

---

# Projet Fil Rouge

## Modèles de diffusion pour l'analyse de matériaux

---

**Arnaud Aillaud**  
arnaud.aillaud@telecom-paris.fr

**Clément Bourguize-Ramel**  
bourguize-ramel@telecom-paris.fr

**Pierre-Antoine Clouzeau**  
clouzeau@telecom-paris.fr

**Hippolyte Sibleau**  
hippolyte.sibleau@telecom-paris.fr

**Nacim Belkhir**  
Tuteur Entreprise  
Safran  
nacim.belkhir@safrangroup.com

**Arturo Mendoza Quispe**  
Tuteur Entreprise  
Safran  
arturo.mendoza-quispe@safrangroup.com

**Gianni Franchi**  
Tuteur Académique  
ENSTA Paris  
gianni.franchi@ensta-paris.fr

### Résumé

Your abstract.

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>État de l'art des modèles génératifs</b>	<b>6</b>
2.1	Autoencodeurs . . . . .	6
2.1.1	Architecture classique . . . . .	6
2.1.2	Variational Autoencoders (VAE) . . . . .	6
2.1.3	Vector-Quantised VAE (VQ-VAE) . . . . .	8
2.2	Generative adversarial network . . . . .	10
2.2.1	Vanilla GAN . . . . .	10
2.2.2	WGAN . . . . .	12
2.3	VQGAN . . . . .	14
2.4	Diffusion models . . . . .	14
<b>3</b>	<b>Denoising Diffusion Probabilistic Models</b>	<b>16</b>
3.1	Justifications théoriques . . . . .	16
3.1.1	Notations . . . . .	16
3.1.2	Expression de la ELBO Loss . . . . .	17
3.1.3	Simplification de la fonction de perte . . . . .	18
3.2	Implémentation . . . . .	20
3.2.1	Architecture initiale . . . . .	20
3.2.2	Résultats . . . . .	22
3.3	DDIM . . . . .	25
<b>4</b>	<b>Latent Diffusion Models</b>	<b>26</b>
4.1	Fonctionnement de Stable Diffusion . . . . .	26
4.1.1	Compression perceptuelle des images . . . . .	26
4.1.2	Mécanisme de conditionnement . . . . .	28
4.1.3	Diffusion dans l'espace latent . . . . .	30

4.2	Entraînements de LDMs . . . . .	30
4.2.1	Finetuning classique . . . . .	30
4.2.2	Textual Inversion . . . . .	33
4.2.3	DreamBooth . . . . .	36
4.2.4	LoRA . . . . .	41
4.2.5	Récapitulatif . . . . .	45

**Références****47**

## 1 Introduction

Lors du développement d'une pièce, du fait du contexte industriel particulier de Safran, les chercheurs sont tenus de caractériser pleinement l'impact de leurs processus de fabrication sur la qualité des pièces. Bien qu'impérative, cette exigence doit être atteinte en composant avec les coûts de réalisation des expériences. Dans ce contexte, la division I.A. de Safran entraîne des modèles génératifs afin de fournir aux chercheurs un outils d'aide à l'exploration de l'espace expérimental. Ces modèles permettent ainsi de générer des données correspondantes à des zones de l'espace expérimental qui n'auraient pas pu être explorées, car trop coûteuses et/ou difficilement atteignables techniquement. Ils permettent également d'effectuer un maillage plus fin de certaines zones ou encore de constituer une aide à la décision des zones à explorer en laboratoire. Enfin, lors d'un résultat expérimental, ces outils permettent de mieux en étudier la configuration expérimentale.

Les équipes de Safran sont donc intéressées par une étude de l'état de l'art des modèles génératifs, tout particulièrement des modèles de diffusion qui commencent à s'imposer en tant que potentiels successeurs des GANs, et de leur utilisation dans un contexte industriel (type d'images spécifiques / non présents dans les jeux de données classiques sur lesquels les gros modèles sont entraînés, nombre d'images limité, ressources de calcul modérées, etc.).

Notre approche pour ce projet a donc d'abord consisté en une analyse des grandes familles de modèles génératifs (VAE, VQ VAE, GANs) pour se familiariser avec les méthodes génératives, en comprendre les forces et limites, mais aussi pour commencer avec des modèles plus simples pour nos premières implémentations de modèles de Deep Learning avec Pytorch. Lors de cette première phase, nous avons validé l'implémentation de nos modèles avec des jeux de données classiques de traitement d'images (datasets simples, de petite taille, permettant d'obtenir des résultats et d'itérer rapidement) :

- **MNIST** (Modified National Institute of Standards and Technology database) [1]  
Il s'agit d'une collection de chiffres manuscrits en noir et blanc, qui comporte 60 000 images d'entraînement et 10 000 images de test. Les images sont de petite dimension (28 x 28 pixels)

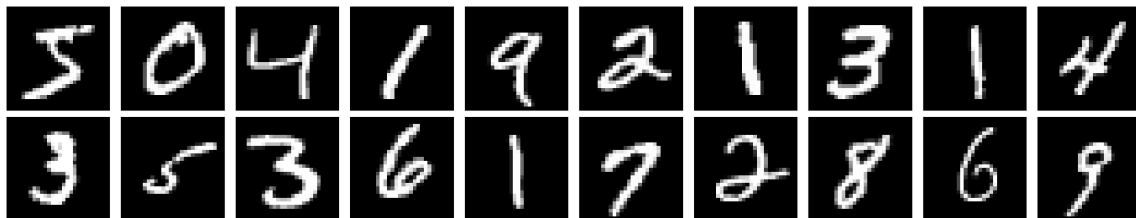


FIGURE 1 – Exemple d'images extraites du dataset MNIST

- **CIFAR-10** (Canadian Institute for Advanced Research, 10 classes) [2]  
Il s'agit d'un jeu de données contenant 60 000 images couleur de petite taille (32 x 32 pixels) réparties selon 10 classes : avion, automobile, oiseau, chat, cerf, chien, grenouille, cheval, bateau et camion



FIGURE 2 – Exemple d'images extraites du dataset CIFAR-10

Après ce premier état de l'art, nous nous sommes focalisés sur l'étude du premier article à obtenir de bons résultats de génération d'images avec un modèle de diffusion. Nous avons analysé le cadre théorique de cet article, et en avons implanté une version en Pytorch. Nous avons généré des images à partir du dataset CIFAR-10, présenté ci-dessus, et **DTD** (Describable Textures Dataset) [3], un dataset contenant 5639 images de texture de tailles variables entre 300 x 300 pixels et 640 x 640 pixels, réparties selon 47 classes : banded, braided, bubbly, cobwebbed, chequered, cracked, dotted, honeycombed, wrinkled, etc. Les matériels sur lesquels les équipes Safran travaillent étant des tressages de fibres de carbones, l'utilisation d'un dataset de texture nous permettait de nous rapprocher du contexte entreprise.



FIGURE 3 – Exemple d'images extraites du dataset DTD

Dans une dernière phase, après avoir maîtrisé l'utilisation de modèles de diffusion avec conditionnement, nous avons comparé différentes méthodes d'entraînement plus efficaces sur un jeu de données interne fourni par Safran.

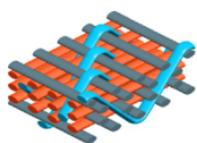


FIGURE 4 – Exemple de tissage 3D des fibres

Les équipes qui nous encadrent pour ce projet travaillent notamment sur les aubes de réacteurs d'avions **de type LEAP** (figure 5) fabriquées via un processus innovant impliquant le tissage 3D de fibres carbones composites. Le tissage de ces fibres en 3 dimensions est assez complexe et implique différents types de fibres, notamment chaîne (*warp* en anglais) et trame (*weft* en anglais).

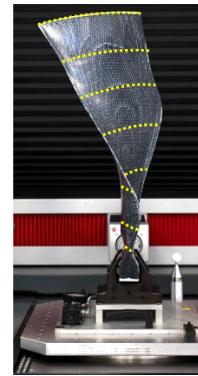


FIGURE 5 – Exemple d'aube de réacteur produite par Safran

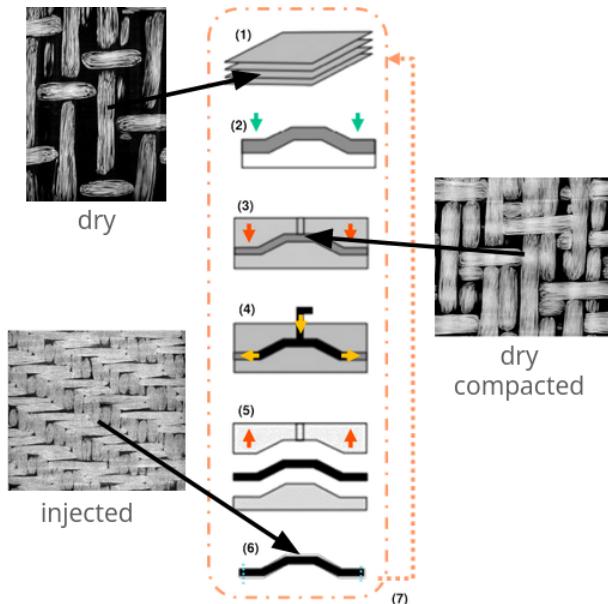


FIGURE 6 – Processus de fabrication d'une aube

Le tissage constitue la première des 7 étapes principales nécessaires pour créer une aube via moulage par transfert de résine :

1. Préparation du textile
2. Mise en forme
3. Fermeture du moule (compression)
4. Injection et cuisson de la résine
5. Démoulage de la pièce durcie
6. Traitement final
7. Optimisation du moule

Pour contrôler la qualité des pièces au cours de la fabrication, une analyse par tomographie 3D des pièces est effectuée à 3 moments cruciaux du processus : avant insertion dans le moule (ce qui donne des images de type **dry**), après compression des fibres dans le moule (images de type **dry-compacted**) et en sortie du moule après injection et durcissement de la résine (images de type **injected**).

Les images obtenues sont en 3 dimensions (hauteur, largeur et profondeur). Pour une même pièce, en fonction de la coupe 2D effectuée, il est donc possible d'observer la *thickness*, *warp* ou *weft* du matériau (cf. figure 7). Trois images 3D au format **tif** nous ont été fournies : une de type **dry** de profondeur 359, une de type **dry-compacted** de profondeur 197 et une de type **injected** de profondeur 287, ainsi qu'un script permettant de récupérer chacune des coupes 2D constituant ces images. Au total, nous avons donc travaillé avec :

- 2031 images **dry** dont
  - 359 images de *thickness* (résolution 702 x 970)

- 702 images de *warp* (résolution 970 x 359)
- 970 images de *weft* (résolution 702 x 359)
- 1833 images **dry-compacted** dont
  - 197 images de *thickness* (résolution 863 x 773)
  - 863 images de *warp* (résolution 773 x 197)
  - 773 images de *weft* (résolution 863 x 197)
- 1621 images **injected** dont
  - 287 images de *thickness* (résolution 701 x 633)
  - 701 images de *warp* (résolution 633 x 287)
  - 633 images de *weft* (résolution 701 x 287)

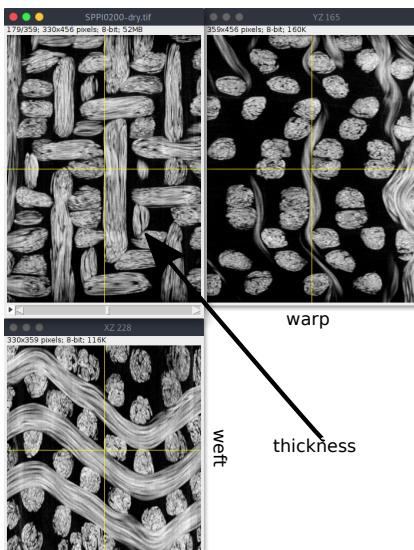


FIGURE 7 – Coupes 2D d'une image *dry* du dataset Safran

	thickness	warp	weft	Type
<b>dry</b>				
<b>dry-compacted</b>				
<b>injected</b>				

TABLE 1 – Exemple des 9 types d'images composant le dataset Safran

L'ensemble de nos résultats et analyses sont détaillés dans le présent rapport. Le code que nous avons utilisé lors de nos implémentations est disponible sur le dépôt GitHub suivant : <https://github.com/Pnyaa/Diffusion-models>.

## 2 État de l'art des modèles génératifs

Il existe deux grandes catégories de modèles génératifs en vision par ordinateur :

- Modèles génératifs explicites, basés sur la vraisemblance, qui apprennent la distribution d'un jeu de données avant de pouvoir générer de nouveaux exemples, tels que les modèles à énergie, auto-encodeurs variationnels, flux normalisants et modèles autorégressifs
- Modèles génératifs implicites, tels que les GANs, qui apprennent à générer de nouvelles données sans être directement exposés à la distribution initiale des données, et retiennent implicitement des informations sur cette distribution lors de cet apprentissage

Les architectures les plus récentes de modèles génératifs combinent ces différents modèles. Nous allons donc brièvement expliquer ces modèles avant de présenter les architectures de modèles à l'état de l'art.

### 2.1 Autoencodeurs

Bien que les autoencodeurs classiques ne fassent pas réellement partie de la famille des modèles génératifs, leur architecture sert de base à certains d'entre eux. Elle est donc brièvement introduite afin de mieux présenter ses dérivations.

#### 2.1.1 Architecture classique

L'objectif poursuivit par les autoencodeurs classiques est la réduction de dimensions ([4]). Pour obtenir une bonne représentation réduite des données, la stratégie est d'entraîner conjointement deux Multi Layer Perceptron (MLP) : un encodeur qui produit une représentation de dimension réduite et un décodeur qui reconstruit les données à partir de cette représentation. La sortie du décodeur peut ainsi être comparée à l'entrée de l'encodeur, et le réseau entraîné à en minimiser la différence. Un schéma de cette architecture est présenté dans la figure 8.

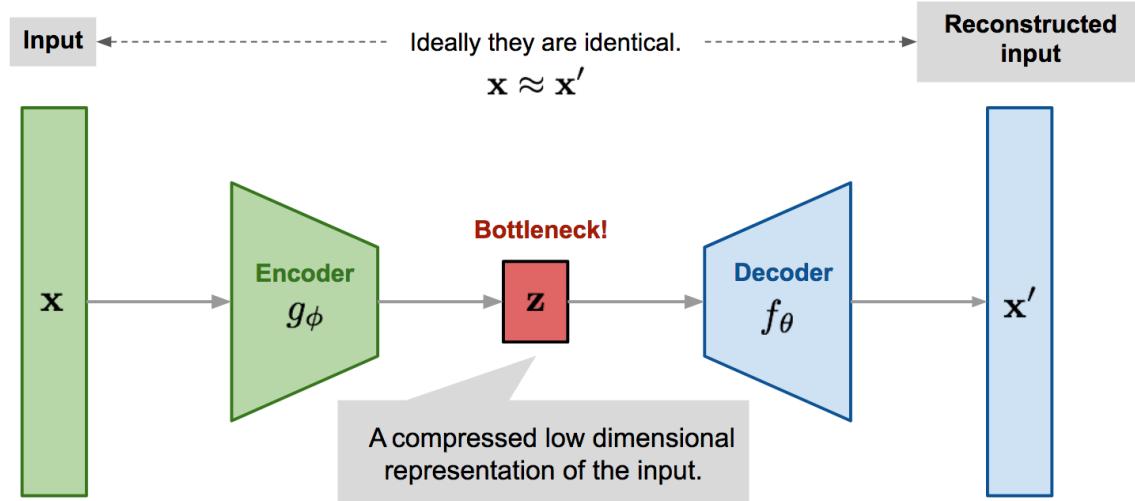


FIGURE 8 – Architecture générique d'un autoencodeur [5]

#### 2.1.2 Variational Autoencoders (VAE)

Les VAE ([6]) s'introduisent dans un contexte de maximisation de vraisemblance dans le cadre de modèles à variables cachées. On fait donc l'hypothèse que les données dont on dispose ont été générées en deux temps : une première valeur d'une variable aléatoire de distribution  $p_\theta(\mathbf{z})$  a été tirée puis utilisée pour échantillonner une distribution  $p_\theta(\mathbf{x}|\mathbf{z})$ . On cherche donc à maximiser la log-vraisemblance :

$$\log(p_\theta(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)})) = \sum_{i=1}^N \log(p_\theta(\mathbf{x}^{(i)}))$$

Les auteurs de [6] se placent dans une situation où ni  $p_\theta(\mathbf{x})$ , ni  $p_\theta(\mathbf{z}|\mathbf{x})$  ne sont calculables. On ne peut donc pas maximiser directement cette vraisemblance, ni faire appel à l'algorithme EM ([7]). On introduit une fonction d'approximation de  $p_\theta(\mathbf{z}|\mathbf{x})$  :  $q_\phi(\mathbf{z}|\mathbf{x})$ . On peut montrer ([5]) la relation suivante :

$$\log(p_\theta(\mathbf{x}^{(i)})) = D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}^{(i)}) || p_\theta(\mathbf{z}|\mathbf{x}^{(i)})) + \mathcal{L}(\theta, \phi, \mathbf{x}^{(i)})$$

Le premier terme du membre de droite est la divergence de Kullback-Leibler([8]), c'est une mesure de dissimilarité entre deux distributions. D'après l'inégalité de Gibbs, cette mesure est toujours positive et ne s'annule que lors d'égalité entre les deux distributions. Cette observation permet d'écrire l'inégalité suivante et de s'affranchir du terme  $p_\theta(\mathbf{z}|\mathbf{x})$  :

$$\log(p_\theta(\mathbf{x}^{(i)})) \geq \mathcal{L}(\theta, \phi, \mathbf{x}^{(i)})$$

De cette inégalité, on tire le nom du membre de droite de l'équation : variational lower bound. On donne l'expression de ce terme ci-dessous :

$$\mathcal{L}(\theta, \phi, \mathbf{x}^{(i)}) = -D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}^{(i)})||p_\theta(\mathbf{z})) + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}[\log(p_\theta(\mathbf{x}^{(i)}|\mathbf{z}))]$$

Pour ce qui est du premier terme, les auteurs de [6] choisissent  $p_\theta(\mathbf{z}) \sim \mathcal{N}(0, \mathbf{I})$  et  $q_\phi(\mathbf{z}|\mathbf{x}^{(i)}) \sim \mathcal{N}(\boldsymbol{\mu}^i, (\boldsymbol{\sigma}^i)^2 \mathbf{I})$ . Ce choix permet d'obtenir une expression analytique de la divergence KL :

$$\mathcal{L}(\theta, \phi, \mathbf{x}^{(i)}) = \frac{1}{2} \sum_{j=1}^J (1 + \log(\sigma_j^{(i)})^2 - (\mu_j^{(i)})^2 - (\sigma_j^{(i)})^2) + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}[\log(p_\theta(\mathbf{x}^{(i)}|\mathbf{z}))]$$

où  $j$  est la dimension de  $\mathbf{z}$ . Pour ce qui est du deuxième terme, les auteurs suggèrent de l'estimer par la méthode de Monte Carlo. L'échantillonnage se fait à partir de  $q_\phi(\mathbf{z}|\mathbf{x}^{(i)}) = \boldsymbol{\mu}^{(i)} + \boldsymbol{\sigma}^{(i)} \odot \boldsymbol{\epsilon}$  avec  $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$ . On obtient donc, pour  $L$  échantillons Monte Carlo.

$$\mathcal{L}(\theta, \phi, \mathbf{x}^{(i)}) = \frac{1}{2} \sum_{j=1}^J (1 + \log(\sigma_j^{(i)})^2 - (\mu_j^{(i)})^2 - (\sigma_j^{(i)})^2) + \frac{1}{L} \sum_{l=1}^L \log(p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,l)})) \quad (1)$$

Les valeurs de moyenne et d'écart-type de  $q_\phi(\mathbf{z}|\mathbf{x}^{(i)})$  sont prises comme sorties d'un réseau MLP. Pour ce qui est du deuxième terme, on ne le maximise pas tel quel. Une fois l'échantillonnage effectué, on entraîne un deuxième réseau MLP à reconstruire la donnée d'entrée à partir des valeurs  $\mathbf{z}^{(i,l)}$ . Entraîner un réseau à reconstruire la donnée d'entrée revient à maximiser la probabilité d'obtenir cette même image en sortie de ce même réseau sans pour autant travailler explicitement avec la fonction de densité  $p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,l)})$ . On peut donc remplacer le terme de Monte Carlo dans 1 par un terme de binary cross entropy entre l'image d'entrée et l'image de sortie. On obtient ainsi la fonction de coût finale d'un VAE :

$$\mathcal{L}(\theta, \phi, \mathbf{x}^{(i)}) = \frac{1}{2} \sum_{j=1}^J (1 + \log(\sigma_j^{(i)})^2 - (\mu_j^{(i)})^2 - (\sigma_j^{(i)})^2) + \sum_{i=1}^D x_i \log(y_i) + (1 - x_i) \log(1 - y_i) \quad (2)$$

où  $D$  est la dimension des données,  $\mathbf{y}$  est la sortie du réseau et  $\mathbf{x}$  la donnée en entrée. On peut à présent comparer ce réseau à un auto-encodeur dans le sens suivant : les données d'entrée sont représentées par des objets de dimension plus petite (les paramètres des distributions gaussiennes). Ces objets sont obtenus comme sortie d'un réseau MLP qui joue un rôle similaire à l'encodeur. De même, on entraîne également un autre réseau à reconstruire les données d'entrée, lui faisant jouer un rôle similaire au décodeur. On présente un schéma de cette architecture dans la figure 9

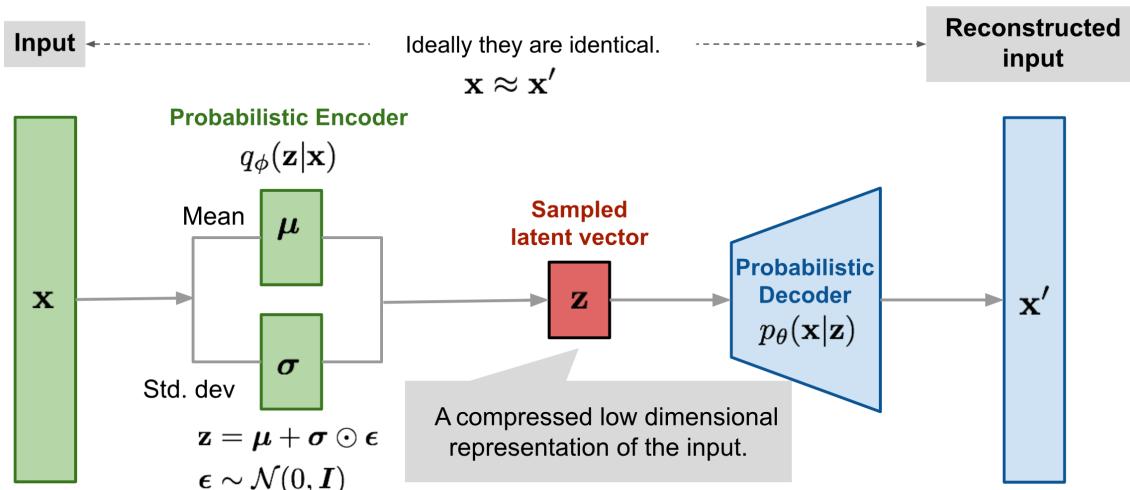


FIGURE 9 – Architecture d'un VAE [5]

On implémente ce modèle pour l'entraîner sur le jeu de données MNIST. Pour l'encodeur, on fait le choix d'une couche cachée à 400 noeuds, suivie d'une fonction ReLU puis de deux couches à 20 noeuds qui donnent les

vecteurs de moyenne et d'écart-type des gaussiennes. Enfin, pour le décodeur, on choisit également une couche cachée à 400 noeuds suivie d'une fonction d'activation ReLU. On termine le décodeur par une couche à 784 noeuds suivie d'une fonction d'activation sigmoïde. L'évolution de la fonction de coût 2 lors de l'entraînement est présentée dans la figure 10

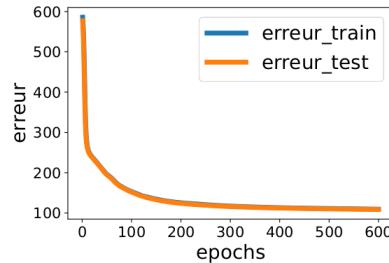


FIGURE 10 – Evolution de la fonction de coût 2 en fonction du nombre d'éPOCHS

Une fois le modèle entraîné, on peut comparer les images d'origine et celles reconstruites par le modèle visuellement. La figure 11 présente un exemple de comparaison.

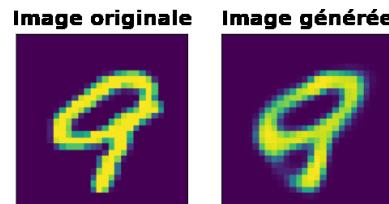


FIGURE 11 – Comparaison entre une même image en entrée et en sortie du VAE

On s'intéresse finalement à la répartition des moyennes des vecteurs gaussiens grâce à une représentation t-SNE présentée dans la figure 12

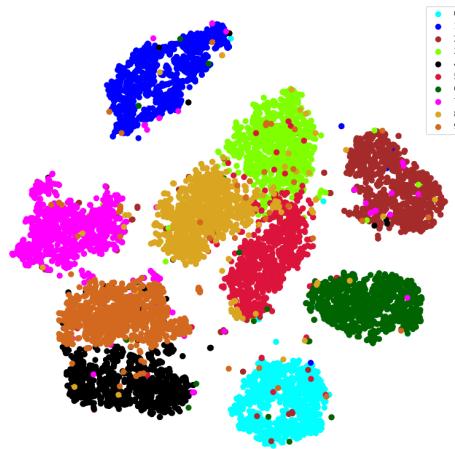


FIGURE 12 – Représentation t-SNE des vecteurs moyennes des gaussiennes avec coloration en fonction du label

Comme on peut le voir dans la figure 12, les  $\mu^{(i)}$  appris lors de la phase d'entraînement forment des clusters en fonction de la classe de la donnée d'entrée correspondante.

### 2.1.3 Vector-Quantised VAE (VQ-VAE)

Tout comme un VAE, un VQ-VAE est un modèle de deep learning avec un encoder et un decoder. Il possède également un Vector Quantisation Layer (VQL), qui lui permet d'interagir avec un dictionnaire d'embedding (un codebook). La principale différence entre un VAE et un VQ-VAE est que le VAE utilise un espace latent continu tandis que le VQ-VAE utilise un espace latent discret grâce à un dictionnaire d'embedding, ce qui permet d'obtenir des résultats souvent moins irréalistes, et des reconstructions plus proches de la réalité, notamment sur des images.

L'encoder prend en entrée une image de taille  $(n, h, w, d)$  et retourne un encoding de cette image ( $z_e$ , de même taille). Le codebook est un dictionnaire de  $k$  vecteurs à  $d$  dimensions initialisé grâce à une loi uniforme sur  $[-1/k, 1/k]$ . La première étape dans le VQL est de reformater  $z_e$  pour obtenir une matrice de taille  $(n * h * w, d)$ . On peut ensuite calculer les distances (euclidiennes) entre chaque vecteur de  $z_e$  et les  $e_k$ , éléments du codebook. On détermine alors pour chacun des éléments de  $z_e$  le vecteur le plus proche dans le codebook (argmin), et on renvoie les vecteurs correspondants dans  $z_q$ , qui sera donc de taille  $(n * h * w, d)$ . La dernière étape consiste à reformater  $z_q$  pour retrouver une matrice de taille  $(n, h, w, d)$ . Toutes ces étapes sont résumées dans le schéma ci-dessous.

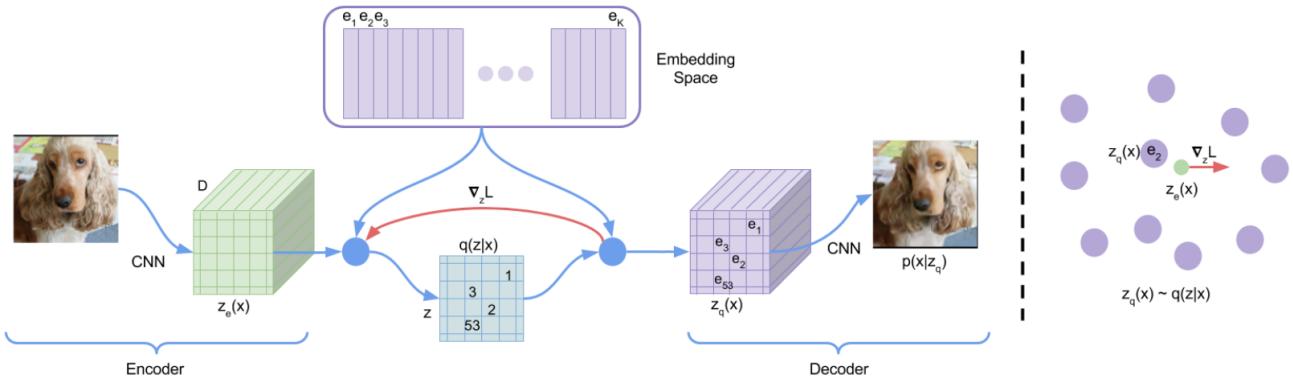


FIGURE 13 – A gauche, schéma d'architecture d'un VQ-VAE / A droite, visualisation de l'espace d'embedding : la sortie du décodeur  $z_q$  est associée au point le plus proche (e2). Le gradient, représenté en rouge, entraîne la sortie de l'encodeur vers une valeur discrète différente pour la prochaine passe [9]

Le principal problème de cette méthode est qu'on ne peut effectuer une backpropagation classique puisque la fonction argmin n'est pas différentiable. On se contente donc de transférer le gradient de la loss directement depuis la sortie du VQL ( $z_q$ ) jusqu'à l'encoder ( $z_e$ ) pour lui transférer de l'information pour améliorer sa sortie lors du training (et minimiser l'erreur de reconstruction).

$$L = \log(p(x|z_q(x))) + \|\text{sg}[z_e(x)] - e\|_2^2 + \beta\|z_e(x) - \text{sg}[e]\|_2^2$$

La loss du VQ-VAE est composée de trois termes. Le premier terme est la reconstruction loss, elle est optimisée par l'encoder et le décodeur. Elle permet de voir la proximité entre l'image d'origine et l'image reconstruite. Vu qu'il n'y a pas de backpropagation, elle n'est pas utilisée pour optimiser le dictionnaire d'embedding. Le deuxième terme, ou la codebook loss, permet de pousser les vecteurs  $e_k$  vers la sortie de l'encoder  $z_e$ . C'est ce qui permet de mettre à jour le codebook. L'expression "sg" signifie "stop gradient", qui permet de bloquer le calcul du gradient de ce qu'elle prend en argument dans le graphe de calcul. Ceci est dû à l'impossibilité de faire une backpropagation classique. Le dernier terme permet à l'inverse de pousser les vecteurs  $z_e$  vers les vecteurs du dictionnaire d'embedding. Le scalaire  $\beta$  permet de ralentir l'optimisation de la sortie de l'encoder.  $\beta$  prend 0.25 comme valeur par défaut, et il a été démontré que ce terme n'impacte pas vraiment le modèle quand il prend ses valeurs entre 0.1 et 2. Ci-dessous, voici un exemple de résultat de reconstruction que l'on peut obtenir avec un VQ-VAE à partir du dataset CIFAR10. Le modèle a été entraîné avec 5 000 epochs, et nous avons gardé la valeur par défaut de  $\beta$ , soit 0.25. On commence à pouvoir distinguer des formes similaires, mais plus d'epochs permettraient d'obtenir un meilleur résultat.

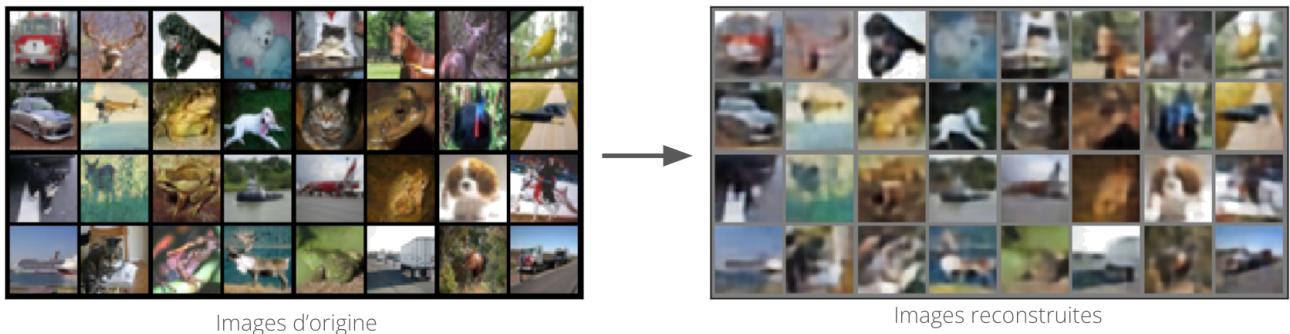


FIGURE 14 – Exemple d'images reconstruites avec un VQ-VAE

## 2.2 Generative adversarial network

### 2.2.1 Vanilla GAN

Contrairement aux modèles précédents, un modèle Generative adversarial network (GAN) apprend à générer des images sans être exposé à la distribution initiale, et n'est donc pas basé sur la log-vraisemblance. L'architecture d'un GAN est composée de 2 blocs

- Un **discriminateur**, qui est optimisé pour classifier les images qu'il reçoit en entrée. Les images de la distribution réelle doivent être associées au label 0, celles générées par le générateur au label 1.
- Un **générateur**, qui génère des images ressemblant à la distribution initiale à partir d'un espace latent. Son objectif est de créer les images les plus similaires à la distribution initiale pour tromper le discriminateur

Le schéma suivant représente cette architecture simplifiée :

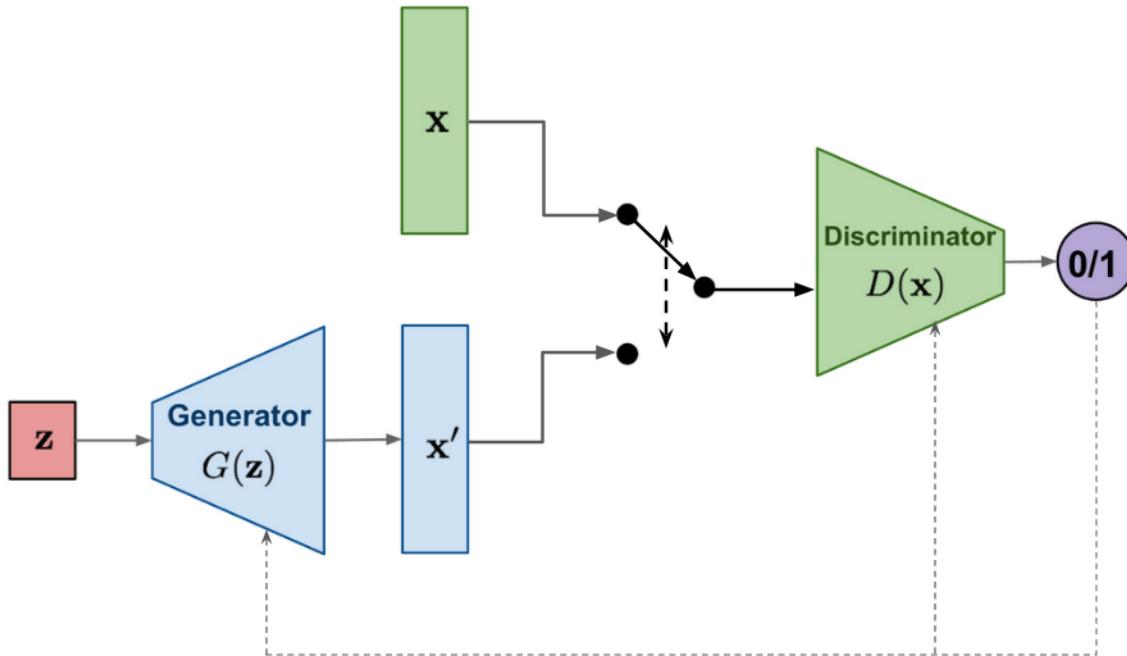


FIGURE 15 – Architecture simplifiée d'un GAN

Par la suite, les notations suivantes seront utilisées pour décrire les différentes distributions :

- $p_z$  distribution de données du bruit  $z$  en entrée du générateur
- $p_g$  distribution de données générées par le générateur, dont  $x'$  est un échantillon
- $p_r$  distribution de données réelles, dont  $x$  est un échantillon

Le discriminateur est optimisé pour labelliser une image en 0 si elle est issue de  $p_r$  et 1 si elle est générée par  $G$ , i.e. maximiser  $\mathbb{E}_{x \sim p_r(x)}[\log(D(x))]$  et  $\mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$ .

Le générateur est optimisé pour faire passer les images qu'il génère pour réelles, i.e. minimiser  $\mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$ .

La fonction de perte  $\mathcal{L}$  d'un GAN prend donc la forme suivante :

$$\begin{aligned}\mathcal{L}(G, D) &= \mathbb{E}_{x \sim p_r(x)}[\log(D(x))] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))] \\ &= \mathbb{E}_x[\log(D(x))] + \mathbb{E}_{x \sim p_g(x)}[\log(1 - D(x))]\end{aligned}$$

Elle porte le nom de **minimax loss**, car son optimisation force une minimisation et une maximisation de paramètres concurrents :  $\min_G \max_D \mathcal{L}(G, D)$ . L'équation de la fonction de loss permet de l'écrire de manière analytique

$$\mathcal{L}(G, D) = \int_x (p_r(x) \log(D(x)) + p_g(x) \log(1 - D(x))) dx$$

On peut chercher  $D(x)$  qui maximise cette équation en calculant pour quelles valeurs de  $\tilde{x} = D(x)$  la fonction

$f$  suivante s'annule.

$$\begin{aligned} f(\tilde{x}) &= p_r(x) \log(\tilde{x}) + p_g(x) \log(1 - \tilde{x}) \\ \frac{\partial f}{\partial \tilde{x}} &= \frac{p_r(x)}{\ln(10)\tilde{x}} - \frac{p_g(x)}{\ln(10)(1 - \tilde{x})} \\ &= \frac{p_r(x) - (p_r(x) + p_g(x))\tilde{x}}{\ln(10)\tilde{x}(1 - \tilde{x})} \end{aligned}$$

Cette valeur s'annule pour

$$p_r(x) - (p_r(x) + p_g(x))\tilde{x}^* = 0 \Leftrightarrow \boxed{\tilde{x}^* = \frac{p_r(x)}{p_r(x) + p_g(x)}} \in [0, 1]$$

Lorsque le générateur est parfaitement entraîné,  $p_g = p_r$  d'où  $D^*(x) = \tilde{x}^* = \frac{1}{2}$ . L'équation de la loss prend alors la forme suivante :

$$\begin{aligned} \mathcal{L}(G, D^*) &= \int_x (p_g(x) \log(\frac{1}{2}) + p_r(x) \log(\frac{1}{2})) dx \\ &= -\log(2) \int_x p_g(x) dx - \log(2) \int_x p_r(x) dx \\ &= -2 \log(2) \end{aligned}$$

La valeur minimale théorique de loss d'un GAN est donc de  $-2 \log(2)$ .

Cette valeur peut se retrouver en partant de l'équation de la divergence de Jensen-Shannon entre  $p_r$  et  $p_g$ , à savoir

$$\begin{aligned} D_{JS}(p_r || p_g) &= \frac{1}{2} D_{KL}(p_r || \frac{p_r + p_g}{2}) + \frac{1}{2} D_{KL}(p_g || \frac{p_r + p_g}{2}) \\ &= \frac{1}{2} (\log(2) + \int_x p_r(x) \log(\frac{p_r(x)}{p_r(x) + p_g(x)}) dx + \log(2) + \int_x p_g(x) \log(\frac{p_g(x)}{p_r(x) + p_g(x)}) dx) \\ &= \frac{1}{2} (\log(4) + \mathcal{L}(G, D^*)) \\ \Leftrightarrow \mathcal{L}(G, D^*) &= 2D_{JS}(p_r || p_g) - 2 \log(2) \end{aligned}$$

La fonction de perte d'un GAN quantifie donc la similarité entre la distribution réelle des données et la distribution du générateur par divergence de Jensen Shannon **lorsque le discriminateur est optimal**.

Malgré ce résultat, il n'y a aucune garantie de convergence pour un GAN, et donc aucune garantie d'atteindre un discriminateur optimal. Les problèmes liés à l'entraînement d'un GAN sont en réalité nombreux. Parmi les plus fréquents, on rencontre :

- Divergence lors de l'entraînement. En effet, la fonction de perte nécessite d'atteindre un **équilibre de Nash** pour espérer converger, ce qui est en pratique relativement compliqué avec l'implémentation standard d'un GAN
- Évanescence du gradient. En observant la loss du GAN, on remarque que lorsque le discriminateur est optimal, la fonction de perte s'annule. Il n'y a donc plus de mise à jour des paramètres de  $G$ , qui ne peut donc plus s'améliorer. Ainsi, si le discriminateur devient trop vite optimisé, l'entraînement est paralysé, mais s'il ne s'améliore pas assez, le génératuer n'a aucune incitation à produire des images de qualité. Ce dilemme rend l'entraînement d'un GAN complexe
- Mode collapse. Il arrive que lors de l'entraînement, le générateur trouve un type d'image qui trompe le discriminateur de manière systématique, il ne génère plus que ce type d'images. La diversité de la génération d'images est donc grandement affectée. Ce type de problème est nommé "Mode Collapse"
- Absence de métrique d'évaluation objective. La valeur de la fonction de perte d'un GAN n'est pas révélatrice de la qualité ni de la diversité des images générées, et le meilleur moyen de vérifier la qualité des images générées reste une observation humaine. Il existe cependant certaines métriques d'évaluation qui permettent de mesurer la qualité et la diversité des images générées dans une moindre mesure. La plus populaire est la Distance d'Inception de Fréchet (FID en anglais), qui mesure l'écart entre moyennes et matrices de covariance de la distribution de données réelles et générées. Plus formellement, elle revient à calculer la distance de Fréchet à partir des estimateurs des moyennes et matrices de covariance des 2 distributions, calculées d'une part sur les images d'entraînement pour  $p_r$  et sur un échantillon d'images générées pour  $p_g$  (les auteurs de l'article [10] qui introduit cette métrique conseillent de générer au moins

10000 images pour estimer ces valeurs). Pour calculer ces valeurs, les représentations générées par le modèle Inception v3 [11] sont utilisées (2048 features), d'où son nom.

$$\text{FID} = \|\mu_r - \mu_g\|_2^2 + \text{tr}(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{\frac{1}{2}})$$

S'agissant de la distance entre distribution initiale et distribution générée, plus sa valeur est faible, plus les images générées seront fidèles à la distribution initiale.

Nous avons rencontré certains de ces problèmes en entraînant un GAN, avec un générateur et un discriminateur chacun composé de 4 couches de convolution, sur CIFAR-10. Les résultats sont présentés ci-dessous :

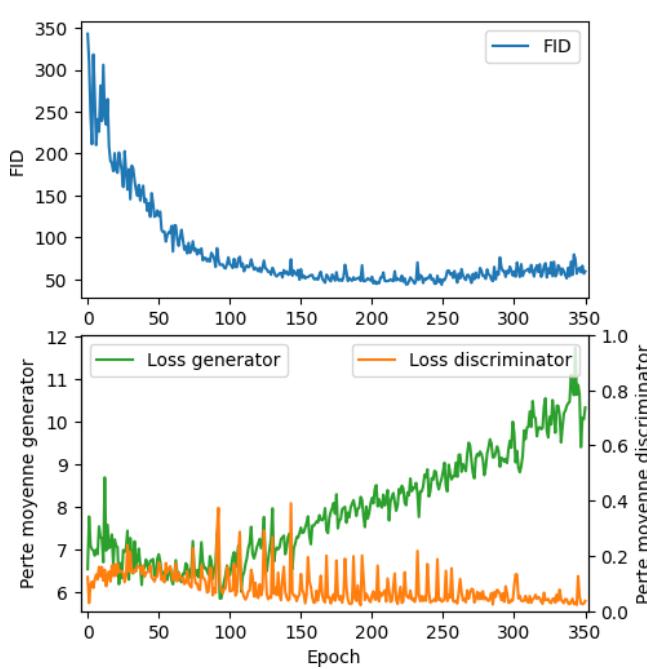


FIGURE 16 – Courbes de FID et de pertes pendant l'entraînement d'un GAN sur CIFAR-10

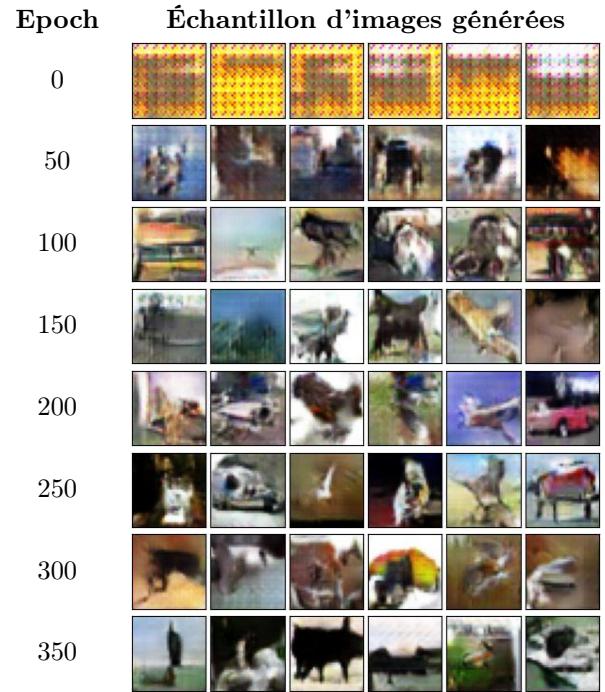


TABLE 2 – Exemples d'images générées à différentes epochs

Un premier constat est que la FID décroît jusqu'à atteindre un plateau entre les epochs 175 et 250, alors que la courbe de loss du générateur ne semble décroître que jusqu'à l'epoch 80, epoch à partir de laquelle elle se met à croître. Il y a donc bien décorrélation entre valeur de la fonction de perte et qualité des images générées. L'augmentation conjointe de la FID et de la valeur de la fonction de perte du générateur (avec une fonction de perte du discriminateur stagnante) à partir des epochs 300 est synonyme de divergence lors de l'entraînement : l'équilibrage de Nash n'est pas atteint. Poursuivre l'entraînement amène à un mode collapse : le générateur ne crée que des images semblables (en outre assez éloignées du dataset initial).



FIGURE 17 – Mode collapse observé à partir de 500 epochs d'entraînement

L'adéquation entre FID et qualité visuelle est par ailleurs confirmée par l'observation des images générées à chaque epoch. Les images les plus proches du dataset sont obtenues pour les epochs 150, 200 et 250 : on reconnaît quelques avions, voitures et animaux. Les images générées ne sont cependant pas très nettes, et bien qu'il y ait un clair progrès depuis l'epoch 0, les résultats ne semblent pas très satisfaisants pour un dataset d'entraînement a priori relativement simple.

### 2.2.2 WGAN

Pour réduire les problèmes de convergence décrits précédemment, Arjovsky et al. [12] ont proposé une fonction de perte différente pour entraîner un GAN, basée sur la distance de Wasserstein. La fonction de perte prend la forme suivante, qui ne contient plus de logarithme (noter la suppression du min, qui évite la concurrence entre maximisation et minimisation du Vanilla GAN) :

$$\mathcal{L}(p_r, p_g) = \max_{w \in W} \mathbb{E}_x[f_w(x)] - \mathbb{E}_z[f_w(g_\theta(z))]$$

Quelques autres légères modifications de l'algorithme sont proposées, notamment l'entraînement du discriminateur plusieurs fois par epoch (par défaut, 5 fois plus que le générateur), et l'écrêtage des poids du modèles (weight clipping) dans l'intervalle  $[-0.01, 0.01]$

---

**Algorithm 1** WGAN, our proposed algorithm. All experiments in the paper used the default values  $\alpha = 0.00005$ ,  $c = 0.01$ ,  $m = 64$ ,  $n_{\text{critic}} = 5$ .

---

**Require:** :  $\alpha$ , the learning rate.  $c$ , the clipping parameter.  $m$ , the batch size.  
 $n_{\text{critic}}$ , the number of iterations of the critic per generator iteration.

**Require:** :  $w_0$ , initial critic parameters.  $\theta_0$ , initial generator's parameters.

- 1: **while**  $\theta$  has not converged **do**
- 2:     **for**  $t = 0, \dots, n_{\text{critic}}$  **do**
- 3:         Sample  $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}_r$  a batch from the real data.
- 4:         Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
- 5:          $g_w \leftarrow \nabla_w [\frac{1}{m} \sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))]$
- 6:          $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
- 7:          $w \leftarrow \text{clip}(w, -c, c)$
- 8:     **end for**
- 9:     Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
- 10:      $g_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))$
- 11:      $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
- 12: **end while**

---

FIGURE 18 – Algorithme du WGAN

Nous avons testé cette implémentation en conservant l'architecture du GAN précédent (4 couches de convolution pour le générateur et le discriminateur), et modifiant simplement l'objectif d'entraînement, toujours sur CIFAR-10. Les résultats obtenus sont présentés ci-dessous :

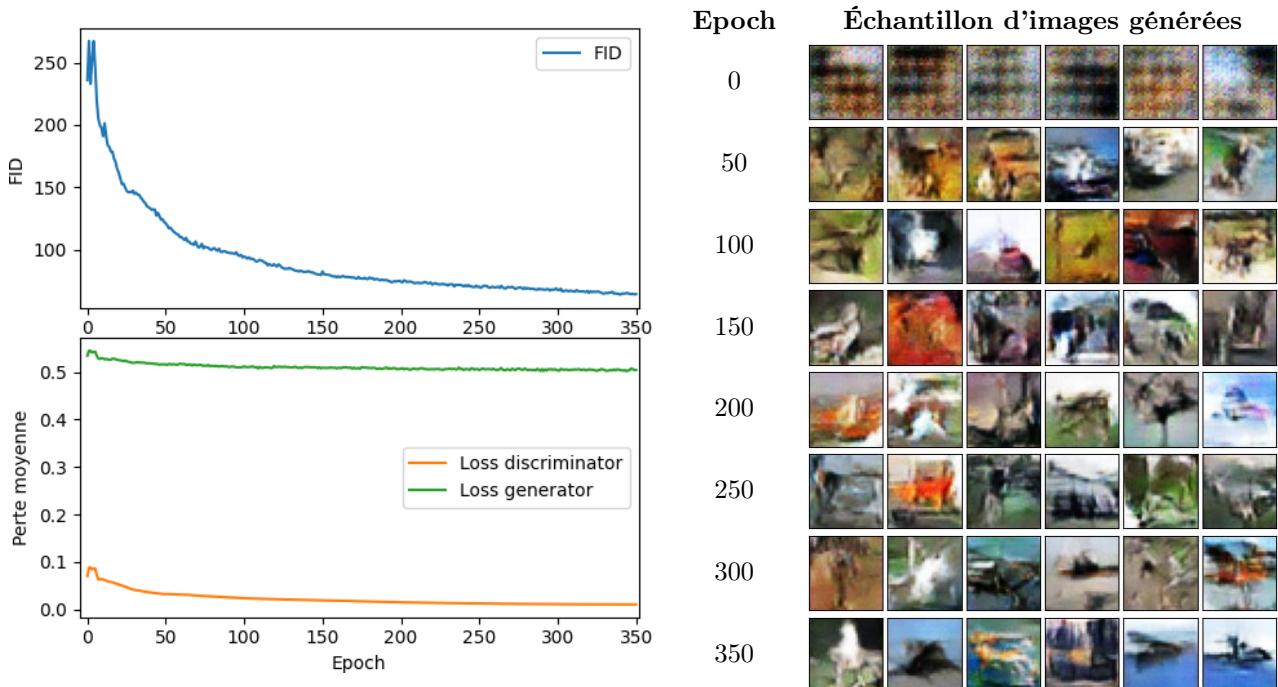


FIGURE 19 – Courbes de FID et de pertes pendant l'entraînement d'un WGAN sur CIFAR-10

TABLE 3 – Exemples d'images générées à différentes epochs

On remarque cette fois que tant la courbe de FID que les courbes de perte décroissent de manière continue pendant l'entraînement : le problème de stabilité observé pour le Vanilla GAN a bien été réglé.

Cependant, pour un nombre d'epoch identique, la FID est modérément plus élevée que pour le vanilla GAN, et les images générées plus floues. Étant donné les tendances observées, un entraînement plus long donnerait probablement des résultats légèrement, mais pas drastiquement meilleurs. L'implémentation WGAN permet donc effectivement de simplifier l'entraînement, mais n'aboutit pas nécessairement à des résultats plus probants.

Les images générées par un autoencodeur sont finalement plus réalistes que celles générées via des GANs sur CIFAR-10. Ceci est représentatif d'une autre faiblesse des GANs, à savoir qu'ils sont très utiles pour générer des images d'une distribution d'images similaires (typiquement des visages), mais ont plus de mal à apprendre des distributions plus diversifiées telles que celles du dataset CIFAR. Ce dataset considéré comme simple pour des tâches de classification, car images de petites tailles et classes facilement séparables, n'est en réalité pas si simple pour des tâches de génération.

### 2.3 VQGAN

Ce modèle hybride, présenté dans l'article *Taming Transformers for High-Resolution Image Synthesis* [13] de 2021, améliore la richesse du codebook d'un VQVAE grâce à l'ajout d'une fonction de perte perceptuelle et adversariale. Le codebook ainsi appris est ensuite utilisé conjointement avec un transformeur pour générer des images conditionnellement à du texte (ou autre type d'embedding)

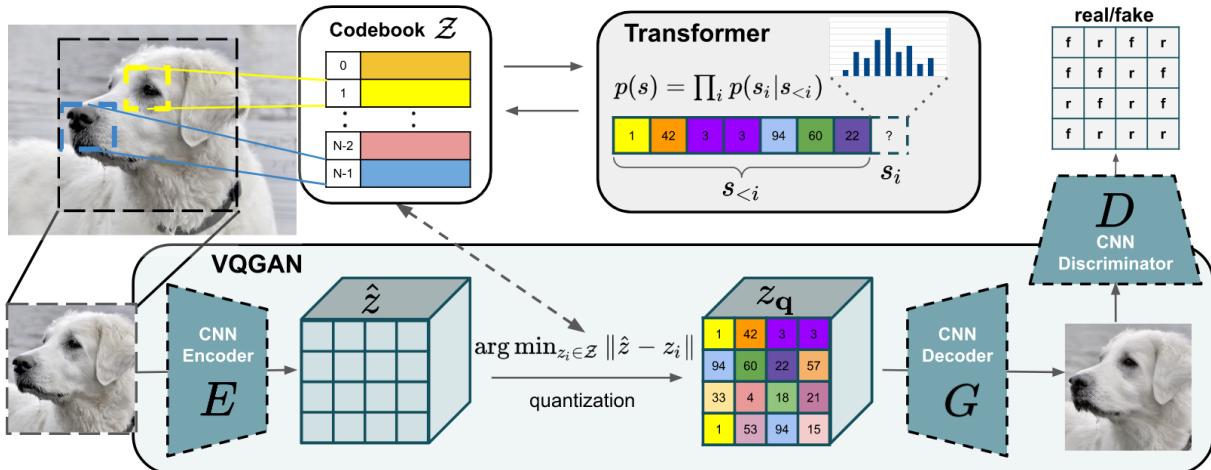


FIGURE 20 – Architecture simplifiée d'un modèle VQGAN utilisé pour de la génération d'images conditionnée

Première étape : Apprentissage de l'encodage des données

$$\mathcal{Q}^* = \arg \min_{E, G, \mathcal{Z}} \max_D \mathbb{E}_{x \sim p(x)} [\mathcal{L}_{VQ}(E, G, \mathcal{Z}) + \lambda \mathcal{L}_{GAN}(\{E, G, \mathcal{Z}\}, D)]$$

avec

$$\begin{cases} \mathcal{L}_{VQ}(E, G, \mathcal{Z}) = \|x - \hat{x}\|_2^2 + \|\text{sg}[E(x)] - z_q\|_2^2 + \|\text{sg}[z_q] - E(x)\|_2^2 \\ \mathcal{L}_{GAN}(\{E, G, \mathcal{Z}\}, D) = [\log(D(x)) + \log(1 - D(\hat{x}))] \end{cases}$$

Deuxième étape : Entraînement d'un transformeur autorégressif

$$\mathcal{L}_{Transformer} = \mathbb{E}_{x \sim p(x)} [-\log(p(s))]$$

### 2.4 Diffusion models

Les modèles de diffusion sont des algorithmes génératifs, introduits pour la première fois en 2015 dans un article des Université de Stanford et de Berkeley[14], avec déjà des résultats similaires aux modèles adversariaux de l'époque en terme de log-vraisemblance sur le dataset MNIST.

Cependant, l'article qui a réellement lancé les modèles de diffusion pour réaliser de la génération d'images est celui de 2020, toujours de l'université de Berkeley, intitulé *Denoising Diffusion Probabilistic Model* [15]. Cet article décrit un modèle de diffusion "simple", dans le sens où de nombreuses alternatives faisant appel à plus de complexité dans le choix des distributions de probabilités ont été étudiées par la suite.

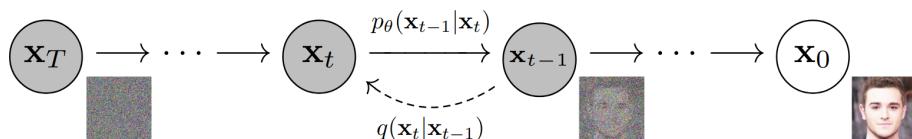


FIGURE 21 – Principe des modèles de diffusion

Le principe général qui sous-tend les modèles de diffusion est l'idée d'apprendre au modèle à débruiter des images corrompues par un certain bruit, et ce à différents niveaux de bruit.

Ces niveaux de corruption sont formalisés par une suite  $(x_{t,for})_{t \in [0,T]}$ , où  $x_0$  représente l'image originale et  $x_t$  l'image corrompue après  $t$  étapes. Cette suite s'appelle "forward process". Le processus inverse, appelé reverse process, doit permettre à partir d'une image aléatoire, de reconstruire une image cohérente à l'aide d'une suite  $(x_{t,rev})_{t \in [T,0]}$ . Dans la suite, nous noterons les deux processus, forward ou reverse, simplement  $(x_t)$ , en accord avec la littérature dans le domaine.

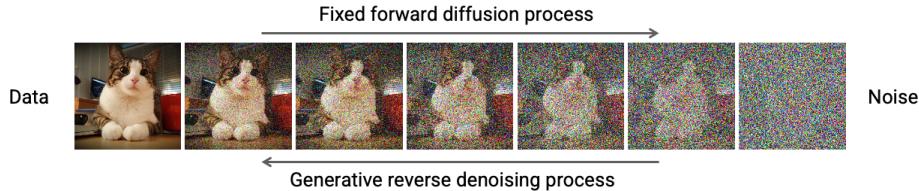


FIGURE 22 – Illustration d'un processus forward et reverse

Le principal intérêt des modèles de diffusion est leur capacité à générer des images de qualité équivalente à celles générées via des GANs (jusqu'alors état de l'art pour la génération d'images), sans les problèmes liés à l'instabilité de l'entraînement de ce type de modèles. L'image ci-dessous illustre par exemple les résultats obtenus sur le dataset CIFAR-10 par l'équipe de Berkeley en 2020. Les images échantillonées sont présentées de gauche à droite, de  $x_T$  jusqu'à  $x_0$ . Beaucoup d'images intermédiaires ne sont pas présentées, le nombre d'étapes de bruitage utilisé étant important ( $T = 1000$  steps). Sur ce type de dataset très varié (les 10 classes à générer sont complètement différentes, contrairement à un dataset comme MNIST ou Celeb A), les modèles de diffusion surpassent même les GANs.

En contrepartie, le processus de génération d'images avec un modèle de diffusion implique d'utiliser le modèle en inférence  $T$  fois sur un bruit initial pour aboutir à une image de bonne qualité : il est donc extrêmement lent en comparaison à ce qu'un GAN peut produire. Ceci est problématique pour de la génération d'images en temps réel (en particulier de vidéos). La figure suivante, extraite d'un article de NVIDIA intitulé "the generative learning trilemma" [16], permet de résumer les avantages et inconvénients de chacun des modèles génératifs introduits jusqu'à présent :

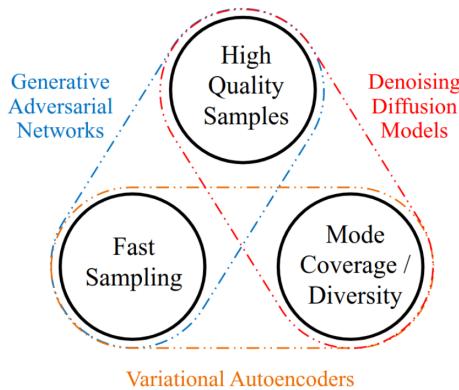


FIGURE 23 – Forces et faiblesses des différents algorithmes génératifs

L'équipe de NVIDIA décrit également dans leur article une architecture mixte entre GAN et modèle de diffusion, appelée *Denoising Diffusion GAN*, tentant de résoudre le problème du temps d'échantillonage. Nous détaillerons d'autres approches plus populaires pour aborder ce problème, qui est actuellement un axe de recherche très actif.

L'étude des modèles de diffusion étant le principalement objet de notre projet fil rouge, seront présentées dans des sections plus importantes deux implémentations de ces types de modèle : celle de l'article initial (*Denoising Diffusion Probabilistic Model*) et celle des Latent Diffusion Models (*Stable Diffusion*).

### 3 Denoising Diffusion Probabilistic Models

#### 3.1 Justifications théoriques

Dans cette section, le but sera de montrer comment, pour le processus de diffusion initial introduit dans l'article *Denoising Diffusion Probabilistic Model*[15] :

- Il est possible de relier le critère de maximisation de la vraisemblance des images d'entraînement  $\mathbf{x}_0$  à une fonction de perte permettant de réaliser l'entraînement d'un réseau de neurones.
- Cette fonction de perte peut s'écrire sous une forme simplifiée **ne faisant intervenir que le bruit** entre deux étapes de bruitage : le modèle ne prédit que le bruit ajouté sur l'image précédente, pas l'image en elle-même

##### 3.1.1 Notations

En reprenant les notations définies à la section 2.3. Diffusion Models, l'équation décrivant le bruitage donnant l'élément  $\mathbf{x}_t$  à partir de l'élément  $\mathbf{x}_{t-1}$  est la suivante. Nous l'écrivons sous forme de loi de probabilité conditionnelle  $q$  dont on tire un échantillon :

$$q(\mathbf{x}_t | \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1 - \beta_t} \mathbf{x}_{t-1}, \beta_t \mathbf{I}) \quad (\text{forward process})$$

Nous observons ainsi un processus markovien, car  $\mathbf{x}_t$  ne dépend explicitement que de  $\mathbf{x}_{t-1}$ . Les paramètres  $\beta_t$  déterminent le *scheduling* du processus de diffusion. De manière générale, le scheduling représente la dynamique du processus de diffusion (forward et backward, qui sont mathématiquement liés). Il décrit notamment la forme des pas de diffusion (ici gaussiens), ainsi que leur nombre (représentés ici par les paramètres  $\beta_t$ ).  $\beta_t$  paramétrise à la fois la moyenne et la variance du bruit ajouté à chaque étape. Il est intéressant de remarquer que choisir un  $\beta_t$  proche de 0 revient à générer une image de moyenne égale à l'image à l'instant précédent, de variance nulle (i.e. à n'ajouter aucun bruit), alors qu'un  $\beta_t$  grand (proche de 1) va échantillonner une valeur de pixel à partir d'une distribution de moyenne nulle et de variance 1 (i.e. générer une image complètement aléatoire, sans lien avec l'image à l'instant précédent). Dans l'article, les auteurs font varier linéairement  $\beta_t$  entre  $10^{-4}$  et  $2 \times 10^{-2}$ , sur une échelle de 1000 pas. Ce choix sera justifié après avoir évoqué le processus inverse.

Une autre notation utile à introduire pour le processus forward est  $\alpha_t = 1 - \beta_t$  et  $\bar{\alpha}_t = \prod_{s=0}^t \alpha_s$ . Ainsi défini, on peut réécrire ce processus sous la forme :

$$\begin{aligned} q(\mathbf{x}_t | \mathbf{x}_{t-1}) &= \sqrt{\alpha_t} \mathbf{x}_{t-1} + \sqrt{1 - \alpha_t} \epsilon_{t-1}^* && \text{avec } \epsilon_{t-1}^* \sim \mathcal{N}(0, 1) \\ &= \sqrt{\alpha_t} (\sqrt{\alpha_{t-1}} \mathbf{x}_{t-2} + \sqrt{1 - \alpha_{t-1}} \epsilon_{t-2}^*) + \sqrt{1 - \alpha_t} \epsilon_{t-1}^* && \text{avec } \epsilon_{t-2}^* \sim \mathcal{N}(0, 1) \\ &= \sqrt{\alpha_t \alpha_{t-1}} \mathbf{x}_{t-2} + \sqrt{\alpha_t - \alpha_t \alpha_{t-1}} \epsilon_{t-2}^* + \sqrt{1 - \alpha_t} \epsilon_{t-1}^* && \text{avec } \epsilon_{t-2} \sim \mathcal{N}(0, 1) \text{ car la somme de 2 vecteurs gaussiens est un vecteur gaussien de variance la somme des 2 variances} \\ &= \sqrt{\alpha_t \alpha_{t-1}} \mathbf{x}_{t-2} + \sqrt{\sqrt{\alpha_t - \alpha_t \alpha_{t-1}}^2 + \sqrt{1 - \alpha_t}^2} \epsilon_{t-2} && \text{avec } \epsilon_{t-2} \sim \mathcal{N}(0, 1) \\ &= \sqrt{\alpha_t \alpha_{t-1}} \mathbf{x}_{t-2} + \sqrt{\cancel{\alpha_t} - \alpha_t \alpha_{t-1} + 1 - \cancel{\alpha_t}} \epsilon_{t-2} && \text{avec } \epsilon_{t-2} \sim \mathcal{N}(0, 1) \\ &= \sqrt{\alpha_t \alpha_{t-1}} \mathbf{x}_{t-2} + \sqrt{1 - \alpha_t \alpha_{t-1}} \epsilon_{t-2} && \text{avec } \epsilon_{t-2} \sim \mathcal{N}(0, 1) \\ &= \dots && \text{avec } \epsilon_{t-2} \sim \mathcal{N}(0, 1) \\ &= \sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon && \text{équivalent à } q(\mathbf{x}_t | \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t} \mathbf{x}_0, (1 - \bar{\alpha}_t) \mathbf{I}) \end{aligned}$$

La distribution de probabilité qui servira à l'échantillonnage est la suivante :

$$p(\mathbf{x}_{t-1} | \mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t)) \quad (\text{reverse process})$$

Le réseau entraîné pourrait être utilisé pour apprendre à générer à la fois la moyenne  $\mu_\theta$  et la matrice de covariance  $\Sigma_\theta$ . En pratique, les auteurs fixent  $\Sigma_\theta(\mathbf{x}_t, t) = \sigma^2 \mathbf{I}$  constante et ne se servent du réseau de neurones que pour prédire  $\mu_\theta$  (d'autres implémentations ne font pas ce choix).

Les données d'entraînement sont désignées par  $\mathbf{x}_0$ , les données entièrement bruitées par  $\mathbf{x}_T$ .

Les paramètres que le modèle devra apprendre à partir de ces images d'entraînement sont pour chaque timestep  $t$  la loi normale, **caractérisée uniquement par sa moyenne**  $\mu_\theta(\mathbf{x}_t, t)$  (la valeur de  $\sigma^2$  est fixée). Le processus forward est fixé à l'avance et ses paramètres ne sont pas entraînés.

Le théorème de Bayes nous permet d'affirmer que  $q(\mathbf{x}_t | \mathbf{x}_{t-1}) \propto p(\mathbf{x}_{t-1} | \mathbf{x}_t) q(\mathbf{x}_{t-1})$ . Après calculs que nous ne

détaillons pas ici, cette relation constraint à devoir considérer le cas  $\beta \rightarrow 0$  pour que le reverse process soit lui aussi gaussien [17].

### 3.1.2 Expression de la ELBO Loss

L'objectif du modèle est de générer l'image la plus probable correspondant au bruit fourni en entrée, c'est-à-dire à maximiser la vraisemblance (ou la log-vraisemblance par monotonie du log) de  $x_0$ , i.e. la quantité  $\log(p(x_0))$ . De même que pour un VAE, cette vraisemblance n'est pas calculable directement, mais on peut la minorer par une expression qui l'est :

$$\begin{aligned}
\log(p(\mathbf{x})) &= \log\left(\int p(\mathbf{x}_{0:T}) d\mathbf{x}_{1:T}\right) \\
&= \log\left(\int p(\mathbf{x}_{0:T}) \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} d\mathbf{x}_{1:T}\right) \\
&= \log\left(\mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\frac{p(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\right]\right) && \text{d'après le théorème de transfert } x \rightarrow q(\mathbf{x}_{1:T}|\mathbf{x}_0) \\
&\geq \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log\left(\frac{p(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\right)\right] && \text{d'après l'inégalité de Jensen (log est convexe)} \\
&= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log\left(\frac{p(\mathbf{x}_T)\prod_{t=1}^T p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{\prod_{t=1}^T q(\mathbf{x}_t|\mathbf{x}_{t-1})}\right)\right] && \text{factorisation sous hypothèse markovienne} \\
&= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log\left(\frac{p(\mathbf{x}_T)p_\theta(\mathbf{x}_0|\mathbf{x}_1)\prod_{t=2}^T p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_1|\mathbf{x}_0)\prod_{t=2}^T q(\mathbf{x}_t|\mathbf{x}_{t-1})}\right)\right] && \text{extraction du 1<sup>er</sup> terme des produits}
\end{aligned}$$

Une astuce de calcul consiste ici à réécrire la transition forward en conditionnant en plus sur  $\mathbf{x}_0$ , ce qui est superflu dans l'hypothèse markovienne mais reste correct :  $q(\mathbf{x}_t|\mathbf{x}_{t-1}) = q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)$ . D'où,

$$\begin{aligned}
\log(p(\mathbf{x})) &\geq \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log\left(\frac{p(\mathbf{x}_T)p_\theta(\mathbf{x}_0|\mathbf{x}_1)\prod_{t=2}^T p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_1|\mathbf{x}_0)\prod_{t=2}^T q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)}\right)\right] \\
&= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log\left(\frac{p(\mathbf{x}_T)p_\theta(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_1|\mathbf{x}_0)}\right) + \log\left(\frac{\prod_{t=2}^T p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{\prod_{t=2}^T q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)}\right)\right]
\end{aligned}$$

Or, d'après le théorème de Bayes,

$$\begin{aligned}
q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0) &= \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)q(\mathbf{x}_t|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_0)} \\
\Rightarrow \prod_{t=2}^T q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0) &= \prod_{t=2}^T \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)q(\mathbf{x}_t|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_0)} \\
&= \frac{q(\mathbf{x}_1|\mathbf{x}_2, \mathbf{x}_0)q(\mathbf{x}_2|\mathbf{x}_0)q(\mathbf{x}_2|\mathbf{x}_3, \mathbf{x}_0)q(\mathbf{x}_3|\mathbf{x}_0)\dots q(\mathbf{x}_{T-1}|\mathbf{x}_0)q(\mathbf{x}_{T-1}|\mathbf{x}_T, \mathbf{x}_0)q(\mathbf{x}_T|\mathbf{x}_0)}{q(\mathbf{x}_1|\mathbf{x}_0)q(\mathbf{x}_2|\mathbf{x}_0)q(\mathbf{x}_3|\mathbf{x}_0)\dots q(\mathbf{x}_{T-1}|\mathbf{x}_0)} \\
&= \frac{q(\mathbf{x}_T|\mathbf{x}_0)}{q(\mathbf{x}_1|\mathbf{x}_0)} \prod_{t=2}^T q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)
\end{aligned}$$

Ainsi,

$$\begin{aligned}
\log(p(\mathbf{x})) &\geq \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log\left(\frac{p(\mathbf{x}_T)p_\theta(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_1|\mathbf{x}_0)}\right) + \log\left(\frac{q(\mathbf{x}_1|\mathbf{x}_0)}{q(\mathbf{x}_T|\mathbf{x}_0)}\right) + \log\left(\prod_{t=2}^T \frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}\right)\right] \\
&= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log\left(\frac{p(\mathbf{x}_T)p_\theta(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_T|\mathbf{x}_0)}\right) + \sum_{t=2}^T \log\left(\frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}\right)\right] \\
&= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}[\log(p_\theta(\mathbf{x}_0|\mathbf{x}_1))] + \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}[\log\left(\frac{p(\mathbf{x}_T)}{q(\mathbf{x}_T|\mathbf{x}_0)}\right)] + \sum_{t=2}^T \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}[\log\left(\frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}\right)]
\end{aligned}$$

Cette dernière équation peut être simplifiée en ne conservant que les variables contenues dans la distribution à laquelle on applique l'espérance.

En effet,

$$\begin{aligned}
\mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}[\log(p_\theta(\mathbf{x}_1|\mathbf{x}_0))] &= \int_{\mathbf{x}_{1:T}} q(\mathbf{x}_{1:T}|\mathbf{x}_0) \log(p_\theta(\mathbf{x}_1|\mathbf{x}_0)) d\mathbf{x}_{1:T} \\
&= \int_{\mathbf{x}_{1:T}} q(\mathbf{x}_1|\mathbf{x}_0) q(\mathbf{x}_{2:T}|\mathbf{x}_1, \mathbf{x}_0) \log(p_\theta(\mathbf{x}_1|\mathbf{x}_0)) d\mathbf{x}_{1:T} \\
&= \int_{\mathbf{x}_1} q(\mathbf{x}_1|\mathbf{x}_0) \log(p_\theta(\mathbf{x}_1|\mathbf{x}_0)) d\mathbf{x}_1 \underbrace{\int_{\mathbf{x}_{2:T}} q(\mathbf{x}_{2:T}|\mathbf{x}_1, \mathbf{x}_0) d\mathbf{x}_{2:T}}_{=1 \text{ par définition}} \\
&= \int_{\mathbf{x}_1} q(\mathbf{x}_1|\mathbf{x}_0) \log(p_\theta(\mathbf{x}_1|\mathbf{x}_0)) d\mathbf{x}_1 \\
&= \mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}[\log(p_\theta(\mathbf{x}_1|\mathbf{x}_0))]
\end{aligned}$$

D'où

$$\begin{aligned}
\log(p(\mathbf{x})) &\geq \mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}[\log(p_\theta(\mathbf{x}_0|\mathbf{x}_1))] + \mathbb{E}_{q(\mathbf{x}_T|\mathbf{x}_0)}[\log(\frac{p(\mathbf{x}_T)}{q(\mathbf{x}_T|\mathbf{x}_0)})] + \sum_{t=2}^T \mathbb{E}_{q(\mathbf{x}_t, \mathbf{x}_{t-1}|\mathbf{x}_0)}[\log(\frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)})] \\
\Rightarrow \log(p(\mathbf{x})) &\geq \underbrace{\mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}[\log(p_\theta(\mathbf{x}_0|\mathbf{x}_1))]}_{\text{reconstruction term}} - \underbrace{D_{\text{KL}}(q(\mathbf{x}_T|\mathbf{x}_0)||p(\mathbf{x}_T))}_{\text{prior matching term}} - \underbrace{\sum_{t=2}^T \mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)}[D_{\text{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)||p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))]}_{\text{denoising matching term}}
\end{aligned}$$

L'expression ainsi obtenue est la Evidence Lower BOund (ELBO) des modèles de diffusion. Elle est composée de trois termes :

- Reconstruction term : Mesure la qualité de reconstruction des images originales à partir du premier état latent ( $\mathbf{x}_1$ ). Comme pour un VAE, ce terme peut être optimisé ou approché par méthode de Monte Carlo
- Prior matching term : Ce terme ne contient pas de paramètre entraînable, il représente juste la proximité de la distribution finale bruitée  $q(\mathbf{x}_T|\mathbf{x}_0)$  par rapport au prior gaussien choisi  $p(\mathbf{x}_T)$ . Il s'agit donc d'une constante qui peut être ignorée dans la fonction de perte
- Denoising matching term : Minimiser ce terme revient à réduire l'écart entre la distribution  $q$  (spécifiquement construite à partir des paramètres  $\beta$ ) utilisée pour bruiter les images, et la distribution  $p_\theta$  que le réseau cherche à apprendre pour débruiter les images en passant d'un état  $t-1$  à  $t$ . C'est ce terme qui a le plus d'impact dans la loss (somme de termes contenant les paramètres d'entraînement)

### 3.1.3 Simplification de la fonction de perte

Le Denoising matching term est donc une divergence de Kullback-Leibler entre deux distributions. La distribution  $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$  est gaussienne dans l'hypothèse initiale faite par les auteurs, et la distribution  $q(\mathbf{x}_t|\mathbf{x}_{t-1})$  est gaussienne par construction. Or, la divergence de Kullback-Leibler entre deux variables aléatoires gaussiennes de même dimension  $d$  prend la forme analytique simple suivante :

$$D_{KL}(\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_x, \boldsymbol{\Sigma}_x) || \mathcal{N}(\mathbf{y}; \boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y)) = \frac{1}{2} [\log \frac{|\boldsymbol{\Sigma}_y|}{|\boldsymbol{\Sigma}_x|} - d + \text{tr}(\boldsymbol{\Sigma}_y^{-1} \boldsymbol{\Sigma}_x) + (\boldsymbol{\mu}_y - \boldsymbol{\mu}_x)^\top \boldsymbol{\Sigma}_y^{-1} (\boldsymbol{\mu}_y - \boldsymbol{\mu}_x)]$$

Il semble donc raisonnable de vérifier si  $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$  est gaussien pour utiliser cette forme, et alléger l'expression de la ELBO loss. Le théorème de Bayes nous permet une fois de plus de réécrire le processus forward

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \frac{q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_t|\mathbf{x}_0)}$$

Dans la section 3.1.1. Notations, nous avons déjà montré que  $q(\mathbf{x}_t|\mathbf{x}_0)$  est gaussien de moyenne  $\sqrt{\bar{\alpha}_t}$  et de variance  $(1 - \bar{\alpha}_t)\mathbf{I}$  lors de l'introduction de la notation  $\bar{\alpha}_t$ . D'où

$$\begin{aligned}
q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) &= \frac{q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_t|\mathbf{x}_0)} \\
&= \frac{\mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \quad \mathcal{N}(\mathbf{x}_{t-1}; \sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0, (1 - \bar{\alpha}_{t-1})\mathbf{I})}{\mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1 - \bar{\alpha}_t)\mathbf{I})}
\end{aligned}$$

i.e.

$$\begin{aligned}
q(\mathbf{x}_{t-1} | \mathbf{x}_t, \mathbf{x}_0) &\propto \exp\left(-\frac{1}{2}\left(\frac{(\mathbf{x}_t - \sqrt{\alpha_t} \mathbf{x}_{t-1})^2}{\beta_t} + \frac{(\mathbf{x}_{t-1} - \sqrt{\bar{\alpha}_{t-1}} \mathbf{x}_0)^2}{1 - \bar{\alpha}_{t-1}} - \frac{(\mathbf{x}_t - \sqrt{\bar{\alpha}_t} \mathbf{x}_0)^2}{1 - \bar{\alpha}_t}\right)\right) \\
&= \exp\left(-\frac{1}{2}\left(\frac{\mathbf{x}_t^2 - 2\sqrt{\alpha_t} \mathbf{x}_t \mathbf{x}_{t-1} + \alpha_t \mathbf{x}_{t-1}^2}{\beta_t} + \frac{\mathbf{x}_{t-1}^2 - 2\sqrt{\bar{\alpha}_{t-1}} \mathbf{x}_0 \mathbf{x}_{t-1} + \bar{\alpha}_{t-1} \mathbf{x}_0^2}{1 - \bar{\alpha}_{t-1}} - \frac{(\mathbf{x}_t - \sqrt{\bar{\alpha}_t} \mathbf{x}_0)^2}{1 - \bar{\alpha}_t}\right)\right) \\
&= \exp\left(-\frac{1}{2}\left(\left(\frac{\alpha_t}{\beta_t} + \frac{1}{1 - \bar{\alpha}_{t-1}}\right) \mathbf{x}_{t-1}^2 + \left(\frac{2\sqrt{\alpha_t}}{\beta_t} \mathbf{x}_t + \frac{2\sqrt{\bar{\alpha}_{t-1}}}{1 - \bar{\alpha}_{t-1}} \mathbf{x}_0\right) \mathbf{x}_{t-1} + C(\mathbf{x}_t, \mathbf{x}_0)\right)\right) \quad \text{avec } C \text{ cte w.r.t. } \mathbf{x}_{t-1}
\end{aligned}$$

Il est ensuite possible de montrer que  $q(\mathbf{x}_{t-1} | \mathbf{x}_t, \mathbf{x}_0)$  suit une loi gaussienne, de moyenne  $\mu_q(\mathbf{x}_t, \mathbf{x}_0)$ , et d'écart-type  $\Sigma_q(t)$  dépendant tous les deux des  $\alpha_t$ . Les calculs sont lourds, et impliquent de passer par les densités des 3 distributions de probabilités. L'auteur obtient les résultats suivants pour la distribution :

$$\mathcal{N}(\mathbf{x}_{t-1}, \underbrace{\frac{\sqrt{\alpha_t}(1 - \bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1 - \alpha_t)\mathbf{x}_0}{1 - \bar{\alpha}_t}, \underbrace{\frac{(1 - \alpha_t)(1 - \bar{\alpha}_{t-1})}{1 - \bar{\alpha}_t} \mathbf{I}}_{\Sigma_q(t)}})$$

FIGURE 24 – Paramètres de la distribution "ground truth"

Nous pouvons donc enfin calculer analytiquement la divergence de Kullback-Leibler entre notre distribution "ground truth". Rappelons à ce stade que la seule différence entre les deux distributions est la connaissance de  $\mathbf{x}_0$  !

Nous faisons une hypothèse supplémentaire, celle d'homoscédasticité entre ces deux distributions, c'est-à-dire que leurs matrices de covariances sont les mêmes. Cela permet de simplifier la fonction à optimiser. Ainsi, la minimisation de la divergence KL entre les deux distributions revient au problème d'optimisation suivant :

$$\operatorname{argmin}_{\theta} \frac{1}{\sigma_q^2} \|\mu_{\theta} - \mu_q\|^2$$

Si on reprend la formule pour  $\mu_q$ , on constate bien que la seule quantité inconnue est  $\mathbf{x}_0$ . On va donc chercher à estimer cette quantité par un réseau de neurones  $\hat{x}_{\theta}(\mathbf{x}_t, t)$  (en général un U-Net dans ce cas).

La moyenne qu'on va apprendre sera donc :

$$\mu_{\theta}(\mathbf{x}_t, t) = \frac{\sqrt{\alpha_t}(1 - \bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1 - \alpha_t)\hat{x}_{\theta}(\mathbf{x}_t, t)}{1 - \bar{\alpha}_t}$$

FIGURE 25 – Moyenne de la distribution backward à apprendre

En remplaçant ces deux moyennes dans l'objectif d'optimisation précédent  $\operatorname{argmin}_{\theta} \frac{1}{\sigma_q^2} \|\mu_{\theta} - \mu_q\|^2$ , on obtient la fonction de perte suivant, toujours à minimiser :

$$\arg \min_{\theta} \frac{1}{2\sigma_q^2(t)} \frac{\bar{\alpha}_{t-1}(1 - \alpha_t)^2}{(1 - \bar{\alpha}_t)^2} \left[ \|\hat{x}_{\theta}(\mathbf{x}_t, t) - \mathbf{x}_0\|_2^2 \right]$$

FIGURE 26 – Fonction de perte

Le réseau de neurones  $\hat{x}_{\theta}(\mathbf{x}_t, t)$  dépend du temps et de l'image à un niveau de corruption arbitraire : il devra donc apprendre à prédire l'image originale à partir de n'importe quel niveau de bruit. La fonction de perte est au final une MSE Loss (à un facteur dépendant des  $\alpha_t$  près) entre l'image originale et l'image reconstruite par le réseau de neurones à partir de sa version corrompue à un temps  $t$ .

Ainsi, en pratique, la loss va être calculée à partir de la prédiction à un niveau de bruit arbitraire, choisi selon une loi uniforme sur les timesteps. On utilise par dessus une descente de gradient stochastique (SGD), ce qui signifie qu'on choisit d'abord aléatoirement l'image qui va passer dans le réseau.

<b>Algorithm 1</b> Training	<b>Algorithm 2</b> Sampling
<pre> 1: <b>repeat</b> 2:   <math>\mathbf{x}_0 \sim q(\mathbf{x}_0)</math> 3:   <math>t \sim \text{Uniform}(\{1, \dots, T\})</math> 4:   <math>\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})</math> 5:   Take gradient descent step on       <math>\nabla_{\theta} \ \epsilon - \epsilon_{\theta}(\sqrt{\alpha_t} \mathbf{x}_0 + \sqrt{1 - \alpha_t} \epsilon, t)\ ^2</math> 6: <b>until</b> converged </pre>	<pre> 1: <math>\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})</math> 2: <b>for</b> <math>t = T, \dots, 1</math> <b>do</b> 3:   <math>\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})</math> if <math>t &gt; 1</math>, else <math>\mathbf{z} = \mathbf{0}</math> 4:   <math>\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} (\mathbf{x}_t - \frac{1 - \alpha_t}{\sqrt{1 - \alpha_t}} \epsilon_{\theta}(\mathbf{x}_t, t)) + \sigma_t \mathbf{z}</math> 5: <b>end for</b> 6: <b>return</b> <math>\mathbf{x}_0</math> </pre>

FIGURE 27 – Algorithmes d'entraînement et d'échantillonage d'un modèle de diffusion

On retrouve bien ces étapes dans l'algorithme 1. On choisit parmi les images d'entraînement une image  $\mathbf{x}_0$ , puis un niveau de bruit  $t$ .

La fonction de loss présentée ici, extraite de l'article [15], est améliorée par un reparametrization trick, qui fait que le réseau de neurones apprend à prédire un bruit  $\epsilon$  qui caractérise parfaitement (via un reparametrization trick) l'image originale.

Le deuxième algorithme est celui utilisé pour réaliser l'échantillonnage. Une fois que les paramètres sont appris, on part d'une image aléatoire, puis on suit le processus inverse (lui aussi reparamétrisé). Cela permet finalement d'obtenir une image débruitée en  $\mathbf{x}_0$ .

Pour terminer, nous proposons une vue d'ensemble qui relie les concepts qui sont rentrés en jeu pour définir une fonction de perte, et par conséquent un processus de training/sampling.

## 3.2 Implémentation

### 3.2.1 Architecture initiale

Comme expliqué de manière théorique, le réseau de neurones entraîné doit prédire le bruit contenu dans une image à partir d'une image bruitée fournie en entrée. Le tenseur en sortie du réseau doit donc être un tenseur de même dimension que celui d'entrée. Pour cette tâche, les auteurs de l'article DDPM ont donc naturellement utilisé un modèle basé sur l'architecture d'un réseau U-Net [18], en s'inspirant de l'architecture de PixelCNN++ [19]. Le schéma ci-dessous présente ce schéma avec les paramètres utilisés dans l'article :

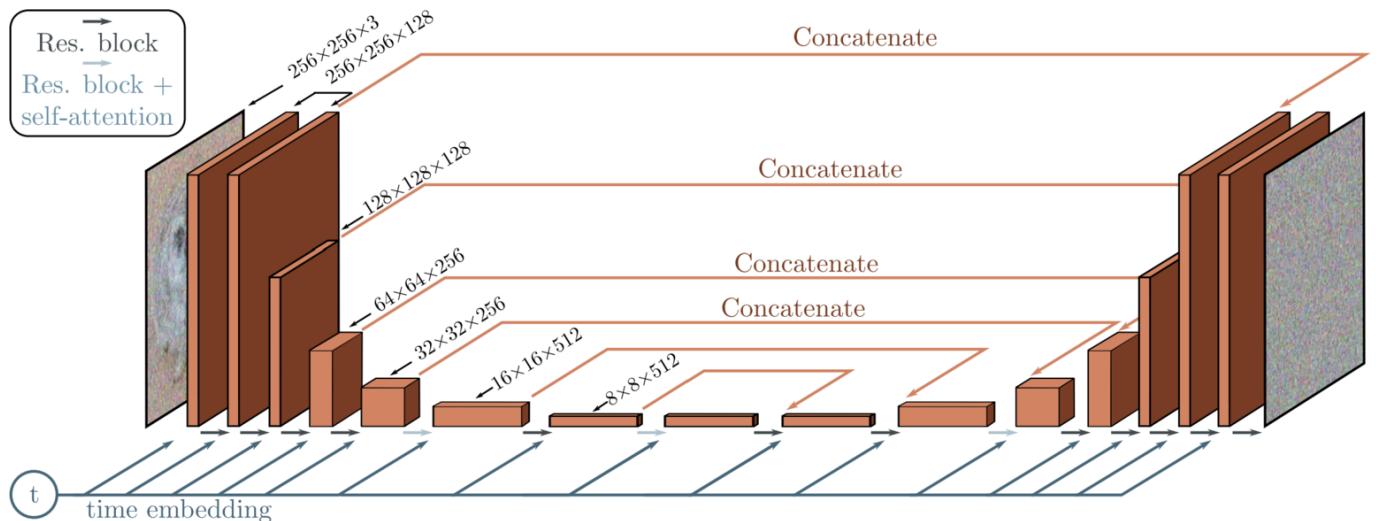


FIGURE 28 – Architecture UNet du modèle DDPM [20]

Plusieurs modifications ont été apportées à l'architecture originale du U-Net dans le contexte des modèles de diffusions.

La première modification importante est l'utilisation de modules de **Self-Attention à 4 attention heads à 2 endroits dans le réseau**, lorsque la résolution des features maps est de 16 x 16 pixels (représentés en gris-clair dans la figure ci-dessus). L'équivalent des text embeddings que l'on rencontre en NLP sont ici des "embeddings de pixels" dont la dimension est le nombre de features maps de la couche considérée, comme illustré sur la figure 29. Ces modules de self-attention participent aux bons résultats du modèle (de manière difficile à quantifier), mais seront surtout essentiels pour permettre le conditionnement de la génération d'images par du texte via cross attention dans les modèles s'inspirant de cette implémentation.

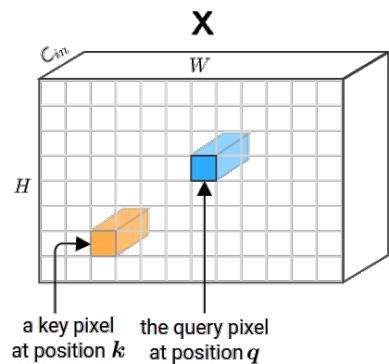


FIGURE 29 – Self-Attention dans un réseau convolutionnel [21]

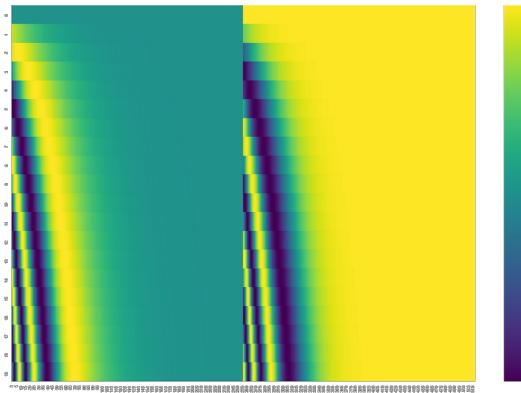


FIGURE 30 – Visualisation des time embeddings utilisés (abscisses : feature map  $c$  parmi les  $C$  feature maps de la couche considérée, ordonnées : step de diffusion  $t$ ) [22]

Une deuxième modification, elle aussi inspirée de l'architecture des Transformers [23], est l'emploi de **Positional Embeddings** pour encoder l'instant  $t$  (et donc le niveau de bruit) auquel l'image en entrée du réseau est associée. L'encoding est fait de manière sinusoïdal : sinus sur la première moitié de dimension d'embeddings, cosinus sur la seconde, comme illustré sur la figure 30 pour 20 embeddings de dimension 512 (les couleurs sombres sont proches de 0, les couleurs jaunes proches de 1). À un instant  $t$ , pour une couche avec  $C$  feature maps, le tenseur de positional encoding est la concaténation des 2 tenseurs suivants :

$$\begin{cases} P_{\sin}(t, C) = \sin\left(\frac{\lfloor t/2 \rfloor}{f_{inv}}\right) \\ P_{\cos}(t, C) = \cos\left(\frac{\lfloor t/2 \rfloor}{f_{inv}}\right) \end{cases} \quad \text{avec } f_{inv} = \frac{1}{10000^c}, c \in [0, 2, 4, \dots, C]$$

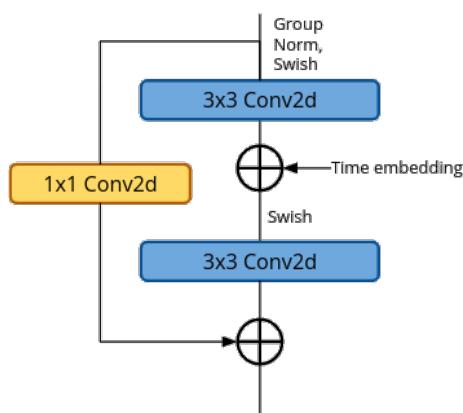


FIGURE 31 – Focus sur un bloc ResNet

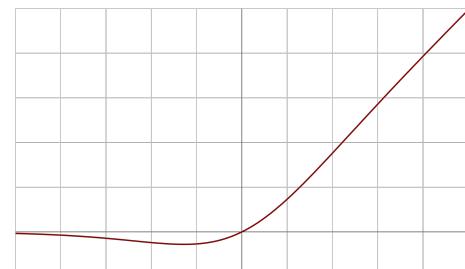


FIGURE 32 – Fonction swish

Une troisième modification est l'utilisation de **blocs ResNet** à la place de simples convolutions 2D dans les blocs du UNet (représentés en marron sur la figure 28). La convolution 2D de kernel 1x1 pixel permet d'ajuster le nombre de feature maps de la skip layer par rapport à l'enchaînement des 2 couches convolution 2D de kernel 3x3 pixels. Le positional encoding est ajouté entre les 2 couches de convolution formant le bloc ResNet. Par rapport à PixelCNN++, la normalisation utilisée est group normalization au lieu de weight normalization, pour simplifier l'implémentation. La fonction d'activation utilisée est swish, définie ainsi :

$$\text{swish}(x) = x \text{ sigmoid}(\beta x) \text{ avec } \beta \text{ valant 1 par défaut.}$$

Depuis ce premier modèle, de nombreux articles ont repris ce U-Net en en modifiant certains aspects pour améliorer les performances ou simplifier l'implémentation, notamment :

- Modification des dimensions et nombre de couches (réduction de la largeur et augmentation de la profondeur)
- Ajout de modules de self-attention supplémentaires
- Augmentation du nombre d'attention heads
- Modification de l'architecture des blocs ResNet
- Modification de la fonction d'activation
- Alternatives à la group normalization
- etc.

### 3.2.2 Résultats

Pour l'implémentation, nous nous sommes inspirés du [repository GitHub de awjuliani](#), qui a le mérite d'être plus simple à apprécier que celle de [Phillip Wang](#), bien que moins fidèle au modèle initial. Nous avons testé cette implémentation sur les datasets CIFAR-10 et DTD. Par rapport à l'article initial,

- Nous avons ajouté une couche de self-attention au milieu du U-Net, lorsque la résolution des images est de 8x8 pixels
- La fonction d'activation utilisée est une Gaussian Error Linear Unit (GELU) [24], dont l'expression est la suivante :

$$\text{GELU}(x) = x \Phi(x) \text{ avec } \Phi \text{ fonction de répartition gaussienne standard}$$

$$\text{ou GELU}(x) = x \frac{1}{2} [1 + \text{erf}(\frac{x}{\sqrt{2}})]$$

Cette fonction peut être approximée par la fonction swish avec  $\beta = 1.702$  (i.e.  $\text{GELU}(x) \sim x\sigma(1.702x)$ )

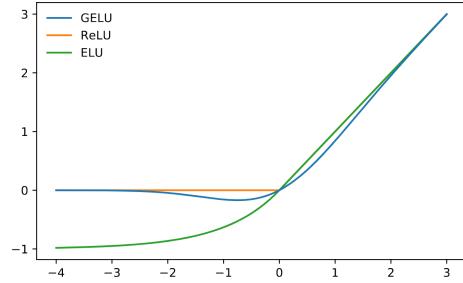


FIGURE 33 – Comparaison GELU, ReLU et ELU

- Les premières couches de convolution amènent à la création de 64 features maps et non 128 (mais le bottleneck possède bien 512 feature maps). Pour le dataset DTD, les images étant de plus grandes dimensions, et de dimension supérieure à CIFAR-10, elles ont toutes été redimensionnées en résolution 256 x 256 pixels (via RandomCrop lors de la phase de data augmentation dans la création du Dataset Pytorch) et les dimensions  $H$  et  $W$  du bottleneck ont été ajustées pour les couches d'attention.

Les premiers résultats obtenus sur CIFAR-10 après 1000 epochs d'entraînement sont affichés ci-dessous (figure 34). Les images sont sauvegardées à différents instants  $t$  durant le processus d'échantillonnage pour montrer le débruitage progressif



FIGURE 34 – Évolution de la génération de 9 images à différents instants  $t$  (CIFAR-10)

Les images générées sont de plutôt bonne qualité par rapport aux images du jeu d’entraînement, surtout lorsqu’on compare ces résultats à ceux obtenus précédemment avec des GANs. En revanche, les couleurs sont très pâles. Les images n’ont pas été normalisées avant l’entraînement du modèle utilisé, ce qui suggère que l’intérêt de la normalisation réside plus dans la qualité des images générées que dans la facilité d’entraînement, tout du moins pour ce dataset simple. La courbe d’évolution de la perte fait également penser que le nombre d’epochs d’entraînement est inutilement élevé.

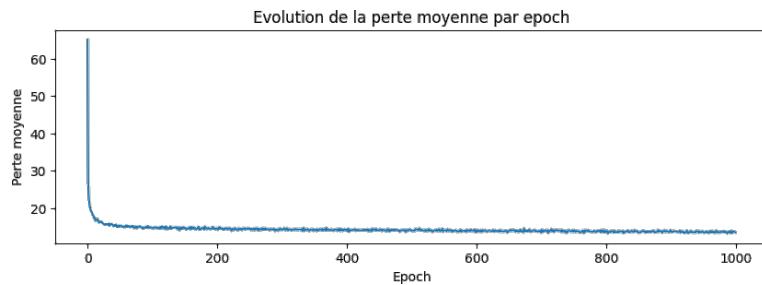


FIGURE 35 – Évolution de la perte moyenne au cours de l'entraînement (CIFAR-10)

Un autre constat est qu'un nombre important de pas de diffusion doivent être effectués pour qu'un niveau de bruit suffisant soit retiré. En effet, on ne commence pas à apercevoir l'image finale avant le pas 650. Ce même constat est à l'origine de propositions faites dans certains articles, notamment Improved DDPM [25], de modifier l'échelle linéaire de scheduling des  $\beta_t$  du processus forward par une échelle cosinus (fonction en cosinus<sup>2</sup> en réalité), permettant d'arriver plus rapidement à des images débruitées de bonne qualité.



FIGURE 36 – Comparaison d'un scheduling linéaire (en haut) et cosinus (en bas) pour un même nombre  $t$  de pas de diffusion

Nous avons réentraîné un modèle DDPM sur CIFAR-10 en normalisant les images. Ci-dessous sont affichées 20 images générées avec ce modèle après 350 epochs d'entraînement et 1000 pas d'échantillonnage. On peut remarquer que le problème de couleurs a disparu, les couleurs semblent même saturées.



FIGURE 37 – Images obtenues après normalisation des images (CIFAR-10)

Les résultats obtenus sur le dataset DTD sont également satisfaisants. On constate que la saturation des couleurs augmente au cours de l'entraînement, le modèle précédemment entraîné sur CIFAR-10 a donc très probablement overfitté les données. Une des limites du modèles semble cependant être la diversité des images générées. En effet, le dataset DTD comporte 47 classes diverses, mais les images générées semblent toutes être similaires. Nous verrons par la suite que le conditionnement par du texte permet de s'assurer que le modèle est bien capable de générer des images de l'ensemble de ces classes.



TABLE 4 – Images générées à partir d'un même bruit initial pour différentes epochs d'entraînement avec 1000 steps de diffusion (DTD). Les caractéristiques de haut niveau sont déjà observables sur les images générées après 50 epochs. La poursuite de l'entraînement entraîne l'apparition de détails / hautes fréquences, et une saturation des couleurs

### 3.3 DDIM

## 4 Latent Diffusion Models

Après avoir étudié les premiers modèles de diffusion via l'article DDPM, nous nous sommes penchés sur les modèles permettant de conditionner la génération d'images. Le conditionnement par du texte est actuellement la méthode la plus populaire et la plus simple à mettre en place. Nous nous sommes particulièrement intéressés aux Latent Diffusion Models, qui réalisent les étapes de diffusion dans un espace latent, dont Stable Diffusion est une implémentation open source (code et poids d'entraînement disponibles sur GitHub et HuggingFace), facilement utilisable et personnalisable.

### 4.1 Fonctionnement de Stable Diffusion

La principale motivation derrière les Latent Diffusion Models, proposés en 2021 dans l'article *High-Resolution Image Synthesis with Latent Diffusion Models* [26] de l'Université de Munich, est de rendre l'entraînement et l'utilisation des modèles de diffusion moins exigeants en ressources de calcul, en particulier pour des images en haute résolution. La solution présentée consiste à réaliser les pas de diffusion dans l'espace latent d'un autoencodeur, au lieu des effectuer sur les images (décris comme "pixel space" dans l'article). Les tenseurs en entrée du U-Net sont donc de faible dimension par rapport aux images initiales, ce qui :

- Accélère l'entraînement / le finetuning du U-Net, et réduit les besoins en RAM des GPUs utilisés pendant l'entraînement
- Permet d'écourter la durée d'échantillonage, **indépendamment du scheduler utilisé pour les  $\beta_t$** , car chaque passe dans le U-Net est plus rapide. Cette accélération de l'inférence se cumule donc avec toute amélioration de scheduling

Ce deuxième point est particulièrement intéressant car, comme déjà évoqué, le principal point faible des modèles de diffusion est l'important temps d'échantillonnage nécessaire pour générer une image.

Une autre amélioration apportée par cette article est une proposition de **mécanisme de conditionnement multimodal via cross-attention**. La figure 38 présente les 3 composantes du modèle Stable Diffusion : autoencodeur sur la gauche, modèle de diffusion au centre et conditionnement sur la droite.

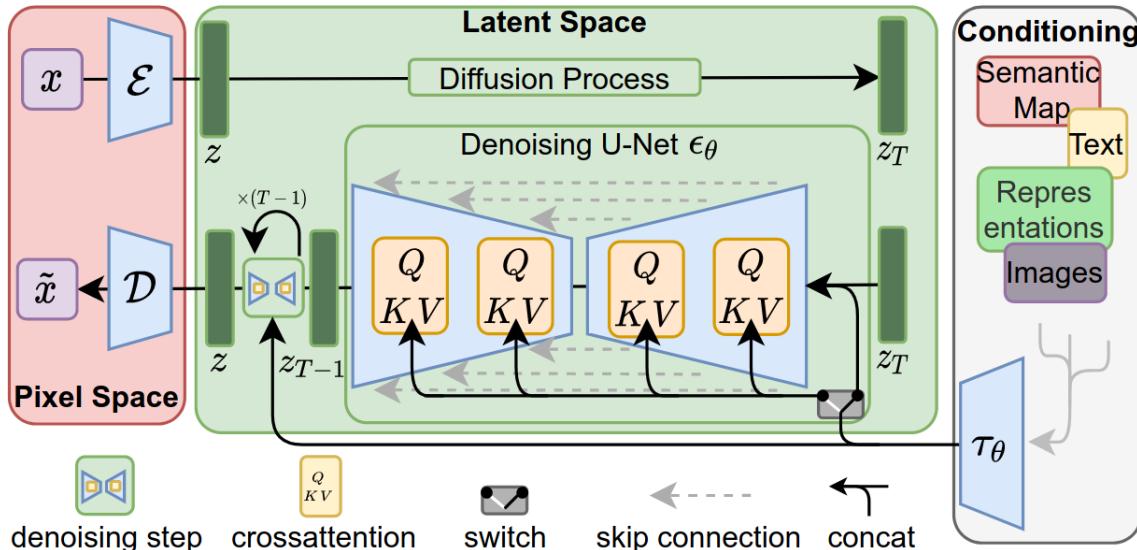


FIGURE 38 – Architecture simplifiée du modèle Stable Diffusion

#### 4.1.1 Compression perceptuelle des images

L'utilisation d'un autoencodeur permet logiquement de réduire la dimension des images de travail, mais introduit également de nouveaux défis lors de l'entraînement du modèle :

- Comment garantir que les images générées soient de qualité équivalente à des images générées sans autoencoder ?
- Comment garantir que l'entraînement global du modèle soit simplifié - un des objectifs principaux de l'article reste de démocratiser la synthèse d'images de haute résolution - malgré l'ajout de 2 modèles supplémentaires (encodeur et décodeur) ?

Pour adresser ces problèmes, les auteurs proposent un entraînement en 2 étapes :

1. Entraînement d'un autoencodeur variationnel (discret dans certains cas) de manière adversariale
2. Entraînement du modèle de diffusion dans l'espace latent (avec les poids de l'autoencodeur gelés)

L'intuition derrière cette proposition vient d'une analyse faite sur la capacité de compression d'un modèle de diffusion entraîné sur CelebA-HQ dans l'article *DDPM*.



FIGURE 39 – Images générées par un DDPM à partir d'une même image bruitée, dans le quadrant en bas à droite, pour différents niveaux de bruit initial

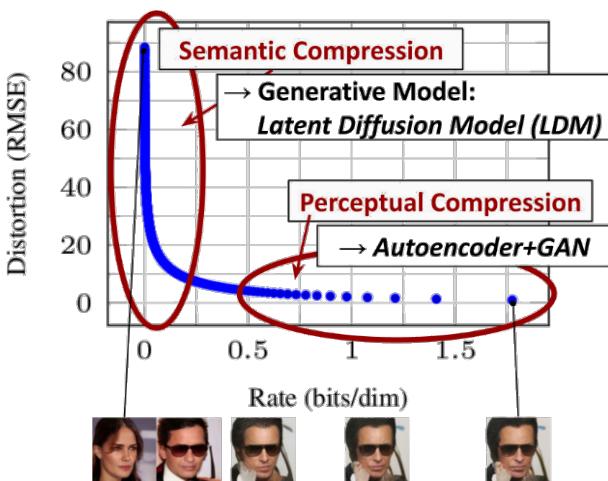


FIGURE 40 – Rate-distortion plot

La courbe 40 emploie des termes issus de la théorie de l'information pour décrire les capacités de compression d'un modèle de diffusion entraîné. Dans notre cas, la distorsion est la mesure de la Root Mean Square Error, donc de l'écart calculé au niveau des pixels, entre les images générées et l'image initiale (Share  $x_0$  en bas à droite sur la figure 39). Pour chaque bloc de 4 images, la source est Share  $x_0$ , l'image fournie en entrée (source bruitée entre 0 et 1000 fois) est le récepteur, et l'image générée correspond à la décompression avec pertes de la source.

Le taux (*rate*), calculé ici comme la divergence  $KL$  entre  $q$  et  $p$  normalisée par le nombre de pixels dans une image, correspond à la quantité d'information fournie en entrée du modèle pour reconstruire l'image : des images très bruitées comme Share  $x_{1000}$  ou Share  $x_{750}$ , qui semblent entièrement aléatoires, ont donc un taux proche de 0, alors que l'image Share  $x_0$ , qui contient déjà toute l'information de la source, a un taux maximal.

La courbe ci-dessus illustre bien que dans une image, la majorité des pixels contient peu d'information sémantique. La distorsion diminue de manière drastique dans la zone de taux faible. Ceci est interprété comme une phase de *compression sémantique/conceptuelle* : l'ajout de peu d'informations (Share  $x_{750}$  ressemble toujours à une image composée uniquement de bruit) permet au modèle de générer des features de haut niveau fidèles à l'image initiale. A contrario, la zone de faible distorsion correspond à une *compression perceptuelle* : un gros apport d'information supplémentaire ne se traduit que par une modification des hautes-fréquences de l'image et un apport faible en information sémantique.

L'allure fortement non linéaire de la courbe laisse à penser qu'il est possible de découpler ces deux étapes de compression pendant l'entraînement et l'inférence du modèle. Les auteurs proposent d'utiliser un autoencodeur entraîné de manière adversariale pour réaliser la compression perceptuelle, car ce type de modèle est très efficace sur cette tâche. Le niveau de compression n'a pas besoin d'être très élevé, car les modèles de diffusion sont de toute évidence également capables de réaliser de la compression perceptuelle, mais suffisant pour un allégement des ressources de calcul nécessaires au modèle de diffusion. Les 2 tâches de compression étant orthogonales, il devrait être possible d'entraîner les deux modèles séparément, ce qui permet d'expérimenter plus facilement sur différentes architectures du modèles de diffusion une fois l'autoencodeur entraîné.

Le protocole d'entraînement de l'autoencodeur s'inspire de celui réalisé dans l'article VQGAN [13] (dont Rombach est également l'un des auteurs principaux). La fonction de perte utilisée prend cette forme :

$$L_{\text{autoencodeur}} = \min_{\mathcal{E}, \mathcal{D}} \max_{\Psi} (L_{\text{rec}}(x, \mathcal{D}(\mathcal{E}(x))) - L_{\text{adv}}(\mathcal{D}(\mathcal{E}(x))) + \log(D_\Psi) + L_{\text{reg}}(x; \mathcal{E}, \mathcal{D}))$$

où les différentes fonctions de perte intermédiaires sont définies comme suit :

- $L_{rec}$  : Loss de reconstruction, qui force l'image reconstruite  $\mathcal{D}(\mathcal{E}(x))$  à être proche de l'image initiale  $x$  et inclut une loss perceptuelle [27]
- $L_{adv} + \log(D_\Psi)$  : Loss adversariale
- $L_{reg}$  : Régularisation. 2 types de régularisation testés : KL et VQ

Les auteurs de l'article ont entraîné plusieurs autoencodeurs en testant différents facteurs de compression  $f \in \{1, 2, 4, 8, 16, 32\}$  sur ImageNet. Les résultats sont présentés sur la figure 41. La notation LDM- $f$  indique que le modèle a été entraîné avec un facteur de compression  $f$ . Le facteur 1 indique un entraînement dans l'espace des pixels, i.e. sans utiliser d'autoencodeur.

Pour un nombre de steps d'entraînement fixé à 2 millions, les modèles avec peu ou pas de compression n'ont pas encore convergé (LDM-1 et 2), ceux avec une trop forte compression ont vite convergé vers une mauvaise qualité d'images. Le meilleur compromis semble donc être un facteur compris entre 4 et 16 (d'autres tests réalisés par les auteurs les font finalement opter pour un facteur  $f = 8$ ).

Pour de la génération text-to-image (seul cas d'usage que nous ayons testé), Stable Diffusion utilise un autoencodeur entraîné :

1. Avec un **facteur de compression de 8**
2. Avec une **régularisation de Kullback-Leibler** et un coefficient de régularisation assez faible de  **$10^{-6}$**
3. Sur le **dataset OpenImages** [28]. Il s'agit d'un dataset contenant 9.2 millions d'images annotées permettant à la fois des tâches de classification d'images, de détection d'objets et de détection de relations visuelles, ce qui explique son intérêt pour entraîner un modèle multimodal

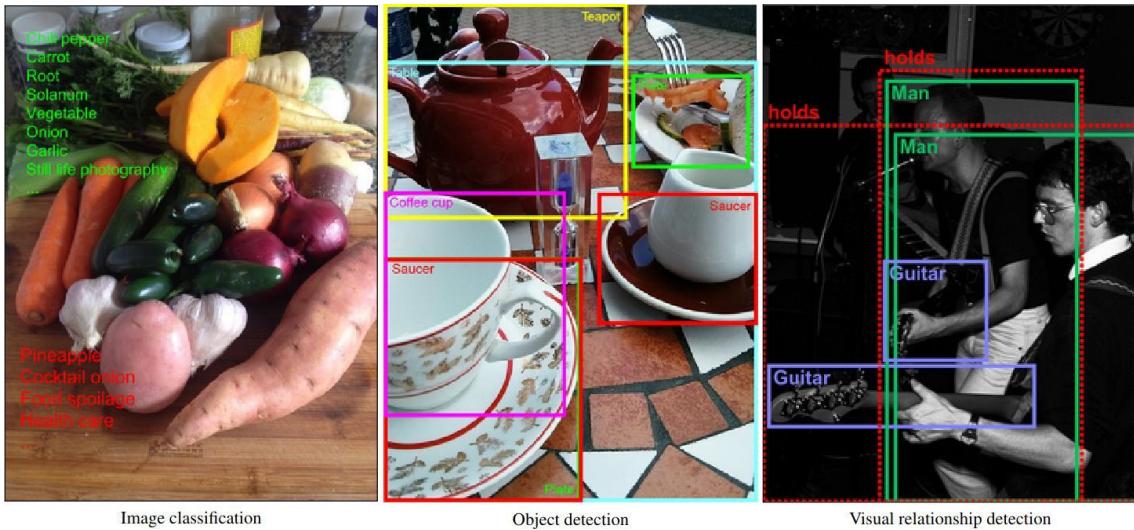


FIGURE 42 – Exemple d'images annotées issues du dataset OpenImages

D'autres autoencodeurs entraînés avec des facteurs de compression différents et/ou avec une régularisation QV sont également disponibles sur [le GitHub de Stable Diffusion v1](#), mais nous ne les avons pas testé.

Les images en sortie du décodeur sont de dimension 512x512 avec Stable Diffusion v1, et 768x768 avec StableDiffusion v2. Pour une image d'entrée de dimension  $H \times W \times 3$ , les tenseurs de l'espace latent sont de dimension  $H/f \times W/f \times 4$ .

#### 4.1.2 Mécanisme de conditionnement

Stable Diffusion permet de conditionner la génération d'images via plusieurs modalités : texte, images ou semantic map. Un encodeur, noté  $\tau_\theta$  sur la figure 38, est utilisé pour projeter le conditionnement fourni en

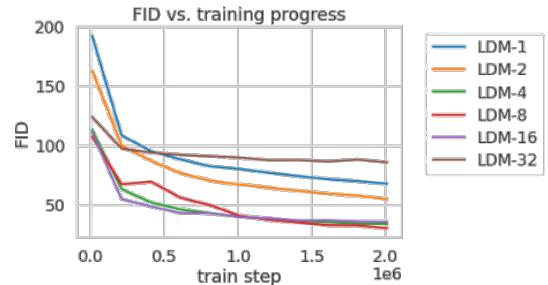


FIGURE 41 – Évolution de la FID pour différents facteurs de compression sur ImageNet

entrée  $y$  dans un espace d'embeddings. L'embedding obtenu est noté  $\tau_\theta(y)$ .

Pour des tâches de image-to-image translation, synthèse sémantique, super-résolution ou inpainting, l'autoencodeur est utilisé avec une régularisation VQ, et les images de conditionnement sont juste projetées dans l'espace latent et concaténées aux représentations des images d'entraînement. Pour de la génération text-to-image,  $\tau_\theta$  est le text encoder d'un modèle de type CLIP [29]. Le texte est convertit en tokens via l'algorithme WordPiece [30], aussi connu sous le nom de BERT-tokenizer, avant d'être converti en tenseur par  $\tau_\theta$ . Le mécanisme de cross-attention représenté sur la figure 43 est alors utilisé au niveau des couches d'attention du U-Net (cf. figure 28 ou blocs QKV sur la figure 38) pour guider la génération d'images.

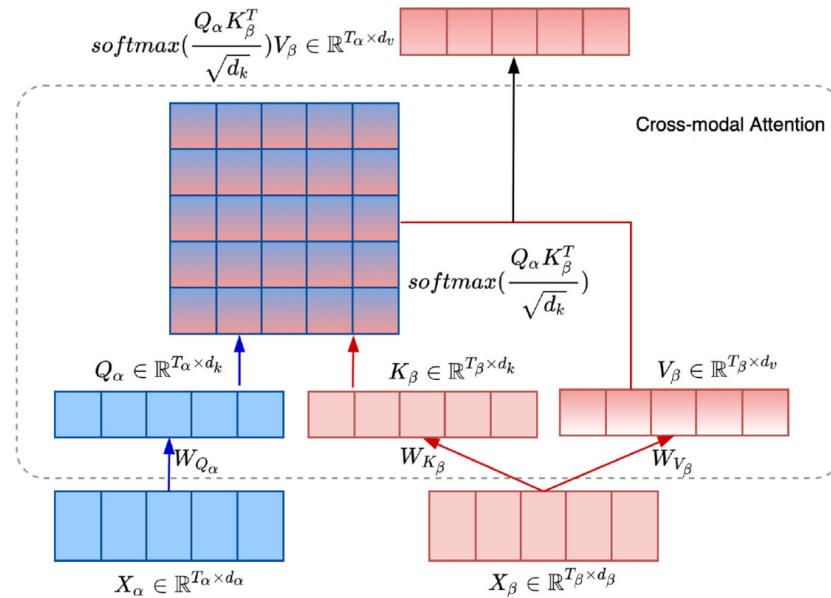


FIGURE 43 – Mécanisme de cross-Attention

Dans le cas de Stable Diffusion, les **représentations d'images** sont utilisées pour générer les vecteurs de **Query** (vecteurs bleus sur la figure), et sont donc associés à la matrice  $W_Q$ . Les **représentations textuelles** servent à générer les vecteurs de **Key** et **Value** (vecteurs rouges sur la figure) et sont donc associés aux matrices  $W_K$  et  $W_V$ . L'attention est calculée comme le produit scalaire normalisé de la Query avec la Key, et est utilisée en tant que coefficient de pondération de la Value pour générer le tenseur de sortie.

Le mécanisme de cross attention est également utilisé dans le cadre de la génération layout-to-image : le conditionnement est réalisé par une image contenant une bounding box et un label par objet à représenter.

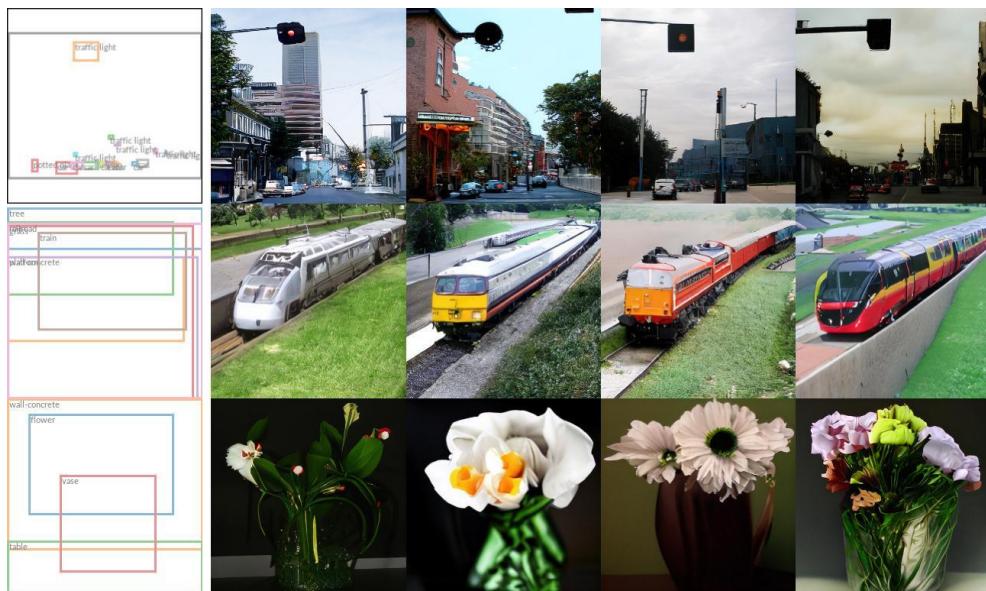


FIGURE 44 – Exemple de génération layout-to-image conditionnée via cross-attention

Pour la synthèse text-to-image, la principale différence entre Stable Diffusion v1.x et Stable Diffusion 2.x provient de l'encodeur utilisé :

- Stable Diffusion v1.x utilise un modèle CLIP ViT-L/14 [29] (ViT Large  $\sim 300M$  de paramètres pour le ViT [31], taille de patchs 14x14 pixels). Il s'agit du premier type de modèle basé sur des Transformers [23] à apprendre conjointement des embeddings d'images et de texte. Il a été développé et rendu public (code et poids d'entraînement) par OpenAI début 2021 et offre de très bonnes performances, mais il a été entraîné sur des données propriétaires et n'est pas disponible pour des tailles supérieures à Large.
- Stable Diffusion v2.x utilise OpenCLIP ViT-H/14 [32] (ViT Huge  $\sim 600M$  de paramètres pour le ViT, taille de patchs 14x14 pixels) à la place. Le modèle global est environ 5 fois plus gros que le modèle CLIP utilisé dans la v1.x. Il s'agit d'une réimplémentation open-source de CLIP entraînée sur le dataset LAION5B [33] en filtrant les images NSFW (Not Safe For Work) et en ne gardant que les images ayant un bon **score esthétique**.

#### 4.1.3 Diffusion dans l'espace latent

En ce qui concerne le modèle de diffusion en lui-même, son objectif est finalement similaire à celui d'un modèle DDPM classique, ce qui est facilement observable en comparant les fonctions de perte des 2 modèles :

$$\begin{aligned} L_{DPPM} &= \mathbb{E}_{x, \epsilon \sim \mathcal{N}(0,1), t} [\|\epsilon - \epsilon_\theta(x_t, t)\|_2^2] && \text{modèle DDPM} \\ L_{LDM} &= \mathbb{E}_{\mathcal{E}(x), y, \epsilon \sim \mathcal{N}(0,1), t} [\|\epsilon - \epsilon_\theta(z_t, t, \tau_\theta(y))\|_2^2] && \text{modèle LDM} \end{aligned}$$

Le conditionnement de l'espérance ne se fait plus sur  $x$ , l'image originale, mais sur  $\mathcal{E}(x)$ , sa représentation dans l'espace latent, et le bruit prédit dépend à présent aussi de l'embedding de conditionnement  $\tau_\theta(y)$ .

Pour l'entraînement de Stable Diffusion v2.x, certains modèles fournis ont été entraînés avec une fonction de perte alternative proposée dans la section 4 de cet article [34], nommée v-objective ou v-prediction. Cette fonction cherche à prédire la variable  $v \equiv \alpha_t \epsilon - \sigma_t x$ , et permet notamment de réaliser des distillations du modèle initial pour entraîner des modèles qui échantillonneront plus rapidement. La fonction de perte associée est :

$$L_\theta = \|v_t - \hat{v}_t\|_2^2$$

## 4.2 Entraînements de LDMs

Nous avons entraîné des modèles de diffusion latents conditionnés sur du texte en reprenant des modèles **Stable Diffusion préentraînés disponibles sur HuggingFace**, et les scripts d'entraînement fournis **sur le GitHub de la librairie diffusers**. Nous avons préféré utiliser des modèles déjà préentraînés plutôt que de réaliser un entraînement à partir de zéro. En effet, bien que les LDM soient des modèles de taille raisonnable comparés aux autres modèles de diffusion, un entraînement complet nécessite tout de même plusieurs jours de calculs sur un seul GPU (voire semaines si plusieurs versions d'autoencodeurs sont comparées), et créerait un modèle moins riche car l'alignement avec le text encoder serait fait sur un nombre limité de concepts (le dataset DTD ne contient que 5600 images, à comparer aux milliards de paires image-label utilisées pour entraîner Stable Diffusion). Nous pouvons ainsi tirer parti du fort *prior* sémantique appris lors du préentraînement et vérifier si la génération d'images est aussi efficace pour des concepts non inclus dans les images d'entraînement (via le dataset Safran notamment).

Nous avons testé différentes méthodes d'entraînement, en nous intéressant plus particulièrement à l'efficacité de l'entraînement (compromis entre qualité d'images générées et ressources de calcul utilisée pour l'entraînement). Nos résultats sont présentés dans les sous-sections ci-dessous.

### 4.2.1 Finetuning classique

Cette méthode d'entraînement est la plus simple à appréhender. Elle consiste à geler les paramètres de l'autoencodeur et du text encoder, et à n'entraîner que les paramètres du U-Net (comme un entraînement classique initialisé depuis des poids préentraînés).

Nous avons testé cette méthode sur le dataset DTD. L'entraînement a duré environ **16 heures pour 334 epochs d'entraînement sur les 5639 images du jeu de données sur 1 GPU A100 avec 40Go de RAM**, avec une taille de batch effective de 128 (34/40 Go de RAM occupés). Pour chaque image, le texte fourni était la concaténation de la chaîne de caractère "dtd\_" et de la classe de l'image (e.g. dtd\_braided, dtd\_cobwebbed, dtd\_spiralled, ...). L'ajout de "dtd\_" devant le nom de la classe permettait de créer un nouveau mot unique servant de prompt pour générer spécifiquement une image à partir du dataset DTD une

fois l'entraînement terminé. Sans cette "astuce", il aurait été impossible de savoir si l'image générée était le résultat de notre finetuning ou de l'entraînement initial (les noms de classe du dataset DTD sont tous des mots courants de la langue anglaise, pour lesquels le modèle dispose déjà d'une représentation suite à l'entraînement initial de Stable Diffusion).

Une autre astuce utilisée pendant l'entraînement est la **technique dite du "Offset Noise"**. Cette technique part du constat que les modèles de diffusion sont incapables de créer des images presque entièrement blanches ou presque entièrement noires : la moyenne de la valeur des pixels semble toujours être autour de 0.5 (pour des valeurs de pixels ramenées entre 0 et 1). La solution proposée dans l'article de blog [35] consiste à **ajouter un bruit constant pour tous les pixels d'une même feature map lors du forward process dans l'espace latent**. En code Python, la nouvelle forme du bruit peut s'exprimer ainsi :

```
# Initialement
noise = torch.randn_like(latents)
# Avec offset
offset_noise = torch.randn_like(latents) +
    0.1 * torch.randn(latents.shape[0], latents.shape[1], 1, 1)
```

Pour comprendre ce qui motive cette modification, il est intéressant de se pencher sur l'impact du bruit ajouté dans le processus forward sur les différentes fréquences présentes dans une image.

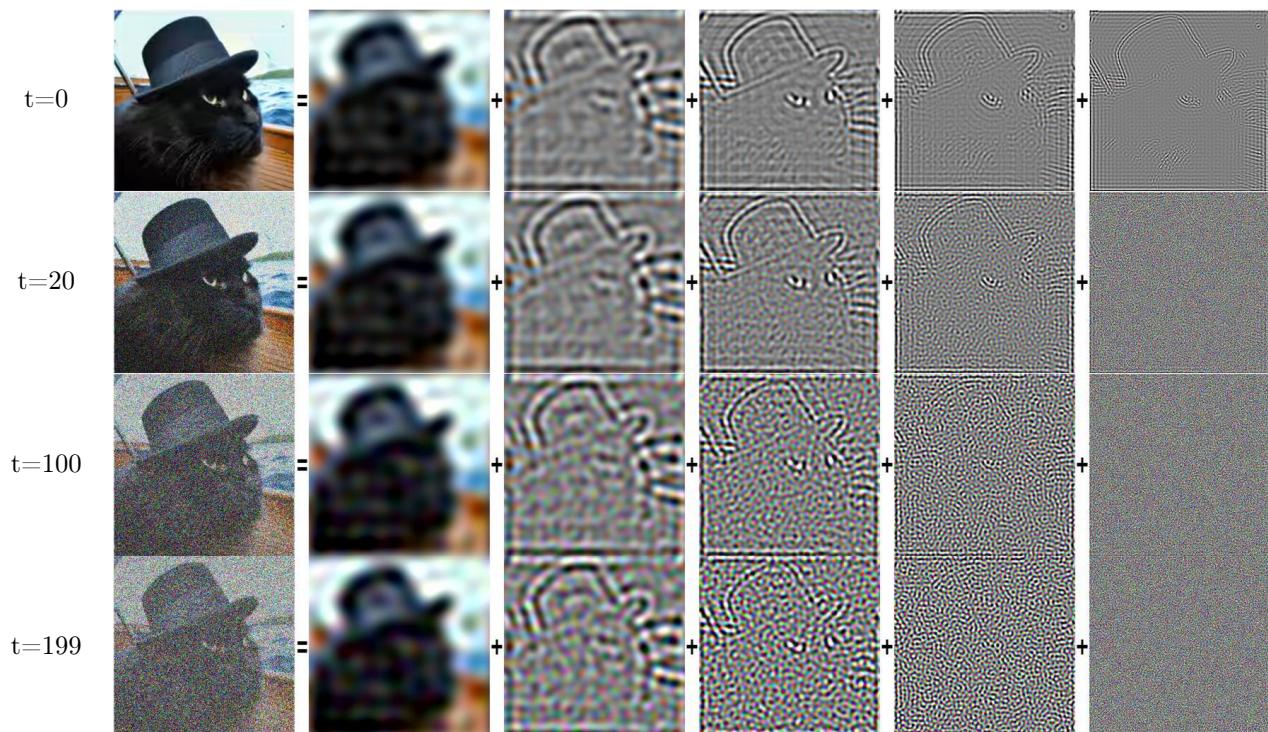


TABLE 5 – Impact de l'ajout progressif d'un bruit gaussien sur les différentes fréquences d'une même image. Dans la colonne de gauche est représentée l'image complète pour différents niveaux d'ajout de bruit. Les images de droite sont les reconstructions de certaines bandes de fréquence de cette image après calcul via Transformée de Fourier. Les images sont placées de gauche à droite par bandes de fréquence croissante (basses fréquences à gauche, hautes fréquences à droite)

Le tableau 5 illustre le fait que les hautes fréquences sont détruites beaucoup plus rapidement que les basses fréquences par ajout progressif de bruit sur une image : les plus hautes fréquences ne sont plus visibles au bout de 20 steps de bruit, alors que les plus basses fréquences semblent inchangées au bout de 200 steps. Au bout de 1000 steps de diffusion (valeur par défaut utilisée dans les modèles de diffusion), les basses fréquences de l'image n'ont donc pas entièrement été détruites. Par conséquent, le modèle n'apprend pas à reconstruire entièrement les basses fréquences, puisqu'une partie de l'information qu'elles contiennent est encore présente à la fin des étapes de bruitage. Le modèle ne peut donc pas générer des images contenant des basses fréquences avec des valeurs éloignées de la moyenne du bruit fourni en entrée du processus reverse (d'où l'observation initiale). Une solution serait de rajouter des étapes de bruitage pendant l'entraînement, mais cela forcerait un temps d'échantillonage plus long en inférence... L'ajout de bruit proposé force la destruction des basses fréquences lors du processus forward, et donne donc plus de liberté au modèle en phase d'inférence, sans impact sur la durée d'échantillonnage. Cette astuce semble produire des résultats convenables, mais n'est pas au goût de tous,

notamment des auteurs de [36], qui lui reprochent notamment de rompre la condition i.i.d. du bruit appliqué aux pixels, et proposent une autre solution pour adresser le même problème.

Des exemples d'images générées avec cette méthode d'entraînement sont affichées dans le tableau 6, les images sont de très bonne qualité. Pour un prompt donné, on remarque cependant que les images générées sont très similaires (pour dtd\_braided ou dtd\_wrinkled par exemple). Toutefois, la combinaison de nos prompts avec d'autres concepts déjà connus du modèle permet de vérifier que le modèle a bien appris de nouveaux concepts et ne se contente pas de régénérer la même image du dataset. Ainsi, le prompt "dtd\_braided brioche" génère bien des brioches tressées, "dtd\_spiralled galaxy" génère des galaxies en forme de spirale et "dtd\_wrinkled old person" génère bien des personnes âgées très ridées. Nous n'avons pas expérimenté avec l'ensemble des 47 classes du dataset, mais ces résultats semblent suggérer que l'entraînement a permis de faire apprendre correctement les nouveaux concepts de DTD sans catastrophic forgetting / overfit.

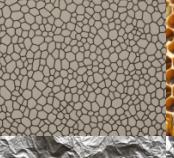
Prompt	Échantillon d'images générées					
dtd_braided						
dtd_braided brioche						
dtd_cobwebbed						
dtd_spiralled						
dtd_spiralled galaxy						
dtd_honeycombed						
dtd_wrinkled						
dtd_wrinkled old person						

TABLE 6 – Images générées à partir des prompts dans la colonne de gauche après finetuning de Stable Diffusion sur DTD

Une des limites de l'approche est visible à travers la tokenisation WordPiece [30] utilisée. Les prompts de type "dtd\_class" sont décomposés en tokens de la manière suivante :

- dtd\_honeycombed → ['d', 'td</w>', '\_</w>', 'honey', 'com', 'bed</w>']
- dtd\_braided → ['d', 'td</w>', '\_</w>', 'braided</w>']
- dtd\_spiralled → ['d', 'td</w>', '\_</w>', 'spir', 'alled</w>']
- ...
- dtd\_class → ['d', 'td</w>', '\_</w>', 'class</w>']

où les tokens générés à partir de la classe sont des concepts déjà connus du modèle. Il est donc difficile de savoir si les bons résultats obtenus sont entièrement attribuables à la méthode d'entraînement, ou s'ils sont en grande partie dûs à la connaissance antérieure du modèle d'une partie des tokens utilisés.

La méthode de finetuning classique produit donc de très bons résultats, mais nécessite tout de même l'accès à de bons GPUs pour produire des résultats en un temps raisonnable (ce qui n'a plus été notre cas sur la fin du projet à cause de problèmes de disques sur les serveurs de l'école). Nous avons donc testé d'autres approches en parallèle pour trouver des méthodes moins gourmandes en ressources de calcul.

#### 4.2.2 Textual Inversion

Une technique de finetuning populaire d'un LDM conditionné sur du texte est **Textual Inversion**, introduite dans l'article [37]. Cette méthode, originale pour entraîner un modèle générant des images, consiste à finetuner *uniquement une partie du text encoder* en gardant le reste du modèle gelé, comme illustré sur la figure 45.

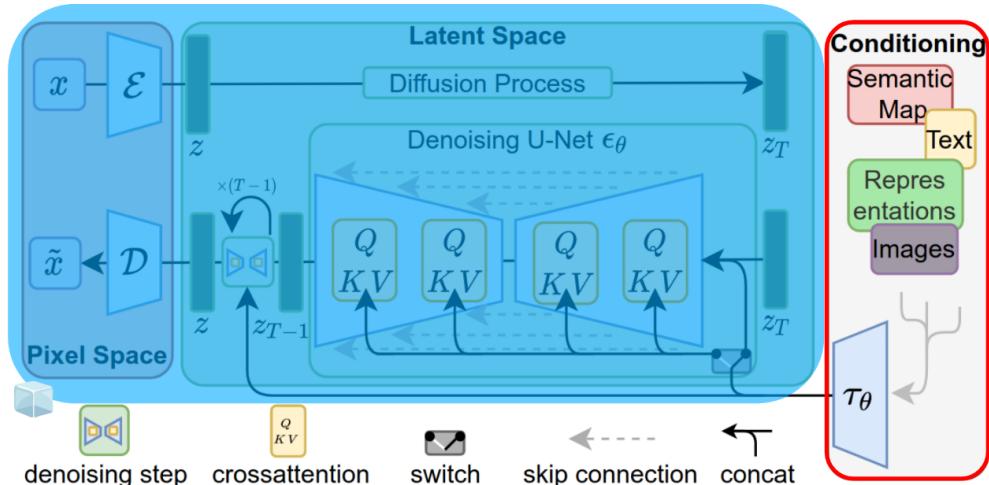


FIGURE 45 – Paramètres entraînés lors de l'utilisation de Textual Inversion

Le but de la méthode est de générer un ou plusieurs nouveaux embeddings textuels pour un concept particulier. Trois à cinq images d'un même concept suffisent pour entraîner le text encoder, qui permet ensuite la génération d'images incluant le nouveau concept dans le cas d'un objet, ou s'en inspirant en traitant le concept comme un style.



FIGURE 46 – Exemple d'images générées à partir d'une statuette d'oiseau, traitée à la fois comme un objet (dans la peinture) et comme un style (maison)

Plus en détails, pour réaliser l'entraînement, un caractère générique ( $S_*$  sur la figure 47) est utilisé pour décrire textuellement le nouveau concept. Une fois tokenisé, ce caractère est utilisé pour créer un vecteur, noté

$v$ , via la matrice d'embedding du text encoder. Lors de l'entraînement, les seuls paramètres non gelés (pour lesquels `require_grad = True`) sont les composantes du vecteur  $v$ . Seule la modification de ces valeurs permet l'alignement entre le texte d'un côté et les images de l'autre. L'objectif d'entraînement reste le même que pour la modèle Stable Diffusion, qui dans ce cas se réécrit ainsi :

$$v_* = \arg \min_v \mathbb{E}_{\mathcal{E}(x), y, \epsilon \sim \mathcal{N}(0,1), t} [\|\epsilon - \epsilon_\theta(z_t, t, c_\theta(y))\|_2^2]$$

Cette méthode, qui s'apparente finalement plus à du prompt-tuning qu'à du finetuning, produit donc uniquement un **embedding optimal du nouveau concept : les poids du modèle ne sont pas modifiés !**

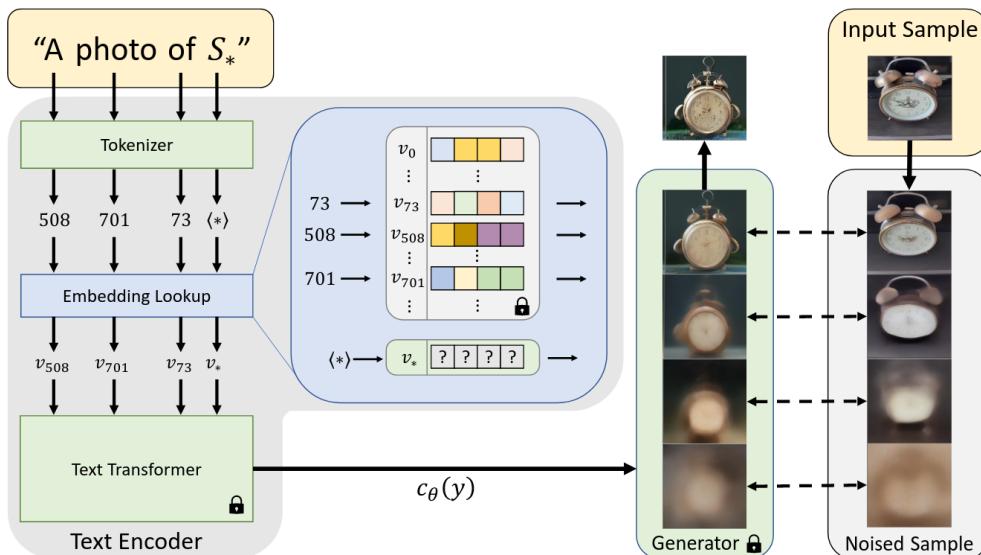


FIGURE 47 – Principe de Textual Inversion

Nous avons testé cette méthode d'entraînement sur le dataset Safran (table 1) pour apprendre 3 concepts : dry, thickness et warp. Pour chacun des concepts, nous avons utilisé 3 images de chaque type tirées au hasard, soit 9 images par concept (e.g. 3 "dry thickness", 3 "dry warp" et 3 "dry weft" pour le concept *dry*). Le [script fourni par HuggingFace](#) prend en arguments :

- le type de concept à apprendre : `object` ou `style`. Nous avons utilisé le type `object` pour les 3 concepts
- le token générique à utiliser pour représenter le nouveau concept, écrit entre chevrons. Nous avons utilisé le format `<concept-slice>` pour les 3 concepts
- le token d'initialisation : vecteur d'embedding avec lequel le nouveau vecteur va être initialisé. Nous avons utilisé le nom du concept comme token d'initialisation (i.e. "dry" pour `<dry-slice>`, "thickness" pour `<thickness-slice>`, etc.)

L'entraînement dure environ **1 heure 20 par concept pour 40 epochs sur 1 GPU A100 avec 40Go de RAM**, avec une taille de batch effective de 16 (23/40 Go de RAM occupés). Les résultats obtenus sont présentés dans la table 7.

On constate que les résultats sont très inégaux en fonction du concept d'entraînement. Pour **dry**, le modèle a déjà une connaissance bien ancrée du concept initial, comme le prouvent les images générées à la première epoch (sol desséché typique d'un environnement désertique). Après 40 epochs, les images générées ressemblent principalement à des images de type "thickness", mais la courbure des lignes horizontales rappelle plutôt les images de type "warp" ou "weft" : le modèle a semble-t-il fait une fusion des images thickness, warp et weft lors de l'apprentissage. Les images sont cependant de bonne qualité, relativement proches des images du dataset et éloignées du token d'initialisation (ce qui illustre bien l'impact de l'entraînement). Les résultats sont encore plus satisfaisants pour le concept **thickness**, pour lequel le modèle ne semble pas avoir de définition aussi précise initialement (cf images générées à l'epoch 1), mais qui aboutit à des images de très bonnes qualités après 40 epochs d'entraînement. En revanche, les résultats pour le concept **warp** sont beaucoup plus mitigés. Le modèle semble avoir une conception initiale assez forte du concept (bien qu'éloignée du concept recherché) : les 4 images générées après 1 epoch sont très semblables. Mais à partir de 20 epochs, l'entraînement semble avoir convergé vers un concept assez éloigné du dataset.

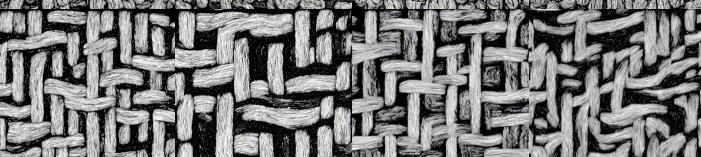
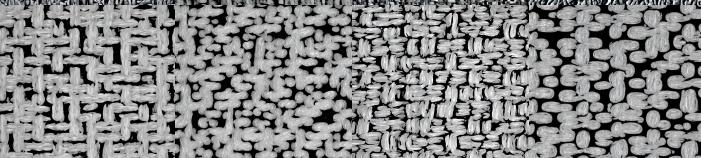
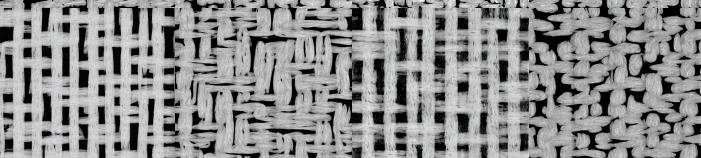
Prompt	Epoch	Échantillon d'images générées
A <dry-slice> of a carbon composite material	1 20 40	  
A <thickness-slice> of a carbon composite material	1 20 40	  
A <warp-slice> of a carbon composite material	1 20 40	  

TABLE 7 – Images générées à partir des prompts dans la colonne de gauche après finetuning via Textual Inversion sur DTD pour différentes epoch d'entraînement

Cette méthode est donc intéressante car elle permet de faire générer un nouveau concept à un modèle Stable Diffusion avec un entraînement relativement rapide / peu coûteux en ressources. Les embeddings générés sont facilement partageables et combinables, générer une images en intégrant plusieurs nouveaux embeddings est très facile, ce qui en fait une méthode populaire dans les communautés utilisant Stable Diffusion comme [civitai](#). Cependant, la qualité des images générées est hétérogène, et pas toujours optimale. S'agissant d'une méthode

qui s'appuie uniquement sur le text encoder, elle est particulièrement sensible aux prompts utilisés / tokens d'initialisation, et peut avoir du mal à apprendre des concepts éloignés de son contexte d'entraînement (comme c'est le cas avec warp). Le script de HuggingFace permet l'apprentissage de plusieurs embeddings pour un seul concept, ce qui peut potentiellement améliorer la qualité des images générées. Trouver et optimiser des prompts efficaces étant notoirement complexe, nous avons préféré tester d'autres approches plutôt que de chercher à optimiser les résultats avec cette méthode.

#### 4.2.3 DreamBooth

Une approche aux objectifs similaires à Textual Inversion est *Dreambooth*, décrite dans un article de Google [38] également paru en 2022. Le but de la méthode est d'intégrer un nouveau concept dans le modèle Stable Diffusion de manière efficace (entraînement rapide et peu coûteux, besoin de quelques images uniquement). Contrairement à Textual Inversion, les poids du modèle sont ici bien modifiés : par défaut le script d'HuggingFace permet de finetuner le U-Net pour se conformer à l'article, mais le finetuning du text encoder peut également être activé via une option, ce qui produit en général de meilleurs résultats. Les paramètres de l'autoencodeur sont dans tous les cas gelés, une option permet même d'en désactiver l'utilisation (la diffusion a alors lieu dans le pixel space).

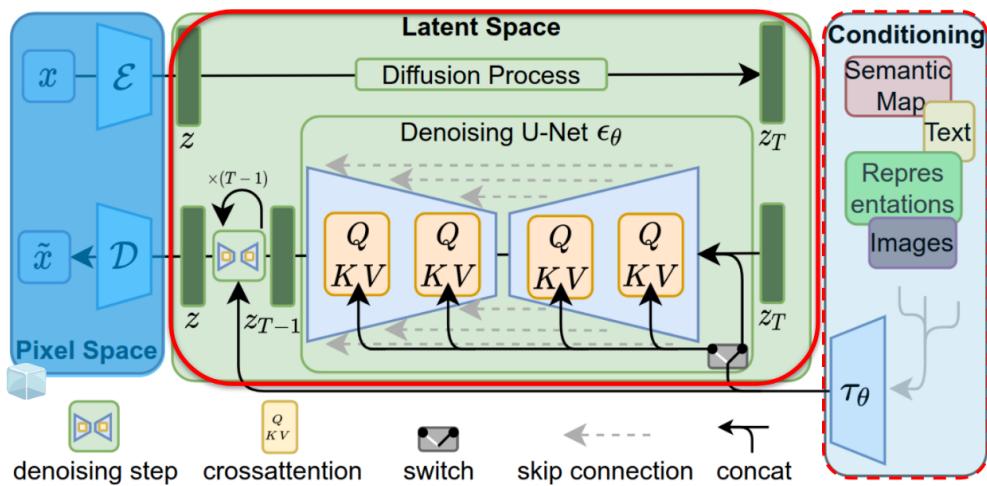


FIGURE 48 – Paramètres entraînés lors de l'utilisation de DreamBooth

La méthode proposée s'apparente à du finetuning classique, à l'exception près que seules 3 à 5 images sont suffisantes pour faire apprendre un nouveau concept au modèle. Contrairement aux difficultés rencontrées pour faire apprendre de nouveaux concepts à des GANs en few-shot learning, les latent diffusion models semblent capables d'intégrer de nouvelles informations sans overfitter ni oublier le *prior* sémantique appris par le modèle, **à condition de bien choisir le prompt utilisé pendant l'entraînement**. Les auteurs de l'article ont observé de bons résultats en utilisant des prompts (en anglais) de la forme "a [identifier] [class noun]" où

- **identifier** est un identifiant unique qui va servir à caractériser le nouveau concept. Contrairement à Textual inversion, l'entraînement n'insère pas de nouveau concept dans le dictionnaire d'embeddings du text encoder. Ainsi, l'identifiant utilisé doit perdre le sens qui y était attaché pour se lier au nouveau concept présent dans les images d'entraînement. Les auteurs mettent donc en avant l'importance de choisir un identifiant qui ait un faible *prior* à la fois dans le modèle de langue et dans le modèle de diffusion utilisé. Ils déconseillent de générer un identifiant à partir de caractères aléatoires (eg "xxy5syt00"), car la tokenisation WordPiece [30] risque de séparer les lettres, et de générer des embeddings pour lesquels le modèle a déjà un fort *prior*. A la place, ils proposent de chercher les tokens rares dans un grand corpus de texte et d'utiliser les mots générant ces tokens comme identifiants. Le script d'HuggingFace propose d'utiliser *sks* comme mot pour le nouveau concept car il s'agit d'un mot rare. Cela semble cependant poser problème dans certains cas, [comme décrit dans cette issue GitHub](#), car le SKS est carabine créée par un concepteur d'arme soviétique en 1945. Transformer le mot initial en y insérant du leet semble être une alternative efficace.
- **class noun** est une description approximative de la classe à laquelle appartient le nouveau concept (e.g. chien, chat, horloge, etc.). Elle peut être choisie par l'utilisateur ou proposée par un classifieur. Les auteurs de l'article constatent que donner une mauvaise classe, ou ne pas en préciser du tout dans le prompt, augmente le temps d'entraînement nécessaire pour la convergence, et le risque de dérive linguistique, tout en faisant diminuer les performances. Choisir une classe appropriée est donc nécessaire pour obtenir de

bons résultats, le but étant de tirer profit du *prior* sémantique déjà inclus dans le modèle pour guider l'apprentissage.

Malgré ces précautions, il arrive tout de même que l'entraînement conduise à de la dérive linguistique (en particulier pour du finetuning sur des datasets de visages si l'on en croit [ce blogpost d'HuggingFace](#)). La dérive linguistique est un phénomène à l'origine observé dans les modèles de langage, où un modèle pré-entraîné sur un grand corpus de textes et affiné par la suite pour une tâche spécifique perd progressivement la connaissance syntaxique et sémantique de la langue. Dans le contexte des modèles de diffusion, le finetuning sur quelques images d'un nouveau concept fait progressivement oublier au modèle comment générer des images de la même classe que le nouvel objet d'entraînement. Les auteurs ajoutent donc un nouveau terme de régularisation à la fonction de perte, qu'ils nomment **Prior Preservation Loss**. L'idée est de générer des images de la même classe que le nouveau concept en amont du finetuning, et de forcer les nouvelles images générées par le modèle au cours de l'entraînement à rester proche de ces images (donc à ne pas dégénérer en ne créant que des images du nouveau concept lorsque la classe plus générale est requise). La figure 49 illustre le calcul de ces 2 termes de la fonction de perte :

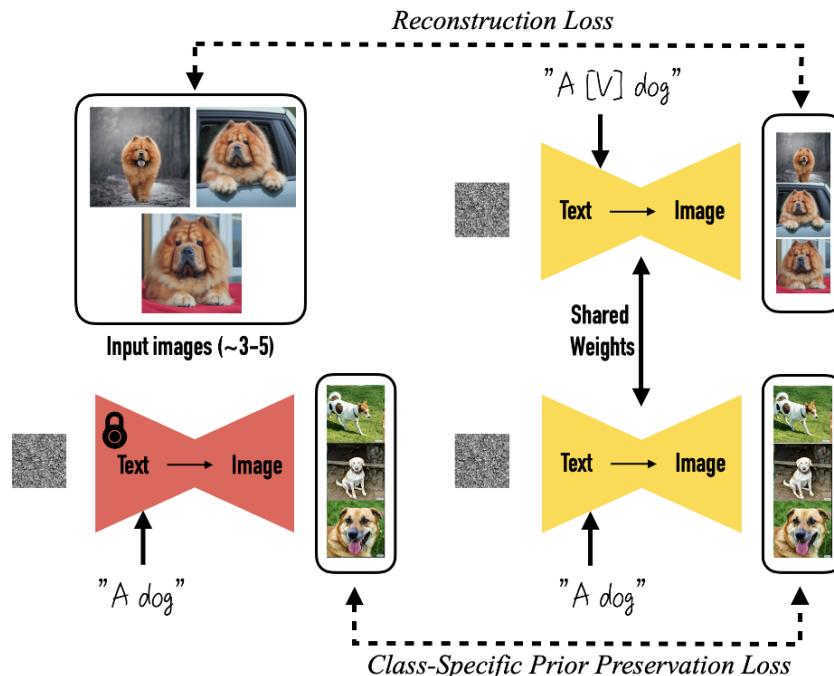


FIGURE 49 – Illustration des 2 termes de la fonction de perte utilisée dans Dreambooth

La fonction de perte utilisée prend finalement cette forme :

$$L = \underbrace{\mathbb{E}_{\mathbf{x}, \mathbf{c}, \boldsymbol{\epsilon}, \boldsymbol{\epsilon}', t} [w_t \|\hat{\mathbf{x}}_\theta(\alpha_t \mathbf{x} + \sigma_t \boldsymbol{\epsilon}, \mathbf{c}) - \mathbf{x}\|_2^2]}_{\text{Reconstruction Loss}} + \underbrace{\lambda w_{t'} \|\hat{\mathbf{x}}_\theta(\alpha_{t'} \mathbf{x}_{\text{pr}} + \sigma_{t'} \boldsymbol{\epsilon}', \mathbf{c}_{\text{pr}}) - \mathbf{x}_{\text{pr}}\|_2^2}_{\text{Prior Preservation Loss}}$$

où  $\mathbf{x}_{\text{pr}}$  représente les images de la classe à préserver (dog sur la figure 49),  $\mathbf{c}_{\text{pr}}$  correspond au prompt de la classe à préserver ("dog"),  $\mathbf{x}$  correspond aux images du nouveau concept et  $\mathbf{c}$  correspond à l'identifiant du nouveau concept ("[V]" sur la figure 49). En pratique, un  $\lambda$  de 1 semble être efficace.

Nous avons testé cette méthode d'entraînement sur la classe *zigzagged* du dataset DTD, qui contient 120 images. Nous avons comparé plusieurs paramètres d'entraînement :

1. Entraînement du U-Net et du text encoder sur les 120 images de la classe sans prior preservation loss. L'entraînement dure **58 minutes pour 800 epochs sur 1 GPU A100**
2. Entraînement du U-Net et du text encoder sur les 120 images de la classe avec prior preservation loss. L'entraînement dure environ **1 heure 30 pour 800 epochs sur 1 GPU A100 et 10 minutes pour générer 200 images de la classe texture à préserver**
3. Entraînement du U-Net et du text encoder sur 10 images sélectionnées aléatoirement sans prior preservation loss. L'entraînement dure **12 minutes pour 800 epochs sur 1 GPU A100**

Dans tous les cas, le prompt d'entraînement est "a dtd\_zigzagged texture" (dtd\_zigzagged est l'**identifier**, texture le **class noun**), le batch size est de 128 et le learning rate de  $2 \times 10^{-6}$ .

Avec la 1<sup>ère</sup> méthode, les résultats obtenus sont déjà de très bonne qualité au bout de 99 epochs ( 8 minutes d'entraînement)

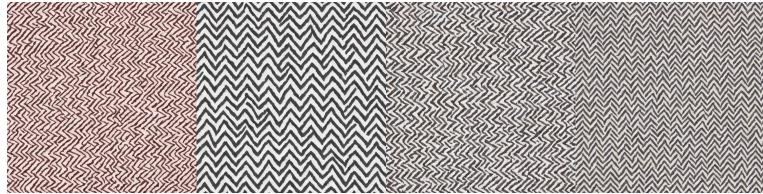


FIGURE 50 – Images générées avec le prompt "a dtd\_zigzagged texture" au bout de 99 epochs d'entraînement

Cependant, le modèle semble avoir subi une dérive linguistique. Les concepts ont été mélangés suite à l'apprentissage, comme l'illustrent les images suivantes : lorsque le prompt demande des zèbres avec des rayures en zigzag, des zigzags noirs et blancs sont générés mais pas de zèbres ; lorsque le prompt demande 2 zèbres (sans zigzag), des hybrides de zèbre et de zigzags sont générés :

Prompt	Échantillon d'images générées			
Zebras with dtd_zigzagged skin				
Two zebras				

TABLE 8 – Images générées à partir des prompts dans la colonne de gauche via Dreambooth sur les 120 images de la classe zigzagged du dataset DTD **sans prior preservation loss**

Pour tenter de résoudre ce problème, le même entraînement a été réalisé en rajoutant la *prior preservation loss* de l'article. La classe préservée est "texture". Le script d'HuggingFace demande un dossier contenant un certain nombre  $n$  d'images de la classe à préserver. Si le nombre d'images dans ce dossier est inférieur à  $n$ , des images sont générées via le modèle pré-entraîné jusqu'à arriver au nombre  $n$  exigé. Dans notre cas, nous avons laissé le modèle Stable Diffusion 2.1 générer 200 images de texture, dont un échantillon est présenté dans la table 9 (certaines sont plutôt artistiques, d'autres semblent incongrues voire cauchemardesques)

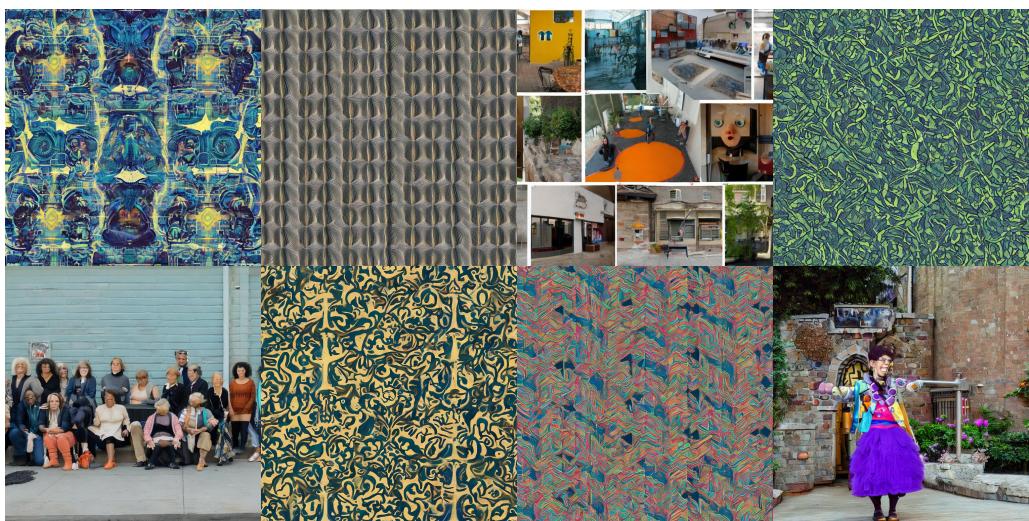


TABLE 9 – Echantillon d'images générées par le modèle pré-entraîné pour préserver la classe texture

La qualité des images de zigzag générées après entraînement est similaire à celle obtenue sans prior preservation :



FIGURE 51 – Images générées avec le prompt "a dtd\_zigzagged texture" au bout de 399 epochs d'entraînement

En revanche, les images de texture générées sont bien plus diversifiées ! L'ajout de la nouvelle loss a bien un effet notable.



TABLE 10 – Images générées pour le prompt "a texture" (à gauche : sans preservation loss, à droite : avec preservation loss)

Cependant, le problème pour des images en dehors de la classe préservée n'est pas réglé par l'ajout de ce terme dans la fonction de perte.

Prompt	Échantillon d'images générées			
Zebras with dtd_zigzagged skin	A generated image showing a zebra's body covered in a uniform zigzag pattern.	A generated image showing a zebra's head and neck with a zigzag pattern on the neck.	A generated image showing a zebra's body covered in a uniform zigzag pattern.	A generated image showing a zebra's body with a zigzag pattern on the lower half.
Two zebras	A generated image showing two zebras with a zigzag pattern on their bodies.	A generated image showing two zebras with a zigzag pattern on their heads.	A generated image showing two zebras with a zigzag pattern on their bodies.	A generated image showing two zebras with a zigzag pattern on their heads.

TABLE 11 – Images générées à partir des prompts dans la colonne de gauche via Dreambooth sur les 120 images de la classe zigzagged du dataset DTD **avec prior preservation loss**

Nous avons également regardé l'évolution des images générées pendant l'entraînement pour le prompt "two zebras" sur 800 epochs, en utilisant seulement 10 images sélectionnées aléatoirement sans prior preservation, pour mesurer l'impact du nombre d'images fourni en entrée (d'après l'article, 3 à 5 images sont suffisantes pour faire apprendre un nouveau concept au modèle). Les résultats sont présentés dans le tableau 12. Le modèle semble réussir à générer des images de zèbre non influencées par l'entraînement jusqu'à l'epoch 300, à partir de laquelle on commence à observer plus de zigzags sur la peau des zèbres. A partir de l'epoch 400, les images de zèbres sont progressivement remplacées par des images de textures de zigzags. Ce comportement est observé vers l'epoch 750 lorsque les 120 images de la classe sont utilisées. Le meilleur moyen de réduire le catastrophic forgetting observé semble donc de limiter le nombre d'epochs d'entraînement.

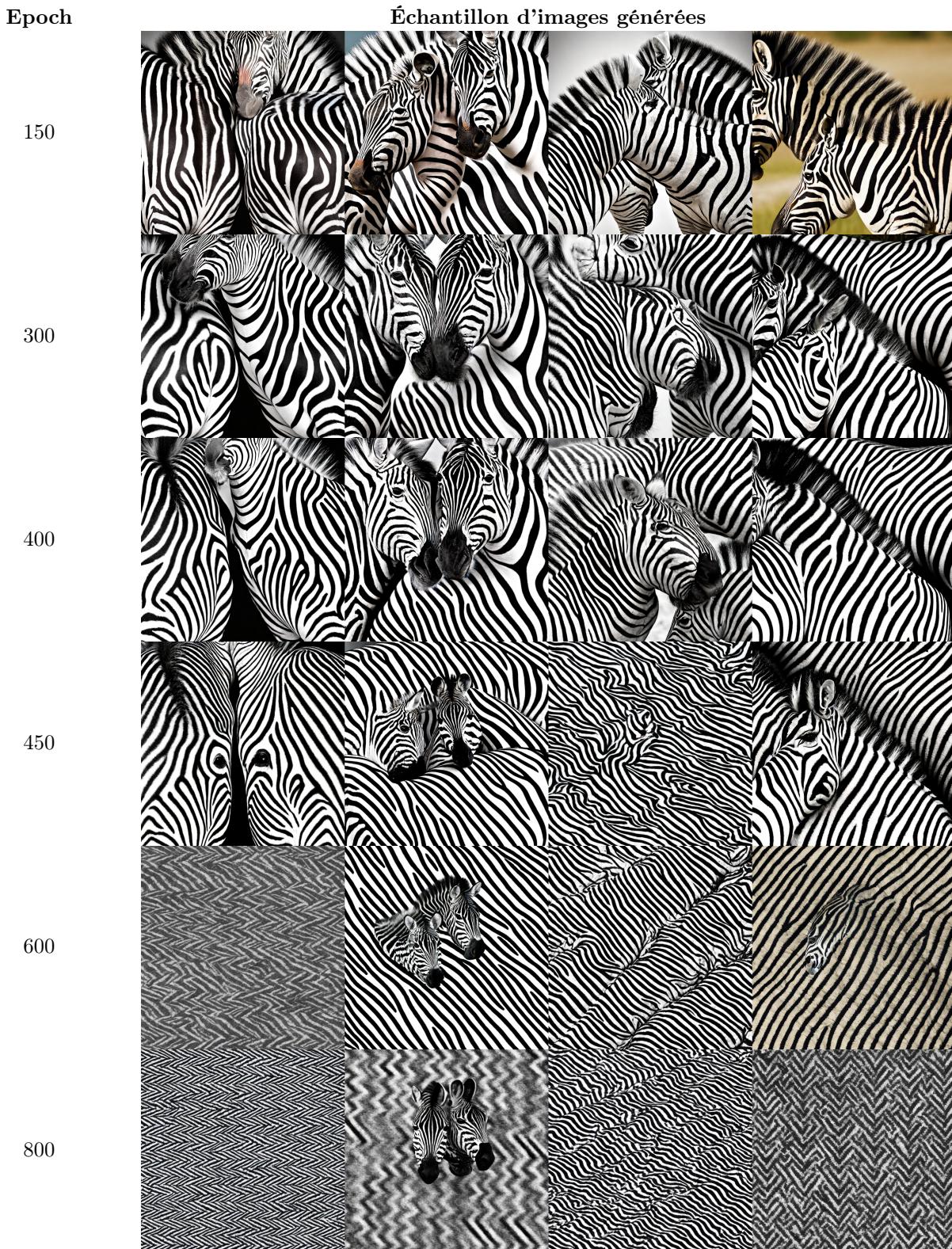


TABLE 12 – Images générées à partir du prompt "two zebras" pour différentes epochs d'entraînement sur 10 images de la classe zigzagged du dataset DTD

Dreambooth est donc une méthode très puissante et très rapide pour faire apprendre UN nouveau concept à un LDM à partir de très peu d'images. Le seul réel inconvénient de cette méthode est la facilité avec laquelle l'apprentissage du nouveau concept efface les connaissances du pre-training, ce qui pose problème si on veut se servir de cette méthode pour finetuner sur plusieurs classes (seule la dernière classe entraînée risque d'être retenue par le modèle), ou si l'on souhaite pouvoir réutiliser les connaissances du pré-entraînement conjointement au nouveau concept. Dans notre cas, un entraînement sur les 120 images d'une classe pour 100 epochs semble être suffisant pour incorporer un nouveau concept. Un tel entraînement prendrait environ 8 minutes sur un GPU A100. Nous n'avons pas essayé de faire apprendre un second concept à un modèle entraîné sur

lequel un premier concept aurait déjà été rajouté via Dreambooth. Utiliser moins d'images pour l'entraînement permet un apprentissage plus rapide, mais réduit logiquement la diversité des images générées et augmente le risque de catastrophic forgetting / language drift observé (il faut faire plus attention au nombre d'epochs pour l'entraînement).

Un [script expérimental](#) est proposé dans la librairie diffusers pour permettre l'apprentissage de plusieurs concepts en parallèle via Dreambooth, mais les résultats ne sont pas très bons d'après la communauté : <https://github.com/huggingface/diffusers/issues/2599>

#### 4.2.4 LoRA

La méthode **Low-Rank Adaptation** (LoRA) a à l'origine été proposée pour finetuner de manière efficace les grands modèles de langage (LLM) comportant plusieurs dizaines de milliards de paramètres, tels que GPT-3 (175B paramètres). Les auteurs de l'article introduisant cette méthode [39] s'inspirent des résultats d'autres équipes [40] [41] ayant montré que les LLMs sont souvent sur-paramétrés : la dimension du paramètre  $\theta$  optimisé par un réseau de neurones pour résoudre une tâche donnée est généralement plus grande que la dimension minimale permettant d'obtenir une solution (définie comme *dimension intrinsèque* par les auteurs de [40]). LoRA repose sur l'hypothèse que le finetuning d'un modèle pré-entraîné peut également être réalisé sur un espace de faible dimension (i.e. avec des matrices de poids de rang faible) avec des performances équivalentes à un entraînement réalisé sur l'ensemble des paramètres. L'entraînement via LoRA s'appuie sur 3 idées principales, illustrées sur la figure 52 :

##### 1. Injection de poids d'entraînement

Plutôt que de finetuner tous les poids du modèle pré-entraîné initial, les auteurs décomposent les matrices  $W$  du modèle finetuné ainsi :  $W = W_0 + \Delta W$ , avec

- $W_0$  poids gelés du modèle pré-entraîné (non modifiés pendant le finetuning)
- $\Delta W$  matrice de poids injectée, de même dimension que  $W_0$

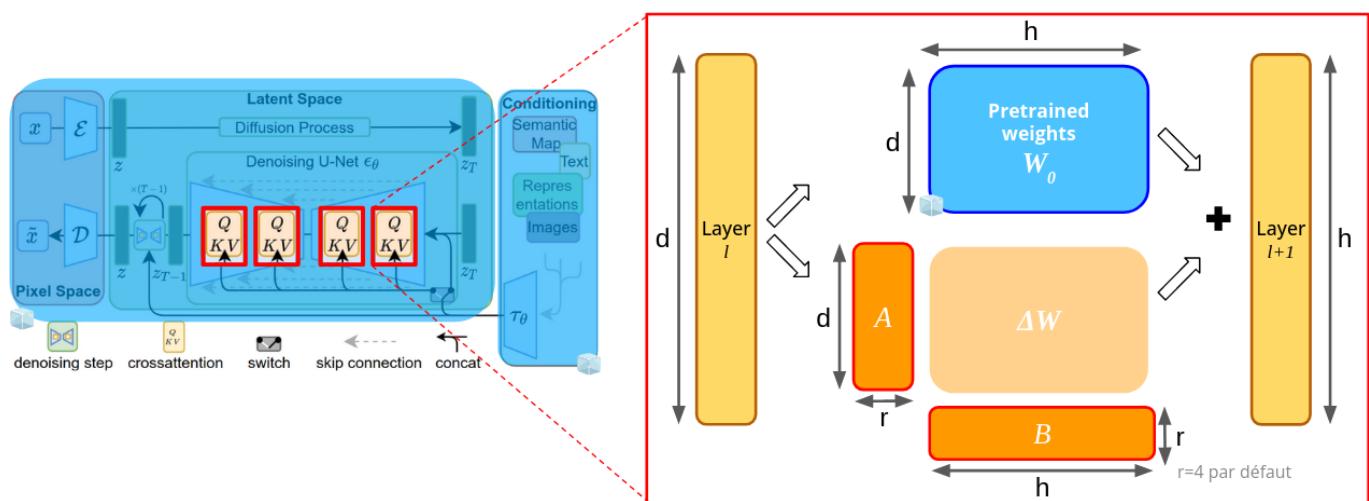
Un même modèle pré-entraîné peut donc être utilisé pour différentes sous-tâches simplement en remplaçant les matrices  $\Delta W$  finetunées, ce qui évite d'avoir à générer un nouveau modèle complet à chaque finetuning

##### 2. Décomposition en matrices de rang faible

L'hypothèse de rang faible se traduit par la décomposition de la matrice  $\Delta W \in \mathbb{R}^{d \times h}$  comme le produit de 2 matrices  $A \in \mathbb{R}^{d \times r}$  et  $B \in \mathbb{R}^{r \times h}$  :  $\Delta W = A \cdot B$  avec  $r \ll \min(d, h)$ . Ainsi, pour chaque matrice de poids à finetuner, seules les matrices  $A$  et  $B$  sont entraînées, ce qui revient à entraîner uniquement  $r \times (d + h)$  paramètres au lieu de  $d \times h$ . Typiquement, pour GPT-3, un rang  $r$  de 2 ou 4 suffit à donner de bons résultats, alors que le rang des matrices  $W_0$  atteint 12 288 lors du pré-entraînement. Cette amélioration permet dans ce cas de réduire le nombre de paramètres entraînables par 10 000 et la RAM occupée par le modèle par 3. Un rang de 4 est utilisé par défaut dans [le script fourni dans diffusers](#) pour Stable Diffusion. La forme des matrices de poids du modèle finetuné final est donc  $W = W_0 + A \cdot B$

##### 3. Entraînement des couches d'attention uniquement

Les auteurs n'ont testé leur méthode que sur les couches d'attention (ce qui semble logique pour une première approche sur des architectures de type transformer) et obtenaient déjà de bons résultats, mais n'excluent pas de généraliser la méthode à tout type de couche d'un réseau de neurones.



Nous avons testé cette méthode sur le dataset Safran avec Stable Diffusion 1.5, puis avec Stable Diffusion 2.1 en réduisant la résolution des images à 512 px (les GPUs A100 n'étant pas disponibles à cette période). L'entraînement a duré environ **22 heures pour 88 epochs d'entraînement sur les 5485 images du jeu de données sur 1 GPU V100 avec 32Go de RAM**, avec une taille de batch effective de 64. Les résultats sont disponibles dans le tableau 13. Avec Stable Diffusion 1.5, les résultats semblent encourageants vers 25 epochs, mais finissent par dégénérer. En créant une association de tokens uniques pour notre dataset (ajout du préfixe "sf\_" comme dans la section [4.2.1. Finetuning classique](#)), les résultats obtenus sont nettement meilleurs et convergent à partir de l'epoch 20.

Prompt	Epoch	Échantillon d'images générées
dry compacted thickness avec Stable Diffusion 1.5	1	
	10	
	25	
	42	
	1	
	3	
	10	
	20	

TABLE 13 – Images générées à partir des prompts dans la colonne de gauche après finetuning avec LoRA sur DTD pour différentes epoch d'entraînement

L'utilisation d'un prompt spécifique est donc très importante tant pour la qualité des images obtenues que pour la rapidité de convergence. Des exemples d'images générées à partir du modèle finetuné sur Stable Diffusion

2.1 pour les autres classes sont présentées dans le tableau 14.

Prompt	Échantillon d'images générées			Prompt	Échantillon d'images générées		
sf_dry				sf_dry			
sf_thickness				sf_warp			
sf_dry				sf_dry			
sf_compacted				sf_weft			
sf_thickness				sf_dry			
sf_injected				sf_compacted			
sf_thickness				sf_weft			
sf_injected				horse			
sf_weft				on a beach			
sf_compacted				sf_compacted			

TABLE 14 – Images générées à partir des prompts dans les colonnes extérieures suite au finetuning via LoRA avec Stable Diffusion 2.1

Les images générées sont très fidèles aux images d'entraînement, leur qualité est très bonne. LoRA semble donc tenir ses promesses : rendre l'entraînement plus efficace sans perte de qualité pour les images générées. L'apprehension générée par cette méthode parmi les équipes de Google semble donc justifiée.

Le prompt "horse on a beach" a été utilisé pour évaluer l'overfitting / le risque de catastrophic forgetting lié à cette méthode d'entraînement. Il est intéressant de noter que, malgré un entraînement sur des images uniquement en noir et blanc, les images générées sont toutes en couleur ; le cheval reste cependant toujours noir / très sombre, avec peu de détails sur l'animal en lui-même. Si la seconde image est rassurante quant aux capacités de rétention du *prior* du modèle, la première photo générée révèle tout de même un début de dérive linguistique : l'arrière-plan est constitué de motifs ressemblant à une coupe "dry compacted thickness" jaunâtre. Il y a donc bien un risque d'overfitting avec LoRA, mais il semble plus facile à maîtriser que pour Dreambooth. Le modèle utilisé a été entraîné sur 88 epochs alors que images de bonnes qualité étaient déjà générées à partir de l'epoch 18 : l'early stopping semble donc une solution valable pour limiter ce problème.

La dernière ligne du tableau montre des images générées pour le prompt "sf\_compacted", qui correspond à une classe qui n'existe pas seule dans le jeu de données (elle est toujours liée au concept sf\_dry). Les images générées ressemblent à une interpolation entre les classes dry compacted et injected, tout en restant plus proches de la classe dry compacted : les images de thickness ont l'air plus aplatis / moins fibreuses que des dry compacted classiques, les images warp et weft sont moins zoomées et se rapprochent plus des images injected. Le modèle a donc bien réussi à intégrer les différents concepts sémantiques du jeu de données.

Pour rester fidèle à l'article initial, l'implémentation utilisée n'appliquait LoRA qu'aux couches d'attention

du U-Net. Cependant, cloneofsimo (développeur dont l'implémentation de LoRA a été reprise dans la librairie diffusers) propose également sur [son repository GitHub](#) une extension de LoRA aux paramètres des couches ResNet du U-Net.

Un des autres avantages de LoRA est que cette méthode de finetuning est orthogonale aux autres méthodes présentées auparavant. Elle peut notamment être combinée à Dreambooth pour rendre l'apprentissage d'un concept encore plus efficace, à condition d'ajuster les hyperparamètres d'entraînement. Le tableau ci-dessous présente les résultats obtenus lors du finetuning de Stable Diffusion 2.1 via LoRA + Dreambooth sur les 4 concepts suivants : dry (associé à l'identifiant *w17h3r*), thickness (associé à l'identifiant *th1ck*), warp (associé à l'identifiant *w@rp*) et injected (associé à l'identifiant *1nj3C7*). De même que pour Textual Inversion, pour chacun des concepts, nous avons utilisé 3 images de chaque type tirées au hasard, soit 9 images par concept (e.g. 3 "dry thickness", 3 "dry warp" et 3 "dry weft" pour le concept *dry*). L'entraînement dure environ **12 minutes pour 500 epochs d'entraînement sur 9 images sur 1 GPU V100 avec 32Go de RAM**, avec une taille de batch effective de 64.

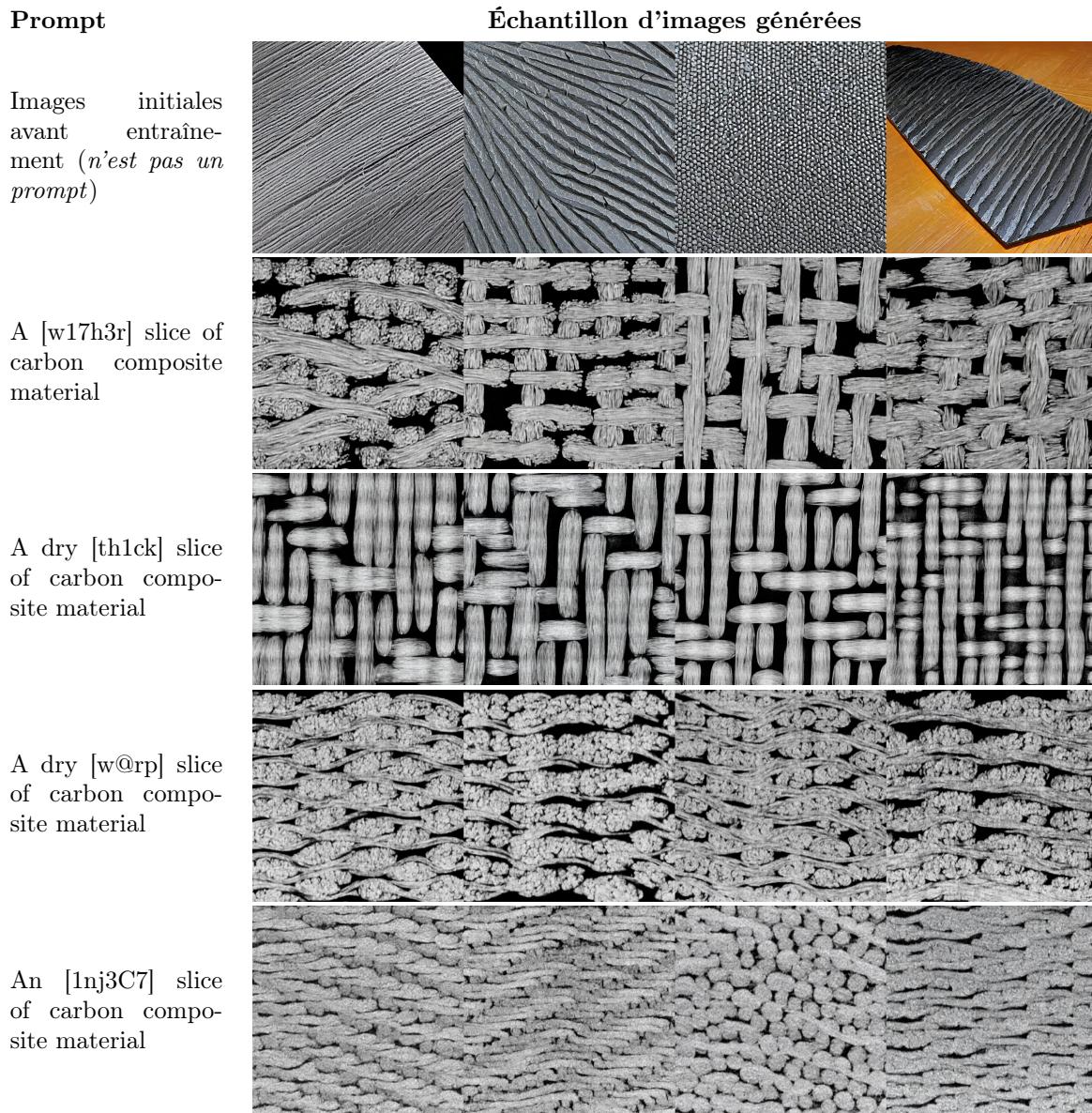


TABLE 15 – Images générées à partir des prompts dans la colonne de gauche via LoRA + Dreambooth sur 9 images sans prior preservation loss. Les identifiants utilisés pour l'apprentissage sont placés entre crochets dans le prompt par soucis de visibilité (crochets non inclus dans le prompt de génération)

Les images obtenues sont de très bonne qualité et fidèles aux images du dataset d'entraînement, seules une dizaine de minutes sont nécessaires pour l'entraînement. Les prompts *1nj3C7* et *w17h3r* permettent bien de générer des images de thickness et de warp, mais le modèle ne connaît qu'un seul concept à la fois : avec la méthode utilisée on ne contrôle pas si une image injected générée sera de thickness, warp ou weft. Ce problème peut facilement être résolu en finetunant sur chacune des 9 sous-catégories du dataset Safran (cf. 1). Le fait

que LoRA injecte de nouvelles matrices de poids, et conserve le modèle original gelé, aide même à réduire les problèmes d'overfitting observés précédemment, comme l'illustrent les images suivantes, générées à partir du prompt "horse on a beach" avec le modèle finetuné sur les images injectées. L'unique inconvénient de cette méthode est donc qu'elle est difficilement généralisable à plusieurs concepts.



TABLE 16 – Images générées à partir du prompt "horse on a beach" avec le modèle finetuné via LoRA + Dreambooth sur 9 images injectées du dataset Safran

LoRA peut également être utilisée en combinaison avec Textual Inversion et Dreambooth, dans une méthode que cloneofsimo appelle **Pivotal Tuning** en s'inspirant d'un travail réalisé sur des GANs [42]. Le principe est de commencer par apprendre un nouvel embedding textuel via Textual Inversion, puis d'utiliser cet embedding conjointement avec la *prior preservation loss* de Dreambooth pour obtenir des résultats de meilleure qualité. Le script implémentant cette méthode, que nous n'avons pas testé, est disponible à cette URL : [https://github.com/cloneofsimo/lora/blob/master/training\\_scripts/run\\_lorpt.sh](https://github.com/cloneofsimo/lora/blob/master/training_scripts/run_lorpt.sh).

#### 4.2.5 Récapitulatif

Le tableau ci-dessous résume les avantages et inconvénients des différentes méthodes de finetuning testées au cours de notre projet.

Méthode	Forces	Faiblesses	Commentaires
Finetuning classique	<ul style="list-style-type: none"> <li>- Entraînement multi-concepts / multi-classes</li> <li>- Résultats de très bonne qualité</li> <li>- Méthode classique : sélection standard d'hyperparamètres, de régularisation, etc.</li> </ul>	Entraînement long et peu efficace	Méthode peu intéressante, LoRA propose des résultats de qualité équivalente en moins de temps et en utilisant moins de ressources
Textual Inversion	<ul style="list-style-type: none"> <li>- Entraînement relativement efficace</li> <li>- Conservation des poids du modèle initial : pas de risque d'oubli de concepts ou d'overfitting</li> <li>- Embeddings générés de petite taille, facilement combinables</li> </ul>	Qualité d'images moins bonne qu'avec Dreambooth (concept unique) ou LoRA (multi-concepts)	Méthode intéressante en combinaison avec LoRA ou Dreambooth, mais moins efficace que ces 2 méthodes quand utilisée seule

Continued on next page

Méthode	Forces	Faiblesses	Commentaires
Dreambooth	<ul style="list-style-type: none"> <li>- Entraînement très efficace</li> <li>- Excellente qualité d'images générées</li> </ul>	<ul style="list-style-type: none"> <li>- Utilisable pour l'apprentissage <b>d'un seul concept</b></li> <li>- Fort risque d'overfitting / catastrophic forgetting / language drift</li> </ul>	Méthode très efficace pour l'apprentissage d'un seul concept, encore plus efficace quand combinée avec LoRA
LoRA	<ul style="list-style-type: none"> <li>- Entraînement très efficace</li> <li>- Excellente qualité d'images générées</li> <li>- Conservation des poids du modèle initial : réduit le risque d'oubli de concepts</li> <li>- Matrices de poids générées de petite taille, facilement combinables</li> </ul>	∅	Méthode la plus appropriée pour du finetuning multi-classes
Dreambooth + LoRA	<ul style="list-style-type: none"> <li>- Entraînement très efficace</li> <li>- Excellente qualité d'images générées</li> <li>- Conservation des poids du modèle initial : réduit le risque d'oubli de concepts</li> <li>- Matrices de poids générées de petite taille, facilement combinables</li> </ul>	Utilisable pour l'apprentissage <b>d'un seul concept</b>	Méthode la plus appropriée pour l'incorporation d'un nouveau concept à un modèle pré-entraîné

Les différentes méthodes testées permettent donc de grandement améliorer l'efficacité de l'entraînement, rendant le finetuning d'un LDM réellement accessible à tous. En revanche, les problèmes liés à la durée d'échantillonnage sont toujours présents.

Un autre point d'attention est la forte dépendance de toutes ces méthodes aux prompts utilisés pendant l'entraînement

- Token d'initialisation pour Textual inversion
- Légende de chaque image pour LoRA (cf. tableau 13)
- Embedding à réapprendre pour Dreambooth

Le prompt utilisé a finalement plus d'importance que la configuration précise des hyperparamètres, et comme avec toute méthode s'appuyant sur du texte, trouver le prompt idéal n'est pas toujours évident.

## Références

- [1] Li DENG. « The mnist database of handwritten digit images for machine learning research ». In : *IEEE Signal Processing Magazine* 29.6 (2012), p. 141-142.
- [2] A. KRIZHEVSKY et G. HINTON. « Learning multiple layers of features from tiny images ». Thèse de doct. 2009.
- [3] Mircea CIMPOI et al. *Describing Textures in the Wild*. 2013. arXiv : [1311.3618 \[cs.CV\]](https://arxiv.org/abs/1311.3618).
- [4] G. E. HINTON et R. R. SALAKHUTDINOV. « Reducing the Dimensionality of Data with Neural Networks ». In : *Science* 313.5786 (juill. 2006), p. 504-507. DOI : [10.1126/science.1127647](https://doi.org/10.1126/science.1127647). URL : <https://doi.org/10.1126/science.1127647>.
- [5] Lilian WENG. « From Autoencoder to Beta-VAE ». In : *lilianweng.github.io* (2018). URL : <https://lilianweng.github.io/posts/2018-08-12-vae/>.
- [6] Diederik P KINGMA et Max WELLING. *Auto-Encoding Variational Bayes*. 2022. arXiv : [1312.6114 \[stat.ML\]](https://arxiv.org/abs/1312.6114).
- [7] Dempsterand A. P., N. M. LAIRD et D. B. RUBIN. « Maximum Likelihood from Incomplete Data via the EM Algorithm. ». In : 39 (1977), p. 1-38. URL : [https://www.ece.iastate.edu/~namrata/EE527\\_Spring08/Dempster77.pdf](https://www.ece.iastate.edu/~namrata/EE527_Spring08/Dempster77.pdf).
- [8] S.KULLBACK et LEIBLER. « On information and sufficiency ». In : 22 (1951), p. 79-86. DOI : [10.1214/aoms/1177729694](https://doi.org/10.1214/aoms/1177729694).
- [9] Aaron van den OORD, Oriol VINYALS et Koray KAVUKCUOGLU. *Neural Discrete Representation Learning*. 2018. arXiv : [1711.00937 \[cs.LG\]](https://arxiv.org/abs/1711.00937).
- [10] Martin HEUSEL et al. *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*. 2018. arXiv : [1706.08500 \[cs.LG\]](https://arxiv.org/abs/1706.08500).
- [11] Christian SZEGEDY et al. *Rethinking the Inception Architecture for Computer Vision*. 2015. arXiv : [1512.00567 \[cs.CV\]](https://arxiv.org/abs/1512.00567).
- [12] Martin ARJOVSKY, Soumith CHINTALA et Léon BOTTOU. *Wasserstein GAN*. 2017. arXiv : [1701.07875 \[stat.ML\]](https://arxiv.org/abs/1701.07875).
- [13] Patrick ESSER, Robin ROMBACH et Björn OMMER. *Taming Transformers for High-Resolution Image Synthesis*. 2021. arXiv : [2012.09841 \[cs.CV\]](https://arxiv.org/abs/2012.09841).
- [14] Jascha SOHL-DICKSTEIN et al. *Deep Unsupervised Learning using Nonequilibrium Thermodynamics*. 2015. arXiv : [1503.03585 \[cs.LG\]](https://arxiv.org/abs/1503.03585).
- [15] Jonathan Ho, Ajay JAIN et Pieter ABBEEL. *Denoising Diffusion Probabilistic Models*. 2020. arXiv : [2006.11239 \[cs.LG\]](https://arxiv.org/abs/2006.11239).
- [16] Zhisheng XIAO, Karsten KREIS et Arash VAHDAT. *Tackling the Generative Learning Trilemma with Denoising Diffusion GANs*. 2022. arXiv : [2112.07804 \[cs.LG\]](https://arxiv.org/abs/2112.07804).
- [17] W. FELLER. « On the theory of stochastic processes, with particular reference to applications ». In : *Proceedings of the [First] Berkeley Symposium on Mathematical Statistics and Probability* (1949). URL : <https://projecteuclid.org/ebooks/berkeley-symposium-on-mathematical-statistics-and-probability/Proceedings-of-the-First-Berkeley-Symposium-on-Mathematical-Statistics-and/Chapter/On-the-Theory-of-Stochastic-Processes-with-Particular-Reference-to/bmsp/1166219215.pdf>.
- [18] Olaf RONNEBERGER, Philipp FISCHER et Thomas BROX. *U-Net : Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv : [1505.04597 \[cs.CV\]](https://arxiv.org/abs/1505.04597).
- [19] Tim SALIMANS et al. *PixelCNN++ : Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications*. 2017. arXiv : [1701.05517 \[cs.LG\]](https://arxiv.org/abs/1701.05517).
- [20] Vaibhav SINGH. « An In-Depth Guide to Denoising Diffusion Probabilistic Models – From Theory to Implementation ». In : *learnopencv.com* (mars 2023). URL : <https://learnopencv.com/denoising-diffusion-probabilistic-models/#Model-Architecture-Used-In-DDPMs>.
- [21] Jean-Baptiste CORDONNIER, Andreas LOUKAS et Martin JAGGI. *On the Relationship between Self-Attention and Convolutional Layers*. 2020. arXiv : [1911.03584 \[cs.LG\]](https://arxiv.org/abs/1911.03584).
- [22] Jay ALAMMAR. « The Illustrated Transformer ». In : *jalammar.github.io* (juin 2018). URL : [http://jalammar.github.io/illustrated-transformer/](https://jalammar.github.io/illustrated-transformer/).
- [23] Ashish VASWANI et al. *Attention Is All You Need*. 2017. arXiv : [1706.03762 \[cs.CL\]](https://arxiv.org/abs/1706.03762).
- [24] Dan HENDRYCKS et Kevin GIMPEL. *Gaussian Error Linear Units (GELUs)*. 2023. arXiv : [1606.08415 \[cs.LG\]](https://arxiv.org/abs/1606.08415).
- [25] Alex NICHOL et Prafulla DHARIWAL. *Improved Denoising Diffusion Probabilistic Models*. 2021. arXiv : [2102.09672 \[cs.LG\]](https://arxiv.org/abs/2102.09672).
- [26] Robin ROMBACH et al. « High-Resolution Image Synthesis with Latent Diffusion Models ». In : (2021). arXiv : [2112.10752 \[cs.CV\]](https://arxiv.org/abs/2112.10752).
- [27] Richard ZHANG et al. *The Unreasonable Effectiveness of Deep Features as a Perceptual Metric*. 2018. arXiv : [1801.03924 \[cs.CV\]](https://arxiv.org/abs/1801.03924).

- [28] Alina KUZNETSOVA et al. « The Open Images Dataset V4 ». In : *International Journal of Computer Vision* 128.7 (mars 2020), p. 1956-1981. DOI : [10.1007/s11263-020-01316-z](https://doi.org/10.1007/s11263-020-01316-z). URL : <https://doi.org/10.1007/s11263-020-01316-z>.
- [29] Alec RADFORD et al. *Learning Transferable Visual Models From Natural Language Supervision*. 2021. arXiv : [2103.00020 \[cs.CV\]](https://arxiv.org/abs/2103.00020).
- [30] Yonghui WU et al. *Google's Neural Machine Translation System : Bridging the Gap between Human and Machine Translation*. 2016. arXiv : [1609.08144 \[cs.CL\]](https://arxiv.org/abs/1609.08144).
- [31] Alexey DOSOVITSKIY et al. *An Image is Worth 16x16 Words : Transformers for Image Recognition at Scale*. 2021. arXiv : [2010.11929 \[cs.CV\]](https://arxiv.org/abs/2010.11929).
- [32] Mehdi CHERTI et al. *Reproducible scaling laws for contrastive language-image learning*. 2022. arXiv : [2212.07143 \[cs.LG\]](https://arxiv.org/abs/2212.07143).
- [33] Christoph SCHUHMANN et al. *LAION-5B : An open large-scale dataset for training next generation image-text models*. 2022. arXiv : [2210.08402 \[cs.CV\]](https://arxiv.org/abs/2210.08402).
- [34] Tim SALIMANS et Jonathan HO. *Progressive Distillation for Fast Sampling of Diffusion Models*. 2022. arXiv : [2202.00512 \[cs.LG\]](https://arxiv.org/abs/2202.00512).
- [35] Nicholas GUTTENBERG. « Diffusion With Offset Noise ». In : *crosslabs.org* (jan. 2023). URL : <https://www.crosslabs.org/blog/diffusion-with-offset-noise>.
- [36] Shanchuan LIN et al. *Common Diffusion Noise Schedules and Sample Steps are Flawed*. 2023. arXiv : [2305.08891 \[cs.CV\]](https://arxiv.org/abs/2305.08891).
- [37] Rinon GAL et al. *An Image is Worth One Word : Personalizing Text-to-Image Generation using Textual Inversion*. 2022. arXiv : [2208.01618 \[cs.CV\]](https://arxiv.org/abs/2208.01618).
- [38] Nataniel RUIZ et al. *DreamBooth : Fine Tuning Text-to-Image Diffusion Models for Subject-Driven Generation*. 2023. arXiv : [2208.12242 \[cs.CV\]](https://arxiv.org/abs/2208.12242).
- [39] Edward J. HU et al. *LoRA : Low-Rank Adaptation of Large Language Models*. 2021. arXiv : [2106.09685 \[cs.CL\]](https://arxiv.org/abs/2106.09685).
- [40] Chunyuan LI et al. *Measuring the Intrinsic Dimension of Objective Landscapes*. 2018. arXiv : [1804.08838 \[cs.LG\]](https://arxiv.org/abs/1804.08838).
- [41] Armen AGHAJANYAN, Luke ZETTLEMOYER et Sonal GUPTA. *Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning*. 2020. arXiv : [2012.13255 \[cs.LG\]](https://arxiv.org/abs/2012.13255).
- [42] Daniel ROICH et al. *Pivotal Tuning for Latent-based Editing of Real Images*. 2021. arXiv : [2106.05744 \[cs.CV\]](https://arxiv.org/abs/2106.05744).