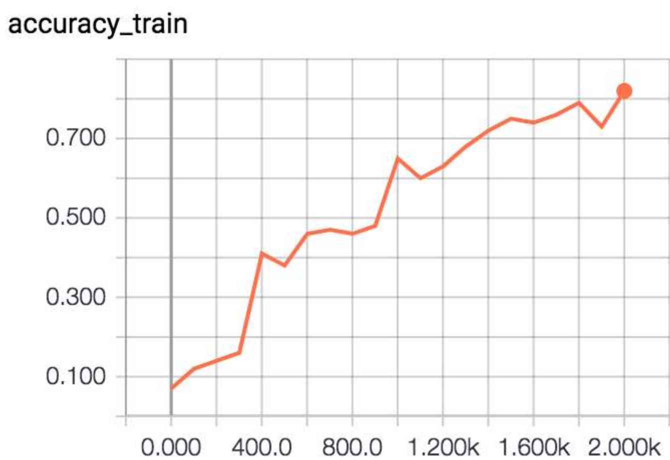
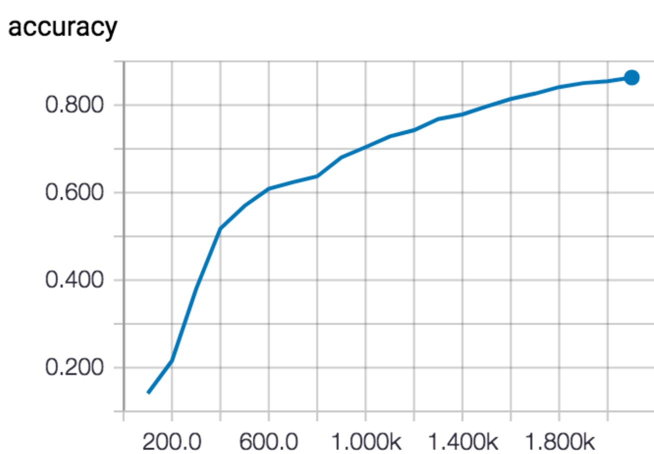
CS498AML_HW10_panz2

Tutorial Mnist:
After running 2000 steps, we observed that the training accuracy kept increasing after 1300 steps. We stopped at the 2000th step because the training after 2000 steps did not increase the accuracy a lot compared with the cost running time. We picked up the best model right at the 2000th step's training. The reason that training accuracy was a little lower than validation accuracy was probably because the training process was not sufficient enough due to our batch size and training step.

Train accuracy: 82.00%



Validation accuracy: 86.29%



Test accuracy: 86.82%

Modified Mnist:

<u>Basic ideas</u>: Reducing the max-pooling layers will reduce the re-sampling levels of the whole model and thus preserve more dimensions and original image features. Also, adding convolutional layers can extract more features from the image data. Both of them can improve the mode accuracy.

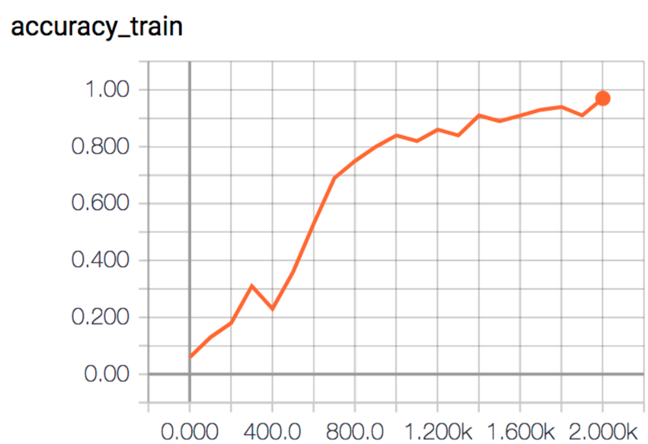
<u>Description:</u>

We firstly deleted all max-pooling layers to maximize the dimensions and image features' preservations. Then, we added one convolutional layer after the existing 3 convolutional layers, with 5*5 size, and 32 filters. For the dropout layer and the dense layer, I kept them as they were. Then, we got up to 94.64% test accuracy compared with the unmodified 86.82%.

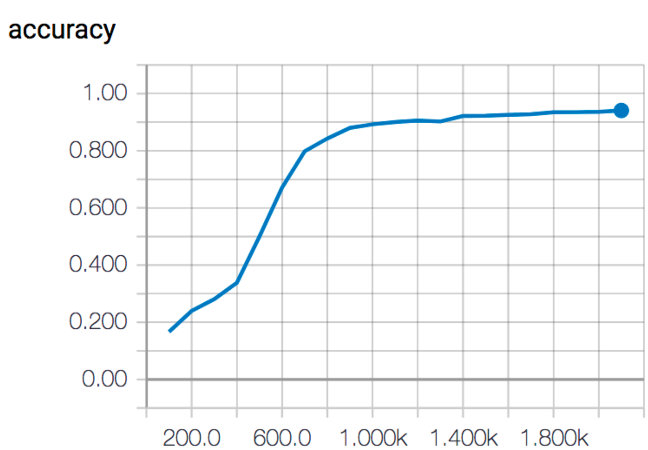
<u>Best model:</u>

After running 2000 and more steps, we found that the training accuracy kept an increasing trend after 400 steps. We stopped at the 2000th step because the training after 2000 steps did not increase the accuracy a lot compared with the cost running time. We picked up the best model right at the 2000th step's training.

<u>Train accuracy: 97.00%</u>



<u>Validation accuracy: 94.09%</u>

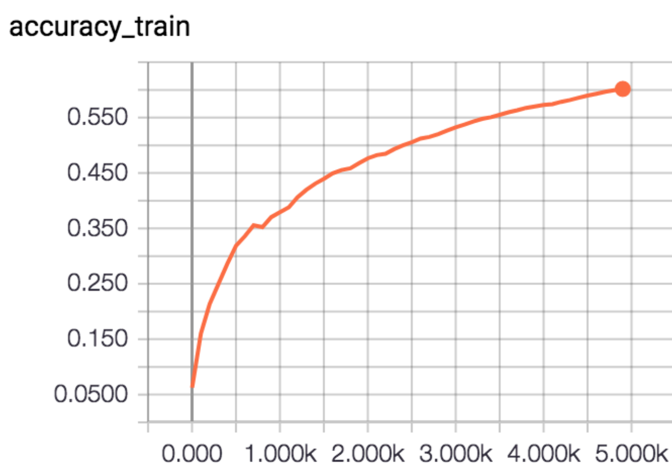


<u>Test accuracy: 94.64%</u>

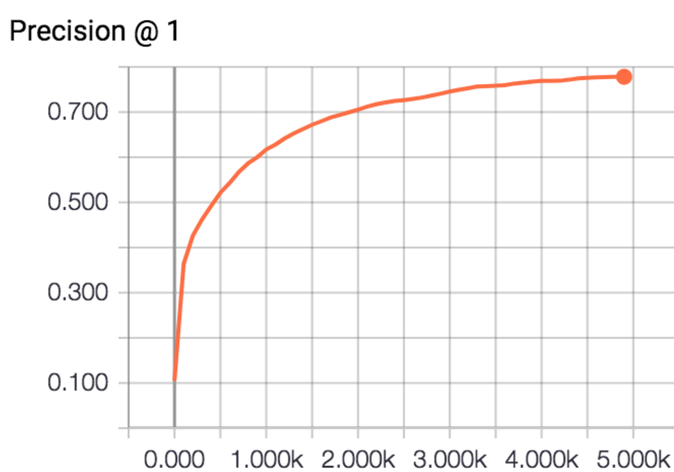
Tutorial Cifar10:

We run 10k steps and observed that the training accuracy kept an increasing trend. After 5k steps, the curve of validate accuracy tended to be gentle. So we stopped at the 5000th step, and picked the best model (the model had the best time cost efficiency based on our laptop).

Train accuracy: 60.17%



Validate accuracy: 77.82%



Test accuracy: 77.60%

Modified Cifar-10:

Basic ideas: Adding convolutional layers can extract more features from the image data, thus usually can result in better accuracy and improve the performance of the model.

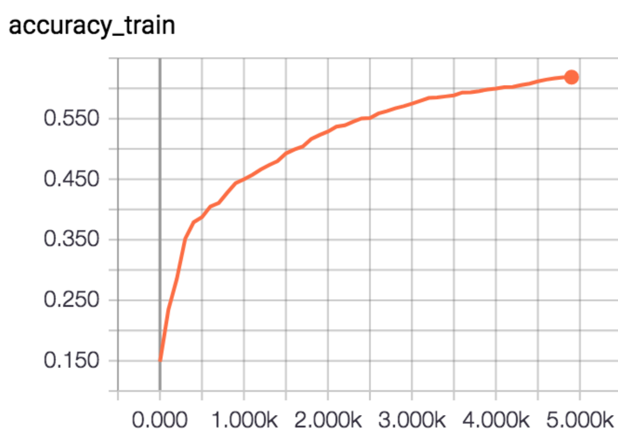
Description:

We added one convolutional layer name cov4 after cov1 layer and before pool1 layer, with shape of [128,24,64,64] and biases of [64]. For the max-pooling layers, norm levels, dropout layer and the dense layers, I kept them as they were. Then, we got up to 78.80% test accuracy compared with the unmodified 77.6%.

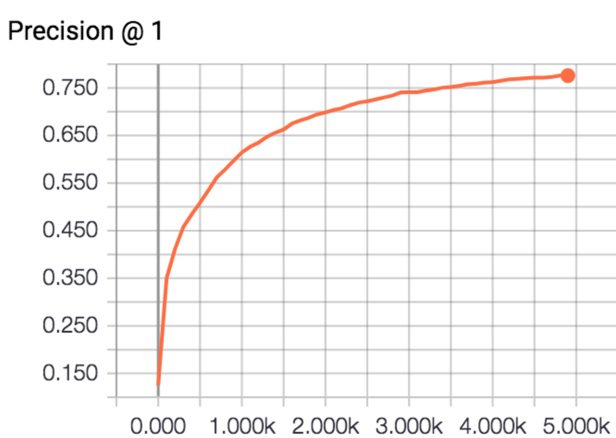
Best model:

After running 5,000 steps, we found that the training accuracy kept an increasing trend. We stopped at the 5000th step because the cost running time had been extremely long on our laptops. We picked up the best model right at the 5000th step's training.

Train accuracy: 61.86%



Validate accuracy: 77.58%



Test accuracy: 78.80%

```
def cnn_model_fn(features, labels, mode):
  """Model function for CNN."""
  # Input Layer
  # Reshape X to 4-D tensor: [batch_size, width, height, channels]
  # MNIST images are 28x28 pixels, and have one color channel
  input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])

  conv1 = tf.layers.conv2d(
      inputs=input_layer,
      filters=32,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

  conv2 = tf.layers.conv2d(
      inputs=conv1,
      filters=32,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

  conv3 = tf.layers.conv2d(
      inputs=conv2,
      filters=32,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)


  conv4 = tf.layers.conv2d(
      inputs=conv3,
      filters=32,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

  # Flatten tensor into a batch of vectors
  # Input Tensor Shape: [batch_size, 7, 7, 64]
  # Output Tensor Shape: [batch_size, 7 * 7 * 64]
  conv4_flat = tf.reshape(conv4, [-1, 28 * 28 * 32])

  # Dense Layer
  # Densely connected layer with 1024 neurons
  # Input Tensor Shape: [batch_size, 7 * 7 * 64]
  # Output Tensor Shape: [batch_size, 1024]
  dense = tf.layers.dense(inputs=conv4_flat, units=1024, activation=tf.nn.relu)

  # Add dropout operation; 0.6 probability that element will be kept
  dropout = tf.layers.dropout(
      inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)

  # Logits layer
  # Input Tensor Shape: [batch_size, 1024]
  # Output Tensor Shape: [batch_size, 10]
  logits = tf.layers.dense(inputs=dropout, units=10)

  predictions = {
      # Generate predictions (for PREDICT and EVAL mode)
      "classes": tf.argmax(input=logits, axis=1),
      # Add `softmax_tensor` to the graph. It is used for PREDICT and by the
      # `logging_hook`.
      "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
  }
  if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

  # Calculate Loss (for both TRAIN and EVAL modes)
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

  accuracy = tf.metrics.accuracy(labels=labels,
                                 predictions=predictions["classes"],
                                 name='acc_op')
  metrics = {'accuracy': accuracy}


  # Configure the Training Op (for TRAIN mode)
  if mode == tf.estimator.ModeKeys.TRAIN:
    tf.summary.scalar('accuracy_train', accuracy[1])
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op, eval_metric_ops=metrics)

  # Add evaluation metrics (for EVAL mode)
  if mode == tf.estimator.ModeKeys.EVAL:
    tf.summary.scalar('accuracy_val', accuracy[1])
    return tf.estimator.EstimatorSpec(
          mode=mode, loss=loss, eval_metric_ops=metrics)
```

```python
def main(unused_argv):
  # Load training and eval data
  mnist = tf.contrib.learn.datasets.load_dataset("mnist")
  train_data = mnist.train.images  # Returns np.array
  train_labels = np.asarray(mnist.train.labels, dtype=np.int32)


  # training
  training = train_data[0: int(len(train_data) * 0.8)]
  training_labels = train_labels[0: int(len(train_labels) * 0.8)]

  #validation
  validate = train_data[int(len(train_data) * 0.8):]
  validate_labels = train_labels[int(len(train_labels) * 0.8):]

  #test
  eval_data = mnist.test.images  # Returns np.array
  eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)


  # Create the Estimator
  mnist_classifier = tf.estimator.Estimator(
      model_fn=cnn_model_fn, model_dir="validate")

  # Train the model
  train_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": training},
      y=training_labels,
      batch_size=100,
      num_epochs=None,
      shuffle=True, queue_capacity=400, num_threads=2)

  # validation and print results
  validate_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": validate},
      y=validate_labels,
      num_epochs=1,
      shuffle=False)


  # Evaluate the model and print results
  eval_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": eval_data},
      y=eval_labels,
      num_epochs=1,
      shuffle=False)

  for epoch in range(21):
      mnist_classifier.train(
          input_fn=train_input_fn,
          steps=100,
          # hooks=[logging_hook]
      )

      validate_results = mnist_classifier.evaluate(input_fn=validate_input_fn)
  eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)


if __name__ == "__main__":
    tf.app.run()
```

Modified CNN model:

```
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=None)
    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)
  print(conv1.shape)

  with tf.variable_scope('conv4') as scope:
      kernel = _variable_with_weight_decay('weights',
                                           shape=[128, 24, 64, 64],
                                           stddev=5e-2,
                                           wd=None)
      conv = tf.nn.conv2d(conv1, 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)
      conv4 = tf.nn.relu(pre_activation, name=scope.name)
      _activation_summary(conv4)

  # 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=None)
   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, [images.get_shape().as_list()[0], -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)

 with tf.variable_scope('softmax_linear') as scope:
   weights = _variable_with_weight_decay('weights', [192, NUM_CLASSES],
                                         stddev=1/192.0, wd=None)
   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
```

Defining loss & optimizer & train:

```
def train():
  """Train CIFAR-10 for a number of steps."""
  with tf.Graph().as_default():
    global_step = tf.train.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 = cifar10.distorted_inputs()

    # Build a Graph that computes the logits predictions from the
    # inference model.
    logits = cifar10.inference(images)

    # Calculate loss.
    loss = cifar10.loss(logits, labels)

    # Calculate train accuracy.
    predictions = {
        # Generate predictions (for PREDICT and EVAL mode)
        "classes": tf.argmax(input=logits, axis=1),
        # Add `softmax_tensor` to the graph. It is used for PREDICT and by the
        # `logging_hook`.
        "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
    }
    accuracy_train = tf.metrics.accuracy(labels=labels,
                                         predictions=predictions["classes"],
                                         name='acc_op')
    tf.summary.scalar('accuracy_train', accuracy_train[1])

    # Build a Graph that trains the model with one batch of examples and
    # updates the model parameters.
    train_op = cifar10.train(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 % FLAGS.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 = FLAGS.log_frequency * FLAGS.batch_size / duration
         sec_per_batch = float(duration / FLAGS.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))

         cifar10_eval.evaluate(data_type="val")

   with tf.train.MonitoredTrainingSession(
       checkpoint_dir=FLAGS.train_dir,
       hooks=[tf.train.StopAtStepHook(last_step=FLAGS.max_steps),
              tf.train.NanTensorHook(loss),
              _LoggerHook()],
       config=tf.ConfigProto(
                             log_device_placement=FLAGS.log_device_placement),
       save_summaries_steps=100,
       save_summaries_secs=None,
       save_checkpoint_steps=100,
       save_checkpoint_secs=None) as mon_sess:
     while not mon_sess.should_stop():
       mon_sess.run(train_op)
```

Other relevant code:

I used 4 data_batch for training and used data_batch_5.bin as validate_batch.bin.

```
def inputs(data_type, 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 data_type == "train":
    filenames = [os.path.join(data_dir, 'data_batch_%d.bin' % i)
                 for i in xrange(1, 5)]
    num_examples_per_epoch = NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN
  elif data_type == "test":
    filenames = [os.path.join(data_dir, 'test_batch.bin')]
    num_examples_per_epoch = NUM_EXAMPLES_PER_EPOCH_FOR_EVAL
  elif data_type == "val":
    filenames = [os.path.join(data_dir, 'validate_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)
```

Turn off sleep and make it run one time.

```
tf.app.flags.DEFINE_boolean('run_once', True,
    """Whether to run eval only once.""")
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 = FLAGS.eval_data == 'test'
    images, labels = cifar10.inputs(eval_data=eval_data)

    # Build a Graph that computes the logits predictions from the
    # inference model.
    logits = cifar10.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(
        cifar10.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(FLAGS.eval_dir, g)

    while True:
      eval_once(saver, summary_writer, top_k_op, summary_op)
      if FLAGS.run_once:
        break
#      time.sleep(FLAGS.eval_interval_secs)
```