```python
import numpy as np
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from skimage.feature import local_binary_pattern, hog
import cv2
import warnings
warnings.filterwarnings('ignore')

# 加载 CIFAR-100 数据集
from keras.datasets import cifar100
(X_train, y_train), (X_test, y_test) = cifar100.load_data()

# LBP 特征提取
def extract_lbp_features(image):
    #print(image.shape)
    gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    lbp = local_binary_pattern(gray_image, P=8, R=1, method='uniform')
    lbp_hist, _ = np.histogram(lbp, bins=np.arange(257), density=True)
    return lbp_hist

# HOG 特征提取
def extract_hog_features(image):
    gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    hog_features = hog(gray_image, pixels_per_cell=(8, 8), cells_per_bl
ock=(2, 2), feature_vector=True)
    return hog_features


print(X_train[0].shape)  # 检查图像数据的形状



# SIFT 特征提取（带默认值）
def extract_sift_features(image):
    gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    sift = cv2.SIFT_create()
    keypoints, descriptors = sift.detectAndCompute(gray_image, None)
    if descriptors is not None and descriptors.shape[0] > 256:
        descriptors = descriptors[:256]
    elif descriptors is None:
        descriptors = np.zeros((256, 128))
    return descriptors.flatten() if descriptors is not None else np.zer
os((256*128,))
```

```python
# 提取所有图像的特征
def extract_features(images, feature_extractor):
    features = []
    for img in images:
        features.append(feature_extractor(img))
    return np.array(features)

# 特征提取和降维
def process_features(X_train, X_test, feature_extractor):
    train_features = extract_features(X_train, feature_extractor)
    test_features = extract_features(X_test, feature_extractor)

    # 使用PCA 进行降维
    pca = PCA(n_components=50)
    print(train_features.shape)
    train_features_pca = pca.fit_transform(train_features)
    test_features_pca = pca.transform(test_features)

    return train_features_pca, test_features_pca

# 分类模型训练和评估
import joblib
def evaluate_model(model_name, model, X_train, y_train, X_test, y_test):
    model.fit(X_train, y_train)
    # 将模型参数持久化存储下来
    joblib.dump(model, model_name + '.pkl')
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    return accuracy

# 提取并处理LBP 特征
X_train_lbp, X_test_lbp = process_features(X_train, X_test, extract_lbp
_features)
# 提取并处理HOG 特征
X_train_hog, X_test_hog = process_features(X_train, X_test, extract_hog
_features)
# 提取并处理SIFT 特征
X_train_sift, X_test_sift = process_features(X_train, X_test, extract_s
ift_features)

# 使用分类模型进行分类
models = {
    GaussianNB(),#'朴素贝叶斯模型 '
     KNeighborsClassifier(),#'KNN 模型 '
    LogisticRegression(max_iter=100)# '逻辑回归模型 '
}
```

```
# 对每种特征进行分类评估
for feature_name, (X_train_feat, X_test_feat) in zip(['LBP', 'HOG', 'SIFT'],
                                                     [(X_train_lbp, X_test_lbp), (X_train_hog, X_test_hog), (X_train_sift, X_test_sift)]):
    print(f"采用 {feature_name} 特征进行分类的结果为:")
    for model_name, model in models.items():
        accuracy = evaluate_model(model_name, model, X_train_feat, y_train.ravel(), X_test_feat, y_test.ravel())
        print(f"{model_name} Accuracy: {accuracy:.4f}")
```

```
(32, 32, 3)
(50000, 256)
(50000, 324)
(50000, 32768)
采用 LBP 特征进行分类的结果为:
GaussianNB Accuracy: 0.0732
保存 GaussianNB 模型为: GaussianNB.pkl
KNeighborsClassifier Accuracy: 0.0494
保存 KNeighborsClassifier 模型为: KNeighborsClassifier.pkl
LogisticRegression Accuracy: 0.0578
保存 LogisticRegression 模型为: LogisticRegression.pkl
采用 HOG 特征进行分类的结果为:
GaussianNB Accuracy: 0.1988
保存 GaussianNB 模型为: GaussianNB.pkl
KNeighborsClassifier Accuracy: 0.1785
保存 KNeighborsClassifier 模型为: KNeighborsClassifier.pkl
LogisticRegression Accuracy: 0.1937
保存 LogisticRegression 模型为: LogisticRegression.pkl
采用 SIFT 特征进行分类的结果为:
GaussianNB Accuracy: 0.0550
保存 GaussianNB 模型为: GaussianNB.pkl
KNeighborsClassifier Accuracy: 0.0307
保存 KNeighborsClassifier 模型为: KNeighborsClassifier.pkl
LogisticRegression Accuracy: 0.0610
保存 LogisticRegression 模型为: LogisticRegression.pkl
```

表 1：不同特征提取算法和分类器的实验数据(%)

	模型		
	朴素贝叶斯	KNN	逻辑回归
LBP	0.073	0.049	0.0578
HOG	0.1988	0.1785	0.1937
SIFT	0.055	0.0307	0.0610

发现：从提供的实验数据中，我们可以观察到不同特征提取算法（LBP、HOG、SIFT）与不同分类器（朴素贝叶斯、KNN、逻辑回归）组合的表现，具体体现在它们对数据分类的准确率上。基于这些数据，可以得出以下几个结论：

1.HOG 特征表现最佳：在所有测试的特征提取方法中，HOG（Histogram of Oriented

Gradients）特征不论与哪种分类器结合（朴素贝叶斯、KNN、逻辑回归），都表现出最高的分类准确率（分别为 0.1988%, 0.1785%, 和 0.1937%）。这表明 HOG 特征在所研究的场景或数据集中对于图像内容的描述能力最强。
2.SIFT 特征与 LBP 特征比较：相比之下，SIFT（Scale-Invariant Feature Transform）特征虽然在某些情况下表现优于 LBP（Local Binary Patterns），但总体上两者都不是最优选择。特别是 LBP 特征在所有配置下的表现都是最低的，说明它可能不太适合用于当前数据集的特征表达。SIFT 在与逻辑回归结合时的表现略好于其他两种特征提取方法与逻辑回归的组合，但依然低于 HOG 的表现。
3. 分类器性能差异：在使用 HOG 特征时，三种分类器中逻辑回归给出了最高的准确率（0.1937%），而 KNN 在 HOG 特征上的表现次之，朴素贝叶斯则在这三种特征上都给出了最低的准确率。这可能意味着逻辑回归在处理这类问题时具有更好的泛化能力，能够更好地拟合复杂的决策边界。
4.综合考虑：如果目标是追求最高分类准确率，基于实验数据，应优先考虑使用 HOG 特征配合逻辑回归分类器。然而，实际应用中还需考虑计算效率、模型复杂度以及训练时间等因素，因此最终选择可能会有所调整。
综上所述，HOG 特征提取方法在此次实验中展现出了最佳的性能，特别是在与逻辑回归分类器结合时，而 LBP 特征则表现相对较弱。选择最合适的特征提取方法和分类器组合对于提升特定任务的性能至关重要。

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.models as models


class MLP(nn.Module):
    def __init__(self):
        super().__init__()

        # self.conv = nn.Conv2d(3, 1, 3, 1, 1)
        self.fc = nn.Linear(3 * 32 * 32, 32 * 32)

        self.fc1 = nn.Linear(32 * 32, 512)
        self.ac = nn.LeakyReLU()
        self.fc2 = nn.Linear(512, 100)
        self.drop = nn.Dropout(0.01)


    def forward(self, x):
        B,_, _, _ = x.shape
        y = self.drop(self.ac(self.fc(x.view(B, -1))))
        y = self.drop(self.ac(self.fc1(y)))
        return self.fc2(y)


class SE(nn.Module):
```

```
    def __init__(self, in_chnls, ratio):
        super(SE, self).__init__()
        self.squeeze = nn.AdaptiveAvgPool2d((1, 1))
        self.compress = nn.Conv2d(in_chnls, in_chnls // ratio, 1, 1, 0)
        self.excitation = nn.Conv2d(in_chnls // ratio, in_chnls, 1, 1,
0)

    def forward(self, x):
        out = self.squeeze(x)
        out = self.compress(out)
        out = F.relu(out)
        out = self.excitation(out)
        return x*torch.sigmoid(out)


import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.models as models

device = 'cuda' if torch.cuda.is_available() else 'cpu'
class BasicBlock(nn.Module):
    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
        # 第一个卷积层
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride
=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        # 第二个卷积层
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1,
 padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        # shortcut
        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != planes:
            # 如果输入输出通道数不同或者步长不为1，则使用1x1 卷积层
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, planes, kernel_size=1, stride=stri
de, bias=False),
                nn.BatchNorm2d(planes)
            )

        # SE layers
        self.fc1 = nn.Conv2d(planes, planes//16, kernel_size=1)  # Use
nn.Conv2d instead of nn.Linear
        self.fc2 = nn.Conv2d(planes//16, planes, kernel_size=1)
```

```
    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))

        # Squeeze
        w = F.avg_pool2d(out, out.size(2))
        w = F.relu(self.fc1(w))
        w = F.sigmoid(self.fc2(w))
        # Excitation
        out = out * w

        out += self.shortcut(x)
        out = F.relu(out)
        return out
class PreActBlock(nn.Module):
    def __init__(self, in_planes, planes, stride=1):
        super(PreActBlock, self).__init__()
        self.bn1 = nn.BatchNorm2d(in_planes)
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride
=stride, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1,
 padding=1, bias=False)

        if stride != 1 or in_planes != planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, planes, kernel_size=1, stride=stri
de, bias=False)
            )

        # SE layers
        self.fc1 = nn.Conv2d(planes, planes//16, kernel_size=1)
        self.fc2 = nn.Conv2d(planes//16, planes, kernel_size=1)

    def forward(self, x):
        out = F.relu(self.bn1(x))
        shortcut = self.shortcut(out) if hasattr(self, 'shortcut') else
 x
        out = self.conv1(out)
        out = self.conv2(F.relu(self.bn2(out)))

        # Squeeze
        w = F.avg_pool2d(out, out.size(2))
        w = F.relu(self.fc1(w))
        w = F.sigmoid(self.fc2(w))
        # Excitation
        out = out * w

        out += shortcut
```

```
        return out

class SENet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(SENet, self).__init__()
        self.in_planes = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=
1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.layer1 = self._make_layer(block,  64, num_blocks[0], strid
e=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], strid
e=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], strid
e=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], strid
e=2)
        self.linear = nn.Linear(512, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out


def SENet18(num_classes=100):
    return SENet(PreActBlock, [2,2,2,2], num_classes=num_classes)


net = SENet(num_classes=100)


import torch.optim as optim
optimizer = optim.SGD(net.parameters(), lr=1e-1, momentum=0.9, weight_d
```

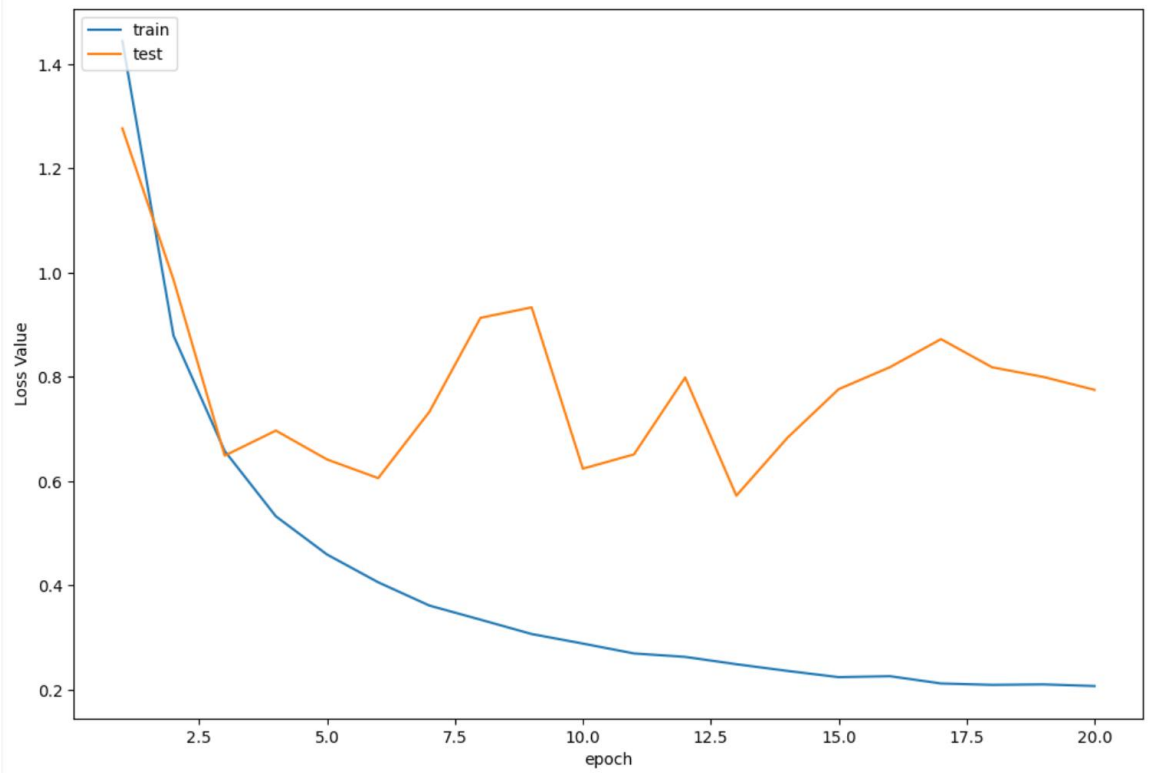
```
ecay=5e-4)
criterion = nn.CrossEntropyLoss()
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', fact
or=0.94 ,patience = 1,min_lr = 0.000001) # 动态更新学习率



import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# 定义数据预处理
transform = transforms.Compose([
    # transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2
761))  # CIFAR-100 mean and std
])

# 加载 CIFAR-100 数据集
train_dataset = datasets.CIFAR100(root='./data', train=True, download=F
alse, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=512, shuffle=True,
num_workers=0)

test_dataset = datasets.CIFAR100(root='./data', train=False, download=F
alse, transform=transform)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False, nu
m_workers=0)

import torch.optim as optim

def train(model, train_loader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

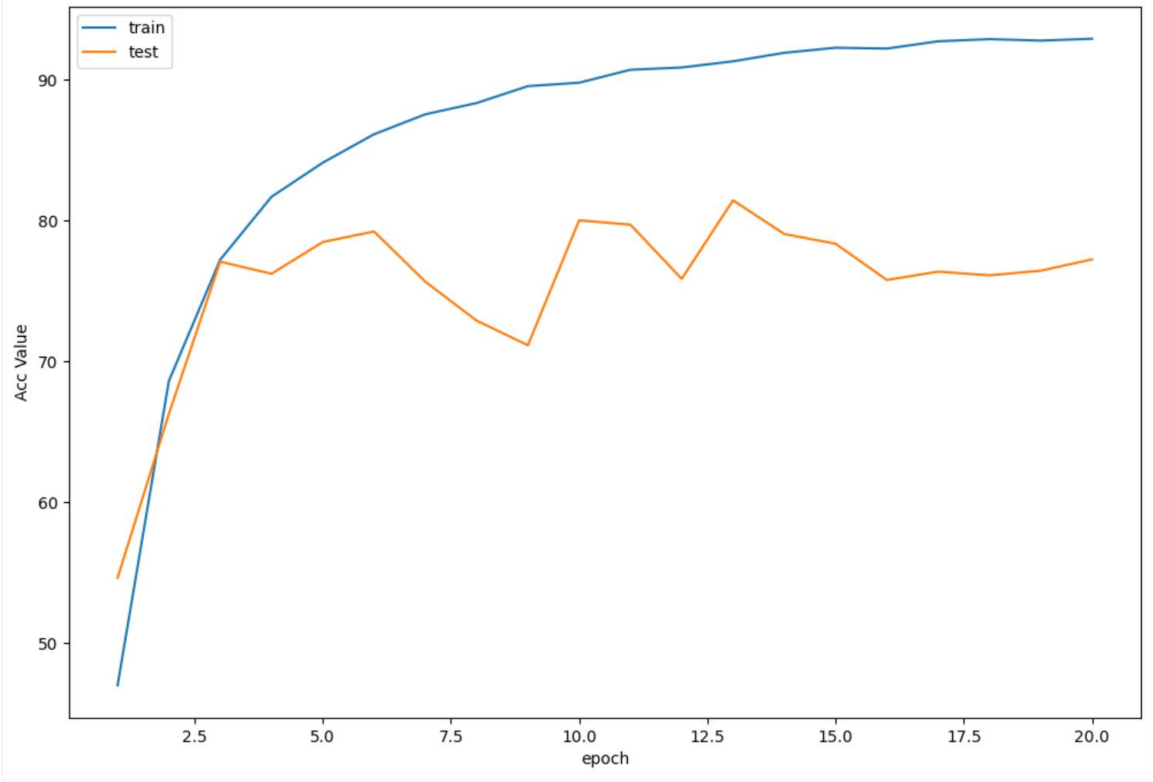
```
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader)
        epoch_acc = correct / total
        return epoch_loss, epoch_acc

    def validate(model, test_loader, criterion, device):
        model.eval()
        running_loss = 0.0
        correct = 0
        total = 0

        with torch.no_grad():
            for inputs, labels in test_loader:
                inputs, labels = inputs.to(device), labels.to(device)

                outputs = model(inputs)
                loss = criterion(outputs, labels)

                running_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(test_loader)
        epoch_acc = correct / total
        return epoch_loss, epoch_acc

    from model import *

    def main():
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu
    ')

        transform = transforms.Compose([
            #transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565,
     0.2761))  # CIFAR-100 mean and std
        ])

        train_dataset = datasets.CIFAR100(root='./data', train=True, downlo
    ad=True, transform=transform)
        train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True,
     num_workers=0)
```

```
    test_dataset = datasets.CIFAR100(root='./data', train=False, downlo
ad=True, transform=transform)
    test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False,
 num_workers=0)

    model = SENet()
    # model = MLP()
    model = model.to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=.01)

    num_epochs = 50
    best_acc = 0.0

    for epoch in range(num_epochs):
        train_loss, train_acc = train(model, train_loader, criterion, o
ptimizer, device)
        val_loss, val_acc = validate(model, test_loader, criterion, dev
ice)

        print(f'Epoch {epoch + 1}/{num_epochs}, Train Loss: {train_los
s:.4f}, Train Acc: {train_acc:.4f}, '
              f'Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}')

        # Save the best model
        if val_acc > best_acc:
            best_acc = val_acc
            torch.save(model.state_dict(), 'best_model.pth')

    print(f'Best Validation Accuracy: {best_acc:.4f}')


if __name__ == '__main__':
    main()
```

```
Train Epoch 1/20: Train Acc=0.47, Train Loss=1.44]
Test Epoch 1/20:  Test Acc=0.546, Test Loss=1.28]
Epoch [  1/ 20]  Train Loss:1.444286  Train Acc:46.98% Test Loss:1.276667  Test Acc:54.61%  Learning Rate:0.100000
Train Epoch 2/20:  Train Acc=0.686, Train Loss=0.879]
Test Epoch 2/20: Test Acc=0.663, Test Loss=0.986]
Epoch [  2/ 20]  Train Loss:0.878925  Train Acc:68.59% Test Loss:0.986204  Test Acc:66.26%  Learning Rate:0.100000
Train Epoch 3/20: Train Acc=0.772, Train Loss=0.658]
Test Epoch 3/20: Test Acc=0.771, Test Loss=0.649]
Epoch [  3/ 20]  Train Loss:0.657689  Train Acc:77.20% Test Loss:0.648814  Test Acc:77.06%  Learning Rate:0.100000
Train Epoch 4/20: Train Acc=0.817, Train Loss=0.532]
Test Epoch 4/20: Test Acc=0.762, Test Loss=0.697]
Epoch [  4/ 20]  Train Loss:0.532490  Train Acc:81.67% Test Loss:0.696845  Test Acc:76.20%  Learning Rate:0.100000
Train Epoch 5/20: Train Acc=0.841, Train Loss=0.459]
Test Epoch 5/20: Test Acc=0.785, Test Loss=0.642]
Epoch [  5/ 20]  Train Loss:0.459192  Train Acc:84.09% Test Loss:0.641512  Test Acc:78.46%  Learning Rate:0.100000
Train Epoch 6/20:  Train Acc=0.861, Train Loss=0.406]
Test Epoch 6/20: Test Acc=0.792, Test Loss=0.606]
Epoch [  6/ 20]  Train Loss:0.405891  Train Acc:86.11% Test Loss:0.605668  Test Acc:79.21%  Learning Rate:0.100000
Train Epoch 7/20: Train Acc=0.875, Train Loss=0.361]
Test Epoch 7/20:  Test Acc=0.756, Test Loss=0.733]
Epoch [  7/ 20]  Train Loss:0.361291  Train Acc:87.53% Test Loss:0.732757  Test Acc:75.64%  Learning Rate:0.100000
Train Epoch 8/20:  Train Acc=0.883, Train Loss=0.334]
Test Epoch 8/20:  Test Acc=0.729, Test Loss=0.913]
Epoch [  8/ 20]  Train Loss:0.334168  Train Acc:88.33% Test Loss:0.913161  Test Acc:72.88%  Learning Rate:0.100000
Train Epoch 9/20: Train Acc=0.895, Train Loss=0.307]
Test Epoch 9/20: Test Acc=0.711, Test Loss=0.933]
Epoch [  9/ 20]  Train Loss:0.306639  Train Acc:89.53% Test Loss:0.933174  Test Acc:71.12%  Learning Rate:0.100000
Train Epoch 10/20: Train Acc=0.898, Train Loss=0.288]
Test Epoch 10/20:  Test Acc=0.8, Test Loss=0.624]
Epoch [ 10/ 20]  Train Loss:0.288383  Train Acc:89.77% Test Loss:0.623786  Test Acc:80.00%  Learning Rate:0.100000
Train Epoch 11/20:Train Acc=0.907, Train Loss=0.269]
Test Epoch 11/20: Test Acc=0.797, Test Loss=0.651]
Epoch [ 11/ 20]  Train Loss:0.269391  Train Acc:90.69% Test Loss:0.651273  Test Acc:79.69%  Learning Rate:0.100000
Train Epoch 12/20: Train Acc=0.909, Train Loss=0.263]
Test Epoch 12/20: Test Acc=0.758, Test Loss=0.799]
Epoch [ 12/ 20]  Train Loss:0.262791  Train Acc:90.85% Test Loss:0.798698  Test Acc:75.84%  Learning Rate:0.100000
Train Epoch 13/20: Train Acc=0.913, Train Loss=0.249]
Test Epoch 13/20: Test Acc=0.814, Test Loss=0.572]
```

```
Epoch [ 13/ 20]  Train Loss:0.248744  Train Acc:91.29% Test Loss:0.572026  Test Acc:81.42%  Learning Rate:0.100000
Train Epoch 14/20:  Train Acc=0.919, Train Loss=0.236]
Test Epoch 14/20:  Test Acc=0.79, Test Loss=0.683]
Epoch [ 14/ 20]  Train Loss:0.235874  Train Acc:91.90% Test Loss:0.683216  Test Acc:79.03%  Learning Rate:0.100000
Train Epoch 15/20: Train Acc=0.923, Train Loss=0.224]
Test Epoch 15/20: Test Acc=0.783, Test Loss=0.776]
Epoch [ 15/ 20]  Train Loss:0.223898  Train Acc:92.26% Test Loss:0.776317  Test Acc:78.34%  Learning Rate:0.100000
Train Epoch 16/20: Train Acc=0.922, Train Loss=0.226]
Test Epoch 16/20:  Test Acc=0.758, Test Loss=0.818]
Epoch [ 16/ 20]  Train Loss:0.225781  Train Acc:92.20% Test Loss:0.818387  Test Acc:75.76%  Learning Rate:0.100000
Train Epoch 17/20: Train Acc=0.927, Train Loss=0.212]
Test Epoch 17/20: Test Acc=0.764, Test Loss=0.872]
Epoch [ 17/ 20]  Train Loss:0.211836  Train Acc:92.71% Test Loss:0.872311  Test Acc:76.35%  Learning Rate:0.100000
Train Epoch 18/20:  Train Acc=0.929, Train Loss=0.209]
Test Epoch 18/20:Test Acc=0.761, Test Loss=0.818]
Epoch [ 18/ 20]  Train Loss:0.209121  Train Acc:92.87% Test Loss:0.818234  Test Acc:76.09%  Learning Rate:0.100000
Train Epoch 19/20:  Train Acc=0.928, Train Loss=0.21]
Test Epoch 19/20: Test Acc=0.764, Test Loss=0.8]
Epoch [ 19/ 20]  Train Loss:0.210039  Train Acc:92.76% Test Loss:0.799828  Test Acc:76.42%  Learning Rate:0.100000
Train Epoch 20/20:Train Acc=0.929, Train Loss=0.207]
Test Epoch 20/20:  Test Acc=0.772, Test Loss=0.775]
Epoch [ 20/ 20]  Train Loss:0.206847  Train Acc:92.89% Test Loss:0.775048  Test Acc:77.22%  Learning Rate:0.100000
```


train
test
Loss Value
epoch
1.4
1.2
1.0
0.8
0.6
0.4
0.2
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


train
test
Acc Value
epoch
90
80
70
60
50
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0