Classification non linéaire grâce à l'IA avancée

Rapport IA02

GELBART Simon - Romain GOLDENCHTEIN

utt
UNIVERSITÉ DE TECHNOLOGIE
TROYES

1. Introduction

Dans ce projet réalisé, dans le cadre d'IA02, on s'est intéressé à deux problématiques bien différentes, mais complémentaires, avec un objectif commun : concevoir des modèles d'intelligence artificielle capables de classer efficacement des données complexes, en particulier des images ou des signaux que l'on peut représenter visuellement.

La première partie se concentrera sur le dataset CIFAR-10, une base d'images contenant dix classes visuelles. Cette première partie constitue un bon point de départ pour comparer plusieurs approches : des algorithmes classiques de machine learning, des réseaux convolutifs de différentes architectures, et un modèle hybride. L'idée ici est de bien comprendre les forces et limites de chaque méthode sur un problème de vision par ordinateur standardisé.

Pour la deuxième partie, notre choix s'est porté sur un cas d'usage plus concret : la détection de chutes à partir. Même si ce ne sont pas des images à proprement parler, les signaux sont organisés sous forme de matrices, ce qui nous permet de les traiter avec des modèles inspirés de la vision, comme les CNN. Ce dataset est particulièrement intéressant, car il est facile d'imaginer une application réelle et donc à évaluer les enjeux que doit couvrir notre modèle.

Ce rapport présente les différentes étapes de notre démarche : l'analyse des données, le choix et le réglage des algorithmes, les performances obtenues, mais aussi les pistes d'amélioration, notamment via des techniques d'augmentation des données.

2. Partie 1 – Classification sur CIFAR-10

2.1 Présentation et préparation du dataset CIFAR-10

Ce jeu de données contient 60 000 images couleur de taille 32x32 pixels, réparties équitablement en 10 catégories visuelles : avion, voiture, oiseau, chat, cerf, chien, grenouille, cheval, bateau et camion. Ces classes sont mutuellement exclusives, ce qui signifie qu'aucune image ne peut appartenir à deux catégories à la fois.

Le dataset est divisé en deux parties : 50 000 images pour l'entraînement, et 10 000 pour le test. Les labels associés sont des entiers entre 0 et 9, ce qui nous permet de faire la distinction entre les différentes images. Malgré la faible résolution des images, ce dataset représente un bon compromis entre simplicité d'utilisation et richesse des cas à traiter.

Représentation des images du dataset CIFAR-10 sur Python

Avant d'entraîner nos modèles, nous avons procédé à une phase de prétraitement des données du dataset CIFAR-10. Dans un premier temps, nous avons séparé les données d'entraînement en deux sous-ensembles distincts : un jeu d'entraînement (80 %) et un jeu de validation (20 %), ce qui donne donc 40 000 images pour le train et 10 000 images pour la validation. Cette étape est essentielle pour les CNN que l'on utilisera plus tard.

```
#Split le data set en train et en validation (pour les CNN)

N_train_t = np.shape(x_train_t)[0]
N_train = round(0.8*N_train_t) # 80% pour le train et 20% pour la validation
x_train,y_train = x_train_t[0:N_train],y_train_t[0:N_train]
x_val,y_val = x_train_t[N_train:],y_train_t[N_train:]
```

Ensuite, les images ont été normalisées en divisant les valeurs des pixels par 255, afin d'avoir des valeurs entre 0 et 1. Cette étape facilite l'entraînement des modèles en stabilisant les gradients et en accélérant la convergence

Pour adapter les images à l'entrée des CNN, nous avons redimensionné les tenseurs d'entrée au format attendu par les couches convolutives : (32, 32, 3) correspondant à des images RGB (3 canaux). Puis enfin, les étiquettes sont encodées en one-hot vectors pour la classification multiclasse.

```
# Redimensionner les images (32x32 -> 32x32x1) pour la convolution
x_train = x_train.reshape(-1, 32, 32, 3)
x_val = x_val.reshape(-1, 32, 32, 3)
x_test = x_test.reshape(-1, 32, 32, 3)
# Encodage One-Hot des labels
y_train_c = to_categorical(y_train, 10)
y_val_c = to_categorical(y_val, 10)
y_test_c = to_categorical(y_test, 10)
```

2.2 Classification avec modèles de ML

Avant d'aborder les méthodes plus avancées basées sur les CNN, nous avons commencé par appliquer des algorithmes classiques de machine learning sur le dataset CIFAR-10. L'objectif était de mesurer la performance de modèles simples sur ce jeu de données pour avoir une base de comparaison.

Préparation des données

Les images, initialement de dimension (32, 32, 3), ont été aplaties en vecteurs à une dimension afin de pouvoir être utilisées par les algorithmes classiques de ML. On a également ajouté un StandardScaler pour normaliser les données et que ce soit plus adapté aux modèles de ML.

Modèle 1 : Arbre de décision

Nous avons fait le choix d'entraîner un arbre de décision pour le 1er algorithme de ML. Dans le but d'optimiser les hyperparamètres, nous avons réalisé un GridSearch, testant différentes profondeurs, critères de division (gini ou entropy) et tailles minimales de splits. Nous avons donc choisi en fonction des résultats de GridSearch, une profondeur max de 10, une utilisation de gini et 20 de taille minimales de split.

Résultats obtenus :

- Précision : 10%
- Rappel macro : 10%

Modèle 2 : Forêt aléatoire

Pour améliorer la robustesse et la généralisation, une forêt aléatoire composée de 100 arbres a été entraînée sur les mêmes données normalisées. Cette méthode permet de réduire le surapprentissage souvent rencontré avec un seul arbre et améliore la performance globale.

Résultats obtenus :

- Précision : 10%
- Rappel : 10%

Ces résultats montrent que les algorithmes classiques de ML ne sont pas du tout adaptés pour la classification d'images complexes comme celles de CIFAR-10, justifiant l'existence de méthodes plus sophistiquées, notamment les CNN, capables de mieux exploiter la structure spatiale et les caractéristiques visuelles des images. Les performances limitées de ces modèles s'expliquent notamment par la perte de l'information spatiale cruciale lorsque l'on aplatit les images en vecteurs, ainsi que par leur incapacité à extraire automatiquement des caractéristiques visuelles pertinentes, telles que les bords, textures ou formes, à partir des pixels bruts.

2.3 Classification avec CNN

Introduction

Les CNNs sont particulièrement adaptés à la classification d'images, car ils exploitent la structure spatiale des données visuelles. Contrairement aux algorithmes classiques qui nécessitent d'aplatir les images en vecteurs, les CNN conservent les relations locales entre pixels grâce aux filtres convolutifs, ce qui permet d'extraire des caractéristiques visuelles pertinentes ce que les algos de ML ne prennent pas en compte par exemple.

Optimisation des hyperparamètres

Afin d'obtenir de bonnes performances tout en conservant un temps d'entraînement raisonnable, nous avons utilisé Optuna, une librairie d'optimisation pour les hyperparamètres des réseaux neuronaux, pour rechercher les valeurs optimales de certains hyper paramètres

Les meilleurs paramètres trouvés ont ensuite été réutilisés pour tous les modèles CNN, afin de garantir des conditions d'entraînement équitables et comparables.

```
Optimisation terminée.
Meilleurs hyperparamètres trouvés pour le CNN classique:  {'epochs': 22, 'batch_size': 64,
Meilleure val_accuracy obtenue :  0.7092999815940857
```

Nous avons finalement obtenu

- 22 d'époques d'entraînement (epochs évalués entre 10 et 30)
- 64 de taille du batch (batch_size évalués parmi 32, 64, 128)

Nous avons également obtenu que le meilleur learning rate était de 0.01, mais il est propre à ce modèle et est compliqué à généraliser.

Modèle 1 : CNN classique

Nous avons d'abord utilisé une architecture simple, similaire à celle vue en TD. Elle est composée de deux blocs convolution + max pooling, suivis d'un flatten et d'une couche dense :

- Convolution 2D (32 filtres, 3x3) + ReLU
- MaxPooling (2x2)
- Convolution 2D (64 filtres, 3x3) + ReLU
- MaxPooling (2x2)
- Flatten
- Dense (128 neurones) + ReLU
- Dense (10 neurones) + Softmax

Le modèle a été compilé avec l'optimiseur Adam et entraîné avec une stratégie d'early stopping sur la perte de validation pour éviter le surapprentissage.

Résultats obtenus :

```
Évaluation du modèle CNN Classique en cours...
Évaluation de : CNN Classique (model)


Tableau Récapitulatif des Performances (CNN Classique):

| Modèle                | Set        |   Loss |   Accuracy |   Recall |
|:----------------------|:-----------|-------:|-----------:|---------:|
| CNN Classique (model) | Train      | 0.4982 |     0.8347 |   0.7674 |
| CNN Classique (model) | Validation | 0.8714 |     0.7086 |   0.6434 |
| CNN Classique (model) | Test       | 0.8893 |     0.7042 |   0.6395 |
```

On remarque un léger overfitting dû premièrement au fait qu'il n'y ait pas de dropout et à la difficulté qu'à un modèle aussi simple à généraliser. L'accuracy reste cependant assez satisfaisante

Modèle 2 : CNN plus complexe

Pour ce second modèle, nous avons cherché à augmenter la capacité d'apprentissage du réseau tout en limitant le surapprentissage. Pour cela, nous avons ajouté une troisième série de couches convolutionnelles, ainsi que des couches de Batch Normalization et de Dropout à différents niveaux du réseau.

L'architecture est la suivante :

- Trois blocs convolutionnels successifs, avec un nombre de filtres croissant (32, 64, 128). Chaque bloc comprend deux couches de convolution avec activation ReLU, suivies d'une normalisation par lot, d'un MaxPooling pour la réduction de dimension, et d'un Dropout pour la régularisation.

- Un flattening (la sortie des blocs est aplatie (Flatten)), puis passée dans une couche dense de 256 neurones avec activation ReLU et régularisation L2, suivie d'un Dropout supplémentaire pour limiter le surapprentissage ; suivi d'une couche dense de 512 neurones, à nouveau normalisée et suivie d'un dropout.

- Enfin, une couche de sortie à 10 neurones avec une activation softmax.

Cette version permet une meilleure extraction des caractéristiques tout en réduisant le risque de surapprentissage.

Résultats obtenus :

```
Évaluation du modèle CNN Classique en cours...
Évaluation de : CNN Intermédiaire


Tableau Récapitulatif des Performances (CNN Intermédiaire):

| Modèle           | Set        |   Loss |   Accuracy |   Recall |
|:-----------------|:-----------|-------:|-----------:|---------:|
| CNN Intermédiaire | Train      | 0.4219 |     0.9049 |   0.8758 |
| CNN Intermédiaire | Validation | 0.7208 |     0.8181 |   0.7848 |
| CNN Intermédiaire | Test       | 0.7549 |     0.8124 |   0.7773 |
```

On remarque que le modèle est beaucoup moins sujet à l'overfitting et a une accuracy très élevéé.

Modèle 3 : Architecture inspirée de MCDNN

Enfin, une architecture plus coûteuse inspirée du MCDNN (Multi-column Deep Neural Networks) a été testée. Cette architecture, comme son nom l'indique, créée des "colonnes" de réseaux de neurones et fait la moyenne de leur résultat. Nous avons simplement pris

l'architecture donnée dans l'article : 3x32x32-300C3-MP2-300C2-MP2-300C3-MP2-300C2-MP2-300N-100N-10N.

Cela se résume donc à :

- Une entréer de dimension 3x32x32
- 300C3 : Une couche convolutive avec filtre 3x3 de 300 filtres
- MP2 : Du MaxPooling 2D
- 300C2 : Une couche convolutive avec filtre 2x2 de 300 filtres
- MP2 : Du MaxPooling 2D
- 300C3 : Une couche convolutive avec filtre 3x3 de 300 filtres
- MP2 : Du MaxPooling 2D
- 300C2 : Une couche convolutive avec filtre 2x2 de 300 filtres
- MP2 : Du MaxPooling 2D
- 300N : Une couche dense de 300 neurones,
- 100N : Une de 100 neurones
- 10N : Et enfin la couche de sortie de 10 neurones

Cette architecture ne représente qu'une des 8 colonnes que l'on a créées, ce qui la rend d'une complexité démesurée. Le papier de recherche cite 800 époques, ce qui était impossible pour nos machines. Nous nous sommes contentés de 30, mais voulions être les plus fidèles possibles sur l'architecture.

Résultats obtenus :

Tableau Récapitulatif des Performances (MCDNN):

Modèle	Set	Loss	Accuracy	Recall
MCDNN	Train	0.1178	0.9883	0.975
MCDNN	Validation	0.6285	0.8285	0.7716
MCDNN	Test	0.6593	0.8266	0.7703

Le modèle basé sur MCDNN a des résultats bien meilleur que le 1er concurrent CNN avec une accuracy de presque 83% pour la validation et le test ce qui est très satisfaisant. Dans l'article originel, ils obtiennent 88%. Vu la diminution d'époques, c'est satisfaisant. Nous observons de l'overfitting, malgré notre ajout de BatchNormalization et Dropout. L'article ne précise pas exactement la régularisation faîte et nous pensions que cela serait suffisant, mais vu le temps de calcul, nous n'avons pas réessayé.

Ainsi, bien que l'approche soit intéressante, le coût de calcul ne vaut pas les quelques pourcents d'accuracy en plus par rapport au modèle intermédiaire. En effet, l'entrainement du modèle sur 30 époques a pris environ 8h. Le prochain fait mieux, et plus vite.

2.4 L’option CNN Hybride

Dans cette approche, nous avons développé un modèle hybride combinant les avantages du deep learning et des méthodes de machine learning plus traditionnelles. L’objectif principal était de tirer parti de la puissance d’un réseau de neurones convolutif pré-entraîné pour l’extraction automatique de caractéristiques visuelles, tout en conservant une architecture légère et rapide à entraîner pour la phase de classification.

Nous avons ainsi utilisé MobileNetV2, un modèle léger et performant, pré-entraîné sur ImageNet. Dans un premier temps, toutes les couches de MobileNetV2 ont été gelées afin de conserver les poids appris et de limiter les besoins en calcul lors de l'entraînement initial. À la suite de cette extraction de features, nous avons ajouté une tête de classification composée :

- d’une couche dense (avec Dropout),
- du Batch Normalization pour stabiliser l'entraînement,
- la couche de sortie pour la classification

Cette configuration a été entraînée pendant 15 époques dans un premier temps.

Afin d’améliorer encore les performances du modèle et de se familiariser avec de nouveaux concepts, nous avons pris l’initiative de faire du fine-tuning. Nous avons donc ensuite dégelé les 20 dernières couches de MobileNetV2. Cela permet au modèle de s’adapter plus finement à notre jeu de données, tout en conservant l'essentiel des représentations apprises avant. Un second entraînement a alors été réalisé sur 30 époques supplémentaires.

Ce processus en deux temps (entraînement initial avec couches gelées, puis fine-tuning partiel) permet de bénéficier d’une bonne généralisation tout en s’ajustant à CIFAR10.

Résultats obtenus :

Tableau Récapitulatif des Performances (MobileNetv2 Hybride):

Modèle	Set	Loss	Accuracy	Recall
MobileNetv2_Fine-tuné	Train	0.1066	0.9661	0.9589
MobileNetv2_Fine-tuné	Validation	0.4073	0.8667	0.8508
MobileNetv2_Fine-tuné	Test	0.4215	0.865	0.8498

On observe un peu d’overffiting, mais les résultats restent très bons, surtout considérant le temps d’entrainement qui a pris moins de 25 minutes.

2.5 Analyse des performances

L'analyse globale des différents modèles testés met en évidence une nette supériorité des approches basées sur les réseaux de neurones convolutifs (CNN) par rapport aux algorithmes classiques de machine learning. Les arbres de décision et les forêts aléatoires, bien que simples et rapides à entraîner, peinent à dépasser 10 % de précision, ce qui équivaut au hasard dans un problème à 10 classes. Cette faible performance s'explique notamment par la perte d'informations spatiales lors de l'aplatissement des images, rendant ces modèles inadaptés à la nature visuelle du dataset CIFAR-10.

À l'inverse, les CNN, conçus pour ça, offrent de bien meilleurs résultats. L'architecture simple obtient déjà des performances satisfaisantes, mais reste sensible à un léger surapprentissage. Le CNN plus complexe introduit des mécanismes de régularisation (Dropout, Batch Normalization) qui améliorent la robustesse et la généralisation. Le modèle inspiré du MCDNN atteint quant à lui des performances très élevées (jusqu'à 82 % de précision), au prix d'un coût de calcul plus important.

Enfin, le modèle hybride, combinant plusieurs approches, permet d'obtenir d'excellents résultats, en tirant parti des forces de chaque méthode et un temps de calcul relativement faible grâce aux couches pré-entrainées.

3. Partie 2 – Classification pour la détection de chutes

3.1 Présentation et préparation du dataset

Le second jeu de données est dédié à la détection de chute à partir de signaux électromagnétiques multidimensionnels captés dans différents environnements. Il contient 757 échantillons de signaux, dont 321 correspondent à des chutes, et les autres à de diverses activités. Les signaux sont issus de 22 participants et de 4 environnements différents, permettant d'étudier la robustesse des modèles en présence de variations inter-individuelles et contextuelles.

L'objectif est double :

- Phase 1 : entraîner les modèles avec l'ensemble des participants et environnements, puis tester sur des signaux non vus (mais de participants/environnements déjà connus).

- Phase 2 : tester les modèles sur deux environnements et participants totalement inconnus, simulant une généralisation à des situations nouvelles.

Le jeu de données utilisé pour la détection de chute est issu de l'ensemble ENetFall, qui regroupe des signaux représentant les variations du champ généré par le mouvement des personnes. Ces données ont été collectées dans quatre environnements distincts : salon, salle de conférence, salle de cours et domicile (côté gauche et droit).

Chaque fichier .mat contient deux éléments :

- dataset_CSI_t : un tenseur représentant les signaux temporels multi-capteurs,
- dataset_labels : un vecteur binaire associé à chaque signal (1 = chute, 0 = non-chute).

Nous avons chargé les cinq fichiers fournis :

- dataset_home_lab(L).mat et dataset_home_lab(R).mat
- dataset_lecture_room.mat
- dataset_living_room.mat
- dataset_meeting_room.mat

Les signaux et leurs étiquettes respectives ont été extraits, puis concaténés pour former un jeu de données global.

On obtient 757 échantillons comme prévu.

```
Chargé: dataset_home_lab(L).mat, Signaux: (85, 625, 90), Étiquettes: (85,)
Chargé: dataset_home_lab(R).mat, Signaux: (85, 625, 90), Étiquettes: (85,)
Chargé: dataset_lecture_room.mat, Signaux: (205, 625, 90), Étiquettes: (205,)
Chargé: dataset_living_room.mat, Signaux: (248, 625, 90), Étiquettes: (248,)
Chargé: dataset_meeting_room.mat, Signaux: (134, 625, 90), Étiquettes: (134,)

Données totales - signaux: (757, 625, 90), étiquettes: (757,)
```

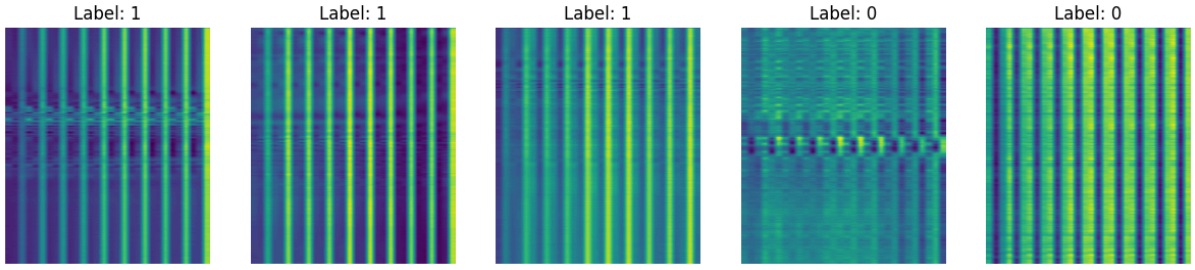
Il faut maintenant rendre les données compatibles avec nos modèles, il faut donc réorganiser les signaux bruts. De base, chaque échantillon est un tenseur de dimension (N, 625, 90), représentant une séquence temporelle de 625 pas de temps, avec 90 variables à chaque instant. Ces 90 variables peuvent correspondre à différentes mesures provenant de plusieurs capteurs.

Il faut donc restructurer chaque vecteur de 90 dimensions en une image 2D de forme (30, 3) pour être compatible avec nos couches de convolution notamment. Ainsi, la nouvelle forme finale des données devient (N, 625, 30, 3), où :

- N est le nombre total d’échantillons,
- 625 est la dimension temporelle ,
- 30 et 3 forment la représentation spatiale du vecteur de 90 valeurs par pas de temps.

```
✓ 0.5s
Shape train : (375, 625, 30, 3)
Shape test : (382, 625, 30, 3)
```

Enfin, les données ont été divisées en un ensemble d'entraînement (75 %) et un ensemble de test (25 %) en conservant la proportion chute / non-chute. Nous avons fait le choix de faire la validation de nos modèles sur l’ensemble de test, car nous avons trop peu de données pour se permettre de faire un ensemble de validation.



Représentation des images du dataset ENetFall sur Python

Cette représentation visuelle n’a pas de valeur d’image au sens traditionnel, mais elle permet d’appréhender la distribution des données. Bien que la visualisation ne soit pas nécessaire pour le traitement, elle aide à conceptualiser comment les réseaux de neurones convolutionnels peuvent extraire des motifs locaux.

Les données ont été normalisées en divisant par la valeur maximale de l’ensemble d’entraînement, puis centrées en retirant la moyenne. Cela permet de stabiliser et d’accélérer l’apprentissage du réseau. Ensuite, les ensembles d'entraînement et de test ont été transformés en objets tf.data.Dataset, permettant une gestion efficace des lots

On entraînera nos modèles sur 30 époques (avec de l’early stopping) et un batch size de 32.

3.2 Présentation des modèles

Modèle 1 : CNN simple

Le premier modèle est un réseau convolutionnel simple conçu pour extraire des caractéristiques pertinentes des signaux.

- Trois couches convolutionnelles avec filtres 3x3 (32, 64, 64), chacune avec activation ReLU, et deux couches MaxPooling 2x2 pour réduire la dimension spatiale.

- Sortie aplatie (Flatten), suivie d'une couche dense de 64 neurones avec activation ReLU et d'un Dropout à 50 % pour limiter le surapprentissage.

- Couche de sortie à un neurone avec d'activation sigmoïde, adaptée à la classification binaire chute / non-chute.

Résultats obtenus :

```
Évaluation de : CNN simple


Tableau Récapitulatif des Performances du Modèle 1:

| Modèle     | Set   |   Loss |   Accuracy |   Precision |   Recall |
|:-----------|:------|-------:|-----------:|------------:|---------:|
| CNN simple | Train | 0.3035 |     0.8571 |      0.8412 |   0.8167 |
| CNN simple | Test  | 0.3936 |     0.8316 |      0.7952 |   0.8148 |
```

Le modèle simple affiche déjà une accuracy, une loss et une précision très satisfaisante

Modèle 2 : Modèle hybride avec MobileNetV2 pré-entraîné

Pour ce deuxième modèle, nous avons opté pour une approche hybride combinant un réseau MobileNetV2 pré-entraîné sur ImageNet, utilisé comme extracteur de caractéristiques, avec une couche dense classique en sortie adaptée à la classification binaire.

- Les signaux ont d'abord été redimensionnés en images 64x64 pixels

- Le modèle MobileNetV2 est utilisé avec ses poids gelés afin de préserver les connaissances acquises lors de l'entraînement sur ImageNet et d'éviter un surapprentissage sur notre petit dataset.

- La sortie de MobileNetV2 est globalement moyennée grâce à une couche GlobalAveragePooling2D, puis alimentée dans une couche dense de 64 neurones avec activation ReLU.

- Un dropout de 50 % est ajouté pour régulariser et limiter le surapprentissage.

- Enfin, une couche dense finale avec activation sigmoid produit la probabilité d'appartenance à la classe chute.

Nous avons testé plusieurs architectures de modèles pré-entraînés, notamment VGG16 par exemple. Parmi eux, MobileNetV2 nous a permis d'obtenir de bien meilleurs résultats en termes de performance. Sur ce problème de classification, nous avons vu que EfficientNet était le plus utilisé (d'où le nom ENetFall). Nous avons donc choisi MobileNetV2 pour apporter une approche différente et originale dans notre étude et ne pas copier coller simplement du code trouvable en ligne.

Résultats obtenus :

Tableau Récapitulatif des Performances du Modèle Hybride:

Modèle	Set	Loss	Accuracy	Precision	Recall
Hybrid MobileNetV2	Train	0.1637	0.9347	0.8919	0.9625
Hybrid MobileNetV2	Test	0.3558	0.8684	0.85	0.8395

Les résultats obtenus sont très satisfaisants et meilleurs que pour le modèle simple.

Modèle 3 : CNN-LSTM

Pour capturer à la fois les caractéristiques spatiales et temporelles des signaux, nous avons essayé de développer un modèle hybride combinant CNN et LSTM.

Le modèle est structuré en deux grandes parties :

- Extraction des caractéristiques spatiales avec deux blocs convolutionnels 2D successifs. Chaque bloc comprend une couche convolutionnelle avec activation ReLU, suivie d'un Batch Normalization, d'une couche de max-pooling 2D pour réduire la dimension spatiale, et d'un dropout (0.25) afin de limiter le surapprentissage.

- Modélisation temporelle avec 2 couches LSTM bidirectionnelles.
  La première couche LSTM avec return_sequences=True pour retourner la séquence complète, facilitant ainsi l'apprentissage des dépendances temporelles complexes.
  La deuxième couche LSTM avec return_sequences=False, qui condense la

séquence en un vecteur d'état final.
Entre ces deux couches et après la dernière, des couches de normalisation et de dropout sont ajoutées pour stabiliser et régulariser l'entraînement.

La sortie finale est une couche dense avec une activation sigmoïde, adaptée à la classification binaire chute/non-chute. Pour ce modèle, nous avons augmenté le nombre d'époques à 50 (toujours avec de l'early stopping).

Résultats obtenus :

```
Évaluation détaillée de : CNN-LSTM
  Shape des données d'entraînement normalisées pour évaluation : (567, 625, 30, 3), (567,)
  Train - Loss: 0.2519, Accuracy: 0.9083, Precision: 0.9052, Recall: 0.8750
  Shape des données test normalisées pour évaluation : (190, 625, 30, 3), (190,)
  Test - Loss: 0.4086, Accuracy: 0.8000, Precision: 0.7654, Recall: 0.7654

Tableau récapitulatif des performances du CNN-LSTM :

| Modèle   | Set   |  Loss |  Accuracy |  Precision |  Recall |
|:---------|:------|------:|----------:|-----------:|--------:|
| CNN-LSTM | Train | 0.2519 |    0.9083 |     0.9052 |  0.875  |
| CNN-LSTM | Test  | 0.4086 |    0.8    |     0.7654 |  0.7654 |
```

Malgré cette architecture plus sophistiquée, les performances obtenues sont un peu inférieures aux modèles CNN plus simples. Sur le jeu de test, l'accuracy atteint environ 80%, ce qui reste en dessous des résultats observés avec les autres architectures.

Cela peut s'expliquer par plusieurs facteurs :

- la complexité du modèle LSTM, plus difficile à optimiser avec un jeu de données de taille limitée,
- un encodage temporel peu pertinent dans ce cas précis si les séquences temporelles sont peu discriminantes,
- un besoin d'ajustement plus fin des hyperparamètres (nombre d'unités LSTM, régularisation, etc.).

3.3 Partie 2

Dans cette partie, nous simulons un scénario plus réaliste où le modèle doit généraliser à des environnements et des participants qu'il n'a jamais vus pendant l'apprentissage.

- Environnements d'entraînement : env1 et env2
- Environnements de test : env3 et env4

Pour renforcer la robustesse du modèle face à cette configuration difficile, nous appliquons des techniques d'augmentation de données sur l'ensemble d'entraînement.

Préparation de données

On associe chaque fichier à un environnement (env1 à env4) et sépare les données en deux groupes : un ensemble d'entraînement (env1, env2) et un ensemble de test sur des environnements inconnus (env3, env4). Les signaux et leurs étiquettes sont ensuite concaténés pour former les jeux X_train, y_train, X_test, y_test.

```
Début du chargement des données pour la Partie 2...
Chargé: dataset_home_lab(L).mat (Env: env1), Signaux: (85, 625, 90), Étiquettes: (85,)
Chargé: dataset_home_lab(R).mat (Env: env1), Signaux: (85, 625, 90), Étiquettes: (85,)
Chargé: dataset_lecture_room.mat (Env: env2), Signaux: (205, 625, 90), Étiquettes: (205,)
Chargé: dataset_living_room.mat (Env: env3), Signaux: (248, 625, 90), Étiquettes: (248,)
Chargé: dataset_meeting_room.mat (Env: env4), Signaux: (134, 625, 90), Étiquettes: (134,)
Partie 2 - Données d'entraînement: (375, 625, 90), Étiquettes d'entraînement: (375,)
Partie 2 - Données de test: (382, 625, 90), Étiquettes de test: (382,)
Échantillons 'chute' dans l'entraînement (Partie 2): 172
Échantillons 'chute' dans le test (Partie 2): 149
```

On applique le même prétraitement que dans la phase 1, les signaux bruts sont resize pour les traiter comme des images puis une normalisation est ensuite appliquée à partir des données d'entraînement pour faciliter l'apprentissage du modèle. Voilà la shape de nos ensemble données de train et de test finales :

```
Shape train : (375, 625, 30, 3)
Shape test : (382, 625, 30, 3)
```

Augmentation de données

Pour améliorer la robustesse et du fait que nous avons très peu de données, nous avons mis en place une stratégie d'augmentation de données dynamique. On applique directement sur les données d'entraînement lors de la construction du pipeline TensorFlow tf.data.Dataset.

On ne peut pas simplement faire une augmentation de données classique comme ce que l'on pourrait faire avec des images. Nous avons donc fait une fonction augment qui sert à transformer les signaux à chaque époque. Elle applique trois modifications simples, mais pertinentes dans notre cas de données temporelles :

- l'ajout d'un bruit gaussien léger pour simuler des perturbations,
- un décalage temporel aléatoire pour varier la position des événements,

- un masquage aléatoire de points (dropout temporel) pour éviter que le modèle ne dépende trop de motifs spécifiques.

On utilise map() et donc nos augmentations sont effectuées dynamiquement, sans rajouter des données au dataset en mémoire.

Enfin, les données sont organisées sous forme de train_ds et test_ds, prêtes à être utilisées pour l'entraînement et l'évaluation du modèle.

Voici les résultats d'un entraînement simple d'un CNN sur 15 epochs, on peut voir que l'augmentation apporte une hausse globale de toutes nos métriques.

```
...
| Type d'apprentissage   |   Loss |   Accuracy |   Precision |   Recall |
|:-----------------------|-------:|-----------:|------------:|---------:|
| Sans augmentation      | 1.4731 |     0.4712 |      0.3147 |   0.302  |
| Avec augmentation      | 1.3936 |     0.5157 |      0.3966 |   0.4631 |
```

On entraîne donc les mêmes modèles que dans la phase 1 sur nos environnements 1 et 2. Nous avons encore une fois choisi un batch size de 32, mais avec un nombre d'époques plus grand pour essayer d'avoir de meilleurs résultats (toujours de l'early stopping pour éviter l'overfitting).

Lors de l'apprentissage du modèle 1, nous avons remarqué que le recall de notre modèle tendait vers 0 et donc que le modèle n'était pas performant pour détecter les chutes. Or dans le cas où nous avons du mal à généraliser, maximiser le recall est une bonne solution, car il permet que le modèle ne fasse pas de faux négatifs, c'est-à-dire manquer une chute.

Nous avons donc ajusté manuellement les poids. L'objectif de cette manipulation est de dire au modèle de prêter plus d'attention à la classe des chutes (label 1) pendant l'entraînement. En augmentant le poids de la classe 1 et en diminuant celui de la classe 0, on pénalise plus lourdement le modèle s'il se trompe sur une chute et moins lourdement s'il se trompe sur une non-chute.

Modèle 1 (CNN Classique) :

```
Tableau récapitulatif des performances du modèle CNN simple (partie2) :

| Modèle          | Set   |   Loss |   Accuracy |   Precision |   Recall |
|:----------------|:------|-------:|-----------:|------------:|---------:|
| CNN simple (p2) | Train | 0.4637 |     0.7573 |      0.7455 |   0.7151 |
| CNN simple (p2) | Test  | 0.9527 |     0.623  |      0.5141 |   0.6107 |
```

Les performances moyennes du modèle s'expliquent par la complexité de la détection de chutes à partir des signaux, qui sont sûrement sensibles à l'environnement. La variabilité des signaux entre les environnements (env 1 et 2 vs env 3 et 4) rendent potentiellement la généralisation difficile comme on peut le voir avec la loss plutôt élevée sur les environnements 3 et 4 de train.
Enfin, l'architecture CNN simple utilisée, même si elle est renforcée par de l'augmentation de données, reste limitée pour capturer les relations complexes pour obtenir une bonne détection.

Modèle 2 : CNN Hybride avec MobileNetV2

Tableau récapitulatif des performances du modèle Hybrid MobileNetV2 (partie2) :

Modèle	Set	Loss	Accuracy	Precision	Recall
Hybrid MobileNetV2 (p2)	Train (Original Resized)	0.6804	0.64	0.5611	0.9884
Hybrid MobileNetV2 (p2)	Test (Original Resized)	0.6034	0.7487	0.6268	0.8792

On obtient une accuracy assez faible, en revanche le recall est très bon, montrant que le modèle parvient à détecter la majorité des chutes, même s'il génère quelques fausses alertes.

Modèle 3 : CNN-LSTM

Tableau récapitulatif des performances du modèle CNN-LSTM (partie2) :

Modèle	Set	Loss	Accuracy	Precision	Recall
CNN-LSTM (p2)	Train (Original)	0.6958	0.5467	0.5109	0.2733
CNN-LSTM (p2)	Test (Original)	0.681	0.5759	0.4606	0.5101

Les résultats sont globalement décevants, similaires à ceux du premier modèle. Malgré l'ajout d'une composante temporelle via le LSTM, le modèle n'a pas réussi à améliorer significativement la performance, en particulier sur les environnements inconnus. Cela confirme la difficulté de généralisation du modèle.

4. Difficultés Rencontrées

Lors de la première partie du projet, nous avons été confrontés à des limitations au niveau de notre matériel. En effet, les modèles étaient longs à entraîner sur nos ordinateurs personnels qui n'ont pas de GPU, ce qui a restreint nos possibilités d'exploration de certaines architectures plus complexes ou d'optimisation approfondie des hyperparamètres. Par exemple, l'utilisation d'Optuna pour une recherche systématique aurait pu grandement améliorer les performances, mais s'est avérée difficilement exploitable dans nos conditions.

Nous voulions aussi utiliser Resnet50 comme modèle hybride de la partie 1. Néanmoins, pour une utilisation optimale, il faut redimensionner les images au format 224x224x3, ce qui demandait d'allouer 30GB de RAM.

Pour la deuxième partie, la mise en œuvre de notre idée de modèle CNN-LSTM a été plus complexe que prévu, en raison de la difficulté à combiner efficacement les dimensions spatiales et temporelles des données.

La phase 2 s'est révélée aussi particulièrement difficile, les données de signaux sont peu intuitives à manipuler avec de la reconnaissance d'image et nécessitent de toujours réfléchir à ce que l'on fait. Obtenir de très bons résultats s'est donc avéré difficile, malgré nos efforts.

5. Conclusion

Ce projet nous a permis d'explorer des problématiques intéressantes liées à la classification d'images en mettant en œuvre différents modèles, notamment des CNN.

Malgré certaines limitations matérielles et des difficultés techniques, notamment pour la détection de chutes, nous avons pu tester plusieurs approches et mieux comprendre les enjeux liés à la généralisation et au déséquilibre des données. Ce travail nous a également sensibilisés à l'importance du prétraitement et du choix des architectures dans des contextes réels, où les données peuvent être peu nombreuses ou de qualité imparfaite.