

ÉCOLE CENTRALE CASABLANCA



---

## Mini-projet Docker - DEEP LEARNING

---

Réalisé par :

LARAISSÉ Hamza  
KERMOSS Manal

Février 2026

## Résumé

Ce projet présente la conception et le déploiement d'un système complet de classification d'images basé sur le dataset CIFAR-10. Un réseau de neurones convolutionnel (CNN) a été développé afin de classifier des images RGB de dimension  $32 \times 32$  pixels en dix catégories distinctes.

Le modèle, entraîné sur processeur (CPU), atteint une accuracy de validation de 82.28% avec un mécanisme d'Early Stopping déclenché à l'époque 67. Plusieurs techniques de régularisation, notamment le Dropout, la Batch Normalization, la Data Augmentation et l'ajustement dynamique du taux d'apprentissage, ont été mises en œuvre afin d'améliorer la capacité de généralisation.

Au-delà de la performance du modèle, ce projet s'inscrit dans une démarche MLOps complète. L'entraînement et le déploiement ont été conteneurisés via Docker, avec une architecture multi-services orchestrée par Docker Compose. Le modèle est exposé sous forme d'API REST à l'aide de Flask et intégré à une interface web interactive permettant la classification d'images en temps réel.

Ce travail illustre ainsi l'intégration d'un pipeline data-driven complet, depuis l'entraînement jusqu'au déploiement opérationnel, en mettant en évidence les enjeux de reproductibilité, de portabilité et de structuration logicielle dans un projet d'Intelligence Artificielle.

# Table des matières

<b>Résumé</b>	<b>2</b>
<b>1 Introduction</b>	<b>4</b>
<b>2 Modélisation Deep Learning</b>	<b>5</b>
2.1 Compréhension du problème . . . . .	5
2.2 Présentation du dataset CIFAR-10 . . . . .	5
2.3 Prétraitement et Data Augmentation . . . . .	6
2.4 Architecture du réseau de neurones convolutionnel . . . . .	6
2.5 Analyse des dimensions et complexité du modèle . . . . .	6
2.6 Stratégie d'entraînement . . . . .	7
2.7 Early Stopping et performances obtenues . . . . .	7
2.8 Sauvegarde du modèle et gestion des checkpoints . . . . .	7
2.9 Limites du modèle et perspectives d'amélioration . . . . .	8
<b>3 Conteneurisation, Déploiement et Dimension MLOps</b>	<b>9</b>
3.1 Objectifs de la conteneurisation . . . . .	9
3.2 Architecture multi-services avec Docker Compose . . . . .	9
3.3 Conteneurisation du pipeline d'entraînement . . . . .	9
3.4 Conteneurisation et déploiement de l'API de prédiction . . . . .	10
3.5 Implémentation technique des images Docker . . . . .	10
3.6 Gestion des volumes : persistance des données et des modèles . . . . .	10
3.7 Configuration effective des volumes . . . . .	11
3.8 Orchestration et dépendances entre services . . . . .	11
3.9 Dimension MLOps : reproductibilité, traçabilité et séparation des responsabilités . .	11
3.10 Comparaison exécution locale vs exécution dockerisée . . . . .	11
<b>4 Validation expérimentale et analyse des résultats</b>	<b>13</b>
<b>5 Conclusion et perspectives</b>	<b>16</b>
5.1 Contraintes et difficultés rencontrées . . . . .	16
5.2 Conclusion . . . . .	16
5.3 Perspectives . . . . .	16

# 1 Introduction

L'essor du Deep Learning a profondément transformé le domaine de la vision par ordinateur, notamment dans les tâches de classification d'images. Les réseaux de neurones convolutionnels (CNN) se sont imposés comme une architecture de référence grâce à leur capacité à extraire automatiquement des caractéristiques hiérarchiques à partir de données visuelles.

Dans ce contexte, le dataset CIFAR-10 constitue un benchmark académique largement utilisé pour l'évaluation de modèles de classification d'images. Malgré la faible résolution des images ( $32 \times 32$  pixels), la diversité des classes et la similarité visuelle entre certaines catégories rendent la tâche non triviale.

L'objectif de ce projet est double. D'une part, développer et entraîner un modèle de Deep Learning capable d'atteindre des performances satisfaisantes sur CIFAR-10. D'autre part, mettre en œuvre une architecture logicielle complète intégrant la conteneurisation, la séparation des services d'entraînement et d'inférence, ainsi que le déploiement d'un service de prédiction accessible via une API.

L'utilisation de Docker permet de répondre aux enjeux de reproductibilité et de portabilité des environnements d'exécution, souvent critiques dans les projets d'Intelligence Artificielle. La structuration du projet selon une approche MLOps vise à rapprocher le développement expérimental d'un cadre plus industriel.

Le présent rapport décrit successivement la modélisation Deep Learning, la stratégie d'entraînement, l'architecture conteneurisée mise en place, ainsi que la validation expérimentale du système déployé.

## 2 Modélisation Deep Learning

### 2.1 Compréhension du problème

L'objectif de ce projet est de concevoir et entraîner un modèle de Deep Learning capable de classer automatiquement des images issues du dataset CIFAR-10.

Le problème étudié est un problème de classification multi-classes supervisée comportant dix catégories distinctes : avion, automobile, oiseau, chat, cerf, chien, grenouille, cheval, bateau et camion.

Chaque image est une image couleur RGB de dimension  $32 \times 32$  pixels. Le défi principal réside dans la faible résolution des images ainsi que dans la similarité visuelle entre certaines classes (par exemple chat et chien), ce qui rend la tâche de classification non triviale.

L'objectif final est d'obtenir un modèle :

- capable de généraliser correctement sur des données non vues,
- optimisé pour limiter le surapprentissage,
- suffisamment léger pour être déployé sous forme de service API.

### 2.2 Présentation du dataset CIFAR-10

Le dataset CIFAR-10 est un jeu de données public largement utilisé dans la recherche en vision par ordinateur.

Il contient un total de 60 000 images RGB de dimension  $32 \times 32$  pixels réparties en 10 classes équilibrées.

La répartition est la suivante :

- 50 000 images pour l'entraînement,
- 10 000 images pour le test.

Dans notre projet, le dataset d'entraînement est subdivisé en :

- 90% pour l'entraînement effectif,
- 10% pour la validation.

Ce découpage permet d'évaluer la capacité de généralisation du modèle durant l'entraînement.

[< Back to Alex Krizhevsky's home page](#)

The CIFAR-10 and CIFAR-100 datasets are labeled subsets of the 80 million tiny images dataset. CIFAR-10 and CIFAR-100 were created by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.

#### The CIFAR-10 dataset

The CIFAR-10 dataset consists of 60000  $32 \times 32$  colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

Here are the classes in the dataset, as well as 10 random images from each:

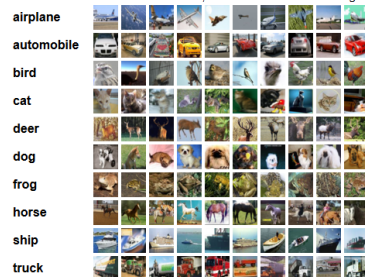


FIGURE 1 – Illustration des classes du dataset CIFAR-10 issue du site officiel

## 2.3 Prétraitement et Data Augmentation

Afin d'améliorer la robustesse du modèle et de limiter le surapprentissage, plusieurs techniques de data augmentation ont été appliquées aux images d'entraînement :

- Random Horizontal Flip,
- Random Rotation ( $\pm 15^\circ$ ),
- Random Affine Transformations (translations jusqu'à 10%),
- Random Resized Crop (échelle entre 0.9 et 1.0).

Ces transformations permettent d'augmenter artificiellement la diversité du dataset en introduisant des variations géométriques réalistes.

Les images sont ensuite converties en tenseurs PyTorch et normalisées selon :

$$mean = (0.5, 0.5, 0.5), \quad std = (0.5, 0.5, 0.5)$$

La normalisation permet d'améliorer la stabilité de la descente de gradient et d'accélérer la convergence.

## 2.4 Architecture du réseau de neurones convolutionnel

Le modèle développé est un réseau de neurones convolutionnel composé de trois blocs convolutionnels suivis de trois couches entièrement connectées.

Chaque bloc convolutionnel comprend :

- Deux couches Conv2D avec noyaux  $3 \times 3$ ,
- Batch Normalization,
- Fonction d'activation ReLU,
- MaxPooling  $2 \times 2$ ,
- Dropout 2D (0.25).

Le nombre de filtres évolue progressivement :

- Bloc 1 : 32 filtres,
- Bloc 2 : 64 filtres,
- Bloc 3 : 128 filtres.

Les couches fully connected sont structurées comme suit :

- $2048 \rightarrow 256$  (BatchNorm + ReLU + Dropout 0.5),
- $256 \rightarrow 128$  (BatchNorm + ReLU + Dropout 0.5),
- $128 \rightarrow 10$  (sortie finale).

## 2.5 Analyse des dimensions et complexité du modèle

L'évolution des dimensions spatiales est la suivante :

- Entrée :  $32 \times 32 \times 3$
- Après premier MaxPooling :  $16 \times 16 \times 32$
- Après second MaxPooling :  $8 \times 8 \times 64$
- Après troisième MaxPooling :  $4 \times 4 \times 128$

Avant l'entrée dans les couches fully connected, le tenseur est aplati :

$$128 \times 4 \times 4 = 2048$$

Ce nombre correspond à la dimension d'entrée de la première couche entièrement connectée.

Le modèle comporte environ 1.2 million de paramètres entraînaibles, ce qui représente un compromis entre capacité d'expression et complexité computationnelle.

## 2.6 Stratégie d'entraînement

L'entraînement du modèle a été réalisé sur processeur (CPU), le dispositif CUDA n'étant pas activé lors de l'exécution.

La fonction de perte utilisée est la **CrossEntropyLoss**, adaptée aux problèmes de classification multi-classes.

L'optimiseur choisi est **Adam** avec un taux d'apprentissage initial de :

$$\eta = 0.001$$

Un scheduler de type **ReduceLROnPlateau** est utilisé afin de réduire dynamiquement le taux d'apprentissage lorsque la perte de validation cesse de diminuer.

Le modèle est entraîné avec :

- Batch size : 64
- Maximum d'époques : 80
- Patience (early stopping) : 7

Le découpage des données est le suivant :

- 45 000 images pour l'entraînement,
- 5 000 images pour la validation,
- 10 000 images pour le test.

## 2.7 Early Stopping et performances obtenues

Un mécanisme d'**Early Stopping** est implémenté afin d'éviter le surapprentissage.

Le critère surveillé est l'accuracy de validation. Lorsque celle-ci n'augmente plus pendant un certain nombre d'époques consécutives (patience fixée à 7), l'entraînement est arrêté automatiquement.

Dans notre cas, l'entraînement s'est arrêté à l'époque 67.

Les performances obtenues sont :

- Accuracy entraînement : 79.13%
- Accuracy validation : 82.28%
- Loss validation : 0.5138

Le fait que l'accuracy de validation soit légèrement supérieure à celle d'entraînement peut s'expliquer par l'effet de la régularisation (dropout actif uniquement en mode entraînement).

## 2.8 Sauvegarde du modèle et gestion des checkpoints

Le modèle sauvegarde automatiquement les meilleurs poids obtenus sur la validation.

Le fichier `best_model.pth` contient :

- Les poids du modèle (`state_dict`),
- L'accuracy associée,
- L'historique complet d'entraînement (loss et accuracy).

Cette stratégie garantit :

- La reproductibilité des résultats,
- La possibilité de redémarrer l'entraînement,
- L'utilisation du meilleur modèle pour le déploiement.

## **2.9 Limites du modèle et perspectives d'amélioration**

Bien que le modèle atteigne une accuracy de validation supérieure à 82%, certaines limites subsistent :

- Les images CIFAR-10 ont une résolution très faible ( $32 \times 32$ ), limitant l'extraction de détails fins.
- Certaines classes sont visuellement proches (ex : chat et chien), rendant la séparation complexe.
- L'entraînement sur CPU augmente significativement le temps d'apprentissage.

Des améliorations potentielles incluent :

- L'utilisation d'un GPU pour accélérer l'entraînement,
- L'intégration d'architectures plus profondes (ResNet),
- L'utilisation du transfer learning,
- L'ajout de techniques d'augmentation plus avancées.

## 3 Conteneurisation, Déploiement et Dimension MLOps

### 3.1 Objectifs de la conteneurisation

L'utilisation de Docker dans ce projet répond à plusieurs objectifs essentiels dans un cycle MLOps :

- **Reproductibilité** : garantir que l'entraînement et l'inférence s'exécutent dans un environnement identique (versions Python, bibliothèques, dépendances système), indépendamment de la machine hôte.
- **Portabilité** : permettre l'exécution du projet sur n'importe quelle machine disposant de Docker, sans installation manuelle complexe.
- **Séparation des responsabilités** : isoler l'entraînement du modèle (service `train`) et le service de prédiction (service `api`), afin de refléter une architecture proche des systèmes industriels.
- **Déploiement simplifié** : fournir un service Flask conteneurisé accessible via une interface web et une API REST.

### 3.2 Architecture multi-services avec Docker Compose

Le projet adopte une architecture multi-services orchestrée via **Docker Compose**. Deux conteneurs distincts sont définis :

- **Service `train`** : responsable de l'entraînement du modèle CNN et de la sauvegarde du meilleur modèle.
- **Service `api`** : responsable du chargement du modèle entraîné et de l'exposition d'un service de prédiction via Flask (interface web + endpoints REST).

Cette séparation permet de distinguer clairement les phases *training* et *serving*, ce qui correspond aux pratiques standards en MLOps.

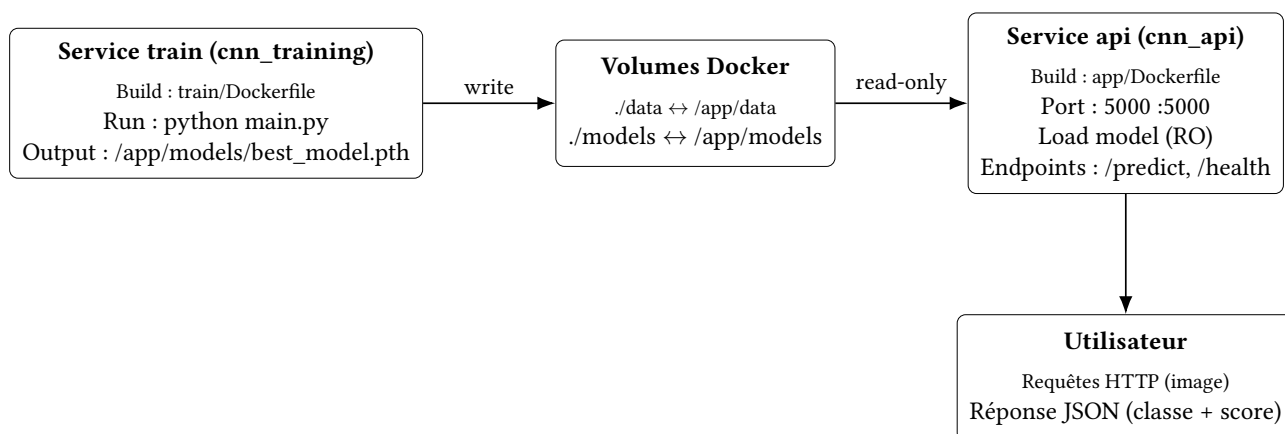


FIGURE 2 – Architecture multi-services du projet

### 3.3 Conteneurisation du pipeline d'entraînement

Le service `train` est construit à partir du fichier `train/Dockerfile`. L'image repose sur `python : 3.12-slim` et installe les dépendances système nécessaires (compilateurs `gcc` et `g++`) ainsi que les dépendances Python via `requirements.txt`.

Le point d'entrée du conteneur lance directement l'entraînement :

```
python main.py
```

L'entraînement télécharge automatiquement le dataset CIFAR-10 via "torchvision.datasets.CIFAR10" et sauvegarde le meilleur modèle sous la forme d'un fichier **best\_model.pth**.

### 3.4 Conteneurisation et déploiement de l'API de prédiction

Le service `api` est construit à partir du fichier `app/Dockerfile`. Il installe les dépendances puis copie le code de l'application, incluant le backend Flask (`app/app.py`) et le template HTML (`app/templates/index.html`).

L'API est exposée sur le port 5000 :

```
http://localhost:5000
```

Au démarrage, l'application charge le modèle sauvegardé depuis :

```
/app/models/best_model.pth
```

Le modèle est placé en mode évaluation (`model.eval()`) et les prédictions sont effectuées sans calcul de gradients (`torch.no_grad()`), ce qui optimise l'inférence.

### 3.5 Implémentation technique des images Docker

Les deux services reposent sur des Dockerfiles distincts adaptés à leurs responsabilités.

Les images sont construites à partir de `python:3.12-slim`, assurant un environnement Python homogène pour l'entraînement et l'inférence. Les compilateurs `gcc` et `g++` sont installés afin de garantir la compatibilité avec certaines dépendances nécessitant une compilation native.

Les dépendances sont centralisées dans le fichier `requirements.txt` et installées lors de la phase de build, garantissant la reproductibilité de l'environnement d'exécution.

Dans le service `train`, seul le dossier `train/` est copié dans l'image, et l'exécution démarre via `python main.py`. Dans le service `api`, l'ensemble de l'application est copié, incluant le backend Flask et les templates HTML, puis lancé via `python app/app.py`.

### 3.6 Gestion des volumes : persistance des données et des modèles

Deux volumes Docker sont utilisés afin d'assurer la persistance des artefacts clés :

- **Données** : `./data` monté sur `/app/data`
- **Modèles** : `./models` monté sur `/app/models`

Cette stratégie permet :

- de conserver le dataset téléchargé entre plusieurs exécutions (évite de re-télécharger CIFAR-10),
- de conserver les modèles entraînés et checkpoints même après arrêt/suppression des conteneurs,
- de partager le modèle entraîné entre le service `train` et le service `api`.

Dans le service `api`, le volume des modèles est monté en **lecture seule** (`:ro`), ce qui constitue une bonne pratique de sécurité : l'API ne peut pas modifier le modèle, elle ne fait que le charger.

### 3.7 Configuration effective des volumes

Le volume `./models` est monté en écriture dans le service `train`, permettant la sauvegarde du fichier `best_model.pth`.

Dans le service `api`, ce même volume est monté en lecture seule (`:ro`), garantissant que le modèle chargé en production ne puisse pas être modifié.

Le volume `./data` permet de conserver localement le dataset téléchargé, évitant son re-téléchargement lors des exécutions successives.

### 3.8 Orchestration et dépendances entre services

Docker Compose est utilisé pour orchestrer l'exécution des services et assurer un ordre logique :

- le service `train` s'exécute en premier et entraîne le modèle,
- le service `api` ne démarre que lorsque `train` a terminé avec succès.

Cette dépendance est implémentée via :

```
depends_on: condition: service_completed_successfully
```

Ainsi, l'API est garantie de disposer d'un modèle entraîné avant de démarrer, ce qui évite les erreurs de démarrage liées à l'absence du fichier `best_model.pth`.

### 3.9 Dimension MLOps : reproductibilité, traçabilité et séparation des responsabilités

Le projet met en évidence plusieurs principes MLOps :

- **Reproductibilité** : l'environnement d'entraînement et d'inférence est défini par les Dockerfiles et `requirements.txt`.
- **Séparation training/serving** : l'entraînement et le déploiement sont isolés en deux services indépendants.
- **Traçabilité** : le checkpoint `best_model.pth` inclut l'accuracy et l'historique d'entraînement (`history`), permettant d'analyser a posteriori les performances.
- **Persistence des artefacts** : dataset et modèles sont conservés via des volumes, évitant les pertes et favorisant la réutilisation.

Cette organisation facilite le passage d'un prototype à une solution déployable et réutilisable, tout en limitant les divergences entre environnement de développement et environnement d'exécution.

### 3.10 Comparaison exécution locale vs exécution dockerisée

Une exécution locale du projet nécessite l'installation manuelle des dépendances (Python 3.12, PyTorch, torchvision, Flask, etc.) ainsi que la gestion des versions et des bibliothèques système. Cette approche peut conduire à des différences d'exécution selon les machines (OS, versions, conflits de dépendances).

À l'inverse, l'exécution dockerisée encapsule l'environnement dans des images reproductibles. Les étapes de build et d'exécution via Docker Compose permettent de :

- standardiser l'environnement d'entraînement et de déploiement,

- automatiser le téléchargement des données,
- persister dataset et modèles via des volumes,
- garantir un déploiement identique entre plusieurs machines.

Ainsi, même si le temps d'entraînement dépend toujours des ressources matérielles (CPU/GPU), Docker améliore significativement la reproductibilité et la facilité de déploiement.

## 4 Validation expérimentale et analyse des résultats

L'entraînement du modèle s'est déroulé sur processeur (CPU) et s'est arrêté automatiquement à l'époque 67 grâce au mécanisme d'Early Stopping. L'accuracy de validation obtenue est de 82.28%, ce qui confirme la capacité du modèle à généraliser correctement sur des données non vues.

L'accuracy d'entraînement (79.13%) étant légèrement inférieure à celle de validation, ce comportement s'explique par l'effet des techniques de régularisation mises en œuvre, notamment le Dropout, actif uniquement en mode entraînement.

### Validation du déploiement Docker

Le déploiement via Docker Compose a permis de valider le bon fonctionnement de l'architecture multi-services. Les conteneurs `cnn_training` et `cnn_api` ont été correctement orchestrés, et l'API ne s'est lancée qu'après la fin réussie de l'entraînement grâce à la directive `depends_on`.

La Figure 3 illustre l'exécution simultanée des deux services.

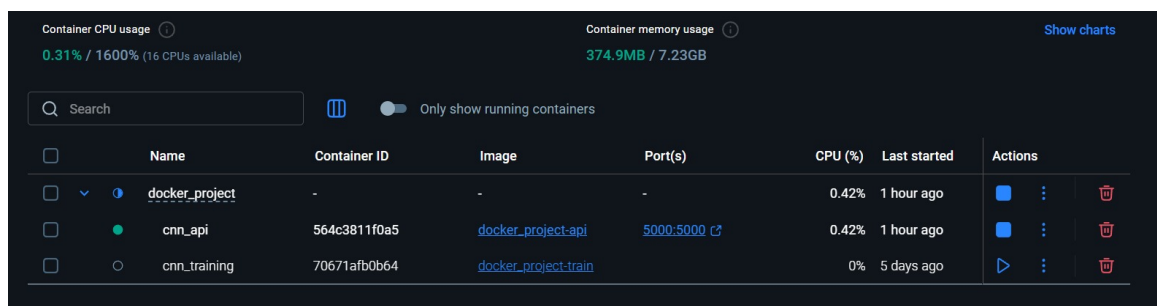
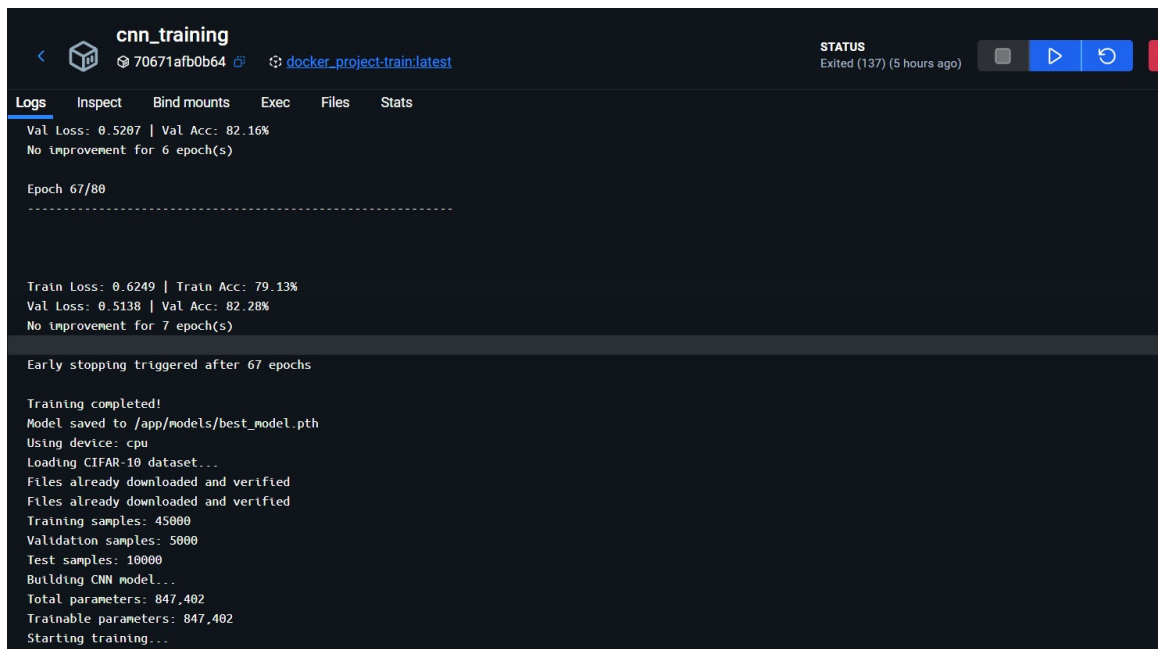


FIGURE 3 – Exécution des services Docker (`cnn_training` et `cnn_api`) via Docker Compose.

### Analyse des logs d'exécution

Les logs du service d'entraînement (Figure 4) confirment :

- l'initialisation du modèle comportant 847 402 paramètres entraînaibles,
- le téléchargement et le chargement du dataset CIFAR-10,
- l'affichage des métriques (loss et accuracy) à chaque époque,
- le déclenchement de l'Early Stopping à l'époque 67,
- la sauvegarde du modèle dans `/app/models/best_model.pth`.



```
cnn_training
70671afb0b64 docker_project-train:latest
STATUS
Exited (137) (5 hours ago)

Logs
Val Loss: 0.5207 | Val Acc: 82.16%
No improvement for 6 epoch(s)

Epoch 67/80
-----

Train Loss: 0.6249 | Train Acc: 79.13%
Val Loss: 0.5138 | Val Acc: 82.28%
No improvement for 7 epoch(s)

Early stopping triggered after 67 epochs

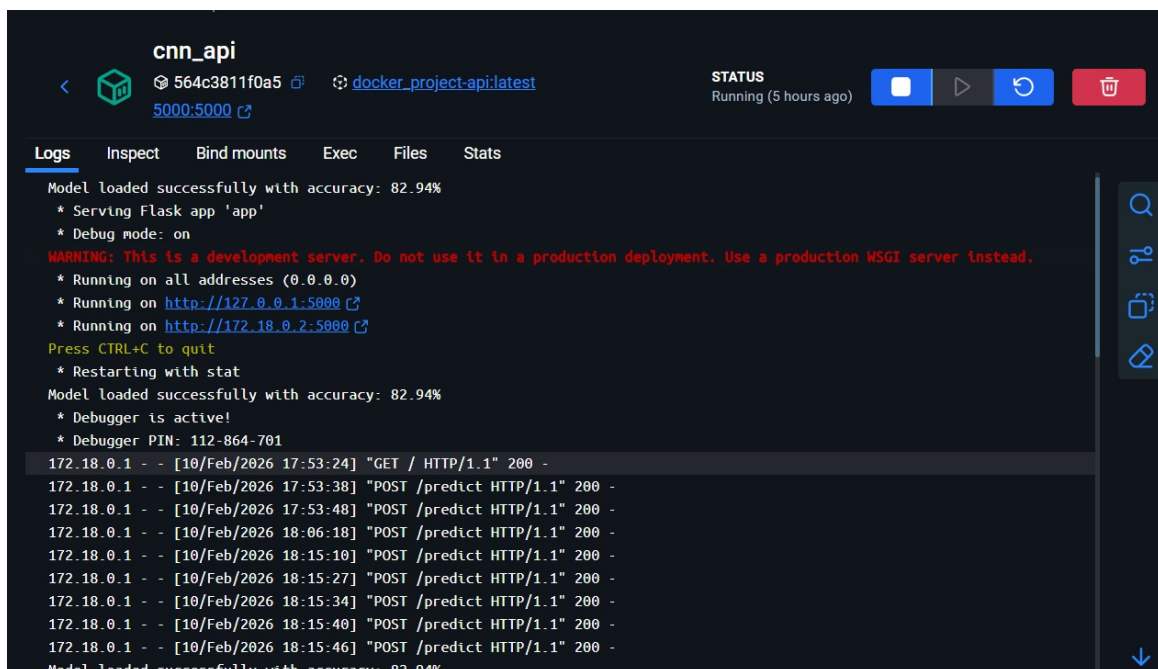
Training completed!
Model saved to /app/models/best_model.pth
Using device: cpu
Loading CIFAR-10 dataset...
Files already downloaded and verified
Files already downloaded and verified
Training samples: 45000
Validation samples: 5000
Test samples: 10000
Building CNN model...
Total parameters: 847,402
Trainable parameters: 847,402
Starting training...
```

FIGURE 4 – Extrait des logs du service `cnn_training` montrant l'arrêt anticipé à l'époque 67.

Les logs du service API (Figure 5) indiquent :

- le chargement réussi du modèle avec une accuracy sauvegardée de 82.94%,
- le démarrage du serveur Flask sur l'adresse `0.0.0.0:5000`,
- la réception et le traitement de requêtes HTTP sur l'endpoint `/predict`.

Ces éléments confirment la cohérence entre la phase d'entraînement et la phase de déploiement.



```
cnn_api
564c3811f0a5 docker_project-api:latest
STATUS
Running (5 hours ago)

Logs
Model loaded successfully with accuracy: 82.94%
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.18.0.2:5000
Press CTRL+C to quit
* Restarting with stat
Model loaded successfully with accuracy: 82.94%
* Debugger is active!
* Debugger PIN: 112-864-701

172.18.0.1 - - [10/Feb/2026 17:53:24] "GET / HTTP/1.1" 200 -
172.18.0.1 - - [10/Feb/2026 17:53:38] "POST /predict HTTP/1.1" 200 -
172.18.0.1 - - [10/Feb/2026 17:53:48] "POST /predict HTTP/1.1" 200 -
172.18.0.1 - - [10/Feb/2026 18:06:18] "POST /predict HTTP/1.1" 200 -
172.18.0.1 - - [10/Feb/2026 18:15:10] "POST /predict HTTP/1.1" 200 -
172.18.0.1 - - [10/Feb/2026 18:15:27] "POST /predict HTTP/1.1" 200 -
172.18.0.1 - - [10/Feb/2026 18:15:34] "POST /predict HTTP/1.1" 200 -
172.18.0.1 - - [10/Feb/2026 18:15:40] "POST /predict HTTP/1.1" 200 -
172.18.0.1 - - [10/Feb/2026 18:15:46] "POST /predict HTTP/1.1" 200 -
Model loaded successfully with accuracy: 82.94%
```

FIGURE 5 – Extrait des logs du service `cnn_api` confirmant le chargement du modèle et la réception des requêtes.

## Validation fonctionnelle via l'interface web

Les tests réalisés via l'interface graphique montrent que le modèle est capable de classifier correctement des images envoyées par l'utilisateur.

La Figure 6 présente un exemple de prédiction d'une image de chien.

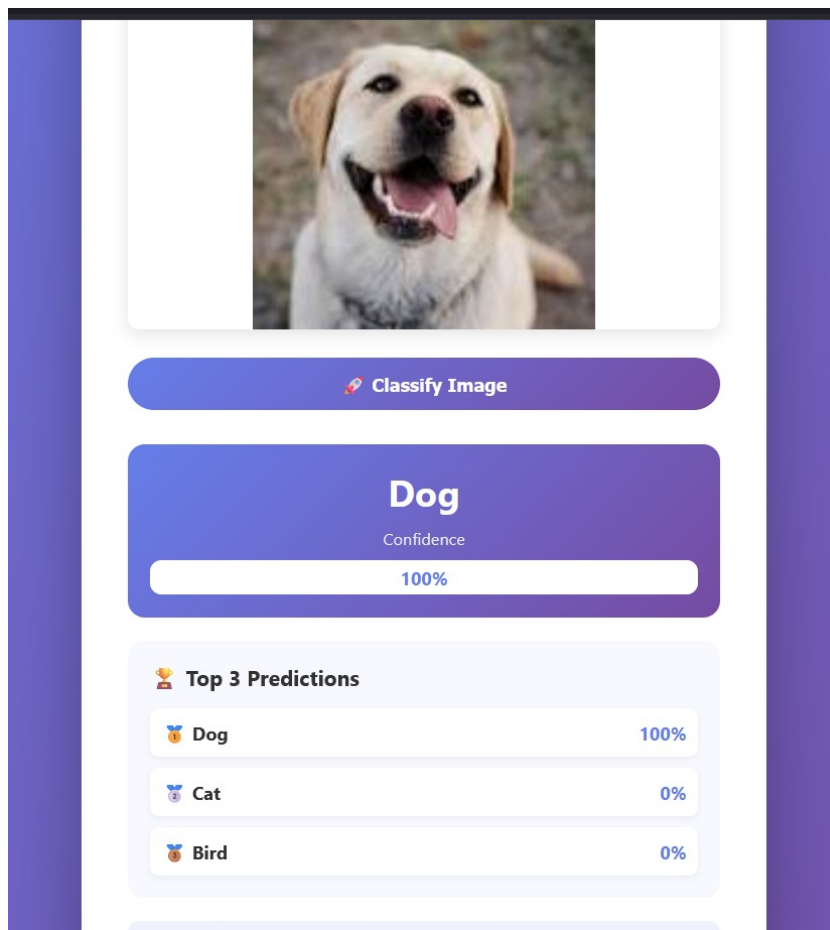


FIGURE 6 – Exemple de classification d'une image via l'interface web avec score de confiance et top-3 des classes.

Dans cet exemple, le modèle attribue la probabilité maximale à la classe `dog`. Le score de confiance élevé indique que les caractéristiques visuelles extraites par le réseau correspondent fortement à cette catégorie.

La présence des trois classes les plus probables améliore l'interprétabilité des prédictions et permet d'observer la hiérarchie des scores issue de la couche Softmax.

Ces résultats démontrent le bon fonctionnement de l'ensemble du pipeline, depuis l'entraînement jusqu'à l'inférence en environnement conteneurisé.

## 5 Conclusion et perspectives

### 5.1 Contraintes et difficultés rencontrées

Plusieurs contraintes techniques ont été rencontrées lors du développement du projet.

L'entraînement sur processeur (CPU) a limité la rapidité d'expérimentation et d'optimisation des hyperparamètres. La gestion des volumes Docker a également nécessité une attention particulière afin d'assurer la persistance correcte des modèles et des données.

La séparation stricte entre entraînement et inférence a impliqué une duplication contrôlée de l'architecture du modèle entre les services `train` et `api`, nécessitant une cohérence rigoureuse dans l'implémentation.

Ces contraintes ont toutefois permis de structurer le projet de manière plus robuste et reproductible.

### 5.2 Conclusion

Ce projet a permis de développer un système complet de classification d'images intégrant à la fois la conception d'un modèle de Deep Learning et son déploiement opérationnel.

Le réseau de neurones convolutionnel implémenté atteint une accuracy de validation de 82.28%, démontrant une capacité de généralisation satisfaisante compte tenu de la complexité du dataset CIFAR-10 et de la résolution limitée des images. L'utilisation combinée de techniques de régularisation et d'un mécanisme d'Early Stopping a permis de stabiliser l'apprentissage et de limiter le surapprentissage.

Sur le plan logiciel, la conteneurisation via Docker et l'orchestration multi-services via Docker Compose ont permis de structurer clairement le pipeline en séparant les phases d'entraînement et d'inférence. Cette organisation améliore la reproductibilité, la portabilité et la maintenabilité du projet.

L'intégration d'une API REST accompagnée d'une interface web interactive démontre la capacité à transformer un modèle expérimental en une application fonctionnelle, illustrant ainsi le passage d'un prototype académique à une solution exploitable.

### 5.3 Perspectives

Plusieurs pistes d'amélioration peuvent être envisagées afin d'étendre ce travail.

Sur le plan algorithmique, l'utilisation d'architectures plus profondes telles que ResNet ou EfficientNet pourrait permettre d'améliorer les performances. L'intégration de techniques de transfer learning constituerait également une évolution naturelle vers des performances accrues.

Sur le plan computationnel, l'entraînement sur GPU permettrait de réduire significativement le temps d'apprentissage et faciliterait l'exploration d'hyperparamètres plus complexes.

D'un point de vue MLOps, l'intégration d'un système de versioning des modèles (par exemple MLflow), la mise en place d'une pipeline CI/CD pour l'automatisation des builds Docker ou le déploiement sur une infrastructure cloud constitueraient des évolutions vers un environnement industriel.

Enfin, l'ajout d'un système de monitoring des performances en production permettrait de détecter d'éventuelles dérives du modèle et d'assurer une maintenance continue du service.