Project 2: MNIST Classification

Name: 吴曾宇
Student ID: 519021910981
GitHub：https://github.com/DDuanCang/SJTU-AICourse-Proj1---MSA

目录

SVM-MNIST

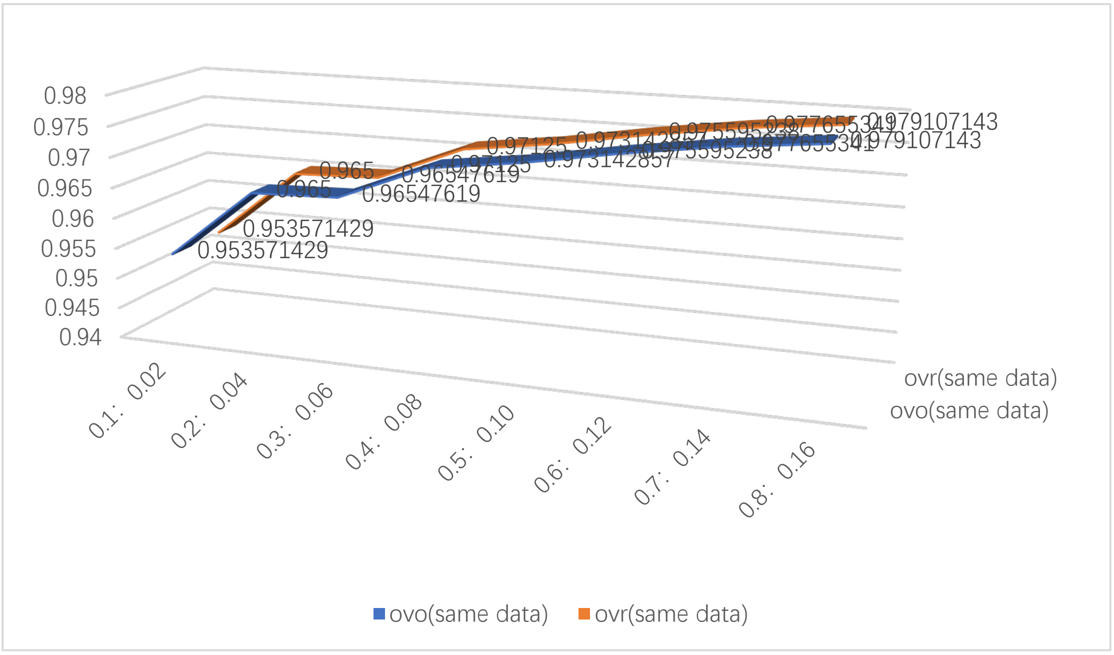
“ovo” or “ovr”？

“One vs One”：将一个 n 分类问题改为 n 个二进制分类问题。我们将一个类视为正类，将其他类视为负类，然后为该类训练分类器。重复这个过程 n 次，我们可以得到 n 个分类器。在测试阶段，对于新样本，我们将其输入到 n 个分类器中。预测得分最高的类是新样本的标签。

“One vs Rest”：为所有可能的类对训练分类器。在n分类任务中，我们有C2n个不同的类对。我们为所有对训练C2n个二元分类器。新样本的预测值可以通过a majority voting of those classifiers获得。

一般而言，“ovr”训练的分类器较少，但其性能受到数据不平衡的影响。“ovo”通常更准确，但需要大量计算。

这里通过调整hyperparameters of SVM来比较两者的区别。



PS：data总数为70000，这里通过rate来表示数据量；hyperparameters of SVM：gamma = ’scale’，C = 1，decision_function_shape，kernel = ’rbf’

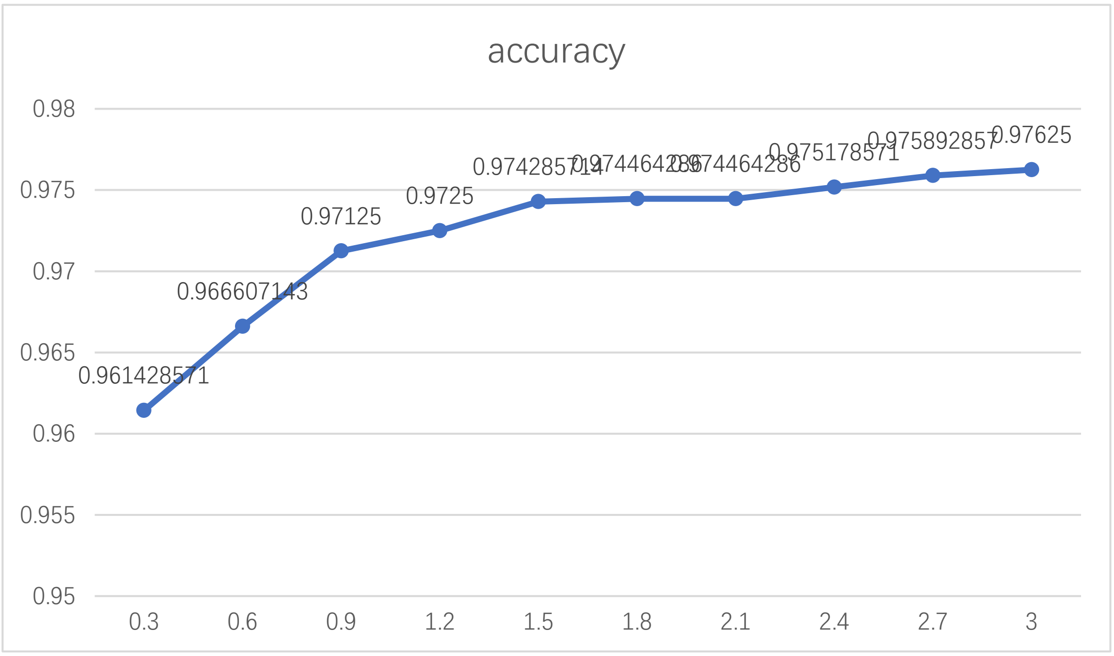
用时：ovr：1130.978s；ovo：1579.001s（这里用时为random_state=4，C=1，gamma=’scale’，kernel=’rbf’情况下，对以上所有数据情况下的计算用时）

因此可以得出，ovo与ovr在对相同数据集、相同参数的情况下，accuracy差不多(也可能是因为数据量不够大导致没有出现较大差别)，但ovo的情况下，计算耗时有明显的差别。

Hyperparameters：

当前可以调整的Hyperparameters有：gamma、C、decision_function_shape、kernel。而其中decision_function_shape用于控制ovo、ovr，kernel用于控制核函数种类，gamma为kernel中的参数。而其中C为惩罚参数，C越大，对错误的惩罚越大，C越小，泛化能力更强。

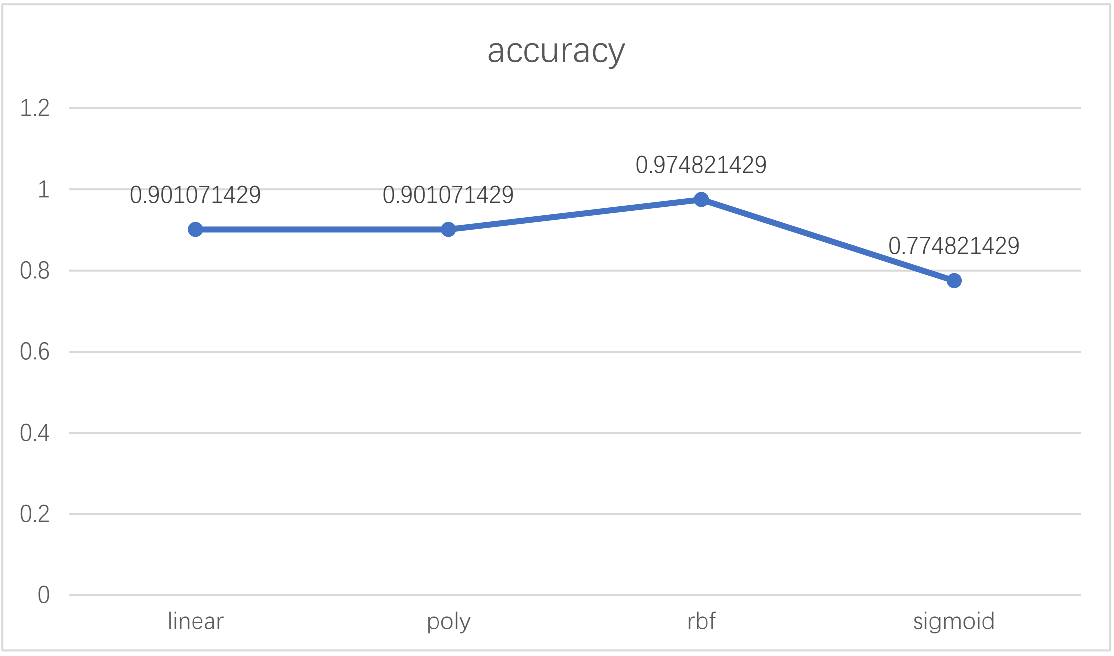
这里通过取数据集train:test=0.5:0.1(random_state=4，用于控制每次数据集相同)，decision_function_shape=‘ovr’，kernel=’rbf’，gamma=’scale’的情况下调整C来观察：



由数据可以得出，C越大时，对同样的数据集会有更好的区分效果。

这里通过调整kernel，即使用不同的核函数，来测试相同数据集且其余参数相同时的区分效果：

PS：kernel可取‘linear’，‘poly’，‘rbf’，‘sigmoid’等核函数，默认值为‘rbf’

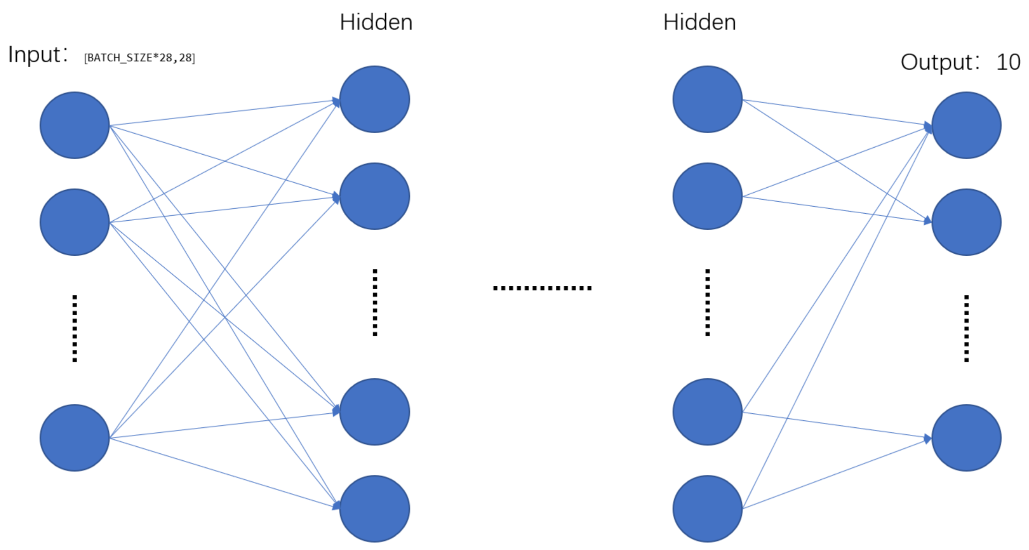


由对比数据基本可以得出，rbf为核函数时，accuracy更好。

Neural network

Build an ANN：

本模型训练采用 GPU 加速。



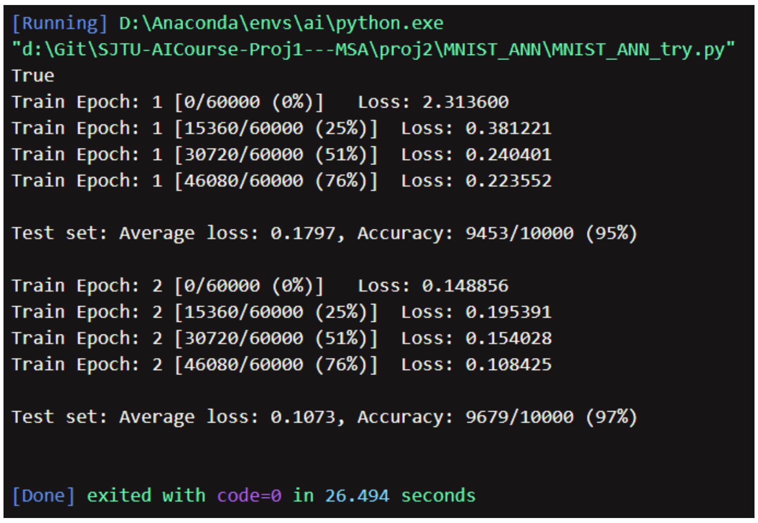
本ANN构建中，BATCH_SIZE决定了一次训练输入的数据量(本网络中取为512，需要一定的现存)；Hidden层取为3个线性层，分别为fc1:1*28*28->500、fc2:500->150、fc3:150->10，且激活函数采用torch.nn.functional.relu()；Output为输出层，在fc3输出后进行一次概率计算，该计算使用torch.nn.functional.log_softmax(output,dim=1)。

在训练过程中，优化器采用Adam(也可使用SGD优化器，取learning_rate为0.02)，需要注意的是，每次训练开始需要将优化器梯度重置。

而在训练和测试过程中，训练前需要将model转为train()模式，测试时需要将model转为测试模式。

最后进行训练，并打印出训练过程中的结果，即loss与accuracy。然后将模型进行测试，打印出loss与正确率。

在进行一次训练与测试后，将模型进行下一次的训练，此时模型中parameters为上一次训练中得出的最优参数，然后再进行一次测试，将训练与测试结果打印以观察训练效果，并与上一次进行对比。



```
[Running] D:\Anaconda\envs\ai\python.exe
"d:\Git\SJTU-AICourse-Proj1---MSA\proj2\MNIST_ANN\MNIST_ANN_try.py"
True
Train Epoch: 1 [0/60000 (0%)]	Loss: 2.313600
Train Epoch: 1 [15360/60000 (25%)]	Loss: 0.381221
Train Epoch: 1 [30720/60000 (51%)]	Loss: 0.240401
Train Epoch: 1 [46080/60000 (76%)]	Loss: 0.223552

Test set: Average loss: 0.1797, Accuracy: 9453/10000 (95%)

Train Epoch: 2 [0/60000 (0%)]	Loss: 0.148856
Train Epoch: 2 [15360/60000 (25%)]	Loss: 0.195391
Train Epoch: 2 [30720/60000 (51%)]	Loss: 0.154028
Train Epoch: 2 [46080/60000 (76%)]	Loss: 0.108425

Test set: Average loss: 0.1073, Accuracy: 9679/10000 (97%)


[Done] exited with code=0 in 26.494 seconds
```

Code:
SVM:

```
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn.datasets import fetch_openml
import numpy as np
import joblib

from scipy.io import loadmat

def load_mnist():
    '''
    output: x, y as numpy matrix, x and y type as np.uint8(0-255)
    '''
    mnist = fetch_openml("mnist_784", version=1,
                        as_frame=False, data_home='./datasets')
    x, y = fetch_openml("mnist_784", version=1,
                       return_X_y=True, as_frame=False)
    x, y = np.uint8(x), np.uint8(y)
    return x, y, mnist

# due to the bad network, use the local mnist-original.mat data
#mnist = fetch_openml("mnist_784",data_home='./datasets')
mnist = loadmat(
    'D:\\Git\\SJTU-AICourse-Proj1---MSA\\proj2\\MNIST_svm\\datasets\\mnist-
original.mat')
mnist.keys()
print(mnist)

x = mnist['data']
y = mnist['label']
x = np.swapaxes(x, 0, 1)
print(x.shape)
print(x)
y = np.squeeze(y)
print(y.shape)
print(y)

shape = 'ovr'

x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.08, train_size=0.4, random_state=4)
# parameters:
```

```
# X_train,X_test, y_train, y_test
=sklearn.model_selection.train_test_split(train_data,train_target,test_size=0.4,
random_state=0,stratify=y_train)
# train_data
# train_target
# test_size: float or int,default=None; if float, should be between 0.0 and 1.0 and
represent the proportion of the dataset to include in the test split.If int,
represents the absolute number of test samples. If None, the value is set to the
complement of the train size. If train_size is also None, it will be set to 0.25.
# train_size: float or int, default=None; If float, should be between 0.0 and 1.0
and represent the proportion of the dataset to include in the train split. If int,
represents the absolute number of train samples. If None, the value is
automatically set to the complement of the test size.
# random_state: int,RandomState instance or None,default=None; Controls the
shuffling applied to the data before applying the split. Pass an int for
reproducible output across multiple function calls.
# shuffle: bool,default=True; Whether or not to shuffle the data before splitting.
If shuffle=False then stratify must be None.
# stratify: array-like,default=None;
# return: splitting: list, length=2*len(arrays)

"""
svm.attribute:
    svm.LinearSVC
    svm.LinearSVR
    svm.NuSVC
    svm.VuSVR
    svm.OneClassSVM
    svm.SVC
    svm.SVR
    svm.l1_min_c
"""
predictor = svm.SVC(gamma='scale', C=1,
                   decision_function_shape=shape, kernel='sigmoid')
# parameters:
# C: float,default=1.0; must be strictly positive
# kernel: 'linear','poly','rbf','sigmoid','precomputed',default='rbf'
# degree: int,default=3; degree of the poly nomial kernel function('poly'), ignored
by other kernel
# gamma: {'scale','auto'} or float, default='scale'; kernel coefficient for
'rbf','poly' and 'sigmoid'
# coef0: float, default=0.0; independent term in kernel function, only significant
in 'poly' and 'sigmoid'
# shrinking: bool,default=True; whether to use the shrinking heuristic
# probability: bool,default=False; whether to enable probability estimates. this
must be enabled prior to calling fit, will slow down that method as it internally
uses 5-fold cross-validation, and predict_proba may be inconsistent with predict.
# tol: float, default=1e-3; tolerance for stopping criterion
# cache_size:fit,default=200; specify the size of the kernel cache(in MB)
# class_weight: dict or 'balanced',default=None; Set the parameter C of class i to
class_weight[i]*C for SVC. If not given, all classes are supposed to have weight
one. The “balanced” mode uses the values of y to automatically adjust weights
inversely proportional to class frequencies in the input data as n_samples /
(n_classes * np.bincount(y))
# verbose: bool,default=False; enable verbose output.
# max_iter: int,default=-1; hard limit on iterations within solver, or -1 for no
limit
# decision_function_shape: {'ovo','ovr'},default='ovr';
# ovo: one versus one, it will build k*(k-1)/2 classifiers for k kinds
# ovr: one versus rest, it will build k classifiers for k kinds
# break_ties: bool,default=False; If true, decision_function_shape='ovr', and
number of classes > 2, predict will break ties according to the confidence values
of decision_function; otherwise the first class among the tied classes is returned.
# random_state: int,RandomState instance or None,default=None; Controls the pseudo
random number generation for shuffling the data for probability estimates. Ignored
when probability is False. Pass an int for reproducible output across multiple
function calls.

predictor.fit(x_train, y_train)
# fit(X,y,sample_weight=None): fit the SVM model according to the given training
data
# parameters:
# X:train data
# y:train data label
# sample_weight:weight of the training data
# return: self:object; fitted estimator

joblib.dump(predictor, 'svm.pkl')
# save trained model to local file .pkl

predictor = joblib.load('./svm.pkl')
# load .pkl model from local file

result = predictor.predict(x_test)
# predict(X): perform classification on samples in X; for an one-class model,+1 or
-1 is returned
# parameters:
```

```
# X:{array-like, sparse matrix} of shape (n_samples, n_features) or
(n_samples_test, n_samples_train)
# return: y_pred:ndarray of shape (n_samples,); Class labels for samples in X

print(result)
print(y_test)
print(predictor.score(x_test, y_test))

# for report data
'''
for temp in range(1, 11):
    print(temp)
    shape = 'ovo'

    x_train, x_test, y_train, y_test = train_test_split(
        x, y, test_size=0.08, train_size=0.4, random_state=4)

    predictor = svm.SVC(gamma='scale', C=0.3*temp,
                        decision_function_shape=shape, kernel='rbf')

    predictor.fit(x_train, y_train)
    joblib.dump(predictor, 'svm.pkl')
    predictor = joblib.load('./svm.pkl')
    result = predictor.predict(x_test)

    print(predictor.score(x_test, y_test))
'''
```

ANN:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision.datasets import MNIST
from torchvision.transforms import Compose, ToTensor, Normalize
from torch.utils.data import DataLoader

BATCH_SIZE = 512
# 一次输入量
EPOCHS = 2
# 总共训练批次
gpu_abailable = torch.cuda.is_available()
print(gpu_abailable)
```

```python
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# 判断是否使用GPU

# 拿数据
def get_dataloader(train, batch_size=BATCH_SIZE):
    transform_fn = Compose([
        ToTensor(),
        Normalize(mean=(0.1307,), std=(0.3081,))
    ])  # mean 和std 的形状与通道数相同

    dataset = MNIST(download=True, root='./data',
                    train=train, transform=transform_fn)
    data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
    return data_loader

train_loader = get_dataloader(train=True)
test_loader = get_dataloader(train=False)

# Create ANN Model
class ANNModel(nn.Module):

    def __init__(self):
        super(ANNModel, self).__init__()

        # Linear function 1: 784 --> 500
        self.fc1 = nn.Linear(1*28*28, 500)

        # Linear function 2: 500 --> 150
        self.fc2 = nn.Linear(500, 150)

        # Linear function 3: 150 --> 10
        self.fc3 = nn.Linear(150, 10)

    def forward(self, input):

        # reshape input
        x = input.view([-1, 1*28*28])

        # Linear function 1
        x = self.fc1(x)

        # Non-linearity 1
        x = F.relu(x)
        # Linear function 2
        x = self.fc2(x)

        # Non-linearity 2
        x = F.relu(x)

        # Linear function 3
        out = self.fc3(x)

        out = F.log_softmax(out, dim=1)

        return out

# 连接到GPU
model = ANNModel().to(DEVICE)
# 创建优化器并传入参数
#learning_rate = 0.02
#optimizer = optim.SGD(model.parameters(), lr=learning_rate)
optimizer = optim.Adam(model.parameters())

def train(model, device, train_loader, optimizer, epoch):
    # 将模型转为train 模式
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        # 重置优化器梯度
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        # backward 传播
        loss.backward()
        # 更新参数
        optimizer.step()
        if((batch_idx) % 30) == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))

def test(model, device, test_loader):
    # 将模型转为test 模式
    model.eval()
    # loss
    test_loss = 0
    # 正确数
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()
            pred = output.max(1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))

# 进行两次训练，第二次训练取第一次训练得出的hyperparameters
for epoch in range(1, EPOCHS+1):
    train(model, DEVICE, train_loader, optimizer, epoch)
    test(model, DEVICE, test_loader)
```