PROJET8 – PARTICIPEZ A UNE COMPETITION KAGGLE !

Francesco Loiudice

---

1. Objectif

L'objectif du projet est la participation à la compétition Kaggle 'Petals to the metal' avec la création d'un modèle performant de classification d'images de fleurs avec l'utilisation pour l'entrainement d'une TPU (Tensor Prodessing Unit).

2. TPU (Tensor Processing Unit)

Une TPU est un accélérateur HW spécialisé en deep learning tasks constitué de 8 cores qui contiennent 8 différentes replicas du modèle. La TPU est utilisée par Tensorflow sur la base d'une stratégie qui distribue l'entrainement sur les différentes cores. Chaque core TPU possède une VPU (Vector Processing Unit) ainsi qu'une partie HW dédiée à la multiplication de matrices capable de traiter des matrices 128x128 et donc très performante avec les réseaux de neurones.

Pour l'utilisation de la TPU sur Kaggle il faut que le database soit stocké dans GCS (Google Cloud Storage) Bucket et que les données soient sérialisées dans le format TF Records avec binary files qui contiennent binary string.

Lors de l'entrainement le modèle doit être instancié dans la stratégie d'utilisation de la TPU.

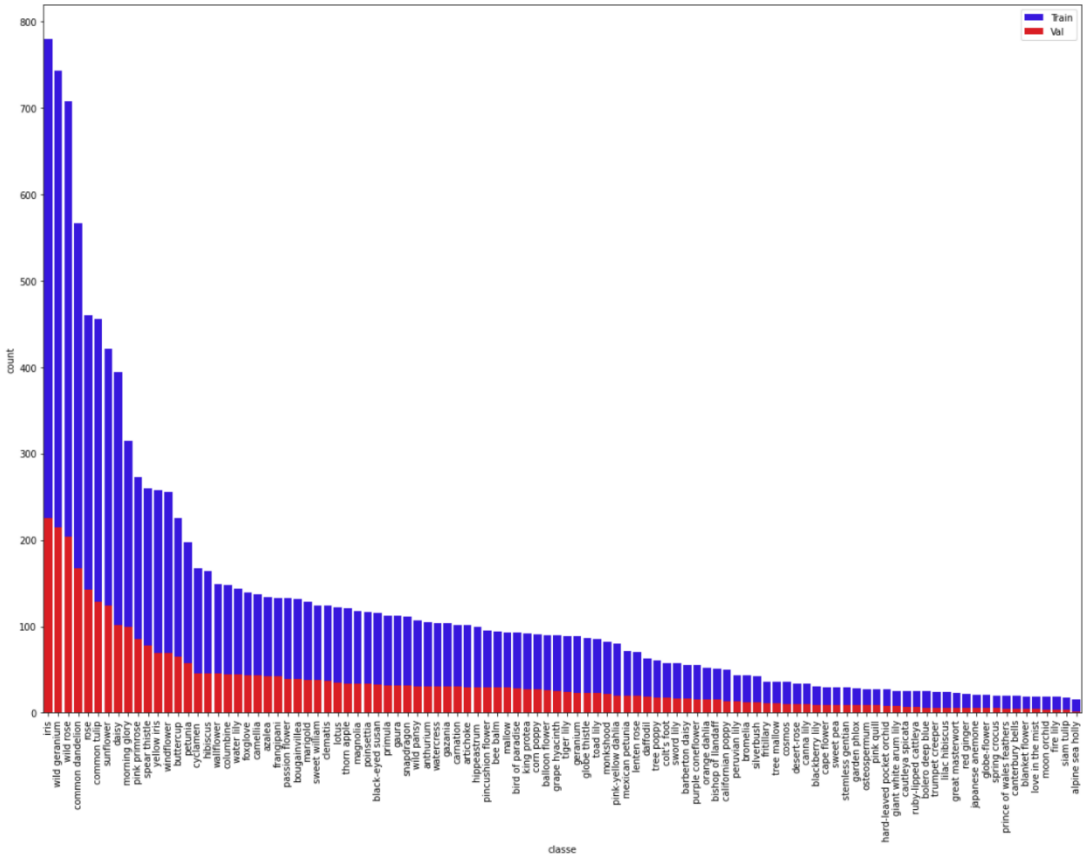
3. Analyse exploratoire du jeu de donnée

Le jeu de données 'tpu-getting-started' contient 192 files d'images de fleurs de 104 type différents (classes) en format TF Records et avec tailles 192x192, 224x224, 331x331, 512x512. Pour chaque taille les jeux de training, validation et test sont déjà définis pour la compétition avec un split 70/30 entre jeu de traning/validation et jeu de test.

Les dimensions du jeu de données sont : 12753 images pour le jeu de training, 3712 images pour le jeu de validation et 7382 images pour le jeu de test.

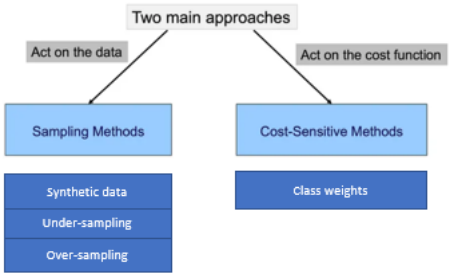
Chaque image est identifiée avec un id et une étiquette (classe) pour le jeu de training et validation et seulement un id pour le jeu de test. Le jeu de validation est obtenu avec une stratification sur les classes pour le split entre données de training et validation.

On a choisi les images de taille 512x512 et analysé la distribution des images par classe.

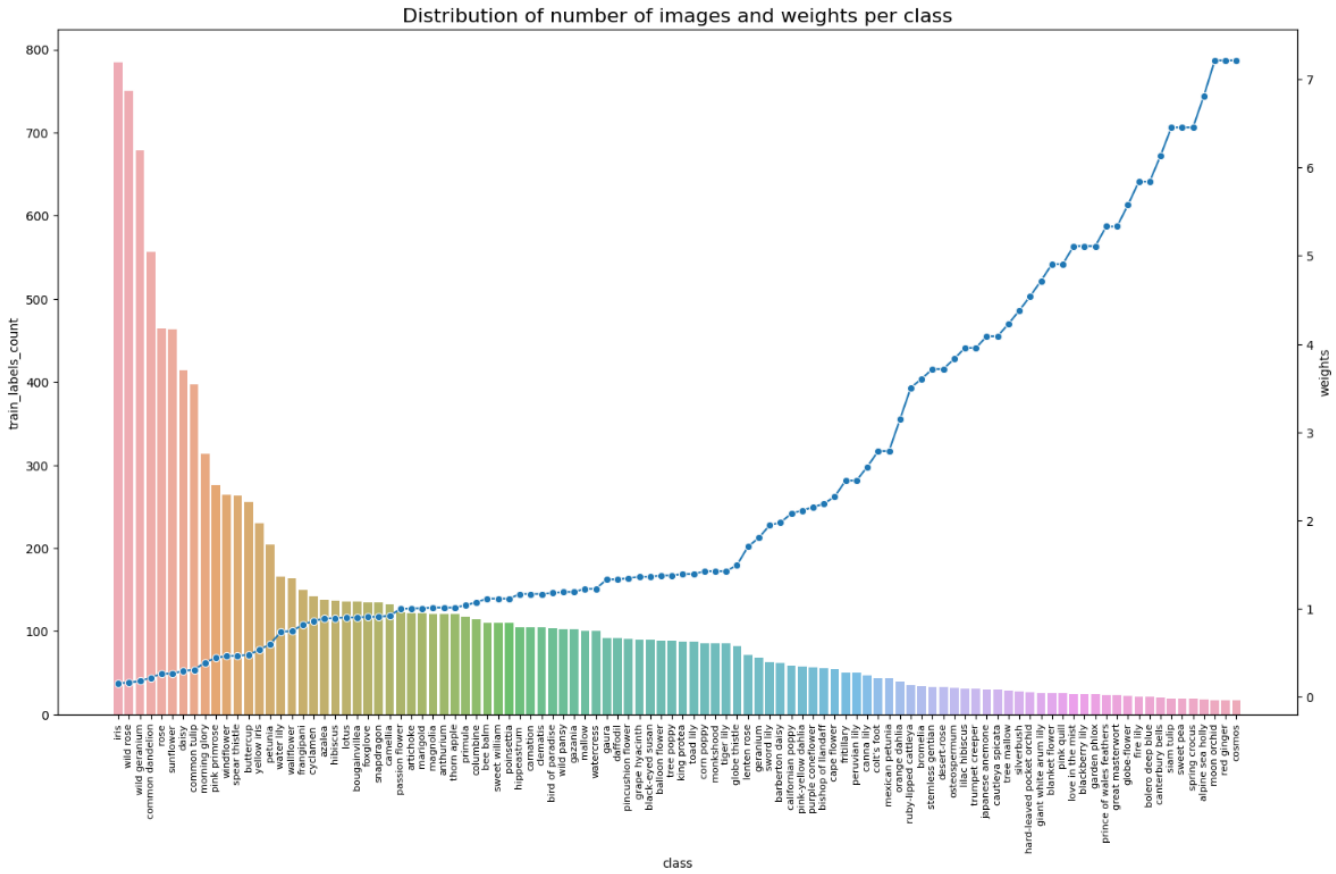


Le jeu de données est donc déséquilibré car plusieurs classes ont petites proportions par rapport à d'autres classes. Pour éviter que les modèles aient mauvaises performances par rapport aux classes minoritaire on peut adopter deux techniques :

1. 'Sampling methods' qui consistent à créer des nouveaux échantillons pour les classes minoritaires ou supprimer des échantillons des classes majoritaires
2. 'Cost-sensitive methods' qui consistent à augmenter le poids des échantillons des classes minoritaires par rapport aux classes majoritaires afin d'obtenir des poids équilibrés par rapport aux nombre relatif d'échantillons entre les classes.



On adopte la deuxième approche avec l'utilisation de la fonction class_weight de la librairie sklearn.



4. Data augmentation

Le processus de data augmentation des données dérive de nouvelles images de celles existantes en appliquant des transformations géométriques aléatoires afin d'incorporer un niveau de variation dans le jeu de training qui permet au modèle de mieux généraliser sur des données inconnues. L'augmentation des données du jeu de training est extrêmement puissante en termes d'augmentation de la précision et de réduction de l'overfitting.

Les transformations appliquées au jeu de training sont :

1) Random blockout - sélection au hasard d'une région rectangle dans une image et remplacement de ses pixels par des pixels noirs.

2) Random_flip_left_right - retournement de l'image au hasard horizontalement (de gauche à droite).

3) Random_saturation – ajustement de la saturation de l'image RGB par un facteur aléatoire.

4) Random_contrast - ajustement du contraste de l'image par un facteur aléatoire.

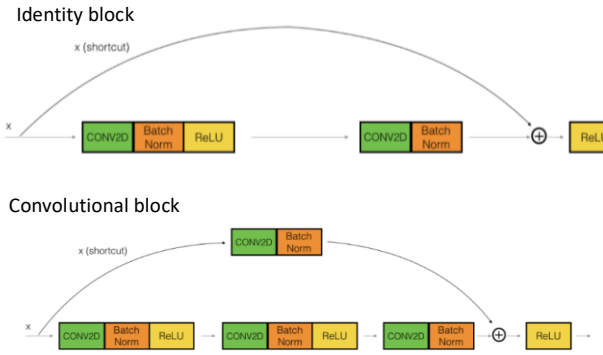
5) Rotation aléatoire - rotation de l'image dans le sens antihoraire par un angle aléatoire.

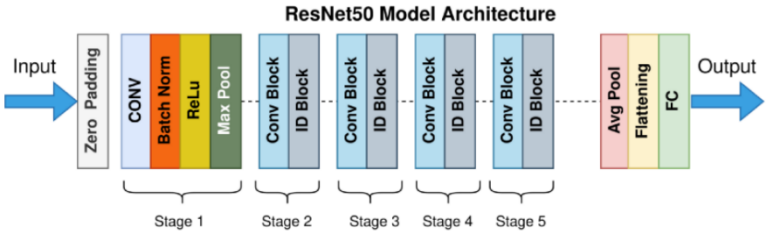
5. Classification des images

Pour la classification des images on utilise des modèles pre-entrainé sur un jeu de données de grande dimension (e.g. Imagenet) avec la technique de transfer learning avec Fine Tuning Totale.

6. Modèle ResNet50

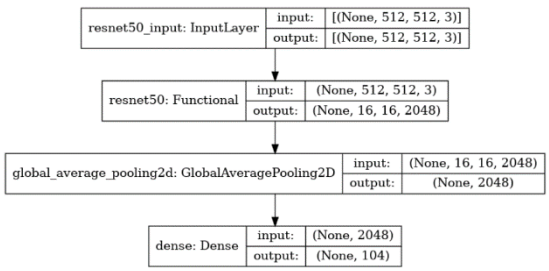
L'architecture ResNet (Residual Network) a été introduit afin de faciliter l'entrainement des réseaux profonds en éliminant le problème de dégradation liée à la diminution de performance de l'architecture CNN classique avec l'augmentation de la profondeur du réseau. Le modèle ResNet50, caractérisé par une architecture simple avec 'Identity blocks' et 'Convolutional blocks', a été donc utilisé comme modèle baseline.

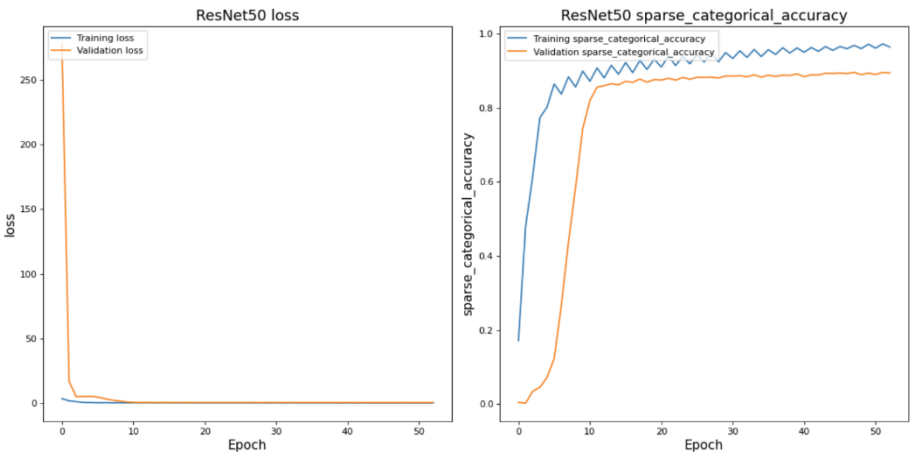





```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resnet50 (Functional)        (None, 16, 16, 2048)      23587712
_________________________________________________________________
global_average_pooling2d (Gl (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 104)               213096
=================================================================
Total params: 23,800,808
Trainable params: 23,747,688
Non-trainable params: 53,120
_________________________________________________________________
```

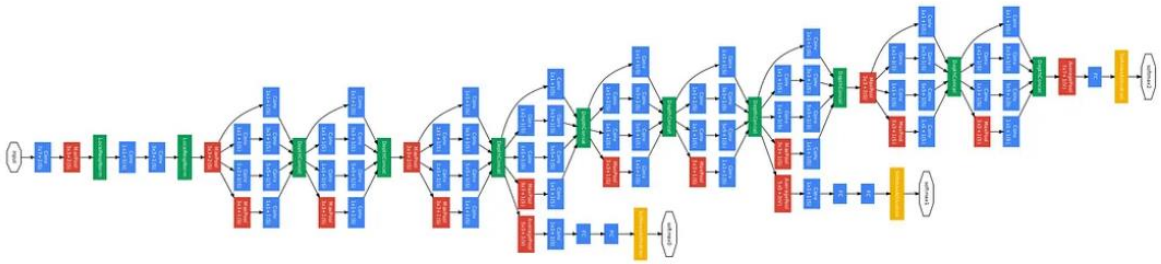
	parameters	epochs	min_loss	max_train_accuracy	max_val_accuracy	F1_score	precision	recall
ResNet50	23800808	41	0.438	0.971	0.898	0.892	0.889	0.901



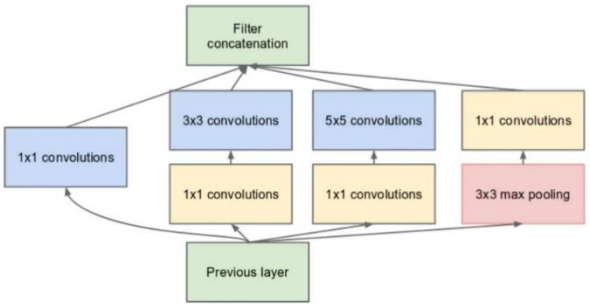


7. Inception Network

L'architecture Inception introduit le bloc Inception qui contient plusieurs couches de convolution et de pooling superposées, pour obtenir de meilleurs résultats et réduire le nombre de paramètres et donc les ressources de calcul par rapport à un réseau CNN standard



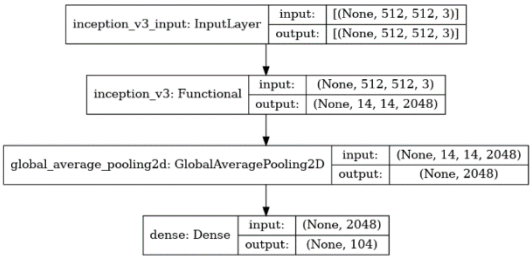
Inception bloc



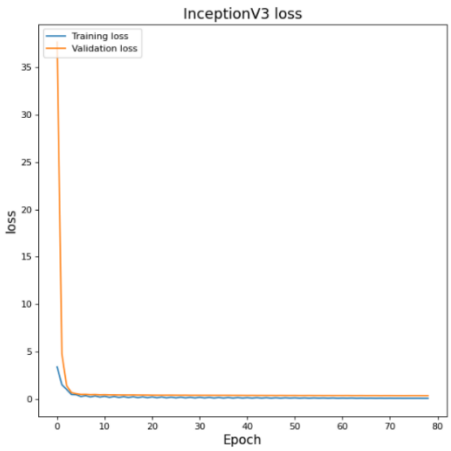
	# parameters	FLOPS
Inception	0.16 M	128 M
3x3 Conv	0.44 M	346 M
5x5 Conv	1.22 M	963 M

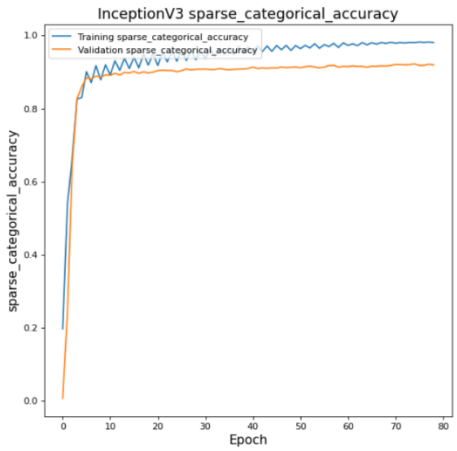
Table 1. Inception blocks vs 3×3 and 5×5 Convolutional blocks — to create 256 output channels (Source: Image created by author)

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inception_v3 (Functional)    (None, 14, 14, 2048)      21802784
_________________________________________________________________
global_average_pooling2d (Gl (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 104)               213096
=================================================================
Total params: 22,015,880
Trainable params: 21,981,448
Non-trainable params: 34,432
_________________________________________________________________
```



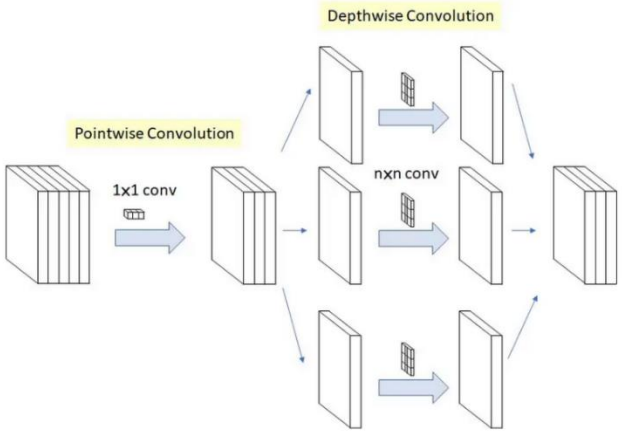
	parameters	epochs	min_loss	max_train_accuracy	max_val_accuracy	F1_score	precision	recall
ResNet50	23800808	41	0.438	0.971	0.898	0.892	0.889	0.901
InceptionV3	22015880	79	0.325	0.982	0.922	0.918	0.920	0.922





8. Xception Network

Xception est une extension de l'architecture Inception qui remplace le bloc Inception avec le bloc Modified Depthwise Separable Convolutions constitué d'une Pointwise Convolution suivie d'une Depthwise Convolution.



The Modified Depthwise Separable Convolution used as an Inception Module in Xception, so called "extreme" version of Inception module (n=3 here)

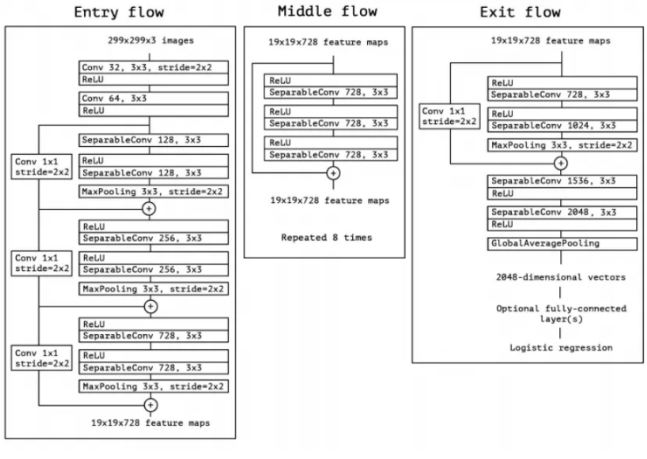
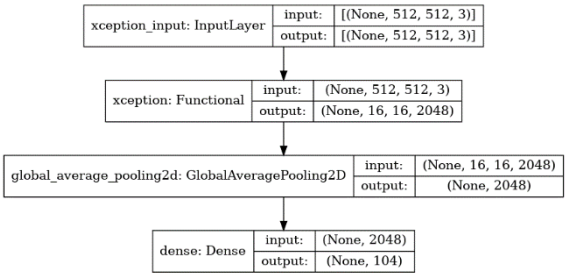


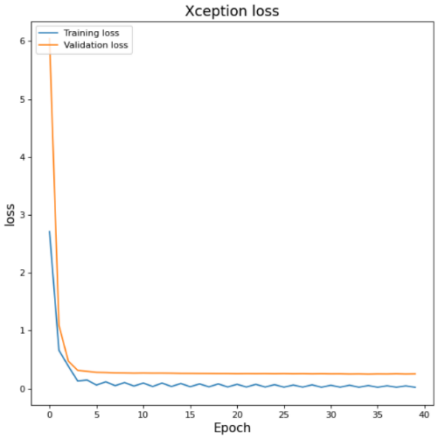
Figure 1. Xception architecture (Source: Image from the original paper)

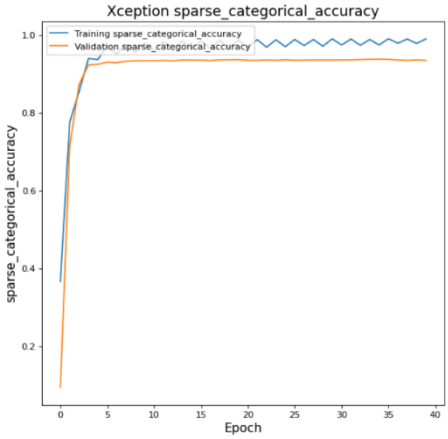
Par rapport à la convolution conventionnelle, nous n'avons pas besoin d'effectuer la convolution sur tous les canaux. Cela signifie que le nombre de connexions est moindre et que le modèle est plus léger.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
xception (Functional)        (None, 16, 16, 2048)      20861480
_________________________________________________________________
global_average_pooling2d (Gl (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 104)               213096
=================================================================
Total params: 21,074,576
Trainable params: 21,020,048
Non-trainable params: 54,528
_________________________________________________________________
```



	parameters	epochs	min_loss	max_train_accuracy	max_val_accuracy	F1_score	precision	recall
ResNet50	23800808	41	0.438	0.971	0.898	0.892	0.889	0.901
InceptionV3	22015880	79	0.325	0.982	0.922	0.918	0.920	0.922
Xception	21074576	40	0.250	0.991	0.938	0.942	0.942	0.946

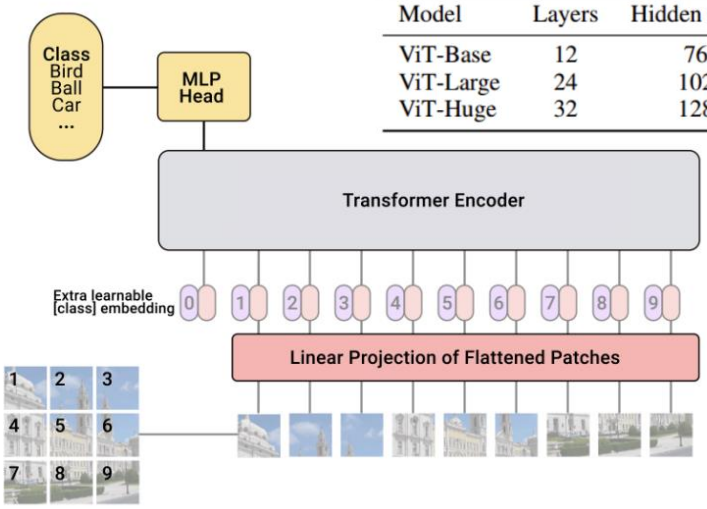




9. ViT - Vision Transformer

Vision Transformer est un modèle de classification d'image récent (Oct 2020) qui remplace le réseau CNN (Convolutional Neural Network) avec une architecture transformer-based.

Les Transformers ont représenté la base pour le développement d'algorithmes NLP très performants comme BERT, GPT-3 et ViT est la première implémentation des transformers qui a depassé le SOTA avec plusieurs benchmarks.



Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

L'architecture de ViT consiste des blocs suivants :

9.1. Split de l'image en patches

ViT fait un split de l'image avec sections de 16x16 pixels, non superposés. On obtient donc une matrice [16x16x3] pour chaque section.

9.2. Flattening des patches

Les sections sont 'flattened' et on obtient des vecteurs de dimension 768, 1024 ou 1280 selon les modèles.

9.3. Linear Embedding de dimension réduite des vecteurs

On projette les vecteurs sur une espace de dimension réduite et on obtient une 'embedded version' des vecteurs flattened.

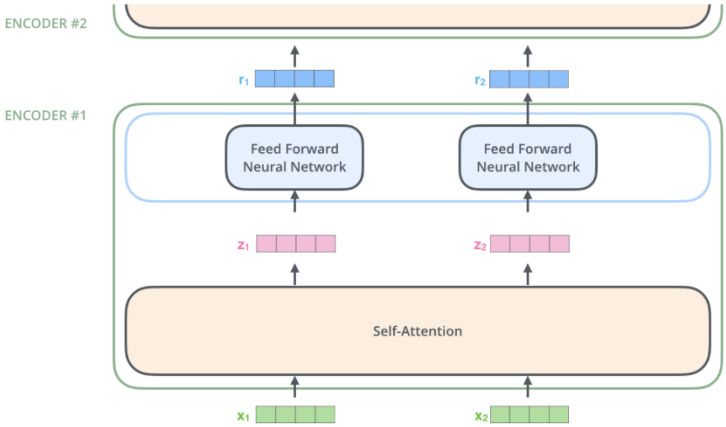
9.4. Position Embedding vecteur

On concatène une classe d'apprentissage avec les autres embedding vectors, dont l'état à la sortie contient l'information de la classe. Ce vecteur supplémentaire est donc responsable de l'agrégation des informations d'image globale et de la classification finale. Il est capable d'apprendre cette agrégation globale pendant qu'il passe et apprend à travers les couches d'attention.

Un vecteur 'position embedding' entrainable est ajouté aux vecteurs pour indiquer la position de chaque section dans l'image initiale. La dimension des embedded vecteurs est 512.

9.5. Transformer Encoder

On envoie les séquences obtenues comme input pour un Transformer Encoder qui consiste de 12, 24 ou 32 encoders selon les modèles.
Les encodeurs sont tous identiques dans leur structure (mais ils ne partagent pas de poids). Chacun est divisé en deux sous-couches :



Le **Self-Attention layer** est une couche qui aide l'encodeur à regarder les autres patches dans la séquence d'entrée pendant qu'il encode un patch spécifique.

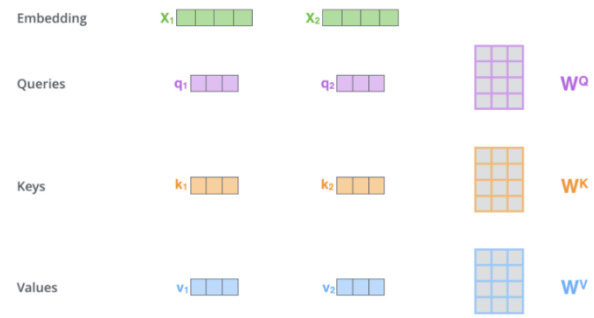
Les sorties du Self-Attention layer sont transmises à un **FF neural network**. Le même FF network est utilisé pour chaque vecteur.

Après le passage des vecteurs dans La couche 'Self-Attention' et dans le réseau de neurones FF, l'encoder envoie la sortie vers l'encoder suivant.

9.6. ViT Encoder - Self-Attention layer

La **première étape** du calcul du Self-Attention layer consiste à créer trois vecteurs à partir de chacun des vecteurs d'entrée de l'encodeur (embedded vectors) : **Query vector, Key vector et Value vector.**

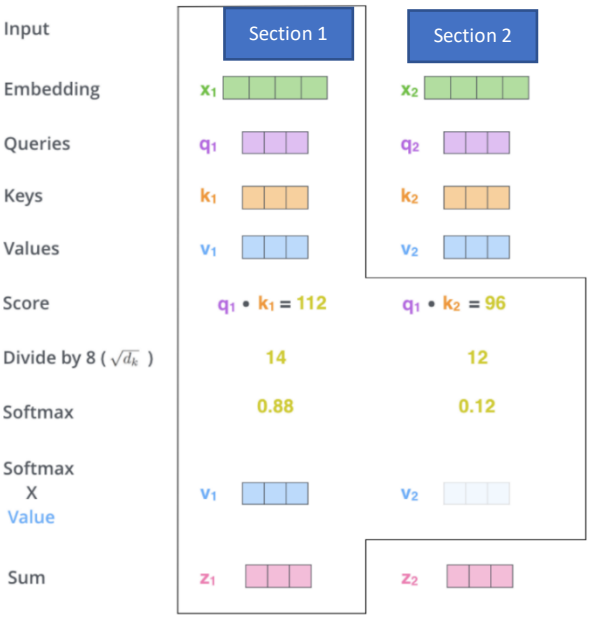
Ces vecteurs sont créés en multipliant les embedded vecteurs par trois matrices que nous avons formées pendant l'entrainement. La dimension des trois vecteurs est réduite à 64.



La **deuxième étape** du calcul du Self-Attention layer consiste à calculer un **score**. On doit donc noter chaque section de l'image d'entrée (vecteur embedded) par rapport à toutes les autres sections. Le score détermine le focus à placer sur d'autres parties de l'image d'entrée (vecteurs) lors que nous encodons une certaine partie (vecteur) à une certaine position.

Les **troisième et quatrième étape** consistent à diviser les scores par 8 (la racine carrée de la dimension des principaux vecteurs utilisés dans le document – 64) pour stabiliser les gradients et passer ensuite le résultat à travers une opération softmax qui normalise les scores afin qu'ils soient tous positifs et additionnent jusqu'à 1.

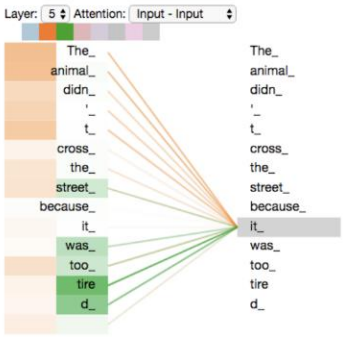
La **cinquième étape** consiste à multiplier chaque Value vector par le score softmax pour conserver intactes les valeurs des parties d'image sur lesquels nous voulons nous concentrer et noyer les parties non pertinentes. La **sixième étape** consiste dans la somme des Value vectors



pondérés. Ceci produit la sortie de la couche de Self-Attention à cette position (pour la première section de l'image) envoyée ensuite au FF network.

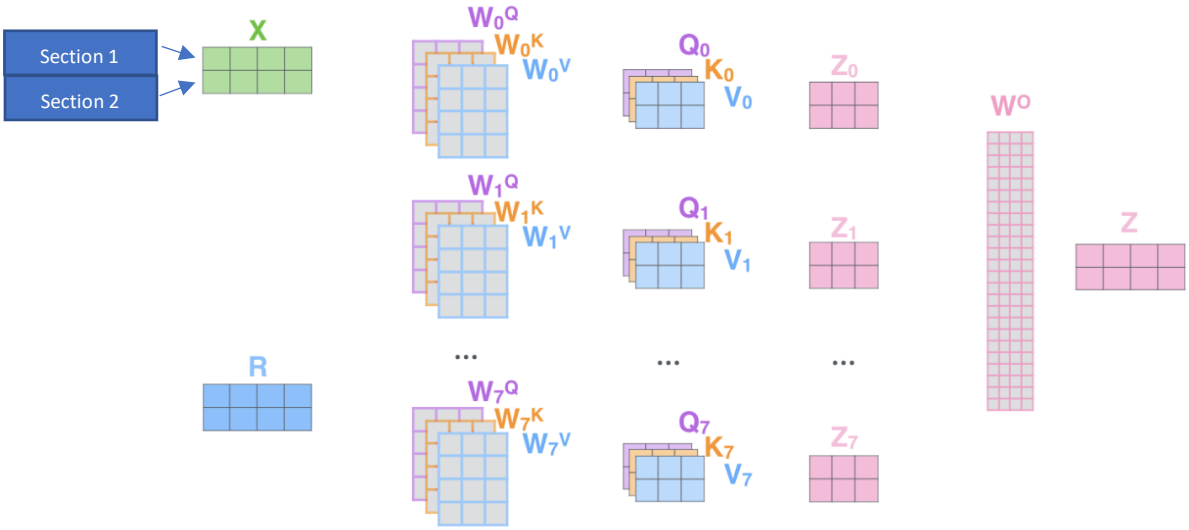
9.7. ViT Encoder - Multi Headed Self-Attention layer

Le Multi-Headed Self-Attention layer accroît la capacité du modèle de se concentrer sur différentes positions dans l'image.
Ci-contre un exemple de l'application d'un Multi-Headed Self-Attention layer pour NLP.

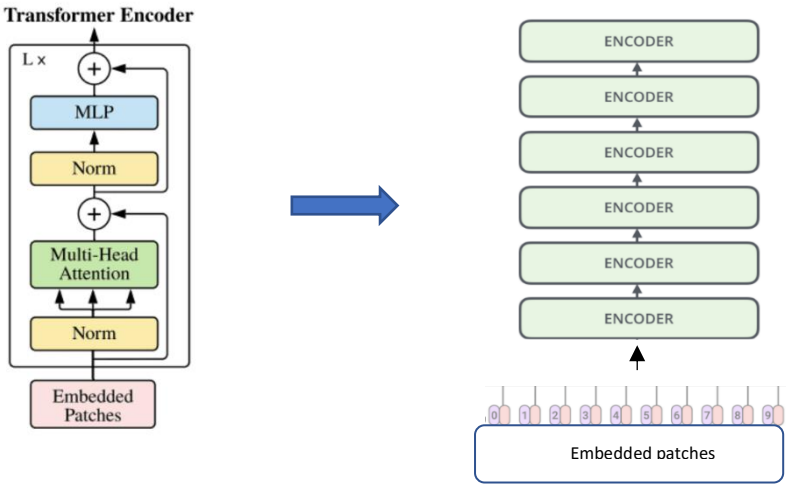


As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

Avec une Multi-Headed Self-Attention nous avons plusieurs ensembles de matrices de poids de Query/Key/Value. L'Encoder utilise 12 ou 16 Self-Attention layers selon les modèles, donc nous finissons avec 12 ou 16 sets pour chaque encodeur. Pour le premier encoder l'input est X pour les suivants il est R, sortie de l'encoder précèdent)



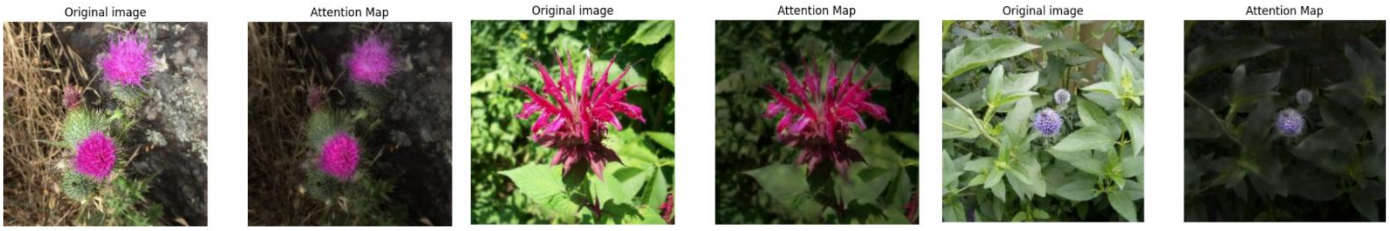
On obtient en sortie 12 ou 16 différentes matrices Zi qui on concatène dans une matrice Z et on ramène à la dimension d'entrée du FF network avec la multiplication pour une matrice poids W0 entrainée avec le modèle.
Chaque sous-couche (Multi-Head Self-Attention layer, MLP) dans chaque encoder a une connexion résiduelle autour de lui et elle est précédée d'une étape de normalisation.



Le bloc MLP final, également appelé tête MLP est utilisé comme sortie du transformateur à laquelle on envoie le vecteur Z0 de la matrice Z de sortie. L'application de softmax sur cette sortie permet enfin d'obtenir les étiquettes de classification.

9.8. Attention maps

On a visualisé les Attention maps sur l'image d'entrée avec la fonction 'visualize.attention_map' qui calcule la moyenne des 'Attention Weights' entre toute les têtes du Multi-Head Self-Attention de chaque Encoder et ensuite multipliés de façon récursive par les matrices de poids de chaque layer. Le masque obtenu est donc appliqué sur l'image d'origine.

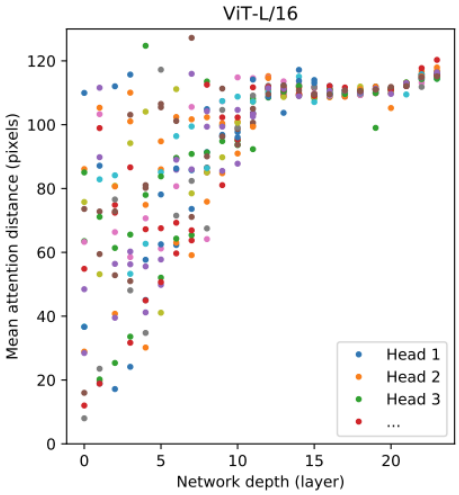


9.9. Efficacité du modèle

Les caractéristiques principales qui rendent les modèles CNN adaptées aux tâches de classification d'images sont :

- la **locality**, basée sur l'assomption que les pixels plus proches entre eux sont plus importantes des pixels éloignés et pour cette raison l'opérateur convolution du CNN est capables d'extraire efficacement les détails de l'image (bords et angles)
- la **Translation equivariance** qui exprime la propriété de translation de la sortie de l'opération de convolution de la même quantité par rapport à une translation de l'entrée. Donc le même filtre sera capable de détecter les mêmes caractéristiques sur différentes parties de l'image
- la **Translation invariance**, qui exprime la propriété de la convolution d'être invariante par rapport à petites translation de l'entrée

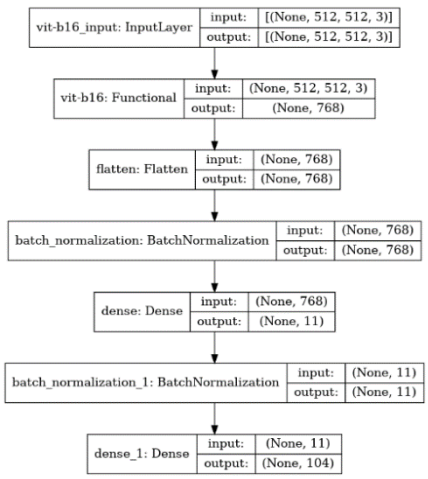
Le modèle ViT n'a pas les mêmes propriétés des modèles CNN et donc demande un entrainement plus profond avec plus de data pour apprendre les mêmes caractéristiques d'un modèle CNN.
En revanche le modèle CNN est capable dans les premières layers de détecter exclusivement des caractéristiques locales de l'image et devient capables de détecter des caractéristiques globales sur l'image seulement dans les layers plus profonds différemment du modèle ViT qui peut extraire des caractéristiques globales du premier layer.



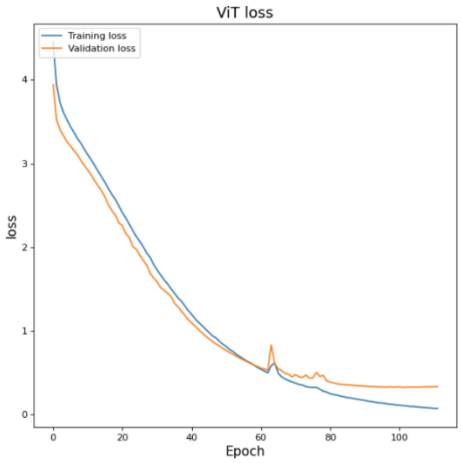
9.10. Entrainement et performances du modèle

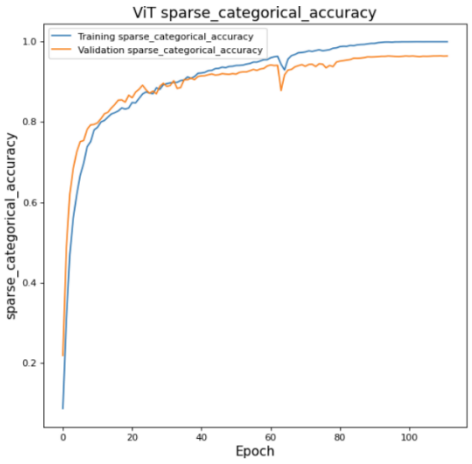
Pour l'entrainement du ViT avec le jeu de données Petals to the Metal, on importe le modèle ViT B16 implementé en Keras et pre-entrainé avec le jeu de données ImageNet 2012.

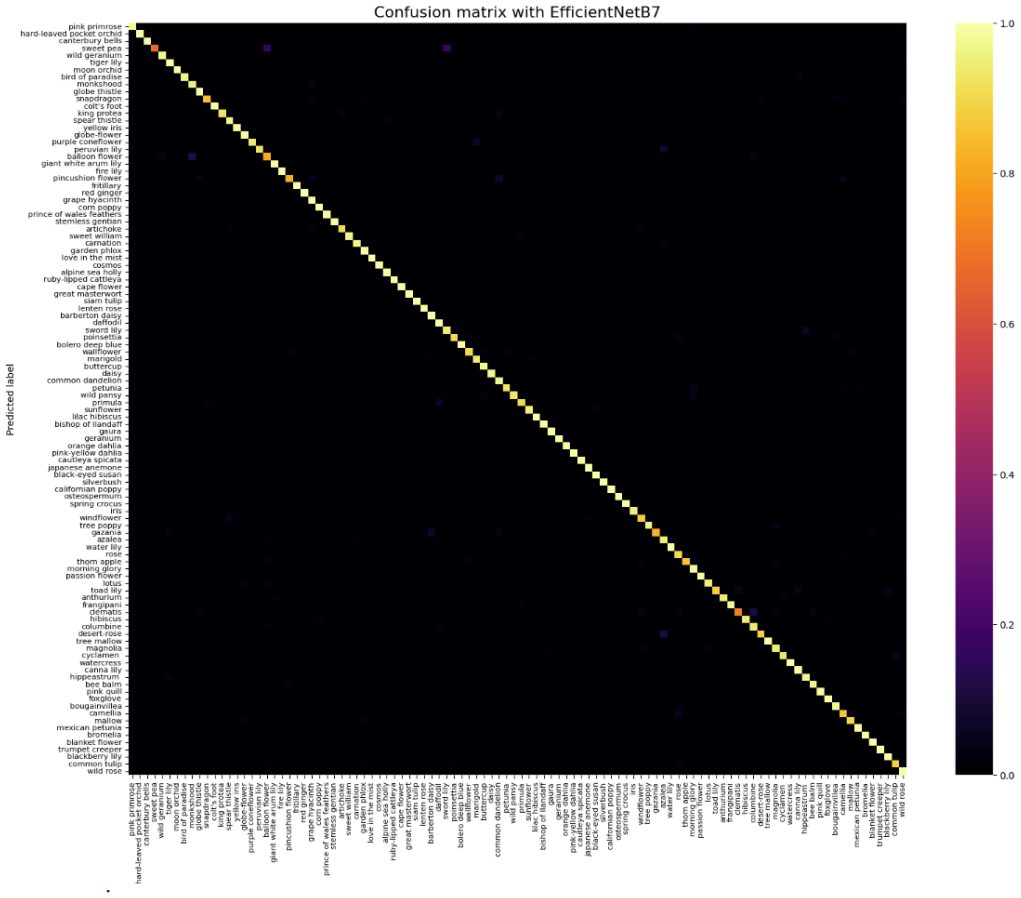
Ensuite on utilise le Transfer Learning avec Fine Tuning avec remplacement du bloc MLP Head de sortie avec un Dense layer avec GELU (Gaussian Error Linear Unit) activation function et Softmax layer pour la classification en sortie.




```
Model: "vision_transformer"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vit-b16 (Functional)         (None, 768)               86434560
_________________________________________________________________
flatten (Flatten)            (None, 768)               0
_________________________________________________________________
batch_normalization (BatchNo (None, 768)               3072
_________________________________________________________________
dense (Dense)                (None, 11)                8459
_________________________________________________________________
batch_normalization_1 (Batch (None, 11)                44
_________________________________________________________________
dense_1 (Dense)              (None, 104)               1248
=================================================================
Total params: 86,447,383
Trainable params: 86,445,825
Non-trainable params: 1,558
```







ViT	86447383	112	0.320	1.000	0.965	0.960	0.962	0.961

10. EfficientNet

EfficientNet est un groupe de modèles de réseau convolutionnel qui a permis d'atteindre très bonnes performances avec la base de données Imagenet avec très peu de paramètres par rapport aux autres modèles. Le groupe de modèles EfficientNet comprend 8 modèles, de B0-B7, où chaque modèle renvoie chronologiquement à une plus grande précision et à un plus grand nombre de paramètres.

EfficientNet définit une architecture de base très simple EfficientNet-B0 et une méthode efficace, le 'Compound Scaling', pour augmenter la taille du modèle afin d'obtenir une amélioration des performances.

MBConv est le bloc principal du modèle EfficientNet. MBConv est constitué d'une 'Shortcut Connection' entre le début et la fin d'un bloc convolutif.

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

Table 1. Architecture Details for the baseline network

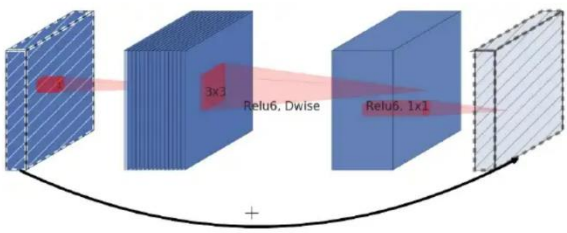


Figure 1. Inverted residual block

On applique d'abord des convolutions 1x1 aux feature maps d'entrée pour augmenter la profondeur et ensuite des convolutions 3x3 Depth-wise et Point-wise qui réduisent le nombre de canaux des feature maps en sortie. Cette structure permet de réduire le nombre total d'opérations requises ainsi que la taille du modèle.

La méthode de 'Compond Scaling' considère que les différentes propriétés d'un réseau de neurones sont interdépendantes. Lorsque nous augmentons la largeur ou la profondeur de l'architecture ou même la résolution de l'image d'entrée, nous obtenons généralement une meilleure précision. Mais au-delà d'une certaine limite, cela n'améliore pas la performance du modèle. Par conséquent, il est important d'équilibrer ces trois éléments.

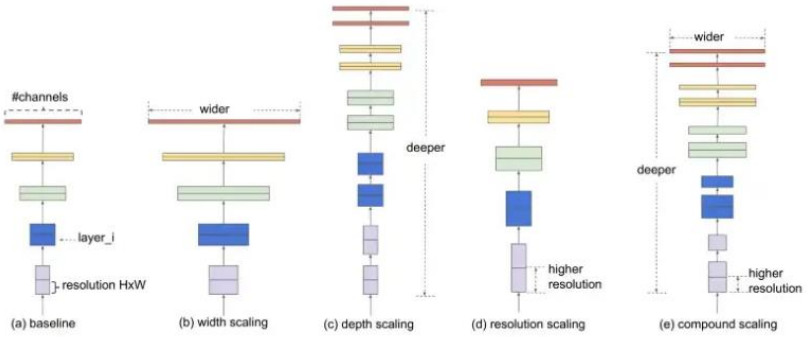


Figure 1. Different scaling methods vs. Compound scaling (Source: image from the original paper)

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

La technique 'Compound Scaling' définit donc les valeurs des résolution, profondeur et largeur basée sur les relations suivantes :

ϕ est un facteur défini par l'utilisateur qui contrôle le nombre de ressources disponibles

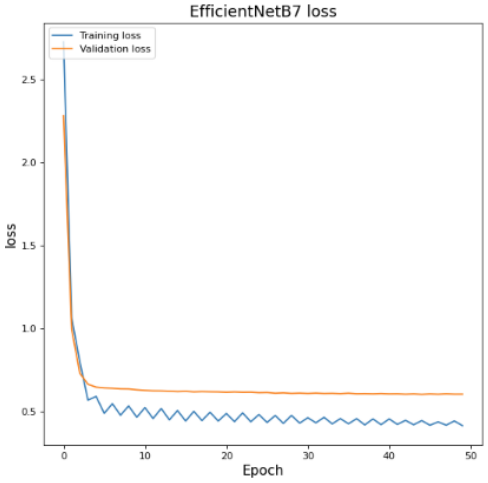
α, β et γ déterminent la façon d'affecter ces ressources à la profondeur, à la largeur et à la résolution du réseau respectivement.

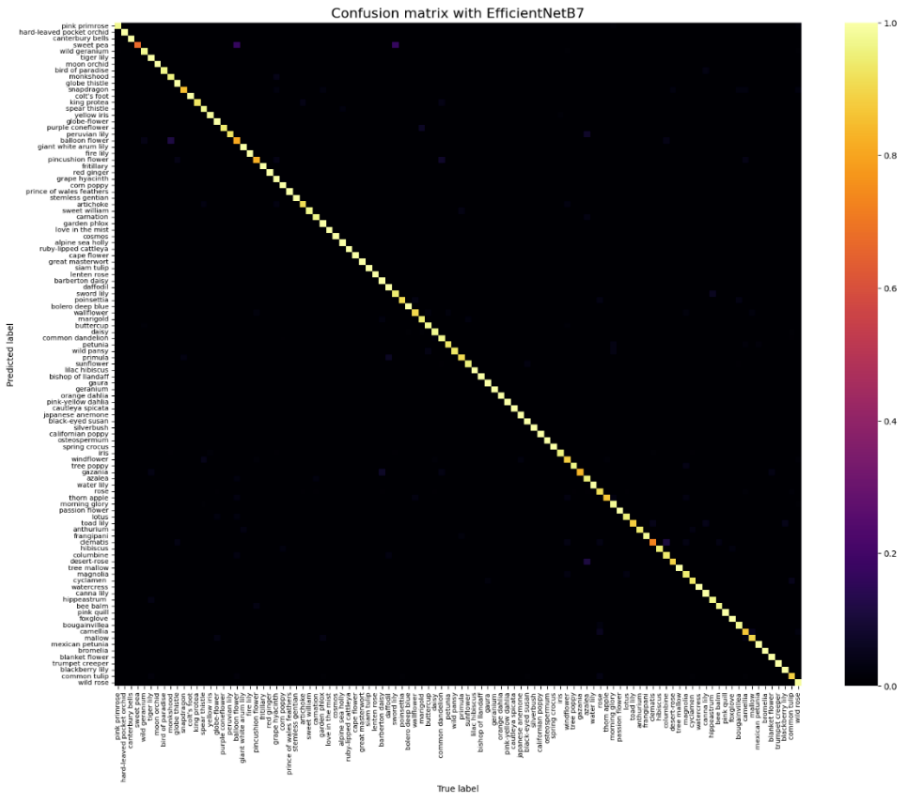
Les FLOPS d'une opération de convolution sont proportionnels à d, w^2 et r^2 et donc une opération de scaling de l'architecture va augmenter les FLOPS de $(\alpha * \beta^2 * \gamma^2)^{\phi}$. Afin d'éviter que le nombre de FLOPS dépasse 2^{ϕ} on applique la contrainte $(\alpha * \beta^2 * \gamma^2) \approx 2$ sur l'architecture baseline B0. Donc si on double les ressources on peut fixer ϕ=1 pour doubler le nombre de FLOPS.
α, β et γ sont déterminé à l'aide d'une Gridsearch avec ϕ=1 et en trouvant les paramètres qui donnent la meilleure précision avec la contrainte $(\alpha * \beta^2 * \gamma^2) \approx 2$.
Une fois trouvés, ces paramètres peuvent être fixés, et le compound coefficient ϕ peut être augmenté pour obtenir des modèles plus grands mais plus précis. C'est ainsi que EfficientNet-B1 à EfficientNet-B7 sont construits.

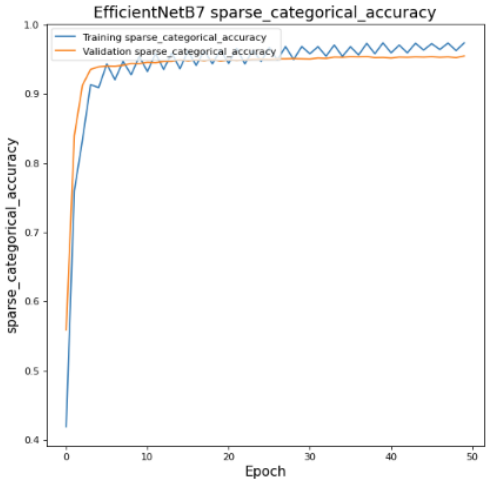
10.1. Entrainement et performances du modèle

Pour le modèle de classification avec images de taille 512x512 on a choisi le modèle EfficientNetB7.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda (Lambda)              (None, 512, 512, 3)       0
_________________________________________________________________
keras_layer (KerasLayer)     (None, 2560)              64097680
_________________________________________________________________
dense (Dense)                (None, 104)               266344
=================================================================
Total params: 64,364,024
Trainable params: 64,053,304
Non-trainable params: 310,720
```

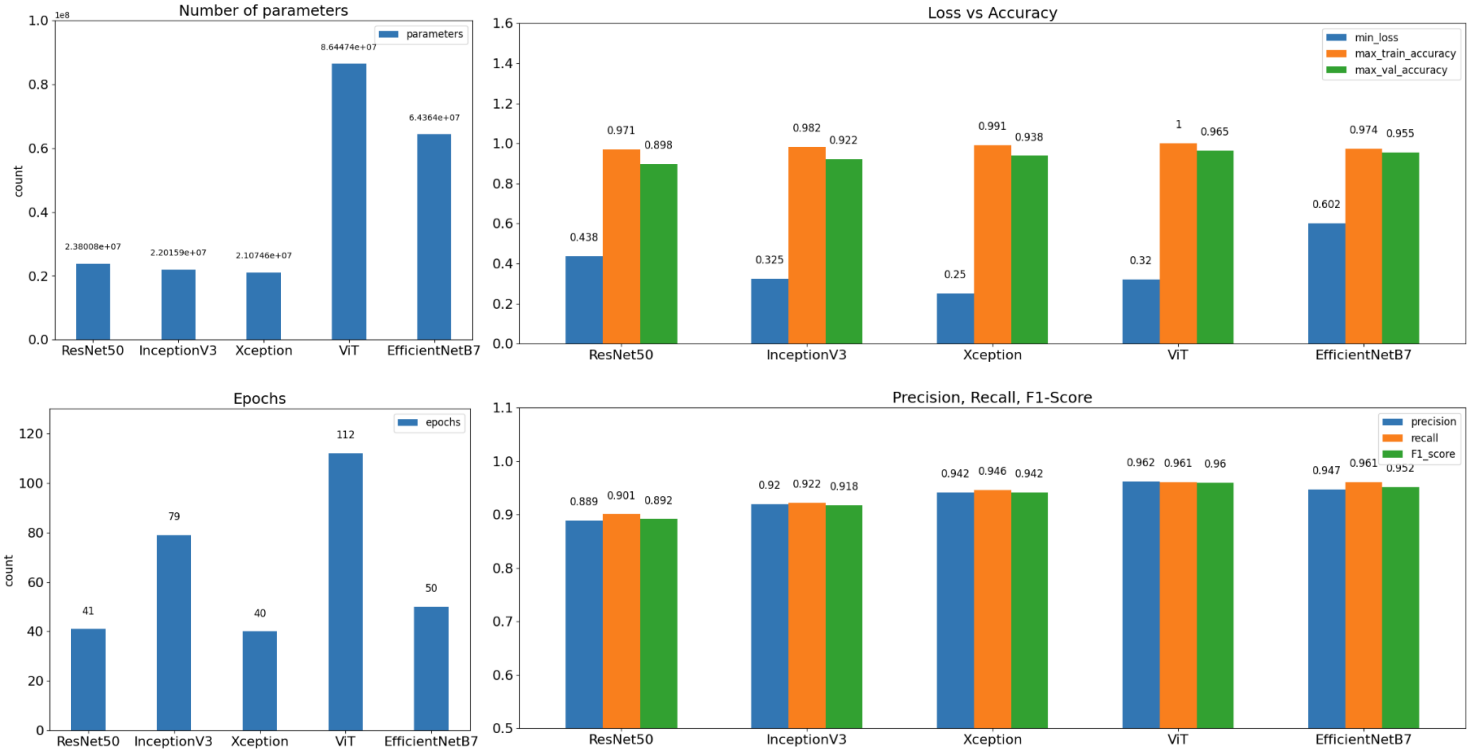






	parameters	epochs	min_loss	max_train_accuracy	max_val_accuracy	F1_score	precision	recall
ResNet50	23800808	41	0.438	0.971	0.898	0.892	0.889	0.901
InceptionV3	22015880	79	0.325	0.982	0.922	0.918	0.920	0.922
Xception	21074576	40	0.250	0.991	0.938	0.942	0.942	0.946
ViT	86447383	112	0.320	1.000	0.965	0.960	0.962	0.961
EfficientNetB7	64364024	50	0.602	0.974	0.955	0.952	0.947	0.961

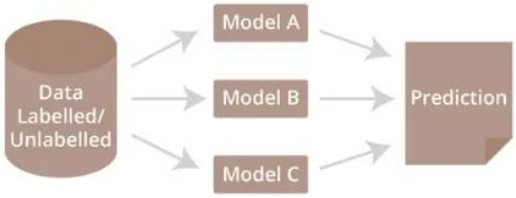
11. Evaluation des performances des modèles



Le modèle de classification plus performant est donc Vision Transformer suivi par EfficientNetB7 et Xception. Vision Transformer est le modèle plus complexe en termes de nombre de paramètres et celui qui a été entrainé plus longtemps.

12. Modèles Ensemble

Afin de maximiser la performance sur le jeu de test on utilise les Ensembles Models qui sont des modèles prédictifs qui combinent les prévisions de deux ou plusieurs autres modèles. Les modèles qui contribuent à l'ensemble, peuvent être du même type ou de types différents et peuvent ou non être formés sur les mêmes données de formation.



Les prédictions faites par les membres de l'ensemble peuvent être combinées à l'aide de statistiques, comme le mode ou la moyenne.

Il y a deux raisons principales d'utiliser un ensemble plutôt qu'un seul modèle, et elles sont liées :

- Performance : un ensemble peut faire de meilleures prévisions et obtenir de meilleures performances que n'importe quel modèle contributif.
- Robustesse : un ensemble réduit la dispersion des prédictions et de la performance du modèle.

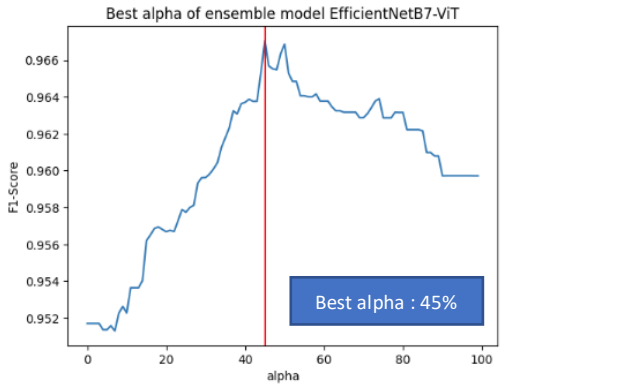
Les modèles Ensemble sont divisé en quatre catégories :

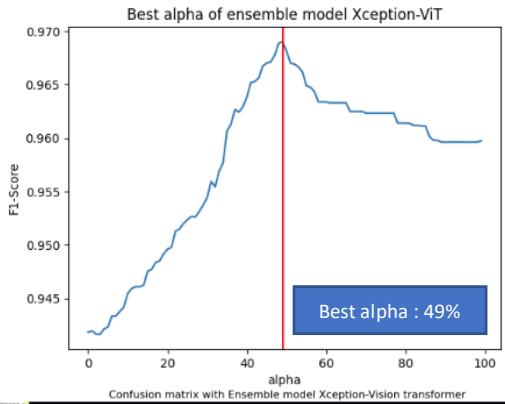
- Sequential methods – Méthodes avec 'base learners' qui utilisent données d'entrée qui dépendent du 'base learner' précèdent (BOOSTING)
- Parallel methods – Méthodes qui utilisent données d'entrée indépendants (RANDOM FOREST)
- Homogeneous Ensemble" : combinaison de la même typologie de classifieur
- Heterogeneous Ensemble" : combinaison de classifieurs de typologies différentes

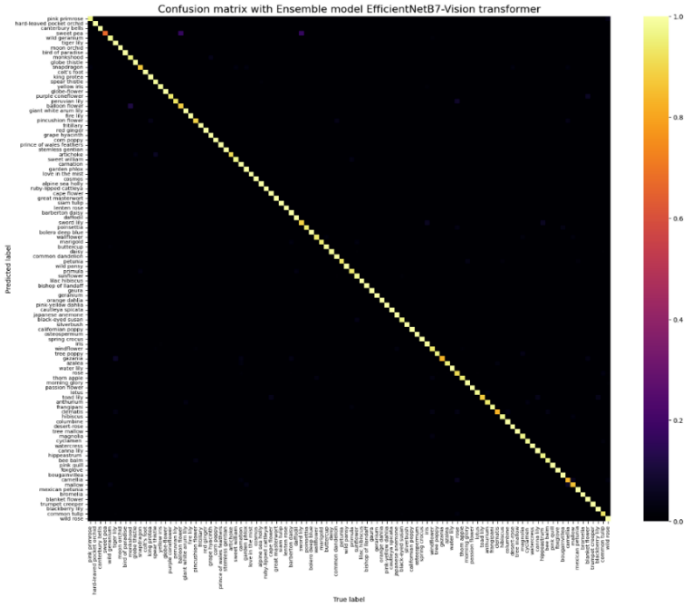
On a utilisé une méthode parallèle avec un Heterogeneous ensemble car on a combiné avec une moyenne pesée les résultats des classifieurs Xception et Vision Transformer et EfficientNetB7 et Vision Transformer, tous entrainées sur le même jeu de données.

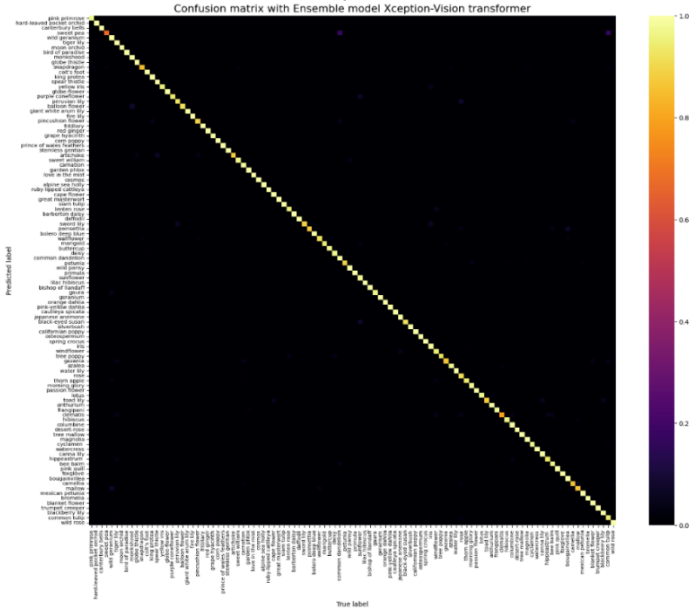
Les poids de la moyenne pesées des résultats ont été calculés sur la base des prédictions sur le jeu de validation des deux modèles afin de maximiser le F1-score.

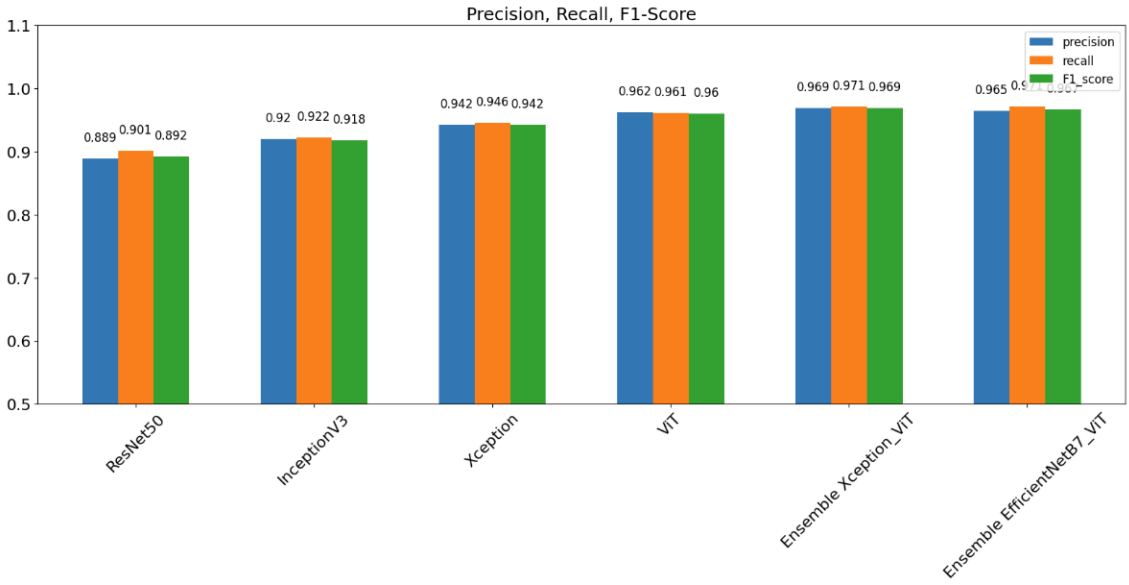
12.1. Performance Ensembles Xception - ViT et EfficientNetB7 - ViT











	F1_score	precision	recall
ResNet50	0.892	0.889	0.901
InceptionV3	0.918	0.920	0.922
Xception	0.942	0.942	0.946
ViT	0.960	0.962	0.961
Xception_ViT	0.969	0.969	0.971
EfficientNetB7_ViT	0.967	0.965	0.971

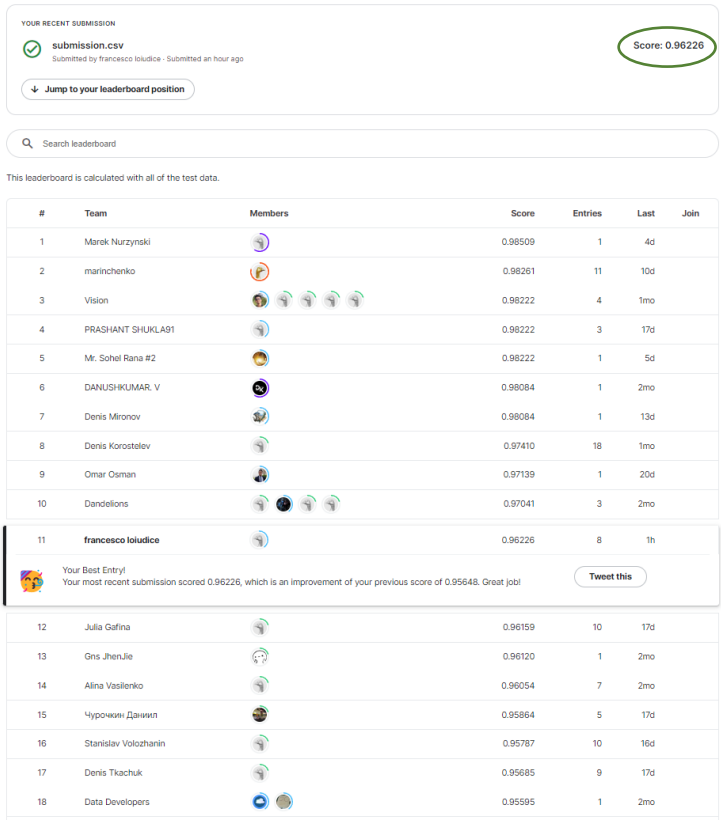
Le modèle plus performent est donc l'Ensemble Xception-Vision Transformer qui est donc sélectionné comme modèle final.

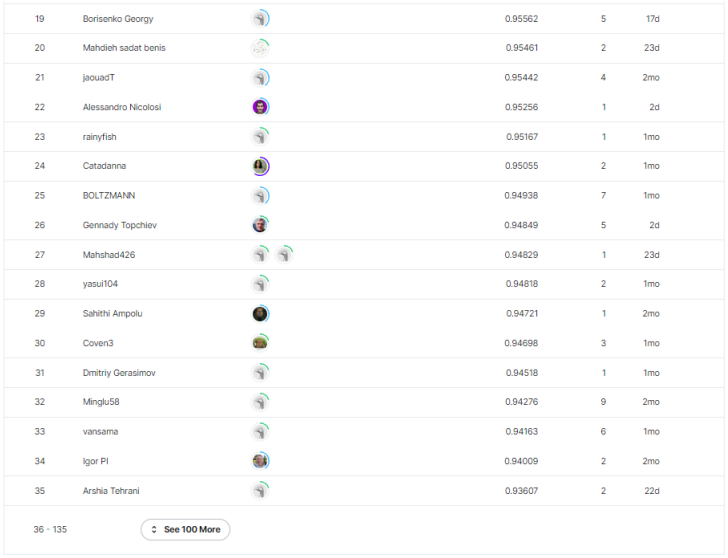
12.2. Performance modèle final sur le jeu de validation

Ci-dessous une validation visuelle pour un batch de 20 images.

Nombre d'erreurs sur le jeu de validation : 117/3712 (3,17%)

12.3. Performance modèle final sur le jeu de test : résultat de la compétition





Score sur jeu de test : 0.96226