

# Normalizing Flows

estimation de densités de probabilités

J.E Campagne

3 Oct. 2022

# Table des matières

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>   | <b>4</b>  |
| <b>2</b> | <b>Principe général</b>   | <b>4</b>  |
| <b>3</b> | <b>Différences entre NF, VAE et GAN</b>                               | <b>7</b>  |
| <b>4</b> | <b>Autoregressive Flows</b>   | <b>9</b>  |
| 4.1      | Schéma général . . . . .  | 10        |
| 4.2      | Autoregressive Models . . . . .                                       | 11        |
| 4.3      | Exemple de transformations: les <i>bijector</i> . . . . .             | 11        |
| 4.3.1    | Type Affine . . . . .   | 11        |
| 4.3.2    | Type MLP . . . . .  | 13        |
| 4.3.3    | Type Polynomial . . . . .   | 14        |
| 4.4      | Exemple de fonctions de sélection: les <i>conditionner</i> . . . . .  | 15        |
| 4.4.1    | Masquage MADE . . . . .   | 16        |
| 4.4.2    | Applications du masquage MADE: modèles MAF-like et IAF-like . . . . . | 18        |
| 4.4.3    | Alternative au masquage: <i>splitting</i> (Real NVP) . . . . .        | 21        |
| 4.4.4    | Généralisation: <i>multiple splits</i> . . . . .                      | 23        |
| 4.5      | Quelques remarques sur les Deep NF . . . . .                          | 24        |
| <b>5</b> | <b>Quelques mises en pratique</b>                                     | <b>25</b> |
| 5.1      | Simple bijector polynomial en 1D (TF) . . . . .                       | 25        |
| 5.2      | Cas en 2D: . . . . .  | 27        |
| 5.2.1    | ex. MADE vs MAF avec TensorFlow . . . . .                             | 27        |
| 5.2.2    | ex. MAF vs Real NVP avec Distrax . . . . .                            | 30        |

**6 Résumer & perspectives 33****A Transformation d'une distribution en une autre par NF 34**

## 1. Introduction

Il n'est pas difficile de se convaincre que l'estimation d'une densité de probabilité (notée *pdf* par la suite)  $p_X$  telle que

$$\mathbb{P}(X \in I \subset \mathbb{R}^d) = \int_{\mathbb{R}^d} \mathbb{1}_{X \in I} p_X(x) dx \quad (1)$$

est un problème qui survient dans les domaines liés aux statistiques que cela soit en Math ou en Physique. Beaucoup de méthodes paramétriques ou non sont disponibles. Concernant les méthodes non-paramétriques on peut citer les méthodes à noyaux avec le fameux Gaussian Kernel; et concernant les méthodes paramétriques, on peut tout aussi citer la toute aussi fameuse inférence variationnelle bayésienne. Il y a une autre méthode qui prend ses racines dans le domaine mathématique de la Théorie du transport dans l'élaboration de transformations qui changent par un exemple un bruit blanc en une autre densité de probabilité, il s'agit des *Normalizing Flows* (NF par la suite).

Ces NF sont à l'origine d'avancées récentes de *modèles génératifs* (ex. [Kingma and Dhariwal \(2018\)](#)), ils sont à l'œuvre dans la méthode de *Stochastic Variational Inference* (aka SVI) ([Gregory and Delbourgo, 1982](#)), et sont un ingrédient dans une méthode d'*inférence sans likelihood* (ex. [Papamakarios et al. \(2018\)](#)).

Cette note ne peut couvrir tout le spectre des usages des NF. Les personnes intéressées pourront se référer aux articles de [Papamakarios et al. \(2019\)](#) et [Kobyzev et al. \(2019\)](#).

## 2. Principe général

Le principe général est relativement simple car il s'agit ni plus ni moins d'opérer un changement de variable tel que si  $Z$  est *pdf*  $p_z(Z)$ :

$$X = T(Z) \quad \text{avec} \quad Z \sim p_z(Z) \quad (2)$$

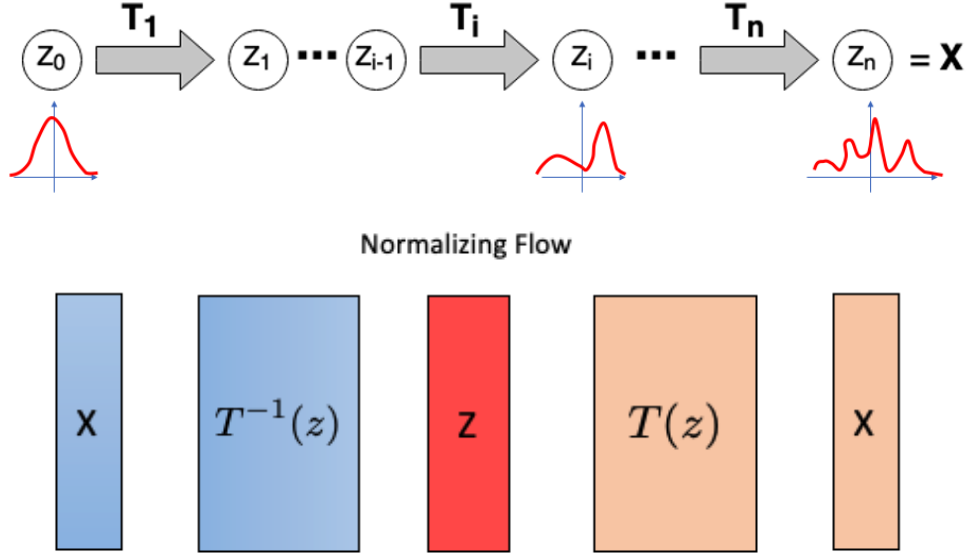


FIGURE 1 – Haut: Schéma qui montre qu'à partir d'une distribution simple de  $Z_0$  en opérant des transformations  $T_i$  inversibles et différentiables, l'on aboutit à une distribution de  $X$  laquelle est beaucoup plus complexe (par ex. multimodale). Bas: Schéma qui met en lumière une architecture proche des VAE de la section 3 (Fig. 2).

La *pdf*  $p_x(X)$  de  $X$  s'obtient alors selon<sup>1</sup>

$$Z = T^{-1}(X); \quad p_x(X) = p_z(Z) |\det J_T(Z)|^{-1} = p_z(T^{-1}(X)) |\det J_{T^{-1}}(X)| \quad (3)$$

Ceci n'est possible que si  $T$  est **inversible** et si  $T$  et  $T^{-1}$  sont des **applications différentiables**, il faut en effet pouvoir calculer le Jacobien  $J$  qui mesure le changement de volume. Ce sont donc des **difféomorphismes**. Au passage, **cela fixe la dimensionnalité de  $Z$  à celle de  $X$** . Dans les bibliothèques (et la littérature), la transformation  $T$  est souvent appelée un **bijector**, avec son mode **forward** et son mode **inverse**<sup>2</sup> quand on applique respectivement  $T$  ou  $T^{-1}$ .

L'intérêt est que l'on peut composer les transformations de telle sorte que

$$T = T_1 \circ T_2 \circ \dots \circ T_n \quad (4)$$

1.  $p_x(a)$  signifie que la *pdf* de  $x$  est évaluée avec la variable  $a$ , ex.  $p_x = \mathcal{N}(\mu, 1)$  alors  $p_x(a) \propto \exp\{-(a - \mu)^2/2\}$ .

2. En effet, les bibliothèques n'utilisent pas le vocable "backward".

avec toutes les  $T_i$  inversibles et différentiables. Primo l'inverse  $T^{-1}$  est facilement calculable selon  $T^{-1} = T_n^{-1} \circ T_{n-1}^{-1} \circ \dots \circ T_1^{-1}$  et secundo le déterminant du jacobien de  $T$  est le produit des déterminants des jacobiens des  $n$  transformations ( $Z_0 = Z$  et  $Z_n = X$ ):

$$Z_i = T_i(Z_{i-1}) \quad \log |\det J_T| = \sum_i \log |\det J_{T_i}(z_{i-1})| \quad (5)$$

On peut ainsi à partir d'une distribution de base  $p_u(u)$  simple (ex. une gaussienne multivariée de  $d$  variables  $\mathcal{N}(0, \mathbf{1}_d)$ ) modéliser des distributions de  $X$  complexes comme schématisé sur la figure 1. Notons que le calcul des déterminants (Eq. 5) peut être couteux sauf si les transformations sont spéciales, en particulier si leurs jacobiens sont des **matrices triangulaires**.

Connaissant  $p_u$  et disposant d'un lot de valeurs  $(X^{(i)})_{i \leq N}$ , que l'on suppose être issues de tirages *iid* d'une loi  $p_x$ , seule la transformation  $T$  (et ses composantes dans un schéma itératif) est inconnue. Mettons que  $T$  soit un élément d'une famille de transformations dont il faut déterminer les paramètres  $\phi$  afin de la définir pleinement, alors on procède en *maximisant la vraisemblance* que les  $(X^{(i)})_i$  soit effectivement issus de la loi  $p_x$ , c'est-à-dire en *minimisant*

$$\mathcal{L}(\phi) = - \sum_{i=1}^N \log p_x(X^{(i)}) \quad (6)$$

En fait:

- pour *l'entraînement*, on a uniquement besoin de  $T^{-1}$  car on utilise la seconde égalité de l'équation 3 pour calculer  $p_x(X)$  et  $\mathcal{L}(\phi)$ .
- si l'on veut *générer de nouveaux échantillons* de  $X$  à partir de  $p_x$  obtenue, il vaudrait mieux avoir accès à la transformation  $T$  directement pour utiliser les relations 2.

Évidemment, on cherche des familles de transformations qui permettent (1) de calculer la *pdf*  $p_x$  et (2) de générer des nouveaux échantillons le plus efficacement possible, en particulier sans avoir à inverser la transformation.

Avant de détailler quelques méthodes pratiques, si les NF permettent de générer de nouveaux échantillons  $X$ , nous connaissons d'autres méthodes qui permettent de telles générations. Quelles sont les différences entre ces différentes méthodes? La section suivante apporte quelques éléments de réponse.

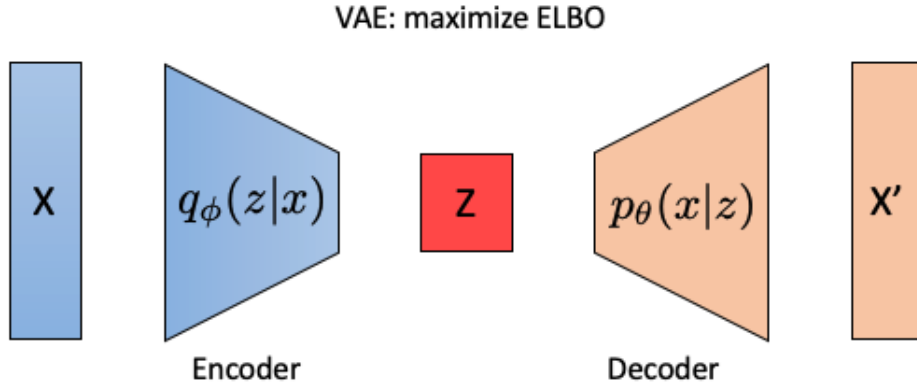


FIGURE 2 – Schéma des deux parties (encoder/decoder) d'un Variational Autoencoders.

### 3. Différences entre NF, VAE et GAN

Une fois mis en place un NF reliant d'une manière biunivoque  $(X, p_x(X))$  à  $(Z, p_z(Z))$ , on peut tout à fait générer des échantillons  $(X_i)$  à partir d'échantillons  $(Z_i)$  tirés selon  $p_z$  qui peut être par exemple un loi normale centrée. Or, il y a d'autres méthodes qui permettent d'en faire autant: les *Variational Autoencoders* (VAE) et les *Generative Adversarial Networks* (GAN).

Un simple *auto-encodeur* apprend à réduire la dimensionalité de  $X$  par compression en une variable  $Z$ , mais il est très limité par conception pour ce qui concerne la génération de nouveaux échantillons  $X'$  (il n'a pas appris à générer des  $X$  réalistes). La version *variationnelle* (Kingma and Welling, 2013) est présentée sur la figure 2. Initialement, on se pose la question de pouvoir générer des échantillons  $X$  à partir de la génération d'une variable latente  $Z$  selon un prior  $\pi(Z)$  et l'usage d'un likelihood (*decodeur*)  $p_\theta(X|Z)$ . Mais, faute de pouvoir accéder à la distribution a posteriori vraie  $p(Z|X)$ , on a recours à une approximation  $q_\phi(Z|X)$  (*encodeur*). Or, l'évidence  $p_\theta(X)$  ou *marginal likelihood* quand on

prend le log peut s'exprimer selon <sup>3</sup>

$$\begin{aligned} \log p_\theta(X) &= D_{KL}(q_\phi(Z|X)||p(Z|X)) + \mathcal{L}(\phi, \theta; X) \geq \mathcal{L}(\phi, \theta; X) \\ \text{avec } \mathcal{L}(\phi, \theta; X) &= \mathbb{E}_{Z \sim q_\phi(Z|X)}[\log p_\theta(X|Z)] - D_{KL}(q_\phi(Z|X)||\pi(Z)) \end{aligned} \quad (7)$$

où la fonction  $\mathcal{L}(\phi, \theta; X)$  n'est autre que *l'evidence lower bound* (aka ELBO) que l'on souhaite maximiser pour rendre compte que les observations  $X$  sont générées par la *pdf* sous-jacente  $p_\theta(X)$ . Imaginons un instant que

$$p_\theta(X|Z) = \mathcal{N}(d(Z), \sigma^2 I) \quad (8)$$

avec  $d$  une fonction, alors le premier terme de la fonction  $\mathcal{L}(\phi, \theta; X)$  devient

$$\mathbb{E}_{Z \sim q_\phi(Z|X)}[\log p_\theta(X|Z)] = \mathbb{E}_{Z \sim q_\phi(Z|X)} \left[ -\frac{\|X - d(Z)\|^2}{2\sigma^2} \right] \quad (9)$$

Le maximum de ce terme atteint pour " $d(Z) = X$ " assure le principe du maximum de vraisemblance afin que le decodeur fournisse bien des échantillons  $X$  vraisemblables. Le second terme est une sorte de régularisation, car il contraint l'encodeur à donner une distribution aussi proche que possible de  $\pi(Z)$  laquelle est choisie à l'avance de telle sorte que l'espace de  $Z$  soit bien couvert (ex. une gaussienne centrée assez large). Sans entrer dans les détails de la mise en œuvre d'un VAE, celui-ci peut donc apprendre une *transformation bi-directionnelle* entre une distribution complexe de  $X$  et une distribution plus simple connue à l'avance de  $Z$ , et l'on peut faire de l'inférence et de la génération. **Cependant, on voit tout de suite une différence majeure avec les NF: pour un VAE la dimensionnalité de  $Z$  est largement plus petite que celle de  $X$  (*compression*) alors que dans le cas des NF  $Z$  et  $X$  sont de même dimension.**

Concernant un GAN dans la version de base (Goodfellow et al., 2014), on construit non seulement un *générateur* d'échantillons  $X$ , noté  $G(Z)$  où  $Z$  est généré selon le prior  $\pi(Z)$  gaussien comme pour le VAE, mais aussi un *discriminateur*  $D(X)$  qui donne la probabilité que  $X$  provienne des données ou bien du générateur. Il faut donc entraîner à la fois  $D$  à maximiser son score à bien discriminer le vrai de vrai du simili-vrai, et à

---

3. Nb. on considère des tirages *iid* donc on peut raisonner sur 1 échantillon. Rappel: la divergence de Kullback-Leibler est définie par  $D_{KL}(p||q) = \mathbb{E}_{x \sim p(x)}[\log(p(x)/q(x))]$ . *hint*: On note que  $p(Z|X) = p_\theta(X|Z)\pi(Z)/p_\theta(X)$  et l'on prend alors l'espérance vie-à-vis de  $Z \sim q_\phi(Z|X)$ .



optimiser  $G$  pour leurrer le discriminateur. Cela se fait via la résolution du problème de minimax suivant

$$\min_G \max_D \left\{ \mathbb{E}_{X \sim p_{data}(X)} [\log D(X)] + \mathbb{E}_{Z \sim p_Z(Z)} [\log(1 - D(G(Z)))] \right\} \quad (10)$$

En fait, à fonction  $G$  fixée, le discriminateur optimal  $D^*$  est obtenu pour

$$D^*(X) = \frac{p_{data}(X)}{p_{data}(X) + p_G(X)} \quad (11)$$

avec  $p_G(X) = p_Z(Z) |\det J_G(Z)|^{-1}$  (nb. la transformation  $G$  serait un NF si elle était inversible), et le problème minimax peut être reformulé alors comme

$$\min_G \left\{ D_{KL} \left( p_{data} \left\| \frac{p_{data} + p_G}{2} \right\| \right) + D_{KL} \left( p_G \left\| \frac{p_{data} + p_G}{2} \right\| \right) \right\} \quad (12)$$

qui montre bien alors que le minimum est atteint pour  $p_{data} = p_G$ . **Cependant, comme pour le VAE,  $Z$  n'a pas la même dimensionnalité que  $X$ , et il n'y a de plus aucune approximation du likelihood.** Cependant, au schéma précédent il faut apporter des modifications pour éviter des problèmes de stabilité d'optimisation et éviter que le discriminateur ne fasse de l'overfitting qui obère la qualité du générateur pour vraiment échantillonner  $p(X)$  (pb. de généralisation).

Donc, la principale différence structurelle est la dimension de l'espace des variables latentes  $Z$  bien plus grande pour les Normalizing Flows. Il semble que cela confère expérimentalement à ces derniers plus de stabilité.

## 4. Autoregressive Flows

Parmi les différents types de Normalizing Flow, il y en a un particulièrement intéressant, il s'agit des *Autoregressive Flows*.

## 4.1 Schéma général

Soit  $Z = (z_1, \dots, z_d) \in \mathcal{X}_d \subset \mathbb{R}^d$ , et  $X$  un vecteur à  $d$ -composantes  $(x_1, \dots, x_d)$  obtenues par <sup>4</sup>

$$\forall i \in \llbracket 1..d \rrbracket \quad x_i = T(z_i; h_i) \quad h_i = C_i[z_1, \dots, z_{i-1}] = C_i[z_{1:i-1}] \quad (13)$$

où  $T$  est une **transformation strictement monotone croissante**, donc inversible, de paramètre  $h_i$ , lui-même une **fonction ne dépendant que des  $(i-1)$ -premières composantes de  $Z$** . La fonction  $C_i$  conditionne en quelque sorte les paramètres  $h_i$  dont dépend la transformation. Dans la littérature, on trouve le vocable de **conditionner** pour les  $C_i$ . Remarquons de suite deux propriétés essentielles:

- **inversibilité**: la transformation inverse est immédiate

$$\forall i \in \llbracket 1..d \rrbracket \quad z_i = T^{-1}(x_i; h_i) \quad h_i = C_i[z_{1:i-1}] \quad (14)$$

- **Jacobien triangulaire**: le jacobien de la transformation  $T$  a pour structure

$$J_T = \left[ \frac{\partial x_i}{\partial z_j} \right] = \begin{cases} 0 & \forall i, j \text{ tq. } i > j \\ \frac{\partial x_i}{\partial z_i}(z_i; h_i) & \forall i \end{cases} \quad (15)$$

les autres éléments ne rentrant pas dans le calcul du déterminant, il n'y a pas lieu de les mentionner. La première relation vient directement de la structure des conditionneurs  $C_i$ . Le déterminant est alors égal à

$$\log |\det J_T| = \sum_{i=1}^d \log \left| \frac{\partial x_i}{\partial z_i}(z_i; h_i) \right| \quad (16)$$

Ces deux propriétés rendent les Autoregressive Flows très attrayants au regard de la discussion de la section 2. Du schéma général, il nous faut préciser les transformations ( $T$ ) et les conditioners ( $C$ ). Notons que les conditioners prennent tout leur sens uniquement pour le cas où  $d > 1$ . Avant cela faisons une remarque qui va s'avérer importante.

---

4. Notons ici le rôle de  $X$  et  $Z$  sont interchangeables, c'est-à-dire que  $T$  pourrait très bien représenter la transformation de  $X$  à  $Z$ , c'est purement un choix de notation.

## 4.2 Autoregressive Models

Dans l'Appendix A, on utilise le fait qu'en dimension  $d$ ,  $p_x(X)$  peut s'écrire comme le produit de  $d$  *pdf* conditionnelles:

$$p_x(x_1, \dots, x_d) = \prod_{i=1}^d p_x(x_i | x_{1:i-1}) \quad (17)$$

Alors il devient naturel de modéliser  $p_x(x_i | x_{1:i-1})$  selon

$$p_x(X) = p_x(x_i | x_{1:i-1}) = p(x_i; h_i) \quad h_i = C_i[x_{1:i-1}] \quad (18)$$

Au passage, une méthode variationnelle modéliserait directement  $p_x(X)$  par une probabilité paramétrée  $q(X; \phi)$  dont il faudrait déterminer  $\phi$  pour approcher  $p_x(X)$  en minimisant  $D_{KL}(q||p_X)$ .

D'après le schéma (Eq. 18) et l'Appendix A, on sait que la v.a  $X$  est issue de la v.a  $U = \mathcal{U}([0, 1]^d)$  par action de la fonction cumulatrice des  $p(x_i; h_i)$ . Ainsi, les *modèles autoregressifs* sont en fait des *flows autoregressifs* et sont conçus pour **modéliser des densités de probabilités** comme par exemple dans les références (Huang et al., 2018; Papamakarios et al., 2017; Germain et al., 2015), ou pour **procéder à la génération de nouveaux échantillons** par exemple dans la méthode d'inférence variationnelle stochastique (Kingma et al., 2016) (voir Sec. 4.4.2).

## 4.3 Exemple de transformations: les *bijector*

### 4.3.1 Type Affine

Un premier exemple qui vient à l'esprit est d'utiliser des transformations affines où

$$T(z_i; h_i) = e_i^\alpha z_i + \mu_i, \quad h_i = (\alpha_i, \mu_i) \quad (19)$$

que l'on peut étendre au cas où  $\alpha_i$  et  $\mu_i$  sont des fonctions comme

$$\alpha_i = \alpha_i[z_{1:i-1}], \quad \mu_i = \mu_i[z_{1:i-1}] \quad (20)$$

et la transformation <sup>5</sup> peut s'écrire d'une manière matricielle ( $\odot$  *element-wise product*, aka *Hadamard product*)

$$X = Z \odot e^\alpha + \boldsymbol{\mu} \quad (21)$$

Ces transformations affines sont simples à implémenter, et l'inversibilité vient de ce que  $e^\alpha > 0$ . Elles sont utilisées comme brique élémentaires et associées à d'autres types de modélisations.

Par exemple, la référence (Papamakarios et al., 2017) pour modéliser la loi  $p(X)$  où  $X \in \mathbb{R}^d$ , procède selon le schéma des Autoregressive Models (Sec. 4.2) et utilise une loi normale pour modéliser  $p(x_i|x_{1;i-1})$

$$p(x_i|x_{1;i-1}) = \mathcal{N}(x_i; h_i = \{\mu_i, e^{2\alpha_i}\}) \quad \text{avec} \quad \begin{cases} \mu_i &= f_{\mu_i}(x_{1;i-1}) = \mu_i[x_{1;i-1}] \\ \alpha_i &= f_{\alpha_i}(x_{1;i-1}) = \alpha_i[x_{1;i-1}] \end{cases} \quad (22)$$

Or, cela revient à tirer les composantes  $x_i$  de  $X$  selon

$$\begin{aligned} x_1 &= u_1, & x_2 &= u_2 e^{\alpha_2[x_1]} + \mu_2[x_1], & x_3 &= u_3 e^{\alpha_3[x_1, x_2]} + \mu_3[x_1, x_2], & \dots \\ \text{avec } (u_i &\sim \mathcal{N}(0, 1))_{i \leq d} \end{aligned} \quad (23)$$

qui en notation vectorielle donne une unification des relations précédentes selon (voir également la figure 5 de la section 4.4.2)

$$X = \mathbf{F}(U) = U \odot e^\alpha + \boldsymbol{\mu} \quad U \sim \mathcal{N}(0, \mathbf{1}_d) \quad \text{avec} \quad \forall i \leq d \quad \begin{cases} \mu_i &= \mu_i[x_{1;i-1}] \\ \alpha_i &= \alpha_i[x_{1;i-1}] \end{cases} \quad \text{(MAF)} \quad (24)$$

qui est bien un *NF avec une transformation affine*. La transformation  $\mathbf{F}$  est inversible, on a en effet

$$u_1 = x_1, \quad u_2 = (x_2 - \mu_2[x_1])e^{-\alpha_2[x_1]}, \quad u_3 = (x_3 - \mu_3[x_1, x_2])e^{-\alpha_3[x_1, x_2]}, \quad \dots \quad (25)$$

---

5. Les librairies font référence à des fonctions de type `shift_and_log_scale` pour faire référence à  $\mu$  et  $\alpha$ .

que l'on peut écrire

$$U = \mathbf{F}^{-1}(X) = (X - \boldsymbol{\mu}) \odot e^{-\boldsymbol{\alpha}} \quad \text{avec} \quad \forall i \leq d \begin{cases} \mu_i &= \mu_i[x_{1;i-1}] \\ \alpha_i &= \alpha_i[x_{1;i-1}] \end{cases} \quad (\text{MAF}^{-1}) \quad (26)$$

Le Jacobien de  $\mathbf{F}^{-1}$  est tel que  $(\alpha_1 = 1, \mu_1 = 0)$

$$\forall j > i \quad (J_{\mathbf{F}^{-1}})_{i,j} = \frac{\partial u_i}{\partial x_j} = 0; \quad \text{et} \quad (J_{\mathbf{F}^{-1}})_{i,i} = e^{-\alpha_i} \quad (27)$$

C'est donc **une matrice triangulaire inférieure** telle que

$$\det J_{\mathbf{F}^{-1}} = \prod_{i=1}^d e^{-\alpha_i} = \exp\left\{-\sum_{i=1}^d \alpha_i\right\} > 0 \quad (28)$$

On a bien à faire avec un Normalizing Flow et selon Eq. 3

$$p(X) = \frac{1}{(2\pi)^{d/2}} \exp\left\{-\frac{1}{2} \sum_{i=1}^d u_i^2\right\} \exp\left\{-\sum_{i=1}^d \alpha_i[x_{1;i-1}]\right\} \quad (29)$$

où les  $u_i[\mathbf{x}]$  sont définis par les relations Eqs. 25. Il nous faudrait implémenter les fonctions  $\mu_i[x_{1;i-1}]$  et  $\alpha_i[x_{1;i-1}]$  pour être complet. La référence (Papamakarios et al., 2017) utilise le systèmes des masques (Sec. 4.4.1). Un autre modèle similaire (Kingma et al., 2016) est introduit à la section 4.4.2.

### 4.3.2 Type MLP

Au-delà de ces transformations linéaires, les transformations non-linéaires ont été recherchées comme brique de base afin d'obtenir une classe de transformation plus large (on parle souvent d'*expressivity*). Or, on sait que le problème de classification de type "XOR" a été résolu par l'usage d'un neurone Perceptron, et donc on peut penser à ajouter une fonction d'activation non-linéaire de type: tanh, sigmoïde, ou bien un des rectificateurs

linéaires. Par exemple, on peut élaborer un flow à partir de la transformation suivante

$$T(z_i; h_i) = w_{i0} + \sum_{k=1}^K w_{ik} \rho(\alpha_{ik} z_i + \beta_{ik})$$

$$h_i = \{w_{i0}, (w_{ik}, \alpha_{ik}, \beta_{ik})_{k \leq K}\} \quad (30)$$

(on peut également ajouter à la liste de paramètres ceux qui interviennent éventuellement pour décrire la fonction *point-wise*  $\rho$ ). Pour assurer l'inversibilité, il faut contraindre tous les poids  $w_{ik}$  et tous les facteurs d'échelle  $\alpha_{ik}$  à être strictement positifs. On peut également enchaîner ces transformations pour construire des *Multi-Layers Perceptrons* qui réalisent des transformations strictement monotones croissantes. Le caractère "expressif" de ce type de flow est à mettre en parallèle avec le théorème d'Universalité des réseaux à 1-couche cachée.

Le mode "forward" est assez simple à mettre en œuvre, et le déterminant du jacobien est calculé soit par auto-diff ou par back-prop. Par contre, pour procéder au mode "inverse" on est généralement confronté à l'impossibilité pratique d'obtenir une forme analytique de la transformation inverse. On a alors recours à un algorithme générique d'inversion<sup>6</sup>. Ces modèles de flow sont utilisés comme brique de base pour des modèles plus complexes. Remarquons que les MLP en tant que tels sont souvent utilisés pour générer des modèles de paramètres de transformations (voir section 4.4.1 sur la technique de masquage).

### 4.3.3 Type Polynomial

Quand on pense non-linéarité, on peut naturellement envisager l'usage de polynômes simples (ex. section 5) ou des polynômes qui forment une base de  $[0, 1]$  par exemple comme les polynômes de Bernstein (Ramasinghe et al., 2021). Mais, en règle général, le problème reste l'inversion comme pour les MLP, quand bien même il existe des algorithmes efficaces de recherche de racines. Donc, avoir accès directement à l'inverse est une requête non négligeable.

Dans ce contexte, on était mis en œuvre les "**Spline Flows**" comme par exemple dans la référence (Durkan et al., 2019). S'il existe plusieurs moutures de transformations basées

---

6. ex. on suppose que l'inverse existe dans un intervalle borné; pour  $x$ , rechercher  $z$  tel que  $T(z) = x$  par dichotomie.

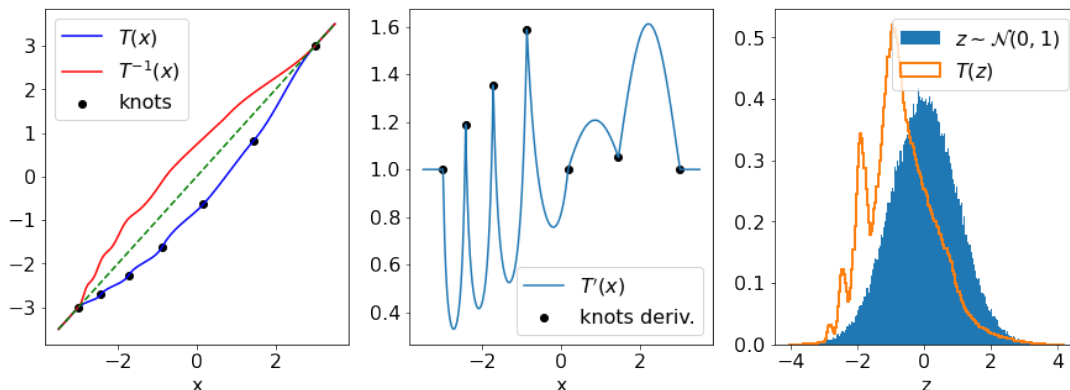


FIGURE 3 – Transformation à base de rational-quadratic splines dont on a fixé les positions  $(x, y)$  des nœuds (knots noirs) ainsi que leurs dérivées sachant que les points extrêmes  $(-3, -3)$  et  $(3, 3)$  sont fixés. Au-delà du range  $[-3, 3]^2$  dans cette version  $T(x) = x$ . A droite, est montrée la transformation d'une distribution  $\mathcal{N}(0, 1)$  par la transformation de gauche.

sur des splines, une retient l'attention: les **rational-quadratic splines**. Sur chaque morceau, la transformation se présente comme le rapport entre 2 polynômes de degré 2 et l'on ajoute des contraintes au niveau des jointures et des bords afin d'obtenir une description complète. [Gregory and Delbourgo \(1982\)](#) ont explicité les formules des transformations et de leurs inverses<sup>7</sup>. Le lecteur intéressé peut s'y référer pour les détails. Elles font partie de la boîte à outils standard. Les variantes de cette famille de transformations diffèrent par les conditions aux limites. Un exemple de transformation définie sur l'intervalle  $[-3, 3]$ , et son impact sur une distribution Gaussienne sont présentés sur la figure 3. Cela illustre bien qu'à partir d'une distribution uni-modale, on peut obtenir une distribution multi-modale.

#### 4.4 Exemple de fonctions de sélection: les *conditionner*

Dans le cas où  $X$  a plusieurs composantes ( $d > 1$ ), l'implémentation des conditionners donne des architectures complexes. A priori, on pourrait penser qu'il n'y a pas de contraintes sur les  $C_i$ , cependant si on les considère comme des modèles indépendants, il en faudrait  $d$ , et chacun aurait en moyenne  $d/2$  variables. Cela manifestement ne passe pas à l'échelle. Tout comme les grands réseaux de neurones ne sont pas uniquement des interconnexions denses, il nous faut introduire de la structuration. Une chose par contre

7. Le fait qu'il est simple de trouver l'expression inverse vient du fait que la transformation est monotone sur le support défini à l'avance (cf. au-delà la transformation est l'identité) et ainsi on peut trouver la formule explicite de la racine de l'équation du second degré avec le bon signe.

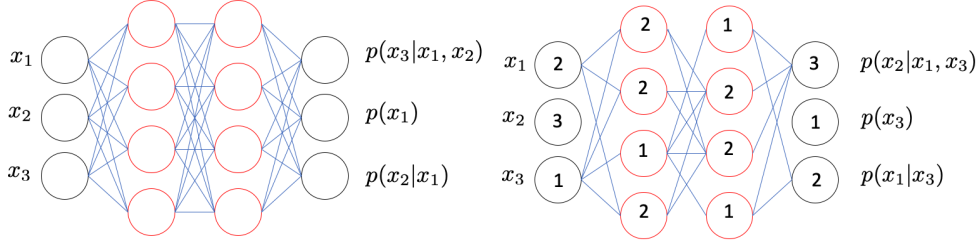


FIGURE 4 – Algorithme MADE de masquage. Haut: schéma d'un MLP qui à partir de l'entrée  $(x_1, x_2, x_3) = X \in \mathbb{R}^3$  (cf.  $d = 3$ ) voudrait prédire les probabilités conditionnelles  $p(x_1)$ ,  $p(x_2|x_1)$  et  $p(x_3|x_1, x_2)$ . Cependant, au minimum ce schéma est déficient car il y a trop de connexions. Bas: par masquage de certaines interconnexions le schéma peut être rendu viable (voir texte). Si dans le schéma du bas le triplet de labels des entrées est  $\{2, 3, 1\}$  et donne les probabilités mentionnées, le triplet initial  $\{1, 2, 3\}$  donnerait les probabilités en sortie du schéma du haut. En fait la sortie ayant pour label  $i$  peut être écrite en fonction des labels des entrées selon le pseudo-code  $i \leftarrow o[i\{j\} < i]$  (nb. dans les schémas de la figure  $o$  fait référence aux probabilités conditionnelles).

qui rend certains modèles de conditionners plus adaptés que d'autres, est leurs aptitudes à garder le caractère parallèle de leurs calculs<sup>8</sup>. On trouve dans la littérature des modèles de conditioners à base de masquage (*mask*) (ex. Papamakarios et al. (2017); Uria et al. (2016); Germain et al. (2015)) et de couplage de couches (*coupling layers*) (ex. Durkan et al. (2019); Dinh et al. (2016)).

#### 4.4.1 Masquage MADE

L'idéal pour le calcul des paramètres  $h_i = C_i[z_{1:i-1}]$  (Eq. 13) serait de pouvoir tous les calculer à partir de  $(z_1, \dots, z_d)$  en 1 seule passe (forward). On pourrait alors mettre en œuvre un réseau dense de type MLP ayant les  $(z_i)_{i \leq d}$  en entrée et délivrant en sortie les  $(h_i)_{i \leq d}$ . Soit, mais il faut pouvoir contraindre  $h_i$  à ne dépendre que des  $z_{i:i-1}$ . Rien de plus "simple", il suffit de couper des connections. C'est ce qui a été introduit par les auteurs de la référence Germain et al. (2015) dans le cadre de modélisation de densité (Sec. 4.2). Voyons comment cela marche.

Tout d'abord, mettons nous dans le contexte original de Germain et al. (2015) (MADE) où la problématique était de *transformer l'architecture d'un Auto-Encoder* afin de prédire la *pdf*  $p(X)$  (ici  $X$  est une notation générique). Pour mémoire, un AE a pour tâche de trouver à partir d'entrée  $x \in [0, 1]^d$ , une représentation de variables latentes

8. Cela rend d'une certaine mesure des modèles à base de mémoire récurrente (RNN, GRU, LSTM) inefficaces.



$z = h(x)$  telles que la sortie  $o$  soit donnée dans le cas le plus simple par l'enchaînement

$$z = h(x) = \rho(Wx + b) \quad o = \sigma(Vz + c) \quad (31)$$

avec  $\rho$  une non-linéarité,  $\sigma$  une sigmoïde,  $W$  et  $V$  des matrices de poids, et  $(b, c)$  des vecteurs de biais. Dans le cas d'un pure AE, celui-ci est entraîné pour que  $o$  reproduise l'entrée, alors que dans le cas qui nous intéresse  $o$  représente des probabilités conditionnelles. Le schéma AE ci-dessus peut-être étendu au cas d'un MLP qui connecte l'entrée à la sortie.

Dans le cas où  $d = 3$ , la figure 4 de gauche voudrait donc prétendre à prédire  $p(x_1)$ ,  $p(x_2|x_1)$  et  $p(x_3|x_1, x_2)$ . Mais on aurait tout aussi modéliser  $p(X)$  selon le produit d'autres probabilités conditionnelles, en effet:

$$p(X) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2) = p(x_3)p(x_2|x_3)p(x_1|x_2, x_3) \quad (32)$$

Donc en fait, il y a une question sous-jacente: quel est le schéma le plus pertinent pour décrire les données? Nous y reviendrons. Maintenant, le réseau de la figure 4 (gauche) ne peut répondre correctement à un quelconque schéma, car on voit bien qu'au travers des interconnexions  $p(x_2|x_1)$  dépend de  $x_3$ . D'où l'idée de procéder à des coupures en multipliant par 0 (1) les connexions qu'il faut éliminer (garder). La figure 4 (droite) donne le résultat de l'algorithme de masquage de [Germain et al. \(2015\)](#) (MADE). Comment cela fonctionne-t'il?

- Premièrement, on étiquette les entrées (couche  $\ell = 0$ ) d'une manière aléatoire à chaque itération d'entraînement (epoch). En pratique, l'expérience montre qu'il suffit de faire un rolling d'une seule dimension, mais mettons qu'au cours d'une boucle d'optimisation le triplet  $\{1, 2, 3\}$  soit étiqueté  $m^0 = \{2, 3, 1\}$ . L'entrée  $j$  qui porte le label  $m^0(j) = d$  (ici  $j = 2$  en numérotant à partir de 1) a un statut spécial comme nous allons le voir.
- Prenons ensuite la première couche cachée ( $\ell = 1$ ). On affecte à chaque neurone  $i$  un entier  $m^1(i) \in \{1, \dots, d-1\}$ . Notons que le nombre de neurones  $K^1$  de la couche peut être plus petit, égal ou plus grand que  $d$ . La connexion à l'entrée  $j$  ( $m^0$ ) du neurone  $i$  de label  $m^1(i)$  est validée (mise à 1) si  $m^1(i) \geq m^0(j)$ , sinon elle est invalidée (mise à 0). Par la même, on se rend compte que l'entrée qui a pour label  $m^0(j) = d$  n'a pas de connexion.

- Concernant la seconde couche cachée ( $\ell = 2$ ), on procède à la labelisation des neurones ainsi qu'à la validation de leurs connexions avec les neurones de la couche précédente ( $\ell = 1$ ) comme précédemment.
- La couche de sortie est composée de  $d$  neurones et les connexions avec les neurones de la dernière couche cachée sont validées uniquement dans le cas où  $m^L(j) < m^s(i)$ . Ainsi, le neurone de sortie de label égal à 1, qui n'est pas connecté, correspond alors à  $p(x_{i_0})$  tel que  $m^0(i_0) = 1$ . Et en suivant les interconnexions validées on trouve à quelle probabilité conditionnelle correspond chaque neurone de sortie. On se rend compte alors que la sortie  $m^s = 2$  est connectée à l'entrée  $i$  telle que  $m^0(i) = 1$ , donnant  $p(2|1) = p(x_1|x_3)$ , et la sortie telle que  $m^s = 3$  est connectées aux entrées  $(i, j)$  telles que  $m^0(i) = 1$  et  $m^0(j) = 2$  donnant alors  $p(3|1, 2) = p(x_2|x_3, x_1)$ . Ainsi, on a bien le schéma de la figure 4 (droite). Le triplet  $m^0 = \{1, 2, 3\}$  donne les probabilités de la figure 4 (gauche) avec les mêmes labels des couches cachées. Ainsi, avec le même schéma de masquage des couches cachées, on peut obtenir différentes probabilités conditionnelles.

On peut bien entendu formaliser toutes les conditions de génération des labels mentionnées ci-dessus. Notons que les sorties notées  $s = \{1, 2, 3\}$  satisfont bien à schéma autoregressif, car  $s_i$  ne dépend que des entrées dont les labels sont strictement inférieurs: ex.  $s = 3$  ne dépend que de  $e = 1$  ( $x_3$ ) et  $e = 2$  ( $x_1$ ) d'où  $p(x_2|x_1, x_3)$ . Reste à générer les labels des neurones des couches cachées. Pour obtenir  $(m^\ell(k))_{k \leq K^\ell}$ , on procède en tirant uniformément des entiers dans l'intervalle  $[\min_{k \leq K^{\ell-1}} m^{\ell-1}(k) .. d - 1]$  (et non  $[1 .. d - 1]$ ) afin de ne pas avoir de neurones non connectés de la couche  $\ell$ .

Maintenant, il faut voir ces opérations de connexion/déconnexion comme une *régularisation* identique dans l'esprit à celle qui est à l'œuvre dans l'opération de **Dropout** (Srivastava et al., 2014). Mais qui dit régularisation, dit possibilité d'underfitting, donc cela impacte la stratégie d'usage des différents masques au cour de l'entraînement. Il y a donc des détails d'implémentations qu'il serait excessif d'exposer ici.

#### 4.4.2 Applications du masquage MADE: modèles MAF-like et IAF-like

Le schéma de masquage introduit à la section précédente a été appliqué au cas où l'on utilise de base un Auto-Encoder, et donc **la modélisation des pdf par MADE en tant que telle n'est pas basée sur un Normalizing Flow**. Néanmoins, la technique de

masquage a été réutilisée. En effet, on constate qu'au lieu de considérer des probabilités conditionnelles, il est tout au possible de considérer d'une manière générique, les fonctions  $C_i[z_{1:i-1}] = h_i$  comme sorties du réseau. Le système de masquage permet de contraindre  $h_i$  à ne dépendre que des entrées  $z_i$  étiquetées dans  $\{1, \dots, i-1\}$ . *En fait toute sortie respectant un schéma autoregressif est envisageable d'être implémentée de la sorte.* Ainsi, considérant le modèle autoregressif de la référence (Papamakarios et al., 2017) présenté à la section 4.3.1, les fonctions  $\mu_i[x_{1:i-1}]$  et  $\alpha_i[x_{1:i-1}]$  (Eq. 22) sont implémentées par un réseau MLP auquel on adjoint l'algorithme MADE de masquage. **C'est ce qu'on appelle un Masked Autoregressive Flow (aka MAF).**

Donc, une fois mise en place l'architecture du réseau et le masquage, pour  $Z = (z_i)_{i \leq d}$  donné, on peut calculer en 1 passe tous les  $(h_i)_{i \leq d}$  et calculer en parallèle tous les  $(x_i)_{i \leq d} = X$  (Eq. 13). **Avec les GPUs ce mode "forward" de génération de  $X$  est donc très efficace.** Par contre, le mode "inverse" exposé par les relations 14 pose problème. En principe,  $z_1$  dépend de  $h_1$  qui peut être connu à l'avance, mais  $z_2$  dépend de  $h_2$  qui dépend de  $z_1$ , et ainsi de suite, pour connaître  $z_i$ , il faut déjà avoir calculé  $z_{1:i-1}$  pour obtenir  $h_i$ . **On peut mettre en place un schéma itératif qui garantie l'inversion, mais en tout état de cause il requière  $d$  opérations séquentielles** donc le passage à l'échelle peut être compromis dans certain cas comme le processing d'images ou de flux video/audio. Une alternative à la recherche d'une inversion exacte est de procéder à une recherche par descente de gradient, ou bien par recherche de la solution  $Z$  telle que étant donné  $X$ ,  $X = T(Z)$ , en procédant par itération  $Z^0 = X$  puis<sup>9</sup> pour  $0 < \lambda < 2$

$$Z^k = Z^{k-1} - \lambda \text{diag}[J_T^{-1}(Z^{k-1})](T(Z^{k-1}) - X) \quad (33)$$

Comme on ne prend que des transformations bijectives, le jacobien est non singulier. Maintenant, la convergence n'est pas garantie en général mais localement seulement. A part ce caveat la méthode peut être plus rapide que l'inversion exacte en grande dimension.

Un modèle similaire à MAF procède selon la formulation d'une transformation affine

$$X = U \odot e^\alpha + \mu \quad U \sim \mathcal{N}(0, \mathbf{1}_d) \quad \text{avec} \quad \forall i \leq d \begin{cases} \mu_i &= \mu_i[u_{1:i-1}] \\ \alpha_i &= \alpha_i[u_{1:i-1}] \end{cases} \quad (\text{IAF}) \quad (34)$$

---

9.  $\text{diag}[A^{-1}]$  est une matrice diagonale obtenue par extraction des éléments diagonaux de l'inverse de  $A$ .

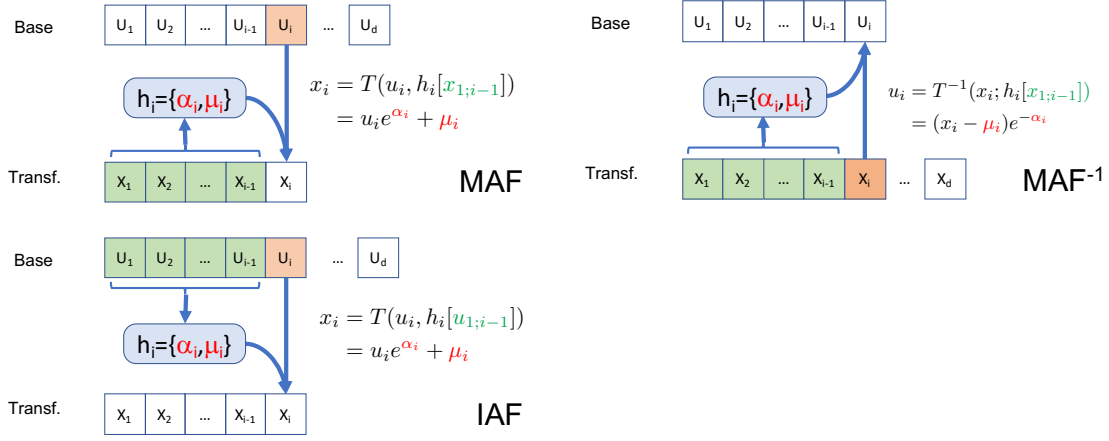


FIGURE 5 – Schémas de processing des méthodes MAF (Eq. 24), MAF inverse (Eq. 26) et IAF (Eq. 34). Les variables  $(u_i)_{i \leq d} = U$  sont issues d'une distribution dite de *base* qualifiée de *bruit non structuré*, tandis que les variables  $(x_i)_{i \leq d} = X$  sont issues d'une distribution dite *transformée* qualifiée parfois de *bruit structuré*. Le schéma  $MAF^{-1}$  est utilisé pour évaluer la pdf d'échantillons externes  $X^{(k)} = (x_i^{(k)})_{i \leq d}$  afin de procéder à la détermination des paramètres  $h_i[x_{1:i-1}]$ . Si par la suite on veut générer de nouveaux échantillons (via  $MAF$ ) cela coûte  $d$  opérations séquentielles pour générer un nouvel  $X$ . Le schéma  $IAF$  permet de calculer la *pdf* d'échantillon  $X$  générer au vol à l'aide des variables de base, ce qui est utilisé par exemple par la méthode SVI.

C'est un schéma autorégressif, mais différent de celui donné par Eq. 24, car ici  $(\mu_i, \alpha_i)$  **dépendent des valeurs antérieures des tirages des composantes  $u_{j < i}$  et non des  $x_{j < i}$** . On remarque la similitude avec l'équation 26 de l'inversion de MAF, cf. le membre de droite ne dépend pas des éléments du membre de gauche, d'où l'appellation d'**Inverse Autoregressive Flow** introduit dans la référence (Kingma et al., 2016).

Les motivations pour utiliser soit MAF soit IAF sont reliées à la discussion générale sur les NF (Sec. 2) et les considérations sur la facilité ou non de procéder aux modes "forward"/"inverse" exposées ci-dessus (voir figure 5):

**MAF inverse:** Si on fournit  $X$  alors en utilisant les relations 26 ( $MAF^{-1}$ ), il est simple de générer en 1 passe les composantes de  $U$ , et ainsi en utilisant l'expression 29 on peut directement obtenir  $p(X)$ . Faire de l'échantillonnage par contre nécessite  $d$  opérations séquentielles pour produire successivement  $x_1, x_2$ , etc. par la transformation MAF même une fois l'entraînement fait (c'est-à-dire que les paramètres du réseau dense permettant le calcul des  $h_i$  sont déterminés).

**IAF:** C'est l'inverse, générer un nouvel échantillon  $X$  une fois l'entraînement effectué, se fait simplement à l'aide du tirage de  $U$  et l'application des relations 34. On peut également calculer facilement la *pdf* d'un nouvel échantillon. Par contre, si l'on

fournit  $X$ , connaître  $p(X)$  demande une inversion nécessitant là encore  $d$  opérations séquentielles.

Le schéma IAF est utilisé (Kingma et al., 2016) comme ingrédient possible de la méthode d'*inférence variationnelle stochastique* (aka SVI). En effet, il nous suffit de procéder à la génération de nouveaux échantillons  $X$  à partir de tirage de  $U$ , et au calcul de la *pdf* de ces nouveaux échantillons. Le modèle  $\text{MAF}^{-1}$  quand à lui est adapté à *l'estimation de pdf*.

Notons que l'on peut allier les performances optimales de  $\text{MAF}^{-1}$  et IAF dans un schéma en deux étapes: la première apprend une *pdf*  $p_t(X)$  à partir d'une base de données d'échantillons  $X_t$  via le schéma  $\text{MAF}^{-1}$  (étape dite du *Teacher*); la seconde (dite du *Student*) procède au tirage à partir du bruit  $U$  de nouveaux échantillons  $X_s$  par un schéma IAF dont on calcule  $p_s(X_s)$  ainsi que la comparaison avec  $p_t(X_s)$ . A la fin du processus d'apprentissage du Student (minimisation de la distance  $D_{KL}(p_s||p_t)$ ), la *pdf*  $p_s(X)$  s'accorde avec celle du Teacher  $p_t(X)$ . Ce schéma a été mis en œuvre par l'équipe DeepMind pour synthétiser en temps réel des échantillons audios (van den Oord et al., 2017) [WaveNet]<sup>10</sup>.

On peut généraliser "facilement" les schémas MAF (Eqs. 22,26) et IAF (Eq. 34) en changeant la transformation linéaire reliant  $x_i$  et  $u_i$  par un autre type de bijector comme ceux exposés à la section 4.3. Par exemple, les auteurs de la référence (Huang et al., 2018) (NAF) ont modifié le schéma IAF en utilisant un MLP de paramètres  $\phi$  tel que

$$x_i = \text{MLP}(u_i, \phi[u_{1:i-1}]) \quad (\text{NAF}) \quad (35)$$

où  $\phi[u_{1:i-1}]$  est le résultat d'un conditioner à masquage MADE. On peut tout aussi bien en faire de même pour étendre le schéma MAF. De même au lieu d'un MLP, on peut tout aussi bien utiliser les Spline Flows (Sec. 4.3.3).

#### 4.4.3 Alternative au masquage: *splitting* (Real NVP)

Le masquage MADE décrit dans les sections précédentes est une opération asymétrique par essence à cause de la dépendance des fonctions  $h_i[x_{1:i-1}]$ , c'est pour cette raison qu'a été mise au point une méthode alternative dite de **split unique** (voir la généralisation

---

10. <https://www.deepmind.com/research/highlighted-research/wavenet>

à la section suivante). Cette méthode a été mise en place originellement sous le vocable de "*coupling layer*" dans la référence (Dinh et al., 2014) puis étendue dans (Dinh et al., 2016) avec une transformation affine du type de celle utilisée dans le schéma MAF. Nous verrons que nous pourrions unifier ces méthodes, mais voyons de quoi il retourne.

La méthode est illustrée sur la figure 6. Le principe est de fixer un entier  $s$  (typiquement  $d/2$ ) et de procéder comme suit pour la méthode forward:

$$\begin{cases} \forall i \leq s, & x_i = u_i \\ \forall i > s, & x_i = T(u_i; h_i[u_{1:s}]) \\ & = u_i e^{\alpha_i} + \mu_i \end{cases} \quad (\text{NVP}) \quad (36)$$

On remarque que ce schéma ressemble à IAF (Fig. 5) à la **différence essentielle que les paramètres pour calculer les  $x_i$  ( $i > s$ ) ne dépendent exclusivement que des variables  $(u_i)_{i \leq s}$**  à travers le calcul des paramètres  $h_i$ , et que les  $s$  premières variables  $x_i$  sont également directement calculable à partir des mêmes  $(u_i)_{i \leq s}$ . Une fois obtenus les  $(u_i)_{i \leq s}$ , on peut procéder à un calcul en parallèle.

L'inversion est immédiate:

$$\begin{cases} \forall i \leq s, & u_i = x_i \\ \forall i > s, & u_i = T^{-1}(x_i; h_i[u_{1:s}]) \\ & = (x_i - \mu_i) e^{-\alpha_i} \end{cases} \quad (\text{NVP}^{-1}) \quad (37)$$

Elle ressemble au schéma  $\text{MAF}^{-1}$  car en fait c'est la donnée des  $(x_i)_{i \leq s}$  qui permet non seulement de calculer les  $(u_i)_{i \leq s}$ , mais aussi les  $(h_i)_{i > s}$  et donc les  $(u_i)_{i < s}$ . Là encore on peut opérer en parallèle une fois obtenus les  $(u_i)_{i < s}$  à partir des échantillons  $(x_i)_{i \leq s}$ .

La figure 6 illustre les opérations Eqs. 36,37 à mettre en regard de la figure 5. Tout comme IAF le schéma NVP est utilisé pour générer de nouveaux échantillons et en calculer la *pdf*; tandis que le schéma  $\text{NVP}^{-1}$  est utilisé tout comme  $\text{MAF}^{-1}$  pour d'estimation de la *pdf* d'échantillons préexistant comme lors d'un entraînement. Les deux opérations s'effectuent avec les mêmes paramètres  $h[u_{i:s}]$ .

Ceci dit étant donné la non transformation des variables d'indice  $i \leq s$ , ce schéma NVP n'est pas suffisant à générer des distributions complexes quand bien même la trans-

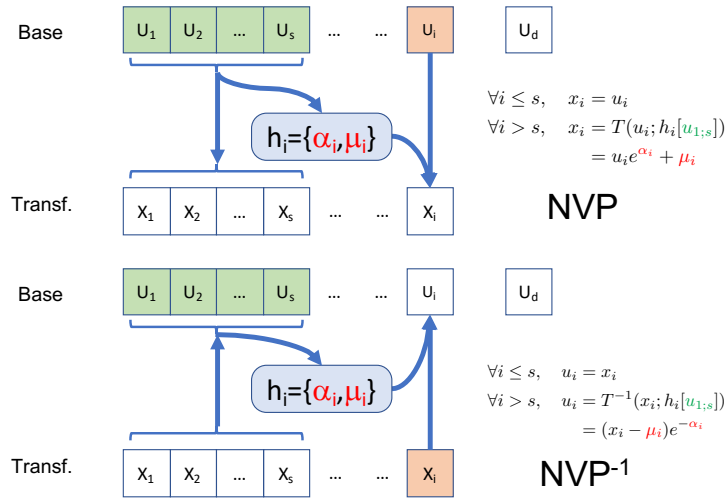


FIGURE 6 – Sur le modèle de la figure 5 sont illustrées les opérations Eqs. 36,37 du schéma dit NVP.

formation  $T$  soit "expressive". On a besoin de connecter plusieurs couches de couplage entre lesquelles, et on procède à **une permutations des variables** pour que toutes soient affectées par une transformation non triviale.

#### 4.4.4 Généralisation: *multiple splits*

Le schéma NVP<sup>-1</sup> décrit dans la section précédente est basé sur le duo suivant:

1. on pratique 1 unique découpage (split) de l'ensemble  $\llbracket 1..d \rrbracket$  ( $\llbracket 1..s \rrbracket \cup \llbracket s+1..d \rrbracket$ )
2. les paramètres  $h_i$  de la transformation non triviale invoquée (ie.  $T^{-1}$ ) ne dépendent que des variables  $u_{1:s}$ .

Une généralisation naturelle illustrée sur la figure 7 consiste:

1. à pratiquer un découpage de  $\llbracket 1..d \rrbracket$  en  $K$  sous-intervalles notés avec des bulles numérotées (1, 2, 3, etc);
2. les paramètres  $h^p$  servant à calculer les variables du sous-intervalle  $p+1$  dépendent des variables de tous les sous-intervalles  $\llbracket 1..p \rrbracket$  ( $p \leq K$ ). Pour  $p = 1$ , on peut procéder comme pour NVP<sup>-1</sup> en utilisant l'identité comme transformation triviale.

On peut tout aussi bien appliquer cette généralisation au schéma NVP. Dans ce contexte, on note alors que **les schémas MAF/IAF (et leurs inverses) peuvent être considérés**

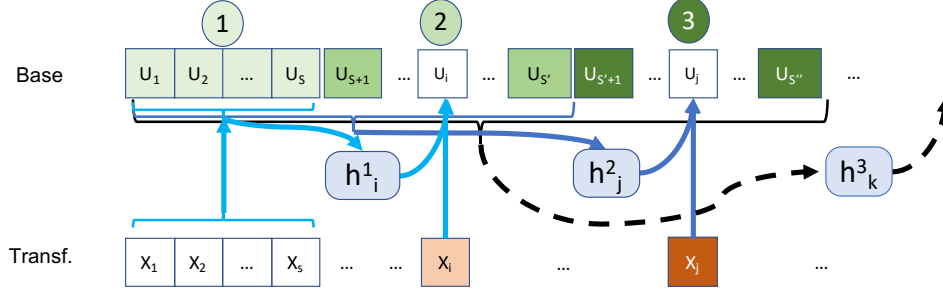


FIGURE 7 – Exemple de la généralisation du schéma  $\text{NVP}^{-1}$  en considérant plusieurs découpages (*splitting*) de l'ensemble  $\llbracket 1 \dots d \rrbracket$  (voir texte).

comme des **splitting maximaux** pour lesquels  $K = d$  laissant 1 variable unique par sous-intervalle. On peut également considérer l'enchaînement de plusieurs "couches" de splitting tout en pensant à opérer une permutation des variables.

#### 4.5 Quelques remarques sur les Deep NF

L'enchaînement des transformations évoqué dans le principe général (Sec. 2) fixant le cadre de l'élaboration des normalizing flows est mis en œuvre à travers l'enchaînement des couches auto-regressives détaillées dans les sections précédentes. On peut alors élaborer des schémas parallèles aux Deep Neural Networks (DNN) que l'on peut qualifier de Deep Normalizing Flows (DNF).

De plus, si dans les sections ci-dessus  $X = (x_i)_{i \leq d} \in \mathbb{R}^d$ , il n'y a pas de mention de la structuration interne, on sait que les couches convolutionnelles ont boosté le traitement d'images et pas que. Dans le domaine de l'inférence de *pdf* et la génération de nouvelles images, on a trouvé la façon de mettre en place des NF adaptés avec une extension des techniques de masquages associées aux convolutions et changement d'échelles comme dans la référence (Dinh et al., 2016). De même de nouveaux bijectors ont vu le jour comme ceux associés aux couches de normalisations *BatchNorm* et *Actnorm* (Dinh et al., 2016; Kingma and Dhariwal, 2018) qui rendent certaines architectures plus stables (mais à manier avec précaution). Il est également important de procéder à une permutation des variables entre chaque couche (voir Secs. 4.4.1, 4.4.4).

Maintenant, il faut garder à l'esprit que les paramètres de ces structures de DNF, sont obtenus par une *optimisation* de la fonction de coût, *negative log-likelihood* (Eq. 6).



Or, tout comme pour les DNN, on utilise la mise en min-batch des données et l'on fait appel à des minimiseurs de type SGD, Adam, etc avec leurs lots de paramétrages à tuner. De même il faut mettre en place des procédures de test/validation. Et in fine, rappelons que l'architecture est une donnée *a priori* et faute de cadre théorique complet général, il n'y a que l'expérimentation qui valide telle ou telle architecture dans telle ou telle application.

Ceci étant dit la section suivante présente quelques exercices pour se faire la main mais sans grande prétention.

## 5. Quelques mises en pratique

Dans cette section, je vais donner quelques exemples de mise en œuvre mais il faut avoir à l'esprit que les références déjà mentionnées donnent des réalisations bien plus élaborées car elles sont face à des situations plus complexes. Donc, il faut prendre la suite comme une mise en bouche. Certains cas sont déjà malgré tout porteurs de quelques enseignements. Pour procéder à la mise en code, j'ai utilisé tout d'abord la librairie TensorFlow<sup>11</sup> fournissant un ensemble cohérent. Cependant, la version 2 mise en production en Sept. 2019 n'intègre pas le bijector NVP, ainsi j'ai utilisé la librairie `distrax` de DeepMind écrite en JAX ce qui donne par la même une occasion de voir une autre implémentation. Les exercices sont suffisamment légers pour tourner sur CPU.


### 5.1 Simple bijector polynomial en 1D (TF)

Prenons par exemple le cas en 1D d'un NF basé sur la transformation non-linéaire suivante<sup>12</sup>

$$z = T(x) = (ax + b)^3 \Leftrightarrow x = T^{-1}(z) = (z^{1/3} - b)/a \quad (38)$$

En fixant  $(a, b)$  et  $z \sim \mathcal{N}(0, 1)$ , on peut générer des échantillons de training  $(x_i^{train})_i$  et de test/validation  $(x_i^{test})$  pour entraîner le NF, en maximisant la vraisemblance comme vu à

11. [https://www.tensorflow.org/probability/api\\_docs/python/tfp](https://www.tensorflow.org/probability/api_docs/python/tfp)

12. En pratique on peut implémenter  $T(x)$  et utiliser une méthode "Invert" pour instancier  $T^{-1}$ . Le notebook  permet de reproduire cet exercice.

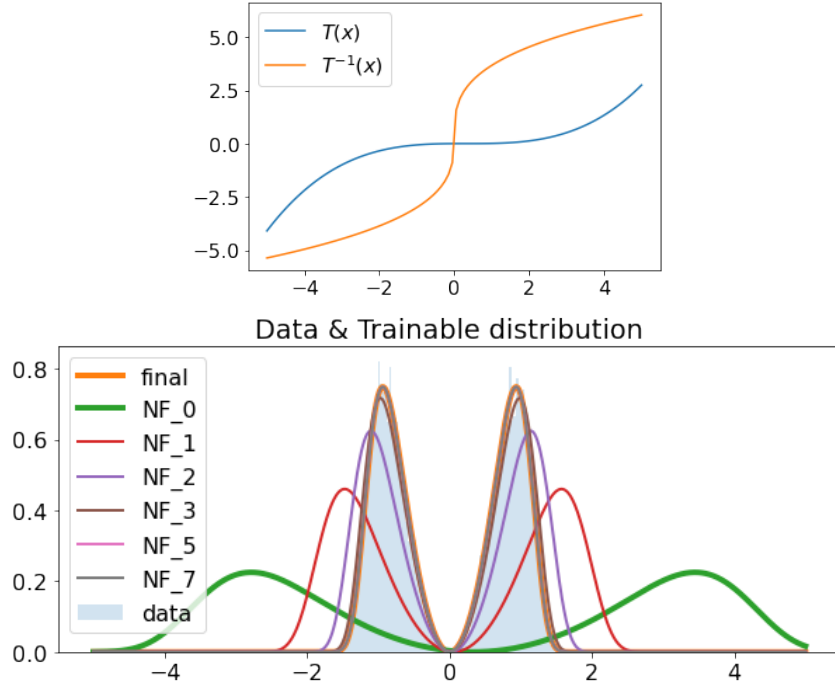


FIGURE 8 – En haut: les graphes du flow Eq. 38 pour  $(a, b) = (1, 0)$ . En bas: évolution des *pdf* au fur et à mesure de l'entraînement.

la section 2. Pour cet exemple, j'ai utilisé TensorFlow Probability sans le backend JAX. Le résultat est donné sur la figure 8. On constate une bonne convergence.

Cependant, on remarque que ce type de NF n'est pas très "expressif", les types de distributions accessibles depuis une loi normale, sont nécessairement symétriques et l'on aurait vite des désillusions. Par exemple, considérons le cas d'un mélange de deux gaussiennes de même sigma égal à 0.4, de poids respectifs (0.6, 0.4) dont la première est centrée en 2.3 et la seconde en  $-0.8$ . La figure 9 révèle alors la défaillance à décrire le mélange via la transformation d'une loi normale par ce NF. Ainsi, il n'y a pas de miracle, tout comme vouloir interpoler une fonction avec un polynôme de bas degré ou même un réseau de neurones trop peu profond et large, peut s'avérer un choix trop étriqué, il en est de même pour les NF. Il faut que la distribution cible soit dans l'espace accessible à la famille des NF considérés en faisant varier ses paramètres  $((a, b)$  dans le cas Eq. 38).

Une façon de vérifier si le NF a bien été entraîné est basé sur le fait que les échantillons  $(z_i)_i$  obtenus par l'application de  $T$  sur le lot  $(x^{test})_i$  (plutôt un lot de validation

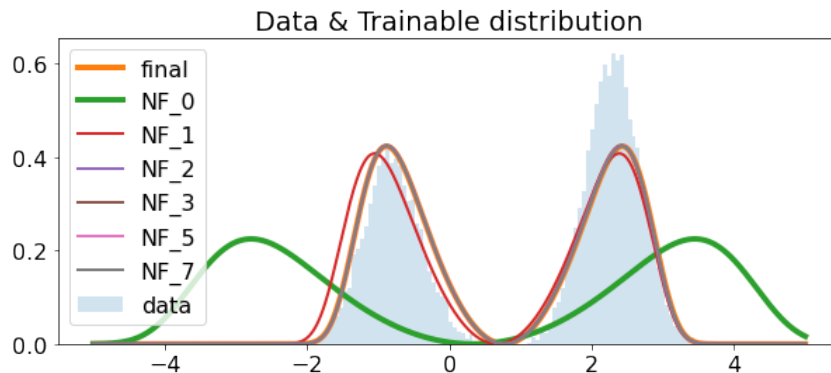


FIGURE 9 – Echec du NF (Fig. 8, Eq. 38) pour décrire le mélange asymétrique de deux loi normales.

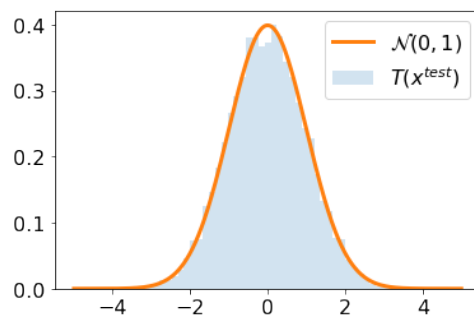


FIGURE 10 – Les échantillons  $z$  issus de la transformation  $T$  une fois entraînée sont bien issus d'une loi normale si tout se passe bien. Ce qui est le cas en prenant le résultat de la figure 8.

d'ailleurs) est bien conforme à une loi normale. Par exemple, avec le cas de la figure 8 le résultat est parfait (Fig. 10). Il faut donc avoir en tête qu'en principe on a *in fine* un moyen de contrôler que le NF a été entraîné avec succès.

## 5.2 Cas en 2D:

### 5.2.1 ex. MADE vs MAF avec TensorFlow

Dans la référence [Papamakarios et al. \(2017\)](#) les auteurs proposent comme exercice simple le cas suivant:

$$p(x_1, x_2) = \mathcal{N}(x_2; \mu = 0, \sigma = 2) \mathcal{N}(x_1; \mu = x_2^2/4, \sigma = 1) \quad (39)$$

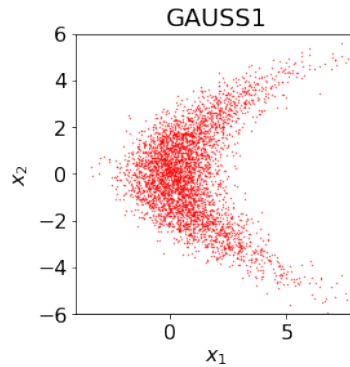


FIGURE 11 – Les échantillons issus de la *pdf* dont l'expression est donnée par Eq. 39.

dont quelques échantillons issus de cette *pdf* sont présentés dans la figure 11. On peut tenter de procéder à un entraînement d'une couche autogressive avec le masquage MADE (voir le notebook [4b](#)).

```
flow_bijector = tfb.MaskedAutoregressiveFlow(name='MADE',
      shift_and_log_scale_fn=tfb.AutoregressiveNetwork(params=2,
      hidden_units=[512, 512], activation='relu'),
  ))
```

Je ne vais pas rentrer dans le détail de l'implémentation, il vaut mieux se référer à la documentation. Cependant, l'objet `AutoregressiveNetwork` (ARN) fournit un réseau dense à 2 couches cachées de 512 neurones chacune et fournissant 2 sorties par entrées, en l'occurrence les 2 facteurs  $(\mu, \alpha)$  pour opérer le shift et le scaling (Sec. 4.3.1). Notons que l'implémentation TF de l'ARN fait en sorte que la sortie  $i$  ne dépend que des entrées  $1; i - 1$ , en cela on est dans la configuration  $\text{MAF}^{-1}$  (Fig. 5) ce qui peut dérouter.

Une fois le flow initialisé, on procède à la description de la distribution de base (une loi normale en général) ainsi que sa transformation par le flow:

```
base_dist = tfd.MultivariateNormalDiag(loc=tf.zeros([2], DTYPE), name='base dist')
trans_dist = tfd.TransformedDistribution(
    distribution=base_dist,
    bijector=flow_bijector)
```

L'entraînement se fait via l'API de Keras en construisant un modèle qui prend en entrée les échantillons  $X^i = (x_1^i, x_2^i)$  ( $i \leq N_{train}$ ) et en calcule le log de la probabilité (`log_prob`):

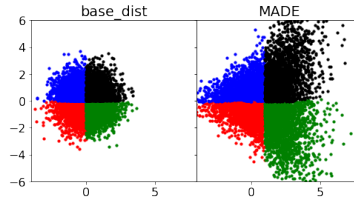


FIGURE 12 – Transformation des échantillons tirés selon une loi normale 2D par la couche MADE une fois l’entraînement fait. Les quadrants de couleur montrent l’évolution des différents lots de données de la distribution de base. Si tout se passait bien on devrait obtenir une distribution identique à celle de la figure 11.

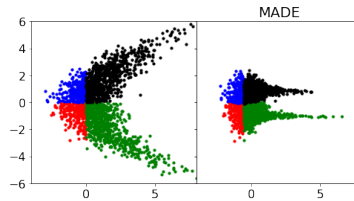


FIGURE 13 – Transformation des échantillons tirés selon une loi Eq. 39 par la couche MADE optimisée. On devrait obtenir une loi normale...

```
x_ = tfkl.Input(shape=(2,), dtype=DTYPE)
log_prob_ = trans_dist.log_prob(x_)
model = tfk.Model(x_, log_prob_)
```

L’optimisation de la moyenne des `log_prob` se fait avec Adam en précisant le learning rate, le nombre d’échantillons par batch, le nombre d’epochs, etc. Une fois l’entraînement fait, on peut comparer déjà visuellement, la distribution d’échantillons tirés selon une loi normale transformée par le flow optimisé. Si le flow est "expressif" et l’entraînement bien mené, alors la distribution doit être celle de la figure 11. Le résultat est donné sur la figure 12 et manifestement il y a un problème. D’ailleurs si on essaye de faire la transformation inverse d’échantillons de test tirés selon la vraie *pdf* Eq. 39 au lieu de trouver une loi normale, on trouve quelque chose de nettement différent montré sur la figure 13.

C’est en fait le manque d’expressivité d’une seule couche MADE qui est en cause. Pour s’en convaincre, on construit un DNF (Sec. 4.5) par enchainement de bijectors comme suit:

```
bijectors = []
for i in range(num_bijectors):
    bijectors.append(tfb.MaskedAutoregressiveFlow( name = 'MAF%d' %i,
```

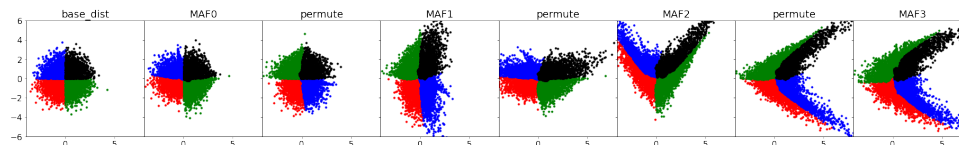


FIGURE 14 – Transformation des échantillons tirés selon une loi normale 2D par l’enchaînement de plusieurs couche MAF et de permutation une fois l’entraînement fait. En comparant avec la figure 12, on se doute que le NF a bien été entraîné.

```
shift_and_log_scale_fn=tfb.AutoregressiveNetwork(
    params=2, hidden_units=[128, 128], activation='relu'))
bijectors.append(tfb.Permute(permutation=[1, 0]))

# Discard the last Permute layer.
flow_bijector = tfb.Chain(list(reversed(bijectors[:-1])))
```


Notons qu’il faut penser à permuter les entrées à chaque couche, et la liste entrant dans `tfb.Chain` doit être inversée pour que la première transformation soit la première à être appliquée sur les échantillons  $X$ . Une fois l’entraînement fait la transformation d’échantillons issus d’une loi normale est donné sur la figure 14. Dans ce cas, à première vue, le DNF a bien capturé la densité de probabilité 39. Au passage, les permutations après chaque couche MAF sont indispensables, vous pouvez en faire l’expérience...

Maintenant, il n’est pas très difficile de se convaincre que la transformation inverse d’échantillons tirés selon Eq. 39 est meilleure que pour 1 couche MADE. On peut quantifier cela au-delà du simple aspect visuel. Pour se faire, on peut utiliser le test de Henze-Zirkler (Ebner and Henze, 2020) disponible dans la librairie `pingouin`<sup>13</sup> (Vallat, 2018). Le test à 95%CL mis en place dans le notebook montre alors une bonne adéquation avec une loi normale.

### 5.2.2 ex. MAF vs Real NVP avec Distrax

L’exercice de la section précédente peut être répété avec d’autres types de distributions, mais voyons plutôt une implémentation de NVP et MAF en JAX. En fait j’utilise la partie de TensorFlow qui traite des probabilités avec un backend en JAX, la librai-

13. <https://pingouin-stats.org/>

rie légère Haiku<sup>14</sup> qui traite des réseaux de neurones, et Distrax<sup>15</sup> qui prend en charge quelques bijectors de TF réimplémentés en JAX. Le notebook  permet de rejouer ce qui suit.

L'architecture du code reflète le caractère fonctionnel de JAX. Pour définir un NF, il faut:

1. une fonction qui crée un le *conditioner* qui donne les paramètres de chaque *bijector* interne, il s'agit en général d'un MLP
2. une fonction qui crée un *bijector* interne, avec pour NVP un *bijector* de type affine (Sec. 4.3.1) (*ScalarAffine*) et pour MAF un *bijector* de type rational-quadratic-splines aka RQS (Sec. 4.3.3) (*RationalQuadraticSpline*)
3. enfin pour chaque couche du *bijector* global il faut appliqué un couplage entre couches ou masquage (*SplitCoupling*/*MaskedCoupling*, la seconde méthode est plus générale):

```
layer = distrax.MaskedCoupling(
    mask=mask,
    bijector=bijector_fn,
    conditioner=make_conditioner_nvp(
        event_shape=event_shape,
        hidden_sizes=hidden_sizes,
        num_bijector_params=2,
    ),
)
layers.append(layer)
# Flip mask after each layer
mask = jnp.logical_not(mask)
```

Ensuite l'entraînement utilise le minimizer Adam de la librairie Optax rencontrée dans les notes sur les Tutos JAX<sup>16</sup>. La figure 15 donne quelques exemples de distributions ainsi que les échantillons tirés une fois les *bijectors* entraînés. On voit une (petite) différence en faveur de MAF (RQS).

14. <https://github.com/deepmind/dm-haiku>

15. <https://github.com/deepmind/distrax>

16. <https://github.com/jecampagne/JaxTutos>

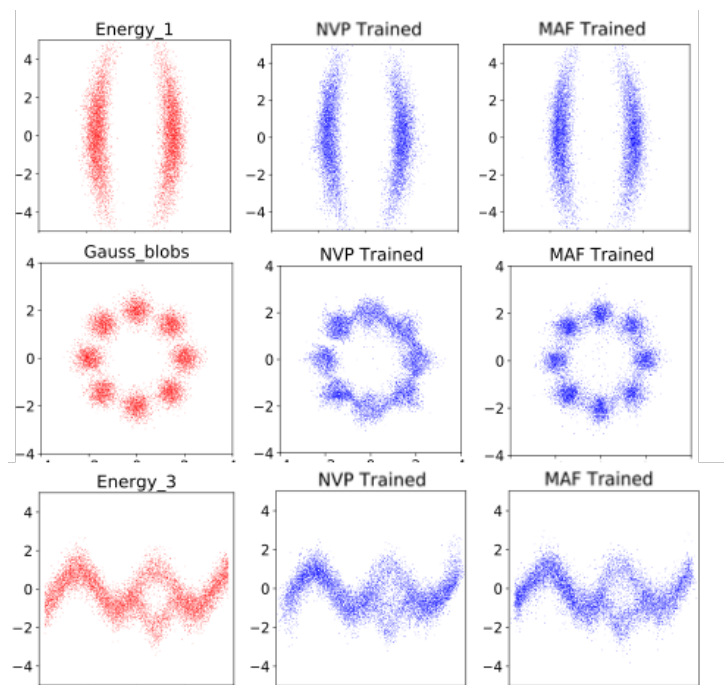


FIGURE 15



## 6. Résumer & perspectives

Dans cette note j'ai exposé la façon d'utiliser des *transformations bijectives* (Normalizing Flow) afin de modéliser/calculer/échantillonner des densités de probabilités  $p_x(X)$ . Les NF les plus expressifs sont des enchainements multi-couches à la manière des réseaux de neurones profonds. J'ai exposé comment entraîner de telles architectures sur des exemples simples.

La référence [Rouhiainen et al. \(2021\)](#) quand à elle explore l'usage de NF en Cosmologie pour l'estimation de densité et la génération de champs "near-Gaussian". Des généralisations au traitement d'images sont exposées dans les références mentionnées dans cette note. Pour les adeptes de géométries différentielles on peut citer la mise en œuvre de NF sur des tores et sphères de dimensions quelconque comme dans la référence [Jimenez Rezende et al. \(2020\)](#).

Il y a bien d'autres généralisations possibles, cependant une nous intéresse particulièrement: en effet, rien n'empêche de considérer  $p(X|Y)$  ([Papamakarios et al., 2018](#)) et de l'approximer avec une représentation paramétrique  $q_\phi(X|Y)$  obtenue par un NF. On a alors par exemple accès à une estimation d'un likelihood  $p(X|\theta)$  uniquement en disposant d'un générateur de  $\{X^i, \theta^i\}$ , ce qui ouvre la boîte des *likelihood-free inference*.

## A. Transformation d'une distribution en une autre par NF

Tout d'abord faisons remarquer que si  $X = (x_1, \dots, x_d) \in \mathbb{R}^d$  est une v.a de densité  $p_x(\mathbf{x}) > 0$ , on peut exprimer cette dernière selon

$$p_x(x_1, \dots, x_d) = p_x(x_1)p_x(x_2|x_1)p_x(x_3|x_1, x_2) \dots p_x(x_d|x_1, x_2, \dots, x_{d-1}) = \prod_{i=1}^d p_x(x_i|x_{1;i-1}) \quad (40)$$

On peut se convaincre de cette formule en faisant remarquer que si on note  $X_i$  l'ensemble dans lequel évolue  $x_i$

$$p_x(x_2|x_1) = \frac{p_x(X_2 \cap X_1)}{p_x(X_1)}; \quad p_x(x_3|x_1, x_2) = \frac{p_x(X_3 \cap X_2 \cap X_1)}{p_x(X_2 \cap X_1)}; \quad \text{etc}$$

donc

$$\begin{aligned} p_x(x_1, \dots, x_d) &= p_x(X_1 \cap \dots \cap X_d) \\ &= \frac{p_x(X_1 \cap \dots \cap X_d)}{p_x(X_1 \cap \dots \cap X_{d-1})} \times \frac{p_x(X_1 \cap \dots \cap X_{d-1})}{p_x(X_1 \cap \dots \cap X_{d-2})} \times \dots \times \frac{p_x(X_2 \cap X_1)}{p_x(X_1)} \times p_x(X_1) \\ &= p_x(x_d|x_1, x_2, \dots, x_{d-1}) \times p_x(x_{d-1}|x_1, x_2, \dots, x_{d-2}) \times \dots \times p_x(x_2|x_1) \times p_x(x_1) \end{aligned}$$

Ensuite pour tout  $i$ , soit la variable aléatoire  $z_i$  définie par

$$z_i = \int_{-\infty}^{x_i} p_x(u|x_{1;i-1}) du = F_i(x_i; x_{1;i-1}) \quad (41)$$

$F_i(x_i; x_{1;i-1})$  n'est autre que la fonction de répartition de  $p_x(\cdot; x_{1;i-1})$ . C'est une fonction de  $\mathbb{R}$  sur  $[0, 1]$ , différentiable et  $\partial_{x_i} F = p_x(x_i|x_{1;i-1}) > 0$  donc elle est inversible<sup>17</sup>. Un résultat connu de nos étudiants nous dit alors que  $z_i$  **suit une loi uniforme** sur  $[0, 1]$ , et donc  $Z \sim \mathcal{U}([0, 1]^d)$ .

Donc, toute pdf  $p_x$  peut être transformée en une loi uniforme via  $F$ , alors  $F^{-1}$  transforme en retour la loi uniforme en  $p_x$ . Ainsi, en utilisant la loi uniforme comme intermédiaire, on peut élaborer deux transformations qui combinées fait passer de  $p_x(x)$  à une autre  $p_u(u)$  et vice-versa.

---

17. En effet  $p_x(x) > 0$  par hypothèse donc il en est de même de toutes les densités de probabilités conditionnelles de l'expression 40)

## Références

- Dinh, L., Krueger, D., Bengio, Y., 2014. NICE: Non-linear Independent Components Estimation. arXiv e-prints , arXiv:1410.8516arXiv:1410.8516.
- Dinh, L., Sohl-Dickstein, J., Bengio, S., 2016. Density estimation using Real NVP. arXiv e-prints , arXiv:1605.08803arXiv:1605.08803.
- Durkan, C., Bekasov, A., Murray, I., Papamakarios, G., 2019. Neural Spline Flows. arXiv e-prints , arXiv:1906.04032arXiv:1906.04032.
- Ebner, B., Henze, N., 2020. Tests for multivariate normality – a critical review with emphasis on weighted  $L^2$ -statistics. arXiv e-prints , arXiv:2004.07332arXiv:2004.07332.
- Germain, M., Gregor, K., Murray, I., Larochelle, H., 2015. MADE: Masked Autoencoder for Distribution Estimation. arXiv e-prints , arXiv:1502.03509arXiv:1502.03509.
- Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y., 2014. Generative Adversarial Networks. arXiv e-prints , arXiv:1406.2661arXiv:1406.2661.
- Gregory, J.A., Delbourgo, R., 1982. Piecewise Rational Quadratic Interpolation to Monotonic Data. IMA Journal of Numerical Analysis 2, 123–130. URL: <https://doi.org/10.1093/imanum/2.2.123>, doi:10.1093/imanum/2.2.123, arXiv:<https://academic.oup.com/ima/jna/article-pdf/2/2/123/2267745/2-2-123.pdf>.
- Huang, C.W., Krueger, D., Lacoste, A., Courville, A., 2018. Neural Autoregressive Flows. arXiv e-prints , arXiv:1804.00779arXiv:1804.00779.
- Jimenez Rezende, D., Papamakarios, G., Racanière, S., Albergo, M.S., Kanwar, G., Shanan, P.E., Cranmer, K., 2020. Normalizing Flows on Tori and Spheres. arXiv e-prints , arXiv:2002.02428arXiv:2002.02428.
- Kingma, D.P., Dhariwal, P., 2018. Glow: Generative Flow with Invertible 1x1 Convolutions. arXiv e-prints , arXiv:1807.03039arXiv:1807.03039.
- Kingma, D.P., Salimans, T., Jozefowicz, R., Chen, X., Sutskever, I., Welling, M., 2016. Improving Variational Inference with Inverse Autoregressive Flow. arXiv e-prints , arXiv:1606.04934arXiv:1606.04934.

- Kingma, D.P., Welling, M., 2013. Auto-Encoding Variational Bayes. arXiv e-prints , arXiv:1312.6114arXiv:1312.6114.
- Kobyzev, I., Prince, S.J.D., Brubaker, M.A., 2019. Normalizing Flows: An Introduction and Review of Current Methods. arXiv e-prints , arXiv:1908.09257arXiv:1908.09257.
- Papamakarios, G., Nalisnick, E., Jimenez Rezende, D., Mohamed, S., Lakshminarayanan, B., 2019. Normalizing Flows for Probabilistic Modeling and Inference. arXiv e-prints , arXiv:1912.02762arXiv:1912.02762.
- Papamakarios, G., Pavlakou, T., Murray, I., 2017. Masked Autoregressive Flow for Density Estimation. arXiv e-prints , arXiv:1705.07057arXiv:1705.07057.
- Papamakarios, G., Sterratt, D.C., Murray, I., 2018. Sequential Neural Likelihood: Fast Likelihood-free Inference with Autoregressive Flows. arXiv e-prints , arXiv:1805.07226arXiv:1805.07226.
- Ramasinghe, S., Fernando, K., Khan, S., Barnes, N., 2021. Robust normalizing flows using Bernstein-type polynomials. arXiv e-prints , arXiv:2102.03509arXiv:2102.03509.
- Rouhiainen, A., Giri, U., Münchmeyer, M., 2021. Normalizing flows for random fields in cosmology. arXiv e-prints , arXiv:2105.12024arXiv:2105.12024.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* 15, 1929–1958. URL: <http://jmlr.org/papers/v15/srivastava14a.html>.
- Ulyanov, D., Vedaldi, A., Lempitsky, V., 2017. It Takes (Only) Two: Adversarial Generator-Encoder Networks. arXiv e-prints , arXiv:1704.02304arXiv:1704.02304.
- Uria, B., Côté, M.A., Gregor, K., Murray, I., Larochelle, H., 2016. Neural Autoregressive Distribution Estimation. arXiv e-prints , arXiv:1605.02226arXiv:1605.02226.
- Vallat, R., 2018. Pingouin: statistics in Python. *The Journal of Open Source Software* 3, 1026. doi:10.21105/joss.01026.
- van den Oord, A., Li, Y., Babuschkin, I., Simonyan, K., Vinyals, O., Kavukcuoglu, K., van den Driessche, G., Lockhart, E., Cobo, L.C., Stimberg, F., Casagrande, N., Grewe, D., Noury, S., Dieleman, S., Elsen, E., Kalchbrenner, N., Zen, H., Graves, A., King, H., Walters, T., Belov, D., Hassabis, D., 2017. Parallel WaveNet: Fast High-Fidelity Speech Synthesis. arXiv e-prints , arXiv:1711.10433arXiv:1711.10433.