

实验五：Fashion-MNIST 数据分类

学号：2018329621262

姓名：刘恒玮

一、实验目的

综合利用机器学习知识解决分类问题。

二、实验环境

PC 机，Python，numpy 库，sklearn 库。

三、实验内容

给定 fashion-MNIST 数据集，对其进行分类训练，在测试集验证准确性。

四、实验原理（算法）

SVM 算法原理：

SVM 又称为支持向量机，是一种二分类的模型。当然如果进行修改之后也是可以用于多类别问题的分类。支持向量机可以分为线性核非线性两大类。其主要思想为找到空间中的一个更够将所有数据样本划开的超平面，并且使得本本集中所有数据到这个超平面的距离最短。SVM 学习的基本想法是求解能够正确划分训练数据集并且几何间隔最大的分离超平面。 $\omega x + b = 0$ 即为分离超平面，对于线性可分的数据集来说，这样的超平面有无穷多个（即感知机），但是几何间隔最大的分离超平面却是唯一的。

MLP 算法原理：

多层感知机（MLP，Multilayer Perceptron）除了输入输出层，它中间可以有多个隐层，最简单的 MLP 只含一个隐层，即三层的结构。多层感知机层与层之间是全连接的：上一层的任何一个神经元与下一层的所有神经元都有连接。多层感知机最底层是输入层，中间是隐藏层，最后是输出层。隐藏层的神经元怎么得来？首先它与输入层是全连接的，假设输入层用向量 X 表示，则隐藏层的输出就是 $f(\omega_1 x + b_1)$ ， ω_1 是权重（也叫连接系数）， b_1 是偏置，函数 f 可以是常用的 sigmoid 函数或者 tanh 函数。其实隐藏层到输出层可以看成是一个多类别的逻辑

回归，也即 softmax 回归，所以输出层的输出就是 $\text{softmax}(\omega_2 x_1 + b_2)$ ， x_1 表示隐藏层的输出 $f(\omega_1 x + b_1)$ 。因此，MLP 所有的参数就是各个层之间的连接权重以及偏置，包括 ω_1 、 b_1 、 ω_2 、 b_2 。对于一个具体的问题，怎么确定这些参数？求解最佳的参数是一个最优化问题，解决最优化问题，最简单的就是梯度下降法了（SGD）：首先随机初始化所有参数，然后迭代地训练，不断地计算梯度和更新参数，直到满足某个条件为止（比如误差足够小、迭代次数足够多时）。这个过程涉及到代价函数、规则化（Regularization）、学习速率（learning rate）、梯度计算等。

PCA 算法原理：

PCA 降维：

降维就是一种对高维度特征数据预处理方法。降维是将高维度的数据保留下最重要的一些特征，去除噪声和不重要的特征，从而实现提升数据处理速度的目的。降维具有如下一些优点：使得数据集更易使用、降低算法的计算开销、去除噪声、使得结果容易理解。PCA (Principal Component Analysis)，即主成分分析方法，是一种使用最广泛的数据降维算法。PCA 的主要思想是将 n 维特征映射到 k 维上，这 k 维是全新的正交特征也被称为主成分，是在原有 n 维特征的基础上重新构造出来的 k 维特征。PCA 的工作就是从原始的空间中顺序地找一组相互正交的坐标轴，新的坐标轴的选择与数据本身是密切相关的。其中，第一个新坐标轴选择是原始数据中方差最大的方向，第二个新坐标轴选取是与第一个坐标轴正交的平面中使得方差最大的，第三个轴是与第 1, 2 个轴正交的平面中方差最大的。依次类推，可以得到 n 个这样的坐标轴。通过这种方式获得的新的坐标轴，我们发现，大部分方差都包含在前面 k 个坐标轴中，后面的坐标轴所含的方差几乎为 0。于是，我们可以忽略余下的坐标轴，只保留前面 k 个含有绝大部分方差的坐标轴。事实上，这相当于只保留包含绝大部分方差的维度特征，而忽略包含方差几乎为 0 的特征维度，实现对数据特征的降维处理。

总结一下 PCA 的算法步骤：设有 m 条 n 维数据。将原始数据按列组成 n 行 m 列矩阵 X ；将 X 的每一行（代表一个属性字段）进行零均值化，即减去这一行的均值；求出协方差矩阵；求出协方差矩阵的特征值及对应的特征向量；将特征向

量按对应特征值大小从上到下按行排列成矩阵，取前 k 行组成矩阵 P ，即为降维到 k 维后的数据。

五、实验步骤（分析过程）

1. 导入实验过程中所需要的相关库。

```
[1]: import gzip
import numpy as np
import struct
from sklearn.svm import SVC
%matplotlib inline
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn import metrics
from sklearn.neural_network import MLPClassifier
```

2. 编写导入数据的函数。

```
[2]: # Load compressed MNIST gz files and return numpy arrays
def load_data(filename, label=False):
    with gzip.open(filename) as gz:
        struct.unpack('I', gz.read(4))
        n_items = struct.unpack('>I', gz.read(4))
        if not label:
            n_rows = struct.unpack('>I', gz.read(4))[0]
            n_cols = struct.unpack('>I', gz.read(4))[0]
            res = np.frombuffer(gz.read(n_items[0] * n_rows * n_cols), dtype=np.uint8)
            res = res.reshape(n_items[0], n_rows * n_cols)
        else:
            res = np.frombuffer(gz.read(n_items[0]), dtype=np.uint8)
            res = res.reshape(n_items[0], 1)
    return res


# one-hot encode a 1-D array
def one_hot_encode(array, num_of_classes):
    return np.eye(num_of_classes)[array.reshape(-1)]
```

3. 导入 fashionMNIST 的训练集和测试集。

```
[3]: X_train = load_data("data/MNIST/train-images-idx3-ubyte.gz") / 255.0
X_test = load_data("data/MNIST/t10k-images-idx3-ubyte.gz") / 255.0
y_train = load_data("data/MNIST/train-labels-idx1-ubyte.gz", True).reshape(-1)
y_test = load_data("data/MNIST/t10k-labels-idx1-ubyte.gz", True).reshape(-1)
```

4. 随机在训练集中选取 30 个数据，展示 fashionMNIST 图像及其标签。

```
[4]: count = 0
sample_size = 30
plt.figure(figsize=(16, 6))
for i in np.random.permutation(X_train.shape[0])[:sample_size]:
    count = count + 1
    plt.subplot(1, sample_size, count)
    plt.axhline('')
    plt.axvline('')
    plt.text(x=10, y=-10, s=y_train[i], fontsize=18)
    plt.imshow(X_train[i].reshape(28, 28), cmap=plt.cm.Greys)
plt.show()
```



5. 输出 fashionMNIST 数据集和测试集的数量。

```
[5]: print(X_train.shape, y_train.shape, X_test.shape, y_test.shape, sep = '\n')
(60000, 784)
(60000,)
(10000, 784)
(10000,)
```

6. 采用 SVM 模型进行预测，将目标函数的惩罚系数 C 改为 10，验证集分数有所提高，提高了大约百分之二，但是效果提高还不够明显。

```
[6]: svm = SVC()
svm.fit(X_train,y_train)

[6]: SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
max_iter=-1, probability=False, random_state=None, shrinking=True,
tol=0.001, verbose=False)

[7]: y_pred = svm.predict(X_test)
print("Training set score: %f" % svm.score(X_train, y_train))
print("Test set score: %f" % svm.score(X_test, y_test))

Training set score: 0.912800
Test set score: 0.882800

[8]: svm = SVC(C = 10)
svm.fit(X_train,y_train)
y_pred = svm.predict(X_test)
print("Training set score: %f" % svm.score(X_train, y_train))
print("Test set score: %f" % svm.score(X_test, y_test))

Training set score: 0.971700
Test set score: 0.900200
```

7. 采用 MLP 模型进行预测，调整隐藏层的数量，将 hidden_layer_sizes 改为 (1000,)，验证集的准确率提升了大约百分之一，但效果不如 SVM。

```
[9]: mlp = MLPClassifier()
      mlp.fit(X_train, y_train)
      y_pred = mlp.predict(X_test)
      print("Training set score: %f" % mlp.score(X_train, y_train))
      print("Test set score: %f" % mlp.score(X_test, y_test))
```

```
Training set score: 0.994000
Test set score: 0.880900
```

```
[10]: mlp = MLPClassifier(hidden_layer_sizes=(1000,))
      mlp.fit(X_train, y_train)
      y_pred = mlp.predict(X_test)
      print("Training set score: %f" % mlp.score(X_train, y_train))
      print("Test set score: %f" % mlp.score(X_test, y_test))
```

```
Training set score: 0.991500
Test set score: 0.893100
```

8. 归一化处理后验证集的准确率几乎没有变，这是因为读取数据时/255 相当于是进行归一化过了。

```
[11]: scaler = StandardScaler()
      scaler.fit(X_train)
      X_train = scaler.transform(X_train)
      svm = SVC(C = 10)
      svm.fit(X_train, y_train)

      scaler = StandardScaler()
      scaler.fit(X_test)
      X_test = scaler.transform(X_test)
      y_pred = svm.predict(X_test)
      print("Training set score: %f" % svm.score(X_train, y_train))
      print("Test set score: %f" % svm.score(X_test, y_test))
```

```
Training set score: 0.982133
Test set score: 0.898700
```

```
[12]: scaler = StandardScaler()
      scaler.fit(X_train)
      X_train = scaler.transform(X_train)
      mlp = MLPClassifier(hidden_layer_sizes=(1000,))
      mlp.fit(X_train, y_train)

      scaler = StandardScaler()
      scaler.fit(X_test)
      X_test = scaler.transform(X_test)
      y_pred = mlp.predict(X_test)
      print("Training set score: %f" % mlp.score(X_train, y_train))
      print("Test set score: %f" % mlp.score(X_test, y_test))
```

```
Training set score: 0.988133
Test set score: 0.893400
```


9. 为提高训练的速度，对数据集进行 PCA 降维处理，用 for 循环选取出准确度最高的 PCA 维数。

```
[15]: svc_score_train = []
      mlp_score_train = []
      svc_score = []
      mlp_score = []

      for i in range(2,100):
          print(i)
          pca = PCA(n_components = i)
          train_x = pca.fit_transform(X_train)
          svm_clf = SVC()
          mlp_clf = MLPClassifier()
          svm_clf.fit(train_x, y_train)
          mlp_clf.fit(train_x, y_train)
          test_x = pca.transform(X_test)
          svm_y_predict_train = svm_clf.predict(train_x)
          svm_y_predict = svm_clf.predict(test_x)
          mlp_y_predict_train = mlp_clf.predict(train_x)
          mlp_y_predict = mlp_clf.predict(test_x)
          svc_score_train.append(metrics.accuracy_score(y_train, svm_y_predict_train))
          svc_score.append(metrics.accuracy_score(y_test, svm_y_predict))
          mlp_score_train.append(metrics.accuracy_score(y_train, mlp_y_predict_train))
          mlp_score.append(metrics.accuracy_score(y_test, mlp_y_predict))

      ...

[18]: print("SVM训练集的准确率: ",max(svc_score_train),"SVM验证集的准确率: ",max(svc_score),"验证集准确率最高时数据的维数",svc_score.index(max(svc_score))+2)
      print("MLP训练集的准确率: ",max(mlp_score_train),"MLP验证集的准确率: ",max(mlp_score),"验证集准确率最高时数据的维数",mlp_score.index(max(mlp_score))+2)

SVM训练集的准确率:  0.9045666666666666 SVM验证集的准确率:  0.8786 验证集准确率最高时数据的维数 98
MLP训练集的准确率:  0.9639666666666666 MLP验证集的准确率:  0.8765 验证集准确率最高时数据的维数 50
```

10. 调整 SVM 模型中的目标函数的惩罚系数 C，MLP 模型中隐藏层的数目，输出最优的 PCA 维数对应的验证集分数，但是验证集的准确率均有所下降。

```
[19]: pca = PCA(n_components = 98)
      train_x = pca.fit_transform(X_train)
      svm_clf = SVC(C = 10)
      svm_clf.fit(train_x, y_train)
      test_x = pca.transform(X_test)
      svm_y_predict_train = svm_clf.predict(train_x)
      svm_y_predict = svm_clf.predict(test_x)
      print("Training set score: %f" % svm_clf.score(train_x, y_train))
      print("Test set score: %f" % svm_clf.score(test_x, y_test))
```

```
Training set score: 0.957183
Test set score: 0.899300
```

```
[20]: pca = PCA(n_components = 50)
      train_x = pca.fit_transform(X_train)
      mlp_clf = MLPClassifier(hidden_layer_sizes=(1000,))
      mlp_clf.fit(train_x, y_train)
      test_x = pca.transform(X_test)
      mlp_y_predict_train = mlp_clf.predict(train_x)
      mlp_y_predict = mlp_clf.predict(test_x)
      print("Training set score: %f" % mlp_clf.score(train_x, y_train))
      print("Test set score: %f" % mlp_clf.score(test_x, y_test))
```

```
Training set score: 0.998783
Test set score: 0.887300
```

11. 由于利用 SVM 和 MLP 训练出来的模型去预测的效果准确率最好的情况也只

能达到 90%，更具网上的教程利用 Keras 框架搭建了四层的 CNN 神经网络模型，在经过 3 个 Dense 层之前，将第 3 个 Dropout 层的(4, 4, 128)输出形状为(2048,) 的向量。

```
[4]: import keras
      from keras.models import Sequential
      from keras.layers import Dense, Dropout, Flatten
      from keras.layers import Conv2D, MaxPooling2D, BatchNormalization

      cnn4 = Sequential()
      cnn4.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
      cnn4.add(BatchNormalization())

      cnn4.add(Conv2D(32, kernel_size=(3, 3), activation='relu'))
      cnn4.add(BatchNormalization())
      cnn4.add(MaxPooling2D(pool_size=(2, 2)))
      cnn4.add(Dropout(0.25))

      cnn4.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
      cnn4.add(BatchNormalization())
      cnn4.add(Dropout(0.25))

      cnn4.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
      cnn4.add(BatchNormalization())
      cnn4.add(MaxPooling2D(pool_size=(2, 2)))
      cnn4.add(Dropout(0.25))

      cnn4.add(Flatten())

      cnn4.add(Dense(512, activation='relu'))
      cnn4.add(BatchNormalization())
      cnn4.add(Dropout(0.5))

      cnn4.add(Dense(128, activation='relu'))
      cnn4.add(BatchNormalization())
      cnn4.add(Dropout(0.5))

      cnn4.add(Dense(10, activation='softmax'))

      cnn4.compile(loss=keras.losses.categorical_crossentropy,
                   optimizer=keras.optimizers.Adam(),
                   metrics=['accuracy'])
```

12. 在训练和验证数据上训练了 batch = 256 , epoch=10 的模型,验证集的准确率达到了 91.98%。

```
[17]: history = cnn4.fit(X_train, y_train,
                        batch_size=256,
                        epochs=10,
                        verbose=1,
                        validation_data=(X_val, y_val))

score = cnn4.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

Epoch 1/10
188/188 [=====] - 95s 505ms/step - loss: 0.7520 - accuracy: 0.7421 - val_loss: 3.9385 - val_accuracy: 0.1007
Epoch 2/10
188/188 [=====] - 95s 504ms/step - loss: 0.4336 - accuracy: 0.8444 - val_loss: 2.0589 - val_accuracy: 0.4474
Epoch 3/10
188/188 [=====] - 94s 501ms/step - loss: 0.3630 - accuracy: 0.8702 - val_loss: 0.5192 - val_accuracy: 0.8104
Epoch 4/10
188/188 [=====] - 96s 511ms/step - loss: 0.3276 - accuracy: 0.8845 - val_loss: 0.2930 - val_accuracy: 0.8947
Epoch 5/10
188/188 [=====] - 100s 533ms/step - loss: 0.2990 - accuracy: 0.8930 - val_loss: 0.2706 - val_accuracy: 0.9042
Epoch 6/10
188/188 [=====] - 97s 515ms/step - loss: 0.2833 - accuracy: 0.8983 - val_loss: 0.2593 - val_accuracy: 0.9104
Epoch 7/10
188/188 [=====] - 102s 545ms/step - loss: 0.2697 - accuracy: 0.9042 - val_loss: 0.2535 - val_accuracy: 0.9121
Epoch 8/10
188/188 [=====] - 93s 494ms/step - loss: 0.2528 - accuracy: 0.9099 - val_loss: 0.2645 - val_accuracy: 0.9073
Epoch 9/10
188/188 [=====] - 94s 502ms/step - loss: 0.2439 - accuracy: 0.9134 - val_loss: 0.2735 - val_accuracy: 0.9039
Epoch 10/10
188/188 [=====] - 95s 505ms/step - loss: 0.2344 - accuracy: 0.9166 - val_loss: 0.2137 - val_accuracy: 0.9258
Test loss: 0.22941437363624573
Test accuracy: 0.9197999835014343
```

13. 数据增强采用了通过大量随机变换对样本进行增强的方法，从而从现有的训练样本中生成更多的训练数据，从而产生看起来可信的图像。目的是在训练时，模型永远不会得到两张完全相同的图片。这对于模型的准确率会有所提升。在 Keras 中，这可以通过配置要对 ImageDataGenerator 实例读取的图像执行各种随机转换来完成。

```
[5]: from keras.preprocessing.image import ImageDataGenerator
gen = ImageDataGenerator(rotation_range=8, width_shift_range=0.08, shear_range=0.3,
                        height_shift_range=0.08, zoom_range=0.08)
batches = gen.flow(X_train, y_train, batch_size=256)
val_batches = gen.flow(X_val, y_val, batch_size=256)
```

14. epoch 设置为 50，训练大概一个小时候得到的最终分数，验证集的准确率仍然是在 92%左右，并没有得到显著提高，可能是由于模型出现了过拟合的现象。

```
[6]: history = cnn4.fit_generator(batches, steps_per_epoch=48000//256, epochs=50,
                                validation_data=val_batches, validation_steps=12000//256)
score = cnn4.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

Epoch 47/50
187/187 [=====] - 103s 549ms/step - loss: 0.2278 - accuracy: 0.9175 - val_loss: 0.2089 - val_accuracy: 0.9254
Epoch 48/50
187/187 [=====] - 109s 584ms/step - loss: 0.2245 - accuracy: 0.9183 - val_loss: 0.2111 - val_accuracy: 0.9249
Epoch 49/50
187/187 [=====] - 96s 513ms/step - loss: 0.2265 - accuracy: 0.9174 - val_loss: 0.2352 - val_accuracy: 0.9137
Epoch 50/50
187/187 [=====] - 97s 517ms/step - loss: 0.2251 - accuracy: 0.9192 - val_loss: 0.2265 - val_accuracy: 0.9194
Test loss: 0.2180289775133133
Test accuracy: 0.9186999797821045
```


六. 实验结果（训练集准确度，验证集准确度等指标）

SVM 算法准确度：

```
Training set score: 0.971700  
Test set score: 0.900200
```

MLP 算法准确度：

```
Training set score: 0.991500  
Test set score: 0.893100
```

四层 CNN 神经网络：

```
Test loss: 0.22941437363624573  
Test accuracy: 0.9197999835014343
```

七. 实验心得（碰到的问题，如何解决）

对于 MLP 算法训练时，输入层的节点数为 784，将隐藏层数量调到 1000 个节点，发现验证集分数有所提升，多层感知机隐藏层节点个数可能就是要大于输入层，这样的结构模型才会较为理想，验证集的分数才会较为理想。对于 SVM 算法识别 fashion-MNIST 数据集，准确率相比于 MLP 并没有明显的优势，甚至更差，而且所消耗的时间远大于 MLP，SVM 算法最大的缺点就是对于大量且复杂的数据集训练速度过慢，训练时间不稳定，复杂的数据集会导致训练过程要占大量资源。自己利用 Keras 框架搭建了四层的 CNN 神经网络模型，验证集的准确率有所提高，但准确率也只达到了 91.98%。

总的来说这个学期机器学习课中所学的各种方法，相对来说简单易用，对于我们所要识别和分类的数据集，不管网络内部的参数情况如何，最终能够得到比较理想的识别和分类效果，各种方法最大的缺点就是都需要消耗较多的时间去得到一个理想的效果（一个较高的分数），但对于内部的结构和各种参数，还需要时间去继续深入学习。