Travaux Pratiques — Cours d'Apprentissage Profond

Aurélien Plyer & Kevin Helvig

26 septembre 2025

Table des matières

Session 1 — Premiers pas, MLP et PINNs

Cette première série de travaux pratiques a pour but d'amener les étudiants à produire eux-mêmes les scripts de référence tout en bénéficiant d'un guidage précis et d'extraits de code à compléter. Chaque sous-TP correspond à un livrable autonome (notebook ou script documenté) accompagné de figures ou de commentaires analytiques.

0.1 Préparation de l'environnement de travail

Deux options sont possibles pour réaliser les TPs de la session 1 :

— Google Colab (simple et adapté si vous n'avez pas de droits d'installation),
— Installation locale Jupyter avec Miniconda (recommandée si vous pouvez installer des logiciels sur votre machine).

Option A — Google Colab Google Colab fournit gratuitement un environnement Jupyter hébergé chez Google. À chaque lancement, une machine virtuelle GPU/CPU est réservée pendant quelques heures : elle est non permanente (toute modification locale est perdue après déconnexion prolongée ou fin de session). Travaillez donc depuis Google Drive ou sauvegardez vos scripts vers un dépôt distant.

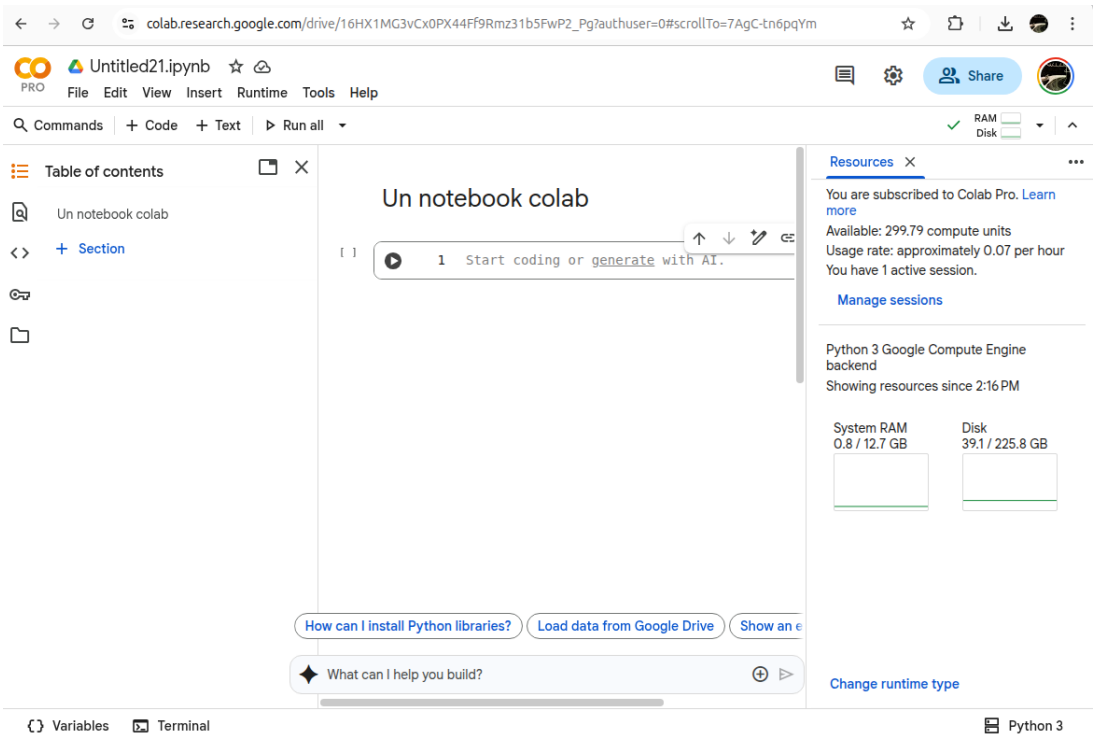

FIGURE 1 – Interface de création de notebook sur Google Colab.

1. Ouvrir Google Colab et créer un nouveau notebook dans l'espace Drive souhaité (fichier → *Nouveau notebook*).

2. (Facultatif) Activer le GPU : menu *Exécution* → *Modifier le type d'exécution* → *Accélérateur matériel : GPU*.
3. Vérifier l'accès GPU et la version de PyTorch dans une cellule Colab :

```
import torch, sys
print("Python:", sys.version)
print("Torch:", torch.__version__)
print("CUDA dispo:", torch.cuda.is_available())
```

4. Importer ou copier-coller les bouts de code dans des cellules Colab afin d'exécuter les blocs pas à pas.
5. Montez Google Drive pour accéder à vos données persistantes :

```
from google.colab import drive
drive.mount("/content/drive")  # Choisir un répertoire, puis
   autoriser l'accès
```

Une fois le montage effectué, vos fichiers Drive deviennent accessibles sous `/content/drive/MyDrive/`. Il est important d'avoir a l'esprit que le stoquage Drive est pas forcément co-localisé avec la machine virtuel Colab que vous utilisez pour éviter de faire traverser l'océan a vos fichiers une première étape est de les copier depuis votre drive vers `/content/` pour accélérer leur utilisation puis, lors de la fin de votre travail effectuez l'inverse et sauvegardez les résultats importants sur Drive avant de fermer la session (les ressources locales sont effacées après expiration de la machine virtuelle).

```
# Exemple d'utilisation après montage
import shutil
shutil.copy("/content/drive/MyDrive/coursDL/code/tp_1_1_perceptron.
   py",
            "/content/")
```

Pour libérer proprement le montage, exécuter :

```
from google.colab import drive
drive.flush_and_unmount()
```

Option B — Installation locale (Jupyter + Miniconda) 1) Installer Miniconda

— Télécharger Miniconda depuis docs.conda.io (choisir l'installateur pour votre OS).
— Sous Linux/macOS, lancer l'installateur en ligne de commande, puis initialiser conda (accepter la question `conda init`) :

```
bash Miniconda3-latest-Linux-x86_64.sh
# Relancer le terminal après l'installation (ou 'source ~/.bashrc')
```

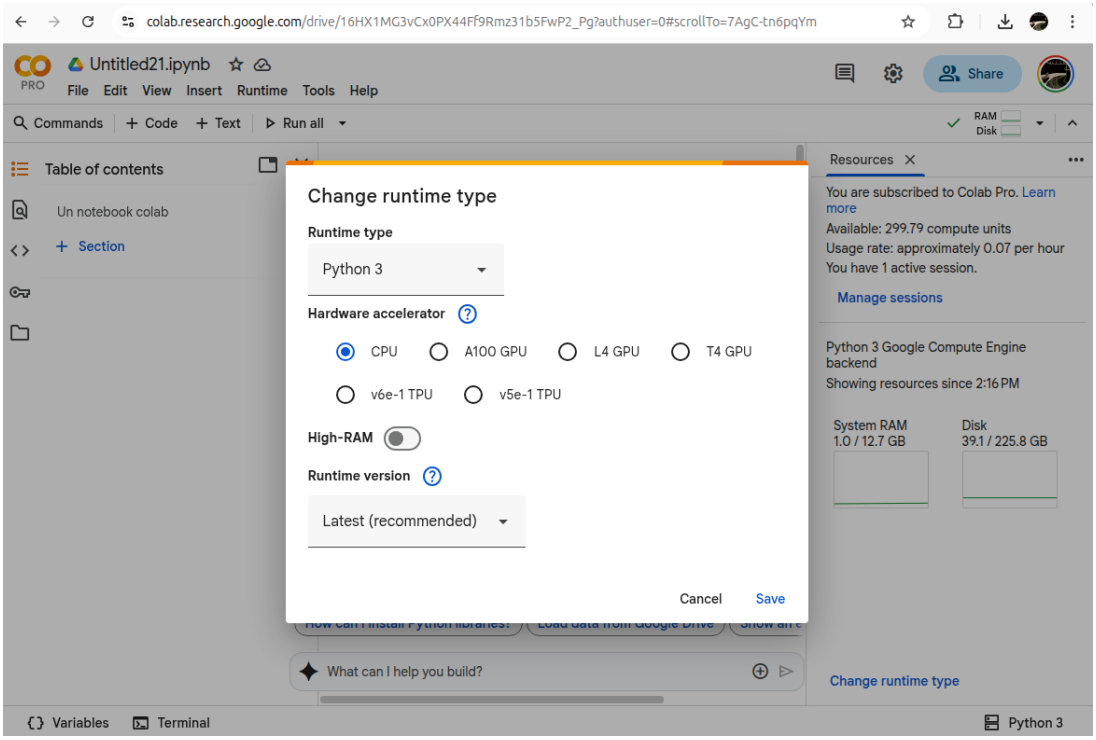

FIGURE 2 – Exécution d'un bloc PyTorch sur Colab avec GPU activé.

Sous Windows, exécuter l'installeur graphique (PowerShell/Invite de commandes non requis) et accepter `Add Miniconda to my PATH` si proposé.

2) Créer l'environnement « coursDL »

Le dépôt fournit deux fichiers d'environnement dans `code/` : `coursDL_cpu.yml` (par défaut) et `coursDL_gpu.yml` (si vous avez un GPU CUDA opérationnel). Vous pouvez créer un environnement nommé `coursDL` à partir de l'un ou l'autre :

```
# Ouvrir un terminal puis :
cd /chemin/vers/le/depot

# Option CPU (recommandée par défaut)
conda env create -f code/coursDL_cpu.yml -n coursDL

# Option GPU (si CUDA + drivers OK)
conda env create -f code/coursDL_gpu.yml -n coursDL
```

Activation et enregistrement du noyau Jupyter :

```
conda activate coursDL
python -m ipykernel install --user --name=coursDL --display-name "
    Python (coursDL)"
```

3) Lancer Jupyter et choisir le noyau

```
# Depuis la racine du dépôt (pour voir code/ et TPs/)
```

```
jupyter lab    # ou : jupyter notebook
```

Dans le notebook, choisir le noyau « Python (coursDL) » via le menu *Kernel / Noyau*.

4) Vérifier PyTorch et le GPU (facultatif)

```
import torch
print("CUDA disponible:", torch.cuda.is_available())
print("Nom du GPU:", torch.cuda.get_device_name(0) if torch.cuda.
    is_available() else "(CPU)")
```

5) Exécuter un script en ligne de commande

```
# Exemple : lancer le perceptron de la session 1
conda activate coursDL
python code/tp_1_1_perceptron.py
```

0.2 TP 1.0 — Reprises de Python et NumPy

Objectifs. Manipuler les structures de base de Python, maîtriser les tableaux NumPy et Matplotlib, puis tester un premier calcul PyTorch en comparant CPU et GPU.

Livrables.

— Notebook de prise en main comprenant fonctions, classes, démonstrations NumPy et un graphique sin/cos commenté.

— Petit script qui crée un tenseur PyTorch, tente de le déplacer sur GPU et mesure le temps de calcul.

Point de départ suggéré.

```
# TP 1 : Prise en main de l'ecosysteme de la Science des Donnees
#
# OBJECTIFS :
# 1. Se familiariser avec l'environnement de type notebook.
# 2. Revoir quelques concepts fondamentaux de Python.
# 3. Decouvrir NumPy pour le calcul numerique.
# 4. Decouvrir Matplotlib pour la visualisation de donnees.
# 5. Introduction a PyTorch et a la notion de calcul sur CPU vs.
    GPU.
#
# INSTRUCTIONS :
# Ce script est concu pour etre execute dans un environnement
    interactif comme
# Jupyter Notebook ou Google Colab. Chaque section peut etre copiee
```

```
# dans une cellule de code distincte. Les commentaires commencant
   par "##"
# sont des explications qui seraient idealement dans des cellules
   de texte (Markdown).

##
   ------------------------------------------------------------------------

## PARTIE 1 : RAPPELS DE PYTHON
##
   ------------------------------------------------------------------------

## Python est le langage de predilection en science des donnees. Sa
    syntaxe
## est simple et lisible. Un des aspects les plus importants est
   que les blocs
## de code (fonctions, boucles, etc.) sont definis par leur
   INDENTATION.

print("Bonjour le monde !")

# Definition d'une fonction
def carre(x):
    """Cette fonction retourne le carre de son argument."""
    return x ** 2

# Utilisation d'une boucle for pour iterer sur une liste
nombres = [1, 2, 3, 4, 5]
carres = []
for nombre in nombres:
    resultat = carre(nombre)
    carres.append(resultat)
    print(f"Le carre de {nombre} est {resultat}")

print("\nListe des carres :", carres)

# Python est aussi un langage oriente objet. On peut definir nos
   propres types de donnees.
class Point:
    """Une classe simple pour representer un point dans un plan 2D.
       """
```

```
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_origine(self):
        return (self.x**2 + self.y**2)**0.5

p = Point(3, 4)
print(f"\nLe point ({p.x}, {p.y}) est a une distance de {p.
   distance_origine()} de l'origine.")


##
   ------------------------------------------------------------------------

## PARTIE 2 : NUMPY - LE CALCUL SCIENTIFIQUE
##
   ------------------------------------------------------------------------

## NumPy est la bibliotheque fondamentale pour le calcul
   scientifique en Python.
## Elle fournit un objet puissant : le ndarray (tableau N-
   dimensionnel).
## Les operations sur les tableaux NumPy sont beaucoup plus rapides
    que sur
## les listes Python natives car elles sont implementees en C.

import numpy as np

# Creation d'un tableau NumPy a partir d'une liste Python
liste_py = [1, 2, 3, 4, 5]
array_np = np.array(liste_py)

print("\n--- NumPy ---")
print("Liste Python :", liste_py)
print("Tableau NumPy :", array_np)

# La grande difference : les operations vectorielles
liste_py_2 = liste_py + liste_py
array_np_2 = array_np + array_np
```

```
print("\nAddition d'une liste a elle-meme (concatenation) :",
    liste_py_2)
print("Addition d'un tableau a lui-meme (terme a terme) :",
    array_np_2)

# Creation de tableaux
zeros = np.zeros((2, 3)) # Un tableau 2x3 rempli de zeros
uns = np.ones((3, 2))   # Un tableau 3x2 rempli de uns
aleatoire = np.random.rand(2, 2) # Un tableau 2x2 avec des nombres
    aleatoires entre 0 et 1

print("\nTableau de zeros :\n", zeros)
print("Tableau de uns :\n", uns)
print("Tableau aleatoire :\n", aleatoire)

# Multiplication de matrices
mat_A = np.array([[1, 2], [3, 4]])
mat_B = np.array([[5, 6], [7, 8]])
produit_matriciel = mat_A @ mat_B # L'operateur @ est utilise pour
    le produit matriciel

print("\nProduit matriciel de A et B :\n", produit_matriciel)
print("Forme du tableau (shape) :", produit_matriciel.shape)


##
    ------------------------------------------------------------------------

## PARTIE 3 : MATPLOTLIB - LA VISUALISATION DE DONNEES
##
    ------------------------------------------------------------------------

## "Un bon croquis vaut mieux qu'un long discours". Matplotlib est
    la
## bibliotheque la plus utilisee pour creer des graphiques et des
    visualisations.

import matplotlib.pyplot as plt

# Creons quelques donnees avec NumPy
x = np.linspace(0, 2 * np.pi, 100) # 100 points entre 0 et 2*pi
```

```
y_sin = np.sin(x)
y_cos = np.cos(x)

# Creation du graphique
plt.figure(figsize=(10, 6)) # Definit la taille de la figure
plt.plot(x, y_sin, label='sin(x)', color='blue', linestyle='-')
plt.plot(x, y_cos, label='cos(x)', color='red', linestyle='--')

# Ajout de titres et legendes
plt.title("Fonctions Sinus et Cosinus")
plt.xlabel("Angle [rad]")
plt.ylabel("Valeur")
plt.grid(True)
plt.legend()

# Affichage du graphique
plt.show()


##
   ------------------------------------------------------------------------

## PARTIE 4 : PYTORCH - INTRODUCTION A L'APPRENTISSAGE PROFOND
##
   ------------------------------------------------------------------------

## PyTorch est l'un des frameworks d'apprentissage profond les plus
    populaires.
## Son objet de base est le Tenseur, qui est tres similaire au
   ndarray de NumPy,
## mais avec deux super-pouvoirs :
## 1. Il peut calculer automatiquement les gradients (pour la
   retropropagation).
## 2. Il peut effectuer des calculs sur des accelerateurs materiels
    (GPU, TPU).

import torch
import time

print("\n--- PyTorch ---")
print("Version de PyTorch :", torch.__version__)
```

```
# Creation d'un tenseur
tenseur = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
print("\nUn tenseur PyTorch :\n", tenseur)

# La question cruciale : ou sont mes donnees ? Sur quel "device" ?
print("Device du tenseur :", tenseur.device)

# Verifions si un GPU est disponible
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("\nUn GPU est disponible ! Nous allons l'utiliser.")
else:
    device = torch.device("cpu")
    print("\nAucun GPU detecte. Utilisation du CPU.")

# Deplacons notre tenseur sur le device selectionne
tenseur_gpu = tenseur.to(device)
print("Device du nouveau tenseur :", tenseur_gpu.device)

# Tenter une operation entre un tenseur CPU et un tenseur GPU
   provoquera une erreur.
# C'est une erreur tres courante en pratique !
try:
    resultat_erreur = tenseur + tenseur_gpu
except RuntimeError as e:
    print("\nErreur attendue :", e)
    print("On ne peut pas operer sur des tenseurs qui ne sont pas
       sur le meme device.")

##
   ------------------------------------------------------------------------

## PARTIE 5 : EXERCICE DE SYNTHESE - LE GAIN DE PERFORMANCE DU GPU
##
   ------------------------------------------------------------------------

## Nous allons maintenant illustrer concretement l'interet du GPU
   pour les
## calculs massivement paralleles, comme la multiplication de
   grandes matrices.
```

```
## Nous allons chronometrer cette operation sur CPU puis sur GPU.

taille_matrice = 2000
iterations = 100

# --- Calcul sur CPU ---
print(f"\nDebut du calcul sur CPU (matrice {taille_matrice}x{
    taille_matrice}, {iterations} iterations)...")
mat_A_cpu = torch.randn(taille_matrice, taille_matrice, device='cpu
    ')
mat_B_cpu = torch.randn(taille_matrice, taille_matrice, device='cpu
    ')

start_time_cpu = time.time()
for _ in range(iterations):
    resultat_cpu = mat_A_cpu @ mat_B_cpu
end_time_cpu = time.time()

temps_cpu = end_time_cpu - start_time_cpu
print(f"Temps de calcul sur CPU : {temps_cpu:.4f} secondes.")

# --- Calcul sur GPU (si disponible) ---
if torch.cuda.is_available():
    print(f"\nDebut du calcul sur GPU (matrice {taille_matrice}x{
        taille_matrice}, {iterations} iterations)...")
    mat_A_gpu = mat_A_cpu.to(device)
    mat_B_gpu = mat_B_cpu.to(device)

    # Synchronisation pour une mesure de temps precise sur GPU
    # Les operations GPU sont asynchrones, il faut attendre qu'
        elles soient finies.
    torch.cuda.synchronize()

    start_time_gpu = time.time()
    for _ in range(iterations):
        resultat_gpu = mat_A_gpu @ mat_B_gpu
    torch.cuda.synchronize()
    end_time_gpu = time.time()

    temps_gpu = end_time_gpu - start_time_gpu
    print(f"Temps de calcul sur GPU : {temps_gpu:.4f} secondes.")
```

```
    print(f"\nLe calcul sur GPU etait environ {temps_cpu /
        temps_gpu:.2f} fois plus rapide !")

##
   ------------------------------------------------------------------------

## CONCLUSION DU TP
##
   ------------------------------------------------------------------------

## Dans ce TP, nous avons :
## - Revu les bases de Python.
## - Manipule des tableaux avec NumPy.
## - Cree un graphique simple avec Matplotlib.
## - Decouvert les tenseurs PyTorch et l'importance du "device" (
   CPU/GPU).
## - Demontre l'acceleration massive que peut apporter un GPU sur
   des taches
##   parallelisables.
##
## Vous etes maintenant prets a construire votre premier reseau de
   neurones !
```

0.3 TP 1.1 — Perceptron et séparation linéaire

Objectifs. Implémenter l'algorithme du perceptron et visualiser la frontière de décision sur un jeu de données synthétique.

Livrables.

- Script `tp1_perceptron.py` (ou notebook équivalent) incluant génération des données, boucle d'entraînement, suivi des erreurs.
- Figure présentant les points et la frontière finale, accompagnée d'un court commentaire sur la convergence.

Point de départ suggéré.

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

rng = np.random.default_rng(0)
n = 100
```

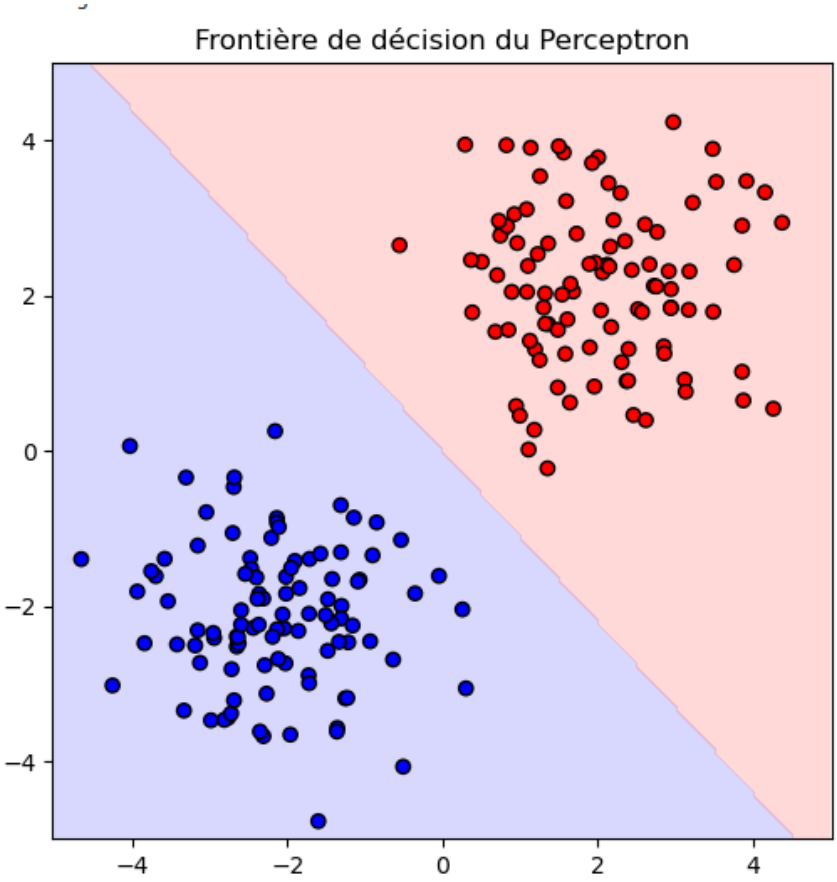

FIGURE 3 – Frontière de décision du perceptron (script d'exemple : `code/tp_1_1_perceptron.py`).

```
X_pos = rng.normal(loc=2.0, scale=1.0, size=(n, 2))
X_neg = rng.normal(loc=-2.0, scale=1.0, size=(n, 2))
X = np.vstack([X_pos, X_neg])
y = np.concatenate([np.ones(n), -np.ones(n)])
perm = rng.permutation(len(X))
X, y = X[perm], y[perm]

w = np.zeros(2)
b = 0.0
eta = 0.1
history = []
for epoch in range(15):
    errors = 0
    for xi, yi in zip(X, y):
        score = np.dot(w, xi) + b
        if yi * score <= 0:
            w += eta * yi * xi
            b += eta * yi
```

```
            errors += 1
    history.append(errors)
    if errors == 0:
        break

# À FAIRE : construire un maillage via np.meshgrid, calculer la pré
   diction du perceptron sur cette grille, tracer contourf/contour,
    superposer les points d'entraînement et enregistrer la figure
   dans un fichier image (par exemple 'perceptron_frontiere.png').
```

Ajouter la visualisation (fonction `np.meshgrid`, `plt.contourf`) et interpréter l'évolution de `history`.

Expérimenter en écartant plus ou moins les deux distributions de points et en jouant sur leur variance.

0.4 TP 1.2 — MLP et compromis biais-variance

Objectifs. Concevoir un perceptron multi-couches pour l'ensemble `moons`, comparer sous-apprentissage, sur-apprentissage et régularisation par dropout.

Livrables.

- Module Python `mlp_moons.py` définissant une classe `MLP` paramétrable.
- Notebook contenant trois expériences (simple / profond / profond + dropout) avec courbes de pertes et frontières de décision.

Point de départ suggéré.

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

X, y = make_moons(n_samples=500, noise=0.3, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size
   =0.3, random_state=42)
scaler = StandardScaler().fit(X_train)
X_train = torch.tensor(scaler.transform(X_train), dtype=torch.
   float32)
X_val = torch.tensor(scaler.transform(X_val), dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.long)
y_val = torch.tensor(y_val, dtype=torch.long)
```

```python
class MLP(nn.Module):
    def __init__(self, hidden_layers, dropout_p=0.0):
        super().__init__()
        layers = []
        in_dim = 2
        for hidden_dim in hidden_layers:
            layers.append(nn.Linear(in_dim, hidden_dim))
            layers.append(nn.ReLU())
            if dropout_p > 0:
                layers.append(nn.Dropout(p=dropout_p))
            in_dim = hidden_dim
        layers.append(nn.Linear(in_dim, 2))  # 2 classes
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)

# À FAIRE : implémenter une fonction train_and_evaluate qui entraî
   ne pendant N époques, retourne les pertes/performance train et
   validation, puis coder des fonctions de tracé des courbes et des
    frontières de décision pour chaque expérimentation.
```

Compléter le code d'entraînement (perte `CrossEntropyLoss`, optimiseur Adam, suivi des courbes) et produire les comparatifs demandés.

Exemple de résultat visé.

0.5 TP 1.3 — Rétropropagation manuelle et optimiseurs

Objectifs. Comparer SGD, RMSProp et Adam sur l'ensemble `moons`, puis dériver manuellement les gradients d'un petit MLP pour valider ceux de PyTorch.

Livrables.

— Notebook séparé en deux parties : comparaison des pertes, puis rétropropagation manuelle avec tableau des écarts.

Point de départ suggéré.

```python
class SimpleMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
```

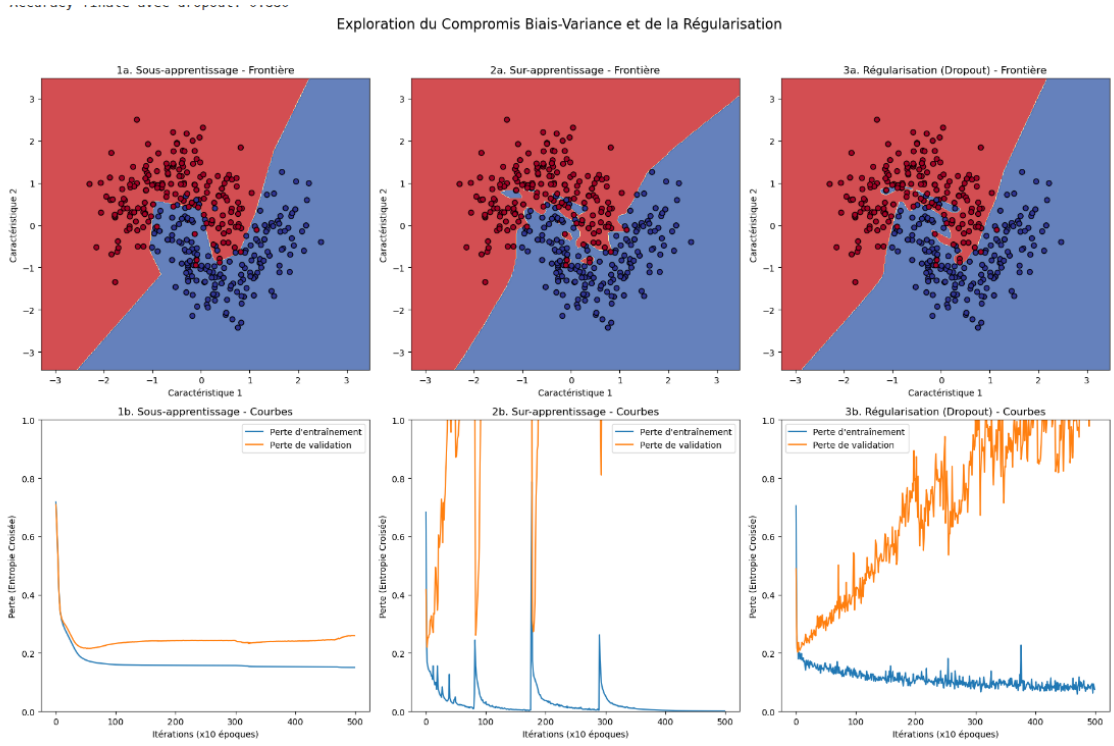

FIGURE 4 – Frontières de décision et/ou courbes de pertes selon l'architecture et le dropout (script d'exemple : `code/tp_1_2_mlp_moons.py`).

```
            nn.Linear(2, 32),
            nn.ReLU(),
            nn.Linear(32, 32),
            nn.ReLU(),
            nn.Linear(32, 2)
        )

    def forward(self, x):
        return self.net(x)

# À FAIRE : coder la fonction train_with_optimizer(optimizer_class,
    **kwargs) qui instancie SimpleMLP, entraîne 500 époques avec l'
   optimiseur fourni, stocke la perte toutes les 5 époques et
   renvoie la liste.
# À FAIRE : produire un graphique unique comparant les pertes
   renvoyées par chaque optimiseur (SGD momentum, RMSProp, Adam)
   avec légende et axes annotés.

# Section rétropropagation : initialiser W1, b1, W2, b2 avec
   requires_grad=True
# puis reproduire la chaîne de calcul manuelle vue en cours.
```

L'étudiant doit implémenter la fonction `train_with_optimizer` (500 époques, enregistrement toutes les 5 époques) et la dérivation manuelle en suivant la règle de la chaîne.

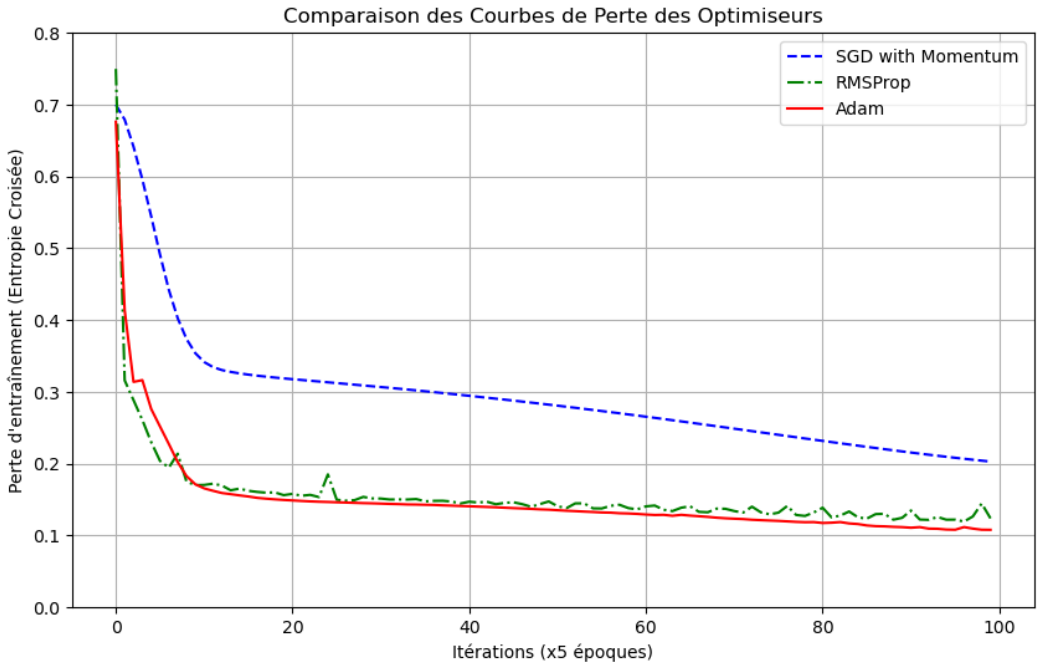

FIGURE 5 – Comparaison de la descente de la perte pour SGD/RMSProp/Adam (script d'exemple : `code/tp_1_3_backprop_manual.py`).

Exemple de résultat visé.

0.6 TP 1.4 — PINNs pour l'oscillateur amorti

Objectifs. Étudier trois scénarios autour d'un oscillateur harmonique amorti : extrapolation, résolution d'équation différentielle et identification de paramètres.

Livrables.

— Trois scripts ou notebooks (extrapolation, résolution, inverse) avec figures comparatives et journal des pertes.

Point de départ suggéré.

```
import torch
import torch.nn as nn

def exact_solution(t, m=1.0, mu=2.0, k=100.0):
    w0 = torch.sqrt(torch.tensor(k / m))
    d = mu / (2 * m)
    w = torch.sqrt(w0**2 - d**2)
    phi = torch.atan(-d / w)
```

```
    A = 1.0 / (2 * torch.cos(phi))
    return torch.exp(-d * t) * 2 * A * torch.cos(phi + w * t)

class FCN(nn.Module):
    def __init__(self, input_dim=1, hidden_dim=32, depth=3,
       output_dim=1):
        super().__init__()
        layers = [nn.Linear(input_dim, hidden_dim), nn.Tanh()]
        for _ in range(depth - 1):
            layers += [nn.Linear(hidden_dim, hidden_dim), nn.Tanh()
               ]
        layers.append(nn.Linear(hidden_dim, output_dim))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)

# À FAIRE : écrire deux fonctions train_regression_pure(...) et
   train_pinn(...) qui gèrent l'entraînement respectif sans
   contrainte physique et avec résidu différentiel (autograd),
   journalisent les pertes et renvoient le modèle entraîné.
# en suivant la décomposition expliquée dans l'énoncé.
```

Pour chaque scénario, compléter les boucles d'entraînement, tracer les résultats et commenter l'apport de la contrainte physique ou de l'estimation de paramètres.

Exemples de résultats visés.

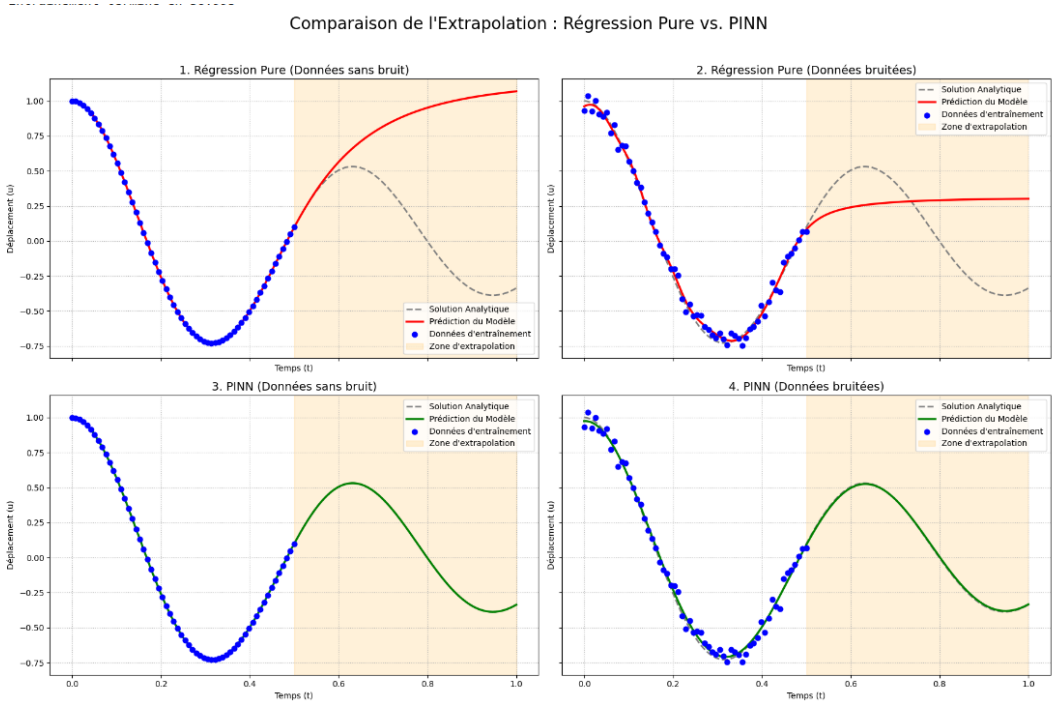

FIGURE 6 – Extrapolation (régression pure) vs vérité analytique (script : `code/tp_1_4_1_pinn_extrapo.py`).

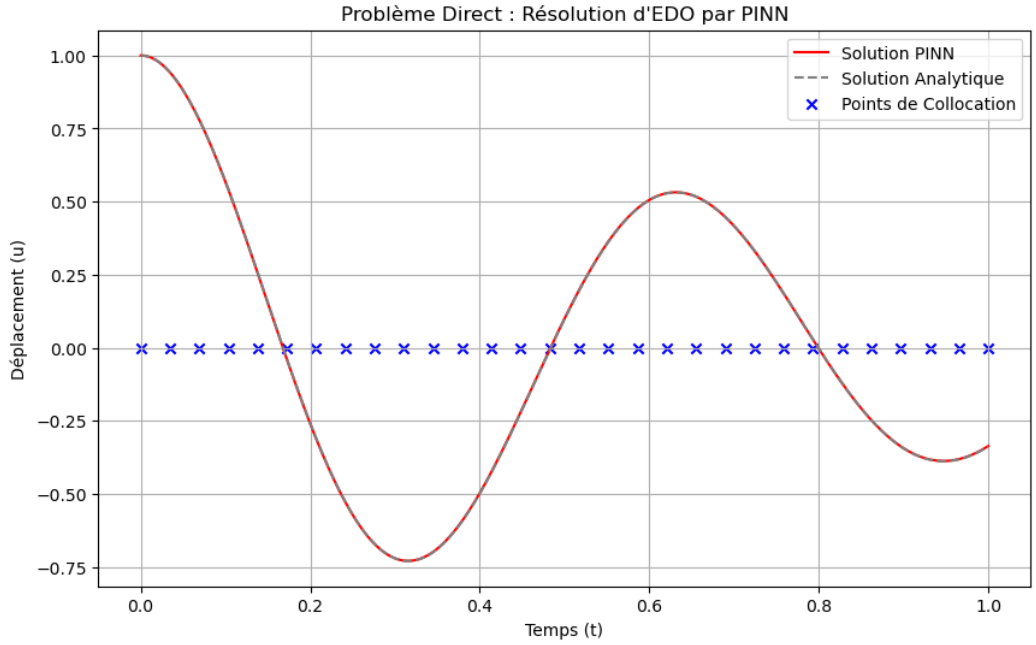

FIGURE 7 – PINN pour l'EDO (contrainte physique via résidu) (script : `code/tp_1_4_2_pinn_edo_simu.py`).

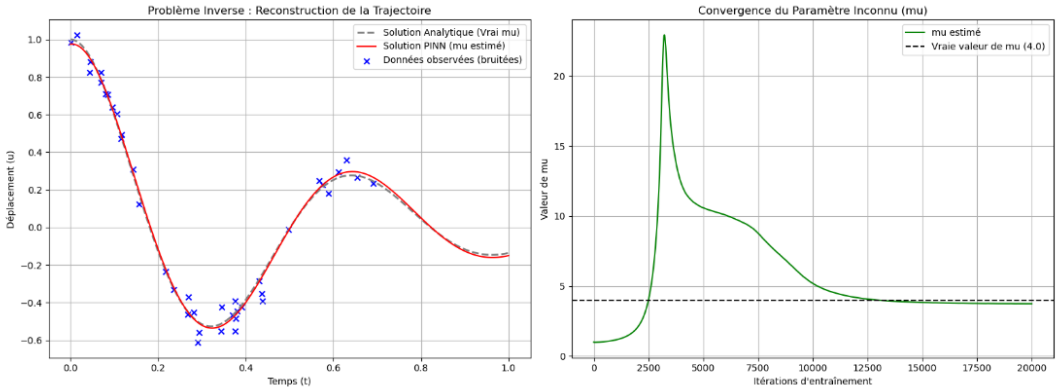

FIGURE 8 – Identification de paramètres (inférence des constantes de l'oscillateur) (script : `code/tp_1_4_3_pinn_edo_inverse.py`).