TF-Slim Walkthrough

This notebook will walk you through the basics of using TF-Slim to define, train and evaluate neural networks on various tasks. It assumes a basic knowledge of neural networks.

Table of contents

Installation and setup

As of 8/28/16, the latest stable release of TF is r0.10, which does not contain the latest version of slim. To obtain the latest version of TF-Slim, please install the most recent nightly build of TF as explained here (https://github.com/tensorflow/models/tree/master/slim#installing-latest-version-of-tf-slim).

To use TF-Slim for image classification (as we do in this notebook), you also have to install the TF-Slim image models library from here (https://github.com/tensorflow/models/tree/master/slim). Let's suppose you install this into a directory called TF_MODELS. Then you should change directory to TF_MODELS/slim **before** running this notebook, so that these files are in your python path.

To check you've got these two steps to work, just execute the cell below. If it complains about unknown modules, restart the notebook after moving to the TF-Slim models directory.

In [1]:
```
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import math
import numpy as np
import tensorflow as tf
import time

from datasets import dataset_utils

# Main slim library
slim = tf.contrib.slim
```

Creating your first neural network with TF-Slim

Below we give some code to create a simple multilayer perceptron (MLP) which can be used for regression problems. The model has 2 hidden layers. The output is a single node. When this function is called, it will create various nodes, and silently add them to whichever global TF graph is currently in scope. When a node which corresponds to a layer with adjustable parameters (eg., a fully connected layer) is created, additional parameter variable nodes are silently created, and added to the graph. (We will discuss how to train the parameters later.)

We use variable scope to put all the nodes under a common name, so that the graph has some hierarchical structure. This is useful when we want to visualize the TF graph in tensorboard, or if we want to query related variables. The fully connected layers all use the same L2 weight decay and ReLu activations, as specified by **arg_scope**. (However, the final layer overrides these defaults, and uses an identity activation function.)

We also illustrate how to add a dropout layer after the first fully connected layer (FC1). Note that at test time, we do not drop out nodes, but instead use the average activations; hence we need to know whether the model is being constructed for training or testing, since the computational graph will be different in the two cases (although the variables, storing the model parameters, will be shared, since they have the same name/scope).

In [2]:
```
def regression_model(inputs, is_training=True, scope="deep_regression"):
    """Creates the regression model.

    Args:
        inputs: A node that yields a `Tensor` of size [batch_size, dimen
        is_training: Whether or not we're currently training the model.
        scope: An optional variable_op scope for the model.

    Returns:
        predictions: 1-D `Tensor` of shape [batch_size] of responses.
        end_points: A dict of end points representing the hidden layers.
    """
    with tf.variable_scope(scope, 'deep_regression', [inputs]):
        end_points = {}
        # Set the default weight _regularizer and acvitation for each fu
        with slim.arg_scope([slim.fully_connected],
                            activation_fn=tf.nn.relu,
                            weights_regularizer=slim.l2_regularizer(0.01

            # Creates a fully connected layer from the inputs with 32 hi
            net = slim.fully_connected(inputs, 32, scope='fc1')
            end_points['fc1'] = net

            # Adds a dropout layer to prevent over-fitting.
            net = slim.dropout(net, 0.8, is_training=is_training)

            # Adds another fully connected layer with 16 hidden units.
            net = slim.fully_connected(net, 16, scope='fc2')
            end_points['fc2'] = net

            # Creates a fully-connected layer with a single hidden unit.
            # layer is made linear by setting activation_fn=None.
            predictions = slim.fully_connected(net, 1, activation_fn=Non
            end_points['out'] = predictions

            return predictions, end_points
```

Let's create the model and examine its structure.

We create a TF graph and call regression_model(), which adds nodes (tensors) to the graph. We then examine their shape, and print the names of all the model variables which have been implicitly created inside of each layer. We see that the names of the variables follow the scopes that we specified.

In [3]:

```
with tf.Graph().as_default():
    # Dummy placeholders for arbitrary number of 1d inputs and outputs
    inputs = tf.placeholder(tf.float32, shape=(None, 1))
    outputs = tf.placeholder(tf.float32, shape=(None, 1))

    # Build model
    predictions, end_points = regression_model(inputs)

    # Print name and shape of each tensor.
    print "Layers"
    for k, v in end_points.iteritems():
        print 'name = {}, shape = {}'.format(v.name, v.get_shape())

    # Print name and shape of parameter nodes  (values not yet initializ
    print "\n"
    print "Parameters"
    for v in slim.get_model_variables():
        print 'name = {}, shape = {}'.format(v.name, v.get_shape())
```

```
Layers
name = deep_regression/fc1/Relu:0, shape = (?, 32)
name = deep_regression/fc2/Relu:0, shape = (?, 16)
name = deep_regression/prediction/BiasAdd:0, shape = (?, 1)


Parameters
name = deep_regression/fc1/weights:0, shape = (1, 32)
name = deep_regression/fc1/biases:0, shape = (32,)
name = deep_regression/fc2/weights:0, shape = (32, 16)
name = deep_regression/fc2/biases:0, shape = (16,)
name = deep_regression/prediction/weights:0, shape = (16, 1)
name = deep_regression/prediction/biases:0, shape = (1,)
```

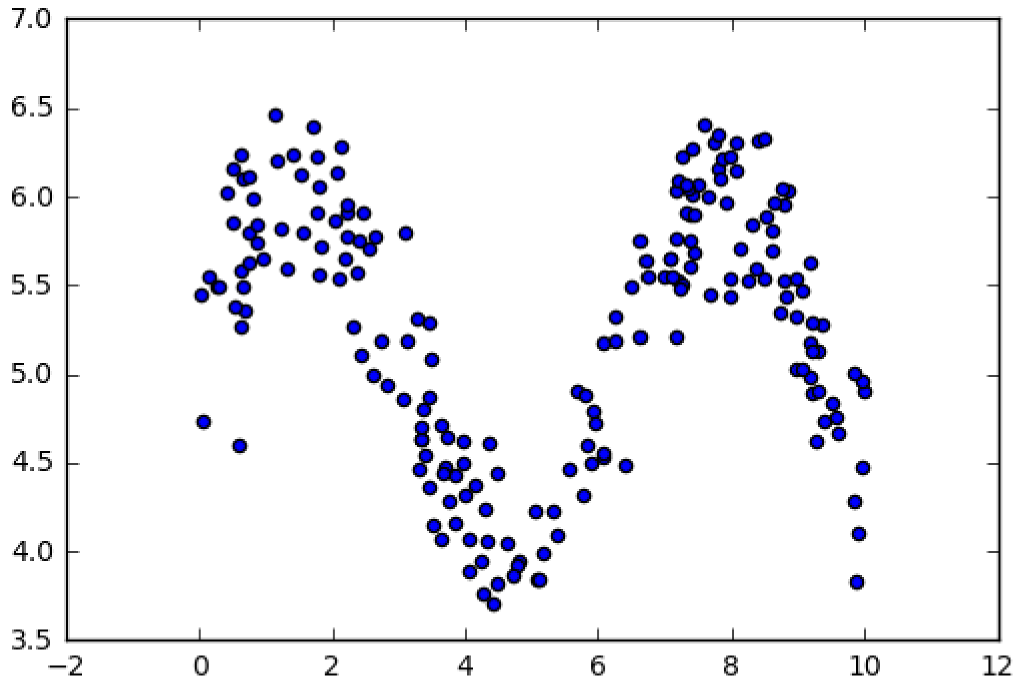
Let's create some 1d regression data .

We will train and test the model on some noisy observations of a nonlinear function.

In [4]:
```
def produce_batch(batch_size, noise=0.3):
    xs = np.random.random(size=[batch_size, 1]) * 10
    ys = np.sin(xs) + 5 + np.random.normal(size=[batch_size, 1], scale=n
    return [xs.astype(np.float32), ys.astype(np.float32)]

x_train, y_train = produce_batch(200)
x_test, y_test = produce_batch(200)
plt.scatter(x_train, y_train)
```

Out[4]: <matplotlib.collections.PathCollection at 0x7feb88e0ea10>



Let's fit the model to the data

The user has to specify the loss function and the optimizer, and slim does the rest. In particular, the slim.learning.train function does the following:

- For each iteration, evaluate the train_op, which updates the parameters using the optimizer applied to the current minibatch. Also, update the global_step.
- Occasionally store the model checkpoint in the specified directory. This is useful in case your machine crashes - then you can simply restart from the specified checkpoint.

In [5]:
```
def convert_data_to_tensors(x, y):
    inputs = tf.constant(x)
    inputs.set_shape([None, 1])

    outputs = tf.constant(y)
    outputs.set_shape([None, 1])
    return inputs, outputs
```

In [6]:

```
# The following snippet trains the regression model using a sum_of_squar
ckpt_dir = '/tmp/regression_model/'

with tf.Graph().as_default():
    tf.logging.set_verbosity(tf.logging.INFO)

    inputs, targets = convert_data_to_tensors(x_train, y_train)

    # Make the model.
    predictions, nodes = regression_model(inputs, is_training=True)

    # Add the loss function to the graph.
    loss = slim.losses.sum_of_squares(predictions, targets)

    # The total loss is the uers's loss plus any regularization losses.
    total_loss = slim.losses.get_total_loss()

    # Specify the optimizer and create the train op:
    optimizer = tf.train.AdamOptimizer(learning_rate=0.005)
    train_op = slim.learning.create_train_op(total_loss, optimizer)

    # Run the training inside a session.
    final_loss = slim.learning.train(
        train_op,
        logdir=ckpt_dir,
        number_of_steps=5000,
        save_summaries_secs=5,
        log_every_n_steps=500)

print("Finished training. Last batch loss:", final_loss)
print("Checkpoint saved in %s" % ckpt_dir)
```

```
INFO:tensorflow:Starting Session.
INFO:tensorflow:Starting Queues.
INFO:tensorflow:global_step/sec: 0
INFO:tensorflow:global step 500: loss = 0.4025 (0.00 sec/step)
INFO:tensorflow:global step 1000: loss = 0.2622 (0.00 sec/step)
INFO:tensorflow:global step 1500: loss = 0.2330 (0.00 sec/step)
INFO:tensorflow:global_step/sec: 336.873
INFO:tensorflow:global step 2000: loss = 0.2118 (0.01 sec/step)
INFO:tensorflow:global_step/sec: 108.201
INFO:tensorflow:global step 2500: loss = 0.2076 (0.01 sec/step)
INFO:tensorflow:global_step/sec: 99.5984
INFO:tensorflow:global step 3000: loss = 0.1935 (0.01 sec/step)
INFO:tensorflow:global_step/sec: 101.402
INFO:tensorflow:global step 3500: loss = 0.1958 (0.01 sec/step)
INFO:tensorflow:global_step/sec: 100.597
INFO:tensorflow:global step 4000: loss = 0.1804 (0.01 sec/step)
INFO:tensorflow:global_step/sec: 99.2014
INFO:tensorflow:global step 4500: loss = 0.1565 (0.01 sec/step)
INFO:tensorflow:global_step/sec: 98.7976
INFO:tensorflow:global step 5000: loss = 0.1734 (0.01 sec/step)
INFO:tensorflow:Stopping Training.
INFO:tensorflow:Finished training! Saving model to disk.

('Finished training. Last batch loss:', 0.17344266)
Checkpoint saved in /tmp/regression_model/
```

Training with multiple loss functions.

Sometimes we have multiple objectives we want to simultaneously optimize. In slim, it is easy to add more losses, as we show below. (We do not optimize the total loss in this example, but we show how to compute it.)

In [7]:
```
with tf.Graph().as_default():
    inputs, targets = convert_data_to_tensors(x_train, y_train)
    predictions, end_points = regression_model(inputs, is_training=True)

    # Add multiple loss nodes.
    sum_of_squares_loss = slim.losses.sum_of_squares(predictions, target
    absolute_difference_loss = slim.losses.absolute_difference(predictio

    # The following two ways to compute the total loss are equivalent
    regularization_loss = tf.add_n(slim.losses.get_regularization_losses
    total_loss1 = sum_of_squares_loss + absolute_difference_loss + regul

    # Regularization Loss is included in the total loss by default.
    # This is good for training, but not for testing.
    total_loss2 = slim.losses.get_total_loss(add_regularization_losses=T

    init_op = tf.initialize_all_variables()

    with tf.Session() as sess:
        sess.run(init_op) # Will initialize the parameters with random w

        total_loss1, total_loss2 = sess.run([total_loss1, total_loss2])

        print('Total Loss1: %f' % total_loss1)
        print('Total Loss2: %f' % total_loss2)

        print('Regularization Losses:')
        for loss in slim.losses.get_regularization_losses():
            print(loss)

        print('Loss Functions:')
        for loss in slim.losses.get_losses():
            print(loss)
```

```
Total Loss1: 47.347176
Total Loss2: 47.347176
Regularization Losses:
Tensor("deep_regression/fc1/weights/Regularizer/l2_regularizer:0", shap
e=(), dtype=float32)
Tensor("deep_regression/fc2/weights/Regularizer/l2_regularizer:0", shap
e=(), dtype=float32)
Tensor("deep_regression/prediction/weights/Regularizer/l2_regularizer:
0", shape=(), dtype=float32)
Loss Functions:
Tensor("sum_of_squares_loss/value:0", shape=(), dtype=float32)
Tensor("absolute_difference/value:0", shape=(), dtype=float32)
```

Let's load the saved model and use it for prediction.

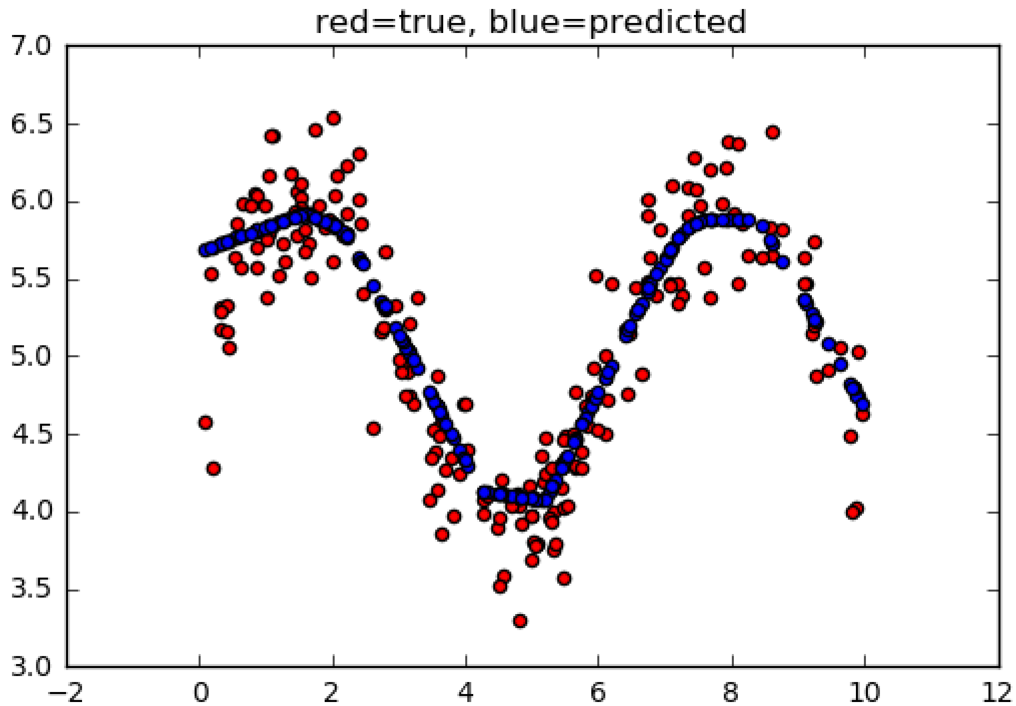
In [8]:
```
with tf.Graph().as_default():
    inputs, targets = convert_data_to_tensors(x_test, y_test)

    # Create the model structure. (Parameters will be loaded below.)
    predictions, end_points = regression_model(inputs, is_training=False

    # Make a session which restores the old parameters from a checkpoint
    sv = tf.train.Supervisor(logdir=ckpt_dir)
    with sv.managed_session() as sess:
        inputs, predictions, targets = sess.run([inputs, predictions, ta

plt.scatter(inputs, targets, c='r');
plt.scatter(inputs, predictions, c='b');
plt.title('red=true, blue=predicted')
```

Out[8]: <matplotlib.text.Text at 0x7feb844dce90>



Let's compute various evaluation metrics on the test set.

In TF-Slim termiology, losses are optimized, but metrics (which may not be differentiable, e.g., precision and recall) are just measured. As an illustration, the code below computes mean squared error and mean absolute error metrics on the test set.

Each metric declaration creates several local variables (which must be initialized via tf.initialize_local_variables()) and returns both a value_op and an update_op. When evaluated, the value_op returns the current value of the metric. The update_op loads a new batch of data, runs the model, obtains the predictions and accumulates the metric statistics appropriately before returning the current value of the metric. We store these value nodes and update nodes in 2 dictionaries.

After creating the metric nodes, we can pass them to slim.evaluation.evaluation, which repeatedly evaluates these nodes the specified number of times. (This allows us to compute the evaluation in a streaming fashion across minibatches, which is usefulf for large datasets.) Finally, we print the final value of each metric.

In [9]:
```
with tf.Graph().as_default():
    inputs, targets = convert_data_to_tensors(x_test, y_test)
    predictions, end_points = regression_model(inputs, is_training=False

    # Specify metrics to evaluate:
    names_to_value_nodes, names_to_update_nodes = slim.metrics.aggregate
      'Mean Squared Error': slim.metrics.streaming_mean_squared_error(pr
      'Mean Absolute Error': slim.metrics.streaming_mean_absolute_error(
    })

    # Make a session which restores the old graph parameters, and then r
    sv = tf.train.Supervisor(logdir=ckpt_dir)
    with sv.managed_session() as sess:
        metric_values = slim.evaluation.evaluation(
            sess,
            num_evals=1, # Single pass over data
            eval_op=names_to_update_nodes.values(),
            final_op=names_to_value_nodes.values())

    names_to_values = dict(zip(names_to_value_nodes.keys(), metric_value
    for key, value in names_to_values.iteritems():
      print('%s: %f' % (key, value))
```

```
INFO:tensorflow:Executing eval ops
INFO:tensorflow:Executing eval_op 1/1
INFO:tensorflow:Executing final op
```

```
Mean Squared Error: 0.117779
Mean Absolute Error: 0.264122
```

Reading Data with TF-Slim

Reading data with TF-Slim has two main components: A Dataset (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/slim/python/slim/data/datas and a DatasetDataProvider (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/slim/python/slim/data/datas The former is a descriptor of a dataset, while the latter performs the actions necessary for actually reading the data. Lets look at each one in detail:

Dataset

A TF-Slim Dataset (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/slim/python/slim/data/datas contains descriptive information about a dataset necessary for reading it, such as the list of data files and how to decode them. It also contains metadata including class labels, the size of the train/test splits and descriptions of the tensors that the dataset provides. For example, some datasets contain images with labels. Others augment this data with bounding box annotations, etc. The Dataset object allows us to write generic code using the same API, regardless of the data content and encoding type.

TF-Slim's Dataset works especially well when the data is stored as a (possibly sharded) TFRecords file (https://www.tensorflow.org/versions/r0.10/how_tos/reading_data/index.html#file-formats), where each record contains a tf.train.Example protocol buffer (https://github.com/tensorflow/tensorflow/blob/r0.10/tensorflow/core/example/example.proto). TF-Slim uses a consistent convention for naming the keys and values inside each Example record.

DatasetDataProvider

A DatasetDataProvider (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/slim/python/slim/data/datas is a class which actually reads the data from a dataset. It is highly configurable to read the data in various ways that may make a big impact on the efficiency of your training process. For example, it can be single or multi-threaded. If your data is sharded across many files, it can read each files serially, or from every file simultaneously.

Demo: The Flowers Dataset

For convenience, we've include scripts to convert several common image datasets into TFRecord format and have provided the Dataset descriptor files necessary for reading them. We demonstrate how easy it is to use these dataset via the Flowers dataset below.

Download the Flowers Dataset

We've made available a tarball of the Flowers dataset which has already been converted to TFRecord format.

In [10]:
```
import tensorflow as tf
from datasets import dataset_utils

url = "http://download.tensorflow.org/data/flowers.tar.gz"
flowers_data_dir = '/tmp/flowers'

if not tf.gfile.Exists(flowers_data_dir):
    tf.gfile.MakeDirs(flowers_data_dir)

dataset_utils.download_and_uncompress_tarball(url, flowers_data_dir)
```

```
>> Downloading flowers.tar.gz 100.0%
Successfully downloaded flowers.tar.gz 228649660 bytes.
```

Display some of the data.

In [11]:
```
from datasets import flowers
import tensorflow as tf

slim = tf.contrib.slim

with tf.Graph().as_default():
    dataset = flowers.get_split('train', flowers_data_dir)
    data_provider = slim.dataset_data_provider.DatasetDataProvider(
        dataset, common_queue_capacity=32, common_queue_min=1)
    image, label = data_provider.get(['image', 'label'])

    with tf.Session() as sess:
        with slim.queues.QueueRunners(sess):
            for i in xrange(4):
                np_image, np_label = sess.run([image, label])
                height, width, _ = np_image.shape
                class_name = name = dataset.labels_to_names[np_label]

                plt.figure()
                plt.imshow(np_image)
                plt.title('%s, %d x %d' % (name, height, width))
                plt.axis('off')
                plt.show()
```

dandelion, 212 x 320

dandelion, 213 x 320

daisy, 333 x 500

tulips, 239 x 320

Convolutional neural nets (CNNs).

In this section, we show how to train an image classifier using a simple CNN.

Define the model.

Below we define a simple CNN. Note that the output layer is linear function - we will apply softmax transformation externally to the model, either in the loss function (for training), or in the prediction function (during testing).

In [12]:
```
def my_cnn(images, num_classes, is_training):  # is_training is not used
    with slim.arg_scope([slim.max_pool2d], kernel_size=[3, 3], stride=2):
        net = slim.conv2d(images, 64, [5, 5])
        net = slim.max_pool2d(net)
        net = slim.conv2d(net, 64, [5, 5])
        net = slim.max_pool2d(net)
        net = slim.flatten(net)
        net = slim.fully_connected(net, 192)
        net = slim.fully_connected(net, num_classes, activation_fn=None)
        return net
```

Apply the model to some randomly generated images.

In [13]:
```
import tensorflow as tf

with tf.Graph().as_default():
    # The model can handle any input size because the first layer is con
    # The size of the model is determined when image_node is first passe
    # Once the variables are initialized, the size of all the weight mat
    # Because of the fully connected layers, this means that all subsequ
    # input size as the first image.
    batch_size, height, width, channels = 3, 28, 28, 3
    images = tf.random_uniform([batch_size, height, width, channels], ma

    # Create the model.
    num_classes = 10
    logits = my_cnn(images, num_classes, is_training=True)
    probabilities = tf.nn.softmax(logits)

    # Initialize all the variables (including parameters) randomly.
    init_op = tf.initialize_all_variables()

    with tf.Session() as sess:
        # Run the init_op, evaluate the model outputs and print the resu
        sess.run(init_op)
        probabilities = sess.run(probabilities)

print('Probabilities Shape:')
print(probabilities.shape)  # batch_size x num_classes

print('\nProbabilities:')
print(probabilities)

print('\nSumming across all classes (Should equal 1):')
print(np.sum(probabilities, 1)) # Each row sums to 1
```

```
Probabilities Shape:
(3, 10)

Probabilities:
[[ 0.07842434  0.13125037  0.1266212   0.08651967  0.09464223  0.100894
   0.10940798  0.0892633   0.07391185  0.10906509]
 [ 0.07968696  0.13326079  0.12446737  0.08938582  0.0915783   0.095829
89
   0.10769549  0.09414004  0.07029302  0.11366235]
 [ 0.08005995  0.13112688  0.12579846  0.08674899  0.09396919  0.104734
35
   0.10336404  0.08925078  0.07239325  0.11255412]]

Summing across all classes (Should equal 1):
[ 1.  1.  1.]
```

Train the model on the Flowers dataset.

Before starting, make sure you've run the code to Download the Flowers dataset. Now, we'll get a sense of what it looks like to use TF-Slim's training functions found in learning.py (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/slim/python/slim/learning.p

First, we'll create a function, `load_batch`, that loads batches of dataset from a dataset. Next, we'll train a model for a single step (just to demonstrate the API), and evaluate the results.

In [19]:
```
from preprocessing import inception_preprocessing
import tensorflow as tf

slim = tf.contrib.slim


def load_batch(dataset, batch_size=32, height=299, width=299, is_trainin
    """Loads a single batch of data.

    Args:
      dataset: The dataset to load.
      batch_size: The number of images in the batch.
      height: The size of each image after preprocessing.
      width: The size of each image after preprocessing.
      is_training: Whether or not we're currently training or evaluating

    Returns:
      images: A Tensor of size [batch_size, height, width, 3], image sam
      images_raw: A Tensor of size [batch_size, height, width, 3], image
      labels: A Tensor of size [batch_size], whose values range between
    """
    data_provider = slim.dataset_data_provider.DatasetDataProvider(
        dataset, common_queue_capacity=32,
        common_queue_min=8)
    image_raw, label = data_provider.get(['image', 'label'])

    # Preprocess image for usage by Inception.
    image = inception_preprocessing.preprocess_image(image_raw, height, 

    # Preprocess the image for display purposes.
    image_raw = tf.expand_dims(image_raw, 0)
    image_raw = tf.image.resize_images(image_raw, [height, width])
    image_raw = tf.squeeze(image_raw)

    # Batch it up.
    images, images_raw, labels = tf.train.batch(
          [image, image_raw, label],
          batch_size=batch_size,
          num_threads=1,
          capacity=2 * batch_size)

    return images, images_raw, labels
```

In [20]:

```
from datasets import flowers

# This might take a few minutes.
train_dir = '/tmp/tfslim_model/'
print('Will save model to %s' % train_dir)

with tf.Graph().as_default():
    tf.logging.set_verbosity(tf.logging.INFO)

    dataset = flowers.get_split('train', flowers_data_dir)
    images, _, labels = load_batch(dataset)

    # Create the model:
    logits = my_cnn(images, num_classes=dataset.num_classes, is_training

    # Specify the loss function:
    one_hot_labels = slim.one_hot_encoding(labels, dataset.num_classes)
    slim.losses.softmax_cross_entropy(logits, one_hot_labels)
    total_loss = slim.losses.get_total_loss()

    # Create some summaries to visualize the training process:
    tf.scalar_summary('losses/Total Loss', total_loss)

    # Specify the optimizer and create the train op:
    optimizer = tf.train.AdamOptimizer(learning_rate=0.01)
    train_op = slim.learning.create_train_op(total_loss, optimizer)

    # Run the training:
    final_loss = slim.learning.train(
      train_op,
      logdir=train_dir,
      number_of_steps=1, # For speed, we just do 1 epoch
      save_summaries_secs=1)

    print('Finished training. Final batch loss %d' % final_loss)
```

```
Will save model to /tmp/tfslim_model/
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-b3a5e7b602d0> in <module>()
      9 
     10     dataset = flowers.get_split('train', flowers_data_dir)
---> 11     images, _, labels = load_batch(dataset)
     12 
     13     # Create the model:

<ipython-input-19-2e6a4873fa1b> in load_batch(dataset, batch_size, height, width, is_training)
     30     # Preprocess the image for display purposes.
     31     image_raw = tf.expand_dims(image_raw, 0)
---> 32     image_raw = tf.image.resize_images(image_raw, [height, width])
     33     image_raw = tf.squeeze(image_raw)
     34 

TypeError: resize_images() takes at least 3 arguments (2 given)
```

Evaluate some metrics.

As we discussed above, we can compute various metrics besides the loss. Below we show how to compute prediction accuracy of the trained model, as well as top-5 classification accuracy. (The difference between evaluation and evaluation_loop is that the latter writes the results to a log directory, so they can be viewed in tensorboard.)

In [18]:

```
from datasets import flowers

# This might take a few minutes.
with tf.Graph().as_default():
    tf.logging.set_verbosity(tf.logging.DEBUG)

    dataset = flowers.get_split('train', flowers_data_dir)
    images, _, labels = load_batch(dataset)

    logits = my_cnn(images, num_classes=dataset.num_classes, is_training
    predictions = tf.argmax(logits, 1)

    # Define the metrics:
    names_to_values, names_to_updates = slim.metrics.aggregate_metric_ma
        'eval/Accuracy': slim.metrics.streaming_accuracy(predictions, la
        'eval/Recall@5': slim.metrics.streaming_recall_at_k(logits, labe
    })

    print('Running evaluation Loop...')
    checkpoint_path = tf.train.latest_checkpoint(train_dir)
    metric_values = slim.evaluation.evaluate_once(
        master='',
        checkpoint_path=checkpoint_path,
        logdir=train_dir,
        eval_op=names_to_updates.values(),
        final_op=names_to_values.values())

    names_to_values = dict(zip(names_to_values.keys(), metric_values))
    for name in names_to_values:
        print('%s: %f' % (name, names_to_values[name]))
```

```
---------------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-18-0ffeb83dbdb3> in <module>()
      6
      7     dataset = flowers.get_split('train', flowers_data_dir)
----> 8     images, _, labels = load_batch(dataset)
      9
     10     logits = my_cnn(images, num_classes=dataset.num_classes, is
_training=False)

<ipython-input-16-2e6a4873fa1b> in load_batch(dataset, batch_size, heig
ht, width, is_training)
     30     # Preprocess the image for display purposes.
     31     image_raw = tf.expand_dims(image_raw, 0)
---> 32     image_raw = tf.image.resize_images(image_raw, [height, widt
h])
     33     image_raw = tf.squeeze(image_raw)
     34

TypeError: resize_images() takes at least 3 arguments (2 given)
```

Using pre-trained models

Neural nets work best when they have many parameters, making them very flexible function approximators. However, this means they must be trained on big datasets. Since this process is slow, we provide various pre-trained models - see the list here (https://github.com/tensorflow/models/tree/master/slim#pre-trained-models).

You can either use these models as-is, or you can perform "surgery" on them, to modify them for some other task. For example, it is common to "chop off" the final pre-softmax layer, and replace it with a new set of weights corresponding to some new set of labels. You can then quickly fine tune the new model on a small new dataset. We illustrate this below, using inception-v1 as the base model. While models like Inception V3 are more powerful, Inception V1 is used for speed purposes.

Download the Inception V1 checkpoint

In [ ]:
```
from datasets import dataset_utils

url = "http://download.tensorflow.org/models/inception_v1_2016_08_28.tar
checkpoints_dir = '/tmp/checkpoints'

if not tf.gfile.Exists(checkpoints_dir):
    tf.gfile.MakeDirs(checkpoints_dir)

dataset_utils.download_and_uncompress_tarball(url, checkpoints_dir)
```

Apply Pre-trained model to Images.

We have to convert each image to the size expected by the model checkpoint. There is no easy way to determine this size from the checkpoint itself. So we use a preprocessor to enforce this.

In [ ]:

```
import numpy as np
import os
import tensorflow as tf
import urllib2

from datasets import imagenet
from nets import inception
from preprocessing import inception_preprocessing

slim = tf.contrib.slim

batch_size = 3
image_size = inception.inception_v1.default_image_size

with tf.Graph().as_default():
    url = 'https://upload.wikimedia.org/wikipedia/commons/7/70/EnglishCo
    image_string = urllib2.urlopen(url).read()
    image = tf.image.decode_jpeg(image_string, channels=3)
    processed_image = inception_preprocessing.preprocess_image(image, im
    processed_images  = tf.expand_dims(processed_image, 0)

    # Create the model, use the default arg scope to configure the batch
    with slim.arg_scope(inception.inception_v1_arg_scope()):
        logits, _ = inception.inception_v1(processed_images, num_classes
    probabilities = tf.nn.softmax(logits)

    init_fn = slim.assign_from_checkpoint_fn(
        os.path.join(checkpoints_dir, 'inception_v1.ckpt'),
        slim.get_model_variables('InceptionV1'))

    with tf.Session() as sess:
        init_fn(sess)
        np_image, probabilities = sess.run([image, probabilities])
        probabilities = probabilities[0, 0:]
        sorted_inds = [i[0] for i in sorted(enumerate(-probabilities), k

    plt.figure()
    plt.imshow(np_image.astype(np.uint8))
    plt.axis('off')
    plt.show()

    names = imagenet.create_readable_names_for_imagenet_labels()
    for i in range(5):
        index = sorted_inds[i]
        print('Probability %0.2f%% => [%s]' % (probabilities[index], nam
```

Fine-tune the model on a different set of labels.

We will fine tune the inception model on the Flowers dataset.

In [ ]:
```
# Note that this may take several minutes.

import os

from datasets import flowers
from nets import inception
from preprocessing import inception_preprocessing

slim = tf.contrib.slim
image_size = inception.inception_v1.default_image_size


def get_init_fn():
    """Returns a function run by the chief worker to warm-start the trai
    checkpoint_exclude_scopes=["InceptionV1/Logits", "InceptionV1/AuxLog

    exclusions = [scope.strip() for scope in checkpoint_exclude_scopes]

    variables_to_restore = []
    for var in slim.get_model_variables():
        excluded = False
        for exclusion in exclusions:
            if var.op.name.startswith(exclusion):
                excluded = True
                break
        if not excluded:
            variables_to_restore.append(var)

    return slim.assign_from_checkpoint_fn(
      os.path.join(checkpoints_dir, 'inception_v1.ckpt'),
      variables_to_restore)


train_dir = '/tmp/inception_finetuned/'

with tf.Graph().as_default():
    tf.logging.set_verbosity(tf.logging.INFO)

    dataset = flowers.get_split('train', flowers_data_dir)
    images, _, labels = load_batch(dataset, height=image_size, width=ima

    # Create the model, use the default arg scope to configure the batch
    with slim.arg_scope(inception.inception_v1_arg_scope()):
        logits, _ = inception.inception_v1(images, num_classes=dataset.n

    # Specify the loss function:
    one_hot_labels = slim.one_hot_encoding(labels, dataset.num_classes)
    slim.losses.softmax_cross_entropy(logits, one_hot_labels)
    total_loss = slim.losses.get_total_loss()

    # Create some summaries to visualize the training process:
    tf.scalar_summary('losses/Total Loss', total_loss)

    # Specify the optimizer and create the train op:
    optimizer = tf.train.AdamOptimizer(learning_rate=0.01)
    train_op = slim.learning.create_train_op(total_loss, optimizer)
```

```
    # Run the training:
    final_loss = slim.learning.train(
        train_op,
        logdir=train_dir,
        init_fn=get_init_fn(),
        number_of_steps=2)


print('Finished training. Last batch loss %f' % final_loss)
```

Apply fine tuned model to some images.

In [ ]:
```
import numpy as np
import tensorflow as tf
from datasets import flowers
from nets import inception

slim = tf.contrib.slim

image_size = inception.inception_v1.default_image_size
batch_size = 3

with tf.Graph().as_default():
    tf.logging.set_verbosity(tf.logging.INFO)

    dataset = flowers.get_split('train', flowers_data_dir)
    images, images_raw, labels = load_batch(dataset, height=image_size, w

    # Create the model, use the default arg scope to configure the batch
    with slim.arg_scope(inception.inception_v1_arg_scope()):
        logits, _ = inception.inception_v1(images, num_classes=dataset.n

    probabilities = tf.nn.softmax(logits)

    checkpoint_path = tf.train.latest_checkpoint(train_dir)
    init_fn = slim.assign_from_checkpoint_fn(
      checkpoint_path,
      slim.get_variables_to_restore())

    with tf.Session() as sess:
        with slim.queues.QueueRunners(sess):
            sess.run(tf.initialize_local_variables())
            init_fn(sess)
            np_probabilities, np_images_raw, np_labels = sess.run([proba

            for i in xrange(batch_size):
                image = np_images_raw[i, :, :, :]
                true_label = np_labels[i]
                predicted_label = np.argmax(np_probabilities[i, :])
                predicted_name = dataset.labels_to_names[predicted_label
                true_name = dataset.labels_to_names[true_label]

                plt.figure()
                plt.imshow(image.astype(np.uint8))
                plt.title('Ground Truth: [%s], Prediction [%s]' % (true_
                plt.axis('off')
                plt.show()
```