

Université Paris-Saclay
Master 1 d'Informatique
Introduction à l'apprentissage automatique

RECONNAISSANCE D'IMAGE

AKHARAZ MAJID
BIGNET CAMILLE



01-05-2020

Contents

1	Introduction	1
2	Méthode du plus proches voisins.	4
2.0.1	Définition formelle de l'algorithme du plus proche voisin:	4
2.0.2	Implémentation :	5
3	Méthode des K plus proches voisins.	8
3.0.1	Définition formelle de l'algorithme des K plus proches voisins:	8
3.0.2	Stratégie de recherche pour le choix de K et de la Distance:	8
3.0.3	L'implémentation:	9
4	Conclusion.	11
5	Implémentation de l'article intitulé: An Analysis of Single Layer Networks in Unsupervised Feature Learning.	12
5.0.1	Définition des concepts :	12
5.0.2	Construction de notre modèle :	13
5.0.3	Résultat:	15
5.0.4	Problèmes rencontrés	17
5.0.5	Amélioration possible	17
6	Conclusion finale.	19

1. Introduction

Motivation: Classification d'images

La classification d'images est un problème central en vision par ordinateur, ayant de nombreuses applications concrètes. L'objectif est de construire un système ayant la capacité d'assigner à n'importe quelle image d'entrée une catégorie.

Evidemment un ordinateur ne peut pas voir une image comme un humain. Pour l'ordinateur une image couleur, est représentée par une matrice tridimensionnelle. Dans l'exemple ci dessous, le chien est représenté par une image de 64 pixels de hauteur et 64 pixels de largeur, sur lesquelles chaque pixels est représenté par 3 entiers, un entier pour le rouge un autre pour le vert et un dernier pour le bleu. Ainsi une image de 64×64 est, pour l'ordinateur, représenté par: $64 \times 64 \times 3 = 12288$ entiers. Notre objectif est de donner un sens à ces 12288 nombre de telle sorte que l'ordinateur puisse nous retourner la catégorie d'appartenance de l'image en entrée.

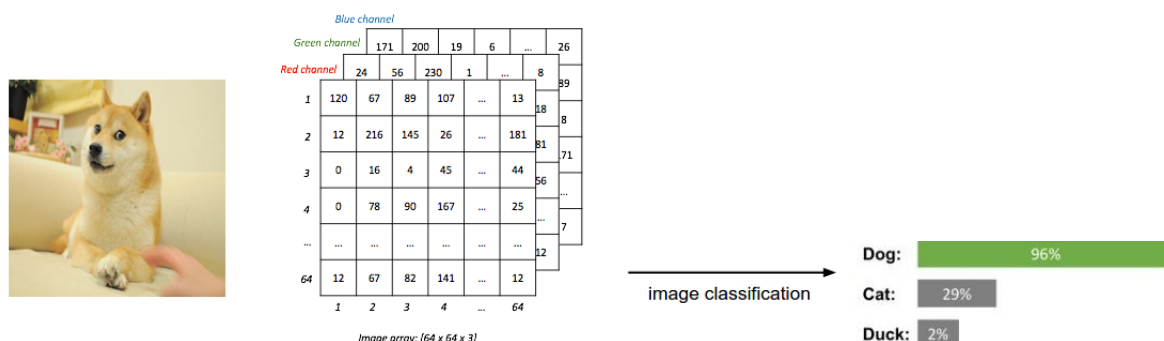


Figure 1.1: L'objectif est de prendre en entrée une image et de lui associer une "étiquette" (ou plusieurs étiquettes auxquelles on associe un probabilité de vraisemblance).

Difficulté :

Une telle tâche est triviale pour un humain, mais qu'en est-il pour un ordinateur ? Nous détaillons ici quelques pistes permettant de comprendre pourquoi l'élaboration d'un algorithme de classification d'image peut présenter de réelle difficulté.

- **Les différences d'un même groupe:** Des objets d'une même classe peuvent présenter des différences considérables.;
- **L'arrière plan de la photo:** Il peut arriver que l'arrière plan de la photo se confonde avec l'image;

- **Différence de luminosité:** La coloration des pixels peut considérablement changer en fonction de la luminosité.
- **La non complétude de l'objet sur l'image:** Il est possible que l'objet qui nous intéresse soit rogné sur l'image, qu'on puisse en voir qu'une partie.
- **Déformation:** L'objet d'intérêt peut être très souple, non rigide. Cela pourrait entraîner des déformations.
- **Différence de taille:** Un même objet peut représenter d'une taille différente.
- **Le point de vue:** Un objet de face et un objet de profil sont "vue" différemment pour un ordinateur.



Figure 1.2: De gauche à droite: Des images illustrant les problématiques de la luminosité, angle de vue, taille de l'image, différence d'objet du même groupe, déformation de l'objet, un arrière plan confondu avec l'image, et enfin une image dont l'objet est que partiellement apparent

Pourquoi attaquer ce problème avec l'apprentissage automatique:

Ainsi on se questionne sur savoir comment créer un algorithme qui pourrait classer chaque image dans la catégorie qui lui correspond sachant que pour l'ordinateur une image n'est qu'une liste de nombre?

La difficulté est qu'il nous ait impossible de répertorier exhaustivement les caractéristique d'un chat, d'une voiture, d'un chien le tout dans des contextes qui peuvent altérer ces caractéristiques.

Dès lors, nous allons donc fournir à l'ordinateur de nombreux exemples de chaque catégorie, puis développer des algorithmes d'apprentissage capable d'analyser « regarder » ses exemples et assimiler assez de connaissance pour apprendre qu'elle est l'apparence visuelle de chaque catégorie.

Cette approche est donc basée sur les données, puisqu'elle repose sur l'accumulation d'un grands nombre de données dit d'apprentissage.

Les données utilisées pour ce projet:

Dans le cadre de ce projet nous utiliserons un ensemble de donnée appelé **CIFAR-10**. Cette base de donnée comprends 60 000 petites images de dimension 32×32 . Chaque image est labelisé par une des 10 catégories(avion, voiture, oiseau, etc) présentent dans la base. Les 60 000 images sont partitionnées en deux catégories:

- 50 000 images d'entrainement.
- 10 000 images de testes.



Pour accomplir notre tache nous explorerons plusieurs méthodes, nous tenterons d'aborder des approches considéré comme étant de plus en plus sophistiqué, nous avons pour ambition de détailler ces différentes approches et de les expliquer autant que faire ce peut.

2. Méthode du plus proches voisins.

Dis moi qui est ton voisin je te dirais qui tu es.

Information: Les data étant trop volumineuses pour être traité par nos ordinateurs, nous avons choisis de faire "tourner" nos algorithmes sur des échantillons réduits. 5000 images d'entraînement et 500 de tests.

Cette méthode consiste à prendre successivement toutes nos images de test et à mesurer la dissimilarité avec chacune des images de notre ensemble d'entraînement. Ici pour notre ensemble de données, il s'agit de calculer la dissimilarité de chacune de nos 10 000 images, avec chacune des 50 000 images, on se rends tout de suite compte de deux choses. Cette méthode est facile à implémenter mais est très coûteuse en temps de calcul lorsque le nombre d'objet devient très grand.

Pour calculer cette dissimilarité, nous calculons la différence entre chaque pixel des matrices en question. Nous avons fait le choix de prendre comme distance la distance L^1 , formellement cette distance s'écrit comme suit, soit deux images représenté par deux vecteur X_1 et X_2 :

Distance L^1 :

$$d_1(X_1, X_2) = \sum_i |X_1^i - X_2^i|$$

2.0.1 Définition formelle de l'algorithme du plus proche voisin:

Soit $X = (x_1, \dots, x_N) \subset E$ un ensemble d'éléments d'un espace métrique quelconque, soit $(c(x_1), \dots, c(x_N))$ les classes associées à chacun des éléments de X . On note d la distance définie sur l'espace métrique (Un espace que l'on peut munir d'une norme) E . Soit x un élément à classer, on cherche à déterminer la classe \hat{c} associé à x . On définit x_i^* comme étant:

$$x_i^* = \arg \min_{i \in \{1, \dots, N\}} d(x_i, x)$$

Alors $\hat{c}(x) = c(x_i^*)$

test image					training image					pixel-wise absolute value differences				
56	32	10	18	-	10	20	24	17	=	46	12	14	1	→ 456
90	23	128	133		8	10	89	100		82	13	39	33	
24	26	178	200		12	16	178	170		12	10	0	30	
2	0	255	220		4	32	233	112		2	32	22	108	

Figure 2.1: Ici nous représentons un exemple simplifié (une seule couleur) de l'utilisation du calcul de dissimilarité par utilisation de la distance L^1 . Deux images sont soustraites pixel par pixel, puis nous additionner l'ensemble des soustractions, cet entier est la distance entre les deux images. Ainsi si deux images sont identiques la somme de leurs différences est zero. A contrario si les images sont très différentes le resultat sera grand.

2.0.2 Implémentation :

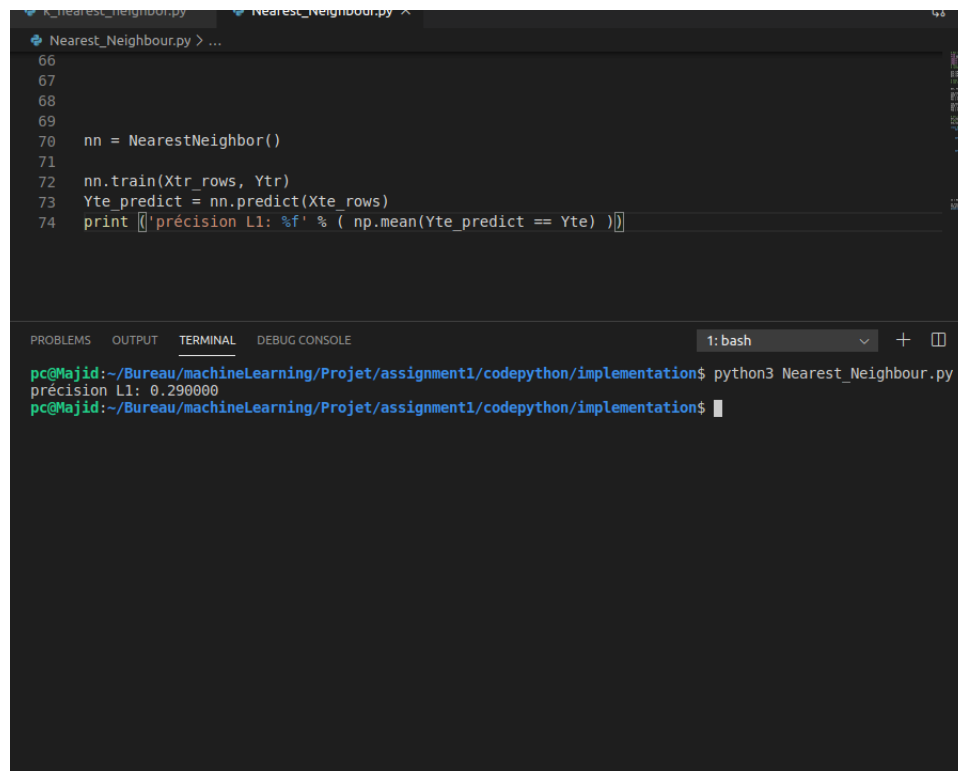
Dans un premier temps, nous chargeons Cifar10 avec la fonction `loadcifar10()` en mémoire dans 4 tableaux: Xtr est toutes les données d'entraînement ce tableau est de taille $(50000 \times 32 \times 32 \times 3)$. Ytr est un vecteur de taille 50 000 qui contient tous les labels de 0 à 10.

Puis avec la fonction **reshape** on redimensionne les tableaux de telle manière à représenter les images en lignes. Ce traitement des données étant fait, il ne reste plus qu'à entraîner notre modèle avec la fonction **train**, dans le cadre du plus proche voisin, l'entraînement consiste simplement à mettre en mémoire. Ensuite vient l'étape de prédiction avec la fonction **predict()**, cette fonction, pour chaque image test calcule sa distance avec toutes les images d'entraînement, et retiens la distance la plus petite distance ainsi que l'indice du label.

Nous évaluons l'efficacité de notre algorithme en calculant le ratio :

$$\text{Efficacité} = \frac{\text{Nombre de bonne prédiction}}{\text{Nombre total de prédiction}}$$

Notre algorithme naïf présente une précision de 29%. Les images toujours mal prédites sont celles avec les fonds de couleurs homogènes. On explique cela par le fait que les images ayant les mêmes nuances de couleur en arrière plan, minimisent la dissimilarité, et ceux même si notre objet d'intérêt est totalement différent.



```
Nearest_Neighbour.py > ...
66
67
68
69
70 nn = NearestNeighbor()
71
72 nn.train(Xtr_rows, Ytr)
73 Yte_predict = nn.predict(Xte_rows)
74 print('précision L1: %f' % ( np.mean(Yte_predict == Yte) ))

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE 1: bash
pc@Majid:~/Bureau/machineLearning/Projet/assignment1/codepython/implementation$ python3 Nearest_Neighbour.py
précision L1: 0.290000
pc@Majid:~/Bureau/machineLearning/Projet/assignment1/codepython/implementation$
```

Une telle méthode n'a pas de rempart contre les valeurs abberantes.
Nous essayons la distance L^2 pour avoir une idée du résultat:

Distance L^2 :

$$d_1(X_1, X_2) = \sqrt{\sum_i |X_1^i - X_2^i|^2}$$

Nous obtenons un résultat infieur à celui de L1 de 27%.

The image shows a code editor with two tabs: `k_nearest_neighbor.py` and `Nearest_Neighbour.py`. The `Nearest_Neighbour.py` tab is active, displaying the following Python code:

```
66
67
68
69
70 nn = NearestNeighbor()
71
72 nn.train(Xtr_rows, Ytr)
73 Yte_predict = nn.predict(Xte_rows)
74 print(['précision L2: %f' % ( np.mean(Yte_predict == Yte) )])
```

Below the code editor is a terminal window with tabs for `PROBLEMS`, `OUTPUT`, `TERMINAL`, and `DEBUG CONSOLE`. The `TERMINAL` tab is selected, showing the command `python3 Nearest_Neighbour.py` being executed. The output of the script is displayed in the terminal:

```
pc@Majid:~/Bureau/machineLearning/Projet/assignment1/codepython/implementation$ python3 Nearest_Neighbour.py
précision L2: 0.274000
pc@Majid:~/Bureau/machineLearning/Projet/assignment1/codepython/implementation$
```

3. Méthode des K plus proches voisins.

Dis moi qui sont tes voisins je te dirais qui tu es.

L'idée est assez proche de celle du plus proche voisin, sauf qu'au lieu de trouver l'image la plus proche, nous chercherons à trouver les **K** images les plus proches, puis parmi ces K images nous regardons la classe qui revient le plus souvent, c'est cette dernière que nous étiqueterons nous image test.

Ainsi lorsque $K = 1$, nous sommes exactement dans un problème du plus proche voisin.

3.0.1 Définition formelle de l'algorithme des K plus proches voisins:

Soit $X = (x_1, \dots, x_N) \subset E$ un ensemble d'éléments d'un espace métrique quelconque, soit $(c(x_1), \dots, c(x_N))$ les classes associées à chacun des éléments de X. On note d la distance définie sur l'espace métrique (Un espace que l'on peut munir d'une norme(distance)) E . $\omega(x, y)$ est une fonction strictement positive mesurant la ressemblance entre x et y . Soit x un élément à classer, on cherche à déterminer la classe $\hat{c}(x)$ associée à x . On définit l'ensemble S_k^* incluant les k -plus proches voisins de x , cet ensemble vérifie:

$$\text{Card} S_k^* = k \text{ et } \max_{y \in S_k^*} d(y, x) \leq \min_{y \in X - S_k^*} d(y, x)$$

On calcule les occurrences $f(i)$ de chaque classe i dans l'ensemble S_k^* :

$$f(i) = \sum_{y \in S_k^*} \omega(x, y) \mathbb{1}_{\{c(y)=i\}}$$

On assigne alors à x la classe $c(x)$ choisie dans l'ensemble:

$$\hat{c}(x) \in \arg \max_{i \in \mathbb{N}} f(i)$$

L'une des premières questions que l'on se pose lors de l'implémentation de cet algorithme est le choix de **K**.

3.0.2 Stratégie de recherche pour le choix de K et de la Distance:

Le nombre K et la distance sont des **Hypeparamètres**. Pour choisir le bon K et la bonne distance, nous avons choisit une stratégie naïve qui consiste à essayer plusieurs distance et plusieurs K et garder ceux qui offrent les meilleurs résultats.

Le problème qui s'est posé à nous est que nous ne pouvions pas utiliser l'ensemble de test pour ajuster nos paramètres, le risque de cela aurait été de paramétrer notre en algorithme en fonction du jeu de test que nous avons, cela rendre notre algorithme totalement innéficace sur un autre jeu de test.

La stratégie mise en place consiste à diviser nos données d'entraînements, nous gardons 49 000 images pour l'entraînement et 1000 pour la validation des paramètres, en utilisant la méthode de cross-validation.

3.0.3 L'implémentation:

Nous procédons en deux étapes:

- On calcule toutes les distances entre les données de test et celle d'entraînement;
- Ayant ces distances, pour chaque donnée de test on trouve les K plus proches images et on effectue la sélection du label en fonction du nombre de label le plus présent parmi les K;

```
pc@mlj: /home/pc/cv231/assignment1/cv231n/classifiers$ python3 k_nearest_neighbor.py
(5000, 3072) (500, 3072)
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.257000
k = 3, accuracy = 0.263000
k = 3, accuracy = 0.273000
k = 3, accuracy = 0.282000
k = 3, accuracy = 0.270000
k = 5, accuracy = 0.265000
k = 5, accuracy = 0.275000
k = 5, accuracy = 0.295000
k = 5, accuracy = 0.298000
k = 5, accuracy = 0.284000
k = 8, accuracy = 0.272000
k = 8, accuracy = 0.295000
k = 8, accuracy = 0.284000
k = 8, accuracy = 0.298000
k = 8, accuracy = 0.290000
k = 10, accuracy = 0.272000
k = 10, accuracy = 0.303000
k = 10, accuracy = 0.289000
k = 10, accuracy = 0.292000
k = 10, accuracy = 0.285000
k = 12, accuracy = 0.271000
k = 12, accuracy = 0.305000
k = 12, accuracy = 0.285000
k = 12, accuracy = 0.289000
k = 12, accuracy = 0.281000
k = 15, accuracy = 0.260000
k = 15, accuracy = 0.302000
k = 15, accuracy = 0.292000
k = 15, accuracy = 0.292000
k = 15, accuracy = 0.285000
k = 20, accuracy = 0.268000
k = 20, accuracy = 0.293000
k = 20, accuracy = 0.291000
k = 20, accuracy = 0.287000
k = 20, accuracy = 0.286000
k = 50, accuracy = 0.273000
k = 50, accuracy = 0.291000
k = 50, accuracy = 0.274000
k = 50, accuracy = 0.267000
k = 50, accuracy = 0.273000
k = 100, accuracy = 0.261000
k = 100, accuracy = 0.272000
k = 100, accuracy = 0.267000
k = 100, accuracy = 0.260000
k = 100, accuracy = 0.267000
```

En choisissant le meilleur K d'après le test sur les hyperparamètres nous obtenons un pourcentage de réussite de 28% ce qui est, de manière contre-intuitive, inférieur à l'algorithme du plus proche voisin.

```
k_nearest_neighbor.py X
k_nearest_neighbor.py > ...
215
216     best_k = 10
217
218     classfier = KNearestNeighbor()
219     classfier.train(X_train, y_train)
220     y_test_pred = classfier.predict(X_test, k=best_k)
221
222     # On calcule et affiche la précision
223     num_correct = np.sum(y_test_pred == y_test)
224     accuracy = float(num_correct) / num_test
225     print('On obtient %d bonne estimation / %d estimation => on a donc une précision de : %f' % (num_correct, num_test, accuracy))
226
227
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE 1: bash + □

```
pc@Majid:~/Bureau/machineLearning/Projet/assignment1/codepython/implementation$ python3 k_nearest_neighbor.p
On obtient 144 bonne estimation / 500 estimation => on a donc une précision de : 0.288000
pc@Majid:~/Bureau/machineLearning/Projet/assignment1/codepython/implementation$
```

4. Conclusion.

Les algorithmes de plus proches voisins présentent l'avantage d'être facile à comprendre cependant leurs performances restent très relatives. En effet, le calculs de dissimilarité par distance des pixels, présentent de nombreux inconvénients (différentes nuances de couleurs d'un objet d'une même classe, arrière plan de couleur identique de deux objets de même classe etc ...) difficiles à résoudre.

5. Implémentation de l'article intitulé: An Analysis of Single Layer Networks in Unsupervised Feature Learning.

Remarque : Nous nous sommes globalement aidé des deux sources internet que sont [1] et [4].

Cette article nous enseigne qu'il est possible d'atteindre des résultats impressionnant sans utiliser d'algorithme d'apprentissage complexe et sans que le modèle d'apprentissage soit profond.

En effet, l'article affirme qu'il est possible d'atteindre un excellent résultat en jouant sur deux critères essentiels, le nombre de noeud dans la couche intermédiaire ainsi que la densité d'extraction des features, à tel point que si ces deux paramètres sont choisis de manière optimal on peut atteindre les meilleurs résultats existants.

5.0.1 Définition des concepts :

Dans cette section nous définissons certains concept clé à l'implémentation de l'algorithme présenté dans l'article.

Normalisation:

Il s'agit de la soustraction suivante:

$$X' = X - \bar{x}$$

Ou X' est l'ensemble de donné normalisé, X l'ensemble de donné original et \bar{x} la moyenne de X . On soustrait au vecteur sa moyenne. Cela aura l'effet de centrer les données autour de 0.

Blanchiment:

L'objectif du blanchiment est de supprimer les corrélations entre pixel afin de forcer l'algorithme à se concentrer sur des données pertinentes.

$$X = U \cdot \text{diag}\left(\frac{1}{\sqrt{\text{diag}(S) + \epsilon}}\right) \cdot U^T \cdot X$$

Avec U le vecteur singulier gauche et S les valeurs singulières de la covariance des données initial après normalisation. ϵ est un hyper-parametre appelé coefficient de blanchiment. $\text{diag}(S)$ est une matrice avec le vecteur S en diagonal et 0 ailleurs.

Features:

Les features sont le resultat de la transformation numérique des caractéristiques que l'algorithme devra analyser pour faire ces prédictions. On obtient ces features par des calculs plus ou moins complexe. Les features sont les parties des données qu'on utilise pour faire notre analyse.

Dropout:

Un procédé qui consiste à detuire aleatoirement des noeuds du reséaux de neurones. C'est un procédé ayant pour objectif de diminuer la différence entre les erreurs d'entraînement et les erreurs de test (Overfitting).

Fonction d'activation:

La fonction d'activation d'un noeud défini la valeur de sortie de ce noeud. Le noeud prend en valeur d'entrée un input qui est transformé en output par la fonction d'activation.

Learning rate (taux d'apprentissage):

Le learning rate est un paramètre qui contrôle la quantité de changement que nous effectuons sur les poids de notre réseau, guidé par la valeur de notre gradient.

Momentum:

Pertubation émise lorsque le gradient est égale à 0 pour sortir du "plat".

Overfitting:

Le modèle prends en considération des points de nos images qui ne représentent pas réellement les propriétés de l'image en question.

Regularization:

Un paramètre qui permet de regulariser la fonction de perte (Loss function) qui parfois a un comportement indésirable. Nous limitons ce comportement en ajoutant un terme de régularisation.

5.0.2 Construction de notre modèle :

Paramétrisation:

Dans cette section nous présenterons succinctement la méthode utilisé pour optimiser le choix du nombre de neurones dans notre unique couche, la méthode utilisé est celle dite de "Grid search" il s'agit, comme dans la première partie du rapport, de tester différentes combinaison possibles pour nos paramètres. Nous avons utilisé cette méthode pour fixer 3 paramètres qui sont:

- Nombre de neurones;
- Taux d'apprentissage (Learning rate);
- La force de regularisation.

Les différents résultats obtenus nous ont permis d'obtenir un score de 51% avec le paramétrage suivant: 350 neurones, un taux d'apprentissage de 0.001 et une force de régularisation de 0.05.

Pour y parvenir nous avons utilisé une exploration du nombre de noeuf allant de 150 à 550 avec un saut de 50.

Pour le taux d'apprentissage nous avons exploré toutes les possibilités entre $1e - 3 * 10^{-3}$ à $1e - 3 * 10^3$. avec un ratio de 10.

Et pour la force de régularisation nous avons exploré toutes les possibilités entre $0.5 * 10^{-3}$ à $0.5 * 10^3$ avec un ratio de 10 aussi.

Preprocessing:

Avant de lancer l'entraînement sur les données nous les avons traités en utilisant deux méthodes. L'analyse en composante principal afin de blanchir les données en effectuant une decorrélation des images en entrée et supprimer les informations redondantes. Ensuite nous utilisons un algorithme de clustering par K-means avec 1600 centroïdes.

Cette étape accélère l'extraction des features et augmente la qualité des prédictions.

Passage en arrière Forward pass et fonction de perte:

L'objectif étant de trouver les bons changements de poids dans la matrice W et de biais dans le vecteur b . Nous avons, pour se faire défini une fonction d'erreur qui, durant l'entraînement compare le résultat de notre modèle avec la valeur exact, notre objectif est des lors de **minimiser** cette **fonction d'erreur (loss function)**.

Comme fonction d'activation, nous avons utilisé la fonction Relu pour éviter les problèmes de saturation qui se présente comme suit:

$$\text{Fonction ReLu [2]} \quad f(x) = \max(0, x)$$

Si l'entrée est négative la sortie est 0 si elle est positive la sortie est x.

Construction de la fonction de perte :

$$\text{Somme pondéré} = \text{np.max}(X.\text{dot}(W1) + b1, 0)$$

$$\text{Score} = \text{Somme pondéré}.\text{dot}(W2) + b2$$

$$\text{Score expo} = \exp(\text{Score}) / (\text{np.sum}(\exp(\text{Score}, 1)).T$$

fonction de perte:

$$\text{loss} = -\text{np.sum}(\text{np.log}(\text{Score expo}[\text{range}(\text{len}(\text{Score expo}), y]))) / \\ \text{len}(\text{Score expo}) + 0.5 * \text{reg} * (\text{np.sum}(\text{np.power}(W1, 2)) + \text{np.sum}(\text{np.power}(W2, 2)))$$

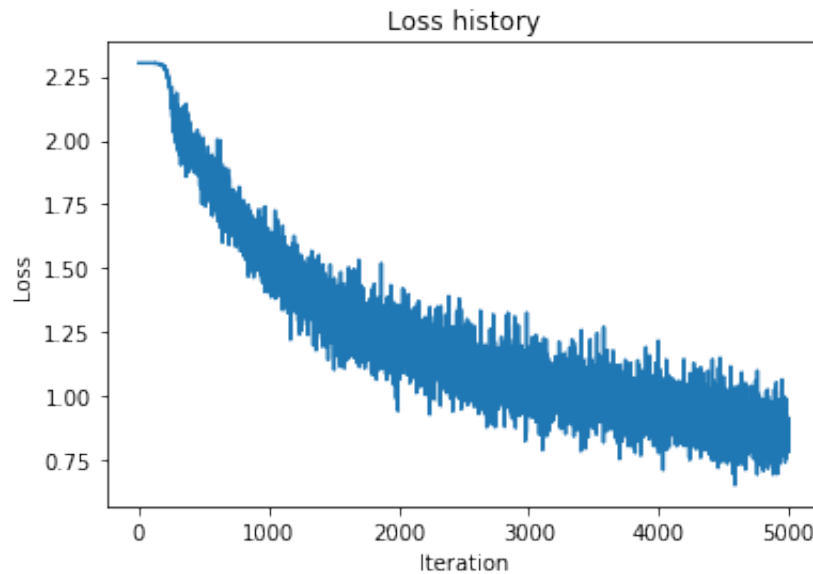


Figure 5.1: Décroissance de la fonction de perte, pour converger à 0.75

Dropout: [3]

Puisque nous avons seulement une seule couche le dropout a seulement besoin d'être effectué une seule fois par échantillon d'image d'entraînement (batch).

Comme dit plus haut, nous faisons cela pour éviter l'overfitting. Pour avoir un bon paramétrage du bon nombre de noeud en fonction du dropout nous sommes avec un nombre de noeud plus conséquent (500) puis avec un taux de relachement de 30,50 puis 70%. Cette méthode nous permet de monter à un résultat de 56% de reconnaissance sans de difference entre les différents taux.

Durant nos tests nous n'avons pas vraiment réussi à minimiser l'overfitting nous pensons que nos résultats sont inférieurs à ceux de l'article à cause de cela.

5.0.3 Résultat:

Comme mentionné plus haut nous avons utilisé la méthode de grid search pour trouver les meilleures hyper-paramètres. L'un des inconvénients non négligeable que nous avons rencontré pour trouver les bons paramètres est le temps que cela met, une solution à ce problème aurait été peut être d'implémenter l'article aurait été de passer sur Pytorch pour faire les calculs facilement sur GPU.

Il y'a des paramétrages qui n'ont pas pu se terminer (augmenter le nombre de neuronne, augmenter le nombre d'itération et autre) pour des raisons d'erreurs mémoire.

En somme le paramétrage optimal que nous avons trouvé est le suivant:

Paramétrage optimal trouvé	
Noeuds dans le réseaux	200
Taux d'apprentissage	5×10^{-4}
Décroissance du taux d'apprentissage	0.98
Régularisation	0.3
Fonction d'activation	ReLu
Itération	70 000
Taille des Batch	128
Précision sur les données d'entrainement	93%
Précision du modèle réels	61.5%
Temps	59 minutes

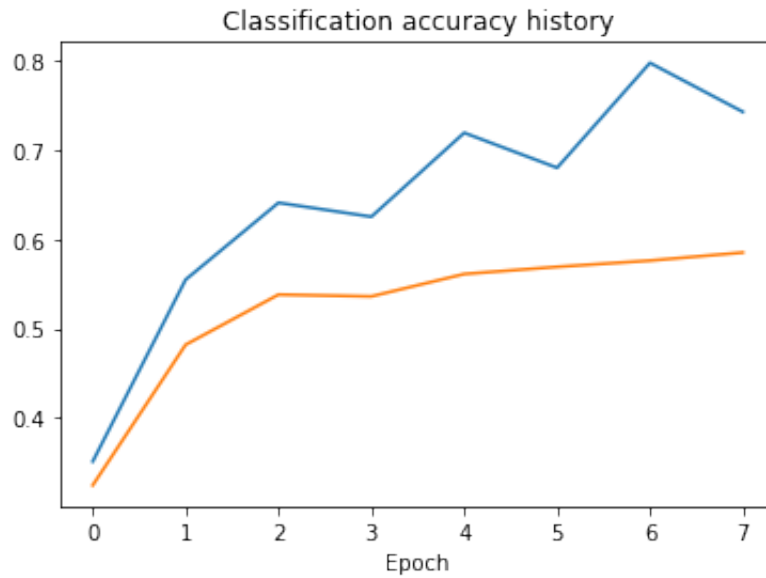


Figure 5.2: en bleu la courbe de reconnaissance des images sur le set d'entraînement en orange sur le set de test. 200 noeuds 5 itérations du K-means 5000 itérations d'entraînement

```
iteration 10300 / 12000: loss 0.517733
iteration 10400 / 12000: loss 0.432989
iteration 10500 / 12000: loss 0.543728
iteration 10600 / 12000: loss 0.526003
train acc 0.898438, val_acc 0.642000, time 37
iteration 10700 / 12000: loss 0.518136
iteration 10800 / 12000: loss 0.442448
iteration 10900 / 12000: loss 0.481236
iteration 11000 / 12000: loss 0.478605
iteration 11100 / 12000: loss 0.405504
iteration 11200 / 12000: loss 0.505862
train acc 0.914062, val_acc 0.619000, time 39
iteration 11300 / 12000: loss 0.470052
iteration 11400 / 12000: loss 0.443891
iteration 11500 / 12000: loss 0.514476
iteration 11600 / 12000: loss 0.496593
iteration 11700 / 12000: loss 0.373349
iteration 11800 / 12000: loss 0.441997
train acc 0.945312, val_acc 0.634000, time 41
iteration 11900 / 12000: loss 0.344892
iteration 12000 / 12000: loss 0.364922
Précision sur les données d'entraînement: 0.9324
Précision réel: 0.615
time 58.76020249525706
pc@Majid:~/Bureau/machineLearning/Projet/codepython/implementation/Articles$
```

5.0.4 Problèmes rencontrés

Nous avons rapidement rencontré des problèmes de taille (mémoire) et de temps (temps d'exécution du code) chaque nouvelle paramétrisation demande une attente d'une heure afin de vérifier son efficacité. Aussi malgré tous nos efforts, nous n'avons pas été en mesure d'atteindre le résultat énoncé dans l'algorithme à ce jour nous pensons que cela est dû à notre échec quant à la minimisation de l'Overfitting.

5.0.5 Amélioration possible

Il existe des fonctions d'activations plus complexes que ReLU notamment ReLU qui pourrait donner de meilleurs résultats.

Une utilisation du multi-thread ou du GPU pour augmenter la vitesse de calcul, permettrait d'augmenter le nombre de nœuds dans la couche ce qui permettrait, je pense, d'obtenir des résultats plus précis. Mais aussi d'augmenter le nombre de degrés de liberté, par exemple augmenter la taille des patches augmenter leurs nombres et augmenter du dropout.

Concernant l'implémentation de l'article je n'utilise que la distance L2 il est possible que la L1 donne de meilleur résultats.

6. Conclusion finale.

Nous n'avions pas considéré à sa juste valeur l'importance des hyperparamètres avec la méthode naïve, en effet ceci était facilement réglable et les combinaisons étaient limitées. L'article nous a vraiment permis de considérer les hyperparamètres comme étant des facteurs centraux dans l'efficacité d'un algorithme. Nous avons à présent le sentiment, que l'un des enjeux majeurs lors de la création d'un modèle et de trouver sa bonne paramétrisation.

Bibliography

- [1] Hadrien. <https://hadrienj.github.io/tags/#numpy>.
- [2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.
- [3] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.
- [4] trongr. <https://trongr.github.io/>.