

KNN实验分享

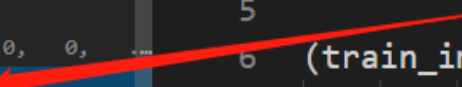
CSEE1801 李书涵

数据导入

- 使用TensorFlow的在线数据库导入MNIST
(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.mnist.load_data()
- 得到的图像为二维数组，其数据类型为uint8，即范围在[0, 255]的无符号数；

```
✓ Locals
> special variables
> np: <module 'numpy' from 'C:\\Progr...
> plt: <module 'matplotlib.pyplot' fr...
✓ test_images: array([[0, 0, 0, ..., ...
> special variables
> [0:10000] : [array([[ 0,  0, ...
> dtype: dtype('uint8')
```

```
1 import tensorflow as tf
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import time
5
6 (train_images, train_labels), (test_images,
7 test_labels) = tf.keras.datasets.mnist.load_data()
8 # Using tensorflow online dataset library to load MNIST
```



数据处理

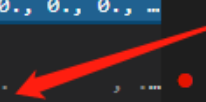
- 由于实现KNN算法需要计算图像间的距离，且涉及到平方操作；而uint8类型会使得平方结果存在溢出的可能，因此需要对数据进行类型转换；
- 在实验中需考虑计算精度和类型转换两个因素，选择将数据归一化到[0.0, 1.0]范围内；由下面两行代码可得到float64类型的图像数据。

```
train_images = train_images / 255.0
```

```
test_images = test_images / 255.0
```

```
> tf: <module 'tensorflow' from 'C:\\...>
> time: <module 'time' (built-in)>
v train_images: array([[[0., 0., 0., ...
> special variables
> [0:60000]: [array([[0., ..., ...
> dtype: dtype('float64')
```

```
6 (train_images, train_labels), (test_images,
7 | | | | | | | | | | test_labels) = tf.keras.datasets.mnist.load_data()
8 # Using tensorflow online dataset library to load MNIST
9 train_images = train_images / 255.0
10 test_images = test_images / 255.0
```



数据处理

- 为保险起见，获取数据集的一些基本信息备用：

```
num_train = train_images.shape[0]
```

```
#训练集数据量
```

```
num_test = test_images.shape[0]
```

```
#测试集数据量
```

```
img_shape = (test_images.shape[1], test_images.shape[2])
```

```
#图像尺寸
```

```
num_pixel = test_images.shape[1] * test_images.shape[2]
```

```
#图像总像素数目
```

计算思路

- KNN算法中，需要得到对某张测试照片的预测，需要得到其与训练集中每张图像的距离；由于是否对距离取平方根不影响KNN的结果，考虑略去这一步骤以降低计算规模；
- 如若使用循环比较，需要逐张照片计算距离、求和、扩充距离矩阵；这个过程会耗费大量时间；
- 假设选中的训练集数目为n，若能将测试图片视作一个向量拷贝n份放入矩阵A，再将其与相同形状的训练集矩阵B做如下操作：
$$(A - B) ** 2$$
- 即可充分利用numpy对矩阵运算的优化，减少运算所需时间

数据处理

- 由上述的计算思路，对数据做进一步处理；
- 为了方便矩阵的表示和运算，对每张图像数据进行向量化处理；

```
train_images = np.reshape(train_images, (num_train, num_pixel))
```

```
test_images = np.reshape(test_images, (num_test, num_pixel))
```

```
max: 'ndarray too big, calculating...' 17 train_images = np.reshape(train_images, (num_train, num_pixel))
min: 'ndarray too big, calculating...' 18 test_images = np.reshape(test_images, (num_test, num_pixel))
> shape: (60000, 784) ← 19 # Vectorizing 2-D images
size: 47040000
```

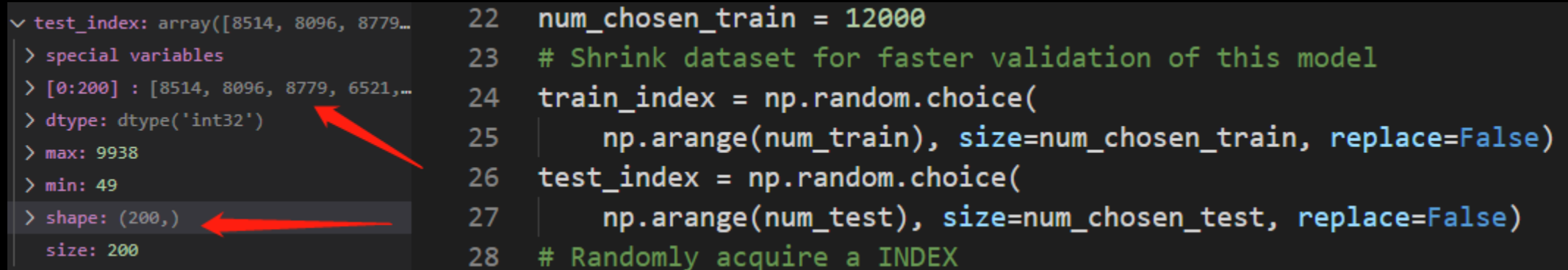
数据处理

- 在MNIST上运行KNN算法会消耗大量的算力；因此需要允许对数据进行合理的缩减，一方面在算法设计初期用于快速故障排除；另一方面便于抽取部分图像研究个别的计算结果。
- 通过numpy内建的随机抽取实现。

```
num_chosen_test = 2000  
num_chosen_train = 12000
```

数据处理

```
train_index = np.random.choice(  
    np.arange(num_train), size=num_chosen_train, replace=False)  
test_index = np.random.choice(  
    np.arange(num_test), size=num_chosen_test, replace=False)  
# Randomly acquire a INDEX
```



```
22 num_chosen_train = 12000  
23 # Shrink dataset for faster validation of this model  
24 train_index = np.random.choice(  
25     np.arange(num_train), size=num_chosen_train, replace=False)  
26 test_index = np.random.choice(  
27     np.arange(num_test), size=num_chosen_test, replace=False)  
28 # Randomly acquire a INDEX
```

✓ test_index: array([8514, 8096, 8779, ...])
> special variables
> [0:200] : [8514, 8096, 8779, 6521, ...]
> dtype: dtype('int32')
> max: 9938
> min: 49
> shape: (200,)
size: 200

数据处理

```
train_images = train_images[train_index]  
train_labels = train_labels[train_index]
```

```
test_images = test_images[test_index]  
test_labels = test_labels[test_index]
```

```
# Use INDEX to make slices
```

KNN设计

- KNN的主要任务是：

- 计算test_images中每张图片与train_images中的距离；
- 根据k和train_labels得到最近邻的k个图片的标签；
- 得到test_images中每张图片的预测结果result；
- 将result作为返回值返回；
- *根据任务需要，对最近邻的k张图片作图。

- 给出KNN函数的声明：

```
def KNN(train_labels, train_images, test_images, k):
```

- 以下是具体设计：

KNN设计

```
result = np.zeros((num_chosen_test), dtype='uint8')
    #存储返回结果, 初始化为0
for i in range(num_chosen_test): #np.tile()用于堆叠向量
    X = np.reshape(np.tile(test_images[i],
        num_chosen_train), (num_chosen_train, num_pixel))
    #得到复制num_chosen_train次的测试图像向量矩阵
    X = (X - train_images) ** 2
    #计算距离
    X = np.sum(X, axis=1)
    #对各点距离求和, 由于不影响结果, 未做平均
```

KNN设计

```
for i in range(num_chosen_test):  
    .....  
    .....  
    topk = train_labels[(np.argsort(X))[0:k]]  
    #argsort()排序后取对应下标、取前k个下标作为索引  
    #根据索引对train_labels进行切片  
    result[i] = np.argmax(np.bincount(topk))  
    #计数、数量最大结果作为对第i张图像的预测  
return result
```

KNN设计

- *绘图功能:

```
for i in range(num_chosen_test):
```

```
.....
```

```
.....
```

```
show_list = train_images[np.argsort(X)[0:k]]
```

```
for j in range(k):
```

```
    plt.subplot(num_chosen_test, k, i * k + j + 1)
```

```
    plt.imshow(np.reshape(show_list[j] * 255, img_shape))
```

```
    plt.xticks([])
```

```
    plt.yticks([])
```

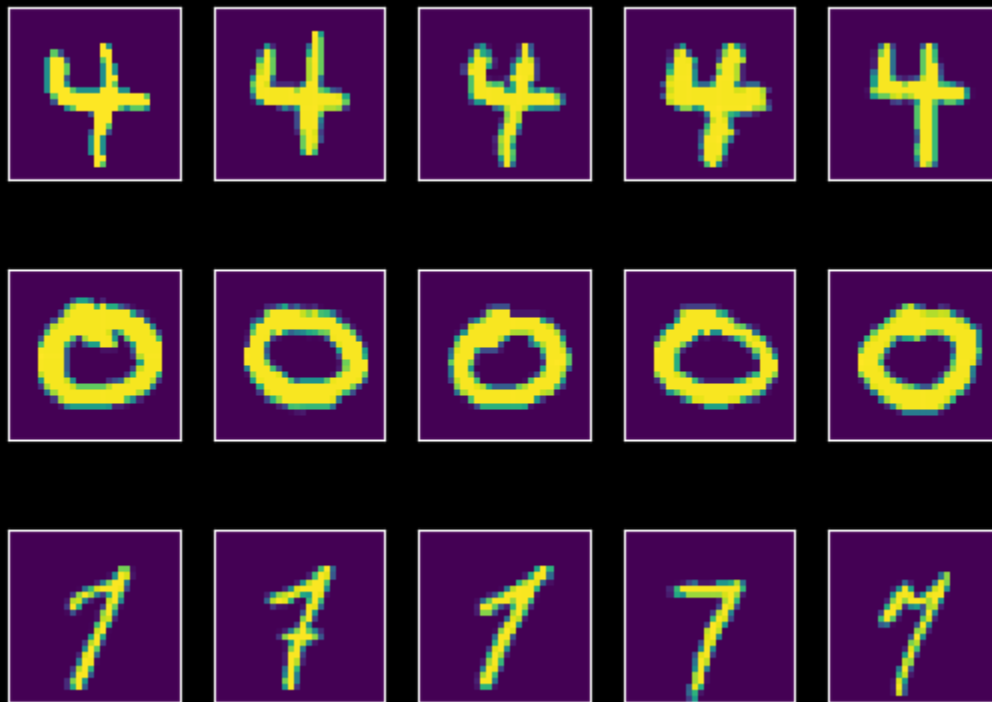
```
plt.show()
```

#计算图像位置



KNN设计

- *以 $k=5$, $\text{num_chosen_test} = 3$ 为例:



三张测试图片的5-最近邻图片

Accuracy-K曲线绘制

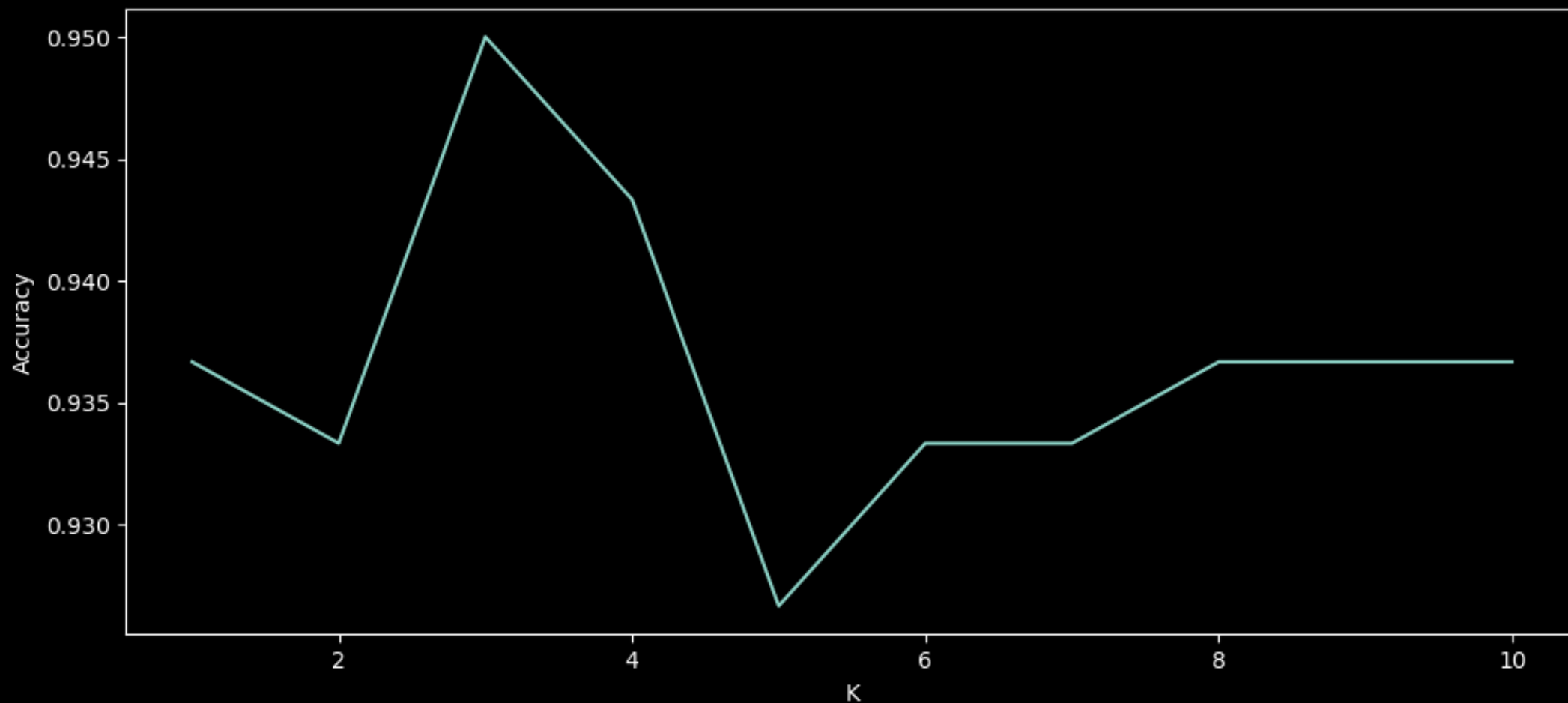
```
k_range = range(1, 11)
accuracy = []
for i in k_range:
    result = KNN(train_labels, train_images, test_images, i)
    diff = result - test_labels
    cur_accu = (np.bincount(diff)[0]) / num_chosen_test
    #np.bincount()得到的数组[0]号元素为分类正确的计数
    print("K = %d, accuracy = %f" % (i, cur_accu))
    accuracy.append(cur_accu)
plt.plot(k_range, accuracy)
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.show()
```

#计算不同K的准确度
#将准确度放入accuracy数组

#根据数据绘制Accuracy-K图像

Accuracy-K曲线绘制

- 取 `num_chosen_test = 300`, `num_chosen_train = 10000`
- K取范围 `range(1, 11)`



EOF