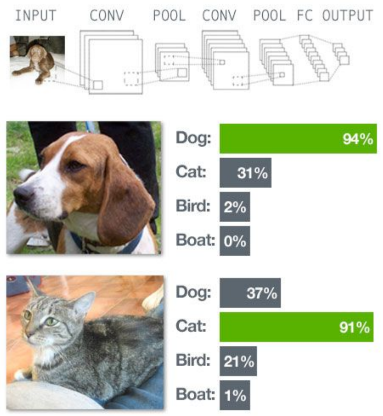
Classifying Traffic Signs With Convolutional Neural Networks

Ronald Wahome
January 16, 2018

1. Definition

1.1 Project Overview

Convolutional Neural Networks (CNN) are a type of Deep Learning (DL) which is a branch of Supervised Machine Learning (ML) that have proven very successful at recognizing and classifying images through their unique structures that model the visual cortex of a mammal's brain[1]. The visual cortex has small regions of cells that are sensitive to specific regions of the visual field. Like the visual cortex of a brain, a CNN has multiple layers that act as filters that process input information for different sub-regions on an image and then pass that information down to the next layers. Every layer learns different features and the final layers put all the learnt information together to output a label. An example would be a CNN learning to classify an image of a dog from a cat image or for the case of this paper, identifying different traffic signs.



Training a CNN involves exposing these neurons to multiple versions of the same label and the more data they are exposed to, the better they will get at classifying an unseen image. A big advantage of CNNs over traditional ML algorithms is that they can discern patterns[2] in data by learning their own filters. For example if we were classifying faces in images, in a traditional algorithm like Decision Trees, we would have to create a filter for eyes if we want this feature to be picked up but with CNNs they discover the eye feature on their own.

We will attempt to use one of these CNNs to classify Traffic Signs from the The German Traffic Sign Benchmark (GTSRB) dataset[3] which is freely available from Institute Fur Neuroinformatik website. The classification on this data was first attempted in 2011 as a classification challenge[4] held at the International Joint Conference on Neural Networks. The challenge was a single-image, multi-class classification with 43 unbalanced classes.

The total number of images is 51,839 that has further been split up into 39,209 training images and 12,360 test images with labels. The competition yielded some very impressive results with the winner achieving an accuracy of 99.46%[5]. This impressive score was achieved with a “Committee of CNNs” and was the only one above the Human Performance[6] of 98.84%. The third place winner achieved an accuracy of 98.31% and was also a form of CNN. Over the years, numerous research has been performed with some classifications proposing an accuracy as high as 99.81%[7].

Rank	**Team**	**Method**	**Correct recognition rate**
1	IDSIA	Committee of CNNs	99.46 %
2	INI	Human Performance	98.84 %
3	sermanet	Multi-Scale CNNs	98.31 %
4	CAOR	Random Forests	96.14 %

Due to recent technological advancements in computer processing power and particularly the Graphic Processing Unit which is faster than a CPU for training neural networks, there has been a surge in interest in CNNs both in the research and business communities.

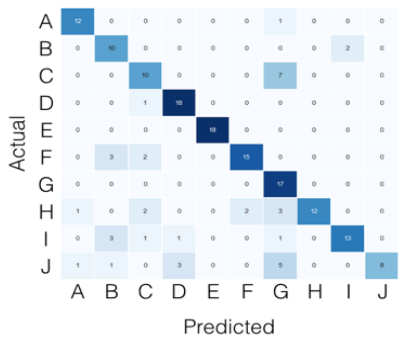
1.2 Problem Statement

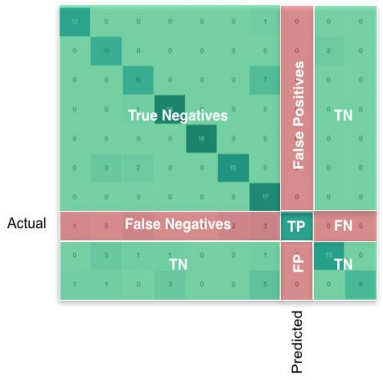
We will build and train a CNN with the goal of surpassing the Human Recognition accuracy rate of 98.84% at classifying traffic images from the dataset. It is important that we exceed these benchmark before we can deploy our algorithm in the real world where real human lives could be at stake. We first explore the data in detail and do appropriate visualizations to understand the kinds of data preprocessing that is needed. After processing the data, we then design a CNN architecture that fits our classification task.

The processed data to be fed into our network is first normalized to allow for efficient computations. We also split our training set to two parts, 20% validation data and 80% training data. Validation data is used by CNNs to validate what they learn as they iterate through the training data.

1.3 Metrics

We use the Accuracy metric to evaluate our CNN model performance but that is only possible after balancing the severe class distribution that exists in the dataset. It was important to pursue this metric so as to appropriately benchmark our model against previous successes on the same dataset as they have also used the same metric. For our case, accuracy is the number of correct predictions made vs all predictions made. We have also generated a multi-class Confusion Matrix (CM) to further help us understand the model's performance. Not just where it excels but more importantly where it fails. A CM is in itself not a metric but it represents a way to obtain other metrics including accuracy.



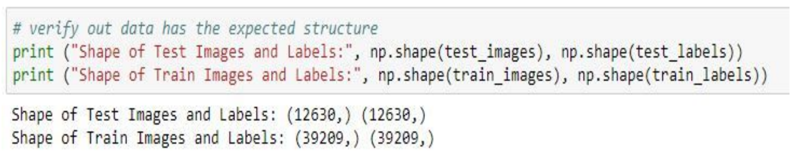


To interpret a multi-class CM like the one above, the values along the mostly populated diagonal represents the intersection of Predicted vs. Actual classes. A useful key for for reading the matrix is shown to the right.

2. Analysis

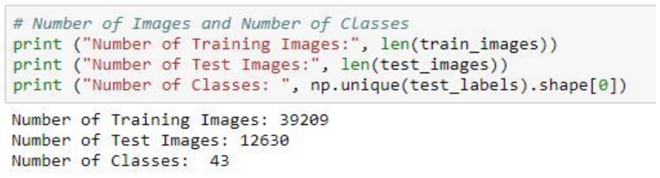
2.1 Data Exploration

We want to answer a few basic questions about our dataset before we can decide if it is ready for input into a machine learning algorithm. First we check to see how many dimensions the images are and then verify the count on both test and train data.



```
# verify out data has the expected structure
print ("Shape of Test Images and Labels:", np.shape(test_images), np.shape(test_labels))
print ("Shape of Train Images and Labels:", np.shape(train_images), np.shape(train_labels))
```

```
Shape of Test Images and Labels: (12630,) (12630,)
Shape of Train Images and Labels: (39209,) (39209,)
```

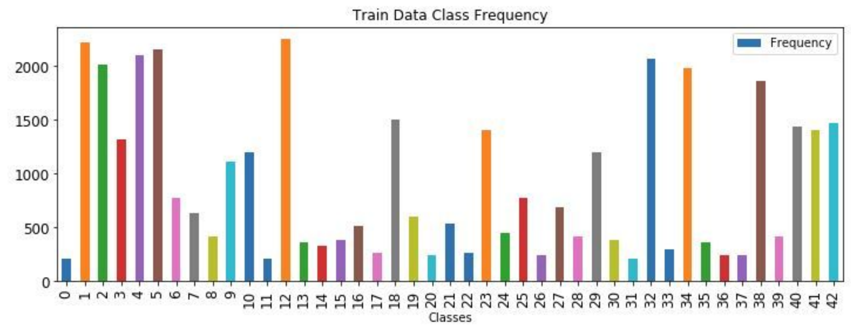


```
# Number of Images and Number of Classes
print ("Number of Training Images:", len(train_images))
print ("Number of Test Images:", len(test_images))
print ("Number of Classes: ", np.unique(test_labels).shape[0])
```

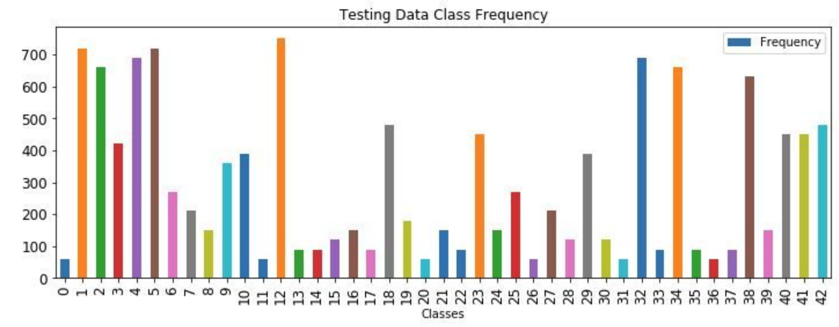
```
Number of Training Images: 39209
Number of Test Images: 12630
Number of Classes:  43
```

The class count is 43 on both sets of data and test images count is 12630 while in the train images we have a count of 39209. Unfortunately though, the class distribution as shown below is highly imbalanced with the class 13 having 2250 images while class 10 and class 13 both have 210 images. We can say that about a fifth of the classes in the training set have a count of between 1300 and 2250 and majority of the rest of the classes are well below that number. The test set have about the same class distribution but since that test data is only used for testing, then the class distribution does not matter as much. The images are 3 dimensional PIL format.

```
Maximum number of Images in a Class: 2250
Minimum amount of Images in a Class: 210
```



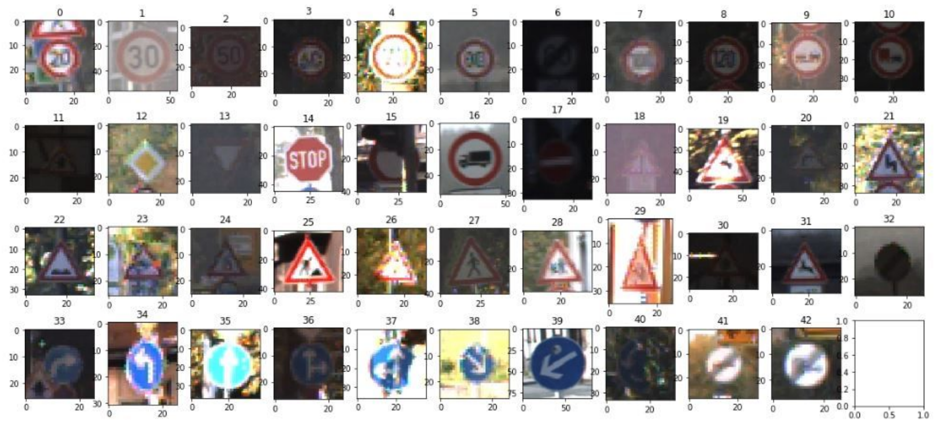
```
Maximum number of Images in a Class: 750
Minimum amount of Images in a Class: 60
```



As mentioned before, using accuracy on an imbalanced classes set will cause us to draw inaccurate conclusions on the model as it will have a bias towards the heavily weighted classes.

2.2 Exploratory Visualization

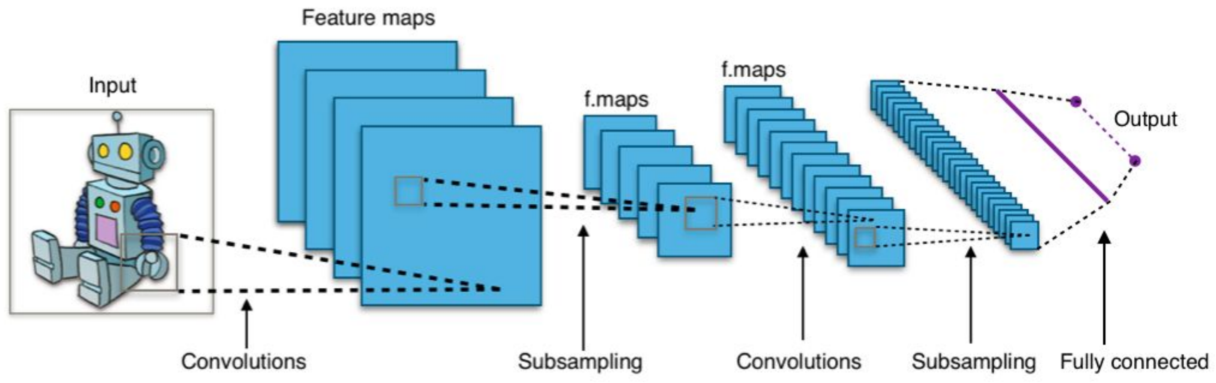
Next we explore the data to learn a few general characteristics about our dataset. We are looking for cues on what kind of image preprocessing we need to perform on the data.



From the visualization above from the training set, we can observe that the images vary in size, light intensity, color intensity, sharpness and even backgrounds. For our model to form a better generalized picture of the input data and representative classes, we will need to work on the above shortcomings in the data preprocessing stage.

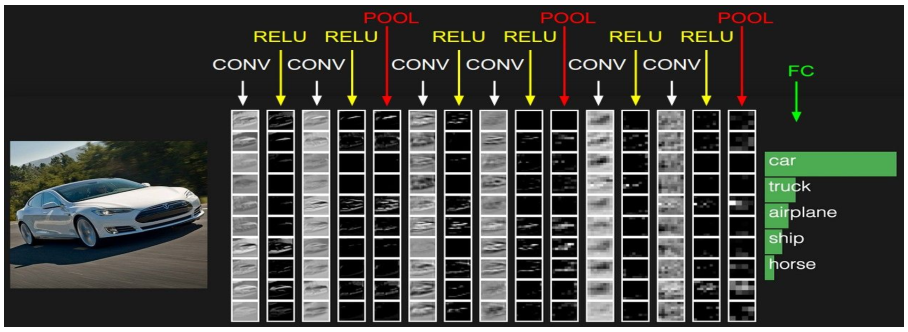
2.3 Algorithms and Techniques

There are multiple Machine Learning algorithms used for tasks like image classifications but few shine[8] in this field as CNNs do. We will use one of these CNNs for our classification task. CNNs build on much of the insights that we have on regular Neural Networks (NN) which receive single vector inputs and transform it through a series of hidden layers where each layer has neurons and each of these neurons is connected to all the neurons in the previous layer and a final layer that does the classification. CNNs however take multi dimensional data like images (width, height and depth) and their neurons are only connected to a small sub-region of the previous layer and their output layer is a smaller (width, height) dimensions but much larger in depth.

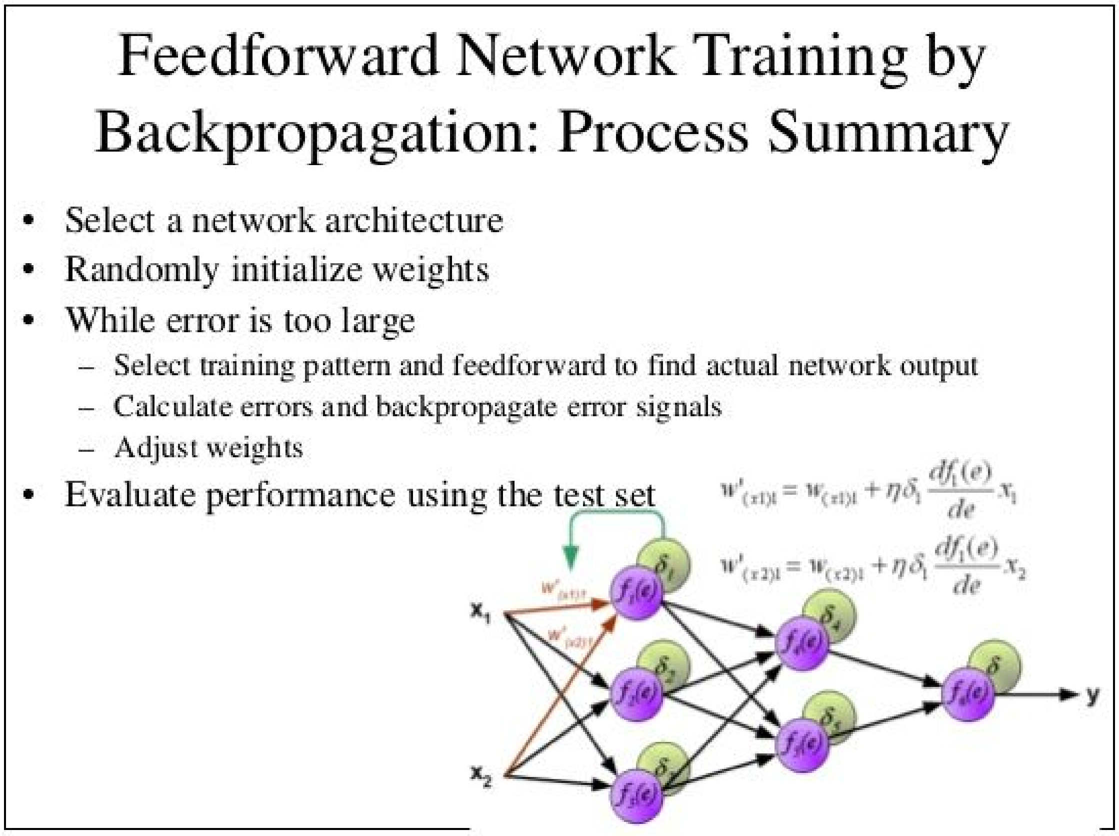


This small subset regions within the CNN layers are called features and they contain information that the network tries to find everywhere in every input. The way the network looks for a match to these filters is how we come up with convolutions. It is basically a filter process that the network tries to match across the entire input image and pass it on to the next layer. A filter is always smaller than the input image. The filter has weights assigned to every filter position. The filter moves across the image from left to right until it covers the entire image. For every new position on the image the filter moves over, it's weights are multiplied by the original pixel values of the image and the multiplictions are summed up at current position. Because the filter size relative to image size is smaller, the resulting outputs for every move across the input image will be of a smaller dimension. This new filter output is called a feature map. The more filters we use the more information we can preserve from the original input image. This process of using filters on inputs is called a convolution. As we go deeper into the network, the output of the previous layer becomes input to the new layer which in turn has its own filters and subsequent filter maps.

What's really fascinating is that the deeper we go into the network the harder it is for the human eye to discern the patterns that the filters are picking up.The first few layers usually pick up easy targets like vertical and horizontal lines and as you go deeper these features are more abstract. Finally you have the fully connected layers that put all this information together and assign probabilities to one the target classes.



However for the network to learn anything and predict classes, it has to do some learning and this is where backpropagation comes in. It can be broken down to four steps. The forward pass, the loss function, backward pass and the weights update.



Feedforward Network Training by Backpropagation: Process Summary

- Select a network architecture
- Randomly initialize weights
- While error is too large
  - Select training pattern and feedforward to find actual network output
  - Calculate errors and backpropagate error signals
  - Adjust weights
- Evaluate performance using the test set

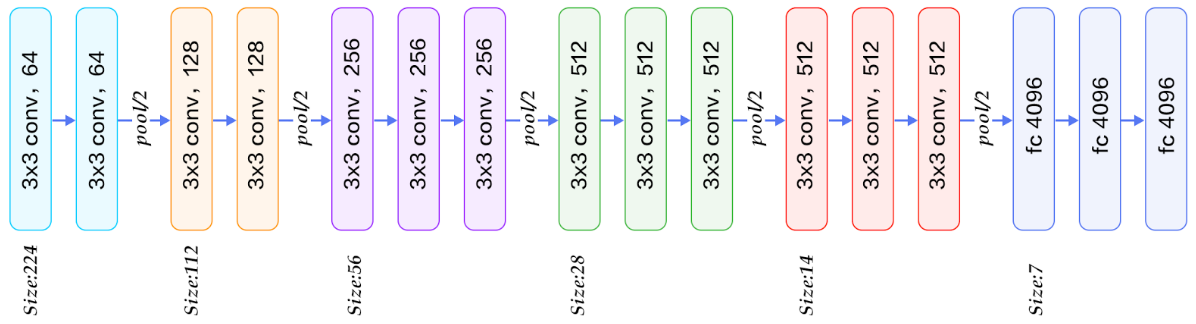
$$w'_{(x1)1} = w_{(x1)1} + \eta\delta_1 \frac{df_1(e)}{de} x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta\delta_1 \frac{df_1(e)}{de} x_2$$

During the forward pass, the network doesn't learn anything since the weights are randomly initialized. Next comes the loss function where the network tries to minimize this error between the predicted and actual image. This is where the network does a backward pass and determines which weights contributed to highest error and adjusts them accordingly. This is the weights update stage of learning where the weights of the filter are adjusted in the opposite direction of the gradient. An optimizer is chosen at this stage and it affects the rate at which the learning occurs. This training stage happens back and forth for a number of chosen batches and hopefully by then the weights have tuned correctly.

CNNs also have Pooling layers and Fully Connected Layers. Pooling is a form of down sampling which helps reduce the spatial size of the layer before input to next layer while preserving the most important information. This pooling has the effect of reducing the total network parameters, reducing computational costs and thus could also help reduce overfitting. The fully connected layers is where the final classifications are made. These layers have connections to activations from previous layer. Other important aspects of our CNN are the Rectified Linear Units (ReLU) and Dropout layers. ReLUs apply activation functions to the convolutional layers which increases nonlinear properties of the decision function and the overall network while Dropout layers help reduce overfitting by dropping out a chosen probability of nodes in a layer and the model is trained on the remaining nodes. The nodes are then reinserted with original weights to the output. By leaving out some of these nodes, we reduce overfitting to the model and has the added benefit of reducing training time.

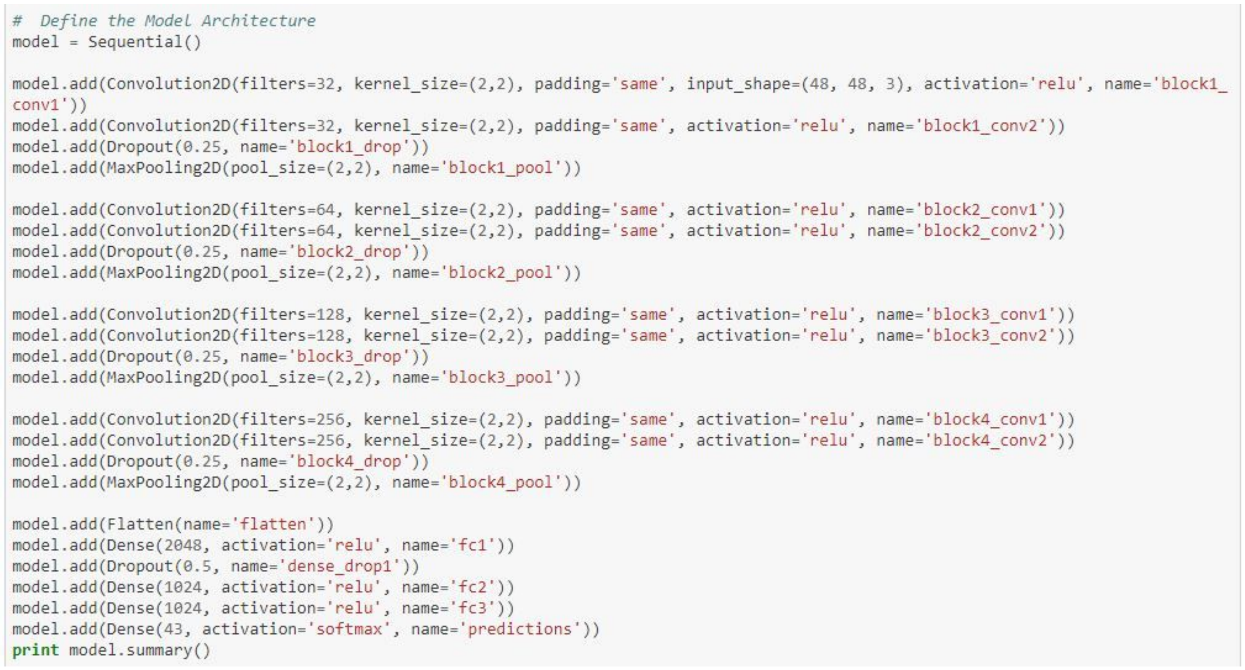
For our classification task, we will draw our inspiration from the VGG16 network[9] which is a CNN named after the Visual Geometry Group from Oxford who developed it and used it to win the ILSVR (ImageNet) competition[10] in 2014. It is a very elegant model and quite easy to understand. We will use the general architecture and change a few things to suit our purposes and after a bit of training we can narrow down the final parameters. Below is an image of the VGG16 architecture.



Our model will be built with the Keras library running a TensorFlow backend. Keras is a high level library for machine learning that allows for fast experimentation and iteration when doing research. The inputs for the model will consist of preprocessed training images and labels. The images are of size 48*48*3 with the first two dimensions representing pixels and the last dimension representing the RGB color channels. After quite a bit of experimentation (trial and error) we believe the following architecture will be a good place to start. The major differences between the VGG16 architecture and our starting architecture at this point is the input shape to match our input, we also use a 2*2 kernel size compared to the original 3*3. A dropout layer of 0.25 after every Convolutional layer is also added and, two fully connected layers of size 1024 followed by a 0.5 dropout layer before the last classification layer.

We will apply one-hot encoding to the labels for the model to accept our class labels. The input images will also be rescaled by dividing the image numpy array by 255 and the ‘categorical cross entropy’ error function which is the appropriate function for our multiclass classification task will be used for training. Finally, the input will also be reshaped to a four dimensional array which is the format that Keras/Tensorflow accepts as input data.

Same Padding is used on all the Convolutional layers so as not to loose any extra information at the image edges. In total the final model has 21 layers. For our optimizers, we'll go for the 'adam' optimizer with default parameters as it has been shown to be most effective[11] as it converges faster with fewer iterations. Below is a visualization of the model configuration.



```
#  Define the Model Architecture
model = Sequential()

model.add(Convolution2D(filters=32, kernel_size=(2,2), padding='same', input_shape=(48, 48, 3), activation='relu', name='block1_
conv1'))
model.add(Convolution2D(filters=32, kernel_size=(2,2), padding='same', activation='relu', name='block1_conv2'))
model.add(Dropout(0.25, name='block1_drop'))
model.add(MaxPooling2D(pool_size=(2,2), name='block1_pool'))

model.add(Convolution2D(filters=64, kernel_size=(2,2), padding='same', activation='relu', name='block2_conv1'))
model.add(Convolution2D(filters=64, kernel_size=(2,2), padding='same', activation='relu', name='block2_conv2'))
model.add(Dropout(0.25, name='block2_drop'))
model.add(MaxPooling2D(pool_size=(2,2), name='block2_pool'))

model.add(Convolution2D(filters=128, kernel_size=(2,2), padding='same', activation='relu', name='block3_conv1'))
model.add(Convolution2D(filters=128, kernel_size=(2,2), padding='same', activation='relu', name='block3_conv2'))
model.add(Dropout(0.25, name='block3_drop'))
model.add(MaxPooling2D(pool_size=(2,2), name='block3_pool'))

model.add(Convolution2D(filters=256, kernel_size=(2,2), padding='same', activation='relu', name='block4_conv1'))
model.add(Convolution2D(filters=256, kernel_size=(2,2), padding='same', activation='relu', name='block4_conv2'))
model.add(Dropout(0.25, name='block4_drop'))
model.add(MaxPooling2D(pool_size=(2,2), name='block4_pool'))

model.add(Flatten(name='flatten'))
model.add(Dense(2048, activation='relu', name='fc1'))
model.add(Dropout(0.5, name='dense_drop1'))
model.add(Dense(1024, activation='relu', name='fc2'))
model.add(Dense(1024, activation='relu', name='fc3'))
model.add(Dense(43, activation='softmax', name='predictions'))
print model.summary()
```

Finally, before out input image is fed to our network but first it will need to be converted to a four dimensional tensor in line with the format that TensorFlow takes as input. Since we are going to experiment with both RGB and Grayscale images, their current shape needs to be changed to a four dimensional tensor which is the format that our CNN can take. The current shapes of our inputs ave visualized below.

2.4 Benchmark

We would like to design and train a CNN that will exceed the human performance accuracy of 98.84%. This may be a relatively high goal for a conventional CNN considering that the other models that have performed better than these threshold were advanced CNNs. Our experience at this level may also be a bit restricting in achieving this high bar but we would still like to build a network with this goal in mind. Worst case scenario, it should be a great opportunity to understand where our model might need more attention.

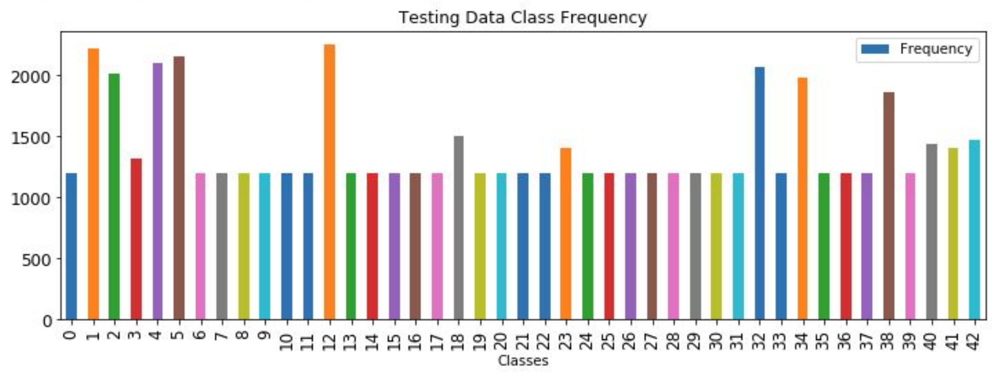
3. Methodology

3.1 Data Preprocessing

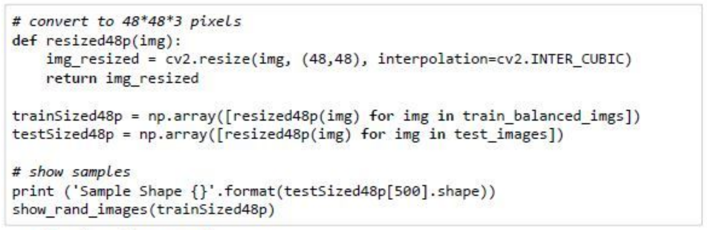
For our model to achieve a good performance, we are going to have to deal with the irregular data observations we made above.

i.)
Balance Classes: To balance the classes we generate additional images in the under represented classes through geometric image transformations. It's important that we also don't over do it to the point where the images don't actually represent the class anymore. We do this on the training set by using OpenCV's extensive image manipulation library[14] to rotate new images by a range of 30 degrees, we also shear and translate these new images within a small range. We loop through the train image directory and generate random augmentations if the total class count is less than 1200. Some resulting random visualizations of these sample images and the new class distribution graph are shown below.

```
Maximum number of Images in a Class: 2250
Minimum amount of Images in a Class: 1200
```



ii.)
Resize the test and train images to a uniform 48*48*3. A point to note here is that any preprocessing done from here on will happen on the new train set that contains the generated augmented images from above.



```python
# convert to 48*48*3 pixels
def resized48p(img):
    img_resized = cv2.resize(img, (48,48), interpolation=cv2.INTER_CUBIC)
    return img_resized

trainSized48p = np.array([resized48p(img) for img in train_balanced_imgs])
testSized48p = np.array([resized48p(img) for img in test_images])

# show samples
print ('Sample Shape {}'.format(testSized48p[500].shape))
show_rand_images(trainSized48p)
```

```
Sample Shape (48, 48, 3)
```

iii.)
Histogram Equalization. Type 1

```
# second variation of Histogram Equalization

def hist_equalCLAHE(img):
    lab = cv2.cvtColor(img, cv2.COLOR_RGB2LAB)
    lab_planes = cv2.split(lab)
    clahe = cv2.createCLAHE(clipLimit=4.0, tileGridSize=(2,2)) # default clip=2.0
    lab_planes[0] = clahe.apply(lab_planes[0])
    lab = cv2.merge(lab_planes)
    rgb = cv2.cvtColor(lab, cv2.COLOR_LAB2RGB)
    return rgb

trainSizedColorCLAHE48p = np.array([hist_equalCLAHE(img) for img in trainSized48p])
testSizedColorCLAHE48p = np.array([hist_equalCLAHE(img) for img in testSized48p])

show_rand_images(trainSizedColorCLAHE48p)
```

Histogram Equalization. Type 2

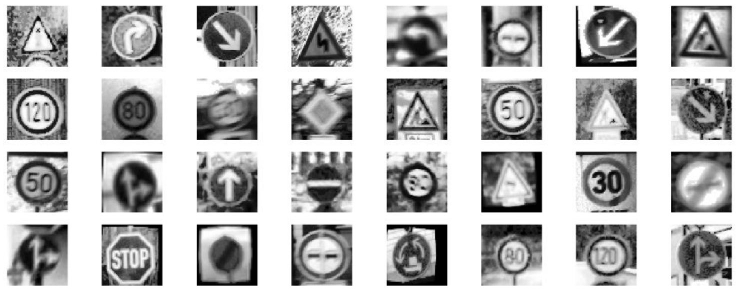
```
# one variation of Histogram Equalization
def color_img_hist_equal(img):
    img_yuv = cv2.cvtColor(img, cv2.COLOR_BGR2YUV)
    # equalize the histogram of the Y channel
    img_yuv[:,:,0] = cv2.equalizeHist(img_yuv[:,:,0])
    # convert the YUV image back to RGB format
    img_output = cv2.cvtColor(img_yuv, cv2.COLOR_YUV2BGR)
    return img_output

trainSizedColorHist48p = np.array([color_img_hist_equal(img) for img in trainSized48p])
testSizedColorHist48p = np.array([color_img_hist_equal(img) for img in testSized48p])

# show samples
show_rand_images(trainSizedColorHist48p)
```

Note: We used two different functions to experiment with different OpenCV histogram equalizations methods.

iv.) Create Grayscale Images visualizations.

```
# convert to grayscale
def conv_2_gray(img):
    im_gray = np.zeros(img.shape,dtype=np.uint8)
    im_gray = cv2.cvtColor(img,cv2.COLOR_RGB2GRAY)
    return im_gray

trainEqualizedGray48p = np.array([conv_2_gray(img) for img in trainSizedColorHist48p])
testEqualizedGray48p = np.array([conv_2_gray(img) for img in testSizedColorHist48p])

# show samples
print ('Sample Shape {}'.format(trainEqualizedGray48p[500].shape))
show_rand_images((trainEqualizedGray48p),grayscale=True)
```

```
Sample Shape (48, 48)
```



At this point we have two unique sets of train and test datasets and we hope to experiment with both RGB histogram equalized and Grayscale histogram equalized Images.

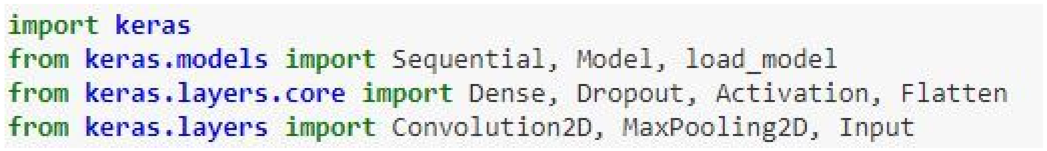
3.2 Implementation

First thing we did was run our model architecture from above that borrows from the VGG16 with both sets of data and then after multiple observations of different combination of parameters, we were able to start leaning in one direction over the other on which parameters to tweak further.

To use Keras for running our model, we first have to import it with the command `import keras` then initiate the model with the command `model = Sequential()`. This creates an instance of the model architecture passed after this command. We add the different layers to our model by using the Keras `model.add(“layer details go here!”)`. For our purpose, we will need to `import from Keras Convolution2D` which lets us add the convolutional layers by specifying the number of filters, size of the convolutional kernel, padding option which controls how Keras deals with image edges depending on kernel size and finally the activation function for that layer.

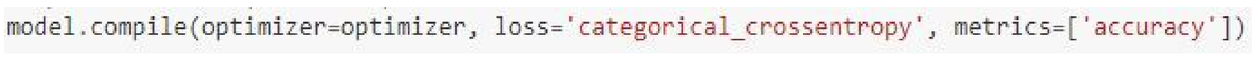
The first layer is always a convolutional layer but we also have to add the input at this stage by specifying the input shape which is always a 4D tensor. At the moment, these inputs are 3D for the RGB images and 2D for the Grayscale but we will have to convert them to 4D tensors before we can feed them to out model. Tensorflow takes inputs of shape `[None, inputSize, inputSize, depth]`. None represents batch size since tensorflow trains models with batches and can be 1 for single image to length of input data array. So for our case we will have to convert them as follows. RGBs to `inputArray.reshape(inputArray.shape[0], 48, 48, 3)` and Grayscale to `inputArray.reshape(inputArray.shape[0], 48, 48, 1)`.

To add the dropout layers, maxpooling and dense layers, we use the same `model.add()` function but pass in the correct layers which have to be first imported from Keras. See imports image below.



```
import keras
from keras.models import Sequential, Model, load_model
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D, Input
```

To run the model, we have to first compile with the `model.compile()` Keras function which takes an optimizer function, loss function and a choice of evaluation metrics. The optimizer has to be imported first but our compile function is as follows:



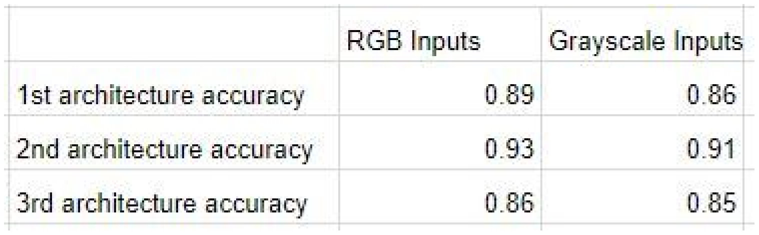
```
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

Next , to run our model we will use the `model.fit()` function and pass in numpy arrays of our train images and labels. The labels will first need to be converted to a format that Keras can understand by using a one-hot encode function `np_util` function also from Keras. In the fit function we will also specify batch sizes, number of epochs, validation data and callbacks. The validation data will be obtained by setting aside 20% of the train data. Callbacks is a useful Keras utility that lets us save the model and specify some metrics to keep track of as the model is training. We'll use this function to save the best training weights and an option to stop training if these weights don't improve after 10 epochs. To be able to analyze the model's performance, we will assign the fit function to a Keras history object that stores the training details. This will enable us to retrieve this information later for analysis. Our fit function is shown below:

```
    history = model.fit(X_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    validation_data=(X_valid,y_valid),
                    verbose=1,
                    callbacks=[checkpointer,early_stopping])
```

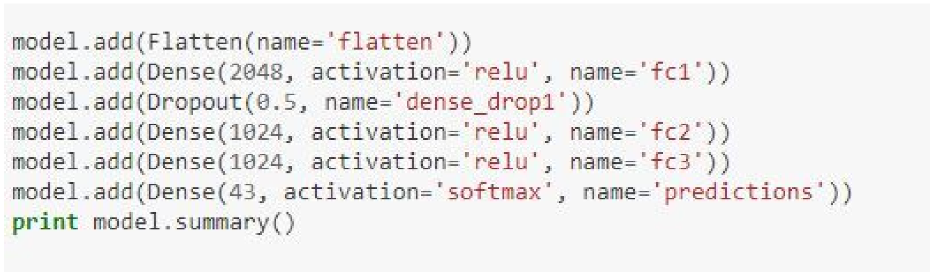
Finally, to get the performance from the model, we'll pass in the model name to the `model.load_weights()` where we can retrieve the accuracy. The Keras model.fit() function can also take a dictionary of the class weights which is useful for imbalanced classes like ours which we tried to implement but didn't seem to make any difference. The Keras documentation says the dictionary mapping is used for weighting the loss function during training which basically tells the model to pay more attention to underrepresented classes. We plan to revisit this property in the near future to fully understand its implementation.

The model training is done in a random trial and error fashion. Right from the beginning the Grayscale dataset seemed to underperformed the RGB everytime when ran both on the same network parameters.



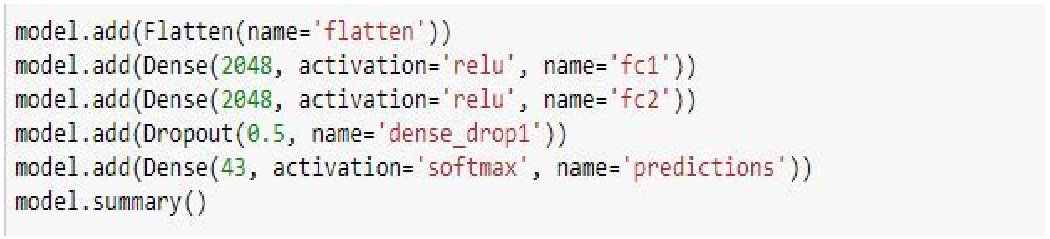
	RGB Inputs	Grayscale Inputs
1st architecture accuracy	0.89	0.86
2nd architecture accuracy	0.93	0.91
3rd architecture accuracy	0.86	0.85

We then concentrate on the 2nd architecture and try out more different parameters. We increase the number of epochs and implement an early stopping keras function to end training and return best weights if the validation error does not increase after a chosen number of epochs. Varying the batch size didn't seem to do much for the accuracy but it did slow down the training when using large batch sizes which made sense since we now have a much larger train dataset. A few more trial runs and we get a better accuracy by adding a third fully connected layer after the first fully connected layer as follows.



```
model.add(Flatten(name='flatten'))
model.add(Dense(2048, activation='relu', name='fc1'))
model.add(Dropout(0.5, name='dense_drop1'))
model.add(Dense(1024, activation='relu', name='fc2'))
model.add(Dense(1024, activation='relu', name='fc3'))
model.add(Dense(43, activation='softmax', name='predictions'))
print model.summary()
```

With the above configuration the accuracy on the RGB inputs goes up to 0.9682 and 0.9657 for the Grayscale set. This is a big improvement on our model and we might be heading in the right direction. With so much of the accuracy tied to the dense layers, we spend some more time experimenting with different variations of these layers. After more experimentation, we arrive at a better accuracy with 2 dense layers with 2048 neurons each and a 0.5 Dropout layer. Below is the final dense layers configurations.



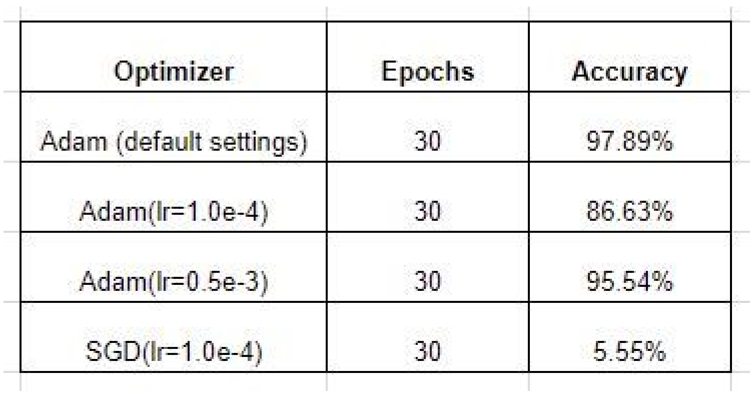
```
model.add(Flatten(name='flatten'))
model.add(Dense(2048, activation='relu', name='fc1'))
model.add(Dense(2048, activation='relu', name='fc2'))
model.add(Dropout(0.5, name='dense_drop1'))
model.add(Dense(43, activation='softmax', name='predictions'))
model.summary()
```

This configuration yields an accuracy of 97.63% for RGB inputs and 96.99% for Greyscale inputs. As we mentioned earlier, we had two versions of histogram equalized images Type 1 and Type 2. Type 2 from above gave the better results.

3.3 Refinement

So far we have experimented with model layers, batch size and epochs and our classification models seemed to be performing relatively well in theory but we had a goal of surpassing the human accuracy benchmark of 98.84%. We will experiment with the hyper parameters such as the optimizers and corresponding learning rates.

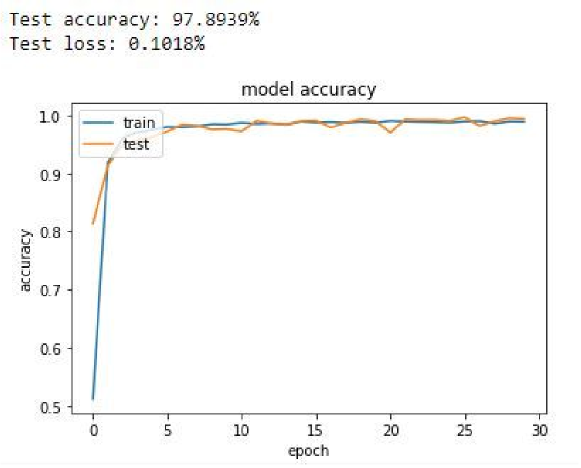
We first change the learning rate on the adam optimizer to 1.0e-4, then 0.5e-3. We want to see if the speed at which the learning rate converges has any effect of our accuracy result. Lastly we try a different optimizer and this time we choose the Stochastic Gradient Descent (SGD) with a learning curve of 1.0e-4. The SGD is a popular optimizer in the field of machine learning and this would be great chance to try it on our model. Below is a chart showing how the different optimizers performed on 30 epochs of training.



Optimizer	Epochs	Accuracy
Adam (default settings)	30	97.89%
Adam(lr=1.0e-4)	30	86.63%
Adam(lr=0.5e-3)	30	95.54%
SGD(lr=1.0e-4)	30	5.55%

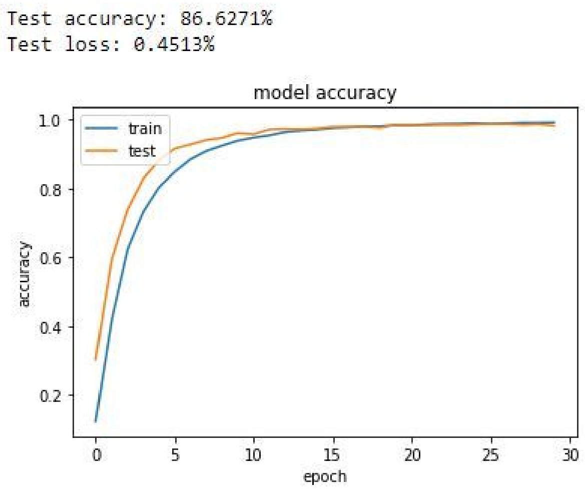
The Adam(lr=1.0e-4) does not converge as fast as the Adam(lr=0.5e-3) but when they do converge they do not allow much for the validation error correction. The SGD is just too slow for our model and it looks like we may have to run well above 100 epochs which is not a viable option given the computational time expense.

Adam(default settings):



```
Test accuracy: 97.8939%
Test loss: 0.1018%
```

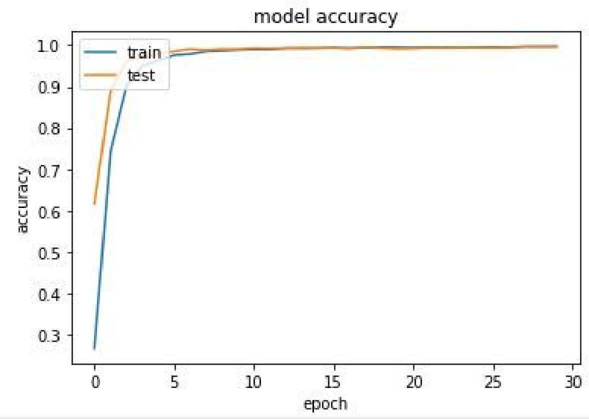
Adam(lr=1.0e-4):



```
Test accuracy: 86.6271%
Test loss: 0.4513%
```

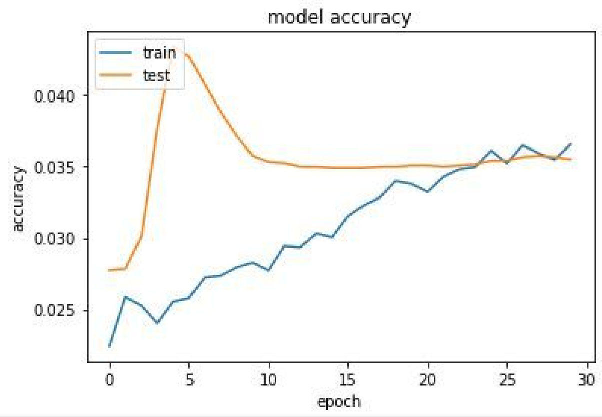
Adam(lr=0.5e-3) :

```
Test accuracy: 95.5424%
Test loss: 0.1589%
```

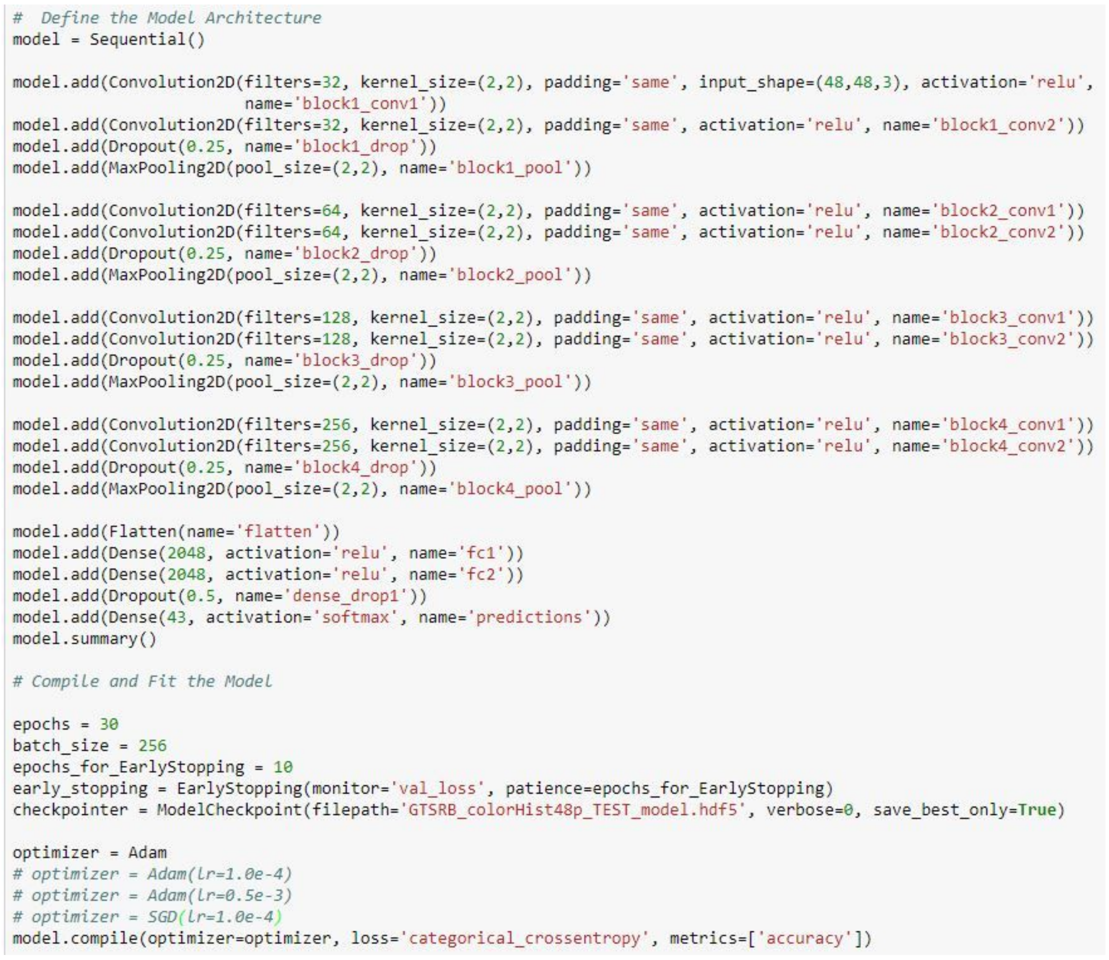


SGD(lr=1.0e-4):

```
Test accuracy: 5.5503%
Test loss: 3.7523%
```



The different optimizers that we tried failed to match the original Adam optimiser and as such we will stick with the original Adam. Shown below is the final model configuration:

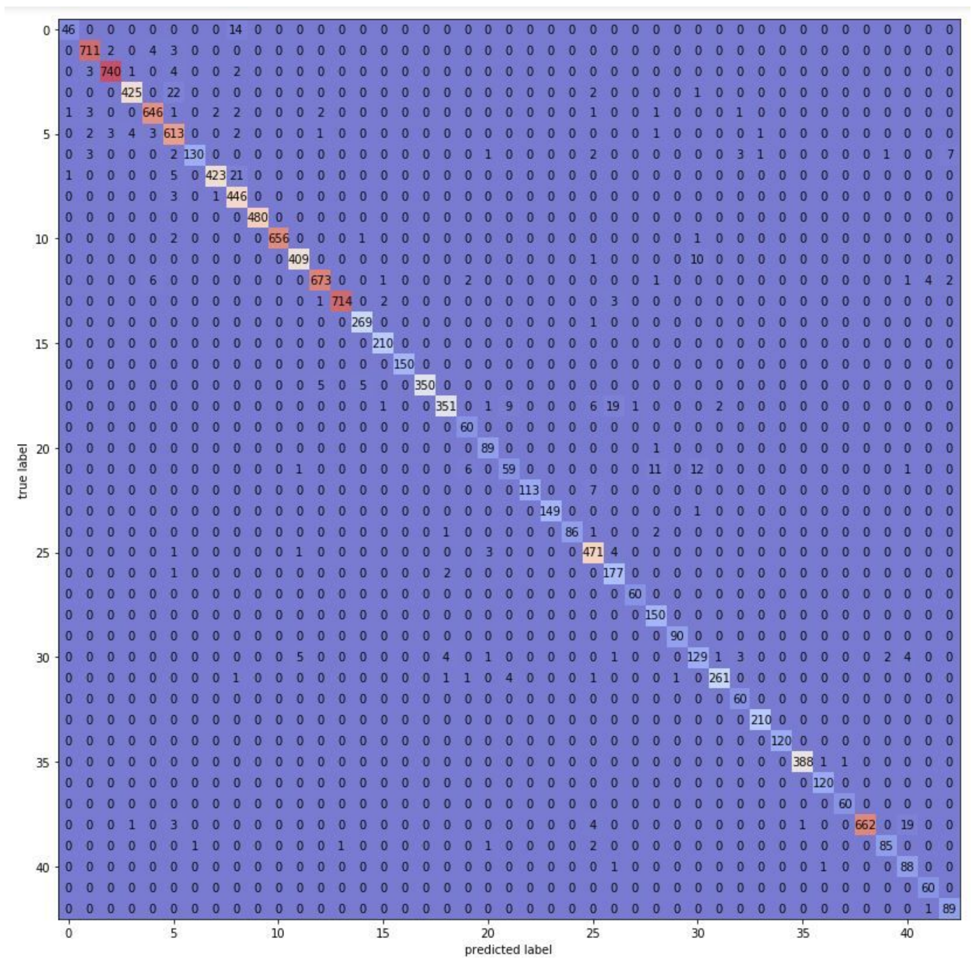


```
#  Define the Model Architecture
model = Sequential()

model.add(Convolution2D(filters=32, kernel_size=(2,2), padding='same', input_shape=(48,48,3), activation='relu',
                        name='block1_conv1'))
model.add(Convolution2D(filters=32, kernel_size=(2,2), padding='same', activation='relu', name='block1_conv2'))
model.add(Dropout(0.25, name='block1_drop'))
model.add(MaxPooling2D(pool_size=(2,2), name='block1_pool'))

model.add(Convolution2D(filters=64, kernel_size=(2,2), padding='same', activation='relu', name='block2_conv1'))
model.add(Convolution2D(filters=64, kernel_size=(2,2), padding='same', activation='relu', name='block2_conv2'))
model.add(Dropout(0.25, name='block2_drop'))
model.add(MaxPooling2D(pool_size=(2,2), name='block2_pool'))

model.add(Convolution2D(filters=128, kernel_size=(2,2), padding='same', activation='relu', name='block3_conv1'))
model.add(Convolution2D(filters=128, kernel_size=(2,2), padding='same', activation='relu', name='block3_conv2'))
model.add(Dropout(0.25, name='block3_drop'))
model.add(MaxPooling2D(pool_size=(2,2), name='block3_pool'))

model.add(Convolution2D(filters=256, kernel_size=(2,2), padding='same', activation='relu', name='block4_conv1'))
model.add(Convolution2D(filters=256, kernel_size=(2,2), padding='same', activation='relu', name='block4_conv2'))
model.add(Dropout(0.25, name='block4_drop'))
model.add(MaxPooling2D(pool_size=(2,2), name='block4_pool'))

model.add(Flatten(name='flatten'))
model.add(Dense(2048, activation='relu', name='fc1'))
model.add(Dense(2048, activation='relu', name='fc2'))
model.add(Dropout(0.5, name='dense_drop1'))
model.add(Dense(43, activation='softmax', name='predictions'))
model.summary()

# Compile and Fit the Model

epochs = 30
batch_size = 256
epochs_for_EarlyStopping = 10
early_stopping = EarlyStopping(monitor='val_loss', patience=epochs_for_EarlyStopping)
checkpointer = ModelCheckpoint(filepath='GTSRB_colorHist48p_TEST_model.hdf5', verbose=0, save_best_only=True)

optimizer = Adam
# optimizer = Adam(lr=1.0e-4)
# optimizer = Adam(lr=0.5e-3)
# optimizer = SGD(lr=1.0e-4)
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

4. Results

4.1 Model Evaluation and Validation

Our best model implementation has yielded some promising results and now should be a good time to look at this performance in depth. While an accuracy of **97.89%** is great at this point we should look at the Confusion Matrix (CM) of the model’s performance to understand the results better.



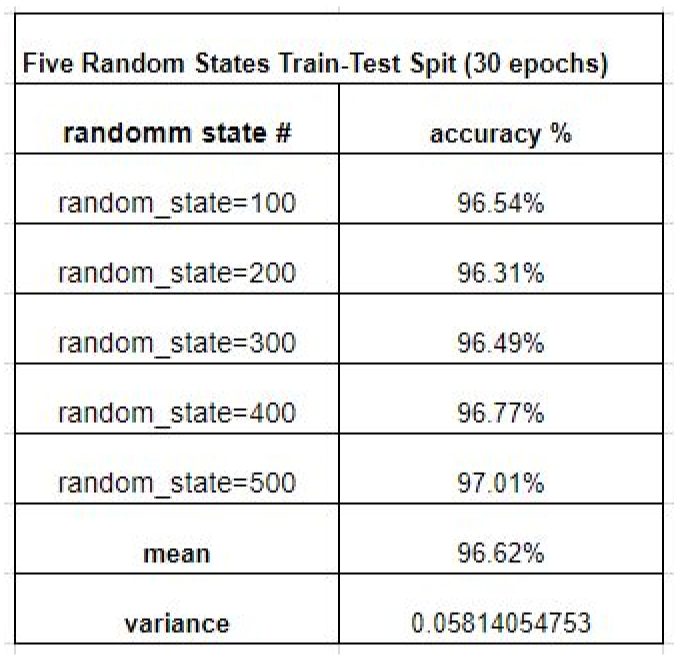
From the above CM we can make a few observation:-

- The model is mostly making the right predictions by matching the correct labels to the test images.
- There are a few misclassifications at the top left where the speed signs are clustered which means the model is sometimes having a difficulty interpreting the numbers on the signs but picking up the circle around it. We can observe that Class 5 was misclassified as Class 3 (80km/h as 60km/h)
- This can be tackled with the use of Branch Convolutional Networks to tackle this sort of hierarchy feature existence in data.
- The top right part of the CM also contains signs with circular images which account for the majority of the dataset even after class balancing.
- Class 18 was misclassified a 19 times as Class 26 (Caution sign for Traffic sign). Arguably similar when supplied with poor images.
- Class 40 was misclassified a 19 times as Class 38 (Roundabout mandatory for Keep right). They both have circular outer sign with an arrow pointing in a sideways direction.
- An interesting note to make is how well the under represented classes have performed reasonably well after generating more samples through data

augmentation. Class 39 only had 270 samples before more samples were generated and only has 2 misclassification. Classes 41 and 42 had the least amount of samples at 210 and have one misclassification between them.

- This is proof that more data is always better when training a machine learning algorithm.

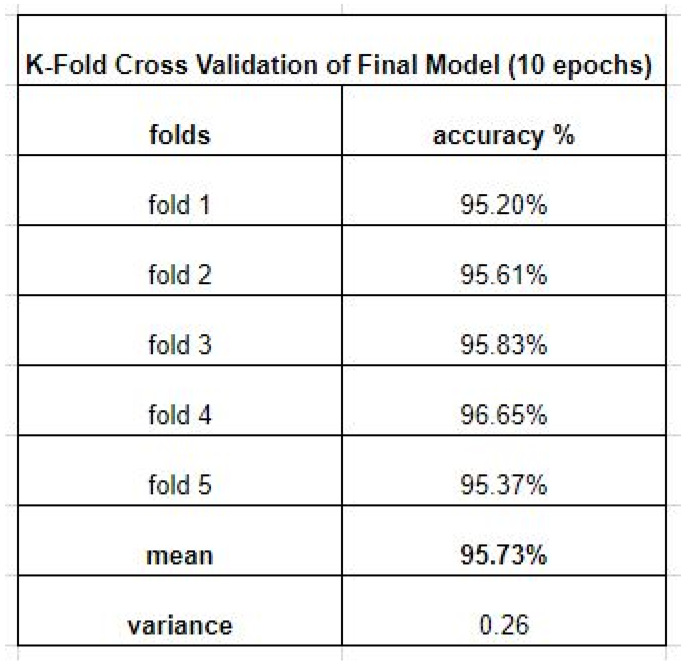
For further exploration into the models depth we ran the model 5 times with different random_states during the train-test-split stages (as per reviewer's recommendations) so we could analyze the mean and variance of the 5 different resulting accuracies. The results are shown in the table below.



Five Random States Train-Test Spit (30 epochs)	
randomm state #	**accuracy %**
random_state=100	96.54%
random_state=200	96.31%
random_state=300	96.49%
random_state=400	96.77%
random_state=500	97.01%
mean	96.62%
variance	0.05814054753

As we can see from the table above, the 5 different accuracies are in line with our expectations and confirm the models robustness because they all fall within a very reasonable variance window of 0.058 and also have a high precision to the mean. An interesting point to note is that our model's mean score is much lower than our original accuracy with the same exact model parameters. We have started the notebook in a different EC2 instance a few times and the weights initializations may be different. Either way, this is one more thing we would like to understand as we delve deeper into machine learning.

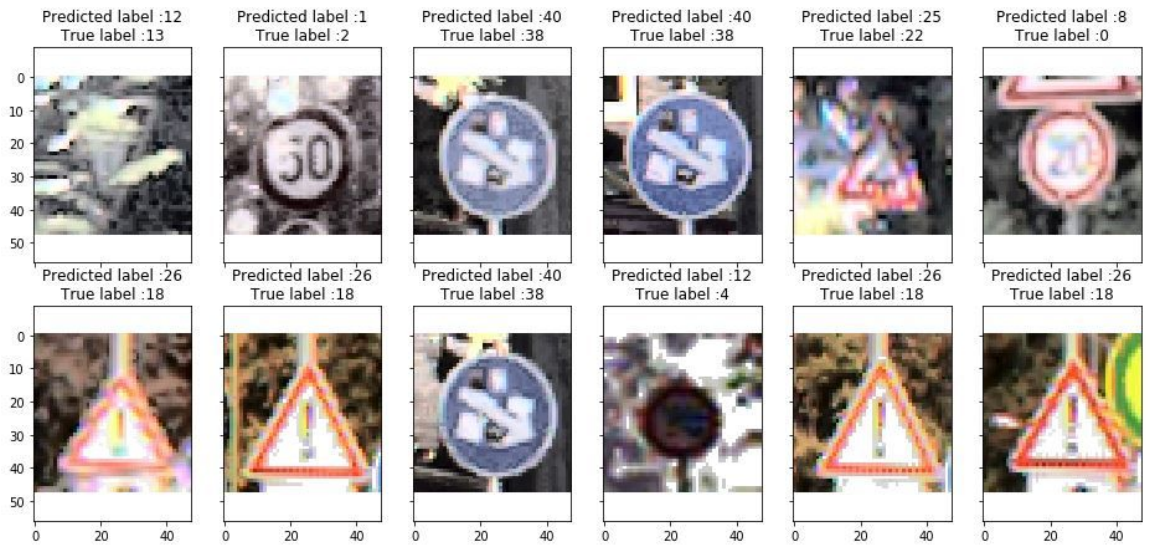
We experimented with K-Fold cross validation but the amount of data and the size of the network was very computationally expensive but we would still like to learn more on efficient ways of testing a CNNs model's robustness and apply it in the next projects. As far as how far we got with the second test, we did run the K-Fold validation function from `sklearn.model_selection.KFold()`. This function splits the training data into K consecutive folds and each fold is used once as validation while K-1 remaining folds forms the training set. We split our data into 5 folds and looked at the semi-final accuracies from all K folds as shown below. We could only afford to run the test for 10 epochs per fold compared to the 30 used for training but the results are again in line with our expectations of a robust model albeit non conclusive. A second Jupyter Notebook is provided with this test for additional evaluation but the 10 epoch tests are shown below and even though inconclusive, we hope to show that the right numbers were converging and the accuracies do have a high precision towards the mean. Additionally, the data was shuffled because the original data was just a concatenation of new augmented data to the original, so it was not entirely random.



K-Fold Cross Validation of Final Model (10 epochs)	
folds	**accuracy %**
fold 1	95.20%
fold 2	95.61%
fold 3	95.83%
fold 4	96.65%
fold 5	95.37%
mean	**95.73%**
variance	0.26

4.2 Justification

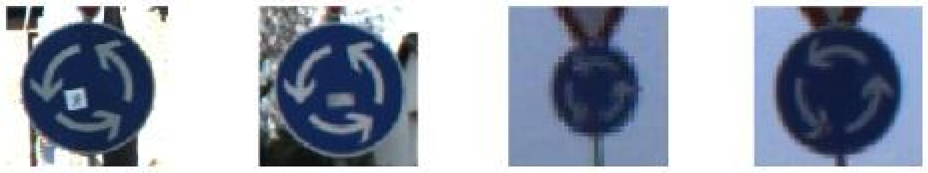
Compared to the benchmark model of 98.86% our model falls short by about a percentage point at 97.89%. To further understand where our model is going wrong, we visualize the worst probability predictions that our model made.



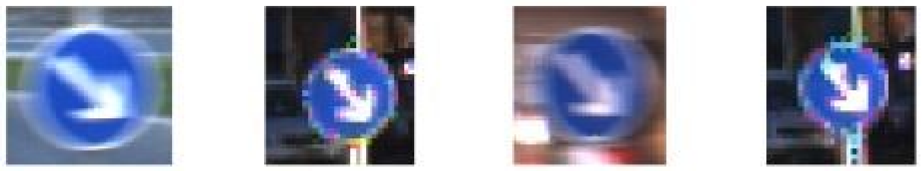
From the image above;

- Index(0,0) makes sense as it is hard even for a human to make sense of that image.
- Index(0,1) could easily be misclassified as a 3 because the front part of the 5 is missing.
- Index(0,2), (0,3) and (1,2) are predicted as “Roundabout mandatory” but their true label is “Keep right”. Below we show extra samples from both classes to get a better picture.

Predicted Label =

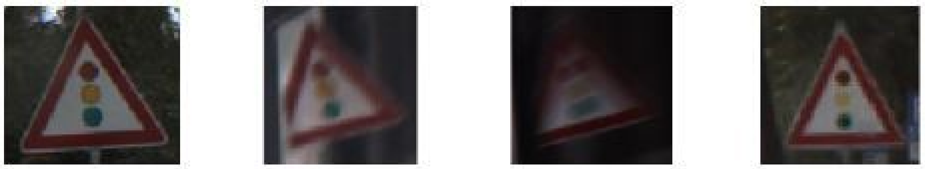


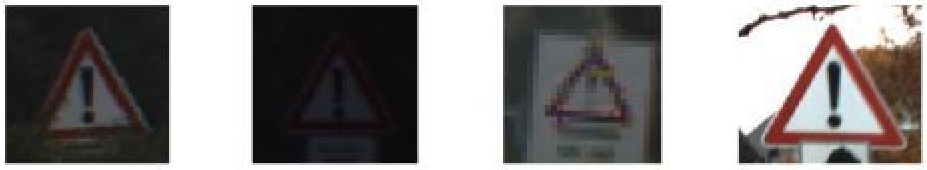
True Label =



This is obviously a poor job at predicting the class. It looks like the model sees a diagonal arrow and puts it in class 40 where there is a curved arrow.

- Index (0,4) and (1,3) are very poor images for a human eye to classify either. The model’s prediction is wrong but the bounding shape of the sign is the same in the predicted label.
- Index (1,0),(1,1),(1,4) and (1,5) all belong to the same class 26 and the same wrong prediction of class 18 is made on them.

Predicted Label = 

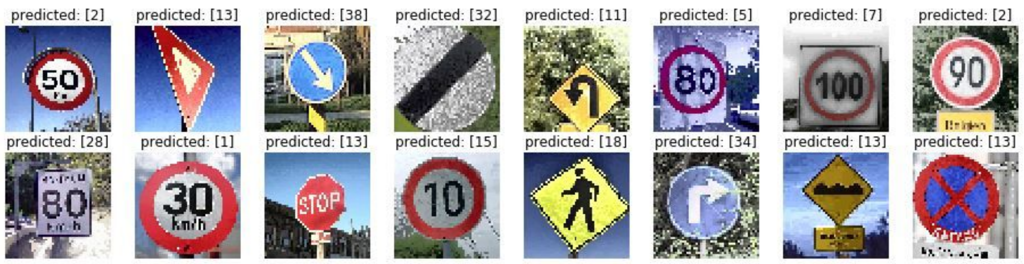
True Label = 

Here we see that the model is learning something but not enough to make the correct prediction. The model sees the green lower dot as the same dot in the true label. So we can conclude that even though we have achieved a high accuracy, the model is not acceptable for a real world application.

5. Conclusion

5.1 Free-Form Visualization

We are not content with our model but we would like to see how it performs on real world data. For this we will make some predictions on some samples collected from the internet. Our samples include images from both the learned class labels but also some images that are don't belong to any of the 43 classes. Furthermore, the samples have been preprocessed in the same way the training data was processed. Let us visualize the model's predictions.



In total we used 16 samples which the model predicted 7 correctly. That sounds like a really bad classification so let us understand the results.

- Indexes (0,0),(0,1),(0,2),(0,3),(0,5),(0,6), and (1,1) were the correct predictions and they are very clear to understand.
- Indexes (0,7), (1,0),(1,3),(1,4) and (1,7) are not in the trained labels so the model had to pick something close which it does.

- That leaves us with (0,4),(1,2),(1,5) and (1,6) which the model should have classified correctly.
- It is interesting to note that (1,5) is actually classified as a class that is the very exact opposite.

5.2 Reflection

Building our CNN has been quite the experience in terms of of really understanding what it takes to go from idea inception to getting a machine learning algorithm that works. What may look like a very straight forward concept to test does not always translate to an easy one to execute. A few elements of this process come to mind.

The first one is data processing. This stage is very time consuming to get data with the most information landscape for the machine to learn from but without passing on any biases. Take for example the image augmentation we did to generate more samples for the imbalanced classes. Certain questions come to mind. How much augmentation should we do and how much is too much. A right arrow rotated too much becomes a left turn sign which is one of the other class labels. However in the case of a stop sign, it does not matter how much rotation and more rotation actually may be better for the model to generalize better on that label. The perfect balance here is very elusive.

The second aspect of building this model that comes to mind is the model parameter and hyper-parameter tuning. Again the question remains as to how much is too much or too little. For example you could achieve some good metrics with a first of set of settings but they may also create a second window of improvement in the second set of settings. Take the model architecture for example, does adding another layer affect the other layers, and should we now change those layers, does that affect the optimal number of batches or epoch, what about the optimizer settings? There are times we can use a grid search to find these optimal parameters but it’s not always feasible for a model like hours from a computational expense perspective.

It is however a great experience to understand some of these challenges that come up when developing a successful model. Experience will help us think about how we will go about our next model. In conclusion, as with most research, the idea is to keep exploring, testing and improving even without enough data because in the real world the right kind of data is not always available.

5.3 Improvement

To improve on the model, a different approach can be taken when performing the classification. For example the images could be further classified by their common features like triangles and circles and then passed on to a different model that specializes in that task. There is a good paper relevant to this topic on pattern recognition - Branch Convolutional Neural Network for Hierarchical Classification[12].

There are also many ways of handling imbalanced classes and they are worth exploring to see if the model would perform better. A few that come to mind are oversampling of minority classes and undersampling of majority classes.

It may also make particular sense to consider every label individually when performing augmentations. Performing this process may help create better representation of individual classes in the dataset.

Transferred Learning can also be explored for a better classification performance. Models weights are available from these pre trained models like VGG16 and Xception[13] to name a few. These models have been trained on a much superior hardware, input data quality and a huge amount of time that we can not afford to train our own model.

References:

[1] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1359523/
[2] https://cs.nyu.edu/~fergus/papers/zeilerECCV2014.pdf
[3] http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset#Downloads
[4] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In Proceedings of the IEEE International Joint Conference on Neural Networks, pages 1453–1460. 2011.
[5] http://benchmark.ini.rub.de/?section=gtsrb&subsection=news
[6] Human Performance, INI-RTCV , Man vs. computer: Benchmarking machine learning algorithms to traffic sign recognition, Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition, J. Stallkamp, M. Schlipsing, J. Salmen, C. Igel, August 2012, Neural Networks (32), pp. 323-332
[7] arXiv:1511.02992 [cs.CV]
[8] CNN Image Classification Rankings:
http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#43494641522d3130

[9] arXiv:1409.1556 [cs.CV]
[10] http://www.image-net.org/challenges/LSVRC/2014/results
[11] arXiv:1412.6980 [cs.LG]
[12] arXiv:1709.09890 [cs.CV]
[13] arXiv:1610.02357 [cs.CV]
[14] https://docs.opencv.org/2.4/modules/imgproc/doc/
geometric_transformations.html?highlight=warpaffine

Image Credits:

- https://image.slidesharecdn.com/nncollovcapaldo2013-131220052427-phpapp01/95/machine-learning-introduction-to-neural-networks-33-638.jpg?cb=1393073301
- http://book.paddlepaddle.org/03.image_classification/
- By Aphex34 - Own work, CC BY-SA 4.0, - CNN Architecture
- http://book.paddlepaddle.org/03.image_classification/ - VGG16 Architecture