第七章 图像数据处理

7.1 TFRecord输入数据格式

TensorFlow提供了一种统一的格式来存储数据——TFRecord格式

7.1.1 TFRecord格式介绍

7.1.2 TFRecord样例程序

7.2 图像数据处理

7.2.1 TensorFlow图像处理函数

1. 图像编码处理

2. 图像大小调整

3. 图像翻转

4. 图像色彩调整

5. 处理标注框

7.2.2 图像预处理完整样例

7.3 多线程输入数据处理框架

7.3.1 队列与多线程

7.3.2 输入文件队列

7.3.3 组合训练数据（batching）

7.3.4 输入数据处理框架

总结

第七章 图像数据处理

第6章中详细介绍了卷积神经网络，并提到了通过卷积神经网络给图像识别技术带来了突破性进展，本章从另外一个维度来进一步提升图像识别的精度以及训练的速度。

在很多图像识别问题中，光照、对比度等外界因素会对识别效果造成很大的影响，所以本章介绍如何对图像数据进行预处理使得训练得到的神经网络模型尽可能小的被无关因素所影响。

复杂的预处理过程可能导致训练效率下降，为了减小预训练对训练速度的影响，本章也将介绍TensorFlow中多线程处理输入数据的解决方案。

7.1 TFRecord输入数据格式

TensorFlow提供了一种统一的格式来存储数据——TFRecord格式

7.1.1 TFRecord格式介绍

tf.train.Example Protocol Buffer：TFRecord文件中的数据都是通过该格式储存的。

下列代码给出了tf.train.Example的定义

message Example{

Features features=1;

};

message Features{

map<string,Feature> feature=1;

};

message Feature{

oneof kind{

BytesList bytes\_list=1;

FloatList bytes\_list=2;

Int64List bytes\_list=3;

}

};

tf.train.Example中包含了一个从属性名称到取值的字典：

属性名称：一个字符串

属性取值：字符串（BytesList）、实数列表（FloatList）、整数列表（Int64List）

7.1.2 TFRecord样例程序

1. 将mnist输入数据转化为TFRecord格式

import tensorflow as tf

from tensorflow.examples.tutorials.mnist import input\_data

import numpy as np

# 1. 将输入转化成TFRecord格式并保存

# 定义函数转化变量类型。

def \_int64\_feature(value):

return tf.train.Feature(int64\_list=tf.train.Int64List(value=[value]))

def \_bytes\_feature(value):

return tf.train.Feature(bytes\_list=tf.train.BytesList(value=[value]))

# 读取mnist数据。

mnist = input\_data.read\_data\_sets("../../datasets/MNIST\_data",dtype=tf.uint8, one\_hot=True)

images = mnist.train.images

labels = mnist.train.labels

pixels = images.shape[1]

num\_examples = mnist.train.num\_examples

# 输出TFRecord文件的地址。

filename = "Records/output.tfrecords" # 需要存在Records目录

writer = tf.python\_io.TFRecordWriter(filename)

for index in range(num\_examples):

image\_raw = images[index].tostring()

example = tf.train.Example(features=tf.train.Features(feature={

'pixels': \_int64\_feature(pixels),

'label': \_int64\_feature(np.argmax(labels[index])),

'image\_raw': \_bytes\_feature(image\_raw)

}))

writer.write(example.SerializeToString())

writer.close()

print("TFRecord文件已保存。")

2. 读取TFRecord文件中的数据

reader = tf.TFRecordReader()

filename\_queue = tf.train.string\_input\_producer(["Records/output.tfrecords"])

\_,serialized\_example = reader.read(filename\_queue)

# 解析读取的样例。

features = tf.parse\_single\_example(

serialized\_example,

features={

'image\_raw':tf.FixedLenFeature([],tf.string),

'pixels':tf.FixedLenFeature([],tf.int64),

'label':tf.FixedLenFeature([],tf.int64)

})

images = tf.decode\_raw(features['image\_raw'],tf.uint8)

labels = tf.cast(features['label'],tf.int32)

pixels = tf.cast(features['pixels'],tf.int32)

sess = tf.Session()

# 启动多线程处理输入数据。

coord = tf.train.Coordinator()

threads = tf.train.start\_queue\_runners(sess=sess,coord=coord)

for i in range(10):

image, label, pixel = sess.run([images, labels, pixels])

7.2 图像数据处理

7.2.1 TensorFlow图像处理函数

1. 图像编码处理

图像在存储时并不是直接记录像素矩阵中的数字，而是记录了压缩编码之后的结果，所以要将一张图像还原成一个三维矩阵，需要解码的过程。TensorFlow提供了对jpeg和png格式图像的编码/解码函数。

使用TensorFlow中对jpeg格式图像的编码/解码函数

# matplotlib.pyplot是一个python画图工具

import matplotlib.pyplot as plt

import tensorflow as tf

# 读取图像的原始数据

image\_raw\_data=tf.gfile.FastGFile("E:\\Opencv Image\\anglababy.jpg",'rb').read()

with tf.Session() as sess:

# 将图像使用的jpeg的格式解码从而得到图像对应的三维矩阵

# TensorFlow还提供了tf.image.decode\_png函数对png格式的图像进行解码

# 解码之后的结果为一个张量，在使用它的取值之前需要明确调用运行的过程

img\_data=tf.image.decode\_jpeg(image\_raw\_data)

print(img\_data.eval())

# 使用pyplot得到图像

plt.imshow(img\_data.eval())

plt.show()

# 将数据的类型转化成实数方便后续处理

img\_data=tf.image.convert\_image\_dtype(img\_data,dtype=tf.uint8)

# 将表示一张图像的三维矩阵重新按照jpeg个数编码并存到文件中

# 打开该图，可以得到和原图一样的图像

encode\_image=tf.image.encode\_jpeg(img\_data) #输入必须为uint8形式的，不然会报错

with tf.gfile.GFile("E:\\Opencv Image\\an.jpg",'wb') as f:

f.write(encode\_image.eval())

2. 图像大小调整

因为获取的图像大小不固定，但神经网络输入节点的个数是固定的，所以在将图像像素作为输入提供给神经网络之前，需要先将图像大小进行统一。

1.通过算法调整，使得得到的新图像尽量保存原始图像的所有信息

TensorFlow实现：提供了四种不同的算法，并封装到了tf.image.resize\_images函数

tf.image.resize\_images(images, new\_height, new\_width, method=0)

# Resize images to new\_width, new\_height using the specified method.

1

2

tf.image.resize\_images函数中method对应的取值

Method取值 图像大小调整算法

0 双线性差值法（Bilinear interpolation）

1 最近邻法（Nearest neighbour interpolation）

2 双三次差值法（Bicubic interpolation）

3 面积差值法（Area interpolation）

代码实现：

# matplotlib.pyplot是一个python画图工具

import matplotlib.pyplot as plt

import tensorflow as tf

import numpy as np

# 读取图像的原始数据

image\_raw\_data=tf.gfile.FastGFile("E:\\Opencv Image\\anglababy.jpg",'rb').read()

with tf.Session() as sess:

# 将图像使用的jpeg的格式解码从而得到图像对应的三维矩阵

# TensorFlow还提供了tf.image.decode\_png函数对png格式的图像进行解码

# 解码之后的结果为一个张量，在使用它的取值之前需要明确调用运行的过程

img\_data=tf.image.decode\_jpeg(image\_raw\_data)

print(img\_data.eval())

# 使用pyplot得到图像

# plt.imshow(img\_data.eval())

# plt.show()

# 将数据的类型转化成实数方便后续处理

img\_data=tf.image.convert\_image\_dtype(img\_data,dtype=tf.uint8)

# 将表示一张图像的三维矩阵重新按照jpeg个数编码并存到文件中

# 打开该图，可以得到和原图一样的图像

# encode\_image=tf.image.encode\_jpeg(img\_data)

# with tf.gfile.GFile("E:\\Opencv Image\\an.jpg",'wb') as f:

# f.write(encode\_image.eval())

with tf.Session() as sess:

resized = tf.image.resize\_images(img\_data, [300, 300], method=3)

print(img\_data.get\_shape())

# TensorFlow的函数处理图片后存储的数据是float32格式的，需要转换成uint8才能正确打印图片。

print( "Digital type: ", resized.dtype)

angelababy2 = np.asarray(resized.eval(), dtype='uint8')

# tf.image.convert\_image\_dtype(rgb\_image, tf.float32)

plt.imshow(angelababy2)

plt.show()

原始图像 原始图像

双线性差值 最近邻

双三次插值 面积插值

2. 裁剪和填充

tf.image.resize\_image\_with\_crop\_or\_pad(image, target\_height, target\_width)

# Crops and/or pads an image to a target width and height.

# Resizes an image to a target width and height by either centrally cropping the image or

# padding it evenly with zeros.

# image:原始图像

# target\_height, target\_width：目标大小

# 如果原始图像的尺寸大于目标图像：自动截取居中的部分

# 如果原始图像的尺寸小于目标图像：四周全0填充

import matplotlib.pyplot as plt

import tensorflow as tf

# 读取图像的原始数据

image\_raw\_data=tf.gfile.FastGFile("E:\\Opencv Image\\anglababy.jpg",'rb').read()

with tf.Session() as sess:

img\_data=tf.image.decode\_jpeg(image\_raw\_data)

print(img\_data.eval())

# 将数据的类型转化成实数方便后续处理

img\_data=tf.image.convert\_image\_dtype(img\_data,dtype=tf.uint8)

with tf.Session() as sess:

croped = tf.image.resize\_image\_with\_crop\_or\_pad(img\_data, 1000, 1000)

padded = tf.image.resize\_image\_with\_crop\_or\_pad(img\_data, 500, 500)

plt.imshow(croped.eval())

plt.show()

plt.imshow(padded.eval())

plt.show()

填充到1000\*1000 裁剪到500\*500

通过比例调整图像大小——截取中间的部分

crop = tf.image.central\_crop(image, central\_fraction=0.5)

# 第一个参数：原始图像

# 第二个参数：调整比例，(0,1]直接的实数

1

2

3

import matplotlib.pyplot as plt

import tensorflow as tf

# 读取图像的原始数据

image\_raw\_data=tf.gfile.FastGFile("E:\\Opencv Image\\anglababy.jpg",'rb').read()

with tf.Session() as sess:

img\_data=tf.image.decode\_jpeg(image\_raw\_data)

print(img\_data.eval())

# 将数据的类型转化成实数方便后续处理

img\_data=tf.image.convert\_image\_dtype(img\_data,dtype=tf.uint8)

# 5. 截取中间50%的图片

with tf.Session() as sess:

central\_cropped = tf.image.central\_crop(img\_data, 0.5)

plt.imshow(central\_cropped.eval())

plt.show()

3. 图像翻转

实现图像的上下翻转、左右翻转及对角线翻转

import matplotlib.pyplot as plt

import tensorflow as tf

# 读取图像的原始数据

image\_raw\_data=tf.gfile.FastGFile("E:\\Opencv Image\\anglababy.jpg",'rb').read()

with tf.Session() as sess:

img\_data=tf.image.decode\_jpeg(image\_raw\_data)

print(img\_data.eval())

# 将数据的类型转化成实数方便后续处理

img\_data=tf.image.convert\_image\_dtype(img\_data,dtype=tf.uint8)

# 6. 翻转图片

with tf.Session() as sess:

# 上下翻转

# flipped1 = tf.image.flip\_up\_down(img\_data)

# 左右翻转

# flipped2 = tf.image.flip\_left\_right(img\_data)

# 对角线翻转

transposed = tf.image.transpose\_image(img\_data)

plt.imshow(transposed.eval())

plt.show()

# 以一定概率上下翻转图片。

# flipped = tf.image.random\_flip\_up\_down(img\_data)

# 以一定概率左右翻转图片。

# flipped = tf.image.random\_flip\_left\_right(img\_data)

4. 图像色彩调整

1. 亮度

import matplotlib.pyplot as plt

import tensorflow as tf

# 读取图像的原始数据

image\_raw\_data=tf.gfile.FastGFile("E:\\Opencv Image\\anglababy.jpg",'rb').read()

with tf.Session() as sess:

img\_data=tf.image.decode\_jpeg(image\_raw\_data)

print(img\_data.eval())

# 将数据的类型转化成实数方便后续处理

img\_data=tf.image.convert\_image\_dtype(img\_data,dtype=tf.uint8)

plt.imshow(img\_data.eval())

plt.show()

# 将图片的亮度-0.5。

# adjusted = tf.image.adjust\_brightness(img\_data, -0.5)

# 将图片的亮度+0.5

adjusted = tf.image.adjust\_brightness(img\_data, 0.5)

# 在[-max\_delta, max\_delta)的范围随机调整图片的亮度。

# adjusted = tf.image.random\_brightness(img\_data, max\_delta=0.6)

plt.imshow(adjusted.eval())

plt.show()

|

亮度增加0.5

2. 对比度

import matplotlib.pyplot as plt

import tensorflow as tf

# 读取图像的原始数据

image\_raw\_data=tf.gfile.FastGFile("E:\\Opencv Image\\anglababy.jpg",'rb').read()

with tf.Session() as sess:

img\_data=tf.image.decode\_jpeg(image\_raw\_data)

print(img\_data.eval())

# 将数据的类型转化成实数方便后续处理

img\_data=tf.image.convert\_image\_dtype(img\_data,dtype=tf.uint8)

plt.imshow(img\_data.eval())

plt.show()

# 将图片的对比度-5

# adjusted = tf.image.adjust\_contrast(img\_data, -5)

# 将图片的对比度+5

adjusted = tf.image.adjust\_contrast(img\_data, 5)

# 在[lower, upper]的范围随机调整图的对比度。

# adjusted = tf.image.random\_contrast(img\_data, lower, upper)

plt.imshow(adjusted.eval())

plt.show()

对比度加5

3. 色相

import matplotlib.pyplot as plt

import tensorflow as tf

# 读取图像的原始数据

image\_raw\_data=tf.gfile.FastGFile("E:\\Opencv Image\\anglababy.jpg",'rb').read()

with tf.Session() as sess:

img\_data=tf.image.decode\_jpeg(image\_raw\_data)

print(img\_data.eval())

# 将数据的类型转化成实数方便后续处理

img\_data=tf.image.convert\_image\_dtype(img\_data,dtype=tf.uint8)

adjusted = tf.image.adjust\_hue(img\_data, 0.1)

# adjusted = tf.image.adjust\_hue(img\_data, 0.3)

# adjusted = tf.image.adjust\_hue(img\_data, 0.6)

# adjusted = tf.image.adjust\_hue(img\_data, 0.9)

# 在[-max\_delta, max\_delta]的范围随机调整图片的色相。max\_delta的取值在[0, 0.5]之间。

# adjusted = tf.image.random\_hue(image, max\_delta)

plt.imshow(adjusted.eval())

plt.show()

4. 饱和度

import matplotlib.pyplot as plt

import tensorflow as tf

# 读取图像的原始数据

image\_raw\_data=tf.gfile.FastGFile("E:\\Opencv Image\\anglababy.jpg",'rb').read()

with tf.Session() as sess:

img\_data=tf.image.decode\_jpeg(image\_raw\_data)

print(img\_data.eval())

# 将数据的类型转化成实数方便后续处理

img\_data=tf.image.convert\_image\_dtype(img\_data,dtype=tf.uint8)

# 将图片的饱和度-5。

adjusted = tf.image.adjust\_saturation(img\_data, -5)

# 将图片的饱和度+5。

# adjusted = tf.image.adjust\_saturation(img\_data, 5)

# 在[lower, upper]的范围随机调整图的饱和度。

# adjusted = tf.image.random\_saturation(img\_data, lower, upper)

# 将代表一张图片的三维矩阵中的数字均值变为0，方差变为1。

# adjusted = tf.image.per\_image\_standardization(img\_data)

plt.imshow(adjusted.eval())

plt.show()

TensorFlow还提供了API来完成图像标准化的过程，将亮度均值变为0，方差变为1

# 将代表一张图片的三维矩阵中的数字均值变为0，方差变为1。

adjusted = tf.image.per\_image\_standardization(img\_data)

1

2

|

5. 处理标注框

在很多图像识别的数据集中，图像中需要关注的物体通常会被标注框圈出来，利用下述函数实现：

tf.image.draw\_bounding\_boxs(images,boxes,name=None)

# images：是 [batch, height, width, depth] 形状的四维矩阵，

# 数据类型为 float32、half 中的一种，第一个值batch是因为处理的是一组图片。

# boxes： 形状 [batch, num\_bounding\_boxes, 4] 的三维矩阵，

# num\_bounding\_boxes 是标注框的数量，标注框由四个数字标示 [y\_min, x\_min, y\_max, x\_max]，数组类型为float32

# 例如：tf.constant([[[0.05, 0.05, 0.9, 0.7], [0.35, 0.47, 0.5, 0.56]]])

# shape 为 [1,2,4] 表示一张图片中的两个标注框；

# tf.constant([[[ 0. 0. 1. 1.]]]) 的 shape 为 [1,1,4]表示一张图片中的一个标注框

# name：操作的名称（可选）。

代码示例：

import matplotlib.pyplot as plt

import tensorflow as tf

# 读取图像的原始数据

image\_raw\_data=tf.gfile.FastGFile("E:\\Opencv Image\\anglababy.jpg",'rb').read()

with tf.Session() as sess:

img\_data=tf.image.decode\_jpeg(image\_raw\_data)

# 先将图像缩小，可让标注框更为清楚

img\_data=tf.image.resize\_images(img\_data,180,260,methed=1)

# tf.image.draw\_bounding\_boxes的输入加一个batch，也就是多张图像组成的思维矩阵

# 所以要将解码后的加一维度

# tf.image.draw\_bounding\_boxes输入为实数

batched = tf.expand\_dims(tf.image.convert\_image\_dtype(img\_data, tf.float32), 0)

boxes = tf.constant([[[0.01, 0.2, 0.5, 0.7],[0.25, 0.4, 0.32, 0.55]]])

# [0.05, 0.05, 0.9, 0.7]，（y min，x min，y max，x max）坐标点的相对位置，和原始大小相乘

result=tf.image.draw\_bounding\_boxes(batched,boxes)

plt.subplot(121), plt.imshow(img\_data.eval())

plt.subplot(122), plt.imshow(result[0].eval())

plt.show()

随机截取图像上有信息含量的部分也是提高模型健壮性的一种方式，可以使得训练得到的模型不受识别物体的大小的影响。

tf.image.sample\_distorted\_bounding\_box(

image\_size,

bounding\_boxes,

seed=None,

seed2=None,

min\_object\_covered=None,

aspect\_ratio\_range=None,

area\_range=None,

max\_attempts=None,

use\_image\_if\_no\_bounding\_boxes=None,

name=None)

# image\_size： 是包含 [height, width, channels] 三个值的一维数组。数值类型必须是 uint8，int8，int16，int32，int64 中的一种。

# bounding\_boxes： 是一个 shape 为 [batch, N, 4] 的三维数组，数据类型为float32，第一个batch是因为函数是处理一组图片的，N表示描述与图像相关联的N个边界框的形状，而标注框由4个数字 [y\_min, x\_min, y\_max, x\_max] 表示出来。例如：tf.constant([[[0.05, 0.05, 0.9, 0.7], [0.35, 0.47, 0.5, 0.56]]]) 的 shape 为 [1,2,4] 表示一张图片中的两个标注框；tf.constant([[[ 0. 0. 1. 1.]]]) 的 shape 为 [1,1,4]表示一张图片中的一个标注框

# seed： （可选）数组类型为 int，默认为0。如果任一个seed或被seed2设置为非零，随机数生成器由给定的种子生成。否则，由随机种子生成。

# seed2： （可选）数组类型为 int，默认为0。第二种子避免种子冲突。

# min\_object\_covered：（可选）数组类型为 float，默认为 0.1。图像的裁剪区域必须包含所提供的任意一个边界框的至少 min\_object\_covered 的内容。该参数的值应为非负数，当为0时，裁剪区域不必与提供的任何边界框有重叠部分。

# aspect\_ratio\_range： （可选）数组类型为 floats 的列表，默认为 [0.75, 1.33] 。图像的裁剪区域的宽高比（宽高比=宽/高） 必须在这个范围内。

# area\_range： （可选）数组类型为 floats 的列表，默认为 [0.05, 1] 。图像的裁剪区域必须包含这个范围内的图像的一部分。

# max\_attempts： （可选）数组类型为 int，默认为100。尝试生成图像指定约束的裁剪区域的次数。经过 # max\_attempts 次失败后，将返回整个图像。

# use\_image\_if\_no\_bounding\_boxes：（可选）数组类型为 bool，默认为False。如果没有提供边框，则用它来控制行为。如果为True，则假设有一个覆盖整个输入的隐含边界框。如果为False，就报错。

# name： 操作的名称（可选）。

代码实现：

import matplotlib.pyplot as plt

import tensorflow as tf

# 读取图像的原始数据

image\_raw\_data = tf.gfile.FastGFile("E:\\Opencv Image\\anglababy.jpg", 'rb').read()

with tf.Session() as sess:

img\_data = tf.image.decode\_jpeg(image\_raw\_data)

print(img\_data.eval())

img\_data = tf.image.resize\_images(img\_data, (330, 200), method=1)

boxes = tf.constant([[[0.01, 0.2, 0.5, 0.7], [0.25, 0.4, 0.32, 0.55]]])

# 随机图像截取

begin, size, bbox\_for\_draw = tf.image.sample\_distorted\_bounding\_box(

tf.shape(img\_data), bounding\_boxes=boxes,min\_object\_covered=0.1)

batched = tf.expand\_dims(tf.image.convert\_image\_dtype(img\_data, tf.float32), 0)

image\_with\_box = tf.image.draw\_bounding\_boxes(batched, bbox\_for\_draw)

distorted\_image = tf.slice(img\_data, begin, size)

plt.imshow(distorted\_image.eval())

plt.show()

7.2.2 图像预处理完整样例

# 《TensorFlow实战Google深度学习框架》07 图像数据处理

# win10 Tensorflow1.0.1 python3.5.3

# CUDA v8.0 cudnn-8.0-windows10-x64-v5.1

# filename:ts07.03.py # 图像预处理完整样例

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

# 1. 随机调整图片的色彩，定义两种顺序

def distort\_color(image, color\_ordering=0):

if color\_ordering == 0:

image = tf.image.random\_brightness(image, max\_delta=32./255.)

image = tf.image.random\_saturation(image, lower=0.5, upper=1.5)

image = tf.image.random\_hue(image, max\_delta=0.2)

image = tf.image.random\_contrast(image, lower=0.5, upper=1.5)

else:

image = tf.image.random\_saturation(image, lower=0.5, upper=1.5)

image = tf.image.random\_brightness(image, max\_delta=32./255.)

image = tf.image.random\_contrast(image, lower=0.5, upper=1.5)

image = tf.image.random\_hue(image, max\_delta=0.2)

return tf.clip\_by\_value(image, 0.0, 1.0)

# 2. 对图片进行预处理，将图片转化成神经网络的输入层数据

# 给定一张解码的图像、目标尺寸、及图像上的标注图，此函数可以对给出的图像进行预处理

# 输入：原始训练图像

# 输出：神经网络模型的输入层

# 注意：此处只处理模型的训练数据，对预测数据不需要使用随机变换的步骤

def preprocess\_for\_train(image, height, width, bbox):

# 查看是否存在标注框，如果没有标注框，则认为图像就是整个需要关注的部分

if bbox is None:

bbox = tf.constant([0.0, 0.0, 1.0, 1.0], dtype=tf.float32, shape=[1, 1, 4])

# 转换图像的张量类型

if image.dtype != tf.float32:

image = tf.image.convert\_image\_dtype(image, dtype=tf.float32)

# 随机的截取图片中一个块，减小物体大小对图像识别算法的影响

bbox\_begin, bbox\_size, \_ = tf.image.sample\_distorted\_bounding\_box(

tf.shape(image), bounding\_boxes=bbox, min\_object\_covered=0.1)

bbox\_begin, bbox\_size, \_ = tf.image.sample\_distorted\_bounding\_box(

tf.shape(image), bounding\_boxes=bbox, min\_object\_covered=0.1)

distorted\_image = tf.slice(image, bbox\_begin, bbox\_size)

# 将随机截取的图片调整为神经网络输入层的大小，大小调整的算法是随机选择的

distorted\_image = tf.image.resize\_images(distorted\_image, [height, width], method=np.random.randint(4))

# 随机左右翻转图像

distorted\_image = tf.image.random\_flip\_left\_right(distorted\_image)

# 使用一种随机的顺序调整图像的色彩

distorted\_image = distort\_color(distorted\_image, np.random.randint(2))

return distorted\_image

# 3. 读取图片

image\_raw\_data = tf.gfile.FastGFile("E:\\Opencv Image\\dog.jpg", "rb").read()

with tf.Session() as sess:

img\_data = tf.image.decode\_jpeg(image\_raw\_data)

boxes = tf.constant([[[0.05, 0.05, 0.9, 0.7], [0.35, 0.47, 0.5, 0.56]]])

# 运行6次获得6种不同的图像

for i in range(6):

result = preprocess\_for\_train(img\_data, 299, 299, boxes)

plt.imshow(result.eval())

plt.show()

7.3 多线程输入数据处理框架

上述的预处理方法可以减小无关因素对图像识别模型效果的影响，但这些复杂的操作会减慢整个训练过程，为了避免预处理带来的影响，TensorFlow提供了多线程处理输入数据的框架。

经典输入数据处理流程图

7.3.1 队列与多线程

队列是计算图上“有状态”的节点，其他节点可以修改其内容，也就是其他队列可以把新元素插入到队列后端（rear），也可以将队列前端（front）的元素删除。

1. TensorFlow中提供了 FIFOQueue 和 RandomShuffleQueue 两种队列

对于队列来说，修改队列状态的操作主要有Enqueue、EnqueueMany、Dequeue，他们需要获取队列指针，而非普通的值，如此才能修改队列内容，在python API中，它们就是队列的方法，例如：q.enqueue()

EnqueueMany：队列的初始化

Dequeue：出队

Enqueue：入队

FIFOQueue——先进先出队列

import tensorflow as tf

#创建一个先进先出队列，指定队列中可以保存两个元素，并指定类型为整形

q=tf.FIFOQueue(2,"int32")

#使用enqueue\_many函数来初始化队列中的元素

#和变量初始化类似，在使用队列之前需要明确调用这个初始化过程

init=q.enqueue\_many(([0,10],))

#使用Dequeue函数将队列中的第一个元素出队列，该元素的值将被存在变量x中

x=q.dequeue()

#将得到的值加1

y=x+1

#将加1后的值重新加入队列

q\_inc=q.enqueue([y])

with tf.Session() as sess:

#运行初始化队列操作

init.run()

for \_ in range(5):

# 运行q\_inc将执行数据出队列、出队的元素+1、重新加入队列的整个过程

v,\_ =sess.run([x,q\_inc])

# 打印出队元素的取值

print(v)

输出：

# 队列开始有[0,10]两个元素，第一个出队的为0，加1之后再次入队得到的队列为[10,1]；

# 第二次出队的为10，加1之后为11，得到的队列为[1,11]...

0

10

1

11

2

RandomShuffleQueue——会将队列中的元素打乱，每次出队列操作得到的是当前队列中的随机某个元素

神经网络训练中更希望使用的训练数据尽量随机，所以该方法更多用。

2. 队列的作用

如上所示，队列是一种数据结构

同时也是异步张量取值的一个重要机制（如多个线程可以同时向一个队列中写元素，或者同时读取一个队列中的元素）

3. 多线程协同功能

tf.Coordinator、tf.QueueRunner两个类来完成多线程协同的功能

tf.Coordinator主要用于协同多个线程一起停止（并提供了should\_stop、request\_stop、join三个函数）

工作过程：

声明一个tf.Coordinator的类，并将该类传入每一个创建的线程中去

启动的线程要一直查询should\_stop函数，为True时当前线程需要退出

每一个启动的线程都可以通过调用request\_stop函数来通知其他线程退出（当一个线程调用request\_stop函数时，should\_stop函数就好被设置为True，这样其他线程就可以同时终止）

import tensorflow as tf

import numpy as np

import threading

import time

# 线程中运行的程序，这个程序每隔1s判断是否需要停止并打印自己的ID

def MyLoop(coord,worker\_id):

# 使用tf.Coordinator类提供的协同工具判断当前线程是否需要停止

while not coord.should\_stop():

# 随机停止所有线程

if np.random.rand()<0.1:

print("stop from id: %d\n" % worker\_id)

# 调用coord.request\_stop()函数来通知其他线程停止

coord.request\_stop()

else:

# 打印当前线程的ID

print("working on id: %d" % worker\_id)

# 暂停1s

time.sleep(1)

# 声明一个tf.train.Coordinator类来协同多个线程

coord=tf.train.Coordinator()

# 声明创建5个线程

threads=[threading.Thread(target=MyLoop,args=(coord,i,)) for i in range(5)]

#启动所有的线程

for t in threads:t.start()

#等待所有线程退出

coord.join(threads)

输出：

working on id: 0

working on id: 1

working on id: 2

working on id: 3

working on id: 4

working on id: 0

working on id: 3

working on id: 2

working on id: 1

working on id: 4

working on id: 3

working on id: 0

working on id: 2

working on id: 4

working on id: 1

stop from id: 4

当所有线程启动后，每个线程会打印各自的ID，于是前4行打印了它们的ID，暂停1s之后，所有的线程将会第二遍打印ID。

tf.QueueRunner主要用于启动多个线程来操作同一个队列

启动的所有线程可以通过tf.Coordinator类来同一管理

下列代码展示了如何使用来管理多线程队列操作

import tensorflow as tf

# 声明一个先进先出的队列，队列中最多100个元素，类型为实数

queue = tf.FIFOQueue(100,"float")

# 定义队列的入队操作

enqueue\_op = queue.enqueue([tf.random\_normal([1])])

# 使用tf.train.QueueRunner来创建多个线程运行队列的入队操作

# tf.train.QueueRunner的第一个参数给出了被操作的队列

# [enqueue\_op] \* 5表示需要启动5个线程，每个线程中运行的是enqueue\_op的操作

qr = tf.train.QueueRunner(queue, [enqueue\_op] \* 5)

# 将定义过的QueueRunner加入TensorFlow计算图上指定的集合

# tf.train.QueueRunner函数没有指定集合，则加入默认集合tf.GraphKeys.QUEUE\_RUNNERS

# 下面的函数就是刚刚定义的qr加入默认的集合tf.GraphKeys.QUEUE\_RUNNERS

tf.train.add\_queue\_runner(qr)

# 定义出队操作

out\_tensor = queue.dequeue()

with tf.Session() as sess:

# 使用tf.train.Coordinator来协同启动的线程

coord = tf.train.Coordinator()

# 使用tf.train.QueueRunner()时，需要明确调用tf.train.start\_queue\_runners来启动所有线程

# 否则因为没有线程运行入队操作

# 当调用出队操作时，程序会一直等待入队操作被运行

# tf.train.start\_queue\_runners函数会默认启动tf.GraphKeys.QUEUE\_RUNNERS集合中所有的QueueRunner

# 因为该函数只支持启动指定集合中的QueueRunner

# 所以一般来说tf.train.add\_queue\_runner函数和tf.train.start\_queue\_runners函数会指定同一个集合

threads = tf.train.start\_queue\_runners(sess=sess, coord=coord)

#获取队列中的取值

for \_ in range(3):

print(sess.run(out\_tensor)[0])

# 使用tf.train.Coordinator来停止所有线程

coord.request\_stop()

coord.join(threads)

输出：

-0.574549

1.83348

-0.67578

7.3.2 输入文件队列

本节介绍如何使用TensorFlow中的队列管理输入文件列表

虽然一个TFRecord文件可以存储多个训练样本，但当训练数据量较大时，可以将数据分成多个TFRecord文件来提高处理效率。

获取一个正则表达式的所有文件：tf.train.match\_filenames\_once

进行有效的管理：tf.train.string\_input\_producer

该函数会使用初始化时提供的文件列表创建一个输入队列，输入队列中的原始元素为文件列表中的所有文件，创建好的输入队列可以作为文件读取函数的参数，每层调用文件读取函数时，该函数会先判断当前是否已经有打开的文件可读，如果没有或者打开的文件已经读完，则该函数会从输入队列中出队一个文件并从该文件中读取数据。

当shuffle=True时，文件在加入队列之前会被打乱顺序，所以出队顺序也是随机的；

随机打乱文件顺序以及加入输入队列的过程是一个单独的线程，不会影响获取文件的速度。

当输入队列中的所有文件都被处理完之后，会将初始化时提供的文件列表中的文件全部重新加入队列。

num\_epochs：限制加载初始文件列表的最大轮数

当设置为1时，计算完一轮之后，程序将自动停止。

神经网络模型测试时，所有测试数据仅仅需要使用一次即可，所以将其设置为1。

生成样例数据的简单程序：

import tensorflow as tf

# 创建TFRecord帮助函数

def \_int64\_feature(value):

return tf.train.Feature(int64\_list=tf.train.Int64List(value=[value]))

# 模拟海量数据CIA将数据写入不同的文件

# num\_shards 定义了总共写入多少个文件

# instances\_per\_shard定义了每个文件中有多少个数据

num\_shards=2

instances\_per\_shard=2

for i in range(num\_shards):

# 将数据分为多个文件时，可以将不同文件以类似0000n-of-0000m的后缀区分

# m：表示数据总共被存在了多少个文件中

# n：表示当前文件的编号

# 式样的方式既方便了通过正则表达式获取文件列表，又在文件名中加入了更多的信息

filename=("E:\\Opencv Image\\data.tfrecords-%.5d-of-%.5d" % (i,num\_shards))

writer=tf.python\_io.TFRecordWriter(filename)

# 将数据封装改成example结构并写入TFRecord文件

for j in range(instances\_per\_shard):

# example结构仅包含当前样例属于第几个文件以及是当前文件的第几个样本

example=tf.train.Example(features=tf.train.Features(feature={

'i':\_int64\_feature(i),

'j':\_int64\_feature(j)}))

writer.write(example.SerializeToString())

writer.close()

程序运行之后，会在指定目录下生成两个文件：

每个文件存储了两个样例，生成样例之后，以下代码展示了两个函数的使用方法

import tensorflow as tf

# 使用tf.train.match\_filenames\_once函数获取文件列表

files = tf.train.match\_filenames\_once("E:\\pycharm\\TensorFlow chap7\\data.tfrecords-\*")

filename\_queue = tf.train.string\_input\_producer(files, shuffle=False)

reader = tf.TFRecordReader()

\_, serialized\_example = reader.read(filename\_queue)

features = tf.parse\_single\_example(

serialized\_example,

features={

'i': tf.FixedLenFeature([], tf.int64),

'j': tf.FixedLenFeature([], tf.int64),

})

with tf.Session() as sess:

# # tf.global\_variables\_initializer().run() #报错

sess.run([tf.global\_variables\_initializer(),tf.local\_variables\_initializer()])

print(sess.run(files))

# 声明tf.train.Coordinator类来协同不同线程，并启动线程

coord = tf.train.Coordinator()

threads = tf.train.start\_queue\_runners(sess=sess, coord=coord)

# 多次执行获取数据的操作

for i in range(6):

print(sess.run([features['i'], features['j']]))

coord.request\_stop()

coord.join(threads)

输出：

[b'E:\\pycharm\\TensorFlow chap7\\data.tfrecords-00000-of-00002'

b'E:\\pycharm\\TensorFlow chap7\\data.tfrecords-00001-of-00002']

[0, 0]

[0, 1]

[1, 0]

[1, 1]

[0, 0]

[0, 1]

7.3.3 组合训练数据（batching）

将多个输入样例组织成一个batch可以提高模型训练的效率，所以在得到单个样例的预处理结果之后，还需要将其组织成batch，再提供给神经网络的输入层。

1. tf.train.batch：可以将样例组织成batch会生成一队列，队列的入队操作是生成单个样例的方法，每次出队会得到一个样例。

2. tf.train.shuffle\_batch：可以交给你样例组织成batch，会生成一队列，队列的入队操作是生成单个样例的方法，每次出队会得到一个样例，但是tf.train.shuffle\_batch函数会将数据顺序打乱。

tf.train.batch代码示例：

import tensorflow as tf

# 获取文件列表

files = tf.train.match\_filenames\_once("E:\\pycharm\\TensorFlow chap7\\data.tfrecords-\*")

# 创建文件输入队列

filename\_queue = tf.train.string\_input\_producer(files, shuffle=False)

# 读取并解析Example

reader = tf.TFRecordReader()

\_, serialized\_example = reader.read(filename\_queue)

features = tf.parse\_single\_example(

serialized\_example,

features={

'i': tf.FixedLenFeature([], tf.int64),

'j': tf.FixedLenFeature([], tf.int64)

})

# i代表特征向量，j代表标签

example, label = features['i'], features['j']

# 一个batch中的样例数

batch\_size = 3

# 文件队列中最多可以存储的样例个数

capacity = 1000 + 3 \* batch\_size

# 组合样例

example\_batch, label\_batch = tf.train.batch(

[example, label], batch\_size=batch\_size, capacity=capacity)

with tf.Session() as sess:

# 使用match\_filenames\_once需要用local\_variables\_initializer初始化一些变量

sess.run(

[tf.global\_variables\_initializer(),

tf.local\_variables\_initializer()])

# 用Coordinator协同线程，并启动线程

coord = tf.train.Coordinator()

threads = tf.train.start\_queue\_runners(coord=coord)

# 获取并打印组合之后的样例。真实问题中一般作为神经网路的输入

for i in range(2):

cur\_example\_batch, cur\_label\_batch = sess.run(

[example\_batch, label\_batch])

print(cur\_example\_batch, cur\_label\_batch)

coord.request\_stop()

coord.join(threads)

输出：

[0 0 1] [0 1 0]

[1 0 0] [1 0 1]

# tf.train.batch 函数可以将单个的数据组织成3个一组的batch

# 在example，lable中读到的数据依次为：

example：0, lable：0

example：0, lable：1

example：1, lable：0

example：1, lable：1

# 这是因为函数不会随机打乱顺序，所以组合之后得到的数据组合成了上面给出的输出

tf.train.shuffle\_batch代码示例如下：

import tensorflow as tf

# 获取文件列表

files = tf.train.match\_filenames\_once("E:\\pycharm\\TensorFlow chap7\\data.tfrecords-\*")

# 创建文件输入队列

filename\_queue = tf.train.string\_input\_producer(files, shuffle=False)

# 读取并解析Example

reader = tf.TFRecordReader()

\_, serialized\_example = reader.read(filename\_queue)

features = tf.parse\_single\_example(

serialized\_example,

features={

'i': tf.FixedLenFeature([], tf.int64),

'j': tf.FixedLenFeature([], tf.int64)

})

# i代表特征向量，j代表标签

example, label = features['i'], features['j']

# 一个batch中的样例数

batch\_size = 3

# 文件队列中最多可以存储的样例个数

capacity = 1000 + 3 \* batch\_size

# 组合样例

# `min\_after\_dequeue` 是该函数特有的参数，参数限制了出队时队列中元素的最少个数，

# 但当队列元素个数太少时，随机的意义就不大了

example\_batch,label\_batch = tf.train.shuffle\_batch(

[example,label],batch\_size=batch\_size,

capacity=capacity,min\_after\_dequeue=30)

with tf.Session() as sess:

# 使用match\_filenames\_once需要用local\_variables\_initializer初始化一些变量

sess.run( [tf.global\_variables\_initializer(),tf.local\_variables\_initializer()])

# 用Coordinator协同线程，并启动线程

coord = tf.train.Coordinator()

threads = tf.train.start\_queue\_runners(coord=coord)

# 获取并打印组合之后的样例。真实问题中一般作为神经网路的输入

for i in range(2):

cur\_example\_batch, cur\_label\_batch = sess.run(

[example\_batch, label\_batch])

print(cur\_example\_batch, cur\_label\_batch)

coord.request\_stop()

coord.join(threads)

输出：

[0 1 1] [0 1 0]

[1 0 0] [0 0 1]

7.3.4 输入数据处理框架

本节给出以上步骤整合之后的代码

框架主要是三方面的内容：

TFRecord 输入数据格式

图像数据处理

多线程输入数据处理

以下代码只是描绘了一个输入数据处理的框架，需要根据实际使用环境进行修改

import tensorflow as tf

# 创建文件列表

files = tf.train.match\_filenames\_once("E:\\pycharm\\TensorFlow chap7\\data.tfrecords-\*")

# 创建输入文件队列

filename\_queue = tf.train.string\_input\_producer(files, shuffle=False)

# 解析数据。假设image是图像数据，label是标签，height、width、channels给出了图片的维度

reader = tf.TFRecordReader()

\_, serialized\_example = reader.read(filename\_queue)

features = tf.parse\_single\_example(

serialized\_example,

features={

'image': tf.FixedLenFeature([], tf.string),

'label': tf.FixedLenFeature([], tf.int64),

'height': tf.FixedLenFeature([], tf.int64),

'width': tf.FixedLenFeature([], tf.int64),

'channels': tf.FixedLenFeature([], tf.int64)

})

image, label = features['image'], features['label']

height, width = features['height'], features['width']

channels = features['channels']

# 从原始图像中解析出像素矩阵，并还原图像

decoded\_image = tf.decode\_raw(image, tf.uint8)

decoded\_image.set\_shape([height, width, channels])

# 定义神经网络输入层图片的大小

image\_size = 299

# preprocess\_for\_train函数是对图片进行预处理的函数

distorted\_image = preprocess\_for\_train(decoded\_image, image\_size, image\_size,None)

# 组合成batch

min\_after\_dequeue = 10000

batch\_size = 100

capacity = min\_after\_dequeue + 3 \* batch\_size

image\_batch, label\_batch = tf.train.shuffle\_batch(

[distorted\_image, label],

batch\_size=batch\_size,

capacity=capacity,

min\_after\_dequeue=min\_after\_dequeue)

# 定义神经网络的结构及优化过程

logit = inference(image\_batch)

loss = calc\_loss(logit, label\_batch)

train\_step = tf.train.GradientDescentOptimizer(learning\_rate).minimize(loss)

with tf.Session() as sess:

sess.run(

[tf.global\_variables\_initializer(),

tf.local\_variables\_initializer()])

coord = tf.train.Coordinator()

threads = tf.train.start\_queue\_runners(coord=coord)

# 神经网络训练过程

for i in range(TRAINING\_ROUNDS):

sess.run(train\_step)

coord.request\_stop()

coord.join()

总结

对于输入数据的处理，大体上流程都差不多，可以归结如下：

将数据转为 TFRecord 格式的多个文件

用 tf.train.match\_filenames\_once() 创建文件列表（图中为{A,B,C}）

用 tf.train.string\_input\_producer() 创建输入文件队列，可以将输入文件顺序随机打乱，并加入输入队列（是否打乱为可选项，该函数也会生成并维护一个输入文件队列，不同进程中的文件读取函数可以共享这个输入文件队列）

用 tf.TFRecordReader() 读取文件中的数据

用 tf.parse\_single\_example() 解析数据

对数据进行解码及预处理

用 tf.train.shuffle\_batch() 将数据组合成 batch

将 batch 用于训练

上图就是以上代码中输入数据处理的全部流程