

Projet – Optimisation et Convexe

Sujet A : Applications des techniques de l'optimisation dans un problème de Machine Learning concret

Application : Classification de baskets Nike, Adidas et Converse



Groupe de Projet

- Da Cruz Mathis
- Sebastiao Esteves Kevin
- Sivananthan Sarankan

Professeur

- Benedetti Léonard

Table des matières

1. Présentation du Projet	3
1.1. Préambule	3
1.2. Datasets	3
1.3. Ressources	3
1.3.1. Convolution	3
1.3.2. InceptionV3	4
2. Optimisation	5
2.1. Optimisation d'un réseau de neurone	5
2.2. Softmax	5
2.3. Optimisateur Adam	5
2.4. Categorical cross-entropy	7
2.5. Evolution de l'accuracy et de la valeur de la cross-entropy en fonction des epochs	9
2.6. Matrice de confusion pour le meilleur et le plus faible modèle	9
2.7. Courbe ROC pour le meilleur et le plus faible modèle	11
2.8. Aperçu 3D de la categorical cross-entropy	12
3. Partie Code	13
3.1. Librairies importées	13
3.1.1. TensorFlow	13
3.1.2. Keras	13
3.1.3. Sous librairies Keras	14
3.2. Chargement des Images pour le Train et le Test	15
3.3. Initialisation du Modèle Pré-Entraîné InceptionV3	17
3.4. Ajout d'un Layer Personnalisé	18
3.5. Visualisation Sommaire du Modèle	19
3.6. Entraînement du modèle	20
3.7. Evaluation du modèle	21
3.8. Tests sur le modèle avec des graphes	23
3.9. Comparaison avec et sans optimisateur Adam	24
4. Ouverture	25

1. Présentation du Projet

1.1. Préambule

Le projet consiste à classer des images de baskets de différentes marques (Nike, Adidas et Converse), c'est-à-dire associer une image en entrée à un label de sortie accompagné par le taux de confiance de la prédiction faite.

1.2. Datasets

Le projet [Shoe Brand Classification | Nike-Adidas-Converse](#) de Kaggle publié par Arya Shah est un projet réalisé sur un notebook et à partir de plusieurs ressources.

1.3. Ressources

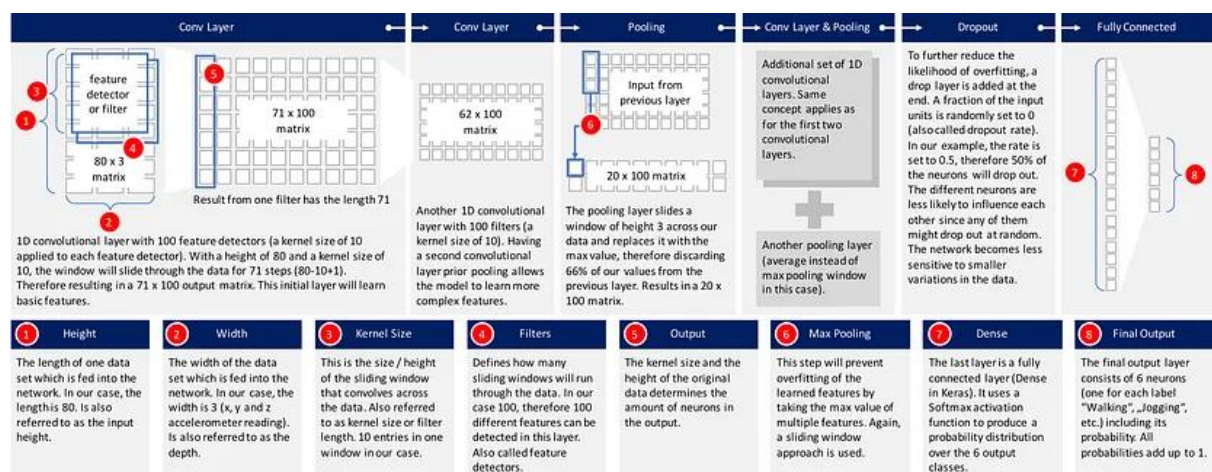
Pour entraîner ce modèle de deep-learning, il suffit d'utiliser l'architecture InceptionV3 proposée par TensorFlow Keras. Avant de parler d'InceptionV3, nous allons introduire la Convolution.

1.3.1. Convolution

La convolution est une opération mathématique utilisée dans le traitement de signal et d'image. Dans le contexte de l'apprentissage en profondeur, la convolution est une opération fondamentale pour les réseaux de neurones convolutifs (CNNs).

La convolution consiste à faire glisser un petit noyau (appelé filtre) sur l'image d'entrée et à calculer le produit scalaire entre les valeurs des pixels de l'image et les valeurs des pixels du filtre à chaque position. Ce produit scalaire est ensuite utilisé pour produire un seul pixel de la sortie, qui représente une combinaison pondérée des pixels d'entrée couverts par le filtre.

En utilisant des filtres différents, les CNNs peuvent extraire des caractéristiques différentes de l'image d'entrée, telles que des bords, des coins, des textures et des formes. En combinant plusieurs couches de convolution avec des opérations non-linéaires telles que la fonction d'activation ReLU et des opérations de mise en commun (pooling), les CNNs peuvent apprendre des représentations hiérarchiques de l'image d'entrée qui sont utilisées pour la classification ou la segmentation d'images.

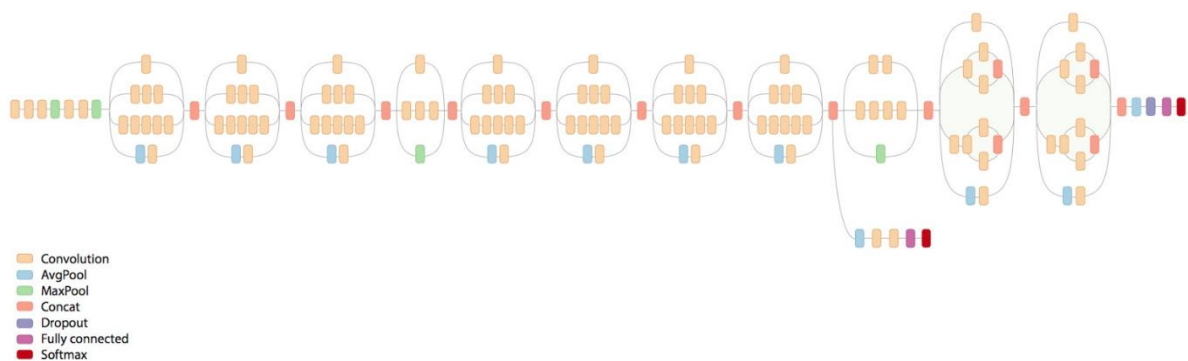


1.3.2. InceptionV3

InceptionV3 est un modèle de réseau de neurones convolutifs (CNN) pré-entraîné qui a été introduit par Google en 2015 pour la reconnaissance d'images à grande échelle. Il a été construit en utilisant une architecture Inception, qui est une famille de modèles de réseau de neurones convolutifs qui utilisent des modules Inception pour extraire des caractéristiques de l'image d'entrée.

Le modèle InceptionV3 est plus profond et plus complexe que les versions précédentes de l'architecture Inception, avec 48 couches de convolution et plus de 23 millions de paramètres entraînaables. Il a été pré-entraîné sur la base de données d'images ImageNet, qui contient plus d'un million d'images réparties sur 1 000 catégories.

InceptionV3 est souvent utilisé comme base pour la classification d'images dans les tâches de vision par ordinateur, car il a démontré de très bonnes performances sur les ensembles de données standard, tels que ImageNet et CIFAR-100. Il a également été utilisé dans des applications de détection d'objets et de segmentation d'images.



2. Optimisation

2.1. Optimisation d'un réseau de neurone

Les poids d'un réseau de neurones sont appris par l'intermédiaire d'un processus d'apprentissage appelé entraînement. L'objectif de l'entraînement est de trouver les poids qui minimisent la fonction de coût du réseau de neurones pour un ensemble de données d'entraînement donné. Pour trouver les poids, le réseau de neurones est entraîné sur un ensemble de données d'entraînement étiquetées. Pendant l'entraînement, le réseau de neurones ajuste progressivement les poids en utilisant un algorithme d'optimisation, tel que Adam dans notre cas, pour minimiser la fonction de coût.

La fonction de coût mesure l'écart entre les sorties du réseau de neurones, c'est-à-dire les prédictions, par rapport à sa vraie valeur. Plus précisément, elle mesure à quel point les prédictions du réseau de neurones diffèrent des étiquettes réelles. L'objectif de l'entraînement est donc de trouver les poids qui minimisent cet écart. Pour ce faire, l'algorithme d'optimisation utilise le gradient de la fonction de coût (dans notre cas la catégorical cross entropy) par rapport aux poids du réseau de neurones. Le gradient va permettre d'indiquer la direction de la pente la plus raide de la fonction de coût, c'est-à-dire la direction dans laquelle les poids doivent être ajustés pour minimiser la fonction de coût. L'algorithme d'optimisation ajuste ensuite les poids dans cette direction.

L'entraînement se déroule par itérations sur l'ensemble de données d'entraînement. À chaque itération, le réseau de neurones calcule les sorties pour chaque exemple d'entraînement, puis ajuste les poids pour minimiser la fonction de coût globale pour l'ensemble de données d'entraînement.

2.2. Softmax

La Softmax est couramment utilisée pour prédire une sortie pour une classification multi-classes d'images à partir d'un réseau de neurones.

Ici, pour une classification multi-classes, le réseau de neurones a une couche de sortie composée de plusieurs neurones, où chaque neurone représente une classe de sortie possible. Chaque neurone de sortie a une activation qui représente la probabilité que l'image d'entrée appartienne à cette classe.

Pour $a = (a_1, a_2, \dots, a_n)$ un vecteur d'entrée, et $k = 3$ (le nombre de classes de classifications)

La Softmax est appliquée à la sortie de la couche de neurones de sortie pour normaliser les activations et les transformer en une distribution de probabilité. Cela signifie que la somme de toutes les activations des neurones de sortie sera égale à 1, et chaque activation sera dans l'intervalle $[0, 1]$, ce qui peut être interprété comme la probabilité que l'image d'entrée appartienne à chaque classe de sortie.

Plus précisément, pour une image d'entrée donnée, le réseau de neurones calcule les activations de chaque neurone de sortie en effectuant une propagation avant à travers le réseau. Ensuite, la Softmax est appliquée aux activations pour normaliser la sortie en une distribution de probabilité. La sortie normalisée peut ensuite être interprétée comme la probabilité que l'image d'entrée appartienne à chaque classe de sortie.

2.3. Optimisateur Adam

L'algorithme d'optimisation Adam est une version améliorée de la descente de gradient stochastique. Elle s'adapte bien pour les applications d'apprentissage profond, des réseaux de neurones ou encore le NLP.

Sa principale fonction est donc de mettre à jour itérativement les poids du réseau, basés sur les données de formation.

- Avantages de l'utilisation d'Adam sur les problèmes d'optimisation non vexatoire, comme suit :
- Facile à mettre en œuvre.
- Calcul efficace.
- Peu de mémoire requise.
- Bien adapté aux problèmes qui sont importants en termes de données et/ou de paramètres.
- Les hyperparamètres ont une interprétation intuitive et nécessitent généralement peu de réglage.

Adam est différent de la descente de gradient classique (et stochastique). La descente de gradient stochastique maintient un taux d'apprentissage unique (appelé alpha, learning rate) pour toutes les mises à jour de poids et le taux d'apprentissage ne change pas pendant l'entraînement.

Adam quant à lui, a un taux d'apprentissage qui est maintenu pour chaque poids du réseau (paramètre) et adapté séparément au fur et à mesure de l'apprentissage.

Cet optimiseur combine de 2 extensions de la descente de gradient classique :

- **Adaptive Gradient Algorithm** (AdaGrad) qui maintient un taux d'apprentissage par paramètre qui améliore les performances sur les problèmes avec des gradients épars
- **Root Mean Square Propagation** (RMSProp) qui maintient également des taux d'apprentissage par paramètre qui sont adaptés en fonction de la moyenne des amplitudes récentes des gradients pour le poids.

Au lieu d'adapter les taux d'apprentissage des paramètres en fonction du premier moment moyen (la moyenne) comme dans RMSProp, Adam utilise également la moyenne des deuxièmes moments des gradients (la variance non interrompue). Plus précisément, l'algorithme calcule une moyenne mobile exponentielle du gradient et du gradient carré, et les paramètres β_1 et β_2 contrôlent les taux de décroissance de ces moyennes mobiles.

Les paramètres β_1 et β_2 de l'optimiseur Adam sont des hyperparamètres qui régissent la décroissance des moyennes mobiles des gradients et de leur carré, respectivement.

Le paramètre β_1 ($0 < \beta_1 < 1$) est utilisé pour calculer la moyenne mobile des gradients. Plus précisément, c'est le coefficient d'atténuation exponentielle pour la moyenne mobile des gradients, qui est utilisée pour estimer la moyenne (ou premier moment) des gradients. Il contrôle la pondération relative entre le gradient actuel et la moyenne mobile des gradients précédents. Par défaut, sa valeur est de 0,9.

Le paramètre β_2 ($0 < \beta_2 < 1$) est utilisé pour calculer la moyenne mobile du carré des gradients. Il est également utilisé pour estimer variance (ou deuxième moment) des gradients. Il contrôle la pondération relative entre le carré du gradient actuel et la moyenne mobile des carrés de gradients précédents.

En général, des valeurs par défaut de 0,9 pour β_1 et de 0,999 pour β_2 sont souvent utilisées dans la pratique, mais ces valeurs peuvent être ajustées pour optimiser les performances pour un problème spécifique.

En modifiant ces paramètres, on peut influencer la façon dont Adam met à jour les poids du réseau neuronal lors de l'entraînement, ce qui peut affecter les performances du modèle.

La moyenne mobile des gradients et la moyenne mobile du carré des gradients sont deux moyennes mobiles utilisés dans l'optimiseur Adam pour estimer respectivement la moyenne et la variance des gradients des poids du réseau.

La moyenne mobile des gradients (mean of gradients) est une estimation de la moyenne des gradients de la fonction de coût pour les poids du réseau neuronal. Elle est calculée à chaque étape d'entraînement en prenant une moyenne pondérée des gradients actuels et des moyennes mobiles des gradients précédents, où le coefficient d'atténuation exponentielle β_1 contrôle le taux d'oubli des anciens gradients.

- Le premier moment m_t des gradients (moyenne mobile des gradients) correspond à cette moyenne mobile des gradients. Il s'agit d'une estimation de la direction du gradient optimal pour minimiser la fonction de coût, utilisée pour mettre à jour les poids du réseau. Son expression, $m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * \text{Gradient}$
- Le deuxième moment v_t des gradients (moyenne mobile du carré des gradients) est une estimation de la variance des gradients de la fonction de coût pour les poids du réseau neuronal. Elle est également calculée à chaque étape d'entraînement en prenant une moyenne pondérée des carrés des gradients actuels et des moyennes mobiles des carrés de gradients précédents, où le coefficient d'atténuation exponentielle β_2 contrôle le taux d'oubli des anciens carrés de gradients. Son expression, $v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * \text{Gradient}^2$

En utilisant ces deux moyennes mobiles, l'optimiseur Adam peut normaliser les gradients des poids du réseau avant la mise à jour des poids, afin d'adapter la taille de la mise à jour des poids en fonction de la variance des gradients. Cela permet à l'optimiseur de converger plus rapidement et d'atteindre des solutions de meilleure qualité.

2.4. Categorical cross-entropy

$$LCE = - \sum_{i=1}^3 y_i * \log(p_i)$$

On utilise dans notre modèle une fonction de coût nommée la cross-entropy catégorique (cf formule en LaTeX) avec 3 labels, c'est une fonction très souvent utilisée dans les modèles de classification dû à sa simplicité et également le fait qu'elle accorde plus d'importance à une petite valeur et qu'elle minimise les grosses valeurs grâce au log qui va donc accorder plus ou moins de poids en fonction de la valeur, c'est la version généralisée de la binary cross-entropy qui elle est une fonction pour seulement deux labels à entraîner. Il s'agit d'une fonction convexe qui est mise à jour à chaque traitement par batch dont elle va ensuite calculer les paramètres internes du modèle grâce à l'optimiseur dans notre cas. Nous allons le prouver en quelques lignes avec la méthode de la différentielle d'ordre 2.

Démonstration :

$$\forall p \in [0, 1], y \in \{0, 1, \dots, n-1\}$$

$$L(y, p) = - \sum_{i=1}^n y_i * \log(p_i)$$

$$\frac{\partial L}{\partial p_i} = -y_i / p_i$$

$$\frac{\partial^2 L}{\partial p_i^2} = y_i / p_i^2$$

$$\frac{\partial^2 L}{\partial p_i^2} \geq 0$$

La dérivée seconde pour tout i est supérieure ou égale à 0 et comme chacun des termes de la somme est positif pour tout i alors la fonction cross-entropy categorical est bien convexe.

Est-ce que cette fonction est concave ?

On sait que si $-f(x)$ est convexe, alors $f(x)$ est concave.

Appliquons $-f(x)$ à la fonction de loss précédente :

$$L(y, p) = \sum_{i=1}^n y_i * \log(p_i)$$

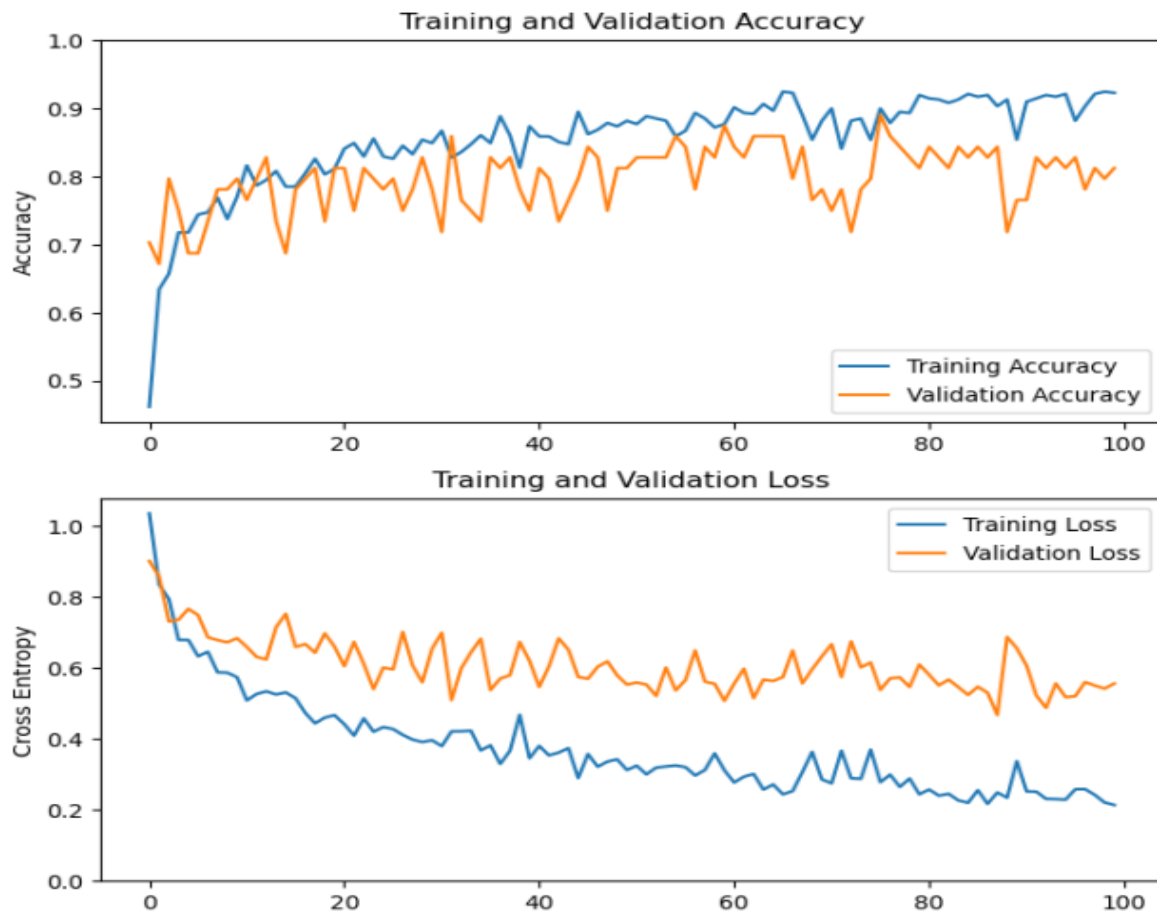
$$\frac{\partial L}{\partial p_i} = y_i / p_i$$

$$\frac{\partial^2 L}{\partial p_i^2} = -y_i / p_i^2$$

$$\frac{\partial^2 L}{\partial p_i^2} \leq 0$$

La dérivée seconde pour tout i est inférieure ou égale à 0, donc la fonction de cross-entropy categorical n'est pas concave.

2.5. Evolution de l'accuracy et de la valeur de la cross-entropy en fonction des epochs



On remarque qu'ici qu'on a un graphique correspond à l'entraînement jusqu'à 100 epochs (nombre d'itérations à travers l'ensemble des données du jeu d'entraînement) de notre réseau de neurones, il est tout à fait normal qu'au début les poids sont considérés de manière aléatoire mais au fil des epochs, on peut remarquer que notre fonction de coût baisse pour le jeu d'entraînement et qu'elle reste stable pour le jeu de validation, cela veut dire que le jeu de validation ne peut être mieux prédit à un certain degré d'entraînement de notre jeu d'entraînement. Il faut donc impérativement ne pas rendre la fonction de coût très faible pour l'entraînement car elle pourrait overfitter et rendre donc notre jeu de validation et notre jeu de test avoir une fonction de coût élevée, donc les images seront moins reconnaissables et performantes par notre modèle. On constate aussi que lorsque la fonction de coût baisse, la précision (accuracy) augmente, ce qui est tout à fait normal car la fonction de coût est la relation inverse de la précision. Quand une baisse, l'autre augmente et vice-versa. Le but d'un réseau de neurones étant de baisser le plus possible sans overfitter la fonction de coût pour avoir une précision (accuracy) la meilleure possible.

On obtient ici notre précision maximale à la 76e epoch pour une précision équivalente à 0.89 pour notre jeu de validation.

2.6. Matrice de confusion pour le meilleur et le plus faible modèle

On sélectionne donc ce réseau de neurones précédent auquel on va mesurer les prédictions de ce modèle sur un total de 114 photos mélangeant des paires de Converse, Nike et Adidas.

Voici notre matrice de confusion obtenant une loss de 0.54 pour notre jeu de validation (0.88 d'accuracy) et aussi de 0.24 pour notre jeu d'entraînement avec aussi 0.91 d'accuracy.

	Converse	Nike	Adidas
Converse	32	1	5
Nike	3	33	2
Adidas	4	5	29

Voici notre matrice de confusion avec un autre modèle obtenant une loss de 0.89 pour notre jeu de validation (0.62 d'accuracy) et aussi de 0.89 pour notre jeu d'entraînement avec aussi 0.62 d'accuracy.

	Converse	Nike	Adidas
Converse	15	16	7
Nike	6	28	4
Adidas	3	10	25

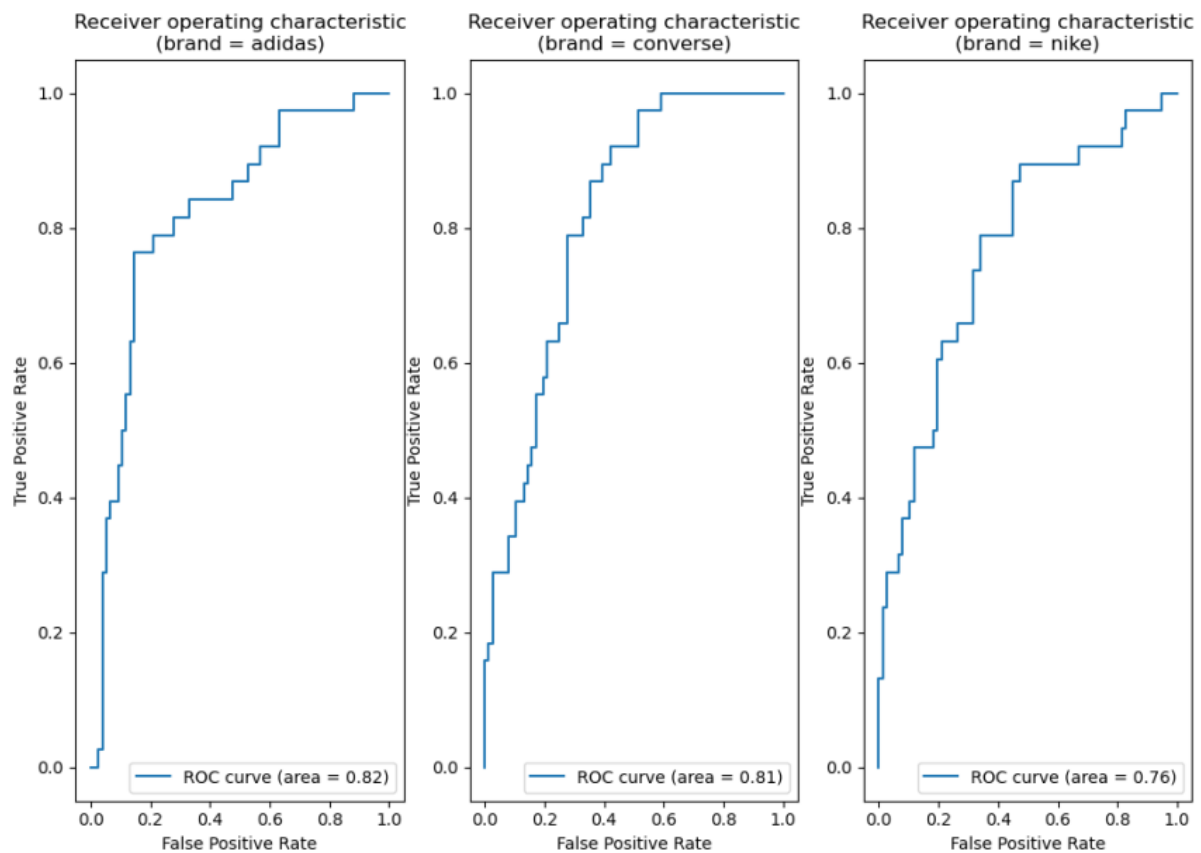
Quand on compare au modèle précédent, on remarque bien qu'ici que notre modèle n'a pas été assez entraîné. En effet les vrais prédictions (true predictions) passent de 94 à 68. (Somme des diagonales). Ce qui nous fait un écart relatif de -27.7%, donc c'est la raison pour laquelle il faut entraîner un bon nombre de données en augmentant les epochs mais à faire bien aussi des mini-batch pour que l'optimiseur Adam se fasse plus rapidement. Il ne faut pas négliger la loss car c'est grâce en partie à elle que l'apprentissage de nos paramètres internes se fait pour obtenir au fur et à mesure de meilleures précisions mais attention à ne pas trop l'optimiser dû à l'overfitting.

On obtient ici des score F1 de 0,824 pour le meilleur modèle et de 0,589 pour le plus faible modèle qui indiquent la capacité de chaque modèle à classer correctement les données en prenant en compte la précision et le rappel.

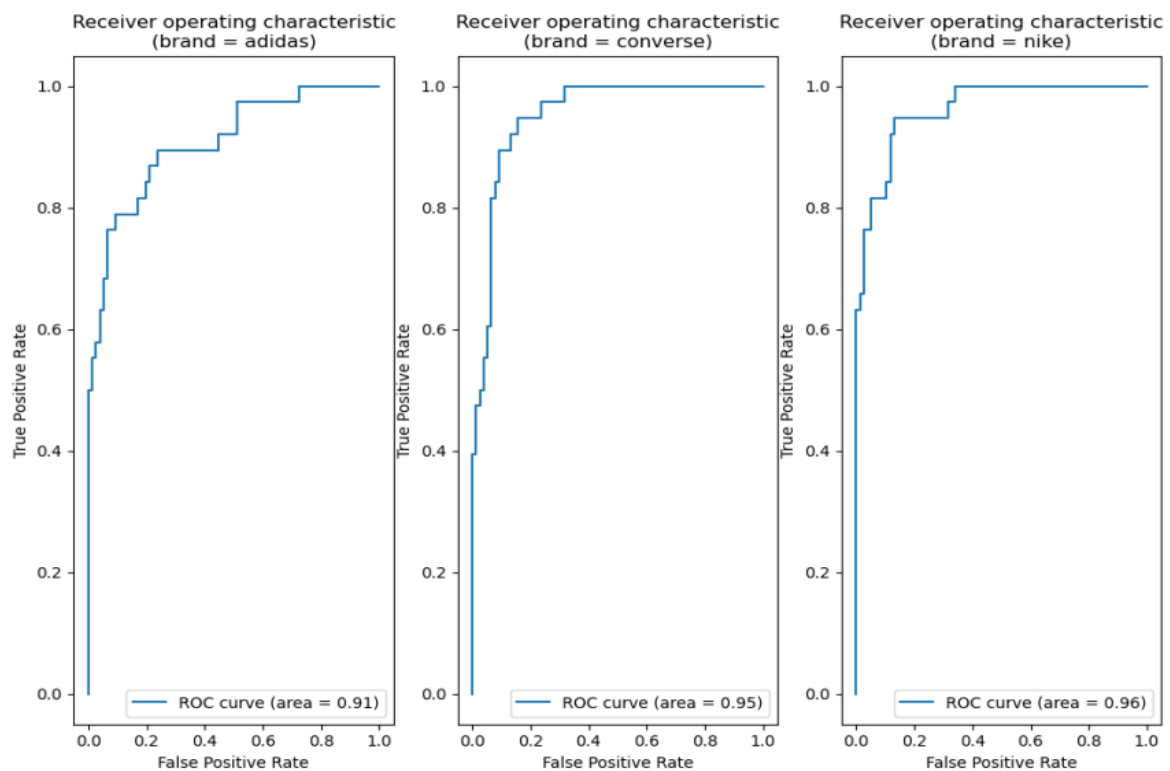
Un score F1 élevé indique que le modèle a un bon équilibre entre la précision et le rappel, c'est-à-dire qu'il arrive à reconnaître avec précision les bonnes prédictions tout en minimisant les faux positifs et les faux négatifs. En revanche, un score F1 faible indique que le modèle a des difficultés à bien classer certains exemples.

2.7. Courbe ROC pour le meilleur et le plus faible modèle

Courbe ROC du modèle ayant la loss la plus haute :



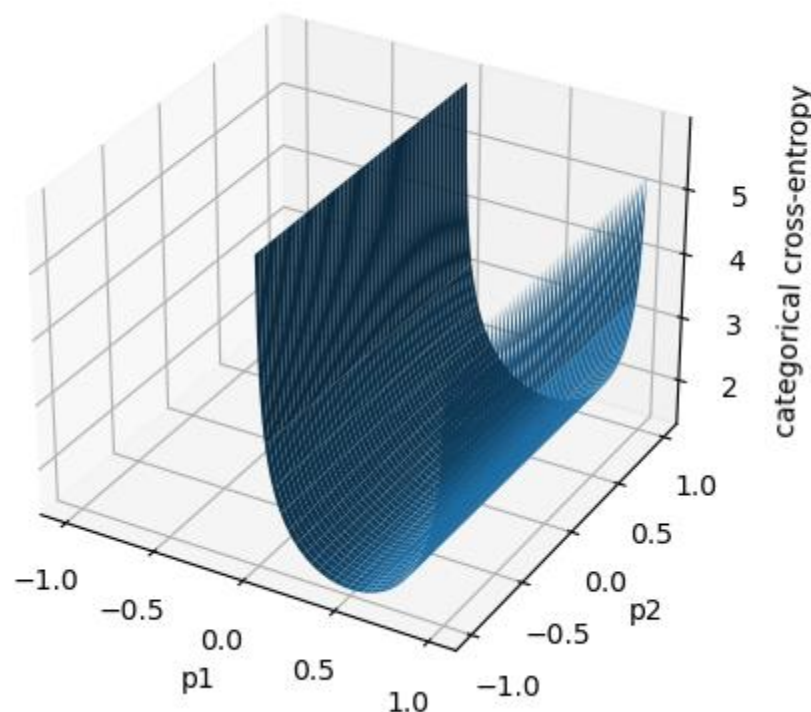
Courbe ROC du modèle ayant la loss la plus faible :



La courbe ROC (Receiver Operating Characteristic) décrit le taux de faux positifs (specificity) en abscisses et le taux de vrais positifs (sensitivity) en ordonnées selon plusieurs seuils de classification définis au préalable par l'algorithme. Elle permet de calculer la qualité d'un modèle de classification. En effet, plus le modèle va savoir différencier les labels positifs et négatifs en fonction de plusieurs seuils, plus l'AUC (Area Under the ROC Curve) sera élevée : c'est un indicateur très utile pour comparer des modèles qui ont un peu près la même accuracy et la même valeur de loss.

On constate que dans notre modèle OvR (One vs Rest) avec la valeur de loss la plus faible, on est quand même à plus de 90% de réussite, donc il y a environ plus de 90% de chances que notre modèle sache reconnaître laquelle correspond à une paire de chaussures et laquelle ne l'est pas.

2.8. Aperçu 3D de la categorical cross-entropy



3. Partie Code

3.1. Bibliothèques importées



3.1.1. TensorFlow

TensorFlow est une bibliothèque open-source d'apprentissage automatique et de calcul numérique développée par Google Brain Team. Elle est utilisée pour créer des modèles d'apprentissage automatique, tels que des réseaux de neurones, et pour résoudre des problèmes dans des domaines tels que la vision par ordinateur, la reconnaissance de la parole, la traduction automatique et la prédiction.

TensorFlow est conçu pour être hautement extensible et modulaire, ce qui facilite la création de modèles personnalisés et l'adaptation de modèles existants. Il est également conçu pour fonctionner efficacement sur des systèmes distribués, ce qui permet de traiter de grandes quantités de données et de faire des prédictions en temps réel.

TensorFlow prend en charge de nombreux langages de programmation, notamment Python, C++, Java et Go, ce qui le rend accessible à un large éventail de développeurs.

3.1.2. Keras

Keras est une bibliothèque open-source d'apprentissage profond et de réseaux de neurones, qui est également développée par Google. Keras est conçu pour être facile à utiliser et à comprendre, tout en offrant une grande flexibilité pour créer des modèles d'apprentissage profond personnalisés.

Keras est une interface d'application de programmation (API) qui permet aux développeurs de créer des modèles d'apprentissage profond avec une syntaxe simple et intuitive. Il permet également de créer rapidement des prototypes et de tester différents modèles, en fournissant des modules pré-construits pour la création de couches de réseau de neurones, la régularisation, la normalisation et autres opérations courantes.

Keras peut être utilisé avec différentes bibliothèques de calcul numérique telles que TensorFlow, Theano et Microsoft Cognitive Toolkit. En utilisant l'un de ces frameworks de calcul, Keras permet de déployer facilement des modèles sur une variété de plateformes, notamment les ordinateurs de bureau, les serveurs et les appareils mobiles.

3.1.3. Sous librairies Keras

Le module **InceptionV3** contient le modèle InceptionV3 pré-entraîné qui peut être utilisé pour l'apprentissage par transfert.

Le module **ImageDataGenerator** vous permet d'effectuer une augmentation de données et un prétraitement sur vos données d'image.

Le module **Sequential** vous permet de définir une séquence de couches pour votre modèle d'apprentissage en profondeur.

Le module **Dense** est utilisé pour définir une couche entièrement connectée dans votre modèle d'apprentissage en profondeur.

Le module **Dropout** est utilisé pour abandonner de manière aléatoire certains neurones de votre modèle d'apprentissage en profondeur pendant l'entraînement pour éviter le surapprentissage.

Le module **GlobalAveragePooling2D** est utilisé pour réduire les dimensions des cartes de caractéristiques des couches de convolution de votre modèle d'apprentissage en profondeur.

Le module **optimizers** contient des algorithmes d'optimisation qui peuvent être utilisés pour entraîner votre modèle d'apprentissage en profondeur.

Le module **ModelCheckpoint** vous permet de sauvegarder les poids de votre modèle d'apprentissage en profondeur pendant l'entraînement.

Le module **image** contient des fonctions pour charger et prétraiter les données d'image.

3.2. Chargement des Images pour le Train et le Test

La classe **ImageDataGenerator** de Keras permet de créer des générateurs de données en temps réel pour l'entraînement et la validation d'un modèle de réseau de neurones convolutionnels (CNN) à partir de données d'images stockées dans un répertoire.

```
datagenerator = {  
    "train": ImageDataGenerator(horizontal_flip=True,  
                                vertical_flip=True,  
                                rescale=1. / 255,  
                                validation_split=0.1,  
                                shear_range=0.1,  
                                zoom_range=0.1,  
                                width_shift_range=0.1,  
                                height_shift_range=0.1,  
                                rotation_range=30,  
                                ).flow_from_directory(directory=base_dir,  
                                                    target_size=(300, 300),  
                                                    subset='training',  
                                                    ),  
    "valid": ImageDataGenerator(rescale=1 / 255,  
                                validation_split=0.1,  
                                ).flow_from_directory(directory=base_dir,  
                                                    target_size=(300, 300),  
                                                    subset='validation',  
                                                    ),  
}
```

Les paramètres utilisés dans la configuration de l'objet **ImageDataGenerator** sont les suivants :

- **horizontal_flip=True** et **vertical_flip=True** : cela permet d'effectuer une augmentation de données (data augmentation) en effectuant des retournements horizontaux et verticaux aléatoires sur les images d'entraînement.
- **rescale=1./255** : cela permet de normaliser les valeurs de pixels des images en les divisant par 255, afin que les valeurs se situent entre 0 et 1.
- **validation_split=0.1** : cela divise les données en ensembles d'entraînement et de validation avec une proportion de 90 % pour l'ensemble d'entraînement et 10 % pour l'ensemble de validation.
- **shear_range=0.1, zoom_range=0.1, width_shift_range=0.1, height_shift_range=0.1** et **rotation_range=30** : ces paramètres permettent de réaliser d'autres augmentations de données en effectuant des rotations, des déformations, des zooms, et des translations aléatoires sur les images d'entraînement.
- **flow_from_directory()** : cette méthode crée un générateur de lots d'images en temps réel à partir des images stockées dans un répertoire, en prenant en compte les différentes transformations de données spécifiées.

L'objet **datagenerator** créé contient deux générateurs de lots d'images : un pour l'ensemble d'entraînement et un pour l'ensemble de validation. Ces générateurs peuvent être utilisés pour alimenter un modèle de réseau de neurones CNN pendant l'entraînement et la validation.

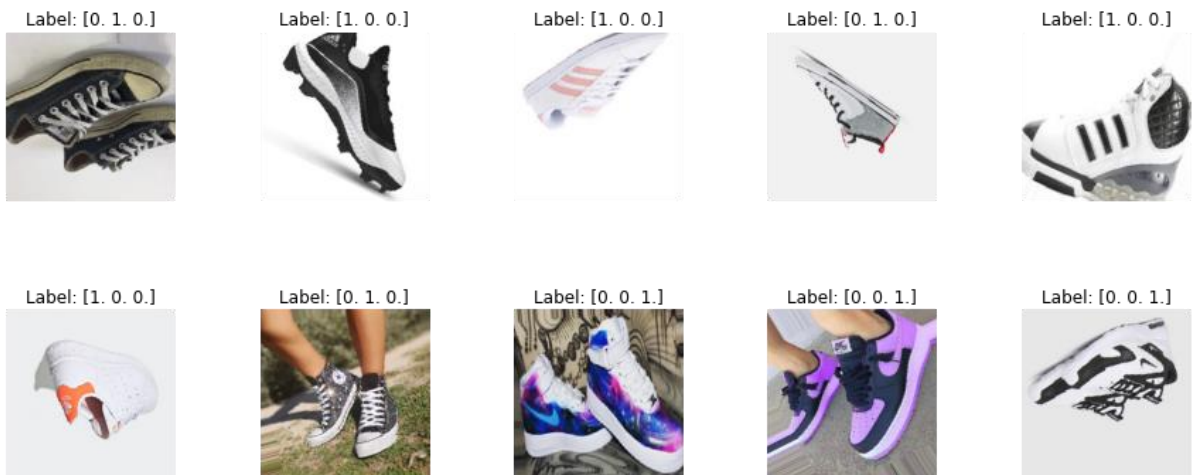
Sélection aléatoire d'images du **datagenerator** puis affichage de ces dernières :

```
import matplotlib.pyplot as plt

# extraire un lot d'images d'entraînement
batch_images, batch_labels = datagenerator["train"].next() #batch_images array d'images

# afficher les images transformées
fig, axs = plt.subplots(2, 5, figsize=(15, 6))
fig.subplots_adjust(hspace=.5, wspace=.5)
axs = axs.ravel()
for i in range(10):
    axs[i].imshow(batch_images[i])
    axs[i].set_title(f"Label: {batch_labels[i]}")
    axs[i].axis('off')
plt.show()
```

Affichage :



3.3. Initialisation du Modèle Pré-Entraîné InceptionV3

Ici, on crée une instance de modèle InceptionV3 à l'aide de la fonction **InceptionV3()**. Cette fonction crée un modèle pré-entraîné InceptionV3, initialement entraîné sur ImageNet, avec les poids chargés à partir de weights.

```
# Initializing InceptionV3 (pretrained) model with input image shape as (300, 300, 3)
base_model = InceptionV3(weights=None, include_top=False, input_shape=(300, 300, 3))
```

- **weights=None**, on initialise les poids aléatoirement.
- **include_top** contrôle si on souhaite inclure les couches de sortie du modèle. Définis sur False, les layers de classification d'InceptionV3 sont exclues et on pourra ajouter nos propres layers de sortie pour une tâche de classification différente.
- **input_shape** spécifie la forme de l'entrée du modèle, qui est généralement une image de taille (largeur, hauteur, canaux). Dans ce cas, la taille d'entrée est (300, 300, 3), ce qui signifie que l'image d'entrée aura une largeur et une hauteur de 300 pixels et trois canaux pour les couleurs Rouge, Vert et Bleu.

En résumé, cette ligne de code crée une instance du modèle InceptionV3 non-pré-entraîné, sans les couches de sortie et avec une taille d'entrée de (300, 300, 3). On pourra ensuite entraîner le modèle sur nos propres données et ajouter nos propres layer de sortie.

Chargement de poids pré-entraînés du modèle InceptionV3 à partir d'un fichier h5

```
# Load Weights for the InceptionV3 Model
base_model.load_weights('inception_v3/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5')
```

Ensuite, nous avons défini **base_model.trainable** à False pour empêcher l'entraînement des layers du modèle de base lors de l'entraînement de notre propre modèle. Cela signifie que les poids pré-entraînés dans le modèle de base resteront fixes pendant l'entraînement de notre modèle personnalisé, et seules les layers que nous avons ajoutées seront entraînées pour adapter le modèle à notre tâche spécifique.

```
# Setting the Training of all layers of InceptionV3 model to false
base_model.trainable = False
```

Cela permet de réduire le temps d'entraînement et d'éviter la détérioration des performances des layers pré-entraînées.

3.4. Ajout d'un Layer Personnalisé

Ajout de couches supplémentaires à la fin du modèle pré-entraîné **base_model**.

```
# Adding some more layers at the end of the Model as per our requirement
model = Sequential([
    base_model,
    GlobalAveragePooling2D(),
    Dropout(0.15),
    Dense(1024, activation='relu'),
    Dense(3, activation='softmax') # 3 Output Neurons for 3 Classes
])
```

La première couche que nous avons ajoutée est **GlobalAveragePooling2D()**, qui calcule la moyenne des valeurs de chaque canal de chaque carte de fonctionnalité en sortie du modèle pré-entraîné, ce qui réduit le nombre de paramètres à entraîner et permet de mieux généraliser le modèle à de nouvelles données.

Ensuite, nous avons ajouté une couche **Dropout(0.15)**, qui désactive de manière aléatoire certaines unités d'entrée du réseau pendant l'entraînement, ce qui aide à prévenir le surapprentissage.

Ensuite, nous avons ajouté une couche **Dense(1024, activation='relu')**, qui est une couche entièrement connectée avec 1024 neurones et une fonction d'activation ReLU.

Enfin, nous avons ajouté une couche de sortie **Dense(3, activation='softmax')** avec 3 neurones correspondant aux 3 classes de notre tâche de classification d'images. La fonction d'activation softmax normalise les valeurs de sortie en une distribution de probabilité sur les 3 classes, ce qui nous permet d'interpréter les sorties du modèle comme des probabilités pour chaque classe.

3.5. Visualisation Sommaire du Modèle

Ce résumé montre l'architecture d'un modèle séquentiel avec quatre couches : **inception_v3**, **global_average_pooling2d**, **dropout** et deux couches **denses**.

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
inception_v3 (Functional)	(None, 8, 8, 2048)	21802784
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 2048)	0
dropout_1 (Dropout)	(None, 2048)	0
dense_2 (Dense)	(None, 1024)	2098176
dense_3 (Dense)	(None, 3)	3075

```

=====
Total params: 23,904,035
Trainable params: 2,101,251
Non-trainable params: 21,802,784
=====

```

- La première couche est un modèle d'**inception_v3** préformé avec des paramètres non-entraînés, qui prend une entrée de forme (None, 300, 300, 3) et produit un tenseur de forme (None, 8, 8, 2048). Cela signifie que la sortie de la couche inception_v3 a des cartes de fonctions 8x8, chacune avec 2048 canaux.
- La deuxième couche est une couche **GlobalAveragePooling2D**, qui calcule la moyenne des cartes de caractéristiques le long des dimensions spatiales (hauteur et largeur) du tenseur, résultant en un tenseur de forme (None, 2048).
- La troisième couche est une couche **drop**, qui laisse tomber au hasard une fraction des unités d'entrée pendant l'entraînement pour éviter le débordement.
- La quatrième couche est une couche **dense** avec 1024 unités et la fonction d'activation ReLU. Cette couche a 2.098.176 paramètres de formation.
- La cinquième et dernière couche est une autre couche **dense** avec 3 unités et une fonction d'activation softmax, qui est utilisée pour les tâches de classification multi-classes. Cette couche a 3 paramètres de formation.

Le nombre total de paramètres dans le modèle est de 23.904.035, dont seulement 2.101.251 sont entraînables. Les 21 802 784 paramètres restants sont non entraînables, provenant du modèle d'inception_v3 préformé.

3.6. Entraînement du modèle

Cette partie est utilisée pour entraîner le modèle de réseau de neurones convolutifs en utilisant les générateurs de données d'entraînement et de validation, qui ont été créés précédemment à l'aide de **ImageDataGenerator**.

La fonction **fit_generator** prend plusieurs paramètres, notamment :

- **generator** : le générateur de données d'entraînement
- **epochs** : le nombre d'itérations sur les données d'entraînement
- **steps_per_epoch** : le nombre de lots à extraire du générateur d'entraînement à chaque époque
- **validation_data** : le générateur de données de validation
- **validation_steps** : le nombre de lots à extraire du générateur de validation à chaque époque
- **callbacks_list** : une liste d'objets de rappel qui sont appelés à différents moments de l'entraînement pour effectuer des tâches telles que la sauvegarde de modèles, l'arrêt précoce et l'ajustement du taux d'apprentissage.

Pendant l'entraînement, le modèle effectue des itérations sur les lots d'images d'entraînement et ajuste les poids des différentes couches pour minimiser la fonction de perte. Le générateur de données d'entraînement et de validation fournit des lots d'images de manière itérative, ce qui permet à l'entraînement de se poursuivre jusqu'à ce que le nombre d'époques spécifié soit atteint. Les métriques d'entraînement et de validation telles que la précision et la perte sont également calculées à chaque époque. Les objets de rappel spécifiés sont appelés à différents moments pendant l'entraînement pour effectuer des tâches telles que la sauvegarde de modèles, l'arrêt précoce et l'ajustement du taux d'apprentissage.

```
# File Path to store the trained models
filepath = "model/model_{epoch:02d}-{val_accuracy:.2f}.h5"

# Using the ModelCheckpoint function to train and store all the best models
checkpoint1 = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1, save_best_only=True, mode='max')

callbacks_list = [checkpoint1]
# Training the Model
history = model.fit_generator(generator=train_generator, epochs=epochs, steps_per_epoch=steps_per_epoch,
                             validation_data=valid_generator, validation_steps=validation_steps,
                             callbacks=callbacks_list)
```

Pour chaque **epochs**, on surveille si la **val_accuracy** du modèle simulé est plus grande que toute celle des modèles précédemment créés, si oui on crée le modèle dans notre répertoire :

```
Output exceeds the size limit. Open the full output data in a text editor
Epoch 1/10
20/20 [=====] - ETA: 0s - loss: 1.1109 - accuracy: 0.4082
Epoch 1: val_accuracy improved from -inf to 0.70312, saving model to model\model_01-0.70.h5
20/20 [=====] - 198s 10s/step - loss: 1.1109 - accuracy: 0.4082 - val_loss: 0.9387 - val_accuracy: 0.7031
Epoch 2/10
20/20 [=====] - ETA: 0s - loss: 0.9078 - accuracy: 0.5738
Epoch 2: val_accuracy improved from 0.70312 to 0.75000, saving model to model\model_02-0.75.h5
20/20 [=====] - 188s 9s/step - loss: 0.9078 - accuracy: 0.5738 - val_loss: 0.8536 - val_accuracy: 0.7500
Epoch 3/10
20/20 [=====] - ETA: 0s - loss: 0.8715 - accuracy: 0.5918
Epoch 3: val_accuracy did not improve from 0.75000
20/20 [=====] - 162s 8s/step - loss: 0.8715 - accuracy: 0.5918 - val_loss: 0.7893 - val_accuracy: 0.7031
Epoch 4/10
20/20 [=====] - ETA: 0s - loss: 0.7477 - accuracy: 0.6836
Epoch 4: val_accuracy did not improve from 0.75000
20/20 [=====] - 155s 8s/step - loss: 0.7477 - accuracy: 0.6836 - val_loss: 0.9306 - val_accuracy: 0.5469
Epoch 5/10
20/20 [=====] - ETA: 0s - loss: 0.7209 - accuracy: 0.6803
Epoch 5: val_accuracy did not improve from 0.75000
20/20 [=====] - 157s 8s/step - loss: 0.7209 - accuracy: 0.6803 - val_loss: 0.8088 - val_accuracy: 0.6406
Epoch 6/10
20/20 [=====] - ETA: 0s - loss: 0.6801 - accuracy: 0.7246
Epoch 6: val_accuracy did not improve from 0.75000
20/20 [=====] - 154s 8s/step - loss: 0.6801 - accuracy: 0.7246 - val_loss: 0.7821 - val_accuracy: 0.7188
Epoch 7/10
...
Epoch 10/10
20/20 [=====] - ETA: 0s - loss: 0.5714 - accuracy: 0.7639
Epoch 10: val_accuracy improved from 0.76562 to 0.78125, saving model to model\model_10-0.78.h5
20/20 [=====] - 154s 8s/step - loss: 0.5714 - accuracy: 0.7639 - val_loss: 0.6569 - val_accuracy: 0.7812
```

3.7. Evaluation du modèle

Premièrement, on va charger le modèle avec l'accuracy la plus haute parmi ceux générés précédemment dans la variable **loaded_best_model**:

```
# Check our folder and import the model with best validation accuracy
loaded_best_model = keras.models.load_model("model/model_10-0.78.h5")
```

Ensuite dans cette première partie de la fonction **predict()** on va afficher dans le terminal l'image de la paire de basket à partir de l'URL renseigné en paramètre. L'ouverture de l'image est pris en charge par la librairie PIL et non par keras (la fonction **load_img()** étant devenu obsolète entre-temps). On affiche ensuite la prédiction du label (la marque de basket) faite par **loaded_best_model** ainsi que le taux de confiance de cette prédiction.

```
# Custom function to load and predict label for the image
def predict(img_rel_path):
    # Import Image from the path with size of (300, 300)
    img = Image.open(img_rel_path)
    img = img.resize((300, 300))

    # Convert Image to a numpy array
    img = np.array(img)

    # Scaling the Image Array values between 0 and 1
    img = np.array(img)/255.0

    # Plotting the Loaded Image
    plt.title("Loaded Image")
    plt.axis('off')
    plt.imshow(img.squeeze())
    plt.show()

    # Get the Predicted Label for the loaded Image
    p = loaded_best_model.predict(img[np.newaxis, ...])

    # Label array
    labels = {0: 'adidas', 1: 'converse', 2: 'nike'}

    print("\n\nMaximum Probability: ", np.max(p[0], axis=-1))
    predicted_class = labels[np.argmax(p[0], axis=-1)]
    print("Classified:", predicted_class, "\n\n")

    classes=[]
    prob=[]
```

Enfin, dans cette seconde partie de la fonction **predict()**, on affiche l'historgramme représentant les probabilités pour chaque label :

```
print("\n-----Individual Probability-----\n")

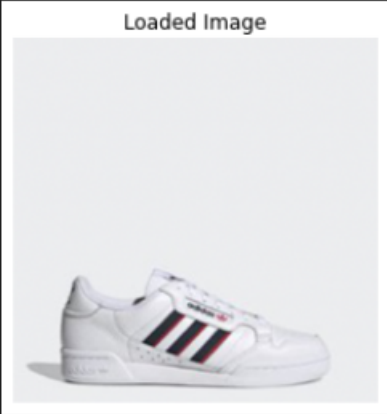
for i,j in enumerate (p[0],0):
    print(labels[i].upper(),':',round(j*100,2),'%')
    classes.append(labels[i])
    prob.append(round(j*100,2))

def plot_bar_x():
    # this is for plotting purpose
    index = np.arange(len(classes))
    plt.bar(index, prob)
    plt.xlabel('Labels', fontsize=8)
    plt.ylabel('Probability', fontsize=8)
    plt.xticks(index, classes, fontsize=8, rotation=20)
    plt.title('Probability for loaded image')
    plt.show()
plot_bar_x()
```


3.8. Tests sur le modèle avec des graphes

Appel de la fonction **predict()** avec le chemin de l'image, affichage de l'image chargée, prédiction et taux de confiance de la prédiction :

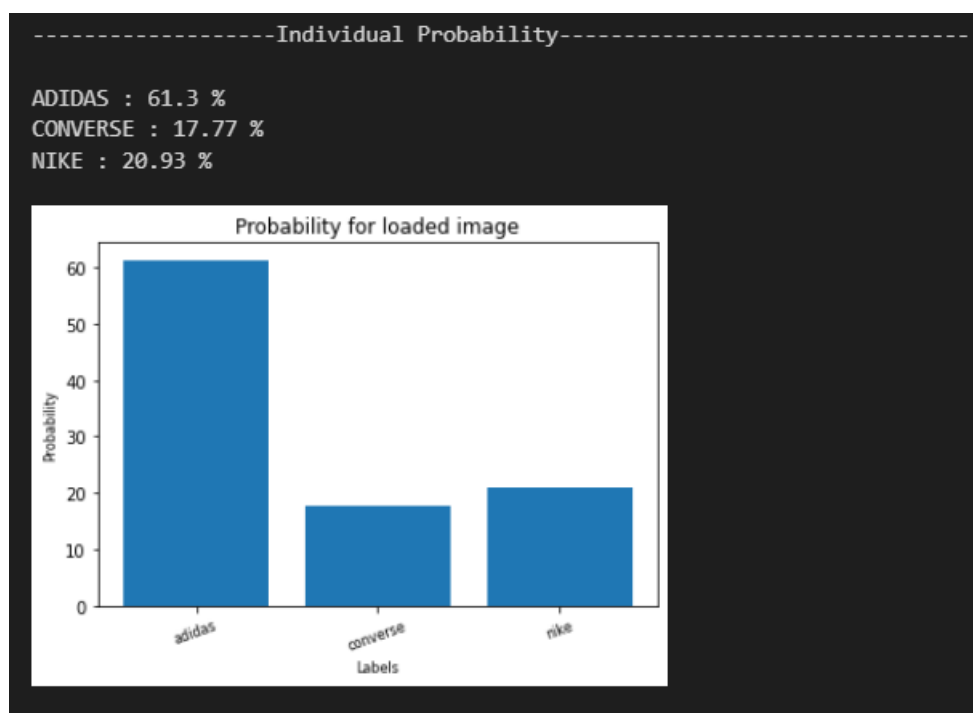
```
predict("input/validation/adidas/adidas_ (149).jpg")
```



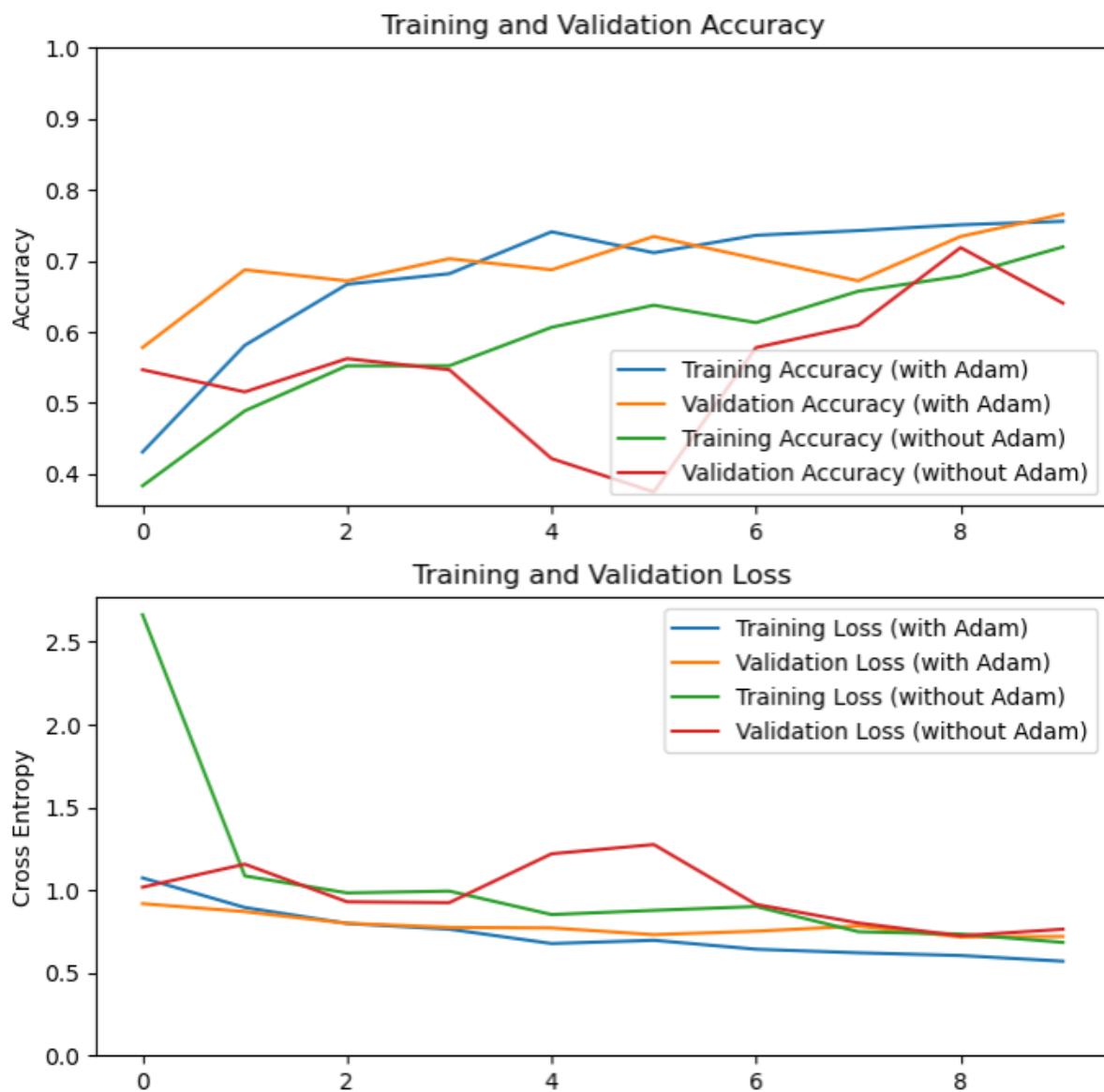
```
1/1 [=====] - 5s 5s/step
```

```
Maximum Probability: 0.613032  
Classified: adidas
```

Histogramme des probabilités pour chaque label :



3.9. Comparaison avec et sans optimiseur Adam



Pour montrer l'efficacité de l'optimiseur Adam, nous avons tracé un plot comparant la loss et l'accuracy au fil de dix itérations sur les ensembles de train et test de deux modèles : l'un avec Adam un autre sans. On observe en général que le maximum (resp. minimum) pour l'accuracy (resp. loss) est trouvé bien avant dans l'itération pour le modèle muni d'un optimiseur, par rapport à celui sans.

4. Ouverture

Nous avons vu comment un optimisateur comme Adam permet d'optimiser les poids d'un réseau de neurones pour la classification d'images.

De plus, on pourrait s'intéresser à comment optimiser les hyperparamètres utilisés avant d'entraîner ce modèle tels que le Learning rate, β_1 et β_2 (utilisés par l'optimiseur Adam), qui dans notre cas ont été choisis par défaut. La Library keras tuner permet d'effectuer ce travail.