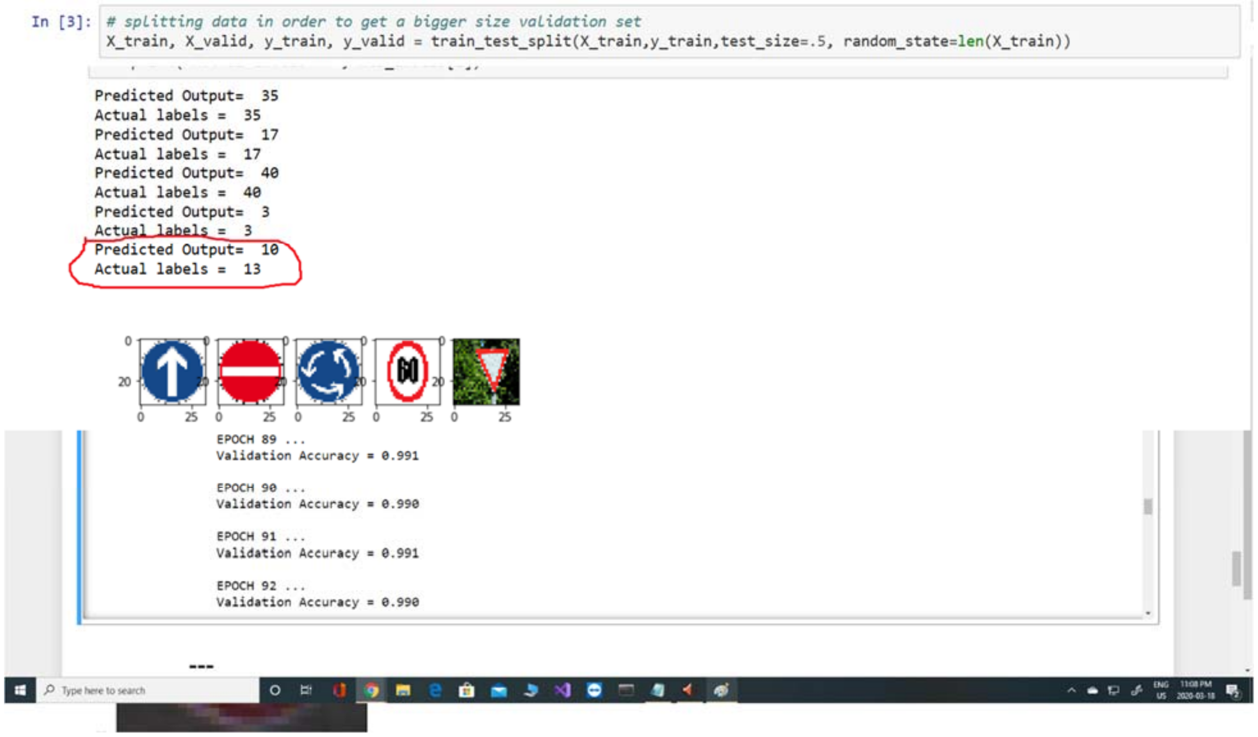


```
In [3]: # splitting data in order to get a bigger size validation set
        X_train, X_valid, y_train, y_valid = train_test_split(X_train,y_train,test_size=.5, random_state=len(X_train))
```

```
Predicted Output=  35
Actual labels =  35
Predicted Output=  17
Actual labels =  17
Predicted Output=  40
Actual labels =  40
Predicted Output=  3
Actual labels =  3
Predicted Output=  10
Actual labels =  13
```

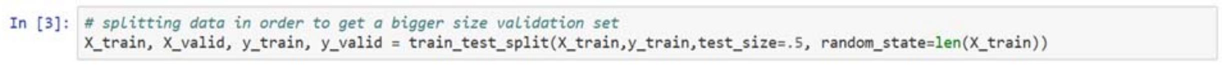
```
EPOCH 89 ...
Validation Accuracy = 0.991

EPOCH 90 ...
Validation Accuracy = 0.990

EPOCH 91 ...
Validation Accuracy = 0.991

EPOCH 92 ...
Validation Accuracy = 0.990

---
```

Refer to report.html for all details about coding and results.

I started the coding for this project in line 1 by importing training, validation, and test files, then I extracted training features and labels, validation features and labels, and test features and labels.

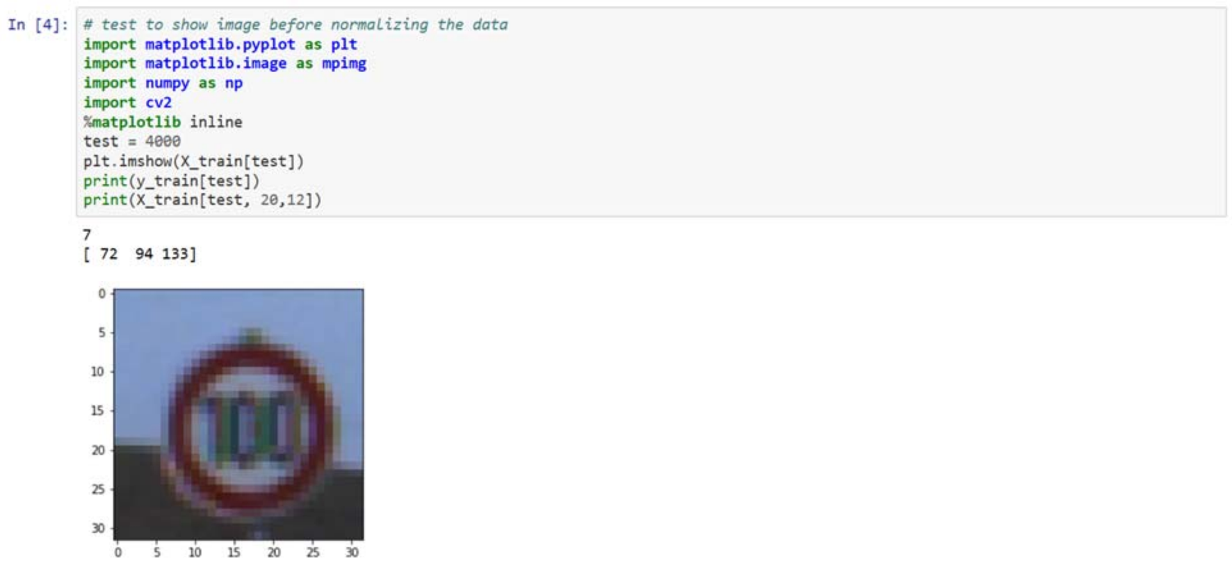
In my first trials of development of the project, I used the by default validation data, however I noticed the it is not a big data and has only around 4000 or so images. By trial I noticed that if I give more validation data to my model, then I can get better results in training my model. So, I decided to split the training data in to two sections: one section for training and the other section for validation.

For this purpose, I first shuffled the training data (line 2), then I divided them into half training set and half validation set (line 3).



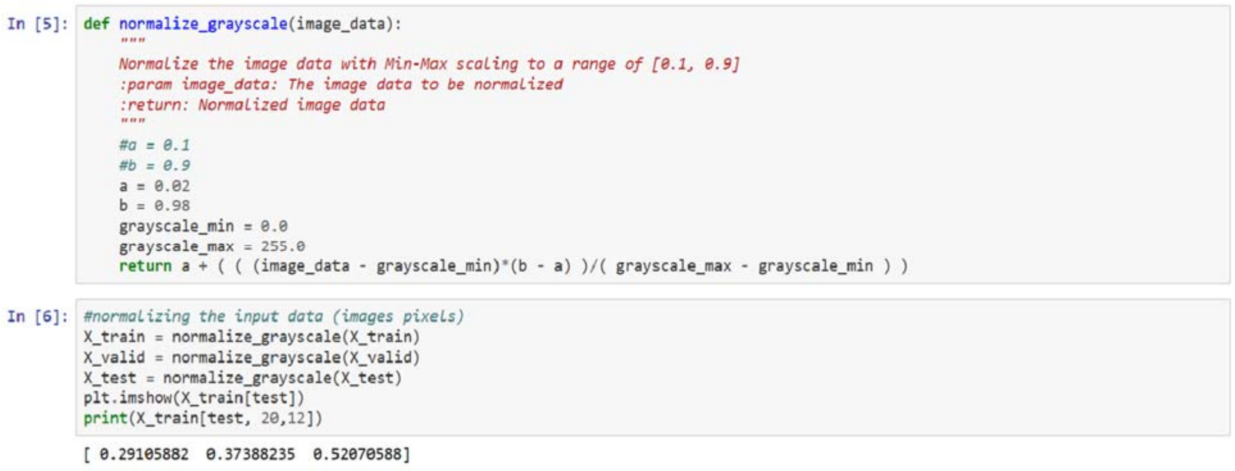
```
In [3]: # splitting data in order to get a bigger size validation set
        X_train, X_valid, y_train, y_valid = train_test_split(X_train,y_train,test_size=.5, random_state=len(X_train))
```

In line 4, I tested my training set and showed one image (image number 4000), and its associated label. It worked fine.



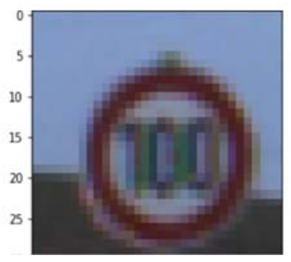
```
In [4]: # test to show image before normalizing the data
        import matplotlib.pyplot as plt
        import matplotlib.image as mpimg
        import numpy as np
        import cv2
        %matplotlib inline
        test = 4000
        plt.imshow(X_train[test])
        print(y_train[test])
        print(X_train[test, 20,12])

        7
        [ 72  94 133]
```

Again, in my first trials after developing the model, I did train my model without normalizing the training data inputs. I realized that I can not achieve excellent accuracies. Hence, I decided to normalize my training data inputs in line 5. Then in line6 I showed the same image that I had shown in line 4 (image number 4000). It showed that my normalization works. I also showed one pixel for RGB in both line 4 and line 6 to compare.



```
In [5]: def normalize_grayscale(image_data):
            """
            Normalize the image data with Min-Max scaling to a range of [0.1, 0.9]
            :param image_data: The image data to be normalized
            :return: Normalized image data
            """
            #a = 0.1
            #b = 0.9
            a = 0.02
            b = 0.98
            grayscale_min = 0.0
            grayscale_max = 255.0
            return a + ( ( (image_data - grayscale_min)*(b - a) )/( grayscale_max - grayscale_min ) )
```

```
In [6]: #normalizing the input data (images pixels)
        X_train = normalize_grayscale(X_train)
        X_valid = normalize_grayscale(X_valid)
        X_test = normalize_grayscale(X_test)
        plt.imshow(X_train[test])
        print(X_train[test, 20,12])

        [ 0.29105882  0.37388235  0.52070588]
```



Then in line 7, I did some statistics about number of training, validation, and test sets, classes, etc.:

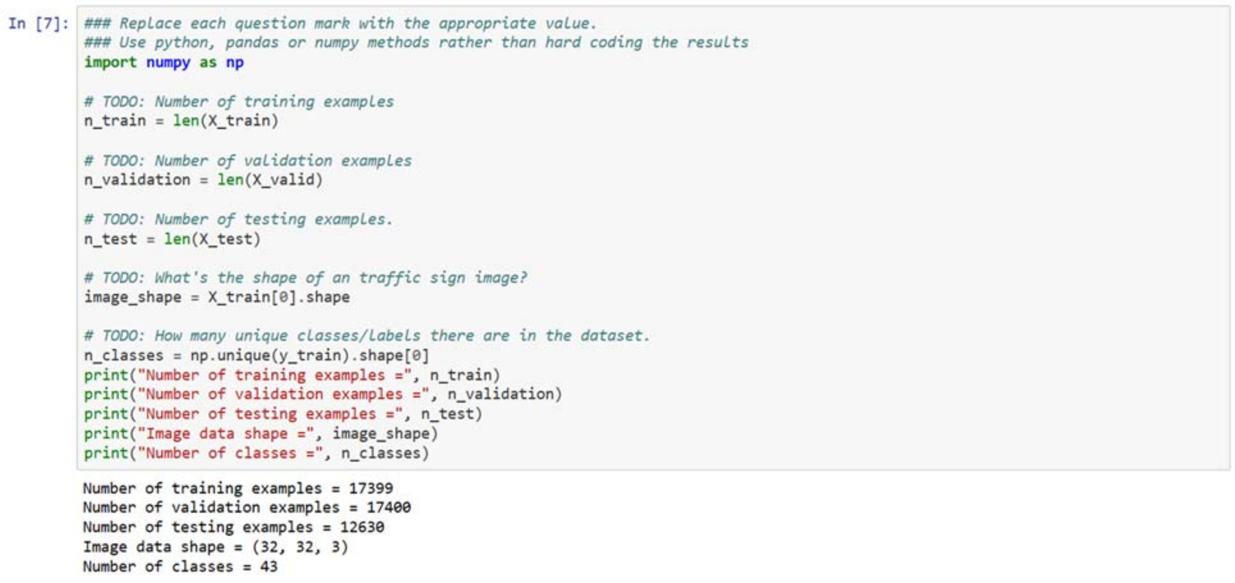
Number of training examples = 17399

Number of validation examples = 17400

Number of testing examples = 12630

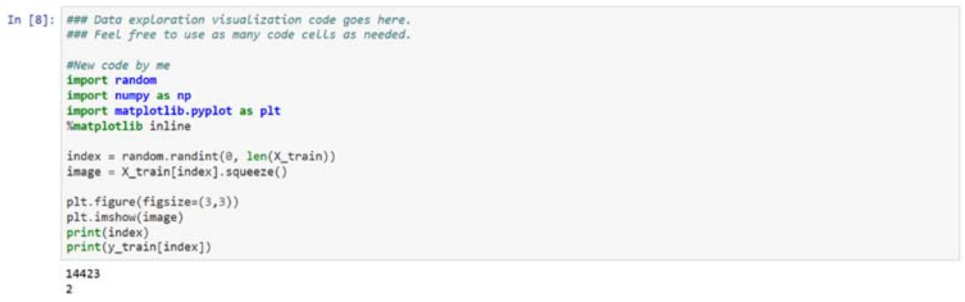
Image data shape = (32, 32, 3)

Number of classes = 43

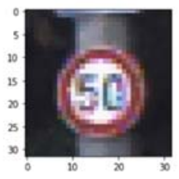


```
In [7]: ### Replace each question mark with the appropriate value.
        ### Use python, pandas or numpy methods rather than hard coding the results
        import numpy as np

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

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

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

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

        # TODO: How many unique classes/labels there are in the dataset.
        n_classes = np.unique(y_train).shape[0]
        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 = 17399
        Number of validation examples = 17400
        Number of testing examples = 12630
        Image data shape = (32, 32, 3)
        Number of classes = 43
```

In line 8, I randomly plotted one image with its associated class and then compared with German Traffic sign data base. It worked fine.



```
In [8]: ### Data exploration visualization code goes here.
        ### Feel free to use as many code cells as needed.

        #New code by me
        import random
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

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

        plt.figure(figsize=(3,3))
        plt.imshow(image)
        print(index)
        print(y_train[index])

        14423
        2
```



In line 9, I shuffled my training data to avoid any possible biases

In line 11, I built my training architecture, I adjusted parameters for RGB images (3 channels). In my initial trails after developing my model and all codes, I noticed I can not get desired accuracy results. Hence, I added two drop out layers between fc0 and fc1, and then between fc1 and fc2. Below is asummary of artchtecture I used:

```
Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.

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

Pooling. Input = 28x28x6. Output = 14x14x6.

Layer 2: Convolutional. Output = 10x10x16.

Activation: conv2 = tf.nn.relu(conv2)
```

```
Pooling. Input = 10x10x16. Output = 5x5x16.
Flatten. Input = 5x5x16. Output = 400.
Activation: fc1 = tf.nn.relu(fc1)
            fc1 = tf.nn.dropout(fc1, keep_prob)
Layer 4: Fully Connected. Input = 120. Output = 84.
Activation: fc2 = tf.nn.relu(fc2)
            fc2 = tf.nn.dropout(fc2, keep_prob)
Layer 5: Fully Connected. Input = 84. Output = 43.
```

Then I added line 12 for placeholder for features and labels.

In line 13 I decided to put learning rate as .001. again, when I was teaching my model (after finish) by learning rate of .01, my model was trapped in a local optimum and could not escape from it. I also tried learning rate of .0005 (half the .001), and I noticed it does not improve the accuracy of the model significantly.

In line 14, I defined my training pipeline including loss_operation, training_operation, and cross_entropy, and adam optimizer. I also provided values to number of ephocs and batch size. As we will see down in this report, I tried several combinations of ephocs and batch sizes that I will explain in summary during this report.

In line 15, I defined another pipeline for evaluating my model. I defined evaluate function for evaluating the accuracy of my model in each ephocs.

The combination of training pipeline and evaluating pipeline defines the cross entropy mean calculations and back propagation and combined build the epochs.

Then in line 16, I started training my model by my training data set, and validation of my model by validation data set in each ephocs. I used batch method which is a kind of stochastic gradient descent method. I did several trials as per below:

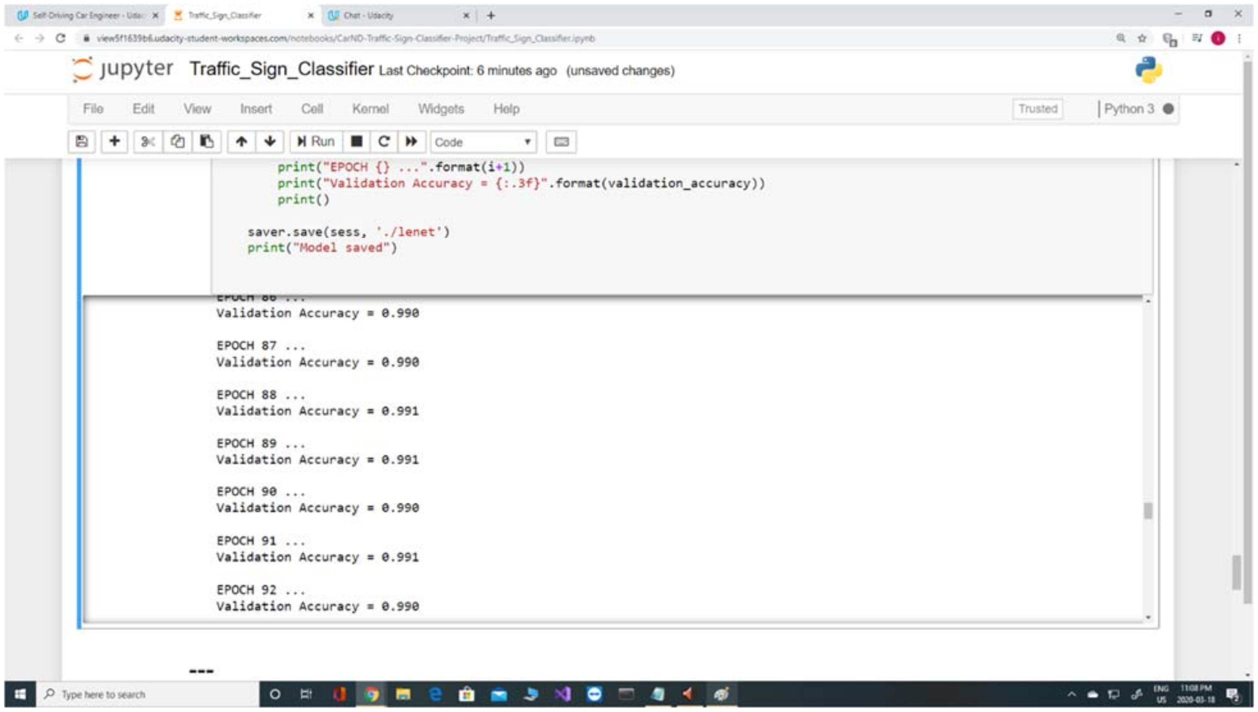
1) Rate .01: Un successful. Model is trapped in a bad local optimum point.
2) Normalization, rate = .001, ephocs = 100, batch = 64, two drop outs, original smaller set of validation data. Best validation accuracy: 0.956 in epochs 109
3) Normalization, rate = .001, ephocs = 150, batch = 128, two drop outs, original smaller set of validation data. Best validation accuracy: 0.977 in epochs 128
4) Normalization, rate = .0005(half above), ephocs = 150, batch = 128, two drop outs, original smaller set of validation data. Best validation accuracy: 0.968 in epochs 114
5) **Normalization, rate = .001, ephocs = 150, batch = 128, two drop outs, split to 50% training50% validation. best validation accuracy: 0.990 in ephocs 118→ best result**

6) -normalization, rate = .001, ephocs = 150, batch = 128, two drop outs, split to 70% training-30% validation. best validation accuracy: 0.983 in epochs 139

7) normalization, rate = .001, ephocs = 100, batch = 256, two drop outs--> best validation accuracy:0.963 in epochs 138

I picked up scenario 5 for training my model. After finishing the training, I saved my model.

Below is screen shot from a similar scenario t oscenario 5 above, In which the model reached accuracies roughly above 99%:

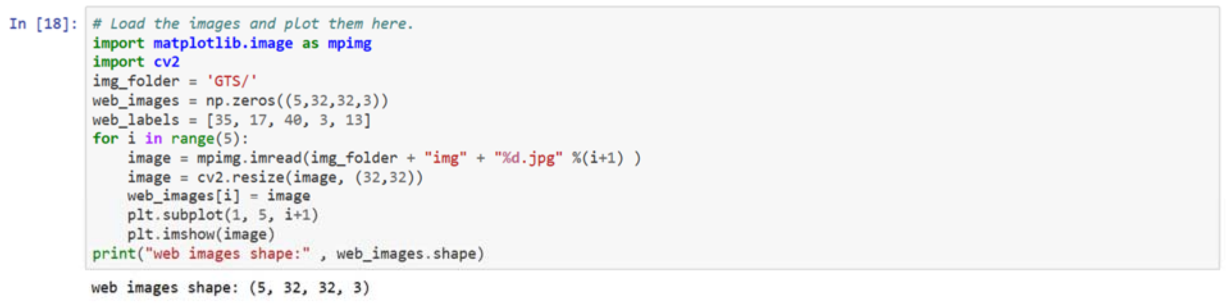

jupyter Traffic_Sign_Classifier Last Checkpoint: 6 minutes ago (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
        print("EPOCH {} ...".format(i+1))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()

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

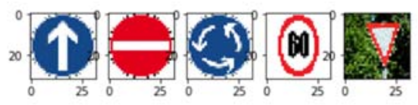
```
Validation Accuracy = 0.990

EPOCH 87 ...
Validation Accuracy = 0.990

EPOCH 88 ...
Validation Accuracy = 0.991

EPOCH 89 ...
Validation Accuracy = 0.991

EPOCH 90 ...
Validation Accuracy = 0.990

EPOCH 91 ...
Validation Accuracy = 0.991

EPOCH 92 ...
Validation Accuracy = 0.990
```

Then in line 17, I loaded my model and checked it with my test data, as seen in html file, it had .936 93.6%) accuracy.

Then in line 18, I did some image processing on the 5 images that I had downloaded from the web for German traffic signs and saved in a directory named: GTS. I showed them herewith their associated labels ([35, 17, 40, 3, 13]).



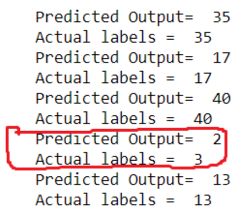
```
In [18]: # Load the images and plot them here.
         import matplotlib.image as mpimg
         import cv2
         img_folder = 'GTS/'
         web_images = np.zeros((5,32,32,3))
         web_labels = [35, 17, 40, 3, 13]
         for i in range(5):
             image = mpimg.imread(img_folder + "img" + "%d.jpg" %(i+1) )
             image = cv2.resize(image, (32,32))
             web_images[i] = image
             plt.subplot(1, 5, i+1)
             plt.imshow(image)
         print("web images shape:" , web_images.shape)

web images shape: (5, 32, 32, 3)
```



In line 19, I tested my model against those 5 images, and got the accuracy of 80%. Then I printed the results of model against the actual labels in line 22. For this I also had some coding above this line and in the model architecture for softmax and top_k and also to show the top 5 softmax probabilities for each of the 5 images from web:

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

top_k = tf.nn.top_k(softmax, k =5)
```

From the results, it showed that only the last image was classified wrongly, and the other 4 images were classified correctly.

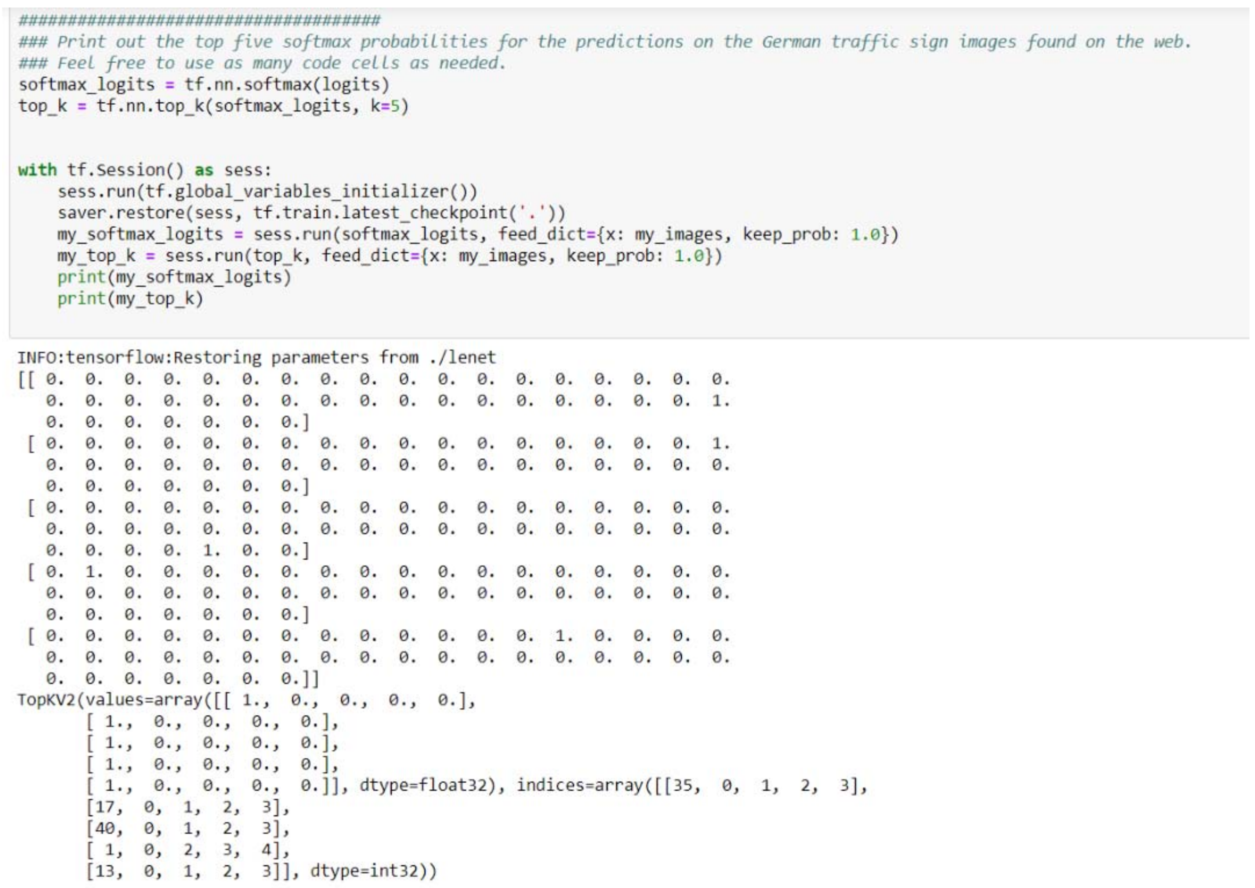


```
Predicted Output=  35
Actual labels =  35
Predicted Output=  17
Actual labels =  17
Predicted Output=  40
Actual labels =  40
Predicted Output=  2
Actual labels =  3
Predicted Output=  13
Actual labels =  13
```

The model was able to correctly guess 4 of the 5 traffic signs, which gives an accuracy of 80%. This compares less to the accuracy on the test set of 93.6%. Since accuracy on test model is 93.6%, there should be some images on the web that my model will not recognize. It looks that the model has some difficulties in recognizing images that have text or image in the middle (e.g. Speed limit signs). A reason could be that the texts or images inside are small in a 32 * 32 image and difficult for visibility. At the 32x32 resolution and low brightness or contrast, they are hardly distinguishable.

For future improvements, images can be processed to increase contrast and reduce the size.

Then I executed below code for showing the top 5 softmax probabilities and got below results:



```
#######################################
### 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.
softmax_logits = tf.nn.softmax(logits)
top_k = tf.nn.top_k(softmax_logits, k=5)


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    my_softmax_logits = sess.run(softmax_logits, feed_dict={x: my_images, keep_prob: 1.0})
    my_top_k = sess.run(top_k, feed_dict={x: my_images, keep_prob: 1.0})
    print(my_softmax_logits)
    print(my_top_k)
```

```
INFO:tensorflow:Restoring parameters from ./lenet
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.
   0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  1.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.]]
TopKV2(values=array([[ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.]], dtype=float32), indices=array([[35,  0,  1,  2,  3],
       [17,  0,  1,  2,  3],
       [40,  0,  1,  2,  3],
       [ 1,  0,  2,  3,  4],
       [13,  0,  1,  2,  3]], dtype=int32))
```

As seen above, it shows the location of predictions correctly as 1.0, except for the 4th image. The network is so confident of the predictions that it is outputting a probability of 1.0. This is normal

Some lessons learned:

- Normalization changes the game and improves the performance significantly

- 128 batch size is best for me

- Two drop outs between fc0 and fc1, and then fc1 and fc2 improve the performance significantly

- Rate .01 does not help and traps in local optimum

- More validation data improves the performance