Introduction à PyTorch

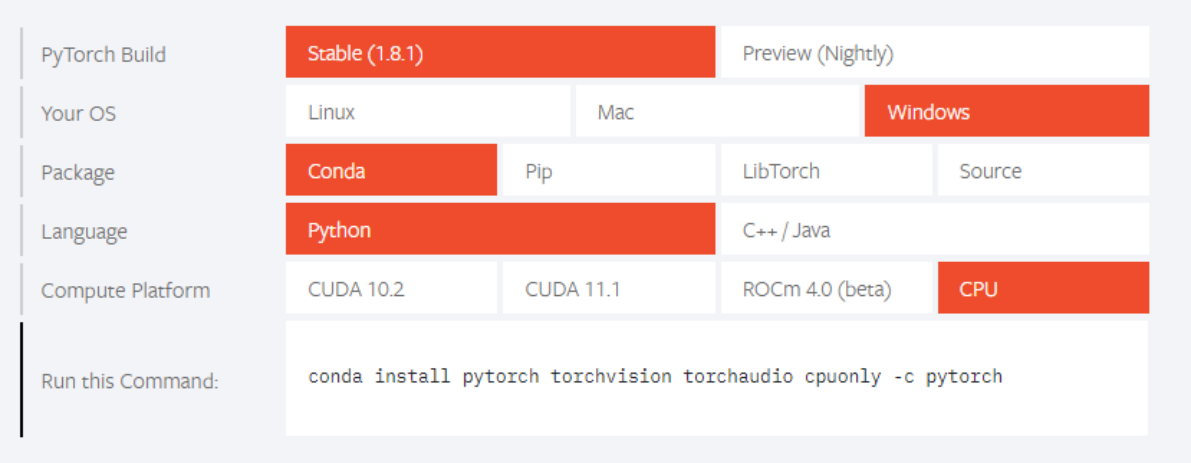
1 Découverte de PyTorch

1.1 Installer PyTorch

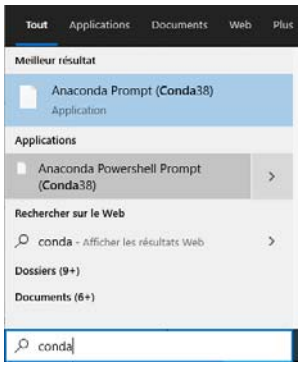
La librairie PyTorch, maintenue par Facebook, permet de mettre en place de puissants réseaux de neurones. De plus, cette libraire offre une grande simplicité d'utilisation et dispose d'une documentation complète en ligne. Elle peut d'exploiter les performances des CPU ou des GPU. Pour son installation, une page web interactive a été mise en place pour expliquer la marche à suivre suivant votre système et votre configuration. Rendez-vous donc sur www.PyTorch.org. Descendez dans la page pour arriver dans la zone contenant le tableau avec les cases orange. Voici les différents choix qui s'offre à vous :

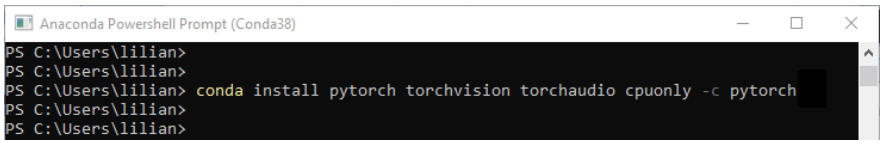
- Par défaut vous installerez avec la version Stable et non le Night Build.
- Choisissez votre OS.
- Si vous travaillez avec Anaconda, vous devez cliquer sur la case CONDA. Si vous travaillez avec une version de Python classique, cliquez sur Pip.
- Nous travaillons avec le langage Python uniquement.
- Nous vous conseillons d'installer la version CPU surtout si vous êtes débutant.

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

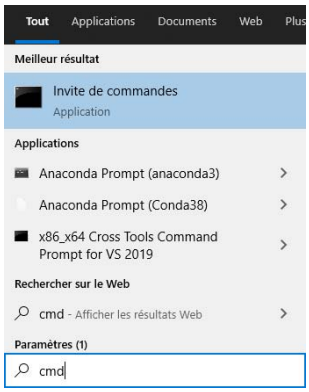


- **Windows/Anaconda** : lancez une fenêtre Anaconda prompt et recopiez la ligne proposée dans « Run this Command » :





- **Windows/Python** : lancez une fenêtre cmd et recopiez la ligne proposée dans « Run this Command » :



- **Linux/Mac** : lancez une fenêtre shell et recopiez la ligne proposée dans « Run this Command ». Utilisez la commande **pip3** et non pip car par défaut sur les distributions Linux/Mac, pip et python pointe vers une version historique de Python2 présente par défaut sur l'OS. Il faut donc aussi utiliser la commande **python3** pour lancer l'interpréteur Python dans sa dernière version.
- **Plusieurs installations de Python**. Si vous voulez installer Python spécifiquement pour une des versions de Python installée sur votre système, ouvrez une fenêtre de commande (powershell/cmd/shell) et déplacez-vous dans le répertoire où se trouve installée la version de Python que vous recherchez. Utilisez ensuite la commande : python -m pip install… en complétant la ligne proposée après « Run this Command ». Si vous ne faites pas ainsi, l'installation s'effectuera avec la version de Python désignée par votre PATH.

1.2 Rappels et conventions

Par convention, dans ce document :

- Le terme **liste** désigne une liste Python
- Le terme **tableau** dans ce document désigne un Numpy array
- Le terme **tenseur** désigner un tenseur Pytorch

Création d'une liste en Python :

```
L = [ 1, 2, 3, 5, 8]          # liste
L = [ [1,2,3,4], [5,6,7,8]]   # liste de listes
```

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
```

Il est possible de convertir des listes en tableaux et réciproquement :

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

Remarque : dans les exercices, nous allons donner des valeurs d'exemple souvent représentées par des listes de nombres. Ces listes seront converties en tenseurs pour être traitées par la librairie Pytorch qui fournira en sortie un tenseur résultat. Pour afficher des graphiques afin d'illustrer ces résultats, nous utiliserons ensuite des librairies tierces qui requièrent souvent des Numpy Array. Il faudra donc convertir le tenseur résultat vers une liste ou un tableau.

Ce document n'a pas pour but de présenter les manipulations des listes et des tableaux. Nous allons 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 définir aléatoirement un tenseur avec la fonction rand() :

```
a = torch.rand((2,3))
==> tensor([ [0.6004, 0.9491, 0.7909],
             [0.3673, 0.5234, 0.1315] ])
```

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 l'extraire. Voici un exemple :

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

1.4 Taille d'un tenseur : fonction shape

La propriété shape d'un tenseur nous permet de connaître sa taille :

```
import torch
A = torch.FloatTensor([ [[0,0,0,0],[1,1,1,1],[2,2,2,2]],
                        [[0,0,0,0],[1,1,1,1],[2,2,2,2]] ])
print(A.shape)
=> torch.Size([2, 3, 4])
```

Si nous examinons la liste de listes de listes Python : [[[0,0,0,0], [1,1,1,1], [2,2,2,2]], [[0,0,0,0], [1,1,1,1], [2,2,2,2]]], le plus bas niveau est constitué de listes de 4 entiers. Cette taille correspond à la valeur la plus à droite dans la dimension [2,3,4] du tenseur. Plus on imbrique de niveaux, plus le tenseur a de dimensions :

- Une liste de 4 entiers est associée à un tenseur de taille (4)
- 3 listes de listes de 4 entiers sont associées à un tenseur de taille (3,4).
- 2 listes de 3 listes de listes de 4 entiers sont associées à un tenseur de taille (2,3,4).

1.5 Indexer un tenseur

La syntaxe pour indexer un tenseur utilise des **virgules**. Faîtes attention, car la syntaxe avec des crochets existent aussi mais offre moins de possibilités. Chaque indice doit respecter la plage de valeurs correspondant à la dimension du tenseur :

$$A.shape = [2, 3, 4] \Rightarrow A[i, j, k] \quad \text{avec} \quad 0 \leq i < 2, \quad 0 \leq j < 3, \quad 0 \leq k < 4$$

Pour accéder à un élément du tenseur, nous écrivons :

```
A = torch.FloatTensor([ [[0,0,0,0],[1,1,1,1],[2,2,2,2]],
                        [[0,0,0,0],[1,1,1,1],[2,2,2,7]] ])
print(A[1,2,3])
=> 7
```

1.6 Correspondance matrice / tenseur

Dans le cas deux dimensions, si vous voulez faire correspondre les lignes de votre tenseur avec les lignes d'une matrice, vous devez permuter les indices x et y correspondant aux indices de colonnes et de lignes. Ainsi, il faut écrire ces indices dans l'ordre inverse :

```
A[y,x]    # élément de la matrice, colonne x et rangée y
```

1.7 Plage d'indices et opérateur :

Il est possible lorsque l'on modifie un tenseur de traiter en une commande plusieurs valeurs associées à une plage d'indices, on parle du mécanisme de **slicing**. Pour cela, on utilise l'opérateur « : » qui a plusieurs significations suivant le contexte. Pour ceux connaissant Matlab ou Python/Numpy, le principe de fonctionnement est similaire :

```
a = torch.tensor([[1,2,3,4],[6,7,8,9]])
print( a[0] )      # 1ere ligne
print( a[1] )      # 2eme ligne
print( a[0,1:] )   # 1ere ligne, tous les indices >=1
print( a[0,:2] )   # 1ere ligne, tous les indices < 2
print( a[0,-1] )   # 1ere ligne, 1er élément en partant de la fin
print( a[0,:] )    # 1ere ligne, et tous les éléments
```

Ce qui donne :

```
==> tensor([1, 2, 3, 4])
==> tensor([6, 7, 8, 9])
==> tensor([2, 3, 4])
==> tensor([1, 2])
==> tensor(4)
==> tensor([1, 2, 3, 4])
```

On peut étendre encore ce système pour sélectionner un sous-tenseur :

```
a = torch.tensor( [ [10,11,12,13],
                    [20,21,22,23],
                    [30,31,32,33],
                    [40,41,42,43] ] )
print( a[1:3,1:3] )     # sélection des a[i,j] avec 1≤i<3 et 1≤j<3
a[1:3,1:3] = 99         # affection des indices sélectionnées
print(a)
```

Ce qui donne comme résultat :

```
==> tensor([[21, 22],
            [31, 32]])
==> tensor([[10, 11, 12, 13],
            [20, 99, 99, 23],
            [30, 99, 99, 33],
            [40, 41, 42, 43]])
```

Si vous voulez utiliser un pas, la syntaxe est la suivante :

Indice de début : Indice de fin : pas

```
a = torch.tensor( [ [10,11,12,13,14,15],
                    [20,21,22,23,24,25],
```

```
                    [30,31,32,33,34,35],
                    [40,41,42,43,44,45] ] )
a[:, 0:6:2 ] = 99    # affection des colonnes 0 2 4
==>  tensor([ [99, 11, 99, 13, 99, 15],
              [99, 21, 99, 23, 99, 25],
              [99, 31, 99, 33, 99, 35],
              [99, 41, 99, 43, 99, 45]])
```

1.8 Indexage complexe

Chaque nouvelle version de PyTorch propose de nouvelles fonctionnalités pour faciliter l'utilisation des tenseurs. Nous présentons certaines de ces syntaxes, pas forcément très courantes, mais qui pourront parfois vous aider.

Indexation sur une liste d'indices :

```
a = torch.tensor( [ [10,11,12,13],
                    [20,21,22,23],
                    [30,31,32,33],
                    [40,41,42,43] ] )
i = torch.LongTensor( [1,3])  # les indices sont des entiers 64 bits
a[i,:] = 7      # affectation des lignes d'indice 1 et 3
==> tensor([ [10, 11, 12, 13],
             [ 7, 7, 7, 7],
             [30, 31, 32, 33],
             [ 7, 7, 7, 7]])
```

Indexation sur une liste de booléens :

```
a = torch.tensor( [ [10,11,12,13],
                    [20,21,22,23],
                    [30,31,32,33],
                    [40,41,42,43] ] )
i = torch.BoolTensor( [True,True,False,True])
a[i,:] = 7     # affectation sur les lignes associées à la valeur True
==> tensor([ [ 7, 7, 7, 7],
             [ 7, 7, 7, 7],
             [30, 31, 32, 33],
             [ 7, 7, 7, 7] ])
```

Ce qui peut donner des utilisations originales comme :

```
a = torch.tensor( [ [10,11,12,13],
                    [20,21,22,23],
                    [30,31,32,33],
                    [40,41,42,43] ] )
b = torch.tensor([1,2,13,14])
a[b>10,:] = 99     # affectation pour les lignes associées à un indice  > 10
==>  tensor( [ [10,11,12,13],
               [20,21,22,23],
               [99,99,99,99],
               [99,99,99,99] ] )
```

1.9 La recopie

Nous rappelons que dans le langage Python, la syntaxe « A = B » fait en sorte que la variable A désigne le même objet que la variable B. Ainsi, A ne désigne pas un nouvel objet et par conséquent toute modification sur A, se reporte sur B. Cette remarque est valable pour les tenseurs :

```
T = torch.ones((2,2))
A = T
A[0,0] = 5
print(T)
print(A)
```

Ce qui donne comme résultat :

```
==> tensor([[5., 1.],
            [1., 1.]])
==> tensor([[5., 1.],
            [1., 1.]])
```

Pour déclencher une copie d'un tenseur, il faut utiliser la fonction clone(). Cependant, nous travaillons dans une librairie de deep learning et un tenseur contient aussi des informations supplémentaires nécessaires pour la phase de rétropropagation. Ainsi, si vous ne voulez copier que les données, et désassocier le tenseur du graph de calcul, il est préférable d'utiliser en plus la fonction detach().

```
T = torch.ones((2,2))
A = T.clone().detach()   # ou clone() si besoin
A[0,0] = 5
print(T)
print(A)
```

Ce qui donne :

```
==> tensor([[1., 1.],
            [1., 1.]])
==> tensor([[5., 1.],
            [1., 1.]])
```

1.10 Les vues

Lorsque l'on manipule des tenseurs de grande taille, il est préférable de mettre en place un système qui évite les copies inutiles. Ainsi, la librairie PyTorch permet à un tenseur d'être une « vue partielle » d'un tenseur existant. Le tenseur/vue et le tenseur d'origine partage les mêmes données.

1.10.1 Vue par slicing

Par exemple, un slicing retourne une vue :

```
T = torch.ones((100,100))
A = T[0:50,0:50]  # A est une vue partielle de T
A[0,0] = 5
print(T.storage().data_ptr() == A.storage().data_ptr())
print(T[0,0])
```

Ce qui retourne :

```
==> True        # création d'une vue, A et T ont le même champ de données
==> tensor(5.)
```

1.10.2 Vue par sélection

Une autre manière d'obtenir une vue de manière implicite est de sélectionner une dimension d'un tenseur :

```
T = torch.FloatTensor([ [[0,1,2,3],[1,2,3,4],[2,3,4,5]],
                        [[10,11,12,13],[11,12,13,14],[12,13,14,15]] ] )
A = T[0]
B = T[0][0]
print(A)
print(B)
print(T.storage().data_ptr() == A.storage().data_ptr())
print(T.storage().data_ptr() == B.storage().data_ptr())
```

Ce qui retourne :

```
==> tensor([[0., 1., 2., 3.],    # T[0] : sélection du tenseur T de dim (3,4)
            [1., 2., 3., 4.],
            [2., 3., 4., 5.] ])
==> tensor( [0., 1., 2., 3.])    # T[0][0] : sélection du tenseur T de dim (4)
==> True
==> True
```

1.10.3 Vue par reshape

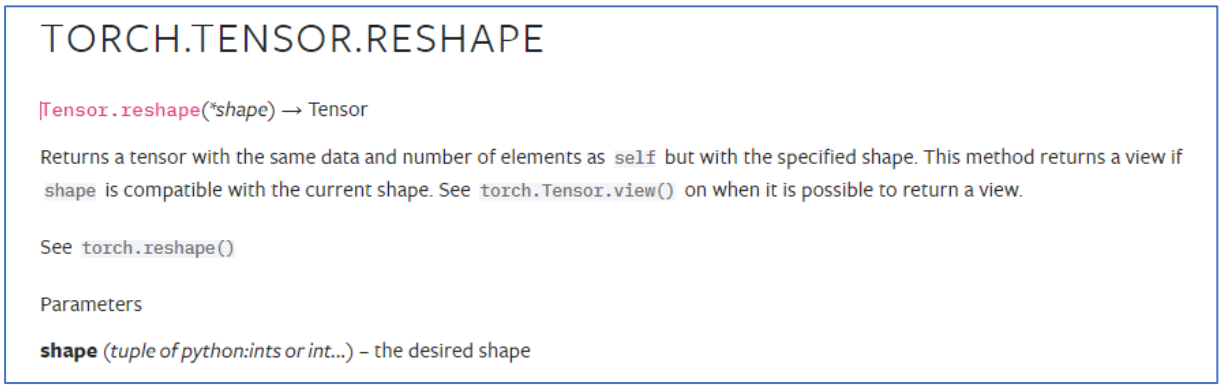
Lorsque nous effectuons un reshape sur un tenseur, on obtient généralement une vue :

```
T = torch.ones((2,3))
A = T.reshape(6)
A[0] = 9
print(T)
print(T.storage().data_ptr() == A.storage().data_ptr())
```

Ce qui donne :

```
==> tensor([9., 1., 1., 1., 1., 1.])
==> True
```

Cependant, les choses ne sont pas toujours évidentes avec la fonction reshape, regardons la documentation :



TORCH.TENSOR.RESHAPE

Tensor.reshape(*shape) → Tensor

Returns a tensor with the same data and number of elements as `self` but with the specified shape. This method returns a view if `shape` is compatible with the current shape. See `torch.Tensor.view()` on when it is possible to return a view.

See `torch.reshape()`

Parameters

shape (*tuple of python:ints or int...*) – the desired shape

La dernière ligne nous indique que la fonction reshape() prend comme paramètre un tuple. Ainsi, pour redimensionner un tenseur T, l'appel à la fonction reshape s'écrit :

T = T.reshape((…,…))

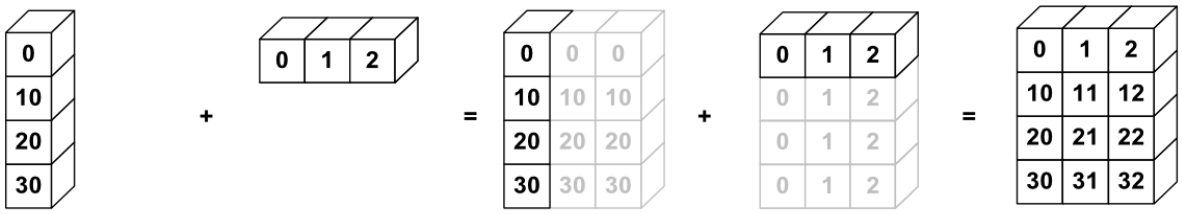
Prenons un exemple avec 6 valeurs dans un tenseur 1D : [1,2,3,4,5,6]. Nous pouvons grâce à la fonction reshape() modifier la dimension du tenseur pour :

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

Remarque : l'utilisation de la fonction reshape peut retourner une vue mais aussi une copie lorsque sa mécanique interne le décide. La plupart du temps, dans les cas standards : création/chargement d'un tenseur puis application d'un reshape, une vue sera créée. Mais si vous appliquez un reshape sur un tenseur dont la taille n'est pas totalement compatible, le retour sera probablement une copie de cet élément.

2 Tenseurs et broadcasting

Pourquoi avoir choisi le terme de tenseur (tensor en anglais) plutôt que celui de matrice ? En fait, les tenseurs sont des containers de données comme les tableaux, les listes ou les dictionnaires. Ils ne sont pas spécialement liés aux applications linéaires : un tenseur n'a pas vocation à rechercher des valeurs propres ou à servir pour 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 vecteur colonne à un vecteur ligne est en math impossible, cependant, pour un tenseur cela a un sens. Voici comment on procède :



Si vous ajoutez un tenseur de taille (4,1) avec un tenseur de taille (1,3), tout se comporte comme si chaque tenseur était dupliqué autant de fois que nécessaire pour obtenir deux tenseurs de taille (4,3). Le résultat obtenu est ainsi un tenseur de taille (4,3).

2.1 Logique du broadcasting

Si intuitivement on comprend comment les opérations se déroulent, voici une manière plus rigoureuse de les expliquer. Notons $(a_0, \ldots, a_n)$ et $(b_0, \ldots, b_m)$ les dimensions des deux tenseurs en entrée. Si $n \neq m$, il suffit de compléter par la gauche avec des 0 la taille d'un des deux tenseurs afin d'obtenir $n = m$.

Tout d'abord avant une opération de broadcasting, PyTorch vérifie si les dimensions des tenseurs sont compatibles :

Les dimensions des tenseurs A et B sont compatibles si $\begin{cases} a_i = 0/1 \ ou \\ b_i = 0/1 \ ou \\ a_i = b_i \end{cases}$

Voici quelques exemples :

- (3,4) + (3) donne $\rightarrow (a_0, a_1) = (3,4)$ et $b_0 = 3$. Nous devons ajouter un 0 sur la gauche de la dimension de b pour obtenir : $(b_0, b_1) = (0,3)$. Nous avons ainsi pour l'indice 0 : $a_0 = 0$ ce qui convient. Mais, pour l'indice 1 : $a_1, b_1 \neq 0,1$ et $a_1 \neq b_1$ donc les dimensions sont incompatibles.
- (1,1,4) + (1,3) donne $(a_0, a_1, a_2) = (1,1,4)$ et $(b_0, b_1, b_2) = (0,1,3)$. Nous avons $a_0, a_1 = 1$ ce qui convient, mais cependant $a_2, b_2 \neq 0,1$ et $a_2 \neq b_2$ donc les dimensions sont incompatibles.

Finalement, la taille $(s_0, \ldots, s_n)$ du tenseur de sortie est donnée par la formule :

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

Voici quelques exemples pour trouver la taille du tenseur résultat :

- (3,4) + (1) $\rightarrow$ (3,4) + (0,1) $\rightarrow$ (3,4)

- (3,4) + (1,1) → (3,4)
- (3,1) + (4) → (3,1) + (0,4) → (3,4)
- (3,1) + (1,4) → (3,4)
- (1,1,4) + (1,3,1) → (1,3,4)
- (1,1,4) + (3,2,1) → (3,2,4)

Notons (a_0, a_1, a_2) et (b_0, b_1, b_2) les tailles des deux tenseurs en entrée A et B. Le comportement de PyTorch lorsque l'on effectue l'opération C = A + B en mode broadcasting se traduit par le code suivant :

fonction : γ (i,size) retourne i si size > 1, 0 sinon

Avec $0 \leq i < \max(a_0, b_0)$:
  Avec $0 \leq j < \max(a_1, b_1)$:
    Avec $0 \leq k < \max(a_2, b_2)$:
      $C[i, j, k] = A[\gamma(i, a_0), \gamma(j,a_1), \gamma(k,a_2)] + B[\gamma(i,b_0), \gamma(j,b_1), \gamma(k,b_2)]$

2.2 Exemple : tenseur 2D + tenseur 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)
```

Le tenseur A a une dimension $(a_0, a_1) = (4,3)$ et le tenseur B une dimension $(b_0, b_1) = (1,1)$. Ces deux dimensions sont compatibles, car b_0 et $b_1 = 1$. Le tenseur résultat a donc une dimension de $(max(4,1), max(3,1)) = (4,3)$. Voici comment est construit le tenseur R par broadcasting :

Avec $0 \leq i < 4$ et $0 \leq j < 3$: R[i, j] = A[i, j] + B[0,0]

Voici le résultat obtenu :

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

2.3 Exemple : tenseur 2D + tenseur 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)
```

Le tenseur A a une dimension $(a_0, a_1) = (4,3)$ et le tenseur B une dimension $(b_0, b_1) = (4,1)$. Ces deux tailles sont compatibles, car $a_0 = b_0$ et $b_1 = 1$. Le tenseur résultat a donc une taille de $(max(4,4),max(3,1)) = (4,3)$. Voici comment est construit le tenseur R par broadcasting :

Avec $0 \leq i < 4$ et $0 \leq j < 3$: R[i, j] = A[i, j] + B[i,0]

Nous avons comme résultat affiché :

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

2.4 Exemple : tenseur 2D + tenseur ligne

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

Le tenseur A a une dimension $(a_0, a_1) = (4,3)$ et le tenseur B une dimension $(b_0, b_1) = (0,3)$. Ces deux dimensions sont compatibles, car $b_0 = 0$ et $a_1 = b_1 > 1$. Le tenseur résultat a donc une dimension de $(max(4,0),max(3,3)) = (4,3)$. Voici comment est construit le tenseur R par broadcasting :

Avec $0 \leq i < 4$ et $0 \leq j < 3$: R[i, j] = A[i, j] + B[i]

Voici donc le résultat affiché :

```
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 (1,3) ce qui produit un tenseur de taille (4,3). Le résultat est donc équivalent : même dimension et mêmes valeurs.

2.5 Exemple : 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)
```

Le tenseur A a une dimension $(a_0, a_1) = (4,1)$ et le tenseur B une dimension $(b_0, b_1) = (1,3)$. Ces deux dimensions sont compatibles, car $b_0 = 1$ et $a_2 = 1$. Le tenseur résultat a donc une dimension de $(max(4,1), max(1,3)) = (4,3)$. Voici comment est construit le tenseur R par broadcasting :

Avec $0 \leq i < 4$ et $0 \leq j < 3$: R[i, j] = A[i, 0] + B[0, j]

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

On peut se demander ce qu'il se passe si l'on ajoute un vecteur ligne à un vecteur colonne :

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

Dans la logique du broadcasting, le résultat est identique, ce ne serait pas le cas en algèbre linéaire.

2.6 Exemple : tenseur 2D + tenseur colonne 3D

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

La dimension du tenseur A est (3,3) et celle du tenseur B est (3,1,1).

Le tenseur A a une dimension $(a_0, a_1, a_2) = (0,3,3)$ et le tenseur B une dimension $(b_0, b_1, b_2) = (3,1,1)$. Ces deux dimensions sont compatibles, car $b_1 = b_2 = 1$ et $a_0 = 0$. Le tenseur résultat a donc une dimension de $(max(0,3), max(3,1), max(3,1)) = (3,3,3)$. Voici comment est construit le tenseur R par broadcasting :

Avec $0 \leq i, j, k < 3$: R[i, j, k] = A[j, k] + B[i,0,0]

```
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.]]])
```

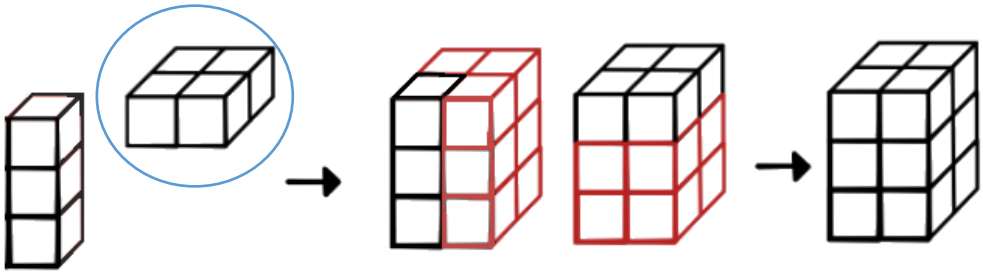
Remarque : si nous oublions par erreur une paire de crochets en écrivant le tenseur B, nous sommes d'après les règles du broadcasting dans un cas tenseur 2D + tenseur colonne : (3,3) + (3,1) ce qui donne un tenseur de taille (3,3) :

```
import torch
A = torch.FloatTensor([[0,0,0],[10,10,10],[20,20,20]])
B = torch.FloatTensor([[4],[5],[6]])
print(A+B)
tensor([ [ 4.,  4.,  4.],
         [15., 15., 15.],
         [26., 26., 26.] ])
```

Ainsi l'oubli d'une paire de crochets a totalement changé le résultat. Il faut donc faire attention, car une erreur dans les manipulations des tenseurs ne provoque pas forcément un message d'erreur. De ce fait, la souplesse et l'adaptabilité des opérations sur les tenseurs fait que l'on doit redoubler de vigilance lorsqu'on les manipule.

2.7 Représentation sous forme graphique

On trouve ce type de schéma sur certains tutoriels internet :



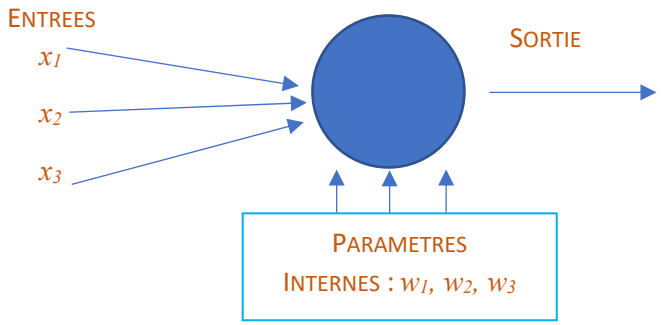
Le tenseur 2D entouré en bleu est un tenseur 2D de dimension (2,2). Il est représenté couché en travers. Le vecteur colonne sur la gauche occupe une troisième dimension, il est donc de taille (3,1,1), ce qui donne après broadcasting un tenseur de taille (3,2,2).

2.8 Exercices

Ouvrez le fichier « Ex 4 tensor.py » et complétez chaque exercice.

3 Rôle des neurones

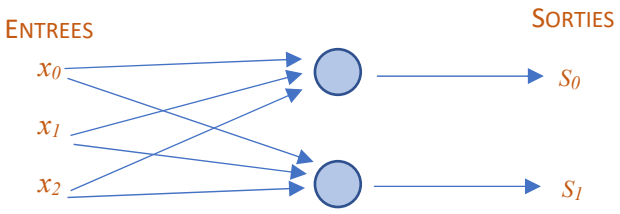
3.1 Le neurone informatique



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 :

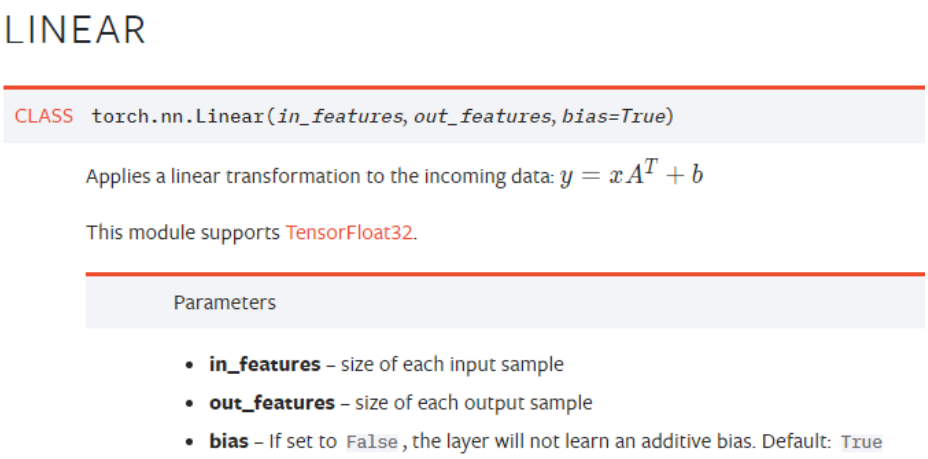
$$Sortie = \sum_{i=0}^{n-1} x_i * w_i + b$$

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 couche Linear. En consultant la documentation PyTorch, nous trouvons à la page :

https://PyTorch.org/docs/stable/generated/torch.nn.Linear.html



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.

Les poids sont stockés dans le tenseur A, les entrées dans le tenseur x et le biais dans le tenseur b. La formule : $y = xA^T + b$ décrit le calcul mis en place par la couche Linear : une multiplication matricielle est effectuée entre x *et* A pour construire les valeurs en sortie des neurones. Puis si besoin, un biais est ajouté à chaque valeur.

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

- 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)* et correspond aux $n\_out \times n\_in$ poids de la couche

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)
Input  = torch.FloatTensor([1,2,3,4])
Output = layer(Input)
```

```
print("Weight : ", layer.weight)
print("Biais  : ", layer.bias)
print("Input  : ", Input)
print("Output : ", Output)

print("Weight shape : ", layer.weight.shape)
print("Biais shape  : ", layer.bias.shape)
print("Input shape  : ", Input.shape)
print("Output shape : ", Output.shape)
```

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

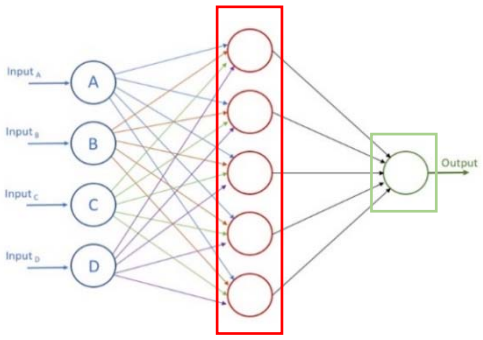
```
Weight : Parameter containing:
      tensor([[ 0.1304, -0.2287, 0.0151, -0.0708],
      [-0.0865, -0.3633, -0.3858, 0.1804]], requires_grad=True)
Biais  : Parameter containing:
      tensor([0.1493, 0.0297], requires_grad=True)
Input  : tensor([1., 2., 3., 4.])
Output : tensor([-0.4158, -1.2191], grad_fn=<AddBackward0>)

Weight shape : torch.Size([2, 4])
Biais shape : torch.Size([2])
Input shape : torch.Size([4])
Output shape : torch.Size([2])
```

Remarque : les tenseurs stockant les poids et les biais de la couche Linear ont été initialisés directement lors de la création de cette couche. Ces deux tenseurs représentent des paramètres internes du problème d'apprentissage et nous devons donc déterminer leurs gradients. La librairie PyTorch le sait déjà et elle a ainsi annoté ces deux tenseurs avec un flag spécial : *requires_grad=True*. Le tenseur de sortie, Output, a aussi ce flag, car nous avons vu dans l'algorithme de rétropropagation que tous les résultats intermédiaires dans la chaîne de traitement vont servir à rétropropager les calculs intermédiaires du gradient. Le tenseur Input lui n'a pas ce flag car il représente des informations qui restent constantes durant la phase d'apprentissage et aucun calcul de gradient n'est nécessaire à leur niveau.

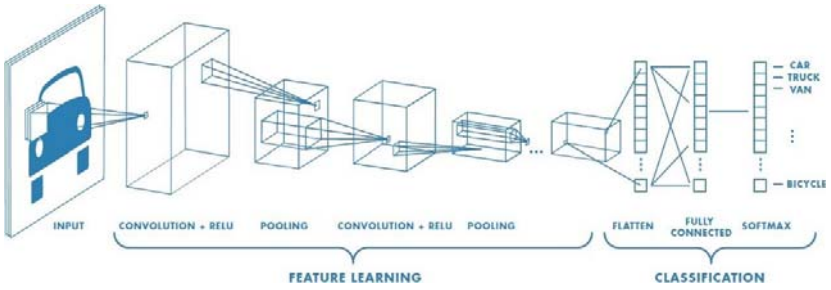
3.2 Topologie des réseaux

Une couche Linear peut être représentée graphiquement à partir d'un rectangle vertical, contenant des ronds correspondant aux neurones. Les entrées peuvent être aussi regroupées dans une couche d'entrée appelée « input layer » même si aucun neurone n'est présent. Cependant, la couche d''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.

3.3 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 $\mathbb{R} \rightarrow \mathbb{R}$ 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.

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 coûteuse 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 utilisée ainsi que sa fonction d'activation. Si cette fonction est performante, elle a le désavantage d'annuler le gradient lorsque x est négatif et lorsque le gradient est nul… impossible d'évoluer dans une direction ou une autre. Ainsi, certains auteurs préfèrent utiliser sa variante $Leaky\ ReLU(x)$ qui masque cet inconvénient :

$$Leaky\ ReLU(x) = max(\alpha x,x) \quad avec\ 0<\alpha<0.1$$

On peut trouver d'autres fonctions définies de $\mathbb{R}$ dans $\mathbb{R}$ 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.

3.4 Créer une couche avec 1 neurone

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

- En premier, nous créons une couche de type Linear nommée *layer* et nous lui donnons ses caractéristiques internes.
- Ensuite, nous utilisons cette couche *layer* comme une fonction en lui donnant un tenseur d'entrée. La couche *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], …)
```

3.5 Affichage du graph associé à un neurone+ReLU

Nous allons tracer le graph d'une fonction associée à un neurone avec sa fonction d'activation ReLU.

La fonction linspace() de numpy permet d'échantillonner un nombre donné de valeurs dans un intervalle ceci avec un pas régulier. Ainsi linspace(-2,2,5) échantillonne 5 valeurs dans l'intervalle [-2,2] et donne comme résultat : [-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 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.

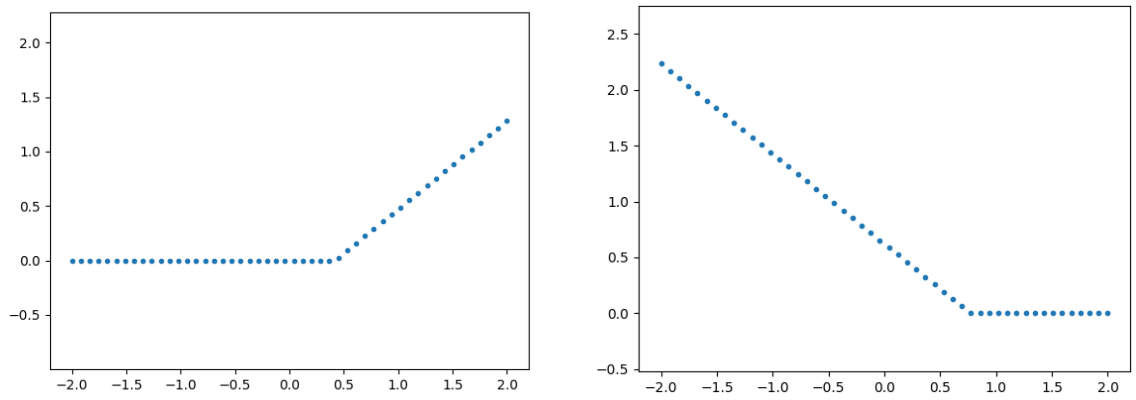
Pour tracer le graph de fonction associée à un neurone, une première approche consiste à faire une boucle pour calculer chaque valeur de sortie. Examinons le code :

```
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 fenêtre d'affichage
```

La première ligne : layer = torch.nn.Linear(1,1) crée une couche Linear avec un seul neurone. La deuxième ligne : activ = torch.nn.ReLU() nous permet de créer une fonction d'activation. Résumons la chaîne de calcul. Nous avons échantillonné différentes valeurs d'entrée stockées dans le tableau Lx. Pour chacune de ces valeurs, nous créons un tenseur *input* de dimension (1). Nous passons ce tenseur à la première couche en écrivant : *v1 = layer(input)* ; 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 le tracé de la sortie d'une couche Linear 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 obtenons un graph correspondant à la fonction $y = max(0, a.x+b)$ donnant à cette forme de droite tronquée :



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

3.6 Couche Linear en mode parallèle

Pour traiter plusieurs images, un réflexe serait de créer une boucle et de traiter les images itérativement en les injectant une à une dans la couche Linear. Cependant, la couche Linear peut

gérer elle-même cette boucle en interne. Pour cela, au lieu d'injecter itérativement k tenseurs image de dimension (n), nous injectons en une fois un seul tenseur stockant l'ensemble des images et qui sera de dimension (k,n). Cette deuxième approche permet d'effectuer les calculs des différentes images en parallèle sur plusieurs cœurs CPU ou GPU. A l'inverse, l'approche basée sur n appels itératifs ne permet pas cette optimisation. En résumé, voici comment se comporte une couche Linear :

Si une couche Linear traite un tenseur de taille (n) pour produire un tenseur de taille (m)

ALORS

Cette couche peut traiter un tenseur de taille (k,n) pour produire un tenseur de taille (k,m)

Nous trouvons cette information dans la documentation officielle de PyTorch au paragraphe Shape : https://PyTorch.org/docs/stable/generated/torch.nn.Linear.html

Shape:

- Input: $(N, *, H_{in})$ where $*$ means any number of additional dimensions and $H_{in} = \text{in\_features}$
- Output: $(N, *, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

3.7 Exercice

Ouvrez le fichier « Ex 5 Graph Linear.py ». Il reprend le code du tracé d'une couche Linear avec un seul neurone comme vu précédemment.

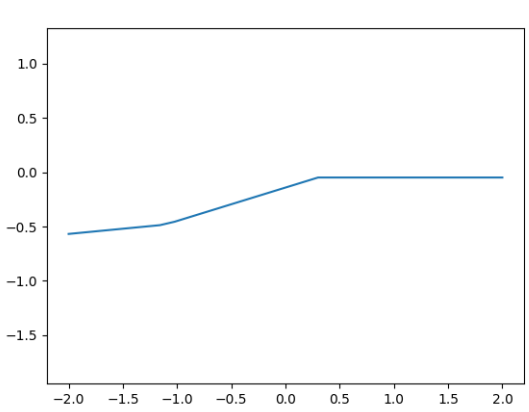
Etape 1 : Retirez la boucle for et utilisez le mécanisme de parallélisation vu ci-dessus.

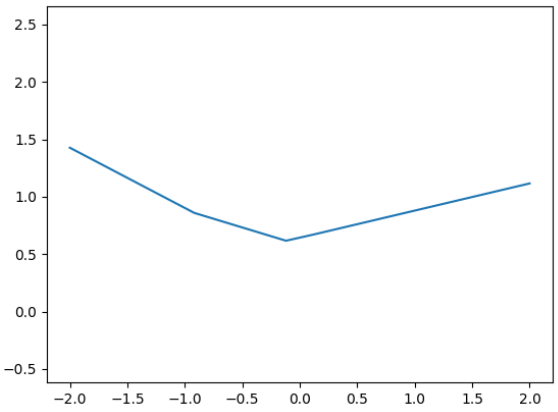
Etape 2 : Améliorez le réseau : la première couche doit contenir 3 neurones et la deuxième 1 neurone unique. La fonction d'activation en sortie de la première couche sera la fonction ReLU. Conserver le mécanisme de parallélisation et tracez le graphique associé.

Question subsidiaire : Quelle est la dimension du tenseur en entrée de la deuxième couche ?

Pour extraire les données d'un tenseur vers un tableau numpy, il faut d'abord le dissocier du graph de calcul en utilisant la fonction detach(). Vous pouvez ensuite utiliser la fonction .numpy() pour convertir ce tenseur en tableau numpy.

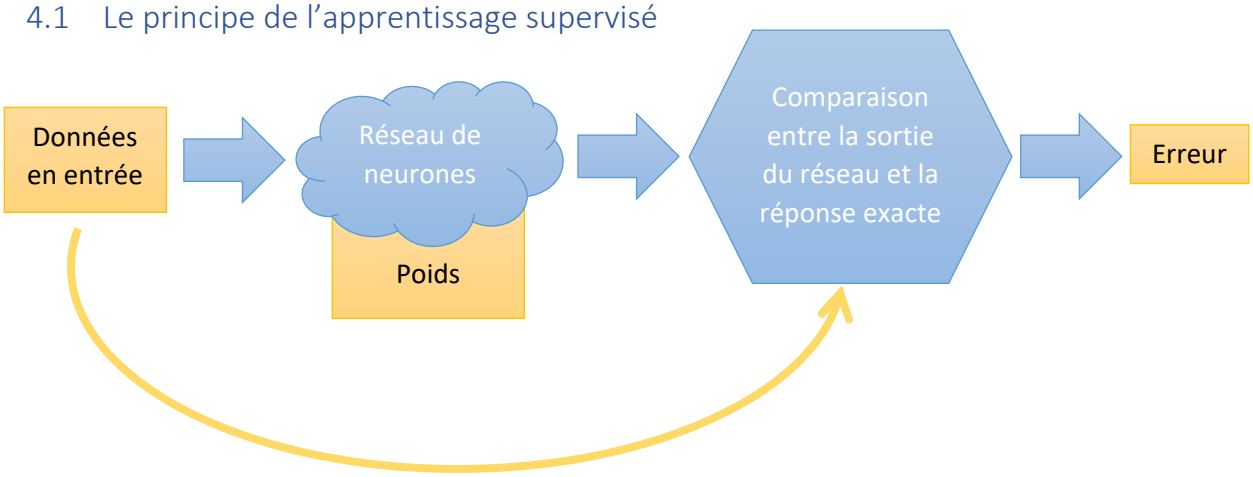
Vous allez obtenir des tracés correspondant à des fonctions linéaires par morceaux. Cela est normal, car la sortie d'un neurone + ReLU est une fonction linéaire par morceau et la combinaison linéaire de plusieurs fonctions linéaires par morceaux est une fonction linéaire par morceaux. De manière pragmatique, le tracé d'un neurone + ReLU crée une seule discontinuité (une cassure) sur le tracé. 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 :


1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

4 L'apprentissage

4.1 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 de $\mathbb{R}^n \rightarrow \mathbb{R}$ 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 cela sous-entend que 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.

Comment trouver des valeurs pour les poids qui permettent d'obtenir une erreur minimale ? Cette question est difficile car plusieurs minima locaux peuvent exister et lorsque nous détectons un minimum, nous ne savons pas s'il en existe un meilleur. Cependant, si le minimum trouvé est associé à une très faible erreur sur le set de validation, nous pouvons considérer que l'apprentissage s'est bien passé.

4.2 Une méthode d'optimisation naïve

Comment faire évoluer les valeurs W des poids pour diminuer l'erreur du réseau ? Générons au hasard un nouveau vecteur de poids W' proche de W. Calculons alors la nouvelle valeur de l'erreur associée à ces nouveaux poids W'. Si cette nouvelle erreur est plus faible, alors nous remplaçons les poids actuels par ceux de W'. En répétant ce processus, l'erreur diminue ! Voici l'algorithme de principe de cette méthode naïve :

```
Initialiser les poids W au hasard
ErrCourante = N(W,Data)   # N représente la fonction d'apprentissage
```

```
Tant qu'il nous reste du temps :
      Choisir des poids W2 proches des valeurs de W
      Err = N(W2,Data)
      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 à générer de nouveaux poids diminuant l'erreur courante. Lorsque le nombre de poids augmente, les choses deviennent plus difficiles.

4.3 Optimisation par la méthode du gradient

Heureusement, les mathématiques nous viennent en aide et nous donne un moyen de calculer une variation des poids W qui nous permettent de diminuer l'erreur quasiment à chaque fois. Et cerise sur le gâteau, PyTorch, ainsi que toutes les librairies de deep-learning, savent calculer ceci ! La variation idéale s'appelle le gradient, noté $grad\ W$ ou encore ∇W. L'algorithme de descente du gradient utilise la formule suivante :

$$W' = W - pas \times grad\ W$$

Le choix de la valeur du pas se fera pour l'instant par essais successifs. Si cette valeur est trop petite, l'erreur va diminuer très lentement. Si cette valeur est trop importante, l'erreur lorsqu'elle approche d'un optimum se mettra à osciller en faisant des rebonds, comme ici : 2, 1.8, 2.5, 2.3, 2.1, 1.5, 2.7, 2.3…

La méthode du gradient s'effectue en deux temps :

- En premier, on effectue la passe Forward où on évalue l'erreur courante en propageant les calculs de l'entrée du réseau vers la sortie. La librairie PyTorch en profite pour stocker des informations intermédiaires utiles au futur calcul du gradient.
- En second, on effectue la passe Backward où on calcule le gradient des poids en rétropropageant les calculs de la sortie du réseau vers l'entrée. Ainsi les gradients des poids présents à chaque étage de la chaîne de calcul sont déterminés.

Nous obtenons donc comme algorithme de principe pour la méthode du gradient sur un réseau :

```
Initialiser aléatoirement les poids W
Choisir une valeur pour le pas
Tant qu'il nous reste du temps :
      ErrCourante = N(W,Data)   # passe Forward
      grad = PyTorch_grad()     # passe Backward
      W = W – grad * pas        # mise à jour des poids
```

4.4 Exemple : mise en place de la méthode du gradient à pas constant

Nous allons programmer une version simple de l'algorithme de la descente du gradient sans utiliser les fonctions avancées de PyTorch. Pour cela, nous traitons une fonction d'apprentissage $f(x,a)$ où x représente les valeurs d'entrée et a un paramètre interne (poids) de la fonction.

Nous notons X le tenseur contenant les n valeurs en entrée et Ref le tenseur des n réponses de référence. Une fonction d'erreur $Err(estim,ref)$ calcule l'erreur entre l'estimation et la valeur de référence.

Il reste quelques points importants à rappeler sous PyTorch :

- Pour que PyTorch calcule le gradient du paramètre interne a, il faut explicitement déclarer que nous avons besoin de connaître son gradient. Pour cela, nous écrivons :

  a.requires_grad = True

- L'erreur totale correspond à la somme des erreurs associées à chaque échantillon. Une fois cette erreur connue et stockée dans un tenseur nommé ErrTot, il faudra utiliser la syntaxe suivante pour déclencher la passe Backward :

  ErrTot.Backward()

- Pour accéder à la valeur du gradient d'un tenseur T, on utilise la syntaxe :

  T.grad

- En appliquant la formule de l'algorithme de descente : a = a – pas * ∇a, on effectue une opération sur le tenseur a ce qui a pour effet de modifier son gradient ∇a durant ce calcul ce qui représente une source d'erreur. Ainsi, avant de faire évoluer les valeurs du tenseur a, il faut préciser au moteur PyTorch qu'il ne doit pas, durant un court instant, calculer d'informations relatives au gradient et à la rétropropagation. Pour cela, on utilise la syntaxe :

  with torch.no_grad() :

  qui désactive les calculs de gradient dans le bloc de code associé au with.

- Une fois tous les paramètres internes modifiés, il faut réinitialiser le gradient du tenseur a pour la prochaine passe Forward. Pour cela, on utilise la fonction suivante :

  a.grad.zero_()

Ainsi, nous pouvons construire l'algorithme de principe suivant :

```
Fonction Descente du gradient(f,Err,X,Ref,pas) :

    a = torch.FloatTensor(0)
    a.require_grad = True


    Tant qu'il nous reste du temps :

      # passe Forward
      ErrTot = torch.FloatTensor(0)
      Pour i = 0 à n-1
          y = f(X[i],a)
          erreur = Err(y,Ref[i])
          ErrTot += erreur
      print(ErrTot)

      # passe Backward
      ErrTot.Backward()


      # algorithme de descente du gradient à pas constant
      with torch.no_grad() :

         a -= pas * a.grad
         a.grad.zero_()
```

NB : on peut remarquer que le tenseur ErrTot est créé sans positionner son flag require_grad à True alors que ce tenseur intervient dans la rétropropagation. Cela est possible car la librairie PyTorch propage le flag require_grad. Ainsi nous avons :

- A la ligne : y = f(X[i],a), le tenseur a avec le flag require_grad à True se combine avec le tenseur X dont le flag est à False. Ainsi le tenseur résultat y a son flag require_grad à True.
- La tenseur erreur est créé à partir d'un tenseur Ref avec son flag à False et d'un tenseur y avec son Flag à True, il aura donc son flag à True.
- Le tenseur ErrTot est créé avec un flag require_grad à False. Cependant, à la ligne : « ErrTot += erreur », nous le combinons avec un tenseur avec son flag require_grad à True, ainsi son flag est modifié et passe à True.

4.5 Exercice 1 : apprentissage d'une fonction booléenne

On se propose de faire apprendre à une fonction le comportement de l'opérateur booléen $\neq$ (différent) donnant comme résultat :

$\rightarrow$ 1 si différent $\rightarrow$ 0 si égal

La fonction d'apprentissage choisie est :

$$f(x,y,a) = min(a \times |x-y|, 1)$$

avec a paramètre d'apprentissage et x et y les deux valeurs en entrée.

L'apprentissage doit s'effectuer sur le set d'échantillons suivant :

x	y	Réponse de référence
4	2	1
6	-3	1
1	1	0
3	3	0

Rappel : si l'apprentissage réussit, l'évaluation en dehors de ces échantillons peut cependant être erronée.

Voici les consignes :

- Prenez comme fonction d'erreur : $Err(estim,ref) = (estim - ref)^2$
- Dans cet exercice, vous devez mettre en place l'algorithme de descente du gradient avec un pas fixe correctement choisi.
- Le gradient sera calculé par la librairie PyTorch.
- Vous devez utiliser les fonctions fournies par PyTorch : torch.abs(.), torch.min(.), torch.sum(.) sinon la librairie ne sera pas en mesure d'effectuer la rétropropagation.
- Trouvez une valeur pour le pas qui permettent de converger en moins de 100 itérations.

Ouvrez et complétez le ficher d'exercice «Ex grad intro.py ».

Pour aller plus loin : on se propose de supprimer la boucle itérant sur les 4 échantillons. Pour cela, il faudra utiliser un unique tenseur stockant les informations des 4 expériences en parallèle. L'erreur totale sera calculée en sommant l'ensemble des valeurs du tenseur stockant les résultats.

5 Utilisation des optimiseurs PyTorch

5.1 SGD

La méthode Stochastic Gradient Descent correspond à notre méthode de la descente de gradient à pas fixe. Voici un extrait de la documentation PyTorch :

SGD

CLASS `torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0, weight_decay=0, nesterov=False)` [SOURCE]

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate

Le premier paramètre de SGD contient la liste des paramètres internes à optimiser. Le second paramètre intitulé « learning rate » correspond au pas dans l'équation de la descente du gradient. Que signifie le terme stochastique ? Pour l'instant nos échantillons sont de petite taille et nous pouvons calculer l'erreur sur la totalité des données en entrée. Cependant, lorsque la taille des données commence à devenir importante, une méthode consiste à sélectionner un sous-ensemble de l'ensemble des informations en entrée pour estimer le gradient. Cette sélection étant faite aléatoirement à chaque itération, on qualifie le gradient de stochastique.

5.2 Pour aller plus loin

Dans la méthode de descente du gradient, vous remarquez que l'on utilise un pas constant ceci pour tous les paramètres internes du système. Cependant, différents paramètres internes peuvent avoir besoin d'être mis à jour à des rythmes différents, en particulier dans les cas de données éparses, où certains paramètres sont plus activement impliqués dans l'extraction de caractéristiques comparés à d'autres. Pour améliorer cette faiblesse, l'optimiseur Adagrad introduit l'idée de mise à jour des paramètres internes avec un pas propre à chaque paramètre :

$$\beta_i^{t+1} = \beta_i^t - \frac{\alpha}{\sqrt{SSG_i^t + \epsilon}} * \frac{\delta L(X, y, \beta)}{\delta \beta_i^t}$$

L'indice i dans l'équation désigne le i-ème paramètre interne et l'indice t représente le numéro de l'itération. La grandeur SSG est la somme des gradients au carré pour le i-ème paramètre à partir de l'itération 0 jusqu'à l'itération t. La grandeur ϵ est utilisée pour indiquer une petite valeur ajoutée à SSG pour éviter une division par zéro. En divisant le pas noté ici α par la racine carrée de SSG, on s'assure que le taux d'apprentissage pour les paramètres changeant fréquemment diminue plus rapidement que le taux d'apprentissage pour les paramètres rarement mis à jour.

5.3 Les optimiseurs avancés

Dans la librairie PyTorch (https://PyTorch.org/docs/stable/optim.html), on peut trouver une dizaine d'optimiseurs avancés. Ils ont pour objectif d'apporter une amélioration par rapport à une faiblesse d'un optimiseur plus classique. L'inconvénient est l'apparition de multiples paramètres plus ou moins faciles à exploiter. La différence d'un optimiseur à l'autre est parfois subtile, chacun étant une variation ou le mélange d'un ou plusieurs voisins.

Si nous recherchons une faiblesse dans Adagrad, on peut facilement remarquer que le dénominateur sous le terme α (correspondant au pas constant) a une valeur qui ne cesse d'augmenter car ce terme cumule au fil des itérations de plus en plus de valeurs positives. Cela peut ainsi faire tendre le pas pondéré vers une valeur infinitésimale. Pour résoudre ce problème, on peut limiter l'emprise de ce terme en le moyennant avec le carré du dernier gradient connu :

$$SSG_i^t = \gamma * SSG_i^{t-1} + (1-\gamma) * (\frac{\delta L(X,y,\beta)}{\delta \beta_i^t})^2$$

Le facteur γ, compris entre 0 et 1, introduit un facteur d'affaiblissement (decay factor) des valeurs antérieures, ce qui évite ainsi la saturation à l'infini de ce terme. Cette idée sera le point de départ de plusieurs autres optimiseurs comme Adadelta, RMSprop ou Adam…

Par curiosité, ceux qui voudront tester un autre optimiseur que SGD, pourront prendre **Adam** car cet optimiseur est connu pour avoir souvent de très bonnes performances.

5.4 Mise en place de la forme standard

Beaucoup de tutoriaux sur internet ont tendance à instancier un réseau à partir d'un objet de type Sequential(…). Cet objet permet de donner la description de chaque couche du réseau pour le créer. Après cela, il est transmis à un objet optimiseur qui déroule la séquence de minimisation. Cette approche a le mérite de fournir des programmes très compacts. Elle est pratique et simple pour les non-informaticiens cherchant à mettre en place rapidement et facilement un réseau de neurones et ceci en un minimum de lignes de code.

Cependant, cette approche a un problème important : elle ne permet aucun débogage ou, en tout cas, elle le complexifie. Le suivi de l'évolution de la convergence n'est pas facilité non plus. Nous recommandons donc dans la suite des exercices de créer votre réseau en utilisant la forme type présentée sur la page suivante :

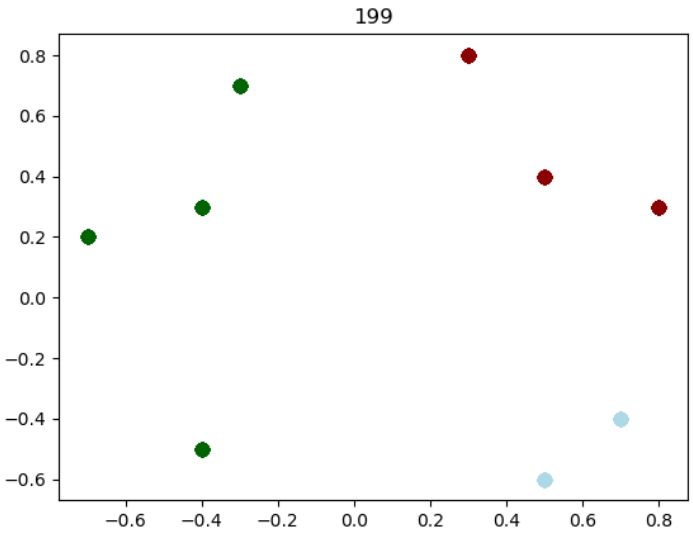
```
import torch
import torch.nn as nn
import torch.nn.functionnal as F
import torch.optim as optim


class Net(nn.Module) :
   def __init__(self):
       super().__init__()
       self.couche1    = …
       self.couche2    = …
       self.couche3    = …

   def forward(self,x):
       x = self.couche1(x)
       x = F.relu(x)
       x = self.couche2(x)
       x = F.max_pool2D(x,2)
       …
       return x

model = Net()
optim = optim.SGD(model.parameters(), lr=0.5)

for i in range(10000):
       optim.zero_grad()               # remet à zéro le calcul du gradient
       X = input(…)                    # choisit les data
       Y = answers(…)
       predictions = model(X)          # démarrage de la passe Forward
       loss = F.loss(predictions, Y)   # choisit une function de loss de PyTorch
       loss.backward()                 # effectue la rétropropagation
       optim.step()                    # algorithme de descente


# n'oubliez pas de rajouter des print pour vérifier un maximum d'info
```

5.5 Exercice 1 : apprentissage de catégories

L'exercice consiste à faire deviner à un réseau la catégorie d'un échantillon. Il existe trois catégories numérotées 0,1 et 2.

Chaque échantillon contient deux paramètres (p_1,p_2) et la catégorie associée. Nous représentons donc un échantillon à l'écran aux coordonnées (p_1,p_2) avec un code couleur correspondant à sa catégorie:

- 0 : rouge
- 1 : vert
- 2 : bleu



Nous allons pour commencer utiliser une unique couche Linear pour prédire trois scores. Chaque score correspond au score de chaque catégorie. Le plus haut score donne la catégorie prédite par le réseau.

Par exemple, un input est injecté dans la couche Linear. La sortie donne pour valeurs : (1,5,3). Le plus haut score est 5 et il correspond au score de la catégorie 1. Cette catégorie correspond donc à la catégorie prédite par le réseau pour cet échantillon.

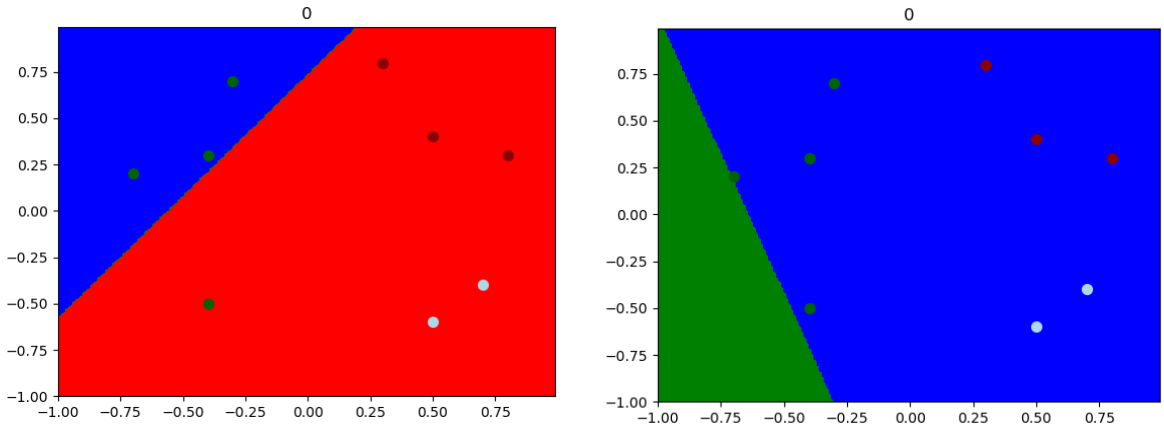
Ouvrez le fichier : « Ex apprentissage de forme V1.py ». Vous trouverez dans le code les diverses fonctions Matplotib utilisées pour le tracé.

Etape 1 : Mise en place de l'affichage interactif

Créez la couche Linear décrite ci-dessus. La première partie de l'exercice va consister à visualiser en couleur la catégorie de chaque point du graphique associée à la réponse de cette couche :

- Construisez un tenseur qui contient toutes les coordonnées (x,y) des pixels du graphique, pour cela, utilisez les deux tableaux XXXX et YYYY stockant ces valeurs. Si le graphique a une taille de 200x200 pixels, la taille du tenseur T correspondant sera : (200,200,2).
- Construisez le tenseur des scores contenant les scores de chaque pixel. Ce tenseur doit être le résultat du tenseur T injecté dans la couche Linear. Sa taille sera donc de (200,200,3) : 3 scores pour chaque pixel.
- Utilisez la fonction torch.argmax(axis= ???) pour retourner l'indice du score maximal. Ainsi, nous obtenons la catégorie prédite pour chaque pixel en repérant le maximum des trois scores.
- Convertissez ce tenseur contenant les catégories prédites en un tableau Numpy ayant les mêmes dimensions que le tableau XXXX.
- Complétez la fonction : ComputeCatPerPixel() servant à effectuer l'ensemble de ces traitements.

Au lancement du programme, la fonction ComputeCatPerPixel() retourne par défaut la catégorie 1, le fond est donc vert uniforme. Maintenant, vous devriez obtenir des écrans de ce type :



Etape 2 : Mise en place de l'apprentissage

Nous vous demandons pour l'instant de traiter les 10 échantillons itérativement. L'erreur totale correspondra à l'erreur générée par chaque échantillon. Créez votre réseau en le mettant sous la forme standard :

- Création d'une classe Python héritant de nn.Module()
- Création de la fonction forward qui prend un input *(x,y)* et retourne trois scores
- Création de la fonction d'erreur qui prend en paramètres :
  - Un tenseur $Scores$ contenant les trois scores des différentes catégories
  - La catégorie exacte de l'échantillon : id_cat
  - Pour calculer l'erreur, on utilise la formule suivante :

$$Err(Scores, id\_cat) = \sum_{j=0}^{2} \max(0, Scores[j] - Scores[id\_cat])$$

Comment interpréter cette formule ?

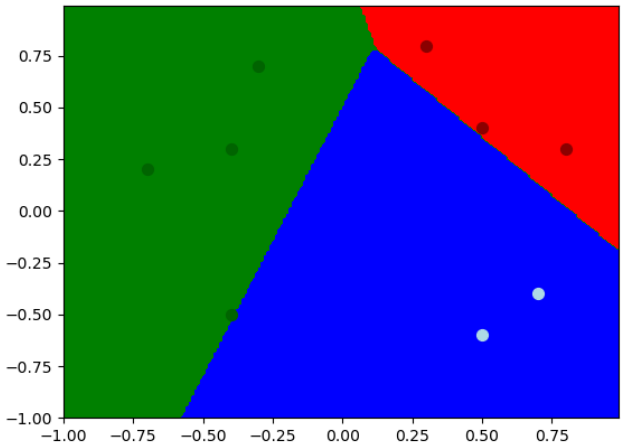
- La grandeur $Scores[j] - Scores[id\_cat]$ nous donne l'écart entre le score de la bonne catégorie et le score de la j-ième catégorie.
- Si j correspond à id_cat, la contribution à l'erreur vaut 0.
- Lorsque $j \neq id\_cat$:
  - Si cet écart est positif, cela sous-entend que $Scores[j] > Scores[id\_cat]$ et le plus grand score ne correspond pas à la bonne catégorie et ainsi l'erreur augmente.
  - Si cet écart est négatif, cela sous-entend que $Scores[j] < Scores[id\_cat]$ et le score de la bonne catégorie est supérieur au score courant. Dans ce cas, tout va bien et aucune contribution à l'erreur n'est ajoutée car la fonction max retourne la valeur 0.

Lors de la phase d'apprentissage, pour voir défiler l'animation où l'on voit les zones associées aux catégories s'adapter en live aux points d'entrée, votre code doit avoir la structure suivante :

```
for iteration in range(…):
   …
   Calcul …
```

```
…
# affichage
print("Iteration : ",iteration, " ErrorTot : ",ErrTot.item())
DessineFond()                       # dessine les zones
DessinePoints()                     # dessine les données sous forme de points
plt.title(str(iteration))           # insère le n° de l'itération dans le titre
plt.pause(2)                        # pause en secondes pour obtenir un refresh
plt.show(block=False)               # affichage sans blocage du thread
```

Voici le résultat obtenu par cette première version :

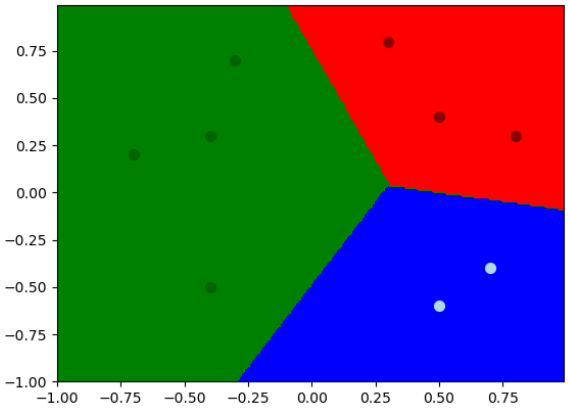


Etape 3 : amélioration de la fonction d'erreur

Vous remarquez que le résultat de l'étape 2 nous permet d'atteindre une erreur nulle car chaque point se situe dans la bonne zone de prédiction. Cependant, comme vous pouvez le remarquer, certains points se trouvent vraiment proches des frontières. Ainsi, si l'on donne en entrée un point très proche du point vert sur le bas du graphique, ce nouveau point peut se situer dans la zone bleue ce qui serait interprété comme une mauvaise prédiction. Nous aurions aimé une disposition des zones plus équilibrée où les points de référence se trouvent plus à l'intérieur et non aux bords. Pour cela, nous allons modifier la formule de calcul d'erreur pour qu'elle intègre un paramètre supplémentaire permettant d'augmenter l'écart entre le score de la bonne catégorie et les autres scores :

$$Err(Scores, id\_cat) = \sum_{j=0}^{nb\_cat-1} \max\left(0, Scores[j] - (Scores[id\_cat] - \delta)\right)$$

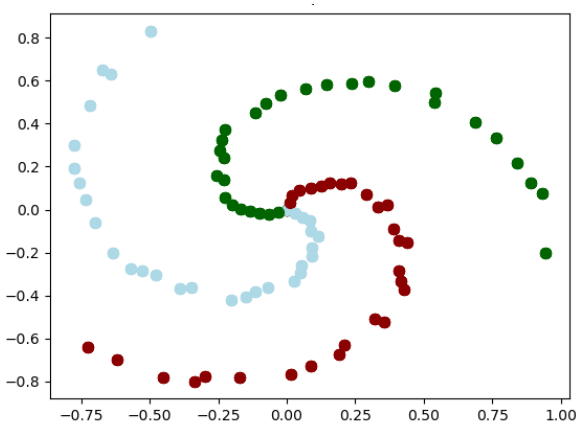
Le paramètre δ permet d'augmenter l'écart entre le score de la bonne catégorie et les autres scores. En effet, pour que l'erreur soit nulle, il faut que $Scores[j] - (Scores[id\_cat] - \delta) < 0$ soit $Scores[j] < Scores[id\_cat] - \delta$. Autrement dit, l'erreur devient nulle, lorsque l'écart entre le score de la bonne catégorie et les autres scores est d'au moins δ. Dans la suite, on pourra prendre δ = 1 ou 2.



Cette modification se traduit graphiquement par l'éloignement des points de référence des frontières, ces points se retrouvent maintenant plus à l'intérieur des zones de prédiction.

5.6 Exercice 2 : apprentissage de formes

On se propose de poursuivre l'exercice précédent sur un ensemble de points plus complexes décrivant une forme en hélice :



Etape 4 : mise en place d'un réseau à deux couches

Reprenez le code de l'exercice précédent et créez un nouveau fichier pour ne pas écraser votre travail. Remplacez la portion de code correspondant à la création des points par le code se trouvant dans le fichier « Ex Appr formes V2.py ».

On se propose de travailler avec 2 couches de neurones :

- Couche 1 : Linear + ReLU
- Couche 2 : Linear

La couche 2 est maintenant chargée de calculer les trois scores en sortie du réseau.

Quelques remarques :

- La fonction ReLU est fournie dans PyTorch : torch.nn.functional.relu
- Lancez un affichage toutes les 20 itérations, car ce dernier consomme des ressources

Pour la couche 1, comparez la qualité des résultats obtenus avec 10 / 30 / 100 / 300 / 500 et 1000 neurones. Choisissez un nombre de neurones minimal mais suffisant pour apprendre la forme de l'hélice.

Etape 5 : utilisation du broadcasting

Cette partie est difficile et nécessite que vous soyez à l'aise avec les manipulations de tenseurs.

L'objectif de cette étape est de retirer toute boucle de la passe Forward en utilisant le mécanisme de parallélisation des couches Linear et les opérations sur les tenseurs (max sur un tenseur par exemple).

Voici quelques conseils :

- Préparez un tenseur contenant les données d'entrée et un autre contenant les catégories de référence. Pour les construire, vous pouvez utiliser des boucles 😊 car cette opération est effectuée une unique fois en début de programme.
- Pour extraire les scores en sortie du réseau qui correspondent aux bonnes catégories (scores[Id_cat]), vous pouvez utiliser la syntaxe : Scores[I, CAT] où I est un tenseur d'indices d'entiers 64bits allant de 0 à nb_points-1 et où le tenseur CAT contient les catégories codées sous la forme 0 1 2.
- Pour créer le tenseur d'indices I, utilisez la syntaxe : torch.arange(0,nb,dtype=torch.int64)