机器学习实践作业1 分类任务

1 任务目标

本次任务目标为使用python编写传统机器学习算法中的多层感知机（使用pytorch自行编写）和决策树（使用scikit-leran库）算法实现Fashion-mnist数据集上的图像识别任务，对两种算法的不同参数进行对比，寻找优化参数，同时对比两种算法的准确率（accuracy），精度（precision），召回率（recall），f1-score以及训练时间和执行时间，从中分析它们的优劣。

# 2 实现过程

## 2.0 数据集的读取与使用

使用torchvision来对Fashion-mnist数据集进行读取和使用。使用transforms对数据进行预处理，将像素值从[0,1]范围归一化到均值为0.5，标准差为0.5的范围；使用datasets.FashionMNIST()读取训练集和测试集，Dataloader()分批次加载训练集和测试机以备后面训练和测试使用。

同时，在训练前，将部分训练数据可视化，以便于观察训练数据的特征。打印图片之后发现，该数据集中有用的像素点像素值较高（偏白色），无用的像素点像素值较低（偏黑色），这在后面预测任意实际图片时有很大帮助。

## 2.1 多层感知机

### 2.1.1 算法原理

多层感知机（MLP）是一种基本的神经网络模型，也是深度学习的基础。它的原理相对简单，但在处理复杂的非线性关系时表现出色。

基本结构：

输入层：接收输入数据特征。

隐藏层：通过一系列线性变换和非线性激活函数将输入数据转换为更高层次的表示。

输出层：产生最终的预测结果。

工作原理：

前向传播：输入数据经过权重和偏置的线性变换后，通过激活函数（如ReLU、Sigmoid等）进行非线性变换，得到隐藏层的输出。这个过程逐层进行，直到输出层生成最终的预测结果。

反向传播：通过损失函数计算预测值与真实值之间的误差，并利用反向传播算法调整每一层的权重和偏置，以最小化误差。

### 2.1.2 实现思路

本次主要使用pytorch中的torchnn.module模块来构建多层感知机模型，包含输入层（维度为28x28=784），隐藏层（可作为待优化参数），输出层（维度为10，10个类别），激活函数使用Relu实现。

模型的损失函数采用交叉熵，优化方式选用了Adam，SGD进行对比。

关键代码：

模型定义

|  |
| --- |
| #使用pytorch构建一个多层感知机（MLP）模型  class MLP(nn.Module):  def \_\_init\_\_(self, input\_size, hidden\_size, num\_classes):  super(MLP, self).\_\_init\_\_()  self.fc1 = nn.Linear(input\_size, hidden\_size)  self.relu = nn.ReLU()  self.fc2 = nn.Linear(hidden\_size, num\_classes)    def forward(self, x):  out = self.fc1(x)  out = self.relu(out)  out = self.fc2(out)  return out  # 设置MLP模型的参数  input\_size = 784 # Input size for MNIST dataset (28x28 pixels)  hidden\_size = 128 # Number of neurons in the hidden layer  num\_classes = 10 # Number of classes (0-9 digits)  # 构建模型  model = MLP(input\_size, hidden\_size, num\_classes)  # 定义损失函数和优化器  criterion = nn.CrossEntropyLoss()  optimizer = optim.Adam(model.parameters(), lr=0.001) |

训练过程

|  |
| --- |
| # 训练模型  device=torch.device("cpu")  model=model.to(device)  max\_epoch = 10  for epoch in range(max\_epoch):  running\_loss=0    print(f'Iteration {epoch+1}/{max\_epoch}')  for i, data in enumerate(trainload,start=0):  images, labels = data # 获取训练集的批图像和标签    # images变为列向量  inputs = images.view(images.size(0), -1)  inputs,labels=inputs.to(device),labels.to(device)  outputs = model(inputs)  loss = criterion(outputs, labels)    # 后向传播  optimizer.zero\_grad()  loss.backward()  optimizer.step()  running\_loss += loss.item()  # 打印loss  if i % 100 == 99:  print('[%d, %3d/%3d] loss: %.3f' %(epoch + 1, i + 1,len(trainload), running\_loss / 100))  running\_loss = 0.0 |

## 2.2 决策树

### 2.1.1 算法原理

决策树是一种基于特征条件的分类和回归算法。它通过递归地选择最佳特征进行数据划分，构建一棵规则树，用于预测新样本的类别或数值。

决策树的一般构建步骤如下：

1.选择划分特征：

从训练数据集中选择最佳的划分特征。通常使用信息增益、基尼不纯度等准则来评估特征的优劣。

2.划分数据集：

使用选定的划分特征将数据集划分为多个子集。每个子集包含相同特征值的样本。

3.递归构建子树：

对每个子集递归地应用步骤1和步骤2，直到满足停止条件（如样本数小于阈值、树的深度达到预设值等）为止。

4.确定叶子节点：

当达到停止条件或无法继续划分时，确定当前节点为叶子节点，并确定叶子节点的类别或数值。

5.剪枝处理：

对构建完成的决策树进行剪枝处理，减少过拟合的风险，提高泛化能力。

6.预测：

使用构建好的决策树对新样本进行预测。从根节点开始，根据特征值逐步遍历树的分支，最终确定样本的类别或数值。

### 2.1.2 实现思路

我在自行编写决策树代码时遭遇了很多困难，只能写出二元分类的决策树，但由于本次任务为多元分类任务，考虑过将标签值进行one-hot-encoding后再进行分类，但又发现此时标签值维度不再为1，遂遭遇困难，最终决定使用scikit-learn库函数。

使用scikit-learn库函数构建决策树十分简单，只需要准备好数据，创建分类器，对模型进行拟合即可。

|  |
| --- |
| # 创建分类器  clf = DecisionTreeClassifier()  #训练模型  clf.fit(x\_train, y\_train)  #预测  y\_pred = clf.predict(x\_train) |

DecisionTreeClassifier()不添加任何参数时，使用默认参数构建决策树，而实际上，它有很多的参数可供调整。于是，我决定将重心由自己构建决策树转至观察不同的参数对决策树构建的影响上。

|  |
| --- |
| #参数及默认值  class sklearn.tree.DecisionTreeClassifier(  criterion='gini',  splitter='best',  max\_depth=None,  min\_samples\_split=2,  min\_samples\_leaf=1,  min\_weight\_fraction\_leaf=0.0,  max\_features=None,  random\_state=None,  max\_leaf\_nodes=None,  min\_impurity\_decrease=0.0,  class\_weight=None,  ) |

其中：

1.criterion ： string，optional（default =“gini”）

特征选择标准，可选参数，默认是gini，可以设置为entropy。gini是基尼不纯度，是将来自集合的某种结果随机应用于某一数据项的预期误差率，是一种基于统计的思想。entropy是香农熵，是一种基于信息论的思想。

2.splitter ： string，optional（default =“best”）

特征划分点选择标准，可选参数，默认是best，可以设置为random。每个结点的选择策略。best参数是根据算法选择最佳的切分特征，例如gini、entropy。random随机的在部分划分点中找局部最优的划分点。默认的”best”适合样本量不大的时候，而如果样本数据量非常大，此时决策树构建推荐”random”。

3.max\_depth ： int或None，可选（default =None）

决策树最大深，可选参数，默认是None。这个参数是这是树的层数的。层数的概念就是，比如在贷款的例子中，决策树的层数是2层。如果这个参数设置为None，那么决策树在建立子树的时候不会限制子树的深度。一般来说，数据少或者特征少的时候可以不管这个值。或者如果设置了min\_samples\_slipt参数，那么直到少于min\_smaples\_split个样本为止。如果模型样本量多，特征也多的情况下，推荐限制这个最大深度，具体的取值取决于数据的分布。常用的可以取值10-100之间。

4. min\_samples\_split ： int，float，optional（default = 2）

内部节点再划分所需最小样本数，可选参数，默认是2。这个值限制了子树继续划分的条件。如果min\_samples\_split为整数，那么在切分内部结点的时候，min\_samples\_split作为最小的样本数，也就是说，如果样本已经少于min\_samples\_split个样本，则停止继续切分。如果min\_samples\_split为浮点数，那么min\_samples\_split就是一个百分比，ceil(min\_samples\_split \* n\_samples)，数是向上取整的。如果样本量不大，不需要管这个值。如果样本量数量级非常大，则推荐增大这个值。

5. min\_samples\_leaf ： int，float，optional（default = 1）

叶节点所需的最小样本数。只有min\_samples\_leaf在每个左右分支中留下至少训练样本时，才会考虑任何深度的分裂点。这可能具有平滑模型的效果，尤其是在回归中。

6.min\_weight\_fraction\_leaf ： float，optional（default= 0。）

叶子节点最小的样本权重和，可选参数，默认是0。这个值限制了叶子节点所有样本权重和的最小值，如果小于这个值，则会和兄弟节点一起被剪枝。一般来说，如果我们有较多样本有缺失值，或者分类树样本的分布类别偏差很大，就会引入样本权重，这时我们就要注意这个值了。

7.max\_features ： int，float，string或None，可选（default =None）

划分时考虑的最大特征数，可选参数，默认是None。寻找最佳切分时考虑的最大特征数(n\_features为总共的特征数)，有如下6种情况：

如果max\_features是整型的数，则考虑max\_features个特征；

如果max\_features是浮点型的数，则考虑int(max\_features \* n\_features)个特征；

如果max\_features设为auto，那么max\_features = sqrt(n\_features)；

如果max\_features设为sqrt，那么max\_featrues = sqrt(n\_features)，跟auto一样；

如果max\_features设为log2，那么max\_features = log2(n\_features)；

如果max\_features设为None，那么max\_features = n\_features，也就是所有特征都用。

一般来说，如果样本特征数不多，比如小于50，我们用默认的”None”就可以了，如果特征数非常多，我们可以灵活使用刚才描述的其他取值来控制划分时考虑的最大特征数，以控制决策树的生成时间。

8.random\_state ： int，RandomState实例或None，可选（default =None）

可选参数，默认是None。随机数种子。如果是证书，那么random\_state会作为随机数生成器的随机数种子。随机数种子，如果没有设置随机数，随机出来的数与当前系统时间有关，每个时刻都是不同的。如果设置了随机数种子，那么相同随机数种子，不同时刻产生的随机数也是相同的。如果是RandomState instance，那么random\_state是随机数生成器。如果为None，则随机数生成器使用np.random。

9.max\_leaf\_nodes ： int或None，可选（default =None）

最大叶子节点数，可选参数，默认是None。通过限制最大叶子节点数，可以防止过拟合。如果加了限制，算法会建立在最大叶子节点数内最优的决策树。如果特征不多，可以不考虑这个值，但是如果特征分成多的话，可以加以限制，具体的值可以通过交叉验证得到。

10.class\_weight ： dict，dicts 列表，“balanced”或None，default =None

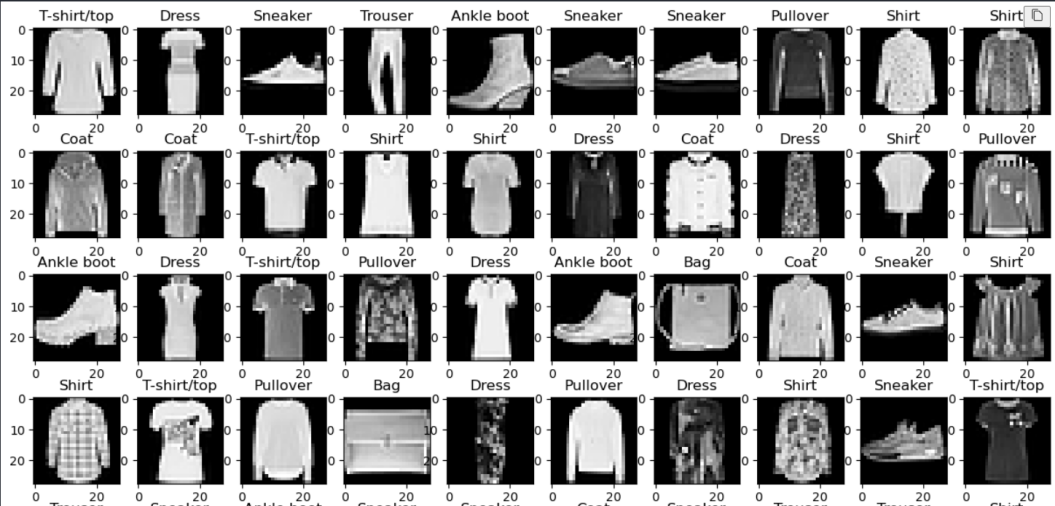
类别权重，可选参数，默认是None，也可以字典、字典列表、balanced。指定样本各类别的的权重，主要是为了防止训练集某些类别的样本过多，导致训练的决策树过于偏向这些类别。类别的权重可以通过{class\_label：weight}这样的格式给出，这里可以自己指定各个样本的权重，或者用balanced，如果使用balanced，则算法会自己计算权重，样本量少的类别所对应的样本权重会高。当然，如果你的样本类别分布没有明显的偏倚，则可以不管这个参数，选择默认的None。

# 3 训练结果及对比分析

## 3.0 训练数据组成

在寻找最优参数时，两个算法分别使用1个epoch，每个epoch里使用enumerate(trainload,start=0)加载训练数据，训练数据样本数量为60000；

在对比两个算法时，两个算法分别使用10个epoch，每个epoch里使用enumerate(trainload,start=0)加载训练数据，训练数据样本数量为600000。



**Fig. 1.**部分训练数据可视化结果

## 3.1 训练结果评价指标

在本次分类任务中，使用Weight-average方法作为评价指标，即对每类的评价指标进行加权求和,求得整个的评价指标

1. 精度（Accuracy）

Precision = TP / (TP + FP)

1. 召回率（Recall）

Recall = TP / (TP + FN)

1. 准确率（Precision）

Accuracy = (TP + TN) / (TP + FN + FP + TN)

1. F1-score

F1 = (2×Precision×Recall) / (Precision+Recall）

1. 误差

loss=loss\_function(outputs,labels)，实际计算时，每一个批次训练后取loss取平均

1. 训练时间，执行时间

由于使用notebook进行编写，可以直接查看

## 3.2 寻找最优参数

采用控制变量法寻找最优参数。评价指标训练三次，取平均值。

### 3.2.1 多层感知机

#### 1. 隐层数量

#### 隐层数量为128，256，512时对应的数据如下：

**Table 1.** 不同隐层数量评价指标对比（optimizer=Adam）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 隐层数量 | Loss | Accuracy | Precision | Recall | F1-socre | 训练时间/s | 执行时间/s |
| 64 | 0.463 | 0.966 | 0.832 | 0.831 | 0.829 | 10.0 | 1.7 |
| 128 | 0.431 | 0.967 | 0.837 | 0.841 | 0.832 | 10.0 | 1.6 |
| 256 | 0.403 | 0.968 | 0.840 | 0.843 | 0.839 | 10.1 | 1.7 |
| 512 | 0.400 | 0.970 | 0.848 | 0.849 | 0.847 | 10.5 | 1.7 |
| 1024 | 0.399 | 0.969 | 0.846 | 0.849 | 0.847 | 11.6 | 1.8 |

#### 可以看到，隐层数量增加后，对同样的训练样本，训练效果更好，但训练时间和执行时间也更长，因为模型更加复杂。

#### 所以这里选用隐层数量为256层。

#### 优化器

#### 优化器为SGD，Adam时对应的数据如下：

**Table 2.** 不同优化器评价指标对比（隐层数量=256）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Optimizer | Loss | Accuracy | Precision | Recall | F1-socre | 训练时间/s | 执行时间/s |
| SGD | 1.482 | 0.930 | 0.652 | 0.667 | 0.627 | 9.5 | 1.6 |
| Adam | 0.409 | 0.968 | 0.842 | 0.847 | 0.843 | 10.0 | 1.6 |

#### 

可以看到SGD算法训练时间更短，但训练效果差了不少。所以这里选用Adam算法。

### 3.2.2 决策树

1.critierion

**Table 3.** 不同criterion评价指标对比

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Criterion | Loss | Accuracy | Precision | Recall | F1-socre | 训练时间/s | 执行时间/s |
| gini | 0.532 | 0.958 | 0.790 | 0.791 | 0.790 | 41.1 | 1.3 |
| entropy | 0.532 | 0.960 | 0.798 | 0.799 | 0.799 | 32.8 | 1.3 |

Entropy的训练时间更短，选用entropy。

2.splitter

**Table 4.** 不同splitter评价指标对比

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Splitter | Loss | Accuracy | Precision | Recall | F1-socre | 训练时间/s | 执行时间/s |
| Best | 0.532 | 0.960 | 0.801 | 0.802 | 0.801 | 32.9 | 1.3 |
| Random | 0.532 | 0.958 | 0.791 | 0.793 | 0.792 | 6.1 | 1.3 |

可以看到使用splitter=Random和Best的训练效果差不多，但Random的训练速度显著快于Best，所以这里采用Random。

3.max\_depth，

**Table 5.** 不同max\_depth评价指标对比

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Max\_depth | Loss | Accuracy | Precision | Recall | F1-socre | 训练时间/s | 执行时间/s |
| 10 | 0.544 | 0.961 | 0.804 | 0.804 | 0.802 | 3.9 | 1.4 |
| 20 | 0.533 | 0.958 | 0.791 | 0.792 | 0.791 | 5.9 | 1.3 |
| 50 | 0.532 | 0.959 | 0.795 | 0.796 | 0.795 | 5.8 | 1.3 |
| 100 | 0.532 | 0.959 | 0.794 | 0.796 | 0.795 | 5.9 | 1.3 |

可以发现不同max\_depth下训练效果相差无几，但max\_depth越大，模型越复杂，根据剃刀原理，这里选用max\_depth-=10.

1. max\_samples\_split

**Table 6.** 不同max\_samples\_split评价指标对比

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| max\_samples\_split | Loss | Accuracy | Precision | Recall | F1-socre | 训练时间/s | 执行时间/s |
| 2 | 0.543 | 0.960 | 0.799 | 0.800 | 0.798 | 3.8 | 1.3 |
| 3 | 0.544 | 0.960 | 0.799 | 0.800 | 0.798 | 3.9 | 1.3 |
| 5 | 0.546 | 0.958 | 0.791 | 0.791 | 0.788 | 3.8 | 1.3 |
| 10 | 0.545 | 0.960 | 0.799 | 0.800 | 0.797 | 3.8 | 1.3 |

可以发现不同max\_samples\_split下训练效果相差无几，但max\_depth越大，模型越复杂，根据剃刀原理，这里选用max\_samples\_split-=2.

1. max\_samples\_leaf

**Table 7.** 不同max\_samples\_leaft评价指标对比

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| max\_samples\_leaf | Loss | Accuracy | Precision | Recall | F1-socre | 训练时间/s | 执行时间/s |
| 1 | 0.542 | 0.960 | 0.799 | 0.801 | 0.800 | 3.9 | 1.3 |
| 2 | 0.544 | 0.959 | 0.794 | 0.796 | 0.794 | 3.9 | 1.3 |
| 4 | 0.546 | 0.960 | 0.798 | 0.799 | 0.796 | 3.8 | 1.3 |
| 8 | 0.546 | 0.960 | 0.802 | 0.802 | 0.801 | 3.8 | 1.3 |

选用max\_samples\_leaf=1.

1. max\_features

**Table8.** 不同max\_features评价指标对比

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Max\_features | Loss | Accuracy | Precision | Recall | F1-socre | 训练时间/s | 执行时间/s |
| 1(int) | 0.542 | 0.904 | 0.521 | 0.518 | 0.502 | 0.0 | 1.3 |
| 5(int) | 0.551 | 0.926 | 0.631 | 0.634 | 0.631 | 0.0 | 1.3 |
| 0.5(float) | 0.544 | 0.958 | 0.791 | 0.792 | 0.790 | 2.0 | 1.3 |
| 1.0(float) | 0.543 | 0.959 | 0.795 | 0.796 | 0.793 | 4.0 | 1.3 |
| sqrt | 0.549 | 0.947 | 0.737 | 0.734 | 0.734 | 0.2 | 1.3 |
| Log2 | 0.546 | 0.942 | 0.709 | 0.710 | 0.708 | 0.1 | 1.3 |
| None | 0.542 | 0.960 | 0.801 | 0.803 | 0.800 | 4.0 | 1.3 |

选用max\_features=0.5。

1. max\_leaf\_nodes

**Table 9.** 不同max\_leaf\_nodes评价指标对比

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| max\_leaf\_nodes | Loss | Accuracy | Precision | Recall | F1-socre | 训练时间/s | 执行时间/s |
| None | 0.544 | 0.958 | 0.789 | 0.791 | 0.787 | 2.0 | 1.3 |
| 10 | 0.570 | 0.923 | 0.615 | 0.616 | 0.615 | 1.1 | 1.3 |
| 20 | 0.551 | 0.937 | 0.683 | 0.683 | 0.665 | 1.4 | 1.3 |
| 40 | 0.547 | 0.947 | 0.733 | 0.736 | 0.730 | 1.6 | 1.3 |
| 80 | 0.550 | 0.952 | 0.760 | 0.759 | 0.755 | 1.8 | 1.3 |
| 160 | 0.547 | 0.952 | 0.761 | 0.756 | 0.757 | 1.9 | 1.3 |
| 320 | 0.546 | 0.957 | 0.784 | 0.784 | 0.782 | 2.1 | 1.3 |

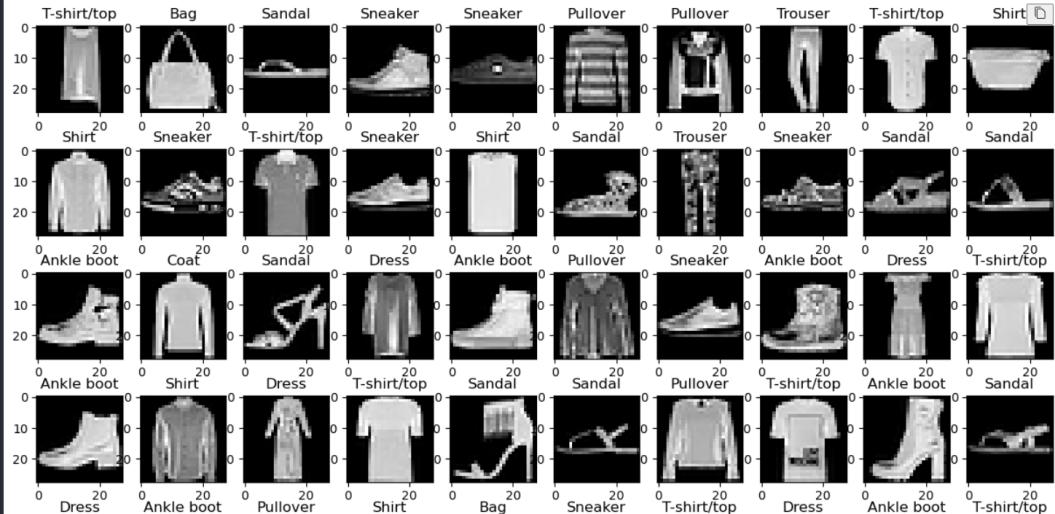
选用max\_leaf\_nodes=None.

## 3.3 不同算法的对比

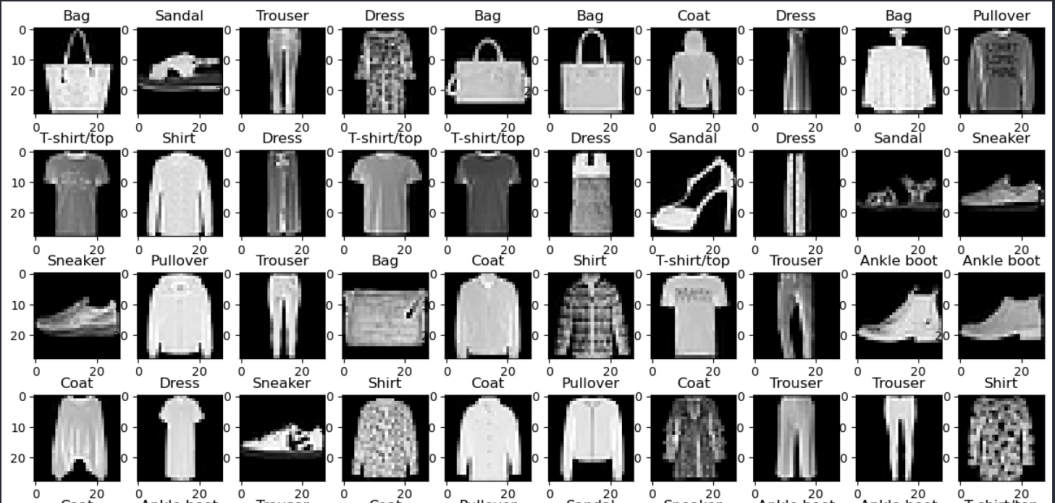
**Table10.** 不同算法评价指标对比（训练三次取平均）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 算法 | Loss | Accuracy | Precision | Recall | F1-socre | 训练时间/s | 执行时间/s |
| MLP | 0.242 | 0.977 | 0.885 | 0.885 | 0.884 | 97.2 | 0.242 |
| Desicion\_tree | 0.647 | 0.958 | 0.788 | 0.789 | 0.786 | 106.8 | 1.6 |

通过对比可以发现，无论是训练时间（包括从trainload取数据的时间），还是各项指标，MLP都优于决策树。这可能是因为MLP已经使用了简单的神经网络架构，从结构上来说算法更加完善，而决策树在面对如此庞大的数据集时，很可能出现了过拟合问题。



**Fig. 2.**决策树算法部分预测结果可视化

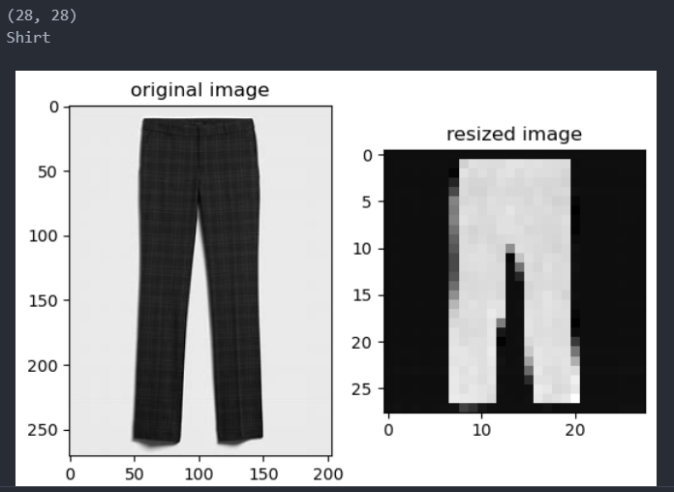


**Fig.3.**MLP算法部分预测结果可视化

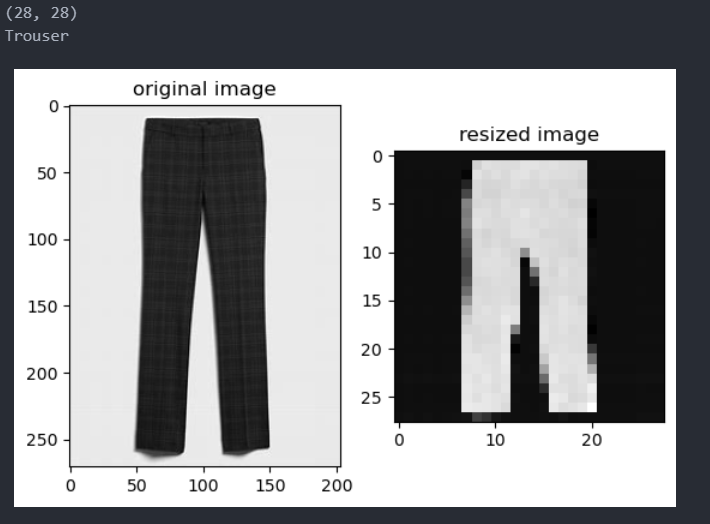
## 3.4 分类真实图片

可以使用这两个模型来进行真实图片的预测：

|  |
| --- |
| #MLP预测图片  im = cv2.imread('./source/trouser.jpg')  im = cv2.cvtColor(im, cv2.COLOR\_BGR2GRAY)  resized\_image = cv2.resize(im, (28, 28))  print(resized\_image.shape)  #将图片反相  resized\_image = 255 - resized\_image  #预处理图片  input = transform(resized\_image)  # 把resize后的图片转化为列向量  input = input.view(1, -1)  # 把im转化为tensor  input = input.clone().detach().requires\_grad\_(True)  plt.subplot(1,2,1)  plt.imshow(im, cmap='gray')  plt.title('original image')  plt.subplot(1,2,2)  plt.imshow(resized\_image, cmap='gray')  plt.title('resized image')  with torch.no\_grad():  outputs = model(input)  \_, predict = torch.max(outputs.data, 1) # 索引即classed中的类别  print(classes[predict]) |
| #desicion\_tree 预测图片  #预测图片  im = cv2.imread('./source/trouser.jpg')  im = cv2.cvtColor(im, cv2.COLOR\_BGR2GRAY)  resized\_image = cv2.resize(im, (28, 28))  print(resized\_image.shape)  #将图片反相  resized\_image = 255 - resized\_image  #预处理图片  input = transform(resized\_image)  # 把resize后的图片转化为列向量  input = input.view(1, -1)  plt.subplot(1,2,1)  plt.imshow(im, cmap='gray')  plt.title('original image')  plt.subplot(1,2,2)  plt.imshow(resized\_image, cmap='gray')  plt.title('resized image')  outputs = clf.predict(input)  predict = outputs[0].astype(int)  print(classes[predict]) |



**Fig. 4.**决策树算法预测真实图片



**Fig.5**MLP算法预测真实图片

# 4 总结

传统机器学习算法相对于现今常使用的transformer或编码解码器模型来说架构较简单，但对于小规模的分类任务也可以达到一定的指标要求。传统机器学习算法是我们学习更复杂的深度学习算法等的基础，他们之中有很多地方，如如何优化模型参数的过程都是相通的。

通过对比不同算法的优劣，也可以发现不同算法之间的优缺点，对应不同场合，选择更合适的算法。

References

1. [机器学习——二分类、多分类的精确率和召回率\_多分类的准确率和召回率-CSDN博客](https://blog.csdn.net/Alex_81D/article/details/115233536?ops_request_misc=%7B%22request%5Fid%22%3A%22171465328016800225571707%22%2C%22scm%22%3A%2220140713.130102334.pc%5Fall.%22%7D&request_id=171465328016800225571707&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-2-115233536-null-null.142^v100^control&utm_term=%E5%A4%9A%E5%88%86%E7%B1%BB%E9%97%AE%E9%A2%98%E7%B2%BE%E5%BA%A6%E5%8F%AC%E5%9B%9E%E7%8E%87%E8%AF%AF%E5%B7%AE&spm=1018.2226.3001.4187)
2. [《机器学习实战》学习笔记（三）：决策树\_希望通过所给的训练数据学习一个根据头发和-CSDN博客](https://blog.csdn.net/TeFuirnever/article/details/99955515)
3. [Pytorch模型训练&保存/加载（搭建完整流程）\_pytorch训练模型-CSDN博客](https://blog.csdn.net/FUTEROX/article/details/122724634)
4. [peymanh/ID5-Incremental-Desicion-Tree: Implementation of ID5 algorithm in Python Language, an incremental approach to build a decision Tree. (github.com)](https://github.com/peymanh/ID5-Incremental-Desicion-Tree)