TP Implementation of DAN

Sara Rool

March 2025

1 Introduction

L'objectif de ce TP est d'implémenter un Data Assimilation Network (DAN) pour modéliser un système linéaire 2D, représentant un mouvement circulaire. Ce travail s'inscrit dans la continuité du TP précédent, où nous avons étudié les phases d'assimilation et de propagation dans un système d'assimilation de données 2D.

2 Description du modèle

Dans ce TP, le modèle physique repose sur l'équation de propagation suivante :

$$x_t = Mx_{t-1} + \eta_t$$

où x_t représente l'état du système à l'instant t. $\eta_t \sim \mathcal{N}(0, \sigma_p I)$ correspond au bruit du processus, avec une variance fixée $\sigma_p = 0.01$. Et avec M une matrice de rotation 2×2 définie comme :

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

Les observations du système y_t sont liées aux états par l'équation suivante:

$$y_t = Hx_t + \epsilon_t$$

où $H = I$ est la matrice d'observation, et $\epsilon_t \sim \mathcal{N}(0, \sigma_o I)$ représente le bruit d'observation donc l'erreur de mesure, avec $\sigma_o = 10 \times \sigma_p = 0.1$.

3 Pré-entraînement de c pour `Lin2d`

3.1 Question 1.1 : comprendre les codes

3.1.1 manage_exp.py

Le code `manage_exp.py` contient l'entraînement et l'évaluation de notre DAN. Ses principales fonctionnalités sont:

- Initialisation des états : le code génère les états initiaux x_0 pour l'ensemble d'apprentissage et de test avec les fonctions `get_x0` et `get_x0_test`. Il génère également l'état caché initial h_0^a avec la fonction `get_ha0`.
- Pré-entraînement du module **c** : La fonction `pre_train_full` entraîne uniquement le module **c**, qui modélise la distribution initiale des états x_0. L'optimisation se fait en minimisant la fonction de perte $L_0(q_0^a)$, qui évalue la divergence entre la distribution de **c** et les données observées.
- Entraînement complet des modules **a**, **b** et **c** : Il a deux modes d'entraînement possibles, full et online. Le mode full, qui s'exécute avec la fonction `train_full`, entraîne l'ensemble du réseau **a**, **b**, **c** sur toute la séquence temporelle en utilisant la rétropropagation à travers le temps. La fonction appelle `pre_train_full` qui effectue le pré-entraînement pour stabiliser le modèle. Le mode online, exécutable avec la fonction `train_online`, entraîne le réseau de manière progressive, en mettant à jour ses paramètres au fur et à mesure de l'arrivée des données.
- Évaluation et tests : La fonction `test` permet de tester le réseau entraîné sur un nouvel ensemble de données. Elle génère des trajectoires d'états et d'observations, puis applique le réseau pour prédire les états cachés à partir des observations.
- Gestion : La fonction `experiment` orchestre l'ensemble du processus (initialisation, pré-entraînement, entraînement et test du modèle).

3.1.2 filters.py

Ce fichier contient la classe DAN ainsi que plusieurs modules pour la propagation d'état, l'analyse des observations et la modélisation des distributions de probabilité.

- classe DAN : C'est la classe principale du réseau. Elle orchestre le flux des données à travers différents modules : l'analyseur d'état **a** qui assimile les nouvelles observations, le propagateur d'état **b** qui fait évoluer l'état du système et le convertisseur en distribution probabiliste **c** qui génère des distributions gaussiennes représentant l'incertitude. Cette classe contient 2 méthodes : `forward` qui effectue la forward pass et `clear_scores` qui réinitialise les scores d'évaluation.

- Modules de transformation d'état (`Lin2d` et `EDO`) : Ces modules sont utilisés pour modéliser l'évolution temporelle des états latents. `Lin2d` implémente la dynamique de rotation 2D décrit précédemment. Et `EDO` modélise une équation différentielle ordinaire avec intégration RK4.

- Réseaux de neurones : Différentes architectures sont utilisées pour apprendre les transformations d'état et les observations. `FullyConnected` est un réseau de neurones dense classique. `FcZero` est une version améliorée avec le trick ReZero. `FcZeroLin` est une version de `FcZero` se terminant par une couche linéaire.

- Modélisation probabiliste : Les distributions probabilistes permettent de capturer l'incertitude du modèle. `GaussianDiag` est une distribution gaussienne avec covariance diagonale. `Gaussian` est une distribution gaussienne avec covariance complète et optimisée.

- Constructeurs de modules dynamiques : Les constructeurs facilitent l'initialisation des différentes parties du modèle. `ConstructorA`, `ConstructorB` et `ConstructorC` instancient respectivement les modulesb**a**, **b**, **c** décrits précédemment. `ConstructorProp` et `ConstructorObs` construisent respectivement les modules de propagation et d'observation.

3.2 Question 1.2 : Implémentation du module `Gaussian`

Dans ce module, on transforme le vecteur d'entrée x en une distribution gaussienne de moyenne μ et de matrice de covariance Λ. On calcule ensuite la log probabilité de cette distribution. Le calcul de la log probabilité est le suivant :

$$\log(p(x)) = -\frac{1}{2}(z^T z) - \sum_i \left(\frac{1}{2}\log(2\pi) + \tilde{\Lambda}_i\right)$$

On note z la solution du système linéaire $\Lambda z = x - \mu$. $\tilde{\Lambda}$ se construit en appliquant un seuil min-max $[-8, 8]$ à chaque terme diagonal de la matrice tridiagonale Λ. Ceci est pour protéger contre les instabilités numériques lors du calcul de puissance et de la fonction de perte. L'implémentation se fait avec les lignes de codes suivantes :

```
# calcul de la log probabilite
bs = x.shape[0]
logprob = torch.zeros(bs,1)
mean_diff = x - self.mu
L = self.Lambda
z = torch.linalg.solve_triangular(L, mean_diff.unsqueeze
    (-1), upper=False)
logprob = -0.5 * torch.bmm(z.transpose(1, 2), z).squeeze() -
          self.cov_log_det_

# seuil
```

```
minexp = torch.Tensor([-8.0])
maxexp = torch.Tensor([8.0])
diaga = torch.clamp(diaga, min=minexp, max=maxexp)
```

3.3 Question 1.3 : Implémentation du module `Lin2d`

x_0 est instancié avec la fonction `get_x0` dans le fichier `manage_exp.py`. x_0 est de taille $mb \times n = 128 \times 2$.

$$x_0 = 3 \cdot \mathbf{1}_{b\_size \times x\_dim} + \sigma \cdot \mathcal{N}(0,1)_{b\_size \times x\_dim}$$

L'implémentation de `class Lin2d` se fait comme dans le TP3, en explicitant le modèle physique décrit dans la première partie.

3.4 Question 1.4 : Implémentation de la fonction `closure0`

Cette partie consiste à implémenter la perte L_0. Sa formule orginale est la suivante :

$$L_0(q_0^a) = \int \left((x_0 - \mu_0^a)^\top \left(\Lambda_0^a (\Lambda_0^a)^\top \right)^{-1} (x_0 - \mu_0^a) \right)^2 p(x_0)\, dx_0 + \frac{1}{2} \log \left(2\pi |\Lambda_0^a (\Lambda_0^a)^\top | \right)$$

Or dans notre cas, nous allons utiliser une estimation de Monte-Carlo pour approximer l'intégrale. La formule devient alors, avec I étant la taille du batch:

$$L_0(q_0^a) = \frac{1}{I} \sum_{i=1}^{I} \left((x_0^{(i)} - \mu_0 a)^\top \left(\Lambda_0^a (\Lambda_0^a)^\top \right)^{-1} (x_0^{(i)} - \mu_0^a) \right)^2 + \frac{1}{2} \log \left(2\pi |\Lambda_0^a (\Lambda_0^a)^\top | \right)$$

Cette partie est implémentée comme suivant :

```
optimizer0.zero_grad()
pdf_a0 = net.c(ha0)
logpdf_a0 = -torch.mean(pdf_a0.log_prob(x0))
logpdf_a0.backward()
```

Les sorties dans le terminal sont les suivantes :

```
Pre-train c at t=0
empirical mean of x0 is tensor([3.0002, 3.0006])
## INIT a0 mean tensor([3.0002, 3.0006], grad_fn=SliceBackward0>)
## INIT a0 var tensor([9.7341e-05, 9.6891e-05],
        grad_fn=<SliceBackward0>)
## INIT a0 covar tensor([[ 9.7341e-05, -5.3670e-06],
        [-5.3670e-06,  9.7187e-05]],grad_fn=<SliceBackward0>)
```

Et à la fin du pré-entraînement, L_0 est égal à -6.3408.

3.5 Question 1.5 : Optimisation du code `Gaussian`

Le calcul de `cov`, `cov_inv` et de `cov_log_det` se fait pour l'instant avec le code suivant :

```
self.covariance_matrix = torch.zeros(bs, x_dim, x_dim)
self.cov_inv_ = torch.zeros(bs, x_dim, x_dim)
self.cov_log_det_ = torch.zeros(bs, 1)
logcst = np.log(2*np.pi)/2
for i in range(bs):
    cov = torch.mm(self.Lambda[i,:,:], self.Lambda[i,:,:]
          .T)
    cov_inv = torch.cholesky_inverse(self.Lambda[i,:,:])
    cov_log_det = torch.sum( logcst + diaga[i,:] )
    self.covariance_matrix[i,:,:] = cov
    self.cov_inv_[i,:,:] = cov_inv
    self.cov_log_det_[i,0] = cov_log_det
```

Ce code est effectivement sous optimé car on utilise une boucle `for` qui traite chaque batch individuellement, ce qui entraîne des appels répétitifs aux opérations matricielles, rendant le code lent. De plus, on effectue une inversion de cholesky qui est coûteuse en calcul car elle nécessite une inversion explicite.

Le code optimisé proposé est le suivant :

```
logcst = np.log(2*np.pi)/2
self.covariance_matrix = torch.bmm(self.Lambda, self.Lambda
                         .transpose(1,2))
identity = torch.eye(x_dim, device=vec.device).expand(bs,
           x_dim, x_dim)
self.cov_inv_ = torch.linalg.solve_triangular(self.Lambda,
                identity, upper=False)
self.cov_log_det_ = torch.sum( logcst + diaga, dim=1)
                    .unsqueeze(1)
```

Dans cette version optimisée, toutes les opérations sont vectorisées. La fonction `torch.bmm` effectue la multiplication batchée directement, évitant ainsi la boucle `for`. Et la fonction `torch.linalg.solve_triangular`, est beaucoup plus efficace car la fonction résout un système triangulaire, ce qui est plus stable numériquement et plus rapide que l'inversion explicite.

Pour vérifier l'amélioration, on calcul le temps d'exécution entre les deux algorithmes et on constate une nette différence.

```
Temps code de base :  0.1917741298675537s
Temps code amélioré :  0.0004515647888183594s
```

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

4.1 Question 2.1 : Corriger une erreur dans l'initialisation de `FcZero`

Comme expliqué dans la première partie, `FcZero` est une version améliorée d'un réseau de neurones dense avec le trick ReZero. Le trick ReZero est une technique d'initialisation pour faciliter l'entraînement. Il consiste à ajouter un paramètre α, généralement initialisé à zéro, aux résidus des connexions résiduelles. Le paramètre α est donc un paramètre entrainable et il faut donc le préciser dans son instanciation.

```
self.alphas = nn.Parameter(torch.zeros(deep),
                requires_grad=True)
```

4.2 Question 2.2 : Implémentation de la fonction forward dans le module `DAN`

Dans le cours, nous avons vu que l'implémentation de la fonction forward d'un DAN est la suivante :

Calcul de la perte (et du gradient) sur t
Initialisation: h_0^a
Entrée: h_{t-1}^a, x_t, y_t
Sortie: $\mathcal{L}_t(q_t^b) + \mathcal{L}_t(q_t^a), h_t^a$
Étapes clés:
Calcul de $h_t^b = \mathbf{b}(h_{t-1}^a)$
Calcul de $q_t^b = \mathbf{c}(h_t^b)$
Calcul de $h_t^a = \mathbf{a}(h_t^b, y_t)$
Calcul de $q_t^a = \mathbf{c}(h_t^a)$

Dans le code, cela se traduit par les lignes suivantes :

```
# propagate past mem into prior mem
hb = self.b(ha)
# translate prior mem into prior pdf
pdf_b = self.c(hb)
# analyze prior mem
ha = self.a(torch.cat((hb, y), dim=1))
# translate post mem into post pdf
pdf_a = self.c(ha)
logpdf_a = -torch.mean(pdf_a.log_prob(x))
logpdf_b = -torch.mean(pdf_b.log_prob(x))
```

4.3 Question 2.3 : Générer un échantillon d'entraînement dans la fonction `train_full`

On réutilise le code du précédent TP. Il est le suivant :

```
x_prev = x0
xt = []
yt = []

for t in range(T + 1):
    xt.append(x_prev)
    x_next = prop(x_prev).sample(sample_shape=torch.Size([
            1])).squeeze(0)
    yt.append(obs(x_next).sample(sample_shape=torch.Size([
    1])).squeeze(0))
    x_prev = x_next

xt = torch.stack(xt, dim=0)
yt = torch.stack(yt, dim=0)
```

4.4 Question 2.4 : Calcul et minimisation de la loss d'entraînement

4.4.1 Loss globale

La perte globale se définit par :

$$\mathcal{L} = \frac{1}{T} \sum_{t \leq T} \mathcal{L}_t(q_t^b) + \mathcal{L}_t(q_t^a) + \mathcal{L}_0(q_0^a)$$

4.4.2 Variation du paramètre `deep`

Le paramètre `deep` représente la profondeur du réseau. Les résultats (pour 1200 itérations), en fonction de différentes valeur de `deep`, sont résumés dans cette table:

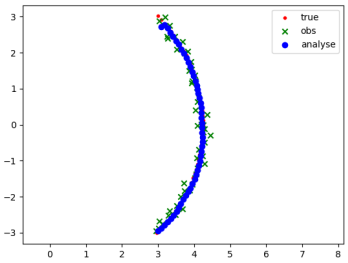
	Deep = 1		Deep = 3		Deep = 5		Deep = 10	
	1 iter	Last iter	1 iter	Last iter	1 iter	Last iter	1 iter	Last iter
RMSE_b	4.91	0.0324	3.67	0.0499	4.66	0.0303	4.18	0.0277
RMSE_a	4.89	0.0329	3.68	0.0421	4.65	0.0283	4.19	0.0269
LOGPDF_b	1184380	-3.29	8519779	-3.14	112460	-3.89	666902384	-4.03
LOGPDF_a	1073384	-3.48	7948544	-3.42	103977	-4.01	677130471	-4.09
LOSS	2257764	-6.77	16468324	-6.57	216438	-7.91	1344032856	-8.13

Table 1: Table des résultats de l'entraînement selon le paramètre `deep`

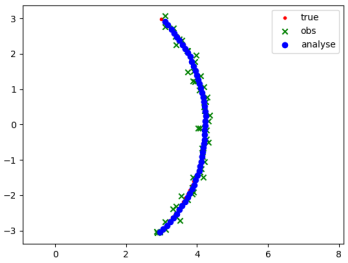
On observe que la perte et la RMSE diminuent à mesure que la valeur du paramètre deep augmente. L'augmentation de la valeur de deep signifie que le

réseau dispose de plus de couches cachées, ce qui lui permet de capturer des représentations plus complexes des données.

4.4.3 Affichage de x_t , y_t and μ_t^a avec un échantillon d'entraînement pour $t \leq T$



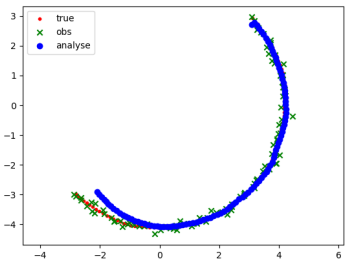
(a) Plot des données d'entraînement pour `deep` = 1



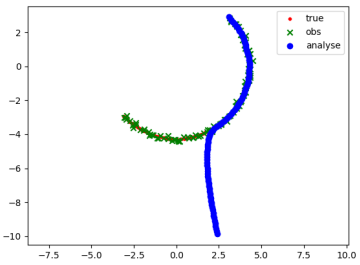
(b) Plot des données d'entraînement pour `deep` = 10

Que ce soit pour `deep` = 1 ou `deep` = 10, le DAN parvient à très bien apprendre les données d'entraînement et à retrouver la rotation pour $t \leq T$.

4.4.4 Affichage de x_t , y_t and μ_t^a avec un échantillon de test pour $t \leq 2T$



(a) Plot des données de test pour `deep` = 1



(b) Plot des données de test pour `deep` = 10

On observe que les prédictions de l'échantillon de test ne sont pas très correctes surtout pour celle entre T et $2T$. En effet, le DAN n'a pas été entraîné sur des données comprises entre T et $2T$, ce qui entraîne de mauvaises prédictions dans cet intervalle. De plus, on remarque un surapprentissage sur les données entre 0 et T.

En examinant les résultats des pertes, cette tendance se confirme. Pour `deep` $= 1$ et `deep` $= 10$, la perte finale atteint un ordre de grandeur de 10^6. Cette observation est également visible sur les graphes ci-dessous : pour les données de test, les RMSE ainsi que les pertes augmentent à partir de $t = 60$.

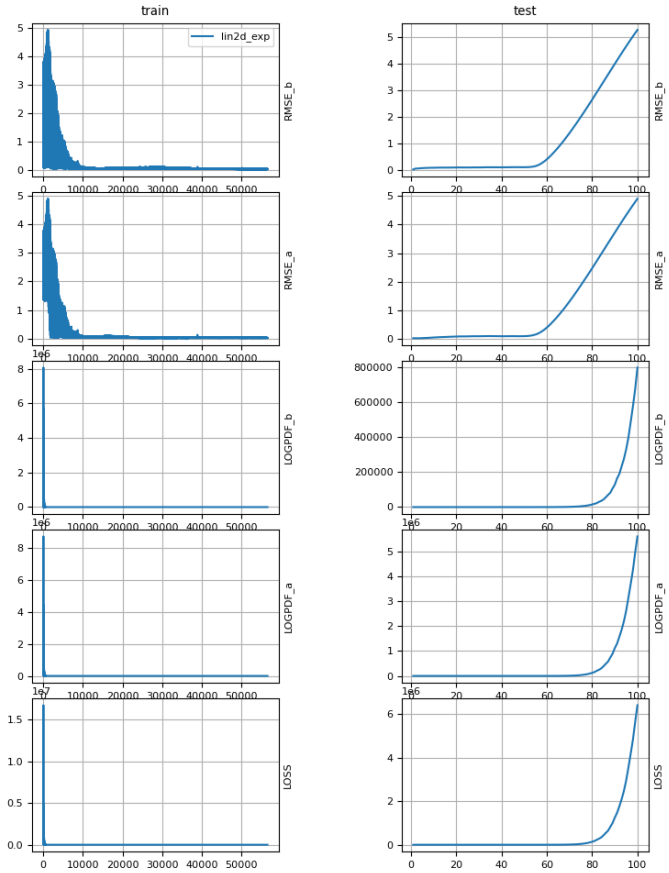


Figure 3: Evolution des RMSE et des pertes pour les données d'entraînement et de test pour `deep` $= 1$

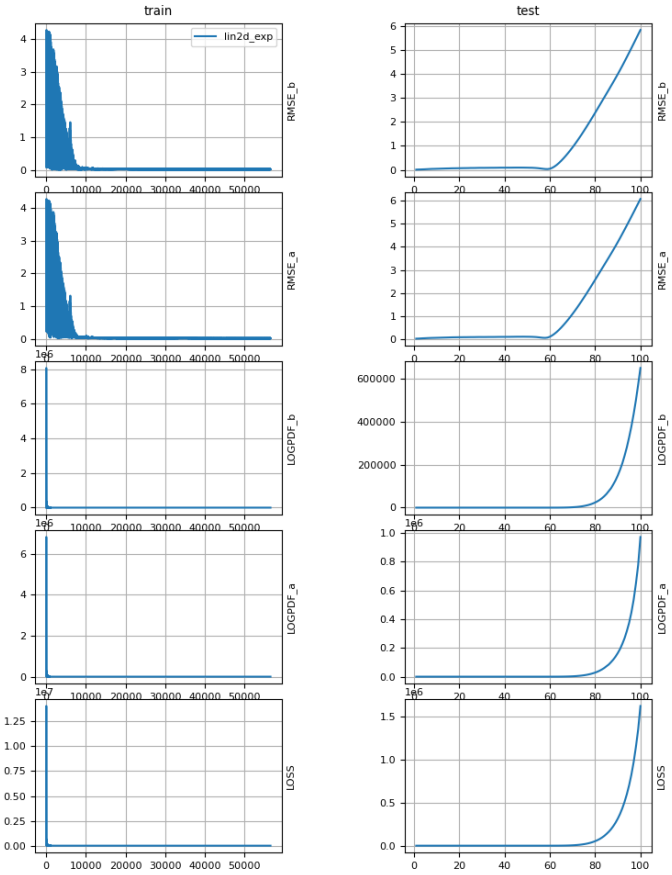


Figure 4: Evolution des RMSE et des pertes pour les données d'entraînement et de test pour `deep` = 10

De plus, on constate qu'on utilise pas la même instanciation de x_0 entre l'entraînement et le test. En effet, `get_x0_test` génère un nouvel x_0 différent à chaque appel, ce qui entraîne une variation des résultats d'un test à l'autre. De plus, comme x_0 est recréé pour chaque test, l'évaluation ne se fait pas sur les mêmes données que celles utilisées pendant l'entraînement.

5 Conclusion

Dans ce projet, on a entraîné un DAN pour modéliser un système physique de rotation en 2D. Le modèle fonctionne très bien sur les données d'entraînement, mais il généralise mal sur des données de test. En particulier sur des données différentes en temporalité que celles d'entraînement, cela suggère un problème

de sur-apprentissage. On a aussi remarqué que plus le modèle est profond, meilleurs sont les résultats d'entraînement mais cela ne s'étend pas aux données de test.