

**FORMATION IA LÉGÈRE**  
**Exercices – Deep Learning pour l'embarqué et les systèmes légers**

Selva Systems

28 juin 2025

## Table des matières

<b>1 Premier exemple : Réseau dense sur jeu de données Digits</b>	<b>3</b>
1.1 Les données : Digits . . . . .	3
1.2 Composition et entraînement du modèle . . . . .	3
<b>2 Réseau convolutif sur jeu de données Digits</b>	<b>9</b>
2.1 Pourquoi un CNN ? . . . . .	9
2.2 Composition et entraînement du modèle . . . . .	9
2.3 L'entraînement et les résultats . . . . .	10
<b>3 Mines, roches : des inférences de réseaux denses sur ESP32</b>	<b>13</b>
3.1 L'ESP32-S3 : une cible idéale pour les réseaux compacts . . . . .	13
3.2 Présentation du jeu de données Sonar . . . . .	13
3.3 Préparation des données pour l'entraînement . . . . .	14
3.4 Entraînement et composition du modèle . . . . .	16
3.5 Evaluation du modèle . . . . .	17
3.6 Quantification, export en TFLITE et déploiement sur ESP32 . . . . .	20
3.7 Déploiement sur l'ESP32 . . . . .	21
3.8 Questions de fond sur le code embarqué ESP32-S3 . . . . .	23
<b>4 Prévision de température par CNN 1D pour l'embarqué</b>	<b>25</b>



4.1	Contexte . . . . .	25
4.2	Travail demandé . . . . .	25
4.3	Un exemple déployé sur ESP32 . . . . .	30
4.4	Comparaison des architectures séquentielles . . . . .	31
4.5	Ressources associées . . . . .	31
4.5.1	Questions de comprehension . . . . .	31
4.5.2	Elements de réponse . . . . .	31
<b>5</b>	<b>Detection et localisation d'entités extraterrestres par vision par ordinateur</b>	<b>33</b>
5.1	Code détaillé : détection + localisation avec MobileNetV2 gelé . . . . .	33
5.2	Explications détaillées . . . . .	36
5.3	Questions de comprehension . . . . .	37
5.4	Elements de reponse developpes . . . . .	37
<b>6</b>	<b>Détection de mot-clé (Keyword Spotting, KWS) par CNN sur Spectrogrammes : entraînement</b>	<b>39</b>
6.1	Contexte et organisation de l'exercice . . . . .	39
6.2	Commentaires sur le pipeline et le code . . . . .	39
6.3	Questions . . . . .	40
6.4	Corrigé des réponses . . . . .	40

# 1 Premier exemple : Réseau dense sur jeu de données Digits

## 1.1 Les données : Digits

L'entraînement est réalisé à partir du célèbre jeu de données **Digits** de Scikit-learn. Ce jeu de données est largement utilisé pour tester des algorithmes de classification, car il est simple, bien annoté et permet d'expérimenter rapidement de nouveaux modèles.

Pourquoi est-il célèbre ? Digits rassemble 1797 images en niveaux de gris, de petite taille (8x8 pixels), représentant des chiffres manuscrits de 0 à 9. Ce jeu de données est un standard pédagogique, notamment pour tester et illustrer les méthodes de reconnaissance de formes, car il reste accessible en termes de calcul, tout en conservant une vraie diversité d'écriture.

Chaque entrée est composée :

- d'une image (8x8 pixels, soit 64 valeurs réelles),
- d'un label (entier entre 0 et 9).

## 1.2 Composition et entraînement du modèle

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.utils import to_categorical

# Chargement des données
digits = load_digits()
X = digits.data # shape (1797, 64)
y = digits.target # valeurs entre 0 et 9

# Normalisation
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Encodage one-hot des labels
y_encoded = to_categorical(y, num_classes=10)

# Split train/test
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y_encoded, test_size=0.2, random_state=42)

# Définition du modèle dense
model = Sequential([
    Input(shape=(64,)),
    Dense(32, activation='relu'),
    Dense(32, activation='relu'),
    Dense(10, activation='softmax')
])
# Compilation
```

```

model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Entrainement
history = model.fit(X_train, y_train, epochs=20, batch_size=32,
                      validation_split=0.1)

# Evaluation
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Accuracy sur test: {test_acc:.4f}')
    
```

Listing 1 – Réseau dense sur les digits (Keras)

Ce modèle dense est composé de deux couches cachées de 32 neurones chacune, activées par une fonction ReLU, puis d'une couche de sortie à 10 neurones avec une activation softmax (pour la classification multiclasse).

Le réseau est volontairement compact : cela permet de tester rapidement sur un petit système embarqué, tout en obtenant déjà d'excellents résultats sur des jeux de données simples.

La taille du modèle (nombre de paramètres ajustables) reste faible, ce qui permet une exécution rapide et une faible empreinte mémoire.

TABLE 1 – Composition du modèle séquentiel avec Dropout

Layer (type)	Output Shape	Paramètres
Dense (dense_9)	(None, 64)	4 160
Dropout (dropout_2)	(None, 64)	0
Dense (dense_10)	(None, 32)	2 080
Dropout (dropout_3)	(None, 32)	0
Dense (dense_11)	(None, 10)	330
<b>Total</b>		<b>6 570</b>

On constate ainsi que le modèle n'embarque que 6 570 paramètres entraînables, ce qui est très faible pour un réseau de neurones moderne.

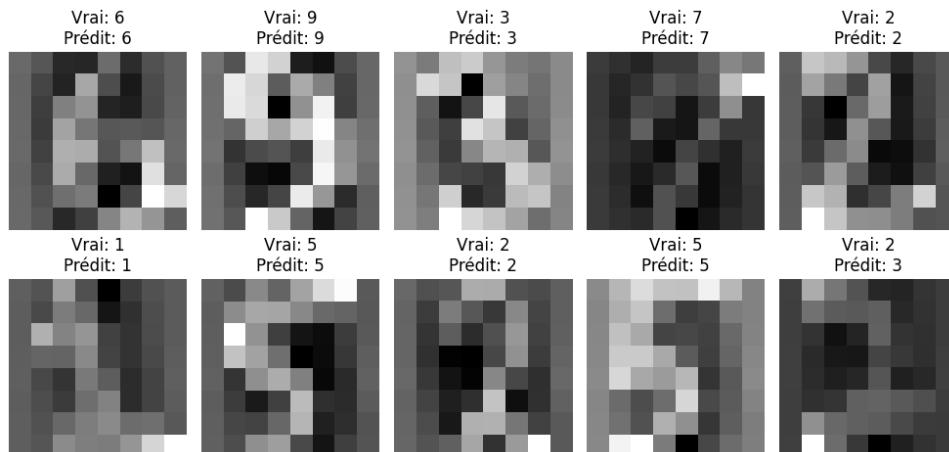


FIGURE 1 – Inférences du modèle et résultats associés : les performances obtenues sont très bonnes en dépit de la légèreté du modèle, avec une bonne correspondance entre les prédictions du modèle et les données attendues.

Les résultats montrent que ce réseau, bien que très simple et peu volumineux, est capable de bien généraliser et de fournir des prédictions précises. Cela souligne l'efficacité du deep learning, même avec des architectures réduites, dès lors que les données sont suffisamment informatives et bien préparées.

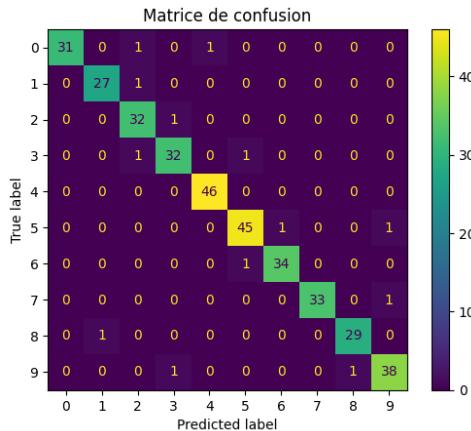


FIGURE 2 – Matrice de confusion.

La matrice de confusion permet de visualiser le nombre de bonnes et de mauvaises prédictions pour chaque chiffre. La diagonale représente les bonnes classifications ; plus les valeurs sont élevées sur cette diagonale, meilleur est le modèle. Les valeurs hors diagonale indiquent les confusions entre certains chiffres, typiquement lorsque les écritures sont proches ou ambiguës.

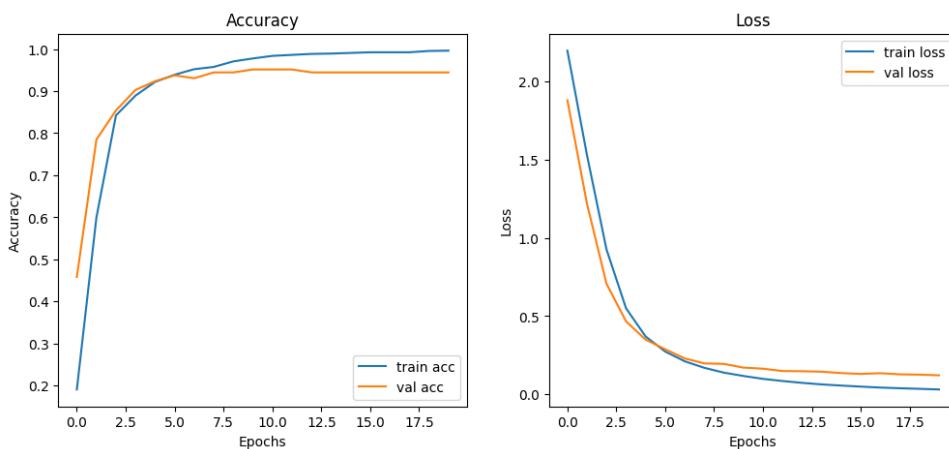


FIGURE 3 – Fonction de coût après 20 époques (2e itération).

La courbe de la fonction de coût (loss) décroît régulièrement au fil des époques, tandis que la précision (accuracy) progresse. C'est généralement le signe d'un bon apprentissage : le modèle s'améliore progressivement sur le jeu d'entraînement comme sur le jeu de validation.

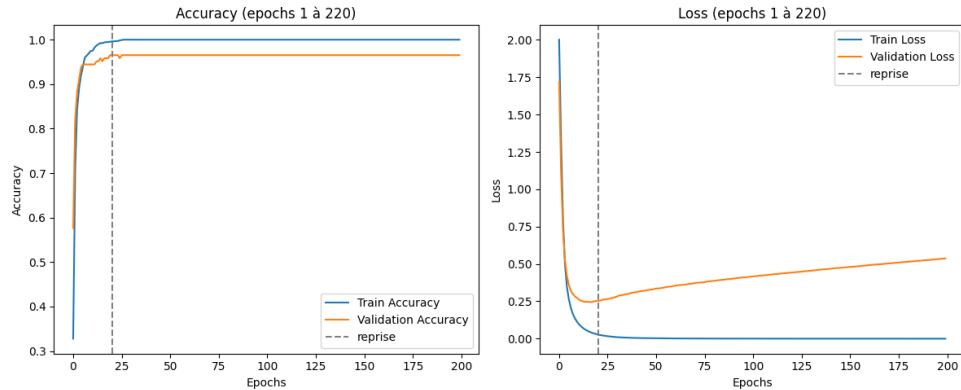


FIGURE 4 – Fonction de coût après 220 époques (sans régularisation).

Lorsqu'on entraîne un modèle de réseau de neurones pendant un grand nombre d'époques sans mécanisme de régulation, on est très souvent confronté au phénomène de **surapprentissage** (*overfitting*). Ce phénomène est caractérisé par un écart croissant entre la performance sur le jeu d'entraînement et celle sur le jeu de validation.

Plus précisément, la courbe de précision (*accuracy*) sur l'entraînement continue de progresser, ce qui peut donner l'impression d'une amélioration constante. Mais simultanément, la précision sur le jeu de validation atteint un plateau, voire diminue progressivement. Cela signifie que le modèle s'adapte trop spécifiquement aux exemples du jeu d'entraînement, en mémorisant même le bruit ou les détails non significatifs, au détriment de sa capacité à généraliser à de nouvelles données.

Ce comportement est particulièrement marqué dans les modèles à forte capacité (grande profondeur, nombreux paramètres) ou lorsqu'on dispose d'un volume de données relativement limité. Sur le jeu **Digits**, bien que les données soient nombreuses pour un exemple pédagogique, elles restent modestes au regard de la complexité potentielle d'un réseau profond.

Pour atténuer, voire prévenir le surapprentissage, plusieurs **techniques de régularisation** peuvent être mises en œuvre. En voici quelques-unes parmi les plus courantes :

- **Dropout** : cette méthode consiste à désactiver aléatoirement un certain pourcentage de neurones lors de chaque itération d'entraînement. Cela oblige le réseau à ne pas trop dépendre de certaines connexions internes. Le dropout agit comme une forme d'ensemble implicite de réseaux plus petits et indépendants.
- **Early stopping** : on surveille la performance sur le jeu de validation à chaque époque, et on interrompt l'entraînement dès que cette performance cesse de s'améliorer. Cela permet d'éviter de sur-entraîner inutilement le modèle.
- **Ajout de bruit** : dans les données d'entrée ou au niveau de certains neurones internes, le bruit agit comme un stimulant à la généralisation, en empêchant le réseau de trop s'adapter aux motifs exacts présents dans le jeu d'entraînement.
- **Régularisation  $L^2$  ou  $L^1$  (poids faibles)** : en ajoutant un terme de pénalité à la fonction de coût, on contraint les poids du réseau à rester faibles (ou à favoriser la parcimonie dans le cas de  $L^1$ ). Cela évite les connexions trop dominantes, souvent associées à du surajustement.
- **Augmentation des données** : bien que parfois impossible, la génération de nouvelles données synthétiques (par transformations, rotations, bruit, etc.) permet de rendre le modèle plus robuste et de compenser la rareté des données réelles.

- **Réduction de la complexité du modèle** : si les données sont simples ou peu nombreuses, il est parfois préférable de réduire le nombre de couches ou de neurones, pour éviter un modèle surdimensionné qui apprendrait les détails inutiles.

Le choix de la régularisation dépend du contexte : taille du jeu de données, capacité du modèle, objectifs de généralisation. Dans le cas présenté ici, l'ajout d'un **dropout** après chaque couche cachée permet de retarder l'apparition du surapprentissage et d'améliorer la performance finale sur le jeu de test, au prix d'une convergence parfois un peu plus lente.

On remarque, sur la figure suivante, que malgré un apprentissage plus bruité, le modèle avec dropout conserve une meilleure capacité de généralisation au-delà de 50 époques, là où le modèle sans régularisation commence à se détériorer.

Voici le même modèle avec Dropout :

```
# Définition du modèle dense
model = Sequential([
    Input(shape=(64,)),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(10, activation='softmax')
])
```

Listing 2 – Réseau dense sur les digits (Keras)

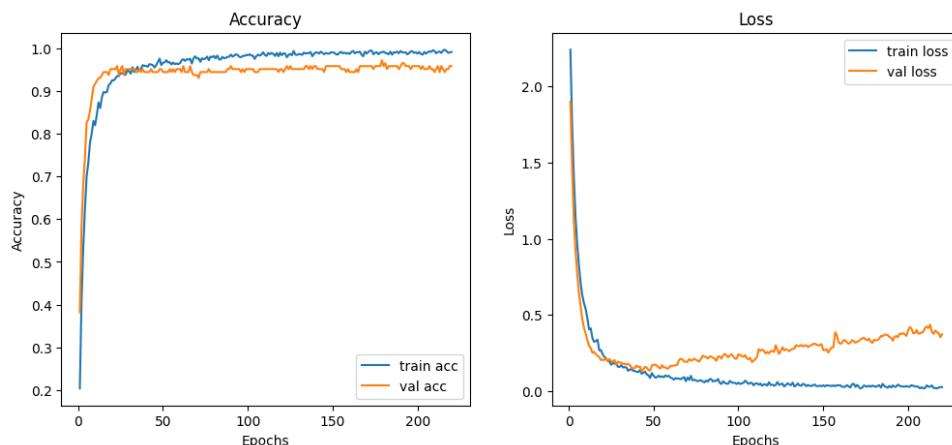


FIGURE 5 – Fonction de coût après 220 époques (avec dropout).

L'utilisation du dropout consiste à "masquer" aléatoirement certains neurones à chaque itération de l'entraînement, ce qui force le modèle à ne pas devenir dépendant de certains chemins privilégiés. On observe alors que la courbe de perte devient plus bruitée (fluctuations), mais le surapprentissage apparaît plus tardivement, aux alentours de la cinquantième époque dans ce cas. Cela permet d'obtenir un modèle qui généralise mieux, au prix d'une légère baisse de performance sur le jeu d'entraînement, mais d'une meilleure robustesse globale.

**Questions pour approfondir (fond et code)**

- Pourquoi le jeu de données Digits, bien qu'ancien et "petit", reste-t-il si utilisé pour l'apprentissage de la classification ?
- Quelles seraient les conséquences de ne pas normaliser les données d'entrée ( $X$ ) avant l'entraînement du réseau dense ? Sur quels types de modèles cette étape est-elle la plus critique ?
- Pourquoi utilise-t-on l'encodage one-hot pour les labels de classification multiclasse dans ce contexte ? Que se passerait-il si on gardait les labels comme entiers simples ?
- Pourquoi la fonction d'activation Softmax est-elle obligatoire à la sortie pour une classification à 10 classes ? Que se passerait-il si on mettait une activation linéaire ou ReLU ?
- Quelle est la logique de l'architecture dense choisie ? Pourquoi deux couches de 32 neurones, et pas plus ou moins ? En quoi le choix de la taille du modèle influence-t-il le risque de surapprentissage ?
- Pourquoi la fonction de cout "categorical crossentropy" est-elle adaptée ici, et non par exemple l'erreur quadratique moyenne ("mse") ?
- Dans la deuxième version, quel est le rôle exact du Dropout, et à quel moment est-il le plus utile lors de l'apprentissage ? Peut-on en abuser ?
- D'un point de vue pratique : à quoi sert la validation\_split dans l'appel à fit() ? Pourquoi ne pas tout simplement évaluer le modèle à la fin sur le jeu de test ?
- Sur quels critères pratiques jugerais-tu qu'un réseau dense n'est "pas assez puissant" pour ce type de problème ?
- En étudiant la matrice de confusion, que révèlent les erreurs du modèle ? Pourrait-on améliorer la robustesse en jouant uniquement sur la préparation des données, sans changer l'architecture ?
- Pourquoi l'ajout de bruit ou l'augmentation des données ("data augmentation") a-t-il un effet positif, même sur un petit jeu de données ?
- Le modèle serait-il adapté à une implémentation embarquée (ex : microcontrôleur, ESP32) sans modification ? Quelles contraintes (mémoire, calcul, etc.) faut-il anticiper ?
- Quel serait l'effet d'un nombre d'époques trop grand sans régularisation, d'après les figures d'évolution de la loss ?

## 2 Réseau convolutif sur jeu de données Digits

### 2.1 Pourquoi un CNN ?

Les réseaux de neurones convolutifs (CNN) sont particulièrement adaptés au traitement d'images. Contrairement aux réseaux entièrement connectés (denses), les CNN exploitent la structure spatiale des images à travers des filtres convolutifs qui apprennent à détecter localement des motifs (traits, courbes, angles). Ces motifs sont ensuite combinés pour former des représentations plus complexes.

Dans le cas du jeu de données **Digits**, bien que les images soient de petite taille (8x8), un CNN permet de tirer profit des structures locales présentes dans les chiffres manuscrits. Cela se traduit souvent par une meilleure capacité de généralisation, une réduction du risque de surapprentissage, et une efficacité accrue pour des modèles de taille équivalente.

### 2.2 Composition et entraînement du modèle

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
, Input
from tensorflow.keras.utils import to_categorical

# Chargement des données
digits = load_digits()
X = digits.images # images 8x8
y = digits.target

# Normalisation
X = X / 16.0 # les pixels vont de 0 à 16
X = X[..., np.newaxis] # reshape pour (8, 8, 1)

# Onehot encoding
y_encoded = to_categorical(y, num_classes=10)

# Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y_encoded, test_size=0.2, random_state=42)

# Modèle CNN
model = Sequential([
    Input(shape=(8, 8, 1)),
    Conv2D(8, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(16, activation='relu'),
    Dense(10, activation='softmax')
])

# Compilation
```

```

model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Entrainement
history = model.fit(X_train, y_train, epochs=20, batch_size=32,
                      validation_split=0.1)

# Evaluation
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"Accuracy sur test : {test_acc:.4f}")
    
```

Listing 3 – Réseau dense sur les digits (Keras)

TABLE 2 – Composition du modèle convolutif

Layer (type)	Output Shape	Paramètres
Conv2D (conv2d_1)	(None, 8, 8, 8)	80
MaxPooling2D (max_pooling2d_1)	(None, 4, 4, 8)	0
Flatten (flatten_1)	(None, 128)	0
Dense (dense_2)	(None, 16)	2 064
Dense (dense_3)	(None, 10)	170
<b>Total</b>		<b>2 314</b>

Le modèle a été choisi pour avoir une taille comparable à celui du réseau dense utilisé précédemment (environ 2314 paramètres ici contre 6570 avec régularisation dans l'exemple dense). Cela permet de comparer les performances à capacité équivalente, tout en exploitant la spécificité du traitement spatial permis par les convolutions.

### 2.3 L'entraînement et les résultats

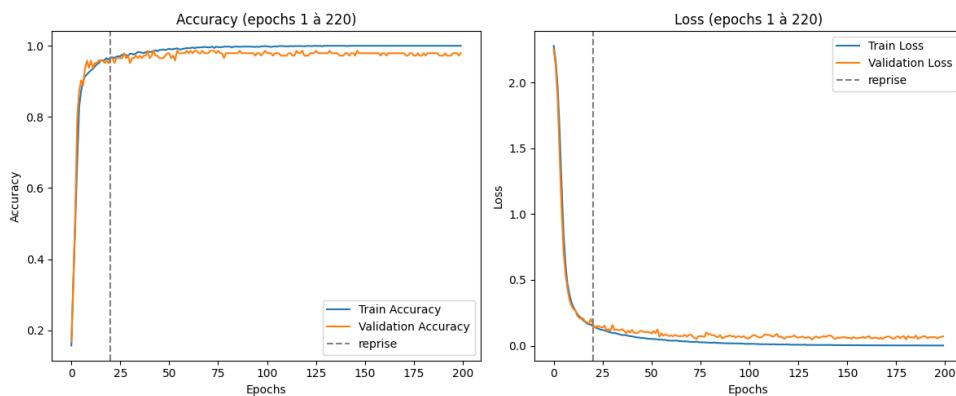


FIGURE 6 – Fonction de coût après 220 époques. On observe que la fonction de coût reste constante au-delà de 40 époques et que la précision sur le jeu de validation ne se détériore pas, indiquant une bonne généralisation du modèle sans apparition de surapprentissage.

On constate que le modèle convolutif conserve de bonnes performances même après un entraînement prolongé. Contrairement au modèle dense sans régularisation, il ne montre pas de signes clairs de surapprentissage. La précision sur le jeu de validation reste stable, et la courbe de perte ne se dégrade pas.

Ce comportement s'explique par plusieurs facteurs inhérents aux CNN :

- **Partage des poids** : les filtres convolutifs sont appliqués sur toute l'image, ce qui réduit considérablement le nombre de paramètres à apprendre et diminue le risque de suradaptation.
- **Inductive bias spatial** : la convolution impose au modèle une forme de connaissance a priori sur la structure locale des images, ce qui améliore la robustesse aux variations d'écriture.
- **Pooling et réduction de dimension** : la couche de max pooling réduit la taille des représentations intermédiaires, limitant le surajustement et favorisant la généralisation.

En résumé, même avec un nombre de paramètres inférieur à celui du réseau dense régularisé, le CNN tire pleinement parti de la structure des images et parvient à généraliser plus efficacement. C'est un résultat classique mais important à retenir : dans les tâches de vision par ordinateur, les CNN surpassent presque toujours les architectures denses, surtout lorsque la structure spatiale des données est exploitable.

### Questions/Réponses

#### Questions

- Pourquoi la structure spatiale des images manuscrites rend-elle les CNN plus performants que les réseaux denses ?
- Quel est l'intérêt d'utiliser un kernel de convolution  $3 \times 3$  plutôt qu'un kernel  $5 \times 5$  ou  $7 \times 7$  dans ce cas précis ?
- Comment le partage des poids dans les couches Conv2D influence-t-il la capacité de généralisation du modèle, surtout avec peu de données ?
- Si on supprimait la couche MaxPooling2D, que se passerait-il sur le plan du surapprentissage et du temps de calcul ?
- Pourquoi le CNN reste-t-il performant même avec moins de paramètres qu'un réseau dense ?

#### Réponses

- **Structure spatiale et CNN** : Les CNN tirent parti de la disposition des pixels grâce à leurs filtres locaux, qui détectent les motifs où qu'ils apparaissent. Cette capacité "géométrique" manque aux réseaux denses, qui voient l'image comme une simple liste de valeurs : ils perdent donc l'information spatiale utile à la reconnaissance de chiffres manuscrits.
- **Pourquoi un kernel  $3 \times 3$  ?** Sur une image de  $8 \times 8$ , un kernel  $3 \times 3$  permet de détecter des motifs locaux tout en limitant le nombre de paramètres. Un noyau plus grand ( $5 \times 5$  ou  $7 \times 7$ ) risquerait de "brouiller" l'information locale, d'augmenter le surapprentissage et de ralentir l'entraînement, sans réel gain sur ce type de données.
- **Partage des poids et généralisation** : Un filtre de convolution "scanne" toute l'image avec les mêmes paramètres, ce qui oblige le modèle à apprendre des motifs généraux. Cela réduit fortement le risque de surapprentissage, car le modèle nécessite moins de paramètres et moins d'exemples pour bien généraliser.
- **Suppression de MaxPooling2D** : Sans cette couche, la taille des représentations intermédiaires resterait grande. Il y aurait plus de paramètres dans les couches suivantes, donc plus de risque d'overfitting et un temps de calcul plus long. Le pooling agit aussi comme une forme de régularisation.

- **Efficacité des CNN avec peu de paramètres :** Grace à la convolution, chaque filtre extrait une information locale pertinente sur toute l'image. Il n'est donc pas nécessaire de multiplier les paramètres : le réseau est plus "intelligent" et exploite mieux chaque poids appris.

### 3 Mines, roches : des inférences de réseaux denses sur ESP32

#### 3.1 L'ESP32-S3 : une cible idéale pour les réseaux compacts

L'**ESP32-S3**, grâce à sa fréquence de **240 MHz**, sa capacité de calcul d'environ **250 MFLOPS** et ses **320 ko de RAM**, constitue une plateforme performante pour le déploiement de modèles de deep learning légers.

Il prend en charge l'exécution de réseaux de neurones via la bibliothèque **TensorFlow Lite Micro**, qui permet d'exécuter des inférences à partir de modèles quantifiés (**.tflite**) directement en C/C++, sans système d'exploitation, ni dépendance lourde.

Jusqu'à quelques milliers de paramètres (typiquement moins de 10000), il est possible de faire tourner un modèle complet, en temps réel, sur l'ESP32-S3, avec des temps d'inférence très faibles (souvent < 1 ms), sans surcharge notable du système embarqué.

Ce potentiel est particulièrement intéressant en dehors du champ de la vision par ordinateur — souvent trop coûteux pour une telle cible —, dans des applications plus sobres et industrielles, telles que :

- la classification de signaux capteurs (acoustiques, vibratoires, climatiques, etc.),
- la reconnaissance de séquences temporelles (via réseaux denses ou 1D),
- la détection d'anomalies ou la prédiction d'événements locaux.

Ce couplage entre **acquisition directe de données capteurs** et traitement par réseau neuronal ouvre des perspectives nombreuses. Par exemple, il a été possible de développer un **prédicteur de gelée** embarqué, basé sur des données climatiques locales mesurées par capteurs environnementaux. Le modèle apprend à prédire les risques de gel à partir de l'évolution des températures, de l'humidité et de la pression atmosphérique. Le dispositif fonctionne entièrement en local, sans dépendre d'une infrastructure cloud ou d'un service météo externe.

Une présentation détaillée de cette réalisation est disponible à l'adresse suivante : <https://selvasystems.net/deep-learning>

#### 3.2 Présentation du jeu de données Sonar

Le jeu de données **Sonar** (Connectionist Bench Sonar Mines vs. Rocks), proposé par le *UCI Machine Learning Repository*, est un jeu de classification binaire bien connu dans la communauté du machine learning. Il contient **208 échantillons**, chacun décrit par **60 caractéristiques numériques**. Ces caractéristiques représentent des intensités de retour de signal sonar à différentes fréquences, probablement issues d'un spectrogramme, bien que les unités ne soient pas spécifiées.

Chaque échantillon est étiqueté comme suit :

- "R" pour *Rock* (rocher),
- "M" pour *Mine* (mine sous-marine).

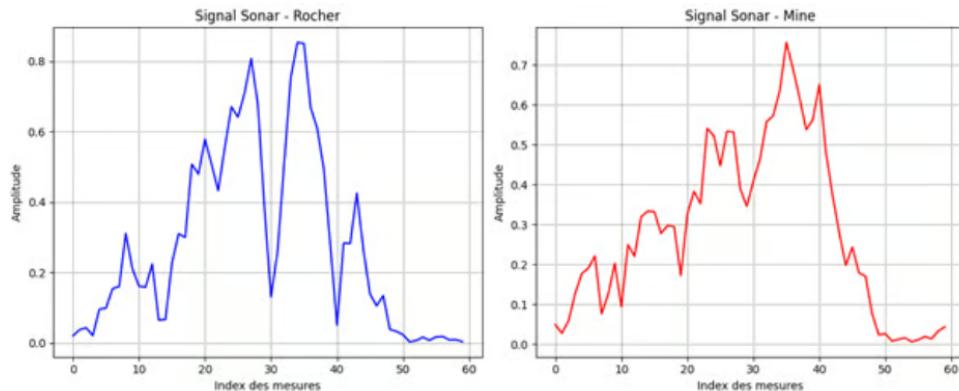


FIGURE 7 – Exemples de spectres issus du jeu Sonar : à gauche, un échantillon étiqueté R ; à droite, un échantillon M.

Ce dataset présente plusieurs atouts pédagogiques :

- une taille réduite, idéale pour des entraînements rapides ;
- une classification binaire, qui simplifie la structure du modèle ;
- une bonne variabilité des données, qui rend l'exercice réaliste ;
- une compatibilité directe avec des réseaux déployables sur microcontrôleurs.

Le code d'intégration est disponible sur le dépôt GitHub : [https://github.com/Bertrand-selvasystems/DeepLearning\\_ESP32\\_S3](https://github.com/Bertrand-selvasystems/DeepLearning_ESP32_S3)

Cette approche illustre parfaitement le lien entre apprentissage sur poste de travail, export du modèle et exécution en contexte embarqué. Elle constitue une base solide pour construire des systèmes intelligents autonomes dans des environnements contraints et embarqués.

### 3.3 Préparation des données pour l'entraînement

Cette sous-section détaille le pipeline de préparation des données, fondamental avant tout appren-tissage. Le script commence par charger le fichier CSV en ligne, puis sépare les caractéristiques (X) des étiquettes (y). L'encodage des labels transforme les classes texte ("R", "M") en valeurs numériques (0, 1), comme attendu par Keras.

Avant l'entraînement, l'auteur choisit d'afficher un exemple typique pour chaque classe. Les tracés montrent visuellement les différences de réponse du sonar, ce qui aide à valider l'intuition que ces signaux sont séparables.

Enfin, l'environnement TensorFlow est configuré, avec une vérification de la présence éventuelle de GPU et la définition d'une politique de calcul en mixed float16. Cette dernière ligne est surtout utile en entraînement sur machine puissante, mais elle est sans effet direct sur l'inférence embarquée, qui sera en quantifié int8 ou en float32.

```
# CHARGEMENT DES DONNEES
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
```

```
# Charger les données depuis un fichier CSV
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/
       undocumented/connectionist-bench/sonar/sonar.all-data"
data = pd.read_csv(url, header=None)

# Separer les caractéristiques et les étiquettes
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

# Nombre de caractéristiques
num_échantillons = X.shape[0]
print(f"Nombre d'échantillons : {num_échantillons}")
num_features = X.shape[1]
print(f"Nombre de caractéristiques (features) : {num_features}")

# AFFICHAGE DE RESULTATS BRUTES
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Charger le dataset Sonar
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/
       undocumented/connectionist-bench/sonar/sonar.all-data"
data = pd.read_csv(url, header=None)

# Separer les caractéristiques et les étiquettes
X = data.iloc[:, :-1].values # Les 60 caractéristiques
y = data.iloc[:, -1].values # Les étiquettes "R" (rock) et "M" (mine)

# Trouver un exemple de rocher et un exemple de mine
rock_sample = X[np.where(y == "R")[0][0]] # Premier échantillon de type
                                             "R"
mine_sample = X[np.where(y == "M")[0][0]] # Premier échantillon de type
                                             "M"

# Générer un axe des indices (de 0 à 59, correspondant aux 60 mesures du
# sonar)
time_steps = np.arange(len(rock_sample))

# Affichage des signaux bruts
plt.figure(figsize=(12, 5))

# Signal du rocher
plt.subplot(1, 2, 1)
plt.plot(time_steps, rock_sample, label="Rock", color='blue')
plt.title("Signal Sonar - Rocher")
plt.xlabel("Index des mesures")
plt.ylabel("Amplitude")
plt.grid()

# Signal de la mine
plt.subplot(1, 2, 2)
plt.plot(time_steps, mine_sample, label="Mine", color='red')
plt.title("Signal Sonar - Mine")
plt.xlabel("Index des mesures")
plt.ylabel("Amplitude")
plt.grid()
```

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

# Encoder les étiquettes
encoder = LabelEncoder()
y = encoder.fit_transform(y)

import tensorflow as tf

print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU'))
      )
print("Liste des GPU disponibles : ", tf.config.list_physical_devices(
    'GPU'))

from tensorflow.keras import mixed_precision

mixed_precision.set_global_policy('mixed_float16')
```

Listing 4 – Réseau dense sur les digits (Keras)

### 3.4 Entraînement et composition du modèle

L’entraînement du modèle repose sur une architecture dense (fully connected), constituée de trois couches cachées, chacune composée de 32 neurones. Ce choix d’architecture reflète un compromis pertinent entre compacité et capacité d’apprentissage. Il est particulièrement adapté aux jeux de données de petite taille comme Sonar, qui ne justifient ni des architectures très profondes, ni le recours à des réseaux convolutifs.

Chaque couche cachée utilise la fonction d’activation ReLU (Rectified Linear Unit), une fonction très répandue dans le deep learning moderne. Elle présente plusieurs avantages : sa simplicité de calcul, sa capacité à introduire de la non-linéarité, et surtout, son aptitude à éviter le problème du gradient qui disparaît (ce qui peut survenir avec des fonctions comme la sigmoïde ou la tangente hyperbolique dans les couches profondes).

La couche de sortie contient un seul neurone, avec une activation sigmoïde. Cette fonction transforme la sortie du réseau en une valeur comprise entre 0 et 1, interprétée comme une probabilité d’appartenance à la classe cible. Ce choix est parfaitement adapté au cadre d’une classification binaire (dans notre cas, distinguer une mine d’un rocher).

Avant l’entraînement proprement dit, les données sont divisées en deux sous-ensembles : 80 pourcent sont réservés à l’apprentissage, et 20 pourcent sont conservés pour tester la capacité du modèle à généraliser. En plus de cette séparation, une fraction de l’ensemble d’entraînement est utilisée pour la validation croisée : cela permet de suivre en temps réel la performance du modèle sur des données non vues pendant les itérations d’apprentissage, et de détecter d’éventuels signes de surapprentissage (overfitting), comme une baisse des performances en validation alors que la perte en entraînement continue à diminuer.

L’entraînement s’effectue sur 75 époques, c’est-à-dire que le réseau parcourt 75 fois l’ensemble des données d’apprentissage. La taille de batch choisie est de 16, ce qui signifie que les mises à jour des poids du réseau se font toutes les 16 données. Cette taille relativement petite permet de profiter de la stochasticité de l’apprentissage tout en assurant une convergence suffisamment rapide et stable.

Enfin, le modèle entraîné est sauvegardé dans un fichier au format HDF5. Ce format stocke non seulement la structure du réseau et ses poids, mais également les informations de compilation (fonction de perte, optimiseur, métriques), ce qui permet de le recharger ultérieurement pour inférence, poursuite de l'entraînement, ou conversion vers d'autres formats (comme TensorFlow Lite pour les microcontrôleurs).

```
# ENTRAINEMENT DU MODELE
# Diviser les données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# Normaliser les caractéristiques
#scaler = StandardScaler()
#X_train = scaler.fit_transform(X_train)
#X_test = scaler.transform(X_test)

# Construire le modèle
model = Sequential([
    Dense(32, activation='relu', input_shape=(X_train.shape[1],)),
    Dense(32, activation='relu'),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compiler le modèle
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[accuracy])

# Entrainer le modèle
model.fit(X_train, y_train, epochs=75, batch_size=16, validation_split=0.2)

# Sauvegarder le modèle entraîné
model.save("mon_modele.h5")

print("Modèle entraîné et sauvégarde sous 'mon_modele.h5'")
```

Listing 5 – Réseau dense sur les digits (Keras)

### 3.5 Evaluation du modèle

Une fois le modèle entraîné, il est évalué sur les données de test. La précision globale est calculée, mais une analyse plus fine est proposée : la matrice de confusion permet de visualiser les erreurs de classification, tandis que la courbe ROC donne une idée du compromis entre vrais positifs et faux positifs selon le seuil choisi.

En complément, l'auteur trace aussi l'évolution de la perte et de la précision au fil des époques. Cela permet de vérifier que l'apprentissage progresse correctement sans divergence ni stagnation prématuée. Toutefois, pour que ces courbes s'affichent, il faudrait que l'objet history soit récupéré explicitement depuis le fit() — ce détail peut être ajusté dans le code. Ces visualisations sont essentielles pour comprendre les performances du modèle avant déploiement.

```
# Evaluer le modèle
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Precision sur les données de test : {accuracy * 100:.2f}%)')

# AFFICHAGE DE LA MATRICE DE CONFUSION ET DE LA COURBE ROC
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.metrics import confusion_matrix, roc_curve, auc

# Faire des prédictions sur X_test
y_pred = model.predict(X_test)
y_pred_labels = (y_pred > 0.5).astype(int) # Convertir en classes binaires

# Matrice de confusion
conf_matrix = confusion_matrix(y_test, y_pred_labels)

# Afficher la matrice de confusion
plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Classe 0', 'Classe 1'], yticklabels=['Classe 0', 'Classe 1'])
plt.xlabel("Predictions")
plt.ylabel("Vraies classes")
plt.title("Matrice de Confusion")
plt.show()

# Courbe ROC
fpr, tpr, _ = roc_curve(y_test, y_pred)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='grey', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Taux de Faux Positifs')
plt.ylabel('Taux de Vrais Positifs')
plt.title('Courbe ROC')
plt.legend(loc='lower right')
plt.show()

# Afficher l'historique de l'entraînement
if 'history' in locals():
    history = history.history # Recuperer l'historique
    epochs = range(len(history['loss']))

    # Tracer la courbe de perte
    plt.figure(figsize=(6, 5))
    plt.plot(epochs, history['loss'], label='Perte entraînement')
    plt.plot(epochs, history['val_loss'], label='Perte validation')
    plt.xlabel('Epochs')
    plt.ylabel('Perte')
    plt.title('Evolution de la Perte')
    plt.legend()
    plt.show()
```

```
# Tracer la courbe de precision
plt.figure(figsize=(6, 5))
plt.plot(epochs, history['accuracy'], label='Precision entrainement',
         )
plt.plot(epochs, history['val_accuracy'], label='Precision validation')
plt.xlabel('Epoques')
plt.ylabel('Precision')
plt.title('Evolution de la Precision')
plt.legend()
plt.show()
```

Listing 6 – Réseau dense sur les digits (Keras)

La **courbe ROC** (Receiver Operating Characteristic) permet d'évaluer la qualité d'un modèle de classification binaire en examinant ses performances pour différents seuils de décision. Elle trace le **taux de vrais positifs** (True Positive Rate, ou rappel) en fonction du **taux de faux positifs** (False Positive Rate), indépendamment du seuil choisi pour discréteriser les sorties du modèle. Une courbe idéale se rapproche du coin supérieur gauche du graphique (TPR proche de 1 et FPR proche de 0), ce qui indique que le modèle parvient à bien séparer les deux classes. L'**aire sous la courbe** (AUC — Area Under Curve) est un indicateur synthétique de performance, compris entre 0,5 (modèle aléatoire) et 1 (modèle parfait).

Dans notre cas, la courbe ROC ci-dessous présente une forme caractéristique d'un modèle performant, avec un **AUC de 0,95**. Cela signifie que dans 95 % des cas, le modèle attribue une probabilité plus élevée à un échantillon de la classe positive (classe 1) qu'à un échantillon de la classe négative (classe 0) pris au hasard. Cette courbe met en évidence la bonne capacité du modèle à discriminer les classes, tout en offrant une flexibilité pour ajuster le compromis entre rappel et précision selon les contraintes de l'application. Par exemple, dans un contexte sensible à la détection (sécurité, maintenance prédictive, etc.), on pourra abaisser le seuil de décision afin d'augmenter le taux de détection au prix d'un léger surcroît de faux positifs.

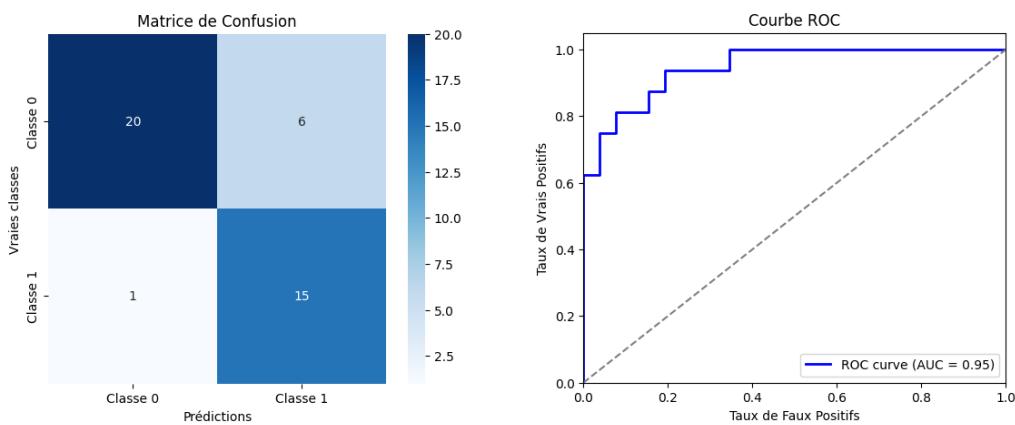


FIGURE 8 – Matrice de confusion (à gauche) et courbe ROC (à droite) du modèle dense entraîné sur le jeu de données *Sonar*. La courbe ROC montre une aire sous la courbe (AUC) de 0,95, révélatrice d'une excellente capacité de séparation entre les deux classes.

### 3.6 Quantification, export en TFLITE et déploiement sur ESP32

Cette dernière étape constitue un moment clé dans le processus de déploiement d'un modèle de deep learning sur une cible embarquée. Une fois le modèle entraîné et validé sur un poste de travail (typiquement à l'aide de Keras et TensorFlow), il est nécessaire de le convertir dans un format compatible avec l'exécution sur microcontrôleur. Pour cela, on utilise l'outil de conversion officiel fourni par TensorFlow : le TFLiteConverter.

Ce convertisseur transforme un modèle Keras (.h5) en un fichier binaire au format TensorFlow Lite (.tflite), qui est ensuite interprété à l'exécution par la bibliothèque TensorFlow Lite Micro sur la cible. Cette bibliothèque ne gère qu'un sous-ensemble du format TFLite, adapté aux contraintes des systèmes sans système d'exploitation et disposant de très peu de mémoire (comme l'ESP32-S3).

Dans le cadre de cette démonstration, on a fait le choix de conserver les poids et calculs du modèle au format float32 lors de la conversion. Ce format flottant simple précision présente l'avantage de la simplicité : il ne nécessite ni quantification, ni calibration, ni gestion d'erreurs d'arrondi spécifiques, ce qui en fait un bon choix pour les phases de test initiales ou pour un public débutant. En revanche, il n'est pas optimal pour une exécution en production sur microcontrôleur, en particulier sur des architectures où la mémoire RAM est limitée.

En effet, un modèle float32 peut consommer jusqu'à quatre fois plus de mémoire qu'un modèle quantifié en int8, tant pour le stockage des poids que pour l'exécution des inférences. Cette surconsommation peut rapidement devenir un goulot d'étranglement, surtout lorsqu'on souhaite embarquer plusieurs modèles ou traiter de longues séquences.

Cela dit, dans certains cas bien spécifiques — notamment lorsque la cible matérielle dispose d'unités matérielles de calcul flottant ou de bibliothèques optimisées pour les instructions DSP — le float32 peut rester compétitif. Sur l'ESP32-S3, certaines bibliothèques C/C++ optimisées permettent de tirer parti des instructions SIMD ou des accélérations matérielles pour les calculs flottants, ce qui peut rendre l'usage du float32 acceptable, voire avantageux, dans des contextes bien calibrés.

En résumé, le choix de rester en float32 est justifié ici par une volonté de simplification, mais il constitue une première étape. Pour un déploiement final sur produit embarqué, il serait préférable de passer par une quantification post-entraînement en int8, accompagnée éventuellement d'une phase de calibration avec données réelles, afin d'optimiser la taille mémoire et la vitesse d'exécution du modèle sur la cible ESP32.

```
# EXPORT DU MODELE VERS TENSORFLOW LITE
import tensorflow as tf

# Charger le modèle entraîné
model = tf.keras.models.load_model("mon_modele.h5")

# Convertir en TensorFlow Lite en conservant le FLOAT32
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [] # Pas de quantification
converter.target_spec.supported_types = [tf.float32] # Forcer le
#FLOAT32

tflite_model = converter.convert()

# Sauvegarder le modèle converti
with open("modele.tflite", "wb") as f:
```

```

f.write(tfLite_model)

print("Modele converti en TensorFlow Lite en FLOAT32.")


# Lire le fichier TFLite
with open("model.tflite", "rb") as f:
    tfLite_bytes = f.read()

# Convertir en tableau C
hex_array = ', '.join(f'0x{b:02x}' for b in tfLite_bytes)
c_code = f"""
unsigned char model_tfLite[] = {{
    {hex_array}
}};
unsigned int model_tfLite_len = {len(tfLite_bytes)};
"""

# Sauvegarder dans un fichier
with open("model_data.cc", "w") as f:
    f.write(c_code)

print("Modele converti en tableau C et sauvegarde sous 'model_data.cc'")

from google.colab import files

# Telecharger model.tflite
files.download("model.tflite")

# Telecharger model_data.cc
files.download("model_data.cc")

```

Listing 7 – Réseau dense sur les digits (Keras)

### 3.7 Déploiement sur l'ESP32

Le modèle ainsi converti en tableau C est désormais prêt à être intégré dans un programme embarqué sur ESP32-S3. Le code suivant présente l'intégration complète du modèle dans un projet C++ utilisant la bibliothèque **TensorFlow Lite Micro**, sans système d'exploitation.

L'ensemble repose sur cinq composants essentiels :

- la **déclaration du modèle** sous forme de tableau binaire (`model.h`) ;
- l'**initialisation du moteur d'inférence** (`neurone_init()`) ;
- le **remplissage du tenseur d'entrée** avec des données capteur ou simulées ;
- l'**exécution de l'inférence** via `interpreter->Invoke()` ;
- et enfin, la **lecture du résultat**, contenu dans le tenseur de sortie.

```

#include "neurone.h" // init and inference functions
#include "model.h"   // binary data of the exported .tflite model

#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/micro/system_setup.h"
#include "tensorflow/lite/schema/schema_generated.h"

```

```
static const tflite::Model *model = nullptr;
static tflite::MicroInterpreter *interpreter = nullptr;
static TfLiteTensor *input = nullptr;
static TfLiteTensor *output = nullptr;
static int inference_count = 0;

static constexpr int kTensorArenaSize = 40000;
static uint8_t tensor_arena[kTensorArenaSize];

esp_err_t neurone_init() {
    model = tflite::GetModel(g_model);
    if (model->version() != TFLITE_SCHEMA_VERSION) {
        MicroPrintf("Model version mismatch.");
        return ESP_FAIL;
    }

    static tflite::MicroMutableOpResolver<8> resolver;
    resolver.AddFullyConnected();
    resolver.AddRelu();
    resolver.AddLogistic();
    resolver.AddQuantize();
    resolver.AddDequantize();
    resolver.AddReshape();
    resolver.AddAdd();
    resolver.AddMul();

    static tflite::MicroInterpreter static_interpreter(model, resolver,
        tensor_arena, kTensorArenaSize);
    interpreter = &static_interpreter;

    if (interpreter->AllocateTensors() != kTfLiteOk) {
        MicroPrintf("Tensor allocation failed.");
        return ESP_FAIL;
    }

    input = interpreter->input(0);
    output = interpreter->output(0);

    return ESP_OK;
}

esp_err_t neurone_inference(const float *input_values, float *
    output_values) {
    for (int i = 0; i < 60; ++i) {
        input->data.f[i] = input_values[i];
    }

    if (interpreter->Invoke() != kTfLiteOk) {
        MicroPrintf("Inference failed.");
        return ESP_FAIL;
    }

    for (int i = 0; i < 1; ++i) {
        output_values[i] = output->data.f[i];
    }

    inference_count++;
    return ESP_OK;
}
```

{}

## Listing 8 – Code embarqué sur ESP32-S3 pour inférence du modèle Sonar

Ce code exécute un réseau dense compact, entraîné à partir du jeu de données *Sonar*, directement sur microcontrôleur ESP32-S3. Il démontre qu'il est possible d'effectuer une inférence complète, en moins de 1 milliseconde, sans dépendre d'un système d'exploitation, et avec une empreinte mémoire très contenue (moins de 50 ko RAM utilisés au total, code compris).

L'approche illustrée ici est généralisable à tous les réseaux denses ou 1D de petite taille, et peut s'étendre à des cas d'usage concrets : détection d'anomalies mécaniques, reconnaissance de profils vibratoires, surveillance de paramètres environnementaux, ou toute application sobre en données mais riche en intelligence locale.

## Code source complet

Le code complet du projet est disponible à l'adresse suivante :

[https://github.com/Bertrand-selvasystems/DeepLearning\\_ESP32\\_S3](https://github.com/Bertrand-selvasystems/DeepLearning_ESP32_S3)

### 3.8 Questions de fond sur le code embarqué ESP32-S3

- Quel est le rôle du tableau `tensor_arena` et pourquoi sa taille est-elle critique ?
- Pourquoi utilise-t-on un `MicroMutableOpResolver` avec une liste explicite d'opérations ?
- Que se passe-t-il si le modèle TFLite utilise une opération non ajoutée dans le resolver ?
- Pourquoi recopie-t-on les données d'entrée une par une dans `input->data.f[i]` ?
- Quelle est la différence entre les fonctions `neurone_init()` et `neurone_inference()` ?
- Pourquoi le code ne gère-t-il que des tenseurs float32 alors que la quantification en int8 est possible ?
- A quoi sert le compteur `inference_count` ?
- Quelles stratégies seraient possibles pour debugger l'inference si `Invoke()` échoue ?

#### Reponses

- Le tableau `tensor_arena` est la mémoire préallouée utilisée par TensorFlow Lite Micro pour stocker les tenseurs (entrées, sorties, intermédiaires). Sa taille doit être suffisante pour couvrir tous les besoins du modèle. Si elle est trop petite, l'allocation des tenseurs échoue.
- Le `MicroMutableOpResolver` déclare les opérations que l'on autorise à être utilisées par le modèle. Cela permet de réduire l'empreinte mémoire en embarquant uniquement les opérations nécessaires.
- Si une opération du modèle TFLite n'est pas ajoutée dans le resolver, le programme échoue à l'initialisation, généralement avec un message du type "Op not found" ou "Registration failed".
- On recopie les données manuellement dans `input->data.f[i]` car TensorFlow Lite Micro ne fournit pas d'API de chargement direct de tableau. Cela garantit le contrôle sur le format et les valeurs d'entrée.

- `neurone_init()` initialise le modèle, le resolver, les allocations et les pointeurs vers les tenseurs. `neurone_inference()` remplit le tenseur d'entrée, exécute le modèle et extrait les résultats.
- Le choix de float32 est fait ici pour simplifier la conversion et le debug initial. La quantification en int8 permettrait de réduire l'utilisation mémoire, mais requiert une calibration et un support spécifique dans le resolver.
- Le compteur `inference_count` permet de savoir combien d'inferences ont été réalisées depuis l'initialisation. Il peut être utilisé pour le profilage ou les statistiques.
- Si `Invoke()` échoue, il est utile de vérifier le modèle (format, version), la taille de `tensor_arena`, les opérations manquantes dans le resolver, ou la validité des entrées. Des messages via `MicroPrintf()` ou un port série aident au debug.

## 4 Prévision de température par CNN 1D pour l'embarqué

### 4.1 Contexte

Dans le cadre de la collecte et de l'exploitation locale de données environnementales, un capteur météorologique (station low-cost, LoRaWAN ou IoT WiFi) enregistre en continu température, pression, humidité et autres variables physiques. L'objectif est de prédire l'évolution de la température à court terme (ex. 4 heures), à partir d'une séquence de mesures passées (ex. 12 heures), grâce à un réseau neuronal compact (CNN 1D), optimisé pour être embarqué sur une carte ESP32-S3.

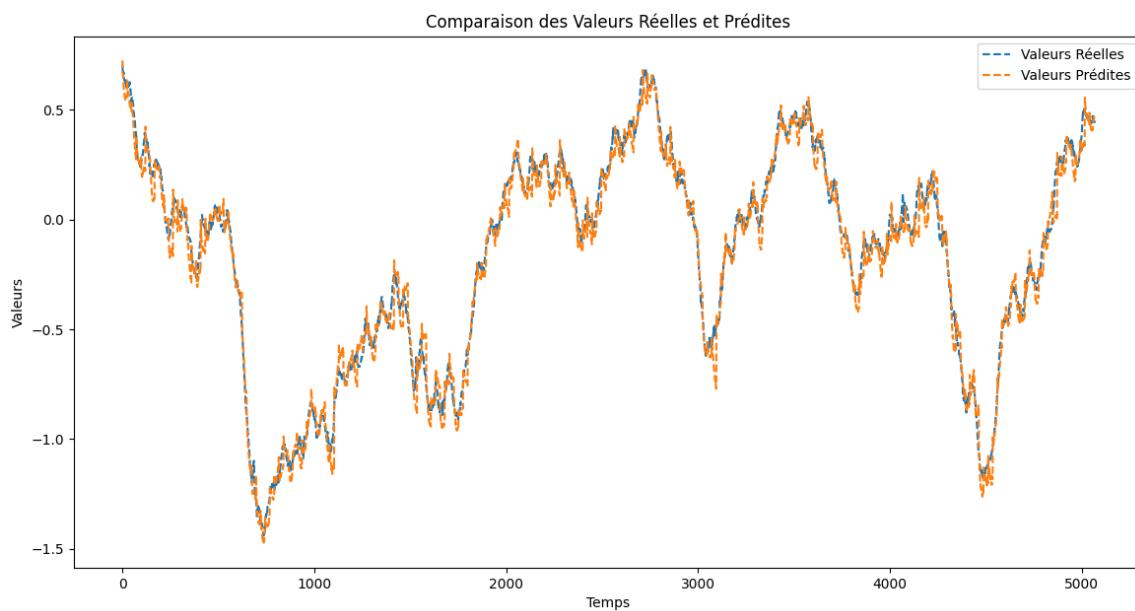


FIGURE 9 – Comparaison des valeurs réelles et prédictes par le modèle CNN 1D sur l'ensemble de test. L'axe des abscisses correspond à l'heure (échantillonnage horaire), l'axe des ordonnées à la température normalisée. Le modèle prédit l'évolution de la température sur un horizon de 4 heures à partir d'une séquence de 12 heures précédentes. Les courbes montrent que le réseau parvient à reproduire fidèlement la dynamique temporelle, même en présence de variations rapides ou lentes. Ce résultat illustre la pertinence d'une approche CNN embarquée pour la prévision locale à partir de capteurs physiques.

### 4.2 Travail demandé

L'exercice consiste à construire un modèle de deep learning compact, à l'entraîner sur un jeu de données météorologiques réel, et à préparer son export vers un microcontrôleur.

Le fichier `sorted_weather_data.csv` contient les mesures horaires d'une station météo (température, pression, humidité, etc.). La première étape est de transformer ces données brutes en entrées exploitables : normalisation des valeurs et création de variables cycliques pour l'heure, le jour et le mois (avec sinus/cosinus, pour rendre la représentation circulaire).

Les données sont ensuite organisées en séquences temporelles : pour chaque fenêtre glissante de

12h (72 pas), le modèle devra prédire la température des 4 heures suivantes (24 pas). On découpe ensuite les données en un ensemble d'entraînement (90%) et un ensemble de test (10%).

```

import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Creation du dataset pour le modèle convolutionnel (no accents)
def create_conv_dataset(dataset, n_timesteps, n_output):
    X, y = [], []
    for i in range(len(dataset) - n_timesteps - n_output + 1):
        x = dataset.iloc[i:i+n_timesteps].values
        X.append(x)
        y.append(dataset.iloc[i + n_timesteps:i + n_timesteps + n_output
                             , 1].values)
    return np.array(X), np.array(y)

df = pd.read_csv('./sorted_weather_data.csv')
df['dh_utc'] = pd.to_datetime(df['dh_utc'], format='%Y-%m-%d %H:%M:%S')
df['Year'] = df['dh_utc'].dt.year
df['Month'] = df['dh_utc'].dt.month
df['Day'] = df['dh_utc'].dt.day
df['Hour'] = df['dh_utc'].dt.hour
df.drop(['dh_utc', 'id_station'], axis=1, inplace=True)
df['hour_sin'] = np.sin(2 * np.pi * df['Hour'] / 24.0)
df['hour_cos'] = np.cos(2 * np.pi * df['Hour'] / 24.0)
df['day_sin'] = np.sin(2 * np.pi * df['Day'] / 31.0)
df['day_cos'] = np.cos(2 * np.pi * df['Day'] / 31.0)
df['month_sin'] = np.sin(2 * np.pi * df['Month'] / 12.0)
df['month_cos'] = np.cos(2 * np.pi * df['Month'] / 12.0)
df.drop(['Hour', 'Year', 'Day', 'Month'], axis=1, inplace=True)
df = df.fillna(0)
df.dropna(subset=['temperature'], inplace=True)
scaler = StandardScaler()
df_normalized = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)
df_train, df_test = train_test_split(df_normalized, test_size=0.1,
                                     shuffle=False)

```

Listing 9 – Préparation des données

Le modèle est un CNN simple, avec une unique couche Conv2D de noyau  $(5 \times 1)$ , suivie d'un MaxPooling2D, d'un Flatten, puis d'un Dense(24). Ce choix permet de capter les évolutions locales tout en maintenant une faible complexité, essentielle pour l'embarqué. On utilise Conv2D (et pas Conv1D) car chaque pas temporel comporte plusieurs capteurs : la 2<sup>e</sup>dimension du noyau "regarde" simultanément le temps et les différentes variables.

#### Questions à discuter :

- Pourquoi utiliser Conv2D au lieu de Conv1D dans ce cas précis ?
- En quoi la dimension des données (temps  $\times$  capteurs) influence-t-elle ce choix de convolution ?

Après 20 époques d'apprentissage, les performances sont évaluées sur l'ensemble de test. On

visualise les courbes température réelle vs prédictive pour juger la pertinence du modèle. Dans le cas de séries naturelles comme la météo, la prédiction à court terme reste soumise à de fortes incertitudes.

### Questions à discuter :

- À partir de combien d'heures les prédictions deviennent-elles manifestement peu fiables ?
- Comment évaluer si le modèle généralise bien, ou s'il a mémorisé des motifs locaux du jeu de données ?

```
# Create the model
model = Sequential([
    Conv2D(filters=32, kernel_size=(5, 1), activation='relu',
           input_shape=(n_timesteps, 10, 1)),
    MaxPooling2D(pool_size=(2, 1)),
    Flatten(),
    #Dense(24, activation='relu'),
    Dense(24, activation='linear')
])

model.compile(optimizer='adam', loss='MeanSquaredError')

# Train the model with callback
history = model.fit(X_train_conv, y_train_conv, epochs=20, batch_size
                      =32, validation_data=(X_test_conv, y_test_conv), verbose=1)

# Prediction on the test dataset
output_data = []
for i in range(0, len(df_test) - n_timesteps, n_output):
    print(f"Current index i: {i}")

    # Select rows i to i + n_timesteps from df_test for model input
    input_data = df_test.iloc[i:i + n_timesteps]

    # Convert to numpy array and add batch_size dimension
    input_data_conv = np.expand_dims(input_data.values, axis=0).reshape
        ((1, n_timesteps, 10, 1))

    # Predict values using the model
    output_data_conv = model.predict(input_data_conv)

    # Add predicted values to output_data list
    output_data.extend(output_data_conv.flatten())

# Convert the lists to numpy arrays for easier manipulation
output_data = np.array(output_data)

# Show output data shape and values
print("Output data shape:", output_data.shape)
print("Predicted values:", output_data)

import matplotlib.pyplot as plt
# Visualization of the results
plt.figure(figsize=(14, 7))
plt.plot(y_test_conv[:, 0], label='True Values', linestyle='--')
plt.plot(output_data[:len(y_test_conv)], label='Predicted Values',
```

```

        linestyle='--')
plt.xlabel('Time')
plt.ylabel('Values')
plt.title('Comparison of True and Predicted Values')
plt.legend()
plt.show()

```

Listing 10 – Entrainement et test du modèle

Une fois validé, le modèle est exporté dans trois formats : `SavedModel`, `.h5`, et `TFLite`. Il est ensuite converti en tableau hexadécimal `.cc` pour embarquement dans un firmware C++ sur ESP32.

Pour aller plus loin :

- Que gagne-t-on à quantifier le modèle en `int8`? Quel impact sur la précision ?
- Est-ce souhaitable dans ce contexte ?
- Pourquoi est-il utile d'extraire les types de tenseur utilisé par le modèle ?

```

# Save model in SavedModel format
model.export('model_local')
print("Model saved as 'saved_model/my_model'")

# Save model in H5 format
model.save('my_model.h5')
print("Model saved as 'my_model.h5'")

# Create TFLite converter
converter = tf.lite.TFLiteConverter.from_saved_model('model_local')

# Convert model
tflite_model = converter.convert()

# Save converted model
with open('model.tflite', 'wb') as f:
    f.write(tflite_model)

print("Model successfully converted to TensorFlow Lite format")

# Read TFLite model
with open('model.tflite', 'rb') as f:
    tflite_model = f.read()

# Convert model to hex string
hex_array = ', '.join(f'{b:02x}' for b in tflite_model)

# Create .cc file content
cc_content = f"""
#include <stdint.h>

const unsigned char model_tflite[] = {{
    {hex_array}
}};
const int model_tflite_len = {len(tflite_model)};
"""

```

```
# Write .cc file
with open('model.cc', 'w') as f:
    f.write(cc_content)

print("Model successfully converted to .cc file")

import tensorflow as tf

def get_ops_details(interpreter):
    ops_details = []
    tensor_details = interpreter.get_tensor_details()

    for detail in tensor_details:
        tensor_index = detail['index']
        tensor_name = detail['name']
        tensor_shape = detail['shape']
        tensor_dtype = detail['dtype']
        ops_details.append({
            'index': tensor_index,
            'name': tensor_name,
            'shape': tensor_shape,
            'dtype': tensor_dtype
        })

    return ops_details

def get_operator_details(interpreter):
    op_details = []
    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()
    all_tensor_details = interpreter.get_tensor_details()

    # Map tensor index to tensor name for easy reference
    tensor_index_to_name = {tensor['index']: tensor['name'] for tensor
                           in all_tensor_details}

    for input_detail in input_details:
        op_details.append({
            'type': 'Input',
            'name': tensor_index_to_name[input_detail['index']],
            'shape': input_detail['shape'],
            'dtype': input_detail['dtype']
        })

    for output_detail in output_details:
        op_details.append({
            'type': 'Output',
            'name': tensor_index_to_name[output_detail['index']],
            'shape': output_detail['shape'],
            'dtype': output_detail['dtype']
        })

    return op_details

# Load TFLite model
interpreter = tf.lite.Interpreter(model_path='model.tflite')
interpreter.allocate_tensors()
```

```

# Get tensor details
tensor_ops_details = get_ops_details(interpreter)

# Get operator details
op_details = get_operator_details(interpreter)

# Print tensor details
print("Tensor details:")
for tensor in tensor_ops_details:
    print(f"Index: {tensor['index']}, Name: {tensor['name']}, Shape: {tensor['shape']}, Dtype: {tensor['dtype']}")

# Print operator details
print("\nOperator details:")
for op in op_details:
    print(f"Type: {op['type']}, Name: {op['name']}, Shape: {op['shape']}, Dtype: {op['dtype']}")
    
```

Listing 11 – Export du modèle pour l'embarqué

### 4.3 Un exemple déployé sur ESP32

Cette approche a été testée en condition réelle. Le modèle CNN 1D est entraîné à partir de données publiques, converti au format **TFLite**, puis intégré à un firmware sans OS pour ESP32-S3. Le boîtier autonome (voir Figure 10) embarque un capteur environnemental et un module RTC. L'inférence complète s'effectue en moins de 200 ms, pour moins de 50 kB de RAM consommés, ce qui démontre la pertinence de ce type de modèle pour des systèmes embarqués à ressources limitées.



FIGURE 10 – Déploiement réel d'un modèle CNN 1D pour la prévision de température sur ESP32. Le dispositif mesure température, pression, humidité, et calcule la température de rosée. L'horloge RTC fournit les variables temporelles. <https://selvasystems.net/deep-learning>

#### 4.4 Comparaison des architectures séquentielles

Modèle	Taille	Vitesse	Déploiement	Performance sur tâche simple
CNN	★★★	★★★★	★★★★	★★★★
LSTM	★★	★★	★	★★★★
Transformer	★★★★	★	Incompatible	★★★★

FIGURE 11 – Comparatif des réseaux séquentiels pour le traitement embarqué. Les CNN sont souvent le meilleur compromis entre simplicité, efficacité et compatibilité matérielle.

#### À retenir

Les CNN sont parfaitement adaptés à l'analyse de séquences temporelles embarquées. Bien que des architectures plus complexes existent (LSTM, Transformers), elles sont rarement justifiées dans ce contexte : plus lentes, plus lourdes, et souvent pas plus performantes. Des CNN simples à kernels étroits, profonds, et bien entraînés permettent déjà une détection efficace de motifs temporels dans des flux de données capteurs.

#### 4.5 Ressources associées

- Code source du projet : [https://github.com/Bertrand-selvasystems/DeepLearning\\_ESP32\\_S3](https://github.com/Bertrand-selvasystems/DeepLearning_ESP32_S3)
- Présentation détaillée : <https://selvasystems.net/deep-learning>
- Données d'entraînement : à fournir par l'enseignant

##### 4.5.1 Questions de comprehension

- Pourquoi utiliser Conv2D au lieu de Conv1D dans ce cas précis ?
- En quoi la dimension des données (temps x capteurs) influence-t-elle ce choix de convolution ?
- A partir de combien d'heures les predictions deviennent-elles manifestement peu fiables ?
- Comment évaluer si le modèle généralise bien, ou s'il a memorisé des motifs locaux du jeu de données ?
- Que gagne-t-on à quantifier le modèle en int8 ? Quel impact sur la précision ?
- Est-ce souhaitable dans ce contexte embarqué ?
- Pourquoi est-il utile d'extraire les types de tenseur utilisés par le modèle ?

##### 4.5.2 Éléments de réponse

- Conv2D permet d'exploiter à la fois la structure temporelle (1ère dimension) et la structure multivariée (2ème dimension : température, pression, humidité, etc.). Conv1D ne pourrait filtrer que sur le temps, en considérant toutes les variables comme une seule série plate.

- Le choix de la convolution bidimensionnelle est directement lié au format ( $n\_timesteps \times n\_variables$ ). Cela permet au filtre de capturer conjointement des motifs dans le temps et entre les capteurs.
- Sur le graphe (Figure 9), les prédictions deviennent moins fiables au-delà de 2 ou 3 heures dans certains cas. Cela reste variable selon la dynamique locale du signal.
- Une bonne généralisation se vérifie si les prédictions restent cohérentes sur l'ensemble de test (non vu pendant l'apprentissage), et si le modèle ne surestime pas certains motifs. La présence de bruit, ou de séquences aléatoires, met aussi en évidence la capacité du modèle à abstraire.
- La quantification en `int8` permet de réduire drastiquement la taille du modèle (par un facteur 4) et son empreinte mémoire en inference. Elle permet aussi d'accélérer les calculs sur certains MCU.
- Dans le cas d'une inference sur ESP32, la quantification est souvent souhaitable pour passer sous les 100 kB de RAM et tenir dans le `tensor_arena`. Mais elle peut réduire la précision, en particulier sur des données bruitées.
- Extraire les types et formes des tenseurs permet de configurer correctement l'inference embarquée : alignement mémoire, vérifications, et debug. Cela permet aussi de savoir si le modèle est quantifié, et avec quel format exact.

## 5 Detection et localisation d'entités extraterrestres par vision par ordinateur

Dans cet exercice, nous allons apprendre à détecter et localiser une entité dans une image couleur, en réutilisant un modèle de computer vision pré-entraîné (MobileNetV2) et en ajoutant une tête dense personnalisée pour effectuer à la fois de la régression (coordonnées x, y) et de la classification binaire (style de l'entité). Ce protocole est typique de l'apprentissage par transfert (transfer learning), très utilisé quand on ne dispose que d'un faible volume de données annotées.

Le but est d'extraire des représentations visuelles génériques (features) avec un modèle gelé, et d'optimiser uniquement les couches de sortie sur la tâche cible.

### 5.1 Code détaillé : détection + localisation avec MobileNetV2 gelé

```

import os
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.utils import load_img, img_to_array
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import Input, Dense, GlobalAveragePooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# ----- Configuration générale -----
IMAGE_DIR = "images"          # Dossier contenant les images RGB
CSV_FILE = "labels.csv"        # Fichier CSV : nom de fichier, x, y, style
IMG_SIZE = (224, 224)          # Dimension des images d'entrée pour
                              # MobileNetV2
EPOCHS = 50                   # Nombre maximal d'epochs
BATCH_SIZE = 32                # Taille de batch

# ----- Chargement et prétraitement -----
# Le fichier CSV doit contenir 4 colonnes : filename, x, y, style_cible
df = pd.read_csv(CSV_FILE, header=None, names=["filename", "x", "y", "style_cible"])

def preprocess_image(filename):
    """Charge une image RGB, la redimensionne et l'applique le
       prétraitement MobileNetV2."""
    path = os.path.join(IMAGE_DIR, filename)
    img = load_img(path, target_size=IMG_SIZE)
    img = img_to_array(img)
    img = tf.keras.applications.mobilenet_v2.preprocess_input(img)
    return img

# On charge toutes les images et on construit les targets
images = np.array([preprocess_image(fname) for fname in df["filename"]])

# Les coordonnées x, y sont normalisées dans [0,1] pour ne pas dépendre
# de la taille image
targets_xy = df[["x", "y"]].values.astype("float32") / [320, 240]
# style_cible : label binaire (0 ou 1), reshape pour un batch (N,1)

```

```

targets_style = df["style_cible"].values.astype("float32").reshape(-1,
    1)
# On regroupe tout dans un seul tableau pour faciliter le split
targets_all = np.concatenate([targets_xy, targets_style], axis=1)

# ----- Split train/test -----
# Split unique et stratifie pour garder la proportion des classes (
#   style_cible)
X_train, X_test, y_train_all, y_test_all = train_test_split(
    images, targets_all,
    test_size=0.2, stratify=targets_style, random_state=42
)
# On extrait séparément les cibles régression (xy) et classification (
#   style)
y_train_xy = y_train_all[:, :2]
y_train_style = y_train_all[:, 2].reshape(-1, 1)
y_test_xy = y_test_all[:, :2]
y_test_style = y_test_all[:, 2].reshape(-1, 1)

print("Taille du set d'entraînement :", X_train.shape)
print("Taille du set de test :", X_test.shape)
print("y_train_xy shape :", y_train_xy.shape)
print("y_train_style shape :", y_train_style.shape)

# ----- Construction du modèle -----
# On charge MobileNetV2 pré-entraîné sur ImageNet, sans la tête d'
#   origine
base_model = MobileNetV2(input_shape=(*IMG_SIZE, 3), include_top=False,
    weights="imagenet")
base_model.trainable = False      # On gèle les poids, le backbone ne sera
    pas mis à jour

# Input layer
inputs = Input(shape=(*IMG_SIZE, 3), name="input_image")
# Extraction des features avec MobileNetV2
x = base_model(inputs, training=False)
x = GlobalAveragePooling2D()(x)
x = Dense(256, activation="relu")(x)    # Première couche dense pour
    extraire des représentations plus abstraites
x = Dense(128, activation="relu")(x)    # Seconde couche dense
# Deux sorties parallèles
xy_output = Dense(2, activation="sigmoid", name="xy_output")(x)
    # Régression sur x, y (normalisé)
style_output = Dense(1, activation="sigmoid", name="style_output")(x)
    # Classification binaire

# Modèle multi-sortie
model = Model(inputs=inputs, outputs={"xy_output": xy_output, "
    style_output": style_output})

# Compilation avec des pertes adaptées à chaque sortie
model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-4),
    loss={
        "xy_output": "mse",                                # Pour la régression (x,
        "y")                                             
        "style_output": "binary_crossentropy"  # Pour la classification
        binaire
    }
)

```

```

},
loss_weights={
    "xy_output": 1.0,
    "style_output": 0.1      # On ponder le style pour ne pas
                           # desequilibrer l'apprentissage
},
metrics={
    "xy_output": "mae",
    "style_output": "accuracy"
}
)

print("Résumé du modèle :")
model.summary()

# Callback pour stopper si pas d'amélioration sur validation
callbacks = [EarlyStopping(patience=5, restore_best_weights=True)]

# ----- Entrainement du modèle -----
print("Début de l'entraînement ...")
history = model.fit(
    X_train,
    {"xy_output": y_train_xy, "style_output": y_train_style},
    validation_data=(X_test, {"xy_output": y_test_xy, "style_output":
        y_test_style}),
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    callbacks=callbacks
)

# ----- Sauvegarde du modèle -----
print("Sauvegarde du modèle en .h5 ...")
model.save("mobilenetv2_xy_style_model.h5")

# ----- Prediction et évaluation sur le test set
-----
y_pred = model.predict(X_test)
# Selon la version de TensorFlow, la sortie peut être dict ou tuple
if isinstance(y_pred, dict):
    y_pred_xy = y_pred["xy_output"]
    y_pred_style = y_pred["style_output"]
else:
    y_pred_xy, y_pred_style = y_pred

y_true_xy = y_test_all[:, :2]
y_true_style = y_test_all[:, 2]
y_pred_style_bin = (y_pred_style > 0.5).astype("float32").flatten()

from sklearn.metrics import mean_absolute_error, accuracy_score

# Calcul de la MAE sur les coordonnées, ramenées en pixels
mae_xy = mean_absolute_error(y_true_xy * [320, 240], y_pred_xy * [320,
    240])
acc_style = accuracy_score(y_true_style, y_pred_style_bin)

print(f"Erreur moyenne absolue (MAE) sur (x, y) : {mae_xy:.2f} pixels")
print(f"Précision (accuracy) sur style_cible : {acc_style:.2%}")

```

```

# ----- Visualisation des résultats -----
def show_predictions(n=5):
    """Affiche n predictions aleatoires avec verite terrain et
    prediction."""
    indices = np.random.choice(len(X_test), n, replace=False)
    for i in indices:
        img = X_test[i]
        true_xy = y_true_xy[i] * [224, 224]
        pred_xy = y_pred_xy[i] * [224, 224]
        true_style = int(y_true_style[i])
        pred_style = int(y_pred_style_bin[i])

        plt.imshow((img + 1) / 2) # Les images sont dans [-1,1], on les
                                # ramene dans [0,1] pour affichage
        plt.scatter(*pred_xy, c="red", label="prediction")
        plt.scatter(*true_xy, c="green", marker="x", label="verite")
        plt.title(f"Style vrai : {true_style} | Style predit : {
                  pred_style}")
        plt.legend()
        plt.axis("off")
        plt.show()

# On affiche 5 exemples aleatoires
show_predictions(5)

```

Listing 12 – Pipeline complet de détection et localisation avec MobileNetV2

## 5.2 Explications détaillées

Dans cette approche, on commence par exploiter MobileNetV2, un modèle pré-entraîné sur ImageNet, dont les poids sont figés ("gelés") afin de profiter de sa capacité à détecter des formes, textures et motifs génériques, tout en évitant de le suradapter au petit jeu de données cible. Au sommet de ce modèle, on ajoute une tête dense personnalisée composée de deux branches distinctes : l'une est dédiée à la prédiction des coordonnées (régression, valeurs continues), l'autre à la classification binaire du style de l'entité. Ce schéma permet de partager les représentations extraites tout en optimisant simultanément deux objectifs différents.

Les coordonnées x et y sont normalisées en les divisant par la largeur et la hauteur de l'image, ce qui rend la régression indépendante du format d'entrée et améliore la stabilité de l'entraînement. La séparation des données en ensemble d'apprentissage et de test est réalisée de manière stratifiée : la proportion de chaque classe (**style\_cible**) est conservée, ce qui garantit l'absence de biais dans l'évaluation finale.

Pour l'optimisation, le modèle combine deux fonctions de perte : l'erreur quadratique moyenne (MSE) pour la localisation et la cross-entropie binaire pour la classification. La perte de classification est pondérée de façon à ce que la régression ne soit pas négligée, car la cross-entropie peut dominer numériquement. Un mécanisme d'EarlyStopping est mis en place : l'entraînement s'arrête automatiquement si la performance en validation cesse de s'améliorer, ce qui limite le surapprentissage.

L'évaluation finale de la localisation s'exprime en pixels grâce à la MAE (erreur absolue moyenne), fournissant ainsi un indicateur concret de la qualité de la prédiction. Enfin, la visualisation des résultats se fait en affichant les images de test avec la position prédite (point rouge) et la vérité

terrain (croix verte), ce qui permet d'interpréter visuellement les écarts entre la prédiction et la réalité.

### 5.3 Questions de comprehension

1. Pourquoi doit-on figer les poids de MobileNetV2 lors de l'apprentissage sur ce jeu de données ?
2. Quel est l'avantage d'utiliser un backbone pre-entraîné plutôt que d'apprendre tout le modèle à partir de zéro ?
3. Pourquoi la normalisation des coordonnées  $(x, y)$  est-elle cruciale pour la régression ?
4. Que se passerait-il si on utilisait la croix-entropie binaire pour les coordonnées  $(x, y)$  au lieu de la MSE ?
5. Pourquoi faut-il pondérer la perte de classification par rapport à la régression ?
6. Quelles stratégies permettaient d'interpréter visuellement si le modèle se trompe sur la localisation ?
7. Pourquoi le split stratifié est-il important, même si le dataset est petit ?
8. Comment expliquer un score de classification très élevé mais une localisation très mauvaise ?
9. En quoi EarlyStopping contribue-t-il à la robustesse du modèle obtenu ?
10. Que faudrait-il changer pour effectuer du fine-tuning (adaptation des features du backbone au jeu cible) ?
11. Si on disposait de beaucoup plus d'images, quels ajustements proposeriez-vous sur le pipeline ?

### 5.4 Éléments de réponse développés

1. Figer les poids permet d'éviter que le backbone MobileNetV2 n'oublie les représentations générales apprises sur un dataset massif. On limite ainsi le surapprentissage, surtout si notre dataset est réduit et ne couvre pas la diversité de ImageNet.
2. Utiliser un backbone pré-entraîné apporte des features robustes déjà apprises. Cela accélère la convergence et permet d'obtenir de bonnes performances même avec peu de données cibles.
3. La normalisation  $(x, y)$  (division par largeur et hauteur image) permet de standardiser l'échelle du problème, de rendre le modèle plus stable et d'accélérer la descente de gradient.
4. La croix-entropie n'est pas adaptée à une sortie continue. Elle force le modèle à produire des valeurs entre 0 et 1, interprétées comme probabilités, ce qui n'a pas de sens pour des coordonnées spatiales réelles.
5. Sans pondération, la classification pourrait dominer la perte totale et le modèle ne progresserait pas sur la localisation. La pondération ajuste l'importance relative des deux tâches pendant l'apprentissage.
6. On peut superposer les prédictions et vérités terrains sur les images testées : si les points rouges sont proches des croix vertes, la localisation est correcte ; sinon, il y a une erreur visible.
7. Le split stratifié assure que la proportion de chaque classe (ici, le style de l'entité) est la même dans le train et le test. Sinon, on risque de surestimer ou sous-estimer la performance sur des classes rares.
8. Un bon score de classification mais une mauvaise localisation indique que le modèle arrive à reconnaître la présence/l'absence d'une entité, mais ne sait pas la situer avec précision (mauvaise régression).

9. EarlyStopping limite le risque de surapprentissage sur le set d'entraînement, et garantit que les poids retenus sont ceux correspondant à la meilleure performance sur le set de validation.
10. Pour du fine-tuning, il suffit de débloquer (mettre trainable=True) tout ou partie du backbone, souvent les derniers blocs, et d'appliquer un learning rate plus faible pour éviter d'effacer les connaissances acquises.
11. Si le dataset devient très important, on pourrait augmenter la taille des batchs, re-entraîner (partiellement ou totalement) MobileNetV2, utiliser des data augmentations plus poussées, et réduire l'effet d'early stopping.

## 6 Détection de mot-clé (Keyword Spotting, KWS) par CNN sur Spectrogrammes : entraînement

### 6.1 Contexte et organisation de l'exercice

Le **Keyword Spotting** (KWS) est une brique fondamentale des systèmes de reconnaissance vocale embarqués. Son principe est de détecter, dans un flux audio continu, l'apparition d'un mot-clé prédéfini, par exemple pour déclencher un assistant vocal. Dans une architecture dite *en cascade (cascading)*, un microcontrôleur ultra-basse consommation reste en veille et analyse en continu le flux audio. Dès qu'un mot-clé est détecté, le système complet est réveillé pour traiter la commande vocale complète. Cela permet de concilier réactivité et faible consommation énergétique.

Le jeu de données est structuré autour de trois répertoires :

- **BRUIT** : extraits de bruit de fond ou silence,
- **LUMIERE** : extraits contenant le mot-clé « lumière »,
- **INCONNU** : extraits de mots ou sons non pertinents.

Chaque extrait dure 1 seconde et est stocké sous forme de fichier `.csv` représentant un spectrogramme. Il y a environ 150 fichiers par dossier, soit au total environ 450 exemples. Ce volume de données reste modeste : il s'agit ici d'un jeu de démonstration, loin des jeux massifs utilisés pour l'apprentissage profond en production.

### 6.2 Commentaires sur le pipeline et le code

#### Préparation et chargement des données :

- Pour chaque dossier, le script parcourt les fichiers `.csv` et convertit leur contenu (matrice de spectrogramme) en tableau numpy. À chaque exemple est associé un label numérique correspondant à la classe (`BRUIT` → 0, `INCONNU` → 1, `LUMIERE` → 2).
- Tous les exemples sont stockés dans un tableau de données `X`, et leurs labels dans `y`. On conserve aussi le nom du fichier et le dossier d'origine pour d'éventuelles analyses d'erreur.
- Avant l'apprentissage, les données sont mélangées puis séparées en deux ensembles : un jeu d'apprentissage (80 %) et un jeu de test (20 %), afin de mesurer la capacité du modèle à généraliser.

#### Prétraitement pour le modèle :

- Chaque spectrogramme est traité comme une « image » en niveaux de gris. Il faut donc lui ajouter une dimension « canal » (1 seule couche), afin d'être compatible avec les couches convolutionnelles 2D de Keras/TensorFlow.

#### Modélisation :

- Le modèle retenu est un CNN très compact, adapté aux ressources limitées de l'embarqué (type ESP32) : deux couches convolutionnelles (avec max pooling), suivies d'un aplatissement et d'une couche dense, puis d'une sortie softmax à 3 classes.
- L'entraînement du modèle prend en compte un poids de classe pour mieux équilibrer les erreurs sur les classes sous-représentées (par exemple, donner plus d'importance à la détection du mot-clé « lumière » si c'est le but principal du système).
- On utilise la perte `sparse_categorical_crossentropy` adaptée à la classification multi-classes avec des labels entiers.

**Évaluation :**

- Après entraînement, la performance du modèle est mesurée par la précision sur le jeu de test, ainsi qu'avec une matrice de confusion qui permet d'analyser finement la nature des erreurs (confusion entre bruit et inconnu, détection correcte ou non du mot-clé, etc.).

**Déploiement :**

- Une fois entraîné, le modèle est converti au format **TFLite** (TensorFlow Lite), qui permet de l'exécuter efficacement sur un microcontrôleur ou une cible embarquée. La quantification et le pruning sont possibles pour optimiser encore le modèle.

### 6.3 Questions

1. Pourquoi utiliser des spectrogrammes comme représentation d'entrée du réseau, au lieu des MFCC ?
2. Comment sont organisées les données et comment les charge-t-on pour le machine learning ?
3. Pourquoi sépare-t-on les données en deux jeux distincts : apprentissage et test ?
4. Pourquoi faut-il ajouter une dimension supplémentaire à chaque entrée avant de l'envoyer dans le réseau ?
5. Décrivez brièvement l'architecture du modèle CNN utilisé. Pourquoi cette simplicité ?
6. Quel est l'intérêt d'utiliser des poids de classe lors de l'apprentissage ?
7. Comment interpréter la matrice de confusion obtenue après l'évaluation du modèle ?
8. Quel est l'intérêt de convertir le modèle entraîné au format TFLite ?
9. Expliquez brièvement ce que sont la quantification et le pruning, et pourquoi ces techniques sont essentielles pour le deep learning embarqué.

### 6.4 Corrigé des réponses

1. Les spectrogrammes fournissent une image « brute » de l'évolution fréquentielle du signal audio, ce qui permet aux réseaux convolutifs (CNN) d'apprendre directement les motifs discriminants. Ils sont simples à générer, adaptés à l'embarqué et laissent plus de liberté au réseau pour apprendre ce qui est pertinent. À l'inverse, les MFCC filtrent et résument l'information de façon plus contraignante.
2. Les exemples audio sont répartis dans trois dossiers selon leur classe (**BRUIT**, **INCONNU**, **LUMIERE**). Chaque fichier CSV représente un spectrogramme associé à un extrait d'audio d'une seconde. On lit chaque fichier en mémoire et on associe chaque matrice de spectrogramme à un label numérique, regroupés dans deux grands tableaux de données.
3. Cette séparation permet d'évaluer la capacité du modèle à généraliser à des exemples inédits. Sans cela, on ne mesure que la capacité du réseau à mémoriser les exemples vus pendant l'entraînement.
4. Les couches convolutionnelles attendent des images à 4 dimensions (nombre d'exemples, hauteur, largeur, canaux). Pour des spectrogrammes (niveaux de gris), il faut ajouter une dimension canal de taille 1.
5. Le modèle comprend deux couches convolutionnelles avec activation ReLU et pooling, suivies d'un aplatissement, d'une couche dense, puis d'une couche de sortie softmax à trois classes. Cette architecture compacte est suffisante pour la tâche et compatible avec les contraintes matérielles de l'embarqué.

6. Si une classe (par exemple, le mot-clé) est rare, le réseau risque de l'ignorer. En augmentant son poids lors de l'entraînement, on équilibre l'importance des classes et on améliore la détection des cas minoritaires.
7. Chaque ligne correspond à la vraie classe, chaque colonne à la prédiction. Les valeurs diagonales indiquent les bonnes prédictions, les valeurs hors diagonale les confusions. Cela permet d'identifier les erreurs typiques du modèle (par exemple, confondre « bruit » et « lumière »).
8. Le format TFLite est conçu pour l'exécution sur microcontrôleurs ou petits systèmes embarqués. Il réduit la taille du modèle, accélère l'inférence et s'intègre facilement à l'ESP32 ou autres cibles edge.
9. **Quantification** : réduire la précision des nombres stockant les poids du modèle (par exemple, de 32 bits à 8 bits), ce qui diminue la mémoire requise et accélère les calculs.  
**Pruning** : supprimer les connexions ou filtres les moins importants, ce qui allège le modèle et le rend plus rapide. Ces techniques sont essentielles pour adapter les modèles aux contraintes des dispositifs embarqués.

*Remarque pédagogique : Ce jeu de données volontairement limité met l'accent sur la méthodologie, la compréhension du pipeline d'un système KWS, et la maîtrise des contraintes spécifiques à l'embarqué. Il permet de faire les entraînements dans un temps contenu et permet ainsi de comparer différentes solutions pour la structure du réseau.*



## Formation IA légère : Deep Learning pour l'embarqué

---

Pour tout complément d'information, Selva Systems reste à votre disposition :  
[contact@selvasystems.net](mailto:contact@selvasystems.net) / 06 77 02 79 30.