Aix-Marseille Université

Implémentation et analyse de VAE denses et convolutionnels pour la génération d'images MNIST

Antoine Legendre

Mathématiques pour la Science des Données

Année Universitaire 2025-2026

Table des matières

1 Introduction

L'apprentissage non supervisé occupe une place centrale dans le domaine de l'intelligence artificielle, notamment pour la représentation et la génération de données complexes. Parmi les modèles récents les plus influents, les autoencodeurs variationnels (VAE) constituent une approche probabiliste permettant à la fois la compression d'informations et la génération de nouvelles données cohérentes avec un jeu d'apprentissage. Introduits par Kingma et Welling (2013), les VAE combinent les principes des réseaux de neurones profonds et de l'inférence bayésienne, offrant un cadre théorique solide pour l'apprentissage d'espaces latents continus.
L'objectif de cette séance de travaux pratiques est de concevoir et d'expérimenter plusieurs architectures de VAE à l'aide de la bibliothèque PyTorch, en les appliquant à la base de données MNIST, qui contient des images de chiffres manuscrits. Tout d'abord, nous allons construire les modèles de VAE, en développant successivement les modules d'encodage et de décodage, sous deux formes architecturales : l'une basée sur des couches entièrement connectées (denses), et l'autre sur des couches convolutionnelles. Ensuite, le modèle est entraîné sur les données MNIST, à l'aide d'une fonction de perte combinant un terme de reconstruction et un terme de régularisation. Enfin, la dernière partie consiste à expérimenter le VAE appris : génération de nouvelles images, visualisation de l'espace latent, et comparaison des performances entre les deux architectures proposées. L'impact du facteur de pondération β du terme de régularisation sur la qualité de reconstruction et la structuration de l'espace latent sera également étudié.

2 VAE dense

2.1 Encodeur

L'encodeur `DenseEncoder` transforme une image x (de dimension 28×28) en une représentation latente de plus faible dimension. Le code suivant définit un réseau entièrement connecté dont la sortie correspond aux paramètres $\mu_\phi(x)$ et $\log \sigma^2_\phi(x)$ d'une distribution gaussienne multivariée $q_\phi(z|x)$:

```
class DenseEncoder(nn.Module):
    def __init__(self, input_dim=784, latent_dim=20, hidden_dims
       =[512, 256]):
        super(DenseEncoder, self).__init__()
        self.fc_layers = nn.ModuleList()
        prev_dim = input_dim
        for h_dim in hidden_dims:
            self.fc_layers.append(nn.Linear(prev_dim, h_dim))
            prev_dim = h_dim
        self.fc_mu = nn.Linear(prev_dim, latent_dim)
        self.fc_logvar = nn.Linear(prev_dim, latent_dim)
    def forward(self, x):
        x = x.view(x.size(0), -1)
        for layer in self.fc_layers:
            x = F.relu(layer(x))
        mu = self.fc_mu(x)
        logvar = self.fc_logvar(x)
        return mu, logvar
```

Le réseau comporte ici deux couches cachées de tailles 512 et 256, suivies de deux couches linéaires distinctes produisant les vecteurs $\mu_\phi(x)$ et $\log \sigma^2_\phi(x)$. L'entrée est aplatie en un vecteur de 784 composantes, puis transformée par des activations ReLU. Cet encodeur paramètre la distribution latente à partir de laquelle sera échantillonnée la variable z.

2.2 Décodeur

Le décodeur `DenseDecoder` reconstruit une image $\hat{x}$ à partir d'un vecteur latent z selon la distribution $p_\theta(x|z)$. Il est défini comme suit :

```
class DenseDecoder(nn.Module):
    def __init__(self, latent_dim=20, output_dim=784, hidden_dims
       =[256, 512]):
        super(DenseDecoder, self).__init__()
        self.fc_layers = nn.ModuleList()
        prev_dim = latent_dim
        for h_dim in hidden_dims:
            self.fc_layers.append(nn.Linear(prev_dim, h_dim))
            prev_dim = h_dim
        self.fc_out = nn.Linear(prev_dim, output_dim)
    def forward(self, z):
        x = z
        for layer in self.fc_layers:
            x = F.relu(layer(x))
        x = torch.sigmoid(self.fc_out(x))
        x = x.view(-1, 1, 28, 28)
        return x
```

Ce décodeur adopte une structure symétrique à celle de l'encodeur. À partir du vecteur latent $z \in \mathbb{R}^{20}$, il reconstruit un vecteur de dimension 784, remodelé ensuite en image 28×28. L'activation sigmoïde finale assure que les valeurs de sortie soient comprises entre 0 et 1, cohérentes avec les intensités des pixels normalisés.

2.3 Architecture du VAE

Le module `VAE` combine les deux sous-réseaux précédents — l'encodeur et le décodeur — afin de constituer un autoencodeur variationnel complet. Son fonctionnement repose sur le principe du rééchantillonnage, qui permet de rendre le processus d'échantillonnage différentiable.

```python
class VAE(nn.Module):
    def __init__(self, encoder, decoder):
        super(VAE, self).__init__()
        self.encoder = encoder
        self.decoder = decoder

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        z = mu + eps * std
        return z

    def forward(self, x):
        mu, logvar = self.encoder(x)
        z = self.reparameterize(mu, logvar)
        x_recon = self.decoder(z)
        return x_recon, mu, logvar

    def decode(self, z):
        return self.decoder(z)

    def encode(self, x):
        return self.encoder(x)
```

La classe reçoit en argument un `encoder` et un `decoder`, préalablement définis (par exemple sous la forme dense ou convolutionnelle). L'initialisation `__init__` associe ces deux modules à l'objet `VAE`.
La méthode `reparameterize` met en œuvre le *trick de rééchantillonnage*. À partir des paramètres μ et $\log \sigma^2$ produits par l'encodeur, on calcule :

$$\text{std} = e^{\frac{1}{2}\log \sigma^2}, \qquad z = \mu + \epsilon \times \text{std}, \quad \text{avec } \epsilon \sim \mathcal{N}(0, I).$$

Cette opération permet de rendre la variable z différentiable par rapport à μ et $\log \sigma^2$, ce qui est essentiel pour la rétropropagation du gradient.
La méthode `forward` définit le passage complet des données dans le réseau :

$$x \xrightarrow{\text{encodeur}} (\mu, \log \sigma^2) \xrightarrow{\text{rééchantillonnage}} z \xrightarrow{\text{décodeur}} \hat{x}.$$

Elle retourne à la fois l'image reconstruite $\hat{x}$, et les paramètres μ et $\log \sigma^2$, nécessaires pour le calcul de la fonction de perte.
Les méthodes auxiliaires `encode` et `decode` permettent d'utiliser séparément les deux parties du modèle : la première pour obtenir la représentation latente, la seconde pour générer de nouvelles images à partir d'un vecteur z.
Le modèle ainsi défini constitue la structure complète d'un autoencodeur variationnel. Il permet à la fois d'apprendre une représentation probabiliste compacte des données et de générer de nouveaux exemples plausibles à partir de l'espace latent.

2.4 Fonction de perte et apprentissage du VAE

L'entraînement du VAE repose sur la minimisation d'une fonction de perte combinant deux termes : une erreur de reconstruction entre l'image d'origine et l'image reconstruite,

et une régularisation par la divergence de Kullback–Leibler (KL), qui contraint la distribution latente apprise à rester proche d'une loi normale standard.

```
def vae_loss(x_recon, x, mu, logvar):
    recon_loss = F.binary_cross_entropy(x_recon, x, reduction='sum')
    kl_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return recon_loss + beta * kl_loss
```

Le terme de reconstruction :

$$\mathcal{L}_{\text{recon}} = \text{BCE}(x_{\text{recon}}, x)$$

correspond à une *binary cross-entropy* (BCE) entre les pixels de l'image originale x et ceux de l'image reconstruite $\hat{x}_{\text{recon}}$. Il mesure la fidélité de la reconstruction. L'option `reduction='sum'` indique que la perte est sommée sur tous les pixels et sur le lot d'images.
Le terme de régularisation :

$$\mathcal{L}_{\text{KL}} = -\frac{1}{2}\sum(1 + \log\sigma^2 - \mu^2 - e^{\log\sigma^2})$$

est la divergence de Kullback–Leibler entre la distribution latente $q_\phi(z|x)$ et la loi normale standard $\mathcal{N}(0, I)$. Ce terme encourage la distribution apprise à rester proche d'une gaussienne centrée et réduit le surapprentissage.
Le terme de pondération β : le coefficient `beta` permet de moduler l'importance du terme KL. Plus β est grand, plus on impose une régularisation plus forte (VAE-β), favorisant un espace latent plus structuré mais au prix d'une reconstruction légèrement moins précise.
La perte totale à minimiser est donc :

$$\mathcal{L}_{\text{VAE}} = \mathcal{L}_{\text{recon}} + \beta * \mathcal{L}_{\text{KL}}.$$

On entraîne ensuite le modèle sur plusieurs époques à l'aide de cette fonction de perte :

```
def train_vae(model, dataloader, optimizer, num_epochs=10):
    model.train()
    for epoch in range(num_epochs):
        train_loss = 0
        for x, _ in dataloader:
            x = x.to(device)
            optimizer.zero_grad()
            x_recon, mu, logvar = model(x)
            loss = vae_loss(x_recon, x, mu, logvar)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
        avg_loss = train_loss / len(dataloader.dataset)
        print(f"Epoch [{epoch+1}/{num_epochs}] - Average loss: {avg_loss:.4f}")
```

Les images x sont envoyées sur le périphérique de calcul (`CPU` ou `GPU`). Le passage avant (`forward`) produit la reconstruction $\hat{x}$ et les paramètres latents $(\mu, \log\sigma^2)$. La perte totale est calculée, rétropropagée (`loss.backward()`), puis les poids du réseau sont mis à jour par l'optimiseur. À chaque époque, la perte moyenne sur l'ensemble du jeu d'entraînement est affichée.

2.5 Génération et visualisation à partir de l'espace latent

Une fois le VAE entraîné, il est possible de générer de nouvelles images en échantillonnant directement dans l'espace latent, ainsi que d'analyser la structure de cet espace. Les fonctions suivantes assurent la génération, la visualisation et l'interpolation de points latents.

2.5.1 Génération d'échantillons aléatoires

```
def generate_samples(vae, latent_dim=2, n_samples=25, device='cpu'):
    vae.eval()
    with torch.no_grad():
        z = torch.randn(n_samples, latent_dim).to(device)
        generated = vae.decode(z).cpu()
        if generated.dim() == 2:
            generated = generated.view(-1, 1, 28, 28)
    return generated
```

Cette fonction met le modèle en mode évaluation (`vae.eval()`) et désactive le calcul des gradients. Elle génère n_{samples} vecteurs latents $z \sim \mathcal{N}(0, I)$ dans $\mathbb{R}^{d_z}$, puis les décode en images via le décodeur du VAE. Les images obtenues sont ensuite remodelées au format $(1, 28, 28)$ et renvoyées pour affichage.

2.5.2 Affichage des images générées

```
def plot_generated_images(images, latent_dim, n_rows=5, n_cols=5,
    save_dir="results"):
    os.makedirs(save_dir, exist_ok=True)
    fig, axes = plt.subplots(n_rows, n_cols, figsize=(6,6))
    for i, ax in enumerate(axes.flat):
        ax.imshow(images[i][0], cmap='gray')
        ax.axis('off')
    plt.suptitle(f"Images␣générées␣par␣le␣VAE␣(latent_dim={
        latent_dim})", fontsize=14)
    save_path = os.path.join(save_dir, f"images_gen_latent{
        latent_dim}.png")
    plt.savefig(save_path, bbox_inches='tight')
    plt.close(fig)
```

Cette fonction affiche un ensemble d'images générées à partir de l'espace latent et les sauvegarde dans le dossier `results`. Chaque sous-graphe représente une image synthétisée par le modèle.

2.5.3 Visualisation de l'espace latent

```
def plot_latent_space(vae, data_loader, device='cpu', latent_dim=2,
    save_dir="results"):
    os.makedirs(save_dir, exist_ok=True)
    vae.eval()
    all_mu, all_labels = [], []
    with torch.no_grad():
        for x, y in data_loader:
            x = x.to(device)
            mu, logvar = vae.encode(x)
            all_mu.append(mu.cpu().numpy())
            all_labels.append(y.numpy())
    all_mu = np.concatenate(all_mu)
    all_labels = np.concatenate(all_labels)
    if latent_dim > 2:
        all_mu_2d = TSNE(n_components=2, learning_rate='auto', init
           ='random').fit_transform(all_mu)
    else:
        all_mu_2d = all_mu
    plt.figure(figsize=(8,6))
    scatter = plt.scatter(all_mu_2d[:, 0], all_mu_2d[:, 1], c=
       all_labels, cmap='tab10', s=5)
    plt.colorbar(scatter, label='Chiffre␣réel')
    plt.title(f"Représentation␣de␣l'espace␣latent␣(latent_dim={
       latent_dim})")
    plt.xlabel("z_1")
    plt.ylabel("z_2")
    save_path = os.path.join(save_dir, f"latent_space_latent{
       latent_dim}.png")
    plt.savefig(save_path, bbox_inches='tight')
    plt.close()
    print(f"Image␣enregistrée␣:␣{save_path}")
```

Cette fonction permet de visualiser la structure de l'espace latent en projetant les représentations $\mu_\phi(x)$ des données d'entrée. Pour une dimension latente $d_z = 2$, les points (z_1, z_2) sont directement tracés. Pour $d_z > 2$, une réduction de dimension est réalisée à l'aide de l'algorithme `t-SNE`. Chaque point du nuage correspond à une image MNIST et est coloré selon le chiffre réel associé, permettant d'observer la séparation des classes dans l'espace latent.

2.5.4 Interpolation dans l'espace latent

```
def interpolate_points(z1, z2, n_steps=10):
    ratios = np.linspace(0, 1, n_steps)
    return torch.stack([(1 - r) * z1 + r * z2 for r in ratios])
```

Cette fonction calcule une interpolation linéaire entre deux vecteurs latents z_1 et z_2, générant une série de points intermédiaires :

$$z(r) = (1 - r)z_1 + rz_2, \quad r \in [0, 1].$$

Cela permet de visualiser la continuité et la cohérence des représentations apprises par le modèle.

```
def plot_interpolation(vae, latent_dim=2, device='cpu', save_dir="
    results"):
    os.makedirs(save_dir, exist_ok=True)
    vae.eval()
    z1, z2 = torch.randn(2, latent_dim).to(device)
    z_interp = interpolate_points(z1, z2, n_steps=10)
    with torch.no_grad():
        imgs = vae.decode(z_interp).cpu().view(-1, 28, 28)
    fig, axes = plt.subplots(1, 10, figsize=(10,1))
    for i, ax in enumerate(axes.flat):
        ax.imshow(imgs[i], cmap='gray')
        ax.axis('off')
    plt.suptitle(f"Interpolation dans l'espace latent (latent_dim={
        latent_dim})", fontsize=14)
    save_path = os.path.join(save_dir, f"interpolation_latent{
        latent_dim}.png")
    plt.savefig(save_path, bbox_inches='tight')
    plt.close(fig)
    print(f"Image enregistrée : {save_path}")
```

Cette dernière fonction illustre la capacité du VAE à effectuer une interpolation entre deux points de l'espace latent. Les images générées montrent une transition progressive entre deux chiffres aléatoires, témoignant de la continuité de la représentation apprise par le modèle.

2.6 Test de différentes configurations

Afin de déterminer la meilleure architecture pour l'encodeur et le décodeur denses, nous avons testé plusieurs configurations en variant la dimension de l'espace latent et les tailles des couches cachées. L'objectif était de trouver un compromis satisfaisant entre nombre de paramètres et qualité des images générées.
Nous avons testé les configurations suivantes :

- Dimension latente : nous avons testé $d_z = 2, 5, 10, 20, 50, 100$.
- Réseaux cachés : trois architectures ont été évaluées :
  1. $128 \to 256$
  2. $256 \to 512$
  3. $512 \to 1024$

Pour chaque configuration, nous avons entraîné le VAE sur le jeu de données MNIST et évalué qualitativement la reconstruction des images et la génération d'exemples synthétiques à partir de l'espace latent.
Les observations principales sont les suivantes :

- Une dimension latente trop petite limite la capacité du modèle à représenter la variété des chiffres, entraînant des reconstructions floues.
- Une dimension latente trop grande ($d_z \geq 50$) augmente significativement le nombre de paramètres sans amélioration perceptible de la qualité des images générées.

— Des réseaux très larges ($512 \to 1024$) offrent peu de gain qualitatif par rapport aux architectures moyennes mais augmentent considérablement le coût en mémoire et le temps d'entraînement.

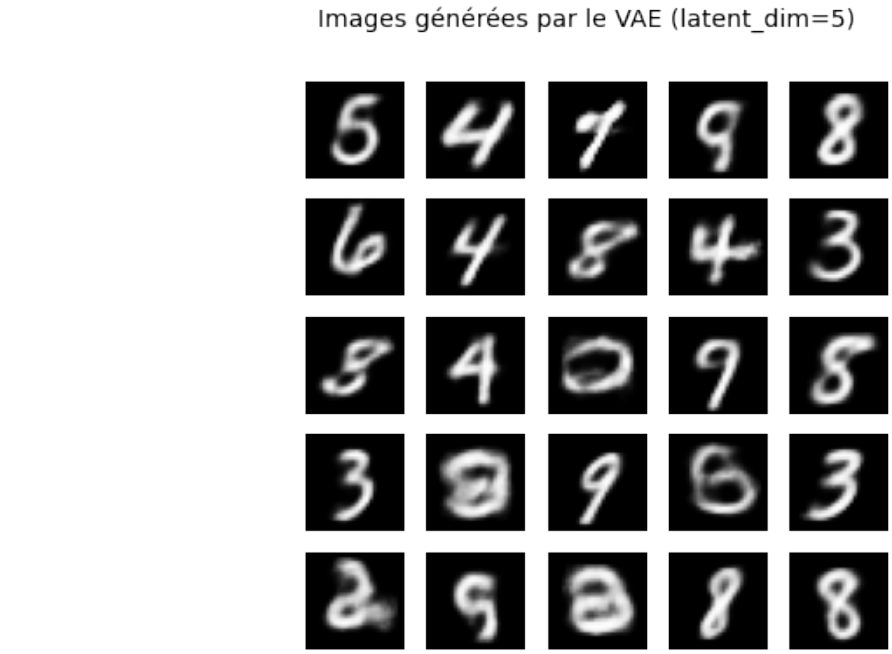
Ainsi, pour MNIST, nous avons choisi une architecture intermédiaire :

— Couches cachées : $256 \to 512$

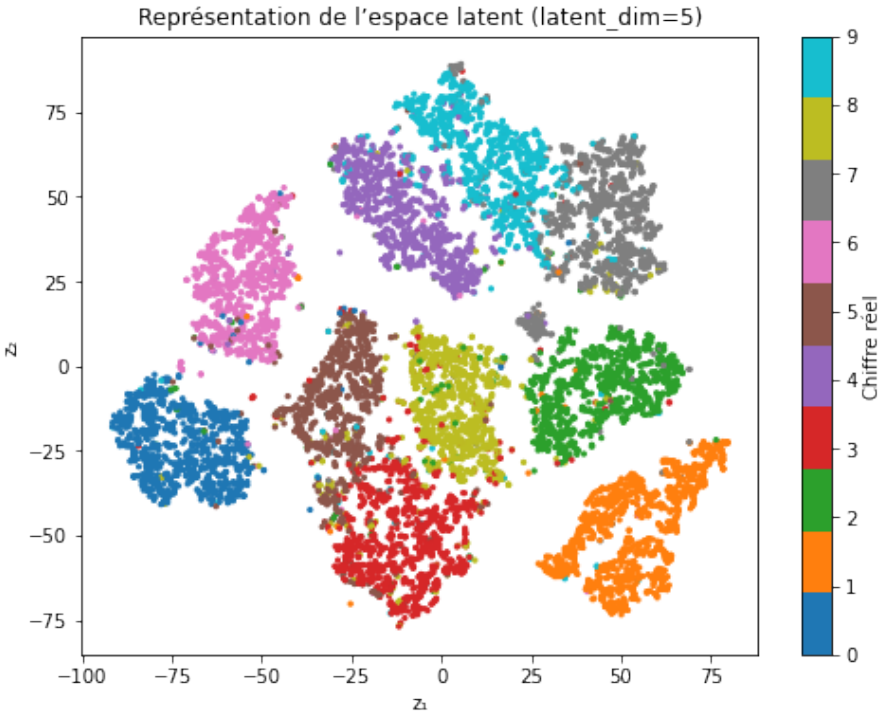
— Dimension latente : $d_z = 5$

Cette configuration permet de générer des images suffisamment réalistes tout en maintenant un nombre de paramètres raisonnable (1 071 130), garantissant un entraînement rapide et une bonne généralisation.

Voici ci-dessous trois types d'images générées par le modèle choisi.



(a) Images générées à partir de z aléatoires



(b) Visualisation de l'espace latent

(c) Interpolation dans l'espace latent

FIGURE 1 – Exemples d'images obtenues avec le VAE dense (256,512, latent dim=5)

3 VAE convolutionnel

Pour tirer parti de la structure spatiale des images, nous allons également implémenter une version convolutionnelle du VAE. Cette architecture permet de capturer des motifs locaux plus efficacement qu'un réseau entièrement connecté.

3.1 Encodeur

```
class ConvEncoder(nn.Module):
    def __init__(self, latent_dim=20):
        super(ConvEncoder, self).__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=0),
            nn.ReLU(),
        )
        self.flatten = nn.Flatten()
        self.fc_mu = nn.Linear(128 * 5 * 5, latent_dim)
        self.fc_logvar = nn.Linear(128 * 5 * 5, latent_dim)

    def forward(self, x):
        x = self.conv_layers(x)
        x = self.flatten(x)
        mu = self.fc_mu(x)
        logvar = self.fc_logvar(x)
        return mu, logvar
```

L'encodeur reçoit une image $x \in \mathbb{R}^{1\times 28\times 28}$. Trois couches convolutionnelles successives, chacune suivie d'une activation ReLU, extraient des caractéristiques locales. La sortie des convolutions est aplatie en un vecteur, puis transformée par deux couches linéaires séparées pour produire les paramètres $\mu_\phi(x)$ et $\log \sigma^2_\phi(x)$ de la distribution latente.

3.2 Décodeur

```
class ConvDecoder(nn.Module):
    def __init__(self, latent_dim=20):
        super(ConvDecoder, self).__init__()
        self.fc = nn.Linear(latent_dim, 128 * 5 * 5)
        self.deconv_layers = nn.Sequential(
            nn.ConvTranspose2d(128, 64, kernel_size=3, stride=1,
               padding=0),
            nn.ReLU(),
            nn.ConvTranspose2d(64, 32, kernel_size=4, stride=2,
               padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 1, kernel_size=4, stride=2,
               padding=1),
            nn.Sigmoid()
        )

    def forward(self, z):
        x = F.relu(self.fc(z))
        x = x.view(-1, 128, 5, 5)
        x = self.deconv_layers(x)
        return x
```

Le décodeur prend un vecteur latent $z \in \mathbb{R}^{d_z}$. Une couche linéaire le projette en un tenseur de forme $(128, 5, 5)$. Trois couches de transposition de convolution (`ConvTranspose2d`) reconvertissent progressivement ce tenseur en image de taille $(1, 28, 28)$. La dernière activation sigmoïde contraint les pixels reconnus entre 0 et 1.
Le VAE convolutionnel est ensuite construit, entraîné et évalué de la même manière que le VAE dense (voir sous-sections 2.3, 2.4 et 2.5).

3.3 Test de différentes configurations

Pour la partie convolutionnelle, nous avons également exploré différentes architectures afin de trouver un compromis optimal entre nombre de paramètres et qualité des images générées.
Nous avons testé plusieurs dimensions de l'espace latent :

$$d_z = 2, 5, 10, 20, 50, 100$$

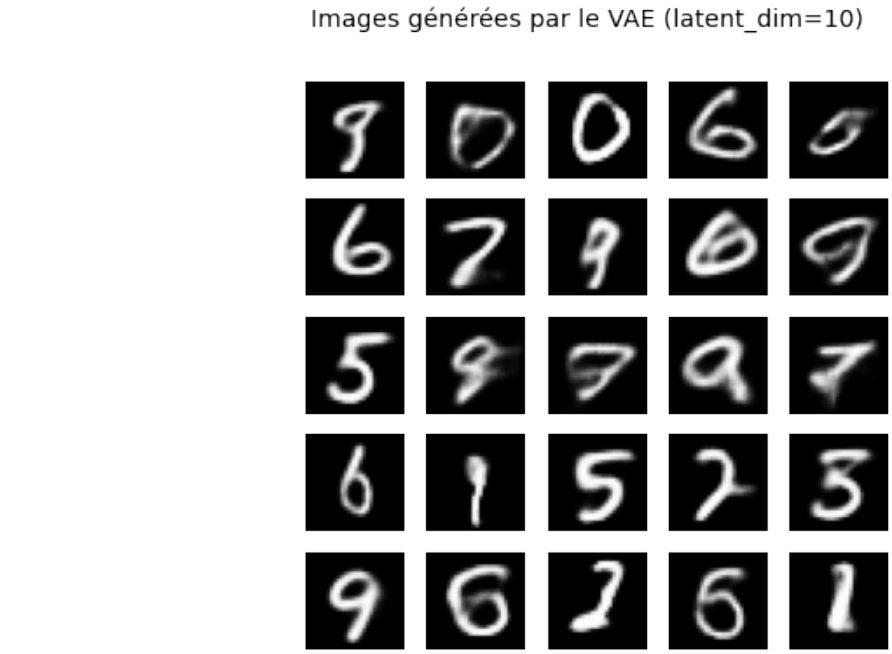
Et trois types de réseaux convolutifs différents :

1. Réseau à 2 couches : $1 \to 32$, $32 \to 64$
2. Réseau à 3 couches : $1 \to 16$, $16 \to 32$, $32 \to 64$
3. Réseau à 3 couches : $1 \to 32$, $32 \to 64$, $64 \to 128$

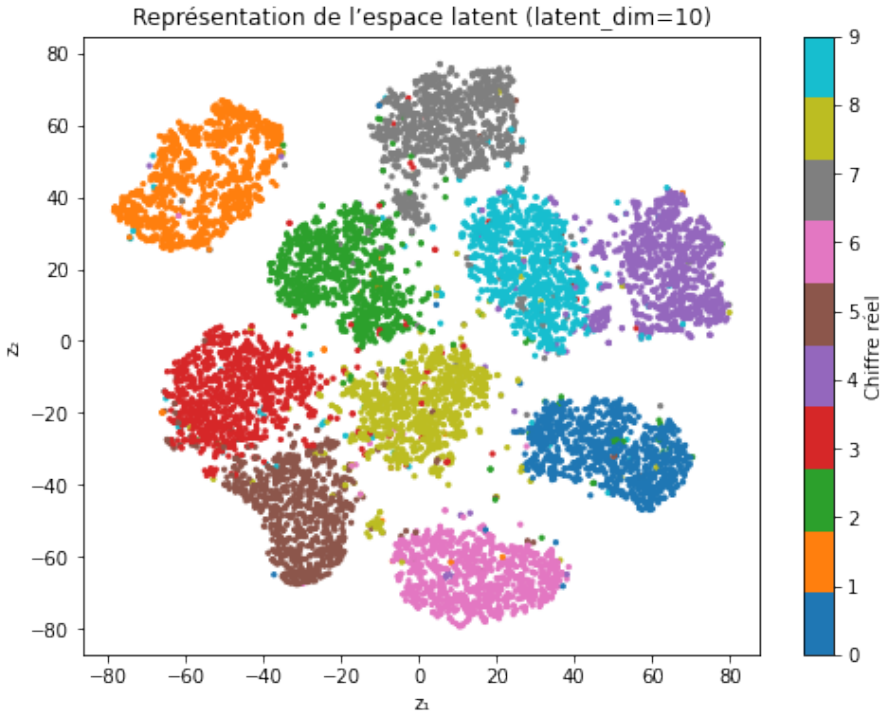
Pour chaque configuration, nous avons entraîné le VAE sur le jeu de données MNIST et évalué qualitativement les reconstructions et les images générées à partir de l'espace latent.
Les observations principales sont les suivantes :

— Les architectures à 2 couches étaient trop simples pour représenter la variété des chiffres lorsque la dimension latente était élevée, produisant des reconstructions légèrement floues.
— Le réseau 3 couches avec $1 \to 16 \to 32 \to 64$ offrait une qualité satisfaisante mais bien inférieure au réseau 3 couches avec $1 \to 32 \to 64 \to 128$.
— Le réseau 3 couches avec $1 \to 32 \to 64 \to 128$ combiné à une dimension latente $d_z = 10$ offre une reconstruction et une génération d'images de très bonne qualité tout en gardant un nombre de paramètres raisonnable (313 557).

Voici ci-dessous trois types d'images générées par le modèle final.



(a) Images générées à partir de z aléatoires



(b) Visualisation de l'espace latent

(c) Interpolation dans l'espace latent

FIGURE 2 – Exemples d'images obtenues avec le VAE convolutionnel (3 couches 1→32→64→128, latent dim=10)

On note, comme attendu, que les images générées par le VAE convolutionnel sont de meilleures qualités que celles générées par le VAE dense, l'espace latent est également plus structuré et le nombre de paramètres du modèle choisi pour les VAE convolutionnels est grandement inférieur aux nombres de paramètres du modèle choisi pour les VAE denses (313 557 contre 1 071 130). On va donc se pencher sur l'étude du réseau 3 couches avec $1 \to 32 \to 64 \to 128$ combiné à une dimension latente $d_z = 10$ dans la suite de ce rapport.

4 Effet de la pondération β

Après avoir choisi notre architecture convolutionnelle optimale (3 couches : $1 \to 32 \to 64 \to 128$, dimension latente $d_z = 10$), nous avons étudié l'effet de la pondération β dans la fonction de perte VAE. Rappelons que la perte VAE pondérée se définit comme :

$$\mathcal{L}_\beta = \text{BCE}(x, \hat{x}) + \beta \cdot \text{KL}(q_\phi(z|x) \| p(z))$$

Nous avons testé plusieurs valeurs de β :

$$\beta \in \{0.2, 0.5, 1.0, 2.0, 5.0\}$$

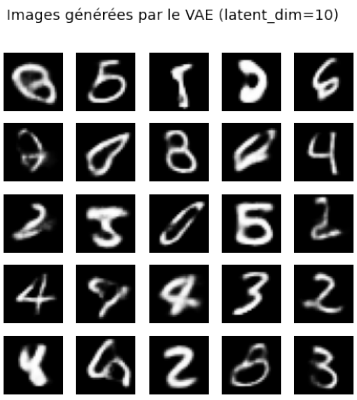
Pour $\beta < 1$ (ex. $\beta = 0.2$), le modèle tend à privilégier la reconstruction des images. Les chiffres reconstruits sont nets et proches des données d'entrée, mais l'espace latent est moins régularisé, ce qui peut conduire à des zones "vides" ou incohérentes lors de la génération d'échantillons aléatoires.
Pour $\beta = 1$, on obtient un compromis entre reconstruction et régularisation. L'espace latent est plus structuré et les images générées restent de bonne qualité.
Pour des $\beta > 1$ (ex. $\beta = 5$), le modèle favorise fortement la régularisation du latent. Cela produit un espace latent très structuré, mais les reconstructions et générations deviennent plus floues et certaines caractéristiques des chiffres peuvent être perdues.

L'effet principal de β est de contrôler le compromis entre fidélité de reconstruction et structure de l'espace latent. Une valeur trop faible conduit à un latent peu exploitable pour la génération, tandis qu'une valeur trop grande détériore la qualité visuelle des images.
Pour notre architecture et le jeu de données MNIST, un $\beta \approx 1$ semble optimal, car il offre un bon compromis : des images générées suffisamment nettes tout en maintenant un espace latent régulier et exploitable.
Des illustrations des images reconstruites et générées pour diférents β sont présentées en Figure 3.

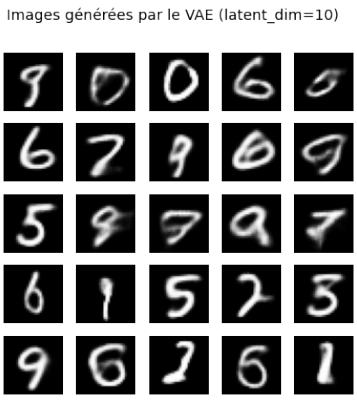
(a) Images reconstruites pour $\beta = 0.2$

(b) Images reconstruites pour $\beta = 1$

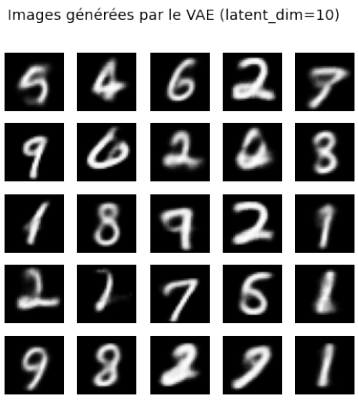
(c) Images reconstruites pour $\beta = 5$



(d) Images générées pour $\beta = 0.2$



(e) Images générées pour $\beta = 1$



(f) Images générées pour $\beta = 5$

FIGURE 3 – Images reconstruites et générées pour différents β

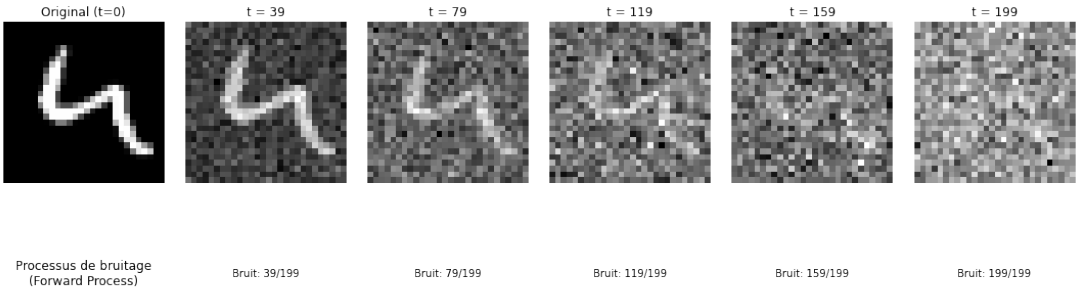
5 Conclusion et perspectives

Dans ce travail pratique, nous avons exploré la mise en œuvre de Variational Autoencoders (VAE) pour la génération d'images de chiffres manuscrits à partir du jeu de données MNIST.
Nous avons d'abord implémenté et testé des VAE à base de réseaux denses, en étudiant différentes architectures et tailles de dimension latente. Nous avons observé qu'un réseau dense avec deux couches cachées de taille $256 \to 512$ et une dimension latente $d_z = 5$ offrait un bon compromis entre nombre de paramètres et qualité des images générées.
Nous avons ensuite exploré les réseaux convolutionnels, en comparant trois architectures :

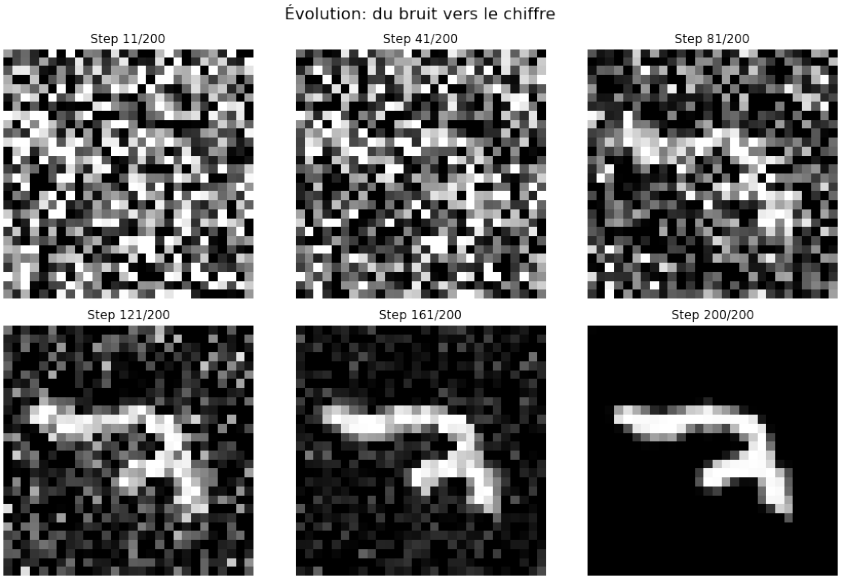
1. 2 couches : $1 \to 32 \to 64$,
2. 3 couches : $1 \to 16 \to 32 \to 64$,
3. 3 couches : $1 \to 32 \to 64 \to 128$.

Le troisième type de réseau avec une dimension latente $d_z = 10$ a été retenu, offrant à la fois une bonne qualité de génération et un nombre de paramètres raisonnable.
Nous avons étudié l'effet de la pondération β dans la fonction de perte VAE, en testant $\beta \in \{0.2, 0.5, 1.0, 2.0, 5.0\}$. Nous avons constaté qu'un $\beta < 1$ favorise la reconstruction, mais l'espace latent est moins structuré, qu'un $\beta \approx 1$ fournit un bon compromis entre qualité des reconstructions et régularisation du latent, et que $\beta > 1$ améliore la structure du latent mais détériore légèrement la qualité visuelle des images reconstruites.

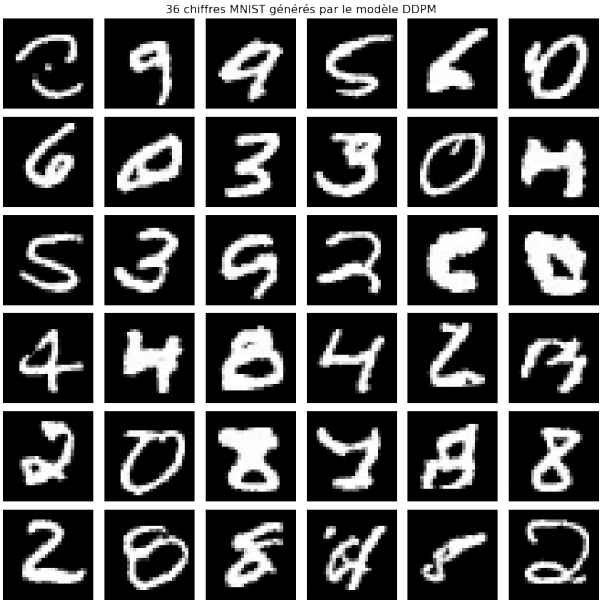
Bien que les VAE permettent de générer des images relativement réalistes et de manipuler l'espace latent, leur qualité est limitée par la capacité des modèles et la tendance à produire des images légèrement floues. Des méthodes plus récentes, telles que les Denoising Diffusion Probabilistic Models (DDPM), offrent des résultats de génération bien plus précis et détaillés.
Un DDPM apprend à générer des images en apprenant un processus de diffusion inverse : il part d'un bruit aléatoire et apprend progressivement à reconstruire des images réalistes en inversant un processus de corruption progressive par bruit. Ces modèles ont récemment permis de générer des images de haute qualité dans de nombreux domaines, allant des chiffres manuscrits aux images naturelles complexes.
Le DDPM que nous avons implémenté repose sur un réseau de type U-Net, adapté au jeu de données MNIST. À chaque étape temporelle t, un encodage sinusoïdal (`SinusoidalPositionEmbeddings`) est utilisé pour injecter l'information temporelle dans le réseau. Celui-ci est composé de blocs convolutionnels (`SimpleBlock`) intégrant ces encodages temporels à chaque niveau de l'encodeur et du décodeur.
La classe `DDPM` gère le processus complet : la planification des coefficients de bruit β_t, la phase d'entraînement, où le réseau apprend à prédire le bruit ajouté à chaque étape et la phase de génération, où le modèle part d'un bruit aléatoire et applique le processus inverse pour produire une image (voir code en Annexes). Ainsi, le DDPM apprend implicitement la distribution des images en inversant le processus de diffusion gaussienne.
Les images présentées en Figure 4 illustrent le processus de bruitage, de débruitage et quelques résultats obtenus à l'aide de ce réseau.



(a) Processus de bruitage



(b) Évolution du bruit vers le chiffre



(c) Exemple de chiffres MNIST générés par le modèle DDPM

FIGURE 4 – Résultats du modèle DDPM

Ainsi, une évolution naturelle de ce travail serait d'étudier et de comparer les performances des VAE convolutionnels et des DDPM, en termes de qualité des images générées, ainsi que dans la manière dont chacun apprend et exploite des représentations internes, latentes pour les VAE ou implicites pour les DDPM.

Annexes

Modèle DDPM

```
class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = np.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device=device) * -embeddings)
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        return embeddings

class SimpleBlock(nn.Module):
    def __init__(self, in_channels, out_channels, time_emb_dim):
        super().__init__()

        self.conv1 = nn.Conv2d(in_channels, out_channels, 3, padding=1)
        self.conv2 = nn.Conv2d(out_channels, out_channels, 3, padding=1)
        self.time_mlp = nn.Linear(time_emb_dim, out_channels)

    def forward(self, x, t_emb):
        h = F.silu(self.conv1(x))

        time_emb = self.time_mlp(t_emb)
        time_emb = time_emb[(..., ) + (None, ) * 2]
        h = h + time_emb

        h = F.silu(self.conv2(h))
        return h

class SimpleMNistUNet(nn.Module):
    def __init__(self, in_channels=1, out_channels=1, time_emb_dim=64):
        super().__init__()

        self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbeddings(time_emb_dim),
            nn.Linear(time_emb_dim, time_emb_dim),
            nn.SiLU(),
            nn.Linear(time_emb_dim, time_emb_dim)
        )
```

```
43
44          self.enc1 = SimpleBlock(in_channels, 32, time_emb_dim)
45          self.enc2 = SimpleBlock(32, 64, time_emb_dim)
46          self.enc3 = SimpleBlock(64, 128, time_emb_dim)
47
48          self.bottleneck = SimpleBlock(128, 256, time_emb_dim)
49
50          self.up1 = nn.ConvTranspose2d(256, 128, 4, stride=2,
               padding=1)
51          self.dec1 = SimpleBlock(256, 128, time_emb_dim)
52
53          self.up2 = nn.ConvTranspose2d(128, 64, 4, stride=2, padding
               =1)
54          self.dec2 = SimpleBlock(128, 64, time_emb_dim)
55
56          self.up3 = nn.ConvTranspose2d(64, 32, 4, stride=2, padding
               =1)
57          self.dec3 = SimpleBlock(64, 32, time_emb_dim)
58
59          self.final_conv = nn.Conv2d(32, out_channels, 3, padding=1)
60
61      def forward(self, x, t):
62          t_emb = self.time_mlp(t)
63
64          x1 = self.enc1(x, t_emb)
65          x2 = self.enc2(F.max_pool2d(x1, 2), t_emb)
66          x3 = self.enc3(F.max_pool2d(x2, 2), t_emb)
67
68          x_bottleneck = self.bottleneck(F.max_pool2d(x3, 2), t_emb)
69
70          x = self.up1(x_bottleneck)
71
72          if x.shape[2:] != x3.shape[2:]:
73              x = F.interpolate(x, size=x3.shape[2:], mode='nearest')
74
75          x = torch.cat([x, x3], dim=1)
76          x = self.dec1(x, t_emb)
77
78          x = self.up2(x)
79          if x.shape[2:] != x2.shape[2:]:
80              x = F.interpolate(x, size=x2.shape[2:], mode='nearest')
81
82          x = torch.cat([x, x2], dim=1)
83          x = self.dec2(x, t_emb)
84
85          x = self.up3(x)
86          if x.shape[2:] != x1.shape[2:]:
87              x = F.interpolate(x, size=x1.shape[2:], mode='nearest')
88
89          x = torch.cat([x, x1], dim=1)
90          x = self.dec3(x, t_emb)
```

```
91
92          return self.final_conv(x)
93
94  class DDPM:
95      def __init__(self, model, timesteps=1000, beta_start=1e-4,
            beta_end=0.02, device="cuda"):
96          self.model = model
97          self.timesteps = timesteps
98          self.device = device
99
100         self.betas = torch.linspace(beta_start, beta_end, timesteps
                , device=device)
101         self.alphas = 1. - self.betas
102         self.alpha_bars = torch.cumprod(self.alphas, dim=0)
103
104         print(f"Beta␣schedule:␣{beta_start}␣->␣{beta_end}")
105         print(f"Alpha␣bars:␣{self.alpha_bars[0]:.3f}␣->␣{self.
                alpha_bars[-1]:.3f}")
106         print(f"Nombre␣de␣timesteps:␣{timesteps}")
107
108     def forward_process(self, x0, t, noise=None):
109         if noise is None:
110             noise = torch.randn_like(x0)
111
112         t = torch.clamp(t, 0, self.timesteps - 1)
113
114         alpha_bar_t = self.alpha_bars[t].view(-1, 1, 1, 1)
115         x_t = torch.sqrt(alpha_bar_t) * x0 + torch.sqrt(1 -
                alpha_bar_t) * noise
116
117         return x_t
118
119     def reverse_process(self, x, t):
120         t = torch.clamp(t, 0, self.timesteps - 1)
121
122         pred_noise = self.model(x, t)
123
124         alpha_t = self.alphas[t].view(-1, 1, 1, 1)
125         alpha_bar_t = self.alpha_bars[t].view(-1, 1, 1, 1)
126         beta_t = self.betas[t].view(-1, 1, 1, 1)
127
128         if t[0] > 0:
129             mean = (x - beta_t * pred_noise / torch.sqrt(1 -
                    alpha_bar_t)) / torch.sqrt(alpha_t)
130
131             noise = torch.randn_like(x)
132             std = torch.sqrt(beta_t)
133             x_prev = mean + std * noise
134         else:
135             x_prev = (x - torch.sqrt(1 - alpha_bar_t) * pred_noise)
                    / torch.sqrt(alpha_bar_t)
```

```
136
137         return x_prev
138
139     def train_step(self, x0, optimizer):
140         self.model.train()
141
142         t = torch.randint(0, self.timesteps, (x0.shape[0],), device
                =self.device)
143
144         noise = torch.randn_like(x0)
145
146         x_t = self.forward_process(x0, t, noise)
147
148         pred_noise = self.model(x_t, t)
149
150         loss = F.mse_loss(pred_noise, noise)
151
152         loss.backward()
153         optimizer.step()
154
155         return loss.item()
156
157     def sample(self, num_samples=16, image_size=(28, 28)):
158         self.model.eval()
159
160         with torch.no_grad():
161             x = torch.randn(num_samples, 1, image_size[0],
                    image_size[1], device=self.device)
162
163             for i in tqdm(reversed(range(self.timesteps)), desc="Gé
                    nération"):
164                 t = torch.full((num_samples,), i, device=self.
                        device, dtype=torch.long)
165                 x = self.reverse_process(x, t)
166
167             x = torch.clamp(x, -1., 1.)
168             x = (x + 1.) / 2.
169
170         return x
```