

最近在刷cs231n的课程和作业，在这里分享下自己的学习过程，同时也希望能够得到大家的指点。

写在前面：

1. 这仅仅是自己的学习笔记，如果侵权，还请告知;
2. 代码是参照[huyouare的github](#)，在其基础之上做了一些修改;
3. 在我之前，已有前辈在他的[知乎](#)上分享过类似内容;
4. 讲义是参照[杜客](#)等人对cs231n的中文翻译。

温馨提醒：

1. 这篇文档是建立在你已经知道knn算法的基本原理，如果在阅读本篇文档过程中有些原理不是很清楚，请移步温习下相关知识;
2. 文档的所有程序是使用python3.5实现的，如果你是python2的用户，可能要对代码稍作修改。

完成cifar10的分类工作，主要有三个步骤：

1. 模型构建；
2. 数据载入；
3. 训练和预测。

下面我们就直接进入正题，用knn算法实现对cifar10图像的分类

1 构建模型

模型的构建过程主要有训练模型和预测模型两个阶段，我们分别对这两个阶段进行分析。

1.1 训练模型

模型的训练也就是模型的参数学习过程，由于knn算法并不具有显示的学习过程，所以只需要将训练数据载入即可。

1.2 预测模型

实现knn算法的预测需要完成以下两个工作：

1. 计算测试集每张图片的每个像素点与训练集每张图片的每个像素点的距离，本文采用的是欧氏距离；

2. 将距离排序，选取前k个最小距离的类别输出。

下面是模型构建的完整代码：

```
1 import numpy as np
2
3 class KNearestNeighbor:
4
5     def __init__(self):
6         pass
7     def train(self,X,y):
8         self.X_train=X
9         self.y_train=y
10
11     def predict(self,X,k=1,num_loops=0):    #1
12         if num_loops== 0:
13             dists=self.compute_distances_no_loops(X)
14         elif num_loops==1:
15             dists=self.compute_distances_one_loop(X)
16         elif num_loops==2:
17             dists=self.compute_distances_two_loops(X)
18         else:
19             raise ValueError('Invalid value %d for num_loops' %
num_loops)
20         return self.predict_labels(dists,k=k)
21
22     def compute_distances_two_loops(self,X):
23         num_test=X.shape[0]
24         num_train=self.X_train.shape[0]
25         dists=np.zeros((num_test,num_train))
26         print(X.shape,self.X_train.shape)
27         for i in range(num_test):
28             for j in range(num_train):
29                 dists[i,j]=np.sqrt(np.sum((X[i,:]-self.X_train[j,:])**2))
30         return dists
31
32     def compute_distances_one_loop(self,X):
33         num_test=X.shape[0]
34         num_train=self.X_train.shape[0]
35         dists=np.zeros((num_test,num_train))
36         for i in range(num_test):
37             dists[i,:]=np.sqrt(np.sum(np.square(self.X_train-
X[i,:]),axis=1))
38         return dists
39
40     def compute_distances_no_loops(self,X):
```

```

41     num_test = X.shape[0]
42     num_train = self.X_train.shape[0]
43     dists = np.zeros((num_test, num_train))
44     test_sum=np.sum(np.square(X),axis=1)
45     train_sum=np.sum(np.square(self.X_train),axis=1)
46     inner_product=np.dot(X,self.X_train.T)
47     dists=np.sqrt(-2*inner_product+test_sum.reshape(-1,1)+train_sum)
48     return dists
49
50     def predict_labels(self,dists,k=1):    #2
51         num_test=dists.shape[0]
52         y_pred=np.zeros(num_test)
53         for i in range(num_test):
54             closest_y=[]
55             y_indicies=np.argsort(dists[i,:],axis=0)    #2.1
56             closest_y=self.y_train[y_indicies[: k]]    #2.2
57             y_pred[i]=np.argmax(np.bincount(closest_y))    #2.3
58         return y_pred

```

代码解释：#1 是欧氏距离的三种不同实现方式，稍后我们会比较这三种方式的效率。其中 `X` 表示测试集数据，`k` 表示需要选取的近邻点个数（默认是1），`num_loops` 表示采用哪种方式计算欧氏距离（默认采用第三种），最后得到的距离存储在 `dists` 中。#2 是将距离排序，输出与测试集距离最小的前k个训练集图像类别。#2.1 将距离按照从小到大的顺序排列，输出序号；#2.2 输出前k个图像的类别；#2.3 对得到的k个数进行投票，选取出现次数最多的类别作为最后的预测类别，当k=1时，`closest_y = y_pred`









至此，我们便完成了对knn模型的构建工作，我们将这部分代码另存为 `knn.py` 文件，方便训练时候的调用。接下来需要载入数据。

2 数据载入

我们采用的是cifar10数据集，该数据集被分成5份训练集和1份测试集，其中每份有1000张32*32的RGB图，共有10类。

我们接下来将这个5份训练集和1份测试集分别转成数组的形式，为后续的训练工作做准备。

将数据下载下来，我们发现，这不是图片数据，因此我们需要按照官网的要求使用 `pickle` 模块。官网下载数据集格式如图：

 batches.meta	2009/3/31 12:45	META 文件	1 KB
 data_batch_1	2009/3/31 12:32	文件	30,309 KB
 data_batch_2	2009/3/31 12:32	文件	30,308 KB
 data_batch_3	2009/3/31 12:32	文件	30,309 KB
 data_batch_4	2009/3/31 12:32	文件	30,309 KB
 data_batch_5	2009/3/31 12:32	文件	30,309 KB
 readme	2009/6/5 4:47	HTML 文件	1 KB
 test_batch	2009/3/31 12:32	文件	30,309 KB

下面是这部分的完整代码：

```

1  import pickle
2  import numpy as np
3  import os
4
5  def load_cifar_batch(filename):
6      with open(filename, 'rb') as f :
7          datadict=pickle.load(f,encoding='bytes')
8          x=datadict[b'data']
9          y=datadict[b'labels']
10         x=x.reshape(10000,3,32,32).transpose(0,2,3,1).astype('float')
11         y=np.array(y)
12         return x,y
13
14 def load_cifar10(root):
15     xs=[]
16     ys=[]
17     for b in range(1,6):
18         f=os.path.join(root, 'data_batch_%d' % (b,))
19         x,y=load_cifar_batch(f)
20         xs.append(x)
21         ys.append(y)
22     Xtrain=np.concatenate(xs) #1
23     Ytrain=np.concatenate(ys)
24     del x ,y
25     Xtest,Ytest=load_cifar_batch(os.path.join(root, 'test_batch')) #2
26     return Xtrain,Ytrain,Xtest,Ytest

```

代码解释：#1 将5份训练集转成数组。#2 将1份测试集转成数组。

至此，我们便完成了数据的转化工作，为了方便后续的工作，我们将这份这份代码另存为 `data_utils.py` 接下来就要进行模型的训练和预测。

3 训练和预测

训练部分主要有三个过程：

1. 将数据集载入模型；
2. 对测试集进行预测；
3. 交叉验证选取最优的k值。

我们分别对这三个过程进行分析。

3.1 将数据集载入模型

下面是载入数据的代码：

```
1 import numpy as np
2 from data_utils import load_cifar10
3 import matplotlib.pyplot as plt
4 from knn import KNearestNeighbor
5 x_train,y_train,x_test,y_test=load_cifar10('cifar-10-batches-py')
```

为了验证我们的结果是否正确，我们可以打印输出下

```
1 print('training data shape:',x_train.shape)
2 print('training labels shape:',y_train.shape)
3 print('test data shape:',x_test.shape)
4 print('test labels shape:',y_test.shape)
```

结果如下：

```
1 training data shape: (50000, 32, 32, 3)
2 training labels shape: (50000,)
3 test data shape: (10000, 32, 32, 3)
4 test labels shape: (10000,)
```

如同我们第二章叙述的，共有50000张训练集，10000张测试集。

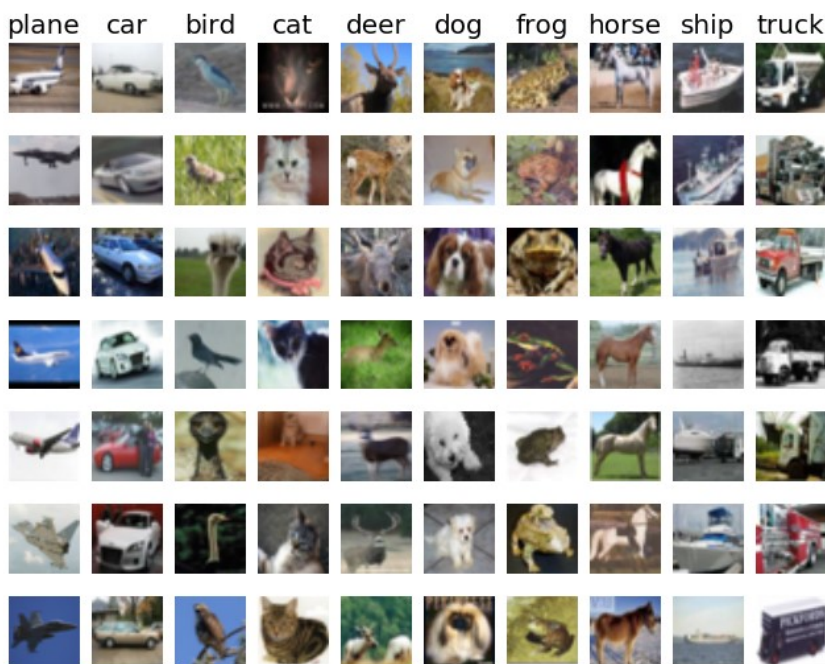
下面我们从这50000张训练集每一类中随机挑选 `samples_per_class` 张图片进行展示，代码如下：

```

1 classes=
  ['plane','car','bird','cat','deer','dog','frog','horse','ship','truck']
2 num_claesses=len(classes)
3 samples_per_class=7
4 for y ,cls in enumerate(classes):
5     idxs=np.flatnonzero(y_train==y)
6     idxs=np.random.choice(idxs,samples_per_class,replace=False)
7     for i ,idx in enumerate(idxs):
8         plt_idx=i*num_claesses+y+1
9         plt.subplot(samples_per_class,num_claesses,plt_idx)
10        plt.imshow(x_train[idx].astype('uint8'))
11        plt.axis('off')
12        if i ==0:
13            plt.title(cls)
14 plt.show()

```

结果如下图：



为了加快我们的训练速度，我们只选取5000张训练集，500张测试集（读者可之后使用全部数据集进行训练和测试），代码如下：

```

1 num_training=5000
2 mask=range(num_training)
3 x_train=x_train[mask]
4 y_train=y_train[mask]
5 num_test=500

```

```
6 mask=range(num_test)
7 x_test=x_test[mask]
8 y_test=y_test[mask]
```

至此，数据载入部分已经算是完成了，但是为了欧氏距离的计算，我们把得到的图像数据拉长成行向量，代码如下：

```
1 x_train=np.reshape(x_train,(x_train.shape[0],-1))
2 x_test=np.reshape(x_test,(x_test.shape[0],-1))
3 print(x_train.shape,x_test.shape)
```

得到的结果是：

```
1 (5000, 3072) (500, 3072)
```

也就是说，原来(32,32,3)的图片转化成了(3072,)，其中 $3072=32 * 32 * 3$

下面就是对测试集进行预测部分了。

3.2 测试集预测

为了能够预测每个图像类别，我们首先要计算每个测试集图像与训练集图像的欧氏距离，计算代码如下：

```
1 classifier=KNearestNeighbor()
2 classifier.train(x_train,y_train)
3 dists=classifier.compute_distances_two_loops(x_test)
4 print(dists)
```

注意，这里我们采用的是其中一种欧氏距离计算方式：`compute_distances_two_loops`。

可以得到计算结果如下：

```
1 [[ 3803.92350081 4210.59603857 5504.0544147 ..., 4007.64756434
2  4203.28086142 4354.20256764]
3  [ 6336.83367306 5270.28006846 4040.63608854 ..., 4829.15334194
4  4694.09767687 7768.33347636]
5  [ 5224.83913628 4250.64289255 3773.94581307 ..., 3766.81549853
6  4464.99921613 6353.57190878]
7  ...,
```

```

8 [ 5366.93534524 5062.8772452 6361.85774755 ..., 5126.56824786
9 4537.30613911 5920.94156364]
10 [ 3671.92919322 3858.60765044 4846.88157479 ..., 3521.04515734
11 3182.3673578 4448.65305458]
12 [ 6960.92443573 6083.71366848 6338.13442584 ..., 6083.55504619
13 4128.24744898 8041.05223214]]

```

距离得出之后，就可以预测测试集的分类了，代码如下：

```

1 y_test_pred = classifier.predict_labels(dists, k=1)

```

这里我们设置的 `k=1`，也就是最近邻。

那么我们如何评判我们得到的预测结果是否正确呢？模型评估也是机器学习中的一个重要概念，这里我们使用准确率作为模型的评价指标，代码如下：

```

1 num_correct = np.sum(y_test_pred == y_test)
2 accuracy = float(num_correct) / num_test
3 print 'Got %d / %d correct => accuracy: %f' % (num_correct, num_test,
    accuracy)

```

可以得到预测的准确率，如下：

```

1 got 137 / 500 correct => accuracy: 0.274000

```

这里的 `137` 代表预测正确的个数，27%的准确率不是很好，但这只是我们的第一个算法，后面做到深度卷积网络时候，准确率会得到大幅度提升。

至此，模型的训练和预测过程已经大体完成，但是在第一章叙述中，我们说有三种计算欧氏距离的方式，上文只是采用了其中一种，那么剩下采用剩下两种的效率是不是会有所提升呢？我们来实验比较下。

采用 `compute_distances_one_loop` 的计算方式，代码如下：

```

1 dists_one=classifier.compute_distances_one_loop(x_test)
2 difference=np.linalg.norm(dists-dists_one,ord='fro')
3 print('difference was: %f' % difference)

```

输出结果为：

```

1 difference was: 0.000000

```


采用 `compute_distances_np_loops` 的计算方式，同上，也可得出同样的结果（读者可自行验证）：

```
1 difference was: 0.000000
```

既然三种方法得到的答案是一样的，那么我采用任意一种方法计算都是可以的吧？答案是肯定的，这三种方法的差别在于它们的计算时间不同，我们来做下比较。比较代码如下：

```
1 def time_function(f,*args):
2     tic=time.time() ← import time
3     f(*args)
4     toc=time.time()
5     return toc-tic
6
7 two_loop_time=time_function(classifier.compute_distances_two_loops,x_test)
8 print('two loops version took %f seconds' % two_loop_time)
9
10 one_loop_time=time_function(classifier.compute_distances_one_loop,x_test)
11 print('one loop version took %f seconds' % one_loop_time)
12
13 no_loops_time=time_function(classifier.compute_distances_no_loops,x_test)
14 print('no loops version took %f seconds' % no_loops_time)
```

得到的输出结果如下：

```
1 two loops version took 31.083703 seconds
2 one loop version took 79.102377 seconds
3 no loops version took 1.464427 seconds
```

因此，我们可以发现，同样的结果，但是计算所花费的时间却是不一样的，明显 `compute_distances_no_loops` 所花费的时间更少。在时间就是生命的当下，选择一个快速计算的算法显得尤为重要。

至此，模型训练和预测过程已经全部完成，但是这里还有一个超参数 `k` 是可以修改的，不同的 `k` 是不是会得到不一样的准确率呢？我们将在下一节进行分析。

3.3 交叉验证

在机器学习中，当数据量不是很充足时，交叉验证是一种不错的模型选择方法（深度学习数据量要求很大，一般是不采用交叉验证的，因为它**太费时间**），本节我们就利用交叉验证来选择最好的 **k** 值来获得较好的预测的准确率。

这里，我们采用S折交叉验证的方法，即将数据平均分成S份，一份作为测试集，其余作为训练集，一般S=10，本文将S设为5，即代码中的 **num_folds=5**。

完整代码如下：

```
1 num_folds=5
2 k_choices=[1,3,5,8,10,12,15,20,50,100]
3 x_train_folds=[]
4 y_train_folds=[]
5
6 y_train=y_train.reshape(-1,1)
7 x_train_folds=np.array_split(x_train,num_folds) #1
8 y_train_folds=np.array_split(y_train,num_folds)
9
10 k_to_accuracies={} #2
11
12 for k in k_choices:
13     k_to_accuracies.setdefault(k,[])
14 for i in range(num_folds): #3
15     classifier=KNearestNeighbor()
16     x_val_train=np.vstack(x_train_folds[0:i]+x_train_folds[i+1:]) #3.1
17     y_val_train = np.vstack(y_train_folds[0:i] + y_train_folds[i + 1:])
18     y_val_train=y_val_train[:,0]
19     classifier.train(x_val_train,y_val_train)
20     for k in k_choices:
21         y_val_pred=classifier.predict(x_train_folds[i],k=k) #3.2
22         num_correct=np.sum(y_val_pred==y_train_folds[i][:,0])
23         accuracy=float(num_correct)/len(y_val_pred)
24         k_to_accuracies[k]=k_to_accuracies[k]+[accuracy]
25
26 for k in sorted(k_to_accuracies): #4
27     sum_accuracy=0
28     for accuracy in k_to_accuracies[k]:
29         print('k=%d, accuracy=%f' % (k,accuracy))
30     sum_accuracy+=accuracy
31     print('the average accuracy is :%f' % (sum_accuracy/5))
```

代码解释：#1 表示将数据集平均分成5份；#2 以字典形式存储 **k** 和 **accuracy**；#3 对每个 **k** 值，选取一份测试，其余训练，计算准确率；#3.1表示除i之外的作为训练集；#3.2第i份作为测试集并预测；#4 表示输出每次得到的准确率以及每个k值对应的平均准确率。

计算结果如下：

```
1 k=1, accuracy=0.263000
2 k=1, accuracy=0.257000
3 k=1, accuracy=0.264000
4 k=1, accuracy=0.278000
5 k=1, accuracy=0.266000
6 the average accuracy is :0.265600
7 k=3, accuracy=0.239000
8 k=3, accuracy=0.249000
9 k=3, accuracy=0.240000
10 k=3, accuracy=0.266000
11 k=3, accuracy=0.254000
12 the average accuracy is :0.249600
13 k=5, accuracy=0.248000
14 k=5, accuracy=0.266000
15 k=5, accuracy=0.280000
16 k=5, accuracy=0.292000
17 k=5, accuracy=0.280000
18 the average accuracy is :0.273200
19 k=8, accuracy=0.262000
20 k=8, accuracy=0.282000
21 k=8, accuracy=0.273000
22 k=8, accuracy=0.290000
23 k=8, accuracy=0.273000
24 the average accuracy is :0.276000
25 k=10, accuracy=0.265000
26 k=10, accuracy=0.296000
27 k=10, accuracy=0.276000
28 k=10, accuracy=0.284000
29 k=10, accuracy=0.280000
30 the average accuracy is :0.280200
31 k=12, accuracy=0.260000
32 k=12, accuracy=0.295000
33 k=12, accuracy=0.279000
34 k=12, accuracy=0.283000
35 k=12, accuracy=0.280000
36 the average accuracy is :0.279400
37 k=15, accuracy=0.252000
38 k=15, accuracy=0.289000
39 k=15, accuracy=0.278000
40 k=15, accuracy=0.282000
41 k=15, accuracy=0.274000
42 the average accuracy is :0.275000
43 k=20, accuracy=0.270000
```

```

44 k=20, accuracy=0.279000
45 k=20, accuracy=0.279000
46 k=20, accuracy=0.282000
47 k=20, accuracy=0.285000
48 the average accuracy is :0.279000
49 k=50, accuracy=0.271000
50 k=50, accuracy=0.288000
51 k=50, accuracy=0.278000
52 k=50, accuracy=0.269000
53 k=50, accuracy=0.266000
54 the average accuracy is :0.274400
55 k=100, accuracy=0.256000
56 k=100, accuracy=0.270000
57 k=100, accuracy=0.263000
58 k=100, accuracy=0.256000
59 k=100, accuracy=0.263000
60 the average accuracy is :0.261600
61

```

通过比较，我们知道当 **k=10** 时，准确率最高。

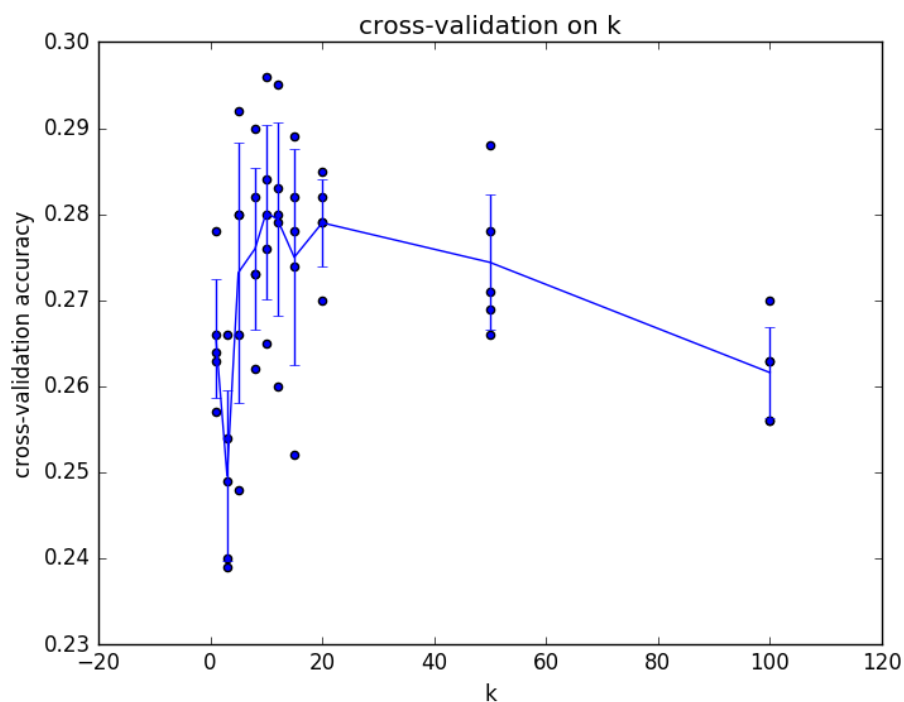
为了更形象的表现准确率，我们借助 `matplotlib.pyplot.errorbar` 函数来均值和偏差对应的趋势线，代码如下：

```

1  for k in k_choices:
2      accuracies=k_to_accuracies[k]
3      plt.scatter([k]*len(accuracies),accuracies)
4
5  accuracies_mean=np.array([np.mean(v) for k,v in
6  sorted(k_to_accuracies.items())])
7  accuracies_std=np.array([np.std(v) for k ,v in
8  sorted(k_to_accuracies.items())])
9  plt.errorbar(k_choices,accuracies_mean,yerr=accuracies_std)
10 plt.title('cross-validation on k')
11 plt.xlabel('k')
12 plt.ylabel('cross-validation accuracy')
13 plt.show()

```

得到结果如下图：



如图，也可以看出，当 $k=10$ 时，准确率最高。

已经通过交叉验证选择了最好的k值，下面我们就要使用最好的k值来完成预测任务，代码如下：

```

1 best_k=10
2 classifier=KNearestNeighbor()
3 classifier.train(x_train,y_train)
4 y_test_pred=classifier.predict(x_test,k=best_k)
5
6 num_correct=np.sum(y_test_pred==y_test)
7 accuracy=float(num_correct)/num_test
8 print('got %d / %d correct => accuracy: %f' %
    (num_correct,num_test,accuracy))

```

运行结果如下：

```

1 got 141 / 500 correct => accuracy: 0.282000

```

根据准确率，我们可以看到，使用经过交叉验证得到的最好的k值（当然仅限于我们前面所列举的几个），模型的准确率虽有所提升，但还是不够高，我们将在接下来完成cs231n其他作业的同时使用机器学习其他算法提升模型的分类性能。

至此，我们便完成了将测试集输入knn模型、输出相应类别以及模型的准确率的工作。

4 总结

以上我们完成了模型构建、数据载入、预测输出工作，根据预测测试集准确率可知，该分类器的准确率并不高（相对于卷积神经网络接近100%的准确率来说），在实际中也很少使用knn算法，但是通过这个算法我们可以对图像分类问题的基本过程有个大致的了解。

从上面我们可以看到，分类器有一些不足，我们这里可以总结下：

- 分类器必须记住所有训练数据并将其存储起来，以便于未来测试数据用于比较。这在存储空间上是低效的，数据集的大小很容易就以GB计；
- 对一个测试图像进行分类需要和所有训练图像作比较，算法计算资源耗费高。