Rapport TP Apprentissage sous contraintes physiques

Implémentation d'un DAN

Année 2024

MUSSARD Cassandra

Table des matières

1 Introduction

Lors du TP 3 nous avons vu comment implémenter les phases d'assimilations et de propagations d'un système d'assimilation de données 2D. Dans ce TP, nous allons voir comment faire pour entraîner un Data Assimilation Network (DNN) pour réaliser cela. Nous allons implémenter cet entraînement avec deux modes : "full" et "online".

2 Description du modèle

Comme expliqué précédemment en assimilation de données nous avons deux phases : une phase d'assimilation et une phase de propagation.
Dans notre cas 2D, la phase de propagation est définie comme :

$$x_t = M * x_{t-1} + \eta_t$$

avec M la matrice de rotation 2x2 définie par :

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

et η un bruit blanc gaussien représentant l'incertitude sur la prédiction du système définit par $\eta_t \sim \mathcal{N}(0, \sigma_p I)$

Ensuite la phase d'observation est construit tel que :

$$y_t = Hx_t + \epsilon_t$$

avec H représentant l'observation mais comme $H = I$ c'est directement l'état du système. Enfin ϵ_t représente l'erreur de mesure modélisée par un bruit blanc gaussien tel que $\epsilon_t \sim \mathcal{N}(0, \sigma_0 I)$

3 Pre-training of c for Linear 2d

3.1 TODO 1.1

3.1.1 Fichier manage_exp

Le fichier manage_exp gère les états initiaux en initialisant les états x_0 pour l'ensemble d'entraînement et de test et l'état h_0^a. Il permet aussi de lancer le pré-entraînement du module c (qui va modéliser la distribution initiale des états x_0) ainsi que l'entraînement complet des modules a, b et c. Enfin, il évalue les performances du réseau DAN entraîné sur un ensemble de données de test.

3.1.2 Fichier filters

Dans ce fichier, on va créer trois modules a, b et c. Le module a va effectuer une mise à jour de l'état du système en intégrant les informations provenant des observations. Ensuite le module b va se charger de propager l'état du système d'un instant au suivant. Enfin, le module c calcule la distribution probabiliste des états du système.
Pour continuer, on retrouve la classe DAN qui permet d'initialiser les modules a, b et c.
Enfin, les modules ConstructorProp et ConstructorObs vont construire des propagateurs et des observateurs personnalisés pour le modèle.

3.2 TODO 1.2

Cette partie est implémentée dans le fichier filters à la ligne 263. On va transformer le vecteur d'entrée en une distribution gaussienne de moyenne μ et de matrice de covariane Λ. On applique ensuite un seuil minimum et maximum aux termes diagonaux de la matrice Λ pour prévenir les instabilités numériques. Enfin on calcule la probabilité logarithmique $\log(p(x))$ des échantillons par :

$$\log p(x) = -\frac{1}{2}(z^T z) - \sum_{i=1}^{n} \frac{\log(2\pi)}{2} + \tilde{\Lambda}$$

avec $z = L^{-1}(x - \mu)$ solution du système $Lz = (x - \mu)$

3.3 TODO 1.3

Dans cette partie, on demande de comprendre comment est générer x0. Ceci est fait dans le fichier manage_exp plus précisemment dans la fonction get_x0. x0 est une matrice de taille [b_size, x_dim] avec b_size la taille du batch et $x\_dim = 2$. Au début, x0 est initialisé de la façon suivante :

$$x0 = 3 * ones(b_{size},) + \sigma * + sigma * torch.randn(b_{size}, x_{dim})$$

.
Ensuite, la classe Lin2d est implémentée comme au TP3.

3.4 TODO 1.4

Dans cette partie nous implémentons la loss L0 qui est définie par :

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

Par estimation de Monte-Carlo on obtient alors :

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

avec I le nombre d'expériences.
On obtient les résultats suivants :

```
## INIT a0 mean tensor([3.0001, 2.9999], grad_fn=<SliceBackward0>)
## INIT a0 var tensor([8.7407e-05, 9.4784e-05], grad_fn=<SliceBackward0>)
## INIT a0 covar tensor([[ 8.7407e-05, -8.0081e-06],
        [-8.0081e-06,  9.5518e-05]], grad_fn=<SliceBackward0>)
```

FIGURE 1 – Moyenne, variance et matrice de covariance de pdf_a0

De plus, $L_0 = -6.4665$.

3.5 TODO 1.5

Dans cette partie l'objectif est d'accélérer le code. Pour réalier cela nous avons utilisé des fonctions de pytorch permettant de faire du calcul matricielle par batch en utilisant par exemple la fonction torch.bmm. Ensuite, les matrices de covariance et les déterminants log sont stockés pour l'ensemble du batch en une seule opération.
En ce qui concerne l'accélération du code voici les résultats obtenus :
Avec amélioration : **0.001s**
Sans amélioration : **0.8534s**

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

4.1 TODO 2.1

La classe FcZeros définit un réseau de neurones Fully Connected avec l'utilisation de la technique ReZero. Cette technique est une stratégie de normalisation pour améliorer l'entraînement des réseaux profonds. La méthode ReZero modifie la sortie de chaque bloc en y ajoutant directement l'entrée du bloc. Ceci est pondéré par un paramètre α, initialisé à zéro au début de l'entraînement et qui est appris par le réseau.
Dans le code fournit il y avait une erreur qui est la suivante :

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

Ceci a été modifé par :

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

Où on définit alphas comme un paramètre du modèle qui est appris par celui-ci et qui va nécessiter une rétropropagation des gradients lors de l'optimisation de ce paramètre (requires_grad = True).

4.2 TODO 2.2

L'implémentation de la fonction forward de la classe DAN a été implémentée dans le fichier filters.py de la ligne 37 à 51 de la manière suivante :

- Input: $h^{\mathbf{a}}_{t-1}, x_t, y_t$
- Output: $\mathcal{L}_t(q^{\mathbf{b}}_t) + \mathcal{L}_t(q^{\mathbf{a}}_t)$, $h^{\mathbf{a}}_t$
- Key internal steps:
  - Compute $h^{\mathbf{b}}_t = \mathbf{b}(h^{\mathbf{a}}_{t-1})$
  - Compute $q^{\mathbf{b}}_t = \mathbf{c}(h^{\mathbf{b}}_t)$
  - Compute $h^{\mathbf{a}}_t = \mathbf{a}(h^{\mathbf{b}}_t, y_t)$
  - Compute $q^{\mathbf{a}}_t = \mathbf{c}(h^{\mathbf{a}}_t)$

FIGURE 2 – Implémentation de la fonction forward de la classe DAN

Avec comme loss finale :

$$\mathcal{L} = \sum_{t \leq T} (\mathcal{L}_t(qt^b) + \mathcal{L}_t(qt^a)) + \mathcal{L}_0(q0^a)$$

4.3 TODO 2.3

L'implémentation de cette partie se trouve dans le fichier manage_exp à la ligne 124.

4.4 TODO 2.4

Nous allons dans un premier temps faire varier le paramètre deep qui gère le nombre de couches du réseau.
Nous fixons le nombre d'itérations à 1000 pour l'entraînement et faisons varier uniquement ce paramètre deep.

TABLE 1 – Résultats obtenus pour différentes valeurs du paramètre deep à la première et dernière itération pour le dataset d'entraînement

	Deep=1		Deep=5		Deep=10	
	1ère itération	Dernière it.	1ère itération	Dernière it.	1ère itération	Dernière it.
RMSE b	5.69	0.065	3.97	0.029	4.43	0.024
RMSE a	5.69	0.043	3.98	0.028	4.43	0.024
LOGPDF_b	1668329.01	-2.31	395083.47	-3.95	149930.94	-4.24
LOGPDF_a	1355690.21	-2.30	316689.98	-4.033	113462.45	-4.34
LOSS	3024019.23	-4.62	711773.45	-7.99	263393.40	-8.58

On remarque que plus le nombre de couches augmente (deep = 10) plus la loss diminue ainsi que la RMSE. Finalement, les meilleurs résultats sont obtenus pour deep =10 avec 1000 itérations.

TABLE 2 – Résultats obtenus pour différentes valeurs du paramètre deep à la première et dernière itération pour le dataset de test

	Deep=1		Deep=5		Deep=10	
	1ère itération	Dernière it.	1ère itération	Dernière it.	1ère itération	Dernière it.
RMSE b	0.022	2.28	0.015	2.56	0.018	2.62
RMSE a	0.021	2.45	0.019	2.099	0.016	2.50
LOGPDF_b	-2.92	5908122646.72	-4.56	89270.46	-4.70	7578.41
LOGPDF_a	-3.09	1137594417.32	4.78	7521.86	-4.74	24229.08
LOSS	-6.02	7045717064.04	-9.35	96792.32	-9.45	31807.50

En ce qui concerne les résultats sur le dataset de test, on voit que à la première itération les résultats sont très bons mais à la dernière ils se dégradent fortement. Ceci est sûrement dû au fait que le réseau n'est plus capable de généraliser à un certain moment. Ensuite, comme pour le dataset d'entraînement, on remarque que plus le paramètre deep est grand mieux sont les résultats.

Pour Deep =1 on obtient les plots suivants :

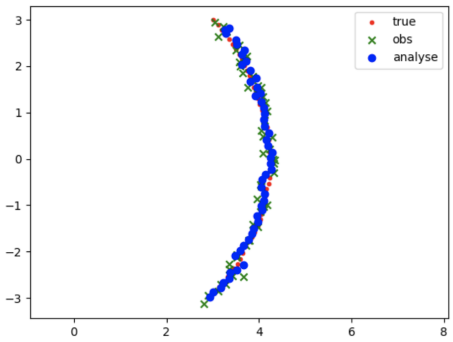

FIGURE 3 – Plot de x_t, y_t et h_t^a pour un échantillon du dataset d'entraînement et pour $t \leq T$

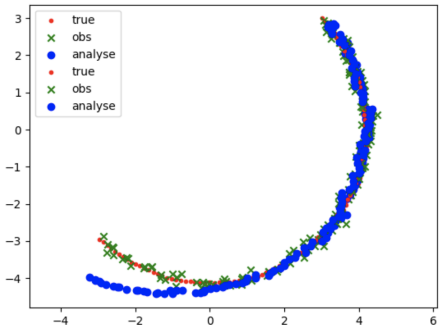

FIGURE 4 – Plot de x_t, y_t et h_t^a pour un échantillon du dataset de test et pour $t \leq 2 * T$

Pour deep = 5 :

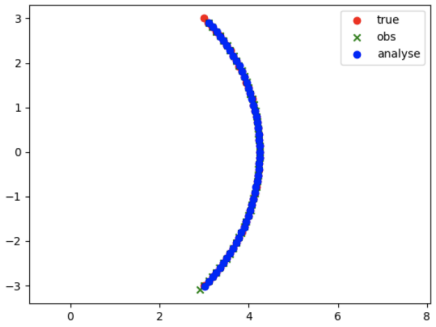

FIGURE 5 – Plot de x_t, y_t et h_t^a pour un échantillon du dataset d'entraînement et pour $t \leq T$

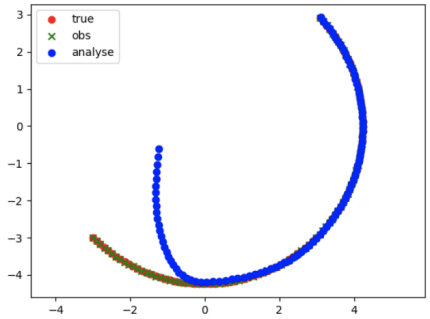

FIGURE 6 – Plot de x_t, y_t et h_t^a pour un échantillon du dataset de test et pour $t \leq 2 * T$

Pour deep =10 :

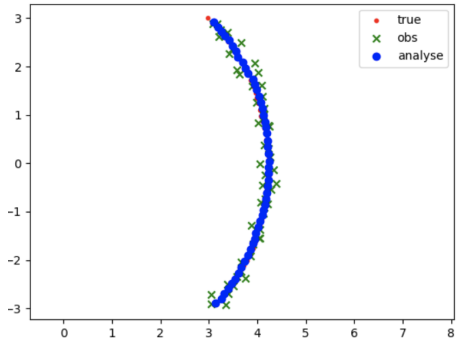

FIGURE 7 – Plot de x_t, y_t et h_t^a pour un échantillon du dataset d'entraînement et pour $t \leq T$

Sur ces plots on voit bien là aussi que pour le dataset d'entraînement on arrive bien à coller aux labels mais pour le test à partir d'un certain moment, les résultats se dégradent fortement.

Ensuite pour la RMSE nous obtenons les résultats suivants pour deep=1 avec 1000 itérations :

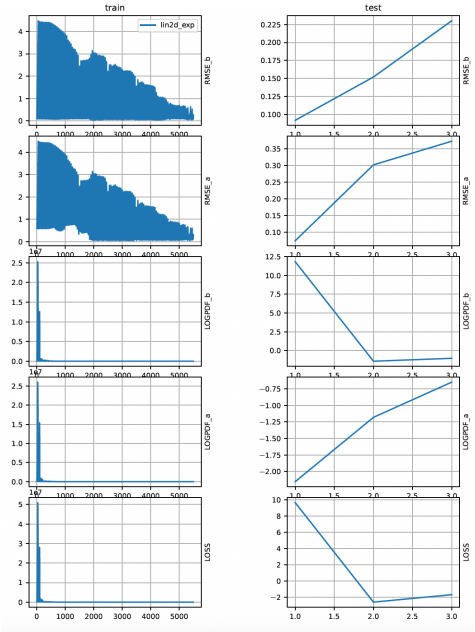

FIGURE 8 – RMSE, Logpdf et loss

5 Conclusion

En conclusion, nous avons réussi à entraîner un DAN dans le cas 2D. Notre réseau est très bon sur le dataset d'entraînement mais généralise assez mal sur le dataset test. Finalement, on remarque que le paramètre deep améliore les résultats.