Traffic Sign Classifier Project

Writeup

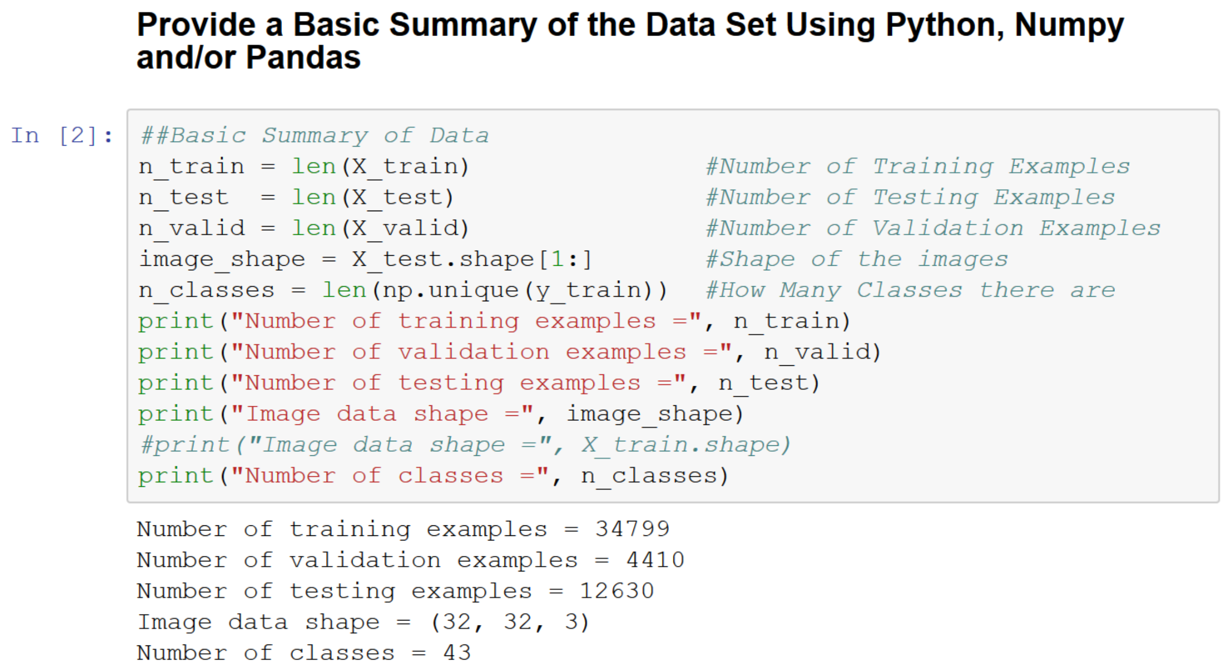
This is the readme file!

Data Set Summary and Exploration

Basic Summary

This code is listed in the second cell, listed immediately below the descriptor ‘Step1: Dataset Summary and Exploration’.

Here, I used the ‘len’ function, ‘shape’ function, and displayed the sizes of the images.



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

In [2]:

```
##Basic Summary of Data
n_train = len(X_train)                    #Number of Training Examples
n_test  = len(X_test)                     #Number of Testing Examples
n_valid = len(X_valid)                    #Number of Validation Examples
image_shape = X_test.shape[1:]            #Shape of the images
n_classes = len(np.unique(y_train))       #How Many Classes there are
print("Number of training examples =", n_train)
print("Number of validation examples =", n_valid)
print("Number of testing examples =", n_test)
print("Image data shape =", image_shape)
#print("Image data shape =", X_train.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
```

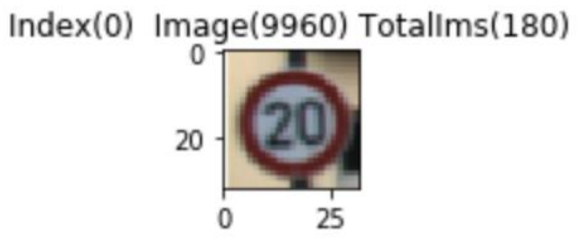
Note that the number of training examples is 34799, validation examples is 4410, and testing examples are 12630. Initial image data shape is 32x32x3. Total classes are 43.

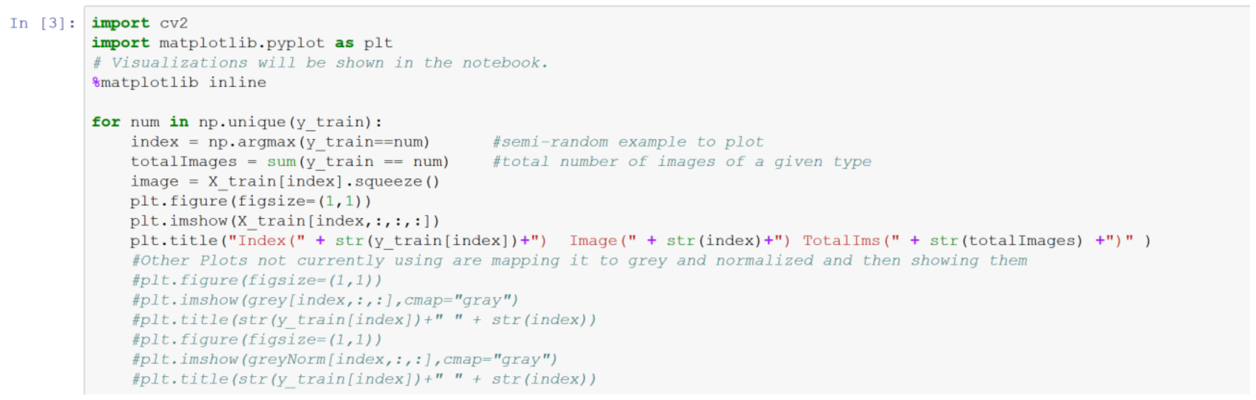
Exploration of Dataset

In the third cell, shown following the example image, I go through each unique identifier in the y_train database, and for each I select an image, display it, and show how many of them there are.

Since there are 43 classes of images this loops through 43 times. In the example image I included, I show index 0, which happens to be a 20 KPH speed limit. I display image 9960 for reference, and show that there are 180 total images that have this same identifier.

Although I’m not currently displaying greyscale images, I have some of that code included but commented out.



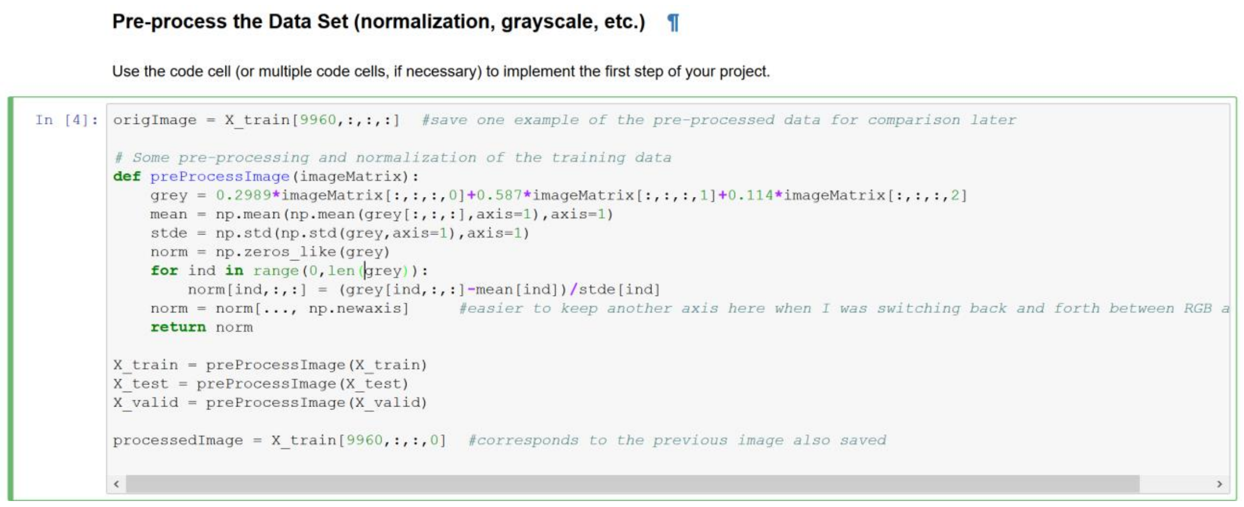


```
In [3]: import cv2
        import matplotlib.pyplot as plt
        # Visualizations will be shown in the notebook.
        %matplotlib inline

        for num in np.unique(y_train):
            index = np.argmax(y_train==num)       #semi-random example to plot
            totalImages = sum(y_train == num)     #total number of images of a given type
            image = X_train[index].squeeze()
            plt.figure(figsize=(1,1))
            plt.imshow(X_train[index,:,:,:])
            plt.title("Index(" + str(y_train[index])+")  Image(" + str(index)+") TotalIms(" + str(totalImages) +")" )
            #Other Plots not currently using are mapping it to grey and normalized and then showing them
            #plt.figure(figsize=(1,1))
            #plt.imshow(grey[index,:,:],cmap="gray")
            #plt.title(str(y_train[index])+" " + str(index))
            #plt.figure(figsize=(1,1))
            #plt.imshow(greyNorm[index,:,:],cmap="gray")
            #plt.title(str(y_train[index])+" " + str(index))
```

Design and Test a Model Architecture

Preprocess

My pre-processing script is shown in the following code, which is the fourth cell in the program. I convert the image to grey and then normalize the images. (more explanation follows)



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

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

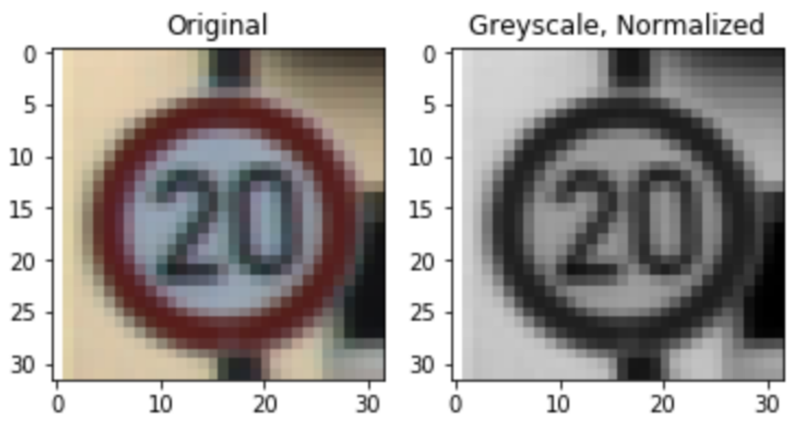
```
In [4]: origImage = X_train[9960,:,:,:]  #save one example of the pre-processed data for comparison later

        # Some pre-processing and normalization of the training data
        def preProcessImage(imageMatrix):
            grey = 0.2989*imageMatrix[:,:,:,0]+0.587*imageMatrix[:,:,:,1]+0.114*imageMatrix[:,:,:,2]
            mean = np.mean(np.mean(grey[:,:,:],axis=1),axis=1)
            stde = np.std(np.std(grey,axis=1),axis=1)
            norm = np.zeros_like(grey)
            for ind in range(0,len(grey)):
                norm[ind,:,:] = (grey[ind,:,:]-mean[ind])/stde[ind]
            norm = norm[..., np.newaxis]      #easier to keep another axis here when I was switching back and forth between RGB a
            return norm

        X_train = preProcessImage(X_train)
        X_test = preProcessImage(X_test)
        X_valid = preProcessImage(X_valid)

        processedImage = X_train[9960,:,:,0]  #corresponds to the previous image also saved
```

For convenience, a portion of Appendix A is shown in the following figure. It shows that, after 10 epochs, for example, going from Greyscale to RGB architecture lost 6% accuracy (as seen in trial runs 16 and 17). Therefore I concentrated my efforts on greyscale imaging; although given more time I might try to use both RGB and greyscale in a 4-D image. Additionally, when compared non-normalized and normalized data (say, trials 21 and 22), normalized images showed a 6% improvement over non-normalized images; so I concentrated on normalized images.

	Inputs				Architecture			Preproc.											
Trial #	Batch	Sigma	Mu	Rate	Version	Height	Pooling	Color	Normalized	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5	Epoch 6	Epoch 7	Epoch 8	Epoch 9	Epoch 10
16	50	0.1	0	0.001	V4	10	5x5	Grey	N	0.78	0.85	0.86	0.90	0.91	0.88	0.91	0.91	0.91	0.92
17	50	0.1	0	0.001	V5	10	5x5	RGB	N	0.66	0.75	0.81	0.83	0.83	0.84	0.86	0.85	0.88	0.86
21	50	0.1	0	0.001	V5	10	5x5	Grey	N	0.66	0.78	0.80	0.83	0.85	0.82	0.85	0.87	0.84	0.84
22	50	0.1	0	0.001	V4	10	5x5	Grey	Y	0.74	0.80	0.85	0.87	0.87	0.87	0.87	0.87	0.89	0.90

The effect of the normalization is shown on the figure below, which is the fifth cell in the program. Although both are clear to the human eye, the algorithm chosen had more capability determining the greyscale, normalized image.

```
In [5]: fig = plt.figure()
        a=fig.add_subplot(1,2,1)
        imgplot = plt.imshow(origImage)
        a.set_title('Original')
        a=fig.add_subplot(1,2,2)
        imgplot = plt.imshow(processedImage,cmap="gray")
        a.set_title('Greyscale, Normalized')

Out[5]: <matplotlib.text.Text at 0x24a3223f5f8>
```



Training, Validation, Testing Data

There is some confusion here – the rubric asked how I split the data into training, validation, and testing sets; and what I actually did was use the data given to me which was already split. I did not generate additional data. Earlier in this document, in ‘Data Set Summary’, I detail how many of each type are in the data set.

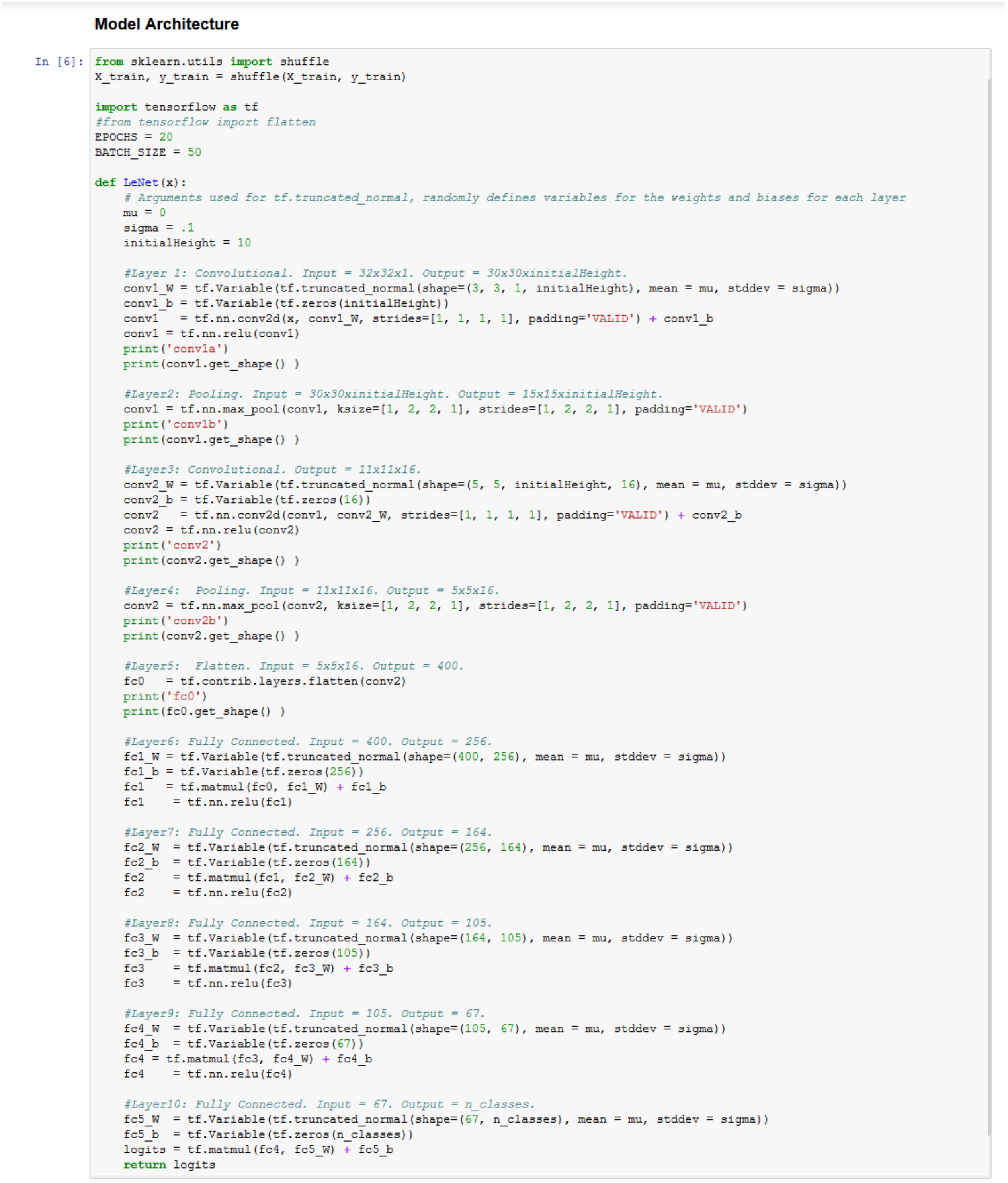
Final Architecture

The code for the model is in the sixth cell of the notebook. As can be seen, it is heavily based on the Lenet architecture. From Appendix A, it can be seen that I spent time optimizing initial heights, pooling levels, and batch size. Not seen, but implicit in the 'Version' Column in Appendix A, is the fact that I attempted to optimize number of layers as well, but with only limited success. I eventually settled on 10 total layers, listed in the table below. I also tried a nested function to increase the number of layers (rather than 5 roughly-copied-and-pasted layers, i.e., layers 6 through 10) but ran into problems with that architecture that I was unable to resolve. Nevertheless the architecture and code below were able to achieve the desired accuracy.

The difference between adding successive FC layers was perhaps an additional 1% per layer. Changing from a 5x5 to a 3x3 didn't change accuracy very much; see, e.g., trials 33 and 34 in Appendix A.

		Input				Output		
Layer1	Convolutional	32	32	1		30	30	10
Layer2	Max Pool	30	30	10		15	15	10
Layer3	Convolutional	15	15	10		11	11	16
Layer4	Max Pool	11	11	16		5	5	16
Layer5	Flatten	5	5	16		400		
Layer6	FC	400				256		
Layer7	FC	256				164		
Layer8	FC	164				105		
Layer9	FC	105				67		
Layer10	FC	67				43		

The code is on the following page is for reference; but of course it will be more legible in the html file.



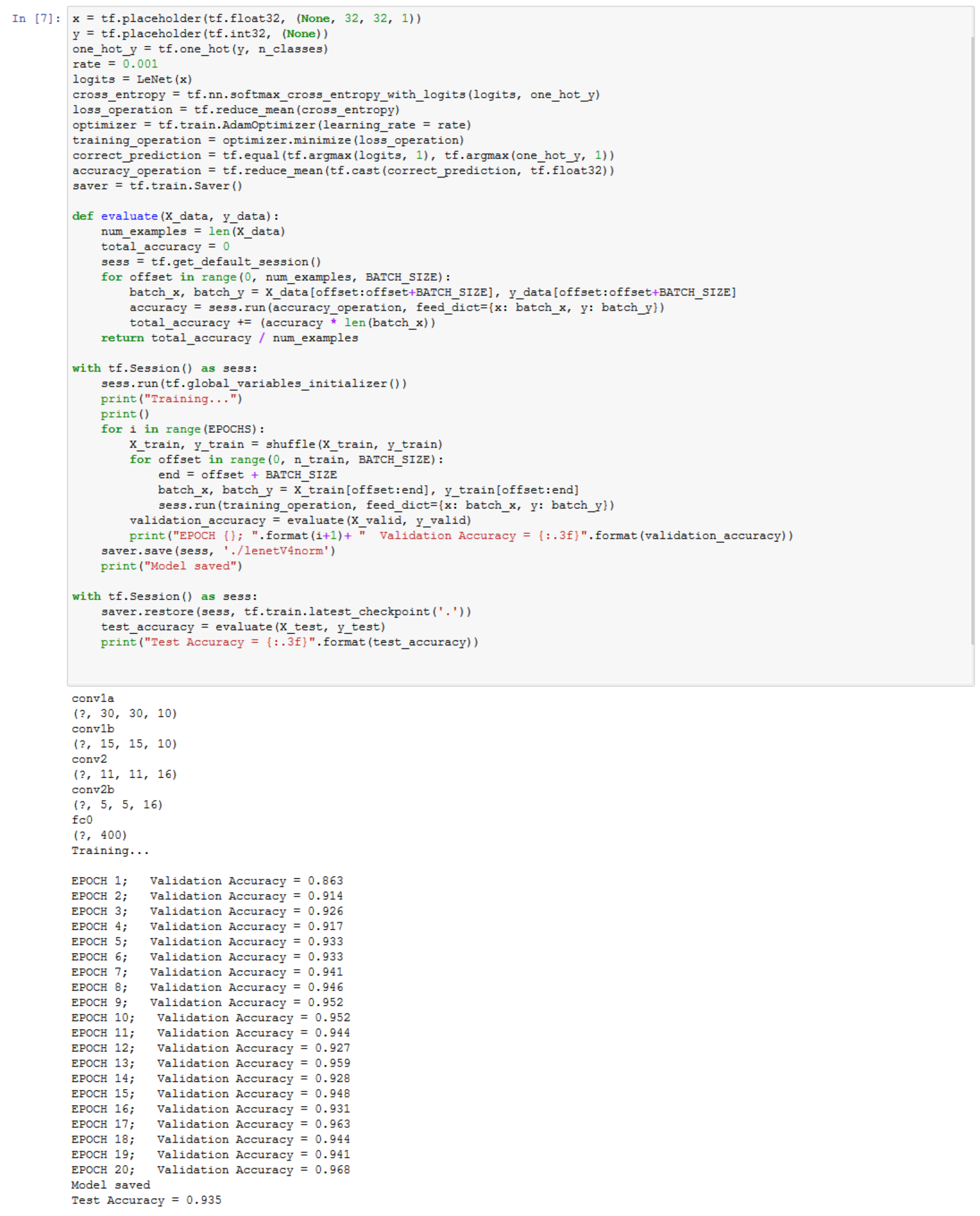
Model Architecture

```
In [6]: from sklearn.utils import shuffle
        X_train, y_train = shuffle(X_train, y_train)

        import tensorflow as tf
        #from tensorflow import flatten
        EPOCHS = 20
        BATCH_SIZE = 50

        def LeNet(x):
            # Arguments used for tf.truncated_normal, randomly defines variables for the weights and biases for each layer
            mu = 0
            sigma = .1
            initialHeight = 10

            #Layer 1: Convolutional. Input = 32x32x1. Output = 30x30xinitialHeight.
            conv1_W = tf.Variable(tf.truncated_normal(shape=(3, 3, 1, initialHeight), mean = mu, stddev = sigma))
            conv1_b = tf.Variable(tf.zeros(initialHeight))
            conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') + conv1_b
            conv1 = tf.nn.relu(conv1)
            print('conv1a')
            print(conv1.get_shape() )

            #Layer2: Pooling. Input = 30x30xinitialHeight. Output = 15x15xinitialHeight.
            conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')
            print('conv1b')
            print(conv1.get_shape() )

            #Layer3: Convolutional. Output = 11x11x16.
            conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, initialHeight, 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
            conv2 = tf.nn.relu(conv2)
            print('conv2')
            print(conv2.get_shape() )

            #Layer4:  Pooling. Input = 11x11x16. Output = 5x5x16.
            conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')
            print('conv2b')
            print(conv2.get_shape() )

            #Layer5:  Flatten. Input = 5x5x16. Output = 400.
            fc0   = tf.contrib.layers.flatten(conv2)
            print('fc0')
            print(fc0.get_shape() )

            #Layer6: Fully Connected. Input = 400. Output = 256.
            fc1_W = tf.Variable(tf.truncated_normal(shape=(400, 256), mean = mu, stddev = sigma))
            fc1_b = tf.Variable(tf.zeros(256))
            fc1   = tf.matmul(fc0, fc1_W) + fc1_b
            fc1    = tf.nn.relu(fc1)

            #Layer7: Fully Connected. Input = 256. Output = 164.
            fc2_W  = tf.Variable(tf.truncated_normal(shape=(256, 164), mean = mu, stddev = sigma))
            fc2_b  = tf.Variable(tf.zeros(164))
            fc2    = tf.matmul(fc1, fc2_W) + fc2_b
            fc2    = tf.nn.relu(fc2)

            #Layer8: Fully Connected. Input = 164. Output = 105.
            fc3_W  = tf.Variable(tf.truncated_normal(shape=(164, 105), mean = mu, stddev = sigma))
            fc3_b  = tf.Variable(tf.zeros(105))
            fc3    = tf.matmul(fc2, fc3_W) + fc3_b
            fc3    = tf.nn.relu(fc3)

            #Layer9: Fully Connected. Input = 105. Output = 67.
            fc4_W  = tf.Variable(tf.truncated_normal(shape=(105, 67), mean = mu, stddev = sigma))
            fc4_b  = tf.Variable(tf.zeros(67))
            fc4 = tf.matmul(fc3, fc4_W) + fc4_b
            fc4    = tf.nn.relu(fc4)

            #Layer10: Fully Connected. Input = 67. Output = n_classes.
            fc5_W  = tf.Variable(tf.truncated_normal(shape=(67, n_classes), mean = mu, stddev = sigma))
            fc5_b  = tf.Variable(tf.zeros(n_classes))
            logits = tf.matmul(fc4, fc5_W) + fc5_b
            return logits
```

Training Model

Most of the code used to train my model is in the seventh cell of the notebook. At first, I concentrated mainly on a 10-epoch model, to speed development. Later I extended that to 20 epochs and of course further training could improve things, but the desired result was obtained with a 20 epoch model. In the Appendix you can see I tried batch sizes of 50, 105, and 210; with the smallest size yielding early improvements over the larger sizes but with the overall result rather unchanged between the three (see, for example, trial runs 13, 14, and 15 in the appendix).

In terms of learning rate, I tried 0.01, 0.001, and 0.0001, with the middle value giving marked improvements over the other two rates (see trial runs 10, 11, and 12 in the Appendix).



```
In [7]: x = tf.placeholder(tf.float32, (None, 32, 32, 1))
        y = tf.placeholder(tf.int32, (None))
        one_hot_y = tf.one_hot(y, n_classes)
        rate = 0.001
        logits = LeNet(x)
        cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, one_hot_y)
        loss_operation = tf.reduce_mean(cross_entropy)
        optimizer = tf.train.AdamOptimizer(learning_rate = rate)
        training_operation = optimizer.minimize(loss_operation)
        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})
                total_accuracy += (accuracy * len(batch_x))
            return total_accuracy / num_examples

        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            print("Training...")
            print()
            for i in range(EPOCHS):
                X_train, y_train = shuffle(X_train, y_train)
                for offset in range(0, n_train, 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})
                validation_accuracy = evaluate(X_valid, y_valid)
                print("EPOCH {}; ".format(i+1)+ "  Validation Accuracy = {:.3f}".format(validation_accuracy))
            saver.save(sess, './lenetV4norm')
            print("Model saved")

        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))
```

```
conv1a
(?, 30, 30, 10)
conv1b
(?, 15, 15, 10)
conv2
(?, 11, 11, 16)
conv2b
(?, 5, 5, 16)
fc0
(?, 400)
Training...

EPOCH 1;   Validation Accuracy = 0.863
EPOCH 2;   Validation Accuracy = 0.914
EPOCH 3;   Validation Accuracy = 0.926
EPOCH 4;   Validation Accuracy = 0.917
EPOCH 5;   Validation Accuracy = 0.933
EPOCH 6;   Validation Accuracy = 0.933
EPOCH 7;   Validation Accuracy = 0.941
EPOCH 8;   Validation Accuracy = 0.946
EPOCH 9;   Validation Accuracy = 0.952
EPOCH 10;   Validation Accuracy = 0.952
EPOCH 11;   Validation Accuracy = 0.944
EPOCH 12;   Validation Accuracy = 0.927
EPOCH 13;   Validation Accuracy = 0.959
EPOCH 14;   Validation Accuracy = 0.928
EPOCH 15;   Validation Accuracy = 0.948
EPOCH 16;   Validation Accuracy = 0.931
EPOCH 17;   Validation Accuracy = 0.963
EPOCH 18;   Validation Accuracy = 0.944
EPOCH 19;   Validation Accuracy = 0.941
EPOCH 20;   Validation Accuracy = 0.968
Model saved
Test Accuracy = 0.935
```

Solution Approach

As mentioned previously, my approach was iterative. Some details are shown in Appendix A (although it is not an exhaustive list of what I attempted, it is rather an exhaustive list of my notes!). I started with the Lenet architecture as it was reviewed in lectures before this project, was simple, well understood, and produced adequate results with fine tuning.

Overall as can be seen on the previous page, my validation accuracy was 96.8% and my test accuracy was 93.5%. Furthermore (further on in this document) you will see my downloaded-images accuracy was 100% with strong correlation between the correct answer and the programs' answer.

The first problem I had with my architecture was that I had set the learning rate too low; of course I changed that to an improved learning rate as mentioned previously.

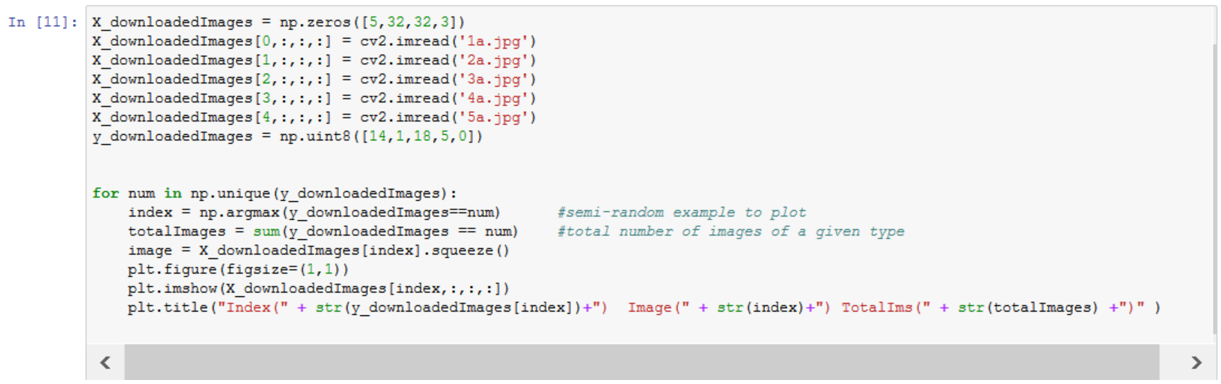
I experimented with red, green, blue, grey, and RGB architectures. Eventually I used a grey, normalized pre-processing scheme as mentioned previously.

I experimented with the number of layers included and was able to improve validation accuracy by a few percentage points with that.

If I was to go further, I would experiment with dropout, using an RGB+greyscale pre-processing, and maybe add a few more layers. As-is I felt the architecture was producing adequate results.

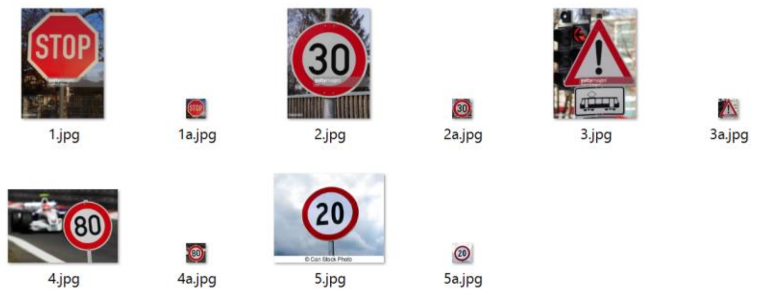
Testing New Images

In the eighth cell, I load in new images and show them. The code is shown below.



```
In [11]: X_downloadedImages = np.zeros([5,32,32,3])
         X_downloadedImages[0,:,:,:] = cv2.imread('1a.jpg')
         X_downloadedImages[1,:,:,:] = cv2.imread('2a.jpg')
         X_downloadedImages[2,:,:,:] = cv2.imread('3a.jpg')
         X_downloadedImages[3,:,:,:] = cv2.imread('4a.jpg')
         X_downloadedImages[4,:,:,:] = cv2.imread('5a.jpg')
         y_downloadedImages = np.uint8([14,1,18,5,0])


         for num in np.unique(y_downloadedImages):
             index = np.argmax(y_downloadedImages==num)      #semi-random example to plot
             totalImages = sum(y_downloadedImages == num)    #total number of images of a given type
             image = X_downloadedImages[index].squeeze()
             plt.figure(figsize=(1,1))
             plt.imshow(X_downloadedImages[index,:,:,:])
             plt.title("Index(" + str(y_downloadedImages[index])+")  Image(" + str(index)+") TotalIms(" + str(totalImages) +")" )
```

The images are shown, before and after cropping them in an external program. ~~Other than cropping and resizing, no further work was done on the images as downloaded.~~



New comments, about images and how they might be challenging: to be fair, the images that I chose were not very challenging, and were chosen because I thought my network would work on them and didn't want an oddball image to cause me to question the system architecture or implementation when, in reality, the problem might be in the image.

Since I'd proven the architecture to be able to recognize all of the images with a great deal of confidence, I decided (with somewhat less confidence) to try a new experiment – feeding in more challenging images and determine what the system could do with those images.

To that end I post-processed the images to be 25% as bright as the original image and 400% as bright as the original image, to show what the effects of underexposing or overexposing the original image would be. Underexposing or overexposing an image is a common failure mode, which can be caused by a variety of factors.   An example of the original, cropped/resized, underexposed, and overexposed images are below. All three can still be plainly seen by eye. These were intended to be challenging but as can be seen below were all clearly still well recognized.

2.jpg

2a.jpg

2b.jpg

2c.jpg

Pre-processing, predicting, and analyzing the quality of predictions were done in cells 9-12. The program was able to successfully predict all ~~5~~ 15 images, which was not surprising, but was able to do so with a large amount of confidence, as can be seen by the output of cell 12, as shown below. In fact for unknown reasons the confidence went up.

```
In [12]: for i in range(15):
    print("Answer: " + str(y_downloadedImages[i]))
    print("   Predicted Answer: {:2.0f}".format(predictions[i,0]) +" With Probability {:3.3f}".format(100*probs[i,0]) +"%")
    print("         2nd Answer: {:2.0f}".format(predictions[i,1]) +" With Probability {:3.3f}".format(100*probs[i,1]) +"%")
    print("         3rd Answer: {:2.0f}".format(predictions[i,2]) +" With Probability {:3.3f}".format(100*probs[i,2]) +"%")
    print("         4th Answer: {:2.0f}".format(predictions[i,3]) +" With Probability {:3.3f}".format(100*probs[i,3]) +"%")
    print("         5th Answer: {:2.0f}".format(predictions[i,4]) +" With Probability {:3.3f}".format(100*probs[i,4]) +"%")
```

```
Answer: 14
   Predicted Answer: 14 With Probability 100.000%
         2nd Answer:  3 With Probability 0.000%
         3rd Answer: 34 With Probability 0.000%
         4th Answer: 30 With Probability 0.000%
         5th Answer: 20 With Probability 0.000%
Answer: 14
   Predicted Answer: 14 With Probability 100.000%
         2nd Answer:  3 With Probability 0.000%
         3rd Answer: 34 With Probability 0.000%
         4th Answer: 20 With Probability 0.000%
         5th Answer: 30 With Probability 0.000%
Answer: 14
   Predicted Answer: 14 With Probability 100.000%
         2nd Answer:  3 With Probability 0.000%
         3rd Answer: 34 With Probability 0.000%
         4th Answer: 20 With Probability 0.000%
         5th Answer: 30 With Probability 0.000%
Answer: 1
   Predicted Answer:  1 With Probability 100.000%
         2nd Answer:  2 With Probability 0.000%
         3rd Answer: 31 With Probability 0.000%
         4th Answer:  0 With Probability 0.000%
         5th Answer: 25 With Probability 0.000%
Answer: 1
   Predicted Answer:  1 With Probability 100.000%
         2nd Answer:  2 With Probability 0.000%
         3rd Answer: 31 With Probability 0.000%
         4th Answer:  0 With Probability 0.000%
         5th Answer: 30 With Probability 0.000%
Answer: 1
   Predicted Answer:  1 With Probability 100.000%
         2nd Answer: 31 With Probability 0.000%
         3rd Answer:  2 With Probability 0.000%
         4th Answer:  0 With Probability 0.000%
         5th Answer: 30 With Probability 0.000%
Answer: 18
   Predicted Answer: 18 With Probability 100.000%
         2nd Answer: 28 With Probability 0.000%
         3rd Answer: 26 With Probability 0.000%
         4th Answer: 27 With Probability 0.000%
         5th Answer: 20 With Probability 0.000%
Answer: 18
   Predicted Answer: 18 With Probability 100.000%
         2nd Answer: 26 With Probability 0.000%
         3rd Answer: 28 With Probability 0.000%
         4th Answer: 27 With Probability 0.000%
         5th Answer: 20 With Probability 0.000%
Answer: 18
   Predicted Answer: 18 With Probability 100.000%
         2nd Answer: 28 With Probability 0.000%
         3rd Answer: 20 With Probability 0.000%
         4th Answer: 38 With Probability 0.000%
         5th Answer: 27 With Probability 0.000%
Answer: 5
   Predicted Answer:  5 With Probability 100.000%
         2nd Answer:  2 With Probability 0.000%
         3rd Answer:  3 With Probability 0.000%
         4th Answer: 10 With Probability 0.000%
         5th Answer:  4 With Probability 0.000%
Answer: 5
   Predicted Answer:  5 With Probability 100.000%
         2nd Answer:  2 With Probability 0.000%
         3rd Answer:  3 With Probability 0.000%
         4th Answer: 10 With Probability 0.000%
         5th Answer: 23 With Probability 0.000%
Answer: 5
   Predicted Answer:  5 With Probability 100.000%
         2nd Answer:  2 With Probability 0.000%
         3rd Answer:  3 With Probability 0.000%
         4th Answer: 10 With Probability 0.000%
         5th Answer: 23 With Probability 0.000%
Answer: 0
   Predicted Answer:  0 With Probability 100.000%
         2nd Answer:  4 With Probability 0.000%
         3rd Answer:  8 With Probability 0.000%
         4th Answer:  1 With Probability 0.000%
         5th Answer:  5 With Probability 0.000%
```

~~As can be seen, for each of the five images the prediction capability was over 96%, and the only image with less than 99.8% prediction confidence was (surprisingly, to me at least) the stop sign, which had identifier '14'. For that, the program though there was a 2.3% chance the sign was a 38: Keep right, and a 1.4% chance of 18: General Caution. This was unexpected but with such low probabilities I chose to ignore them and move on.~~

Re-submission: all images were predicted with high confidence this time around, even though the intent was to provide a more challenging set of images.

Conclusions

I don't believe I've come up with anything particularly noteworthy here. Nevertheless, I had a very simple, easy to train model come up with high validation and test accuracy. Further work could move the accuracy still further but I think I will learn more by moving on to the next project.

Comments and criticism are very welcome. I'm new to python and (as I'm sure you can tell) I don't yet have a full grasp of the language.

Appendix A (Summary of most of the runs

	Inputs				Architecture			Preproc.																					
Trial #	Batch	Sigma	Mu	Rate	Version	Height	Pooling	Color	Normalized	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5	Epoch 6	Epoch 7	Epoch 8	Epoch 9	Epoch 10	Epoch 11	Epoch 12	Epoch 13	Epoch 14	Epoch 15	Epoch 16	Epoch 17	Epoch 18	Epoch 19	Epoch 20
1	210	0.1	0	0.0001	V1	8	5x5	Red	N	0.13	0.21	0.28	0.33	0.38	0.44	0.47	0.50	0.54	0.56										
2	210	0.1	0	0.0001	V1	8	5x5	Green	N	0.12	0.22	0.34	0.43	0.51	0.56	0.59	0.63	0.66	0.68										
3	210	0.1	0	0.0001	V1	8	5x5	Blue	N	0.13	0.25	0.33	0.41	0.48	0.55	0.60	0.64	0.68	0.71										
4	210	0.1	0	0.0001	V1	8	5x5	Grey	N	0.05																			
5	210	0.1	0	0.0001	V1	8	5x5	Grey	N	0.13	0.22	0.34	0.42	0.48	0.54	0.58	0.62	0.65	0.67										
6	210	0.1	0	0.0001	V1	8	5x5	Grey	N	0.10	0.23	0.34	0.42	0.49	0.56	0.61	0.65	0.68	0.71										
7	210	0.1	0	0.0001	V1	8	5x5	Grey	N	0.21	0.37	0.48	0.56	0.63	0.67	0.70	0.73	0.75	0.77										
8	210	0.2	0	0.0001	V1	8	5x5	Grey	N	0.07	0.11	0.13	0.10	0.06	0.05	0.05	0.05	0.05	0.05										
9	210	0.1	0	0.0001	V1	8	5x5	Grey	N	0.19	0.32	0.42	0.50	0.55	0.59	0.63	0.66	0.70	0.72										
10	210	0.1	0	0.0001	V1	8	5x5	Grey	N	0.06	0.14	0.26	0.38	0.47	0.53	0.58	0.62	0.66	0.67										
11	210	0.1	0	0.001	V2	8	5x5	Grey	N	0.67	0.78	0.82	0.82	0.84	0.85	0.85	0.87	0.86	0.88										
12	210	0.1	0	0.01	V2	8	5x5	Grey	N	0.63	0.76	0.77	0.78	0.82	0.77	0.82	0.84	0.84	0.84										
13	210	0.1	0	0.001	V2	8	5x5	Grey	N	0.67	0.80	0.82	0.85	0.87	0.88	0.87	0.88	0.88	0.89										
14	105	0.1	0	0.001	V2	8	5x5	Grey	N	0.75	0.83	0.86	0.87	0.89	0.89	0.90	0.90	0.90	0.90										
15	50	0.1	0	0.001	V3	8	5x5	Grey	N	0.80	0.86	0.88	0.89	0.89	0.90	0.90	0.90	0.90	0.89										
16	50	0.1	0	0.001	V4	10	5x5	Grey	N	0.78	0.85	0.86	0.90	0.91	0.88	0.91	0.91	0.91	0.92										
17	50	0.1	0	0.001	V5	10	5x5	RGB	N	0.66	0.75	0.81	0.83	0.83	0.84	0.86	0.85	0.88	0.86										
18	50	0.1	0	0.001	V5	10	5x5	Grey	N	0.72	0.79	0.81	0.84	0.84	0.86	0.87	0.85	0.86	0.86										
19	50	0.1	0	0.001	V4	10	5x5	Grey	N	0.78	0.83	0.86	0.87	0.86	0.88	0.87	0.89	0.89	0.88										
20	50	0.1	0	0.001	V4	10	5x5	Grey	N	0.80	0.86	0.86	0.87	0.88	0.88	0.87	0.89	0.88	0.88										
21	50	0.1	0	0.001	V5	10	5x5	Grey	N	0.66	0.78	0.80	0.83	0.85	0.82	0.85	0.87	0.84	0.84										
22	50	0.1	0	0.001	V4	10	5x5	Grey	Y	0.74	0.80	0.85	0.87	0.87	0.87	0.87	0.87	0.89	0.90										
23	50	0.1	0	0.001	V4	12	5x5	Grey	Y	0.75	0.84	0.84	0.88	0.88	0.89	0.87	0.90	0.88	0.88										
24	50	0.1	0	0.001	V6	12	5x5	Grey	Y	0.81	0.86	0.89	0.91	0.89	0.91	0.92	0.93	0.90	0.92										
25	50	0.1	0	0.001	V6	12	5x5	Grey	N	0.78	0.84	0.87	0.87	0.91	0.88	0.90	0.88	0.89	0.89										
26	50	0.1	0	0.001	V7	12	5x5	Grey	Y	0.83	0.85	0.85	0.85	0.90	0.88	0.91	0.90	0.89	0.91										
27	50	0.1	0	0.001	V7	12	5x5	Grey	Y	0.80	0.84	0.88	0.88	0.89	0.90	0.89	0.90	0.89	0.89										
28	50	0.1	0	0.001	V7	12	5x5	Grey	Y	0.84	0.90	0.86	0.89	0.88	0.90	0.92	0.91	0.91	0.91										
29	50	0.1	0	0.001	V7	16	5x5	Grey	Y	0.77	0.84	0.88	0.88	0.89	0.90	0.89	0.90	0.90	0.92										
30	50	0.1	0	0.001	V7	16	5x5	Grey	Y	0.77	0.84	0.87	0.88	0.90	0.89	0.91	0.92	0.90	0.92										
31	50	0.1	0	0.001	V7	8	5x5	Grey	Y	0.81	0.85	0.86	0.88	0.90	0.87	0.91	0.91	0.92	0.90										
32	50	0.1	0	0.001	V7	8	5x5	Grey	Y	0.78	0.83	0.88	0.89	0.88	0.89	0.91	0.91	0.91	0.91										
33	50	0.1	0	0.001	V7	8	3x3	Grey	Y	0.80	0.86	0.87	0.89	0.90	0.92	0.92	0.92	0.94	0.91										
34	50	0.1	0	0.001	V7	8	3x3	Grey	Y	0.77	0.85	0.87	0.89	0.88	0.90	0.90	0.91	0.90	0.90	0.92	0.91	0.89	0.91	0.92	0.92	0.92	0.93	0.92	0.93
35	50	0.1	0	0.001	V7	10	3x3	Grey	Y	0.82	0.84	0.84	0.90	0.90	0.89	0.91	0.87	0.90	0.89	0.91	0.89	0.92	0.90	0.93	0.92	0.92	0.91	0.92	0.94