专业英语期末大作业：CIFAR-10

学号：10161511322 姓名：洪语晨

目录

Dataset : The CIFAR-10 dataset

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

Goals

To build a relatively small convolution neural network for recognizing imagine.

Model Architecture

The model is a multi-layer architecture consisting of alternating convolutions and nonlinearities. These layers are followed by fully connected layers leading into a softmax classfier.

代码组成：

文件	作用
Cifar10_input.py	读取原始的CIFAR-10二进制格式文件
Cifar10.py	建立 CIFAR-10 网络模型
Cifar10_train.py	在单块 CPU 上训练模型
Cifar10_eval.py	在测试集上评估 CIFAR-10 的表现

核心的运算包括卷积、relu 激活、最大池化和局部相应归一化

网络训练过程中包括可视化操作

实现学习效率衰减策略来训练，采用指数衰减(exponential_decay)的方法。

使用队列操作获取输入数据。

数据集介绍

CIFAR-10 数据集分类是机器学习领域很经典的任务，该任务旨在把 32x32 的 RGB 图像分成十类：

airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck

解压数据集，其中包括 6 个文件：

data_batch_1,data_batch_2,data_batch_3,data_batch_4,data_batch_5,test_batch

为 5 个训练文件，一个测试文件，每个文件里包含 10000 个样本，共计 50000 个训练样本，10000 个测试样本。

文件中数据结构为：

```
1<1 x label><3072 x pixel>
2...
3<1 x label><3072 x pixel>
```

其中，第一个字节为标签，范围 0-9 代表 10 类，接下来 3072 个字节代表着图像的像素值，前 1024 个字节是 red 通道的值，接着 1024 个字节是 green 通道值，最后 1024 个字节是 blue 通道值。字节排列是以行为主的顺序。

具体步骤

Step1：读取数据(cifar10_input.py)

主要函数：

read_cifar10():从文件名队列读取二进制数据并提取出单张图片数据

_generate_image_and_label_batch():利用单张图片数据生成 batch 数据

Distorted_inputs():构建训练数据并进行预处理

Inputs():构建测试数据并进行预处理

具体代码如下：

```
NUM_CLASSES = 10
NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN = 50000
NUM_EXAMPLES_PER_EPOCH_FOR_EVAL = 10000
```

定义了图片尺寸为 24，类别数目为 10，训练样本数为 50000，测试样本数目为 10000.

函数 read_cifar10()代码如下：

```
class CIFAR10Record(object): =
    pass
  result = CIFAR10Record()
  label_bytes = 1
  result.height = 32
  result.width = 32
  result.depth = 3
  image_bytes = result.height * result.width * result.depth


  record_bytes = label_bytes + image_bytes
  reader = tf.FixedLengthRecordReader(record_bytes=record_bytes)
  result.key, value = reader.read(filename_queue)
  record_bytes = tf.decode_raw(value, tf.uint8)
  result.label = tf.cast(
      tf.strided_slice(record_bytes, [0], [label_bytes]), tf.int32)
      depth_major = tf.reshape(
      tf.strided_slice(record_bytes, [label_bytes],
                                [label_bytes + image_bytes]),
      [result.depth, result.height, result.width])
  result.uint8image = tf.transpose(depth_major, [1, 2, 0])
```

该函数定义了固定长度为 record_bytes 的数据，从文件名队列中获取文件并读出

单张图像数据，并用解码器把字符串类型转化为 uint8 类型数据。通过观察数据集结构，其中第一个字节代表着标签数据，因此，将其格式从 unit8 转化为 int32 根据 CIFAR-0 的数据排列[depth*height*width]将它转化为[depth,height,width]形状的张量，使用 tf.reshape()。

函数 2：_generate_image_and_label_batch(image,label,min_queue_examples

生成图像的标签和 batch 数据

```
num_preprocess_threads = 16
  if shuffle:
          images, label_batch = tf.train.shuffle_batch(
          [image, label],
          batch_size=batch_size,
          num_threads=num_preprocess_threads,
          capacity=min_queue_examples + 3 * batch_size,
          min_after_dequeue=min_queue_examples)
  else:
     images, label_batch = tf.train.batch(
          [image, label],
          batch_size=batch_size,
          num_threads=num_preprocess_threads,
          capacity=min_queue_examples + 3 * batch_size)
     tf.summary.image('images', images)
```

创建一个样本队列，根据需要决定是否对其进行随机排序，每次从队列中取出 batch_size 个图像数据

3.函数 distorted_inputs()

对训练数据图像预处理，包括多个随机失真操作。

tf.radom_crop(),将原始的 32*32 的图片随机剪裁为 24*24

tf.image.random_flip_left_right(),随机左右翻转

tf.image.random_brightness(),tf.imagin.random_contrast:随机改变图片亮度和对比度

tf.image.per_image_standardization()图像进行归一化

函数 3：inputs（）

首先根据输入的参数，判断是测试集还是训练集

图像预处理之后，设置成张量形状，设置 min_after_dequeue 参数为 20000(训练数据)，4000(测试数据)，保证足够随机性。

Step2：建立模型(cifar10.py)

全局常量：

```
IMAGE_SIZE = cifar10_input.IMAGE_SIZE # 28

NUM_CLASSES = cifar10_input.NUM_CLASSES # 10

NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN =

cifar10_input.NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN # 50000
```

```
NUM_EXAMPLES_PER_EPOCH_FOR_EVAL =
cifar10_input.NUM_EXAMPLES_PER_EPOCH_FOR_EVAL # 10000
```

```
MOVING_AVERAGE_DECAY = 0.9999        # 计算滑动平均(moving average)时
的衰减(decay)
NUM_EPOCHS_PER_DECAY = 350.0         # 学习率衰减的 epochs
LEARNING_RATE_DECAY_FACTOR = 0.1   # 学习率衰减因子
INITIAL_LEARNING_RATE = 0.1          # 初始学习率
```

函数 1：_activation_summary():为激活函数创造可视化 summary

```
tensor_name = re.sub('%s_[0-9]*/' % TOWER_NAME, '', x.op.name)
  tf.summary.histogram(tensor_name + '/activations', x)
  tf.summary.scalar(tensor_name + '/sparsity',
                                       tf.nn.zero_fraction(x))
```

函数 2：_variable_with_weight_decay(name,shape,stddev,wd)

其中 wd 是正则化系数

```
dtype = tf.float16 if FLAGS.use_fp16 else tf.float32
  var = _variable_on_cpu(
        name,
        shape,
        tf.truncated_normal_initializer(stddev=stddev, dtype=dtype))
```

```
  if wd is not None:
      weight_decay = tf.multiply(tf.nn.l2_loss(var), wd, name='weight_loss')
     tf.add_to_collection('losses', weight_decay)
     return var
```

函数 3：distorted_inputs()

对 cifar10_input.py 里的 distorted_inputs()进行封装

函数 4：inference(images)

搭建 CIFAR-10 模型

Conv1

```
with tf.variable_scope('conv1') as scope:
      kernel = _variable_with_weight_decay('weights',
                                           shape=[5, 5, 3, 64],
                                           stddev=5e-2,
                                           wd=0.0)
      conv = tf.nn.conv2d(images, kernel, [1, 1, 1, 1], padding='SAME')
      biases = _variable_on_cpu('biases', [64], tf.constant_initializer(0.0))
     pre_activation = tf.nn.bias_add(conv, biases)
      conv1 = tf.nn.relu(pre_activation, name=scope.name)
      _activation_summary(conv1)
```

Pool1

```
pool1 = tf.nn.max_pool(conv1, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
                       padding='SAME', name='pool1')
```

norm1：

```
norm1 = tf.nn.lrn(pool1, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                  name='norm1')
```

conv2

```
with tf.variable_scope('conv2') as scope:
    kernel = _variable_with_weight_decay('weights',
                                         shape=[5, 5, 64, 64],
                                         stddev=5e-2,
                                         wd=0.0)
    conv = tf.nn.conv2d(norm1, kernel, [1, 1, 1, 1], padding='SAME')
    biases = _variable_on_cpu('biases', [64], tf.constant_initializer(0.1))
    pre_activation = tf.nn.bias_add(conv, biases)
    conv2 = tf.nn.relu(pre_activation, name=scope.name)
    _activation_summary(conv2)
```

Norm2

```
norm2 = tf.nn.lrn(conv2, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                  name='norm2')
```

pool2

```
pool2 = tf.nn.max_pool(norm2, ksize=[1, 3, 3, 1],
                       strides=[1, 2, 2, 1], padding='SAME', name='pool2')
```

local3

```
with tf.variable_scope('local3') as scope:
```

```
    reshape = tf.reshape(pool2, [FLAGS.batch_size, -1])
    dim = reshape.get_shape()[1].value
    weights = _variable_with_weight_decay('weights', shape=[dim, 384],
                                          stddev=0.04, wd=0.004)
    biases = _variable_on_cpu('biases', [384], tf.constant_initializer(0.1))
    local3 = tf.nn.relu(tf.matmul(reshape, weights) + biases, name=scope.name)
    _activation_summary(local3)
```

Local4

```
  with tf.variable_scope('local4') as scope:
    weights = _variable_with_weight_decay('weights', shape=[384, 192],
                                          stddev=0.04, wd=0.004)
    biases = _variable_on_cpu('biases', [192], tf.constant_initializer(0.1))
    local4 = tf.nn.relu(tf.matmul(local3, weights) + biases, name=scope.name)
    _activation_summary(local4)
```

线性层：

```
with tf.variable_scope('softmax_linear') as scope:
    weights = _variable_with_weight_decay('weights', [192, NUM_CLASSES],
                                          stddev=1/192.0, wd=0.0)
    biases = _variable_on_cpu('biases', [NUM_CLASSES],
                              tf.constant_initializer(0.0))
    softmax_linear = tf.add(tf.matmul(local4, weights), biases,
name=scope.name)
```

```
    _activation_summary(softmax_linear)
  return softmax_linear
```

函数 5：loss(logits,labels)

计算 batch 的平均交叉熵损失

如果是 label 是 one-hot 码则用 tf.nn.softmax_cross_entropy_with_logits()

这里 label 是从 0-9 的数字来表示，则用 tf.nn.sparse_softmax_cross_entropy_with_logits()

```
labels = tf.cast(labels, tf.int64)
  cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
      labels=labels, logits=logits, name='cross_entropy_per_example')
  cross_entropy_mean = tf.reduce_mean(cross_entropy, name='cross_entropy')
  tf.add_to_collection('losses', cross_entropy_mean)
```

函数 6：train(total_loss,global_step)

执行顺序如下：

计算损失的滑动平均——>计算学习率——>计算梯度——>更新参数——>计算训练变量的滑动平均——>train_op(返回的值)

通过执行 train_op 来依次执行之前的所有步骤

学习率随着迭代次数指数衰减

```
lr = tf.train.exponential_decay(INITIAL_LEARNING_RATE,
```

```
                                        global_step,
                                        decay_steps,
                                        LEARNING_RATE_DECAY_FACTOR,
                                        staircase=True)
    tf.summary.scalar('learning_rate', lr)
```

计算梯度

```
    with tf.control_dependencies([loss_averages_op]):
        opt = tf.train.GradientDescentOptimizer(lr)
        grads = opt.compute_gradients(total_loss)
```

应用梯度更新参数

```
apply_gradient_op = opt.apply_gradients(grads, global_step=global_step)
```

为所有训练变量添加滑动平均

```
variable_averages = tf.train.ExponentialMovingAverage(
            MOVING_AVERAGE_DECAY, global_step)
    variables_averages_op = variable_averages.apply(tf.trainable_variables())
    with tf.control_dependencies([apply_gradient_op, variables_averages_op]):
        train_op = tf.no_op(name='train')
```

step3：评估模型(cifar10_eval.py)

Evaluate()负责创建和维护整个评估过程：

获得测试数据

搭建神经网络模型

创建 saver，saver 负责恢复 shadow variable 的值并赋给 variable

每隔固定的间隔，运行一次 eval_once()

Eval_once()负责完成一次评估，步骤：

从 checkpoint 中取出最新的模型

运行神经网络，对测试集的数据按批次进行预测

计算整个测试集的预测精度

运行结果：

训练模型运行结果

```
2018-06-16 21:44:05.599500: step 330, loss = 3.79 (202.4 examples/sec; 0.632 sec/batch)
2018-06-16 21:44:11.961500: step 340, loss = 3.66 (201.2 examples/sec; 0.636 sec/batch)
2018-06-16 21:44:18.316500: step 350, loss = 3.58 (201.4 examples/sec; 0.636 sec/batch)
2018-06-16 21:44:24.758500: step 360, loss = 3.43 (198.7 examples/sec; 0.644 sec/batch)
2018-06-16 21:44:31.963500: step 370, loss = 3.54 (177.7 examples/sec; 0.720 sec/batch)
2018-06-16 21:44:39.166500: step 380, loss = 3.56 (177.7 examples/sec; 0.720 sec/batch)
2018-06-16 21:44:45.509500: step 390, loss = 3.28 (201.8 examples/sec; 0.634 sec/batch)
2018-06-16 21:44:51.998500: step 400, loss = 3.28 (197.3 examples/sec; 0.649 sec/batch)
2018-06-16 21:44:58.343500: step 410, loss = 3.38 (201.7 examples/sec; 0.635 sec/batch)
2018-06-16 21:45:04.697500: step 420, loss = 3.38 (201.4 examples/sec; 0.635 sec/batch)
2018-06-16 21:45:11.042500: step 430, loss = 3.26 (201.7 examples/sec; 0.634 sec/batch)
2018-06-16 21:45:17.401500: step 440, loss = 3.17 (201.3 examples/sec; 0.636 sec/batch)
2018-06-16 21:45:23.737500: step 450, loss = 3.45 (202.0 examples/sec; 0.634 sec/batch)
2018-06-16 21:45:30.099500: step 460, loss = 3.02 (201.2 examples/sec; 0.636 sec/batch)
2018-06-16 21:45:36.433500: step 470, loss = 3.15 (202.1 examples/sec; 0.633 sec/batch)
2018-06-16 21:45:42.793500: step 480, loss = 3.15 (201.3 examples/sec; 0.636 sec/batch)
2018-06-16 21:45:49.122500: step 490, loss = 3.17 (202.2 examples/sec; 0.633 sec/batch)
2018-06-16 21:45:55.540500: step 500, loss = 3.13 (199.4 examples/sec; 0.642 sec/batch)
2018-06-16 21:46:01.980500: step 510, loss = 3.21 (198.8 examples/sec; 0.644 sec/batch)
2018-06-16 21:46:08.547500: step 520, loss = 3.01 (194.9 examples/sec; 0.657 sec/batch)
2018-06-16 21:46:14.949500: step 530, loss = 3.12 (199.9 examples/sec; 0.640 sec/batch)
2018-06-16 21:46:22.064500: step 540, loss = 3.18 (179.9 examples/sec; 0.712 sec/batch)
```

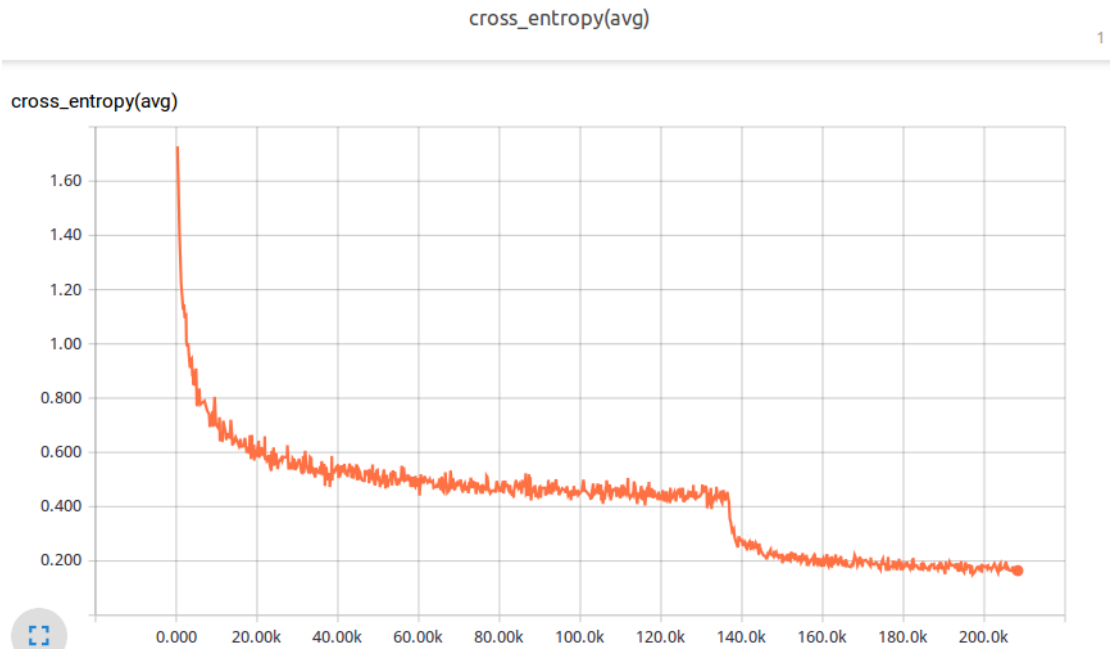
```
2018-06-18 17:32:32.589000: step 43440, loss = 0.68 (213.9 examples/sec; 0.598 sec/batch)
2018-06-18 17:32:38.569000: step 43450, loss = 0.73 (214.0 examples/sec; 0.598 sec/batch)
2018-06-18 17:32:44.556000: step 43460, loss = 0.82 (213.8 examples/sec; 0.599 sec/batch)
2018-06-18 17:32:50.537000: step 43470, loss = 0.88 (214.0 examples/sec; 0.598 sec/batch)
2018-06-18 17:32:56.518000: step 43480, loss = 0.79 (214.0 examples/sec; 0.598 sec/batch)
2018-06-18 17:33:02.526000: step 43490, loss = 0.96 (213.0 examples/sec; 0.601 sec/batch)
2018-06-18 17:33:10.958000: step 43500, loss = 0.66 (151.8 examples/sec; 0.843 sec/batch)
2018-06-18 17:33:21.773000: step 43510, loss = 0.68 (118.4 examples/sec; 1.081 sec/batch)
2018-06-18 17:33:27.809000: step 43520, loss = 0.81 (212.0 examples/sec; 0.604 sec/batch)
2018-06-18 17:33:33.795000: step 43530, loss = 0.72 (213.8 examples/sec; 0.599 sec/batch)
2018-06-18 17:33:39.780000: step 43540, loss = 0.83 (213.9 examples/sec; 0.598 sec/batch)
2018-06-18 17:33:45.746000: step 43550, loss = 0.67 (214.5 examples/sec; 0.597 sec/batch)
2018-06-18 17:33:51.724000: step 43560, loss = 0.68 (214.1 examples/sec; 0.598 sec/batch)
2018-06-18 17:33:57.770000: step 43570, loss = 0.80 (211.7 examples/sec; 0.605 sec/batch)
2018-06-18 17:34:03.828000: step 43580, loss = 0.77 (211.3 examples/sec; 0.606 sec/batch)
2018-06-18 17:34:09.898000: step 43590, loss = 0.83 (210.9 examples/sec; 0.607 sec/batch)
2018-06-18 17:34:15.980000: step 43600, loss = 0.89 (210.5 examples/sec; 0.608 sec/batch)
2018-06-18 17:34:21.958000: step 43610, loss = 0.71 (214.1 examples/sec; 0.598 sec/batch)
2018-06-18 17:34:27.980000: step 43620, loss = 0.59 (212.6 examples/sec; 0.602 sec/batch)
2018-06-18 17:34:33.969000: step 43630, loss = 0.66 (213.7 examples/sec; 0.599 sec/batch)
2018-06-18 17:34:39.970000: step 43640, loss = 0.82 (213.3 examples/sec; 0.600 sec/batch)
2018-06-18 17:34:46.275000: step 43650, loss = 0.71 (203.0 examples/sec; 0.631 sec/batch)
2018-06-18 17:34:52.318000: step 43660, loss = 0.73 (211.8 examples/sec; 0.604 sec/batch)
2018-06-18 17:34:58.346000: step 43670, loss = 1.09 (212.3 examples/sec; 0.603 sec/batch)
2018-06-18 17:35:04.406000: step 43680, loss = 0.76 (211.2 examples/sec; 0.606 sec/batch)
2018-06-18 17:35:10.445000: step 43690, loss = 0.75 (212.0 examples/sec; 0.604 sec/batch)
2018-06-18 17:35:16.615000: step 43700, loss = 0.65 (207.5 examples/sec; 0.617 sec/batch)
2018-06-18 17:35:23.704000: step 43710, loss = 0.65 (180.6 examples/sec; 0.709 sec/batch)
2018-06-18 17:35:29.859000: step 43720, loss = 0.78 (208.0 examples/sec; 0.615 sec/batch)
```

对模型的评估：

```
2018-06-18 10:05:52.725600: precision @ 1 = 0.101
2018-06-18 10:11:07.439600: precision @ 1 = 0.101
2018-06-18 10:16:22.715600: precision @ 1 = 0.607
2018-06-18 10:21:37.779600: precision @ 1 = 0.607
2018-06-18 10:26:52.503600: precision @ 1 = 0.693
2018-06-18 10:32:07.658600: precision @ 1 = 0.693
2018-06-18 10:37:23.179600: precision @ 1 = 0.735
2018-06-18 10:42:38.218600: precision @ 1 = 0.735
2018-06-18 10:47:53.504600: precision @ 1 = 0.759
2018-06-18 10:53:08.649600: precision @ 1 = 0.759
2018-06-18 10:58:23.811600: precision @ 1 = 0.771
2018-06-18 11:03:38.875600: precision @ 1 = 0.771
2018-06-18 11:08:53.499600: precision @ 1 = 0.782
2018-06-18 11:14:08.304600: precision @ 1 = 0.782
2018-06-18 11:19:23.208600: precision @ 1 = 0.792
2018-06-18 11:24:38.165600: precision @ 1 = 0.792
2018-06-18 11:29:53.261600: precision @ 1 = 0.799
2018-06-18 11:35:08.014600: precision @ 1 = 0.799
2018-06-18 11:40:22.626600: precision @ 1 = 0.804
2018-06-18 11:45:36.687600: precision @ 1 = 0.804
2018-06-18 11:50:51.560600: precision @ 1 = 0.809
2018-06-18 11:56:06.262600: precision @ 1 = 0.813
2018-06-18 12:01:21.137600: precision @ 1 = 0.813
2018-06-18 12:06:35.797600: precision @ 1 = 0.817
```

```
2018-06-18 15:00:26.986000: precision @ 1 = 0.843
2018-06-18 15:05:43.132000: precision @ 1 = 0.843
2018-06-18 15:10:59.298000: precision @ 1 = 0.843
2018-06-18 15:16:15.519000: precision @ 1 = 0.845
2018-06-18 15:21:31.681000: precision @ 1 = 0.845
2018-06-18 15:26:47.707000: precision @ 1 = 0.847
2018-06-18 15:32:04.180000: precision @ 1 = 0.847
2018-06-18 15:37:21.202000: precision @ 1 = 0.848
2018-06-18 15:42:37.546000: precision @ 1 = 0.848
2018-06-18 15:47:53.764000: precision @ 1 = 0.847
2018-06-18 15:53:10.305000: precision @ 1 = 0.847
2018-06-18 15:58:26.466000: precision @ 1 = 0.846
2018-06-18 16:03:42.732000: precision @ 1 = 0.846
2018-06-18 16:08:59.622000: precision @ 1 = 0.845
2018-06-18 16:14:15.641000: precision @ 1 = 0.845
2018-06-18 16:19:31.783000: precision @ 1 = 0.846
2018-06-18 16:24:48.255000: precision @ 1 = 0.846
2018-06-18 16:30:04.127000: precision @ 1 = 0.847
2018-06-18 16:35:20.276000: precision @ 1 = 0.847
2018-06-18 16:40:37.168000: precision @ 1 = 0.848
2018-06-18 16:45:52.960000: precision @ 1 = 0.848
2018-06-18 16:51:09.291000: precision @ 1 = 0.849
2018-06-18 16:56:26.248000: precision @ 1 = 0.850
2018-06-18 17:01:42.402000: precision @ 1 = 0.850
2018-06-18 17:06:58.468000: precision @ 1 = 0.851
2018-06-18 17:12:15.184000: precision @ 1 = 0.851
2018-06-18 17:17:31.928000: precision @ 1 = 0.851
2018-06-18 17:22:48.175000: precision @ 1 = 0.851
```

Tensorboard



Precision @ 1

1