

Projet IN52/54

Reconnaissance de signes avec la main

Introduction	2
Base d'image	2
Classifieurs vus en TP	3
Traitement des images	3
Classifieur par profil et distance minimale	6
Classifieur de densités et KPPV	7
Classifieur méthode Deep learning	8
Bibliothèque de Deep Learning	8
Présentation du réseau	9
Couches	9
Régularisation	10
Algorithme de mise à jour des gradients	11
Démarche amélioration du réseau	12
Conclusion	17

1) Introduction

Dans le cadre des UVs IN52 et IN54, nous avons choisi de travailler sur le sujet suivant la reconnaissance de signes avec la main. C'est un sujet très important en vision par ordinateur, la détection des gestes d'une main est utilisable dans de nombreux domaines dont la réalité virtuelle avec l'outil Leap Motion qui se base sur cela.

Dans un premier temps nous avons utilisé les classificateurs vus en TP : Profils et distance minimale, densités et KPPV (K Plus Proches Voisins). Ensuite nous avons choisi d'utiliser la méthode du Deep Learning que nous avons vu en cours. Enfin, nous allons conclure par une comparaison des résultats obtenus avec les différentes méthodes.

2) Base d'image

Nous avons utilisé la base d'image "Hand Gesture Recognition Database" du site internet Kaggle. Cette base de données est composée de 10 signes et de 2000 images par signes.

Nous avons choisi de n'utiliser que les signes "Doigt", "OK", "Paume", "Poing", "Pouce" afin de pouvoir potentiellement placer un Emoji sur la main par la suite (but initial du projet).

Ce qui nous fait un total de **10 000 images d'entraînement**.

Les images de tests sont des images prises manuellement à l'aide d'un smartphone puis traitées. Il y en a 10 par signe.

Ce qui fait un total de **50 images de test**.

Exemple d'exemple d'une image de la base d'image.



3) Classifieurs vus en TP

Afin de commencer à utiliser les classifieurs vus en TP, nous avons besoin de traiter les images de notre base en entrée.

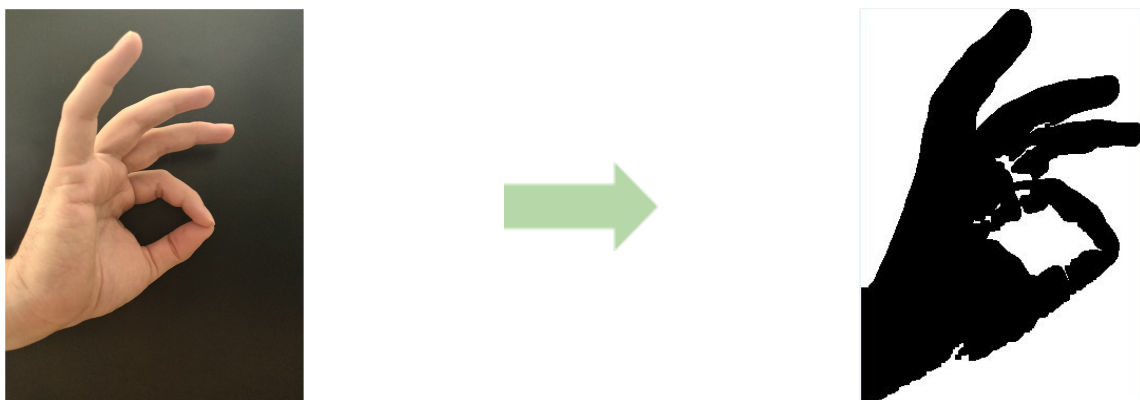
a. Traitement des images

Le but du traitement des images est de nous permettre d'avoir une représentation fidèle de la main au format binaire.

Image de la base d'apprentissage :

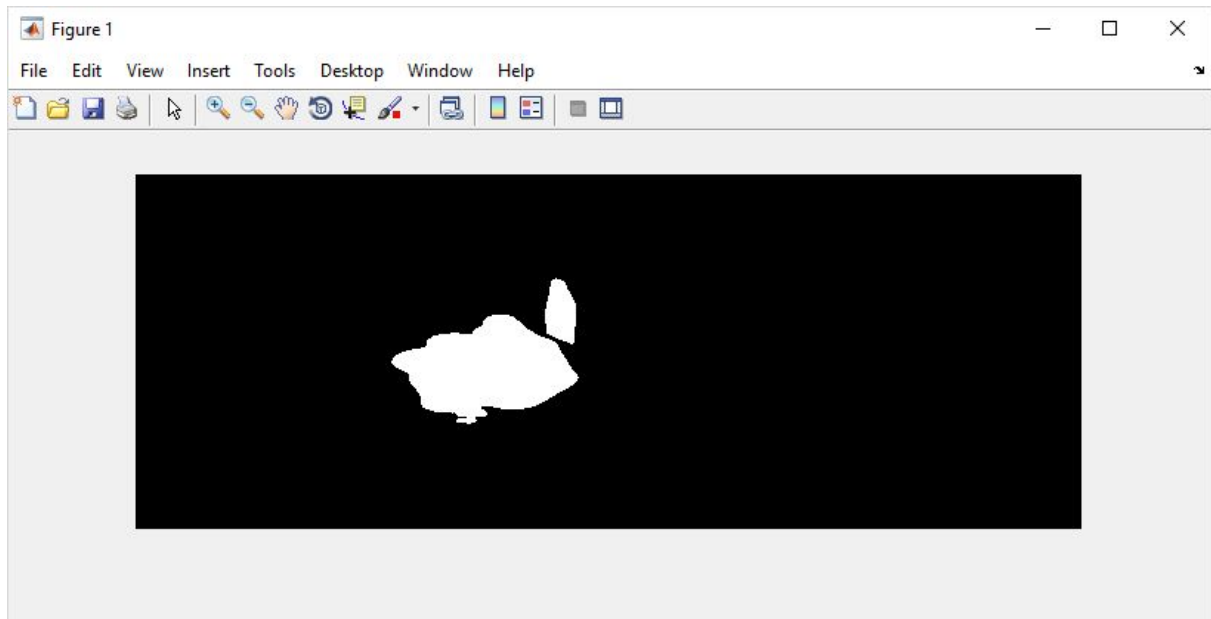


Image de test :

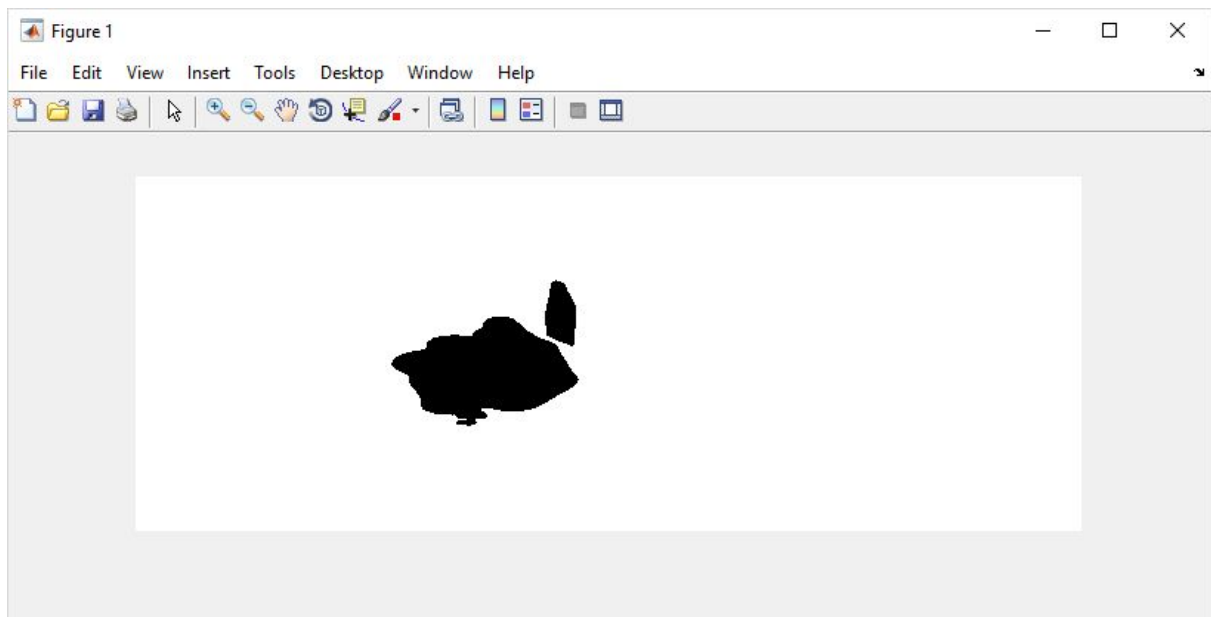


Pour obtenir les images ci-dessus, nous avons utilisé un seuil automatique pour certaines classes (obtenu avec la méthode d'Otsu), puis pour d'autres un seuil commun a été utilisé car nous obtenions des résultats plus satisfaisants.

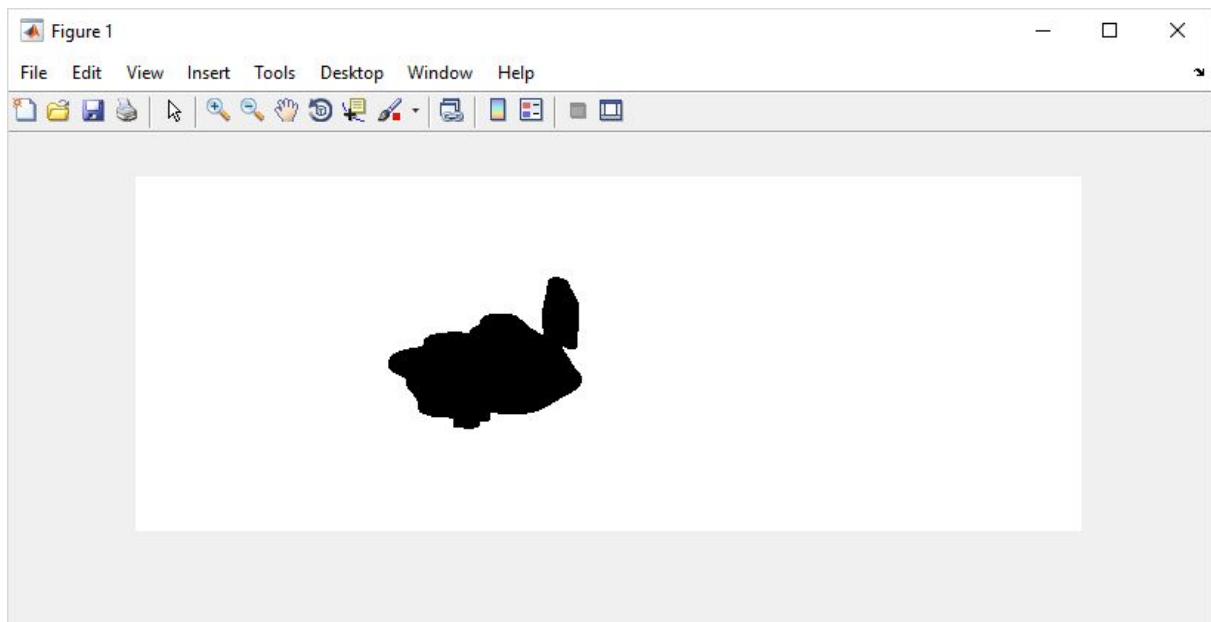
Une fois le seuil obtenu, nous avons binarisé l'image (avec *im2bw*) puis rempli les "trous" de l'image (avec *imfill*).



Ensuite, nous avons utilisé l'érosion avec un masque rectangulaire (grâce à la fonction *strel*). Mais pour l'appliquer à l'image, nous devons inverser le blanc et le noir.

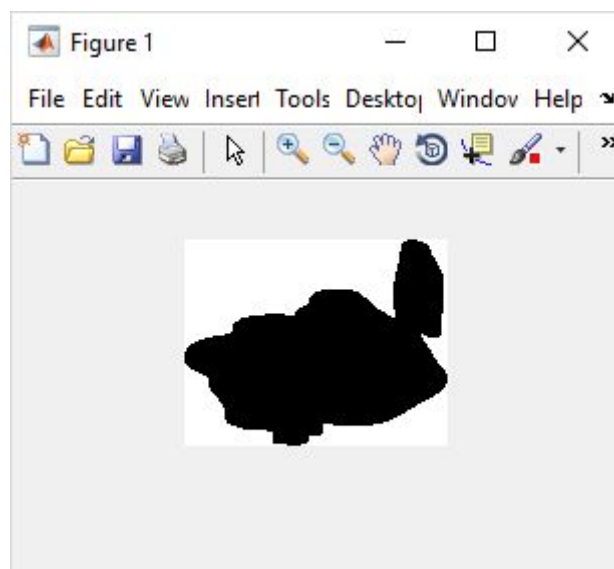


On applique donc ensuite l'érosion ce qui nous donne :



Dans l'exemple ci-dessus l'érosion n'apporte pas grand chose mais s'il y a du bruit suite à la binarisation sur l'image, cela nous permet de l'enlever.

Finalement nous avons enlevé toutes la partie blanche autour de l'image afin de faciliter la classification par la suite.



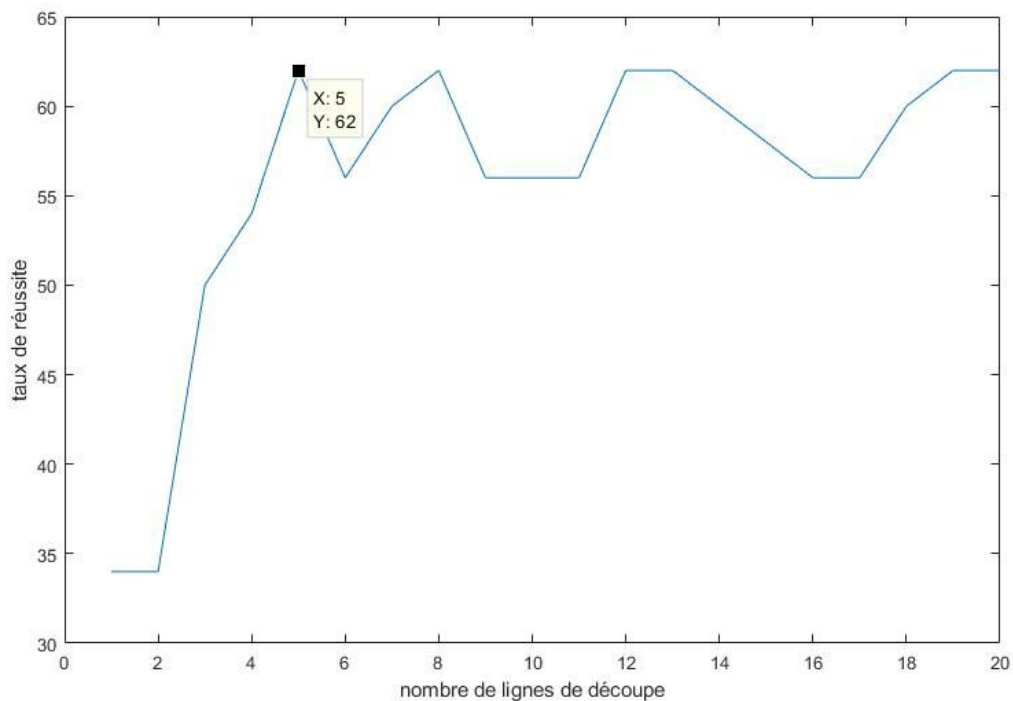
Certaines images de la base d'image de départ ont pu ne pas être retenues car lors du traitement des images, si du bruit persiste autour de la main, ces images ne sont pas gardées, tout cela de manière automatique.

Par exemple, le signe "Doigt" ne contient que 1966 images après traitement.

b. Classifieur par profil et distance minimale

Le but ici n'est pas de réexpliquer les classifieurs vus en TP mais de les appliquer tel quel à notre base d'apprentissage et à analyser leurs performances.

Voici le taux de reconnaissance obtenu en fonction du nombre de lignes par profil. On voit que le taux de reconnaissance est de **62%** pour un nombre de lignes de 5.



L'apprentissage prend plusieurs minutes et il augmente significativement avec le nombre de lignes par profil. Il est alors judicieux de sauvegarder dans un fichier les résultats de cet apprentissage.

La phase de test est quant à elle très rapide.

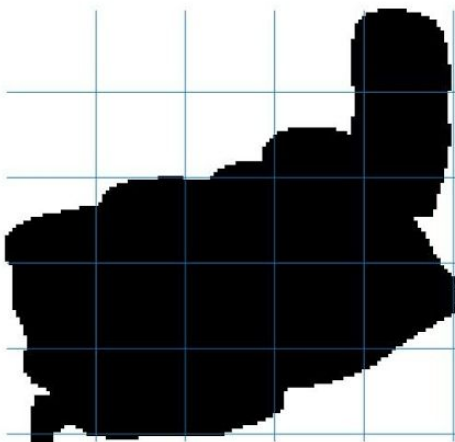
Le faible taux de reconnaissance peut s'expliquer par une différence entre la taille des mains de la base d'apprentissage, où les mains sont plus tassées et des images de tests, où les mains sont plus allongées.

c. Classifieur de densités et KPPV

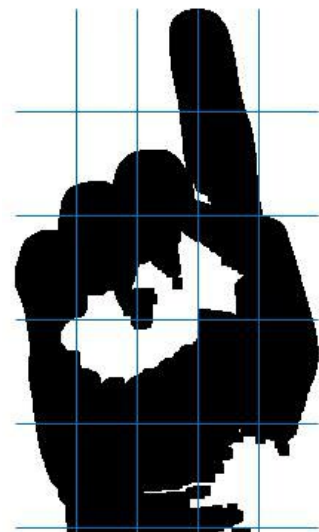
Ce classifieur découpe l'image en n lignes et m colonnes et compte le nombre de pixels noirs dans chaque zone de l'image découpée. Les densités ainsi calculées sont toutes sauvegardées. Cet apprentissage est très long (plusieurs dizaines de minutes pour traiter l'ensemble des images de la base de test).

Nous avons décidé d'utiliser ce classifieur avec comme paramètre 5 lignes et colonnes et 10 lignes et colonnes.

Résultat avec 5 lignes et colonnes et en gardant les cent meilleures densités ($k=100$) :



Signe doigt de la base d'apprentissage



Signe doigt de la base de test

On voit que l'image à tester ressemble à un index levé ainsi que sur l'image de la base d'apprentissage. Le problème est lorsque l'on prend chaque zone individuellement (c'est ce que fait le classifieur) : certaines zones sont en partie blanches dans l'image de test alors qu'elles devraient être complètement noire et l'index levé ne se situe pas dans les mêmes zones d'une image à l'autre.

Cette situation fait ainsi que le taux de reconnaissance pour l'ensemble des images de test est de **24%** avec les paramètres ci-dessus. Il tombe même à **20%** en augmentant le nombre de meilleures densités à garder, on obtiendrait ce résultat en choisissant aléatoirement un signe pour l'image à tester (il y a 5 signes différents, $100/5 = 20\%$).

Résultat avec 10 lignes et colonnes et en gardant les cent meilleures densités : **8%**

On pourrait expliquer ces très faibles résultats par le fait que certaines images de la base d'apprentissage sont très ressemblantes lorsqu'elles sont prises d'une même personne pour un même signe (200 images par personne par signe) : le classifieur peut se tromper mais lorsqu'il détermine qu'une image de la base d'apprentissage correspond à l'image de test, il va aussi déterminer que toutes les autres images qui ressemblent à cette image correspondent à l'image de test et ainsi il accroît d'autant plus son erreur.

3) Classifieur méthode Deep learning

Pour la seconde méthode de classification, nous avons choisi d'utiliser le Deep Learning avec un réseau de neurones profonds de type convolution couramment nommé CNN (Convolutional Neural Networks). C'est un type de réseau particulièrement adapté à la classification d'image.

Pour entraîner le réseau de neurones, nous avons séparé la base de données de 20 000 images en 80% d'images pour l'apprentissage, et 20% d'images pour les tests.

Le réseau de neurone ainsi obtenu est alors utilisé pour classifier les images de smartphones que nous avons nous même acquis.

a. Bibliothèque de Deep Learning

Le réseau de neurones a été codé avec la bibliothèque Keras de python, qui utilise lui-même TensorFlow, la bibliothèque de deep learning développée par Google. Lors de son installation, TensorFlow a été paramétré pour fonctionner avec CUDA, une bibliothèque de NVIDIA permettant d'utiliser la carte graphique NVIDIA. Cela permet d'accélérer de manière significative les calculs réalisés en Deep Learning (parallélisme de calcul matriciel par exemple).



Présentation du réseau

Le but de la démarche a été d'entraîner un réseau de neurones profond convolutionnel. Nous avons un réseau de neurones profonds composé de 6 couches, 4 composées de Convolution, MaxPooling et Dropout suivit de 2 entièrement connectées et utilisant la régularisation Dropout. La sortie du réseau est égale au nombre de classes attendu dans notre classification, ici 5 classes.

Voici le code en Keras de l'implémentation de la description du réseau de neurones.

```
model.add(Convolution2D(32, (3, 3), input_shape=(1, height, width), padding='same', activation='relu', kernel_constraint=maxnorm(3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))
model.add(Convolution2D(32, (3, 3), padding='same', activation='relu', kernel_constraint=maxnorm(3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

model.add(Convolution2D(32, (3, 3), activation='relu', padding='same', kernel_constraint=maxnorm(3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))
model.add(Convolution2D(64, (3, 3), activation='relu', padding='same', kernel_constraint=maxnorm(3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

model.add(Flatten())
model.add(Dense(256, activation='relu', kernel_constraint=maxnorm(3)))
model.add(Dense(128, activation='relu', kernel_constraint=maxnorm(3)))
model.add(Dropout(0.4))
model.add(Dense(num_classes, activation='softmax'))
```

b. Couches

Voici les caractéristiques des couches du réseau optimal testé.

Les convolutions sont:

- Type "same" (la taille de l'image en entrée égale à la taille de l'image en sortie grâce à un zero-padding)
- Taille de la fenêtre de convolution (3x3)

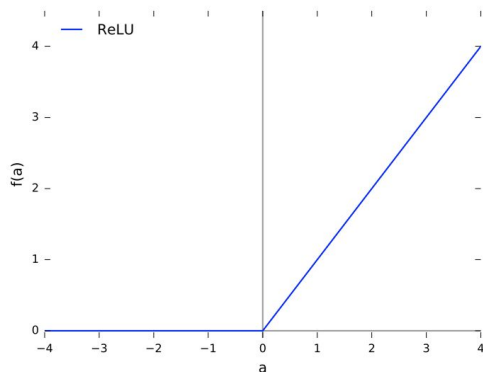
La convolution permet d'avoir d'excellent résultat en deep learning pour le traitement d'image.

MaxPooling:

- Taille de la fenêtre de pooling (2x2).

Les avantages du Max Pooling sont de réduire la taille de l'image et de compresser les données du résultat de la convolution. Ce qui permet de réduire la quantité de données à traiter pour les couches suivantes et également de concentrer les données ce qui améliore les performances du réseau.

Fonction d'activation: ReLU (Rectified Linear Unit): $f(x) = \max(0, x)$.



Quelques avantages de ReLU:

- Réduire la probabilité que le gradient ne disparaisse avec l'augmentation du nombre de couches
- Apprentissage plus rapide
- Demande moins de calculs que l'activation avec la fonction Sigmoid.

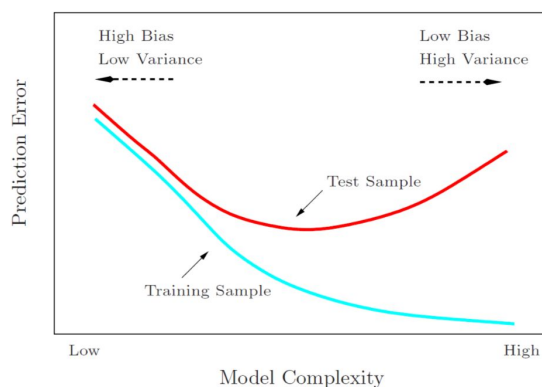
c. Régularisation

Voici les caractéristiques de la régularisation du réseau optimal testé.

Un des principaux problèmes qui a retardé le progrès de la recherche en deep learning était le surapprentissage (en anglais Overfitting): le modèle fonctionne bien sur les données d'entraînement mais pas sur les données de test.

Plus la complexité du modèle augmente, plus le modèle requiert des données afin de ne pas surapprendre.

Le schéma ci-dessous extrait du livre Deep Learning (2016) de Ian Goodfellow, Yoshua Bengio et Aaron Courville représente bien cela.



La régularisation (ou regularization en Anglais) est un moyen d'éviter le surapprentissage. Toutes les modifications sur l'algorithme d'apprentissage dans le processus de régularisation ont pour but de réduire l'erreur générale mais pas l'erreur d'entraînement.

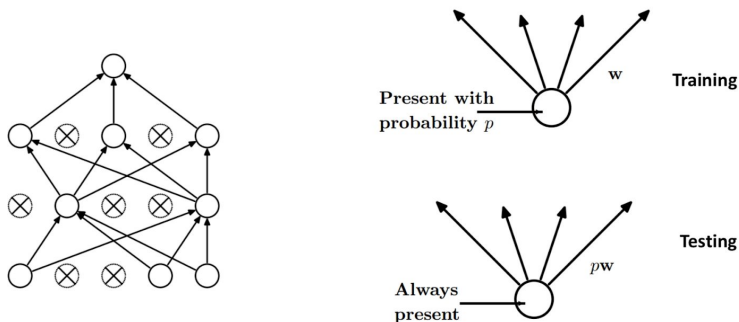
Il existe plusieurs régularisations. Nous avons utilisé la régularisation Dropout (Source: Dropout: A Simple Way to Prevent Neural Networks from Overfitting <https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf>) couplée avec la méthode de régularisation MaxNorm.

(Le papier sur le Dropout prouve que le couplage de dropout avec MaxNorm améliore les résultats finaux.)

Dropout:

Le taux de Dropout est de 0.2 dans notre configuration. Sauf pour la dernière couche ou il est de 0.4.

Le dropout fonctionne de la façon suivante, il fait ignorer aléatoirement (direction forward/avant et également backward/arrière) des neurones sur chaque couche en fonction d'une probabilité prédéfinie (dans notre cas nous avons choisi 0.2 et 0.4).



(Source: Dropout: A Simple Way to Prevent Neural Networks from Overfitting <https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf>)

MaxNorm:

MaxNorm limite la norme du vecteur des entrées d'un neurone.

(Source: Rank, Trace-Norm and Max-Norm <http://www2.mta.ac.il/~adish/Pubs/Papers/SSCOLT05.pdf>)

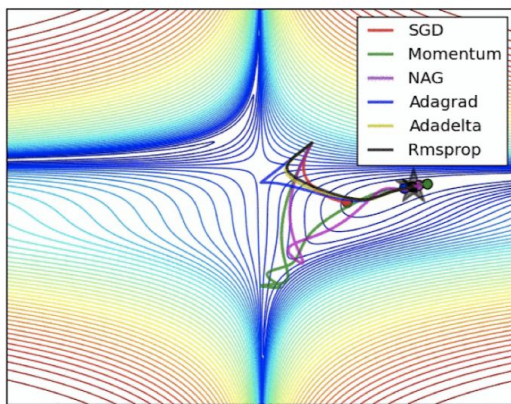
d. Algorithme de mise à jour des gradients

La mise à jour des gradients est un des éléments indispensables pour le fonctionnement d'un réseau de neurones. Pour mettre à jour les gradients, il existe plusieurs algorithmes d'optimisation.

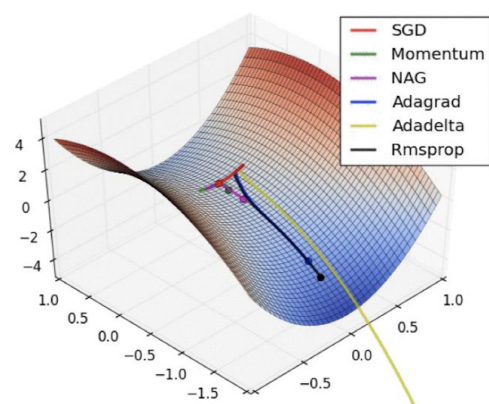
Voici des exemples d'algorithmes, "Adam", "Momentum", "RMSprop".

Certains algorithmes sont plus rapide que d'autres, certains sont précis que d'autres, certains ajoutent des hyperparamètres, etc. Ils ont tous des forces et des faiblesses. Il y est donc assez complexe de choisir un algorithme de descente de gradient. En général la méthode de sélection est de tester plusieurs algorithmes pour notre réseau de neurones. Avec beaucoup d'expérience en deep learning on peut sélectionner à l'intuition un algorithme.

Voici une visualisation de différents algorithmes pour un exemple précis.



(a) SGD optimization on loss surface contours



(b) SGD optimization on saddle point

[Source: An overview of gradient descent optimization algorithms; Sebastian Ruder; <https://arxiv.org/pdf/1609.04747.pdf>]

e. Démarche amélioration du réseau

La première étape a été de construire un réseau de neurones avec les éléments indiqués ci-dessus, avec une configuration qui permet d'exécuter quelques epoch en un temps raisonnable.


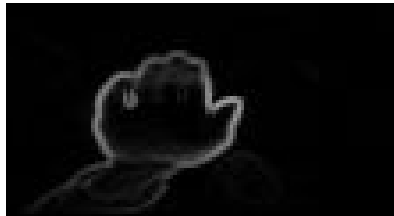
Remarque: Un epoch correspond au nombre de fois où l'on va mettre à jour les poids. 1 epoch égal toutes les données on était traité et le réseau à eu une unique mise à jour des poids.

Pour cela un premier pré-traitement a été appliqué sur les images de la base de données. Tout d'abord, nous sommes passés d'une image couleur à une image en niveau de gris, puis nous avons réduit la taille de l'image par 4.

A ce stade, le réseau de neurones s'exécutait, mais la précision ne dépassait que rarement les 20%. Pour rappel, nous avons 5 classes, donc 20% correspondent à une réponse aléatoire. Les résultats étaient très mauvais.

Lors de cette phase de développement, nous avons cherché à modifier plusieurs paramètres pour permettre au réseau d'apprendre, tel que : le nombre de blocs de couches, le nombre de convolutions à faire par bloc de convolution, la modification du taux de dropout, la suppression des dropout, le changement de la learning rate et le changement de l'optimiseur utilisé (SGD(Stochastic Gradient Descent), Adam, ...). Malgré toutes nos tentatives, l'efficacité du réseau de neurones n'a pas changé.

Cela nous a amenés à revoir notre étape de pré-process des images. Une autre approche a été utilisée. Tout d'abord, les photos de la base de données ont une dimension très particulière de 640*240 (largeur*hauteur), ce qui donne des images très allongées. De plus la main ne se trouve jamais sur les bords gauche et droit de l'image. Après examen, on peut retirer les 100 premières et dernières colonnes de pixels sur chaque image. On divise ensuite la taille de l'image par 4, et on obtient ainsi la plus petite taille d'image qui permet toujours de reconnaître tous les détails de la main. Enfin un filtre de détection des contours (Sobel) est utilisé. Cela permet de réduire le nombre d'informations disponibles sur l'image en ne gardant que les détails intéressants et de s'émanciper de la couleur de la main et du fond, très spécifique à la base de données.

Image de départ	Image après pré-process
	
640*240	110*60

Avec ce nouveau pré-process, le réseau de neurones a tout de suite présenté des résultats plus intéressants, avec 99% de bons résultats sur la base d'apprentissage, et 80% sur la base de tests.

Voici un résumé de l'architecture du réseau à ce point :

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 32, 60, 110)	320
max_pooling2d_1 (MaxPooling2)	(None, 32, 30, 55)	0
dropout_1 (Dropout)	(None, 32, 30, 55)	0
conv2d_2 (Conv2D)	(None, 32, 30, 55)	9248
max_pooling2d_2 (MaxPooling2)	(None, 32, 15, 27)	0
dropout_2 (Dropout)	(None, 32, 15, 27)	0
conv2d_3 (Conv2D)	(None, 32, 15, 27)	9248
max_pooling2d_3 (MaxPooling2)	(None, 32, 7, 13)	0
conv2d_4 (Conv2D)	(None, 64, 7, 13)	18496
max_pooling2d_4 (MaxPooling2)	(None, 64, 3, 6)	0
flatten_1 (Flatten)	(None, 1152)	0
dense_1 (Dense)	(None, 256)	295168
dense_2 (Dense)	(None, 128)	32896
dropout_3 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 5)	645
Total params: 366,021		
Trainable params: 366,021		
Non-trainable params: 0		

Ce décalage entre précision sur la base d'apprentissage et base de tests est un cas de surapprentissage très courant pour les CNN, pour réduire cet écart, nous avons ajouté deux couches de dropout là où nous les avons supprimées précédemment. Cela a eu pour effet de passer le taux de précision sur la base d'apprentissage à 99%.

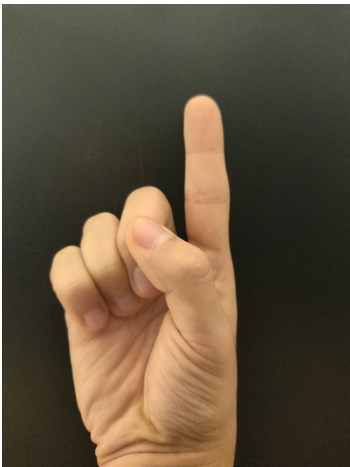


```

Train on 7200 samples, validate on 1800 samples
Epoch 1/10
- 9s - loss: 0.8746 - acc: 0.6756 - val_loss: 0.1808 - val_acc: 0.8800
Epoch 2/10
- 7s - loss: 0.1522 - acc: 0.9504 - val_loss: 0.3920 - val_acc: 0.8622
Epoch 3/10
- 8s - loss: 0.0715 - acc: 0.9771 - val_loss: 0.3254 - val_acc: 0.8822
Epoch 4/10
- 8s - loss: 0.0476 - acc: 0.9842 - val_loss: 0.0647 - val_acc: 0.9894
Epoch 5/10
- 7s - loss: 0.0361 - acc: 0.9910 - val_loss: 0.0433 - val_acc: 0.9911
Epoch 6/10
- 8s - loss: 0.0244 - acc: 0.9918 - val_loss: 0.2755 - val_acc: 0.8789
Epoch 7/10
- 7s - loss: 0.0198 - acc: 0.9936 - val_loss: 0.1247 - val_acc: 0.8878
Epoch 8/10
- 7s - loss: 0.0143 - acc: 0.9954 - val_loss: 0.0890 - val_acc: 0.9639
Epoch 9/10
- 7s - loss: 0.0147 - acc: 0.9953 - val_loss: 0.1000 - val_acc: 0.9789
Epoch 10/10
- 7s - loss: 0.0110 - acc: 0.9969 - val_loss: 0.0281 - val_acc: 0.9950
[[400  0  0  0  0]
 [  0 400  0  0  0]
 [  0  0 400  0  0]
 [  1  8  0 391  0]
 [  0  0  0  0 200]]
took 129.72890543937683 seconds

```

Le chiffre à retenir est la dernière valeur de "val_acc" : 99,5%. Il s'agit du taux de bons résultats sur les images n'ayant pas servi à l'apprentissage. En dessous se trouve la matrice de confusion. On peut donc voir que le réseau de neurone a bien classifié toutes les classes, sauf pour les images poing : Une image de poing a été détectée en temps qu'une image de paume, et 8 images de poings ont été détectés comme index.

En passant les images issues de notre acquisition à la même définition de pixel (ce qui implique de rajouter des marges pour conserver le même ratio), puis en appliquant le filtre de Sobel, nous obtenons un taux de réussite de 100%.

	
image brute	image pré-traité, lisible pour le CNN

4) Conclusion

Suite aux tests des différents classifieurs avec des images prises de nos mains, on observe que le Deep Learning est celui qui offre un taux de réussite parfait.

La principale différence entre le deep learning avec profils et distance minimale ou densités et KPPV, est que le deep learning sélectionne lui même les caractéristiques à retenir. Le deep learning est plus robuste aux changements de détails dans l'image, si le réseau n'a pas de problème de surapprentissage.

Bien que le Deep Learning soit plus complexe pour l'ordinateur (quantité de calculs) que les deux autres classifieurs (profils et distance minimale, densités et KPPV), le taux de réussite est nettement supérieur : 100% contre 62% ou 24% respectivement. En effet, la différence entre la base d'apprentissage et les images de tests affecte les résultats de ces deux derniers.

Pour conclure, nous pouvons dire que le Deep Learning est une meilleure solution que les classifieurs distance minimale et KPPV car il s'adapte mieux aux différences entre les images de la base d'apprentissage et celles des images de test.