TP3 : AutoEncoder
Apprentissage non supervisé

SINADINOVIC Marko

Semestre 6 : 2024-2025

Contents

1 Motivation

Dans le cadre du cours de Recherche Opérationnelle (Machine Learning), j'ai été amené à réaliser le TP numéro 3, qui portait sur l'apprentissage non supervisé. Mon objectif était d'implémenter un réseau de neurones auto-encodeur. Passionné par cet univers mêlant mathématiques et informatique, j'ai pris l'initiative de documenter mon travail afin de mieux comprendre les concepts abordés et de faciliter ma préparation pour l'examen.

2 Introduction

Le but de ce TP est d'implémenter un réseau de neurones auto-encodeur. Les auto-encodeurs sont des algorithmes conçus pour réaliser des tâches de compression et de décompression de données. Ces modèles reposent sur des réseaux de neurones qui apprennent automatiquement à accomplir cette tâche, sans nécessiter de labélisation préalable des données, ce qui en fait une méthode d'apprentissage non supervisé.

Le principe de l'auto-encodeur repose sur la représentation des données initiales par des vecteurs $\mathbf{X}_i \in \mathbb{R}^n$ que l'on souhaite compresser en des vecteurs réduits $\mathbf{Z}_i \in \mathbb{R}^p$, avec $p < n$. À cet effet, on construit un réseau de neurones divisé en deux parties :

- **Un encodeur** $E_r : \mathbb{R}^n \to \mathbb{R}^p$, chargé de compresser les données initiales.
- **Un décodeur** $D_r : \mathbb{R}^p \to \mathbb{R}^n$, qui reconstruit les données compressées vers leur forme originale.

L'objectif est d'entraîner ce réseau de neurones de manière à ce que la composition des deux parties soit proche de l'identité:

$$D_r \circ E_r \approx \mathrm{Id}.$$

Dans ce TP, cette méthode sera appliquée à la reconnaissance de chiffres calligraphiés, en utilisant un ensemble de données adapté. Cette exploration permettra d'examiner les performances de compression et de reconstruction des auto-encodeurs ainsi que leur application au clustering dans un espace latent réduit.

3 Exercice 1

L'encodeur prend en entrée n valeurs (avec $n = 28 \times 28 = 784$, correspondant au nombre de pixels d'une image) et les réduit dans un vecteur de $\mathbb{R}^p$. Plus p est petit, plus la compression sera importante. Par exemple, nous pouvons choisir $p = 2$ pour une forte compression ou $p = 16$ pour une compression plus modérée.

Le décodeur, quant à lui, prend en entrée le vecteur compressé dans $\mathbb{R}^p$ et reconstruit un vecteur dans $\mathbb{R}^n$, dans le but de retrouver une approximation de l'image d'origine. La qualité de reconstruction dépend à la fois de la valeur de p et de l'architecture interne des couches du réseau.

Nous allons comparer les performances du réseau en considérant deux configurations pour l'espace latent: $p = 2$ et $p = 16$. Ces comparaisons permettront de mettre en évidence les compromis entre le degré de compression et la fidélité de la reconstruction.

3.1 Auto-encoder d'espace latent de dimension dimension p

Un espace latent est une représentation compressée et abstraite des données initiales. C'est la sortie de l'encodeur, qui transforme des données de haute dimension en un vecteur de dimension réduite, noté p. Cet espace latent préserve les caractéristiques principales des données tout en éliminant les détails non essentiels pour leur reconstruction.Cet espace latent a plusieurs propriétés essentielles. Dans un premier temps, la **compression de données**, en effet, il "réduit" la taille des données tout en conservant leurs principaux attributs. Ensuite nous avons **l'extraction de motifs** qui permet un output des données (motifs) comme elles étaient dans un espace augmenté.

De ces faits, le choix de la dimension de l'espace latent (p) a un impact significatif sur les performances du réseau :

- **Petite dimension ($p = 2$) :** Une faible dimension est synonyme de forte compression. Le défaut d'une petite dimension est la perte d'information, qui est importante. Cela rend difficile la différenciation de données qui se ressemble, par exemple ici le chiffre 5 avec le chiffre 8 ou le 4 avec 9.

- **Grande dimension ($p = 16$) :** Une dimension plus élevée permet de conserver davantage d'informations. La reconstruction des données est donc plus fidèle. Or, une grande dimension rend le modèle plus complexe à entraîner (temps de calculs et mémoire plus important) et la compression des données est moins élevée mais l'objectif principal des auto-encodeurs est de compresser les données. Donc une très grande dimension de l'espace de latence diminue voir annule l'intérêt de l'utilisation d'un auto-encodeur.

3.1.1 Auto-encodeur avec une couche intermédiaire

Nous disposons d'un jeu de données de 60000 items de taille 28x28, soit 784 pixels. Construisons notre réseau de neurones constitué de deux parties, la partie encodeur, et la partie décodeur, expliqué dans l'introduction. En remplaçant p par 2, nous obtenons le code suivant :

```
import torch.nn as nn

class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()
        # Definition de l'encodeur (R^n -> R^p avec une couche intermediaire)
        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, 128),  # Couche intermediaire
            nn.Tanh(),                # Fonction d'activation Tanh
            nn.Linear(128, 2)         # Espace latent de dimension 2
        )
        # Definition du decodeur (R^p -> R^n avec une couche intermediaire)
        self.decoder = nn.Sequential(
            nn.Linear(2, 128),        # Couche intermediaire
            nn.Tanh(),                # Fonction d'activation Tanh
            nn.Linear(128, 28 * 28), # Reconstruction vers R^n
            nn.Sigmoid()              # Fonction d'activation Sigmoid
        )

    def forward(self, x):
        encoded = self.encoder(x)   # Encodage
        decoded = self.decoder(encoded)  # Decodage
        return encoded, decoded
```

Listing 1: Auto-encodeur avec une couche intermédiaire et espace latent de dimension deux

Ce code comporte qu'une seule couche intermédiaire, qui est insufisant pour bien traiter notre problème. Pour l'instant nous nous restreignons à cette structure de code. Par le même procédé, pour p = 16 :

```python
import torch.nn as nn

class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()
        # Definition de l'encodeur (R^n -> R^p avec une couche intermediaire)
        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, 128),  # Couche intermediaire
            nn.Tanh(),                # Fonction d'activation Tanh
            nn.Linear(128, 16)         # Espace latent de dimension 16
        )
        # Definition du decodeur (R^p -> R^n avec une couche intermediaire)
        self.decoder = nn.Sequential(
            nn.Linear(16, 128),        # Couche intermediaire
            nn.Tanh(),                # Fonction d'activation Tanh
            nn.Linear(128, 28 * 28), # Reconstruction vers R^n
            nn.Sigmoid()              # Fonction d'activation Sigmoid
        )

    def forward(self, x):
        encoded = self.encoder(x)   # Encodage
        decoded = self.decoder(encoded)  # Decodage
        return encoded, decoded
```

Listing 2: Auto-encodeur avec une couche intermédiaire et espace latent de dimension seize

Maintenant que nous avons en notre possession 2 auto-encodeurs de dimensions latentes $p = 2$ et $p = 16$, nous pouvons les entraîner. Pour ce faire, nous allons adopté une approche d'optimisation basée sur l'utilisation de la fonction coût MSE qui mesure l'erreur quadratique moyenne entre l'entrée initiale et la sortie reconstruite. Mathématiquement, elle est représenté de la manière suivante :

Définition 1. Erreur Quadratique Moyenne

$$MSE = \mathbb{E}\left[(Y - \hat{Y})^2\right]$$

où:

- Y *est la variable aléatoire représentant les valeurs réelles.*
- $\hat{Y}$ *est la variable aléatoire représentant les prédictions faites par le modèle.*
- $\mathbb{E}$ *désigne l'espérance (ou la moyenne).*

L'erreur quadratique moyenne (ou Mean Squared Error, MSE) mesure la différence moyenne entre les valeurs réelles Y et les valeurs prédites par un modèle $\hat{Y}$:

$$\text{MSE} = \mathbb{E}\left[(Y - \hat{Y})^2\right]$$

où:

- Y est la variable aléatoire représentant les valeurs réelles.
- $\hat{Y}$ est la variable aléatoire représentant les prédictions faites par le modèle.

- $\mathbb{E}$ désigne l'espérance (ou la moyenne).

L'erreur quadratique moyenne peut être décomposée en trois termes : le biais, la variance, et l'erreur irréductible. Cette décomposition, appelée théorème du biais-variance, s'exprime ainsi:

$$\text{MSE} = \underbrace{\left(\mathbb{E}[\hat{Y}] - Y\right)^2}_{\text{Biais}^2} + \underbrace{\mathbb{E}\left[(\hat{Y} - \mathbb{E}[\hat{Y}])^2\right]}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Erreur irréductible}}$$

- **Le biais** représente l'erreur systématique introduite par le modèle. Il correspond à la différence entre la vraie valeur Y et la moyenne des prédictions $\mathbb{E}[\hat{Y}]$.
- **La variance** décrit la variabilité des prédictions $\hat{Y}$ autour de leur moyenne $\mathbb{E}[\hat{Y}]$. Une variance élevée indique que le modèle est très sensible aux variations des données d'entraînement.
- **L'erreur irréductible** σ^2 est due au bruit inhérent dans les données, qui ne peut être réduit par aucun modèle.

En outre, elle évalue la qualité de reconstruction du réseau et guide l'apprentissage en minimisant l'erreur.

Pour optimiser les paramètres du réseau, nous avons utilisé l'algorithme **Adam** qui ajuste les taux d'apprentissage pour chaque paramètre, ce qui accélère la convergence. Cependant, étant donné le grand nombre d'images (60 000) dans l'ensemble d'entraînement, il serait coûteux de calculer la fonction de coût sur l'intégralité des données à chaque itération. Nous avons donc opté pour une optimisation par *batches*, où les données sont divisées en lots de taille fixe à l'aide de la fonction `DataLoader` de PyTorch. Le code d'optimisation est donné ci-dessous :

```
BATCH_SIZE = 64
train_loader = Data.DataLoader(dataset=train_data, batch_size=BATCH_SIZE,
    shuffle=True)

EPOCH = 100
LEARNING_RATE = 0.005

autoencoder = AutoEncoder()  # Modele a entrainer
criterion = nn.MSELoss()  # Fonction de cout
optimizer = torch.optim.Adam(autoencoder.parameters(), lr=LEARNING_RATE)  # ADAM

# Boucle d'entrainement
for epoch in range(EPOCH):
    for step, (x, _) in enumerate(train_loader):
        inputX = x.view(-1, 28 * 28)  # Transformation en vecteurs
        _, decoded = autoencoder(inputX)  # Encodage et reconstruction

        loss = criterion(decoded, inputX)  # Calcul de la perte
        optimizer.zero_grad()  # Reinitialisation des gradients
        loss.backward()  # Calcul des gradients
        optimizer.step()  # Mise a jour des parametres

```

Listing 3: Optimisation de l'auto-encodeur avec ADAM

À l'issue de l'entraînement, les performances des modèles avec $p = 2$ et $p = 16$ seront comparées. Cette analyse portera sur :

- La qualité de reconstruction des chiffres à partir de l'espace latent.
- La représentation des clusters dans l'espace latent, en visualisant les différences entre $p = 2$ et $p = 16$ (plus tard, je n'ai pas eu le temps désolé).
- La convergence de la fonction de coût pour chaque configuration .

Ces comparaisons permettront d'évaluer l'impact de la dimension de l'espace latent sur les performances de l'auto-encodeur.

3.2 Comparaisons

Nous allons comparer les performances du réseau dans deux configurations : $p = 2$ et $p = 16$. Cette analyse permettra de mettre en évidence les compromis entre le degré de compression et la qualité de reconstruction, et la différence entre une et plusieurs couches de neurones.

3.2.1 Qualité de reconstruction des chiffres à partir de l'espace latent

A) Auto-encodeur avec une couche intermédiaire

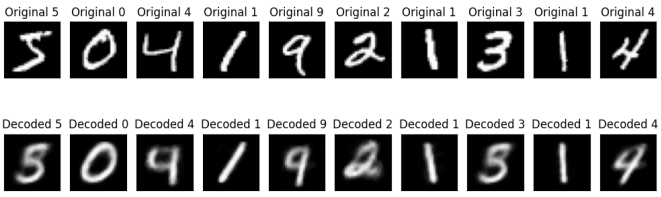

Figure 1: Reconstruction avec un auto-encodeur d'une couche intermédiaire, de dimension d'espace latent deux et EPOCH cent.

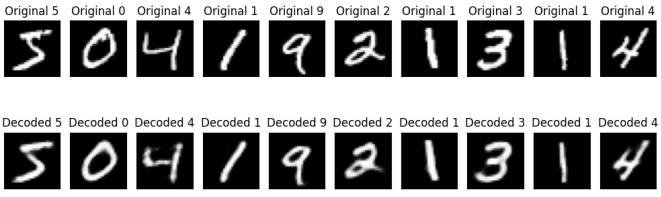

Figure 2: Reconstruction avec un auto-encodeur d'une couche intermédiaire, de dimension d'espace latent seize et EPOCH cent.

Nous remarquons que dans la Figure 1 les images décodées ont comme un effet de flou, comme un bruit, et certains chiffres ressemblent à d'autres comme le 5 qui ressemble à un 8 ou le 4 qui ressemble à un 9. C'est l'inconvéniant d'un espace latent de petite dimension, on a bien la compression des

données, mais il existe une perte d'information et une confusion. Contrairement à la Figure 2, d'espace latent de dimension 16 où on distingue parfaitement les chiffres décodés mais on se doute qu'ils ne sont pas autant compressé que dans le cas de l'espace latent de plus petite dimension.

B) Auto-encodeur avec une deux couches intermédiaires

Introduisons notre code avec deux couches intermédiaires avec des fonctions d'activations pour un espace latent p quelconque à définir :

```python
import torch.nn as nn

class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()
        # Definition de l'encodeur (R^n -> R^p avec deux couches intermediaires)
        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, 256),  # Premiere couche intermediaire
            nn.Tanh(),                # Fonction d'activation Tanh
            nn.Linear(256, 128),      # Deuxieme couche intermediaire
            nn.Tanh(),                # Fonction d'activation Tanh
            nn.Linear(128, p)         # Espace latent de dimension p
        )
        # Definition du decodeur (R^p -> R^n avec deux couches intermediaires)
        self.decoder = nn.Sequential(
            nn.Linear(p, 128),        # Premiere couche intermediaire
            nn.Tanh(),                # Fonction d'activation Tanh
            nn.Linear(128, 256),      # Deuxieme couche intermediaire
            nn.Tanh(),                # Fonction d'activation Tanh
            nn.Linear(256, 28 * 28), # Reconstruction vers R^n
            nn.Sigmoid()              # Fonction d'activation Sigmoid
        )

    def forward(self, x):
        encoded = self.encoder(x)   # Etape d'encodage
        decoded = self.decoder(encoded)  # Etape de decodage
        return encoded, decoded
```

Listing 4: Auto-encodeur avec deux couches intermédiaires

Après l'entraînement, nous obtenons :

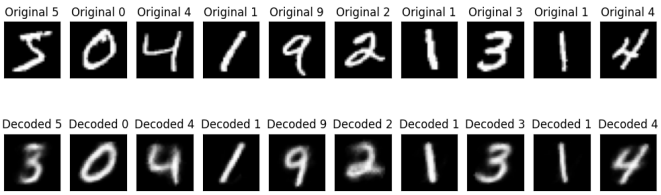

Figure 3: Reconstruction avec un auto-encodeur de 2 couche intermédiaire, de dimension d'espace latent deux et EPOCH cent.

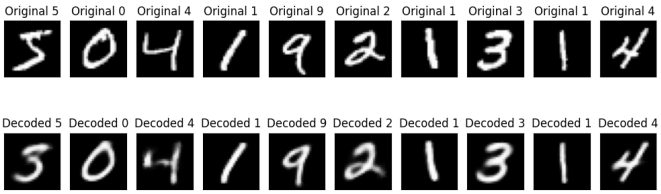

Figure 4: Reconstruction avec un auto-encodeur de 2 couche intermédiaire, de dimension d'espace latent huit et EPOCH cent.

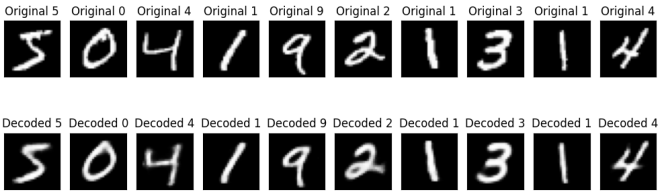

Figure 5: Reconstruction avec un auto-encodeur de 2 couche intermédiaire, de dimension d'espace latent seize et EPOCH cent.

3.3 Erreur quadratique moyenne

Convergence de la fonction coût pour l'auto-encodeur avec 2 couches de neurones, un espace latent de dimension 2.....