(CS529)

Rendu Final

ESISAR (2024-2025) TP2

Etudiante: **Dorra BEN HAJ KALBOUSSI**
Classe: 5e année IR&C
E-mail: *Dorra.Ben-Haj-Kalboussi@grenoble-inp.org*

Tuteur: **M. Simon Gay**
E-mail:
simon.gay@lcis.grenoble-inp.fr

November 26, 2024

Contents

1 Introduction

Ce TP portait sur l'implémentation d'un réseau de neurones profond à l'aide de TensorFlow et Keras. Nous avons travaillé avec le dataset MNIST.

2 Objectif

- Charger et prétraiter les données du dataset MNIST.
- Implémenter un réseau de neurones à plusieurs couches à l'aide de TensorFlow et Keras.
- Analyser la structure et les paramètres du modèle.
- Visualiser les données et comprendre leur transformation à travers le modèle.

3 Petit échauffement avec TensorFlow et Keras

Méthodologie

3.1 Préparation des données

- On commence par importer les bibliothèques nécessaires : TensorFlow, Keras, NumPy et Matplotlib.
- Ensuite, le jeu de données MNIST est chargé à l'aide de la fonction `keras.datasets.mnist.load_data()`.
- Les images sont normalisées en divisant les valeurs des pixels par 255 pour les ramener dans l'intervalle [0, 1].
- Les étiquettes sont converties en vecteurs catégoriels à l'aide de `keras.utils.to_categorical()`.
- Un échantillon d'images a été affiché pour vérifier la qualité des données.

3.2 Construction du modèle

Un modèle séquentiel est créé avec Keras, comprenant les couches suivantes :

1. Une couche Flatten pour transformer les images 28×28 en vecteurs de 784 éléments.
2. Une couche Dense de 20 neurones avec activation sigmoïde.
3. Une couche Dense de sortie de 10 neurones avec activation sigmoïde.

Résultats

3.3 Exploration des données

- **Dimensions des données :**
  - Ensemble d'entraînement : $(60000, 28, 28)$
  - Ensemble de test : $(10000, 28, 28)$
- **Dimensions des sorties après encodage :**
  - Labels d'entraînement : $(60000, 10)$
  - Labels de test : $(10000, 10)$
- Un échantillon de 10 images a été affiché permettant de visualiser les chiffres manuscrits.

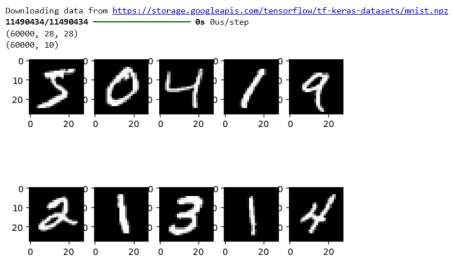


Figure 1: Résultats obtenus de l'exploration de données

3.4 Structure du modèle

Le modèle créé a la structure suivante :

- **Couche `Flatten` :** Aucun paramètre modifiable.
- **Couche `Dense` (20 neurones) :** 15,700 paramètres modifiables.
  - Calcul : $784 \times 20 + 20 = 15,700$
- **Couche `Dense` (10 neurones) :** 210 paramètres modifiables.
  - Calcul : $20 \times 10 + 10 = 210$

Total des paramètres : 15,910.

Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 20)	15,700
dense_1 (Dense)	(None, 10)	210

Total params: 15,910 (62.15 KB)
Trainable params: 15,910 (62.15 KB)
Non-trainable params: 0 (0.00 B)

Figure 2: Résultats obtenus après création de structure

3.5 Compilation du réseau

Le réseau a été compilé avec les paramètres suivants :

- **Fonction de coût :** `MeanSquaredError`.
- **Optimiseur :** `SGD (Stochastic Gradient Descent)` avec un taux d'apprentissage (`learning_rate`) de 0.05.

3.6 Entraînement du réseau

Le réseau a été entraîné avec les paramètres suivants :

- Nombre d'**epochs** : 20
- Taille des **batches** : 4
- Les données ont été mélangées (`shuffle=True`).

Après compilation et entraînement, les résultats obtenus sont :

```
2500/2500 - 4s - 1ms/step - accuracy: 0.9507 - loss: 0.0084
[0.008440840058028698, 0.9506999850273132]
```

Figure 3: Résultats obtenus après compilation et entraînement

3.7 Analyse des performances

- La précision (`accuracy`) obtenue est de 95.07%, avec une perte (`loss`) de 0.0084.
- Comparaison avec une implémentation précédente en Java :
  - Précision en Java : 96%.
  - Précision en Python : 95%.

On observe que l'utilisation de plusieurs exemples simultanément (**batching**) accélère l'apprentissage au détriment de l'efficacité globale du réseau.

3.8 Évaluation sur le dataset de test

Pour évaluer l'impact de la taille des `batches`, nous avons testé le réseau avec une taille de `batch_size = 4` et après avec `batch_size = 1`. Cela signifie que chaque exemple est traité individuellement lors de l'entraînement et de l'évaluation.

Les résultats obtenus pour `batch_size = 1` sont :

```
10000/10000 - 11s - 1ms/step - accuracy: 0.9493 - loss: 0.0088
[0.008800952695310116, 0.9492999911308289]
```

Figure 4: Résultats obtenus après compilation et entraînement

Comparaison avec `batch_size = 4`

- **Précision (`accuracy`) :**
  - Avec `batch_size = 1` : 94.93%
  - Avec `batch_size = 4` : 95.07%

  La précision est légèrement inférieure avec `batch_size = 1`.

- **Perte (`loss`) :**
  - Avec `batch_size = 1` : 0.0088
  - Avec `batch_size = 4` : 0.0084

  La perte est légèrement plus élevée avec `batch_size = 1`, ce qui peut indiquer un apprentissage un peu moins optimal.

- **Temps d'exécution :**
  - Avec `batch_size = 1` : 11 secondes
  - Avec `batch_size = 4` : 4 secondes

  Le traitement individuel des exemples (taille de batch = 1) est beaucoup plus lent, car chaque exemple nécessite une mise à jour distincte des poids.

Conclusion

Une taille de batch plus grande (`batch_size = 4`) permet une mise à jour plus efficace des poids, entraînant une meilleure précision et une exécution plus rapide.

3.9 Prédictions

Le réseau a été utilisé pour effectuer des prédictions sur le dataset de test. Le résultat obtenu :

```
4 , 4
```

Interprétation du résultat :

- Le label réel (`label_test[i]`) est **4**.
- La prédiction du modèle (`output.argmax()`) est également **4**.
- Cela indique que le réseau a correctement classé l'image testée pour cet exemple 4.

4 Un premier réseau profond

Les données du dataset MNIST Fashion ont été chargées et normalisées.
Les dimensions obtenues sont :

- Ensemble d'entraînement : (60000, 28, 28)
- Ensemble de test : (10000, 28, 28)

Pour préparer les images pour le réseau convolutif, une dimension a été ajoutée.
Les nouvelles dimensions sont :

- Ensemble d'entraînement : (60000, 28, 28, 1)
- Ensemble de test : (10000, 28, 28, 1)

Visualisation

Un échantillon d'images a été affiché pour vérifier le dataset.

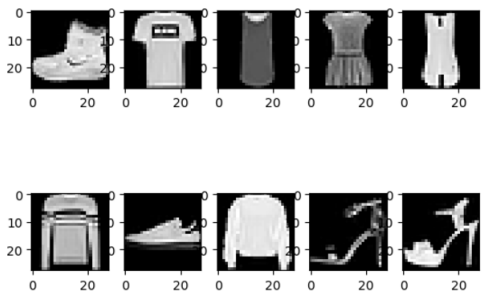


Figure 5: Résultats obtenus de visualisation

4.1 Architecture du modèle

Un modèle séquentiel a été créé avec l'architecture suivante :

- Une couche `Conv2D` avec 16 filtres de taille (3x3), activation ReLU.
- Une couche `MaxPooling2D` avec un pool de taille (2x2).
- Une couche `Conv2D` avec 32 filtres de taille (3x3), activation ReLU.
- Une couche `MaxPooling2D` avec un pool de taille (2x2).
- Une couche `Conv2D` avec 64 filtres de taille (3x3), activation ReLU.
- Une couche `Flatten`.
- Deux couches denses (`Dense`) de 512 neurones chacune, activation ReLU.
- Une couche de sortie (`Dense`) avec 10 neurones, activation ReLU.

Le modèle a été compilé avec la fonction de coût `CategoricalCrossentropy` et l'optimiseur Adam (taux d'apprentissage = 0.001). La structure du réseau est la suivante :

Construction du Réseau Séquentiel

Le nombre total de paramètres entraînables dans ce modèle est de **16,151,306**.

4.1.1 Analyse des paramètres

1. **Première couche convolutive (Conv2D avec 16 filtres de taille 3x3)** Le nombre de paramètres est donné par la formule :

$$\text{Paramètres} = (\text{taille filtre}) \times (\text{canaux d'entrée}) \times (\text{nombre de filtres}) + \text{biais}$$

$$= (3 \times 3) \times 1 \times 16 + 16 = 160$$

2. **Deuxième couche convolutive (Conv2D avec 32 filtres de taille 3x3)**

$$\text{Paramètres} = (3 \times 3) \times 16 \times 32 + 32 = 4,640$$

3. **Troisième couche convolutive (Conv2D avec 64 filtres de taille 3x3)**

$$\text{Paramètres} = (3 \times 3) \times 32 \times 64 + 64 = 18,496$$

4. **Couche dense (512 neurones, après Flatten)** La sortie de la couche `Flatten` est un vecteur de taille $22 \times 22 \times 64 = 30,976$.

$$\text{Paramètres} = (30,976 \times 512) + 512 = 15,860,224$$

5. **Deuxième couche dense (512 neurones)**

$$\text{Paramètres} = (512 \times 512) + 512 = 262,656$$

6. **Couche de sortie (10 neurones)**

$$\text{Paramètres} = (512 \times 10) + 10 = 5,130$$

4.1.2 Total des paramètres "Trainable"

En additionnant tous les paramètres, on obtient :

$$160 + 4,640 + 18,496 + 15,860,224 + 262,656 + 5,130 = 16,151,306$$

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 16)	160
conv2d_1 (Conv2D)	(None, 24, 24, 32)	4,640
conv2d_2 (Conv2D)	(None, 22, 22, 64)	18,496
flatten (Flatten)	(None, 30976)	0
dense (Dense)	(None, 512)	15,860,224
dense_1 (Dense)	(None, 512)	262,656
dense_2 (Dense)	(None, 10)	5,130

Total params: 16,151,306 (61.61 MB)
Trainable params: 16,151,306 (61.61 MB)
Non-trainable params: 0 (0.00 B)

Figure 6: Résultats obtenus après création de l'architecture

4.1.3 Taille et raisonnement

Le nombre de paramètres est significatif. Le réseau est bien dimensionné pour traiter le dataset MNIST Fashion tout en maintenant un bon équilibre entre complexité et capacité d'entraînement.

Résultats

4.2 Entraînement du modèle

Le modèle a été entraîné sur 20 epochs avec une taille de batch de 16. Le taux de réussite (`accuracy`) et la perte (`loss`) ont été suivis.

4.3 Mesure des performances du réseau

Une fois le réseau entraîné, nous avons évalué ses performances sur le dataset de test.

Les résultats obtenus sont les suivants :

```
625/625 - 2s - 3ms/step - accuracy: 0.6690 - loss: 4.9953
[4.995344638824463, 0.6690000295639038]
```

Figure 7: Résultats obtenus après compilation et entraînement

Le taux de réussite (*accuracy*) du réseau est de **66.90%**, ce qui indique que le modèle a correctement classé environ 67% des images du dataset de test.

Conclusion

Le réseau convolutif implémenté atteint une précision de 66.9% sur le dataset de test MNIST Fashion. Cette performance est raisonnable pour un modèle de cette taille.

5 Continuons avec le transfer learning

5.1 Chargement et préparation des données

Le dataset contenant les images de chats et de chiens a été récupéré en ligne, décompressé, et structuré sous la forme d'une arborescence exploitable avec la classe `ImageDataGenerator` de Keras.

Résultats :

- Ensemble d'entraînement : 2000 images appartenant à 2 classes
- Ensemble de validation : 1000 images appartenant à 2 classes

5.2 Chargement du modèle VGG16 complet

Le modèle `VGG16` pré-entraîné sur ImageNet est chargé avec ses poids, et ses couches entièrement connectées sont incluses.

Nombre total de paramètres entraînables : 138,357,544.
Cette taille est trop importante pour être réentraînée sur notre jeu de données limité.

Model: "vgg16"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147,584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295,168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590,080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102,764,544
fc2 (Dense)	(None, 4096)	16,781,312
predictions (Dense)	(None, 1000)	4,097,000

Total params: 138,357,544 (527.79 MB)
Trainable params: 138,357,544 (527.79 MB)
Non-trainable params: 0 (0.00 B)

Figure 8: Résultats obtenus des paramètres après chargement du modèle

5.3 Prédictions avec le modèle complet

Une image de test est chargée, mise à l'échelle, et une prédiction est effectuée. Exemple avec une image d'un chat :

Les prédictions renvoient une des 1000 classes d'ImageNet. Par exemple, la classe 281 correspond à *tabby cat*.

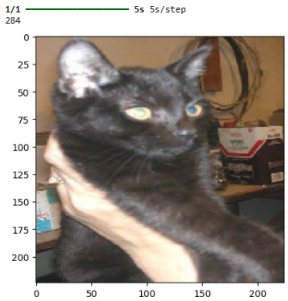


Figure 9: Résultats obtenus de visualisation

5.4 Utilisation de VGG16 tronqué

Pour adapter le modèle à notre problème spécifique, seules les couches convolutives du réseau sont utilisées (`include_top=False`), éliminant ainsi la partie *fully-connected* d'origine.

Observation :

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 ━━━━━━━━━━━━━━━━━━━━ 4s 0us/step
Model: "vgg16"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147,584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295,168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590,080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0

Total params: 14,714,688 (56.13 MB)
Trainable params: 14,714,688 (56.13 MB)
Non-trainable params: 0 (0.00 B)

Figure 10: Résultats obtenus après l'utilisation de VGG16 tronqué

- Le réseau est tronqué après le bloc `block5_pool`.
- La taille de sortie devient `(None, 7, 7, 512)`, correspondant à une extraction de caractéristiques de haut niveau.

Paramètres totaux : 14,714,688, ce qui réduit considérablement la taille et rend le réseau plus adapté au *transfer learning*.

5.5 Gel des couches convolutives du modèle

Pour l'extraction de caractéristiques, toutes les couches convolutives du modèle tronqué VGG16 ont été gelées.

Observation : Après cette opération, tous les paramètres du modèle sont passés en *Non-trainable*, ce qui signifie qu'ils ne seront pas mis à jour lors de l'entraînement. Cela garantit que les poids des couches convolutives pré-entraînées restent inchangés.

5.6 Ajout des couches fully-connected manquantes

Pour compléter le réseau, on a ajouté une architecture entièrement connectée avec:

- Une couche `Flatten` pour transformer la sortie tridimensionnelle de la couche convolutive en vecteur.
- Une couche dense avec 50 neurones et une activation ReLU.

- Une couche dense de sortie avec 2 neurones (correspondant aux classes *cat* et *dog*) et une activation SoftMax.

5.6.1 Résumé du modèle

La structure du modèle est présentée ci-dessous :

Résumé du modèle VGG16 tronqué :

- **Paramètres non-entrainables :** 14,714,688.
- **Taille des sorties :** `(None, 7, 7, 512)` après la couche `block5_pool`.

Résumé du modèle complet :

- **Paramètres totaux :** 15,969,240.
- **Paramètres entraînables :** 1,254,552 (issus des couches fully-connected ajoutées).
- **Paramètres non-entraînables :** 14,714,688.

Model: "sequential"

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 7, 7, 512)	14,714,688
flatten (Flatten)	(None, 25088)	0
dense (Dense)	(None, 50)	1,254,450
dense_1 (Dense)	(None, 2)	102

Total params: 15,969,240 (60.92 MB)
Trainable params: 1,254,552 (4.79 MB)
Non-trainable params: 14,714,688 (56.13 MB)

Figure 11: Résultats obtenus des paramètres du modèle VGG16 tronqué

5.7 Compilation et entraînement du modèle

Le modèle a été compilé avec :

- Une fonction de perte : `CategoricalCrossentropy`
- Un optimiseur : `Adam` avec un taux d'apprentissage de 0.001

5.7.1 Performances obtenues

- **Perte :** 0.2271
- **Précision :** 91.2%

Ces résultats sont impressionnants en tenant compte de la taille limitée du dataset (2000 images).

5.8 Utilisation du modèle pour les prédictions

En ajoutant une architecture fully-connected simple, le modèle atteint une précision impressionnante de 91.2% sur un dataset réduit. Cette méthodologie montre l'impact du gel des poids et des architectures modulaires.

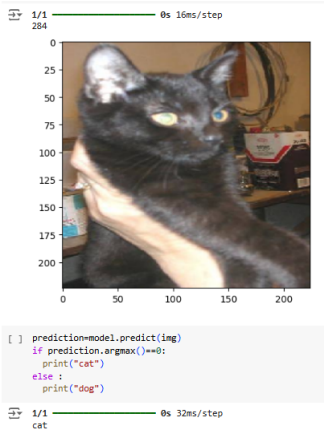


Figure 12: Résultats obtenus après compilation et entraînement du modèle

6 Construisons un GAN générateur d'images

Étapes

6.1 Préparation des données

- Le dataset MNIST est chargé et normalisé (valeurs entre 0 et 1).
- Les images sont redimensionnées avec une profondeur de 1 (grayscale).

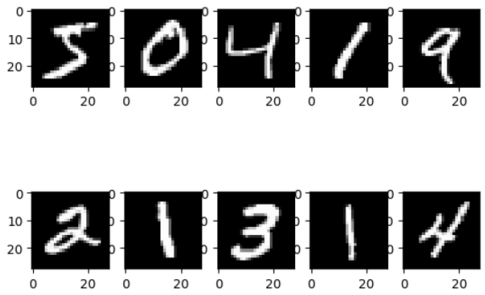


Figure 13: Résultats obtenus de visualisation de données

6.2 Définition des modèles

- **Discriminateur :** Un réseau convolutif détecte si une image est "réelle" ou "fausse" via deux couches convolutionnelles suivies de couches denses. Il est compilé avec la fonction de coût *BinaryCrossEntropy*.
- **Générateur :** Le générateur part d'un espace latent aléatoire (vecteurs de 100 dimensions) et agrandit progressivement des images jusqu'à une taille de 28x28 pixels grâce à des convolutions transposées.
- **GAN complet :** Le générateur et le discriminateur sont combinés, avec le discriminateur figé durant l'entraînement du générateur.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 14, 14, 64)	640
leaky_re_lu (LeakyReLU)	(None, 14, 14, 64)	0
conv2d_1 (Conv2D)	(None, 7, 7, 64)	36,928
leaky_re_lu_1 (LeakyReLU)	(None, 7, 7, 64)	0
flatten (Flatten)	(None, 3136)	0
dense (Dense)	(None, 1)	3,137

Total params: 40,705 (159.00 KB)
Trainable params: 40,705 (159.00 KB)
Non-trainable params: 0 (0.00 B)

Figure 14: Résultats obtenus de paramètres après définition du modèle 1

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 6272)	633,472
leaky_re_lu_2 (LeakyReLU)	(None, 6272)	0
reshape (Reshape)	(None, 7, 7, 128)	0
conv2d_transpose (Conv2DTranspose)	(None, 14, 14, 128)	262,272
leaky_re_lu_3 (LeakyReLU)	(None, 14, 14, 128)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 28, 28, 128)	262,272
leaky_re_lu_4 (LeakyReLU)	(None, 28, 28, 128)	0
conv2d_2 (Conv2D)	(None, 28, 28, 1)	6,273

Total params: 1,164,289 (4.44 MB)
Trainable params: 1,164,289 (4.44 MB)
Non-trainable params: 0 (0.00 B)

Figure 15: Résultats obtenus de paramètres après définition du modèle 2

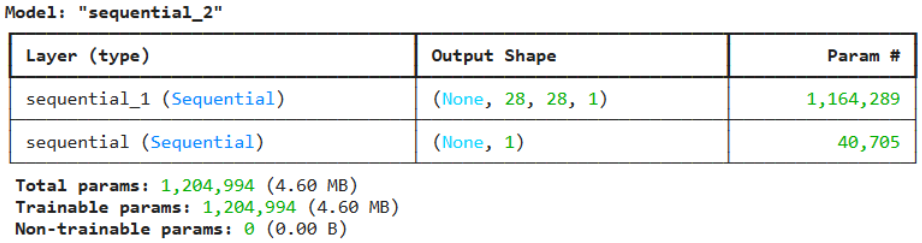
Entraînement

- Alternance entre :
  - L'entraînement du discriminateur avec un mélange d'images réelles et générées.
  - L'entraînement du générateur via le GAN complet, en forçant le générateur à produire des images suffisamment convaincantes pour tromper le discriminateur.
- Utilisation de petits batchs pour un apprentissage progressif (par défaut, batch de 256).
- Génération périodique d'un échantillon d'images pour suivre l'évolution des performances du générateur.

Résultats

6.3 Structure des réseaux

- **Discriminateur :** Total de 40,705 paramètres (taille réduite pour minimiser la surcharge mémoire).
- **Générateur :** Total de 1,164,289 paramètres, conçu pour transformer un vecteur aléatoire en une image de 28x28 pixels.
- **GAN complet :** Total de 1,204,994 paramètres combinés.



Model: "sequential_2"

Layer (type)	Output Shape	Param #
sequential_1 (Sequential)	(None, 28, 28, 1)	1,164,289
sequential (Sequential)	(None, 1)	40,705

Total params: 1,204,994 (4.60 MB)
Trainable params: 1,204,994 (4.60 MB)
Non-trainable params: 0 (0.00 B)

Figure 16: Résultats de la structure de réseaux

6.4 Évolution de l'apprentissage

- Initialement, les images générées par le générateur sont floues et aléatoires.
- Après plusieurs epochs, les images prennent une forme plus définie et commencent à ressembler aux chiffres manuscrits.
- Le discriminateur montre une amélioration dans sa capacité à distinguer les images réelles des fausses.

6.5 Problème rencontré

- La session Colab a planté en raison d'une saturation de la mémoire RAM. Cela a interrompu l'entraînement avant l'achèvement de toutes les epochs.

Observations et explications

- **Qualité des résultats :** Avec seulement 10 epochs, les images générées commencent à ressembler à des chiffres, mais elles restent bruitées. Une augmentation du nombre d'epochs améliorerait probablement la qualité.
- **Saturation mémoire :** Le crash est dû à la taille des batchs (256), qui engendre une forte demande de RAM, combinée à la complexité des réseaux.

- **Alternance de l'entraînement :** Entraîner le discriminateur et le générateur de manière alternée s'est avéré efficace. Cela a permis au GAN de progresser simultanément dans les deux directions : le discriminateur devient meilleur pour détecter les "fakes", tandis que le générateur devient plus habile à les tromper.

Figure 17: Résultats obtenus

Améliorations possibles

- Augmenter le nombre d'epochs et réduire la taille des batchs pour mieux gérer la mémoire.
- Sauvegarder périodiquement les modèles pour éviter de perdre les progrès.
- Utiliser un environnement avec plus de RAM ou des outils spécialisés comme *TensorFlow Dataset* pour charger les données par lot.