

Exercise2-LeNet5

学院：网络空间安全学院

专业：密码科学与技术

学号：2112155

姓名：梁婧涵

实验要求

在这个练习中，需要使用 Python 实现 LeNet5 来完成对 MNIST 数据集中 0-9 共 10 个手写数字的分类。代码只能使用 Python 实现，其中数据读取可使用 PIL、opencv-python 等库，矩阵运算可使用 numpy 等计算库，网络前向传播和梯度反向传播需要手动实现，不能使用 PyTorch、TensorFlow、Jax 或 Caffe 等自动微分框架。

MNIST 数据集可在 <http://yann.lecun.com/exdb/mnist/> 下载。

实验环境

pycharm+numpy

windows 11操作系统

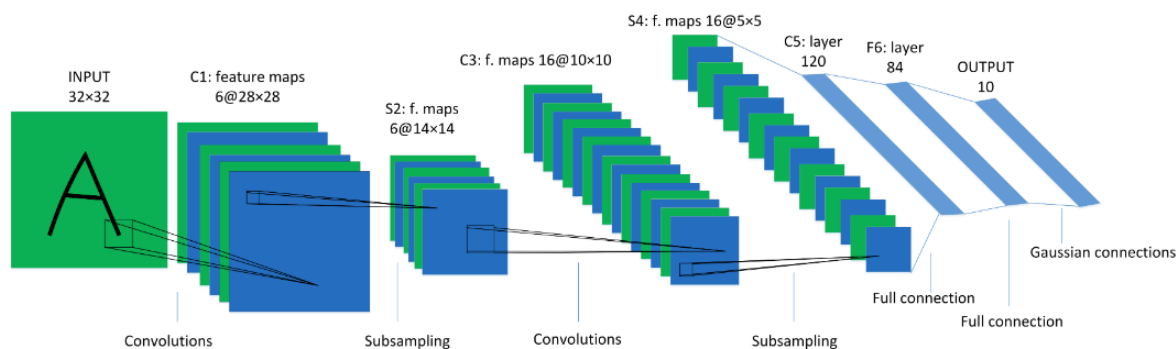
实验准备

Lenet5网络

1、模型介绍

LeNet-5出自论文《Gradient-Based Learning Applied to Document Recognition》，是由LeCun于1998年提出的一种用于识别手写数字和机器印刷字符的卷积神经网络，其命名来源于作者LeCun的名字，5则是其研究成果的代号，在LeNet-5之前还有LeNet-4和LeNet-1鲜为人知。LeNet-5阐述了图像中像素特征之间的相关性能够由参数共享的卷积操作所提取，同时使用卷积、下采样（池化）和非线性映射这样的组合结构，是当前流行的大多数深度图像识别网络的基础。

2、模型结构



LeNet-5虽然是早期提出的一个小网络，但是却包含了深度学习卷积神经网络的基本模块：卷积层、池化层和全连接层。LeNet-5一共包含7层（输入层不作为网络结构），分别由2个卷积层、2个池化层和3个连接层组成，网络的参数配置如表1所示，其中下采样层和全连接层的核尺寸分别代表采样范围和连接矩阵的尺寸

- 输入层

输入层是32x32像素的图像

- C1层——卷积层

手写数字数据集是灰度图像，输入为 $32 \times 32 \times 1$ 的图像，卷积核大小为 5×5 ，卷积核数量为6，步长为1，零填充。最终得到的C1的feature maps大小为 $(32 - 5 + 1 = 28)$ 。可训练参数： $(5 \times 5 + 1) \times 6$ ，其中有6个滤波器，每个滤波器 5×5 个units参数和一个bias参数，总共需要学习156个参数，这些参数是权值共享的。

- S2层——下采样层

卷积层C1之后接着就是池化运算，池化核大小为 2×2 ，LeNet-5池化运算的采样方式为4个输入相加，乘以一个可训练参数，再加上一个可训练偏置，结果通过sigmoid，所以下采样的参数个数是 $(1 + 1) \times 6$ 而不是零

- C3——卷积层

在LeNet-5中，C3中的可训练参数并未直接连接S2中所有的特征图（Feature Map），而是采用如图2所示的采样特征方式进行连接（稀疏连接）。具体地，C3的前6个feature map（对应图2第一个红框的前6列）与S2层相连的3个feature map相连接（图2第一个红框），后面6个feature map与S2层相连的4个feature map相连接（图2第二个红框），后面3个feature map与S2层部分不相连的4个feature map相连接，最后一个与S2层的所有feature map相连。卷积核大小依然为 5×5 ，所以总共有 $6 \times (3 \times 5 \times 5 + 1) + 6 \times (4 \times 5 \times 5 + 1) + 3 \times (4 \times 5 \times 5 + 1) + 1 \times (6 \times 5 \times 5 + 1) = 1516$ 参数。在原论文中解释了使用这种采样方式原因包含两点：限制了连接数不至于过大（当年的计算能力比较弱）；强制限定不同特征图的组合可以使映射得到的特征图学习到不同的特征模式

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X	X	X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

- S4——下采样层

与下采样层S2类似，采用大小为 2×2 ，步距为2的池化核对输入feature maps下采样，输出feature maps大小为 5×5

- C5——卷积层

与卷积层C3不同，卷积层C5的输入为S4的全部feature maps，由于S4层的16个图的大小为 5×5 ，与卷积核的大小相同，所以卷积后形成的图的大小为 1×1

- 全连接层F6和Output

F6和Output层在图1中显示为全连接层，原论文中解释这里实际采用的是卷积操作，只是刚好在 5×5 卷积后尺寸被压缩为 1×1 ，输出结果看起来和全连接很相似。

F6层共有84个神经元，与C5层进行全连接，即每个神经元都与C5层的120个特征图相连。计算输入向量和权重向量之间的点积，再加上一个偏置，结果通过sigmoid函数输出。

F6层共有84个节点，对应于一个 7×12 的比特图，-1表示白色，1表示黑色，这样每个符号的比特图的黑白色对应一个编码，该层的训练参数和连接数是 $(120+1) \times 84 = 10164$

Output层共有10个节点，分别代表0-9，假设x是上一层输入，y是RBF输出，则RBF输出的计算方式：

$$y_i = \sum_{j=0}^{83} (x_j - w_{ij})^2$$

3、模型特性

- 卷积网络使用一个3层的序列组合：卷积、下采样（池化）、非线性映射（LeNet-5最重要的特性，奠定了目前深层卷积网络的基础）
- 使用卷积提取空间特征
- 使用映射的空间均值进行下采样
- 使用tanh或sigmoid进行非线性映射
- 多层神经网络（MLP）作为最终的分类器
- 层间的稀疏连接矩阵以避免巨大的计算开销

代码展示

main

对数据读取和正则化，mnist数据集的图片为 28×28 ，需要padding到 32×32 。然后将处理之后的数据输入train函数开始训练，训练完成在测试集测试训练网络性能

```
# coding=utf-8
from data_processing import *
from train import *
from LeNet5 import *

if __name__ == '__main__':
    # get the data
    train_images, train_labels, test_images, test_labels = load_data()

    print("Got data...\n")

    # data processing, normalization&zero-padding
    print("Normalization and zero-padding...\n")
    train_images = normalize(zero_pad(train_images[:, :, :, np.newaxis], 2),
                              'LeNet5')
```

```

test_images = normalize(zero_pad(test_images[:, :, :, np.newaxis], 2),
'LeNet5')
print("The shape of training image with padding: ", train_images.shape)
print("The shape of testing image with padding: ", test_images.shape)
print("Finish data processing...\n")

# train LeNet5
LeNet5 = LeNet5()
print("Start training...")
start_time = time.time()
train(LeNet5, train_images, train_labels)
end_time = time.time()
print("Finished training, the total training time is {}s \n".format(end_time
- start_time))

# read model
# with open('model_data_13.pkl', 'rb') as input_:
#     LeNet5 = pickle.load(input_)

# evaluate on test dataset
print("Start testing...")
error01, class_pred = LeNet5.Forward_Propagation(test_images, test_labels,
'test')
print("error rate:", error01 / len(class_pred))
print("Finished testing, the accuracy is {} \n".format(1 - error01 /
len(class_pred)))

```

数据处理——data_processing

```

# coding=utf-8
import numpy as np
import struct
import os

data_dir = "E:\机器学习\ex2\mnist_data"
train_data_dir = "train-images.idx3-ubyte"
train_label_dir = "train-labels.idx1-ubyte"
test_data_dir = "t10k-images.idx3-ubyte"
test_label_dir = "t10k-labels.idx1-ubyte"

#函数用于从指定文件加载MNIST数据。
def load_mnist(file_dir, is_images='True'):
    # 读取二进制数据
    bin_file = open(file_dir, 'rb')
    bin_data = bin_file.read()
    bin_file.close()
    # 分析文件头部
    if is_images:
        # 读取图像数据
        fmt_header = '>iiii'
        magic, num_images, num_rows, num_cols = struct.unpack_from(fmt_header,
bin_data, 0)
        data_size = num_images * num_rows * num_cols

```

```

        mat_data = struct.unpack_from('>' + str(data_size) + 'B', bin_data,
struct.calcsize(fmt_header))
        mat_data = np.reshape(mat_data, [num_images, num_rows, num_cols])
    else:
        # 读取标签数据
        fmt_header = '>ii'
        magic, num_images = struct.unpack_from(fmt_header, bin_data, 0)
        mat_data = struct.unpack_from('>' + str(num_images) + 'B', bin_data,
struct.calcsize(fmt_header))
        mat_data = np.reshape(mat_data, [num_images])
        print('Load images from %s, number: %d, data shape: %s' % (file_dir,
num_images, str(mat_data.shape)))
        return mat_data

```

调用 load_mnist 函数，加载训练集和测试集的图像和标签数据

```

def load_data():
    print('Loading MNIST data from files...')
    train_images = load_mnist(os.path.join(data_dir, train_data_dir), True)
    train_labels = load_mnist(os.path.join(data_dir, train_label_dir), False)
    test_images = load_mnist(os.path.join(data_dir, test_data_dir), True)
    test_labels = load_mnist(os.path.join(data_dir, test_label_dir), False)
    return train_images, train_labels, test_images, test_labels

```

将图像转换为二进制形式，并生成独热编码的标签。

```

def data_convert(x, y, m, k):
    x[x <= 40] = 0
    x[x > 40] = 1
    ont_hot_y = np.zeros((m, k))
    for t in range(m):
        ont_hot_y[t, y[t]] = 1
    return x, ont_hot_y

```

用于对图像矩阵进行零填充

```

def zero_pad(X, pad):
    X_pad = np.pad(X, ((0, 0), (pad, pad), (pad, pad), (0, 0)), 'constant',
constant_values=(0, 0))
    return X_pad

```

对输入图像进行归一化处理，根据不同的模式进行归一化。

```

def normalize(image, mode='LeNet5'):
    image -= image.min()
    image = image / image.max()
    if mode == 'Op1':
        return image # range = [0,1]
    elif mode == 'nlp1':
        image = image * 2 - 1 # range = [-1,1]
    elif mode == 'LeNet5':
        image = image * 1.275 - 0.1 # range = [-0.1,1.175]
    return image

```

LeNet5

定义了一个基于LeNet-5模型的类 `LeNet5`，包括了前向传播和反向传播的方法。该模型由卷积层（C1、C3、C5）、池化层（S2、S4）、全连接层（F6）和RBF层（Output）组成。层与层间的参数传递，通过SDLM算法进行计算得到每一轮的学习率

```
import numpy as np

from pooling_utils import *
from convolution_utils import *
from activation_utils import *
from RBF_init_weight import *

bitmap = rbf_init_weight()

class LeNet5(object):
    """
    C1 -> S2 -> C3 -> S4 -> C5 -> F6 -> Output

    Reference: https://www.cnblogs.com/fengff/p/10173071.html
    """

    def __init__(self):
        # 初始化各层的结构和参数，包括卷积层的核形状、池化层的超参数、全连接层的权重形状等。
        # 使用了一个RBF初始化权重的函数rbf_init_weight来初始化RBF层的权重。
        C3_mapping = [[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 0], [5,
0, 1],
                    [0, 1, 2, 3], [1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 0],
[4, 5, 0, 1],
                    [5, 0, 1, 2], [0, 1, 3, 4], [1, 2, 4, 5], [0, 2, 3, 5],
[0, 1, 2, 3, 4, 5]]

        kernal_shape = {"C1": (5, 5, 1, 6),
                        "C3": (5, 5, 6, 16),
                        "C5": (5, 5, 16, 120),
                        "F6": (120, 84),
                        "OUTPUT": (84, 10)}

        hyper_parameters_convolution = {"stride": 1, "pad": 0}
        hyper_parameters_pooling = {"stride": 2, "f": 2}

        self.C1 = ConvolutionLayer(kernal_shape["C1"],
hyper_parameters_convolution)
        self.a1 = Activation("LeNet5_squash")
        self.S2 = PoolingLayer(hyper_parameters_pooling, "average")

        self.C3 = ConvolutionLayer_maps(kernal_shape["C3"],
hyper_parameters_convolution, C3_mapping)
        self.a2 = Activation("LeNet5_squash")
        self.S4 = PoolingLayer(hyper_parameters_pooling, "average")

        self.C5 = ConvolutionLayer(kernal_shape["C5"],
hyper_parameters_convolution)
        self.a3 = Activation("LeNet5_squash")
```

```

self.F6 = FCLayer(kernal_shape["F6"])
self.a4 = Activation("LeNet5_squash")

self.Output = RBFLayer(bitmap)
# 前向传播方法 Forward_Propagation
# 对输入图像执行前向传播，依次经过卷积层C1、激活层a1、池化层S2、卷积层C3、激活层a2、池化层S4、
卷积层C5、激活层a3、全连接层F6、激活层a4和RBF层Output。
# 返回的结果根据模式 ('train'或'test') 不同，用于计算损失或测试精度
def Forward_Propagation(self, input_image, input_label, mode):
    self.label = input_label
    self.C1_FP = self.C1.forward_propagation(input_image)
    self.a1_FP = self.a1.forward_propagation(self.C1_FP)
    self.S2_FP = self.S2.forward_propagation(self.a1_FP)

    self.C3_FP = self.C3.forward_propagation(self.S2_FP)
    self.a2_FP = self.a2.forward_propagation(self.C3_FP)
    self.S4_FP = self.S4.forward_propagation(self.a2_FP)

    self.C5_FP = self.C5.forward_propagation(self.S4_FP)
    self.a3_FP = self.a3.forward_propagation(self.C5_FP)

    self.flatten = self.a3_FP[:, 0, 0, :]
    self.F6_FP = self.F6.forward_propagation(self.flatten)
    self.a4_FP = self.a4.forward_propagation(self.F6_FP)

    # output sum of the loss over mini-batch when mode = 'train'
    # output tuple of (0/1 error, class_predict) when mode = 'test'
    out = self.Output.forward_propagation(self.a4_FP, input_label, mode)

    return out
#反向传播方法 Back_Propagation:对模型进行反向传播，计算梯度，并更新各层的权重和偏置。
def Back_Propagation(self, momentum, weight_decay):
    dy_pred = self.Output.back_propagation()

    dy_pred = self.a4.back_propagation(dy_pred)
    F6_BP = self.F6.back_propagation(dy_pred, momentum, weight_decay)
    reverse_flatten = F6_BP[:, np.newaxis, np.newaxis, :]

    reverse_flatten = self.a3.back_propagation(reverse_flatten)
    C5_BP = self.C5.back_propagation(reverse_flatten, momentum,
weight_decay)

    S4_BP = self.S4.back_propagation(C5_BP)
    S4_BP = self.a2.back_propagation(S4_BP)
    C3_BP = self.C3.back_propagation(S4_BP, momentum, weight_decay)

    S2_BP = self.S2.back_propagation(C3_BP)
    S2_BP = self.a1.back_propagation(S2_BP)
    C1_BP = self.C1.back_propagation(S2_BP, momentum, weight_decay)

# 使用SDLM方法进行学习率调整，用于每个epoch开始前的学习率确定
def SDLM(self, mu, lr_global):
    d2y_pred = self.Output.SDLM()
    d2y_pred = self.a4.SDLM(d2y_pred)

```

```

F6_SDLM = self.F6.SDLM(d2y_pred, mu, lr_global)
reverse_flatten = F6_SDLM[:, np.newaxis, np.newaxis, :]

reverse_flatten = self.a3.SDLM(reverse_flatten)
C5_SDLM = self.C5.SDLM(reverse_flatten, mu, lr_global)

S4_SDLM = self.S4.SDLM(C5_SDLM)
S4_SDLM = self.a2.SDLM(S4_SDLM)
C3_SDLM = self.C3.SDLM(S4_SDLM, mu, lr_global)

S2_SDLM = self.S2.SDLM(C3_SDLM)
S2_SDLM = self.a1.SDLM(S2_SDLM)
C1_SDLM = self.C1.SDLM(S2_SDLM, mu, lr_global)

```

#卷积层类 `ConvolutionLayer` 和 `ConvolutionLayer_maps`:
#分别表示常规卷积层和带有映射关系的卷积层。
#包含前向传播、反向传播、SDLM等方法。

```

class ConvolutionLayer(object):
    def __init__(self, kernel_shape, parameters, init_mode='Gaussian_dist'):
        """
        :param kernel_shape: (n_f, n_f, n_C_prev, n_C)
        :param parameters: {"stride": s, "pad": p}
        :param init_mode:
        """
        self.parameters = parameters
        self.weight, self.bias = initialize(kernel_shape, init_mode)
        self.v_w, self.v_b = np.zeros(kernel_shape), np.zeros((1, 1, 1,
kernel_shape[-1]))

    def forward_propagation(self, input_map):
        output_map, self.cache = conv_forward(input_map, self.weight, self.bias,
self.parameters)
        return output_map

    def back_propagation(self, dz, momentum, weight_decay):
        dA_prev, dw, db = conv_backward(dz, self.cache)
        self.weight, self.bias, self.v_w, self.v_b = \
            update(self.weight, self.bias, dw, db, self.v_w, self.v_b, self.lr,
momentum, weight_decay)
        return dA_prev

    def SDLM(self, d2Z, mu, lr_global):
        d2A_prev, d2W = conv_SDLM(d2Z, self.cache)
        h = np.sum(d2W) / d2Z.shape[0]
        self.lr = lr_global / (mu + h)
        return d2A_prev

```

C3: convolution layer with assigned combination between input maps and weight

```

class ConvolutionLayer_maps(object):
    def __init__(self, kernel_shape, hyper_parameters, mapping,
init_mode='Gaussian_dist'):
        """
        kernel_shape: (n_f, n_f, n_C_prev, n_C)
        hyper_parameters = {"stride": s, "pad": p}

```



```

"""
self.hyper_parameters = hyper_parameters
self.mapping = mapping
self.wb = [] # list of [weight, bias]
self.v_wb = [] # list of [v_w, v_b]
for i in range(len(self.mapping)):
    weight_shape = (kernel_shape[0], kernel_shape[1],
len(self.mapping[i]), 1)
    w, b = initialize(weight_shape, init_mode)
    self.wb.append([w, b])
    self.v_wb.append([np.zeros(w.shape), np.zeros(b.shape)])

def forward_propagation(self, input_map):
    self.inputmap_shape = input_map.shape # (n_m,14,14,6)
    self.caches = []
    output_maps = []
    for i in range(len(self.mapping)):
        output_map, cache = conv_forward(input_map[:, :, :,
self.mapping[i]], self.wb[i][0], self.wb[i][1],
self.hyper_parameters)
        output_maps.append(output_map)
        self.caches.append(cache)
    output_maps = np.swapaxes(np.array(output_maps), 0, 4)[0]
    return output_maps

def back_propagation(self, dZ, momentum, weight_decay):
    dA_prevs = np.zeros(self.inputmap_shape)
    for i in range(len(self.mapping)):
        dA_prev, dw, db = conv_backward(dZ[:, :, :, i:i + 1],
self.caches[i])
        self.wb[i][0], self.wb[i][1], self.v_wb[i][0], self.v_wb[i][1] = \
            update(self.wb[i][0], self.wb[i][1], dw, db, self.v_wb[i][0],
self.v_wb[i][1], self.lr, momentum, weight_decay)
        dA_prevs[:, :, :, self.mapping[i]] += dA_prev
    return dA_prevs

# Stochastic Diagonal Levenberg-Marquadt

def SDLM(self, d2Z, mu, lr_global):
    h = 0
    d2A_prevs = np.zeros(self.inputmap_shape)
    for i in range(len(self.mapping)):
        d2A_prev, d2W = conv_SDLM(d2Z[:, :, :, i:i + 1], self.caches[i])
        d2A_prevs[:, :, :, self.mapping[i]] += d2A_prev
        h += np.sum(d2W)
    self.lr = lr_global / (mu + h / d2Z.shape[0])
    return d2A_prevs

#池化层类 PoolingLayer, 包含前向传播、反向传播、SDLM等方法
class PoolingLayer(object):
    def __init__(self, hyper_parameters, mode):
        self.hyper_parameters = hyper_parameters
        self.mode = mode

    def forward_propagation(self, input_map): # n,28,28,6 -> n,10,10,16

```

```

        A, self.cache = pool_forward(input_map, self.hyper_parameters,
self.mode)
        return A

    def back_propagation(self, dA):
        dA_prev = pool_backward(dA, self.cache, self.mode)
        return dA_prev

    def SDLM(self, d2A):
        d2A_prev = pool_backward(d2A, self.cache, self.mode)
        return d2A_prev

class Subsampling(object):
    def __init__(self, n_kernel, hyper_parameters):
        self.hyper_parameters = hyper_parameters
        self.weight = np.random.normal(0, 0.1, (1, 1, 1, n_kernel))
        self.bias = np.random.normal(0, 0.1, (1, 1, 1, n_kernel))
        self.v_w = np.zeros(self.weight.shape)
        self.v_b = np.zeros(self.bias.shape)

    def foward_prop(self, input_map): # n,28,28,6 / n,10,10,16
        A, self.cache = subsampling_forward(input_map, self.weight, self.bias,
self.hyper_parameters)
        return A

    def back_prop(self, dA, momentum, weight_decay):
        dA_prev, dw, db = subsampling_backward(dA, A_, weight, b, self.cache)
        self.weight, self.bias, self.v_w, self.v_b = \
            update(self.weight, self.bias, dw, db, self.v_w, self.v_b, self.lr,
momentum, weight_decay)
        return dA_prev

# Stochastic Diagonal Levenberg-Marquaedt
def SDLM(self, d2A, mu, lr_global):
    d2A_prev, d2w, _ = subsampling_backward(dA, A_, weight, b, self.cache)
    h = np.sum(d2w) / d2A.shape[0]
    self.lr = lr_global / (mu + h)
    return d2A_prev

#激活函数类 Activation, 包含前向传播、反向传播、SDLM等方法
class Activation(object):
    def __init__(self, mode):
        (act, d_act), actfName = activation_func()
        act_index = actfName.index(mode)
        self.act = act[act_index]
        self.d_act = d_act[act_index]

    def forward_propagation(self, input_image):
        self.input_image = input_image
        return self.act(input_image)

    def back_propagation(self, dz):
        dA = np.multiply(dz, self.d_act(self.input_image))
        return dA

```

```

# Stochastic Diagonal Levenberg-Marquardt
def SDLM(self, d2Z): # d2_LeNet5_squash
    dA = np.multiply(d2Z, np.power(self.d_act(self.input_image), 2))
    return dA

# 全连接层类 FCLayer
class FCLayer(object):
    def __init__(self, weight_shape, init_mode='Gaussian_dist'):
        self.v_w, self.v_b = np.zeros(weight_shape),
        np.zeros((weight_shape[-1],))
        self.weight, self.bias = initialize(weight_shape, init_mode)

    def forward_propagation(self, input_array):
        self.input_array = input_array # (n_m, 120)
        return np.matmul(self.input_array, self.weight) # (n_m, 84)

    def back_propagation(self, dZ, momentum, weight_decay):
        dA = np.matmul(dZ, self.weight.T) # (256, 84) * (84, 120) = (256, 120)
        (n_m, 84) * (84, 120) = (n_m, 120)
        dw = np.matmul(self.input_array.T, dZ) # (256, 120).T * (256, 84) =
        (256, 1, 120, 84) (n_m, 120).T * (n_m, 84) = (120, 84)
        db = np.sum(dZ.T, axis=1) # (84,)
        self.weight, self.bias, self.v_w, self.v_b = update(self.weight,
        self.bias, dw, db, self.v_w, self.v_b, self.lr, momentum, weight_decay)
        return dA

# Stochastic Diagonal Levenberg-Marquardt
def SDLM(self, d2Z, mu, lr_global):
    d2A = np.matmul(d2Z, np.power(self.weight.T, 2))
    d2W = np.matmul(np.power(self.input_array.T, 2), d2Z)
    h = np.sum(d2W) / d2Z.shape[0]
    self.lr = lr_global / (mu + h)
    return d2A

# RBF层类 RBFLayer 和 RBFLayer_trainable_weight: 分别表示常规RBF层和具有可训练权重的RBF
层。包含前向传播、反向传播、SDLM等方法。
class RBFLayer_trainable_weight(object):
    def __init__(self, weight_shape, init_weight=None,
    init_mode='Gaussian_dist'):
        self.weight_shape = weight_shape # (10, 84)
        self.v_w = np.zeros(weight_shape)

        if init_weight.shape == (10, 84):
            self.weight = init_weight
        else:
            self.weight, _ = initialize(weight_shape, init_mode)

    def forward_propagation(self, input_array, label, mode):
        """
        :param input_array: (n_m, 84)
        :param label: (n_m, )
        :param mode:
        :return:

```

```

"""
    if mode == 'train':
        self.input_array = input_array
        self.weight_label = self.weight[label, :] # (n_m, 84) labeled
version of weight
        loss = 0.5 * np.sum(np.power(input_array - self.weight_label, 2),
axis=1, keepdims=True) # (n_m, )
        return np.sum(np.squeeze(loss))

    if mode == 'test':
        subtract_weight = (
            input_array[:, np.newaxis, :] - np.array([self.weight] *
input_array.shape[0])) # (n_m,10,84)
        rbf_class = np.sum(np.power(subtract_weight, 2), axis=2) # (n_m,
10)

        class_pred = np.argmin(rbf_class, axis=1) # (n_m,)
        error01 = np.sum(label != class_pred)
        return error01, class_pred

def back_propagation(self, label, lr, momentum, weight_decay):
    # n_m = label.shape[0]

    # d_output = np.zeros((n_m, n_class))
    # d_output[range(n_m), label] = 1 # (n_m, 10) one-hot version of
gradient w.r.t. output

    dy_predict = -self.weight_label + self.input_array # (n_m, 84)

    dw_target = -dy_predict # (n_m, 84)

    dw = np.zeros(self.weight_shape) # (10,84)

    for i in range(len(label)):
        dw[label[i], :] += dw_target[i, :]

    self.v_w = momentum * self.v_w - weight_decay * lr * self.weight - lr *
dw
    self.weight += self.v_w

    return dy_predict

# RBF层类 RBFLayer 和 RBFLayer_trainable_weight:分别表示常规RBF层和具有可训练权重的RBF
层。包含前向传播、反向传播、SDLM等方法。
class RBFLayer(object):
    def __init__(self, weight):
        self.weight = weight # (10, 84)

    def forward_propagation(self, input_array, label, mode):
        """
        :param input_array: (n_m, 84)
        :param label: (n_m, )
        :param mode:
        :return:
        """
        if mode == 'train':

```

```

        self.input_array = input_array
        self.weight_label = self.weight[label, :] # (n_m, 84) labeled
version of weight
        loss = 0.5 * np.sum(np.power(input_array - self.weight_label, 2),
axis=1, keepdims=True) # (n_m, )
        return np.sum(np.squeeze(loss))
    if mode == 'test':
        # (n_m,1,84) - n_m*[(10,84)] = (n_m,10,84)
        subtract_weight = (
            input_array[:, np.newaxis, :] - np.array([self.weight] *
input_array.shape[0])) # (n_m,10,84)
        rbf_class = np.sum(np.power(subtract_weight, 2), axis=2) # (n_m,
10)

        class_pred = np.argmin(rbf_class, axis=1) # (n_m,)
        error01 = np.sum(label != class_pred)
        return error01, class_pred

    def back_propagation(self):
        dy_predict = -self.weight_label + self.input_array # (n_m, 84)
        return dy_predict

    def SDLM(self):
        # d2y_predict
        return np.ones(self.input_array.shape)

# 权重初始化函数 initialize:提供不同初始化方式（高斯分布、Fan-in）的权重初始化
def initialize(kernel_shape, mode='Fan-in'):
    bias_shape = (1, 1, 1, kernel_shape[-1]) if len(kernel_shape) == 4 else
(kernel_shape[-1],)
    if mode == 'Gaussian_dist':
        mu, sigma = 0, 0.1 # mu: mean value, sigma: standard deviation
        weight = np.random.normal(mu, sigma, kernel_shape)
        bias = np.ones(bias_shape) * 0.01
    elif mode == 'Fan-in': # original init. in the paper
        Fi = np.prod(kernel_shape) / kernel_shape[-1]
        weight = np.random.uniform(-2.4 / Fi, 2.4 / Fi, kernel_shape)
        bias = np.ones(bias_shape) * 0.01
    return weight, bias

# 权重更新函数 update:根据梯度和动量更新权重和偏置
def update(weight, bias, dw, db, vw, vb, lr, momentum=0, weight_decay=0):
    vw_u = momentum * vw - weight_decay * lr * weight - lr * dw
    vb_u = momentum * vb - weight_decay * lr * bias - lr * db
    weight_u = weight + vw_u
    bias_u = bias + vb_u
    return weight_u, bias_u, vw_u, vb_u

```

卷积层前向传播和后向传播——convolution_utils

```

import numpy as np
from data_processing import zero_pad

```

```
# Numpy version: compute with np.tensordot()
def conv_forward(A_prev, w, b, hyper_parameters):
    """
    输入:
    A_prev: 前一层输出激活值, 形状为 (m, n_H_prev, n_W_prev, n_C_prev)。
    w: 权重, 形状为 (f, f, n_C_prev, n_C)。
    b: 偏置, 形状为 (1, 1, 1, n_C)。
    hyper_parameters: 包含 "stride" 和 "pad" 的字典。
    输出:
    Z: 卷积输出, 形状为 (m, n_H, n_W, n_C)。
    cache: 包含 (A_prev, w, b, hyper_parameters) 的元组, 用于后续反向传播。
    描述:
    使用给定的输入和参数计算卷积层的前向传播。
    根据 hyper_parameters 中指定的填充方式使用零填充 (使用 zero_pad 函数)。
    遍历输出体积执行卷积操作, 并使用 np.tensordot 计算加权和。
    返回结果和后续反向传播所需的缓存。
    """
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
    (f, f, n_C_prev, n_C) = w.shape

    stride = hyper_parameters["stride"]
    pad = hyper_parameters["pad"]

    n_H = int((n_H_prev + 2 * pad - f) / stride + 1)
    n_W = int((n_W_prev + 2 * pad - f) / stride + 1)

    # Initialize the output volume Z with zeros.
    Z = np.zeros((m, n_H, n_W, n_C))
    A_prev_pad = zero_pad(A_prev, pad)
    for h in range(n_H): # loop over vertical axis of the output volume
        for w in range(n_W): # loop over horizontal axis of the output volume
            # Use the corners to define the (3D) slice of a_prev_pad.
            A_slice_prev = A_prev_pad[:, h * stride:h * stride + f, w * stride:w
            * stride + f, :]
            # Convolve the (3D) slice with the correct filter w and bias b, to
            get back one output neuron.
            Z[:, h, w, :] = np.tensordot(A_slice_prev, w, axes=([1, 2, 3], [0,
            1, 2])) + b

    assert (Z.shape == (m, n_H, n_W, n_C))
    cache = (A_prev, w, b, hyper_parameters)
    return Z, cache

# Numpy version: compute with np.dot
def conv_backward(dZ, cache):
    """
    输入:
    dZ: 相对于卷积层输出 (Z) 的损失梯度, 形状为 (m, n_H, n_W, n_C)。
    cache: 从前向传播步骤得到的缓存。
    输出:
    dA_prev: 相对于卷积层输入 (A_prev) 的损失梯度, 形状为 (m, n_H_prev, n_W_prev,
    n_C_prev)。
    dw: 相对于卷积层权重 (w) 的损失梯度, 形状为 (f, f, n_C_prev, n_C)。
    """

```

db: 相对于卷积层偏置 (**b**) 的损失梯度, 形状为 $(1, 1, 1, n_C)$ 。

描述:

计算卷积层的反向传播, 以获得输入、权重和偏置的梯度。

利用链式法则计算梯度。

遍历输出体积以使用卷积操作的转置计算梯度。

根据需要应用零填充 (使用 `zero_pad` 函数)。

返回计算得到的梯度。

```
"""
(A_prev, w, b, hyper_parameters) = cache
(m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
(f, f, n_C_prev, n_C) = w.shape
(m, n_H, n_W, n_C) = dz.shape
stride = hyper_parameters["stride"]
pad = hyper_parameters["pad"]

dA_prev = np.zeros((m, n_H_prev, n_W_prev, n_C_prev))
dw = np.zeros((f, f, n_C_prev, n_C))
db = np.zeros((1, 1, 1, n_C))

if pad != 0:
    A_prev_pad = zero_pad(A_prev, pad)
    dA_prev_pad = zero_pad(dA_prev, pad)
else:
    A_prev_pad = A_prev
    dA_prev_pad = dA_prev

for h in range(n_H): # loop over vertical axis of the output volume
    for w in range(n_W): # loop over horizontal axis of the output volume
        # Find the corners of the current "slice"
        vert_start, horiz_start = h * stride, w * stride
        vert_end, horiz_end = vert_start + f, horiz_start + f

        # Use the corners to define the slice from a_prev_pad
        A_slice = A_prev_pad[:, vert_start:vert_end, horiz_start:horiz_end, :]

        # Update gradients for the window and the filter's parameters
        dA_prev_pad[:, vert_start:vert_end, horiz_start:horiz_end, :] +=
np.transpose(np.dot(W, dz[:, h, w, :].T), (3, 0, 1, 2))

        dw += np.dot(np.transpose(A_slice, (1, 2, 3, 0)), dz[:, h, w, :])
        db += np.sum(dz[:, h, w, :], axis=0)

# Set dA_prev to the unpadded dA_prev_pad
dA_prev = dA_prev_pad if pad == 0 else dA_prev_pad[:, pad:-pad, pad:-pad, :]

# Making sure your output shape is correct
assert (dA_prev.shape == (m, n_H_prev, n_W_prev, n_C_prev))

return dA_prev, dw, db
"""
```

输入:

dz: 相对于卷积层输出 (**Z**) 的损失梯度, 形状为 (m, n_H, n_W, n_C) 。

cache: 从前向传播步骤得到的缓存。

输出:

dA_prev: 相对于卷积层输入 (**A_prev**) 的损失梯度, 形状为 (**m**, **n_H_prev**, **n_W_prev**, **n_C_prev**)。

dw: 相对于卷积层权重 (**w**) 的损失梯度, 形状为 (**f**, **f**, **n_C_prev**, **n_C**)。

描述:

计算相对于卷积层输入和权重的损失函数的二阶导数 (**Hessian**)。

类似于 **conv_backward**, 但通过在权重更新中使用元素-wise 平方 (**np.power**) 来整合二阶导数。

返回计算得到的二阶梯度。

"""

```
def conv_SDLM(dZ, cache):
    (A_prev, w, b, hparameters) = cache
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
    (f, f, n_C_prev, n_C) = w.shape
    stride = hparameters["stride"]
    pad = hparameters["pad"]
    (m, n_H, n_W, n_C) = dZ.shape
    dA_prev = np.zeros((m, n_H_prev, n_W_prev, n_C_prev))
    dw = np.zeros((f, f, n_C_prev, n_C))
    db = np.zeros((1, 1, 1, n_C))

    if pad != 0:
        A_prev_pad = zero_pad(A_prev, pad)
        dA_prev_pad = zero_pad(dA_prev, pad)
    else:
        A_prev_pad = A_prev
        dA_prev_pad = dA_prev

    for h in range(n_H): # loop over vertical axis of the output volume
        for w in range(n_W): # loop over horizontal axis of the output volume
            # Find the corners of the current "slice"
            vert_start, horiz_start = h * stride, w * stride
            vert_end, horiz_end = vert_start + f, horiz_start + f

            # Use the corners to define the slice from a_prev_pad
            A_slice = A_prev_pad[:, vert_start:vert_end, horiz_start:horiz_end, :]

            # Update gradients for the window and the filter's parameters
            dA_prev_pad[:, vert_start:vert_end, horiz_start:horiz_end, :] +=
np.transpose(
    np.dot(np.power(w, 2), dZ[:, h, w, :].T), (3, 0, 1, 2))

            dw += np.dot(np.transpose(np.power(A_slice, 2), (1, 2, 3, 0)), dZ[:,
h, w, :])
            # Set dA_prev to the unpadded dA_prev_pad
            dA_prev = dA_prev_pad if pad == 0 else dA_prev_pad[:, pad:-pad, pad:-pad, :]
            assert (dA_prev.shape == (m, n_H_prev, n_W_prev, n_C_prev))
            return dA_prev, dw
```

池化层前向传播和后向传播——pooling_utils

```
import numpy as np
```



```

def pool_forward(A_prev, hyper_parameters, mode):
    m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape
    f = hyper_parameters["f"]
    stride = hyper_parameters["stride"]

    n_H = int(1 + (n_H_prev - f) / stride)
    n_W = int(1 + (n_W_prev - f) / stride)
    n_C = n_C_prev

    A = np.zeros((m, n_H, n_W, n_C))
    for h in range(n_H): # loop on the vertical axis of the output volume
        for w in range(n_W): # loop on the horizontal axis of the output volume
            # Use the corners to define the current slice on the ith training
            # example of A_prev, channel c
            A_prev_slice = A_prev[:, h * stride:h * stride + f, w * stride:w *
            stride + f, :]
            # Compute the pooling operation on the slice. Use an if statement to
            # differentiate the modes.
            if mode == "max":
                A[:, h, w, :] = np.max(A_prev_slice, axis=(1, 2))
            elif mode == "average":
                A[:, h, w, :] = np.average(A_prev_slice, axis=(1, 2))

    cache = (A_prev, hyper_parameters)
    assert (A.shape == (m, n_H, n_W, n_C))
    return A, cache

def pool_backward(dA, cache, mode):
    """
    Implements the backward pass of the pooling layer

    :param dA: gradient of cost with respect to the output of the pooling layer,
    same shape as A
    :param cache: cache output from the forward pass of the pooling layer,
    contains the layer's input and hyper-parameters
    :param mode: the pooling mode you would like to use, defined as a string
    ("max" or "average")
    :return: dA_prev -- gradient of cost with respect to the input of the pooling
    layer, same shape as A_prev
    """
    A_prev, hyper_parameters = cache

    stride = hyper_parameters["stride"]
    f = hyper_parameters["f"]

    m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape # 256,28,28,6
    m, n_H, n_W, n_C = dA.shape # 256,14,14,6

    dA_prev = np.zeros((m, n_H_prev, n_W_prev, n_C_prev)) # 256,28,28,6

    for h in range(n_H): # loop on the vertical axis
        for w in range(n_W): # loop on the horizontal axis
            # Find the corners of the current "slice"
            vert_start, horiz_start = h * stride, w * stride

```

```

        vert_end, horiz_end = vert_start + f, horiz_start + f

        # Compute the backward propagation in both modes.
        if mode == "max":
            A_prev_slice = A_prev[:, vert_start: vert_end, horiz_start:
horiz_end, :]
            A_prev_slice = np.transpose(A_prev_slice, (1, 2, 3, 0))
            mask = A_prev_slice == A_prev_slice.max((0, 1))
            mask = np.transpose(mask, (3, 2, 0, 1))
            dA_prev[:, vert_start: vert_end, horiz_start: horiz_end, :] \
                += np.transpose(np.multiply(dA[:, h, w, :][:, :, np.newaxis,
np.newaxis], mask), (0, 2, 3, 1))

        elif mode == "average":
            da = dA[:, h, w, :][:, np.newaxis, np.newaxis, :] # 256*1*1*6
            dA_prev[:, vert_start: vert_end, horiz_start: horiz_end, :] +=
np.repeat(np.repeat(da, 2, axis=1), 2, axis=2) / f / f

    assert (dA_prev.shape == A_prev.shape)
    return dA_prev

def subsampling_forward(A_prev, weight, b, hparameters):
    A_, cache = pool_forward(A_prev, hparameters, 'average')
    A = A_ * weight + b
    cache_A = (cache, A_)
    return A, cache_A

def subsampling_backward(dA, weight, b, cache_A_):
    (cache, A_) = cache_A_
    db = dA
    dw = np.sum(np.multiply(dA, A_))
    dA_ = dA * weight
    dA_prev = pool_backward(dA_, cache, 'average')
    return dA_prev, dw, db

```

训练——train

```

# coding=utf-8
import sys
import time
import math
import pickle
import numpy as np
import matplotlib.pyplot as plt

# Number of epochs & learning rate in the original paper
epochs_original, lr_global_original = 16, np.array([5e-4] * 2 + [2e-4] * 3 +
[1e-4] * 3 + [5e-5] * 4 + [1e-5] * 8)
# Number of epochs & learning rate I used
epochs, lr_global_list = epochs_original, lr_global_original * 100

```

```

def train(LeNet5, train_images, train_labels):
    momentum = 0.9
    weight_decay = 0
    batch_size = 256

    # Training loops
    cost_last, count = np.Inf, 0
    err_rate_list = []
    for epoch in range(0, epochs):
        print("----- epoch{} begin -----
        -----".format(epoch + 1))

        # Stochastic Diagonal Levenberg-Marquardt method for determining the
        learning rate
        batch_image, batch_label = random_mini_batches(train_images,
train_labels, mini_batch_size=500, one_batch=True)
        LeNet5.Forward_Propagation(batch_image, batch_label, 'train')
        lr_global = lr_global_list[epoch]
        LeNet5.SDLM(0.02, lr_global)

        # print info
        print("global learning rate:", lr_global)
        print("learning rates in trainable layers:", np.array([LeNet5.C1.lr,
LeNet5.C3.lr, LeNet5.C5.lr, LeNet5.F6.lr]))
        print("batch size:", batch_size)
        print("momentum:", momentum, ", weight decay:", weight_decay)

        # loop over each batch
        ste = time.time()
        cost = 0
        mini_batches = random_mini_batches(train_images, train_labels,
batch_size)
        for i in range(len(mini_batches)):
            batch_image, batch_label = mini_batches[i]

            loss = LeNet5.Forward_Propagation(batch_image, batch_label, 'train')
            cost += loss

            LeNet5.Back_Propagation(momentum, weight_decay)

            # print progress
            if i % (int(len(mini_batches) / 100)) == 0:
                #sys.stdout.write("\033[F") # CURSOR_UP_ONE
                #sys.stdout.write("\033[K") # ERASE_LINE
                print("progress:", int(100 * (i + 1) / len(mini_batches)), "%,
", "cost =", cost, end='\r')
                sys.stdout.write("\033[F") # CURSOR_UP_ONE
                sys.stdout.write("\033[K") # ERASE_LINE

            print("Done, cost of epoch", epoch + 1, ":", cost, "
            ")

        error01_train, _ = LeNet5.Forward_Propagation(train_images,
train_labels, 'test')

```

```

err_rate_list.append(error01_train / 60000)
# error01_test, _ = LeNet5.Forward_Propagation(test_images, test_labels,
'test')
# err_rate_list.append([error01_train / 60000, error01_test / 10000])
print("0/1 error of training set:", error01_train, "/",
len(train_labels))
# print("0/1 error of testing set: ", error01_test, "/",
len(test_labels))
print("Time used: ", time.time() - ste, "sec")
print(
    "----- epoch{} end -----
-----\n".format(epoch + 1))

# conserve the model
# with open('model_data_' + str(epoch) + '.pkl', 'wb') as output:
#     pickle.dump(LeNet5, output, pickle.HIGHEST_PROTOCOL)

err_rate_list = np.array(err_rate_list).T

# This shows the error rate of training and testing data after each epoch
# x = np.arange(epochs)
# plt.xlabel('epochs')
# plt.ylabel('error rate')
# plt.plot(x, err_rate_list[0])
# plt.plot(x, err_rate_list[1])
# plt.legend(['training data', 'testing data'], loc='upper right')
# plt.show()

# return random-shuffled mini-batches
def random_mini_batches(image, label, mini_batch_size=256, one_batch=False):
    m = image.shape[0] # number of training examples
    mini_batches = []

    # Shuffle (image, label)
    permutation = list(np.random.permutation(m))
    shuffled_image = image[permutation, :, :, :]
    shuffled_label = label[permutation]

    # extract only one batch
    if one_batch:
        mini_batch_image = shuffled_image[0: mini_batch_size, :, :, :]
        mini_batch_label = shuffled_label[0: mini_batch_size]
        return mini_batch_image, mini_batch_label

    # Partition (shuffled_image, shuffled_Y). Minus the end case.
    num_complete_minibatches = math.floor(m / mini_batch_size)
    for k in range(0, num_complete_minibatches):
        mini_batch_image = shuffled_image[k * mini_batch_size: k *
mini_batch_size + mini_batch_size, :, :, :]
        mini_batch_label = shuffled_label[k * mini_batch_size: k *
mini_batch_size + mini_batch_size]
        mini_batch = (mini_batch_image, mini_batch_label)
        mini_batches.append(mini_batch)
    # Handling the end case (last mini-batch < mini_batch_size)

```

```

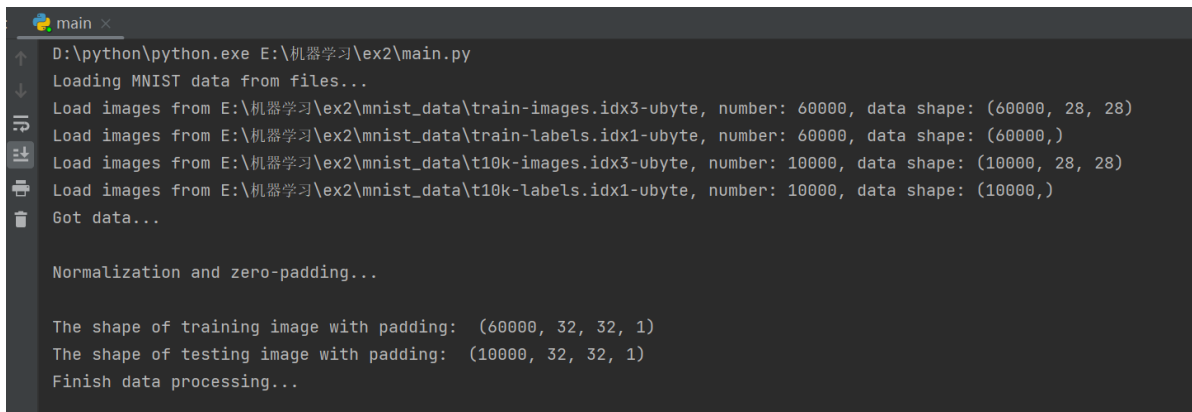
        if m % mini_batch_size != 0:
            mini_batch_image = shuffled_image[num_complete_minibatches *
mini_batch_size: m, :, :, :]
            mini_batch_label = shuffled_label[num_complete_minibatches *
mini_batch_size: m]
            mini_batch = (mini_batch_image, mini_batch_label)
            mini_batches.append(mini_batch)

    return mini_batches

```

实验结果

加载数据集处理：



```

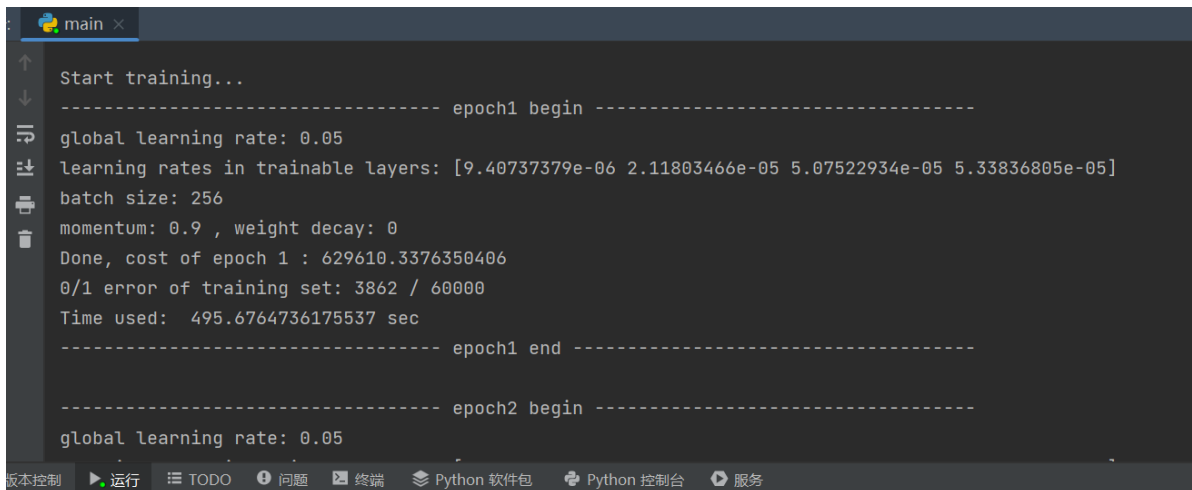
main x
D:\python\python.exe E:\机器学习\ex2\main.py
Loading MNIST data from files...
Load images from E:\机器学习\ex2\mnist_data\train-images.idx3-ubyte, number: 60000, data shape: (60000, 28, 28)
Load images from E:\机器学习\ex2\mnist_data\train-labels.idx1-ubyte, number: 60000, data shape: (60000,)
Load images from E:\机器学习\ex2\mnist_data\t10k-images.idx3-ubyte, number: 10000, data shape: (10000, 28, 28)
Load images from E:\机器学习\ex2\mnist_data\t10k-labels.idx1-ubyte, number: 10000, data shape: (10000,)
Got data...

Normalization and zero-padding...

The shape of training image with padding: (60000, 32, 32, 1)
The shape of testing image with padding: (10000, 32, 32, 1)
Finish data processing...

```

训练模型：

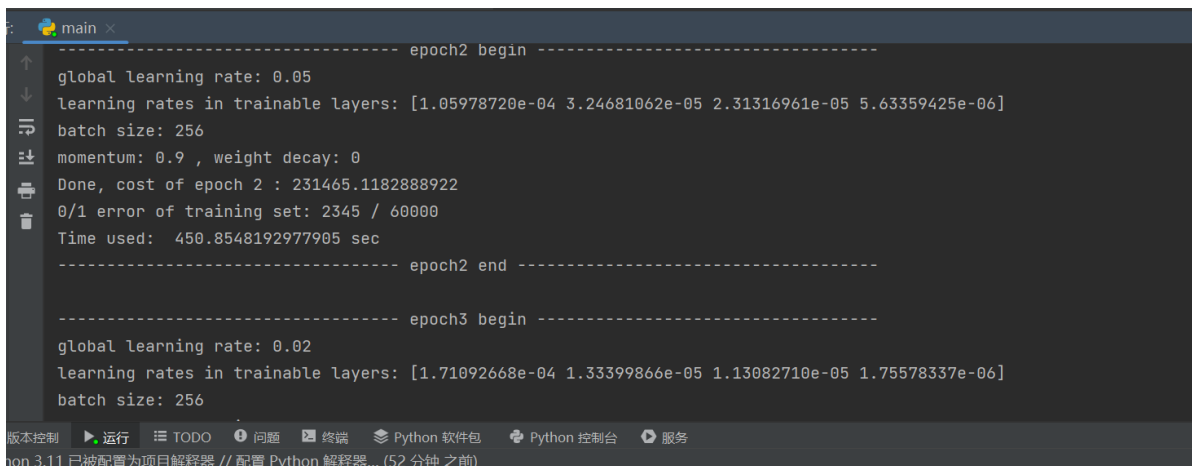


```

main x
Start training...
----- epoch1 begin -----
global learning rate: 0.05
learning rates in trainable layers: [9.40737379e-06 2.11803466e-05 5.07522934e-05 5.33836805e-05]
batch size: 256
momentum: 0.9 , weight decay: 0
Done, cost of epoch 1 : 629610.3376350406
0/1 error of training set: 3862 / 60000
Time used: 495.6764736175537 sec
----- epoch1 end -----

----- epoch2 begin -----
global learning rate: 0.05

```



```

----- epoch2 begin -----
global learning rate: 0.05
learning rates in trainable layers: [1.05978720e-04 3.24681062e-05 2.31316961e-05 5.63359425e-06]
batch size: 256
momentum: 0.9 , weight decay: 0
Done, cost of epoch 2 : 231465.118288922
0/1 error of training set: 2345 / 60000
Time used: 450.8548192977905 sec
----- epoch2 end -----

----- epoch3 begin -----
global learning rate: 0.02
learning rates in trainable layers: [1.71092668e-04 1.33399866e-05 1.13082710e-05 1.75578337e-06]
batch size: 256

```

```
main x
----- epoch3 end -----
----- epoch4 begin -----
global learning rate: 0.02
learning rates in trainable layers: [3.44543469e-04 1.37746028e-05 1.30995675e-05 1.60220161e-06]
batch size: 256
momentum: 0.9 , weight decay: 0
Done, cost of epoch 4 : 137613.32850608797
0/1 error of training set: 1550 / 60000
Time used: 465.58939719200134 sec
----- epoch4 end -----
----- epoch5 begin -----
global learning rate: 0.02
```

```
main x
----- epoch5 begin -----
global learning rate: 0.02
learning rates in trainable layers: [5.87628743e-04 1.38401418e-05 1.43192574e-05 1.50856187e-06]
batch size: 256
momentum: 0.9 , weight decay: 0
Done, cost of epoch 5 : 118989.40132147726
0/1 error of training set: 1329 / 60000
Time used: 469.6607255935669 sec
----- epoch5 end -----
----- epoch6 begin -----
global learning rate: 0.01
learning rates in trainable layers: [4.88708930e-04 7.62942251e-06 8.05196457e-06 7.11390614e-07]
batch size: 256
```

```
main x
----- epoch5 end -----
----- epoch6 begin -----
global learning rate: 0.01
learning rates in trainable layers: [4.88708930e-04 7.62942251e-06 8.05196457e-06 7.11390614e-07]
batch size: 256
momentum: 0.9 , weight decay: 0
Done, cost of epoch 6 : 101936.21294731769
0/1 error of training set: 1272 / 60000
Time used: 464.594176530838 sec
----- epoch6 end -----
----- epoch7 begin -----
```

```
运行: main x
----- epoch8 begin -----
global learning rate: 0.01
learning rates in trainable layers: [7.32344883e-04 8.53917800e-06 8.93783893e-06 6.97136875e-07]
batch size: 256
momentum: 0.9 , weight decay: 0
Done, cost of epoch 8 : 90534.04772297767
0/1 error of training set: 1114 / 60000
Time used: 515.0457110404968 sec
----- epoch8 end -----
----- epoch9 begin -----
global learning rate: 0.005
learning rates in trainable layers: [4.12872686e-04 4.07801886e-06 4.18554799e-06 3.44279491e-07]
```

结果:

```
运行: main ×
Done, cost of epoch 16 : 53527.16686486266
0/1 error of training set: 716 / 60000
Time used: 218.15260910987854 sec
----- epoch16 end -----
Finished training, the total training time is 3557.5133595466614s

Start testing...
error rate: 0.0146
Finished testing, the accuracy is 0.9854

进程已结束,退出代码0
```

16轮训练后, 正确率98.54%