CIFAR10-example

August 3, 2017

1 CIFAR10 Training and Evalutaion Notebook

1.0.1 Runs a CNN on CIFAR10 and evaluates the accuracy.

Code adapted from tensorflow's CIFAR10 tutorial: https://www.tensorflow.org/tutorials/deep_cnn

```
In [1]: from __future__ import absolute_import
        from __future__ import division
        from __future__ import print_function

        from datetime import datetime
        import time
        import math
        import os
        import re
        import sys
        import tarfile


        from six.moves import urllib
        import tensorflow as tf
        import numpy as np
```

```
In [2]: # Global constants describing the CIFAR-10 data set.
        IMAGE_SIZE = 24
        NUM_CLASSES = 10
        NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN = 50000
        NUM_EXAMPLES_PER_EPOCH_FOR_EVAL = 10000

        # Constants describing the training process.
        MOVING_AVERAGE_DECAY = 0.9999     # The decay to use for the moving average.
        NUM_EPOCHS_PER_DECAY = 350.0      # Epochs after which learning rate decays.
        LEARNING_RATE_DECAY_FACTOR = 0.1  # Learning rate decay factor.
        INITIAL_LEARNING_RATE = 0.1       # Initial learning rate.

        # Program parameters
        f_batch_size = 128 #Number of images to process in a batch.
```

```
f_use_fp16 = False #Train the model using fp16.
f_max_steps = 100 #Number of batches to run.
f_log_device_placement = False #Whether to log device placement.
f_log_frequency = 10 #How often to log results to the console.
f_eval_data = 'test' #Either 'test' or 'train_eval'.
f_eval_interval_secs = 30 #How often to run the eval
f_num_examples = 10000 #Number of examples to run.
f_run_once = True #Whether to run eval only once.

# If a model is trained with multiple GPUs, prefix all Op names with tower_name
# to differentiate the operations. Note that this prefix is removed from the
# names of the summaries when visualizing a model.
TOWER_NAME = 'tower'

DATA_URL = 'http://www.cs.toronto.edu/~kriz/cifar-10-binary.tar.gz'

# Paths
f_data_dir = '../CIFAR10_data/cifar10_data' #CIFAR-10 data
f_eval_dir = '../CIFAR10_data/cifar10_eval' #event logs
f_train_dir = '../CIFAR10_data/cifar10_train' #event logs and checkpoint
f_checkpoint_dir = '../CIFAR10_data/cifar10_train' #read model checkpoint
```

1.0.2 CIFAR-10 Input

Functions used to prepare the CIFAR10 images

```
In [3]: def read_cifar10(filename_queue):
          """Reads and parses examples from CIFAR10 data files.

          Recommendation: if you want N-way read parallelism, call this function
          N times.  This will give you N independent Readers reading different
          files & positions within those files, which will give better mixing of
          examples.

          Args:
            filename_queue: A queue of strings with the filenames to read from.

          Returns:
            An object representing a single example, with the following fields:
              height: number of rows in the result (32)
              width: number of columns in the result (32)
              depth: number of color channels in the result (3)
              key: a scalar string Tensor describing the filename & record number
                for this example.
              label: an int32 Tensor with the label in the range 0..9.
              uint8image: a [height, width, depth] uint8 Tensor with the image data
          """
```

```
  class CIFAR10Record(object):
    pass
  result = CIFAR10Record()

  # Dimensions of the images in the CIFAR-10 dataset.
  # See http://www.cs.toronto.edu/~kriz/cifar.html for a description of the
  # input format.
  label_bytes = 1  # 2 for CIFAR-100
  result.height = 32
  result.width = 32
  result.depth = 3
  image_bytes = result.height * result.width * result.depth
  # Every record consists of a label followed by the image, with a
  # fixed number of bytes for each.
  record_bytes = label_bytes + image_bytes

  # Read a record, getting filenames from the filename_queue.  No
  # header or footer in the CIFAR-10 format, so we leave header_bytes
  # and footer_bytes at their default of 0.
  reader = tf.FixedLengthRecordReader(record_bytes=record_bytes)
  result.key, value = reader.read(filename_queue)

  # Convert from a string to a vector of uint8 that is record_bytes long.
  record_bytes = tf.decode_raw(value, tf.uint8)

  # The first bytes represent the label, which we convert from uint8->int32.
  result.label = tf.cast(
      tf.strided_slice(record_bytes, [0], [label_bytes]), tf.int32)

  # The remaining bytes after the label represent the image, which we reshape
  # from [depth * height * width] to [depth, height, width].
  depth_major = tf.reshape(
      tf.strided_slice(record_bytes, [label_bytes],
                       [label_bytes + image_bytes]),
      [result.depth, result.height, result.width])
  # Convert from [depth, height, width] to [height, width, depth].
  result.uint8image = tf.transpose(depth_major, [1, 2, 0])

  return result


def _generate_image_and_label_batch(image, label, min_queue_examples,
                                    batch_size, shuffle):
  """Construct a queued batch of images and labels.

  Args:
    image: 3-D Tensor of [height, width, 3] of type.float32.
    label: 1-D Tensor of type.int32
    min_queue_examples: int32, minimum number of samples to retain
      in the queue that provides of batches of examples.
    batch_size: Number of images per batch.
    shuffle: boolean indicating whether to use a shuffling queue.

  Returns:
    images: Images. 4D tensor of [batch_size, height, width, 3] size.
    labels: Labels. 1D tensor of [batch_size] size.
  """
  # Create a queue that shuffles the examples, and then
  # read 'batch_size' images + labels from the example queue.
  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)

  # Display the training images in the visualizer.
  tf.summary.image('images', images)

  return images, tf.reshape(label_batch, [batch_size])


def distorted_inputs2(data_dir, batch_size):
  """Construct distorted input for CIFAR training using the Reader ops.

  Args:
    data_dir: Path to the CIFAR-10 data directory.
    batch_size: Number of images per batch.

  Returns:
    images: Images. 4D tensor of [batch_size, IMAGE_SIZE, IMAGE_SIZE, 3] size.
    labels: Labels. 1D tensor of [batch_size] size.
  """
  filenames = [os.path.join(data_dir, 'data_batch_%d.bin' % i)
               for i in range(1, 6)]
  for f in filenames:
    if not tf.gfile.Exists(f):
      raise ValueError('Failed to find file: ' + f)
```

```
# Create a queue that produces the filenames to read.
filename_queue = tf.train.string_input_producer(filenames)

# Read examples from files in the filename queue.
read_input = read_cifar10(filename_queue)
reshaped_image = tf.cast(read_input.uint8image, tf.float32)

height = IMAGE_SIZE
width = IMAGE_SIZE

# Image processing for training the network. Note the many random
# distortions applied to the image.

# Randomly crop a [height, width] section of the image.
distorted_image = tf.random_crop(reshaped_image, [height, width, 3])

# Randomly flip the image horizontally.
distorted_image = tf.image.random_flip_left_right(distorted_image)

# Because these operations are not commutative, consider randomizing
# the order their operation.
# NOTE: since per_image_standardization zeros the mean and makes
# the stddev unit, this likely has no effect see tensorflow#1458.
distorted_image = tf.image.random_brightness(distorted_image,
                                             max_delta=63)
distorted_image = tf.image.random_contrast(distorted_image,
                                           lower=0.2, upper=1.8)

# Subtract off the mean and divide by the variance of the pixels.
float_image = tf.image.per_image_standardization(distorted_image)

# Set the shapes of tensors.
float_image.set_shape([height, width, 3])
read_input.label.set_shape([1])

# Ensure that the random shuffling has good mixing properties.
min_fraction_of_examples_in_queue = 0.4
min_queue_examples = int(NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN *
                         min_fraction_of_examples_in_queue)
print ('Filling queue with %d CIFAR images before starting to train. '
       'This will take a few minutes.' % min_queue_examples)

# Generate a batch of images and labels by building up a queue of examples.
return _generate_image_and_label_batch(float_image, read_input.label,
                                       min_queue_examples, batch_size,
                                       shuffle=True)
```

```
def inputs2(eval_data, data_dir, batch_size):
  """Construct input for CIFAR evaluation using the Reader ops.

  Args:
    eval_data: bool, indicating if one should use the train or eval data set.
    data_dir: Path to the CIFAR-10 data directory.
    batch_size: Number of images per batch.

  Returns:
    images: Images. 4D tensor of [batch_size, IMAGE_SIZE, IMAGE_SIZE, 3] size.
    labels: Labels. 1D tensor of [batch_size] size.
  """
  if not eval_data:
    filenames = [os.path.join(data_dir, 'data_batch_%d.bin' % i)
                 for i in range(1, 6)]
    num_examples_per_epoch = NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN
  else:
    filenames = [os.path.join(data_dir, 'test_batch.bin')]
    num_examples_per_epoch = NUM_EXAMPLES_PER_EPOCH_FOR_EVAL

  for f in filenames:
    if not tf.gfile.Exists(f):
      raise ValueError('Failed to find file: ' + f)

  # Create a queue that produces the filenames to read.
  filename_queue = tf.train.string_input_producer(filenames)

  # Read examples from files in the filename queue.
  read_input = read_cifar10(filename_queue)
  reshaped_image = tf.cast(read_input.uint8image, tf.float32)

  height = IMAGE_SIZE
  width = IMAGE_SIZE

  # Image processing for evaluation.
  # Crop the central [height, width] of the image.
  resized_image = tf.image.resize_image_with_crop_or_pad(reshaped_image,
                                                         height, width)

  # Subtract off the mean and divide by the variance of the pixels.
  float_image = tf.image.per_image_standardization(resized_image)

  # Set the shapes of tensors.
  float_image.set_shape([height, width, 3])
  read_input.label.set_shape([1])

  # Ensure that the random shuffling has good mixing properties.
```

```
          min_fraction_of_examples_in_queue = 0.4
          min_queue_examples = int(num_examples_per_epoch *
                                   min_fraction_of_examples_in_queue)

          # Generate a batch of images and labels by building up a queue of examples.
          return _generate_image_and_label_batch(float_image, read_input.label,
                                                 min_queue_examples, batch_size,
                                                 shuffle=False)
```

1.0.3 CIFAR10 main

Main functions used in the program/for training and inference/evaluation.

```
In [4]: def _activation_summary(x):
          """Helper to create summaries for activations.

          Creates a summary that provides a histogram of activations.
          Creates a summary that measures the sparsity of activations.

          Args:
            x: Tensor
          Returns:
            nothing
          """
          # Remove 'tower_[0-9]/' from the name in case this is a multi-GPU training
          # session. This helps the clarity of presentation on tensorboard.
          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))


        def _variable_on_cpu(name, shape, initializer):
          """Helper to create a Variable stored on CPU memory.

          Args:
            name: name of the variable
            shape: list of ints
            initializer: initializer for Variable

          Returns:
            Variable Tensor
          """
          with tf.device('/cpu:0'):
            dtype = tf.float16 if f_use_fp16 else tf.float32
            var = tf.get_variable(name, shape, initializer=initializer, dtype=dtype)
          return var
```

```
def _variable_with_weight_decay(name, shape, stddev, wd):
  """Helper to create an initialized Variable with weight decay.

  Note that the Variable is initialized with a truncated normal distribution.
  A weight decay is added only if one is specified.

  Args:
    name: name of the variable
    shape: list of ints
    stddev: standard deviation of a truncated Gaussian
    wd: add L2Loss weight decay multiplied by this float. If None, weight
        decay is not added for this Variable.

  Returns:
    Variable Tensor
  """
  dtype = tf.float16 if f_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


def distorted_inputs():
  """Construct distorted input for CIFAR training using the Reader ops.

  Returns:
    images: Images. 4D tensor of [batch_size, IMAGE_SIZE, IMAGE_SIZE, 3] size.
    labels: Labels. 1D tensor of [batch_size] size.

  Raises:
    ValueError: If no data_dir
  """
  if not f_data_dir:
    raise ValueError('Please supply a data_dir')
  data_dir = os.path.join(f_data_dir, 'cifar-10-batches-bin')
  images, labels = distorted_inputs2(data_dir=data_dir,
                                                  batch_size=f_batch_size)
  if f_use_fp16:
    images = tf.cast(images, tf.float16)
    labels = tf.cast(labels, tf.float16)
  return images, labels
```

```
def inputs(eval_data):
  """Construct input for CIFAR evaluation using the Reader ops.

  Args:
    eval_data: bool, indicating if one should use the train or eval data set.

  Returns:
    images: Images. 4D tensor of [batch_size, IMAGE_SIZE, IMAGE_SIZE, 3] size.
    labels: Labels. 1D tensor of [batch_size] size.

  Raises:
    ValueError: If no data_dir
  """
  if not f_data_dir:
    raise ValueError('Please supply a data_dir')
  data_dir = os.path.join(f_data_dir, 'cifar-10-batches-bin')
  images, labels = inputs2(eval_data=eval_data,
                                        data_dir=data_dir,
                                        batch_size=f_batch_size)
  if f_use_fp16:
    images = tf.cast(images, tf.float16)
    labels = tf.cast(labels, tf.float16)
  return images, labels


def inference(images):
  """Build the CIFAR-10 model.

  Args:
    images: Images returned from distorted_inputs() or inputs().

  Returns:
    Logits.
  """
  # We instantiate all variables using tf.get_variable() instead of
  # tf.Variable() in order to share variables across multiple GPU training runs.
  # If we only ran this model on a single GPU, we could simplify this function
  # by replacing all instances of tf.get_variable() with tf.Variable().
  #
  # 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:
    # Move everything into depth so we can perform a single matrix multiply.
    reshape = tf.reshape(pool2, [f_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)
  # linear layer(WX + b),
  # We don't apply softmax here because
  # tf.nn.sparse_softmax_cross_entropy_with_logits accepts the unscaled logits
  # and performs the softmax internally for efficiency.
  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


def loss1(logits, labels):
  """Add L2Loss to all the trainable variables.

  Add summary for "Loss" and "Loss/avg".
  Args:
    logits: Logits from inference().
    labels: Labels from distorted_inputs or inputs(). 1-D tensor
            of shape [batch_size]

  Returns:
    Loss tensor of type float.
  """
  # Calculate the average cross entropy loss across the batch.
  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)

  # The total loss is defined as the cross entropy loss plus all of the weight
  # decay terms (L2 loss).
  return tf.add_n(tf.get_collection('losses'), name='total_loss')


def _add_loss_summaries(total_loss):
  """Add summaries for losses in CIFAR-10 model.

  Generates moving average for all losses and associated summaries for
  visualizing the performance of the network.

  Args:
    total_loss: Total loss from loss().
  Returns:
    loss_averages_op: op for generating moving averages of losses.
  """
  # Compute the moving average of all individual losses and the total loss.
  loss_averages = tf.train.ExponentialMovingAverage(0.9, name='avg')
  losses = tf.get_collection('losses')
  loss_averages_op = loss_averages.apply(losses + [total_loss])

  # Attach a scalar summary to all individual losses and the total loss; do the
  # same for the averaged version of the losses.
  for l in losses + [total_loss]:
    # Name each loss as '(raw)' and name the moving average version of the loss
    # as the original loss name.
    tf.summary.scalar(l.op.name + ' (raw)', l)
    tf.summary.scalar(l.op.name, loss_averages.average(l))

  return loss_averages_op


def train1(total_loss, global_step):
  """Train CIFAR-10 model.

  Create an optimizer and apply to all trainable variables. Add moving
  average for all trainable variables.

  Args:
    total_loss: Total loss from loss().
    global_step: Integer Variable counting the number of training steps
      processed.
  Returns:
    train_op: op for training.
  """
  # Variables that affect learning rate.
  num_batches_per_epoch = NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN / f_batch_size
  decay_steps = int(num_batches_per_epoch * NUM_EPOCHS_PER_DECAY)

  # Decay the learning rate exponentially based on the number of steps.
  lr = tf.train.exponential_decay(INITIAL_LEARNING_RATE,
                                  global_step,
                                  decay_steps,
                                  LEARNING_RATE_DECAY_FACTOR,
                                  staircase=True)
  tf.summary.scalar('learning_rate', lr)

  # Generate moving averages of all losses and associated summaries.
  loss_averages_op = _add_loss_summaries(total_loss)

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

  # Apply gradients.
  apply_gradient_op = opt.apply_gradients(grads, global_step=global_step)

  # Add histograms for trainable variables.
  for var in tf.trainable_variables():
    tf.summary.histogram(var.op.name, var)

  # Add histograms for gradients.
  for grad, var in grads:
    if grad is not None:
      tf.summary.histogram(var.op.name + '/gradients', grad)

  # Track the moving averages of all trainable variables.
  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')

  return train_op


def maybe_download_and_extract():
  """Download and extract the tarball from Alex's website."""
  dest_directory = f_data_dir
  if not os.path.exists(dest_directory):
    os.makedirs(dest_directory)
  filename = DATA_URL.split('/')[-1]
  filepath = os.path.join(dest_directory, filename)
  if not os.path.exists(filepath):
    def _progress(count, block_size, total_size):
      sys.stdout.write('\r>> Downloading %s %.1f%%' % (filename,
          float(count * block_size) / float(total_size) * 100.0))
      sys.stdout.flush()
    filepath, _ = urllib.request.urlretrieve(DATA_URL, filepath, _progress)
    print()
    statinfo = os.stat(filepath)
    print('Successfully downloaded', filename, statinfo.st_size, 'bytes.')
  extracted_dir_path = os.path.join(dest_directory, 'cifar-10-batches-bin')
  if not os.path.exists(extracted_dir_path):
    tarfile.open(filepath, 'r:gz').extractall(dest_directory)
```

1.0.4 CIFAR10 Train

below is the code for training the CIFAR10 model. The parameters to adjust the training are found at the top of the notebook.

```
In [5]: def train2():
          """Train CIFAR-10 for a number of steps."""
          with tf.Graph().as_default():
            global_step = tf.contrib.framework.get_or_create_global_step()

            # Get images and labels for CIFAR-10.
            # Force input pipeline to CPU:0 to avoid operations sometimes ending up on
            # GPU and resulting in a slow down.
            with tf.device('/cpu:0'):
              images, labels = distorted_inputs()

            # Build a Graph that computes the logits predictions from the
            # inference model.
            logits = inference(images)

            # Calculate loss.
            loss = loss1(logits, labels)

            # Build a Graph that trains the model with one batch of examples and
            # updates the model parameters.
            train_op = train1(loss, global_step)

            class _LoggerHook(tf.train.SessionRunHook):
              """Logs loss and runtime."""

              def begin(self):
                self._step = -1
                self._start_time = time.time()

              def before_run(self, run_context):
                self._step += 1
                return tf.train.SessionRunArgs(loss)  # Asks for loss value.

              def after_run(self, run_context, run_values):
                if self._step % f_log_frequency == 0:
                  current_time = time.time()
                  duration = current_time - self._start_time
                  self._start_time = current_time

                  loss_value = run_values.results
                  examples_per_sec = f_log_frequency * f_batch_size / duration
                  sec_per_batch = float(duration / f_log_frequency)

                  format_str = ('%s: step %d, loss = %.2f (%.1f examples/sec; %.3f '
                                'sec/batch)')
                  print (format_str % (datetime.now(), self._step, loss_value,
                                       examples_per_sec, sec_per_batch))

            with tf.train.MonitoredTrainingSession(
                checkpoint_dir=f_train_dir,
                hooks=[tf.train.StopAtStepHook(last_step=f_max_steps),
                       tf.train.NanTensorHook(loss),
                       _LoggerHook()],
                config=tf.ConfigProto(
                    log_device_placement=f_log_device_placement)) as mon_sess:
              while not mon_sess.should_stop():
                mon_sess.run(train_op)
```

```
In [6]: maybe_download_and_extract()
        #if tf.gfile.Exists(f_train_dir):
         #   tf.gfile.DeleteRecursively(f_train_dir)
        tf.gfile.MakeDirs(f_train_dir)
        train2()
```

```
Filling queue with 20000 CIFAR images before starting to train. This will take a few minutes.
INFO:tensorflow:Summary name conv1/weight_loss (raw) is illegal; using conv1/weight_loss__raw_
INFO:tensorflow:Summary name conv2/weight_loss (raw) is illegal; using conv2/weight_loss__raw_
INFO:tensorflow:Summary name local3/weight_loss (raw) is illegal; using local3/weight_loss__raw
INFO:tensorflow:Summary name local4/weight_loss (raw) is illegal; using local4/weight_loss__raw
INFO:tensorflow:Summary name softmax_linear/weight_loss (raw) is illegal; using softmax_linear/
INFO:tensorflow:Summary name cross_entropy (raw) is illegal; using cross_entropy__raw_ instead
INFO:tensorflow:Summary name total_loss (raw) is illegal; using total_loss__raw_ instead.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Saving checkpoints for 1 into ../CIFAR10_data/cifar10_train/model.ckpt.
2017-08-03 08:44:09.552473: step 0, loss = 4.67 (148.6 examples/sec; 0.861 sec/batch)
2017-08-03 08:44:17.689911: step 10, loss = 4.62 (157.3 examples/sec; 0.814 sec/batch)
2017-08-03 08:44:25.010844: step 20, loss = 4.49 (174.8 examples/sec; 0.732 sec/batch)
2017-08-03 08:44:32.141323: step 30, loss = 4.42 (179.5 examples/sec; 0.713 sec/batch)
2017-08-03 08:44:39.486599: step 40, loss = 4.29 (174.3 examples/sec; 0.735 sec/batch)
2017-08-03 08:44:46.874105: step 50, loss = 4.33 (173.3 examples/sec; 0.739 sec/batch)
2017-08-03 08:44:54.038493: step 60, loss = 4.24 (178.7 examples/sec; 0.716 sec/batch)
2017-08-03 08:45:01.207913: step 70, loss = 4.08 (178.5 examples/sec; 0.717 sec/batch)
2017-08-03 08:45:08.334230: step 80, loss = 4.23 (179.6 examples/sec; 0.713 sec/batch)
2017-08-03 08:45:15.452594: step 90, loss = 4.07 (179.8 examples/sec; 0.712 sec/batch)
INFO:tensorflow:Saving checkpoints for 100 into ../CIFAR10_data/cifar10_train/model.ckpt.
```

1.0.5 CIFAR10 Eval

Below is the code for evaluating the CIFAR model post training.

```
In [7]: def eval_once(saver, summary_writer, top_k_op, summary_op):
          """Run Eval once.
```

```
Args:
  saver: Saver.
  summary_writer: Summary writer.
  top_k_op: Top K op.
  summary_op: Summary op.
"""
with tf.Session() as sess:
  ckpt = tf.train.get_checkpoint_state(f_checkpoint_dir)
  if ckpt and ckpt.model_checkpoint_path:
    # Restores from checkpoint
    saver.restore(sess, ckpt.model_checkpoint_path)
    # Assuming model_checkpoint_path looks something like:
    #   /my-favorite-path/cifar10_train/model.ckpt-0,
    # extract global_step from it.
    global_step = ckpt.model_checkpoint_path.split('/')[-1].split('-')[-1]
  else:
    print('No checkpoint file found')
    return

  # Start the queue runners.
  coord = tf.train.Coordinator()
  try:
    threads = []
    for qr in tf.get_collection(tf.GraphKeys.QUEUE_RUNNERS):
      threads.extend(qr.create_threads(sess, coord=coord, daemon=True,
                                       start=True))

    num_iter = int(math.ceil(f_num_examples / f_batch_size))
    true_count = 0  # Counts the number of correct predictions.
    total_sample_count = num_iter * f_batch_size
    step = 0
    while step < num_iter and not coord.should_stop():
      predictions = sess.run([top_k_op])
      true_count += np.sum(predictions)
      step += 1

    # Compute precision @ 1.
    precision = true_count / total_sample_count
    print('%s: precision @ 1 = %.3f' % (datetime.now(), precision))

    summary = tf.Summary()
    summary.ParseFromString(sess.run(summary_op))
    summary.value.add(tag='Precision @ 1', simple_value=precision)
    summary_writer.add_summary(summary, global_step)
  except Exception as e:  # pylint: disable=broad-except
    coord.request_stop(e)
    coord.request_stop()
    coord.join(threads, stop_grace_period_secs=10)


def evaluate():
  """Eval CIFAR-10 for a number of steps."""
  with tf.Graph().as_default() as g:
    # Get images and labels for CIFAR-10.
    eval_data = f_eval_data == 'test'
    images, labels = inputs(eval_data=eval_data)

    # Build a Graph that computes the logits predictions from the
    # inference model.
    logits = inference(images)

    # Calculate predictions.
    top_k_op = tf.nn.in_top_k(logits, labels, 1)

    # Restore the moving average version of the learned variables for eval.
    variable_averages = tf.train.ExponentialMovingAverage(
        MOVING_AVERAGE_DECAY)
    variables_to_restore = variable_averages.variables_to_restore()
    saver = tf.train.Saver(variables_to_restore)

    # Build the summary operation based on the TF collection of Summaries.
    summary_op = tf.summary.merge_all()

    summary_writer = tf.summary.FileWriter(f_eval_dir, g)

    while True:
      eval_once(saver, summary_writer, top_k_op, summary_op)
      if f_run_once:
        break
      time.sleep(f_eval_interval_secs)
```

```
In [19]: #if tf.gfile.Exists(f_eval_dir):
         #    tf.gfile.DeleteRecursively(f_eval_dir)
         tf.gfile.MakeDirs(f_eval_dir)
         evaluate()
```

```
INFO:tensorflow:Restoring parameters from ../CIFAR10_data/cifar10_train/model.ckpt-100
2017-08-03 09:34:09.954916: precision @ 1 = 0.354
```