Rapport

TP4 : Implémentation d'un DAN

Réalisé par

Félix Foucher de Brandois

ENSEEIHT - Formation ModIA, 5e année *2024-2025*

1 Introduction

Dans ce TP, nous avons implémenté un Data Assimilation Network (DAN), un modèle d'apprentissage automatique conçu pour assimiler des données dans des systèmes dynamiques observés. Le DAN combine des techniques d'assimilation de données bayésiennes avec des réseaux de neurones pour approximer les densités de probabilité conditionnelles p_t^a et p_t^b, qui représentent respectivement l'état du système conditionné aux observations jusqu'à l'instant t et l'état prédit à partir des observations jusqu'à l'instant $t-1$.

Le DAN est structuré autour de trois modules principaux :

— **Analyseur (a)** : Met à jour la mémoire cachée h_t^a en fonction de l'observation y_t.

— **Propagateur (b)** : Prédit la mémoire cachée h_{t+1}^b à partir h_t^a.

— **Procédure (c)** : Convertit les mémoires cachées en distributions Gaussiennes q_t^a et q_t^b.

2 Description du modèle physique

On modélise un système linéaire 2D qui simule un mouvement circulaire, défini par les équations suivantes :

$$\textbf{Propagation :}\ x_t = Mx_{t-1} + \eta_t \tag{1}$$

$$\textbf{Observation :}\ y_t = Hx_t + \epsilon_t \tag{2}$$

où x_t représente l'état du système à l'instant t, M est la matrice de rotation définie par :

$$M = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \qquad \text{avec } \theta = \frac{\pi}{100}$$

et $\eta_t \sim \mathcal{N}(0, \sigma_p I)$ est le bruit de propagation, y_t est l'observation à l'instant t, H est la matrice d'observation ($H = I$), et $\epsilon_t \sim \mathcal{N}(0, \sigma_o I)$ est le bruit d'observation.

3 Pre-training of c for Linear 2d

On commence par l'entraînement de (c) pour approximer la distribution initiale $p(x_0)$.

TODO 1.1 : Architecture du DAN

Le fichier `manage_exp.py` contient la fonction experiment qui gère l'exécution globale du code. Cette fonction initialise le réseau DAN et les modules de propagation et d'observation. Elle exécute ensuite les étapes de pré-entraînement et d'entraînement complet ou en ligne selon le mode spécifié.

Le fichier `filters.py` contient l'implémentation des différents modules du DAN (a, b, c) ainsi que les classes pour la génération des données (Lin2d, EDO).

TODO 1.2 : Implémentation du module Gaussien dans ConstructorC

Le module `Gaussian` dans `ConstructorC` convertit un vecteur en une distribution Gaussienne paramétrée par une moyenne μ et une matrice de covariance $\Sigma = \Lambda\Lambda^T$, où Λ est une matrice triangulaire inférieure définie par :

$$\Lambda = \begin{pmatrix} e^{v_n} & 0 & \cdots & 0 \\ v_{2n} & e^{v_{n+1}} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ v_{n+\frac{n(n+1)}{2}-1} & \cdots & v_{3n-2} & e^{v_{2n-1}} \end{pmatrix}$$

avec v le vecteur de sortie du module `FullyConnected`.

Pour assurer la stabilité numérique, les termes diagonaux de Λ sont limités à l'intervalle $[-8, 8]$ via la fonction `torch.clamp` :

```
# Dans la classe Gaussian, methode __init__
minexp = torch.Tensor([-8.0])
maxexp = torch.Tensor([8.0])
diaga = torch.clamp(diaga, min=minexp, max=maxexp)
```

La log-vraisemblance d'une observation x sous cette distribution Gaussienne est donnée par :

$$\log p(x) = -\frac{1}{2}(z^T \cdot z) - \sum_{i=1}^{n}\left(\frac{\log(2\pi)}{2} + \log(\Lambda_{ii})\right) \tag{3}$$

avec $z = \Lambda^{-1}(x - \mu)$.

TODO 1.3 : Génération des données initiales x_0

Les données initiales x_0 sont générées à partir d'une distribution Gaussienne centrée en 3 avec un écart-type σ_0 :

```
# Dans manage_exp.py, fonction get_x0
x0 = 3 * torch.ones(b_size, x_dim) + sigma0 * torch.randn(b_size,
    x_dim)
```

TODO 1.4 : Implémentation de la fonction `closure0`

La fonction `closure0` calcule la perte $L_0(q_0^a)$ définie par :

$$L_0(q_0^a) = \int \left(\frac{(x_0 - \mu_0^a)^T(\Lambda_0^a\Lambda_0^{aT})^{-1}(x_0 - \mu_0^a)}{2}\right) p(x_0)dx_0 + \log|2\pi\Lambda_0^a\Lambda_0^{aT}|^{1/2} \tag{4}$$

Par estimation de Monte-Carlo on obtient alors :

$$L_0(q_0^a) = \frac{1}{I}\sum_{i\leq I}\left(\frac{(x_0^i - \mu_0^a)^T(\Lambda_0^a\Lambda_0^{aT})^{-1}(x_0^i - \mu_0^a)}{2}\right) + \log|2\pi\Lambda_0^a\Lambda_0^{aT}|^{1/2} \tag{5}$$

La fonction `closure0` met également à jour les gradients via la rétropropagation :

```
# Dans manage_exp.py, fonction pre_train_full
def closure0():
    optimizer0.zero_grad()
    pdf_a0 = net.c(ha0)
    logpdf_a0 = -pdf_a0.log_prob(x0).mean()
    logpdf_a0.backward()
    return logpdf_a0
```

On obtient les résultats suivants :

```
## INIT a0 mean tensor([3.0002, 2.9997], grad_fn=<SliceBackward0>)
## INIT a0 var tensor([0.0001, 0.0001], grad_fn=<SliceBackward0>)
## INIT a0 covar tensor([[1.1664e-04, 1.1830e-05],
[1.1830e-05, 1.1293e-04]], grad_fn=<SliceBackward0>)
```

On obtient comme valeur finale de la fonction de perte :

```
LOSS= -7.151272021266152
```

TODO 1.5 : Optimisation du calcul de la matrice de covariance

Nous avons optimisé le calcul de la matrice de covariance $\Sigma = \Lambda\Lambda^T$ en utilisant la fonction `torch.bmm` qui permet de faire du calcul matriciel par batch.
Voici les résultats obtenus après optimisation :

```
Fast computation time: 0.005449056625366211
Slow computation time: 0.17865300178527832
```

Soit un gain de temps de ~32x.

4 Full-training of a, b, c for Linear 2d

Sur la base du module pré-entraîné c, nous allons entraîner conjointement les modules a, b, c.

TODO 2.1 : Correction de l'initialisation dans `FcZero`

Le module `FcZero` implémente un réseau résiduel où chaque couche est de la forme :

$$h \mapsto h + \alpha_\ell \rho(W_\ell h + b_\ell)$$

Les paramètres α_ℓ (initialisés à 0) n'étaient pas correctement déclarés comme des paramètres du module, ce qui empêchait leur mise à jour lors de l'entraînement.

```
# Dans filters.py, classe FcZero
self.alphas = nn.Parameter(torch.zeros(deep), requires_grad=True)
```

TODO 2.2 : Étapes forward du DAN

Les étapes forward du DAN impliquent :

1. Propagation de l'état caché h^a_{t-1} à l'état caché h^b_t via le module de propagation (b).
2. Conversion de l'état caché h^b_t en une distribution Gaussienne q^b_t via le module (c).
3. Mise à jour de l'état caché h^a_t via le module d'analyse (a) en utilisant l'état caché h^b_t et l'observation y_t.
4. Conversion de l'état caché h^a_t en une distribution Gaussienne q^a_t via le module (c).

La fonction `forward` du module DAN effectue ces étapes en appelant les modules (a), (b) et (c) dans l'ordre approprié.

```
# Dans filters.py, methode forward de DAN
prior_mem = self.b(ha)
pdf_b = self.c(prior_mem)
posterior_state = self.a(torch.cat((prior_mem, y), dim=1))
pdf_a = self.c(posterior_state)
```

TODO 2.3 : Génération des séquences d'entraînement

Pour entraîner le DAN, nous générons des séquences d'état et d'observation $\{x_t, y_t\}_{t \leq T}$ via :

— **Propagation** : $x_{t+1} \sim \mathcal{N}(Mx_t, \sigma_p I)$

— **Observation** : $y_t \sim \mathcal{N}(x_{t+1}, \sigma_o I)$

```
# Dans manage_exp.py, fonction train_full
x_prev = x0
xt, yt = [], []
for t in range(T + 1):
    xt.append(x_prev)
    x_next = prop(x_prev).sample().squeeze(0)
    yt.append(obs(x_next).sample().squeeze(0))
    x_prev = x_next
xt = torch.stack(xt, dim=0)  # Shape (T+1, mb, x_dim)
yt = torch.stack(yt, dim=0)
```

TODO 2.4 : Calcul de la perte totale

La fonction de coût complète combine les erreurs de prédiction à chaque étape, ainsi que l'erreur initiale ajustée par L_0 :

$$\mathcal{L}_{\text{complète}} = \sum_{t \leq T} \Big(L_t(q^b_t) + L_t(q^a_t) \Big) + L_0(q^a_0) \tag{6}$$

où $L_t(q_t^b)$ et $L_t(q_t^a)$ sont les pertes de la distribution prédite et de la distribution observée respectivement.

```
# Dans manage_exp.py, fonction train_full
loss_total = 0
for t in range(1, T+1):
    loss, hat = net(hat, xt[t], yt[t])
    loss_total += loss
loss_total = loss_total / T + logpdf_a0
```

5 Résultats sur l'entraînement complet

5.1 Impact de la profondeur du réseau

Nous avons testé différentes valeurs du paramètre `deep` (nombre de couches résiduelles) dans le module `FcZero` pour évaluer l'impact de la profondeur du réseau sur la performance du DAN.

Profondeur	Itération	RMSE_b	RMSE_a	LOGPDF_b	LOGPDF_a	LOSS
1	1	4.19	4.20	556734.16	587653.87	1144388.03
	1000	0.047	0.030	-3.18	-3.98	-7.15
5	1	3.97	3.98	395083.47	316689.98	711773.45
	1000	0.029	0.028	-3.95	-4.03	-7.98
10	1	4.43	4.43	149930.94	113462.45	263393.40
	1000	0.024	0.024	-4.05	-4.35	-8.10

TABLE 1 – Impact de la profondeur du réseau sur la performance du DAN

On observe que l'augmentation de la profondeur du réseau améliore la performance du DAN, mesurée par la RMSE et la log-vraisemblance.

5.2 Affichage de la prédiction à $t = T$ sur les données d'entrainement

On affiche la prédiction du DAN à l'instant $t = T$ sur les données d'entraînement.

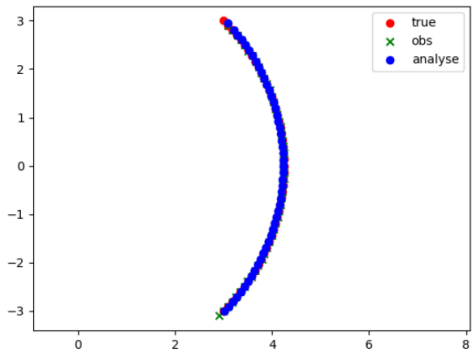

FIGURE 1 – Prédiction du DAN à l'instant $t = T$ sur les données d'entraînement

On observe sur la figure 1 que la trajectoire prédite par le DAN est très proche de la trajectoire réelle : le réseau a bien assimilé les données et a appris à prédire l'état du système.

6 RMSE

La figure 2 montre l'évolution de la RMSE pour les modules (a) et (b) au cours de l'entraînement et de la phase de test.

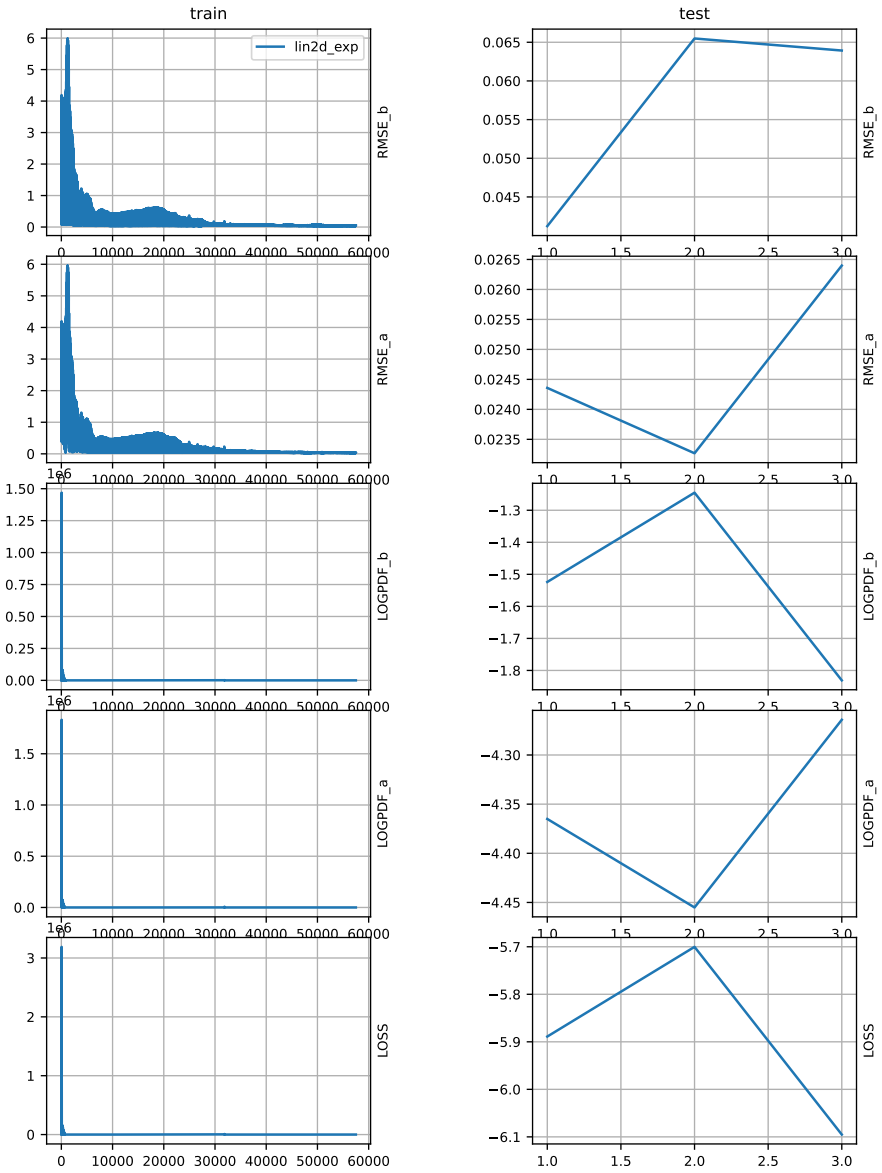

FIGURE 2 – Évolution de la RMSE pour les modules (a) et (b) au cours de l'entraînement

On observe que la RMSE diminue au fil des itérations pour les deux modules durant l'entraînement, indiquant une amélioration de la performance du DAN.