Table des matières

Depuis quelques mois, avec la sortie de l'IphoneX et sa technologique Face ID pour le déverrouillage du téléphone, la reconnaissance faciale est sur le devant de la scène des technologies en vogue. En réalité ce problème de reconnaissance de visages est un problème qui remonte aux années 60 avec les travaux de Bledsoe, Helen Chan et Charles Bisson. Ce n'est que récemment avec le développement du deeplearning que le niveau de reconnaissance devient suffisant pour être utilisé de manière fiable dans des applications. 2014 DeepFace 97,35% de reconnaissance et 2015 FaceNet 99,63% de reconnaissance.

Nous avons donc décidé d'intégrer de la reconnaissance faciale à notre gestionnaire de mots de passes afin de rendre l'expérience utilisateur la plus simple possible. Le but du gestionnaire de mots de passes est de réduire le nombre de mot de passe que nous utilisons quotidiennement à un seul mot de passe sécurisé. Avec la reconnaissance faciale nous pouvons réduire à zéro le nombre de mot de passe et rendre la connexion à un site rapide, facile et fiable. Contrairement à Face ID de Apple qui nécessite un capteur infrarouge spécial, nous voulions que le système puisse tourner sur n'importe quelle machine, ordinateur voir même mobile, et donc n'utilisant qu'un capteur optique. Nous avons donc décidé d'expérimenter différentes techniques de machine learning pour reconnaitre nos utilisateurs.

Le machine learning est un type d'algorithmes nécessitant un apprentissage, supervisé ou non, permettant d'effectuer des tâches de classification d'images par exemple. Cela nous permettrai donc de classifier nos utilisateurs de manière que si on lui présente une photo d'une personne, il puisse nous dire de quel utilisateur il s'agit. Pour nous, le principe est d'entrainer de manière supervisé un modèle avec une base de données de visage de nos utilisateurs afin qui puisse apprendre les reconnaitre et que si on lui présente une nouvelle image d'un utilisateur qu'il a appris il puisse dire de quelle personne il s'agit.

b. Intégration de la reconnaissance faciale dans le projet

Le problème de reconnaissance est découpé en 3 parties : la localisation du visage dans l'image de la webcam de l'utilisateur, la reconnaissance de ce visage qui implique l'apprentissage du visage.

Le problème de certaines techniques de machine learning, en particulier le deeplearning, c'est qu'il demande beaucoup de ressources au moment de l'apprentissage (temps et puissance de calcul). Cette étape ne peut donc pas être réalisée sur la machine de l'utilisateur. Nous allons donc devoir la déporter sur le serveur qui devra avoir une puissance de calcul suffisante.

Pour réaliser ce genre de calcul qui deviennent de plus en plus courant, le département informatique de L'Université en partenariat avec Orange à investit dans un ordinateur pour réaliser des calculs sur carte graphique. En effet les cartes graphiques permettent de paralléliser un maximum de calcul, ce qui est particulièrement efficace pour les calculs de type vectoriel et matriciel utilisé en machine learning. Nous avons donc eu l'honneur et la responsabilité de monter, installer, configurer pour le calcul sur carte graphique et inaugurer cette machine dans le cadre de notre projet. C'est donc cette machine qui nous servira de serveur.

Si la partie d'apprentissage du modèle demande beaucoup de ressources, l'utilisation du modèle entrainé pour reconnaitre les visages peut se faire avec très peu de ressources et peut donc être réalisé sur la machine de l'utilisateur.

De plus la détection du visage n'est également pas très couteuse et peut être faite sur la machine de l'utilisateur.

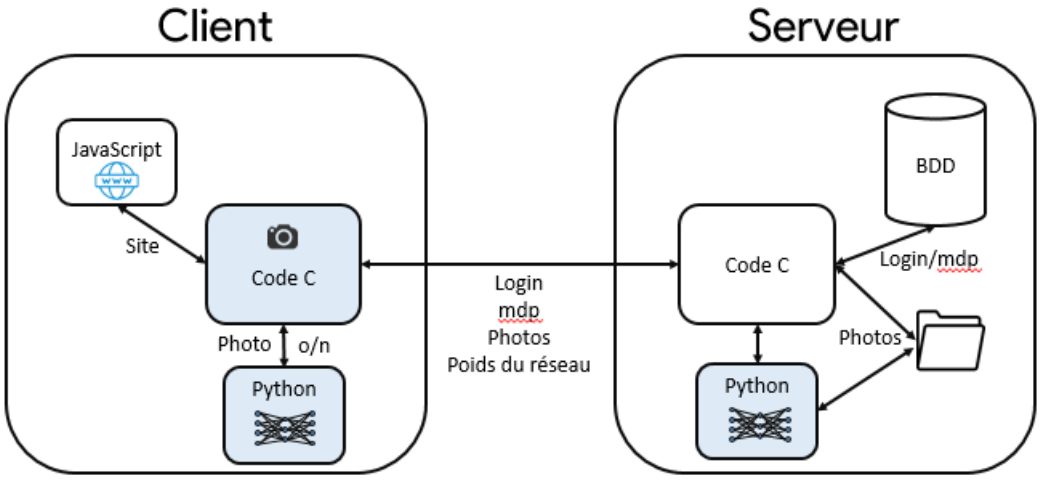


Figure 1 Localisation des blocs de reconnaissance dans l'architecture de Face Key, en bleu.

Nous avons donc sur notre client le code principal en C qui possède une fonction qui prend une photo, détecte le visage de l'utilisateur et envoi la partie de l'image correspondant au visage au code de reconnaissance. Ce dernier, en python, possède une version entrainée du modèle de reconnaissance et va prédire la classe de l'image qu'il a reçu et la renvoyer au code principal. Le code principal vérifie

qu'il s'agit bien de la bonne classes en fonction du numéro de l'utilisateur et si c'est bien le cas il fait la demande de login/mot au serveur. En réalité, pour augmenter la fiabilité, le système va prendre plusieurs images et regarder la moyenne des identifications. Nous avons fixé le nombre d'images prisent à 5 avec un très léger temps entre les images pour minimiser la possibilité d'erreur. Si la personne est bien identifié les images du visage sont envoyaient au serveur pour l'apprentissage continue.

Sur le serveur se déroule l'apprentissage du modèle de reconnaissance, il est réalisé une fois par jour. Il prend en compte les nouvelles classes créées (nouveaux utilisateurs) et les images utilisateurs courant qu'il a reçus à chaque connexion à un site. Ce système permet un apprentissage continu des classes. Si un utilisateur se laisse pousser la barbe tout en continuant à utiliser notre application alors le modèle apprendra à le reconnaitre avec sa nouvelle barbe. Une fois le nouveau modèle appris, il le transmet à tous les utilisateurs pour que les prochaines reconnaissances sur les machines des utilisateurs prennent bien en compte les nouvelles images apprissent.

c. Construction de la base de données d'images

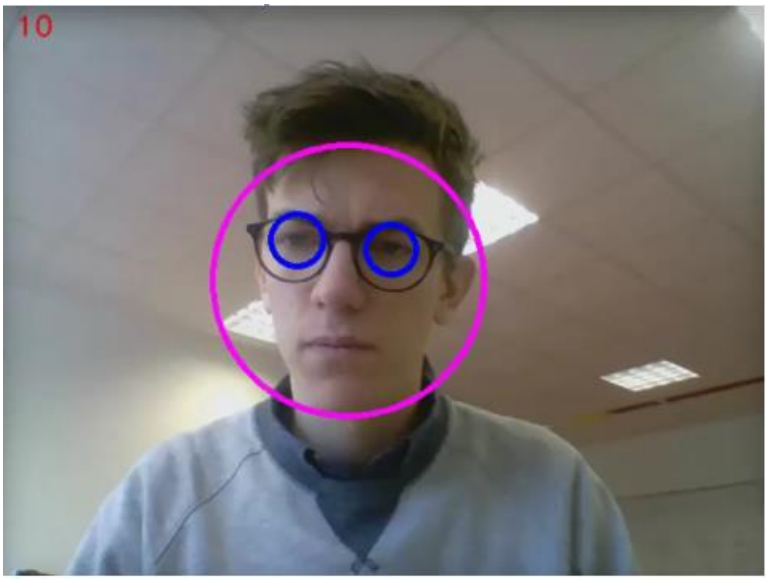
La partie commune des différentes méthodes de machine learning que nous allons utiliser est qu'elles nécessitent une base de données pour entrainer le modèle de reconnaissance. Cette base de données doit être labélisé car nous utilisons des méthodes d'apprentissage dites supervisées, c'est-à-dire que lors de l'entrainement, on présente l'image à apprendre au modèle et on lui dit à quelle classe appartient cette image. Ensuite, selon la technique d'apprentissage, la modèle va se modifier pour faire la corrélation entre l'image et la classe lui ont été donnée. La construction de la base de données est donc une étape primordiale car si les images et les labels qui lui sont présentés ne sont pas cohérant alors le modèle apprendra de mauvaises choses.

Pour notre application il faut donc construire une base d'images des visages de nos utilisateurs, labélisé avec leur nom. Pour des raisons de relatif anonymat nous n'utilisons pas le nom des utilisateurs comme label mais leur ID client que nous sommes les à connaitre.

Le problème est de réunir assez de photos du visage d'un utilisateur sans que ce processus soit trop lourd pour l'utilisateur.

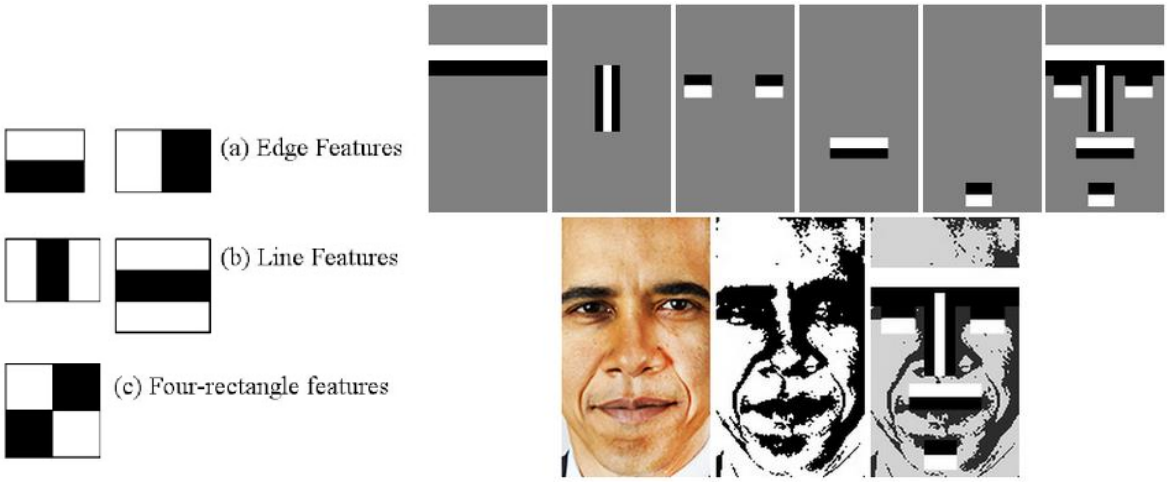
Comment l'utilisateur entre-t-il dans la base de données Face Key ?

Pour cela, à la création du compte Face Key, nous demandons à l'utilisateur s'il veut que le programme apprenne à reconnaitre son visage. S'il refuse, il ne pourras pas utiliser la reconnaissance faciale mais pourra utiliser Face Key comme un simple gestionnaire de mot de passe. Si il accepte, la fonction d'enregistrement d'images se lance, sa webcam s'allume, on détecte le visage de l'utilisateur dans chaque image du flux vidéo et on les envois au serveur pour entrainement. Le flux vidéo est pris pendant 1 minutes, pendant ce temps nous demandons à l'utilisateur de tourner sa tête sous différents angles et de réaliser différentes expression faciale pour que le modèle apprenne à reconnaitre l'utilisateur dans différents contextes. Les images transmissent au serveur sont des images couleur de 100px/100px.



Comment détecte-on le visage de l'utilisateur dans une image ?

Pour ce faire nous utilisons un algorithme appelé Haar Cascade qui utilise le principe de détection de features. Un visage est décomposable selon certain nombre de features simple.



Ensuite on utilise nos features simple comme des convolutions sur notre image et on regarde a quel endroit de la nouvelle image obtenu se trouve le pattern ressemblant le plus à notre modèle de visage simplifié.

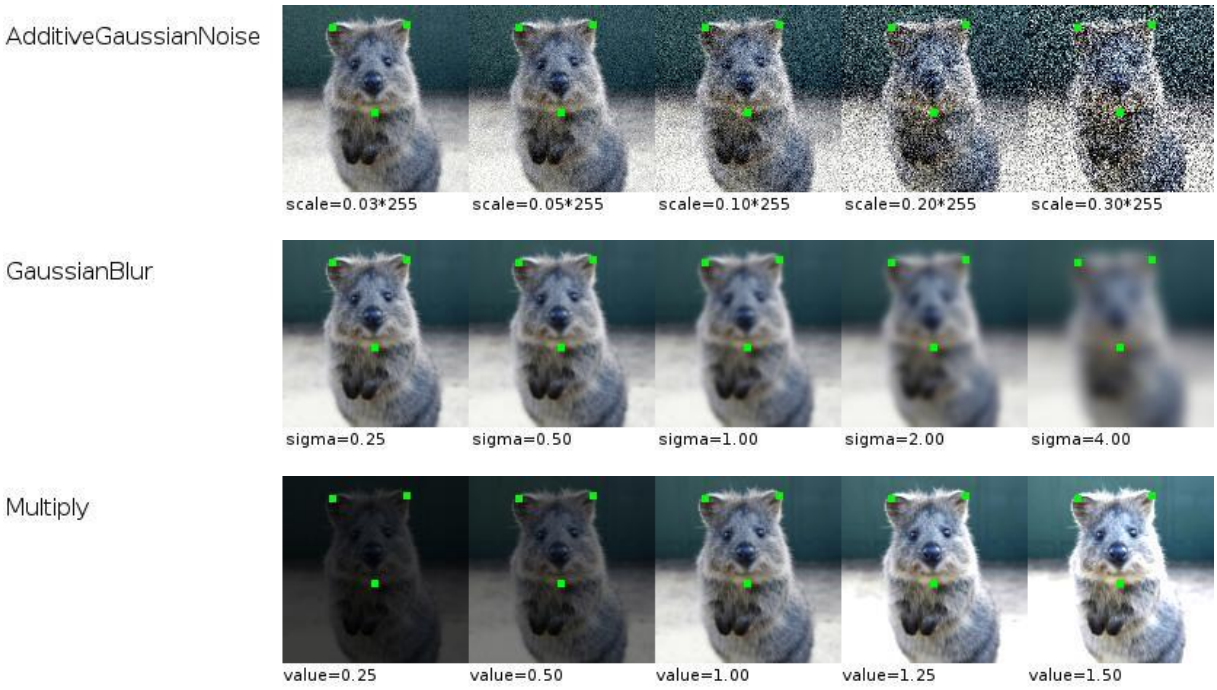
Le projet portant plus sur la reconnaissance de visage que la localisation de visage, nous avons fait le choix de ne pas implémenter l'algorithme même si initialement nous voulions implémenter une technique appelé HOG[1] (Histogram Oriented Gradient) permettant également de détecter des visages mais en vue de la charge de travail et du peu de temps impartit nous avons décidé de ne pas prioriser cette partie. Nous avons donc utilisé la fonction HaarCascadeDetection d'OpenCV.

Et une fois les photos collectées sur le serveur ?

Une fois les photos des utilisateurs enregistrées sur le serveur, elles sont passées dans un augmenteur de données. En effet certaines technique de machine learning, principalement le deeplearning,

[1]medium.com/@ageitgey/machine-learning-is-fun-part-4-modern-face-recognition-with-deep-learning-c3cffc121d78

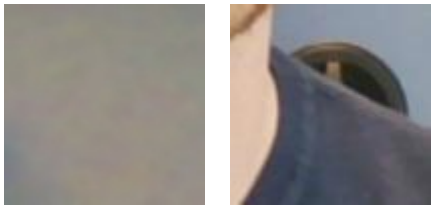
nécessite un grand nombre d'exemple d'entrainement pour être performant. Le but de l'augmenteur de données va être d'à partir d'une image de la base, créer de nouvelles images en ajoutant du bruit, floutant l'image, baisser et augmenter la luminosité …



Ce système va permettre de créer artificiellement de nouveaux contextes et donc d'aider notre modèle à bien généralisé les classes qu'il apprend. Pour cela nous avons utilisé la librairie imaug[2] dédié au machine learning et nous avons appliqué 13 transformations différentes sur nos images.

Comment avez-vous concrètement construit votre base ?

Pour construire la base de données il nous fallait des utilisateurs à pouvoir identifier. Nous avons donc fait un appel au prêt des étudiants du département informatique et des étudiants de CMI. 13 de ces derniers ainsi que 2 professeurs ont répondu à notre appel et nous avons pu construire une base de données de 17 personnes (en ajoutant nos 3 classes et en retirant la classe de Pierre F. qui a était victime délétion involontaire). Nous les avons fait suivre le protocole d'une création de compte Face Key comme décrit plus haut. Cette collecte nous a permis d'enregistrer 19462 images répartit relativement uniformément sur 17 classes et de construire une base de données de 225306 images après augmentation (soit 1.3Go d'images). Ces chiffres sont relevés après le tri de la base. En effet Haar Cascade (ou du moins son implémentation dans OpenCV) est relativement sensible à l'environnement. Des conditions de luminosité changeante pouvaient générer entre 2% et 15% d'images indésirable, des images où Haar Cascade voyait un visage alors qu'il n'y en avait pas. N'ayant pas trouvé de moyen simple pour automatiser la délétion des images indésirables, nous avons dû nettoyer la base de données à la main.



[2] github.com/aleju/imgaug

Remarques

La construction de la base de données est une étape extrêmement importante avant d'entrainer un modèle de machine learning. Nous avons passé beaucoup de temps à construire la basse de données cependant, avec le temps, nous remarquons que cette étape à était mal faite. En effet nous nous sommes laissé embarquer dans l'objectif d'avoir une base de données très large car d'après les dires de tout le monde, le deeplearning nécessite un très grand nombre de données. Dans notre cas la base est beaucoup trop redondante. Nous tirons les images d'un flux vidéo donc de l'une à l'autre les images se ressemble beaucoup. De nous avons mal fait l'augmentation de données car trop hésitant à faire de grosse transformation. On se retrouve donc avec des images qui se ressemblent beaucoup les unes aux autres et on les duplique 13 fois sans faire de transformations suffisamment importantes. La base est donc constituée d'énormément de redondance, ce qui n'aidera pas lors de la phase d'apprentissage. Il aurait mieux valu une base beaucoup plus petite mais beaucoup plus diversifiée.

d. Apprentissage/Reconnaissance des visages

Maintenant que la base de données des visages des utilisateurs est constituée et le système de captation des visages mit en place, il nous faut entrainer notre modèle pour qu'il soit capable par la suite d'identifier les utilisateurs.

Nos travaux de références sont ceux de DeepFace et FaceNet qui obtiennent respectivement 97.35% et 99.63% de reconnaissance, c'est-à-dire que dans plus de 97% des cas ils arrivent à bien classifier le visage qui leur est présenté. Notre objectif est de se rapprocher le plus possible de ces scores.

Nous allons donc essayer différentes méthodes d'apprentissage.

Découpe de la base de données

Avant de commencer à entrainer nos modèles il nous faut, encore un peu, travailler sur notre base de données.

L'un des problèmes en machine learning est le sur-apprentissage (overfitting). En effet quand on entraine notre modèle sur notre base de données, si on le fait trop apprendre alors il se peut qu'il apprenne par cœur la base de données. Il aura alors de très bonne performance sur les éléments de la base mais lorsque l'on déploie l'application dans le monde réel alors le modèle ne sera plus capable de correctement généraliser et aura de très mauvaises performances. Pour ne pas se faire surprendre au moment du déploiement, la solution est de découper notre base en 3 sous-ensembles : le training set, le validation set et le test set. On entraine le modèle avec le training set, à chaque époque (chaque fois que l'on passe le training set tout entier) on fait un suivi des performance du modèle en regardant son taux de réussite sur le validation set sans entrainer le modèle sur ces images. A la toute fin de l'apprentissage on regarde les performances finale sur le test set, sans apprentissage, et on peut voir directement si l'on a sur-appris ou pas. Mais pourquoi utiliser un test set et un validation set ? Un des deux ne suffit pas ? Même si il n'y a pas d'apprentissage sur le validation set le modèle peu l'apprendre indirectement. En effet on va continuer à apprendre si l'on remarque que le taux de performance sur le validation set ne suffit pas, on va donc apprendre implicitement à s'adapter au validation set. Le test set est donc là pour tester le modèle sur des images qu'il n'a jamais vu, même pas implicitement. On pourra donc se fier aux chiffres de performance du test set.

Méthode par régression logistique

Théorie

La régression logistique est un modèle de classification linéaire probabiliste. C'est un outil simple de classification en machine learning, que l'on utilise souvent en premier pour tester son la complexité de son problème. Il est initialement fait pour faire de la classification binaire, classification en 2 classes.

Il est composé d'une fonction de coût S :

$$S(X^{(i)}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + ... + \theta_n x_n$$

avec :

- $X^{(i)}$: une observation (que ce soit du *Training Set* ou du *Test Set*), cette variable est un vecteur contenant $x_1, x_2, ..., x_n$
- x_i : est une variable prédictive (feature) qui servira dans le calcul du modèle prédictif
- θ_i : est un poids/paramètre de la fonction hypothèse. Ce sont ces θ_i qu'on cherche à calculer pour obtenir notre fonction de prédiction
- θ_0 est une constante nommée le **bias (biais)**

Cette fonction est ensuite passée dans une fonction sigmoid qui donnera un résultat y entre 0 et 1. Ce résultat peut être interprété comme la probabilité que le vecteur d'entré X soit de la classe 1. On a donc :

Sigmoid(S(X)) = P(y>=0,5 | X ; θ) et donc P(y<0,5 | X ; θ) = 1 - P(y>=0,5 | X ; θ)

Pour faire de la classification multi-classes on peut utiliser la technique du One-vs-Rest, c'est-à-dire que de la même manière, on calcul la probabilité 1 sorte par rapport au reste des classes. On obtient donc un ensemble de poids θ pour chaque classe.

Pour qu'il y ait adaptation il faut que les poids θi soient appris. On va donc les mettre à jour en fonction de l'erreur faite par le modèle.

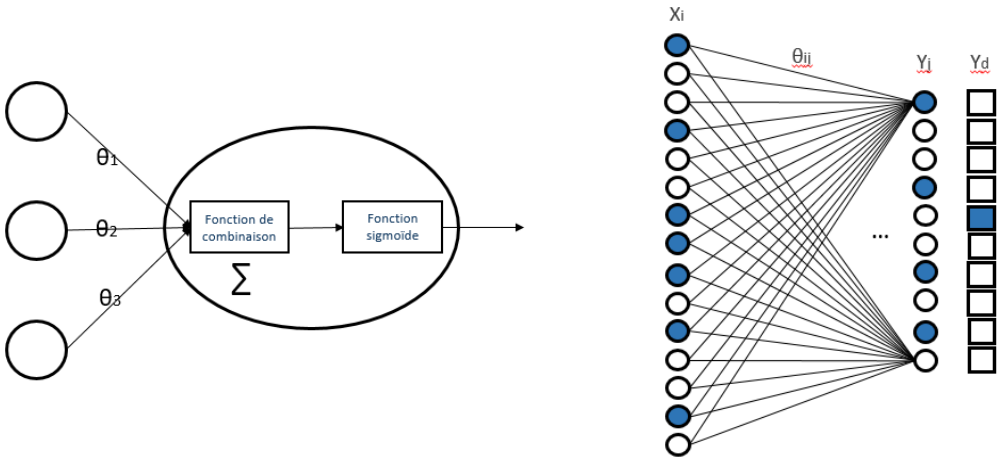
Tout d'abord on calcul l'erreur faites par la fonction Sigmoid par rapport au résultat attendu yd : err = yd – y

Ensuite on calcul de combien on va mettre à jour le poids θi : Δθi = λ . err . xi avec λ un coefficient d'apprentissage.

Enfin on met à jour le poids : θi(t+1) = θi(t) + Δθi

On met ainsi à jour tous les poids du modèle de façon à ce qu'ils s'adaptent en fonction de l'erreur qu'ils ont provoqués.

La régression logistique peut aussi se représenter sous forme de réseau de neurones (perceptron simple) où les neurones prennent un vecteur d'entré qu'il passe dans une fonction sigmoid et renvoi le résultat en sortie. Le réseau sera donc défini comme suit :



Implémentation

On utilise donc les images de visages de nos utilisateurs (en échelle de gris) sous forme de vecteur, c'est-à-dire avec les colonnes mise les unes à la suite des autres, pour alimenter ce modèle. L'implémentation de l'algorithme est faite « from scratch » en C.

Résultats

Après avoir passé l'entièreté du training set on obtient 53,2% de reconnaissance sur le test set.

Ce résultat très faible, mais au-dessus de l'aléatoire qui est à 6%, n'est pas étonnant car la régression logistique est une méthode de classification linéaire et notre problème semble être non linéaire.

Nous ne retiendrons pas cette méthode pour le système d'identification de Face Key.

Nous avons également essayé avec la bibliothèque sklearn qui utilise une version amélioré avec une fonction softmax à la place de la sigmoïde et offre un affichage facile des résultats. Nous obtenons un score plus élevé de 62.3% de reconnaissance sur 5000 exemple d'apprentissage.

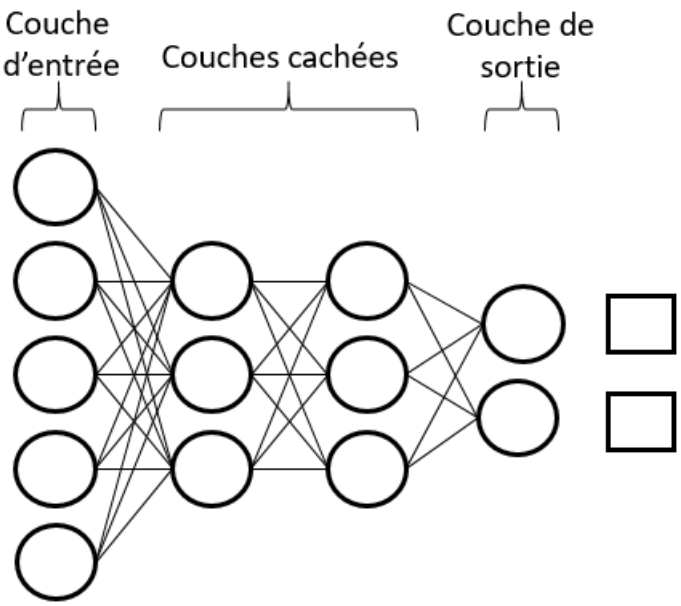
Si l'on s'amuse à afficher les ensembles θ de chaque classe on peut voir quels pixels ont de l'importance dans la reconnaissance de chaque classe et on voit se dessiner des ébauches des visages.

Méthode par perceptron multicouches

Théorie

Le perceptron multicouche (Hinton 1986) est basé sur le principe de la régression logistique sous forme de réseau de neurones. Cependant contrairement au perceptron simple qui est une méthode de classification linéaire, le perceptron multicouche pourra résoudre des problèmes non linéaires.

Le principe est d'ajouter des couches, dites cachées, entre le couche d'entré et la couche de sortie.



Les neurones on le même comportement que dans le perceptron simple (somme des poids d'entrées et sigmoïde). La différence et dans la manière de mettre à jour les poids du réseau. L'idée est de minimiser la fonction d'erreur par le calcul du gradient et la mise à jour des poids en conséquence.

Sous la forme matricielle on suit les étapes suivantes :

Calcul de l'erreur des neurones de sortie : EQ

Calcul du signal d'erreur des neurones : EQ

Calcul du signal d'erreur de la couche précédente : EQ

Mise à jour des poids : EQ

Calcul du signal d'erreur : EQ

Et on répète ces étapes pour mettre à jour tous les poids du réseau.

Ainsi, de la même manière qu'avec la régression logistique, on peut modifier nos poids en fonction de l'erreur commise par le modèle et donc le faire apprendre. Cet algorithme s'appelle la backpropagation ou gradient-descent.

Pour aider à la généralisation, nous utilisons sur ce genre de réseau une technique de régularisation appelé dropout. Le principe est simple, on désactive aléatoirement un certain pourcentage de neurones sur chaque couche à chaque tour d'apprentissage. Cette technique va permettre un renforcement plus rapide des poids des neurones en fonction de leur importance et prévient également de l'overfitting.

Implémentation

Cette technique à également été programmé from scratch mais nous n'avons pas pu tester sa performance sur notre base de données car celle-ci est trop grande et trop complexe. Elle nécessite un réseau avec beaucoup de neurones en couche caché et le nombre d'images à entrainer ont rendu l'apprentissage trop long pour un être calculé sur CPU sur un seul thread.

Nous avons donc utilisé la librarie de machine learning Tensorflow qui propose du calcule sur carte graphique.

Modèle choisi :

Couche	**Nombre neurones**
Input	10000
Hidden 1 + dropout 0.5	1024
Hidden 2 + dropout 0.5	512
Hidden 3 + dropout 0.5	256
Output	17
Total de poids : 10 899 712	

Résultats

Avec le modèle décrit ci-dessus on obtient 67,1% de reconnaissance sur notre base de données. Ce résultat n'est pas significativement supérieur au résultat obtenu avec la régression softmax de sklearn. Le problème vient peut-être donc du type de réseau de neurones utilisé.

Avec 67,1% de reconnaissance nous sommes encore loin des 99% de FaceNet, nous ne gardons donc pas le perceptron multicouche comme solution de reconnaissance faciale.

Méthode par réseau de neurones convolutif

Théorie

Le problème avec le perceptron multicouche c'est qu'il est long à entrainer quand on augmente son nombre de neurones car tous les neurones des couches successives sont connectés entre eux. Il y a aussi une perte d'informations lorsque l'on aplatit notre image en un vecteur car la continuité qu'il a entre les différentes lignes d'une image n'est pas prise en compte.

C'est ce qu'apporte l'arrivé des réseaux de neurones convolutif (CNN, Y.Lecun 1998) et le développement du deeplearning. Le principe des CNN est d'ajouter des couches dites de convolutions avant un perceptron multicouche.

Le principe de la convolution est d'utiliser des noyaux, initialisé aléatoirement puis appris, que l'on la convoluer sur l'image d'entrée pour obtenir des filtres. Dans un filtre obtenu, chaque valeur sera le taux de correspondance entre le noyau et la partie de l'image centré en ce point.

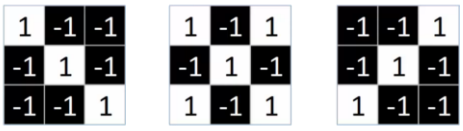


Figure 2 Noyaux

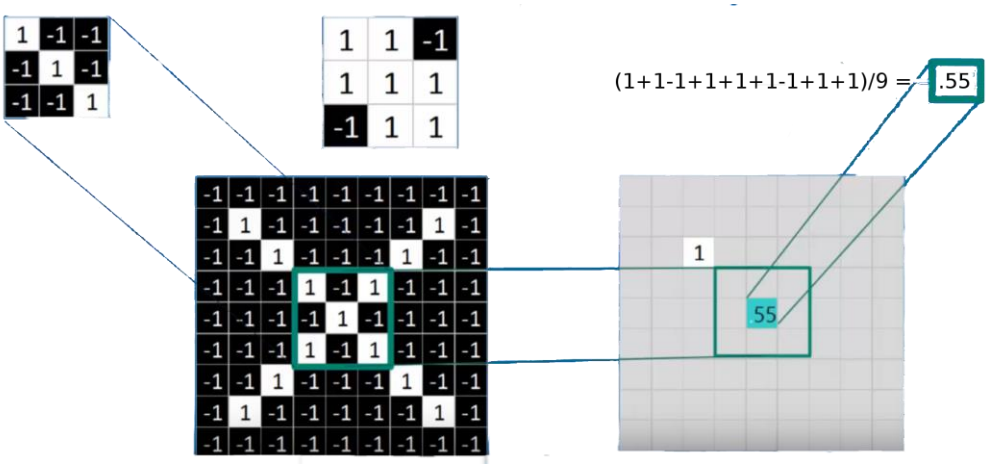


Figure 4 Opération de convolution

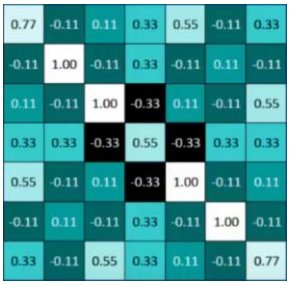


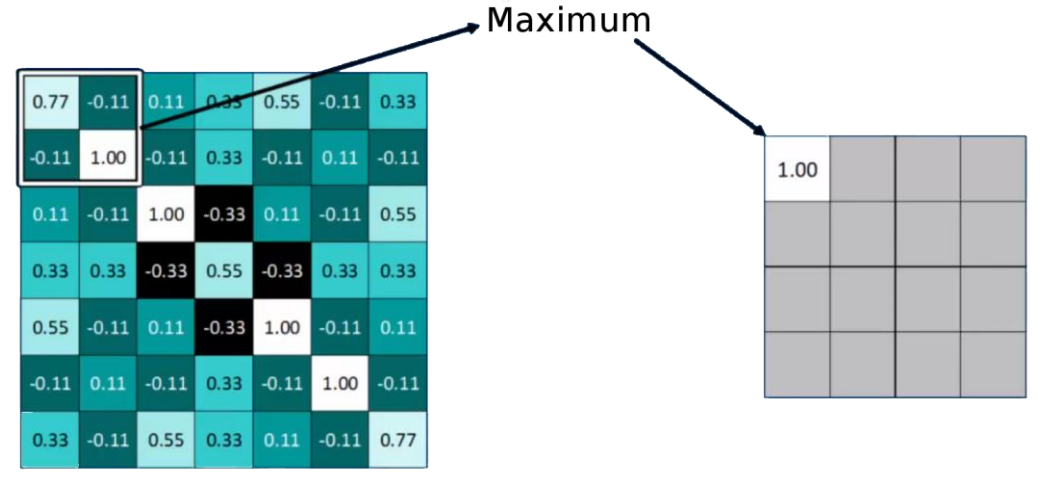
Figure 3 Filtre obtenu

On peut utiliser plusieurs paramètres pour customiser l'opération de convolution :

- Le padding : Si l'on effectue une convolution simple sur l'image (avec un 'same padding') on aura une réduction de dimension des filtres obtenu par rapport à l'image d'entrée. On peut donc entouré notre image d'entré de 0 pour obtenir la même dimension en sortie ('zero padding', utilisé dans l'exemple ci-dessus)
- Le stride : Le décalage en pixel du noyau lorsqu'on le 'glisse' pendant l'opération de convolution. Le stride à un impact sur la taille des filtres de sortie.

On peut par la suite enchainer les couches de convolution en utilisant les filtres comme image d'entrée pour la couche suivante. Les couches de convolution vont permettre de décrire l'image en différents pattern. On va donc avoir des filtres qui permettront de décrire de plus en plus précisément notre image originale.

Une autre opération que l'on introduit avec le CNN est le pooling, en particulier le max polling. C'est une couche qui va permettre de réduire la dimension de nos filtres en gardant l'information importante. On prend un noyau d'une certaine taille, en fonction de la façon dont on veut diminuer la dimension, et on ne garde que la valeur maximale de l'image dans le noyau.



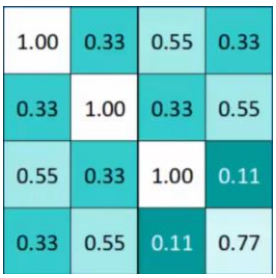
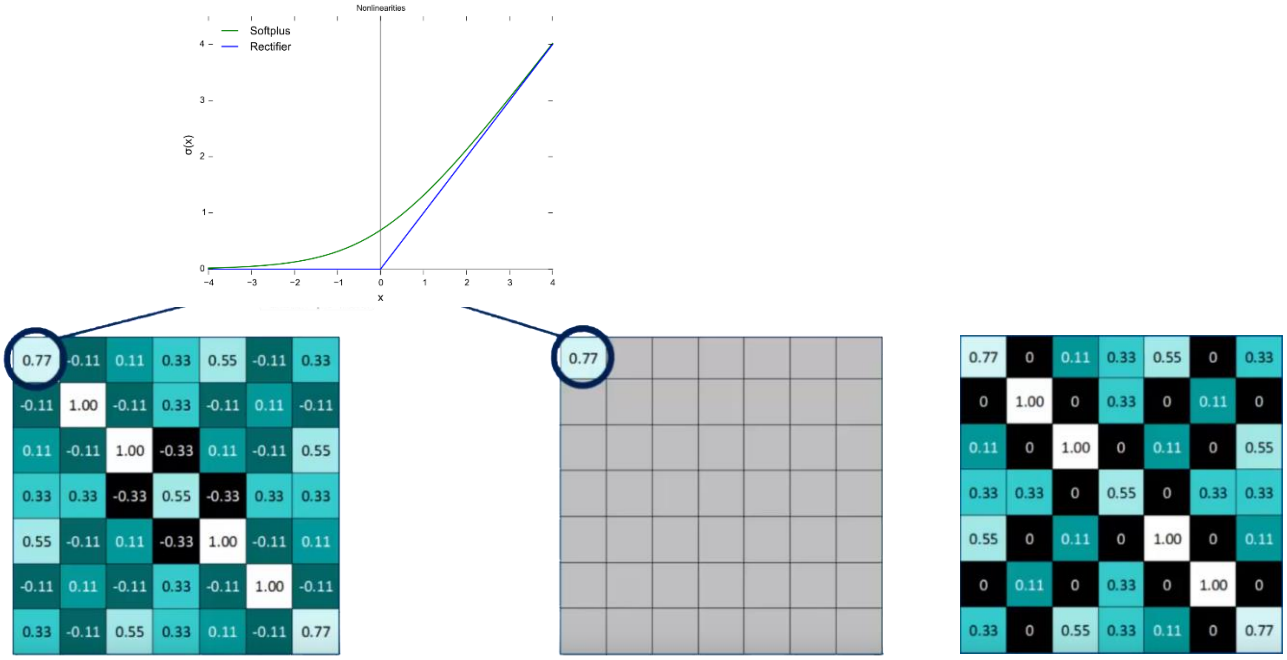


Figure 5 Résultat maxPooling

Figure 6 Opération de Pooling

On peut de la même manière joué sur le stride du pooling.

Enfin on change notre fonction d'activation sigmoïde par une fonction ReLU (Rectify Linear Unit) car la fonction sigmoïde à pour désavantage d'être 'aspirante' à ses extrémité.

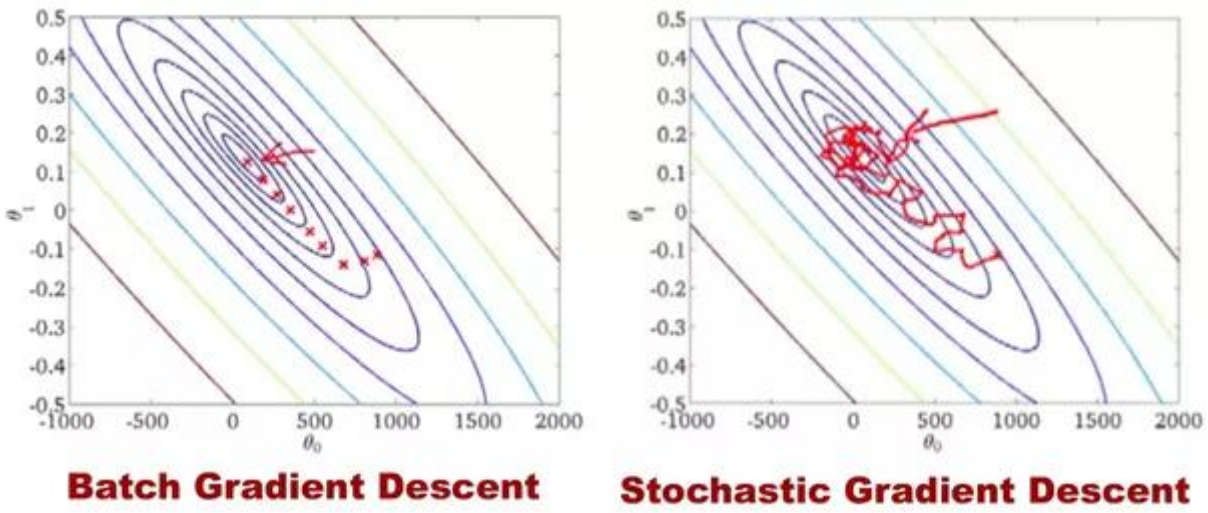


On peut maintenant mettre bout à bout toutes ces couches pour obtenir notre CNN. On suit généralement une architecture comme :

```
INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC
```

Avec FC (Fully Connected) des couches de perceptron.

Pour l'apprentissage on utilise le même principe de minimisation par descente du gradient mais de manière stochastique (stochastic gradient descent). Cette technique nécessite plus d'étapes pour converger mais est beaucoup plus rapide à calculer.



Implémentation

Cette fois ci nous n'avons pas implémenté ce modèle from scratch mais nous avons utilisé la librarie Keras qui tourne avec Tensorflow pour un entrainement rapide sur carte graphique.

Nous voulions initialement implémenter les mêmes architectures que DeepFace et FaceNet mais au moment de la création de la base de données nous avons sauvegarder les images en 100x100 alors que Deepface utilise des images de 152x152 et FaceNet .

Nous avons donc décidé de designer notre propre architecture. Nous décidons également de rester sur des images en noir et blanc, en pensant que l'humain ne nécessite pas la couleur pour identifier des gens sur une photo et nous voulions que notre réseau 'se focalise ' sur les formes des visages. Nous sommes conscients que c'est un partie-prit osé de par la quantité d'informations que l'on retire. Les résultats présenté si dessous sont donc réalisé sur des images noir et blanc, de nouveaux résultats seront peut être présenté en couleur pour la présentation.

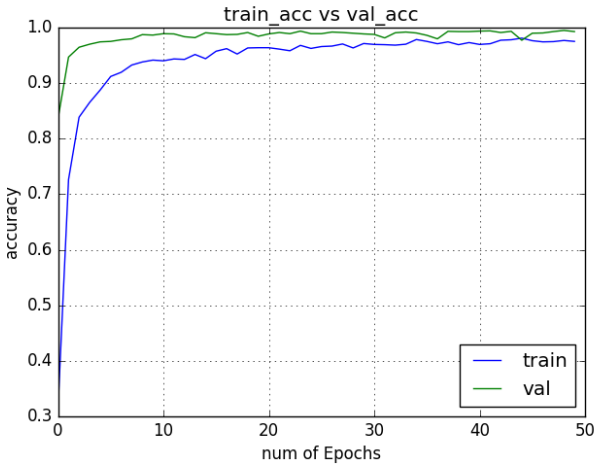
Architecture choisit :

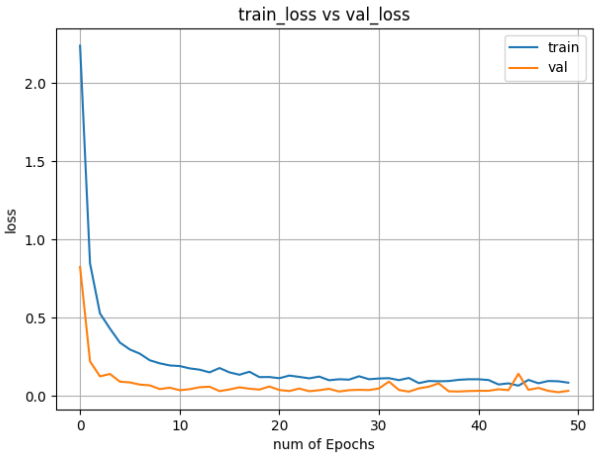
Couches	**Description**
Convolution	32 filtres, noyau (3x3)
ReLU	
Convolution	32 filtres, noyau (3x3)
MaxPooling + Dropout 0.5	Noyau (2x2)
Convolution	64 filtres, noyau (3x3)
ReLU	
Convolution	64 filtres, noyau (3x3)
ReLU	
MaxPooling + Dropout 0.5	Noyau (2x2)
Flatten	Mise en vecteur des filtres
Fully connected	64 neurones
ReLU + dropout 0.5	
Fully connected	17 neurones
SoftMax	
Total de poids : 2 052 000	

Résultats

Avec le modèle décrit ci-dessus on obtient 97,1% de reconnaissance sur notre base de données. Avec ce niveau de reconnaissance on se rapproche des résultats de DeepFace et FaceNet, ce qui est logique comme on utilise le même type de réseau qu'eux.

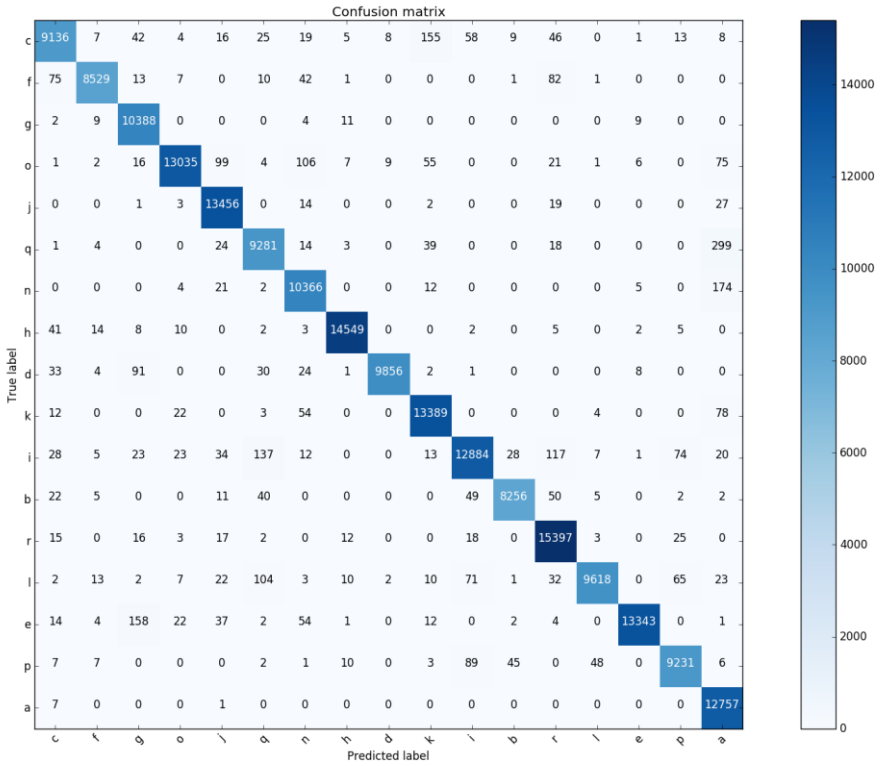
On peut observer l'évolution du loss (erreur que fait le réseau) et de l'accuracy (taux de réussite) sur la base de training et de validation pendant l'apprentissage.





L'entrainement à était fait sur 50 époques mais on constate que le l'accuracy n'augmente plus significativement à partir de la 7emes époque, les poids enregistrés sont donc ceux de la 7eme époque pour éviter l'overfitting.

On peut également afficher la matrice de confusion entre les classes :

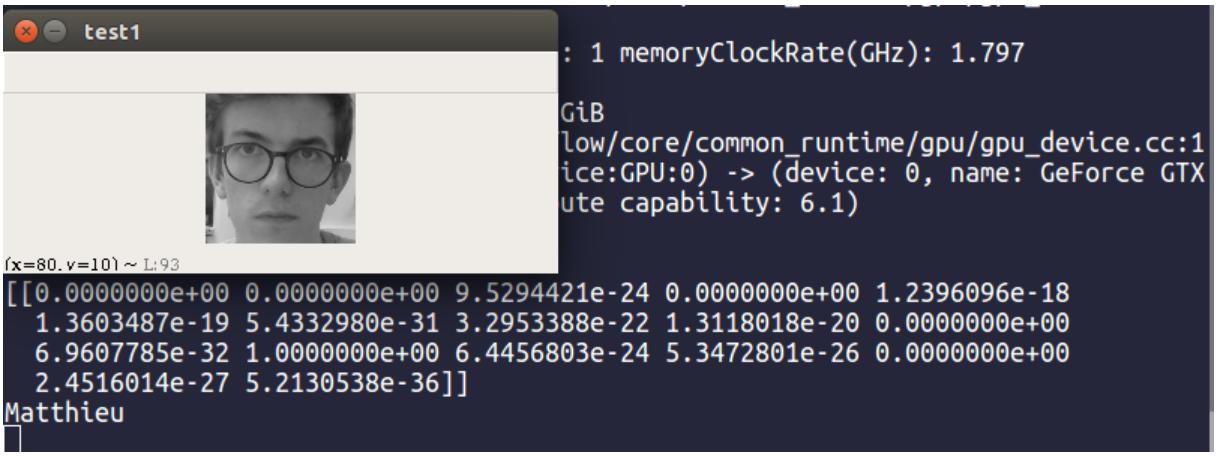


On remarque qu'il y à très peu de confusion entre les classes. De plus certaines confusion sont explicable, par exemple il y a eu 255+12=267 erreur entre la classe c et la classe k :

Les deux individus ont une forme de visage assez ronde, des yeux qui se ressemble, des cheveux longs … Les erreurs commise entre ces deux classes sont donc compréhensible (ce nombre d'erreur reste très faible par rapport au nombre d'images total)

On remarque que l'on obtient de meilleures performances sur la validation que sur le training ce qui s'explique par le fait que comme notre base est très, trop, beaucoup trop, redondante. Lors de la division en 3 sous bases, une grande partie, voir la totalité, de nos validation set et test set sont composés d'images du training set. Nous n'avons donc aucun moyen de prévenir l'overfitting de notre modèle.

On remarque effectivement l'overfitting lorsque l'on déploie notre application et que l'on test des images qui n'appartiennent pas à notre base de données. Nous avons testé sur 3 images de 3 classes différentes. Pour 2 des 3 classes le visage est reconnu.



Nous prenons donc une nouvelle photo de l'individu de la 3eme classe dans d'autres conditions de luminosité, d'angle ect. Le modèle n'arrive toujours pas à l'identifier.

Nous n'avons malheureusement pas pu faire de réel test chiffré pour l'utilisation du modèle dans le monde réel. Nous n'avons donc aucun moyen de savoir de manière fiable la performance de notre solution de reconnaissance faciale. Nous espérons pouvoir faire ces tests avant la soutenance.

e. Conclusion : remarques, avancement et améliorations possibles

Nous remarquons que la plupart de nos problèmes viennent de la mal formation de la base de données. Nous avons passé beaucoup de temps pour la constitué mais il à sans doute manqué un temps de recherche sur comment bien constituer une base de données.

La finition de cette partie n'est pas parfaite, nous avons passé beaucoup de temps à nous documenter, constituer la base, choisir les modèle, prendre en mains les outils et implémenter tous les algorithmes (y compris ceux qui ne se voient pas sur la manipulation de données pour la construction de la base). Il ne nous restait donc plus beaucoup de temps pour faire le lien entre tout ce qui a était développé.

Au niveau des améliorations possible nous pensons bien évidement à la refonte totale de la base de données (nombre, d'images, taille, augmentation différentes …). Nous pensons également à la recherche d'un modèle réellement efficace, voir sir FaceNet marche aussi bien sur une base de données maison (99,63% ont été obtenu sur la base de données Labeled Faces in the Wild LFW). Et enfin une nouvelle méthode de localisation des visages, implémentation de HOG pour test ou utiliser un réseau qui localise également le visage (Object detector, exemple réseau YOLO).

D'un point de vue pratique, mais qui soulève un problème théorique, comment allons-nous gérer l'ajout d'un nouvel utilisateur ? Il faut rajouter une classe de sortie mais doit-on ré entrainer le réseau avec uniquement les photos de l'utilisateur ? A priori ce poserais des problèmes de généralisation de cette classe, mais doit-on alors réapprendre toute la base ? Ce qui est difficile pour une application qui possède plusieurs milliers d'utilisateurs. Ce genre de problème est pour nous sans réponse pour le moment mais ils vaudraient la peine de se pencher dessus.

VIII. Sécurité

IX. Conclusion