



Master 1 – Informatique / Computer Science

Projet Cifar 10: An Analysis of Single-Layer Networks in Unsupervised Feature Learning

Rafiei Marzieh

Gueddou Chaouki

Guerard des Lauriers Cédric

Sous la direction de Monsieur le Professeur Guillaume Wisniewski.

Année universitaire 2017/2018

TABLES DES MATIÈRES

INTRODUCTION.....**3**

Chapitre 1 Naive Method :.....**7**

Chapitre 2 Image Preprocessing :.....**9**

- 1) Sélections des Patches
- 2) Normalisation
- 3) Whitening
- 4) Batch Kmeans
- 5) Extraction de Features

Chapitre 3 Support Vector Machine.....**17**

- 1) Rappel du SVM
- 2) La fonction l2_svmLoss L2-SVM
- 3) Méthode L-BFGS

Chapitre 4 Network in Network.....**23**

- 1) Why we choose this method
- 2) Abstract
- 3) Main idea
- 4) Convolutional Neural Networks

Chapitre 5 Two Layers Network.....**29**

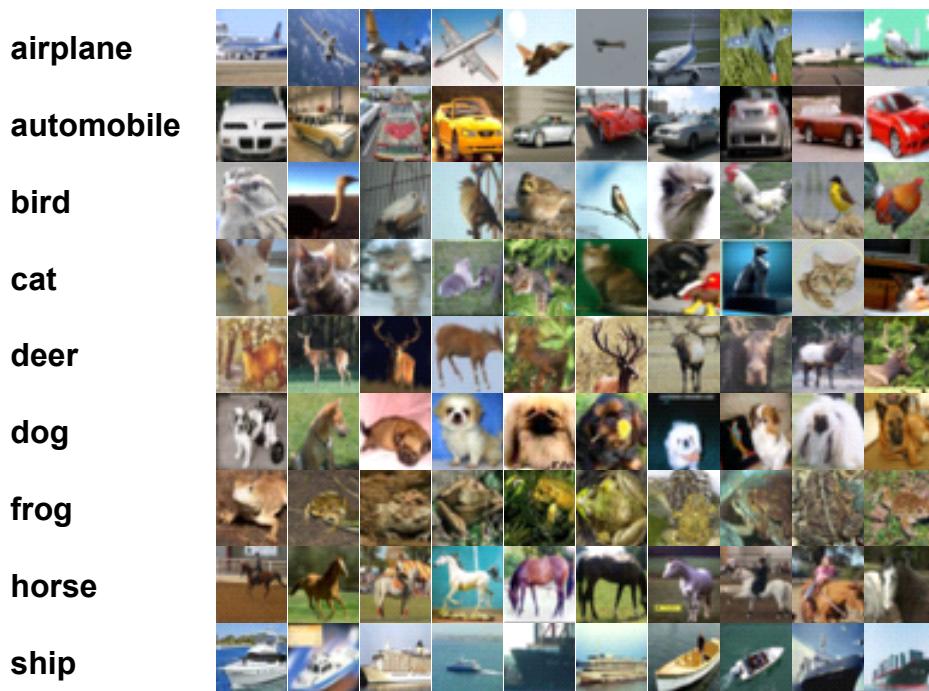
INTRODUCTION

In this research project, the approach is design an image recognition system aims at developing an image recognition system able of automatically identifying the object represented on an image. We will use the Cifar-10 dataset1 that consists of 60000 32x32 pixel color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

Each image has a size of only 32 by 32 pixels. The small size makes it sometimes difficult for us humans to recognize the correct category, but it simplifies things for our computer model and reduces the computational load required to analyze the images. The way we input these images into our model is by feeding the model a whole bunch of numbers. Each pixel is described by three floating point numbers representing the red, green and blue values for this pixel. This results in $32 \times 32 \times 3 = 3072$ values for each image.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

Here are the classes in the dataset, as well as 10 random images from each:

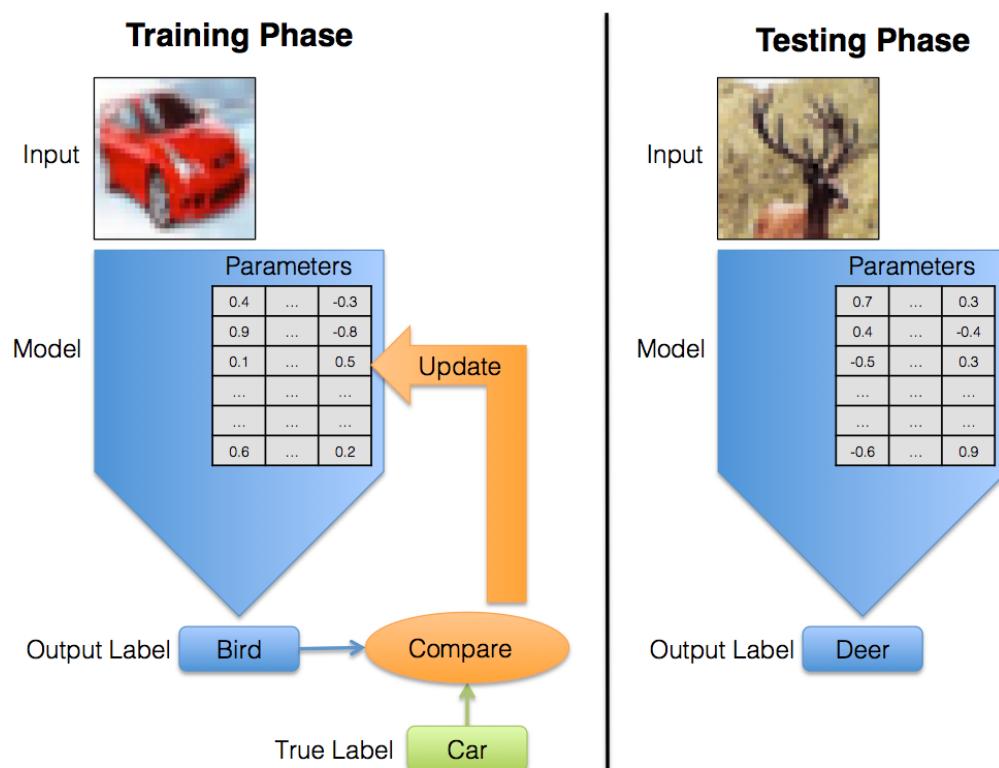




We're defining a general mathematical model of how to get from input image to output label. The model's concrete output for a specific image then depends not only on the image itself, but also on the model's internal parameters. These parameters are not provided by us, instead they are learned by the computer.

The whole thing turns out to be an optimization problem. We start by defining a model and supplying starting values for its parameters. Then we feed the image dataset with its known and correct labels to the model. That's the training stage. During this phase the model repeatedly looks at training data and keeps changing the values of its parameters. The goal is to find parameter values that result in the model's output being correct as often as possible. This kind of training, in which the correct solution is used together with the input data, is called supervised learning. There is also unsupervised learning, in which the goal is to learn from input data for which no labels are available, but that's beyond the scope of this post.

After the training has finished, the model's parameter values don't change anymore and the model can be used for classifying images which were not part of its training dataset.



The first step is the functions that loads data:

-Cifar10 is serparate into 6 batch of data, the function *load_Cifar10()* allow us to extract data for each batch

- In the function *load_Cifar_batch ()* we extract the data X and the label from the dictionary *datadict*, the database Cifar10 (a batch here) is at the start a dictionary (we load it with pickle ,pickled data-file). We reshape (X) the array to 4-dimensions and we reorder the indices of the array (transpose...).

```

import pickle
import numpy as np
import os
#from scipy.misc import imread
from sklearn.cluster import KMeans
from sklearn.cluster import MiniBatchKMeans
import numpy as np
from sklearn.feature_extraction import image

from collections import Counter
from scipy.stats.mstats import mode

def load_CIFAR_batch(filename):
    """ load single batch of cifar """
    with open(filename, 'rb') as f:
        datadict = pickle.load(f,encoding='latin1')
        X = datadict['data']

    Y = datadict['labels']
    X = X.reshape(10000, 3, 32, 32).transpose(0,2,3,1).astype("double")
    Y = np.array(Y)
    return X, Y

def load_CIFAR10(ROOT):
    """ load all of cifar """
    xs = []
    ys = []
    for b in range(1,6):
        f = os.path.join(ROOT, 'data_batch_%d' % (b, ))
        X, Y = load_CIFAR_batch(f)
        xs.append(X)
        ys.append(Y)
    Xtr = np.concatenate(xs)
    Ytr = np.concatenate(ys)
    del X, Y
    Xte, Yte = load_CIFAR_batch(os.path.join(ROOT, 'test_batch'))
#Normalise la base - la moyenne , divise par std, resultat final toujours mauvais normalisé ou non
#Normalize the data, by subtracting the mean and dividing the standard deviation
    mean_image = np.mean(Xtr, axis=0)
    std_image=np.std(Xtr, axis=0)
    #Xtr -= mean_image
    #Xtr /= std_image

    #Xte -= mean_image
    #Xte /= std_image
    return Xtr, Ytr, Xte, Yte

```

```

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    cifar10_dir = 'C:/Users/MyPC/Desktop/Projet Apprentissage Image/cifar-10-batches-py/'

    print (len(load_CIFAR10(cifar10_dir)))
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_train=X_train.swapaxes(1,3)
    X_val=X_val.swapaxes(1,3)
    X_test=X_test.swapaxes(1,3)
    return X_train, y_train, X_val, y_val, X_test, y_test

X_train, y_train, X_val, y_val, X_test, y_test=get_CIFAR10_data()

```

- To finish in `get_Cifar10_data()` we choose a number of images for the training data, the number of images for the test data, and the number of images for the val data. After that we normalize the database, by subtracting the mean.

The last function returns `X_train, y_train, X_val, y_val, X_test, y_test`, those data can allow us to work now.

CHAPITRE 1: Naive Method

Kmeans appliqué directement sur Cifar10:

Le kmeans appliqué directement sur X_train , en formant 10 clusters(10 classes) ne permet pas d'identifier de classe, le résultat de précisions sur les prédictions de X_test est de 7,6% (<0.1 classe uniformément distribuée on a en moyenne 1/10 d'avoir une bonne prédictions). La distance utilisée est la distance euclidienne (soit A (xa, ya) et B (xb,yb) deux points la distance est Racine((xa-xb)^2+(ya-yb)^2)).

Dans une image 32 par 32, il y a 32*32 pixels, et chaque pixel possède une intensité de couleur (R,G,B) qui est la dernière dimension de l'image (dimensions pour chaque image [3,32,32])

Classifieur appliqués directement sur Cifar10

Ici on applique directement des classifieurs sur X_train, on a reshape les images en vecteur de (1,3072) (précédemment chaque image était de dimension [3,32, 32] et chacune des images a été normalisée)., On n'obtient pas loin de 33% de précisions sur les prédictions avec le K plus proches voisins et le Random Forest Classifieur, comme on peut le voir ci-dessous.

```
#appliquer directement des classifieurs sur
XtrainBis= np.zeros((len(X_train),3072))
for i in range(0,len(X_train)):
    XtrainBis[i,:]=X_train[i,:].reshape(1,-1)

from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier(n_neighbors=3)
neigh.fit(XtrainBis, y_train)

KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
      metric_params=None, n_jobs=1, n_neighbors=3, p=2,
      weights='uniform')

XtestBis= np.zeros((len(X_test),3072))
for i in range(0,len(X_test)):
    XtestBis[i,:]=X_test[i,:].reshape(1,-1)

y_pred= neigh.predict(XtestBis)

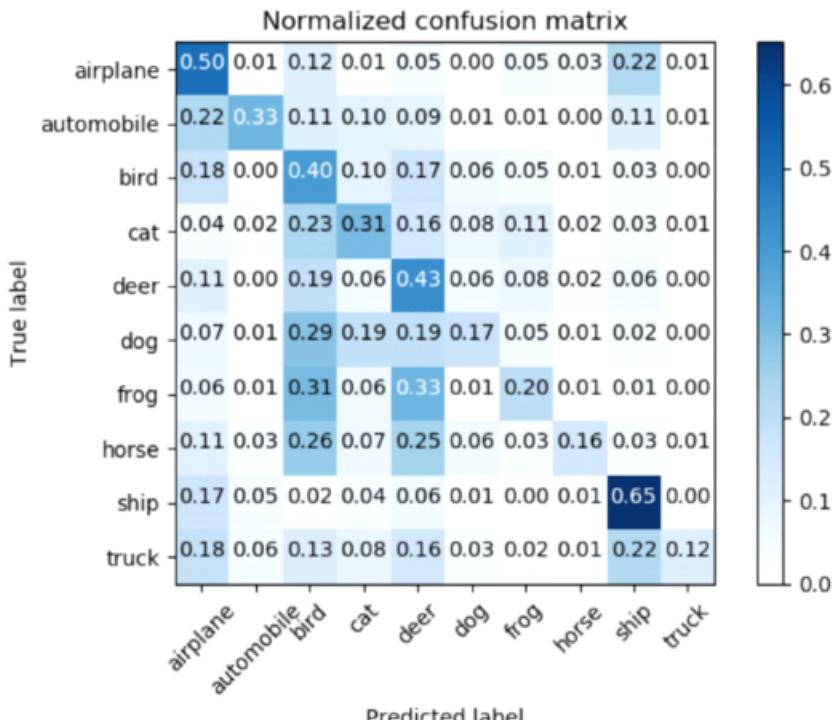
(y_pred==y_test).mean()
0.327

from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(n_estimators=10)
clf = clf.fit(XtrainBis, y_train)
y_pred1= neigh.predict(XtestBis)
(y_pred1==y_test).mean()

0.327
```

La matrice de confusion des 2 classifieurs (qui donne le même résultat ici)



Les bateaux, les avions et dans une moindre mesure les cerfs sont plutôt bien reconnus. Cela est sûrement dû à leur forme particulière, c'est à dire qu'ils possèdent des caractéristiques bien précises qui aident à les différencier. Par exemple l'algorithme classifieur pourra plus souvent se tromper entre un chat et un chien car leurs formes respectives sont proches.
Les 2 algorithmes fonctionnent déjà assez bien car ils confondent souvent un véhicule avec un autre type de véhicule (avion, bateau, voiture) ; et un animal avec un autre type d'animal.

La démarche que l'on va suivre tout au long de ce projet, va consister, en résumé, à isoler précisément ces caractéristiques à l'aide des Kmeans, et ensuite associer ces images à ces caractéristiques, pour former une nouvelle représentation de ces images.

Avec l'aide de cette nouvelle représentation des images, on obtiendra des résultats bien plus précis >76% en suivant la méthode d'Adam Coates.

CHAPITRE 2: Image Preprocessing

1) Sélection des patches aléatoires

Ici on extrait de manière aléatoire, plusieurs patches par image. Dans notre exemple ci-dessous, il y a un peu plus de 4 patches par images, car $\text{len}(X_{\text{train}})=49000$ et $\text{numPatches}=200000$ ($200000/49000$). Les dimensions des patches sont de 6 par 6, chacun des patches sera donc un vecteur de dimension $(1, 6*6*3)$ où *3 la dimension des couleurs.
Pour s'assurer d'extraire plusieurs patches par image on utilise le modulo, comme cela (ainsi pour ?) tous les $\text{len}(X_{\text{train}})$, on obtient un patche associé à telle image.

```
CIFAR_DIM=[32,32,3]
numPatches = 200000
rfSize = 6
patches= np.zeros((numPatches, rfSize*rfSize*3))
for i in range(0, numPatches):
    if i%1000==0:
        print('Extraction patche:', i, '/', numPatches)
    # selection d'un entier aléatoire entre 0 et 26 (pour pas dépasser le format image 32 par 32)
    r= np.random.randint(CIFAR_DIM[0]-rfSize)
    c= np.random.randint(CIFAR_DIM[0]-rfSize)
    # on utilise ici le modulo pour extraire plusieurs patchs par image
    #(tous les len(X_train) un patche sera issu d'une même image
    patch= (X_train[(i%len(X_train)),:])

    #1ère dimension 3 , 2eme 32 , 3eme 32, ici taille du patch à taille voulut [3,6,6]

    patch= patch[:,r:r+rfSize, c:c+rfSize]
    #on reshape le patch pour qu'il soit un vecteur (1, rfsiz*rfisize*3) et le fait rent
    # dans le tableau de patches à la ième ligne
    patches[i]=(patch.reshape(1,-1))

#Normalise les patches
patches=(patches-patches.mean(1)[:,None])/np.sqrt(patches.var(1)+10)[:,None]
```

2) Normalisation des Images :

La normalisation des images est une étape importante en "preprocessing", car elle assure que chaque pixel ait une distribution similaire. Cela permet d'entraîner plus rapidement le classifieur. Ici la normalisation des patches permet de d'uniformiser le contraste et la luminosité. La normalisation permet d'éviter au classifieur de ne pas prendre en compte des variations qui ne sont pas réelles (ex variation de luminosité, de contraste).

3) Le withening :

Le withening est une transformation linéaire qui transforme un vecteur de variables avec une matrice de covariance M, en un vecteur de variables ayant pour matrice de covariance la matrice identité. Cela veut dire que les variables sont désormais décorrélées ($\text{cov}(x_1, x_2)=0$) et que chaque variance est égale à 1 ($\text{cov}(x_1, x_1)=1$). Cette opération est souvent utilisée en Analyse de Composantes Principales (ACP).

Dans notre cas il nous semble intéressant de décorrélérer les patches avant de les passer dans le kmeans, à la manière d'une PCA qui cherchera à réduire les dimensions suivant les corrélations., Ici former un certains nombre de clusters, à l'aide des kmeans et des patches normalisés et passés au withening, est en quelque sorte une "réduction de dimension", puisqu'on cherchera par la suite à évaluer les images selon ces différents clusters.

Le withening va permettre d'éviter les redondances en décorrélant les variables. Cela permettra au kmeans d'isoler plus facilement des caractéristiques.

Par exemple à nombre de centroïdes équivalents, -25 ici-, on passe de 53% d'accuracy sur le test à 45%, soit une perte de 8%. Le withening se révèle donc bénéfique dans la formation des clusters, et ainsi dans l'extraction de features .(Résultat en annexe)

4) Batch Kmeans :

L'idée de cet algorithme est de représenter les données, par un sous-ensemble plus petit de ces données. Par exemple sur 50000 enregistrements, on utilisera seulement 10000 enregistrements. Tout en restant précis, car cette algorithme généralise sur l'ensemble des données. Travailler sur moins de données permet donc un gain de temps dans de grandes dimensions, ce qui est le cas ici.

Le Batch Kmeans sélectionne des sous échantillons de données (le batch, *choisi de manières aléatoires*) à chaque itération. Il assigne ensuite un cluster à chaque donnée contenue dans le batch, dépendant de la location précédente du cluster centroïdes. Ensuite il met à jour les locations du cluster centroïde, en se basant sur les nouveaux points contenus dans le batch.

Le code du Batch Kmeans ci-dessous :

```

import random
def run_kmeans(X,k,iterations):
    x2= np.sum(np.power(X,2), axis=1)
    #centroids = valeur aléatoire suivant la loi normale
    centroids= np.random.randn(k, np.size(X,1))*0.1

    BATCH_SIZE=1000

    for itr in range(1,iterations+1):
        print('kmeans iteration ', itr , '/', iterations)
        c2= 0.5*np.sum(np.power(centroids,2), axis=1)
        summation= np.zeros((k, np.size(X,1)))

        count1=[]
        loss=0
        C=0
        for i in range(1,len(X), BATCH_SIZE ):
            lastindex=min(i+BATCH_SIZE-1, len(X))

            m= lastindex - i +1

            batch= X[i-1:lastindex,:]
            batch_t=batch.transpose()
            tmp= centroids.dot(batch_t)
            for n in range(1, len(batch)):
                tmp[:,n]=tmp[:,n]-c2

            val=[]
            labels=[]
            [val ,labels] = [np.max(tmp, axis=0),np.argmax(tmp, axis=0) ]

            loss= loss + np.sum(0.5*x2[i-1:lastindex]-val.transpose())
            #transformer Label en matrice indicatrice |
            S=np.zeros((m,k))

            for j in range(1, lastindex-i):
                S[j][labels[j]]=1

            summation = summation+(S.transpose()).dot(X[i-1:lastindex,:])
            counts= (np.sum(S,axis=0).transpose())
            count1.append(counts)

            C=C+1

            counts=np.sum(np.array(count1),axis=0)

            for i in range(1, k):

                if counts[i].all!=0:
                    centroids[i]=summation[i,:]/counts[i]
                elif counts[i]==0:
                    centroids[i]=centroids[i]*0

                #on zappe les centroids vide , pour ne pas avoir NAN value
                badIndex= [i for i,val in enumerate(counts) if val.any()==0]
                centroids[badIndex,:]=0

return centroids, counts

```

4) Procédure Extract Features :

1ère Procédure d'extraction de features (convolution, data augmentation) OverLapping Patches :

Un fois les N centres de clusters trouvés à l'aide des K-means, nous allons construire une nouvelle représentation de l'image, en réduisant la dimension de l'images et en calculant la distance entre les clusters trouvés précédemment et les patches contenue contenus dans les images.
Nous ne récupérons pas les anciens patches, mais nous en construisons de nouveaux, et qui se suivent (overlapping patches, patches qui se suivent).

Pour ce faire :

```
patches =np.concatenate([im2col_sliding_broadcasting((X[i].reshape(-1,1)[0:1024]).reshape(32,32) , [6,6], stepsize=1),  
,im2col_sliding_broadcasting((X[i].reshape(-1,1)[1024:2048]).reshape(32,32) , [6,6], stepsize=1),  
im2col_sliding_broadcasting((X[i].reshape(-1,1)[2048:])).reshape(32,32),[ 6,6], stepsize=1) ], axis=0)  
patches=patches.transpose()  
  
patches=np.array(patches)
```

Pour chaque image dans X_train, afin d'obtenir tous les patches possibles différents, suivant une certaine dimension (dans notre cas les patchs sont de dimension 6*6*3) on reshape l'image en vecteur colonne(reshape(-1,1), sépare en 3 (séparation des couleurs) en prenant les 1024 premiers termes, puis les 1024 suivants (3072/3) .On obtient ainsi , nous obtenons donc 3 tableaux de (1,1024) et on les reshape en (32,32).

Ensuite on place ces trois tableaux dans la fonction 'im2col_sliding_broadcasting' avec pour paramètres [6,6] (les 2 premières dimensions du patche)

Pour se faire une idée de ce que fait cette fonction, 'im2col_sliding_broadcasting' avec pour paramètres (2,2) et de stride =1 :

1 2 3	devient : 1 2 4 5
4 5 6	2 3 5 6
7 8 9	4 5 7 8
	5 6 8 9

Et cette dernière avec un stride=2 et de paramètres (2,2) devient :

1 4 4 5
2 5 5 6
2 5 5 8
3 6 6 9

Le code de 'im2col_sliding_broadcasting' :

```
def im2col_sliding_broadcasting(A, BSZ, stepsize=1):
    # Parameters
    M,N = A.shape
    col_extent = N - BSZ[1] + 1
    row_extent = M - BSZ[0] + 1

    # Get Starting block indices
    start_idx = np.arange(BSZ[0])[:,None]*N + np.arange(BSZ[1])

    # Get offsetted indices across the height and width of input array
    offset_idx = np.arange(row_extent)[:,None]*N + np.arange(col_extent)

    # Get all actual indices & index into input array for final output
    return np.take (A,start_idx.ravel()[:,None] + offset_idx.ravel()[:,::stepsize])
```

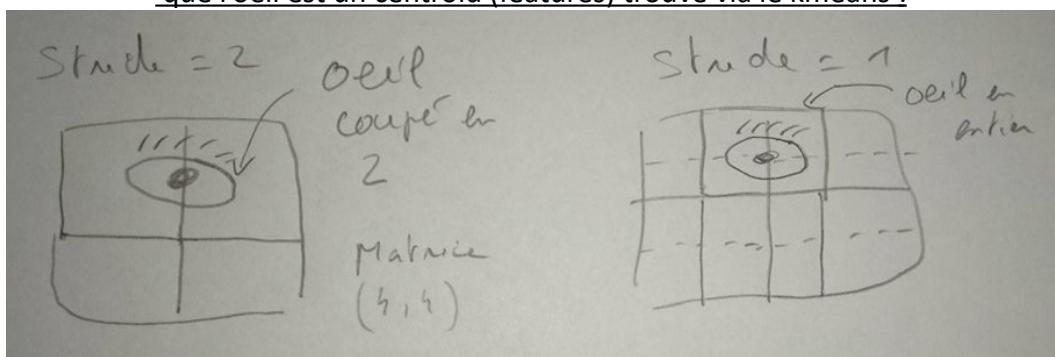
Dans une dimension (32 , 32) avec un stride = 1 , où le stride est le décalage entre chaque voisinage. Ici (Size1, Size2)= (6,6) , on peut donc construire 27 (32-Size1 + 1) voisinages sur la largeur * (32-Size1 + 1) voisinage sur la hauteur , de plus chaque voisinage est de 6 par 6 et est transformé en vecteur colonne (36,1)

La dimension de ce tableau retourné par la fonction ci-dessous est donc de (36, 27*27=729)
(Si le stride était fixé à 2 on aurait eu des dimensions (36, 14*14)

Le tableau "patches " est donc de shape (729,108), une image est donc associée à 729 patches différents (un patche est de dimension (1, 108))

L'intérêt d'avoir tous ces différents patches associés est de réduire la dimensions (3->2). C, cette opération se nomme une convolution, mais aussi on peut désormais comparer chaque patch différent au sein de l'image avec les clusters. Si l'on n'appliquait pas cette méthode, ou si l'on appliquait un stride plus élevé, on pourrait perdre des caractéristiques (en termes de poids / volume, les éléments compris dans le stride seraient comptés moins de fois)

Petit schéma d'un œil se situant dans le stride, en supposant que l'œil est un centroid (features) trouvé via le kmeans :



On pourrait donc perdre en précision comme cela est constaté sur le rapport (-3% de précision pour un stride = 2 et -4 % pour un stride = 4). Ensuite on normalise au sein de chaque patche ces patchés pour éviter les problèmes de contraste ,comme vu précédemment

```
patches=(patches-patches.mean(1)[ :,None])/(np.sqrt(patches.var(1)+10)[ :,None])
#map to feature space
patches=patches.dot(P)
```

Soit P la matrice trouvée à l'aide du whitening (voir précédemment)

Calcul de la matrice des distances

Ensuite on calcule la matrice des distances entre tous les patchés et tous les clusters, en utilisant la $(A-B)^2 = A^2 - 2AB + B^2$ pour un gain évident en termes de temps de calcul , sinon cela prendrait un temps conséquent de calculer les distances une par une.

Rappelons aussi que la distance euclidienne, ou norme 2, entre 2 points X=(x1,x2) et Y=(y1,y2) est égale à $d = \sqrt{(x_1-y_1)^2 + (x_2-y_2)^2}$.

La matrice des distances est donc de dimensions (729, Nombres de Centre de clusters).

```
patches=(patches-patches.mean(1)[ :,None])/(np.sqrt(patches.var(1)+10)[ :,None])
#map to feature space
patches=patches.dot(P)
#calculate distance using x2-2xc+c2
x2=np.power(patches,2).sum(1)
c2=np.power(centroids,2).sum(1)
xc=patches.dot(centroids.T)
dist=np.sqrt(-2*xc+x2[ :,None]+c2)
```

Fonction d'activation : "Above the average"

Nous On ?calculons la moyenne des distances de chaque patche avec les clusters.

Nous soustrayons la moyenne des distances aux distances du patche, si le résultat est négatif, c'est à dire si la distance du patche vis à vis d'un cluster est supérieur à la moyenne des distances du même patche avec tous les autres clusters, alors on met la valeur 0, sinon le on met le résultat de la différence. Plus le résultat de cette différence est grand, plus le patche est comparativement plus proche du cluster en questions, on peut voir cela comme une variable de poids (ou de score, de volume) . Si la valeur est égale à 0 alors le poids est nul.

```
u=dist.mean(1)
patches=np.maximum(-dist+u[ :,None],0)
rs=CIFAR_DIM[0]-rfSize+1
cs=CIFAR_DIM[1]-rfSize+1
patches=np.reshape(patches,[rs,cs,-1])
rs= cs=27
```

L'opération suivante consiste à sommer tous les poids associés au même cluster de tous les patches, la dimensions du tableaux passe de (729, Nombre de clusters) à Nombre de clusters de carrés de 27*27, chaque carré contient les poids associés à un seul cluster.

Pooling : On somme sur les 4 quadrants

Ensuite nous appliquons une méthode de pooling simple qui consiste à sommer les 4 quadrants de chaque carré.

On obtient donc au final un vecteur de taille (1, 4* Nombre de Clusters), ce vecteur contient 4 vecteurs de poids, suivant la position dans l'images.

Ces vecteurs de poids vont donc permettre à des classifieurs de s'entraîner et faire de meilleures prédictions.

```
q=[]
q.append(patches[0:int(rs/2)+1,0:int(cs/2)+1].sum(0).sum(0))
q.append(patches[0:int(rs/2)+1,int(cs/2)+1:cs-1].sum(0).sum(0))
q.append(patches[int(rs/2)+1:rs-1,0:int(cs/2)+1].sum(0).sum(0))
q.append(patches[int(rs/2)+1:rs-1,int(cs/2)+1:cs-1].sum(0).sum(0))
q=np.array(q).ravel()
trainXC.append(q)
trainXC=np.array(trainXC)
trainXC=(trainXC-trainXC.mean(1)[:,None])/(np.sqrt(trainXC.var(1)+.01)[:,None])
return trainXC
```

On ravel pour former un tableaux de dimension (len(X_train), 4*Nombres de clusters), chaque ligne de trainXC est une nouvelle représentation de l'image, cette dernière va nous permettre d'entrainer un classifieur.

2ème procédure d'extraction de features (Méthode Suggérée dans le sujet):

Dans la méthode suggérée dans le sujet, on trouve 4 patches aléatoires par image, en suivant la même méthode que précédemment pour trouver les clusters.

Ces 4 patches par image représenteront l'image. Ensuite on calculera la distance de chacun de ces patches avec les clusters et on ne gardera que l'index du minimum (par rapport au cluster) des distances par patche pour chaque image.

Ces index nous permettront donc de construire des vecteur de la forme : (0 0 0 1 0 0.....)

1 si c'est l'index minimum des distances, 0 dans le cas contraire.

Cette nouvelle représentation (de dimensions (len(X_train), 4*numCentroids)) permet de dire, pour résumer, quelles sont les features les plus proches de telles images. Cette représentation est plus simple, et plus rapide à calculer dans l'extraction des features, que celle vue précédemment elle permettra elle aussi à entraîner un classifieur.

Cependant les prédictions seront moins précise, à titre d'exemple pour 25 features (clusters) , on passera de 55% d'accuracy (score) sur le X_train et y_train avec la méthode précédente à 22% sur d'accuracy sur le X_train et y_train, c'est pour cela que nous n'avons pas plus développez cette méthode qui est bien moins précise. Et de 66% à 27% pour 100 features.

Le code de cette méthode ci-dessous :

```
# Generation d'autre patches aléatoire
patches1= np.zeros((numPatches, rfSize*rfSize*3))
for i in range(0, numPatches):
    if i%1000==0:
        print('Extraction patche:', i, '/', numPatches)
    r= np.random.randint(CIFAR_DIM[0]-rfSize)
    c= np.random.randint(CIFAR_DIM[0]-rfSize)
    patch= (X_train[(i%len(X_train)),:])

    patch= patch[:,r:r+rfSize, c:c+rfSize]

    patches1[i]=(patch.reshape(1,-1))
patches1=(patches1-patches1.mean(1)[:,None])/np.sqrt(patches1.var(1)+10)[:,None]

#matrice withening
patches1=patches1.dot(P)

#Calcul des distances patches Centroids
x2=np.power(patches1,2).sum(1)
c2=np.power(centroids,2).sum(1)
xc=patches1.dot(centroids.T)
dist=np.sqrt(-2*xc*x2[:,None]+c2)

#Recuperation des Index Minimum
indexMin=[]
for i in range(0, len(dist)):
    indexMin.append(np.argmin(dist[i]))

NewRep= np.zeros((len(X_train),4))
j=0

for i in range(0, len(dist)-4000):
    NewRep[i%len(X_train),j]=indexMin[i]

    if i== (j+1)*len(X_train):
        j=j+1
#Nouvelle représentation vectorielle |
NewRepBis=np.zeros((len(X_train),4*numCentroids))
for i in range(0, len(NewRep)):
    NewRepBis[i, int(NewRep[i,0])]=1
    NewRepBis[i, numCentroids+int(NewRep[i,1])]=1
    NewRepBis[i, 2*numCentroids+int(NewRep[i,2])]=1
    NewRepBis[i, 3*numCentroids+int(NewRep[i,3])]=1
```

CHAPITRE 3: Support Vector Machine

1) Rappel du SVM (Wikipedia), Cas simple 2 classes :

"La première idée clé est la notion de marge maximale. La marge est la distance entre la frontière de séparation et les échantillons les plus proches. Ces derniers sont appelés vecteurs supports. Dans les SVM, la frontière de séparation est choisie comme celle qui maximise la marge. Le problème est de trouver cette frontière séparatrice optimale, à partir d'un ensemble d'apprentissage. Ceci est fait en formulant le problème comme un problème d'optimisation quadratique, pour lequel il existe des algorithmes connus.

Les SVM peuvent être utilisés pour résoudre des problèmes de discrimination, c'est-à-dire décider à quelle classe appartient un échantillon, ou de régression, c'est-à-dire prédire la valeur numérique d'une variable. La résolution de ces deux problèmes passe par la construction d'une fonction h qui ,à un vecteur d'entrée x fait correspondre une sortie y :

$y=h(x)$, avec $h(x)=\text{Transposée}(w)^*x + w_0$, w_0 le biais

y appartient $\{-1,1\}$ et x dans un espace X muni d'un produit scalaire, par exemple R^N "

Dans notre cas nous ne sommes pas dans un support vecteur machines(SVM) linéaire, mais dans le cas d'un SVM à marge souple, c'est à dire qu'il n'est pas possible de trouver une ligne séparatrice à l'aide de la fonction noyau (Kernel trick)., il se peut aussi que des échantillons soient mal étiquetés, et que l'hyperplan séparateur ne soit pas la meilleure solution au problème de classement.

"La technique de la marge souple tolère les mauvais classements. La technique cherche un hyperplan séparateur qui minimise le nombre d'erreurs grâce à l'introduction de variables ressort **EpsilonK** (slack variables en anglais), qui permettent de relâcher les contraintes sur les vecteurs d'apprentissage, autrement dit EpsilonK pénalise les points qui violent la marge.

Exemple ci-dessous du L1-SVM

$$l_k(w^T x_k + w_0) \geq 1 - \xi_k \quad \xi_k \geq 0, \quad 1 \leq k \leq p$$

Avec les contraintes précédentes, le problème d'optimisation est modifié par un terme de pénalité, qui pénalise les variables ressort élevées :

$$\text{Minimiser } \frac{1}{2} \|w\|^2 + C \sum_{k=1}^p \xi_k \quad , \quad C > 0$$

Notons ici que nous incluons le biais en ajoutant 1 à tous les scalaires de trainXC (ici noté Asc) à l'aide de ce code :

```
Asc=np.c_[ Asc, np.ones(len(A)) ]
```

La dimensions de **trainXC** (ici écrit Asc) sera donc (M, N) , où M le nombre d'image de trainXC et $N=4*\text{nombre de Centroids}+1$

```
[M,N]= np.shape(X)
theta= w.reshape(N,K)
theta de dimension (N,K)
(4*Nombre de Centroids+1=N, Nombre de Classes=K)
```

2) La fonction myl2svmloss résolution d'un L2-SVM:

L'apprentissage du SVM consiste à trouver w^* (ici theta*) la matrice de poids de vecteurs tel que w^* minimise la fonction loss (L2-SVM erreur quadratique):

$$\begin{aligned}
 \min_{\theta} \text{loss} &= \frac{1}{2} \sum_{m=1}^M \sum_{k=1}^K \theta_{m,k}^2 + C \sum_{m=1}^M \sum_{k=1}^K \max(1 - y_{m,k} (X_m \cdot \dot{\theta}_k), 0)^2 \\
 &\quad \underbrace{\qquad\qquad\qquad}_{\text{theta}} \qquad \underbrace{\qquad\qquad\qquad}_{\text{margin quadratique}} \qquad \underbrace{\qquad\qquad\qquad}_{\text{epsilon}} \\
 &= \|\theta\|^2 \\
 &\quad \underbrace{\qquad\qquad\qquad}_{\text{norme euclidienne}} \qquad \underbrace{\qquad\qquad\qquad}_{\text{epsilon}} \\
 \text{sous contraintes : } \epsilon_k &\geq 1 - y_{m,k} (X_m \cdot \dot{\theta}_k) \quad \forall k \\
 \epsilon_k &\geq 0 \quad \forall k
 \end{aligned}$$

Pour cela on écrit dans le programme :

```
loss= (0.5* np.sum((np.power(theta,2)), axis=0))+ C*np.mean((np.power(margin,2)), axis=0)
loss= np.sum(loss, axis=0)
```

Nous calculons ensuite le gradient de loss par rapport à theta, et on le reshape en $(N*K, 1)$ afin de pouvoir utiliser la fonction de minimisation par la suite (le gradient doit être de dimension 1 pour déduire une approximation de l'hessienne qui sera elle de dimension 2) :

```
g = theta - 2*C/M*(X.transpose().dot(margin*Y))
g.reshape(np.size(X, 1)*K, 1)
```

La valeur du gradient g de loss à la n-ième ligne et à la k-ième colonne (avant reshape)

$$\theta = (\theta_{m,k})_{\substack{1 \leq m \leq N \\ 1 \leq k \leq K}} \quad y = (y_{m,k})_{\substack{1 \leq m \leq M \\ 1 \leq k \leq K}}$$

$$x_{train}^t = (x_{m,m})_{\substack{1 \leq m \leq N \\ 1 \leq m \leq M}}$$

$$margin \times y = (margin_{m,k} \times y_{m,k})_{\substack{1 \leq m \leq M \\ 1 \leq k \leq K}}$$

$$x_{train}^t \cdot dot(margin \times y) = \sum_{m=1}^M (x_{m,m}) \times (margin_{m,k} \times y_{m,k})$$

$$\text{soit } g \text{ le gradient de loss} = (g_{m,k})_{\substack{1 \leq m \leq N \\ 1 \leq k \leq K}}$$

$$g_{m,k} = \theta_{m,k} - \frac{2}{M} \left(\sum_{m=1}^M (x_{m,m}) \times (margin_{m,k} \times y_{m,k}) \right)$$

↑
la valeur de : $\left(\frac{d \text{loss}}{d \theta_{m,k}} \right)_{\substack{1 \leq m \leq N \\ 1 \leq k \leq K}}$
pour m et k fixé

La fonction `my_l2svmloss` renvoi donc la fonction à minimiser `loss` , et le gradient de `loss` par rapport à `theta`. L'avantage d'avoir utilisé le L2-SVM est que celui-ci est différentiable (cela va nous être utile pour utiliser l'algorithme `l-bfgs`) contrairement au L1-SVM (variable de ressort **EpsilonK** linéaire).

De plus le L2-SVM impose une erreur plus grande (erreur quadratique vs erreur linéaire) pour les points qui violent la marge, avec la variable de ressort qui est mise au carré.

Ici nous sommes dans le cas d'un SVM multiclass, on utilise donc l'approche du *one vs all*. C'est à dire que pour trier 10 classes on subdivise le SVM en 10 SVM binaire, ici les 10 SVM sont compris dans la dimension $K=10$.

Pour mettre en place ces 10 SVM, nous construisons la matrice **Y** témoignant de l'appartenance d'une image à une classe :

```
Y= np.zeros((len(y),K))
for i in range(0, len(y)):
    for k in range(1, K+1):
        if y[i]==k:
            Y[i,k-1]=Y[i,k-1]+1
        else: Y[i,k-1]=Y[i,k-1]-1
```

On a $Y[m,k] = 1$ si l'élément (l'image) à la ligne m de **trainXC** appartient à la classe k où k appartient **[1,10]**, et -1 sinon.(One Hot Vector)

Après avoir réussi à trouver le `theta` optimal, `theta*`, en minimisant `loss`.
`Y_pred` sera simplement égal à :

`Y_pred = argmax (trainXC.dot(theta*))` (argmax par rapport à k)
et `Size(Y_pred) = Size(trainXC,0)` = nombres d'images à classifier

Avant de trouver `Y_pred` il faut donc trouver `theta (w*)` tel que `theta` minimise `loss`.

Pour cela nous allons utiliser la fonction `fmin_l_bfgs_b` qui va minimiser la fonction `my_l2svm_loss` (qui retourne `loss` et son gradient).

La fonction `fmin_l_bfgs_b` implémente l'algorithme Limited-memory BFGS (L-BFGS), cette algorithme est de la famille des méthodes de quasi-Newton.

Petit rappel de la méthode de Newton :

Si on cherche à minimiser la fonction $f(x)$, on cherchera x tel que $\text{gradient}(f(x))=0$, cela nous donnera comme itérations:

$X^{k+1} = X^k - \text{Hessienne}(f(X^k))^{-1} * \text{gradient}(f(X^k))$, où k la k -ième itération

En rappelant que le gradient contient les dérivées premières, la matrice hessienne contient les dérivées secondes (matrices symétriques).

L'idée de la méthode de quasi Newton est de remplacer Hessienne($f(X^k)$) $^{-1}$ par une matrice B_k plus facile à calculer. Dans des dimensions trop grandes, comme c'est le cas ici la matrice hessienne serait trop longue à calculer.

Les itérations des méthodes de quasi-Newton sont alors de la forme suivante :

$$X^{k+1} = X^k - p_k * B_k * \text{gradient}(f(X^k))$$

Dans cette formule, p_k est un coefficient choisi pour optimiser la convergence, B_k est mise à jour à chaque itération selon une formule particulière. Selon les méthodes de quasi-Newton, la formule de mise à jour varie.

3) La méthode L-BFGS:

L'algorithme démarre avec l'estimation d'une valeur optimale qui minimise loss. Ici on démarre avec **w0 (de dimension (N*K,1))**, qui sera redimensionné en **(N,K)** dans la fonction `svm_2loss`, l'algorithme procède manièrre itérative permettant de trouver de meilleure valeur de theta (w) (au sens précisions des prédictions). Les dérivées dans le gradient de loss (trouvées précédemment) permettent d'identifier la direction de la plus forte pente et d'estimer la matrice Hessien de loss.

Voici comment se déroule l'algorithme L-BFGS (Wikipédia), on pourra mettre cette photo en annexe :

We'll take as given x_k , the position at the k -th iteration, and $g_k \equiv \nabla f(x_k)$ where f is the function being minimized, and all vectors are column vectors. We also assume that we have stored the last m updates of the form $s_k = x_{k+1} - x_k$ and $y_k = g_{k+1} - g_k$. We'll define $\rho_k = \frac{1}{y_k^T s_k}$, and H_k^0 will be the 'initial' approximate of the inverse Hessian that our estimate at iteration k begins with.

The algorithm is based on the BFGS recursion for the inverse Hessian as

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T.$$

For a fixed k we define a sequence of vectors q_{k-m}, \dots, q_k as $q_k := g_k$ and $q_i := (I - \rho_i y_i s_i^T) q_{i+1}$. Then a recursive algorithm for calculating q_i from q_{i+1} is to define $\alpha_i := \rho_i s_i^T q_{i+1}$ and $q_i = q_{i+1} - \alpha_i y_i$.

We also define another sequence of vectors z_{k-m}, \dots, z_k as $z_i := H_i q_i$. There is another recursive algorithm for calculating these vectors which is to define $z_{k-m} = H_k^0 q_{k-m}$ and then recursively define $\beta_i := \rho_i y_i^T z_i$ and $z_{i+1} = z_i + (\alpha_i - \beta_i) s_i$. The value of z_k is then our approximation for the direction of steepest ascent.

Thus we can compute the (uphill) direction as follows:

```

 $q = g_k$ 
For  $i = k-1, k-2, \dots, k-m$ 
   $\alpha_i = \rho_i s_i^T q$ 
   $q = q - \alpha_i y_i$ 
 $H_k^0 = y_{k-1} s_{k-1}^T / y_{k-1}^T y_{k-1}$ 
 $z = H_k^0 q$ 
For  $i = k-m, k-m+1, \dots, k-1$ 
   $\beta_i = \rho_i y_i^T z$ 
   $z = z + s_i (\alpha_i - \beta_i)$ 
Stop with  $H_k g_k = z$ 

```

La fonction `fmin_l_bfgs_b` renvoie donc $w = \theta^*$, la valeur de $\text{loss}(\theta^*) = fw$
Et un dictionnaire indiquant la convergence ou non = i

```
w, fw ,i    = fmin_l_bfgs_b(my_l2svmloss,w0,args =([X,y,K,C  ]), maxfun=1000, maxiter=1000)
theta= w.reshape(np.size(trainXC, 1), K )
```

Puis on reshape theta* pour l'utiliser pour nos prédictions

Ensuite après avoir standardisé les données trainXC (on l'a fait avant d'avoir entraîné le modèle) et testXC, on déduit les Y_pred, en récupérant l'indice compris entre 1 et 10, du score maximum pour chaque image (trainXC.dot(theta)), cet indice est la classe prédite.

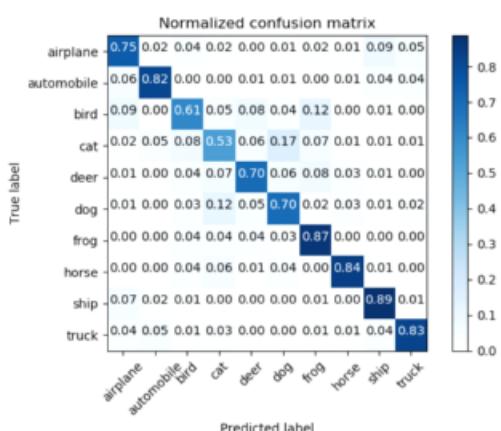
```
Matrice = Asc.dot(theta)      #theta= w  theta shape : (4* num Centroids, number of class ), Asc= x ,
#Asc.shape = 50000, 4*numCentroids, Matrice (50000, 10) argamx axis= 1 sort the number of the class predict
#higher score with weight vector
[val1 ,inds1] = [np.max(Matrice, axis=1),np.argmax(Matrice, axis=1) ] #y_pred= argmax(x*wy)
print('train result :')
(inds1==y_train).mean()
```

On n'a plus qu'à tester la précision du modèle sur l'ensemble d'apprentissage et sur l'ensemble X_test.

```
test=extract_features (X_test )
test=test - np.mean(A,axis=0)
test= test/ np.sqrt(np.var(A,axis=0)+0.01)
Tsc=np.c_[test, np.ones(len(test)) ]
MatriceTest1= Tsc.dot(theta)
[val3 ,inds3] = [np.max(MatriceTest1, axis=1),np.argmax(MatriceTest1, axis=1) ]
print('Accuracy result on Xtest:')
(inds3==y_test).mean()
```

On obtient donc avec le SVM+ le preprocessing avec 25 features 53 % précision de prédictions de classes, 64 % avec 100 features et plus de 76 % avec 1600 features

1600 features SVM 77% de précisions sur X_test



CHAPITRE 4: Network in Network

1) Why do we choose these method

with this method we've obtained 89% accuracy in few time (3.4 hours) with a standard GPU who has this specifications : Tesla K80 · 12 GB Memory. 61 GB RAM · 100 GB SSD

We've based on the work of:

Min Lin: Graduate School for Integrative Sciences and Engineering

Qiang Chen: Department of Electronic & Computer Engineering

Shuicheng Yan: National University of Singapore, Singapore

Who published a research paper where they used CIFAR 10 on September 6th 2013.

In our work we've used version 3 published on March 4th 2014.

2)Abstract

We use a novel deep network structure called “Network in Network” (NIN) to enhance model discriminability for local patches within the receptive field. The conventional layer uses linear filters followed by a nonlinear activation function to scan the input. Instead, we build micro neural networks with more complex structures to abstract the data within the receptive field. We instantiate the micro neural network with a multilayer perceptron, which is a potent function approximator. The feature maps are obtained by sliding the micro networks over the input in a similar manner as CNN; they are then fed into the next layer. We use a global average pooling over feature maps in the classification layer, which is easier to interpret and less prone to over fitting than traditional fully connected layers.

We demonstrated the state-of-the-art classification performances with NIN on CIFAR-10 dataset where we had 89% accuracy (91% in the research paper).

3) Main Idea

Convolutional neural networks (CNNs) consist of alternating convolutional layers and pooling layers. Convolution layers take inner product of the linear filter and the underlying receptive field followed by a nonlinear activation function at every local portion of the input. The resulting outputs are called feature maps.

The convolution filter in CNN is a generalized linear model (GLM) for the underlying data patch, and we argue that the level of abstraction is low with GLM. By abstraction we mean that the feature is invariant to the variants of the same concept. Replacing the GLM with a more potent nonlinear function approximator can enhance the abstraction ability of the local model. GLM can achieve a good extent of abstraction when the samples of the latent concepts are linearly separable, thus conventional CNN implicitly makes the assumption that the latent concepts are linearly separable.

In NIN, the GLM is replaced with a “micro network” structure which is a general nonlinear function approximator.

In this work, we choose multilayer perceptron as the instantiation of the micro network, which is a universal function approximator and a neural network trainable by back-propagation. The resulting structure which we call an Mlpconv layer is compared with CNN in Figure 1. Both the linear convolutional layer and the mlpconv layer map the local receptive field to an output feature vector. The mlpconv maps the input local patch to the output feature vector with a multilayer perceptron (MLP) consisting of multiple fully connected layers with nonlinear activation functions.

The MLP is shared among all local receptive fields. The feature maps are obtained by sliding the MLP over the input in a similar manner as CNN and are then fed into the next layer. The overall structure of the NIN is the stacking of multiple mlpconv layers. It is called “Network In Network” (NIN) as we have micro networks (MLP), which are composing elements of the overall deep network, within mlpconv layers. Instead of adopting the traditional fully connected layers for classification in CNN, we directly output the spatial average of the feature maps from the last mlpconv layer as the confidence of categories via a global average pooling layer, and then the resulting vector is fed into the softmax layer. In traditional CNN, it is difficult to interpret how the category level information from the objective cost layer is passed back to the previous convolution layer due to the fully connected layers which act as a black box in between. In contrast, global average pooling is more meaningful and interpretable as it enforces correspondance between feature maps and categories, which is made possible by a stronger local modeling using the micro network. Furthermore, the fully connected layers are prone to overfitting and heavily depend on dropout regularization, while global average pooling is itself a structural regularizer, which natively prevents overfitting for the overall structure.

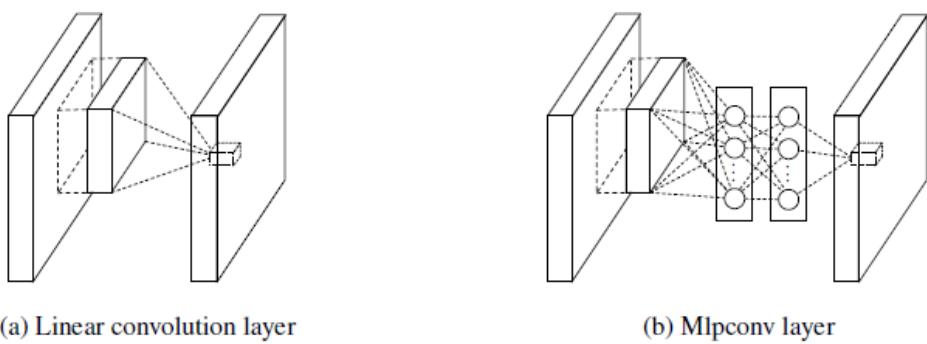


Figure 1: Comparison of linear convolution layer and mlpconv layer. The linear convolution layer includes a linear filter while the mlpconv layer includes a micro network (we choose the multilayer perceptron in this paper). Both layers map the local receptive field to a confidence value of the latent concept.

4) Network In Network

We first highlight the key components of our proposed “Network In Network” structure: the MLP convolutional layer and the global averaging pooling layer in Sec. 4.1 and Sec. 4.2 respectively. Then we detail the overall NIN in Sec. 4.3.

4.1 - MLP Convolution Layers

Given no priors about the distributions of the latent concepts, it is desirable to use a universal function approximator for feature extraction of the local patches, as it is capable of approximating more abstract representations of the latent concepts. We choose multilayer perceptron in this work for two reasons. First, multilayer perceptron is compatible with the structure of convolutional neural networks, which is trained using back-propagation. Second, multilayer perceptron can be a deep model itself, which is consistent with the spirit of feature re-use. This new type of layer is called mlpconv in previous mentioned paper, in which MLP replaces the GLM to convolve over the input.

Figure 1 illustrates the difference between linear convolutional layer and mlpconv layer. The calculation performed by mlpconv layer is shown as follows:

$$\begin{aligned} f_{i,j,k_1}^1 &= \max(w_{k_1}^1{}^T x_{i,j} + b_{k_1}, 0). \\ &\vdots \\ f_{i,j,k_n}^n &= \max(w_{k_n}^n{}^T f_{i,j}^{n-1} + b_{k_n}, 0). \end{aligned}$$

Here n is the number of layers in the multilayer perceptron. Rectified linear unit is used as the activation function in the multilayer perceptron.

4.2 Global Average Pooling

According to the research paper previously mentioned, we propose another strategy called global average pooling to replace the traditional fully connected layers in CNN. The idea is to generate one feature map for each corresponding category of the classification task in the last mlpconv layer. Instead of adding fully connected layers on top of the feature maps, we take the average of each feature map, and the resulting vector is fed directly into the softmax layer. One advantage of global average pooling over the fully connected layers is that it is more native to the convolution structure by enforcing correspondences between Feature maps and categories. Thus the feature maps can be easily interpreted as categories confidence maps. Another advantage is that there is no parameter to optimize in the global average pooling thus overfitting is avoided at this layer. Furthermore, global average pooling sums out the spatial information, thus it is more robust to spatial translations of the input.

4.3 Network In Network Structure

The overall structure of NIN is a stack of mlpconv layers, on top of which lie the global average pooling and the objective cost layer. Sub-sampling layers can be added in between the mlpconv layers as in CNN and maxout networks. Figure 2 shows an NIN with three mlpconv layers. Within each mlpconv layer, there is a three-layer perceptron. The number of layers in both NIN and the micro networks is flexible and can be tuned for specific tasks.

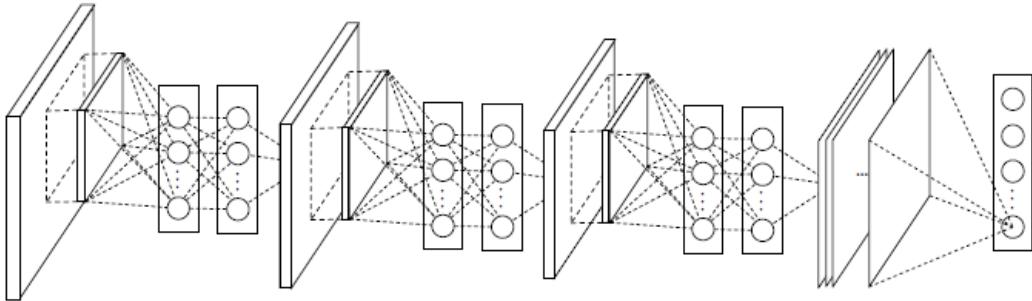


Figure 2: The overall structure of Network In Network. In this paper the NINs include the stacking of three mlpconv layers and one global average pooling layer.

5) Implementation

For our code we have used the keras library and tensorflow as a backend

This is the parameters of our network in network implementation:

```
batch_size      = 128
epochs         = 200
iterations     = 391
num_classes    = 10
dropout        = 0.5
weight_decay   = 0.0001
log_filepath   = './nin'
```

Image preprocessing

```
def color_preprocessing(x_train,x_test):
    x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')
    mean = [125.307, 122.95, 113.865]
    std = [62.9932, 62.0887, 66.7048]
    for i in range(3):
        x_train[:, :, :, i] = (x_train[:, :, :, i] - mean[i]) / std[i]
        x_test[:, :, :, i] = (x_test[:, :, :, i] - mean[i]) / std[i]

    return x_train, x_test
```

The learning rate scheduler

From 1 to 80 epoch →→→ the learning rate equal to 0.01

From 81 to 140 epoch →→→ the learning rate equal to 0.005

From 141 to 200 epoch →→→ the learning rate equal to 0.001

```
def scheduler(epoch):
    if epoch <= 80:
        return 0.01
    if epoch <= 140:
        return 0.005
    return 0.001
```

Building the model

Creating a model based on the architecture shown in figure 2

```
def build_model():
    model = Sequential()

    model.add(Conv2D(192, (5, 5), padding='same', kernel_regularizer=keras.regularizers.l2(weight_decay), kernel_initializer="he_normal"))
    model.add(Activation('relu'))
    model.add(Conv2D(160, (1, 1), padding='same', kernel_regularizer=keras.regularizers.l2(weight_decay), kernel_initializer="he_normal"))
    model.add(Activation('relu'))
    model.add(Conv2D(96, (1, 1), padding='same', kernel_regularizer=keras.regularizers.l2(weight_decay), kernel_initializer="he_normal"))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2), padding = 'same'))

    model.add(Dropout(dropout))

    model.add(Conv2D(192, (5, 5), padding='same', kernel_regularizer=keras.regularizers.l2(weight_decay), kernel_initializer="he_normal"))
    model.add(Activation('relu'))
    model.add(Conv2D(192, (1, 1), padding='same', kernel_regularizer=keras.regularizers.l2(weight_decay), kernel_initializer="he_normal"))
    model.add(Activation('relu'))
    model.add(Conv2D(192, (1, 1), padding='same', kernel_regularizer=keras.regularizers.l2(weight_decay), kernel_initializer="he_normal"))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2), padding = 'same'))

    model.add(Dropout(dropout))

    model.add(Conv2D(192, (3, 3), padding='same', kernel_regularizer=keras.regularizers.l2(weight_decay), kernel_initializer="he_normal"))
    model.add(Activation('relu'))
    model.add(Conv2D(192, (1, 1), padding='same', kernel_regularizer=keras.regularizers.l2(weight_decay), kernel_initializer="he_normal"))
    model.add(Activation('relu'))
    model.add(Conv2D(10, (1, 1), padding='same', kernel_regularizer=keras.regularizers.l2(weight_decay), kernel_initializer="he_normal"))
    model.add(Activation('relu'))

    model.add(GlobalAveragePooling2D())
    model.add(Activation('softmax'))

    sgd = optimizers.SGD(lr=.1, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])
    return model
```

Main method

```

if __name__ == '__main__':
    # Load data
    (x_train, y_train), (x_test, y_test) = cifar10.load_data()
    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)

    x_train, x_test = color_preprocessing(x_train, x_test)

    # build network
    model = build_model()
    print(model.summary())

    # set callback
    tb_cb = TensorBoard(log_dir=log_filepath, histogram_freq=0)
    change_lr = LearningRateScheduler(scheduler)
    cbks = [change_lr, tb_cb]

    # set data augmentation
    print('en utilisant real-time data augmentation !!')
    datagen = ImageDataGenerator(horizontal_flip=True, width_shift_range=0.125, height_shift_range=0.125, fill_mode='constant', cval=0)
    datagen.fit(x_train)

    # start training
    model.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_size), steps_per_epoch=iterations, epochs=epochs, callbacks=cbks)
    model.save('nin.h5')

```

Output

We have run the algorithm three times and we were able to get different results which are :

First run 200 epoch + 3.5 hours >>> 74.39%

Second run 80 epoch + 1.46 hours >>> 87.38%

Third run 200 epoch + 3.4 hours >>> 89.44%

```

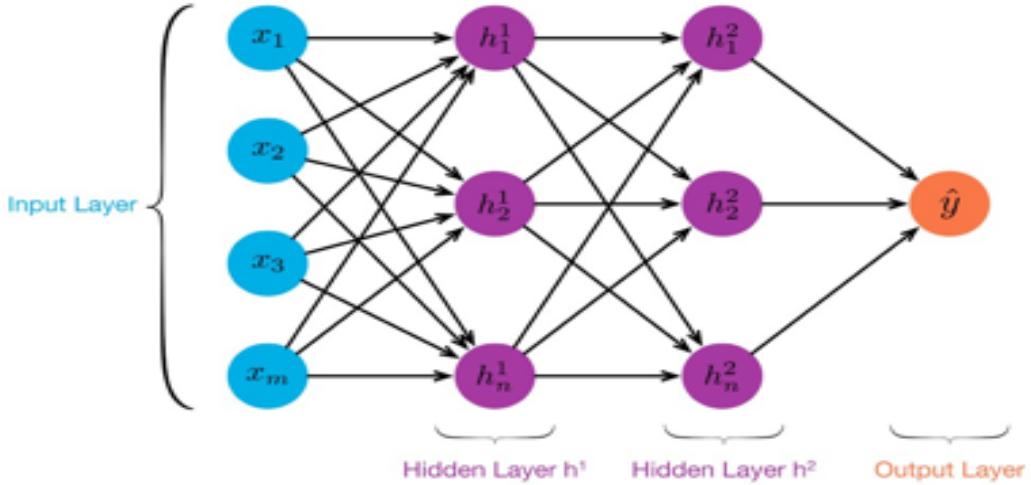
Epoch 189/200
391/391 [=====] - 61s - loss: 0.7986 - acc: 0.7862 - val_loss: 1.0336 - val_acc: 0.7416
Epoch 190/200
391/391 [=====] - 61s - loss: 0.7991 - acc: 0.7863 - val_loss: 1.0231 - val_acc: 0.7439
Epoch 191/200
391/391 [=====] - 61s - loss: 0.8003 - acc: 0.7864 - val_loss: 1.0121 - val_acc: 0.7409
- . . . . . 

Epoch 78/80
391/391 [=====] - 59s - loss: 0.4947 - acc: 0.8933 - val_loss: 0.6098 - val_acc: 0.8620
Epoch 79/80
391/391 [=====] - 59s - loss: 0.4856 - acc: 0.8960 - val_loss: 0.5644 - val_acc: 0.8738
Epoch 80/80
391/391 [=====] - 59s - loss: 0.4891 - acc: 0.8939 - val_loss: 0.6044 - val_acc: 0.8663

Epoch 178/200
391/391 [=====] - 58s - loss: 0.3130 - acc: 0.9519 - val_loss: 0.5426 - val_acc: 0.8917
Epoch 179/200
391/391 [=====] - 58s - loss: 0.3119 - acc: 0.9520 - val_loss: 0.5503 - val_acc: 0.8944
Epoch 180/200
391/391 [=====] - 58s - loss: 0.3115 - acc: 0.9537 - val_loss: 0.5501 - val_acc: 0.8913
- . . . . .

```

CHAPITRE 5: Two Layers Network



A two-layer fully-connected neural network. The net has an input dimension of N, a hidden layer dimension of H, and performs classification over C classes.

We train the network with a softmax loss function and L2 regularization on the weight matrices. The network uses a ReLU nonlinearity (In the context of artificial neural networks, the **rectifier** is an activation function defined as the positive part of its argument: $f(x) = x^+ = \max(0, x)$. A unit employing the rectifier is also called a **rectified linear unit (ReLU)**) after the first fully connected layer.

In other words, the network has the following architecture:

input - fully connected layer - ReLU - fully connected layer - softmax

The outputs of the second fully-connected layer are the scores for each class

```
def __init__(self, input_size, hidden_size, output_size, std=1e-4,
            init_method="Normal"):
```

Initialize the model. Weights are initialized to small random values and biases are initialized to zero. Weights and biases are stored in the variable `self.params`, which is a dictionary with the following keys:

W1: First layer weights; has shape (D, H)

b1: First layer biases; has shape (H,)

W2: Second layer weights; has shape (H, C)

b2: Second layer biases; has shape (C,)

Inputs:

- `input_size`: The dimension D of the input data.

- `hidden_size`: The number of neurons H in the hidden layer.

- output_size: The number of classes C.

```

self.params = {}
self.params['W1'] = std * np.random.randn(input_size, hidden_size)
self.params['b1'] = np.zeros(hidden_size)
self.params['W2'] = std * np.random.randn(hidden_size, output_size)
self.params['b2'] = np.zeros(output_size)

#special initialization
if init_method=="i":
    self.params['W1']=np.random.randn(input_size,hidden_size)/np.sqrt(input_size)
    self.params['W2']=np.random.randn(hidden_size,output_size)/np.sqrt(hidden_size)
elif init_method=="io":
    self.params['W1']=np.random.randn(input_size,hidden_size)*np.sqrt(2.0/(input_size+hidden_size))
    self.params['W2']=np.random.randn(hidden_size,output_size)*np.sqrt(2.0/(hidden_size+output_size))
elif init_method=="ReLU":
    self.params['W1']=np.random.randn(input_size,hidden_size)*np.sqrt(2.0/input_size)
    self.params['W2']=np.random.randn(hidden_size,output_size)*np.sqrt(2.0/(hidden_size+output_size))

def loss(self, X, y=None, reg=0.0, dropout=0, dropMask=None, activation='Relu'):
    """
    
```

The Next stage is computing the loss and gradients for a two layer fully connected neural network.

Inputs:

- X: Input data of shape (N, D). Each X[i] is a training sample.

- y: Vector of training labels. y[i] is the label

for X[i], and each y[i] is an integer in the range $0 \leq y[i] < C$. This parameter is optional; if it is not passed then we only return scores, and if it is passed then we instead return the loss and gradients.

- reg: Regularization strength.

Returns:

If y is None, return a matrix scores of shape (N,C) where scores[i, c] is the score for class c on input X[i].

If y is not None, instead return a tuple of:

- loss: Loss (data loss and regularization loss)for this batch of training samples.

- grads: Dictionary mapping parameter names to gradients of those parameters with respect to the loss function; has the same keys as self.params.

```

# Unpack variables from the params dictionary
W1, b1 = self.params['W1'], self.params['b1']
W2, b2 = self.params['W2'], self.params['b2']
N, D = X.shape

# Compute the forward pass
scores = None

```

TODO: We should Perform the forward pass, computing the class scores for the input.

Store the result in the scores variable, which should be an array of shape (N, C).

```

if activation=='leaky':
    inp=X.dot(W1)+b1
    a2=np.maximum(inp,.01*inp)
else:
    a2=np.maximum(X.dot(W1)+b1,0)

if dropout != 0 and dropout<1:
    a2*=(np.random.randn(*a2.shape)<dropout)/dropout
else:
    if dropout>1:
        W2*=dropMask['W2']/(dropout-1)
        b2*=dropMask['b2']/(dropout-1)

```

For convenient this is drop connect, drop rate= dropout-1

```
scores=a2.dot(W2)+b2
```

If the targets are not given then jump out, we're done

```

if y is None:
    return scores

# Compute the Loss
loss = None

```

TODO: Finish the forward pass, and compute the loss. This should include both the data loss and L2 regularization for W1 and W2. Then is storeing the result in the variable loss, which should be a scalar.

Use the Softmax

Softmax classifier, the function mapping $f(x_i; W) = Wx_i$ stays unchanged, but we now interpret these scores as the unnormalized log probabilities for each class and replace the *hinge loss* with a **cross-entropy loss** that has the form:

$$L_i = -\log \left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right) \quad \text{or equivalently} \quad L_i = -f_{y_i} + \log \sum_j e^{f_j}$$

Classifier loss. So that your results match ours, multiply the regularization loss by 0.5

```

#do a softmax first
if dropout>1:
    print dropMask['W2']
exp_scores=np.exp(scores)

a3=exp_scores/(np.sum(exp_scores,1))[:,None] #h
(x)

loss=-np.sum(np.log(a3[range(len(a3)),y]))/len(a3)
)+\
    0.5*reg*(np.sum(np.power(W1,2))+np.sum(np.power
(W2,2)))

```

Backward pass: compute gradients

```
grads = {}
```

TODO: Compute the backward pass, computing the derivatives of the weights and biases. Store the results in the grads dictionary. For example, grads['W1'] should store the gradient on W1, and be a matrix of same size

```
delta_3=a3
delta_3[range(len(a3)),y]=a3[range(len(a3)),y]-1
delta_3/=len(a3)
grads['W2']=a2.T.dot(delta_3)+reg*W2
grads['b2']=np.sum(delta_3,0)
```

```
dF=np.ones(np.shape(a2))
if activation=='leaky':
    dF[a2<0.0]=0.01
else:
    dF[a2==0.0]=0 #activation res a2 has been ReLUe
```

d

```
delta_2=delta_3.dot(W2.T)*dF
grads['W1']=X.T.dot(delta_2)+reg*W1
grads['b1']=np.sum(delta_2,0)

return loss, grads
```

```
def train(self, X, y, X_val, y_val,
          learning_rate=1e-3, learning_rate_decay=
0.95,
          reg=1e-5, num_iters=100,
          batch_size=200, verbose=False,
          update="SGD", arg=.99,
          dropout=0,
          activation='ReLU'):
```

Train this neural network using stochastic gradient descent.

Choose an initial vector of parameters and learning rate .

Repeat until an approximate minimum is obtained:

Randomly shuffle examples in the training set.

For , do:

Inputs:

- X: A numpy array of shape (N, D) giving training data.
- y: A numpy array f shape (N,) giving training labels; y[i] = c means that X[i] has label c, where 0 <= c < C.
- X_val: A numpy array of shape (N_val, D) giving validation data.
- y_val: A numpy array of shape (N_val,) giving validation labels.
- learning_rate: Scalar giving learning rate for optimization.

- learning_rate_decay: Scalar giving factor used to decay the learning rate after each epoch.
- reg: Scalar giving regularization strength.
- num_iters: Number of steps to take when optimizing.
- batch_size: Number of training examples to use per step.
- verbose: boolean; if true print progress during optimization.

```

    num_train = X.shape[0]
    iterations_per_epoch = max(num_train / batch_size
, 1)

    # Use SGD to optimize the parameters in self.mode
l
    loss_history = []
    train_acc_history = []
    val_acc_history = []
    ##### for tracking top model
    top_params=dict()
    cache_params=dict()
    top_acc=0
    cache=dict()
    dropMask=dict()
    start_time=time.time()
    #####
    for it in xrange(num_iters):
        X_batch = None
        y_batch = None

```

TODO: Create a random minibatch of training data and labels, storing them in X_batch and y_batch respectively.

```

        if num_train >= batch_size:
            rand_idx=np.random.choice(num_train,batch_siz
e)
        else:
            rand_idx=np.random.choice(num_train,batch_siz
e,replace=True)
            X_batch=X[rand_idx]
            y_batch=y[rand_idx]

        if dropout>1:
            for param in ['W2','b2']:
                dropMask[param]=np.random.randn(*self.param
s[param].shape)<(dropout-1)

```

Then for Compute loss and gradients using the current minibatch(Mini-batch gradient descent is the recommended variant of gradient descent)

```

        loss, grads = self.loss(X_batch, y=y_batch, reg
=reg, dropout=dropout, dropMask=dropMask, activation=ac
tivation)
        loss_history.append(loss)
    
```

TODO: Use the gradients in the grads dictionary to update the parameters of the network (stored in the dictionary self.params)

Using stochastic gradient descent. So we need to use the gradients stored in the grads dictionary defined above.

```

if np.isnan(grads['W1']).any() or np.isnan(grads['W2']).any() or \
np.isnan(grads['b1']).any() or np.isnan(grads['b2']).any():
    continue
#cache_params=self.params.copy()
dx=None
for param in self.params:
    if update=="SGD":
        dx=learning_rate*grads[param]
        #self.params[param]-=learning_rate*grads[pa
ram]

    elif update=="momentum":
        if not param in cache:
            cache[param]=np.zeros(grads[param].shape)
        cache[param]=arg*cache[param]-learning_rate
*grads[param]
        dx=-cache[param]
        #self.params[param]+=cache[param]

    elif update=="Nesterov momentum":
        pass
    
```

```

        if not param in cache:
            cache[param]=np.zeros(grads[param].shape)
            v_prev = cache[param] # back this up
            cache[param] = arg * cache[param] - learning_rate * grads[param] # velocity update stays the same
            dx=arg * v_prev - (1 + arg) * cache[param]
            #self.params[param] += -arg * v_prev + (1 + arg) * cache[param] # position update changes form

    elif update=="rmsprop":
        if not param in cache:
            cache[param]=np.zeros(grads[param].shape)
            cache[param]=arg*cache[param]+(1-arg)*np.power(grads[param],2)
            dx=learning_rate*grads[param]/np.sqrt(cache[param]+1e-8)
            #self.params[param]-=Learning_rate*grads[param]/np.sqrt(cache[param]+1e-8)

    elif update=="Adam":
        print "update error"

    elif update=="Adagrad":
        print "update error"

    else:
        # if have time try more update methods
        print "choose update method!"
        if dropout>1:
            if param == 'W2' or param == 'b2':
                dx*=dropMask[param]
                self.params[param]-=dx
            #Bug: learning rate should not decay at first epoch
            it+=1

        if verbose and it % 100 == 0:
            print 'iteration %d / %d: loss %f' % (it, num_iters, loss)

```

For every epoch, we must check train and val accuracy and decay learning rate. Then update the top model and params to to final top params

```

    if it % iterations_per_epoch == 0:
        # Check accuracy
        train_acc = (self.predict(X_batch) == y_batch)
    ).mean()
        val_acc = (self.predict(X_val) == y_val).mean()
    ()
        train_acc_history.append(train_acc)
        val_acc_history.append(val_acc)

        # Decay learning rate
        learning_rate *= learning_rate_decay

        ### update top model
        if val_acc > top_acc:
            top_acc = val_acc
            top_params=self.params.copy()

        if verbose:
            print ('train_acc %f, val_acc %f, time %d'
% (train_acc, val_acc,(time.time()-start_time)/60.0))

        self.params=top_params.copy()
        ### update params to top params finally

    return {
        'loss_history': loss_history,
        'train_acc_history': train_acc_history,
        'val_acc_history': val_acc_history,
    }

def predict(self, X):

```

Use the trained weights of this two-layer network to predict labels for data points. For each data point we predict scores for each of the C classes, and assign each data point to the class with the highest score.

Inputs:

- X: A numpy array of shape (N, D) giving N D-dimensional data points to classify.

Returns:

- y_pred: A numpy array of shape (N,) giving predicted labels for each of the elements of X. For all i, y_pred[i] = c means that X[i] is predicted to have class c, where $0 \leq c < C$.

```
y_pred = None
```

TODO: Implement this function

```
y_pred=np.argmax(np.maximum(0,(X.dot(self.params['W1'])+self.params['b1']))\n                    .dot(self.params['W2'])+self.params['b2'])\n                ],1)\n\n    return y_pred\n\ndef accuracy(self,X,y):
```

Use the trained model to predict labels for X, and compute the accuracy.

Inputs:

- X: A numpy array of shape (N, D) giving N D-dimensional data points to classify.
- y: A numpy array of shape (N,) giving the correct labels.

Returns:

- acc: Accuracy

```

acc = (self.predict(X) == y).mean()

return acc

def gradient_check(self,X,y):
    realGrads=dict()
    _,grads=self.loss(X,y)
    keys=[ 'W1', 'b1',
           'W2', 'b2' ]
    for key in keys:
        W1=self.params[key]
        W1_grad=[]
        delta=1e-4
        if len(np.shape(W1))==2:
            for i in range(np.shape(W1)[0]):
                grad=[]
                for j in range(np.shape(W1)[1]):
                    W1[i,j]+=delta
                    self.params[key]=W1
                    l_plus,_=self.loss(X,y)
                    W1[i,j]-=2*delta
                    self.params[key]=W1
                    l_minus,_=self.loss(X,y)
                    grad.append((l_plus-l_minus)/2.0/delta)
                    W1[i,j]+=delta
                W1_grad.append(grad)
        else:
            for i in range(len(W1)):
                W1[i]+=delta
                self.params[key]=W1
                l_plus,_=self.loss(X,y)
                W1[i]-=2*delta
                self.params[key]=W1
                l_minus,_=self.loss(X,y)
                W1_grad.append((l_plus-l_minus)/2.0/delta)
                W1[i]+=delta

    print(W1_grad)
    print(grads[key])
    print(key,"error",np.mean(np.sum(np.power((W1_grad-grads[key]),2),len(np.shape(W1))-1)\n
                                /np.sum(np.power((W1_grad+grads[key]),2),len(np.shape(W1))-1)))

```

```
quit()

dataset_dir = sys.argv[0]
start_time=time.time()

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    cifar10_dir = 'cifar-10-batches-py'
    print cifar10_dir
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)
    mask = range(num_training, num_training + num_validation)
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = range(num_training)
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = range(num_test)
    X_test = X_test[mask]
    y_test = y_test[mask]
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_train=X_train.swapaxes(1,3)
    X_val=X_val.swapaxes(1,3)
    X_test=X_test.swapaxes(1,3)
    return X_train, y_train, X_val, y_val, X_test, y_test
```

Then is Invoke the above function to get our data.

```
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print "finish loading"
print 'Train data : ', X_train.shape
print 'Validation data : ', X_val.shape
print 'Test data: ', X_test.shape
print "Time", (time.time()-start_time)/60.0
rfSize = 6
numCentroids=1600
whitening=True
numPatches = 400000
CIFAR_DIM=[32,32,3]
```

```
#create unsupervised data
patches=[]
for i in range(numPatches):
    if(np.mod(i,10000) == 0):
        print "sampling for Kmeans",i,"/",numPatches
    start_r=np.random.randint(CIFAR_DIM[0]-rfSize)
    start_c=np.random.randint(CIFAR_DIM[1]-rfSize)
    patch=np.array([])
    img=X_train[np.mod(i,X_train.shape[0])]
    for layer in img:
        patch=np.append(patch,layer[start_r:start_r+r
fSize].T[start_c:start_c+rfSize].T.ravel())
    patches.append(patch)
patches=np.array(patches)
```

For normalizing patches:

```
patches=(patches-patches.mean(1)[:,None])/np.sqrt(patches.var(1)+10)[:,None]
print "time", (time.time()-start_time)/60.0
```

For csil

```
del X_train, y_train, X_val, y_val, X_test, y_test
```

The next step is whitening

```

print "whitening"
[D,V]=np.linalg.eig(np.cov(patches, rowvar=0))

P = V.dot(np.diag(np.sqrt(1/(D + 0.1)))).dot(V.T)
patches = patches.dot(P)

print "time", (time.time()-start_time)/60.0
del D,V

centroids=np.random.randn(numCentroids,patches.shape[1])*1
num_iters=50
batch_size=1000#CSIL do not have enough memory, dam
for ite in range(num_iters):
    print "kmeans iters",ite+1,"/",num_iters

c2=.5*np.power(centroids,2).sum(1) idx=np.argmax(patches.dot(centroids.T)-c2, axis=1)
x2 the same omit
    hf_c2_sum=.5*np.power(centroids,2).sum(1)
    counts=np.zeros(numCentroids)
    summation=np.zeros_like(centroids)
    for i in range(0,len(patches),batch_size):
        last_i=min(i+batch_size,len(patches))
        idx=np.argmax(patches[i:last_i].dot(centroids
.T) -hf_c2_sum.T, axis=1)
        if i==1:
            S=np.zeros([last_i-i,numCentroids])
            S[range(last_i-i),
             np.argmax(patches[i:last_i].dot(centroids.T
)-hf_c2_sum.T
                ,axis=1)]=1
            summation+=S.T.dot(patches[i:last_i])
            counts+=S.sum(0)
        centroids=summation/counts[:,None]
        centroids[counts==0]=0

```

some centroids didn't get members, divide by zero the thing is, they will stay zero forever

```
print "time", (time.time()-start_time)/60.0
```

```

def sliding(img,window=[6,6]):
    out=np.array([])
    for i in range(3):
        s=img.shape
        row=s[1]
        col=s[2]
        col_extent = col - window[1] + 1
        row_extent = row - window[0] + 1
        start_idx = np.arange(window[0])[:,None]*col
+ np.arange(window[1])
        offset_idx = np.arange(row_extent)[:,None]*co
l + np.arange(col_extent)
        if len(out)==0:
            out=np.take (img[i],start_idx.ravel()[:,N
one] + offset_idx.ravel())
        else:
            out=np.append(out,np.take (img[i],start_i
dx.ravel()[:,None] + offset_idx.ravel()),axis=0)
    return out

```

For extracting features first we normalized data and map the feature space then with using $x_2-2xc+c2$ we could calculate the distance

```

def extract_features(X_train):
    trainXC=[]
    idx=0
    for img in X_train:
        idx+=1
        if not np.mod(idx,1000):
            print "extract features",idx,'/',len(X_tr
ain)
            print "time", (time.time()-start_time)/60.
0
        patches=sliding(img,[rfSize,rfSize]).T
        #normalize
        patches=(patches-patches.mean(1)[:,None])/(
np
.sqrt(patches.var(1)+10)[:,None])
        #map to feature space
        patches=patches.dot(P)
        #calculate distance using  $x_2-2xc+c2$ 
        x2=np.power(patches,2).sum(1)
        c2=np.power(centroids,2).sum(1)
        xc=patches.dot(centroids.T)

```

```

    dist=np.sqrt(-2*xc+x2[:,None]+c2)
    u=dist.mean(1)
    patches=np.maximum(-dist+u[:,None],0)
    rs=CIFAR_DIM[0]-rfSize+1
    cs=CIFAR_DIM[1]-rfSize+1
    patches=np.reshape(patches,[rs,cs,-1])
    q=[]
    q.append(patches[0:rs/2,0:cs/2].sum(0).sum(0))
    q.append(patches[0:rs/2,cs/2:cs-1].sum(0).sum(0))
    q.append(patches[rs/2:rs-1,0:cs/2].sum(0).sum(0))
    q.append(patches[rs/2:rs-1,cs/2:cs-1].sum(0).sum(0))
    q=np.array(q).ravel()
    trainXC.append(q)
trainXC=np.array(trainXC)
trainXC=(trainXC-trainXC.mean(1)[:,None])/(np.sqrt((trainXC.var(1)+.01)[:,None]))
return trainXC

X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
# In[112]:
trainXC=extract_features(X_train)
print "time", (time.time()-start_time)/60.0
valXC=extract_features(X_val)

testXC=extract_features(X_test)

```

The next step is importing cPickle as pickle with open("features.pickle","w") as f:
 pickle.dump([trainXC,valXC,testXC,y_train,y_val,y_test],f)

```

from neural_net import *

input_size = trainXC.shape[1]
hidden_size = 200
num_classes = 10

net = TwoLayerNet(input_size, hidden_size, num_classes,
                    reg=1e-4)
stats = net.train(trainXC, y_train, valXC, y_val,
                   num_iters=70000, batch_size=128,
                   learning_rate=5e-4, learning_rate_decay=0.99,
                   reg=0, verbose=True, update_rule="momentum",
                   arg=0.95, dropout=0.3)

val_acc = (net.predict(trainXC) == y_train).mean()
print 'Train accuracy: ', val_acc
val_acc = (net.predict(valXC) == y_val).mean()
print 'Validation accuracy: ', val_acc

val_acc = (net.predict(testXC) == y_test).mean()
print 'Test accuracy: ', val_acc

```

Plot the loss function and train / validation accuracies

```

plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()

plt.savefig("dropout loss_history.eps")
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.show()

plt.ylabel('Classification accuracy')
plt.savefig('dropout accuracy.eps')

```

References:

1. <http://cs231n.github.io/linear-classify/>
2. https://en.wikipedia.org/wiki/Stochastic_gradient_descent
3. Network In Network (Min Lin_{1,2}, Qiang Chen₂, Shuicheng Yan₂)
4. <http://www.doc.ic.ac.uk/~sgc/teaching/pre2012/v231/lecture12.html>