Université de Perpignan Via Domitia (UPVD)
UFR Sciences Exactes et Expérimentales - SEE
Département d'Informatique

Projet du module HPC pour l'Intelligence Artificielle

— Projet 2 —

Conduite autonome : Reconnaissance des panneaux de signalisation

Réalisé par

— Bouzidi Adel

2025/2026

Table des matières

1 Introduction

1.1 Contexte et objectifs du projet

Dans la conduite autonome la reconnaissance automatique des panneaux de signalisation est un point fondamentale. Un véhicule autonome doit etre capable d'identifier de manière fiable et rapide le type de panneau observé afin d'adapter la conduite (interdictions, priorité, vitesse, etc.). Sur des conditions réelles, la teche devient difficile à cause de différents facteurs : variations de luminosité (surexposition, ombre, jour/nuit), angle de prise de vue, flou de mouvement, ou encore une résolution insuffisante de l'objet d'intéret.

Ce projet a pour objectif la conception d'un modèle de *deep learning* reposant sur des réseaux de neurones convolutionnels (CNN) pour classifier des images de panneaux routiers. Le jeu de données utilisé est **GTSRB (German Traffic Sign Recognition Benchmark)**, qui comprend **43 classes** de panneaux. Le pipeline attendu englobe l'ensemble des étapes : a) importation des données, b) prétraitement, c) entrainement, d) évaluation, et c) inférence, incluant une analyse critique des résultats et une étude des confusions entre classes visuellement proches.

1.2 Présentation du sujet

Un modèle CNN a été utilisé pour traiter un problème concret de vision par ordinateur. Une approche **from scratch** a été mise en œuvre afin de : (i) de maitriser bien le pipeline, (ii) d'obtenir une architecture explicable couche par couche et (iii) de faciliter l'analyse des erreurs et des confusions liées à la structure du modèle.

1.3 Environnement de travail

Le développement et l'exécution ont été réalisés dans un environnement linux en utilisant **WSL2** avec un environnement virtuel Python **venv**. L'objectif consistait à mettre en place un environnement reproductible et isolé.

Répertoire de travail. Le projet a été organisé dans le dossier :

```
/home/adel/hpc_pour_ia/
```

Arborescence principale. Après l'installation et la récupération des données, l'arborescence principale du projet est :

```
README.md
data/
figures/
models/
notebooks/
requirements.txt
src/
```

Versions et dépendances. L'environnement Python utilise un `.venv` et Les versions identifiées au cours des tests incluent notamment :

— TensorFlow : `2.20.0`
— NumPy : `1.26.4`
— OpenCV : `4.11.0`

Remarque :
Lors de l'exécution TensorFlow identifie l'absence des pilotes CUDA sur le système hote, à cause de ça l'entraînement a été effectué sur le processeur (CPU), comme l'indiquent les messages du type : `Could not find cuda drivers on your machine, GPU will not be used`. Cette limitation n'empeche pas l'entraînement mais implique une augmentation du temps d'exécution.

1.4 Dataset et organisation des données

Le dataset GTSRB a été obtenu via Kaggle et extrait localement dans le répertoire `data/`. Après le téléchargement et de la décompression, le dossier `data/` est constitué de répertoires d'images ainsi que de fichiers CSV qui décrivent les données :

```
data/
  Meta/      Meta.csv
  Train/     Train.csv
  Test/      Test.csv
  meta/      train/      test/
```

Remarque sur les données :
Dans ce projet le pipeline a été conçu en s'appuyant sur les chemins spécifiés dans les fichiers CSV (`Train.csv`, `Test.csv`, `Meta.csv`) et sur les répertoires correspondants (`Train/`, `Test/`, `Meta/`).

1.5 Organisation du pipeline

Afin de respecter la contrainte « pipeline complet dans un fichier .py ou ipynb », le code a été structuré sous forme de modules Python exécutables, avec des points d'entrée simples :

— `src/data.py` : a) chargement des CSV, b) lecture des images, c) redimensionnement, d) normalisation, e) création des datasets TensorFlow.
— `src/train.py` : a) définition du modèle CNN, b) entrainement, c) sauvegarde du meilleur modèle et du dernier modèle.
— `src/evaluate.py` : a) évaluation sur validation, b) rapport de classification, c) matrice de confusion.

Les commandes utilisées pour exécuter les phases d'apprentissage et d'évaluation sont :

```
python -m src.train
python -m src.evaluate
```

Sorties générées :
Une fois le pipeline exécuté les éléments obtenus sont :

— modèles entrainés : `models/baseline_cnn.keras` (meilleur), `models/baseline_cnn_last.keras` (dernier) ;
— figures : `figures/confusion_matrix.png` ;
— rapports : `reports/classification_report.txt`, `reports/top_confusions.csv`.

2 Jeu de données et prétraitement

2.1 Présentation du jeu de données GTSRB

Comme déjà indiqué, le jeu de données utilisé dans le cadre de ce projet est le *German Traffic Sign Recognition Benchmark* (GTSRB). Il s'agit d'un dataset de référence dans le domaine de la vision par ordinateur appliquée à la conduite autonome. Il contient des images de panneaux de signalisation allemands réparties en **43 classes distinctes**, associés à différents types de panneaux (limitation de vitesse, priorité, interdiction, danger, etc...).

Chaque image est associée à une étiquette de classe (`ClassId`) et à des informations géométriques décrivant la région d'intérêt associée (*Region of Interest*, ROI). le jeu de données contient des variations significatives en termes de :

— conditions d'éclairage,
— orientation et perspective des panneaux,
— qualité visuelle (flou, bruit, compression).

Ces variations contribuent à rendre la tache de classification réaliste et représentative d'un cas d'usage réel en conduite autonome.

2.2 Organisation des données et fichiers CSV

Les données sont organisées dans le dossier `data/` qui contient à la fois les images et les fichiers CSV décrivant les annotations correspondantes. Les principaux fichiers utilisés dans ce projet sont :

— `Train.csv` : métadonnées des images d'entrainement ;
— `Test.csv` : métadonnées des images de test ;
— `Meta.csv` : informations complémentaires sur les classes (forme, couleur, identifiant du panneau).

Le fichier `Train.csv` comprend notamment les colonnes suivantes :

— `Path` : c'est le chemin relatif vers l'image.
— `ClassId` : étiquette de la classe (entier entre 0 et 42).
— `Width`, `Height` : dimensions originales de l'image.
— `Roi.X1`, `Roi.Y1`, `Roi.X2`, `Roi.Y2` : coordonnées de la région d'intéret.

Les images utilisées pour l'entrainement sont stockées dans le répertoire `data/Train/`, organisé en sous dossiers numérotés de `0` à `42`, chaque sous dossier est associé à une classe spécifique.

2.3 Analyse du dataset

Une analyse exploratoire préliminaire a permis de vérifier la cohérence des données et de montrer certaines caractéristiques du dataset.

Nombre de classes :

On dispose, dans l'ensemble d'entrainement de **43 classes** différentes. Ce qui est confirmé à la fois par l'organisation des répertoires et par le contenu du fichier `Train.csv`.

Taille des images :

Les images du dataset n'ont pas toutes la même taille comme le montre un échantillon aléatoire :

— dimensions minimales observées : environ 25×26 pixels
— dimensions maximales observées : jusqu'à 191×168 pixels
— plusieurs centaines de tailles différentes sont présentes dans l'ensemble d'entraînement.

Cette différence de taille rend nécessaire une étape de redimensionnement avant l'entraînement du réseau de neurones

2.4 Séparation entrainement - validation

Afin d'évaluer correctement les performances du modèle les données d'entraînement ont été réparties en deux sous-ensembles séparés : un ensemble d'entrainement et un ensemble de validation.

Cette séparation est effectuée dans le script `src/data.py`. À partir du fichier `Train.csv`, les données sont mélangées de manière reproductible, puis partitionnées selon une proportion de **90% pour l'entraînement et 10% pour la validation**. Les deux sous-ensembles obtenus ont par la suite sauvegardés dans les fichiers suivants :

— `data/train_split.csv`
— `data/val_split.csv`

Cette étape garantit que la séparation est réalisée une seule fois et peut être réutilisée à l'identique lors de tous les entrainements ultérieurs assurant ainsi la reproductibilité des expériences. Les statistiques finales obtenues sont :

— 35 288 images pour l'entrainement ;
— 3 921 images pour la validation ;
— 43 classes présentes dans les deux sous-ensembles.

La différence maximale de distribution entre les classes est inférieure à 3×10^{-5} ce qui montre une répartition presque identique entre l'entraînement et la validation.

2.5 Prétraitement des images

Le prétraitement des images est implémenté dans le script `src/data.py`. Les opérations de prétraitement appliquées à chaque image sont les suivantes :

— lecture de l'image depuis le disque à partir du chemin relatif contenu dans les fichiers CSV,
— conversion du format BGR (utilisé par OpenCV) vers le format RGB,
— redimensionnement de l'image à une taille fixe de 32×32 **pixels**,
— normalisation des valeurs de pixels dans l'intervalle $[0, 1]$.

Le redimensionnement est nécessaire car les images du dataset n'ont pas toutes la même taille tandis que la normalisation aide à l'entraînement du réseau en harmonisant l'échelle des entrées.

2.6 Pipeline de chargement avec `tf.data`

Pour garantir un chargement efficace des données et de bonnes performances à l'entrainement un pipeline reposant sur l'API `tf.data` de TensorFlow a été mis en œuvre dans le script `src/data.py`.

Ce pipeline effectue les opérations suivantes :

— création d'un dataset à partir des chemins d'images et des labels contenus dans les fichiers CSV,
— application du prétraitement image via une fonction Python encapsulée dans `tf.py_function`,
— mise en batch des données,
— mélange des données (`shuffle`) pour l'entraînement,
— préchargement asynchrone (`prefetch`) afin de réduire les temps d'attente entre les itérations.

À la sortie de ce pipeline les données sont transmises au modèle sous forme de tenseurs compatibles avec un réseau de neurones convolutionnel :

$$X \in \mathbb{R}^{(batch\_size,32,32,3)}, \quad y \in \mathbb{N}^{(batch\_size)}$$

Cette approche permet de bien séparer la préparation des données (dans `src/data.py`) de l'entrainement du modèle présenté dans la section suivante.

Lien vers les scripts :

Les étapes détaillées dans cette section sont implémentées dans :

— `src/data.py` : séparation train/validation, prétraitement, pipeline `tf.data`.

3 Entrainement et architecture du réseau convolutionnel

3.1 Scripts d'entrainement

L'entraînement du modèle repose sur deux scripts :

— `src/train.py` : script principal de l'entraînement, chargé de a) la préparation des données, b) la compilation du modèle, c) lancement de l'apprentissage et de la sauvegarde des poids,

— `src/model.py` : définition de l'architecture du réseau de neurones convolutionnel (CNN).

Cette séparation permet d'identifier clairement :

— la **structure du modèle** (architecture),

— la **configuration de l'entraînement** (hyperparamètres, entrainement, callbacks, sauvegarde).

Le script `src/train.py` représente l'élément central du pipeline d'apprentissage du projet. Il est exécuté à l'aide de la commande suivante :

```
python -m src.train
```

3.2 Définition du modèle CNN (`src/model.py`)

Le script `src/model.py` contient la définition de l'architecture du réseau de neurones convolutionnel. Le modèle est construit suivant une approche **from scratch** sans faire appel à un modèle pré-entrainé.

Le réseau prend en entrée des images RGB de dimension :

$$32 \times 32 \times 3$$

et fournit en sortie une distribution de probabilités associée aux **43 classes** du dataset GTSRB.

3.3 Architecture du modèle CNN utilisé - `src/model.py`

Le fichier `src/model.py` n'est pas exécuté directement. Il fournit une fonction `build_baseline_cnn` importée par `src/train.py`. Chaque bloc décrit ci-dessous représente une étape logique du réseau.

Bloc 1 — Import TensorFlow

```
import tensorflow as tf
```

But : importer TensorFlow/Keras pour la construction du modèle (couches, graphe, objets Keras). L'ensemble du réseau est défini à l'aide de l'API fonctionnelle de Keras, rendant l'architecture explicite. (entrées → couches → sortie).

Bloc 2 — Signature de la fonction constructeur

```
def build_baseline_cnn(input_shape=(32, 32, 3), num_classes=43) -> tf.keras.Model:
```

But : définir une fonction réutilisable permettant de construire un CNN,
- — `input_shape=(32,32,3)` : images RGB redimensionnées en 32×32.
- — `num_classes=43` : nombre de classes (panneaux) du dataset GTSRB.
- — Le type de retour `tf.keras.Model` indique que la fonction renvoie un modèle Keras complet.

Bloc 3 — Définition de l'entrée du réseau

```
inputs = tf.keras.Input(shape=input_shape)
```

But : définir le tenseur d'entrée du modèle ce qui permet de créer un point d'entrée explicite, (format attendu : hauteur × largeur × canaux), utilisé par la suite pour enchaîner les couches du CNN.

Bloc 4 — Bloc convolutionnel 1 : (Conv32 + MaxPool)

```
x = tf.keras.layers.Conv2D(32, 3, padding="same", activation="relu")(inputs)
x = tf.keras.layers.MaxPool2D()(x)
```

But : extraire des motifs visuels simples (bords, contours, textures).
- — `Conv2D(32, 3)` : 32 filtres 3×3
- — `padding="same"` : conserve la taille spatiale avant pooling
- — `ReLU` : non-linéarité efficace pour l'apprentissage
- — `MaxPool2D` : réduit la résolution (invariance locale, gain calcul/mémoire)

Bloc 5 — Bloc convolutionnel 2 : (Conv64 + MaxPool)

```
x = tf.keras.layers.Conv2D(64, 3, padding="same", activation="relu")(x)
x = tf.keras.layers.MaxPool2D()(x)
```

But : apprendre des représentations plus complexes, notamment des formes intermédiaires et des parties de symboles. On augmente le nombre de filtres à 64, la diminution de la résolution permettant d'extraire plus de canaux de caractéristiques sans explosion du coût.

Bloc 6 — Bloc convolutionnel 3 : (Conv128 + MaxPool)

```
x = tf.keras.layers.Conv2D(128, 3, padding="same", activation="relu")(x)
x = tf.keras.layers.MaxPool2D()(x)
```

But : capturer des représentations de plus haut niveau (pictogrammes, compositions, formes globales). L'augmentation à 128 filtres renforce la capacité du modèle, facilitant la discrimination de panneaux très similaires visuellement (ex. limitations de vitesse).

Bloc 7 — Passage vers un vecteur de caractéristiques

```
x = tf.keras.layers.Flatten()(x)
```

But : convertir les cartes de caractéristiques 2D (et leurs canaux) en un vecteur 1D. Ce vecteur constitue une représentation compacte de l'image, exploitable par les couches denses pour la classification.

Bloc 8 — Couches fully-connected + régularisation (Dropout)

```
x = tf.keras.layers.Dense(256, activation="relu")(x)
x = tf.keras.layers.Dropout(0.3)(x)
```

But : combiner les caractéristiques extraites pour produire une décision de classe.
- `Dense(256)` : couche de décision (capacité modérée, adaptée à un baseline).
- `Dropout(0.3)` : désactive aléatoirement 30% des neurones pendant l'entraînement, contribuant à diminuer le risque de sur-apprentissage tout en améliorant la généralisation.

Bloc 9 — Couche de sortie Softmax (43 classes)

```
outputs = tf.keras.layers.Dense(num_classes, activation="softmax")(x)
```

But : générer une distribution de probabilité sur les **43 classes**. La fonction *softmax* produit un vecteur de probabilités normalisé (somme égale à 1), la prédiction étant donnée par l'indice de la probabilité maximale (`argmax`).

Bloc 10 — Construction et retour du modèle Keras

```
return tf.keras.Model(inputs, outputs, name="baseline_cnn")
```

But : assembler le graphe complet, de l'entrée à la sortie, et retourner un objet `Model`. Le nom `baseline_cnn` facilite l'identification lors des phases de sauvegarde et de débogage.

L'architecture adoptée repose sur une structure classique et progressive composée de blocs convolutionnels suivis de couches entièrement connectées, comme résumé ici :

Description des couches :

Comme montré dans la figure 1 l'architecture du réseau se décompose en deux parties principales : une phase d'extraction de caractéristiques fondée sur des couches convolutionnelles et de sous-échantillonnage et suivie d'une phase de classification basée sur des couches connectées.

- Les couches **Conv2D** visent à extraire progressivement des caractéristiques locales, telles que les bords et les formes et les motifs, à partir des images d'entrée,
- Les couches **MaxPooling** permettent de réduire la dimension spatiale des cartes de caractéristiques, diminuant ainsi le coût computationnel et améliorant l'invariance aux petites translations,

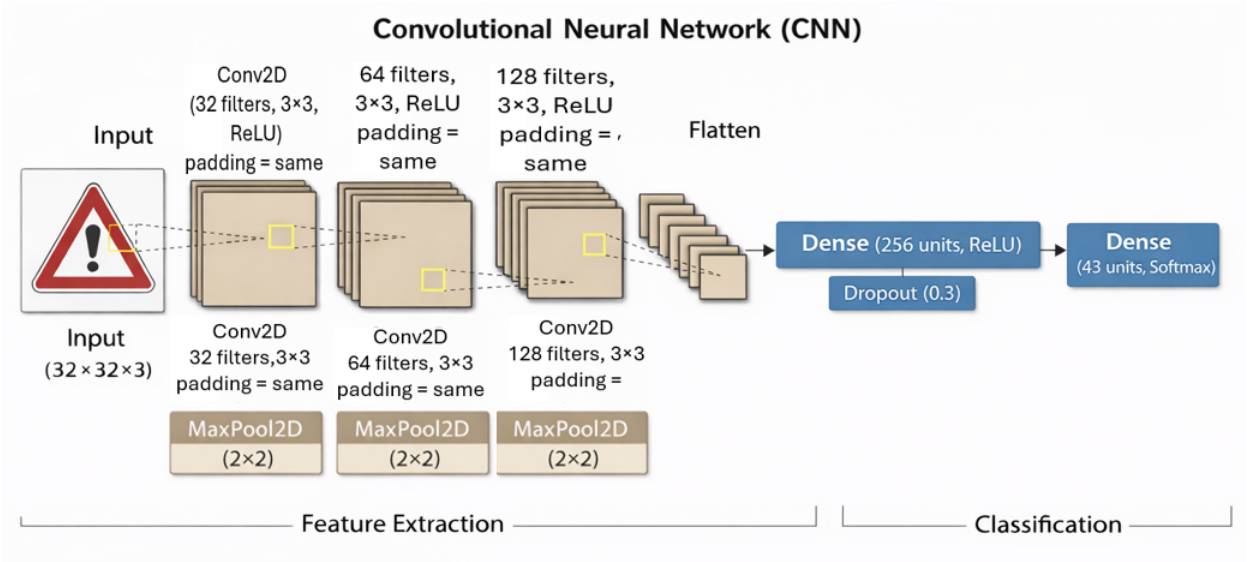

FIGURE 1 – Architecture du réseau de neurones convolutionnel (CNN) utilisé pour la classification des panneaux de signalisation. Le modèle est composé de trois blocs convolutionnels suivis de couches de classification.

— La couche **Flatten** permet de transformer les cartes de caractéristiques en un vecteur adapté aux couches denses,
— La couche **Dense** de 256 neurones sert à combiner les caractéristiques extraites en vue de la classification,
— La couche **Dropout** (taux 0.3) est employée à des fins de régularisation pour atténuer le sur-apprentissage,
— La couche finale **Dense** + **Softmax** fournit une distribution de probabilités associée aux **43 classes**.

3.4 Justification des choix architecturaux

Une architecture progressive (32 → 64 → 128 filtres) est utilisée afin d'extraire les caractéristiques de manière hiérarchique :
— les premières couches sont dédiées à la détection de motifs simples, comme les bords et les couleurs,
— les couches intermédiaires identifient des formes plus complexes,
— les couches profondes capturent des structures propres aux panneaux, telles que les symboles, chiffres et pictogrammes.

3.5 Script d'entrainement `src/train.py`

Comme le montre la figure 2 le script `src/train.py` exécute successivement les étapes suivantes :

1. chargement des ensembles d'entraînement et de validation depuis les fichiers `train_split.csv` et `val_split.csv`,
2. construction des pipelines `tf.data` correspondants,
3. instanciation du modèle CNN défini dans `src/model.py`,
4. compilation du modèle (optimiseur, fonction de coût, métriques),
5. phase d'entrainement accompagnée d'un suivi des performances sur l'ensemble de validation,

6. sauvegarde automatique du modèle offrant les meilleures performances et du modèle final.

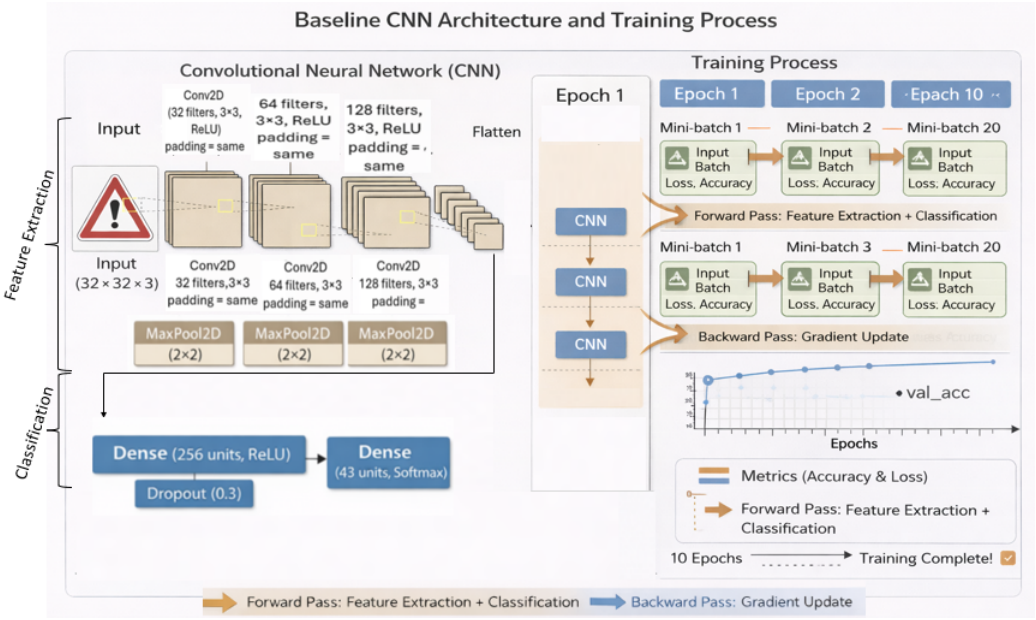

FIGURE 2 – Schéma du processus d'entraînement du CNN. Le modèle est entraîné par mini-batchs de 64 images sur 10 epochs, avec calcul de la fonction de perte et mise à jour des poids à chaque itération.

Script d'entrainement - Code expliqué - `src/train.py`

On décrit ici le script `src/train.py` bloc par bloc :

Bloc 1 — Importations

```
from pathlib import Path
import tensorflow as tf

from src.data import make_tf_dataset
from src.model import build_baseline_cnn

import json
from pathlib import Path
```

But : charger l'ensemble des dépendances requises pour l'entraînement ;

- `tensorflow` : construction/compilation/entrainement du CNN.
- `Path` : gestion robuste des chemins (Linux/WSL, etc...)
- `make_tf_dataset` : construit les pipelines `tf.data` à partir des CSV.
- `build_baseline_cnn` : définit l'architecture du modèle (implémentée dans `src/model.py`) ;
- `json` : sauvegarde de l'historique d'apprentissage pour tracer les courbes.

Bloc 2 — Fonction principale et hyperparamètres

```
def main():
    img_size = (32, 32)
```

```
batch_size = 64
num_classes = 43
epochs = 10
```

But : centraliser les paramètres.
- `img_size=(32,32)` : redimensionnement standard du jeu de données GTSRB pour l'entrainement d'un CNN léger,
- `batch_size=64` : compromis vitesse/mémoire (CPU dans ton cas),
- `num_classes=43` : nombre total de classes de panneaux GTSRB,
- `epochs=10` : nombre maximum d'époques (l'arrêt anticipé peut arrêter avant).

Bloc 3 — Chemins des splits train/validation

```
train_csv = Path("data/train_split.csv")
val_csv = Path("data/val_split.csv")
```

But : pointer vers les deux fichiers CSV décrivant :
- la liste des images d'entrainement (et leurs labels).
- la liste des images de validation (et leurs labels).

Important : la validation doit etre disjointe de l'ensemble d'entrainement afin d'évaluer la capacité de généralisation sur des images non vues durant l'apprentissage.

Bloc 4 — Création des datasets `tf.data`

```
train_ds = make_tf_dataset(train_csv, img_size=img_size,
batch_size=batch_size, shuffle=True)
val_ds = make_tf_dataset(val_csv, img_size=img_size, batch_size=batch_size,
shuffle=False)
```

But : construire deux pipelines `tf.data` optimisés.
- `shuffle=True` pour l'entrainement : évite que le modèle n'apprenne l'ordre des données et améliore la convergence,
- `shuffle=False` pour la validation : assure une évaluation stable et reproductible,

Ces jeux de données appliquent le meme prétraitement que celui utilisé ultérieurement en phase d'inférence (redimensionnement + normalisation).

Bloc 5 — Construction du modèle CNN

```
model = build_baseline_cnn(input_shape=(img_size[0], img_size[1], 3),
num_classes=num_classes)
```

But : instancier le CNN défini dans `src/model.py`.
- `input_shape=(32,32,3)` : images RGB.
- `num_classes=43` : couche *softmax* finale sur 43 classes.

La définition du modèle (`model.py`) est séparée de la phase d'entrainement (`train.py`) pour conserver un code modulaire et réutilisable.

Bloc 6 — Compilation (optimiseur, loss, métrique)

```
model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-3),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
```

But : définir comment le réseau apprend.

- **Adam** : optimiseur efficace et standard pour les CNN,
- **Learning rate** 10^{-3} : valeur initiale classique,
- **sparse_categorical_crossentropy** : adaptée lorsque les labels sont des entiers (0..42) et non des vecteurs one-hot,
- **accuracy** : évalue le taux de prédictions correctes.

Bloc 7 — Callbacks : EarlyStopping et ModelCheckpoint

```
callbacks = [
    tf.keras.callbacks.EarlyStopping(monitor="val_accuracy", patience=3,
restore_best_weights=True),
    tf.keras.callbacks.ModelCheckpoint("models/baseline_cnn.keras",
monitor="val_accuracy", save_best_only=True),
]
```

But : améliorer la robustesse de l'entraînement et automatiser la sauvegarde.

- **EarlyStopping** : surveille `val_accuracy`. En l'absence d'amélioration pendant trois époques consécutives, l'entraînement peut s'arrêter. `restore_best_weights=True` remet automatiquement les meilleurs poids observés.
- **ModelCheckpoint** : sauvegarde le meilleur modèle au format `.keras` dans `models/baseline_cnn.keras`, basé sur `val_accuracy`.

Bloc 8 — Entraînement

```
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=epochs,
    callbacks=callbacks,
)
```

But : lancer l'apprentissage.

- À chaque époque le modèle apprend sur `train_ds`,
- Puis il est évalué sur `val_ds` (produit `val_loss` et `val_accuracy`).

L'objet `history` contient l'ensemble des valeurs par époque utilisées ultérieurement pour le tracé des courbes.

Bloc 9 — Sauvegarde de l'historique (JSON)

```
    history_dict = history.history
    Path("reports").mkdir(exist_ok=True)

    with open("reports/training_history.json", "w") as f:
        json.dump(history_dict, f)

    print(" Historique d'entranement sauvegardé")
```

But : stocker les courbes d'entrainement dans un fichier `reports/training_history.json`. Cela permet :

— de générer **train/val accuracy** et **train/val loss** sans relancer l'entraînement,
— d'obtenir une trace reproductible, correspondant aux valeurs exactes par époque.

Bloc 10 — Sauvegarde du dernier modèle

```
    Path("models").mkdir(exist_ok=True)
    model.save("models/baseline_cnn_last.keras")
    print(" Saved models/baseline_cnn.keras (best) and
    models/baseline_cnn_last.keras (last)")
```

But : sauvegarder deux versions :

— `baseline_cnn.keras` = **meilleur** modèle (checkpoint, selon `val_accuracy`),
— `baseline_cnn_last.keras` = **dernier** état du modèle après la dernière époque.

Le meilleur modèle est généralement retenu dans le rapport pour les phases d'évaluation et d'inférence.

Bloc 11 — Point d'entrée du script

```
if __name__ == "__main__":
    main()
```

But : permettre l'exécution directe du script via : `python -m src.train`. Ce bloc appelle la fonction `main()` afin d'exécuter l'intégralité du pipeline.

Les principaux hyperparamètres utilisés sont :

— taille des images : 32×32,
— taille de batch : 64,
— nombre d'époques maximal : 10,
— optimiseur : Adam avec un taux d'apprentissage initial de 10^{-3},
— fonction de perte : `sparse_categorical_crossentropy`.

Un mécanisme d'arrêt anticipé (*Early Stopping*) est également utilisé dans le but de limiter le sur-apprentissage et de rétablir automatiquement les meilleurs poids mesurés sur l'ensemble de validation.

3.6 Stratégie d'entrainement

L'entrainement est effectué sur l'ensemble `train_split.csv` tandis que l'évaluation des performances est réalisée à chaque époque sur l'ensemble `val_split.csv`. Les métriques suivies sont :

— la fonction de perte (*loss*),
— l'exactitude de classification (*accuracy*).

Une stratégie d'*Early Stopping* paramétrée avec une patience de trois époques est appliquée à l'exactitude de validation afin d'interrompre l'apprentissage lorsque les performances ne s'améliorent plus, et le meilleur modèle est automatiquement sauvegardé dans le fichier :

`models/baseline_cnn.keras`

4 Résultats et analyse

Les performances du modèle et l'analyse des résultats sur le jeu de validation sont présentées ici. Remarque : les résultats sont obtenus à partir du script `src/evaluate.py` exécuté après l'entrainement.

4.1 Protocole d'évaluation

Le modèle est évalué sur le jeu de validation `val_split.csv`, séparé de l'ensemble d'entrainement et correspond à la meilleure version obtenue lors de l'apprentissage.

`models/baseline_cnn.keras`

Le script d'évaluation est exécuté via la commande suivante :

```
python -m src.evaluate
```

Les métriques évaluées sont :

— l'exactitude globale (*accuracy*),
— le rapport de classification (précision, rappel, F1-score par classe),
— la matrice de confusion,
— l'identification des confusions les plus fréquentes entre classes.

4.2 Courbes d'apprentissage (accuracy et loss)

Dans le but de visualiser la convergence du modèle, l'évolution de l'exactitude (*accuracy*) et de la perte (*loss*) est tracée sur les ensembles d'entraînement et de validation. Les courbes sont générées à partir de l'historique sauvegardé au cours de l'exécution de `model.fit()`.

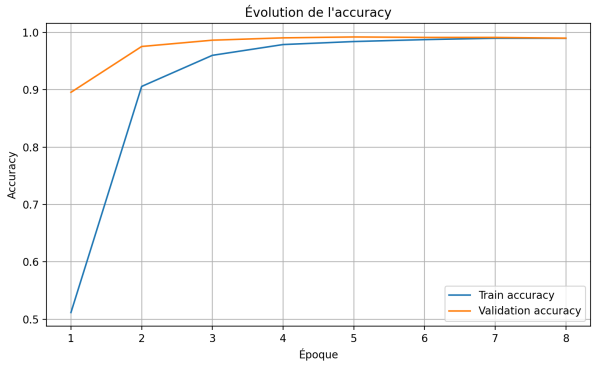

FIGURE 3 – Évolution de l'accuracy pendant l'entraînement : train vs validation.

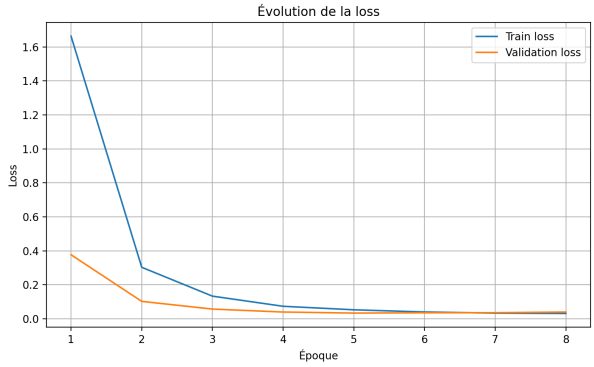

FIGURE 4 – Évolution de la loss pendant l'entraînement : train vs validation.

La figure 3 indique une convergence rapide de l'accuracy qui dépasse rapidement 90% et se stabilise ensuite proche de 99% sur les deux ensembles ; La figure 4 confirme la convergence observée par une réduction rapide de la perte, avant une phase de stabilisation. On observe que les courbes d'entrainement et de validation demeurent très proches ce qui suggère une bonne capacité de généralisation et un sur-apprentissage limité. De légères fluctuations en fin d'apprentissage (par exemple une faible remontée de la *val_loss*) sont normales et peuvent s'expliquer par la variabilité de la difficulté des mini-lots ou par la présence de quelques exemples plus ambigus dans l'ensemble de validation.

4.3 Performances globales

L'évaluation réalisée sur l'ensemble de validation donne les résultats suivants :

— Exactitude de validation : **99.46 %**

Ce résultat met en évidence une forte capacité de généralisation sur des données non vues pendant l'entrainement, malgré la variabilité importante des images, notamment en taille et luminosité, angles de vue et bruit.

4.4 Matrice de confusion

La figure 5 montre la matrice de confusion obtenue sur l'ensemble de validation.

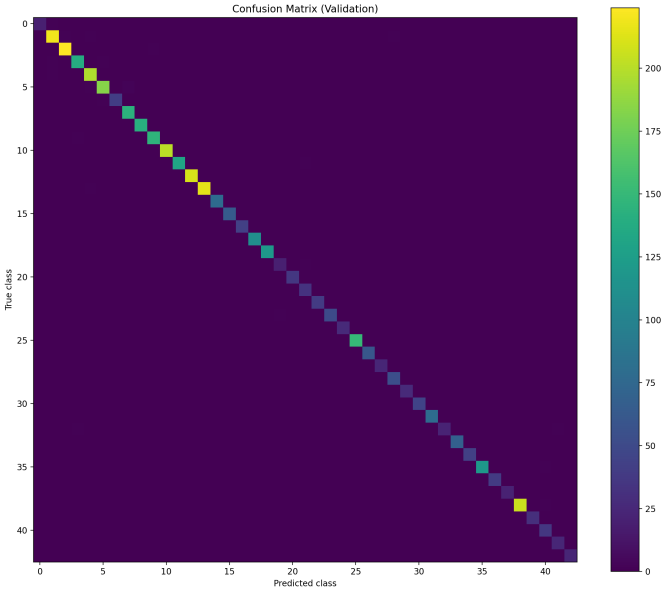

FIGURE 5 – Matrice de confusion sur l'ensemble de validation

La matrice de confusion est fortement diagonale indiquant que la grande majorité des échantillons est correctement classée. Chaque classe de panneau routier est identifiée avec une précision très élevée.

Malgré ces performances, quelques confusions subsistent. Les confusions les plus couramment observées sont listées dans le fichier :

`reports/top_confusions.csv`

Parmi celles-ci, on observe surtout :
— des confusions ponctuelles entre certaines limitations de vitesse proches sur le plan visuel, telles que 50 km/h et 80 km/h,
— des confusions entre panneaux de danger triangulaires aux contours similaires mais aux pictogrammes internes distincts,
— des erreurs ponctuelles dues à des images sombres ou partiellement occultées.

Ces erreurs sont conformes avec la nature du problème : certaines classes se distinguent par des différences très subtiles, difficiles à percevoir meme pour un observateur humain dans des conditions de bruit ou de faible éclairage.

4.5 Analyse qualitative des erreurs

L'analyse qualitative des erreurs indique que les images mal classées possèdent généralement au moins l'une des caractéristiques suivantes :
— faible contraste ou luminosité insuffisante,
— flou de mouvement,
— panneau partiellement masqué,
— résolution effective très faible après redimensionnement.

Ces observations montrent que le modèle a effectivement appris des représentations pertinentes, bien qu'il reste sensible à certaines conditions de prise de vue dégradées.

4.6 Discussion critique des résultats

Les résultats obtenus sont très élevés malgré la simplicité de l'architecture utilisée et l'absence de *transfer learning*. Le choix d'un CNN entraîné *from scratch* a été suffisant pour ce jeu de données, grâce à :
— une architecture adaptée à la taille des images,
— un pipeline de données robuste basé sur `tf.data`,
— une séparation stricte et rigoureuse entre les données d'apprentissage et de validation.

Cependant, certaines limites persistent :
— le modèle n'a pas été testé sur des images provenant de sources totalement externes au dataset GTSRB,
— aucune augmentation de données (*data augmentation*) n'a été utilisée,
— les performances pourraient être encore améliorées par le recours à des modèles pré-entraînés (ResNet, MobileNet).

4.7 Inférence qualitative sur des images inconnues

Afin de mettre en évidence le fonctionnement du modèle entraîné dans des conditions réalistes, une phase d'inférence qualitative est menée sur plusieurs images issues du dossier `data/Test`. Ces images n'ont pas été utilisées pendant l'entrainement ni lors de la validation, ce qui permet d'évaluer la capacité de généralisation du réseau.

L'inférence est réalisée à l'aide du script `src/infer_demo.py`. Celui-ci charge le modèle sauvegardé (`baseline_cnn.keras`), procède à une sélection aléatoire de 12 images à partir du jeu

de test, applique exactement le meme prétraitement que celui utilisé lors de l'entrainement (redimensionnement en 32×32, normalisation des pixels), et calcule ensuite les prédictions à l'aide de la sortie *softmax* du réseau.

Pour chaque image, la prédiction de classe ainsi que la probabilité correspondante sont affichées. Une grille récapitulative est générée puis sauvegardée automatiquement, ce qui permet de préserver la reproductibilité de la démonstration meme dans des environnements non interactifs (WSL / SSH).

FIGURE 6 – Inférence qualitative sur 12 images inconnues du jeu de test. Pour chaque panneau, la classe prédite et la probabilité associée sont indiquées.

La figure 6 montre la capacité du modèle à reconnaître correctement divers types de panneaux (limitations de vitesse, priorité, obligation, etc...), meme en présence de conditions visuelles dégradées, notamment le flou, le faible contraste et des arrière-plans complexes. Les probabilités associées aux prédictions sont généralement élevées, souvent proches de 1.0, ce qui reflète une confiance importante du réseau sur ces exemples.

Certaines images présentent cependant des probabilités légèrement inférieures (par exemple $P = 0.979$), ce qui est généralement associé à des panneaux partiellement flous ou insuffisamment centrés. Ces variations apparaissent cohérentes et montrent que le modèle ajuste sa confiance en fonction de la qualité visuelle de l'image.

Cette étape valide le bon fonctionnement de l'ensemble du pipeline (prétraitement, chargement du modèle et inférence). et fournit une démonstration qualitative répondant aux exigences du projet.

Interprétation visuelle des prédictions.

Dans le cadre de l'inférence qualitative, chacune des 12 images présentées dans la figure 6 a été examinée manuellement. Pour chaque image : une comparaison visuelle est effectuée entre le panneau routier, la classe prédite par le modèle et la probabilité associée.

Les résultats montrent que : pour les 12 exemples examinés, le modèle prédit correctement la catégorie du panneau. Par exemple :

- L'image `08040.png` est prédite par le modèle comme appartenant à la classe 2 avec une probabilité de 1.000, ce qui correspond, d'un point de vue visuel, à un panneau de limitation à 50 km/h, ce qui est vrai visuellement (cette classe contient réellement des panneaux de limitation à 50 km/h).
- L'image `04694.png` a été prédite comme appartenant à la classe 5 (probabilité = 1.000), correspondant à un panneau de limitation à 80 km/h, ce qui est cohérent d'un point de vue visuel, cette classe correspondant effectivement à des panneaux de limitation à 80 km/h.
- et de la même manière pour les 12 exemples présentés dans la démonstration

Cette vérification visuelle montre que le modèle fournit de bonnes prédictions meme sur des images jamais rencontrées lors de l'entrainement. Elle montre aussi que ces prédictions sont cohérentes et correspondent aux catégories attendues lorsqu'elles sont examinées manuellement. Ces observations confirment les bonnes performances du modèle sur des données de test réelles.

5 Conclusion

Dans ce projet, un pipeline complet de vision par ordinateur basé sur un réseau de neurones convolutionnel CNN a été implémenté pour résoudre un problème réel de classification d'images. Le processus complet depuis la préparation des données jusqu'à l'inférence sur des images inconnues a été mis en œuvre de façon rigoureuse et reproductible.

Le modèle développé *from scratch* atteint une précision de validation supérieure à 99 %, ce qui montre qu'il est capable de bien extraire les caractéristiques visuelles des panneaux routiers. L'analyse de la matrice de confusion indique que les erreurs sont rares et concernent principalement des classes visuellement proches ce qui est cohérent avec la nature du problème.

Les résultats de l'inférence qualitative confirment également la bonne capacité de généralisation du modèle, caractérisées par un haut degré de confiance sur des images jamais rencontrées auparavant, meme dans des conditions visuelles difficiles comme le flou, le faible contraste ou des arrière-plans complexes.

Ce travail démontre l'efficacité des CNN dans les taches de vision par ordinateur et montre l'importance d'un pipeline correctement structuré, avec une architecture adaptée et une évaluation rigoureuse.

Dans la perspective d'améliorer et de prolonger ce travail, plusieurs axes peuvent être considérés. Dans un premier temps, l'intégration de techniques de *data augmentation* (rotations, variations de luminosité, ajout de bruit) pourrait renforcer la robustesse du modèle face aux conditions réelles d'acquisition.

Dans un second temps, l'utilisation du *transfer learning* à partir de modèles pré-entraînés comme ResNet ou MobileNet pourrait etre envisagée afin d'accélérer l'entraînement sans perte de performance, notamment dans des environnements disposant de ressources de calcul limitées.

Enfin. il serait possible d'ajouter des analyses sur les probabilités de sortie et sur l'interprétation du modèle (cartes d'activation, Grad-CAM) afin de mieux comprendre ses décisions et de renforcer la confiance dans ses prédictions.