Département Nikola Tesla (2024) , 1–??

Learned data compression: Utilisation d'une architecture encodeur-décodeur pour la compression de données

P.G Basile[1⋆]
[1] *École Normale Supérieure Paris-Saclay, France*

30 Avril 2024

RESUME
Ce rapport contient une explication et une analyse du code disponible dans ce repo. L'étude porte sur la compression de données "apprise" à l'aide d'un modèle encodeur - décodeur.

Mots clés: Encoder Decoder – Data compression – Dithered noise – Generative AI

1 MODÈLE DE L'ENTRAINEMENT

1.1 Architecture du modèle

Réseau d'analyse (transformateur d'encodeur) :

Couche 1 :	Convolution 2D
	Nombre de filtres : 20
	Taille du noyau : 5×5
	Strides : 2
	Activation : Leaky ReLU
	Padding : "same"
Couche 2 :	Convolution 2D
	Nombre de filtres : 50
	Taille du noyau : 5×5
	Strides : 2
	Activation : Leaky ReLU
	Padding : "same"
Couche 3 :	Aplatissement
Couche 4 :	Dense (entièrement connectée)
	Nombre de neurones : 500
	Activation : Leaky ReLU
Couche 5 :	Dense (entièrement connectée)
	Nombre de neurones : dimension latente
	Activation : Aucune

Réseau de synthèse (transformateur de décodeur) :

Couche 1 :	Dense (entièrement connectée)
	Nombre de neurones : 500
	Activation : Leaky ReLU
Couche 2 :	Dense (entièrement connectée)
	Nombre de neurones : 2450
	Activation : Leaky ReLU
Couche 3 :	Remodelage (Reshape)
	Nouvelle forme : (7, 7, 50)
Couche 4 :	Convolution transposée 2D
	Nombre de filtres : 20
	Taille du noyau : 5×5
	Strides : 2
	Activation : Leaky ReLU
	Padding : "same"
Couche 5 :	Convolution transposée 2D
	Nombre de filtres : 1
	Taille du noyau : 5×5
	Strides : 2
	Activation : Leaky ReLU
	Padding : "same"

1.2 Calcul du débit et de la distorsion

On travaille sur le dataset MNIST dont les caractéristiques sont les suivantes :
Dataset Specifications:

Feature	**Description**
Image	Grayscale $28 \times 28 \times 1$, uint8
Label	64-bit int (int64), 10 classes
Key	**Examples**
Train	60,000 examples
Test	10,000 examples

Une image du dataset est représentée figure 1. C'est un dataset de digits écrits à la main.

Nous allons convertir les images `x` de type `uint8` en `y` de type

⋆

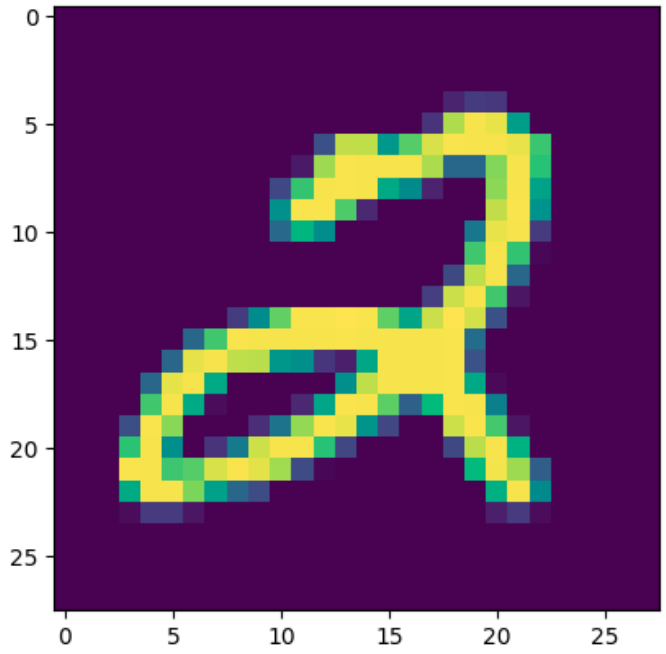

Figure 1. Exemple d'image du dataset MNIST

`float32`. La taille de la donnée est aussi modifiée puisqu'on passe de $28 \times 28 \times 1$ à 10×1.

On souhaite connaitre la forme de la distribution de probabilité de la représentation latente afin de pouvoir construire un quantificateur approprié : ainsi on pourra influer sur le débit et la distorsion.

Nous allons donc entrainer l'encodeur/décodeur à obtenir une représentation latente dont la distribution de probabilité est proche d'une distribution normale (la distribution normale est un choix, nous aurions pu choisir une autre distribution). La distribution que l'on cherche à imiter se nomme *à priori* ou *prior*.

Pour cela, on va minimiser la divergence de Kullback-Leibler entre la distribution de probabilité de la représentation latente et une distribution normale.

1.2.1 Ajout de bruit

On ajoute du bruit sur le vecteur `y` pour obtenir le vecteur `y_tilde`. Ajouter du bruit possède plusieurs avantages :

- **Quantification sans perte** : En ajoutant du bruit uniforme, on introduit de l'incertitude dans la représentation latente. Cela permet de considérer plusieurs valeurs proches comme équivalentes, ce qui facilite la quantification sans perte de la représentation latente. Ainsi, une fois la quantification effectuée, le bruit ajouté peut être inversé, permettant de reconstruire une représentation proche de la représentation latente d'origine : c'est le principe de la *dithered quantization* vue en cours. On rend plus différentiable la quantification.
- **Réduction de la sensibilité aux erreurs de quantification** L'ajout de bruit réduit la sensibilité aux erreurs de quantification. Sans bruit, de petites erreurs de quantification pourraient entraîner des dégradations perceptuelles importantes dans l'image reconstruite. En ajoutant du bruit, on atténue l'impact de ces erreurs.
- **Régularisation** L'ajout de bruit peut également agir comme une forme de régularisation, aidant à prévenir le surajustement (overfitting) en introduisant une certaine robustesse dans le modèle.

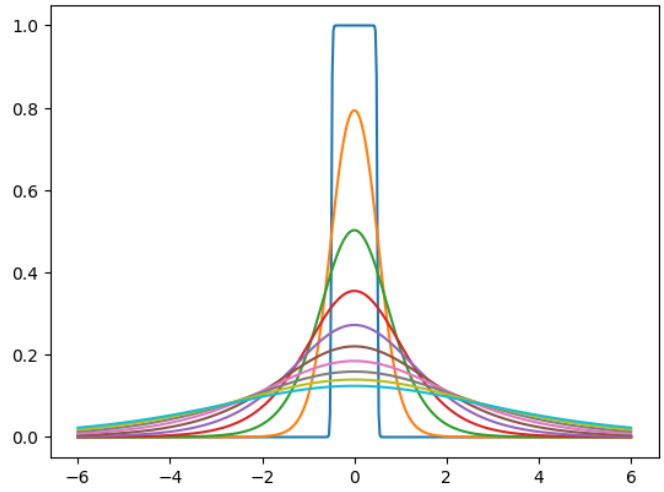

Figure 2. Distributions de probabilités *à priori*

De manière générale la normalisation des données aide à la convergence du modèle.

On va donc ajouter du bruit pendant l'entrainement. L'ajout de bruit fait que la distribution de probabilité de la représentation latente ne sera pas exactement normale. Par exemple figure 2 montre les distributions de probabilités que l'on va chercher à imiter pour la représentation latente. Lorsque l'écart type $\sigma \rightarrow 0$ nous devrions obtenir un dirac, mais du fait du bruit nous obtenons plutôt une distribution uniforme sur [-0.5,0.5].

Le figure 2 représente toutes les distributions de probabilités que l'on va chercher à imiter pour la représentation latente : c'est la somme du variable aléatoire gaussienne et du bruit. C'est donc le produit de convolution entre les deux variables aléatoires.

Le bruit est pris en compte lorsque nous utilisons la fonction :

```
prior = tfc.NoisyLogistic(loc=0., scale=tf.
    linspace(.01, 2., 10))
```

La fonction

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

On ajoute du bruit à la représentation latente. Elle utilise à la fois le bruit et la représentation *à priori* pour calculer le débit maximum atteignable. Ce débit est différentiable ce qui va nous permettre de calculer son gradient et de l'utiliser comme fonction coût pour l'entrainement.

Remarque : Ce débit maximal est atteint si l'encodeur est parfait et que la représentation latente suit exactement la distribution *à priori* à laquelle on ajoute le bruit.

On trouve un débit de 18.334541 bits par pixel (bpp).

1.2.2 Reconstruction de l'image à partir de la représentation latente

Maintenant que nous avons une représentation latente de notre image, nous allons pouvoir la reconstruire à l'aide du décodeur. Sans

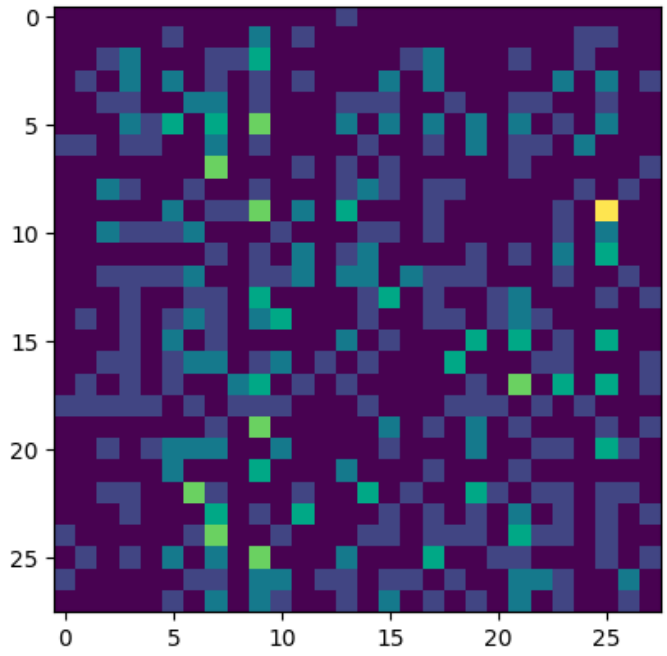

Figure 3. Reconstruction de l'image à partir de la représentation latente

entrainement, la reconstruction obtenue (représentée figure 3) est très mauvaise. Cela est dû au fait que le décodeur n'a pas encore appris à reconstruire correctement l'image à partir de la représentation latente.

Nous allons donc entrainer l'encodeur à construire une représentation latente et le décodeur à reconstruire l'image.

Avant entrainement on trouve avec la fonction

```
(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"])
```

avec un batch de taille 32 images. Le paramètre `10` indique la taille de l'espace latent

1.3 Entrainement du modèle

Pour l'entrainement, on instancie un objet de la classe `MNISTCompressionTrainer` avec la taille de l'espace latent à 10. On lui associe une méthode d'optilization (ici `Adam`), une metrics (ici distorsion et débit) et une fonction coût (ici la somme de la distorsion et du débit). On entraine le modèle sur 15 epochs, c'est à dire 15 passages complets à travers le dataset.

On précise de plus la valeur du paramètre `lmbda` du Lagragien. Ce paramètre précise l'importance relative de la distorsion

[h]

Distorsion	Débit
0.147	20.3

Figure 4. Reconstruction de l'image à partir de la représentation latente

et du débit.

La bibliothèque `TensorFlow` précise chacun de ces paramètres durant l'entrainement.

2 APPLICATION SUR LE DATASET MNIST

On sélectionne un set de 16 images sur lequel on va appliquer notre modèle avec

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

En ajustant le paramètre de `skip` on peut sélectionner un autre set d'images.

On obtient les résultats suivants présentés figure 4.

Ainsi le modèle entrainé est capable de reconstruire les images du dataset MNIST.

La longueur de la représentation en héxa de chaque chiffre peut varier car on encode l'information visuelle, qui varie selon la complexité de la représentation graphique du digit.

2.0.1 Signal to Noise Ratio

Pour évaluer la qualité de l'image reconstruite, on peut calculer le rapport signal à bruit crête en dB (PSNR)

$$PSNR_{(\mathrm{dB})} = 10\log_{10}\frac{255*255}{\sigma_{\mathrm{D}}^2}$$

où σ_{D}^2 est la variance de la différence entre l'image initiale et l'image reconstruite. Pour calculer le rapport signal à bruit, définit la fonction suivante qui sera appelée pour chaque image du batch et dont on affiche la valeur sur la figure 4. On constate que le PSNR est plus élevé pour les chiffres simples comme le 1 et le 7, et plus faible pour le chiffres complexes comme le 8 et le 9 : le modèle a plus de mal à reconstruire les chiffres complexes.

```
def tf_psnr(original, reconstructed):
  # Ensure the pixel values are floats (
  especially important if the images are
  uint8)
  original = tf.cast(original, tf.float32)
  reconstructed = tf.cast(reconstructed, tf.
  float32)
```

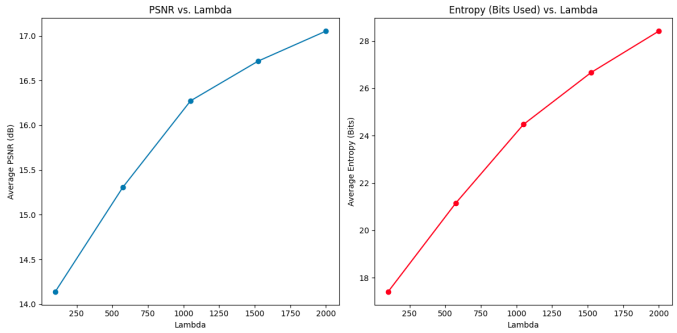

Figure 5. Courbe de compromis entre débit et distorsion

```
    # Calculate MSE using TensorFlow's mean and
     square functions
    mse = tf.reduce_mean(tf.square(original -
    reconstructed))

    # Define the maximum pixel value. Change
    this if you're not using 8-bit images.
    max_pixel = 255.0

    # Calculate PSNR
    psnr = 20 * tf.math.log(max_pixel / tf.sqrt
    (mse)) / tf.math.log(10.0)

    return psnr
```

3 COMPROMIS ENTRE DÉBIT ET DISTORSION

Le compromis entre débit et distorsion est caractérisé par la valeur de λ dans la fonction coût. Nous l'avons fixé à 2000. On peut tracer la courbe de compromis entre débit et distorsion pour différentes valeurs de λ figure 5.

En jouant sur la taille de l'espace latent on s'aperçoit que réudire la taille de l'espace latent réduit la distorsion mais augmente le débit, ce qui est logique : on transmet moins de bits donc de l'information est perdue, mais la communication est plus rapide.

4 DÉCODEUR COMME UN MODÈLE GÉNÉRATIF

Maintenant que nous avons un décodeur entrainé, nous pouvons l'utiliser pour générer des images. En mettant en entrée du décodeur un vecteur d'état latent aléatoire, le modèle génère une image, figure 6.

On constate que cette fois, la dimension de l'espace latent n'influence pas la qualité de l'image générée.

[H]

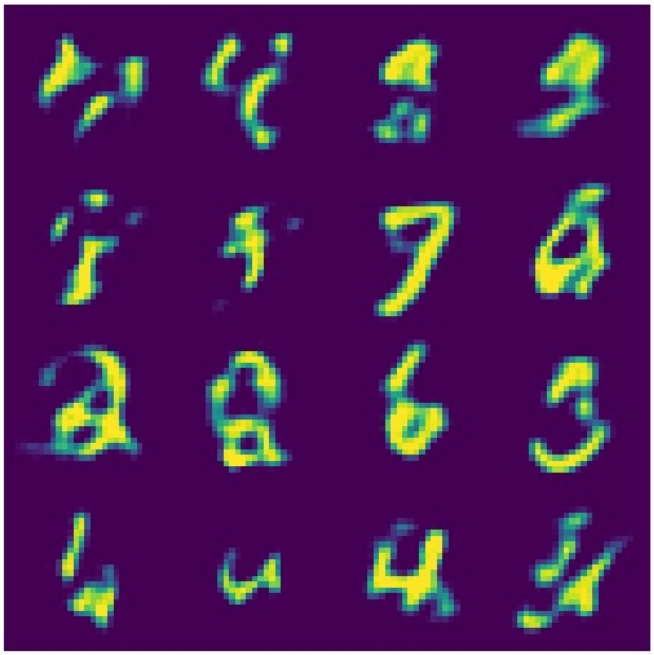

Figure 6. Génération d'images