Image Classification

In this project, you'll classify images from the CIFAR-10 dataset (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).

In [1]:

```
"""
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'

# Use Floyd's cifar-10 dataset if present
floyd_cifar10_location = '/input/cifar-10/python.tar.gz'
if isfile(floyd_cifar10_location):
    tar_gz_path = floyd_cifar10_location
else:
    tar_gz_path = 'cifar-10-python.tar.gz'

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(tar_gz_path):
    with DLProgress(unit='B', unit_scale=True, miniters=1, desc='CIFAR-1
0 Dataset') as pbar:
        urlretrieve(
            'https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz',
            tar_gz_path,
            pbar.hook)

if not isdir(cifar10_dataset_folder_path):
    with tarfile.open(tar_gz_path) as tar:
        tar.extractall()
        tar.close()


tests.test_folder_path(cifar10_dataset_folder_path)
```

```
All files found!
```

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 [2]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import helper
import numpy as np

# Explore the dataset
batch_id = 1 #1
sample_id = 5 #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: 10
30, 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
```

Implement Preprocess Functions

Normalize

In the cell below, implement the `normalize` function to take in image data, `x`, and return it as a normalized Numpy array. The values should be in the range of 0 to 1, inclusive. The return object should be the same shape as `x`.

In [3]:
```
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
    np_x = np.asarray( x, np.float32)
    y = np_x/255
    return y


"""
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: Don't reinvent the wheel.

In [4]:
```
def one_hot_encode(x):
    """
    One hot encode a list of sample labels. Return a one-hot encoded vec
tor for each label.
    : x: List of sample Labels
    : return: Numpy array of one-hot encoded labels
    """
    # TODO: Implement Function
    #print(x)
    num_classes = 10
    y = np.zeros( ( len(x), num_classes ) )
    y[ np.arange( len(x) ), x ] = 1
    return y


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

```
Tests Passed
```

Randomize Data

As you saw from exploring the data above, the order of the samples are randomized. It doesn't hurt to randomize it again, but you don't need to for this dataset.

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 [5]:
```
"""
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 [6]:
```
"""
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`
  - Return a TF Placeholder (https://www.tensorflow.org/api_docs/python/tf/placeholder)
  - Set the shape using `n_classes` with batch size set to `None`.
  - Name the TensorFlow placeholder "y" using the TensorFlow `name` parameter in the TF Placeholder (https://www.tensorflow.org/api_docs/python/tf/placeholder).
- Implement `neural_net_keep_prob_input`
  - Return a TF Placeholder (https://www.tensorflow.org/api_docs/python/tf/placeholder) for dropout keep probability.
  - Name the TensorFlow placeholder "keep_prob" using the TensorFlow `name` parameter in the TF Placeholder (https://www.tensorflow.org/api_docs/python/tf/placeholder).

These names will be used at the end of the project to load your saved model.

Note: `None` for shapes in TensorFlow allow for a dynamic size.

In [7]:
```
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
    x = tf.placeholder( tf.float32, shape=(None, image_shape[0], image_s
hape[1], image_shape[2]), name = "x" )
    return 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
    y = tf.placeholder( tf.float32, shape= (None, n_classes), name = "y"
)
    return y


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


"""
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.
Keep Prob 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.

In [8]:
```
def conv2d_maxpool(x_tensor, conv_num_outputs, conv_ksize, conv_strides,
pool_ksize, pool_strides):
    """
    Apply convolution then max pooling to x_tensor
    :param x_tensor: TensorFlow Tensor
    :param conv_num_outputs: Number of outputs for the convolutional lay
er
    :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
    #print(conv_ksize)
    #print(conv_strides)
    #print(pool_strides)
    #print(x_tensor.shape)

    color_index = 3       #[batch,h,w,color_ch]
    xs = x_tensor.shape
    num_color_channels = int(xs[color_index])
    W = tf.Variable( tf.truncated_normal( [conv_ksize[0], conv_ksize[1],
num_color_channels ,conv_num_outputs], mean=0.0, stddev=0.069  )  )
    b = tf.Variable( tf.zeros(conv_num_outputs )  )
    y = tf.nn.conv2d(x_tensor, W, [1, conv_strides[0], conv_strides[1],
1], padding = 'SAME' )
    y = tf.nn.bias_add(y, b)
    y = tf.nn.relu(y)

    # max pool
    y = tf.nn.max_pool(y, [1,pool_ksize[0],pool_ksize[1],1], [1,pool_str
ides[0],pool_strides[1],1 ], padding='SAME')

    return y


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

```
Tests Passed
```

Flatten Layer

Implement the `flatten` function to change the dimension of `x_tensor` from a 4-D tensor to a 2-D tensor. The output should be the shape (*Batch Size*, *Flattened Image Size*). 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 [9]:
```
def flatten(x_tensor):
    """
    Flatten x_tensor to (Batch Size, Flattened Image Size)
    : x_tensor: A tensor of size (Batch Size, ...), where ... are the im
age dimensions.
    : return: A tensor of size (Batch Size, Flattened Image Size).
    """
    # TODO: Implement Function
    shape = x_tensor.get_shape().as_list()
    dim = np.prod( shape[1:] )
    y = tf.reshape(x_tensor,[-1,dim] )
    return y


"""
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 [10]:
```
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
    #print(x_tensor,shape)
    # print(num_outputs)
    x_shape = x_tensor.shape
    x_len = int( x_shape[1] )
    num_outputs = int(num_outputs)
    W = tf.Variable( tf.truncated_normal( [ x_len, num_outputs ], mean=
0.0, stddev=0.088  )  )
    b = tf.Variable( tf.zeros( num_outputs)  )
    y = tf.matmul( x_tensor, W)
    y = tf.nn.bias_add(y, b)
    y = tf.nn.relu(y)

    return y


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

```
Tests Passed
```

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) 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.

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

In [11]:
```
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
    x_shape = x_tensor.shape
    x_len = int( x_shape[1] )
    num_outputs = int(num_outputs)
    W = tf.Variable( tf.truncated_normal( [ x_len, num_outputs ], mean=
0.0, stddev=0.088  )  )
    b = tf.Variable( tf.zeros( num_outputs)  )
    y = tf.matmul( x_tensor, W)
    y = tf.nn.bias_add(y, b)
    return y


"""
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:

- Apply 1, 2, or 3 Convolution and Max Pool layers
- Apply a Flatten Layer
- Apply 1, 2, or 3 Fully Connected Layers
- Apply an Output Layer
- Return the output
- Apply TensorFlow's Dropout (https://www.tensorflow.org/api_docs/python/tf/nn/dropout) to one or more layers in the model using `keep_prob`.

In [12]:
```
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 s
tride
    # Function Definition from Above:
    #    conv2d_maxpool(x_tensor, conv_num_outputs, conv_ksize, conv_str
ides, pool_ksize, pool_strides)
    filter_size = (2,2)
    filter_strides = (1,1)
    pool_size = (2,2)
    pool_strides=(1,1)
    num_outputs = 128  # num feature maps

    y = conv2d_maxpool(x, num_outputs, filter_size, filter_strides, pool
_size, pool_strides )
    y = conv2d_maxpool(y, num_outputs, filter_size, filter_strides, pool
_size, pool_strides )
    y = conv2d_maxpool(y, num_outputs, filter_size, filter_strides, pool
_size, pool_strides )


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


    # 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)
    y = fully_conn( y, num_outputs )
    y = tf.nn.dropout( y, keep_prob )
    y = fully_conn( y, num_outputs)
    y = tf.nn.dropout( y, keep_prob )
    y = fully_conn( y, 16)


    # TODO: Apply an Output Layer
    #    Set this to the number of classes
    # Function Definition from Above:
    #   output(x_tensor, num_outputs)
    num_classes = 10
    y = output( y, num_classes )


    # TODO: return output
    return y
```

```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""

##############################
## 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=log
its, 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='accur
acy')

tests.test_conv_net(conv_net)
```

```
Neural Network Built!
```

Train the Neural Network

Single Optimization

Implement the function `train_neural_network` to do a single optimization. The optimization should use `optimizer` to optimize in `session` with a `feed_dict` of the following:

- `x` for image input
- `y` for labels
- `keep_prob` for keep probability for dropout

This function will be called for each batch, so `tf.global_variables_initializer()` has already been called.

Note: Nothing needs to be returned. This function is only optimizing the neural network.

In [13]:
```
def train_neural_network(session, optimizer, keep_probability, feature_b
atch, label_batch):
    """
    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
    with session.as_default() as sess:
        try:
          sess.run( optimizer, feed_dict={ x:feature_batch, y:label_batc
h, keep_prob: keep_probability } )
        except:
          pass

"""
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.

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 [14]:
```
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
    """
    # TODO: Implement Function
    with session.as_default() as sess:
      valid_accuracy = accuracy.eval(feed_dict={y:valid_labels,x:valid_f
eatures,keep_prob:1.0} )
      loss = cost.eval(feed_dict={x:feature_batch, y:label_batch, keep_p
rob:1.0} )
      print("loss: " + str(loss))
      print("validation accuracy: "+str(valid_accuracy )  )
```

In [15]:
```
# TODO: Tune Parameters
epochs = 20
batch_size = 128
keep_probability = 0.5
```

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 [16]:
```
"""
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_train
ing_batch(batch_i, batch_size):
            train_neural_network(sess, optimizer, keep_probability, batc
h_features, batch_labels)
        print('Epoch {:>2}, CIFAR-10 Batch {}:  '.format(epoch + 1, batc
h_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: 2.23498
validation accuracy: 0.2284
Epoch  2, CIFAR-10 Batch 1:  loss: 1.99253
validation accuracy: 0.3228
Epoch  3, CIFAR-10 Batch 1:  loss: 1.96227
validation accuracy: 0.3658
Epoch  4, CIFAR-10 Batch 1:  loss: 1.75708
validation accuracy: 0.3878
Epoch  5, CIFAR-10 Batch 1:  loss: 1.56109
validation accuracy: 0.447
Epoch  6, CIFAR-10 Batch 1:  loss: 1.37276
validation accuracy: 0.4598
Epoch  7, CIFAR-10 Batch 1:  loss: 1.44607
validation accuracy: 0.4404
Epoch  8, CIFAR-10 Batch 1:  loss: 1.21647
validation accuracy: 0.4962
Epoch  9, CIFAR-10 Batch 1:  loss: 1.13947
validation accuracy: 0.5118
Epoch 10, CIFAR-10 Batch 1:  loss: 0.971305
validation accuracy: 0.5084
Epoch 11, CIFAR-10 Batch 1:  loss: 0.969991
validation accuracy: 0.5184
Epoch 12, CIFAR-10 Batch 1:  loss: 0.841149
validation accuracy: 0.532
Epoch 13, CIFAR-10 Batch 1:  loss: 0.808493
validation accuracy: 0.5364
Epoch 14, CIFAR-10 Batch 1:  loss: 0.738829
validation accuracy: 0.5512
Epoch 15, CIFAR-10 Batch 1:  loss: 0.670236
validation accuracy: 0.542
Epoch 16, CIFAR-10 Batch 1:  loss: 0.693221
validation accuracy: 0.5324
Epoch 17, CIFAR-10 Batch 1:  loss: 0.608877
validation accuracy: 0.5514
Epoch 18, CIFAR-10 Batch 1:  loss: 0.612272
validation accuracy: 0.5608
Epoch 19, CIFAR-10 Batch 1:  loss: 0.557387
validation accuracy: 0.5694
Epoch 20, CIFAR-10 Batch 1:  loss: 0.478256
validation accuracy: 0.5566
```

Fully Train the Model

Now that you got a good accuracy with a single CIFAR-10 batch, try it with all five batches.

In [17]:
```
"""
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_t
raining_batch(batch_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, accura
cy)

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

```
Training...
Epoch  1, CIFAR-10 Batch 1:  loss: 2.18773
validation accuracy: 0.227
Epoch  1, CIFAR-10 Batch 2:  loss: 1.86077
validation accuracy: 0.332
Epoch  1, CIFAR-10 Batch 3:  loss: 1.64954
validation accuracy: 0.3366
Epoch  1, CIFAR-10 Batch 4:  loss: 1.68228
validation accuracy: 0.3854
Epoch  1, CIFAR-10 Batch 5:  loss: 1.58426
validation accuracy: 0.4358
Epoch  2, CIFAR-10 Batch 1:  loss: 1.65657
validation accuracy: 0.4598
Epoch  2, CIFAR-10 Batch 2:  loss: 1.4244
validation accuracy: 0.485
Epoch  2, CIFAR-10 Batch 3:  loss: 1.27049
validation accuracy: 0.5054
Epoch  2, CIFAR-10 Batch 4:  loss: 1.36507
validation accuracy: 0.5298
Epoch  2, CIFAR-10 Batch 5:  loss: 1.28938
validation accuracy: 0.5422
Epoch  3, CIFAR-10 Batch 1:  loss: 1.3968
validation accuracy: 0.5336
Epoch  3, CIFAR-10 Batch 2:  loss: 1.30198
validation accuracy: 0.5452
Epoch  3, CIFAR-10 Batch 3:  loss: 1.06678
validation accuracy: 0.5424
Epoch  3, CIFAR-10 Batch 4:  loss: 1.14664
validation accuracy: 0.5574
Epoch  3, CIFAR-10 Batch 5:  loss: 1.13447
validation accuracy: 0.5742
Epoch  4, CIFAR-10 Batch 1:  loss: 1.15147
validation accuracy: 0.5748
Epoch  4, CIFAR-10 Batch 2:  loss: 1.08926
validation accuracy: 0.5902
Epoch  4, CIFAR-10 Batch 3:  loss: 0.980972
validation accuracy: 0.5868
Epoch  4, CIFAR-10 Batch 4:  loss: 1.01201
validation accuracy: 0.6022
Epoch  4, CIFAR-10 Batch 5:  loss: 0.948785
validation accuracy: 0.593
Epoch  5, CIFAR-10 Batch 1:  loss: 1.06156
validation accuracy: 0.5996
Epoch  5, CIFAR-10 Batch 2:  loss: 0.891953
validation accuracy: 0.608
Epoch  5, CIFAR-10 Batch 3:  loss: 0.83146
validation accuracy: 0.6044
Epoch  5, CIFAR-10 Batch 4:  loss: 0.929139
validation accuracy: 0.58
Epoch  5, CIFAR-10 Batch 5:  loss: 0.893568
validation accuracy: 0.6198
Epoch  6, CIFAR-10 Batch 1:  loss: 0.979329
validation accuracy: 0.619
Epoch  6, CIFAR-10 Batch 2:  loss: 0.712039
validation accuracy: 0.641
Epoch  6, CIFAR-10 Batch 3:  loss: 0.72366
validation accuracy: 0.62
```

```
Epoch  6, CIFAR-10 Batch 4:  loss: 0.753985
validation accuracy: 0.6412
Epoch  6, CIFAR-10 Batch 5:  loss: 0.857092
validation accuracy: 0.6346
Epoch  7, CIFAR-10 Batch 1:  loss: 0.793168
validation accuracy: 0.6514
Epoch  7, CIFAR-10 Batch 2:  loss: 0.709464
validation accuracy: 0.6168
Epoch  7, CIFAR-10 Batch 3:  loss: 0.711974
validation accuracy: 0.6186
Epoch  7, CIFAR-10 Batch 4:  loss: 0.741894
validation accuracy: 0.653
Epoch  7, CIFAR-10 Batch 5:  loss: 0.762721
validation accuracy: 0.6454
Epoch  8, CIFAR-10 Batch 1:  loss: 0.69014
validation accuracy: 0.6532
Epoch  8, CIFAR-10 Batch 2:  loss: 0.674392
validation accuracy: 0.6542
Epoch  8, CIFAR-10 Batch 3:  loss: 0.545731
validation accuracy: 0.6486
Epoch  8, CIFAR-10 Batch 4:  loss: 0.668319
validation accuracy: 0.6636
Epoch  8, CIFAR-10 Batch 5:  loss: 0.631801
validation accuracy: 0.654
Epoch  9, CIFAR-10 Batch 1:  loss: 0.624855
validation accuracy: 0.6604
Epoch  9, CIFAR-10 Batch 2:  loss: 0.622428
validation accuracy: 0.6516
Epoch  9, CIFAR-10 Batch 3:  loss: 0.553882
validation accuracy: 0.6458
Epoch  9, CIFAR-10 Batch 4:  loss: 0.701822
validation accuracy: 0.6768
Epoch  9, CIFAR-10 Batch 5:  loss: 0.632018
validation accuracy: 0.6662
Epoch 10, CIFAR-10 Batch 1:  loss: 0.543743
validation accuracy: 0.6818
Epoch 10, CIFAR-10 Batch 2:  loss: 0.590026
validation accuracy: 0.6692
Epoch 10, CIFAR-10 Batch 3:  loss: 0.473595
validation accuracy: 0.65
Epoch 10, CIFAR-10 Batch 4:  loss: 0.626424
validation accuracy: 0.6728
Epoch 10, CIFAR-10 Batch 5:  loss: 0.617814
validation accuracy: 0.6434
Epoch 11, CIFAR-10 Batch 1:  loss: 0.544614
validation accuracy: 0.673
Epoch 11, CIFAR-10 Batch 2:  loss: 0.503699
validation accuracy: 0.6682
Epoch 11, CIFAR-10 Batch 3:  loss: 0.428082
validation accuracy: 0.673
Epoch 11, CIFAR-10 Batch 4:  loss: 0.525878
validation accuracy: 0.6792
Epoch 11, CIFAR-10 Batch 5:  loss: 0.535992
validation accuracy: 0.6668
Epoch 12, CIFAR-10 Batch 1:  loss: 0.494917
validation accuracy: 0.6742
Epoch 12, CIFAR-10 Batch 2:  loss: 0.473314
```

```
validation accuracy: 0.6796
Epoch 12, CIFAR-10 Batch 3:  loss: 0.36458
validation accuracy: 0.6686
Epoch 12, CIFAR-10 Batch 4:  loss: 0.462133
validation accuracy: 0.6792
Epoch 12, CIFAR-10 Batch 5:  loss: 0.45809
validation accuracy: 0.6818
Epoch 13, CIFAR-10 Batch 1:  loss: 0.449702
validation accuracy: 0.6794
Epoch 13, CIFAR-10 Batch 2:  loss: 0.49355
validation accuracy: 0.6854
Epoch 13, CIFAR-10 Batch 3:  loss: 0.380194
validation accuracy: 0.6484
Epoch 13, CIFAR-10 Batch 4:  loss: 0.449708
validation accuracy: 0.6856
Epoch 13, CIFAR-10 Batch 5:  loss: 0.426374
validation accuracy: 0.6898
Epoch 14, CIFAR-10 Batch 1:  loss: 0.454345
validation accuracy: 0.676
Epoch 14, CIFAR-10 Batch 2:  loss: 0.448912
validation accuracy: 0.6712
Epoch 14, CIFAR-10 Batch 3:  loss: 0.316436
validation accuracy: 0.6716
Epoch 14, CIFAR-10 Batch 4:  loss: 0.430453
validation accuracy: 0.6832
Epoch 14, CIFAR-10 Batch 5:  loss: 0.379218
validation accuracy: 0.6732
Epoch 15, CIFAR-10 Batch 1:  loss: 0.413772
validation accuracy: 0.6808
Epoch 15, CIFAR-10 Batch 2:  loss: 0.401006
validation accuracy: 0.6856
Epoch 15, CIFAR-10 Batch 3:  loss: 0.249886
validation accuracy: 0.6924
Epoch 15, CIFAR-10 Batch 4:  loss: 0.435337
validation accuracy: 0.6852
Epoch 15, CIFAR-10 Batch 5:  loss: 0.346251
validation accuracy: 0.6724
Epoch 16, CIFAR-10 Batch 1:  loss: 0.348382
validation accuracy: 0.681
Epoch 16, CIFAR-10 Batch 2:  loss: 0.386274
validation accuracy: 0.6818
Epoch 16, CIFAR-10 Batch 3:  loss: 0.260445
validation accuracy: 0.6836
Epoch 16, CIFAR-10 Batch 4:  loss: 0.37473
validation accuracy: 0.6892
Epoch 16, CIFAR-10 Batch 5:  loss: 0.34794
validation accuracy: 0.6742
Epoch 17, CIFAR-10 Batch 1:  loss: 0.337967
validation accuracy: 0.6914
Epoch 17, CIFAR-10 Batch 2:  loss: 0.368635
validation accuracy: 0.6786
Epoch 17, CIFAR-10 Batch 3:  loss: 0.235005
validation accuracy: 0.6656
Epoch 17, CIFAR-10 Batch 4:  loss: 0.333753
validation accuracy: 0.6848
Epoch 17, CIFAR-10 Batch 5:  loss: 0.253867
validation accuracy: 0.685
```

```
Epoch 18, CIFAR-10 Batch 1:  loss: 0.358298
validation accuracy: 0.667
Epoch 18, CIFAR-10 Batch 2:  loss: 0.359959
validation accuracy: 0.6928
Epoch 18, CIFAR-10 Batch 3:  loss: 0.216147
validation accuracy: 0.6652
Epoch 18, CIFAR-10 Batch 4:  loss: 0.342233
validation accuracy: 0.687
Epoch 18, CIFAR-10 Batch 5:  loss: 0.266175
validation accuracy: 0.6772
Epoch 19, CIFAR-10 Batch 1:  loss: 0.347352
validation accuracy: 0.6698
Epoch 19, CIFAR-10 Batch 2:  loss: 0.337837
validation accuracy: 0.6872
Epoch 19, CIFAR-10 Batch 3:  loss: 0.191942
validation accuracy: 0.6846
Epoch 19, CIFAR-10 Batch 4:  loss: 0.354801
validation accuracy: 0.6738
Epoch 19, CIFAR-10 Batch 5:  loss: 0.206867
validation accuracy: 0.6944
Epoch 20, CIFAR-10 Batch 1:  loss: 0.298942
validation accuracy: 0.677
Epoch 20, CIFAR-10 Batch 2:  loss: 0.314409
validation accuracy: 0.6912
Epoch 20, CIFAR-10 Batch 3:  loss: 0.203522
validation accuracy: 0.6938
Epoch 20, CIFAR-10 Batch 4:  loss: 0.288115
validation accuracy: 0.6768
Epoch 20, CIFAR-10 Batch 5:  loss: 0.271713
validation accuracy: 0.6814
```

Checkpoint

The model has been saved to disk.

Test Model

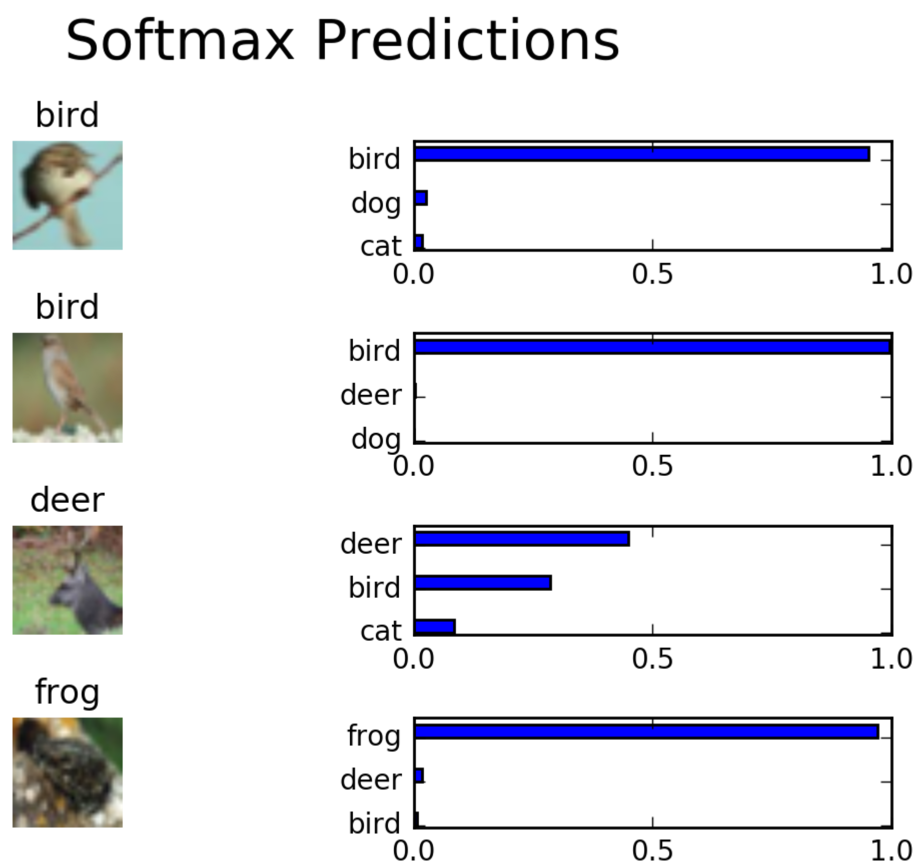
Test your model against the test dataset. This will be your final accuracy. You should have an accuracy greater than 50%. If you don't, keep tweaking the model architecture and parameters.

In [18]:

```
"""
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_test.p', m
ode='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')

        # Get accuracy in batches for memory limitations
        test_batch_acc_total = 0
        test_batch_count = 0

        for test_feature_batch, test_label_batch in helper.batch_feature
s_labels(test_features, test_labels, batch_size):
            test_batch_acc_total += sess.run(
                loaded_acc,
                feed_dict={loaded_x: test_feature_batch, loaded_y: test_
label_batch, loaded_keep_prob: 1.0})
            test_batch_count += 1
```

```
        print('Testing Accuracy: {}\n'.format(test_batch_acc_total/test_
batch_count))

        # Print Random Samples
        random_test_features, random_test_labels = tuple(zip(*random.sam
ple(list(zip(test_features, test_labels)), n_samples)))
        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, loaded_keep_prob: 1.0})
        helper.display_image_predictions(random_test_features, random_te
st_labels, random_test_predictions)


test_model()
```

```
INFO:tensorflow:Restoring parameters from ./image_classification
Testing Accuracy: 0.6889833860759493
```



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. However, you might notice people are getting scores well above 80% (http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#43494641522d3130). That's because we haven't taught you all there is to know about neural networks. We still need to cover a few more techniques.

Submitting This Project

When submitting this project, make sure to run all the cells before saving the notebook. Save the notebook file as "dlnd_image_classification.ipynb" and save it as a HTML file under "File" -> "Download as". Include the "helper.py" and "problem_unittests.py" files in your submission.