Master 2 CSMI

Incertitude sur la densité dans l'équation de Burgers

Mélissa UNTZ

Superviseurs : Emmanuel FRANCK, Vincent VIGON, Laurent NAVORET

Année scolaire 2021-2022

Table des matières

1 Introduction

1.1 Contexte général

La réalité d'un système physique étudié n'est pas toujours décrite exactement par le modèle mathématique, et la prévision est entâchée d'une incertitude. En effet, le modèle est le fruit de plusieurs approximations, notamment l'erreur de discrétisation ou l'estimation des paramètres du modèle, qui ne sont en général pas connus exactement. Ici, nous nous concentrons sur l'incertitude sur la densité dans l'équation de Burgers. Le sujet du projet est proposé par Emmanuel FRANCK, chercheur à l'INRIA Nancy - Grand Est, en collaboration avec Vincent VIGON et Laurent NAVORET, chercheurs à l'Institut de Recherche Mathématique Avancée (IRMA).

L'INRIA est l'institut national de recherche en sciences et technologies du numérique. Le centre de recherche Inria Nancy - Grand Est a été créé en 1986 et est situé à l'université de Lorraine. Il possède également une antenne à Strasbourg qui accompagne le développement de l'université de Strasbourg [4].

1.2 Explication du sujet

Nous avons l'équation de Burgers suivante

$$\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \frac{1}{R_e} \partial_{xx} \rho$$

avec

$$\rho(t=0,x) = \begin{cases} 1.0, & \text{x <0.5} \\ a, & \text{x > 0.5} \end{cases}$$

Lois de probabilité :

$$a \sim \mathcal{U}[0.1, 0.3], \quad logR_e \sim \mathcal{U}[4, 13] \qquad \text{avec } \mathcal{U} \text{ une loi uniforme}$$

$\frac{1}{R_e}$ correspond au coefficient de viscosité, avec R_e le nombre de Reynolds. $x \mapsto \rho(t,x)$ est la densité.

Nous souhaitons quantifier l'incertitude sur la densité dans cette équation sur un intervalle de temps donné. On souhaite alors étudier la possibilité de représenter la loi de probabilité de ρ observée sur cet intervalle de temps lorsque l'on fait varier les paramètres R_e et a, à l'aide de modèles d'apprentissage profond tel que l'auto-encodeur variationnel. En l'appliquant à une densité ρ donnée, on aimerait générer la distribution des densités proches du ρ donné.

Le projet se divise en plusieurs étapes :

- On veut trouver G tel que $G(X(t)) = Y(t)$ avec X correspondant à $\rho \in \mathbb{R}^n$ et $Y \in \mathbb{R}^d, \quad d << n$.
  G correspond à un encodeur non déterministe et G^+ est le décodeur, c'est-à-dire $G^+(Y(t)) = X(t)$. Pour entraîner ces deux réseaux de neurones, on va utiliser le principe de l'auto-encodeur.
- On effectue une simulation de X, de manière à obtenir une trajectoire $X(t_0), ..., X(t_n)$ et on entraîne les réseaux de neurones sur ces données. Afin de générer les lois de $X(t_i)$, on utilise l'auto-encodeur entraîné. Pour cela on fait plusieurs tirages aléatoires du processus $G^+(G(X(t_i)))$.
- On observe si les moyennes et variances obtenues correspondent à ce qu'on obtient en utilisant Monte Carlo directement sur les simulations.

Dans un second temps, si le temps le permet, il s'agirait de ne pas générer une trajectoire complète $X(t_0), ..., X(t_n)$, contrairement à précédemment :

- De la même manière que dans l'étape 1, on entraîne les deux réseaux de neurones G et G^+.
- Cette fois, on entraîne le modèle réduit $Y' = F(Y(t))$ avec $Y(t) = G(X(t))$. On prend $X(t_0)$, et après l'encodeur on génère de nombreux $Y(t_0)$.
  Ensuite, il faut propager les $Y(t_0)$ avec le modèle pour obtenir des $Z(t)$ sur tous les temps. Enfin, le décodeur permet d'avoir de nombreux $X(t)$, on peut alors appliquer Monte Carlo.

1.3 Feuille de route

Pour avoir une vision globale du projet, j'ai créé une feuille de route ("road map"), résumée dans un diagramme de Gantt. Elle définit les différents tâches à effectuer pour atteindre les objectifs. Les dates approximatives des rendus intermédiaires y sont également affichées. Le diagramme est accessible sur le lien ici, et dans le *README.adoc* du repository GitHub.

Diagramme de Gantt UNTZ Mélissa

Intitulé du projet : Incertitudes – INRIA

Superviseurs : Emmanuel FRANCK, Vincent VIGON, Laurent NAVORET

	Détail des tâches	Date début	Date fin
1	**Génération de lois à partir des X(t_i)**	**13 / 10 / 21**	**14 / 01 / 22**
1.1	Compréhension du sujet et du VAE	13 / 10 / 21	01 / 11 / 21
1.2	Rapport V0	02 / 11 / 21	02 / 11 / 21
1.3	Implémentation du VAE sur Burgers	03 / 11 / 21	23 / 11 / 21
1.4	Entraînement de l'auto-encodeur	24 / 11 / 21	29 / 11 / 21
1.5	Rapport V1	30 / 11 / 21	30 / 11 / 21
1.6	Validation de l'auto-encodeur	01 / 12 / 21	07 / 01 / 22
1.7	Monte Carlo sur les lois générées	08 / 01 / 22	23 / 01 / 22
1.8	Rapport V2	20 / 01 / 22	24 / 01 / 22

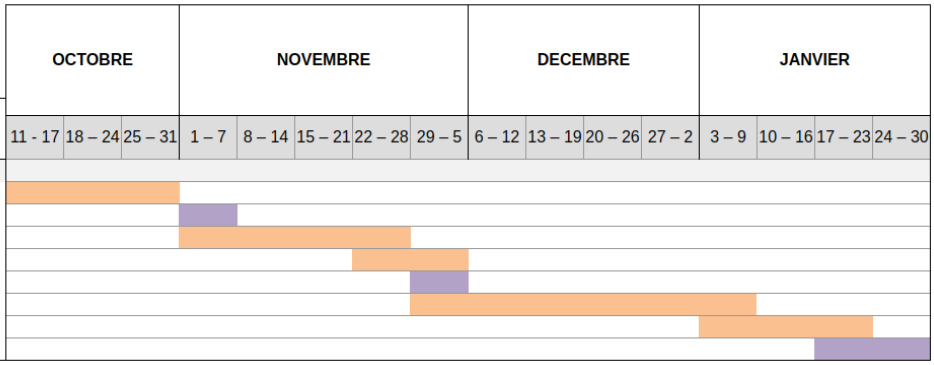

Figure 1 – Diagramme de Gantt

2 Comment générer des distributions ?

2.1 Principe du modèle génératif

On cherche à construire de nouveaux exemples à partir de données de grande dimension, et à évaluer la probabilité de ces générations. Pour cela, on utilise des modèles dits génératifs, que l'on définit juste en-dessous.

Soit $x \in V \subset \mathbb{R}^d$ une donnée observable et $y \in W \subset \mathbb{R}^f$ une donnée cible. On peut donner deux définitions du modèle génératif.

- Il s'agit d'une densité de probabilité paramétrique (discrète ou continue)

$$p_\theta(x|y), \qquad \text{avec} \int_{x \in V} p_\theta(x|y) = 1$$

  Si on possède un échantillon $X = (x_1, ..., x_n)$ et $Y = (y_1, ..., y_n)$, le modèle génératif doit être capable de produire les données de X connaissant Y.

- Il s'agit d'une densité de probabilité paramétrique (discrète ou continue)

$$p_\theta(x, y), \qquad \text{avec} \int_{x \in V, y \in W} p_\theta(x, y) = 1$$

Le modèle génératif cherchera à décrire la probabilité conditionnelle $p_\theta(x|y)$ ainsi que la probabilité $p_\theta(y)$ pour calculer la probabilité $p_\theta(y|x)$.

Le modèle génératif qui nous intéresse pour la suite est l'auto-encodeur variationnel, mais pour en parler il faut introduire la notion d'auto-encodeur. L'idée générale des auto-encodeurs consiste à configurer un encodeur et un décodeur comme réseaux neuronaux et à apprendre le meilleur schéma d'encodage-décodage en utilisant un processus d'optimisation itératif.

Tout d'abord, l'encodeur va permettre de condenser l'information disponible initialement (image, texte, audio, etc.) en extrayant des "features" (caractéristiques) qui caractérisent du mieux possible l'information initiale. Le vecteur qui résulte de l'encodeur est de taille beaucoup plus petite que le vecteur initial. Le décodeur correspond au processus inverse. Ainsi, l'encodeur compresse les données (de l'espace initial à l'espace encodé, aussi appelé espace latent) alors que le décodeur les décompresse.

En fonction de la distribution initiale des données, de l'espace latent et de la définition de l'encodeur, cette compression peut être avec perte, ce qui signifie qu'une partie de l'information est perdue pendant le processus d'encodage et ne peut pas être récupérée lors du décodage.

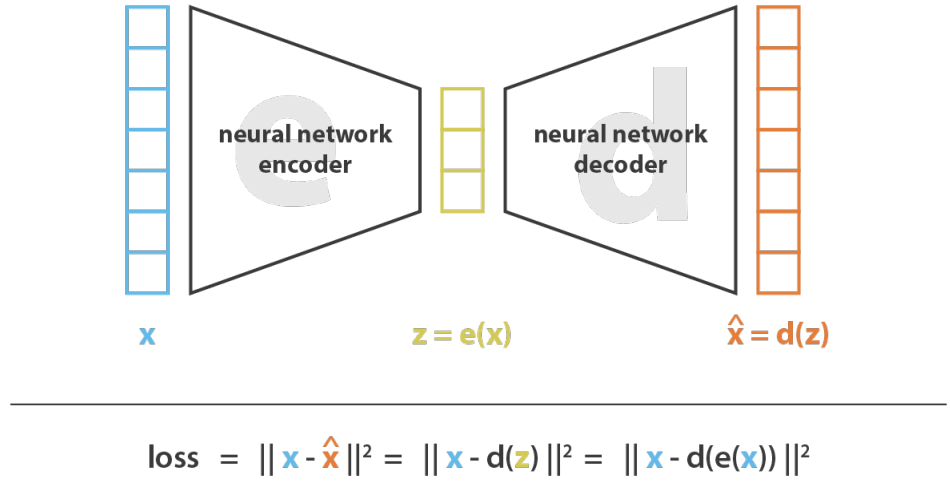

FIGURE 2 – Illustration d'un auto-encodeur (Source : TowardsDataScience [2])

Sur l'image **2**, on a

- x donnée initiale dans $\mathbb{R}^n$
- $e(x)$ donnée encodée dans l'espace latent $\mathbb{R}^d, \quad d << n$
- $d(e(x))$ donnée encodée décodée dans $\mathbb{R}^n$

Revenons à notre problème. Nous ne souhaitons pas juste reconstruire la donnée de départ, mais générer du nouveau contenu qui suivrait les mêmes lois. Lors des phases d'encodage-décodage, si l'espace latent est suffisamment régulier (bien "organisé" par l'encodeur pendant l'entraînement), on pourrait prendre un point aléatoire de cet espace latent et le décoder pour obtenir un nouveau contenu.

Cependant, la régularité de l'espace latent pour les auto-encodeurs dépend de la distribution des données dans l'espace initial, de la dimension de l'espace latent et de l'architecture de l'encodeur. Une solution possible pour obtenir cette régularité est d'introduire une régularisation explicite au cours du processus de formation, à l'aide de l'auto-encodeur variationnel.

2.2 Auto-encodeur variationnel (VAE)

Tout comme un auto-encodeur standard, un auto-encodeur variationnel est composé d'un encodeur et d'un décodeur. Cependant, afin d'introduire une certaine régularisation de l'espace latent, on procède à une légère modification du processus d'encodage-décodage : au lieu d'encoder une entrée comme un point unique, nous l'encodons comme une distribution sur l'espace latent. Le modèle est formé de cette manière :

1. l'entrée est encodée en tant que distribution sur l'espace latent
2. un point de l'espace latent est échantillonné à partir de cette distribution
3. le point échantillonné est décodé et l'erreur de reconstruction peut être calculée
4. l'erreur de reconstruction est propagée à travers le réseau

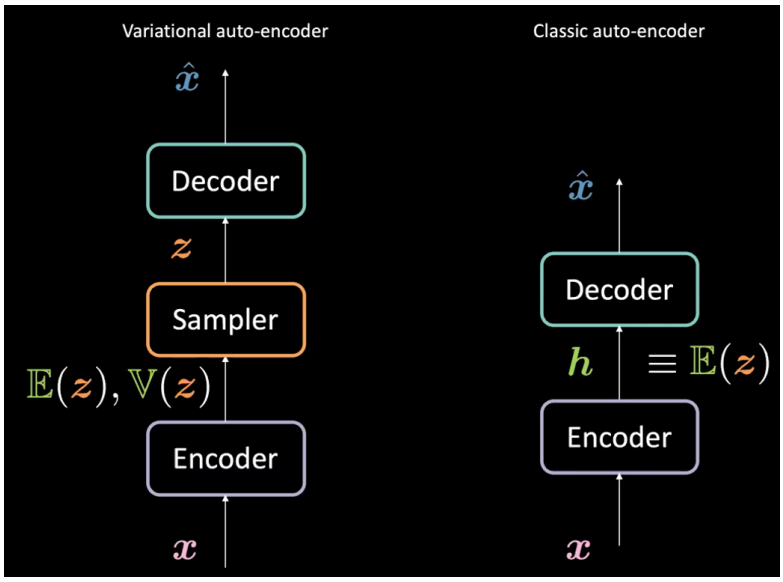

FIGURE 3 – Différence entre l'auto-encodeur classique et l'auto-encodeur variationnel (Source : Pang R., Klevs A., Chou H.-R., Jain M. [5])

On désigne par x la variable qui représente nos données et on suppose que x est généré à partir d'une variable latente z.

La loi $p(z) = N(0, I_d)$ est appelée distribution a priori.

On a alors la distribution d'échantillonnage, aussi appelé décodeur probabiliste :

$$p_\theta(x|z) = N(\mu_\theta^d(z), c_\theta^d(z) I_d)$$

avec $\mu_\theta^d(z)$ moyenne, $c_\theta^d(z)$ variance, $c_\theta^d(z) I_d$ matrice de covariance.

Enfin, la distribution a posteriori, aussi appelée encodeur probabiliste est donnée par la loi suivante :

$$p_\theta(z|x) = \frac{p_\theta(x|z)p(z)}{p_\theta(x)} = \frac{p_\theta(x|z)p(z)}{\int p_\theta(x|z')p(z')dz'} \qquad \text{(règle de Bayes)}$$

Ainsi, pour chaque point de données, une représentation latente z est échantillonnée à partir de la distribution a priori $p(z)$. Puis, la donnée x est échantillonnée à partir de la distribution d'échantillonnage $p_\theta(x|z)$.

2.3 Approximation de la distribution d'échantillonnage $p_\theta(x|z)$

En théorie, comme nous connaissons $p(z)$ et $p_\theta(x|z)$, nous pouvons calculer $p_\theta(z|x)$: il s'agit d'un problème d'inférence bayésien classique. Cependant, en général nous ne pourrons pas calculer l'intégrale du dénominateur, on utilise une technique d'approximation telle que l'inférence variationnelle. On définit alors $q_\phi(z|x)$ une approximation de $p_\theta(z|x)$. Il s'agit de la distribution qui minimise la "distance entre ces distributions de probabilité", que l'on exprime par la divergence de Kullback-Leibler :

$$D_{KL}\left(q_\phi(z|x) \,||\, p_\theta(z|x)\right)$$

Définition : Divergence de Kullback-Leibler

Soit P et Q deux distributions de probabilité. La divergence de Kullback-Leibler est définie par

- dans le cas discret :

$$D_{KL}(P \,||\, Q) = \sum_i P(i) \log\left(\frac{Q(i)}{P(i)}\right)$$

- dans le cas continue :

$$D_{KL}(P \,||\, Q) = \int_{-\infty}^{+\infty} p(x) \log\left(\frac{q(x)}{p(x)}\right) dx$$

On l'appelle aussi *l'entropie relative* entre les deux distributions.

Proposition 1

La divergence de Kullback-Leibler satisfait les propriétés suivantes :

- $D_{KL}(P \,||\, Q) \geq 0$
- $D_{KL}(P \,||\, Q) = 0 \quad \Rightarrow \quad P = Q$ presque sûrement

Ainsi, $q_\phi(z|x)$ est une distribution gaussienne dont la moyenne et la covariance sont définies par deux fonctions, μ_ϕ et c_ϕ, en fonction de x :

$$q_\phi(z|x) = N(\mu_\phi^n(x), c_\phi^n(x) I_d)$$

Ensuite, nous devons trouver θ permettant de maximiser la probabilité que les échantillons soient générés par $p_\theta(x)$. Ceci revient à maximiser la log-vraisemblance

$$\log(p_\theta(x)) = \log\left(\int p_\theta(x|z) p(z)\right)$$

Finalement, on souhaite maximiser l'expression suivante :

$$\log(p_\theta(x)) - D_{KL}\left(q_\phi(z|x) \,||\, p_\theta(z|x)\right)$$

2.4 Fonction de perte $Loss_{VAE}$

Nous venons de voir que la fonction de coût à maximiser est définie par

$$Loss_{VAE} = \log(p_\theta(x)) - D_{KL}\left(q_\phi(z|x) \,||\, p_\theta(z|x)\right)$$

En réécrivant la divergence de Kullback-Leibler on obtient

$$\begin{aligned}
D_{KL}\left(q_\phi(z|x) \,||\, p_\theta(z|x)\right) &= \int q_\phi(z|x) \log\left(\frac{q_\phi(z|x)}{p_\theta(z|x)}\right) \\
&= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(\frac{q_\phi(z|x)}{p_\theta(z|x)}\right)\right] \\
&= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(q_\phi(z|x)\right) - \log\left(p_\theta(z|x)\right)\right] \\
&= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(q_\phi(z|x)\right) - \log\left(\frac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)}\right)\right] \\
&= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(q_\phi(z|x)\right) - \log\left(p_\theta(x|z)\right) - \log\left(p_\theta(z)\right) + \log\left(p_\theta(x)\right)\right] \\
&= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(q_\phi(z|x)\right) - \log\left(p_\theta(x|z)\right) - \log\left(p_\theta(z)\right)\right] + \log\left(p_\theta(x)\right)
\end{aligned}$$

En soustrayant $\log(p_\theta(x))$ dans l'égalité on a alors

$$\begin{aligned}
\log(p_\theta(x)) - D_{KL}\left(q_\phi(z|x)\,||\,p_\theta(z|x)\right) &= -\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(q_\phi(z|x)\right) - \log\left(p_\theta(x|z)\right) - \log\left(p_\theta(z)\right)\right] \\
&= -\mathbb{E}_{z\sim q_\phi(z|x)}\left[-\log\left(p_\theta(x|z)\right) + \log\left(\frac{q_\phi(z|x)}{p_\theta(z)}\right)\right] \\
&= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(p_\theta(x|z)\right)\right] - \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(\frac{q_\phi(z|x)}{p_\theta(z)}\right)\right] \\
&= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(p_\theta(x|z)\right)\right] - \int q_\phi(z|x)\log\left(\frac{q_\phi(z|x)}{p_\theta(z)}\right) \\
&= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log(p_\theta(x|z))\right] - D_{KL}\left(q_\phi(z|x)\,||\,p(z)\right)
\end{aligned}$$

Ainsi, résoudre le problème de maximisation

$$\phi^* = \underset{\phi}{\operatorname{argmax}}\left(\log(p_\theta(x)) - D_{KL}\left(q_\phi(z|x)\,||\,p_\theta(z|x)\right)\right)$$

est équivalent à résoudre le problème de minimisation

$$\phi^*, \theta^* = \underset{\phi,\theta}{\operatorname{argmin}} -\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log(p_\theta(x|z))\right] + D_{KL}\left(q_\phi(z|x)\,||\,p(z)\right)$$

On pourrait aussi réécrire le problème de cette manière :

$$\phi^*, \theta^* = \underset{\phi,\theta}{\operatorname{argmin}}\, \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(\frac{q_\phi(z|x)}{p_\theta(x,z)}\right)\right]$$

car

$$\begin{aligned}
-\mathbb{E}_{z\sim q_\phi(z|x)}\left[-\log\left(p_\theta(x|z)\right) + \log\left(\frac{q_\phi(z|x)}{p_\theta(z)}\right)\right] &= -\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(\frac{q_\phi(z|x)}{p_\theta(x|z)p_\theta(z)}\right)\right] \\
&= -\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(\frac{q_\phi(z|x)}{p_\theta(x,z)}\right)\right]
\end{aligned}$$

Dans l'expression de la fonction de perte

$$Loss_{VAE} = -\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log(p_\theta(x|z)\right] + D_{KL}\left(q_\phi(z|x)\,||\,p(z)\right)$$

le terme contenant la divergence de Kullback-Leibler est appelé "terme de régularisation" (sur la couche latente), l'autre se nomme "terme de reconstruction" (sur la couche finale).

2.5 Expression des termes de $Loss_{VAE}$ pour l'implémentation

Réécrivons les termes de $Loss_{VAE}$ afin de les utiliser dans l'implémentation des réseaux de neurones.

Proposition 2

En posant les distributions $p_1 = \mathcal{N}(\mu_1, \Sigma_1)$ et $p_2 = \mathcal{N}(\mu_2, \Sigma_2)$, on a

$$D_{KL}[p_1\,||\,p_2] = \frac{1}{2}\left[\log\frac{|\Sigma_2|}{|\Sigma_1|} - n + \operatorname{tr}\{\Sigma_2^{-1}\Sigma_1\} + (\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1)\right]$$

Prenons $p_1 = q_\phi(z|x) = \mathcal{N}(\mu^n_\phi(x), c^n_\phi(x)I_n)$ et $p_2 = p(z) = \mathcal{N}(0, I_d)$.
On peut alors réécrire le terme de régularisation

$$
\begin{aligned}
D_{KL}[q_\phi(z|x) \,||\, p(z)] &= \frac{1}{2}\left[\log\frac{|\Sigma_2|}{|\Sigma_1|} - n + \text{tr}\{\Sigma_2^{-1}\Sigma_1\} + (\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1)\right] \\
&= \frac{1}{2}\left[\log\frac{|I_d|}{|c^n_\phi(x)I_n|} - n + \text{tr}\{I_d^{-1}c^n_\phi(x)I_n\} + (\vec{0}-\mu^n_\phi(x))^T I_d^{-1}(\vec{0}-\mu^n_\phi(x))\right] \\
&= \frac{1}{2}\left[-\log|c^n_\phi(x)I_n| - n + \text{tr}\{c^n_\phi(x)I_n\} + \mu^n_\phi(x)^T\mu^n_\phi(x)\right] \\
&= \frac{1}{2}\left[-\log\prod_i |c^n_\phi(x)_i| - n + \sum_i c^n_\phi(x)_i + \sum_i \mu^n_\phi(x)_i^2\right] \\
&= \frac{1}{2}\left[-\sum_i \log|c^n_\phi(x)_i| - n + \sum_i c^n_\phi(x)_i + \sum_i \mu^n_\phi(x)_i^2\right] \\
&= \frac{1}{2}\left[-\sum_i \big(\log|c^n_\phi(x)_i| + 1\big) + \sum_i c^n_\phi(x)_i + \sum_i \mu^n_\phi(x)_i^2\right] \\
&= \frac{1}{2}\sum_i \left[-\log|c^n_\phi(x)_i| - 1 + c^n_\phi(x)_i + \mu^n_\phi(x)_i^2\right]
\end{aligned}
$$

avec n la dimension de l'espace latent (réduit).

```
def kl(mean, logvar):
    kl = 0.5 * tf.reduce_sum(- logvar - 1. + tf.exp(logvar) + tf.square(mean), axis = 1)
    return kl
```

Calculons le logarithme de $p_\theta(x|z) = N(\mu^d_\theta(z), c^d_\theta(z)I_d)$.

$$
\begin{aligned}
\log\left(p_\theta(x|z)\right) &= \log\left[\frac{1}{\sqrt{|2\pi c_\theta^d(z) I_d|}}\exp\left(-\frac{1}{2c_\theta^d(z)}\left(x-\mu_\theta^d(z)\right)^T\left(x-\mu_\theta^d(z)\right)\right)\right] \\
&= \log\left(\frac{1}{\sqrt{|2\pi c_\theta^d(z) I_d|}}\right) - \frac{1}{2c_\theta^d(z)}\left(x-\mu_\theta^d(z)\right)^T\left(x-\mu_\theta^d(z)\right) \\
&= -\log\left(\sqrt{|2\pi c_\theta^d(z) I_d|}\right) - \frac{1}{2c_\theta^d(z)}\left(x-\mu_\theta^d(z)\right)^T\left(x-\mu_\theta^d(z)\right) \\
&= -\frac{1}{2}\log\left(|2\pi c_\theta^d(z) I_d|\right) - \frac{1}{2c_\theta^d(z)}\left(x-\mu_\theta^d(z)\right)^T\left(x-\mu_\theta^d(z)\right) \\
&= -\frac{1}{2}\left[\log(|2\pi c_\theta^d(z) I_d|) + \frac{1}{c_\theta^d(z)}\left(x-\mu_\theta^d(z)\right)^T\left(x-\mu_\theta^d(z)\right)\right] \\
&= -\frac{1}{2}\left[\log\left(\prod_i |c_\theta^d(z)_i|2\pi\right) + \sum_i \frac{\left(x-\mu_\theta^d(z)\right)_i^2}{c_\theta^d(z)_i}\right] \\
&= -\frac{1}{2}\left[\sum_i \log\left(|c_\theta^d(z)_i|2\pi\right) + \sum_i \frac{\left(x-\mu_\theta^d(z)\right)_i^2}{c_\theta^d(z)_i}\right] \\
&= -\frac{1}{2}\left[n\log(2\pi) + \sum_i \log\left(|c_\theta^d(z)_i|\right) + \sum_i \frac{\left(x-\mu_\theta^d(z)\right)_i^2}{c_\theta^d(z)_i}\right] \\
&= -\frac{n\log(2\pi)}{2} - \frac{1}{2}\sum_i\left[\log\left(|c_\theta^d(z)_i|\right) + \frac{\left(x-\mu_\theta^d(z)\right)_i^2}{c_\theta^d(z)_i}\right] \\
&\approx -\frac{1}{2}\sum_i\left[\log\left(|c_\theta^d(z)_i|\right) + \frac{\left(x-\mu_\theta^d(z)\right)_i^2}{c_\theta^d(z)_i}\right]
\end{aligned}
$$

La dernière étape vient du fait que le premier terme est constant, on se permet alors de le négliger.

Ainsi,

$$
\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log(p_\theta(x|z)\right] \approx \mathbb{E}_{z\sim q_\phi(z|x)}\left[-\frac{1}{2}\sum_i\left(\log\left(|c_\theta^d(z)_i|\right) + \frac{\left(x-\mu_\theta^d(z)\right)_i^2}{c_\theta^d(z)_i}\right)\right]
$$

Cependant, après de nombreuses recherches et la lecture de nombreux articles sur le sujet, nous avons constaté que l'auto-encodeur variationnel ne permettait pas de générer la moyenne ET la variance. Le terme de reconstruction était généralement calculé en utilisant l'erreur quadratique moyenne entre la donnée cible et la moyenne générée. Nous avons donc fixé une valeur constante pour la variance, ce qui donne le terme de reconstruction suivant :

```
def reconstruction(targets, mean, logvar):
    se = -0.5 * tf.reduce_sum(tf.square(targets - mean) / tf.exp(logvar))
    return se
```

Finalement, la loss sera définie par

```
total_loss = -reconstruction_loss + beta * kl_loss
```

avec β fixé à 2. Il s'agit d'un hyperparamètre qui permet de découvrir des facteurs latents démêlés. Lorsque $\beta > 1$, on applique une contrainte plus forte sur la couche latente et on limite la capacité de représentation de la variable latente z. Mais il ne faut pas choisir une trop grande valeur pour éviter de compromettre la reconstruction sur la couche finale. [6]

2.6 Calculs des gradients de la fonction de perte

L'étape suivante et d'effectuer un apprentissage par des méthodes de gradient stochastique avec de la rétro-propagation. On souhaite alors calculer les gradients par rapports aux paramètres θ et ϕ.

$$\begin{aligned}\nabla_\theta Loss_{VAE} &= \nabla_\theta \left(-\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log(p_\theta(x|z)\right] + D_{KL}\left(q_\phi(z|x)\,||\,p(z)\right)\right)\\ &= -\mathbb{E}_{z\sim q_\phi(z|x)}\left[\nabla_\theta \log(p_\theta(x|z))\right] + \nabla_\theta(D_{KL})(q_\phi(z|x)\,||\,p(z))\\ &= -\mathbb{E}_{z\sim q_\phi(z|x)}\left[\nabla_\theta \log(p_\theta(x|z))\right]\end{aligned}$$

La méthode de Monte-Carlo nous permet d'obtenir

$$\nabla_\theta Loss_{VAE} \approx \sum_{i=1}^{N} \nabla_\theta \log(p_\theta(x_i|z_i))$$

avec $z_i \sim q_\phi(z_i|x_i)$

Calculons le gradient de la fonction coût par rapport à ϕ.

$$\nabla_\phi Loss_{VAE} = \nabla_\phi \left(-\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log(p_\theta(x|z)\right] + D_{KL}\left(q_\phi(z|x)\,||\,p(z)\right)\right)$$

z étant de distribution dépendant de ϕ, nous savons que

$$\nabla_\phi \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log(p_\theta(x|z)\right] \neq \mathbb{E}_{z\sim q_\phi(z|x)}\left[\nabla_\phi \log(p_\theta(x|z))\right]$$

On utilise alors l'astuce de reparamétrisation.

2.7 Reparamétrisation de z

Cette technique est utilisée pour rendre possible la descente en gradient. Cela consiste à utiliser le fait que z est de distribution gaussienne, alors on réécrit z sous la forme

$$z = T_\phi(x, \epsilon)$$

avec ϵ une variable aléatoire.

Ceci nous permet d'obtenir un gradient déterministe, et donc calculable, puisque

$$\mathbb{E}_{z\sim q_\phi(z|x)}\left[f_\phi(z)\right] = \mathbb{E}_{\epsilon\sim p}\left[f_\phi(T_\phi(x,\epsilon))\right]$$

avec $p \equiv N(0, I_d)$.

Pour déterminer la fonction T_ϕ, il faut se rappeler que si z suit une loi $N(\mu_\phi^n(x), c_\phi^n(x) I_d)$, alors on a

$$z = c_\phi^n(x) \odot \epsilon + \mu_\phi^n(x), \qquad \epsilon \sim N(0, I_d)$$

avec $\odot$ le produit élément par élément.

Finalement, l'image **4** illustre le fonctionnement de l'auto-encodeur variationnel.

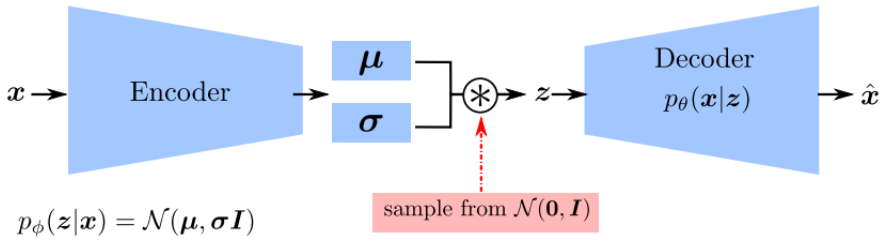

FIGURE 4 – Représentation de l'auto-encodeur variationnel (Source : Clément Rambour [3])

3 Implémentation de l'auto-encodeur variationnel

Détaillons la partie pratique du projet : l'implémentation de l'auto-encodeur variationnel, son entraînement ainsi que les tests permettant de le valider.

3.1 Réseau de neurones de l'encodeur

L'encodeur utilisé est un réseau neuronal convolutif. Son architecture est formée par un empilement de couches de traitement, dont le but est de pré-traiter de petites quantités d'informations. L'objectif est d'obtenir des données de sorties de dimension réduite (ici $d = 10$).

```
Model: "encoder"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_1 (InputLayer)         [(None, 400)]             0
_________________________________________________________________________________________________
reshape (Reshape)            (None, 400, 1)            0           input_1[0][0]
_________________________________________________________________________________________________
conv1d (Conv1D)              (None, 398, 16)           64          reshape[0][0]
_________________________________________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 199, 16)           0           conv1d[0][0]
_________________________________________________________________________________________________
conv1d_1 (Conv1D)            (None, 197, 32)           1568        max_pooling1d[0][0]
_________________________________________________________________________________________________
max_pooling1d_1 (MaxPooling1D) (None, 98, 32)          0           conv1d_1[0][0]
_________________________________________________________________________________________________
conv1d_2 (Conv1D)            (None, 96, 64)            6208        max_pooling1d_1[0][0]
_________________________________________________________________________________________________
max_pooling1d_2 (MaxPooling1D) (None, 48, 64)          0           conv1d_2[0][0]
_________________________________________________________________________________________________
flatten (Flatten)            (None, 3072)              0           max_pooling1d_2[0][0]
_________________________________________________________________________________________________
dense (Dense)                (None, 64)                196672      flatten[0][0]
_________________________________________________________________________________________________
z_mean (Dense)               (None, 10)                650         dense[0][0]
_________________________________________________________________________________________________
z_log_var (Dense)            (None, 10)                650         dense[0][0]
_________________________________________________________________________________________________
sampling_encod (Sampling_encod) (None, 10)             0           z_mean[0][0]
                                                                   z_log_var[0][0]
=================================================================================================
Total params: 205,812
Trainable params: 205,812
Non-trainable params: 0
_________________________________________________________________________________________________
```

FIGURE 5 – Structure de l'encodeur implémenté

Sur la figure **5** on peut voir les différentes couches qui forment le réseau.

La couche de convolution *Conv1D* traite les données d'un réseau convolutif temporel.

- L'argument *filters* fait référence au nombre de filtres à appliquer dans la convolution. Cela affecte la dimension de sortie.
- *kernel_size* correspond à la longueur de la fenêtre de convolution. [7]
- L'argument *strides* fait référence à la longueur de foulée de la convolution. On conserve la valeur par défaut 1.
- *padding* correspond à la façon dont le remplissage doit être effectué sur la sortie de la convolution. On ne veut pas de remplissage (cela est plutôt intéressant dans le cas du traitement d'images) alors on choisit *valid*.
- L'argument *activation* correspond à la fonction d'activation. Les neurones sont interconnectés aux différentes couches alimentées par celle-ci. Elle active le neurone lorsque la sortie atteint la valeur seuil définie de la fonction. Ici on choisit *relu* (unité linéaire rectifiée) qui produira l'entrée si elle est positive, sinon elle produira zéro ($f(x) = \max\{0, x\}$).

La couche *MaxPooling1D* permet de compresser l'information en réduisant la taille des tableaux de données intermédiaires. Le tableau en entrée est découpé en une série de tableaux et le max pooling effectue des opérations de regroupement maximum. L'argument *pool_size* de la fonction correspond à la taille des fenêtres de regroupement. [7]

Reshape est utilisé pour remodeler la dimension de l'entrée de la couche. Par exemple, on pourrait changer la dimension de l'entrée *(batch_size, 2, 10)* en *(batch_size, 4, 5)*.

La couche *Flatten* est utilisée pour aplatir l'entrée d'une couche. Par exemple, on pourrait vouloir passer de l'entrée *(batch_size, 4, 5)* à *(batch_size, 20)*.

Dense est la couche régulière du réseau neuronal. C'est la couche la plus fréquemment utilisée. Elle effectue l'opération `output = activation(dot(input, kernel) + bias)` avec

- *input* représentant les données d'entrée
- *kernel* correspondant aux données de poids
- *dot* représentant le produit scalaire de toutes les entrées et des poids correspondants
- *bias* correspondant à une valeur biaisée utilisée dans pour optimiser le modèle [7]

La sortie de l'encodeur est constituée de la moyenne μ_z de la variable latente z, sa log-variance $logvar_z$ et de z l'échantillon aléatoire reparamétrisé tel que $z = \mu_z + \exp(0.5 * logvar_z) * \varepsilon$ avec $\varepsilon \sim N(0, I_d)$.

3.2 Réseau de neurones du décodeur

Le décodeur est également un réseau neuronal convolutif et son principe est le même que pour l'encodeur, mais il s'agit cette fois d'obtenir une sortie de grande dimension (ici $n = 400$).

```
Model: "decoder"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_2 (InputLayer)         [(None, 10)]              0
_________________________________________________________________________________________________
dense_1 (Dense)              (None, 3456)              38016       input_2[0][0]
_________________________________________________________________________________________________
reshape_1 (Reshape)          (None, 54, 64)            0           dense_1[0][0]
_________________________________________________________________________________________________
conv1d_3 (Conv1D)            (None, 52, 64)            12352       reshape_1[0][0]
_________________________________________________________________________________________________
up_sampling1d (UpSampling1D) (None, 104, 64)           0           conv1d_3[0][0]
_________________________________________________________________________________________________
conv1d_4 (Conv1D)            (None, 102, 32)           6176        up_sampling1d[0][0]
_________________________________________________________________________________________________
up_sampling1d_1 (UpSampling1D) (None, 204, 32)         0           conv1d_4[0][0]
_________________________________________________________________________________________________
conv1d_5 (Conv1D)            (None, 202, 16)           1552        up_sampling1d_1[0][0]
_________________________________________________________________________________________________
up_sampling1d_2 (UpSampling1D) (None, 404, 16)         0           conv1d_5[0][0]
_________________________________________________________________________________________________
conv1d_6 (Conv1D)            (None, 402, 1)            49          up_sampling1d_2[0][0]
_________________________________________________________________________________________________
flatten_1 (Flatten)          (None, 402)               0           conv1d_6[0][0]
_________________________________________________________________________________________________
x_mean (Dense)               (None, 400)               161200      flatten_1[0][0]
_________________________________________________________________________________________________
tf.ones_like (TFOpLambda)    (None, 400)               0           x_mean[0][0]
_________________________________________________________________________________________________
tf.math.multiply (TFOpLambda) (None, 400)              0           tf.ones_like[0][0]
_________________________________________________________________________________________________
sampling_decod (Sampling_decod) (None, 400)            0           x_mean[0][0]
                                                                   tf.math.multiply[0][0]
=================================================================================================
Total params: 219,345
Trainable params: 219,345
Non-trainable params: 0
_________________________________________________________________________________________________
```

FIGURE 6 – Structure de l'encodeur implémenté

La figure **6** décrit la structure du décodeur. La *UpSampling1D* permet de faire du sur-échantillonnage pour les entrées 1D. Ainsi, les fenêtres seront répétées *size* fois le long de l'axe du temps.

La sortie du décodeur est constituée de la moyenne $\mu_{\hat{x}}$ de la variable $\hat{x}$, sa variance $var_{\hat{x}}$ (ou log-variance) fixée (ici 0.1) et de z l'échantillon aléatoire reparamétrisé tel que $z = \mu_{\hat{x}} + \sqrt{var_{\hat{x}}} * \varepsilon$ avec $\varepsilon \sim N(0, I_n)$.

3.3 Entraînement de l'auto-encodeur variationnel

L'entraînement de l'encodeur et du décodeur s'est fait sur un gros jeu de données. Les données correspondent aux solutions de l'équation de Burgers (voir **1.2**) obtenues par un schéma de volumes finis décentré en espace, implicite en temps pour le terme de diffusion et explicite en temps pour le terme de transport. décentrées en temps et centrées en espace.

Rappelons le schéma

$$\frac{\rho^{n+1} - \rho^n}{\Delta t} + \frac{F_{j+\frac{1}{2}}(\rho_j^n, \rho_{j+1}^n) - F_{j-\frac{1}{2}}(\rho_{j-1}^n, \rho_j^n)}{\Delta x} = \frac{1}{R_e} \frac{\rho_{j+1}^{n+1} - 2\rho_j^{n+1} + \rho_{j-1}^{n+1}}{\Delta x^2}$$

avec

$$F_{j+\frac{1}{2}}(\rho_j, \rho_{j+1}) = \frac{1}{2}(\rho_j + \rho_{j+1}) + \frac{c}{2}(\rho_j - \rho_{j+1})$$

et $c = max(\mid \rho_j \mid, \mid \rho_{j+1} \mid)$

Ci-dessous sont décrits les deux jeux de données utilisés. Rappelons que a suit une loi $\mathcal{U}[0.1, 0.3]$ et que $logR_e$ suit une loi $\mathcal{U}[4, 13]$.

- 10000 réalisations de Burgers avec viscosité R_e stochastique et condition initiale $\rho(0, x) = \sin(2\pi x)$. Les réalisations ont des temps finaux T répartis entre 0.0001 et 0.9999.
- 5825 réalisations de Burgers avec viscosité R_e stochastique et condition initiale stochastique $\rho(0, x) = \begin{cases} 1.0, & \text{x <0.5} \\ a, & \text{x > 0.5} \end{cases}$. On a fait varier le temps final T entre 0.0001 et 0.5824.

Les générations aléatoires de $\rho(T, x)$ sont faites à partir de la méthode *solve(T)* de la classe *Burgers*. Celle-ci est accessible dans le notebook *burgers.ipynb* qui m'a été fourni, stocké dans le dossier *Documents*. Les données générées sont stockées dans des fichiers *.dat* puis on crée des fichiers binaires Python *.npy* car ils sont moins volumineux et font gagner du temps de lecture lors des exécutions.

Dans le cas de la condition initiale non aléatoire, le fichier d'entraînement se nomme *final_sinus.npy*. Il est disponible dans le même dossier que les notebook : *Documents*.

Pour la condition initiale $\rho(0, x) = \sin(2\pi x)$, le fichier d'entraînement se nomme *final_creneau.npy*. Il est également disponible dans le dossier *Documents*.

Entraînement avec condition initiale non aléatoire

L'entraînement de l'auto-encodeur avec la condition initiale $\rho(0, x) = \sin(2\pi x)$ a été réalisé en 300 *epochs* et *batch_size* = 50.

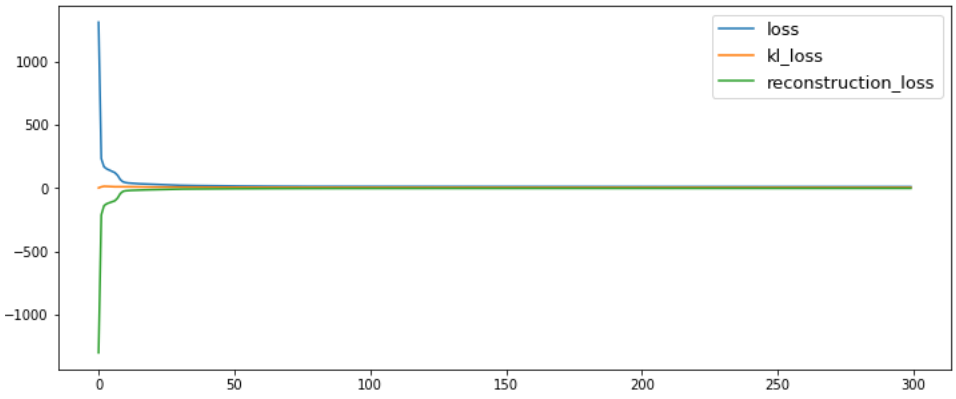

FIGURE 7 – Evolution de la fonction de perte au cours de l'entraînement

On constate sur la figure **7** que la fonction de perte a fortement diminué dès les premières *epochs*.

Entraînement avec condition initiale aléatoire

Le jeu de données pour Burgers stochastique est moins conséquent donc on a rallongé l'entraînement pour obtenir une diminution de la *loss* satisfaisante : 800 *epochs* et *batch_size* = 50.

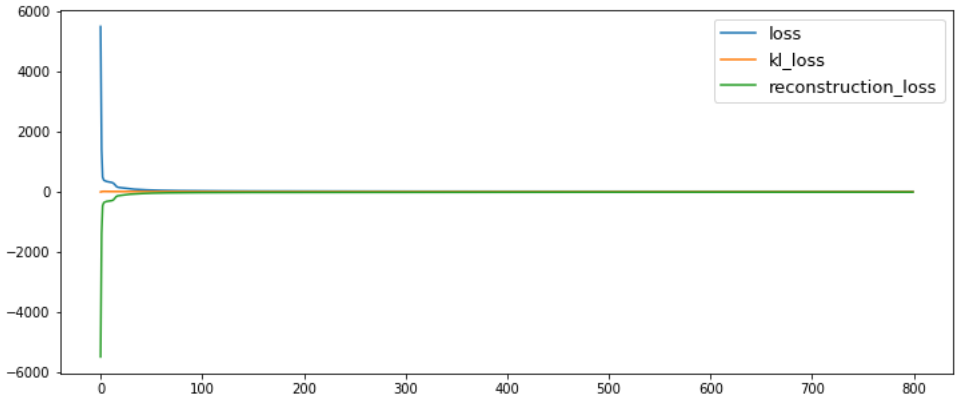

FIGURE 8 – Evolution de la fonction de perte au cours de l'entraînement

La figure **8** montre une également une forte diminution de la fonction de perte dès les premières *epochs*.

4 Test de l'auto-encodeur variationnel

4.1 Condition initiale non aléatoire

Reproduction de la loi de ρ par le VAE, à partir des solutions de Burgers tout temps confondus

Dans cette première étude, on souhaite voir comment se comportent les lois générées par l'auto-encodeur variationnel (VAE) et les comparer avec celles données en entrée.

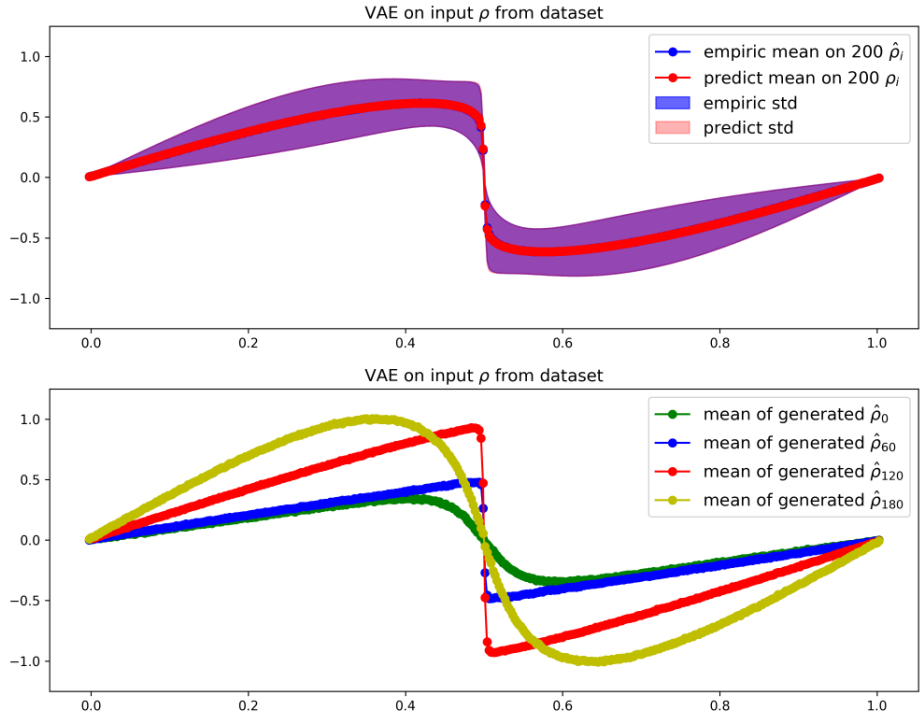

FIGURE 9 – Application du VAE sur le jeu de données initial

La première partie de la figure **9** montre en rouge et rose respectivement la moyenne et l'écart-type pour 200 solutions de Burgers. La courbe bleue correspond à la moyenne sur 200 données générées par le VAE. La zone bleue montre l'écart-type associé à ces données générées. On remarque que la moyenne pour les données générées est identique à la moyenne prévue, sur l'ensemble de l'intervalle. De plus, les zones associées à l'écart-type sont parfaitement superposées.

Les quatre moyennes générées dans la deuxième partie de la figure **9** montrent que l'auto-encodeur génère bien des lois pour différents temps finaux.

La même étude a été effectuée sur un jeu de données plus grand, c'est-à-dire de taille 1000. Les constats étaient environ similaires à ceux faits sur la figure **9**.

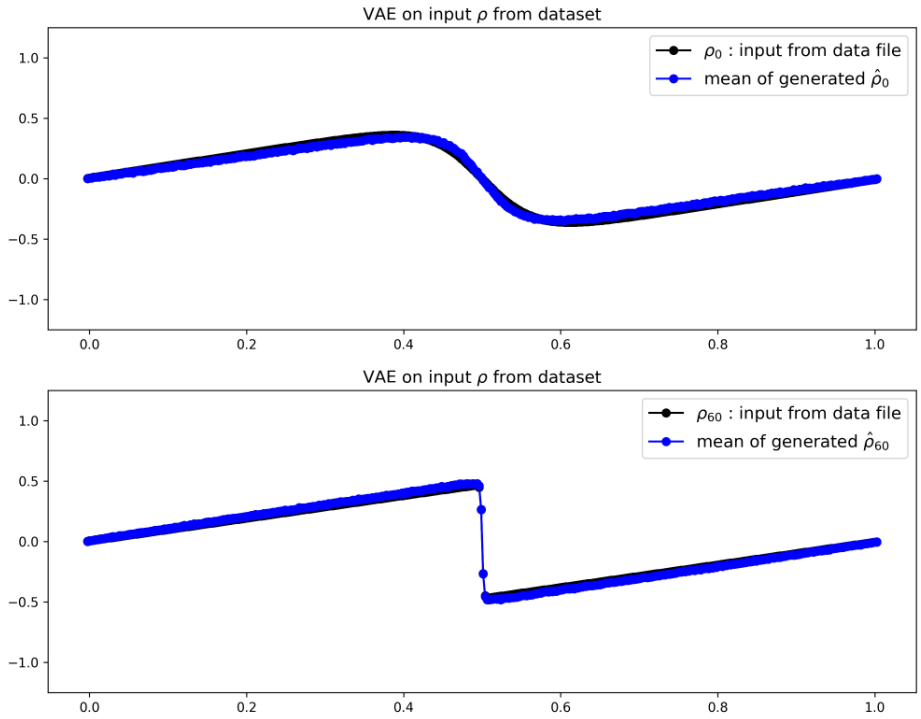

FIGURE 10 – Application du VAE sur le jeu de données initial

On constate sur la figure **10** que les moyennes générées (en bleu) se superposent bien sur les moyennes des données d'entrée associées (en noir).

Génération de la loi de ρ par le décodeur, à partir de données réduites de lois normales multivariées

Dans ce test, il s'agissait d'observer les lois générées par le décodeur, pour des données d'entrée de dimension réduite ($d = 10$). On souhaite vérifier que les sorties ne sont pas toutes identiques. On génère alors 200 réalisations de distribution normale multivariée de dimension 10 et on décode chacune d'entre elles. Enfin, il faut aussi comparer la moyenne et l'écart-type obtenus sur les données générées avec ceux observés pour 200 réalisations de Burgers. L'objectif est d'obtenir la distribution la plus ressemblante possible.

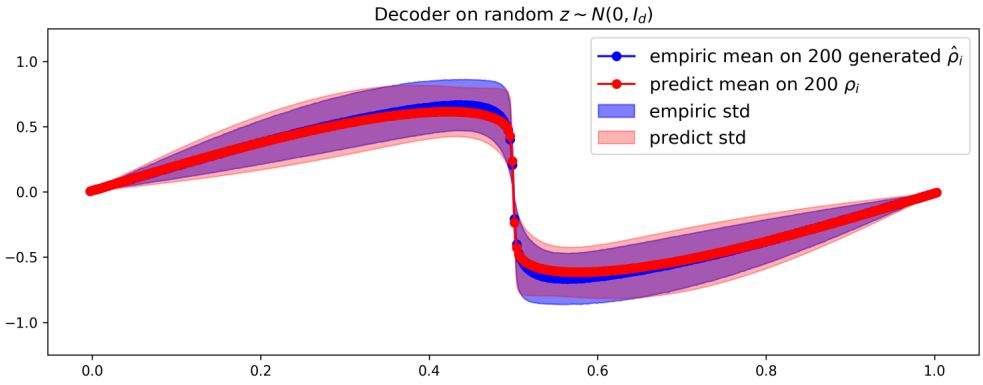

FIGURE 11 – Application du décodeur sur des lois normales multivariées

La figure **11** représente en rouge et rose respectivement la moyenne et l'écart-type pour 200 solutions de Burgers (temps finaux aléatoires). La courbe bleue correspond à la moyenne sur 200 données générées par le décodeur. Comme précisé au-dessus, les entrées du décodeur sont des réalisations de distribution normale multivariée. La zone d'incertitude en bleu est représentée par l'écart-type sur ces données générées.

On constate que la moyenne pour le lot de données générées est très proche de celle prévue. L'écart-type est légèrement différent par rapport à l'écart-type observé avec la résolution de Burgers, mais les résultats sont tout de même satisfaisants.

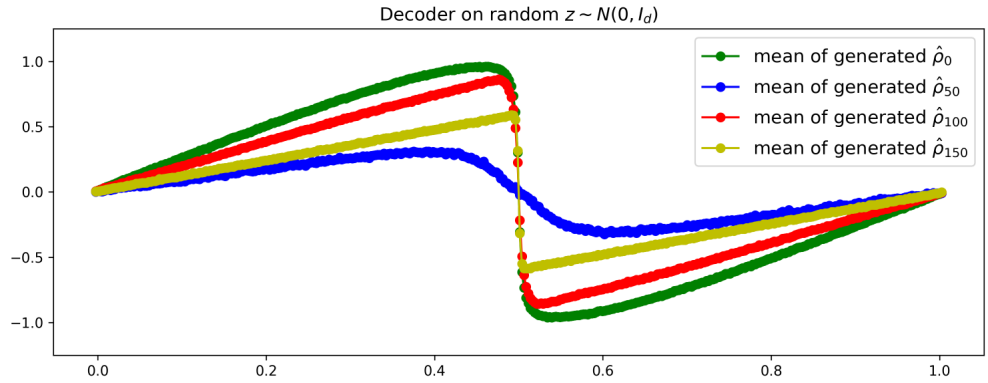

FIGURE 12 – Application du décodeur sur des lois normales multivariées

On remarque également sur la figure **12** que les moyennes des lois générées sont bien différentes les unes par rapport aux autres, on obtient donc des ρ pour différents temps finaux.

Reproduction de la loi de ρ par le VAE, à partir des solutions de Burgers au temps 0.4

Dans cette partie, il s'agissait de considérer uniquement des solutions de Burgers avec condition initiale non aléatoire et un temps final fixé à 0.4. Les données sont stockées dans le fichier *0.4_final_sinus.npy* (dossier *Documents*).

Ainsi, on entre dans l'encodeur 200 générations aléatoires puis on décode les données réduites. L'objectif était de vérifier que la moyenne et l'écart-type sur 200 données générées soient similaires à ceux obtenus en exécutant 200 réalisations de Burgers.

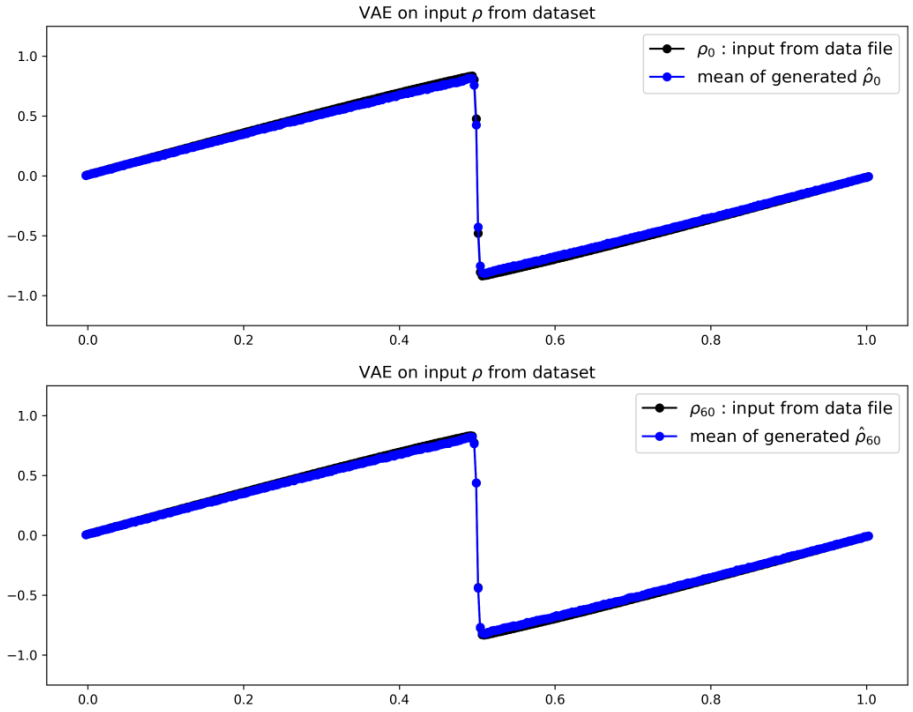

FIGURE 13 – Application du VAE sur le jeu de données de temps final 0.4

Tout d'abord, on remarque sur la figure **13** que les moyennes des données générées (en bleu) se superposent sur les moyennes des données d'entrée (en noir).

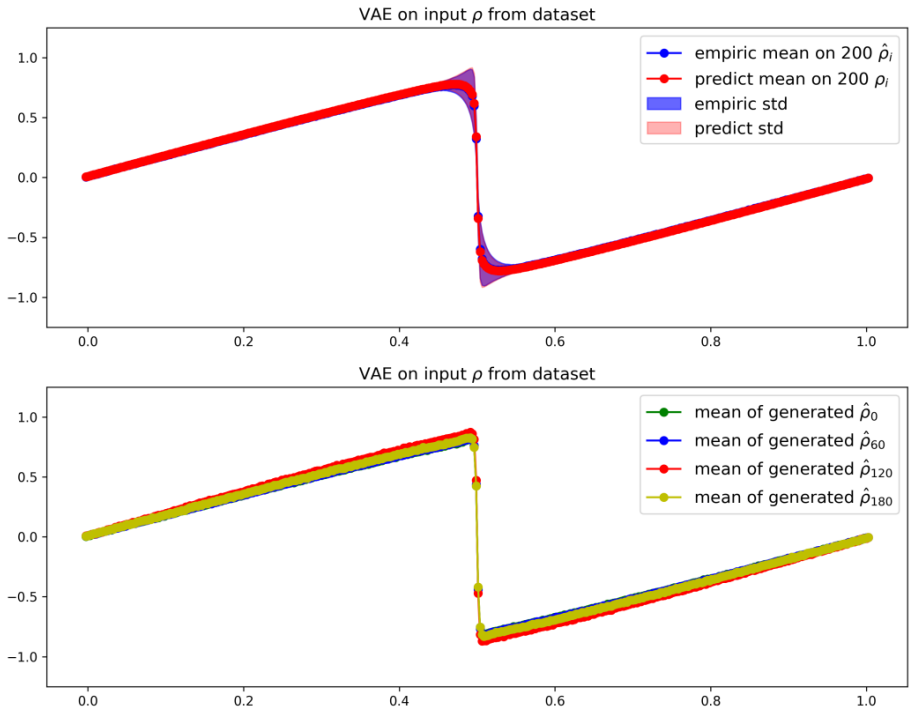

FIGURE 14 – Moyenne et écart-type calculés après encodage et décodage sur 200 réalisations de Burgers

Les premières courbes affichées sur la figure **14** représentent en rouge et rose respectivement la moyenne et l'écart-type pour 200 solutions de Burgers au temps 0.4. La courbe bleue correspond à la moyenne sur 200 données générées par l'auto-encodeur ; les données d'entrée sont les solutions de Burgers. La zone bleue montre l'écart-type associé à ces données générées.

On constate que l'écart-type sur la sortie du VAE prend des grandes valeurs aux mêmes endroits que l'écart-type obtenu par la résolution de Burgers. De plus, les deux courbes de la moyenne sont identiques. Ceci montre que l'auto-encodeur est bien entraîné et efficace pour générer ρ en un temps fixé.

En observant la deuxième partie de la figure **14**, on remarque que les données représentent le même temps, elles sont identiques.

4.2 Condition initiale dans le cas stochastique

Reproduction de la loi de ρ par le VAE, à partir des solutions de Burgers tout temps confondus

On effectue les mêmes études que précédemment mais l'entraînement du VAE se fait dans le cas où les données sont solutions de Burgers pour la condition initiale $\rho(t=0,x) = \begin{cases} 1.0, & \text{x <0.5} \\ a, & \text{x > 0.5} \end{cases}$.

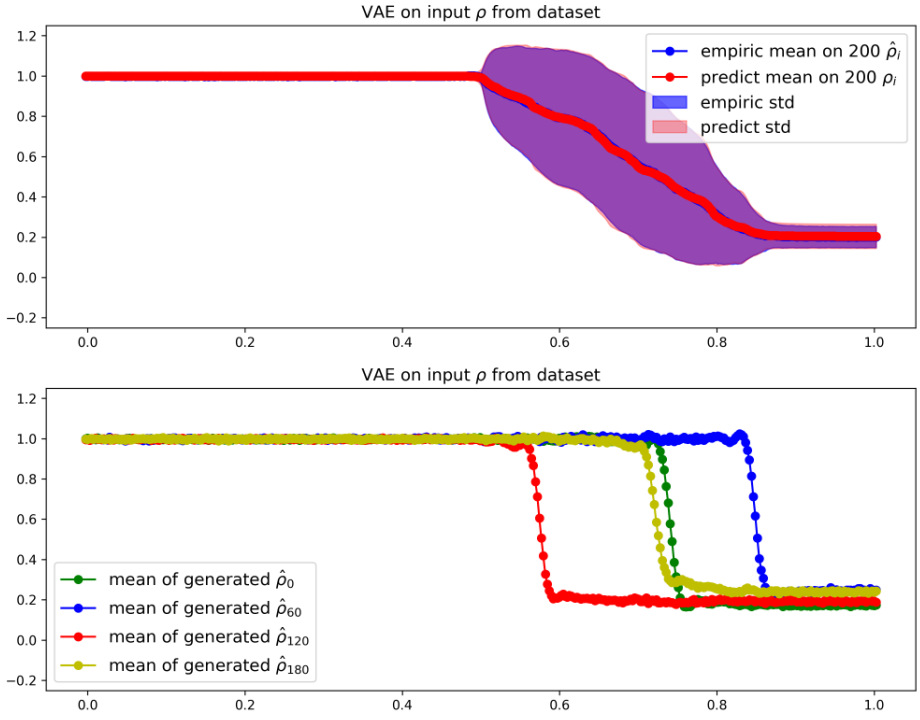

FIGURE 15 – Application du VAE sur le jeu de données initial

La première partie de la figure **15** montre en rouge et rose respectivement la moyenne et l'écart-type pour 200 solutions de Burgers. La courbe bleue correspond à la moyenne sur 200 données générées par le VAE. La zone bleue montre l'écart-type associé à ces données générées. On remarque que la moyenne pour les données générées est identique à la moyenne prévue, sur l'ensemble de l'intervalle. De plus, les zones associées à l'écart-type sont parfaitement superposées.

Les quatre moyennes générées dans la deuxième partie de la figure **15** montrent que l'auto-encodeur génère bien des lois pour différents temps finaux.

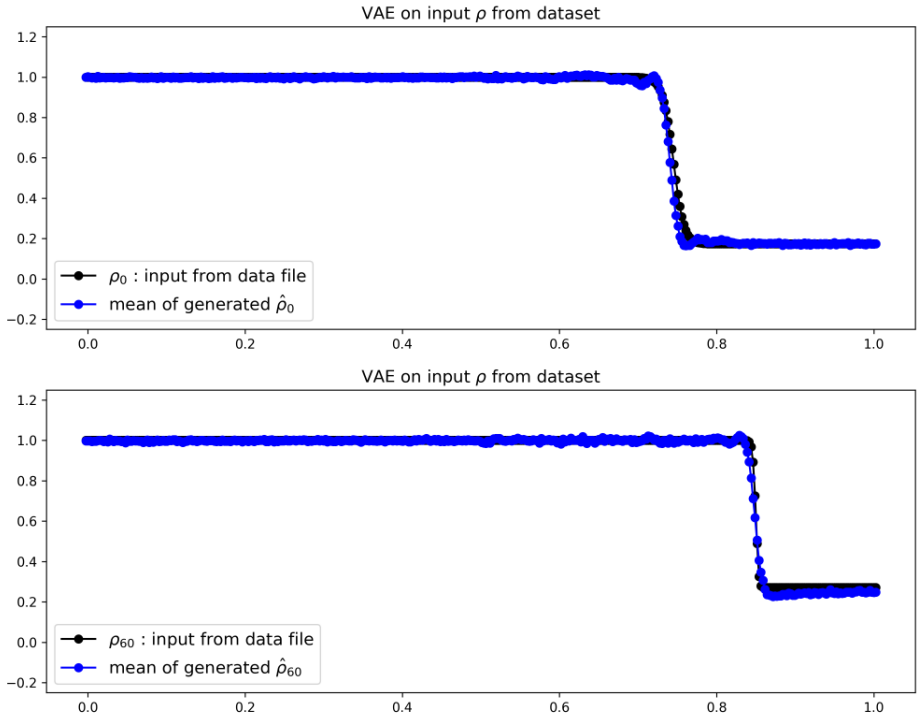

FIGURE 16 – Application du VAE sur le jeu de données initial

On constate sur la figure **16** que les moyennes générées (en bleu) sont proches des moyennes des données d'entrée associées (en noire). En revanche, la précision est moins bonne que dans l'étude précédente.

Génération de la loi de ρ par le décodeur, à partir de données réduites de lois normales multivariées

Dans ce test, on génère 200 réalisations de distribution normale multivariée de dimension 10 et on décode chacune d'entre elles. On souhaite comparer la moyenne et l'écart-type obtenus avec les résultats de 200 réalisations de Burgers.

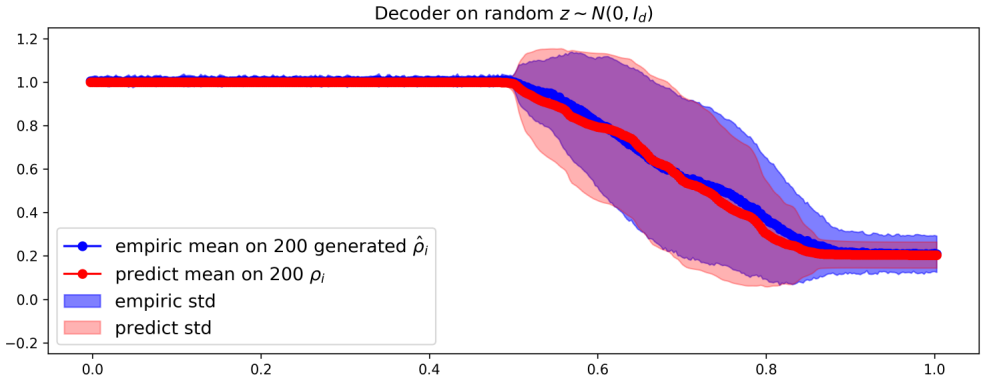

FIGURE 17 – Application du décodeur sur des lois normales multivariées

La figure **17** montre en rouge et rose respectivement la moyenne et l'écart-type pour 200 solutions de Burgers. La courbe bleue correspond à la moyenne sur 200 données générées par le décodeur. La zone bleue montre l'écart-type associé à ces données générées. On remarque que la moyenne pour les données générées se superpose dif-

ficilement sur la moyenne prévue, entre $x = 0.5$ et $x = 0.9$. De même, les zones d'incertitude représentées par l'écart-type sont ressemblantes mais ne se superposent pas.

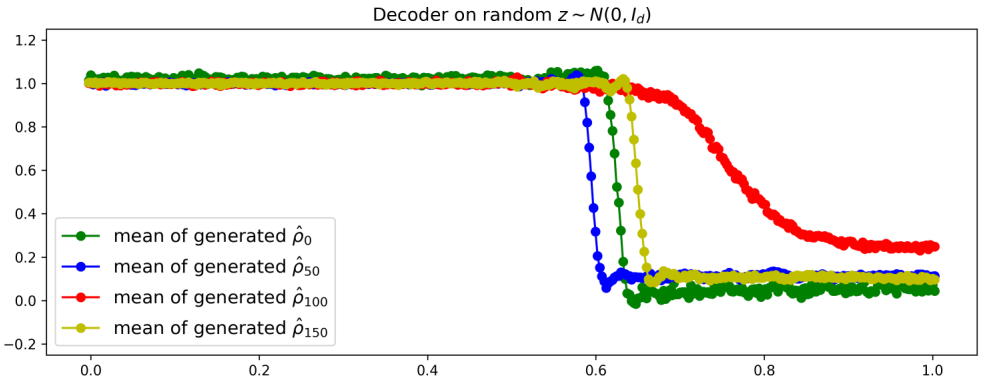

FIGURE 18 – Application du décodeur sur des lois normales multivariées

On constate sur la figure **18** que les moyennes des lois générées sont bien différentes les unes par rapport aux autres, on obtient donc des ρ pour différents temps.

Reproduction de la loi de ρ par le VAE, à partir des solutions de Burgers au temps 0.2

Dans cette partie, il s'agissait de considérer uniquement des solutions de Burgers dans le cas stochastique, et pour un temps final fixé à 0.2. Les données sont stockés dans le fichier *0.2_final_creneau.npy* (dossier *Documents*).

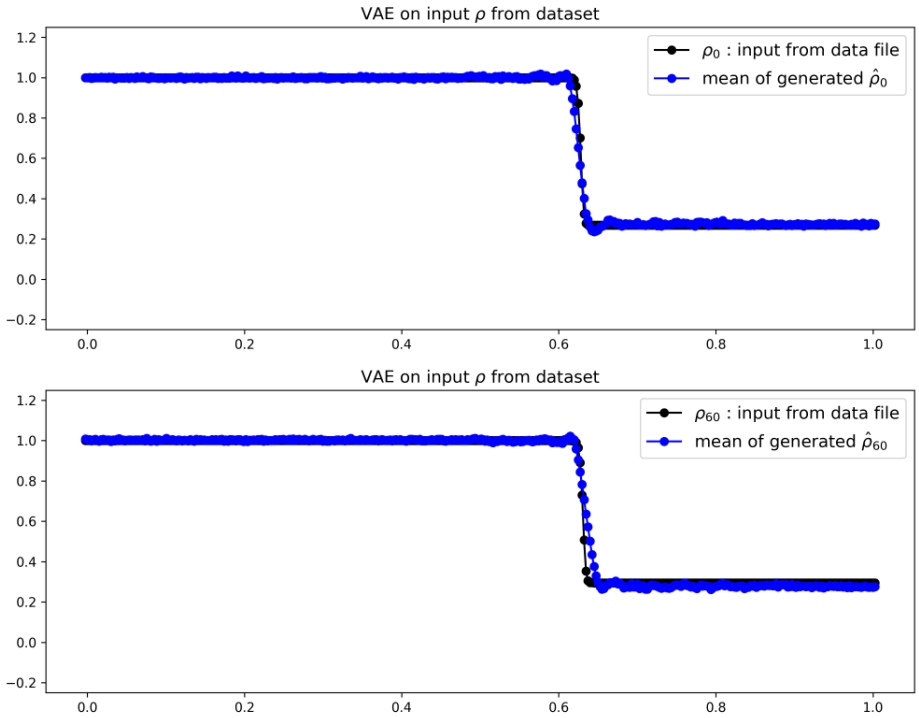

FIGURE 19 – Application du VAE sur le jeu de données de temps final 0.2

Tout d'abord, on remarque sur la figure **19** que les moyennes des données générées (en bleu) sont proches de celles obtenues pour les données d'entrées (en noir) mais elles ne se superposent pas parfaitement.

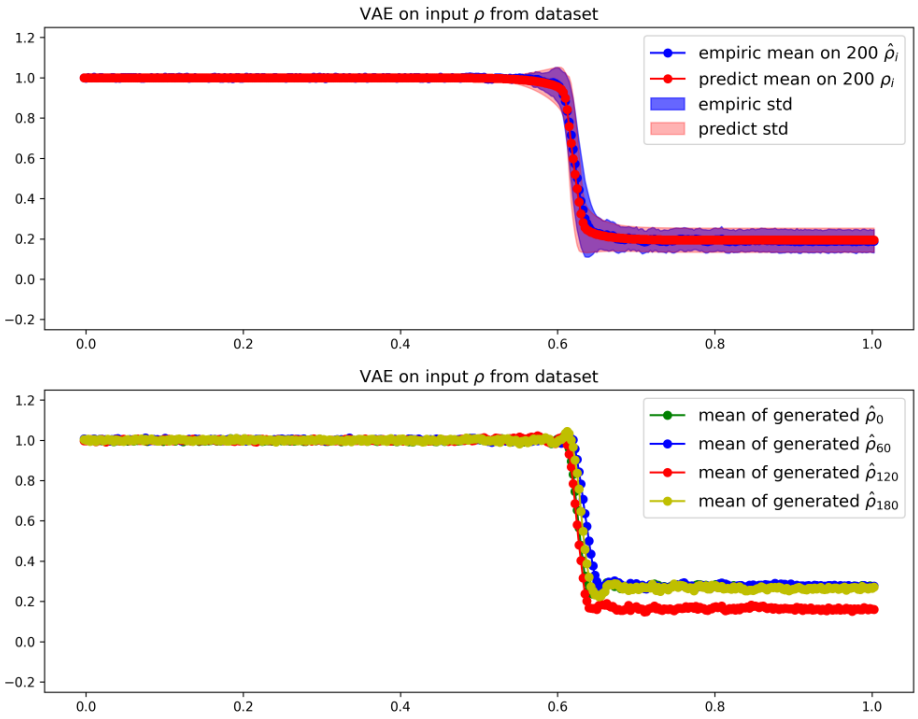

FIGURE 20 – Moyenne et écart-type calculés après encodage et décodage sur 200 réalisations de Burgers

Les premières courbes présentes sur la figure **20** représentent en rouge et rose respectivement la moyenne et l'écart-type pour 200 réalisations de Burgers au temps 0.2. La courbe bleue correspond à la moyenne sur 200 données générées par l'auto-encodeur ; les données d'entrée sont les réalisations de Burgers. La zone bleue montre l'écart-type associé à ces données générées.

On constate que la moyenne des données générées n'est pas parfaitement superposée à celle des données initiales autour de $x = 0.6$, mais sinon le résultat est très satisfaisant. De plus, on constate que les deux zones associées aux écart-types prévus et de sortie sont particulièrement ressemblantes malgré quelques fluctuations sur les bords.

En observant la deuxième partie de la figure **20**, on remarque que les données générées représentent un temps un peu différent car les courbes ne sont pas identiques.

5 Conclusion

Finalement, on constate que le processus de compression-décompression de l'auto-encodeur variationnel (VAE) est particulièrement efficace pour reproduire les lois de $\rho(t)$. Plus particulièrement, lorsque l'on prend en entrée un lot de données pour différents temps, on peut générer des distributions correctes pour différents temps. Lorsque le VAE prend en entrée un lot de données pour un temps fixé, il génère des lois associées à ce temps.

De même, le processus d'échantillonnage de la variable latente z à partir d'une loi normale multivariée, auquel on applique le décodeur, a permis de retrouver les lois de $\rho(t)$.

On peut donc en conclure qu'il est possible de représenter les lois de probabilités de $\rho(t)$ avec un auto-encodeur variationnel, et qu'on pourrait le réutiliser pour générer rapidement des données.

On peut cependant ajouter que l'on rencontre plus de difficultés à reproduire les solutions de Burgers dans le cas stochastique. De plus, le temps ne m'a pas permis d'aborder la deuxième partie du projet qui consistait à entraîner également le modèle réduit avant d'appliquer le décodeur.

Références

[1] Modèle génératif. (2017). *Data Franca.* https://datafranca.org/wiki/Mod%C3%A8le_g%C3%A9n%C3%A9ratif

[2] Understanding Variational Autoencoders (VAEs). (24 Septembre 2019). *Towards Data Science.* https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73

[3] Modèles génératifs : Autoencodeur et autoencodeur variationnel. *Clément Rambour.* http://cedric.cnam.fr/vertigo/Cours/RCP211/docs/VAE_1___autoencoders_et_variational_autoencoders.pdf

[4] Inria et son écosystème. *Inria.* https://www.inria.fr/fr

[5] Modèles Génératifs et Auto-encodeurs variationnels. (24 Mars 2020). *Pang R., Klevs A., Chou H.-R., Jain M.* https://atcold.github.io/pytorch-Deep-Learning/fr/week08/08-3/

[6] From Autoencoder to Beta-VAE. (12 Août 2018). *Weng L.* https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html#beta-vae

[7] Keras - Couches. (2020-2022). *iSolution.pro.* https://isolution.pro/fr/t/keras/keras-layers/keras-couches