Self-Driving Car Engineer Nanodegree

Deep Learning

Project: Build a Traffic Sign Recognition Classifier

In this notebook, a template is provided for you to implement your functionality in stages, which is required to successfully complete this project. If additional code is required that cannot be included in the notebook, be sure that the Python code is successfully imported and included in your submission if necessary.

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the iPython Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to \n", "**File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there is a writeup to complete. The writeup should be completed in a separate file, which can be either a markdown file or a pdf document. There is a write up template (https://github.com/udacity/CarND-Traffic-Sign-Classifier-Project/blob/master/writeup_template.md) that can be used to guide the writing process. Completing the code template and writeup template will cover all of the rubric points (https://review.udacity.com/#!/rubrics/481/view) for this project.

The rubric (https://review.udacity.com/#!/rubrics/481/view) contains "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. The stand out suggestions are optional. If you decide to pursue the "stand out suggestions", you can include the code in this Ipython notebook and also discuss the results in the writeup file.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. In addition, Markdown cells can be edited by typically double-clicking the cell to enter edit mode.

Step 0: Load The Data

In [1]:

```
# Load pickled data
import pickle

# TODO: Fill this in based on where you saved the training and testing data

training_file = 'traffic-signs-data/train.p'
validation_file = 'traffic-signs-data/valid.p'
testing_file = 'traffic-signs-data/test.p'

with open(training_file, mode='rb') as f:
    train = pickle.load(f)
with open(validation_file, mode='rb') as f:
    valid = pickle.load(f)
with open(testing_file, mode='rb') as f:
    test = pickle.load(f)

X_train, y_train = train['features'], train['labels']
X_valid, y_valid = valid['features'], valid['labels']
X_test, y_test = test['features'], test['labels']
```

Step 1: Dataset Summary & Exploration

The pickled data is a dictionary with 4 key/value pairs:

- `'features'` is a 4D array containing raw pixel data of the traffic sign images, (num examples, width, height, channels).
- `'labels'` is a 1D array containing the label/class id of the traffic sign. The file `signnames.csv` contains id -> name mappings for each id.
- `'sizes'` is a list containing tuples, (width, height) representing the original width and height the image.
- `'coords'` is a list containing tuples, (x1, y1, x2, y2) representing coordinates of a bounding box around the sign in the image. **THESE COORDINATES ASSUME THE ORIGINAL IMAGE. THE PICKLED DATA CONTAINS RESIZED VERSIONS (32 by 32) OF THESE IMAGES**

Complete the basic data summary below. Use python, numpy and/or pandas methods to calculate the data summary rather than hard coding the results. For example, the [pandas shape method (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.shape.html)](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.shape.html) might be useful for calculating some of the summary results.

Provide a Basic Summary of the Data Set Using Python, Numpy and/or Pandas

In [2]:

```
### Replace each question mark with the appropriate value.
### Use python, pandas or numpy methods rather than hard coding the results

# TODO: Number of training examples
n_train = len(X_train)

# TODO: Number of validation examples
n_validation = len(X_valid)

# TODO: Number of testing examples.
n_test = len(X_test)

# TODO: What's the shape of an traffic sign image?
image_shape = X_train[0].shape

# TODO: How many unique classes/labels there are in the dataset.
n_classes = len(set(train['labels']))

print("Number of training examples =", n_train)
print("Number of validation examples =", n_validation)
print("Number of testing examples =", n_test)
print("Image data shape =", image_shape)
print("Number of classes =", n_classes)
```

```
Number of training examples = 34799
Number of validation examples = 4410
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43
```

Include an exploratory visualization of the dataset

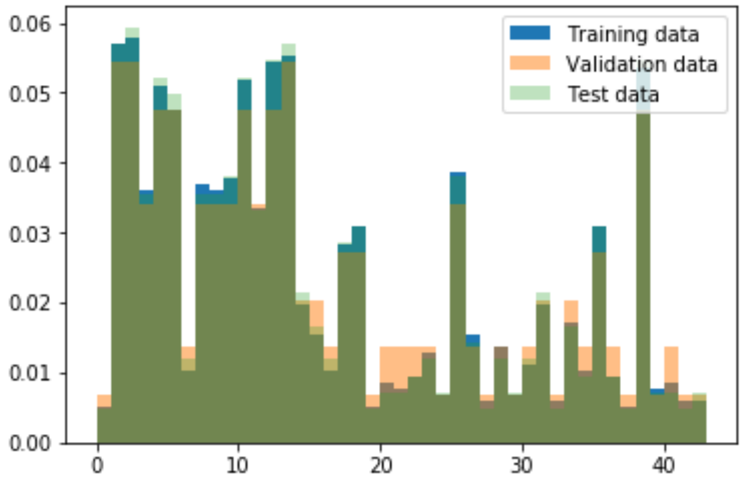
Visualize the German Traffic Signs Dataset using the pickled file(s). This is open ended, suggestions include: plotting traffic sign images, plotting the count of each sign, etc.

The Matplotlib (http://matplotlib.org/) examples (http://matplotlib.org/examples/index.html) and gallery (http://matplotlib.org/gallery.html) pages are a great resource for doing visualizations in Python.

NOTE: It's recommended you start with something simple first. If you wish to do more, come back to it after you've completed the rest of the sections. It can be interesting to look at the distribution of classes in the training, validation and test set. Is the distribution the same? Are there more examples of some classes than others?

In [3]:

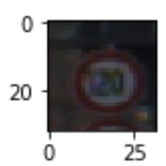
```
import matplotlib.pyplot as plt
histogram_train = plt.hist(y_train, range(0, n_classes + 1), normed = True, labe
l = "Training data", alpha = 1.0)
histogram_validation = plt.hist(y_valid, range(0, n_classes + 1), normed = True,
 label = "Validation data", alpha = 0.5)
histogram_test = plt.hist(y_test, range(0, n_classes + 1), normed = True, label
= "Test data", alpha = 0.3)
plt.legend()
plt.show()
```



In [4]:

```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import random
import numpy as np
import matplotlib.pyplot as plt
import csv
%matplotlib inline

index = random.randint(0, len(X_train))
image = X_train[index].squeeze()

#Load the sign names
with open('signnames.csv', mode='r') as infile:
    reader = csv.reader(infile)
    sign_names = {rows[0]:rows[1] for rows in reader}

plt.figure(figsize=(1,1))
plt.imshow(image, cmap="gray")
print("{:d} - {:s}".format(y_train[index], sign_names[str(y_train[index])] ))
```

```
8 - Speed limit (120km/h)
```



Step 2: Design and Test a Model Architecture

Design and implement a deep learning model that learns to recognize traffic signs. Train and test your model on the German Traffic Sign Dataset (http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset).

The LeNet-5 implementation shown in the classroom (https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/6df7ae49-c61c-4bb2-a23e-6527e69209ec/lessons/601ae704-1035-4287-8b11-e2c2716217ad/concepts/d4aca031-508f-4e0b-b493-e7b706120f81) at the end of the CNN lesson is a solid starting point. You'll have to change the number of classes and possibly the preprocessing, but aside from that it's plug and play!

With the LeNet-5 solution from the lecture, you should expect a validation set accuracy of about 0.89. To meet specifications, the validation set accuracy will need to be at least 0.93. It is possible to get an even higher accuracy, but 0.93 is the minimum for a successful project submission.

There are various aspects to consider when thinking about this problem:

- Neural network architecture (is the network over or underfitting?)
- Play around preprocessing techniques (normalization, rgb to grayscale, etc)
- Number of examples per label (some have more than others).
- Generate fake data.

Here is an example of a published baseline model on this problem (http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf). It's not required to be familiar with the approach used in the paper but, it's good practice to try to read papers like these.

Pre-process the Data Set (normalization, grayscale, etc.)

Minimally, the image data should be normalized so that the data has mean zero and equal variance. For image data, `(pixel - 128)/ 128` is a quick way to approximately normalize the data and can be used in this project.

Other pre-processing steps are optional. You can try different techniques to see if it improves performance.

Use the code cell (or multiple code cells, if necessary) to implement the first step of your project.

In [5]:

```
### Preprocess the data here. It is required to normalize the data. Other prepro
cessing steps could include
### converting to grayscale, etc.
### Feel free to use as many code cells as needed.
```

In [6]:

```
# Shuffle the training data
from sklearn.utils import shuffle

X_train, y_train = shuffle(X_train, y_train)
```

In [7]:

```
# Normalize data
X_train = (X_train/127.5) - 1.0
X_valid = (X_valid/127.5) - 1.0
X_test = (X_test/127.5) - 1.0
```

Model Architecture

In [8]:

```
### Define your architecture here.
### Feel free to use as many code cells as needed.
```

In [9]:

```
import tensorflow as tf

EPOCHS = 30
BATCH_SIZE = 128
```

In [10]:

```
from tensorflow.contrib.layers import flatten

def LeNet(x, dropout):
    # Arguments used for tf.truncated_normal, randomly defines variables for the
 weights and biases for each layer
    mu = 0
    sigma = 0.1

    weights = {
        'wc1': tf.Variable(tf.truncated_normal([5, 5, 3, 6], mean = mu, stddev =
 sigma)),
        'wc2': tf.Variable(tf.truncated_normal([5, 5, 6, 16], mean = mu, stddev
= sigma)),
        'wd1': tf.Variable(tf.truncated_normal([5*5*16, 1200], mean = mu, stddev
 = sigma)),
        'wd2': tf.Variable(tf.truncated_normal([1200, 84], mean = mu, stddev = s
igma)),
        'out': tf.Variable(tf.truncated_normal([84, n_classes], mean = mu, stdde
v = sigma))}

    biases = {
        'bc1': tf.Variable(tf.truncated_normal([6], mean = mu, stddev = sigma)),
        'bc2': tf.Variable(tf.truncated_normal([16], mean = mu, stddev =
sigma)),
        'bd1': tf.Variable(tf.truncated_normal([1200], mean = mu, stddev =
sigma)),
        'bd2': tf.Variable(tf.truncated_normal([84], mean = mu, stddev =
sigma)),
        'out': tf.Variable(tf.truncated_normal([n_classes], mean = mu, stddev =
sigma))}

    # Layer 1: Convolutional. Input = 32x32x3. Output = 28x28x6.
    conv_1 = tf.nn.conv2d(x, weights['wc1'], strides=[1, 1, 1, 1], padding='VALI
D')
    conv_1 = tf.nn.bias_add(conv_1, biases['bc1'])
```

```
    # Activation.
    conv_1 = tf.nn.relu(conv_1)

    # Pooling. Input = 28x28x6. Output = 14x14x6.
    conv_1 = tf.nn.max_pool(conv_1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], pa
dding='SAME')

    # Layer 2: Convolutional. Output = 10x10x16.
    conv_2 = tf.nn.conv2d(conv_1, weights['wc2'], strides=[1, 1, 1, 1],
padding='VALID')
    conv_2 = tf.nn.bias_add(conv_2, biases['bc2'])

    # Activation.
    conv_2 = tf.nn.relu(conv_2)

    # Pooling. Input = 10x10x16. Output = 5x5x16.
    conv_2 = tf.nn.max_pool(conv_2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], pa
dding='SAME')

    # Flatten. Input = 5x5x16. Output = 400.
    conv_fl = tf.contrib.layers.flatten(conv_2)

    # Layer 3: Fully Connected. Input = 400. Output = 1200.
    conn_1 = tf.add(tf.matmul(conv_fl, weights['wd1']), biases['bd1'])

    # Activation.
    conn_1 = tf.nn.relu(conn_1)

    # Dropout
    conn_1 = tf.nn.dropout(conn_1, keep_prob = dropout)

    # Layer 4: Fully Connected. Input = 1200. Output = 84.
    conn_2 = tf.add(tf.matmul(conn_1, weights['wd2']), biases['bd2'])

    # Activation.
    conn_2 = tf.nn.relu(conn_2)

    # Dropout
    conn_2 = tf.nn.dropout(conn_2, keep_prob = dropout)

    # Layer 5: Fully Connected. Input = 84. Output = n_classes (43 here).
    logits = tf.add(tf.matmul(conn_2, weights['out']), biases['out'])

    return logits, conv_1, conv_2
```

Train, Validate and Test the Model

A validation set can be used to assess how well the model is performing. A low accuracy on the training and validation sets imply underfitting. A high accuracy on the training set but low accuracy on the validation set implies overfitting.

In [11]:

```
### Train your model here.
### Calculate and report the accuracy on the training and validation set.
### Once a final model architecture is selected,
### the accuracy on the test set should be calculated and reported as well.
### Feel free to use as many code cells as needed.
```

In [12]:

```
x = tf.placeholder(tf.float32, (None, 32, 32, 3))
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, n_classes)
keep_prob = tf.placeholder(tf.float32)
```

In [13]:

```
rate = 0.001

logits, conv_1, conv_2 = LeNet(x, keep_prob)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y,
logits=logits)
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)
```

In [14]:

```
correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y, 1))
accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
saver = tf.train.Saver()

def evaluate(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0
    total_loss = 0
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[offset:offse
t+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_
y, keep_prob: 1.0})
        loss = sess.run(loss_operation, feed_dict={x: batch_x, y: batch_y, keep_
prob: 1.0})
        total_accuracy += (accuracy * len(batch_x))
        total_loss += (loss * len(batch_x))
    return (total_accuracy / num_examples, total_loss / num_examples)
```

In [15]:

```
import timeit

loss_training_data = []
loss_validation_data = []

accuracy_training_data = []
accuracy_validation_data = []

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_examples = len(X_train)

    print("Training...")
    print()
    for i in range(EPOCHS):
        start_time = timeit.default_timer()
        X_train, y_train = shuffle(X_train, y_train)
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train[offset:end], y_train[offset:end]
            sess.run(training_operation, feed_dict={x: batch_x, y: batch_y, keep
_prob: 0.5})

        validation_accuracy, validation_loss = evaluate(X_valid, y_valid)
        training_accuracy, training_loss = evaluate(X_train, y_train)

        loss_training_data.append(training_loss)
        loss_validation_data.append(validation_loss)

        accuracy_training_data.append(training_accuracy)
        accuracy_validation_data.append(validation_accuracy)

        print("EPOCH {} ...".format(i+1))
        print("Training Accuracy = {:.3f}".format(training_accuracy))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print("Training Loss = {:.3f}".format(training_loss))
        print("Validation Loss = {:.3f}".format(validation_loss))
        print("Elapsed time = {:.1f} seconds".format(timeit.default_timer() - st
art_time))
        print()

    saver.save(sess, './lenet-v2')
    print("Model saved")
```

```
Training...

EPOCH 1 ...
Training Accuracy = 0.830
Validation Accuracy = 0.755
Training Loss = 0.651
Validation Loss = 0.847
Elapsed time = 66.3 seconds

EPOCH 2 ...
Training Accuracy = 0.944
Validation Accuracy = 0.869
Training Loss = 0.233
Validation Loss = 0.448
Elapsed time = 67.6 seconds

EPOCH 3 ...
Training Accuracy = 0.970
Validation Accuracy = 0.900
Training Loss = 0.126
Validation Loss = 0.343
Elapsed time = 65.5 seconds

EPOCH 4 ...
Training Accuracy = 0.983
Validation Accuracy = 0.934
Training Loss = 0.074
Validation Loss = 0.232
Elapsed time = 86.2 seconds

EPOCH 5 ...
Training Accuracy = 0.988
Validation Accuracy = 0.941
Training Loss = 0.057
Validation Loss = 0.226
Elapsed time = 91.0 seconds

EPOCH 6 ...
Training Accuracy = 0.992
Validation Accuracy = 0.950
Training Loss = 0.033
Validation Loss = 0.204
Elapsed time = 67.0 seconds

EPOCH 7 ...
Training Accuracy = 0.992
Validation Accuracy = 0.951
Training Loss = 0.032
Validation Loss = 0.199
Elapsed time = 96.2 seconds

EPOCH 8 ...
Training Accuracy = 0.996
Validation Accuracy = 0.957
Training Loss = 0.019
Validation Loss = 0.181
Elapsed time = 134.4 seconds

EPOCH 9 ...
Training Accuracy = 0.994
```

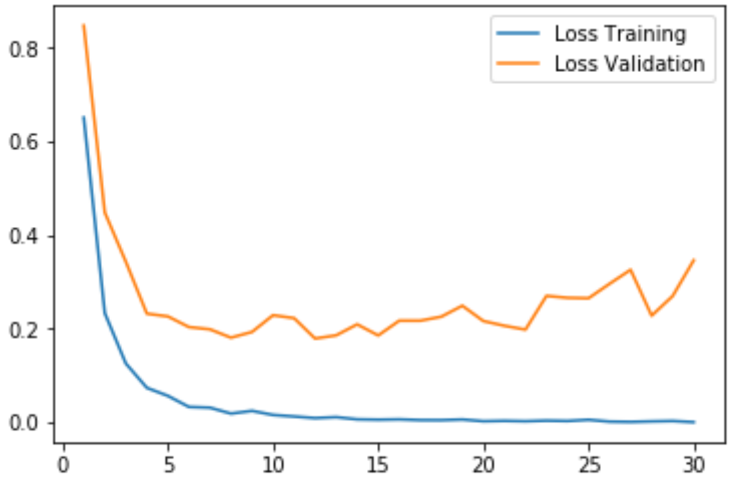
```
Validation Accuracy = 0.950
Training Loss = 0.025
Validation Loss = 0.193
Elapsed time = 102.4 seconds

EPOCH 10 ...
Training Accuracy = 0.996
Validation Accuracy = 0.956
Training Loss = 0.016
Validation Loss = 0.229
Elapsed time = 71.5 seconds

EPOCH 11 ...
Training Accuracy = 0.997
Validation Accuracy = 0.957
Training Loss = 0.013
Validation Loss = 0.223
Elapsed time = 72.6 seconds

EPOCH 12 ...
Training Accuracy = 0.998
Validation Accuracy = 0.958
Training Loss = 0.009
Validation Loss = 0.179
Elapsed time = 68.5 seconds

EPOCH 13 ...
Training Accuracy = 0.998
Validation Accuracy = 0.957
Training Loss = 0.011
Validation Loss = 0.186
Elapsed time = 66.7 seconds

EPOCH 14 ...
Training Accuracy = 0.999
Validation Accuracy = 0.951
Training Loss = 0.007
Validation Loss = 0.209
Elapsed time = 69.8 seconds

EPOCH 15 ...
Training Accuracy = 0.999
Validation Accuracy = 0.962
Training Loss = 0.006
Validation Loss = 0.186
Elapsed time = 68.6 seconds

EPOCH 16 ...
Training Accuracy = 0.998
Validation Accuracy = 0.957
Training Loss = 0.007
Validation Loss = 0.217
Elapsed time = 89.2 seconds

EPOCH 17 ...
Training Accuracy = 0.999
Validation Accuracy = 0.956
Training Loss = 0.005
Validation Loss = 0.217
Elapsed time = 92.2 seconds
```

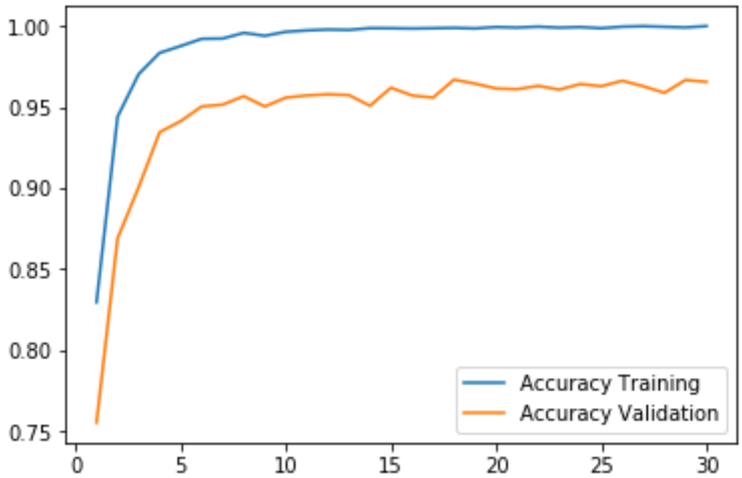
```
EPOCH 18 ...
Training Accuracy = 0.999
Validation Accuracy = 0.967
Training Loss = 0.005
Validation Loss = 0.226
Elapsed time = 94.2 seconds

EPOCH 19 ...
Training Accuracy = 0.998
Validation Accuracy = 0.964
Training Loss = 0.006
Validation Loss = 0.249
Elapsed time = 74.7 seconds

EPOCH 20 ...
Training Accuracy = 0.999
Validation Accuracy = 0.961
Training Loss = 0.003
Validation Loss = 0.216
Elapsed time = 76.7 seconds

EPOCH 21 ...
Training Accuracy = 0.999
Validation Accuracy = 0.961
Training Loss = 0.003
Validation Loss = 0.206
Elapsed time = 70.3 seconds

EPOCH 22 ...
Training Accuracy = 1.000
Validation Accuracy = 0.963
Training Loss = 0.003
Validation Loss = 0.198
Elapsed time = 63.5 seconds

EPOCH 23 ...
Training Accuracy = 0.999
Validation Accuracy = 0.961
Training Loss = 0.004
Validation Loss = 0.270
Elapsed time = 63.5 seconds

EPOCH 24 ...
Training Accuracy = 0.999
Validation Accuracy = 0.964
Training Loss = 0.003
Validation Loss = 0.266
Elapsed time = 63.3 seconds

EPOCH 25 ...
Training Accuracy = 0.999
Validation Accuracy = 0.963
Training Loss = 0.006
Validation Loss = 0.265
Elapsed time = 66.4 seconds

EPOCH 26 ...
Training Accuracy = 1.000
Validation Accuracy = 0.966
```

```
Training Loss = 0.002
Validation Loss = 0.296
Elapsed time = 64.0 seconds

EPOCH 27 ...
Training Accuracy = 1.000
Validation Accuracy = 0.963
Training Loss = 0.001
Validation Loss = 0.326
Elapsed time = 65.8 seconds

EPOCH 28 ...
Training Accuracy = 0.999
Validation Accuracy = 0.959
Training Loss = 0.002
Validation Loss = 0.228
Elapsed time = 70.4 seconds

EPOCH 29 ...
Training Accuracy = 0.999
Validation Accuracy = 0.967
Training Loss = 0.003
Validation Loss = 0.270
Elapsed time = 67.6 seconds

EPOCH 30 ...
Training Accuracy = 1.000
Validation Accuracy = 0.966
Training Loss = 0.001
Validation Loss = 0.346
Elapsed time = 64.9 seconds

Model saved
```

In [16]:

```
plot_ltd = plt.plot( list(range(1, len(loss_training_data) + 1)), loss_training_
data, label='Loss Training')
plot_lvd = plt.plot( list(range(1, len(loss_validation_data) + 1)), loss_validat
ion_data, label='Loss Validation')
plt.legend()
plt.show()

plot_atd = plt.plot( list(range(1, len(accuracy_training_data) + 1)), accuracy_t
raining_data, label='Accuracy Training')
plot_avd = plt.plot( list(range(1, len(accuracy_validation_data) + 1)), accuracy
_validation_data, label='Accuracy Validation')
plt.legend(loc = 4)
plt.show()
```





In [17]:

```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

    test_accuracy, test_loss = evaluate(X_test, y_test)
    print("Test Accuracy = {:.3f}".format(test_accuracy))
    print("Test Loss = {:.3f}".format(test_loss))
```

```
Test Accuracy = 0.955
Test Loss = 0.367
```

Step 3: Test a Model on New Images

To give yourself more insight into how your model is working, download at least five pictures of German traffic signs from the web and use your model to predict the traffic sign type.

You may find `signnames.csv` useful as it contains mappings from the class id (integer) to the actual sign name.

Load and Output the Images

In [18]:

```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
```

In [19]:

```
import numpy as np
import cv2
```

In [20]:

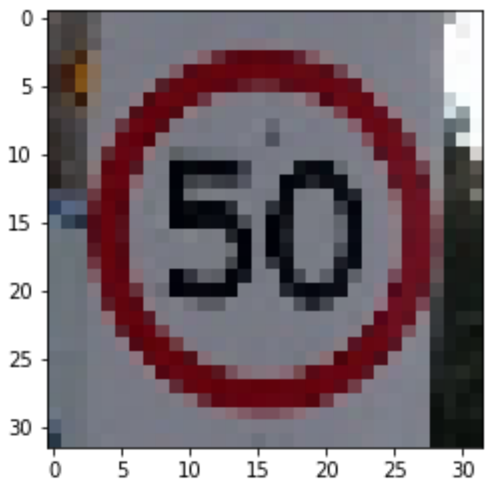
```
import os
files = os.listdir("signs-photos/")
print(files)
```

```
['1.png', '2.png', '3.png', '4.png', '5.png', '6.png', '7.png', '8.p
ng', '9.png', 'off.png']
```

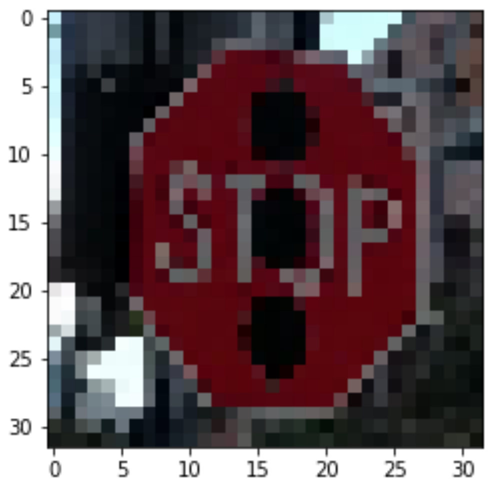
In [21]:

```
X_photos = np.empty([len(files), 32, 32, 3], dtype = np.int32)
for counter, file in enumerate(files):
    print(file)
    bgr_img = cv2.imread('signs-photos/' + file)
    rgb_img = cv2.cvtColor(bgr_img, cv2.COLOR_BGR2RGB)

    image = cv2.resize(rgb_img,(32,32))
    image = np.asarray(image, dtype=np.uint8)
    plt.imshow(image)
    plt.show()
    X_photos[counter] = image
```

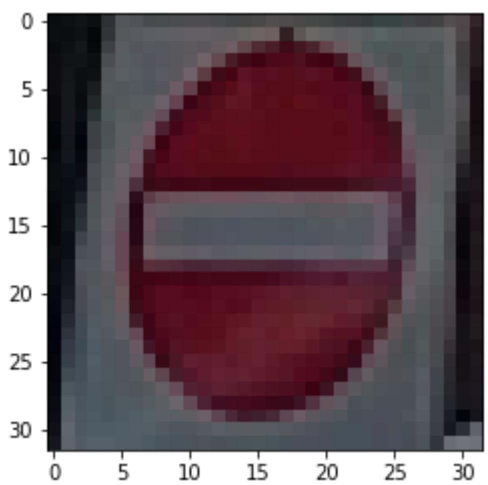
1.png



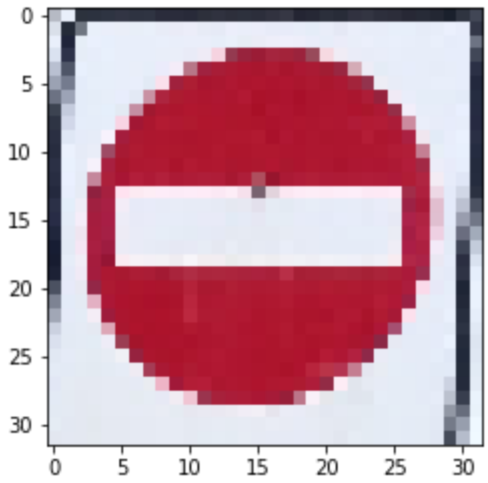
2.png



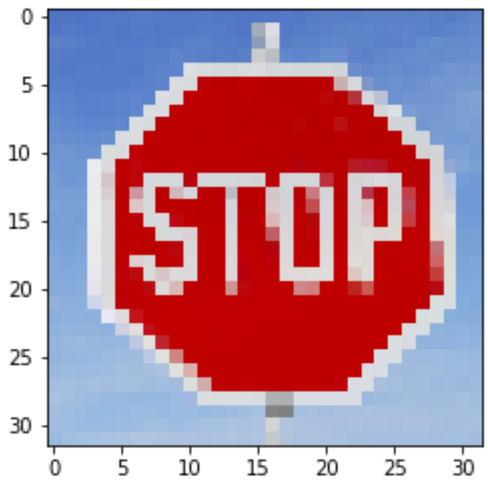
3.png



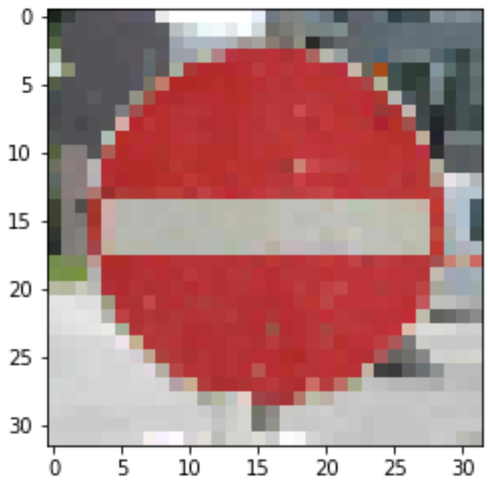
4.png



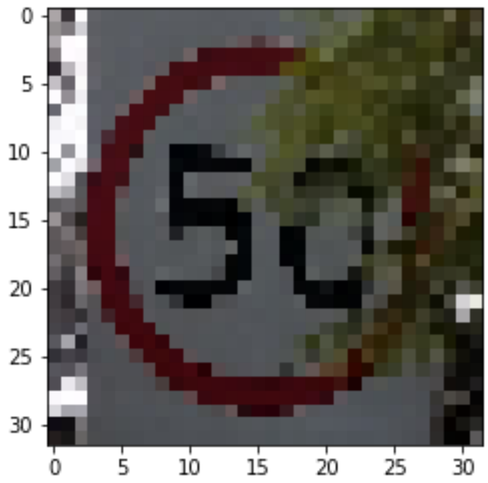
5.png



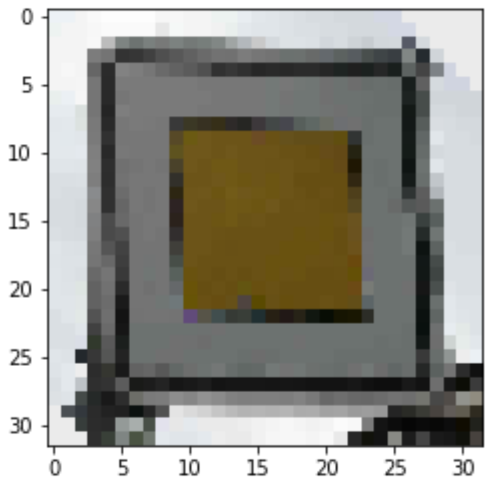
6.png



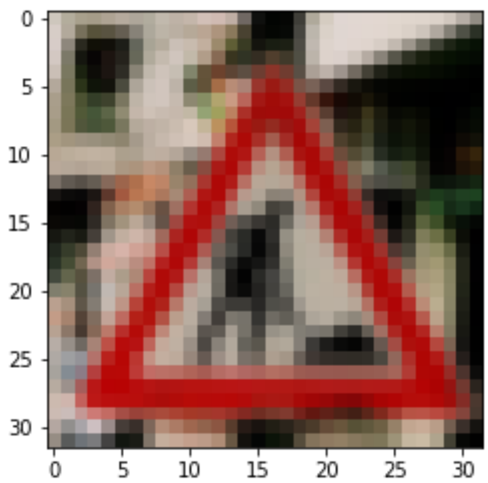
7.png



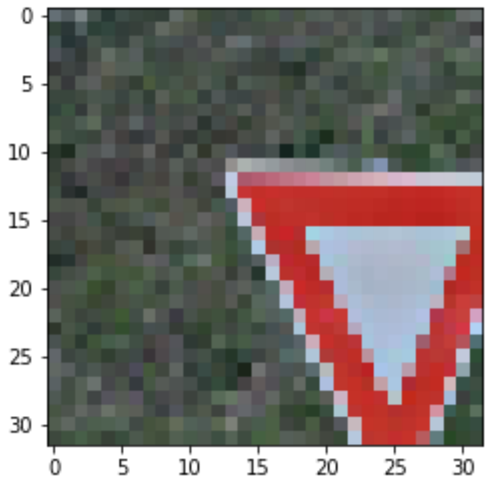
8.png



9.png



off.png



In [22]:

```
Y_photos = np.array([2, 14, 17, 17, 14, 17, 2, 12, 25, 13])
```

In [23]:

```
# Normalize data
X_photos = (X_photos/127.5) - 1.0
```

In [24]:

```
print(X_photos[1].shape)
```

```
(32, 32, 3)
```

In [25]:

```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

    photos_accuracy, photos_loss = evaluate(X_photos, Y_photos)
    print("Photos Accuracy = {:.3f}".format(photos_accuracy))
    print("Photos Loss = {:.3f}".format(photos_loss))
```

```
Photos Accuracy = 0.600
Photos Loss = 4.029
```

Predict the Sign Type for Each Image

In [26]:

```
### Run the predictions here and use the model to output the prediction for each
 image.
### Make sure to pre-process the images with the same pre-processing pipeline us
ed earlier.
### Feel free to use as many code cells as needed.
```

Analyze Performance

In [27]:

```
### Calculate the accuracy for these 5 new images.
### For example, if the model predicted 1 out of 5 signs correctly, it's 20% acc
urate on these new images.
```

Output Top 5 Softmax Probabilities For Each Image Found on the Web

For each of the new images, print out the model's softmax probabilities to show the **certainty** of the model's predictions (limit the output to the top 5 probabilities for each image). `tf.nn.top_k` (https://www.tensorflow.org/versions/r0.12/api_docs/python/nn.html#top_k) could prove helpful here.

The example below demonstrates how tf.nn.top_k can be used to find the top k predictions for each image.

`tf.nn.top_k` will return the values and indices (class ids) of the top k predictions. So if k=3, for each sign, it'll return the 3 largest probabilities (out of a possible 43) and the correspoding class ids.

Take this numpy array as an example. The values in the array represent predictions. The array contains softmax probabilities for five candidate images with six possible classes. `tk.nn.top_k` is used to choose the three classes with the highest probability:

```
# (5, 6) array
a = np.array([[ 0.24879643,  0.07032244,  0.12641572,  0.34763842,  0.0789
3497,
         0.12789202],
       [ 0.28086119,  0.27569815,  0.08594638,  0.0178669 ,  0.18063401,
         0.15899337],
       [ 0.26076848,  0.23664738,  0.08020603,  0.07001922,  0.1134371 ,
         0.23892179],
       [ 0.11943333,  0.29198961,  0.02605103,  0.26234032,  0.1351348 ,
         0.16505091],
       [ 0.09561176,  0.34396535,  0.0643941 ,  0.16240774,  0.24206137,
         0.09155967]])
```

Running it through `sess.run(tf.nn.top_k(tf.constant(a), k=3))` produces:

```
TopKV2(values=array([[ 0.34763842,  0.24879643,  0.12789202],
       [ 0.28086119,  0.27569815,  0.18063401],
       [ 0.26076848,  0.23892179,  0.23664738],
       [ 0.29198961,  0.26234032,  0.16505091],
       [ 0.34396535,  0.24206137,  0.16240774]]), indices=array([[3, 0,
 5],
       [0, 1, 4],
       [0, 5, 1],
       [1, 3, 5],
       [1, 4, 3]], dtype=int32))
```

Looking just at the first row we get `[ 0.34763842,  0.24879643,  0.12789202]`, you can confirm these are the 3 largest probabilities in `a`. You'll also notice `[3, 0, 5]` are the corresponding indices.

In [28]:

```
### Print out the top five softmax probabilities for the predictions on the Germ
an traffic sign images found on the web.
### Feel free to use as many code cells as needed.
```

In [29]:

```
softmax_operation = tf.nn.softmax(logits)

def evaluate_softmax(X_data, y_data):
    num_examples = len(X_data)
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[offset:offse
t+BATCH_SIZE]
        softmax = sess.run(softmax_operation, feed_dict={x: batch_x, y: batch_y,
 keep_prob: 1.0})
    return softmax
```

In [30]:

```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

    softmax = evaluate_softmax(X_photos, Y_photos)
    top_five = sess.run(tf.nn.top_k(softmax, k=5))
    print("Softmax = {}".format(top_five))
```

```
Softmax = TopKV2(values=array([[  1.00000000e+00,   4.30967440e-09,
  1.42218536e-17,
          9.53378776e-20,   1.62866915e-21],
       [  7.73076773e-01,   1.79583982e-01,   3.36047113e-02,
          9.63292457e-03,   3.42484540e-03],
       [  1.00000000e+00,   5.54633986e-12,   1.10284088e-14,
          3.44075206e-17,   4.05204556e-18],
       [  1.00000000e+00,   5.16569819e-15,   4.70565541e-34,
          4.35512306e-35,   3.72084542e-35],
       [  1.00000000e+00,   3.86526966e-38,   0.00000000e+00,
          0.00000000e+00,   0.00000000e+00],
       [  1.00000000e+00,   3.32183042e-22,   1.83248592e-25,
          2.49751475e-27,   2.14229555e-32],
       [  9.99870896e-01,   1.29137581e-04,   4.21303617e-11,
          3.15854357e-11,   2.39201003e-11],
       [  8.42888176e-01,   1.25778332e-01,   1.55824684e-02,
          1.53069142e-02,   2.18308720e-04],
       [  1.00000000e+00,   4.47760942e-13,   3.41418779e-15,
          3.39044438e-18,   2.17524952e-19],
       [  9.99978185e-01,   2.18388923e-05,   7.85878829e-09,
          5.14907539e-09,   3.56576924e-09]], dtype=float32), indice
s=array([[ 2,  1,  5,  3,  7],
       [ 1, 14,  2, 11,  5],
       [17, 14,  0, 26,  1],
       [17, 14,  0, 22, 26],
       [14, 17,  0,  1,  2],
       [17, 14,  0, 26,  1],
       [ 1,  2,  4,  5, 31],
       [11, 21, 30,  7,  6],
       [25, 20, 11, 27, 18],
       [12, 26, 14, 15, 13]], dtype=int32))
```

In [31]:

```
for v, i in zip(top_five.values, top_five.indices):
    print("*****")
    for p, s in zip(v, i):
        print("|{} | {}|".format(p, sign_names[str(s)]))
```

```
*****
|1.0 | Speed limit (50km/h)|
|4.309674395130969e-09 | Speed limit (30km/h)|
|1.422185358360589e-17 | Speed limit (80km/h)|
|9.533787764723268e-20 | Speed limit (60km/h)|
|1.628669148760936e-21 | Speed limit (100km/h)|
*****
|0.7730767726898193 | Speed limit (30km/h)|
|0.17958398163318634 | Stop|
|0.033604711294174194 | Speed limit (50km/h)|
|0.009632924571633339 | Right-of-way at the next intersection|
|0.003424845403060317 | Speed limit (80km/h)|
*****
|1.0 | No entry|
|5.5463398630772875e-12 | Stop|
|1.1028408783799511e-14 | Speed limit (20km/h)|
|3.4407520565041497e-17 | Traffic signals|
|4.052045560216044e-18 | Speed limit (30km/h)|
*****
|1.0 | No entry|
|5.165698191466131e-15 | Stop|
|4.705655414910278e-34 | Speed limit (20km/h)|
|4.3551230568922565e-35 | Bumpy road|
|3.720845418451188e-35 | Traffic signals|
*****
|1.0 | Stop|
|3.865269657252509e-38 | No entry|
|0.0 | Speed limit (20km/h)|
|0.0 | Speed limit (30km/h)|
|0.0 | Speed limit (50km/h)|
*****
|1.0 | No entry|
|3.3218304154487527e-22 | Stop|
|1.8324859188973467e-25 | Speed limit (20km/h)|
|2.4975147518162684e-27 | Traffic signals|
|2.142295548829814e-32 | Speed limit (30km/h)|
*****
|0.9998708963394165 | Speed limit (30km/h)|
|0.00012913758109789342 | Speed limit (50km/h)|
|4.2130361699310725e-11 | Speed limit (70km/h)|
|3.158543565584537e-11 | Speed limit (80km/h)|
|2.3920100275520717e-11 | Wild animals crossing|
*****
|0.8428881764411926 | Right-of-way at the next intersection|
|0.12577833235263824 | Double curve|
|0.01558246836066246 | Beware of ice/snow|
|0.01530691422522068 | Speed limit (100km/h)|
|0.00021830871992278844 | End of speed limit (80km/h)|
*****
|1.0 | Road work|
|4.477609418223477e-13 | Dangerous curve to the right|
|3.4141877930241645e-15 | Right-of-way at the next intersection|
|3.3904443792147465e-18 | Pedestrians|
|2.1752495230338266e-19 | General caution|
*****
|0.9999781847000122 | Priority road|
|2.183889228035696e-05 | Traffic signals|
|7.858788286796425e-09 | Stop|
|5.149075388288793e-09 | No vehicles|
|3.565769235436278e-09 | Yield|
```

Project Writeup

Once you have completed the code implementation, document your results in a project writeup using this template (https://github.com/udacity/CarND-Traffic-Sign-Classifier-Project/blob/master/writeup_template.md) as a guide. The writeup can be in a markdown or pdf file.

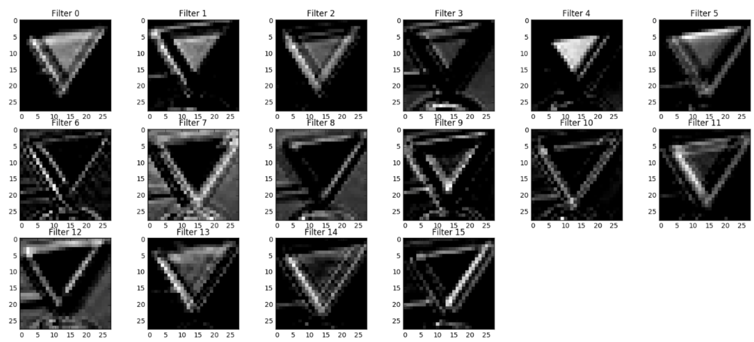
> **Note**: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to \n", "**File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

Step 4 (Optional): Visualize the Neural Network's State with Test Images

This Section is not required to complete but acts as an additional excersise for understaning the output of a neural network's weights. While neural networks can be a great learning device they are often referred to as a black box. We can understand what the weights of a neural network look like better by plotting their feature maps. After successfully training your neural network you can see what it's feature maps look like by plotting the output of the network's weight layers in response to a test stimuli image. From these plotted feature maps, it's possible to see what characteristics of an image the network finds interesting. For a sign, maybe the inner network feature maps react with high activation to the sign's boundary outline or to the contrast in the sign's painted symbol.

Provided for you below is the function code that allows you to get the visualization output of any tensorflow weight layer you want. The inputs to the function should be a stimuli image, one used during training or a new one you provided, and then the tensorflow variable name that represents the layer's state during the training process, for instance if you wanted to see what the [LeNet lab's (https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/6df7ae49-c61c-4bb2-a23e-6527e69209ec/lessons/601ae704-1035-4287-8b11-e2c2716217ad/concepts/d4aca031-508f-4e0b-b493-e7b706120f81)](https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/6df7ae49-c61c-4bb2-a23e-6527e69209ec/lessons/601ae704-1035-4287-8b11-e2c2716217ad/concepts/d4aca031-508f-4e0b-b493-e7b706120f81) feature maps looked like for it's second convolutional layer you could enter conv2 as the tf_activation variable.

For an example of what feature map outputs look like, check out NVIDIA's results in their paper [End-to-End Deep Learning for Self-Driving Cars (https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/)](https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/) in the section Visualization of internal CNN State. NVIDIA was able to show that their network's inner weights had high activations to road boundary lines by comparing feature maps from an image with a clear path to one without. Try experimenting with a similar test to show that your trained network's weights are looking for interesting features, whether it's looking at differences in feature maps from images with or without a sign, or even what feature maps look like in a trained network vs a completely untrained one on the same sign image.



Your output should look something like this (above)

In [33]:

```
### Visualize your network's feature maps here.
### Feel free to use as many code cells as needed.

# image_input: the test image being fed into the network to produce the feature
 maps
# tf_activation: should be a tf variable name used during your training procedur
e that represents the calculated state of a specific weight layer
# activation_min/max: can be used to view the activation contrast in more detai
l, by default matplot sets min and max to the actual min and max values of the o
utput
# plt_num: used to plot out multiple different weight feature map sets on the sa
me block, just extend the plt number for each new feature map entry

def outputFeatureMap(image_input, tf_activation, activation_min=-1, activation_m
ax=-1 ,plt_num=1):
    # Here make sure to preprocess your image_input in a way your network expect
s
    # with size, normalization, ect if needed
    # image_input =
    # Note: x should be the same name as your network's tensorflow data placehol
der variable
    # If you get an error tf_activation is not defined it may be having trouble
 accessing the variable from inside a function
    activation = tf_activation.eval(session=sess,feed_dict={x : image_input})
    featuremaps = activation.shape[3]
    plt.figure(plt_num, figsize=(15,15))
    for featuremap in range(featuremaps):
        plt.subplot(6,8, featuremap+1) # sets the number of feature maps to show
 on each row and column
        plt.title('FeatureMap ' + str(featuremap)) # displays the feature map nu
mber
        if activation_min != -1 & activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", v
min =activation_min, vmax=activation_max, cmap="gray")
        elif activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", v
max=activation_max, cmap="gray")
        elif activation_min !=-1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", v
min=activation_min, cmap="gray")
        else:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", c
map="gray")
```

In [34]:

```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    outputFeatureMap(X_photos, conv_1)
```