UE 455 TP 3 Théorie de l'Information - Codage de Source

SHI Wenli
XU Kaiyuan

30 Avril 2024

Table des matières

1 Objectif

La compression avec perte implique un compromis entre débit et distorsion. Ce TP implémente un modèle similaire à un autoencodeur pour compresser des images du jeu de données MNIST. La méthode est basée sur l'article End-to-end Optimized Image Compression.

Plus de détails sur la learned data compression peuvent être trouvés dans cet article ciblé pour les personnes familières avec la compression de données classique, ou dans cette enquête ciblée pour un public orienté apprentissage automatique.

2 Définir le modèle de l'entrainement

Nous allons considérer un modèle de type autoencodeur qui se compose de trois parties :
— la transformation d'analyse (ou encodeur), convertissant l'image en vecteur dans un espace latent,
— la transformation de synthèse (ou décodeur), convertissant le vecteur de l'espace latent en espace image, et
— unmodèle pour les distributions de probabilités *a priori* marginales des vecteurs latents, utile pour le calcul de l'entropie.

Listing 1 – Lecture de texte

```
def make_analysis_transform(latent_dims):
  """Creates the analysis (encoder) transform."""

  return tf.keras.Sequential([
    tf.keras.layers.Conv2D(20, 5, use_bias=True, strides=2, padding="same",
      activation="leaky_relu", name="conv_1"),
    tf.keras.layers.Conv2D(50, 5, use_bias=True, strides=2, padding="same",
      activation="leaky_relu", name="conv_2"),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(500, use_bias=True, activation="leaky_relu", name="fc_1"),
    tf.keras.layers.Dense(latent_dims, use_bias=True, activation=None, name="fc_2"),
  ], name="analysis_transform")

def make_synthesis_transform():
  """Creates the synthesis (decoder) transform."""

  return tf.keras.Sequential([
    tf.keras.layers.Dense(500, use_bias=True, activation="leaky_relu", name="fc_1"),
    tf.keras.layers.Dense(2450, use_bias=True, activation="leaky_relu", name="fc_2"),
    tf.keras.layers.Reshape((7, 7, 50)),
    tf.keras.layers.Conv2DTranspose(20, 5, use_bias=True, strides=2, padding="same",
      activation="leaky_relu", name="conv_1"),
    tf.keras.layers.Conv2DTranspose(1, 5, use_bias=True, strides=2, padding="same",
      activation="leaky_relu", name="conv_2"),
  ], name="synthesis_transform")
```

Manipulation: 1 Donner la structure des réseaux réalisant l'analyse et la synthèse.

À partir du code ci-dessus, on peut voir la structure du réseau d'analyse et du réseau de synthèse.

On analyse d'abord le réseau de codage. La première est constituée de deux couches convolutives bidimensionnelles. La première couche convolutive a 20 filtres, un noyau de convolution 5x5, est rempli de *same*, la foulée est de 2 et la fonction d'activation est *leaky_relu*. La deuxième couche convolutive comporte 50 filtres. Ensuite, il contient une couche *Flatten* pour aplatir la sortie de la couche convolutive en un tableau unidimensionnel. Enfin, il existe deux couches entièrement connectées *Dense*, la première comportant 500 unités. La seconde consiste à réduire les dimensions de sortie au nombre de *latent_dims*.

Le réseau de décodage peut être considéré comme la structure inverse du réseau de codage. Les premières sont constituées de deux couches *Dense*, contenant respectivement 500 et 2450 unités. Ensuite, il existe une couche *Reshape* pour reconstruire la sortie de la couche Dense dans un format tridimensionnel $7 \times 7 \times 50$ pour les opérations de convolution ultérieures. Enfin, il existe deux couches convolutives 2D et mappent les vecteurs sur l'espace des pixels de l'image.

Un classe pour l'entrainement est définie. Elle contient une instance des deux transformations, ainsi que les paramètres de distributions *a priori*.

Sa méthode `call` est configurée pour calculer :
— le débit, une estimation du nombre de bits nécessaires pour représenter un lot d'images représentant des chiffres, et
— la distorsion, la moyenne de la valeur absolue de la différence entre les pixels des chiffres originaux et leurs reconstructions.

2.1 Calcul du débit et de la distortion

Voyons cela étape par étape, en utilisant une image de l'ensemble de données d'entraînement. Chargez l'ensemble de données MNIST pour l'entraînement et la validation.

Manipulation: 1 Pour cela, la commande suivante peut être utilisée le code.

On peut utiliser les codes suivants pour charger l'ensemble de données MNIST pour les entrainement et les tests ultérieurs.

Listing 2 – Charge de MNIST

```
training_dataset, validation_dataset = tfds.load(
    "mnist",
    split=["train", "test"],
    shuffle_files=True,
    as_supervised=True,
    with_info=False,
)
```

Manipulation: 2 Quelle est la taille du l'ensemble d'entrainement `training_dataset`, de l'ensemble de test `validation_dataset` ?

La taille de l'ensemble d'entrainement est de 60 000 images et la taille de l'ensemble de test est de 10 000 images.

Manipulation: 3 Extrayez et affichez une image x.

Les images incluses dans l'ensemble de données MNIST sont des images de chiffres manuscrits et leur taille d'image est de 28×28.

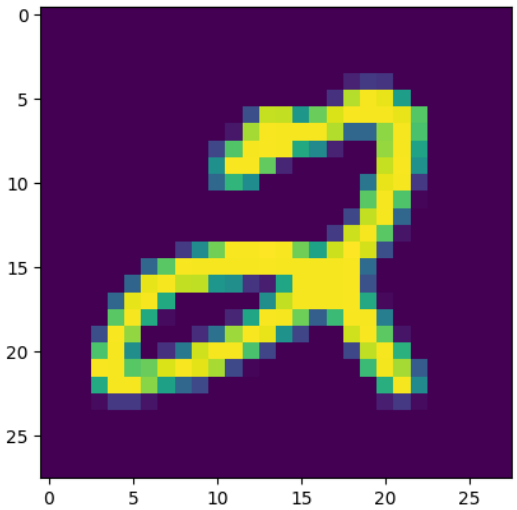

FIGURE 2.1.1 – Image

Listing 3 – Output de Console - Entropie

```
Data type: <dtype: 'uint8'>
Shape: (28, 28, 1)
```

Manipulation: 4 Pour obtenir la représentation latente y, nous devons convertir x en `float32`, ajouter une dimension afin de permettre un traitement en lot et la faire passer par la transformation d'analyse.

Quel est le type de la variable y ?

Le type de variable y obtenu après avoir utilisé le code suivant est un tenseur, où le type de données de chaque donnée est `float32`.

Listing 4 – Charge de MNIST

```
x = tf.cast(x, tf.float32) / 255.
x = tf.reshape(x, (-1, 28, 28, 1))
y = make_analysis_transform(10)(x)
```

Manipulation: 5 Les latents seront quantifiés au moment du test. Nous ajoutons un bruit uniforme dans l'intervalle $[-0.5, 0.5]$ et appelons le résultat $\tilde{y}$. Il s'agit de la même terminologie que celle utilisée dans l'article Compression d'image optimisée de bout en bout. Pourquoi réalise-t-on cette opération d'ajout de bruit ?

On ajoute un bruit uniformément réparti afin de simuler les erreurs de quantification pouvant survenir lors de la compression réelle. Dans le même temps, l'ajout de bruit à la représentation latente aide également le modèle entraîné à être plus robuste aux erreurs.

Listing 5 – Ajout de bruit

```
y_tilde = y + tf.random.uniform(y.shape, -.5, .5)
```

Manipulation: 6 L'*a priori* est une densité de probabilité utilisée pour modéliser la distribution marginale des éléments latents bruités. Par exemple, il peut s'agir d'un ensemble de distributions logistiques indépendantes avec des échelles différentes pour chaque dimension latente. La fonction `tfc.NoisyLogistic` tient compte du fait que les latents sont corrompus par un bruit additif. Lorsque l'échelle se rapproche de zéro, une distribution logistique se rapproche d'un delta de Dirac (pic), mais le bruit ajouté fait que la distribution bruitée se rapproche plutôt de la distribution uniforme sur $[-0.5, 0.5]$.

On peut voir sur la figure 2.1.2 que lorsque l'échelle s'approche de 0, le bruit ajouté amène la distribution du bruit à se rapprocher d'une distribution uniforme sur $[-0.5, 0.5]$. Lorsque l'échelle est plus grande, la distribution du bruit se rapproche de la distribution normale.

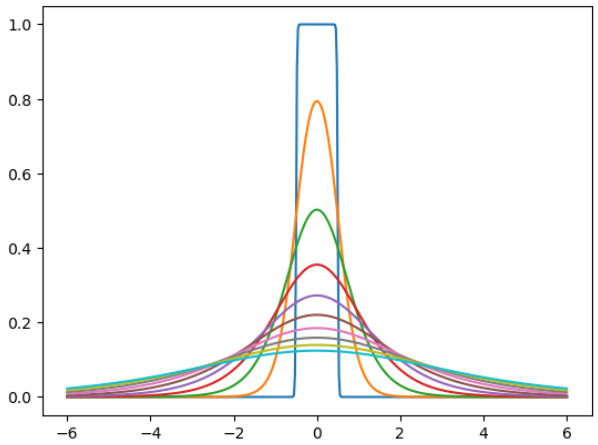

FIGURE 2.1.2 – Distribution bruitée.

Listing 6 – A prior

```
prior = tfc.NoisyLogistic(loc=0., scale=tf.linspace(.01, 2., 10))
_ = tf.linspace(-6., 6., 501)[:, None]
plt.plot(_, prior.prob(_));
```

Manipulation: 7 During training, `tfc.ContinuousBatchedEntropyModel` adds uniform noise, and uses the noise and the prior to compute a (differentiable) upper bound on the rate (the average number of bits necessary to encode the latent representation). That bound can be minimized as a loss.

Quel est le débit obtenu ?

Le modèle `tfc.ContinuousBatchedEntropyModel` ajoute un bruit uniforme à la représentation latente et calcule le débit binaire de codage en fonction de ce bruit et d'un modèle de probabilité a priori. Pendant la formation, le modèle calcule une limite supérieure sur le débit binaire de la représentation potentielle.

Le débit calculé par ce code est de $18,56$ bits/symbole. Cette limite supérieure peut être minimisée en tant que fonction de perte pendant l'entraînement.

Listing 7 – Bound

```
entropy_model = tfc.ContinuousBatchedEntropyModel(prior, coding_rank=1, compression=
    False)

y_tilde, rate = entropy_model(y, training=True)
```

Manipulation: 8 Enfin, les latents bruités sont repassés par la transformée de synthèse pour produire une reconstruction d'image $\tilde{x}$. La distorsion est l'erreur entre l'image originale et la reconstruction.

Pourquoi la reconstruction est-elle de mauvaise qualité ?

Pour le moment, on n'a pas entraîné le modèle, donc la reconstruction directe du `y_tilde` ne peut pas obtenir un bon résultat.

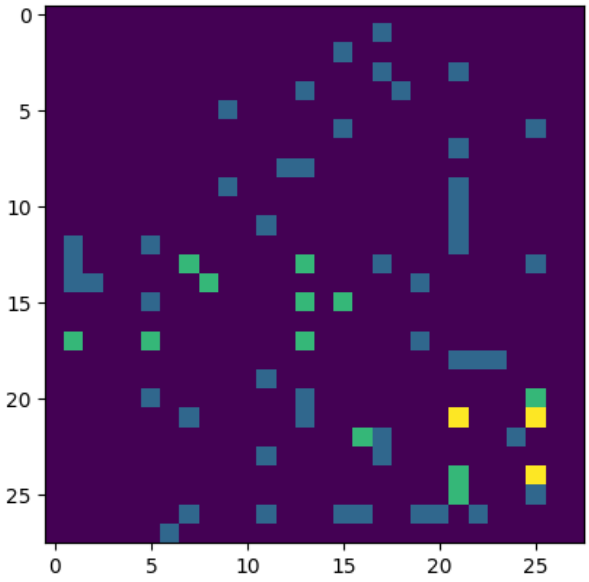

FIGURE 2.1.3 – Image reconstruit

Listing 8 – Output de Console - Entropie

```
distortion: tf.Tensor(0.17085683, shape=(), dtype=float32)
Data type: <dtype: 'uint8'>
Shape: (28, 28, 1)
```

Listing 9 – x_tilde

```
x_tilde = make_synthesis_transform()(y_tilde)

# Mean absolute difference across pixels.
distortion = tf.reduce_mean(abs(x- x_tilde))
print("distortion:", distortion)

x_tilde = tf.saturate_cast(x_tilde[0] * 255, tf.uint8)
plt.imshow(tf.squeeze(x_tilde))
```

Pour chaque lot d'images, un appel de `MNISTCompressionTrainer` produit un débit moyen et une distorsion moyenne évalués sur le lot.

Listing 10 – lot

```
(example_batch, _), = validation_dataset.batch(32).take(1)
trainer = MNISTCompressionTrainer(10)
example_output = trainer(example_batch)

print("rate: ", example_output["rate"])
print("distortion: ", example_output["distortion"])
```

Manipulation: 1 Quelle est la taille du lot (batch) utilisé dans cet exemple ?

D'après le code précédent, on peut savoir que la taille du lot dans cet exemple est de 32. Cela signifie que lors du traitement, 32 images sont traitées par lot à chaque fois.

Manipulation: 2 Quel est le rôle de l'argument 10 de `MNISTCompressionTrainer` ?

Selon le code précédent, ce paramètre spécifie la dimension de la représentation latente dans le modèle d'auto-encodeur. Dans cet exemple, `latent_dims=10` signifie que le modèle encodera chaque image dans un vecteur latent à 10 dimensions.

2.2 Entrainement du modèle

Nous compilons le formateur de manière à optimiser le Lagrangien débit-distorsion, c'est-à-dire une somme du débit et de la distorsion, où l'un des termes est pondéré par le multiplieur de Lagrange.

Cette fonction de perte affecte différemment les différentes parties du modèle :

— La transformation d'analyse est entraînée à produire une représentation latente qui atteint le compromis souhaité entre le taux et la distorsion.

— Latransformation de synthèse est entraînée à minimiser la distorsion, étant donné la représentation latente.

— Les paramètres de la distribution a priori sont entraînés à minimiser le taux étant donné la représentation latente. Cela revient à adapter la distribution a priori à la distribution marginale des latents dans un sens de maximum de vraisemblance.

Listing 11 – Trainer

```
def pass_through_loss(_, x):
    # Since rate and distortion are unsupervised, the loss doesn't need a target.
    return x

def make_mnist_compression_trainer(lmbda, latent_dims=50):
    trainer = MNISTCompressionTrainer(latent_dims)
    trainer.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),

        # Just pass through rate and distortion as losses/metrics.
        loss=dict(rate=pass_through_loss, distortion=pass_through_loss),
        metrics=dict(rate=pass_through_loss, distortion=pass_through_loss),
        loss_weights=dict(rate=1., distortion=lmbda),
```

```
    )
    return trainer
```

Ensuite, nous entraînons le modèle. Les annotations humaines (le chiffre correspondant à chaque image) ne sont pas nécessaires ici, puisque nous voulons simplement compresser les images. Nous les supprimons donc en utilisant une `map` et nous ajoutons à la place des cibles factices pour le débit et la distorsion.

Pour l'entrainement, si vous n'avez pas accès à un GPU, vous pouvez réduire `epochs` à 5.

Listing 12 – Trainer

```
def add_rd_targets(image, label):
    # Training is unsupervised, so labels aren't necessary here. However, we
    # need to add "dummy" targets for rate and distortion.
    return image, dict(rate=0., distortion=0.)

def train_mnist_model(lmbda):
    trainer = make_mnist_compression_trainer(lmbda)
    trainer.fit(
        training_dataset.map(add_rd_targets).batch(128).prefetch(8),
        epochs=5,
        validation_data=validation_dataset.map(add_rd_targets).batch(128).cache(),
        validation_freq=1,
        verbose=1,
    )
    return trainer

trainer = train_mnist_model(lmbda=2000)
```

Manipulation: 3 Quelle est la taille des lots utilisés dans l'entrainement ?

D'après le code précédent, on peut savoir que la taille des lots utilisés dans l'entrainement est de 128.

Manipulation: 4 Quelle est la signification d'`epoch` ?

Une `epoch` fait référence à l'ensemble d'entraînement parcouru une fois. Par conséquent, la signification de l'`epoch` est le nombre de fois où les images sont parcourues tout au long du processus d'entrainement. `epoch=5` signifie que chaque échantillon d'apprentissage sera transmis au modèle pendant 5 itérations.

Manipulation: 5 Quel est le rôle de `lmbda` ?

Dans la fonction `make_mnist_compression_trainer`, le paramètre `lmbda` est le multiplicateur de Lagrange utilisé pour équilibrer le débit binaire et la distorsion.

Il est utilisé pour équilibrer le débit binaire du modèle et la distorsion lors de la compression d'une image. Une valeur `lmbda` plus grande amènera le modèle à se concentrer davantage sur la réduction de la distorsion, tandis qu'une valeur `lmbda` plus petite amènera le modèle à se concentrer davantage sur la réduction du débit binaire.

Manipulation: 6 Quelles sont les informations que l'on obtient lors de la phase d'apprentissage ?

Dans le code, on a défini `verbose=1`, afin que l'on puisse voir les valeurs de la métrique pertinente obtenue après chaque `epoch`. Ces mesures incluent le débit binaire moyen, la distorsion moyenne, etc.

Listing 13 – Output de Console - Entrainement

```
Epoch 2/10
469/469 [==============================] - 62s 132ms/step - loss: 165.7330 -
    distortion_loss: 0.0410 - rate_loss: 83.7771 - distortion_pass_through_loss:
    0.0410 - rate_pass_through_loss: 83.7730 - val_loss: 156.3432 -
    val_distortion_loss: 0.0404 - val_rate_loss: 75.5539 -
    val_distortion_pass_through_loss: 0.0404 - val_rate_pass_through_loss: 75.5695
```

3 Compresser des images de la base MNIST

Pour la compression et la décompression au moment du test, nous divisons le modèle en deux parties :
— Le côté codeur se compose de la transformée d'analyse et du modèle d'entropie.
— Le côté décodeur est constitué de la transformée de synthèse et du même modèle d'entropie.

Au moment du test, les latents n'auront pas de bruit additif, mais ils seront quantifiés puis compressés sans perte. compressés sans perte, nous leur donnons donc de nouveaux noms. Nous les appelons ainsi que la reconstruction de l'image *boldsymbolx* et *boldsymboly*, respectivement.

Listing 14 – Compress

```
class MNISTCompressor(tf.keras.Model):
    """Compresses MNIST images to strings."""

    def __init__(self, analysis_transform, entropy_model):
        super().__init__()
        self.analysis_transform = analysis_transform
        self.entropy_model = entropy_model

    def call(self, x):
        # Ensure inputs are floats in the range (0, 1).
        x = tf.cast(x, self.compute_dtype) / 255.
        y = self.analysis_transform(x)
        # Also return the exact information content of each digit.
        _, bits = self.entropy_model(y, training=False)
        return self.entropy_model.compress(y), bits

class MNISTDecompressor(tf.keras.Model):
    """Decompresses MNIST images from strings."""

    def __init__(self, entropy_model, synthesis_transform):
        super().__init__()
        self.entropy_model = entropy_model
        self.synthesis_transform = synthesis_transform

    def call(self, string):
        y_hat = self.entropy_model.decompress(string, ())
        x_hat = self.synthesis_transform(y_hat)
        # Scale and cast back to 8-bit integer.
        return tf.saturate_cast(tf.round(x_hat * 255.), tf.uint8)
```

Lorsqu'il est instancié avec `compression=True`, le modèle entropique convertit l'*a priori* appris en tables pour un algorithme de codage par plage (c'est plus ou moins un codeur arithmétique). Lors de l'appel de `compression()`, cet algorithme est utilisé pour convertir le vecteur de l'espace latent en séquences de bits. La longueur de chaque chaîne binaire est une approximation de l'information du vecteur latent.

Le modèle d'entropie pour la compression et la décompression doit être le même, car les tables de codage des plages doivent être exactement identiques des deux côtés. Dans le cas contraire, des erreurs de décodage peuvent se produire.

Listing 15 – Compress

```
def make_mnist_codec(trainer, **kwargs):
    # The entropy model must be created with `compression=True` and the same
    # instance must be shared between compressor and decompressor.
    entropy_model = tfc.ContinuousBatchedEntropyModel(
        trainer.prior, coding_rank=1, compression=True, **kwargs)
    compressor = MNISTCompressor(trainer.analysis_transform, entropy_model)
    decompressor = MNISTDecompressor(entropy_model, trainer.synthesis_transform)
    return compressor, decompressor

compressor, decompressor = make_mnist_codec(trainer)
```

Manipulation: 1 Selectionner un lot de 16 images de l'ensemble de validation.

Comment sélectionner un autre lot ?

Selon le code, dans ce code, les images de l'ensemble de validation sont d'abord divisées en lots de taille 16. Ensuite, `skip=3` signifie sauter les 3 premiers lots (c'est-à-dire sauter les 48 premières images), et `take=1` signifie prendre le lot suivant (c'est-à-dire la 49e à la 64e image).

Si on veut sélectionner un autre lot, on peut utiliser le code suivant (changer la valeur de `skip`) :
`(originals, _), = validation_dataset.batch(16).skip(1).take(1)`

Listing 16 – Dataset

```
(originals, _), = validation_dataset.batch(16).skip(3).take(1)
```

Manipulation: 2 Compressez ces images afin d'obtenir des chaines de bits, et calculer l'entropie associée.

En utilisant le code ci-dessous, nous pouvons compresser le lot d'images et également afficher la chaîne et la valeur d'entropie de la première image. Les résultats sont présentés dans le listing ci-dessous.

Listing 17 – Output de Console - Entropie

```
String representation of digit in hex: 0xa6d99e05b32f
Number of bits actually needed to represent it: 44.90
```

Listing 18 – Entropie

```
strings, entropies = compressor(originals)

print(f"String representation of digit in hex: 0x{strings[0].numpy().hex()}")
print(f"Number of bits actually needed to represent it: {entropies[0]:0.2f}")
```

Manipulation: 3 Décompressez les chaines de bits afin d'obtenir à nouveau les images.

En utilisant la fonction de `decompressor()`, on peut décoder les images codées.

Listing 19 – Decompress

```
reconstructions = decompressor(strings)
```

Manipulation: 4 Afficher les 16 images originales ainsi que leur représentation compressée, et les images reconstruites.

Pourquoi la longueur de la chaîne codée diffère-t-elle du contenu informatif de chaque chiffre ?

Étant donné que la chaîne codée affiche une expression hexadécimale dans le code, on peut calculer son nombre correspondant de bits binaires pour le comparer à la valeur d'entropie calculée.

On peut voir que les valeurs de bits et les valeurs d'entropie obtenues sont relativement proches mais présentent certaines différences. En effet, certaines propriétés statistiques sont prises en compte lors du calcul de la valeur d'entropie et il existe une certaine redondance des données dans le processus de compression.

Listing 20 – Calcul de PSNR

```
def display_digits(originals, strings, entropies, reconstructions):
    """Visualizes 16 digits together with their reconstructions."""
    fig, axes = plt.subplots(4, 4, sharex=True, sharey=True, figsize=(12.5, 5))
    axes = axes.ravel()
    for i in range(len(axes))
        psnr_value = calculate_psnr(originals[i], reconstructions[i])
        #0x{strings[i].numpy().hex()}
```

```
        hex_string = strings[i].numpy().hex()
        nombres_bits = len(hex_string) * 4
        image = tf.concat([
            tf.squeeze(originals[i]),
            tf.zeros((28, 14), tf.uint8),
            tf.squeeze(reconstructions[i]),
        ], 1)
        axes[i].imshow(image)
        axes[i].text(
            .5, .5,
            f"␣{nombres_bits}␣bits␣\n{entropies[i]:0.2f}␣bits␣\n␣{psnr_value}dB␣\n␣
                PSNR:␣{psnr_value:.2f}␣dB",
            ha="center", va="top", color="white", fontsize="small",
            transform=axes[i].transAxes)
        axes[i].axis("off")
    plt.subplots_adjust(wspace=0, hspace=0, left=0, right=1, bottom=0, top=1)
display_digits(originals, strings, entropies, reconstructions)
```

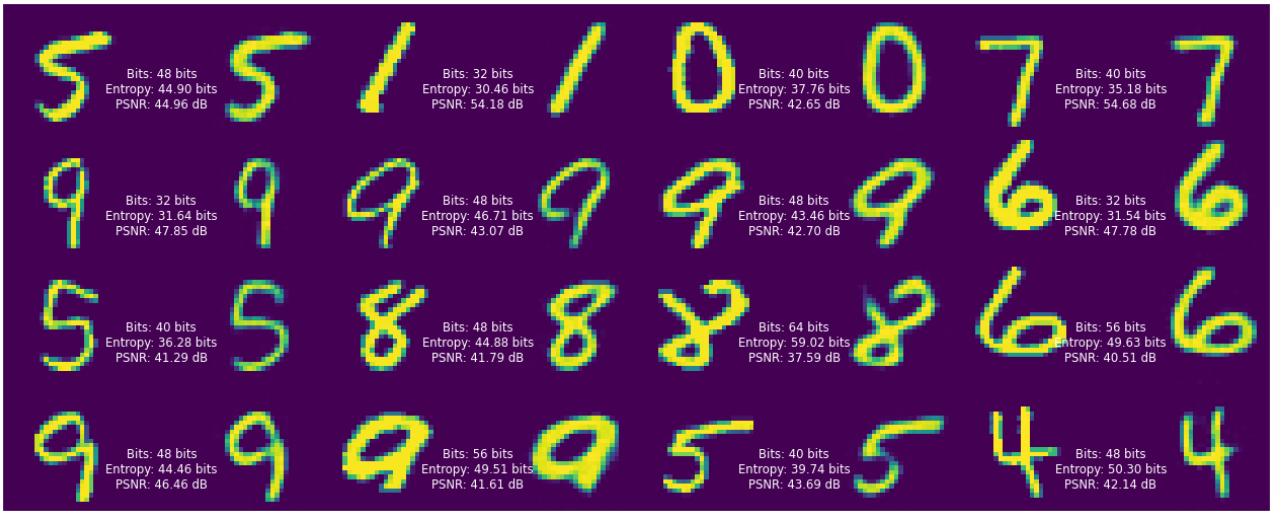

FIGURE 3.0.1 – Images reconstruites (lmbda=2000)

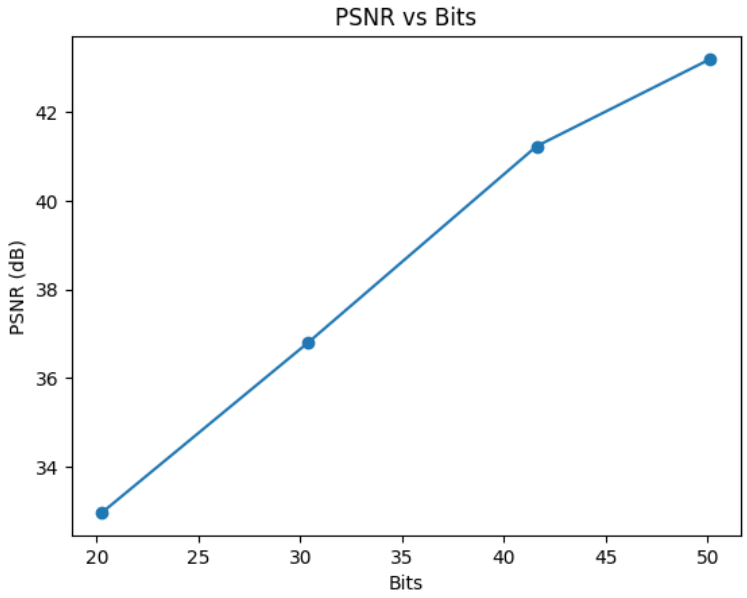

FIGURE 3.0.2 – Courbe PSNR.

Manipulation: 5 Evaluer également le PSNR (en dB).

On a écrit une fonction pour calculer sa valeur PSNR. Afin d'évaluer la courbe PSNR, on a sélectionné `lmbda = 500,1000,1500,2000` comme hyperparamètre pour l'entrainement, et a obtenu quatre ensembles de résultats et les a tracés comme le montre la figure 3.0.2 (On a choisi `epoch=5`). On voit que, comme pour les méthodes de compression JPEG et JPEG2000, lorsque le débit est plus important, la valeur PSNR obtenue est également plus grande.

De même, on a tracé les courbes de PSNR sur le débit et sur la valeur d'entropie, comme le montre la figure 3.0.3 (On a choisi `epoch=1`). Sur cette figure, on peut voir que les deux courbes sont similaires mais présentent certaines différences.

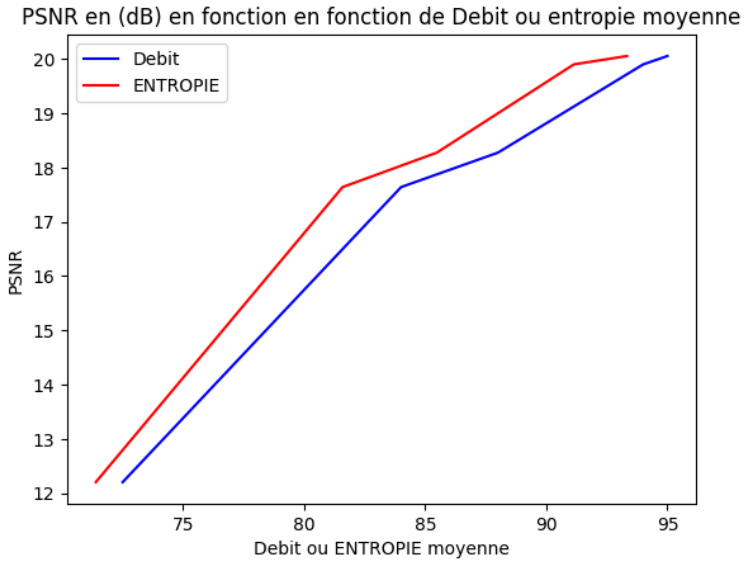

FIGURE 3.0.3 – Courbe PSNR

4 Compromis entre débit et distorsion

Dans ce qui précède, le modèle a été entraîné pour un compromis spécifique (donné par `lmbda=2000`) entre le nombre moyen de bits utilisés pour représenter chaque chiffre et l'erreur de reconstruction.

Manipulation: 6 Quese passe-t-il si l'on répéte l'expérience avec des valeurs différentes, par exemple `lmbda=500` ?

Le résultat est dans la figure 4.0.1.

Comme on l'a analysé dans la question précédente, une valeur `lmbda` plus grande amènera le modèle à se concentrer davantage sur la réduction de la distorsion, tandis qu'une valeur `lmbda` plus petite amènera le modèle à se concentrer davantage sur la réduction du débit binaire.

Par conséquent, pour `lmbda=500`, la valeur d'entropie obtenue sera inférieure à `lmbda=2000`. Mais relativement, l'effet de reconstruction de l'image, c'est-à-dire la valeur PSNR, sera légèrement inférieur.

Manipulation: 7 Que se passe-t-il lorsqu'on réduit `lmbda=300` ?

Le résultat est dans la figure 4.0.2. En comparant les deux chiffres ci-dessous, on peut comparer plus intuitivement les différences apportées par les différentes valeurs lmbda. Lorsque `lmbda=300`, la qualité de reconstruction de l'image est relativement mauvaise et la valeur d'entropie requise est également relativement faible.

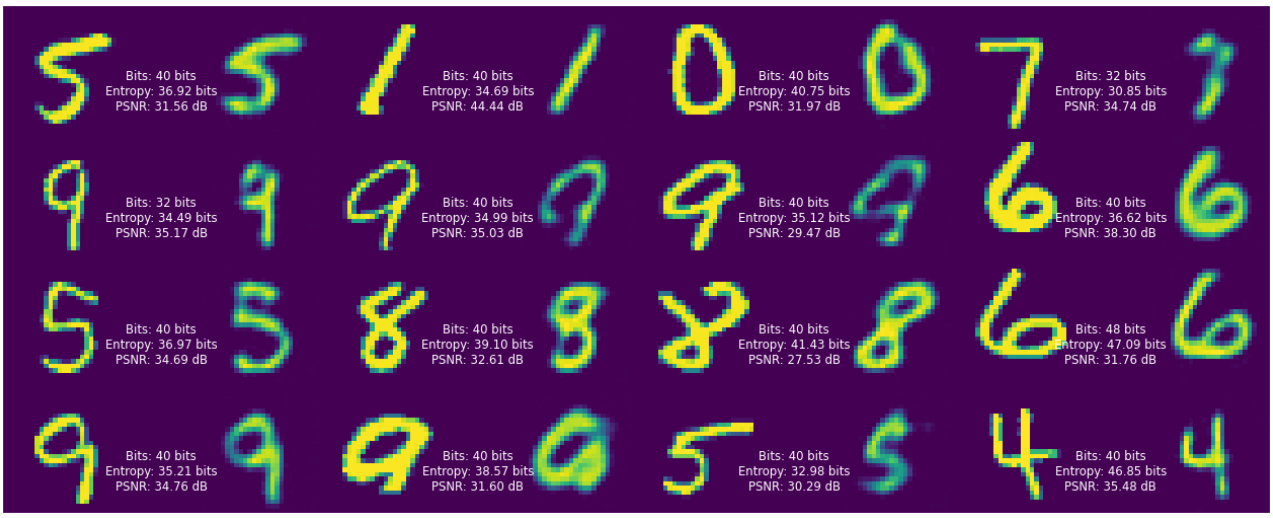

FIGURE 4.0.1 – Images reconstruites (lmbda=500)

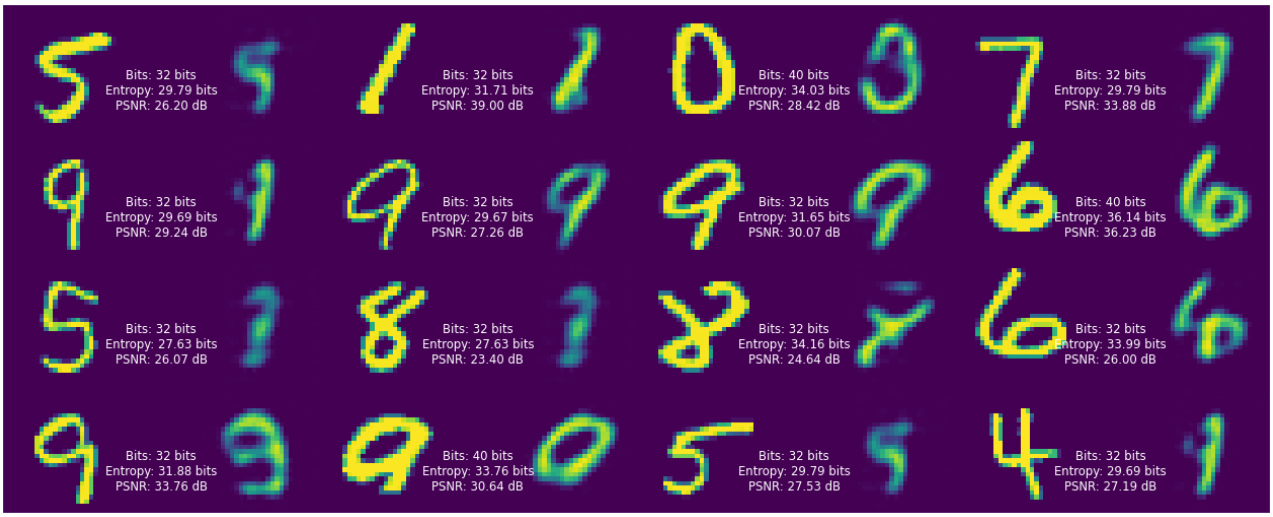

FIGURE 4.0.2 – Images reconstruites (lmbda=300)

Manipulation: 8 Evaluer l'influence de la dimension de l'espace latent.

Les dimensions de l'espace latent représentent la complexité des données dont il peut tirer des enseignements. L'augmentation de la dimensionnalité améliore la qualité des images reconstruites du modèle, car le modèle dispose de plus de degrés de liberté pour capturer les caractéristiques des données.

Cependant, des dimensions plus élevées signifient également que davantage de bits sont nécessaires pour coder la représentation sous-jacente, ce qui entraîne une augmentation du débit binaire.

Manipulation: 9 Evaluer l'influence du choix des probabilités `a priori` pour les composantes de l'espace latent.

Un bon a priori peut refléter avec précision les caractéristiques de distribution des données. Si la distribution antérieure correspond à la véritable distribution des données d'image, le modèle peut alors coder les données plus efficacement et on peut ainsi obtenir de meilleures performances de compression.

5 Utiliser le décodeur comme un modèle génératif

Si nous alimentons le décodeur avec des bits aléatoires, nous échantillonnerons effectivement la distribution que le modèle a apprise pour représenter les chiffres.

Tout d'abord, créer deux instances du compresseur/décompresseur sans contrôle d'intégrité. Cela évite de détecter si la chaîne d'entrée n'est pas complètement décodée.

Listing 21 – Code et decode

```
compressor, decompressor = make_mnist_codec(trainer, decode_sanity_check=False)
```

Maintenant, introduisez des chaînes aléatoires suffisamment longues dans le décompresseur pour qu'il puisse décoder/échantillonner des chiffres à partir de ces chaînes.

Listing 22 – Code et decode

```
import os
strings = tf.constant([os.urandom(8) for _ in range(16)])
samples = decompressor(strings)

fig, axes = plt.subplots(4, 4, sharex=True, sharey=True, figsize=(5, 5))
axes = axes.ravel()
for i in range(len(axes)):
    axes[i].imshow(tf.squeeze(samples[i]))
    axes[i].axis("off")
plt.subplots_adjust(wspace=0, hspace=0, left=0, right=1, bottom=0, top=1)
```

Manipulation: 1 Qu'observez-vous ?

Dans le code, on peut voir que l'on a généré un ensemble de `string` aléatoires et les afficher via le décodeur.

Puisqu'il s'agit d'une chaîne de bits générée aléatoirement, l'image générée après décodage ne correspond pas à un nombre spécifique. Mais comme on a entraîné le réseau, le décodeur aura tendance à générer une image proche de la forme d'un nombre. De là, on peut également voir la compréhension du modèle de l'ensemble d'entrainement MNIST après l'entrainement.

Le résultat est dans la figure 5.0.1. Ce résultat est obtenu à partir du modèle à l'`epoch=15`.

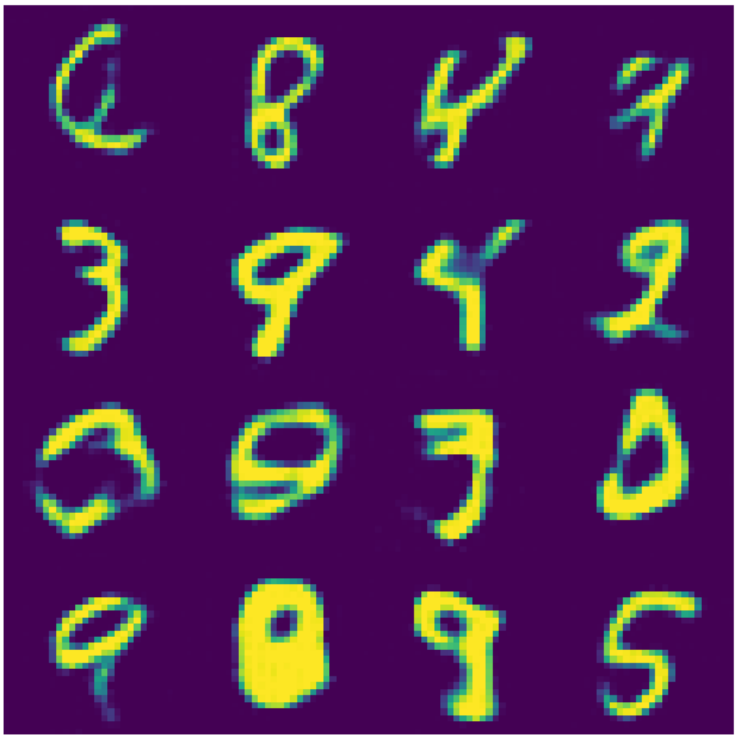

FIGURE 5.0.1 – Images

Manipulation: 2 Etudiez à nouveau l'impact de la dimension de l'espace latent.

Comme on l'a analysé dans le problème précédent, lorsque la dimension de l'espace latent augmente, le modèle peut mieux extraire les caractéristiques de l'image et a une meilleure généralisation.

D'un autre côté, si la dimensionnalité est trop grande, cette chaîne de bits générée de manière aléatoire peut être décodée pour obtenir une image similaire à celle de l'ensemble de données MNIST, conduisant à un surajustement.

6 Conclusion

Pendant ce TP, on a fait
— étudié la compression avec perte implique un compromis entre débit et distorsion.
— implémenté un modèle similaire à un autoencodeur pour compresser des images du jeu de données MNIST.
— appris la méthode de réseau de neurone appliqué à la compression.

[1]

Références

[1] M. Kieffer. *UE 455 – Théorie de l'Information - Codage de Source.* Université Paris-Saclay, 2023-2024.