Détection de feuilles à l'aide d'un perceptron multicouche

Alexandre Koenig
3ème Année ECS

Romane Marchal
3ème Année ECS

Elie Dutheil
3ème Année ECS

Abstract

Le problème abordé dans ce rapport est la classification de feuilles d'arbres. Pour élaborer le classifieur, nous nous appuyons sur une base de données d'images ainsi qu'un fichier d'entraînement labélisé et un fichier de validation. Nous extrayons d'abord des données caractéristiques de l'image qui sont utilisées en entrée d'un perceptron à deux couches. Pour obtenir les labels prédits, nous utilisons dans ce réseau de neurones la fonction d'activation Softmax. Une fois le réseau mis en place, nous maximisons la log vraisemblance afin de régler les paramètres de notre modèle. Pour ce faire, nous utilisons la méthode de descente du gradient afin d'atteindre un minimum local de la fonction de coût de notre réseau et l'algorithme de rétro-propagation du gradient pour ajuster les paramètres du modèle. Nous faisons ensuite varier les différents hyper-paramètres du perceptron afin d'améliorer nos résultats (learning rate, nombre d'itérations, nombre de couches du réseau, nombre de neurones par couche cachée). Nous mettrons enfin en place une validation croisée compte tenu de l'échantillon restreint. Nous obtenons finalement une erreur de validation de 6%.

1. Introduction

La classification des feuilles est un sujet intéressant dans le domaine de la botanique. Il existe en effet environ un demi-million d'espèces de plantes dans le monde et leur classification a posé de nombreux problèmes notamment en raison de la création de doublons C'est un sujet sur lequel on peut utiliser les avancées dans le domaine du Deep Learning et de la vision par ordinateur.

Ce projet a été proposé sur le site www.kaggle.com. Nous utilisons les données mises à disposition sur le site à savoir :

- Un set d'images de feuilles normalisées (la feuille est blanche sur un fond noir)
- Un fichier CSV d'entrainement comportant d'une part les features proposés par Kaggle et les labels de chaque feuille des labels
- Un fichier CSV de test sous le même format

Pour répondre au problème nous avons employé une démarche en deux étapes :

- Etape 1 : extraction des caractéristiques de chaque image au moyen d'un détecteur SIFT
- Etape 2 : Utilisation d'un perceptron à deux couches avec en entrée les caractéristiques des feuilles et en sortie les catégories

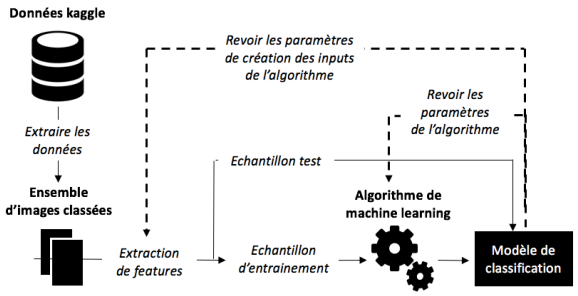

Figure 1: Description du processus

2. Contexte

Le projet a été proposé comme challenge sur Kaggle, ce qui nous a permis d'avoir accès immédiatement à une quantité de données satisfaisante. De plus, nous avons eu accès à certaines soumissions pour inspirer notre étude.

Les données auxquelles nous avons eu accès restent néanmoins assez limitées : le fichier CSV d'entrainement contient 990 échantillons pour 99 classes.

A propos des méthodes utilisées par les participants au challenge, la première chose que nous avons pu remarquer est qu'une partie des soumissions n'utilisaient pas le Deep Learning, mais faisaient simplement appel à des fonctions de classification comme par exemple KNN, RandomTree ou Adaboost de la librairie Scikit Learn.

Nous nous sommes donc renseignés sur les méthodes classiques de création de réseaux de neurones, afin d'en construire un qui soit adapté à notre problème, notamment en analysant l'effet sur la performance de la variation de différents paramètres. Nous allons détailler cette approche dans la suite de ce document.

3. Approche

Dans ce document nous avons décidé d'utiliser comme algorithme de « *machine learning* » un perceptron multicouche. Nous allons décrire dans cette partie son fonctionnement.

L'entrée de notre modèle est un ensemble de vecteurs, chaque vecteur représentant les « *features* » d'une image donnée (X_1 à X_n dans le schéma ci-dessous)

A partir de ces vecteurs, nous calculons la couche cachée puis la sortie de notre algorithme par combinaison linéaires du vecteur d'entrée.

Par exemple, dans la couche cachée nous aurons :

$$couche\_cachee_j = (\sum_{i=1}^{n} W_{1\,ij} x_i) + b_{1\,j}$$

Pour obtenir le vecteur de sortie y', nous appliquons à la sortie la fonction d'activation « *softmax* » :

$$z_j = (\sum_{i=1}^{n} W_{2\,ij} * couche_{cachee_i}) + b_{2\,j}$$

$$y_i' = softmax(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Le but de notre algorithme sera de déterminer les poids W_{ij} des matrices de poids 1et 2 et les biais b_j permettant de réduire au maximum l'erreur de notre modèle : la différence entre le vecteur prédit Y' et le vecteur de labels Y. Notre objectif est donc de maximiser la log vraisemblance que l'on peut exprimer sous cette forme :

$$\log(softmax(z)_i)$$

Cette recherche de maximum n'est pas simple, nous allons donc utiliser la méthode appelée descente de gradient qu'il est possible d'illustrer par la figure suivante et dont nous allons détailler le principe par la suite.

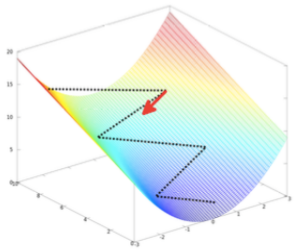

Figure 2: Illustration de la méthode de la descente de gradient

Nous cherchons un extremum local de notre fonction coût f. Le gradient de cette fonction va nous permettre de le trouver :

$$\nabla_x f(x) = [\frac{\partial f}{\partial x1}(x) \dots \frac{\partial f}{\partial x1}(x)]$$

Nous cherchons un point critique à gradient nul. La dérivée dans la direction unitaire de la pente u est la pente de f dans cette direction. Nous minimisons alors itérativement :

$$x_{t+1} = x_t - \varepsilon \nabla_x f(x)$$

On définit un taux d'apprentissage ε

Cette méthode va nous permettre de savoir comment modifier nos paramètres :

$$W = W - \varepsilon \nabla_w f$$

Afin d'appliquer cette méthode, il nous faut donc le gradient de la fonction de coût pour chaque matrice de poids. Nous utilisons pour les calculer, la méthode de rétro-propagation du gradient :

- La passe directe de la chaine permet d'estimer la sortie
- La fonction de coût permet d'évaluer l'erreur
- La rétro-propagation permet d'évaluer les gradients et les propager en arrière afin de modifier les paramètres dans le but de réduire l'erreur

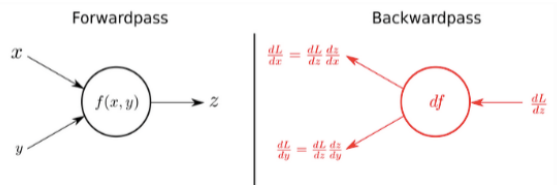

Figure 3: Calcul des sorties dans un premier temps puis calcul des gradients en retour

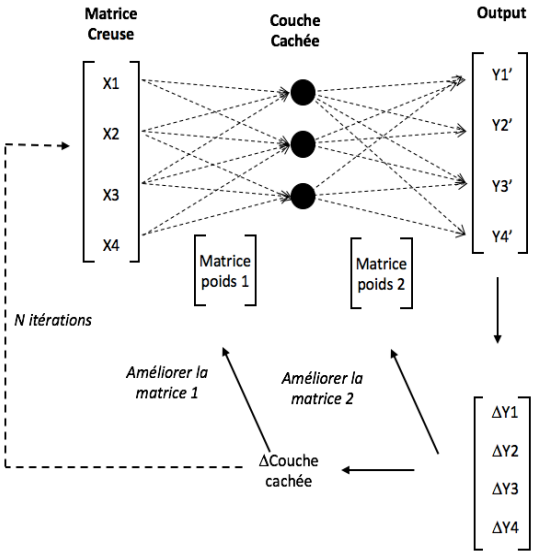

Figure 4: Schéma global de l'algorithme

On répète ensuite N fois cette opération de manière à minimiser et d'attendre un maximum de vraisemblance.

4. Expériences

Pour réaliser nos expériences, nous nous sommes appuyés sur une base de données contenant environ 1500 images normalisées de feuilles : chaque feuille est affichée en blanc sur un fond noir.

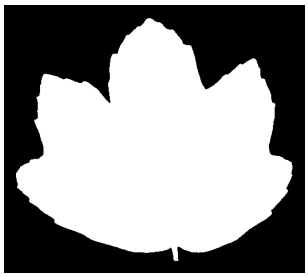

Figure 5: Exemple d'image de feuille disponible dans le set fourni par Kaggle.

Nous avons également utilisé un fichier de *« train »* contenant des labels et un fichier de *« test »* non labélisé. Pour chaque feuille, les données fournies en entrée du perceptron sont des *« features »* extraites des images et divisées en 3 catégories : *« margin »*, *« shape »* et *« texture »* (chaque catégorie contient 64 données). Ce fichier *« train »* correspond au fichier train-kaggle.csv sur le git de notre projet et non aux fichiers de *« features »* que nous avons extraits nous-même (train-orb.csv et train-sift.csv)

Les expériences ont été réalisées en faisant varier les paramètres du modèle et/ou les entrées puis en traitant les résultats obtenus sur Excel afin de pouvoir générer des courbes. La mesure de performance utilisée dans chaque expérience est le taux de justesse (noté *« accuracy »*).

4.1 Nombre d'itérations

La première expérience que nous avons menée est l'observation de l'évolution de l'*« accuracy »* du classifieur en fonction du nombre d'itérations de l'algorithme. Les résultats sur les ensembles de *« train »* et de *« test »* sont présentés ci-dessous.

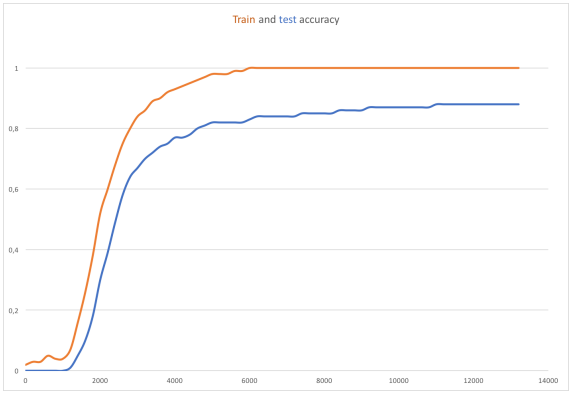

Figure 6: Evolution de l' « accuracy » en fonction du nombre d'itérations pour les ensembles de « train » et de « test » avec un « learning rate » de 0.001

On remarque en premier lieu qu'après 8000 itérations une asymptote est atteinte pour les deux ensembles. Par la suite, nous fixerons donc le nombre d'itérations à 8000 pour réaliser les autres expériences.

L'ensemble de train arrive à une erreur nulle après un certain nombre d'itérations ce qui est attendu (le modèle s'adapte à cet ensemble). En revanche, le modèle plafonne à environ 85% de bonnes réponses quel que soit le nombre d'itérations. Nous allons essayer par la suite d'améliorer cette performance en modifiant les paramètres du modèle.

4.2 *« Learning rate »*

Un premier paramètre que nous avons fait varier pour étudier son incidence sur les résultats est *le « learning rate »*.

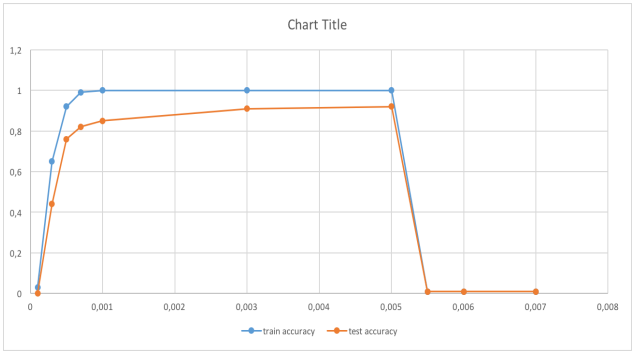

Figure 7: Evolution de l' « accuracy » en fonction du « learning rate » pour 8000 itérations.

De manière générale, les performances s'améliorent avec le *« learning rate »*. On remarque néanmoins un seuil au-delà duquel les performances chutent.

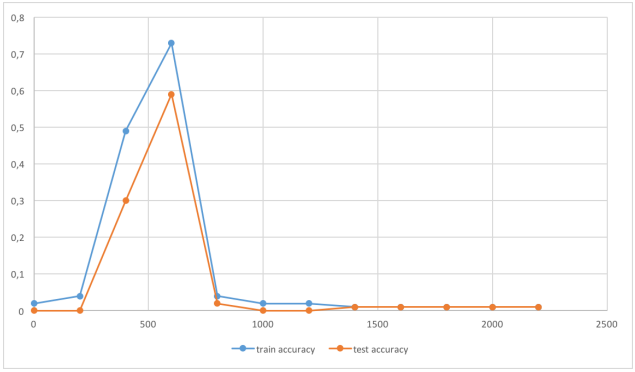

Figure 8: Evolution de l' « accuracy » en fonction du nombre d'itérations avec un haut « learning rate » à 0.0055

La figure ci-dessus montre que pour un *« learning rate »* élevé, le modèle s'améliore lors des premières itérations mais ne fonctionne plus après environ 600 itérations. Cela s'explique par le fait qu'avec un trop grand *« learning rate »* le minimum optimal est manqué par l'algorithme qui va se stabiliser vers un autre minimum local. On retrouve ce résultat sur la figure 3 où les performances chutent pour un *« learning rate »* trop haut.

4.3 Choix des *« features »* en entrée

Nous nous sommes ensuite intéressé à l'impact de chacune des catégories de *« features »* sur les performances de notre modèle. Les courbes ci-dessous représentent l'évolution du taux de précision avec les trois catégories de *« features »* en entrée du modèle et pour chacune des trois catégories seules.

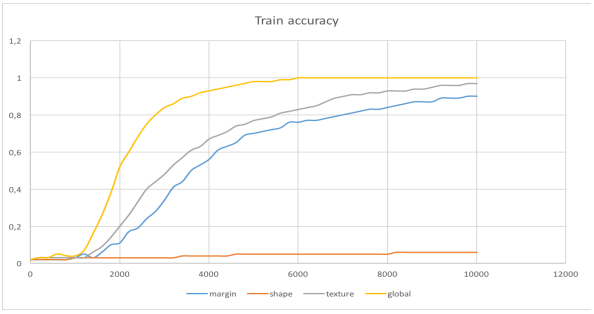

Figure 9: Evolution de l' « accuracy » en fonction du nombre d'itérations avec différentes entrées pour l'ensemble de « train »

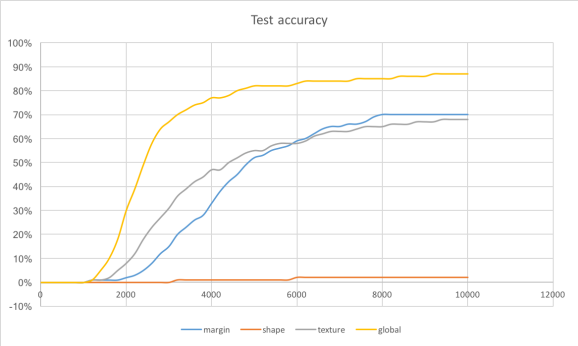

Figure 10: Evolution de l' « accuracy » en fonction du nombre d'itérations avec différentes entrées pour l'ensemble de « test»

La conclusion sur les deux ensembles est la même : c'est la combinaison des trois catégories de *« features »* qui donne les meilleurs résultats. On remarque également que *« shape »* est celle qui a l'impact le plus faible sur la performance. On pourrait donc envisager de s'en passer dans une démarche de simplification du modèle.

4.4 Nombre de neurones dans les couches cachées

Un autre paramètre du modèle qu'il est intéressant de faire évoluer est le nombre de neurones dans les deux couches intermédiaires du modèle. Nous avons fait varier ce nombre entre 1 et 300 pour observer son impact sur les performances (voir courbe ci-dessous)

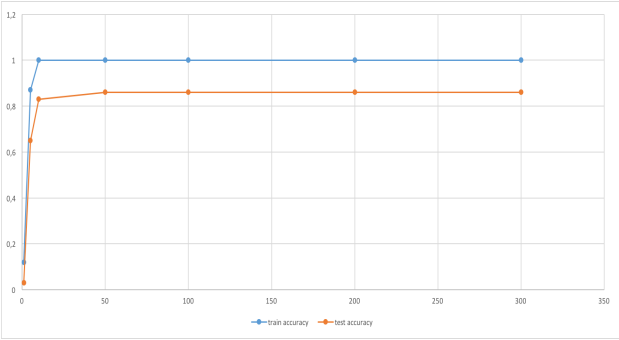

Figure 11: Evolution de l' « accuracy » en fonction du nombre de neurones dans les couches cachées à 8000 itérations. L'ensemble « train » est indiqué en bleu et « test » est en rouge

Pour un nombre de neurones trop faible, les résultats sont sensiblement moins bons (97% d'erreurs sur *« test »* pour un neurone seul par exemple) ce qui est attendu puisqu'il n'y a pas assez de paramètres pour refléter la complexité du problème.

En revanche, au-delà d'un seuil d'environ 10 neurones, les performances ne s'améliorent plus et le modèle est simplement plus coûteux en calculs. Il est donc inutile de dépasser ce seuil.

Il est aussi intéressant d'observer les résultats donnés par un perceptron à une couche pour se convaincre de l'intérêt d'utiliser deux couches :

Nombre de couches	1	2
Train accuracy	71%	100%
Test accuracy	45%	85%

Figure 12: Comparaison des performances d'un perceptron à une couche et à deux couches pour 8000 itérations

Comme nous pouvons l'observer dans la figure ci-dessus, le perceptron à une couche donne des résultats moins bons sur les deux ensembles que le modèle à deux couches que nous avons sélectionné.

4.5 Utilisation des spécificités du problème

Afin de réellement prendre en compte la spécificité de l'exemple sur lequel nous centrons notre étude, nous avons essayé de nous intéresser à la topologie de la classification par les biologistes. En effet, les feuilles sont classées par famille, par exemple la famille des Quercus qui contient environ 40 espèces. Nous avons voulu savoir si les erreurs commises par notre modèle étaient une confusion entre des feuilles de même famille. Si c'était le cas, nous aurions donc déjà un modèle très performant pour les familles de feuilles (qui ont un sens en termes biologiques).

La figure ci-dessous indique les performances du modèle en fonction du nombre d'itérations dans le cas où les espèces de feuilles sont rapportées à leur familles ou non.

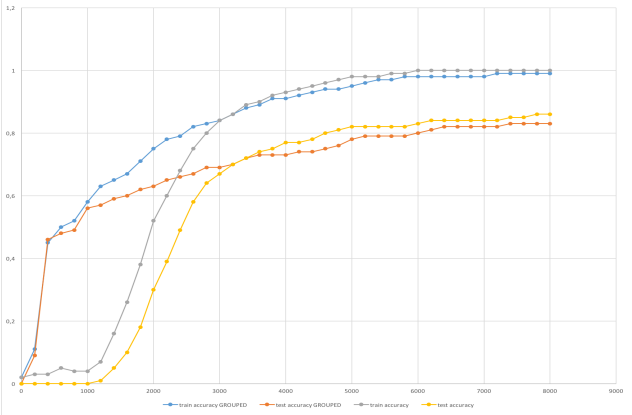

Figure 13: Evolution de l' « accuracy » en fonction du nombre d'itérations dans le cas où les feuilles sont groupées (courbe bleue et rouge) par famille et dans le cas de base (courbes grise et jaune) sur les ensembles « train » et « test »

Contrairement à ce nous attendions, on remarque le taux d'erreur sur les deux ensembles est plus important si on regroupe les espèces en familles. C'est donc une piste que nous avons abandonnée.

4.6 Cross validation

Pour compléter l'étude nous avons voulu trouver le meilleur ensemble de *« train »* possible dans nos données disponibles à l'aide de la technique de cross validation.

Nous avons donc séparé l'ensemble de *« train »* initial (labélisé) en 10 sous-ensembles (plus un de validation) puis nous avons utilisé chaque sous-ensemble successivement comme ensemble de *« train »* (les sous-ensembles restant étant utilisés comme *« test »*). Les résultats sont reportés dans le tableau suivant.

batch	1	2	3	4	5	6	7	8	9	10
Train accuracy	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Test accuracy	88%	90%	84%	85%	90%	88%	89%	82%	93%	84%
Validation accuracy batch 9	94%									

Figure 14: « accuracy » pour les différents couples « train »/ « test » pour 8000 itérations

Pour le *« batch »* 9 on obtient de bons résultats sur l'ensemble de *« test »* qui sont confirmés (à 94%) sur l'ensemble de validation. C'est donc cette répartition qui nous a permis d'entraîner notre modèle final pour obtenir de bons résultats.

5. Conclusion

Pour conclure, nous avons tenté d'analyser l'influence de différents paramètres sur les performances de notre réseau de neurone. Cette analyse nous a permis d'obtenir des résultats que nous jugeons satisfaisants, notamment si nous prenons en compte la faible quantité d'échantillons donc nous disposions.

Cependant, nous avions conscience que pour obtenir un réseau de neurone avec une performance optimale nous aurions besoin d'effectuer une analyse beaucoup plus poussée. En effet, il est possible de conduire plusieurs centaines d'expériences sur un jeu de données en modifiant les paramètres que nous avons identifiés mais aussi la fonction d'activation, la méthode de résolution du maximum de vraisemblance…

La dernière piste d'amélioration que nous n'avons pas pu explorer est le traitement des données en entrée de notre réseau de neurones. Nous avons conscience de l'importance de cette étape car une forme adaptée pour les données que nous appliquons en entrée de notre réseau de neurone peut donner de biens meilleurs résultats.

Nous avons fait le choix de nous concentrer sur la construction du réseau, notamment en raison de la provenance des données : ce jeu provient d'un challenge Kaggle, ce qui nous laisse supposer qu'elles ont déjà été traitées.

Finalement, nous pouvons tout de même dire que l'application du Deep Learning à la reconnaissance de feuille est prometteuse en raison du nombre très élevé d'espèces d'arbres présentes sur Terre. De plus, ce problème n'étant pas relié à des problématiques vitales, les conséquences d'une erreur de classification sont assez faibles, ce qui nous permet de dire que notre modèle apporte une réelle valeur ajoutée pour la résolution de ce problème.

Références

[1] Challenge : https://www.kaggle.com/c/leaf-classification
[2] Kai Xuan Wei. A simple 2-layer neural network model : https://www.kaggle.com/vandermode/digit-recognizer/a-simple-2-layer-neural-network-model/notebook
[3] Pierre. Bag-of-words model with SIFT descriptor : https://www.kaggle.com/pierre54/leaf-classification/bag-of-words-model-with-sift-descriptors
[4] Hervé Le Borgne : Deep Learning / Apprentissage profond