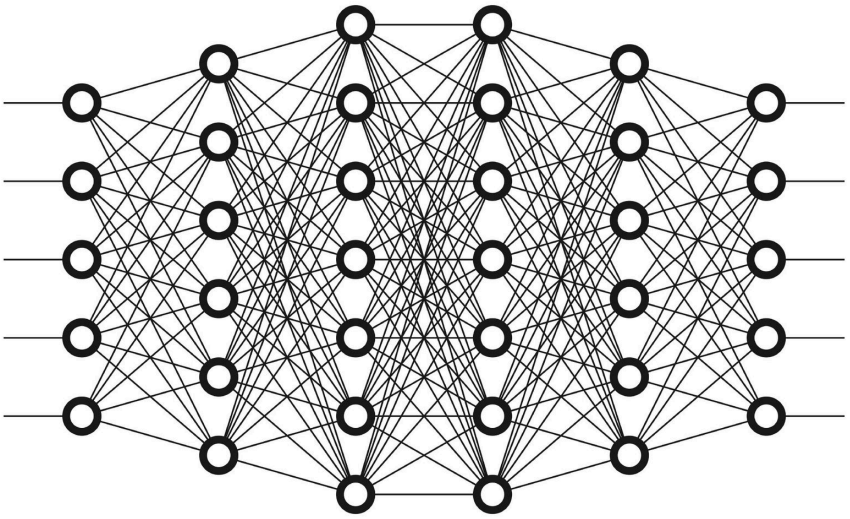
Lichir Mohamed Abderrahmane, Gabriel Rouquet,
Xinyu Zhao, Wajdi Mhamdi

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Projet CVML



Structure du rapport

Ⅰ. Introduction et contexte

Dans le cadre du projet CVML, nous allons concevoir, entraîner et évaluer un modèle d'apprentissage pour une tâche de classification multi-classe. Contrairement aux tâches précédemment abordées en cours TIP, où chaque image était associée à une seule classe, ce projet introduit une nouvelle difficulté en exigeant la prédiction de plusieurs classes simultanément pour chaque image. Ce type de mission reflète davantage les défis du monde réel, où les données ne sont pas limitées à une seule catégorie.

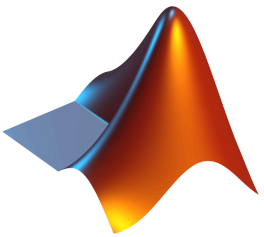
La base de données MS COCO, bien connue pour sa diversité d'images, a été adaptée pour ce projet afin de réduire sa taille et faciliter le traitement. Les données se divisent en deux sous-ensembles : un ensemble d'entraînement (65 000 images) accompagné de leurs étiquettes, et un ensemble de test (environ 5 000 images).

Ce rapport renseigne notre méthodologie de travail, les décisions techniques prises aux différentes étapes du projet, et les performances obtenues par notre modèle sur cette tâche exigeante. Par ailleurs, nous analyserons également les résultats en termes de métriques d'évaluation standards telles que la précision, le rappel et la F1-score. Enfin, des perspectives pour améliorer les performances seront présentées.

Pour réaliser ce projet, nous disposons de trois GPU :

- NVIDIA GeForce RTX 4050 Laptop GPU
- NVIDIA GeForce RTX 4070 Laptop GPU
- NVIDIA GeForce RTX 3050 Laptop GPU

En ce qui concerne le langage de programmation utilisé, après de nombreuses réflexions, nous avons décidé d'utiliser MATLAB en raison de sa simplicité et de sa facilité d'utilisation. Contrairement à PyTorch (Python), qui nécessite un temps d'apprentissage plus important, MATLAB s'est révélé être une solution plus adaptée à nos besoins. De plus, nous avons estimé que PyTorch pourrait être plus exigeant en termes de compilation et de performances sur nos équipements.

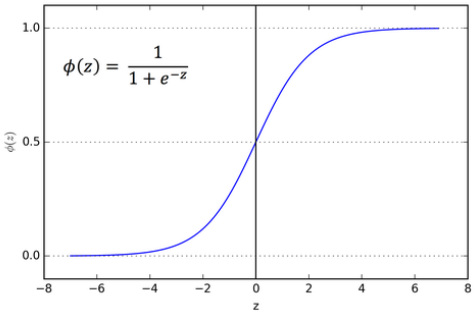


II. Approches techniques

II.1 Adaptation des réseaux pour la classification multi-label

Plusieurs approches existent pour réaliser une classification multi-classes, regroupées en trois grandes catégories

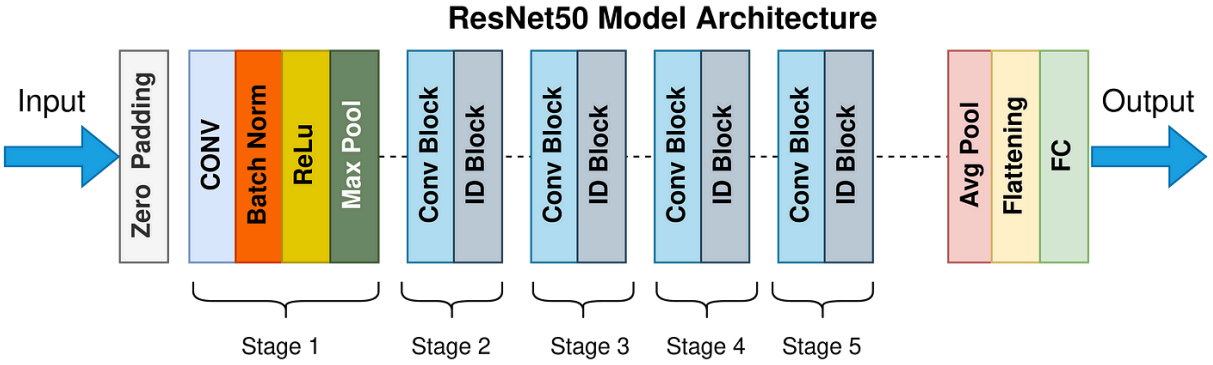
- Type de réseau de neurone utilisé : à première vue, on peut opter pour un réseau de neurones classique composé de couches entièrement connectées, où chaque neurone d'une couche est relié à tous ceux de la couche suivante. Bien que ce type de réseau soit simple à concevoir, il s'avère inefficace pour le traitement d'images. Une alternative plus adaptée est l'utilisation de réseaux convolutifs (CNN), qui permettent de détecter des motifs locaux à l'aide de filtres convolutifs. Ces réseaux offrent une excellente précision pour les tâches liées aux images, mais leur utilisation exige des ressources plus importantes.
- La couche finale à utiliser : la couche finale softmax classique attribue une probabilité normalisée à chaque classe, avec une somme totale des probabilités égale à 1. Elle ne permet cependant de sélectionner qu'une seule classe par image, correspondant à celle ayant la probabilité la plus élevée. Afin de détecter plusieurs classes sur une même image, la couche finale doit être remplacée par une sigmoid. Cette fonction permet de traiter chaque sortie indépendamment en lui attribuant une probabilité. Puis une valeur seuil de probabilité permet de sélectionner les classes qui existent sur l'image.



- La méthode utilisée : La classification multi-classes peut être effectuée en créant un nouveau réseau, ce qui nécessite un investissement important en temps et en données pour l'entraîner, ou en utilisant le principe du transfert learning. Cela consiste à utiliser un réseau déjà pré-entraîné sur une vaste quantité d'images et réentrainer seulement les dernières couches sur nos images afin de conserver les caractéristiques génériques apprises par le réseau et réduire les besoins en données et en calcul. Pour avoir de bons résultats, il est évident qu'il faut choisir un réseau qui a été entraîné pour des données similaires. Par ailleurs, le transfert learning peut être optimisé en utilisant des ensembles de modèles, où plusieurs réseaux de neurones sont combinés pour tirer parti de leurs forces et compenser leurs faiblesses. Les erreurs sont réduites en moyennant leurs résultats ou en appliquant un vote majoritaire. Toutefois, cette approche exige davantage de temps et de puissance de calcul.

II.2. Architecture réseau choisie

Dans ce projet on a décidé d'utiliser Resnet, c'est un réseau de neurones convolutionnel couramment utilisé dans le domaine de l'apprentissage profond pour diverses tâches de vision par ordinateur, comme la classification d'images, la détection d'objets et la segmentation. Il fait partie de la famille des réseaux résiduels, introduits en 2015 par Kaiming He et son équipe, dans l'article intitulé *"Deep Residual Learning for Image Recognition"*. La raison de ce choix repose premièrement sur le fait que ce réseau est déjà entraîné à détecter des classes qui nous intéressent telles que des animaux, des objets communs, des véhicules, des aliments… En effet, il excelle dans la détection de ces éléments, ayant été entraîné sur une base de données contenant plus de 14 millions d'images. De plus, le réseau Resnet avec ses différentes déclinaisons offre un bon compromis entre précision de prédiction et temps d'entraînement. En effet, les différentes variantes de ce réseau nous permettent de faire des tests sur les réseaux les plus rapides à tourner (tel que resnet-18) et les appliquer sur des réseaux plus complexes qui demandent un temps de calcul considérablement plus élevé. Ce réseau est également adapté au fine-tuning, c'est-à-dire à geler toutes les couches, mis à part les dernières pour retenir les caractéristiques spécifiques de notre base de données et s'ajuster aux classes ciblées.
A ce réseau Resnet, on va ajouter une couche sigmoid qui va remplacer la couche softmax par défaut (celle-ci permet la classification multi-label) et on va changer la couche entièrement connectée pour avoir un nombre de neurones égal à celui de nos classes. Durant ce projet, on a également ajouté une couche dropout qui permet de désactiver un pourcentage des neurones de la couche entièrement connectée pendant un moment afin d'observer et de juger si cela améliore les résultats.
Tout au long du projet, notre méthode consiste à maximiser les options d'entraînement en les traitant une par une avec un nombre d'époques assez réduit. Ainsi, le travail peut être divisé sur de nombreuses machines.



III. Méthodologie et entraînement

III.1. Préparation des données et dataset

La première étape réalisée est la mise en forme des données afin de les appliquer dans le réseau qu'on a créé. Parmi les éléments fournis, nous disposions d'un répertoire contenant des fichiers associant chaque image à ses labels correspondants. A partir de cela, nous avons généré un fichier csv, à l'aide d'un programme matlab, facilitant l'accès aux images d'entraînement et à leurs labels. Un aperçu du fichier csv généré par le script matlab est donnée ci-dessous :

ImageFile	person	bicycle	car	motorcycle	airplane	bus	train	truck	boat	traffic light	fire hydrant
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	1	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	1	0	0	0	0	0	0	0	0	0	0
000000000...	1	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	1	0	0	0	0	1	0	0	0
000000000...	0	0	1	0	0	0	1	1	0	0	0
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	0	1	0	0	0	0	0	0	0
000000000...	1	1	0	0	0	0	0	0	0	0	0
000000000...	1	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	0	0	1	0	0	0	0	0	0
000000000...	1	0	0	1	0	0	0	0	0	0	0
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	1	0	0	0	0	1	0	0	0
000000000...	1	0	0	0	0	0	0	0	0	0	0
000000000...	1	0	0	0	0	0	0	0	0	0	0
000000000...	1	0	0	0	0	0	0	0	0	0	0
000000000...	1	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	1	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	0	0	0	0	0	0	0	0	0	0	0
000000000...	1	0	1	0	0	0	0	0	0	0	0

Figure 1: contenu du fichier csv

Le fichier généré contient plusieurs colonnes : la première correspond aux noms des images, alors que les colonnes suivantes représentent les labels associés. Pour chaque label, une valeur de 1 indique la présence de l'objet correspondant, tandis qu'une valeur de 0 signifie son absence.

Une fois le fichier csv obtenu, nous pouvons commencer à préparer un dataset. Celui-ci nous permettra d'organiser les données, séparer l'ensemble d'entraînement et de validation ainsi que de gérer efficacement les ressources en traitant les données par lots (batch processing) sans nécessiter le chargement complet en mémoire. Il y a 65000 images au total pour l'entraînement. Afin d'obtenir un résultat pertinent, nous avons décidé de séparer les 65000 images en deux parties, 80% pour les images d'entraînement (52000 images) et 20 % pour les images de validation (13000 images). Par la suite, nous devons obtenir les informations de la couche "input" du Resnet-50 en utilisant la commande **inputSize = net.Layers(1).InputSize**, afin d'ajuster la taille des images pour l'entraînement.

```
%les images et les labels pour entrainer
dataTableTrain = dataTable(1:52000, :);
dataTrain = augmentedImageDatastore(inputSize(1:2),dataTableTrain, ...
        ColorPreprocessing="gray2rgb", ...
        DataAugmentation=imageAugmenter);

encodedLabelTrain = dataTableTrain.Labels;
numObservations = dataTrain.NumObservations;

%les images et les labels pour tester
dataTableTest = dataTable(52001:65000, :);
dataVal = augmentedImageDatastore(inputSize(1:2),dataTableTest, ...
        ColorPreprocessing="gray2rgb", ...
        DataAugmentation=imageAugmenter);

encodedLabelVal = dataTableTest.Labels;
numObservationsPerClass = sum(encodedLabelTrain,1);
```

Figure 2: implémentation du dataset

III.2. Conception du réseau

La figure ci-dessous illustre les performances des différents modèles pré-entraînés. En tenant compte des capacités de nos GPU, nous avons opté pour un modèle équilibré entre consommation de ressources GPU et précision, ce qui confirme notre choix du ResNet-50.

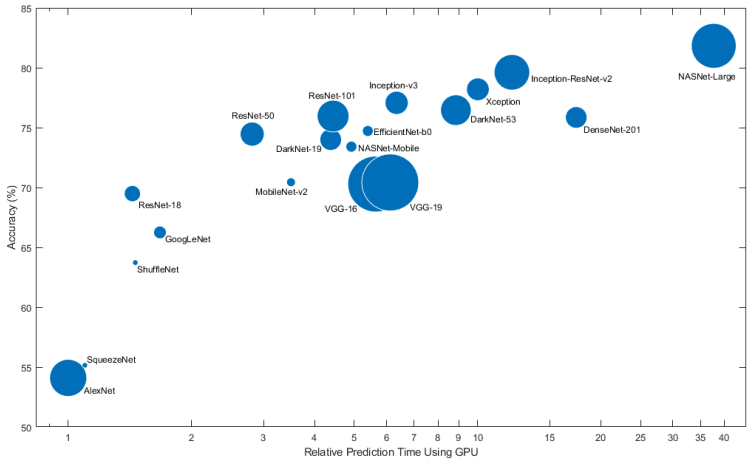


Figure 3: performances des différents modèles pré-entraînés

Afin de créer notre réseau, nous avons tout d'abord implémenté le modèle Resnet-50 et gelé toutes les couches sauf la couche "fully connected"comme montré ci-dessous:

```
% Geler uniquement les couches compatibles avec le facteur d'apprentissage
netlayers=net.Layers;
for i = 1:length(netlayers)
    % Vérifiez si la couche supporte 'WeightLearnRateFactor'
    if isprop(netlayers(i), 'WeightLearnRateFactor') && isprop(netlayers(i), 'BiasLearnRateFactor')
        % Geler les couches en définissant les facteurs à 0
        netlayers(i).WeightLearnRateFactor = 0;
        netlayers(i).BiasLearnRateFactor = 0;
    end
end
```

Figure 4: extrait de code illustrant le gel des couches

Ensuite, comme vu précédemment, nous devons remplacer les couches finales “fully connected” et “softmax” par des couches plus adaptées à la classification de multi-labels. Ainsi, nous avons remplacé la couche "fully connected" d'origine par notre propre couche "fully connected", appelée new_fc, configurée pour 80 classes (correspondant à nos 80 types de labels). Nous avons également spécifié les paramètres **WeightLearnRateFactor = 10** et **BiasLearnRateFactor = 10**. Grâce à ces deux paramètres, nous pouvons désormais entraîner cette couche.

Nous avons également remplacé la couche “softmax” par “sigmoid”, car celle-ci peut prédire la probabilité de chaque catégorie indépendamment, tandis que Softmax force les catégories à s'exclure mutuellement et ne peut pas prendre en charge l'existence de plusieurs étiquettes en même temps. Une fois que le modèle a été bien modifié, nous utilisons la commande **analyzeNetwork(net)** pour visualiser si le modèle a été bien modifié. Les codes ci-dessous montrent très bien les démarches que nous avons mentionnées.

```
%le reseau resnet50 avec 80 classes
numClasses = 80;
net = imagePretrainedNetwork("resnet50",NumClasses=numClasses);
```

```
%add a sigmoid layer
newLayers = [
    fullyConnectedLayer(numLabels, 'Name', 'new_fc', 'WeightLearnRateFactor', 10, 'BiasLearnRateFactor', 10)
    sigmoidLayer('Name', 'sigmoid')
];

%delete the softmax layer of the original network and connect the sigmoid
%layer to the network
changenetwork = true;
if changenetwork
    net = removeLayers(net, 'fc1000');
    net = removeLayers(net, 'fc1000_softmax');
    net = addLayers(net, newLayers);
    net = connectLayers(net, 'avg_pool', 'new_fc');
end
```

Figure 5: extrait de code illustrant la modification des couches du modèle pré-entraîné

Une fois les données préparées et le réseau mis en place, nous allons pouvoir commencer l'entraînement. La prochaine étape est de configurer les options telles que **l'algorithme d'optimisation**, **InitialLearnRate, LearnRateDropFactor, MiniBatchSize, MaxEpochs, ValidationData, ValidationFrequency,ValidationPatience, Shuffle**, etc. Ces options sont importantes puisqu'elles influencent le résultat de notre entraînement. Un petit aperçu des options choisies est donné ci-dessous :

L'algorithme d'optimisation que nous avons choisi est SGDM (Stochastic Gradient Descent with Momentum). Il met à jour les poids du modèle en fonction du gradient de la perte, tout en utilisant un mécanisme de momentum pour lisser ces mises à jour. Cela permet de réduire les oscillations, d'accélérer la convergence et d'assurer un entraînement plus stable, ce qui en fait un choix pertinent pour une tâche de classification multi-classe.

InitialLearnRate est le taux d'apprentissage au début de l'entraînement, qui détermine la taille du pas de mise à jour du poids et joue un rôle clé dans la vitesse d'entraînement et l'effet de convergence du modèle. Après avoir effectué plusieurs tests, nous avons estimé que la meilleure valeur du taux d'apprentissage initial est de 0,01.

LearnRateDropFactor contrôle la réduction progressive du taux d'apprentissage. À chaque cycle, le taux actuel est multiplié par ce facteur pour affiner l'optimisation du modèle. Ici, nous avons utilisé un facteur de 0.1 tous les 5 epochs.

MiniBatchSize est le nombre d'échantillons de données traités dans chaque itération d'entraînement, ce qui affecte l'efficacité de l'entraînement, la stabilité et l'utilisation de la mémoire du modèle. En tenant compte des performances de nos GPU, nous avons choisi 32 pour cette option.

MaxEpochs définit le nombre maximal de passages sur l'ensemble de données pendant l'entraînement, déterminant le nombre total de cycles d'optimisation. Pour ce projet, nous avons fixé cette valeur à 15.

ValidationData est un ensemble de données utilisé pour évaluer la précision du modèle pendant l'entraînement.

ValidationFrequency est utilisé pour évaluer le modèle après un certain nombre d'itération. Afin d'économiser du temps,nous décidons de faire une validation après 100 itérations.

ValidationPatience est un paramètre qui nous permet d'éviter le surapprentissage. Autrement dit, après certains nombre de validation, si la précision du modèle n'augmente toujours pas, l'entraînement va s'arrêter automatiquement. Pour nous, notre ValidationPatience est de 5.

Finalement, nous utilisons l'option **shuffle** pour mélanger l'ordre des images d'entraînement afin de réduire la dépendance du modèle à l'égard de l'ordre des images d'entraînement.

```
%trainning options
options = trainingOptions("sgdm", ...
    InitialLearnRate=0.01, ...
    LearnRateSchedule='piecewise', ...
    LearnRateDropFactor=0.1, ...
    LearnRateDropPeriod=5, ...
    MiniBatchSize=32, ...
    MaxEpochs=15, ...
    Verbose= false, ...
    ValidationData=dataVal, ...
    ValidationFrequency=40, ...
    ValidationPatience=Inf, ...
    Metrics="accuracy", ...
    Shuffle='every-epoch', ...
    Plots="training-progress");
```

Figure 6: options d'entraînement

Après avoir effectué le pré-traitement des images et établi les paramètres d'entrainements, on peut entraîner le réseau.

```
trainedNet = trainnet(dataTrain,net,"binary-crossentropy",options);
```

La fonction de perte choisie est **Binary Cross Entropy (BCE)**. Celle-ci évalue à quel point les prédictions d'un modèle s'écartent des étiquettes réelles pour chaque classe, en considérant chaque sortie indépendamment. Cela en fait un choix idéal pour les problèmes où une instance peut appartenir à plusieurs classes simultanément (multi-label).

$$BCE = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(\widehat{y_i}) + (1 - y_i) log(1 - \widehat{y_i})]$$

avec : N le nombre total de classes, y_i l'étiquette réelle (0 ou 1) pour la classe et $\widehat{y_i}$ la probabilité prédite pour la classe i

La BCE a été choisie car elle s'adapte parfaitement aux tâches multi-label en permettant de traiter chaque classe indépendamment. Cela correspond à notre besoin de gérer des images pouvant appartenir à plusieurs catégories simultanément. De plus, son utilisation avec une sortie sigmoid offre une flexibilité accrue pour attribuer des probabilités individuelles à chaque classe, optimisant ainsi l'apprentissage des relations complexes dans les images.

III.3. Entraînement et évaluation

III.3.i Premiers résultats d'entraînement

Après une session d'entraînement, l'exactitude atteint 31 %. Comparée à celle obtenue après une seule époque, qui est de 19 %, l'entraînement sur 15 époques montre une nette amélioration.

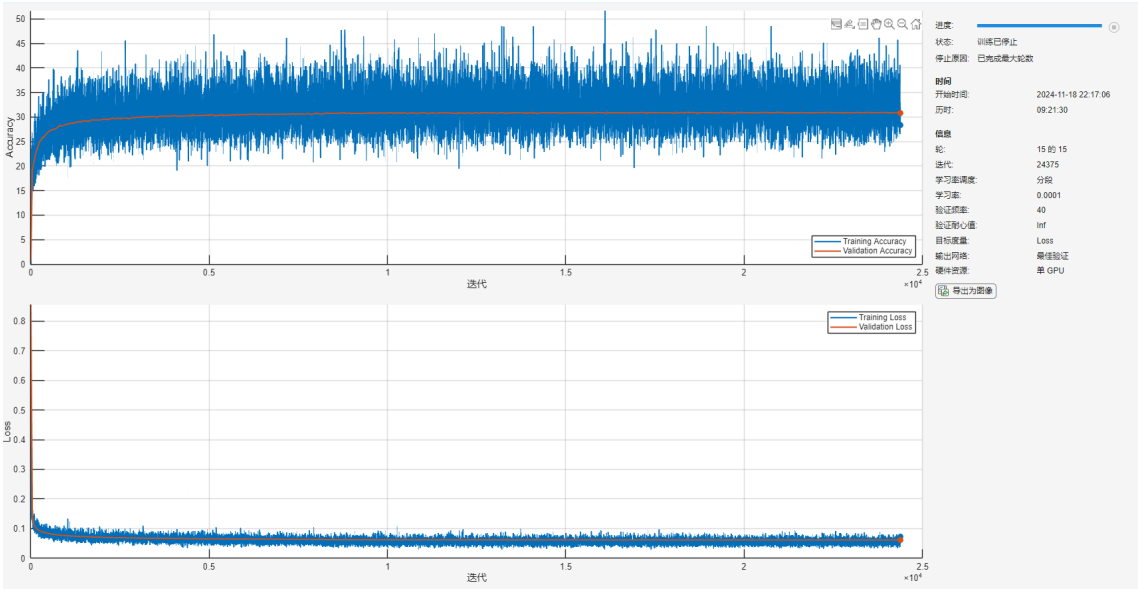


Figure 7: accuracy et loss

Afin de trouver le meilleur seuil, nous avons créé une fonction qui nous permet de visualiser le f1score (par rapport aux image de validation) pour chaque valeur de seuil. En s'aidant de cette figure, nous sommes capables de trouver le meilleur seuil immédiatement. Dans ce cas là, 0.4 est la meilleure valeur qui a un f1score d'environ 0.67.

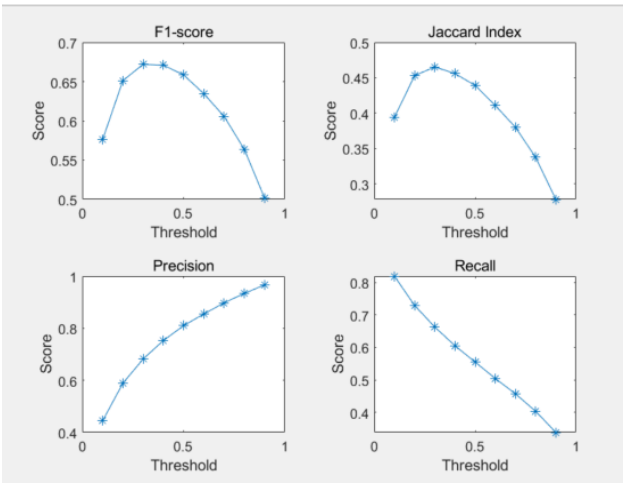


Figure 8: f1score, précision et rappel pour les images de validation

III.3.ii Génération JSON et tests

Afin d'évaluer nos résultats, nous avons testé le modèle sur l'ensemble des images de test. Pour cela, nous avons développé un script permettant d'évaluer les images et générer un fichier JSON contenant les prédictions du modèle.

Malgré quelques soucis rencontrés lors du test des images, notamment à cause de la présence de certaines images en niveaux de gris exigeant une conversion préalable, nous avons réussi à stocker les résultats dans une table pour générer le fichier JSON.

Par ailleurs, nous n'avons pas réussi à générer le fichier JSON dans le format requis, probablement en raison de notre faible familiarité avec le langage MATLAB. Pour contourner cette difficulté, nous avons développé un script en Python permettant de convertir le fichier JSON au format attendu.

Ce problème rencontré, ainsi que le format du fichier JSON généré, est illustré ci-dessous.

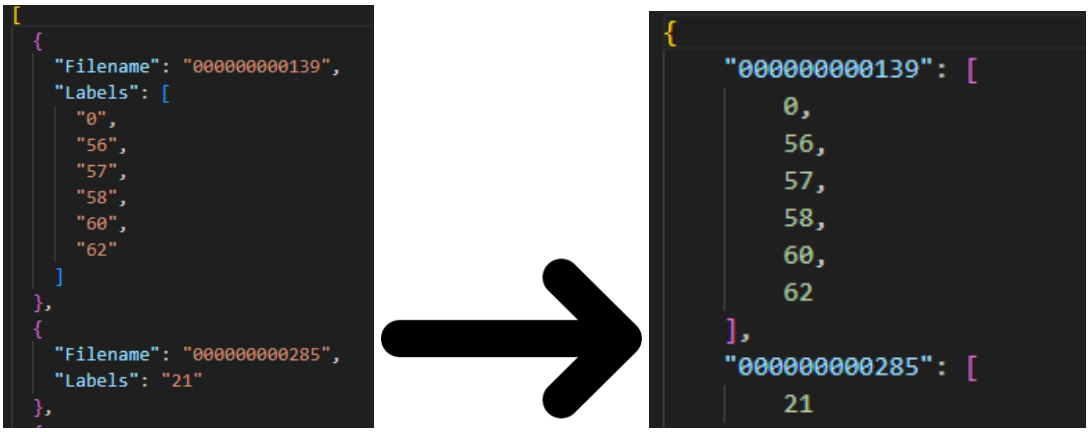


Figure 9 : JSON généré avec MATLAB (gauche) et corrigé par le script Python (droite).

Après avoir déposé le fichier Json sur le site web d'évaluation fourni par le professeur, nous avons constaté que, notre f1score est de 0.4762, l'accuracy est de 0.6087, la précision est de 0.5741 et le rappel est de 0.4068, comme montré ci-dessous

Metric	Value
accuracy	0.6087
f1	0.4762
precision	0.5741
recall	0.4068

Figure 11: les premiers résultats

III.3.iii Optimisation et amélioration des performances

Nous avons également expérimenté l'ajout d'une couche dropout, qui désactive aléatoirement certains neurones afin de réduire la dépendance du modèle à des caractéristiques spécifiques et de prévenir le surapprentissage. En parallèle, nous avons testé l'ajout de la régularisation L2, qui applique une pénalité basée sur la somme des carrés des poids pour limiter leur valeur et réduire le risque de surapprentissage. Plusieurs configurations avec différentes valeurs de dropout et de régularisation L2 ont été explorées. Cependant, nos tests ont révélé que le modèle atteignait un meilleur F1-score sans utiliser la couche dropout ni la régularisation L2. Le tableau suivant présente les résultats obtenus (pour un seul epoch) avec différents paramètres.

dropout	L2Regularization	f1 score
0.4	0.0001	0.6065
0.3	0.0001	0.6094
0.2	0.0001	0.6158
sans dropout	0.0001	0.6224
sans dropout	sans L2Regularization	0.6269

Après plusieurs essais et ajustements, nous avons identifié les paramètres qui donnent les meilleurs résultats pour notre modèle. Voici les options d'entraînement retenues après ce processus d'optimisation :
options = trainingOptions("sgdm", InitialLearnRate=0.01, LearnRateSchedule='piecewise', LearnRateDropFactor=0.1, LearnRateDropPeriod=10, MiniBatchSize=32, MaxEpochs=30, Verbose= false, ValidationData=dataVal, ValidationFrequency=100, ValidationPatience=Inf, Metrics="accuracy", Shuffle='every-epoch', Plots="training-progress");

Ces paramètres ont été obtenus après une série de tests visant à maximiser la performance de notre modèle.
Nos résultats finaux sont ceux ci-dessous :

Metric	Value
accuracy	0.6598
f1	0.5382
precision	0.6361
recall	0.4665

Figure 12: performances finales du modèle

IV. Perspectives et Limitations

Une piste intéressante pour améliorer les performances consisterait à utiliser des modèles récents et performants comme EfficientNet ou Vision Transformers (ViT), qui exploitent les dernières avancées en vision par ordinateur. Par ailleurs, il serait pertinent de tester le modèle sur des données externes à MS COCO afin d'évaluer sa robustesse et sa capacité de généralisation.

Cependant, les performances restent contraintes par les capacités des GPU disponibles, limitant la possibilité d'entraîner des modèles plus complexes ou sur de longues durées. Enfin, bien que MATLAB se soit avéré adapté pour ce projet, son utilisation restreint l'accès à certaines fonctionnalités avancées et à des bibliothèques modernes comme celles proposées par PyTorch ou TensorFlow, qui offrent une plus grande flexibilité pour des projets de grande envergure.

V. Conclusion

Le projet CVML nous a permis de concevoir, d'entraîner et d'évaluer un modèle qui relève le défi de la classification multi-label sur une base de données diversifiée comme MS COCO. Après une exploration des approches classiques et l'adaptation d'un modèle pré-entraîné ResNet, nous avons pu atteindre des performances satisfaisantes en optimisant méthodiquement les hyperparamètres et les configurations du réseau.

Grâce à l'utilisation de techniques comme l'apprentissage par transfert, la couche sigmoid, et la fonction de perte Binary Cross Entropy, nous avons adapté le modèle à la nature multi-label du projet. Des expérimentations, comme l'ajout de dropout et de régularisation L2, ont été menées pour mesurer leur impact sur les performances, même si elles n'ont pas été retenues dans la configuration finale du réseau.

Nos résultats finaux, avec un F1-score de **0.5382** et une précision de **0.6361**, reflètent les efforts déployés pour optimiser le modèle.

En conclusion, ce projet nous a permis d'acquérir une certaine expertise dans la classification multi-label et dans l'optimisation des réseaux de neurones convolutionnels, tout en posant de bonnes bases pour des recherches et développements futurs.