image_classification

September 17, 2017

1 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 [4]:
```
"""
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 p
        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)
```

```
CIFAR-10 Dataset: 171MB [01:13, 2.32MB/s]


All files found!
```

1.1 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 [7]: %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:
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
```

1.2 Implement Preprocess Functions

1.2.1 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 [9]: 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
            """
            # Allocate ndarray for normalized images.
            normalized_x = np.zeros(tuple(x.shape))
            nr_images = x.shape[0]
            # Compute max/min values.
            max_val, min_val = x.max(), x.min()
            # Transform every image.
            for image_index in range(nr_images):
                normalized_x[image_index,...] = (x[image_index, ...] - float(min_val)) / float(m


        """
```

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

```
Tests Passed
```

1.2.2 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 [11]: def one_hot_encode(x,n_values=10):
             """
             One hot encode a list of sample labels. Return a one-hot encoded vector for each la
             : x: List of sample Labels
             : return: Numpy array of one-hot encoded labels
             """

             # Let's use the One-Hot encoder method available in sklearn.
             from sklearn.preprocessing import OneHotEncoder
             enc = OneHotEncoder(n_values=n_values)
             one_hot_encoded_labels = enc.fit_transform(np.array(x).reshape(-1, 1)).toarray()
             return one_hot_encoded_labels


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

```
Tests Passed
```

1.2.3 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.

1.3 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 [12]:
```
"""
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)
```

2 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 [13]:
```
"""
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'))
```

2.1 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!

2.1.1 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 [14]: 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), 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, shape=(None), 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.
```

2.1.2 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 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 [15]: def conv2d_maxpool(x_tensor, conv_num_outputs, conv_ksize, conv_strides, pool_ksize, po
             """
             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
             """
             weights_shape = list(conv_ksize) + [x_tensor.get_shape().as_list()[3], conv_num_out
             # Define our trainable variables.
             weights = tf.Variable(tf.truncated_normal(weights_shape, stddev=5e-2))
             bias = tf.Variable(tf.zeros(conv_num_outputs))

             # 2D Convolution Layer.
             output = tf.nn.conv2d(x_tensor, weights,
                                   strides=[1, conv_strides[0], conv_strides[1], 1],
                                   padding='SAME')
             output = tf.nn.bias_add(output, bias)
             output = tf.nn.relu(output)

             # Pooling layer.
             output = tf.nn.max_pool(output,
                                     ksize=[1, pool_ksize[0], pool_ksize[1], 1],
                                     strides=[1, pool_strides[0], pool_strides[1], 1],
                                     padding='SAME')
```

```
        return output

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

```
Tests Passed
```

2.1.3 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 [16]: 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).
             """

             tensor_shape = x_tensor.get_shape().as_list()
             # Get the length of the flattened dimensions.
             flattened_shape = np.array(tensor_shape[1:]).prod()
             # Batch size is casted by tf.shape.
             return tf.reshape(x_tensor, [tf.shape(x_tensor)[0], flattened_shape])


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

```
Tests Passed
```

2.1.4 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 [17]: 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.
        """
        flattened_shape = np.array(x_tensor.get_shape().as_list()[1:]).prod()
        # Define trainable variables.
        weights = tf.Variable(tf.truncated_normal([flattened_shape, num_outputs], stddev=0.
        bias = tf.Variable(tf.zeros([num_outputs]))

        # Fully convolution layer.
        fc = tf.nn.relu(tf.add(tf.matmul(x_tensor, weights), bias))
        return fc

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

```
Tests Passed
```

2.1.5 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 [18]: 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.
             """
             flattened_shape = np.array(x_tensor.get_shape().as_list()[1:]).prod()
             # Define trainable variables.
             weights = tf.Variable(tf.truncated_normal([flattened_shape, num_outputs], stddev=0.
             bias = tf.Variable(tf.zeros([num_outputs]))

             # Output layer.
             return tf.add(tf.matmul(x_tensor, weights), bias)

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

```
Tests Passed
```

2.1.6 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 [19]: def conv_net(x, keep_prob, num_classes=10):
             """
             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
             """
             # Network architecture inspired from:
             #   https://github.com/tensorflow/models/blob/master/tutorials/image/cifar10/cifar1

             # 2 Convoultion and Max Pool Layers applied.
             conv = conv2d_maxpool(x,
                                   conv_num_outputs=64,
                                   conv_ksize=[5,5],
                                   conv_strides=[1,1],
                                   pool_ksize=[3,3],
                                   pool_strides=[2,2])

             conv = conv2d_maxpool(conv,
                                  conv_num_outputs=64,
                                  conv_ksize=[5,5],
                                  conv_strides=[1,1],
                                  pool_ksize=[3,3],
                                  pool_strides=[2,2])

             # Apply a Flatten Layer
             flattened_conv = flatten(conv)

             # 2 Fully-Connected Layers.
             fc = fully_conn(flattened_conv, 384)
             fc = fully_conn(fc, 192)
```

```
    # Dropout layer.
    fc = tf.nn.dropout(fc, keep_prob)

    # Output Layer.
    return output(fc, num_classes)

"""
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!
```

2.2 Train the Neural Network

2.2.1 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 [20]: def train_neural_network(session, optimizer, keep_probability, feature_batch, label_bat
             """
             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
             """
             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
```

2.2.2 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 [21]: 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
             """
             # Calculate batch loss and accuracy
             loss = sess.run(cost, feed_dict={x: feature_batch,
                                              y: label_batch,
                                              keep_prob: 1.})
             valid_acc = sess.run(accuracy, feed_dict={x: valid_features,
                                                       y: valid_labels,
                                                       keep_prob: 1.})
```

```
            print('Loss: {:>10.4f} Validation Accuracy: {:.6f}'.format(loss, valid_acc))
```

2.2.3 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 [23]: # TODO: Tune Parameters
         epochs = 10
         batch_size = 256
         keep_probability = 0.77
```

2.2.4 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 [26]: """
         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
                     train_neural_network(sess, optimizer, keep_probability, batch_features, bat
                 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:     1.9636 Validation Accuracy: 0.355600
Epoch  2, CIFAR-10 Batch 1:  Loss:     1.5929 Validation Accuracy: 0.451200
Epoch  3, CIFAR-10 Batch 1:  Loss:     1.3847 Validation Accuracy: 0.468200
Epoch  4, CIFAR-10 Batch 1:  Loss:     1.1620 Validation Accuracy: 0.518800
Epoch  5, CIFAR-10 Batch 1:  Loss:     0.9876 Validation Accuracy: 0.529600
Epoch  6, CIFAR-10 Batch 1:  Loss:     0.8329 Validation Accuracy: 0.527800
Epoch  7, CIFAR-10 Batch 1:  Loss:     0.6932 Validation Accuracy: 0.545800
Epoch  8, CIFAR-10 Batch 1:  Loss:     0.5070 Validation Accuracy: 0.570400
Epoch  9, CIFAR-10 Batch 1:  Loss:     0.3882 Validation Accuracy: 0.579400
Epoch 10, CIFAR-10 Batch 1:  Loss:     0.2799 Validation Accuracy: 0.588400
Epoch 11, CIFAR-10 Batch 1:  Loss:     0.2642 Validation Accuracy: 0.582200
```

```
Epoch 12, CIFAR-10 Batch 1:  Loss:     0.2327 Validation Accuracy: 0.590800
Epoch 13, CIFAR-10 Batch 1:  Loss:     0.1661 Validation Accuracy: 0.575000
Epoch 14, CIFAR-10 Batch 1:  Loss:     0.1307 Validation Accuracy: 0.575000
Epoch 15, CIFAR-10 Batch 1:  Loss:     0.1118 Validation Accuracy: 0.591200
Epoch 16, CIFAR-10 Batch 1:  Loss:     0.0521 Validation Accuracy: 0.599800
Epoch 17, CIFAR-10 Batch 1:  Loss:     0.0679 Validation Accuracy: 0.594600
Epoch 18, CIFAR-10 Batch 1:  Loss:     0.0534 Validation Accuracy: 0.596600
Epoch 19, CIFAR-10 Batch 1:  Loss:     0.0618 Validation Accuracy: 0.586800
Epoch 20, CIFAR-10 Batch 1:  Loss:     0.0228 Validation Accuracy: 0.597000
Epoch 21, CIFAR-10 Batch 1:  Loss:     0.0330 Validation Accuracy: 0.581000
Epoch 22, CIFAR-10 Batch 1:  Loss:     0.0418 Validation Accuracy: 0.566600
Epoch 23, CIFAR-10 Batch 1:  Loss:     0.0422 Validation Accuracy: 0.581200
Epoch 24, CIFAR-10 Batch 1:  Loss:     0.0442 Validation Accuracy: 0.586000
Epoch 25, CIFAR-10 Batch 1:  Loss:     0.0249 Validation Accuracy: 0.572000
Epoch 26, CIFAR-10 Batch 1:  Loss:     0.0183 Validation Accuracy: 0.584400
Epoch 27, CIFAR-10 Batch 1:  Loss:     0.0061 Validation Accuracy: 0.614400
Epoch 28, CIFAR-10 Batch 1:  Loss:     0.0123 Validation Accuracy: 0.614600
Epoch 29, CIFAR-10 Batch 1:  Loss:     0.0056 Validation Accuracy: 0.596400
Epoch 30, CIFAR-10 Batch 1:  Loss:     0.0051 Validation Accuracy: 0.614800
Epoch 31, CIFAR-10 Batch 1:  Loss:     0.0029 Validation Accuracy: 0.620400
Epoch 32, CIFAR-10 Batch 1:  Loss:     0.0017 Validation Accuracy: 0.611200
Epoch 33, CIFAR-10 Batch 1:  Loss:     0.0003 Validation Accuracy: 0.620000
Epoch 34, CIFAR-10 Batch 1:  Loss:     0.0011 Validation Accuracy: 0.612200
Epoch 35, CIFAR-10 Batch 1:  Loss:     0.0006 Validation Accuracy: 0.620600
Epoch 36, CIFAR-10 Batch 1:  Loss:     0.0005 Validation Accuracy: 0.619400
Epoch 37, CIFAR-10 Batch 1:  Loss:     0.0006 Validation Accuracy: 0.619000
Epoch 38, CIFAR-10 Batch 1:  Loss:     0.0036 Validation Accuracy: 0.604600
Epoch 39, CIFAR-10 Batch 1:  Loss:     0.0007 Validation Accuracy: 0.609400
Epoch 40, CIFAR-10 Batch 1:  Loss:     0.0010 Validation Accuracy: 0.608000
Epoch 41, CIFAR-10 Batch 1:  Loss:     0.0006 Validation Accuracy: 0.606600
Epoch 42, CIFAR-10 Batch 1:  Loss:     0.0015 Validation Accuracy: 0.590400
```

2.2.5 Fully Train the Model

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

In [27]:
```
"""
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
```

```python
            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
                        train_neural_network(sess, optimizer, keep_probability, batch_features,
                    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:     2.0656 Validation Accuracy: 0.322200
Epoch  1, CIFAR-10 Batch 2:  Loss:     1.6290 Validation Accuracy: 0.341600
Epoch  1, CIFAR-10 Batch 3:  Loss:     1.3327 Validation Accuracy: 0.424000
Epoch  1, CIFAR-10 Batch 4:  Loss:     1.3972 Validation Accuracy: 0.466600
Epoch  1, CIFAR-10 Batch 5:  Loss:     1.2904 Validation Accuracy: 0.512600
Epoch  2, CIFAR-10 Batch 1:  Loss:     1.3961 Validation Accuracy: 0.510400
Epoch  2, CIFAR-10 Batch 2:  Loss:     1.1426 Validation Accuracy: 0.503800
Epoch  2, CIFAR-10 Batch 3:  Loss:     0.9497 Validation Accuracy: 0.554000
Epoch  2, CIFAR-10 Batch 4:  Loss:     0.9899 Validation Accuracy: 0.564200
Epoch  2, CIFAR-10 Batch 5:  Loss:     0.9460 Validation Accuracy: 0.574600
Epoch  3, CIFAR-10 Batch 1:  Loss:     1.0879 Validation Accuracy: 0.600600
Epoch  3, CIFAR-10 Batch 2:  Loss:     0.7231 Validation Accuracy: 0.614800
Epoch  3, CIFAR-10 Batch 3:  Loss:     0.6470 Validation Accuracy: 0.592400
Epoch  3, CIFAR-10 Batch 4:  Loss:     0.8184 Validation Accuracy: 0.622000
Epoch  3, CIFAR-10 Batch 5:  Loss:     0.6223 Validation Accuracy: 0.645400
Epoch  4, CIFAR-10 Batch 1:  Loss:     0.8169 Validation Accuracy: 0.638200
Epoch  4, CIFAR-10 Batch 2:  Loss:     0.5638 Validation Accuracy: 0.641800
Epoch  4, CIFAR-10 Batch 3:  Loss:     0.5506 Validation Accuracy: 0.603800
Epoch  4, CIFAR-10 Batch 4:  Loss:     0.6315 Validation Accuracy: 0.646600
Epoch  4, CIFAR-10 Batch 5:  Loss:     0.4316 Validation Accuracy: 0.678800
Epoch  5, CIFAR-10 Batch 1:  Loss:     0.4625 Validation Accuracy: 0.661000
Epoch  5, CIFAR-10 Batch 2:  Loss:     0.3978 Validation Accuracy: 0.665800
Epoch  5, CIFAR-10 Batch 3:  Loss:     0.3036 Validation Accuracy: 0.665000
Epoch  5, CIFAR-10 Batch 4:  Loss:     0.4671 Validation Accuracy: 0.668200
Epoch  5, CIFAR-10 Batch 5:  Loss:     0.3029 Validation Accuracy: 0.684000
Epoch  6, CIFAR-10 Batch 1:  Loss:     0.2972 Validation Accuracy: 0.672400
Epoch  6, CIFAR-10 Batch 2:  Loss:     0.2659 Validation Accuracy: 0.668800
Epoch  6, CIFAR-10 Batch 3:  Loss:     0.2163 Validation Accuracy: 0.692800
Epoch  6, CIFAR-10 Batch 4:  Loss:     0.3797 Validation Accuracy: 0.688200
Epoch  6, CIFAR-10 Batch 5:  Loss:     0.2186 Validation Accuracy: 0.691800
Epoch  7, CIFAR-10 Batch 1:  Loss:     0.2387 Validation Accuracy: 0.682400
Epoch  7, CIFAR-10 Batch 2:  Loss:     0.2057 Validation Accuracy: 0.695400
Epoch  7, CIFAR-10 Batch 3:  Loss:     0.1624 Validation Accuracy: 0.698400
Epoch  7, CIFAR-10 Batch 4:  Loss:     0.3265 Validation Accuracy: 0.665400
```

```
Epoch  7, CIFAR-10 Batch 5:  Loss:     0.1714 Validation Accuracy: 0.667000
Epoch  8, CIFAR-10 Batch 1:  Loss:     0.1596 Validation Accuracy: 0.698200
Epoch  8, CIFAR-10 Batch 2:  Loss:     0.1601 Validation Accuracy: 0.694200
Epoch  8, CIFAR-10 Batch 3:  Loss:     0.1199 Validation Accuracy: 0.710600
Epoch  8, CIFAR-10 Batch 4:  Loss:     0.1867 Validation Accuracy: 0.690400
Epoch  8, CIFAR-10 Batch 5:  Loss:     0.1273 Validation Accuracy: 0.685000
Epoch  9, CIFAR-10 Batch 1:  Loss:     0.1648 Validation Accuracy: 0.685800
Epoch  9, CIFAR-10 Batch 2:  Loss:     0.0848 Validation Accuracy: 0.707800
Epoch  9, CIFAR-10 Batch 3:  Loss:     0.0872 Validation Accuracy: 0.713400
Epoch  9, CIFAR-10 Batch 4:  Loss:     0.1790 Validation Accuracy: 0.689800
Epoch  9, CIFAR-10 Batch 5:  Loss:     0.0798 Validation Accuracy: 0.689200
Epoch 10, CIFAR-10 Batch 1:  Loss:     0.0807 Validation Accuracy: 0.704000
Epoch 10, CIFAR-10 Batch 2:  Loss:     0.0862 Validation Accuracy: 0.693600
Epoch 10, CIFAR-10 Batch 3:  Loss:     0.0796 Validation Accuracy: 0.694400
Epoch 10, CIFAR-10 Batch 4:  Loss:     0.1008 Validation Accuracy: 0.687200
Epoch 10, CIFAR-10 Batch 5:  Loss:     0.0550 Validation Accuracy: 0.717200
Epoch 11, CIFAR-10 Batch 1:  Loss:     0.0790 Validation Accuracy: 0.719400
Epoch 11, CIFAR-10 Batch 2:  Loss:     0.0529 Validation Accuracy: 0.708200
Epoch 11, CIFAR-10 Batch 3:  Loss:     0.1026 Validation Accuracy: 0.660400
Epoch 11, CIFAR-10 Batch 4:  Loss:     0.0643 Validation Accuracy: 0.680800
Epoch 11, CIFAR-10 Batch 5:  Loss:     0.0438 Validation Accuracy: 0.711000
Epoch 12, CIFAR-10 Batch 1:  Loss:     0.0497 Validation Accuracy: 0.715000
Epoch 12, CIFAR-10 Batch 2:  Loss:     0.0604 Validation Accuracy: 0.683200
Epoch 12, CIFAR-10 Batch 3:  Loss:     0.0745 Validation Accuracy: 0.651400
Epoch 12, CIFAR-10 Batch 4:  Loss:     0.0599 Validation Accuracy: 0.700200
Epoch 12, CIFAR-10 Batch 5:  Loss:     0.0307 Validation Accuracy: 0.709800
Epoch 13, CIFAR-10 Batch 1:  Loss:     0.0442 Validation Accuracy: 0.709200
Epoch 13, CIFAR-10 Batch 2:  Loss:     0.0389 Validation Accuracy: 0.718600
Epoch 13, CIFAR-10 Batch 3:  Loss:     0.0423 Validation Accuracy: 0.659000
Epoch 13, CIFAR-10 Batch 4:  Loss:     0.0480 Validation Accuracy: 0.690000
Epoch 13, CIFAR-10 Batch 5:  Loss:     0.0422 Validation Accuracy: 0.702600
Epoch 14, CIFAR-10 Batch 1:  Loss:     0.0607 Validation Accuracy: 0.694800
Epoch 14, CIFAR-10 Batch 2:  Loss:     0.0305 Validation Accuracy: 0.707600
Epoch 14, CIFAR-10 Batch 3:  Loss:     0.0195 Validation Accuracy: 0.680200
Epoch 14, CIFAR-10 Batch 4:  Loss:     0.0230 Validation Accuracy: 0.700000
Epoch 14, CIFAR-10 Batch 5:  Loss:     0.0177 Validation Accuracy: 0.714800
Epoch 15, CIFAR-10 Batch 1:  Loss:     0.0424 Validation Accuracy: 0.702000
Epoch 15, CIFAR-10 Batch 2:  Loss:     0.0238 Validation Accuracy: 0.696600
Epoch 15, CIFAR-10 Batch 3:  Loss:     0.0416 Validation Accuracy: 0.681800
Epoch 15, CIFAR-10 Batch 4:  Loss:     0.0126 Validation Accuracy: 0.701600
Epoch 15, CIFAR-10 Batch 5:  Loss:     0.0202 Validation Accuracy: 0.715600
Epoch 16, CIFAR-10 Batch 1:  Loss:     0.0252 Validation Accuracy: 0.709600
Epoch 16, CIFAR-10 Batch 2:  Loss:     0.0184 Validation Accuracy: 0.702600
Epoch 16, CIFAR-10 Batch 3:  Loss:     0.0165 Validation Accuracy: 0.704200
Epoch 16, CIFAR-10 Batch 4:  Loss:     0.0164 Validation Accuracy: 0.708400
Epoch 16, CIFAR-10 Batch 5:  Loss:     0.0206 Validation Accuracy: 0.705400
Epoch 17, CIFAR-10 Batch 1:  Loss:     0.0104 Validation Accuracy: 0.707800
Epoch 17, CIFAR-10 Batch 2:  Loss:     0.0113 Validation Accuracy: 0.694000
```

```
Epoch 17, CIFAR-10 Batch 3:  Loss:     0.0149 Validation Accuracy: 0.674800
Epoch 17, CIFAR-10 Batch 4:  Loss:     0.0255 Validation Accuracy: 0.708800
Epoch 17, CIFAR-10 Batch 5:  Loss:     0.0117 Validation Accuracy: 0.715400
Epoch 18, CIFAR-10 Batch 1:  Loss:     0.0130 Validation Accuracy: 0.708000
Epoch 18, CIFAR-10 Batch 2:  Loss:     0.0127 Validation Accuracy: 0.705000
Epoch 18, CIFAR-10 Batch 3:  Loss:     0.0098 Validation Accuracy: 0.704400
Epoch 18, CIFAR-10 Batch 4:  Loss:     0.0122 Validation Accuracy: 0.712200
Epoch 18, CIFAR-10 Batch 5:  Loss:     0.0142 Validation Accuracy: 0.708000
Epoch 19, CIFAR-10 Batch 1:  Loss:     0.0197 Validation Accuracy: 0.693000
Epoch 19, CIFAR-10 Batch 2:  Loss:     0.0339 Validation Accuracy: 0.657200
Epoch 19, CIFAR-10 Batch 3:  Loss:     0.0050 Validation Accuracy: 0.694200
Epoch 19, CIFAR-10 Batch 4:  Loss:     0.0149 Validation Accuracy: 0.711600
Epoch 19, CIFAR-10 Batch 5:  Loss:     0.0092 Validation Accuracy: 0.692200
Epoch 20, CIFAR-10 Batch 1:  Loss:     0.0109 Validation Accuracy: 0.691400
Epoch 20, CIFAR-10 Batch 2:  Loss:     0.0049 Validation Accuracy: 0.706400
Epoch 20, CIFAR-10 Batch 3:  Loss:     0.0047 Validation Accuracy: 0.711800
Epoch 20, CIFAR-10 Batch 4:  Loss:     0.0115 Validation Accuracy: 0.705400
Epoch 20, CIFAR-10 Batch 5:  Loss:     0.0077 Validation Accuracy: 0.698000
Epoch 21, CIFAR-10 Batch 1:  Loss:     0.0166 Validation Accuracy: 0.682400
Epoch 21, CIFAR-10 Batch 2:  Loss:     0.0043 Validation Accuracy: 0.692200
Epoch 21, CIFAR-10 Batch 3:  Loss:     0.0049 Validation Accuracy: 0.696400
Epoch 21, CIFAR-10 Batch 4:  Loss:     0.0090 Validation Accuracy: 0.699400
Epoch 21, CIFAR-10 Batch 5:  Loss:     0.0055 Validation Accuracy: 0.703600
Epoch 22, CIFAR-10 Batch 1:  Loss:     0.0024 Validation Accuracy: 0.688000
Epoch 22, CIFAR-10 Batch 2:  Loss:     0.0034 Validation Accuracy: 0.703000
Epoch 22, CIFAR-10 Batch 3:  Loss:     0.0043 Validation Accuracy: 0.707200
Epoch 22, CIFAR-10 Batch 4:  Loss:     0.0030 Validation Accuracy: 0.707200
Epoch 22, CIFAR-10 Batch 5:  Loss:     0.0026 Validation Accuracy: 0.722000
Epoch 23, CIFAR-10 Batch 1:  Loss:     0.0008 Validation Accuracy: 0.717800
Epoch 23, CIFAR-10 Batch 2:  Loss:     0.0044 Validation Accuracy: 0.698800
Epoch 23, CIFAR-10 Batch 3:  Loss:     0.0036 Validation Accuracy: 0.694000
Epoch 23, CIFAR-10 Batch 4:  Loss:     0.0036 Validation Accuracy: 0.694800
Epoch 23, CIFAR-10 Batch 5:  Loss:     0.0016 Validation Accuracy: 0.707000
Epoch 24, CIFAR-10 Batch 1:  Loss:     0.0019 Validation Accuracy: 0.698200
Epoch 24, CIFAR-10 Batch 2:  Loss:     0.0020 Validation Accuracy: 0.703800
Epoch 24, CIFAR-10 Batch 3:  Loss:     0.0018 Validation Accuracy: 0.692400
Epoch 24, CIFAR-10 Batch 4:  Loss:     0.0064 Validation Accuracy: 0.711400
Epoch 24, CIFAR-10 Batch 5:  Loss:     0.0064 Validation Accuracy: 0.704200
Epoch 25, CIFAR-10 Batch 1:  Loss:     0.0037 Validation Accuracy: 0.696400
Epoch 25, CIFAR-10 Batch 2:  Loss:     0.0031 Validation Accuracy: 0.710200
Epoch 25, CIFAR-10 Batch 3:  Loss:     0.0048 Validation Accuracy: 0.697400
Epoch 25, CIFAR-10 Batch 4:  Loss:     0.0026 Validation Accuracy: 0.706400
Epoch 25, CIFAR-10 Batch 5:  Loss:     0.0083 Validation Accuracy: 0.691800
Epoch 26, CIFAR-10 Batch 1:  Loss:     0.0131 Validation Accuracy: 0.675200
Epoch 26, CIFAR-10 Batch 2:  Loss:     0.0026 Validation Accuracy: 0.700800
Epoch 26, CIFAR-10 Batch 3:  Loss:     0.0015 Validation Accuracy: 0.699600
Epoch 26, CIFAR-10 Batch 4:  Loss:     0.0021 Validation Accuracy: 0.694800
Epoch 26, CIFAR-10 Batch 5:  Loss:     0.0013 Validation Accuracy: 0.687000
```

```
Epoch 27, CIFAR-10 Batch 1:  Loss:     0.0009 Validation Accuracy: 0.694200
Epoch 27, CIFAR-10 Batch 2:  Loss:     0.0031 Validation Accuracy: 0.696600
Epoch 27, CIFAR-10 Batch 3:  Loss:     0.0019 Validation Accuracy: 0.696200
Epoch 27, CIFAR-10 Batch 4:  Loss:     0.0016 Validation Accuracy: 0.705800
Epoch 27, CIFAR-10 Batch 5:  Loss:     0.0070 Validation Accuracy: 0.696800
Epoch 28, CIFAR-10 Batch 1:  Loss:     0.0018 Validation Accuracy: 0.702400
Epoch 28, CIFAR-10 Batch 2:  Loss:     0.0004 Validation Accuracy: 0.702600
Epoch 28, CIFAR-10 Batch 3:  Loss:     0.0016 Validation Accuracy: 0.699400
Epoch 28, CIFAR-10 Batch 4:  Loss:     0.0030 Validation Accuracy: 0.693400
Epoch 28, CIFAR-10 Batch 5:  Loss:     0.0015 Validation Accuracy: 0.686000
Epoch 29, CIFAR-10 Batch 1:  Loss:     0.0003 Validation Accuracy: 0.701200
Epoch 29, CIFAR-10 Batch 2:  Loss:     0.0032 Validation Accuracy: 0.699800
Epoch 29, CIFAR-10 Batch 3:  Loss:     0.0019 Validation Accuracy: 0.697200
Epoch 29, CIFAR-10 Batch 4:  Loss:     0.0013 Validation Accuracy: 0.705400
Epoch 29, CIFAR-10 Batch 5:  Loss:     0.0010 Validation Accuracy: 0.713600
Epoch 30, CIFAR-10 Batch 1:  Loss:     0.0030 Validation Accuracy: 0.705600
Epoch 30, CIFAR-10 Batch 2:  Loss:     0.0050 Validation Accuracy: 0.698200
Epoch 30, CIFAR-10 Batch 3:  Loss:     0.0004 Validation Accuracy: 0.699400
Epoch 30, CIFAR-10 Batch 4:  Loss:     0.0009 Validation Accuracy: 0.699000
Epoch 30, CIFAR-10 Batch 5:  Loss:     0.0010 Validation Accuracy: 0.703800
Epoch 31, CIFAR-10 Batch 1:  Loss:     0.0029 Validation Accuracy: 0.703400
Epoch 31, CIFAR-10 Batch 2:  Loss:     0.0006 Validation Accuracy: 0.699400
Epoch 31, CIFAR-10 Batch 3:  Loss:     0.0001 Validation Accuracy: 0.696000
Epoch 31, CIFAR-10 Batch 4:  Loss:     0.0016 Validation Accuracy: 0.697000
Epoch 31, CIFAR-10 Batch 5:  Loss:     0.0014 Validation Accuracy: 0.698200
Epoch 32, CIFAR-10 Batch 1:  Loss:     0.0007 Validation Accuracy: 0.712800
Epoch 32, CIFAR-10 Batch 2:  Loss:     0.0008 Validation Accuracy: 0.693200
Epoch 32, CIFAR-10 Batch 3:  Loss:     0.0011 Validation Accuracy: 0.696000
Epoch 32, CIFAR-10 Batch 4:  Loss:     0.0026 Validation Accuracy: 0.710800
Epoch 32, CIFAR-10 Batch 5:  Loss:     0.0022 Validation Accuracy: 0.707200
Epoch 33, CIFAR-10 Batch 1:  Loss:     0.0006 Validation Accuracy: 0.713800
Epoch 33, CIFAR-10 Batch 2:  Loss:     0.0007 Validation Accuracy: 0.678600
Epoch 33, CIFAR-10 Batch 3:  Loss:     0.0018 Validation Accuracy: 0.695000
Epoch 33, CIFAR-10 Batch 4:  Loss:     0.0025 Validation Accuracy: 0.709000
Epoch 33, CIFAR-10 Batch 5:  Loss:     0.0007 Validation Accuracy: 0.721800
Epoch 34, CIFAR-10 Batch 1:  Loss:     0.0003 Validation Accuracy: 0.709200
Epoch 34, CIFAR-10 Batch 2:  Loss:     0.0009 Validation Accuracy: 0.695600
Epoch 34, CIFAR-10 Batch 3:  Loss:     0.0003 Validation Accuracy: 0.702600
Epoch 34, CIFAR-10 Batch 4:  Loss:     0.0041 Validation Accuracy: 0.710400
Epoch 34, CIFAR-10 Batch 5:  Loss:     0.0012 Validation Accuracy: 0.712600
Epoch 35, CIFAR-10 Batch 1:  Loss:     0.0031 Validation Accuracy: 0.703400
Epoch 35, CIFAR-10 Batch 2:  Loss:     0.0012 Validation Accuracy: 0.703400
Epoch 35, CIFAR-10 Batch 3:  Loss:     0.0001 Validation Accuracy: 0.702800
Epoch 35, CIFAR-10 Batch 4:  Loss:     0.0078 Validation Accuracy: 0.705400
Epoch 35, CIFAR-10 Batch 5:  Loss:     0.0002 Validation Accuracy: 0.709200
Epoch 36, CIFAR-10 Batch 1:  Loss:     0.0001 Validation Accuracy: 0.710000
Epoch 36, CIFAR-10 Batch 2:  Loss:     0.0005 Validation Accuracy: 0.690600
Epoch 36, CIFAR-10 Batch 3:  Loss:     0.0010 Validation Accuracy: 0.703200
```

```
Epoch 36, CIFAR-10 Batch 4:  Loss:     0.0004 Validation Accuracy: 0.704200
Epoch 36, CIFAR-10 Batch 5:  Loss:     0.0009 Validation Accuracy: 0.717400
Epoch 37, CIFAR-10 Batch 1:  Loss:     0.0007 Validation Accuracy: 0.704000
Epoch 37, CIFAR-10 Batch 2:  Loss:     0.0005 Validation Accuracy: 0.707200
Epoch 37, CIFAR-10 Batch 3:  Loss:     0.0006 Validation Accuracy: 0.705800
Epoch 37, CIFAR-10 Batch 4:  Loss:     0.0002 Validation Accuracy: 0.716200
Epoch 37, CIFAR-10 Batch 5:  Loss:     0.0029 Validation Accuracy: 0.716200
Epoch 38, CIFAR-10 Batch 1:  Loss:     0.0019 Validation Accuracy: 0.704800
Epoch 38, CIFAR-10 Batch 2:  Loss:     0.0016 Validation Accuracy: 0.701400
Epoch 38, CIFAR-10 Batch 3:  Loss:     0.0003 Validation Accuracy: 0.712000
Epoch 38, CIFAR-10 Batch 4:  Loss:     0.0002 Validation Accuracy: 0.712800
Epoch 38, CIFAR-10 Batch 5:  Loss:     0.0004 Validation Accuracy: 0.714400
Epoch 39, CIFAR-10 Batch 1:  Loss:     0.0007 Validation Accuracy: 0.701400
Epoch 39, CIFAR-10 Batch 2:  Loss:     0.0001 Validation Accuracy: 0.716000
Epoch 39, CIFAR-10 Batch 3:  Loss:     0.0001 Validation Accuracy: 0.718200
Epoch 39, CIFAR-10 Batch 4:  Loss:     0.0006 Validation Accuracy: 0.713400
Epoch 39, CIFAR-10 Batch 5:  Loss:     0.0002 Validation Accuracy: 0.722600
Epoch 40, CIFAR-10 Batch 1:  Loss:     0.0012 Validation Accuracy: 0.696600
Epoch 40, CIFAR-10 Batch 2:  Loss:     0.0007 Validation Accuracy: 0.706600
Epoch 40, CIFAR-10 Batch 3:  Loss:     0.0005 Validation Accuracy: 0.716000
Epoch 40, CIFAR-10 Batch 4:  Loss:     0.0006 Validation Accuracy: 0.703400
Epoch 40, CIFAR-10 Batch 5:  Loss:     0.0008 Validation Accuracy: 0.711600
Epoch 41, CIFAR-10 Batch 1:  Loss:     0.0003 Validation Accuracy: 0.719000
Epoch 41, CIFAR-10 Batch 2:  Loss:     0.0002 Validation Accuracy: 0.707600
Epoch 41, CIFAR-10 Batch 3:  Loss:     0.0004 Validation Accuracy: 0.720200
Epoch 41, CIFAR-10 Batch 4:  Loss:     0.0000 Validation Accuracy: 0.713200
Epoch 41, CIFAR-10 Batch 5:  Loss:     0.0025 Validation Accuracy: 0.723000
Epoch 42, CIFAR-10 Batch 1:  Loss:     0.0003 Validation Accuracy: 0.715800
Epoch 42, CIFAR-10 Batch 2:  Loss:     0.0000 Validation Accuracy: 0.709200
Epoch 42, CIFAR-10 Batch 3:  Loss:     0.0008 Validation Accuracy: 0.707000
Epoch 42, CIFAR-10 Batch 4:  Loss:     0.0009 Validation Accuracy: 0.703200
Epoch 42, CIFAR-10 Batch 5:  Loss:     0.0003 Validation Accuracy: 0.725600
```

3 Checkpoint

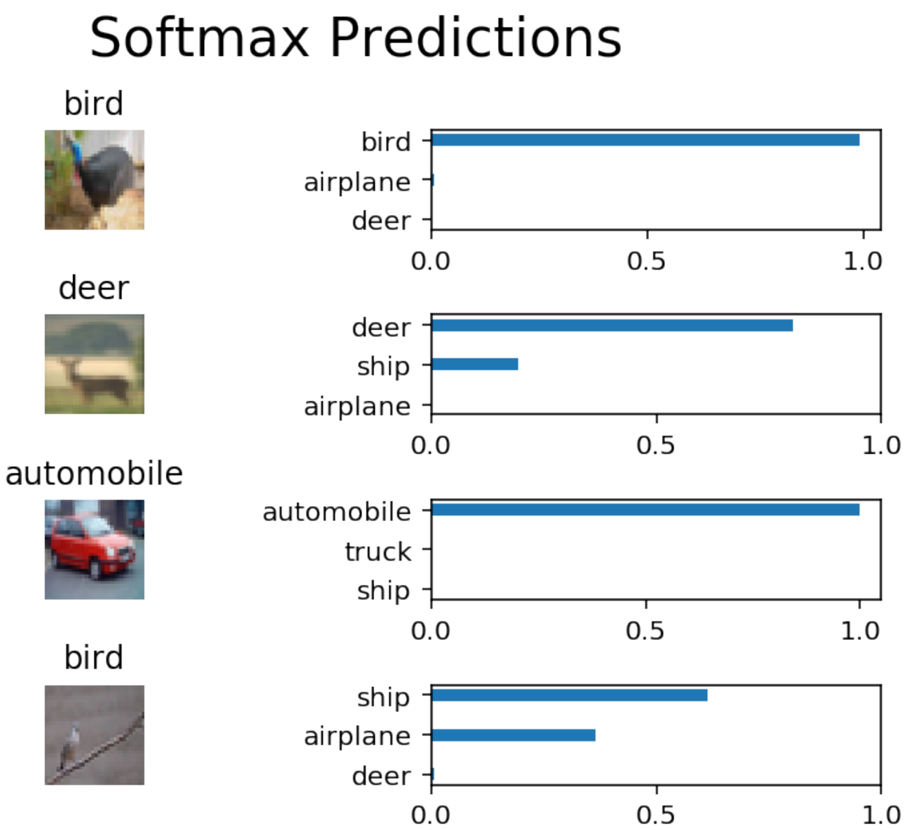
The model has been saved to disk. ## Test Model Test your model against the test dataset. This will be your final accuracy. You should have an accuracy greater than 50%. If you don't, keep tweaking the model architecture and parameters.

In [28]:
```
"""
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
            test_batch_acc_total += sess.run(
                loaded_acc,
                feed_dict={loaded_x: train_feature_batch, loaded_y: train_label_batch,
            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(te
        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
        helper.display_image_predictions(random_test_features, random_test_labels, rand


test_model()
```

```
INFO:tensorflow:Restoring parameters from ./image_classification
Testing Accuracy: 0.71318359375
```



3.1 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!

3.2 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.