

Projet - Classification d'image par un perceptron multicouche

Pather Stevenson

dirigé par M. Djeraba Chaabane

L3 Informatique Groupe 5 - Semestre VI - 2022



Table des matières

1	Paramètres du projet	3
1.1	Choix du thème	3
1.2	Objectif du projet	3
1.3	Source des données	4
1.4	Langage utilisé	4
1.5	Documentation	5
1.6	Librairies utilisées	5
1.6.1	Scikit-learn	5
1.6.2	Imageio et Os	5
1.6.3	Tqdm	6
1.6.4	NumPy	6
1.6.5	Matplotlib	6
1.6.6	Random	6
2	Les réseaux de neurones artificiels	7
2.1	Le neurone	7
2.2	Le perceptron multicouche	9
2.3	Apprentissage	9
2.3.1	Supervisé	10
3	Documentation argumentée	11
3.1	Programme main.py	12
3.2	Phase de pre-processing	12
3.2.1	CategoriesLoader.py	12
3.2.1.1	Création	13
3.2.1.2	Lecture du fichier de classe	13
3.2.2	MatrixLoader.py	14
3.2.2.1	Création	14

	3.2.2.2	Génération du dictionnaire de données	15
3.2.3		FeatureLoader.py	16
	3.2.3.1	Principal Component Analysis	16
	3.2.3.2	Création	17
	3.2.3.3	Singular Value Decomposition	18
	3.2.3.4	Obtention des valeurs singulières	18
3.2.4		DataLoader.py	19
	3.2.4.1	Création	19
	3.2.4.2	Les dictionnaires du module	20
	3.2.4.3	Génération des dictionnaires	20
3.3		Phase d'apprentissage	21
	3.3.1	TrainingModel.py	21
	3.3.1.1	Prototype de la fonction d'entraînement . . .	21
	3.3.1.2	Linear Discriminant Analysis	22
	3.3.1.3	Etapes de la fonction d'entraînement	23
	3.3.2	Précision sans hyperparamètres	24
	3.3.3	HyperparameterTuning.py	25
	3.3.3.1	Création	26
	3.3.3.2	Lancement de la recherche par grille	26
	3.3.3.3	Etapes de la recherche par grille	26
	3.3.3.4	Résultats	27
3.4		Phase de test	28
	3.4.1	GenerateTest.py	28
	3.4.1.1	Création	28
	3.4.1.2	Génération des prédictions	28
	3.4.1.3	Résultats	29

Paramètres du projet

1.1 Choix du thème

Pour ce projet, j'ai fait le choix des expressions faciales car c'est un sujet très souvent abordé dans la littérature de la classification par réseaux de neurones. Ce qui m'a poussé à vouloir comprendre, appliquer et observer de moi-même le résultat que l'on pouvait en tirer. Mais également voir et comprendre les difficultés en fonction de la complexité des classes d'images dans le cas des expressions faciales qui sont plus complexes que d'autres thèmes.

1.2 Objectif du projet

L'objectif de ce projet est d'implémenter un programme qui va nous permettre de lire les données de nos images d'expressions faciales. Pour donner celles-ci à un perceptron multicouche qui effectuera une phase d'apprentissage de manière supervisé avec ces données.

Si la précision du modèle pour sa phase de d'entraînement et de validation ne sont pas satisfaisantes nous proposerons alors une optimisation de la précision de perceptrons multicouches pour différentes catégories données et différents nombres de couches cachées. En appliquant la méthode de grid-search concernant les hyperparamètres.

Et pour conclure nous utiliserons les perceptrons multicouches retenus en fonction de leurs précisions pour générer des exemples d'images où les modèles ont réussi à fournir la catégorie de l'image et sinon la catégorie supposée (et

fausse) par le modèle avec la catégorie correcte de l'image initialement enregistrée lors du pre-processing.

1.3 Source des données

Pour simplifier mon projet et notamment la phase de pre-processing, j'ai fais le choix d'un ensemble d'images d'expressions faciales qui a déjà normalisé la dimension des images et centré le visage. Ainsi que le niveau de gris des images (grayscale) qui est déjà appliqué. Le choix s'est donc porté sur le dataset suivant :

[The Facial Expression Recognition 2013 \(FER-2013\)](#)

On y retrouve notamment les sept catégories suivantes :

- Angry
- Disgust
- Fear
- Happy
- Sad
- Surprise
- Neutral

Pour limiter le temps d'exécution de notre projet lors de la présentation il sera extrait de façon arbitraire environ 1000 images par catégories.

1.4 Langage utilisé

Pour mener à bien le projet, nous allons utiliser le langage de programmation python dans sa version 3.8 et nous appliquerons le plus possible, les principes de la programmation orientée objet dans un but de propreté, facilité de compréhension du code et de sa maintenance.

1.5 Documentation

La documentation sera écrite pour sphinx et généré à l'aide de sphinx. Sphinx est le générateur libre de documentation officielle pour des projets en Python. C'est naturellement pourquoi il sera utilisé ici.

1.6 Librairies utilisées

les librairies utilisées dans ce projet peuvent être installer à l'aide de la commande make suivante à partir de la racine du projet :

```
$ make lib
```

Ce qui va exécuter le script bash **install_lib.sh** présent à la racine du projet.

1.6.1 Scikit-learn

La librairie principale que nous allons utiliser est scikit-learn. C'est une bibliothèque libre destinée à l'apprentissage automatique qui est notamment utilisée et développée dans le monde universitaire et de la recherche comme l'Inria.

Mon enseignant m'a notamment recommandé l'utilisation de cette librairie pour les raisons ci-dessus et pour me permettre de prendre en main celle-ci avec ce premier projet en apprentissage automatique.

Cette bibliothèque va notamment nous permettre la manipulation dans notre projet de MLPClassifier, PCA, LinearDiscriminantAnalysis et de certaines fonctions de la librairie.

1.6.2 Imageio et Os

Imageio est une bibliothèque python qui permettra la lecture des données de nos images. Plus particulièrement nous allons utiliser la fonction imread qui va nous permettre d'obtenir en retour d'un chemin d'image donné, un tableau numpy qui représente la matrice de données de l'image.

Le module `Os` nous servira ici simplement l'obtention des chemins des images dans nos répertoires pour pouvoir donner ceux-ci en argument de la fonction `imread`.

1.6.3 Tqdm

`Tqdm` est une bibliothèque qui va nous permettre de créer des barres de progression pour apporter un meilleur suivi du déroulement du programme lors du lancement de la méthode qui permettra l'obtention des features des images.

1.6.4 NumPy

La bibliothèque `NumPy` va nous permettre la manipulation de matrices, tableaux multidimensionnels ou de méthodes applicables à ceux-ci.

1.6.5 Matplotlib

Nous utiliserons la librairie `matplotlib` pour permettre la génération des images de contrôle à partir des matrices et labels (nom de catégorie prédit et réel) lors de la phase de test.

1.6.6 Random

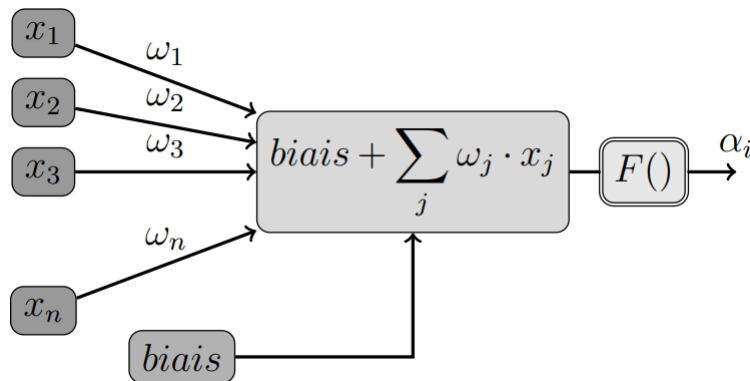
La bibliothèque `Random` sera utilisé uniquement dans la phase de test pour obtenir un index aléatoire dans la liste des images de la section de validation.

Les réseaux de neurones artificiels

les réseaux de neurones informatiques reprennent la modélisation des neurones biologiques. Ils imitent le fonctionnement de ceux-ci dans le but de résoudre des problématiques d'apprentissage machine. Ce qui s'avère être plus performant que l'utilisation des techniques de régressions pour des tâches d'apprentissage automatique.

2.1 Le neurone

Introduit par les travaux de Warren S. McCulloch et Walter Pitts en 1943, le neurone artificiel est la composante de base d'un réseau de neurones. Il définit un modèle mathématique simplifié d'un neurone biologique.

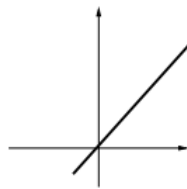


Comme on peut le voir sur Le neurone formel ci-dessus, celui-ci comprend une série d'entrées et d'une sortie. Les entrées x_j sont, de manière indépendante, pondérées (w_{ij} puis leur somme et une valeur de seuillage noté θ_i sont utilisées pour la valeur de sortie par le biais de la fonction d'activation noté F . Ainsi l'on peut déterminer cette sortie de la façon suivante :

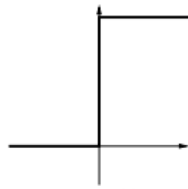
$$a_i = F\left(\sum_{j=1}^N w_{ij}x_j - \theta_i\right) \quad (2.1)$$

Le choix de la fonction d'activation doit être fait en fonction du domaine d'application mais aussi de la position du neurone dans le réseau. Le choix de celle-ci influe sur les résultats. Voici quelques exemples de fonctions d'activation :

- linéaire : fonction identité
- seuil : $1_{[0,+\infty[}(x)$
- sigmoïde : $\frac{1}{(1+e^x)}$
- ReLU : $\max(0, x)$ rectified linear unit
- softmax : $\frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}} \quad \forall k \in \{1 \dots K\}$
- radicale : $\sqrt{\frac{1}{2\pi e^{-x^2/2}}}$



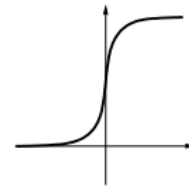
Linéaire



Seuil



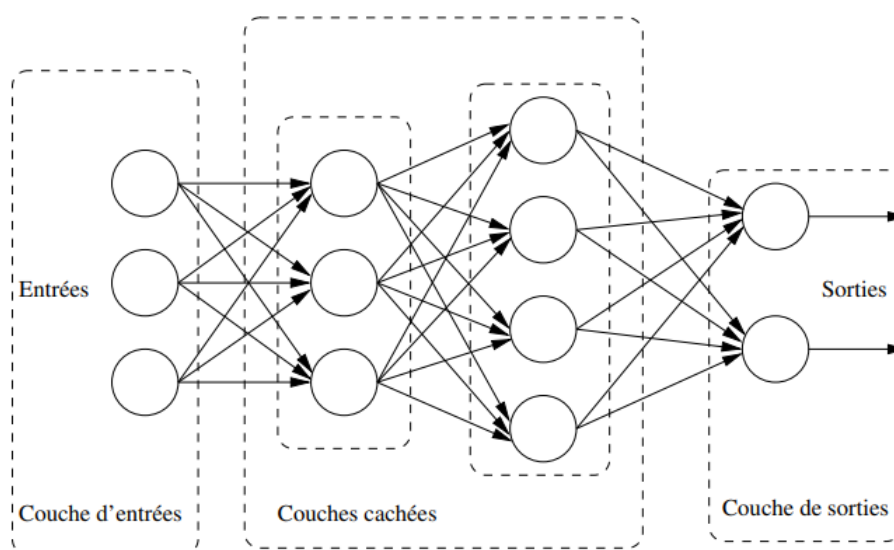
Gaussienne



Sinusoidale

2.2 Le perceptron multicouche

Notamment présenté par Yann Le Cun en 1985, le perceptron multicouche est un réseau à propagation directe (feed-forward). C'est-à-dire n'ayant pas de boucle dans leur connexion. Une telle topologie de réseau comprend une couche d'entrée, des couches cachées (éventuellement aucune) et d'une couche de sortie :



2.3 Apprentissage

Les algorithmes d'apprentissage peuvent être séparés en deux grandes familles : les non-supervisés et les supervisés. L'apprentissage non-supervisé est principalement utilisé pour la clusterisation qui consiste à regrouper un ensemble d'éléments hétérogènes sous forme de groupes homogènes ou liés par des caractéristiques communes. Et ainsi c'est la machine qui elle même fait les rapprochements en fonction de ces caractéristiques sans intervention externe. Ils sont donc utilisés pour des réseaux de neurones dont le résultat attendu n'est pas connu au préalable. Comme nous utiliserons pas dans notre projet ce type d'algorithme, ils ne seront pas présentés ici.

2.3.1 Supervisé

L'apprentissage supervisé consiste à entraîner un réseau de neurones pour qu'il reproduise un comportement déterminé. Ainsi on fournit des exemples pour que le réseau de neurones ajuste ses paramètres, donc ses poids de neurones pour permettre de diminuer l'écart entre le résultats que l'on obtient en sortie et celui attendu.

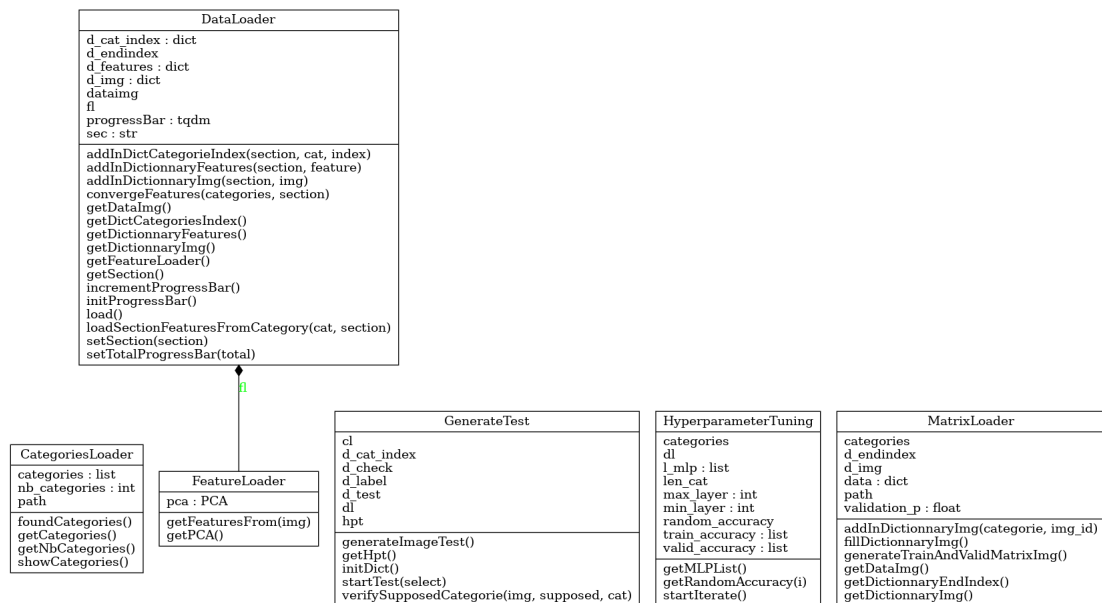
De cette manière la marge d'erreur diminue au fur et à mesure de la phase d'apprentissage supervisé dans le but d'être ainsi capable de généraliser son apprentissage à de nouveaux cas.

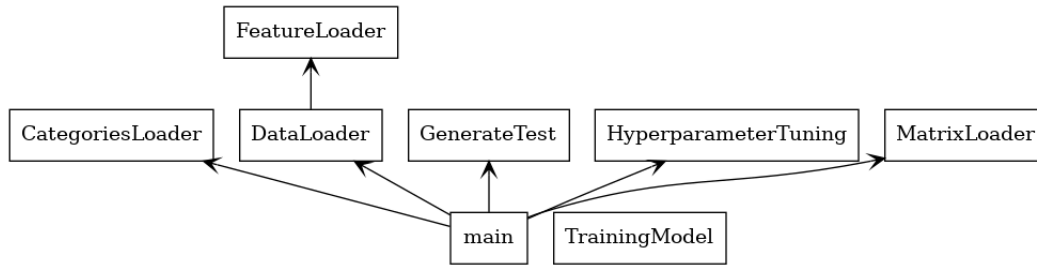
Documentation argumentée

Nous allons maintenant apporter des informations complémentaires aux codes pour venir enrichir celui-ci et justifier notre démarche.

Tous les fichiers permettant l'implémentation du projet sont dans le dossier **src/** du rendu. Nous prenons donc la liberté de ne pas préciser ce chemin si nous devons citer les fichiers de notre implémentation.

Voici le diagramme UML et packages du projet :





3.1 Programme main.py

Le programme main.py sera celui qui va exécuter dans l'ordre attendu les différentes phases ci-dessous à partir des modules qui les composent.

3.2 Phase de pre-processing

La phase de pre-processing consiste à la préparation des données pour la phase d'apprentissage. Dans notre cas ce sont donc les images qui sont réparties en fonction de la catégorie de l'expression faciale.

Il sera donc nécessaire d'effectuer les étapes suivantes :

- Obtenir le nom des catégories
- Récupérer les chemins des images à partir de la racine du projet
- Charger en mémoire les matrices des images par catégorie
- Prendre 25% de l'ensemble des images par catégories dans le but d'obtenir un ensemble d'images de validation. Les 75% restants par catégories représente donc l'ensemble d'entraînement.

Si notre jeu de données n'était pas déjà normalisé (grayscale, même dimension pour toutes les images et visage au centre de l'image), alors il aurait été également nécessaire d'effectuer ces étapes dans la phase de pre-processing.

3.2.1 CategoriesLoader.py

Le module CategoriesLoader va nous permettre la création d'un objet pour obtenir les catégories des images. Ce module considère que notre jeu de données respecte la convention où les noms de classes (catégories) sont

contenues dans un fichier de descriptions où y figure un nom de classe par ligne. Vous pouvez retrouver ce fichier dans notre projet : **data/descriptions/categories.txt**

3.2.1.1 Création

Pour la création de l'objet il est nécessaire de lui donner en paramètre du type *str* qui représente le chemin vers le fichier de descriptions contenant les noms de classes :

```
>>> cl = CategorifesLoader("data/descriptions/categories.txt")
```

3.2.1.2 Lecture du fichier de classe

La méthode *foundCategories* du module va permettre la lecture du fichier donné en paramètre lors de création :

```
>>> cl.foundCategories()  
foundCategories STARTING...
```

```
foundCategories DONE!
```

```
NB founded categories = 7
```

```
showCategories :
```

```
— angry  
— disgust  
— fear  
— happy  
— neutral  
— sad  
— surprise
```

Ainsi il nous sera possible grâce à cet objet d'obtenir directement la liste des catégories à l'aide de la méthode *getCategories*, ainsi que la taille de cette liste avec la méthode *getNbCategories*.

3.2.2 MatrixLoader.py

Ce module va nous permettre d'organiser de façon ordonné dans un dictionnaire le chemin des images de l'ensemble d'entraînement et de validation. Pour cela nous prenons 25% du nombre d'images en partant de la fin de la liste de l'ensemble des images. Ainsi que leur matrices sous forme de tableau NumPy grâce à la fonction *imageio.imread* qui pour un chemin d'image donné en argument retourne le tableau. Voici un exemple :

```
>>> imread("../data/train/angry/Training_3908.jpg")
Array([[163, 128, 114, ..., 139, 141, 134],
       [147, 114, 102, ..., 142, 138, 138],
       [112, 106,  92, ..., 140, 141, 134],
       ...,
       [139, 141, 136, ..., 154, 171, 191],
       [140, 133, 120, ..., 138, 146, 158],
       [136, 134, 113, ..., 146, 144, 144]], dtype=uint8)
```

On obtient donc une matrice où les valeurs représentent le niveau de gris de chaque pixel de l'image. Nous avons donc dans notre cas des tableaux de 48 lignes qui contiennent chacune 48 valeurs comme nous travaillons avec des images de dimensions 48×48 .

3.2.2.1 Création

La création de l'objet se fait de la façon suivante en donnant en argument le chemin du répertoire contenant les dossiers des catégories qui contiennent les images de leur catégories d'expressions faciales et la liste des catégories :

```
>>> ml = MatrixLoader("data/train", cl.getCategories())
```

3.2.2.2 Génération du dictionnaire de données

La méthode *generateTrainAndValidMatrixImg* va permettre la génération du dictionnaire qui contiendra le chemin des images ainsi que leur matrice en fonction de leur catégorie et de leur appartenance soit à l'ensemble d'entraînement ou bien de validation :

```
>>> ml.generateTrainAndValidMatrixImg()
```

Une trace d'exécution affichera l'entrée dans une catégorie ainsi que le dernier index des sections train et valid pour chaque catégories. Par exemple :

```
fillDictionnaryImg STARTING...
```

```
fillDictionnaryImg DONE!
```

```
generateTrainAndValidMatrixImg STARTING...
```

```
PORTION OF VALIDATION SET = 25.00%
```

```
Enter in category : angry
```

```
training_end index = 837
```

```
valid_end index = 931
```

```
...
```

Ainsi ce dictionnaire qui a pour clés le nom des catégories, qui contiennent chacunes un dictionnaire avec pour clés :

- '*train_file*' : tableau des chemins des images de l'ensemble d'entraînement
- '*valid_file*' : tableau des chemins des images de l'ensemble de validation
- '*train_img*' : tableau des matrices de l'ensemble d'entraînement
- '*valid_img*' : tableau des matrices de l'ensemble de validation

Obtention du dictionnaire généré : Il est possible désormais d'obtenir le dictionnaire complété comme cité ci-dessus à partir de la méthode *getDataImg* :

```
>>> ml.getDataImg()
```

3.2.3 FeatureLoader.py

Le module FeatureLoader va nous permettre la manipulation d'un objet Principal Component Analysis (*sklearn.decomposition.PCA*).

3.2.3.1 Principal Component Analysis

En apprentissage automatique cette méthode permet la réduction de la dimensionnalité qui améliore la performance des algorithmes car elle permet d'éliminer les variables corrélées qui ne contribuent à aucune décision du modèle. Le principe est de transformer des variables corrélées en nouvelles variables décorrélées en projetant les données dans le sens de la variance croissante. Les variables avec la variance maximale seront retenues comme les composants principaux.

De cette façon nous conservons le maximum d'informations possibles en utilisant le moins de variables possibles. Ce qui permet d'éviter un sur-apprentissage du modèle pour des images avec de grandes dimensions.

Dans notre cas nous avons des images qui sont déjà de petite taille (48×48). Il n'est donc pas nécessairement utile d'effectuer cette réduction. Mais dans le but d'approfondir notre démarche, nous allons tout de même effectuer une réduction en indiquant 32 composantes à notre PCA.

Voici un exemple de réduction dimensionnelle avec un PCA utilisant le solveur randomized-SVD présent dans la documentation du module *sklearn.decomposition* :

First centered Olivetti faces



igenfaces - PCA using randomized SVD - Train time 0.1



3.2.3.2 Création

Pour créer l'objet il est nécessaire de lui donner en argument le nombre de composantes que l'on souhaite que le PCA cherche à retenir pour faire ressortir les variations. Si aucun argument n'est donné 32 sera utilisé par défaut et donné en argument *n_components* du PCA qui sera créé dans le *FeatureLoader*. Ainsi la création s'effectue de la façon suivante :

```
>>> fl = FeatureLoader()
```

Dans la documentation de Scikit-learn, nous pouvons voir que le solver par défaut est full SVD. Nous utiliserons donc celui-ci.

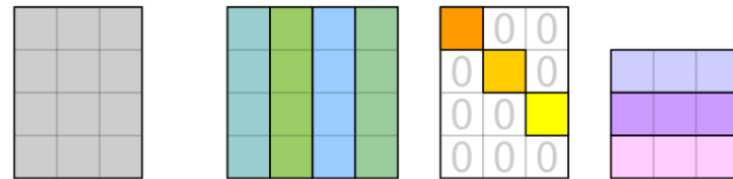
3.2.3.3 Singular Value Decomposition

Ce concept important de l'algèbre linéaire est la décomposition de valeur singulière. Le principe repose sur la décomposition d'une matrice dans le produit unique de trois autres matrices.

Voici la formule de la matrice SVD :

$$M = U\Sigma V^* \quad (3.1)$$

- M : matrice originale à décomposer
- U : matrice singulière gauche
- Σ : matrice diagonale composée des valeurs singulières
- V : matrice singulière droite



$$\begin{matrix} \mathbf{M} & = & \mathbf{U} & \mathbf{\Sigma} & \mathbf{V}^* \\ m \times n & & m \times m & m \times n & n \times n \end{matrix}$$

Ainsi Σ est une compression de M .

3.2.3.4 Obtention des valeurs singulières

Pour pouvoir récupérer ces valeurs singulières, nous devons tout d'abord utiliser la méthode *fit* du module *PCA* en donnant en argument une matrice d'image. Cette méthode va ajusté le modèle *PCA* avec le tableau matriciel donné en argument. Une fois cette méthode appelé nous pouvons récupérer le

tableau dimension 32 de composantes principales pour l'image en récupérant l'attribut *PCA.singular_values_*.

Nous proposons donc la méthode *getFeaturesFrom* qui prend en argument le tableau matriciel d'une image pour laquelle on souhaite obtenir les composantes principales :

```
>>> img = imread("../data/train/angry/Training_3908.jpg")
>>> fl.getFeaturesFrom(img)
array([[1463.20051187, 1041.27752001, 705.40992729, 524.92969204,
        499.25223095, 415.78966322, 389.68028775, 347.20089672,
        335.75781336, 308.69813195, 262.60636846, 238.65044388,
        221.03184971, 195.76879577, 184.66529048, 150.02190822,
        146.27709585, 143.3023071 , 133.10753335, 120.55399981,
        115.19257751, 103.59920413, 91.45798251, 88.6008218 ,
        84.27682517, 75.96782545, 70.70037924, 68.23940521,
        63.69026684, 63.33059541, 50.94433371, 43.75068823])
```

Nous utiliserons notamment cet objet dans le module *DataLoader*.

3.2.4 DataLoader.py

Le module *DataLoader* va nous permettre la création d'un unique objet pour manipuler toutes les données produites précédemment. De façon à centraliser toutes les informations des images en un seul objet.

3.2.4.1 Création

L'initialisation de l'objet demande de lui fournir en argument le dictionnaire de données des images obtenus après exécution de la méthode *generateTrainAndValidMatrixImg* du module *MatrixLoader* comme expliqué en section 4.2.2.2. Et également le dictionnaire du derniers index des sections train et valid qui est complété par cette même méthode.

```
>>> dl = DataLoader(ml.getDataImg(), ml.getDictionnaryEndIndex())
```

A la création de l'objet, un attribut contenant un *PCA* est effectué.

3.2.4.2 Les dictionnaires du module

Dans cet objet sera présent trois dictionnaires qui lui seront propres. Ils auront tous pour clés le nom des sections :

- *'train'* : list
- *'valid'* : list

Et ils ne tiendront compte que de l'appartenance à l'une de ces deux sections pour chaque image sans considération de leur catégorie. Nous aurons donc un dictionnaire pour :

- *d_cat_index* : les tuples (catégorie,index) des images dans les dictionnaires ci-dessous
- *d_img* : les matrices des images
- *d_features* : les tableaux de SVD des images retournés par la méthode *fl.getFeaturesFrom(img)*

Des méthodes de getter et d'ajouts dans les sections pour les dictionnaires sont également proposés.

3.2.4.3 Génération des dictionnaires

Pour la génération des dictionnaires nous mettons à disposition une méthode *load* qui va itérer sur les clés du dictionnaires de données du *MatrixLoader* donc sur les catégories des images. Et qui va appeler la méthode *loadSectionFeaturesFromCategory* qui prendra en argument la catégorie ainsi que la section (train ou valid). Un appel pour les deux sections est effectué à la suite par catégorie. Une trace d'exécution est affiché avant chaque tour de boucle.

La méthode *loadSectionFeaturesFromCategory* aura pour responsabilité d'itérer sur le contenu du dictionnaires de données pour une catégorie et une section donné en argument. Et ajoutera dans les trois dictionnaires respectifs présentés précédemment pour la section donné en paramètre. le tuple

(catégorie,index), la matrice et le tableau de SVD de l'image actuellement traité. Une barre de progression est affiché à l'aide de la librairie *tqdm* pour un meilleur suivi du processus.

Ainsi nous pouvons effectuer le lancement de la génération avec l'appel à la méthode *load* :

```
>>> dl = dl.load()
```

3.3 Phase d'apprentissage

Dans la phase d'apprentissage nous allons fournir à notre modèle, un perceptron multicouche les données obtenues précédemment des images de l'ensemble d'entraînement (section train). Dans le but de lui permettre de reconnaître avec une certaine précision la catégorie de l'expression faciale d'un visage sur une image. Le modèle aura connaissance de la catégorie des tableaux de SVD des images qui lui seront donnés en entrée. Nous allons mesurer la précision du modèle après cette phase d'entraînement.

Ensuite, nous allons contrôler les prédictions des catégories des images de l'ensemble de validation qui sont donc différentes de l'ensemble d'entraînement. Pour venir mesurer la précision et le comportement du modèle suite à la phase d'entraînement. Si la précision est jugée insuffisante en comparaison à la précision aléatoire : $\frac{1}{N}$ où n est le nombre de catégories. Alors nous proposerons une recherche par grille (grid-search) des hyperparamètres en faisant varier le nombre de catégories et le nombre de couches cachées du modèle.

3.3.1 TrainingModel.py

Nous proposons un module *TrainingModel* qui propose une fonction de d'entraînement pour permettre de mener à bien la phase d'entraînement et de validation.

3.3.1.1 Prototype de la fonction d'entraînement

La fonction d'entraînement a pour prototype :

```
train(mlp, dataloader, categories)
```

Cette fonction prend donc en arguments :

- `mlp` : un objet `MLPClassifier` à entraîner
- `dataloader` : un objet `DataLoader` avec ses dictionnaires complétés
- `categories` : la liste des catégories des données

Et retourne le tuple contenant dans l'ordre :

- la précision de la phase d'entraînement
- la précision de la phase de validation
- la précision aléatoire : $\frac{1}{N}$ où $N = |\text{categories}|$

avec $0 \leq \text{precision} \leq 1$.

Pour permettre ces phases d'entraînement et de validation, nous allons utiliser l'algorithme d'analyse discriminante linéaire qui est fourni par l'objet *LinearDiscriminantAnalysis* du module *sklearn.discriminant_analysis*.

3.3.1.2 Linear Discriminant Analysis

L'analyse discriminante linéaire est un algorithme de classification supervisée. Il permet deux approches :

- géométrique : recherche des hyperplans pour séparer les catégories
- modèle : Suppose que les lois des covariables sont des vecteurs gaussiens avec des valeurs de paramètres différentes pour chaque catégorie

Dans notre cas la librairie nous indique que l'objet *LDA* permet la création d'un classificateur avec une limite de décision linéaire, généré en ajustant les densités conditionnelles de classe aux données et en utilisant la règle de Bayes. Et que le modèle adapte une densité gaussienne à chaque classe, en supposant que toutes les classes partagent la même matrice de covariance.

Il est possible d'initialiser un tel objet avec un nombre de composantes indiqués, dans notre cas ce sera 1.

Nous aurons ensuite besoin de la méthode *fit* qui pour une liste de données et une liste de valeurs cibles passés en argument ajustera le modèle du *LDA* en fonction. Ici nos données seront les tableaux de *SVD* des images et les valeurs cibles la liste des catégories ordonnées correspondantes des images de la liste des tableaux de *SVD* (le premier argument).

Nous utiliserons également la méthode *transform* qui pour une liste de données passé en argument projette les données pour maximiser la séparation des catégories.

3.3.1.3 Etapes de la fonction d'entraînement

Ainsi la fonction d'entraînement *train* effectue dans l'ordre les étapes suivantes :

- Récupération de la liste des tableaux de *SVD* de l'ensemble d'entraînement
- Récupération de la liste des noms de catégorie dans l'ordre de la liste ci-dessus des *SVD*
- Initialisation d'un objet *LinearDiscriminantAnalysis* avec pour argument *n_components* = 1
- Appel de la méthode *LDA.fit* avec en argument les deux listes ci-dessus
- Appel de la méthode *LDA.transform* avec pour argument la liste des tableaux de *SVD* de l'ensemble d'entraînement
- Appel de la méthode *MLPClassifier.fit* avec en argument les deux listes des deux premières étapes. Ce qui ajuste le modèle du *MLP* en fonction des *SVD* et de leurs catégories
- Récupération de la précision de training du modèle du *MLP* par la

méthode *MLPClassifier.score* avec en argument les deux listes des deux premières étapes

- On récupéré les deux listes des deux premières étapes mais pour la section de validation/ensemble de validation
- Appel de la méthode *LDA.transform* avec en argument la liste des *SVD* de l'ensemble de validation
- Récupération de la précision de validation du modèle du *MLP* par la méthode *MLPClassifier.score* avec en argument les deux nouvelles listes de l'ensemble de validation
- Retourne le tuple (train_accuracy, valid_accuracy, *LDA*)

3.3.2 Précision sans hyperparamètres

Ainsi dans le programme *main.py* après exécution des phases et étapes précédentes. Nous déclarons un *MLPClassifier* :

```
>>> mlp = MLPClassifier(solver='adam', alpha=1e-5,
                        hidden_layer_sizes=(32, 32),
                        random_state=1, max_iter=10000,
                        warm_start=True)
```

NOTE : Nous faisons le choix du solver *adam* pour nos *MLP* qui fait référence à un optimiseur basé sur un gradient stochastique proposé par Kingma, Diederik et Jimmy Ba. Celui-ci nous permet un meilleur temps d'exécution et de précision que le solver *lbfgs* au vue du nombre de données à traiter. Comme nous l'indique la documentation de *sklearn*. *Adam* fonctionne assez bien sur des ensembles de données relativement volumineux (avec des milliers d'échantillons d'apprentissage ou plus) en termes de temps d'apprentissage et de score de validation. Pour les petits ensembles de données, cependant, *lbfgs* peut converger plus rapidement et mieux fonctionner.

Il est possible maintenant de lancer les phases d'entraînement et de validation pour obtenir les précisions grâce à la fonction *TrainingModel.train* :

```
>>> train_accuracy, valid_accuracy, lda =  
      TrainingModel.train(mlp, dl, cl.getCategories())
```

Nous obtenons l’affichage suivants des résultats lors de l’exécution du programme :

WITHOUT Hyperparameter Tuning :

```
— train_acc 0.26148409893992935  
— val acc 0.26497214484679665  
— rand 0.14285714285714285
```

Nous pouvons observer que nous obtenons pas une précision satisfaisantes. Nous allons donc proposer une optimisation des hyperparamètres pour notre apprentissage pour tenter d’augmenter la précision de notre modèle.

3.3.3 HyperparameterTuning.py

Le module HyperparameterTuning va nous permettre l’implémentation du processus d’optimisation des hyperparamètres. Ce processus est un procédé de recherche de la configuration des hyperparamètres, de la sélection ou non pour nos modèles pour obtenir les meilleures performances. Nous cherchons donc à améliorer la précision du modèle.

Nous pouvons également constater que les choix de paramètres statiques que nous devons fournir à nos objets *MLPClassifier* comprend la taille et le nombre de couches cachées que celui-ci utilisera. Ainsi nous pouvons effectuer une recherche par grille en faisant varier le nombre de catégorie que le perceptron traite et le nombre de couches cachées qu’il comporte. De façon à itérer sur ces deux paramètres et observer les résultats obtenus. Et pouvoir analyser la courbe de précision entre les *MLP* en fonction du nombre de catégories à devoir prédire.

3.3.3.1 Création

Pour pouvoir créer cet objet, il est nécessaire de lui fournir en paramètre l'objet *DataLoader* précédemment utilisé ainsi que la liste des catégories :

```
>>> hpt = HyperparameterTuning(dl, cl.getCategories())
```

3.3.3.2 Lancement de la recherche par grille

La méthode qui va permettre le lancement de cette recherche par grille est *startIterate*. Elle ne nécessite aucun argument :

```
>>> hpt.startIterate()
```

Pour cette recherche nous allons itérer aléatoirement sur les sous-ensembles des catégories qui ont une cardinalité minimum de 2. Si nous arrivons à l'itération où la cardinalité de l'ensemble est égale au nombre total de catégories initial. Alors nous prenons l'ensemble où toutes les catégories sont présentes. Puis nous itérons sur le nombre de couches cachées qui iront de 8 à 32 avec un pas de 8.

3.3.3.3 Etapes de la recherche par grille

Cette méthode va donc suivre les étapes suivantes :

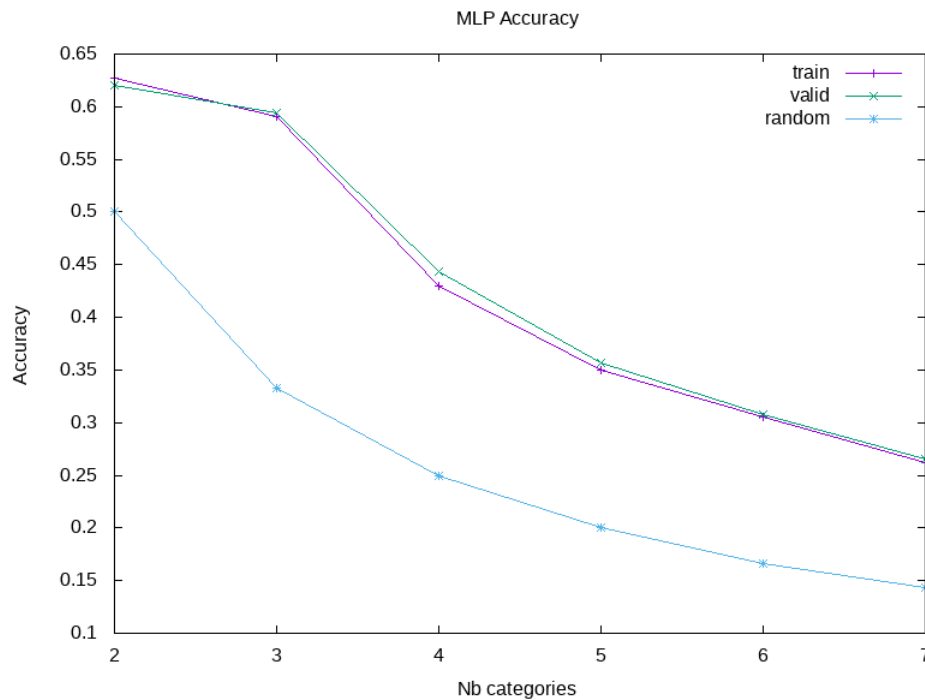
- Pour $i \in [2, N]$ où $N = \text{card}(\text{categories})$:
 - Pour $h \in [8, 32]$ avec un pas de 8 :
 - Si $i == \text{card}(\text{categories})$ alors :
nous traitons toutes les catégories
 - Sinon :
Nous traitons C où $C \subseteq \text{categories}$
 - Création d'un *MLPClassifier* comme précédemment en

modifiant l'argument *hidden_layer_sizes* = (*h*, *h*)

- Récupération du tuple (*t*, *v*, *lda*) retourné par la fonction *TrainingModel.train(mlp, dl, C)*
- On affiche la valeur *t*, *v* et la précision aléatoire $\frac{1}{i}$

3.3.3.4 Résultats

Les précisions des différents *MLP* sont affichés au fur et à mesure de l'obtention de celles-ci par une trace d'exécution. Voici les courbes obtenues :



On peut constater qu'au plus le nombre de catégories augmente au moins la précision est élevée. Ainsi dans le cas de l'utilisation de *MLP* et d'un jeu de données réduit il peut être plus judicieux d'utiliser différents *MLP* qui traiteront des sous-ensembles de l'ensemble des catégories (ensemble stable) pour obtenir par l'utilisation de ces *MLP* un meilleur taux de prédiction. Ce

procédé est coûteux car itérative dans le cas de la recherche par grille mais peut permettre dans certains cas d'obtenir de bien meilleures performances.

3.4 Phase de test

La phase de test est caractérisée par le module *GenerateTest* qui va utiliser l'ensemble de *MLP* qui traitent une taille donnée de catégories pour lequel la précision est la meilleure.

3.4.1 GenerateTest.py

Ce module va donc permettre la génération de prédictions réussies et non-réussies par les *MLP*. On pourra retrouver ces prédictions dans le dossier **data/res/found** et **data/res/fail**. Le module va prendre des images aléatoirement pour ensuite obtenir la prédiction d'un des *MLP*.

3.4.1.1 Création

Pour initialiser le module, il est nécessaire de lui fournir en paramètre les objets précédemment utilisés :

```
>>> test = GenerateTest(dl, hpt, cl)
```

3.4.1.2 Génération des prédictions

Pour lancer la génération des prédictions il est nécessaire d'utiliser la méthode *startTest* en fournissant le premier index du premier *MLP* de l'ensemble que l'on souhaite utiliser. Dans notre cas nous allons prendre l'ensemble où est contenu la meilleure précision de validation :

```
>>> hpt_hl_step = (hpt.max_layer // hpt.min_layer)
>>> nb_cat_slice =
    hpt_hl_step * floor(max(hpt.valid_accuracy)/hpt_hl_step)
>>> test.startTest(hpt_hl_step * nb_cat_slice)
```

3.4.1.3 Résultats

Voici quelques exemples des prédictions :

Found :



Fail :

