Auto-encodeur débruitage

Réalisé par : Mathis Aulagnier
Cours : Vision Artificielle (8INF804)

1 Introduction

1.1 Objectif

L'objectif de ce projet est de concevoir et de travailler avec un autoencodeur afin de pouvoir débruiter des images. De plus, au cours de ce travail, cet autoencodeur a été utilisé pour extraire des caractéristiques pertinentes, permettant de réaliser une classification grâce à des modèles de machine learning.

1.2 Motivation

Ce projet a été motivé par l'amélioration des performances de modèles de classification lorsqu'ils sont confrontés à des données bruitées. De plus, un tel autoencodeur peut être utilisé pour augmenter la taille du dataset en générant des variantes bruitées d'images, qui sont ensuite restaurées via le modèle.

1.3 Jeu de données

Le dataset choisi est le célèbre MNIST, qui contient des images de chiffres manuscrits (0 à 9) de taille 28×28. Nous avons ajouté du bruit gaussien pour simuler un contexte dégradé. Les images bruyées ont servi d'entrée pour l'autoencodeur, tandis que les images originales ont été utilisées comme cibles.

1.4 Importations des bibliothèques

Nous avons utilisé les bibliothèques suivantes :

Listing 1: librairies

```
import torch
import torch.nn as nn
import torch.optim as optim

import torchvision
import torchvision.transforms as transforms

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score

import matplotlib.pyplot as plt

import numpy as np
import pandas as pd

import os
os.makedirs('Models', exist_ok=True)
```

Un dossier nommé `Models` a été créé pour sauvegarder les modèles entraînés.

2 Définition de l'architecture de l'autoencodeur

Depuis le début de ce rapport, nous avons parlé d'autoencodeur, mais sans expliquer ce concept en détail. Un autoencodeur est un modèle constitué de deux parties principales : un **encodeur** et un **décodeur**. Son objectif principal est d'apprendre une fonction identité, c'est-à-dire reconstruire ses données d'entrée après les avoir compressées dans un espace intermédiaire appelé espace latent ou *bottleneck*.

- **L'encodeur** réduit la dimensionnalité des données d'entrée en extrayant leurs caractéristiques essentielles et en les projetant dans l'espace latent.
- **Le décodeur**, à partir de cet espace latent, tente de reconstruire les données d'origine.

Voici un schéma représentant un autoencodeur (CNN) tiré du cours :

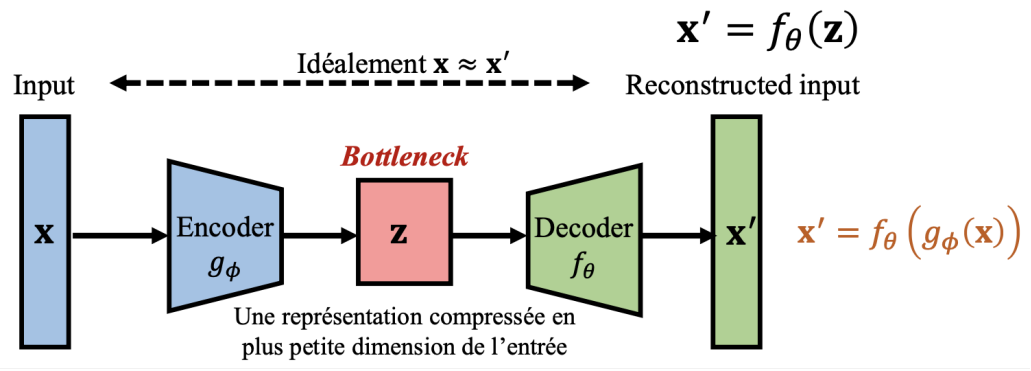

Figure 1: Schéma d'un autoencodeur

2.1 L'autoencodeur utilisé dans ce projet

Nous avons conçu un autoencodeur basé sur des réseaux neuronaux convolutionnels (CNN). L'encodeur réduit les images à un vecteur de caractéristiques dans un espace latent de dimension 128 par défaut. Le décodeur reconstruit ensuite les images à partir de ce vecteur latent.

Dans les sections suivantes, nous détaillons séparément l'architecture de l'encodeur et du décodeur.

2.2 Encodeur (CNNEncoder)

L'encodeur est composé de trois couches convolutives, chacune suivie par une activation `ReLU` et une normalisation par lot (`BatchNorm`). Enfin, la sortie des couches convolutives est aplatie et projetée dans l'espace latent. Par défaut, la taille de cet espace latent est de 128.

Vous trouverez ci-dessous l'implémentation du code en Python.

Listing 2: Implémentation de l'encodeur CNN

```
class CNNEncoder(nn.Module):
    def __init__(self, latent_dim=128, in_channel=1):
        super(CNNEncoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(in_channel, 64, kernel_size=3, padding=1, stride=2),
            nn.ReLU(),
            nn.BatchNorm2d(64),

            nn.Conv2d(64, 128, kernel_size=3, padding=1, stride=2),
            nn.ReLU(),
            nn.BatchNorm2d(128),

            nn.Conv2d(128, 256, kernel_size=3, padding=1, stride=2),
            nn.ReLU(),
            nn.BatchNorm2d(256),

            nn.Flatten(),
            nn.Linear(256 * 4 * 4, latent_dim)
        )
```

Cet encodeur est capable d'extraire des représentations compactes et robustes des images, ce qui est crucial pour la reconstruction et la classification.

2.3 Décodeur (CNNDecoder)

Le décodeur est conçu pour reconstruire les données d'entrée à partir des représentations compactes présentes dans l'espace latent. Il commence par projeter le vecteur latent dans une forme tridimensionnelle (tensor 4D), correspondant à une taille intermédiaire dans les couches convolutives. Ensuite, il applique des couches transposées de convolution (`ConvTranspose2d`) pour augmenter progressivement la résolution des données, jusqu'à atteindre la taille d'entrée initiale.

Chaque couche transposée est suivie d'une activation `ReLU` et d'une normalisation par lot (`BatchNorm`), à l'exception de la dernière couche, qui utilise une activation `Tanh` pour limiter les valeurs des pixels entre -1 et 1.

Vous trouverez ci-dessous l'implémentation du code en Python.

Listing 3: Implémentation du décodeur CNN

```
class CNNDecoder(nn.Module):
    def __init__(self, latent_dim=128, in_channel=1):
        super(CNNDecoder, self).__init__()
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 256 * 4 * 4),
            nn.Unflatten(1, (256, 4, 4)),
            nn.ConvTranspose2d(256, 128, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(128),
            nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(64),
            nn.ConvTranspose2d(64, in_channel, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.Tanh()
        )
```

Ce décodeur utilise des techniques standards pour reconstruire des images tout en préservant une qualité visuelle optimale. Grâce à l'utilisation de `ConvTranspose2d`, la reconstruction des données s'effectue progressivement, en ajoutant des détails à chaque étape jusqu'à retrouver la résolution d'origine.

2.4 Auto-encodeur

Nous avons par la suite associé nos deux modèles dans une classe appelée `AutoEncoder`. Cette classe possède des fonctions permettant d'extraire l'encodeur ou le décodeur, une fois ces derniers entraînés.

Voici l'implémentation de cette classe en Python.

Listing 4: Classe AutoEncoder

```
class AutoEncoder(nn.Module):
    def __init__(self, latent_dim=128, in_channel=1):
        super(AutoEncoder, self).__init__()
        self.encoder = CNNEncoder(latent_dim, in_channel)
        self.decoder = CNNDecoder(latent_dim, in_channel)

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

    def get_encoder(self):
        return self.encoder

    def get_decoder(self):
        return self.decoder
```

3 Auto-encodeur débruiteur (Denoising Autoencoder)

La particularité d'un auto-encodeur débruiteur est qu'il ne cherche pas à apprendre une fonction identité. L'idée est de partir d'une image initiale A, d'ajouter un bruit à celle-ci pour obtenir A_{noise}, et de donner cette image bruitée en entrée à l'auto-encodeur. L'encodeur traite alors cette entrée via $f(A_{\text{noise}})$ pour produire une représentation dans l'espace latent, et le décodeur reconstitue une image débruitée $A' = g(f(A_{\text{noise}}))$. La fonction de perte compare ensuite l'image débruitée A' avec l'image d'origine A, et non avec l'image bruitée A_{noise}. Ainsi, le décodeur est entraîné à restaurer les images dégradées.

Dans cette optique, nous avons utilisé une fonction d'entraînement modulaire qui prend en compte cette spécificité.

Listing 5: Fonction d'entraînement pour auto-encodeur débruiteur

```
def train_cnn_model(model, train_loader, val_loader, criterion, optimizer,
                    num_epochs=10, noisy=0.0, verbose=True, save_path=None):
```

Pour présenter les arguments, on retrouve :

- model : Le modèle à entraîner.
- train_loader : Le DataLoader pour l'ensemble d'entraînement.
- val_loader : Le DataLoader pour l'ensemble de validation.
- criterion : La fonction de perte (loss).
- optimizer : L'optimiseur pour la descente de gradient.
- num_epochs : Nombre d'époques pour l'entraînement.
- noisy : Intensité du bruit ajouté aux images (par défaut 0.0).
- verbose : Affiche des informations durant l'entraînement si True.
- save_path : Chemin pour enregistrer le modèle (None par défaut).

Lorsque l'argument `noisy` est défini à zéro, l'auto-encodeur apprend une fonction identité classique. En revanche, si `noisy` est différent de zéro, le modèle devient un auto-encodeur débruiteur.

La fonction est également modulaire, car elle permet de sauvegarder le modèle via l'argument `save_path` si ce dernier n'est pas égale à **None**, et d'afficher des informations durant l'entraînement grâce à l'option `verbose`.

Avant de passer à l'entraînement de nos modèles, vous pourrez trouver en annexe une visualisation réalisée à partir de Tensorboard.

4 Entraînement de nos modèles

L'entraînement des modèles a été effectué sur 30 époques avec un taux d'apprentissage fixé à `0.001`, `num_epochs = 30` et `learning_rate = 0.001`.

La fonction de perte n'a pas été choisie au hasard. Nous avons hésité entre trois fonctions de perte différentes :

- `nn.L1Loss()`

$$\text{L1Loss} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

- `nn.MSELoss()`

$$\text{MSELoss} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- `nn.SmoothL1Loss()`

$$\text{SmoothL1Loss} = \frac{1}{n}\sum_{i=1}^{n} z_i$$

où

$$z_i = \begin{cases} 0.5(y_i - \hat{y}_i)^2 & \text{si } |y_i - \hat{y}_i| < 1 \\ |y_i - \hat{y}_i| - 0.5 & \text{sinon} \end{cases}$$

Voici une illustration des résultats obtenus avec ces différentes fonctions de perte (Figure 2).

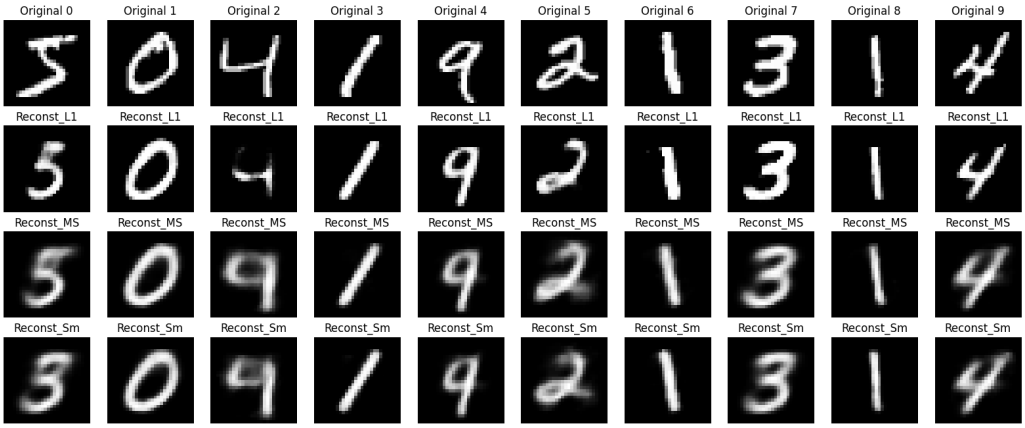

Figure 2: Comparaison des fonctions de perte.

Dans cette image, la première ligne montre les images originales. La deuxième ligne présente les images reconstruites par un auto-encodeur utilisant la fonction de perte `L1Loss()`. La troisième ligne correspond à `MSELoss()`. Enfin, la quatrième ligne illustre les résultats obtenus avec `SmoothL1Loss()`.

À l'œil nu, les résultats obtenus avec la fonction `L1Loss()` semblaient de meilleure qualité. Par conséquent, cette fonction a été retenue pour nos entraînements.

Pour optimiser nos modèles, nous avons utilisé l'optimiseur classique `Adam` avec le taux d'apprentissage mentionné.

Dans le but d'étudier l'effet de la taille de l'espace latent, nous avons entraîné plusieurs modèles avec différentes dimensions latentes.

Nous avons surveillé les risques d'overfitting des différents modèles à l'aide des courbes d'apprentissage. Comme illustré dans la Figure 3, les modèles ne présentent pas de surapprentissage (*overfitting*).

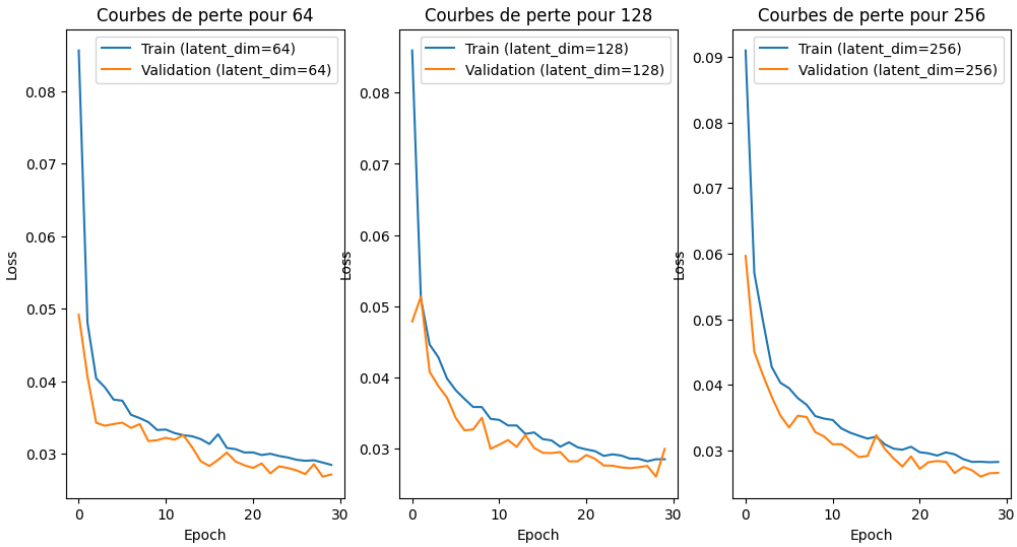

Figure 3: Courbes d'apprentissage pour les différents modèles.

5 Résultats

5.1 Résultats visuels

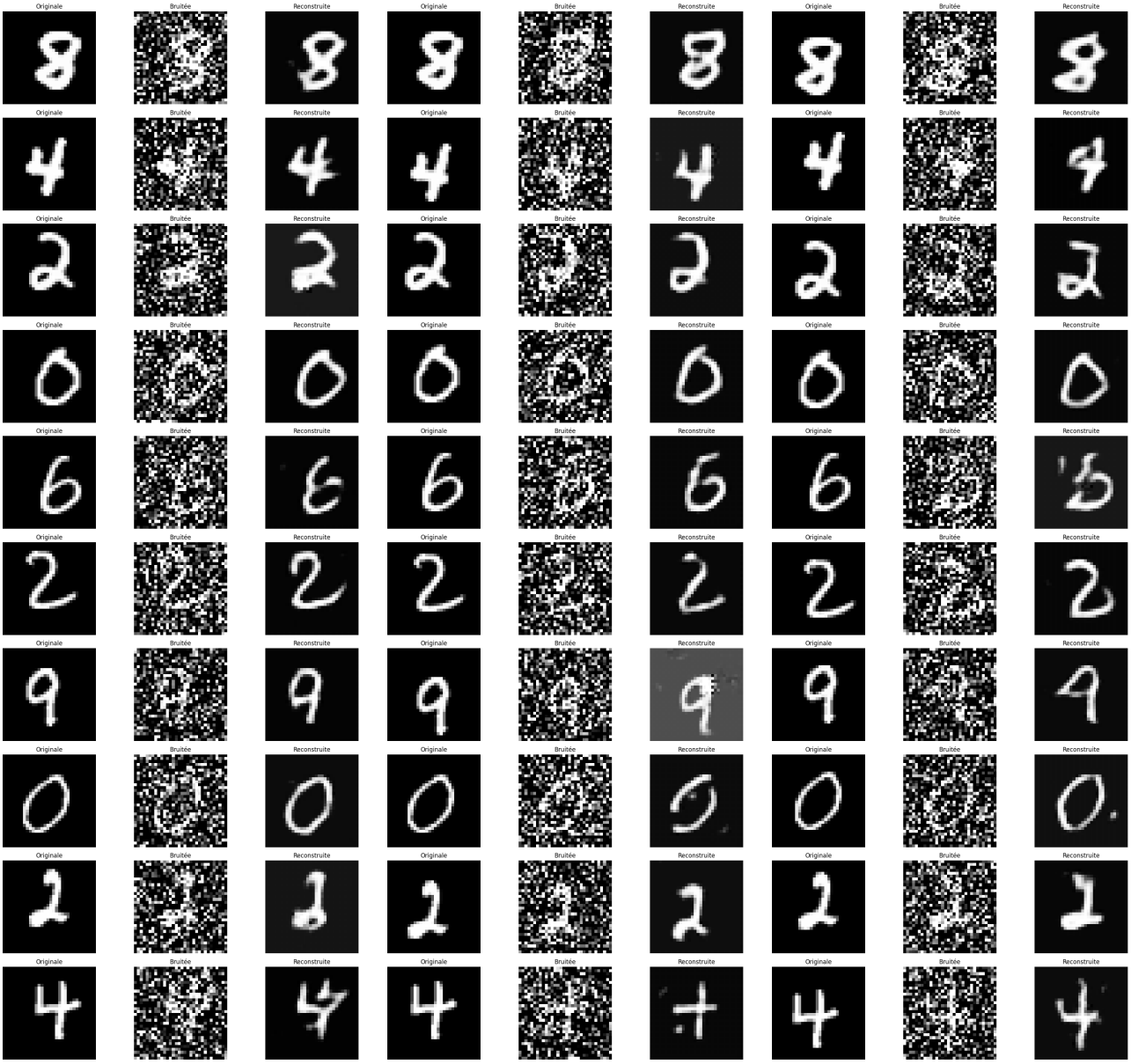

Figure 4:
Modèle 1 (Espace latent : 64)

Figure 5:
Modèle 2 (Espace latent : 128)

Figure 6:
Modèle 3 (Espace latent : 256)

Figure 7: Comparaison des résultats pour différents espaces latents. Chaque colonne correspond à un modèle avec une dimension d'espace latent différente.

A partir de ces résultats, on. observe que plus l'espace lattent est grand, plus des caractéristiques sont extraite. Et donc sans surprises les résultats sont meilleurs averc un espaces lattant important.

5.2 Résultats Basés sur des Métriques

Pour évaluer les performances de notre système global (autoencodeur + CNN), nous avons mesuré l'accuracy sur différentes versions des données de test :

- **Données originales (non bruitées)** : En utilisant le CNN seul, nous obtenons une accuracy de **98.73%**, un résultat qui s'approche des performances humaines sur le jeu MNIST.

- **Données bruitées (sans débruitage)** : Lorsque le CNN est appliqué directement aux données bruitées, l'accuracy chute drastiquement (**19.96%**), indiquant que le bruit affecte significativement la capacité du modèle à classer correctement les images.
- **Données débruitées (via l'autoencodeur)** : Après avoir passé les données bruitées à travers l'autoencodeur, l'accuracy remonte à **87.88%**. Cela démontre la capacité de l'autoencodeur à restaurer des images proches des données originales, permettant au CNN de mieux effectuer ses prédictions.
- **Données originales (passées par l'autoencodeur)** : Les données originales, même après passage par l'autoencodeur, atteignent une accuracy de **96.49%**, suggérant une légère perte d'information introduite par l'autoencodeur.

Type de données	**Accuracy (%)**
Données originales	98.73
Données bruitées (sans AE)	19.96
Données débruitées (avec AE)	87.88
Données originales (avec AE)	96.49

Table 1: Performance du système sur différentes versions des données de test.

Analyse des métriques : Ces résultats mettent en évidence l'impact de l'autoencodeur (AE) sur la qualité des données d'entrée pour le CNN. Les données bruitées, lorsqu'elles ne sont pas traitées par l'AE, mènent à une accuracy très faible (**19.96%**), confirmant que le bruit compromet sévèrement les performances du modèle. En revanche, l'utilisation de l'autoencodeur pour débruiter ces données améliore nettement les performances, avec une accuracy de **87.88%**.

Cependant, l'accuracy obtenue avec les données débruitées reste inférieure à celle des données originales (**98.73%**), ce qui suggère que l'autoencodeur n'élimine pas complètement le bruit ou altère légèrement l'information utile. En outre, les données originales traitées par l'autoencodeur atteignent une accuracy de **96.49%**, montrant que l'AE peut introduire une petite perte d'information, même en l'absence de bruit.

Ces observations confirment le rôle essentiel de l'autoencodeur dans la reconstruction des données bruitées, tout en soulignant ses limites face à un bruit potentiellement trop important ou complexe.

6 Résilience du Modèle Face à Différents Types de Bruit

Afin de tester la résilience de notre modèle face à différents types de bruit appliqué sur les images du jeu de test, nous avons ajouté plusieurs types de bruit à nos données : *Poisson*, *Speckle* et *Périodique*. Les résultats obtenus nous permettent de comparer la performance du modèle sur des données bruitées par rapport à des données non bruitées.

6.1 Test avec du bruit de type Poisson

Le bruit de type *Poisson* a une distribution qui varie en fonction de l'intensité des pixels. Lorsqu'appliqué sur nos images de test, ce bruit a un impact sur la précision du modèle. En effet, le modèle obtient une *accuracy* de **74.6%**, inférieure à celle observée sur des données non bruitées. Les performances varient selon les classes, certaines classes telles que la classe 1 ont un excellent rappel mais souffrent d'une précision relativement faible (0.66).

Résultats :

- *Accuracy globale* : 74.6%
- **Macro moyenne** : 0.79 (précision), 0.74 (rappel), 0.74 (F1-score)

6.2 Test avec du bruit de type Speckle

Le bruit *Speckle* est un bruit multiplicatif, ce qui signifie qu'il affecte l'image en multipliant les pixels par des valeurs aléatoires. Cependant, en dépit de cet effet, notre modèle reste relativement robuste face à ce bruit, obtenant une *accuracy* de **86.5%**. Les résultats montrent une performance largement améliorée par rapport au bruit de type *Poisson*, avec une précision et un rappel plus équilibrés sur la plupart des classes. Le modèle parvient à maintenir une bonne performance générale.

Résultats :

- *Accuracy globale* : 86.5%
- **Macro moyenne** : 0.88 (précision), 0.86 (rappel), 0.86 (F1-score)

6.3 Test avec du bruit de type Périodique

Le bruit de type *Périodique*, qui introduit des oscillations régulières dans les données, semble avoir l'impact le plus faible sur la performance du modèle. Avec une *accuracy* impressionnante de **96.75%**, le modèle conserve une excellente précision et un excellent rappel, notamment pour les classes comme la classe 0, qui obtient une précision de 0.98 et un rappel de 0.99. Ce résultat suggère que le modèle est particulièrement robuste aux effets périodiques, probablement parce que le bruit périodique affecte moins la structure globale des images comparé aux autres types de bruit.

Résultats :

- *Accuracy globale* : 96.75%
- **Macro moyenne** : 0.97 (précision), 0.97 (rappel), 0.97 (F1-score)

6.4 Analyse Comparative

Les tests réalisés montrent que notre modèle est relativement résilient face à différents types de bruit. En particulier :

- Le bruit de type *Poisson* a un effet plus significatif sur les performances du modèle, en particulier pour les classes ayant des formes complexes ou moins distinctes. Cependant, même dans ce cas, certaines classes restent bien prédites.
- Le bruit de type *Speckle* a un impact plus modéré sur la précision globale, avec une performance notablement meilleure par rapport au bruit *Poisson*.
- Le bruit *Périodique* semble avoir peu d'impact sur la performance du modèle, ce qui montre que le modèle est capable de gérer ce type de bruit de manière efficace.

Dans l'ensemble, bien que le modèle montre une légère dégradation des performances sous l'impact du bruit, il reste efficace et montre une bonne résilience face à des bruits de types variés, notamment dans des situations plus complexes comme le bruit *Speckle* et *Poisson*. Ce comportement montre que le modèle est robuste, bien qu'il puisse bénéficier d'améliorations supplémentaires.

7 Exploration des caractéristiques extraites par l'encodeur

Dans cette section, nous avons cherché à vérifier si l'encodeur de notre autoencodeur était capable d'extraire des caractéristiques importantes et significatives à partir des images. Pour ce faire, nous avons utilisé l'encodeur entraîné sur notre jeu de données d'entraînement pour générer les représentations latentes de nos ensembles de données d'entraînement, de validation et de test.

Ces représentations latentes ont ensuite été stockées dans des DataFrames, auxquels nous avons ajouté les labels des images correspondantes. Nous avons sélectionné l'encodeur de notre modèle ayant un espace latent de 64 dimensions, en supposant que si cet encodeur extrait des caractéristiques pertinentes, celles-ci seront également présentes dans les modèles avec des espaces latents de dimensions supérieures.

Afin de tester la qualité des caractéristiques extraites, nous avons entraîné un classifieur SVM (Support Vector Machine) sur les données latentes. Nous avons utilisé un *kernel trick* pour projeter les données dans un espace de dimension infinie, en nous basant sur une approximation de la fonction exponentielle :

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

De plus, pour limiter le risque de surapprentissage, nous avons réglé le paramètre $C = 1$. Ce paramètre contrôle le compromis entre un faible taux d'erreurs d'entraînement et une meilleure généralisation du modèle. Une valeur plus faible de C impose une régularisation plus forte.

Résultats obtenus

Nous avons évalué les performances du SVM en appliquant un rapport de classification sur les ensembles de validation et de test. Voici les résultats obtenus :

Accuracy sur l'ensemble de validation : 93.60%

7.1 Résultats détaillées sur l'ensemble de test

	precision	recall	f1-score	support
0	0.95	0.97	0.96	980
1	0.96	0.99	0.97	1135
2	0.94	0.94	0.94	1032
3	0.94	0.93	0.93	1010
4	0.92	0.93	0.93	982
5	0.94	0.93	0.93	892
6	0.93	0.96	0.95	958
7	0.95	0.93	0.94	1028
8	0.93	0.92	0.92	974
9	0.93	0.91	0.92	1009
accuracy			0.94	10000
macro avg	0.94	0.94	0.94	10000
weighted avg	0.94	0.94	0.94	10000

Accuracy sur l'ensemble de test : 93.96%

Les résultats montrent que les représentations latentes extraites par l'encodeur permettent au SVM de bien classer les données, avec une précision globale de plus de 93% sur les ensembles de validation et de test. Cela confirme que l'encodeur extrait effectivement des caractéristiques pertinentes des données d'entrée, démontrant ainsi l'efficacité de notre autoencodeur.

8 Conclusion

En conclusion, les résultats obtenus démontrent leur efficacité à débruiter des données et à extraire des caractéristiques significatives. À l'avenir, l'intégration de mécanismes tels que l'attention pour découvrir les zones essentielles à la représentation des chiffres pourrait améliorer les performances. De plus, étendre cette approche à des jeux de données plus complexes, comme des images médicales, permettrait de mieux évaluer la généralisation des modèles.

A Annexe 1 : Visualisation de notre autoencodeur via TensorBoard

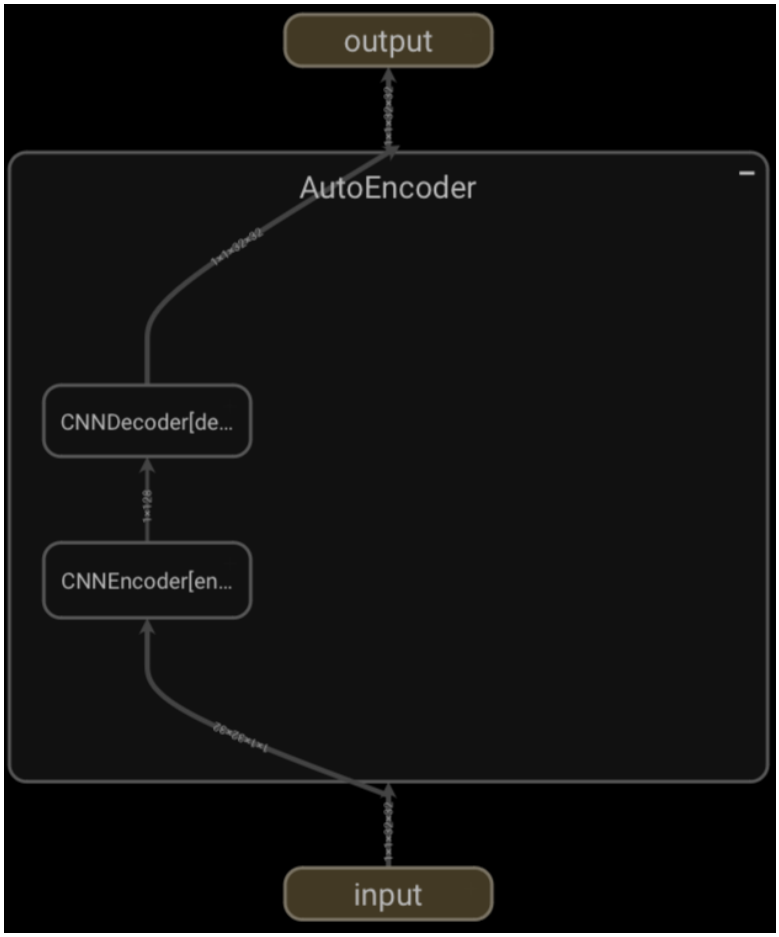

Figure 8: Visualisation de notre autoencodeur via TensorBoard.

B Annexe 2 : Visualisation de notre encodeur et décodeur

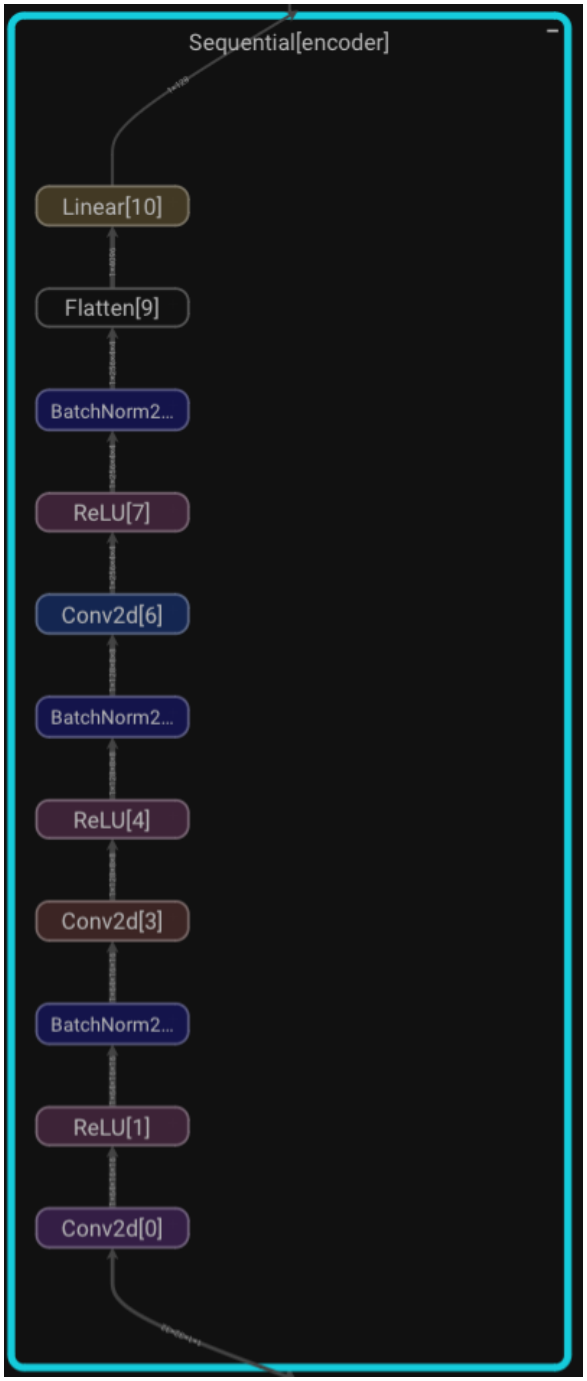

Figure 9: Visualisation de notre encodeur.

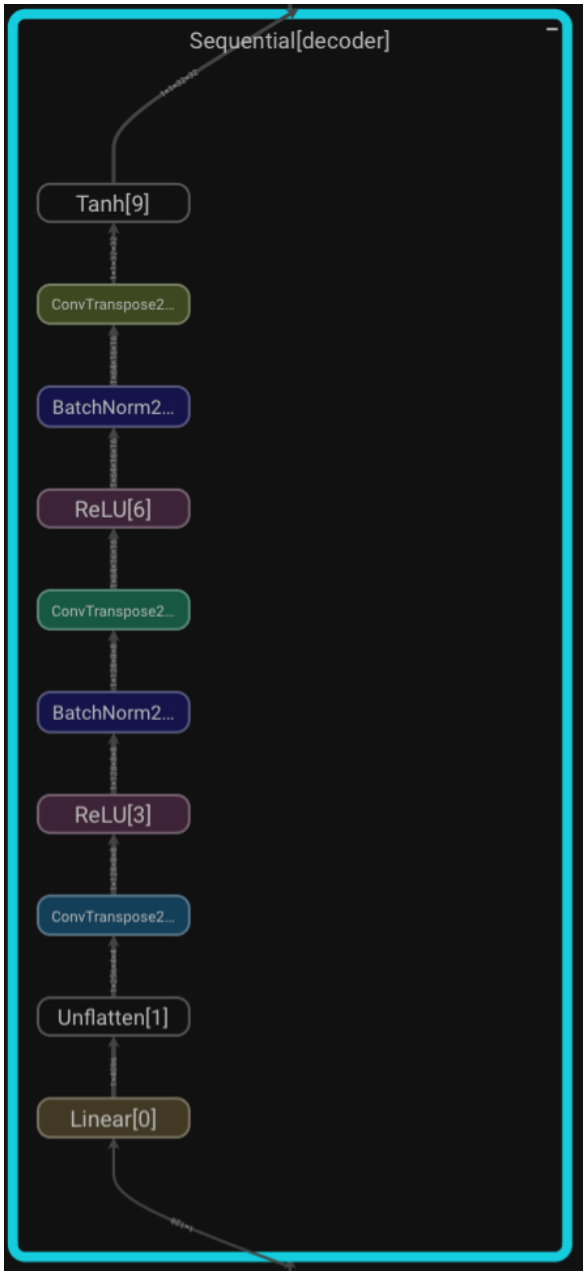

Figure 10: Visualisation de notre décodeur.