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 [ ]:

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

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

tests.test_folder_path(cifar10_dataset_folder_path)
```

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 [ ]:

```
%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)
```

In [ ]:

```
batch_id = 2
sample_id = 5
helper.display_stats(cifar10_dataset_folder_path, batch_id, sample_id)
```

In [ ]:

```
batch_id = 3
sample_id = 5
helper.display_stats(cifar10_dataset_folder_path, batch_id, sample_id)
```

In [ ]:

```
batch_id = 4
sample_id = 5
helper.display_stats(cifar10_dataset_folder_path, batch_id, sample_id)
```

In [ ]:

```
batch_id = 5
sample_id = 5
helper.display_stats(cifar10_dataset_folder_path, batch_id, sample_id)
```

Implement Preprocess Functions

Normalize

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

In [ ]:

```
def normalize(x):
    """
    Normalize a list of sample image data in the range of 0 to 1
    : x: List of image data.  The image shape is (32, 32, 3)
    : return: Numpy array of normalize data
    """
    # TODO: Implement Function
#     a = 0
#     b = 1
#     grayscale_min = 0
#     grayscale_max =255
#     return a + ( ( (x - grayscale_min)*(b - a) )/( grayscale_max - 
grayscale_min ) )
    return x / 255
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_normalize(normalize)
```

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 [ ]:

```
import numpy as np

def one_hot_encode(x):
    """
    One hot encode a list of sample labels. Return a one-hot encoded vector for 
each label.
    Adapted from http://stackoverflow.com/questions/29831489/numpy-1-hot-array

    : x: List of sample Labels
    : return: Numpy array of one-hot encoded labels
    """
    # TODO: Implement Function
    output = np.zeros((len(x), max(x)+1))
    output[np.arange(len(x)), x] = 1
    return output
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_one_hot_encode(one_hot_encode)
```

In [ ]:

```
one_hot_encode([0, 1, 8, 5, 1, 5, 7, 4, 9])
```

Randomize Data

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

Preprocess all the data and save it

Running the code cell below will preprocess all the CIFAR-10 data and save it to file. The code below also uses 10% of the training data for validation.

In [ ]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
# Preprocess Training, Validation, and Testing Data
helper.preprocess_and_save_data(cifar10_dataset_folder_path, normalize,
one_hot_encode)
```

Check Point

This is your first checkpoint. If you ever decide to come back to this notebook or have to restart the notebook, you can start from here. The preprocessed data has been saved to disk.

In [1]:

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

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

Build the network

For the neural network, you'll build each layer into a function. Most of the code you've seen has been outside of functions. To test your code more thoroughly, we require that you put each layer in a function. This allows us to give you better feedback and test for simple mistakes using our unittests before you submit your project.

If you're finding it hard to dedicate enough time for this course a 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 TensorFlow Layers or TensorFlow Layers (contrib) to build each layer, except "Convolutional & Max Pooling" layer. TF Layers is similar to Keras's and TFLearn's abstraction to layers, so it's easy to pickup.

If you would like to get the most of this course, try to solve all the problems without TF Layers. Let's begin!

Input

The neural network needs to read the image data, one-hot encoded labels, and dropout keep probability. Implement the following functions

- Implement `neural_net_image_input`
  - Return a TF Placeholder
  - Set the shape using `image_shape` with batch size set to `None`.
  - Name the TensorFlow placeholder "x" using the TensorFlow `name` parameter in the TF Placeholder.
- Implement `neural_net_label_input`
  - Return a TF Placeholder
  - Set the shape using `n_classes` with batch size set to `None`.

    - Name the TensorFlow placeholder "y" using the TensorFlow `name` parameter in the TF Placeholder.
- Implement `neural_net_keep_prob_input`
    - Return a TF Placeholder for dropout keep probability.
    - Name the TensorFlow placeholder "keep_prob" using the TensorFlow `name` parameter in the TF Placeholder.

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

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

In [2]:

```
import tensorflow as tf

def neural_net_image_input(image_shape):
    """
    Return a Tensor for a bach of image input
    : image_shape: Shape of the images
    : return: Tensor for image input.
    """
    # TODO: Implement Function
    dimensions = [None,] # First parameter is None for dynamic values
    for dimension in image_shape:
        dimensions.append(dimension)
    x = tf.placeholder(tf.float32, dimensions, name='x')
    return x

def neural_net_label_input(n_classes):
    """
    Return a Tensor for a batch of label input
    : n_classes: Number of classes
    : return: Tensor for label input.
    """
    # TODO: Implement Function
    y = tf.placeholder(tf.float32, [None, n_classes], name="y")
    return y

def neural_net_keep_prob_input():
    """
    Return a Tensor for keep probability
    : return: Tensor for keep probability.
    """
    # TODO: Implement Function
    return tf.placeholder(tf.float32, name="keep_prob")
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tf.reset_default_graph()
tests.test_nn_image_inputs(neural_net_image_input)
tests.test_nn_label_inputs(neural_net_label_input)
tests.test_nn_keep_prob_inputs(neural_net_keep_prob_input)
```

```
Image Input Tests Passed.
```

```
Label Input Tests Passed.
Keep Prob Tests Passed.
```

Convolution and Max Pooling Layer

Convolution layers have a lot of success with images. For this code cell, you should implement the function `conv2d_maxpool` to apply convolution then max pooling:

- Create the weight and bias using `conv_ksize`, `conv_num_outputs` and the shape of `x_tensor`.
- 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. You're free to use any TensorFlow package for all the other layers.

In [3]:

```python
def conv2d_maxpool(x_tensor, conv_num_outputs, conv_ksize, conv_strides,
pool_ksize, pool_strides):
    """
    Apply convolution then max pooling to x_tensor
    :param x_tensor: TensorFlow Tensor
    :param conv_num_outputs: Number of outputs for the convolutional layer
    :param conv_strides: Stride 2-D Tuple for convolution
    :param pool_ksize: kernal size 2-D Tuple for pool
    :param pool_strides: Stride 2-D Tuple for pool
    : return: A tensor that represents convolution and max pooling of x_tensor
    """
    # TODO: Implement Function
    x = 0
    y = 1
    z = 3
    weight = tf.Variable(tf.truncated_normal([conv_ksize[x],
                                          conv_ksize[y],
                                          int(x_tensor.shape[z]),
                                          conv_num_outputs],
                                         stddev=0.05))

    bias = tf.Variable(tf.constant(0.05, shape=[conv_num_outputs]))

    x_tensor = tf.nn.conv2d(x_tensor,
                            weight,
                            strides=[1,
                                     conv_strides[x],
                                     conv_strides[y],
                                     1],
                            padding='SAME')

    x_tensor = tf.nn.bias_add(x_tensor, bias)
    x_tensor = tf.nn.relu(x_tensor)

    x_tensor = tf.nn.max_pool(x_tensor,
                              ksize=[1,
                                     pool_ksize[x],
                                     pool_ksize[y],
                                     1],
                              strides=[1,
                                       pool_strides[x],
                                       pool_strides[y],
                                       1],
                              padding='SAME')
    return x_tensor
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_con_pool(conv2d_maxpool)
```

```
Tests Passed
```

Flatten Layer

Implement the `flatten` function to change the dimension of `x_tensor` from a 4-D tensor to a 2-D tensor. The output should be the shape (*Batch Size*, *Flattened Image Size*). You can use TensorFlow Layers or TensorFlow Layers (contrib) for this layer.

In [4]:

```
def flatten(x_tensor):
    """
    Flatten x_tensor to (Batch Size, Flattened Image Size)
    : x_tensor: A tensor of size (Batch Size, ...), where ... are the image
dimensions.
    : return: A tensor of size (Batch Size, Flattened Image Size).
    """
    # TODO: Implement Function
    return tf.reshape(x_tensor, [-1, int(x_tensor.shape[1] * x_tensor.shape[2]
* x_tensor.shape[3])])


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

```
Tests Passed
```

Fully-Connected Layer

Implement the `fully_conn` function to apply a fully connected layer to `x_tensor` with the shape (*Batch Size*, *num_outputs*). You can use TensorFlow Layers or TensorFlow Layers (contrib) for this layer.

In [5]:

```
def fully_conn(x_tensor, num_outputs):
    """
    Apply a fully connected layer to x_tensor using weight and bias
    : x_tensor: A 2-D tensor where the first dimension is batch size.
    : num_outputs: The number of output that the new tensor should be.
    : return: A 2-D tensor where the second dimension is num_outputs.
    """
    # TODO: Implement Function
    weights = tf.Variable(tf.truncated_normal([int(x_tensor.shape[1]),
                                                  num_outputs],
                                                 stddev=0.05))
    bias = tf.Variable(tf.constant(0.05, shape=[num_outputs]))

    fc_layer = tf.add(tf.matmul(x_tensor, weights), bias)

    fc_layer = tf.nn.relu(fc_layer)

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

```
Tests Passed
```

Output Layer

Implement the `output` function to apply a fully connected layer to `x_tensor` with the shape (*Batch Size*, *num_outputs*). You can use TensorFlow Layers or TensorFlow Layers (contrib) for this layer.

Note: Activation, softmax, or cross entropy shouldn't be applied to this.

In [6]:

```
def output(x_tensor, num_outputs):
    """
    Apply a output layer to x_tensor using weight and bias
    : x_tensor: A 2-D tensor where the first dimension is batch size.
    : num_outputs: The number of output that the new tensor should be.
    : return: A 2-D tensor where the second dimension is num_outputs.
    """
    # TODO: Implement Function
    weights = tf.Variable(tf.truncated_normal([int(x_tensor.shape[1]),
                                                  num_outputs],
                                                 stddev=0.05))
     bias = tf.Variable(tf.constant(0.05, shape=[num_outputs]))

    output_layer = tf.add(tf.matmul(x_tensor, weights), bias)

    output_layer = tf.nn.softmax(output_layer)

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

```
Tests Passed
```

Create Convolutional Model

Implement the function `conv_net` to create a convolutional neural network model. The function takes in a batch of images, `x`, and outputs logits. Use the layers you created above to create this model:

- Apply 1, 2, or 3 Convolution and Max Pool layers
- Apply a Flatten Layer
- Apply 1, 2, or 3 Fully Connected Layers
- Apply an Output Layer
- Return the output
- Apply TensorFlow's Dropout to one or more layers in the model using `keep_prob`.

In [7]:

```
def conv_net(x, keep_prob):
    """
    Create a convolutional neural network model
    : x: Placeholder tensor that holds image data.
    : keep_prob: Placeholder tensor that hold dropout keep probability.
    : return: Tensor that represents logits
    """
    # TODO: Apply 1, 2, or 3 Convolution and Max Pool layers
    #    Play around with different number of outputs, kernel size and stride
    # Function Definition from Above:
    #    conv2d_maxpool(x_tensor, conv_num_outputs, conv_ksize, conv_strides,
pool_ksize, pool_strides)
    conv1 = conv2d_maxpool(x_tensor=x,
                           conv_num_outputs=40,
                           conv_ksize=(5, 5),
                           conv_strides=(1, 1),
                           pool_ksize=(2, 2),
                           pool_strides=(1, 1))
    conv2 = conv2d_maxpool(x_tensor=conv1,
                           conv_num_outputs=30,
                           conv_ksize=(4, 4),
                           conv_strides=(1, 1),
                           pool_ksize=(2, 2),
                           pool_strides=(1, 1))


    # TODO: Apply a Flatten Layer
    # Function Definition from Above:
    #   flatten(x_tensor)
    flat_layer = flatten(conv2)


    # TODO: Apply 1, 2, or 3 Fully Connected Layers
    #    Play around with different number of outputs
    # Function Definition from Above:
    #   fully_conn(x_tensor, num_outputs)
    fc1 = fully_conn(flat_layer, 15)
    fc1 = tf.nn.dropout(fc1, keep_prob)

    # TODO: Apply an Output Layer
    #    Set this to the number of classes
    # Function Definition from Above:
    output_layer = output(fc1, 10)


    # TODO: return output
    return output_layer

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

##############################
## Build the Neural Network ##
##############################

# Remove previous weights, bias, inputs, etc..
tf.reset_default_graph()

# Inputs
x = neural_net_image_input((32, 32, 3))
y = neural_net_label_input(10)
keep_prob = neural_net_keep_prob_input()

# Model
logits = conv_net(x, keep_prob)

# Name logits Tensor, so that is can be loaded from disk after training
logits = tf.identity(logits, name='logits')

# Loss and Optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, 
labels=y))
optimizer = tf.train.AdamOptimizer().minimize(cost)

# Accuracy
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32), name='accuracy')

tests.test_conv_net(conv_net)
```

```
Neural Network Built!
```

Train the Neural Network

Single Optimization

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

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

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

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

In [8]:

```
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
    """
    # TODO: Implement Function
    session.run(optimizer,
                feed_dict={x: feature_batch,
                           y: label_batch,
                           keep_prob: keep_probability})
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_train_nn(train_neural_network)
```

```
Tests Passed
```

Show Stats

Implement the function `print_stats` to print loss and validation accuracy. Use the global variables `valid_features` and `valid_labels` to calculate validation accuracy. Use a keep probability of `1.0` to calculate the loss and validation accuracy.

In [9]:

```
def print_stats(session, feature_batch, label_batch, cost, accuracy):
    """
    Print information about loss and validation accuracy
    : session: Current TensorFlow session
    : feature_batch: Batch of Numpy image data
    : label_batch: Batch of Numpy label data
    : cost: TensorFlow cost function
    : accuracy: TensorFlow accuracy function
    """
    # TODO: Implement Function
    loss = session.run(cost,
                       feed_dict={x: feature_batch,
                                  y: label_batch,
                                  keep_prob: 1.0})

    valid_acc = session.run(accuracy,
                            feed_dict={x: valid_features,
                                       y: valid_labels,
                                       keep_prob: 1.0})

    print('Loss: {:>15.4f} Validation Accuracy: {:.4f}'.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 [10]:

```
# TODO: Tune Parameters
epochs = 40
batch_size = 128
keep_probability = 0.7
```

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 [11]:

```
"""
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_i, batch_size):
            train_neural_network(sess, optimizer, keep_probability, 
batch_features, batch_labels)
        print('Epoch {:>2}, CIFAR-10 Batch {}:  '.format(epoch + 1, batch_i), 
end='')
        print_stats(sess, batch_features, batch_labels, cost, accuracy)
```

```
Checking the Training on a Single Batch...
Epoch  1, CIFAR-10 Batch 1:  Loss:          2.2526 Validation Accuracy: 0.2514
```

…….

```
Epoch 40, CIFAR-10 Batch 1:  Loss:          1.7907 Validation Accuracy: 0.4728
```

Fully Train the Model

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

In [12]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
save_model_path = './image_classification'

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

    # Training cycle
    for epoch in range(epochs):
        # Loop over all batches
        n_batches = 5
        for batch_i in range(1, n_batches + 1):
            for batch_features, batch_labels in
helper.load_preprocess_training_batch(batch_i, batch_size):
                train_neural_network(sess, optimizer, keep_probability,
batch_features, batch_labels)
            print('Epoch {:>2}, CIFAR-10 Batch {}:  '.format(epoch + 1,
batch_i), end='')
            print_stats(sess, batch_features, batch_labels, cost, accuracy)

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

```
Training...
Epoch  1, CIFAR-10 Batch 1:  Loss:          2.2311 Validation Accuracy: 0.2568
```

……

Checkpoint

The model has been saved to disk.

Test Model

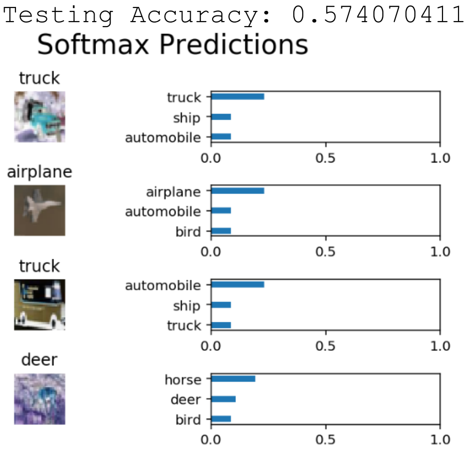
Test your model against the test dataset. This will be your final accuracy. You should have an accuracy greater than 50%. If you don't, keep tweaking the model architecture and parameters.

In [13]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
```

```python
import tensorflow as tf
import pickle
import helper
import random

# Set batch size if not already set
try:
    if batch_size:
        pass
except NameError:
    batch_size = 64

save_model_path = './image_classification'
n_samples = 4
top_n_predictions = 3

def test_model():
    """
    Test the saved model against the test dataset
    """

    test_features, test_labels = pickle.load(open('preprocess_training.p',
mode='rb'))
    loaded_graph = tf.Graph()

    with tf.Session(graph=loaded_graph) as sess:
        # Load model
        loader = tf.train.import_meta_graph(save_model_path + '.meta')
        loader.restore(sess, save_model_path)

        # Get Tensors from loaded model
        loaded_x = loaded_graph.get_tensor_by_name('x:0')
        loaded_y = loaded_graph.get_tensor_by_name('y:0')
        loaded_keep_prob = loaded_graph.get_tensor_by_name('keep_prob:0')
        loaded_logits = loaded_graph.get_tensor_by_name('logits:0')
        loaded_acc = loaded_graph.get_tensor_by_name('accuracy:0')

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

        for train_feature_batch, train_label_batch in
helper.batch_features_labels(test_features, test_labels, batch_size):
            test_batch_acc_total += sess.run(
                loaded_acc,
                feed_dict={loaded_x: train_feature_batch, loaded_y:
train_label_batch, loaded_keep_prob: 1.0})
            test_batch_count += 1
        print('Testing Accuracy:
{}\n'.format(test_batch_acc_total/test_batch_count))

        # Print Random Samples
        random_test_features, random_test_labels =
tuple(zip(*random.sample(list(zip(test_features, test_labels)), n_samples)))
        random_test_predictions = sess.run(
            tf.nn.top_k(tf.nn.softmax(loaded_logits), top_n_predictions),
            feed_dict={loaded_x: random_test_features, loaded_y:
random_test_labels, loaded_keep_prob: 1.0})
        helper.display_image_predictions(random_test_features,
random_test_labels, random_test_predictions)


test_model()
```

```
Testing Accuracy: 0.5740704113924051
```



Why 50-70% 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 70%. 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.

Helper.py

```
import pickle
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelBinarizer

def _load_label_names():
    """
    Load the label names from file
    """
    return ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse', 'ship', 'truck']
def load_cfar10_batch(cifar10_dataset_folder_path, batch_id):
    """
    Load a batch of the dataset
    """
    with open(cifar10_dataset_folder_path + '/data_batch_' + str(batch_id),
mode='rb') as file:
        batch = pickle.load(file, encoding='latin1')
    features = batch['data'].reshape((len(batch['data']), 3, 32,
32)).transpose(0, 2, 3, 1)
    labels = batch['labels']

    return features, labels


def display_stats(cifar10_dataset_folder_path, batch_id, sample_id):
    """
    Display Stats of the the dataset
    """
    batch_ids = list(range(1, 6))

    if batch_id not in batch_ids:
        print('Batch Id out of Range. Possible Batch Ids:
{}'.format(batch_ids))
        return None

    features, labels = load_cfar10_batch(cifar10_dataset_folder_path, batch_id)

    if not (0 <= sample_id < len(features)):
        print('{} samples in batch {}.  {} is out of
range.'.format(len(features), batch_id, sample_id))
        return None

    print('\nStats of batch {}:'.format(batch_id))
    print('Samples: {}'.format(len(features)))
    print('Label Counts: {}'.format(dict(zip(*np.unique(labels,
return_counts=True)))))
    print('First 20 Labels: {}'.format(labels[:20]))

    sample_image = features[sample_id]
    sample_label = labels[sample_id]
    label_names = _load_label_names()

    print('\nExample of Image {}:'.format(sample_id))
    print('Image - Min Value: {} Max Value: {}'.format(sample_image.min(),
sample_image.max()))
    print('Image - Shape: {}'.format(sample_image.shape))
    print('Label - Label Id: {} Name: {}'.format(sample_label,
label_names[sample_label]))
    plt.axis('off')
    plt.imshow(sample_image)


def _preprocess_and_save(normalize, one_hot_encode, features, labels,
filename):
    """
    Preprocess data and save it to file
    """
    features = normalize(features)
    labels = one_hot_encode(labels)

    pickle.dump((features, labels), open(filename, 'wb'))


def preprocess_and_save_data(cifar10_dataset_folder_path, normalize,
one_hot_encode):

    """
    Preprocess Training and Validation Data
    """
    n_batches = 5
    valid_features = []
    valid_labels = []

    for batch_i in range(1, n_batches + 1):
        features, labels = load_cfar10_batch(cifar10_dataset_folder_path, 
batch_i)
        validation_count = int(len(features) * 0.1)

        # Prprocess and save a batch of training data
        _preprocess_and_save(
            normalize,
            one_hot_encode,
            features[:-validation_count],
            labels[:-validation_count],
            'preprocess_batch_' + str(batch_i) + '.p')

        # Use a portion of training batch for validation
        valid_features.extend(features[-validation_count:])
        valid_labels.extend(labels[-validation_count:])

    # Preprocess and Save all validation data
    _preprocess_and_save(
        normalize,
        one_hot_encode,
        np.array(valid_features),
        np.array(valid_labels),
        'preprocess_validation.p')

    with open(cifar10_dataset_folder_path + '/test_batch', mode='rb') as file:
        batch = pickle.load(file, encoding='latin1')

    # load the training data
    test_features = batch['data'].reshape((len(batch['data']), 3, 32, 
32)).transpose(0, 2, 3, 1)
    test_labels = batch['labels']

    # Preprocess and Save all training data
    _preprocess_and_save(
        normalize,
        one_hot_encode,
        np.array(test_features),
        np.array(test_labels),
        'preprocess_training.p')


def batch_features_labels(features, labels, batch_size):
    """
    Split features and labels into batches
    """
    for start in range(0, len(features), batch_size):
        end = min(start + batch_size, len(features))
        yield features[start:end], labels[start:end]



def load_preprocess_training_batch(batch_id, batch_size):
    """
    Load the Preprocessed Training data and return them in batches of 
<batch_size> or less
    """
    filename = 'preprocess_batch_' + str(batch_id) + '.p'
    features, labels = pickle.load(open(filename, mode='rb'))

    # Return the training data in batches of size <batch_size> or less
    return batch_features_labels(features, labels, batch_size)
```

```
def display_image_predictions(features, labels, predictions):
    n_classes = 10
    label_names = _load_label_names()
    label_binarizer = LabelBinarizer()
    label_binarizer.fit(range(n_classes))
    label_ids = label_binarizer.inverse_transform(np.array(labels))

    fig, axies = plt.subplots(nrows=4, ncols=2)
    fig.tight_layout()
    fig.suptitle('Softmax Predictions', fontsize=20, y=1.1)

    n_predictions = 3
    margin = 0.05
    ind = np.arange(n_predictions)
    width = (1. - 2. * margin) / n_predictions

    for image_i, (feature, label_id, pred_indicies, pred_values) in
enumerate(zip(features, label_ids, predictions.indices, predictions.values)):
        pred_names = [label_names[pred_i] for pred_i in pred_indicies]
        correct_name = label_names[label_id]

        axies[image_i][0].imshow(feature*255)
        axies[image_i][0].set_title(correct_name)
        axies[image_i][0].set_axis_off()

        axies[image_i][1].barh(ind + margin, pred_values[::-1], width)
        axies[image_i][1].set_yticks(ind + margin)
        axies[image_i][1].set_yticklabels(pred_names[::-1])
        axies[image_i][1].set_xticks([0, 0.5, 1.0])
```