

Travaux Pratiques — Cours d’Apprentissage Profond

Aurélien Plyer & Kevin Helvig

25 septembre 2025

Table des matières

0.1	Préparation de l’environnement de travail	2
0.2	TP 1.0 — Reprises de Python et NumPy	5
0.3	TP 1.1 — Perceptron et séparation linéaire	12
0.4	TP 1.2 — MLP et compromis biais-variance	14
0.5	TP 1.3 — Rétropropagation manuelle et optimiseurs	15
0.6	TP 1.4 — PINNs pour l’oscillateur amorti	17

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.

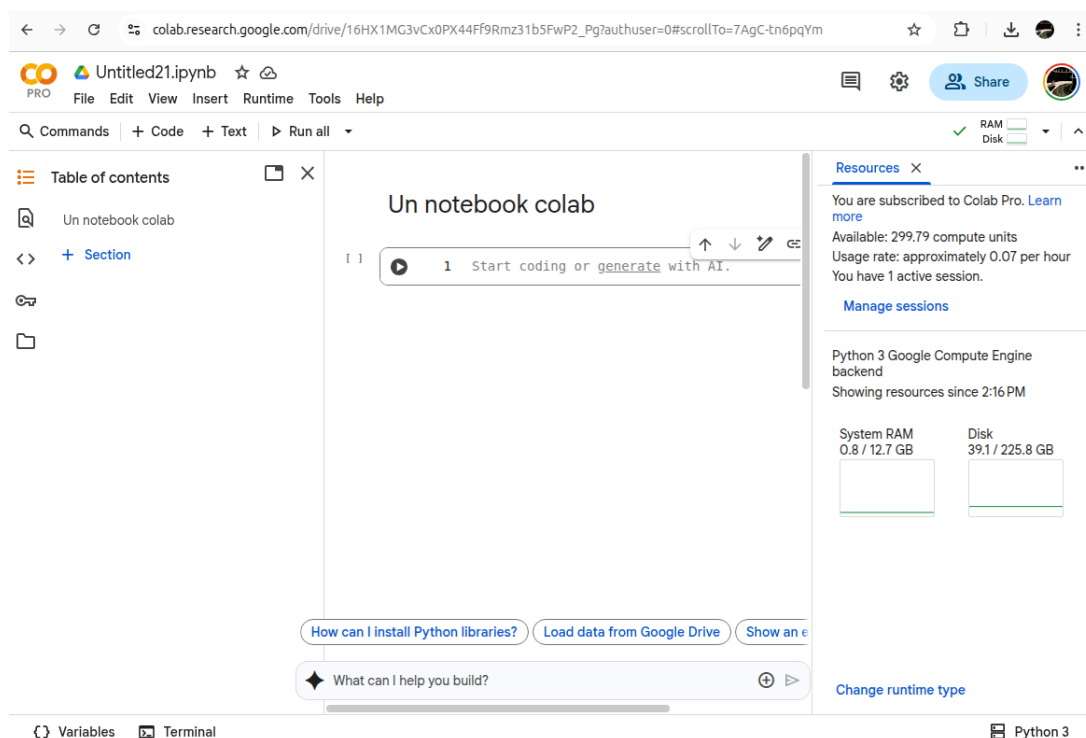


FIGURE 1 – Interface de création de notebook sur Google Colab.

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 :

```
1 import torch, sys
2 print("Python:", sys.version)
3 print("Torch:", torch.__version__)
4 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 :

```
1 from google.colab import drive
2 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 à l'esprit que le stockage Drive est pas forcément co-localisé avec la machine virtuel Colab que vous utilisez pour éviter de faire traverser l'océan à 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).

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

Pour libérer proprement le montage, exécuter :

```
1 from google.colab import drive
2 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`) :

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

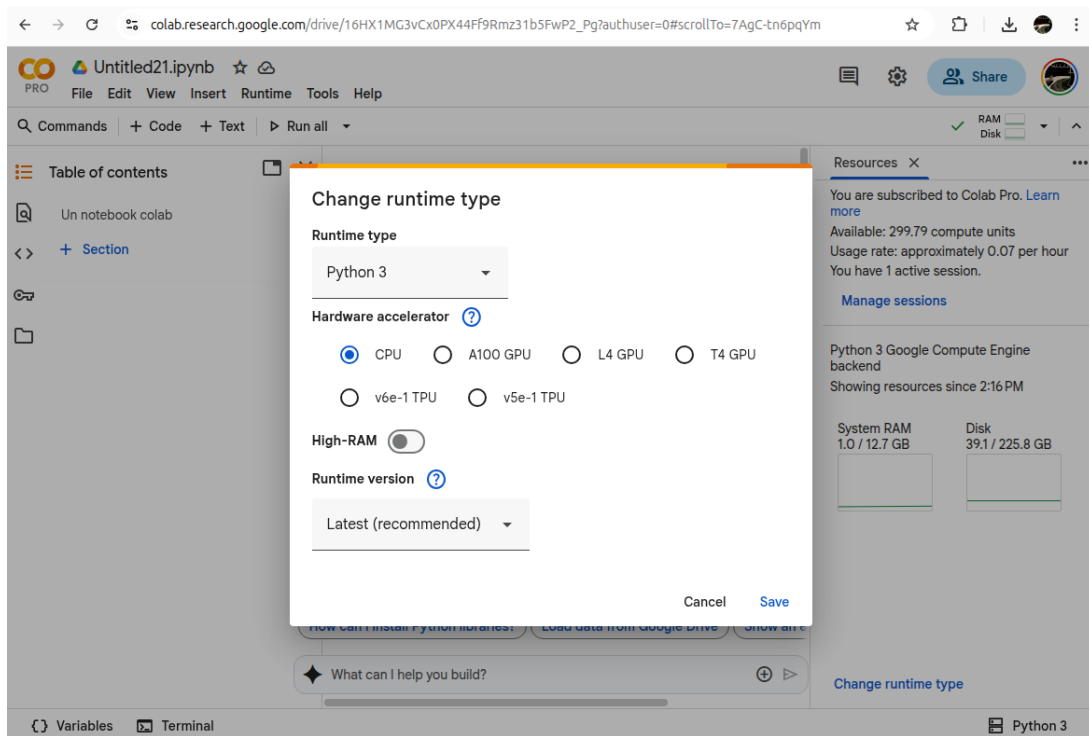


FIGURE 2 – Exécution d'un bloc PyTorch sur Colab avec GPU activé.

Sous Windows, exécuter l'installateur 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 :

```
1 # Ouvrir un terminal puis :
2 cd /chemin/vers/le/depot
3
4 # Option CPU (recommandée par défaut)
5 conda env create -f code/coursDL_cpu.yml -n coursDL
6
7 # Option GPU (si CUDA + drivers OK)
8 conda env create -f code/coursDL_gpu.yml -n coursDL
```

Activation et enregistrement du noyau Jupyter :

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

3) Lancer Jupyter et choisir le noyau

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

```
2 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)

```
1 import torch
2 print("CUDA disponible:", torch.cuda.is_available())
3 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

```
1 # Exemple : lancer le perceptron de la session 1
2 conda activate coursDL
3 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.

```
1 # TP 1 : Prise en main de l'écosystème de la Science des Données
2 #
3 # OBJECTIFS :
4 # 1. Se familiariser avec l'environnement de type notebook.
5 # 2. Revoir quelques concepts fondamentaux de Python.
6 # 3. Découvrir NumPy pour le calcul numérique.
7 # 4. Découvrir Matplotlib pour la visualisation de données.
8 # 5. Introduction à PyTorch et à la notion de calcul sur CPU vs.
    GPU.
9 #
10 # INSTRUCTIONS :
11 # Ce script est conçu pour être exécuté dans un environnement
    interactif comme
12 # Jupyter Notebook ou Google Colab. Chaque section peut être copiée
```

```

13 # dans une cellule de code distincte. Les commentaires commençant
    par "###"
14 # sont des explications qui seraient idéalement dans des cellules
    de texte (Markdown).
15
16 ##
    -----
17 ## PARTIE 1 : RAPPELS DE PYTHON
18 ##
    -----
19 ## Python est le langage de prédilection en science des données. Sa
    syntaxe
20 ## est simple et lisible. Un des aspects les plus importants est
    que les blocs
21 ## de code (fonctions, boucles, etc.) sont définis par leur
    INDENTATION.
22
23 print("Bonjour le monde !")
24
25 # Définition d'une fonction
26 def carre(x):
27     """Cette fonction retourne le carré de son argument."""
28     return x ** 2
29
30 # Utilisation d'une boucle for pour itérer sur une liste
31 nombres = [1, 2, 3, 4, 5]
32 carres = []
33 for nombre in nombres:
34     resultat = carre(nombre)
35     carres.append(resultat)
36     print(f"Le carré de {nombre} est {resultat}")
37
38 print("\nListe des carrés :", carres)
39
40 # Python est aussi un langage orienté objet. On peut définir nos
    propres types de données.
41 class Point:
42     """Une classe simple pour représenter un point dans un plan 2D.
        """

```

```

43     def __init__(self, x, y):
44         self.x = x
45         self.y = y
46
47     def distance_origine(self):
48         return (self.x**2 + self.y**2)**0.5
49
50 p = Point(3, 4)
51 print(f"\nLe point ({p.x}, {p.y}) est a une distance de {p.
52     distance_origine()} de l'origine.")
53
54 ##
55
56 -----
57
58 ## PARTIE 2 : NUMPY - LE CALCUL SCIENTIFIQUE
59 ##
60 -----
61
62 ## NumPy est la bibliotheque fondamentale pour le calcul
63     scientifique en Python.
64 ## Elle fournit un objet puissant : le ndarray (tableau N-
65     dimensionnel).
66 ## Les operations sur les tableaux NumPy sont beaucoup plus rapides
67     que sur
68 ## les listes Python natives car elles sont implementees en C.
69
70 import numpy as np
71
72 # Creation d'un tableau NumPy a partir d'une liste Python
73 liste_py = [1, 2, 3, 4, 5]
74 array_np = np.array(liste_py)
75
76 print("\n--- NumPy ---")
77 print("Liste Python :", liste_py)
78 print("Tableau NumPy :", array_np)
79
80 # La grande difference : les operations vectorielles
81 liste_py_2 = liste_py + liste_py
82 array_np_2 = array_np + array_np
83

```

```

76 print("\nAddition d'une liste a elle-meme (concatenation) :",
    liste_py_2)
77 print("Addition d'un tableau a lui-meme (terme a terme) :",
    array_np_2)
78
79 # Creation de tableaux
80 zeros = np.zeros((2, 3)) # Un tableau 2x3 rempli de zeros
81 uns = np.ones((3, 2)) # Un tableau 3x2 rempli de uns
82 aleatoire = np.random.rand(2, 2) # Un tableau 2x2 avec des nombres
    aleatoires entre 0 et 1
83
84 print("\nTableau de zeros :\n", zeros)
85 print("Tableau de uns :\n", uns)
86 print("Tableau aleatoire :\n", aleatoire)
87
88 # Multiplication de matrices
89 mat_A = np.array([[1, 2], [3, 4]])
90 mat_B = np.array([[5, 6], [7, 8]])
91 produit_matriciel = mat_A @ mat_B # L'operateur @ est utilise pour
    le produit matriciel
92
93 print("\nProduit matriciel de A et B :\n", produit_matriciel)
94 print("Forme du tableau (shape) :", produit_matriciel.shape)
95
96
97 ##
    -----
98 ## PARTIE 3 : MATPLOTLIB - LA VISUALISATION DE DONNEES
99 ##
    -----
100 ## "Un bon croquis vaut mieux qu'un long discours". Matplotlib est
    la
101 ## bibliotheque la plus utilisee pour creer des graphiques et des
    visualisations.
102
103 import matplotlib.pyplot as plt
104
105 # Creons quelques donnees avec NumPy
106 x = np.linspace(0, 2 * np.pi, 100) # 100 points entre 0 et 2*pi

```



```

107 y_sin = np.sin(x)
108 y_cos = np.cos(x)
109
110 # Creation du graphique
111 plt.figure(figsize=(10, 6)) # Definit la taille de la figure
112 plt.plot(x, y_sin, label='sin(x)', color='blue', linestyle='--')
113 plt.plot(x, y_cos, label='cos(x)', color='red', linestyle='--')
114
115 # Ajout de titres et legendes
116 plt.title("Fonctions Sinus et Cosinus")
117 plt.xlabel("Angle [rad]")
118 plt.ylabel("Valeur")
119 plt.grid(True)
120 plt.legend()
121
122 # Affichage du graphique
123 plt.show()
124
125
126 ##
    -----

127 ## PARTIE 4 : PYTORCH - INTRODUCTION A L'APPRENTISSAGE PROFOND
128 ##
    -----

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

134
135 import torch
136 import time
137
138 print("\n--- PyTorch ---")
139 print("Version de PyTorch :", torch.__version__)

```

```

140
141 # Creation d'un tenseur
142 tenseur = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
143 print("\nUn tenseur PyTorch :\n", tenseur)
144
145 # La question cruciale : ou sont mes donnees ? Sur quel "device" ?
146 print("Device du tenseur :", tenseur.device)
147
148 # Verifions si un GPU est disponible
149 if torch.cuda.is_available():
150     device = torch.device("cuda")
151     print("\nUn GPU est disponible ! Nous allons l'utiliser.")
152 else:
153     device = torch.device("cpu")
154     print("\nAucun GPU detecte. Utilisation du CPU.")
155
156 # Deplacons notre tenseur sur le device selectionne
157 tenseur_gpu = tenseur.to(device)
158 print("Device du nouveau tenseur :", tenseur_gpu.device)
159
160 # Tenter une operation entre un tenseur CPU et un tenseur GPU
    provoquera une erreur.
161 # C'est une erreur tres courante en pratique !
162 try:
163     resultat_erreur = tenseur + tenseur_gpu
164 except RuntimeError as e:
165     print("\nErreur attendue :", e)
166     print("On ne peut pas operer sur des tenseurs qui ne sont pas
        sur le meme device.")
167
168 ##
    -----
169 ## PARTIE 5 : EXERCICE DE SYNTHESE - LE GAIN DE PERFORMANCE DU GPU
170 ##
    -----
171 ## Nous allons maintenant illustrer concretement l'interet du GPU
    pour les
172 ## calculs massivement paralleles, comme la multiplication de
    grandes matrices.

```

```

173 ## Nous allons chronometrer cette operation sur CPU puis sur GPU.
174
175 taille_matrice = 2000
176 iterations = 100
177
178 # --- Calcul sur CPU ---
179 print(f"\nDebut du calcul sur CPU (matrice {taille_matrice}x{
    taille_matrice}, {iterations} iterations)...")
180 mat_A_cpu = torch.randn(taille_matrice, taille_matrice, device='cpu
    ')
181 mat_B_cpu = torch.randn(taille_matrice, taille_matrice, device='cpu
    ')
182
183 start_time_cpu = time.time()
184 for _ in range(iterations):
185     resultat_cpu = mat_A_cpu @ mat_B_cpu
186 end_time_cpu = time.time()
187
188 temps_cpu = end_time_cpu - start_time_cpu
189 print(f"Temps de calcul sur CPU : {temps_cpu:.4f} secondes.")
190
191 # --- Calcul sur GPU (si disponible) ---
192 if torch.cuda.is_available():
193     print(f"\nDebut du calcul sur GPU (matrice {taille_matrice}x{
        taille_matrice}, {iterations} iterations)...")
194     mat_A_gpu = mat_A_cpu.to(device)
195     mat_B_gpu = mat_B_cpu.to(device)
196
197     # Synchronisation pour une mesure de temps precise sur GPU
198     # Les operations GPU sont asynchrones, il faut attendre qu'
        elles soient finies.
199     torch.cuda.synchronize()
200
201     start_time_gpu = time.time()
202     for _ in range(iterations):
203         resultat_gpu = mat_A_gpu @ mat_B_gpu
204     torch.cuda.synchronize()
205     end_time_gpu = time.time()
206
207     temps_gpu = end_time_gpu - start_time_gpu
208     print(f"Temps de calcul sur GPU : {temps_gpu:.4f} secondes.")

```

```

209
210     print(f"\nLe calcul sur GPU etait environ {temps_cpu /
211           temps_gpu:.2f} fois plus rapide !")
212 ##
213     -----
214 ## CONCLUSION DU TP
215 ##
216     -----
217 ## Dans ce TP, nous avons :
218 ## - Revu les bases de Python.
219 ## - Manipule des tableaux avec NumPy.
220 ## - Cree un graphique simple avec Matplotlib.
221 ## - Decouvert les tenseurs PyTorch et l'importance du "device" (
222     CPU/GPU).
223 ## - Demontre l'acceleration massive que peut apporter un GPU sur
224     des taches
225 ##     parallelisables.
226 ##
227 ## Vous etes maintenant prêts a construire votre premier reseau de
228     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.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 rng = np.random.default_rng(0)
5 n = 100

```

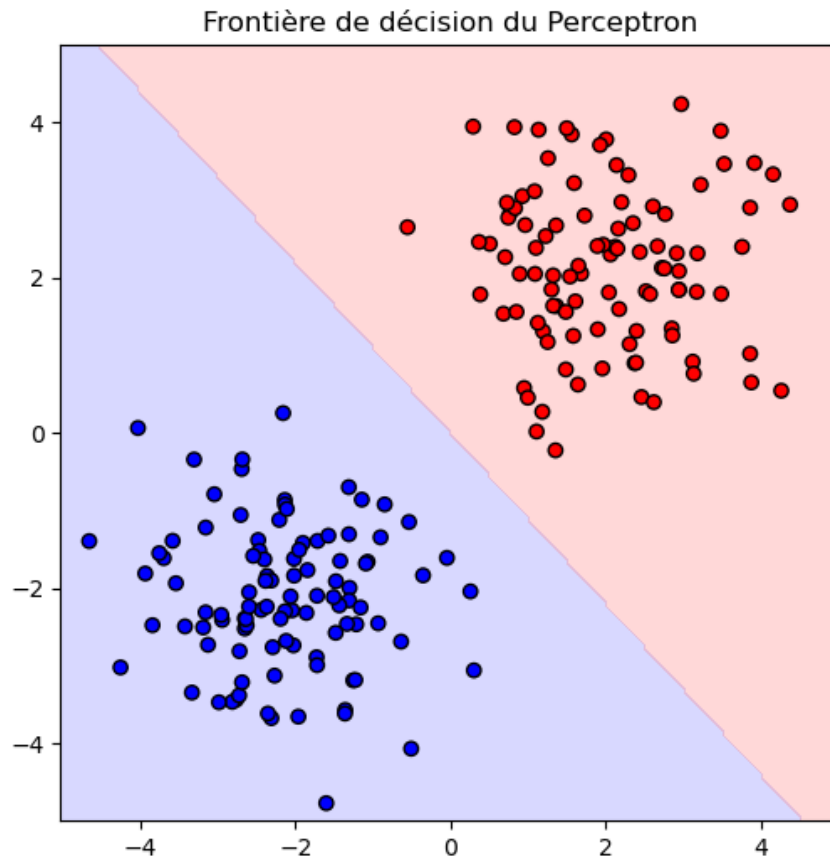


FIGURE 3 – Frontière de décision du perceptron (script d'exemple : code/tp_1_1_perceptron.py).

```

6 X_pos = rng.normal(loc=2.0, scale=1.0, size=(n, 2))
7 X_neg = rng.normal(loc=-2.0, scale=1.0, size=(n, 2))
8 X = np.vstack([X_pos, X_neg])
9 y = np.concatenate([np.ones(n), -np.ones(n)])
10 perm = rng.permutation(len(X))
11 X, y = X[perm], y[perm]
12
13 w = np.zeros(2)
14 b = 0.0
15 eta = 0.1
16 history = []
17 for epoch in range(15):
18     errors = 0
19     for xi, yi in zip(X, y):
20         score = np.dot(w, xi) + b
21         if yi * score <= 0:
22             w += eta * yi * xi
23             b += eta * yi

```

```

24         errors += 1
25     history.append(errors)
26     if errors == 0:
27         break
28
29 # 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.

```

1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 from sklearn.datasets import make_moons
5 from sklearn.model_selection import train_test_split
6 from sklearn.preprocessing import StandardScaler
7
8 X, y = make_moons(n_samples=500, noise=0.3, random_state=42)
9 X_train, X_val, y_train, y_val = train_test_split(X, y, test_size
    =0.3, random_state=42)
10 scaler = StandardScaler().fit(X_train)
11 X_train = torch.tensor(scaler.transform(X_train), dtype=torch.
    float32)
12 X_val = torch.tensor(scaler.transform(X_val), dtype=torch.float32)
13 y_train = torch.tensor(y_train, dtype=torch.long)
14 y_val = torch.tensor(y_val, dtype=torch.long)

```

```

15
16 class MLP(nn.Module):
17     def __init__(self, hidden_layers, dropout_p=0.0):
18         super().__init__()
19         layers = []
20         in_dim = 2
21         for hidden_dim in hidden_layers:
22             layers.append(nn.Linear(in_dim, hidden_dim))
23             layers.append(nn.ReLU())
24             if dropout_p > 0:
25                 layers.append(nn.Dropout(p=dropout_p))
26             in_dim = hidden_dim
27         layers.append(nn.Linear(in_dim, 2)) # 2 classes
28         self.net = nn.Sequential(*layers)
29
30     def forward(self, x):
31         return self.net(x)
32
33 # 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.

```

1 class SimpleMLP(nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.net = nn.Sequential(

```

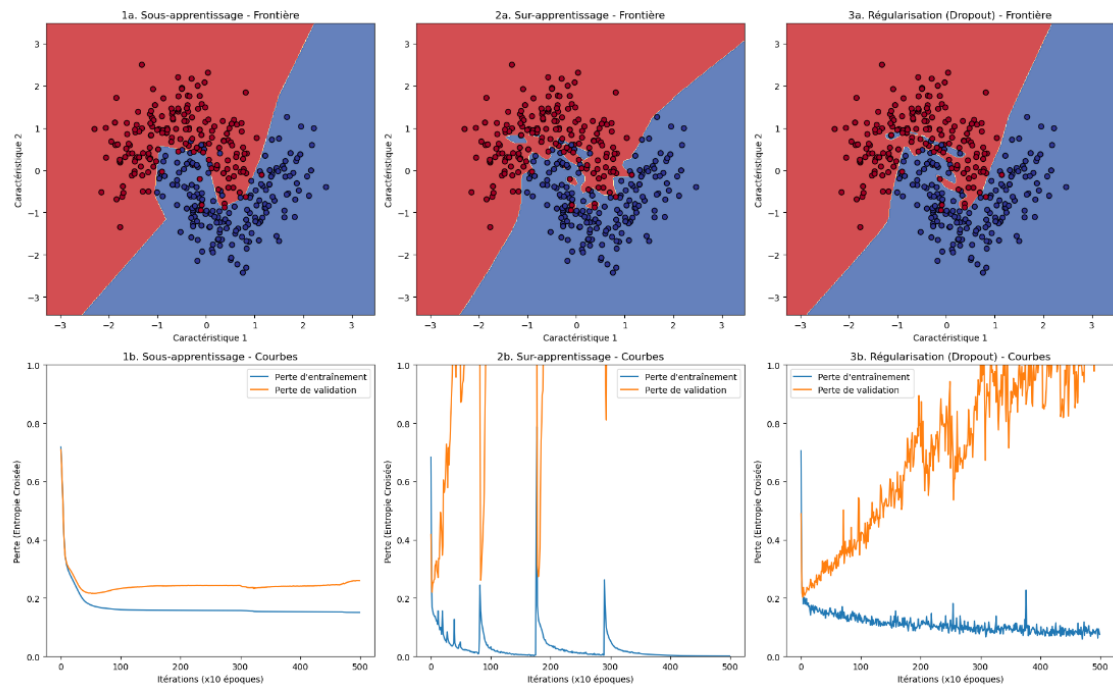


FIGURE 4 – Frontières de décision et/ou courbes de pertes selon l'architecture et le dropout (script d'exemple : code/tp_1_2_mlp_moons.py).

```

5         nn.Linear(2, 32),
6         nn.ReLU(),
7         nn.Linear(32, 32),
8         nn.ReLU(),
9         nn.Linear(32, 2)
10    )
11
12    def forward(self, x):
13        return self.net(x)
14
15    # À FAIRE : coder la fonction train_with_optimizer(optimizer_class,
16               **kwargs) qui instancie SimpleMLP, entraîne 500 époques avec l'
17               optimiseur fourni, stocke la perte toutes les 5 époques et
18               renvoie la liste.
19
20    # À FAIRE : produire un graphique unique comparant les pertes
21               renvoyées par chaque optimiseur (SGD momentum, RMSProp, Adam)
22               avec légende et axes annotés.
23
24    # Section rétropropagation : initialiser W1, b1, W2, b2 avec
25               requires_grad=True
26
27    # 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.

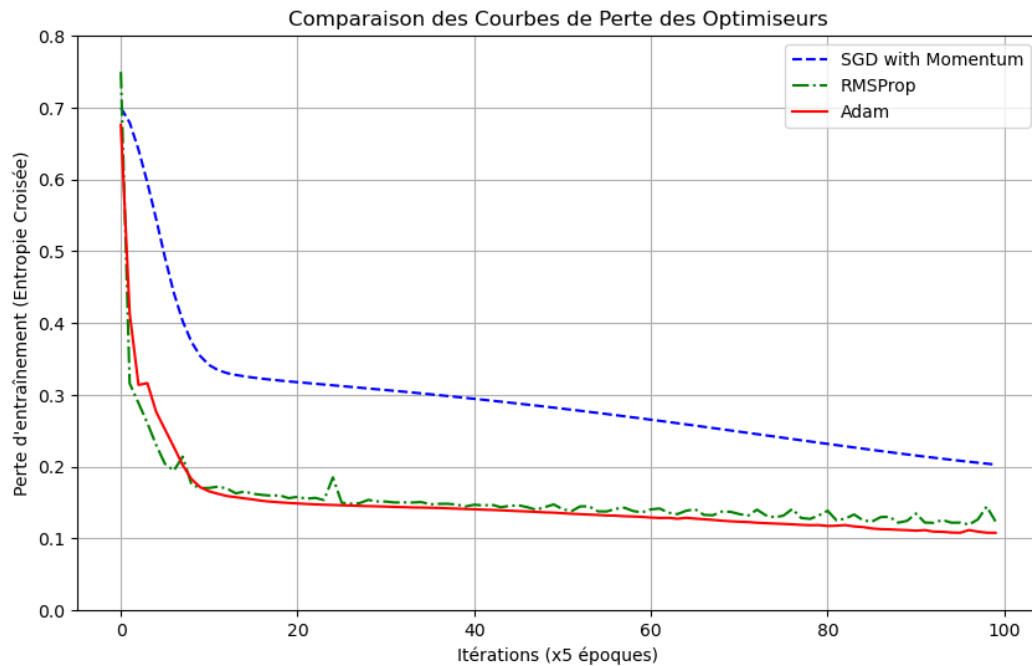


FIGURE 5 – Comparaison de la descente de la perte pour SGD/RMSProp/Adam (script d'exemple : `code/tp_1_3_backprop_manual.py`).

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.

```
1 import torch
2 import torch.nn as nn
3
4 def exact_solution(t, m=1.0, mu=2.0, k=100.0):
5     w0 = torch.sqrt(torch.tensor(k / m))
6     d = mu / (2 * m)
7     w = torch.sqrt(w0**2 - d**2)
8     phi = torch.atan(-d / w)
```

```

9      A = 1.0 / (2 * torch.cos(phi))
10     return torch.exp(-d * t) * 2 * A * torch.cos(phi + w * t)
11
12 class FCN(nn.Module):
13     def __init__(self, input_dim=1, hidden_dim=32, depth=3,
14                 output_dim=1):
15         super().__init__()
16         layers = [nn.Linear(input_dim, hidden_dim), nn.Tanh()]
17         for _ in range(depth - 1):
18             layers += [nn.Linear(hidden_dim, hidden_dim), nn.Tanh()]
19         layers.append(nn.Linear(hidden_dim, output_dim))
20         self.net = nn.Sequential(*layers)
21
22     def forward(self, x):
23         return self.net(x)
24
25 # À FAIRE : écrire deux fonctions train_regression_pure(...) et
26 train_pinn(...) qui gèrent l'entraînement respectif sans
27 contrainte physique et avec résidu différentiel (autograd),
28 journalisent les pertes et renvoient le modèle entraîné.
29 # 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.

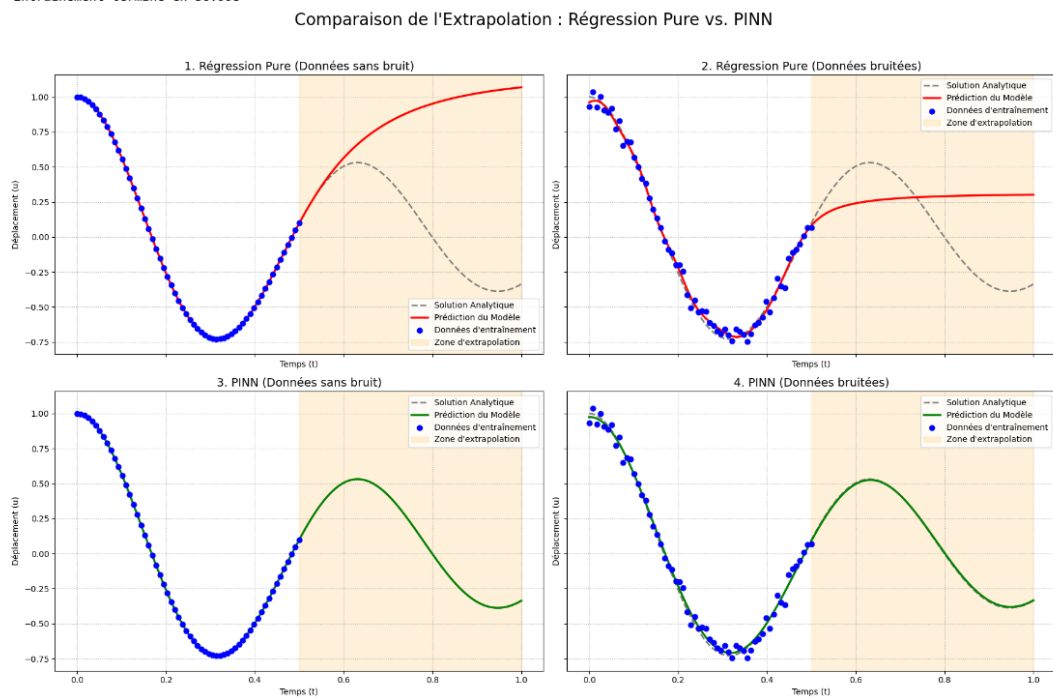


FIGURE 6 – Extrapolation (régression pure) vs vérité analytique (script : `code/tp_1_4_1_pinn_extrapo.py`).

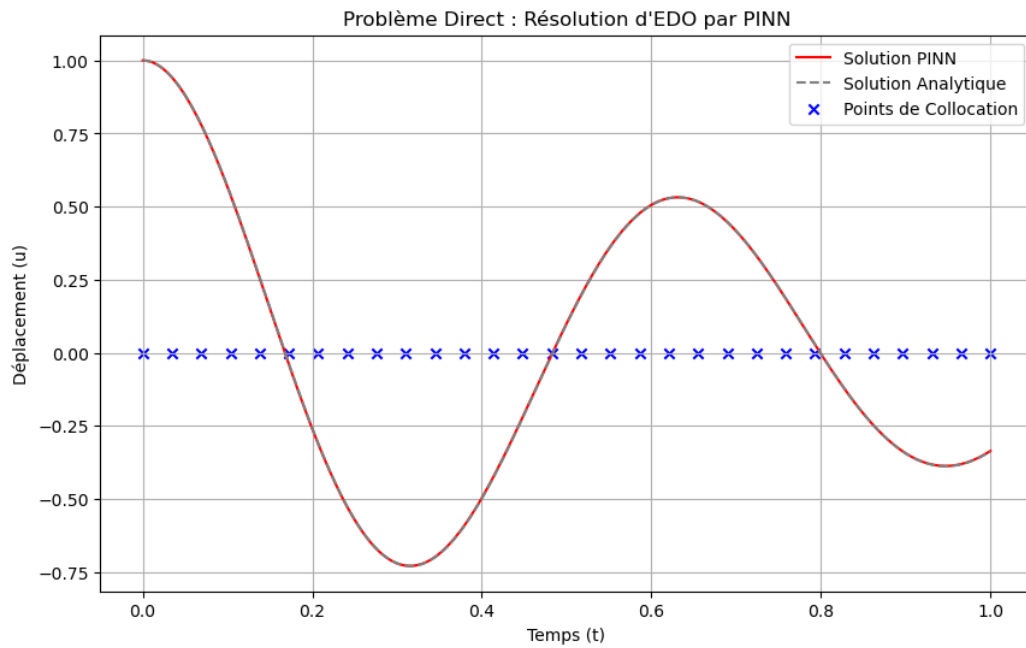


FIGURE 7 – PINN pour l'EDO (contrainte physique via résidu) (script : `code/tp_1_4_2_pinn_edo_simu.py`).

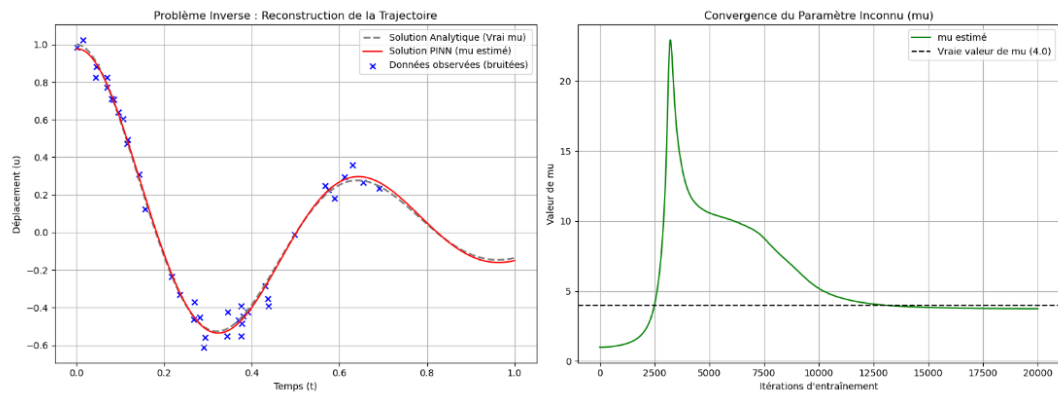


FIGURE 8 – Identification de paramètres (inférence des constantes de l'oscillateur) (script : `code/tp_1_4_3_pinn_edo_inverse.py`).