Image Classification

LIAO,WEN 15307110200

1 Task Description

Use the k-NN and decision tree for classification based on MNIST dataset available at http://yann.lecun.com/exdb/mnist/.
Based on the MNIST dataset, design and implement classifiers including: k-NN, decision tree, least squares with regularization, Fisher discriminant analysis (with kernels), Perceptron (with kernels), logistic regression, SVM (with kernels), MLP-NN with two different error functions.
Based on the MNIST dataset, design and implement a proper convolutional neural network. Optional task.

2 Data Acquisition

MNIST is a well-known dataset for pattern recognition. It consists of images of handwritten digits and the corresponding labels. MNIST dataset is available at http://yann.lecun.com/exdb/mnist/.

```
train-images-idx3-ubyte.gz:  training set images (9912422 bytes)
train-labels-idx1-ubyte.gz:  training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz:   test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz:   test set labels (4542 bytes)
```

The famed deep learning framework PyTorch provides built-in dataset MNIST for image classification problems. The data is split into a training dataset and a test dataset, which consists of 60000 and 10000 images with 28*28 pixels respectively. The corresponding output is an integer ranging from 0 to 9 inclusively. The machine learning toolkit scikit-learn also provides built-in MNIST dataset.

```
import torch, torchvision
from torch import nn,optim
from torch.nn import functional as F
from torch.utils.data import dataloader as Data

train_data = torchvision.datasets.MNIST(
    './mnist', train=True, transform=torchvision.transforms.ToTensor(), download=True
)
test_data = torchvision.datasets.MNIST(
    './mnist', train=False, transform=torchvision.transforms.ToTensor()
)
print("train_data:", train_data.train_data.size())
print("train_labels:", train_data.train_labels.size())
print("test_data:", test_data.test_data.size())

train_loader = Data.DataLoader(dataset=train_data, batch_size=64, shuffle=True)
test_loader = Data.DataLoader(dataset=test_data, batch_size=64)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Processing...
Done!
train_data: torch.Size([60000, 28, 28])
train_labels: torch.Size([60000])
test_data: torch.Size([10000, 28, 28])
```

```
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
X, y = mnist['data'], mnist['target']
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
shuffle_train, shuffle_test = np.random.permutation(60000), np.random.permutation(10000)
X_train,y_train = X_train[shuffle_train],y_train[shuffle_train]
X_test,y_test = X_test[shuffle_test],y_test[shuffle_test]
```

3 Classification with Classifiers

Before we look more deeply into the problem, let's define the criterion by which the performance of the models are judged. For a binary classification problem, precision is defined as the ratio of true positive samples in all positive samples. And recall rate is defined as the ratio of true positive samples in true positive and false negative samples.

$$precision = \frac{TP}{TP + FP}$$

$$recall\ rate = \frac{TP}{TP + FN}$$

For a multiclass classification problem, we the precision and recall rate with respect to a given label is defined as that in the binary case but all samples with other labels are considered as negative samples. The macro precision and recall rate is the mean of the precision and recall rate of all labels respectively. In this classification problem, the macro recall rate is equal to overall accuracy.

$$macro\ precision = \frac{1}{n}\sum_{i=1}^{n} precision_i$$

$$macro\ recall\ rate = \frac{1}{n}\sum_{i=1}^{n} recall\ rate_i$$

```python
#pytorch
def evaluate(prediction):
    macro_recall_rate = sum(torch.tensor([1 if pred == label else 0 for pred,label in prediction]))
    counter = torch.zeros(10,2)
    for pred,label in prediction:
        counter[pred.item()][0] += 1
        if pred == label:
            counter[pred.item()][1] += 1
    macro_precision = 0
    for i in range(10):
        macro_precision += counter[i][1]*1.0/counter[i][0]
    macro_precision /= 10
    return macro_recall_rate, macro_precision.item()
#sklearn
def evaluate(y_pred,y):
    y_pred = np.array(y_pred,dtype = int)
    y = np.array(y,dtype = int)
    macro_recall_rate = np.sum(y_pred == y)/len(y)
    counter = np.zeros((2,10))
    for pred,label in zip(y_pred, y):
        counter[0][pred] += 1
        if pred == label:
            counter[1][pred] += 1
    precision = counter[1]*1.0/counter[0]
    macro_precision = np.mean(precision[~(precision is np.nan)])
    return macro_recall_rate,macro_precision
```

3.1 k-nearest Neighbor

For the k-NN classifier, the distances are defined in a Euclidian space. To calculate the distances between two images, we flatten the matrices to vectors and take the vectors as the feature. The implementation of k-NN classifier is shown as follows. We take Euclidian space as the metric space and set the hyperparameter k as 3. Then we use the training dataset to train the classifier.

```
import numpy as np

def HamiltonianDistance(x,y):
    return sum(abs(np.array(x)-np.array(y)))

def SquaredEuclidianDistance(x,y):
    return sum((np.array(x) - np.array(y))*(np.array(x) - np.array(y)))

def linear_scan(distances,k):
    nearest = sorted([(i,distances[i]) for i in range(k)],key = lambda x:x[1])
    for i in range(k,len(distances)):
        for j in range(k):
            if distances[i] <= nearest[j][1]:
                nearest = nearest[:j]+[(i,distances[i])]+nearest[j:k-1]
    return [val[0] for val in nearest]

def kNN_feature(x):
    return x.reshape((x.shape[0],x.shape[1]*x.shape[2]))

class kNNClassifier():
    def __init__(self):
        self.count = 0
    def train(self,x_vec,y_vec):
        self.images = x_vec
        self.labels = y_vec
    def classify(self,x,k=3,distance = SquaredEuclidianDistance):
        k_nearest = linear_scan([distance(val,x) for val in self.images],k)
        ballot_box = dict()
        for val in k_nearest:
            label = self.labels[val]
            if label in ballot_box:
                ballot_box[label] += 1
            else:
                ballot_box[label] = 1
        ret = 0
        count = 0
        for key in ballot_box:
            if ballot_box[key] > count:
                ret = key
        self.count += 1
        print(self.count)
        return ret

def accuracy(classifier,images,labels):
    result = np.array([classifier.classify(image) for image in images]) == np.array(labels)
    return sum(result)/len(result)
```

It's easy to train and evaluate the model.

```
classifier = kNNClassifier()
classifier.train(kNN_feature(train_images),train_labels)
print(accuracy(classifier,kNN_feature(test_images),test_labels))
```

For the k-NN classifier, the only hyperparameter is k. We can tabulate the accuracy given different k values.

K	1	2	3	4	5	6
Accuracy	0.9691	0.9627	0.9705	0.9682	0.9688	0.9677

3.2 Decision Tree

In preference to writing a decision tree classifier from scratch, we use scikit-learn, the Python toolkit for machine learning.

```
from fileloader import PictureLoader
from sklearn.tree import DecisionTreeClassifier as DTC
import numpy as np

def flatten(image,N=4,threshold=40):
    ret = np.zeros((N,N),dtype = int)
    for i in range(N):
        for j in range(N):
            for u in range(image.shape[0]//N):
                for v in range(image.shape[1]//N):
                    ret[i][j] += image[image.shape[0]//N*i+u][image.shape[0]//N*j+v]
    ret.resize(N*N)
    return ret*N*N//(image.shape[0]*image.shape[1])>threshold

def DecisionTree_feature(x,N,threshold):
    return np.array([flatten(v,N=N,threshold=threshold) for v in x])

def accuracy(classifier,features,labels):
    labels_pred = classifier.predict(features)
    return sum(np.array(labels_pred)==labels)/len(labels)

classifier = DTC()
Ns = [1,2,4,7,14,28]
thresholds = [10,20,30,40,50]
for N in Ns:
    for threshold in thresholds:
        classifier = classifier.fit(DecisionTree_feature(train_images,N = N,threshold = threshold)
                                    ,train_labels)
print(N,threshold,accuracy(classifier,DecisionTree_feature(test_images,N = n,threshold = Threshold
                                                           ),test_labels))
```

For the decision tree classifier, both N, which decides the size of the feature matrix and the threshold, which decides the extent to which two images can be considered different, are hyperparameters. We can also display the accuracies with regard to different parameters in a table.

threshold\N	1	2	4	7	14	28
10	0.1135	0.2336	0.5711	0.8144	0.8913	0.8881
20	0.1709	0.3097	0.5962	0.8287	0.8951	0.8837
30	0.1971	0.3189	0.597	0.8354	0.8886	0.8899
40	0.1676	0.2669	0.5911	0.8402	0.8894	0.8871
50	0.1364	0.2115	0.5703	0.8469	0.8844	0.8811

From the tables we can see that the k-NN classifier performs the best when k equals 3 and the performance of the decision tree classifier reaches a maximum when N equals 14 and the threshold equals 40. It can also be seen that as the hyperparameters become larger, the performance of the classifier will first increase, then decrease.

We can also make a comparison between the k-NN and decision tree classifier with regard to the handwritten digits recognition problem. Basically, the k-NN classifier shows a better performance. But in terms of computational complexity, the decision tree classifier works more quickly in the classification process and do not take much time to train, even though the k-NN classifier takes no time to train.

3.3 Least Square with Regularization

The idea of multiclass classification with least square comes from the intuition that we can define a linear mapping function from a vector in the feature space to the probability distribution of the label. We use L2 regularization term.

This model reaches a macro recall rate of 86.03% and a macro precision of 86.23% on the MNIST dataset, which is a good baseline of a linear classifier.

```
import numpy as np
from numpy.linalg import pinv

class RegularizedLeastSquareClassifier():
    def __init__(self,dim):
        self.dim = dim

    def fit(self,X,y):
        X_ = np.concatenate((X,np.ones((len(X),1))),axis = 1)
        T = np.zeros((len(y),self.dim))
        for i in range(len(y)):
            T[i][int(y[i])] = 1
        self.W_ = np.dot(pinv(X_),T)

    def predict(self,X):
        X_ = np.concatenate((X,np.ones((len(X),1))),axis = 1)
        return np.argmax(np.dot(X_,self.W_),axis = 1)

clf = RegularizedLeastSquareClassifier(10)
clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
print(evaluate(y_pred,y_test))
```

3.4 Fisher Linear Discriminant Analysis with Kernels

Fisher linear discriminant learns a vector so that in the feature space, the projection of vectors with the same label clusters while that of the vectors with different labels scatters. Scikit-learn provides a LDA classifier so that we don't have to write that from scratch. This model reaches a macro recall rate of 87.30% and a macro precision of 87.35%

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
clf = LDA()
clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
print(evaluate(y_pred,y_test))
```

3.5 Perceptron with Kernels

Perceptron is a typical binary classifier. The perceptron algorithm makes an assumption that the samples are linear-separable in the feature space and seeks for a hyperplane that can split the samples with different labels.

To generalize this algorithm to multiclass cases, people put forward the one-versus-one and one-versus-the-rest strategies. In our classifier, we use the one-versus-the-rest strategy. The multiclass perceptron algorithm actually includes 9 perceptrons. We let the first perceptron to decide if the digit should be labeled as zero. If not, we turn to a second perceptron to decide if the digit should be labeled as one, and so on.

The perceptron algorithm has its dual form (i.e. the form with kernels). In the cases where the feature space is of really high dimension and the amount of samples is not that huge, the dual form is more computationally efficient. However, in this classification problem, the amount of samples are huge. It takes a long time and a lot of memory space(measured by **gigabytes**) to compute the Gram matrix. For that reason we only used a small subset from the MNIST dataset to show how this algorithm works.

This model reaches a recall rate of 36.30% and a precision of 66.66%, which is slightly higher than a random guess and indicates that the multiclass perceptron algorithm do not work that well on this classification problem.

```
class Perceptron():
    def __init__(self):
        pass

    def fit(self,X,y,lr = 0.1,it = 30):
        Gram = np.dot(X,X.T)
        alpha = np.zeros(len(y))
        self.b = 0.
        for epoch in range(it):
            for i in range(len(y)):
                if y[i]*(np.sum(alpha*y*Gram[i])+self.b) <= 0:
                    alpha[i] += lr
                    self.b += lr*y[i]
        self.w = np.dot(alpha*y,X)

    def predict(self,X):
        return np.array((np.dot(X,self.w)+self.b)>0, dtype = int)

class MulticlassPerceptron():
    def __init__(self):
        self.n = 2

    def fit(self,X,y,n = 2,lr = 0.1,it = 10):
        self.n = n
        self.clfs = []
        y = np.array(y,dtype = int)
        index = np.array(np.ones(len(y)),dtype=bool)
        for i in range(self.n-1):
            clf = Perceptron()
            y_ = np.sign((y == i)-0.1)
            clf.fit(X[index],y_[index])
            index = index & ~(y_ > 0)
            self.clfs.append(clf)
    def predict(self,X):
        result = np.ones(len(X),dtype = int)
        result *= -1
        for i in range(self.n-1):
            pred = self.clfs[i].predict(X)
            result[(result<0) & (pred>0)] = i
        result[result<0] = self.n-1
        return result
clf = MulticlassPerceptron()
clf.fit(X_train[:5000],y_train[:5000],n=10)
print(evaluate(clf.predict(X_test[:1000]),y_test[:1000]))
```

3.6 Logistic Regression

Logistic regression uses the Cross Entropy Loss as the loss function and Stochastic Gradient Descent as the learning algorithm.

```
In [ ]:
#Logistic Regression
class LogisticRegressor(nn.Module):
    def __init__(self,x_dim,y_dim):
        super(LogisticRegressor,self).__init__()
        self.linear = nn.Linear(x_dim,y_dim)
    def forward(self,x):
        return F.log_softmax(self.linear(x),dim = 1)

lr_model = LogisticRegressor(28*28,10)
loss_function = nn.NLLLoss()
optimizer = optim.SGD(lr_model.parameters(),lr = 0.1)
for epoch in range(10):
    print(epoch)
    for image,label in train_data:
        image = image.view(1,-1)
        lr_model.zero_grad()
        log_probs = lr_model(image)
        loss = loss_function(log_probs,torch.LongTensor([label]))
        loss.backward()
        optimizer.step()
    with torch.no_grad():
        lr_prediction = torch.tensor([(torch.argmax(lr_model(image.view(1,-1))),label) for image,label in test_data])
        recall_rate,precision = evaluate(lr_prediction)
        print("Recall rate:",recall_rate,"Precision:",precision)
```

From the result listed as follows we can see that the accuracy(recall rate) fluctuates between 86% and 88%.

Iteration	0	1	2	3	4	5	6	7	8	9
Recall Rate/%	87.23	86.00	87.00	88.02	87.19	87.45	86.46	87.85	87.32	86.16
Precision/%	87.60	87.19	87.58	88.27	87.88	88.03	87.66	88.54	88.28	87.82

3.7 Support Vector Machine

Support Vector Machine is also a famed classifier. The Python machine learning toolkit sklearn provides an SVM classifier. The performance of the model with different kernel function is listed as follows. Note that due to the limited computational capacity of the computer, only a portion of images and labels are used as the training set.

```
from sklearn.svm import SVC
clf = SVC(kernel = 'sigmoid')
clf.fit(X_train,y_train)
print(evaluate(y_pred,y_test))
```

Kernel	linear	poly	Poly	rbf	sigmoid
Dataset Size	20'000	20'000	60'000	20000	20000
Recall Rate/%	91.47	96.45	97.87	11.35	12.10
Precision/%	90.79	96.67	97.70	nan	Nan

3.8 Multilayer Perceptron Neural Network

Neural networks are gaining more and more popularity in recent years in both research and industry. This model enjoys high expressivity for its capacity to approximate any function with a given accuracy.

We use the sigmoid function as the activation function and Stochastic Gradient Descent as the optimization algorithm. The Cross Entropy Loss and Mean Square Loss are considered as the loss function.

```
In [ ]:
#MultiLayer Perceptron with Sigmoid Activation Function
class MultilayerPerceptronWithSigmoid(nn.Module):
    def __init__(self,x_dim,z_dim,y_dim):
        super(MultilayerPerceptronWithSigmoid,self).__init__()
        self.x2z = nn.Linear(x_dim,z_dim)
        self.z2y = nn.Linear(z_dim,y_dim)
    def forward(self,x):
        return F.log_softmax(self.z2y(F.sigmoid(self.x2z(x))),dim = 1)

mlps_model = MultilayerPerceptronWithSigmoid(28*28,1000,10)
loss_function = nn.NLLLoss()
optimizer = optim.SGD(mlps_model.parameters(),lr = 0.1)
for epoch in range(10):
    i = 0
    for image,label in train_data:
        i += 1
        if i%10000 == 0:
            print(i)
        image = image.view(1,-1)
        mlps_model.zero_grad()
        log_probs = mlps_model(image)
        loss = loss_function(log_probs,torch.LongTensor([label]))
        loss.backward()
        optimizer.step()
        if i%10000 == 0:
            with torch.no_grad():
                mlps_prediction = torch.tensor([(torch.argmax(mlps_model(image.view(1,-1))),label) for image,label in test_da
                recall_rate,precision = evaluate(mlps_prediction)
                print("Recall rate:",recall_rate,"Precision:",precision)
```

The sigmoid function performs well as the activation function. The result listed as follows show that the model with the cross entropy loss function reaches a recall rate of more than 98%.

Iteration	0	1	2	3	4	5	6	7	8	9
Recall Rate/%	95.60	96.42	96.78	97.26	97.23	97.53	97.80	97.93	97.75	98.06
Precision/%	95.67	96.43	96.78	97.24	97.21	97.50	97.78	97.90	97.74	98.04

The model with the mean square loss function as the activation function reaches a recall rate of more than 96.64%.

Iteration	0	1	2	3	4	5	6	7	8	9
Recall Rate/%	90.32	91.59	91.87	92.24	93.08	93.71	94.24	94.72	95.10	95.41
Precision/%	90.57	91.68	92.03	92.41	93.17	93.77	94.29	94.75	95.12	95.42

Iteration	10	11	12	13	14					
Recall Rate/%	95.72	96.02	96.31	96.47	96.64					
Precision/%	95.73	96.03	96.33	96.48	96.65					

3.9 Convolutional Neural Network

3.9.1 CNN with 2 CONV Layers and 3 FC Layers

First, we design a simple convolutional neural network with two convolutional layer and three fully connected layer.

Layer	Size	
Input Layer	32*32*1	
CONV Layer 1	Kernel	5*5*1*6
	Output	32*32*6
Pooling Layer 1	Kernel	2*2 Max Pooling
	Output	16*16*6
Conv Layer 2	Kernel	5*5*6*16
	Output	16*16*16
Pooling Layer 2	Kernel	2*2 Max Pooling
	Output	8*8*16

FC Layer 1	150
FC Layer 2	80
FC Layer 3	10

```python
#CNN
class CNN(nn.Module):
    def __init__(self):
        super(CNN,self).__init__()
        self.conv1 = nn.Conv2d(1,6,5,padding = 2)
        self.pool = nn.MaxPool2d(2,2)
        self.conv2 = nn.Conv2d(6,16,5,padding = 2)
        self.x2z = nn.Linear(16*7*7,150)
        self.z2w = nn.Linear(150,80)
        self.w2y = nn.Linear(80,10)
    def forward(self,x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1,16*7*7)
        x = torch.relu(self.x2z(x))
        x = torch.relu(self.z2w(x))
        x = self.w2y(x)
        return x
```

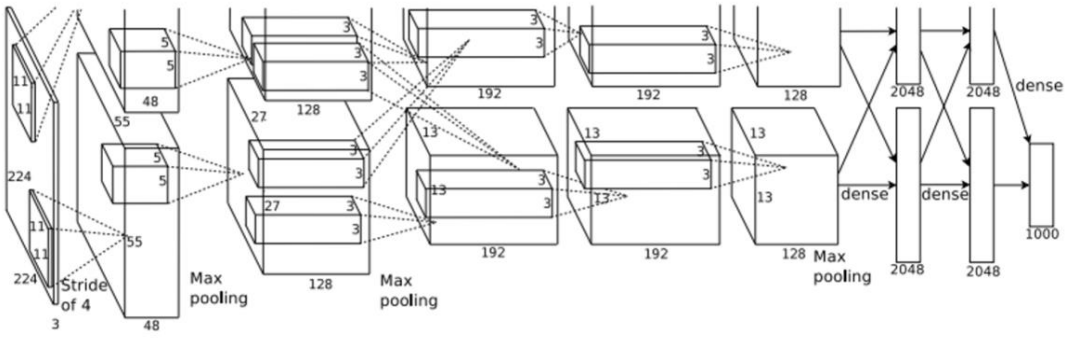
```python
cnn_model = CNN().cuda()
cnn_loss_function = nn.CrossEntropyLoss().cuda()
cnn_optimizer = optim.SGD(cnn_model.parameters(),lr = 0.001,momentum = 0.9)
for epoch in range(2):
    print(epoch)
    for image,label in train_data:
        cnn_model.zero_grad()
        log_probs = cnn_model(image)
        cnn_loss = cnn_loss_function(log_probs,label)
        cnn_loss.backward()
        cnn_optimizer.step()
    accuracy = 0
    for image,label in test_data:
        accuracy += torch.sum(torch.argmax(cnn_model(image),dim = 1) == label)
    print("Accuracy:",torch.tensor(accuracy,dtype = torch.float) /(4*len(test_data)))
```

This model reaches an accuracy of about 99% percent, which outperforms any other model in this assignment.

Epoch	1	2	3	4	5	6	7	8	9	10
Accuracy/%	97.58	98.51	98.79	98.95	99.11	98.82	98.72	99.00	98.59	98.83

3.9.2 AlexNet

AlexNet is the first modern CNN model for image classification, whose architecture is shown as follows.



It's uneasy to train a AlexNet from scratch, therefore we consider applying the transfer learning method. Pytorch provides well-trained models of AlexNet. We keep the architecture of the COVN layer unchanged as the feature extractor and change the parameters of the FC layers. Besides, the size of the input image is incompatible to the model, we have to use a transformer so that the size will fit.
The performance of the model is shown as follows. Further improvements still remains to be made.

Epoche	1	2	3	4	5
Recall Rate/%	69.40	74.23	72.74	77.12	75.20
Precision/%	78.91	78.70	79.33	80.49	80.38

4 Summary

No let's summarize the performances of different classifiers on MNIST dataset.

Classifier	Recall Rate(Accuracy)/%	Precision/%
k-NN	97.05	-
Decision tree	88.99	-
LSQ with regularization	86.03	86.23
LDA	87.30	87.35
Perceptron with kernels (one-versus-the-rest)	36.30	66.66
Logistic regression	87.07	87.89
SVM with polynomial kernels	97.87	97.70
MLP	98.06	98.04
CNN with 2 CONV and 3 FC layers	98.81	-
AlexNet with transfer learning	75.02	80.07