# Project 05 (Image Classification)

# Image Classification

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

## Get the Data

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

# Data

Download the CIFAR-10 dataset for python (https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz) (https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz)). CIFAR-10 is an established computer-vision dataset used for object recognition. It is a subset of the 80 million tiny images dataset and consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Let's get the data by running the following function

In [1]:

```
from urllib.request import urlretrieve
from os.path import isfile, isdir
from tqdm import tqdm
import tarfile

cifar10_dataset_folder_path = 'cifar-10-batches-py'

class DLProgress(tqdm):
    last_block = 0

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

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

# Explore the Data

The dataset is broken into batches to prevent the 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

In [2]:

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

import helper
import numpy as np

# Explore the dataset
batch_id = 2
sample_id = 4
helper.display_stats(cifar10_dataset_folder_path, batch_id, sample_id)
```

```
Stats of batch 2:
Samples: 10000
Label Counts: {0: 984, 1: 1007, 2: 1010, 3: 995, 4: 1010, 5: 988, 6: 1008,
7: 1026, 8: 987, 9: 985}
First 20 Labels: [1, 6, 6, 8, 8, 3, 4, 6, 0, 6, 0, 3, 6, 6, 5, 4, 8, 3, 2,
6]

Example of Image 4:
Image - Min Value: 0 Max Value: 255
Image - Shape: (32, 32, 3)
Label - Label Id: 8 Name: ship
```

## Implement Preprocess Functions

### Normalize

The `normalize` function takes in image data, `x` , and return it as a normalized Numpy array. The values is in the range of 0 to 1, inclusive. The return object is the same shape as `x` .

In [3]:

```
def normalize(x):
    return x - np.min(x) / (np.max(x) - np.min(x))
```

### One-hot encode

The input, `x` , are a list of labels. The function returns the list of labels as One-Hot encoded Numpy array. The possible values for labels are 0 to 9.

In [4]:

```
def one_hot_encode(x):
    z = np.zeros((len(x), 10))
    z[list(np.indices((len(x),))) + [x]] = 1
    return z
```

## Preprocess all the data and save it

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]:

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

# Load the Preprocessed Validation data

The preprocessed data has been saved to disk.

In [7]:

```
import pickle

valid_features, valid_labels = pickle.load(open('preprocess_validation.p', mode='rb'))
```

## Build the network

### Input

The neural network reads the image data, one-hot encoded labels, and dropout keep probability.

In [8]:

```
import tensorflow as tf

def neural_net_image_input(image_shape):
    return tf.placeholder(tf.float32, shape=(None,)+image_shape, name='x')


def neural_net_label_input(n_classes):
    return tf.placeholder(tf.float32, shape=(None, n_classes), name='y')


def neural_net_keep_prob_input():
    return tf.placeholder(tf.float32, name='keep_prob')
```

## Convolution and Max Pooling Layer

Convolution layers have a lot of success with images. The below function is implemented to apply convolution then max pooling. It:

- Creates the weight and bias using `conv_ksize` , `conv_num_outputs` and the shape of `x_tensor` .
- Applies a convolution to `x_tensor` using weight and `conv_strides` .
- Adds bias
- Adds a nonlinear activation to the convolution.
- Applies Max Pooling using `pool_ksize` and `pool_strides` .

In [9]:

```
def conv2d_maxpool(x_tensor, conv_num_outputs, conv_ksize, conv_strides, pool_ksize, pool_s
    """
    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
    """

    # Weight and bias
    weight = tf.Variable(tf.truncated_normal([*conv_ksize, x_tensor.shape.as_list()[3], cor
    bias = tf.Variable(tf.zeros(conv_num_outputs))

    # Apply Convolution
    conv_layer = tf.nn.conv2d(x_tensor,
                              weight,
                              strides=[1, *conv_strides, 1],
                              padding='SAME')

    # Add bias
    conv_layer = tf.nn.bias_add(conv_layer, bias)

    # Apply activation function
    conv_layer = tf.nn.relu(conv_layer)

    # Apply Max Pooling
    conv_layer = tf.nn.max_pool(conv_layer,
                                ksize=[1, *pool_ksize, 1],
                                strides=[1, *pool_strides, 1],
                                padding='SAME')

    return conv_layer
```

### Flatten Layer

changes the dimension of `x_tensor` from a 4-D tensor to a 2-D tensor. The output is of the shape (*Batch Size*, *Flattened Image Size*).

In [10]:

```
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).
    """
    return tf.contrib.layers.flatten(x_tensor)
```

### Fully-Connected Layer

Applies a fully connected layer to `x_tensor` with the shape (*Batch Size*, *num_outputs*).

In [11]:

```
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.
    """
    return tf.layers.dense(x_tensor, num_outputs, activation=tf.nn.relu)
```

### Output Layer

Applies a fully connected layer to `x_tensor` with the shape (*Batch Size*, *num_outputs*).

In [12]:

```
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.
    """
    return tf.layers.dense(x_tensor, num_outputs)
```

### Create Convolutional Model

The below function takes in a batch of images, `x`, and outputs logits. It uses the layers created above to create this model:

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

In [14]:

```
def conv_net(x, keep_prob):
    """
    Create a convolutional neural network model
    : x: Placeholder tensor that holds image data.
    : keep_prob: Placeholder tensor that hold dropout keep probability.
    : return: Tensor that represents logits
    """
    x = conv2d_maxpool(x, 64, (5, 5), (1, 1), (3, 3), (2, 2))
    x = tf.layers.dropout(x, rate=keep_prob)
    x = conv2d_maxpool(x, 64, (5, 5), (1, 1), (3, 3), (2, 2))
    x = tf.layers.dropout(x, rate=keep_prob)
    x = flatten(x)
    x = fully_conn(x, 384)
    x = fully_conn(x, 192)
    x = output(x, 10)

    return x

##############################
## 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')
```

# Train the Neural Network

## Single Optimization

The optimization uses `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

In [15]:

```
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
    """
    session.run(optimizer, feed_dict={
                x: feature_batch,
                y: label_batch,
                keep_prob: keep_probability})
```

## Show Stats

Print loss and validation accuracy. Use the global variables `valid_features` and `valid_labels` to calculate validation accuracy. Use a keep probability of `1.0` to calculate the loss and validation accuracy.

In [16]:

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

## Hyperparameters

Tune the following parameters:

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

- Set `keep_probability` to the probability of keeping a node using dropout

In [17]:

```
epochs = 100
batch_size = 256
keep_probability = 0.75
```

### 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 we iterate on the model to get a better accuracy. Once the final validation accuracy is 50% or greater, the model is run on all the data.

In [16]:

```
print('Checking the Training on a Single Batch...')
with tf.Session() as sess:
    # Initializing the variables
    sess.run(tf.global_variables_initializer())

    # Training cycle
    for epoch in range(epochs):
        batch_i = 1
        for batch_features, batch_labels in helper.load_preprocess_training_batch(batch_i,
            train_neural_network(sess, optimizer, keep_probability, batch_features, batch_l
        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:     2.0985 Validation Accuracy: 0.302600
Epoch  2, CIFAR-10 Batch 1:  Loss:     1.7820 Validation Accuracy: 0.417800
Epoch  3, CIFAR-10 Batch 1:  Loss:     1.5150 Validation Accuracy: 0.451200
Epoch  4, CIFAR-10 Batch 1:  Loss:     1.3137 Validation Accuracy: 0.494200
Epoch  5, CIFAR-10 Batch 1:  Loss:     1.1109 Validation Accuracy: 0.504600
Epoch  6, CIFAR-10 Batch 1:  Loss:     0.8968 Validation Accuracy: 0.512200
Epoch  7, CIFAR-10 Batch 1:  Loss:     0.7711 Validation Accuracy: 0.501000
Epoch  8, CIFAR-10 Batch 1:  Loss:     0.6200 Validation Accuracy: 0.551800
Epoch  9, CIFAR-10 Batch 1:  Loss:     0.4558 Validation Accuracy: 0.577800
Epoch 10, CIFAR-10 Batch 1:  Loss:     0.3167 Validation Accuracy: 0.581800
Epoch 11, CIFAR-10 Batch 1:  Loss:     0.2201 Validation Accuracy: 0.559600
Epoch 12, CIFAR-10 Batch 1:  Loss:     0.1516 Validation Accuracy: 0.569600
Epoch 13, CIFAR-10 Batch 1:  Loss:     0.1788 Validation Accuracy: 0.590800
Epoch 14, CIFAR-10 Batch 1:  Loss:     0.1147 Validation Accuracy: 0.592800
Epoch 15, CIFAR-10 Batch 1:  Loss:     0.0984 Validation Accuracy: 0.573800
Epoch 16, CIFAR-10 Batch 1:  Loss:     0.1360 Validation Accuracy: 0.565600
Epoch 17, CIFAR-10 Batch 1:  Loss:     0.0995 Validation Accuracy: 0.547400
Epoch 18, CIFAR-10 Batch 1:  Loss:     0.0893 Validation Accuracy: 0.544600
Epoch 19, CIFAR-10 Batch 1:  Loss:     0.0592 Validation Accuracy: 0.566600
Epoch 20, CIFAR-10 Batch 1:  Loss:     0.0458 Validation Accuracy: 0.576600
Epoch 21, CIFAR-10 Batch 1:  Loss:     0.0532 Validation Accuracy: 0.564000
Epoch 22, CIFAR-10 Batch 1:  Loss:     0.0365 Validation Accuracy: 0.578600
Epoch 23, CIFAR-10 Batch 1:  Loss:     0.0380 Validation Accuracy: 0.559200
Epoch 24, CIFAR-10 Batch 1:  Loss:     0.0175 Validation Accuracy: 0.568600
Epoch 25, CIFAR-10 Batch 1:  Loss:     0.0230 Validation Accuracy: 0.564800
Epoch 26, CIFAR-10 Batch 1:  Loss:     0.0149 Validation Accuracy: 0.589200
Epoch 27, CIFAR-10 Batch 1:  Loss:     0.0208 Validation Accuracy: 0.595800
Epoch 28, CIFAR-10 Batch 1:  Loss:     0.0145 Validation Accuracy: 0.580800
Epoch 29, CIFAR-10 Batch 1:  Loss:     0.0086 Validation Accuracy: 0.592000
Epoch 30, CIFAR-10 Batch 1:  Loss:     0.0094 Validation Accuracy: 0.581600
Epoch 31, CIFAR-10 Batch 1:  Loss:     0.0064 Validation Accuracy: 0.599600
Epoch 32, CIFAR-10 Batch 1:  Loss:     0.0084 Validation Accuracy: 0.581400
Epoch 33, CIFAR-10 Batch 1:  Loss:     0.0046 Validation Accuracy: 0.602800
Epoch 34, CIFAR-10 Batch 1:  Loss:     0.0031 Validation Accuracy: 0.602800
Epoch 35, CIFAR-10 Batch 1:  Loss:     0.0016 Validation Accuracy: 0.586800
Epoch 36, CIFAR-10 Batch 1:  Loss:     0.0246 Validation Accuracy: 0.566800
Epoch 37, CIFAR-10 Batch 1:  Loss:     0.0056 Validation Accuracy: 0.588200
Epoch 38, CIFAR-10 Batch 1:  Loss:     0.0013 Validation Accuracy: 0.577800
Epoch 39, CIFAR-10 Batch 1:  Loss:     0.0051 Validation Accuracy: 0.568600
Epoch 40, CIFAR-10 Batch 1:  Loss:     0.0013 Validation Accuracy: 0.574600
Epoch 41, CIFAR-10 Batch 1:  Loss:     0.0016 Validation Accuracy: 0.580200
Epoch 42, CIFAR-10 Batch 1:  Loss:     0.0026 Validation Accuracy: 0.586200
Epoch 43, CIFAR-10 Batch 1:  Loss:     0.0084 Validation Accuracy: 0.573400
Epoch 44, CIFAR-10 Batch 1:  Loss:     0.0036 Validation Accuracy: 0.567000
Epoch 45, CIFAR-10 Batch 1:  Loss:     0.0032 Validation Accuracy: 0.561200
```

```
Epoch 46, CIFAR-10 Batch 1:  Loss:     0.0005 Validation Accuracy: 0.594200
Epoch 47, CIFAR-10 Batch 1:  Loss:     0.0008 Validation Accuracy: 0.601600
Epoch 48, CIFAR-10 Batch 1:  Loss:     0.0019 Validation Accuracy: 0.584800
Epoch 49, CIFAR-10 Batch 1:  Loss:     0.0006 Validation Accuracy: 0.591400
Epoch 50, CIFAR-10 Batch 1:  Loss:     0.0006 Validation Accuracy: 0.593000
Epoch 51, CIFAR-10 Batch 1:  Loss:     0.0006 Validation Accuracy: 0.588200
Epoch 52, CIFAR-10 Batch 1:  Loss:     0.0014 Validation Accuracy: 0.603800
Epoch 53, CIFAR-10 Batch 1:  Loss:     0.0005 Validation Accuracy: 0.595000
Epoch 54, CIFAR-10 Batch 1:  Loss:     0.0004 Validation Accuracy: 0.593000
Epoch 55, CIFAR-10 Batch 1:  Loss:     0.0023 Validation Accuracy: 0.594000
Epoch 56, CIFAR-10 Batch 1:  Loss:     0.0004 Validation Accuracy: 0.610400
Epoch 57, CIFAR-10 Batch 1:  Loss:     0.0001 Validation Accuracy: 0.609400
Epoch 58, CIFAR-10 Batch 1:  Loss:     0.0001 Validation Accuracy: 0.617200
Epoch 59, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.621400
Epoch 60, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.623400
Epoch 61, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.626200
Epoch 62, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.625600
Epoch 63, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.625600
Epoch 64, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.625800
Epoch 65, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.625800
Epoch 66, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.626400
Epoch 67, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.626600
Epoch 68, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.626800
Epoch 69, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.626600
Epoch 70, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.626800
Epoch 71, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.626400
Epoch 72, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.626200
Epoch 73, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.626400
Epoch 74, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627000
Epoch 75, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.626800
Epoch 76, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.626600
Epoch 77, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627000
Epoch 78, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627400
Epoch 79, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627600
Epoch 80, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627600
Epoch 81, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.628400
Epoch 82, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.628600
Epoch 83, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.628200
Epoch 84, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.628200
Epoch 85, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.628200
Epoch 86, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627800
Epoch 87, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627600
Epoch 88, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627600
Epoch 89, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627400
Epoch 90, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627600
Epoch 91, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627400
Epoch 92, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627600
Epoch 93, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.628400
Epoch 94, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.628400
Epoch 95, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.628400
Epoch 96, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.628000
Epoch 97, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627600
Epoch 98, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627800
Epoch 99, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627800
Epoch 100, CIFAR-10 Batch 1:  Loss:     0.0000 Validation Accuracy: 0.627400
```

## Fully Train the Model

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

In [17]:

```
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
                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)

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

```
Training...
Epoch  1, CIFAR-10 Batch 1:  Loss:     2.0032 Validation Accuracy: 0.35500
0
Epoch  1, CIFAR-10 Batch 2:  Loss:     1.5729 Validation Accuracy: 0.35400
0
Epoch  1, CIFAR-10 Batch 3:  Loss:     1.2236 Validation Accuracy: 0.47980
0
Epoch  1, CIFAR-10 Batch 4:  Loss:     1.2938 Validation Accuracy: 0.45540
0
Epoch  1, CIFAR-10 Batch 5:  Loss:     1.2348 Validation Accuracy: 0.54200
0
Epoch  2, CIFAR-10 Batch 1:  Loss:     1.2964 Validation Accuracy: 0.54980
0
Epoch  2, CIFAR-10 Batch 2:  Loss:     1.0003 Validation Accuracy: 0.58680
0
Epoch  2, CIFAR-10 Batch 3:  Loss:     0.8154 Validation Accuracy: 0.57380
0
Epoch  2, CIFAR-10 Batch 4:  Loss:     0.8793 Validation Accuracy: 0.59640
0
```

The model has been saved to disk.

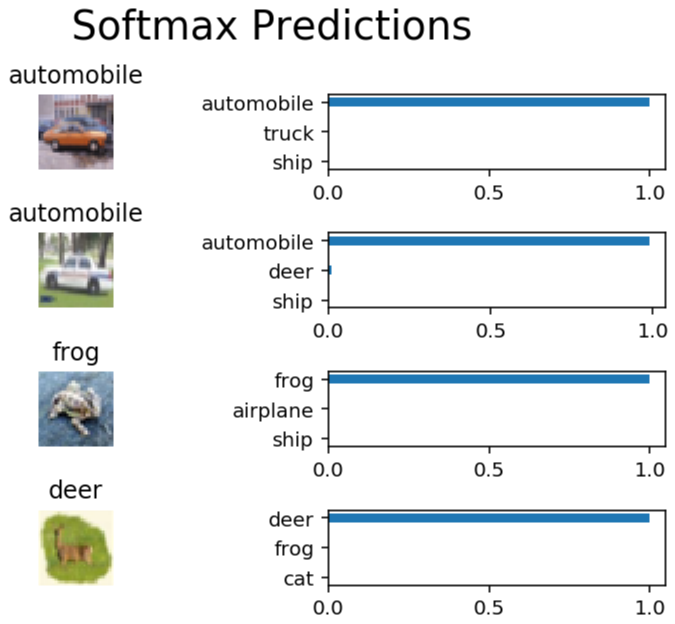
## Test Model

Test the model against the test dataset. This will be the final accuracy.

In [18]:

```
%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_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_featu
            test_batch_acc_total += sess.run(
                loaded_acc,
                feed_dict={loaded_x: test_feature_batch, loaded_y: test_label_batch, loadec
            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_f
        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, loadec
        helper.display_image_predictions(random_test_features, random_test_labels, random_t


test_model()
```

```
Testing Accuracy: 0.6978515625
```


Softmax Predictions
automobile
automobile
truck
ship
0.0
0.5
1.0
automobile
automobile
deer
ship
0.0
0.5
1.0
frog
frog
airplane
ship
0.0
0.5
1.0
deer
deer
frog
cat
0.0
0.5
1.0