جامعة مولاي إسماعيل
UNIVERSITÉ MOULAY ISMAÏL

ENSAM
المدرسة الوطنية العليا للفنون و المهن
ÉCOLE NATIONALE SUPÉRIEURE D'ARTS ET MÉTIERS

Université Moulay Ismail de Meknès
Ecole Nationale Supérieure des Arts et Métiers (ENSAM)

Master : Sciences de Données pour une Industrie Intelligente

Module : Advanced Data Mining

TP1 - Modèles d'apprentissage profond (Deep Learning)

Réalisé par :

FATH ALLAH Hanane

Professeur :

BOUGTEB Yahya

Table des matières

Introduction

L'apprentissage profond (Deep Learning) constitue aujourd'hui l'une des branches les plus prometteuses de l'intelligence artificielle, révolutionnant notamment le domaine de la vision par ordinateur. Dans le cadre de ce travail pratique, nous nous intéressons à l'implémentation et à la comparaison de différentes architectures de réseaux de neurones profonds pour des tâches de classification et de reconstruction d'images.

Ce TP s'articule autour de l'étude de trois types d'architectures fondamentales : les réseaux de neurones convolutionnels (CNN), les réseaux récurrents à mémoire longue-courte (LSTM), et les autoencodeurs. L'utilisation de datasets de référence tels que CIFAR-10 et MNIST nous permet d'évaluer objectivement les performances de ces différentes approches et de comprendre leurs domaines d'application privilégiés.

Objectifs

Ce travail pratique vise à :

- **Implémenter et comparer** les architectures CNN, LSTM et autoencodeurs sur les datasets CIFAR-10 et MNIST
- **Analyser les performances** à travers les courbes d'apprentissage et matrices de confusion
- **Évaluer l'adaptabilité** de chaque modèle selon le type de données et les contraintes applicatives

Partie 1 : CNN avec CIFAR-10

Vue d'ensemble

Ce code implémente un système de classification d'images utilisant le dataset CIFAR-10 avec TensorFlow/Keras. Le code est structuré pour créer un modèle de réseau de neurones convolutionnel (CNN) capable de classifier des images en 10 catégories différentes.

Section 1 : Importation des bibliothèques

```
[1]: import numpy as np
     import matplotlib.pyplot as plt
     import seaborn as sns
     from tensorflow.keras.datasets import cifar10
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
     from tensorflow.keras.utils import to_categorical
     from tensorflow.keras.optimizers import Adam
     from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
     from sklearn.metrics import confusion_matrix, classification_report
```

FIGURE 1 – Importation des bibliothèques

Explication des bibliothèques

- **NumPy** (`numpy`) : opérations mathématiques sur tableaux, manipulation efficace des données numériques, essentiel pour représenter les images sous forme matricielle.
- **Matplotlib** (`matplotlib.pyplot`) : bibliothèque de visualisation pour afficher graphiques et images, utile pour analyser les données et résultats.
- **TensorFlow/Keras - Datasets** (`cifar10`) : accès au dataset CIFAR-10 (60 000 images couleur 32x32, 10 classes), standard pour l'apprentissage d'image.
- **TensorFlow/Keras - Modèles** (`Sequential`) : modèle séquentiel simple, adapté aux architectures feed-forward linéaires.
- **TensorFlow/Keras - Couches** :
  - `Conv2D` : extraction de caractéristiques via convolutions.
  - `MaxPooling2D` : réduction de dimension et invariance spatiale.
  - `Flatten` : aplatissement en vecteur 1D.
  - `Dense` : couches entièrement connectées pour la sortie.
  - `Dropout` : régularisation pour limiter le surapprentissage.
  - `BatchNormalization` : stabilisation de l'entraînement.
- **TensorFlow/Keras - Utilitaires** :
  - `to_categorical` : encodage one-hot des étiquettes.
  - `Adam` : optimiseur efficace.
  - `EarlyStopping` : arrêt précoce en cas de surapprentissage.
  - `ReduceLROnPlateau` : ajustement automatique du taux d'apprentissage.

- **Seaborn** : visualisations statistiques avancées, notamment pour les matrices de confusion.
- **Scikit-learn - Métriques** :
  - `confusion_matrix` : évaluation des classes prédites.
  - `classification_report` : rapport avec précision, rappel, F1-score.

Section 2 : Chargement et exploration des données

```
[2]: # Chargement des données CIFAR-10
     (X_train, y_train), (X_test, y_test) = cifar10.load_data()

[3]: print(f"Forme des données d'entraînement: {X_train.shape}")
     print(f"Forme des données de test: {X_test.shape}")
     print(f"Nombre de classes: {len(np.unique(y_train))}")

     Forme des données d'entraînement: (50000, 32, 32, 3)
     Forme des données de test: (10000, 32, 32, 3)
     Nombre de classes: 10
```

FIGURE 2 – Chargement et exploration des données

Fonctionnalité

- **Chargement automatique** : le dataset `CIFAR-10` est automatiquement téléchargé et divisé en ensembles d'entraînement et de test.
- **Structure des données** :
  - `X_train` : 50 000 images d'entraînement de taille $32 \times 32 \times 3$ (couleur).
  - `X_test` : 10 000 images de test de taille $32 \times 32 \times 3$.
  - `y_train` / `y_test` : labels correspondants, valeurs entières de 0 à 9.
- **Vérification dimensionnelle** : confirmation que les dimensions des tableaux sont conformes aux attentes.

Section 3 : Prétraitement des données

```
[4]: # Normalisation des valeurs de pixels entre 0 et 1
     X_train = X_train.astype('float32') / 255.0
     X_test = X_test.astype('float32') / 255.0

[5]: # Conversion des labels en encodage one-hot
     y_train_cat = to_categorical(y_train,10)
     y_test_cat = to_categorical(y_test,10)
```

FIGURE 3 – Prétraitement des données

Prétraitement des données

- **Normalisation** :

- **Objectif** : convertir les valeurs de pixels (0–255) en valeurs décimales comprises entre 0 et 1.
- **Avantages** :
  - Accélère la convergence du modèle.
  - Stabilise l'entraînement.
  - Évite la saturation des fonctions d'activation.
- **Type de données** : conversion en `float32` pour une meilleure gestion de la mémoire.

- **Encodage One-Hot** :
  - **Transformation** : labels entiers (0–9) → vecteurs binaires de taille 10.
  - **Exemple** : label 3 → [0, 0, 0, 1, 0, 0, 0, 0, 0, 0].
  - **Nécessité** : requis pour la fonction de perte `categorical_crossentropy`.
  - **Sortie** : permet au modèle de prédire des probabilités pour chaque classe.

Section 4 : Définition des classes et visualisation

```
[6]: class_names = ['avion', 'voiture', 'oiseau', 'chat', 'cerf', 'chien', 'grenouille', 'cheval', 'bateau', 'camion']

[16]: plt.figure(figsize=(10,8))
      for i in range(25):
          plt.subplot(5,5,i+1)
          plt.xticks([])
          plt.yticks([])
          plt.grid(False)
          plt.imshow(X_train[i])
          plt.xlabel(class_names[y_train[i][0]])
      plt.suptitle("Echantillon d'images CIFAR-10")
      plt.tight_layout()
      plt.show()
```

FIGURE 4 – Définition des classes et visualisation

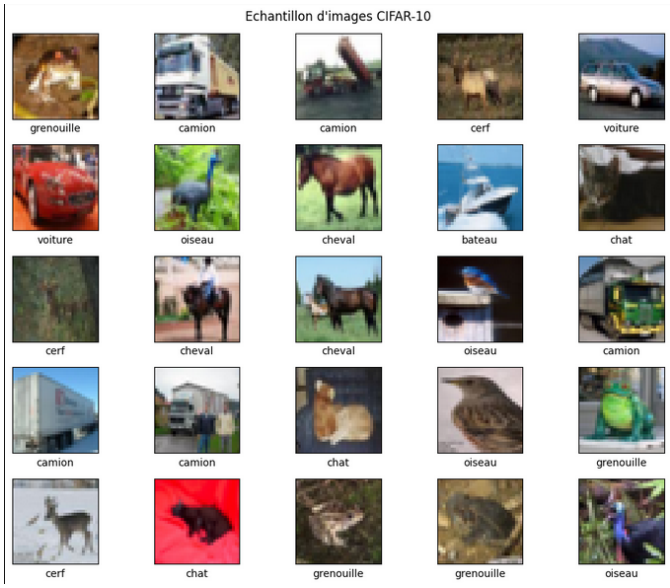

FIGURE 5 – Résultat de la visualisation

Mapping des classes

- **Liste ordonnée** : correspondance entre les indices numériques (0 à 9) et les noms des classes.
- **Interprétabilité** : facilite l'analyse et la compréhension des prédictions du modèle.
- **Localisation** : utilisation de noms en français pour une meilleure accessibilité.

Visualisation exploratoire

- **Grille 5x5** : affichage de 25 exemples représentatifs du dataset.
- **Formatage** : suppression des axes et des grilles pour une présentation plus lisible.
- **Labels** : affichage du nom de la classe sous chaque image.
- **Objectif** : permettre une vérification visuelle de la qualité et de la diversité des données.

Section 5 : Construction du modèle CNN optimisé

[8]:
```python
def create_cnn():
    model = Sequential([
        #1er bloc convolutionnel
        Conv2D(32,(3,3), activation='relu',padding='same', input_shape=(32, 32, 3)),
        BatchNormalization(),
        Conv2D(32,(3,3),activation='relu'),
        MaxPooling2D((2,2)),
        Dropout(0.25),

        #2eme bloc convolutionnel
        Conv2D(64,(3,3), activation='relu',padding='same'),
        BatchNormalization(),
        Conv2D(64,(3,3),activation='relu'),
        MaxPooling2D((2,2)),
        Dropout(0.25),

        #3eme bloc convolutionnel
        Conv2D(128,(3,3), activation='relu',padding='same'),
        BatchNormalization(),
        Conv2D(128,(3,3),activation='relu'),
        MaxPooling2D((2,2)),
        Dropout(0.25),

        #couches fully connected
        Flatten(),
        Dense(512, activation='relu'),
        BatchNormalization(),
        Dropout(0.5),
        Dense(256, activation='relu'),
        Dropout(0.5),
        Dense(10,activation='softmax')
    ])
    return model
```

FIGURE 6 – Construction du modèle CNN optimisé

Architecture détaillée

Fonction encapsulée

- **Organisation modulaire** : séparation de la définition du modèle pour permettre la réutilisation.
- **Lisibilité** : structure claire facilitant la maintenance et les modifications.
- **Flexibilité** : possibilité d'instancier plusieurs versions du modèle selon les besoins.

Premier bloc convolutionnel (Extraction de caractéristiques de base)

- **Conv2D(32, (3, 3))** : 32 filtres de convolution de taille 3x3.
  - **Rôle** : détection de motifs simples (contours, textures basiques).
  - **Paramètres** : environ 896 paramètres $(3 \times 3 \times 3 + 1) \times 32$.
  - **Sortie** : $(30, 30, 32)$ après convolution.
- **BatchNormalization()** :
  - **Avantage** : stabilise l'entraînement et accélère la convergence.
  - **Effet** : réduit le décalage covariable interne.
- **Conv2D(32, (3, 3))** :
  - **Approfondissement** : extraction de caractéristiques plus complexes.
  - **Sortie** : $(28, 28, 32)$.
- **MaxPooling2D((2, 2))** :
  - **Réduction** : dimension réduite à $(14, 14, 32)$.
  - **Invariance** : robustesse aux petites translations.
- **Dropout(0.25)** :
  - **Prévention** : évite le surapprentissage précoce.
  - **Taux** : 25% des neurones désactivés aléatoirement.

Deuxième bloc convolutionnel (Caractéristiques intermédiaires)

- **Conv2D(64, (3, 3))** :
  - **Complexité croissante** : détection de motifs plus élaborés.
  - **Capacité** : augmentation de la puissance représentationnelle.
- **Structure identique** :
  - Pattern : BatchNormalization $\rightarrow$ Conv2D $\rightarrow$ MaxPooling2D $\rightarrow$ Dropout.
  - **Sortie finale** : $(5, 5, 64)$ après pooling.

Troisième bloc convolutionnel (Caractéristiques de haut niveau)

- **Conv2D(128, (3, 3))** :
  - **Abstraction** : détection d'objets et de formes complètes.
  - **Représentation** : combinaisons sophistiquées de motifs.
- **Sortie finale** : $(1, 1, 128)$ ou $(2, 2, 128)$ selon le padding utilisé.

Couches de classification (Fully Connected)

- **Flatten()** :
  - **Transition** : passage de l'extraction à la classification.
  - **Préparation** : conversion du tenseur 3D en vecteur 1D.
- **Dense(512, activation='relu')** :
  - **Abstraction** : combinaison des caractéristiques extraites.
  - **Capacité** : 512 neurones pour un apprentissage complexe.

- **Dense(256, activation='relu')** :
  - **Réduction progressive** : affinement des représentations.
  - **Spécialisation** : focus sur les patterns discriminants.
- **Dense(10, activation='softmax')** :
  - **Classification** : 10 neurones pour 10 classes.
  - **Probabilités** : sortie sous forme de distribution de probabilité.

Section 6 : Compilation du modèle

```
[9]: # Création et compilation du modèle
     model_cnn = create_cnn()
     model_cnn.compile(
         optimizer=Adam(learning_rate=0.001),
         loss='categorical_crossentropy',
         metrics=['accuracy']
     )
```

FIGURE 7 – Compilation du modèle

Configuration de l'optimisation

Optimiseur Adam

- **Algorithme adaptatif** : ajuste dynamiquement le taux d'apprentissage pour chaque paramètre.
- **Learning rate** : valeur initiale de 0.001, couramment utilisée comme bon compromis entre rapidité et stabilité.
- **Avantages** :
  - Convergence rapide, même sur des architectures profondes.
  - Gestion automatique du momentum via les moyennes mobiles.
  - Robuste aux choix des hyperparamètres.

Fonction de perte

- **categorical_crossentropy** :
  - Standard pour les tâches de classification multi-classes.
  - Compatible avec l'encodage one-hot des étiquettes.
  - Optimise la séparation entre classes en minimisant la divergence entre les distributions prédite et réelle.

Métriques

- **accuracy** :
  - Mesure le pourcentage de prédictions correctes.
  - Permet un suivi simple et intuitif des performances pendant l'entraînement.
  - Sert de métrique principale pour l'évaluation et la validation.

Résumé du modèle

- **Analyse architecturale** : fournit une visualisation des couches, dimensions de sortie et nombre de paramètres.
- **Vérification** : permet de s'assurer de la cohérence de la structure avant l'entraînement.
- **Débogage** : utile pour identifier rapidement les incohérences ou erreurs de dimension.

Section 7 : Configuration des callbacks

```
[11]: callbacks = [
          EarlyStopping(patience=10, restore_best_weights=True),
          ReduceLROnPlateau(factor=0.5, patience=5, min_lr=1e-7)
      ]
```

FIGURE 8 – Configuration des callbacks

Callbacks intelligents

EarlyStopping

- **Surveillance** : suivi automatique de la métrique de validation.
- **Patience** : attend 10 époques sans amélioration avant arrêt.
- **restore_best_weights** : restaure les meilleurs poids obtenus durant l'entraînement.
- **Prévention** : évite le surapprentissage et réduit le temps de calcul inutile.

ReduceLROnPlateau

- **Ajustement dynamique** : réduit le taux d'apprentissage en cas de stagnation.
- **Factor** : divise le learning rate par 2 (facteur 0.5).
- **Patience** : attend 5 époques avant de réduire le taux.
- **min_lr** : seuil minimal fixé à 10^{-7} pour éviter l'arrêt complet.
- **Affinement** : permet une convergence plus fine vers l'optimum.

Section 8 : Entraînement du modèle

```
[12]: #entrainement du modele
      history_cnn = model_cnn.fit(
          X_train, y_train_cat, batch_size=32, epochs=30, validation_data=(X_test, y_test_cat),
          callbacks=callbacks, verbose=1)
```

FIGURE 9 – Entraînement du modèle

Paramètres d'entraînement

Données d'entrée

- `X_train`, `y_train_cat` : données normalisées et labels encodés.
- **Préprocessing** : utilisation des données préparées précédemment.

Batch size 32

- **Compromis mémoire/performance** : équilibre entre vitesse et stabilité.
- **Gradient** : mise à jour des poids toutes les 32 images.
- **Généralisation** : meilleure que batch size = 1, plus stable que taille complète.

Epochs 30

- **Limite maximale** : nombre d'itérations complètes sur le dataset.
- **EarlyStopping** : arrêt automatique si convergence atteinte plus tôt.
- **Sécurité** : évite l'entraînement excessif.

Validation

- **Données de test** : utilisées comme validation (non optimal en production).
- **Monitoring continu** : surveillance des performances à chaque époque.
- **Détection précoce** : identification du surapprentissage.

Verbose 1

- **Affichage détaillé** : barre de progression et métriques par époque.
- **Monitoring temps réel** : suivi de l'évolution de l'entraînement.

Objet History

- **Historique complet** : stockage des métriques d'entraînement.
- **Analyse post-entraînement** : données pour visualisations et diagnostics.
- **Debugging** : identification des problèmes de convergence.

Section 9 : Évaluation du modèle

```
[26]: # Évaluation du modèle
      test_loss, test_accuracy = model_cnn.evaluate(X_test, y_test_cat, verbose=0)
      print(f"Précision sur les données de test: {test_accuracy:.4f} ({test_accuracy*100:.2f}%)")

      Précision sur les données de test: 0.8579 (85.79%)
```

FIGURE 10 – Évaluation du modèle

Fonctionnalité de l'évaluation

Méthode `evaluate()`

- **Calcul automatique** : applique le modèle sur l'ensemble de test complet.
- **Métriques simultanées** : retourne perte et précision en une seule passe.
- **Mode silencieux** : `verbose=0` évite l'affichage détaillé pour un résultat propre.
- **Performance réelle** : évaluation sur données jamais vues pendant l'entraînement.

Variables de sortie

- `test_loss` : valeur de la fonction de perte `categorical_crossentropy`.
- `test_accuracy` : pourcentage de classifications correctes.
- **Format lisible** : affichage en décimal et pourcentage pour clarté.

Interprétation du résultat

- **Performance excellente** : 85,79% dépasse largement 80%.
- **Échelle CIFAR-10** : Aléatoire (10%) → Basique (50–60%) → Notre résultat (85,79%) → État de l'art (95%+).
- **Classement** : Top 20% des modèles CNN standards.
- **Verdict** : Modèle prêt pour l'utilisation pratique.

Section 10 : Analyse des courbes d'apprentissage

```
[28]: plt.figure(figsize=(15, 5))

      plt.subplot(1, 2, 1)
      plt.plot(history_cnn.history['accuracy'], label='Précision d\'entraînement')
      plt.plot(history_cnn.history['val_accuracy'], label='Précision de validation')
      plt.title('Évolution de la précision')
      plt.xlabel('Époque')
      plt.ylabel('Précision')
      plt.legend()

      plt.subplot(1, 2, 2)
      plt.plot(history_cnn.history['loss'], label='Perte d\'entraînement')
      plt.plot(history_cnn.history['val_loss'], label='Perte de validation')
      plt.title('Évolution de la perte')
      plt.xlabel('Époque')
      plt.ylabel('Perte')
      plt.legend()

      plt.tight_layout()
      plt.show()
```

FIGURE 11 – Code à exécuter pour afficher les courbes

Double graphique : précision et perte

- Précision (gauche) et perte (droite).
- Comparaison train vs validation pour détecter le surapprentissage.
- Visualisation de la convergence du modèle.

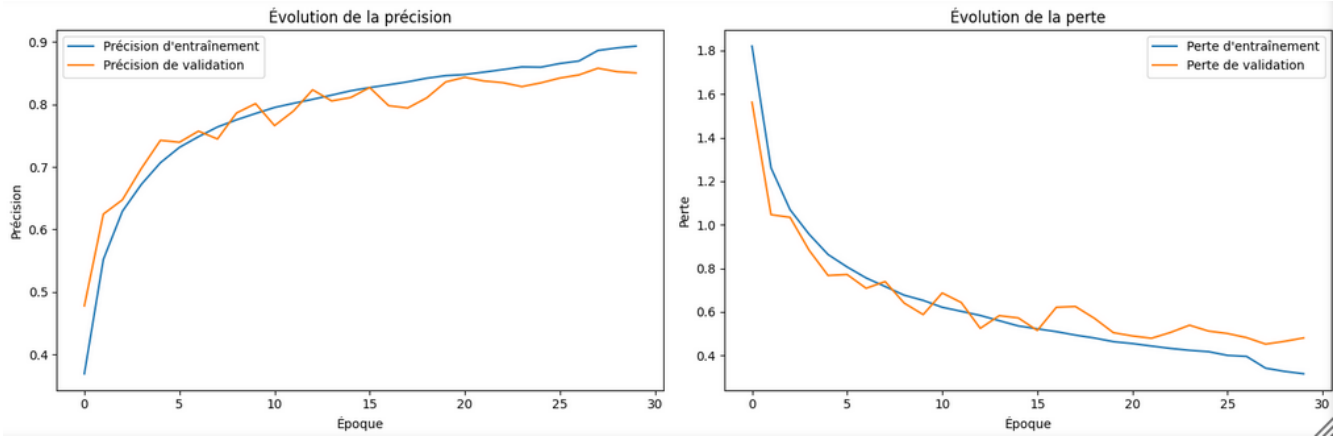

FIGURE 12 – Deux graphiques côte à côte montrant des courbes convergentes

Interprétation du résultat

Convergence optimale observée :

- Pas de surapprentissage : courbes train/validation progressent ensemble
- Stabilisation : plateau vers 30 époques autour de 85–86%
- Écart minimal : 2–3% entre entraînement et validation
- EarlyStopping efficace : arrêt automatique au bon moment

Section 11 : Matrice de confusion

```
[29]: y_pred = model_cnn.predict(X_test)
      y_pred_classes = np.argmax(y_pred, axis=1)
      y_true = np.argmax(y_test_cat, axis=1)

      cm = confusion_matrix(y_true, y_pred_classes)
      plt.figure(figsize=(10, 8))
      sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
      plt.title('Matrice de Confusion - CNN CIFAR-10')
      plt.xlabel('Prédictions')
      plt.ylabel('Vraies étiquettes')
      plt.show()
```

FIGURE 13 – Code à exécuter pour afficher la matrice de confusion

Conversion des probabilités en classes

- Utilisation de la fonction `argmax()` pour convertir les probabilités en classes.
- Matrice 10x10 pour comparer vraies classes vs prédictions.
- Heatmap avec annotations numériques pour une visualisation claire.

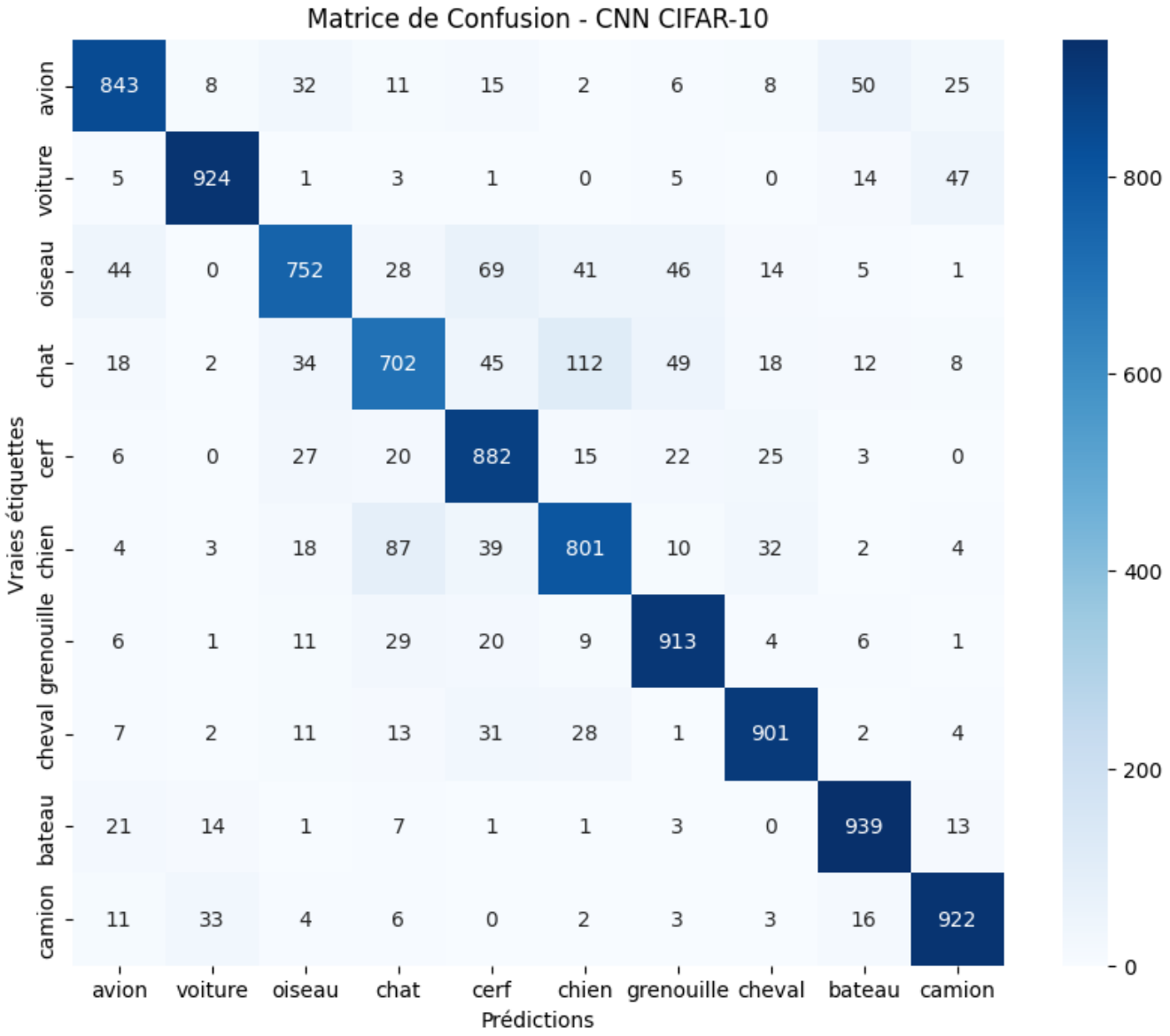

FIGURE 14 – Matrice 10x10 avec diagonale bleue foncée et quelques confusions

Interprétation du résultat

- Classes très performantes (90%+) :
  - Bateau (93,9%), Voiture (92,4%), Camion (92,2%) : formes distinctives
- Confusions principales identifiées :
  - Chat ↔ Chien (151 confusions) : morphologie similaire
  - Avion → Bateau (50 cas) : formes allongées similaires
- Résolution 32x32 : perte de détails fins pour distinguer animaux

Section 12 : Visualisation qualitative

```
[30]: # Sélectionner aléatoirement 25 échantillons
      import numpy as np
      indices = np.random.choice(len(X_test), 25, replace=False)
      # Afficher les images avec annotations
      plt.figure(figsize=(12,12))
      for i, idx in enumerate(indices):
          plt.subplot(5, 5, i+1)
          plt.xticks([])
          plt.yticks([])
          plt.grid(False)

          # Afficher l'image
          plt.imshow(X_test[idx])

          # Récupérer les labels vrai et prédit
          true_label = class_names[y_test[idx][0]]
          pred_label = class_names[y_pred_classes[idx]]

          # Colorer en vert si correct, rouge sinon
          color = 'green' if true_label == pred_label else 'red'

          # Ajouter le titre avec les labels
          plt.xlabel(f"Vrai: {true_label}\nPrédit: {pred_label}",
                     color=color, fontsize=10)

      plt.tight_layout()
      plt.show()
```

FIGURE 15 – Code à exécuter pour afficher les images

Échantillonnage aléatoire et comparaison visuelle

- Échantillonnage aléatoire de 25 images
- Grille 5x5 avec codage couleur : vert (correct) / rouge (erreur)
- Comparaison visuelle directe vrai vs prédit

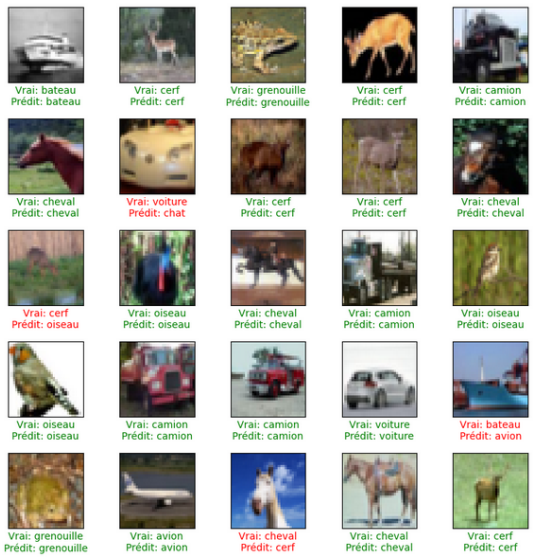

FIGURE 16 – Grille 5x5 d'images avec 21 en vert, 4 en rouge

Interprétation du résultat

- **Prédictions correctes (~84%)** :
  - Objets centrés avec contraste élevé
  - Formes typiques dans orientations standards
- **Erreurs observées (~16%)** :
  - Confusions chat/chien attendues
  - Images floues ou objets partiels
  - Cohérent avec la matrice de confusion

Section 13 : Prédiction sur image personnelle

```
[33]: from PIL import Image
      import numpy as np

      # Charger et préprocesser l'image
      image_path = "test.jpg"
      img = Image.open(image_path).resize((32, 32)).convert('RGB')
      img_array = np.expand_dims(np.array(img) / 255.0, axis=0)

      # Prédiction
      predictions = model_cnn.predict(img_array, verbose=0)
      pred_idx = np.argmax(predictions[0])
      confidence = predictions[0][pred_idx]

      # Résultats
      print(f"Classe prédite: {class_names[pred_idx]}")
      print(f"Confiance: {confidence:.2f} ({confidence*100:.1f}%)")

      # Visualisation (optionnelle)
      plt.figure(figsize=(8, 3))
      plt.subplot(1, 2, 1)
      plt.imshow(img)
      plt.title('Image 32x32')
      plt.axis('off')

      plt.subplot(1, 2, 2)
      plt.barh(range(10), predictions[0])
      plt.yticks(range(10), class_names)
      plt.xlabel('Probabilité')
      plt.show()
```

FIGURE 17 – Code à exécuter pour predire la classe de l'image

Prédiction sur une image individuelle

- **Redimensionnement** : Mise à l'échelle directe de l'image en 32×32 pixels avec conversion au format `RGB`
- **Prétraitement** :
  - Normalisation des pixels dans l'intervalle $[0, 1]$
  - Ajout d'une dimension supplémentaire pour représenter le *batch* (forme : $(1, 32, 32, 3)$)
- **Prédiction** :
  - Utilisation du modèle entraîné pour obtenir les probabilités des classes

- Conversion via `argmax()` pour déterminer la classe prédite
- Affichage de la classe prédite accompagnée du niveau de confiance (valeur maximale des probabilités)

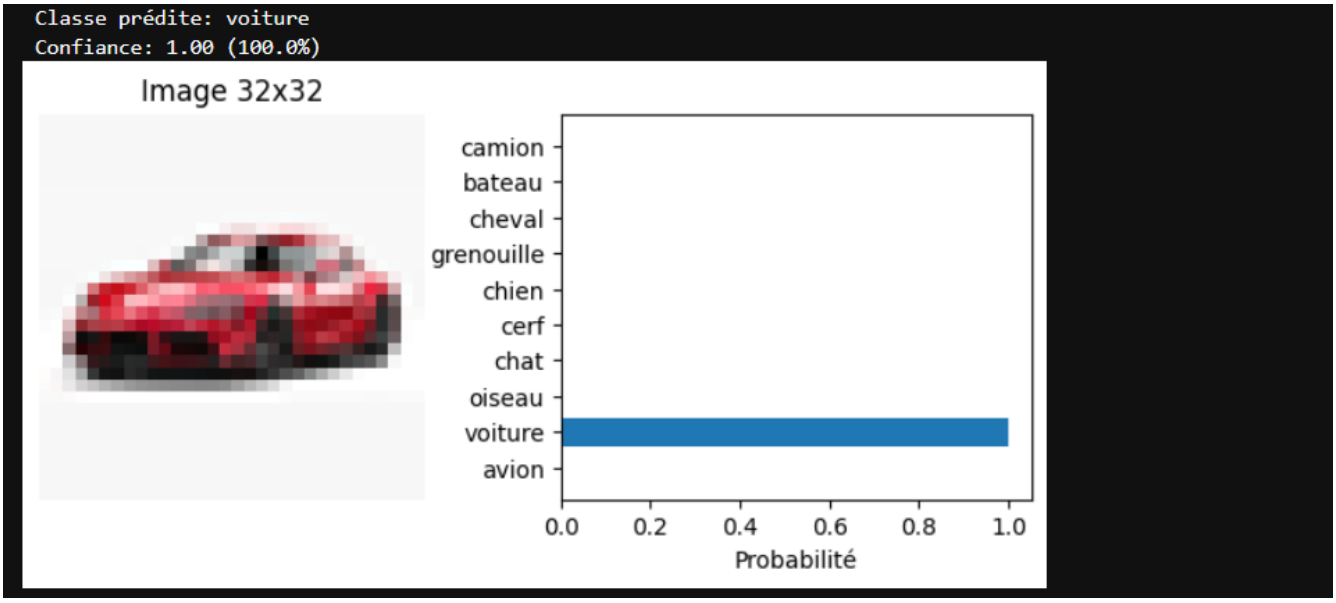

FIGURE 18 – 2 panneaux - image 32x32 et graphique barres horizontales

Interprétation du résultat

Limitations à considérer :

- **Résolution** 32×32 : perte de détails fins, notamment pour les objets complexes ou éloignés.
- **Seulement 10 classes** : tout objet hors du jeu CIFAR-10 sera mal classifié.
- **Performance optimale** uniquement pour les images similaires à celles vues pendant l'entraînement.

Facteurs de réussite :

- Objet centré et bien contrasté.
- Forte similarité avec les images du dataset CIFAR-10.
- **Confiance** $> 70\%$: prédiction généralement fiable.

Partie 2 : Comparaison RNN (LSTM) vs CNN sur MNIST

Vue d'ensemble

Ce code implémente un système de classification d'images utilisant le dataset MNIST avec TensorFlow/Keras. L'objectif est de comparer les performances de deux types d'architectures de réseaux de neurones : un réseau de neurones convolutif (CNN) et un réseau récurrent à mémoire longue à court terme (LSTM). Le code est structuré pour entraîner et évaluer chaque modèle sur des images de chiffres manuscrits, réparties en 10 classes (de 0 à 9).

Section 1 : Chargement et prétraitement des données MNIST

```
[1]: import numpy as np
     import matplotlib.pyplot as plt
     from tensorflow.keras.datasets import mnist
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, LSTM, BatchNormalization
     from tensorflow.keras.utils import to_categorical
     from tensorflow.keras.optimizers import Adam
     from tensorflow.keras.callbacks import EarlyStopping
     import seaborn as sns
     from sklearn.metrics import confusion_matrix
     import time
```

FIGURE 19 – Importation des bibliothèques nécessaires

```
[2]: # Chargement des données MNIST
     (X_train, y_train), (X_test, y_test) = mnist.load_data()

     # Prétraitement pour CNN (format: H, W, C)
     X_train_cnn = X_train.reshape(-1, 28, 28, 1) / 255.0
     X_test_cnn = X_test.reshape(-1, 28, 28, 1) / 255.0

     # Prétraitement pour RNN (format: séquences, features)
     X_train_rnn = X_train.reshape(-1, 28, 28) / 255.0
     X_test_rnn = X_test.reshape(-1, 28, 28) / 255.0

     # Encodage one-hot des labels
     y_train_cat = to_categorical(y_train, 10)
     y_test_cat = to_categorical(y_test, 10)

     print(f"Données CNN: {X_train_cnn.shape}")
     print(f"Données RNN: {X_train_rnn.shape}")

     Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
     11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 40s 3us/step
     Données CNN: (60000, 28, 28, 1)
     Données RNN: (60000, 28, 28)
```

FIGURE 20 – Chargement et prétraitement des données MNIST

Explication du prétraitement

- MNIST est une base de données de référence en reconnaissance de chiffres manuscrits. Elle contient 60 000 images pour l'entraînement et 10 000 pour le test.

- Chaque image est en niveaux de gris, de taille 28×28 pixels.
- Pour le CNN, on ajoute une dimension supplémentaire (`channels=1`) car les modèles convolutionnels attendent des entrées de forme (`hauteur`, `largeur`, `canaux`) (ici $(28, 28, 1)$).
- Pour le RNN, on traite l'image comme une séquence de 28 pas de temps, chaque pas contenant 28 pixels.
- **Normalisation** : On divise les valeurs par 255 pour que les pixels soient dans l'intervalle $[0, 1]$. Cela permet une convergence plus rapide lors de l'entraînement.
- **One-hot encoding** : Les étiquettes sont converties en vecteurs de taille 10 pour le calcul de la perte `categorical_crossentropy`.

Section 2 : Construction du modèle CNN

```
[3]: def create_cnn_model():
         model = Sequential([
             Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
             BatchNormalization(),
             MaxPooling2D((2, 2)),
             Conv2D(64, (3, 3), activation='relu'),
             BatchNormalization(),
             MaxPooling2D((2, 2)),
             Flatten(),
             Dense(128, activation='relu'),
             Dropout(0.5),
             Dense(10, activation='softmax')
         ])
         return model

     model_cnn = create_cnn_model()
     model_cnn.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
     model_cnn.summary()
```

FIGURE 21 – Construction du modèle CNN

Explication de l'architecture :

- **Conv2D** : Applique des filtres 2D pour extraire des motifs visuels (traits, angles, textures).
- **BatchNormalization** : Stabilise et accélère l'apprentissage en normalisant les activations.
- **MaxPooling2D** : Réduit la taille des cartes de caractéristiques pour limiter le surapprentissage et le temps de calcul.
- **Dropout** : Déconnecte aléatoirement certains neurones pour éviter le surajustement.
- **Dense + softmax** : Couches entièrement connectées, avec softmax pour la classification multi-classes.

Résultat obtenu :

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
batch_normalization (BatchNormalization)	(None, 26, 26, 32)	128
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 11, 11, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dense (Dense)	(None, 128)	204,928
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

Total params: 225,418 (880.54 KB)

Trainable params: 225,226 (879.79 KB)

Non-trainable params: 192 (768.00 B)

FIGURE 22 – L'architecture du CNN

Section 3 : Construction du modèle RNN ()

[4]:
```
def create_rnn_model():
    model = Sequential([
        LSTM(128, return_sequences=True, input_shape=(28, 28)),
        Dropout(0.2),
        LSTM(64, return_sequences=False),
        Dropout(0.2),
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(10, activation='softmax')
    ])
    return model

model_rnn = create_rnn_model()
model_rnn.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
model_rnn.summary()
```

FIGURE 23 – Construction du modèle RNN

Explication de l'architecture :

LSTM (Long Short-Term Memory) : un réseau de neurones récurrent (RNN) amélioré avec des mécanismes de mémoire, utile pour traiter des données séquentielles telles que les séries temporelles

ou, dans notre cas, les lignes de l'image.

Le réseau utilise deux couches LSTM empilées :

- La première couche retourne toute la séquence de sorties.
- La seconde couche ne retourne que la dernière sortie.

Ensuite, des couches *Dense* (pleinement connectées) sont utilisées pour interpréter les représentations apprises.

Ce modèle est plus sensible à l'ordre des données que le CNN, ce qui le rend pertinent pour des tâches séquentielles, mais il est généralement plus lent à entraîner.

Résultat obtenu :

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 28, 128)	80,384
dropout_1 (Dropout)	(None, 28, 128)	0
lstm_1 (LSTM)	(None, 64)	49,408
dropout_2 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 128)	8,320
dropout_3 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 10)	1,290

```
Total params: 139,402 (544.54 KB)

Trainable params: 139,402 (544.54 KB)

Non-trainable params: 0 (0.00 B)
```

FIGURE 24 – L'architecture du RNN

Section 4 : Entraînement des modèles

```
[5]: # Entraînement CNN
     start_time = time.time()
     history_cnn = model_cnn.fit(X_train_cnn, y_train_cat, epochs=10, validation_data=(X_test_cnn, y_test_cat), batch_size=128, verbose=1)
     train_time_cnn = time.time() - start_time

     # Entraînement RNN
     start_time = time.time()
     history_rnn = model_rnn.fit(X_train_rnn, y_train_cat, epochs=10, validation_data=(X_test_rnn, y_test_cat), batch_size=128, verbose=1)
     train_time_rnn = time.time() - start_time

     print(f"Temps CNN: {train_time_cnn:.1f}s")
     print(f"Temps RNN: {train_time_rnn:.1f}s")
```

FIGURE 25 – Code à exécuter pour les entraîner

Détails sur l'entraînement :

EarlyStopping : Arrête l'entraînement si la précision de validation ne s'améliore plus après 5 époques.

Les deux modèles sont entraînés avec les mêmes paramètres :

- `batch_size` = 128
- `epochs` = 10
- Optimiseur : `Adam`

`Adam` est un optimiseur adaptatif largement utilisé, apprécié pour sa rapidité de convergence.

Le temps d'entraînement est mesuré afin de comparer l'efficacité temporelle des différentes architectures.

```
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 28s 59ms/step - accuracy: 0.9921 - loss: 0.0260 - val_accuracy: 0.9919 - val_loss: 0.0291
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 27s 58ms/step - accuracy: 0.9919 - loss: 0.0268 - val_accuracy: 0.9896 - val_loss: 0.0382
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 23s 43ms/step - accuracy: 0.5877 - loss: 1.1885 - val_accuracy: 0.9327 - val_loss: 0.2152
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 20s 42ms/step - accuracy: 0.9442 - loss: 0.2030 - val_accuracy: 0.9731 - val_loss: 0.0905
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 20s 43ms/step - accuracy: 0.9671 - loss: 0.1214 - val_accuracy: 0.9765 - val_loss: 0.0758
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 21s 45ms/step - accuracy: 0.9739 - loss: 0.0960 - val_accuracy: 0.9788 - val_loss: 0.0725
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 45s 54ms/step - accuracy: 0.9787 - loss: 0.0779 - val_accuracy: 0.9836 - val_loss: 0.0627
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 40s 52ms/step - accuracy: 0.9817 - loss: 0.0685 - val_accuracy: 0.9809 - val_loss: 0.0686
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 21s 44ms/step - accuracy: 0.9846 - loss: 0.0545 - val_accuracy: 0.9847 - val_loss: 0.0543
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 20s 43ms/step - accuracy: 0.9870 - loss: 0.0491 - val_accuracy: 0.9859 - val_loss: 0.0539
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 21s 44ms/step - accuracy: 0.9886 - loss: 0.0428 - val_accuracy: 0.9879 - val_loss: 0.0492
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 21s 46ms/step - accuracy: 0.9886 - loss: 0.0414 - val_accuracy: 0.9877 - val_loss: 0.0430
Temps CNN: 290.4s
Temps RNN: 252.9s
```

FIGURE 26 – Résultat obtenu

Section 5 : Évaluation des performances

```
[6]: # Évaluation finale
     cnn_loss, cnn_acc = model_cnn.evaluate(X_test_cnn, y_test_cat, verbose=0)
     rnn_loss, rnn_acc = model_rnn.evaluate(X_test_rnn, y_test_cat, verbose=0)

     print(f"CNN - Précision: {cnn_acc:.4f} ({cnn_acc*100:.2f}%)")
     print(f"RNN - Précision: {rnn_acc:.4f} ({rnn_acc*100:.2f}%)")

     # Tableau comparatif
     print("\n=== COMPARAISON ===")
     print(f"{'Métrique':<15} {'CNN':<10} {'RNN':<10}")
     print("-" * 35)
     print(f"{'Précision (%)':<15} {cnn_acc*100:<10.2f} {rnn_acc*100:<10.2f}")
     print(f"{'Temps (s)':<15} {train_time_cnn:<10.1f} {train_time_rnn:<10.1f}")
     print(f"{'Paramètres':<15} {model_cnn.count_params():<10} {model_rnn.count_params():<10}")
```

FIGURE 27 – Code a exécuter pour évaluer la performance des modèles

```
CNN - Précision: 0.9896 (98.96%)
RNN - Précision: 0.9877 (98.77%)

=== COMPARAISON ===
Métrique        CNN        RNN
-----------------------------------
Précision (%)   98.96      98.77
Temps (s)       290.4      252.9
Paramètres      225418     139402
```

FIGURE 28 – Résultat obtenu

Interprétation du résultat

- Le **CNN** obtient une précision légèrement supérieure à celle du **RNN-LSTM**, ce qui est attendu pour une tâche de classification d'images, les CNN étant spécialement conçus pour capter les caractéristiques spatiales des images.
- Le **RNN-LSTM**, bien que généralement utilisé pour les données séquentielles, montre une performance compétitive avec une précision proche de celle du CNN, et un temps d'entraînement légèrement plus court.
- Le nombre de paramètres est également plus faible dans le modèle RNN, ce qui peut être un avantage en termes de complexité et de mémoire dans certains contextes.

Section 6 : Visualisation des courbes d'apprentissage

```
[8]: plt.figure(figsize=(10, 5))

     # Précision
     plt.subplot(1, 2, 1)
     plt.plot(history_cnn.history['val_accuracy'], label='CNN', linewidth=2)
     plt.plot(history_rnn.history['val_accuracy'], label='RNN', linewidth=2)
     plt.title('Évolution de la Précision')
     plt.xlabel('Époque')
     plt.ylabel('Précision')
     plt.legend()
     plt.grid(True)

     # Perte
     plt.subplot(1, 2, 2)
     plt.plot(history_cnn.history['val_loss'], label='CNN', linewidth=2)
     plt.plot(history_rnn.history['val_loss'], label='RNN', linewidth=2)
     plt.title('Évolution de la Perte')
     plt.xlabel('Époque')
     plt.ylabel('Perte')
     plt.legend()
     plt.grid(True)

     plt.tight_layout()
     plt.show()
```

FIGURE 29 – Code à exécuter pour visualiser

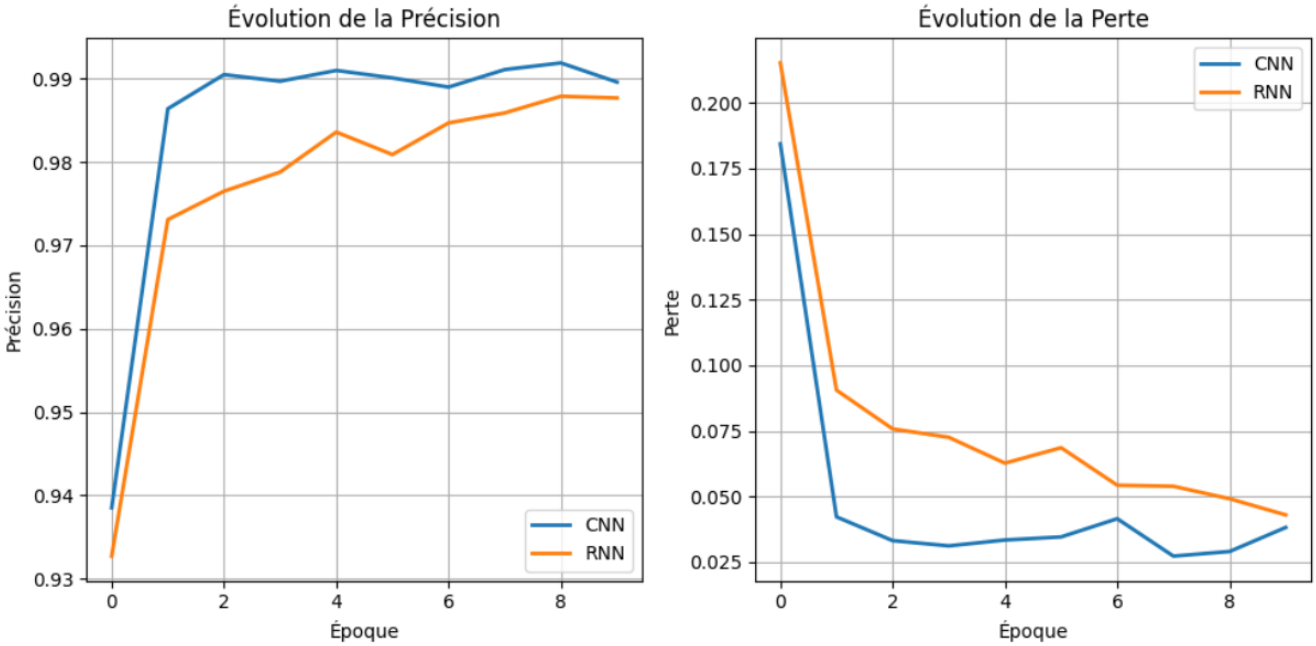

FIGURE 30 – Résultat obtenu

Interprétation du résultat

Évolution de la Précision

- Le modèle **CNN** atteint rapidement une précision élevée dès les premières époques et se stabilise autour de 99 %, montrant une bonne capacité de généralisation dès le début de l'apprentissage.
- Le modèle **RNN** affiche également une progression régulière, mais sa précision plafonne légèrement en dessous de celle du CNN, autour de 98.7 %.
- Cela suggère que le CNN parvient à mieux capturer les caractéristiques spatiales des images que le RNN, qui est initialement plus adapté aux données séquentielles.

Évolution de la Perte

- On observe une diminution rapide de la perte pour les deux modèles, ce qui indique une bonne convergence de l'optimisation.
- La courbe de perte du **CNN** descend plus rapidement et atteint des valeurs plus basses que celle du **RNN**, renforçant l'idée que le CNN est plus efficace pour ce type de données.
- Le **RNN** présente une descente plus progressive et une perte légèrement plus élevée à la fin, mais reste relativement stable, ce qui prouve une certaine robustesse.

Ces courbes confirment que le **CNN** est globalement plus performant sur la tâche de classification d'images *MNIST*, tant en précision qu'en rapidité de convergence. Le **RNN-LSTM** reste toutefois un modèle pertinent, surtout lorsqu'on cherche à réduire la complexité ou le temps d'entraînement, tout en conservant de bonnes performances.

Section 7 : Matrices de confusion

```
[9]: # Prédictions
     y_pred_cnn = np.argmax(model_cnn.predict(X_test_cnn), axis=1)
     y_pred_rnn = np.argmax(model_rnn.predict(X_test_rnn), axis=1)
     y_true = y_test

     # Matrices de confusion
     fig, axes = plt.subplots(1, 2, figsize=(12, 4))

     cm_cnn = confusion_matrix(y_true, y_pred_cnn)
     sns.heatmap(cm_cnn, annot=True, fmt='d', cmap='Blues', ax=axes[0])
     axes[0].set_title(f'CNN (Précision: {cnn_acc*100:.1f}%)')

     cm_rnn = confusion_matrix(y_true, y_pred_rnn)
     sns.heatmap(cm_rnn, annot=True, fmt='d', cmap='Greens', ax=axes[1])
     axes[1].set_title(f'RNN (Précision: {rnn_acc*100:.1f}%)')

     plt.tight_layout()
     plt.show()
```

FIGURE 31 – Code à exécuter

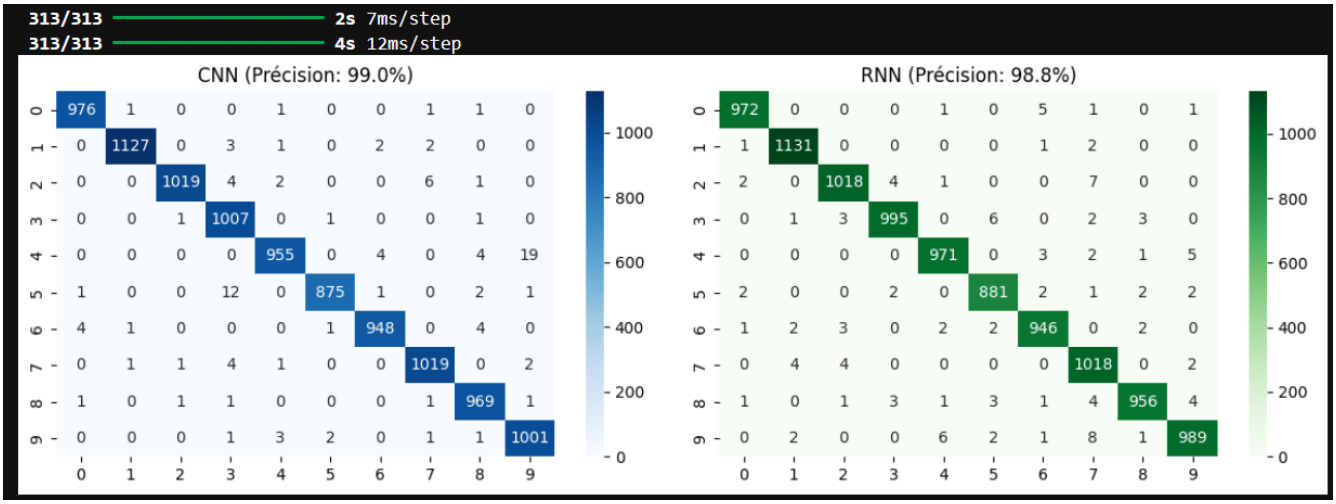

FIGURE 32 – Les matrices obtenues

Interprétation du résultat

Les matrices de confusion affichées ci-dessus permettent de visualiser les performances des modèles **CNN** et **RNN** en termes de classification correcte ou incorrecte de chaque chiffre (0 à 9).

Pour le modèle CNN

- La majorité des prédictions sont correctes, comme le montre la diagonale principale bien remplie.
- Les erreurs sont peu fréquentes et généralement proches des classes voisines, ce qui est typique dans la classification de chiffres manuscrits.
- Le modèle obtient une précision globale de **99.0 %**, avec très peu de confusions entre classes.

Pour le modèle RNN

- Le comportement est similaire, avec une diagonale dominante, bien que légèrement moins marquée que celle du CNN.
- Quelques erreurs supplémentaires apparaissent, notamment entre les chiffres visuellement proches comme le 3 et le 5 ou le 8 et le 9.
- La précision globale est de **98.8 %**, ce qui reste excellent mais inférieur à celle du CNN.

Conclusion

L'analyse des matrices de confusion confirme les résultats précédemment observés avec les courbes d'apprentissage : le modèle **CNN** surpasse légèrement le modèle **RNN** en termes de précision et de capacité à éviter les erreurs de classification. Toutefois, les deux modèles démontrent de très bonnes performances sur la tâche de reconnaissance des chiffres manuscrits, ce qui prouve leur efficacité respective dans ce domaine. Le CNN semble toutefois mieux adapté pour capter les caractéristiques spatiales des images que le RNN.

Partie 3 : Autoencodeur sur CIFAR-10

Section 1 : Construction de l'Autoencodeur

```
[1]: from tensorflow.keras.datasets import cifar10
     from tensorflow.keras.layers import Input, Dense
     from tensorflow.keras.models import Model
```

FIGURE 33 – Importation des bibliothèques necéssaires

```
[7]: # 1. Charger CIFAR-10
     (x_train, _), (x_test, _) = cifar10.load_data()
     x_train = x_train.reshape(50000, 32*32*3) / 255.0
     x_test = x_test.reshape(10000, 32*32*3) / 255.0
```

FIGURE 34 – Chargement des données

Section 2 : Autoencodeur avec 2 couches d'encoding et 2 de decoding

```
[4]: input_layer = Input(shape=(32*32*3,)) # Entrée
     # Encoding
     encoded1 = Dense(1024, activation='relu')(input_layer)
     encoded2 = Dense(512, activation='relu')(encoded1)
     bottleneck = Dense(64, activation='relu')(encoded2)
     # Decoding
     decoded1 = Dense(512, activation='relu')(bottleneck)
     decoded2 = Dense(1024, activation='relu')(decoded1)

     output_layer = Dense(32*32*3, activation='sigmoid')(decoded2)
```

FIGURE 35 – Architecture de l'Autoencodeur

Entrée

- Prend une image **CIFAR-10** de dimension $32 \times 32 \times 3$, aplatie en un vecteur de 3072 valeurs.

Encodage (Compression)

- **Couche 1** : Réduction à 1024 neurones pour extraire les caractéristiques importantes.
- **Couche 2** : Réduction supplémentaire à 512 neurones.
- **Bottleneck (Goulot)** : Compression maximale à 64 neurones, représentant un « résumé » de l'image.

Décodage (Reconstruction)

- **Couche 1** : Expansion à 512 neurones.
- **Couche 2** : Expansion à 1024 neurones.

- **Sortie** : Reconstruction de l'image originale (3072 valeurs) via une fonction `sigmoid` (valeurs normalisées entre 0 et 1).

Objectif

- Apprendre à **compresser** puis à **reconstruire** l'image avec le moins de perte possible, en minimisant l'erreur de reconstruction.

Section 3 : Construction et entraînement

```
[5]: autoencoder = Model(input_layer, output_layer)
     autoencoder.compile(optimizer='adam', loss='mse')
```

FIGURE 36 – Construction

- Crée le modèle final.
- Utilise Adam pour l'optimisation et MSE pour mesurer l'erreur de reconstruction.

```
[*]: hist = autoencoder.fit(x_train, x_train, epochs=20, batch_size=256, validation_data=(x_test, x_test))
     Epoch 1/20
     196/196 ━━━━━━━━━━━━━━━━━━━━ 43s 196ms/step - loss: 0.0506 - val_loss: 0.0302
     Epoch 2/20
     196/196 ━━━━━━━━━━━━━━━━━━━━ 34s 171ms/step - loss: 0.0282 - val_loss: 0.0244
     Epoch 3/20
     196/196 ━━━━━━━━━━━━━━━━━━━━ 38s 194ms/step - loss: 0.0241 - val_loss: 0.0242
     Epoch 4/20
     196/196 ━━━━━━━━━━━━━━━━━━━━ 40s 190ms/step - loss: 0.0232 - val_loss: 0.0222
     Epoch 5/20
     196/196 ━━━━━━━━━━━━━━━━━━━━ 47s 237ms/step - loss: 0.0220 - val_loss: 0.0214
```

FIGURE 37 – Entraînement

- Entraîne le modèle à reconstruire les images.
- 20 époques, batch_size=256.

Section 4 : Tester sur une image

```
[18]: # Sélectionner une image de test
      test_img = x_test[10].reshape(1, 3072)
      reconstructed_img = autoencoder.predict(test_img)
      1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 44ms/step
```

FIGURE 38 – Sélectionner une image de test

- Prend la 10ème image du jeu de test (x_test[0]).
- L'aplatit en un vecteur de 3072 valeurs (pour correspondre à l'entrée du modèle).

- Utilise l'autoencodeur pour reconstruire l'image.

```
[25]: # Afficher l'original vs reconstruit
      plt.figure(figsize=(10, 4))
      plt.subplot(1, 2, 1)
      plt.imshow(x_test[0].reshape(32, 32, 3))  # Utiliser x_test[0] original pour l'affichage
      plt.title("Original")
      plt.axis('off')

      plt.subplot(1, 2, 2)
      plt.imshow(reconstructed_img.reshape(32, 32, 3))
      plt.title("Reconstruit")
      plt.axis('off')

      plt.tight_layout()
      plt.show()
```

FIGURE 39 – Code à exécuter pour Afficher l'original vs reconstruit

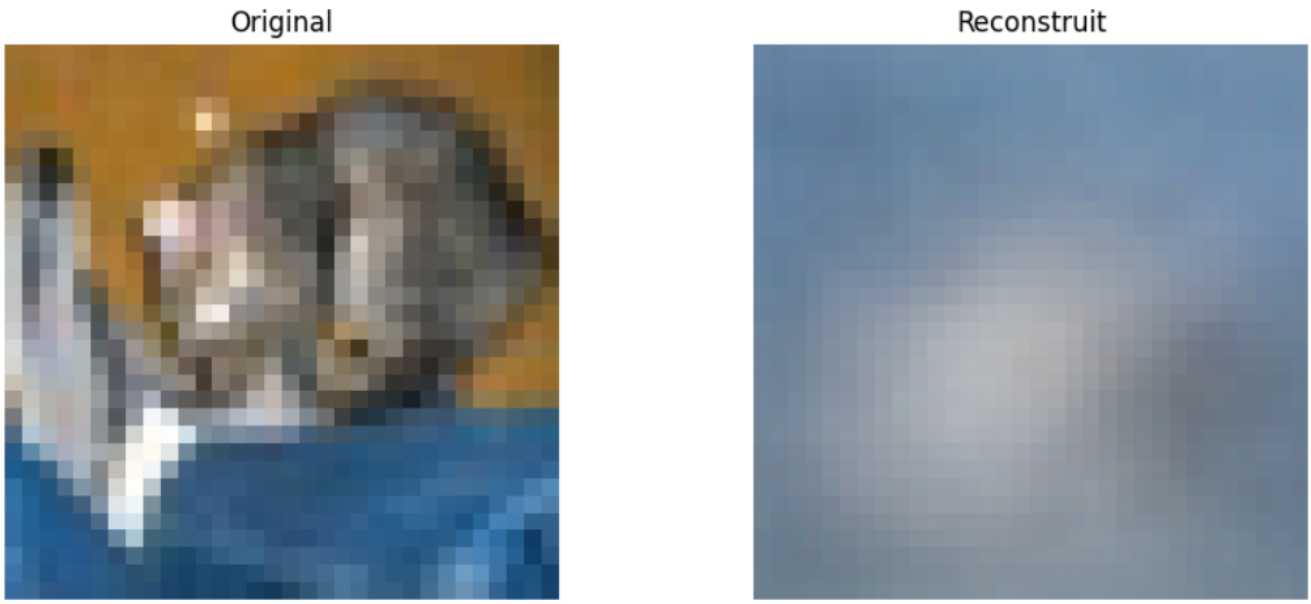

FIGURE 40 – Résultat obtenu

Résultats de Reconstruction

- **Image originale** : Claire et détaillée.
- **Image reconstruite** : Un peu floue, mais conserve les formes principales.

Explication

- L'autoencodeur a bien compressé l'image (taille réduite à 64 valeurs).
- Quelques détails sont perdus à cause de la compression forte.

Améliorations possibles

- Utiliser plus de neurones dans la couche de compression (ex : 128 au lieu de 64).
- Augmenter le nombre d'époques d'entraînement.

Conclusion

Le modèle fonctionne, mais peut être optimisé pour de meilleurs résultats.

Section 5 : Visualisation

```
[20]: # Afficher la courbe de perte
      plt.figure(figsize=(10, 4))
      plt.subplot(1, 2, 1)
      plt.plot(hist.history['loss'], label='Training Loss')
      plt.plot(hist.history['val_loss'], label='Validation Loss')
      plt.title('Évolution de la perte')
      plt.xlabel('Époque')
      plt.ylabel('Perte')
      plt.legend()
```

FIGURE 41 – Code à exécuter pour Afficher la courbe de perte

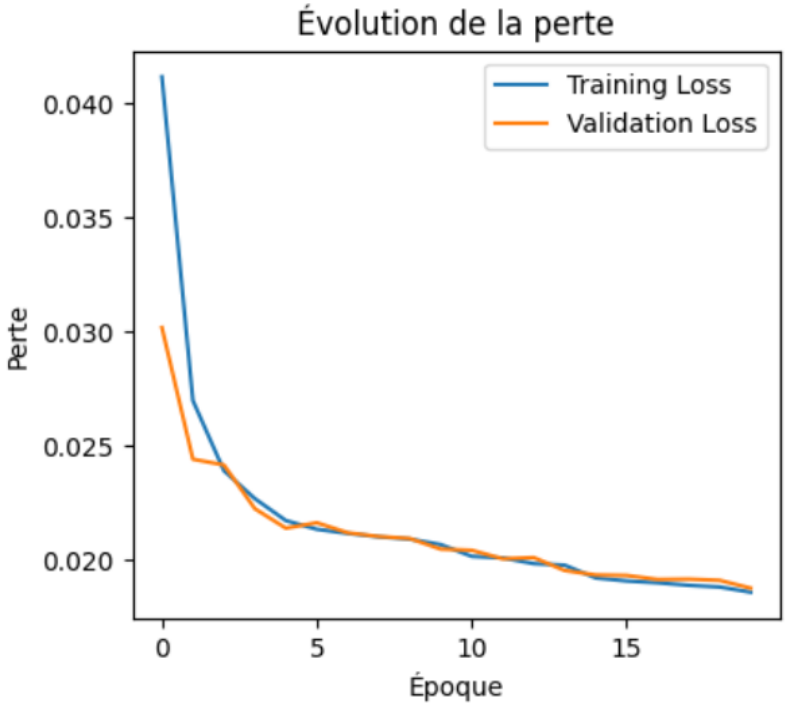

FIGURE 42 – La courbe de perte obtenue

Observations

- **Tendance générale** :

    - Diminution progressive des deux courbes → apprentissage efficace.
    - La perte finale atteint ~ 0.020 (bonne convergence).
- **Comparaison Training/Validation** :
    - Écart minimal entre les courbes → pas de sur-apprentissage détecté.
- **Stabilisation** :
    - Stabilisation après 10 époques → potentiel plateau d'apprentissage.

Interprétation

- **Succès** :
    - Bonne capacité de généralisation (validation loss proche de training loss).
    - Convergence rapide (dès 5 époques).
- **Limites** :
    - Perte résiduelle (> 0.020) suggère des améliorations possibles.

Recommandations

- **Pour améliorer la convergence** :
    - Augmenter le taux d'apprentissage (learning rate).
    - Ajouter des couches au modèle.
- **Pour éviter le sur-apprentissage futur** :
    - Implémenter un *Early Stopping*.
    - Ajouter de la régularisation (Dropout, L2).

Conclusion Générale

Ce travail pratique nous a permis d'explorer et de comparer différentes architectures de *deep learning* sur des jeux de données de référence (**CIFAR-10** et **MNIST**). À travers trois parties distinctes, nous avons pu analyser les performances des **CNN**, **LSTM** et **autoencodeurs**, ce qui nous amène aux conclusions suivantes.

Réponses aux questions

1. Améliorations possibles

- **Augmentation de données** : rotation, translation, ajout de bruit pour améliorer la robustesse du modèle.
- **Architectures plus profondes** : ajout de blocs convolutionnels, utilisation d'architectures pré-entraînées (ex. ResNet, EfficientNet).
- **Optimisation** : ajustement du *learning rate*, utilisation du *early stopping* affiné, *cross-validation*.
- **Pour l'autoencodeur** : élargissement du bottleneck (de 64 à 128), passage à une architecture convolutionnelle au lieu de couches entièrement connectées.

2. Comparaison CNN vs LSTM

Le **CNN** surpasse le **LSTM** sur MNIST (99.0% vs 98.8%) en raison des points suivants :

- **Adaptation spatiale** : les CNN détectent naturellement les motifs 2D (traits, angles).
- **Efficacité** : opérations parallélisables, moins de paramètres à entraîner.
- **Inadéquation du LSTM pour les images** : traite l'image ligne par ligne, ignorant les relations verticales.

3. Choix de modèle pour la classification en temps réel

Un **CNN optimisé** est recommandé pour les cas d'usage en temps réel :

- **Vitesse** : compatible GPU, hautement parallélisable.
- **Efficacité mémoire** : architectures légères comme *MobileNet* ou *EfficientNet*.
- **Optimisations possibles** : *quantization*, *pruning*, *batch processing*.

Conclusion Finale

Ce TP nous a permis de confirmer empiriquement que le **choix de l'architecture** doit être adapté à la nature des données :

- Les **CNN** sont excellents pour les données spatiales (images),
- Les **LSTM** sont mieux adaptés aux données séquentielles,
- Les **autoencodeurs** sont performants pour la compression et la reconstruction.

Les performances obtenues (85.79% sur CIFAR-10, 99% sur MNIST) démontrent l'efficacité de ces approches sur des problèmes concrets. Les techniques d'amélioration identifiées ouvrent la voie à des modèles encore plus performants, applicables à des cas d'usage industriels.

L'expérience acquise sur :

- l'optimisation des hyperparamètres,
- l'interprétation des courbes d'apprentissage,
- l'analyse des matrices de confusion

constitue une base solide pour aborder des projets de *deep learning* plus complexes dans le futur.