Openclassrooms projet 8 : Compétition Kaggle

RSNA-MICCAI Brain Tumor Radiogenomic Classification

Cédric Soares – Octobre 2021

Table des matières

Le contexte

La compétition

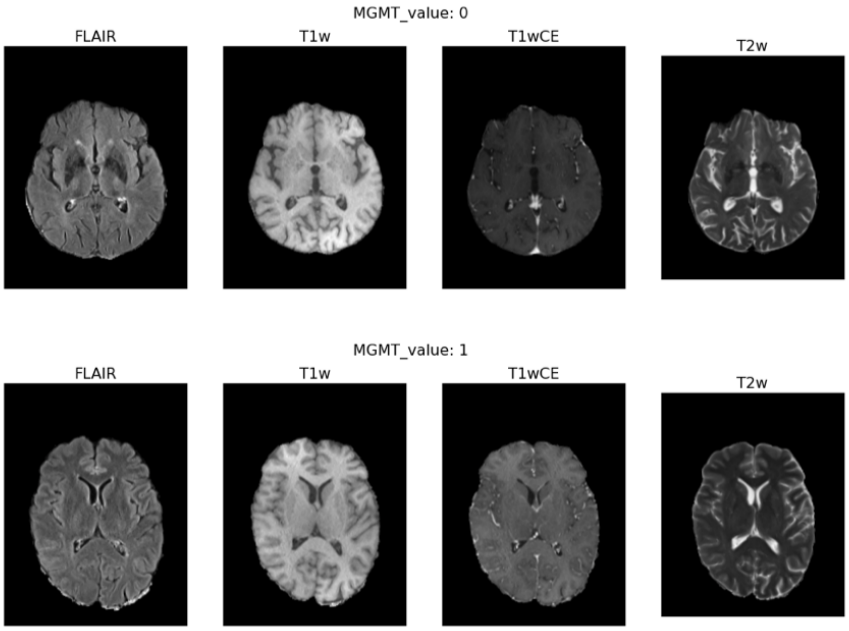
RSNA (Radiological Society of North America) et MICCAI Society (The Medical Image Computing and Computer Assisted Intervention Society) ont lancé un challenge Kaggle afin de détecter la présence d'un biomarqueur génétique favorisant l'efficacité des traitements des cancers du cerveau. Et effet, selon les organisateurs, la présence du MGMT (Méthyle guanine méthyle transférase) chez les patients atteints de glioblastome est un facteur de pronostique favorable et un puissant prédicteur de la réactivité à la chimiothérapie. Cette forme de cancer du cerveau est à la fois la plus courante et la plus grave chez l'adulte avec une survie médiane inférieure à un an.

Actuellement, la détection du MGNT nécessite la réalisation d'une biopsie et peut prendre plusieurs semaines. Selon les résultats et les types de traitements initialement mis en œuvre, une intervention chirurgicale ultérieure peut s'avérer nécessaire. Le développement d'une méthode efficace de détection par imagerie médicale pourrait potentiellement minimiser le nombre d'opérations.

Les participants au challenge ont été invités, du 13 juillet au 25 octobre, à proposer des modèles de machine learning permettant d'indiquer la présence ou non du MGMT à l'aide de clichés d'IRM. L'indicateur AUC (area under the ROC curve) a été retenu pour évaluer les modèles.

Les données

Les organisateurs ont mis à disposition des participants deux jeux de donnés, entrainement et test, composés de clichés d'IRM. Les clichés sont triés par patient. Pour chacun, plusieurs clichés de quatre différents types de scans (FLAIR, T1w, T1wCE, T2) sont disponibles. Le jeu d'entrainement contient les scans de 1010 patients.



Exemple de clichés du jeu d'entrainement – source : *kernel Brain Tumor – EDA with Animation and Modeling publié par Yaroslav Isaienov* – octobre 2021

Les contraintes

L'évaluation est réalisée à partir des notebooks soumis au concours. L'ensemble des opérations de ces derniers doivent pouvoir être réalisées en maximum 9 heures d'utilisation CPU ou GPU. L'utilisation de données libres et gratuites et de modèles pré-entrainés est autorisée.

La solution proposée

Le kernel utilisé

Mes travaux sont basés sur le kernel *EfficientNet transfer learning model full* publié par Bill Qi en septembre 2021. L'AUC a été évaluée à 0.62103 au classement provisoire (public) réalisé sur 22% du jeu de test et 0.5515 au classement final (private) réalisé sur 78% des données restantes.

La solution initiale consiste à réaliser un entrainement, par transfer learning, sur quatre modèles EfficientNet-B3 distincts par types de scan. Chacun calcule la probabilité d'appartenance à une classe 0 (pas de présence du MGMT) ou 1 (présence du MGMT). Le score final par patient est obtenu par l'algorithme suivant :

```
Calcul_de_prediction() :
Début
        all_test_preds : liste
        Pour chaque type de scan :
                train_df ← liens des clichés du type de scan du jeu d'entrainement
                test_df ← liens de clichés du type de scan du jeu de test
                train_g, test_g ←  generation_données (train_df, test_df)
                best_model  ←  entrainement_model(type_cliché, train_g, epochs=5)
                test_pred  ←  best_model.prediction(test_g)
                test_df['pred_y'] ← test_pred
                mean_pred ← Moyenne de test_pred
                test_pred_agg ← pour chaque patient_id :
                        test_pred_agg ← max(test_df['pred_y'])
                                        Si max(test_df['pred_y']) – mean_pred > mean_pred -
                                        min(test_df['pred_y'])
                                        Sinon min(test_df['pred_y'])
                                        Fin Si
                                Fin
                all_test_preds ← test_pred_agg
        Fin
Fin
```

Le modèle utilisé

La solution initiale implémente EfficientNet. La famille de modèles est issue de travaux publiés par les chercheurs de Google AI en 2019. Ceux-ci portaient sur l'augmentation de l'échelle de réseau de neurones convolutionnels. Jusque-là, l'augmentation de la taille des modèles pouvait être réalisée sur trois dimensions distinctes :

- Augmenter de la profondeur du réseau : permet d'apprendre des caractéristiques plus complexes mais rends la convergence du réseau plus difficile
- Augmenter la largeur des couches : permet d'apprendre un plus grand nombre de caractéristiques mais diminue la précision du modèle et la complexité des caractéristiques apprises.
- Augmenter la résolution des images en entrée : permet au modèle de mieux identifier les détails de l'image mais ne permet pas d'augmenter la précision du modèle.

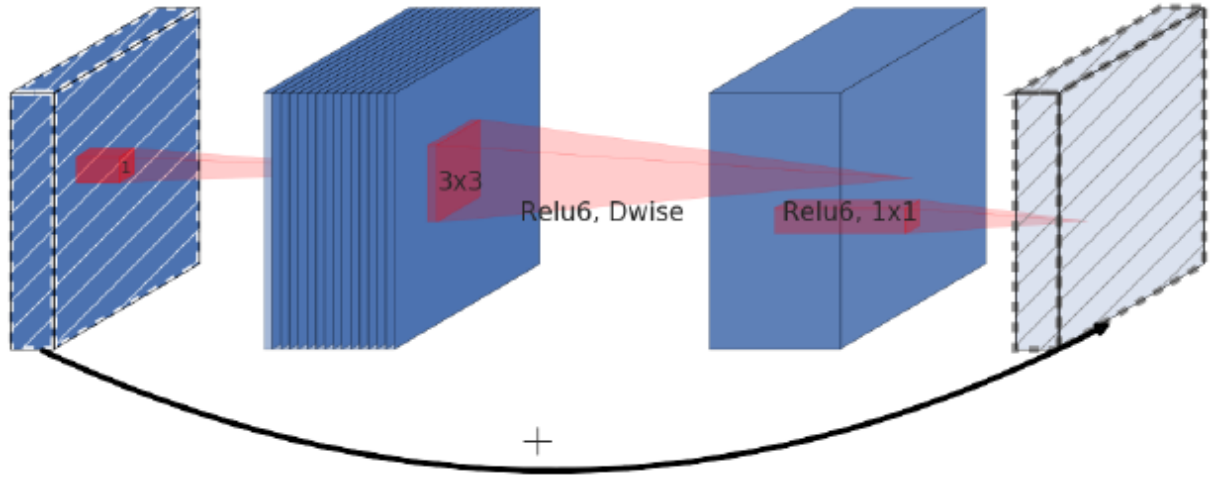
Les équipes de Google ont recherché une heuristique d'augmentation d'échelle, appelée *compoud scaling,* permettant de combiner les trois dimensions en vue de maximiser les performances des modèles tout en minimisant le nombre de paramètres et d'opérations par secondes qu'ils réalisent.

Pour se faire les chercheurs ont d'abord conçu une architecture minimale EfficientNet-B0. Cette dernière devait pouvoir être déployée sur des smartphones.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

Architecture EfficientNet-B0 – Source : Article de recherche *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks* publié sur ArXiv en septembre 2020 – Capture octobre 2021

L'architecture est basée du des blocs MBConv sur lesquels une optimisation squeeze-and-excitation est appliquée. Ces blocs sont similaires aux blocs résiduels inversés utilisés dans l'architecture de MobileNet v2. Ils implémentent une connexion de raccourcis entre la couche d'entrée et la couche de sortie du bloc. Les blocs sont composés d'une première couche de convolution 1 x 1 qui sert à augmenter la profondeur de la carte des caractéristiques. Cette couche est suivie d'une couche 3 x 3 Depth-wise puis couche Point-wise de 1 x 1.



Détail d'un bloc résiduel inversé - Source : Article *EfficientNet: Scaling of Convolutional Neural Networks done right* publié par Armughan Shahid sur towards data science en juin 2020 – Capture octobre 2021

La recherche de l'heuristique a été réalisée à l'aide d'une recherche de paramètres sur grille. Afin de répondre à la modélisation suivante :

$$\text{depth: } d = \alpha^\phi$$
$$\text{width: } w = \beta^\phi$$
$$\text{resolution: } r = \gamma^\phi$$
$$\text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$$
$$\alpha \geq 1, \beta \geq 1, \gamma \geq 1$$

Modélisation du compound scaling – Source : *Article de recherche EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks publié sur ArXiv en septembre 2020* – Capture octobre 2021

- α, β, γ sont des constantes trouvées par recherche de paramètres sur grille
- φ un hyper paramètre de coefficient d'augmentation des ressources disponibles pour l'augmentation de l'échelle du modèle

Grâce au compound scaling, les chercheurs ont décliné l'architecture EfficientNet-B0 en sept versions ayants des constantes de dimensions croissantes. Ils ont comparé les performances, le nombre de paramètres et le nombre d'opération par secondes avec d'autres modèles issus de l'état de l'art.

Model	Top-1 Acc.	Top-5 Acc.	#Params	Ratio-to-EfficientNet	#FLOPs	Ratio-to-EfficientNet
EfficientNet-B0	**77.1%**	**93.3%**	**5.3M**	**1x**	**0.39B**	**1x**
ResNet-50 (He et al., 2016)	76.0%	93.0%	26M	4.9x	4.1B	11x
DenseNet-169 (Huang et al., 2017)	76.2%	93.2%	14M	2.6x	3.5B	8.9x
EfficientNet-B1	**79.1%**	**94.4%**	**7.8M**	**1x**	**0.70B**	**1x**
ResNet-152 (He et al., 2016)	77.8%	93.8%	60M	7.6x	11B	16x
DenseNet-264 (Huang et al., 2017)	77.9%	93.9%	34M	4.3x	6.0B	8.6x
Inception-v3 (Szegedy et al., 2016)	78.8%	94.4%	24M	3.0x	5.7B	8.1x
Xception (Chollet, 2017)	79.0%	94.5%	23M	3.0x	8.4B	12x
EfficientNet-B2	**80.1%**	**94.9%**	**9.2M**	**1x**	**1.0B**	**1x**
Inception-v4 (Szegedy et al., 2017)	80.0%	95.0%	48M	5.2x	13B	13x
Inception-resnet-v2 (Szegedy et al., 2017)	80.1%	95.1%	56M	6.1x	13B	13x
EfficientNet-B3	**81.6%**	**95.7%**	**12M**	**1x**	**1.8B**	**1x**
ResNeXt-101 (Xie et al., 2017)	80.9%	95.6%	84M	7.0x	32B	18x
PolyNet (Zhang et al., 2017)	81.3%	95.8%	92M	7.7x	35B	19x
EfficientNet-B4	**82.9%**	**96.4%**	**19M**	**1x**	**4.2B**	**1x**
SENet (Hu et al., 2018)	82.7%	96.2%	146M	7.7x	42B	10x
NASNet-A (Zoph et al., 2018)	82.7%	96.2%	89M	4.7x	24B	5.7x
AmoebaNet-A (Real et al., 2019)	82.8%	96.1%	87M	4.6x	23B	5.5x
PNASNet (Liu et al., 2018)	82.9%	96.2%	86M	4.5x	23B	6.0x
EfficientNet-B5	**83.6%**	**96.7%**	**30M**	**1x**	**9.9B**	**1x**
AmoebaNet-C (Cubuk et al., 2019)	83.5%	96.5%	155M	5.2x	41B	4.1x
EfficientNet-B6	**84.0%**	**96.8%**	**43M**	**1x**	**19B**	**1x**
EfficientNet-B7	**84.3%**	**97.0%**	**66M**	**1x**	**37B**	**1x**
GPipe (Huang et al., 2018)	84.3%	97.0%	557M	8.4x	-	-

Comparatif des modèles de la famille EfficientNet à l'état de l'art – Source : *Article de recherche EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks publié sur ArXiv en septembre 2020* – Captubre ocotobre 2021

Dans son kernel, Bill Qi a implémenté la déclinaison EfficientNet-B3. Cette dernière a été entrainée sur des images de résolution 300 x 300. Les travaux semblent être une preuve de concept. Les modèles entraînés ne sont ni optimisés ni évalués.

Améliorations apportées au kernel

Les améliorations que j'ai apportées sont de trois ordres :

- Mise en conformité vis-à-vis du type de modèle
- Évaluation des modèles
- Optimisation des modèles

Mise en conformité vis-à-vis du type de modèle

Le kernel initial implémentait une résolution de 224 x 224 alors que le modèle EfficientNet-B3 est pré-entraîné sur des images de résolution 300 x 300. Mon implémentation utilise cette résolution de pré-entrainement.

Évaluation des modèles

Dans l'implémentation initiale aucune partition de validation n'était réalisée. J'en ai implémenté une réalisée de manière stratifiée sur les identifiants des patients et la classe des labels. Elle permet de garantir l'indépendance des données et de garantir un équilibre de classe équivalent entre les jeux d'entraînement et de validation.

Lors de l'entraînent des modèles, l'ajout d'un jeu de validation permet de mesurer la capacité du modèle à généraliser. Dans l'algorithme de prédiction j'ai ajouté un appel à la fonction d'évaluation d'implémenté dans l'API de Keras.

Optimisation des modèles

Dans l'implémentation initiale chaque modèle était entrainé sur 5 epochs sans contrôle de la minimisation de la fonction de coût. Il était donc impossible de vérifier si les modèle apprenaient bien et s'ils ne sur ajustaient pas sur les données d'entraînement.

En premier lieu j'ai allongé le nombre d'epochs à 20 afin de donner l'opportunité d'avoir un apprentissage plus long. J'ai également ajouté un callback d'early stopping paramétré pour arrêter l'entraînement du modèle après 5 epochs sans minimisation de fonction coût sur les données de validation. J'ai également ajouté un callback de réduction du learning rate paramétré pour diviser le learning rate par deux après deux epochs sans minimisation de la fonction de coût.

L'implémentation initiale comprenait une batch normalization et un drop out de 10% avant la première couche complètement connectée. Afin d'améliorer la stabilité du modèle et réduire la probabilité de sur apprentissage j'ai ajouté une batch normalization et un drop out avant la couche de prédiction. Pour les deux couches de drop out j'ai utilisé un paramètre d'extinction de 40% des neurones.

D’après les traces du notebooks les entrainements des modèles se sont déroulés sur 7 à 11 epochs. Au global les fonctions de coûts ont minimisé sur 2 à 4 epochs. In fine on peut en déduire que les entraînements de l’implémentation initiale n’étaient pas optimaux.

L’implémentation initiale utilisait un batch size de 512. J’ai pour ma part testé les valeurs 32, 64, 128. La capacité de calcul hebdomadaire mise à disposition par Kaggle ne m’a pas permis de tester les valeur 256 et 512 avant la date de clôture du concours.

Résultats

Sur les différentes tailles de batch size, voici les évaluations publiques reçues par mon implémentation optimisée sur 22% des données.

Batch size	AUC Publique
32	0,61945
64	0,57875
128	0,65909

J’ai soumis pour évaluation finale le modèle avec une Batch size de 128. Voici la comparaison des évaluations entre la solution initiale et ma version optimisée.

Implémentation	Batch size	AUC Publique (22% des données)	Classement provisoire	AUC Privée (78% des données)	Classement final
Initiale	512	0,62103	906	0,55150	166
Optimisée	128	0,65909	679	0,61732	3

À la lecture des résultats il est possible de déduire que la solution optimisée permet de meilleures performances et une meilleure généralisation des modèles.

Le kernel optimisée est disponible sur Kaggle et une version commentée en français est mise à disposition sur Github.