计算机视觉实验一

实验目的

① 加强对基于Mindspore的神经网络模型构建流程的理解。

② 掌握如何用Mindspore实现卷积神经网络的构建。

③ 学会利用checkpoint函数保存模型参数。

④ 掌握如何利用模型预测单张图像的分类结果。

实验内容

任务一：按照华为平台实验手册进行操作

要求：熟悉实验环境，掌握卷积神经网络模型程序流程
具体：记录并观察实验结果，如损失随迭代轮数变化等

任务二：LeNet-5模型对比

要求：调节实验参数，进行实验对比及分析
　实验参数包括：
　1.训练批次大小，迭代轮数，学习速率，最优化方法等

任务三：卷积神经网络模型设计

要求：改变网络结构，进行实验对比及分析
具体：
　1.卷积层数，卷积核尺寸及步长，激活函数，池化方法，全连接层层数，各层节点数目
　2.添加和不添加BN层

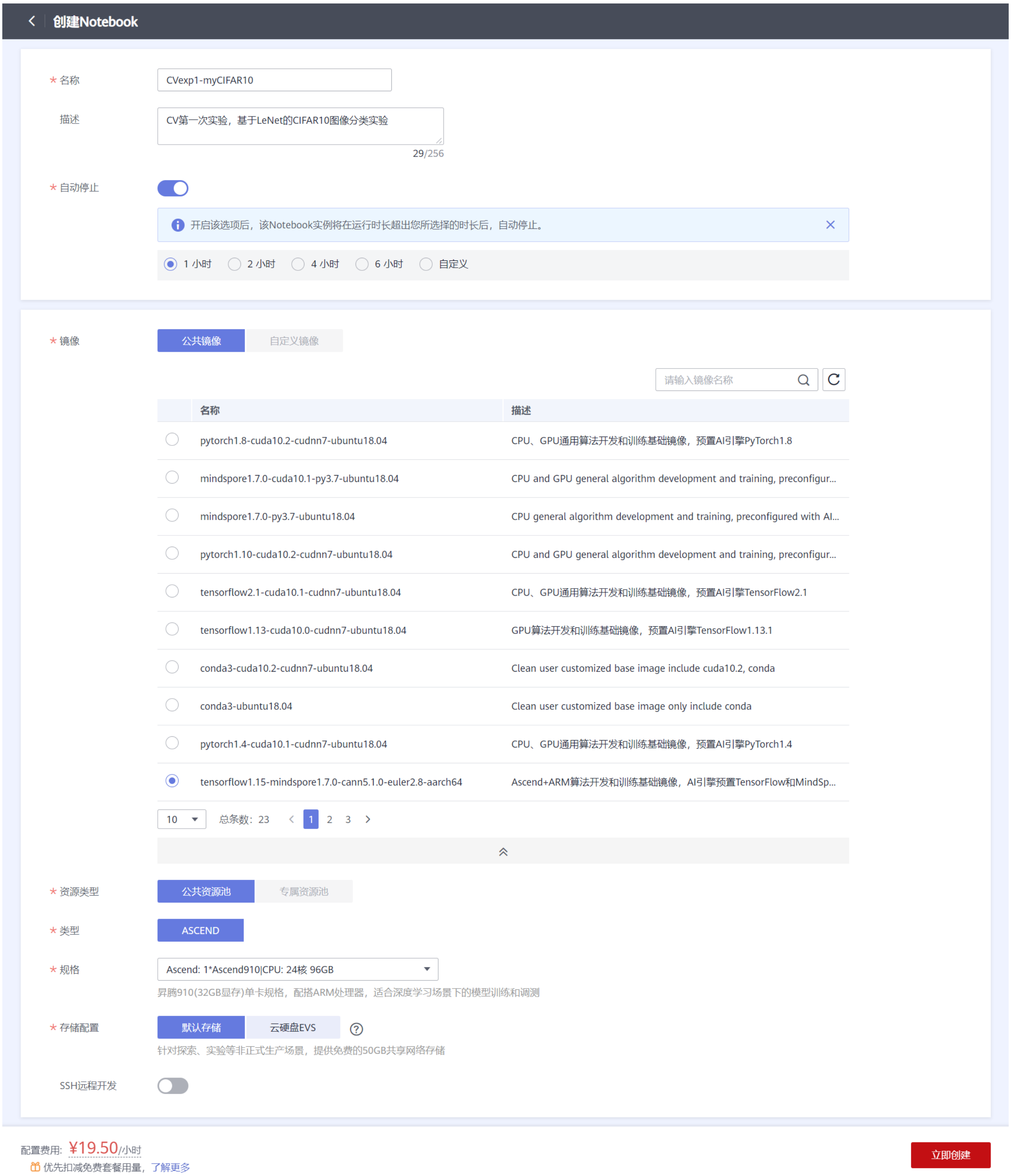
实验过程

任务一：按照华为平台实验手册进行操作

1.创建华为云Notebook

1.1进入Modelarts，选择Notebook

1.2创建Notebook



创建Notebook
* 名称
CVexp1-myCIFAR10
描述
CV第一次实验，基于LeNet的CIFAR10图像分类实验
29/256
* 自动停止
开启该选项后，该Notebook实例将在运行时长超出您所选择的时长后，自动停止。
1 小时
2 小时
4 小时
6 小时
自定义
* 镜像
公共镜像
自定义镜像
请输入镜像名称
名称
描述
pytorch1.8-cuda10.2-cudnn7-ubuntu18.04
CPU、GPU通用算法开发和训练基础镜像，预置AI引擎PyTorch1.8
mindspore1.7.0-cuda10.1-py3.7-ubuntu18.04
CPU and GPU general algorithm development and training, preconfigur...
mindspore1.7.0-py3.7-ubuntu18.04
CPU general algorithm development and training, preconfigured with AI...
pytorch1.10-cuda10.2-cudnn7-ubuntu18.04
CPU and GPU general algorithm development and training, preconfigur...
tensorflow2.1-cuda10.1-cudnn7-ubuntu18.04
CPU、GPU通用算法开发和训练基础镜像，预置AI引擎TensorFlow2.1
tensorflow1.13-cuda10.0-cudnn7-ubuntu18.04
GPU算法开发和训练基础镜像，预置AI引擎TensorFlow1.13.1
conda3-cuda10.2-cudnn7-ubuntu18.04
Clean user customized base image include cuda10.2, conda
conda3-ubuntu18.04
Clean user customized base image only include conda
pytorch1.4-cuda10.1-cudnn7-ubuntu18.04
CPU、GPU通用算法开发和训练基础镜像，预置AI引擎PyTorch1.4
tensorflow1.15-mindspore1.7.0-cann5.1.0-euler2.8-aarch64
Ascend+ARM算法开发和训练基础镜像，AI引擎预置TensorFlow和MindSp...
10
总条数：23
1
2
3
* 资源类型
公共资源池
专属资源池
* 类型
ASCEND
* 规格
Ascend: 1*Ascend910|CPU: 24核 96GB
昇腾910(32GB显存)单卡规格，配搭ARM处理器，适合深度学习场景下的模型训练和调测
* 存储配置
默认存储
云硬盘EVS
针对探索、实验等非正式生产场景，提供免费的50GB共享网络存储
SSH远程开发
配置费用：¥19.50/小时
优先扣减免费套餐用量，了解更多
立即创建

1.3启动Notebook

2.导入相关实验模块

```
import mindspore
# 载入mindspore的默认数据集
import mindspore.dataset as ds
# 常用转化用算子
import mindspore.dataset.transforms.c_transforms as C
# 图像转化用算子
####____####
import mindspore.dataset.vision.c_transforms as CV
from mindspore.common import dtype as mstype
# mindspore的tensor
from mindspore import Tensor


# 各类网络层都在nn里面
import mindspore.nn as nn
# 参数初始化的方式

from mindspore.common.initializer import TruncatedNormal
# 设置mindspore运行的环境
from mindspore import context
# 引入训练时候会使用到回调函数，如checkpoint, lossMoniter
from mindspore.train.callback import ModelCheckpoint, CheckpointConfig, LossMonitor, TimeMonitor
# 引入模型
from mindspore.train import Model
# 引入评估模型的包
from mindspore.nn.metrics import Accuracy

# numpy
import numpy as np
# 画图用
import matplotlib.pyplot as plt
```

```python
####____####
# 下载数据相关的包
import os
import requests
import zipfile
```

3.数据集展示与数据初始化

3.1数据集下载

```
!wget https://ascend-professional-construction-dataset.obs.cn-north-4.myhuaweicloud.com/ComputerVision/cifar10_mindspore.zip
!unzip cifar10_mindspore.zip
```

```
--2022-12-03 10:21:55--  https://ascend-professional-construction-dataset.obs.cn-north-4.myhuaweicloud.com/ComputerVision/cifar10_mindspore.zip
Resolving ascend-professional-construction-dataset.obs.cn-north-4.myhuaweicloud.com (ascend-professional-construction-dataset.obs.cn-north-4.myhuaweicloud.com)...
100.125.81.126, 100.125.81.253, 100.125.81.190
Connecting to ascend-professional-construction-dataset.obs.cn-north-4.myhuaweicloud.com (ascend-professional-construction-dataset.obs.cn-north-4.myhuaweicloud.co
m)|100.125.81.126|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 170441801 (163M) [application/zip]
Saving to: ‘cifar10_mindspore.zip’

cifar10_mindspore.z 100%[===================>] 162.55M   141MB/s    in 1.2s

2022-12-03 10:21:56 (141 MB/s) - ‘cifar10_mindspore.zip’ saved [170441801/170441801]

Archive:  cifar10_mindspore.zip
   creating: data/
   creating: data/10-batches-bin/
  inflating: data/10-batches-bin/batches.meta.txt
  inflating: data/10-batches-bin/data_batch_1.bin
  inflating: data/10-batches-bin/data_batch_2.bin
  inflating: data/10-batches-bin/data_batch_3.bin
  inflating: data/10-batches-bin/data_batch_4.bin
  inflating: data/10-batches-bin/data_batch_5.bin
   creating: data/10-verify-bin/
  inflating: data/10-verify-bin/test_batch.bin
   creating: images/
  inflating: images/01.png
  inflating: images/lenet.jpg
   creating: results/
```

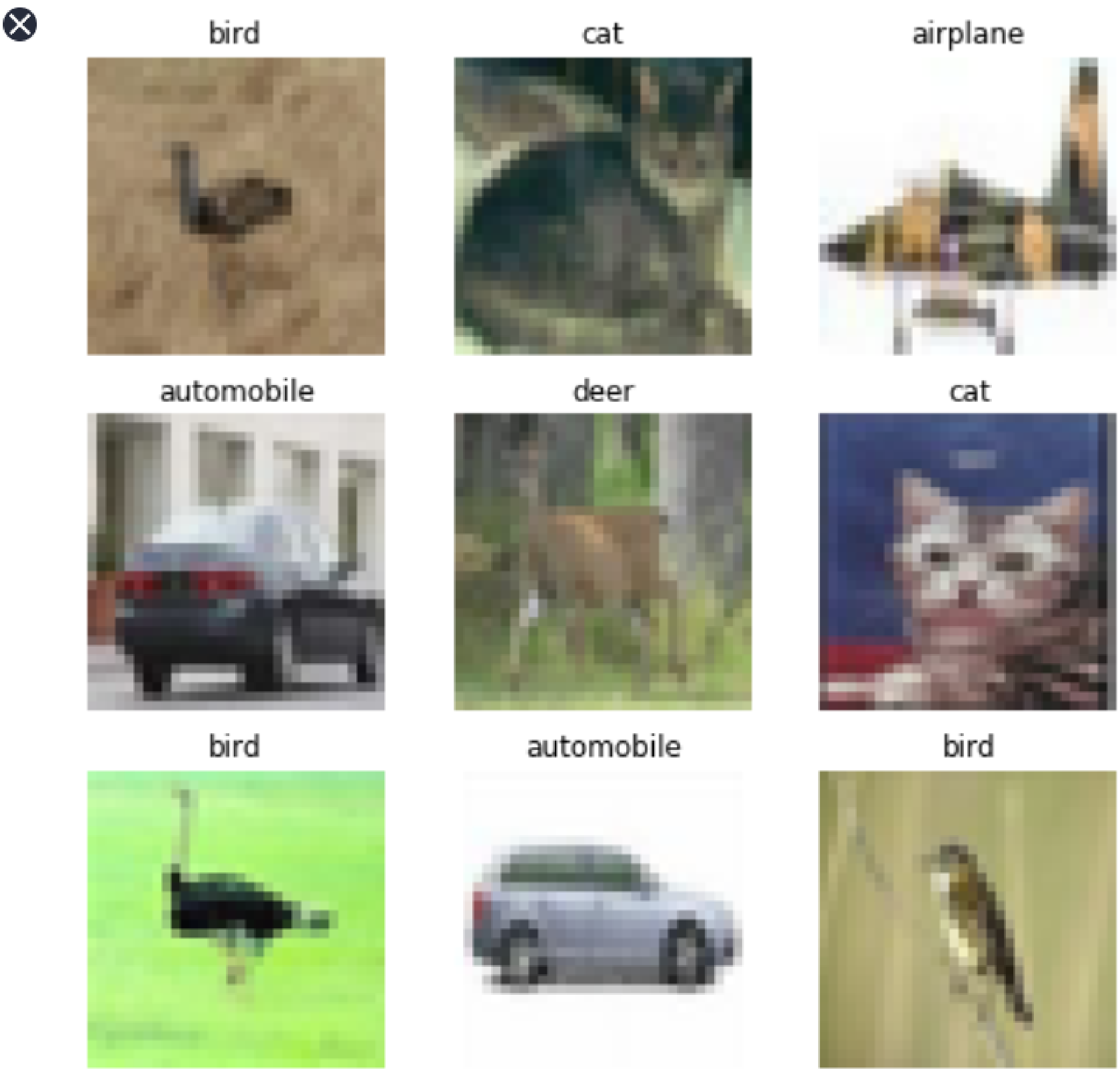
在terminal中查看

```
Using user ma-user
EulerOS 2.0 (SP8), CANN-5.1.RC1.1
Tips:
1) Navigate to the target conda environment. For details, see /home/ma-user/README.
2) Copy (Ctrl+C) and paste (Ctrl+V) on the jupyter terminal.
3) Store your data in /home/ma-user/work, to which a persistent volume is mounted.
[ma-user work]$ls
cifar10_mindspore.zip  cifar10图像分类.ipynb  data  images  myCIFAR.ipynb  results
[ma-user work]$cd data
[ma-user data]$ls
10-batches-bin  10-verify-bin
[ma-user data]$
```

3.2查看数据集

```python
#创建图像标签列表
category_dict = {0:'airplane',1:'automobile',2:'bird',3:'cat',4:'deer',5:'dog',
                 6:'frog',7:'horse',8:'ship',9:'truck'}


####____####
```

```
current_path = os.getcwd()
data_path = os.path.join(current_path, 'data/10-verify-bin')
cifar_ds = ds.Cifar10Dataset(data_path)

# 设置图像大小
plt.figure(figsize=(8,8))
i = 1
# 打印9张子图
for dic in cifar_ds.create_dict_iterator():
    plt.subplot(3,3,i)
    ####____####
    plt.imshow(dic['image'].asnumpy())
    plt.xticks([])
    plt.yticks([])
    plt.axis('off')
    plt.title(category_dict[dic['label'].asnumpy().sum()])
    i +=1
    if i > 9 :
        break

plt.show()
```



3.3定义数据预处理步骤

```python
def get_data(datapath):
    cifar_ds = ds.Cifar10Dataset(datapath)
    return cifar_ds

def process_dataset(cifar_ds,batch_size =32,status="train"):
    '''
    ---- 定义算子 ----
    '''
    # 归一化
    rescale = 1.0 / 255.0
    # 平移
    shift = 0.0

    resize_op = CV.Resize((32, 32))
    rescale_op = CV.Rescale(rescale, shift)
    # 对于RGB三通道分别设定mean和std
    normalize_op = CV.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
    if status == "train":
        # 随机裁剪
        random_crop_op = CV.RandomCrop([32, 32], [4, 4, 4, 4])
        # 随机翻转
        random_horizontal_op = CV.RandomHorizontalFlip()
    # 通道变化
    channel_swap_op = CV.HWC2CHW()
    # 类型变化
    typecast_op = C.TypeCast(mstype.int32)

    '''
    ---- 算子运算 ----
    '''
    cifar_ds = cifar_ds.map(input_columns="label", operations=typecast_op)
    if status == "train":
        cifar_ds = cifar_ds.map(input_columns="image", operations=random_crop_op)
        cifar_ds = cifar_ds.map(input_columns="image", operations=random_horizontal_op)
    cifar_ds = cifar_ds.map(input_columns="image", operations=resize_op)
    cifar_ds = cifar_ds.map(input_columns="image", operations=rescale_op)
    cifar_ds = cifar_ds.map(input_columns="image", operations=normalize_op)
    cifar_ds = cifar_ds.map(input_columns="image", operations=channel_swap_op)

    # shuffle
    cifar_ds = cifar_ds.shuffle(buffer_size=1000)
    # 切分数据集到batch_size
    cifar_ds = cifar_ds.batch(batch_size, drop_remainder=True)

    return cifar_ds
```

3.4生成训练数据集

```python
data_path = os.path.join(current_path, 'data/10-batches-bin')
batch_size=32
status="train"

# 生成训练数据集
cifar_ds = get_data(data_path)
ds_train = process_dataset(cifar_ds,batch_size =batch_size, status=status)
```

4.构建网络模型

4.1定义LeNet网络结构，构建网络

```python
"""LeNet."""


def conv(in_channels, out_channels, kernel_size, stride=1, padding=0):
    """weight initial for conv layer"""
    weight = weight_variable()
    return nn.Conv2d(in_channels, out_channels,
                     kernel_size=kernel_size, stride=stride, padding=padding,
                     weight_init=weight, has_bias=False, pad_mode="same")


def fc_with_initialize(input_channels, out_channels):
    """weight initial for fc layer"""
    weight = weight_variable()
    bias = weight_variable()
    return nn.Dense(input_channels, out_channels, weight, bias)


def weight_variable():
    """weight initial"""
    return TruncatedNormal(0.02)


class LeNet5(nn.Cell):
    """
    Lenet network

    Args:
        num_class (int): Num classes. Default: 10.

    Returns:
        Tensor, output tensor
    Examples:
        >>> LeNet(num_class=10)

    """
    def __init__(self, num_class=10, channel=3):
        super(LeNet5, self).__init__()
        self.num_class = num_class
        self.conv1 = conv(channel, 6, 5)
        self.conv2 = conv(6, 16, 5)
        self.fc1 = fc_with_initialize(16 * 8 * 8, 120)
        self.fc2 = fc_with_initialize(120, 84)
        self.fc3 = fc_with_initialize(84, self.num_class)
        self.relu = nn.ReLU()
        self.max_pool2d = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flatten = nn.Flatten()

    def construct(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.max_pool2d(x)
```

```
        x = self.conv2(x)
        x = self.relu(x)
        x = self.max_pool2d(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x
# 构建网络
network = LeNet5(10)
```

5.模型训练与测试

5.1定义损失函数与优化器

```
# 返回当前设备
device_target = mindspore.context.get_context('device_target')
# 确定图模型是否下沉到芯片上
dataset_sink_mode = True if device_target in ['Ascend','GPU'] else False
# 设置模型的设备与图的模式
context.set_context(mode=context.GRAPH_MODE, device_target=device_target)
# 使用交叉熵函数作为损失函数
net_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction="mean")
# 优化器为Adam
net_opt = nn.Adam(params=network.trainable_params(), learning_rate=0.001)
# 监控每个epoch训练的时间
time_cb = TimeMonitor(data_size=ds_train.get_dataset_size())
```

5.2定义保存路径与训练

```
from mindspore.train.callback import Callback

class EvalCallBack(Callback):
    def __init__(self, model, eval_dataset, eval_per_epoch, epoch_per_eval):
        self.model = model
        self.eval_dataset = eval_dataset
        self.eval_per_epoch = eval_per_epoch
        self.epoch_per_eval = epoch_per_eval

    def epoch_end(self, run_context):
        cb_param = run_context.original_args()
        cur_epoch = cb_param.cur_epoch_num
        if cur_epoch % self.eval_per_epoch == 0:
            acc = self.model.eval(self.eval_dataset, dataset_sink_mode=False)
            self.epoch_per_eval["epoch"].append(cur_epoch)
            self.epoch_per_eval["acc"].append(acc["Accuracy"])
            print(acc)
# 设置CheckpointConfig，callback函数。save_checkpoint_steps=训练总数/batch_size
config_ck = CheckpointConfig(save_checkpoint_steps=1562,
                             keep_checkpoint_max=10)
ckpoint_cb = ModelCheckpoint(prefix="checkpoint_lenet_original",
directory='./results',config=config_ck)
```

```python
# 建立可训练模型
model = Model(network = network, loss_fn=net_loss,optimizer=net_opt, metrics={"Accuracy":
Accuracy()})
eval_per_epoch = 1
epoch_per_eval = {"epoch": [], "acc": []}
eval_cb = EvalCallBack(model, ds_train, eval_per_epoch, epoch_per_eval)
print("============== Starting Training ==============")
model.train(20, ds_train,callbacks=[ckpoint_cb,
LossMonitor(per_print_times=1),eval_cb],dataset_sink_mode=dataset_sink_mode)
```

训练过程：

```
============== Starting Training ==============
epoch: 1 step: 1562, loss is 1.490370750427246
{'Accuracy': 0.42751680537772085}
epoch: 2 step: 1562, loss is 1.6508828401565552
{'Accuracy': 0.4936979833546735}
epoch: 3 step: 1562, loss is 1.4105224609375
{'Accuracy': 0.5192261523687580}
epoch: 4 step: 1562, loss is 0.9732769727706909
{'Accuracy': 0.5687219910371318}
epoch: 5 step: 1562, loss is 1.1279035806655884
{'Accuracy': 0.5823463508322664}
epoch: 6 step: 1562, loss is 1.336413145065307б
{'Accuracy': 0.6002320742637645}
epoch: 7 step: 1562, loss is 0.9465760588645935
{'Accuracy': 0.5984915172855314}
epoch: 8 step: 1562, loss is 0.9877745509147644
{'Accuracy': 0.6265805057618438}
epoch: 9 step: 1562, loss is 0.8933621048927307
{'Accuracy': 0.6227792893725992}
epoch: 10 step: 1562, loss is 0.967821478843689
{'Accuracy': 0.6353433098591549}
epoch: 11 step: 1562, loss is 0.5670925974845886
{'Accuracy': 0.6456866197183099}
epoch: 12 step: 1562, loss is 0.613688113052368
{'Accuracy': 0.6543293854033291}
epoch: 13 step: 1562, loss is 1.234076738357544
{'Accuracy': 0.6433858834827144}
epoch: 14 step: 1562, loss is 0.9503560662269592
{'Accuracy': 0.6482274327784892}
epoch: 15 step: 1562, loss is 0.9050345420837402
{'Accuracy': 0.6608514724711908}
epoch: 16 step: 1562, loss is 0.8360009789466858
```

```
{'Accuracy': 0.6613916453265045}
epoch: 17 step: 1562, loss is 0.7438703179359436
{'Accuracy': 0.6638324263764405}
epoch: 18 step: 1562, loss is 1.1150004863739014
{'Accuracy': 0.6745758642765685}
epoch: 19 step: 1562, loss is 1.0351213216781616
{'Accuracy': 0.6774767925736236}
epoch: 20 step: 1562, loss is 0.6592114567756653
{'Accuracy': 0.6877400768245838}
```

5.3设置测试集参数并测试

```
data_path = os.path.join(current_path, 'data/10-verify-bin')
batch_size=32
status="test"
# 生成测试数据集
cifar_ds = ds.Cifar10Dataset(data_path)
ds_eval = process_dataset(cifar_ds,batch_size=batch_size,status=status)

res = model.eval(ds_eval, dataset_sink_mode=dataset_sink_mode)
# 评估测试集
print('test results:',res)
```

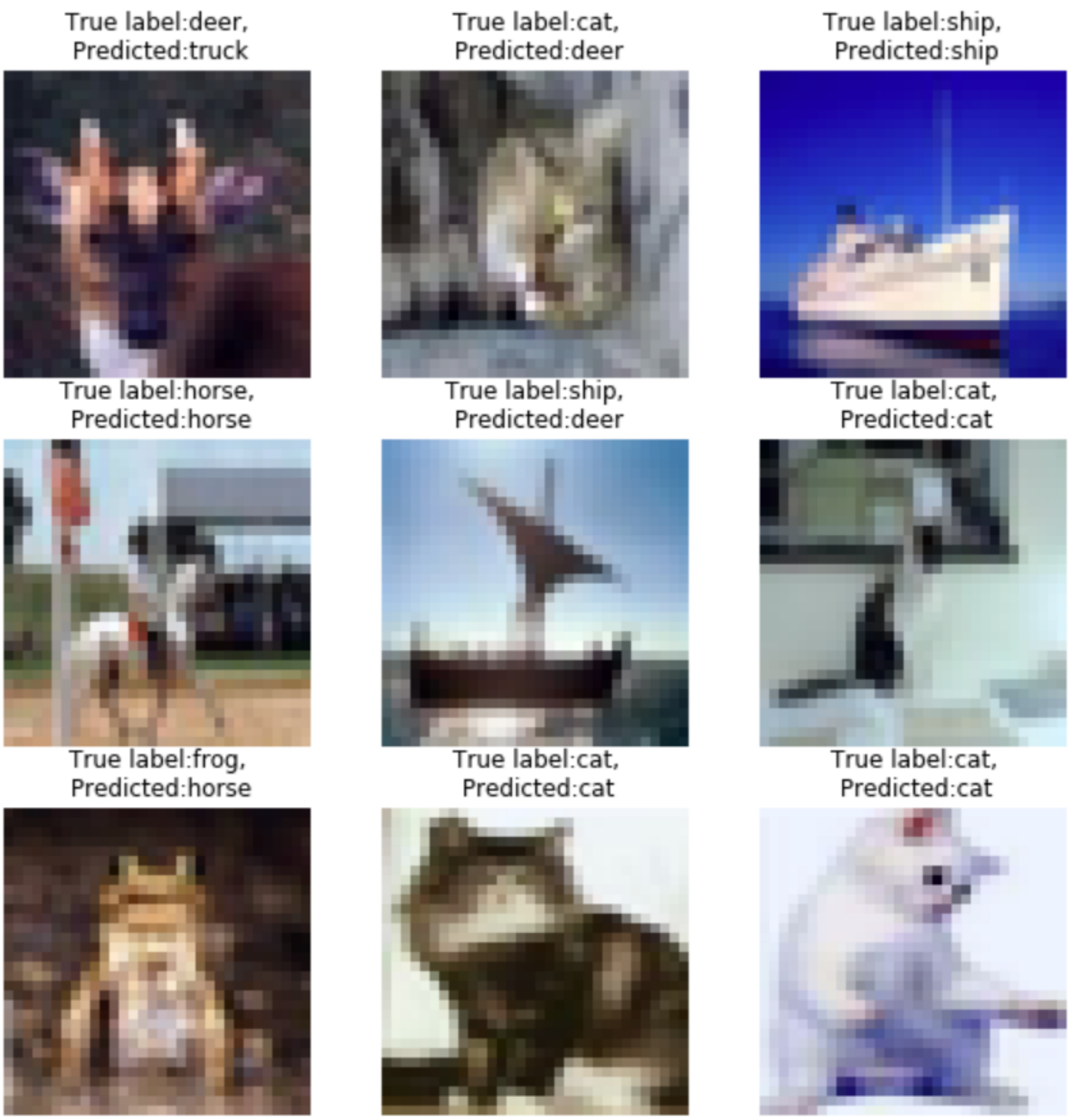
```
test results: {'Accuracy': 0.6913060897435898}
```

5.4图片类别预测与可视化

```
#创建图像标签列表
category_dict = {0:'airplane',1:'automobile',2:'bird',3:'cat',4:'deer',5:'dog',
                 6:'frog',7:'horse',8:'ship',9:'truck'}

cifar_ds = get_data('./data/10-verify-bin')
df_test = process_dataset(cifar_ds,batch_size=1,status='test')

def normalization(data):
    _range = np.max(data) - np.min(data)
    return (data - np.min(data)) / _range

# 设置图像大小
plt.figure(figsize=(10,10))
i = 1
# 打印9张子图
for dic in df_test:
    # 预测单张图片
    input_img = dic[0]
    output = model.predict(Tensor(input_img))
    output = nn.Softmax()(output)
    # 反馈可能性最大的类别
    predicted = np.argmax(output.asnumpy(),axis=1)[0]

    # 可视化
    plt.subplot(3,3,i)
```

```
    # 删除batch维度
    input_image = np.squeeze(input_img.asnumpy(),axis=0).transpose(1,2,0)
    # 重新归一化，方便可视化
    input_image = normalization(input_image)
    plt.imshow(input_image)
    plt.xticks([])
    plt.yticks([])
    plt.axis('off')
    plt.title('True label:%s,\n Predicted:%s'%
(category_dict[dic[1].asnumpy().sum()],category_dict[predicted]))
    i +=1
    if i > 9 :
        break

plt.show()
```

预测结果:



任务二：LeNet-5模型对比

1.调整训练数据和测试数据比例及多少

1.1 训练数据

1.2 测试数据比例

2.训练批次大小，迭代轮数，学习速率，最优化方法等

2.1 训练批次大小

2.2 迭代轮数

2.3 学习速率

2.4 最优化方法

任务三：卷积神经网络模型设计

自己设计的网络：

卷积核从5*5变为3*3；增加两层卷积层，提升模型的非线性映射能力；提升卷积核数量为128；在每一层网络中加入BN层

```
class LeNet5_improve(nn.Cell):
    """
    Lenet network


    Args:
        num_class (int): Num classes. Default: 10.

    Returns:
        Tensor, output tensor
    Examples:
        >>> LeNet(num_class=10)

    """
    def __init__(self, num_class=10, channel=3):
        super(LeNet5_improve, self).__init__()
        self.num_class = num_class
        self.conv1_1 = conv(channel, 12, 3)
        self.bn2_1 = nn.BatchNorm2d(num_features=12)
        self.conv1_2 = conv(12, 24, 3)
        self.bn2_2 = nn.BatchNorm2d(num_features=24)
        self.conv2_1 = conv(24, 48, 3)
        self.bn2_3 = nn.BatchNorm2d(num_features=48)
        self.conv2_2 = conv(48, 96, 3)
        self.bn2_4 = nn.BatchNorm2d(num_features=96)
        self.fc1 = fc_with_initialize(96*8*8, 160)
        self.bn1_1 = nn.BatchNorm1d(num_features=160)
        self.fc2 = fc_with_initialize(160, 120)
        self.bn1_2 = nn.BatchNorm1d(num_features=120)
        self.fc3 = fc_with_initialize(120, self.num_class)
        self.relu = nn.ReLU()
```

```
        self.max_pool2d = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flatten = nn.Flatten()



    def construct(self, x):
        x = self.conv1_1(x)
        x = self.bn2_1(x)
        x = self.relu(x)
        x = self.conv1_2(x)
        x = self.bn2_2(x)
        x = self.relu(x)
        x = self.max_pool2d(x)
        x = self.conv2_1(x)
        x = self.bn2_3(x)
        x = self.relu(x)
        x = self.conv2_2(x)
        x = self.bn2_4(x)
        x = self.relu(x)
        x = self.max_pool2d(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.bn1_1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.bn1_2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x
```

```
ata_path = os.path.join(current_path, 'data/10-batches-bin')
batch_size=32
status="train"

# 生成训练数据集
cifar_ds = get_data(data_path)
ds_train = process_dataset(cifar_ds,batch_size =batch_size, status=status)
network = LeNet5_improve(10)
#network = resnet50(10)
# 返回当前设备
device_target = mindspore.context.get_context('device_target')
# 确定图模型是否下沉到芯片上
dataset_sink_mode = True if device_target in ['Ascend','GPU'] else False
# 设置模型的设备与图的模式
context.set_context(mode=context.GRAPH_MODE, device_target=device_target)
# 使用交叉熵函数作为损失函数
net_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction="mean")
# 优化器为momentum
#net_opt = nn.Momentum(params=network.trainable_params(), learning_rate=0.01, momentum=0.9)
net_opt = nn.Adam(params=network.trainable_params(), learning_rate=0.001)
# 时间监控，反馈每个epoch的运行时间
time_cb = TimeMonitor(data_size=ds_train.get_dataset_size())
# 设置callback函数。
config_ck = CheckpointConfig(save_checkpoint_steps=1562,
                             keep_checkpoint_max=10)
ckpoint_cb = ModelCheckpoint(prefix="checkpoint_lenet_2_verified",directory='./results',
config=config_ck)
```

```
# 建立可训练模型
model = Model(network = network, loss_fn=net_loss,optimizer=net_opt, metrics={"Accuracy":
Accuracy()})
eval_per_epoch = 1
epoch_per_eval = {"epoch": [], "acc": []}
eval_cb = EvalCallBack(model, ds_train, eval_per_epoch, epoch_per_eval)
print("============== Starting Training ==============")

model.train(30, ds_train,callbacks=[ckpoint_cb,
LossMonitor(per_print_times=1),eval_cb],dataset_sink_mode=dataset_sink_mode)
```

训练结果:

```
============== Starting Training ==============
epoch: 1 step: 312, loss is 1.754294753074646
{'Accuracy': 0.4427083333333333}
epoch: 2 step: 312, loss is 1.467523455619812
{'Accuracy': 0.5133213141025641}
epoch: 3 step: 312, loss is 1.0014456510543823
{'Accuracy': 0.5866386217948718}
epoch: 4 step: 312, loss is 1.331483244895935
{'Accuracy': 0.6237980769230769}
epoch: 5 step: 312, loss is 0.9744869470596313
{'Accuracy': 0.64453125}
epoch: 6 step: 312, loss is 1.121692419052124
{'Accuracy': 0.6505408653846154}
epoch: 7 step: 312, loss is 1.0660539865493774
{'Accuracy': 0.67578125}
epoch: 8 step: 312, loss is 0.9659997224807739
{'Accuracy': 0.7022235576923077}
epoch: 9 step: 312, loss is 0.599929690361023
{'Accuracy': 0.6988181089743589}
epoch: 10 step: 312, loss is 0.47205087542533875
{'Accuracy': 0.6818910256410257}
epoch: 11 step: 312, loss is 0.7816133499145508
{'Accuracy': 0.7158453525641025}
epoch: 12 step: 312, loss is 0.7673017978668213
{'Accuracy': 0.7529046474358975}
epoch: 13 step: 312, loss is 0.7378856539726257
{'Accuracy': 0.7626201923076923}
epoch: 14 step: 312, loss is 0.5584795475006104
{'Accuracy': 0.7689302884615384}
epoch: 15 step: 312, loss is 0.8645305633544922
{'Accuracy': 0.784354967948718}
epoch: 16 step: 312, loss is 0.8383089303970337
{'Accuracy': 0.7495993589743589}
epoch: 17 step: 312, loss is 0.507953941822052
{'Accuracy': 0.7983774038461539}
epoch: 18 step: 312, loss is 0.8023295402526855
{'Accuracy': 0.7792467948717948}
epoch: 19 step: 312, loss is 0.5358489751815796
{'Accuracy': 0.8080929487179487}
epoch: 20 step: 312, loss is 0.7109898924827576
{'Accuracy': 0.8130008012820513}
epoch: 21 step: 312, loss is 0.689845085144043
```

```
{'Accuracy': 0.8263221153846154}
epoch: 22 step: 312, loss is 0.5773119330406189
{'Accuracy': 0.8206129807692307}
epoch: 23 step: 312, loss is 0.6138992309570312
{'Accuracy': 0.8356370192307693}
epoch: 24 step: 312, loss is 0.5284755825996399
{'Accuracy': 0.8217147435897436}
epoch: 25 step: 312, loss is 0.33693888783454895
{'Accuracy': 0.8346354166666666}
epoch: 26 step: 312, loss is 0.44585806131362915
{'Accuracy': 0.8439503205128205}
epoch: 27 step: 312, loss is 0.6587333679199219
{'Accuracy': 0.8370392628205128}
epoch: 28 step: 312, loss is 0.21484141051769257
{'Accuracy': 0.8424479166666666}
epoch: 29 step: 312, loss is 0.39074209332466125
{'Accuracy': 0.8662860576923077}
epoch: 30 step: 312, loss is 0.46585550904273987
{'Accuracy': 0.8547676282051282}
```

```python
data_path = os.path.join(current_path, 'data/10-verify-bin')
batch_size=32
status="test"
# 生成测试数据集
cifar_ds = ds.Cifar10Dataset(data_path)
ds_eval = process_dataset(cifar_ds,batch_size=batch_size,status=status)

res = model.eval(ds_eval, dataset_sink_mode=dataset_sink_mode)
# 评估测试集
print('test results:',res)
```

```
test results: {'Accuracy': 0.9072516025641025}
```

任务二：LeNet-5模型对比

在任务二中，通过改变训练数据的多少、训练批次、迭代轮数、学习速率等与任务一中的结果进行比对。

下图是任务一中的训练结果：

```
============== Starting Training ==============
epoch: 1 step: 312, loss is 1.754294753074646
{'Accuracy': 0.4427083333333333}
epoch: 2 step: 312, loss is 1.467523455619812
{'Accuracy': 0.513321314102564l}
epoch: 3 step: 312, loss is 1.0014456510543823
{'Accuracy': 0.5866386217948718}
epoch: 4 step: 312, loss is 1.331483244895935
{'Accuracy': 0.6237980769230769}
epoch: 5 step: 312, loss is 0.9744869470596313
{'Accuracy': 0.64453125}
epoch: 6 step: 312, loss is 1.121692419052124
{'Accuracy': 0.6505408653846154}
epoch: 7 step: 312, loss is 1.0660539865493774
{'Accuracy': 0.67578125}
epoch: 8 step: 312, loss is 0.9659997224807739
{'Accuracy': 0.7022235576923077}
epoch: 9 step: 312, loss is 0.599929690361023
{'Accuracy': 0.6988181089743589}
epoch: 10 step: 312, loss is 0.47205087542533875
{'Accuracy': 0.6818910256410257}
epoch: 11 step: 312, loss is 0.7816133499145508
{'Accuracy': 0.7158453525641025}
epoch: 12 step: 312, loss is 0.7673017978668213
{'Accuracy': 0.7529046474358975}
epoch: 13 step: 312, loss is 0.7378856539726257
{'Accuracy': 0.7626201923076923}
epoch: 14 step: 312, loss is 0.5584795475006104
{'Accuracy': 0.7689302884615384}
epoch: 15 step: 312, loss is 0.8645305633544922
{'Accuracy': 0.784354967948718}
epoch: 16 step: 312, loss is 0.8383089303970337
{'Accuracy': 0.7495993589743589}
epoch: 17 step: 312, loss is 0.507953941822052
{'Accuracy': 0.7983774038461539}
epoch: 18 step: 312, loss is 0.8023295402526855
{'Accuracy': 0.7792467948717948}
epoch: 19 step: 312, loss is 0.5358489751815796
{'Accuracy': 0.8080929487179487}
epoch: 20 step: 312, loss is 0.710989892482757б
{'Accuracy': 0.8130008012820513}
epoch: 21 step: 312, loss is 0.689845085144043
```

```
{'Accuracy': 0.8263221153846154}
epoch: 22 step: 312, loss is 0.5773119330406189
{'Accuracy': 0.8206129807692307}
epoch: 23 step: 312, loss is 0.6138992309570312
{'Accuracy': 0.8356370192307693}
epoch: 24 step: 312, loss is 0.5284755825996399
{'Accuracy': 0.8217147435897436}
epoch: 25 step: 312, loss is 0.33693888783454895
{'Accuracy': 0.8346354166666666}
epoch: 26 step: 312, loss is 0.44585806131362915
{'Accuracy': 0.8439503205128205}
epoch: 27 step: 312, loss is 0.6587333679199219
{'Accuracy': 0.8370392628205128}
epoch: 28 step: 312, loss is 0.21484141051769257
{'Accuracy': 0.8424479166666666}
epoch: 29 step: 312, loss is 0.39074209332466125
{'Accuracy': 0.8662860576923077}
epoch: 30 step: 312, loss is 0.46585550904273987
{'Accuracy': 0.8547676282051282}
```

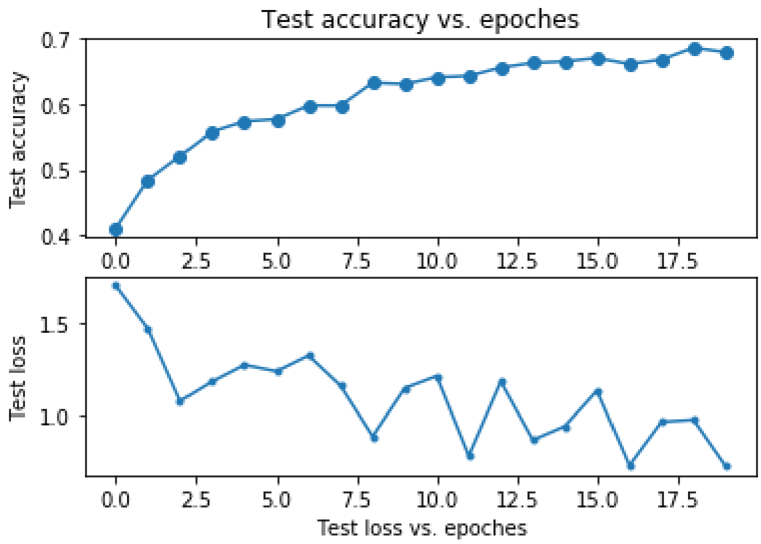
1.训练批次大小，迭代轮数，学习速率，最优化方法等

这是任务一中模型的训练结果：

1.1 改变训练批次大小

将batch_size从 32 改为 64，结果：

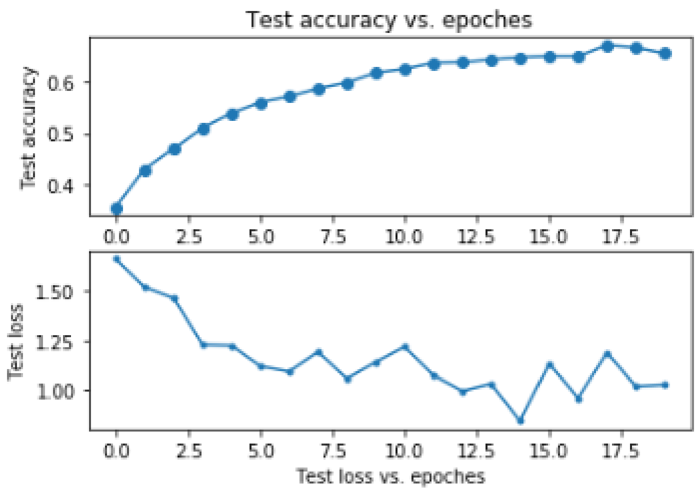
```
============== Starting Training ==============
epoch: 1 step: 781, loss is 1.7090344429016113
{'Accuracy': 0.41099151728553135}
epoch: 2 step: 781, loss is 1.4728960990905762
{'Accuracy': 0.48469510243277847}
epoch: 3 step: 781, loss is 1.0774487257003784
{'Accuracy': 0.5213068181818182}
epoch: 4 step: 781, loss is 1.1843929290771484
{'Accuracy': 0.558258642765685}
epoch: 5 step: 781, loss is 1.2746100425720215
{'Accuracy': 0.5737235915492958}
epoch: 6 step: 781, loss is 1.2396060228347778
{'Accuracy': 0.5769646286811779}
epoch: 7 step: 781, loss is 1.3256103992462158
{'Accuracy': 0.5978913252240717}
epoch: 8 step: 781, loss is 1.1607106924057007
{'Accuracy': 0.6220590588988476}
epoch: 9 step: 781, loss is 0.8843802213668823
{'Accuracy': 0.6328225032010243}
epoch: 10 step: 781, loss is 1.1479308605194092
{'Accuracy': 0.6308618758002561}
epoch: 11 step: 781, loss is 1.213714838027954
{'Accuracy': 0.6410851472471191}
epoch: 12 step: 781, loss is 0.7778354287147522
{'Accuracy': 0.6432258322663252}
epoch: 13 step: 781, loss is 1.1842436790466309
{'Accuracy': 0.6560499359795134}
epoch: 14 step: 781, loss is 0.8643943071365356
{'Accuracy': 0.6632322343149808}
epoch: 15 step: 781, loss is 0.9397111535072327
{'Accuracy': 0.665312900128041}
epoch: 16 step: 781, loss is 1.1341917514801025
{'Accuracy': 0.6707146286811779}
epoch: 17 step: 781, loss is 0.7263000011444092
{'Accuracy': 0.6613516325224071}
epoch: 18 step: 781, loss is 0.963616132736206
{'Accuracy': 0.668213828425096}
epoch: 19 step: 781, loss is 0.9746869206428528
{'Accuracy': 0.6862996158770807}
epoch: 20 step: 781, loss is 0.7225057482719421
{'Accuracy': 0.6795174455825864}
```




```
test results: {'Accuracy': 0.6761818910256411}
```

将batch_size再改为128，结果：

```
============== Starting Training ==============
epoch: 1 step: 390, loss is 1.6629327535629272
{'Accuracy': 0.3554487179487l796}
epoch: 2 step: 390, loss is 1.520633578300476
{'Accuracy': 0.42978766025641024}
epoch: 3 step: 390, loss is 1.4648629426956177
{'Accuracy': 0.4701522435897436}
epoch: 4 step: 390, loss is 1.2270786762237549
{'Accuracy': 0.5102363782051282}
epoch: 5 step: 390, loss is 1.2240411043167114
{'Accuracy': 0.5385817307692308}
epoch: 6 step: 390, loss is 1.1194146871566772
{'Accuracy': 0.5600360576923077}
epoch: 7 step: 390, loss is 1.0917320251464844
{'Accuracy': 0.5713942307692308}
epoch: 8 step: 390, loss is 1.191634178161621
{'Accuracy': 0.5870592948717949}
epoch: 9 step: 390, loss is 1.057187795639038
{'Accuracy': 0.5984975961538461}
epoch: 10 step: 390, loss is 1.1397819519042969
{'Accuracy': 0.61796875}
epoch: 11 step: 390, loss is 1.2169818878173828
{'Accuracy': 0.6253605769230769}
epoch: 12 step: 390, loss is 1.0742069482803345
{'Accuracy': 0.6372395833333333}
epoch: 13 step: 390, loss is 0.9935532808303833
{'Accuracy': 0.6383613782051282}
epoch: 14 step: 390, loss is 1.0304205417633057
{'Accuracy': 0.6436899038461539}
epoch: 15 step: 390, loss is 0.8439756631851196
{'Accuracy': 0.6482972756410257}
epoch: 16 step: 390, loss is 1.1316914558410645
{'Accuracy': 0.6503405448717948}
epoch: 17 step: 390, loss is 0.9597344398498535
{'Accuracy': 0.6502604166666667}
epoch: 18 step: 390, loss is 1.1873278617858887
{'Accuracy': 0.6717548076923077}
epoch: 19 step: 390, loss is 1.0174975395202637
{'Accuracy': 0.6671274038461539}
epoch: 20 step: 390, loss is 1.0253665447235107
{'Accuracy': 0.6557291666666667}
```

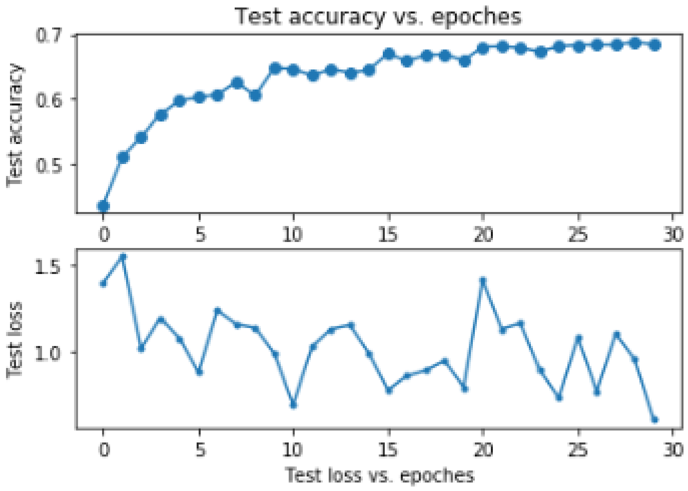



```
test results: {'Accuracy': 0.6656650641025641}
```

1.2 改变迭代轮数

将epoch从20改为30，结果：

```
============== Starting Training ==============
epoch: 1 step: 1562, loss is 1.4001972675323486
{'Accuracy': 0.4373199423815621}
epoch: 2 step: 1562, loss is 1.5505421161651611
{'Accuracy': 0.5109635083226632}
epoch: 3 step: 1562, loss is 1.0231988430023193
{'Accuracy': 0.542173495518566}
epoch: 4 step: 1562, loss is 1.196373701095581
{'Accuracy': 0.5763044174135723}
epoch: 5 step: 1562, loss is 1.081119418144226
{'Accuracy': 0.5975912291933418}
epoch: 6 step: 1562, loss is 0.8857177495956421
{'Accuracy': 0.6025128040973111}
epoch: 7 step: 1562, loss is 1.241542100906372
{'Accuracy': 0.6070742637644047}
epoch: 8 step: 1562, loss is 1.1615225076675415
{'Accuracy': 0.6260203265044815}
epoch: 9 step: 1562, loss is 1.1411819458007812
{'Accuracy': 0.6053337067861716}
epoch: 10 step: 1562, loss is 0.9944120645523071
{'Accuracy': 0.647827304737516}
epoch: 11 step: 1562, loss is 0.7024457454681396
{'Accuracy': 0.646606914212548}
epoch: 12 step: 1562, loss is 1.0324710607528687
{'Accuracy': 0.6363236235595391}
epoch: 13 step: 1562, loss is 1.1341512203216553
{'Accuracy': 0.6463468309859155}
epoch: 14 step: 1562, loss is 1.1569676399230957
{'Accuracy': 0.6407250320102432}
epoch: 15 step: 1562, loss is 0.9935476779937744
{'Accuracy': 0.6447863316261203}
epoch: 16 step: 1562, loss is 0.7857153415679932
{'Accuracy': 0.6701544494238156}
epoch: 17 step: 1562, loss is 0.865190327167511
{'Accuracy': 0.6587908130601793}
epoch: 18 step: 1562, loss is 0.8974545001983643
{'Accuracy': 0.6675936299615877}
epoch: 19 step: 1562, loss is 0.9522240161895752
{'Accuracy': 0.6691141165172856}
epoch: 20 step: 1562, loss is 0.8008358478546143
{'Accuracy': 0.659330985915493}
epoch: 21 step: 1562, loss is 1.4165325164794922
{'Accuracy': 0.6805177656850192}
epoch: 22 step: 1562, loss is 1.1342244148254395
{'Accuracy': 0.6811379641485276}
epoch: 23 step: 1562, loss is 1.1670312881469727
{'Accuracy': 0.6790572983354674}
epoch: 24 step: 1562, loss is 0.8984321355819702
{'Accuracy': 0.673235435339308 6}
epoch: 25 step: 1562, loss is 0.7403361201286316
{'Accuracy': 0.6814180537772087}
epoch: 26 step: 1562, loss is 1.0863641500473022
{'Accuracy': 0.683318661971831}
epoch: 27 step: 1562, loss is 0.7768865823745728
{'Accuracy': 0.6835187259923176}
epoch: 28 step: 1562, loss is 1.1063823699951172
{'Accuracy': 0.6902408770806658}
epoch: 29 step: 1562, loss is 0.9597190022468567
{'Accuracy': 0.6879601472471191}
epoch: 30 step: 1562, loss is 0.6170095205307007
{'Accuracy': 0.6840188860435339}
```




```
test results: {'Accuracy': 0.6913060897435898}
```

将epoch从30改为40，结果：

```
============== Starting Training ==============
epoch: 1 step: 1562, loss is 1.6681767702102661
{'Accuracy': 0.4368798015364917}
epoch: 2 step: 1562, loss is 1.5303082466125488
{'Accuracy': 0.48619558258642764}
epoch: 3 step: 1562, loss is 1.3278591632843018
{'Accuracy': 0.53020966709347}
epoch: 4 step: 1562, loss is 1.397017240524292
{'Accuracy': 0.5527768886043534}
epoch: 5 step: 1562, loss is 1.0204720497131348
{'Accuracy': 0.5826464468629962}
epoch: 6 step: 1562, loss is 1.1125963926315308
{'Accuracy': 0.5932098271446863}
epoch: 7 step: 1562, loss is 1.2506515979766846
{'Accuracy': 0.611235595390525}
epoch: 8 step: 1562, loss is 1.2103956937789917
{'Accuracy': 0.6187780089628682}
epoch: 9 step: 1562, loss is 0.8361906409263611
{'Accuracy': 0.6325824263764405}
epoch: 10 step: 1562, loss is 1.0395419597625732
{'Accuracy': 0.6366237195902689}
epoch: 11 step: 1562, loss is 1.0460056066513062
{'Accuracy': 0.6402248719590269}
epoch: 12 step: 1562, loss is 0.8070405721664429
{'Accuracy': 0.6299615877080665}
epoch: 13 step: 1562, loss is 0.9855949878692627
{'Accuracy': 0.6517485595390525}
epoch: 14 step: 1562, loss is 1.0698835849761963
{'Accuracy': 0.6551896606914213}
epoch: 15 step: 1562, loss is 1.5145652294158936
{'Accuracy': 0.6566301216389244}
epoch: 16 step: 1562, loss is 0.9458028078079224
{'Accuracy': 0.6405449743918054}
epoch: 17 step: 1562, loss is 1.1336663961410522
{'Accuracy': 0.652648847631242}
epoch: 18 step: 1562, loss is 1.1910443305969238
{'Accuracy': 0.6679137323943662}
epoch: 19 step: 1562, loss is 0.8705137968063354
{'Accuracy': 0.671114756722151}
epoch: 20 step: 1562, loss is 0.9497054219245911
{'Accuracy': 0.6675336107554417}
epoch: 21 step: 1562, loss is 1.1559308767318726
{'Accuracy': 0.6670334507042254}
epoch: 22 step: 1562, loss is 0.846859335899353
{'Accuracy': 0.6672735275288092}
epoch: 23 step: 1562, loss is 0.7861065864562988
{'Accuracy': 0.6825984314980794}
epoch: 24 step: 1562, loss is 0.9432480931282043
{'Accuracy': 0.6722151088348272}
epoch: 25 step: 1562, loss is 1.1506741046905518
{'Accuracy': 0.6878000960307298}
epoch: 26 step: 1562, loss is 0.9197145104408264
{'Accuracy': 0.6772967349551856}
epoch: 27 step: 1562, loss is 1.22108447551727 3
{'Accuracy': 0.683218629961587 7}
epoch: 28 step: 1562, loss is 1.3810433149337769
{'Accuracy': 0.6770566581306018}
epoch: 29 step: 1562, loss is 0.6915632486343384
{'Accuracy': 0.6872799295774648}
epoch: 30 step: 1562, loss is 0.778640627861023
{'Accuracy': 0.6831786171574904}
epoch: 31 step: 1562, loss is 1.0036625862121582
```

```
epoch: 31 step: 1562, loss is 1.[illegible]
{'Accuracy': 0.6858794814340589}
epoch: 32 step: 1562, loss is 0.9554892778396606
{'Accuracy': 0.6866397247119078}
epoch: 33 step: 1562, loss is 0.8396035432815552
{'Accuracy': 0.6991837387964148}
epoch: 34 step: 1562, loss is 0.8455663919448853
{'Accuracy': 0.6957626440460948}
epoch: 35 step: 1562, loss is 0.5509024858474731
{'Accuracy': 0.6937419974391805}
epoch: 36 step: 1562, loss is 0.9377868175506592
{'Accuracy': 0.6963628361075545}
epoch: 37 step: 1562, loss is 0.9480979442596436
{'Accuracy': 0.6922415172855314}
epoch: 38 step: 1562, loss is 0.7274941205978394
{'Accuracy': 0.7014244558258643}
epoch: 39 step: 1562, loss is 0.7659074068069458
{'Accuracy': 0.6993037772087067}
epoch: 40 step: 1562, loss is 0.7372831702232361
{'Accuracy': 0.700224071702945}
```

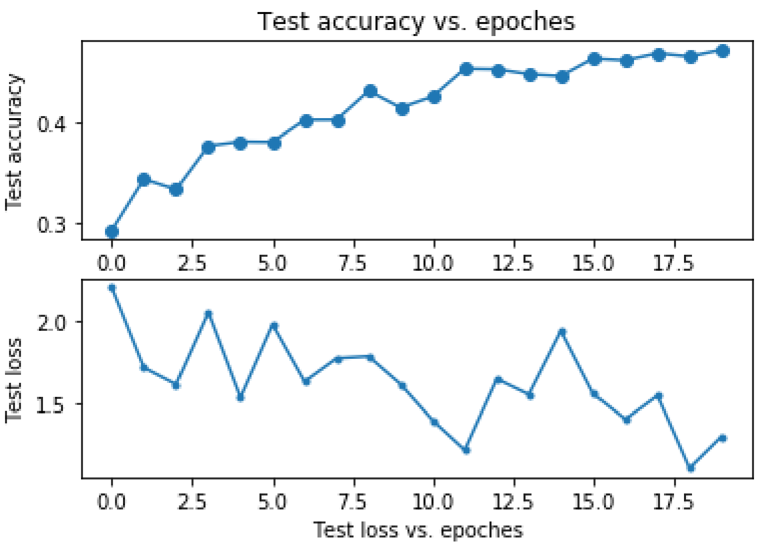
```
[10] data_path = os.path.join(current_path, 'data/10-verify-b
```

```
test results: {'Accuracy': 0.7030248397435898}
```

1.3 改变学习速率

learning_rate从0.001改为0.005：

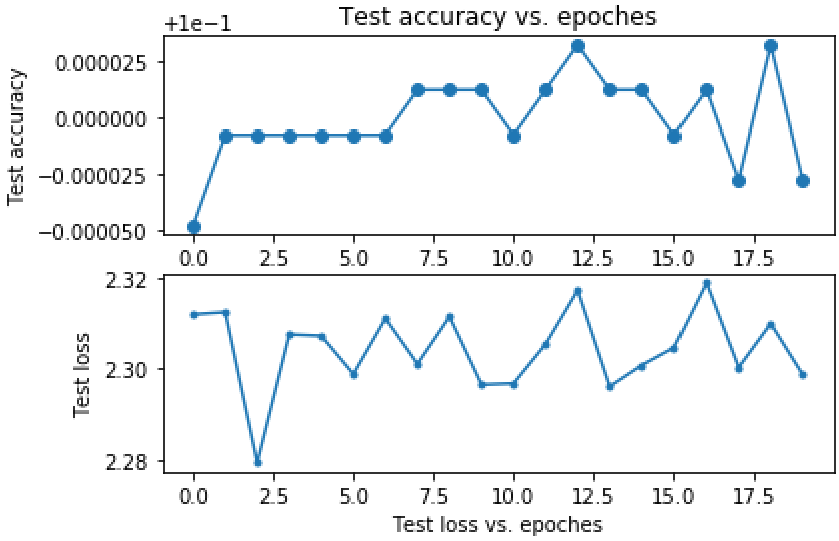
```
============== Starting Training ==============
epoch: 1 step: 1562, loss is 2.1959691047668457
{'Accuracy': 0.29207346350832264}
epoch: 2 step: 1562, loss is 1.717069387435913
{'Accuracy': 0.34375}
epoch: 3 step: 1562, loss is 1.6178371906280518
{'Accuracy': 0.3334467029449424}
epoch: 4 step: 1562, loss is 2.048318386077881
{'Accuracy': 0.37732074263764404}
epoch: 5 step: 1562, loss is 1.5381343364715576
{'Accuracy': 0.38150208066581304}
epoch: 6 step: 1562, loss is 1.9760026931762695
{'Accuracy': 0.3811219590268886}
epoch: 7 step: 1562, loss is 1.6335136890411377
{'Accuracy': 0.4034090909090909}
epoch: 8 step: 1562, loss is 1.773061752319336
{'Accuracy': 0.40348911651728553}
epoch: 9 step: 1562, loss is 1.784604787826538
{'Accuracy': 0.4326584507042254}
epoch: 10 step: 1562, loss is 1.617903709411621
{'Accuracy': 0.41571302816901406}
epoch: 11 step: 1562, loss is 1.4019289016723633
{'Accuracy': 0.4269766325224072}
epoch: 12 step: 1562, loss is 1.2251296043395996
{'Accuracy': 0.4551456466069142}
epoch: 13 step: 1562, loss is 1.650319218635559
{'Accuracy': 0.45458546734955185}
epoch: 14 step: 1562, loss is 1.5573270320892334
{'Accuracy': 0.4496638924455826}
epoch: 15 step: 1562, loss is 1.9376988410949707
{'Accuracy': 0.447663252240717}
epoch: 16 step: 1562, loss is 1.5618650913238525
{'Accuracy': 0.46538892445582586}
epoch: 17 step: 1562, loss is 1.408649206161499
{'Accuracy': 0.46360835467349554}
epoch: 18 step: 1562, loss is 1.55056631565094
{'Accuracy': 0.47085067221510885}
epoch: 19 step: 1562, loss is 1.1176893711090088
{'Accuracy': 0.46746959026888607}
epoch: 20 step: 1562, loss is 1.3065859079360962
{'Accuracy': 0.4742917733674776}
```




```
test results: {'Accuracy': 0.5078125}
```

learning_rate从0.005改为0.01：（learning_rate = 0.01的时候loss很大且一直降不下来，accuracy一直非常低，说明此时learning_rate过大，模型无法学到有用的特征，已经没有必要再增加learning_rate了）

```
============== Starting Training ==============
epoch: 1 step: 1562, loss is 2.3119072914123535
{'Accuracy': 0.09995198463508323}
epoch: 2 step: 1562, loss is 2.312417984008789
{'Accuracy': 0.09999199743918054}
epoch: 3 step: 1562, loss is 2.2794549465179443
{'Accuracy': 0.09999199743918054}
epoch: 4 step: 1562, loss is 2.3075382709503174
{'Accuracy': 0.09999199743918054}
epoch: 5 step: 1562, loss is 2.3072025775909424
{'Accuracy': 0.09999199743918054}
epoch: 6 step: 1562, loss is 2.2988624572753906
{'Accuracy': 0.09999199743918054}
epoch: 7 step: 1562, loss is 2.311034679412842
{'Accuracy': 0.09999199743918054}
epoch: 8 step: 1562, loss is 2.301069736480713
{'Accuracy': 0.1000120038412292}
epoch: 9 step: 1562, loss is 2.311424493789673
{'Accuracy': 0.1000120038412292}
epoch: 10 step: 1562, loss is 2.2965474128723145
{'Accuracy': 0.1000120038412292}
epoch: 11 step: 1562, loss is 2.2968099117279053
{'Accuracy': 0.09999199743918054}
epoch: 12 step: 1562, loss is 2.305422782897949
{'Accuracy': 0.1000120038412292}
epoch: 13 step: 1562, loss is 2.3172011375427246
{'Accuracy': 0.10003201024327785}
epoch: 14 step: 1562, loss is 2.2961347103118896
{'Accuracy': 0.1000120038412292}
epoch: 15 step: 1562, loss is 2.3007993698120117
{'Accuracy': 0.1000120038412292}
epoch: 16 step: 1562, loss is 2.304628372192383
{'Accuracy': 0.09999199743918054}
epoch: 17 step: 1562, loss is 2.3188090324401855
{'Accuracy': 0.1000120038412292}
epoch: 18 step: 1562, loss is 2.3003153800964355
{'Accuracy': 0.09997199103713188}
epoch: 19 step: 1562, loss is 2.3099145889282227
{'Accuracy': 0.10003201024327785}
epoch: 20 step: 1562, loss is 2.298717975616455
{'Accuracy': 0.09997199103713188}
```




```
test results: {'Accuracy': 0.10016025641025642}
```

1.4 改变最优化方法

最优化方法设置为momentum结果：

```
============== Starting Training ==============
epoch: 1 step: 1562, loss is 2.303001880645752
{'Accuracy': 0.10003201024327785}
epoch: 2 step: 1562, loss is 2.3037257194519043
{'Accuracy': 0.1000120038412292}
epoch: 3 step: 1562, loss is 2.301295280456543
{'Accuracy': 0.09995198463508323}
epoch: 4 step: 1562, loss is 2.3020517826080322
{'Accuracy': 0.1492477592829705 4}
epoch: 5 step: 1562, loss is 2.1639244556427
{'Accuracy': 0.20788652368758004}
epoch: 6 step: 1562, loss is 1.8455278873443604
{'Accuracy': 0.28223031370038415}
epoch: 7 step: 1562, loss is 1.622389793395996
{'Accuracy': 0.34699103713188223}
epoch: 8 step: 1562, loss is 1.3402007818222046
{'Accuracy': 0.3941461267605634}
epoch: 9 step: 1562, loss is 1.2828444242477417
{'Accuracy': 0.4364796734955186}
epoch: 10 step: 1562, loss is 1.501384973526001
{'Accuracy': 0.45454545454545453}
epoch: 11 step: 1562, loss is 1.284022331237793
{'Accuracy': 0.4838348271446863}
epoch: 12 step: 1562, loss is 1.1977553367614746
{'Accuracy': 0.5050016005121639}
epoch: 13 step: 1562, loss is 1.2376580238342285
{'Accuracy': 0.5211067541613317}
epoch: 14 step: 1562, loss is 1.1963856220245361
{'Accuracy': 0.5464348591549296}
epoch: 15 step: 1562, loss is 1.0246236324310303
{'Accuracy': 0.5631602112676056}
epoch: 16 step: 1562, loss is 1.100939154624939
{'Accuracy': 0.5713028169014085}
epoch: 17 step: 1562, loss is 1.1708446741104126
{'Accuracy': 0.5784250960307298}
epoch: 18 step: 1562, loss is 1.2316393852233887
{'Accuracy': 0.6023927656850192}
epoch: 19 step: 1562, loss is 1.3428165912628174
{'Accuracy': 0.6077344750320103}
epoch: 20 step: 1562, loss is 1.3183668851852417
{'Accuracy': 0.6170374519846351}
```

测试集上结果:

```
test results: {'Accuracy': 0.6294070512820513}
```

效果没有阿达姆好，但相差并不大。

任务三：卷积神经网络模型设计

改进的神经网络LeNet5_improve（算是比较成功吧，测试的acc有0.9）

所有的卷积核从5*5变成3*3。

增加了两层卷积层，提升模型的非线性映射能力。

提升了卷积核数量，以及linear层的输出width，使模型可以提取更多的特征（12 -> 24 -> 48 -> 96 -> 160 -> 120）。

在每一层网络层中加入 BN（批归一化）层。

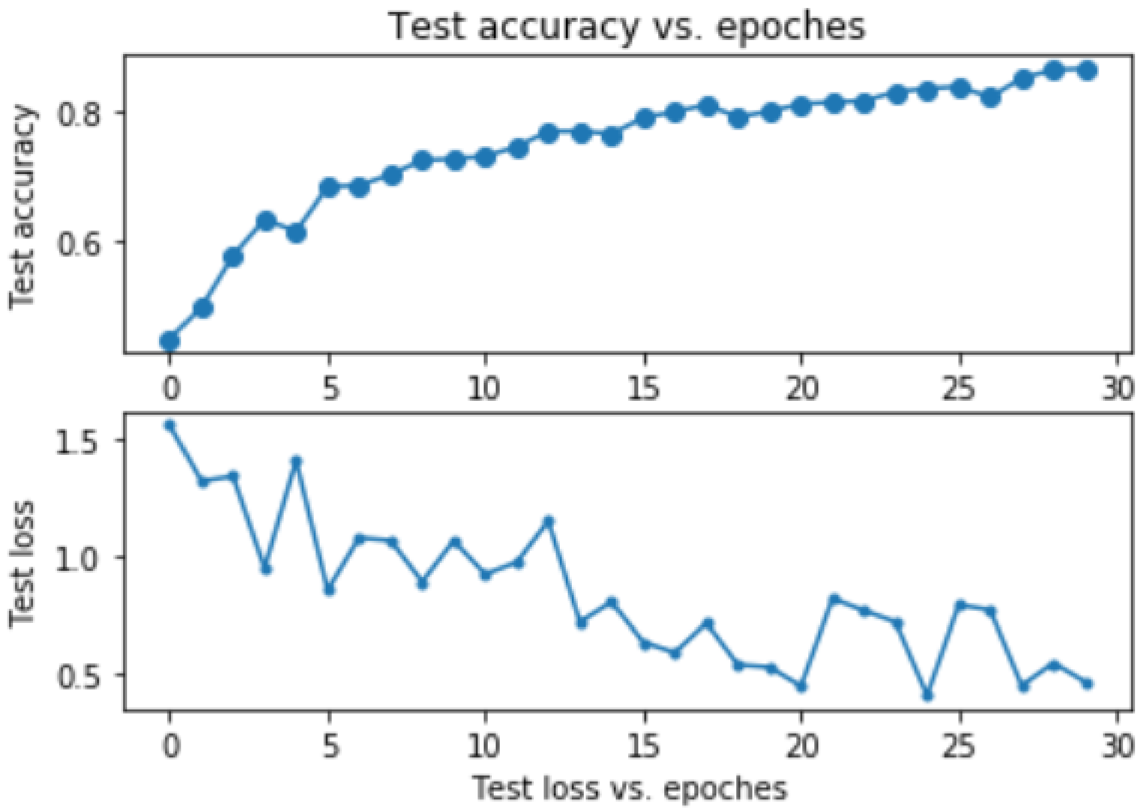
```
class LeNet5_improve(nn.Cell):
    """
    Lenet network


    Args:
        num_class (int): Num classes. Default: 10.

    Returns:
        Tensor, output tensor
    Examples:
        >>> LeNet(num_class=10)

    """
    def __init__(self, num_class=10, channel=3):
        super(LeNet5_improve, self).__init__()
        self.num_class = num_class
        self.conv1_1 = conv(channel, 12, 3)
        self.bn2_1 = nn.BatchNorm2d(num_features=12)
        self.conv1_2 = conv(12, 24, 3)
        self.bn2_2 = nn.BatchNorm2d(num_features=24)
        self.conv2_1 = conv(24, 48, 3)
        self.bn2_3 = nn.BatchNorm2d(num_features=48)
        self.conv2_2 = conv(48, 96, 3)
        self.bn2_4 = nn.BatchNorm2d(num_features=96)
        self.fc1 = fc_with_initialize(96*8*8, 160)
        self.bn1_1 = nn.BatchNorm1d(num_features=160)
        self.fc2 = fc_with_initialize(160, 120)
        self.bn1_2 = nn.BatchNorm1d(num_features=120)
        self.fc3 = fc_with_initialize(120, self.num_class)
        self.relu = nn.ReLU()
        self.max_pool2d = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flatten = nn.Flatten()



    def construct(self, x):
        x = self.conv1_1(x)
        x = self.bn2_1(x)
        x = self.relu(x)
        x = self.conv1_2(x)
        x = self.bn2_2(x)
        x = self.relu(x)
        x = self.max_pool2d(x)
```

```
        x = self.conv2_1(x)
        x = self.bn2_3(x)
        x = self.relu(x)
        x = self.conv2_2(x)
        x = self.bn2_4(x)
        x = self.relu(x)
        x = self.max_pool2d(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.bn1_1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.bn1_2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x
```

```
ata_path = os.path.join(current_path, 'data/10-batches-bin')
batch_size=32
status="train"

# train_loss = []
# train_accuracy = []

# 生成训练数据集
cifar_ds = get_data(data_path)
ds_train = process_dataset(cifar_ds,batch_size =batch_size, status=status)
network = LeNet5_improve(10)
#network = resnet50(10)
# 返回当前设备
device_target = mindspore.context.get_context('device_target')
# 确定图模型是否下沉到芯片上
dataset_sink_mode = True if device_target in ['Ascend','GPU'] else False
# 设置模型的设备与图的模式
context.set_context(mode=context.GRAPH_MODE, device_target=device_target)
# 使用交叉熵函数作为损失函数
net_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction="mean")
# 优化器为momentum
#net_opt = nn.Momentum(params=network.trainable_params(), learning_rate=0.01, momentum=0.9)
net_opt = nn.Adam(params=network.trainable_params(), learning_rate=0.001)
# 时间监控，反馈每个epoch的运行时间
time_cb = TimeMonitor(data_size=ds_train.get_dataset_size())
# 设置callback函数。
config_ck = CheckpointConfig(save_checkpoint_steps=1562,
                             keep_checkpoint_max=10)
ckpoint_cb = ModelCheckpoint(prefix="checkpoint_lenet_2_verified",directory='./results',
config=config_ck)
# 建立可训练模型
model = Model(network = network, loss_fn=net_loss,optimizer=net_opt, metrics={"Accuracy":
Accuracy()})
eval_per_epoch = 1
epoch_per_eval = {"epoch": [], "acc": []}
eval_cb = EvalCallBack(model, ds_train, eval_per_epoch, epoch_per_eval)
print("============== Starting Training ==============")

model.train(30, ds_train,callbacks=[ckpoint_cb,
LossMonitor(per_print_times=1),eval_cb],dataset_sink_mode=dataset_sink_mode)
```

训练结果（损失值和精确率的变化）：

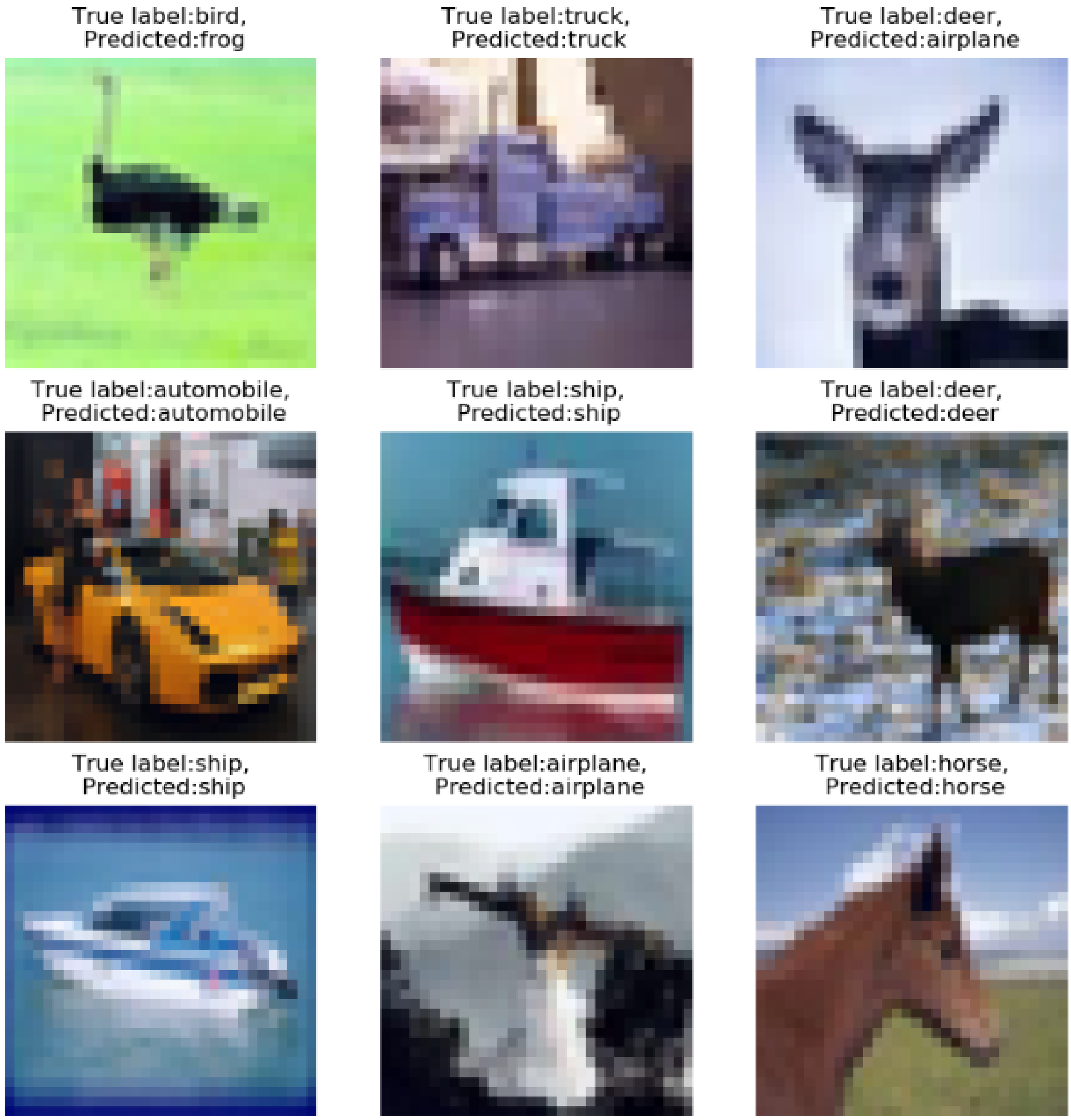
```
============== Starting Training ==============
epoch: 1 step: 312, loss is 1.5620884895324707
{'Accuracy': 0.4475160256410256}
epoch: 2 step: 312, loss is 1.3239355087280273
{'Accuracy': 0.4961939102564l024}
epoch: 3 step: 312, loss is 1.3438165187835693
{'Accuracy': 0.5765224358974359}
epoch: 4 step: 312, loss is 0.9501688480377197
{'Accuracy': 0.632011217948718}
epoch: 5 step: 312, loss is 1.4156862497329712
{'Accuracy': 0.6137820512820513}
epoch: 6 step: 312, loss is 0.8565905094146729
{'Accuracy': 0.6820913461538461}
epoch: 7 step: 312, loss is 1.0814460515975952
{'Accuracy': 0.6849959935897436}
epoch: 8 step: 312, loss is 1.068793773651123
{'Accuracy': 0.7019230769230769}
epoch: 9 step: 312, loss is 0.8926912546157837
{'Accuracy': 0.723457532051282}
epoch: 10 step: 312, loss is 1.0673104524612427
{'Accuracy': 0.7256610576923077}
epoch: 11 step: 312, loss is 0.9242600202560425
{'Accuracy': 0.7297676282051282}
epoch: 12 step: 312, loss is 0.9753320217132568
{'Accuracy': 0.7442908653846154}
epoch: 13 step: 312, loss is 1.1511681079864502
{'Accuracy': 0.7683293269230769}
epoch: 14 step: 312, loss is 0.720867395401001
{'Accuracy': 0.7685296474358975}
epoch: 15 step: 312, loss is 0.8044866323471069
{'Accuracy': 0.7645232371794872}
epoch: 16 step: 312, loss is 0.6349259614944458
{'Accuracy': 0.7896634615384616}
epoch: 17 step: 312, loss is 0.5902111530303955
{'Accuracy': 0.7984775641025641}
epoch: 18 step: 312, loss is 0.7167503237724304
{'Accuracy': 0.8098958333333334}
epoch: 19 step: 312, loss is 0.5388635396957397
{'Accuracy': 0.7902644230769231}
epoch: 20 step: 312, loss is 0.5280959606170654
{'Accuracy': 0.7998798076923077}
epoch: 21 step: 312, loss is 0.44707363843917847
{'Accuracy': 0.8102964743589743}
epoch: 22 step: 312, loss is 0.8202043175697327
{'Accuracy': 0.8136017628205128}
epoch: 23 step: 312, loss is 0.7687522768974304
{'Accuracy': 0.8152043269230769}
epoch: 24 step: 312, loss is 0.7202112674713135
{'Accuracy': 0.8298277243589743}
epoch: 25 step: 312, loss is 0.40762272477149963
{'Accuracy': 0.8340344551282052}
epoch: 26 step: 312, loss is 0.7934075593948364
{'Accuracy': 0.8378405448717948}
epoch: 27 step: 312, loss is 0.771365046501l597
{'Accuracy': 0.8218149038461539}
epoch: 28 step: 312, loss is 0.44923135638237
{'Accuracy': 0.8509615384615384}
epoch: 29 step: 312, loss is 0.5456302165985107
{'Accuracy': 0.8633814102564102}
epoch: 30 step: 312, loss is 0.46565136313438416
{'Accuracy': 0.8651842948717948}
```



测试预测精度：

```
test results: {'Accuracy': 0.9046474358974359}
```

再拿九张测试一下：七张的结果正确（对比之前4张正确，虽如此小的数据量不严谨但已经说明测试精度有明显提升了）



实验结果及分析

结果见上述过程中的展示。

可以得出以下几点结论：

①一定范围内，epoch越大，最终的acc越高，loss越低（这都是平均而言）。实际上当epoch超过一定范围时（一般这个范围还挺大的），继续迭代难以提升模型的效果。并且，epoch是基于train_set的，在train_test上效果好也并不代表在test_set上效果也好，尤其是二者相差较大的时候。为了提升模型的泛化能力，防止over fitting，epoch也不必很大。可以加入dropout来防止overfitting。

②对于learning_rate，并不是越大越好。learining_rate就代表了模型的学习速度，这并不意味着总体上模型能学到更多"好的"东西，learning_rate较大时，模型学习错误信息/特征的能力也更强，所以总体而言，难以断定learning_rate是多大，可以在较小epoch范围内，对不同的learning_rate进行测试试验，绘制acc/loss~learning_rate大致曲线来确定learning_rate的值。当数据集质量很高的时候，learning_rate可能较大比较好，当数据集质量不太高的时

候,learning_rate不能设太大，否则模型将在错误的道路上越走越远，变成一无是处的模型。learning_rate=0.001似乎就是最常见的设置方法。（好多模型都是设置的这个值，或者不知道该设置多少的时候也会用0.001先试一下）

③batch_size越大效果似乎会越好，但是效果的提升并不明显。

④对于模型效果的提升，除了epoch、learning_rate等网络以外参数的选择，网络本身的参数更加重要（决定性的）。nn自身的参数，层数的搭建中，ksize越小，其能关注的细节越多，但相应的计算量也会更大。多增加一些conv层和linear层（全连接层，fully-connected层，fc层），可以增加模型的拟合能力。其实就是增加了nn的深度，向深度学习靠近。牺牲性能，提升精度。其实这里网络太深的时候，可解释性并不好，难以解释其特征是如何提取的。网络比较浅的时候，可解释性会好一些。

⑤对于不同的优化器。实际效果与tarin_set中数据的分布有很大关系，也与optimizer的特点有关。