Study in image classification using nearest neighbors and perceptrons

Jiexiang Qin
Lake highland preparatory school, Orlando, FL 32803, US.

Abstract: In this paper, we describe our experiments on training a computer to recognize (classify) a black and white image of hand-written digits. The experiments were done using Python. We try two methods to achieve the goal. The first method is k Nearest Neighbors, and the second is a Perceptron (modified to work for more than two classes). The paper describes each of these in detail, including some important implementation details, and reports results for these two methods, as well as some observations about their behavior.

Keywords: Black and White Image Of Hand-Written Digits**;** Classify; Pixels; Classification; Learning; Database; Epoch; Training; Perceptron; Accuracy; Percentage; Implementation; Matches; Terminates; Algorithm

Introduction

This particular python project can help computers process, store, and assign an input picture to one of a set of categories or types. In this case, the types are digits from 0 to 9, and the images contain handwritten examples of these digits. Every imported image is a two-dimensional array of numbers that has 28x28 pixels; each pixel is represented by a number between 0 to 255; the closer the number to 255, the darker the pixels are.

The main experiments involve the “nearest neighbor” classification method. This program will receive an imported picture, compare it to our database that comes from 100 default pictures that contain examples of images with the known correct label (class)", and return the index of the picture that best matches the

imported picture. The implementation measures the similarity of two images as sum (|X-Y|), where |X-Y| is the element-wise absolute difference between pixel values in the two images, and sum () is the sum of the values. In other words, we find the picture with the most pixels in common with the input. The other way used is by detecting how different the import picture is from our pictures in the database, storing the difference, and calculating the most similar one.

We also experimented with another method, the Perceptron. It works differently by learning a single set of numbers ("weights") for each class so that the inner product (scalar product) of those weights with the image is highest for the correct class. We talk about it in more detail below.

1. Implementation and algorithms

The program starts with creating a database. Inside the database, there are N different images. Each image is a number between zero and nine and has 28x28 pixels. Each pixel is a number between 0 to 255; 0 means black, 255 means white, 128 is the gray in the middle; etc. N in our experiments could be as small as 100 (10 images per class) or as large as 10,000 per class.We also have a test set, consisting of 10 images per class that are not included in the database; we use these images only to evaluate various methods and report results.

1.1 K nearest neighbors classification

Then we have the imCompare method that will use two for loops to compare the number of brightness and accumulate the amount of brightness differences. For every pixel, we compute the absolute brightness difference (abs of the difference) and sum this up over all the pixels. This gives us a number which is high if the two images are very different, and low if they are similar (zero if they are identical).

The main approach works by finding k nearest neighbors (k-NN) for the test images: the k images in the database that are most similar to the test image (has the lowest value of imCompare as explained above). We omit the details of how this is implemented, other than say that it's done by exhaustive search over the database, while maintaining the list of the k-NN found so far, and updating it whenever we find a database example that is closer to the test image than at least one of the k-NN found so far. The function implementing this is called kNNDiff.

After all the setup is done, we will call a function that receive the data base, the test data and k to use in the k-NN as explained above. We called kNNDiff and that function will help us to find the best k matches. Last we will call a function findMajority that return the best answer according to the result we get from kNNDiff. Specifically, it looks up the known labels for the k-NN examples, and takes the label that the majority of them have.

```
def kNNDiff(xTest,DB,k):
    Best=[np.inf]*k  # List of Best values
    index=[0]*k # This is where we can find the best values
    WorstOfTheBest=0
    for i in range(len(DB)):
        # sim=imCompareDiff(DB[i][0],xTest) # Sim will be small if image is similar
        sim=np.abs(DB[i][0]-xTest).sum()
        if sim<Best[WorstOfTheBest]:
            Best[WorstOfTheBest]=sim
            index[WorstOfTheBest]=i
            for j in range(k):
                if(Best[j]>Best[WorstOfTheBest]):
                    WorstOfTheBest=j


    return index,Best


def findMajority(index,DB):

    count=[0]*10
    for i in index:
        count[DB[i][1]]+=1
    indexL=0
    for i in range(len(count)):
        if count[indexL]<count[i]:
            indexL=i
    return indexL
```

1.2 Classification by Perceptron

The second method is programming a Perceptron that will tell what number is appearing on the picture. First we create a function that will make one database with all the training images from the given two classes. And each element of the database is a tuple of (image, label) where the image is an array. Then we program a "perceptronTrain" function that trains a classifier.

The 2-class perceptron has the form sign(w*x+b). Here w is a set of numbers of the same size as the image x. this means treating the 2D PxP pixel image x like a vector of length P^2 -- so a 28x28 image is interpreted as a 784-dimensional vector, and then x and w each have 784 numbers. * denotes inner product (also known as scalar or dot product; we use np.dot to compute it, from the NumPy package). Finally b is a single number which allows us to offset the dot product (it can be positive or negative). The sign function converts the offset value of the dot product to a positive or negative sign.

However, we have 10 classes. So, we modify the classifier as follows. We have a (w_c,b_c) pair for each class c of the 10 classes. Then, given an image x that we need to classify, we compute a “score” for each class c as w_c*x+b_c. Finally, we choose the class with the highest score and assign it as the predicted label for x.

The training of this modified Perceptron is similar to the training of the original, but with a modification as well. It works as follows: Each time we pick an image x from the database (for which we know the correct label y), and try to predict the label with our current perceptron. If the prediction is correct we move on to the next image. If not, we update w’s and b’s: we increase the w_y and b_y for the correct y, and decrease them for the incorrect one.

```
def perceptronTrain(data, mistakesFraction=0.02, maxEpoch=10, lrDrop=0.95):

    numpix = np.prod(data[0][0].shape) # how many pixels in an image
    N = len(data) # how many images
    nclass=10 # could try to infer from data as # of unique labels, but will for now hard code

    # initialize w, b
    w = np.zeros((nclass, numpix))
    b = np.zeros(nclass)

    learningRate=1.0

    for nEpoch in range(maxEpoch):

        # code for one epoch:
        mistakes=0
        for i in range(len(data)):
            yhat = predict(data[i][0], w, b) # note: prediction could be any of class labels
            if yhat != data[i][1]:
                mistakes += 1

                # update the classifier
                # update two sets of parameters: make the params for the correct class have higher score
                w[data[i][1]] = w[data[i][1]] + learningRate*data[i][0].flatten()
                b[data[i][1]] = b[data[i][1]] + learningRate*data[i][1]

                # update the params for the wrongly predicted class to make the score of that class lower
                w[yhat] = w[yhat] - learningRate*data[i][0].flatten()
                b[yhat] = b[yhat] - learningRate*data[i][1]

        print('Done with epoch %d,  %d mistakes; LR=%.4f' % (nEpoch+1, mistakes, learningRate) )
        if mistakes < N*mistakesFraction:
            print('we are done')
            break # stop the training
        learningRate=learningRate*lrDrop

    return w,b
```

After an epoch (full pass once over all the images in the database), the program checks how many mistakes were made in this epoch; if the number of mistakes is less than the threshold, we are done, otherwise do another epoch. Furthermore, the user will set the max epoch to prevent the program from running forever. Lastly, we have the "evalPerceptron" function, which calculates the accuracy of the trained perceptron on a set of images (we use the test set of course)

```
Done with epoch 1,  8075 mistakes; LR=1.0000
Done with epoch 2,  6676 mistakes; LR=0.9000
Done with epoch 3,  6218 mistakes; LR=0.8100
Done with epoch 4,  6067 mistakes; LR=0.7290
Done with epoch 5,  5953 mistakes; LR=0.6561
Done with epoch 6,  5734 mistakes; LR=0.5905
Done with epoch 7,  5708 mistakes; LR=0.5314
Done with epoch 8,  5553 mistakes; LR=0.4783
Done with epoch 9,  5503 mistakes; LR=0.4305
Done with epoch 10,  5326 mistakes; LR=0.3874
Done with epoch 11,  5273 mistakes; LR=0.3487
Done with epoch 12,  5184 mistakes; LR=0.3138
Done with epoch 13,  5017 mistakes; LR=0.2824
Done with epoch 14,  5081 mistakes; LR=0.2542
Done with epoch 15,  4950 mistakes; LR=0.2288
Done with epoch 16,  4914 mistakes; LR=0.2059
Done with epoch 17,  4810 mistakes; LR=0.1853
Done with epoch 18,  4730 mistakes; LR=0.1668
Done with epoch 19,  4729 mistakes; LR=0.1501
```

2. Experiments and results

2.1 Results for k-NN:

We tried different k and different N (# of images in the training database). The table below shows the results; we highlight the best accuracy (in percentage of test images) for each N.

#training images	k=1	3 5 7 10	25
#100	68%	72%79% 79% 69%	79%

#1000	65%	74% 80% 83% 72%	83%
#1000	70%	75% 85% 84% 74%	84%
#3000	67%	77%86% 82% 70%	82%

#20000 70% 74% 85% 84% 84% 84% #50000 71% 76% 80% 82%82% 82%

```
def testImageByDiff(xTest,DB,threshold):
    smallest=0
    index=0
    for i in range(len(DB)):
        sim=imCompareDiff(DB[i][0],xTest,threshold)
        if sim>smallest:
            smallest=sim
            index=i
    return DB[index][1],index

def evaluate(DB,testDB,k):
    correct=0
    for i in range(len(testDB)):
        xTest=testDB[i][0]
        index,_=kNNDiff(xTest,DB,k)
        prediction=findMajority(index,DB)
        if prediction==testDB[i][1]:
            correct+=1
    return correct/len(testDB)
```

According to the result, as the number of training images increases, the correctness percentage also tends to increase. This is because there are more images to compare to so the best one will be more accurate. We are not sure why the largest N (50000) actually has slightly worse results. For a given N as values of k increase, the accuracy increases too, but then it flattens or drops off. The best k seems to be 5 or 7. We believe it's because with too few samples (such as k=1 or k=3) the matches are more likely to be wrong, creating noise in the labels. If k is too big, there are probably too many unreliable matches (which are not quite similar to the test image?) so again we have more noisy labels.

2.2 Results for the Perceptron

We noticed that two steps are very important to get the perceptron to work. First, we must shuffle the database -- randomly permute the order of the images. Otherwise, if all the 0s are together first, then all the 1s, etc., then the algorithm makes very few updates and terminates, even though it has not really learned well.

The other step is to use a "learning rate" -- multiply the updates by a number smaller than one. Without it, the updates are very chaotic and it takes longer for the algorithm to finish (and it seems to learn less well).

3. Discussion and conclusions

According to the data analyses, both Perceptron and kNN classifiers have advantages and disadvantages. The advantage of a Perceptron is that it is more precise, and accurate, and it's faster at test time (faster to classify an image) But of course, it is slower to train -- since you have to actually train it, and there is no training for kNN. Worth spelling this out explicitly., but on the other hand side, it is challenging to replicate a perceptron. For kNN classifiers, it is much easier to understand and replicate, but when the

data gets bigger, it will take minutes or even hours to run through every picture.

References

[1] Verma, S. (2021, April 17). *Implementing the perceptron algorithm in Python*. Medium. Towards Data Science.

[2] Rosebrock A. (2021, May 12). *Implementing the perceptron neural network with python*. PyImageSearch.

[3] Corrigan, M. (2016). An introduction to python machine learning with Perceptrons. Codementor.

[4] Paolo D'Elia, Python-Perceptron, Retrieved from: https://pypi.org/project/PyPerce ptron/.