HW2 KNN

M113040105 劉東霖

壹. compute_distances_two_loops:

一.介紹:利用最簡單的兩個 for 迴圈來算出 train 和 test 用 KNN 在歐基里德距離下算出來的距離，此方法因為要跑雙層 for 迴圈所以花費時間最長。

二.程式說明:

1.算距離公式如下圖:

$$d_2(I_1, I_2) = \sqrt{\sum_p \left(I_1^p - I_2^p\right)^2}$$

1.第 1，2 個 for 代表有幾個 test 和 train 要跑

2.利用 torch.sqrt(torch.sum((x_test[i]-x_train[j])**2))

把(第 i 個 test 和第 j 個 train 相減)平方總和，再把結果開根號就是距離矩陣[i，j]的值。

3.算完 num_test*num_train 次後回傳距離矩陣。

4.程式碼如下:

```
num_train = x_train.shape[0]
num_test = x_test.shape[0]
for i in range(num_test):
    for j in range(num_train):
        dists[j,i]=torch.sqrt(torch.sum((x_test[i]-x_train[j])**2))
```

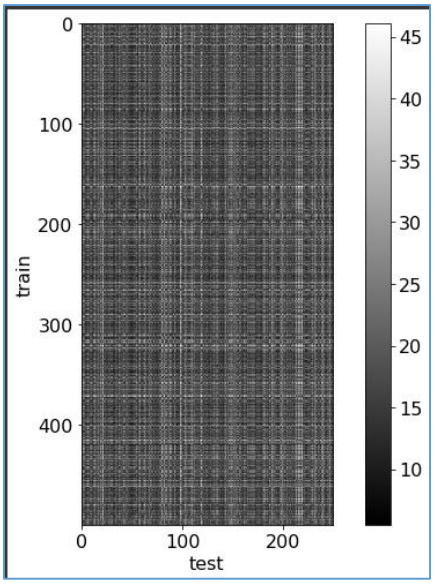
三.執行結果:

1.距離矩陣的 shape:

```
torch.manual_seed(0)
num_train = 500
num_test = 250
x_train, y_train, x_test, y_test = data.cifar10(num_train, num_test)

dists = compute_distances_two_loops(x_train, x_test)
print('dists has shape: ', dists.shape)
```

```
dists has shape: torch.Size([500, 250])
```

2.每一個點的距離大小用灰階圖表示，距離越小代表點越黑。如下圖:

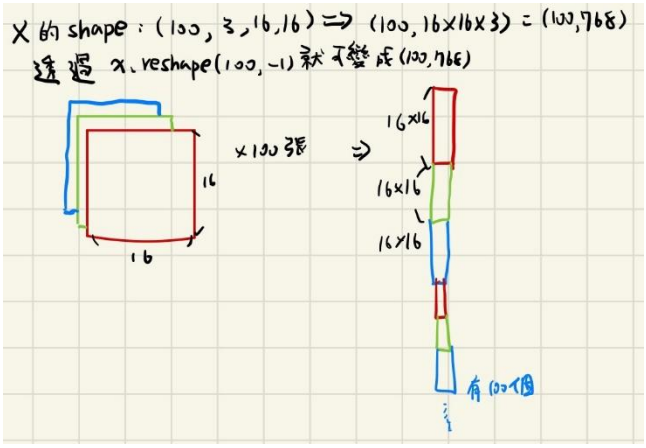


貳. compute_distances_one_loops:

一.介紹:在前一個例子中，因為要用雙層 loop 跑數據，速度非常的慢，所以我嘗試只用一個 loop 跑數據，並利用 broadcast 的方式來計算距離矩陣。

二. 程式說明:

1. 把 test 和 train 資料變成從 4 維變成 2 維，詳情如下圖:



2. 用一個 loop for test 數量，利用下圖程式來用 broadcast 的方式來計算距離矩陣。

```
dists[:,i]=torch.sqrt(torch.sum((x_test[i,:]-x_train)**2,axis=1))
```

3. 程式碼:

```
num_train = x_train.shape[0]
num_test = x_test.shape[0]
dists = x_train.new_zeros(num_train, num_test)
```

```
x_train,x_test=x_train.reshape(num_train,-1),x_test.reshape(num_test,-1)
for i in range(num_test):
    dists[:,i]=torch.sqrt(torch.sum((x_test[i,:]-x_train)**2,axis=1))
```

三. 執行結果:

1. 與 two loop 計算時的差異:

```
Difference: 0.0
Good! The distance matrices match
```

叁. compute_distances_no_loops:

一. 介紹:no loop 計算將距離的計算方法用多個矩陣相加就可以實現，此方法實現了向量化的方法，大量減少因 loop 所花費的時間。

二. 程式說明:

1.首先，我們可以把歐基里德距離公式拆解，如下圖所示，I1 和 12 分別代表 train 和 test。

$$d_2(\mathbf{I}_1, \mathbf{I}_2) = ||\mathbf{I}_1 - \mathbf{I}_2|| = \sqrt{||\mathbf{I}_1||^2 + ||\mathbf{I}_2||^2 - 2\mathbf{I}_1 \cdot \mathbf{I}_2}$$

2.跟前面一樣，把 train 和 test 從 4 維變 2 維。

3.利用

x_train_square=torch.sum(x_train**2,axis=1,keepdim=True)來計算 x_train 的平方。並把平方完的結果相加。

3.利用

x_test_square=torch.sum(x_test**2,axis=1)來計算 x_test 的平方。並把平方完的結果相加。

4.利用 cross=torch.matmul(x_train,x_test.T)算出 train 和 test 的內積。

5.利用 torch.sqrt(x_test_square-2*cross+x_train_square)

代入 1.圖上的公式，並回傳 dists。

6.程式碼:

```
x_train,x_test=x_train.reshape(num_train,-1),x_test.reshape(num_test,-1)
x_train_square=torch.sum(x_train**2,axis=1,keepdim=True)
x_test_square=torch.sum(x_test**2,axis=1)
cross=torch.matmul(x_train,x_test.T)
dists=torch.sqrt(x_test_square-2*cross+x_train_square)
```

三.執行結果:

1.與 two loop 的差異:

```
Difference:  3.1752595889861834e-13
Good! The distance matrices match
```

2.比較每一個 loop 的執行時間，發現 no loop 花的時間比較少。如下圖所示:

```
Two loop version took 7.23 seconds
One loop version took 0.92 seconds (7.8X speedup)
No loop version took 0.05 seconds (151.2X speedup)
```

四.參考文獻:

https://ljvmiranda921.github.io/notebook/2017/02/09/k-nearest-neighbors/

肆. Predict_label:

一.介紹:先決定 k 的大小，通常是奇數。再從距離矩陣中找出 k 個最小距離和對應的 label，再從這 k 個資料裡頭進行投票，出現次數最多的就是他預測到的 label。

二. 程式說明:

1.用一個 loop for test 數量

2.利用 indices=torch.argsort(dists[:,i])將距離矩陣的第 i 筆距離由大到小進行參數排列。

3.利用 index=indices[range(k)]把前 k 個最小的距離放到 index 陣列裡面。

4. 利用 closest_y=y_train[index]把 index 對應到 y_train 的 label。

5. 利用 counts=torch.bincount(closest_y)記錄著 label 的出現次數。

6. 利用 y_pred[i]=torch.argmax(counts)紀錄出現次數最多的 label。

7.程式碼:

```
num_train, num_test = dists.shape
#num_test=dists.shape[0]
y_pred = torch.zeros(num_test, dtype=torch.int64)
#########################################################
# TODO: Implement this function. You may use
# samples. Hint: Look up the function torch.to
#########################################################
# Replace "pass" statement with your code
for i in range(num_test):
    indices=torch.argsort(dists[:,i])
    index=indices[range(k)]
    closest_y=y_train[index]
    counts=torch.bincount(closest_y)
    y_pred[i]=torch.argmax(counts)
```

三. 執行結果:

```
dists = torch.tensor([
        [0.3, 0.4, 0.1],
        [0.1, 0.5, 0.5],
        [0.4, 0.1, 0.2],
        [0.2, 0.2, 0.4],
        [0.5, 0.3, 0.3],
])
y_train = torch.tensor([0, 1, 0, 1, 2])
y_pred_expected = torch.tensor([1, 0, 0])
y_pred = predict_labels(dists, y_train, k=3)
correct = y_pred.tolist() == y_pred_expected.tolist()
print('Correct: ', correct)

Correct: True
```

伍. knn 利用 class:

```
class KnnClassifier:
```

一. __init__:在呼叫 class 時，把 x_train 和 y_train 初始化，如下圖所示:

```
classifier = KnnClassifier(x_train, y_train)
```

```
def __init__(self, x_train, y_train):
    """
    Create a new K-Nearest Neighbor classi
    In the initializer we simply memorize

    Inputs:
    - x_train: Torch tensor of shape (num
    - y_train: int64 torch tensor of shap
    """
    ###########################################
    # TODO: Implement the initializer for
    # computation and simply memorize the
    ###########################################
    # Replace "pass" statement with your
    self.x_train,self.y_train=x_train,y_train
```

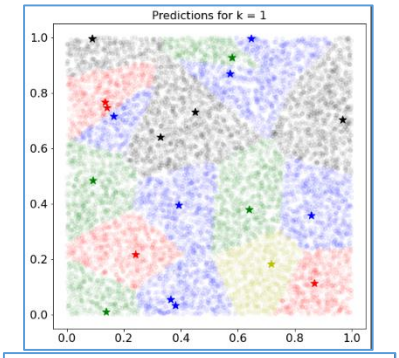
二. predict:題目指定要用 no loop 算出距離矩陣，所以算距離矩陣時代入我剛剛寫的 compute_distances_no_loops 的 function 裡，預測時代入我剛剛寫的 predict function。完整程式如下:

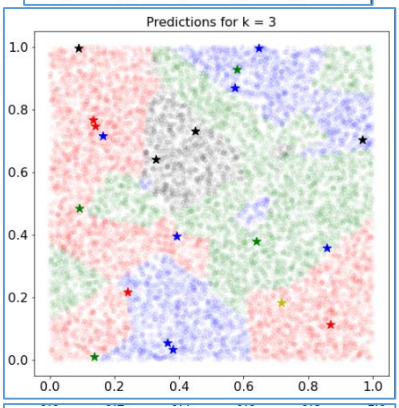
```
y_test = classifier.predict(x_test, k=k)
```

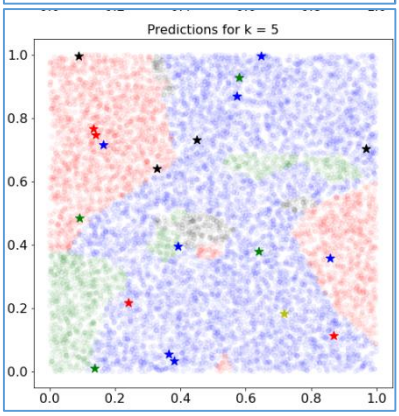
```
def predict(self, x_test, k=1):
    """
    Make predictions using the classifier.

    Inputs:
    - x_test: Torch tensor of shape (num_test, C, H, W) giving test sample
    - k: The number of neighbors to use for predictions

    Returns:
    - y_test_pred: Torch tensor of shape (num_test,) giving predicted labels
      for the test samples.
    """
    y_test_pred = None
    ###########################################################################
    # TODO: Implement this method. You should use the functions you wrote
    # above for computing distances (use the no-loop variant) and to predic
    # output labels.
    ###########################################################################
    # Replace "pass" statement with your code
    dists=compute_distances_no_loops(self.x_train, x_test)
    y_test_pred=predict_labels(dists,self.y_train,k=k)
    ###########################################################################
    #                                                       END OF YOUR CODE
    ###########################################################################
    return y_test_pred
```

三. 執行結果:

1. 不同的 k 預測出來的結果:可以看出 k 的越大，可以與更多人參與投票，有機會把原本可能會猜錯的給預測正確。K 比較小時可能會因為雜訊而預測錯誤







2. k=1 時的準確率:

```
from knn import KnnClassifier

torch.manual_seed(0)
num_train = 5000
num_test = 500
x_train, y_train, x_test, y_test = data.cifar10(num_train, num_test)

classifier = KnnClassifier(x_train, y_train)
classifier.check_accuracy(x_test, y_test, k=1)
```

```
Got 137 / 500 correct; accuracy is 27.40%
27.4
```

3. k=5 時的準確率:

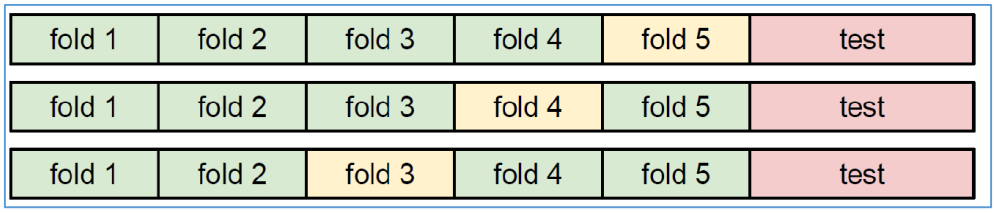
```
from knn import KnnClassifier

torch.manual_seed(0)
num_train = 5000
num_test = 500
x_train, y_train, x_test, y_test = data.cifar10(num_train, num_test)

classifier = KnnClassifier(x_train, y_train)
classifier.check_accuracy(x_test, y_test, k=5)
```

```
Got 139 / 500 correct; accuracy is 27.80%
27.8
```

伍. Cross-validation:

一. 介紹:把資料放入 F 個資料夾中，將其中 F-1 份的資料當作訓練集，剩下來的那份做為驗證集，算出準確率，再從沒當過驗證集的資料挑一份出來當驗證集，如此反覆直到每一份資料都當過驗證集，總共會執行 F 次，算出 F 個準確率。如下圖所示:

二. 程式說明:



1. 利用 torch.chunk 把資料放入 num_folds 個資料夾，如下圖所示:

```
x_train_folds=torch.chunk(x_train,num_folds)
y_train_folds=torch.chunk(y_train,num_folds)
```

2.題目要求我們在多個 k 底下評估準確率，所以先有一個 for k-list 的 for loop，並把各個 k 的準確率用字典存起來，如下圖所示:

```
for k_num in (k_choices):
    k_to_accuracies[k_num]=[]
```

3.for 裡面在放一個 for loop 用來做 num_folds 次交叉驗證，如下圖所示:

```
for flod in range(num_folds):
```

4.把 train_folads 的第 fold 筆資料夾當作驗證集存起來，如下圖所示:

```
x_val=x_train_folds[flod]
y_val=y_train_folds[flod]
```

5.把不是驗證集的資料夾用 torch.cat 串起來，如下圖所示:

```
x_train_flod_4=torch.cat(x_train_folds[:flod]+x_train_folds[flod+1:],dim=0)
y_train_flod_4=torch.cat(y_train_folds[:flod]+y_train_folds[flod+1:],dim=0)
```

6.把剛剛的 train data 丟到 model 裡訓練，再把驗證集丟到模型裡預測，如下圖所示:

```
classifier=KnnClassifier(x_train_flod_4,y_train_flod_4)
y_val_pred=classifier.predict(x_val,k=k_num)
```

7. 計算剛剛預測出來的準確率，並把結果存入字典裡。如下圖所示:

```
num_correct=torch.sum(y_val_pred==y_val)
accuracy=float(num_correct)/y_val.shape[0]
k_to_accuracies[k_num].append(accuracy)
```

8. 完整程式碼:

```
x_train_folds=torch.chunk(x_train,num_folds)
y_train_folds=torch.chunk(y_train,num_folds)
```

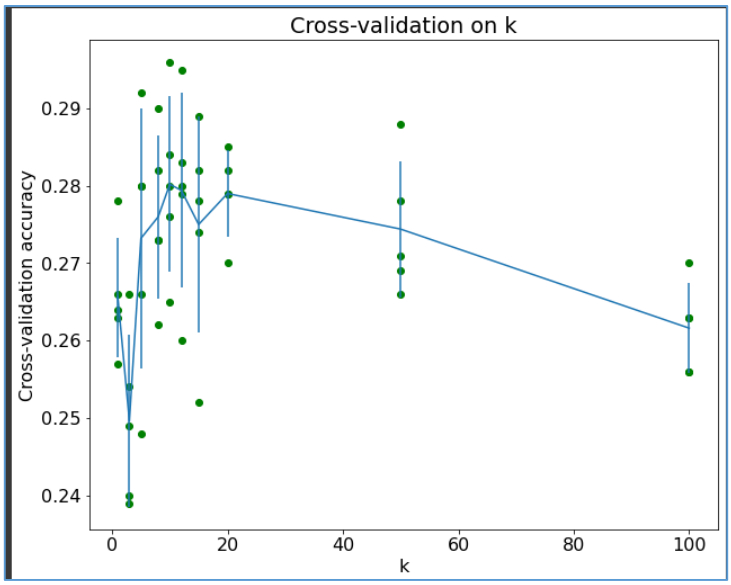
```
for k_num in (k_choices):
    k_to_accuracies[k_num]=[]
    for flod in range(num_folds):
        x_val=x_train_folds[flod]
        y_val=y_train_folds[flod]
        x_train_flod_4=torch.cat(x_train_folds[:flod]+x_train_folds[flod+1:],dim=0)
        y_train_flod_4=torch.cat(y_train_folds[:flod]+y_train_folds[flod+1:],dim=0)
        classifier=KnnClassifier(x_train_flod_4,y_train_flod_4)
        y_val_pred=classifier.predict(x_val,k=k_num)
        num_correct=torch.sum(y_val_pred==y_val)
        accuracy=float(num_correct)/y_val.shape[0]
        k_to_accuracies[k_num].append(accuracy)
```

三. 執行結果:

1. 每個字典裡的資料:

```
k = 1 got accuracies: [0.263, 0.257, 0.264, 0.278, 0.266]
k = 3 got accuracies: [0.239, 0.249, 0.24, 0.266, 0.254]
k = 5 got accuracies: [0.248, 0.266, 0.28, 0.292, 0.28]
k = 8 got accuracies: [0.262, 0.282, 0.273, 0.29, 0.273]
k = 10 got accuracies: [0.265, 0.296, 0.276, 0.284, 0.28]
k = 12 got accuracies: [0.26, 0.295, 0.279, 0.283, 0.28]
k = 15 got accuracies: [0.252, 0.289, 0.278, 0.282, 0.274]
k = 20 got accuracies: [0.27, 0.279, 0.279, 0.282, 0.285]
k = 50 got accuracies: [0.271, 0.288, 0.278, 0.269, 0.266]
k = 100 got accuracies: [0.256, 0.27, 0.263, 0.256, 0.263]
```

2. 資料分布和 errorbar，可大約看到最佳 k 在 0-20 之間:



四. 參考資料:

https://scikit-learn.org/stable/modules/cross_validation.html

陸. knn_best_k:

一. 介紹:剛剛做完 cross_validation 之後，我們再將這幾個資料夾裡的準確率取平均，用平均分數來評斷模型的好壞，再從多個 k 中取出準確率最高的。

二. 程式碼:

```
best_k = 0
################################################
# TODO: Use the results of cross-v
# choose the value of k, and store
# the value of k that has the hig
################################################
# Replace "pass" statement with you
best_acc=-1
for k,acc in k_to_accuracies.items():
    mean_acc=sum(acc)/float(len(acc))
    if mean_acc>best_acc:
        best_acc=mean_acc
        best_k=k
```

三.執行結果:

```
from knn import KnnClassifier
from knn import knn_get_best_k

best_k = 1
torch.manual_seed(0)

best_k = knn_get_best_k(k_to_accuracies)
print('Best k is ', best_k)

classifier = KnnClassifier(x_train, y_train)
classifier.check_accuracy(x_test, y_test, k=best_k)
```

```
Best k is  10
Got 141 / 500 correct; accuracy is 28.20%
28.2
```

柒. Run cifar-10 full data to get accuracy from best k:

一.執行結果:

```
from knn import KnnClassifier

torch.manual_seed(0)
x_train_all, y_train_all, x_test_all, y_test_all = data.cifar10()
classifier = KnnClassifier(x_train_all, y_train_all)
classifier.check_accuracy(x_test_all, y_test_all, k=best_k)
```

```
Got 3386 / 10000 correct; accuracy is 33.86%
33.86
```