Traffic_Sign_Classifier

July 17, 2019

1 Self-Driving Car Engineer Nanodegree

1.1 Deep Learning

1.2 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 ",”**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 that can be used to guide the writing process. Completing the code template and writeup template will cover all of the rubric points for this project.

The rubric 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.

1.3 Step 0: Load The Data

[1]:
```
# Load pickled data
import pickle

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

training_file = '../data/train.p'
validation_file='../data/valid.p'
testing_file = '../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']
```

1.4 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 might be useful for calculating some of the summary results.

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

[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 = X_train.shape[0]

# TODO: Number of validation examples
n_validation = X_valid.shape[0]

# TODO: Number of testing examples.
n_test = X_test.shape[0]

# 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 = max(y_train) + 1

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
```

1.4.2 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 examples and gallery 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?

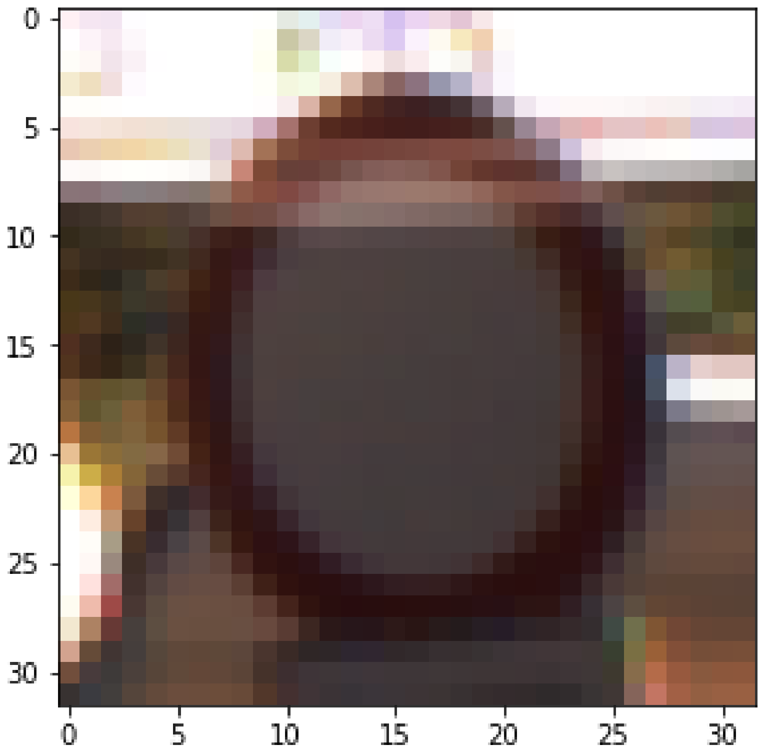
[3]:
```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import matplotlib.pyplot as plt
# Visualizations will be shown in the notebook.
%matplotlib inline
```

```
import random
import numpy as np
```

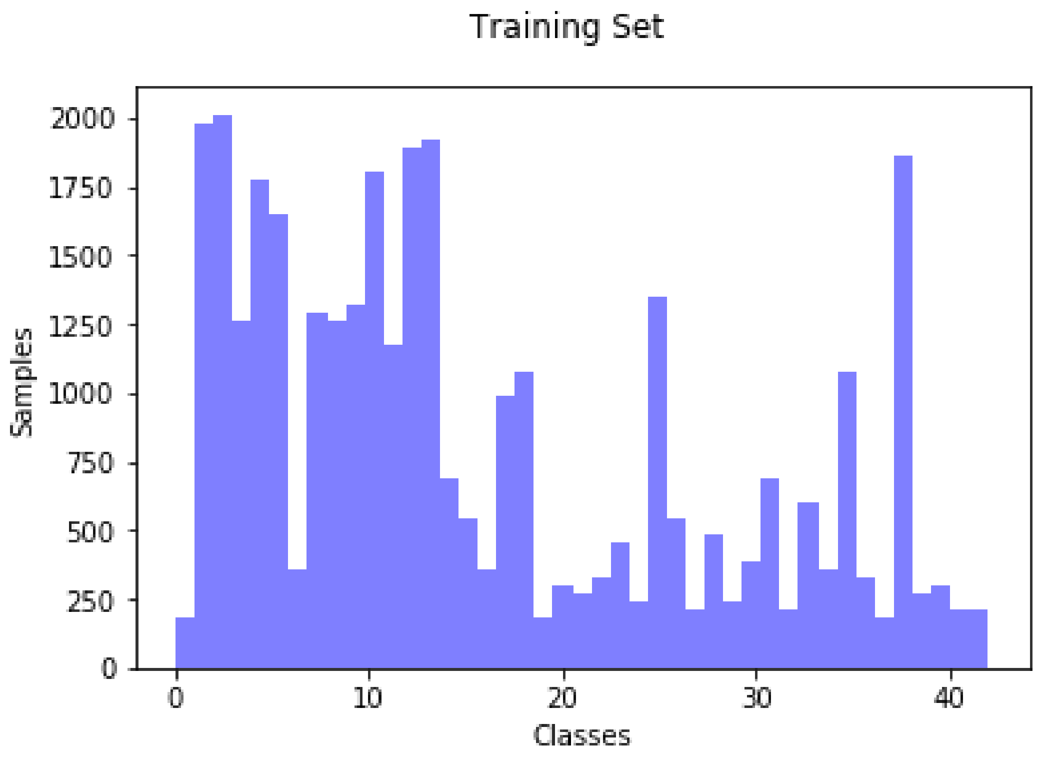
[4]:
```
X_train_ = X_train.copy()
X_valid_ = X_valid.copy()
X_test_ = X_test.copy()
```

[5]:
```
index = random.randint(0, n_train)
image = X_train_[index].squeeze()

plt.figure(figsize=(6.4,4.8))
plt.imshow(image)
print(y_train[index])
```

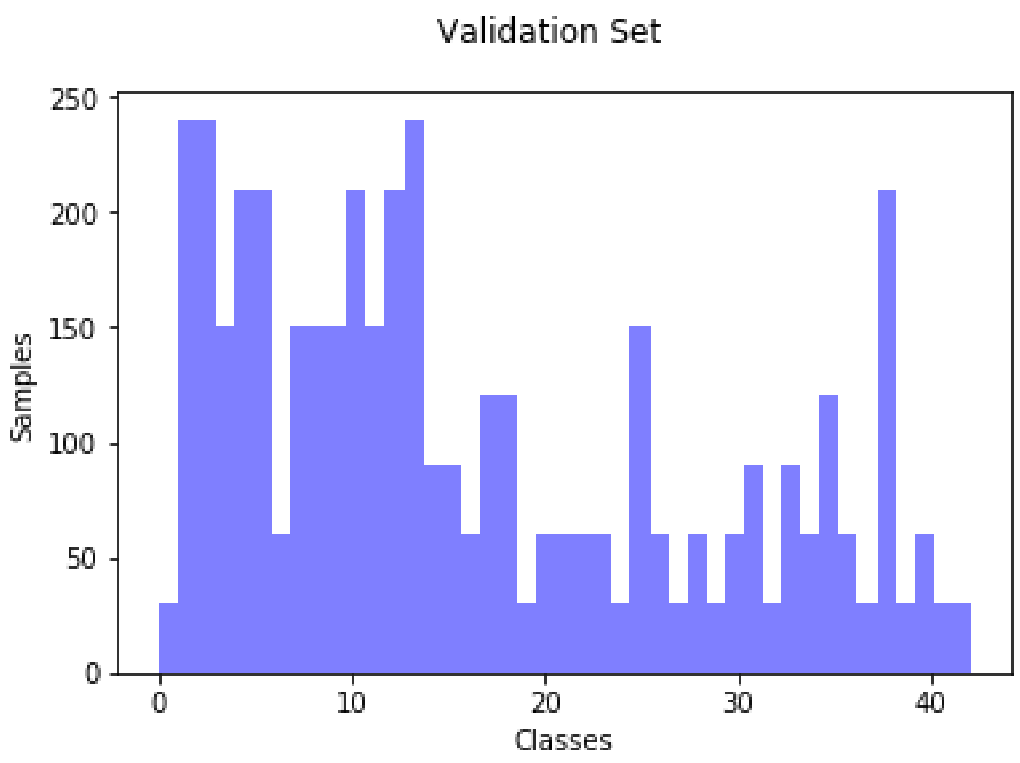
```
15
```



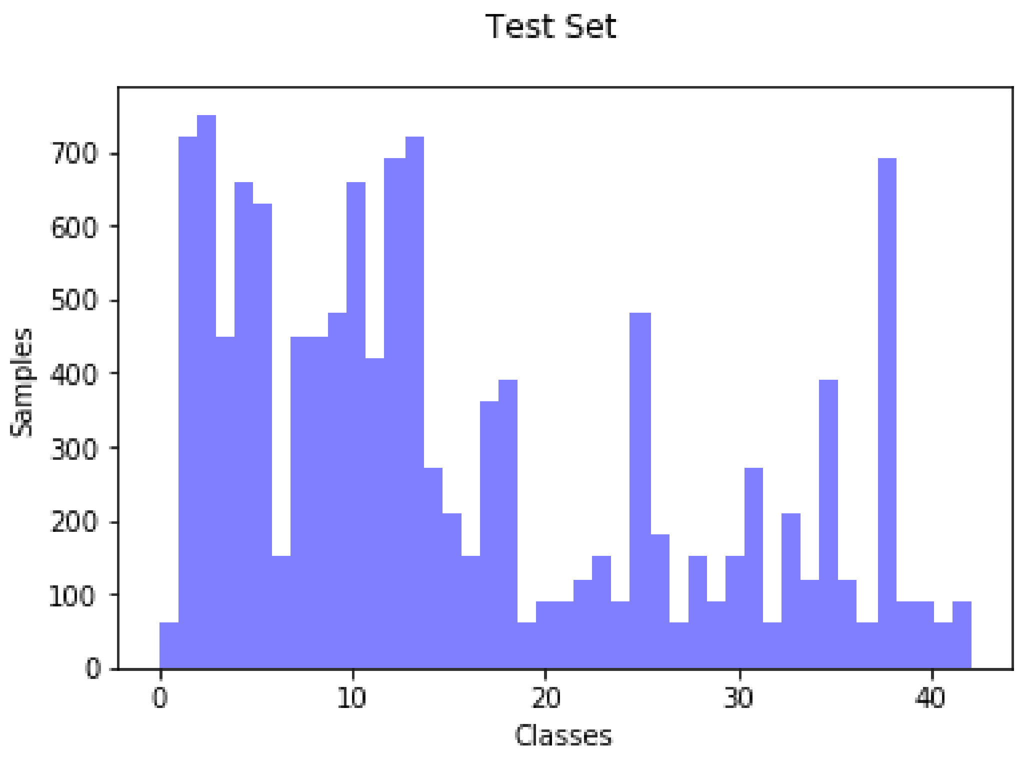
[6]:
```
fig, axs = plt.subplots(1, 1)
n_train_, bins_train, patches_train = axs.hist(y_train, n_classes,␣
 ↪facecolor='blue', alpha=0.5)
axs.set_xlabel("Classes")
axs.set_ylabel("Samples")
fig.suptitle("Training Set")
plt.savefig("Figures/train_class_hist.jpg")
```



[7]:
```
fig, axs = plt.subplots(1, 1)
n_valid_, bins_valid, patches_valid = axs.hist(y_valid, n_classes,␣
 ↪facecolor='blue', alpha=0.5)
axs.set_xlabel("Classes")
axs.set_ylabel("Samples")
fig.suptitle("Validation Set")
plt.savefig("Figures/valid_class_hist.jpg")
```



[8]:
```
fig, axs = plt.subplots(1, 1)
n_test_, bins_test, patches_test = axs.hist(y_test, n_classes,␣
 ↪facecolor='blue', alpha=0.5)
axs.set_xlabel("Classes")
axs.set_ylabel("Samples")
fig.suptitle("Test Set")
plt.savefig("Figures/test_class_hist.jpg")
```



[9]:
```
print(bins_train)
```

```
[ 0.          0.97674419  1.95348837  2.93023256  3.90697674  4.88372093
  5.86046512  6.8372093   7.81395349  8.79069767  9.76744186 10.74418605
 11.72093023 12.69767442 13.6744186  14.65116279 15.62790698 16.60465116
 17.58139535 18.55813953 19.53488372 20.51162791 21.48837209 22.46511628
 23.44186047 24.41860465 25.39534884 26.37209302 27.34883721 28.3255814
 29.30232558 30.27906977 31.25581395 32.23255814 33.20930233 34.18604651
 35.1627907  36.13953488 37.11627907 38.09302326 39.06976744 40.04651163
 41.02325581 42.        ]
```

[10]:
```
print(n_train_)
```

```
[ 180. 1980. 2010. 1260. 1770. 1650.  360. 1290. 1260. 1320. 1800. 1170.
 1890. 1920.  690.  540.  360.  990. 1080.  180.  300.  270.  330.  450.
  240. 1350.  540.  210.  480.  240.  390.  690.  210.  599.  360. 1080.
  330.  180. 1860.  270.  300.  210.  210.]
```

1.5 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.

The LeNet-5 implementation shown in the classroom 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. 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.

1.5.1 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.

[11]:
```
### Preprocess the data here. It is required to normalize the data. Other␣
↪preprocessing steps could include
### converting to grayscale, etc.
### Feel free to use as many code cells as needed.
```

1.5.2 Data Augmentation

[12]:
```
# Data augmentation code from:
# https://medium.com/ymedialabs-innovation/
↪data-augmentation-techniques-in-cnn-using-tensorflow-371ae43d5be9
```

[13]:
```
import tensorflow as tf
import matplotlib.image as mpimg
import cv2

IMAGE_SIZE = image_shape[0]
```

1.5.3 Data Augmentation: Scaling

[14]:
```
def central_scale_images(X_imgs, scales):
    # Various settings needed for Tensorflow operation
    boxes = np.zeros((len(scales), 4), dtype = np.float32)
    for index, scale in enumerate(scales):
        x1 = y1 = 0.5 - 0.5 * scale # To scale centrally
        x2 = y2 = 0.5 + 0.5 * scale
        boxes[index] = np.array([y1, x1, y2, x2], dtype = np.float32)
    box_ind = np.zeros((len(scales)), dtype = np.int32)
    crop_size = np.array([IMAGE_SIZE, IMAGE_SIZE], dtype = np.int32)

    X_scale_data = []
    tf.reset_default_graph()
    X = tf.placeholder(tf.float32, shape = (1, IMAGE_SIZE, IMAGE_SIZE, 3))
    # Define Tensorflow operation for all scales but only one base image at a␣
 ↪time
    tf_img = tf.image.crop_and_resize(X, boxes, box_ind, crop_size,␣
 ↪extrapolation_value=0)
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())

        for img_data in X_imgs:
            batch_img = np.expand_dims(img_data, axis = 0)/255.
            scaled_imgs = sess.run(tf_img, feed_dict = {X: batch_img})
            X_scale_data.extend(scaled_imgs)

    X_scale_data = np.array(X_scale_data, dtype = np.float32)
    X_scale_data = (255.*X_scale_data).astype(np.uint8)
    return X_scale_data
```

1.5.4 Data Augmentation: Translation

[15]:
```
def translate_images(X_imgs, offsets):
    X_imgs = X_imgs/255.
    # offsets = np.zeros((len(X_imgs), 2), dtype = np.float32)
    # base_offset = np.random.randint(low=-2, high=3, size=2).astype(np.
 ↪float32)
    # offsets[:, :] = base_offset
    # print(offsets)
    base_offset = offsets[0]
    size = np.array([IMAGE_SIZE, IMAGE_SIZE], dtype = np.int32)

    X_translated_arr = np.zeros((len(X_imgs), IMAGE_SIZE, IMAGE_SIZE, 3), dtype␣
 ↪= np.float32)
    X_translated_arr.fill(0.0) # Filling background color
    h_start = 0
    h_end = IMAGE_SIZE
    if base_offset[0]>0:
        h_end -= base_offset[0]
    elif base_offset[0]<0:
        h_start -= base_offset[0]

    w_start = 0
    w_end = IMAGE_SIZE
    if base_offset[1]>0:
        w_end -= base_offset[1]
    elif base_offset[1]<0:
        w_start -= base_offset[1]

    h_start = int(h_start)
    h_end = int(h_end)
    w_start = int(w_start)
    w_end = int(w_end)

    tf.reset_default_graph()
    glimpses = tf.image.extract_glimpse(X_imgs, size, offsets, centered=True,␣
↪normalized=False)
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())

        X_translated = sess.run(glimpses)
        X_translated_arr[:, h_start:h_end, w_start:w_end, :] = X_translated[:,␣
↪h_start:h_end, w_start:w_end, :]
        X_translated_ = (255.*X_translated_arr).astype(np.uint8)

    return X_translated_
```

1.5.5 Data Augmentation: Rotation

[16]:
```
from math import pi
```

[17]:
```
def rotate_images(X_imgs, radian_arr):

    tf.reset_default_graph()
    X = tf.placeholder(tf.float32, shape = (None, IMAGE_SIZE, IMAGE_SIZE, 3))
    radian = tf.placeholder(tf.float32, shape = (len(X_imgs)))
    tf_img = tf.contrib.image.rotate(X, radian)
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        rotated_imgs = sess.run(tf_img, feed_dict = {X: X_imgs, radian:␣
↪radian_arr})

    rotated_imgs = np.array(rotated_imgs, dtype = np.float32)
    rotated_imgs_ = (rotated_imgs).astype(np.uint8)
    return rotated_imgs_
```

1.5.6 Data Augmentation on Training Images: Scaling, Translation, and Rotation

[18]:
```
def apply_transforms(X_data, y_data, n_jittered):

    X_trans = X_data.copy()
    y_trans = y_data.copy()
    y_trans_ = y_data.copy()

    for i in range(n_jittered):

        scales = list(np.random.uniform(0.9, 1.1, 1))
        base_offset = np.random.randint(low=-2, high=3, size=2).astype(np.
↪float32)
        offsets = np.zeros((len(X_data), 2), dtype = np.float32)
        offsets[:, :] = base_offset
        angle_degree_ = np.random.uniform(-15, 15, 1)
        angle_degree = (angle_degree_ + 360) % 360
        angle_radian = angle_degree * pi / 180  # Convert to radian
        angle_arr = list(angle_radian)*len(X_data)

        scaled_imgs = central_scale_images(X_data, scales)
        translated_imgs = translate_images(scaled_imgs, offsets)
        rotated_imgs = rotate_images(translated_imgs, angle_arr)

        X_trans = np.vstack((X_trans, rotated_imgs))
        y_trans = np.hstack((y_trans, y_trans_))

    return X_trans, y_trans
```

[19]:
```
print(type(X_train))
print(X_train.dtype)
print(X_train.shape)
print(np.amax(X_train))
print(np.amin(X_train))
```

```
<class 'numpy.ndarray'>
uint8
(34799, 32, 32, 3)
255
0
```

[20]:
```
print(type(y_train))
print(y_train.dtype)
print(y_train.shape)
print(np.amax(y_train))
print(np.amin((y_train)))
```

```
<class 'numpy.ndarray'>
uint8
(34799,)
42
0
```

[21]:
```
n_data_trans = 5
X_train, y_train = apply_transforms(X_train, y_train, n_data_trans)
```

```
WARNING: The TensorFlow contrib module will not be included in TensorFlow 2.0.
For more information, please see:
  * https://github.com/tensorflow/community/blob/master/rfcs/20180907-contrib-
sunset.md
  * https://github.com/tensorflow/addons
If you depend on functionality not listed there, please file an issue.
```

[22]:
```
print(type(X_train))
print(X_train.dtype)
print(X_train.shape)
print(np.amax(X_train))
print(np.amin(X_train))
```

```
<class 'numpy.ndarray'>
uint8
(208794, 32, 32, 3)
255
0
```

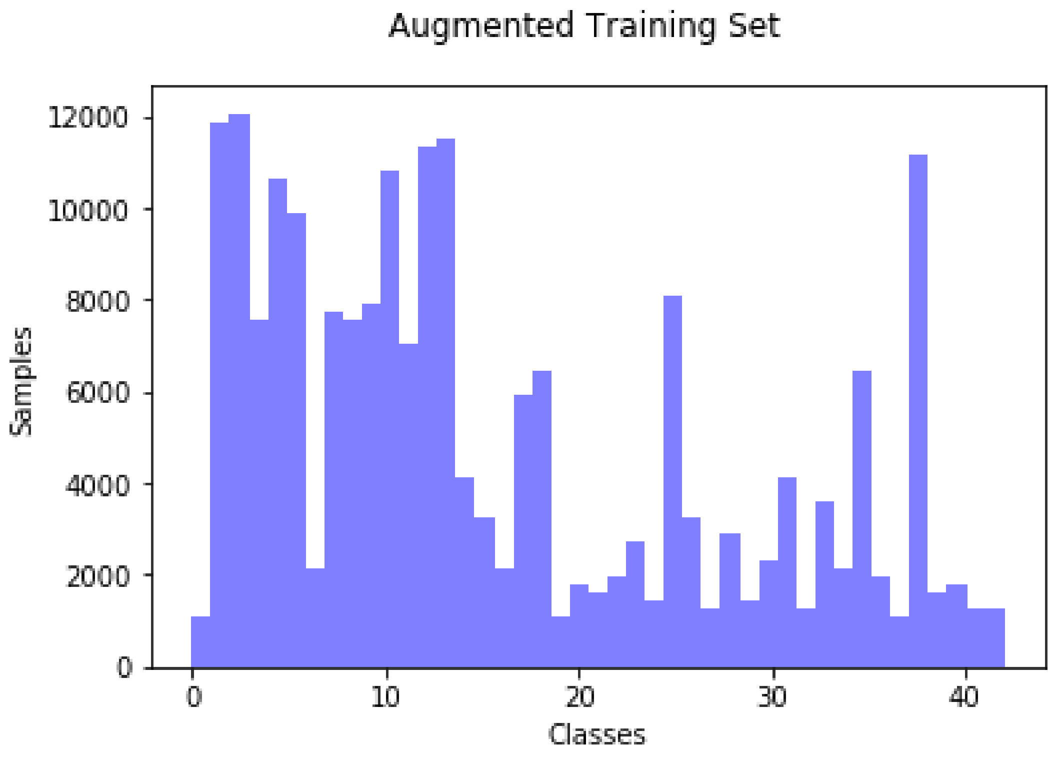
[23]:
```
print(type(y_train))
print(y_train.dtype)
print(y_train.shape)
print(np.amax(y_train))
print(np.amin(y_train))
```

```
<class 'numpy.ndarray'>
uint8
(208794,)
42
0
```

[24]:
```
print(6*34799)
```

```
208794
```

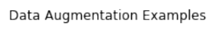
[25]:
```
fig, axs = plt.subplots(1, 1)
n_train_, bins_train, patches_train = axs.hist(y_train, n_classes,␣
 ↪facecolor='blue', alpha=0.5)
axs.set_xlabel("Classes")
axs.set_ylabel("Samples")
fig.suptitle("Augmented Training Set")
plt.savefig("Figures/aug_train_class_hist.jpg")
```

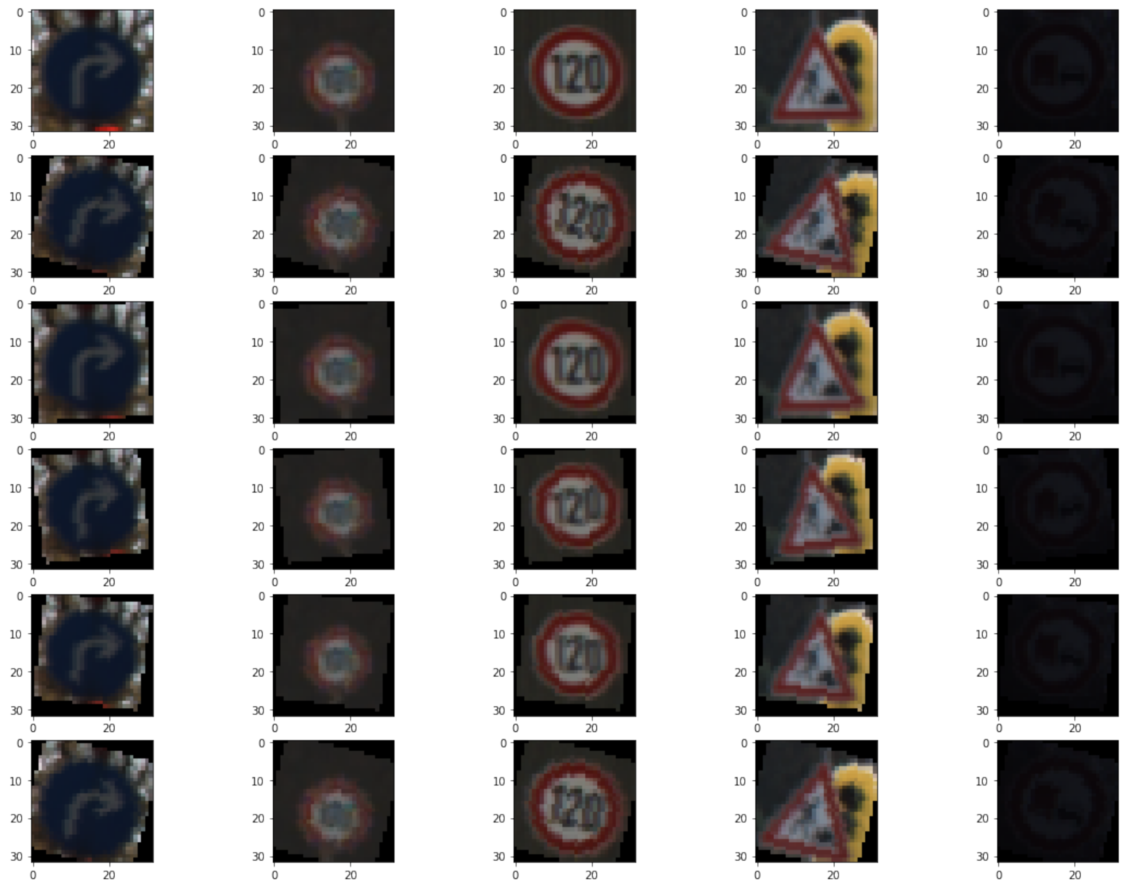


[26]:
```
n_im = 5
im_index = []
for i in range(n_im):
    index = random.randint(0, n_train)
    im_index.append(index)
print(im_index)
```

```
[26514, 24990, 16973, 33473, 18627]
```

[27]:
```
Nr = n_data_trans + 1
Nc = n_im
fig, axs = plt.subplots(Nr, Nc, figsize=[3*6.4, 3*4.8])
for i in range(Nr):
    for j in range(Nc):
        axs[i,j].imshow(X_train[i*n_train + im_index[j]])
fig.suptitle("Data Augmentation Examples")
plt.savefig("Figures/data_aug.jpg")
```





1.5.7 Data Pre-processing: Grayscale

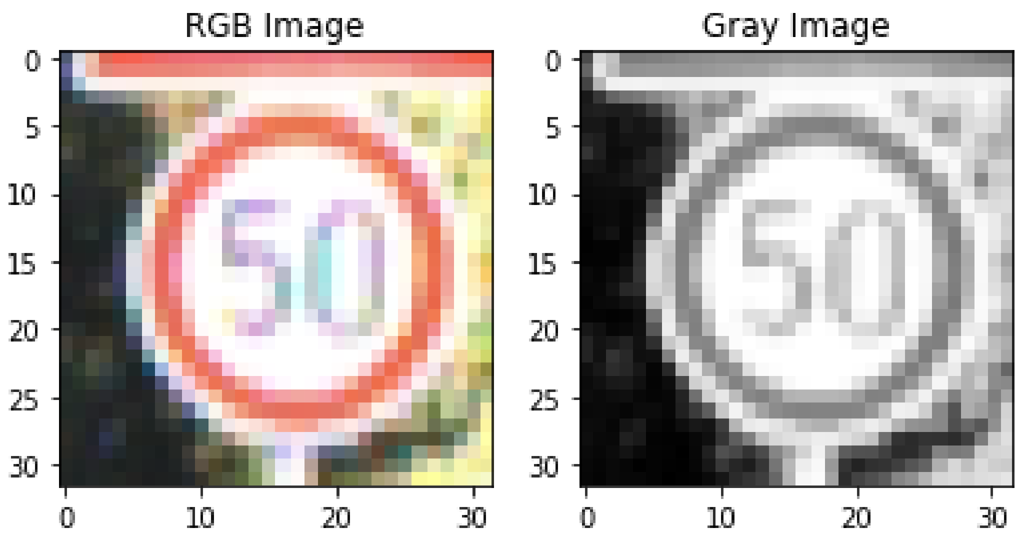
[28]:
```
print(X_train.shape)
```

```
(208794, 32, 32, 3)
```

[29]:
```
X_train = np.mean(X_train, axis=3, keepdims=True)
print(X_train.shape)
```

```
(208794, 32, 32, 1)
```

[30]:
```
index = random.randint(0, X_train_.shape[0])
fig, axs = plt.subplots(1, 2, figsize=[6.4,4.8])
axs[0].imshow(X_train_[index])
axs[0].set_title("RGB Image")
axs[1].imshow(X_train[index].squeeze(), cmap='gray')
axs[1].set_title("Gray Image")
plt.savefig("Figures/grayscale.jpg")
```



[31]:
```
X_valid = np.mean(X_valid, axis=3, keepdims=True)
X_test = np.mean(X_test, axis=3, keepdims=True)
```

1.5.8 Data Pre-processing: Normalization

[32]:
```
print(np.mean(X_train))
print(np.amin(X_train))
print(np.amax(X_train))
```

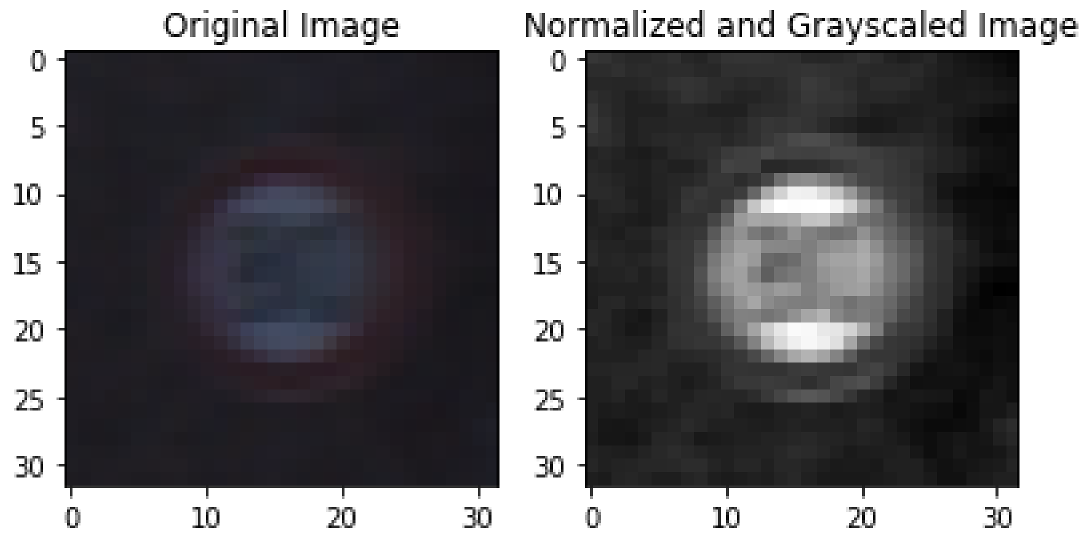
```
70.09362678651227
0.0
255.0
```

[33]:
```
X_train = (X_train - 128)/128
```

[34]:
```
print(np.mean(X_train))
print(np.amin(X_train))
print(np.amax(X_train))
```

```
-0.4523935407303742
-1.0
0.9921875
```

[35]:
```
index = random.randint(0, X_train_.shape[0])
fig, axs = plt.subplots(1, 2, figsize=[6.4,4.8])
axs[0].imshow(X_train_[index])
axs[0].set_title("Original Image")
axs[1].imshow(X_train[index].squeeze(), cmap='gray')
axs[1].set_title("Normalized and Grayscaled Image")
plt.savefig("Figures/normalization.jpg")
```



[36]:
```
X_valid = (X_valid - 128)/128
X_test = (X_test - 128)/128
```

1.5.9 Model Architecture

[37]:
```
### Define your architecture here.
### Feel free to use as many code cells as needed.
from sklearn.utils import shuffle
```

[38]:
```
from tensorflow.contrib.layers import flatten

def LeNet(x, keep_prob):
    # 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, 1, 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([400, 120], mean=mu,␣
↪stddev=sigma)),
        'wd2': tf.Variable(tf.truncated_normal([120, 84], mean=mu,␣
↪stddev=sigma)),
        'out': tf.Variable(tf.truncated_normal([84, n_classes], mean=mu,␣
↪stddev=sigma))
    }

    biases = {
        'bc1': tf.Variable(tf.zeros([6])),
        'bc2': tf.Variable(tf.zeros([16])),
        'bd1': tf.Variable(tf.zeros([120])),
        'bd2': tf.Variable(tf.zeros([84])),
        'out': tf.Variable(tf.zeros([n_classes]))
    }

    # TODO: Layer 1: Convolutional. Input = 32x32x3. Output = 28x28x6.
    conv1 = tf.nn.conv2d(x, weights['wc1'], strides=[1, 1, 1, 1],␣
↪padding='VALID')
    conv1 = tf.nn.bias_add(conv1, biases['bc1'])

    # TODO: Activation.
    conv1 = tf.nn.relu(conv1)

    # TODO: Pooling. Input = 28x28x6. Output = 14x14x6.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],␣
↪padding='VALID')

    # TODO: Layer 2: Convolutional. Output = 10x10x16.
    conv2 = tf.nn.conv2d(conv1, weights['wc2'], strides=[1, 1, 1, 1],␣
↪padding='VALID')
    conv2 = tf.nn.bias_add(conv2, biases['bc2'])

    # TODO: Activation.
    conv2 = tf.nn.relu(conv2)
    # TODO: Pooling. Input = 10x10x16. Output = 5x5x16.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],␣
 ↪padding='VALID')

    # TODO: Flatten. Input = 5x5x16. Output = 400.
    fc1 = flatten(conv2)

    # TODO: Layer 3: Fully Connected. Input = 400. Output = 120.
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])

    # TODO: Activation.
    fc1 = tf.nn.relu(fc1)

    # Dropout
    fc1 = tf.nn.dropout(fc1, keep_prob)

    # TODO: Layer 4: Fully Connected. Input = 120. Output = 84.
    fc2 = tf.add(tf.matmul(fc1, weights['wd2']), biases['bd2'])

    # TODO: Activation.
    fc2 = tf.nn.relu(fc2)

    # Dropout
    fc2 = tf.nn.dropout(fc2, keep_prob)

    # TODO: Layer 5: Fully Connected. Input = 84. Output = 43.
    logits = tf.add(tf.matmul(fc2, weights['out']), biases['out'])

    return logits
```

1.5.10 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.

[39]:
```
### 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.
EPOCHS = 50
BATCH_SIZE = 128

learn_rate =  0.001
```

[40]:
```
keep_prob = tf.placeholder(tf.float32) # probability to keep units
x = tf.placeholder(tf.float32, (None, 32, 32, 1))
```

```
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, n_classes)
```

[41]:
```
logits = 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 = learn_rate)
training_operation = optimizer.minimize(loss_operation)
```

```
WARNING:tensorflow:From C:\Users\hamad\Anaconda3\envs\tf-gpu\lib\site-
packages\tensorflow\python\framework\op_def_library.py:263: colocate_with (from
tensorflow.python.framework.ops) is deprecated and will be removed in a future
version.
Instructions for updating:
Colocations handled automatically by placer.
WARNING:tensorflow:From C:\Users\hamad\Anaconda3\envs\tf-gpu\lib\site-
packages\tensorflow\contrib\layers\python\layers\layers.py:1624: flatten (from
tensorflow.python.layers.core) is deprecated and will be removed in a future
version.
Instructions for updating:
Use keras.layers.flatten instead.
WARNING:tensorflow:From <ipython-input-38-ef2fce923896>:54: calling dropout
(from tensorflow.python.ops.nn_ops) with keep_prob is deprecated and will be
removed in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1 -
keep_prob`.
WARNING:tensorflow:From <ipython-input-41-34de70c8e41d>:2:
softmax_cross_entropy_with_logits (from tensorflow.python.ops.nn_ops) is
deprecated and will be removed in a future version.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See `tf.nn.softmax_cross_entropy_with_logits_v2`.
```

[42]:
```
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
    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:
↪offset+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y:␣
↪batch_y, keep_prob: 1.0})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

[43]:
```
valid_acc_prev = 0

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

    print("Training...")
    print()
    for i in range(EPOCHS):
        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})

        train_accuracy = evaluate(X_train, y_train)
        print("EPOCH {} ...".format(i+1))
        print("Training Accuracy = {:.3f}".format(train_accuracy))

        validation_accuracy = evaluate(X_valid, y_valid)
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()

        if validation_accuracy > valid_acc_prev:
            saver.save(sess, './lenet')
            print("Model saved")
            print()
            valid_acc_prev = validation_accuracy
```

```
Training...

EPOCH 1 ...
Training Accuracy = 0.930
Validation Accuracy = 0.884

Model saved

EPOCH 2 ...
Training Accuracy = 0.968
Validation Accuracy = 0.933
```

```
Model saved

EPOCH 3 ...
Training Accuracy = 0.981
Validation Accuracy = 0.947

Model saved

EPOCH 4 ...
Training Accuracy = 0.987
Validation Accuracy = 0.951

Model saved

EPOCH 5 ...
Training Accuracy = 0.989
Validation Accuracy = 0.951

EPOCH 6 ...
Training Accuracy = 0.990
Validation Accuracy = 0.960

Model saved

EPOCH 7 ...
Training Accuracy = 0.991
Validation Accuracy = 0.954

EPOCH 8 ...
Training Accuracy = 0.994
Validation Accuracy = 0.959

EPOCH 9 ...
Training Accuracy = 0.993
Validation Accuracy = 0.956

EPOCH 10 ...
Training Accuracy = 0.994
Validation Accuracy = 0.952

EPOCH 11 ...
Training Accuracy = 0.995
Validation Accuracy = 0.958

EPOCH 12 ...
Training Accuracy = 0.992
Validation Accuracy = 0.958
```

```
EPOCH 13 ...
Training Accuracy = 0.995
Validation Accuracy = 0.961

Model saved

EPOCH 14 ...
Training Accuracy = 0.996
Validation Accuracy = 0.957

EPOCH 15 ...
Training Accuracy = 0.996
Validation Accuracy = 0.962

Model saved

EPOCH 16 ...
Training Accuracy = 0.996
Validation Accuracy = 0.963

Model saved

EPOCH 17 ...
Training Accuracy = 0.997
Validation Accuracy = 0.966

Model saved

EPOCH 18 ...
Training Accuracy = 0.997
Validation Accuracy = 0.965

EPOCH 19 ...
Training Accuracy = 0.997
Validation Accuracy = 0.965

EPOCH 20 ...
Training Accuracy = 0.997
Validation Accuracy = 0.963

EPOCH 21 ...
Training Accuracy = 0.996
Validation Accuracy = 0.961

EPOCH 22 ...
Training Accuracy = 0.997
Validation Accuracy = 0.965
```

```
EPOCH 23 ...
Training Accuracy = 0.998
Validation Accuracy = 0.968

Model saved

EPOCH 24 ...
Training Accuracy = 0.997
Validation Accuracy = 0.959

EPOCH 25 ...
Training Accuracy = 0.997
Validation Accuracy = 0.964

EPOCH 26 ...
Training Accuracy = 0.998
Validation Accuracy = 0.966

EPOCH 27 ...
Training Accuracy = 0.998
Validation Accuracy = 0.968

EPOCH 28 ...
Training Accuracy = 0.998
Validation Accuracy = 0.968

EPOCH 29 ...
Training Accuracy = 0.998
Validation Accuracy = 0.971

Model saved

EPOCH 30 ...
Training Accuracy = 0.998
Validation Accuracy = 0.964

EPOCH 31 ...
Training Accuracy = 0.998
Validation Accuracy = 0.970

EPOCH 32 ...
Training Accuracy = 0.998
Validation Accuracy = 0.964

EPOCH 33 ...
Training Accuracy = 0.998
Validation Accuracy = 0.973
```

```
Model saved

EPOCH 34 ...
Training Accuracy = 0.998
Validation Accuracy = 0.966

EPOCH 35 ...
Training Accuracy = 0.996
Validation Accuracy = 0.962

EPOCH 36 ...
Training Accuracy = 0.998
Validation Accuracy = 0.966

EPOCH 37 ...
Training Accuracy = 0.998
Validation Accuracy = 0.968

EPOCH 38 ...
Training Accuracy = 0.999
Validation Accuracy = 0.972

EPOCH 39 ...
Training Accuracy = 0.998
Validation Accuracy = 0.962

EPOCH 40 ...
Training Accuracy = 0.998
Validation Accuracy = 0.968

EPOCH 41 ...
Training Accuracy = 0.999
Validation Accuracy = 0.971

EPOCH 42 ...
Training Accuracy = 0.999
Validation Accuracy = 0.969

EPOCH 43 ...
Training Accuracy = 0.998
Validation Accuracy = 0.964

EPOCH 44 ...
Training Accuracy = 0.999
Validation Accuracy = 0.973

EPOCH 45 ...
```

```
Training Accuracy = 0.999
Validation Accuracy = 0.969

EPOCH 46 ...
Training Accuracy = 0.999
Validation Accuracy = 0.975

Model saved

EPOCH 47 ...
Training Accuracy = 0.998
Validation Accuracy = 0.971

EPOCH 48 ...
Training Accuracy = 0.999
Validation Accuracy = 0.974

EPOCH 49 ...
Training Accuracy = 0.998
Validation Accuracy = 0.973

EPOCH 50 ...
Training Accuracy = 0.998
Validation Accuracy = 0.974
```

[43]:
```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

    class_valid_accuracy = []
    class_train_accuracy = []
    for class_num in range(n_classes):
        one_class = y_valid==class_num
        test_accuracy = evaluate(X_valid[one_class], y_valid[one_class])
        class_valid_accuracy.append(test_accuracy)

        one_class = y_train==class_num
        test_accuracy = evaluate(X_train[one_class], y_train[one_class])
        class_train_accuracy.append(test_accuracy)
```

```
WARNING:tensorflow:From C:\Users\hamad\Anaconda3\envs\tf-gpu\lib\site-
packages\tensorflow\python\training\saver.py:1266: checkpoint_exists (from
tensorflow.python.training.checkpoint_management) is deprecated and will be
removed in a future version.
Instructions for updating:
Use standard file APIs to check for files with this prefix.
INFO:tensorflow:Restoring parameters from .\lenet
```

[44]:
```python
for item in zip(range(n_classes), n_valid_, class_valid_accuracy):
    print(item)
```

```
(0, 30.0, 0.800000011920929)
(1, 240.0, 0.962500003973643)
(2, 240.0, 0.9916666666666667)
(3, 150.0, 1.0)
(4, 210.0, 1.0)
(5, 210.0, 0.9857142947968982)
(6, 60.0, 1.0)
(7, 150.0, 0.9933333333333333)
(8, 150.0, 0.9733333333333334)
(9, 150.0, 1.0)
(10, 210.0, 1.0)
(11, 150.0, 0.9866666666666667)
(12, 210.0, 0.9952380952380953)
(13, 240.0, 0.9916666666666667)
(14, 90.0, 1.0)
(15, 90.0, 1.0)
(16, 60.0, 0.9166666865348816)
(17, 120.0, 0.949999988079071)
(18, 120.0, 0.9833333492279053)
(19, 30.0, 1.0)
(20, 60.0, 0.7833333611488342)
(21, 60.0, 0.7666666507720947)
(22, 60.0, 1.0)
(23, 60.0, 0.9833333492279053)
(24, 30.0, 0.800000011920929)
(25, 150.0, 0.9599999976158142)
(26, 60.0, 0.9833333492279053)
(27, 30.0, 0.9666666388511658)
(28, 60.0, 1.0)
(29, 30.0, 1.0)
(30, 60.0, 1.0)
(31, 90.0, 1.0)
(32, 30.0, 1.0)
(33, 90.0, 0.9888888597488403)
(34, 60.0, 0.9833333492279053)
(35, 120.0, 1.0)
(36, 60.0, 1.0)
(37, 30.0, 1.0)
(38, 210.0, 1.0)
(39, 30.0, 1.0)
(40, 60.0, 0.7166666388511658)
(41, 30.0, 0.8999999761581421)
(42, 30.0, 0.9333333373069763)
```

[45]:
```python
for item in zip(range(n_classes), n_train_, class_train_accuracy):
    print(item)
```

```
(0, 1080.0, 0.8518518509688201)
(1, 11880.0, 0.9036195286596664)
(2, 12060.0, 0.8939469319670947)
(3, 7560.0, 0.9529100529100529)
(4, 10620.0, 0.9858757062146892)
(5, 9900.0, 0.9539393938671459)
(6, 2160.0, 0.979166665342119)
(7, 7740.0, 0.9052971574687219)
(8, 7560.0, 0.9560846560846561)
(9, 7920.0, 0.9816919191919192)
(10, 10800.0, 0.9866666666666667)
(11, 7020.0, 0.9756410252334725)
(12, 11340.0, 0.976984126984127)
(13, 11520.0, 0.99453125)
(14, 4140.0, 0.9741545894295697)
(15, 3240.0, 0.991358024691358)
(16, 2160.0, 0.9532407420652884)
(17, 5940.0, 0.9856902356902357)
(18, 6480.0, 0.9564814817758254)
(19, 1080.0, 0.9777777782192937)
(20, 1800.0, 0.9155555555555556)
(21, 1620.0, 0.8623456803368933)
(22, 1980.0, 0.9631313137333802)
(23, 2700.0, 0.8933333334216366)
(24, 1440.0, 0.8423611111111111)
(25, 8100.0, 0.9677777778366465)
(26, 3240.0, 0.8895061725451623)
(27, 1260.0, 0.9484126970881508)
(28, 2880.0, 0.9701388888888889)
(29, 1440.0, 0.9444444444444444)
(30, 2340.0, 0.9239316235240708)
(31, 4140.0, 0.9661835749368161)
(32, 1260.0, 0.9150793664039127)
(33, 3594.0, 0.9808013355592654)
(34, 2160.0, 0.9291666657836349)
(35, 6480.0, 0.9847222222222223)
(36, 1980.0, 0.9540404041608175)
(37, 1080.0, 0.9583333337748492)
(38, 11160.0, 0.9793010752688172)
(39, 1620.0, 0.9499999988226243)
(40, 1800.0, 0.8738888888888889)
(41, 1260.0, 0.932539681404356)
(42, 1260.0, 0.9214285719962347)
```

[46]:
```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

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

```
INFO:tensorflow:Restoring parameters from .\lenet
Test Accuracy = 0.959
```

1.6 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.

1.6.1 Load and Output the Images

[47]:
```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
import os, sys
```

[48]:
```
path = 'german-traffic-sign-examples/'
dirs = os.listdir(path)
print(dirs)
```

```
['bicycles-crossing.jpg', 'no-entry.jpg', 'priority-road.jpg', 'roundabout-
mandatory.jpg', 'yield.jpg']
```

[49]:
```
new_ims = []
for file in dirs:
    if file[-4:] == '.jpg':
        # print(file)
        img = plt.imread(path + file)
        new_ims.append(img)
print(type(new_ims))
print(len(new_ims))
im_num = 0
print(type(new_ims[im_num]))
print(new_ims[im_num].shape)
print(new_ims[im_num].dtype)
print(np.amax(new_ims[im_num]))
print(np.amin(new_ims[im_num]))
```

```
<class 'list'>
5
<class 'numpy.ndarray'>
(590, 640, 3)
uint8
255
0
```

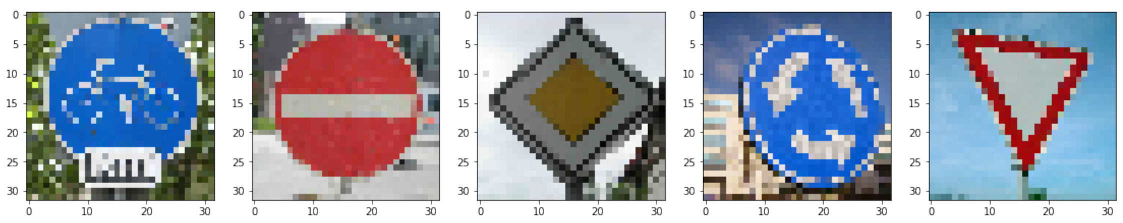
[50]:
```python
def scale_new_images(X_imgs):

    scales = []
    for img in X_imgs:
        # print(img.shape)
        im_dim = np.amin(img.shape[:2])
        # print(im_dim)
        scales.append((im_dim/img.shape[0], im_dim/img.shape[1]))
        # print(scales)
    # print(scales)

    boxes = np.zeros((len(X_imgs), 4), dtype = np.float32)
    for index, scale in enumerate(scales):
        y1 = 0.5 - 0.5 * scale[0]
        y2 = 0.5 + 0.5 * scale[0]
        x1 = 0.5 - 0.5 * scale[1]
        x2 = 0.5 + 0.5 * scale[1]
        boxes[index] = np.array([y1, x1, y2, x2], dtype = np.float32)
    # print(boxes)
    # print(type(boxes[0]))
    # print(boxes[0].shape)
    # print(boxes[0])

    box_ind = np.zeros(len(X_imgs), dtype = np.int32)
    # print(box_ind)
    # print(type(box_ind[0]))
    # print(box_ind[0].shape)
    # print(box_ind[0])


    crop_size = np.array([IMAGE_SIZE, IMAGE_SIZE], dtype = np.int32)
    # print(crop_size)

    X_scale_data = []


    # tf.reset_default_graph()
    X = tf.placeholder(tf.float32, shape = (1, IMAGE_SIZE, IMAGE_SIZE, 3))
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())

        for im_num, img in enumerate(X_imgs):
            batch_img = np.expand_dims(img, axis = 0)/255.
            X = tf.placeholder(tf.float32, shape = (1, img.shape[0], img.
 ↪shape[1], 3))
            tf_img = tf.image.crop_and_resize(X, np.expand_dims(boxes[im_num],␣
 ↪axis = 0),
                                                np.expand_dims(box_ind[im_num],␣
 ↪axis = 0), crop_size, extrapolation_value=0)
            scaled_imgs = sess.run(tf_img, feed_dict = {X: batch_img})
            X_scale_data.extend(scaled_imgs)

    X_scale_data = np.array(X_scale_data, dtype = np.float32)
    X_scale_data = (255.*X_scale_data).astype(np.uint8)



    return X_scale_data
```

[51]:
```
new_ims_scale = scale_new_images(new_ims)
```

[52]:
```
print(type(new_ims_scale))
print(new_ims_scale.shape)
print(new_ims_scale.dtype)
print(np.amax(new_ims_scale))
print(np.amin(new_ims_scale))
```

```
<class 'numpy.ndarray'>
(5, 32, 32, 3)
uint8
255
0
```

[53]:
```
Nr = 1
Nc = len(new_ims_scale)
fig, axs = plt.subplots(Nr, Nc, figsize=[3*6.4, 3*4.8])
for i in range(Nr):
    for j in range(Nc):
        axs[j].imshow(new_ims_scale[j].squeeze())
```



[54]:
```
sign_names = [(12,'Priority road'), (13,'Yield'), (17,'No entry'),␣
 ↪(40,'Roundabout mandatory'), (29,'Bicycles crossing')]
print(sign_names)
```

```
[(12, 'Priority road'), (13, 'Yield'), (17, 'No entry'), (40, 'Roundabout
mandatory'), (29, 'Bicycles crossing')]
```

[55]:
```
new_ims_labels = np.array([29, 17, 12, 40, 13], dtype = np.uint8)
print(type(new_ims_labels))
print(new_ims_labels.shape)
print(new_ims_labels.dtype)
print(new_ims_labels)
```

```
<class 'numpy.ndarray'>
(5,)
uint8
[29 17 12 40 13]
```

1.6.2 Predict the Sign Type for Each Image

[56]:
```
### 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␣
 ↪used earlier.
### Feel free to use as many code cells as needed.
new_ims_scale = np.mean(new_ims_scale, axis=3, keepdims=True)
```

[57]:
```
print(type(new_ims_scale))
print(new_ims_scale.shape)
print(new_ims_scale.dtype)
print(np.amax(new_ims_scale))
print(np.amin(new_ims_scale))
```

```
<class 'numpy.ndarray'>
(5, 32, 32, 1)
float64
255.0
2.6666666666666665
```

[58]:
```
print(np.mean(new_ims_scale))
print(np.amin(new_ims_scale))
print(np.amax(new_ims_scale))
```

```
136.68033854166666
2.6666666666666665
255.0
```

[59]:
```
new_ims_scale = (new_ims_scale - 128)/128
```

[60]:
```
print(np.mean(new_ims_scale))
print(np.amin(new_ims_scale))
print(np.amax(new_ims_scale))
```

```
0.0678151448567708
-0.9791666666666666
0.9921875
```

[61]:
```
softmax_layer = tf.nn.softmax(logits)
predict = tf.argmax(softmax_layer, 1)

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

    new_im_predict = sess.run(predict, feed_dict={x: new_ims_scale, keep_prob:␣
 ↪1.0})

    for i in range(len(new_im_predict)):
        print("Image number: {}, True label: {}, Predicted label: {}".format(i,␣
 ↪new_ims_labels[i], new_im_predict[i]))
```

```
INFO:tensorflow:Restoring parameters from .\lenet
Image number: 0, True label: 29, Predicted label: 12
Image number: 1, True label: 17, Predicted label: 17
Image number: 2, True label: 12, Predicted label: 12
Image number: 3, True label: 40, Predicted label: 11
Image number: 4, True label: 13, Predicted label: 13
```

1.6.3 Analyze Performance

[62]:
```
### Calculate the accuracy for these 5 new images.
### For example, if the model predicted 1 out of 5 signs correctly, it's 20%␣
 ↪accurate on these new images.
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

    new_im_accuracy = evaluate(new_ims_scale, new_ims_labels)
    print("New Image Accuracy = {:.3f}".format(new_im_accuracy))
```

```
INFO:tensorflow:Restoring parameters from .\lenet
New Image Accuracy = 0.600
```

1.6.4 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` 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. `tf.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,  0.07893497,
         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.

[63]:
```
### Print out the top five softmax probabilities for the predictions on the␣
↪German traffic sign images found on the web.
### Feel free to use as many code cells as needed.
top_five = tf.nn.top_k(softmax_layer, k=5)

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

    new_im_vals, new_im_inds = sess.run(top_five, feed_dict={x: new_ims_scale,␣
↪keep_prob: 1.0})
```

```
    for i in range(len(new_im_vals)):
        print('Image {}:'.format(i))
        print("{:<12s} {:<8d} {:<8d} {:<8d} {:<8d} {:<8d}".format("k:", 1, 2,␣
↪3, 4, 5))
        print("{:<12s} {:<8.3f} {:<8.3f} {:<8.3f} {:<8.3f} {:<8.3f}".
↪format("Class ID:",new_im_inds[i,0],new_im_inds[i,1],
                                                                       ␣
↪new_im_inds[i,2],new_im_inds[i,3],new_im_inds[i,4]))
        print("{:<12s} {:<8.3f} {:<8.3f} {:<8.3f} {:<8.3f} {:<8.3f}".
↪format("Probability:",new_im_vals[i,0],new_im_vals[i,1],
                                                                       ␣
↪new_im_vals[i,2],new_im_vals[i,3],new_im_vals[i,4]))
        print()
```

```
INFO:tensorflow:Restoring parameters from .\lenet
Image 0:
k:           1        2        3        4        5
Class ID:    12.000   35.000   34.000   11.000   16.000
Probability: 0.687    0.219    0.088    0.006    0.000

Image 1:
k:           1        2        3        4        5
Class ID:    17.000   0.000    1.000    2.000    3.000
Probability: 1.000    0.000    0.000    0.000    0.000

Image 2:
k:           1        2        3        4        5
Class ID:    12.000   17.000   40.000   9.000    0.000
Probability: 1.000    0.000    0.000    0.000    0.000

Image 3:
k:           1        2        3        4        5
Class ID:    11.000   34.000   3.000    28.000   35.000
Probability: 0.398    0.160    0.114    0.078    0.056

Image 4:
k:           1        2        3        4        5
Class ID:    13.000   0.000    1.000    2.000    3.000
Probability: 1.000    0.000    0.000    0.000    0.000
```

1.6.5 Project Writeup

Once you have completed the code implementation, document your results in a project writeup using this template 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 ",”**File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

1.7 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 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 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)

[ ]:
```
### 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␣
 ↪procedure that represents the calculated state of a specific weight layer
# activation_min/max: can be used to view the activation contrast in more␣
 ↪detail, by default matplot sets min and max to the actual min and max values␣
 ↪of the output
# plt_num: used to plot out multiple different weight feature map sets on the␣
 ↪same block, just extend the plt number for each new feature map entry

def outputFeatureMap(image_input, tf_activation, activation_min=-1,␣
 ↪activation_max=-1 ,plt_num=1):
    # Here make sure to preprocess your image_input in a way your network␣
↪expects
    # with size, normalization, ect if needed
    # image_input =
    # Note: x should be the same name as your network's tensorflow data␣
↪placeholder 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␣
↪number
        if activation_min != -1 & activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest",␣
↪vmin =activation_min, vmax=activation_max, cmap="gray")
        elif activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest",␣
↪vmax=activation_max, cmap="gray")
        elif activation_min !=-1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest",␣
↪vmin=activation_min, cmap="gray")
        else:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest",␣
↪cmap="gray")
```