# 实验介绍

深度学习(DL, Deep Learning)是机器学习(ML, Machine Learning)领域中一个新的研究方向，它被引入机器学习使其更接近于最初的目标——人工智能(AI, Artificial Intelligence)。本实验主要涉及深度学习中的卷积神经网络，将利用华为MindSpore深度学习框架实现MNIST手写体识别实验。

## 实验目的

本章实验的主要目的是了解并掌握深度学习卷积神经网络相关基础知识，在此基础上，基于华为自研MindSpore深度学习框架构建网络模型，实现图像识别相关任务，使大家能熟悉并掌握使用MindSpore深度学习框架实现深度学习实验任务。

## 实验清单

表格：实验、简述、难度、软件环境、硬件环境。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实验 | 简述 | 难度 | 软件环境 | 开发环境 |
| 手写体图片识别实验 | 基于MindSpore深度学习框架搭建LeNet网络，完成手写体数字识别。 | 初级 | Python3.7  Mindspore1.7 | ModelArts / PC 64bit |

## 实验开发环境

Mindspore-1.7（云端）

MindSpore-1.7（本地）

若选择在华为云ModelArts上快速搭建开发环境，可参考文末附录：ModelArts开发环境搭建。

若选择在本地搭建MindSpore开发环境进行实验，请参考《MindSpore环境搭建实验手册》。

# MNIST手写体识别实验

实验要求：请编程实现一个MNIST图片分类实现，补充2.3步骤中缺失的代码。本文档以学号-姓名的命名提交到智慧树平台。

## 实验介绍

请使用MindSpore深度学习框架实现一个简单的图片分类实验，整体流程如下：

1. 处理需要的数据集，这里使用了MNIST数据集。
2. 定义一个网络，这里我们使用LeNet网络。
3. 定义损失函数和优化器。
4. 加载数据集并进行训练，训练完成后，查看结果及保存模型文件。
5. 加载保存的模型，进行推理。
6. 验证模型，加载测试数据集和训练后的模型，验证结果精度。

## 实验准备

后续过程主要介绍在华为云ModelArts平台上运行此实验，如果在本地运行此实验，请参考章节1.3《MindSpore环境搭建实验手册》在本地安装MindSpore。

## 实验详细设计与实现

### 数据准备

我们示例中用到的MNIST数据集是由10类28\*28的灰度图片组成，训练数据集包含60000张图片，测试数据集包含10000张图片。

数据集可通过以下链接下载：

<https://zhuanyejianshe.obs.cn-north-4.myhuaweicloud.com/chuangxinshijianke/cv-nlp/MNIST.zip>，在浏览器打开链接即可下载数据集，也可以通过手写数字识别官方网站下载数据集，具体步骤如下：

MNIST数据集下载页面：http://yann.lecun.com/exdb/mnist/。页面提供4个数据集下载链接，其中前2个文件是训练数据需要，后2个文件是测试结果需要。

将数据集下载并解压到本地路径下，这里将数据集解压分别存放到工作区的./MNIST\_Data/train、./MNIST\_Data/test路径下。

目录结构如下：

└─MNIST\_Data

├─ test

│ t10k-images.idx3-ubyte

│ t10k-labels.idx1-ubyte

│

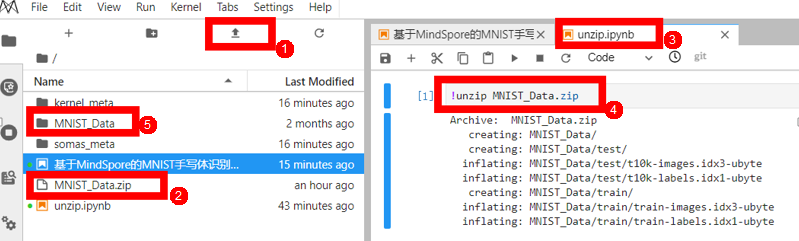
└─ train

train-images.idx3-ubyte

train-labels.idx1-ubyte

准备好数据后，将MNIST\_Data数据文件夹压缩(.zip格式)，然年上传到modelarts平台notebook开发界面，如下所示。

点击上传按钮，将MNIST\_Data.zip压缩文件上传，然后创建一个notebook文件，如图中的unzip.ipynb，使用命令”!unzip MNIST\_Data.zip”解压压缩文件，刷新后可以在左边栏中看到MNIST\_Data文件夹。



### 实验步骤

导入Python库&模块并配置运行信息

在使用前，导入需要的Python库。

目前使用到os库，为方便理解，其他需要的库，我们在具体使用到时再说明。

详细的MindSpore的模块说明，可以在MindSpore API页面中搜索查询。

可以通过context.set\_context来配置运行需要的信息，譬如运行模式、后端信息、硬件等信息。

导入context模块，配置运行需要的信息。

import os

import mindspore as ms

import mindspore.context as context

#transforms.c\_transforms用于通用型数据增强，vision.c\_transforms用于图像类数据增强

import mindspore.dataset.transforms.c\_transforms as C

import mindspore.dataset.vision.c\_transforms as CV

#nn模块用于定义网络，model模块用于编译模型，callback模块用于设定监督指标

from mindspore import nn

from mindspore.train import Model

from mindspore.train.callback import LossMonitor

#设定运行模式为图模式，运行硬件为昇腾芯片

context.set\_context(mode=context.GRAPH\_MODE, device\_target=CPU) # Ascend, CPU, GPU

在样例中我们配置样例运行使用图模式。根据实际情况配置硬件信息，譬如代码运行在Ascend AI处理器上，则device\_target选择Ascend，代码运行在CPU、GPU同理。详细参数说明，请参见context.set\_context接口说明。

数据处理

数据集对于训练非常重要，好的数据集可以有效提高训练精度和效率。在加载数据集前，我们通常会对数据集进行一些处理。

定义数据集及数据操作

我们定义一个函数create\_dataset来创建数据集。在这个函数中，我们定义好需要进行的数据增强和处理操作：

1. 定义数据集。

2. 定义进行数据增强和处理所需要的一些参数。

3. 根据参数，生成对应的数据增强操作。

4. 使用map映射函数，将数据操作应用到数据集。

5. 对生成的数据集进行处理。

#根据数据集存储地址，生成数据集

#函数参数设置：数据集路径、每个批次的样本（默认为32）、每个epoch开始之前是否随机打乱数据（默认为True）、数据加载时的并行工作者数量（默认为4）

def create\_dataset(data\_dir, batch\_size=32, shuffle=True, num\_parallel\_workers=4):

# 定义数据集

dataset = ds.MnistDataset(data\_dir, num\_parallel\_workers=num\_parallel\_workers)

# 数据增强和处理参数

resize\_op = CV.Resize(size=(32, 32)) # 调整图像大小为 (32, 32)

normalize\_op = CV.Rescale(1.0 / 255.0, 0.0) # 将像素值从[0, 255]缩放到 [0, 1] 的范围

dtype\_op = C.TypeCast(mstype.int32) # 将标签数据类型转换为 int32

change\_channels\_op = CV.HWC2CHW() # 转换通道顺序，从 HWC 到 CHW

# 应用数据处理操作

dataset = dataset.map(operations=dtype\_op, input\_columns="label", num\_parallel\_workers=4)

dataset = dataset.map(operations=[resize\_op, normalize\_op, change\_channels\_op], input\_columns="image",

num\_parallel\_workers=4)

# 对数据集进行shuffle操作

if shuffle:

dataset = dataset.shuffle(buffer\_size=1000)

# 设置batch\_size，并丢弃不足batch\_size的数据

dataset = dataset.batch(batch\_size, drop\_remainder=True)

return dataset

其中，

batch\_size：每组包含的数据个数，现设置每组包含32个数据。

先进行修改图片尺寸，归一化，修改图像频道数等工作，再修改标签的数据类型。最后进行shuffle操作，同时设定batch\_size，设置drop\_remainder为True，则数据集中不足最后一个batch的数据会被抛弃。

MindSpore支持进行多种数据处理和增强的操作，各种操作往往组合使用，具体可以参考数据处理与数据增强章节。

# 运行函数 查看结果

train\_data\_dir = 'C:\\Users\\hkhk3\\Desktop\\file\\pythonProject\\MNIST\_Data\\train'

test\_data\_dir = 'C:\\Users\\hkhk3\\Desktop\\file\\pythonProject\\MNIST\_Data\\test'

batch\_size = 32

train\_dataset = create\_dataset(train\_data\_dir, batch\_size=batch\_size, shuffle=True, num\_parallel\_workers=4)

test\_dataset = create\_dataset(test\_data\_dir, batch\_size=batch\_size, shuffle=True, num\_parallel\_workers=4)

data\_next=train\_dataset.create\_dict\_iterator(output\_numpy=True).\_\_next\_\_()

print('训练数据集数量：', train\_dataset.get\_dataset\_size())

print('测试数据集数量：', test\_dataset.get\_dataset\_size())

print('通道数/图像长/宽：', data\_next['image'].shape)

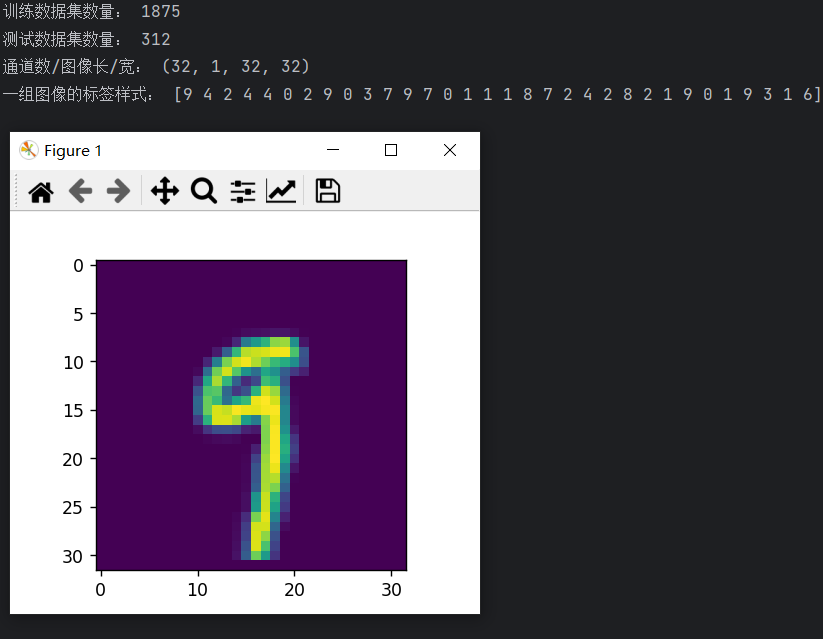
print('一组图像的标签样式：', data\_next['label'])

plt.figure()

plt.imshow(data\_next['image'][0,0, ...])

plt.grid(False)

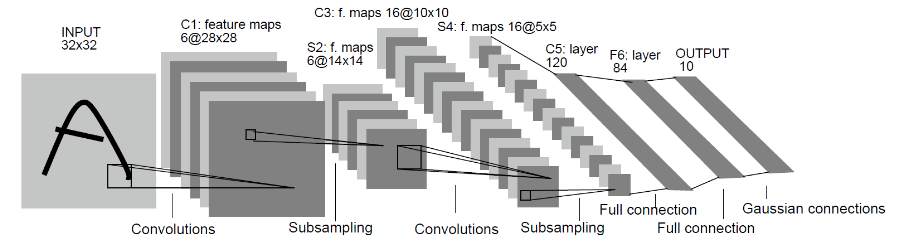
plt.show()



定义网络

我们选择相对简单的LeNet网络。LeNet网络不包括输入层的情况下，共有7层：2个卷积层、2个下采样层（池化层）、3个全连接层。每层都包含不同数量的训练参数，如下图所示：

LeNet-5



更多的LeNet网络的介绍不在此赘述，希望详细了解LeNet网络，可以查询http://yann.lecun.com/exdb/lenet/。

使用MindSpore定义神经网络需要继承mindspore.nn.cell.Cell。Cell是所有神经网络（Conv2d等）的基类。

神经网络的各层需要预先在\_\_init\_\_方法中定义，然后通过定义construct方法来完成神经网络的前向构造。按照LeNet的网络结构，定义网络各层如下：

#定义模型结构，MindSpore中的模型时通过construct定义模型结构，在\_\_init\_\_中初始化各层的对象

class Identification\_Net(nn.Cell):

def \_\_init\_\_(self, num\_class=10, channel=1, dropout\_ratio=0.5, trun\_sigma=0.02):

super(Identification\_Net, self).\_\_init\_\_()

self.num\_class = num\_class

self.channel = channel

self.dropout\_ratio = dropout\_ratio

# 设置卷积层

# 第一个参数是输入通道数，根据输入数据来设置

# 第二个参数是输出通道数，即特征映射的数量（6）

# 第三个参数是卷积核的大小（5\*5）

# 第四个参数是卷积操作时滑动卷积核的步幅。幅为1，表示卷积核每次移动一个像素

# 第五个参数是用于输入特征图周围的零填充数量，以控制输出特征图的尺寸。填充为0，意味着没有填充。

# 第六个参数是填充模式，用于控制填充的方式。设置 "valid"，表示使用有效卷积，即没有填充。

# 输出高度 = [(输入高度 - 卷积核高度 + 2 \* 填充高度) / 步幅] + 1。即 28 = (32 - 5 + 2 \* 0) + 1

self.c1 = nn.Conv2d(in\_channels=self.channel, out\_channels=6,

kernel\_size=5,stride=1, padding=0,

pad\_mode="valid")

self.c3 = nn.Conv2d(in\_channels=6, out\_channels=16,

kernel\_size=5,stride=1, padding=0,

pad\_mode="valid")

# 设置 ReLU 激活函数

self.relu = nn.ReLU()

# 设置最大池化层

# 将每个2x2区域压缩为一个单一的值。

self.s2 = nn.MaxPool2d(kernel\_size=2, stride=2)

self.s4 = nn.MaxPool2d(kernel\_size=2, stride=2)

# 创建一个用于将特征图展平的层

self.flatten = nn.Flatten()

# 设置全连接层

# 第一个参数为输入的特征数

# 第一个参数为输出的特征数

self.c5 = nn.Dense(16 \* 5 \* 5, 120, weight\_init=TruncatedNormal(trun\_sigma))

self.f6 = nn.Dense(120,84,

weight\_init=TruncatedNormal(trun\_sigma))

self.output = nn.Dense(84, self.num\_class,

weight\_init=TruncatedNormal(trun\_sigma))

# 构建模型

def construct(self, x):

x = self.c1(x)

x = self.relu(x)

x = self.s2(x)

x = self.c3(x)

x = self.relu(x)

x = self.s4(x)

x = self.flatten(x)

x = self.c5(x)

x = self.relu(x)

x = self.f6(x)

x = self.relu(x)

x = self.output(x)

return x

定义损失函数及优化器

在进行定义之前，先简单介绍损失函数及优化器的概念。

损失函数：又叫目标函数，用于衡量预测值与实际值差异的程度。深度学习通过不停地迭代来缩小损失函数的值。定义一个好的损失函数，可以有效提高模型的性能。

优化器：用于最小化损失函数，从而在训练过程中改进模型。

定义了损失函数后，可以得到损失函数关于权重的梯度。梯度用于指示优化器优化权重的方向，以提高模型性能。

MindSpore支持的损失函数有SoftmaxCrossEntropyWithLogits、L1Loss、MSELoss等。这里使用SoftmaxCrossEntropyWithLogits损失函数。

MindSpore提供了callback机制，可以在训练过程中执行自定义逻辑，这里使用框架提供的ModelCheckpoint为例。 ModelCheckpoint可以保存网络模型和参数，以便进行后续的fine-tuning（微调）操作。

# 构建训练、验证函数进行模型训练和验证，提供数据路径，设定学习率，epoch数量

# 实例化网络

net = Identification\_Net()

# 计算 softmax 交叉熵，以此作为损失函数

net\_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')

# 定义优化器

net\_opt = nn.Momentum(net.trainable\_params(), learning\_rate=0.01, momentum=0.9)

# 创建模型

model = Model(net, loss\_fn=net\_loss, optimizer=net\_opt, metrics={"acc"})

# 设定 callback 监控指标

config\_ck = CheckpointConfig(save\_checkpoint\_steps=1875, keep\_checkpoint\_max=10)

ckpoint\_cb = ModelCheckpoint(prefix="checkpoint\_classification", directory='C:\\Users\\hkhk3\\Desktop\\meta',

config=config\_ck)

# 模型训练函数

def train\_net(model, epoch\_size, train\_data\_dir, ckpoint\_cb):

train\_dataset = create\_dataset(train\_data\_dir, batch\_size=batch\_size, shuffle=True, num\_parallel\_workers=4)

model.train(epoch\_size, train\_dataset, callbacks=[ckpoint\_cb, LossMonitor(100)], dataset\_sink\_mode=False)

# 模型验证函数

def test\_net(model, test\_data\_dir):

test\_dataset = create\_dataset(test\_data\_dir, batch\_size=batch\_size, shuffle=True, num\_parallel\_workers=4)

acc = model.eval(test\_dataset, dataset\_sink\_mode=False)

print(acc)

开始训练及验证过程

#main函数负责调用之前定义的函数，完成整个训练验证过程

def main():

train\_data\_dir = 'C:\\Users\\hkhk3\\Desktop\\file\\pythonProject\\MNIST\_Data\\train'

test\_data\_dir = 'C:\\Users\\hkhk3\\Desktop\\file\\pythonProject\\MNIST\_Data\\test'

batch\_size = 32

net = Identification\_Net()

net\_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')

net\_opt = nn.Momentum(net.trainable\_params(), learning\_rate=0.01, momentum=0.9)

model = Model(net, loss\_fn=net\_loss, optimizer=net\_opt, metrics={"acc"})

config\_ck = CheckpointConfig(save\_checkpoint\_steps=1875, keep\_checkpoint\_max=10)

ckpoint\_cb = ModelCheckpoint(prefix="checkpoint\_classification", directory='C:\\Users\\hkhk3\\Desktop\\ck',config=config\_ck)

epoch\_size = 10

train\_net(model, epoch\_size, train\_data\_dir, ckpoint\_cb)

test\_net(model, test\_data\_dir)

训练过程中会打印loss值，类似下图。loss值会波动，但总体来说loss值会逐步减小，精度逐步提高。每个人运行的loss值有一定随机性，不一定完全相同。 训练过程中loss打印示例如下：

epoch: 1 step: 100, loss is 2.2944586277008057

epoch: 1 step: 200, loss is 2.29561710357666

epoch: 1 step: 300, loss is 2.3176252841949463

epoch: 1 step: 400, loss is 2.315075635910034

epoch: 1 step: 500, loss is 2.2904446125030518

epoch: 1 step: 600, loss is 2.322044849395752

epoch: 1 step: 700, loss is 2.2967653274536133

epoch: 1 step: 800, loss is 2.2988762855529785

epoch: 1 step: 900, loss is 2.320201873779297

epoch: 1 step: 1000, loss is 2.311863899230957

epoch: 1 step: 1100, loss is 2.2926671504974365

epoch: 1 step: 1200, loss is 2.309640645980835

epoch: 1 step: 1300, loss is 2.2834720611572266

epoch: 1 step: 1400, loss is 2.3059637546539307

epoch: 1 step: 1500, loss is 2.299394130706787

epoch: 1 step: 1600, loss is 2.310436725616455

epoch: 1 step: 1700, loss is 2.315976142883301

epoch: 1 step: 1800, loss is 2.296213388442993

epoch: 2 step: 25, loss is 2.283597707748413

epoch: 2 step: 125, loss is 2.3012449741363525

epoch: 2 step: 225, loss is 2.31105375289917

epoch: 2 step: 325, loss is 2.2985899448394775

epoch: 2 step: 425, loss is 2.315383195877075

epoch: 2 step: 525, loss is 2.324939727783203

epoch: 2 step: 625, loss is 2.3003408908843994

epoch: 2 step: 725, loss is 2.30863094329834

epoch: 2 step: 825, loss is 2.2963743209838867

epoch: 2 step: 925, loss is 2.3090014457702637

epoch: 2 step: 1025, loss is 2.3041305541992188

epoch: 2 step: 1125, loss is 2.296637535095215

epoch: 2 step: 1225, loss is 2.3036141395568848

epoch: 2 step: 1325, loss is 2.296628952026367

epoch: 2 step: 1425, loss is 2.2813730239868164

epoch: 2 step: 1525, loss is 2.3168323040008545

epoch: 2 step: 1625, loss is 2.3077688217163086

epoch: 2 step: 1725, loss is 2.3149056434631348

epoch: 2 step: 1825, loss is 2.3015007972717285

epoch: 3 step: 50, loss is 2.306347131729126

epoch: 3 step: 150, loss is 2.3097147941589355

epoch: 3 step: 250, loss is 2.2957308292388916

epoch: 3 step: 350, loss is 2.3107025623321533

epoch: 3 step: 450, loss is 2.3149778842926025

epoch: 3 step: 550, loss is 2.3092494010925293

epoch: 3 step: 650, loss is 2.3036224842071533

epoch: 3 step: 750, loss is 2.2976107597351074

epoch: 3 step: 850, loss is 2.318385362625122

epoch: 3 step: 950, loss is 2.300006628036499

epoch: 3 step: 1050, loss is 2.310497999191284

epoch: 3 step: 1150, loss is 2.2936794757843018

epoch: 3 step: 1250, loss is 2.301745653152466

epoch: 3 step: 1350, loss is 2.326392650604248

epoch: 3 step: 1450, loss is 2.2857747077941895

epoch: 3 step: 1550, loss is 2.282460927963257

epoch: 3 step: 1650, loss is 2.3091485500335693

epoch: 3 step: 1750, loss is 2.3016796112060547

epoch: 3 step: 1850, loss is 2.3052446842193604

epoch: 4 step: 75, loss is 2.2996761798858643

epoch: 4 step: 175, loss is 2.2928831577301025

epoch: 4 step: 275, loss is 2.295219659805298

epoch: 4 step: 375, loss is 2.3130242824554443

epoch: 4 step: 475, loss is 2.3057148456573486

epoch: 4 step: 575, loss is 2.2966322898864746

epoch: 4 step: 675, loss is 2.286890745162964

epoch: 4 step: 775, loss is 2.298489570617676

epoch: 4 step: 875, loss is 2.3036305904388428

epoch: 4 step: 975, loss is 2.304704189300537

epoch: 4 step: 1075, loss is 2.3042778968811035

epoch: 4 step: 1175, loss is 2.27960205078125

epoch: 4 step: 1275, loss is 2.323477268218994

epoch: 4 step: 1375, loss is 2.278388261795044

epoch: 4 step: 1475, loss is 2.332613706588745

epoch: 4 step: 1575, loss is 2.2883265018463135

epoch: 4 step: 1675, loss is 2.2878711223602295

epoch: 4 step: 1775, loss is 2.2967846393585205

epoch: 4 step: 1875, loss is 2.3145997524261475

epoch: 5 step: 100, loss is 2.3026680946350098

epoch: 5 step: 200, loss is 2.301093339920044

epoch: 5 step: 300, loss is 2.3109707832336426

epoch: 5 step: 400, loss is 2.3028388023376465

epoch: 5 step: 500, loss is 2.2780370712280273

epoch: 5 step: 600, loss is 2.295452117919922

epoch: 5 step: 700, loss is 2.3092658519744873

epoch: 5 step: 800, loss is 2.2779204845428467

epoch: 5 step: 900, loss is 2.306297540664673

epoch: 5 step: 1000, loss is 2.2962653636932373

epoch: 5 step: 1100, loss is 2.3095626831054688

epoch: 5 step: 1200, loss is 2.2958292961120605

epoch: 5 step: 1300, loss is 2.307589530944824

epoch: 5 step: 1400, loss is 2.2833151817321777

epoch: 5 step: 1500, loss is 2.306239128112793

epoch: 5 step: 1600, loss is 2.285278558731079

epoch: 5 step: 1700, loss is 2.319387435913086

epoch: 5 step: 1800, loss is 2.3055667877197266

epoch: 6 step: 25, loss is 2.277600049972534

epoch: 6 step: 125, loss is 2.3105907440185547

epoch: 6 step: 225, loss is 2.293999195098877

epoch: 6 step: 325, loss is 2.3080716133117676

epoch: 6 step: 425, loss is 2.292941093444824

epoch: 6 step: 525, loss is 2.3230860233306885

epoch: 6 step: 625, loss is 2.3008463382720947

epoch: 6 step: 725, loss is 2.2947893142700195

epoch: 6 step: 825, loss is 1.3611323833465576

epoch: 6 step: 925, loss is 0.9378198385238647

epoch: 6 step: 1025, loss is 0.2870473265647888

epoch: 6 step: 1125, loss is 0.19610761106014252

epoch: 6 step: 1225, loss is 0.1440526396036148

epoch: 6 step: 1325, loss is 0.16402409970760345

epoch: 6 step: 1425, loss is 0.24512581527233124

epoch: 6 step: 1525, loss is 0.18240363895893097

epoch: 6 step: 1625, loss is 0.14576420187950134

epoch: 6 step: 1725, loss is 0.15554696321487427

epoch: 6 step: 1825, loss is 0.04604561626911163

epoch: 7 step: 50, loss is 0.013771263882517815

epoch: 7 step: 150, loss is 0.10263610631227493

epoch: 7 step: 250, loss is 0.058849919587373734

epoch: 7 step: 350, loss is 0.007950274273753166

epoch: 7 step: 450, loss is 0.12979188561439514

epoch: 7 step: 550, loss is 0.03240228816866875

epoch: 7 step: 650, loss is 0.38169434666633606

epoch: 7 step: 750, loss is 0.02163003943860531

epoch: 7 step: 850, loss is 0.021358449012041092

epoch: 7 step: 950, loss is 0.1347445398569107

epoch: 7 step: 1050, loss is 0.2562379837036133

epoch: 7 step: 1150, loss is 0.01448530051857233

epoch: 7 step: 1250, loss is 0.1077776774764061

epoch: 7 step: 1350, loss is 0.05154682323336601

epoch: 7 step: 1450, loss is 0.011656306684017181

epoch: 7 step: 1550, loss is 0.057940684258937836

epoch: 7 step: 1650, loss is 0.022721847519278526

epoch: 7 step: 1750, loss is 0.02413080632686615

epoch: 7 step: 1850, loss is 0.010984539985656738

epoch: 8 step: 75, loss is 0.08162587881088257

epoch: 8 step: 175, loss is 0.023043718189001083

epoch: 8 step: 275, loss is 0.09092047810554504

epoch: 8 step: 375, loss is 0.03600817546248436

epoch: 8 step: 475, loss is 0.02323400042951107

epoch: 8 step: 575, loss is 0.03532204404473305

epoch: 8 step: 675, loss is 0.09841301292181015

epoch: 8 step: 775, loss is 0.01737687923014164

epoch: 8 step: 875, loss is 0.002248379634693265

epoch: 8 step: 975, loss is 0.0324462354183197

epoch: 8 step: 1075, loss is 0.25813376903533936

epoch: 8 step: 1175, loss is 0.011362861841917038

epoch: 8 step: 1275, loss is 0.03574710711836815

epoch: 8 step: 1375, loss is 0.0630628764629364

epoch: 8 step: 1475, loss is 0.0633833184838295

epoch: 8 step: 1575, loss is 0.037189189344644547

epoch: 8 step: 1675, loss is 0.0005154651589691639

epoch: 8 step: 1775, loss is 0.009948762133717537

epoch: 8 step: 1875, loss is 0.028038926422595978

epoch: 9 step: 100, loss is 0.010636284947395325

epoch: 9 step: 200, loss is 0.0019423938356339931

epoch: 9 step: 300, loss is 0.012915506027638912

epoch: 9 step: 400, loss is 0.1430349200963974

epoch: 9 step: 500, loss is 0.14049935340881348

epoch: 9 step: 600, loss is 0.0010314933024346828

epoch: 9 step: 700, loss is 0.011100462637841702

epoch: 9 step: 800, loss is 0.004534405190497637

epoch: 9 step: 900, loss is 0.0007584925624541938

epoch: 9 step: 1000, loss is 0.08401961624622345

epoch: 9 step: 1100, loss is 0.003946717828512192

epoch: 9 step: 1200, loss is 0.04281981289386749

epoch: 9 step: 1300, loss is 0.37934744358062744

epoch: 9 step: 1400, loss is 0.0013046488165855408

epoch: 9 step: 1500, loss is 0.22481177747249603

epoch: 9 step: 1600, loss is 0.10323499888181686

epoch: 9 step: 1700, loss is 0.0008904807036742568

epoch: 9 step: 1800, loss is 0.015997357666492462

epoch: 10 step: 25, loss is 0.09334563463926315

epoch: 10 step: 125, loss is 0.005168928764760494

epoch: 10 step: 225, loss is 0.015141031704843044

epoch: 10 step: 325, loss is 0.033707909286022186

epoch: 10 step: 425, loss is 0.0001276988914469257

epoch: 10 step: 525, loss is 0.0011788326082751155

epoch: 10 step: 625, loss is 0.05111776292324066

epoch: 10 step: 725, loss is 0.03882221132516861

epoch: 10 step: 825, loss is 0.006086088251322508

epoch: 10 step: 925, loss is 0.053583331406116486

epoch: 10 step: 1025, loss is 0.003766985610127449

epoch: 10 step: 1125, loss is 0.2244722545146942

epoch: 10 step: 1225, loss is 0.09338925033807755

epoch: 10 step: 1325, loss is 0.0034541001077741385

epoch: 10 step: 1425, loss is 0.09924978762865067

epoch: 10 step: 1525, loss is 0.02115337923169136

epoch: 10 step: 1625, loss is 0.014407618902623653

epoch: 10 step: 1725, loss is 0.03635920584201813

epoch: 10 step: 1825, loss is 0.0005026772269047797

{'acc': 0.9870793269230769}