Institut National des Postes et Télécommunications

Rapport de Travaux Pratiques

Deep Learning : TP2

Implémentation d'un CNN sous Keras pour la classification d'images (CIFAR-10)

Réalisé par (Groupe N2) :

EL KARFI SOUFIA

IDRISSI SALMA

SELLAME SALWA

LAZAAR EL MEHDI

Sous la direction de :

E. IBN-ELHAJ

Année académique 2025–2026

1 Objectifs

— Comprendre la structure d'un CNN (convolutions, pooling, couches denses).
— Implémenter un CNN avec Keras/TensorFlow pour classifier des images (CIFAR-10).
— Étudier l'impact de l'architecture et des hyperparamètres sur les performances.

2 Jeu de données : CIFAR-10

CIFAR-10 est un jeu de données de classification d'images couleur de taille 32×32 réparties en 10 classes : *airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck*. Les images sont normalisées dans $[0, 1]$.

3 Méthode

3.1 Architecture CNN (baseline)

Le modèle implémenté suit une architecture simple et standard :
— Trois blocs convolution + max-pooling : $32 \to 64 \to 128$ filtres (noyaux 3×3, padding `same`, activation ReLU).
— Une couche `Flatten`.
— Une couche dense de 128 neurones (ReLU).
— Dropout (0,5) pour la régularisation.
— Couche de sortie de taille 10 avec `softmax`.

3.2 Fonction de perte et métrique

On utilise une classification multi-classe exclusive (une seule classe correcte parmi 10). La sortie `softmax` produit des probabilités :

$$p_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

La perte de cross-entropie catégorielle est :

$$\mathcal{L} = -\sum_i y_i \log(p_i).$$

Dans le code, l'étiquetage est géré en `sparse_categorical_crossentropy` (labels entiers), équivalente à `categorical_crossentropy` avec one-hot.

3.3 Entraînement

Paramètres typiques utilisés lors des comparaisons :
— **Batch size** : 64
— **Epochs** : 1 pour un test de bon fonctionnement ; 3 pour les comparaisons rapides.
— **Optimiseurs (énoncé)** : Adam (exemple), SGD avec momentum, RMSprop.

3.4 Data augmentation

Une augmentation simple est testée (rotation, translations, flip horizontal, zoom) via extttImageDataGenerator. Cela augmente la robustesse mais ralentit l'entraînement car chaque batch est transformé à la volée.

4 Résultats expérimentaux

4.1 Validation de l'environnement (run court)

Un run de 1 epoch (Adam sans augmentation) valide l'exécution complète : entraînement, évaluation, courbes, matrice de confusion et exemples d'erreurs.

4.2 Comparaison Adam vs SGD, avec/sans augmentation (3 epochs)

Les expériences suivantes ont été exécutées (3 epochs) et archivées dans le dossier `runs/`.

Expérience	**Test accuracy**	**Remarque**
Adam (sans augmentation)	0.6341	Convergence rapide
Adam (avec augmentation)	0.6111	Augmentation plus lente au début
SGD + momentum (sans augmentation)	0.5881	Plus lent qu'Adam en 3 epochs
SGD + momentum (avec augmentation)	0.5107	Très lent au début (3 epochs)
RMSprop (sans augmentation)	0.6650	Bon score en 3 epochs
RMSprop (avec augmentation)	0.6175	Convergence plus lente avec augmentation

TABLE 1 – Comparaison rapide (3 epochs). L'augmentation peut nécessiter plus d'epochs pour montrer son bénéfice.

4.3 Courbes d'apprentissage (accuracy/loss)

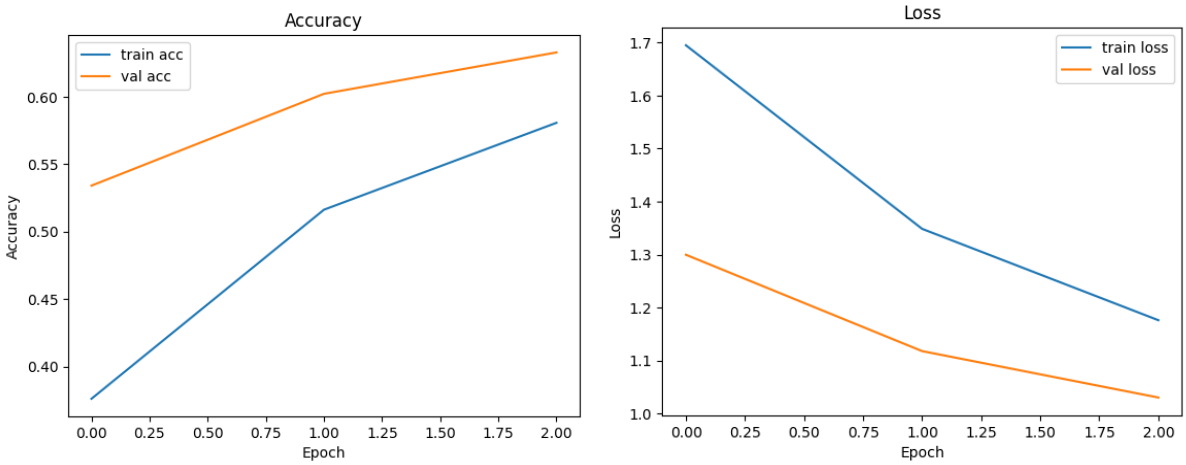

FIGURE 1 – Adam sans augmentation : accuracy et loss (train/validation).

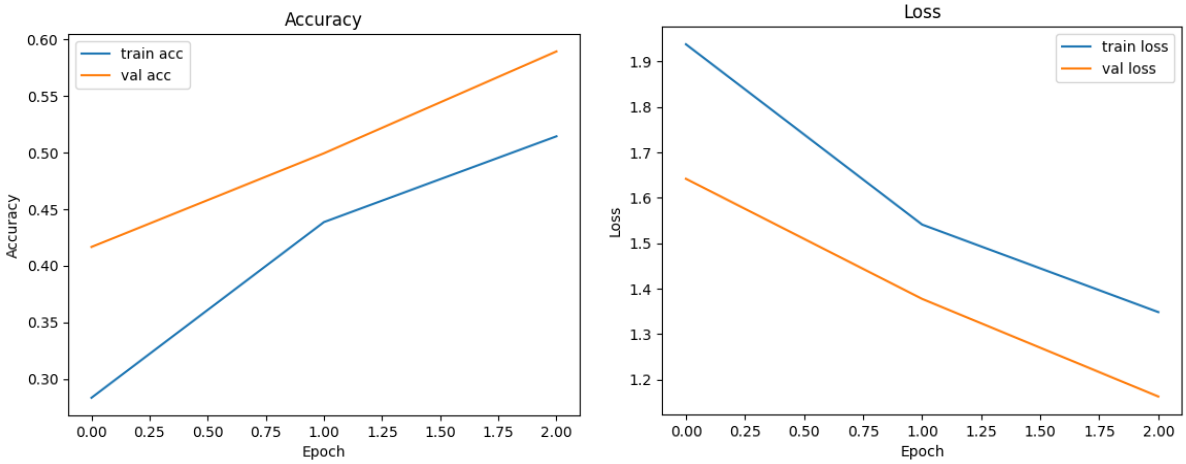

FIGURE 2 – SGD + momentum sans augmentation : accuracy et loss (train/validation).

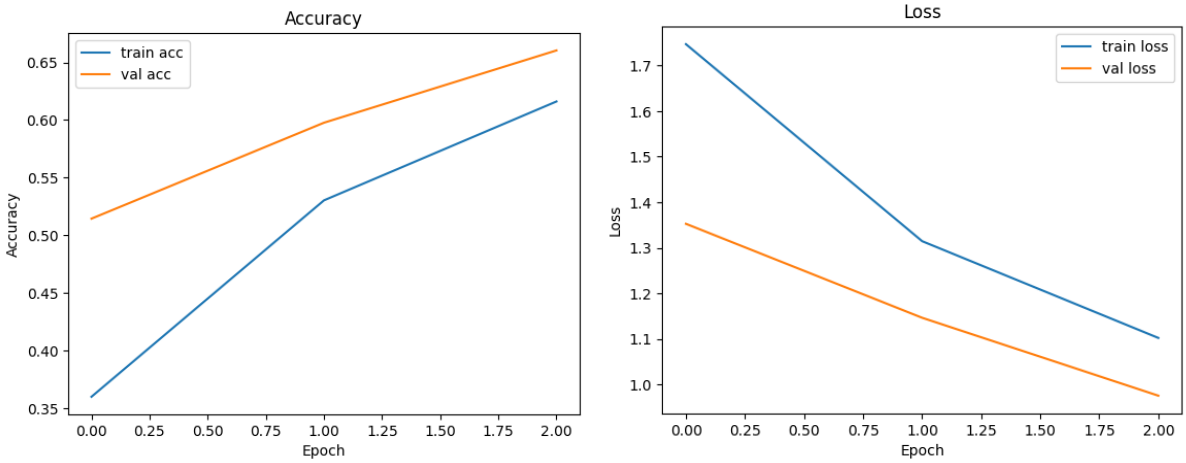

FIGURE 3 – RMSprop sans augmentation : accuracy et loss (train/validation).

4.4 Matrice de confusion et erreurs

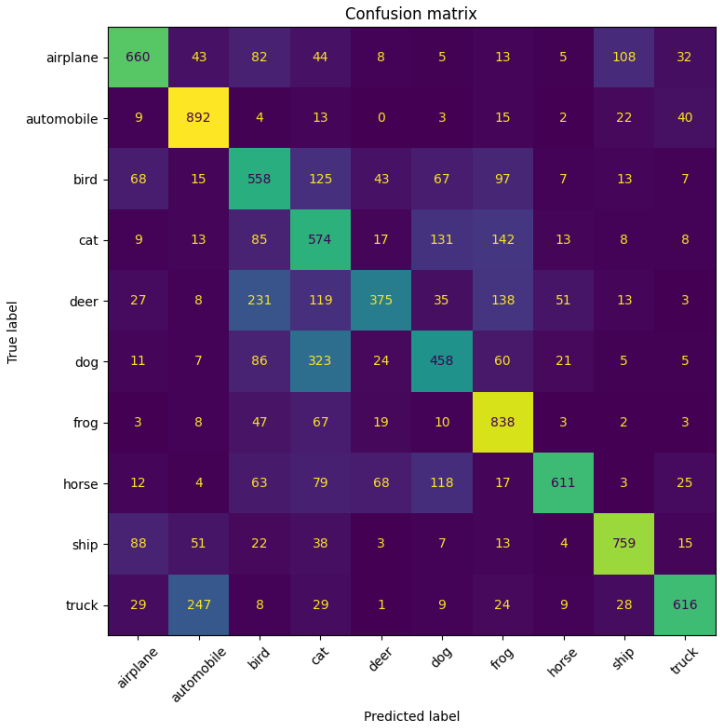

FIGURE 4 – Matrice de confusion (Adam sans augmentation).

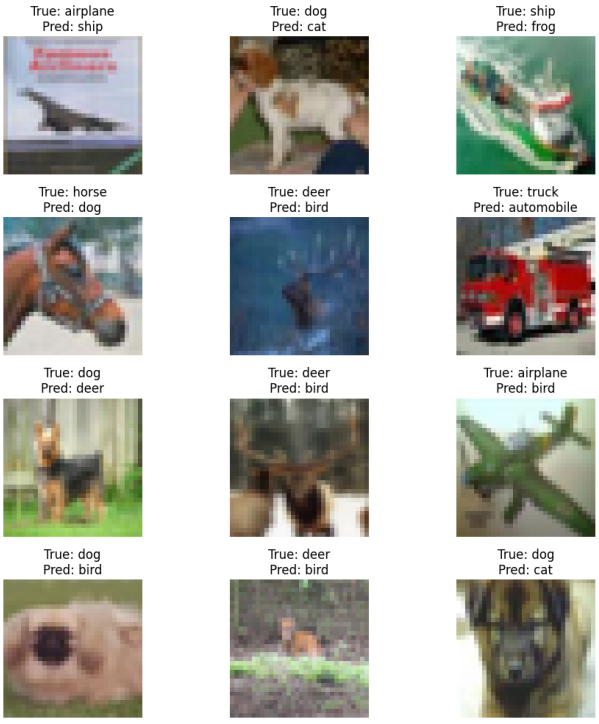

FIGURE 5 – Exemples d'images mal classées (Adam sans augmentation).

5 Questions de compréhension

Q1 — Rôle de `Conv2D` et `MaxPooling2D` ; pourquoi pas du dense dès le départ ?

— `Conv2D` applique des **filtres (noyaux)** qui glissent sur l'image pour détecter des **motifs locaux** (bords, textures, parties d'objets). Ses avantages clés :
  — **Partage des poids** : un même filtre est réutilisé sur toute l'image.
  — **Connectivité locale** : chaque neurone « voit » une petite zone (champ récepteur).
  — **Moins de paramètres** qu'une couche dense sur pixels bruts.
  — **Invariance approximative aux translations** (si un motif se déplace, il reste détectable).

— `MaxPooling2D` fait un **sous-échantillonnage** (ex : max sur des blocs 2×2) :
  — réduit la taille des cartes de caractéristiques $\rightarrow$ **moins de calcul**
  — rend la représentation plus robuste à de petites variations locales.

— **Pourquoi les convolutions avant le dense ?**

  Une couche dense sur une image brute explose en paramètres. Exemple CIFAR-10 : $32 \times 32 \times 3 = 3072$ entrées.
  — Dense(256) $\rightarrow 3072 \times 256 \approx 786\,000$ poids (+ biais)
  — À l'inverse, Conv2D(32, 3×3) $\rightarrow 3 \times 3 \times 3 \times 32 = 864$ poids (+ biais)

Donc la convolution est beaucoup plus efficace et mieux adaptée à la structure spatiale des images.

Q2 — Pourquoi `softmax` en sortie et `categorical_crossentropy` ?

— Le problème est une **classification multi-classe** (10 classes CIFAR-10), **une seule classe vraie** par image.

— **Softmax** transforme les logits $(z_1, \ldots, z_K)$ en probabilités :

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

On obtient une distribution (somme = 1), interprétable comme « probabilité par classe ».

— **Cross-entropy** mesure l'écart entre la distribution vraie y (one-hot) et la prédite p :

$$\mathcal{L} = -\sum_{i=1}^{K} y_i \log(p_i)$$

— En pratique, si vos labels sont **entiers** (0..9), on utilise souvent `sparse_categorical_crossentropy`. Si vos labels sont **one-hot**, on utilise `categorical_crossentropy`.

Q3 — Comment détecter sur/sous-apprentissage via `accuracy` et `loss` ?

- **Sur-apprentissage (overfitting)** typique :
  - `train_accuracy` monte fortement
  - `val_accuracy` stagne ou baisse
  - `train_loss` baisse alors que `val_loss` remonte

  → le modèle mémorise le train et généralise mal.
- **Sous-apprentissage (underfitting)** typique :
  - `train_accuracy` reste faible
  - `val_accuracy` reste faible
  - les pertes restent èlevées

  → le modèle est trop simple, ou pas assez entraîné, ou LR inadapté.
- Actions classiques :
  - overfitting → data augmentation, dropout/L2, réduire capacité, early stopping
  - underfitting → augmenter capacité (filtres/profondeur), plus d'époques, meilleur optimiseur/hyperparamètres

6 Questions d'approfondissement

Q4 – Architecture

Code Python utilisé (expériences Q4)

```python
import os

from cifar10_cnn import (
  build_cnn,
  evaluate,
  load_data,
  make_optimizer,
  plot_history,
  train,
)


def run_cnn(*, out_dir: str, filters=(32, 64, 128), kernel_size=3) -> None:
  os.makedirs(out_dir, exist_ok=True)
  (x_train, y_train), (x_test, y_test) = load_data()

  model = build_cnn(
    input_shape=x_train.shape[1:],
    filters=filters,
    kernel_size=kernel_size,
    dense_units=128,
    dropout=0.5,
  )
  opt = make_optimizer("adam", learning_rate=1e-3, momentum=0.9)
  model.compile(optimizer=opt, loss="sparse_categorical_crossentropy", metrics=["accuracy"])

  history = train(model, x_train, y_train, batch_size=64, epochs=1, data_aug=False)
  plot_history(history, out_dir)
  test_loss, test_acc = evaluate(model, x_test, y_test, show_confusion=False, misclassified=0, out_dir=out_dir)

```

```
  with open(os.path.join(out_dir, "metrics.txt"), "w", encoding="utf-8")
      as f:
    f.write(f"test_accuracy={test_acc:.6f}\n")
    f.write(f"test_loss={test_loss:.6f}\n")


run_cnn(out_dir="runs/q_baseline_cnn_1ep", filters=(32, 64, 128),
   kernel_size=3)
run_cnn(out_dir="runs/q4_filters_16_32_64_1ep", filters=(16, 32, 64),
   kernel_size=3)
run_cnn(out_dir="runs/q4_kernel5_1ep", filters=(32, 64, 128),
   kernel_size=5)
```

Résultats (métriques enregistrées)

— Baseline (filtres 32/64/128, noyau 3×3) :

```
model=cnn
optimizer=adam
learning_rate=0.001
momentum=0.9
batch_size=64
epochs=1
data_aug=False
filters=[32, 64, 128]
kernel_size=3
dense_units=128
dropout=0.5
test_accuracy=0.462100
test_loss=1.431880
```

— Réseau plus petit (filtres 16/32/64) :

```
model=cnn
optimizer=adam
learning_rate=0.001
momentum=0.9
batch_size=64
epochs=1
data_aug=False
filters=[16, 32, 64]
kernel_size=3
dense_units=128
dropout=0.5
test_accuracy=0.498900
test_loss=1.399446
```

— Noyaux 5×5 :

```
model=cnn
optimizer=adam
learning_rate=0.001
momentum=0.9
batch_size=64
epochs=1
data_aug=False
filters=[32, 64, 128]
kernel_size=5
dense_units=128
dropout=0.5
test_accuracy=0.532700
test_loss=1.308588
```

— **Modifier les filtres / profondeur** : réduire (16/32/64) diminue le temps d'entraînement et la capacité du modèle ; l'accuracy peut baisser si la capacité devient insuffisante. Augmenter la profondeur peut améliorer la performance mais augmente temps et risque d'overfitting.

— **Noyaux** 5×5 **au lieu de** 3×3 : un noyau plus grand voit un champ réceptif plus large mais coûte plus cher. Empiler plusieurs 3×3 donne souvent un bon compromis (non-linéarités supplémentaires, moins de paramètres qu'un gros noyau à largeur comparable).

Q5 – Régularisation et data augmentation

Code Python utilisé (expériences Q5)

```python
import os

from cifar10_cnn import build_cnn, evaluate, load_data, make_optimizer, plot_history, train


def run_cnn(*, out_dir: str, dropout: float, data_aug: bool) -> None:
  os.makedirs(out_dir, exist_ok=True)
  (x_train, y_train), (x_test, y_test) = load_data()

  model = build_cnn(
    input_shape=x_train.shape[1:],
    filters=(32, 64, 128),
    kernel_size=3,
    dense_units=128,
    dropout=dropout,
  )
  opt = make_optimizer("adam", learning_rate=1e-3, momentum=0.9)
  model.compile(optimizer=opt, loss="sparse_categorical_crossentropy", metrics=["accuracy"])

  history = train(model, x_train, y_train, batch_size=64, epochs=1, data_aug=data_aug)
  plot_history(history, out_dir)
  test_loss, test_acc = evaluate(model, x_test, y_test, show_confusion=False, misclassified=0, out_dir=out_dir)

  with open(os.path.join(out_dir, "metrics.txt"), "w", encoding="utf-8") as f:
    f.write(f"test_accuracy={test_acc:.6f}\n")
    f.write(f"test_loss={test_loss:.6f}\n")


run_cnn(out_dir="runs/q5_dropout0_1ep", dropout=0.0, data_aug=False)
run_cnn(out_dir="runs/q5_dataaug_1ep", dropout=0.5, data_aug=True)
```

Résultat (métriques enregistrées)

— Sans Dropout :

```
model=cnn
optimizer=adam
learning_rate=0.001
momentum=0.9
batch_size=64
epochs=1
data_aug=False
filters=[32, 64, 128]
kernel_size=3
dense_units=128
dropout=0.0
test_accuracy=0.577500
test_loss=1.204666
```

— Avec data augmentation :

```
model=cnn
optimizer=adam
learning_rate=0.001
momentum=0.9
batch_size=64
epochs=1
data_aug=True
filters=[32, 64, 128]
kernel_size=3
dense_units=128
dropout=0.5
test_accuracy=0.494600
test_loss=1.389557
```

— **Supprimer le Dropout** : l'accuracy train monte souvent plus vite, mais la validation peut plafonner et la `val_loss` peut remonter (surapprentissage).
— **Ajouter la data augmentation** : améliore généralement la robustesse et la généralisation, mais ralentit l'apprentissage et peut nécessiter plus d'epochs pour battre le modèle sans augmentation. Dans nos runs (3 epochs), les modèles augmentés sont encore en phase de convergence.

Q6 – Hyperparamètres

Code Python utilisé (expériences Q6)

```python
import os

from cifar10_cnn import build_cnn, evaluate, load_data, make_optimizer, plot_history, train


def run_cnn(*, out_dir: str, batch_size: int) -> None:
  os.makedirs(out_dir, exist_ok=True)
  (x_train, y_train), (x_test, y_test) = load_data()

  model = build_cnn(
    input_shape=x_train.shape[1:],
    filters=(32, 64, 128),
    kernel_size=3,
    dense_units=128,
    dropout=0.5,
  )
  opt = make_optimizer("adam", learning_rate=1e-3, momentum=0.9)
  model.compile(optimizer=opt, loss="sparse_categorical_crossentropy", metrics=["accuracy"])

  history = train(model, x_train, y_train, batch_size=batch_size, epochs=1, data_aug=False)
  plot_history(history, out_dir)
  test_loss, test_acc = evaluate(model, x_test, y_test, show_confusion=False, misclassified=0, out_dir=out_dir)

  with open(os.path.join(out_dir, "metrics.txt"), "w", encoding="utf-8") as f:
    f.write(f"test_accuracy={test_acc:.6f}\n")
    f.write(f"test_loss={test_loss:.6f}\n")


run_cnn(out_dir="runs/q6_batch32_1ep", batch_size=32)
run_cnn(out_dir="runs/q6_batch128_1ep", batch_size=128)
```

Résultats (métriques enregistrées)

— Batch size 32 :

```
model=cnn
optimizer=adam
learning_rate=0.001
momentum=0.9
batch_size=32
epochs=1
data_aug=False
filters=[32, 64, 128]
kernel_size=3
dense_units=128
dropout=0.5
test_accuracy=0.522700
test_loss=1.333299
```

— Batch size 128 :

```
model=cnn
optimizer=adam
learning_rate=0.001
momentum=0.9
batch_size=128
epochs=1
data_aug=False
filters=[32, 64, 128]
kernel_size=3
dense_units=128
dropout=0.5
test_accuracy=0.470700
test_loss=1.440253
```

— **Batch size** : plus petit (32) peut généraliser mieux mais est plus lent ; plus grand (128) accélère mais peut nécessiter ajustement du learning rate.
— **Optimiseurs (énoncé)** : Adam converge rapidement sans trop de réglages ; SGD+momentum peut être plus lent au début mais efficace avec un bon réglage de learning rate ; RMSprop est souvent stable et converge bien sur des tâches de vision.

Q7 – Analyse des erreurs

Code Python utilisé (expérience Q7)

```python
import os

from cifar10_cnn import build_cnn, evaluate, load_data, make_optimizer, plot_history, train


out_dir = "runs/q7_error_analysis_1ep"
os.makedirs(out_dir, exist_ok=True)

(x_train, y_train), (x_test, y_test) = load_data()
model = build_cnn(
  input_shape=x_train.shape[1:],
  filters=(32, 64, 128),
  kernel_size=3,
  dense_units=128,
  dropout=0.5,
)
opt = make_optimizer("adam", learning_rate=1e-3, momentum=0.9)
model.compile(optimizer=opt, loss="sparse_categorical_crossentropy", metrics=["accuracy"])

history = train(model, x_train, y_train, batch_size=64, epochs=1, data_aug=False)
plot_history(history, out_dir)

test_loss, test_acc = evaluate(
  model,
  x_test,
  y_test,
  show_confusion=True,
  misclassified=12,
  out_dir=out_dir,
)

with open(os.path.join(out_dir, "metrics.txt"), "w", encoding="utf-8") as f:
  f.write(f"test_accuracy={test_acc:.6f}\n")
  f.write(f"test_loss={test_loss:.6f}\n")
```

Résultats (métriques + visualisations)

```
model=cnn
optimizer=adam
learning_rate=0.001
momentum=0.9
batch_size=64
epochs=1
data_aug=False
filters=[32, 64, 128]
kernel_size=3
dense_units=128
dropout=0.5
test_accuracy=0.546400
test_loss=1.270216
```

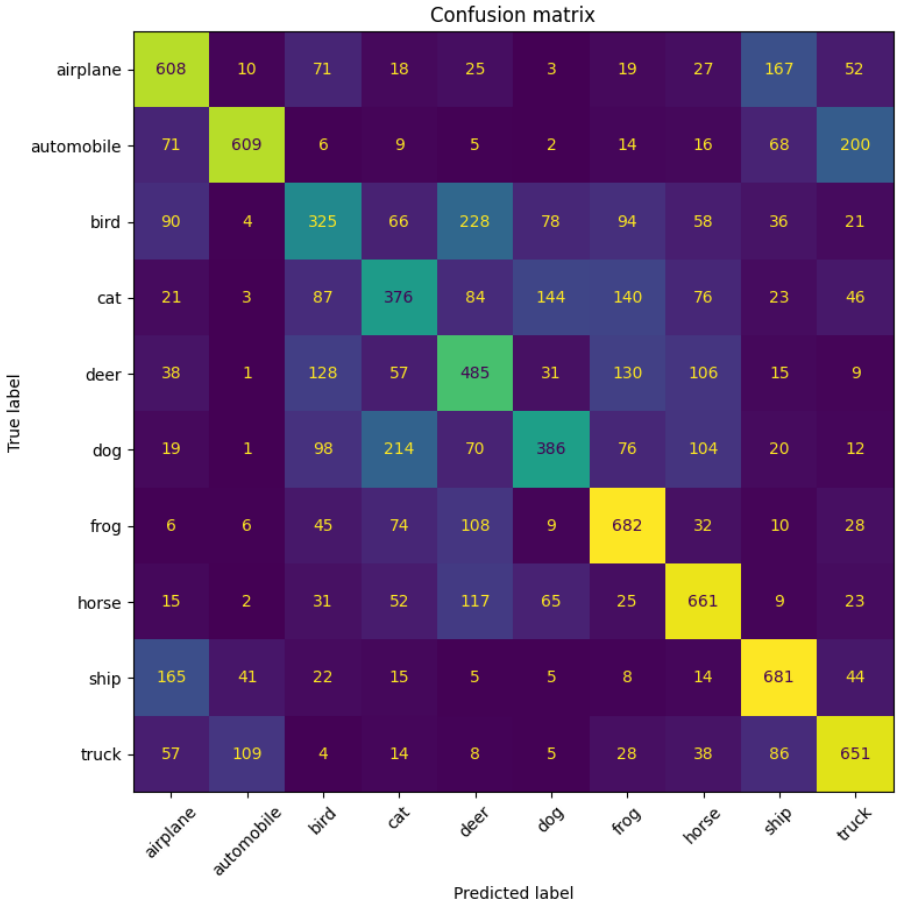

FIGURE 6 – Matrice de confusion (run Q7).

La matrice de confusion met en évidence les confusions entre classes visuellement proches (ex. *cat* vs *dog*, *airplane* vs *bird*). L'affichage des images mal classées permet d'émettre des hypothèses : faible résolution, flou, arrière-plan dominant, pose inhabituelle, similarité de texture/couleur.

Q8 – Extension (dataset personnel) et comparaison avec MLP aplati

Code Python utilisé (MLP aplati, expérience Q8)

```python
import os

from cifar10_cnn import build_mlp, evaluate, load_data, make_optimizer, plot_history, train


out_dir = "runs/q8_mlp_1ep"
os.makedirs(out_dir, exist_ok=True)

(x_train, y_train), (x_test, y_test) = load_data()
model = build_mlp(
  input_shape=x_train.shape[1:],
  dense_units=512,
  dropout=0.5,
)
opt = make_optimizer("adam", learning_rate=1e-3, momentum=0.9)
model.compile(optimizer=opt, loss="sparse_categorical_crossentropy", metrics=["accuracy"])

history = train(model, x_train, y_train, batch_size=64, epochs=1, data_aug=False)
plot_history(history, out_dir)

test_loss, test_acc = evaluate(model, x_test, y_test, show_confusion=False, misclassified=0, out_dir=out_dir)
with open(os.path.join(out_dir, "metrics.txt"), "w", encoding="utf-8") as f:
  f.write(f"test_accuracy={test_acc:.6f}\n")
  f.write(f"test_loss={test_loss:.6f}\n")
```

Résultat (métriques enregistrées)

```
model=mlp
optimizer=adam
learning_rate=0.001
momentum=0.9
batch_size=64
epochs=1
data_aug=False
filters=[32, 64, 128]
kernel_size=3
dense_units=512
dropout=0.5
test_accuracy=0.304500
test_loss=1.914783
```

Pour un dataset personnel (2–3 classes), il faut organiser les dossiers (train/val/test), veiller à l'équilibrage des classes, et utiliser davantage de régularisation/augmentation car le surapprentissage arrive plus vite. Comparer avec un MLP *aplati* illustre l'intérêt des convolutions : le MLP perd l'induction de voisinage et nécessite beaucoup de paramètres, ce qui nuit à la généralisation. Le CNN exploite la structure spatiale et partage les poids.

A Code source

```python
"""CNN experiments on CIFAR-10 with Keras.

Features
- Baseline CNN with adjustable depth, filters, and kernel size.
- Optional data augmentation.
- Hyperparameter toggles for optimizer, batch size, epochs, dropout.
- Confusion matrix and misclassified samples reporting.

Usage examples
--------------
# Baseline quick run (few epochs for a fast check)
python cifar10_cnn.py --epochs 5

# Smaller network and SGD optimizer
python cifar10_cnn.py --filters 16 32 64 --optimizer sgd --learning-rate 0.01 --momentum 0.9

# Use 5x5 kernels and disable dropout
python cifar10_cnn.py --kernel-size 5 --dropout 0

# Enable data augmentation
python cifar10_cnn.py --data-aug --epochs 15

# Show misclassified images (limits to 12 for readability)
python cifar10_cnn.py --show-misclassified 12
"""
from __future__ import annotations

import argparse
import os
from typing import Iterable, Tuple

import matplotlib.pyplot as plt
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay


def load_data() -> tuple[Tuple[np.ndarray, np.ndarray], Tuple[np.ndarray, np.ndarray]]:
    (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
    x_train = x_train.astype("float32") / 255.0
    x_test = x_test.astype("float32") / 255.0
    y_train = y_train.flatten()
    y_test = y_test.flatten()
    return (x_train, y_train), (x_test, y_test)


def build_cnn(
    input_shape: tuple[int, int, int],
    filters: Iterable[int] = (32, 64, 128),
    kernel_size: int = 3,
    dense_units: int = 128,
    dropout: float = 0.5,
```

```
) -> keras.Model:
    model = keras.Sequential()
    for i, f in enumerate(filters):
        if i == 0:
            model.add(
                layers.Conv2D(
                    f,
                    kernel_size,
                    padding="same",
                    activation="relu",
                    input_shape=input_shape,
                )
            )
        else:
            model.add(layers.Conv2D(f, kernel_size, padding="same",
                activation="relu"))
        model.add(layers.MaxPooling2D(pool_size=(2, 2)))

    model.add(layers.Flatten())
    model.add(layers.Dense(dense_units, activation="relu"))
    if dropout > 0:
        model.add(layers.Dropout(dropout))
    model.add(layers.Dense(10, activation="softmax"))
    return model


def build_mlp(
    input_shape: tuple[int, int, int],
    dense_units: int = 512,
    dropout: float = 0.5,
) -> keras.Model:
    model = keras.Sequential(
        [
            layers.Input(shape=input_shape),
            layers.Flatten(),
            layers.Dense(dense_units, activation="relu"),
        ]
    )
    if dropout > 0:
        model.add(layers.Dropout(dropout))
    model.add(layers.Dense(10, activation="softmax"))
    return model


def make_optimizer(name: str, learning_rate: float, momentum: float) ->
    keras.optimizers.Optimizer:
    name = name.lower()
    if name == "adam":
        return keras.optimizers.Adam(learning_rate=learning_rate)
    if name == "sgd":
        return keras.optimizers.SGD(learning_rate=learning_rate,
            momentum=momentum)
    if name == "rmsprop":
        return keras.optimizers.RMSprop(learning_rate=learning_rate)
    raise ValueError(f"Unsupported optimizer: {name}")


def train_val_split(
```

```
    x: np.ndarray,
    y: np.ndarray,
    val_split: float,
    seed: int = 42,
) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    if not (0.0 < val_split < 1.0):
        raise ValueError("val_split must be in (0, 1)")
    n = len(x)
    rng = np.random.default_rng(seed)
    indices = np.arange(n)
    rng.shuffle(indices)
    split = int(np.floor(n * (1.0 - val_split)))
    train_idx = indices[:split]
    val_idx = indices[split:]
    return x[train_idx], y[train_idx], x[val_idx], y[val_idx]


def train(
    model: keras.Model,
    x_train: np.ndarray,
    y_train: np.ndarray,
    batch_size: int,
    epochs: int,
    data_aug: bool,
) -> keras.callbacks.History:
    x_tr, y_tr, x_val, y_val = train_val_split(x_train, y_train,
       val_split=0.2, seed=42)

    if data_aug:
        datagen = keras.preprocessing.image.ImageDataGenerator(
            rotation_range=15,
            width_shift_range=0.1,
            height_shift_range=0.1,
            horizontal_flip=True,
            zoom_range=0.1,
        )
        datagen.fit(x_tr)
        steps = int(np.ceil(len(x_tr) / batch_size))
        history = model.fit(
            datagen.flow(x_tr, y_tr, batch_size=batch_size),
            epochs=epochs,
            validation_data=(x_val, y_val),
            steps_per_epoch=steps,
            verbose=1,
        )
    else:
        history = model.fit(
            x_tr,
            y_tr,
            batch_size=batch_size,
            epochs=epochs,
            validation_data=(x_val, y_val),
            verbose=1,
        )
    return history


```

```python
def plot_history(history: keras.callbacks.History, out_dir: str | None = None) -> None:
    acc = history.history.get("accuracy", [])
    val_acc = history.history.get("val_accuracy", [])
    loss = history.history.get("loss", [])
    val_loss = history.history.get("val_loss", [])

    plt.figure()
    plt.plot(acc, label="train acc")
    plt.plot(val_acc, label="val acc")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.title("Accuracy")
    if out_dir:
        plt.savefig(os.path.join(out_dir, "accuracy.png"), bbox_inches="tight")
    else:
        plt.show()

    plt.figure()
    plt.plot(loss, label="train loss")
    plt.plot(val_loss, label="val loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.title("Loss")
    if out_dir:
        plt.savefig(os.path.join(out_dir, "loss.png"), bbox_inches="tight")
    else:
        plt.show()


CLASS_NAMES = [
    "airplane",
    "automobile",
    "bird",
    "cat",
    "deer",
    "dog",
    "frog",
    "horse",
    "ship",
    "truck",
]


def evaluate(
    model: keras.Model,
    x_test: np.ndarray,
    y_test: np.ndarray,
    show_confusion: bool,
    misclassified: int,
    out_dir: str | None = None,
) -> tuple[float, float]:
    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
```

```
    print(f"Test accuracy = {test_acc:.4f}, test loss = {test_loss:.4f}")

    y_pred = model.predict(x_test, verbose=0)
    pred_labels = np.argmax(y_pred, axis=1)

    if show_confusion:
        cm = confusion_matrix(y_test, pred_labels)
        disp = ConfusionMatrixDisplay(cm, display_labels=CLASS_NAMES)
        fig, ax = plt.subplots(figsize=(8, 8))
        disp.plot(ax=ax, xticks_rotation=45, colorbar=False)
        plt.title("Confusion matrix")
        plt.tight_layout()
        if out_dir:
            plt.savefig(os.path.join(out_dir, "confusion_matrix.png"), bbox_inches="tight")
        else:
            plt.show()

    if misclassified > 0:
        incorrect_indices = np.where(pred_labels != y_test)[0]
        n_show = min(misclassified, len(incorrect_indices))
        if n_show == 0:
            print("No misclassified samples to show.")
            return
        plt.figure(figsize=(10, 10))
        for i, idx in enumerate(incorrect_indices[:n_show]):
            plt.subplot(int(np.ceil(n_show / 3)), 3, i + 1)
            plt.imshow(x_test[idx])
            plt.axis("off")
            plt.title(
                f"True: {CLASS_NAMES[y_test[idx]]}\nPred: {CLASS_NAMES[pred_labels[idx]]}"
            )
        plt.tight_layout()
        if out_dir:
            plt.savefig(os.path.join(out_dir, "misclassified.png"), bbox_inches="tight")
        else:
            plt.show()

    return test_loss, test_acc


def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(description="Train a CNN on CIFAR-10 with Keras")
    parser.add_argument(
        "--model",
        type=str,
        default="cnn",
        choices=["cnn", "mlp"],
        help="Model type: cnn (default) or mlp (flattened baseline)",
    )
    parser.add_argument("--filters", type=int, nargs="+", default=[32, 64, 128])
    parser.add_argument("--kernel-size", type=int, default=3)
    parser.add_argument("--dense-units", type=int, default=128)
```

```
    parser.add_argument("--dropout", type=float, default=0.5)
    parser.add_argument("--optimizer", type=str, default="adam", choices
        =["adam", "sgd", "rmsprop"])
    parser.add_argument("--learning-rate", type=float, default=1e-3)
    parser.add_argument("--momentum", type=float, default=0.9)
    parser.add_argument("--batch-size", type=int, default=64)
    parser.add_argument("--epochs", type=int, default=10)
    parser.add_argument("--data-aug", action="store_true")
    parser.add_argument("--show-confusion", action="store_true")
    parser.add_argument("--show-misclassified", type=int, default=0,
        metavar="N")
    parser.add_argument("--save-plots", type=str, default=None, help="
        Directory to save plots instead of showing")
    return parser.parse_args()


def main() -> None:
    args = parse_args()
    (x_train, y_train), (x_test, y_test) = load_data()

    if args.save_plots:
        os.makedirs(args.save_plots, exist_ok=True)

    if args.model == "mlp":
        model = build_mlp(
            input_shape=x_train.shape[1:],
            dense_units=args.dense_units,
            dropout=args.dropout,
        )
    else:
        model = build_cnn(
            input_shape=x_train.shape[1:],
            filters=args.filters,
            kernel_size=args.kernel_size,
            dense_units=args.dense_units,
            dropout=args.dropout,
        )

    opt = make_optimizer(args.optimizer, args.learning_rate, args.
        momentum)
    model.compile(optimizer=opt, loss="sparse_categorical_crossentropy",
        metrics=["accuracy"])
    model.summary()

    history = train(
        model,
        x_train,
        y_train,
        batch_size=args.batch_size,
        epochs=args.epochs,
        data_aug=args.data_aug,
    )

    plot_history(history, args.save_plots)
    test_loss, test_acc = evaluate(
        model,
        x_test,
        y_test,
```

```
        show_confusion=args.show_confusion,
        misclassified=args.show_misclassified,
        out_dir=args.save_plots,
    )

    if args.save_plots:
        with open(os.path.join(args.save_plots, "metrics.txt"), "w",
            encoding="utf-8") as f:
            f.write(f"model={args.model}\n")
            f.write(f"optimizer={args.optimizer}\n")
            f.write(f"learning_rate={args.learning_rate}\n")
            f.write(f"momentum={args.momentum}\n")
            f.write(f"batch_size={args.batch_size}\n")
            f.write(f"epochs={args.epochs}\n")
            f.write(f"data_aug={args.data_aug}\n")
            f.write(f"filters={args.filters}\n")
            f.write(f"kernel_size={args.kernel_size}\n")
            f.write(f"dense_units={args.dense_units}\n")
            f.write(f"dropout={args.dropout}\n")
            f.write(f"test_accuracy={test_acc:.6f}\n")
            f.write(f"test_loss={test_loss:.6f}\n")


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