Jakub Nowicki, May 22nd, 2020

# Road Signs Classifier

Machine Learning Engineer Nanodegree Capstone Project

## Project Overview

Augmented Reality can give us super-human capabilities. It has potential of completely transforming how we discover world around us, shop, communicate, experience sport or commute. It is one of the areas I investigate in my work. While in many ways, the state of AR displays in 2020 is still disappointing, some solutions are actually well established, like Heads up Displays in cars and planes or in head mounted scanners for warehouse workers. Some AR glasses look almost like regular glasses and consumers could adopt them one day, i.e. North Focals[1] or Vuzix Blade[2].

In this project, I would like to demonstrate how AR navigation or driving assistant applications could benefit from computer vision. I created Road Sign Classifier (RSC) application, which recognises common road signs with high accuracy. The application uses classifier trained on German Traffic Sign Recognition Benchmark images[3] of 43 road sign types and achieves 99.18% accuracy during test, exceeding human performance of 98.84%. It is also very fast with the average prediction time of around 6ms on CPU (AWS t2.medium) and uses very low-resolution (32x32 pixels) grayscale input images, making it suitable for embedding in real-time applications.

I integrated RSC with web application so it can be tried outdoors using smartphones. RSC web application is available here: https://rscwebapp.herokuapp.com.

### Online resources

Summary of online resources created in the project:
- Model development project: https://github.com/j99nowicki/ml_road_signs
- Web application project: https://github.com/j99nowicki/ml_road_signs_web
- Web application live deployment: https://rscwebapp.herokuapp.com

GitHub projects README.md files contain execution instructions.

## Problem statement

The goal is to create an application for common road sign recognition running on any smartphone. The application should have accuracy on par with humans. It should also be fast and have compute and memory requirements suitable for embedded systems.

Summary of tasks:
- Download and prepare German Traffic Sign Recognition Benchmark images (GTSRB),
- Train classifier,
- Deploy a classifier so that can collect images from smartphone cameras,

[1] https://www.bynorth.com/focals
[2] https://www.vuzix.com/products/blade-smart-glasses
[3] http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset

- Show recognised road sign type, confidence level and additional information helping quickly assess potential wrong classifications.

The final application should allow anyone to point their phone camera at sign of one of the recognised types and get classification.

Input image should only contain a road sign. Detecting a road sign in bigger input image is a separate computer vision problem, which I left out of scope for this project. Nevertheless, a complete AR solution would have road sign detection module, therefore classifier must be easy to integrate with it. This should to be taken into account, when assessing classifier metrics.

## Metrics

The project goal is to create solution applicable to real-life AR system so I evaluated it with multiple metrics. I selected one of them to be primary, and others as additional metrics to select one variant over another if they both perform similarly with primary metric. Success of complex real products is often decided by secondary features. I describe all metrics in detail below.

### Primary metric: accuracy

A main metric that I used was accuracy because it is highly relevant to end users and intuitive:

$$accuracy = \frac{correctly\ classified\ images}{all\ images}$$

Accuracy also makes it easy to compare my model's performance with other models.

### Additional metric: confidence

Confidence is a highest value of probability for the model output nodes, defined as:

$$confidence = \max\big(Softmax(output\ tensor)\big)$$

Where $Softmax(output\ tensor)$ is a 1D-tensor with length *j* rescaled so sum of all elements is 1:

$$Softmax(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

### Additional metric: probabilities for all classes

If confidence is low or prediction is incorrect, it is interesting to visualize scores for all classes:

$$probability(x_i) = Softmax(x_i)$$

### Additional metric: confusion matrix

When the model has high performance, incorrectly classified images are rare but can carry important clues for improving the model. Confusion matrix is an effective way of visualizing predictions vs. ground truth labels for all test images.

Table below demonstrates confusion matrix with colour map showing confused classes in shades of yellow, for a test set with 3 classes and 30 images, 10 in each class:

*Table 1 Example confusion matrix*

| | Count of predicted class 1 images | Count of predicted class 1 images | Count of predicted class 1 images |
|---|---|---|---|
| Count of ground truth class 1 images | 10 | 0 | 0 |
| Count of ground truth class 2 images | 2 | 7 | 1 |
| Count of ground truth class 3 images | 2 | 0 | 8 |

When the number of classes is high, confusion matrix is large and it becomes hard to see patterns in errors. Possible solutions is to zero diagonal values before colouring the table, so that full range of colours applies to errors only. If that is not enough, we can print top confused classes cases only.

## Additional metric: top confused classes

To fine-tune a model, which performs well, it is useful to show only the top confused cases, sorted by number of errors. In example above, we might want to list only classes where number of errors is at least 2:

Ground truth class 2, Predicted class 1, count: 2
Ground truth class 3, Predicted class 1, count: 2

Above values would suggest that model has bias towards predicting class 1, which could be result of an imbalanced training dataset, containing more images of class 1 than other classes.

**Important note on confusion analysis**: For solutions like road sign classification, it is particularly important to analyse top confused classes. Imagine for example the model, which has very high accuracy, but confuses "Priority road" with "Yield". I is probable, as both signs have special shapes, so the model could learn to prioritise "unusual shape" feature to differentiate them from other signs, but miss other features relevant for the difference between them. Despite high accuracy, such model could be dangerous.

## Additional metric: evaluation time on CPU

To be applicable in real-time solutions, the model must perform inference quickly on generic purpose CPU. GPUs typically are not used for inference because the acceleration they offer is based on loading multiple images in the same time. To benefit from GPU the system would have to queue many images before inferring classes, usually defying the purpose of acceleration.

In the lack of exact timing requirements for my use case, as a starting point, I rely on some opinions from field of gaming[4]: "*Anything at 100ms or less is considered acceptable for gaming. However, 20-40ms is optimal.*" Inference time will only be part of the total latency. Based on that I set the inference time limit to 20ms, half of the worse optimal value for gaming.

I assume that real-life deployment would use local compute so I only measure time it takes to load image to model and get predicted class label, without end-to-end latency and rendering results in the browser.

[4] https://www.actiontec.com/wifihelp/wifibooster/how-to-reduce-latency-or-lag-in-gaming-2/

### Additional metric: size of model parameters after compression

Big models have high number of parameters, which can be challenging for embedded deployments. That value however needs to be assessed together with the size of baseline environment for inference, before loading model parameters, which is around 230MB for python environment containing PyTorch for CPU inference and other dependencies. Taking into account that in embedded systems we would also need other software, I estimate that the maximum model size should be 100MB.

### Additional metric: input image resolution

Although road sign detection is not in scope of this project, we should consider that the complete AR system would also require detection. Road sign images extracted from larger scenes are likely to be small. A classifier, which performs better on low resolution images, would be preferred because AR system using it would be able to pick up signs from larger distance.

I set the acceptable limit to 100x100 pixels. Images of road sign with that resolution are normally very easy to categorise by humans.

## Data analysis

### German Traffic Sign Recognition Benchmark Overview

Training multiclass image classifier requires tens of thousands of images. Collecting and labelling such dataset is a very complex task. Fortunately, road sign recognition is a not a new problem and there exist publicly available dataset. I used German Traffic Sign Recognition Benchmark (GTSRB) dataset created by Institut für Neuroinformatik[5]. GTSRB was used during a competition in 2011 and has since been used in many ML research projects.

Comparison of my model with some reference designs:

- The best result I found reference of: 99.75%[6]
- Current official state-of-the-art solution: 99.71%[7]
- **My solution: 99.18%**
- Original competition wining solution: 98.98%[8]
- Human performance: 98.84%

### Downloading dataset

Training and test datasets are available here:
Test labels: https://sid.erda.dk/public/archives/daaeac0d7ce1152aea9b61d9f1e19370/GTSRB_Final_Test_GT.zip
Test images: https://sid.erda.dk/public/archives/daaeac0d7ce1152aea9b61d9f1e19370/GTSRB_Final_Test_Images.zip
Training dataset: https://sid.erda.dk/public/archives/daaeac0d7ce1152aea9b61d9f1e19370/GTSRB_Final_Training_Images.zip

[5] http://benchmark.ini.rub.de/?section=gtsrb&subsection=news
[6] https://books.google.co.uk/books?id=jnFdDwAAQBAJ&printsec=frontcover#v=onepage&q&f=false
[7] http://benchmark.ini.rub.de/?section=gtsrb&subsection=results
[8] http://benchmark.ini.rub.de/index.php?section=gtsrb&subsection=results&subsubsection=ijcnn

## Training dataset

Training dataset has 39209 images in 43 directories named 000NN, where NN is a number from 0 to 42. There is also a description text file and additional file with some image features that I did not use because I would not have these features in AR application. Images are in the .ppm file format which is readable by python PIL library.

Training dataset format is supported directly by torchvision.datasets.ImageFolder class.

Below are random examples from each class:

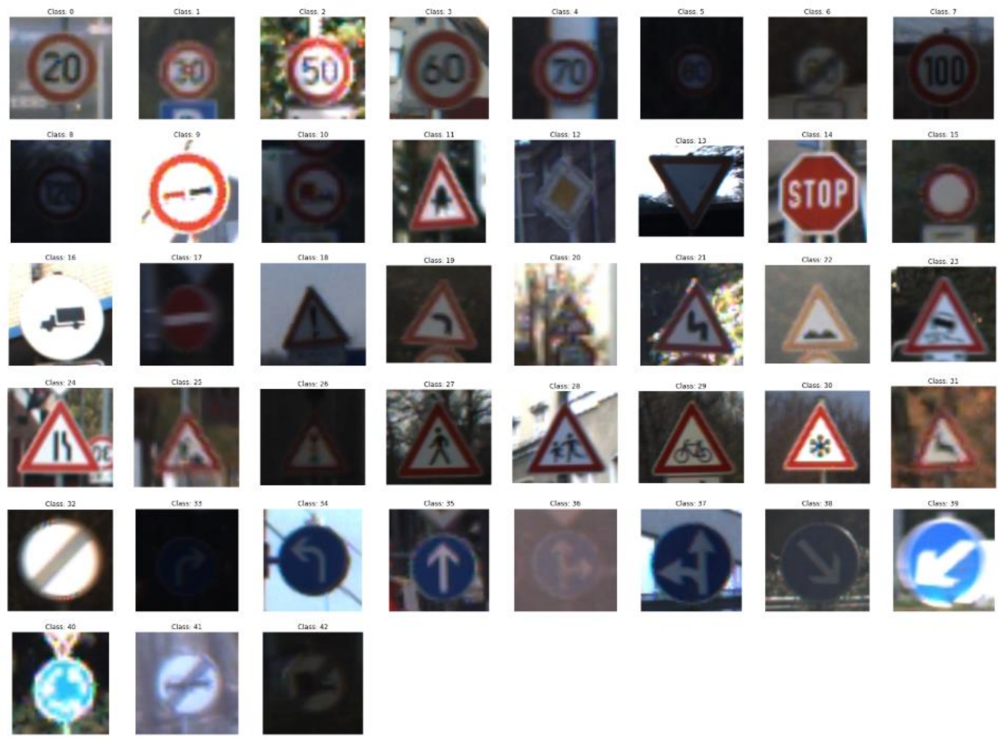


*Figure 1 Sample images from all GTSRB classes*

## Correlation inside the train dataset – tracks

The data contains readme file with the following clarification:

```
**********************************************
Image format and naming
**********************************************
The images are PPM images (RGB color). Files are numbered in two parts:

   XXXXX_YYYYY.ppm

The first part, XXXXX, represents the track number. All images of one class
with identical track numbers originate from one single physical traffic sign.
The second part, YYYYY, is a running number within the track. The temporal order
of the images is preserved.
```

Train images come from short video clips so there are multiple images of the same sign, with slight variation to angle, size etc. If I used images of the same sign to both train and validate I would risk data leakage and overfitting the model. For example, if particular road sign was damaged or had a sticker, we would want the model to learn to ignore such feature. If we use images with the same defect for both training and validation, we would effectively train the model to recognise this defect as integral part of the class. Such model would score low in test.

Below is an example of one such case of the sign with a sticker.

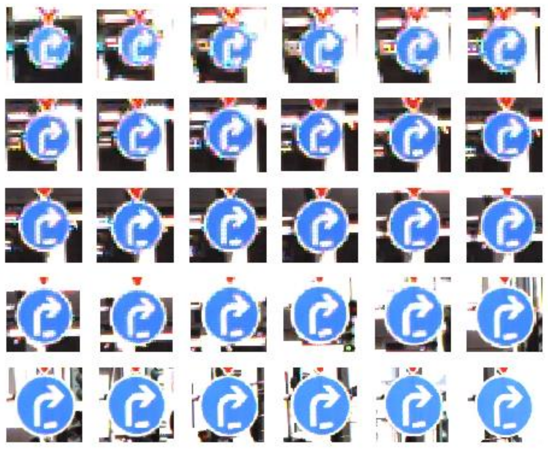


*Figure 2 Visualisation of all images in a single track*

This means that train and evaluation split needs to separate images by sign ("track") name prefix, not a full file name. Each track has 30 images. After counting the tracks in each class directory and drawing a histogram, I see that some of the classes have very few tracks:

```
Lowest number of tracks per class: 7
Highest number of tracks per class:  75
```

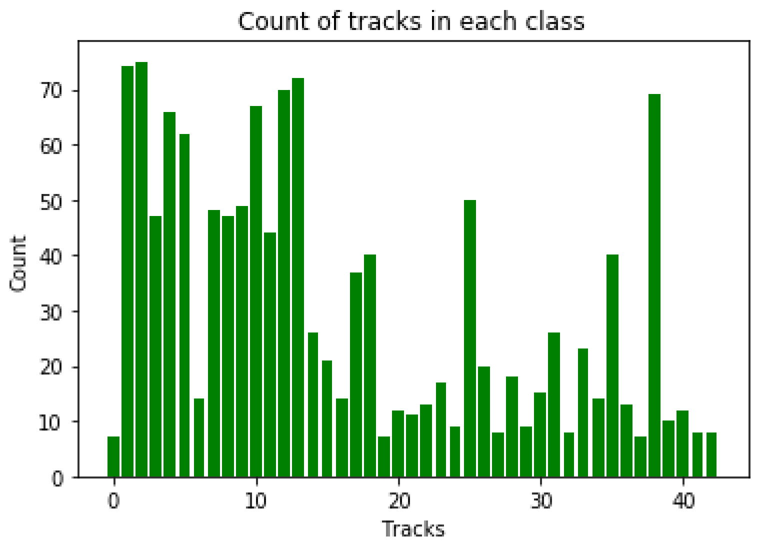


*Figure 3 Histogram of original tracks for each class*

For example, class 00000 with "Speed limit 20 km/h" only has 7 tracks. That high class imbalance was later addressed during model training.

### Test dataset

GTSRB test dataset contains 12630 new images. Test dataset format is different: all image files are in one directory, and there is a separate CSV file with ground truth labels in NN format (not 000NN like training set), and some additional features, that we are not going to use.

Sample from test ground truth labels:

```
Filename;Width;Height;Roi.X1;Roi.Y1;Roi.X2;Roi.Y2;ClassId
00000.ppm;53;54;6;5;48;49;16
00001.ppm;42;45;5;5;36;40;1
00002.ppm;48;52;6;6;43;47;38
00003.ppm;27;29;5;5;22;24;33
00004.ppm;60;57;5;5;55;52;11
```

To load this data I defined a custom Dataset implementation iterating over the data, as defined in PyTorch Dataset documentation: https://pytorch.org/docs/stable/torchvision/datasets.html, changing class name to 000NN format along the way:

In __init__ function uses pandas to create a DataFrame object with all test set information:

```
self.gt_data = pd.read_csv(gt_csv_path, header=0, sep=';')
self.gt_data['_000ClassId']=self.gt_data.ClassId.astype(str).str.zfill(5)
self.transform = transform
```

and returns image and classId in __getitem__ method:

```
img_path = os.path.join(self.images_dir, self.gt_data.iloc[idx, 0])
img = Image.open(img_path)
classId = self.gt_data.iloc[idx, self.gt_data.columns.get_loc('_000ClassId')]
```

**Note on using test set:** After original competition was finished, organisers published test images and ground truth labels. We could in theory use them as a verification data and train using whole training set. I only used test dataset to evaluate models after training, but I still used it multiple times. I noticed it is a common practice even though this way some test data is leaked to the training process and might result in bias, only noticeable when the model is deployed in production.

## Preparing data

After analysing the dataset, I decided to create a separate copy of uniform datasets split to train, validation and test sets and use them with torchvision.datasets.ImageFolder utility class.

### Training data augmentation by reuse and rotation

Despite seemingly big size, after taking into account a high number of classes and the correlation within tracks, the dataset is smaller than it seems. As noticed by some researchers[9] data can be extended by rotating or flipping some signs or by reusing images for other classes after transformations. For example, Left and right turn signs can be flipped horizontally and reused, and "Yield" sign has horizontal symmetry.
I implemented my own version of the augmentation. I managed to find a few more reuse opportunities that the original author missed, and fixed their mistake with roundabout sign reuse.

[9] https://medium.com/@wolfapple/traffic-sign-recognition-2b0c3835e104

Reusing signs for similar class is especially helpful because it creates new samples for classes, which share a lot of common features so are easy to confuse by the model. Instead, when training with reused images, model weights would adjust to detect these differences.

- Classes which can reuse images flipped horizontally from other class: (19,20), (33,34), (36,37), (38,39)
- Classes with horizontal symmetry: 11, 12, 13, 15, 17, 18, 22, 26, 30, 35
- Classes with vertical symmetry: 01, 05, 12, 15, 17
- Classes which can be rotated by 120 or 240 degrees: 40
- Classes which can be rotated by any degree: 15
- Classes which can be reused by rotating by 135 degrees: 35 -> 39
- Classes which diagonal symmetry : 32

I used PIL.Image class to implement transforms. Newly added image files have different prefixes.

This method generated new training data but actually increased imbalance because some rare classes like “Speed limit to 20 km/h” did not benefit from augmentation:

```
Lowest number of tracks per class: 7
Highest number of tracks per class:  280
```

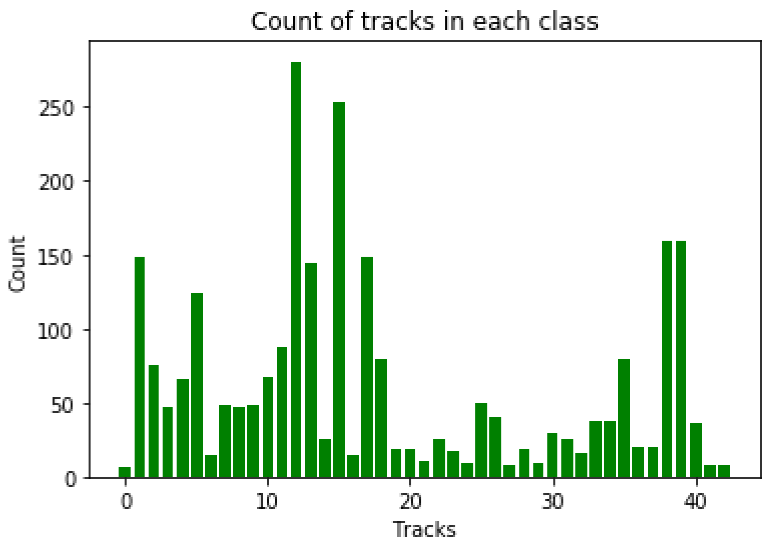


*Figure 4 Histogram of tracks for each class after augmentation by reuse*

I will address this imbalance later. First, I split the training set to training and validation sets, keeping only whole tracks in each set.

I choose 10% validation to training set with minimum of one track for each class in the validation set:

```
train_validation_ratio = 0.1, min_track = 1
```

Before proceeding, I did visual inspection of examples images from each track and check if I did not miss anything. It turned out to be good idea because I discovered that the way I used python os.scandir method to iterate over images for reuse, was causing duplicates in some classes.

In most tracks the last image has the best quality so I used only them to inspect the whole dataset.

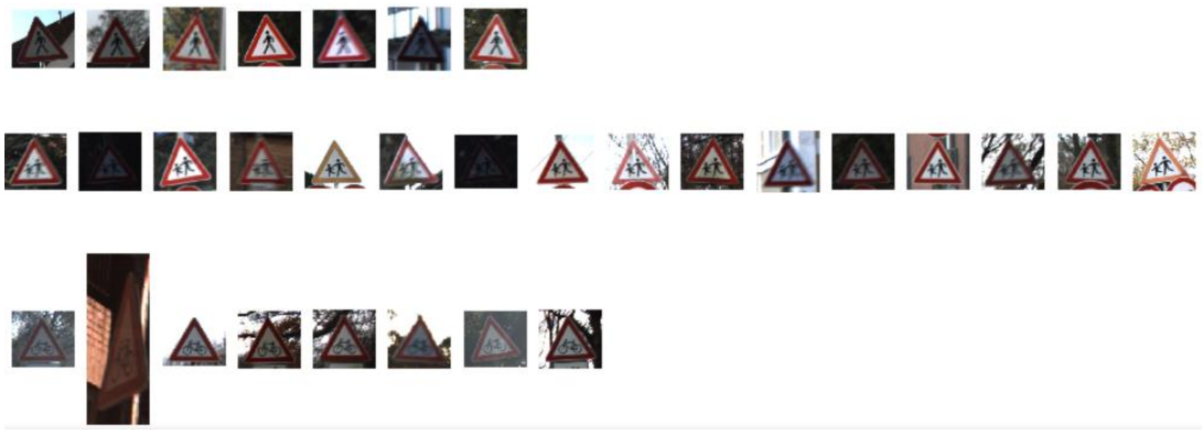


Figure 5 Screenshot from visual inspection of sample images from each track after augmentation by reuse

## Resizing to 32x32 pixels and using grayscale images

One of the leading teams[10] participating in the original competition advised to resize images to 32x32 pixels and transformed them to grayscale. Removing colour information helped to train model faster. The model also scored very high. However, when later investigating some incorrectly labelled images, I could see that colour could help the model avoid the error. Here is an example of poor quality Priority road sign image classified as Roundabout sign. Lack of colour information could be a factor in this inference:

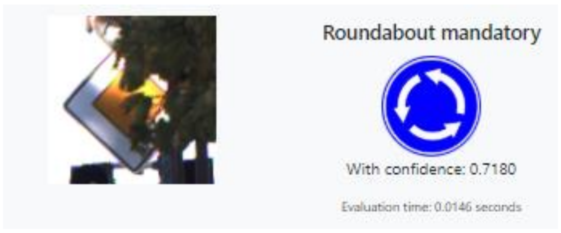


Figure 6 Example of incorrect inference, which could be caused by grayscale input

In future fine-tuning I would like to try again training the model with colour images.

## Addressing training data imbalance

The data has high imbalance so we risk that the model would favour bigger classes as more probable based on input distribution, not features. The imbalance got even worse after reusing images because some small classes like "Speed limit to 20 km/h" could not benefit from it while other classes counts increased significantly.

There are multiple ways to address it. One would be to add weights to CrossEntropyLoss[11] function during training. However, with over 10 times difference between biggest and smallest classes, rare samples from small classes would be giving high error input to back propagation and the model might be difficult to train.

Another approach, which I choose, was data augmentation by applying random transforms with the probability based on class weights, to equalize class sizes in training set. Traffic signs are good candidates for transformations because they would be seen from different angles. We can apply some affine transformations or rotate them.

[10] http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf
[11] https://pytorch.org/docs/stable/nn.html#torch.nn.CrossEntropyLoss

Below code shows the transforms that I applied. I initially followed the guidance on transforms from one of the original competition teams[12], but later decided to remove rotation because some signs could be confused if rotated 15 degrees.

```
normalize = transforms.Normalize(mean=[0.5], std=[0.5])
data_transform = transforms.Compose([
        transforms.Resize((32,32)),
        transforms.Grayscale(1),
        transforms.RandomApply([
            transforms.RandomAffine(0, translate=(0.2, 0.2),
resample=Image.BICUBIC),
            transforms.RandomAffine(0, shear=20, resample=Image.BICUBIC),
            transforms.RandomAffine(0, scale=(0.8, 1.2), resample=Image.BICUBIC)
        ]),
#        transforms.RandomRotation(15),
        transforms.ToTensor(),
        normalize
])
```

The code below plots a histogram of class sizes after weights were used to balance and augment data size to have around 15000 samples in each class.

```
train_size = 43 * 15000

def get_weighted_loader():
    class_sample_count = pd.read_csv('sign_count.csv')['count'].to_numpy()
    class_weights = 1 / class_sample_count
    class_id=0
    sample_weights = list()
    for class_weight in class_weights:
        sample_weights += ([class_weight] * class_sample_count[class_id])
        class_id += 1

    #replace train datsaset
    samp = sampler.WeightedRandomSampler(sample_weights, train_size)
    a = torch.utils.data.DataLoader(image_datasets['train'],
                                    batch_size=batch_size,
                                    num_workers=num_workers_train, sampler=samp)
    return WrappedDataLoader(a, to_device)
```

[12] http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf

```
iterations: 10079
images: 645000
```

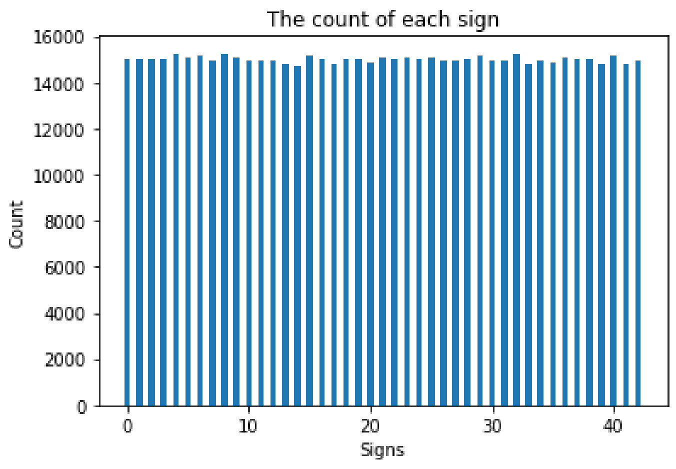


*Figure 7 Count of images in each category after weighted random sampling*

## Test data preparation and uploading to S3

To simplify test data loading I created a copy of test data using the same file structure as training and validation sets, also using the same class names. I did all data investigation and preparation in SageMaker jupyter notebook running on AWS ml.t2.medium instance. Before training, I pushed the data to S3, and pulled them later from another jupyter notebook instance running on ml.p2.xlarge, which is GPU enabled.

I also uploaded a CSV file with class counts, which I used on GPU-enabled instance for randomized data augmentation to remove class imbalance. This helped save time on GPU-enabled instance when re-running the whole training notebook.

## CNN with STN layer

After experimenting with various Convolutional Neural Networks, I selected the architecture shown on the right hand side, which performed the best. The CNN architecture was based on research by Barney Kim[13]. The graph was generated using Netron online tool[14].

The right hand side is the base of the network, which comprises of two parts chained together:

- 3 convolutional layers with batch normalisation steps and additional max pooling layer between 1st and 2nd convolutional layers. Convolutional layers have kernel sizes 5, 3 and 1 respectively.
- 2-layer fully connected classifier with batch normalization and dropout layer

The left hand side is Spatial Transformer Networks, STN, module.

Next, I will highlight key features of the network.

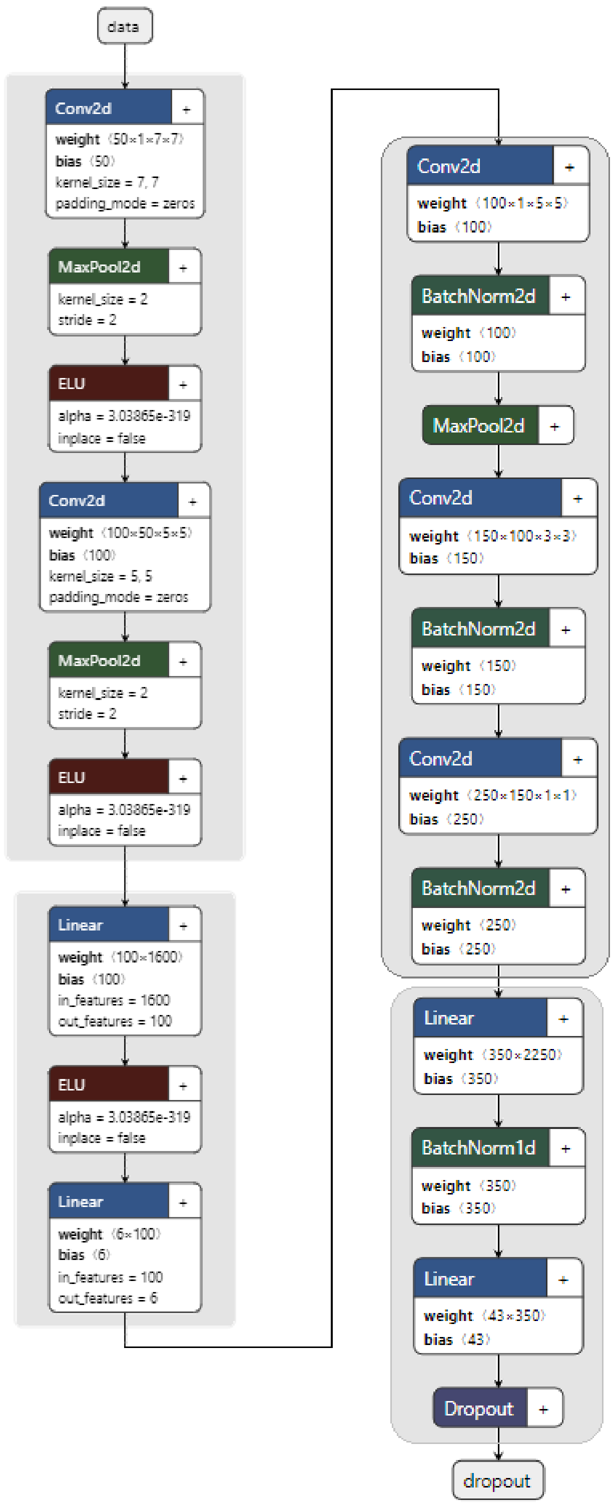


*Figure 8 Final Neural Network architecture*

### Batch Normalization

Neural networks train better if each layer receives normalized inputs. It becomes challenging in multi-layer networks because inputs depend on weights in earlier layers, which change during training. To solve that problem I use Batch Normalization[15]. Batch Normalization 2D variant is applied after each convolutional layer. It is also applied, using 1D variant after first fully connected layer in the classifier.

Using Batch Normalization allows training the network with higher learning rates. It also acts as regularizer, helping to prevent overfitting.

### Dropout

Due to high number of weights, neural networks often suffer from overfitting. This can be limited by using dropout layers, which disable nodes with some probability, in my case 0.5, together with incoming and outgoing edges. This technique prevents individual nodes from "specializing" in some features - accumulating weights very specific to rare inputs, which results in overfitting.

[13] https://medium.com/@wolfapple/traffic-sign-recognition-2b0c3835e104
[14] https://lutzroeder.github.io/netron/
[15] https://arxiv.org/abs/1502.03167

## Spatial Transformer Networks Layer

Spatial Transformer Networks, STN, are visual attention mechanism discovered by DeepMind in 2015[16]. I followed PyTorch tutorial[17], which summarizes it well:

*"Spatial transformer networks (STN for short) allow a neural network to learn how to perform spatial transformations on the input image in order to enhance the geometric invariance of the model. For example, it can crop a region of interest, scale and correct the orientation of an image. It can be a useful mechanism because CNNs are not invariant to rotation and scale and more general affine transformations."*

STN role can be the best visualized on example of images from training of my model. On the left hand side is a grid of batch of 64 input images, resized to 32x32 pixels, transformed to grayscale and with random affine transformations applied. On the right hand side are the same images processed by STN layer. They are cantered and rotated to remove spatial variance.

Adding STN to my model improved test accuracy by around 1.5%.

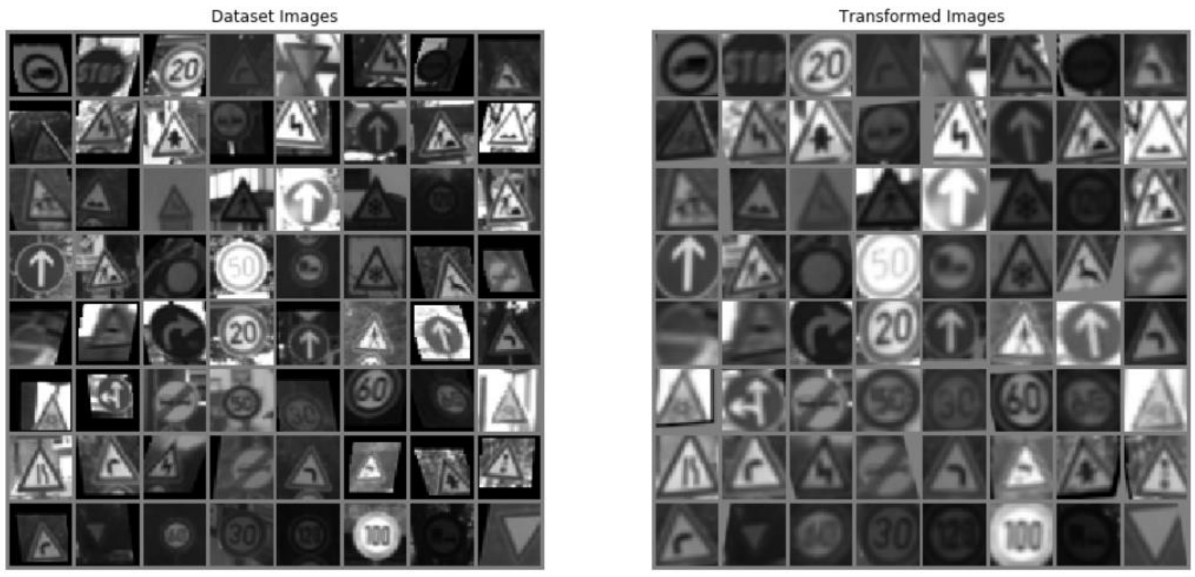


*Figure 9 Input images to the model and output from STN layer*

## ELU activation function

The most common activation function in CNNs is ReLU defined as:

$$ReLU(x) = \max(0, x)$$

Because of its simplicity ReLU is often a default choice and performs well for most networks. However, it suffers from Dead Neuron phenomena: for negative activations, if gradient drops to zero, weights are not adjusted during descent. Neurons which go into that state stop responding to variations in input - because gradient is 0, nothing changes.

[16] https://arxiv.org/abs/1506.02025?context=cs
[17] https://pytorch.org/tutorials/intermediate/spatial_transformer_tutorial.html

That problem can be mitigated by using Exponential Linear Unit, ELU[18], activation function, which has a small slope for negative values.

$$ELU(x) = \begin{cases} x, & x > 0 \\ \alpha(e^x - 1), & x \leq 0 \end{cases}$$

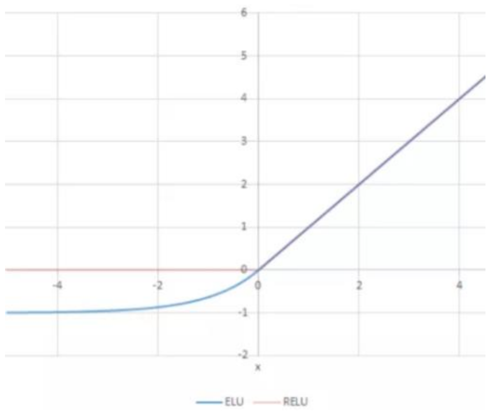


*Figure 10. RELU and ELU activation functions comparison[19]*

## Initialization with normal distribution weights

To speed up training the network I initialize each fully connected layer, except in STN, with random weights with normal distribution:

```
n = m.in_features
y = (1.0/np.sqrt(n))
m.weight.data.normal_(0, y)
m.bias.data.fill_(0)
```

STN layers are already initialized with values for identity transformation.

[18] https://arxiv.org/abs/1511.07289
[19] https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html

## Complete network definition

Below code shows complete definition of the network.

```
class Stn(nn.Module):
    def __init__(self):
        super(Stn, self).__init__()
        # Spatial transformer localization-network
        self.loc_net = nn.Sequential(
            nn.Conv2d(1, 50, 7),
            nn.MaxPool2d(2, 2),
            nn.ELU(),
            nn.Conv2d(50, 100, 5),
            nn.MaxPool2d(2, 2),
            nn.ELU()
        )
        # Regressor for the 3 * 2 affine matrix
        self.fc_loc = nn.Sequential(
            nn.Linear(100 * 4 * 4, 100),
            nn.ELU(),
            nn.Linear(100, 3 * 2)
        )
        # Initialize the weights/bias with identity transformation
        self.fc_loc[2].weight.data.zero_()
        self.fc_loc[2].bias.data.copy_(torch.tensor([1, 0, 0, 0, 1, 0], dtype=torch.float))

    def forward(self, x):
        xs = self.loc_net(x)
        xs = xs.view(-1, 100 * 4 * 4)
        theta = self.fc_loc(xs)
        theta = theta.view(-1, 2, 3)

        grid = F.affine_grid(theta, x.size())
        x = F.grid_sample(x, grid)

        return x

class BaselineNet(nn.Module):
    def __init__(self):
        super(BaselineNet, self).__init__()
        self.stn = Stn()
        self.conv1 = nn.Conv2d(1, 100, 5)
        self.conv1_bn = nn.BatchNorm2d(100)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(100, 150, 3)
        self.conv2_bn = nn.BatchNorm2d(150)
        self.conv3 = nn.Conv2d(150, 250, 1)
        self.conv3_bn = nn.BatchNorm2d(250)
        self.fc1 = nn.Linear(250 * 3 * 3, 350)
        self.fc1_bn = nn.BatchNorm1d(350)
        self.fc2 = nn.Linear(350, 43)
        self.dropout = nn.Dropout(p=0.5)

    def forward(self, x):
        x = self.stn(x)
        x = self.pool(F.elu(self.conv1(x)))
        x = self.dropout(self.conv1_bn(x))
        x = self.pool(F.elu(self.conv2(x)))
        x = self.dropout(self.conv2_bn(x))
        x = self.pool(F.elu(self.conv3(x)))
        x = self.dropout(self.conv3_bn(x))
        x = x.view(-1, 250 * 3 * 3)
        x = F.elu(self.fc1(x))
        x = self.dropout(self.fc1_bn(x))
        x = self.fc2(x)
        return x
model = BaselineNet().to(device)
```

## Model training

I trained the model on Sage Maker jupyter notebook running on GPU enabled ml.p2.xlarge instance. I used Cross Entropy Loss as a criterion and Adam optimizer. I tried with Stochastic Gradient Descent also but it trained very slowly. I experimented with various learning rates for Adam optimizer, and eventually concluded on the following training steps:

1. I start with learning rate 1e-3, train for one epoch, save parameters: parameters _start
2. Reduce learning rate to 1e-4, train for one to three epochs, save parameters: pre_trainted_1
3. Reduce learning rate to 1e-5, train for one to three epochs, save parameters: pre_trainted_2
4. Reduce learning rate to 1e-6, train for one to six epochs, save parameters: pre_trainted_3
5. Reduce learning rate to 1e-7, train for one to six epochs, save parameters: pre_trainted_4
6. Reduce learning rate to 1e-8, train for one to six epochs, save parameters: pre_trainted_5

After each step, if I did not see the model improving, I went back to the last saved parameters from previous step and tried again. It took up to around 10 tries for each step before I could see the results improving enough to move the next step.

I stopped when I could not see any improvements despite many retries with the same learning rate. After each step, I evaluated the model using the test data, which as I described earlier, could be considered "a cheat" because it is not so different to validation process. The model never overfit: validation and test performance were almost identical to the end.

After a few hours of training and experimenting with different parameters, I reached 99.18% accuracy on the test set which I think is very good result, considering the best available network achieve 99.75%.

### Model assessment

I assessed the model by evaluating it on sample images and calculating metrics selected for the project.

#### Evaluation of STN

PyTorch STN tutorial provides nice utilities for STN visualization like in Figure 9 above. It was also useful first visual check if the model was improving. If there was something wrong with the training setup, the model usually trained to the end of planned epochs but did not converge. It was easy to notice something was wrong because STN output images were all grey.

### Evaluation of sample images

I hand-picked a few images from the test dataset:

```
validation_img_paths = ["./source_data_2/test/00/00243.ppm",
                        "./source_data_2/test/01/00001.ppm",
                        "./source_data_2/test/02/00034.ppm"]
img_list = [Image.open(img_path) for img_path in validation_img_paths]
validation_classes = [0, 1, 2]
validation_batch = torch.stack([data_transform(img).to(device)
                                for img in img_list])
```

and calculated predictions for them and normalized them with Softmax:

```
pred_logits_tensor = model(validation_batch)
pred_probs = F.softmax(pred_logits_tensor, dim=1).cpu().data.numpy()
```

I then visualized them with matplotlib, alongside Ground Truth labels, top predicted labels, and confidence scores.

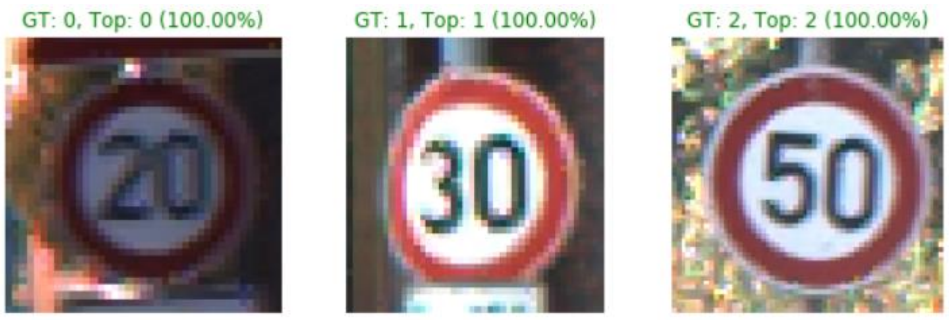


*Figure 11 Evaluation of sample images to quickly assessing the model*

## Evaluation of model metrics on CPU

After quick check if model trained well on GPU, I uploaded the parameters to S3 and evaluated the model on CPU. This way, the evaluation already happened in setup similar to production.

The complete model parameters size is 5MB.

To evaluate accuracy mode I first set it in eval() mode and then wrap the model with torch.no_grad() utility which prevents tracking gradients. Gradients are only needed for learning. Here is the meta code for evaluation method:

```
def evaluate(model, loss_func, dl):
    model.eval()
    with torch.no_grad():
        losses, corrects, nums = [ evaluate the model on given dl ]
        test_loss = np.sum(np.multiply(losses, nums)) / np.sum(nums)
        test_accuracy = np.sum(corrects) / np.sum(nums) * 100

    print(f"Test loss: {test_loss:.6f}\t"
          f"Test accuracy: {test_accuracy:.3f}%")
```

Output (time metrics are added by jupyter *%%time* directive):

```
Test loss: 0.029522      Test accuracy: 99.177%
CPU times: user 1min 11s, sys: 1.26 s, total: 1min 12s
Wall time: 40.5 s
```

### Confusion matrix

To create confusion matrix I iterated over all ground truth labels and predictions and added +1 to a square array with true labels in rows and predicted labels in columns:

```
for t, p in zip(all_labels.cpu().numpy(), all_preds.cpu().numpy()):
        conf_matrix[t, p] += 1
```

I then plot with matplotlib using cmap=plt.cm.Blues to colour the values. As foreseen, correct predictions stretch the colour scale so errors are not visible.

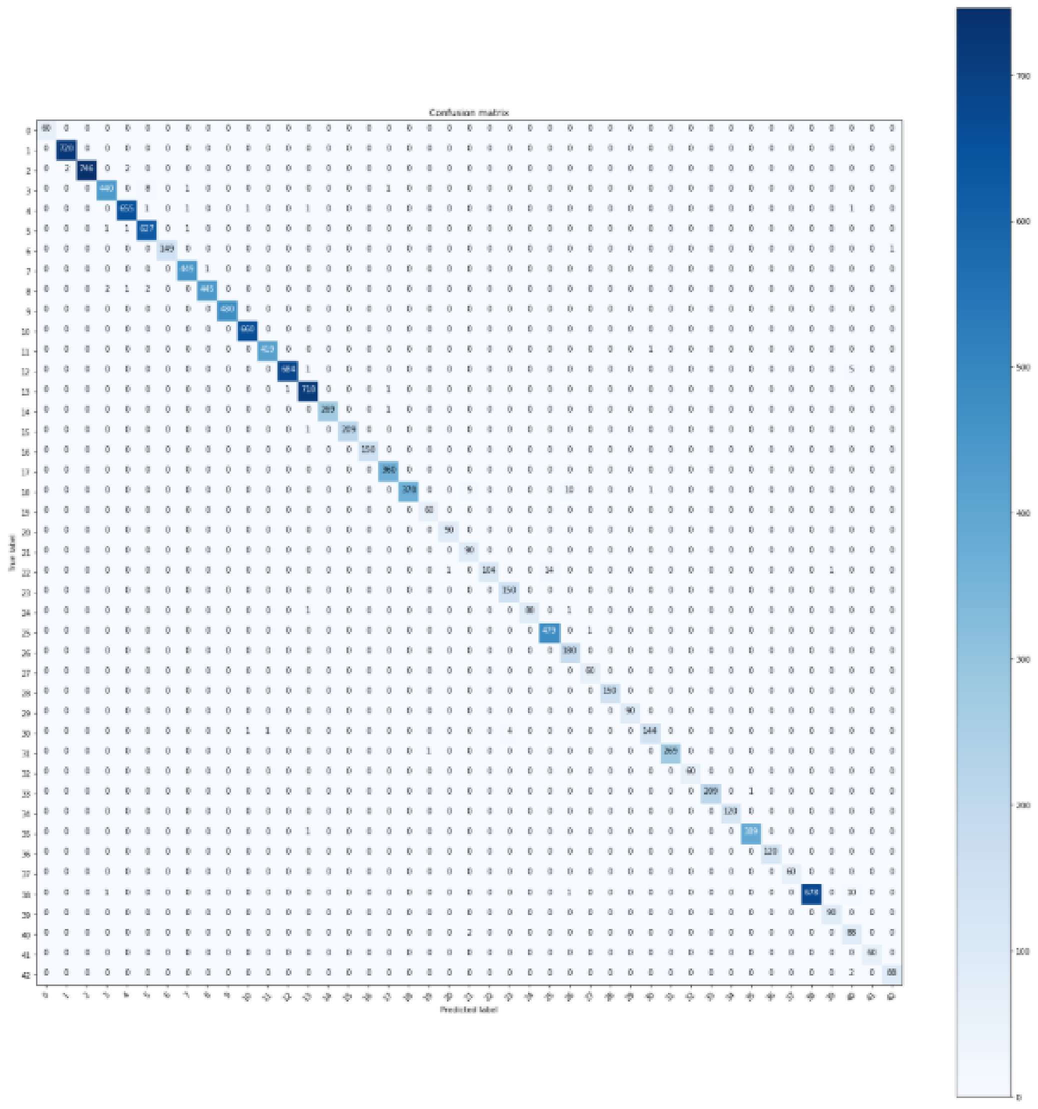


*Figure 12 Confusion matrix*

We can prevent it by filling diagonal with zeros:

```
np.fill_diagonal(conf_matrix, 0)
```

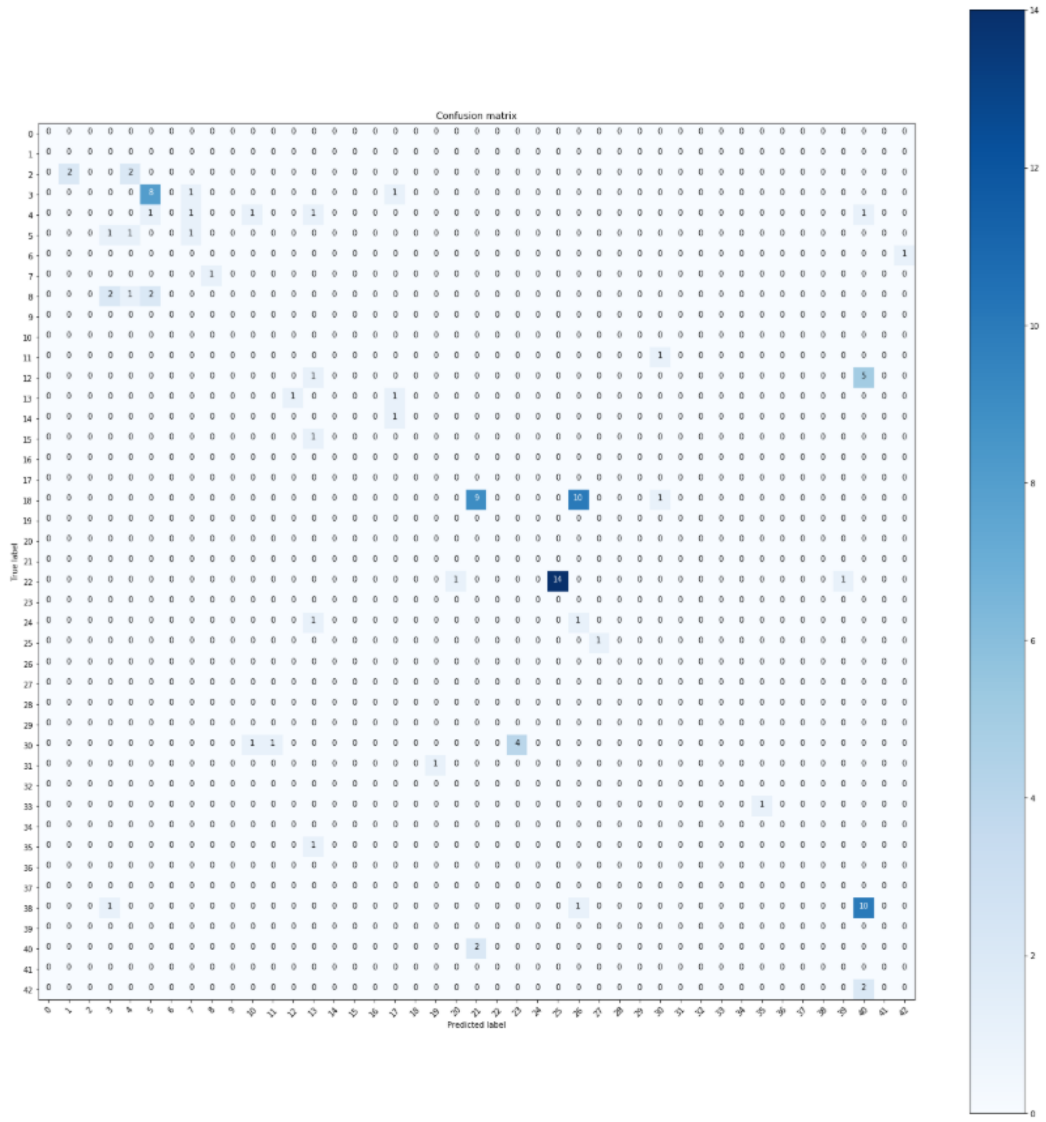


*Figure 13 Confusion matrix with zeros on the diagonal*

Now errors are more visible but they are scattered across the whole matrix and it is difficult to use this information. I used a solution borrowed from fast.ai most_confused()[20] method to print only the classes that had minimum 3 errors:

```
def take_third(elem):
    return elem[2]

def get_top_conf(conf_matrix, min_val = 3):
    np.fill_diagonal(conf_matrix, 0)
    classes = range(43)
    res = [(classes[i],classes[j],conf_matrix[i,j])
                for i,j in zip(*np.where(conf_matrix>=min_val))]
    return sorted(res, key=take_third, reverse=True)

print(get_top_conf(conf_matrix))
```

The output is:

[20] https://github.com/fastai/fastai/blob/master/fastai/train.py#L219

```
[(22, 25, 14), (18, 26, 10), (38, 40, 10), (18, 21, 9), (3, 5, 8), (12, 40, 5), (30
, 23, 4)]
```

Translating that to sign names I can see that the model confused:

| | | |
|---|---|---|
| ["22", "Bumpy road"] | with ["25", "Road works"] | in 14 cases |
| ["18", "General caution"] | with ["26", "Traffic signals"] | in 10 cases |
| ["38", "Keep right"] | with ["40", "Roundabout mandatory"] | in 10 cases |
| ["18", "General caution"] | with ["21", "Double curve"] | in 9 cases |
| ["3", "Speed limit (60km/h)"] | with ["5", "Speed limit (80km/h)"] | in 8 cases |
| ["12", "Priority road"] | with ["40", "Roundabout mandatory"] | in 5 cases |
| ["30", "Beware of ice/snow"] | with ["23", "Slippery road"] | in 4 cases |

Ideally, the model should not be confusing any signs, but none of the errors above would render it dangerous. To further minimise errors, final AR application could perform continuous inference averaging predictions from multiple input images over time. It could also use minimum confidence threshold.

Finally, I measured the time it took to load images and perform inference. It performed 1000 cycles in 5.63s so average inference time on one image was 5.63ms. This is very good for my application.

## Transfer learning

Apart from defining a new Neural Network architecture, I also tried transfer learning[21]. I choose the smallest ResNet architecture, ResNet18[22], which has 45MB parameters size and I followed the PyTorch tutorial "Transfer Learning for Computer Vision" [23].

ResNet18 uses RGB images with 224x224 size. Architecture allows arbitrary size of images but if we want to use pre-trained model it is advised to use that input size. For experiment, I tried it with 32x32, but I was only getting around 44% accuracy. Also, there is no simple way to change input from 3-channel RGB to 1-channel grayscale without losing pre-train weights.

To train a model, I first created a new dataset using image size and normalization parameters recommend for ResNet networks:

```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
data_transform = transforms.Compose([
        transforms.Resize((224,224)),
#        transforms.Grayscale(1),
        transforms.RandomApply([
            transforms.RandomAffine(0, translate=(0.2, 0.2),
resample=Image.BICUBIC),
            transforms.RandomAffine(0, shear=20, resample=Image.BICUBIC),
            transforms.RandomAffine(0, scale=(0.8, 1.2), resample=Image.BICUBIC)
        ]),
#        transforms.RandomRotation(15),
        transforms.ToTensor(),
        normalize
])
```

I then downloaded pre-trained model:

[21] https://cs231n.github.io/transfer-learning/
[22] https://arxiv.org/abs/1512.03385
[23] https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

```
model = models.resnet18(pretrained=True).to(device)
```

froze all its parameters:

```
for param in model.parameters():
    param.requires_grad = False
model.fc = nn.Linear(512, 43).to(device)
```

and changed final fully connected layer to match number of sign categories in GTSRB dataset:

```
model.fc = nn.Linear(512, 43).to(device)
```

I then trained the model using CrossEntropyLoss and Adam optimizer.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=1e-3)
```

I experimented with various learning rates. I received the best results when training:

- 3 epochs with learning rate 1e-3 and then
- one more epoch with 1e-5

The highest test accuracy I achieved was 71.77%. This is much worse than with previous model. The model also takes much longer to train – each epoch takes 54 minutes.

I could see other researchers getting above 99% accuracy with ResNet architectures with the same dataset, particularly using fast.ai[24] framework. Fast.ai applies number of advanced techniques in optimisation and trains deeper layers with lower learning rate than final layers. I would like to investigate that approach more but ResNet18 already scored low on some of my secondary metrics, which I would not be able to improve on, even with much better trained ResNet18 model:

- Bigger models parameters size: 45MB
- Requires higher resolution input image: 224x224 pixels

Therefore, I decided not to use ResNet18 for my application.

## Web application design and implementation

To demonstrate the model I deployed it in the web application. The backend uses Flask web server The frontend of the application uses bootstrap 4.5 responsive grid and stylesheets and some additional CSS for styling. The application is hosted on Heroku cloud platform under URL: http://rscwebapp.herokuapp.com/.

To deploy the webapp I first created locally a Python virtual environment with all dependencies, exported dependencies to requirements.txt using *pip freeze* command, and pushed the application to Heroku git repository.

The application can also be started locally, which is convenient for testing.
Below, I describe main components of the application split into Model View Controller pattern.

### Web application - controller

The main part of the application is routes.py file, which defines entry point URLs for two pages:

- Main page under URL "/" or "*/index*"

[24] https://www.fast.ai/

It renders main page of the application using index.html template. It also selects a random image from test images that the model predicted properly but with low confidence, below 70%.
render.py calls a backend helper method from classifier/cnn_classifier.py, which performs inference and provides data for template rendered.

- Upload image page under URL *"/upload-image"*
  It renders upload_image.html template, which allows user to upload their own image or select an image from the test set samples.
  Before saving the image, it's file name is sanitised to avoid security issues.
  Maximum allowed file size is 5MB and Flask will produce 413 error if user tries to upload bigger image.

There are also auxiliary URLs, which redirect to index.html, populating input file name with user's choice. Different URLs are used to encode different paths to sample images:

- */image/high_confidence/<file_path>*
- */image/low_confidence/<file_path>*
- */image/medium_confidence/<file_path>*
- */image/losers/<file_path>*

render.py also contains initialization method which is called before the first request is served. It:

- Downloads model weights from public S3 bucket
- Downloads sample images from public S3 bucket
- Creates model using the architecture defined during training
- Initializes the model with trained weights
- Creates a PyTorch transform Compose object for input images

Model and transform.Compose objects are then stored in application configuration list object.

## Web application - model

Controller invokes application model, which comprises of a single method call to a method ml_figures() in file cnn_classifier.py.

The method takes as an input a path to the input file and performs the following steps:

- Cleans old files – this is a simple clean-up mechanism to prevent overload
- Applies transformations to the image, resizing, normalization, and mapping to grayscale
- Loads the image to the torch device "cpu"
- Performs inference on the model, gets predictions for all classes and evalution time
- Performs inference on STN layer and gets an output image from STN class
- Creates images for STN input and output visualisation and saves them on the server using random file names
- Produces a plotly graph with predictions for all classes

Then it returns data to be rendered by the frontend:

- figures: dict list containing plotly graph with probabilities for all classes
- predicted_label: a string with predicted label name
- iconpath: a path to predicted sign iconmaxConfidenceValue_str: confidence value for the top prediction
- eval_time_str: time it took to load and evaluate the model
- filename_stn_in: path to STN input file
- filename_stn_out: path to STN output file

## Web application - view

Application fronted comprises of two main HTML template files:

- index.html – renders the main page, including input image, inference and all output results
- upload_image.html – renders Bootstrap accordion object with upload image form and galleries of sample images.
  When upload image form is opened on the smartphone, the browser offers uploading an image directly form phone camera.

I use auxiliary templates for rendering static parts of pages and for galleries with sample images.

## Conclusion

### Example screenshots from the Road Sign Classifier web page

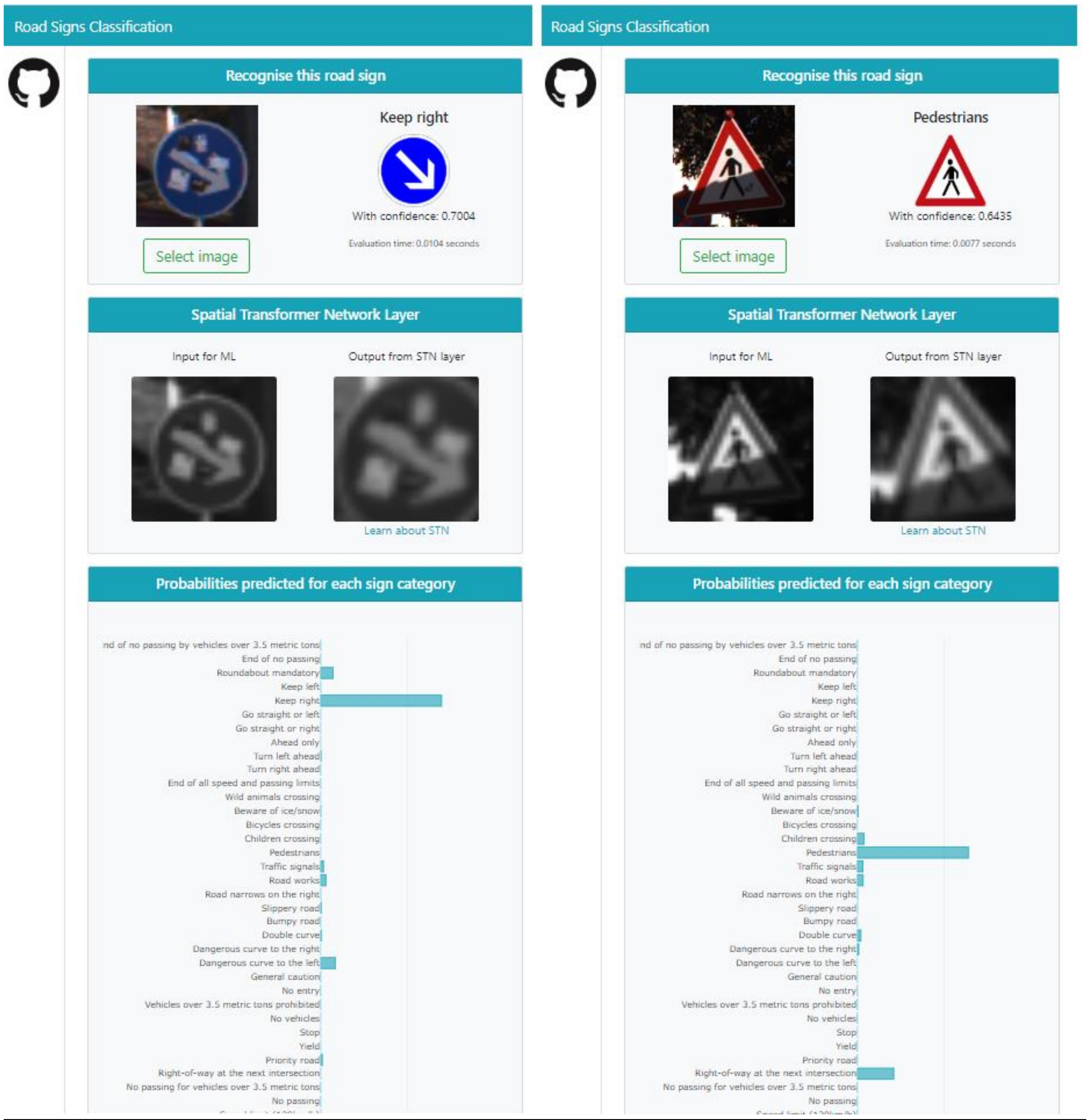
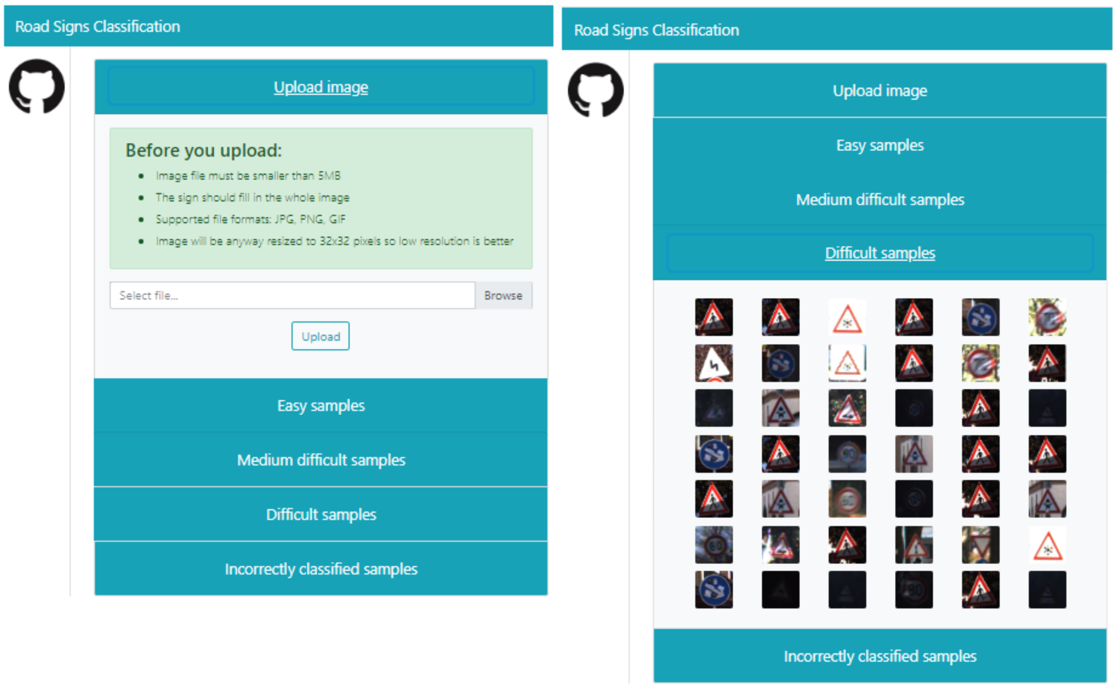


*Figure 14 Screenshots from Road Sign Classifier web application main page*



Before you upload:

- Image file must be smaller than 5MB
- The sign should fill in the whole image
- Supported file formats: JPG, PNG, GIF
- Image will be anyway resized to 32x32 pixels so low resolution is better

*Figure 15 Screenshots from Road Sign Classifier web application image selection page*

## Reflection

Here is the summary of the steps in this project:

1. Defining an initial problem and publicly available dataset
2. Downloading and analysing GTSRB data
3. Pre-processing data
4. Researching for the best model and implementing CNN model
5. Training the model
6. Evaluation of the model metrics
7. Training and evaluation of alternative transform learning model with ResNet18
8. Creating a web application with Flask
9. Deploying the web application in Heroku cloud

I found researching for the model and its training very interesting. The GTSRB dataset has proven to be good choice as it is popular among data scientists and I could find its use with every machine learning technology.

However, most of the time in the project I spent on preparing the dataset and later implementing the web application.

I am very happy that I completed my learning objectives. In particular, I learned how to use PyTorch with more recent innovations like STN, batch normalisation, ELU activation function, transfer learning and apply them to a complete project.

I am also very glad that the final model compares to state of the art models and exceeds human performance.

## Improvements

There are number of improvements I would like to implement in the project:

- I would like to train the CNN model more and with additional automation of the training process.
- I would like to train some ResNet architectures with fast.ai and see how they compare
- The current web page is not easily readable on smartphones
- Error handling in the web page is very rudimental
- Plotly does not work well on mobile Firefox browser

Apart from that, I would also like to implement an image detection model, which would bring my solution a step closer towards a complete AR system.