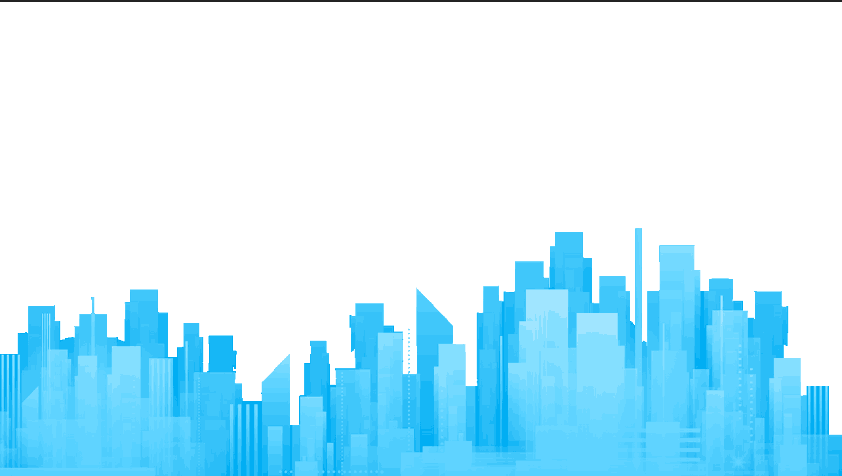
**MNIST手写数字识别任务及进阶**

**任务手册**

V1.0



深圳市腾讯计算机系统有限公司

目录

[文件介绍 2](#_Toc73354241)

[准备工作 2](#_Toc73354242)

[安装Anaconda 2](#_Toc73354243)

[创建虚拟环境 4](#_Toc73354244)

[MNIST手写数字识别实验 5](#_Toc73354245)

[进入代码文件 5](#_Toc73354246)

[引入需要的python包 5](#_Toc73354247)

[加载数据 6](#_Toc73354248)

[搭建神经网络模型CNN 8](#_Toc73354249)

[训练准备 9](#_Toc73354250)

[进行训练 10](#_Toc73354251)

[保存模型 11](#_Toc73354252)

[模型输出 12](#_Toc73354253)

[进阶：MNIST手写数字角度回归实验 14](#_Toc73354254)

[数据加载 14](#_Toc73354255)

[模型差异 15](#_Toc73354256)

[模型训练 17](#_Toc73354257)

[模型训练 19](#_Toc73354258)

[开拓思考 20](#_Toc73354259)

# 文件介绍

spark\_playground

|­­\_\_ data/mnist: 数据文件夹，其中存放了图片和对应的label文件

|\_\_ weights：模型文件夹，保存了两个训好的模型，供参考

|\_\_ MNIST手写数字识别.ipynb：实验代码1

|\_\_ 扩展实验-MNIST角度回归.ipynb：实验代码2

|\_\_ MNIST手写数字识别任务及进阶.docx：任务手册

|\_\_ requirements.txt：配准环境所需文件

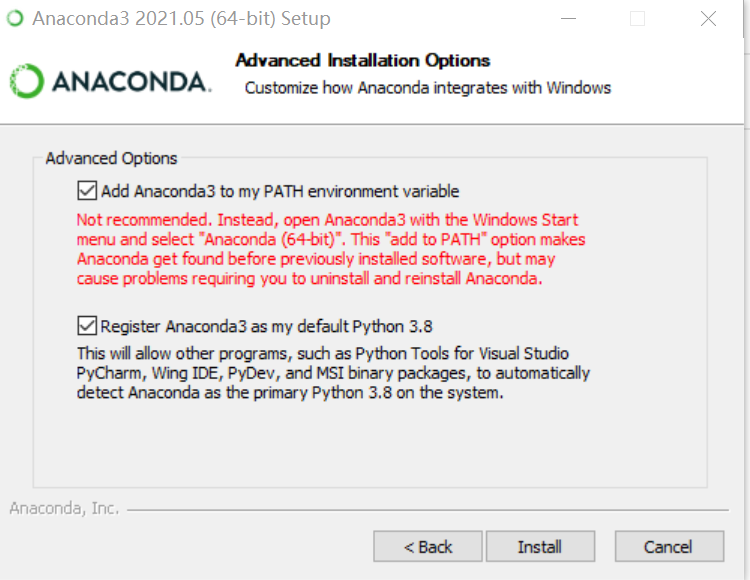
# 准备工作

## 安装Anaconda

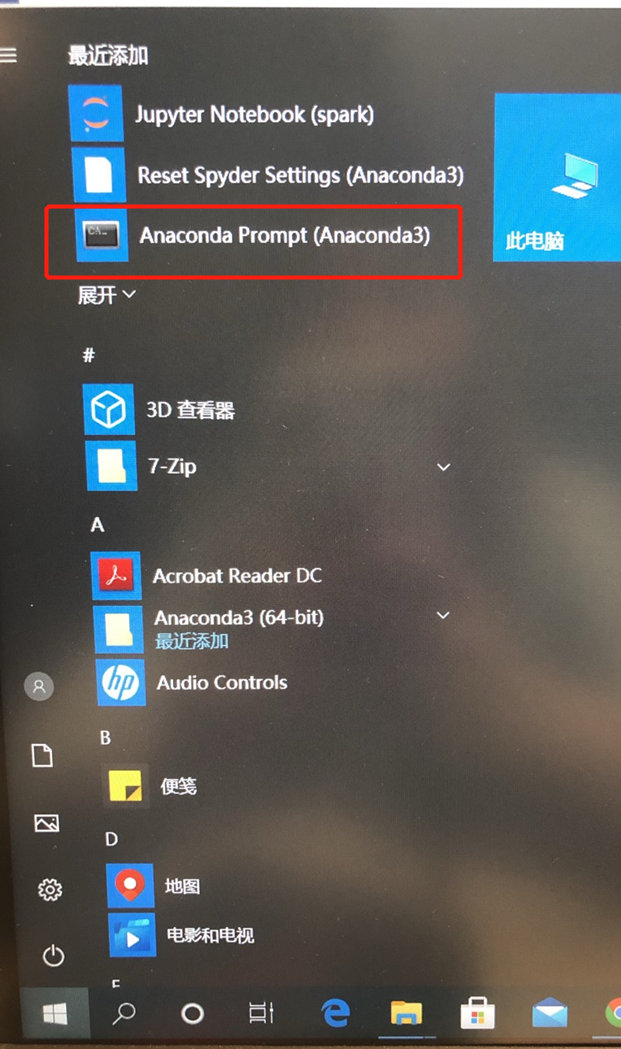
进入<https://www.anaconda.com/products/individual#Downloads> ，安装最新版Anaconda。

安装时请注意：

两个选项都要勾选，后续创建虚拟环境时可节省操作步骤。



Anaconda安装完成了，Windows可通过下述组件进入终端：



MAC系统直接打开终端（Terminal）即可。

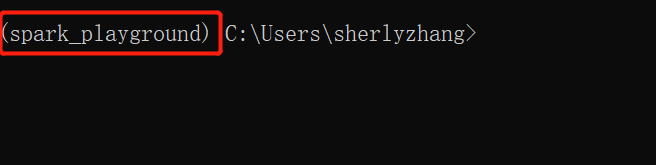
## 创建虚拟环境

使用如下指令创建虚拟环境：  
**conda create -n spark\_playground python=3.7**

创建完成后，输入下述指令进入该虚拟环境：

**conda activate spark\_playground**

若成功进入该虚拟环境，命令行前方会显示该虚拟环境的名字。



使用下面的指令安装jupyter notebook：

**conda install jupyter notebook**

安装完成后，进入到准备好的文件路径下，使用下述指令安装其他python包：

**cd /path\_to\_spark\_playground/spark\_playground/**

**pip install -r requirements.txt**

使用下述指令打开jupyter notebook，开始你们的练习：

**jupyter notebook**

# MNIST手写数字识别实验

## 进入代码文件

这是一个对手写数字进行分类的实验，数据集已经准备好，放置在data文件夹中。

打开《MNIST手写数字识别.ipynb》文件，按照指引开始学习。

## 引入需要的python包

from \_\_future\_\_ import print\_function

import numpy as np

import argparse

import os

import torch

import torchvision

import torch.nn as nn

import torch.nn.functional as F

import torch.optim as optim

from torchvision import datasets, transforms

from torch.utils.data import Dataset, DataLoader

import cv2

from PIL import Image

from sklearn.metrics import \*

import matplotlib.pyplot as plt

%matplotlib inline

## 加载数据

在这里我们使用的是本地的图片和标签，因此需要自行撰写一个从本地加载图片和标签的函数。

class LocalDataset(Dataset):

def \_\_init\_\_(self, root\_dir, record\_dir, transform):

super(LocalDataset, self).\_\_init\_\_()

self.transform = transform

self.root\_dir = root\_dir

self.datas = self.read\_samples\_from\_record(root\_dir, record\_dir)

def \_\_getitem\_\_(self, index):

path, target = self.datas[index]

sample = cv2.imread(path, cv2.IMREAD\_UNCHANGED)

sample = Image.fromarray(sample)

if self.transform is not None:

sample = self.transform(sample)

return sample, target

def \_\_len\_\_(self):

return len(self.datas)

def read\_samples\_from\_record(self, root\_dir, record\_dir):

samples = []

with open(record\_dir, "r") as f:

for index, line in enumerate(f):

line = line.split()

if len(line) < 2:

print('Error, Label is missing')

exit()

else:

image\_dir, label = line[0], line[1]

label = int(label)

image\_dir = os.path.join(root\_dir, image\_dir)

samples.append((image\_dir, label))

return samples

接着我们加载数据。

data\_dir = 'data/mnist/' #放置本地mnist数据集的地址

train\_label = 'data/mnist/mnist\_classification\_train.txt'

test\_label = 'data/mnist/mnist\_classification\_test.txt'

transform = transforms.Compose(

[transforms.ToTensor(),

transforms.Normalize((0.1307,), (0.3081,))])

train\_set = LocalDataset(data\_dir, train\_label, transform)

train\_loader = torch.utils.data.DataLoader(

train\_set,

batch\_size=64,

shuffle=True)

test\_set = LocalDataset(data\_dir, test\_label, transform)

test\_loader = torch.utils.data.DataLoader(

test\_set,

batch\_size=64,

shuffle=True)

使用以下的函数和代码，我们可以看到训练集中的图片大概长什么样。

def imshow(img):

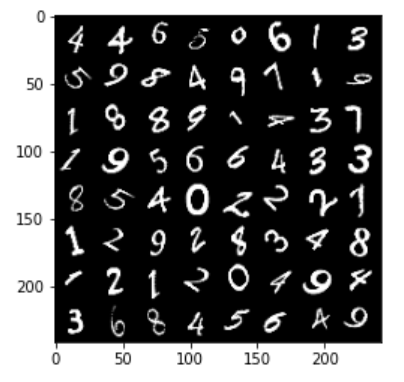
arr = img.numpy()

plt.imshow(np.transpose(arr, (1, 2, 0)))

data = iter(train\_loader)

images, labels = data.next()

imshow(torchvision.utils.make\_grid(images))



## 搭建神经网络模型CNN

下面的类就是我们的网络模型。

class CNN(nn.Module):

def \_\_init\_\_(self):

super(CNN, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(1, 20, 5, 1)

self.conv2 = nn.Conv2d(20, 50, 5, 1)

self.fc1 = nn.Linear(4\*4\*50, 500)

self.fc2 = nn.Linear(500, 10)

def forward(self, x):

x = F.relu(self.conv1(x))

x = F.max\_pool2d(x, 2, 2)

x = F.relu(self.conv2(x))

x = F.max\_pool2d(x, 2, 2)

x = x.view(-1, 4\*4\*50)

x = F.relu(self.fc1(x))

x = self.fc2(x)

return F.log\_softmax(x, dim=1)

使用这个网络结构创建我们的模型。

model = CNN()

## 训练准备

创建我们的训练函数，在每个epoch使用我们的图片和标签计算loss。

def train(model, device, train\_loader, optimizer, epoch):

losses = []

model.train()

for batch\_idx, (data, target) in enumerate(train\_loader):

data, target = data.to(device), target.to(device)

optimizer.zero\_grad()

output = model(data)

loss = F.nll\_loss(output, target)

loss.backward()

optimizer.step()

losses.append(loss.item())

if batch\_idx > 0 and batch\_idx % 50 == 0:

print('Train Epoch: {} [{}/{}\t({:.0f}%)]\tLoss: {:.6f}'.format(

epoch, batch\_idx \* len(data), len(train\_loader.dataset),

100. \* batch\_idx / len(train\_loader), loss.item()))

return losses

创建我们的测试函数，每个epoch训练结束后在我们的测试集上计算准确率。

def test(model, device, test\_loader):

model.eval()

test\_loss = 0

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() # sum up batch loss

pred = output.argmax(dim=1, keepdim=True) # get the index of the max log-probability

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)))

return (float(correct) / len(test\_loader.dataset))

设置我们的训练参数：

optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

#optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

device = torch.device("cpu") # or 'gpu'

losses = []

accuracies = []

epochs = 10

## 进行训练

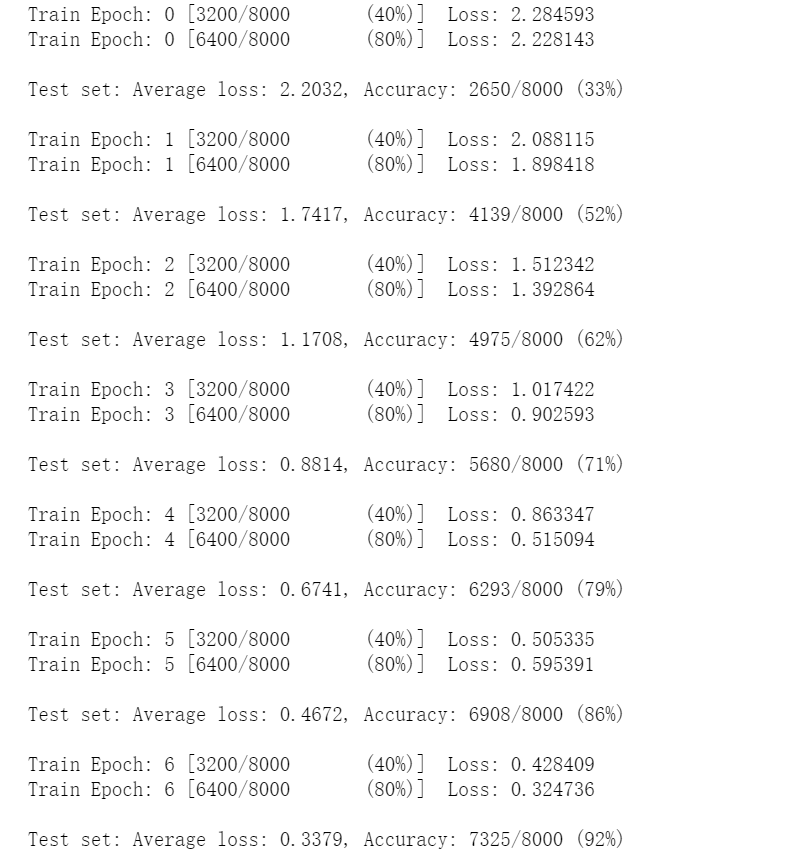
下面的代码将使用训练集训练我们的模型。

for epoch in range(epochs):

losses.extend(train(model, device, train\_loader, optimizer, epoch))

accuracies.append(test(model, device, train\_loader))

训练开始后，将打印出训练中的loss和每个epoch的准确率。



在这里我们可以看到，模型的loss一直在下降，而准确率不断地上升。

## 保存模型

使用下面的代码，我们就能将训练好的模型参数保存下来。

OUT\_PATH = 'weights/mnist-epoch'+str(epochs)+'.pth'

torch.save(model.state\_dict(), OUT\_PATH)

使用下面的函数，能够绘制出训练过程中的loss下降曲线。

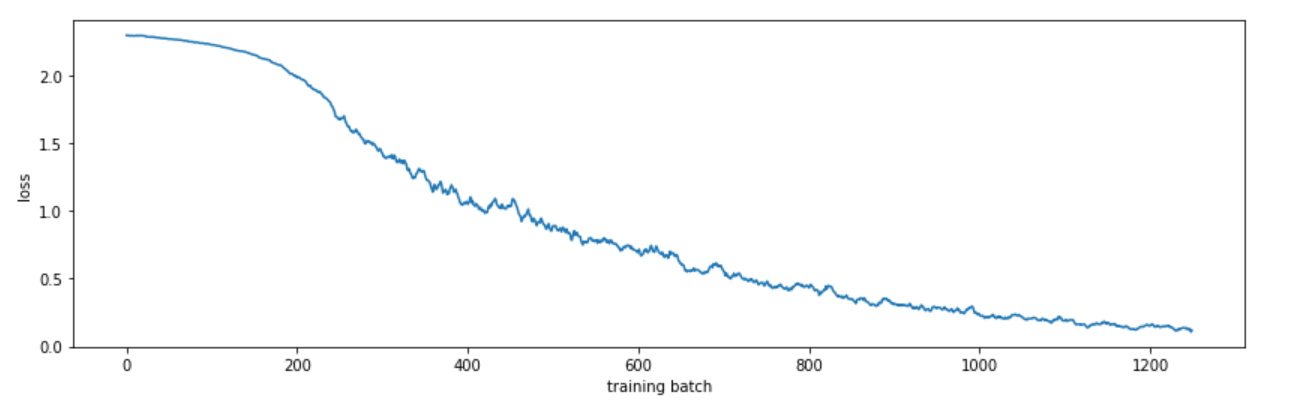
def mean(li): return sum(li)/len(li)

plt.figure(figsize=(14, 4))

plt.xlabel('training batch')

plt.ylabel('loss')

plt.plot([mean(losses[i:i+10]) for i in range(len(losses))])



下面的函数能够画出训练过程中的准确率上升曲线。

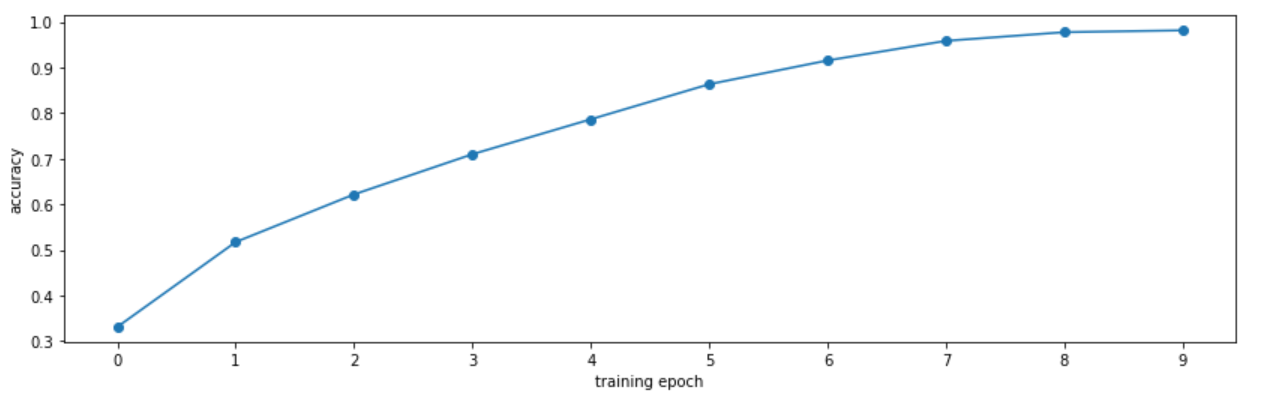
plt.figure(figsize=(14, 4))

plt.xticks(range(len(accuracies)))

plt.xlabel('training epoch')

plt.ylabel('accuracy')

plt.plot(accuracies, marker='o')



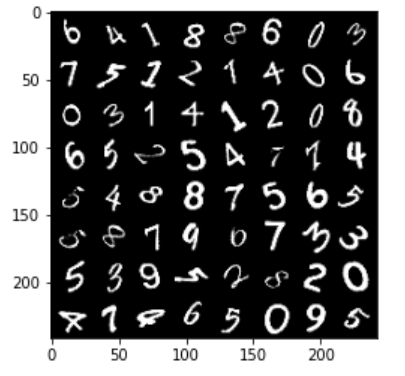
## 模型输出

取几张测试集上的图片，将它们显示出来。

data = iter(test\_loader)

images, labels = data.next()

imshow(torchvision.utils.make\_grid(images))



再将模型的对应输出打印出来，可以看到，模型输出的数字就是图中文字所代表的数字。

model.eval()

outputs = model(images)

\_, predicted = torch.max(outputs, 1)

print('Predicted: ')

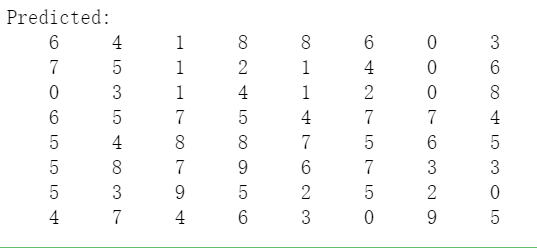
predicted = np.array(predicted)

for i in range(8):

cur\_out = predicted[i\*8:(i+1)\*8]

res = ' '.join('%5s' % cur\_out[j] for j in range(8))

print(res)



# 进阶：MNIST手写数字角度回归实验

在分类识别实验的基础上，我们将对模型进行微调，并训练一个角度回归网络。这个实验中包含不少内容与上一个实验有所重合，因此下面只介绍不同的部分。

打开《扩展实验-MNIST角度回归.ipynb》文件，按照指引开始学习。

## 数据加载

角度回归实验的标签与分类实验有所不同，因此加载函数、加载的标签文件也有所不同。

在这里，我们通过一个角度的余弦值和正弦值来唯一地表示这个角度，角度范围在-180~180度之间。

class LocalDataset(Dataset):

def \_\_init\_\_(self, root\_dir, record\_dir, transform):

super(LocalDataset, self).\_\_init\_\_()

self.transform = transform

self.root\_dir = root\_dir

self.datas = self.read\_samples\_from\_record(root\_dir, record\_dir)

def \_\_getitem\_\_(self, index):

path, target = self.datas[index]

sample = cv2.imread(path, cv2.IMREAD\_UNCHANGED)

sample = Image.fromarray(sample)

if self.transform is not None:

sample = self.transform(sample)

cos = math.cos(target/180\*math.pi)

sin = math.sin(target/180\*math.pi)

return sample, target, cos, sin

def \_\_len\_\_(self):

return len(self.datas)

def read\_samples\_from\_record(self, root\_dir, record\_dir):

samples = []

with open(record\_dir, "r") as f:

for index, line in enumerate(f):

line = line.split()

if len(line) < 2:

print('Error, Label is missing')

exit()

else:

image\_dir, label = line[0], line[1]

label = int(label)

image\_dir = os.path.join(root\_dir, image\_dir)

samples.append((image\_dir, label))

return samples

data\_dir = 'data/mnist/' #放置本地mnist数据集的地址

train\_label = 'data/mnist/mnist\_rotation\_train.txt'

test\_label = 'data/mnist/mnist\_rotation\_test.txt'

## 模型差异

分类网络最后通过一个全连接层和一个softmax层来输出不同类别的概率分数，回归网络最后一层直接输出图片的余弦值和正弦值，因此在网络结构上有所差异。

class CNN2(nn.Module):

def \_\_init\_\_(self):

super(CNN2, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(1, 20, 5, 1)

self.conv2 = nn.Conv2d(20, 50, 5, 1)

self.fc1 = nn.Linear(4\*4\*50, 500)

self.fc2 = nn.Linear(500, 2) # 最后一个fc层的输出尺寸变了

def forward(self, x):

x = F.relu(self.conv1(x))

x = F.max\_pool2d(x, 2, 2)

x = F.relu(self.conv2(x))

x = F.max\_pool2d(x, 2, 2)

x = x.view(-1, 4\*4\*50)

x = F.relu(self.fc1(x))

x = self.fc2(x)

return x

可以通过以下的代码查看两个模型的差异。

old\_model = CNN()

print('---------------CNN----------------')

for i in old\_model.\_modules.items():

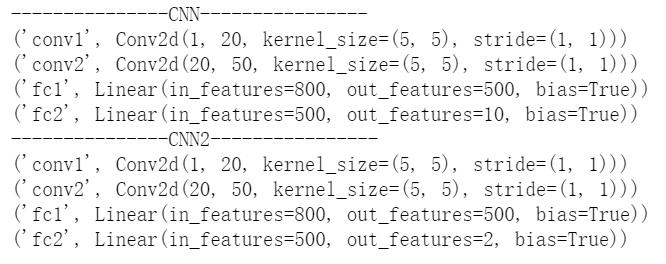
print(i)

print('---------------CNN2----------------')

model = CNN2()

for i in model.\_modules.items():

print(i)



由于网络的结构有差异，因此我们读取在分类任务中预训练好的模型之后，需要删除最后一个全连接层fc2的参数，再将其加载到我们的新模型中。

state\_dict = torch.load('weights/mnist-epoch10.pth')

state\_dict.pop('fc2.weight')

state\_dict.pop('fc2.bias')

model\_dict = model.state\_dict()

model\_dict.update(state\_dict)

model.load\_state\_dict(model\_dict)

对于我们的模型，我们将加载了预训练参数的层冻结，只训练最后一层的参数。

for name, value in model.named\_parameters():

if name != 'fc2.weight' and name != 'fc2.bias':

value.requires\_grad = False

params = filter(lambda p: p.requires\_grad, model.parameters())

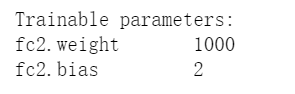
optimizer = torch.optim.Adam(params, lr=0.001)

print('Trainable parameters:')

for name, param in model.named\_parameters():

if param.requires\_grad:

print(name, '\t',param.numel())



## 模型训练

在回归任务的训练中，我们不再采用分类所使用NLL loss，而是换成了能够更好衡量回归三角函数值差异的MSE loss。

训练函数的代码如下，余弦值和正弦值的loss加在一起计算总loss：

def train(model, device, train\_loader, optimizer, epoch):

losses = []

model.train()

loss\_func = torch.nn.MSELoss() #更换分类用的nll loss为MSE loss

for batch\_idx, (data, target, target\_cos, target\_sin) in enumerate(train\_loader):

data, target, target\_cos, target\_sin = data.to(device), target.to(device).float(), target\_cos.to(device).float(), target\_sin.to(device).float()

optimizer.zero\_grad()

output = model(data)

output1 = output[:,0]

output2 = output[:,1]

#print(output1, target\_cos)

loss = loss\_func(output1, target\_cos) + loss\_func(output2, target\_sin)

loss.backward()

optimizer.step()

losses.append(loss.item())

if batch\_idx > 0 and batch\_idx % 25 == 0:

print('Train Epoch: {} [{}/{}\t({:.0f}%)]\tLoss: {:.6f}'.format(

epoch, batch\_idx \* len(data), len(train\_loader.dataset),

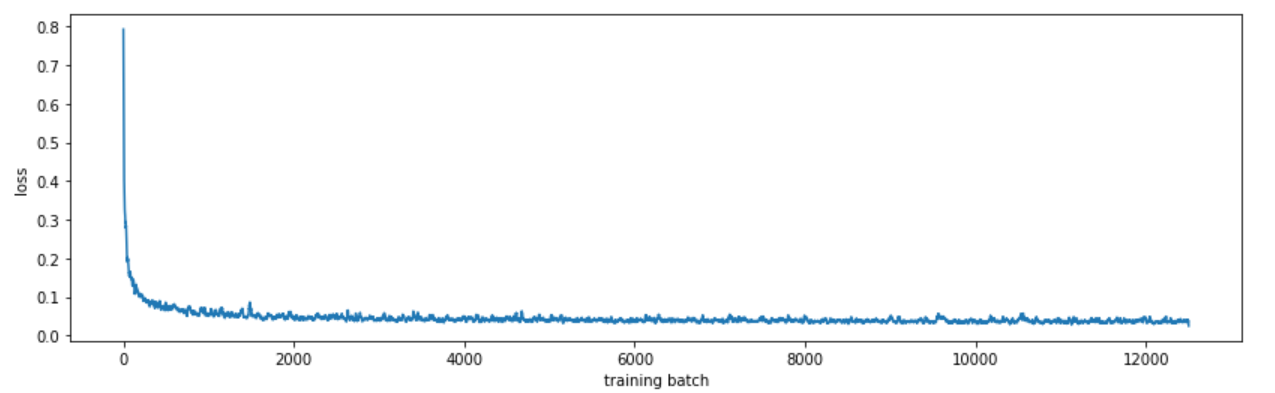
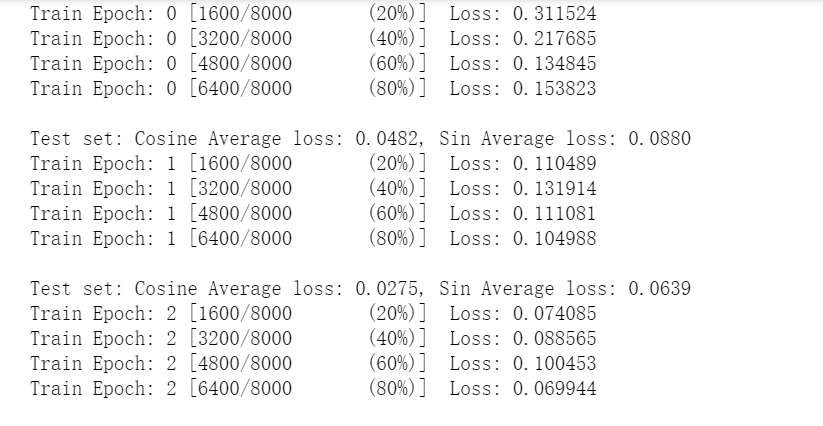
100. \* batch\_idx / len(train\_loader), loss.item()))

#print(model.state\_dict()['fc2.weight'])

#print(model.state\_dict()['fc2.bias'])

return losses

开始训练后，模型逐渐收敛。



## 模型测试

最终，我们获得模型输出的余弦值和正弦值，并将其转换为角度。

model.eval()

outputs = model(images)

predicted = []

for out in outputs:

if out[0]>=-1 and out[0]<=1:

angle = math.acos(out[0])/math.pi\*180

if out[1]<0:

angle = -angle

elif out[1]>=-1 and out[1]<=1:

angle = math.asin(out[1])/math.pi\*180

if out[0]<0:

angle = 180 - angle

else: angle = 0

predicted.append(angle)

print('Predicted: ')

predicted = np.array(predicted)

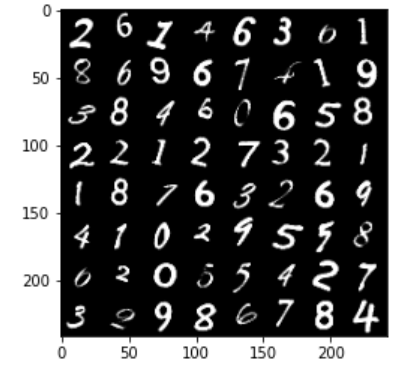
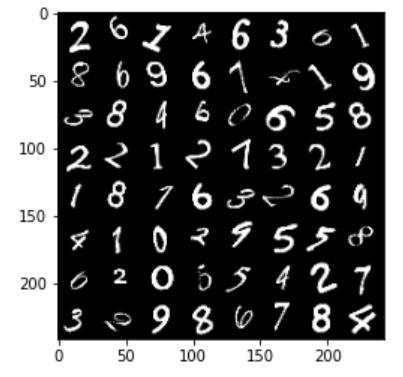
for i in range(8):

cur\_out = predicted[i\*8:(i+1)\*8]

res = ' '.join('%5s' % int(cur\_out[j]) for j in range(8))

print(res)

并将其应用在原图上。



原图 旋转校正后

可以看出，图中不少数字被正确地旋转了过来。

## 开拓思考

进阶练习中的回归模型，是否能够通过一些调整，让它输出的角度更准确，从而让图片旋转地更正呢？

提示：batch size、learning rate、数据扩增……