Image Classification

In this project, you'll classify images from the [CIFAR-10 dataset (https://www.cs.toronto.edu/~kriz/cifar.html)](https://www.cs.toronto.edu/~kriz/cifar.html). The dataset consists of airplanes, dogs, cats, and other objects. You'll preprocess the images, then train a convolutional neural network on all the samples. The images need to be normalized and the labels need to be one-hot encoded. You'll get to apply what you learned and build a convolutional, max pooling, dropout, and fully connected layers. At the end, you'll get to see your neural network's predictions on the sample images.

Get the Data

Run the following cell to download the [CIFAR-10 dataset for python (https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz)](https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz).

In [3]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
from urllib.request import urlretrieve
from os.path import isfile, isdir
from tqdm import tqdm
import problem_unittests as tests
import tarfile

cifar10_dataset_folder_path = 'cifar-10-batches-py'

class DLProgress(tqdm):
    last_block = 0

    def hook(self, block_num=1, block_size=1, total_size=None):
        self.total = total_size
        self.update((block_num - self.last_block) * block_size)
        self.last_block = block_num

if not isfile('cifar-10-python.tar.gz'):
    with DLProgress(unit='B', unit_scale=True, miniters=1, desc='CIFAR-10 Dataset') as
pbar:
```

Explore the Data

The dataset is broken into batches to prevent your machine from running out of memory. The CIFAR-10 dataset consists of 5 batches, named `data_batch_1`, `data_batch_2`, etc.. Each batch contains the labels and images that are one of the following:

- airplane
- automobile
- bird
- cat
- deer
- dog
- frog
- horse
- ship
- truck

Understanding a dataset is part of making predictions on the data. Play around with the code cell below by changing the `batch_id` and `sample_id`. The `batch_id` is the id for a batch (1-5). The `sample_id` is the id for a image and label pair in the batch.

Ask yourself "What are all possible labels?", "What is the range of values for the image data?", "Are the labels in order or random?". Answers to questions like these will help you preprocess the data and end up with better predictions.

In [4]:

```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import helper
import numpy as np

# Explore the dataset
batch_id = 1
sample_id = 5
helper.display_stats(cifar10_dataset_folder_path, batch_id, sample_id)
```

```
Stats of batch 1:
Samples: 10000
Label Counts: {0: 1005, 1: 974, 2: 1032, 3: 1016, 4: 999, 5: 937, 6: 1030,
7: 1001, 8: 1025, 9: 981}
First 20 Labels: [6, 9, 9, 4, 1, 1, 2, 7, 8, 3, 4, 7, 7, 2, 9, 9, 9, 3, 2,
6]

Example of Image 5:
Image - Min Value: 0 Max Value: 252
Image - Shape: (32, 32, 3)
Label - Label Id: 1 Name: automobile
```

In [5]:

```
def normalize(x):
    """
    Normalize a list of sample image data in the range of 0 to 1
    : x: List of image data.  The image shape is (32, 32, 3)
    : return: Numpy array of normalize data
    """
    # TODO: Implement Function
    return np.array(x / 255)


"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_normalize(normalize)
```

```
Tests Passed
```

One-hot encode

Just like the previous code cell, you'll be implementing a function for preprocessing. This time, you'll implement the `one_hot_encode` function. The input, `x`, are a list of labels. Implement the function to return the list of labels as One-Hot encoded Numpy array. The possible values for labels are 0 to 9. The one-hot encoding function should return the same encoding for each value between each call to `one_hot_encode`. Make sure to save the map of encodings outside the function.

Hint:

Look into LabelBinarizer in the preprocessing module of sklearn.

In [6]:

```
def one_hot_encode(x):
    """
    One hot encode a list of sample labels. Return a one-hot encoded vector for each la
bel.
    : x: List of sample labels
    : return: Numpy array of one-hot encoded labels
    """
    # TODO: Implement Function
    return np.eye(10)[x]
```

Preprocess all the data and save it

Running the code cell below will preprocess all the CIFAR-10 data and save it to file. The code below also uses 10% of the training data for validation.

In [7]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
# Preprocess Training, Validation, and Testing Data
helper.preprocess_and_save_data(cifar10_dataset_folder_path, normalize, one_hot_encode)
```

Check Point

This is your first checkpoint. If you ever decide to come back to this notebook or have to restart the notebook, you can start from here. The preprocessed data has been saved to disk.

In [8]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
import pickle
import problem_unittests as tests
import helper

# Load the Preprocessed Validation data
valid_features, valid_labels = pickle.load(open('preprocess_validation.p', mode='rb'))
```

Build the network

For the neural network, you'll build each layer into a function. Most of the code you've seen has been outside of functions. To test your code more thoroughly, we require that you put each layer in a function. This allows us to give you better feedback and test for simple mistakes using our unittests before you submit your project.

> **Note:** If you're finding it hard to dedicate enough time for this course each week, we've provided a small shortcut to this part of the project. In the next couple of problems, you'll have the option to use classes from the TensorFlow Layers (https://www.tensorflow.org/api_docs/python/tf/layers) or TensorFlow Layers (contrib) (https://www.tensorflow.org/api_guides/python/contrib.layers) packages to build each layer, except the layers you build in the "Convolutional and Max Pooling Layer" section. TF Layers is similar to Keras's and TFLearn's abstraction to layers, so it's easy to pickup.
>
> However, if you would like to get the most out of this course, try to solve all the problems *without* using anything from the TF Layers packages. You **can** still use classes from other packages that happen to have the same name as ones you find in TF Layers! For example, instead of using the TF Layers version of the `conv2d` class, tf.layers.conv2d (https://www.tensorflow.org/api_docs/python/tf/layers/conv2d), you would want to use the TF Neural Network version of `conv2d`, tf.nn.conv2d (https://www.tensorflow.org/api_docs/python/tf/nn/conv2d).

Let's begin!

Input

The neural network needs to read the image data, one-hot encoded labels, and dropout keep probability. Implement the following functions

- Implement `neural_net_image_input`
  - Return a TF Placeholder (https://www.tensorflow.org/api_docs/python/tf/placeholder)
  - Set the shape using `image_shape` with batch size set to `None`.
  - Name the TensorFlow placeholder "x" using the TensorFlow `name` parameter in the TF Placeholder (https://www.tensorflow.org/api_docs/python/tf/placeholder).
- Implement `neural_net_label_input`

In [9]:

```
import tensorflow as tf

def neural_net_image_input(image_shape):
    """
    Return a Tensor for a batch of image input
    : image_shape: Shape of the images
    : return: Tensor for image input.
    """
    # TODO: Implement Function
    return tf.placeholder(tf.float32, shape=(None, image_shape[0], image_shape[1], imag
e_shape[2]), name='x')


def neural_net_label_input(n_classes):
    """
    Return a Tensor for a batch of label input
    : n_classes: Number of classes
    : return: Tensor for label input.
    """
    # TODO: Implement Function
    return tf.placeholder(tf.float32, shape=(None, n_classes), name='y')


def neural_net_keep_prob_input():
    """
    Return a Tensor for keep probability
    : return: Tensor for keep probability.
    """
    # TODO: Implement Function
    return tf.placeholder(tf.float32, name='keep_prob')


"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tf.reset_default_graph()
tests.test_nn_image_inputs(neural_net_image_input)
tests.test_nn_label_inputs(neural_net_label_input)
tests.test_nn_keep_prob_inputs(neural_net_keep_prob_input)
```

```
Image Input Tests Passed.
Label Input Tests Passed.
```

Convolution and Max Pooling Layer

Convolution layers have a lot of success with images. For this code cell, you should implement the function `conv2d_maxpool` to apply convolution then max pooling:

- Create the weight and bias using `conv_ksize`, `conv_num_outputs` and the shape of `x_tensor`.
- Apply a convolution to `x_tensor` using weight and `conv_strides`.
  - We recommend you use same padding, but you're welcome to use any padding.
- Add bias
- Add a nonlinear activation to the convolution.
- Apply Max Pooling using `pool_ksize` and `pool_strides`.
  - We recommend you use same padding, but you're welcome to use any padding.

Note: You **can't** use TensorFlow Layers (https://www.tensorflow.org/api_docs/python/tf/layers) or TensorFlow Layers (contrib) (https://www.tensorflow.org/api_guides/python/contrib.layers) for **this** layer, but you can still use TensorFlow's Neural Network (https://www.tensorflow.org/api_docs/python/tf/nn) package. You may still use the shortcut option for all the **other** layers.

Hint:

When unpacking values as an argument in Python, look into the unpacking (https://docs.python.org/3/tutorial/controlflow.html#unpacking-argument-lists) operator.

In [10]:

```
def conv2d_maxpool(x_tensor, conv_num_outputs, conv_ksize, conv_strides, pool_ksize, po
ol_strides):
    """
    Apply convolution then max pooling to x_tensor
    :param x_tensor: TensorFlow Tensor
    :param conv_num_outputs: Number of outputs for the convolutional layer
    :param conv_ksize: kernal size 2-D Tuple for the convolutional layer
    :param conv_strides: Stride 2-D Tuple for convolution
    :param pool_ksize: kernal size 2-D Tuple for pool
    :param pool_strides: Stride 2-D Tuple for pool
    : return: A tensor that represents convolution and max pooling of x_tensor
    """
    # TODO: Implement Function
    conv_ksize = [1, conv_ksize[0], conv_ksize[1], 1]
    conv_n_inputs = x_tensor.get_shape().as_list()[3]

    conv_weight = tf.Variable(tf.truncated_normal([conv_ksize[0], conv_ksize[1], conv_n
_inputs, conv_num_outputs], stddev=0.1))
    conv_bias = tf.Variable(tf.zeros(conv_num_outputs))

    str = [1, conv_strides[0],conv_strides[1],1]

    convn = tf.nn.conv2d(x_tensor, conv_weight, strides=str,padding='SAME') + conv_bias
    convn = tf.nn.relu(convn)

    pool_ksize = [1, pool_ksize[0], pool_ksize[1], 1]
    pool_strides = [1, pool_strides[0], pool_strides[1], 1]
    conv_max_pooling = tf.nn.max_pool(convn, ksize=pool_ksize,strides=pool_strides,padd
ing='SAME')

    return conv_max_pooling


"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_con_pool(conv2d_maxpool)
```

Tests Passed

In [11]:

```
def flatten(x_tensor):
    """
    Flatten x_tensor to (Batch Size, Flattened Image Size)
    : x_tensor: A tensor of size (Batch Size, ...), where ... are the image dimensions.
    : return: A tensor of size (Batch Size, Flattened Image Size).
    """
    # TODO: Implement Function
    return tf.contrib.layers.flatten(x_tensor)


"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_flatten(flatten)
```

```
Tests Passed
```

Fully-Connected Layer

Implement the `fully_conn` function to apply a fully connected layer to `x_tensor` with the shape (*Batch Size*, *num_outputs*). Shortcut option: you can use classes from the TensorFlow Layers (https://www.tensorflow.org/api_docs/python/tf/layers) or TensorFlow Layers (contrib) (https://www.tensorflow.org/api_guides/python/contrib.layers) packages for this layer. For more of a challenge, only use other TensorFlow packages.

In [12]:

```
def fully_conn(x_tensor, num_outputs):
    """
    Apply a fully connected layer to x_tensor using weight and bias
    : x_tensor: A 2-D tensor where the first dimension is batch size.
    : num_outputs: The number of output that the new tensor should be.
    : return: A 2-D tensor where the second dimension is num_outputs.
    """
    # TODO: Implement Function
    v_width = x_tensor.get_shape().as_list()[1]
    v_weight = tf.Variable(tf.truncated_normal(([v_width, num_outputs]), stddev=0.1))
    v_bias = tf.Variable(tf.zeros(num_outputs))
    v_return = tf.add(tf.matmul(x_tensor, v_weight), v_bias)
```

Output Layer

Implement the `output` function to apply a fully connected layer to `x_tensor` with the shape (*Batch Size*, *num_outputs*). Shortcut option: you can use classes from the [TensorFlow Layers (https://www.tensorflow.org/api_docs/python/tf/layers)](https://www.tensorflow.org/api_docs/python/tf/layers) or [TensorFlow Layers (contrib) (https://www.tensorflow.org/api_guides/python/contrib.layers)](https://www.tensorflow.org/api_guides/python/contrib.layers) packages for this layer. For more of a challenge, only use other TensorFlow packages.

Note: Activation, softmax, or cross entropy should **not** be applied to this.

In [13]:

```
def output(x_tensor, num_outputs):
    """
    Apply a output layer to x_tensor using weight and bias
    : x_tensor: A 2-D tensor where the first dimension is batch size.
    : num_outputs: The number of output that the new tensor should be.
    : return: A 2-D tensor where the second dimension is num_outputs.
    """
    # TODO: Implement Function
    v_width = x_tensor.get_shape().as_list()[1]
    v_weight = tf.Variable(tf.truncated_normal([v_width, num_outputs], stddev=0.1))
    v_bias = tf.Variable(tf.zeros(num_outputs))
    v_return = tf.nn.bias_add(tf.matmul(x_tensor,v_weight), v_bias)
    return v_return


"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_output(output)
```

```
Tests Passed
```

Create Convolutional Model

Implement the function `conv_net` to create a convolutional neural network model. The function takes in a batch of images, x, and outputs logits. Use the layers you created above to create this model:

In [14]:

```
def conv_net(x, keep_prob):
    """
    Create a convolutional neural network model
    : x: Placeholder tensor that holds image data.
    : keep_prob: Placeholder tensor that hold dropout keep probability.
    : return: Tensor that represents logits
    """
    # TODO: Apply 1, 2, or 3 Convolution and Max Pool layers
    #    Play around with different number of outputs, kernel size and stride
    # Function Definition from Above:
    #    conv2d_maxpool(x_tensor, conv_num_outputs, conv_ksize, conv_strides, pool_ksiz
e, pool_strides)
    conv_num_outputs_1 = 32
    conv_num_outputs_2 = 64
    conv_num_outputs_3 = 128

    conv_ksize = [3, 3]
    conv_strides = [1, 1]
    pool_ksize = [2, 2]
    pool_strides = [2, 2]

    conv_layer_1 = conv2d_maxpool(x, conv_num_outputs_1, conv_ksize, conv_strides, pool
_ksize, pool_strides)
    conv_layer_2 = conv2d_maxpool(conv_layer_1, conv_num_outputs_2, conv_ksize, conv_st
rides, pool_ksize, pool_strides)
    conv_layer_3 = conv2d_maxpool(conv_layer_2, conv_num_outputs_3, conv_ksize, conv_st
rides, pool_ksize, pool_strides)

    # TODO: Apply a Flatten Layer
    # Function Definition from Above:
    #   flatten(x_tensor)
    flatten_layer = flatten(conv_layer_3)


    # TODO: Apply 1, 2, or 3 Fully Connected Layers
    #    Play around with different number of outputs
    # Function Definition from Above:
    #   fully_conn(x_tensor, num_outputs)
    fully_connected_layer_1 = fully_conn(flatten_layer, 100)
    fully_dropout = tf.nn.dropout(fully_connected_layer_1, keep_prob)
    fully_connected_layer_2 = fully_conn(fully_dropout, 100)
```

```
## Build the Neural Network ##
##############################

# Remove previous weights, bias, inputs, etc..
tf.reset_default_graph()

# Inputs
x = neural_net_image_input((32, 32, 3))
y = neural_net_label_input(10)
keep_prob = neural_net_keep_prob_input()

# Model
logits = conv_net(x, keep_prob)

# Name logits Tensor, so that is can be loaded from disk after training
logits = tf.identity(logits, name='logits')

# Loss and Optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
optimizer = tf.train.AdamOptimizer().minimize(cost)

# Accuracy
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32), name='accuracy')

tests.test_conv_net(conv_net)
```

```
WARNING:tensorflow:From <ipython-input-14-621daeaf5b92>:74: softmax_cross_
entropy_with_logits (from tensorflow.python.ops.nn_ops) is deprecated and
will be removed in a future version.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See @{tf.nn.softmax_cross_entropy_with_logits_v2}.

Neural Network Built!
```

Train the Neural Network

In [15]:

```
def train_neural_network(session, optimizer, keep_probability, feature_batch, label_bat
ch):
    """
    Optimize the session on a batch of images and labels
    : session: Current TensorFlow session
    : optimizer: TensorFlow optimizer function
    : keep_probability: keep probability
    : feature_batch: Batch of Numpy image data
    : label_batch: Batch of Numpy label data
    """
    # TODO: Implement Function
    session.run(optimizer, feed_dict={x: feature_batch, y:label_batch, keep_prob: keep_
probability})


"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_train_nn(train_neural_network)
```

```
Tests Passed
```

Show Stats

Implement the function `print_stats` to print loss and validation accuracy. Use the global variables `valid_features` and `valid_labels` to calculate validation accuracy. Use a keep probability of `1.0` to calculate the loss and validation accuracy.

In [16]:

```
def print_stats(session, feature_batch, label_batch, cost, accuracy):
    """
    Print information about loss and validation accuracy
    : session: Current TensorFlow session
    : feature_batch: Batch of Numpy image data
    : label_batch: Batch of Numpy label data
    : cost: TensorFlow cost function
    : accuracy: TensorFlow accuracy function
    """
```

Hyperparameters

Tune the following parameters:

- Set `epochs` to the number of iterations until the network stops learning or start overfitting
- Set `batch_size` to the highest number that your machine has memory for. Most people set them to common sizes of memory:
  - 64
  - 128
  - 256
  - ...
- Set `keep_probability` to the probability of keeping a node using dropout

In [17]:

```
# TODO: Tune Parameters
epochs = 10
batch_size = 512
keep_probability = 0.8
```

Train on a Single CIFAR-10 Batch

Instead of training the neural network on all the CIFAR-10 batches of data, let's use a single batch. This should save time while you iterate on the model to get a better accuracy. Once the final validation accuracy is 50% or greater, run the model on all the data in the next section.

In [18]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
print('Checking the Training on a Single Batch...')
with tf.Session() as sess:
    # Initializing the variables
    sess.run(tf.global_variables_initializer())

    # Training cycle
    for epoch in range(epochs):
        batch_i = 1
        for batch_features, batch_labels in helper.load_preprocess_training_batch(batch
_i, batch_size):
            train_neural_network(sess, optimizer, keep_probability, batch_features, bat
ch_labels)
        print('Epoch {:>2}, CIFAR-10 Batch {}:  '.format(epoch + 1, batch_i), end='')
        print_stats(sess, batch_features, batch_labels, cost, accuracy)
```

```
Checking the Training on a Single Batch...
Epoch  1, CIFAR-10 Batch 1:   Loss : 2.118286609649658
 Validation Accuracy : 0.24719999730587006
Epoch  2, CIFAR-10 Batch 1:   Loss : 2.017404556274414
 Validation Accuracy : 0.28519999980926514
Epoch  3, CIFAR-10 Batch 1:   Loss : 1.8904938697814941
 Validation Accuracy : 0.3271999955177307
Epoch  4, CIFAR-10 Batch 1:   Loss : 1.8039027452468872
 Validation Accuracy : 0.37040001153945923
Epoch  5, CIFAR-10 Batch 1:   Loss : 1.6856184005737305
 Validation Accuracy : 0.4034000039100647
Epoch  6, CIFAR-10 Batch 1:   Loss : 1.6039159297943115
 Validation Accuracy : 0.4277999997138977
Epoch  7, CIFAR-10 Batch 1:   Loss : 1.5421185493469238
 Validation Accuracy : 0.45179998874664307
Epoch  8, CIFAR-10 Batch 1:   Loss : 1.4694674015045166
 Validation Accuracy : 0.4758000075817108
Epoch  9, CIFAR-10 Batch 1:   Loss : 1.4200575351715088
 Validation Accuracy : 0.4896000027656555
Epoch 10, CIFAR-10 Batch 1:   Loss : 1.3803186416625977
 Validation Accuracy : 0.49480000138282776
```

In [19]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
save_model_path = './image_classification'

print('Training...')
with tf.Session() as sess:
    # Initializing the variables
    sess.run(tf.global_variables_initializer())

    # Training cycle
    for epoch in range(epochs):
        # Loop over all batches
        n_batches = 5
        for batch_i in range(1, n_batches + 1):
            for batch_features, batch_labels in helper.load_preprocess_training_batch(b
atch_i, batch_size):
                train_neural_network(sess, optimizer, keep_probability, batch_features,
 batch_labels)
            print('Epoch {:>2}, CIFAR-10 Batch {}:  '.format(epoch + 1, batch_i), end=
'')
            print_stats(sess, batch_features, batch_labels, cost, accuracy)

    # Save Model
    saver = tf.train.Saver()
    save_path = saver.save(sess, save_model_path)
```

```
Training...
Epoch  1, CIFAR-10 Batch 1:   Loss : 2.1363701820373535
 Validation Accuracy : 0.2370000034570694
Epoch  1, CIFAR-10 Batch 2:   Loss : 1.9591078758239746
 Validation Accuracy : 0.2824000120162964
Epoch  1, CIFAR-10 Batch 3:   Loss : 1.7971372604370117
 Validation Accuracy : 0.34279999136924744
Epoch  1, CIFAR-10 Batch 4:   Loss : 1.6542925834655762
 Validation Accuracy : 0.3901999890804291
Epoch  1, CIFAR-10 Batch 5:   Loss : 1.68010675907135
 Validation Accuracy : 0.3885999917984009
Epoch  2, CIFAR-10 Batch 1:   Loss : 1.6566356420516968
 Validation Accuracy : 0.41359999775886536
Epoch  2, CIFAR-10 Batch 2:   Loss : 1.513199806213379
 Validation Accuracy : 0.451200008392334
Epoch  2, CIFAR-10 Batch 3:   Loss : 1.3914072513580322
 Validation Accuracy : 0.45399999618530273
Epoch  2, CIFAR-10 Batch 4:   Loss : 1.3748313188552856
 Validation Accuracy : 0.4731999933719635
Epoch  2, CIFAR-10 Batch 5:   Loss : 1.3948787450790405
 Validation Accuracy : 0.4779999852180481
Epoch  3, CIFAR-10 Batch 1:   Loss : 1.4656791687011719
 Validation Accuracy : 0.4903999865055084
Epoch  3, CIFAR-10 Batch 2:   Loss : 1.3480420112609863
 Validation Accuracy : 0.49799999594688416
Epoch  3, CIFAR-10 Batch 3:   Loss : 1.2632367610931396
 Validation Accuracy : 0.49799999594688416
Epoch  3, CIFAR-10 Batch 4:   Loss : 1.2248228788375854
 Validation Accuracy : 0.5052000284194946
Epoch  3, CIFAR-10 Batch 5:   Loss : 1.254950761795044
 Validation Accuracy : 0.5135999917984009
Epoch  4, CIFAR-10 Batch 1:   Loss : 1.3745954036712646
 Validation Accuracy : 0.5281999707221985
Epoch  4, CIFAR-10 Batch 2:   Loss : 1.2504956722259521
 Validation Accuracy : 0.5270000100135803
Epoch  4, CIFAR-10 Batch 3:   Loss : 1.166024088859558
 Validation Accuracy : 0.5306000113487244
Epoch  4, CIFAR-10 Batch 4:   Loss : 1.1304821968078613
 Validation Accuracy : 0.5375999808311462
Epoch  4, CIFAR-10 Batch 5:   Loss : 1.1716175079345703
 Validation Accuracy : 0.5360000133514404
Epoch  5, CIFAR-10 Batch 1:   Loss : 1.3070701360702515
 Validation Accuracy : 0.5404000282287598
Epoch  5, CIFAR-10 Batch 2:   Loss : 1.1985963582992554
```

```
Epoch  7, CIFAR-10 Batch 1:   Loss : 1.1952072381973267
 Validation Accuracy : 0.5702000260353088
Epoch  7, CIFAR-10 Batch 2:   Loss : 1.0627775192260742
 Validation Accuracy : 0.5777999758720398
Epoch  7, CIFAR-10 Batch 3:   Loss : 0.9696932435035706
 Validation Accuracy : 0.5839999914169312
Epoch  7, CIFAR-10 Batch 4:   Loss : 0.9486395120620728
 Validation Accuracy : 0.5878000259399414
Epoch  7, CIFAR-10 Batch 5:   Loss : 0.9808236360549927
 Validation Accuracy : 0.5892000198364258
Epoch  8, CIFAR-10 Batch 1:   Loss : 1.1309258937835693
 Validation Accuracy : 0.5943999886512756
Epoch  8, CIFAR-10 Batch 2:   Loss : 0.9929926991462708
 Validation Accuracy : 0.5902000069618225
Epoch  8, CIFAR-10 Batch 3:   Loss : 0.9199864268302917
 Validation Accuracy : 0.5917999744415283
Epoch  8, CIFAR-10 Batch 4:   Loss : 0.9054796099662781
 Validation Accuracy : 0.6001999974250793
Epoch  8, CIFAR-10 Batch 5:   Loss : 0.9592823386192322
 Validation Accuracy : 0.5928000211715698
Epoch  9, CIFAR-10 Batch 1:   Loss : 1.0899767875671387
 Validation Accuracy : 0.6021999716758728
Epoch  9, CIFAR-10 Batch 2:   Loss : 0.9690217971801758
 Validation Accuracy : 0.5938000082969666
Epoch  9, CIFAR-10 Batch 3:   Loss : 0.8873429298400879
 Validation Accuracy : 0.6000000238418579
Epoch  9, CIFAR-10 Batch 4:   Loss : 0.8630309104919434
 Validation Accuracy : 0.6007999777793884
Epoch  9, CIFAR-10 Batch 5:   Loss : 0.9130008816719055
 Validation Accuracy : 0.6155999898910522
Epoch 10, CIFAR-10 Batch 1:   Loss : 1.0589079856872559
 Validation Accuracy : 0.6083999872207642
Epoch 10, CIFAR-10 Batch 2:   Loss : 0.9298615455627441
 Validation Accuracy : 0.607200026512146
Epoch 10, CIFAR-10 Batch 3:   Loss : 0.8609654307365417
 Validation Accuracy : 0.6074000000953674
Epoch 10, CIFAR-10 Batch 4:   Loss : 0.8289995789527893
 Validation Accuracy : 0.6136000156402588
Epoch 10, CIFAR-10 Batch 5:   Loss : 0.8618745803833008
 Validation Accuracy : 0.6172000169754028
```

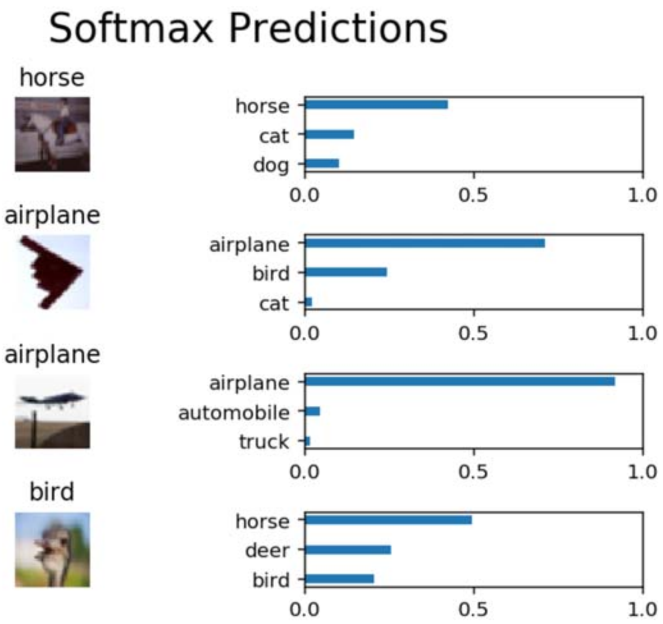
Checkpoint

In [20]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import tensorflow as tf
import pickle
import helper
import random

# Set batch size if not already set
try:
    if batch_size:
        pass
except NameError:
    batch_size = 64

save_model_path = './image_classification'
n_samples = 4
top_n_predictions = 3

def test_model():
    """
    Test the saved model against the test dataset
    """

    test_features, test_labels = pickle.load(open('preprocess_training.p', mode='rb'))
    loaded_graph = tf.Graph()

    with tf.Session(graph=loaded_graph) as sess:
        # Load model
        loader = tf.train.import_meta_graph(save_model_path + '.meta')
        loader.restore(sess, save_model_path)

        # Get Tensors from loaded model
        loaded_x = loaded_graph.get_tensor_by_name('x:0')
        loaded_y = loaded_graph.get_tensor_by_name('y:0')
        loaded_keep_prob = loaded_graph.get_tensor_by_name('keep_prob:0')
        loaded_logits = loaded_graph.get_tensor_by_name('logits:0')
        loaded_acc = loaded_graph.get_tensor_by_name('accuracy:0')
```

```
        random_test_predictions = sess.run(
            tf.nn.top_k(tf.nn.softmax(loaded_logits), top_n_predictions),
            feed_dict={loaded_x: random_test_features, loaded_y: random_test_labels, lo
aded_keep_prob: 1.0})
        helper.display_image_predictions(random_test_features, random_test_labels, rand
om_test_predictions)


test_model()
```

```
INFO:tensorflow:Restoring parameters from ./image_classification
Testing Accuracy: 0.6270278036594391
```



Why 50-80% Accuracy?

You might be wondering why you can't get an accuracy any higher. First things first, 50% isn't bad for a simple CNN. Pure guessing would get you 10% accuracy. That's because there are many more techniques that can be applied to your model and we recemmond that once you are done with this project, you explore!

Submitting This Project