Réseaux de neurones avec Keras sur PyTorch

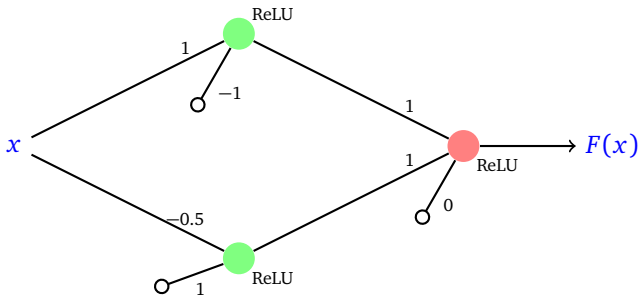
Objectifs [a] *:*

- *Création d'un réseau simple avec keras.*
- *Création et entrainement d'un réseau pour l'approximation de fonctions.*

a. Version 2023 adapté BUT 3 Informatique (Orléans), inspiré du livre d'Arnaud Bodin et François Recher

1. Implémenter un réseau de neurones avec KERAS

Keras est une bibliothèque open source pour l'apprentissage automatique. Nous l'utiliserons ici avec le moteur PyTorch, développé par Meta (Facebook). Le module *keras* permet de définir facilement des réseaux de neurones en les décrivant couche par couche. Pour l'instant nous définissons les poids à la main, en attendant de voir plus tard comment les calculer avec la machine (en Section 2). Pour commencer nous allons créer le réseau de neurones correspondant à la figure suivante :



1.1. Les imports KERAS

En plus d'importer le module `numpy` (abrégé par `np`), il faut importer le sous-module `keras` avec backend `pytorch` et quelques outils spécifiques :

```
import os
os.environ["KERAS_BACKEND"] = "torch"
import keras
from keras import layers
from keras import Sequential
```

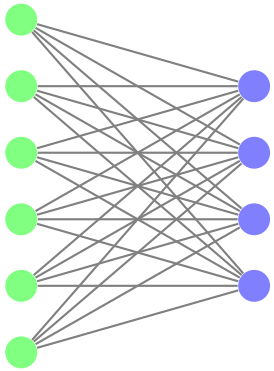
1.2. Implémentation des couches de neurones

Nous allons définir l'architecture d'un réseau très simple, en le décrivant couche par couche.

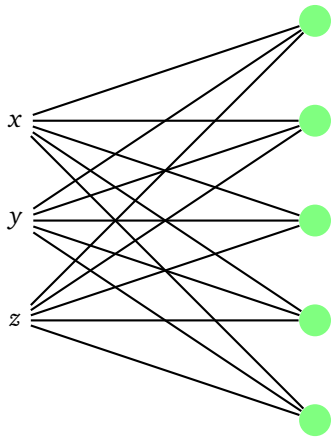
```
# Architecture du réseau
modele = Sequential()
# Couches de neurones
modele.add(layers.Input(shape=(1,)))
modele.add(layers.Dense(2, activation="relu"))
modele.add(layers.Dense(1, activation="relu"))
# Pour vérifier l'achitecture du réseau
modele.summary()
```

Explications :

- Notre réseau s'appelle `modele`, il est du type `nn.Sequential`, c'est-à-dire qu'il va être décrit par une suite de couches les unes à la suite des autres.
- Chaque couche est ajoutée à la précédente par `modele.add()`. L'ordre d'ajout est donc important.
- Chaque couche est ajoutée par une commande :
  `modele.add(nn.Linear(nb_neurones, activation=ma_fonction))`
- Une couche de type `Dense` signifie que chaque neurone de la nouvelle couche est connecté à toutes les sorties des neurones de la couche précédente.



- Pour chaque couche, il faut préciser le nombre de neurones qu'elle contient. S'il y a n neurones alors la couche renvoie n valeurs en sortie. On rappelle qu'un neurone renvoie la même valeur de sortie vers tous les neurones de la couche suivante.
- Pour la première couche, il faut préciser le nombre de valeurs en entrée (par l'option `input_dim = ...`). Dans le code ici, on a une entrée d'une seule variable. Sur la figure ci-dessous un exemple d'une entrée de dimension 3.



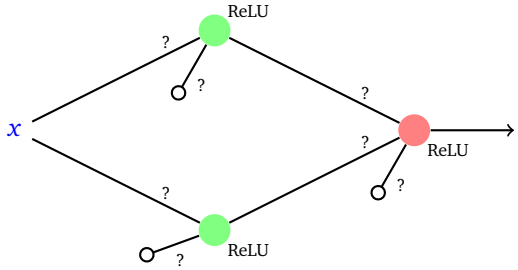
- Pour les autres couches, le nombre d'entrées est égal au nombre de sorties de la couche précédente. Il n'est donc pas nécessaire de le préciser.
- Pour chaque couche, il faut également préciser une fonction d'activation (c'est la même pour tous les

neurones d'une même couche). Plusieurs fonctions d'activation sont prédéfinies :

`'relu'` (ReLU), `'sigmoid'` (σ), `'linear'` (identité).

Remarquez que la fonction de Heaviside n'apparait pas, en effet malgré ses bonnes propriétés (sa simplicité en particulier) elle a un défaut essentiel, rédhibitoire pour la descente de gradient : elle n'est pas dérivable en tout point.

- Notre exemple ne possède qu'une entrée et comme il n'y a qu'un seul neurone sur la dernière couche alors il n'y a qu'une seule valeur en sortie. Ainsi notre réseau va définir une fonction $F : \mathbb{R} \to \mathbb{R}$, $x \mapsto F(x)$.
- Mais attention, pour l'instant ce n'est qu'un *modèle* de réseau puisque nous n'avons pas fixé de poids.



- Pour vérifier que tout va bien jusque là, on peut exécuter la commande

`modele.summary()`

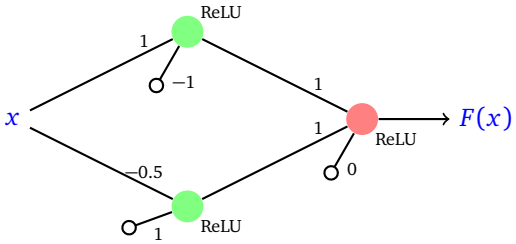
qui affiche un résumé des couches et du nombre de poids à définir.

1.3. Affectation des poids

Lors de la définition d'un réseau et de la structure de ses couches, des poids aléatoires sont attribués à chaque neurone. La démarche habituelle est ensuite d'entraîner le réseau, automatiquement, afin qu'il trouve de « bons » poids. Mais pour l'instant, nous continuons de fixer les poids de chaque neurone à la main.
La commande pour fixer les poids est `set_weights()`.

Voici la définition des poids de la première couche, numérotée 0 :

```
import numpy as np
# Couche 0 (1->2)
coeff = np.array([[1., -0.5]])   # (1, 2)
biais = np.array([-1, 1])        # (2,)
modele.layers[0].set_weights([coeff, biais])
```



Définissons les poids de la couche numéro 1 :

```
# Poids de la deuxième couche (2->1)
coeff2 = np.array([[1], [1]])  # (2, 1)
biais2 = np.array([0])         # (1,)
modele.layers[1].set_weights([coeff2, biais2])
```

Voici quelques précisions concernant la commande `set_weights()`. Son utilisation n'est pas très aisée.

- Les poids sont définis pour tous les éléments d'une couche, par une commande `set_weights(poids)`.
- Les poids sont donnés sous la forme d'une liste : `poids = [coeff,biais]`.
- Les biais sont donnés sous la forme d'un vecteur de biais (un pour chaque neurone).
- Les coefficients sont donnés sous la forme d'un tableau à deux dimensions. Il sont définis par entrée. Attention, la structure n'est pas naturelle (nous y reviendrons).

Pour vérifier que les poids d'une couche sont corrects, on utilise la commande `get_weights()`, par exemple pour la première couche :

`modele.layers[0].get_weights()`

Cette instruction renvoie les poids sous la forme d'une liste [coefficients,biais] du type :

```
[ [[ 1. -0.5]], [-1. 1.] ]
```

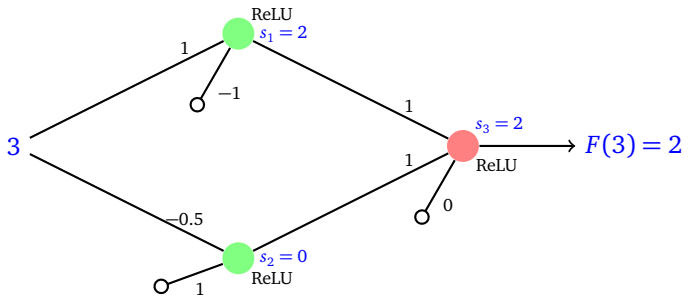
Astuce ! Cette commande est aussi très pratique avant même de fixer les poids, pour savoir quelle est la forme que doivent prendre les poids afin d'utiliser `set_weights()`.

1.4. Évaluation par le réseau d'une entrée

Comment utiliser le réseau ? C'est très simple avec `predict()`. Notre réseau définit une fonction $x \mapsto F(x)$. L'entrée correspond donc à un réel et la sortie également. Voici comment faire :

```
entree = np.array([[3.0]])
sortie = modele.predict(entree)
print(sortie)
```

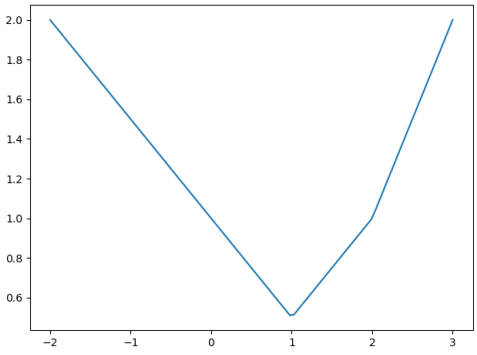
Ici `sortie` vaut `[[2.0]]` et donc $F(3) = 2$. Ce que l'on peut vérifier à la main en calculant les sorties de chaque neurone.



1.5. Visualisation

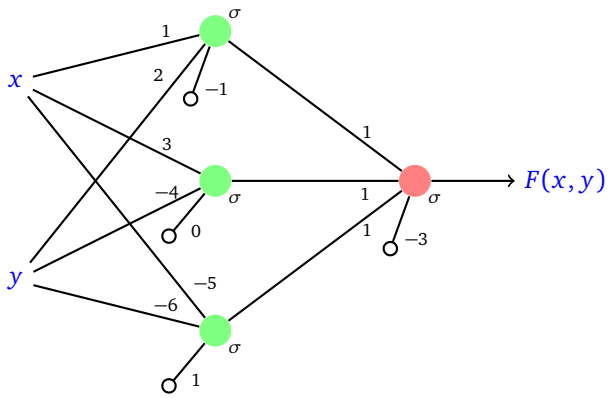
Afin de tracer le graphe de la fonction $F : \mathbb{R} \to \mathbb{R}$, on peut calculer d'autres valeurs :

```
import matplotlib.pyplot as plt
liste_x = np.linspace(-2, 3, num=100)
entree =  np.array([[x] for x in liste_x])

sortie = modele.predict(entree)

liste_y = np.array([y[0] for y in sortie])
plt.plot(liste_x,liste_y)
plt.show()
```



1.6. A vous !

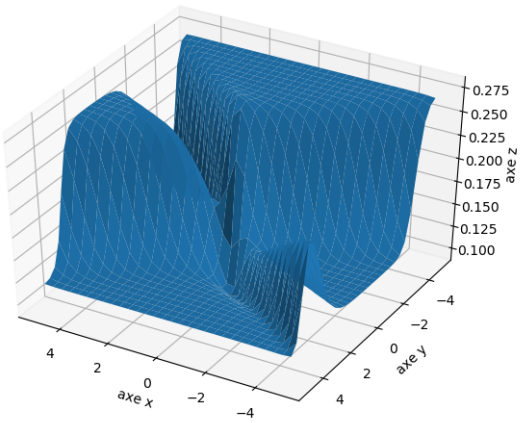
Créer avec keras, le réseau de neurones ci-dessous :



La première couche possède 3 neurones, chacun ayant deux entrées. La seconde couche n'a qu'un seul neurone (qui a automatiquement 3 entrées). La fonction d'activation est partout la fonction σ, est la fonction sigmoid, notée `sigmoid` dans `keras`. Pour construire ce réseau, vous utiliserez pas à pas la démarche utilisée précédemment. Vérifiez que $F(7,-5) \simeq 0.123$. Que vaut $F(1,-1)$?

Graphique. Voici comment tracer le graphe de $F : \mathbb{R}^2 \to \mathbb{R}$. C'est un peu trop technique, ce n'est pas la peine d'en comprendre les détails.

```
from mpl_toolkits.mplot3d import Axes3D

# --- Visualisation 3D du modèle ---

# Création d'une grille de valeurs pour les axes X et Y
VX = np.linspace(-5, 5, 20)
VY = np.linspace(-5, 5, 20)

# Construction du maillage 2D (toutes les
X, Y = np.meshgrid(VX, VY)

# Préparation des données d'entrée pour le
entree = np.c_[X.ravel(), Y.ravel()]

# Prédiction du modèle sur chaque point de
Z = modele.predict(entree).reshape(X.shape

# Création de la figure 3D
fig = plt.figure()
ax = plt.axes(projection='3d')

# Tracé de la surface prédite
ax.plot_surface(X, Y, Z, cmap='viridis')

# Affichage du graphique
plt.show()
```



2. Approximation d'une fonction à une variable

Le théorème d'approximation universelle pour les fonctions permet par l'existence d'un réseau de neurones qui imite n'importe quelle fonction. Prenons l'exemple de la fonction $f : [0,5] \to \mathbb{R}$ définie par

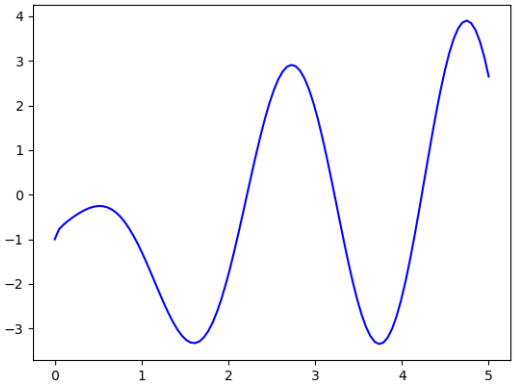
$$f(x) = \cos(2x) + x\sin(3x) + \sqrt{x} - 2$$

que l'on souhaite approcher par un réseau de neurones.

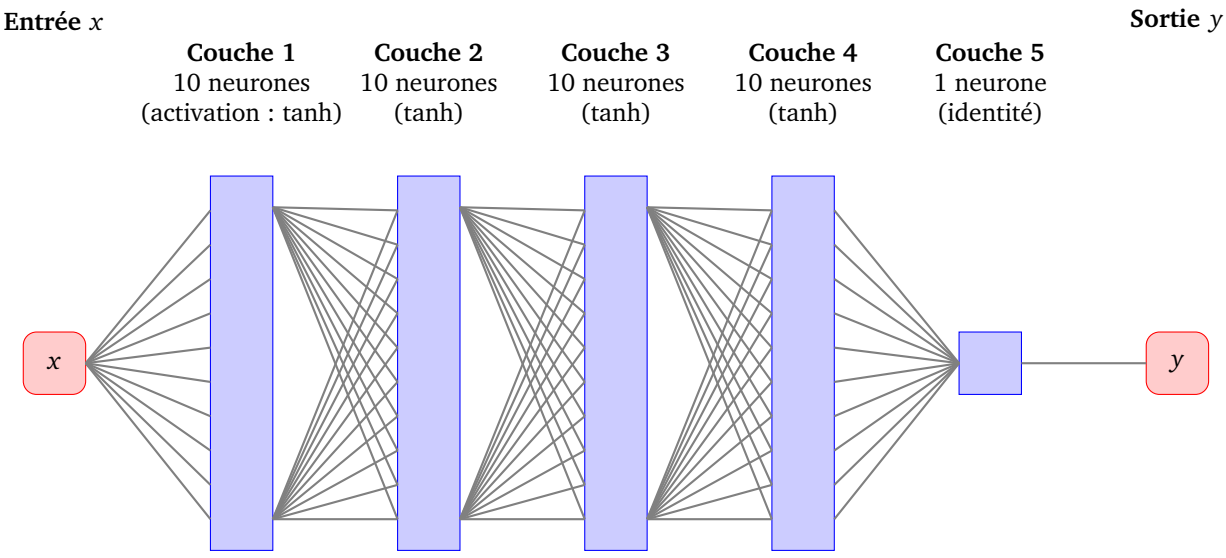
2.1. Données

Le but est donc de construire un réseau et de calculer ses poids afin d'approcher la fonction :

$$f(x) = \cos(2x) + x\sin(3x) + \sqrt{x} - 2 \qquad \text{avec} \quad x \in [0,5].$$



On divise pour cela l'intervalle de départ $[0,5]$ pour obtenir $N = 100$ abscisses x_i qui forment la première partie des données d'apprentissage `X_train`. On calcule ensuite les $y_i = f(x_i)$, ce qui donne 100 ordonnées qui forment l'autre partie des données d'apprentissage `Y_train`.
On propose un réseau avec 4 couches de 10 neurones, tous de fonction d'activation la fonction tangente hyperbolique, et d'une couche de sortie formée d'un seul neurone de fonction d'activation l'identité. L'entrée et la sortie sont de dimension 1. La fonction associée au réseau est donc $F : \mathbb{R} \to \mathbb{R}$. On souhaite calculer les poids du réseau de sorte que $F(x) \simeq f(x)$, pour tout $x \in [0,5]$.



2.2. Programme

```
# Partie A. Données

# Fonction à approcher
def f(x):
    return np.cos(2*x) + x*np.sin(3*x) + x**0.5 - 2

a, b = 0, 5                 # intervalle [a,b]
N = 100                     # volume de données
X = np.linspace(a, b, N)    # abscisses
Y = f(X)                    # ordonnées

# transforme X en une colonne, comportant le bon nombre de ligne (i.e. "-1"), idem pour Y
X_train = X.reshape(-1,1)
Y_train = Y.reshape(-1,1)

# Partie B. Réseau


# Modèle : 1 entrée -> 4 couches tanh,  p neurones par couche, 1 sortie linéaire
p = 10
modele = Sequential([
    layers.Input(shape=(1,)),
    layers.Dense(p, activation='tanh'),
    layers.Dense(p, activation='tanh'),
    layers.Dense(p, activation='tanh'),
    layers.Dense(p, activation='tanh'),
    layers.Dense(1, activation='linear'),   # linear par défaut, mais on explicite
])

# --- Compilation : descente de gradient
modele.compile(loss='mean_squared_error')

# --- Résumé ---
print(modele.summary())

#Entraînement du modèle ; verbose=1 affiche la progression et la loss à chaque epoch
history = modele.fit(X_train, Y_train, epochs=4000, batch_size=len(X_train), verbose=0)
# Attention ici comme batch_size=len(X_train) c'est une descente de gradient classique
# (non stochastique)


# Affichage de la fonction et de son approximation :
# A exécuter sur une cellule indépendante
Y_predict = modele.predict(X_train) # calcul de la prédiction
plt.plot(X_train, Y_train, color='blue')
plt.plot(X_train, Y_predict,  color='red')
plt.show()
```

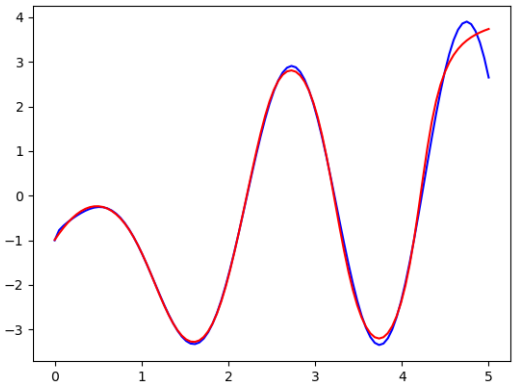
```
# Affichage de l'erreur au fil des époques
plt.plot(history.history['loss'])
plt.show()
```

2.3. Explications

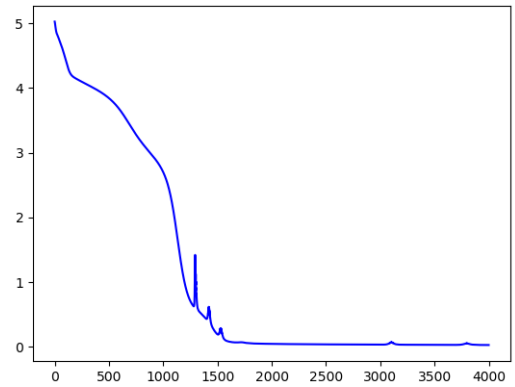
La fonction d'erreur est ici l'erreur quadratique moyenne. Notre méthode de gradient débute avec un taux d'apprentissage $\delta = 0.001$ (variable `lr` pour *learning rate*). Le momentum est une autre possibilité d'optimisation dont nous ne parlerons pas dans un premier temps.
On lance l'apprentissage, c'est à dire le calcul des poids :

```
history = modele.fit(X_train, Y_train, epochs=4000)
```

Ici on effectue 4000 époques (on utilise 4000 fois les données). La taille de l'échantillon est égale à la taille des données N, c'est une descente de gradient classique, il y a donc aussi 4000 étapes dans la descente de gradient. La variable `history` contient l'historique des valeurs renvoyées par la fonction `fit()` et permet par exemple d'afficher la valeur de l'erreur au fil des époques.
Le réseau fournit donc une fonction $F : \mathbb{R} \to \mathbb{R}$ qui approche correctement la fonction f sur l'intervalle $[0, 5]$.



Voici le comportement de l'erreur au fil des époques.



L'erreur n'est pas partout une fonction décroissante, c'est le cas lorsque le taux d'apprentissage est trop grand. D'où l'intérêt de faire diminuer ce taux d'apprentissage au fil des époques.

2.4. A vous !

Nous voyons que nous pouvons jouer ici sur 4 paramètres :

1. Le nombre de couches,
2. Le nombre de neurones p,
3. Le pas d'apprentissage ou `learning_rate` noté δ en cours,
4. Le nombre d'époques ep,

et nous conserverons la fonction d'activation `tanh`. En plus de cela nous pouvons aussi choisir différentes méthode d'optimisation et nous verrons aussi qu'un travail sur les données (normalisation) est d'une grande importance.

Exercice 1
1) Exprimez, en fonction des paramètres pertinents, le nombre total de neurones utilisés et le nombre de paramètres à estimer par la descente de gradient.
2) Quelle est la différence entre `Y_train` et `Y_predict` ?
3) Que trace `plt.plot(X_train, Y_train, color='blue')` ?
4) Quelle est la fonction python qui effectue la descente de gradient ?

Exercice 2 Dans cet exercice on fait varier les paramètres du réseau afin d'observer leur influence sur la qualité d'apprentissage et sur le temps d'exécution. Pour chaque cas, on tracera la fonction f et celle obtenue par le réseau, ainsi que la courbe de la loss. À chaque essai vous devez modifier le modèle (nombre de neurones et/ou nombre de couche si besoin), puis réinitialiser les paramètres et modifier (éventuellement) les méthodes d'optimisation pour pouvoir comparer.
1) Prendre un petit réseau (2 neurones par couche : `p=2`), puis faire évoluer le nombre d'epoch : `ep=50`, `ep=400` et `ep=1000`. Que remarquez-vous ?
2) Ici on va progressivement augmenter le nombre de neurones par couche : prendre `p=5` puis `p=10` on prendra `ep=1000` puis `ep=4000`. Que remarquez-vous ? La loss diminue-t-elle ? Que devient le temps d'exécution ?
3) Que se passe-t-il lorsque l'on diminue le taux d'apprentissage (`learning_rate`), si on prend `learning_rate` $= 0.001$ au lieu de 0.01 (par défault), l'apprentissage est-il plus stable, plus lent ?
Pour cette question nous utiliserons la descente de gradient stochastique (SGD) : voici la partie du code à modifier, tester d'abord

```
modele.compile(optimizer=optimizers.SGD(learning_rate=0.01),loss='mean_squared_error')
```

puis

```
modele.compile(optimizer=optimizers.SGD(learning_rate=0.001),loss='mean_squared_error')
```

4) En testant un optimiseur plus avancé, `Adam` ici, que constatez-vous sur la rapidité et la stabilité de la convergence ?

```
modele.compile(optimizer=optimizers.Adam(learning_rate=0.01),loss='mean_squared_error')
```

5) Vous avez pu remarquer parfois que l'apprentissage sature rapidement (par exemple, le début de la fonction est bien reproduit mais pas la fin). Améliorons cela *sans aucun coût supplémentaire* : à la place de faire un entraînement avec `X_train`, nous le faisons à partir de sa version centrée-réduite :

```
X_mean, X_std = X_train.mean(), X_train.std()
Xn = (X_train - X_mean) / X_std
modele.fit(Xn, Y_train, epochs=..., batch_size=..., verbose=1)
Y_pred = modele.predict((X_train - X_mean)/X_std)
```

Faites varier les hyper-paramètres (`p et ep`), que remarquez vous ? Pourriez vous l'expliquer ?

Exercice 3 : Challenge – Trouver le plus petit réseau efficace.
En gardant la même fonction f, recherchez empiriquement le plus petit réseau de neurones (c'est-à-dire le moins de couches et de neurones par couche) et le plus petit nombre d'epochs qui permettent encore de reproduire correctement la fonction sur tout l'intervalle.

Indications :

- Commencez par quelques neurones cachés et 2 ou 3 couches, puis augmentez progressivement.
- Tracez à chaque fois la prédiction et observez la convergence.
- Comparez également la stabilité de la loss selon le nombre d'epochs.

2.5. Une variante, ressemblant plus à des vraies données

Pour obtenir une suite de données plus réaliste (moins régulière que la fonction f), nous décidons de bruiter les données. Pour cela nous allons ajouter aux valeur `Y_train` une valeur tirée aléatoirement selon une variable aléatoire gaussienne (le bruit).

```
import os
os.environ["KERAS_BACKEND"] = "torch"
import keras
#On ajoute un bruit selon une  loi gaussienne
#de paramètres "mu" et "sigma" :
def gaussnoise(mu, sigma, size):
   return np.random.normal(mu, sigma, size)
```

```
import os
os.environ["KERAS_BACKEND"] = "torch"
import keras
# Création du jeu de données
a, b = 0, 5
N = 60
X = np.linspace(a, b, N)
#print(X)
Y = f(X)
Ynoisy=f(X)+gaussnoise(0,0.3,N)
#print(Y)
X_train = X.reshape(-1,1)
Y_train = Ynoisy.reshape(-1,1)

# Tracer du nuage de points des données bruitées color='blue'
plt.plot(X_train, Y_train,"ob")
```

Faites une analyse similaire à celle du paragraphe précédent, faites attention au sur-apprentissage !