R)
```

Ici, Pytorch met en place une double boucle pour construire le tenseur R : R[y][x]= A[y] + B[x]

Le tenseur A a une taille (4,1) et le tenseur B (1,3), le résultat a donc une taille (4,3) :

```
tensor([[10., 11., 12.],
```

```
[10., 11., 12.],
[10., 11., 12.],
[10., 11., 12.]])
```

Il est légitime de se demander ce qu'il se passe si l'on ajoute un vecteur ligne à un vecteur colonne :

```
R = B+A
print(R)
```

Le résultat associé est identique.

1.5.7 Autres cas

Nous ne pouvons pas tester toutes les configurations car cette mécanique s'étend aisément aux dimensions supérieures. Par exemple, en ajoutons un tenseur 2D à un tenseur 3D correspondant à un vecteur colonne, nous obtenons une matrice 3D :

```
import torch
A = torch.FloatTensor([[0,0,0],[10,10,10],[20,20,20]])
B = torch.FloatTensor([[[4]],[[5]],[[6]]])
print(A+B)
```

Examinez bien les crochets dans le tenseur B, cela nous indique que pour les deux premières dimensions x/y, il n'y a qu'une seule valeur et que ces valeurs 4, 5 et 6 correspondent ainsi aux « étages » 0, 1 et 2 de la dimension z. Le tenseur A a une taille (3,3) et le tenseur B une taille (3,1,1), le résultat obtenu a donc une taille (3,3,3) :

```
tensor([[[ 4., 4., 4.],
[14., 14., 14.],
[24., 24., 24.]],

[[ 5., 5., 5.],
[15., 15., 15.],
[25., 25., 25.]],

[[ 6., 6., 6.],
[16., 16., 16.],
[26., 26., 26.]]])
```

Il faut ici faire attention aux dimensions. Si pour le tenseur B nous oublions par erreur une paire de crochets, nous aurions :

```
import torch
A = torch.FloatTensor([[0,0,0],[10,10,10],[20,20,20]])
B = torch.FloatTensor([[4],[5],[6]])
print(A+B)
```

Avant cette modification, nous ajoutions un tenseur de taille (3,3) à un tenseur de taille (1,1,3) ce qui par les règles de broadcasting donnait un tenseur 3D de taille 3D (3,3,3). En oubliant la paire de

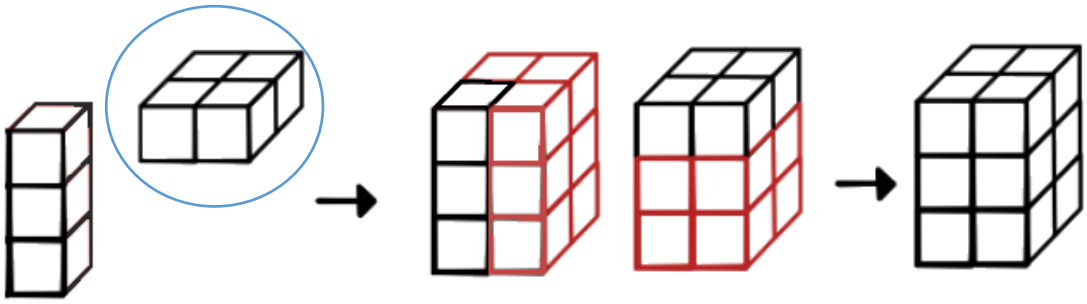
crochets, le tenseur B a maintenant une taille (1,3) ce qui donne par broadcasting avec un tenseur (3,3) un nouveau tenseur de taille (3,3) :

```
tensor([[ 4., 4., 4.],
[15., 15., 15.],
[26., 26., 26.]])
```

Dans ce cas, l'oubli d'une paire de crochets a totalement changé le résultat contrairement aux exemples précédents. Il faut donc vraiment faire attention, car tout oubli ne provoque pas forcément une erreur lorsqu'on manipule des tenseurs.

1.5.8 Schéma sous forme de cubes

On trouve ce type de schéma sur les tutoriels internet. Il existe une ambiguïté sur les dimensions comme dans cet exemple :



Le tenseur 2D entouré en bleu est un tenseur 2D de dimension (2,2). Il est représenté couché en travers. Le vecteur colonne sur la gauche n'est pas un tenseur 1D contrairement à ce que l'on pourrait penser. En effet, dans cette configuration nous serions dans le cas tenseur 2D (2,2) + 1 tenseur 1D (3) ce qui donnerait une erreur. Le tenseur sur la gauche correspond donc à un tenseur 3D de dimension (3,1,1) ce qui donne après broadcasting un tenseur de taille (2,2,3).

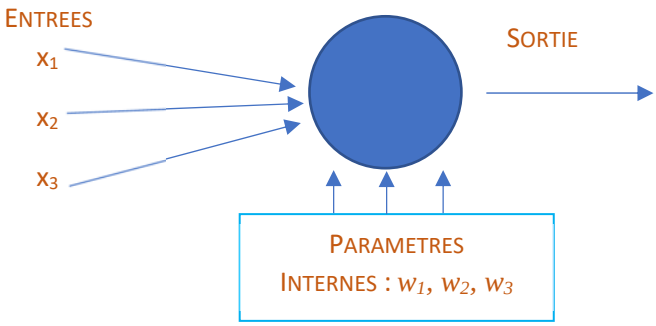
1.5.9 Conclusion

Du point de vue du développeur, la souplesse et l'adaptabilité des tenseurs doit amener une certaine attention lorsqu'on les manipule. En effet, autant le calcul matriciel est strict et sans ambiguïtés, autant le calcul avec des tenseurs est très permissif.

En effet, lorsque vous allez vous trompez en combinant deux tenseurs, une fois sur deux, aucune erreur ne sera signalée, car Pytorch aura fait quelque chose : le résultat obtenu sera par exemple un tenseur 3D alors que vous auriez dû obtenir un tenseur 2D, oups ! L'accident est vite arrivé et en toute discrétion. Il faudra ainsi souvent vérifier les dimensions de vos tenseurs durant les calculs, car les surprises sont parfois au rdv !

2 Introduction

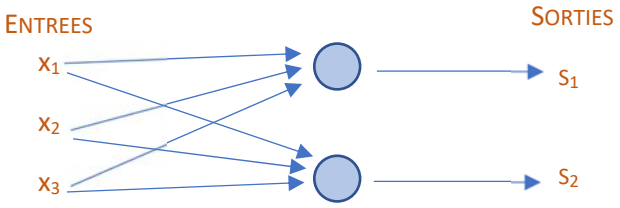
2.1 Le neurone informatique



Voici le schéma d'un neurone informatique. Il compte plusieurs entrées pour une UNIQUE sortie. Les entrées sont des valeurs flottantes notées x_i. Ses paramètres internes sont appelés poids, en anglais weights, ce qui donne la notation w_i. On peut parfois rajouter un paramètre optionnel à la sortie appelé biais et noté b. Dans tout le réseau, chaque neurone effectue le même type d'opération que l'on peut écrire de la manière suivante :

$$Sortie = \sum_{i=0}^{n-1} x_i * w_i + b$$

La sortie est donc un nombre flottant. Que se passe-t-il lorsque l'on met deux neurones dans la même couche ? Le deuxième neurone aura exactement les mêmes entrées donc le même nombre de poids. Il produira, comme les autres neurones, sa propre valeur de sortie. Voici le schéma correspondant :



Ce type de couche est appelée Fully Connected (car chaque entrée est connectée à chaque neurone) ou encore Linear. En consultant la documentation PyTorch, nous trouvons :

LINEAR

CLASS `torch.nn.Linear(in_features, out_features, bias=True)`

Applies a linear transformation to the incoming data: $y = xA^T + b$

This module supports TensorFloat32.

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

Vous remarquerez que la description d'une couche Linear ne fait pas mention de neurones ! Dans la documentation, on cite : la taille de l'entrée et la taille de la sortie ainsi qu'un booléen autorisant la présence d'un biais ou non. Cependant, nous savons que la taille de la sortie correspond aux nombres de neurones. Si le tenseur d'entrée est de taille 4 et si nous voulons une couche contenant deux neurones, nous créons donc une couche Linear en donnant comme paramètres : in_features égal à 4 et out_features égal à 2.

Variables

- **~Linear.weight** – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **~Linear.bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

Les poids et si besoin les biais des neurones sont initialisés aléatoirement à partir d'une loi uniforme dont l'intervalle est défini en fonction du nombre d'entrées.

Nous nous intéressons plus en détail à la formule : $y = xA^T + b$ qui décrit le calcul mise en place par la couche Linear. On pourrait croire que les tenseurs x et b étaient des tenseurs colonnes. Cependant, le placement à gauche dans la multiplication matricielle nous indique qu'il s'agit en fait d'un tenseur ligne ainsi que le tenseur de biais b et le tenseur résultat y. Dans cette configuration, la matrice A^T à droite de x doit avoir une taille égale à : *(nombre d'entrées, nombre de sorties)* pour que la multiplication matricielle opère. Comme le tenseur A est transposée, sa taille doit donc être égale à : *(nombre de sorties, nombre d'entrées)*.

Voici donc ce que nous obtenons lorsque nous avons *n_out* neurones et *n_in* valeurs d'entrée :

- Nous avons *n_out* valeurs de sortie
- Nous avons *n_out* $\times$ *n_in* poids dans cette couche
- Le tenseur de sortie y et le tenseur de biais b sont de taille *(n_out)*
- Le tenseur d'entrée x est de taille *(n_in)*
- Le tenseur A est de taille *(n_out, n_in)*

Voici un code qui va nous permettre de tester les paramètres internes de la couche Linear :

```
import torch
layer = torch.nn.Linear(in_features = 4, out_features = 2)
print("Poids :", layer.weight, layer.weight.shape)
print("Biais :", layer.bias, layer.bias.shape)
Input = torch.FloatTensor([1,2,3,4])
Sortie = layer(Input)
print("Sortie :", Sortie, Sortie.shape)
```

Nous vous conseillons de tester ce code. Voici les résultats obtenus, avec $n=2$ et $m=4$:

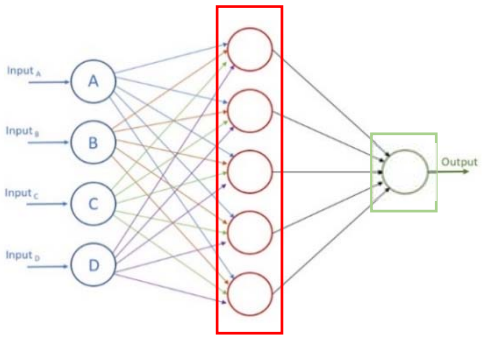
```
Poids :
      tensor([[-0.1733, 0.2653, -0.1198, -0.0571],
              [-0.2377, 0.4335, -0.2727, 0.1650]],…)
     torch.Size([2, 4])
```

```
Biais :
      tensor([0.1651, 0.3105], ..)
      torch.Size([2])
Sortie :
     tensor([-0.0654, 0.7820], …)
     torch.Size([2])
```

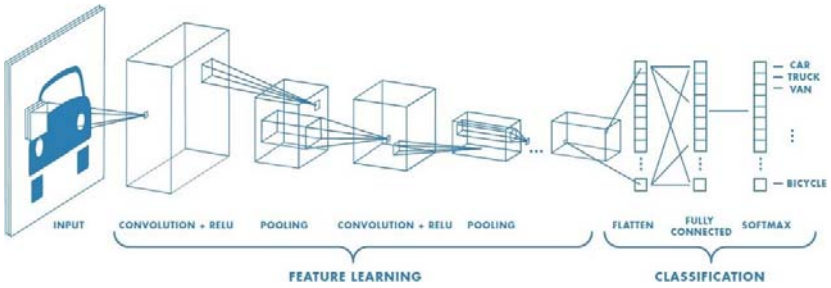
2.2 Couches et réseaux

Une couche (layer) de neurones peut être représentée à partir d'un rectangle vertical, contenant des ronds correspondant aux neurones, lorsqu'il n'y en a pas trop en tout cas ! Les entrées, peuvent être aussi regroupées dans une couche d'entrée appelée « input layer » même si aucun neurone n'est présent. Cependant, l'entrée ayant pour fonction de « charger » les données, cette couche a donc un rôle « actif » et peut être considérée comme une couche à part entière même si elle n'effectue aucun calcul.

L'empilement de plusieurs couches permet de construire un réseau. Les entrées sont généralement situées sur la gauche, les couches successives sont empilées de gauche à droite et la sortie se situe à l'extrémité droite :



Un petit réseau : 4 valeurs d'entrée, puis une couche Linear au centre avec 5 neurones et une deuxième couche Linear à droite avec un seul neurone donnant une valeur de sortie unique.



Un réseau complexe : les entrées correspondent à plusieurs images.
Chaque sortie d'une couche est symbolisée par un volume pour décrire la taille de son tenseur.
Chaque couche porte un nom : CONV / POOL / FLATTEN / FULLY CONNECTED.
Lorsque qu'une fonction d'activation est présente, son nom est donné : RELU.

2.3 La fonction d'activation

Suffit-il d'empiler des couches pour construire un réseau ? Pas si sûr. En effet, prenons le cas de trois couches Linear sans biais. En utilisant la formule associée, nous obtenons :

$$Out_{Réseau} = Input.{}^tA_1.{}^tA_2.{}^tA_3$$

Où chaque tenseur A_i correspond à une couche du réseau. Ce réseau à trois couches équivaut donc à un réseau ayant une seule couche. En effet, il suffit de créer une matrice $M = A_3.A_2.A_1$ ce qui nous permet d'écrire :

$$Out_{Réseau} = Input.{}^tM$$

Pour avoir un réseau à plusieurs couches, il faut donc ajouter entre chaque couche une « non-linéarité », c'est le rôle de la fonction d'activation. Cette fonction est une fonction de $\mathcal{R} \rightarrow \mathcal{R}$ non-linéaire qui se branche à la sortie de chaque neurone. Dans une même couche, tous les neurones utilisent la même fonction d'activation. Les fonctions d'activation n'ont pas de poids internes. Une fonction assez populaire aujourd'hui est la fonction ReLU définie ainsi :

$$ReLU(x) = max(0,x)$$

Il s'agit d'une fonction continue, dérivable quasiment partout et peu couteuse en calcul ! Dans les schémas des réseaux de neurones, vous trouverez ainsi des commentaires sous chaque couche du type LINEAR+RELU pour indiquer le type de couche ainsi que sa fonction d'activation.

On peut trouver d'autres fonctions définies de $\mathcal{R}$ dans $\mathcal{R}$ comme *tanh()* dont les valeurs de sortie sont entre -1 et 1 ou encore la fonction sigmoïde $\sigma(x)$ dont les valeurs de sortie sont entre 0 et 1.

2.4 Créer son premier neurone

Nous allons maintenant utiliser une couche Linear avec un seul neurone. Pour cela, nous avons deux étapes :

- En premier, nous instancions une couche de type Linear nommé *layer* et nous lui donnons ses caractéristiques internes.
- Ensuite, nous utilisons cet objet layer comme une fonction en lui donnant un tenseur d'entrée. L'objet layer calcule les valeurs de sortie et retourne un tenseur de sortie.

```
import torch
layer = torch.nn.Linear(1,1)
input = torch.FloatTensor([4])
output = layer(input)
print(output)
```

Ce qui donne en sortie un tenseur de taille (1) :

```
tensor([-2.5396], …)
```

2.5 Affichage du graph associé à un neurone+Relu

Nous allons tracer la fonction associée à un neurone avec une fonction d'activation de type Relu.

La fonction linspace() de numpy sert à échantillonner avec un pas régulier un certain nombre de valeurs entre deux bornes. Ainsi linspace(-2,2,5) donne le tableau suivant : [-2,-1,0,1,2]. Pour l'affichage, nous allons utiliser la librairie matplotlib et plus particulièrement les fonctions de tracé issues de pyplot (plt). La fonction plt.plot() prend en paramètres une liste d'abscisses x_i et une liste d'ordonnée y_i et trace l'ensemble des points aux coordonnées (x_i,y_i). La fonction plt.axis('equal') force les échelles d'affichage des deux axes à être égales et pour terminer la fonction plt.show() crée la fenêtre d'affichage à l'écran et lance un rendu.

Pour tracer le graph de fonction associé à un neurone, une première approche consiste à faire une boucle pour calculer chaque valeur de sortie. Examinons le code :

```
import torch, numpy, matplotlib.pyplot as plt


layer = torch.nn.Linear(1,1)    # creation de la couche Linear
activ = torch.nn.ReLU()         # fonction d'activation Relu
Lx = numpy.linspace(-2,2,50)    # échantillonnage de 50 valeurs dans [-2,2]
Ly = []
for x in Lx:
      input = torch.FloatTensor([x])  # création d'un tenseur de taille 1
      v1 = layer(input)               # utilisation du neurone
      v2 = activ(v1)                  # application de la fnt activation ReLU
      Ly.append(v2.item())            # on stocke le résultat dans la liste
plt.plot(Lx,Ly,'.')       # dessine un ensemble de points
plt.axis('equal')         # repère orthonormé
plt.show()                # ouvre la fenetre d'affichage
```

Vous remarquez que nous devons d'abord créer la couche avant de l'utiliser. Ainsi la ligne :

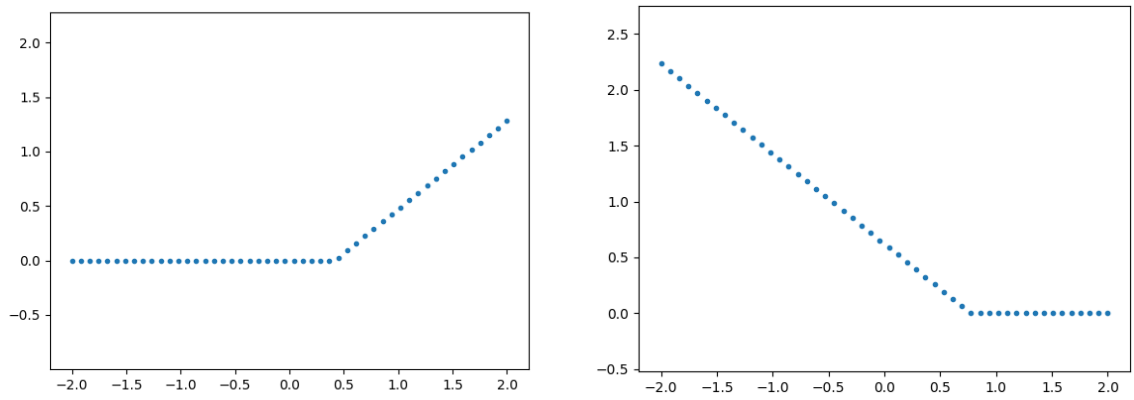
```
layer = torch.nn.Linear(1,1)
```

sert à créer un objet *layer* correspondant à une couche de neurones. De la même manière, nous créons un objet *activ* qui nous permettra d'appliquer la fonction *ReLU* : activ = torch.nn.ReLU(). Cette approche peut vous paraître étrange car il suffirait normalement d'appliquer les fonctions Linear ou ReLU pour calculer les résultats de chaque couche ! Pourtant, nous créons des objets-foncteurs aussi appelés foncteurs. Ces objets remplissent le même rôle qu'une fonction. L'utilisation des objets foncteurs a le désavantage de doubler le nombre de lignes de code car il y a une création puis un appel. Pourquoi ce choix ? Cela vient du fait que PyTorch, ou tout autre librairie de deep learning, doit stocker des informations supplémentaires permettant notamment de calculer plus tard le gradient des différentes couches. Pour cela, de nombreuses informations doivent être conservées d'un calcul à l'autre et ces objets-foncteurs vont nous permettre de stocker dans leurs paramètres internes ces informations intermédiaires sans alourdir le code.

Résumons la chaîne de calcul. Une fois le tenseur *input* initialisé, nous le passons en paramètre à la première couche en écrivant : *v1 = layer(x)* ; le résultat est ainsi stocké dans le tenseur *v1*. Nous écrivons ensuite : *v2 = activ(v1)* pour appliquer la fonction d'activation *Relu* sur le tenseur *v1* et créer le tenseur *v2*. La valeur de l'ordonnée *y* est récupérée dans le tenseur *v2* en utilisant la fonction item().

Voici des exemples du résultats associée à 1 neurone avec une fonction d'activation ReLU. Avec une seule valeur en entrée, le neurone effectue le calcul $y=a.x+b$ qui correspond graphiquement à une

droite. En appliquant ensuite la fonction Relu(), cela a pour effet de supprimer toutes les valeurs négatives. Nous et de créer ces formes de droite coupée :



Visualisation du signal de sortie d'un neurone avec une fonction d'activation ReLU

2.6 Utilisation de la fonction reshape

Dans la partie précédente, nous avons tracé le signal de sortie d'un neurone, cependant pour cela, nous avons calculé les points de tracé un à un. Comme la couche Linear effectue une multiplication matricielle $y = xA^T + b$, nous pouvons utiliser une astuce pour calculer toutes les valeurs y_i en un seul appel. Pour cela, nous allons transformer le tenseur *Lx* de taille (50) en un tenseur de taille (50,1). Cette taille est compatible avec la multiplication matricielle par un tenseur de taille (1). Le résultat obtenu sera donc un tenseur *Ly* de taille (50,1) contenant les 50 valeurs d'ordonnées. Une nouvel appel à la fonction reshape() permet de remettre ce tenseur sous la forme d'un tenseur de taille (50).

La fonction reshape() ne déplace pas de valeurs en mémoire. En fait, les valeurs restent à leur place, seule la taille du tenseur est modifiée. Comment cela est-il possible ? Il suffit que la taille de départ et la taille recherchée soient compatibles, c'est-à-dire que les deux teneurs contiennent le même nombre d'éléments. Il n'y a donc aucune modification à apporter sur les valeurs stockées en mémoire. Par exemple, si en mémoire nous avons les 6 valeurs [1,2,3,4,5,6], nous pouvons grâce à la fonction reshape() modifier la taille du tenseur pour :

- Une taille (6) : [1,2,3,4,5,6]
- Une taille (3,2) : [[1,2], [3,4], [5,6]]
- Une taille (2,3) : [[1,2,3], [4,5,6]]
- Une taille (2,1,3) : [[[1,2,3]], [[4,5,6]]]
- Une taille (6,1,1) : [[[1]], [[2]], [[3]], [[4]], [[5]], [[6]]]
- Une taille (1,1,6) : [[[1,2,3,4,5,6]]]

La fonction detach() est utilisée pour extraire un tenseur en dehors du réseau. Le tenseur est donc copié et désassocié des autres couches puis converti en tableau Numpy en utilisant la fonction numpy(). Nous passons les deux tableaux d'abscisses et d'ordonnées à la fonction plot() qui effectue le rendu.

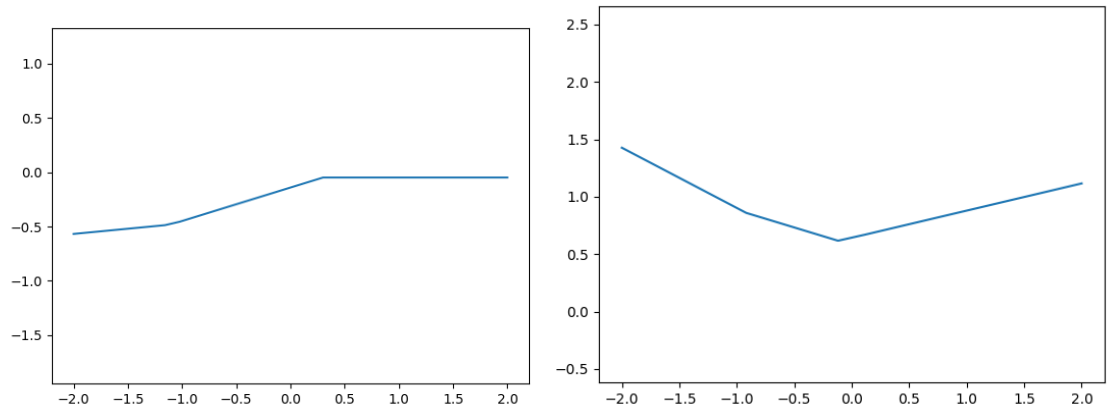
```
import torch, numpy, matplotlib.pyplot as plt


layer = torch.nn.Linear(1,1)
relu = torch.nn.ReLU()
```

```
Lx = numpy.linspace(-2,2,50)
Lx = torch.FloatTensor(Lx)      # crée un tenseur de taille 50
Lx = Lx.reshape(50,1)           # change la taille du tenseur pour (50,1)
Ly = layer(Lx)                  # calcule pour chaque xi la sortie du neurone
Ly = relu(Ly)                   # applique la fonction d'activation
Ly = Ly.detach()                # extrait le tenseur du réseau
Ly = Ly.numpy()                 # conversion vers un tableau numpy
…
```

2.7 Exercice 1 : mise en place d'un réseau à deux couches

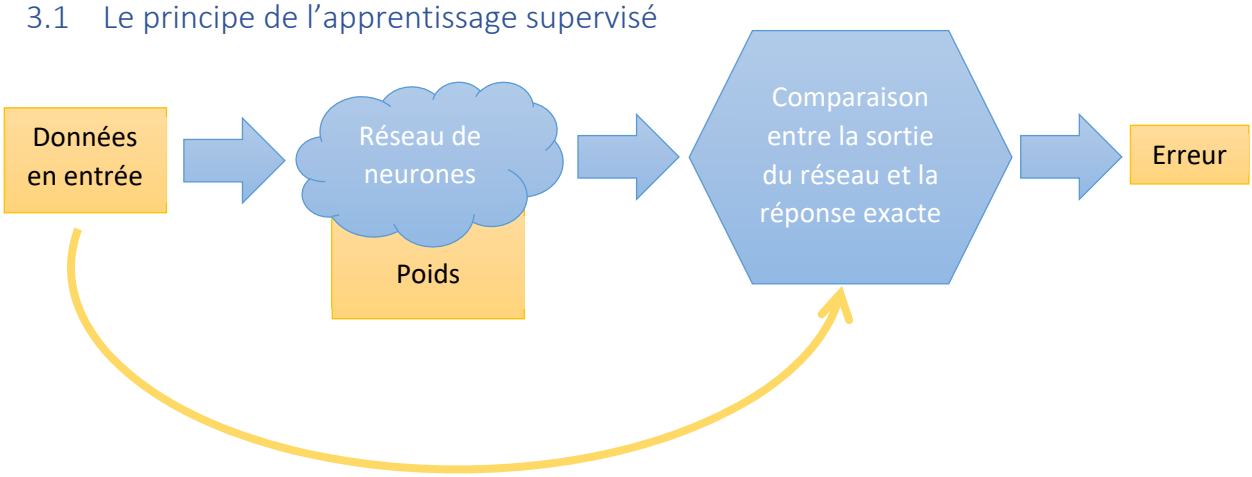
Comme dans la partie précédente, vous devez construire un réseau avec deux couches et tracer le graphique associé. La première couche va contenir 3 neurones et la deuxième un neurone unique. La fonction d'activation en sortie de la première couche sera une ReLU. Complétez le programme Ex01.py.



Vous allez obtenir des tracés correspondant à des fonctions linéaires par morceaux. Cela est normal, car la sortie d'un neurone + ReLU est une fonction linéaire par morceau. Ainsi la combinaison linéaire de plusieurs fonctions linéaires par morceaux est une fonction linéaire par morceaux. De manière pragmatique, le tracé d'un neurone + ReLU permet de créer une discontinuité (une cassure). Avec trois neurones sur la première couche, nous pouvons donc avoir un maximum de trois cassures sur la courbe finale, ce que nous constatons sur nos tracés.

3 L'apprentissage

3.1 Le principe de l'apprentissage supervisé



Nous disposons de données sous forme de tenseurs (une image, un son, un texte…) et nous voulons que notre réseau calcule lorsqu'on lui présente un échantillon la réponse idéale : un score, une température, une probabilité, un âge… Dans l'apprentissage supervisée, nous connaissons la réponse exacte associée à chaque échantillon. En comparant la sortie du réseau de neurones et la réponse attendue, on peut alors mesurer l'équivalent d'une erreur sous la forme d'un nombre réel.

Un tel système peut apprendre ! Pourquoi ? Les données en entrée sont représentées par des tenseurs constants car ces données ne changent pas et restent fixes pendant toute la phase d'apprentissage. Seuls les poids du réseau peuvent varier. On peut ainsi modéliser le réseau comme une fonction N de $\mathcal{R}^n \rightarrow \mathcal{R}$ qui associe à un ensemble de poids donnés un nombre réel correspondant à l'erreur.

Plus l'erreur est importante, plus la réponse du réseau s'est éloignée de la réponse idéale. Si l'erreur vaut 0 alors toutes les réponses sont exactes. Ainsi, faire apprendre un réseau de neurones consiste à trouver des valeurs pour les poids qui minimisent l'erreur de sortie associée à l'ensemble des échantillons en entrée. On se ramène donc à un problème d'optimisation où l'on recherche le minimum d'une fonction.

3.2 Optimisation par la méthode du gradient

Comment trouver des valeurs pour les poids qui permettent d'obtenir une erreur minimale. Cette question est difficile plusieurs minima locaux peuvent exister et lorsque nous détectons un minimum local, nous ne savons pas vraiment si nous pouvons en trouver un meilleur. Exception faite si le minimum trouvé est proche de zéro, dans ce cas, nous savons que nous avons une erreur faible et que l'apprentissage s'est bien passé.

A la question, quels poids choisir au démarrage ? La réponse sera : choisissons-les au hasard. Nous obtenons ainsi une valeur courante pour l'erreur. Comment faire évoluer les valeurs W des poids pour diminuer l'erreur ? Générons au hasard un nouveau vecteur de poids W' proche de W. Calculons alors la nouvelle valeur de l'erreur associé à ces nouveaux poids W'. Si cette nouvelle erreur est plus faible, alors nous remplaçons les poids actuels par ceux de W'. Nous répétons ce processus et finalement l'erreur finit par diminuer notablement ! Voici donc l'algorithme d'optimisation par recherche dans le voisinage :

```
Initialiser les poids W au hasard
ErrCourante = N(W)
Tant qu'il nous reste du temps :
      Choisir des poids W2 proches des valeurs de W
      Err = N(W2)
      Si Err < ErrCourante :
            W = W2
            ErrCourante = Err
```

Cet algorithme fonctionne correctement et nous permet de minimiser l'erreur. Cependant il n'est pas optimal. En effet, sa performance dépend de sa probabilité de générer une nouvelle série de poids qui diminuent l'erreur courante. Par exemple, avec un seul poids, c'est 1 chance sur 2, mais lorsque le nombre de poids augmentent les choses deviennent plus difficiles.

Heureusement, les mathématiques nous viennent en aide et nous pouvons calculer une variation des poids W qui nous permettent de diminuer l'erreur quasiment à chaque fois. Et cerise sur le gâteau,

PyTorch et toutes les librairies de deep-learning savent calculer ceci pour vous ! La variation idéale s'appelle le gradient, noté *grad*. Ce vecteur correspond à la différence entre W et W' tout simplement.

Il reste cependant une ambiguïté, car Si $W' = W + grad$ donne des poids qui améliorent notre score, peut-être pourrions-nous tenter d'aller un peu plus vite en choisissant : $W' = W + 2 \times grad$ par exemple. Ce problème s'appelle le choix du pas du gradient. Le pas, lui, sera choisi par essais successifs. S'il est trop petit, l'erreur ne va presque pas varier. S'il est trop important, l'erreur oscille en faisant des rebonds : 2, 1.8, 2.5, 2.3, 2.1, 1.5, 2.7, 2.3…

En résumé, voici la méthode du gradient :

```
Initialiser les poids W
Choisir une valeur pour le pas
Tant qu'il nous reste du temps :
      Erreur = N(W)
      W = W – grad(W) * pas
```

3.3 Ex 2 : Apprentissage d'une courbe

Notre apprentissage va consister à faire apprendre à notre réseau la fonction cosinus sur l'intervalle [-π/2, π/2]. Pour cela, nous utilisons un réseau à 2 couches avec 10 neurones sur la première couche et un seul neurone sur la dernière couche.

```
import numpy as np
import random, math
import torch.nn as nn
import torch

# courbe d'origine
xmin = -math.pi
xmax = math.pi
nbpt = 100
Lx = np.linspace(xmin,xmax, nbpt)
Ly = np.empty(nbpt)
for i in range(nbpt):
Ly[i] = math.cos(Lx[i])
```

A ce niveau, nous avons créé les tableaux *Lx* et *Ly* qui contiennent les abscisses et les ordonnées de points échantillonnés sur le graph de $y=cos(x)$. Le nombre de points a été fixé à 100.

```
# création du réseau
layer1  = nn.Linear(1,10)
relu1   = nn.ReLU()
layer2  = nn.Linear(10,1)
loss    = nn.MSELoss()
```

Ici nous créons notre réseau. La première couche est une couche Linear avec une entrée et 10 sorties, elle contient donc 10 neurones. Ces neurones utilisent une fonction d'activation ReLU. La deuxième couche est aussi une couche Linear avec un seul neurone ayant 10 entrées (correspondant aux 10 neurones de la couche précédente) et une unique sortie.

Nous choisissons comme fonction d'erreur la fonction MSELoss() qui somme les écarts au carré entre la sortie du réseau et les ordonnées associées à la fonction cosinus. Ainsi, la sortie du réseau est un réel positif. Si cette erreur est nulle, la fonction associée au réseau passe exactement par tous les points échantillonnés. Plus l'erreur augmente, plus la fonction s'en éloigne.

```
for t in layer1.parameters() : print(t)
```

Grâce à cette ligne, nous affichons l'ensemble des paramètres de la couche layer1 en utilisant la fonction parameters(). Ceci nous donne ainsi les valeurs des poids et des biais associés à cette couche :

```
Parameter containing:
tensor([[-0.4985],[-0.1893],[-0.0131],[ 0.0458],[ 0.1936],[-0.0853],[-0.4405],[-
0.8209],[ 0.4674],[-0.5536]], …)
Parameter containing:
tensor([-0.2364, -0.1675, 0.1367, -0.5873, 0.3293, 0.6181, -0.8515, -0.7904,
0.6012, 0.7431], …)
```

```
# création des tenseurs
input_tensor  = torch.FloatTensor(Lx)
input_tensor  = input_tensor.reshape((nbpt,1))
output_tensor = torch.FloatTensor(Ly)
output_tensor = output_tensor.reshape((nbpt,1))
```

Nous transformons les tableaux en tenseurs comme vu précédemment.

```
# apprentissage et gradient descent
pas = 0.005
LResults = []
err = 1000
iter = 0
while err > 0.001 and iter < 20000 :
```

Nous mettons en place les hyperparamètres d'apprentissage. Ces paramètres n'interviennent pas dans les calculs, pourtant ils gèrent globalement la convergence, le résultat final dépend ainsi de ces valeurs. La boucle va opérer tant que l'erreur n'est pas en dessous d'un certain seuil et elle s'arrête au pire au bout d'un certain nombre d'itérations.

```
while err > 0.001 and iter < 20000 :
   # reinitialise la valeur du gradient
   layer1.zero_grad()
   layer2.zero_grad()
   # passe FORWARD
   v = layer1(input_tensor)
   v = relu1(v)
   v = layer2(v)
   erreur = loss(v,output_tensor)
   err = erreur.item()
   # passe BACKWARD
   erreur.backward()
   # gradient descent W -= grad * pas
   layer1.weight.data -= layer1.weight.grad * pas
   layer2.weight.data -= layer2.weight.grad * pas
```

Nous sommes ici dans le cœur de la bouche d'apprentissage. Voici les étapes qui la composent :

- **Réinitialisation du gradient** : chaque couche contenant des informations pour calculer le gradient de ses paramètre internes, voit ces informations remises à zéro. Si vous oubliez cette réinitialisation, la méthode va se comporter étrangement car la direction du gradient sera incorrecte !
- **Passe Forward** : à ce niveau, nous prenons les valeurs d'entrée et nous les propageons d'une couche à l'autre jusqu'à obtenir la valeur de sortie. Durant cette passe, de nombreuses informations sont stockés dans chaque couche pour le futur calcul du gradient.
- **Passe Backward** : Pytorch déclenche une propagation de la dernière couche vers la couche d'entrée, dans le sens inverse de la passe Forward. Cette passe sert à finaliser le calcul du gradient pour chaque couche du réseau.
- **Algorithme de descente du gradient** : pour chaque couche ayant des paramètres à modifier, nous appliquons la méthode de la descente du gradient.

Les dernières lignes servent à générer l'affichage d'une animation montrant l'apprentissage du réseau. Expérimentez :

- Lancez plusieurs runs avec des valeurs différentes pour le pas et pour l'erreur.
- Lancez plusieurs runs avec les mêmes valeurs de pas. Vous constaterez que pour les mêmes valeurs, des simulations peuvent échouer et d'autres convergent rapidement. Cela vient de l'initialisation aléatoire des paramètres au démarrage du programme. Certaines fois, le hasard fera que les conditions de départ sont optimales, d'autres fois la situation de départ produira un résultat médiocre.

Exercice 02 : Modifiez le réseau pour qu'ils contiennent trois couches Linear 10x10x1. Examinez les nouveaux résultats. La convergence est-elle facilitée ?

3.4 Ex 03 : Les optimiseurs de PyTorch

Nous avons dans l'exercice 02 programmé nous-même l'algorithme de la descente du gradient afin de bien saisir toutes les étapes de cette méthode. Cependant, la librairie PyTorch fournit des facilités pour simplifier la création et l'apprentissage d'un réseau.

Pour les réseaux séquentiels que nous avons construits, PyTorch offre des outils pour faciliter la création de ce genre de réseau. Consultez la documentation Pytorch dans la section : torch.nn.Sequential. Cette formulation est plus compacte, car elle permet de décrire le réseau une seule fois et la passe forward s'écrira seulement en une seule ligne. Cependant, du point de vue du développeur, cette écriture est plus dangereuse. En effet, en cas d'erreur dans la conception des couches, elle rend le réseau plus difficile à débuguer.

La librairie Pytorch fournit aussi des algorithmes plus avancés que la descente du gradient. Ces méthodes vont être accessibles à travers des objets que l'on doit associer à un réseau. Ces optimiseurs gèrent automatiquement la phase de backward et la modification des paramètres du réseau. Consultez la documentation Pytorch dans la section torch.optim.adam à ce sujet.

Exercice 03 : reprenez l'exercice 02 et intégrez un objet de type Sequential pour décrire le réseau. Mettez en place un optimiseur de type Adam ou SGD.