Travail d'Etude et de Recherche : Détection d'orientation de textures à partir de réseaux de neurones

Dray Jonathan Marouf Ilyas
encadré par Richard Frédéric

22 Avril 2022

REMERCIEMENTS

Nous souhaitons remercier notre tuteur de T.E.R, Monsieur Frédéric Richard, qui a accepté de nous encadrer et de nous suivre au cours de ce projet et qui a su nous réorienter quand cela était nécessaire.

Résumé

Notre T.E.R s'inscrit dans le domaine de la reconnaissance d'image et présente quelques outils permettant de mettre en place différents modèles de classifications. Dans ce rapport, nous allons mettre en évidence différentes méthodes issues du domaine du machine learning de manière théorique pour donner au lecteur les notions essentielles et suffisantes afin de comprendre notre démarche d'expérimentation.

Our T.E.R. is part of the image recognition domain and presents some tools allowing to set up different classifications models. In this report, we will highlight different methods from the field of machine learning in a theoretical way to give the reader the essential and sufficient notions to understand our experimental approach.

Introduction

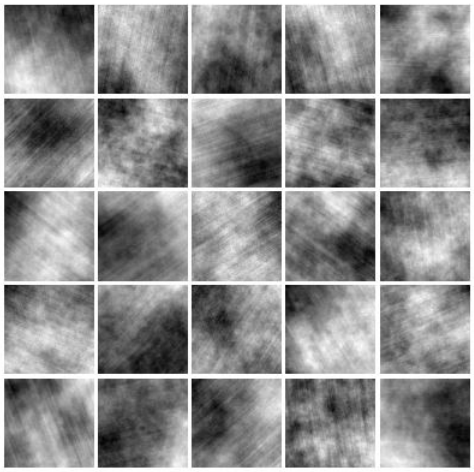
Les réseaux de neurones convolutionnels se sont présentés au fil des années depuis 1989 comme étant une solution presque indispensable dans l'imagerie et la vision par ordinateur, et sont devenu des outils importants dans le domaine du machine learning. Le travail de recherche de Yann le Cun s'inspirant du neocoginitron (précurseur du convolutional neural network), "gradient based learning applied to document recognition" relança un engouement autour des réseaux inspiré par la biologie.

Un des premiers travaux du professeur Yann Le Cun a été d'entraîner un modèle de CNN sur une base de données afin de répondre à une problématique de prédiction de reconnaissance de chiffre manuscrite, présentant en 2011 un pourcentage d'erreur de 25.8, un modèle dont nous nous inspirerons pour notre travail.

Un domaine dans lequel l'usage de ces CNN trouve son application est la recherche de l'orientions d'un motif. En effet, l'orientation d'un motif est une étude qui peut trouver ses applications dans le domaine de la médecine et de l'imagerie médical. Par exemple, Dans la détection de certaines lésion, l'étude de l'orientation des tissus peut être déterminant. En effet, une lésion peut se remarquer par une rupture dans un certain schéma de tissus.

Ce pourquoin nous avons décider d'orienter notre TER dans l'application de couche convolutive afin de prédire l'orientation d'angle de texture brownienne.

Nous utiliserons le package PyAFBF pour générer des textures d'image anisotropes. Les textures sont échantillonnées à partir d'un modèle mathématique appelé champ brownien fractionnaire anisotrope. Elles sont générées selon une certaine direction connue. Voici quelques exemples de ces textures :



Exemple de texture

L'objectif va être de mettre en œuvre un réseau de neurone dont nous allons mener l'apprentissage afin qu'il puisse à terme calculer l'angle de cette direction pour une image nouvelle.

1 Réseau de neurone

Pour commencer, le terme "réseaux de neurones" fait référence à la biologie mais n'en reste pas moins une application purement mathématique et statistique. Les classes de problèmes que les réseaux de neurones sont susceptibles de résoudre sont rappelées : modélisation non linéaire statique ou dynamique, classification, modélisation semi-physique (« boîte grise ») et traitement de données structurées (graphes).

Nous allons nous intéresse dans un premier temps à la forme la plus élémentaire d'un réseau de neurone : le perceptron.

1.1 Perceptron

Définition d'un neurone : Un neurone est un modèle, fonction à p variables, qui relie p entrées x^1,...,x^p à une sortie y :

$$y = g\left(\alpha_0 + \sum_{j=1}^{p} \alpha_j x^j\right)$$

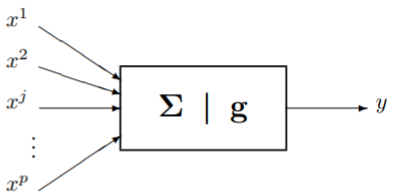


Figure 1: Représentation d'un neurone

- Σ : combinaison linéaire des entrées
- g : fonction d'activation

Σ est le produit scalaire entre le vecteur x ayant pour composante les entrées de notre modèle et le vecteur w ayant pour composante les poids des "synapses", $w = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ ... \\ \alpha_n \end{pmatrix}$

Définition fonction d'activation : la fonction d'activation est une fonction mathématique appliquée à un signal en sortie d'un neurone artificiel. Le terme de fonction d'activation vient de l'équivalent biologique « potentiel d'activation », seuil de stimulation qui, une fois atteint entraîne une réponse du neurone. La fonction d'activation est souvent une fonction non-linéaire.

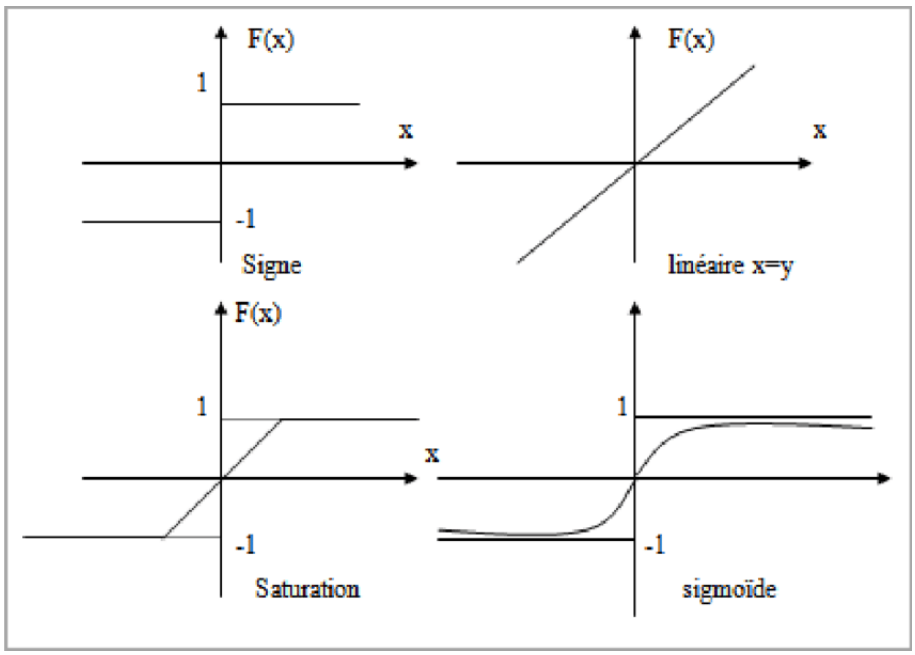


Figure 2 : Exemple de fonctions d'activations

Quelques exemples de fonctions d'activations, on remarque qu'elles sont ici comprises entre -1 et 1 (hormis $y = x$) et toute symétrique par rapport à l'origine.

Nous avons fait l'expérimentation du perceptron sur notre problématique, l'objectif étant de trouver un angle entre $[-\frac{\pi}{2}; \frac{\pi}{2}]$. Après un aplatissage de notre image, et le choix de l'identité comme fonction d'activation, nous nous ramenions à un cas de régression linéaire simple.

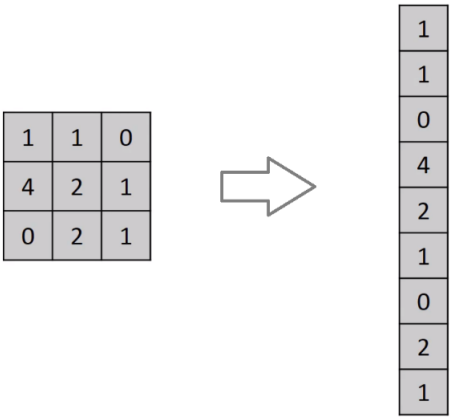


Figure 3 : Aplatissage

Nos images sont de taille 30x30. Une fois aplatit, on obtient un vecteur $x_i \in \mathbb{R}^{900}$ pour notre modèle d'apprentissage. Or d'après le modèle, pour chaque neurone, il y a un paramètre à estimer. La complexité de la problématique dans le cas continue ne pouvant pas se résoudre avec une simple régression linéaire à 900 paramètres, nous n'avons obtenu aucun résultat satisfaisant.

1.2 Perceptron multicouche

Nous cherchons donc une architecture de réseau qui nous permettrait d'augmenter notre nombre de paramètre. Nous nous sommes donc attardés sur le perceptron multicouche. (En plus de répondre à notre problème de paramètre, l'étude de plusieurs nous permettrait d'utiliser la propriété "d'approximateur universel" des réseaux de neurones.)

Le perceptron est organisé en trois parties :

- La couche d'entrée (input layer) = un ensemble de neurones qui portent le signal d'entrée.

- La couche cachée (hidden layer) dans un réseau de neurones artificiels est une couche entre les couches d'entrée et les couches de sortie, où les neurones artificiels prennent un ensemble d'entrées pondérées et produisent une sortie via une fonction d'activation.

- La couche de sortie (output layer) : cette couche représente le résultat final de notre réseau, sa prédiction.

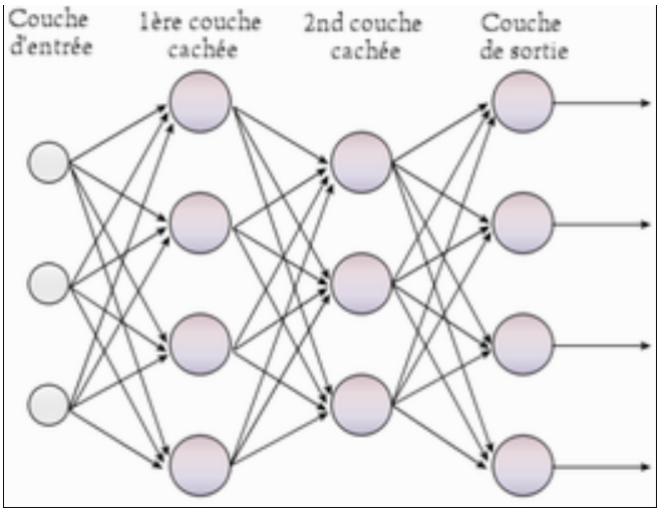


Figure 3 : Présentation du modèle de perceptron multicouche

Chaque neurone est reliée par les neurones adjacents par des poids, portant le nombre de paramètre à estimer à $p_e * p_1 * ... * p_n * p_s$

avec p_e = taille de la couche d'entrée
p_i = taille de la $i^{ème}$ couche cachée
p_s = taille de la couche de sortie.

Le nombre de paramètre ici dépend donc de la taille des couches cachés. On peut donc très facilement faire exploser le nombre de paramètre en multipliant les couches et en augmentant le nombre de neurone par couche. Cependant après plusieurs tests de diverses architectures, faisant parfois monter le nombre de paramètre à 100 000, nous n'obtenions toujours aucun apprentissage satisfaisant. Une raison assez évidente à cela serait la faite que le perceptron n'est conçu que pour des problèmes à une dimension. Or pour des problématiques qui concerne le traitement d'image, il nous fallait un outil qui puisse gérer des entrées à deux dimensions.

2 Réseaux de neurones convolutifs

Les réseaux de neurones convolutionnels sont les modèles les plus performants pour classer des images. L'idée du réseau est simple, en entrée une image est fournie sous la forme d'une matrice de pixels, un dégradé de gris dans notre situation. La convolution agit comme un filtre, représenté par une fenêtre d'une taille prédéfini. Cette fenêtre parcourt l'image en partant du coin supérieur gauche au coin inférieur droit avec un pas prédéfini.

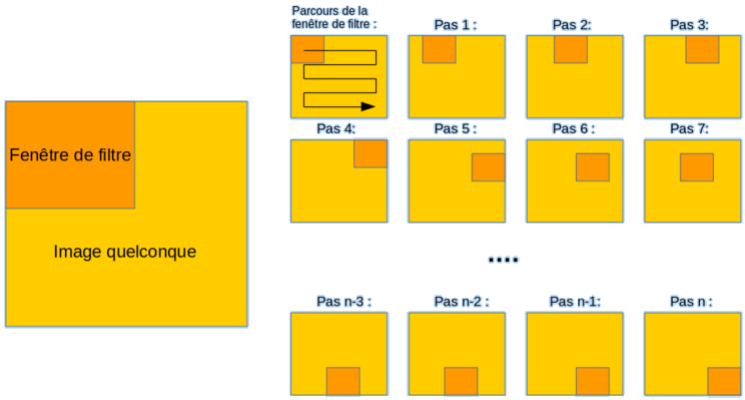


Figure 4 : Fonctionnement de la convolution

Par exemple, si on prend une convolution de taille **3x3**, le filtre va intervenir sur **9 pixels**. Pour le cas où la convolution venait à prendre le maximum des 9 pixels, on obtiendrait ce résultat.

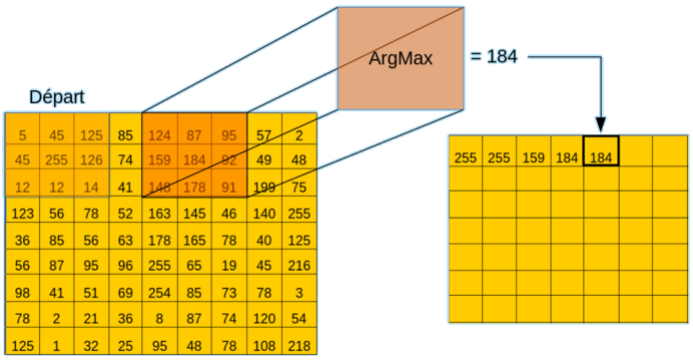


Figure 5 : Fonctionnement d'un neurone convolutif

2.1 Modèle de CONV-ReLu-Pooling

Les modèles les plus classiques de réseau neuronal convolutifs sont des empilements de CONV-ReLu-Pooling.

2.1.1 Le pooling

Le pooling est une opération qui remplace un carré de pixels de taille 2x2 par exemple (ou 3x3) par une unique valeur. Il existe plusieurs types de pooling : - Le « max pooling« , qui revient à prendre la valeur maximale du carré de pixels. Ce pooling est le plus utilisé car il est rapide à calculer (immédiat), et permet de simplifier efficacement l'image. - Le « mean pooling » (ou average pooling) : on calcule la somme de toutes les valeurs et on divise par le nombre de valeurs. - Le « sum pooling« , c'est la moyenne sans avoir divisé par le nombre de valeurs (on ne calcule que leur somme)

Il existe une grande différence entre le max et les autres formes de pooling. En effet, max-pooling est plus intéressant dans les cas particuliers comme la présence d'une arrête verticale et se comporte globalement de la même manière dans la reconnaissance de motifs dans les autres cas.

2.1.2 ReLu

La fonction ReLu dans le cas de réseau de neurone de convolution est intéressante, elle se définit comme ceci :

$f(x) = max(0, x)$

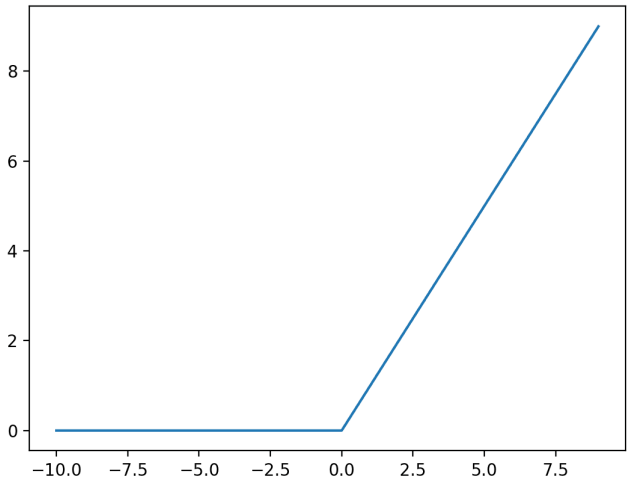


Figure 6 : Fonction ReLu

Plusieurs avantages offrent cette fonction :

- Peu couteux, elle supprime les valeurs négatives et donc facilite le temps d'exécution et possède les mêmes propriétés que les fonctions sigmoïdales comme la tanh.
- Nous avons vu que l'opération de convolution est un composé d'addition et/ou multiplication, la fonction ReLu supprime partiellement la linéarité et met en évidence les valeurs positives. En supprimant les valeurs négatives, elle permet de creuser l'écart entre deux caractéristiques de l'image.

2.1.3 CONV-ReLu-Pooling

L'empilement de Conv-ReLu-Pooling se traduit de la sorte :

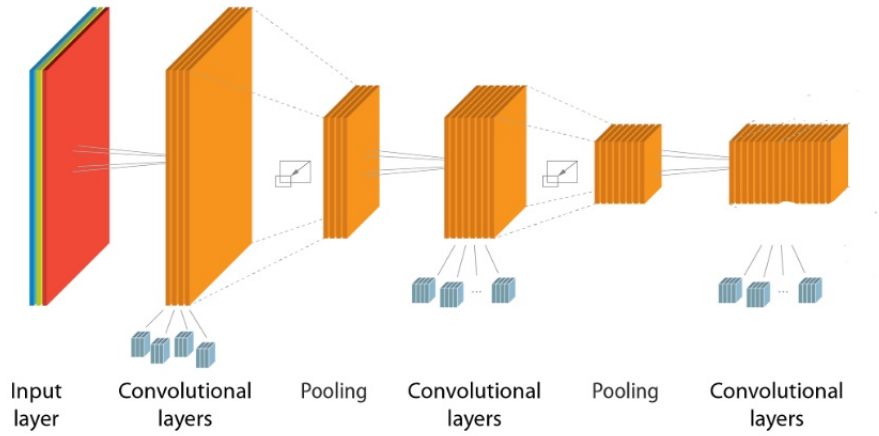


Figure 7 : Exemple d'empilement de Conv-ReLu-Pooling

Une première succession de couche de convolution avec une fonction d'activation ReLu, suivi d'un max pooling qui vient réduire la taille de l'image. A l'étape suivante, le nombres de couche de convolution est doublé. Cette étape est suivie de la même étape de max pooling. Ce processus est reproduit jusqu'à obtenir une taille d'image à la sortie où il n'est quasiment plus pertinent de reproduire ces étapes.

2.2 Le principale avantage du CNN au perceptron multicouche

Les couches Dense ont une approche globale là où les couches de convolutions ont une approche locale.
Cette approche locale permet aux couches de convolution d'extraire les motifs, les tendances d'une donnée.
Pour les réseaux de neurones simples, nous avons vu qu'il existe un problème pouvant engendrer une "explosion" du nombre de paramètres. Cependant, le CNN réduit drastiquement le nombre de paramètre en sortie.
Pour une unique couche de convolution de taille kx*ky :

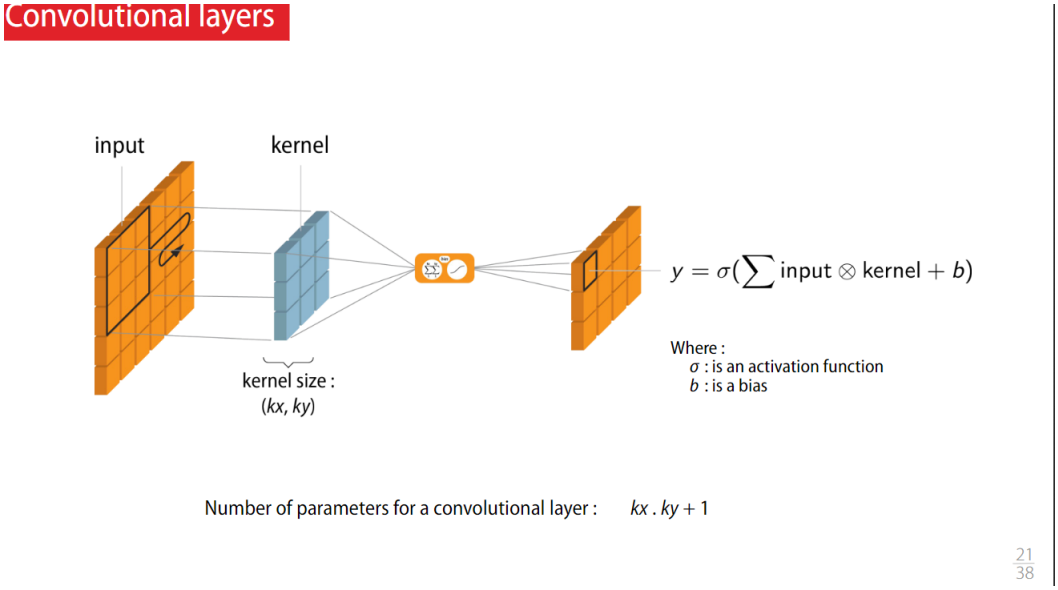


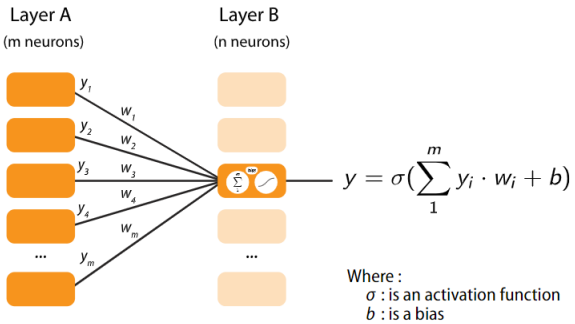
Figure 8 : Nombre de paramètres pour une couche de convolution

En prenant par exemple, pour la couche de convolution une taille 3x3, nous obtenons ainsi 10 paramètres (en oubliant pas le biais). En comparaisons, pour une couche dense

nous avons pour le nombre de paramètres : nombre de neurones * (taille de l'entrée + 1).

Number of parameters

For a fully connected layer :



Number of parameters for a DNN layer : $n\,(m+1)$

27
38

Figure 9 : Nombre de paramètres pour une couche de neurones

Note : Sigma désigne la fonction d'activation

3 Expérimentation

3.1 Problématique

Nous avons à disposition des textures browniennes que nous avons générée via le package PyAFBF de Fréderic Richard. Elles sont générées selon une certaine orientation.

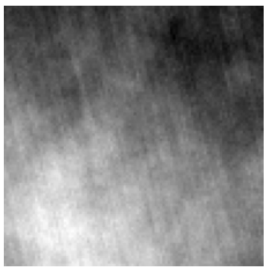


Figure 11 : Exemple d'une texture

L'idée de l'approche expérimental est d'approximer la fonction f qui lie l'image et l'orientation de la texture brownienne.
L'objectif va être de présenter un réseau de neurone que nous allons entraîner afin qu'il soit capable de donner une bonne approximation de l'orientation.

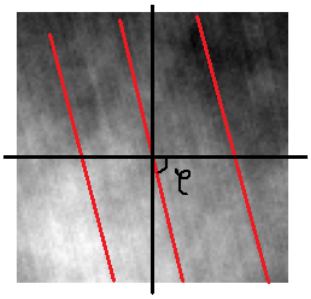


Figure 12 : Présentation de l'orientation de l'angle

En premier lieu, nous avons essayé d'étudier cette problématique dans le cas d'une régression, ou l'objectif était d'approcher la valeur de l'angle $y \in [-\pi/2, \pi/2]$. Nous avons essayé différents schémas de réseaux mais aucune d'entre-elle nous a fourni des résultats exploitables.

Nous nous sommes donc ramenés à un problème de classification, rendant l'étude plus simple. En effet, nous avons pu très vite dégager des architectures fonctionnelles. Nous allons présenter l'une d'entre-elle, que nous étudierons par la suite.

3.2 Jeu de donnée

On dispose d'une famille de 40 000 images, de 30 pixels par 30, X = $(x_1, ...x_{40000})$ $\forall i, x_i \in \mathcal{M}_{30,30}$, que nous stockerons dans la variable Images.

Nous disposons aussi de 1000 images qui serviront de validation durant l'apprentissage Que nous stockerons dans Images_val, et de 1000 autres sur lesquelles nous feront des prédictions pour tester notre réseau, Images_test.

Après importation de ces images nous les avons normalisés. Nous avons utilisé la Normalisation par lots : On a calculé la moyenne et la variance de chaque caractéristique du lot , puis la moyenne est soustraite et chaque caractéristique est divisée par l'écart type du lot. Cela stabilise le processus d'apprentissage et réduit considérablement le nombre d'époques nécessaires pour entraîner les réseaux profonds, permettant au réseau de s'entraîner plus rapidement.

Aussi, pour chacune de ces images, nous avons à disposition l'angle d'orientation, stockés dans les variables Lphi, Lphi_test, Lphi_val.
Afin de se ramener à un problème de classification, nous avons procéder à une répartition des phi en k classe, Cphi, Cphi_test, Cphi_val, avec k pouvant prendre les valeurs 2, 5, 10, 20 et 30
Nous avons programmer un algorithme qui classe les angles des images de notre base de données, de l'ensemble de validation ainsi que l'ensemble de valeur test. Le programme va catégoriser chaque angle selon s'en appartenance à un intervalle. Dans le cas de deux classes, si phi appartient à l'intervalle $[-\pi/2, 0]$ alors il sera placé dans la classe 0, sinon dans la classe 1.

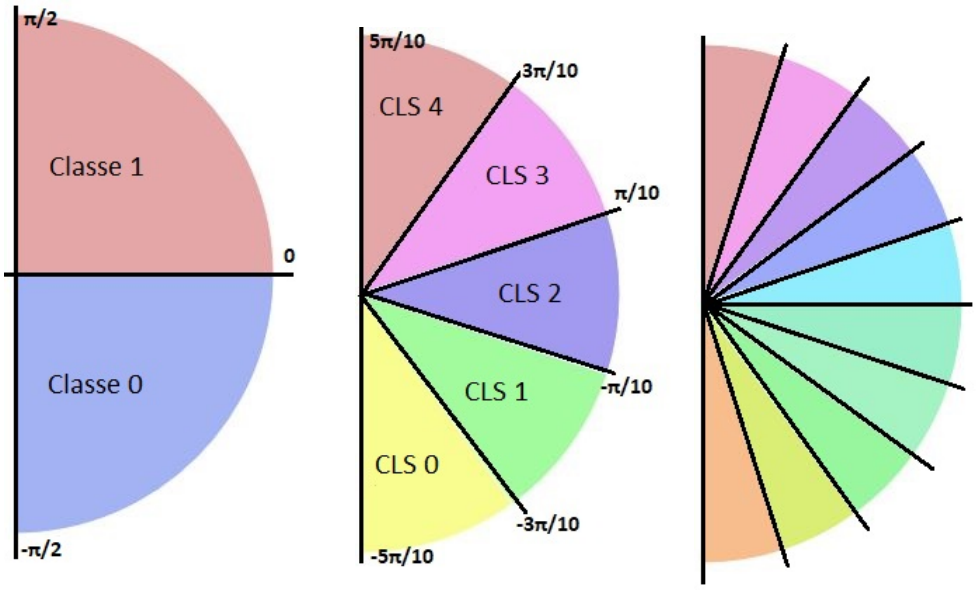


Figure 13 : Classification en 2, 5 et 10 classes

3.3 Construction du réseau de neurone avec Keras

Nous avons donc fait le choix d'étudier un modèle de CNN que nous avons présenté précédemment. Nous avons utilisé Keras tout au long de notre projet.

```
model = keras.models.Sequential()

model.add(keras.layers.Input((30,30,1)))

model.add(keras.layers.Conv2D(16,(3,3),activation="relu"))
model.add(keras.layers.MaxPool2D((2,2)))

model.add(keras.layers.Conv2D(32,(3,3),activation="relu"))
model.add(keras.layers.MaxPool2D((2,2)))

model.add(keras.layers.Conv2D(64,(3,3),activation="relu"))
model.add(keras.layers.MaxPool2D((2,2)))



model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(200,activation='relu'))
model.add(keras.layers.Dense(100,activation='relu'))
model.add(keras.layers.Dense(50,activation='relu'))
model.add(keras.layers.Dense(k,activation='softmax'))
```

La commande Sequential() de Keras permet d'initialiser un réseau de neurone.

```
model = keras.models.Sequential()
```

Par la suite, notre modèle se décomposera en deux parties, une première partie convolutionnel et d'une seconde partie constituée d'un perceptron multicouche.

3.3.1 Convolutional Neuronal Network

La commande layers est ce qui va nous permettre de rajouter les couches successivement. Ici, en l'occurrence, il s'agit de la couche Input() qui correspond à la couche d'entrée.

```
model.add(keras.layers.Input((30,30,1)))
```

Notre réseau va prendre en entrée des matrices de taille 30x30x1, il s'agit donc d'une matrice 30 par 30 qui va contenir comme information un dégradé de gris. Cette précision est importante car il est possible de mettre en entrée des images en RGB, où chaque pixel est défini par trois informations, un dégradé de rouge, de vert et de bleu.

La commande conv2D() permet de créer les couches de convolution.

```
model.add(keras.layers.Conv2D(16,(3,3),activation="relu"))
```

Ici nous avons donc 16 couches de convolution, avec un filtre de taille 3x3, et en fonction d'activation, la ReLU.

La commande MaxPool2D() est ce qui va nous permettre de réaliser un MaxPooling.

```
model.add(keras.layers.MaxPool2D((2,2)))
```

Nous reproduisons donc la structure d'empilement de CONV-ReLu-Pooling que nous avons présenté précédemment ...

```
model.add(keras.layers.Conv2D(16,(3,3),activation="relu"))
model.add(keras.layers.MaxPool2D((2,2)))

model.add(keras.layers.Conv2D(32,(3,3),activation="relu"))
model.add(keras.layers.MaxPool2D((2,2)))

model.add(keras.layers.Conv2D(64,(3,3),activation="relu"))
model.add(keras.layers.MaxPool2D((2,2)))
```

3.3.2 Perceptron Multicouche

Ensuite, nous utilisons la commande flatten afin de se ramener à un problème unidimensionnel, puis à l'aide de la commande dense() nous ajoutons des couches de réseaux : 200 pour la première couche caché, 100 pour la deuxième couche caché et 50 pour la troisième couche cachée. Nous utilisons la fonction d'activation "relu" grâce à l'argument activation. Pour finir, notre dernière couche dense a k neurones, avec k désignant le nombre de classe pour lequel nous souhaitons travailler. Nous utilisons la fonction d'activation "softmax", qui associe à chaque classe une probabilité, et choisis la valeur probabiliste la plus élevé pour classifier l'image dans une classe.

```
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(200,activation='relu'))
model.add(keras.layers.Dense(100,activation='relu'))
model.add(keras.layers.Dense(50,activation='relu'))
model.add(keras.layers.Dense(k,activation='softmax'))
```

L'architecture de notre réseau pour une problématique à 2 classes par exemple donnerait ceci :

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 16)        160

max_pooling2d (MaxPooling2D  (None, 14, 14, 16)       0
)

conv2d_1 (Conv2D)            (None, 12, 12, 32)        4640

max_pooling2d_1 (MaxPooling  (None, 6, 6, 32)         0
2D)

conv2d_2 (Conv2D)            (None, 4, 4, 64)          18496

max_pooling2d_2 (MaxPooling  (None, 2, 2, 64)         0
2D)

flatten (Flatten)            (None, 256)               0

dense (Dense)                (None, 200)               51400

dense_1 (Dense)              (None, 100)               20100

dense_2 (Dense)              (None, 50)                5050

dense_3 (Dense)              (None, 2)                 102

=================================================================
Total params: 99,948
```

Notre modèle possède environs 100 000 paramètres, pour une base d'image de 40 000, ce qui nous semble plutôt cohérent pour la validation du modèle.

3.4 Apprentissage

Avons de procéder à l'apprentissage de notre réseau de neurone, il nous faut configurer le modèle. La fonction **compile()** nous permet de définir la méthode d'optimisation, la fonction de perte et la métrique.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics='accuracy')
```

Pour la méthode d'optimisation, nous utilisons la méthode **Adam**. C'est une méthode d'optimisation, qui est une extension de l'algorithme de gradient stochastique. Le principe est le même que pour l'algorithme de gradient, cependant nous mélangeons le jeu de donnée au début de l'algorithme.
Nous utilisons en fonction de perte l'entropie croisé.
Et pour évaluer l'erreur de notre modèle nous utiliserons la métrique "accuracy", qui donne le pourcentage de prédiction valide sur nos données de validation.

```
model.fit(  Images, Cphi,
            batch_size=100,
            epochs=20,
            validation_data=(Images_test,Cphi_test))
```

Avec la fonction **fit()**, nous entraînons notre modèle avec notre base de données et notre vecteur des classes de phi.
L'argument "batch-size" est le nombre d'exemple donnée au modèle en une itération. Le batch-size va être assez déterminant lors de l'apprentissage de notre modèle. Un petit batch-size va rendre l'apprentissage plus précis mais moins rapide. Un plus gros batch-size le rendra plus rapide mais potentiellement moins précis. Un batch-size trop faible ou trop élevé peut entrainer des défauts d'apprentissage. Nous avons utilisé un batch-size de 50 pour l'apprentissage avec 2, 5 et 10 classes. Mais nous avons dû augmenter notre batch-size à 100 pour l'apprentissage avec 20 et 30 classes.
L'argument "epochs" est le nombre de fois que l'on entraîne notre réseau de neurone avec le même ensemble de donnée.

3.5 Prédiction

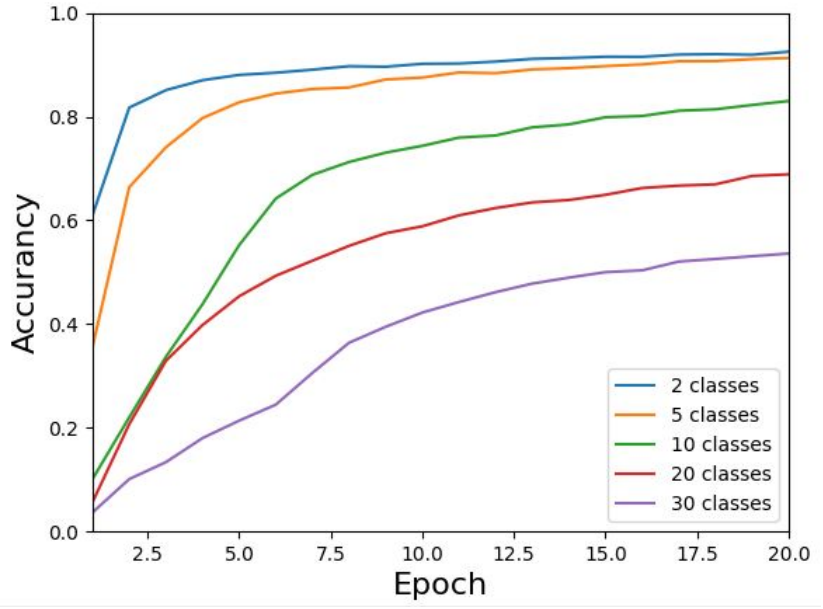


Figure 14 : Evolution de l'Accuracy en fonction de l'epoch pour 2,5 et 10 classes

L'accuracy est satisfaisante pour une classification avec 2 classes, 5 classes voir 10 classes. En effet, pour les cas de 2 et 5 classes, on atteint une précision de 90% au bout de 20 epochs.
Pour le cas de 10 classes, la courbe dépasse les 80% de précision. On peut estimer qu'en augmentant le nombre d'epoch, on peut atteindre sans trop de difficulté les 90% de précisions.
Elle commence à être plus faible lorsque l'on augmente considérablement le nombre de classe, jusqu'à ne pas dépasser les 50% de précision pour une classification en 30 classes.
On peut ainsi conclure que la précision radiale que nous propose le modèle est de $\pi/10$.

On peut aussi vérifier la part de précision lorsque le modèle se trompe de prédiction.

442.0	56.0
59.0	443.0

166.0	6.0	1.0	1.0	13.0
11.0	193.0	15.0	6.0	4.0
1.0	11.0	170.0	3.0	2.0
1.0	5.0	5.0	177.0	6.0
16.0	5.0	2.0	11.0	169.0

59.0	3.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	3.0
8.0	82.0	7.0	0.0	0.0	2.0	0.0	0.0	1.0	1.0
2.0	15.0	95.0	9.0	0.0	0.0	1.0	0.0	0.0	0.0
0.0	1.0	8.0	84.0	9.0	2.0	0.0	0.0	3.0	2.0
1.0	0.0	0.0	8.0	60.0	13.0	1.0	0.0	0.0	1.0
1.0	1.0	0.0	4.0	16.0	76.0	8.0	0.0	0.0	3.0
1.0	0.0	2.0	1.0	0.0	13.0	79.0	4.0	2.0	1.0
0.0	0.0	0.0	0.0	0.0	0.0	17.0	77.0	7.0	1.0
2.0	3.0	1.0	0.0	0.0	0.0	1.0	9.0	74.0	5.0
16.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	8.0	82.0

Figure 15 : Répartition des prédictions selon leurs vrais valeurs

Sur les colonnes, nous avons la répartitions des valeurs prédites pour une certaine classe. Par exemple, pour le cas de la classification en 10 classes, on a 113 éléments appartenant à la classe 2. On remarque qu'il y a 95 bonnes prédictions, et sur les 18 restants, il y en a 15 qui sont répartis sur les classes adjacentes et donc seulement 8 répartis sur les 7 autres classes. Donc même lorsque le modèle se trompe, il y a une certaine part de précision.

Conclusion

En conclusion, nous avons présentés les outils mathématiques essentiels à la compréhension de notre démarche d'expérimentation : le perceptron simple et multicouche, les réseaux de neurones convolutifs.

Pour la partie expérimentation, nous notons quelques résultats intéressants. D'une part, l'utilisation de perceptron multicouche pour un problème de détection d'angle n'est pas très concluante. Nous avons pu le confirmer sur Keras avec une explosion des paramètres, mais aucun apprentissage des différents modèles testés. Nous nous sommes alors intéressés au couplage de couche convolutive avec un perceptron multicouche, mais sommes tombés sur de mauvais résultats dans le cas d'une régression. Nous pensons que nous aurions pu répondre à cette problématique en ajoutant dans la partie dense plusieurs couches de neurones pour approcher bien plus précisément la fonction qui lie l'image en entrée et l'angle en sortie (propriété d'approximateur universel du réseau de neurone), mais cela aurait nécessité une base de donnée bien plus grande. Ainsi, nous sommes alors passé à un modèle de classification et avons obtenus de bon résultats. Nous avons obtenu une précision radial de nos prédiction de $\pi/10$. Nous aurions peut-être obtenu une meilleur précision avec un modèle un peu plus complexe, mais surtout avec une bien plus grande base de donnée. Nous aurions pu pour se faire augmenter artificiellement notre base de donnée, via des opérations assez simple telles des rotations.

La conclusion qui vient naturellement après la partie expérience sur python, est qu'un modèle de classification pour la détection d'angle est moins complexe qu'un modèle dans le cas continu. L'utilisateur souhaitant réguler le nombre de paramètres et possédant une base de données limitée, se tournera vers un modèle de classification.

Bibliographie

[1] "Gradient based learning applied to document recognition" Yann Le Cun,Léon Boutou, Yoshua Bengio, Patrick Haffner, 1998

[2] https://fjprichard.github.io/PyAFBF/, Frédéric Richard

[3] https://github.com/fjprichard/PyAFBF, Frédéric Richard

[4] https://en.wikipedia.org/wiki/Convolutional$_n eural_n etwork$

[5] https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[6] https://www.natural-solutions.eu/blog/la-reconnaissance-dimage-avec-les-rseaux-de-neurones-convolutifs

[7] https://penseeartificielle.fr/focus-reseau-neurones-convolutifs/

[8] https://keras.io/

[9] https://elitedatascience.com/keras-tutorial-deep-learning-in-python

[10] https://openclassrooms.com/fr/courses/4470531-classez-et-segmentez-des-donnees-visuelles/5082166-quest-ce-quun-reseau-de-neurones-convolutif-ou-cnn

[11] https://stanford.edu/ shervine/l/fr/teaching/cs-230/pense-bete-reseaux-neurones-convolutionnels

[12] https://deepnote.com/home

[13] https://www.youtube.com/watch?v=581X9wsnWJsab$_c hannel = CNRS-FormationFIDLE$