Travaux Pratiques — Cours d'Apprentissage Profond

Aurélien Plyer & Kevin Helvig

24 septembre 2025

Table des matières

Session 1 — Premiers pas, MLP et PINNs

Cette première série de travaux pratiques doit 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

1. Ouvrir Google Colab et créer un 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 fichiers de `code/` nécessaires (ex : `tp_1_1_perceptron.py`) dans des cellules Colab afin d'exécuter les blocs pas à pas.

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')
```

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 coursDL_cpu.yml -n coursDL

# Option GPU (si CUDA + drivers OK)
conda env create -f 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.

```python
from __future__ import annotations
import time
import numpy as np
import torch

# 1. Fonctions et boucles
print("Bonjour le monde !")

def carre(x: float) -> float:
    """Retourne le carré de x."""
    return x ** 2

nombres = [1, 2, 3, 4, 5]
carres = []
for nombre in nombres:
    carres.append(carre(nombre))
print("Carrés:", carres)

# 2. Classe simple
class Point:
    def __init__(self, x: float, y: float) -> None:
        self.x, self.y = x, y

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

print(Point(3, 4).distance_origine())  # A compléter par des tests supplémentaires

# 3. Placeholders pour les analyses NumPy / Matplotlib / PyTorch
#    FAIRE : compléter ici les blocs NumPy (création de matrices, produit matriciel), Matplotlib (tracé sin/cos avec titre, axes, grille et légende) et PyTorch (mesure de temps CPU vs GPU avec torch.cuda.is_available()).
```

Compléter les blocs NumPy (création de matrices, produit matriciel) et Matplotlib (tracé des fonctions sin et cos). Pour PyTorch, mesurer le temps d'une multiplication matricielle sur CPU puis, si `torch.cuda.is_available()` renvoie `True`, répéter sur GPU et commenter la différence.

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.

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

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

#    FAIRE : construire un maillage via np.meshgrid, calculer la pr
   édiction 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`.

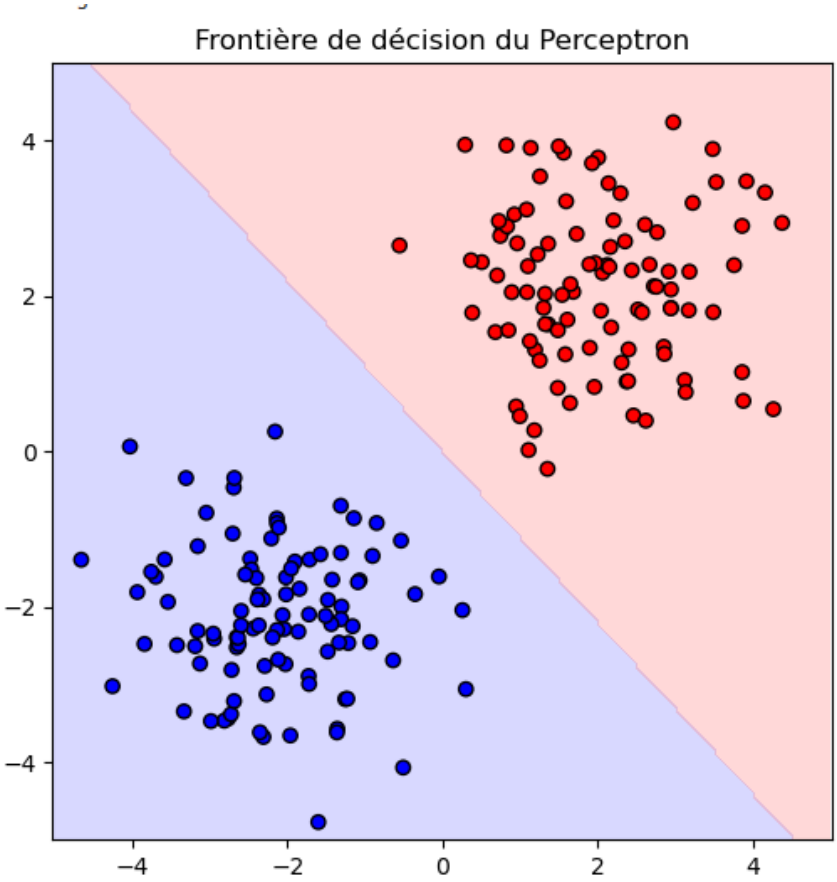

FIGURE 1 – Frontière de décision du perceptron (script d'exemple : `code/tp_1_1_perceptron.py`).

Exemple de résultat visé.

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.

```python
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)

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)

#    FAIRE : implémenter une fonction train_and_evaluate qui entraîne 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.

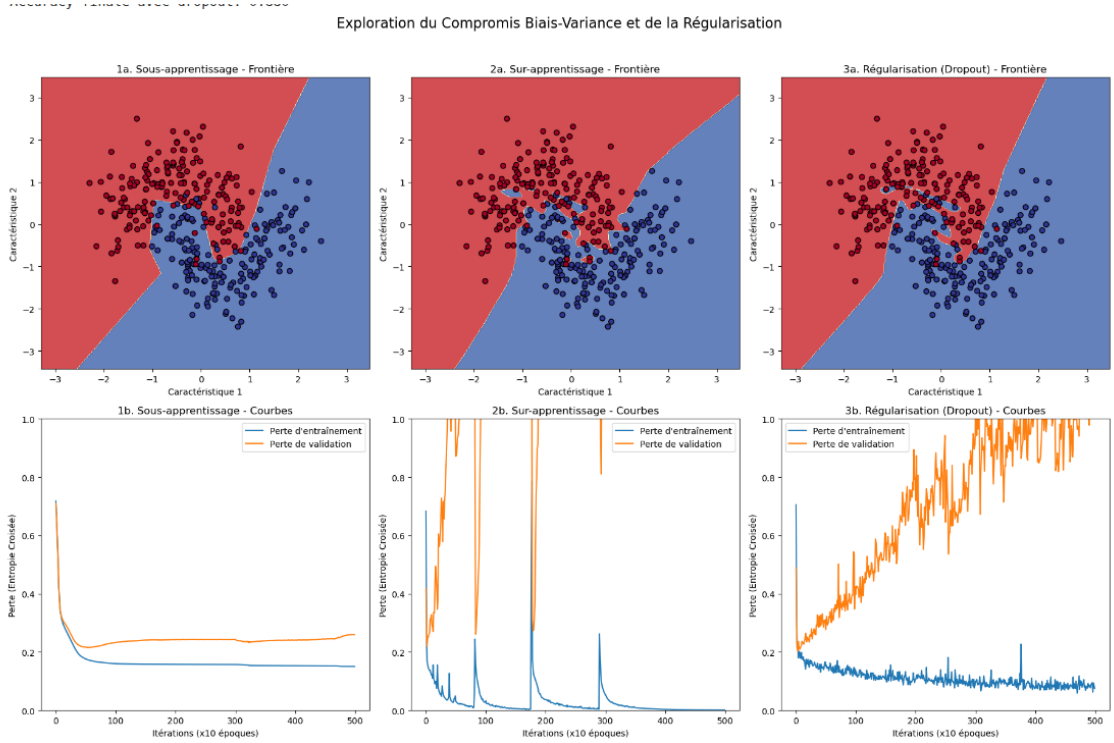

FIGURE 2 – Frontières de décision et/ou courbes de pertes selon l'architecture et le dropout (script d'exemple : `code/tp_1_2_mlp_moons.py`).

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

```
11
12      def forward(self, x):
13          return self.net(x)
14
15  #     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.
16  #     FAIRE : produire un graphique unique comparant les pertes
        renvoyées par chaque optimiseur (SGD momentum, RMSProp, Adam)
        avec légende et axes annotés.
17
18  # Section rétropropagation : initialiser W1, b1, W2, b2 avec
        requires_grad=True
19  # 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.

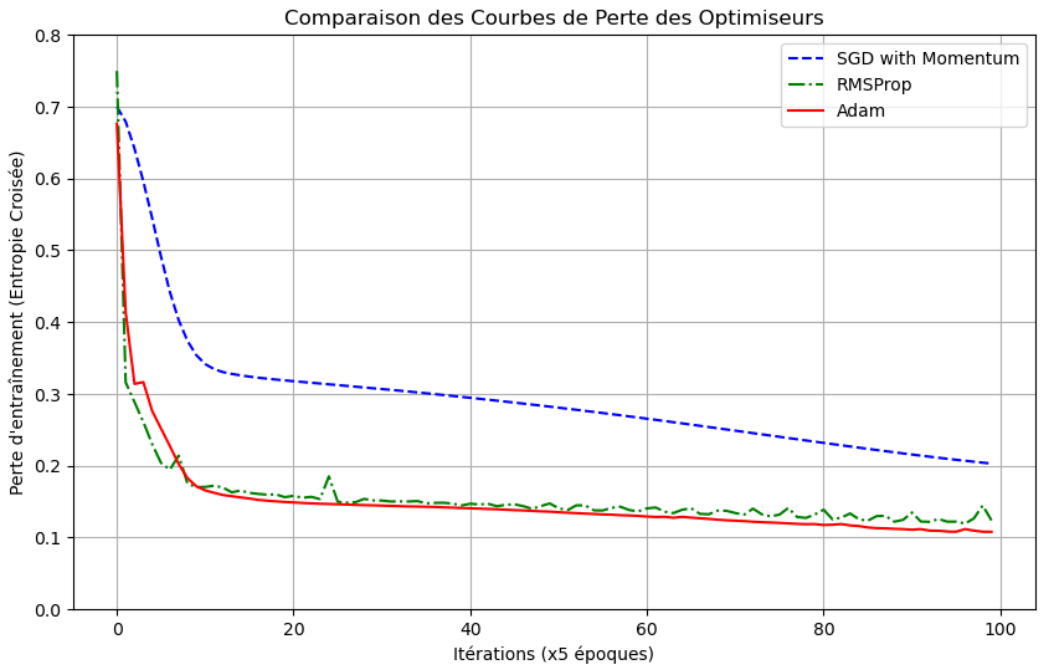

FIGURE 3 – Comparaison de la descente de la perte pour SGD/RMSProp/Adam (script d'exemple : `code/tp_1_3_backprop_manual.py`).

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.

```python
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)

#    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.

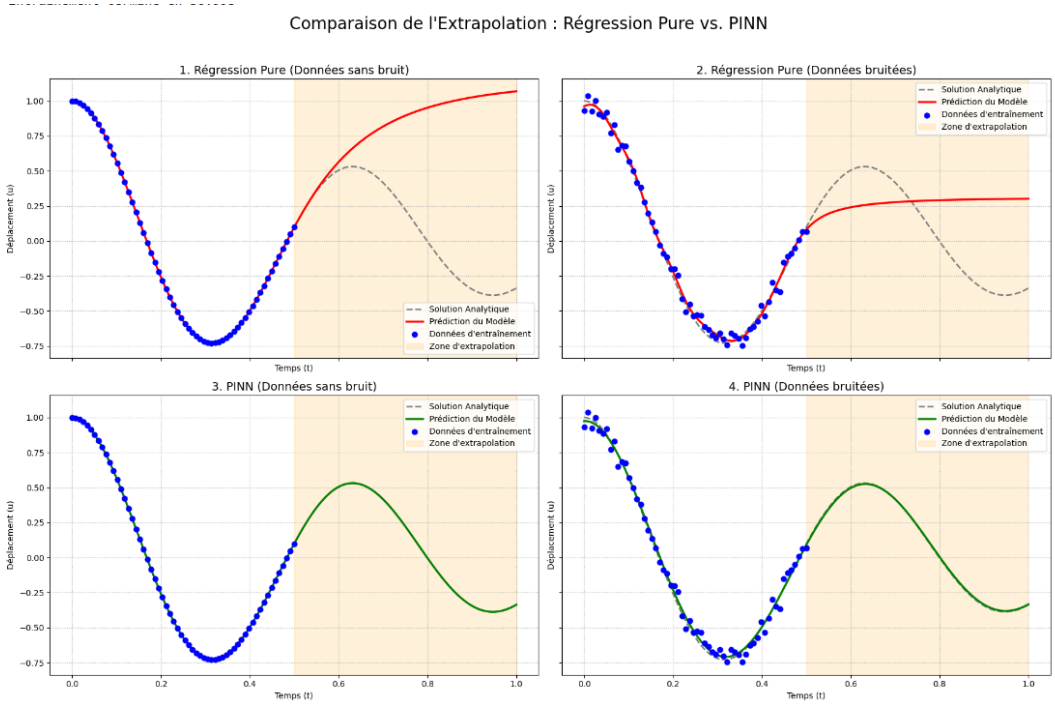

FIGURE 4 – Extrapolation (régression pure) vs vérité analytique (script : `code/tp_1_4_1_pinn_extrapo.py`).

Exemples de résultats visés.

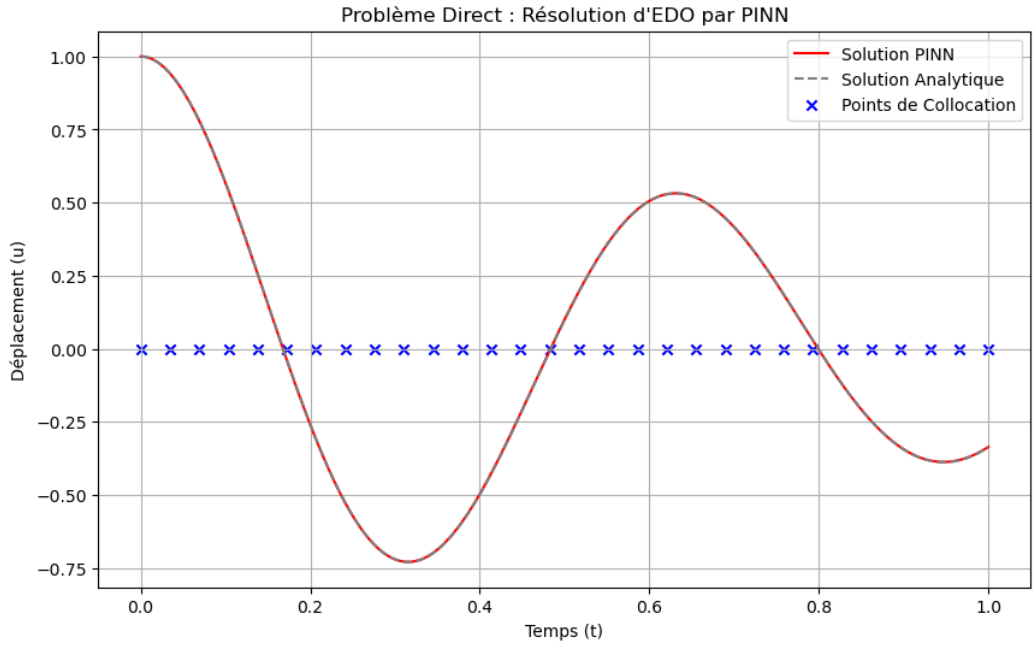

FIGURE 5 – PINN pour l'EDO (contrainte physique via résidu) (script : `code/tp_1_4_2_pinn_edo_simu.py`).

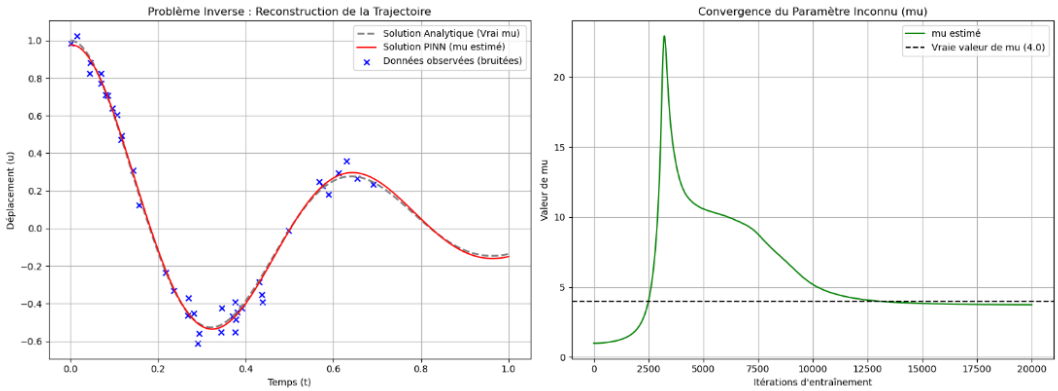

FIGURE 6 – Identification de paramètres (inférence des constantes de l'oscillateur) (script : `code/tp_1_4_3_pinn_edo_inverse.py`).