Modèles de flux (Flow-based)

1 Petit rappel sur les flots

L'idée des modèles de flot est de composer un modèle via des fonctions inversibles. La différence principale avec les GANs est qu'il n'y a pas de discriminateur; par rapport aux VAEs, on n'utilise pas une approximation de l'inverse – mais l'inverse véritable, en utilisant le théorème du changement de variable qui permet de passer d'une densité de probabilité à une autre.

De manière plus précise, on modélise une succession de distributions $\mathbf{z_i} \sim p_i(\mathbf{z_i})$ où chaque $\mathbf{z_i}$ est obtenu comme une transformation f_i **inversible** de $\mathbf{z_{i-1}}$. En résumé, on a donc (voir figure 1)

$$\mathbf{z_i} \sim p_i(\mathbf{z_i}), \mathbf{z_{i+1}} = f_{i+1}(\mathbf{z_i}), \mathbf{z_i} = f_{i+1}^{-1}(\mathbf{z_{i+1}})$$

où f est le *décodeur* et f^{-1} l'*encodeur*.

Cela nous permet d'exprimer la distribution de z_i en fonction de z_{i+1} (ou inversement, mais c'est plus pratique dans ce sens), en utilisant la formule de changement de variable dans une intégrable multiple :

$$p_i(\mathbf{z_i}) = p_{i+1}(\mathbf{z_{i+1}}) \left| \det \frac{d\mathbf{z_{i+1}}}{d\mathbf{z_i}} \right| = p_{i+1}(\mathbf{z_{i+1}}) \left| \det \frac{df_{i+1}}{d\mathbf{z_i}} \right|$$

De manière symétrique, on a

$$p_{i+1}(\mathbf{z_{i+1}}) = p_i(\mathbf{z_i}) \left| \det \frac{d\mathbf{z_i}}{d\mathbf{z_{i+1}}} \right| = p_i(\mathbf{z_i}) \left| \det \frac{df_{i+1}^{-1}}{d\mathbf{z_{i+1}}} \right|$$

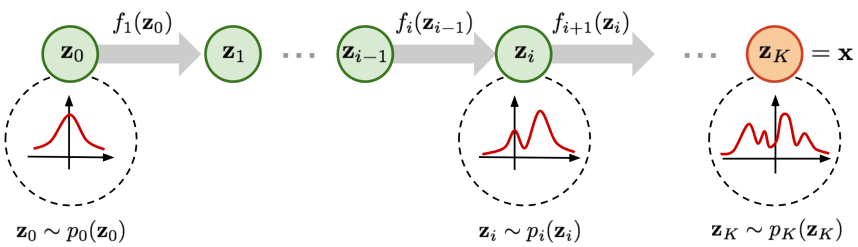


Figure 1: Flow (source Lil'Log)

À noter une propriété importante du déterminant pour les fonctions inversibles:

$$det\left(J_{f^{-1}}\right) = det\left(\frac{\partial f^{-1}}{\partial \mathbf{x}}\right) = det\left(\frac{\partial f}{\partial \mathbf{y}}\right)^{-1} = det\left(J_f\right)^{-1}$$

Cela montre qu'il faut prendre la forme la plus simple pour calculer les déterminants des Jacobiennes

On obtient donc par récurrence avec la variable observée $\mathbf{x} = \mathbf{z_K}$:

$$\begin{aligned}
\log p(\mathbf{x}) &= \log p(\mathbf{z_K}) \\
&= \log p_0(\mathbf{z_0}) - \sum_{i=0}^{K-1} \log \left| \det \frac{df_{i+1}}{d\mathbf{z_i}} \right| \\
&= \log p_0(f_1^{-1} \circ \ldots \circ f_K^{-1}(\mathbf{z_K})) - \sum_{i=0}^{K-1} \log \left| \det \frac{df_{i+1}}{d\mathbf{z_i}} \right| \\
&= \log p_0(f_1^{-1} \circ \ldots \circ f_K^{-1}(\mathbf{z_K})) + \sum_{i=0}^{K-1} \log \left| \det \frac{df_{i+1}^{-1}}{d\mathbf{z_i}} \right|
\end{aligned}$$

Il est donc possible de maximiser directement la vraisemblance des données, contrairement aux GANs (pas de dépendance à un discriminateur) et VAE (pas d'approximation ELBO), comme illustré sur la figure 2, mais le prix à payer est une moindre expressivité car il faut que les fonctions f_i :

- soient inversibles (pour calculer $\mathbf{z_0}$ lors de la maximisation de vraisemblance, et $\mathbf{x} = \mathbf{z_K}$ lors de la génération),
- aient une jacobienne dont le déterminant peut être calculé

Propriétés du déterminant Matrices triangulaires

Au sujet du déterminant, la plupart des modèles Flow s'arrangent pour que la jacobienne soit triangulaire, et dans ce cas le calcul du déterminant est très simple :

$$\left| \begin{pmatrix} d_1 & ? & \cdots & ? \\ 0 & d_2 & ? & \vdots \\ \vdots & \ddots & \ddots & ? \\ 0 & \cdots & 0 & d_n \end{pmatrix} \right| = \prod_{i=1}^{n} d_n$$

Produit de matrices

Si $\mathbf{A}$ et $\mathbf{B}$ sont deux matrices carrés d'ordre n ($= \in \mathbb{R}^{n \times n}$), alors

$$det\left(\mathbf{AB}\right) = det(\mathbf{A}) \times det(\mathbf{B})$$

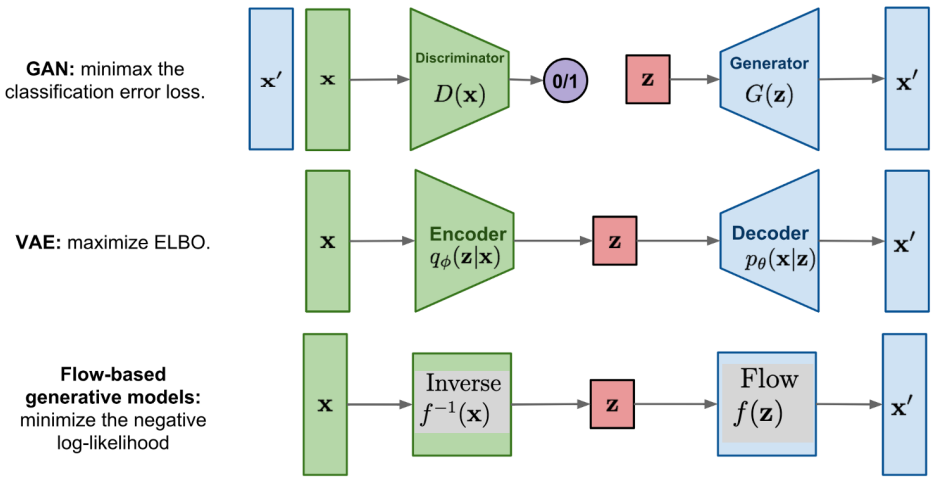


Figure 2: Commpairison GAN, VAE et Flow (source Lil'Log)

2 Modèle du code

Vous utiliserez le fichier `utils.py` qui contient:

- Une classe `FlowModule` générique qui définit deux fonctions, f (`decoder`) et f^{-1} (`encoder`). Cette classe contient également une méthode `check(x: torch.Tensor)` qui permet de vérifier que l'inverse est bien calculée : vous vous en servirez pour vérifier le module avant toute utilisation. Vous pouvez également en profiter pour vérifier que $\log|\det J_f|$ est bien calculé.
- Une classe `FlowSequential` qui ressemble à `torch.nn.Sequential` : elle permet d'enchaîner les appels aux décodeurs et encodeurs.
- Une classe `FlowModel` qui permet de définir un modèle de flow complètement, en intégrant le prior.

Dans l'ensemble du TP, vous définirez des classes dérivant de `FlowModule` (dérivant elle même de `torch.nn.Module`), qui définissent f (`decoder`) et f^{-1} (`encoder`). Le `encoder`($\mathbf{y}$) – et de manière similaire la fonction `decoder` – doit renvoyer deux tenseurs (n'oubliez pas que vous travaillez avec des batchs) :

1. La valeur $[f^{-1}(\mathbf{y})]$ (tableau à un élément);
2. Le logarithme du déterminant de la jacobienne $\log|\det J_{f^{-1}}|$.

L'optimisation se fait par descente de gradient.

3 Échauffement : Transformation affine

La première transformation est simple : il s'agit d'une transformation affine, permettant de changer la variance et la moyenne d'une variable aléatoire.

$$f(\mathbf{y}; \mathbf{s}, \mathbf{t}) = \mathbf{y} \odot \exp(\mathbf{s}) + \mathbf{t}$$

où $\mathbf{s}$ et $\mathbf{t}$ sont les paramètres, et $\odot$ est un produit terme à terme.
Apprenez la transformation de $\mathbf{z_{i0}} \sim \mathcal{N}(0, 1)$ à $\mathbf{x} = \mathbf{z_1} \sim \mathcal{N}(\mu, \sigma)$. Comparez les valeurs apprises.

Conseils Utilisez `torch.distributions` pour définir le prior. En particulier, pensez à utiliser Independent pour définir une normale multivariée avec covariance diagonale.

4 Glow (part I)

Les transformations affines ne permettent pas de faire grand chose; pour apprendre des distributions de probabilité plus complexes, nous allons implémenter le modèle Glow [1] qui repose sur trois opérateurs (ainsi qu'une série d'autres opérations que nous verrons par la suite):

ActNorm Une transformation affine comme celle de la section 3 avec les paramètres initialisés (au premier batch) de façon à obtenir une distribution centrée et réduite.

Affine Coupling Layer Une transformation affine de la moitié des valeurs, l'autre servant à paramétriser la transformation. Soit $\mathbf{x}$ la valeur actuelle de dimension $2 \times l$:

$$\begin{aligned} \mathbf{s} &= \mathrm{MLP}_{\mathrm{scale}}(\mathbf{x}_{1:l}) \\ \mathbf{t} &= \mathrm{MLP}_{\mathrm{shift}}(\mathbf{x}_{1:l}) \\ \mathbf{y}_{1:l} &= \mathbf{x}_{1:l} \\ \mathbf{y}_{l+1:2l} &= \mathbf{x}_{l+1:2l} \odot \exp(\mathbf{s}) + \mathbf{t} \end{aligned}$$

Afin de pouvoir modifier les deux parties du vecteur $\mathbf{x}$, on alterne à chaque couche le bloc de dimension qui subit la transformation affine ($\mathbf{x}_{l+1:2l}$) et celui qui est inchangé ($\mathbf{x}_{1:l}$).

Convolution 1x1 inversible Une convolution 1x1 (i.e. noyau 1) correspond à une transformation linéaire des canaux d'entrées. Dans cet exercice, nous avons 2 canaux d'entrée (et de sortie) donc W est de taille 2×2. Quand nous travaillerons avec des images, il y aura autant de transformations que de "points" (ici, nous n'avons qu'un point). Dans tous les cas, la transformation correspond (pour chaque point) à

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

où W est une matrice inversible.

Afin de conserver cette propriété au fur et à mesure de l'optimisation, on procède de la sorte : on décompose la matrice $\mathbf{W}$ via une décomposition LU, i.e.

$$\mathbf{W} = \mathbf{PLU}$$

avec $\mathbf{P}$ une matrice de permutation, $\mathbf{L}$ une matrice triangulaire inférieure, et $\mathbf{U}$ une matrice triangulaire supérieure (avec diagonale $\mathbf{L}$ composée seulement de 1 pour garantir l'unicité). On isole les diagonales pour obtenir

$$\mathbf{W} = \mathbf{P}\ (\mathbf{L}' + \mathbf{Id})(\mathbf{U}' + \mathbf{S}) \tag{1}$$

avec $\mathbf{L}'$ et $\mathbf{U}'$ des triangulaires avec des diagonales à zéro, et $\mathbf{S}$ une matrice diagonale.
À partir de l'équation 1, on peut calculer $|det(\mathbf{W})|$ en se rappelant que l'inversion de 2 lignes (ou colonnes) dans une matrice multiplie le déterminant par -1.

Dans `utils.py`, nous fournissons le code permettant d'initialiser les paramètres et de calculer la matrice W à partir de ceux-ci.

Vous pourrez utiliser les fonctions `torch.inverse` afin d'implémenter les différentes fonctions.

Données jouet Commencez par tester un modèle Glow (10 couches alternant) en utilisant des données jouet provenant de sklearn (moons) en utilisant une alternance de 10 couches ActNorm, Affine Coupling et Convolution1x1. Vous utiliserez le MLP défini dans utils.py comme fonction pour calculer la transformation (dans Affine Coupling).

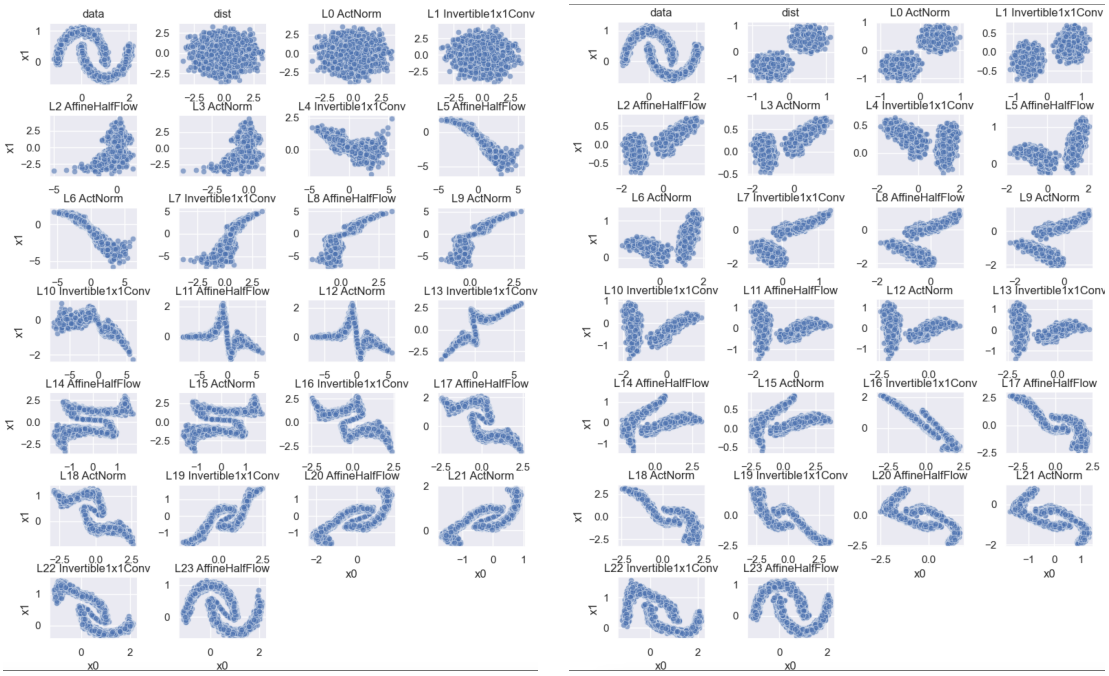
```
from sklearn import datasets
# data, _ = datasets.make_moons(
    n_samples=n_samples, shuffle=True, noise=0.05, random_state=0
)
```

Vous devriez arriver à un solution similaire à celle de la figure 3 : figure 3a) prior Gaussien classique, figure 3b) prior mélange de deux Gaussiennes.

5 Bonus: Glow (part II)

Extension à la 2D Pour exploiter ce que nous venons de voir sur des images (tenseurs $w \times h \times c$), appliquer les opérations vues précédemment ne suffit plus ; il est nécessaire d'exploiter les symétries 2D des réseaux de convolution. Pour cela, les trois opérateurs vus précédemment doivent être adaptés pour travailler sur les images $\mathbf{x} \in \mathbb{R}^{c \times h \times w}$:

- Actnorm/Convolution : La même transformation va être appliqué à chaque élément du tenseur $\mathbf{x}_{ij}$
- Affine Coupling : La transformation est calculé et appliquée pour chaque élément $\mathbf{x}_{ij}$ (elle n'est pas globale)



a) Prior Gaussien b) Prior mélange de 2 Gaussiennes

Figure 3: Une solution pour le problème des lunes (en haut à gauche les données d'origine, ensuite les données après chaque transformation)

Multi-scale (squeeze/split) On définit deux autres opérations. La première (*squeeze*) est basée sur la compression de l'espace dans les canaux: un tenseur dans $\mathbb{R}^{c\times w\times h}$ devient un tenseur dans $\mathbb{R}^{4c\times \frac{h}{2}\times \frac{w}{2}}$. Ainsi, par exemple pour une image de 2x2 pixels de 1 canal, on obtient 1 seul pixel mais de 4 canaux. C'est une manière détournée de faire du Pooling.

Afin de conserver un temps de calcul rapide et de profiter des différentes échelles de l'image, on suppose de plus pour l'opération *split*, qu'une partie de la représentation ne va plus être modifiée par la suite. À noter que cette opération est appliquée au niveau des canaux, i.e. on passe d'un tenseur dans $\mathbb{R}^{c\times h\times w}$ à un tenseur dans $\mathbb{R}^{\frac{c}{2}\times h\times w}$. Ainsi, split est définie par:

$$z_{i-1} = concat(channels(z_i, 1 : c/2); f_i^{-1}(channels(z_i, c/2, c)))$$

où $channels(z, range)$ retourne les canaux de z appartenant à la plage d'indices passée en paramètre, et *concat* concatène les tenseurs passés en paramètres, selon la dimension des canaux.

L'ensemble des opérations est représenté dans la figure 5. Vous utiliserez $L = 3$ et $K = 16$ pour CIFAR10. Au niveau du réseau utilisé pour l'opération *Affine Coupling*, vous utiliserez un réseau composé de trois convolution (3x3, 1x1, puis 3x3), avec une activation ReLU et 512 canaux (pour les couches cachées).

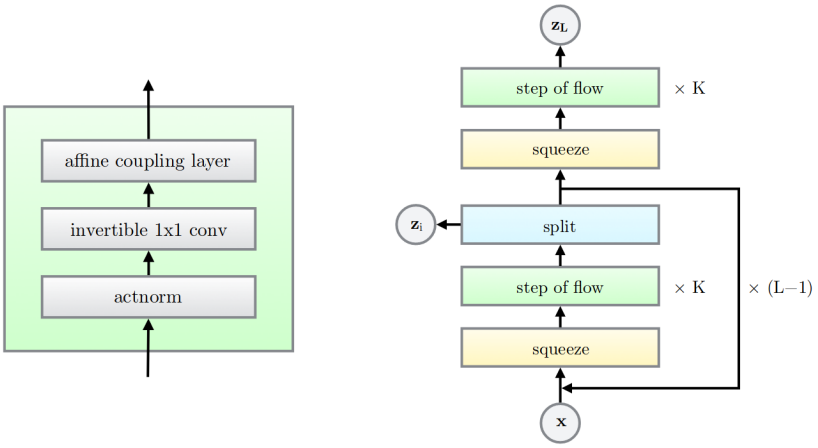


Figure 4: Modèle Glow [1]

Expériences Vous utiliserez alors le modèle Glow sur les données CIFAR10 :

```
from torchvision import datasets, transforms
train_dataset = datasets.CIFAR10(
    root='./cifar_data/', train=True,
    transform=transforms.ToTensor(), download=True
)
```

Quelques fonctions utiles

Pour afficher une série d'échantillons (1D ou 2D), vous pourrez utiliser la fonction `scatterplots` définie dans le fichier `utils.py`.

References

[1] D. P. Kingma and P. Dhariwal, "Glow: Generative Flow with Invertible 1x1 Convolutions," *arXiv:1807.03039 [cs, stat]*, Jul. 2018, arXiv: 1807.03039. [Online]. Available: http://arxiv.org/abs/1807.03039