dlnd_image_classification

July 20, 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 [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-10 Dataset') as p
                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!
```

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

        import helper
        import numpy as np

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

```
Stats of batch 1:
Samples: 10000
Label Counts: {0: 1005, 1: 974, 2: 1032, 3: 1016, 4: 999, 5: 937, 6: 1030, 7: 1001, 8: 1025, 9:
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 [5]:
```python
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
    """
    x_norm = x.reshape(x.size)
    x_norm = (x_norm - min(x_norm))/(max(x_norm)-min(x_norm))

    return x_norm.reshape(x.shape)

"""
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: Don't reinvent the wheel.

```
In [6]: def one_hot_encode(x):
            """
            One hot encode a list of sample labels. Return a one-hot encoded vector for each lab
            : x: List of sample Labels
            : return: Numpy array of one-hot encoded labels
            """

            one_hot_array = np.zeros((len(x), 10))
            for index in range(len(x)):
                val_index = x[index]
                one_hot_array[index][val_index] = 1

            return one_hot_array


        """
        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 [7]: """
        DON'T MODIFY ANYTHING IN THIS CELL
        """
```

```
# Preprocess Training, Validation, and Testing Data
helper.preprocess_and_save_data(cifar10_dataset_folder_path, normalize, one_hot_encode)
```

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 [8]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
import pickle
import problem_unittests as tests
import helper

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

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 [9]: import tensorflow as tf

        def neural_net_image_input(image_shape):
            """
            Return a Tensor for a batch of image input
            : image_shape: Shape of the images
            : return: Tensor for image input.
            """
            return tf.placeholder(tf.float32, shape=(None, image_shape[0], image_shape[1], image



        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.
            """
            return tf.placeholder(tf.uint8, (None, n_classes), name='y')


        def neural_net_keep_prob_input():
            """
            Return a Tensor for keep probability
            : return: Tensor for keep probability.
            """
            return tf.placeholder(tf.float32, 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.

```
In [11]: 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
             """
             x_tensor_dims = x_tensor._shape.ndims
             channel_num = x_tensor._shape.dims[x_tensor_dims - 1].value

             mu = 0
             sigma = 0.1

             conv_weight = tf.Variable(tf.truncated_normal(shape=(conv_ksize[0], conv_ksize[1],
             conv_bias = tf.Variable(tf.zeros(conv_num_outputs))

             conv = tf.nn.conv2d(x_tensor, conv_weight, strides=[1, conv_strides[0], conv_stride
             conv = tf.nn.relu(conv)

             return tf.nn.max_pool(conv, ksize=[1, pool_ksize[1], pool_ksize[1], 1], strides=[1,


         """
         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 [12]: 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).
             """
             shaped = x_tensor.get_shape().as_list()
             reshaped = tf.reshape(x_tensor, [-1, shaped[1] * shaped[2] * shaped[3]])

             return reshaped


         """
         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 [13]: 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.
             """
             weight = tf.Variable(tf.truncated_normal(shape=[x_tensor.get_shape().as_list()[1],
             bias = tf.Variable(tf.zeros(shape=num_outputs))

             return tf.nn.relu(tf.matmul(x_tensor, weight) + bias)


         """
         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 [14]: 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.
             """
             weight = tf.Variable(tf.truncated_normal(shape=[x_tensor.get_shape().as_list()[1],
             bias = tf.Variable(tf.zeros(shape=num_outputs))

             return tf.matmul(x_tensor, weight) + 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 [15]: 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
             """
             #    Play around with different number of outputs, kernel size and stride
             # Function Definition from Above:
             #    conv2d_maxpool(x_tensor, conv_num_outputs, conv_ksize, conv_strides, pool_ksiz

             conv_num_outputs = 10
             conv_ksize = (3, 3)
             conv_strides = (1, 1)
             pool_ksize = (2, 2)
             pool_strides = (2, 2)
             x_tensor = conv2d_maxpool(x, conv_num_outputs, conv_ksize, conv_strides, pool_ksize
             # Function Definition from Above:
             #   flatten(x_tensor)

             x_tensor = flatten(x_tensor)
             #    Play around with different number of outputs
             # Function Definition from Above:
             #   fully_conn(x_tensor, num_outputs)
             num_outputs = 100
             x_tensor = fully_conn(x_tensor, num_outputs)
             x_tensor = tf.nn.dropout(x_tensor, keep_prob)

             #    Set this to the number of classes
             # Function Definition from Above:
             #   output(x_tensor, num_outputs)
             num_outputs = 10
             model = output(x_tensor, num_outputs)

             # TODO: return output
             return model


         """
         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 [16]: 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
             """
             feed_dict = {
                     x: feature_batch,
                     y: label_batch,
                     keep_prob: keep_probability}
```

```
            session.run(optimizer, feed_dict=feed_dict)


        """
        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 [17]: 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
             """
             current_cost = session.run(cost,feed_dict={x: feature_batch, y: label_batch, keep_p
             valid_accuracy = session.run(accuracy,feed_dict={x: valid_features, y: valid_labels

             print('Loss: {:<8.3} Valid Accuracy: {:<5.3}'.format(current_cost,valid_accuracy))
```

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

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 [21]: """
         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: 2.09     Valid Accuracy: 0.321
Epoch  2, CIFAR-10 Batch 1:  Loss: 1.93     Valid Accuracy: 0.397
Epoch  3, CIFAR-10 Batch 1:  Loss: 1.81     Valid Accuracy: 0.432
Epoch  4, CIFAR-10 Batch 1:  Loss: 1.68     Valid Accuracy: 0.458
Epoch  5, CIFAR-10 Batch 1:  Loss: 1.58     Valid Accuracy: 0.461
Epoch  6, CIFAR-10 Batch 1:  Loss: 1.48     Valid Accuracy: 0.478
Epoch  7, CIFAR-10 Batch 1:  Loss: 1.38     Valid Accuracy: 0.484
Epoch  8, CIFAR-10 Batch 1:  Loss: 1.3      Valid Accuracy: 0.495
Epoch  9, CIFAR-10 Batch 1:  Loss: 1.25     Valid Accuracy: 0.498
Epoch 10, CIFAR-10 Batch 1:  Loss: 1.14     Valid Accuracy: 0.504
Epoch 11, CIFAR-10 Batch 1:  Loss: 1.1      Valid Accuracy: 0.512
Epoch 12, CIFAR-10 Batch 1:  Loss: 1.04     Valid Accuracy: 0.513
Epoch 13, CIFAR-10 Batch 1:  Loss: 0.968    Valid Accuracy: 0.526
Epoch 14, CIFAR-10 Batch 1:  Loss: 0.916    Valid Accuracy: 0.525
Epoch 15, CIFAR-10 Batch 1:  Loss: 0.92     Valid Accuracy: 0.522
Epoch 16, CIFAR-10 Batch 1:  Loss: 0.854    Valid Accuracy: 0.526
Epoch 17, CIFAR-10 Batch 1:  Loss: 0.794    Valid Accuracy: 0.525
Epoch 18, CIFAR-10 Batch 1:  Loss: 0.773    Valid Accuracy: 0.528
Epoch 19, CIFAR-10 Batch 1:  Loss: 0.723    Valid Accuracy: 0.532
Epoch 20, CIFAR-10 Batch 1:  Loss: 0.684    Valid Accuracy: 0.536
```

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 [22]: """
         DON'T MODIFY ANYTHING IN THIS CELL
         """
         save_model_path = './image_classification'
```

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

    # Training cycle
    for epoch in range(epochs):
        # Loop over all batches
        n_batches = 5
        for batch_i in range(1, n_batches + 1):
            for batch_features, batch_labels in helper.load_preprocess_training_batch(b
                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: 2.1      Valid Accuracy: 0.31
Epoch  1, CIFAR-10 Batch 2:  Loss: 1.8      Valid Accuracy: 0.391
Epoch  1, CIFAR-10 Batch 3:  Loss: 1.64     Valid Accuracy: 0.411
Epoch  1, CIFAR-10 Batch 4:  Loss: 1.61     Valid Accuracy: 0.438
Epoch  1, CIFAR-10 Batch 5:  Loss: 1.65     Valid Accuracy: 0.467
Epoch  2, CIFAR-10 Batch 1:  Loss: 1.74     Valid Accuracy: 0.468
Epoch  2, CIFAR-10 Batch 2:  Loss: 1.39     Valid Accuracy: 0.491
Epoch  2, CIFAR-10 Batch 3:  Loss: 1.29     Valid Accuracy: 0.499
Epoch  2, CIFAR-10 Batch 4:  Loss: 1.33     Valid Accuracy: 0.514
Epoch  2, CIFAR-10 Batch 5:  Loss: 1.49     Valid Accuracy: 0.525
Epoch  3, CIFAR-10 Batch 1:  Loss: 1.55     Valid Accuracy: 0.517
Epoch  3, CIFAR-10 Batch 2:  Loss: 1.2      Valid Accuracy: 0.529
Epoch  3, CIFAR-10 Batch 3:  Loss: 1.15     Valid Accuracy: 0.531
Epoch  3, CIFAR-10 Batch 4:  Loss: 1.27     Valid Accuracy: 0.533
Epoch  3, CIFAR-10 Batch 5:  Loss: 1.38     Valid Accuracy: 0.55
Epoch  4, CIFAR-10 Batch 1:  Loss: 1.41     Valid Accuracy: 0.545
Epoch  4, CIFAR-10 Batch 2:  Loss: 1.09     Valid Accuracy: 0.556
Epoch  4, CIFAR-10 Batch 3:  Loss: 1.09     Valid Accuracy: 0.546
Epoch  4, CIFAR-10 Batch 4:  Loss: 1.15     Valid Accuracy: 0.556
Epoch  4, CIFAR-10 Batch 5:  Loss: 1.31     Valid Accuracy: 0.556
Epoch  5, CIFAR-10 Batch 1:  Loss: 1.28     Valid Accuracy: 0.558
Epoch  5, CIFAR-10 Batch 2:  Loss: 1.03     Valid Accuracy: 0.566
Epoch  5, CIFAR-10 Batch 3:  Loss: 0.96     Valid Accuracy: 0.565
Epoch  5, CIFAR-10 Batch 4:  Loss: 1.08     Valid Accuracy: 0.566
Epoch  5, CIFAR-10 Batch 5:  Loss: 1.23     Valid Accuracy: 0.581
Epoch  6, CIFAR-10 Batch 1:  Loss: 1.18     Valid Accuracy: 0.576
Epoch  6, CIFAR-10 Batch 2:  Loss: 0.946    Valid Accuracy: 0.574
Epoch  6, CIFAR-10 Batch 3:  Loss: 0.943    Valid Accuracy: 0.576
```

```
Epoch  6, CIFAR-10 Batch 4:  Loss: 0.993    Valid Accuracy: 0.581
Epoch  6, CIFAR-10 Batch 5:  Loss: 1.14     Valid Accuracy: 0.581
Epoch  7, CIFAR-10 Batch 1:  Loss: 1.15     Valid Accuracy: 0.587
Epoch  7, CIFAR-10 Batch 2:  Loss: 0.884    Valid Accuracy: 0.59
Epoch  7, CIFAR-10 Batch 3:  Loss: 0.855    Valid Accuracy: 0.589
Epoch  7, CIFAR-10 Batch 4:  Loss: 0.966    Valid Accuracy: 0.586
Epoch  7, CIFAR-10 Batch 5:  Loss: 1.07     Valid Accuracy: 0.596
Epoch  8, CIFAR-10 Batch 1:  Loss: 1.02     Valid Accuracy: 0.596
Epoch  8, CIFAR-10 Batch 2:  Loss: 0.827    Valid Accuracy: 0.597
Epoch  8, CIFAR-10 Batch 3:  Loss: 0.826    Valid Accuracy: 0.593
Epoch  8, CIFAR-10 Batch 4:  Loss: 0.896    Valid Accuracy: 0.594
Epoch  8, CIFAR-10 Batch 5:  Loss: 1.02     Valid Accuracy: 0.596
Epoch  9, CIFAR-10 Batch 1:  Loss: 0.968    Valid Accuracy: 0.598
Epoch  9, CIFAR-10 Batch 2:  Loss: 0.782    Valid Accuracy: 0.601
Epoch  9, CIFAR-10 Batch 3:  Loss: 0.746    Valid Accuracy: 0.602
Epoch  9, CIFAR-10 Batch 4:  Loss: 0.849    Valid Accuracy: 0.596
Epoch  9, CIFAR-10 Batch 5:  Loss: 0.958    Valid Accuracy: 0.597
Epoch 10, CIFAR-10 Batch 1:  Loss: 0.984    Valid Accuracy: 0.591
Epoch 10, CIFAR-10 Batch 2:  Loss: 0.736    Valid Accuracy: 0.603
Epoch 10, CIFAR-10 Batch 3:  Loss: 0.672    Valid Accuracy: 0.608
Epoch 10, CIFAR-10 Batch 4:  Loss: 0.794    Valid Accuracy: 0.601
Epoch 10, CIFAR-10 Batch 5:  Loss: 0.92     Valid Accuracy: 0.607
Epoch 11, CIFAR-10 Batch 1:  Loss: 0.895    Valid Accuracy: 0.606
Epoch 11, CIFAR-10 Batch 2:  Loss: 0.711    Valid Accuracy: 0.61
Epoch 11, CIFAR-10 Batch 3:  Loss: 0.663    Valid Accuracy: 0.61
Epoch 11, CIFAR-10 Batch 4:  Loss: 0.762    Valid Accuracy: 0.608
Epoch 11, CIFAR-10 Batch 5:  Loss: 0.887    Valid Accuracy: 0.603
Epoch 12, CIFAR-10 Batch 1:  Loss: 0.902    Valid Accuracy: 0.613
Epoch 12, CIFAR-10 Batch 2:  Loss: 0.671    Valid Accuracy: 0.606
Epoch 12, CIFAR-10 Batch 3:  Loss: 0.568    Valid Accuracy: 0.615
Epoch 12, CIFAR-10 Batch 4:  Loss: 0.736    Valid Accuracy: 0.604
Epoch 12, CIFAR-10 Batch 5:  Loss: 0.855    Valid Accuracy: 0.611
Epoch 13, CIFAR-10 Batch 1:  Loss: 0.899    Valid Accuracy: 0.615
Epoch 13, CIFAR-10 Batch 2:  Loss: 0.648    Valid Accuracy: 0.616
Epoch 13, CIFAR-10 Batch 3:  Loss: 0.587    Valid Accuracy: 0.614
Epoch 13, CIFAR-10 Batch 4:  Loss: 0.694    Valid Accuracy: 0.613
Epoch 13, CIFAR-10 Batch 5:  Loss: 0.818    Valid Accuracy: 0.611
Epoch 14, CIFAR-10 Batch 1:  Loss: 0.865    Valid Accuracy: 0.619
Epoch 14, CIFAR-10 Batch 2:  Loss: 0.581    Valid Accuracy: 0.618
Epoch 14, CIFAR-10 Batch 3:  Loss: 0.565    Valid Accuracy: 0.618
Epoch 14, CIFAR-10 Batch 4:  Loss: 0.651    Valid Accuracy: 0.615
Epoch 14, CIFAR-10 Batch 5:  Loss: 0.761    Valid Accuracy: 0.621
Epoch 15, CIFAR-10 Batch 1:  Loss: 0.852    Valid Accuracy: 0.62
Epoch 15, CIFAR-10 Batch 2:  Loss: 0.575    Valid Accuracy: 0.624
Epoch 15, CIFAR-10 Batch 3:  Loss: 0.533    Valid Accuracy: 0.619
Epoch 15, CIFAR-10 Batch 4:  Loss: 0.604    Valid Accuracy: 0.608
Epoch 15, CIFAR-10 Batch 5:  Loss: 0.783    Valid Accuracy: 0.605
Epoch 16, CIFAR-10 Batch 1:  Loss: 0.826    Valid Accuracy: 0.618
```

```
Epoch 16, CIFAR-10 Batch 2:  Loss: 0.607    Valid Accuracy: 0.623
Epoch 16, CIFAR-10 Batch 3:  Loss: 0.536    Valid Accuracy: 0.609
Epoch 16, CIFAR-10 Batch 4:  Loss: 0.546    Valid Accuracy: 0.617
Epoch 16, CIFAR-10 Batch 5:  Loss: 0.72     Valid Accuracy: 0.608
Epoch 17, CIFAR-10 Batch 1:  Loss: 0.791    Valid Accuracy: 0.622
Epoch 17, CIFAR-10 Batch 2:  Loss: 0.538    Valid Accuracy: 0.621
Epoch 17, CIFAR-10 Batch 3:  Loss: 0.486    Valid Accuracy: 0.621
Epoch 17, CIFAR-10 Batch 4:  Loss: 0.571    Valid Accuracy: 0.623
Epoch 17, CIFAR-10 Batch 5:  Loss: 0.68     Valid Accuracy: 0.616
Epoch 18, CIFAR-10 Batch 1:  Loss: 0.783    Valid Accuracy: 0.621
Epoch 18, CIFAR-10 Batch 2:  Loss: 0.536    Valid Accuracy: 0.624
Epoch 18, CIFAR-10 Batch 3:  Loss: 0.462    Valid Accuracy: 0.625
Epoch 18, CIFAR-10 Batch 4:  Loss: 0.555    Valid Accuracy: 0.622
Epoch 18, CIFAR-10 Batch 5:  Loss: 0.642    Valid Accuracy: 0.617
Epoch 19, CIFAR-10 Batch 1:  Loss: 0.728    Valid Accuracy: 0.622
Epoch 19, CIFAR-10 Batch 2:  Loss: 0.507    Valid Accuracy: 0.625
Epoch 19, CIFAR-10 Batch 3:  Loss: 0.442    Valid Accuracy: 0.631
Epoch 19, CIFAR-10 Batch 4:  Loss: 0.495    Valid Accuracy: 0.624
Epoch 19, CIFAR-10 Batch 5:  Loss: 0.606    Valid Accuracy: 0.62
Epoch 20, CIFAR-10 Batch 1:  Loss: 0.726    Valid Accuracy: 0.624
Epoch 20, CIFAR-10 Batch 2:  Loss: 0.493    Valid Accuracy: 0.626
Epoch 20, CIFAR-10 Batch 3:  Loss: 0.466    Valid Accuracy: 0.623
Epoch 20, CIFAR-10 Batch 4:  Loss: 0.487    Valid Accuracy: 0.616
Epoch 20, CIFAR-10 Batch 5:  Loss: 0.592    Valid Accuracy: 0.622
```

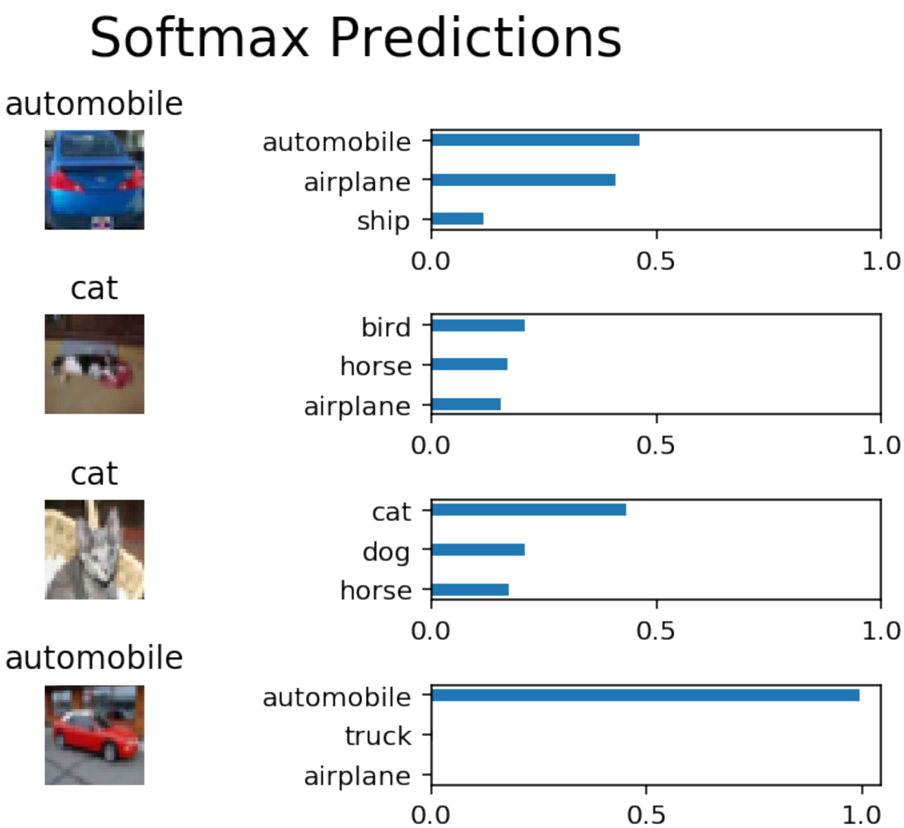
3 Checkpoint

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 [24]:
```
"""
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', 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 test_feature_batch, test_label_batch in helper.batch_features_labels(test_f
            test_batch_acc_total += sess.run(
                loaded_acc,
                feed_dict={loaded_x: test_feature_batch, loaded_y: test_label_batch, lo
            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: 0.6132318037974683
```



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. However, you might notice people are getting scores well above 80%. 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.