Réf : . . ..../. . . . . .

AU : 2024-2025

Université de Sousse

École Nationale d'Ingénieurs de Sousse

Rapport de Projet Semestriel :
Modèle de Few-Shot Learning
pour la classification d'images thermiques de moteurs à induction en régime normal et défectueux.

Réalisé par :

Ben Saad Ismail
Lejmi Wael

Filière :

Informatique Appliquée

Groupe :

IA 2.2

Encadrents :

M. Aroui Tarek
M. Mekki Hassan

Table des matières

1 1 - Introduction **4**
1.1 Contexte 4
1.2 Problématique 4
1.3 Objectifs du projet 4

2 2 - Exploration des données **5**
2.1 Présentation du dataset 5
2.2 Répartition des classes et du nombre d'images 5

3 3 - Apprentissage Few-Shot **7**
3.1 Présentation du Few-Shot Learning 7
3.1.1 Définition et utilité 7
3.1.2 Problèmes que le FSL résout 7
3.2 Défis de l'Apprentissage Few-Shot 7
3.2.1 Risque de surapprentissage 8
3.2.2 Architecture du modèle 8
3.2.3 Augmentation de données 8
3.3 Approches du Few-Shot Learning 8
3.3.1 Apprentissage par transfert 8
3.3.2 Méta-apprentissage 8

4 4 - Épisodes et Stratégie d'Échantillonnage **11**
4.1 N-way, K-shot, Q-query 11
4.1.1 Importance de cette stratégie 11

5 5 - Approche Proposée **12**
5.1 Prototypical Networks 12
5.1.1 Fonctionnement 12
5.1.2 Avantages pour la tâche 12
5.2 Encodeur : ResNet18 13
5.2.1 Choix du modèle 13
5.2.2 Résumé de l'architecture ResNet18 14
5.2.3 Architecture du modèle utilisé 14

6 6 - Entraînement et Résultats / Expérimentations **16**
6.1 Premières observations 16
6.2 Dégel partiel du backbone 16
6.3 Variantes de configuration N, K, Q 16
6.4 Optimiseur et planification du taux d'apprentissage 17
6.5 Early stopping 17
6.6 Tentative avec MobileNetV2 17
6.7 Importance de l'augmentation des données 17
6.8 Résultats finaux 18
6.9 Test du modèle en ligne (Colab) 19

7 7 - Conclusion et perspectives **20**

1 - Introduction

1.1 Contexte

L'avènement de l'intelligence artificielle et de l'apprentissage profond a permis des avancées majeures dans le domaine de la vision par ordinateur, notamment en classification d'images. Toutefois, la performance de ces modèles dépend généralement de la disponibilité de grandes quantités de données annotées, ce qui représente un défi dans de nombreux cas pratiques.

Dans ce contexte, le *Few-Shot Learning* émerge comme une solution prometteuse. Il permet d'entraîner des modèles capables de généraliser à de nouvelles classes avec très peu d'exemples. Cette approche est particulièrement adaptée aux situations où l'acquisition de données est coûteuse ou difficile, comme c'est le cas pour les images thermiques de moteurs industriels.

1.2 Problématique

Dans le domaine industriel, la détection précoce de défauts dans les moteurs à induction est essentielle pour éviter des pannes coûteuses. Cependant, les données d'entraînement représentant des conditions anormales sont souvent rares ou difficiles à obtenir. Cela rend les méthodes d'apprentissage traditionnelles peu efficaces, car elles nécessitent un grand volume de données équilibrées.

1.3 Objectifs du projet

L'objectif de ce projet est de développer un modèle de *Few-Shot Learning* capable de classifier des images thermiques de moteurs à induction, en distinguant les états de fonctionnement normal et défectueux. Pour cela, nous avons utilisé un jeu de données contenant des images thermiques prises dans diverses conditions. Le modèle développé devra être capable de s'adapter à de nouvelles classes ou situations en ne recevant qu'un petit nombre d'exemples.

2 - Exploration des données

2.1 Présentation du dataset

Le jeu de données utilisé dans ce projet est intitulé *Thermal Images of Induction Motor* et est disponible publiquement sur la plateforme Kaggle.[1] Il contient des images thermiques capturées par une caméra infrarouge de la marque Dali-Tech (modèles T4/T8) à une température ambiante d'environ 23°C.

Les images sont réparties en plusieurs dossiers représentant différents états de fonctionnement de moteurs à induction et de transformateurs, notamment des états normaux ainsi que divers types de défaillances (court-circuit, ventilateur défaillant, rotor endommagé, etc.). Chaque image est au format `.bmp` avec une résolution de 320x240 pixels.

2.2 Répartition des classes et du nombre d'images

Pour ce projet, nous avons choisi de nous concentrer exclusivement sur les images thermiques des **moteurs à induction**, en excluant les données relatives aux transformateurs. Cela représente un total de **369 images**, réparties selon différents états de fonctionnement normaux et défectueux. Les défauts simulés incluent notamment des courts-circuits dans une, deux ou trois phases du stator, à différents taux (10%, 30%, 50%), ainsi que des défauts de refroidissement, de rotor, ou encore des moteurs sains.

La répartition des images par condition est donnée dans le tableau ci-dessous :

Table 2.1 – Répartition des images selon les défauts du moteur à induction

Condition	**Nombre d'images**
Court-circuit 50% stator (multi-phases)	38
Court-circuit 30% stator (multi-phases)	42
Court-circuit 10% stator (multi-phases)	31
Court-circuit 30% stator (2 phases)	38
Court-circuit 10% stator (2 phases)	31
Court-circuit 10% stator (1 phase)	34
Court-circuit 30% stator (1 phase)	37
Court-circuit 50% stator (1 phase)	35
Défaut de refroidissement (Fan)	28
Défaut de rotor	30
Moteur sain (Noload)	25
Total	**369**

1. https://www.kaggle.com/datasets/amirberenji/thermal-images-of-induction-motor

Afin d'avoir une première visualisation qualitative du dataset, nous avons utilisé une fonction de visualisation aléatoire pour afficher un échantillon de 10 images issues de différentes classes. Cela permet d'observer l'homogénéité visuelle des images et la similarité apparente entre les classes.

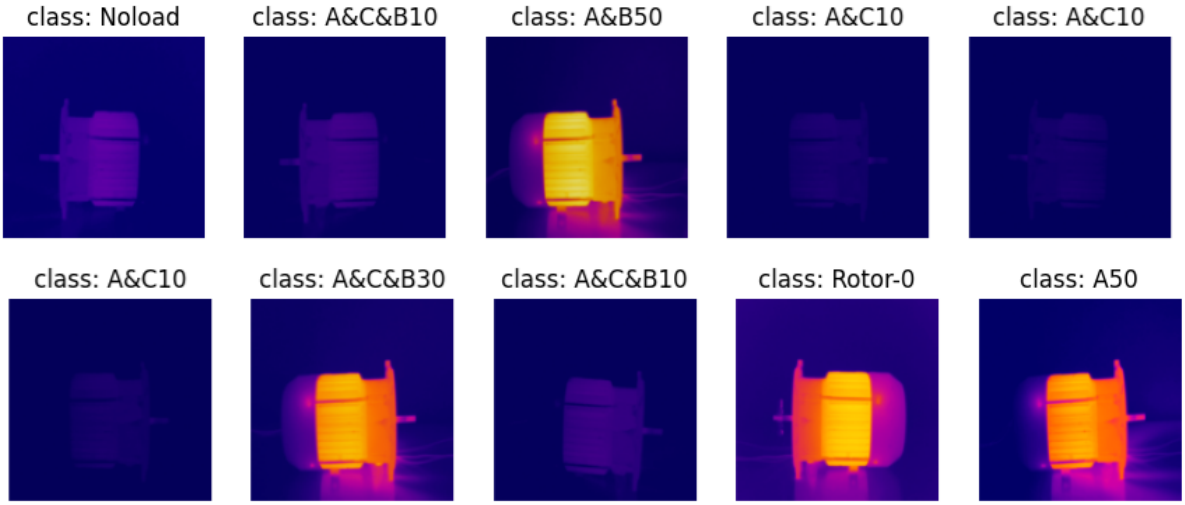

FIGURE 2.1 – Exemple de 10 images thermiques aléatoires issues du dataset moteur

3 - Apprentissage Few-Shot

3.1 Présentation du Few-Shot Learning

3.1.1 Définition et utilité

L'apprentissage few-shot (FSL - Few-Shot Learning) est un paradigme de machine learning où un modèle apprend à faire des prédictions précises à partir d'un très petit nombre d'exemples étiquetés. Contrairement à l'apprentissage supervisé classique, qui nécessite des milliers d'exemples pour chaque classe, le FSL permet à un modèle de généraliser avec seulement quelques exemples par classe. Ce paradigme inclut des variantes comme l'apprentissage one-shot (un seul exemple par classe) et zero-shot (aucun exemple étiqueté).

Le FSL est particulièrement utile dans des contextes où les données sont difficiles à obtenir ou coûteuses à annoter, tels que la reconnaissance de maladies rares, la classification d'espèces en voie d'extinction, ou la détection de défauts dans des équipements industriels. Par exemple, dans ce projet, le FSL est appliqué à la classification d'images thermiques de moteurs à induction, ce qui permet de surmonter la limitation des données étiquetées rares tout en maintenant des performances robustes.

3.1.2 Problèmes que le FSL résout

L'apprentissage few-shot répond à plusieurs défis majeurs du machine learning traditionnel, notamment :

— **Rareté des données** : Dans des domaines comme la médecine ou la reconnaissance d'espèces rares, il est difficile d'obtenir une grande quantité d'exemples étiquetés. Par exemple, dans le cas des moteurs à induction, les défaillances spécifiques peuvent être rares.

— **Coût d'annotation** : L'étiquetage manuel de grandes quantités de données est souvent coûteux et chronophage, en particulier lorsqu'il nécessite une expertise spécialisée.

— **Adaptabilité** : Le FSL permet d'adapter rapidement des modèles à de nouvelles classes sans nécessiter un réentraînement complet. Par exemple, un modèle préentrainé pour détecter un type de défaut peut facilement être adapté à de nouvelles défaillances.

— **Généralisation** : Le FSL permet de développer des modèles capables de généraliser à partir de peu d'exemples, ce qui est crucial pour éviter la simple mémorisation des données.

3.2 Défis de l'Apprentissage Few-Shot

Le FSL présente plusieurs défis techniques importants :

3.2.1 Risque de surapprentissage

Avec très peu d'échantillons d'entraînement, les modèles risquent de mémoriser les exemples plutôt que d'apprendre des représentations généralisables. Ce problème est particulièrement critique lorsqu'on utilise des architectures complexes avec de nombreux paramètres, comme les réseaux de neurones convolutifs (CNN). Le surapprentissage peut conduire à des performances élevées sur l'ensemble d'entraînement, mais à une faible capacité de généralisation.

3.2.2 Architecture du modèle

L'architecture du modèle doit être suffisamment flexible pour s'adapter à des données limitées tout en extrayant des caractéristiques généralisables. Le défi consiste à trouver un équilibre entre la complexité du modèle et sa capacité à généraliser à partir de peu d'exemples. Des modèles trop complexes risquent de surapprendre, tandis que des modèles trop simples peuvent ne pas capturer les nuances nécessaires pour bien généraliser.

3.2.3 Augmentation de données

Pour compenser le manque d'échantillons, l'augmentation de données devient cruciale. Cette technique permet de créer artificiellement de nouveaux exemples d'entraînement en appliquant diverses transformations aux échantillons disponibles (rotation, translation, zoom, etc.), enrichissant ainsi l'ensemble d'apprentissage sans recueillir de nouvelles données. L'augmentation de données permet d'introduire une plus grande variété d'exemples et aide à améliorer la robustesse du modèle.

3.3 Approches du Few-Shot Learning

Le FSL repose principalement sur deux grandes approches :

3.3.1 Apprentissage par transfert

L'apprentissage par transfert consiste à adapter un modèle pré-entraîné pour apprendre de nouvelles tâches ou classes inconnues. Cette approche est efficace car elle exploite les représentations déjà apprises par le modèle, ce qui permet de réduire la quantité de données nécessaires pour l'entraînement. Dans un contexte few-shot, les techniques courantes incluent :

— Le réglage fin (fine-tuning) des couches externes d'un réseau neuronal
— Le gel des poids des couches internes pour préserver les connaissances acquises
— L'adaptation de l'architecture réseau pour de nouvelles tâches

3.3.2 Méta-apprentissage

Le méta-apprentissage, ou "apprendre à apprendre", consiste à entraîner un modèle non pas pour une tâche spécifique, mais pour qu'il puisse s'adapter rapidement à une large variété de tâches, même avec peu de données. Contrairement à l'apprentissage traditionnel, il cherche à extraire des connaissances de plusieurs tâches d'entraînement afin de les transférer à de nouvelles tâches.

Dans ce paradigme :

— Les modèles sont entraînés sur une distribution de tâches plutôt qu'une seule.

— Ils apprennent à identifier les régularités entre les tâches afin de faciliter l'adaptation future.

— L'objectif est de capturer une structure commune qui permet une généralisation rapide.

Les approches de méta-apprentissage se divisent principalement en deux grandes familles : les méthodes basées sur les métriques et celles basées sur l'optimisation.

Méta-apprentissage basé sur les métriques

Ces méthodes consistent à apprendre un espace d'embedding dans lequel les comparaisons entre exemples sont facilitées. La classification repose sur la similarité entre vecteurs de caractéristiques.

Voici les principales architectures :

— **Réseaux siamois** : Deux entrées passent par des branches identiques du réseau. L'objectif est de minimiser la distance entre les paires similaires et de la maximiser entre les paires dissemblables à l'aide d'une perte contrastive.

— **Matching Networks** : Utilisent une attention basée sur la distance cosinus entre les représentations des échantillons pour effectuer une classification directe par pondération des labels du support set.

— **Prototypical Networks** : Calculent un prototype (moyenne vectorielle) pour chaque classe dans l'espace d'embedding, puis utilisent la distance euclidienne entre une requête et les prototypes pour assigner une classe. Cette méthode est à la fois simple et efficace, particulièrement adaptée aux données bien structurées.

— **Relation Networks** : Apprennent une fonction de distance non linéaire à l'aide d'un second réseau neuronal. Plutôt que de calculer des distances fixes (cosinus, euclidienne), ils apprennent à juger de la similarité entre une requête et un exemple de support.

Méta-apprentissage basé sur l'optimisation

Aussi appelées approches à base de gradients (Gradient-Based Meta Learning, GBML), ces méthodes cherchent à optimiser les paramètres d'un modèle de manière à ce qu'il puisse être ajusté rapidement à de nouvelles tâches à l'aide de seulement quelques étapes de descente de gradient. L'idée est de rendre le processus d'optimisation lui-même efficace.

— Plutôt que d'entraîner un modèle pour une tâche fixe, on apprend une stratégie d'initialisation ou d'optimisation efficace pour toute une distribution de tâches.

— L'objectif est d'atteindre de bonnes performances en quelques étapes d'ajustement (fine-tuning), même avec très peu de données.

— Cela permet de tirer parti de l'efficacité de l'apprentissage par gradient tout en assurant une généralisation rapide.

Des exemples notables incluent :

— **MAML (Model-Agnostic Meta-Learning)** : Propose une initialisation des paramètres du modèle qui peut être adaptée à n'importe quelle tâche par quelques mises à jour de gradient. Son caractère "agnostique" le rend compatible avec toute architecture optimisable par descente de gradient.

— **LSTM meta-learner** : Utilise un réseau LSTM pour apprendre comment ajuster les paramètres d'un modèle principal, agissant comme un optimiseur appris.

— **Latent Embedding Optimization (LEO)** : Apprend dans un espace latent de plus faible dimension, ce qui permet une adaptation plus efficace avec peu de données. Il optimise les représentations avant même d'optimiser le classificateur.

Cette catégorie de méthodes est particulièrement utile dans des contextes où la rapidité d'adaptation est essentielle et où les nouvelles tâches peuvent varier fortement en contenu ou en structure.

4 - Épisodes et Stratégie d'Échantillonnage

4.1 N-way, K-shot, Q-query

L'entraînement et l'évaluation des modèles FSL suivent généralement un cadre appelé "N-way-K-shot", où :

— **N** représente le nombre de classes différentes dans chaque tâche.

— **K** indique le nombre d'exemples ("shots") fournis pour chaque classe.

— **Q** désigne le nombre d'exemples de requête par classe utilisés pour l'évaluation.

Chaque épisode d'entraînement comprend :

— Un **ensemble de support** : N classes avec K exemples par classe, utilisés pour apprendre les représentations.

— Un **ensemble de requêtes** : Q exemples supplémentaires par classe, utilisés pour évaluer le modèle.

Par exemple, dans une tâche 5-way-1-shot avec 15-query, chaque épisode présente 5 classes différentes, avec 1 exemple d'entraînement et 15 exemples de test par classe.

4.1.1 Importance de cette stratégie

Cette approche par épisodes est fondamentale pour le FSL pour plusieurs raisons :

— **Simulation réaliste** : Elle reproduit fidèlement les conditions dans lesquelles le modèle sera utilisé en production, où de nouvelles classes peuvent apparaître avec peu d'exemples disponibles.

— **Adaptabilité** : Le modèle doit apprendre à s'adapter rapidement aux nouvelles classes avec un nombre limité d'exemples.

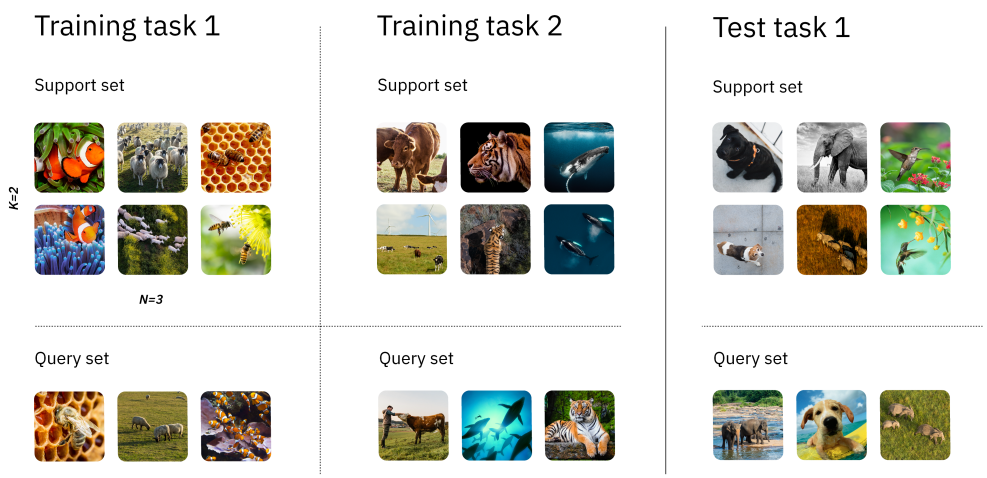

FIGURE 4.1 – Exemple de tâche de classification 3-way-2-shot pour l'apprentissage few-shot

5 - Approche Proposée

5.1 Prototypical Networks

5.1.1 Fonctionnement

Les réseaux prototypiques représentent une approche particulièrement efficace du Few-Shot Learning (FSL) basée sur les distances entre exemples dans un espace de représentation. Leur fonctionnement repose sur les étapes suivantes :

1. Chaque image d'entrée est transformée en un vecteur de caractéristiques (embedding) à l'aide d'un réseau de neurones convolutionnel (CNN). Ces vecteurs permettent de projeter les données dans un espace latent où les similarités s'expriment géométriquement.
2. Pour chaque classe de la tâche (épisode), on calcule un *prototype*, défini comme la moyenne des embeddings des exemples de cette classe. Ce prototype agit comme un représentant central de la classe dans l'espace latent.
3. Une nouvelle image (appelée *requête*) est encodée de la même manière, puis sa distance euclidienne est calculée par rapport à chacun des prototypes.
4. L'image est ensuite classée selon la classe dont le prototype est le plus proche — c'est-à-dire celle avec la plus faible distance euclidienne dans l'espace des caractéristiques.

Ce mécanisme repose sur l'hypothèse que les classes forment des groupes cohérents dans l'espace des embeddings. L'utilisation de distances permet une classification directe, sans couche de prédiction dense.

5.1.2 Avantages pour la tâche

Les réseaux prototypiques offrent de nombreux atouts pour les scénarios à faible nombre d'exemples :

— **Simplicité** : L'approche est conceptuellement simple et ne nécessite pas de fine-tuning complexe.

— **Efficacité computationnelle** : Moins coûteuse que les méthodes à base de gradients multiples (comme MAML).

— **Excellente généralisation** : Les prototypes permettent une bonne séparation des classes même avec peu d'exemples.

— **Interprétabilité** : Chaque prototype peut être interprété comme la "moyenne" de sa classe, facilitant l'analyse du comportement du modèle.

— **Robustesse** : L'approche est moins sensible au surapprentissage sur de petits jeux de données.

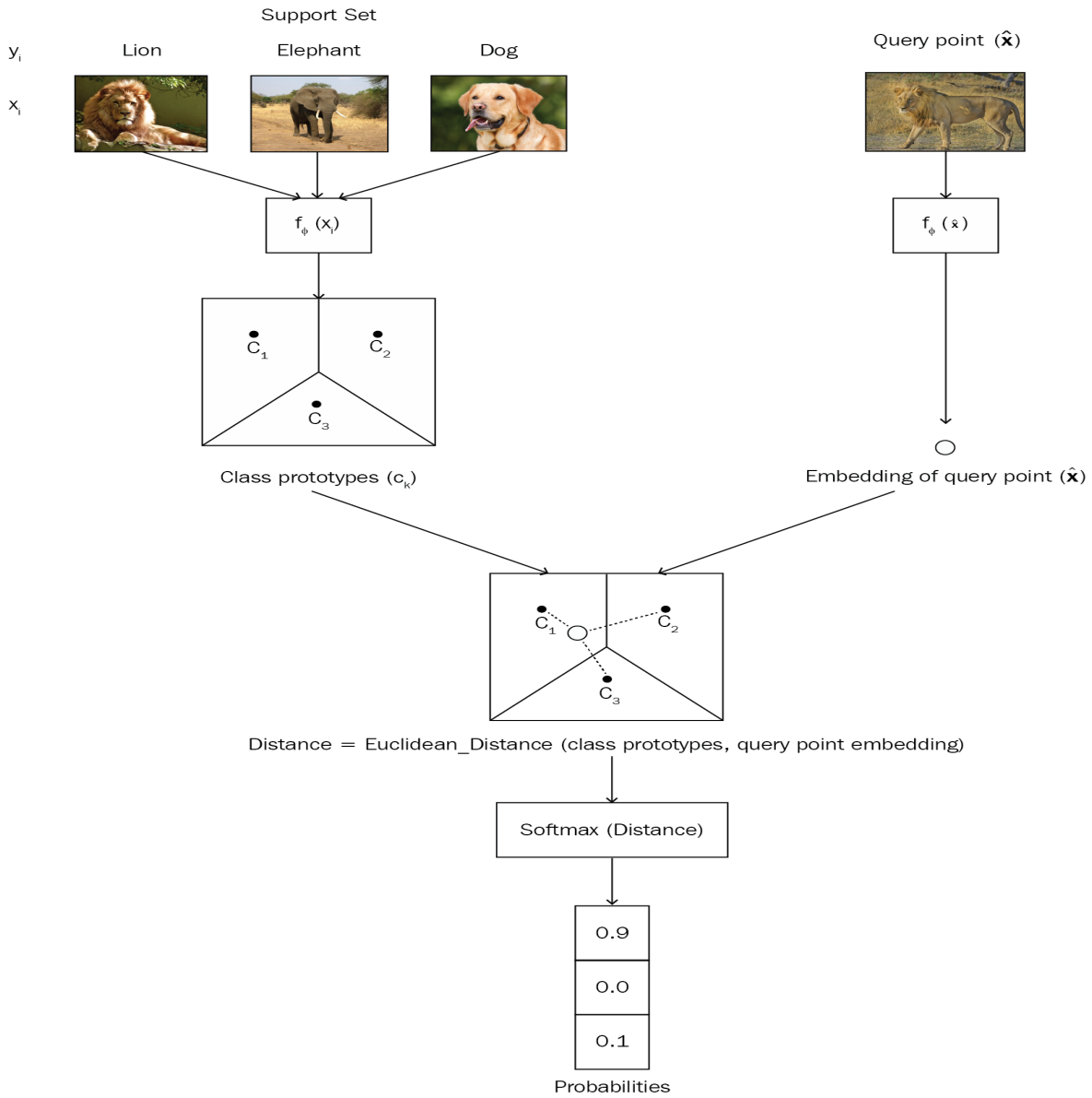

FIGURE 5.1 – Graphe de fonctionnement des Prototypical Networks

5.2 Encodeur : ResNet18

5.2.1 Choix du modèle

L'encodeur joue un rôle essentiel dans le fonctionnement des réseaux prototypiques : il permet de transformer les images d'entrée en vecteurs de caractéristiques exploitables. Nous avons choisi d'utiliser **ResNet18**, un modèle léger mais suffisamment expressif pour extraire des représentations pertinentes même à partir d'un petit jeu de données thermique.

Les raisons de ce choix sont multiples :

— Architecture bien connue et largement utilisée en vision par ordinateur

— Nombre modéré de paramètres, adapté aux ressources limitées

— Bon compromis entre profondeur, performance et temps d'entraînement

— Facilité à l'adapter comme extracteur de caractéristiques

5.2.2 Résumé de l'architecture ResNet18

ResNet18 est un réseau de 18 couches profondes introduit dans le cadre des architectures à connexions résiduelles. Chaque bloc résiduel contient une série de convolutions et une *skip connection* qui permet de préserver les gradients pendant l'entraînement :

— Une couche de convolution initiale suivie d'un batch normalization et ReLU

— Quatre groupes de blocs résiduels (chaque groupe contenant 2 blocs, soit 8 blocs au total)

— Chaque *BasicBlock* contient deux convolutions 3×3 avec normalisation (BatchNorm) et activation ReLU

— Une couche de pooling global (moyenne adaptative)

— Une couche fully connected (remplacée dans notre cas par une tête d'encodage)

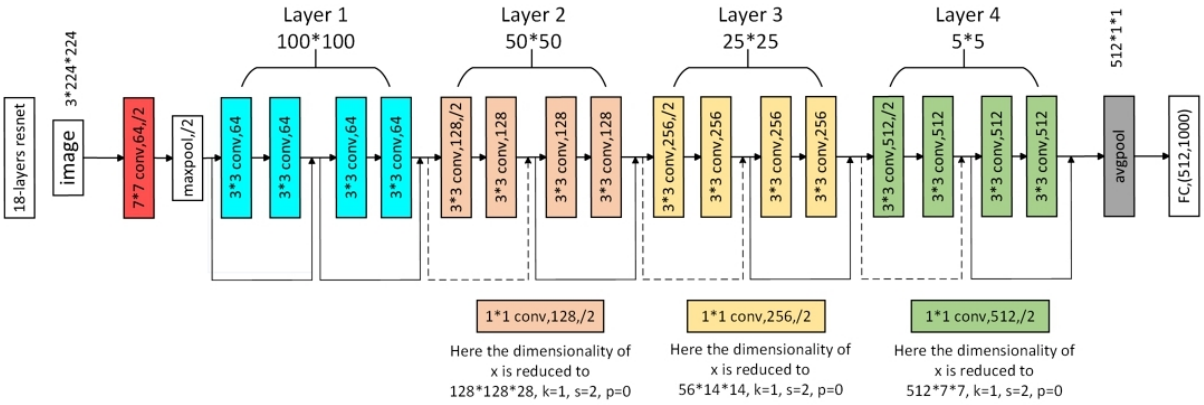

FIGURE 5.2 – Architecture simplifiée de ResNet18

5.2.3 Architecture du modèle utilisé

Le modèle que nous avons utilisé repose sur la structure de **ResNet18**, modifiée pour l'adapter à l'apprentissage par prototypes. L'architecture exacte de notre encodeur est présentée ci-dessous :

— Les premières couches de ResNet18 (jusqu'au bloc 6 inclus) sont complètement gelées.

— Seuls les deux derniers blocs résiduels (bloc 7) et la tête d'encodage sont entraînables.

— La couche fully connected d'origine est remplacée par une tête composée d'un `Dropout`, d'un `Linear` vers 128 dimensions, suivi d'une `BatchNorm1d`.

L'architecture détaillée, avec les dimensions d'entrée/sortie et le nombre de paramètres, est présentée ci-dessous :

```
----------------------------------------------------------------------------------------------------------------------------
Layer (type (var_name))                  Input Shape          Output Shape         Param #              Trainable
============================================================================================================================
Resnet18 (Resnet18)                      [32, 3, 128, 128]    [32, 128]            --                   Partial
├─Sequential (encoder)                   [32, 3, 128, 128]    [32, 512, 1, 1]      --                   Partial
│    └─Conv2d (0)                        [32, 3, 128, 128]    [32, 64, 64, 64]     (9,408)              False
│    └─BatchNorm2d (1)                   [32, 64, 64, 64]     [32, 64, 64, 64]     (128)                False
│    └─ReLU (2)                          [32, 64, 64, 64]     [32, 64, 64, 64]     --                   --
│    └─MaxPool2d (3)                     [32, 64, 64, 64]     [32, 64, 32, 32]     --                   --
│    └─Sequential (4)                    [32, 64, 32, 32]     [32, 64, 32, 32]     --                   False
│    │    └─BasicBlock (0)               [32, 64, 32, 32]     [32, 64, 32, 32]     (73,984)             False
│    │    └─BasicBlock (1)               [32, 64, 32, 32]     [32, 64, 32, 32]     (73,984)             False
│    └─Sequential (5)                    [32, 64, 32, 32]     [32, 128, 16, 16]    --                   False
│    │    └─BasicBlock (0)               [32, 64, 32, 32]     [32, 128, 16, 16]    (230,144)            False
│    │    └─BasicBlock (1)               [32, 128, 16, 16]    [32, 128, 16, 16]    (295,424)            False
│    └─Sequential (6)                    [32, 128, 16, 16]    [32, 256, 8, 8]      --                   False
│    │    └─BasicBlock (0)               [32, 128, 16, 16]    [32, 256, 8, 8]      (919,040)            False
│    │    └─BasicBlock (1)               [32, 256, 8, 8]      [32, 256, 8, 8]      (1,180,672)          False
│    └─Sequential (7)                    [32, 256, 8, 8]      [32, 512, 4, 4]      --                   Partial
│    │    └─BasicBlock (0)               [32, 256, 8, 8]      [32, 512, 4, 4]      (3,673,088)          False
│    │    └─BasicBlock (1)               [32, 512, 4, 4]      [32, 512, 4, 4]      4,720,640            Partial
│    └─AdaptiveAvgPool2d (8)             [32, 512, 4, 4]      [32, 512, 1, 1]      --                   --
├─Sequential (embedding)                 [32, 512]            [32, 128]            --                   True
│    └─Dropout (0)                       [32, 512]            [32, 512]            --                   --
│    └─Linear (1)                        [32, 512]            [32, 128]            65,664               True
│    └─BatchNorm1d (2)                   [32, 128]            [32, 128]            256                  True
============================================================================================================================
Total params: 11,242,432
Trainable params: 2,426,752
Non-trainable params: 8,815,680
Total mult-adds (Units.GIGABYTES): 18.95
============================================================================================================================
```

FIGURE 5.3 – Résumé de l'architecture du modèle avec gel partiel des couches

6 - Entraînement et Résultats / Expérimentations

6.1 Premières observations

Lors des premiers essais, notre modèle montrait des signes clairs de **surapprentissage**. Une différence de plus de 20% était observée entre la perte d'entraînement et celle de test, comme illustré dans la figure 6.1. Augmenter le nombre d'époques (5 à 10) ou la taille du vecteur d'embedding (128 à 256) n'a apporté que peu d'améliorations significatives.

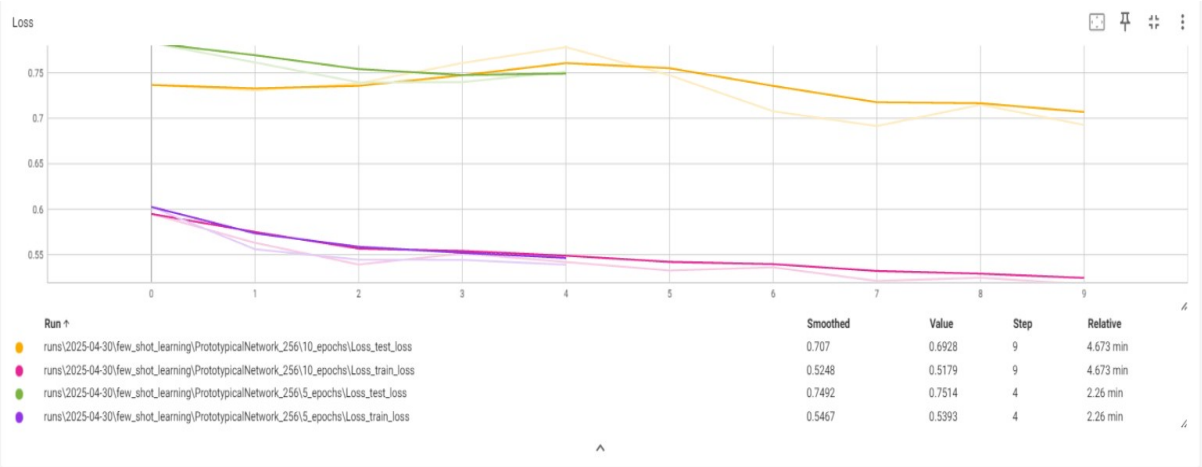

FIGURE 6.1 – Différence importante entre la perte d'entraînement et celle de test

6.2 Dégel partiel du backbone

Pour réduire le surapprentissage, nous avons décidé de **dégeler le dernier bloc résiduel** de ResNet18. Cela a permis au modèle d'apprendre des caractéristiques plus discriminantes et de mieux généraliser, ce qui s'est traduit par une perte test significativement plus faible et des courbes d'entraînement plus stables.

6.3 Variantes de configuration N, K, Q

Plusieurs combinaisons de N, K, Q ont été testées (par exemple $N = 3$, $K = 5$, $Q = 8$ vs $N = 5$, $K = 3$, $Q = 5$). Toutefois, ces variations n'ont pas montré d'impact significatif sur les performances du modèle, à l'exception d'un **ralentissement notable du temps d'entraînement**.

6.4 Optimiseur et planification du taux d'apprentissage

Nous avons comparé `Adam` et `AdamW`, ce dernier offrant de meilleurs résultats en test. Le planificateur de taux d'apprentissage initial était de type `StepLR`, mais le passage à un `CosineAnnealingLR` a considérablement amélioré la stabilité de l'entraînement.

6.5 Early stopping

Un `early stopping` avec une patience de 5 était trop sévère, surtout sur des entraînements de 100 époques, et arrêtait le modèle prématurément. En passant à une patience de 14, le modèle s'arrêtait en moyenne autour de l'époque 60, démontrant que l'augmentation du nombre d'époques ne garantit pas une meilleure performance.

6.6 Tentative avec MobileNetV2

Nous avons brièvement testé **MobileNetV2**, un choix cohérent puisque l'auteur du dataset l'avait utilisé. Cependant, les résultats se sont avérés très inférieurs à ceux de ResNet18 : la perte stagnait autour de **1.15** même après 50 époques, avec une précision souvent inférieure à **0.5**. Ce constat nous a convaincus de revenir à ResNet18.

6.7 Importance de l'augmentation des données

Notre pipeline initial d'augmentation était minimal (quelques rotations et zooms). Or, nos images sont très similaires entre classes, ce qui rend l'apprentissage difficile sans diversification.

Nous avons donc introduit une **augmentation de données plus agressive**, incluant :

- Transformations aléatoires variées via `TrivialAugmentWide`
- Flips horizontaux et verticaux
- Petites rotations et translations
- `ColorJitter` pour simuler différentes intensités thermiques
- `GaussianNoise`, `RandomErasing`, et inversion aléatoire de la signature thermique

Des exemplessur les nouveaux jeux de données augmentés sont présentés ci-dessous :

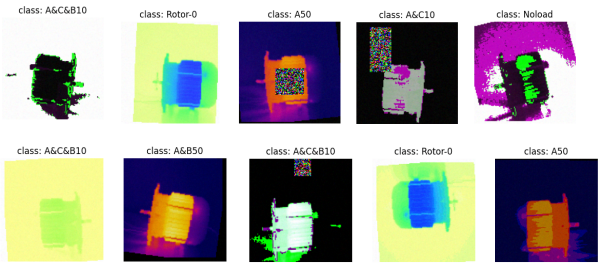

FIGURE 6.2 – Exemple de 10 images augmentés thermiques aléatoires issues du dataset

Cette nouvelle stratégie d'augmentation a joué un rôle clé dans la réduction du surapprentissage : la différence entre la perte d'entraînement et celle de test est devenue négligeable.

6.8 Résultats finaux

Après toutes les modifications apportées à notre pipeline, notre modèle final a atteint des performances **stables et généralisables**, comme le montrent les figures ci-dessous.

Dans un premier temps, nous avons entraîné le modèle pendant 50 époques. Il a atteint une perte d'entraînement de **0.60** et une perte de test de **0.67**, avec une précision de test atteignant **83%**. Ces résultats sont encourageants et indiquent une généralisation correcte sans surapprentissage excessif.

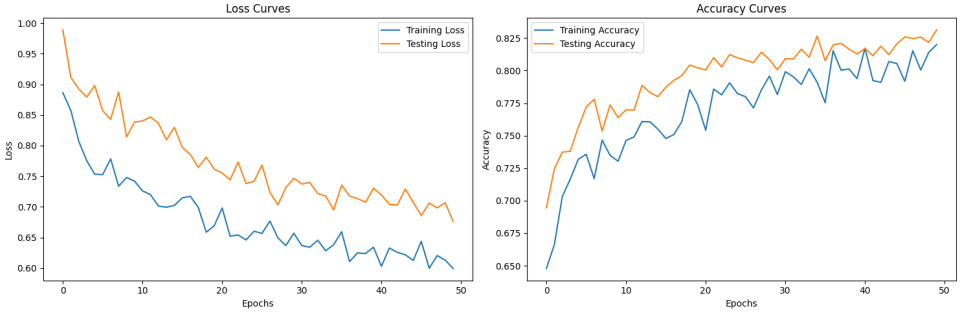

FIGURE 6.3 – Résultats après 50 époques — perte entraînement : 0.60, perte test : 0.67, précision test : 83%

Par la suite, nous avons relancé l'entraînement avec une configuration de 100 époques, en activant le `early stopping` avec une patience de 14. Ce mécanisme a arrêté l'entraînement automatiquement autour de l'époque **60**, ayant détecté un plateau dans l'amélioration de la validation. Les performances finales étaient très proches de celles obtenues à 50 époques, confirmant que l'augmentation du nombre d'époques **n'améliore pas nécessairement les résultats**.

FIGURE 6.4 – Entraînement interrompu par early stopping à l'époque 60 — performances comparables à 50 époques

À ce stade, j'ai appris que les modèles peuvent être **rechargés depuis leur dernier checkpoint** pour reprendre l'entraînement. En exploitant cette fonctionnalité, j'ai continué l'entraînement pendant **35 époques supplémentaires**, portant le total à environ 95

époques effectives. Le graphique ci-dessous montre une lente mais régulière diminution de la perte, suggérant un potentiel d'amélioration encore présent avec plus d'entraînement.

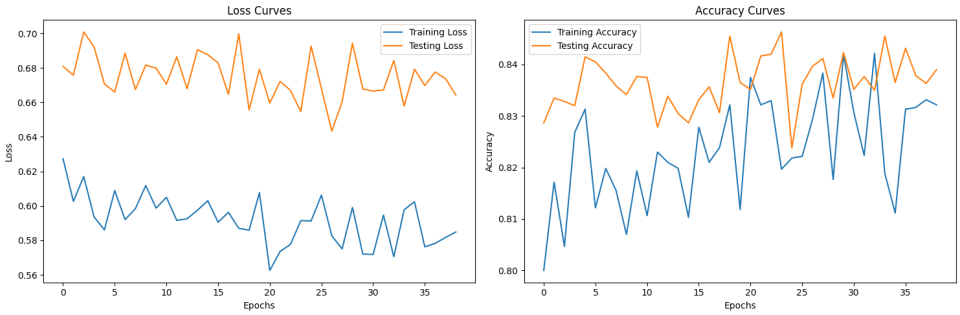

FIGURE 6.5 – Reprise d'entraînement depuis le checkpoint — amélioration progressive au-delà de 60 époques

Grâce à ces ajustements progressifs, il s'agit du modèle le plus performant obtenu au cours du projet.

6.9 Test du modèle en ligne (Colab)

Pour faciliter l'évaluation et la reproductibilité, nous avons mis à disposition un notebook Google Colab permettant d'exécuter l'inférence sur de nouvelles images. Aucune installation locale de PyTorch ou dépendance complexe n'est requise : tout peut être exécuté dans le cloud avec un simple navigateur.

- Il suffit de copier deux fichiers nécessaires dans votre Google Drive :
  - `Resnet18_RetrainedV2.pth` : les poids du modèle final
  - `support_embeddings.pt` : les embeddings de référence générés à partir des images du support
- Utilisez ensuite les commandes suivantes dans le notebook pour les charger :

```
!cp "/content/drive/MyDrive/FSL/Resnet18_RetrainedV2.pth" .
!cp "/content/drive/MyDrive/FSL/support_embeddings.pt" .
```

- Le notebook gère le reste : chargement des données, inférence, visualisation du résultat.

Vous pouvez accéder au notebook ici :
https://colab.research.google.com/drive/1MRQpNeDNyf9UlkfFFWMt2FKMREf9MGxe?usp=sharing

7 - Conclusion et perspectives

Ce projet a représenté notre première réelle opportunité de mettre en pratique les connaissances acquises ces derniers mois dans le domaine de l'apprentissage automatique et de l'apprentissage profond. Jusqu'à présent, notre parcours s'était surtout limité à la lecture de livres, d'articles et de guides théoriques. Ce travail nous a permis de passer de la théorie à la pratique, en explorant un cas concret de classification d'images thermiques en contexte de Few-Shot Learning.

Au cours de ce projet, nous avons découvert les spécificités de la méthode des réseaux prototypiques, construit un pipeline d'entraînement, analysé les performances du modèle, et expérimenté différentes stratégies pour améliorer sa généralisation. L'un des points les plus marquants a été de constater à quel point la qualité des données et les choix d'augmentation influencent les résultats, surtout dans des contextes où les données sont visuellement très similaires.

À l'avenir, deux directions principales sont envisageables. D'une part, nous pourrions chercher à améliorer encore les performances du modèle actuel, ce que nous n'avons pas pu faire entièrement en raison des limitations matérielles : notre machine locale présente des contraintes de refroidissement (température élevée du CPU lors de l'entraînement), ce qui ralentit considérablement les expérimentations. Nous prévoyons de remédier à cela en investissant dans un boîtier à meilleur flux d'air, ce qui permettrait d'accélérer les futurs entraînements.

D'autre part, une alternative serait de nous tourner vers des solutions cloud comme les notebooks Kaggle ou Google Colab Pro, qui offrent une puissance de calcul GPU sans contrainte matérielle locale. Cela ouvrirait la voie à des entraînements plus longs, plus complexes, voire à l'exploration de modèles plus profonds ou d'approches génératives.

Enfin, ce projet nous a aussi sensibilisés à l'importance d'une bonne organisation du code, à la reproductibilité via les notebooks partagés, et à la possibilité de proposer des modèles testables facilement par d'autres via des outils comme Google Colab.

Ce rapport ne constitue pas une fin, mais plutôt un point de départ vers des projets plus ambitieux en vision par ordinateur, en optimisation de modèles, et en apprentissage sur petits jeux de données.