Learned Data Compression using MNIST dataset

Yanis Gomes[1,†]

[1] *Ecole Normale Supériure de Paris-Saclay*

May 1, 2024

Abstract

Ce rapport présente une étude sur la compression de données apprise. Nous commençons par discuter de l'importance de la transformation DCT-IDCT dans le processus de compression d'image, en mettant l'accent sur son rôle dans la décorrélation des coefficients de l'image pour une compression plus efficace. Cependant, nous soulignons également les limites de la DCT, notamment son incapacité à prendre en compte le schéma de compression et à capturer des structures complexes. Par conséquent, notre objectif est de trouver une transformation optimale non-linéaire qui peut prendre en compte ces structures complexes tout en tenant compte de l'architecture du schéma de compression. Nous définissons ensuite le modèle d'entraînement et l'architecture de l'auto-encodeur, qui joue un rôle crucial dans la réalisation de la transformation d'analyse.

Keywords: Compression de données, Transformation DCT-IDCT, Auto-encodeur variationnel

INTRODUCTION

La compression d'image, un sous-domaine de la compression de données, est essentielle pour optimiser le stockage et la transmission des données. Ce rapport se concentre sur l'utilisation et les limites de la transformation DCT-IDCT, une technique courante dans ce domaine. Nous visons à proposer une approche améliorée pour une compression d'image de qualité supérieure.

Définition du modèle d'entrainement

0.1. Architecture de l'auto-encodeur

Architecture de l'encodeur

L'encodeur réalisant la transformation d'analyse, c'est-à-dire convertissant l'image en vecteur dans un espace latent.

```
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")
```

Code 1. Architecture de l'encodeur

Structure de l'encodeur :

- **Couches de convolution : Conv2D** 20 filtres de taille 5x5 avec un stride de 2 et un padding de "same". Nombre de paramètres entrainables : $(5 \times 5 \times 1 + 1) \times 20 = 520$.
- **Couches de convolution : Conv2D** 50 filtres de taille 5x5 avec un stride de 2 et un padding de "same". Nombre de paramètres entrainables : $(5 \times 5 \times 20 + 1) \times 50 = 25050$.
- **Flatten :** Aplatit les données en un vecteur. Nombre de paramètres entrainables : 0.
- **Dense :** Couche dense de 500 neurones avec une fonction d'activation leaky relu. Nombre de paramètres entrainables : $50 \times 500 + 500 = 250500$.
- **Dense :** Couche dense de latent_dims neurones avec une fonction d'activation linéaire. Nombre de paramètres entrainables pour $latent\_dims$: $500 \times latent\_dims + latent\_dims = 500 \times 50 + 50 = 25050$.

Espace latent

La dimension de l'espace latent est un hyperparamètre du modèle réglé à la main. Dans la pratique on choisit une dimension de l'espace latent plus petite que la taille de l'image en entrée pour forcer le modèle à apprendre une représentation plus compacte de l'image. On prendra ici une dimension de 50 arbitrairement.

Remarque Le nombre de pramètres dépend de la taille de l'image en entrée. Pour une image de taille 28x28, le nombre total de paramètres entraînables de l'encodeur est de $520 + 25,050 + 0 + 1,225,500 + 25,050 = 1,276,120$.

Nombre de paramètres entrainables 1 276 120

Architecture du décodeur

Le décodeur réalisant la transformation de synthèse, c'est-à-dire convertissant le vecteur dans l'espace latent en un vecteur dans l'espace image.

```
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")
```

Code 2. Architecture du décodeur

Structure du décodeur :

- **Dense :** Couche dense de 500 neurones avec une fonction d'activation leaky relu. Nombre de paramètres entrainables : $50 \times 500 + 500 = 250500$.
- **Dense :** Couche dense de 2450 neurones avec une fonction d'activation leaky relu. Nombre de paramètres entrainables : $500 \times 2450 + 2450 = 1225500$.
- **Reshape :** Redimensionne les données en un tenseur de taille (7, 7, 50). Nombre de paramètres entrainables : 0.
- **Conv2DTranspose :** 20 filtres de taille 5x5 avec un stride de 2 et un padding de "same". Nombre de paramètres entrainables : $(5 \times 5 \times 1 + 1) \times 20 = 520$.
- **Conv2DTranspose :** 1 filtre de taille 5x5 avec un stride de 2 et un padding de "same". Nombre de paramètres entrainables : $(5 \times 5 \times 20 + 1) \times 1 = 501$.

Remarque : Le nombre de pramètres dépend de la taille de l'image en sortie. Pour une image de taille 28x28, le nombre de paramètres est de $250,500 + 1,225,500 + 0 + 520 + 501 = 1,476,521$.

Nombre de paramètres etrainables : 1 476 521

0.2. Classe de l'auto-encodeur

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.

Calcul du débit et de la distorsion

0.3. Aspect théorique de l'entrainement

Modèle

$$y \sim \mathbb{P}_Y(y) = \prod \mathcal{N}(y; \mu, \sigma^2) \tag{1}$$

Réel

$$y \sim \mathbb{E}_{x \sim P_X} \left[q_{Y|X}(y|x; \phi) \right] \tag{2}$$

Paramètres entrainables

- Encodeur : ϕ
- Décodeur : θ

Entrainement

$$\phi^*, \theta^* = \arg\min_{(\phi,\theta)} \mathbb{E}_{x \sim P_X} \left[\text{Divergence} + \text{Débit} \right] \tag{3}$$

On force l'espace latent à suivre la distribution a priori :

$$\phi^*, \theta^* = \arg\min_{(\phi,\theta)} \mathbb{E}_{x \sim P_X} \left[\text{Distorsion} + \text{Contrainte sur l'espace latent} \right] \tag{4}$$

Ce qui se réécrit :

Fonction coût

On peut définir le coût en fonction des paramètres comme :

$$\mathcal{C}(\phi^*, \theta^*) = \|x - x_{rec}(\phi, \theta)\|^2 + KL \left[q_{Y|X}(y|x; \phi) \| \mathbb{P}_Y(y, \sigma) \right] \tag{5}$$

ON cherche alors à minimiser l'espérance de ce coût :

$$\phi^*, \theta^* = \arg\min_{(\phi,\theta)} \mathbb{E}_{x \sim P_X} \left[\mathcal{C}(\phi^*, \theta^*) \right] \tag{6}$$

0.4. Ensemble de données d'entrainement

On charge le dataset MNIST, qui contient des images de chiffres manuscrits de taille 28x28.

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

Code 3. Génération des datasets

Observations :

- **Taille de l'ensemble d'entrainement :** 60 000 images
- **Taille de l'ensemble de validation :** 10 000 images
- **Taille des images :** 28x28
- **Nombre de classes :** 10
- **Type des images :** uint8
- **Type des labels :** int64

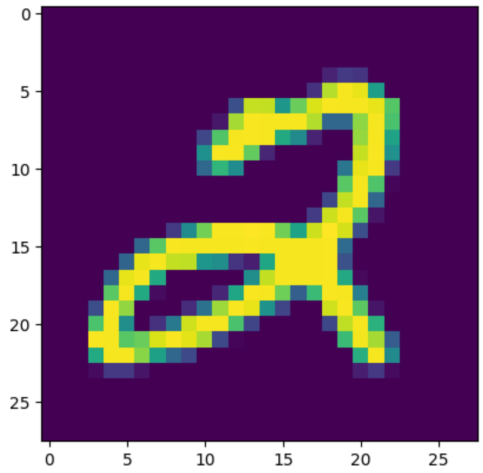

Figure 1. Exemple d'image du dataset MNIST

Représentation latente y est de type `tensorflow.python.framework.ops.EagerTensor` de taille.

Ajout de bruit Les latents seront quantifiés au moment du test. Nous ajoutons un bruit uniforme dans l'intervalle [-0.5, 0.5] et appelons le résultat $\tilde{y}$. L'*a priori* est une densité de probabilité utilisée pour modéliser la distribution marginale des éléments latents bruités. L'opération d'ajout de bruit est effectuée pour simuler l'ajout du bruit de quantification qui sera appliqué lors des tests.

Modélisation du bruit de quantification En ajoutant un bruit uniforme dans l'intervalle [-0.5, 0.5], nous introduisons de l'aléatoire dans les variables latentes, ce qui aide à entraîner le modèle à être robuste face aux erreurs de quantification.

Différentiabilité Par ailleurs l'ajout de bruit permet de rétropropager le gradient dans le réseau car la fonction de coût est différentiable puisqu'elle suit une distribution uniforme. En effet $\hat{y} = \tilde{y} + \epsilon$ avec $\epsilon \sim \mathcal{U}(-0.5, 0.5)$. Donc $\hat{y}$ est différentiable alors que $\tilde{y}$ ne l'est pas.

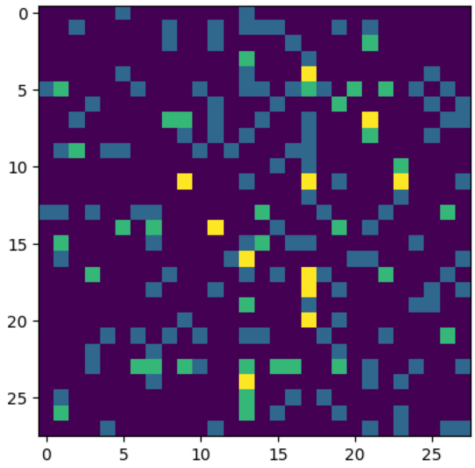

Figure 3. Reconstruction d'une image

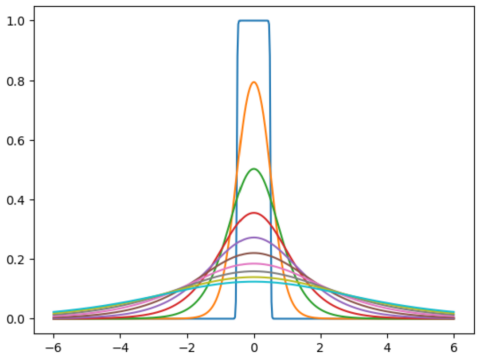

Figure 2. Distributions du bruit de quantification

Première reconstruction 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.

Observation Lorsque le modèle n'est pas encore entraîné, les poids et les paramètres du modèle sont généralement initialisés de manière aléatoire. Par conséquent, lorsque les latents bruités sont passés par la transformée de synthèse, le modèle ne dispose pas encore des connaissances nécessaires pour produire une reconstruction précise de l'image originale. Au lieu de cela, la sortie de cette première reconstruction sera principalement du bruit, car le modèle n'a pas encore appris à extraire les caractéristiques pertinentes de l'image et à les utiliser pour générer une reconstruction de haute qualité.
Il est important de noter que cette première reconstruction bruitée est normale et attendue lorsqu'un modèle n'est pas encore entraîné. Au fur et à mesure que le modèle est entraîné sur des données d'apprentissage, il apprendra à ajuster ses poids et paramètres pour minimiser la distorsion entre les images originales et les reconstructions. Cela permettra au modèle de générer des reconstructions de meilleure qualité au fil du temps.

```
(example_batch, _), = validation_dataset.batch
    (32).take(1)
trainer = MNISTCompressionTrainer(10)
example_output = trainer(example_batch)
print("rate: ", round(float(example_output["rate
    "]), 2))
print("distortion: ", round(float(example_output
    ["distortion"]), 2))
```

Code 4. Numberless section code.

Résultats :

- **Débit :** 20.3
- **Distorsion :** 0.15

Taille du batch La taille du lot utilisé dans cet exemple est de 32 images.

Argument de MNISTCompressionTrainer ? L'argument 10 de MNISTCompressionTrainer est la dimension de l'espace latent.

0.5. Entrainement du modèle

```
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.)
```

Code 5. Numberless section code.

```
def train_mnist_model(lmbda):
    trainer = make_mnist_compression_trainer(
    lmbda)
    trainer.fit(
    training_dataset.map(add_rd_targets).batch
    (128).prefetch(8),
    epochs=15,
    validation_data=validation_dataset.map(add_
    rd_targets).batch(128).cache(),
    validation_freq=1,
    verbose=1,
    )
return trainer
```

Code 6. Numberless section code.

Taille du batch d'entrainement 128

Epoch Une **epoch** ou **époque** renvoie au nombre de fois que le modèle voit l'ensemble de données d'entraînement.

Côté technique Pendant une epoch, l'algorithme d'apprentissage travaille à ajuster les poids du réseau de neurones en fonction de l'erreur qu'il fait sur l'ensemble de données d'entraînement. Cela se fait généralement par rétropropagation de gradient, où l'erreur est calculée pour chaque exemple d'entraînement, puis utilisée pour mettre à jour les paramètres.

Côté pratique Le nombre d'epochs est un hyperparamètre que vous pouvez régler. Si vous entraînez votre modèle pour trop peu d'epochs, il se peut qu'il n'apprenne pas suffisamment à partir des données et sous-performe. Si vous entraînez votre modèle pour trop d'epochs, il se peut qu'il apprenne trop bien les détails spécifiques de l'ensemble de données d'entraînement et surajuste, ce qui signifie qu'il performera moins bien sur de nouvelles données.

Paramètre λ

Le paramètre λ est un hyperparamètre qui contrôle le compromis entre le débit et la distorsion. Il est utilisé pour régler l'importance relative de ces deux objectifs dans la fonction de coût globale du modèle. Un λ plus élevé signifie que le modèle accordera plus d'importance à la distorsion, tandis qu'un λ plus faible signifie que le modèle accordera plus d'importance au débit. Ici λ est réglé à 2000 arbitrairement.

```
trainer = train_mnist_model(lmbda=2000)
```

Code 7. Entrainement du modèle

Phase d'apprentissage

1. **Perte d'entraînement (Training Loss) :** C'est la mesure de l'erreur de votre modèle sur les données d'entraînement. Une perte d'entraînement faible indique que votre modèle s'adapte bien aux données d'entraînement.
2. **Perte de validation (Validation Loss) :** C'est la mesure de l'erreur de votre modèle sur les données de validation. Une perte de validation faible indique que votre modèle généralise bien aux nouvelles données.
3. **Précision (Accuracy) :** C'est le pourcentage de prédictions correctes faites par le modèle. C'est généralement utilisé pour les problèmes de classification.
4. **Époque (Epoch) :** Le nombre d'époques indique combien de fois l'ensemble de données d'entraînement a été utilisé pour ajuster les paramètres du modèle.
5. **Temps d'entraînement :** C'est le temps que le modèle a passé à s'entraîner. Cela peut être important si vous avez des contraintes de temps ou de ressources.
6. **Valeurs des paramètres du modèle :** Les valeurs actuelles des paramètres du modèle (par exemple, les poids dans un réseau de neurones) sont souvent affichées ou enregistrées pendant l'entraînement.

La valeur hexadécimale c'est le flux binaire représenté en hexa. Pour avoir le nombre de bits on fait fois 4 pour obtenir le nombre de bits qu'ils faut pour encoder l'image.

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.

0.6. Compresseur MNIST

```
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
```

Code 8. Numberless section code.

0.7. Décompresseur MNIST

```
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)
```

Code 9. Numberless section code.

0.8. Compression et décompression d'une image

Construction de la chaine de compression

```
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)
```

Code 10. Numberless section code.

Explication La fonction `make_mnist_codec` prend en entrée un modèle entraîné et renvoie un compresseur et un décompresseur pour les images MNIST.

Représentation du flux binaire

- String representation of digit in hex: 0x70aef99c6020
- Number of bits actually needed to represent it: 42.74

A propos du flux binaire

La représentation hexadécimale représnete le flux binaire en hexadécimal. Pour obtenir le nombre de bits nécessaires pour encoder l'image, on multiplie par 4 pour obtenir le nombre de bits.

Reconstructions de digit MNIST

Figure 4. Reconstruction d'un chiffre MNIST

Longueur de la chaîne codée La longueur de la chaîne codée diffère du contenu informatif de chaque chiffre car elle inclut également les informations de codage nécessaires pour reconstruire l'image.

Calcul du PSNR en dB

$$\text{PSNR} = 20 \log_{10} \left(\frac{255}{\sqrt{\text{MSE}}} \right) \tag{7}$$

```
def calculate_psnr(img1, img2):
    mse = np.mean((img1 - img2) ** 2)
    if mse == 0:
        return float('inf')
    max_pixel = 255.0
    psnr = 20 * np.log10(max_pixel / np.sqrt(mse
    ))
    return psnr
```

Code 11. Calcul du PSNR

On peut alors calculer le PSNR entre l'image originale et sa reconstruction.

Figure 5. PSNR entre l'image originale et sa reconstruction

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.

λ	Débit	PSNR
300	12.84	18.37
500	18.85	22.72
1000	30.128	26.81
2000	49.27	34.5

Table 1. Débit et PSNR pour 5 epochs

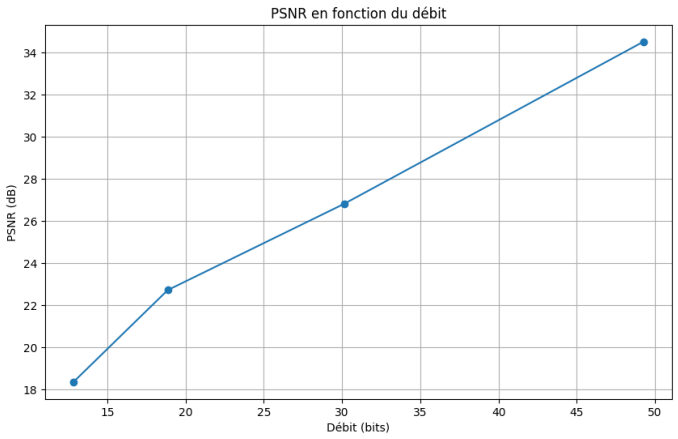

Figure 6. Débit et PSNR pour 5 epochs

Influence de la dimension de l'espace latent La dimension de l'espace latent n'influence pas directement le compromis entre débit et distorsion dans la mesure où sa dimension est suffisamment élevée pour capturer les informations pertinentes de l'image.

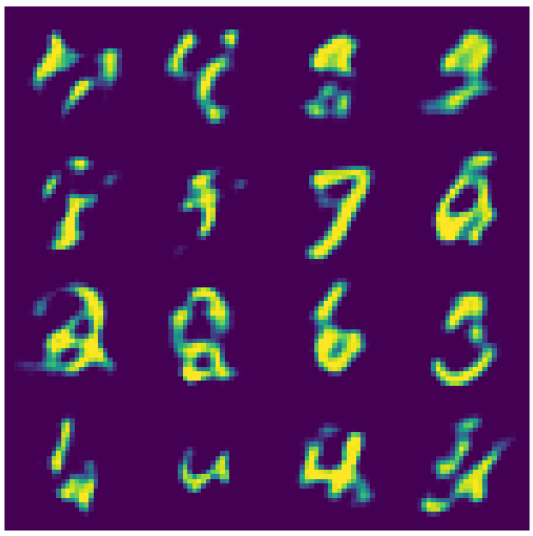

Figure 7. Représentation manuscrites "inventées" par le décodeur

Utiliser un décodeur comme un modèle génératif

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

Code 12. Numberless section code.

Utiliser le décodeur comme modèle génératif correspond à utiliser le décompresseur à partir d'un encodge aléatoire

```
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)
```

Code 13. Numberless section code.

L'information contenu dans l'encdage aléatoire est projetté sur l'espace latent, le modèle cherche donc à donner du sens à ce qui n'en a pas. C'est comme si le modèle "inventait" des nombres manuscrits

Conclusion

Dans ce travail, nous avons exploré l'application des encodeurs variationnels pour la génération de chiffres manuscrits. Nous avons commencé par l'initialisation de notre modèle, en définissant une distribution a priori sur l'espace latent. Cette distribution a priori joue un rôle crucial dans la régularisation de notre modèle et dans la définition de la complexité de celui-ci.
Nous avons aussi démontré qu'il est possible d'utiliser le décodeur pour générer des représentation manuscrites à partir d'un encodage quelconque - aléatoire. Il reste encore beaucoup à explorer, notamment en ce qui concerne le choix optimal des probabilités a priori et la manière dont elles influencent la capacité du modèle à généraliser à partir des données d'entraînement.

CONTACT

yanis.gomes@ens-paris-saclay.fr
Yanis Gomes