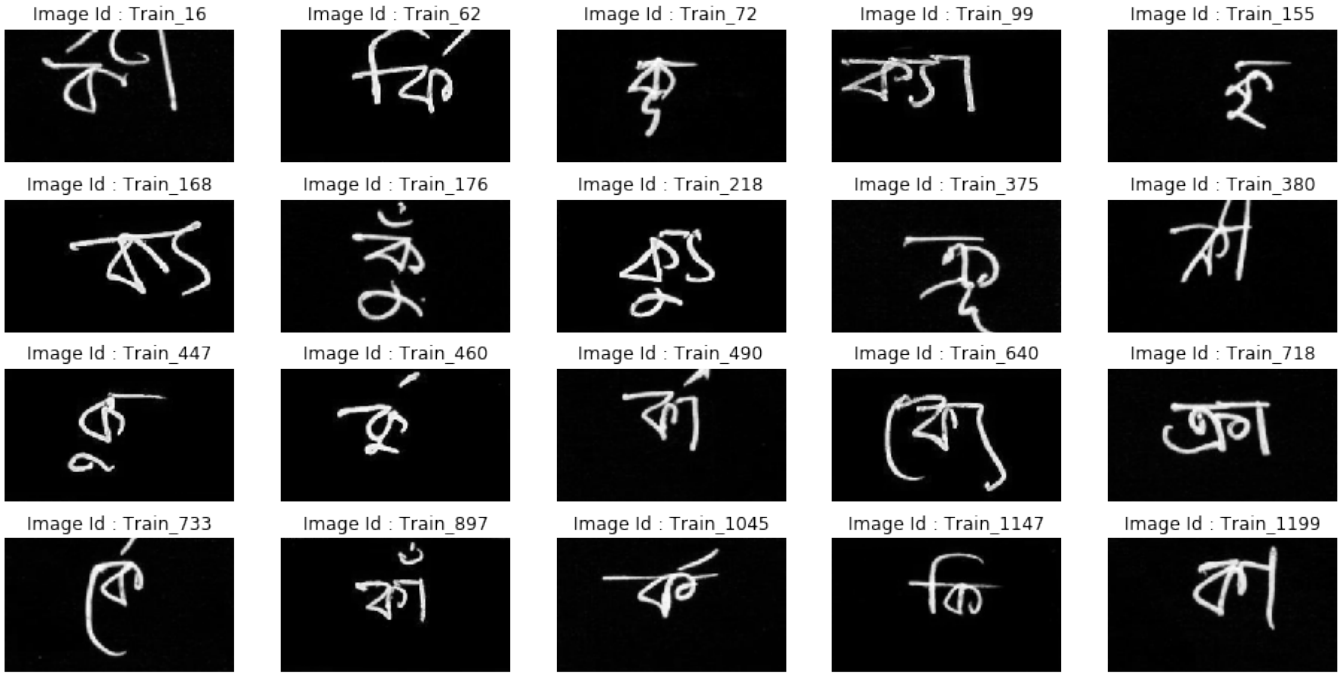
Projet 8 : Participation à une compétition Kaggle

Bengali.AI Handwritten Grapheme Classification



Auteur : Antoine Chesnais
Date dernière version : 19/02/2020

Projet réalisé dans le cadre de la formation
« Ingénieur Machine Learning » d'Openclassrooms.

Table des matières

I. Introduction

A. Contexte

La compétition lancée sur Kaggle par l'association Bengali.AI a pour but la reconnaissance visuelle de caractères Bengali (langue du Bangladesh). Le langage Bengali est complexe, composé de 50 lettres (39 consonnes et 11 voyelles) et 118 autres consonnes conjointes (assemblages de consonnes) utilisées comme racine d'un graphème (un graphème est la plus petite unité dans un système d'écriture). En plus de ces 168 racines de base (grapheme root) peuvent se greffer sur celles ci des « accentuations » (diacritics) : 11 « vowel diacritic » et 8 « consonant diacritic ».

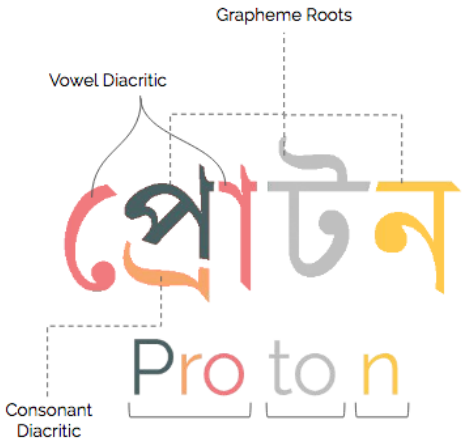


Figure 1: Exemple de mot Bengali

Le nombre de combinaisons entre ces 168 racines et accentuations est d'environ 13 000, mais elles ne sont pas toutes possibles . Nos données contiennent 1 292 combinaisons différentes (soir 1292 graphèmes différents) utilisées couramment dans le langage Bengali.

Le but du projet est donc à partir d'une banque de données de 200 580 images de graphèmes écrits d'apprendre à détecter les différentes possibilités pour chacune des trois classes : grapheme root (168), vowel diacritic (11) et consonant diacritic (7).

A noter que les modèles sont évalués avec un métrique spécifique qui est une moyenne pondérée du rappel obtenu sur les 3 cibles, 'grapheme root' comptant double par rapport aux deux autres.

B. Les données

- 4 fichiers parquet contenant les images (137 x 236 pixels) du train set
- Un dataframe contenant les cibles du train set
- Un dataframe mappant les différentes classes des composantes
- Le set de test est constitué de 4 fichiers parquet contenant les images mais seuls 3 échantillons par fichier nous sont accessibles directement.
- Avant d'être utilisées pour la modélisation, les images ont subi un cropping centré sur le grapheme et mise à une taille de 80 x 80 pixels

C. Ressources

Les ressources suivantes ont été utilisées pour mener à bien le projet :

- Kernels Kaggle GPU (Tesla P100, 16Gb RAM)
- Système de versionning Kaggle, permettant de suivre les versions de son code et de créer différentes branches de ses notebooks.
- Le Kernel de Kaushal Shah pour avoir une structure de travail de base
- Le Kernel de Maxime Lenormand pour le preprocessing des données

II. Analyse exploratoire

Dans cette partie seront présentés les éléments intéressants issus de l'analyse exploratoire des données permettant d'avoir une meilleure compréhension de celles ci.

Nos données sont composées de 200 580 images de taille 137 x 236 et nous disposons également de trois variables cibles pour chaque image (grapheme root, vowel diacritic et consonant diacritic). Il existe dans notre set d'entrainement 1292 combinaisons possibles de ces trois variables.

A. Distribution des données

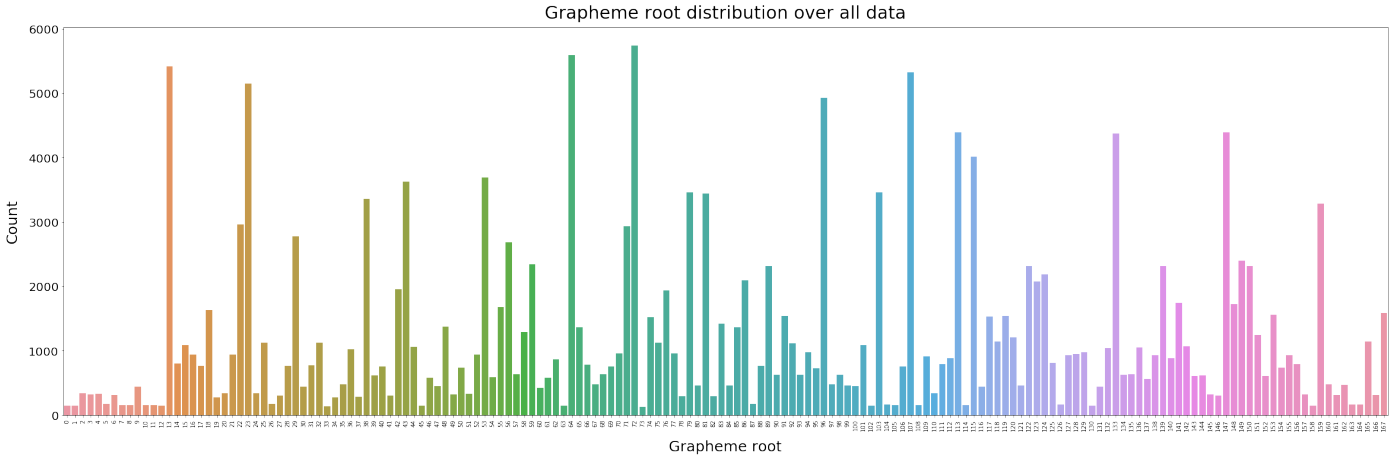


Figure 2: Distribution des 'Grapheme root'

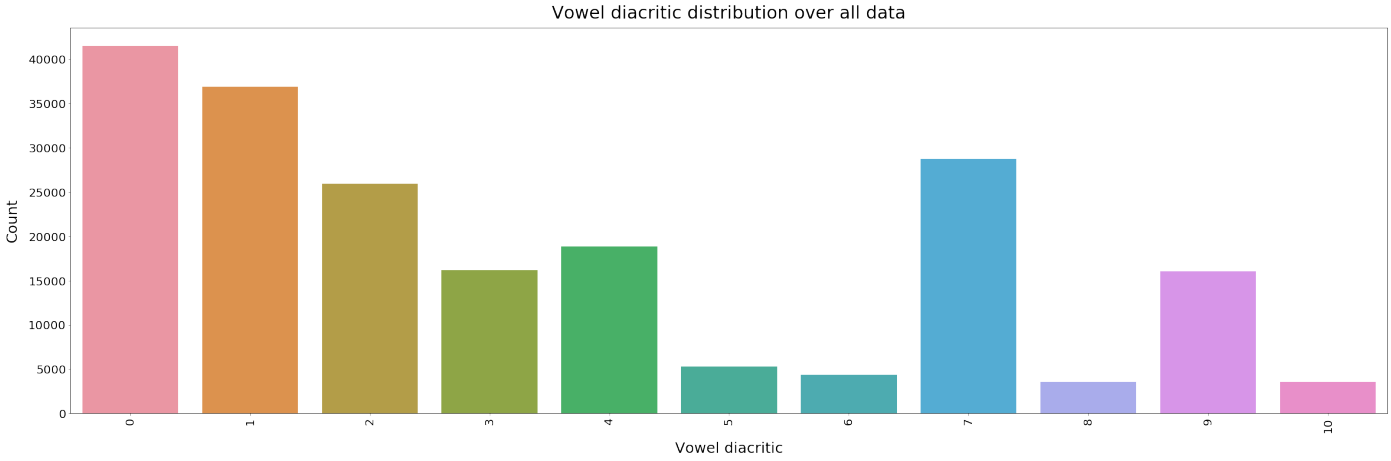


Figure 3: Distribution des 'vowel diacritic'

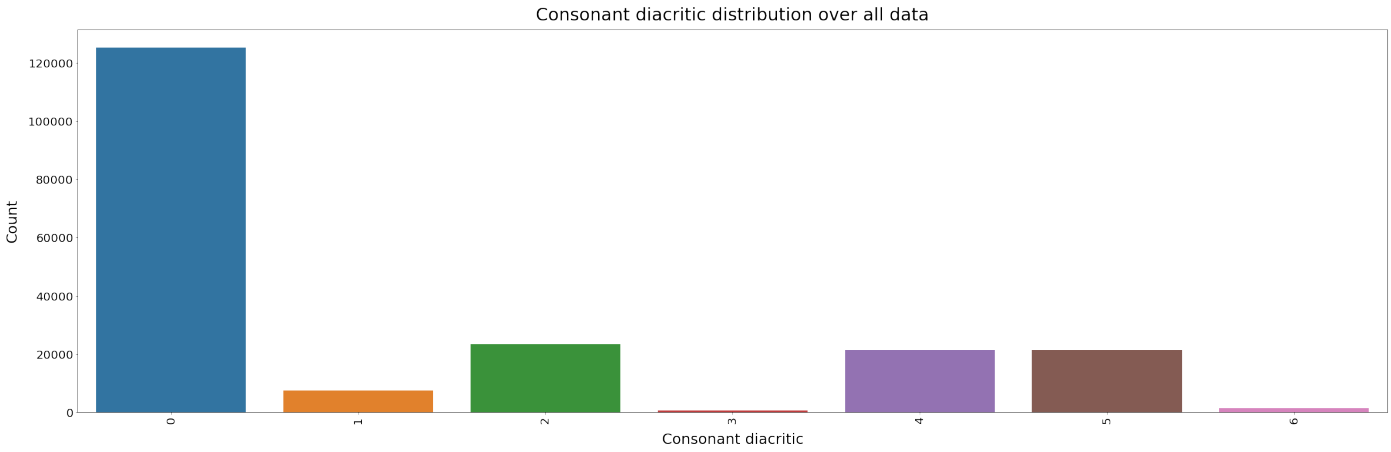


Figure 4: Distribution des 'consonant diacritic'

Quelle que soit la composante, les distributions sont très déséquilibrées.

B. Classes les plus représentées par composante

Afin d'avoir une idée de ce à quoi ressemblent les différentes composantes, on peut regarder les classes les plus représentées au sein de chacune d'entre elles (affichage des « consonant diacritics » non disponibles) :

Figure 5: Top 10 'Grapheme root' dans le train set

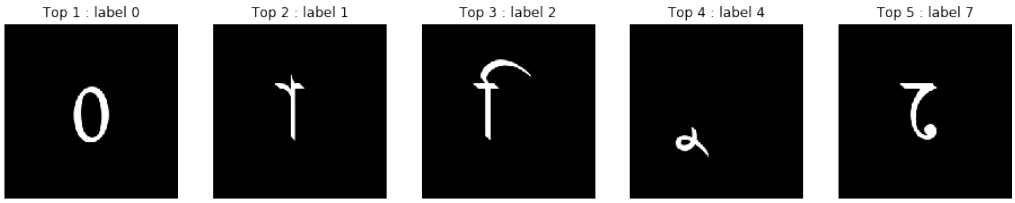


Figure 6: Top 5 'vowel diacritic' dans le train set

C. Exemple de variantes pour un racine de graphème

Ci dessous un exemple de différents graphèmes construit avec la racine la plus répandue (13). On voit que selon l'écriture et les autres composantes sa détection peut être parfois complexe.

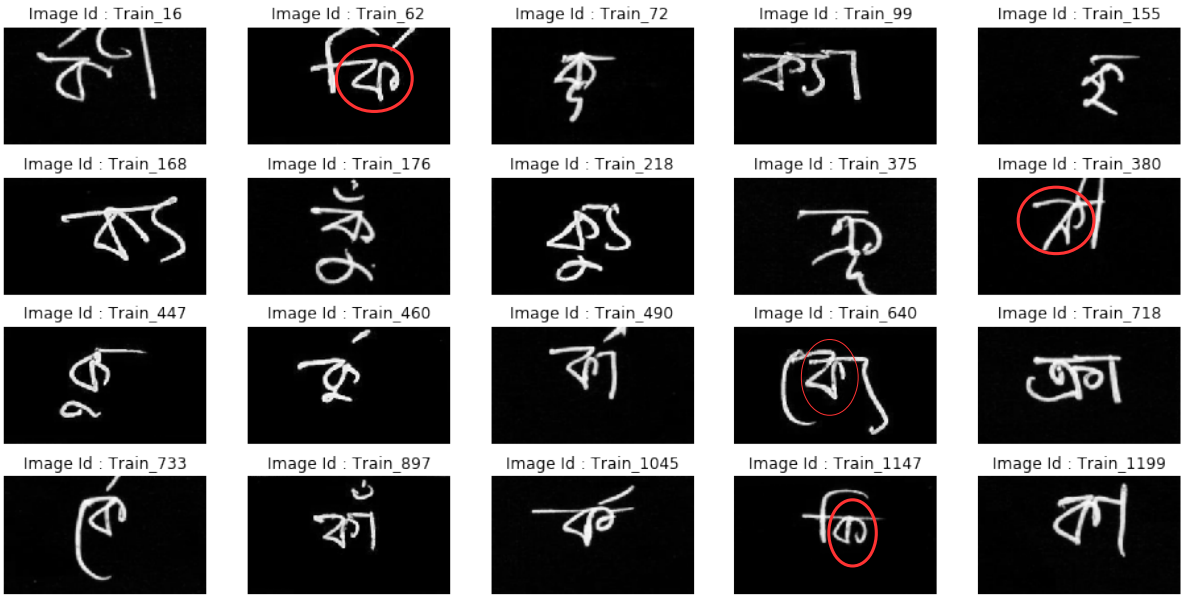


Figure 7: Exemple de combinaisons du 'grapheme root' 13 avec d'autres composantes

III. Modélisation

A. Modèle de base

1. Structure de base

Pour une première approche, un modèle basé sur la structure ci dessous a été défini :

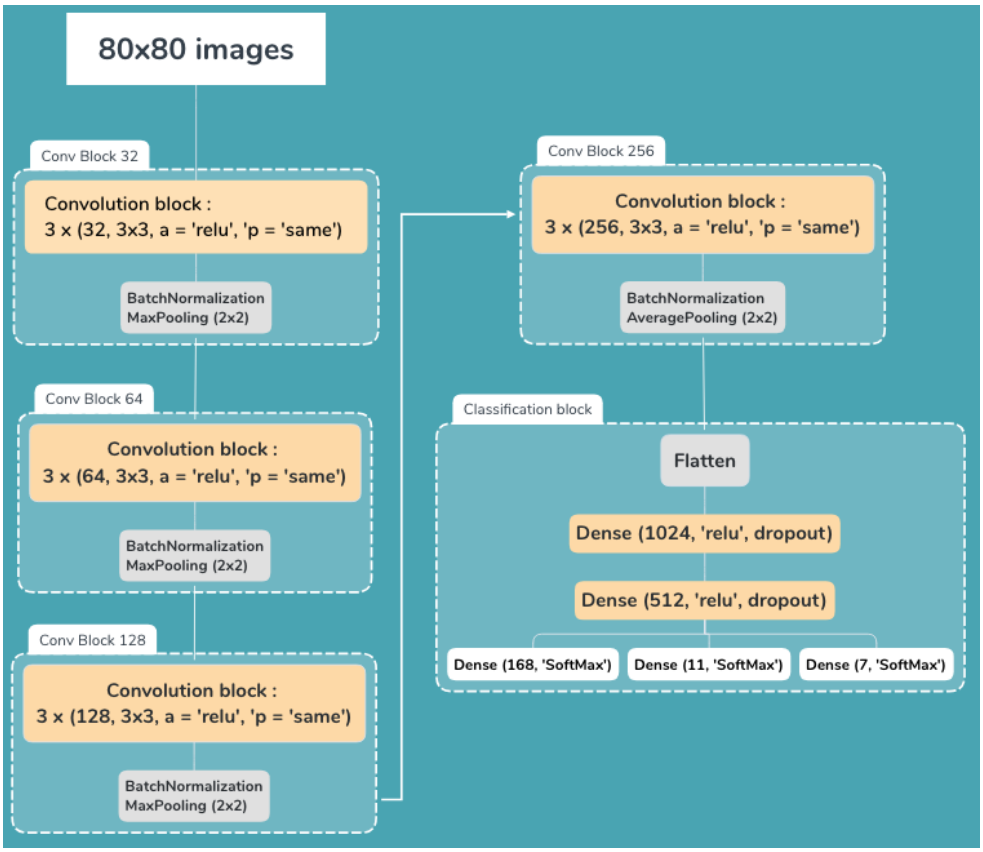


Figure 8: Détails de la structure du CNN de base

Il est composé des « blocs » suivants :

- **« Conv Block 32 »** : 3 couches de convolution à 32 filtres de taille 3x3 avec activation 'relu', suivies d'une BatchNomalization et d'un MaxPooling 2x2
- **« Conv Block 64 »** : 3 couches de convolution à 64 filtres de taille 3x3 avec activation 'relu', suivies d'une BatchNomalization et d'un MaxPooling 2x2
- **« Conv Block 128 »** : 3 couches de convolution à 128 filtres de taille 3x3 avec activation 'relu', suivies d'une BatchNomalization et d'un MaxPooling 2x2
- **« Conv Block 256 »** : 3 couches de convolution à 256 filtres de taille 3x3 avec activation 'relu', suivies d'une BatchNomalization et d'un AveragePooling 2x2
- **« Classification block » :** Un couche Flatten remettant la sortie des couches de convolution en 1D, suivie successivement d'une couche Dense à 1024 neurones et d'une à 512 neurones. (Les couches de sortie en blanc ne sont pas comprises).
- Trois couches de sortie Dense avec activation 'softmax', une pour chaque cible :
  - 168 neurones pour la classification de 'grapheme_root'
  - 11 neurones pour la classification de 'vowel_diacritic'
  - 7 neurones pour la classification de 'consonant_diacritic'

La méthode d'optimisation utilisée est Adam, avec un learning rate de 0.001 et la fonction de coût est la 'categorical crossentropy'.

2. Variantes

Deux autres variantes de ce modèle, constituées des mêmes blocs mais agencés différemment ont également été évaluées :

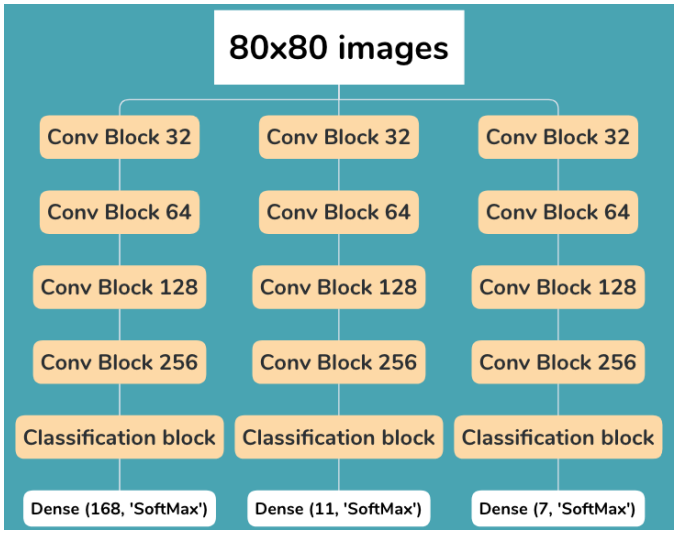


Figure 9: Variante 'Multi-Branch' du CNN de base

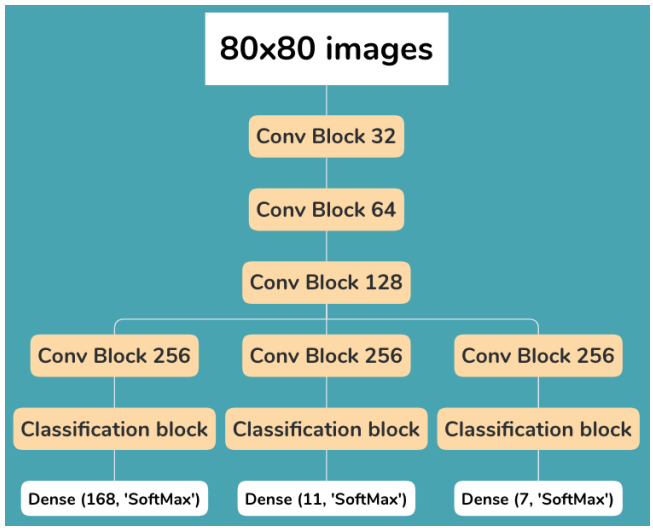


Figure 10: Variante 'Forked-Branchs' du CNN de base

3. Résultats

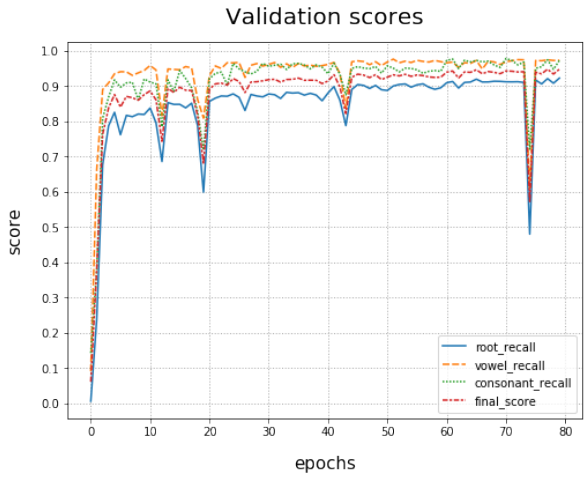


Figure 11: Performances du modèle de base

Figure 12: Performances variante 'Multi-branch'

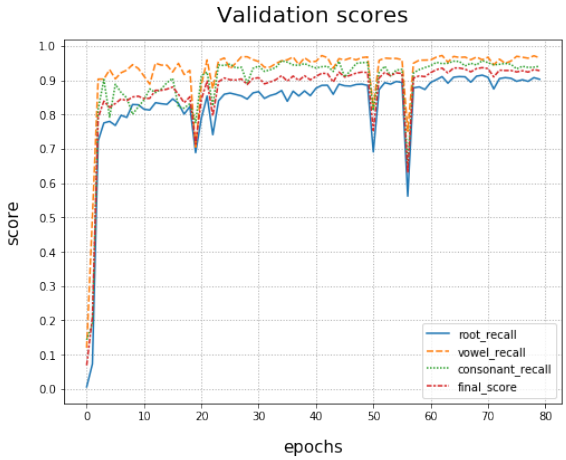


Figure 13: Performances variante 'forked-branchs'

Au final l'architecture de base donne des résultats similaires aux autres, mais est plus rapide à entrainer, c'est donc celle ci qui a été retenue.

B. Architectures complexes

En plus des modèles précédemment testés, deux autres architectures plus complexes et déjà éprouvées ont été testées : le RestNet50 et l'InceptionV3.

1. ResNet50V2 :

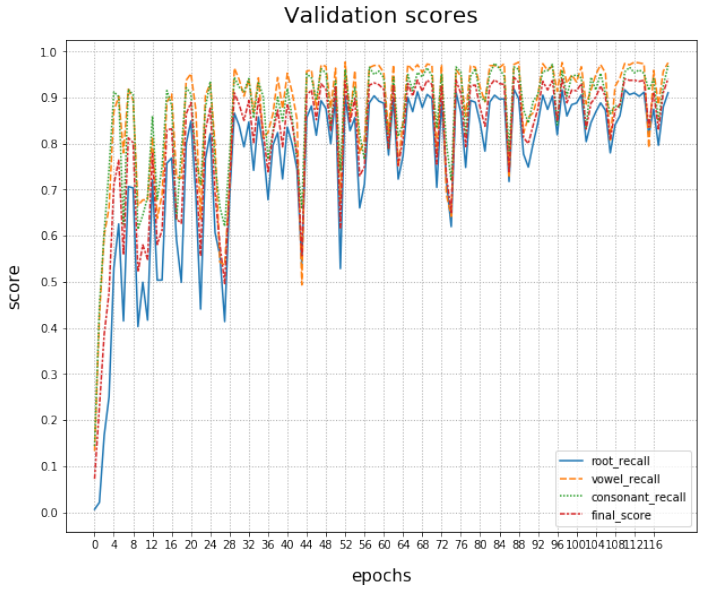


Figure 14: Performances du Resnet50

Le ResNet50 ne donne pas de meilleures performances que le modèle de base et introduit beaucoup de fluctuations lors de l'entrainement. Initialement l''hypothèse était que ce phénomène pouvait provenir de la méthode d'optimisation utilisée (Adam) et une méthode alternative (RMSProp) a été testée sans succès. L'origine de ses fluctuations pourraient plutôt provenir des couches de « Batch normalization », comme on le verra dans la partie « optimisation du modèle de base », mais cela n'a pas été vérifié pour ce modèle.

2. InceptionV3 :

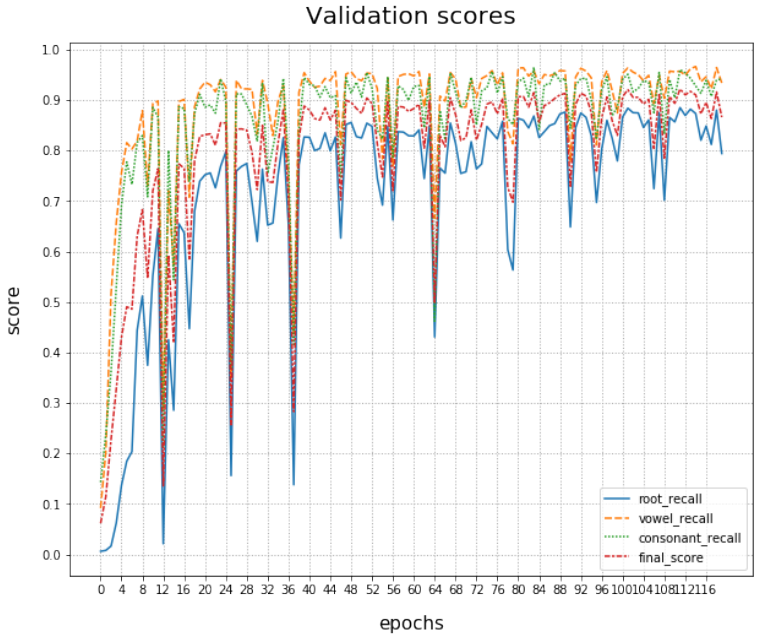


Figure 15: Performances de l'InceptionV3

Même constat pour l'InceptionV3 que pour le Resnet50. Au final il semblerait que l'architecture du modèle ait peu d'influence sur les performances et que passé un certain point la complexité n'apporte rien si ce n'est du temps de calcul supplémentaire. On retiendra donc la structure basique du modèle.

C. Optimisation de la structure de base

Les optimisations suivantes ont été apportées soit sur l'architecture de base, soit sur le processus d'entrainement, pour arriver sur le version finale :

- Optimisation de la boucle d'entrainement
- Ajout d'un couche de convolution au sein de chaque bloc de convolution
- Ajout d'un moment de 0.9 dans les couches de « Batch Normalization »
- Simplification du bloc de classification (division par 2 du nombre de neurones)

1. Optimisation du processus d'entrainement

Initialement, pour des raisons de limitations en mémoire et de format de données (parquet), le modèle était entrainé sur plusieurs époques pour un des sets de données, puis sur le suivant et ainsi de suite. Cela générait des « piques » dans la fonction de perte à chaque changement de set. Pour remédier à cela le format des données a été changé pour un format plus rapide (feather) et le modèle entrainé pour une époque sur chaque set successivement. Cette opération a été répétée plusieurs fois pour être l'équivalent d'un entrainement sur plusieurs époques. Cela a permis de retirer ces « piques » dans la fonction de pertes.

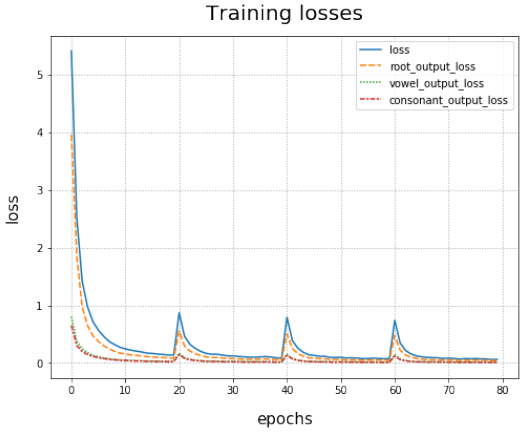


Figure 16: Processus d'entrainement de base

Figure 17: Processus d'entrainement optimisé

2. Optimisation de la structure

L'optimisation de la structure a permis d'améliorer légèrement les performances (passage de 0.9378 à 0.9424 sur le set de test public) mais aussi une plus grande stabilité du modèle, les chutes brutales de performances d'une époque à l'autre ayant disparue. Cette plus grande stabilité est essentiellement due à l'introduction d'un moment dans les couches de Batch Normalization et donc en considérant pour la normalisation une moyenne glissante des Batch plutôt que les statistiques du Batch seules.

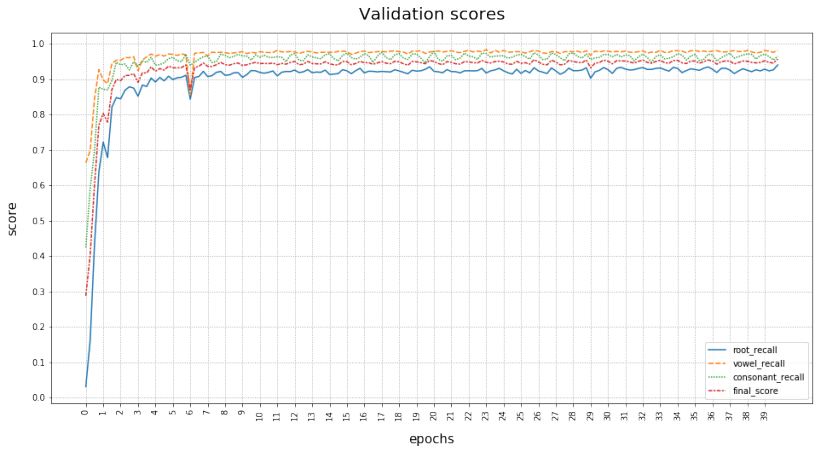


Figure 18: Scores sur le set de validation du modèle optimisé

D. Analyse des erreurs

N'ayant pas accès direct aux données de test, l'analyse des erreurs commises par le modèle a été réalisée sur le set d'entrainement, qui contient tout de même en partie des données que le modèle a vu uniquement pour la validation.

1. Répartition des erreurs par type

Les erreurs peuvent être de 4 types :

- Erreur sur 'grapheme_root' uniquement (59%)
- Erreur sur 'vowel_diacritic' uniquement (14%)
- Erreur sur 'grapheme_root' uniquement (13%)
- Erreurs multiples, sur au moins deux des cibles (14%)

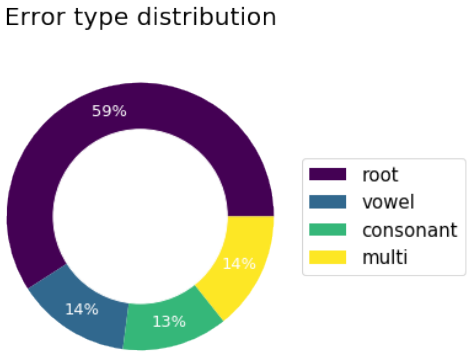


Figure 19: Répartition des types d'erreur

Au final presque 60% des erreurs sont dues à une prédiction erronée sur la racine du graphème. Améliorer la prédiction de cette composante semble être la clé de l'amélioration de la performance globale du modèle.

2. Répartition des erreurs par classe

Il est intéressant d'aller analyser les erreurs commises par classe au sein de chaque composante pour pouvoir détecter potentiellement des classes problématiques causant la majorité des erreurs. L'analyse a été effectuée en regardant le taux de présence de chaque classe parmi les erreurs faites par le modèle sur chacune des cibles (root, vowel et consonant) . On présentera visuellement dans ce rapport uniquement les résultats obtenus pour la cible 'grapheme_root', les conclusions pour les autres cibles étant relativement les mêmes.

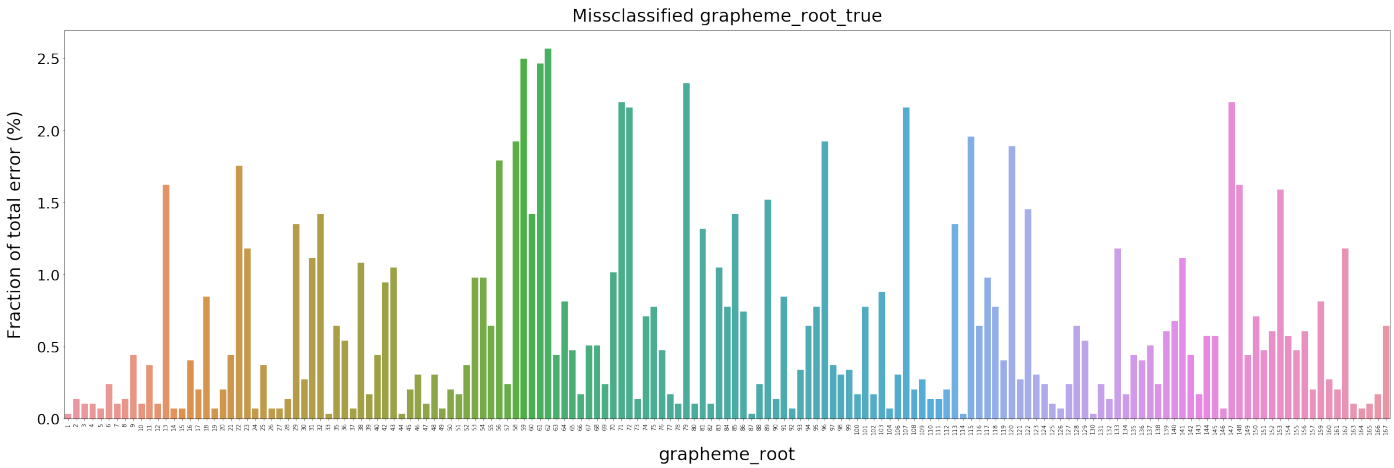


Figure 20: % d'erreur liées à chaque classes de 'grapheme root'

Pour la variable 'grapheme_root', on ne distingue pas de classe en particulier pour laquelle le taux d'erreur est sensiblement plus élevé et qui serait la source de la majorité des erreurs. Pour les variables 'vowel diacritic' et 'consonant_diacritic' certaines classes se distinguent des autres en termes de participation à l'erreur globale, mais il s'agit également des classes les plus représentées. Cela ne traduit donc pas forcément une difficulté particulière d'apprentissage pour le modèle. Au final cette analyse des erreurs commises n'aura pas permis d'identifier des points de défaillance particuliers du modèle.

IV. Conclusion

Le notebook contenant la partie entrainement et inférence du modèle retenu est disponible sur Kaggle.

Le modèle final a obtenu une performance de 0.9424 sur le set de test public, ce qui est proche des performances obtenues sur le set de validation (0.95). Le modèle se généralise donc plutôt bien, sachant qu'en plus le set de test contient potentiellement des combinaisons qui ne sont pas présentes dans le set d'entrainement.

Dans l'ensemble, le modèle arrive à détecter chacune des classes de chacune des composantes. Il n'existe pas de classe qui n'est jamais prédite, et ce malgré des distributions très déséquilibrées dans les données.

En plus de ce constat, cette étude a permis de mettre en avant les points suivants :

- L'architecture ne semble pas avoir un impact très important sur les performances passé un certain niveau de complexité. Des modèles avec une structure et des mécaniques simples permettent d'obtenir de bons résultats.
- Les erreurs proviennent majoritairement de la racine des graphèmes, c'est sur cette composante en priorité qu 'il faut chercher à améliorer le modèle.
- Les erreurs ne semblent pas provenir majoritairement de certaines classes, on peut donc penser que pour le moment il s'agit surtout d'erreurs car l'écriture est particulière et rend l'identification difficile.

Afin d'améliorer le modèle, on pourrait se pencher sur les points suivants :

- Essayer de travailler avec des images d'une résolution supérieure et limiter les déformations lors de l'étape de redimensionnement
- Mette en place de la data augmentation (CutMix, AugMix, rotations ...) couplée à un entrainement plus long. Cela pourrait permettre de palier à la déformation de certains caractères liée à la spécificité d'une écriture.
- Creuser l'analyse de l'erreur en visualisant les images mal classées pour confirmer / infirmer l'hypothèse de composantes difficiles à classer car écrites de manière particulière qui rend leur distinction difficile.