

Traffic Sign Recognition

Contents

1	Reference to Project Code	2
2	Data Set Summary & Exploration	2
2.1	Data Set Summary.....	2
2.2	Exploratory Visualization	3
3	Design and Test a Model Architecture.....	4
3.1	Image preprocessing.....	4
3.2	Training, Validation and Test Data Sets	5
4	Final Model Architecture	7
5	Training Process	8
5.1	Initializing Constants and Variables	8
5.2	Defining the Training Pipeline (code cell #29)	8
5.3	Defining the Evaluation Model (code cell #30).....	9
5.4	Running the Training Process (code cell #31)	9
6	Approach for Final Solution	11
7	Testing the Model on New Images	12
7.1	New Images from the Web	12
7.2	Model's Predictions on New Images.....	13
7.3	Softmax Probabilities on New Images	14

1 Reference to Project Code

Here is a link to my [project code](#).

2 Data Set Summary & Exploration

2.1 Data Set Summary

The code for this step is contained in the **7th code cell** of the IPython notebook.

I used the **Numpy library** to calculate summary statistics of the traffic signs data set:

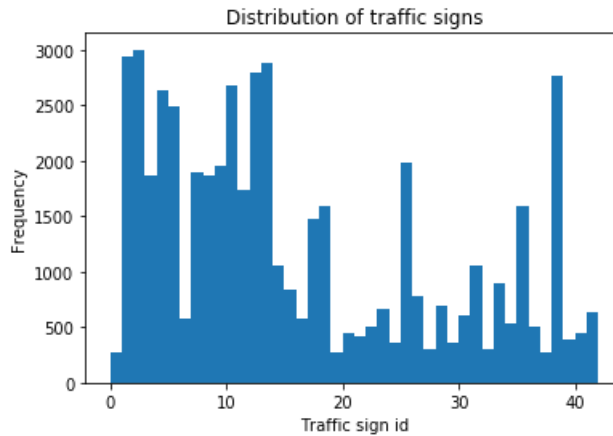
Size of training set	34,799
Size of validation set	4,410
Size of test set	12,630
Shape of sign images	32 x 32 x 3 (32 by 32 pixels with 3 color channels)
Number of unique classes / labels	43

Note: the number of unique classes was calculated using the Numpy “unique” function after concatenating all sets together (see 7th code cell).

2.2 Exploratory Visualization

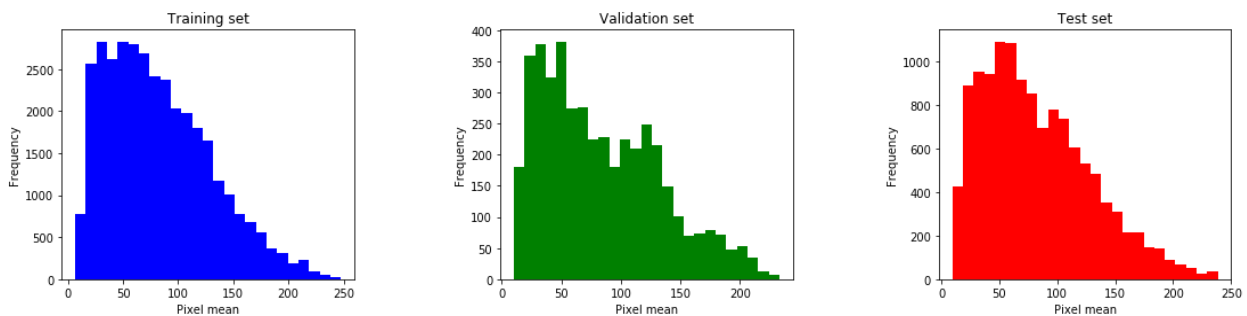
The code for this step is contained in the 9th and 10th code cells of the IPython notebook.

The following histogram shows the distribution of images **by traffic sign labels** in the training set:



→ We can see that images are **unevenly distributed**: for example, there are significantly more examples of signs with labels around 4, 12 and 38.

The following bar charts show the distribution of images by **mean of pixels** in each set (training, validation and test). We obtained them by calculating the mean of pixels for all images in each set (*see cell code #10*).



→ We can see that, for each set, the majority of images have a mean around 60 units of colors index.

It is delicate to interpret this metric without further investigation but we can safely say that the higher the mean, the brighter the image; which sounds logical given that a white image is such as $RGB = \{255, 255, 255\}$.

When looking at sample images, it turns out that a mean of 60 corresponds to **rather dark** images.

3 Design and Test a Model Architecture

3.1 Image preprocessing

The code for this step is contained in the following code cells of the IPython notebook:

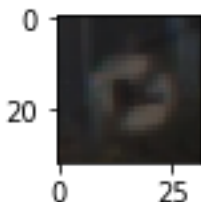
- Code cell #12 contains the **definition** of grayscale and normalization functions:
 - o image_gray converts image sets to **grayscale**.
 - o image_ab **normalizes** pixels to a [a, b] interval, more specifically [0.1, 0.9].
 - o image_z **normalizes** pixels as a zero mean and a unit variance distribution.
- Code cell #13 allows **selecting** grayscale and normalization functions and **running** them on the training, validation and test sets.
- Code cell #14 is there to **check** the aforementioned pre-processing pipeline with a sample image.

After testing multiple scenarios for preprocessing, iterations I made the following choices:

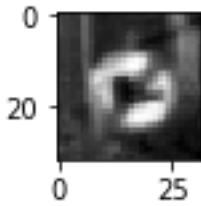
- I decided to convert the image to **grayscale** for two reasons:
 - o It made the model significantly **faster** (one color channel instead of three) without a major loss in accuracy.
 - o As explained above, a majority of images in our data sets is rather dark. I noticed that the grayscale provided an efficient solution for dark images, as it better separated regions with **different color nuances**.
- I decided to use the **[a, b] normalization** to make the model less sensitive to large numbers (e.g. color index > 200).

Here is an example of a traffic sign image before and after pre-processing:

- Example of "End of no passing by vehicles over 3.5 metric tons" sign before pre-processing:



- Same "End of no passing by vehicles over 3.5 metric tons" sign after pre-processing:



This example illustrates our statement mentioned above: surprisingly, turning images to grayscale helps better **separating color patterns**.

3.2 Training, Validation and Test Data Sets

We were already provided with three distinct data sets: **training**, **validation** and **test**. So there was no need to split the training set between training and validation. The code was still prepared in code cell #2 but disabled.

As mentioned above, the different sets were as following:

- Number of **training** examples: 34,799
- Number of **validation** examples: 4,410
- Number of **testing** examples: 12,630

I decided to **augment the training set** as I noticed a **gain by 3% in validation accuracy**. The augmentation pipeline is the following:

- Code cell #19: to randomly **select images** from the training set for augmentation.
 - o 40% of the training data for **rotation**
 - o 40% of the training data for **translation**
- Code cell #20: to **visualize** a sample image **before rotation**.
- Code cell #21: to **visualize** a sample image **before translation**.
- Code cell #22: to **run augmentation** using the two following functions
 - o `image_translate` **translates** pixels in images with a randomly generated factor of 20% maximum, both vertically and horizontally.

- image_rotate **rotates** images with a randomly selected angle between 5 and 10 degrees, either positive (rotation towards right direction) or negative (rotation towards left direction).
- Code cell #23: to check the same sample image from code cell #19 **after rotation**.
- Code cell #24: to check the same sample image from code cell #20 **after translation**.
- Code cells #25 and 26: to run the same **preprocessing** functions on augmented images as the original training set, and visualize the output.
- Code cell #27: to add augmented data to the training set.
 - After multiple tests I decided to only retain **rotated images**, as translated images did not help improving accuracy.
 - I used the Numpy concatenate function.

Below are sample images to illustrate the data augmentation:

- "Speed limit (80km/h)" sign:



- "Pedestrians" sign:



As mentioned above, the original training set had **34,799** examples. The augmented training set had **48,719** examples (40% more).

4 Final Model Architecture

The code for my final model is located in **code cell #17** of the ipython notebook. It features an alternative version of the **LeNet convolutional network**, consisting of the following layers:

Layer #	Sub Layer	Description / Comment	Output
0	Input	32x32x1 RGB images (32x32x3 images converted to grayscale)	N/A
1	Convolution 5x5	5x5 filter, 1x1 stride, valid padding. Weights and biases generated using a truncated normal distribution with ($\mu=0$, $\sigma=0.1$). Main function used: tf.nn.conv2d	28x28x6
1	Activation	ReLU	28x28x6
1	Max pooling	2x2 stride, valid padding. Main function used: tf.nn.max_pool	14x14x6
2	Convolution 5x5	5x5 filter, 1x1 stride, valid padding. Weights and biases generated using a truncated normal distribution with ($\mu=0$, $\sigma=0.1$). Main function used: tf.nn.conv2d	10x10x6
2	Activation	ReLU. Main function used: tf.nn.relu	10x10x6
2	Max pooling	2x2 stride, valid padding. Main function used: tf.nn.max_pool	5x5x6
3	Flattening	Main function used: flatten class from tensorflow.contrib.layers.	400
4	Fully connected	Matrix multiplication. Weights and biases generated using a truncated normal distribution with ($\mu=0$, $\sigma=0.1$). Main function used: tf.matmul	120
4	Activation	ReLU. Main function used: tf.nn.relu	120
5	Fully connected	Matrix multiplication. Weights and biases generated using a truncated normal distribution with ($\mu=0$, $\sigma=0.1$). Main function used: tf.matmul	84
5	Activation	ReLU. Main function used: tf.nn.relu	84
6	Fully connected	Matrix multiplication. Weights and biases generated using a truncated normal distribution with ($\mu=0$, $\sigma=0.1$). Main function used: tf.matmul	43

5 Training Process

The code for training the model is the following:

5.1 Initializing Constants and Variables

- Code cell #16: to define the following hyper-parameters.
 - o Number of **epochs**: set to 10 after multiple test iterations
 - o **Batch size**: 128. That was the maximum we could use, given that the model was run on a **CPU** instance.
- Code cell #28: to instantiate **placeholders** for TensorFlow input variables.
 - o x for **images**
 - o y for **labels**

5.2 Defining the Training Pipeline (code cell #29)

This training pipeline will be run at **each epoch**, for **each batch** during training (see 5.4).

- Defining the following hyper-parameters:
 - o **Learning rate**: set to 0.001 after multiple test iterations
 - o Beta for **L2 regularization**: set to 0.01
- Generating **logits** from the neural network:

```
logits = LeNet(x)
```
- Calculating the **cross-entropy** against labels:

```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, one_hot_y)
```
- Applying **L2 regularization** to prevent overfitting by artificially penalizing large weights:

```
l2_regul = cross_entropy + l2_beta * tf.nn.l2_loss(logits)
```
- Calculating **loss**:

```
loss_operation = tf.reduce_mean(l2_regul)
```
- Setting **loss optimizer** to Adam optimizer:


```
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
```

- Running **loss optimization**:

```
training_operation = optimizer.minimize(loss_operation)
```

5.3 Defining the Evaluation Model (code cell #30)

Code cell #30 builds the **evaluation model**, to be run on the **validation set**. This evaluation model will be run at **each epoch** during training (see 5.4).

- Extracting **batches** (128 images with their 128 labels) from the data set.
- For each batch, calculating the **accuracy of predictions** and accumulating the total accuracy:

```
accuracy = sess.run(accuracy_operation, feed_dict={x:batch_x, y:batch_y})
total_accuracy += (accuracy * len(batch_x))
```

The accuracy operation is defined as following:

- o For each image in the batch, extracting the **prediction** and comparing it to the actual "one-hot" encoded label. This operation is handled by a single line of code:

```
correct_prediction = tf.equal(tf.argmax(logits,1),tf.argmax(one_hot_y,1))
```

*Note: the `tf.equal` function returns a **Boolean value** "element-wise", i.e. for each individual image.*

*Note: the prediction is defined as the label returned in logits with the **highest probability** (`tf.argmax` function).*

- o Computing the **mean of Boolean values** over the entire batch, after casting them as float numbers:

```
accuracy_operation=tf.reduce_mean(tf.cast(correct_prediction,tf.float32))
```

- Finally, calculating the weighted average total accuracy:

```
return total_accuracy / num_examples
```

5.4 Running the Training Process (code cell #31)

For each of the **10 epochs**:

- **Shuffling** the data set, using the shuffle class from the sklearn package.
- Extracting batches (128 images with their 128 labels) from the data set.
- For each batch, running the **training pipeline**

```
sess.run(training_operation, feed_dict={x: batch_x, y: batch_y})
```

- o This step leverages the previously defined "training_operation" tensor.
 - o The training_operation tensor is instantiated using images from the batch with their labels (feed_dict dictionary).
- Once all batches are processed, the trained model is evaluated against the **validation set**, using the previously defined "**evaluate**" function:

```
validation_accuracy = evaluate(X_valid_pre, y_valid)
```

Finally, the model is saved for future use, using the **tf.train.Saver()** procedure.

6 Approach for Final Solution

As explained in the previous paragraph, the code for calculating the accuracy of the model is located in code cell #30 of the Ipython notebook ("**evaluate**" function).

My final model results were:

- Final accuracy on the **validation set** after 10 epochs: 96.40%
- Accuracy on the **five images** downloaded from the web: 100.00%
- Accuracy on the provided **test set** (12,630 examples): 94.05%






I chose an iterative approach:

- After looking for models on the web, I decided to use the **LeNet** model.
- The initial architecture was static. I improved it the following way:
 - o By updating the **initial image depth** according to the prior grayscale conversion: 3 color channels if no conversion, 1 single color channel after conversion.
 - o By updating the final **number of classes** to account for new classes if needed.
- In early iterations, I noticed a **high validation accuracy** (>92%) with a **low test accuracy** (<80%), which was a sign of **over-fitting**. In order to mitigate this phenomenon, I decided to add a L2 regularization layer to penalize large weights. The code is explained in paragraph 5.2.
- The batch size was lowered to 128 to make it computable **without using a GPU instance**.

7 Testing the Model on New Images

7.1 New Images from the Web

Here are five German traffic signs that I found on the web:

Image	Potential Challenge
	<p>The original image is rectangular. After resizing to 32x32, shapes will be significantly modified (oval vs. circle).</p> <p>Otherwise, there does not seem to be any noticeable difficulty.</p>
	<p>The image is again rectangular.</p> <p>Also, the stop sign is not centered, when comparing to the first image.</p>
	<p>Besides the rectangular shape, the picture includes additional features (lower scripted sign) which might affect the model's ability to recognize the main sign (children crossing).</p>
	<p>The original image is rectangular. Resizing to a square image will significantly modify shapes (oval vs. circle).</p> <p>Also, the picture is not exactly front-facing.</p>
	<p>Here again, the image is rectangular.</p> <p>In addition, the picture is taken from a higher position, which significantly alter its shape.</p>

7.2 Model's Predictions on New Images






The code for making predictions on my final model is located in **code cell #36** of the Ipython notebook:

```
# Running predictions on images downloaded on the web
with tf.Session() as sess:
    # Reloading saved model
    saver.restore(sess, tf.train.latest_checkpoint('.'))

    # Running model on images
    softmax = tf.nn.softmax(logits)

    # Getting prediction with highest probability (k = 1 in tf.nn.top_k)
    output = sess.run(softmax, feed_dict={x: X_test_def})
    values, indices = tf.nn.top_k(output, 1)
    probas = sess.run(values)
    predictions = sess.run(indices)
```

Here are the results of the prediction:

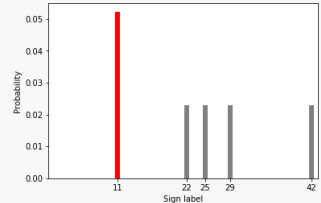

Image Label	Image	Model Prediction
Priority Road		Priority Road
Stop		Stop
Children crossing		Children crossing
Speed limit (50 km/h)		Speed limit (50 km/h)
Keep right		Keep right

The model was able to correctly guess **all the five traffic signs**, which gives an accuracy of **100%**. This compares favorably to the accuracy on the test set of **94.05%** (see 6).

7.3 Softmax Probabilities on New Images

As mentioned here above, the code for making predictions on my final model is located in **code cell #36** of the Ipython notebook. It is re-executed in **code cell #42** to visualize softmax probabilities with **bar charts**.

The following table shows the **top 5 predicted labels** with their corresponding probabilities. The last column includes a bar chart to **visualize** probabilities with respect to each other.

Image Label	Image	Label Id	Top 5 Labels Predicted	Top 5 Corresponding Softmax Probabilities (rounded)	Bar Chart of Probabilities by Labels
Priority Road		11	[11, 22, 42, 29, 25]	[0.052, 0.023, 0.023, 0.023, 0.023]	
Stop		14	[14, 5, 39, 0, 30]	[0.029, 0.028, 0.024, 0.024, 0.024]	
Children crossing		28	[28, 26, 3, 12, 23]	[0.040, 0.024, 0.023, 0.023, 0.023]	
Speed limit (50 km/h)		2	[2, 24, 20, 32, 1]	[0.062, 0.023, 0.023, 0.023, 0.023]	
Keep right		38	[38, 10, 32, 2, 27]	[0.050, 0.023, 0.023, 0.023, 0.023]	

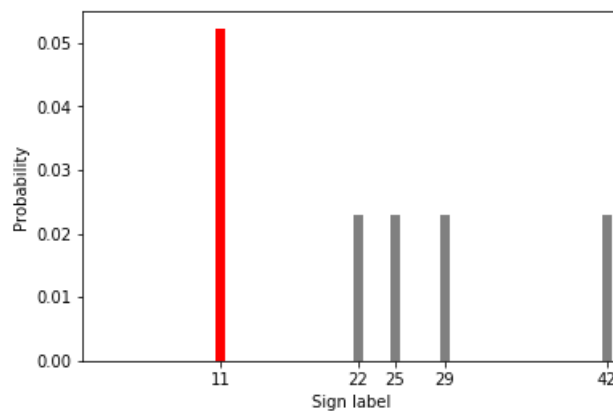
*Note: predicted labels are sorted by descending order of probabilities. So the **first element** is the best prediction.*

Note: bar chart can be viewed with a higher resolution here below.

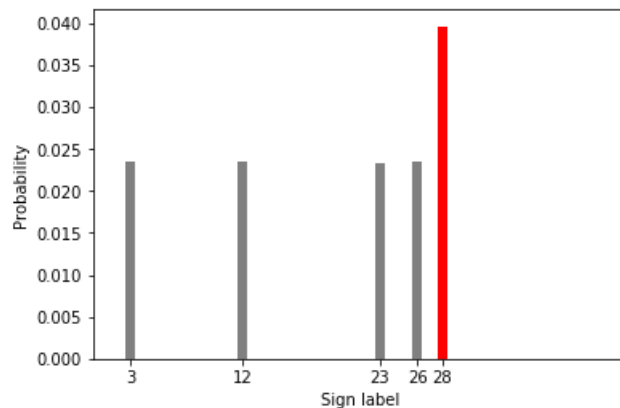
The model does not return high probabilities for each of the five images, 6.2% being the overall maximum (probability for the "Speed limit (50 km/h)" sign. So it looks like it is **quite uncertain** about its predictions.

That being said, most certain labels are almost always distinguished by the **relative magnitude** of their probabilities compared to other labels:

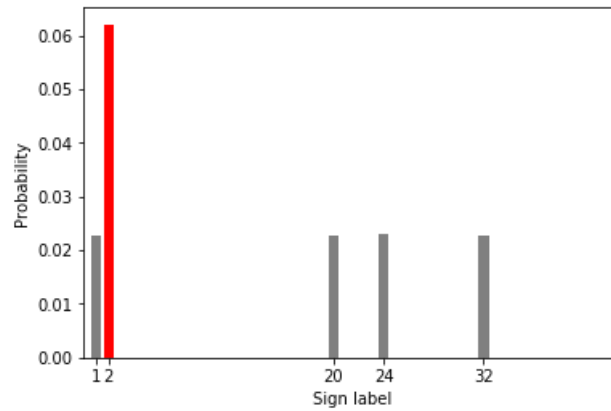
- Priority road sign: **5.2% vs. 2.3%** for the second higher probability.



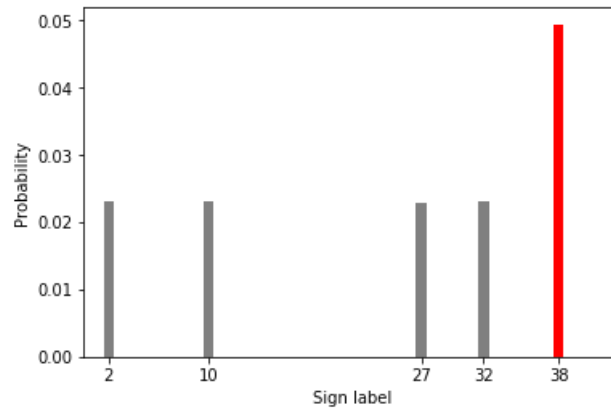
- Children crossing sign: **4.0% vs. 2.4%**.



- Speed limit (50 km/h) sign: **6.2 vs. 2.3%**.



- Keep right sign: **5.0% vs. 2.3%**.



- Only the Stop sign returns a low highest probability: **2.9% vs. 2.8%**.

