Capstone Project

Hai Xiao

Machine Learning Engineer Nanodegree

January 28th, 2017

Definition

Project Overview

In the autonomous car industry, computer vision and deep learning have many important use cases to both provide accurate scene or context understanding to driving safety and intelligent augmentation or automation to the vehicle control, examples range from lane lines recognition & tracking, pedestrian classification, vehicle detection and tracking or traffic sign classification, etc.

With applications in demand and rapid evolving of GPU, image processing method, architecture, and algorithm deep learning infrastructure software have made great advancement in the recent years, popular ones including Caffe, Torch, Keras, Theano and TensorFlow.

In this capstone, I will use TensorFlow to develop traffic signs image classifiers in addition to traditional machine learning libraries such as sklearn, using Python 2.7.

Problem Statement

The goal is to use TensorFlow implementing deep neural networks and specifically convolutional neural networks to classify traffic signs, specifically the German Traffic Sign Dataset.

The high level tasks involved are the following:

1. Acquire dataset
2. Dataset exploration
3. Prepare dataset, data preprocessing
4. Design, implement and train the model with model evaluation and validation
5. Test model on new images with analysis and justification
6. Reflection and improvement discussion

Reasonably final model trained should reach ~95.0% accuracy (on moderate CPU with none extensive network architecture) on the test dataset, provided following GTSRB 2011 benchmark:

Rank	Team	Representative	Method	Correct recognition rate
1	IDSIA	Dan Ciresan	Committee of CNNs	99.46 %
2	INI		Human Performance	98.84 %
3	sermanet	Pierre Sermanet	Multi-Scale CNNs	98.31 %
4	CAOR	Fatin Zaklouta	Random Forests	96.14 %

Metrics

Accuracy is a common metric for classification problems, even it is not the only one, others are ***precision*** and ***recall***. Since label space is not very small (43 classes in total) in this problem, it makes more sense to use ***Accuracy*** as metrics to measure our model. It is defined as:

$$Accuracy = \frac{ture\ positives + ture\ negatives}{total\ dataset\ size}$$

In my *TensorFlow Graph*, Accuracy (not on global dataset, but a batch instead) is computed as:

(SGD is proved to be an effective online learner, where partial or batch optimization works just fine)

```
good_pred = tf.equal(y_pred_class, y_true_class)
accuracy = tf.reduce_mean(tf.cast(good_pred, tf.float32))
```

Behind the scene, accuracy itself might not be the best to guide the DNN training or numeric optimization per se, for example even a model is at 100% training accuracy, its learning parameter (weights) still have room to improve, particularly in case of DNN (Deep Neural Network). So I use ***Loss* vs. *Accuracy*** to optimize the learner in *TensorFlow Graph*, as:

```
loss = tf.reduce_mean(cross_entropy)
optimizer = \
tf.train.AdamOptimizer(learning_rate = learn_rate).minimize(loss)
```

Analysis

Data Exploration

The GTS (German Traffic Sign) data come with separate training and testing dataset (from Udacity), I only did re-pickle to protocol version 2 (to work with Python 2.7 vs. 3). All the images are uniformly formatted RGB as `(32, 32, 3).astype(np.uint8),` there is no abnormalities, neither the size (`32x32x3`)nor the color encoding.

- training dataset ('train.pkl'): total 39209 samples across all 43 traffic sign classes
- testing dataset ('test.pkl'): total 12630 samples across all 43 traffic sign classes

Each sample has an image of shape=(32, 32, 3) as X, a number of range [0:43] as classId Y. Here are 4 example images with their class names (*N.B. signnames.csv maps classId to className*) from this dataset:

Exploratory Visualization

The plot below shows the sample distributions across different traffic sign classes, in both training set (prior to validation split) and testing set. This visualization is helpful in that it tells us:

1. *Sample distributions are unbalanced across classes*
2. *Samples of all 43 traffic sign classes exist in both data set, and*
3. ***Sample distributions are similar between training data set and testing data set***

Common machine learning practices to deal with unbalanced data set in image applications are:

1. *Augment images at pipeline Input, generate more artificial images to minority classes*
2. *Sample more often or increase sample weight of minority classes during training*

These technique are particularly necessary assuming unknown test data set. Here since we already know that testing distribution is very similar (in shape) to training data set across labels, I expect a good result achievable even without image augmentation or sample weighting.

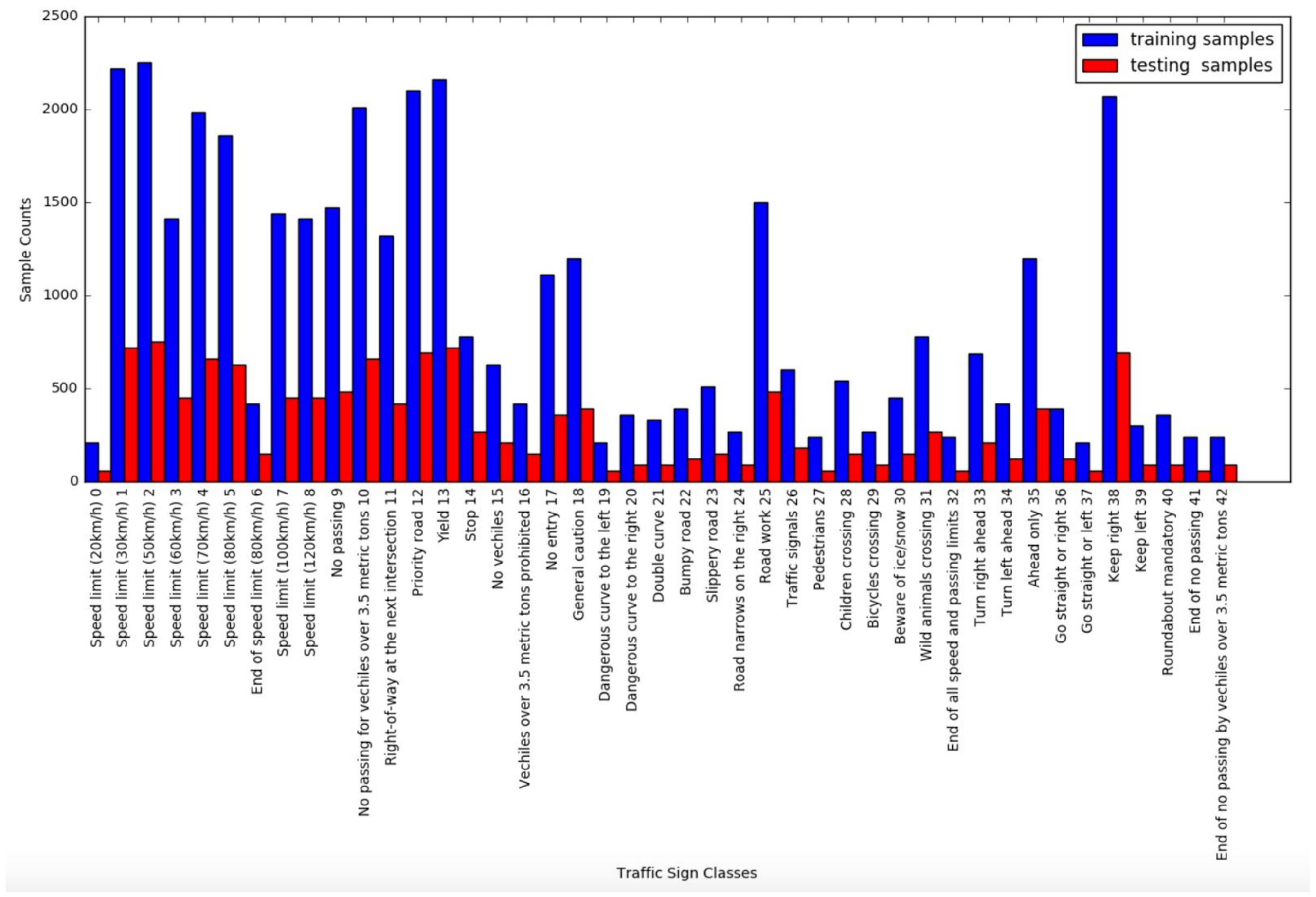

Figure 2. Sample Distributions across Classes (blue: training vs. red: testing)

Algorithms and Techniques

The classifier is one *Deep Neural Network* or ***Convolutional Neural Network (CNN)*** in specific. Inputs are normalized image data, outputs are ***Onehot encode Logits*** since it is a classification problem, ***loss*** is evaluated as ***cross entropy*** *between* ***softmax*** *of output* ***Logits*** *and true* ***label. Optimizer (Adam)*** is using *Stochastic Gradient Descent (SGD)* method to minimize the ***Loss***. ***Prediction*** is the classId with the ***maximum softmax probability*** of output ***Logits.***

From design perspective, following network aspects and training techniques are considered:

- Convolutional neural network architecture
  - Number of convolutional (Conv) layers
  - Specification (kernel, stride, pooling) of each Conv layer

        - Number of fully connected (FC) layers
        - Specification (width) of each FC layer
        - Activation function (ReLU), Dropout, or Regularization if any
- Training parameters
    - Training length (number of epochs)
    - Training method (use batching), and
    - Batch size (128)
    - Optimizer type (Adam)
    - Learning rate and its decay
    - Momentum if any
- Initialization parameters
    - How to initialize network weights and bias

Training is done with the Mini-batch gradient descent method using Adam optimizer, it requires an initial learning rate with dynamic adjustment coming afterward. Considered that the project is not done with GPU and limited available memory, I use a moderate batch size of 128 samples at each training batch. Also training accuracy, validation accuracy and testing accuracy are logged at end of each epoch, then used to produce learning curve at the end to evaluate and visualize the training quality.

Benchmark

As mentioned in Problem Statement, reasonable performance of final model should reach around ~95.0% accuracy on moderate CPU and an inextensive network architecture for an affordable training time (expense).

Methodology

Data Preprocessing

The main data preprocessing before feed them to learner are highlighted as:

- Normalize Input X (image data)
    - Keep using 3 R/G/B color channels vs. converting to grayscale
    - Implement zero-mean normalization scheme on each pixel of each channel (this is important in later steps also, when 5 extra none-GTS images are evaluated)
    - Resize images to shape `(32, 32, 3)` whenever necessary before feed to pipeline, specially this is only needed for the extra 5 none-GTS testing images, since GTS provided training and testing images are already uniformly shaped

- Onehot encoding of output Y (class label)

- For multi-classes classification, this CNN outputs ***Onehot*** encoded ***Logits*** vs. 0~42 scalar number. In line with this and cross entropy computation, I implement One-Hot encoding to all Ys (class labels) in training, validation and testing set.
- Todo this: `y_train_onehot = np.eye(n_classes)[y_train]`

- Split training data set to training and validation set
  - Follow machine learning technique and practice, I split training data to training and validation set using `train_test_split` from `sklearn`, and keep testing data solely used for testing purpose.

Here is an example pair of original and normalized GTS traffic sign images:

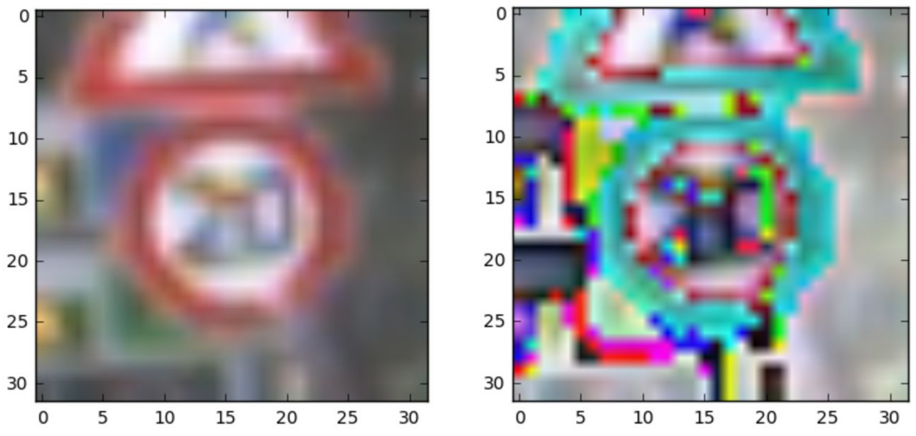

Figure 3. Original (Left) and Normalized (Right) Input Image

Implementation

The implementation process is comprised of two main stages:

1. CNN network implementation stage
2. Training and tuning stage

- For the first stage, I started with very simple network architecture, and gradually increased network complexity to a point that preset benchmark can be achieved. So network design and choice is done along with the training and tuning exercise.

  Here is satisfactory network architecture in the end (not one without room to improve):

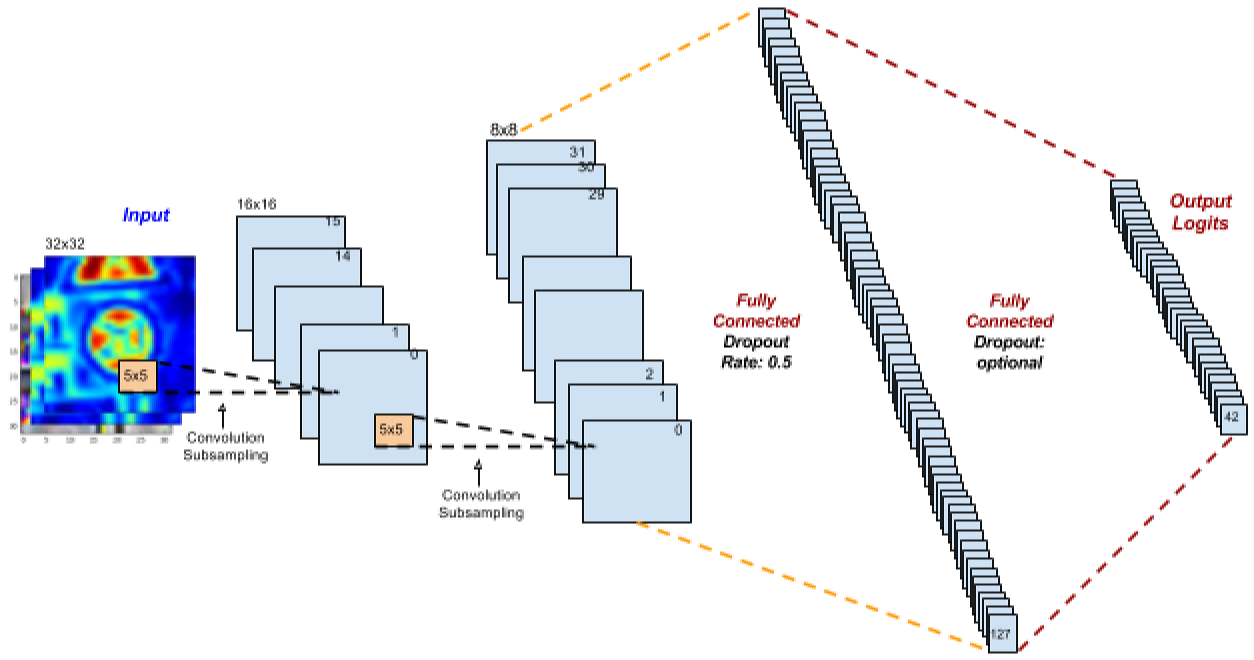

Figure 4. Network Architecture and Layers

- Specifically we have:
  - Conv layer1: [5 x 5] kernel and [1 x 1] stride, *ReLU*
  - Maxpooling1: [2 x 2] pooling
  - Conv layer2: [5 x 5] kernel and [1 x 1] stride, *ReLU*
  - Maxpooling2: [2 x 2] pooling
  - Flatten to: 2048 neurons
  - Fully Connected layer1: 2048 x 128, *ReLU, Dropout (½ keep probability)*
  - Fully Connected layer2: 128 x 43, *Dropout (optional)*
  - Output: ***43 Logits***
- To train and tune the network above, I have:
  - Initialize network weight with truncated normal distribution at very small stddev as `tf.Variable(tf.truncated_normal(shape, stddev=0.001))`
  - Used adaptive ***Adam*** optimizer with a small initial learning rate: `optimizer=tf.train.AdamOptimizer(learning_rate=0.001).minimize(loss)`
  - Implemented `next_batch(...)` to get next training batch samples
  - Evaluated following ***accuracies*** at end of each epoch:
    - *training accuracy:* use one batch accuracy to proximate
    - *validation accuracy:* use whole validation set
    - *testing accuracy:* use whole testing data set
  - Collected accuracy data across epochs, plot following learning curves to guide tuning and exit:
    - Learning Curve: *Accuracy vs. Epoch*

■ Learning Curve: *Loss vs. Epoch*

Below are typical learning curves in the end, following an satisfactory training as:

```
… … …. … … … …. … … … …. …
Epoch:  54, Training   Accuracy: 100.0%, Training   Loss: 0.00001974
Epoch:  54, Validation Accuracy:  99.3%, Validation Loss: 0.03525029
Epoch:  54, Testing    Accuracy:  95.2%, Testing    Loss: 0.40294984
Epoch:  55, Training   Accuracy: 100.0%, Training   Loss: 0.00004667
Epoch:  55, Validation Accuracy:  99.3%, Validation Loss: 0.03255020
Epoch:  55, Testing    Accuracy:  94.9%, Testing    Loss: 0.38307288
Epoch:  56, Training   Accuracy: 100.0%, Training   Loss: 0.00002236
Epoch:  56, Validation Accuracy:  99.3%, Validation Loss: 0.03559887
Epoch:  56, Testing    Accuracy:  95.0%, Testing    Loss: 0.41634461
```

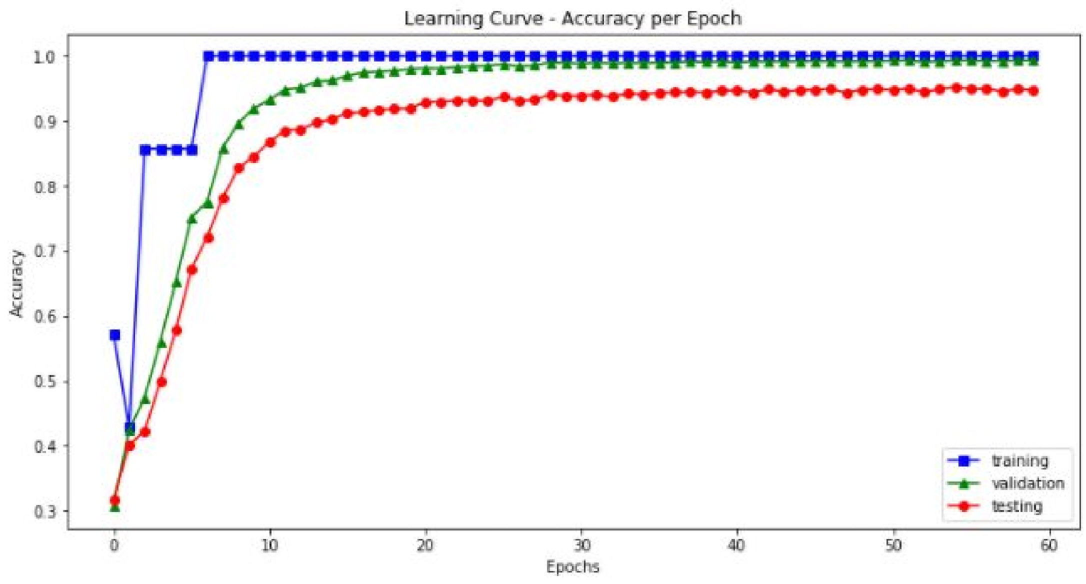

Figure 5. Learning Curve: *Accuracy vs. Epoch*

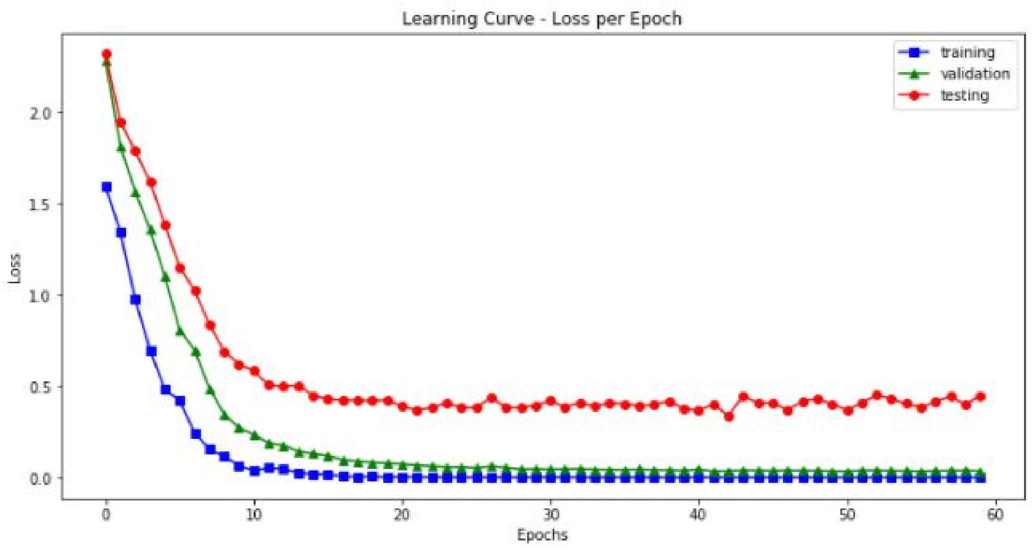

Figure 6. Learning Curve: *Loss vs. Epoch*

Learning curves have demonstrated a clear good training result at end, we have:

- ❖ Neither notable underfit: validation curves converge with training curves
- ❖ Nor notable overfit: testing curves not diverge away from training/validation curve
- ❖ Trained model achieved preset goals from benchmark study:
    - ➢ Overall performed is at ***~95.0% testing Accuracy***
    - ➢ ***Testing Loss at ~0.40***
- ❖ It is an appealing CPU only result w/o using GPU and immense network architecture

Refinement

Deep neural network is prone to overfit, at tuning ***dropout*** is added to ***fully connected layer1*** for final result. Another *Learning Curve: Loss vs Epoch* ***with dropout*** is also provided below for comparison. As we see dropout has improved a lot to the model generalization with better result, without dropout we have higher ***Loss***, and it even started to ***diverge upwards*** as training epochs increases.

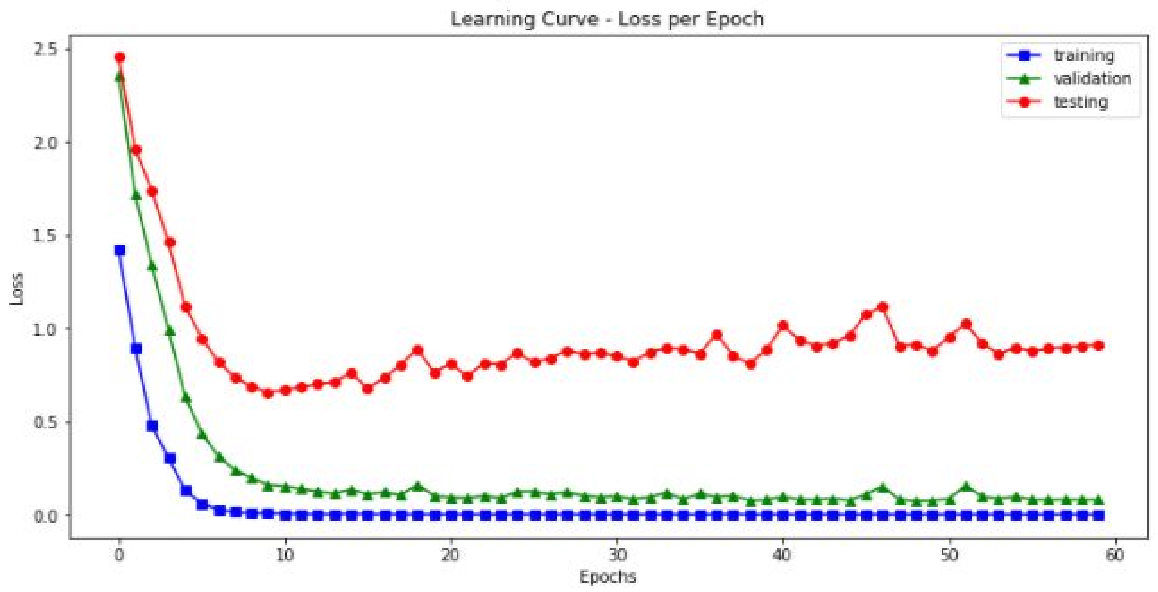

Figure 7. Learning Curve: *Loss vs. Epoch without Dropout (overfitting)*

Another important refinement is use `train_test_split` correctly to split original train dataset ('train.pkl') to training set and validation set.

As Figure 2 depicts, original training dataset is unbalanced, initially when I split it to 2 different (training and validation) subsets, I have not use `train_test_split` carefully to maintain a similar (to original) sample distributions in training and validation set. This has led to a suffered validation accuracy during training and worse overall performance on testing dataset (*`testing accuracy ~= 80.0%`*).

After I fixed training/validation split by using ***stratify*** as

```
X_train, X_val, y_train, y_val = \
    train_test_split(X_train_norm, y_train, test_size=0.2, random_state=101, stratify=y_train)
```

I am able to obtain a much better final result as target (***~95.0% testing Accuracy***)

- ❖ train_test_split need to use stratify=y_label to make sure particular labels ratio is maintained in both training set & validation set, see Sklearn train_test_split document

Here is the plot of final train/validation split using stratify, with sample distribution maintained:

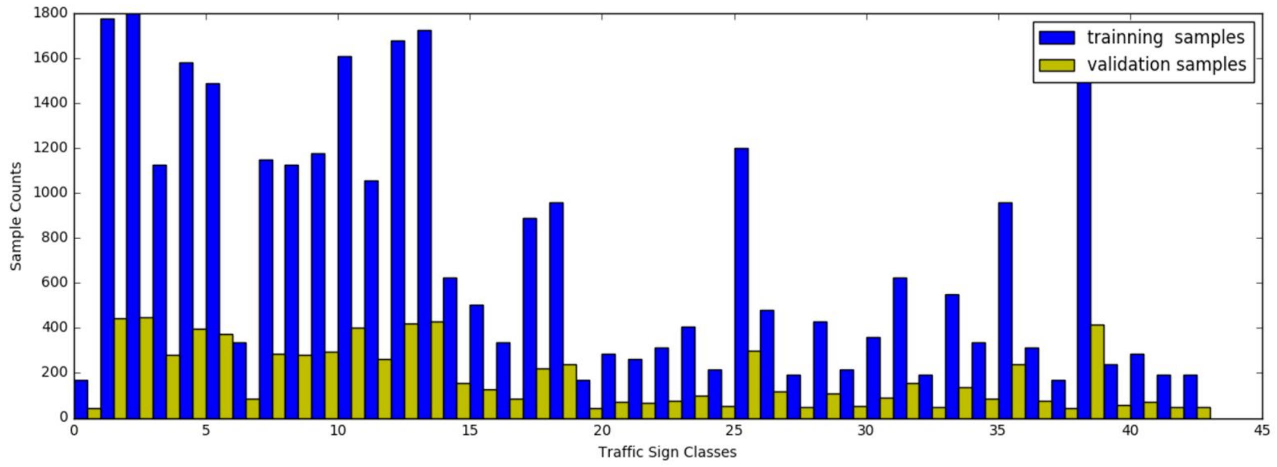

Figure 8. Train/Validation Split with maintained sample ratio using ***Stratify***

Results

Model Evaluation and Validation

As defined in Metrics, ***Accuracy*** is used as metrics to evaluate our model, and indeed I used it's deriving metrics ***Loss*** to guide the whole training process, with ***Accuracy*** itself reported at each step to visualize the quality of the network, robustness of the model and to tell a stop (good enough) threshold.

And here is the summary (*Accuracy and Loss vs. Epoch Learning Curves*):

Validation curve converged with training curve *at* ***HIGH plateau (99.0~100.0%) in Accuracy*** *and* ***very LOW (< ~0.04) in Loss****. It means:*

- *Model is not underfitting (low bias) as both testing and validation have reached very low errors*
- *Model is also not overfitting (low variance), it is general enough with validation; mainly benefit from dropout at Fully Connected Layer1 (1/2 keep probability)*

Testing curve *not converged with training/validation curves, but it* ***plateaued HIGH (~95%) in Accuracy*** *and* ***flattened LOW (< ~0.4) in Loss***

- *It means model is not overfit (on training set), it's general enough to testing dataset as well; mainly benefit from dropout at Fully Connected Layer1*
- *Train-Test performance gap (~4.0% in Accuracy) is mainly due to out-of-samples data (some types of test images never seen in training set)*

❖ Overall model performance (*~95.0% Accuracy, ~0.4 Loss*) in final test is competitive(on CPU)

Justification

Final results **meet** preset target from GTSRB 2011 benchmark at **~95.0% testing accuracy,** and it is significant enough to have the given GTSR problem resolved, mainly because:

- Model accuracy on GTB testing data is close to benchmark
- Training (learning curves) is adequately good enough in that ***Accuracy plateaued High*** and ***Loss flattened Low***

As an interesting experiment, the final model is also used to predict 5 new traffic sign images outside of Germany, to further validate its generality:

- 5 new images:
  - 3 from California: 'Yield_CA', 'STOP_CA', 'NoUTurn_CA'
  - 2 from China: '20kmh_CHN', 'KeepRightLED_CHN'
- Results:

  60.0% Test Accuracy
  - Misclassified: '20kmh_CHN', 'NoUTurn_CA'
- Takeaway:
  - Problem dataset is not big enough to represent global or multi-regional traffic signs

Conclusion

Model Free-Form Visualization

Continue the above discussion of moderate performance (Accuracy: 60.0%) on 5 new (out region) traffic sign images, this diagram below visualizes the test:

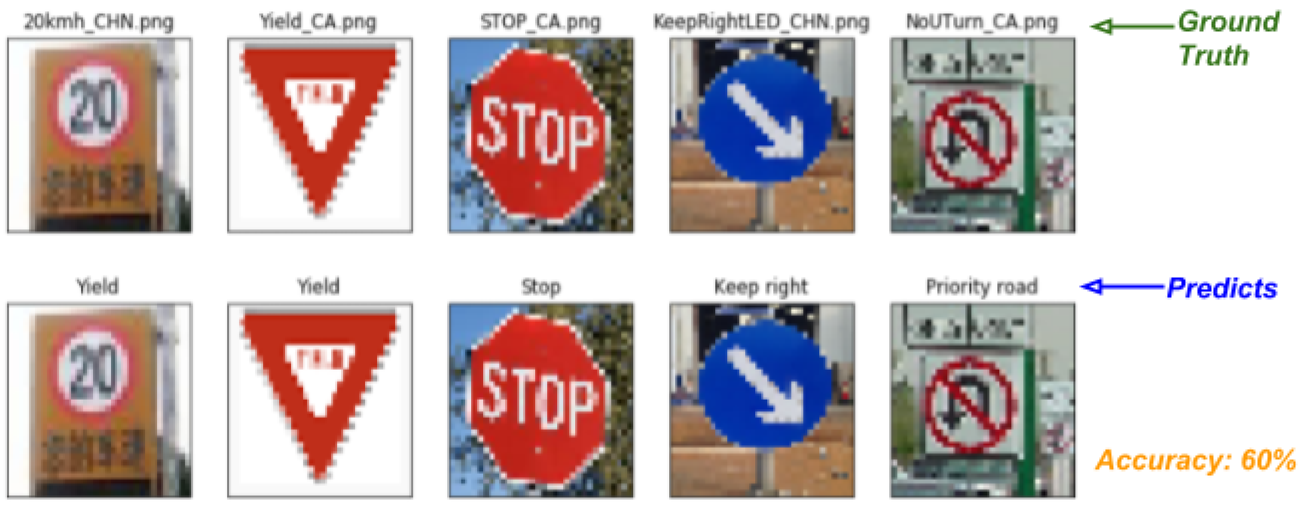

Figure 9. Model performance on out region traffic signs

Diagram below shows the top-5 predictions of misclassified out-of-label-space 'NoUTurn_CA' :

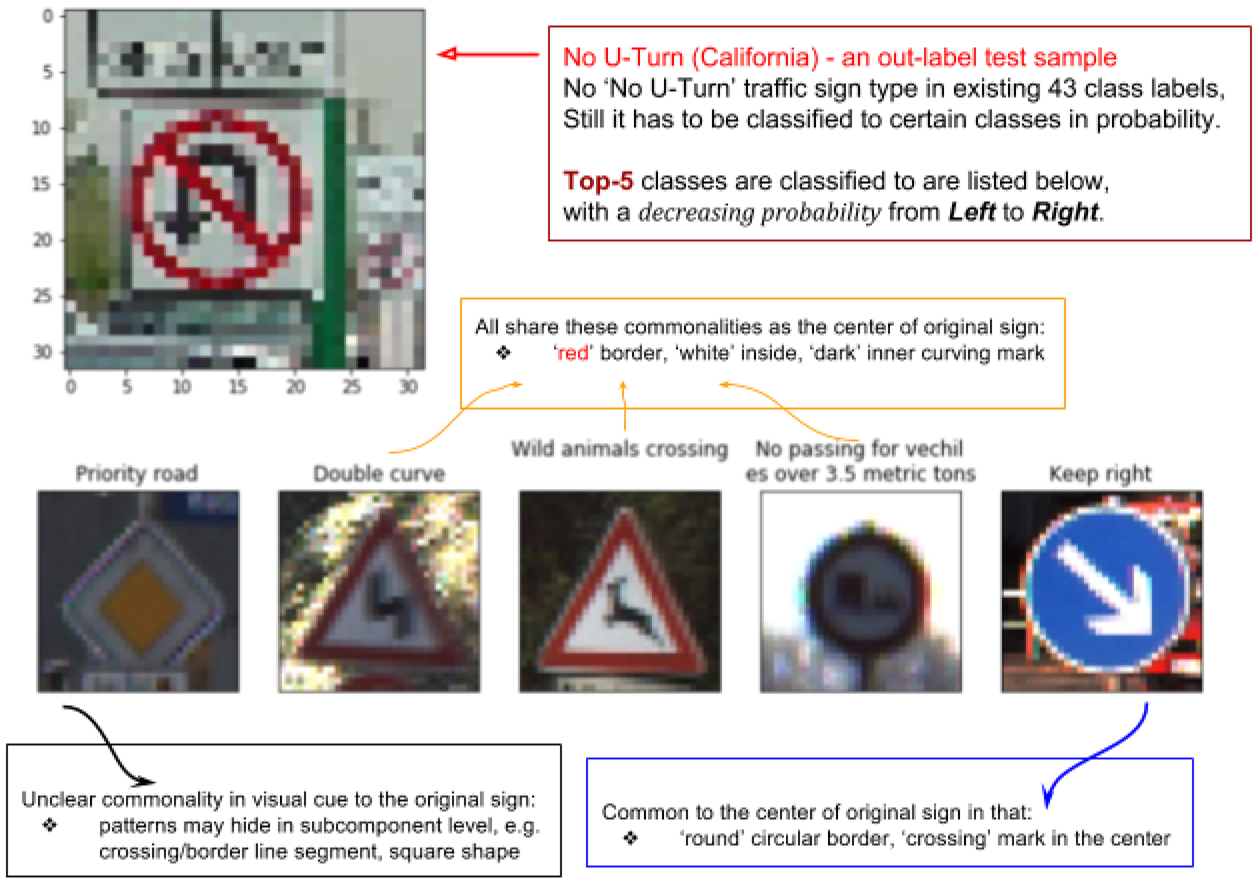
No U-Turn (California) - an out-label test sample
No 'No U-Turn' traffic sign type in existing 43 class labels,
Still it has to be classified to certain classes in probability.
Top-5 classes are classified to are listed below,
with a *decreasing probability* from ***Left*** to ***Right***.
All share these commonalities as the center of original sign:
❖ 'red' border, 'white' inside, 'dark' inner curving mark
Priority road
Double curve
Wild animals crossing
No passing for vechil es over 3.5 metric tons
Keep right
Unclear commonality in visual cue to the original sign:
❖ patterns may hide in subcomponent level, e.g. crossing/border line segment, square shape
Common to the center of original sign in that:
❖ 'round' circular border, 'crossing' mark in the center

Figure 10. visualize Top 5 predictions

Reflection

As a project of deep neural networks in TensorFlow, I have come to following problem solving experience, and some after-thought:

❖ DNN is time consuming to train, harder to debug, so start with simpler model first, increase its complexity as validation or testing results suggest, e.g. observable underfit.

❖ Accompany design with good visualization (e.g. learning curve) to help both evaluate and communicate the model and design.

- ❖ There are quite many DNN specific hyperparameters (learning rate, dropout rate, number of layers, layer width) to tune to reach a good result, I am looking forward to a good DNN/ML library or learning framework, which provides greater optimization automation in requiring minimum human intervention, similar to GridSearchCV in sklearn

- ❖ How does supervised learner, especially expensive DNN learner deal with label space change (e.g. add new traffic sign labels) efficiently and economically? Retrain whole dataset again just for one label insertion seems to be non-optimal. I am interested in finding an answer with some study and research.

Improvement

With current implementation, I have following improvements and enhancements in consideration

- ❖ Transform the model to a Global (or multi-regional) Traffic Sign Classifier, using extended traffic sign data from different countries or regions.

- ❖ Add image augmentation to data preprocessing, to generate more images to traffic signs, especially to classes of lower number of samples, expectedly this will improve generality and accuracy (marginally better given already at ~95.0%) in the same time.

- ❖ Add support or code up other (than dropout) generalization technique for Deep Neural Network, for example to add L1/L2 Regularization terms.

- ❖ Use Computer Vision (CV) technique to normalize images at higher level of the context, for example, automatically find proper bounding box of traffic sign in an image, then extract (and normalize) the traffic sign found to classifier. It is also relevant to the next.

- ❖ Have the model (checkpoint) saved and exported to different systems, there is good opportunity to build a real time Traffic Sign Classification CV Application on mobile (iOS or Android) devices.