Projet PAS :
Deep Learning et traitement d'images

Rapport PAS soumis au

Département IBO de l'ESiLV

Anas Hadhri

Johann de Soyres

Summary

1. Introduction

L'objectif de ce projet et d'effectuer une revue des techniques de traitement d'images basées sur des méthodes d'apprentissage à base de réseaux de neurones puis d'évaluer une ou plusieurs méthodes pour améliorer les performances des algorithmes existants. Pour ce projet nous nous sommes exclusivement penchés sur la classification d'images. La classification d'images consiste à attribuer à une image donnée une classe. On fournit a l'algorithme une image en entrée, puis il doit choisir une classe parmi une liste préalablement définie. Par exemple, si l'on a une image de chien l'algorithme doit être capable de dire que c'est un chien plutôt qu'un chat. Ces dernières années le Deep Learning s'est avéré très efficace pour la classification d'images dans différents domaines : imagerie médicale, conduite autonome, cartographie, etc. Pour notre part nous avons choisi le domaine de la peinture pour la classification d'images. Deux sujets d'étude étaient alors possibles :

- Classification d'une peinture par rapport à l'artiste
- Classification d'une peinture par rapport au style artistique

La classification d'images avec le Deep Learning nécessitant un jeu de données important, il était plus facile pour nous d'acquérir un grand dataset avec une classification par rapport à un style artistique. La reconnaissance d'un style artistique peut s'avérer très complexe du fait de la grande variété de peinture au sein d'un même style.

Avant de parler de notre projet nous ferons un bref résumé des techniques existantes sur le sujet. Puis nous parlerons de l'organisation qui a été mise en place pour mener a bien ce projet. Viendra ensuite une description de notre solution. Enfin nous conclurons par un feedback sur les difficultés rencontrées, les pistes d'améliorations, etc.

2. Problématique

Quels sont les techniques d'apprentissage profond les plus pertinentes pour la reconnaissance du style artistique d'une peinture ?

3. Etat de l'art

Le style artistique d'une peinture est un descripteur riche qui capture à la fois des informations visuelles et historiques sur la peinture. Jusqu'à maintenant, il est rare de trouver des modèles qui identifient correctement le style artistique d'une peinture. Dans nos recherches, on s'est inspiré d'un projet fait par des étudiants de l'INSA Rouen et l'université Paris-Dauphine. Leur projet consistait à construire des modèles d'apprentissage profond (deep learning) pour prédire le style artistique d'une peinture. Identifier le style d'une image de manière entièrement automatique est un problème difficile. Bien que les tâches de classification standard telles que la reconnaissance faciale puissent se baser sur des caractéristiques identifiables telles que les yeux ou le nez, la classification des styles visuels ne peut se baser sur aucune caractéristique.

Historiquement, plusieurs documents de recherche universitaire ont abordé le problème de la reconnaissance du style artistique avec des approches d'apprentissage automatique existantes. Par exemple, Florea et al. (2016) ont évalué les performances de différentes combinaisons de caractéristiques d'image populaires (histogrammes de dégradés, spatial, noms de couleurs discriminants, etc.) avec différents algorithmes de classification (SVM,random forests, etc.).

Malgré la taille de l'ensemble de données et le nombre limité d'étiquettes à prédire (seulement 12 mouvements artistiques au total), ils ont observé que plusieurs styles restent difficiles à distinguer avec ces techniques. Ils démontrent également que l'ajout de nouvelles caractéristiques ne va pas améliorer d'avantage la précision des modèles, sans doute à cause de la malédiction de fléau de la dimension. En 2014, Karayev et al. ont observé que la plupart des systèmes conçus pour la reconnaissance du style artistique ont été construits sur des caractéristiques artisanales. Ils réussissent à reconnaître une plus grande variété en utilisant un classificateur linéaire entraîné avec des caractéristiques extraites automatiquement à l'aide du réseau de neurone convolutionnel profond (CNN). Le CNN qu'ils ont utilisé était AlexNet (Krizhevsky et al., 2012) et il était entrainé sur ImageNet pour reconnaître les objets dans les photographies d'objets divers.

Plus récemment, Tan et ses collaborateurs (2016) ont abordé le problème à l'aide du même réseau de neurones et ont réussi à obtenir les meilleures performances avec une procédure entièrement automatique pour la première fois, avec une précision de 54,5% sur 25 styles.

Dans le domaine de la génération d'art visuel, Gatys et al. (2015) ont pu construire un modèle d'un style de peinture spécifique et le transférer ensuite sur des photographies non artistiques. D'un point de vue technique, la génération de l'art n'est pas très différente de la reconnaissance du style artistique : dans les deux cas, la première étape consiste à modeler avec précision un ou plusieurs styles artistiques. Gatys et al. (2015) ont commencé par l'entraînement du modéle VGG profond (Simonyan et Zisserman, 2014) avec un grand nombre de photos d'un style artistique donné. Puis ils ont généré une image qui avait un style correspondant à l'image d'entrée originale. Cependant, une différence importante avec le problème de la reconnaissance du style artistique, est que le modèle doit séparer le style le plus possible du contenu afin de réussir le transfert du style vers un nouveau contenu. A partir de nos recherches, on a construit nos propres modèles afin d'obtenir un modèle le plus adapté à notre problème.

4. Workflow

Pour mener à bien ce projet nous avons utilisé plusieurs outils. Globalement nous avons travailler avec la méthode agile en faisant de sprint de deux semaines. A chaque début de semaine une liste des tâches à accomplir était définie ainsi que leur répartition dans l'équipe. En milieu de sprint il arrivait parfois de faire un point si jamais l'un ou l'autre rencontrai des difficultés ou s'il était nécessaire de redéfinir la tâche. Une fois le sprint terminer une évaluation des objectifs à atteindre était menée. Les outils que nous avons utilisés sont les suivants :

- **Google Colab**

  Nous avons choisi Google Colab comme IDE pour notre projet. Cet IDE a été développé par Google et est totalement gratuit. Il a la particularité d'être un IDE cloud ce qui signifie que les ressources utilises ne sont celles de notre propre ordinateur. Cet IDE présente de nombreux avantages : ressources matériels importantes (serveurs de google à disposition), partage de fichiers interactif, interaction directe avec Google Drive.

- **Google Drive**
  Nous avons stocké nos données sur Google Drive pour faciliter le chargement des données dans notre IDE cloud ainsi que pour éviter la perte des données.

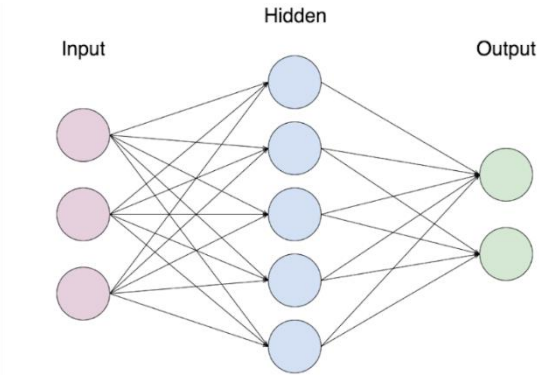
- **Skype**
  Nous avons utilisé Skype pour nos réunions à distance. De plus Skype permet le partage d'écran et le transfert de fichiers.

- **Udemy**
  La plateforme en ligne Udemy fournie des cours sur des sujets divers et varies qui sont très bien détaillés. Nous nous sommes inscrits à un cours sur le fonctionnement du Deep Learning. Ce cours nous a permis de bien comprendre notre problématique.

5. Construction du réseau de neurones

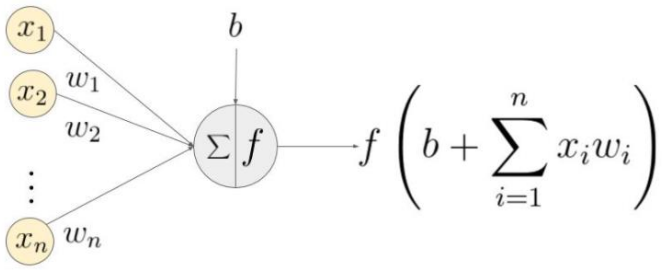
Les modèles de prédiction utilise dans le Deep Learning sont les réseaux de neurones. Ils sont généralement decomposes en trois parties :

- **La couche entrante** (celle qui va recevoir les données d'entrées)
- **Les couches cachées**
- **La couche sortante** (celle qui va donner la prédiction du modèle)



Structure d'un réseau de neurones

Chaque couche est composée d'un ou plusieurs neurones. Dans notre cas chaque neurone de la couche d'entrée va recevoir une couche d'un pixel (R, G ou B) tandis que les neurones de la couche sortantes vont chacun correspondre à un style artistique. Ainsi si notre image fait 150x150 pixels alors notre réseau de neurones aura 67 500 (150x150x3) neurones dans la couche d'entrée. Chaque neurone de la couche sortante va recevoir une valeur comprise entre 0 et 1. Le réseau de neurones sélectionnera comme réponse final le neurone qui a la plus grande valeur. Les neurones des couches cachées ont eu une structure particulière :



Structure d'un neurone d'une couche cachée

Ce neurone va recevoir les données contenues dans chaque neurone de la couche precedente auquel il est connecté. Ces donnees, avant d'être additionnées, vont etre multipliées par un **poids** (valeurs entre 0 et 1). Le tout va ensuite passer en parametre d'une **fonction d'activation** à laquelle on ajoutera un **biais** (valeur en -1 et 1). Au départ tous les poids du réseau sont initialisés avec une valeur prise au hasard entre 0 et 1. Pour faire apprendre notre reseau de neurones on va lui faire des prédictions sur toute une serie de donnees. On appel ce jeu de données le train set. A chaque prédiction, le réseau compare la **réponse obtenue** (yi) avec la **réponse attendue** (ti) et calcule une certaine **erreur** (E). Dans le cas d'une régression le réseau de neurone va utiliser une **fonction de cout** tandis que pour une classification il utilisera la **cross entropie**.

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C}\mathbf{1}_{y_i \in C_c}\log p_{model}[y_i \in C_c]$$

Cross entropie catégorique

$$E=\frac{1}{2}\sum_{i}(y_i-t_i)^2$$

Fonction de cout

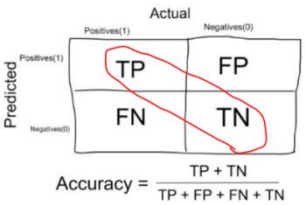
Le réseau va ensuite changer la valeur de tous ses poids de sorte à minimiser cette erreur. Cette minimisation passe par des calculs de dérivées complexes que nous ne detaillerons pas ici. Le réseau va donc faire une prediction et une correction de ses poids pour chaque donnée

du jeu de donnée d'entrainement. Une fois que le reseau a effectué toutes ces prédictions on dit qu'il a parcouru **une époque**. On peut faire parcourir au réseau de neurones autant d'epoques que l'on souhaite. Généralement on s'arrete lorsque la l'erreur ne diminue plus.

Pour évaluer notre réseau de neurones, on va se servir du jeu de données de test qu'on appelle test set. Le réseau de neurones n'a jamais vu ce jeu de donnees auparavant. On va lui faire faire des prédictions sur la totalité du jeu de données. Différentes mesures peuvent être utilisées ensuite pour déterminer la pérformance de notre modèle. Dans le cas d'une régression on utilisera la **MSE** (mean squared error), dans le cas d'une classification on utilisera **l'accuracy**.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Formule MSE



Formule Accuracy

La MSE désigne la moyenne des erreurs au carre. L'accuracy est un taux, il désigne la proportion sur laquelle le réseau ne s'est pas trompé. Pour avoir un modèle précis on vise une MSE proche de 0 pour une régression et une accuracy proche de 1 pour une classification.

5.1 Chargement des données

Nous allons maintenant décrire comment nous avons implémenter notre réseau de neurones. Toutes les étapes que nous allons décrire ont été réaliser en langage **Python** dans Colab avec plusieurs librairies connues (**pandas, numpy, PIL, Keras, os, etc.).** La première étape de ce processus a été charger les données dans Colab. Nous avons d'abord récupéré un jeu de données sur **Kaggle**. Kaggle est une plateforme web organisant des compétitions en data science. Sur cette plateforme, les entreprises proposent des problèmes en science des données et offrent un prix aux data scientists obtenant les meilleures performances. C'est donc une excellente source pour avoir des datasets de qualité.

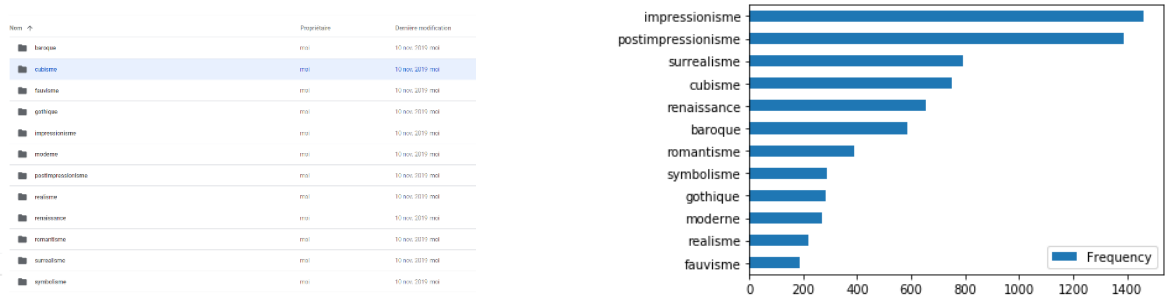
Nous avons ainsi pu trouver un dataset sur des peintures de 50 artistes. On retrouve les artistes les plus connus comme **Van Gogh** ou **Andy Warhol** comme on peut le voir ci-dessous. Chaque artiste contient un certain nombre d'image, chacune représentant une de ses œuvres.

Peintures Van Gogh

Peintures Andy Warhol

Source :*https://www.kaggle.com/paultimothymooney/collections-of-paintings-from-50-artists/data)*

Un premier travail a donc été de réorganiser le dataset de façon à avoir non plus les peintures classes par artiste mais plutôt par style artistique. Certains artistes peuvent avoir des peintures de plusieurs styles artistiques voir meme avoir leur propre style. Pour éviter d'introduire des erreurs dans notre classification, nous avons preferé supprimer les peintures de ces artistes de notre dataset. On se retrouve donc avec un dataset de 1,27 Go contenant 7266 images de peinture repartis à travers 12 styles artistiques différents. On visualiser ci-dessous cette répartition ci-dessous :



Répartition des peintures

Pour faciliter le chargement des donnes dans l'IDE cloud Colab, nous avons placé la totalité du dataset sur un Google Drive. Il nous suffit ensuit de deus lignes de code pour créer un lien entre Colab et notre Drive.

Lien entre Colab et Drive

Pour charger les données nous avons utiliser deux librairies : **PIL.Image** et **os**. Os nous a permis de récupérer le chemin d'accès de chaque image du dataset. PIL.Image permet ensuite de récupérer cette image dans l'IDE à partir de son chemin d'acces. Pour avoir des donnees uniforme, nous avons formater toutes les images à une résolution de 150x150 pixels. Chaque image a été dupliquer avec une version en noir et blanc. Pour réaliser cette opération nous avons utilisé la fonction **grayscale** de la librairie **PIL.ImageOps**. Ceci nous permettra plus tard d'avoir des datasets pour tous nos types de réseaux de neurones. Malgré le fait que nous disposions de ressources importantes sur Colab, le chargement de toutes les images pour l'apprentissage de notre réseau de neurones conduisait systématiquement à un plantage de la session. Un autre problème que nous avons pu constater avec notre dataset est la répartition des peintures à travers les styles. En effet on peut voir par exemple que l'impressionnisme possède plus de 1400 images alors que le fauvisme en contient à peine 200. Avec cette mauvaise répartition on risque d'obtenir une faible précision lors de l'évaluation du modèle pour les classes les moins représentées. Le modèle n'aura pas eu suffisamment de données d'entrainement pour ces classes-là. Pour éviter un plantage de session et avoir une répartition homogène des données entre toutes les classes, nous avons pris au hasard 80 images dans chacun des styles artistiques. On se retrouve donc avec un dataset d'images en noir et blanc et un dataset d'images en couleurs. Chacun des dataset comporte 960 (12 x 80) images.

Il a ensuite été nécessaire de labelliser ces images. Les images étant placée chacune dans un dossier portant le nom du style, il suffisait de récupérer le nom du dossier pour avoir le label de l'image. Le Framework **Keras** n'accepte que des labels sous forme de nombre, c'est pourquoi nous avons attribuer un nombre à chaque style en allant de 0 à 11.

Style	baroque	cubisme	fauvisme	gothique	impressionnisme	moderne	post -impressionnisme	réalisme
Label	0	1	2	3	4	5	6	7

Style	renaissance	romantisme	surréalisme	symbolisme
Label	8	9	10	11

On se retrouve donc avec une liste d'image de 960 (80x12) images et une liste de 960 labels. Il nous a fallu créer un jeu d'apprentissage (**train set**) et un jeu de test (**test set**) pour chacune des listes. Le jeu de. On a séparé ces deux listes en attribuant 75% des données au train set et 25% au test set. Pour s'assurer d'une bonne homogénéité des données entre le train set et le test set nous avons sélection au hasard les données pour chaque subset. Le train set est parcouru un certain nombre de fois par le model pour que celui-ci améliore sa MSE ou son accuracy. Le test set est utiliser pour évaluer les performances du modèle sur des données qu'ils n'ont jamais été vues. Au final on se trouve donc avec 4 groupes de données :

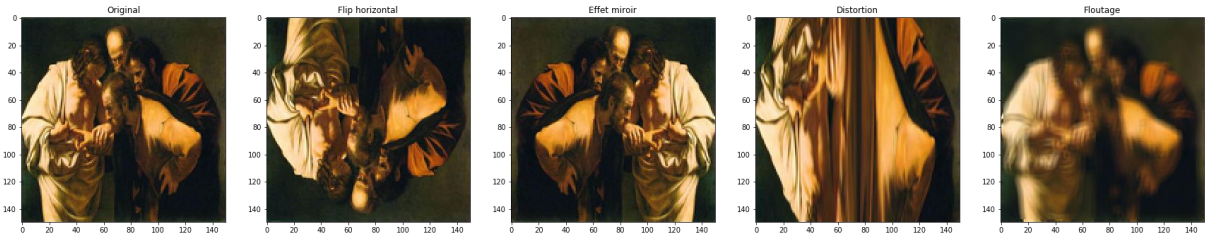
- **Train X** (images du train set)
- **Train Y** (labels du train set)
- **Test X** (images du test set)
- **Test Y** (labels du test set)

5.2 Prétraitements des images

Avant de commencer l'implémentation de nos réseaux de neurones, nous avons réaliser toute une série de pré-traitements sur nos images de sorte à optimiser les performances de nos modèles.

- **Dataset extension**

Comme nous l'avons évoqué précédemment le chargement des données a été très problématique pour nous. A tel point que nous avons dû réduire notre dataset de 7266 images à 960 images (12 x 80). Pour palier à cette perte de données nous avons décidés de dupliquer chacune des images en y appliquant des filtres. Cette approche présente plusieurs avantages, elle permet d'une part d'augmenter notre dataset très rapidement et d'autre part d'accroitre la précision de nos modèles et leurs capacités à se généraliser à des images inconnues. Au total nous avons appliqué 5 filtres :



Filtres appliqués sur une peinture de Caravaggio

Nous avons pu appliquer ces filtres directement avec la librairie **PIL.ImageOps**. Le module ImageOps contient un certain nombre d'opérations de traitement d'images prêt à l'emploi. Il nous a donc suffit d'appliquer les méthodes déjà toutes faites sur nos images. Nous avons multiplié notre dataset de 960 images par 5 soit 4800 images.

- **Normalization**

La **normalization** est une étape essentielle du prétraitement des données dans le domaine du machine learning. La normalization permet de s'assurer d'avoir des données d'entrée qui sont toutes dans la même plage de valeurs. Il a été montré dans quasiment tous les cas d'études la normalization optimise les performances du modèle utilisé par la suite :

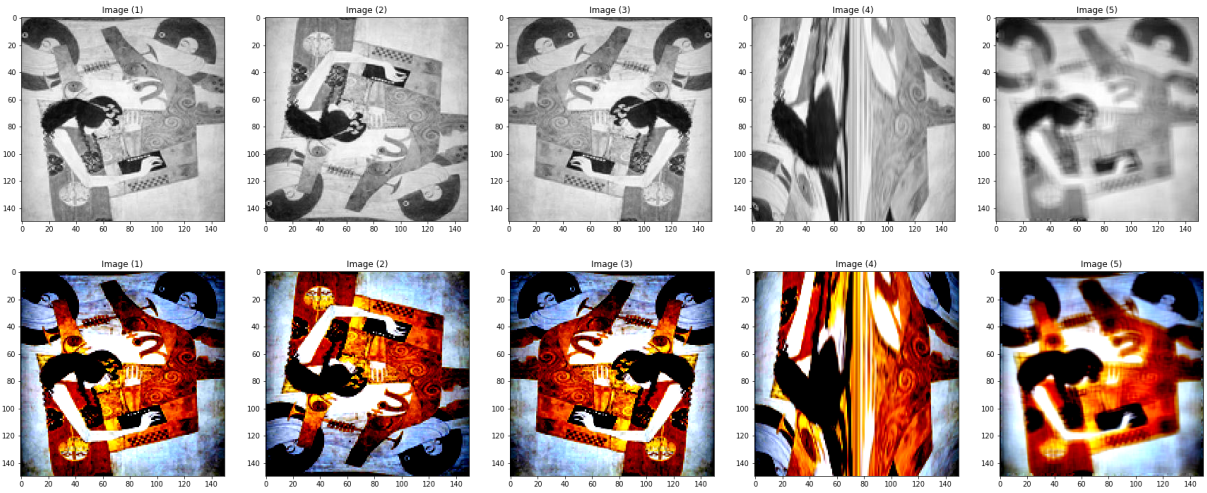
$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \qquad x' = \frac{x - \text{average}(x)}{\max(x) - \min(x)} \qquad x' = \frac{x - \bar{x}}{\sigma}$$

Min-Max Normalization **Mean Normalization** **Z-score Normalization**

Min-Max normalization est la méthode la plus simple et consiste à redimensionner la gamme des caractéristiques pour mettre à l'échelle la gamme en [0, 1] ou [-1, 1]. **Mean normalization** est une variante de la première normalization. Enfin **Z-score normalization** est la normalization la plus utilisée en machine learning. Elle consiste à soustraire à chaque donnée la moyenne et à diviser par l'écart type. Cette méthode à l'avantage centré les données autour de 0, ce qui facilite grandement l'apprentissage du réseau de neurones. Grace à cette normalization, le modèle va être en mesure d'ajuster ses poids plus rapidement. Pour notre projet nous avons utilisé la normalization Z-score. Soit X un pixel d'une image I. On effectue la transformation suivante sur chacun de nos pixels :

pixel X ⟶ *[X – moyenne (I)] / écart type (I)* ⟶ *nouveau pixel X*

Vous pouvez voir ci-dessous un aperçu des images provenant de la même peinture après avoir appliqué tous les prétraitements :

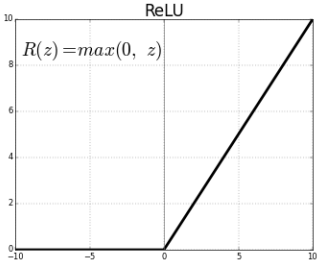


Visualisation des 10 images d'une même peinture après pré-traitements

Nous allons maintenant vous décrire comment nous avons implémenter nos premiers réseaux de neurones.

5.3 Premiers modéles

Les premiers modèles que nous avons implémenté sont des réseaux de neurones classiques. Leur structure est composée d'une couche entrante, d'une ou plusieurs couches cachées et d'une couche sortante. Ils peuvent avoir autant de couches cachées que l'on souhaite et chaque couche cachée peut avoir n'importe quel nombre de neurones. Néanmoins les réseaux de neurones ont rarement plus de 3 couches cachées et leur nombre de neurones ont tendances à diminuer au fur à mesure des couches. C'est pourquoi nous avons décidé d'implémenter 3 réseaux de neurones comportant respectivement 1 couche, 2 couches et 3 couches cachées avec un nombre de neurones à chaque fois qui diminue. Pour toutes les couches cachées nous avons choisi la fonction **ReLU.**



Fonction d'activation ReLU

La fonction ReLU est interprétée par la formule : f(x) = max(0, x). Si l'entrée est négative la sortie est 0 et si elle est négative alors la sortie est x. Elle fait partie des fonctions d'activations les plus efficaces pour l'apprentissage d'un réseau de neurones. L'avantage de cette fonction d'activation est qu'elle augmente considérablement la convergence du réseau et ne sature pas. En effet avec cette fonction le réseau va avoir tendance à minimiser son erreur, et donc à ajuster ses poids, plus rapidement.

Cependant la fonction ReLU présente un inconvénient. Si la valeur d'entrée est négative, la fonction renvoie 0. La conséquence est que le neurone ne verra pas ses poids ajustés lors de la rétropropagation et donc le réseau n'apprendra pas autant qu'il devrait le faire. Une solution pour remédier à ce problème aurait été d'utiliser la fonction **Leaky ReLU** à la place mais son implémentation dans **TensorFlow 2.0** n'est pas stable.
Tous nos modèles possèdent la même couche sortante. La couche possède 12 neurones, chacun de ces neurones correspond à un style artistique de notre dataset. Le modèle sélectionne comme réponse finale le neurone ayant la valeur la plus élevée. Comme nous l'avons évoqué plutôt ces valeurs sont comprises entre 0 et 1. Pour une classification les valeurs attendues ne sont pas des nombres mais des classes. On ne peut donc pas essayer d'approcher une valeur comme on a l'habitude le faire dans une régression classique avec un seul neurone dans la couche sortante. A la place on va mettre un neurone dans la couche sortante pour chaque style artistique. Chacun de ces neurones recevra une probabilité, on sélectionnera donc le neurone avec la probabilité la plus forte comme réponse finale.

Un problème récurrent observé avec les couches sortantes dans une classification est d'observer des probabilités en couche sortante qui ne sont pas cohérente. En effet leur valeur étant trop élevée, leur somme est souvent plus grande que 1, ce qui mathématiquement n'a aucun sens. La fonction d'activation **Softmax** permet de remédier à ce problème.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

Fonction d'activation Softmax

Grace a cette fonction d'activation on obtient une somme totale des probabilités égale a 1. On peut ainsi s'assurer que nos résultats soient cohérents. On se retrouve donc avec trois modèles qu'on peut voir ci-dessous :

```
model1 = tf.keras.Sequential([tf.keras.layers.Flatten(input_shape=(150,150)),#couche d'entrée
tf.keras.layers.Dense(256, activation='relu'),#couche cachée
tf.keras.layers.Dense(128, activation='elu'),#couche cachée
tf.keras.layers.Dense(12, activation='softmax')])#couche sortante

model1 = tf.keras.Sequential([tf.keras.layers.Flatten(input_shape=(150,150)),#couche d'entrée
tf.keras.layers.Dense(512, activation='relu'),#couche cachée
tf.keras.layers.Dense(256, activation='relu'),#couche cachée
tf.keras.layers.Dense(128, activation='elu'),#couche cachée
tf.keras.layers.Dense(12, activation='softmax')])#couche sortante

model1 = tf.keras.Sequential([tf.keras.layers.Flatten(input_shape=(150,150)),#couche d'entrée
tf.keras.layers.Dense(784, activation='relu'),#couche cachée
tf.keras.layers.Dense(512, activation='relu'),#couche cachée
tf.keras.layers.Dense(256, activation='relu'),#couche cachée
tf.keras.layers.Dense(128, activation='elu'),#couche cachée
tf.keras.layers.Dense(12, activation='softmax')])#couche sortante
```

Architecture de nos 3 modèles

Pour ces trois modèles nous avons choisis les mêmes paramètres d'apprentissage. L'algorithme utilisé pour réajuster les poids se nomme l'optimizer. Il en existe beaucoup, le plus connu se nomme **SGD** (Stochastique Gradient Descente). Pour notre projet nous avons utiliser l'algorithme **ADAM** qui est très efficace pour des problèmes de classification. Nous avons du ensuite définir l'erreur qui serait calculer à chaque fois entre la réponse obtenue et la réponse attendue. Il existe deux types d'erreur pour une classification :

- **Categorical crossentropy** (utilisée pour des classes non exclusives)

$$-\frac{1}{N}\sum_{s\in S}\sum_{c\in C}1_{s\in c}logp(s\in c)$$

- **Sparse categorical crossentropy** (utilisée pour des classes exclusives)

$$-logp(s\in c)$$

Dans notre cas nous sommes partis de l'hypothèse que chaque peinture possédait un seul style artistique. Nous avons donc choisi Sparse categorical crossentropy comme type d'erreur. Le dernier paramètre d'apprentissage qu'on nomme **metrics** concerne l'évaluation du modèle. A chaque foin d'époque, le modèle effectue une évaluation sur toutes les réponses qu'il a donné. Comme nous nous trouvons dans un problème de classification, il nous a paru

évident de choisir **l'accuracy**. Nous ensuite pu entrainer chacun des 3 modèles sur 15 époques. On évalue chacun des modèles sur le jeu de test et on obtient les résultats suivants :

	nb hidden layers	nb weights	test loss	test acc
Model 1	2.0	5794700.0	11.658021	0.131667
Model 2	3.0	11686284.0	3.482978	0.182500
Model 3	4.0	18208476.0	2.374448	0.222500

Evaluation de nos 3 modèles

A chaque fois on obtient une accuracy très faible. On remarque que celle-ci augmente légèrement si on rajoute des couches cachées. Il faudrait donc tester des modèles avec un grand nombre de couches cachées et espérer avoir une accuracy importante. Cette approche n'est pas viable puisqu'elle nécessiterait énormément de temps de calcul. Nous nous sommes donc tournes vers les réseaux de neurones convolutifs. Les réseaux de neurones convolutifs possèdent des couches de convolutions qui viennent s'interfacer avant la couche entrante. Les couches de convolutions vont appliquer toute sorte de filtre sur les images. Chaque couche possède un certain nombre de filtre, l'image de base sera dupliquée autant de fois qu'il y aura de filtre. Ainsi si on envoie une image dans une couche de convolution qui possède 20 filtres, il en ressortira 20 images. Voici l'architecture de notre réseau de neurones convolutif :

```
model2 = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32,(3,3),activation='relu',input_shape=(150,150,3)),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(64,(3,3),activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(12, activation='softmax')
])
```

Réseau de neurones convolutif

Encore une fois on choisit les mêmes paramètres d'apprentissage et on entraine le modèle sur 15 époques. Lors de l'évaluation sur le jeu de test on obtient une accuracy d'environ 0.15. On se retrouve avec le même problème que précédemment. Il faudrait ajouter un grand nombre de couches de convolutions et tester toutes combinaisons possibles pour espérer tomber sur la bonne. Cette approche nécessite également trop de temps de calcul. Pour résoudre ce problème nous nous sommes donc tournes vers le **transfer learning.**

5.4 Transfert d'apprentissage (Transfer Learning)

Après avoir développé de zéro des réseaux de neurones classiques et convolutifs (que nous avons entrainé sur notre dataset) nous avons introduit le **transfert d'apprentissage** afin d'améliorer la précision de nos modèles.

Le transfert d'apprentissage, c'est profiter d'un apprentissage acquis précédemment, pour, par analogie, résoudre un problème similaire mais différent. Par exemple, il est plus facile d'apprendre à piloter une moto si on sait déjà faire du vélo. Le transfert d'apprentissage des réseaux de neurones s'appuie sur le même principe. Si on a entraîné un réseau de neurones à différencier des chiens, des chats, à partir de photos, alors on pourra s'appuyer sur ce réseau pour deviner à quelle race appartient un chien. Et même encore mieux, on pourra utiliser notre réseau pour catégoriser des grenouilles.

Ceci est possible parce les réseaux de neurones sont empilés en couches, chacune apprenant de la précédente. C'est ainsi qu'un CNN (réseau de neurones par convolution) aura ses premières couches spécialisées dans la reconnaissance de formes simples (lignes horizontales, lignes verticales, diagonales, …), ses couches suivantes dédiées à la reconnaissance de formes un peu plus complexes (cercle, carré, triangle, …), ses couches suivantes orientées vers, par exemple la reconnaissance de visages, la reconnaissance de parties du corps et les dernières couches se consacreront à ce qui fait l'objet de l'apprentissage de ce réseau (par exemple chiens ou chats).

Au cours de la phase d'apprentissage, le réseau de de neurones modifie ses poids. Les poids (qui sont des nombres) et l'architecture du réseau suffisent à le caractériser (hormis quelques paramètres non décrits ici). Il est donc très facile de bénéficier d'un réseau de neurones existant, sans avoir à recalculer ce qui lui a permis d'atteindre sa configuration optimale, calculée pour le jeu de données et le problème pour lequel il a été conçu. Dans notre cas, nous avons utilisé **ImageNet** qui est une base de données d'images, librement accessibles sur Internet. Cette base de données contient 14 millions d'images, réparties en 1000 catégories.

L'objectif alors est utilisé les modèles qui sont près entrainés et bénéficier de leurs précisions.

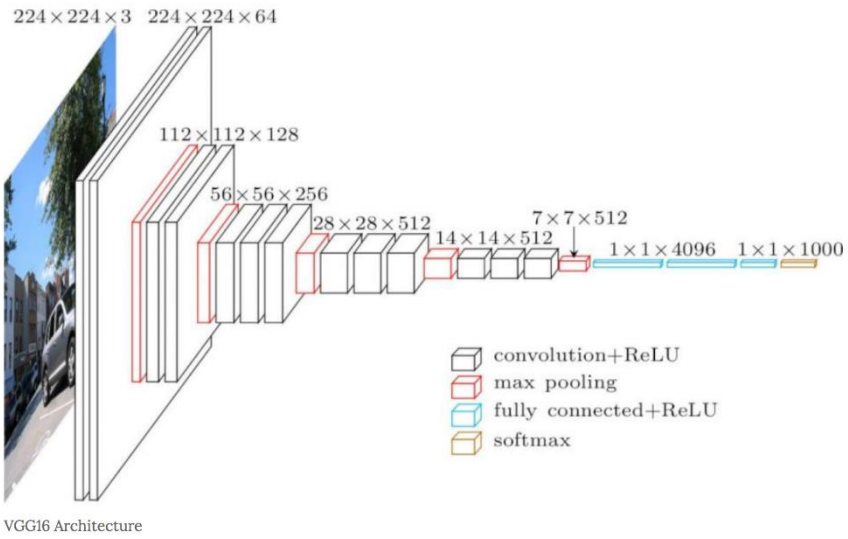
Ce qu'nous avons fait c'est construire un modèle qui est une concaténation du modèle pré entrainé avec notre modèle que nous avons entrainé sur nos données. Afin d'empêcher la mise à jour des poids sur les couches du modèle pré entrainé, on a gelé les couches de convolution. Pour le modèle qu'on a ajouté, il s'agit de trois couches une **couche entrante, une couche cachée et une couche sortante.**

```
#construction du model
model3 = [tf.keras.layers.Flatten(),
          tf.keras.layers.Dense(128, activation='relu'),
          tf.keras.layers.Dense(12, activation='softmax')]

#Jointure des deux modeles
model_using_pre_trained_one = tf.keras.Sequential( imagenet_model.layers + model3 )
```

Jointure des deux modèles (modèle pré-entrainé + RNN)

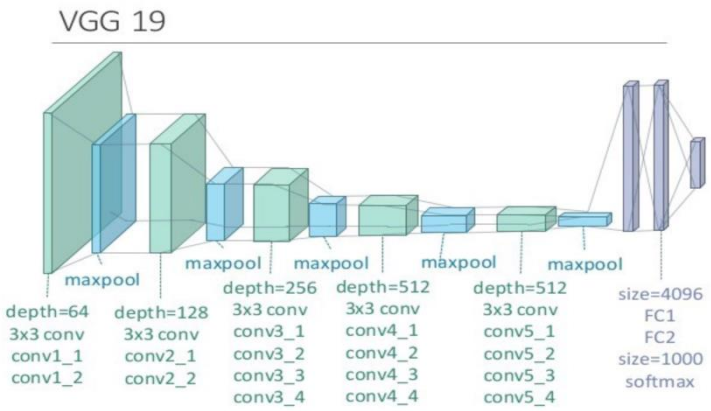
Nous avons utilisé deux modèles VGG16 et VGG19. Le VGG16 est un réseau de neurones convolutif proposé par K. Simonyan et A. Zisserman de l'Université d'Oxford. Le modèle atteint une précision de 92,7 % dans le top 5 des tests d'ImageNet, qui est un ensemble de données de plus de 14 millions d'images appartenant à 1000 classes. Il s'agit d'un des fameux modèles soumis à l'ILSVRC-2014. Il apporte l'amélioration par rapport à AlexNet. Le VGG16 a été formé pendant des semaines et utilisait les GPU NVIDIA Titan Black.



VGG16 Architecture

Architecture du réseau de neurones VGG-16

Le VGG-19 est un réseau neuronal convolutif formé par le Visual Geometry Group du département des sciences de l'ingénieur de l'Université d'Oxford. Le nombre 19 représente le nombre de couches avec des poids entraînables. 16 couches de convolution et 3 couches entièrement connectées.



Architecture du réseau de neurones VGG-19

Maintenant, nous allons maintenant présenter les résultats de nos modèles en utilisant le transfert d'apprentissage. Nous avons évalué nos deux modèles hybrides **(VGG-16 + RNN classique et VGG-19 + RNN classique)** sur 7 époques. Pour les deux modèles nous avons pris 25% des données (1200 images) pour l'évaluation et le reste (3600 images) pour l'apprentissage. Sur les figures ci-dessous on peut voir les 7 époques d'apprentissage avec à chaque fois l'erreur (loss) qui diminue et l'accuracy qui augmente.

```
Train on 3600 samples
Epoch 1/7
3600/3600 [==============================] - 512s 142ms/sample - loss: 1.9739 - accuracy: 0.3275
Epoch 2/7
3600/3600 [==============================] - 499s 138ms/sample - loss: 1.3751 - accuracy: 0.5372
Epoch 3/7
3600/3600 [==============================] - 521s 145ms/sample - loss: 1.0686 - accuracy: 0.6436
Epoch 4/7
3600/3600 [==============================] - 526s 146ms/sample - loss: 0.8483 - accuracy: 0.7128
Epoch 5/7
3600/3600 [==============================] - 520s 144ms/sample - loss: 0.6759 - accuracy: 0.7731
Epoch 6/7
3600/3600 [==============================] - 525s 146ms/sample - loss: 0.5623 - accuracy: 0.8097
Epoch 7/7
3600/3600 [==============================] - 528s 147ms/sample - loss: 0.4539 - accuracy: 0.8511
1200/1 - 152s - loss: 2.2610 - accuracy: 0.6208

Test accuracy: 0.62083334
```

Apprentissage et évaluation du réseau de neurones VGG-16

```
Train on 3600 samples
Epoch 1/7
3600/3600 [==============================] - 663s 184ms/sample - loss: 1.9802 - accuracy: 0.3231
Epoch 2/7
3600/3600 [==============================] - 659s 183ms/sample - loss: 1.4545 - accuracy: 0.5111
Epoch 3/7
3600/3600 [==============================] - 662s 184ms/sample - loss: 1.1657 - accuracy: 0.6064
Epoch 4/7
3600/3600 [==============================] - 659s 183ms/sample - loss: 0.9407 - accuracy: 0.6861
Epoch 5/7
3600/3600 [==============================] - 651s 181ms/sample - loss: 0.7947 - accuracy: 0.7253
Epoch 6/7
3600/3600 [==============================] - 623s 173ms/sample - loss: 0.6389 - accuracy: 0.7822
Epoch 7/7
3600/3600 [==============================] - 623s 173ms/sample - loss: 0.5433 - accuracy: 0.8069
1200/1 - 177s - loss: 1.4977 - accuracy: 0.6550

Test accuracy: 0.655

Test loss: 1.2531030782063801
```

Apprentissage et évaluation du réseau de neurones VGG-19

Avec VGG-16 on obtient une accuracy de 0.62 tandis qu'avec VGG-19 on obtient une accuracy de 0.65. Cela signifie que sur **65 %** des images notre modèle hybride VGG-19 a prédit le bon style artistique. Obtient donc une meilleure précision avec le modèle hybride (VGG-19 + RNN classique) par rapport à tous les modèles précédents. Même si l'accuracy reste encore relativement faible, on s'aperçoit qu'elle fortement augmenter entre la 6éme et la 7éme époque. Si nous disposions de ressources plus importantes on pourrait donc entrainer notre modèle hybride VGG-19 sur plus d'époques et obtenir une précision plus élevée.

6. Retour d'expériences

Les majeures difficultés que nous avons trouvé dans ce projet se résume en trois points :

- La difficulté de trouver un dataset des peintures. En effet, le dataset qu'on a finalement trouvé correspond à un dataset qui permet de prévoir l'artiste qui a fait une peinture et non pas le style de peinture d'où nous avons dû changer la réparation des images et nous avons fait le ménage de quelques images afin de construire notre propre dataset.

- Chargement des données n'était pas aussi évident. Au début, nous avons mis les données en local mais le chargement des données n'était pas rapide. Donc nous avons partagés tous nos images sur Google Drive et nous les avons importées dans Colab. Ceci était beaucoup plus rapide par rapport à la première méthode.

- Entrainement des modèles. En effet, l'étape la plus difficile dans ce projet à été d'entraîner le modèle. Cette étape consomme beaucoup de ressources (Mémoire surtout) et même en utilisant Colab, nous étions limités à 25GB de mémoire Ram. Ce n'était pas suffisant si jamais nous voulions augmenter le nombre des couches du modèle par exemple.

Au niveau des points d'amélioration, nous pourrons ajouter des prétraitements sur les images :

- **Boostrap**, technique qui consiste à faire un tirage au hasard avec remise des images du dataset jusqu'à obtenir un nouveau dataset de taille équivalente.

- **Enlever les données exotiques**, donc les peintures qui ne ressemble à aucune peinture du même style

- **Appliquer d'autres filtres** sur les images pour augmenter le dataset

Nous pouvons aussi utiliser d'autres modèles pré entraînés comme **Alexnet** ou **ResNet50** qui sont des modèles encore plus complexes mais plus adaptés à notre problème.