

# image classification with MobileNet model on CIFAR-100 dataset

计科1804 张曦辰 201808010425

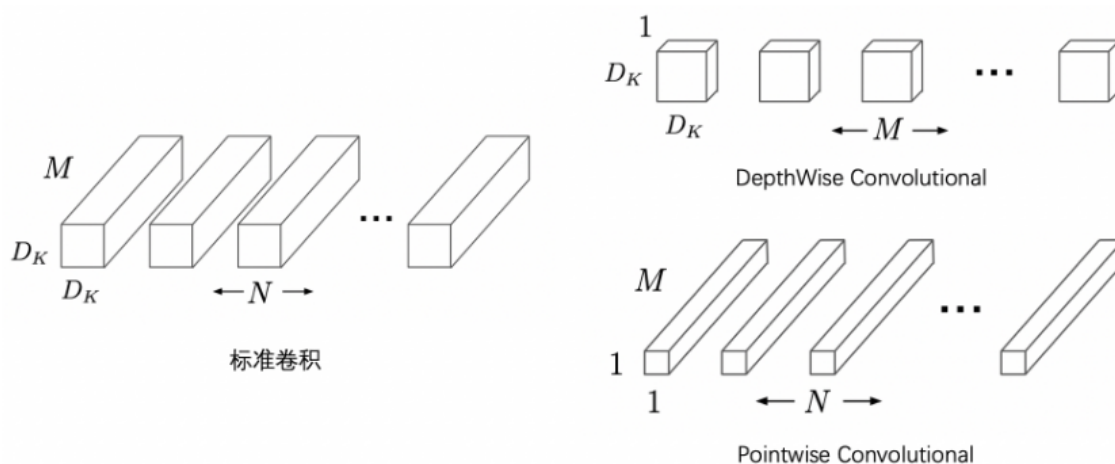
## 模型理论

### 项目简介

本项目使用paddle实现图像分类模型 MobileNet-V1网络的训练和预测。MobileNet-V1是针对传统卷积模块计算量大的缺点进行改进后，提出的一种更高效的能够在移动设备上部署的轻量级神经网络，建议使用GPU运行。

### 模型结构

MobileNet的核心思想是将传统卷积分解为深度可分离卷积与 $1 \times 1$ 卷积。深度可分离卷积是指输入特征图的每个channel都对应一个卷积核，这样输出的特征的每个channel只与输入特征图对应的channel相关，具体的例如输入一个 $K \times M \times N$ 的特征图，其中K为特征图的通道数，M、N为特征图的宽高，假设传统卷积需要一个大小为 $C \times K \times 3 \times 3$ 的卷积核来得到输出大小为 $C \times M' \times N'$ 的新的特征图。而深度可分离卷积则是首先使用K个大小为 $3 \times 3$ 的卷积核分别对输入的K个channel进行卷积得到K个特征图（DepthWise Conv部分），然后再使用大小为 $C \times K \times 1 \times 1$ 的卷积来得到大小为 $C \times M' \times N'$ 的输出（PointWise Conv部分）。这种卷积操作能够显著的降低模型的大小和计算量，而在性能上能够与标准卷积相当，深度可分离卷积具体的结构如下图。



假设输入特征图大小为： $C_{in} \times H_{in} \times W_{in}$ ，使用卷积核为 $K \times K$ ，输出的特征图大小为 $C_{out} \times H_{out} \times W_{out}$ ，对于标准的卷积，其计算量为：

$$K \times K \times C_{in} \times C_{out} \times H_{out} \times W_{out}$$

对于分解后的深度可分离卷积，计算量可通过DW部分和PW部分计算量的和得到，公式如下：

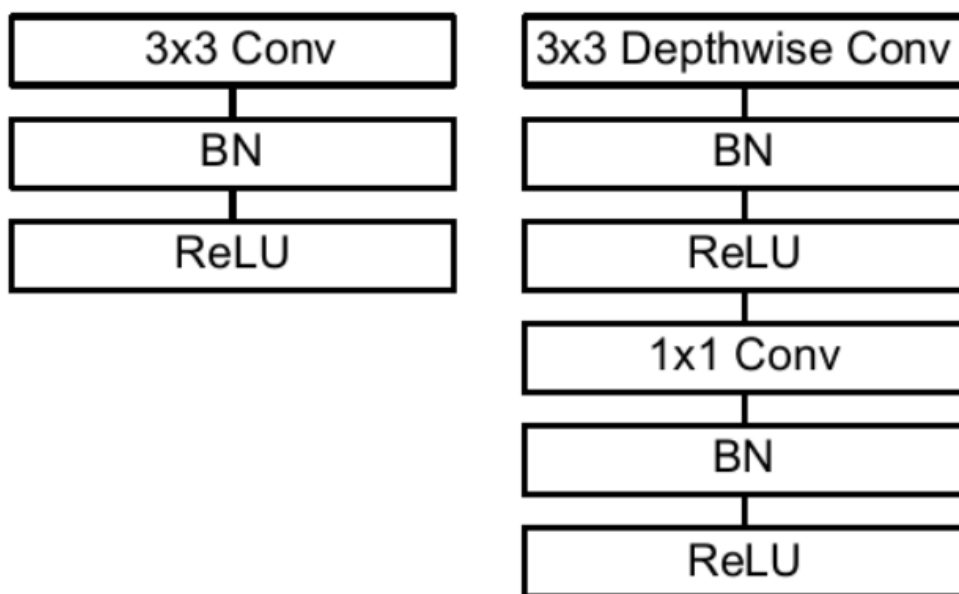
$$K \times K \times C_{in} \times H_{out} \times W_{out} + C_{in} \times C_{out} \times H_{out} \times W_{out}$$

因此相比于标准卷积，深度可分离卷积的计算量降低了：

$$\frac{K \times K \times C_{in} \times H_{out} \times W_{out} + C_{in} \times C_{out} \times H_{out} \times W_{out}}{K \times K \times C_{in} \times C_{out} \times H_{out} \times W_{out}} = \frac{1}{C_{out}} + \frac{1}{K^2}$$

由上式可知，对于一个大小为 $3 \times 3$ 的卷积核，计算量降低了约7-9倍。

简单来说，MobileNet v1就是将常规卷积替换为深度可分离卷积的VGG网络。下图分别是VGG和MobileNet v1 一个卷积块所包含的网络层。



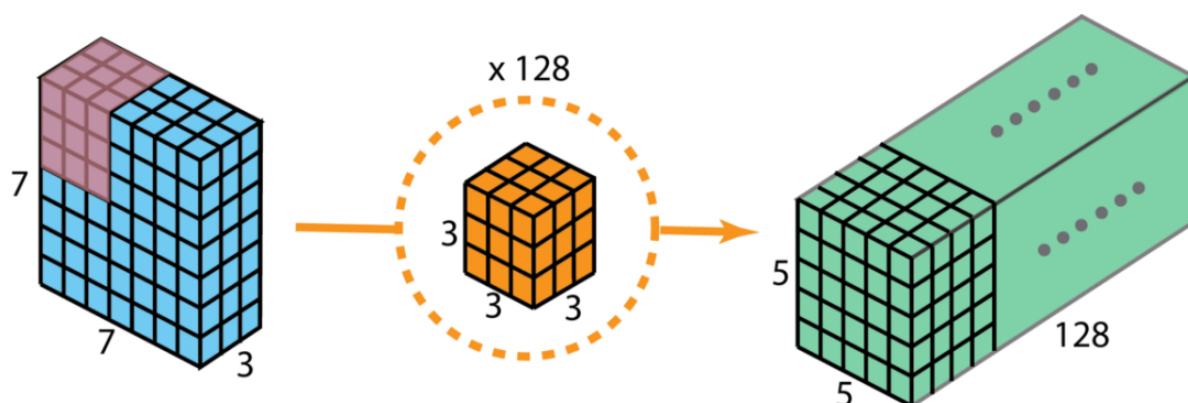
可以看到，VGG的卷积块就是一个常规3\*3卷积和一个BN、一个ReLU激活层。MobileNet v1则是一个33深度可分离卷积和一个1\*1卷积，后面分别跟着一个BN和ReLU层。MobileNet v1的ReLU指的是ReLU6，区别于ReLU的是对激活输出做了一个clip，使得最大最输出值不超过6，这么做的目的是为了防

那么，什么是深度可分离卷积呢？

从维度的角度看，卷积核可以看成是一个空间维(宽和高)和通道维的组合，而卷积操作则视为空间相关性和通道相关性的联合映射。从inception的1\*1卷积来看，卷积中的空间相关性和通道相关性是可以解耦的，将它们分开进行映射，可能会达到更好的效果。

深度可分离卷积是在1\*1卷积基础上的一种创新。主要包括两个部分：深度卷积和1\*1卷积。深度卷积的目的在于对输入的每一个通道都单独使用一个卷积核对其进行卷积，也就是通道分离后再组合。1\*1卷积的目的则在于加强深度。下面以一个例子来看一下深度可分离卷积。

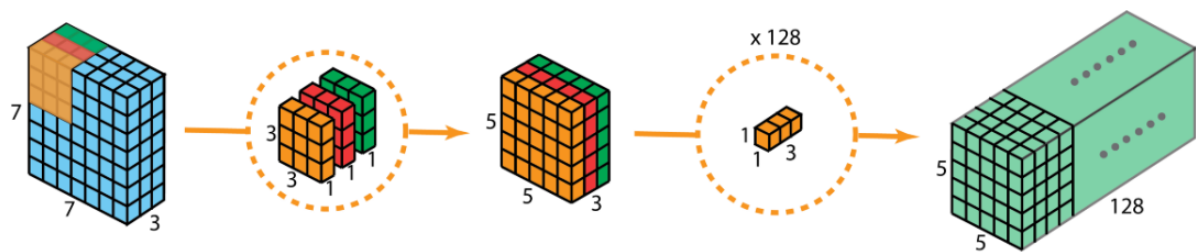
假设我们用128个3\*3\*3的滤波器对一个7\*7\*3的输入进行卷积，可得到5\*5\*128的输出。如下图所示：



其计算量为 $5512833 \times 3 = 86400$ 。

现在看如何使用深度可分离卷积来实现同样的结果。深度可分离卷积的第一步是深度卷积（Depth-Wise）。这里的深度卷积，就是分别用3个3\*3\*1的滤波器对输入的3个通道分别做卷积，也就是说要做3次卷积，每次卷积都有一个5\*5\*1的输出，组合在一起便是5\*5\*3的输出。

现在为了拓展深度达到128，我们需要执行深度可分离卷积的第二步：1\*1卷积（Point-Wise）。现在我们用128个1\*1\*3的滤波器对5\*5\*3进行卷积，就可以得到5\*5\*128的输出。完整过程如下图所示：



那么我们来看一下深度可分离卷积的计算量如何。第一步深度卷积的计算量： $5513313=675$ 。第二步  $1\times 1$  卷积的计算量： $5512811\times 3=9600$ ，合计计算量为10275次。可见，相同的卷积计算输出，深度可分离卷积要比常规卷积节省12倍的计算成本。所以深度可分离卷积是MobileNet v1能够轻量化的关键原因。

MobileNet v1完整网络结构如下图所示。

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$	Conv dw / s1	$3 \times 3 \times 512$ dw
	Conv / s1	$1 \times 1 \times 512 \times 512$
	Conv dw / s2	$3 \times 3 \times 512$ dw
	Conv / s1	$1 \times 1 \times 512 \times 1024$
	Conv dw / s2	$3 \times 3 \times 1024$ dw
	Conv / s1	$1 \times 1 \times 1024 \times 1024$
	Avg Pool / s1	Pool $7 \times 7$
	FC / s1	$1024 \times 1000$
	Softmax / s1	Classifier

MobileNet v1与GoogleNet和VGG 16在ImageNet上的效果对比，如下表所示：

Table 8. MobileNet Comparison to Popular Models

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
GoogleNet	69.8%	1550	6.8
VGG 16	71.5%	15300	138

可以看到，MobileNet v1相比与VGG 16精度损失不超过一个点的情况下，参数量小了32倍之多！MobileNet v1在速度和大小上的优势是显而易见的。

## 代码实现

### 实验设计逻辑

指定数据集：cifar100，通过高层API调用。

### 数据处理

```
import paddle.vision.transforms as T

transforms = T.Compose([
    T.RandomHorizontalFlip(0.5), #水平翻转
    T.RandomRotation(15), #随机反转角度范围
    T.RandomVerticalFlip(0.15),
    T.RandomRotation(15),
    T.ToTensor()
])

# 训练数据集
train_dataset =
paddle.vision.datasets.Cifar100(mode='train', transform=transforms)

# 验证数据集
eval_dataset = paddle.vision.datasets.Cifar100(mode='test', transform=transforms)
eval1_dataset = paddle.vision.datasets.Cifar100(mode='test', transform=
(T.ToTensor()))
```

### 模型设计

```
network =paddle.vision.models.mobilenet_v2(pretrained=True,num_classes=100)
```

## 训练配置

### 硬件信息

当前环境	至尊GPU	<a href="#">切换环境 ?</a>
CPU	4	
RAM	32GB	
GPU	v100	
显存	32GB	
磁盘	100GB	

### GPU资源信息

算力卡余额	106.2点(1.0点/小时)
本周可使用	45.7小时 <a href="#">?</a>

### 环境配置

Python版本	python 3.7
框架版本	PaddlePaddle 2.0.2

### 终端连接 (1/3)

终端 1	<a href="#">关闭终端</a>
<a href="#">+ 新建终端</a>	

## 模型训练与评估

```
model.prepare(paddle.optimizer.Adam(learning_rate=0.001,parameters=model.parameters()),
               paddle.nn.CrossEntropyLoss(),
               paddle.metric.Accuracy())

callback=paddle.callbacks.VisualDL(log_dir='./log_Res101_ss2q')

model.fit(train_dataset,# 训练数据集
          eval_dataset,# 评估数据集
          epochs=50,# 总的训练轮次
          batch_size=4096,# 批次计算的样本量大小
          verbose=1,# 日志展示格式
          shuffle=True,# 是否打乱样本集
          callbacks=callback# 回调函数使用)

result = model.evaluate(eval1_dataset, verbose=1,batch_size=5000)
```

```
print(result)
```

➤ Eval begin...

The loss value printed in the log is the current batch, and the metric is the average value of previous step.

step 2/2 [=====] - loss: 2.2179 - acc: 0.5375 - 2s/step

Eval samples: 10000

{'loss': [2.2179222], 'acc': 0.5375}

## 不用飞桨的Python版本

### 代码

```
import tensorflow as tf
import os
import numpy as np
import pickle

# 文件存放目录
CIFAR_DIR = "./cifar-10-batches-py"

def load_data( filename ):
    '''read data from data file'''
    with open( filename, 'rb' ) as f:
        data = pickle.load( f, encoding='bytes' ) # python3 需要添加上
        encoding='bytes'
        return data[b'data'], data[b'labels'] # 并且 在 key 前需要加上 b

class CifarData:
    def __init__( self, filenames, need_shuffle ):
        '''参数1:文件夹 参数2:是否需要随机打乱'''
        all_data = []
        all_labels = []

        for filename in filenames:
            # 将所有数据,标签分别存放在两个list中
            data, labels = load_data( filename )
            all_data.append( data )
            all_labels.append( labels )

        # 将列表 组成 一个numpy类型的矩阵!!!!
        self._data = np.vstack(all_data)
        # 对数据进行归一化,尺度固定在 [-1, 1] 之间
        self._data = self._data / 127.5 - 1
        # 将列表,变成一个 numpy 数组
        self._labels = np.hstack( all_labels )
        # 记录当前的样本 数量
        self._num_examples = self._data.shape[0]
        # 保存是否需要随机打乱
        self._need_shuffle = need_shuffle
        # 样本的起始点
        self._indicator = 0
        # 判断是否需要打乱
        if self._need_shuffle:
            self._shffle_data()

    def _shffle_data( self ):
        # np.random.permutation() 从 0 到 参数,随机打乱
```

```

p = np.random.permutation( self._num_examples )
# 保存 已经打乱 顺序的数据
self._data = self._data[p]
self._labels = self._labels[p]

def next_batch( self, batch_size ):
    '''return batch_size example as a batch'''
    # 开始点 + 数量 = 结束点
    end_indictor = self._indicator + batch_size
    # 如果结束点大于样本数量
    if end_indictor > self._num_examples:
        if self._need_shuffle:
            # 重新打乱
            self._shffle_data()
            # 开始点归零,从头再来
            self._indicator = 0
            # 重新指定 结束点. 和上面的那一句,说白了就是重新开始
            end_indictor = batch_size # 其实就是 0 + batch_size, 把 0 省略了
        else:
            raise Exception( "have no more examples" )
    # 再次查看是否 超出边界了
    if end_indictor > self._num_examples:
        raise Exception( "batch size is larger than all example" )

    # 把 batch 区间 的data和label保存,并最后return
    batch_data = self._data[self._indicator:end_indictor]
    batch_labels = self._labels[self._indicator:end_indictor]
    self._indicator = end_indictor
    return batch_data, batch_labels

# 拿到所有文件名称
train_filename = [os.path.join(CIFAR_DIR, 'data_batch_%d' % i) for i in range(1,
6)]
# 拿到标签
test_filename = [os.path.join(CIFAR_DIR, 'test_batch')]

# 拿到训练数据和测试数据
train_data = CifarData( train_filename, True )
test_data = CifarData( test_filename, False )

def separable_conv_block(x, output_channel_number, name):
    '''
    mobilenet 卷积块
    :param x:
    :param output_channel_number: 输出通道数量 output channel of 1*1 conv layer
    :param name:
    :return:
    '''
    with tf.variable_scope(name):
        # 获取图像通道数
        input_channel = x.get_shape().as_list()[-1]
        # 按照最后一维进行拆分 eg channel_wise_x: [channel1, channel2, ...]
        channel_wise_x = tf.split(x, input_channel, axis = 3)
        output_channels = []
        # 针对 每一个 通道进行 卷积
        for i in range(len(channel_wise_x)):
            output_channel = tf.layers.conv2d(channel_wise_x[i],

```

```

        1,
        (3, 3),
        strides = (1, 1),
        padding = 'same',
        activation = tf.nn.relu,
        name = 'conv_%d' % i
    )

    # 将卷积后的通道保存到列表中
    output_channels.append(output_channel)

# 合并整个列表
concat_layer = tf.concat(output_channels, axis = 3)
# 再次进行一个 1 * 1 的卷积
conv1_1 = tf.layers.conv2d(concat_layer,
                             output_channel_number,
                             (1, 1),
                             strides = (1, 1),
                             padding = 'same',
                             activation = tf.nn.relu,
                             name = 'conv1_1'
                             )

return conv1_1

```

```

# 设计计算图
# 形状 [None, 3072] 3072 是 样本的维数, None 代表位置的样本数量
x = tf.placeholder( tf.float32, [None, 3072] )
# 形状 [None] y的数量和x的样本数是对应的
y = tf.placeholder( tf.int64, [None] )

# [None, ], eg: [0, 5, 6, 3]
x_image = tf.reshape( x, [-1, 3, 32, 32] )
# 将最开始的向量式的图片,转为真实的图片类型
x_image = tf.transpose( x_image, perm= [0, 2, 3, 1] )

# conv1:神经元 feature_map 输出图像 图像大小: 32 * 32
conv1 = tf.layers.conv2d( x_image,
                          32, # 输出的通道数(也就是卷积核的数量)
                          ( 3, 3 ), # 卷积核大小
                          padding = 'same',
                          activation = tf.nn.relu,
                          name = 'conv1'
                          )

# 池化层 图像输出为: 16 * 16
pooling1 = tf.layers.max_pooling2d( conv1,
                                     ( 2, 2 ), # 核大小 变为原来的 1/2
                                     ( 2, 2 ), # 步长
                                     name='pool1'
                                     )

separable_2a = separable_conv_block(pooling1,
                                     32,
                                     name = 'separable_2a'
                                     )

separable_2b = separable_conv_block(separable_2a,

```



```

        32,
        name = 'separable_2b'
    )

pooling2 = tf.layers.max_pooling2d( separable_2b,
    ( 2, 2 ), # 核大小 变为原来的 1/2
    ( 2, 2 ), # 步长
    name='pool2'
)

separable_3a = separable_conv_block(pooling2,
    32,
    name = 'separable_3a'
)
separable_3b = separable_conv_block(separable_3a,
    32,
    name = 'separable_3b'
)

pooling3 = tf.layers.max_pooling2d( separable_3b,
    ( 2, 2 ), # 核大小 变为原来的 1/2
    ( 2, 2 ), # 步长
    name='pool3'
)

flatten = tf.contrib.layers.flatten(pooling3)
y_ = tf.layers.dense(flatten, 10)

# 使用交叉熵 设置损失函数
loss = tf.losses.sparse_softmax_cross_entropy( labels = y, logits = y_ )
# 该api,做了三件事儿 1. y_ -> softmax 2. y -> one_hot 3. loss = ylogy

# 预测值 获得的是 每一行上 最大值的 索引.注意:tf.argmax()的用法,其实和 np.argmax() 一样的
predict = tf.argmax( y_, 1 )
# 将布尔值转化为int类型,也就是 0 或者 1, 然后再和真实值进行比较. tf.equal() 返回值是布尔类型
correct_prediction = tf.equal( predict, y )
# 比如说第一行最大值索引是6,说明是第六个分类.而y正好也是6,说明预测正确

# 将上句的布尔类型 转化为 浮点类型,然后进行求平均值,实际上就是求出了准确率
accuracy = tf.reduce_mean( tf.cast(correct_prediction, tf.float64) )

with tf.name_scope( 'train_op' ): # tf.name_scope() 这个方法的作用不太明白(有点迷糊!)
    train_op = tf.train.AdamOptimizer( 1e-3 ).minimize( loss ) # 将 损失函数 降到
    最低

# 初始化变量
init = tf.global_variables_initializer()

batch_size = 20
train_steps = 10000
test_steps = 100
with tf.Session() as sess:
    sess.run( init ) # 注意:这一步必须要有!!
    # 开始训练

```

```

for i in range( train_steps ):
    # 得到batch
    batch_data, batch_labels = train_data.next_batch( batch_size )
    # 获得 损失值, 准确率
    loss_val, acc_val, _ = sess.run( [loss, accuracy, train_op], feed_dict=
{x:batch_data, y:batch_labels} )
    # 每 500 次 输出一条信息
    if ( i+1 ) % 500 == 0:
        print('[Train] Step: %d, loss: %4.5f, acc: %4.5f' % ( i+1, loss_val,
acc_val ))
    # 每 5000 次 进行一次 测试
    if ( i+1 ) % 5000 == 0:
        # 获取数据集,但不随机
        test_data = CifarData( test_filename, False )
        all_test_acc_val = []
        for j in range( test_steps ):
            test_batch_data, test_batch_labels = test_data.next_batch(
batch_size )
            test_acc_val = sess.run( [accuracy], feed_dict={
x:test_batch_data, y:test_batch_labels } )
            all_test_acc_val.append( test_acc_val )
            test_acc = np.mean( all_test_acc_val )

            print('[Test ] Step: %d, acc: %4.5f' % ( (i+1), test_acc ))

# 思想: 深度可分离卷积
# 精度也略有损失,但计算量会比较小, 另外 Inception v3 性价比比较高
# 将每个通道拆开,进行卷积,在将通道结果合并起来

'''
训练一万次的最终结果:
=====
[Test ] Step: 10000, acc: 0.64250
=====

训练十万次的最终结果:
=====
[Test ] Step: 100000, acc: 0.69350
=====
'''

```

## 环境


windows10

CPU

AMD Ryzen 7 4700U with Radeon Graphics

解释器版本

Python interpreter:

 Python 3.5 (tensorflow) Z:\environment\Anaconda\envs\tensorflow\python.exe

## 训练结果

```
Z:\environment\Anaconda\envs\tensorflow\python.exe C:/Users/zxc/Desktop/小学期实习/MobileNet/1.py
2021-07-28 18:00:06.037184: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports :
[Train] Step: 500, loss: 1.57771, acc: 0.45000
[Train] Step: 1000, loss: 1.47181, acc: 0.45000
[Train] Step: 1500, loss: 1.33510, acc: 0.50000
[Train] Step: 2000, loss: 1.23004, acc: 0.50000
[Train] Step: 2500, loss: 1.77500, acc: 0.50000
[Train] Step: 3000, loss: 1.43297, acc: 0.45000
[Train] Step: 3500, loss: 1.32651, acc: 0.40000
[Train] Step: 4000, loss: 1.06561, acc: 0.65000
[Train] Step: 4500, loss: 1.00242, acc: 0.55000
[Train] Step: 5000, loss: 0.64725, acc: 0.85000
[Test ] Step: 5000, acc: 0.58350
[Train] Step: 5500, loss: 1.13690, acc: 0.55000
[Train] Step: 6000, loss: 0.77869, acc: 0.75000
[Train] Step: 6500, loss: 0.94624, acc: 0.65000
[Train] Step: 7000, loss: 1.31594, acc: 0.55000
[Train] Step: 7500, loss: 1.00929, acc: 0.65000
[Train] Step: 8000, loss: 1.09062, acc: 0.40000
[Train] Step: 8500, loss: 0.72904, acc: 0.70000
[Train] Step: 9000, loss: 0.89707, acc: 0.65000
[Train] Step: 9500, loss: 0.78010, acc: 0.70000
[Train] Step: 10000, loss: 0.89319, acc: 0.50000
[Test ] Step: 10000, acc: 0.64600
```

```
Process finished with exit code 0
```

## 实训总结

---

这次实训让我对深度学习MobileNet 网络有了初步的了解，同时这次同时使用了百度的飞桨平台提供的paddle库和自己的电脑原生tensorflow基于同一数据集训练了不同的模型进行了对比，从准确率和训练时间上有所比较，感到受益匪浅。