Apprentissage profond avec PyTorch

Partie 1 : Les réseaux de neurones

1 *Objectifs*

- Comprendre le fonctionnement des réseaux de neurones (forward/backward)
- Les réseaux de neurones profonds

2 *Un premier réseau de neurones simple*

2.1 Introduction

Pour bien comprendre comment fonctionne l'entrainement d'un réseau de neurones, nous allons étudier l'exemple très simple suivant :

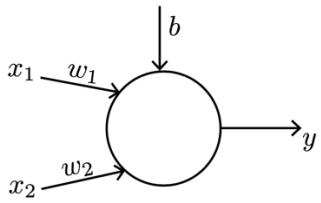
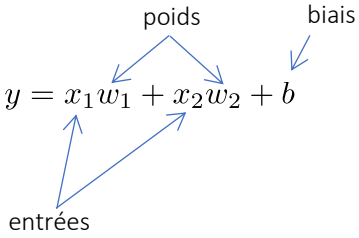


Figure 1 : Un premier réseau de neurones

La sortie de ce mini réseau de neurones est calculée par :



2.2 L'apprentissage dans un réseau de neurones

2.2.1 L'apprentissage par minimisation

Le principe de base de l'apprentissage supervisé est toujours le même : il consiste à ajuster les paramètres du système pour réduire une fonction de coût qui mesure l'erreur moyenne entre la sortie réelle du système et la sortie désirée, calculée sur un ensemble d'apprentissage. Réduire cette fonction de coût et entraîner le système sont une seule et même action.

Voici les différentes étapes de la phase d'apprentissage dans un réseau de neurones.

Initialisation :
On initialise les poids du réseau de neurones de façon aléatoire.

Boucle :

1. On propage les données d'entrées vers la sortie
2. On calcule l'erreur de sortie qui est égale à la différence entre la sortie propagée et la cible
3. On calcule le gradient de cette erreur sous la forme d'une fonction des poids du neurone et on ajuste les poids dans la direction qui réduit le plus cette erreur
4. Dans le cas où le réseau est composé de plusieurs couches :
    a. On propage ces erreurs de sorties vers l'arrière pour déduire les erreurs de la couche cachée
    b. On calcule les gradients de ces erreurs et on ajuste les poids de la couche cachée de la même manière

Nous allons reprendre les étapes précédentes en donnant le code correspondant. D'abord, il nous faut des données pour entraîner le réseau. Pour faire simple, nous allons générer des données « synthétiques » afin d'entraîner notre premier réseau, voici le code Python correspondant (extrait du code **`exercice1.py`**) :

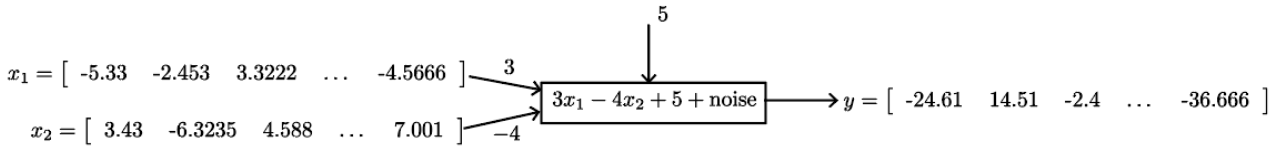
```
import numpy as np
import matplotlib.pyplot as plt

n_observations = 1000

max_val = 10
min_val = -10

x1 = np.random.uniform(min_val,max_val, size = (n_observations,1))
x2 = np.random.uniform(min_val,max_val, size = (n_observations,1))

inputs = np.column_stack((x1,x2))

noise = np.random.uniform(-0.1,.1,size=(n_observations,1))
targets = x1*3 - x2*4 + 5 + noise
```

Ce code permet de générer deux entrées x_1 et x_2 composées de valeurs aléatoires de distribution uniforme entre -10 et +10. On peut voir ces entrées comme deux vecteurs de dimension n_observations. Le biais b est égal à une constante 5. Pour que les données synthétiques soient réalistes on va ajouter un bruit (également de distribution uniforme) lors du calcul de la sortie :

$$\text{targets} = x_1 \times 3 - x_2 \times 4 + 5 + \text{noise}$$

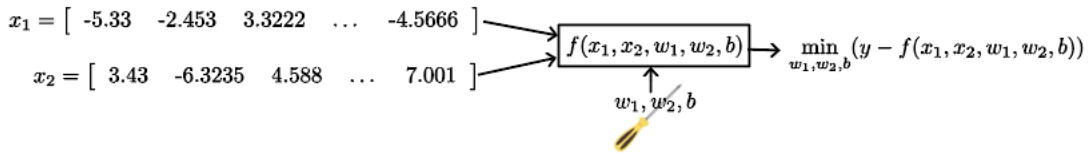
Les valeurs de targets correspondent à la sortie désirée. Il s'agit également d'un vecteur de dimension n_observations. Pour une raison que nous allons voir plus tard, nous agrégeons également les données des deux vecteurs x_1 et x_2 dans un tableau à deux dimension inputs (une matrice de taille n_observations x 2).

Voici un schéma qui résume cette étape :



2.2.2 Résolution du problème d'apprentissage

L'objectif de la phase d'apprentissage est compte tenu des données d'entrées et de la sortie désirée, il faut ajuster les paramètres du réseau (w_1, w_2, b) afin de trouver la fonction qui relie les données d'entrée aux données de sorties.



En Python, nous allons commencer par initialiser les paramètres du réseau de façon aléatoire :

```
weights = np.random.uniform(-0.2,0.2, size=(2,1))
bias = np.random.uniform(-0.2,0.2,size=1)
```

Nous allons maintenant exécuter la boucle de forward/backward afin d'optimiser les paramètres du réseau.

```
Learning_rate = 0.01

for i in range(100) :
    outputs = np.dot(inputs,weights) + bias   # (1)
    deltas = outputs – targets                # (2)
    loss = np.sum(deltas**2)/2/n_observations # (3)

    print(loss)                               # (4)

    deltas_scaled = deltas/n_observations     # (5)
    weights = weights – learning_rate*np.dot(inputs.T, deltas_scaled)   # (6)
    bias = bias – learning_rate*np.sum(deltas_scaled)                   # (7)
```

Pour entraîner le réseau nous allons faire 100 forward/backward. La ligne (1) permet de calculer la sortie du réseau de neurones pour les entrées synthétiques x_1 et x_2. Nous utilisons pour ce calcul les données agrégées dans la matrice `inputs` ainsi que le vecteur de poids `weights` et le `biais`. Pour accélérer le calcul nous utilisons la fonction de produit scalaire `np.dot` de numpy :

$$\begin{bmatrix} 5.33 & 3.43 \\ -2.453 & -6.3235 \\ 3.3222 & 4.588 \\ \vdots & \vdots \\ -4.5666 & 7.001 \end{bmatrix} \cdot \begin{bmatrix} w_1 & w_2 \end{bmatrix} + b = \begin{bmatrix} -5.33w_1 + 3.43w_2 + b \\ -2.453w_1 - 6.3235w_2 + b \\ 3.3222w_1 + 4.588w_2 + b \\ \vdots \\ -4.5666w_1 + 7.0001w_2 + b \end{bmatrix} = y$$

Sachant que les poids (w_1, w_2, b) du réseau ont été initialisés de façon aléatoire on obtient une sortie `output` probablement très éloignée de la sortie désirée `targets.` Une fonction coût/perte va nous permettre de quantifier à quel point le réseau de neurones est proche de la précision que nous souhaitons atteindre. Nous rappelons qu'en ajustant les paramètres (w_1, w_2, b) nous voulons diminuer l'écart. Il existe plusieurs fonctions coûts/pertes en fonction du problème que l'on traite (régression, classification, …). Dans notre exemple, il s'agit d'une régression. Nous allons prendre la fonction de perte que l'on appelle Erreur Quadratique Moyenne (Mean Squared Error (MSE) en anglais) :

$$L(w_1, \ldots, w_n, b) = \frac{1}{N} \sum_{i=1}^{N} (\hat{Y}_i - Y_i)^2 = \frac{1}{\text{n\_observations}} \sum_{i=1}^{\text{n\_observations}} (\text{outputs}_i - \text{targets}_i)^2$$

Cette fonction perte est calculée par les lignes (2) et (3). Très souvent pour des raisons de commodité mathématique une division par 2 est ajoutée lors du calcul de la fonction perte. Sans rentrer dans les détails mathématiques cette division par 2 permet de simplifier les calculs lors de la rétro-propagation du gradient. La ligne (4) permet d'afficher la valeur de la perte. L'objectif de l'apprentissage est de diminuer cette valeur de perte ce qui signifie que la sortie calculée `output` est de plus en plus proche de la sortie désirée `target`.

L'algorithme d'optimisation est codé par les lignes (5),(6) et (7). On va faire varier les paramètres (w_1, w_2, b) de façon à minimiser la fonction de perte. Cette variation des paramètres n'est pas réalisée n'importe comment. On se base sur l'algorithme de la descente de gradient :

$$\begin{aligned} w_{i+1} &= w_i - \eta \nabla_w L(w_i) = w_i - \eta \textstyle\sum_i x_i \delta_i \\ b_{i+1} &= b_i - \eta \nabla_n L(b_i) = b_i - \eta \textstyle\sum_i \delta_i \end{aligned}$$

Le paramètre η correspond au taux d'apprentissage (learning rate). Pour résumer cette phase d'optimisation :

1. calculer le coût ;
2. calculer le gradient ;
3. mettre à jour les paramètres en soustrayant le gradient multiplié par une constante η le taux d'apprentissage.

A force de répéter cette procédure, et à condition que η soit suffisamment petit, la procédure converge vers le minimum de la fonction de perte.

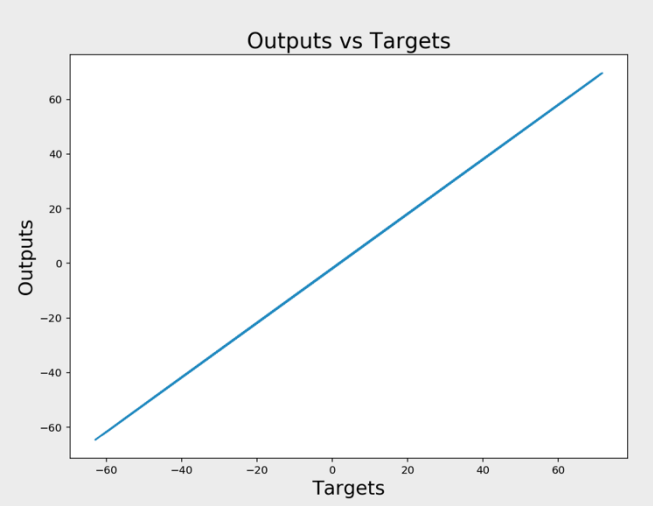
La fin du code permet d'afficher la valeur des paramètres obtenus après la phase d'optimisation. Nous traçons également la relation entre la sortie obtenue et la sortie désirée.

```
print(weights)
print(bias)

plt.plot(targets,outputs)

plt.xlabel('Targets',fontsize=18)
plt.ylabel('Outputs',fontsize=18)
plt.title('Outputs vs Targets',fontsize=20)

plt.show()
```

A la fin de la boucle d'optimisation, nous avons :

```
[[ 2.98997558]
 [-3.99510541]]
[3.19638801]
```

C'est-à-dire $w_1 = 2.9899, w_2 = -3.9951$ et $b = 3.19$. Nous sommes très proche des paramètres initiaux avec lesquels nous avons générés nos données synthétiques : $w_1 = 3, w_2 = -4$ et $b = 5$.



Travail à faire : Modifier le code Python **exercice1.py** afin de :

- Tracer la valeur de la fonction de perte en fonction nombre d'itération de la boucle d'optimisation.

Question : A partir de quel nombre d'itération la valeur de la fonction perte commence à converger.
Question : Recommencer la question précédente pour plusieurs valeurs du taux d'apprentissage. Que constatez-vous ?
Question : Ajouter une entrée de donnée x_3 (toujours uniforme dans l'intervalle [-10,10]) et modifier le calcul de la sortie désirée par :

$$\text{targets} = 2x_1 + 4x_2 - 3x_3 + 3 + noise$$

Modifier le script **exercice1.py** afin de réaliser l'apprentissage du réseau de neurones avec ce nouveau modèle.

2.2.3 Le gradient stochastique – mini-batch

Pour résumer ce que nous venons de mettre en œuvre pour optimiser les poids du réseau. Imaginons que nous ayons 1 millions de données, avec la descente de gradient il faudrait prédire pour chaque exemple la sortie (1), regarder la différence du vecteur prédiction avec le vecteur attendu (2) et refaire 1 million de fois la même opération pour calculer le coût moyen (3) :

```
outputs = np.dot(inputs,weights) + bias   # (1)
deltas = outputs – targets                # (2)
loss = np.sum(deltas**2)/2/n_observations # (3)
```

Il existe une méthode plus rapide, il s'agit du **gradient stochastique**. Le système tire juste un exemple de manière aléatoire dans l'ensemble d'apprentissage, il calcule le gradient du coût pour cet exemple et il fait un pas de gradient. Ensuite, il pioche un autre exemple, il calcule le gradient du coût pour ce nouvel exemple et fait un nouveau pas de gradient. On répète l'opération jusqu'à ce qu'on ne puisse plus descendre.
En pratique, au lieu de prendre un seul exemple pour effectuer le pas, on fait la moyenne du gradient sur un petit groupe d'exemples qu'on appelle « **mini-batch** ». A chaque pas, le gradient pointe dans une direction différente, ce qui amène le vecteur de paramètres à suivre une trajectoire erratique au cours de l'apprentissage. Généralement, l'apprentissage est plus rapide par « mini-batch » que de réaliser le calcul du coût sur l'ensemble d'apprentissage.

Travail à faire : reprendre le script précédent **exercice1.py** et modifier le calcul du gradient par une méthode par mini-batch.
Question : Observer l'évolution du coût *en fonction du nombre d'itération* et du *nombre d'exemple* dans le « mini-batch ». Vous devez observer des courbes moins « lisses » que précédemment.

Remarque : Le problème que nous avons traité dans cette première partie est un problème de régression linéaire. D'ailleurs, la fonction de coût considérée (l'Erreur Quadratique Moyenne) dans cette partie est très adaptée aux problèmes de régression. La structure du réseau était très simple (voire simpliste) avec une seule couche constituée d'un neurone avec deux entrées, une entrée biais et une sortie. De plus, la sortie de notre « réseau » de neurones était de type linéaire. Dans ce cas, nous ne pouvons que traiter efficacement des problèmes linéaires. Dans la partie suivante du TD nous allons considérer un modèle plus complet de réseau de neurones, il s'agit du Perceptron multi-couches.

3 Perceptron multi-couches (PMC)

3.1 Introduction

Dans cette partie, nous allons mettre en œuvre un réseau de neurones moins simpliste que précédemment. Voici l'architecture générale des réseaux PMC :

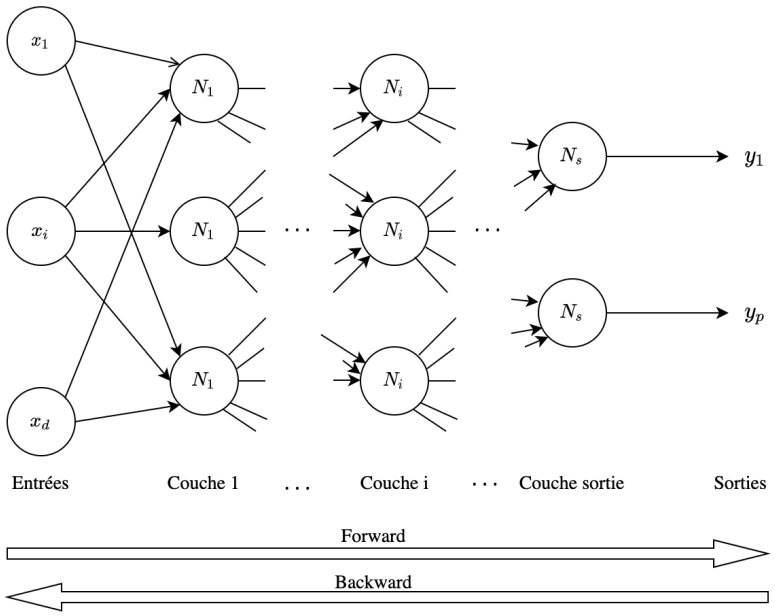


Figure 2 : L'architecture générale d'un PMC

Maintenant, faisons un zoom sur un des neurones qui composent un PMC :

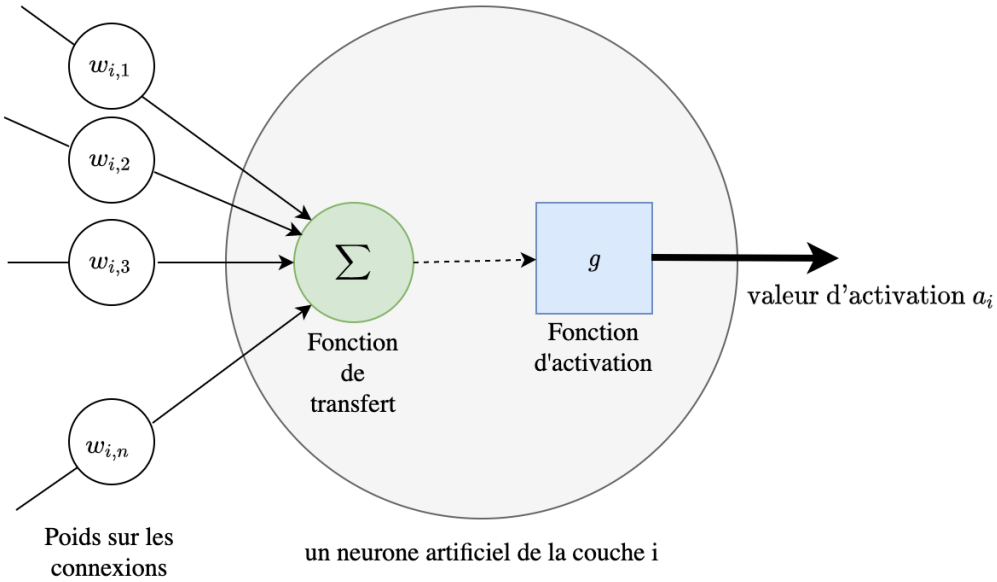


Figure 3 : Détails d'un neurone artificiel dans un réseau de neurones pour un perceptron multi-couches

L'architecture étant choisie, le calcul avec un PMC s'effectue des entrées vers la sortie en appliquant successivement les calculs de la couche 1 aux entrées, puis les calculs de la couche 2 aux sorties de la couche 1, et ainsi de suite jusqu'à obtenir les sorties. Pour une couche le calcul est donné par :

$$a_i = g(W_i.A_i + b)$$

où le produit correspond au produit scalaire et b à l'éventuel biais. Il existe un grand nombre de fonction d'activation $g(.)$ en fonction du problème considéré. On peut citer par exemple des fonctions d'activation linéaire (comme dans la première partie de ce TD), sigmoïde, Tanh, softmax, ReLU, …

Historiquement, la fonction sigmoïde est la fonction d'activation la plus ancienne pour les PMC, elle est de la forme

$$g(x) = \frac{1}{1 + e^{-x}}$$

Sa représentation graphique est donnée par

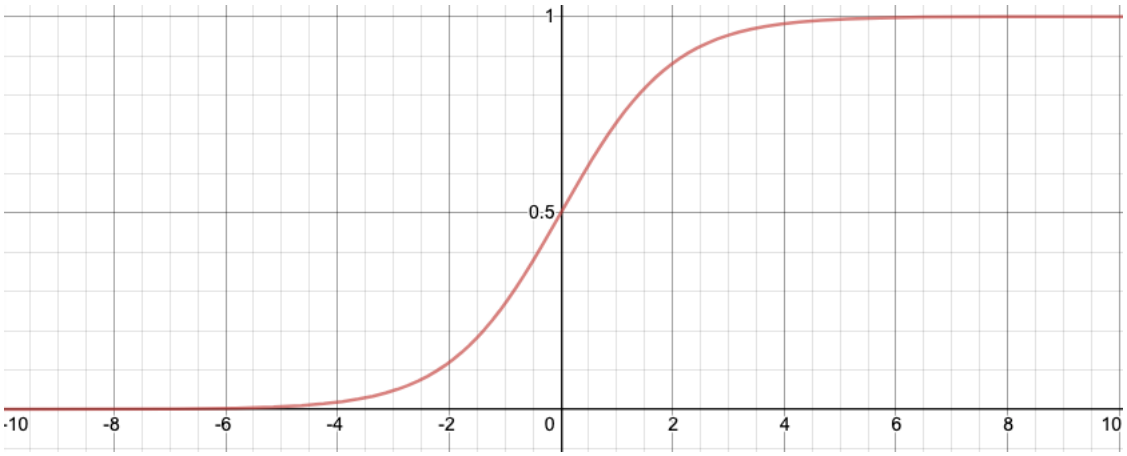


Figure 4: Fonction d'activation Sigmoïde

Nous pouvons voir que $g(x)$ agit comme une sorte de fonction « d'écrasement », qui condense notre sortie non bornée dans l'intervalle [0,1].

Les fonctions d'activation introduisent une non-linéarité dans le modèle, ce qui lui permet d'apprendre des relations entre les entrées et les sorties du réseau plus complexes.

Aujourd'hui cette fonction est quasiment obsolète et on lui préfère la fonction ReLU : $g(x) = \max(0, x)$.

Travail à faire : A l'aide d'un programme Python, tracer la fonction d'activation ReLU.

3.2 L'apprentissage

Comme précédemment, l'apprentissage consiste à estimer les paramètres $w_{i,n}$ par minimisation d'une fonction de perte. Cette fonction de perte permet de mesure les différences entre les sorties du PMC et les sorties désirées. Ces erreurs/différences sont corrigées via la rétro-propagation, les poids des neurones sont alors changés.

La difficulté dans les réseaux PMC c'est que l'on se retrouve avec une composition de fonctions (linéaires et non-linéaires) ce qui complique le calcul du gradient. Pour réaliser la rétro-propagation on utilise la règle de dérivation des fonctions composées apprise au Lycée et finalement malgré un grand nombre de couche, les équations se simplifient.

Travail à faire : Nous allons expérimenter les PMC avec scikit-learn. Dans scikit-learn, le perceptron multi-couches est implémenté avec les fonctions **`MLPClassifier`** pour la classification et **`MPLRegressor`** pour la régression. Nous allons travailler sur le jeu de données IRIS que nous avons déjà utilisé lors d'un précédent TP.
Commencer par exécuter le programme **`exercice2.py`** analyser son fonctionnement (attention il faut fermer la figure pour voir apparaître la suivante).

Pour répondre aux questions qui suivent, vous pouvez consulter la documentation en ligne : https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

Question : A quoi correspond une itération réalisée avec la variable `epoch`
Question : Quelle est la structure de ce PMC ? Combien de neurones dans chaque couche ?
Question : Donner la taille des matrices des poids entre la couche de sortie et la couche d'entrée.
Question : Essayez différents nombres de neurones dans la couche cachée. Observez les différences pendant et après l'entraînement.
Question : Essayez différentes fonctions d'activation telles que la fonction sigmoïde logistique ou tangente hyperbolique. Observez les différences sur les figures.

Question : Configurer la méthode d'optimisation avec une descente de gradient stochastique et configurez le taux d'apprentissage (l**earning_rate**) et le moment (**momentum**) en conséquence. Observez les différences sur les figures.

3.3 Prise en main de PyTorch

Avant d'aborder un exemple d'utilisation de la librairie PyTorch nous allons présenter rapidement la notion de tenseur. PyTorch nécessite l'utilisation des tenseurs quand on manipule des données.

3.3.1 Tenseurs

Dans PyTorch, un tenseur est une structure de données utilisée pour stocker et manipuler des données. Comme un tableau NumPy, un tenseur est un tableau multidimensionnel contenant des éléments d'un seul type de données. Les tenseurs peuvent être utilisés pour représenter des scalaires, des vecteurs, des matrices, et des tableaux à n dimensions et sont dérivés de la classe **`torch.Tensor`**. Les tenseurs présentent des avantages supplémentaires qui les rendent plus adaptés que les tableaux NumPy aux calculs d'apprentissage profond. Premièrement, les opérations sur les tenseurs peuvent être exécutées beaucoup plus rapidement grâce à l'accélération obtenue en utilisant des GPU. Deuxièmement, les tenseurs peuvent être stockés et manipulés en utilisant un traitement distribué sur plusieurs CPU et GPU et sur plusieurs serveurs. Ce qui permet de traiter des problèmes en grandes dimensions.

3.3.2 Régression linéaire avec PyTorch

Nous allons détailler pas à pas ce premier exemple afin de bien comprendre la structure des programmes PyTorch. L'ensemble du code commenté dans la suite est donné dans le programme **`exercice3.py`**. Commençons par créer un ensemble de données synthétiques simple avec une variable indépendante (x) et une variable dépendante (y), pour lesquelles il pourrait y avoir une relation de type linéaire entre x et y. Nous allons créer deux tenseurs (**`x_data`** et `y_data`) de taille [5,1] chacun, représentant ainsi 5 entrées et 5 valeurs de sortie. Voici le code qui permet de générer le jeu de données :

```
import torch
import matplotlib.pyplot as plt
import numpy as np
from torch.autograd import Variable

x = np.random.rand(100)
y = np.sin(x) * np.power(x,3) + 3*x + np.random.rand(100)*0.8

plt.scatter(x, y)
plt.show()

x = torch.from_numpy(x.reshape(-1,1)).float()
y = torch.from_numpy(y.reshape(-1,1)).float()
#print(x, y)

x_data = Variable(x)
y_data = Variable(y)
```

Ci-dessous une représentation du jeu de données :

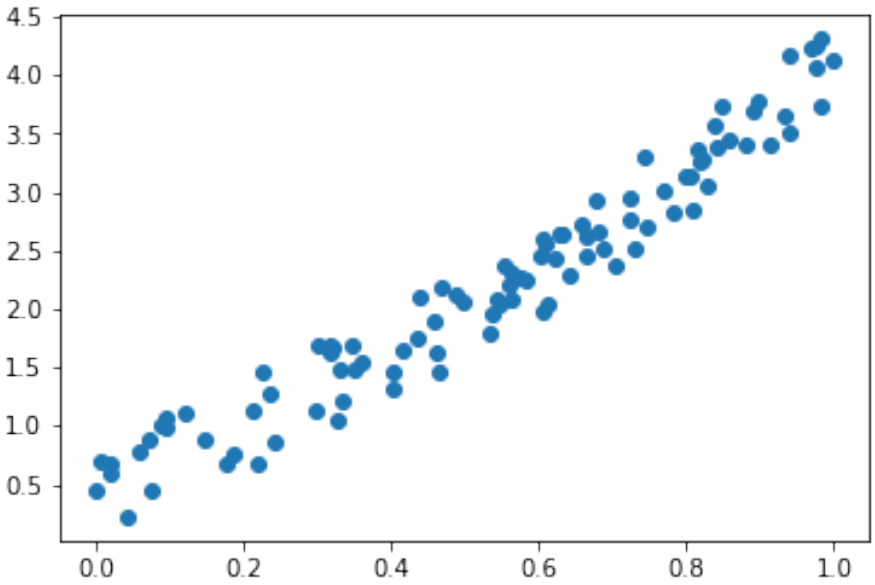


Figure 5: Données d'apprentissage

Les données sont prêtes. Le modèle que nous allons définir pour le réseau de neurones est un simple modèle linéaire composée d'une couche de neurones, cette couche va appliquer une transformation linéaire[1] aux données d'entrées.

```
class RegressionModel(torch.nn.Module):
    def __init__(self):
        super(RegressionModel, self).__init__()
        self.linear = torch.nn.Linear(1, 1)

    def forward(self, x):
        y_pred = self.linear(x)
        return y_pred

our_model = RegressionModel()
```

Pour le moment nous n'avons pas mis de fonction d'activation en sortie. C'est-à-dire que le réseau de neurones va approximer un modèle linéaire de la forme :

$$y = w_1 \times x + b_1$$

Comme nous n'avons qu'une seule variable d'entrée, nous nous attendons à apprendre deux poids, w_1 et b. Ces poids seront initialisés de façon aléatoire et PyTorch va réaliser l'apprentissage de ces poids. Voici la valeur des poids avant l'entraînement :

```
print(our_model)

for name, param in our_model.named_parameters():
    if param.requires_grad:
        print(name, param.data)
```

```
RegressionModel(
  (linear): Linear(in_features=1, out_features=1, bias=True)
)
linear.weight tensor([[-0.3764]])
```

[1] https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

```
linear.bias tensor([-0.4220])
```

Nous pouvons maintenant commencer le processus d'apprentissage des poids qui s'appuie sur la minimisation d'une fonction de perte en utilisant la méthode d'optimisation définie comme « optimizer ».

Nous devons préciser quel type de métrique nous souhaitons pour la fonction de perte. Dans cet exemple, nous prenons une fonction de perte du type : erreur quadratique moyenne (MSE). Cette fonction de perte est bien adaptée au problème de régression linéaire. Nous devons également choisir la méthode d'optimisation (« l'optimiseur ») pour le calcul des poids qui devra être appliquée. Nous prenons la fameuse méthode de la descente de gradient stochastique (SGD) en précisant la valeur du learning rate (lr), ici fixé à 0.05.

```
optimizer = torch.optim.SGD(our_model.parameters(), lr=0.05)
loss_func = torch.nn.MSELoss()
```

Le processus d'apprentissage va se dérouler en 3 étapes :

1. Propagation vers l'avant (Forward) : En utilisant la valeur des poids actuels, on calcule la sortie.
2. Calcul des pertes : Comparer les sorties avec les valeurs réelles.
3. Backpropagation : Utiliser les pertes pour mettre à jour les poids.

Ici, nous utilisons une boucle for pour itérer sur 50 époques. Au cours de ce processus, nous garderons également la trace des pertes afin de pouvoir visualiser ultérieurement l'évolution des erreurs au fil des époques. Nous affichons également, toutes les 10 époques, le résultat l'apprentissage, c'est-à-dire la droite de régression.

```
loss_history = []

for i in range(400):

   prediction = our_model(x_data)

   loss = loss_func(prediction, y_data)
   loss_history.append(loss)
   optimizer.zero_grad()
   loss.backward()
   optimizer.step()

   if i % 10 == 0:
       # plot and show learning process
       plt.cla()
       plt.scatter(x.data.numpy(), y.data.numpy())
       plt.scatter(x.data.numpy(), prediction.data.numpy())
       plt.text(0.5, 0, 'Loss=%.4f' % loss.data.numpy(), fontdict={'size': 10, 'color':
'red'})
       plt.pause(0.1)

plt.show()
```

Après 50 itérations, nous nous attendons à ce que l'erreur soit suffisamment faible pour produire un résultat décent.

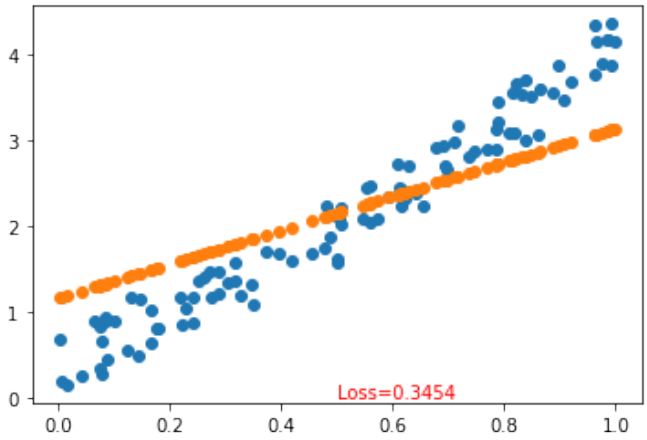


Figure 6 : résultat de la régression au bout de 50 époques

Pourtant, quand on regarde le résultat de la régression (Fig. 6), c'est loin d'être satisfaisant. Il faut encore itérer afin de diminuer plus largement l'erreur. On obtient à la fin des 400 époques le résultat de la Fig. 7.

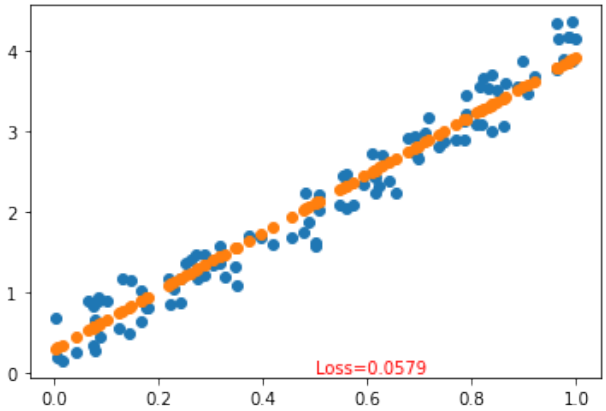


Figure 7: résultat de la régression au bout de 400 époques

Visualisons l'évolution de l'erreur en traçant le contenu du tableau **`loss_history`**. Attention, lossfunction(…) renvoit un tenseur. Pour pouvoir tracer les valeurs avec Matplotlib il faut convertir le tenseur en tableau numpy.

```
plt.plot([x.data.numpy() for x in loss_history], 'o-', markerfacecolor='w',
linewidth=1)
plt.plot()
plt.show()
```

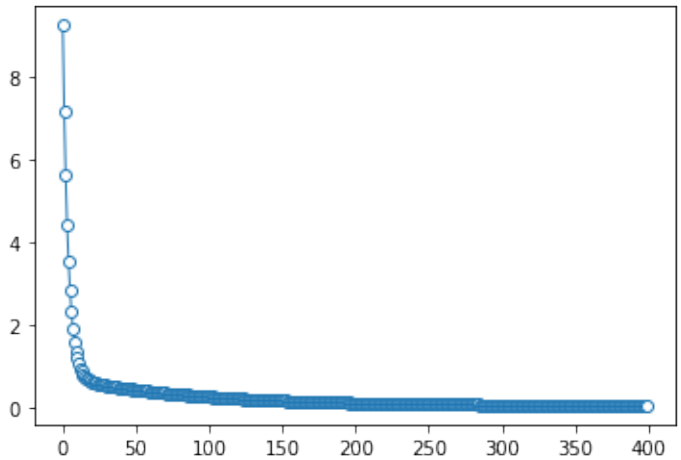


Figure 8 : L'erreur diminue tout au long du processus d'apprentissage.

Il ressort clairement de la Fig.8 que l'erreur diminue rapidement jusqu'à la cinquantième époque, après quoi on voit qu'il faut beaucoup plus d'époque pour diminuer l'erreur.

Travail à faire :

A présent, vous allez considérer un ensemble de données générées par le code qui suit :

```
x = np.random.rand(100)
y = x**4 + np.random.rand(100)*0.1

plt.scatter(x, y)
plt.show()
```

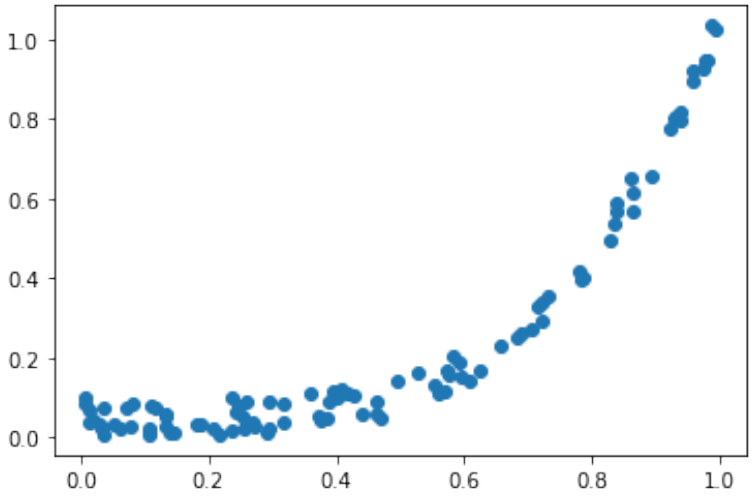


Figure 9: régression non-linéaire

Question : Est-ce que la droite de régression calculée par le réseau de neurones est toujours satisfaisante ?
Question : En jouant sur l'architecture du réseau de neurones :

```
class RegressionModel(torch.nn.Module):
    def __init__(self):
        super(RegressionModel, self).__init__()
        self.linear = torch.nn.Linear(1, ?)
        …

    def forward(self, x):
        y_pred = self.linear(x)
        …

        return ?

our_model = RegressionModel()
```

Essayer de trouver un modèle qui permet d'obtenir une régression satisfaisante, comme ci-dessous :

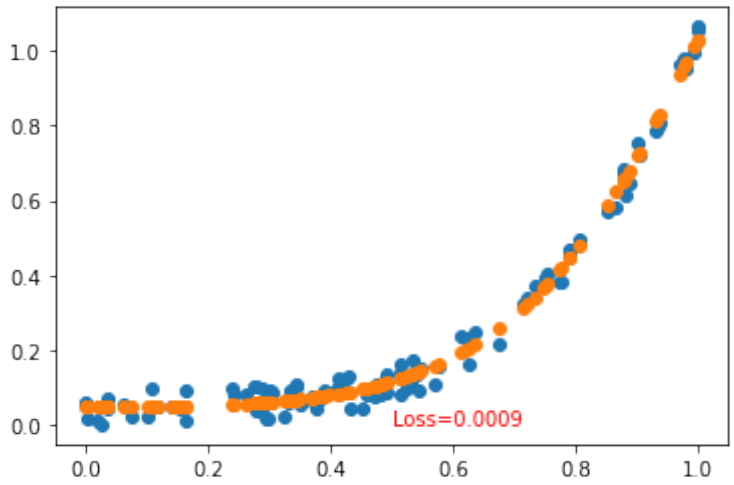


Figure 10 : régression non-linéaire

Indice : il va falloir rendre le modèle non-linéaire, par exemple en ajoutant une fonction d'activation Sigmoïde. Vous allez devoir aussi changer la taille en ajoutant des neurones. Voir la documentation en ligne : https://pytorch.org/tutorials/beginner/introyt/modelsyt_tutorial.html Vous pouvez aussi jouer sur la fonction d'optimisation du gradient. Pour le moment, nous avons pris la fonction « classique » du gradient stochastique (**`optimizer = torch.optim.SGD(our_model.parameters(), lr=0.05)`**) mais d'autres méthodes sont très efficaces, notamment pour éviter de tomber dans un minimum global : **`optimizer = torch.optim.Adam(our_model.parameters(), lr = 0.03)`**