

# 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 \(https://github.com/udacity/CarND-Traffic-Sign-Classifier-Project/blob/master/writeup\\_template.md\)](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 \(https://review.udacity.com/#!/rubrics/481/view\)](https://review.udacity.com/#!/rubrics/481/view) for this project.

The [rubric \(https://review.udacity.com/#!/rubrics/481/view\)](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 lpython 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 = 'train.p'
validation_file='valid.p'
testing_file = '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]: ### BASIC SUMMARY OF THE DATA SET
#
import pandas as pd
import numpy as np

# TODO: Number of training examples
n_train = len(X_train)
n_y_train = len(y_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(np.unique(y_train))
n_test_classes = len(np.unique(y_test))

print("Number of training examples =", n_train)
print("Number of y training labels =", n_y_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 y training labels = 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/) (<http://matplotlib.org/>) [examples](http://matplotlib.org/examples/index.html) (<http://matplotlib.org/examples/index.html>) and [gallery](http://matplotlib.org/gallery.html) (<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]: ### Data exploration visualization code
### Visualizing the image data with the corresponding labels and classIDs
import random
import matplotlib.pyplot as plt
# Visualizations will be shown in the notebook.
%matplotlib inline

# Be able to import the csv lookup table data
import csv

def plot_figures(figures, nrows = 1, ncols=1, labels=None, classID=None):
    fig, axs = plt.subplots(ncols=ncols, nrows=nrows, figsize=(12, 14))
    axs = axs.ravel()
    for index, title in zip(range(len(figures)), figures):
        axs[index].imshow(figures[title], plt.gray())
        if(labels != None):
            axs[index].set_title(labels[index])
        else:
            axs[index].set_title(title)

        axs[index].set_title(classID[index], loc='right')

        axs[index].set_axis_off()

    plt.tight_layout()

name_values = np.genfromtxt('signnames.csv', skip_header=1, dtype=[('myint','i8')])

number_to_stop = 8
figures = {}
labels = {}
classID = {}
for i in range(number_to_stop):
    index = random.randint(0, n_train-1)
    labels[i] = name_values[y_train[index]][1].decode('ascii')
    # print(name_values[y_train[index]][1].decode('ascii'))
    figures[i] = X_train[index]
    classID[i] = name_values[y_train[index]][0].decode('ascii')

plot_figures(figures, 4, 2, labels, classID)

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

#plt.figure(figsize=(5,5))
#plt.imshow(image)
#print(y_train[index])

```

Wild animals crossing 31



Double curve 21



Stop 14



General caution 18



No passing for vehicles over 3.5 metric tons



Vehicles over 3.5 metric tons prohibited



Speed limit (100km/h) 7



Priority road 12



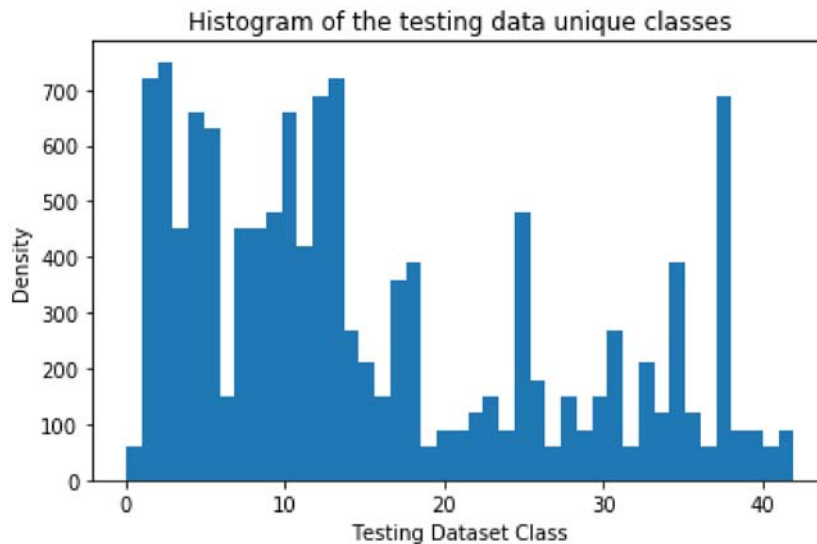
```
In [4]: ### PLOT THE TRAINING SET DATA IN A HISTOGRAM
fig,ax = plt.subplots()
#histogram of the data
n, bins, patches = ax.hist(y_train,n_classes) #normed=1
ax.set_xlabel('Training Dataset Class')
ax.set_ylabel('Density')
ax.set_title('Histogram of the training data unique classes')
#
fig.tight_layout()
plt.show()

fig.savefig('./examples/plot1.png')
```



```
In [5]: ### PLOT THE TESTING SET DATA IN A HISTOGRAM
fig,ax = plt.subplots()
#histogram of the data
n, bins, patches = ax.hist(y_test,n_test_classes) #normed=1
ax.set_xlabel('Testing Dataset Class')
ax.set_ylabel('Density')
ax.set_title('Histogram of the testing data unique classes')
#
fig.tight_layout()
plt.show()

fig.savefig('./examples/plot4.png')
```

**NOTE:**

The testing data has very similar deficiencies as the training data. This might make it harder to have statistically significant results when the verification step is so lightly populated.

```
In [6]: # FUNCTIONS

# import packages
from scipy import ndimage
import cv2

# from numpy import newaxis HOLD

# Normalize the training data and the validation data to be between -1 and 1 (-1
def normalize(image):
    image = (image - 127.5) / 127.5
    return image

def resizeImage(image):
    return cv2.resize(image,(32,32),interpolation = cv2.INTER_AREA)

def cropImage(input):
    roi = input[7:25,7:25]
    return resizeImage(roi)

def sharpen(image):
    img = cv2.GaussianBlur(image,(9,9), 10);
    img1 = cv2.addWeighted(image, 2, img, -1, 0);
    return img1

def Transform(image,rot):
    return resizeImage(ndimage.rotate(image, rot))

# PREPROCESSING PIPELINE
def pipeline(input):
    gray = cv2.cvtColor(input,cv2.COLOR_BGR2GRAY)
    norm = normalize(gray)
    crop = cropImage(norm)
    sharp = sharpen(crop)
    return sharp
```



In [7]: *# Display random 5 images*

```
index = 4 # Choose a Label to draw
```

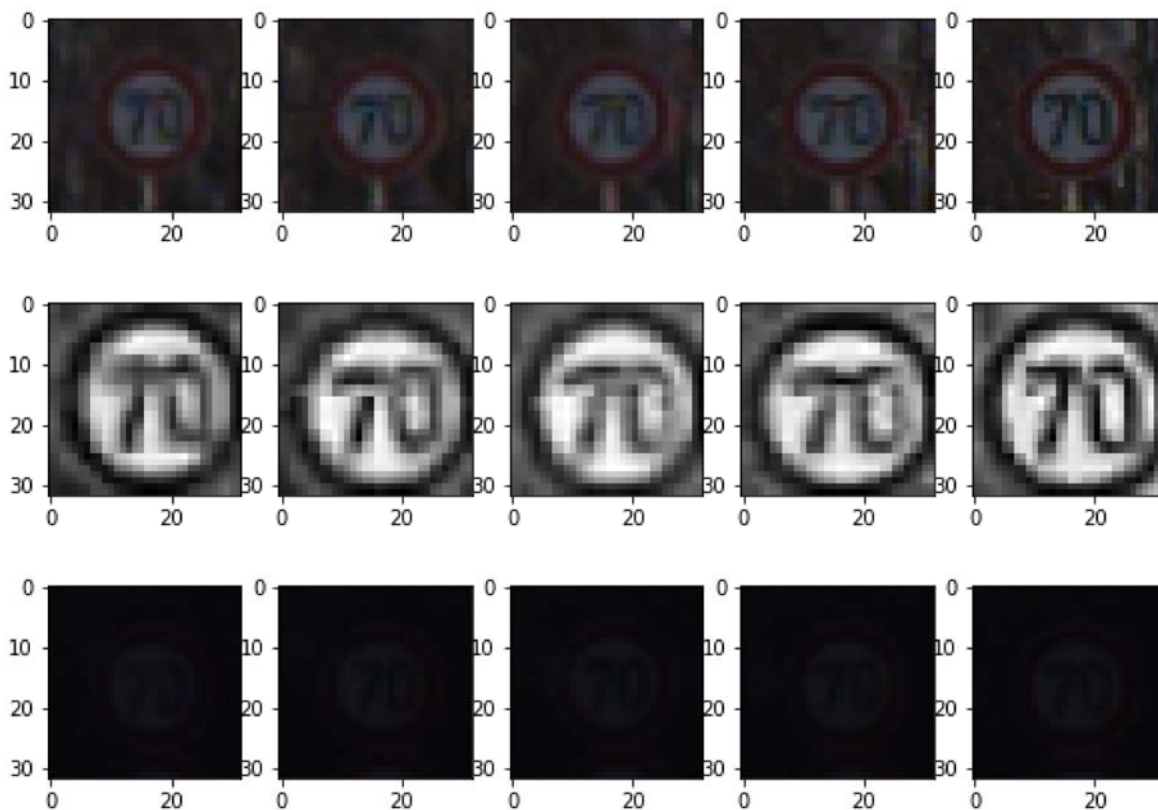
```
disp = [X_train[i] for i in range(len(y_train)) if (y_train[i]==index)]
valid = [X_valid[i] for i in range(len(y_valid)) if (y_valid[i]==index)]
```

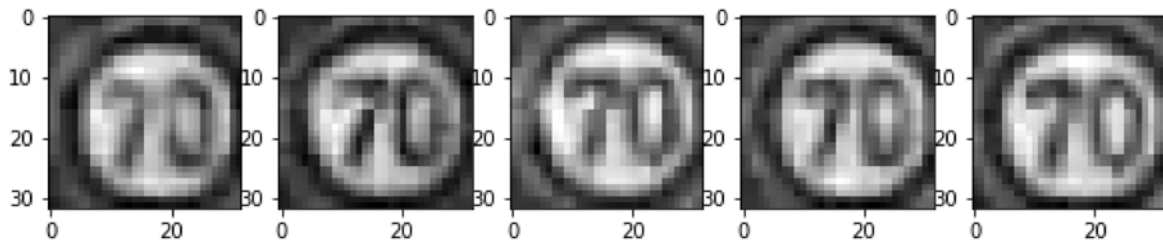
```
fig = plt.figure(figsize=(10,10))
for j in range(0,5):
    plt.subplot(1,5,j+1)
    plt.imshow(disp[j])
```

```
fig1 = plt.figure(figsize=(10,10))
for j in range(0,5):
    plt.subplot(1,5,j+1)
    plt.imshow(pipeline(disp[j]))
```

```
fig2 = plt.figure(figsize=(10,10))
for j in range(0,5):
    plt.subplot(1,5,j+1)
    plt.imshow(valid[j])
```

```
fig3 = plt.figure(figsize=(10,10))
for j in range(0,5):
    plt.subplot(1,5,j+1)
    plt.imshow(pipeline(valid[j]))
```





## 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) (<http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset>).

The LeNet-5 implementation shown in the [classroom](https://classroom.udacity.com/nanodegrees/nd013/parts/6bf77062-5703-404e-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/6bf77062-5703-404e-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) (<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,  $(\text{pixel} - 128) / 128$  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 [8]: ### PREPROCESS ALL DATA
# functions for preprocessing are above

# set up gray scaled placeholder images
X_train_gray = np.zeros((len(X_train),32,32))
X_valid_gray = np.zeros((len(X_valid),32,32))
X_test_gray = np.zeros((len(X_test),32,32))

# populate the placeholders
for i in range(len(X_train)):
    X_train_gray[i] = pipeline(X_train[i])

for i in range(len(X_valid)):
    X_valid_gray[i] = pipeline(X_valid[i])

for i in range(len(X_test)):
    X_test_gray[i] = pipeline(X_test[i])

### PREPROCESSING PIPELINE FUNCTION (steps)
# 1. Converted input RGB image to grayscale in order to reduce the affects of
# 2. Normalize image range -1.0 to 1.0 to properly condition the data for Gra
# 3. Crop image to remove unnecessary pixels to avoid confusing the training
# 4. Sharpen the cropped image in order to create more distinct features

# Normalization of the training data and the validation data orginally was between
# (-1 and .992) from single channel data 0 to 255 using '128'
# changed to 127.5 to further properly condition the data
#     pixel = (pixel - 128) / 128
# pixel = (pixel - 128.) / 128.
#     pixel = (pixel - 127.5) / 127.5

plt.imshow(image)
```

```
In [9]: # Add more data for Low sample Lables
# For each data label, add 6 more images per image - *JUST FOR TRAINING DATA*
index_count = np.bincount(y_train)
indexes = [i for i in range(len(index_count)) if index_count[i] < 1200]
print("Labels that require more data:", indexes)
angles = [-20,-15,-10,10,15,20]

# Collect all class labels that require data generation
X_data, y_data = zip(*[(X_train_gray[i], y_train[i]) for i in range(0,len(y_train)) if i in indexes])

# Compute rotation images for X_data
X_rotate, y_rotate = zip(*[(Transform(X_data[i],rotation), y_data[i]) for rotation in angles for i in range(0,len(X_data)) if i in indexes])

# Append the newly generated data to the original data
X_train_new = np.append(X_train_gray,X_rotate,axis=0)
y_train = np.append(y_train,y_rotate,axis=0)

# Print stuff
print("Additional data samples generated: ", np.shape(X_rotate[0]))
print("New size - X_train: ", np.shape(X_train_new))
print("New size - y_train: ", np.shape(y_train))
```

```
Labels that require more data: [0, 6, 11, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 39, 40, 41, 42]
Additional data samples generated: (32, 32)
New size - X_train: (115433, 32, 32)
New size - y_train: (115433,)
```

```
In [10]: # SHUFFLE AND RESHAPE THE DATA

# import packages
from sklearn.utils import shuffle
from numpy import newaxis

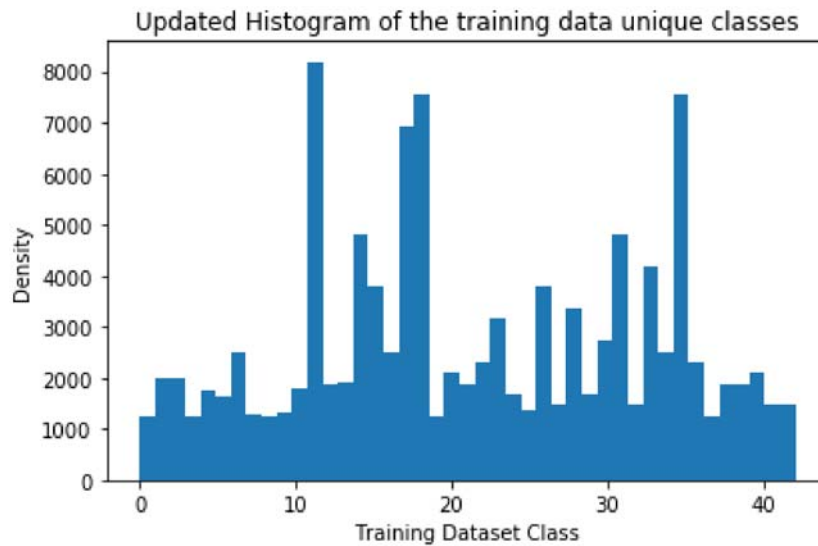
# First conv layer in tensorflow requires data to be in (none,32,32,1) structure
X_train_new = X_train_new[..., newaxis]
X_valid = X_valid_gray[..., newaxis]
X_test = X_test_gray[..., newaxis]

# Convert to Grayscale to make it easier to use the MNIST classifier
#gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# Shuffle the training data
X_train_new, y_train = shuffle(X_train_new, y_train)
# Shuffle the validation data
X_valid, y_valid = shuffle(X_valid, y_valid)
# Shuffle the test data
X_test, y_test = shuffle(X_test, y_test)
```

```
In [11]: # PLOT UPDATED HISTOGRAM
fig,ax = plt.subplots()
#histogram of the data
n, bins, patches = ax.hist(y_train,n_classes) #,normed=1
ax.set_xlabel('Training Dataset Class')
ax.set_ylabel('Density')
ax.set_title('Updated Histogram of the training data unique classes')
#
fig.tight_layout()
plt.show()

fig.savefig('./examples/plot2.png')
```



## Model Architecture

```

In [12]: ### MODEL ARCHITECTURE
# My model architecture is heavily based upon the LeNet example presented for the
# modifications that I made were to add dropout after the first and second RELU a

# My final model consisted of the following layers:

# Input           32x32x1 Grayscale image
# Convolution     5x5x1 filter, stride 1x1, VALID padding, Output 28x28x6
# RELU
# Dropout 1       Used for training ONLY
# Pooling         2x2 kernel, 2x2 stride, VALID padding, Output 14x14x6
# Convolution     5x5x6 filter, 1x1 stride, VALID padding, Output 10x10x16
# RELU
# Dropout 2       Used for training ONLY
# Pooling         2x2 kernel, 2x2 stride, Output 5x5x16
# Flatten         Output 400
# Fully connected Output 120
# Fully connected Output 84
# RELU
# Fully connected Output 43

# experiment with number of layers, number of filters, dropout

# Model Setup
import tensorflow as tf
from tensorflow.contrib.layers import flatten

def LeNet(x):
    # Arguments used for tf.truncated_normal, randomly defines variables for the
    mu = 0
    sigma = 0.1

    # Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6. Changed from 3
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), mean = mu, stddev=sigma))
    conv1_b = tf.Variable(tf.zeros(6))
    conv1 = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') + conv1_b

    # Activation Function
    conv1 = tf.nn.relu(conv1)

    # Dropout ADDED
    conv1 = tf.nn.dropout(conv1, keep_prob)

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

    # Layer 2: Convolutional. Output = 10x10x16.
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean = mu, stddev=sigma))
    conv2_b = tf.Variable(tf.zeros(16))
    conv2 = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], padding='VALID') + conv2_b

    # Activation Function
    conv2 = tf.nn.relu(conv2)

```

```

# Dropout ADDED
conv2 = tf.nn.dropout(conv2, keep_prob)

# Pooling. Input = 10x10x16. Output = 5x5x16.
conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], paddi

# Flatten. Input = 5x5x16. Output = 400.
fc0 = flatten(conv2)

# Layer 3: Fully Connected. Input = 400. Output = 120.
fc1_W = tf.Variable(tf.truncated_normal(shape=(400, 120), mean = mu, stddev =
fc1_b = tf.Variable(tf.zeros(120))
fc1 = tf.matmul(fc0, fc1_W) + fc1_b

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

# ADDED 1st dropout per reference model
#drop1 = tf.nn.dropout(fc1, keep_prob)

# Layer 4: Fully Connected. Input = 120. Output = 84.
fc2_W = tf.Variable(tf.truncated_normal(shape=(120, 84), mean = mu, stddev =
fc2_b = tf.Variable(tf.zeros(84))
# subbed drop1 for fc1
fc2 = tf.matmul(fc1, fc2_W) + fc2_b

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

# ADDED 2nd dropout per reference model
#drop2 = tf.nn.dropout(fc2, keep_prob)

# Layer 5: Fully Connected. Input = 84. Output = 43.
fc3_W = tf.Variable(tf.truncated_normal(shape=(84, 43), mean = mu, stddev =
fc3_b = tf.Variable(tf.zeros(43))

# subbed drop2 for fc2
logits = tf.matmul(fc2, fc3_W) + fc3_b

return logits

```

## 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 [13]: ### 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.  
  
from sklearn.utils import shuffle  
  
# ADDED  
keep_prob = tf.placeholder(tf.float32)  
  
x = tf.placeholder(tf.float32, (None, 32, 32, 1)) # Changed to gray from RGB (3)  
y = tf.placeholder(tf.int32, (None))  
one_hot_y = tf.one_hot(y, 43)  
  
# PARAMETERS  
EPOCHS = 20 # EPOCHS default 10  
BATCH_SIZE = 128 # 128 default  
  
rate = 0.001 # LEARNING RATE default .001  
  
#keep_prob = .7 # 0.9 default USING "feed_dict" FOR EACH evaluate
```

```
In [14]: # TRAINING PIPELINE  
  
logits = LeNet(x)  
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y, logits=  
loss_operation = tf.reduce_mean(cross_entropy)  
optimizer = tf.train.AdamOptimizer(learning_rate = rate)  
training_operation = optimizer.minimize(loss_operation)
```



In [15]: *# MODEL EVALUATION*

```
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, sess):
    num_examples = X_data.shape[0]
    total_accuracy = 0
    total_loss = 0 # ADDED
    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]
        # OBTAIN loss and accuracy in same TF session - need to pass keep_prob?
        loss, accuracy = sess.run([loss_operation, accuracy_operation], feed_dict={x: batch_x, y: batch_y})
        total_accuracy += (accuracy * batch_x.shape[0])
        total_loss += (loss * batch_x.shape[0])
    return total_loss / num_examples, total_accuracy / num_examples

# Return prediction and top_k values ADDED
def predict(X_data, y_data):
    predictions = sess.run(tf.argmax(logits, 1), feed_dict={x: X_data, y: y_data, keep_prob: 1.0})
    softmax = tf.nn.softmax(logits)
    top_k = sess.run(tf.nn.top_k(softmax, k=5), feed_dict={x: X_data, y: y_data, keep_prob: 1.0})
    return predictions, top_k
```

```

In [16]: # TRAIN THE MODEL

# ADDED
train_loss_history = []
validation_loss_history = []
train_accuracy_history = []
validation_accuracy_history = []

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_new, y_train = shuffle(X_train_new, y_train)    # CHANGED X_train
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train_new[offset:end], y_train[offset:end]    # CH
            # need to ADD keep_prob: 0.5
            sess.run(training_operation, feed_dict={x: batch_x, y: batch_y, keep_

        # ADDED
        train_loss, train_accuracy = evaluate(X_train_new, y_train, sess)    # CHANG
        train_loss_history.append(train_loss)
        train_accuracy_history.append(train_accuracy)
        validation_loss, validation_accuracy = evaluate(X_valid, y_valid, sess)
        validation_loss_history.append(validation_loss)
        validation_accuracy_history.append(validation_accuracy)

        print("EPOCH {} ...".format(i+1))
        print("Training Accuracy = {:.3f}".format(train_accuracy))    # ADDED
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()

        # Evaluate on the test data
        #test_loss, test_acc = evaluate(X_test, y_test, sess)
        #print("Test loss = {:.3f}".format(test_loss))
        #print("Test accuracy = {:.3f}".format(test_acc))

    #loss_plot = plt.subplot(2,1,1)
    #loss_plot.set_title('Loss')
    #loss_plot.plot(training_loss_history, 'r', Label='Training Loss')
    #loss_plot.plot(validation_loss_history, 'b', Label='Validation Loss')
    #loss_plot.set_xlim([0, EPOCHS])
    #loss_plot.legend(loc=4)
    #acc_plot = plt.subplot(2,1,2)
    #acc_plot.set_title('Accuracy')
    #acc_plot.plot(training_accuracy_history, 'r', Label='Training Accuracy')
    #acc_plot.plot(validation_accuracy_history, 'b', Label='Validation Accuracy')
    #acc_plot.set_ylim([0, 1.0])
    #acc_plot.set_xlim([0, EPOCHS])
    #acc_plot.legend(loc=4)
    #plt.tight_layout()
    #plt.show()

```

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

Training...

EPOCH 1 ...

Training Accuracy = 0.818,  
Validation Accuracy = 0.772

EPOCH 2 ...

Training Accuracy = 0.909,  
Validation Accuracy = 0.864

EPOCH 3 ...

Training Accuracy = 0.935,  
Validation Accuracy = 0.884

EPOCH 4 ...

Training Accuracy = 0.955,  
Validation Accuracy = 0.911

EPOCH 5 ...

Training Accuracy = 0.963,  
Validation Accuracy = 0.928

EPOCH 6 ...

Training Accuracy = 0.971,  
Validation Accuracy = 0.929

EPOCH 7 ...

Training Accuracy = 0.971,  
Validation Accuracy = 0.923

EPOCH 8 ...

Training Accuracy = 0.982,  
Validation Accuracy = 0.935

EPOCH 9 ...

Training Accuracy = 0.979,  
Validation Accuracy = 0.945

EPOCH 10 ...

Training Accuracy = 0.982,  
Validation Accuracy = 0.942

EPOCH 11 ...

Training Accuracy = 0.981,  
Validation Accuracy = 0.949

EPOCH 12 ...

Training Accuracy = 0.985,  
Validation Accuracy = 0.946

EPOCH 13 ...

Training Accuracy = 0.984,  
Validation Accuracy = 0.948

EPOCH 14 ...  
Training Accuracy = 0.990,  
Validation Accuracy = 0.949

EPOCH 15 ...  
Training Accuracy = 0.983,  
Validation Accuracy = 0.939

EPOCH 16 ...  
Training Accuracy = 0.989,  
Validation Accuracy = 0.946

EPOCH 17 ...  
Training Accuracy = 0.990,  
Validation Accuracy = 0.942

EPOCH 18 ...  
Training Accuracy = 0.988,  
Validation Accuracy = 0.944

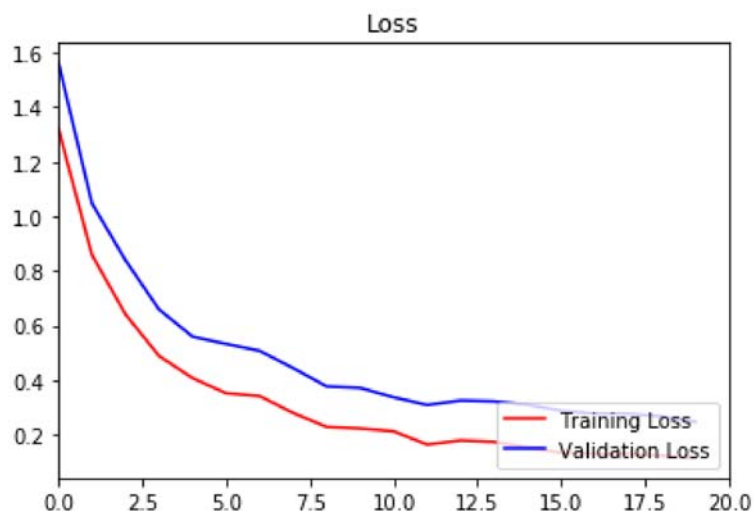
EPOCH 19 ...  
Training Accuracy = 0.993,  
Validation Accuracy = 0.947

EPOCH 20 ...  
Training Accuracy = 0.992,  
Validation Accuracy = 0.951

Model saved

```
In [17]: # LOSS PLOTS
loss_plot = plt.subplot(1,1,1)
loss_plot.set_title('Loss')
loss_plot.plot(train_loss_history, 'r', label='Training Loss')
loss_plot.plot(validation_loss_history, 'b', label='Validation Loss')
loss_plot.set_xlim([0, EPOCHS])
loss_plot.legend(loc=4)

plt.savefig('./examples/plot3.png')
```



```
In [18]: # EVALUATE ACCURACY on test data
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

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

Test Accuracy = 0.934

---

## 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 [19]: ### Load the images and plot them here.
### Feel free to use as many code cells as needed.

import glob
#import matplotlib.image as mpimg

# Test on new images from the internet
X_new_test = []
for filename in glob.glob('./examples/test_*.png'):
    X_new_test.append(cv2.imread(filename))

# Make sure the input images are the same size as training set
X_new_test = [resizeImage(image) for image in X_new_test]

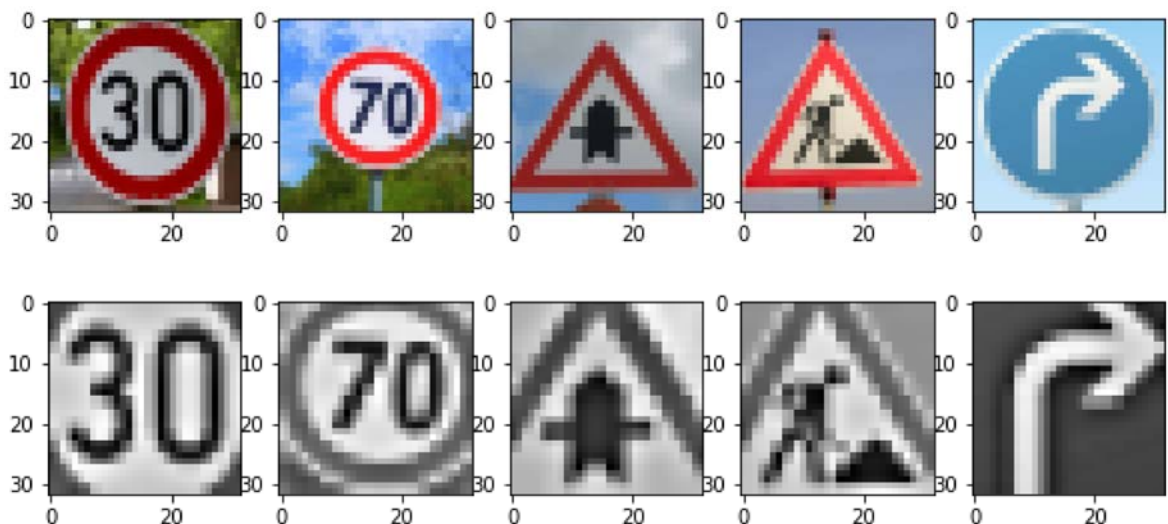
fig1 = plt.figure(figsize=(10,10))
for j in range(0,5):
    plt.subplot(1,5,j+1)
    plt.imshow(cv2.cvtColor(X_new_test[j],cv2.COLOR_BGR2RGB))

fig2 = plt.figure(figsize=(10,10))
for j in range(0,5):
    plt.subplot(1,5,j+1)
    plt.imshow(pipeline(X_new_test[j]))

X_test_new = np.zeros((len(X_new_test),32,32))
# Create labels
for i in range(0,5):
    X_test_new[i] = pipeline(X_new_test[i])

X_new_test = X_test_new[..., newaxis]
y_new_test = [1,4,11,25,33]

```



**Predict the Sign Type for Each Image**

In [20]: *### Run the predictions here and use the model to output the prediction for each  
 ### Make sure to pre-process the images with the same pre-processing pipeline use  
 ### Feel free to use as many code cells as needed.*

```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    predictions, topk = predict(X_new_test, y_new_test)
    print("Labels: ", y_new_test)
    print("Predictions: ", predictions)
```

Labels: [1, 4, 11, 25, 33]  
 Predictions: [ 1 4 11 25 33]

## Analyze Performance

In [21]: *### Calculate the accuracy for these 5 new images.  
 ### For example, if the model predicted 1 out of 5 signs correctly, it's 20% accurate*

```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    loss, accuracy = evaluate(X_new_test, y_new_test, sess)
    print("Accuracy = {:.3f}".format(accuracy))
    # print()
    # print(topk)
```

Accuracy = 1.000

## 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](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 corresponding 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,  0.078
93497,
               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 [22]: *### Print out the top five softmax probabilities for the predictions on the German traffic sign dataset. Feel free to use as many code cells as needed.*

```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    predictions,topk = predict(X_new_test, y_new_test)
    #print("Accuracy = {:.3f}".format(accuracy))
    #float_formatter = lambda x: "%.3f" % x
    np.set_printoptions(precision=4, suppress=True)
    print(topk)
```

```
TopKV2(values=array([[ 0.9983,  0.001 ,  0.0002,  0.0002,  0.0002],
 [ 0.5317,  0.3383,  0.1032,  0.0111,  0.0071],
 [ 0.9965,  0.0028,  0.0004,  0.0003,  0.    ],
 [ 1.    ,  0.    ,  0.    ,  0.    ,  0.    ],
 [ 0.952 ,  0.0166,  0.0056,  0.0051,  0.0047]]), dtype=float32), indices=
array([[ 1, 35,  5,  2, 11],
 [ 4,  0, 18, 23, 28],
 [11, 30, 40, 42, 21],
 [25, 10, 22, 18, 26],
 [33, 26, 14, 12,  7]], dtype=int32))
```



## Project Writeup

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

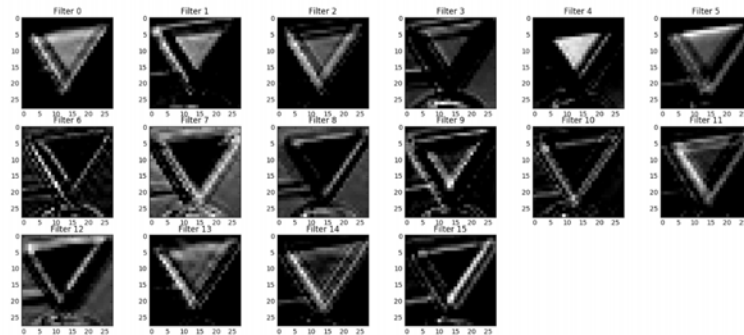
**Note:** Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to "\n", "**File -> Download as -> HTML (.html)**". 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 exercise for understanding 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 its 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 \(https://classroom.udacity.com/nanodegrees/nd013/parts/6bf77062-5703-404e-b60c-95b78b2f3f9e/modules/6df7ae49-c61c-4bb2-a23e-6527e69209ec/lessons/601ae704-1035-4287-8b11-e2c2716217ad/concepts/d4aca031-508f-4e0b-b493-e7b706120f81\)](https://classroom.udacity.com/nanodegrees/nd013/parts/6bf77062-5703-404e-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 its 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 [16]: ### 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 m
# tf_activation: should be a tf variable name used during your training procedure
# activation_min/max: can be used to view the activation contrast in more detail,
# plt_num: used to plot out multiple different weight feature map sets on the sam

def outputFeatureMap(image_input, tf_activation, activation_min=-1, activation_ma
    # 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
    # If you get an error tf_activation is not defined it may be having trouble a
    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
        plt.title('FeatureMap ' + str(featuremap)) # displays the feature map num
        if activation_min != -1 & activation_max != -1:
            plt.imshow(activation[0,:,: , featuremap], interpolation="nearest", vm
        elif activation_max != -1:
            plt.imshow(activation[0,:,: , featuremap], interpolation="nearest", vm
        elif activation_min != -1:
            plt.imshow(activation[0,:,: , featuremap], interpolation="nearest", vm
        else:
            plt.imshow(activation[0,:,: , featuremap], interpolation="nearest", cm
```