Implémentation d'un DAN (Data Assimilation Network)

Apprentissage sous contraintes physiques

Ines BESBES

April 7, 2025

Contents

1 Introduction

Lors de ce TP, on a implémenté un modèle d'apprentissage automatique appelé DAN (Data Assimilation Network), qu'on a entraîné en mode *full*, c'est-à-dire en utilisant l'ensemble des données disponibles pour ajuster les paramètres du modèle sur toute la séquence temporelle.

Le modèle DAN est conçu pour traiter des données séquentielles, ce qui le rend idéal pour les systèmes dynamiques où les observations sont partiellement disponibles ou bruitées. Dans notre cas, on a appliqué DAN à un système linéaire en deux dimensions (Linéaire 2D).

On présentera donc dans ce rapport les étapes de l'implémentation et de l'entraînement du modèle, ainsi que les résultats obtenus.

2 Préparer notre jeu de données pour DAN

L'objectif est d'implémenter les étapes de propagation et d'observation, deux étapes clés en assimilation de données. On a :

- **Étape de propagation** :

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

où η_t est un bruit blanc gaussien $\mathcal{N}(0, \sigma_p I)$ et M une matrice de rotation 2x2 définie comme suit :

$$M = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$$

- **Étape d'observation** :

$$y_t = Hx_t + \epsilon_t$$

avec H qui correspond à l'observation, et ϵ_t est un bruit blanc gaussien $\mathcal{N}(0, \sigma_o I)$. Or $H = I$, donc c'est directement l'état du système.

3 Architecture du DAN

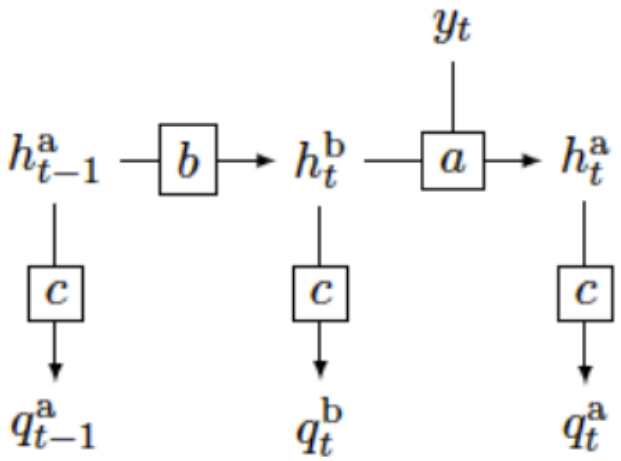

Figure 1: Architecture du DAN

L'architecture du DAN repose sur trois modules principaux notés a, b, et c, construits dynamiquement à l'aide des classes `ConstructorA`, `ConstructorB` et `ConstructorC` dans *filters.py*.

D'abord, le module b reçoit en entrée l'état caché de l'instant précédent h^a_{t-1} et produit une prédiction a priori de l'état suivant, notée h^b_t. Ce vecteur représente une estimation de l'état au temps t sans avoir accès à l'observation courante y_t.

Ensuite, le module a intègre cette prédiction avec l'observation y_t pour produire un état corrigé h^a_t, ce qui correspond à une étape de mise à jour. Chacun de ces états est ensuite passé dans un module c, qui génère une distribution gaussienne multivariée q modélisant l'incertitude associée à cet état (avec la classe **Gaussian** dans *filters.py*).

On a donc $q^b_t = P(x_t|y_{1:t-1})$ qui représente la distribution a priori de l'état x_t conditionnée par les observations passées $y_{1:t-1}$. De même, $q^a_t = P(x_t|y_{1:t})$ représente la distribution a posteriori de l'état x_t conditionnée par toutes les observations jusqu'à l'instant t.

4 Pré-entraînement de *c* pour le cas Linéaire 2D

4.1 Compréhension de la fonction *experiment*

La fonction *experiment* dans *manage_exp.py* permet d'entraîner et de tester le DAN, soit en mode *full* ou en mode *online*. Elle utilise les constructeurs ConstructorProp, ConstructorObs. Ces constructeurs vont construire des propagateurs et des observateurs pour le modèle, afin de faciliter l'apprentissage et l'évaluation du modèle.

Si le mode d'entraînement est "full", la fonction effectue un pré-entraînement complet du réseau, suivi d'un entraînement complet sur l'ensemble des données.

Après l'entraînement, la fonction réinitialise les scores du réseau pour s'assurer qu'ils ne sont pas influencés par les données d'entraînement.

Pour le test, elle utilise de nouveaux paramètres pour évaluer la performance du réseau et vérifier que le modèle généralise bien aux nouvelles données.

4.2 Constructeurs

Les constructeurs ConstructorProp et ConstructorObs permettent de générer des distributions gaussiennes qui modélisent les incertitudes associées aux états du système et aux observations.

4.2.1 ConstructorProp

Le constructeur ConstructorProp crée un propagateur, un modèle qui fait évoluer l'état dans le temps. Ce dernier génère une distribution gaussienne pour l'état futur x_{t+1} basé sur l'état actuel. En effet, il prend en entrée un certain état et renvoie une distribution gaussienne caractérisée par une moyenne (loc) et un écart-type (scale_vec). Ainsi, en concaténant ces deux valeurs, on obtient une distribution gaussienne (avec la classe GaussianDiag), qui représente le nouvel état après propagation.

4.2.2 ConstructorObs

De même, le constructeur ConstructorObs construit un observateur, un modèle qui transforme un état caché en une observation. Comme pour ConstructorProp, il génère une distribution gaussienne définie par une moyenne et un écart-type. Cependant, ici, la moyenne est une transformation identitaire (avec la classe Id) donc l'observation se base directement sur l'état caché.

4.3 Implémentation de la classe *Gaussian*

Comme on a vu précédemment, le constructeur c retourne un module *Gaussian*. En effet, on transforme le vecteur d'entrée en une distribution gaussienne de moyenne μ et de matrice de covariance Λ.

Ensuite, pour prévenir les instabilités numériques lors du calcul de l'exponentiation et de la fonction de perte, on applique un seuil minimum et maximum aux termes diagonaux de la matrice Λ. Enfin, on calcule la probabilité logarithmique d'un échantillon x.

Soit une variable aléatoire $x \in \mathbb{R}^d$ suivant une distribution normale multivariée $\mathcal{N}(\mu, \Sigma)$, avec $\Sigma = \Lambda\Lambda^\top$ où Λ est une matrice triangulaire inférieure. On a :

$$\log p(x) = -\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu) - \frac{1}{2}\log\det(2\pi\Sigma)$$

En utilisant la factorisation de Cholesky, cette expression devient :

$$\log p(x) = -\frac{1}{2}\left\|\Lambda^{-1}(x-\mu)\right\|^2 - \sum_{i=1}^{d}\log\Lambda_{ii} - \frac{d}{2}\log(2\pi)$$

Dans le code, cela correspond à deux termes principaux :

- **Terme 1** : $-\frac{1}{2}\left\|\Lambda^{-1}(x-\mu)\right\|^2$, obtenu en résolvant $z = \Lambda^{-1}(x-\mu)$ via une résolution triangulaire.
- **Terme 2** : $-\sum_{i=1}^{d}\log\Lambda_{ii}$, correspondant au log-déterminant de la matrice de covariance (via la trace du log de Λ).

Un terme constant $\frac{d}{2}\log(2\pi)$ est aussi ajouté dans le code sous la forme : `logcst = np.log(2*np.pi)/2`.

4.4 Implémentation de la loss $L_0(q_0^a)$ dans la fonction *closure*0

On suppose que x_0 suit une distribution de probabilité estimée par le réseau DAN : $x_0 \sim q_0^a(x)$, où q_0^a est une loi gaussienne $\mathcal{N}(\mu, \Sigma)$ paramétrée à partir de l'état caché h_0^a.

Cette hypothèse permet d'optimiser les paramètres du modèle en maximisant la vraisemblance des observations : on cherche à ajuster q_0^a pour qu'elle explique au mieux les données observées. Cela est équivalent à minimiser -log $q_0^a(x_0)$. On a :

$$L_0(q_0^a) = -\int \log q_0^a(x_0) p(x_0)\, dx_0$$

D'où :

$$L_0(q_0^a) = -\mathbb{E}_{x_0 \sim q_0^a(x)}[\log q_0^a(x_0)]$$

On obtient les résultats suivants :

Moyenne initiale ($\mathbf{a_0}$)	(2.9999, 3.0009)
Variance initiale ($\mathbf{a_0}$)	(0.00012433, 0.000098501)
Covariance initiale ($\mathbf{a_0}$)	$\begin{pmatrix} 0.00012433 & 0.000020100 \\ 0.000020100 & 0.00010175 \end{pmatrix}$

Table 1: Statistiques initiales pour $\mathbf{a_0}$.

4.5 Optimisation du module *Gaussian*

Ici, on cherche à accélerer le code du module *Gaussian*. Pour ce faire, on a utilisé les fonctions **torch.bmm** et **torch.triangular_solve** de Pytorch. La fonction **torch.bmm** permet d'effectuer des produits matriciels par lots, ce qui signifie que plusieurs matrices peuvent être multipliées simultanément. Cette approche est beaucoup plus efficace que de traiter chaque matrice individuellement dans une boucle. De même, **torch.triangular_solve** permet résoudre des systèmes linéaires triangulaires par lots, ce qui est meilleur pour le calcul de l'inverse de matrices triangulaires inférieures. On obtient les résultats suivants :

- Avec la boucle : 0.831 secondes
- Avec les fonctions Pytorch : 0.002 secondes

4.6 Génération de x_0

Les résultats obtenus lors de l'entraînement et du test du modèle peuvent varier en fonction de la valeur initiale car elle est générée de manière aléatoire.

Pour l'entraînement (get_x0):

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

Pour le test (get_x0_test):

$$x_{0\_test} = 3 \cdot \mathbf{1}_{(b\_size, x\_dim)} + \sigma \cdot \mathcal{N}(0,1) \tag{2}$$

Où :

- $\mathbf{1}_{(b\_size, x\_dim)}$ représente une matrice de 1 de taille (b_size, x_dim).
- $\mathcal{N}(0,1)$ représente un bruit gaussien. Cela introduit une composante aléatoire dans l'initialisation.
- σ est un paramètre qui contrôle l'amplitude du bruit ajouté.

Ainsi, cette part d'aléatoire montre que chaque exécution du modèle peut commencer avec un point de départ différent, influençant donc la trajectoire d'apprentissage et les performances finales.

5 Entraînement complet de *a, b, c* pour le cas Linéaire 2D

5.1 Initialisation de FcZero

La classe $FcZero$ implémente un réseau de neurones fully connected qui apprend des transformations non linéaires pour l'assimilation des données. Ce réseau utilise la technique ReZero, une technique de normalisation qui permet d'amélirer l'entrainement des réseaux profonds. Pour ce faire, elle modifie la sortie de chaque bloc en y ajoutant directement l'entrée du bloc. Chaque couche est de la forme :

$$h = h + \alpha * act(lin(h)).$$

Dans l'initialisation de la classe $FcZero$, il y avait une erreur :

```
self.alphas =  torch.zeros(deep)
```

qu'on a modifié par :

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

Cela permet de définir α comme un paramètre apprenable du modèle, et qui nécessite une rétropropagation des gradients lors de son optimisation.

5.2 Implémentation des forward steps de (a,b,c)

Dans la classe DAN, la fonction $forward$ a été implémenté suivant ces étapes :

- Calculer la perte (et le gradient) sur t.
- Initialisation h_0^a
- Entrée : h_{t-1}^a, x_t, y_t
- Sortie : $L_t(q_t^b) + L_t(q_t^a), h_t^a$
- Étapes internes clés :
  - Calculer $h_t^b = b(h_{t-1}^a)$
  - Calculer $q_t^b = c(h_t^b)$

– Calculer $h_t^a = a(h_t^b, y_t)$

– Calculer $q_t^a = c(h_t^a)$

5.3 Calcul de la loss

La fonction de perte totale (loss) est définie comme suit :

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

6 Résultats

6.1 Etude de l'influence du paramètre *deep*

Le paramètre *deep* gère le nombre de couches dans le réseau. Il contrôle donc la profondeur du réseau. On va donc faire varier ce paramètre pour connaitre son influence sur les résultats obtenus.

6.1.1 Sur l'ensemble d'entrainement

Profondeur	Itération	RMSE_b	RMSE_a	LOGPDF_b	LOGPDF_a	LOSS
Deep = 1	Première	4.118	4.117	2.97e6	3.00e6	5.97e6
	Dernière	0.041	0.035	-3.234	-3.470	-6.704
Deep = 5	Première	4.590	4.608	9.73e5	7.20e5	1.69e6
	Dernière	0.030	0.030	-3.878	-3.845	-7.723
Deep = 10	Première	4.838	4.829	1.11e5	1.36e5	2.47e5
	Dernière	0.028	0.027	-4.012	-4.147	-8.159
Deep = 20	Première	4.349	4.352	6.09e6	5.30e6	1.14e7
	Dernière	0.029	0.028	-4.014	-4.096	-8.110

Table 2: Résultats d'entraînement par profondeur

On observe que toutes les profondeurs ont des performances satisfaisantes, avec une RMSE ≤ 0.04, et que les modèles profonds sont meilleurs (deep=10/20 atteignent une RMSE=0.028). En effet, plus le nombre de couches augmente, plus la RMSE et la perte diminuent. De plus, la décroissance exponentielle des LOGPDF initiaux, de 2.97e6 à -4.1, montre que l'apprentissage s'est bien stabilisé.

6.1.2 Sur l'ensemble de test

Profondeur	Itération	RMSE_b	RMSE_a	LOGPDF_b	LOGPDF_a	LOSS
Deep = 1	Première	0.074	0.032	-2.087	-3.830	-5.917
	Dernière	0.420	0.358	12.388	6.383	18.771
Deep = 5	Première	0.014	0.016	-5.427	-5.109	-10.536
	Dernière	0.389	0.484	1.05e6	2.44e6	3.49e6
Deep = 10	Première	0.013	0.017	-5.233	-4.899	-10.132
	Dernière	2.598	2.681	1.24e5	2.62e5	3.86e5
Deep = 20	Première	0.013	0.019	-5.134	-4.788	-9.922
	Dernière	10.803	10.374	1308.113	396.549	1704.661

Table 3: Résultats de test par profondeur

Comme pour l'ensemble d'entraînement, plus le paramètre deep est grand mieux sont les résultats. Toutefois, seul le modèle de profondeur deep=1 maintient une stabilité relative, tandis que les autres modèles divergent radicalement (RMSE=10.8, LOGPDF=+1704). De plus, les résultats à la première itération sont très bons, mais ils divergent à la dernière itération. Ceci indique un overfitting, le modèle n'est plus capable de généraliser à partir d'un certain temps.

6.2 Résultats obtenus avec T=50

Les résultats obtenus précédemment confirment les observations suivantes : à partir d'un certain temps t, le modèle a des difficultés à généraliser, surtout pour les profondeurs 10 et 20.

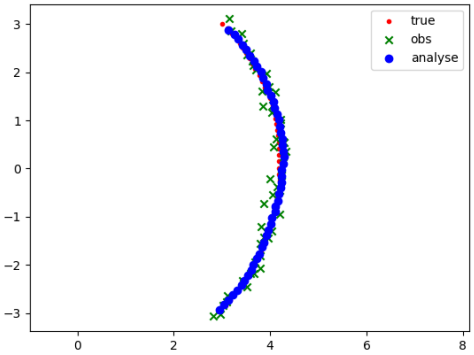

Figure 2: Train avec deep=1 : x_t, y_t et μ_t^a, $t \leq T$

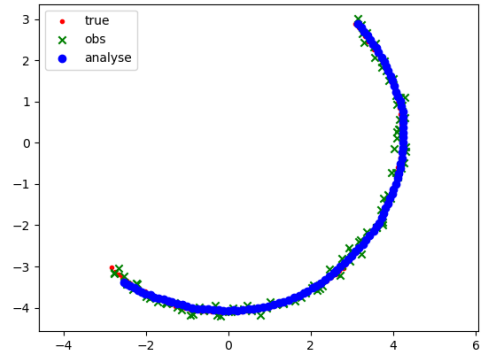

Figure 3: Test avec deep=1 : x_t, y_t et μ_t^a, $t \leq 2T$

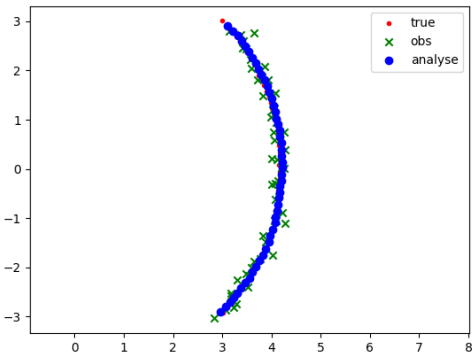

Figure 4: Train avec deep=5 : x_t, y_t et μ_t^a, $t \leq T$

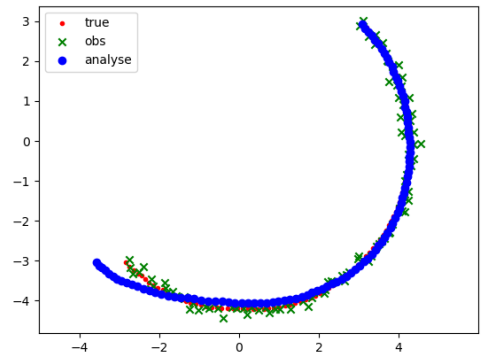

Figure 5: Test avec deep=5 : x_t, y_t et μ_t^a, $t \leq 2T$

En revanche, on observe une certaine stabilité pour les profondeurs 1 et 5 sur les ensembles de test. Mais ces résultats peuvent varier en fonction du paramètre initial x_0 car il est choisi de manière aléatoire et peut influencer les performances du modèle.

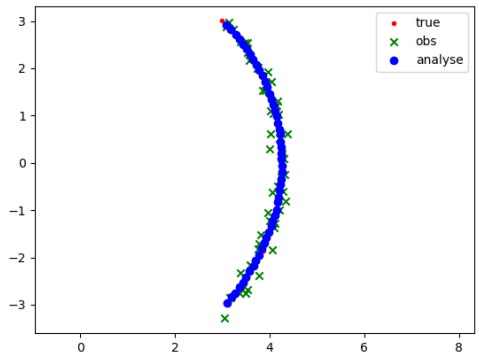

Figure 6: Train avec deep=10 : x_t, y_t et μ_t^a, $t \leq T$

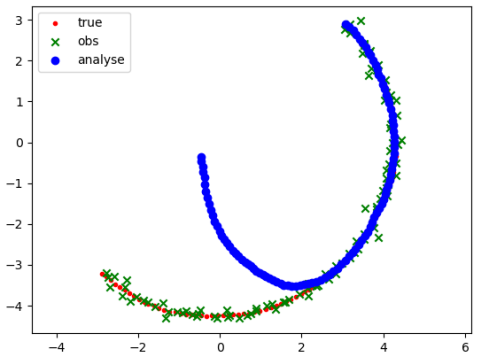

Figure 7: Test avec deep=10 : x_t, y_t et μ_t^a, $t \leq 2T$

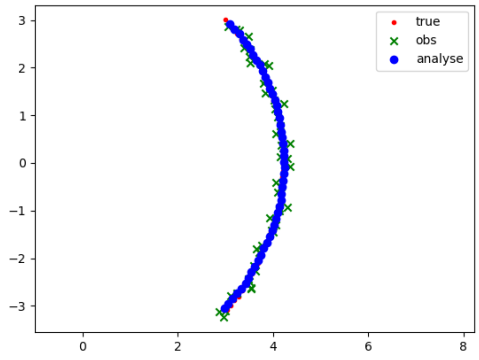

Figure 8: Train avec deep=20 : x_t, y_t et μ_t^a, $t \leq T$

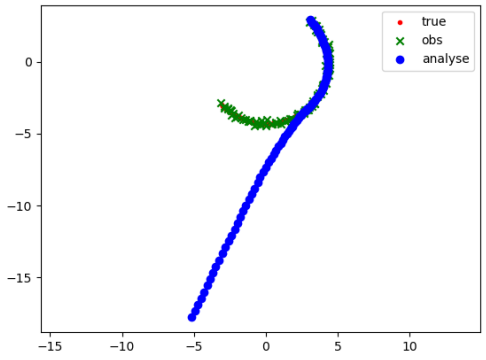

Figure 9: Test avec deep=20 : x_t, y_t et μ_t^a, $t \leq 2T$

Pour deep=10 et deep=20, l'overfitting est très visible. De fait, un modèle avec de telles profondeurs est probablement trop complexe pour les données fournies. D'où le fait qu'il puisse plus mémoriser les données d'entraînement plutôt que d'apprendre les tendances générales. Ceci est bien visible sur le graphe avec les données d'entraînement, où la courbe bleue suit parfaitement les points rouges, mais les points verts sont un peu dispersés autour de la courbe.

De plus, l'overfitting observé peut s'expliquer en partie par la différence entre la durée des séquences utilisées pendant l'entraînement et celles utilisées pendant le test. En effet, le modèle est entraîné avec des séquences de longueur T = 50, mais il est évalué sur des séquences plus longues, de T = 100. Cela signifie qu'il apprend à bien se comporter sur les 50 premiers pas de temps, mais n'a jamais vu de situations au-delà de cette durée pendant l'apprentissage. Lorsqu'on lui demande de prédire sur 100 pas, il peut se retrouver en difficulté, ce qui entraîne une dégradation progressive de ses performances. Cette différence entre les durées d'entraînement et de test rend donc la généralisation plus difficile.

6.3 Résultats pour $deep = 1$

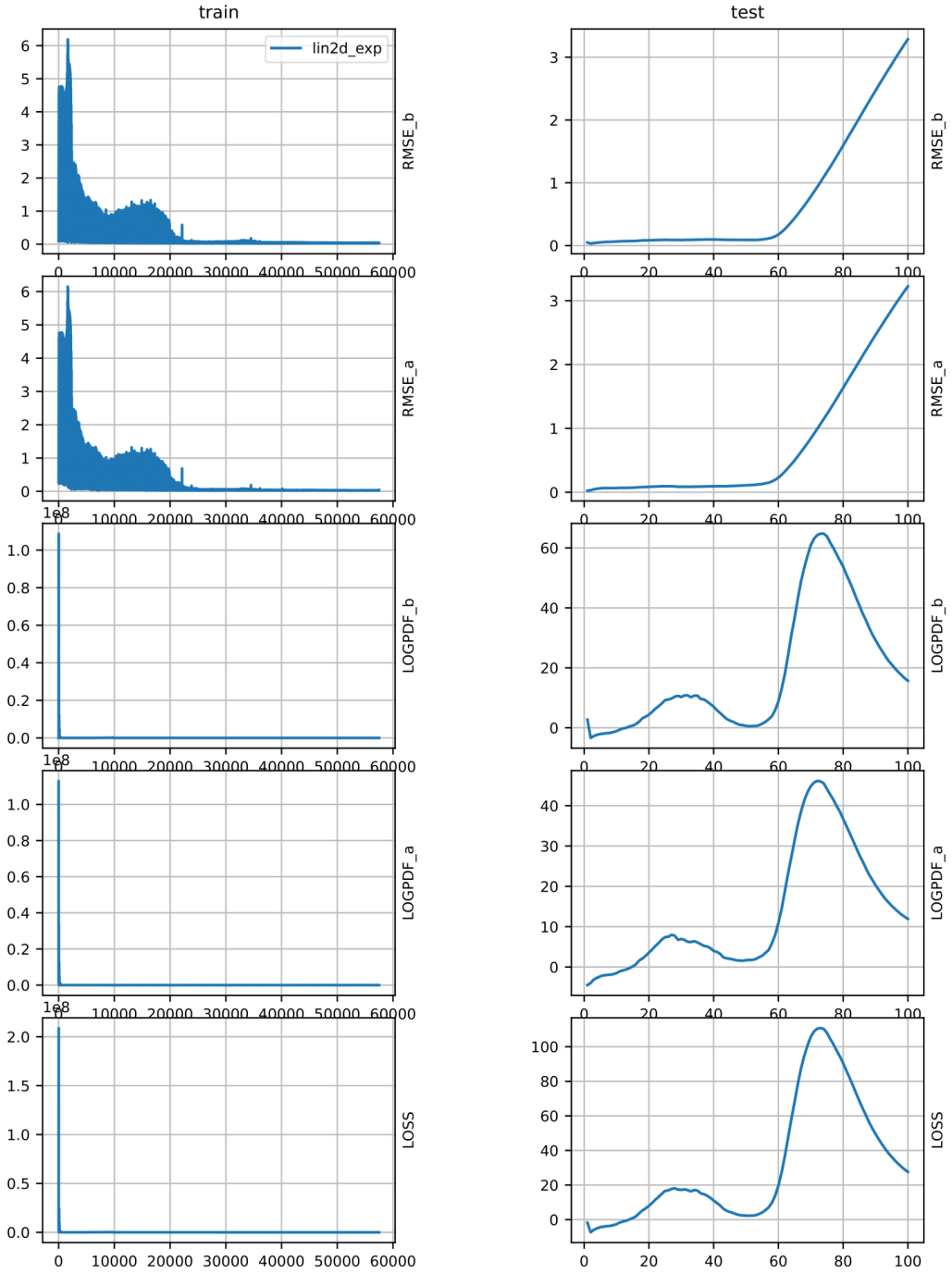

Figure 10: Visualisation des différentes métriques pour le train et le test

Quand on utilise une profondeur deep = 1, le réseau montre des performances assez limitées. Pendant l'entraînement, l'erreur diminue mais reste instable, avec des variations visibles, ce qui indique que le réseau a du mal à bien apprendre. La loss finit par baisser.

En test, les résultats sont clairement moins bons : l'erreur augmente rapidement au fil du temps, allant au-delà de 3, et les mesures de confiance du modèle deviennent très élevées, ce qui signifie qu'il n'est pas sûr de ses prédictions. Ainsi, le réseau n'arrive pas à bien suivre l'évolution de l'état caché sur une longue durée. Cela montre que deep = 1 n'est pas suffisant pour modéliser correctement la dynamique

du système.

6.4 Résultats pour $deep = 10$

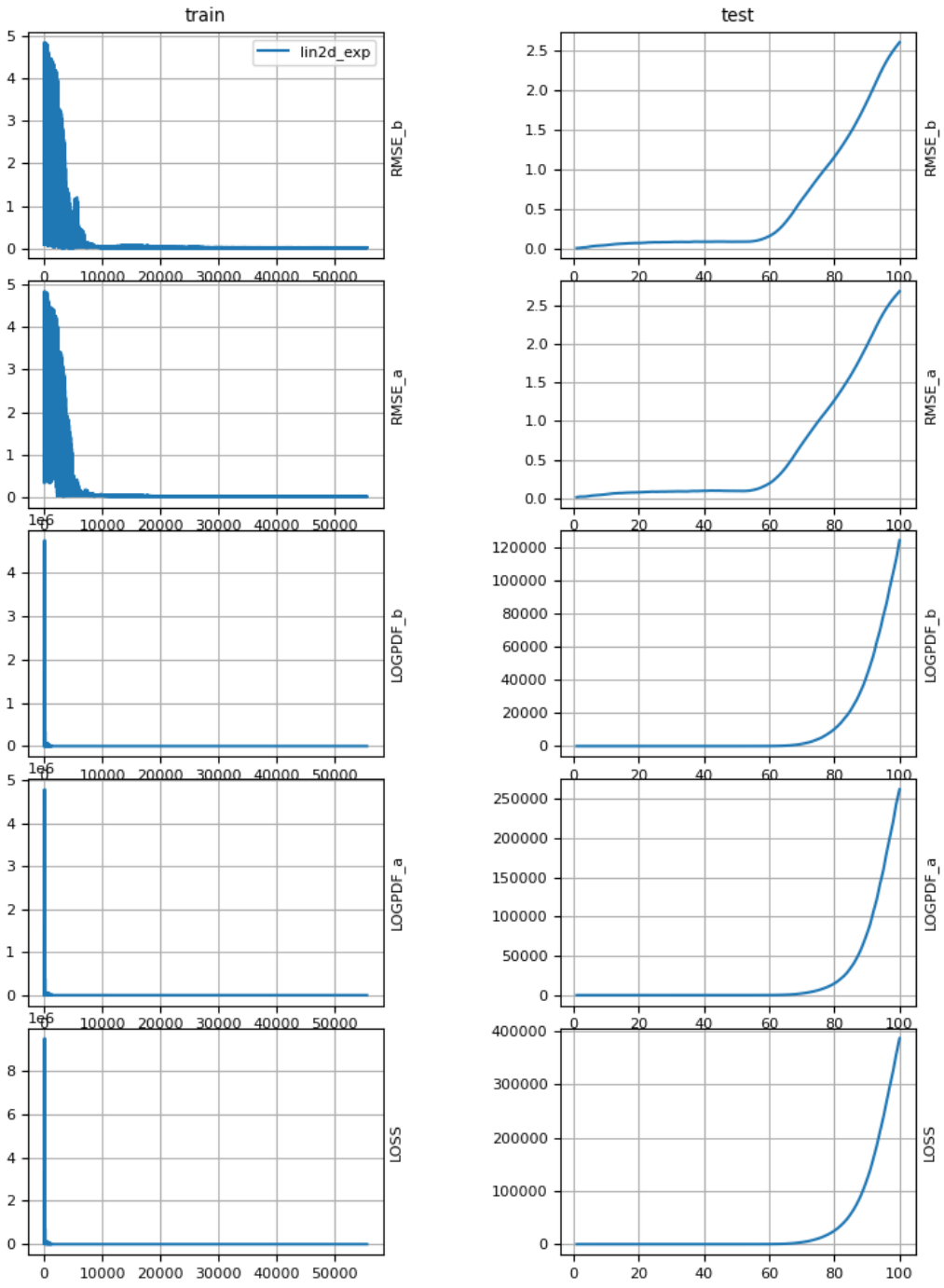

Figure 11: Visualisation des différentes métriques pour le train et le test

Pendant l'entrainement, on observe une convergence rapide des RMSE durant les premières 5000 itérations, suivie d'une stabilisation jusqu'à 55000 itérations. Cela indique que le modèle apprend efficacement à représenter les dynamiques du système.

Toutefois, les graphiques de test montrent que le réseau maintient une faible erreur jusqu'à environ 60 pas de temps, puis on observe d'un coup une augmentation exponentielle des erreurs et des pertes.

Cela témoigne d'un overfitting. Pour améliorer les performances, on peut envisager d'introduire de la régularisation dans le réseau (comme du dropout), ou réduire la profondeur du réseau.

7 Conclusion

En conclusion, le DAN présente une bonne capacité d'apprentissage sur les données d'entraînement, mais souffre d'overfitting, limitant sa performance et sa fiabilité à long terme sur des données de test. Des ajustements sont nécessaires pour améliorer sa capacité de généralisation.