Image Classification

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

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'

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 D
ataset') as pbar:
        urlretrieve(
            'https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz',
            'cifar-10-python.tar.gz',
            pbar.hook)

if not isdir(cifar10_dataset_folder_path):
    with tarfile.open('cifar-10-python.tar.gz') as tar:
        tar.extractall()
        tar.close()


tests.test_folder_path(cifar10_dataset_folder_path)
```

```
All files found!
```

Explore the Data

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

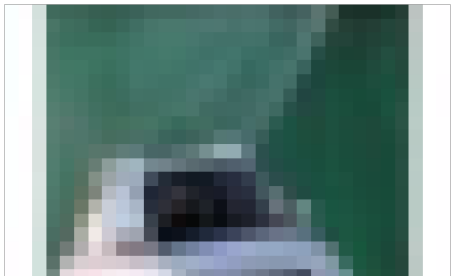
Understanding a dataset is part of making predictions on the data. Play around with the code cell below by changing the `batch_id` and `sample_id`. The `batch_id` is the id for a batch (1-5). The `sample_id` is the id for a image and label pair in the batch.

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
sample_id = 9998
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 9998:
Image - Min Value: 0 Max Value: 255
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
    """
    normalized_array = np.zeros((len(x),32,32,3) )
    # Normalize each image and create array
    # Dimensions of array are (number of images, image width, image height,
image color channels)
    for i, image in enumerate(x):
        normalized_image = (np.max(image) - image) / (np.max(image)- np.min(
image))
        normalized_array[i] = normalized_image
    return normalized_array


"""
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 [4]:

```
from sklearn.preprocessing import LabelBinarizer
labeler = LabelBinarizer()

# Define mapping outside of function so encoding is same every time
encoding_map = range(10)

def one_hot_encode(x):
    """
    One hot encode a list of sample labels. Return a one-hot encoded vector
for each label.
    : x: List of sample Labels
    : return: Numpy array of one-hot encoded labels
    """
    labeler.fit(encoding_map)
    return np.array(labeler.transform(x))

"""
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
```

```
import numpy as np
import math
import time

# 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 or TensorFlow Layers (contrib) 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, you would want to use the TF Neural Network version of `conv2d`, 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
  - 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.
- Implement `neural_net_label_input`
  - Return a 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.
- Implement `neural_net_keep_prob_input`
  - Return a TF Placeholder for dropout keep probability.
  - Name the TensorFlow placeholder "keep_prob" using the TensorFlow `name` parameter in the 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


    return tf.placeholder(tf.float32, shape=(None, image_shape[0],
image_shape[1], image_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.int32, 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.
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

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 or TensorFlow Layers (contrib) for **this** layer, but you can still use TensorFlow's Neural Network 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 operator.

In [8]:

```
def conv2d_maxpool(x_tensor, conv_num_outputs, conv_ksize, conv_strides, po
ol_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 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_ten
sor
    """
    # TODO: Implement Function

    # Find dimensions of input tensor: (batch_size, width, height, depth)
    dimensions = x_tensor.get_shape().as_list()

    # std for initializing weights = sqrt(2/n),  where n =
filter_width*filter_height*input_depth
    std = np.sqrt(2 / (conv_ksize[0] * conv_ksize[1] * dimensions[3]))

    # Shape of weights will be (Filter width, filter height, channels, laye
rs)
    # Shape of bias will be layers
    weights = tf.Variable(tf.truncated_normal([conv_ksize[0], conv_ksize[1]
, dimensions[3], conv_num_outputs],
                                              mean=0.0, stddev=std))
    bias = tf.Variable(tf.zeros([conv_num_outputs]))

    # Perform convolution and add bias
    conv = tf.nn.conv2d(x_tensor, weights, strides=[1, conv_strides[0],
conv_strides[1], 1], padding="SAME")
    conv = tf.nn.bias_add(conv, bias)

    # Implement non-linear activation function and pool results
    conv = tf.nn.relu(conv)
    conv = tf.nn.max_pool(conv, ksize=[1, pool_ksize[0], pool_ksize[1], 1],
strides=[1, pool_strides[0], pool_strides[1], 1], padding="SAME")
    return conv
```

```
"""
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 or TensorFlow Layers (contrib) 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 image
dimensions.
    : return: A tensor of size (Batch Size, Flattened Image Size).
    """
    # TODO: Implement Function

    # Using tf.contrib.layers
    return tf.contrib.layers.flatten(x_tensor)

    ''' Long way: get shape, use numpy to find second dimension of new tens
or, reshape tensor
    dimensions = x_tensor.get_shape().as_list()
    image_flat_length = np.product(dimensions[1:])
    return tf.reshape(x_tensor, [-1, image_flat_length])'''



"""
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 or TensorFlow Layers (contrib) 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.
```

```
    : 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
    dimensions = x_tensor.get_shape().as_list()



    # std for weights intialization = sqrt(2/n) where n is number of inputs
of previous layer
    std = np.sqrt(2 / (dimensions[1]))

    # Create weights; size = (second dimension of x_tensor, num_outputs)
    # Create bias; size = (num_outputs)
    weights = tf.Variable(tf.truncated_normal(shape=[dimensions[1],
num_outputs], mean=0.0, stddev=std))
    bias = tf.Variable(tf.zeros([num_outputs]))

    # Create fully connected layer
    fc = tf.add(tf.matmul(x_tensor, weights), bias)
    fc = tf.nn.relu(fc)
    return fc


"""
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 or TensorFlow Layers (contrib) 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

    dimensions = x_tensor.get_shape().as_list()
    # Create weights; size = (second dimension of x_tensor, num_outputs)
    # Create bias; size = (num_outputs)
    weights = tf.Variable(tf.truncated_normal([dimensions[1], num_outputs])
)
    bias = tf.Variable(tf.zeros([num_outputs]))
```

```
    # Create output layer
    output = tf.add(tf.matmul(x_tensor, weights), bias)
    return output


"""
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 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 stri
de
    # Function Definition from Above:
    #    conv2d_maxpool(x_tensor, conv_num_outputs, conv_ksize,
conv_strides, pool_ksize, pool_strides)
    conv1 = conv2d_maxpool(x, conv_num_outputs=32, conv_ksize=(3,3), conv_st
rides=(1,1), pool_ksize=(2,2), pool_strides=(2,2))
    conv2 = conv2d_maxpool(conv1, conv_num_outputs=64, conv_ksize=(3,3), con
v_strides=(1,1), pool_ksize=(2,2), pool_strides=(2,2))
    conv3 = conv2d_maxpool(conv2, conv_num_outputs=128, conv_ksize=(3,3), co
nv_strides=(1,1), pool_ksize=(2,2), pool_strides=(2,2))
    # TODO: Apply a Flatten Layer
    # Function Definition from Above:
    #   flatten(x_tensor)
    conv3 = flatten(conv3)

    # 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)
    fc1 = fully_conn(conv3, 256)
    fc1 = tf.nn.dropout(fc1, keep_prob)
    fc2 = fully_conn(fc1, 128)
```

```
    # TODO: Apply an Output Layer
    #    Set this to the number of classes
    # Function Definition from Above:
    #   output(x_tensor, num_outputs)
    n_classes = 10
    logits = output(fc2, n_classes)

    # TODO: return output
    return logits


"""
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=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)
```

```
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_batch, 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
    """

    # Run optimizer in session with appropriate feed dictionary
    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 [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

    # Run cost function in session
    loss = session.run(cost, feed_dict={x: feature_batch, y: label_batch, ke
ep_prob: 1.0})

    accuracy_list = []
    batch_size = 256
    # Iterate through validation samples one batch at a time because too la
```

```
rge to hold in GPU memory
    for validation_feature_batch, validation_label_batch in
helper.batch_features_labels(valid_features, valid_labels, batch_size):
        accuracy_list.append(sess.run(accuracy,
feed_dict={x:validation_feature_batch, y: validation_label_batch, keep_prob
: 1.0}))

    print('Loss: {:.3f}'.format(loss))
    print('Validation Accuracy: {:.4f}'.format(np.mean(accuracy_list)))
```

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 [15]:

```
# TODO: Tune Parameters
epochs = 12
batch_size = 256
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...')

# Uncomment to use GPU
'''
config = tf.ConfigProto(
        device_count = {'GPU': 1}
    )
'''

# Use only CPU

config = tf.ConfigProto(
    device_count = {'GPU': 0}
)
```

```
)

with tf.Session(config=config) 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, batch_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: 2.090
Validation Accuracy: 0.2483
Epoch  2, CIFAR-10 Batch 1:  Loss: 1.909
Validation Accuracy: 0.3615
Epoch  3, CIFAR-10 Batch 1:  Loss: 1.768
Validation Accuracy: 0.3711
Epoch  4, CIFAR-10 Batch 1:  Loss: 1.398
Validation Accuracy: 0.4499
Epoch  5, CIFAR-10 Batch 1:  Loss: 1.167
Validation Accuracy: 0.4692
Epoch  6, CIFAR-10 Batch 1:  Loss: 1.001
Validation Accuracy: 0.4965
Epoch  7, CIFAR-10 Batch 1:  Loss: 0.933
Validation Accuracy: 0.5083
Epoch  8, CIFAR-10 Batch 1:  Loss: 0.818
Validation Accuracy: 0.5119
Epoch  9, CIFAR-10 Batch 1:  Loss: 0.684
Validation Accuracy: 0.5384
Epoch 10, CIFAR-10 Batch 1:  Loss: 0.666
Validation Accuracy: 0.5110
Epoch 11, CIFAR-10 Batch 1:  Loss: 0.480
Validation Accuracy: 0.5363
Epoch 12, CIFAR-10 Batch 1:  Loss: 0.408
Validation Accuracy: 0.5530
```

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_training_batch(batch_i, batch_size):
                train_neural_network(sess, optimizer, keep_probability, bat
h_features, batch_labels)
            print('Epoch {:>2}, CIFAR-10 Batch {}:  '.format(epoch + 1, bat
h_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: 2.082
Validation Accuracy: 0.2668
Epoch  1, CIFAR-10 Batch 2:  Loss: 1.838
Validation Accuracy: 0.3525
Epoch  1, CIFAR-10 Batch 3:  Loss: 1.424
Validation Accuracy: 0.3977
Epoch  1, CIFAR-10 Batch 4:  Loss: 1.398
Validation Accuracy: 0.4574
Epoch  1, CIFAR-10 Batch 5:  Loss: 1.568
Validation Accuracy: 0.4433
Epoch  2, CIFAR-10 Batch 1:  Loss: 1.601
Validation Accuracy: 0.4868
Epoch  2, CIFAR-10 Batch 2:  Loss: 1.349
Validation Accuracy: 0.5154
Epoch  2, CIFAR-10 Batch 3:  Loss: 1.134
Validation Accuracy: 0.5191
Epoch  2, CIFAR-10 Batch 4:  Loss: 1.124
Validation Accuracy: 0.5385
Epoch  2, CIFAR-10 Batch 5:  Loss: 1.160
Validation Accuracy: 0.5531
Epoch  3, CIFAR-10 Batch 1:  Loss: 1.275
Validation Accuracy: 0.5385
Epoch  3, CIFAR-10 Batch 2:  Loss: 1.071
Validation Accuracy: 0.5648
Epoch  3, CIFAR-10 Batch 3:  Loss: 0.867
Validation Accuracy: 0.5783
Epoch  3, CIFAR-10 Batch 4:  Loss: 0.930
Validation Accuracy: 0.5798
Epoch  3, CIFAR-10 Batch 5:  Loss: 0.945
Validation Accuracy: 0.5961
Epoch  4, CIFAR-10 Batch 1:  Loss: 0.945
Validation Accuracy: 0.5925
Epoch  4, CIFAR-10 Batch 2:  Loss: 0.772
Validation Accuracy: 0.5904
Epoch  4, CIFAR-10 Batch 3:  Loss: 0.649
Validation Accuracy: 0.6068
Epoch  4, CIFAR-10 Batch 4:  Loss: 0.771
Validation Accuracy: 0.6061
Epoch  4, CIFAR-10 Batch 5:  Loss: 0.812
Validation Accuracy: 0.6142
Epoch  5, CIFAR-10 Batch 1:  Loss: 0.772
```

```
Validation Accuracy: 0.6038
Epoch  5, CIFAR-10 Batch 2:  Loss: 0.636
Validation Accuracy: 0.6132
Epoch  5, CIFAR-10 Batch 3:  Loss: 0.544
Validation Accuracy: 0.6410
Epoch  5, CIFAR-10 Batch 4:  Loss: 0.594
Validation Accuracy: 0.6433
Epoch  5, CIFAR-10 Batch 5:  Loss: 0.637
Validation Accuracy: 0.6391
Epoch  6, CIFAR-10 Batch 1:  Loss: 0.666
Validation Accuracy: 0.6325
Epoch  6, CIFAR-10 Batch 2:  Loss: 0.523
Validation Accuracy: 0.6355
Epoch  6, CIFAR-10 Batch 3:  Loss: 0.470
Validation Accuracy: 0.6498
Epoch  6, CIFAR-10 Batch 4:  Loss: 0.485
Validation Accuracy: 0.6427
Epoch  6, CIFAR-10 Batch 5:  Loss: 0.517
Validation Accuracy: 0.6480
Epoch  7, CIFAR-10 Batch 1:  Loss: 0.461
Validation Accuracy: 0.6610
Epoch  7, CIFAR-10 Batch 2:  Loss: 0.330
Validation Accuracy: 0.6654
Epoch  7, CIFAR-10 Batch 3:  Loss: 0.346
Validation Accuracy: 0.6684
Epoch  7, CIFAR-10 Batch 4:  Loss: 0.361
Validation Accuracy: 0.6698
Epoch  7, CIFAR-10 Batch 5:  Loss: 0.415
Validation Accuracy: 0.6284
Epoch  8, CIFAR-10 Batch 1:  Loss: 0.344
Validation Accuracy: 0.6644
Epoch  8, CIFAR-10 Batch 2:  Loss: 0.250
Validation Accuracy: 0.6821
Epoch  8, CIFAR-10 Batch 3:  Loss: 0.293
Validation Accuracy: 0.6814
Epoch  8, CIFAR-10 Batch 4:  Loss: 0.275
Validation Accuracy: 0.6686
Epoch  8, CIFAR-10 Batch 5:  Loss: 0.302
Validation Accuracy: 0.6801
Epoch  9, CIFAR-10 Batch 1:  Loss: 0.281
Validation Accuracy: 0.6788
Epoch  9, CIFAR-10 Batch 2:  Loss: 0.232
Validation Accuracy: 0.6786
Epoch  9, CIFAR-10 Batch 3:  Loss: 0.234
Validation Accuracy: 0.6785
Epoch  9, CIFAR-10 Batch 4:  Loss: 0.232
Validation Accuracy: 0.6835
Epoch  9, CIFAR-10 Batch 5:  Loss: 0.249
Validation Accuracy: 0.6815
Epoch 10, CIFAR-10 Batch 1:  Loss: 0.187
Validation Accuracy: 0.6802
Epoch 10, CIFAR-10 Batch 2:  Loss: 0.166
Validation Accuracy: 0.6927
Epoch 10, CIFAR-10 Batch 3:  Loss: 0.184
Validation Accuracy: 0.6822
Epoch 10, CIFAR-10 Batch 4:  Loss: 0.166
Validation Accuracy: 0.6984
Epoch 10, CIFAR-10 Batch 5:  Loss: 0.182
Validation Accuracy: 0.6919
Epoch 11, CIFAR-10 Batch 1:  Loss: 0.151
```

```
Validation Accuracy: 0.6992
Epoch 11, CIFAR-10 Batch 2:  Loss: 0.153
Validation Accuracy: 0.7028
Epoch 11, CIFAR-10 Batch 3:  Loss: 0.130
Validation Accuracy: 0.6855
Epoch 11, CIFAR-10 Batch 4:  Loss: 0.164
Validation Accuracy: 0.6873
Epoch 11, CIFAR-10 Batch 5:  Loss: 0.176
Validation Accuracy: 0.6935
Epoch 12, CIFAR-10 Batch 1:  Loss: 0.129
Validation Accuracy: 0.7040
Epoch 12, CIFAR-10 Batch 2:  Loss: 0.165
Validation Accuracy: 0.6902
Epoch 12, CIFAR-10 Batch 3:  Loss: 0.079
Validation Accuracy: 0.6871
Epoch 12, CIFAR-10 Batch 4:  Loss: 0.146
Validation Accuracy: 0.6887
Epoch 12, CIFAR-10 Batch 5:  Loss: 0.134
Validation Accuracy: 0.6861
```

Checkpoint

The model has been saved to disk.

Test Model

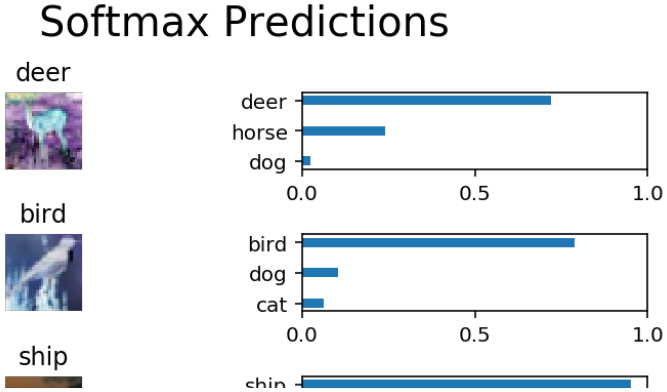
Test your model against the test dataset. This will be your final accuracy. You should have an accuracy greater than 50%. If you don't, keep tweaking the model architecture and parameters.

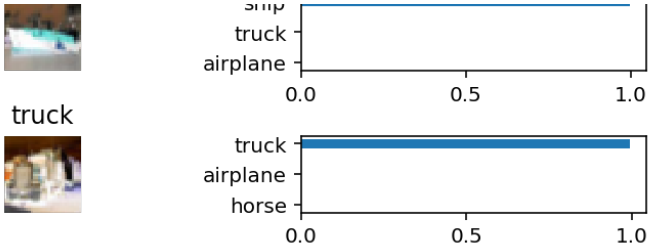
In [18]:

```
"""
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_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')

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

        for train_feature_batch, train_label_batch in
helper.batch_features_labels(test_features, test_labels, batch_size):
            test_batch_acc_total += sess.run(
                loaded_acc,
                feed_dict={loaded_x: train_feature_batch, loaded_y: train_la
bel_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.sample(
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_test_labels, random_test_predictions)


test_model()
```

```
INFO:tensorflow:Restoring parameters from ./image_classification
Testing Accuracy: 0.6837890625
```





truck

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

When submitting this project, make sure to run all the cells before saving the notebook. Save the notebook file as "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.