fgong
Nov 26 · 13 min read

Term1-P2: Traffic Sign Classifier Project

1. Introduction:

In this project I built a traffic sign recognition classifier using convolutional neural network and tensorflow. Moreover, I fine-tuned the hyper-parameters to achieve, at least, 93% accuracy on Germany Traffic Sign Dataset.

- Please find source code with Github link:
  Traffic_Sign_Classifier.ipynb

- Html version is available: **Traffic_Sign_Classifier.html**
  Note: the html file size is 1.9MB so Github cannot display it. Please download and open it with Chrome or IE in your local.

2. Data Set Summary & Exploration

In this stage, I explored and analyzed the dataset so that it can be better manipulated in this project. I did four steps to calculate summary of the traffic sign in this dataset:

- **Basic Summary**

I use numpy and pandas to compute the basic statistic information, such as number of training samples, number of testing samples, image shape (i.e., np.shape(image)) and unique labels (i.e., np.unique). The output is:

```
Number of training examples = 34799
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43
```

Note that the image shape has 3 dimensions: each image has width = 32 pixels, height = 32 pixels, and 3 color channels (i.e., Red, Green, Blue).

- **Exploratory Visualization**

There are total 43 different types of traffic signs in the dataset. I plotted one sample image for each category as shown in the below. Note that the title is the "*name of traffic sign category*" as a reference.

visualization of traffic sign (one sample for each category)

> **Observation**: *some images are rotated, blurred, distorted or darker than others. For example, image of "Slippery road" on the 6th row to the most right column is even harder for human being to recognize! These factors indeed affect recognition accuracy later and need extra work to be carefully addressed.*

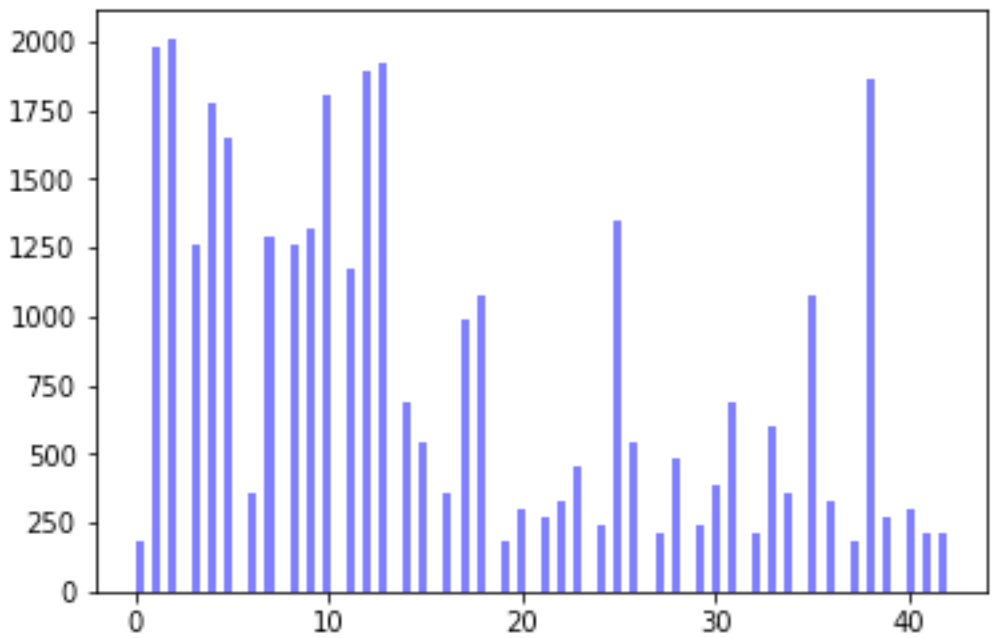
- **Distribution**

I plotted the image distribution of those traffic sign samples to show how many images for each category. The distribution for training set is following:

X: the index of traffic sign category

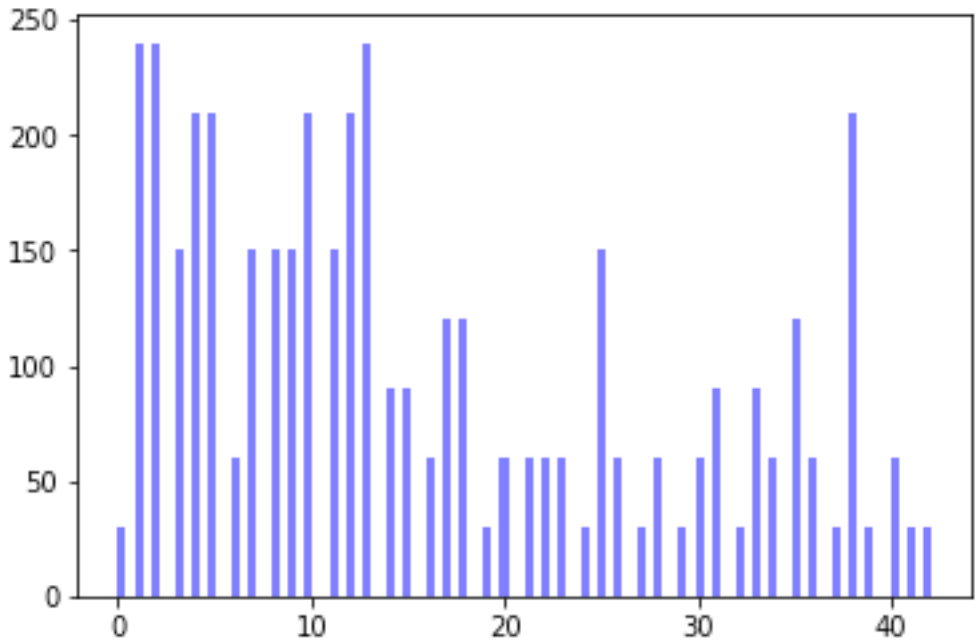
Y: the number of image samples in this category

```
---> Message: distribution of training set
```



I also plotted the distribution of validation set as following:

```
---> Message: distribution of validation set
```



> **Observation**: *both training set and validation set have* ***similar distribution*** *of traffic sign samples. This is vey important, because we train the model on training set, and apply the SAME model on validation set. We implicitly assume both datasets are drawn from the same distribution and share common features.*

- **Calculate total number of images in each category**

I calculated the total number of image samples for each category. In the below, I show a small portion of the summary.

```
0       count := 180.0          traffic sign name := Speed limit (20km/h)
1       count := 1980.0         traffic sign name := Speed limit (30km/h)
2       count := 2010.0         traffic sign name := Speed limit (50km/h)
3       count := 1260.0         traffic sign name := Speed limit (60km/h)
4       count := 1770.0         traffic sign name := Speed limit (70km/h)
5       count := 1650.0         traffic sign name := Speed limit (80km/h)
6       count := 360.0          traffic sign name := End of speed limit (80km/h)
7       count := 1290.0         traffic sign name := Speed limit (100km/h)
8       count := 1260.0         traffic sign name := Speed limit (120km/h)
9       count := 1320.0         traffic sign name := No passing
```

3. Design and Test a Model Architecture

Preprocess the image data

1. **Gray Scale**: As the first step, I use direct computation to convert RGB image with 3 channels into Gray scale image with one single channel.

```
# Y' = 0.299 R + 0.587 G + 0.114 B
def rgb2gray(img):
    return 0.299*img[:,:,0] + 0.587*img[:,:,1] + 0.114*img[:,:,2]
```

Operation to convert RGB image into Grayscale image

Using Grayscale image has multiple advantages:

- *small memory usage*: For each pixel, RGB has 3 channels and each channel needs 8 bit storage, while grayscale use 8 bit only for one channel.
- *color invariance*: Traffic sign classifier is color invariant, that means it only need to detect edges inside the image and color information doesn't help.
- *difficulty of visualization*: grayscale is easier to manipulate and visualize because it has two spatial dimensions and one brightness dimension. On the contrary, RGB has two spatial dimensions and three color dimensions.
- *processing speed*: grayscale image has less data, therefore, making it faster to be processed. In particular, it can save huge amount of time

in video processing if you imagine video to be a long sequence of images.

2. **Scaling or Normalization**: the second step is to perform scaling or normalization on each image. This step changes the range of pixel intensity so all images can have consistent range for pixel values.

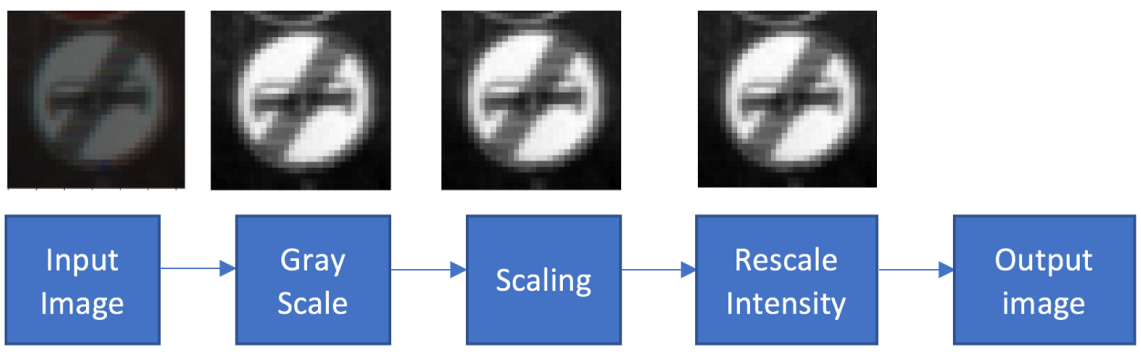
Here we use linear transform “(pixel—128) / 128” on each pixel for simplicity on grayscale images. It can be improved for better results later.

The reason of normalization is following:

- *contrast stretching:* images can have very different contrast due to glare, change of illumination intensity, or etc. It causes images either very bright or too dark, which is difficult to recognize.
- *model stability*: skewed pixel values are harmful, because our model will multiply weights and add bias to these image pixels. If extremely large or small values are involved, both operations can amplify the skewness and cause large error.
- *improve gradient calculation*: model needs to calculate gradients in backward propagation. With skewed pixel values, gradient calculation can be out of control that is a nightmare…

3. **Rescale Intensity**: As an experiment, I found above simple linear transform can be further improved with “exposure.rescale_intensity” method from skimage library. It *uniformly* rescaled image intensity to make pixel values inside consistent range and achieved better results.

Here is the overall flow of our pre-processing.

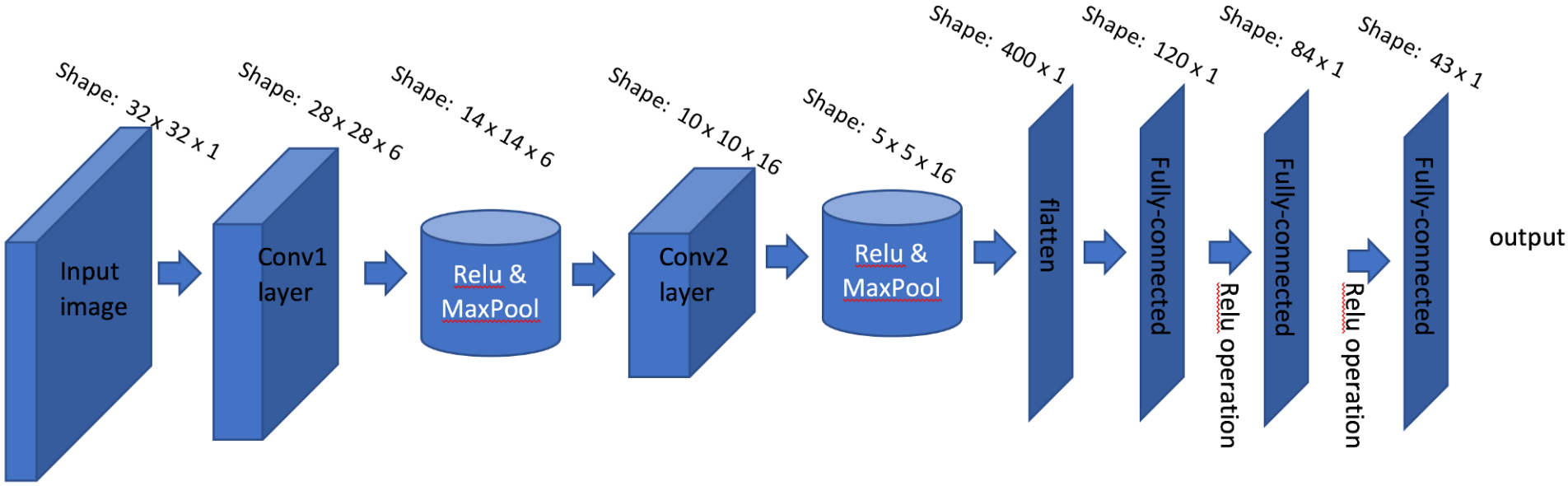


Model architecture

I use a similar model as LeNet introduced in the course. It includes two convolutional layers and three fully-connected layers. The overall architecture is shown in the below. **The shape denotes the dimension of output from this layer. We use "Relu" as our activation function.**

> **Note**:
> *(1) there is **relu** and **max_pool** after each convolutional layer.*
> *(2) there is only **relu** between fully-connected layers.*
> *(3) there is NO operation after linear transform in the last output layer.*
> *(4) **max_pooling** operation will cut image size by half each time because:*
> *- we use 2x2 windows (kernel size = [1, 2, 2, 1])*
> *- move 2 pixels in spatial direction each time(strides=[1, 2, 2, 1]).*

kernel size = [batch_size, height, width, channel]
strides size =[batch_size, height, width, channel]



LeNet-type Model Architecture with output shape size

Train the model

During the training of this model, I use following hyper-parameters:

learning rate = 0.001
EPOCHS = 50
BATCH_SIZE = 256

- **define loss function**: since our estimation and target are arrays, we can use "*soft_max_cross_entropy_with_logits*" to measure their dis-

crepancy. On top of that, the loss function is defined as *mean value* of elements in the cross entropy which is a decent scalar representative of the entire array.

```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y, logits=logits)
loss operation = tf.reduce mean(cross entropy)
```

- **define optimizer**: we use "Adam optimizer" to minimize the loss function using learning rate=0.001. Instead of classical stochastic gradient descent (SGD) algorithm, we use Adam method here for better performance because Adam method update weights iteratively based on training data.

```
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)
```

- **Train the model**: the entire dataset is divided into multiple batches, and each batch contains 256 images. Code of framework is shown below.
  (1) In each epoch, we shuffled the entire training images, such that images in each batch can better represent the entire training dataset.
  (2) the optimizer works on the batches of training images to minimize the loss function (use function "training_operation") to update the weights and bias in our model.
  (3) After optimizer is done for all training data, we apply the model on validation dataset and evaluate the performance of our model.
  (4) The process continues to next epoch iteration until we reach >93% validation accuracy.

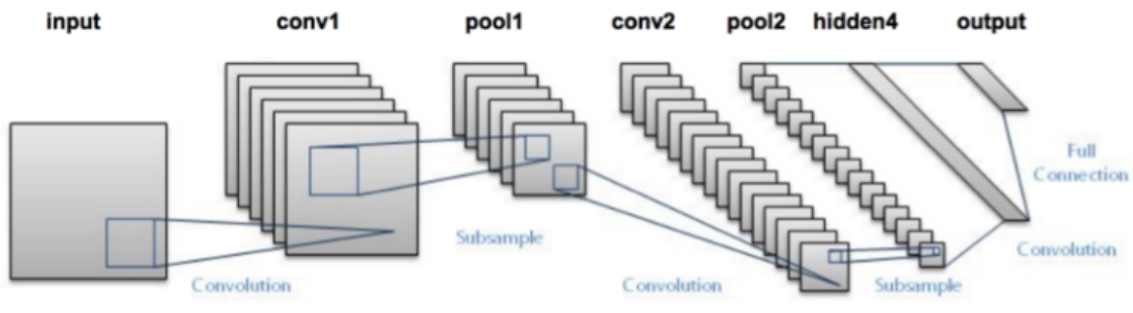
```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_examples = len(X_train)

    print("Training...")
    print()
    for i in range(EPOCHS):
        """
        shuffle your data after each epoch because you will always have the risk to create batches
        that are not representative of the overall dataset, and therefore, your estimate of the gradient will be of
        Shuffling your data after each epoch ensures that you will not be "stuck" with too many bad batches.
        """
        X_train, y_train = shuffle(X_train, y_train)
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train[offset:end], y_train[offset:end]
            accuracy, loss = sess.run([training_operation, loss_operation], feed_dict={x: batch_x, y: batch_y})

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

Approach to find the solution

My final model results were:

```
Training Accuracy = 0.971
Validation Accuracy = 0.941
Test Accuracy = 0.836
```

I choose a well-known architecture that is built by Yan Lecun in 1998. This original model includes two convolutional layers and a fully-connected layer as shown in the below.



LeNet model architecture

It has great performance on image recognition , in particular for handwriting numbers (e.g., MNIST digit recognition).

Since our problem (traffic sign recognition) is very similar to hand-writing recognition, both try to detect patterns in the images and match with the labeled training data. I believe the similar model shall work very well in our project.

Initial result is bad with the LeNet architecture:

EPOCH 1 …Validation Accuracy = 0.054
EPOCH 2 …Validation Accuracy = 0.048
EPOCH 3 …Validation Accuracy = 0.048
EPOCH 4 …Validation Accuracy = 0.048
EPOCH 5 …Validation Accuracy = 0.054

I did several **tunings of hyper-parameters** to improve the accuracy.
Here I show some of tuning as an example:

- **Add dropout in fully connected layer → still not good**

```
EPOCH 1 ...Validation Accuracy = 0.048
EPOCH 2 ...Validation Accuracy = 0.048
EPOCH 3 ...Validation Accuracy = 0.054
EPOCH 4 ...Validation Accuracy = 0.054
EPOCH 5 ...Validation Accuracy = 0.054
```

- **Reduce learning rate from 0.1 to 0.001 → big improvement!**

```
EPOCH 1 ...Validation Accuracy = 0.433 loss = 1.822
EPOCH 2 ...Validation Accuracy = 0.676 loss = 0.929
EPOCH 3 ...Validation Accuracy = 0.765 loss = 0.690
EPOCH 4 ...Validation Accuracy = 0.792 loss = 0.405
EPOCH 5 ...Validation Accuracy = 0.835 loss = 0.536
```

- **Further reduce learning rate from 0.001 to 0.0001 → get worse!**

```
EPOCH 1 ...Validation Accuracy = 0.065 loss = 3.528
EPOCH 2 ...Validation Accuracy = 0.097 loss = 3.338
EPOCH 3 ...Validation Accuracy = 0.170 loss = 3.002
EPOCH 4 ...Validation Accuracy = 0.263 loss = 2.683
EPOCH 5 ...Validation Accuracy = 0.329 loss = 2.320
```

- **Increase epochs from 5 to 10 → much better!**

```
EPOCH 1 ...Validation Accuracy = 0.362 loss = 2.073
EPOCH 2 ...Validation Accuracy = 0.638 loss = 1.035
EPOCH 3 ...Validation Accuracy = 0.731 loss = 0.705
EPOCH 4 ...Validation Accuracy = 0.794 loss = 0.564
EPOCH 5 ...Validation Accuracy = 0.818 loss = 0.513
EPOCH 6 ...Validation Accuracy = 0.823 loss = 0.267
EPOCH 7 ...Validation Accuracy = 0.839 loss = 0.247
EPOCH 8 ...Validation Accuracy = 0.857 loss = 0.312
EPOCH 9 ...Validation Accuracy = 0.860 loss = 0.326
EPOCH 10 ...Validation Accuracy = 0.868 loss = 0.208
```

- **Research on shuffling in each epoch → slight impact on validation**
- **Change “sigma” value for weight initial values → 0.1 is the best!**
- **Use different activation function (e.g., softsign) → no big impact**
- **Use “rescale_intensity” to enhance contrast → big improvement!**

```
EPOCH 1 ...Validation Accuracy = 0.573 loss = 1.256
EPOCH 2 ...Validation Accuracy = 0.735 loss = 0.732
EPOCH 3 ...Validation Accuracy = 0.798 loss = 0.480
EPOCH 4 ...Validation Accuracy = 0.832 loss = 0.400
EPOCH 5 ...Validation Accuracy = 0.850 loss = 0.360
```

- **… (many other tuning work)**
- **Finally, we reach a good accuracy! → 94% validation accuracy.**

```
Training...
EPOCH 1 ...Validation Accuracy = 0.557 loss = 1.365
EPOCH 2 ...Validation Accuracy = 0.733 loss = 0.780
EPOCH 3 ...Validation Accuracy = 0.780 loss = 0.394
EPOCH 4 ...Validation Accuracy = 0.820 loss = 0.333
EPOCH 5 ...Validation Accuracy = 0.841 loss = 0.279
EPOCH 6 ...Validation Accuracy = 0.863 loss = 0.223
EPOCH 7 ...Validation Accuracy = 0.863 loss = 0.209
EPOCH 8 ...Validation Accuracy = 0.879 loss = 0.154
EPOCH 9 ...Validation Accuracy = 0.885 loss = 0.200
EPOCH 10 ...Validation Accuracy = 0.897 loss = 0.174
EPOCH 11 ...Validation Accuracy = 0.895 loss = 0.134
EPOCH 12 ...Validation Accuracy = 0.904 loss = 0.116
EPOCH 13 ...Validation Accuracy = 0.898 loss = 0.081
EPOCH 14 ...Validation Accuracy = 0.900 loss = 0.125
EPOCH 15 ...Validation Accuracy = 0.901 loss = 0.083
EPOCH 16 ...Validation Accuracy = 0.910 loss = 0.077
EPOCH 17 ...Validation Accuracy = 0.903 loss = 0.038
EPOCH 18 ...Validation Accuracy = 0.908 loss = 0.077
EPOCH 19 ...Validation Accuracy = 0.917 loss = 0.057
EPOCH 20 ...Validation Accuracy = 0.915 loss = 0.120
EPOCH 21 ...Validation Accuracy = 0.918 loss = 0.081
EPOCH 22 ...Validation Accuracy = 0.920 loss = 0.036
EPOCH 23 ...Validation Accuracy = 0.918 loss = 0.089
EPOCH 24 ...Validation Accuracy = 0.917 loss = 0.047
EPOCH 25 ...Validation Accuracy = 0.920 loss = 0.075
EPOCH 26 ...Validation Accuracy = 0.917 loss = 0.039
EPOCH 27 ...Validation Accuracy = 0.919 loss = 0.073
EPOCH 28 ...Validation Accuracy = 0.918 loss = 0.051
EPOCH 29 ...Validation Accuracy = 0.927 loss = 0.055
EPOCH 30 ...Validation Accuracy = 0.932 loss = 0.041
EPOCH 31 ...Validation Accuracy = 0.924 loss = 0.053
EPOCH 32 ...Validation Accuracy = 0.932 loss = 0.023
EPOCH 33 ...Validation Accuracy = 0.926 loss = 0.065
EPOCH 34 ...Validation Accuracy = 0.923 loss = 0.022
EPOCH 35 ...Validation Accuracy = 0.923 loss = 0.024
EPOCH 36 ...Validation Accuracy = 0.918 loss = 0.027
EPOCH 37 ...Validation Accuracy = 0.928 loss = 0.069
EPOCH 38 ...Validation Accuracy = 0.924 loss = 0.027
EPOCH 39 ...Validation Accuracy = 0.929 loss = 0.011
EPOCH 40 ...Validation Accuracy = 0.927 loss = 0.049
EPOCH 41 ...Validation Accuracy = 0.932 loss = 0.040
EPOCH 42 ...Validation Accuracy = 0.931 loss = 0.015
EPOCH 43 ...Validation Accuracy = 0.938 loss = 0.027
EPOCH 44 ...Validation Accuracy = 0.932 loss = 0.063
EPOCH 45 ...Validation Accuracy = 0.929 loss = 0.009
EPOCH 46 ...Validation Accuracy = 0.931 loss = 0.022
EPOCH 47 ...Validation Accuracy = 0.931 loss = 0.067
EPOCH 48 ...Validation Accuracy = 0.930 loss = 0.035
EPOCH 49 ...Validation Accuracy = 0.930 loss = 0.037
EPOCH 50 ...Validation Accuracy = 0.941 loss = 0.029
```

4. Test Model on New Images

New images from web

- I found five Germany traffic sign images from google search. I actually crop them away from larger images.

1. the first image is “bumpy road” sign. It has shutters-like changes on the surface of sign, which could cause our model make wrong detection.

2. the second image is “priority road” sign. It has sky and cloud as background. It might be difficult to classify because partial of the sign is white and very close to the background. So the sign can easily mixed the sign with the cloud in the back together.

3. the third image is “no vechicles” sign. On the right bottom corner, there is a person with red jacket. It is very close to the outer red circle of the sign. It may cause mistake for our classifier.

4. the forth image is “turn right ahead” sign. This image is not straight-facing to us. It was tilted towards right side. It may cause issue for our classifier when the training data are perfect straight-facing ones.

5. the fifth image is “stop” sign. It has darker background, and the edges of sign is mixed with background power tower. It could cause difficulity to classify the stop sign correctly.

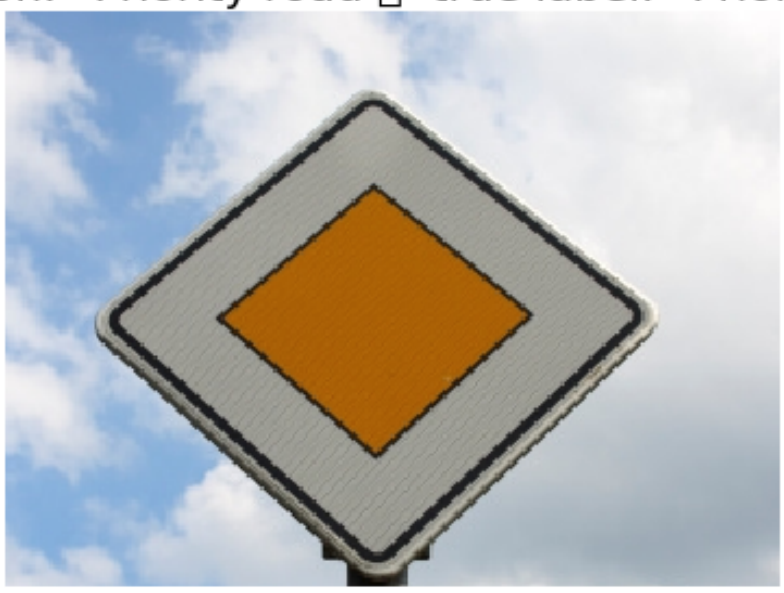
. . .

Test result on new images

In my testing, all the new images can be correctly classified with our model. Following images shows the prediction result from our model v.s. the true label from given.

prediction:=Bumpy road ✅ true label:=Bumpy road

prediction:=Priority road ✅ true label:=Priority road



prediction:=No vehicles ✅ true label:=No vehicles

prediction:=Turn right ahead true label:=Turn right ahead

prediction:=Stop true label:=Stop

Moreover, we shows the top five prediction for each image as following.

- Each group is predictions for each image (**the label is shown as bold**);
- Each line shows the estimated **probability** of each category from model.
- The category with **highest probability** is the prediction result.

For example, for the image “Bumpy road”, our model predicted that this image belongs the category of “Bumpy road” with 99.8% probability. Therefore, the model believes this image is “Bumpy road” sign.

```
--------------------------------------------------------------
-

         True Label is: =    22:Bumpy road

   22: Bumpy road                     99.812%
   29: Bicycles crossing              0.101%
   24: Road narrows on the right      0.087%
   31: Wild animals crossing          0.000%
   26: Traffic signals                0.000%
--------------------------------------------------------------
-
--------------------------------------------------------------
-

         True Label is: =    12:Priority road

   12: Priority road                  100.000%
   40: Roundabout mandatory           0.000%
   26: Traffic signals                0.000%
   38: Keep right                     0.000%
   42: End of no passing by vehicles over 3.5 metric tons
0.000%
--------------------------------------------------------------
-
--------------------------------------------------------------
-

         True Label is: =    15:No vehicles

   15: No vehicles                    99.993%
    2: Speed limit (50km/h)           0.005%
    3: Speed limit (60km/h)           0.002%
    5: Speed limit (80km/h)           0.000%
    1: Speed limit (30km/h)           0.000%
--------------------------------------------------------------
-
--------------------------------------------------------------
-

         True Label is: =    33:Turn right ahead

   33: Turn right ahead               100.000%
    3: Speed limit (60km/h)           0.000%
   35: Ahead only                     0.000%
   39: Keep left                      0.000%
    1: Speed limit (30km/h)           0.000%
--------------------------------------------------------------
-
--------------------------------------------------------------
-

         True Label is: =    14:Stop

   14: Stop                           100.000%
   17: No entry                       0.000%
   38: Keep right                     0.000%
   34: Turn left ahead                0.000%
   32: End of all speed and passing limits 0.000%
```

```
_________________________________________________________
_
```

5. Summary and Conclusion

In this project, we built a traffic sign classifier using convolution neural network and tensorflow. It is a great and inspiring project to practice those techniques, and indeed, it helps me better understand the concept of neural network, layers, relu, max_pooling and etc.

During the training process, I found several parameters are vey critical to the accuracy, such as learning rate, the epoches number, shuffling in each epoch, sigma of weight initialization, and layer depth selection.

On the contrary, some other parameters or operations does not help, such as dropout operation, and “VALID”/“SAME” padding methods. Maybe I can try them in future project and see their impact.

Overall, this is great learning experience. I had learned so much knowledge in the past few weeks. Thanks for all the class arrangement and preparation material!

Happy Thanksgiving! :-) ✌

Nov 26, 2017 3:03 PM