Projet 8

Participez à une compétition Kaggle !

Bengali.AI Handwritten Grapheme Classification

Contents

Introduction

Kaggle

Kaggle est une plateforme web organisant les compétitions en Data Science, créé en 2010 par Anthony Goldbloom. Les compétitions consistent à résoudre un problème sur les données réelles, souvent fourni par les entreprises ou par les organismes de recherche. Chaque compétition a son critère d'évaluation, souvent la précision des prédictions. Les meilleurs résultats peuvent être rémunérés financièrement, parfois aussi par une proposition d'embauche.

Kaggle rempli également une fonction éducative en proposant aussi des nombreux tutoriels. Les participants peuvent partager leurs notebooks et échanger sur le forum de discussion.

Nous pouvons également utiliser une machine virtuelle équipée par GPU dans la limite de 30h/semaine.

Le site met à disposition des données de nombreux domaines, ce qui est très intéressant notamment pour pouvoir travailler sur les datasets réels et intéressants, par exemple pour bâtir un portfolio.

Compétition choisie : Classification de graphèmes de bengali écrit

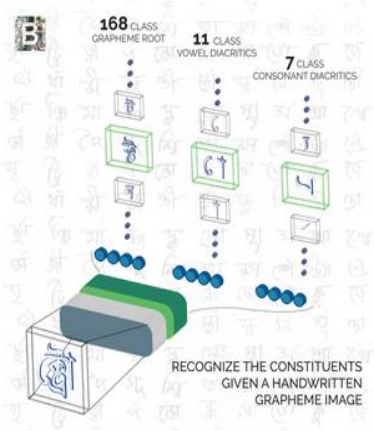
Nous avons choisi la compétition Bengali.ai qui consiste à classifier les graphèmes de bengali écrit.

Bengali est 5ème langue le plus parlé dans le monde, avec une population de centaines de millions de locuteurs. Il s'agit de la langue officielle de Bangladesh et de Bengale Occidentale (Inde).

Bengali est composé de 49 lettres (11 voyelles et 38 consonnes) et 18 accents (11 potentiellement attribuables aux voyelles et 7 aux consonnes).

Les graphèmes sont formés par des syllabes. Le nombre de variations potentielles est donc d'ordre assez important (~13 000 graphèmes différents).

Figure 1 - Schéma de classification de graphèmes écrits



Source : https://www.kaggle.com/c/bengaliai-cv19

Le but de la compétition

Le but de la compétition, classée dans le domaine de recherche, consiste à :

1. Améliorer les approches de reconnaissance de Bengali écrit, qui sont potentiellement extensibles aux autres langues issue de la famille de sanscrit
2. Démocratiser et accélérer la recherche dans les technologies linguistiques
3. Promouvoir l'éducation en Machine Learning

Les conditions

Le critère d'évaluation est recall pondéré, où le graphème racine représente deux fois plus de poids que les deux graphèmes diacritique.

Aussi, le temps d'exécution ne peut pas dépasser 2h sur GPU ou 9h sur CPU.

Le but du projet

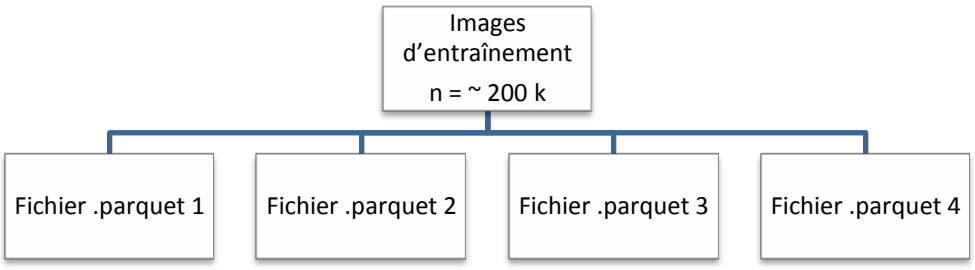
Le but de notre projet est notamment :

1. Construire ou customiser un algorithme de machine learning pour classifier les graphèmes de bengali
2. La particularité de la compétition consiste à une sortie de modèle multiple : le modèle devrait être capable d'identifier à la fois le graphème racine et des potentiels graphèmes de diacritique voyelle et consonne
3. Le but est de participer activement à la compétition Kaggle :
    a. Créer une équipe avec les autres étudiants Opeclassrooms.
    b. S'inspirer des kernels (notebooks) des autres utilisateurs
    c. Publier notre propre kernel et partager notre travail avec la communauté
    d. Soumettre les résultats de notre modèle final

Analyse exploratoire de données

Les données d'entraînement contiennent environ 200k d'images, stockées dans 4 fichiers séparés. Nous avons également un fichier .csv qui contient les labels de classes, ainsi que le graphème imprimé (Figure 2).

Figure 2 - Structure de données d'entraînement

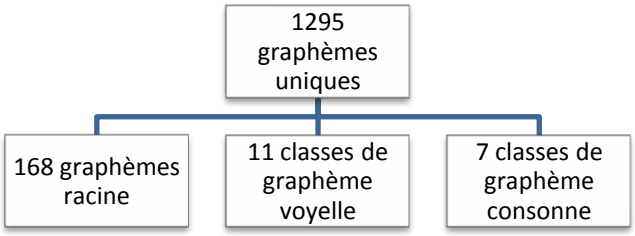


	image_id	grapheme_root	vowel_diacritic	consonant_diacritic	grapheme
0	Train_0	15	9	5	ক্ট্রো
1	Train_1	159	0	0	হ
2	Train_2	22	3	5	খ্রী

Nous avons 12 images de données test, donc 36 prédictions à faire et à stocker dans un fichier .csv.

Les images sont en noir et blanc de taille 137 x 236 pixels. Le jeu de données contient 1 295 graphèmes uniques, formés par 168 graphèmes racine, 11 classes de graphème voyelle et 7 classes de graphème consonne (Figure 3).

Figure 3 - Classification de graphèmes



La Figure 4 représente les fréquences de graphème racine. Nous pouvons constater que les classes sont assez déséquilibrées – le graphème le plus courant est présent 5 736 fois, tandis que le graphème le moins courant est présent seulement 149 fois.

Les graphèmes racine les plus courants (en version imprimée) sont visualisés sur la Figure 5. La Figure 6 représente les graphèmes racine les moins courants.

Figure 4 - Fréquence de graphèmes racine

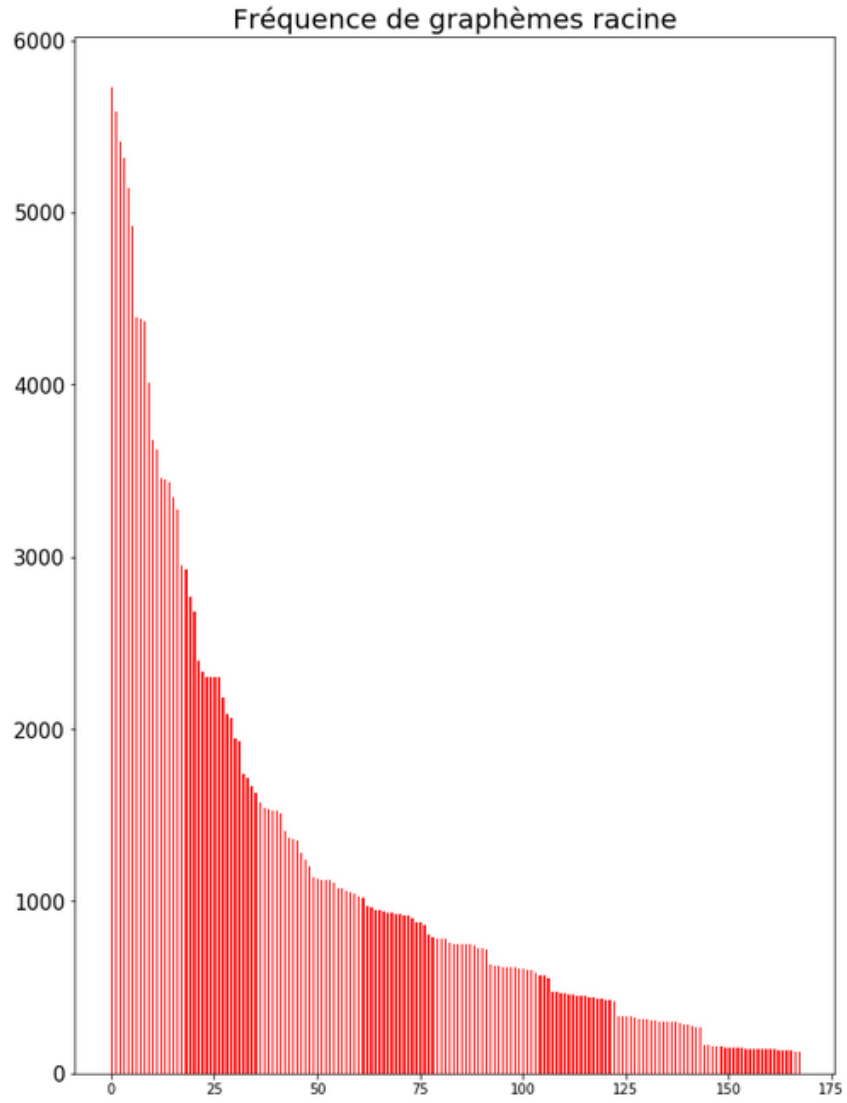


Figure 5 - Graphèmes racine les plus courants (version imprimée)

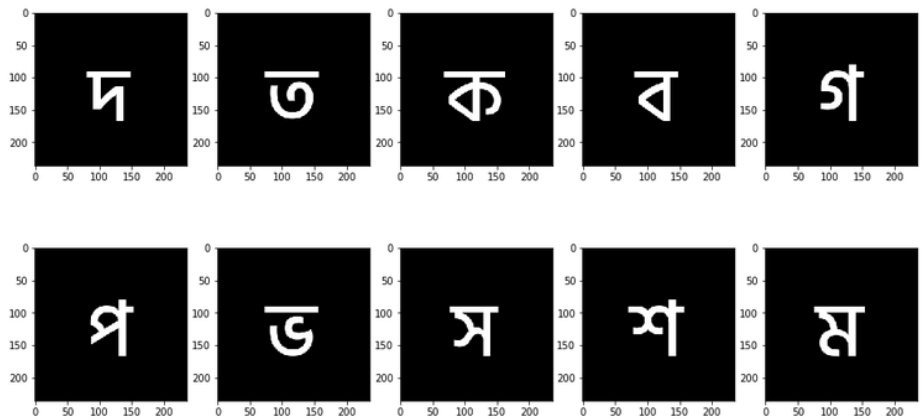


Figure 6 - Graphèmes racine les moins courants (version imprimée)

En ce qui concerne la fréquence des classes de signes de diacritique voyelle, représentée sur la Figure 7, nous constatons que la plupart de graphèmes (>40k) est sans diacritique. Les classes sont également déséquilibrées.

Les classes les plus courantes sont représentées sur la Figure 8.

Figure 7 - Fréquence de classes de diacritique voyelle

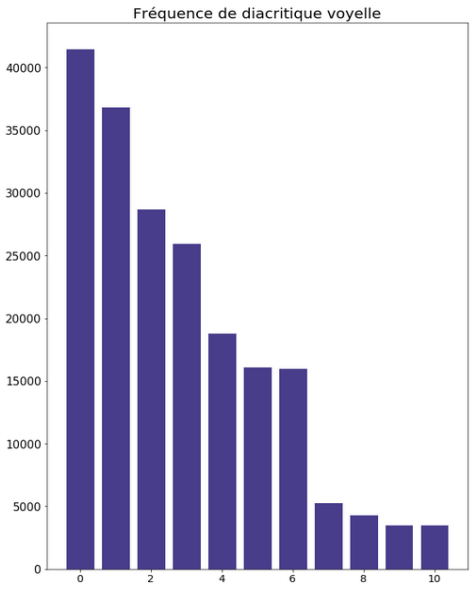
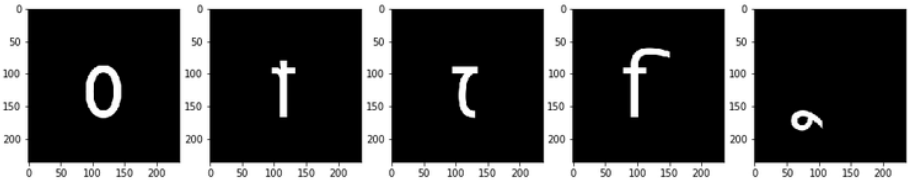


Figure 8 - Signes de diacritique voyelle les plus fréquents



Comme nous pouvons voir sur la Figure 9, la plupart de graphèmes (> 120k) ne contient aucun signe de diacritique consonne.

Les signes les plus courants sont représentés sur la Figure 10.

Figure 9 - Fréquences de classes de diacritique consonne

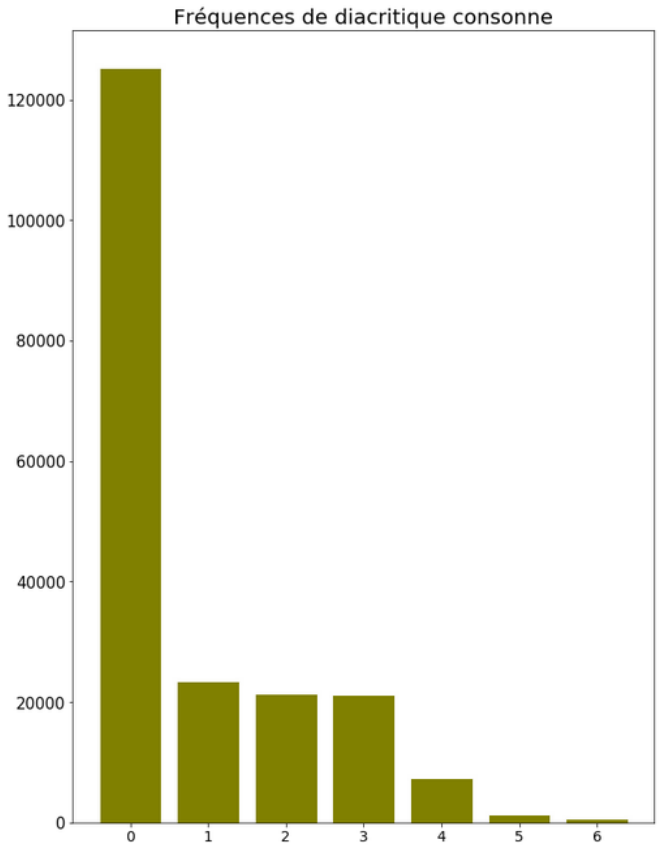
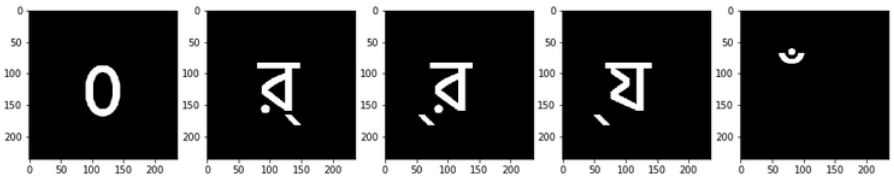
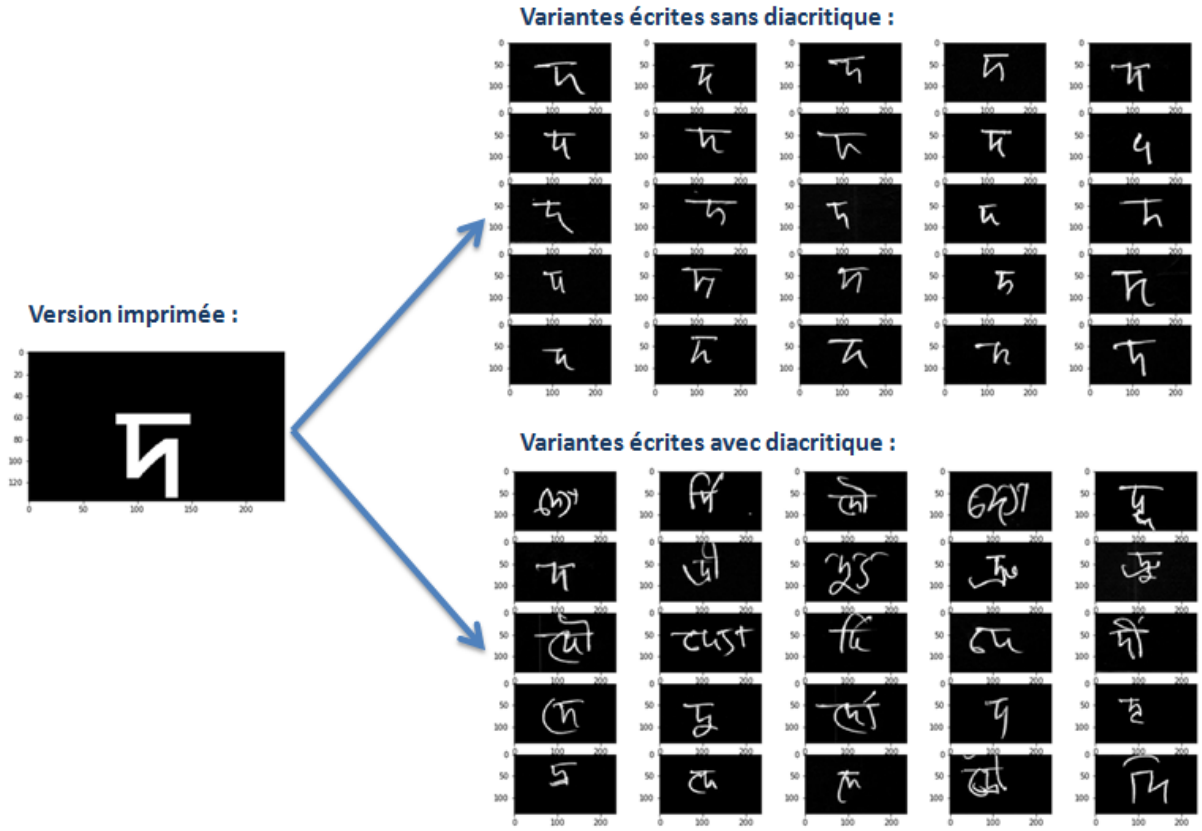


Figure 10 - Signes de diacritique consonne les plus fréquents



La Figure 11 contient les différentes formes de graphème racine le plus courant. La version imprimée est à gauche, tandis que les différentes versions écrites sont à droit : sans et avec signes de diacritique.

Figure 11 - Différents versions de graphème racine le plus courant



Feature engineering

Nous avons envisagés deux possibilités de gestion de données d'entraînement :

1. **Lecture fichier par fichier**

Le principe consiste en création d'une boucle d'entraînement, exécutée pour chaque fichier à la suite. Cette méthode nous permet de faire le traitement des images (par exemple augmentation etc.) à l'intérieur de la boucle d'entraînement. Elle est donc moins exigeante au niveau de mémoire.

Le désavantage de la méthode est le risque de déséquilibre de représentation de classes dans les batchs si les fichiers ne contiennent pas les mêmes graphèmes.

2. **Lecture de 4 fichiers à la fois**

Ici, nous ne risquons pas d'avoir des batchs déséquilibrés, car nous pouvons assurer le tirage d'images de façon aléatoire. Contrairement à la première méthode, nous sommes obligés de lire tous les quatre fichiers dans la mémoire à la fois. Cette méthode est donc beaucoup plus exigeante au niveau de mémoire et le stockage d'images traités.

Etant donné que Kaggle met à disposition un environnement d'exécution avec 13 GB de RAM (si nous optons pour l'utilisation le GPU), nous avons optés pour la première méthode.

Le workflow de feature engineering est le suivant :

1. Redimensionner les images
    - Taille originale : 137 x 236 pixels en NB
    - Taille de sortie : 64 x 64 pixels en NB
    - Normalization => Les valeurs en pixels (entre 0 et 255) divisées par 255
2. Data Augmentation :
    - Rotation de +/- 8°
    - Zoom de +/- 15 % de la taille d'image
    - Recentrage horizontale et verticale de +/- 15% de la taille d'image
3. Création d'une classe MultiOutputDataGenerator (classe enfant d'ImageDataGenerator) afin de pouvoir attribuer plusieurs cibles aux images

Implémentation de modèles multi-output

Familles de modèles

ResNet (Residual Network)

Kaiming He et al. [4] ont introduit le modèle qui a emporté la compétition ILSVCR en 2015. Le modèle développe l'idée d'utiliser un grand nombre de couches avec un peu de paramètres. En même temps, le modèle introduit un nouveau module appelé « unité résiduelle » (Figure 12). Le principe consiste en addition de l'entrée dans une couche à la sortie d'une couche située plus haut. Une « skip connexion » est utilisée dans les phases ou la taille d'entrée est différente de celle de sortie (Figure 13).

Nous parlons de la famille d'algorithmes ResNet, car il existe plusieurs variantes : ResNet18, ResNet34, ResNet50, ResNet152 qui différent en nombre de couches.

Figure 12 - Architecture de ResNet

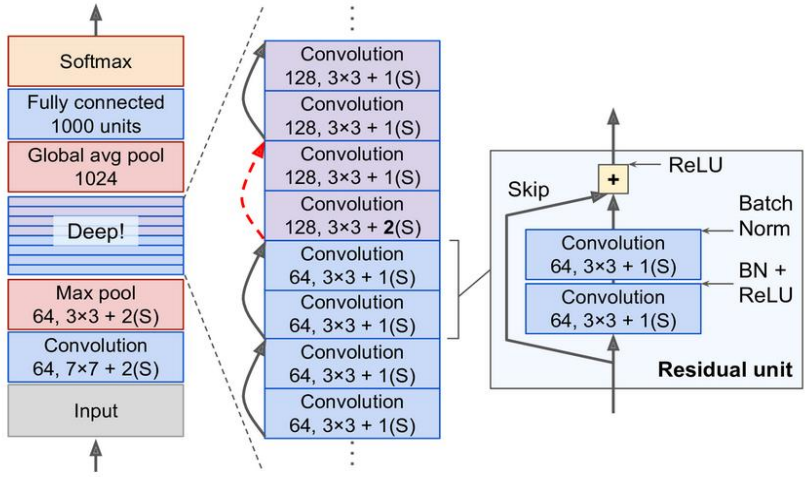
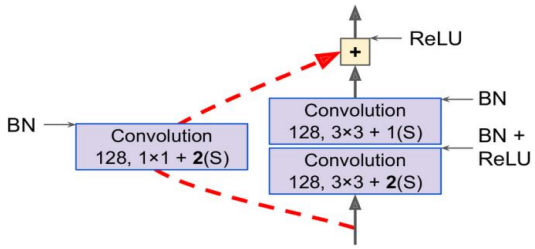


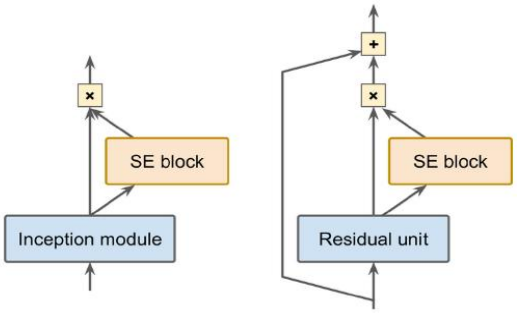
Figure 13 – Skip connexion



Source : Aurélien Géron : Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow

SENet (Squeeze and Excitation Networks)

Figure 14 - Module SE-Inception et SE-ResNet



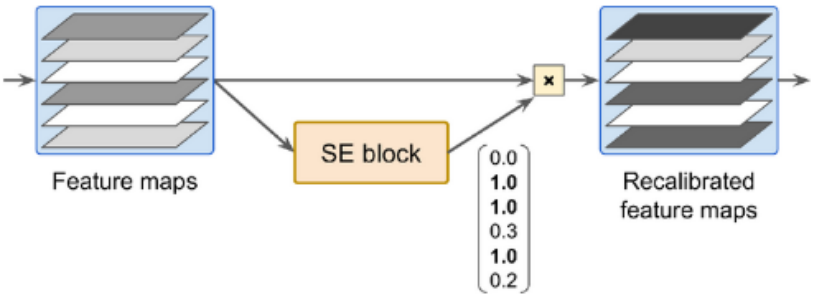
Source: Aurélien Géron: Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow

Les modèles SENet ont été introduits par Jie Hu et col. [6] et gagné la compétition ILSVRC en 2017. Il ne s'agit pas vraiment de nouveaux modèles mais d'une amélioration de modèles type Inception ou ResNet.

En effet, SENet ajoutent un nouveau module, « squeeze and excitation » après chaque module Inception (Inception) ou Unité résiduelle (ResNet).

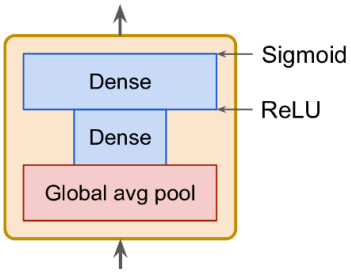
Le module est composé par une couche Global average pooling et deux couches fully connected (Figure 7). Le but de ce module est de re-calibrer les poids de feature maps. Le modèle apprend des liens entre des éléments dans les images. Si nous avons une forte activation des éléments qui représentent par exemple le nez + yeux, mais pas de bouche, le modèle va booster la feature map correspondante à la bouche, comme illustré dans la Figure 16.

Figure 15 - Le module SE re-celibre feature maps à la sortie



Source: Aurélien Géron : Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow

Figure 16 - Architecture de module SE



Source: Aurélien Géron : Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow

ResNet18

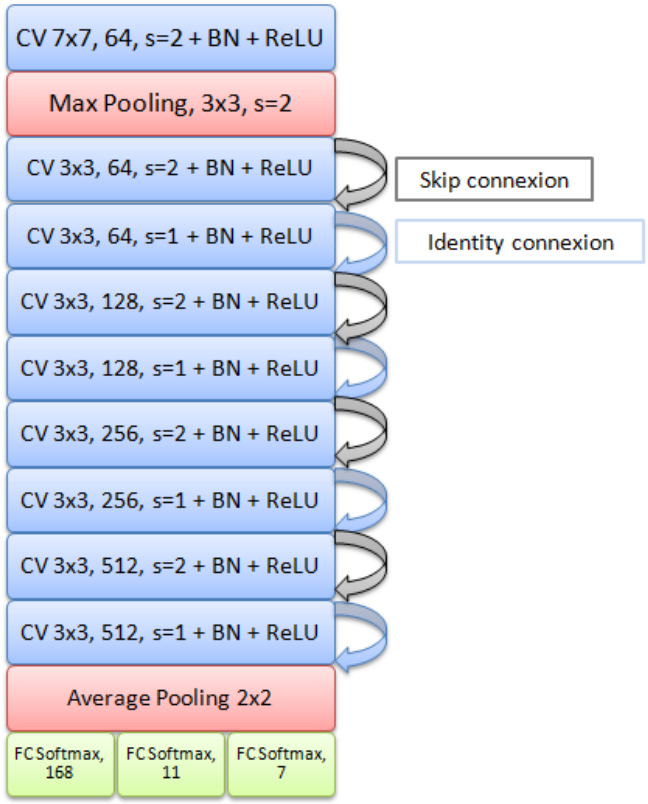
Nous avons choisi d'utiliser le modèle ResNet18, car les modèles de type ResNet sont assez performants et relativement simple à implémenter. En effet, sur Kaggle, il existe plusieurs kernels partagés pour créer des modèles de type ResNet. Nous avons utilisés en particulier le kernel de Kashal Shah : Bengali Graphemes : Multi Output ResNet-50.

Le désavantage de ce modèle est sa profondeur et un grand nombre de paramètres. L'auteur avoue que l'exécution de modèle est plus longue que la durée autorisée. C'est pour cela que nous avons choisi de réduire le nombre de couches de modèle et passer l'architecture à celle de ResNet18. Le nombre de paramètres d'entraînement était ainsi réduit de 24 millions à 5 millions. Le temps d'exécution de ResNet18 est autour de 40 min pour 25 epochs à l'aide de GPU Google colab.

La Figure 17 représente le schéma de modèle customisé pour notre problématique. Le modèle était construit à l'aide de 3 fonctions :

- **Indentity block** qui corresponde à l'unité résiduelle
- **Convolutional block** qui corresponde à skip connexion
- **ResNet18** pour réunir les différentes couches

Figure 17 - Schéma d'architecture de modèle ResNet18



Description de modèle :

1. Première couche convolutionnelle 7x7 avec 64 filtres.
    - Paramètre s=2 signifie que la fenêtre de filtre « se déplace » avec un pas de 2 pixels
    - Batch Normalisation : Normalise output de la couche précédente + ajoute deux paramètres aléatoires. Le premier est additionné et le deuxième multiplie la valeur normalisée.
    - ReLU : Ramène la valeur à 0 si négative.
2. MaxPooling 3x3 => Réduit les dimensions de la sortie de la couche précédente en renvoyant le max d'une fenêtre de taille 3x3 pixels
3. 8 couches convolutionnelles avec le pas de 1 ou 2 pixels, chacune terminée par BN + ReLU

- Si le pas s=2, unité résiduelle est remplacée par skip connexion pour (couche convolutionnelle 1x1, s=2) qui nous permet d'avoir la même taille d'input et d'output

4. Average pooling 2x2 => renvoie la moyenne de la fenêtre de 2x2 pixels
5. 3 couches fully connected activées par Softmax => une pour chaque output

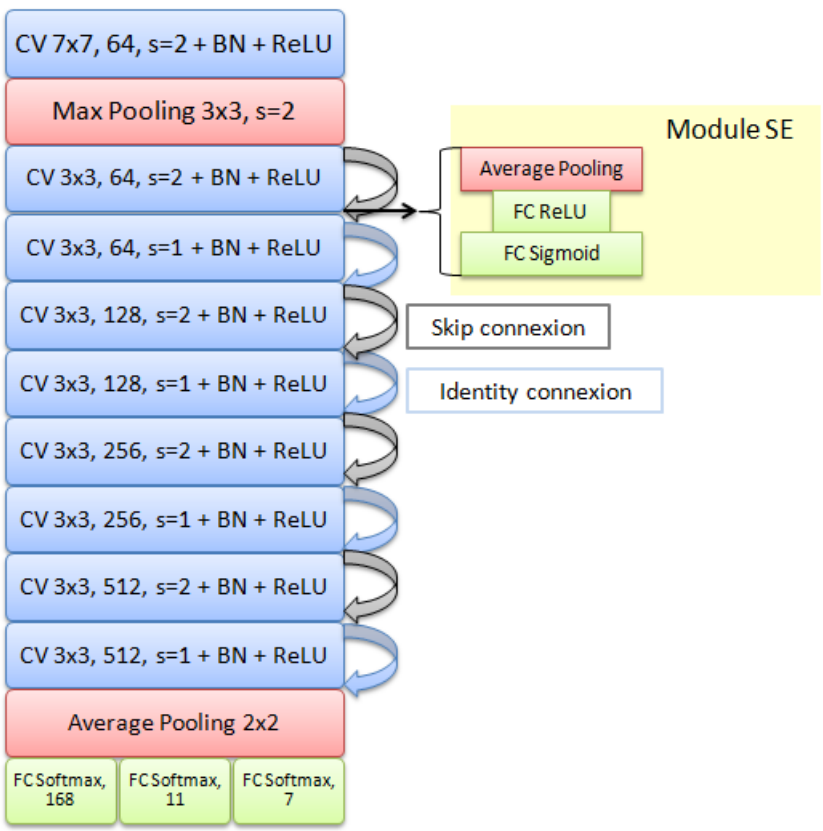
SE-ResNet18

Le modèle possède de la même architecture que le modèle ResNet18. Nous avons uniquement ajouté un module SE (Squeeze and Excitation) à la sortie de chaque unité résiduelle ou skip connexion. L'architecture modifiée est sur la Figure 18.

Le module SE est formé par :

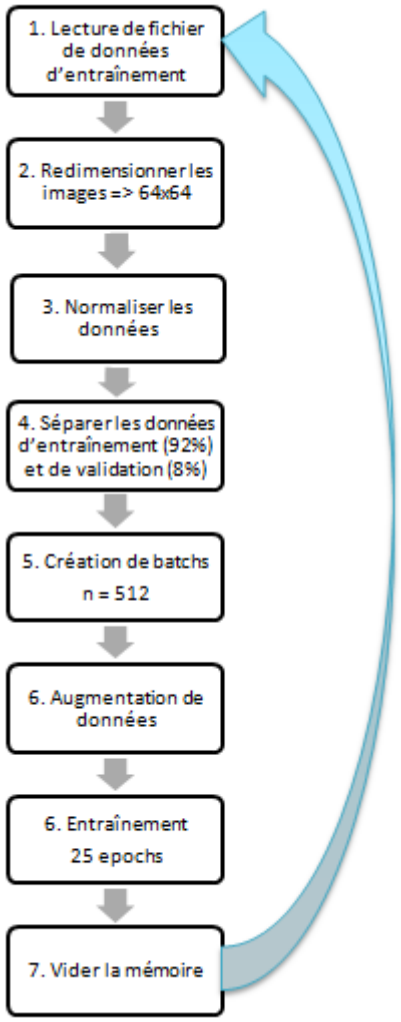
- Une couche Average Pooling
- Une couche fully connected activée par ReLU de taille 1/16 de couche précédente (Squeeze)
- Une couche fully connected de taille originale (Excitation) activée par une fonction sigmoïde

Figure 18 - Schéma d'architecture de modèle SE-ResNet18



Option d'exécution

Figure 19 - Workflow



La Figure 19 représente le schéma d'exécution de modèle qui est fait dans une boucle avec 4 répétitions (une pour chaque fichier de données).

1. Lecture de fichier de données d'entraînement : le premier fichier .parquet est chargé dans la mémoire
2. Redimensionner les images : toutes les images sont ramenées à la taille 64x64 pixels
3. Normaliser les données : les valeurs en pixels (entre 0 et 255) sont divisées par 255 pour obtenir les valeurs entre 0 et 1
4. Séparation de données d'entraînement et de validation. Nous créons une jeu de données de validation en préservant 8 % de données d'entraînement.
5. Nous créons des batchs de taille 512 images.
6. Nous procédons à l'augmentation de données décrit dans le chapitre précédent.
7. Nous entraînons les données pendant 25 epochs.
8. A la fin d'entraînement, nous vidons la mémoire et le cycle recommence

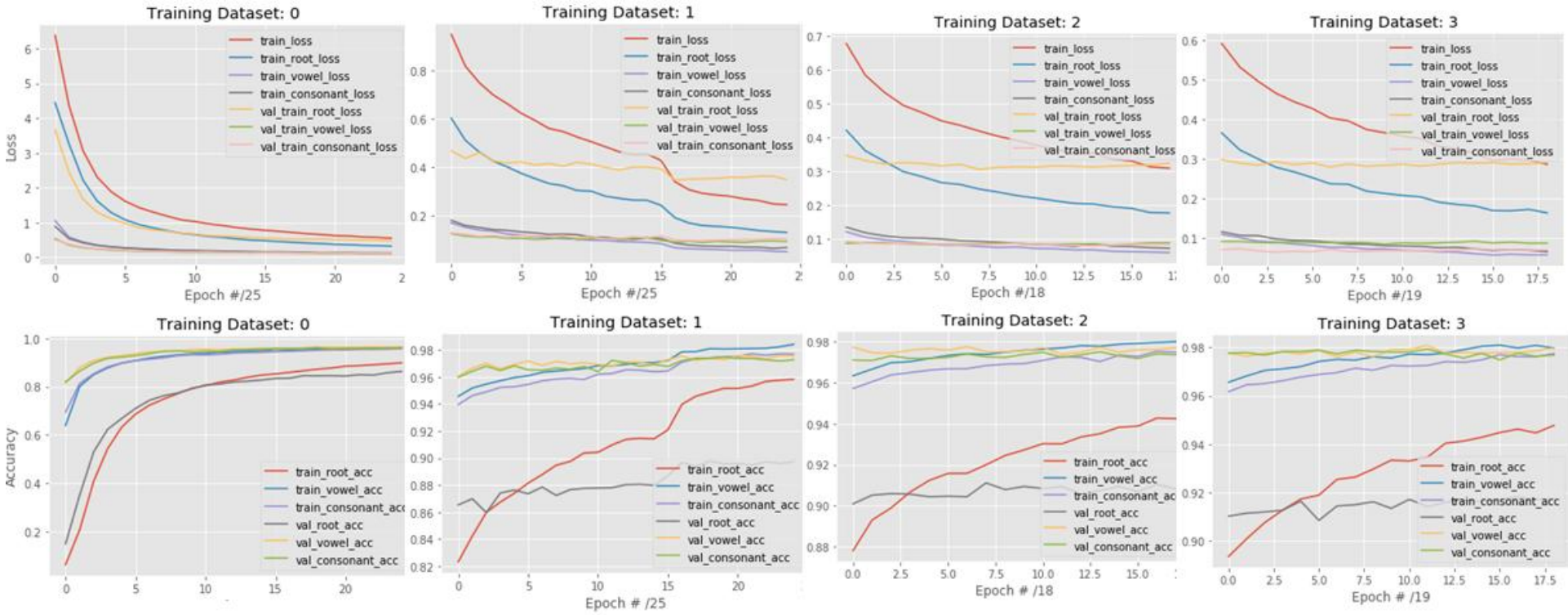
Pendant l'exécution de modèle, nous avons utilisés des outils Keras appelés callbacks :

1. Callback ReduceLROnPlateau : Si l'entraînement ne fait plus de progrès au bout de 3 epochs, réduire learning rate de 0.5. Le learning rate minimal est paramétré à 10e-6.
2. Callback ModelCheckpoint va enregistrer le meilleur modèle selon le minimum de fonction de perte globale calculée à partir de données de validation
3. Callback Early stopping : Si la valeur de fonction de perte arrête de diminuer et suit cette tendance pendant 10 epochs, l'entraînement va s'arrêter automatiquement

Résultats

ResNet18

Figure 20 - Courbes d'apprentissage, ResNet18



La Figure 20 représente les courbes d'apprentissage pour chaque fichier de données. Nous pouvons observer que l'ensemble de fonctions de perte correspondant à aux données d'entraînement et de validation pour chaque output ont globalement une tendance à descendre au fur et à mesure l'entraînement progresse, indépendamment au changement de dataset. En effet, la valeur de la fonction de perte d'entraînement globale commence à 6, puis diminue pour atteindre une valeur inférieure à 1. Il faut remarquer le changement d'échelle dans le graphique correspondant au dataset 1. Ici nous commençons à 0.9 et la fonction va encore descendre.

Lors de l'entraînement sur le dataset 0 et dataset 1, l'algorithme tourne pendant les 25 epochs, mais pendant l'entraînement sur le dataset 2 nous arrêtons à 18ème epoch. C'set à dire que la fonction de perte n'a pas fait de progrès depuis 8ème epoch, le callback programmé a donc arrêté l'entraînement. Une situation similaire survient pendant l'entraînement de dataset 3, où nous arrêtons l'entraînement à 19ème epoch.

En observant la fonction de perte de données d'entraînement (bleu) et de validation (jaune) relatives à l'output de graphème racine, nous constatons que le modèle a une tendance à surapprendre, car la courbe bleu a toujours tendance de diminuer, tandis que la courbe jaune a plutôt une tendance stagnant, voire ascendante.

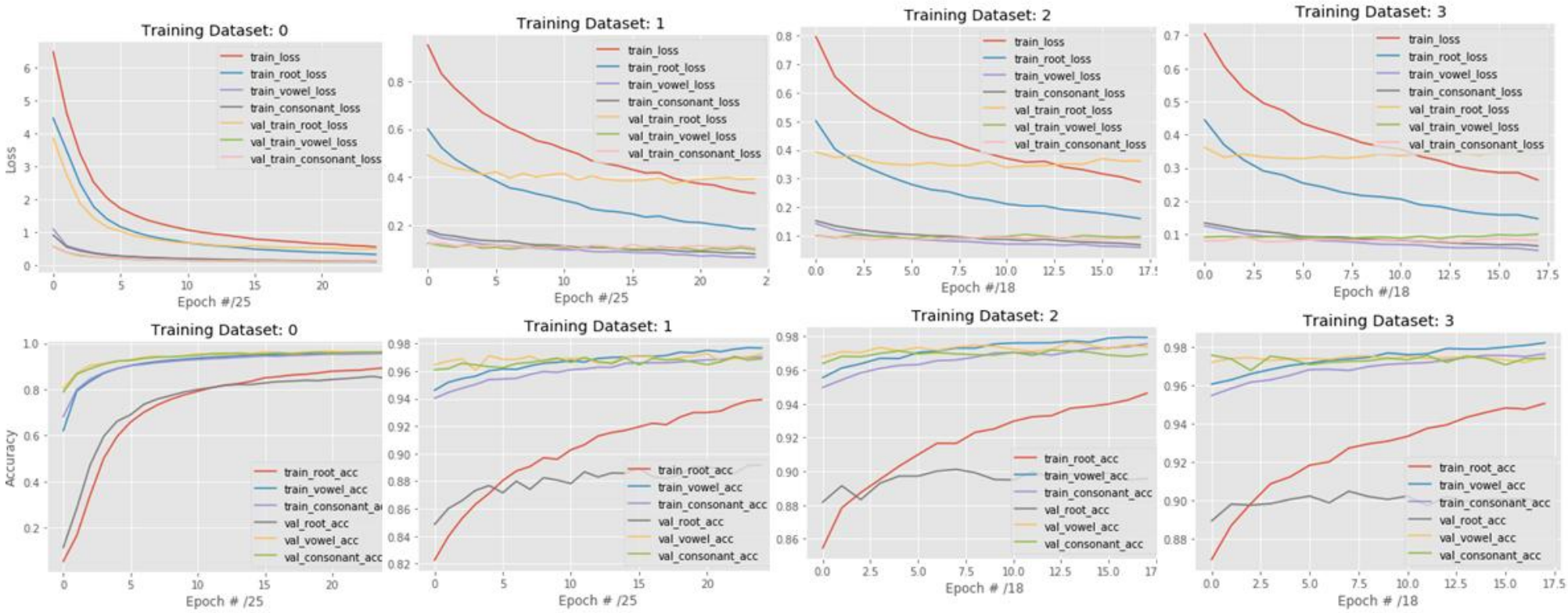
Nous avons atteint la meilleure précision sur les données de validation de 85.66% pour le graphème racine, 96.89 % pour le signe de diacritique voyelle et 96.47% pour le signe de diacritique consonne.

La moyenne pondérée de précisions est 91.47%, où le poids de graphème racine est égal à 2.

Lors de la soumission de ce modèle, nous avons atteint un recall pondéré sur les données test de 92.47 %.

SE-ResNet18

Figure 21 - Courbes d'apprentissage, SE-ResNet18



La Figure 21 représente les courbes d'apprentissage de modèle SE-ResNet18. Nous pouvons observer les mêmes tendances comme pour le modèle ResNet18.

Nous avons atteint la meilleure précision sur les données de validation de 85.88% pour le graphème racine, 96.91 % pour le signe de diacritique voyelle et 96.59% pour le signe de diacritique consonne.

La moyenne pondérée de précisions est 91.30%, où le poids de graphème racine est égal à 2.

Lors de la soumission de ce modèle, nous avons atteint un recall pondéré sur les données test de 80.81 %

Conclusion

Nous avons atteint un résultat relativement satisfaisant dès notre première soumission. Avec un recall de 92.47 %, nous nous plaçons à 671ème place sur 852. Le meilleur résultat est 99.23 %.

Nous avons également créé un kernel public, accessible à l'addresse suivante : https://www.kaggle.com/lenkast/bengali-graphemes-multioutput-resnet18-keras/notebook. Depuis la publication il y a deux jours, nous avons déjà 3 évaluations positives et 109 vues.

Nous serons ravis de continuer à améliorer nos résultats jusqu'à la fin de projet (prévu le 09/03/2020). Nous souhaitons notamment approfondir la recherche dans la littérature et dans les autres kernels et prendre des mesures pour éviter le surapprentissage. Nous pouvons aussi envisager d'effectuer des tests sur différentes options d'augmentation de données, taille d'images, etc., faire tuning de paramètres du modèle (learning rate, optimizers etc.) et tester d'autres modèles ou modifier la structure de modèle actuel.

Bibliographie

Articles :

Kaiming He and Xiangyu Zhang and Shaoqing Ren and Jian Sun. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV] (2015).

Jie Hu and Li Shen and Samuel Albanie and Gang Sun and Enhua Wu. Squeeze-and-Excitation Networks. arXiv:1709.01507 [cs.CV] (2017).

Blogs :

https://towardsdatascience.com/review-senet-squeeze-and-excitation-network-winner-of-ilsvrc-2017-image-classification-a887b98b2883

Livres :

Aurélien Géron : Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow

Liens vers les kernels utilisés dans notre travail :

https://www.kaggle.com/kaushal2896/bengali-graphemes-multi-output-resnet-50
https://www.kaggle.com/kaushal2896/bengali-graphemes-starter-eda-multi-output-cnn
https://www.kaggle.com/pestipeti/bengali-quick-eda