Traffic Sign Classifier

Build a Traffic Sign Recognition Project

The goals / steps of this project are the following:

- Load the data set (see below for links to the project data set)
- Explore, summarize and visualize the data set
- Design, train and test a model architecture
- Use the model to make predictions on new images
- Analyze the softmax probabilities of the new images
- Summarize the results with a written report

Summary

In this project, I started building our classifier from the LeNet convolutional neural network presented in the class material and adapted it to fit the new input requirements and output expected accuracy.

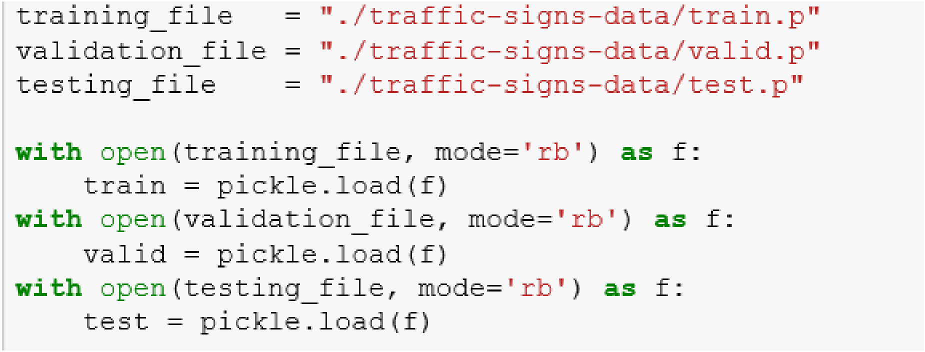
Eventually my model achieved 94.1% accuracy on the test data, and managed to properly classify 9 out of 10 signs randomly taken from the German Traffic Sign Data Set.

The code (python Jupyter notebook) can be found at the following github link:

https://github.com/mgautho/CarND-Traffic-Sign-Classifier-Project

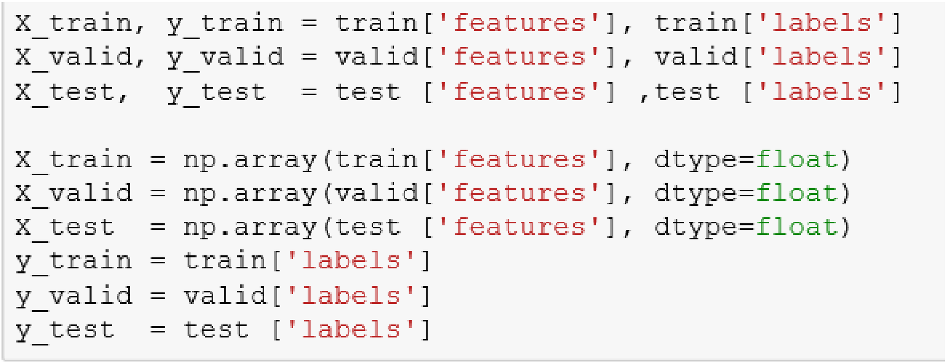
Data Set Summary & Exploration

The data set was provided as part of the project, and is loaded with the following python command:



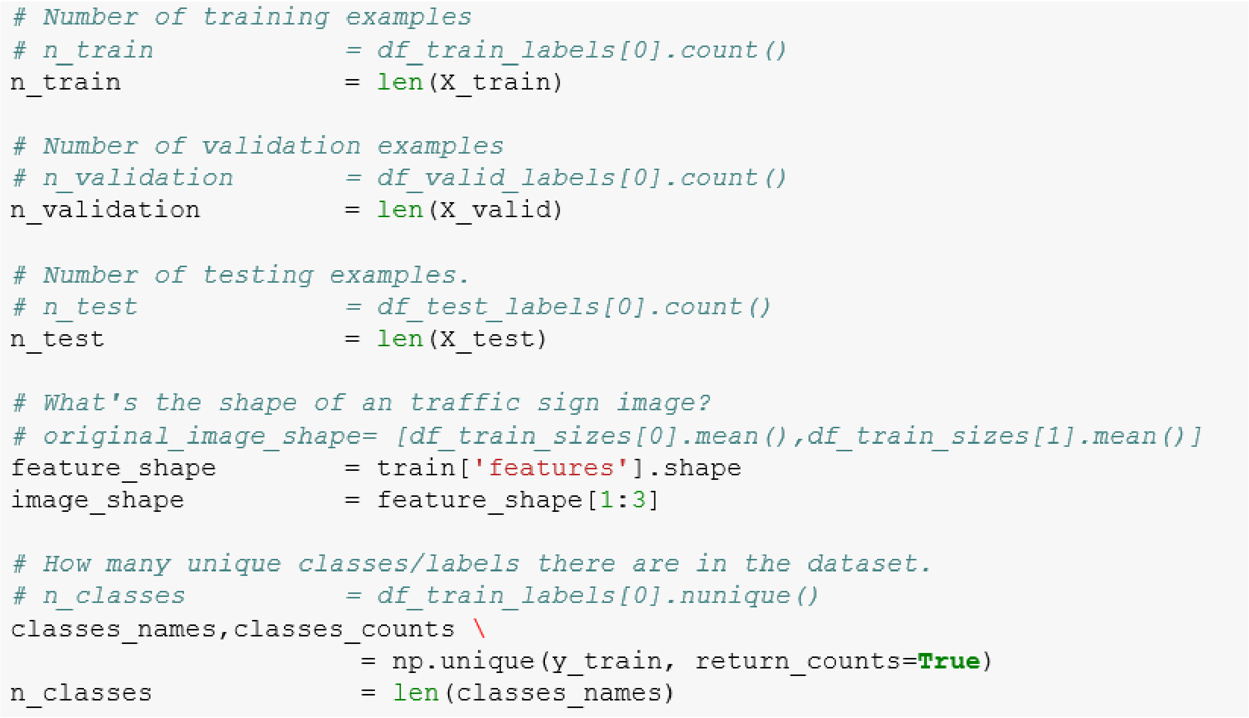
```python
training_file   = "./traffic-signs-data/train.p"
validation_file = "./traffic-signs-data/valid.p"
testing_file    = "./traffic-signs-data/test.p"

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)
```

I am then ready to populate the X,y variables for the training, validation and test data.

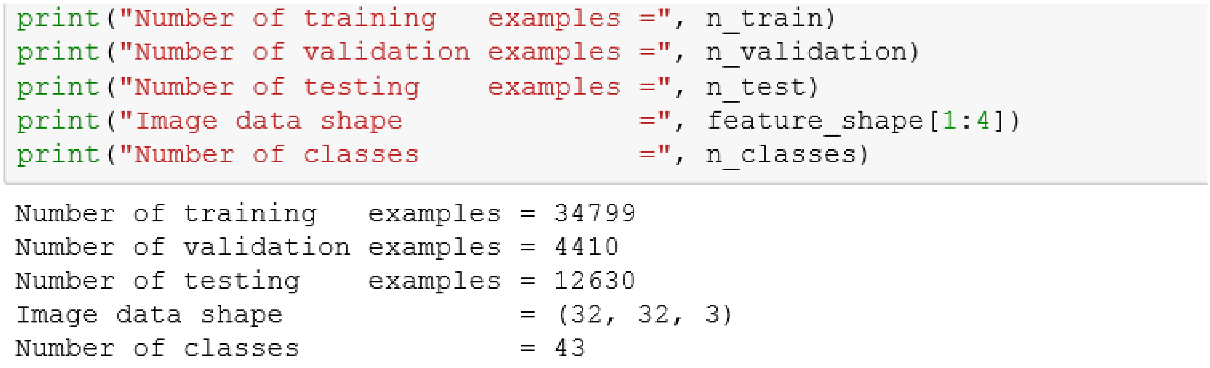


```
X_train, y_train = train['features'], train['labels']
X_valid, y_valid = valid['features'], valid['labels']
X_test,  y_test  = test ['features'] ,test ['labels']

X_train = np.array(train['features'], dtype=float)
X_valid = np.array(valid['features'], dtype=float)
X_test  = np.array(test ['features'], dtype=float)
y_train = train['labels']
y_valid = valid['labels']
y_test  = test ['labels']
```

Using built-in python functions, as well as numpy arrays method, I can extract the number of labels (or sign classes in this case) that I need to classify, the size and shape of the training features, and very importantly the class distribution in the training set.



```
# Number of training examples
# n_train              = df_train_labels[0].count()
n_train                = len(X_train)

# Number of validation examples
# n_validation         = df_valid_labels[0].count()
n_validation           = len(X_valid)

# Number of testing examples.
# n_test               = df_test_labels[0].count()
n_test                 = len(X_test)

# What's the shape of an traffic sign image?
# original_image_shape= [df_train_sizes[0].mean(),df_train_sizes[1].mean()]
feature_shape          = train['features'].shape
image_shape            = feature_shape[1:3]

# How many unique classes/labels there are in the dataset.
# n_classes            = df_train_labels[0].nunique()
classes_names,classes_counts \
                        = np.unique(y_train, return_counts=True)
n_classes               = len(classes_names)
```



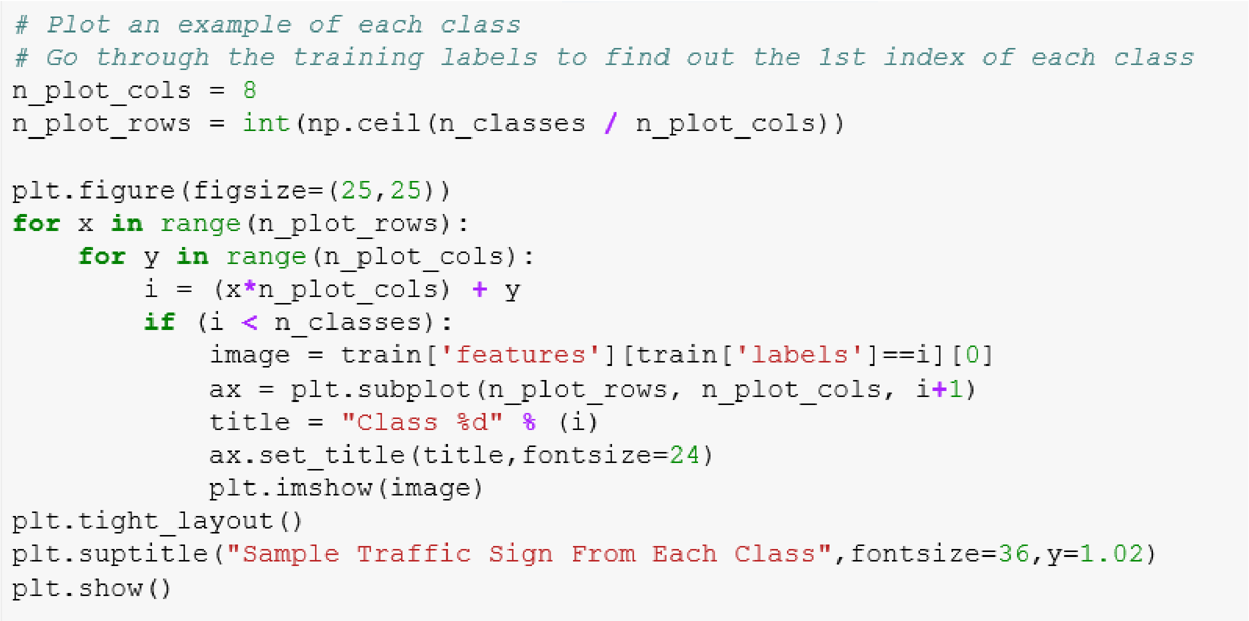
```
print("Number of training   examples =", n_train)
print("Number of validation examples =", n_validation)
print("Number of testing    examples =", n_test)
print("Image data shape              =", feature_shape[1:4])
print("Number of classes             =", n_classes)
```

```
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
```

As expected, since the features are RBG images, the shape of the images is a 3D array (32x32x3).

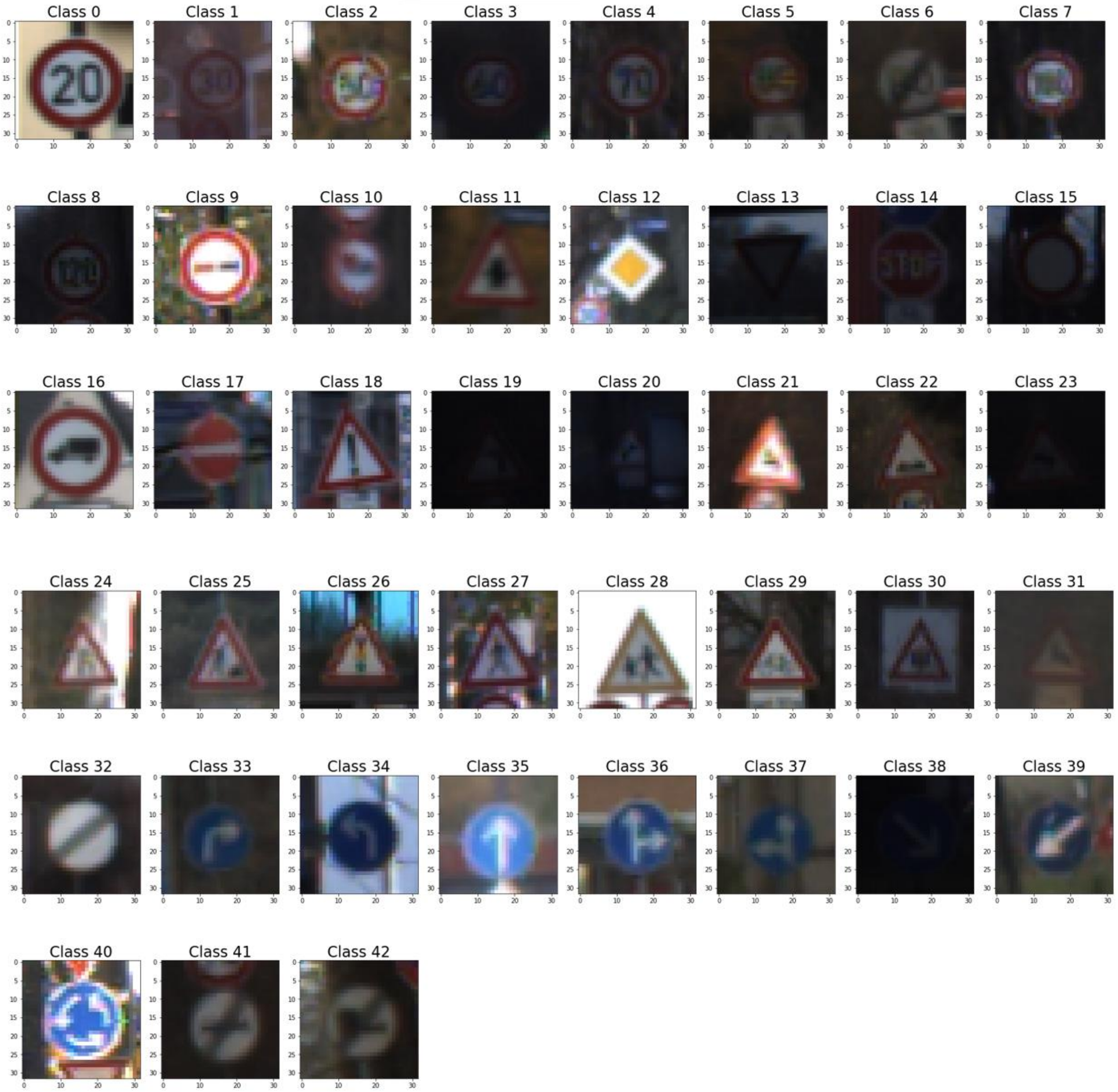
Exploratory Visualization Of The Data Set

I display, for each class, the image of 1st index in the features training set that match that class, using the following piece of code, with a subplot of 8 columns, the number of rows being automatically adjusted based on the number of classes (n_classes).

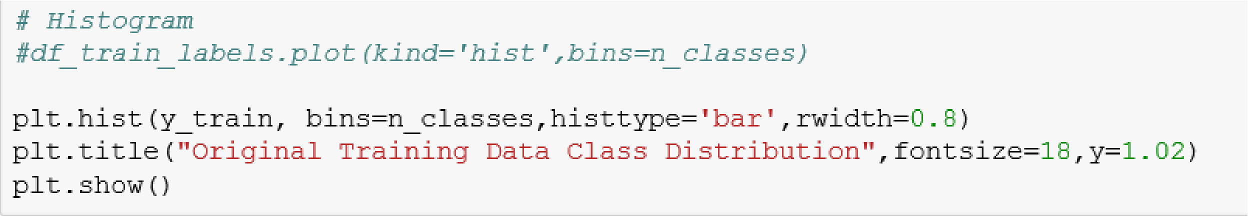


```
# Plot an example of each class
# Go through the training labels to find out the 1st index of each class
n_plot_cols = 8
n_plot_rows = int(np.ceil(n_classes / n_plot_cols))

plt.figure(figsize=(25,25))
for x in range(n_plot_rows):
    for y in range(n_plot_cols):
        i = (x*n_plot_cols) + y
        if (i < n_classes):
            image = train['features'][train['labels']==i][0]
            ax = plt.subplot(n_plot_rows, n_plot_cols, i+1)
            title = "Class %d" % (i)
            ax.set_title(title,fontsize=24)
            plt.imshow(image)
plt.tight_layout()
plt.suptitle("Sample Traffic Sign From Each Class",fontsize=36,y=1.02)
plt.show()
```

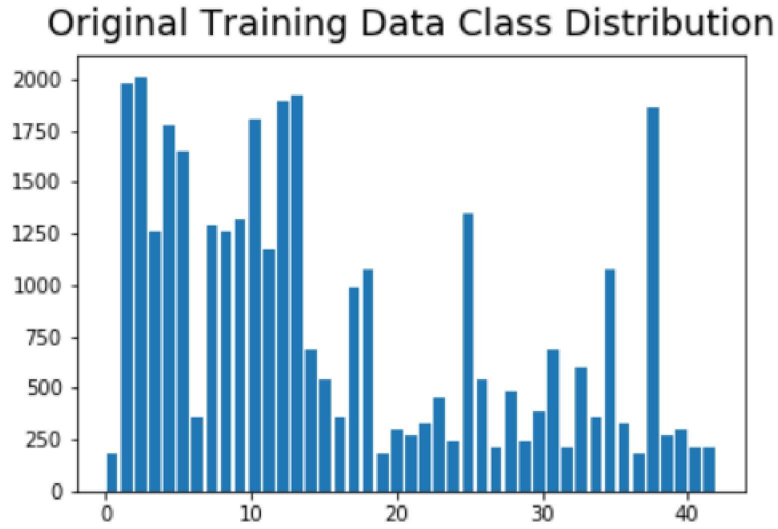
Sample Traffic Sign From Each Class



Finally, I plot a histogram of the class distribution in the features training set, using the following piece of code.



```
# Histogram
#df_train_labels.plot(kind='hist',bins=n_classes)

plt.hist(y_train, bins=n_classes,histtype='bar',rwidth=0.8)
plt.title("Original Training Data Class Distribution",fontsize=18,y=1.02)
plt.show()
```



Data Augmentation

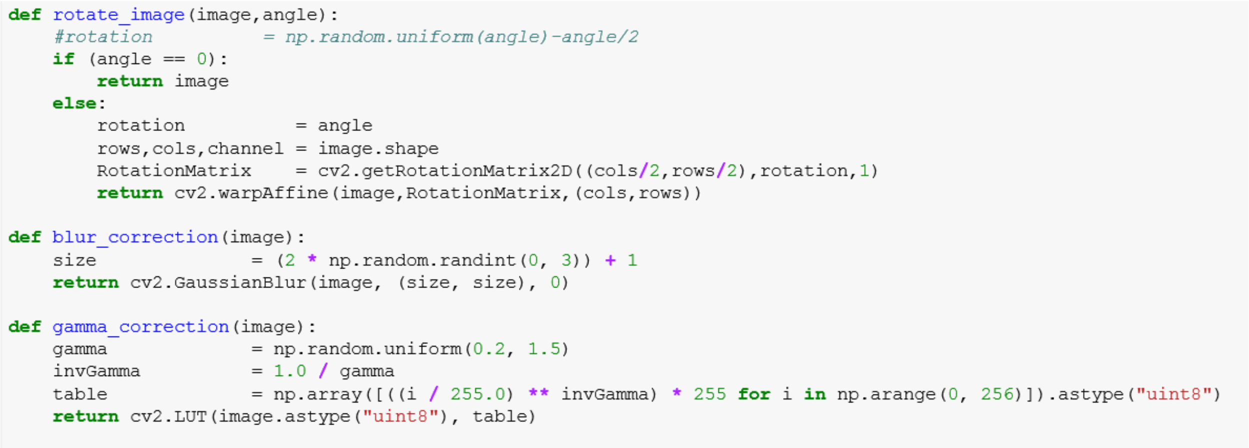
The class distribution in the features training set shows that many classes are under-represented, with the smallest having less than 200 samples while the largest one has more than 2000 samples.

I decided to create additional images using the following transformations:

- Rotation (-5,15,5,15)
- Blurring correction
- Gamma correction

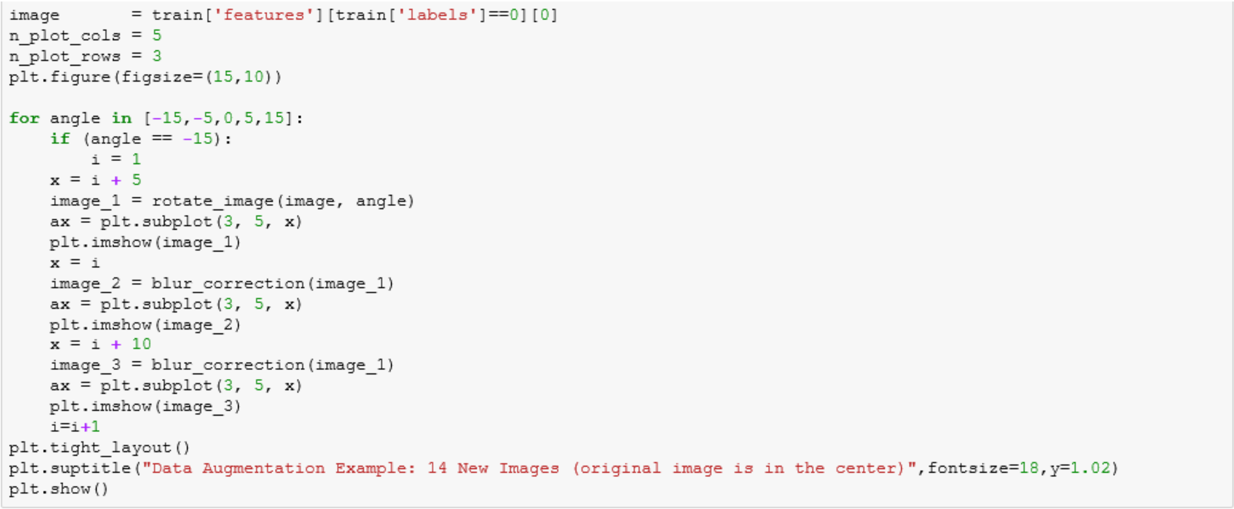
I would have liked to add a different plan rotation, to mimic non-perpendicular viewing angles, but the data augmentation was already taking so long that I dropped that idea. This is something worth considering for future improvement.

I also combine some these transformations (Rotate -> Blurring, Rotate -> Gamma).



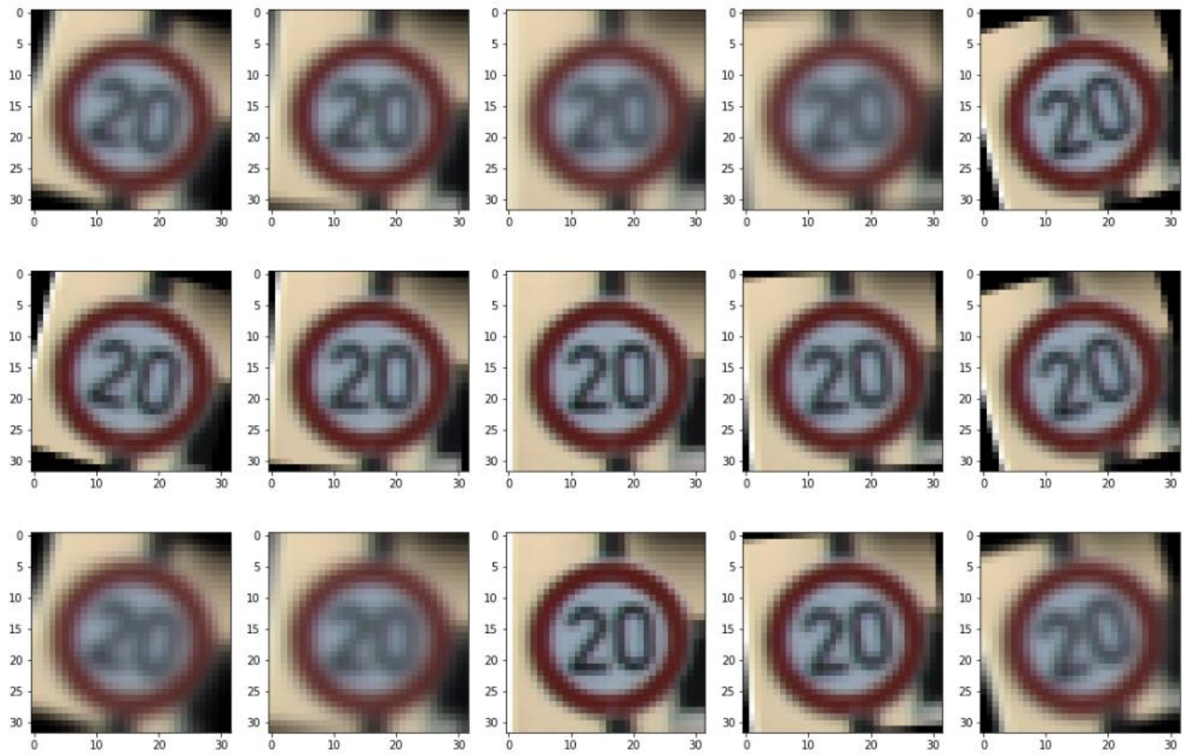
```
def rotate_image(image,angle):
    #rotation          = np.random.uniform(angle)-angle/2
    if (angle == 0):
        return image
    else:
        rotation          = angle
        rows,cols,channel = image.shape
        RotationMatrix    = cv2.getRotationMatrix2D((cols/2,rows/2),rotation,1)
        return cv2.warpAffine(image,RotationMatrix,(cols,rows))

def blur_correction(image):
    size              = (2 * np.random.randint(0, 3)) + 1
    return cv2.GaussianBlur(image, (size, size), 0)

def gamma_correction(image):
    gamma             = np.random.uniform(0.2, 1.5)
    invGamma          = 1.0 / gamma
    table             = np.array([((i / 255.0) ** invGamma) * 255 for i in np.arange(0, 256)]).astype("uint8")
    return cv2.LUT(image.astype("uint8"), table)
```

With a "Stop" sign as an example, I produce 14 new images using the following piece of code, with a subplot of 5 columns and 3 rows



```
image       = train['features'][train['labels']==0][0]
n_plot_cols = 5
n_plot_rows = 3
plt.figure(figsize=(15,10))

for angle in [-15,-5,0,5,15]:
    if (angle == -15):
        i = 1
    x = i + 5
    image_1 = rotate_image(image, angle)
    ax = plt.subplot(3, 5, x)
    plt.imshow(image_1)
    x = i
    image_2 = blur_correction(image_1)
    ax = plt.subplot(3, 5, x)
    plt.imshow(image_2)
    x = i + 10
    image_3 = blur_correction(image_1)
    ax = plt.subplot(3, 5, x)
    plt.imshow(image_3)
    i=i+1
plt.tight_layout()
plt.suptitle("Data Augmentation Example: 14 New Images (original image is in the center)",fontsize=18,y=1.02)
plt.show()
```

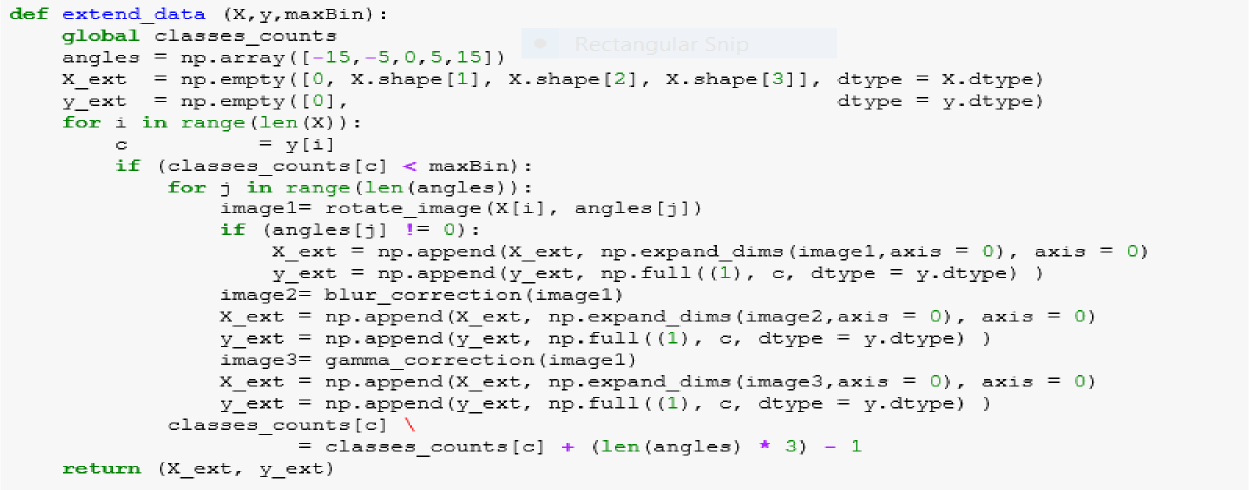
Data Augmentation Example: 14 New Images (original image is in the center)



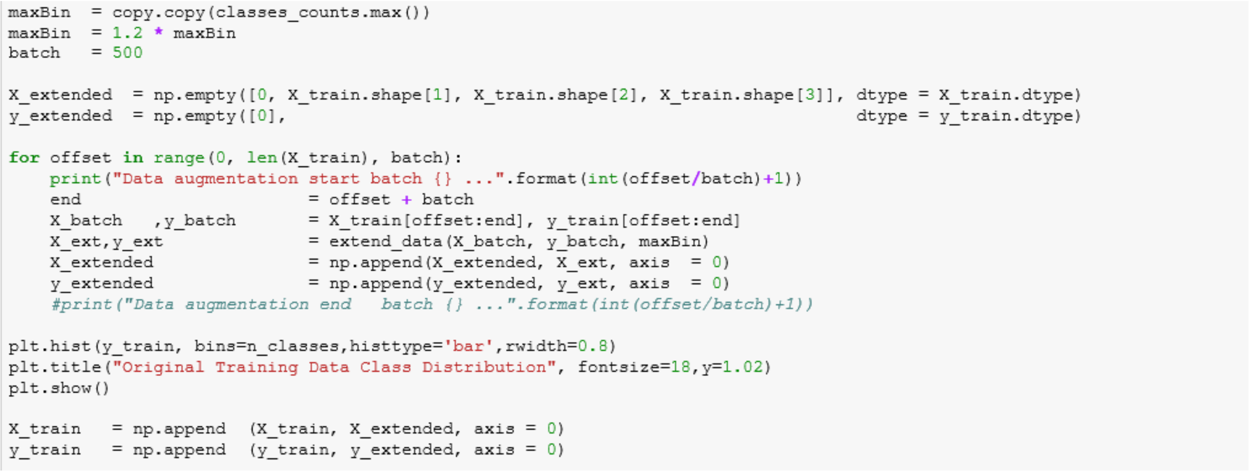
Finally, this process is generalized to the entire features training set.

I decided to limit the final maximum class count to be 1.2X of the original maximum class count, so that the new features training set is about 100K samples (2000 x 1.2 x 43).

The features training set augmentation out to be taking a very large amount of compute resources on a laptop, as well as on an AWS gpu instance. I am not sure exactly which part was causing the slowdown, but I found many internet references to numpy.array append() method being quite slow. The only way I could implement it was to use a batch scheme.

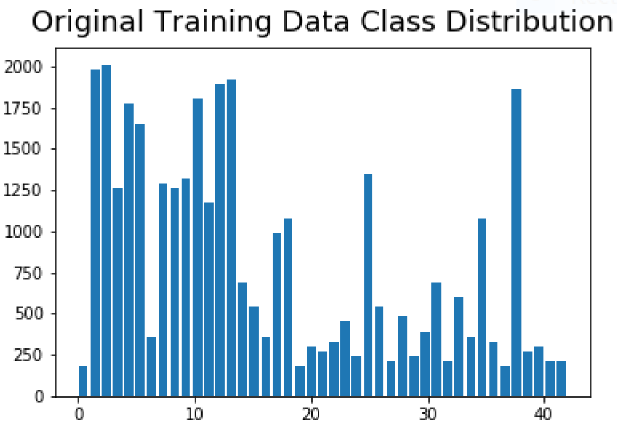


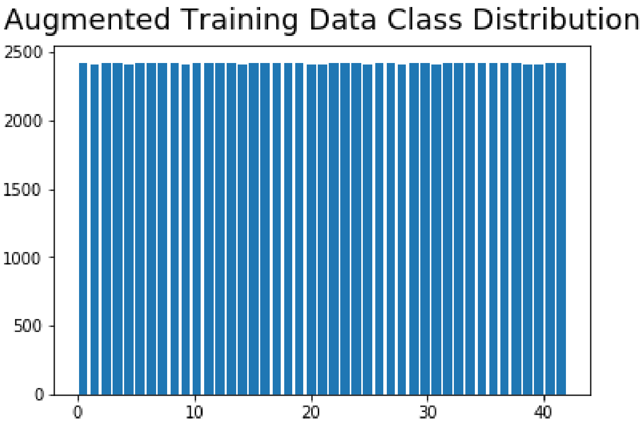
```
def extend_data (X,y,maxBin):
    global classes_counts
    angles = np.array([-15,-5,0,5,15])
    X_ext  = np.empty([0, X.shape[1], X.shape[2], X.shape[3]], dtype = X.dtype)
    y_ext  = np.empty([0],                                     dtype = y.dtype)
    for i in range(len(X)):
        c          = y[i]
        if (classes_counts[c] < maxBin):
            for j in range(len(angles)):
                image1= rotate_image(X[i], angles[j])
                if (angles[j] != 0):
                    X_ext = np.append(X_ext, np.expand_dims(image1,axis = 0), axis = 0)
                    y_ext = np.append(y_ext, np.full((1), c, dtype = y.dtype) )
                image2= blur_correction(image1)
                X_ext = np.append(X_ext, np.expand_dims(image2,axis = 0), axis = 0)
                y_ext = np.append(y_ext, np.full((1), c, dtype = y.dtype) )
                image3= gamma_correction(image1)
                X_ext = np.append(X_ext, np.expand_dims(image3,axis = 0), axis = 0)
                y_ext = np.append(y_ext, np.full((1), c, dtype = y.dtype) )
            classes_counts[c] \
                      = classes_counts[c] + (len(angles) * 3) - 1
    return (X_ext, y_ext)
```



```
maxBin  = copy.copy(classes_counts.max())
maxBin  = 1.2 * maxBin
batch   = 500

X_extended  = np.empty([0, X_train.shape[1], X_train.shape[2], X_train.shape[3]], dtype = X_train.dtype)
y_extended  = np.empty([0],                                                       dtype = y_train.dtype)

for offset in range(0, len(X_train), batch):
    print("Data augmentation start batch {} ...".format(int(offset/batch)+1))
    end                      = offset + batch
    X_batch   ,y_batch       = X_train[offset:end], y_train[offset:end]
    X_ext,y_ext              = extend_data(X_batch, y_batch, maxBin)
    X_extended               = np.append(X_extended, X_ext, axis  = 0)
    y_extended               = np.append(y_extended, y_ext, axis  = 0)
    #print("Data augmentation end   batch {} ...".format(int(offset/batch)+1))

plt.hist(y_train, bins=n_classes,histtype='bar',rwidth=0.8)
plt.title("Original Training Data Class Distribution", fontsize=18,y=1.02)
plt.show()

X_train   = np.append  (X_train, X_extended, axis = 0)
y_train   = np.append  (y_train, y_extended, axis = 0)
```

Finally, I plot a histogram of the class distribution in the features training set, before and augmentation.





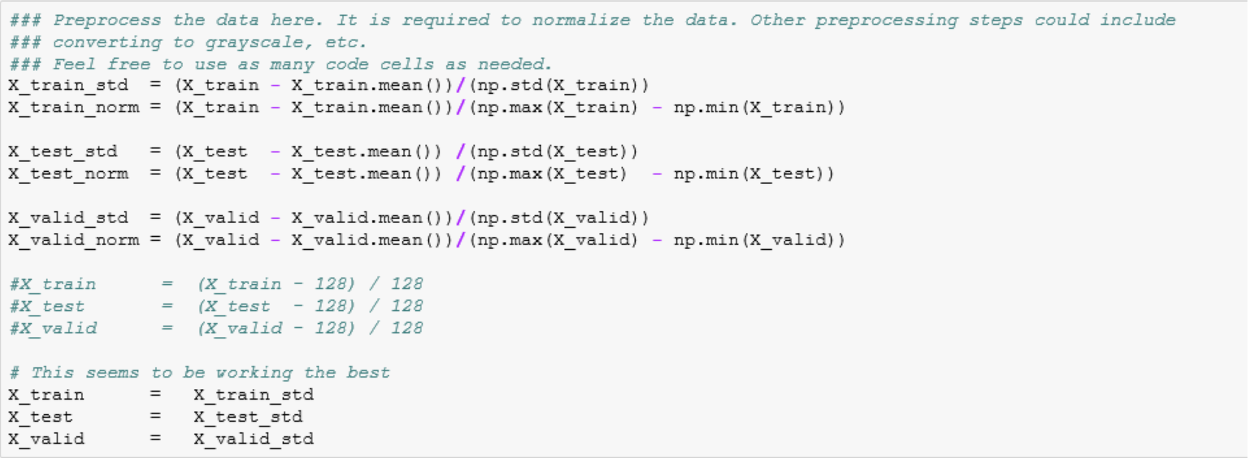
Pre-processing the Data

There are several suggested ways to pre-process the data and convert the image intensity from unint8 values into a floating format between -1 and 1.

I tried the following 3 all the way to features test set accuracy

- Normalization (subtract the mean, and divide by the (max-min))
- Standardization (subtract the mean, and divide by the standard deviation)
- Simpler one (subtract 128 and divide by 128)

As expected the simple scheme performs the worst and the standardization performs slightly better than the normalization.



```
### 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.
X_train_std  = (X_train - X_train.mean())/(np.std(X_train))
X_train_norm = (X_train - X_train.mean())/(np.max(X_train) - np.min(X_train))

X_test_std   = (X_test  - X_test.mean()) /(np.std(X_test))
X_test_norm  = (X_test  - X_test.mean()) /(np.max(X_test)  - np.min(X_test))

X_valid_std  = (X_valid - X_valid.mean())/(np.std(X_valid))
X_valid_norm = (X_valid - X_valid.mean())/(np.max(X_valid) - np.min(X_valid))

#X_train      =  (X_train - 128) / 128
#X_test       =  (X_test  - 128) / 128
#X_valid      =  (X_valid - 128) / 128

# This seems to be working the best
X_train      =   X_train_std
X_test       =   X_test_std
X_valid      =   X_valid_std
```

I decided to keep the images as RGB, given the importance of colors in signs definition, even though it may require 3x the amount of data and computation. Had I had a bit more time, I would have liked to create a single channel CNN to validate that decision.

Model Architecture

My model was heavily inspired by the LeNet convolutional neural network describe and tested in the class, using 3 convolution filters for feature extraction. I believe that the original model had 1 fewer convolution filter, and I figured that given the increase in the number of features to be extracted, this was required.

input	32x32x3 RBG image
Convolution 5x5	padding='Valid', Stride=1, Input=32x32x3,Output=28x28x32
RELU	
Max Pooling	ksize=2x2, stride=2x2, input=28x28x32,Ouput=14x14x32
Convolution 5x5	padding='Valid', Stride=1, Input=14x14x32,Output=10x10x64
RELU	
Max Pooling	ksize=2x2, stride=2x2, input=10x10x64,Ouput=5x5x64
Convolution 1x1	padding='Valid', Stride=1, Input=5x5x64,Output=5x5x64
RELU	
Flatten	Input=5x5x64,Output=1600
Fully Connected	Input=1600,Output=1024
RELU	
Dropout	Prob=0.5
Fully Connected	Input=1024,Output=256
RELU	
Dropout	Prob=0.5
Fully Connected	Input=256,Output=43
softmax	

Had I a bit more time, I would have liked to be a bit more ambitious in the architecture definition and explore the implementation of an inception module.

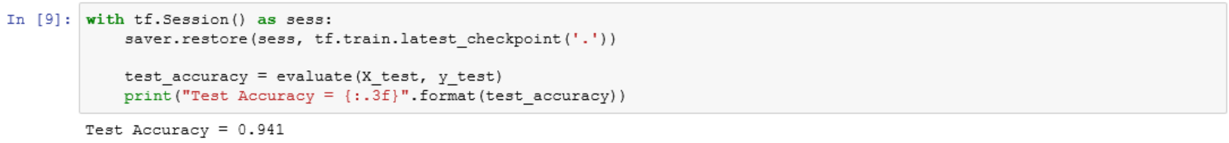
Model Training

The model is trained using the augmented features training set, a TensorFlow AdamOptimizer and the following hyperparameters:

Batch size	= 128
Epoch	= 50
Learning rate	= 0.0003

I have tried to increase the batch to 256 and reduce it to 64, and in both cases, the results were not as good as with 128.
I have tried to increase the learning rate to 0.003 and it was not as good as 0.001.
Learning rates smaller than 0.001 gave a slightly better result.

The features test set accuracy was 0.941 after 50 epoch.



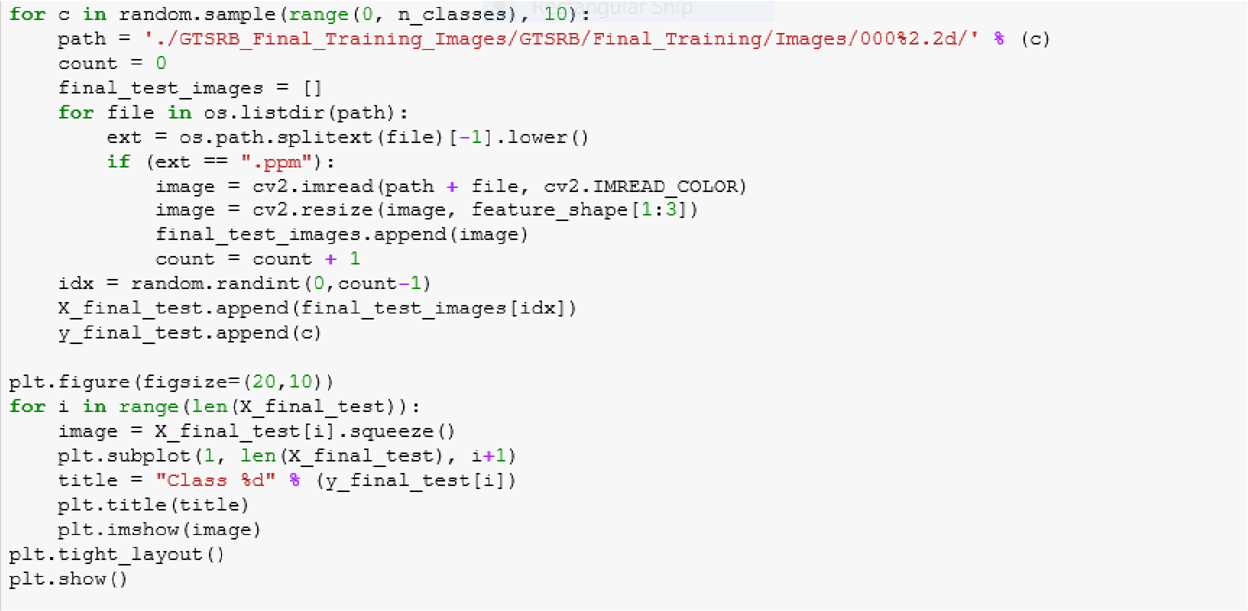
```
In [9]: with tf.Session() as sess:
            saver.restore(sess, tf.train.latest_checkpoint('.'))

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

        Test Accuracy = 0.941
```

Overall the test accuracy seems to be tracking very closely with the training and validation accuracy, which is quite remarkable considering that the CNN was originally intended to classify digits from 0 to 9

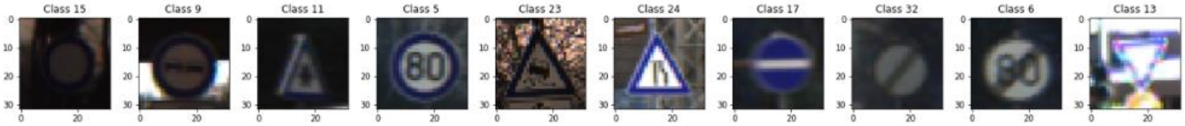
Test On New Images

Load New Images

I selected 10 signs randomly taken from the German Traffic Sign Data Set using the following piece of code. Note that all 10 images are taken from randomly selected classes (and unique).

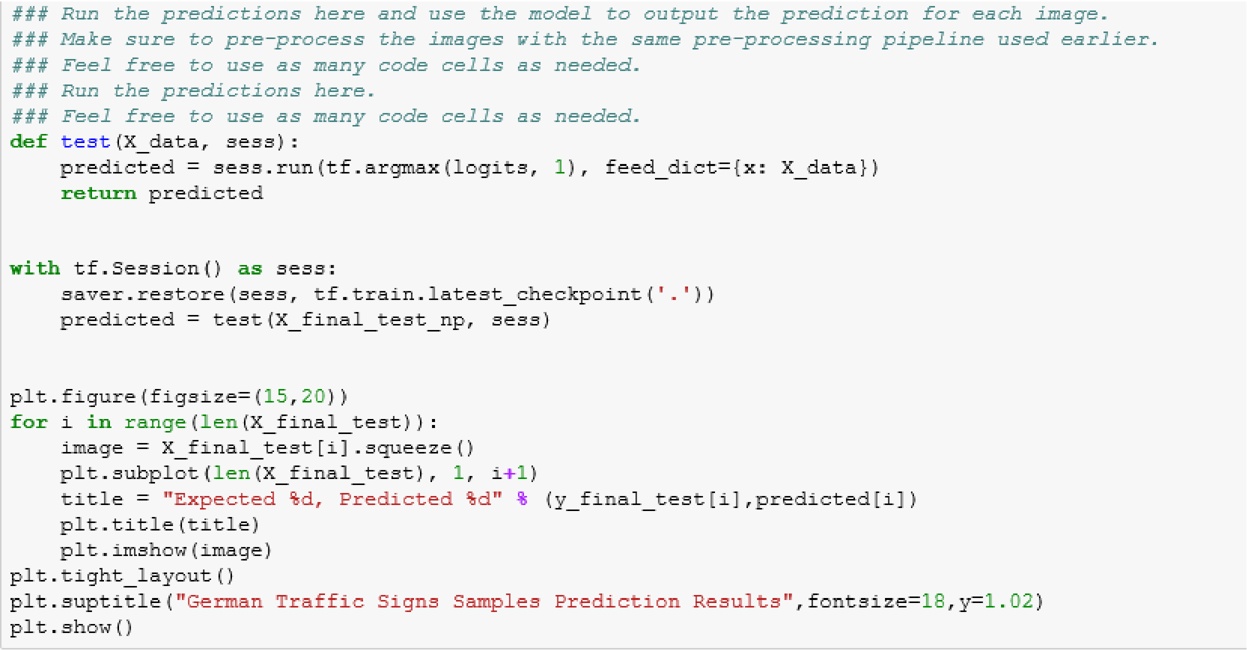


```
for c in random.sample(range(0, n_classes), 10):
    path = './GTSRB_Final_Training_Images/GTSRB/Final_Training/Images/000%2.2d/' % (c)
    count = 0
    final_test_images = []
    for file in os.listdir(path):
        ext = os.path.splitext(file)[-1].lower()
        if (ext == ".ppm"):
            image = cv2.imread(path + file, cv2.IMREAD_COLOR)
            image = cv2.resize(image, feature_shape[1:3])
            final_test_images.append(image)
            count = count + 1
    idx = random.randint(0,count-1)
    X_final_test.append(final_test_images[idx])
    y_final_test.append(c)

plt.figure(figsize=(20,10))
for i in range(len(X_final_test)):
    image = X_final_test[i].squeeze()
    plt.subplot(1, len(X_final_test), i+1)
    title = "Class %d" % (y_final_test[i])
    plt.title(title)
    plt.imshow(image)
plt.tight_layout()
plt.show()
```



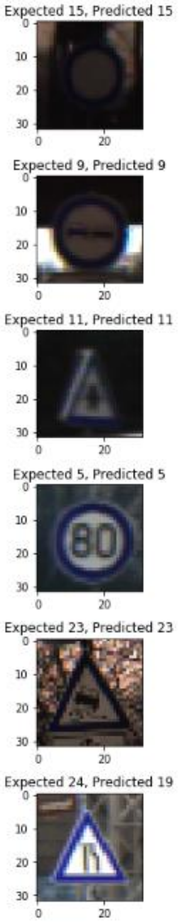
Predict New Images

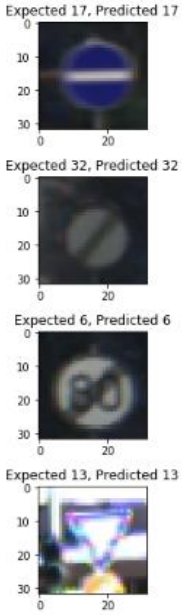
The prediction for the 10 selected new images are plotted using a subplot and the following piece of code:



```
### 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.
### Run the predictions here.
### Feel free to use as many code cells as needed.
def test(X_data, sess):
    predicted = sess.run(tf.argmax(logits, 1), feed_dict={x: X_data})
    return predicted


with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    predicted = test(X_final_test_np, sess)


plt.figure(figsize=(15,20))
for i in range(len(X_final_test)):
    image = X_final_test[i].squeeze()
    plt.subplot(len(X_final_test), 1, i+1)
    title = "Expected %d, Predicted %d" % (y_final_test[i],predicted[i])
    plt.title(title)
    plt.imshow(image)
plt.tight_layout()
plt.suptitle("German Traffic Signs Samples Prediction Results",fontsize=18,y=1.02)
plt.show()
```

German Traffic Signs Samples Prediction Results

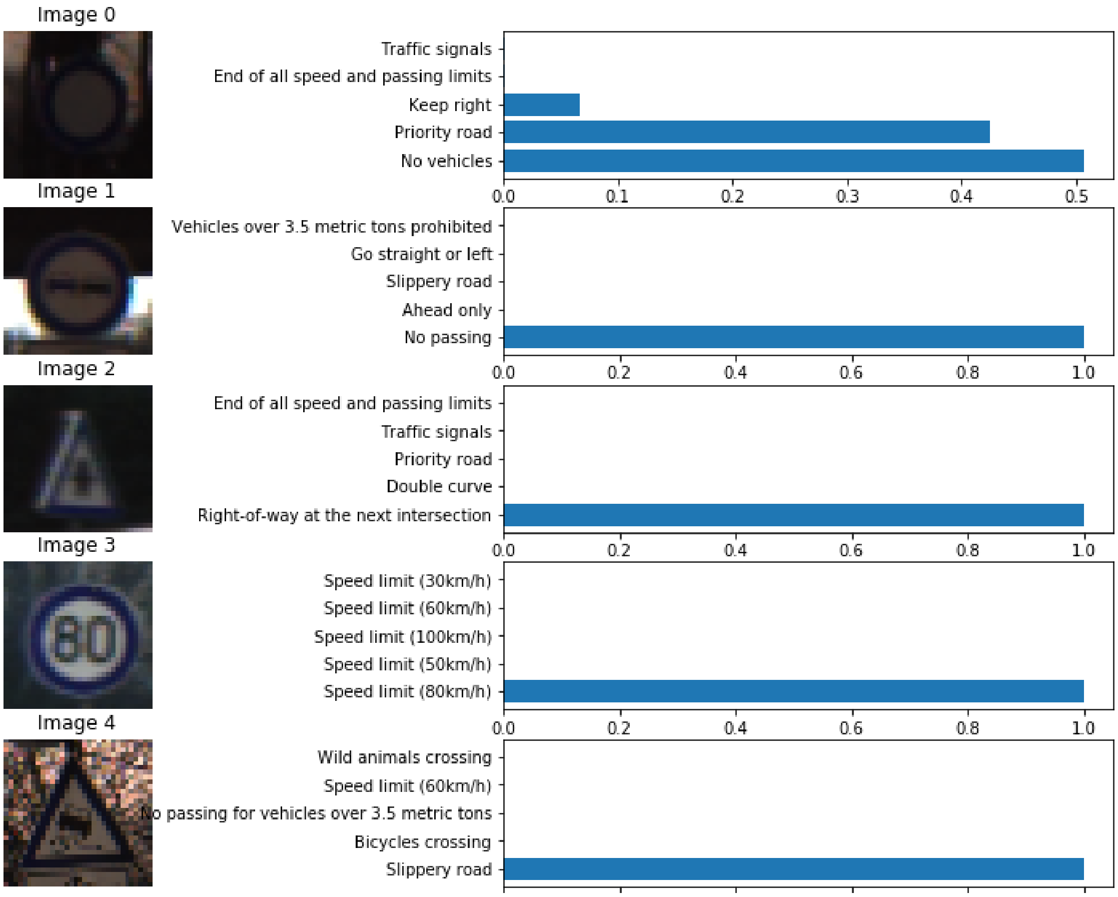

Expected 15, Predicted 15
Expected 9, Predicted 9
Expected 11, Predicted 11
Expected 5, Predicted 5
Expected 23, Predicted 23
Expected 24, Predicted 19
0
10
20
30
0
20


Expected 17, Predicted 17
Expected 32, Predicted 32
Expected 6, Predicted 6
Expected 13, Predicted 13
0
10
20
30
0
20

The model was able to correctly predict 9 out of 10 signs incorrectly predicting class 19 instead of class 24, making it very close to the accuracy of the features test set provided initilly.

Top 5 Softmax Probabilities

Finally, we predict the top 5 softmax probabilities for those 10 images:



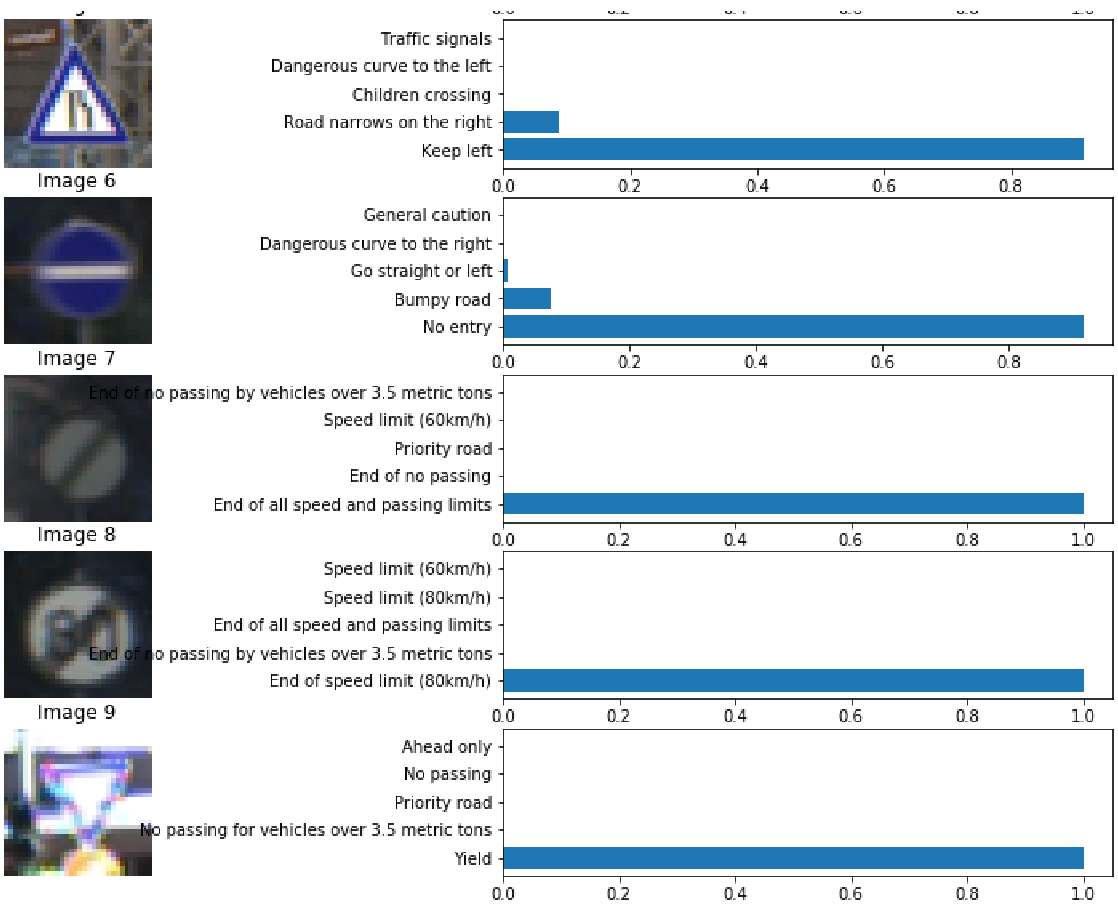


Image 6

Image 7

Image 8

Image 9