Traffic Sign Recognition

May 2020

TEAM MEMBERS
Aastha Saraf
(asaraf4@uic.edu)
675532334
Akshay Narula
(anarul3@uic.edu)
655715365

Introduction & Background

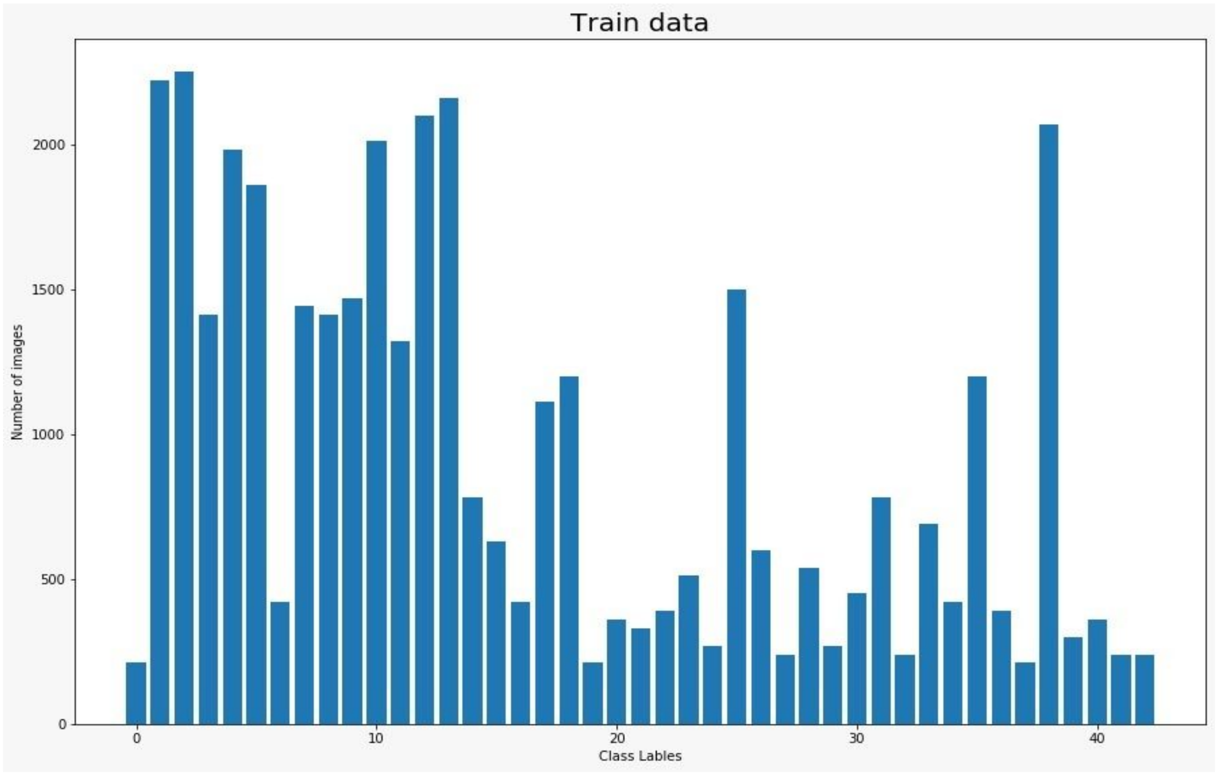
There are several different types of traffic signs like speed limits, no entry, turn left or right, children crossing, no passing of heavy vehicles, etc. Traffic signs recognition is the process of identifying which class a traffic sign belongs to using classification.
In this Python project, we will build a deep neural network model that can classify traffic signs present in the image into different categories. With this model, we are able to read and understand traffic signs which are a very important task for all autonomous vehicles.

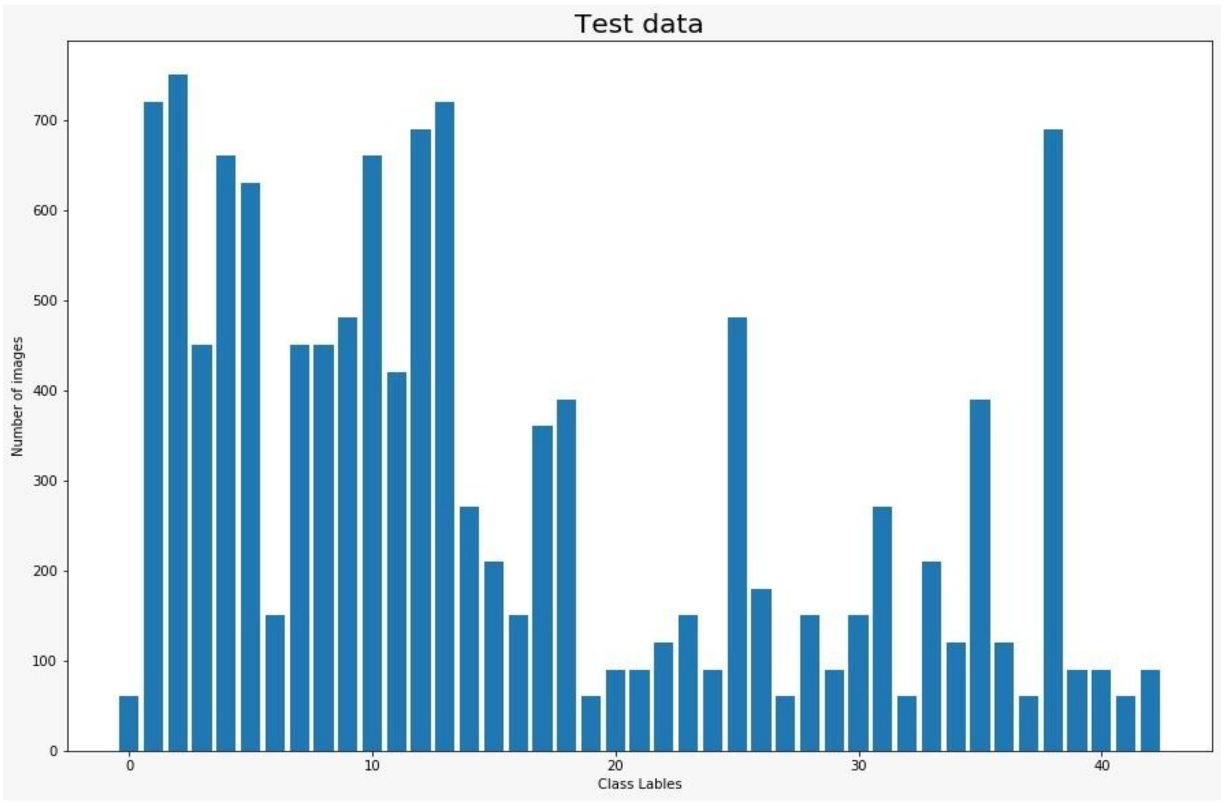
Dataset

The dataset contains more than 50,000 images of different traffic signs. It is further classified into 42 different classes. The dataset is quite varying, some of the classes have many images while some classes have few images. The size of the dataset is around 300 MB. The dataset has a train folder which contains images inside each class and a test folder which we will use for testing our model. The dataset consists of a .csv file, from which we fetched the labels using pandas, from which we extract the Class labels and path of the image, and then we fetch that image using Pillow (Python Imaging library).

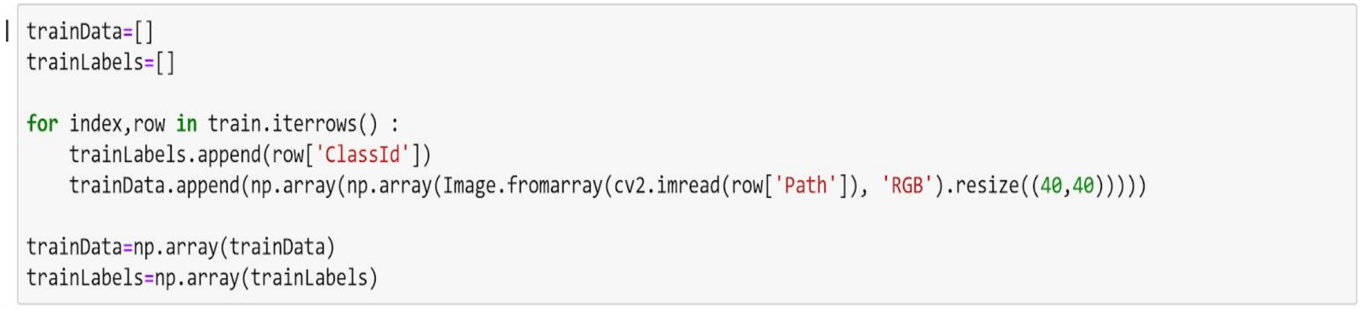
Dataset Analysis

For analysing the training and testing data, we first checked the size of each class, that is the number of images each class has. In order to get an insight of how the network will be trained to identify the classes and till what extent. As we can see that there are classes which are trained better than the others, as they have a plethora of training data. Below are the graphs from the code that illustrates the same.


Train data
Number of images
Class Lables
0
500
1000
1500
2000
0
10
20
30
40


Test data
Number of images
Class Lables
0
100
200
300
400
500
600
700
0
10
20
30
40

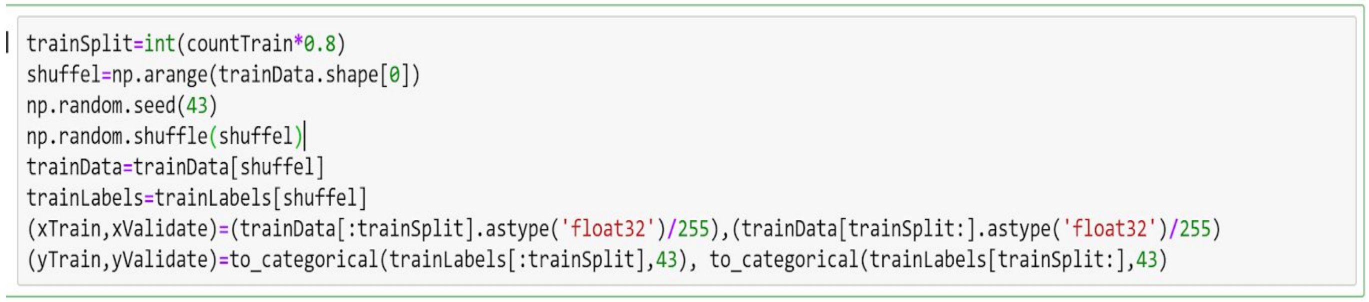
After this, we picked up RGB images in an array and resized each of the images to (40,40) to get a common scale.
Then, we divided the training data into 2 parts, that is 80% training data and 20% validation data, which helped us in getting the accuracy scores on the training set based on each epoch, CNN in keras facilitates the validation of the network by passing both validation and training data at once.
As the data was very sequential, one class after the other, we shuffled the data so that each category got similar priority. Also if we divided the sequential data into training and validation data without the shuffle, the few categories which were a part of the validation set, wouldn't have been trained and so there was a chance that we wouldn't have been able to train the model accurately.



```
trainData=[]
trainLabels=[]

for index,row in train.iterrows() :
    trainLabels.append(row['ClassId'])
    trainData.append(np.array(np.array(Image.fromarray(cv2.imread(row['Path']), 'RGB').resize((40,40)))))

trainData=np.array(trainData)
trainLabels=np.array(trainLabels)
```



```
trainSplit=int(countTrain*0.8)
shuffel=np.arange(trainData.shape[0])
np.random.seed(43)
np.random.shuffle(shuffel)
trainData=trainData[shuffel]
trainLabels=trainLabels[shuffel]
(xTrain,xValidate)=(trainData[:trainSplit].astype('float32')/255),(trainData[trainSplit:].astype('float32')/255)
(yTrain,yValidate)=to_categorical(trainLabels[:trainSplit],43), to_categorical(trainLabels[trainSplit:],43)
```

Literature Review

For this project we took inspiration from the research paper, An Introduction to Convolutional Neural Networks, by Keiron O'Shea and Ryan Nash. It talks about how the rise of the Artificial Neural Network (ANN) has twisted the field of machine learning in recent times, also how the biologically inspired computational models have far exceeded the performance of previous forms of artificial intelligence in common machine learning tasks. It also states that one of the most impressive forms of ANN architecture is that of the Convolutional Neural Network (CNN), as CNNs are primarily used to solve difficult image-driven pattern recognition tasks and with their precise architecture offers a simplified method of getting started with ANNs. The paper provides a brief introduction to CNNs, and discusses about the recently published papers and newly formed techniques in developing these brilliantly fantastic image recognition models.

How are CNNs similar to ANNs?
CNNs are feedforward networks in which information flow takes place in one direction only, from their inputs to their outputs. Just as artificial neural networks (ANN) are biologically inspired, so are CNNs. The visual cortex in the brain, which consists of alternating layers of simple and complex cells, motivates their architecture. CNN architectures come in several variations; however, in general, they consist of convolutional and pooling (or subsampling) layers, which are grouped into modules. Either one or more fully connected layers, as in a standard feedforward neural network, follow these modules. Modules are often stacked on top of each other to form a deep model.

How are CNNs different from ANNs?
The only notable difference between CNNs and traditional ANNs is that CNNs are primarily used in the field of pattern recognition within images. This allows us to encode image-specific features into the architecture, making the network. more suited for image-focused tasks - whilst further reducing the parameters required to set up the model. One of the largest limitations of traditional forms of ANN is that they tend to struggle with the computational complexity required to compute image data.

Therefore we learned that Convolutional Neural Networks differ from other forms of ANNs, in that instead of focusing on the whole problem domain, knowledge about the specific type of input is exploited. Hence, this in turn allows us to design a much simpler network architecture.
This paper has outlined the basic concepts of Convolutional Neural Networks, explaining the layers required to build it and detailing how best to structure the network in most image analysis tasks.
Research in the field of image analysis using neural networks has somewhat slowed in recent times. This is partly due to the incorrect belief surrounding the level of complexity

and knowledge required to begin modelling these superbly powerful machine learning algorithms. The authors hope that this paper has in some way reduced this confusion, and made the field more accessible to beginners.

After reading this paper, we decided to build an Image Recognition model by using CNN. As we were looking for relevant datasets, which could give us an idea of what exactly we wanted to build. Soon, we found this dataset which was intriguing, because it had a lot of scope in building a future AI model. These models could be used for enhancing the driving safety of intelligent vehicles in the actual driving environments and effectively meeting the real-time target requirements of self driven cars. Furthermore, this can offer a strong technical guarantee for the steady development of intelligent vehicle driving assistance. In the future, the traffic sign recognition algorithm can be further optimized and improved to exploit the overall performance of the algorithm.

Machine Learning Applications

Convolutional Neural Network is a deep learning algorithm which is used for recognizing images. This algorithm clusters images by similarity and performs object recognition within the scenes. CNN uses unique features of images to identify the objects on the image. This process is very similar to what our brain does to identify objects.

Traditional neural networks are not ideal for image processing that’s why we are using CNN. Although CNN is not too different from ANN because CNN algorithm uses some layers to gather information and determine some features from the image.
The different layers that we used in the CNN are:

- Convolution
- Pooling
- Flattening
- Full Connection (Dense)

Convolutional Layer

This is the main process for CNN. In this operation there is a feature detector or filter that detects edges or specific shapes. Filter is placed top left of image and multiplied with value on same indices. After that all results are summed and written to the output matrix. Then the filter slips to the right to do this whole process again and again. Usually filters slip one by one and can be changed from model- to- model and this slipping process is called ‘stride’. Bigger stride means smaller output. Sometimes stride value is increased to decrease output size and time.

After convolutional operations we use an activation function to break up linearity. We want to increase non-linearity otherwise the algorithm won't understand image and act like it is a linear function.

Pooling

This layer is used for reducing parameters and computing processes. By using this layer features that don’t change to scale or orientation changes are detected and hence overfitting is prevented. There are some pooling process like average pooling, max pooling etc.

Flattening

This process takes a matrix that came from convolutional and pooling processes and turns it into one dimensional (1D) array. This is important because Rectangular or cubic shapes can’t be direct inputs. We flatten the output of the convolutional layers to create a single long feature vector. And it is connected to the final classification model, which is called a fully-connected layer.

Full Connection

This layer takes data from one dimension array we saw above and starts the learning process. The fully-connected layer contains neurons of which are directly connected to the neurons in the two adjacent layers, without being connected to any layers within them.

Convolutional Neural Network Algorithm with Keras

Dropout

Dropout is a regularization technique that helps in reducing overfitting. A fully connected layer occupies most of the parameters, and hence, neurons develop co-dependency amongst each other during training which curbs the individual power of each neuron leading to over-fitting of training data. It is called “dropout” because it drops the visible or hidden units in the neural network.
More technically, At each training stage, individual nodes are either dropped out of the net with probability *1-p* or kept with probability *p*, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed.

Optimizer

We’ll use ‘Adam Optimizer’. It is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on the training data. SGD pursues a single learning rate for all weights updates and learning rate(alpha) and doesn’t change during the training process. However in adam optimizer we can say adam optimizer updates learning rate dynamically.

Epochs and Batch size

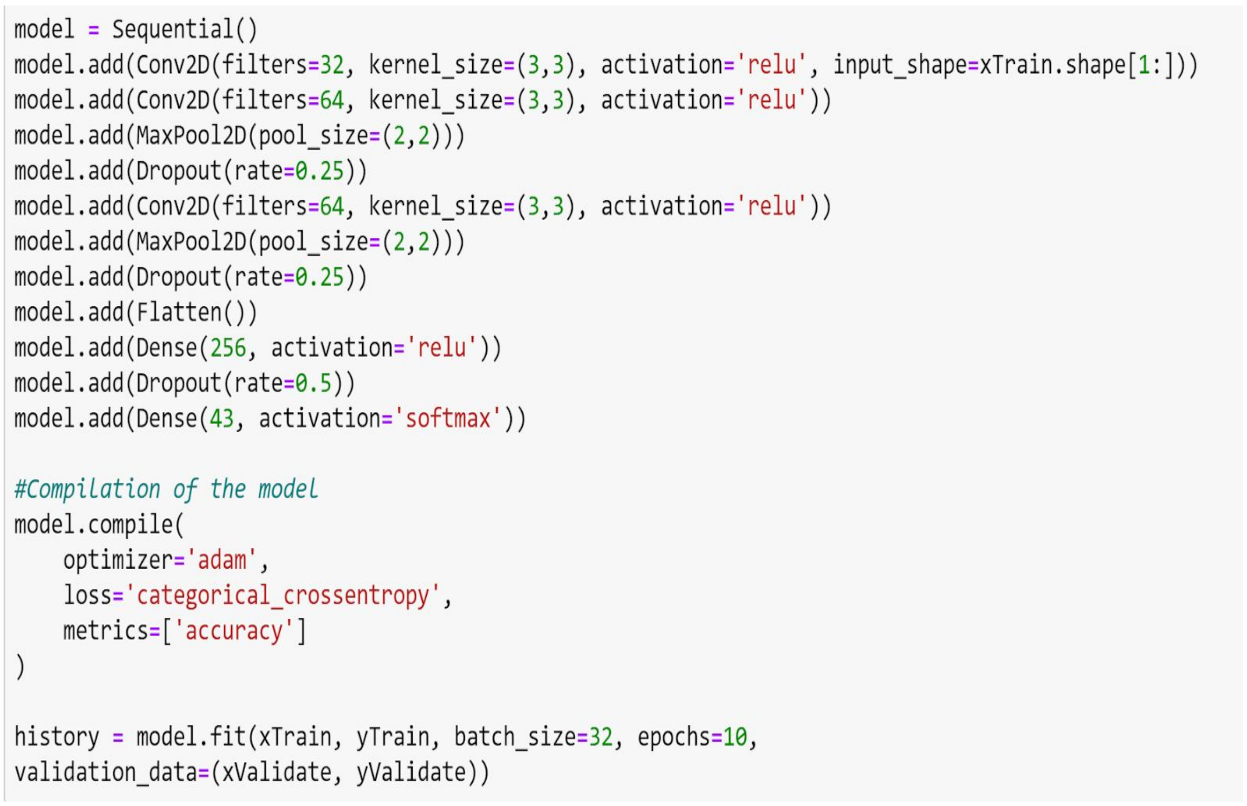
Epoch is the number of times the algorithm sees the entire data set. Since in many cases when the dataset is large, one epoch is too large to run on the computer at once. So, we divide it into smaller parts and the number of these parts is called batch.

Description of the coding process

We used the Sequential model from keras because of the ease of implementation. This model allows adding new layers based on the developer’s convenience.

First, we added the input layer of Conv2D followed by the internal layers. The internal layers are a combination of Conv2D, MaxPool2D and Dropout with a rate of 0.25.
After the convolutional and the pooling, we flatten the data to prepare it for the fully connected Dense layer.
Finally, we add the output layer which has the final classification output.

Now that the network is ready, we compile the model with the adam optimizer for the accuracy metric and fit it with a batch size of 32 and the validation data.
We train the network for 10 epochs. We can see the accuracy improving below. The selection of this model will be explained in the next section.



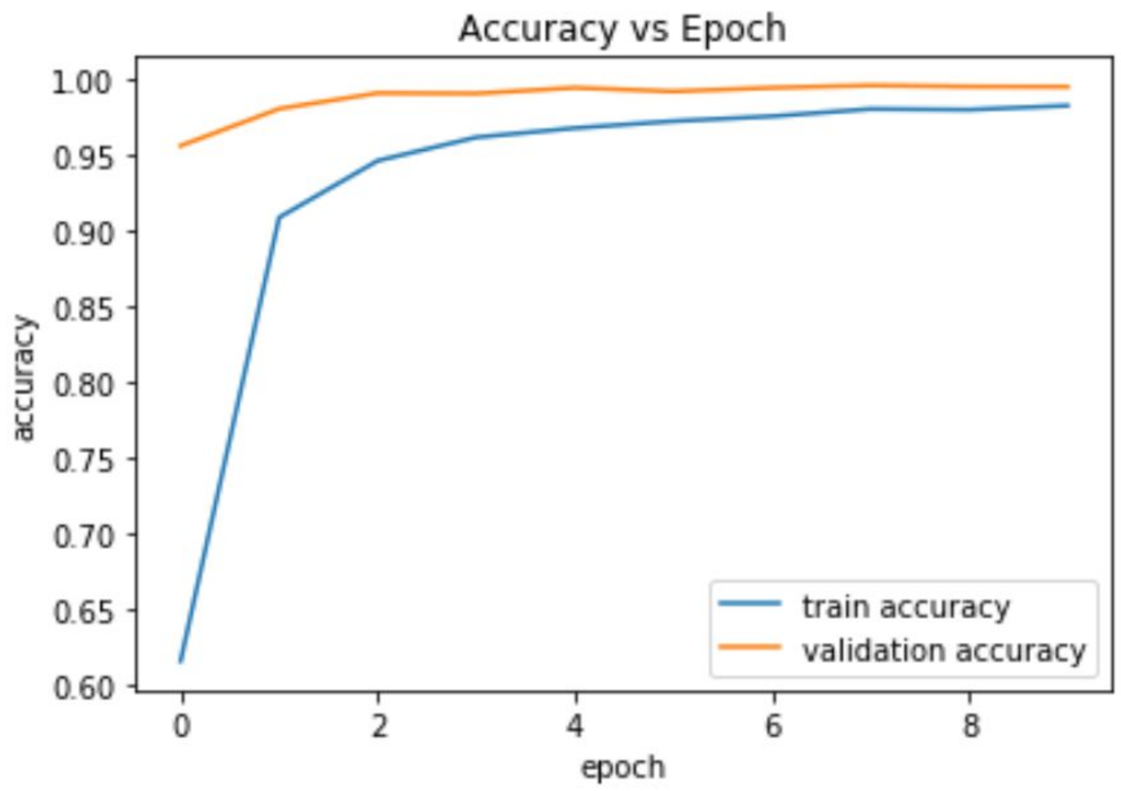
```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu', input_shape=xTrain.shape[1:]))
model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu'))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(rate=0.25))
model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu'))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(rate=0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(43, activation='softmax'))

#Compilation of the model
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

history = model.fit(xTrain, yTrain, batch_size=32, epochs=10,
validation_data=(xValidate, yValidate))
```

```
Train on 31367 samples, validate on 7842 samples
Epoch 1/10
31367/31367 [==============================] - 217s 7ms/step - loss: 1.3137
- accuracy: 0.6150 - val_loss: 0.2197 - val_accuracy: 0.9556
Epoch 2/10
31367/31367 [==============================] - 218s 7ms/step - loss: 0.2938
- accuracy: 0.9082 - val_loss: 0.1035 - val_accuracy: 0.9801
Epoch 3/10
31367/31367 [==============================] - 220s 7ms/step - loss: 0.1736
- accuracy: 0.9458 - val_loss: 0.0408 - val_accuracy: 0.9904
Epoch 4/10
31367/31367 [==============================] - 217s 7ms/step - loss: 0.1228
- accuracy: 0.9612 - val_loss: 0.0444 - val_accuracy: 0.9902
Epoch 5/10
31367/31367 [==============================] - 216s 7ms/step - loss: 0.1014
- accuracy: 0.9673 - val_loss: 0.0266 - val_accuracy: 0.9940
Epoch 6/10
31367/31367 [==============================] - 221s 7ms/step - loss: 0.0871
- accuracy: 0.9719 - val_loss: 0.0319 - val_accuracy: 0.9916
Epoch 7/10
31367/31367 [==============================] - 218s 7ms/step - loss: 0.0767
- accuracy: 0.9751 - val_loss: 0.0242 - val_accuracy: 0.9940
Epoch 8/10
31367/31367 [==============================] - 215s 7ms/step - loss: 0.0654
- accuracy: 0.9799 - val_loss: 0.0192 - val_accuracy: 0.9955
Epoch 9/10
31367/31367 [==============================] - 212s 7ms/step - loss: 0.0650
- accuracy: 0.9795 - val_loss: 0.0190 - val_accuracy: 0.9948
Epoch 10/10
31367/31367 [==============================] - 213s 7ms/step - loss: 0.0545
- accuracy: 0.9821 - val_loss: 0.0197 - val_accuracy: 0.9946
```

As you can see below, the model's accuracy grows with epoch.



Now, using the testing data, we computed the test accuracy score for the best model which turned out to be more than 95%.

```
predictedLabels = model.predict_classes(testData)

print('The accuracy score for cnn with softmax activation for dense layer:')
print(accuracy_score(testLabels, predictedLabels))
```

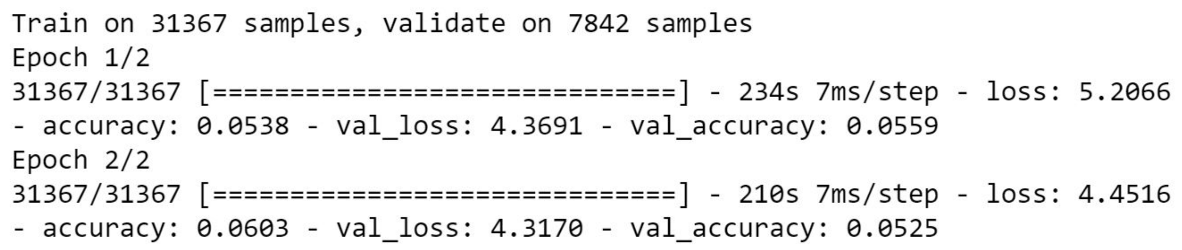
```
The accuracy score for cnn with softmax activation for dense layer:
0.9644497228820269
```

Analysis and Conclusion

As we used CNN for this Image Recognition project, we changed a lot of hyperparameters including the kernel size, epochs filters and optimizers. However, there wasn't much improvement in the initial accuracy of 20% at max.

ReLU:
Applies the rectified linear unit activation function. With default values, this returns the standard ReLU activation: max(x, 0), the element-wise maximum of 0 and the input tensor.

Modifying default parameters allows you to use non-zero thresholds, change the max value of the activation, and to use a non-zero multiple of the input for values below the threshold.



```
Train on 31367 samples, validate on 7842 samples
Epoch 1/2
31367/31367 [==============================] - 234s 7ms/step - loss: 5.2066
- accuracy: 0.0538 - val_loss: 4.3691 - val_accuracy: 0.0559
Epoch 2/2
31367/31367 [==============================] - 210s 7ms/step - loss: 4.4516
- accuracy: 0.0603 - val_loss: 4.3170 - val_accuracy: 0.0525
```

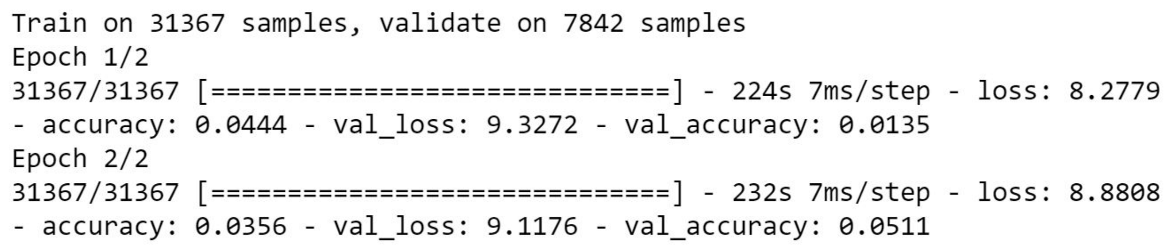
Arguments

- x: Input tensor or variable.
- alpha: A float that governs the slope for values lower than the threshold.
- max_value: A float that sets the saturation threshold (the largest value the function will return).
- threshold: A float giving the threshold value of the activation function below which values will be damped or set to zero.

Returns

A Tensor representing the input tensor, transformed by the relu activation function. Tensor will be of the same shape and dtype of input x.

LeakyReLU:
LeakyRelu is a variant of ReLU. Leaky ReLUs are one attempt to fix the “dying ReLU” problem by having a small negative slope (of 0.01, or so). Leaky ReLU allows a small, non-zero, constant gradient α (Normally, α=0.01).



```
Train on 31367 samples, validate on 7842 samples
Epoch 1/2
31367/31367 [==============================] - 224s 7ms/step - loss: 8.2779
- accuracy: 0.0444 - val_loss: 9.3272 - val_accuracy: 0.0135
Epoch 2/2
31367/31367 [==============================] - 232s 7ms/step - loss: 8.8808
- accuracy: 0.0356 - val_loss: 9.1176 - val_accuracy: 0.0511
```

Our first choice of activation function was ReLU because of its linearity and speed. Upon facing bad results we moved to leaky Relu as we suspected the dying ReLU problem. However, it did not improve anything. Then we decided to use the good old sigmoid function, which gave good results.

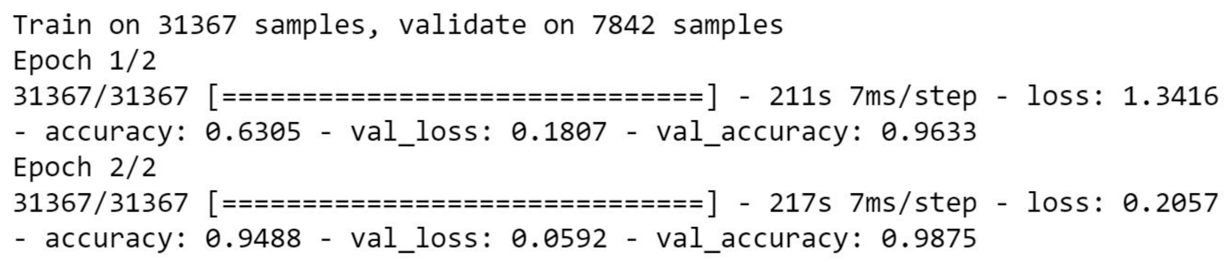
Sigmoid:
Sigmoid activation function, sigmoid(x) = 1 / (1 + exp(-x)). Applies the sigmoid activation function. For small values (<-5), sigmoid returns a value close to zero, and for large values (>5) the result of the function gets close to 1. Sigmoid is equivalent to a 2-element Softmax, where the second element is assumed to be zero. The sigmoid function always returns a value between 0 and 1.

Arguments

- x: Input tensor.

Returns

Tensor with the sigmoid activation: 1 / (1 + exp(-x)).



```
Train on 31367 samples, validate on 7842 samples
Epoch 1/2
31367/31367 [==============================] - 211s 7ms/step - loss: 1.3416
- accuracy: 0.6305 - val_loss: 0.1807 - val_accuracy: 0.9633
Epoch 2/2
31367/31367 [==============================] - 217s 7ms/step - loss: 0.2057
- accuracy: 0.9488 - val_loss: 0.0592 - val_accuracy: 0.9875
```

Further, we decided to use the popular activation functions in the dense layer to improve the accuracy. Although tanh, softsign, selu and elu did not give a good result, softmax and softplus gave results greater than 90% accuracy in just 2 epochs.

Functions that gave good accuracies:

Softmax:
Softmax converts a real vector to a vector of categorical probabilities. The elements of the output vector are in range (0, 1) and sum to 1. Each vector is handled independently. The axis argument sets which axis of the input the function is applied along. Softmax is often used as the activation for the last layer of a classification network because the result could be interpreted as a probability distribution. The softmax of each vector x is computed as exp(x) / tf.reduce_sum(exp(x)). The input values in are the log-odds of the resulting probability.

Arguments

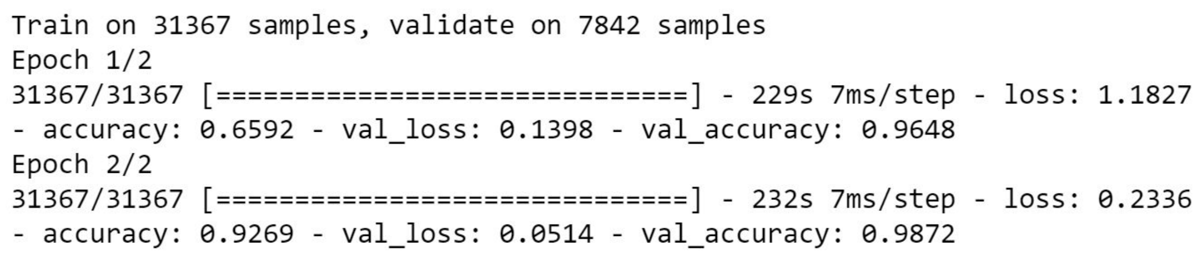
- x : Input tensor.
- axis: Integer, axis along which the softmax normalization is applied.

Returns

Tensor, output of softmax transformation (all values are non-negative and sum to 1).

Raises

- ValueError: In case dim(x) == 1.



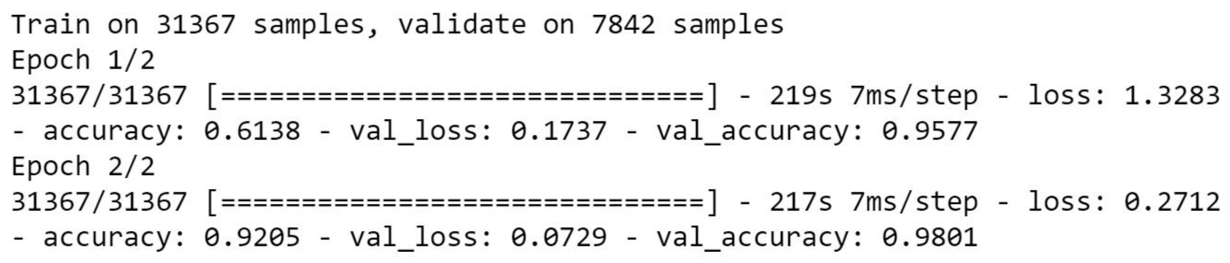
```
Train on 31367 samples, validate on 7842 samples
Epoch 1/2
31367/31367 [==============================] - 229s 7ms/step - loss: 1.1827
- accuracy: 0.6592 - val_loss: 0.1398 - val_accuracy: 0.9648
Epoch 2/2
31367/31367 [==============================] - 232s 7ms/step - loss: 0.2336
- accuracy: 0.9269 - val_loss: 0.0514 - val_accuracy: 0.9872
```

Softplus:
Softplus activation function, softplus(x) = log(exp(x) + 1).
Arguments

- x: Input tensor.

Returns

The softplus activation: log(exp(x) + 1).



```
Train on 31367 samples, validate on 7842 samples
Epoch 1/2
31367/31367 [==============================] - 219s 7ms/step - loss: 1.3283
- accuracy: 0.6138 - val_loss: 0.1737 - val_accuracy: 0.9577
Epoch 2/2
31367/31367 [==============================] - 217s 7ms/step - loss: 0.2712
- accuracy: 0.9205 - val_loss: 0.0729 - val_accuracy: 0.9801
```

Functions that gave bad accuracies:

Softsign:
Softsign activation function, softsign(x) = x / (abs(x) + 1).
Arguments

- x: Input tensor.

Returns

The softsign activation: x / (abs(x) + 1).

```
Train on 31367 samples, validate on 7842 samples
Epoch 1/2
31367/31367 [==============================] - 217s 7ms/step - loss: 8.1237
- accuracy: 0.0117 - val_loss: 9.1833 - val_accuracy: 0.0055
Epoch 2/2
31367/31367 [==============================] - 204s 7ms/step - loss: 8.1126
- accuracy: 0.0099 - val_loss: 9.1834 - val_accuracy: 0.0079
```

Tanh:
Hyperbolic tangent activation function.
Arguments

- x: Input tensor.

Returns

Tensor of same shape and dtype of input x, with tanh activation: tanh(x) = sinh(x)/cosh(x) = ((exp(x) - exp(-x))/(exp(x) + exp(-x))).

```
Train on 31367 samples, validate on 7842 samples
Epoch 1/2
31367/31367 [==============================] - 211s 7ms/step - loss: 8.1886
- accuracy: 0.0376 - val_loss: 8.6880 - val_accuracy: 0.0594
Epoch 2/2
31367/31367 [==============================] - 204s 6ms/step - loss: 7.5732
- accuracy: 0.0289 - val_loss: 7.2636 - val_accuracy: 0.0154
```

Selu:
Scaled Exponential Linear Unit (SELU). The Scaled Exponential Linear Unit (SELU) activation function is defined as:

- if x > 0: return scale * x
- if x < 0: return scale * alpha * (exp(x) - 1)

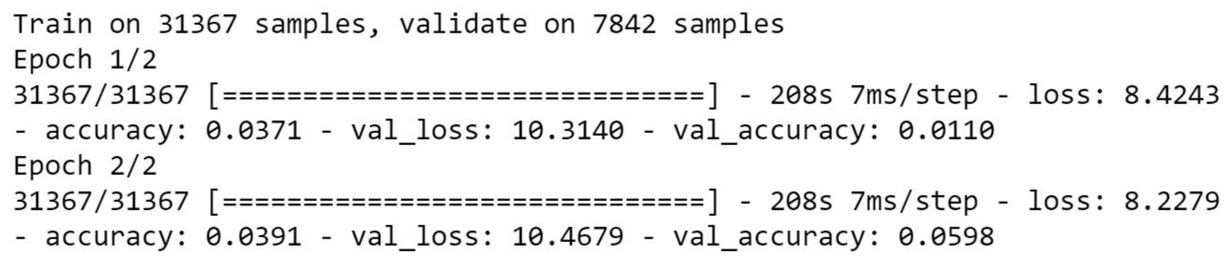
where alpha and scale are predefined constants (alpha=1.67326324 and scale=1.05070098). Basically, the SELU activation function multiplies scale (> 1) with the output of the tf.keras.activations.elu function to ensure a slope larger than one for positive inputs.

Arguments

- x: A tensor or variable to compute the activation function for.

Returns

The scaled exponential unit activation: scale * elu(x, alpha).



```
Train on 31367 samples, validate on 7842 samples
Epoch 1/2
31367/31367 [==============================] - 208s 7ms/step - loss: 8.4243
- accuracy: 0.0371 - val_loss: 10.3140 - val_accuracy: 0.0110
Epoch 2/2
31367/31367 [==============================] - 208s 7ms/step - loss: 8.2279
- accuracy: 0.0391 - val_loss: 10.4679 - val_accuracy: 0.0598
```

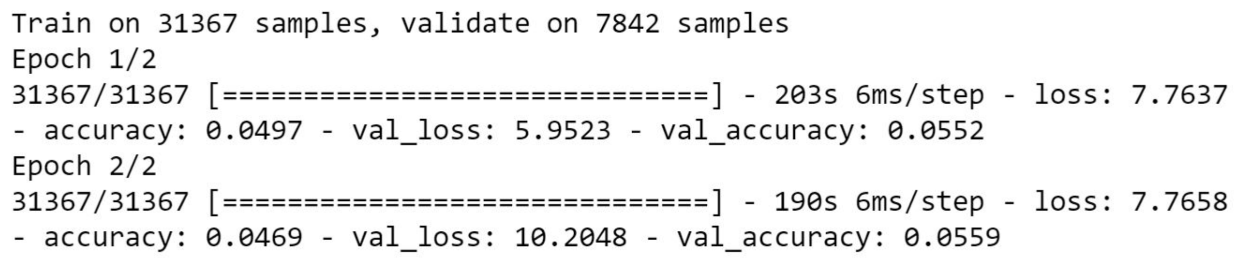
Elu:
Exponential linear unit.

Arguments

- x: Input tensor.
- alpha: A scalar, slope of negative section.

Returns

The exponential linear activation: x if x > 0 and alpha * (exp(x)-1) if x < 0.



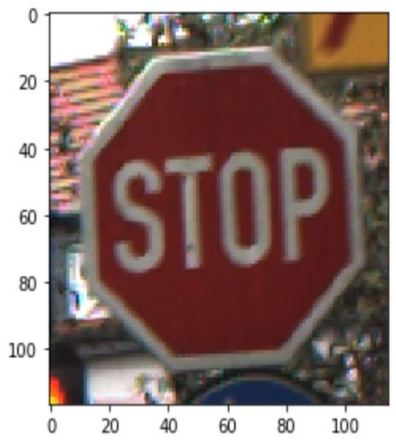
```
Train on 31367 samples, validate on 7842 samples
Epoch 1/2
31367/31367 [==============================] - 203s 6ms/step - loss: 7.7637
- accuracy: 0.0497 - val_loss: 5.9523 - val_accuracy: 0.0552
Epoch 2/2
31367/31367 [==============================] - 190s 6ms/step - loss: 7.7658
- accuracy: 0.0469 - val_loss: 10.2048 - val_accuracy: 0.0559
```

Finally, we selected the softmax activation as softmax is for multiple classes whereas sigmoid is for 2. Sigmoid is a special case of softmax. Moreover, softmax only needs to be applied to the last layer as it gives a probabilistic outcome which sums up to 1.

Here is an illustration on how the model worked,

```
pil_im = Image.open(test.iloc[224]['Path'], 'r')
mp.imshow(np.asarray(pil_im))
print('Lable for test example: '+str(test.iloc[224]['ClassId']))
```

```
Lable for test example: 14
```



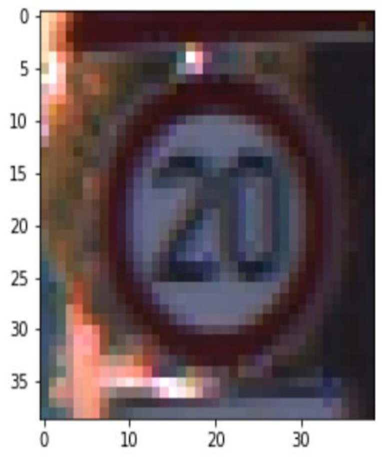
```
testVal=[]
testVal.append(np.array(np.array(Image.fromarray(cv2.imread(test.iloc[224]['Path']), 'RGB').resize((40,40)))))
testVal=np.array(testVal).astype('float32')/255
predictedLabels = model.predict_classes(testVal)
print('The predicted class label is:')
print(predictedLabels[0])
```

```
The predicted class label is:
14
```

Here the model predicts the label correctly. However, this is the case when there is ample training data for the class label 14. In cases like class label 0 where there is not much training data the model does not predict the correct label. For Instance,

```
pil_im = Image.open(test.iloc[243]['Path'], 'r')
mp.imshow(np.asarray(pil_im))
print('Lable for test example: '+str(test.iloc[243]['ClassId']))
```

```
Lable for test example: 0
```



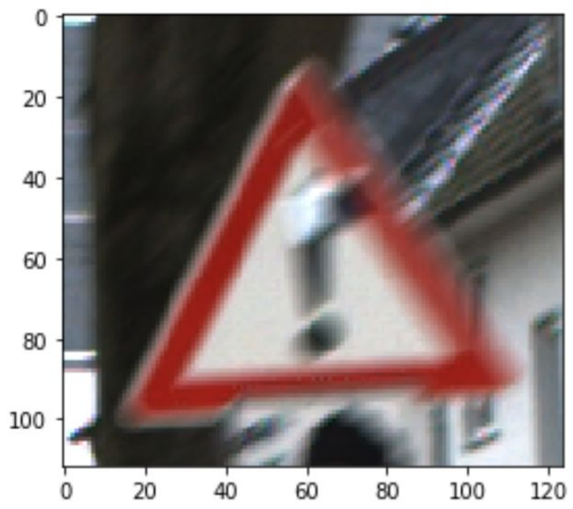
```
testVal=[]
testVal.append(np.array(np.array(Image.fromarray(cv2.imread(test.iloc[243]['Path']), 'RGB').resize((40,40)))))
testVal=np.array(testVal).astype('float32')/255
predictedLabels = model.predict_classes(testVal)
print('The predicted class label is:')
print(predictedLabels[0])
```

```
The predicted class label is:
13
```

At first we thought this is because the picture is very blurry. However, this was not the case for other blurry images.

```
pil_im = Image.open(test.iloc[270]['Path'], 'r')
mp.imshow(np.asarray(pil_im))
print('Lable for test example: '+str(test.iloc[270]['ClassId']))
```

```
Lable for test example: 18
```



```
testVal=[]
testVal.append(np.array(np.array(Image.fromarray(cv2.imread(test.iloc[270]['
testVal=np.array(testVal).astype('float32')/255
predictedLabels = model.predict_classes(testVal)
print('The predicted class label is:')
print(predictedLabels[0])
```

```
The predicted class label is:
18
```

That is how we concluded, model could not predict the sign from class 0 because of the lack of training data.

Bibliography

- Keiron O'Shea's and Ryan Nash's: An Introduction to Convolutional Neural Networks (November 2015)
- Ayyüce Kızrak's: Comparison of Activation Functions for Deep Neural Networks (May 2019)
- Caner Dabakoglu's: What is Convolutional Neural Network (CNN) ? — with Keras (December 2018)
- DataFlair team's: Python Project on Traffic Signs Recognition with 95% Accuracy using CNN & Keras (January 2020)
- Jiwon Jeong's: The Most Intuitive and Easiest Guide for Convolutional Neural Network (Jan 2019)
- O. Azouaoui's : Traffic signs recognition with deep learning (November 2018)
- Jingwei Cao's, Chuanxue Song's, Silun Peng's, Feng Xiao's, and Shixin Song's: Improved Traffic Sign Detection and Recognition Algorithm for Intelligent Vehicles
- Keras Layer activation Functions
- TensorFlow Convolutional Neural Network (CNN)
- Kaggle's German Traffic Sign Recognition Benchmark