Deep Learning for Traffic Sign Recognition: A Convolutional Neural Network Approach

MOHAN

IIT KHARAGPUR

sivakumarmohan18@gmail.com

Abstract—Traffic Sign classifier is very crucial in the current era of self-driving and autonomous cars. Due to high speed movement of vehicles and criticality of the situation, recognition and classification accuracy is very important. This problem has two aspect, first is feature extraction (extracting relevant feature from the traffic sign images) and second is image classification (classifying image to the corresponding classes based on the extracted features. Identification of the traffic and taking appropriate action is crucial for an automated vehicle. This system will assist drivers by recognizing traffic signs which they didn't recognize before passing and will save them from miss-happenings. It can also be applied to urban scene understanding. To solve this problem, I have trained a convolutional neural network which learn from the traffic sign images and use that learning to classify new traffic sign images. I have used German Traffic Sign Dataset for training and validation.

Keywords—CNN, LeNet, ReLu, Adam Optimizer

1. Introduction

Traffic signs recognition and classification are the integral parts of self-driving vehicles and advanced driver assistance systems. Due to lack of attention and presence of different obstacles many times drivers miss some of the traffic signals which can lead to accidents. With the recent research and advancement in deep learning algorithms along with availability of high performance hardware units, such as graphical processing units (GPU), recognition and classification of traffic signs has become much more efficient and fast as compared to traditional machine learning and computer vision approach.

There are variety of factors that make the building of system much difficult, some of these are adverse variation due to illumination of captured images, the speed variation of vehicles, orientation of the signals, image distortion etc. To capture the high-quality traffic sign images and other visual feature, a wide-angle camera is mounted in self driving

vehicle. The images taken from these cameras are distorted due to several external factors including vehicles speed, sunlight, rain etc. Sample images from GTSRB dataset are shown below.

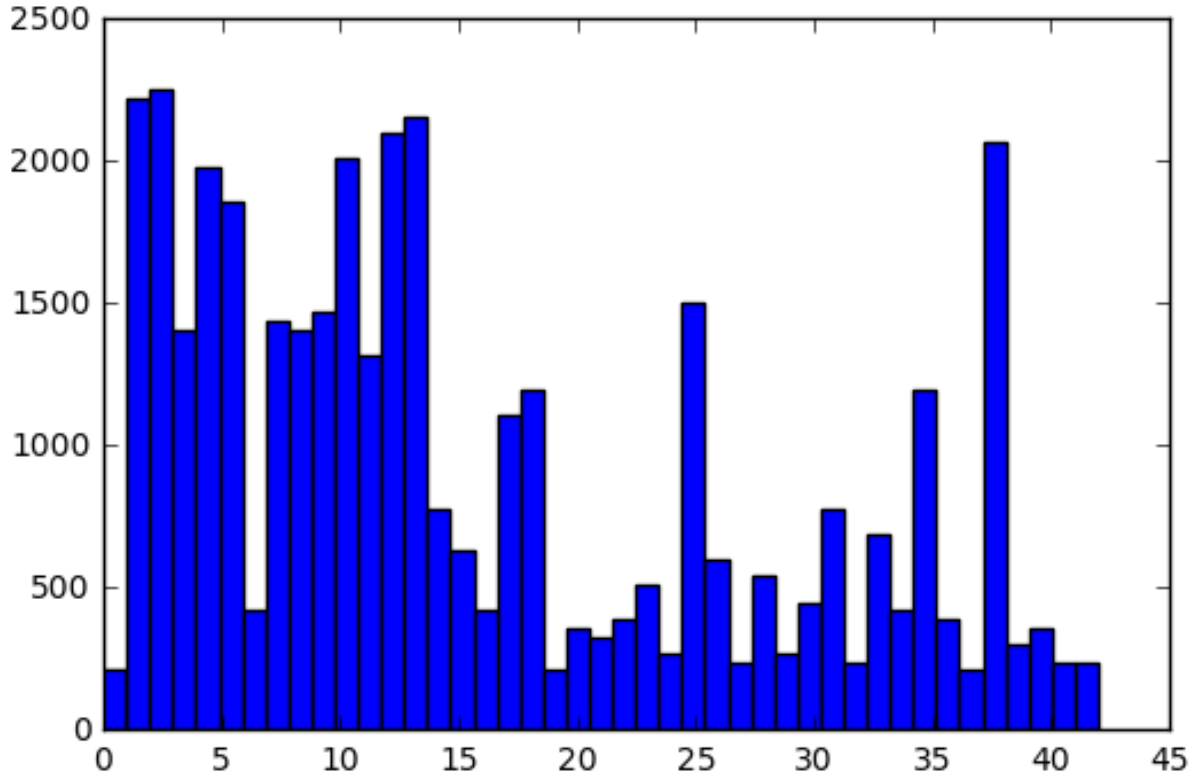


Fig 1. Sample traffic sign images in GTSRB data set

To solve this problem of recognition and classification of the GTSRB dataset I have followed LeNet architecture and trained a convolutional neural network. I have made a classifier which can be used to classify traffic sign images by extracting relevant features from the training images and use those features to learn the pattern and update the weights and parameters. Once classifier is trained I have used this to classify test images and compared the accuracy. I have used TensorFlow for building the classifier as it is most powerful and easy to use deep learning framework available currently. Python is used as programming language.

1. Dataset

I have used German Traffic Sign Recognition Benchmark (GTSRB) which is one of the most reliable datasets for training, testing and validating traffic sign classification and detection algorithms. This dataset has more than 50,000 images with more that 40 classes. Each image is in PPM format and contains one traffic sign. Size of images varies between 15X15 to 250X250. In the image the traffic sign is not necessarily be at center and squared. The image contain border of 10% around the actual traffic sign to facilitate edge

based approach. For this classifier, I have used a training dataset of size 39,209. The test dataset contains 12,630 test images in random order.

2. Data Pre-processing and Augmentation

In order to help the gradient descent optimization for training the model, we need to preprocess the input data set (images). The pre-processing task takes place in multiple steps as follows:

1. Random shuffling: Shuffling helps in reducing overfitting and correlation between two consecutive images
2. Grey scale conversion: This will convert the RGB to gray scale image
3. Image centering: In order to center the data around origin subtract each pixel value from the mean pixel value
4. Normalization: Once the image centering is done then we divide each dimension from its standard deviation. With normalization, each feature will be having similar range to hold control over gradient.

When the total number of images belonging to each labels or class varies significantly, we need to transform the dataset in such a way so that number images in each class is in proportion and greater than the mean count. For this I have applied augmentation on training images. For this I have used following techniques:

- Translation – we translate image by certain value on x axis or y axis to generate new images.

- Rotation – we rotate the image by certain angle to generate new images

- Affine Transformation - An affine transformation is any transformation that preserves collinearity and this helps us generating new images from existing images.

After preprocessing, we will have 52396 images in the training dataset each with size of 32X32X1.

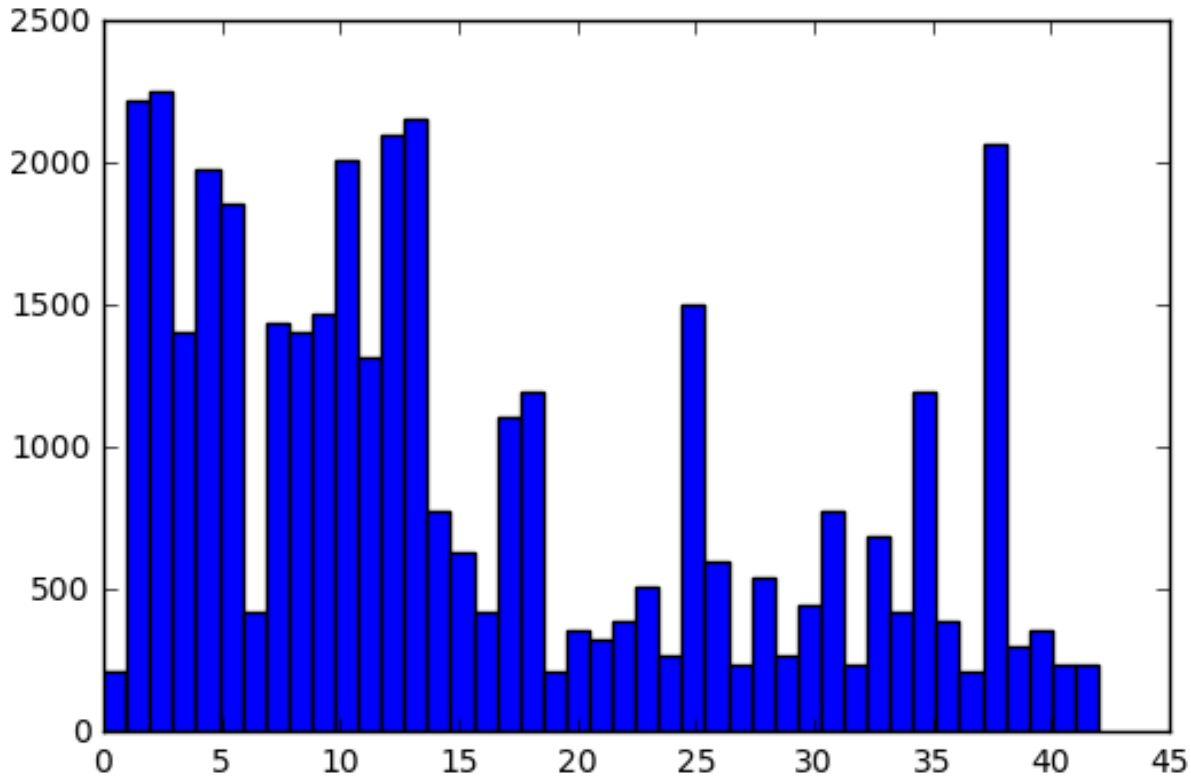


Fig 2. Original image and corresponding Transformed Images

3. Architecture

- **Convolution Neural network**

Convolution neural network is feed forward artificial neural network where neurons connectivity pattern is inspired by animal visual cortex, which are effective in area of image recognition and classification. These neurons have weights and biases which can be learned from the image data. They are efficient in identifying faces, traffic signs, objects apart from powering in vision in robot and self- driving car.

Convolution neural network (CNN) contains multiple layers called convolutional layers (for subsampling) followed by one of more fully connected layer similar to multilayer neural network. CNNs are basically just several layers of convolutions with nonlinear

activation function. I have used ReLu activation function in this implementation. I am also using dropout layers to prevent overfitting.

Architecture of CNN consist of convolutional layer and subsampling layer optionally followed by fully connected layer The HxHxR image is fed to the convolutional layer where H denotes the dimensions (Height and width) of the image and R represents the number of channels(colors) for example in gray images R=1. The convolutional neural network layer contains filter of size n x n x q where n is smaller dimension of input image and q can be less than or equal to the R(channel) or it can vary too. Filters size gives rise to locally connected structure each of which convolved with image to produce K map of size H-n+1 then this is feed forwarded to subsample layer where each map is subsampled by mean or max pooled. A bias and sigmoidal non-linearity layer is applied to each feature map either before or after subsampling

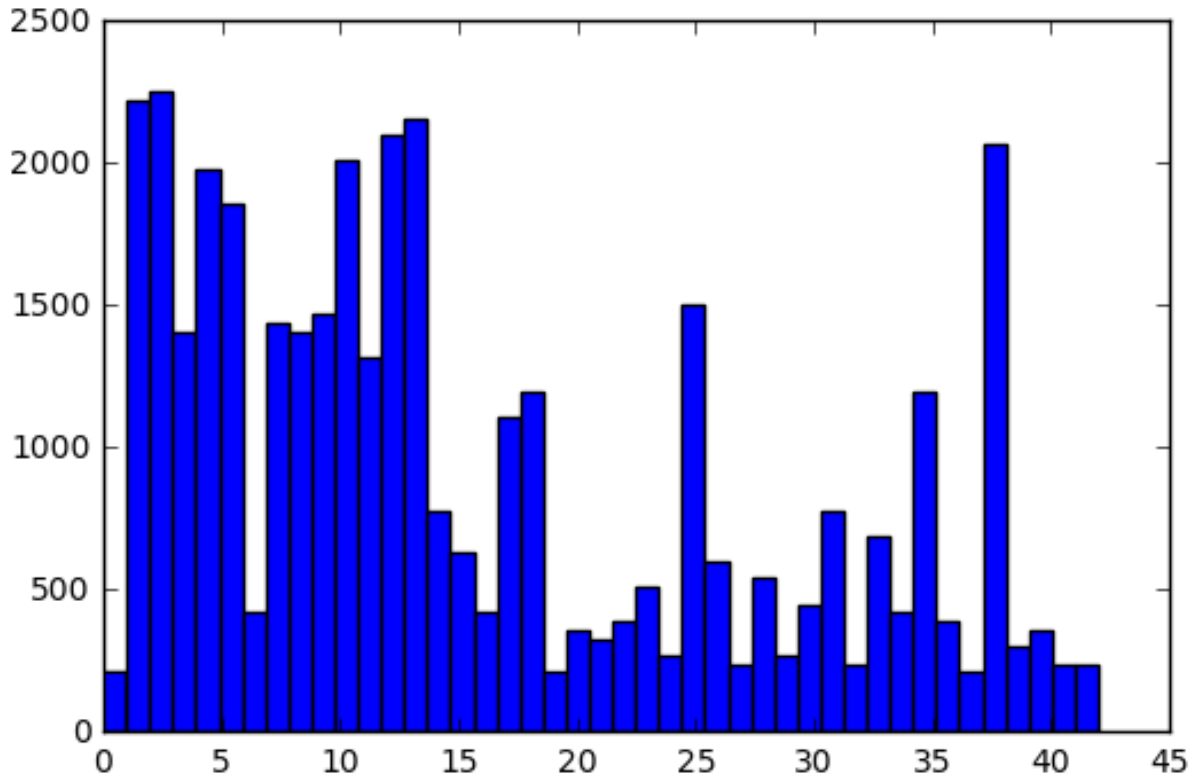


Fig 3.

CNN make assumption that input is images. It comprises of various layers where each layer will apply convolution using filter and transform an input 3-D data into output 3-D data with function which may be parametrized or non-parametrized.

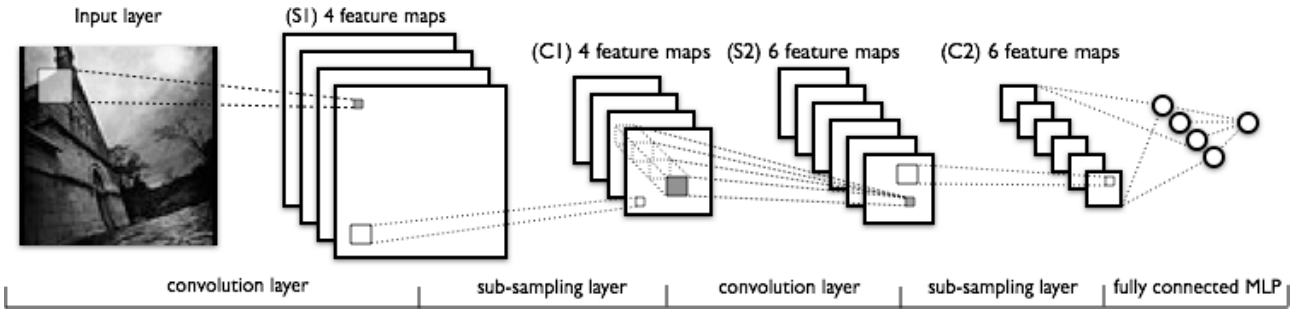
Convolutional Layer, Pooling layer and Fully Connected Layer are the main layers of ConvNet where convolutional layer is the core building block layer. Pooling layer is put in between convolutional layers that progressively reduces the spatial size in order to reduce computation and parameters in the network. Fully connected layer consists of

neuron having full connections to the activation in the previous layer. Main architecture in the field of CNN are LeNet, Alex Net, ZFNet, GoogleNet etc.

There are two main aspects of this CNN implementation: Local variance and compositionality. In practice subsampling returns the invariance to translation, rotation and scaling. The filters contain local patch of lower level feature into higher level representation which makes is a powerful tool in computer vision. Processing works in such a way like you build edges from pixels, shapes from edges, and more complex objects from shapes which is quite intuitive.

LeNet-5 is one of the neural inspired model which is first architecture for convolutional neural network. Its quite easy to learn and understand. Sparse convolutional layers and max pooling (which is used for subsampling) are at the core of the LeNet family. The figure below shows a graphical representation of a LeNet model showing generic internal details which varies as per the requirement.

In addition, we have used dropout layers which is regularization technique and prevent overfitting. In this layer, we drop some of the units while moving from one layer to next layer.



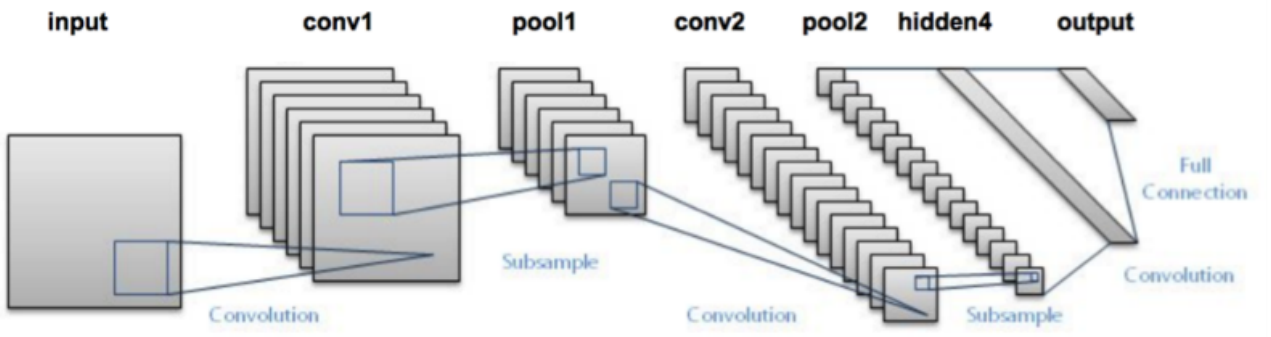


Fig 4.

- **Max Pooling**

This concept is basically the sample-based discretization process with an objective to down sample the input representation by reducing and allowing assumption to be made for the feature within the sub region window. Max pooling layer is used in CNN in order to decrease the size of image to reduce the number of features and computational complexity of the CNN by keeping the depth of the layer same.

The reduction goes from feature map (m, m) to (m/k, m/k) by applying a (k, k) non-overlapping filter. Hence k should be chosen in proportion of the dimension of the input feature map. In contrast, the dimensions on application of a convolution layer would move to (m-k+1, n-k+1). Unlike the convolution layer, the pooling filter does not operate on overlapping segments of the input feature map. However, we can explicitly specify the stride to make the operation overlapping (though it is not recommended in most cases). The pooling can be implemented in quite many other ways to.

On the diagram below we show the most common type of pooling the max-pooling layer, which slides a window, like a normal convolution, and get the biggest value on the window as the output.

It's also valid to point out that there are no learnable parameters on the pooling layer. So, it's backpropagation is simpler.

The most important parameters to play:

- Input: H1 x W1 x depth_in x N
- Stride: Scalar that control the number of pixels that the window slide.
- K: Kernel size
- Output H2 x W2 x depth_out x N:

W2 = (W1−K)/S+1H2
= (H1−K)/S+1 depth_out
= depth_in W2
= (W1−K)/S+1H2
= (H1−K)/S+1 depth_out
= depth_in

It's also valid to point out that there are no learnable parameters on the pooling layer. So, it's backpropagation is simpler.

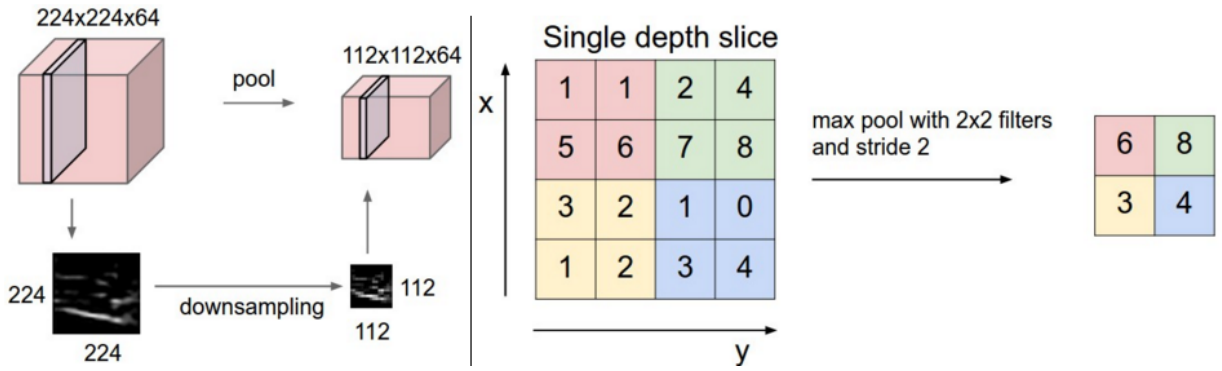


Fig 5. Max-Pooling

- **Adam optimizer**

Adam optimizer employ Adam algorithm a first order gradient based optimization algorithm. The TensorFlow library has **tf.train.AdamOptimizer** function which uses Kingma and Ba's Adam algorithm to regulate the learning rate. It has several advantages over function Gradient Descent Optimizer**.** Main advantage is that it uses the moving averages of parameters (momentum). The simple momentum way is to make the updates directly proportional to the averaged gradient estimator. The aim is to remove some of the noise and oscillations of gradient descent in order to minimize the loss function output. This advantage enable Adam to take large step size and the algorithm will converge it without finetuning. The main cons of Adam optimizer is that it needs complex computation for each parameter in each training step in order to maintain the moving averages and variance and calculate the gradient descent. In addition, more steps are required to be retained for each parameter. A simple tf.train.GradientDescentOptimizer could equally be used here, but would require more hyperparameter tuning before it would converge as quickly.

- **Fine-Tuning**

We initialize the model with random wrights and the idea of fine-tune is using the train data and tuning the parameters and weights with smaller learning rate. Chossing correct value for learning rate is also important.

- **Activation funcation - ReLu**

Activation funcations is crucial factor in deep neural netwrok which brings non linearity into network. A Rectified linear unit (ReLU) is type of activation function that is defined as:

$$\mathbf{f(x) = max(0, x)}$$

which is threshold at zero.

The function returns 0 if x is negative,else x. TensorFlow has built in ReLu function which can be used as tf.nn.relu():

Below is the snippet of it.

hidden_layer = tf.add(tf.matmul(features, hidden_weights), hidden_biases)
hidden_layer = tf.nn.relu(hidden_layer)
output = tf.add(tf.matmul(hidden_layer, output_weights), output_biases)

As ReLu function is applied to the hidden layer which act like on-off switching and turn off the negetive weights.
The above code applies the tf.nn.relu() function to the hidden_layer, effectively turning off any negative weights and acting like an on/off switch. After activation function ,when an additional layer is added turns the model into nonlinear function, and this non linearity makes the network to solve more complex non-linear problems.

- **Dropout**

As overfitting is the serious problem with deep neural network having large number of parameters. Slowness of large networks make it more difficult and over fit by combining the predictions of large number of neural network at test time. To overcome this problem, dropout, a regularization technique, plays the key role in which we randomly drop units along with its connections from the network during the training, which in turn prevents the units to co-adapt too much. At the test time, it is easy to approximate the effect of averaging the prediction of all these dropout units by using single network that has smaller weights.

- **Backpropagation**

Backpropagation, backward propagation of error, is very important method for training neural networks in order to minimize the loss function. Backpropagation is used along with an optimization method such as Adam optimizer or Gradient Descent etc. This process reiterates in two phase, forward propagation then validation followed by weight update. When an input is fed into the network, it is propagated forward through the layers of the network, and get process at each layer, until it reaches the output layer. At output layer, the output is then compared to the expected output, using a loss function. Our goal is to minimize this loss function output, with this goal an error value is computed for each of the neurons in the output layer. This error value is then back propagated to all the layers and each layer get associated with error value which roughly represents its contribution towards the original output. Once this is done weight at each layer is updated as per the error value.

4. Implementation

To solve this recognition and classification problem I am extending popular LeNet architecture and making it more suitable for traffic sign image recognition and classification.

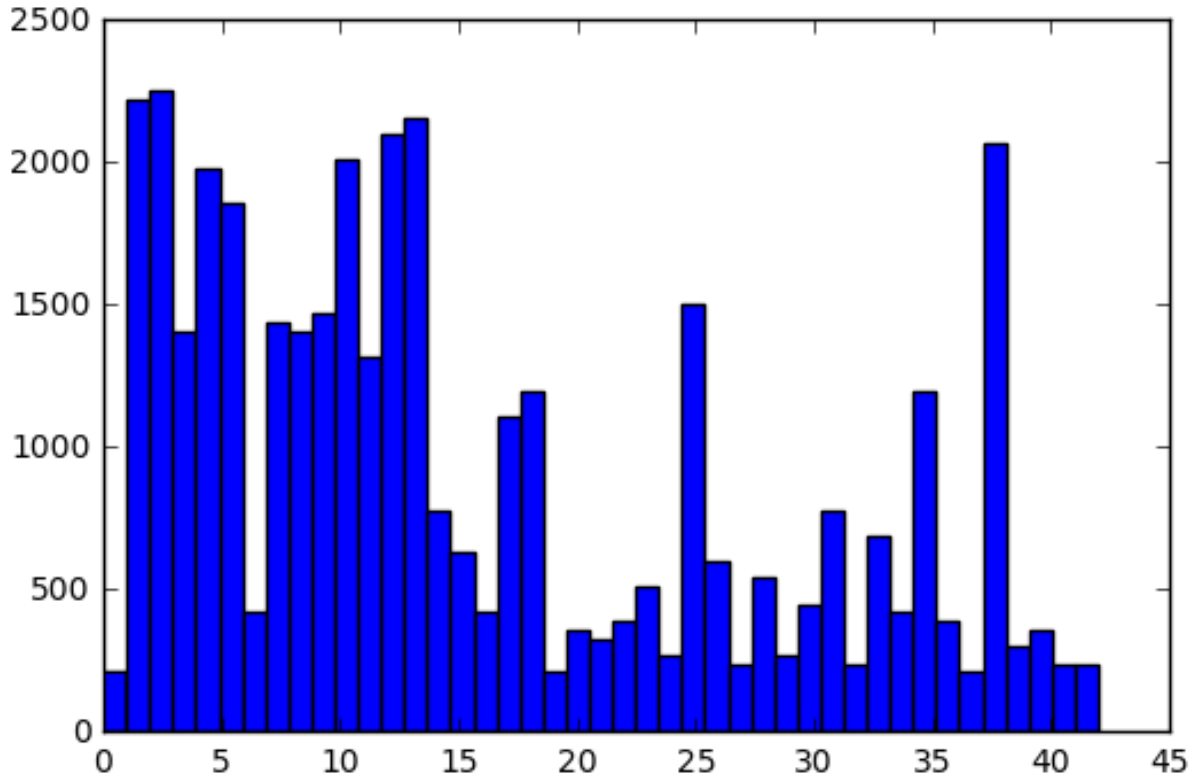


Fig 10. LeNet architecture

Fig 10 shows LeNet architecture which takes 32X32 input images. These images then go to convolutional layer c1 followed by pooling (sub sampling) layer s2. Then followed by sequence of convolutional layer c3 and pooling layer (sub sampling). Then there are three fully connected layer followed by one output layer.

I have implemented LeNet architecture using TensorFlow and Python. I have changed the number of hidden layers used and added drop out layers and tuned the hyper parameters.

- **Data set summary and exploration**

I have used the German traffic sign images dataset (in the pickled form) as training and test data.

Train size - 34799
Test size – 12630

In order to get more number of training images I have transformed training set. After transformation, following is the extended data set size:

Number of training examples = 46714
Number of testing examples = 12630

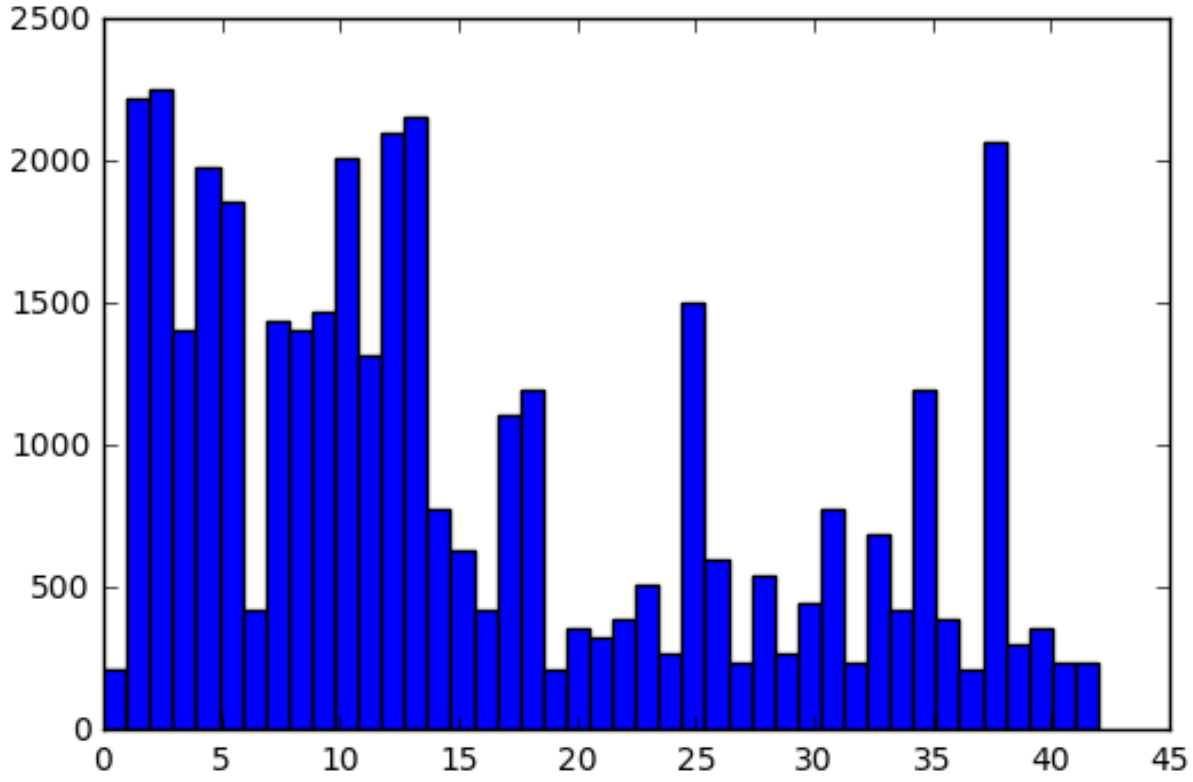


Fig 11. Histogram of image count of each class

After this I have done the gray scaling of the images

Earlier image shape= (32,32,3)
After gray scaling = (32,32,1)

Concluding the data preprocessing (where images are already into 32X32 shape), I have converted it from RGB to grayscale and then did the shuffling of the training and test data set. Shuffling reduces the overfitting and correlation between the two images. In order to scale down the disparity within data images are normalized. After this, additional training data is generated on trained data set. The reason of adding the more images in the training data set is referring the histogram fig 11 where images in each class vary significantly so I

have added images to the class in a way that each class have images equal to the mean Fig 12.

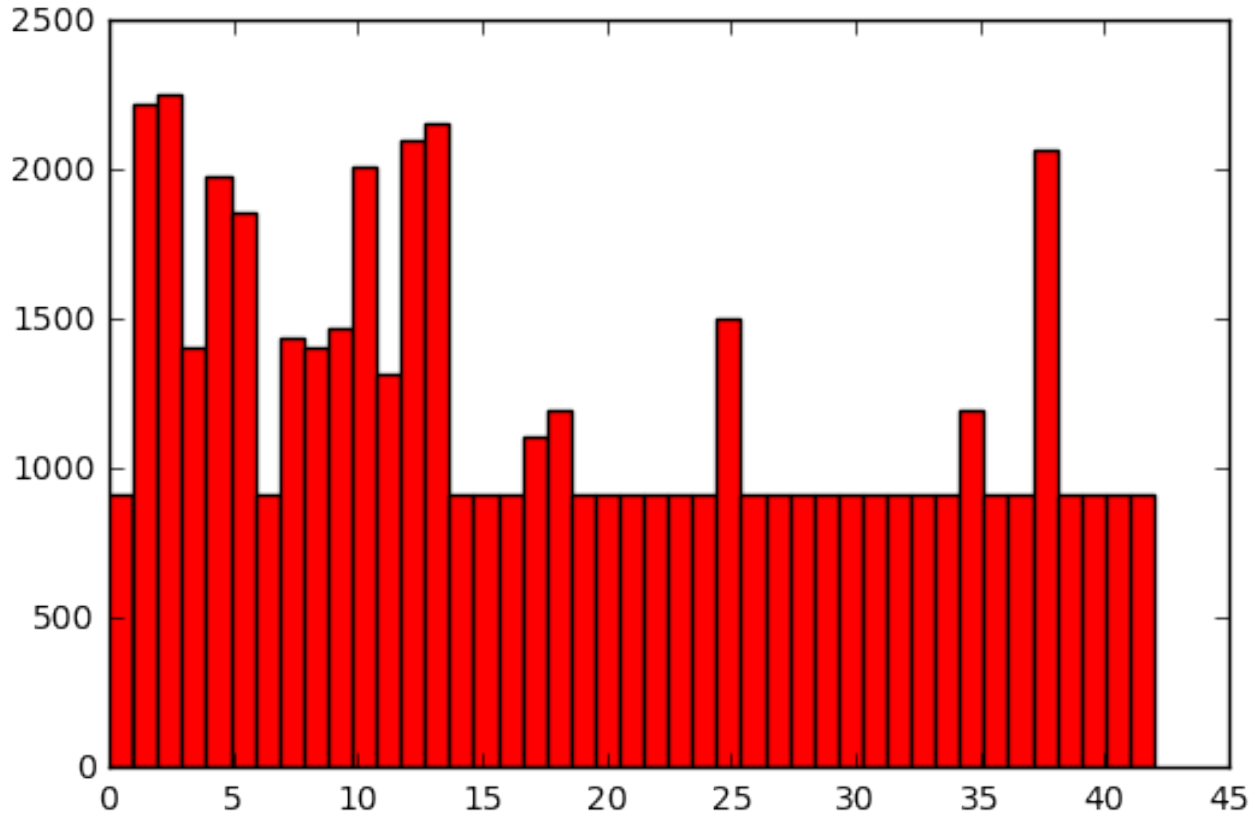


Fig 12. Histogram of training after sub sampling

- **Building the model**

The model consists of 2 convolutional layer and 3 connected layer. Each of the convolutional layer has ReLu activation function which is followed by max pooling. Convolutional layers are followed by 3 fully connected layer and one output layer.

Each layer is implemented by TensorFlow and have below input-output:

1.**Convolutional layer 1-** It takes 32X32X3 image input and output 28X28X6 output image.

```
Weight = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), mean = mu, stddev = sigma))
Bias = tf.Variable(tf.zeros(6))
output   = tf.nn.conv2d(x, weight, strides=[1, 1, 1, 1], padding='VALID') + bias
```

2.**Max pooling (sub sampling)**: This layer takes the input 28X28X6 and output 14X14X6 image. Maxpooling is done using tensor flow function tf.nn.max_pool.

3.**Convolutional layer 2**- It takes 14X14X6 image input and output 10X10X16 output image.

weight = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean = mu, stddev = sigma))
bias = tf.Variable(tf.zeros(16))
conv2 = tf.nn.conv2d(conv1, weight, strides=[1, 1, 1, 1], padding='VALID') + bias

Before going to max pooling layer, I have used one dropout layer to avoid overfitting with keep_prob as the drop rate.

conv1 = tf.nn.dropout(conv1, keep_prob)

4.**Max pooling (sub sampling)**: This layer takes the input 10X10X16 and output 5X5X16 image. Maxpooling is done using tensor flow function tf.nn.max_pool.

Before going to fully connected layer I have flatten the out of last max pooling using

fc0 = flatten(conv2)

where conv2 is the output of max-pooling which is 5X5X16 data which returns output = 400.

5.**Fully Connected layer 1**(**FC1**)-It takes input =400 and output=120

weight = tf.Variable(tf.truncated_normal(shape=(400, 120), mean = mu, stddev = sigma))
bias = tf.Variable(tf.zeros(120))
output = tf.matmul(fc0, weight) + bias

7.**Fully Connected layer 2**(**FC2**)-It takes input =120 and output=84

weight = tf.Variable(tf.truncated_normal(shape=(120, 84), mean = mu, stddev = sigma))
bias = tf.Variable(tf.zeros(84))
output = tf.matmul(fc1, weight) + bias

8.**Fully Connected layer 3**(**FC3)**-It takes input =84 and output=64

weight= tf.Variable(tf.truncated_normal(shape=(84, 64), mean = mu, stddev = sigma))
bias = tf.Variable(tf.zeros(64))
output = tf.matmul(fc2, weight) + bias

9.**Output layer**: It takes input =64 and output=43 classes.

weight= tf.Variable(tf.truncated_normal(shape=(64, 43), mean = mu, stddev = sigma))
bias = tf.Variable(tf.zeros(43))
logits = tf.matmul(fc3, weight) + bias

Each of these layers are activated by ReLu Activation function as
Output = tf.nn.relu(output of the previous layer)

This activation function adds the non-linearity in the network.
I have used Adam optimizer for optimization and loss minimization. One-Hot Encoding was used to convert label numbers to vectors.

- **Training the model**

I have randomly split the train dataset into 80:20 ratio and created two data set one for training and one for validation. Once model is built I trained it using train dataset, I have used below hyper parameters for which I have got the best accuracy. Besides
that, I have tried EPOCH=10, 20 and batch size=64, 256 too.

- EPOCHS = 50, number of epochs
- BATCH_SIZE=128
- mu = 0
- sigma = 0.05
- keep_prob = 0.6
- training rate = 0.001

I have batched train dataset to train the model with above mentioned batch size and compare the predicted labels with original class labels for the validation data set. Based on the cross-entropy Adam optimizer optimizes the weights. For accuracy evaluation, each batch was combined to provide the accuracy for the model on train dataset.

- **Testing the model with unseen test data**

Above model never see test data set during training, it only uses validation data set for optimization. Once model is built and trained with train and validation dataset, in the end I have used unseen test dataset to check the accuracy of the trained model.

- **Results**

As mentioned before, my train data set contains 12630 images. I classified these images using the trained model and compared them against the actual class it belongs to. Total accuracy that I got using my model is equal to 92.4%.

5. Conclusion

In this project, we have learnt about convolutional neural network and its various component. In addition, we have used LeNet architecture for Traffic sign recognition and classification which is a crucial part of self-driving car industry. Our model gave accuracy of 92.4% of unseen test data. In future work, we can try more combination of different layers and try other architectures of CNN to improve the accuracy of classification.

6. References

[1]. Traffic Sign Recognition with Multi-Scale Convolutional Networks
http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf

[2]. ImageNet Classification with Deep Convolutional Neural Networks
http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf

[3].Dataset :
http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset

[4].Color to gray scale
http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0029740

[5]. Gradient based learning applied to document recognition
http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf

[6]. Image Classification: Analysis
http://homepages.inf.ed.ac.uk/rbf/HIPR2/classify