Dauphine | PSL
UNIVERSITÉ PARIS

Préimage empirique en IA générative

MÉMOIRE DE M1 MATHÉMATIQUES

Benjamin KRIEF

Dan WINSZMAN

Tiffany ZEITOUN

Encadré par Gabriel Turinici

24 avril 2024

Table des matières

Remerciements

Avant toute chose, nous tenions à remercier l'ensemble des personnes qui ont contribué à l'organisation et à la mise en place de ce mémoire. En particulier, nous remercions notre encadrant Gabriel Turinici pour sa disponibilité et son expertise. Son accompagnement tout au long de cette passionnante expérience a été extrêmement utile, en nous dispensant d'excellents conseils. Nous souhaitons également exprimer notre gratitude envers Angelina Roche pour ses conseils bienveillants lors de la pré-soutenance ainsi qu'à Pierre Cardaliaguet qui a su se montrer à l'écoute tout au long de l'année.

Introduction

Dans ce rapport de mémoire, nous vous proposons une reconstitution du fruit de notre travail et de nos réflexions profondes concernant la préimage empirique en IA générative.
Cette étude nous a engagé vers une exploration d'un univers nouveau et complexe, nous permettant d'explorer les mécanismes des réseaux de neurones en nous fournissant un aperçu précis des défis mathématiques qui sont à la base des progrès en intelligence artificielle.

À l'heure actuelle, l'IA et plus précisément le deep learning est un domaine en pleine expansion et est à l'origine de nombreux questionnements scientifiques, technologiques mais aussi éthiques et sociétaux. La rapidité avec laquelle cette technologie évolue et se déploie peut soulever certaines interrogations sur des réglementations nécessaires et indispensables pour assurer un développement responsable et bénéfique pour la société.

Ce mémoire a pour but, non seulement de se familiariser avec ces différentes notions et concepts d'un point de vue intuitif et mathématiques, mais aussi de voir une potentielle amélioration d'une technique déjà existante concernant des réseaux de neurones appelés auto-encodeurs variationnels.

1 Présentation du cadre

1.1 Définitions et généralités

Commençons par définir ce qu'est un réseau de neurones. Un réseau de neurones est construit à l'instar du cerveau humain et utilise l'intelligence artificielle afin d'apprendre aux ordinateurs à traiter des données de manière autonome. Pour cela, de nombreux entraînements sont réalisés pour améliorer la capacité et l'efficacité du réseau. Il va alors apprendre de ses erreurs et pouvoir par la suite traiter de nouvelles données. Il est composé de plusieurs couches : une couche d'entrée appelée Input Layer qui reçoit les données, de couches cachées de neurones interconnectés, les Hidden Layers, et une couche de sortie, l'Output Layer. Les réseaux de neurones sont très utilisés pour résoudre divers problèmes, tels que la classification d'images, la prédiction de séries temporelles, ou la génération de texte.

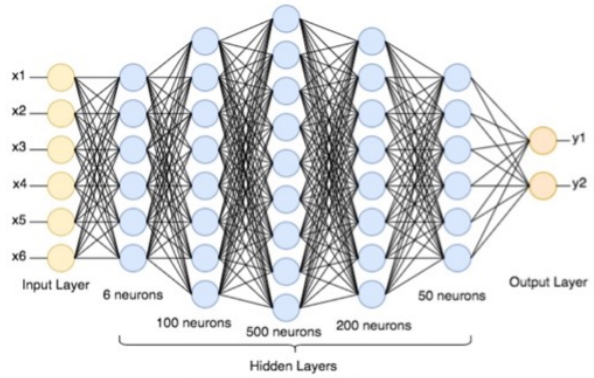


Figure 1: Fonctionnement d'un réseau de neurones

L'auto-encodeur est un réseau de neurones dont l'objectif est d'avoir la sortie la plus proche possible de l'entrée. Dans un auto-encodeur, le processus se déroule en deux phases : l'encodage, où l'encodeur compresse les données qu'il reçoit et les envoie dans l'espace latent, et le décodage, où les données sont reconstruites par le décodeur à partir de cette représentation compressée. La couche d'entrée et de sortie doivent être de même dimension et l'espace latent de dimension inférieure. Ce type d'architecture est appelé "bottleneck". Ainsi, l'espace latent correspond à l'espace contenant les données compressées, se situant entre l'encodeur et le décodeur. L'objectif de sa mise en place est de restreindre le flux d'informations entre les deux éléments de l'auto-encodeur et de minimiser le coût de reconstruction entre la sortie et l'entrée. Cette restriction se manifeste par une élimination du bruit pour ne laisser passer que les informations essentielles malgré la perte d'information inévitable dans le processus.

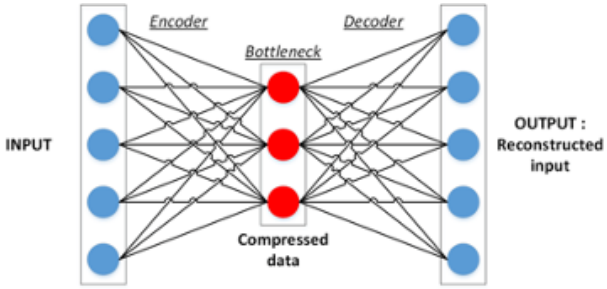


Figure 2: Organisation d'un auto-encodeur

Voici une illustration de ce processus avec une reconstruction du chiffre 2 à l'aide d'un auto-encodeur :

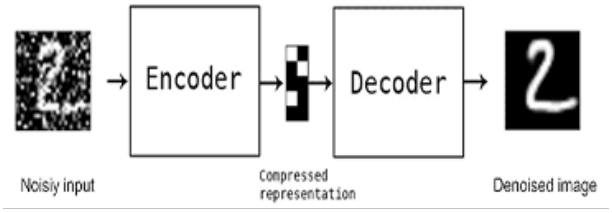


Figure 3: Reconstitution du chiffre 2

Sur cet exemple, l'encodeur reçoit une image bruitée, c'est à dire possédant des altérations compromettant sa qualité. Nous pouvons le voir sur ce chiffre 2 qui a de nombreux points blancs le rendant imparfait. L'encodeur va la compresser en l'envoyant dans l'espace latent afin de conserver uniquement les caractéristiques essentielles des données tout en éliminant le bruit. Enfin, le décodeur va reconstruire l'image à partir des informations de l'espace latent dans le but de fournir l'image la plus proche possible de celle d'origine, tout en l'améliorant. C'est bien le cas ici où le 2 affiché en sortie est de bien meilleure qualité que celui en entrée et donc l'utilisation de l'auto-encodeur a été bénéfique.

L'ACP est également une méthode de réduction de dimension, mais contrairement à l'auto-encodeur, il s'agit d'une méthode de compression linéaire. De ce fait, un auto-encodeur permet d'encoder rapidement les données sans perdre trop d'informations.

Intéressons-nous maintenant à un type d'auto-encodeur particulier défini dans le sujet : le VAE. Possédant les mêmes caractéristiques qu'un auto-encodeur traditionnel, il permet également de modéliser la distribution des données. Ainsi, il peut générer de nouvelles données ne se trouvant pas forcément dans son jeu de données d'entraînement.

Il y a de nombreux exemples concrets d'applications utilisant les différentes architectures de réseaux de neurones mentionnées. Intéressons-nous par exemple à comment les auto-encodeurs peuvent être utilisés dans le contexte spécifique de l'imagerie par résonance magnétique (IRM) pour la reconstruction d'images. L'IRM est une technique médicale créée en 1973 et devenue un outil indispensable pour les professionnels de santé permettant la visualisation de structures internes du corps humain avec une résolution élevée. Dans ce contexte, les auto-encodeurs peuvent permettre de reconstruire des images provenant de scanners IRM.

1.2 Cadre et objectifs

Pour effectuer ce travail, il nous est fourni la base de données MNIST. Ce dataset regroupe 60 000 images de chiffres entre 0 et 9 écrits à la main.

Figure 4: Dataset MNIST

Chaque image est un vecteur de $\mathbb{R}^{784}$. Chaque coordonnée vaut 0 (blanc) ou 1 (noir). Voici un exemple concret avec l'image n°78, correspondant au chiffre 1 :

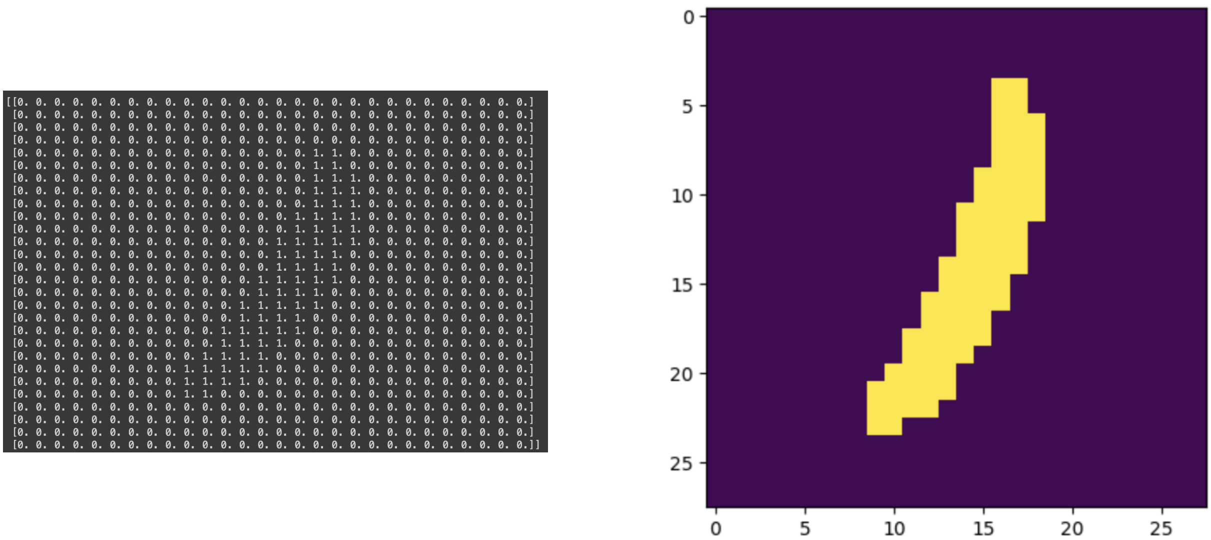


Figure 5: Image n°78 du dataset MNIST

D'une part, la structure d'un auto-encodeur variationnel permet de visualiser la distribution des chiffres dans un espace de dimension inférieure à 3 :

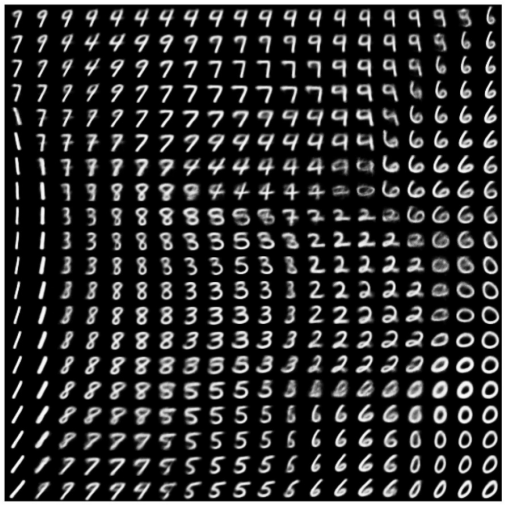


Figure 6: Exemple d'espace latent de dimension 2

D'autre part, elle permet de prédire une image incomplète. Imaginons qu'après entrainement du modèle, on fournisse en amont un chiffre dont il manque une partie. On voudrait que la structure puisse compléter l'image au mieux. Le but serait donc de limiter la perte d'information sur l'espace latent, afin d'obtenir une plus grande diversité.

1.3 Formulations mathématiques

Maintenant que nous avons posé le cadre du problème, nous allons définir mathématiquement les outils utilisés. Pour la suite de ce rapport, notons N la dimension de l'espace de départ et L celle de l'espace latent. En pratique, nous prendrons $N = 784$ et $L = 2$.
Comme expliqué dans le sujet, on considère un dataset de données empiriques :

$$\mu_e = \frac{1}{M} \sum_{l=1}^{M} \delta_{z_l}$$

où M est le nombre d'images du dataset, et $z_l \in \mathbb{R}^N$ une image du dataset donnée en entrée suivant une distribution inconnue μ.

Comme vu précédement, un auto-encodeur nécessite l'utilisation d'une fonction encodeur E.

Définition 2.1: Encodeur

Une fonction encodeur E est une fonction permettant de comprimer des données d'entrée dans un espace de dimension réduite. Elle est définie comme suit :

$$E : \mathbb{R}^N \longrightarrow \mathbb{R}^L$$

$$z_k \mapsto E(z_k)$$

L'encodeur prend ainsi les données d'entrée z_k et les mappe vers une représentation compressée $E(z_k)$ dans l'espace latent de dimension inférieure.

De la même manière, il faut définir une fonction décodeur.

Définition 2.2: Décodeur

Une fonction décodeur D est une fonction permettant de reconstruire des données à partir d'une représentation compressée dans l'espace latent. Elle est définie comme suit :

$$D : \mathbb{R}^L \longrightarrow \mathbb{R}^N$$

$$y_k \mapsto D(y_k)$$

Le décodeur prend la représentation encodée $E(z_k)$ et la décode pour reconstruire les données d'entrée, tout en cherchant à minimiser la perte d'information lors de cette reconstruction.

Nous pouvons donc synthétiser ce fonctionnement par le schéma suivant :

$$\mathbf{z}_k \xrightarrow{E} E(\mathbf{z}_k) \xrightarrow{D} D \circ E(\mathbf{z}_k)$$

Il est également important de noter que ces deux fonctions ont été faites simultanément par un réseau de neurones et ont été fournies à travers un code Python.

2 Inversion du décodeur

2.1 Motivation

Le défi de l'auto-encodeur est de reconstruire une image qui se rapproche le plus possible de l'image initiale reçue par l'encodeur. De ce fait, son objectif est de limiter au maximum la perte d'information lors de la compression du jeu de données par l'encodeur. Mathématiquement, cela signifie que l'on souhaite que l'application $E \circ D$ soit la plus proche possible de l'application identité. Après de nombreuses discussions avec Pr. Turinici, une piste pour améliorer la diversité du modèle est envisagée. En effet, la réduction de dimension par la fonction E entraine une perte d'information significative, qui est recouverte par D. Il serait donc possible d'utiliser la fonction D afin de recréer un nouvel espace latent, à partir du dataset MNIST fourni.
L'idée est donc de remplacer l'application E par l'inverse de l'application D, ce qui nous permettra d'obtenir une reconstruction plus précise. Ceci constitue l'objectif majeur de notre mémoire.
Pour inverser l'application D, nous avons envisagé deux méthodes: la première consiste à reformuler notre problème en un problème de point fixe et la seconde est d'utiliser un algorithme de descente de gradient. Pour implémenter une descente de gradient, il est crucial de choisir de manière judicieuse le pas de descente et l'initialisation. La convergence de cette méthode pouvant être plus lente et dépendant fortement du choix des paramètres, nous avons décidé d'explorer dans un premier temps la piste du point fixe.

2.2 Méthode du point fixe

Dans un premier temps, nous allons reformuler de manière non rigoureuse notre problème initial qui consistait à inverser D, en un problème de point fixe:
Soit $Z \in \mathbb{R}^N$: on cherche $u \in \mathbb{R}^L$ tel que :

$$\begin{aligned} & D(u) = Z \\ \iff & E(D(u)) = E(Z) \\ \iff & u - E(D(u)) = u - E(Z) \\ \iff & E(Z) - E(D(u)) + u = u \\ \iff & h_Z(u) = u \end{aligned}$$

où $h_Z(u) = E(Z) - E(D(u)) + u$.

Ainsi, sous certaines hypothèses, trouver le point fixe de l'application h_Z revient à inverser l'application D. À présent, interrogeons nous sur les conditions sous lesquelles la fonction h_Z possède un unique point fixe.

Commençons par rappeler le théorème du point fixe de Banach-Picard.

Théorème 3.1: Point fixe de Banach-Picard

Soit (X, d) un espace métrique complet, et $f : X \to X$ une application contractante, ie. il existe $k \in [0, 1[$ tel que, pour tout $(x, y) \in X^2$, $d(f(x), f(y)) \leq k \cdot d(x, y)$.
Alors f possède un unique point fixe .

Ici, on a $X = \mathbb{R}^L$ muni de sa norme usuelle qui est complet. De plus, l'application h_Z est contractante si et seulement si l'application $Id - E \circ D$ est contractante :
En effet, soient $u_1, u_2 \in \mathbb{R}^L$:

$$\begin{aligned} \|h_Z(u_1) - h_Z(u_2)\| &= \|u_1 - u_2 - E(D(u_1)) + E(D(u_2))\| \\ &= \|(Id - E \circ D)(u_1) - (Id - E \circ D)(u_2)\| \end{aligned}$$

Donc montrer que h_Z contractante est équivalent à montrer que $Id - E \circ D$ est contractante.

Ainsi, en supposant que $Id - E \circ D$ est contractante, on peut appliquer le théorème du point fixe qui nous assure l'existence et l'unicité d'un point fixe pour l'application h_Z que l'on peut déterminer. En rajoutant les hypothèses qui rendent équivalent notre problème initial au problème de point fixe, on obtient alors que D est inversible et son inverse est précisément le point fixe de h_Z.

Pour justifier ces équivalences, on doit se questionner sur la surjectivité de D et l'injectivité de E. Premièrement, supposer la surjectivité de D reviendrait à écrire l'assertion suivante :

$$\forall Z_i \in \mathbb{R}^{784}, \exists Y_k \in \mathbb{R}^2 \text{ tel que } D(Y_k) = Z_i$$

En d'autres termes, quelque soit l'image caractérisée par un vecteur de $\mathbb{R}^{784}$, on pourrait trouver un point de l'espace latent qui soit son antécédent par la fonction D. Or, l'écart de dimensions (passage de 2 à 784) semble aller à l'encontre du fait que D soit surjective. En effet, on admettrait ainsi que l'espace latent contienne toute l'information nécessaire pour représenter toutes les images possibles, en tenant compte de leur spécificité. Illustrons nos propos par un exemple :

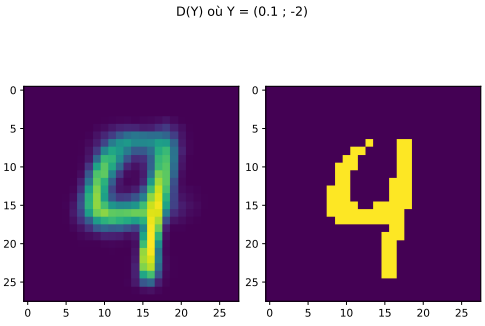


Figure 7: Decodeur d'un vecteur de $\mathbb{R}^2$ en gris et noir et blanc.

NB : Bien qu'en jaune et violet, elle est représentée en noir et blanc en termes de dimensions dans l'espace $\mathbb{R}^2$ (idem pour les nuances de gris).

En plus de ne pas savoir si cette image correspond à un 4 ou un 9, aucune image du dataset MNIST ne correspond à celle-ci. En effet, après avoir comparé tous les 60 000 vecteurs de $\mathbb{R}^{784}$ du dataset, aucun n'est égal à celui obtenu dans l'image ci-dessus.

Deuxièmement, supposer l'injectivité de E reviendrait à écrire l'assertion suivante :

$$\forall Z_{k_1}, Z_{k_2} \in \mathbb{R}^{784}, \qquad E(Z_{k_1}) = E(Z_{k_2}) \quad \Rightarrow \quad Z_{k_1} = Z_{k_2}$$

Si deux points de l'espace latent sont les mêmes, cela impliquerait donc que les images initiales soient exactement les mêmes ? Encore une fois, l'écart entre les dimensions implique une perte d'information et de diversité à ne pas négliger.

En conclusion, nous ne pouvons pas supposer la surjectivité et l'injectivité respectives de D et E. Par conséquent, les recherches sur l'alogrithme du point fixe, bien que prometteuses, doivent être avortées. En substitut, nous allons nous reposer sur un autre algorithme d'optimisation: la descente de gradient.

2.3 Méthode de la descente de gradient

Une autre approche est d'utiliser une méthode de descente de gradient afin d'inverser l'application décodeur D. Cette méthode permet d'approcher, à l'aide d'un algorithme la valeur du minimum d'une fonction. Bien qu'ayant une lente convergence, elle s'avère utile et pratique dans de nombreux domaines. En effet, la simplicité de cette méthode n'empêche pas sa puissance notamment en grande dimension, et en fait donc une option de choix pour la résolution de notre problème.

2.3.1 Cas général

Considérons une fonction $f : \mathbb{R}^d \to \mathbb{R}$ et résolvons le problème de minimisation

$$\min_{x \in \mathbb{R}^d} f(x).$$

Supposons que l'on parte d'un point initial $x_0 \in \mathbb{R}^d$ et d'un pas $\tau > 0$. L'idée est de trouver la direction dans laquelle notre fonction décroît le plus. Pour cela, on cherche la direction $v \in S^{d-1}$ telle que

$$f(x_0 + \tau v) < f(x_0).$$

Le développement de Taylor nous donne :

$$f(x_0 + \tau v) = f(x_0) + \tau \langle v, \nabla f(x_0) \rangle.$$

Or, par l'inégalité de Cauchy-Schwarz,

$$|\langle v, \nabla f(x_0) \rangle| \leq \|\nabla f(x_0)\|,$$

car $v \in S^{d-1}$.
D'où

$$\forall v \in S^{d-1}, \quad f(x_0 + \tau v) = f(x_0) + \tau \langle v, \nabla f(x_0) \rangle \geq f(x_0) - \tau \|\nabla f(x_0)\|.$$

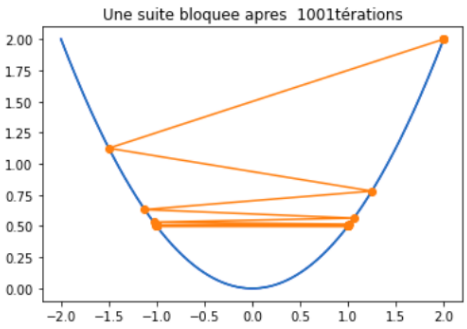
La borne inférieure est atteinte lorsque

$$v^* = -\frac{\nabla f(x_0)}{\|\nabla f(x_0)\|}.$$

Il est donc approprié de choisir ce v^* comme direction de descente.
On définit donc la suite d'itération :

$$\forall k \in \mathbb{N}, \quad x_{k+1} = x_k - \tau \nabla f(x_k).$$

Reste à choisir l'initialisation et le pas. En effet, un mauvais choix de pas ou d'initialisation peut avoir d'énormes conséquences sur l'efficacité ou la vitesse de l'algorithme.
Prenons un exemple qui montre l'importance du choix du pas:



On remarque dans cette figure que la suite est bloquée après 1001 itérations et oscille entre -1 et 1, ne pouvant donc pas atteindre le minimum de la fonction à optimiser. Cela est expliqué par le fait que le pas est toujours bien trop grand. À contrario si le pas est trop petit, la convergence peut être trop lente, entraînant un temps de calcul démesuré. Il est possible de remédier à ce problème par la recherche d'un pas optimal en utilisant par exemple les règles de Wolfe ou d'Armijo que nous ne détaillerons pas ici.
De la même manière le choix de l'initialisation est crucial. En effet, si le point d'initialisation est trop éloigné du minimum, l'algorithme peut nécessiter un nombre conséquent d'itérations avant de converger vers la solution, voire être coincé dans des minimums locaux ou des points selle.

2.3.2 Exemple d'implémentation d'une descente de gradient

```
def algo(f, df, x0, tau, tol, Niter):
    xn = np.array(x0)
    L = []
    for n in range(Niter):
        grad = df(xn)
        if np.linalg.norm(grad) < tol:
            return xn, L
        else:
            L.append(xn.copy())
            xn = xn - tau * grad
    print("Erreur, l algorithme n a pas converge apres", Niter, "
       iterations")
    return xn, L
```

Cet algorithme prend en entrée la fonction f à minimiser, son gradient ∇f (supposé connu), un pas fixé $\tau > 0$, une tolérance tol, et un nombre d'itérations maximal N_{iter}.
À chaque itération n, l'algorithme calcule la valeur de $x_{n+1} = x_n - \tau \cdot \nabla f(x_n)$ et s'arrête lorsque le critère d'arrêt est réalisé. Si le nombre d'itération maximale est atteint avant qu'il soit réalisé, cela renvoie une erreur signifiant que l'algorithme n'a pas convergé (mauvais choix du pas, nombre d'itérations maximal trop faible, etc...).

2.3.3 Application à l'inversion du décodeur

Soit $i \in \{0, 1, \ldots, 59\,999\}$, $k \in \mathbb{N}$.
Soit (Z_i) une suite de $\mathbb{R}^N$, on cherche $(Y_k)_{k\in\mathbb{N}}$ une suite de $\mathbb{R}^L$ telle que $D(Y_k) = Z_i$. Cela revient à trouver $(Y_k)_{k\in\mathbb{N}}$ solution du problème de minimisation suivant:

$$\min_{Y_k\in\mathbb{R}^L} \|D(Y_k) - Z_i\|_2^2 .$$

Pour résoudre cela, on utilise la méthode du gradient.
On se fixe un pas de descente $\tau > 0$ et un point de départ $Y_0 \in \mathbb{R}^L$. Posons alors :

$$Y_{k+1} = Y_k - \tau \nabla f(Y_k),$$

$$\text{avec } f(Y_k) = \|D(Y_k) - Z_i\|_2^2 \ \textit{la loss function.}$$

La démarche est donc la suivante : trouver des points de l'espace latent qui minimiseraient la fonction f. Après avoir réalisé l'algorithme pour 5 000 des 60 000 images du dataset, nous aurons donc un nouvel espace latent.

NB : Le temps de calcul nous oblige à se restreindre à 5 000 images.

3 Code

Le code fourni par Pr. Turinici nous a permis d'exécuter les tâches suivantes :

- importation des données MNIST à partir de son GitHub, stockées dans la liste *train_images*.
- introduction de la classe CVAE, l'autoencodeur variationnel, pour laquelle nous allons rentrer plus en détail par la suite.
- stockage des poids, c'est-à-dire sauvegarde des fonctions encodeur et décodeur post-entrainement.
- importation de *reparam*, une liste de liste de taille 2, correspondant aux 60 000 points de l'espace latent du modèle classique.

Avant de rentrer dans le détail du code et de nos résultats, il nous semblait important de mettre en avant un problème rencontré lors de l'implémentation du code:

ValueError: File format not supported: filepath=./my_checkpoint. Keras 3 only supports V3 '.keras' and '.weights.h5' files, or legacy V1/V2 '.h5' files.

Du fait d'un problème de conversion de fichiers sur MAC OSX, il était impossible d'enregistrer les versions post-entrainement de l'encodeur et du décodeur. Afin d'illustrer ce propos, prenons arbitrairement la 555-ème image du dataset et constatons la différence nette entre l'affichage pré et post entrainement. C'est pourquoi nous avons décidé de travailler sur Google Colab.

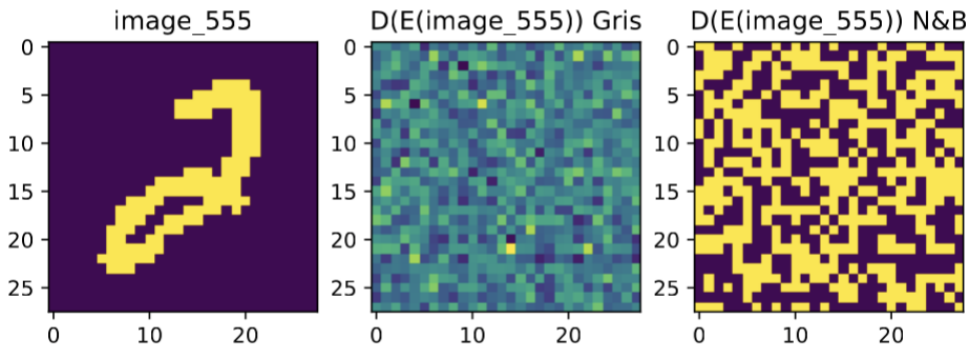


Figure 8: Image 555 sans entraînement: à gauche l'image originelle, au milieu l'image encodée puis décodée en nuance de gris, à droite l'image encodée puis décodée en noir et blanc.

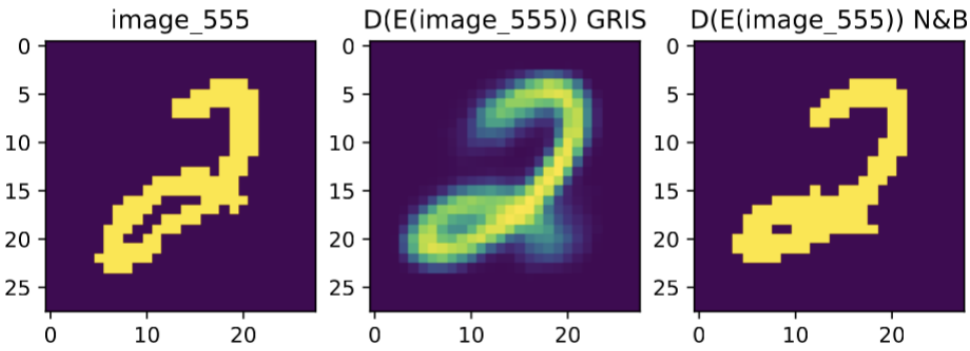


Figure 9: Image 555 avec entraînement: à gauche l'image originelle, au milieu l'image encodée puis décodée en nuance de gris, à droite l'image encodée puis décodée en noir et blanc.

3.1 Classe CVAE

Dans cette partie, nous allons expliquer le fonctionnement d'une classe nommée CVAE (Convolutional variational autoencoder) d'après le code qui suit :

```
N=2
class CVAE(tf.keras.Model):
    """Convolutional variational autoencoder."""

    def __init__(self, latent_dim):
        super(CVAE, self).__init__()
        self.latent_dim = latent_dim

        self.encoder = tf.keras.Sequential(
            [tf.keras.layers.InputLayer(input_shape=(28, 28, 1)),
             tf.keras.layers.Flatten()] +
            [tf.keras.layers.Dense(28*28, activation='relu') for ii
               in range(5)] +
            [tf.keras.layers.Dense(latent_dim + latent_dim)])

        self.decoder = tf.keras.Sequential(
            [tf.keras.layers.InputLayer(input_shape=(latent_dim,)),
             tf.keras.layers.Flatten(),
             tf.keras.layers.Dense(28*28, activation='relu'),
             tf.keras.layers.Dense(28*28, activation='relu'),
             tf.keras.layers.Dense(28*28, activation='relu'),
             tf.keras.layers.Dense(28*28, activation='relu'),
             tf.keras.layers.Dense(28*28),
             tf.keras.layers.Reshape((28,28,1)),
            ])

    @tf.function
    def sample(self, eps=None):
        if eps is None:
```

```
            eps = tf.random.normal(shape=(100, self.latent_dim))
        return self.decode(eps, apply_sigmoid=True)

    def encode(self, x):
        mean, logvar = tf.split(self.encoder(x), num_or_size_splits
            =2, axis=1)
        return mean, logvar

    def reparameterize(self, mean, logvar):
        eps = tf.random.normal(shape=mean.shape)
        return eps * tf.exp(logvar * .5) + mean

    def decode(self, z, apply_sigmoid=False):
        logits = self.decoder(z)
        if apply_sigmoid:
            probs = tf.sigmoid(logits)
            return probs
        return logits
```

Dans un premier temps, nous allons analyser les attributs de la classe CVAE (ie. les données associées):

- **`self.latent_dim`** :
  Stocke la dimension de l'espace latent.

- **`self.encoder`** :
  Contient le modèle de l'encodeur qui transforme les données d'entrée en une représentation latente. La première couche, l'Input Layer correspondant à la première étape de l'encodeur reçoit des images en niveau de gris de taille 28x28 pixels. Puis la couche suivante transforme les images 2D en 1D, étape nécessaire pour la suite du processus. Chaque image est alors représentée par un vecteur de 28x28=784 éléments. Ensuite, on génère cinq couches denses, chacune possédant 784 neurones. Ces dernières permettent de capter des relations complexes dans les données en utilisant la fonction d'activation ReLU (Rectified Linear Unit) dans le cadre de l'entraînement du réseau, étape cruciale pour la configuration de l'espace latent. Enfin, la dernière couche, l'Output Layer a une sortie de taille $2 \times$ *latent_dim* et renvoie deux paramètres *mean* (moyenne) et *logvar* (log-variance) permettant de caractériser la distribution de chaque image dans l'espace latent.

- **`self.decoder`** :
  Contient le modèle du décodeur qui transforme les données de la représentation latente en images. La première couche, l'Input Layer correspondant à la première étape du décodeur reçoit un vecteur de l'espace latent. Puis de la même manière que pour l'encodeur, une couche permet de transformer les images 2D en 1D, étape pouvant être redondante car le vecteur de l'espace latent est déjà en 1D. Ensuite, les cinq couches denses permettent d'élargir la dimensionnalité pour reconstruire l'image à partir du vecteur de l'espace latent en utilisant la fonction d'activation ReLU. Enfin, une couche retourne le vecteur final de taille 784 qui sera ensuite redimensionné par la dernière couche permettant d'obtenir l'image reconstruite en niveau gris de taille 28x28 pixels.

Analysons maintenant les méthodes de la classe CVAE (ie. les fonctions associées):

- **`__init__(self, latent_dim)`** :
  Constructeur de la classe initialisant les atributs de l'instance.

- **`sample(self, eps=None)`** :
  Cette méthode utilise la fonction décodeur pour produire de nouvelles images après l'entraînement. Le paramètre *eps* étant égal à None, elle génère aléatoirement, à partir d'une distribution normale des points dans l'espace latent et applique le décodeur pour obtenir des images de 28x28 pixels. Cela permet d'évaluer la capacité du modèle à généraliser et à créer de nouvelles données.

- **`encode(self, x)`** :
  Cette méthode prend en paramètre des images *x* et fait appel à l'attribut *encoder* pour les transformer et renvoyant *mean* et *logvar*, deux paramètres importants pour la représentation de l'espace latent.

- **`reparameterize(self, mean, logvar)`** :
  Cette méthode utilise les paramètres *mean* et *logvar* renvoyés par l'encodeur pour obtenir une distribution échantillonnée de l'espace latent et pouvoir ainsi générer de nouvelles données.

- **`decode(self, z, apply_sigmoid=False)`** :
  Cette méthode prend en entrée un échantillon de l'espace latent et utilise l'attribut *decoder* défini précédemment.

3.2 Algorithme de descente de gradient

Dans cette partie, nous allons expliquer le fonctionnement de notre descente de gradient:

```
# Importation des bibliotheques utilisees

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output
from tqdm import tqdm
from google.colab import files
from google.colab import drive
drive.mount('/content/drive')


def gradient_descent(mnist_index, file_path_loss_function = "", save
    = False):
  latent_dim = 2
  learning_rate = 1e-3
  optimizer = tf.keras.optimizers.Adafactor(learning_rate=
      learning_rate)

  losses = []

  image_cible = train_images[mnist_index,:,:,0]
```

```
    y = tf.Variable(tf.convert_to_tensor(encoder_function(image_cible))
        ) # Initialisation
    epochs = 150 # Nombre d'iterations

    for epoch in range(epochs):

        with tf.GradientTape() as tape:
            loss = tf.reduce_sum(tf.square(image_cible -
                decoder_function(y))) # fonction a minimiser
            losses.append(loss)

            gradients = tape.gradient(loss, [y]) # calcul du gradient
            optimizer.apply_gradients(zip(gradients, [y])) # ddg
    if save == True: # affichage de la loss function
      clear_output(wait=True)
      plt.figure(figsize=(10, 5))
      plt.plot(losses, label='Train Loss')
      plt.title(f'Loss evolution during the training of image {
          mnist_index}')
      plt.xlabel('Epoch')
      plt.ylabel('Loss')
      plt.legend()
      plt.grid(True)

      plt.savefig(file_path_loss_function)
      plt.show()
      files.download(file_path_loss_function)

    final_loss = losses[99].numpy()
    first_loss = losses[0].numpy()
    delta = ((final_loss - first_loss)/first_loss)*100
    solution = y.numpy()

    return(final_loss, delta, solution)
```

Rappel de notre problème de minimisation:

$$\min_{Y_k \in \mathbb{R}^L} \|D(Y_k) - Z_i\|_2^2$$

Quels sont les paramètres choisis pour effectuer cette descente de gradient ?

- Le pas : après plusieurs essais "à la main", combinés avec des recherches sur StackOverflow, on a choisi la valeur 0.001.
- L'initialisation : sachant que $D \circ E \approx \mathrm{Id}$, on prend $Y_0 = E(Z_i)$ et donc $D(Y_0) \approx Z_i$. Cette quantité est donc la plus proche en norme que l'on pourrait connaître.
- Le nombre d'itérations : après plusieurs essais où l'on a affiché la fonction *loss*, dont chaque valeur est stockée dans la liste *losses*, on choisit 150 itérations par descente de gradient.

Présentons maintenant les arguments de la fonction
gradient_descent(mnist_index, file_path_loss_function, save) :

- *mnist_index* est un entier permettant d'accéder à la *mnist_index*-ième image.
- *file_path_loss_function* est une chaîne de caractères correspondant à un chemin de fichier où sera enregistré le plot de la fonction *loss* si *save* est défini à `True`.

La fonction **gradient_descent** renvoit le tuple *(final_loss, delta, solution)*:

- *final_loss* : un float qui donne le résulat de la fonction à minimiser lors de sa dernière itération.
- *delta* : un float qui représente le taux de variation entre la première itération et la dernière itération de la loss function dans la descente de gradient. Elle nous permet de constater le gain que nous avons réalisé.
- *solution* : une liste de float de taille 2, correspondant à la solution de notre problème de minimisation.

Voici notre fonction **main**, étape finale dans la recherche d'un nouvel espace latent:

```
if __name__ == "__main__":
    nb_images = 5000
    final_errors = []
    deltas = []
    solutions = []
    save = False
    for i in tqdm(range(0,nb_images)):
      file_name_loss_function = f"loss_function_image_{mnist_index}.
         pdf"
      file_path_loss_function = f"/content/drive/My Drive/Thesis/
         image_{mnist_index}/{file_name_loss_function}"
      start_time = time.time()
      result = gradient_descent(mnist_index, file_path_loss_function,
          save)
      end_time = time.time()
      execution_time = end_time - start_time
      print(f"Le temps d'execution est de {execution_time} secondes."
         )

      final_errors.append(result[0])
      deltas.append(result[1])
      solutions.append(result[2].tolist())

    filepath_final_errors = '/content/drive/My Drive/Thesis/data/
       final_errors.txt'  #Simple list
    filepath_deltas = '/content/drive/My Drive/Thesis/data/deltas.txt
       '  #Simple list
    filepath_solutions = '/content/drive/My Drive/Thesis/data/
       solutions.txt' #Sub list
```

```
23
24      create_list_txt(final_errors, filepath_final_errors) # Annexe
25      create_list_txt(deltas, filepath_deltas) # Annexe
26      create_list_sub_txt(solutions, filepath_solutions) # Annexe
```

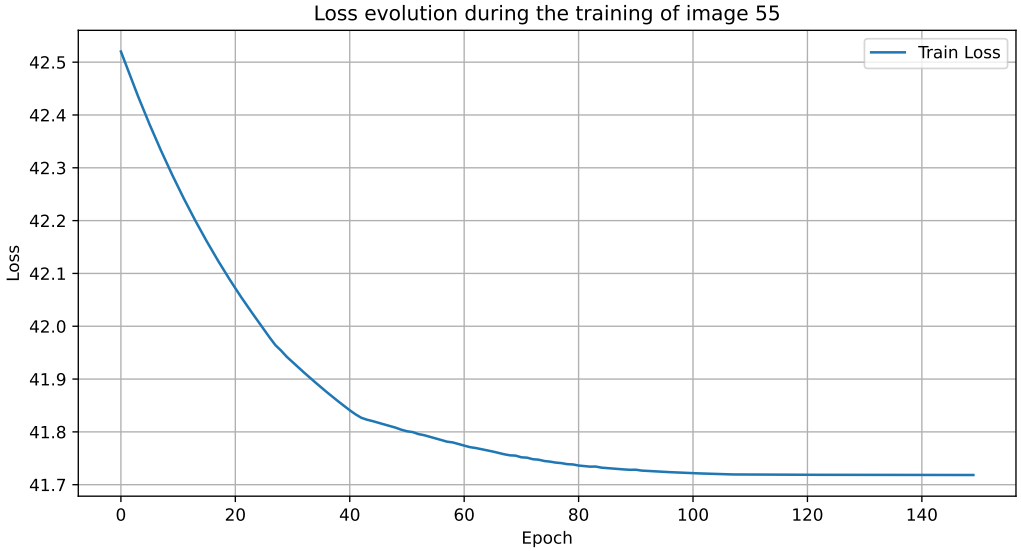
Pour plus de détails, voir l'annexe sur l'enregistrement des données.

On fait logiquement une boucle *for*, dans laquelle on appelle 5 000 fois **gradient_descent** pour 5 000 images. On enregistre ensuite les trois objets du tuple renvoyé à chaque itération dans 3 listes distinctes: *final_errors, deltas, solutions*. Enfin, on enregistre ces 3 listes (deux listes de taille 5 000 et une liste de 5 000 listes de taille 2) dans des fichiers txt.

*Commentaire sur le temps d'exécution : chaque appel de fonction **gradient_descent** (sans affichage de la loss function) prend en moyenne 1.20 seconde. Pour 5 000 images, on a donc un temps total de 6 000 secondes, soit 100 minutes. L'exécution fut donc globalement rapide grâce à l'utilisation du T4 GPU à la place du CPU.*

3.3 Résultats

3.3.1 Un cas particulier: l'image n°55



Dans ce cas particulier, on constate que l'algorithme **gradient_descent** appliqué à l'image n°55 (choisie arbitrairement) permet de trouver un nouveau point de l'espace latent. Ce point auquel on applique le décodeur est 1.8% plus proche en norme que si on avait appliqué au décodeur, l'encodeur de l'image cible (deltas[55] = -1.885710470378399) .

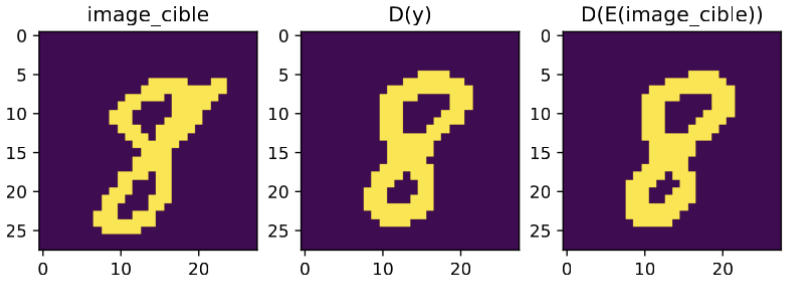


Figure 10: Au milieu D(y), où y est le point obtenu par notre descente de gradient pour l'image n°55: solutions[55].

Visuellement, on constate bien une légère amélioration au niveau de la boucle inférieure du 8, reconstituant plus précisément l'image cible.

3.3.2 Cas général : comparaison entre les deux espaces latents

Nous noterons par la suite **MT** l'espace latent du Pr. Turinici, contenu dans la liste *reparam* et **MN** le notre, contenu dans la liste *solutions*.

Données MT :
La moyenne vectorielle μ_{MT} est donnée par :

$$\mu_{MT} = \begin{bmatrix} -0.123 & 0.0281 \end{bmatrix}$$

La matrice de variance-covariance est donnée par :

$$\text{var\_covar}_{MT} = \begin{pmatrix} 1.16 & 0.092 \\ 0.092 & 1.38 \end{pmatrix}$$

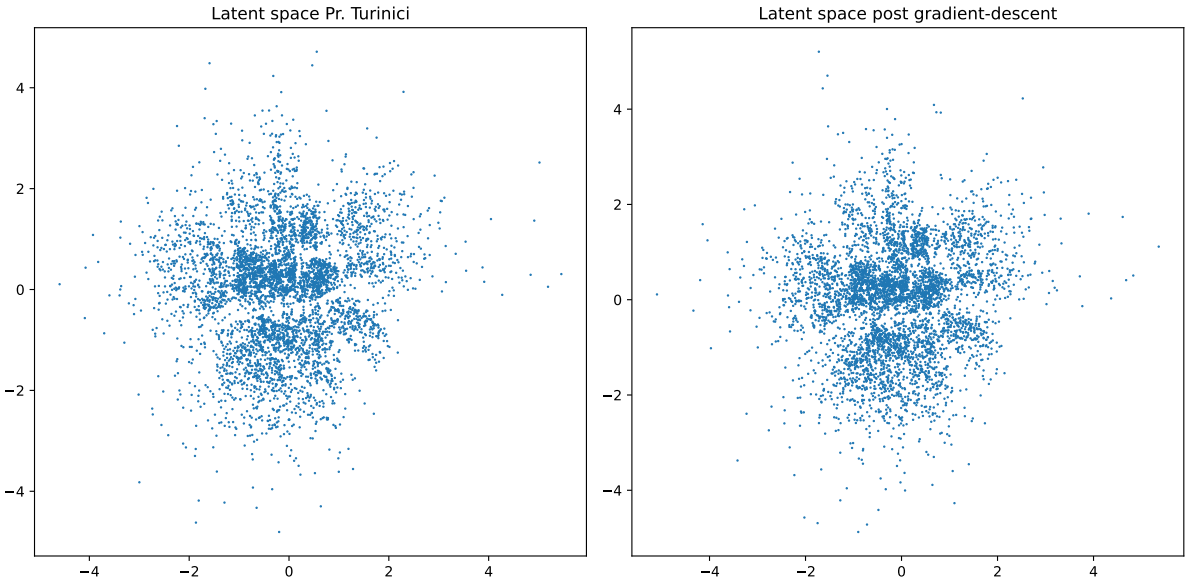
Données MN :
La moyenne vectorielle μ_{MN} est donnée par :

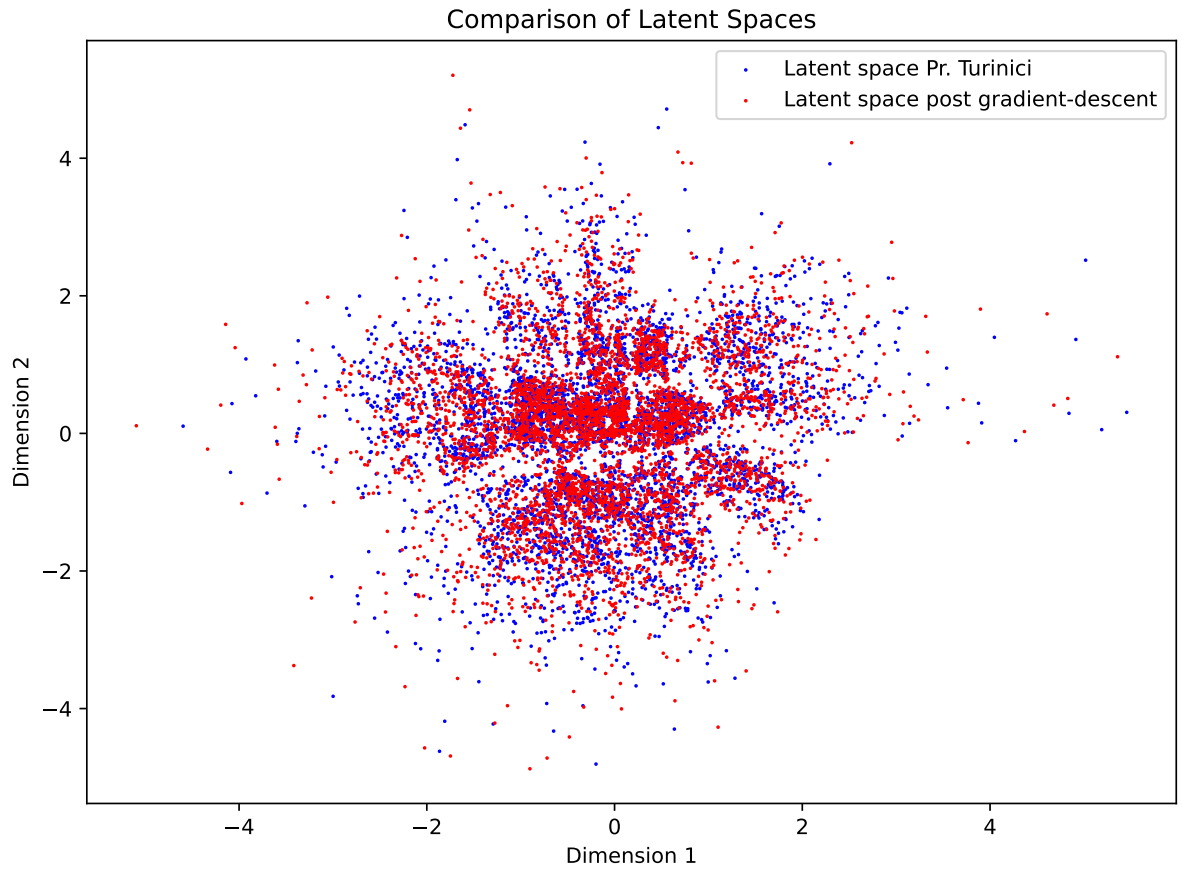
$$\mu_{MN} = \begin{bmatrix} -0.129 & 0.0175 \end{bmatrix}$$

La matrice de variance-covariance est donnée par :

$$\text{var\_covar}_{MN} = \begin{pmatrix} 1.14 & 0.097 \\ 0.097 & 1.36 \end{pmatrix}$$

où:
$z_i \in \mathbb{R}^{784}$ est la i-ème image du dataset,
$MN_i \in \mathbb{R}^2$ le i-ème point de l'espace latent MN,
$MT_i \in \mathbb{R}^2$ le i-ème point de l'espace latent MT.





Nous avons aussi calculé les distances entre les points de l'espace **MT** et de l'espace **MN** associés à l'image n°i. Ces 5 000 distances sont stockées dans un tableau dont on calcule la moyenne et la variance.

```
Mean of distance between MT & MN : 0.09743846487431138
Variance of distance between MT & MN: 0.012126977006570202
```

Conclusion partielle: les données sont assez proches et se ressemblent. Cependant, une question légitime se pose: **MN** est-il un meilleur espace latent que **MT**?

Analysons la liste *deltas*. Pour rappel, elle mesure les gains (variations entre le début et la fin) réalisés sur nos loss functions pour chaque image. Les indicateurs statistiques sur cette liste sont pour le moins intéressants. Bien entendu tous les éléments sont négatifs, par conséquent, nous gagnons en précision dans tous les cas!

Soit δ_{MN} le vecteur qui représente cette variation départ-arrivée: $\forall i \in \{0, 1, \ldots, 4999\}$,

$$(\delta_{\mathrm{MN}})_i = \left(\frac{\|z_i - D(MN_i)\|_2^2 - \|z_i - D(MT_i)\|_2^2}{\|z_i - D(MT_i)\|_2^2} \right) \times 100$$

La moyenne $\mu_{\delta_{MN}}$ est donnée par:

$$\mu_{\delta_{MN}} = -4.79$$

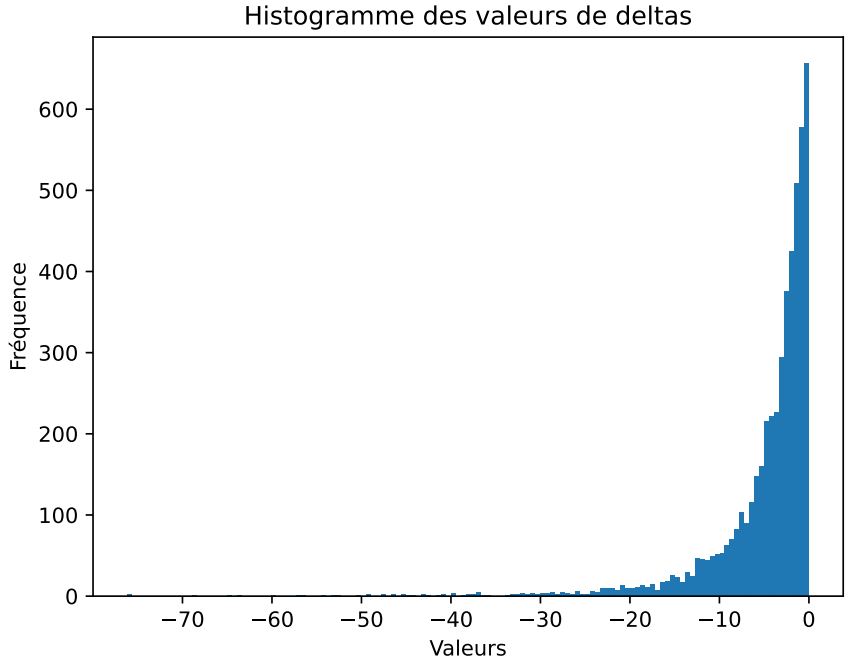
La variance est donnée par:

$$\mathrm{var}_{\delta_{MN}} = 41.91$$

Conclusion:

- Il nous est donc indiqué que notre gain moyen est d'environ 5%, ce qui est non négligeable.
- De plus, la variance est pour la moins surprenante, avec une valeur éloignée de la moyenne en valeur absolue. Il y a donc une forte dispersion du gain de notre algorithme.

Pour visualiser au mieux cette dispersion, regardons l'histogramme de *deltas*.



D’autres questions sont légitimes :

1. Avec quelles images a-t-on le mieux gagné (les meilleurs delta) ?

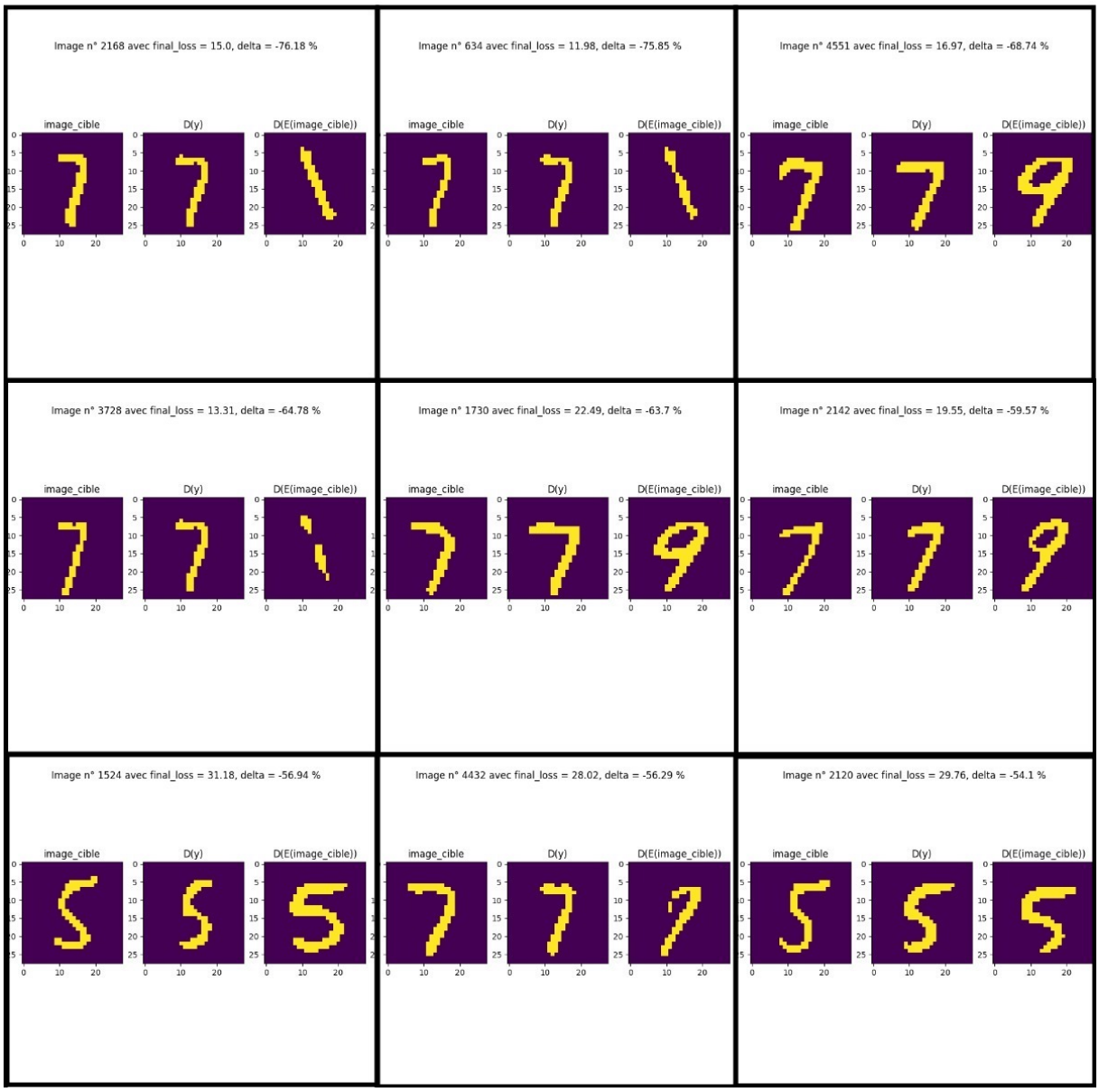


Figure 11: Top 9 des meilleurs delta

Comme nous pouvons le constater, il y a une nette amélioration dans la reconstitution de l’image cible avec la descente de gradient par rapport à l’application simple de l’encodeur suivi du décodeur. Prenons, par exemple, le premier lot d’images (situées en haut à gauche, n°2168). L’objectif est de renvoyer un 7 (image de gauche), mais l’image résultante de la méthode initiale est un 1 (image de droite), ce qui constitue un échec. Cependant, grâce à la descente de gradient, cette erreur est corrigée, permettant de retrouver le 7 escompté (image au milieu). Ainsi, la descente de gradient semble avoir un impact positif sur la reconstitution de l’image cible, marquant ainsi une réussite.

2. Quelles sont les images ayant les plus hautes et basses final_loss?

NB : la moyenne des final_loss pour chaque image vaut 34.66. Bien entendu l'objectif serait de tendre au plus vers 0. De plus, à l'initialisation la moyenne des loss valait 36.8.

Concernant cette question, nous avons obtenu les résultats suivants :

- Max des *final_loss*: 123.8 où ArgMax *final_loss*: 4883.
- Min des *final_loss*: 2.92 où ArgMin *final_loss*: 1002.

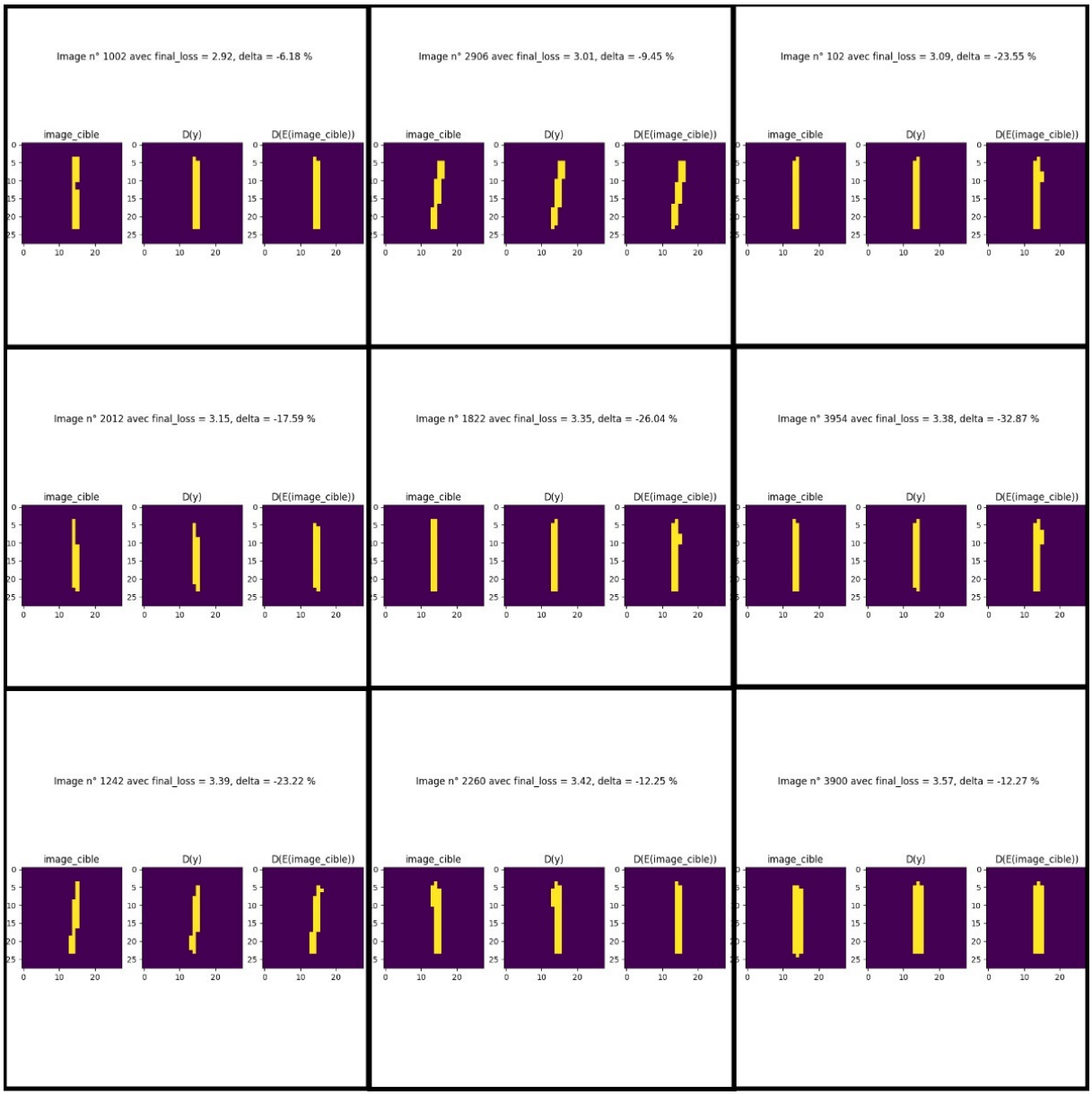


Figure 12: Top 9 des meilleures erreurs

Conclusion: Il semblerait naturel que l'image la plus simple à améliorer serait le chiffre 1. Cette hypothèse est confirmée empiriquement: les 500 meilleures erreurs, c'est-à-dire les plus petites, sont toutes des images ayant comme chiffre le 1.

Nous allons aussi afficher les 9 pires erreurs:

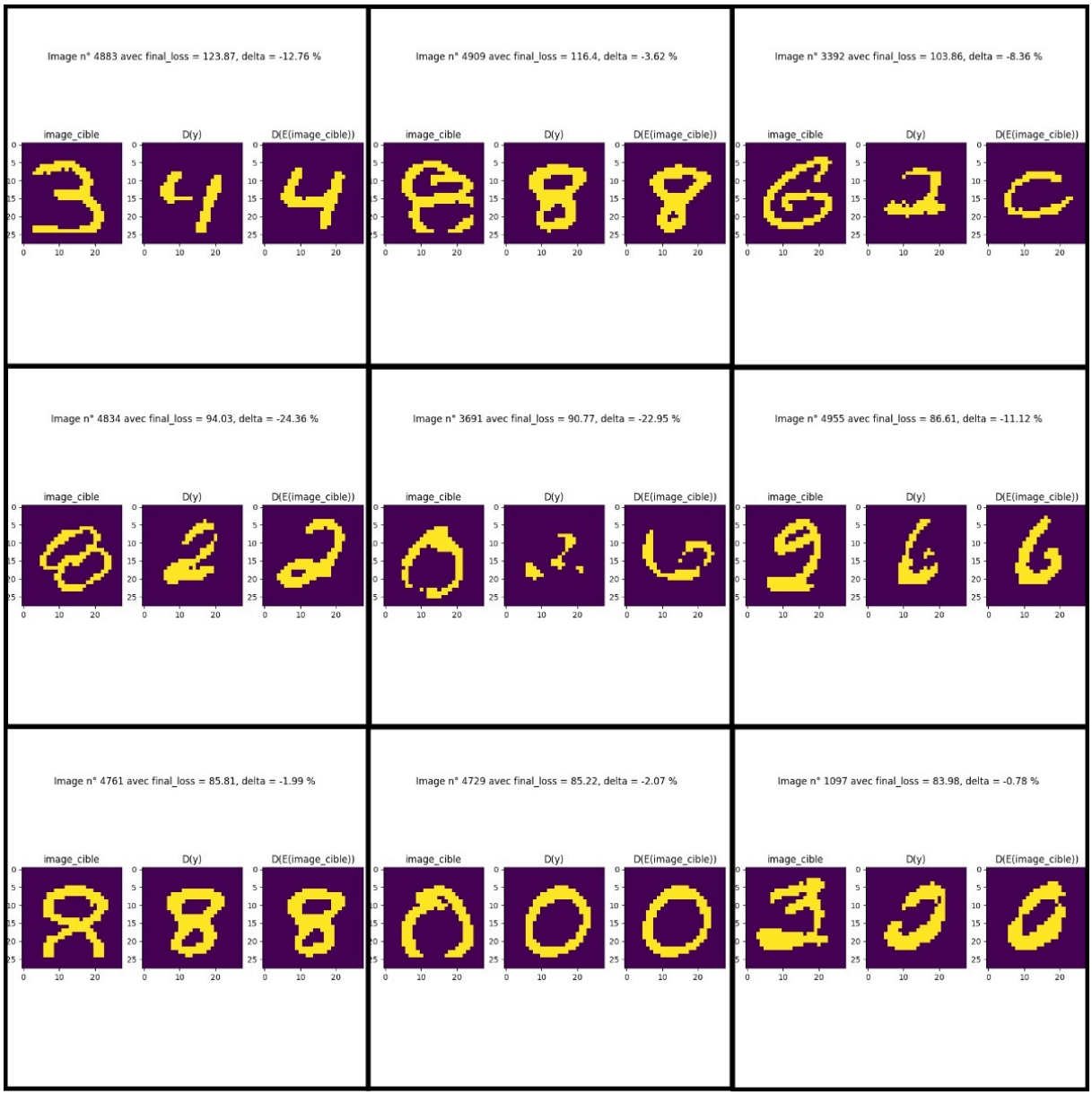


Figure 13: Top 9 des pires erreurs

On constate qu'il n'y a pas nécessairement de chiffre en particulier difficile à prédire par le modèle. La difficulté viendrait hypothétiquement de la nature atypique d'une calligraphie.

Après avoir constaté un corrélation entre les plus petites erreurs et le chiffre 1, nous allons aussi nous intéresser à une corrélation entre deux variables: le delta et l'erreur.

```
correlation = np.corrcoef(deltas, final_errors)[0, 1]
Correlation entre deltas et final_errors : 0.17920843690471547
Il n existe pas de correlation forte entre deltas et final_errors
```

3.4 Taux de réussite

L'analyse de la métrique *delta* soulève une question importante : une amélioration moyenne de 5% sur la fonction de perte signifie-t-elle que **MN** est un espace latent supérieur à **MT** ?
Pour répondre à cette interrogation, nous devons examiner le taux de réussite de chaque espace latent en comparant directement les points décodés de l'espace latent (**MT** ou **MN**) avec les images respectives du dataset MNIST. Cela implique de vérifier pour chaque image si le chiffre prédit par le point décodé correspond au chiffre réel de l'image.

Ce code illustre la création et l'utilisation d'un modèle de réseau de neurones convolutifs (CNN) pour classifier des images de chiffres manuscrits, typiquement des données comme celles du dataset MNIST. Le modèle, nommé **model_bis**, est structuré en plusieurs couches, incluant des couches de convolution pour extraire des caractéristiques des images, des couches de pooling pour réduire la dimensionnalité, et des couches denses pour la classification finale. L'activation softmax dans la dernière couche permet de produire une distribution de probabilité sur les 10 classes possibles de chiffres (0 à 9).

```
# Creation du  modele sequentiel
model_bis = models.Sequential([
  layers.Reshape((28, 28, 1), input_shape=(28, 28)),
  layers.Conv2D(32, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(64, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(64, (3, 3), activation='relu'),
  layers.Flatten(),
  layers.Dense(64, activation='relu'),
  layers.Dense(10, activation='softmax')
])

model_bis.compile(optimizer='adam', loss='
    sparse_categorical_crossentropy', metrics=['accuracy'])

model_bis.summary()

model_bis.fit(train_images, train_labels, epochs=10, validation_data
    =(test_images, test_labels))
```

Le modèle est compilé avec l'optimiseur **adam** et utilise la perte **sparse_categorical_crossentropy**, adaptée aux problèmes de classification où les classes sont représentées comme des entiers. La métrique *accuracy* est utilisée pour évaluer la performance du modèle pendant l'entraînement et les tests.

L'entraînement est effectué sur les *train_images* avec leurs étiquettes correspondantes *train_labels*, sur 10 *epochs*, et les données de validation *(test_images et test_labels)* sont utilisées pour évaluer le modèle après chaque *epoch* pour ajuster les paramètres de manière optimale et éviter le surajustement.

```
313/313 [==============================] - 4s 13ms/step - loss:
    0.0412 - accuracy: 0.9900
Test accuracy: 0.9900000095367432
```

Après l'entraînement, le modèle est évalué sur le jeu de test pour déterminer sa précision *(test_acc)*, qui donne une mesure quantitative de sa capacité à classer correctement les nouvelles images. La précision obtenue est affichée, permettant d'évaluer directement la performance du modèle.

```
MNIST_results = np.array(recup_da_floats(filepath_MNIST_results))
MT_results = np.array(recup_da_floats(filepath_MT_results))
MN_results = np.array(recup_da_floats(filepath_MN_results))

def accuracy(MNIST, MODEL):
  accuracy = []
  for i in range(len(MODEL)):
    if (MNIST[i] == MODEL[i]):
      accuracy.append(1)
    else:
      accuracy.append(0)

  print(f"Your model was right {sum(accuracy)} out of {len(MODEL)}
     images (ie. {sum(accuracy)/len(MODEL)} %)")
  return accuracy

if __name__ == "__main__":
  accuracy_MT = accuracy(MNIST_results, MT_results)
  accuracy_MN = accuracy(MNIST_results, MN_results)
```

Après avoir enregistré les correspondances entre les images et les chiffres pour le dataset MNIST et pour les points des espaces latents **MT** et **MN** décodés (voir l'annexe pour plus de détails), nous utilisons la fonction *accuracy* pour comparer ces correspondances. Cette fonction évalue si chaque image du dataset MNIST correspond au chiffre prédit par les modèles des espaces latents. Si oui, la prédiction est considérée comme correcte et reçoit une valeur de 1, sinon elle reçoit une valeur de 0.

Voici les résultats obtenus :

```
Model MT was right 4176 out of 5000 images (ie. 0.8352 %)

Model MN was right 4304 out of 5000 images (ie. 0.8608 %)
```

Nous avons donc amélioré le taux de réussite de 2.56%. Ce résultat montre de manière objective l'amélioration entre l'ancien **MT** et le nouvel **MN** espace latent.

3.5 Interprétation

À ce stade, il peut être intéressant de voir si les points de l'espace latent fournis par Pr. Turinici ainsi que ceux obtenus par la descente de gradient suivent une distribution gaussienne. Pour ce faire nous avons réalisé deux tests non asymptotiques: le test de Kolmogorov-Smirnov et de Shapiro-Wilk.

3.5.1 Kolmogorov-Smirnov

On observe $X_1^n = (X_1, \ldots, X_n)$ un n-échantillon de fonction de répartition F. On note F_0 une fonction de répartition de référence (dans notre cas celle d'une loi normale dont les paramètres sont évalués à partir des observations).
On veut tester l'hypothèse nulle $H_0 : F = F_0$ contre l'hypothèse alternative $H_1 : F \neq F_0$.
La statistique de test est donc donnée par:

$$h_n(X_1^n, F_0) = \|\hat{F}_n - F_0\|_\infty,$$

où $\hat{F}_n(x) = \frac{1}{n}\sum_{i=1}^n \mathbf{1}_{X_i \leq x}$.

Le test est défini par:

$$\phi_\alpha(X_1^n) = \mathbf{1}_{h_n(X_1^n, F_0) \geq \mathrm{q}_{n,1-\alpha}}$$

où $\mathrm{q}_{n,1-\alpha}$ est le quantile d'ordre $1-\alpha$ de h_n sous H_0.
Lorsque F_0 est continue, le test $\phi_\alpha(X_1^n)$ est de taille α (c'est le cas ici).
Introduisons maintenant notre code et nos résultats:

```python
from scipy.stats import kstest
import numpy as np

def ks_test_normality_2d(data):
    data_array = np.array(data)

    # Projeter les donnees sur un seul axe
    projected_data = np.sqrt(data_array[:, 0]**2 + data_array[:,
        1]**2)

    stat, p_value = kstest(projected_data, 'norm', args=(np.mean(
        projected_data), np.std(projected_data)))

    return stat, p_value


stat_bis, p_value_bis = ks_test_normality_2d(solutions)

print(f"Statistique du test : {stat_bis}")
print(f"p-value : {p_value_bis}")

# Interpretation du test
alpha = 0.05
if p_value_bis > alpha:
    print("L echantillon semble provenir d une distribution normale (
        non rejet de H0)")
```

```
else:
    print("L echantillon ne semble pas provenir d une distribution
       normale (rejet de H0)")
```

```
Statistique du test : 0.059096399885945616
p-value : 1.275118051866383e-15
L echantillon ne semble pas provenir d une distribution normale
(rejet de H0)
```

NB : Bien qu'étant un test adapté à des données unidimensionnelles, nous avons quand même pu l'appliquer ici en projetant nos données de dimension 2 sur un seul axe en calculant la distance euclidienne à partir de l'origine pour chaque point.

3.5.2 Shapiro-Wilk

On observe $X_1^n = (X_1, ..., X_n)$ un n-échantillon. On veut tester H_0 : "la distribution de l'échantillon suit une distribution normale" contre H_1 : "la distribution de l'échantillon ne suit pas une distribution normale". La statistique de test est:

$$W = \frac{\left(\sum_{i=1}^{n} a_i x(i)\right)^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

où les $x(i)$ sont les observations en ordre croissant, $\bar{x}$ est la moyenne de l'échantillon, et les a_i sont donnés par:

$$(a_1, \ldots, a_n) = \frac{m^T V^{-1}}{\sqrt{(m^T V^{-1} V^{-1} m)}}$$

où $m = (m_1, \ldots, m_n)$ et $(m_1, \ldots, m_n)$ sont les espérances des statistiques d'ordre d'un échantillon de variables i.i.d suivant une loi normale, et V est la matrice de variance-covariance de ces statistiques d'ordre. Introduisons maintenant notre code et nos résultats :

```
from scipy.stats import shapiro

def shapiro_test(data):
    # Conversion de la liste de sous-listes en un tableau numpy
    data_array = np.array(data)

    # Effectuer le test de Shapiro-Wilk
    stat, p_value = shapiro(data_array)

    return stat, p_value

stat, p_value = shapiro_test(solutions)

print(f"Statistique du test : {stat}")
print(f"p-value : {p_value}")

# Interpretation du test
```

```
alpha = 0.05
if p_value > alpha:
    print("L echantillon semble provenir d une distribution normale (
        non rejet de H0)")
else:
    print("L echantillon ne semble pas provenir d une distribution
        normale (rejet de H0)")
```

```
Statistique du test : 0.9963952898979187
p-value : 5.3793834282347096e-15
L echantillon ne semble pas provenir d une distribution normale
(rejet de H0)
```

3.5.3 Conclusion des tests

Les p-valeurs obtenues dans les deux tests sont suffisamment basses pour être confiant dans le fait que notre nouvel espace latent ne suit pas une distribution gaussienne de dimension 2. À noter que nous avons effectué ces tests sur l'espace latent **MT**; les résultats sont pratiquement similaires et les mêmes conclusions sont donc tirées.

La prochaine étape serait de généraliser ce processus d'inversion de la fonction décodeur D, nous permettant de nous débarrasser de l'encodeur E. À ce stade, une piste envisageable serait d'utiliser un réseau de neurones afin d'effectuer cette généralisation.

3.6 Etude supplémentaire : généralisation de l'inversion de D

3.6.1 Modèle bonus

Le modèle de réseau de neurones est construit en utilisant la fonction **build_model**, qui crée un modèle séquentiel avec une couche d'entrée correspondant à la dimension de l'espace latent. Deux couches cachées densément connectées suivent, chacune avec 64 unités et utilisant la fonction d'activation ReLU. La dernière couche est une couche dense qui renvoie une sortie de la même taille que l'entrée, permettant ainsi au modèle de prédire les points dans l'espace latent. Le but de ce modèle est de capter une tendance dans la transformation planaire entre les espaces latents **MT** et **MN**.

```
MT_space = np.array(reparam_subset)
MN_space = np.array(solutions)

def build_model(latent_dim):
    model = models.Sequential()
    model.add(layers.Input(shape=(latent_dim,)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(latent_dim))
    return model


```

```
latent_dim = len(MT_space[0])
model_bonus = build_model(latent_dim)

model_bonus.compile(optimizer='adam', loss='mse')
model_bonus.fit(MT_space, MN_space, epochs=100, batch_size=32,
    validation_split=0.2)

# Example
mnist_index = 55
image_cible = train_images[mnist_index,:,:,0]
new_point = tf.squeeze(tf.Variable(model_bonus.predict(np.array([
    encoder_function(image_cible)]))))

# D(model_bonus(E(image_cible)))
bonus_image = decoder_function(new_point).numpy()
```

Le modèle **model_bonus** (aussi noté **MB**) est ensuite compilé avec l'optimiseur *Adam* et la fonction de perte MSE (Mean Squared Error), qui est typique pour des problèmes de régression comme la prédiction de valeurs continues.

Le modèle est entraîné pour 100 *epochs* avec une taille de lot de 32 et une division de validation de 20% pour évaluer la performance du modèle sur des données non vues pendant l'entraînement.

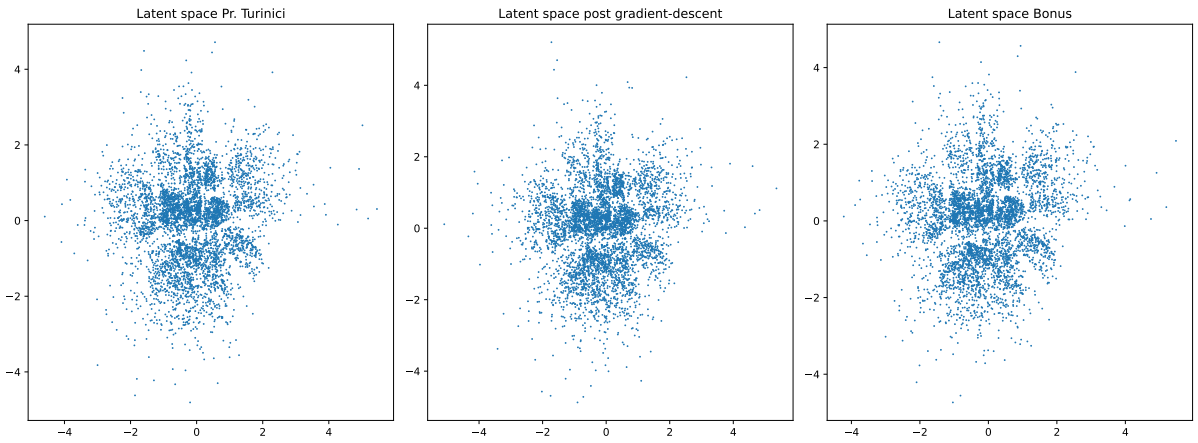
Données MB :
La moyenne vectorielle μ_{MB} est donnée par :

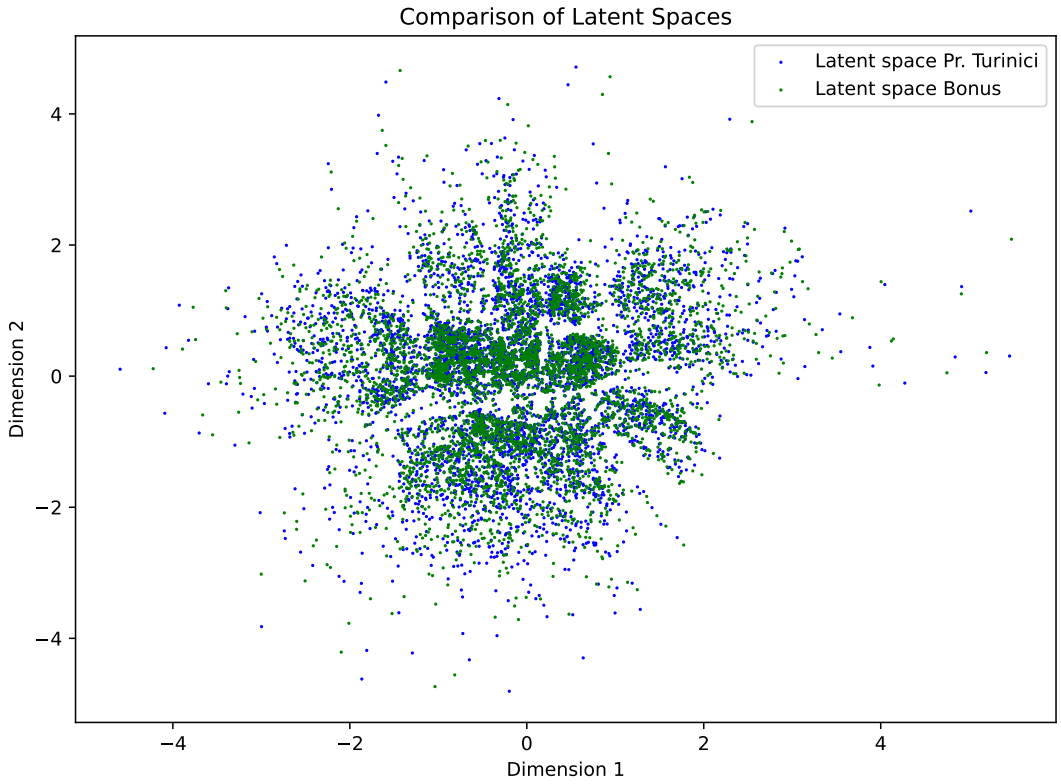
$$\mu_{MB} = \begin{bmatrix} -0.123 & 0.0281 \end{bmatrix}$$

La matrice de variance-covariance est donnée par :

$$\text{var\_covar}_{MB} = \begin{pmatrix} 1.19 & 0.136 \\ 0.136 & 1.33 \end{pmatrix}$$

NB: Pour comparer avec les autres espaces latents, voir tableau récapitulatif des moyennes et des matrices de variance-covariance.





Nous avons aussi calculé les distances entre les points de l'espace **MT** et de l'espace **MB** associés à l'image n°i. Ces 5 000 distances sont stockées dans un tableau dont on calcule la moyenne et la variance.

```
Mean of distance between MT & MB :  0.0676884800195694
Variance of distance between MT & MB: 0.004237209912389517
```

Conclusion partielle : les données sont proches et se ressemblent encore plus qu'entre **MT** et **MN**. Il faudrait regarder aussi les deltas, en calculant la variation entre la valeur de départ (encodeur puis décodeur) et la valeur d'arrivée (encoder puis appliquer le **model_bonus** et ensuite décoder) de la loss function pour chaque image. On s'attendrait à un résultat intermédiaire, moins efficace que **MN**, mais plus efficace que **MT**.

Soit δ_{MB} le vecteur qui représente cette variation départ-arrivée: $\forall i \in \{0, 1, \ldots, 4999\}$,

$$(\delta_{\text{MB}})_i = \left(\frac{\|z_i - D(MB_i)\|_2^2 - \|z_i - D(MT_i)\|_2^2}{\|z_i - D(MT_i)\|_2^2} \right) \times 100$$

où:
$z_i \in \mathbb{R}^{784}$ est la i-ème image du dataset,
$MB_i \in \mathbb{R}^2$ le i-ème point de l'espace latent MB,
$MT_i \in \mathbb{R}^2$ le i-ème point de l'espace latent MT.

La moyenne $\mu_{\delta_{MB}}$ est donnée par :

$$\mu_{\delta_{MB}} = 0.12$$

La variance est donnée par :

$$\text{var}_{\delta_{MB}} = 10.83$$

NB: Pour comparer avec δ_{MN}, voir le tableau récapitulatif de δ_{MN} et δ_{MB}.

Conclusion : La stupeur est totale. Nous avons en moyenne perdu en précision de 0.12 ‰. Cela signifierait donc que notre idée de généraliser l'inversion de D avec des réseaux de neurones pourrait s'avérer obselète. Or, il est important de rappeler que l'entrainement a eu lieu sur 5 000 images. Réaliser l'entrainement du **model_bonus** sur l'entiereté du dataset MNIST pourrait hypothétiquement faire fonctionner notre modèle. Pour ce faire, nous devrions appeler **gradient_descent** 60 000 fois pour avoir notre liste *solutions* — contenant les points de l'espace latent **MN** — complète. Là encore se pose le problème du temps d'exécution...

NB : À raison de 1,2 seconde par appel de **gradient descent** *pour 60 000 images, cela donnerait un total de 72 000 secondes, soit 1 200 minutes, soit 20 heures.*

3.6.2 Taux de réussite du modèle bonus

Cependant, rappelons que la métrique δ_{MB} ne nous informe pas si le modèle **MB** prédit mieux les images MNIST que **MT**. Seule l'utilisation de la fonction *accuracy*, introduite dans la partie taux de réussite, nous permet de savoir lequel des modèles est supérieur.

Voici les résultats obtenus :

```
MT was right 4176 out of 5000 images (ie. 0.8352 %)

MB was right 4184 out of 5000 images (ie. 0.8368 %)
```

Conclusion : Malgré l'interrogation suscitée par la moyenne de δ_{MB}, on constate une infime amélioration entre **MT** et **MB**.

3.6.3 Analyse Vectorielle entre les Espaces Latents MT et MN

Essayons de voir s'il y a une tendance entre les deux espaces latents **MT** et **MN**. La motivation derrière cette initiative est de voir si **model_bonus** peut capter des informations sur la transormation entre un point de **MT** et en un point de **MN**.

Rappelons que :

$$\mu_{MT} = \begin{bmatrix} -0.123 & 0.0281 \end{bmatrix}$$

Considérons un espace où chaque point i, avec $i = 1, \ldots, N$, est représenté par deux vecteurs :

- Le vecteur de translation $\vec{v}_i = \text{MN}_i - \text{MT}_i$, qui décrit le déplacement du point i entre l'espace latent **MT** et **MN**.
- Le vecteur $\vec{u} = \mu_{\text{MT}} - \text{MT}_i$.

Le produit scalaire entre $\vec{u}$ et chaque $\vec{v}_i$ est donné par :

$$\vec{u} \cdot \vec{v}_i := \|\vec{u}\| \|\vec{v}_i\| \cos(\theta_i)$$

où $\|\vec{u}\|$ et $\|\vec{v}_i\|$ désignent respectivement les normes des vecteurs $\vec{u}$ et $\vec{v}_i$ et θ_i est l'angle entre ces deux vecteurs.

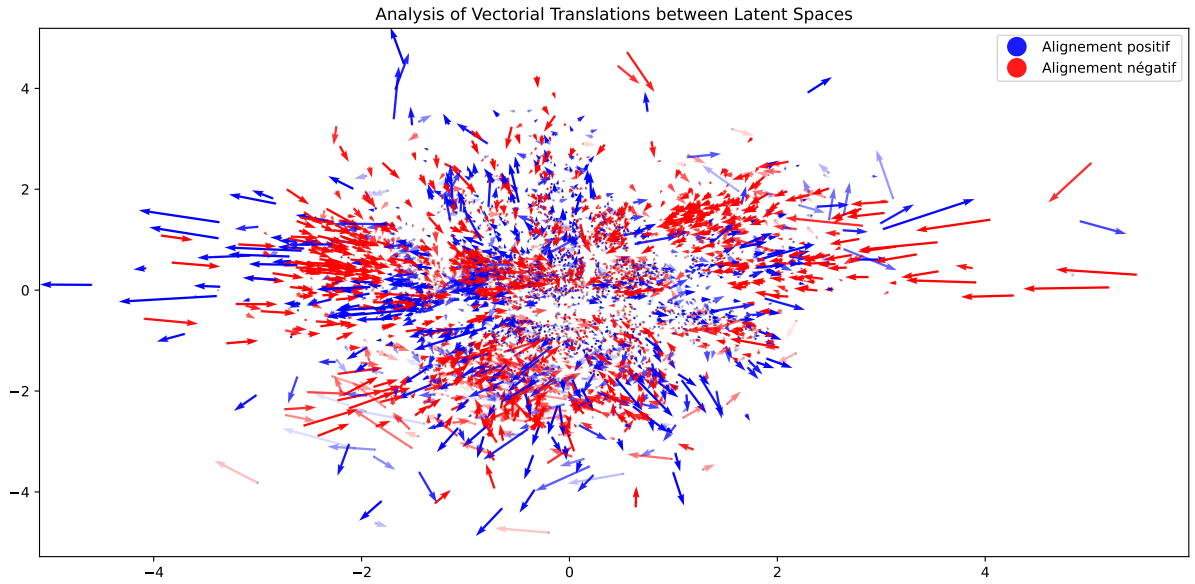


Figure 14: Les vecteurs de translation $\vec{v}_i = MN_i - MT_i$ selon leur produit scalaire $\vec{u} \cdot \vec{v}_i$

Selon le signe du produit scalaire $\vec{u} \cdot \vec{v}_i$, les vecteurs $\vec{v}_i$ sont colorés sur le graphe de la manière suivante :

- Si $\vec{u} \cdot \vec{v}_i > 0$, le vecteur $\vec{v}_i$ est coloré en bleu, indiquant un angle aigu ($\theta_i < 90°$) entre $\vec{u}$ et $\vec{v}_i$, montrant que les directions sont relativement alignées.
- Si $\vec{u} \cdot \vec{v}_i < 0$, le vecteur $\vec{v}_i$ est coloré en rouge, indiquant un angle obtus ($\theta_i > 90°$) entre $\vec{u}$ et $\vec{v}_i$, montrant que les directions sont relativement opposées.

L'opacité de chaque vecteur $\vec{v}_i$ sur le graphe est ajustée en fonction de la valeur absolue de $\vec{u} \cdot \vec{v}_i$ pour refléter la magnitude de l'alignement ou de l'opposition entre $\vec{u}$ et $\vec{v}_i$.

Conclusion : Il n'y a aucun tendance constatée car il y a du bleu et du rouge à peu près partout. Il est donc logique que l'on n'arrive pas à améliorer le modèle à travers une généralisation de l'inverse de D avec des réseaux de neurones.

Conclusion

Bien qu'étant un secteur en pleine expansion, l'IA possède encore de nombreux axes de développement et de progression, pouvant être de très bon augure pour le futur. L'étude approfondie de ce mémoire concernant l'inversion du décodeur via la méthode de la descente de gradient a non seulement permis de mieux comprendre les processus sous-jacents mais a aussi apporté des réponses pratiques afin d'améliorer la précision et l'efficacité des auto-encodeurs.

L'implémentation pratique a démontré une capacité à reconstruire avec le plus de fidélité des images cibles, tout en réduisant la perte d'information inévitable et typique des processus de compression et de décompression. Ces résultats ouvrent la voie à de nouvelles applications potentielles dans des domaines divers et variés. Prenons par exemple le monde de la santé, où une grande précision des images reconstruites est cruciale pour le bien être du patient. Ainsi, cela suggère de nombreuses directions prometteuses pour de futures recherches, et ce mémoire pourrait même être encore plus approfondi pour aller plus loin dans l'optimisation du modèle. Il a été très important de prendre en considération le temps d'exécution qui a souvent été un frein dans l'évolution de nos recherches, nous contraignant à réduire considérablement la quantité d'images du dataset d'entraînement.

Une piste d'amélioration pourrait être d'explorer une autre méthode pouvant inverser le décodeur tout en étant moins couteuse en temps d'exécution, ou plus efficace en taux de convergence. Nous pouvons mentionner par exemple la méthode de quasi-Newton, BFGS (Broyden-Fletcher-Goldfarb-Shanno) permettant de résoudre un problème d'optimisation non linéaire sans contraintes, tout en ayant, à priori un meilleur taux de convergence que la descente de gradient que nous avons implémenté.

Bibliographie

[1] Daphné Wallach: *Deep Learning pour le traitement d'images. Classification, détection et segmentation avec Python et TensorFlow.*

[2] Gabriel Turinici: *Deep Learning.* Polycopié de l'Université Paris Dauphine, M2 ISF, 2024
https://turinici.com/wp-content/uploads/cours/deep_learning/deep_learning_G_Turinici_v3_5.pdf

[3] Yating Liu: *Introduction à l'apprentissage statistique.* Polycopié de l'Université Paris Dauphine, M1 Maths, 2023

[4] Idriss Mazari: *Méthodes numériques et optimisation.* Polycopié de l'Université Paris Dauphine, L3 Maths, 2023

[5] Documentation TensorFlow:
https://www.tensorflow.org/tutorials?hl=fr

[6] Gabriel Turinici:
https://github.com/gabriel-turinici/

[7] Site internet:
https://www.coursera.org/specializations/deep-learning

[8] Site internet:
https://stackoverflow.com/questions/42966393/is-it-good-learning-rate-for-adam-meth

[9] Documentation TensorFlow:
https://www.tensorflow.org/api_docs/python/tf/keras/optimizers

Annexes

A. Enregistrement des données

```
# Enregistre une liste de listes de float, chaque i-eme ligne sera la
    i-eme sous ligne

def create_list_sub_txt(huge_list, filepath = ''):
  with open(filepath, 'w') as file:
      for sublist in huge_list:
          file.write('[' + ', '.join(map(str, sublist)) + ']\n')

# Enregistre une liste simple de float, separes par des virgules dans
    le fichier

def create_list_txt(simple_list, filepath = ''):
  with open(filepath, 'w') as file:
      file.write(', '.join(map(str, simple_list)) + '\n')
```

B. Récupération des données

```
def recup_solutions(filepath = ''):
  with open(filepath, 'r') as file:
      solutions = [ast.literal_eval(line.strip()) for line in file]
      return solutions

def recup_da_floats(filepath):
    with open(filepath, 'r') as file:
        data = file.read().strip().split(',')
        float_list = [float(x) for x in data]
    return float_list

filepath_final_errors = '/content/drive/My Drive/Thesis/data/
   final_errors.txt'
filepath_deltas = '/content/drive/My Drive/Thesis/data/deltas.txt'
filepath_solutions = '/content/drive/My Drive/Thesis/data/solutions.
   txt'

final_errors = recup_da_floats(filepath_final_errors)
deltas = recup_da_floats(filepath_deltas)
solutions = recup_solutions(filepath_solutions)
```

C. Comparaison des images

```
def comparison_images(mnist_index, file_path_comparison = '', save =
    False):

  # Conversion pour appeler correctement le decodeur
  image_cible = train_images[mnist_index,:,:,0]
  y = tf.Variable(solutions[mnist_index])

  plt.figure()
  plt.figure(constrained_layout=True)

  ##################################################
  ##################################################
  ##################################################

  plt.subplot(1,4,1)
  # On plot la vraie image

  plt.imshow(image_cible)
  plt.title("image_cible")

  ##################################################
  ##################################################
  ##################################################

  plt.subplot(1,4,2)
  dec_function_np = decoder_function(y)

  # On plot D(solution trouvee avec la ddg)

  plt.imshow(np.where(dec_function_np > .5, 1.0, 0.0))
  plt.title("D(y)")

  ##################################################
  ##################################################
  ##################################################

  plt.subplot(1,4,3)
  dec_enc_image_cible = decoder_function(encoder_function(image_cible
     )).numpy()

  # On plot D(E(vraie image))

  plt.imshow(np.where(dec_enc_image_cible > .5, 1.0, 0.0))
  plt.title("D(E(.))")

  ##################################################
  ##################################################
  ##################################################

```

```
  plt.subplot(1,4,4)

  new_point = tf.squeeze(tf.Variable(model_bonus.predict(np.array([
      encoder_function(image_cible)]))))
  bonus = decoder_function(new_point).numpy()

  # On plot D(model_bonus(E(vraie image)))

  plt.imshow(np.where(bonus > .5, 1.0, 0.0))
  plt.title("D(model_bonus(E(.)))")

  ################################################
  ################################################
  ################################################

  plt.suptitle(f"Image numero {mnist_index} avec final_loss = {round(
      final_errors[mnist_index], 2)}, delta = {round(deltas[
      mnist_index], 2)} %")

  if save == True:
    plt.savefig(file_path_comparison)

  plt.show()

  if save == True:
    files.download(file_path_comparison)

if __name__ == "__main__":
  mnist_index = 55
  file_name_comparison = f"comparison_image_{mnist_index}.pdf"
  file_path_comparison = f"/content/drive/My Drive/Thesis/image_{
      mnist_index}/{file_name_comparison}"
  save = False # True si on veut enregistrer l image
  comparison_images(mnist_index, file_path_comparison, save)
```

D. Taux de réussite

```
(train_images, train_labels), (test_images, test_labels) = tf.keras.
    datasets.mnist.load_data()

def preprocess_images(images):
  images = images.reshape((images.shape[0], 28, 28, 1)) / 255.
  return np.where(images > .5, 1.0, 0.0).astype('float32')

train_images = preprocess_images(train_images)
test_images = preprocess_images(test_images)
```

```
def decoding_filtered(latent_space): #format adjustement
  list_decoded = []
  for point in tqdm(latent_space):
    y = tf.Variable(point)
    # decoding latent space point
    new_image = decoder_function(y)
    list_decoded.append(np.where(np.array(new_image) > .5, 1.0, 0.0))
  return np.array(list_decoded)

def associate_image_to_number(predictions):
  result = []
  for image in tqdm(predictions):
    predicted_number = np.argmax(image)
    result.append(predicted_number)
  return result

if __name__ == "__main__":
  # Format adjustment & decoding latent space
  MT_images = decoding_filtered(reparam[:5000]).reshape(5000, 28, 28,
      1)
  MN_images = decoding_filtered(solutions).reshape(5000, 28, 28, 1)
  MB_images = decoding_filtered(bonus).reshape(5000, 28, 28, 1)

  # Apply model_bis
  predictions_MT_dataset = model_bis.predict(MT_images)
  predictions_MN_dataset = model_bis.predict(MN_images)
  predictions_MB_dataset = model_bis.predict(MB_images)
  predictions_mnist_dataset = model_bis.predict(train_images)

  # Determine each number corresponding to each image
  MNIST_results = associate_image_to_number(predictions_mnist_dataset
     )
  MT_results = associate_image_to_number(predictions_MT_dataset)
  MN_results = associate_image_to_number(predictions_MN_dataset)
  MB_results = associate_image_to_number(predictions_MB_dataset)

  filepath_MNIST_results = '/content/drive/My Drive/Thesis/data/
     MNIST_results.txt'
  filepath_MT_results = '/content/drive/My Drive/Thesis/data/
```

```
    MT_results.txt'
filepath_MN_results = '/content/drive/My Drive/Thesis/data/
    MN_results.txt'
filepath_MB_results = '/content/drive/My Drive/Thesis/data/
    MB_results.txt'

# Save the results
create_list_txt(MNIST_results, filepath_MNIST_results)
create_list_txt(MT_results, filepath_MT_results)
create_list_txt(MN_results, filepath_MN_results)
create_list_txt(MB_results, filepath_MB_results)
```

E. Récapitulatif des données pour chaque espace latent

Espace Latent	**Moyenne Vectorielle** (μ)	**Matrice de Variance-Covariance** (var_covar)
MT	$\begin{bmatrix} -0.123 & 0.0281 \end{bmatrix}$	$\begin{pmatrix} 1.16 & 0.092 \\ 0.092 & 1.38 \end{pmatrix}$
MN	$\begin{bmatrix} -0.129 & 0.0175 \end{bmatrix}$	$\begin{pmatrix} 1.14 & 0.097 \\ 0.097 & 1.36 \end{pmatrix}$
MB	$\begin{bmatrix} -0.123 & 0.0281 \end{bmatrix}$	$\begin{pmatrix} 1.19 & 0.136 \\ 0.136 & 1.33 \end{pmatrix}$

Table 1: Récapitulatif des moyennes vectorielles et des matrices de variance-covariance

F. Récapitulatif des gains

Espace Latent	**Moyenne** (μ)	**Variance** (var)
δ_{MN}	−4.79	41.91
δ_{MB}	0.12	10.83

Table 2: Récapitulatif des moyennes et variances