Utilisation de l'intelligence artificielle pour la génération de portraits médiévaux - Réalisation d'un classifieur d'attributs

Abstract

Dans le cadre d'un projet ayant pour objectif la modification de l'émotion de portraits datant de la Renaissance, ce rapport de stage porte sur la réalisation d'un classifieur d'émotion. Il reprend le fonctionnement d'un réseau de neurones, du neurone artificiel aux hyper-paramètres, puis il détaille la modification d'un algorithme de détection de visages et la réalisation d'un classifieur d'attributs grâce à un réseau de neurones en apprentissage supervisé. Les résultats indiquent que le classifieur obtient 80% de bonnes classifications sur des images qu'il n'a jamais vu. Ces derniers sont prometteurs mais de futurs résultats devront tenir compte d'images n'appartenant pas à la librairie d'entraînement et le passage du classifieur d'attributs au classifieur d'émotion devrait dégrader la précision actuelle.

Keywords: deep learning, emotion recognition, réseau de neurones, apprentissage supervisé, classifieur

Claire Robin

stage de Juin 2019 au Laboratoire Informatique et Système (LIS)

Licence 2 - Mathématiques, Physique, Chimie, Informatique -

Encadrants de stage : **Dr.Remi Eyraud, Dr.Stephane Ayache et Dr.Cécile Caponni**

Remerciments

Je remercie le Dr. Remi Eyraud, le Dr. Stephane Ayache et la Dr.Cécile Caponni pour m'avoir accueillie durant ce stage, pour leur disponibilité, pour leurs réponses à mes nombreuses questions, pour leurs conseils et pour toutes leurs explications qui m'ont permis d'appréhender l'apprentissage automatique. Je remercie aussi l'ensemble des stagiaires et en particulier mes co-équipiers de projet pour leur esprit d'équipe et pour leur aide. Enfin, je remercie l'ensemble de l'équipe QARMA du LIS avec qui ce fut un plaisir de réaliser ce stage passionnant et enrichissant. Il m'a permis de découvrir l'apprentissage automatique et les réseaux de neurones, de me familiariser avec de nouveaux outils mais aussi de mieux appréhender le fonctionnement de la recherche et du développement informatique.

Je remercie aussi Amidex et le responsable des stages, le Dr. Jean-Marc Debierre, pour m'avoir permis de réaliser ce stage.

Table des matières

Introduction

Les réseaux de neurones sont actuellement au centre de la recherche en intelligence artificielle, les résultats qu'ils obtiennent depuis 2012 avec la victoire du concours de classification d'images *ImageNet* font régulièrement la une des journaux, signe que les possibilités ouvertes par ces derniers provoquent une fascination certaine. Les récentes évolutions des réseaux de neurones, la simplification de leur codage, permet désormais leur utilisation dans des oeuvres artistiques ; le premier tableau peint par une intelligence artificielle *Portrait d'Edmond de Belamy* s'est ainsi vendu à plus de 430 000 dollars fin 2018.

L'équipe QARMA du laboratoire d'informatique et système (le LIS) a été sollicité pour aider un artiste de l'école des Beaux-arts à réaliser une installation. Il souhaite créer une oeuvre où le spectateur et un portrait de l'époque de la Renaissance interagissent. Pour cela on cherche à créer un programme qui modifie l'expression des portraits des tableaux en fonction de l'émotion exprimée par le spectateur. Ce projet conséquent à été réalisé avec trois autres stagiaires, Bastien Gastaldi, Hyppolite Debernardi et Hamed Benezah et avec l'aide du Dr.Cécile Caponni, du Dr.Stephane Ayache et du Dr.Remi Eyraud.

Dans ce projet je me suis occupée de la capture du visage à partir d'une photo ou d'une caméra puis j'ai travaillé sur la détection de l'émotion d'un visage en réalisant un classifieur d'attributs. Pour réaliser ces tâches j'ai passé une première partie de mon stage à comprendre le fonctionnement des réseaux de neurones, puis j'ai programmé un premier réseau avant d'avoir les compétences suffisantes pour réaliser le classifieur dont j'explique le fonctionnement dans la dernière partie.

1. Les différentes étapes du projet

Le projet a été partagé en un ensemble d'étapes et de tâches, j'avais en charge la détection de l'émotion du spectateur. Pour cela j'ai du réaliser un classifieur d'attributs grâce à un réseau de neurones dont j'ai dû apprendre le fonctionnement. En parallèle, j'ai réalisé l'identification des visages et leur enregistrement.

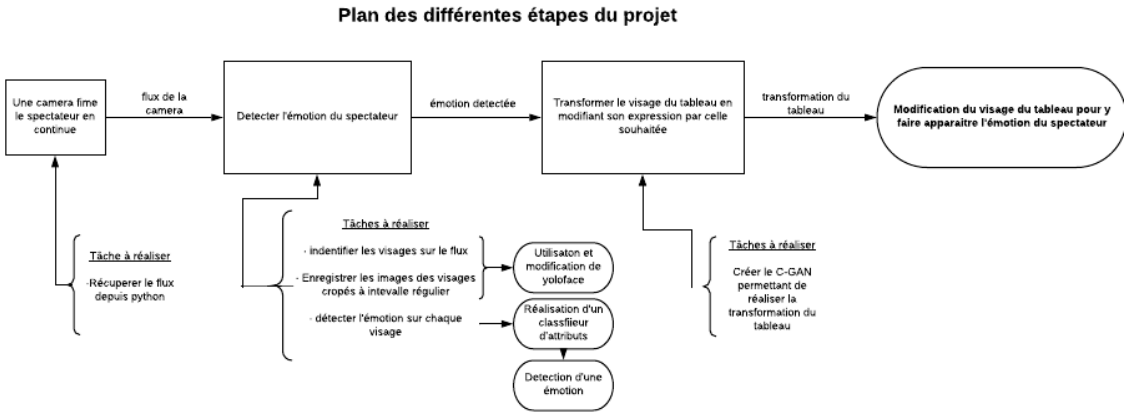

Figure 1: schéma des différentes étapes du projet

2. les réseaux de neurones profonds

Les réseaux de neurones sont composés d'un ensemble de neurones artificiels dont chacun reproduit le fonctionnement d'un neurone biologique.

2.1. neurone artificiel

Un neurone artificiel est un élément qui va faire la somme pondérée de l'ensemble des éléments qu'il a en entrée, auquel il applique une fonction d'activation. cette fonction non-linéaire et dérivable est l'équivalent du "potentiel d'activation" d'un neurone biologique, qui donne en sortie une réponse uniquement si un seuil de stimulation est dépassé.

Figure 1 – Neurone artificiel (McCullogh et Pitts, 1943)

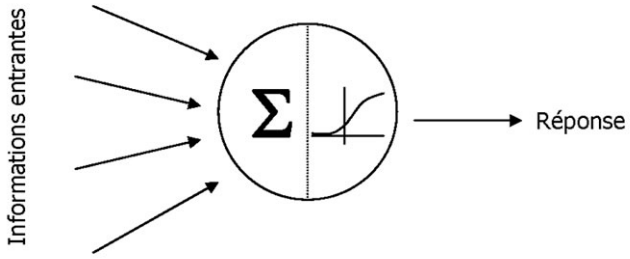

FIGURE 2: neurone artificiel

[1]

Il existe de nombreuses fonctions d'activations, on va s'attarder sur les plus utilisées c'est-à-dire la fonction ReLU, la fonction sigmoïde et la fonction softmax.

2.1.1. la fonction ReLU

La fonction ReLU pour *Rectified Linear Units* est définie par $\forall x$ f(x) = max(0, x). Elle rend le neurone inactif lorsque la somme pondérée est négative sinon le neurone transmet x, la valeur de la somme. Elle accélère la convergence et est surtout utiliser pour les couches denses qui ne sont ni en entrée, ni en sortie du réseau.

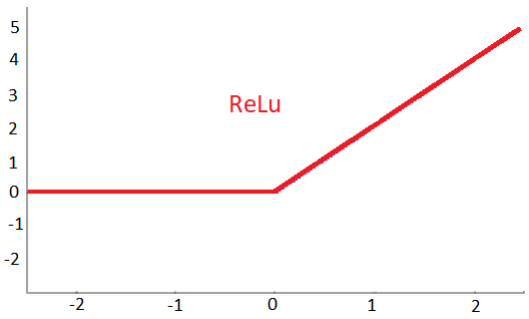

FIGURE 3: la fonction ReLU

[2]

2.1.2. la fonction sigmoïd

La fonction sigmoid est bornée par 0 et 1, elle permet donc de traduire la somme pondérée en une probabilité, elle est surtout utilisée en fonction d'activation sur la couche de sortie lorsque le problème est binaire.

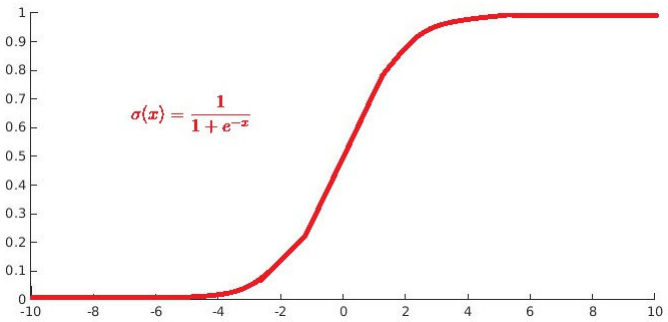

FIGURE 4: la fonction sigmoid

[2]

2.1.3. la fonction softmax

La fonction softmax est surtout utilisée dans le cadre de problème multi-classes (plusieurs classes sont possibles) mais non multi-labels (une unique classe peut-être associée à chaque échantillon). Elle est définie par :

$\forall x \in \mathbb{R}^N$, un vecteur de dimension N, alors $\forall i \in [1, ..., N]$

$$f(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{N} e^{z_k}}$$

Softmax donne ainsi pour un échantillon la probabilité qu'il appartienne à chaque classe et de tel sorte que la somme des probabilités soient égales à 1.

Tout le jeu de l'entraînement d'un neurone consiste à ajuster un coefficient associé à chaque information entrante afin de réduire une fonction d'erreur.
Un réseau de neurones est caractérisé par son architecture, il en existe plusieurs sortes, on va se concentrer sur celle utilisée durant ce stage, c'est-à-dire l'architecture multicouches.

2.2. le réseau connexionniste multicouches - Sequential

Un réseau est un ensemble de neurone. L'entrée est composée d'un nombre de neurones égal à la dimension du vecteur d'entrée, puis de couches de neurones cachés. Les couches de neurones cachés ont pour entrée la couche précédente (la première couche a donc en entrée la couche d'entrée) et sont directement reliées à la couche suivante et ainsi de suite jusqu'à la couche de sortie qui est composée d'un nombre de neurones égal au nombre de classes. Chaque connexion entre 2 neurones est dotée d'un poids w_{ij}.

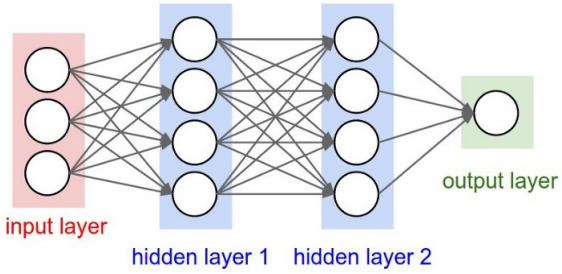

FIGURE 5: Un exemple de réseau séquentiel

[3]

Il existe différentes types de couches, celles utilisées durant ce stage sont :

2.2.1. la couche dense

Dans une couche dense, chaque neurone de la couche à en entrée l'ensemble des neurones de la couche précédente et à en sortie l'ensemble des neurones de la couche suivante, la figure 4 est composée de couches cachés denses.

2.2.2. la couche de convolution

Elle est utilisée dans les réseaux travaillant sur des images. La couche de convolution utilise le produit de convolution pour faire ressortir des features, c'est à dire des zones caractéristiques, des traits particuliers dans une image. Concrètement, une feature est une image de petite dimension qui va passer sur chaque zone de notre image. L'intégrale du produit de convolution correspond à l'aire commune aux deux fonction (l'image et la feature), lorsque l'intégrale est maximale, la feature est présente dans l'image et la couche de convolution la reconnaît.

2.2.3. la couche de pooling

Cette couche va réduire sur une image son nombre de pixels en faisant une moyenne (ou un maximum en fonction de la fonction de pooling utilisée -AveragePooling ou MaxPooling) des valeurs d'un ensemble de pixels sur une zone donnée et ainsi de suite sur chaque zone de l'image : en quelque sorte elle pixelise l'image. De cette façon la couche permet de réduire le nombre de paramètres et de calculs, de garder les informations essentielles et de rendre la feature retenue moins dépendante de son emplacement, évitant ainsi le sur-apprentissage. Elle est aussi souvent utilisé en sortie des couches de convolution où elle peut améliorer grandement l'efficacité du réseau en réduisant le nombre de paramètres et de calculs dans celui-ci.

2.3. Apprentissage de réseau de neurones

Une fois la structure créée, le réseaux doit être entraîné. Dans le cas de réseaux supervisés, on va lui fournir des paires -par exemple images / labels. Les images vont passer dans le réseau dont les poids sont instanciés aléatoirement à la première époque (en anglais epoch, une epoch correspond au passage de l'ensemble des données d'entrainement dans le réseau). A la sortie du

réseau un label est prédit par ce dernier, puis la fonction loss calcule l'erreur entre la prédiction et le véritable label de l'image. Tout le jeu de l'apprentissage consiste à minimiser l'erreur afin de prédire correctement une donnée. Pour cela, on rétropropage ensuite le gradient de l'erreur dans le réseau (voir la section 3.2 optimiseur pour plus de précision), c'est-à-dire que l'on va calculer les dérivées partielles pour chaque poids de la couche de sortie.

2.4. les réseaux de neurones convolutifs - Convolutional Neural Network

Les réseaux de neurones convolutifs sont les réseaux dédiés à l'analyse d'image. Ils sont composés de plusieurs séquences de couches convolutives puis d'une couche de pooling et se terminent par une couche dense.

Concrètement les couches ont tendance à se spécialiser, les premières couches reconnaissent des formes élémentaires (traits, courbes), les dernières identifient des formes plus complexes : une oreille, un oeil, chat, un visage etc. Le réseau contient aussi des séquences qui se spécialisent dans la texture, d'autres dans les couleurs. Les dernières couches denses permettent d'identifier à partir de l'ensemble de ces informations la classe de l'objet.

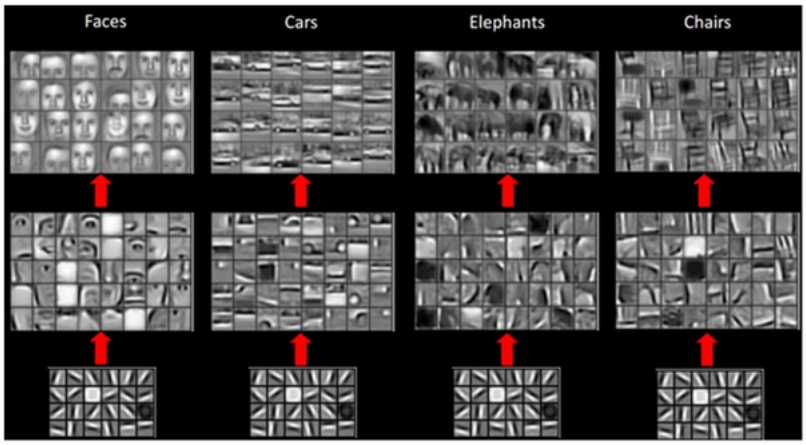

FIGURE 6: représentation des couches d'un CNN

[4]

Nous avons donc vu la structure et le fonctionnement d'un réseau de neurone, il a fallut ensuite en coder un. Le développement du deep learning ces dernières années à permis l'émergence de nombreuses interfaces simplifiant le codage de réseaux de neurones.

3. Programmation d'un premier réseau de neurones

La programmation d'algorithmes de machine learning se fait essentiellement sous python. Nous avons utilisé l'interface de programmation (API) de réseaux de neurones Keras qui utilise l'infrastructure logiciel TensorFlow de Google. J'ai commencé par travailler sur les TP de M. Rémi Eyraud et M. Stéphane Ayache afin de me familiariser avec le codage de réseaux des neurones. J'ai codé ensuite un premier réseau de neurones élémentaire avec la banque de données mnist, une banque de donnée constituée de chiffres écrits à la main.

```
import tensorflow as tf
from keras.utils import np_utils
import numpy as np
tfkl = tf.keras.layers

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data() #x_train est un jeu d'image, y_train est le
# jeu des classes associées
y_train =  np_utils.to_categorical(y_train, 10) #transforme le jeu de classe en un jeu où chaque classe est un vecteur

model = tf.keras.models.Sequential() #Créer un modele sequenciel
model.add(tfkl.Flatten(input_shape=(x_train.shape[1], x_train.shape[2]))) #couche d'entrée, elle applati les données en
# entrée sans affecter la taille du batch
model.add(tfkl.Dense(100, activation="relu"))
model.add(tfkl.Dense(100, activation="relu"))
model.add(tfkl.Dense(100, activation="relu"))
model.add(tfkl.Dense(100, activation="relu"))
model.add(tfkl.Dense(10, activation = "softmax")) # On utilise une softmax plutot qu'une sigmoïd car ???
model.compile(optimizer='adam', loss = 'categorical_crossentropy', metrics= ['accuracy'])
model.summary()
model.fit(x_train, y_train, batch_size=32, epochs=10, validation_split=0.2) #entrainement du réseau de neurone
```

FIGURE 7: code source d'un réseau élémentaire

Les premières lignes correspondent à la création d'un réseau de neurones séquentiel puis à l'ajout de différentes couches. Nous faisons bien attention que la première couche ait la même dimension que les données que l'on va lui fournir, de même que la dernière couche soit d'une dimension égale aux nombre de classes, auquel nous précisons la fonction d'activation softmax adaptée à un problème multi-couches comme vu précédemment. Nous avons ainsi construit la structure de notre réseaux, il s'agit de la première étape pour le faire fonctionner.

Sur cette structure nous allons réaliser un apprentissage supervisé, c'est-à-dire que nous allons forcer le réseau à converger vers un état précis en lui présentant lors de la phase d'apprentissage, pour chaque motif le résultat attendu. Tous les réseaux que j'ai réalisés durant ce stage l'ont été dans le cadre d'apprentissages supervisés.

Il faut ensuite compiler notre modèle, pour cela il faut lui préciser la fonction de perte (en anglais, loss) et l'optimiseur.

3.1. La fonction de perte

Lors de l'entraînement les images vont passer dans le réseau et en fonction des poids de ce dernier, le réseau va prédire une classe. L'écart entre la prédiction et la classe de l'image s'appelle la fonction de perte (loss en anglais), c'est donc une fonction qui dépend de l'ensemble des poids du réseau. Il existe de nombreuses fonctions qui mesurent cet écart. On va s'arrêter sur deux fonctions utilisée au cours de ce stage.

3.1.1. la binary crossentropy

La binary crossentropy est particulièrement bien adaptée aux problèmes binaires de type "est-ce un visage ou non ?" ainsi qu'aux problèmes multi-labels, ce qui correspond au classifieur d'attributs : un visage peut à la fois sourire et avoir la bouche ouverte.

3.1.2. la categorical crossentropy

La fonction catégorical crossentropy est plus dédiée aux problèmes multi-classes, elle va devoir attribuer une unique classe parmi n classes à notre image. C'est le cas du problème des chiffres écrits à la main.

3.2. l'optimiseur

Une fois la loss calculée, on doit modifier les poids du réseau afin de faire converger ce dernier et avoir la loss la plus petite possible à la fin de l'entraînement. Pour trouver le point où cette dernière est la plus petite possible on ne peut pas simplement trouver le minima global de la fonction en dérivant celle-ci selon chaque poids puis en regardant les points où elle s'annule, le nombre de poids à étudier est juste beaucoup trop grand. On va préférer utiliser des algorithmes de descente de gradient dont les optimiseurs font partie, ils vont donc permettre de trouver un minima. Pour tenter de trouver le minima global et non seulement un minima local de la fonction de perte, on va entraîner le réseau afin de maximiser les chances de tomber sur celui-ci.
Le learning rate est le pas d'apprentissage, c'est l'un des paramètres de l'optimiseur. Si il est trop bas, le réseau apprendra trop lentement ce qui amènera à une durée de convergence trop longue ; à l'inverse avoir un learning rate trop haut risque de nous faire sauter le minimum global, notre réseau sera peu performant et instable puisqu'il sautera régulièrement le minimum global, on parle alors d'exploding gradient. L'optimiseur le plus utilisé est Adam [7], paru en 2015, qui adapte le taux d'apprentissage en fonction de la pente du gradient, ce qui a pour effet de limiter le fait d'être bloquer dans un minimum local.

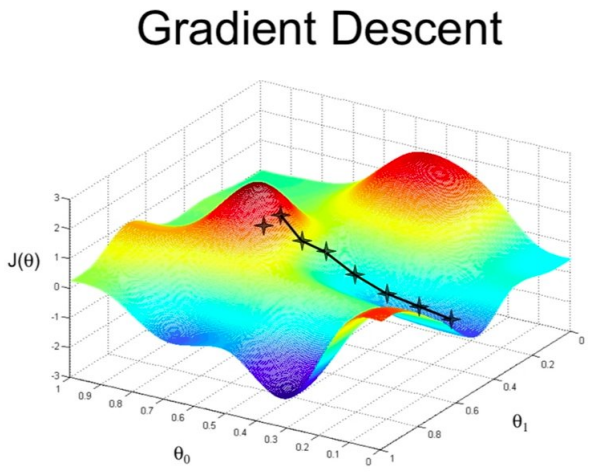

FIGURE 8: la loss en fonction de deux poids

[5]

On précise ensuite la métrique, c'est à dire la façon dont nous allons évaluer la performance de notre réseau.

3.3. la métrique

Il s'agit d'une fonction similaire à une fonction de perte, seulement elle va prendre en entrée des données qui ne sont pas passées dans le réseau et son résultat ne servira pas à l'entraînement du réseau. La métrique doit donc être adaptée au problème à traiter.

3.3.1. accuracy

L'accuracy est la métrique la plus utilisée. La binary_accuracy sera utilisée lors d'entraînement du réseau sur des problèmes binaires et on privilégiera la categorical_accuracy pour des problèmes multi-classes. Mathématiquement, elle correspond au taux de bonne classification.

3.3.2. precision

La métrique precision est plus adaptée aux problème multi-labels que l'accuracy, elle calcule une moyenne de précision par lot et mesure pour chaque objet le nombre de label associé pertinent. Elle correspond à la valeur prédictive positive en statistique, il s'agit du nombre d'attributs pertinents associés à l'image rapporté aux nombre d'attributs associés à l'image par le réseau.

$$precision = \frac{vrai\_positif}{vrai\_positif + faux\_positif}$$

Le réseau maintenant compilé, il ne reste plus qu'à l'entraîner avec la fonction fit. On lui précise un couple (x_train, y_train) qui correspond pour x aux données d'entraînements et pour y aux classes associées à ses données. Le batch_size correspond au nombre d'échantillon par mise à jour du gradient : il permet au réseau de faire une moyenne des erreurs des poids sur le batch et ainsi de modifier les poids sans trop coller à chaque donnée. Préciser le nombre d'epochs correspond au nombre de fois où l'ensemble des données d'entrée seront passées dans le réseau, un trop grand nombre amènera à un sur-apprentissage et un trop faible donnera un réseau trop peu précis. Enfin validation_split correspond au pourcentage de données que l'on va utiliser pour la validation et qui est donc sorti du lot d'apprentissage à chaque début d'epoch. Ces données ne seront pas utilisées pour l'entraînement mais uniquement pour le calcul de la perte et de la métrique.

On peut évaluer le modèle ensuite avec la fonction evaluate, qui prend en entrée un nouveau jeu de données qui ne passera à aucun moment, dans aucune epoch dans l'entraînement et permettra d'évaluer la perte et la métrique sur des données nouvelles. Un bon score à l'evaluate indique que le modèle n'a pas appris par coeur.
Pour savoir si le réseau à sur-appris, on peut étudier l'évolution de la fonction de perte et de la précision, lorsqu'il apprend, l'erreur des deux fonctions diminue, Lorsqu'il sur-apprend, les poids du réseaux apprennent par coeur sur les données d'entraînement, donc l'erreur sur les données d'entraînement continue de diminuer mais il ne sait plus prédire sur des données nouvelles et l'erreur sur ces dernières augmente.

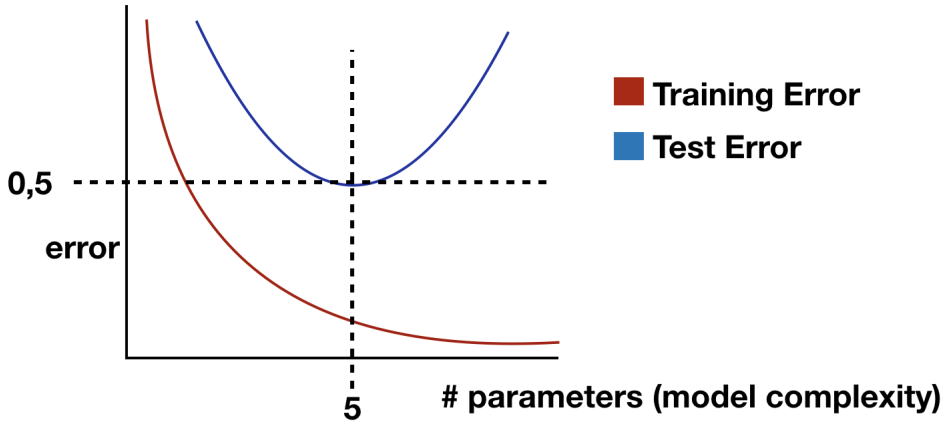

FIGURE 9: diagramme de l'erreur en fonction de l'apprentissage

[6]

Ainsi un grand nombre d'hyper-paramètres (perte, optimiseur, structure du réseau) influe sur la qualité de ce dernier, il s'agit de trouver pour chaque réseau les paramètres les plus adaptés pour rendre l'entraînement le plus efficace possible.

4. Réaliser un classifieur d'émotion

Entraîner un réseau de neurones convolutif à détecter des objets et des images demande une capacité de calcul et un temps de calcul monstrueux, heureusement un certain nombre de de réseaux pré-entraînés sont accessibles en open-source. Nous avons travaillé avec deux d'entre eux, YOLOv3 et VGG-19, implémentés sous Keras.

4.1. La détection du visage - YOLOv3

Nous avons utilisé YOLOv3 [8] pour la détection du visage. Cet algorithme utilise des Bounding Box, c'est à dire qu'il va chercher, pour chaque objet, la boite la plus petite qui contienne l'objet. Ce principe lui permet d'être plus rapide que l'ensemble des autres réseaux pré-entrainés avec un taux d'erreur relativement faible. Dans le cadre de notre projet, le visage d'un spectateur sera dans une image relativement peu complexe (peu d'objets dans l'image, le visage droit), on a en revanche besoin d'avoir une détection du visage la plus rapide possible puisqu'elle doit avoir lieu en temps réel, YOLOv3 est par conséquent le réseau le mieux adapté à notre problème.
Pour l'implémention, nous avons utilisé yoloface [9], une implémentation sous licence MIT accessible sur GitHub qui se concentre uniquement sur la détection de visages. L'installation du logiciel à demandé quelques heures de travail à la suite de problème de mise-à-jour de certains modules. J'ai ensuite modifié l'algorithme pour l'adapter à notre problème.
Lorsqu'il fonctionnait sur une webcam j'ai fait en sorte que le programme prennent des captures à intervalle régulier avec le module OpenCV, puis face des crops de chaque visage sur les captures. J'ai aussi modifié le programme pour que des crops aient lieu sur les visages lorsque le programme

prends une image en entrée. De cette façon, on normalise les images sur lesquelles on fera fonctionner le réseau, ce qui devrait nous permettre d'augmenter significativement la prédiction du réseau sur de nouvelles images. L'algorithme fonctionne aussi bien sur des photos que sur des portraits de la Renaissance.

FIGURE 10: utilisation de yoloface

4.2. la classification d'attributs - VGG-19

Il n'existe pas de base de données dédiée aux émotions en libre accès suffisamment grande pour pouvoir entraîner un réseau. On a donc décidé de travailler avec la base de donnée CelebA [11], une base de donnée de 200 000 images annotées d'une vingtaine d'attributs de description physique (couleur de cheveux, calvitie, pommettes saillantes, bouches ouvertes...). On s'est réduit à un ensemble de 30 000 images où les visages sont alignés, plutôt de face et à 6 attributs pouvant être associés par la suite à une émotion : cernes, grandes lèvres, pommettes hautes, bouche légèrement ouverte, yeux plissés, souriant.

On cherche a entraîner un classifieur multi-label (c'est à dire qu'une image peut se voir associer à plusieurs classes – par exemple la classe cernes et la classe bouche ouverte).

Pour cela on va partir du réseau pré-entrainé VGG-19, réalisé par l'université d'Oxford en Avril 2015, il s'agit d'un réseau convolutif dédié à la détection d'objets dans les images [10], il a été entraîné sur la base d'ImageNet et sur 1000 classes différentes.

On réalise ensuite un transfer learning, c'est à dire que l'on ré-entraîne le réseau avec l'ensemble de ses poids pour qu'il apprennent de nouvelles choses. Il reconnaît donc de nombreuses features

que l'on n'a pas à lui ré-apprendre, mais on va lui enlever la toute dernière couche, celle dédiée à la classification, et la remplacer par quelques couches de neurones qui vont spécialiser le réseau sur notre problème.

4.2.1. réalisation du classifieur

Je suis d'abord partie du réseau représenté par la figure 11, dont le code est en annexe 4.2.3 :

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 256, 256, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 256, 256, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 256, 256, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 128, 128, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 128, 128, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 128, 128, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 64, 64, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 64, 64, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 64, 64, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 64, 64, 256)       590080
_________________________________________________________________
block3_conv4 (Conv2D)        (None, 64, 64, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 32, 32, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 32, 32, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 32, 32, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 32, 32, 512)       2359808
_________________________________________________________________
block4_conv4 (Conv2D)        (None, 32, 32, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 16, 16, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 16, 16, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 16, 16, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 16, 16, 512)       2359808
_________________________________________________________________
block5_conv4 (Conv2D)        (None, 16, 16, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 8, 8, 512)         0
_________________________________________________________________
global_average_pooling2d (Gl (None, 512)               0
_________________________________________________________________
dense (Dense)                (None, 1024)              525312
_________________________________________________________________
dense_1 (Dense)              (None, 512)               524800
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 3078
=================================================================
Total params: 21,077,574
Trainable params: 3,412,998
Non-trainable params: 17,664,576
_________________________________________________________________
```

FIGURE 11: les couches du modèle créé

Pour le réaliser nous avons remplacé la dernière couche de VGG-19 par une couche de pooling, 2 couches denses avec la fonction ReLu et la dernière couche dense composée de 6 neurones pour les 6 classes avec la fonction d'activation sigmoïde. Cette dernière est plus adaptée à une classi-

fication multi-label, puisqu'elle nous donnera pour chaque attribut une probabilité indépendante des autres attributs que ce dernier soit présent sur l'image contrairement à la fonction softmax qui est normalisée, donc qui nous donnera pour chaque attribut une probabilité tenant compte de la probabilités des autres attributs d'être sur l'image.

4.2.2. analyse des résultats

On ré-entraîne ensuite le réseau sur la base de donnée souhaitée, ici celle de celebA, mais on bloque l'entraînement des couches originelles et l'on va entraîner uniquement les dernières couches de tel sorte qu'elles se spécialisent sur la classification des 6 classes que nous lui avons indiquées.

Sur 10 epochs, il a rendu pour premier résultat une loss de 0.2416 et une précision de 0.8788. La loss étant le taux d'erreur, le fait qu'elle soit aussi basse est très bon signe , de plus avoir une précision de près de 90 % indique donc que le réseau associe très bien les bons attributs aux bonnes images.

Notre réseau réalise donc correctement son apprentissage, j'ai ensuite ajouté une évaluation sur des données qu'il n'avait jamais vu avant, afin de vérifier qu'il n'apprend pas par coeur. Le résultat obtenu est le suivant :

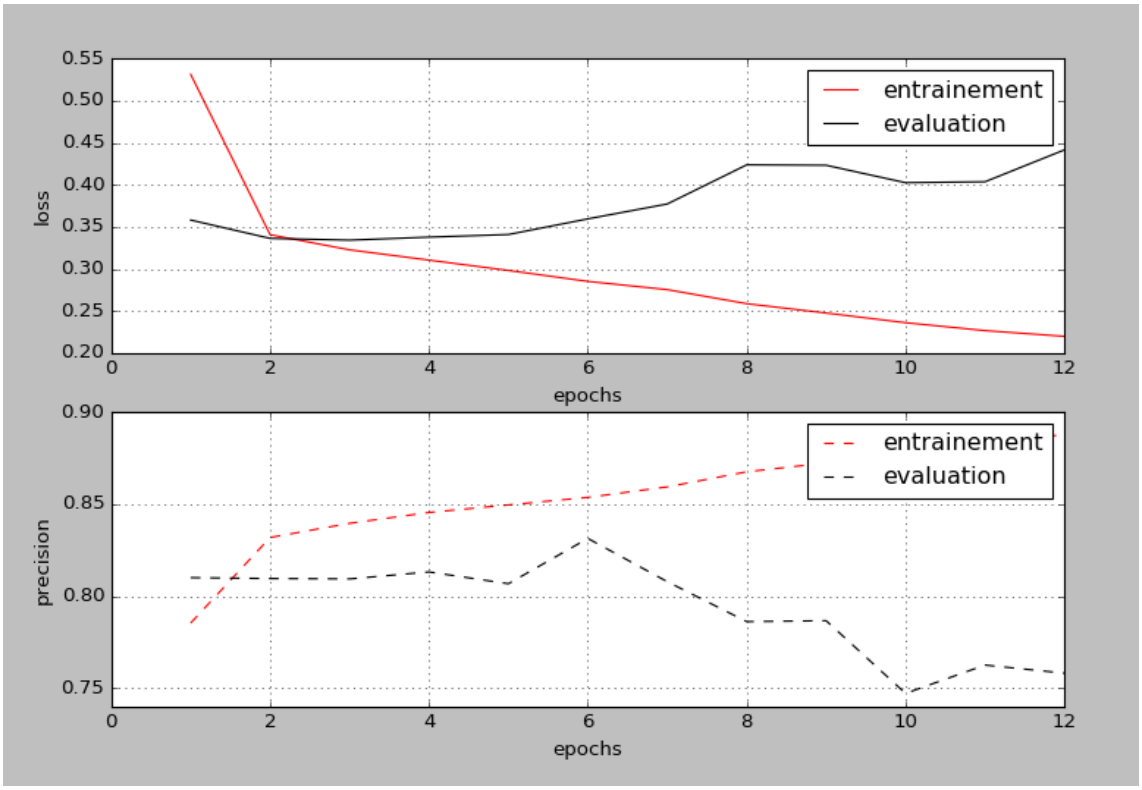

FIGURE 12: la loss et la précision en fonction du nombre d'epochs

On est clairement dans une situation de sur-apprentissage avec un réseau qui colle de plus en plus aux données d'apprentissage et perd en précision sur les données d'évaluation. Le réseau est

le plus performant au bout de 2 epochs, au delà il sur-apprend. C'est étonnant, parce que l'on voit que la précision sur de nouvelles données est maximale dès la première epoch, l'entraînement du réseau est donc presque inutile alors qu'il a 1 053 190 nouveaux paramètres (ceux des couches ajoutées) à apprendre. En même temps, pour chaque epoch, il a 164 lots de données (en anglais, mini batchs) donc 164 mise à jour du gradient.
Afin d'avoir un apprentissage plus précis, et coller moins au données, on peut tenter d'augmenter la taille des batchs afin d'avoir un nombre de mise à jour par epochs moins grand et de mieux généraliser l'apprentissage, mais les mini-batchs contiennent déjà 180 images chacuns (car il y a 30 000 images partagées en 164 lots) il est ainsi peu probable que le réseau s'améliore significativement.
A 2 epochs le réseau à tout de même obtenu une précision de 80 % et une loss de 0.35, il reste donc performant.
Je me suis ensuite penchée sur les hyper-paramètres pour tenter d'améliorer le réseau.

4.2.3. optimisation des hyper-parametres

-La résolution : de 256*256 à 128*128. Les résultats étaient nettement meilleurs pour 256*256 avec une loss de l'ordre de quelques dixièmes face à une loss dépassant 6, je pense que cela est dû au fait que VGG-19 a été entraîner sur des images de 224*224, donc les couches convolutives sont nettement plus adaptées à des images proches de cette résolution.

- le learning rate : de 0.001 à 0.01. Les meilleurs résultats ont été obtenus avec 0.001, le learning rate de 0.01, trop grand, doit sauter les minimas globaux ce qui amène à une forte loss ; en choisissant un learning rate de 0.01 on a un pas d'apprentissage plus petit qui permet d'atteindre les minimas globaux.

- la couche d'aplatissement : de GlobalAveragePooling2D à Flatten. La loss explose pour Flatten alors qu'elle converge vers quelques dixièmes pour GlobalAveragePooling2D. Cela est dû au fait que Flatten transforme l'ensemble des matrices de données de taille n*m en un unique vecteur de données de taille p*n*m où p est le nombre de matrices, tandis que GlobalAveragePooling2D fait une moyenne de l'ensemble des matrices de données, il créer donc un vecteur de taille n*m.
Dans le cadre de notre réseau, avec Flatten on se retrouve avec 36 443 142 paramètres à apprendre (sur 54 107 718 au total) et la couche dense qui suit Flatten fait 33 555 456 paramètres, tandis qu'avec GlobalAveragePooling2D on se retrouve avec 3 412 998 paramètres à apprendre (sur 21 077 574 au total) et la couche dense qui suit GlobalAveragePooling2D fait 525 312 paramètres.
Le réseau créé avec Flatten contient donc onze fois plus de paramètres à apprendre qu'avec GlobalAveragePooling, ce qui explique que celui-ci apprend nettement moins bien avec Flatten puisqu'il travaille toujours avec le même nombre d'exemples d'apprentissage mais contient beaucoup plus de paramètres.

Le meilleur modèle est donc celui initialement écrit, ce qui était plutôt prévisible puisque ce

dernier a été écrit à partir d'articles dédiés au transfer learning (voir la bibliographie) dont les auteurs avaient l'expérience pour identifier directement les hyper-paramètres optimaux du réseau.

Conclusion

Au cours de ce stage, je devais avancer le plus possible les tâches permettant de détecter l'émotion du spectateur. J'ai étudié et appris à coder un réseau de neurones, je me suis familiarisée avec le fonctionnement d'un neurone artificiel et les différents paramètres de ce dernier, puis j'ai étudié la structure d'un réseau et le fonctionnement des différentes couches, j'ai ensuite travaillé sur les hyper-paramètres des réseaux de neurones, notamment leur influence sur la performance de ce dernier. J'ai mis en application tout cela sur un premier réseau de neurone appliqué à la reconnaissance de chiffre écrits.

Pour détecter l'émotion sur un visage, j'ai d'abord modifié une implémentation déjà existante de YOLOv3 appliquée à la détection de visage afin d'obtenir des images normalisées à partir d'un flux vidéo ou d'une photo. J'ai ensuite réalisé puis entraîné un classifieur d'attributs ; l'analyse de ses résultats montre qu'il est le plus performant pour 2 epochs, au delà il sur-apprend. Enfin, j'ai cherché à vérifier le choix de certains hyper-parametres afin d'avoir le réseau le plus performant. Mon stage s'est clôturé sur cette tâche.

Il reste à réaliser à partir du classfieur d'attributs le classifeur d'émotion. L'une des pistes consiste à attribuer pour chaque émotion un ensemble d'attributs. Ainsi la présence de sourcil arquée et/ou de bouche ouverte pourra traduire l'étonnement, de même la présence yeux plissés et et souriant traduira la joie. Il faudra ensuite analyser les résultats pour savoir si les attributs de CelebA permettent de reconnaître une émotion. Il faudra aussi évaluer la performance du réseau de neurones sur de nouvelles images, prises à partir d'un caméra puis traitées par yoloface. Si les performances sont trop dégradées, on pourrait tenter de faire passer la librairie de CelebA dans yoloface puis de ré-entrainer le réseau, de cette façon il s'entrainera sur des images plus proches de celles qu'il devra traiter par la suite.

Références

[1] https://od-datamining.com/knwbase/les-reseaux-de-neurones-expliques-a-ma-fille/

[2] https://www.supinfo.com/articles/single/7923-deep-learning-fonctions-activation

[3] https://hackernoon.com/challenges-in-deep-learning-57bbf6e73bb
structure d'un réseau de neurones

[4] https://towardsdatascience.com/convolution-neural-network-decryption-e323fd18c33

[5] http://dridk.me/gradient_descendant.html

[6] http://coding-maniac.com/data-science/machine-learning-overfitting-and-how-avoid-it.html
Tensorflow
https://www.lebigdata.fr/tensorflow-definition-tout-savoir
choisir la loss et la fonction d'activation de la dernière couche :
https://www.dlology.com/blog/how-to-choose-last-layer-activation-and-loss-function/
l'optimizer :
https://www.charlesbordet.com/fr/gradient-descent/#comment-ca-marche- https://towardsdatascience.com/how-to-train-neural-network-faster-with-optimizers-d297730b3713
ADAM
https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[7] https://arxiv.org/pdf/1412.6980.pdf
détection de visage

[8] [April 2018] Joseph Redmon Ali Farhadi *YOLOv3 : An Incremental Improvement*, https://arxiv.org/pdf/1804.02767.pdf

[9] https://github.com/sthanhng/yoloface
VGG-19

[10] https://arxiv.org/pdf/1409.1556.pdf
celebA

[11] [December 2015], Liu, Ziwei and Luo, Ping and Wang, Xiaogang and Tang, Xiaoou, *Deep Learning Face Attributes in the Wild*,Proceedings of International Conference on Computer Vision (ICCV)

Transfer learning

https://towardsdatascience.com/keras-transfer-learning-for-beginners-6c9b8b7143e

https://riptutorial.com/keras/example/32608/transfer-learning-using-keras-and-vgg

https://medium.com/@14prakash/transfer-learning-using-keras-d804b2e04ef8

precision

https://www.tensorflow.org/api_docs/python/tf/keras/metrics/Precision https://fr.wikipedia.org/wiki/Pr%C3%A9cision_et_rappel

Annexe

Programme python du classifieur

```
from tensorflow.keras import applications
from tensorflow.keras.applications.vgg19 import VGG19, preprocess_input
from tensorflow.keras.preprocessing.image import load_img, img_to_array
import tensorflow as tf
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Flatten, Dense, Dropout, Conv2D, MaxPooling2D,
    GlobalAveragePooling2D, InputLayer
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
from dataset import *

#id_img id_pers id_img_orig Bags_Under_Eyes Big_Lips High_Cheekbones
    Mough_Slithgly_Open Narrow_Eyes Smilling

img_width, img_height = (256,256)

model = applications.VGG19(weights='imagenet', include_top=False,
                                input_shape=(img_width, img_height, 3))

x=model.output
x=GlobalAveragePooling2D()(x)
x=Dense(1024,activation='relu')(x) #we add dense layers so that the model can learn
     more complex functions and classify for better results.
x=Dense(512,activation='relu')(x) #dense layer 2preds=Dense(5,activation='softmax')
    (x) #final layer with softmax activation
preds=Dense(6,activation='sigmoid')(x) #final layer with sigmoid activation
new_model=Model(inputs=model.input,outputs=preds)

for layer in new_model.layers[:20]:
    layer.trainable=False
for layer in new_model.layers[20:]:
    layer.trainable=True
new_model.summary()
ROOT_PATH = '/data1/home/hippolyte.debernardi/data/celebAHQ/images/'
#'/data1/home/hippolyte.debernardi/renaissance/data/celebAHQ/npyfile/'


new_model.compile(optimizer= Adam(lr = 0.001),loss='binary_crossentropy',metrics=[
    tf.keras.metrics.Precision()])
#batch_size = 128
data = TfDataset("celebA-HQ-identities-attributes.csv2", directory=ROOT_PATH,
    desiredSize=[img_width, img_height]).makeClaireDataset(csv="celebA-HQ-
    identities-attributes.csv2", batch_size=128)
DATASET_SIZE = int(30000/128) + 1
train_size = int(0.7 * DATASET_SIZE)
val_size = int(0.15 * DATASET_SIZE)
```

```
test_size = int(0.15 * DATASET_SIZE)

train_dataset = data.take(train_size).cache()
test_dataset = data.skip(train_size)
val_dataset = test_dataset.take(val_size)
test_dataset = test_dataset.skip(test_size)

new_model.fit(train_dataset, validation_data=test_dataset,  epochs=100)
new_model.evaluate(val_dataset)

new_model.save('newmodel.h5')
```

Listing 1: le classifieur d'attributs