EEN 614 Final Project - Classification of 43 Road Signs Using Convolutional Neural Networks

Albert Aninagyei Ofori,

Norfolk State University, 700 Park Avenue, Norfolk, USA, 23504

ABSTRACT

Computer vision plays a paramount role in today's efforts to improve and innovate in the field of self-driving vars. Self-driving cars of today may be required to store route information for various roads in use. Moreover, various information regarding the use of a road, such as hazards and speed limits have to be incorporated into the programming of the self-driving vehicle. These conditions could change with the seasons, with road expansion, or the occurrence of an accident which might require diversions. Here is where the ability of the vehicle to recognize various road signs in real-time is very expedient. This would also improve automation performance on new roads which may not have been documented yet. In this project, a system that can accurately detect and determine the type of road sign in a picture is implemented. A convolutional neural network is designed and trained using the GTSRB - German Traffic Sign Recognition Benchmark dataset available on Kaggle with 43 different classes in order to be able to identify the type of road sign in a given image. Amongst others, one possible application of this system is that its output can then be fed into a bigger system as an additional condition that affects the system autonomy. A simple web application based on the Streamlit library is then developed where an image can be uploaded and the corresponding type of road sign displayed. This web app is then deployed and can be found by visiting `https://morning-anchorage-96843.herokuapp.com`

1. INTRODUCTION

Convolutional neural networks (CNNs) have effectuated computer vision in diverse applications ranging from healthcare, through image processing to finance. Compared to artificial neural networks, convolutional neural networks draw inspiration from the intricacies of the human visual cortex. Similar to this part of the human brain, a convolutional neural network trains various sets of parameters for specific features of the image, which eventually come together to be able to perform holistic image recognition.[1] This occurs by the use of matrices known as kernels or filters, which are sets of weights whose values are optimized to detect a specific aspect of an image, as primitive as a vertical or horizontal line, but can get more advanced, for instance, in edge detection, or shape detection. In a convolutional layer, the weights of the network to be trained are limited to the kernel size, which then traverses the input matrix and produces an output that emphasizes certain features under study based on what the kernel is specialized for. The pooling layer also traverses the graph and based on what the aim is, this layer also traverses an input, which could be the output from a preceding layer, and performs a certain function on a specified region based on its kernel size and the strides. This could be finding the maximum value in that region, or averages all values in the region. The output of each layer is fed into the next until the processed data with some inference is fed into the terminal layers known as the fully connected layers which function like artificial neural networks.

In this project, we employ convolutional neural networks in the classification of various traffic signals using the German Traffic Sign Recognition Benchmark dataset. This dataset has a total of **51,839** different images which fall into 43 classes. The data has been partitioned such that the training data has a size of **39,209** and the test data has **12,630** images. To show this in action, a web application is then developed using the Streamlit library in python. In this web application, we take advantage of the trained model to predict the type of sign that an uploaded image may be.

2. METHODOLOGY

We first begin by training a convolutional neural network with three convolutional layers interspersed with pooling layers, and 2 fully connected layers at its terminus. In order to improve the performance of the model as far as computational burden and accuracy are concerned, we then proceed to make modifications to the structure of the network, first by changing the network design, then adding some dropout layers in order to increase the capacity of the system without overfitting. The best achieved model is then tested randomly with images from the data set in order to confirm whether the actual and predicted class of an image are the same. The various model configurations that were implemented and tested are outlined below;

1. A convolutional neural network with a CONV-POOL-CONV-POOL-CONV-POOL-FC-FC architecture. (128-128-128-256-43)
2. A convolutional neural network with a CONV-CONV-POOL-CONV-CONV-POOL-FC-FC architecture. (64-64-32-16-128-43)
3. A convolutional neural network with a CONV-CONV-POOL-DROPOUT-CONV-CONV-POOL-DROPOUT-FC-DROPOUT-FC-DROPOUT-FC architecture. (64-32-32-16-256-128-43)

2.1 Parameters and Output size

Depending on the configuration, input size and layer width of various layers in a neural network, the number of parameters, and therefore the complexity of a network may vary. The pooling layer, for instance, does not have any parameters. This is because no weighted multiplication and addition calculations are performed within this layer, however, depending on the type of layer, a given condition such as the maximum value or average value of data within a kernel space in a particular stride determines the output. Similarly, any dropout layers utilized within a model do not add any parameters to the model since they only effectuate a random omission of certain outputs from the preceding layer thereby virtually making the layer appear more sparse.

The convolutional layers, like the pooling layers, also utilizes kernels, however, unlike the pooling layer, these kernels traverse over their input matrix and the weighted sum of the values in that kernel space per the weights in the kernel. The kernels therefore always have an identical depth to the input data in order to ensure no dimensional mismatch occurs. The output of this matrix multiplication is then biased by a fixed value, ergo, the number of biases in a network is identical to the number of kernels. The output layer depends on the number of kernels that are used in the convolutional layer. This also determines the size of the bias since each kernel has a bias.

Generally, for a kernel of size $a \times b$ and an input image of depth c, if the number of kernels specified (and therefore the depth of the layer output) is d, then the number of parameters in the convolutional layer (weights and biases) can be represented such that:

$$\text{Number of parameters} = (a \times b \times c) \times d + d = [(a \times b \times c) + 1] \times d.$$

In the dense layer, or fully connected layer, the number of parameters for an input of size m and a layer of size n will be such that:

$$m \times n + n = (m + 1) \times n$$

The output layers were calculated from the size of each pooling or convolutional layer such that:

$$\text{Output size} = \frac{\text{input size} - \text{kernel size}}{\text{number of strides}} + 1$$

2.2 Image Data Processing

The obtained dataset was required to be preconditioned since the data on kaggle existed as grouped folders containing files in .jpg format. The GTSRB data set consists of 51,839 different sets of images belonging to 43 classes.[2] The data could either be downloaded, preprocessed and saved into a numpy array in a .npy file, which could easily be loaded and used elsewhere, or it can be directly downloaded from Kaggle for real-time use, however, this would require an API key which must be generated from the user's account. For this project, the latter of the two methods was used in order to enable access to the latest data in the event where the test and training data is updated. In order to precondition the data into a form that could be easily ingested in code for model training, the data was first downloaded and the image data was opened using the python PIL library which had an "Image" method with which the image data could be processed. The images were then resized and cast as numpy arrays. In order to ensure that data lies within a suitable range for our activation functions, the data was normalized to fall within the range of 0.0-1.0. Since the data points shown in each data set indicate the intensity of that on the red, yellow and blue color scale (and therefore a depth of 3), which ranges from 0-255, the maximum value by which we normalize our data is 255. This value therefore divides each data set value in order to ensure no value exceeds 1. To avoid programmatic division of integers which discard the decimal values, we cast all data set values as floats which then accommodate decimals which may occur during normalization. These arrays were then appended to a python list and the labels also appended to a separate list. This was iteratively carried out to ensure that all training data in the 43 classes were extracted with their labels. The data in the arrays were extracted in chronological order ranging from the 1st class to the 43rd class, a validation split was required, which would sample the data in a random enough manner such that there would be no bias in the split data such that the test data would end up being more optimized for some classes of data over others. This would affect generalization. The sci-kit learn "train_test_split" method comes in handy in dealing with this potential issue and is used to split the preprocessed training data into a training and validation dataset with the ratio 80:20. The first 5 images in the dataset were then plotted using the matplotlib library as shown in Fig. 1.

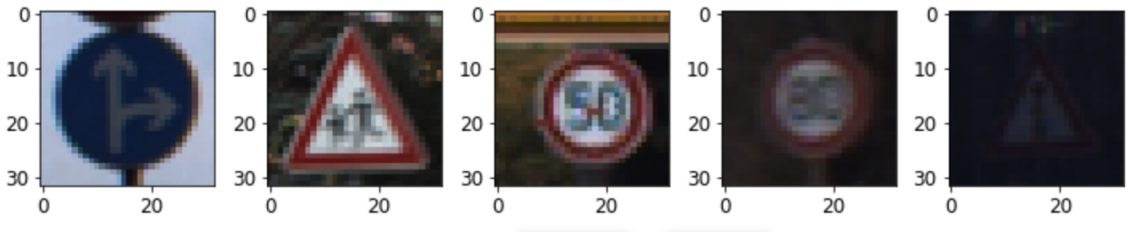

Figure 1. Colored images of GTSRB data set sample images

2.3 CNN with CONV-POOL-CONV-POOL-CONV-POOL-FC-FC

As a starting point from which various improvements could be made, a CNN with 3 convolutional layers alternating with MaxPooling layers is implemented, and its performance and accuracy in detecting the image features are analyzed. The convolutional layer is set up such that the kernels (or filters) have a size of 3 x 3 and a stride of 1 is used in this layer. The MaxPooling layers are configured with kernels of size 2 x 2 and a stride of 1. The fully connected layers ("FC") are the dense layers that were utilized extensively in artificial neural networks. The keras.Sequential model is employed in setting up the requisite layers in the order suggested by the nomenclature of this model.

The parameters were calculated based on formulas indicated above as shown in the table 1 and the total value found to be **17,644,331**.

LAYER(SIZE)	INPUT SIZE	KERNEL SIZE	PARAMETERS	OUTPUT SIZE
CONV1(128)	32 x 32 x 3	3x3	$[(3 \times 3 \times 3) + 1] \times 128 = 3,584$	$\frac{32-3}{1} + 1 = 30$ x 30 x 128
POOL1	30 x 30 x 128	2 x 2	0	$\frac{30-2}{1} + 1 = 29$ x 29 x 128
CONV2(128)	29 x 29 x 128	3 x 3	$[(3 \times 3 \times 128) + 1] \times 128 = 147,584$	$\frac{29-3}{1} + 1 = 27$ x 27 x 128
POOL2	27 x 27 x 128	2 x 2	0	$\frac{27-2}{1} + 1 = 26$ x 26 x 128
CONV3(128)	26 x 26 x 128	3 x 3	$[(3 \times 3 \times 128) + 1] \times 128 = 147,584$	$\frac{26-3}{1} + 1 = 24$ x 24 x 128
POOL3	24 x 24 x 128	2 x 2	0	$\frac{24-2}{1} + 1 = 23$ x 23 x 128
FC1(256)	67712(Flattened)	N/A	$(67712 + 1) \times 256 = 17,334,528$	256 x 1
FC2(43)	256 x 1	N/A	$(256 + 1) \times 43 = 11,051$	43 x 1
			Total = **17,644,331**	

Table 1. Parameters for CONV-POOL-CONV-POOL-CONV-POOL-FC-FC CNN

The Rectified Linear Unit (ReLU) tends to be suitable as the activation function in the convolutional layers. This activation function was chosen over the sigmoid function which is susceptible to the vanishing gradient issue which limits learning for values at its extreme ends. Since this is a classification network, the use of a Softmax activation function proves suitable for the output layer since it provides a probability distribution for the 43 outputs corresponding to each class, adding up to 1. This has proven very suitable for for multi-class image classification as it identifies the output that has the highest probability as the target class of the input.[34] The system was then trained with 80% of the 39,209 training data sets with the 'adam' optimizer and the remaining 20% was used in validation. The validation data, x_val and y_val are also specified in fitting the model. A batch size of 128 was used in the training therefore resulting in 246 training batches.

2.4 REDUCING COMPLEXITY WITH CNN WITH CONV-CONV-POOL-CONV-CONV-POOL-FC-FC ARCHITECTURE

In the second iteration of the model, we proceed to reduce the complexity of the system by changing the network structure as suggested by the name. Moreover, the various convolutional layers are made to taper into the model. This is done with the aim of the reducing the number of parameters since the first model suggests serious complexity which is evident from it runtime of 10 s per epoch even with the GPU. The activation functions used in the the convolutional layers (ReLu) and the outermost fully connected layer (Softmax) are maintained here and throughout the rest of the simulation process. A batch size of 128 and other training parameters such as kernel size were maintained in the training process. The number of strides, however, was doubled in the pooling layers. The parameters in this model were calculated and were observed to be way less than in the previous model as suggested in table 2

LAYER(SIZE)	INPUT SIZE	KERNEL SIZE	PARAMETERS	OUTPUT SIZE
CONV1(64)	32 x 32 x 3	3 x 3	$[(3 \times 3 \times 3) + 1] \times 64 = 1,792$	$\frac{32-3}{1} + 1 = 30$ x 30 x 64
CONV2(32)	30 x 30 x 64	3 x 3	$[(3 \times 3 \times 64) + 1] \times 32 = 18,464$	$\frac{30-3}{1} + 1 = 28$ x 28 x 32
POOL1	28 x 28 x 32	2 x 2	0	$\frac{28-2}{2} + 1 = 14$ x 14 x 32
CONV3(32)	14 x 14 x 32	3 x 3	$[(3 \times 3 \times 32) + 1] \times 32 = 9,248$	$\frac{14-3}{1} + 1 = 12$ x 12 x 32
CONV4(16)	12 x 12 x 32	3 x 3	$[(3 \times 3 \times 32) + 1] \times 16 = 4,624$	$\frac{12-3}{1} + 1 = 10$ x 10 x 16
POOL2	10 x 10 x 16	2 x 2	0	$\frac{10-2}{2} + 1 = 5$ x 5 x 16
FC1(128)	400 x 1(Flattened)	N/A	$(400 + 1) \times 128 = 51,328$	128 x 1
FC2(43)	128 x 1	N/A	$(128 + 1) \times 43 = 5,547$	43 x 1
			Total = **91,003**	

Table 2. Parameters for CONV-CONV-POOL-CONV-CONV-POOL-FC-FC CNN

The total number of parameters employed in this system have been calculated in table 2 and was found to be **91,003**. This is a significant reduction over the previous implementation since the size of the convolutional layers was reduced with each progressing layer. The number of weights and biases required in each layer was therefore less than in our initial model.

2.5 IMPROVING ACCURACY WITH INCREASED CAPACITY AND DROPOUTS

In an attempt to improve the accuracy of the system, we increase the depth by adding an additional dense layer with a size of 256. Moreover, dropout layers were then inserted after each pooling layer and before each fully connected layer. This results in a CONV-CONV-POOL-DROPOUT-CONV-CONV-POOL-DROPOUT-FC-DROPOUT-FC-DROPOUT-FC. All other parameters were maintained so we would know the exact effect of an additional fully-connected layer and dropouts in the model. As can be seen from table 3, the total number of parameters used here was found to be **175,227**.

LAYER(SIZE)	INPUT SIZE	KERNEL SIZE	PARAMETERS	OUTPUT SIZE
CONV1(64)	32 x 32 x 3	3 x 3	$[(3 \times 3 \times 3) + 1] \times 64 = 1,792$	$\frac{32-3}{1} + 1 = 30$ x 30 x 64
CONV2(32)	30 x 30 x 64	3 x 3	$[(3 \times 3 \times 64) + 1] \times 32 = 18,464$	$\frac{30-3}{1} + 1 = 28$ x 28 x 32
POOL1	28 x 28 x 32	2 x 2	0	$\frac{28-2}{2} + 1 = 14$ x 14 x 32
DROPOUT1	14 x 14 x 32	N/A	0	14 x 14 x 32
CONV3(32)	14 x 14 x 32	3 x 3	$[(3 \times 3 \times 32) + 1] \times 32 = 9,248$	$\frac{14-3}{1} + 1 = 12$ x 12 x 32
CONV4(16)	12 x 12 x 32	3 x 3	$[(3 \times 3 \times 32) + 1] \times 16 = 4,624$	$\frac{12-3}{1} + 1 = 10$ x 10 x 16
POOL2	10 x 10 x 16	2 x 2	0	$\frac{10-2}{2} + 1 = 5$ x 5 x 16
DROPOUT2	5 x 5 x 16	N/A	0	5 x 5 x 16
FC1(256)	400 x 1(Flattened)	N/A	$(400 + 1) \times 256 = 102,656$	256 x 1
DROPOUT3	256 x 1	N/A	0	256 x 1
FC2(128)	256 x 1	N/A	$(256 + 1) \times 128 = 32,896$	128 x 1
DROPOUT4	128 x 1	N/A	0	128 x 1
FC3(43)	128 x 1	N/A	$(128 + 1) \times 43 = 5,547$	43 x 1
			Total = **175,227**	

Table 3. Parameters for CONV-CONV-POOL-DROPOUT-CONV-CONV-POOL-DROPOUT-FC-DROPOUT-FC-DROPOUT-FC CNN with diminishing layer sizes and dropouts

3. RESULTS

3.1 CNN with CONV-POOL-CONV-POOL-CONV-POOL-FC-FC

From the observed training time, this initial system is found to have a high computational burden which as can be seen from the training time of **10 s** per epoch even with the GPU runtime enabled. From Fig. 2, it is observed that the training loss keeps reducing with each training epoch. The validation loss starts out relatively randomly however, it tends to maintain a constant difference with the validation loss after the 10th epoch. This model appears to generalize quite favorably and the final training and validation accuracies are **99.93%** and **99.31%** respectively (a variation of **0.62%**) with a test accuracy of **96.19%**.

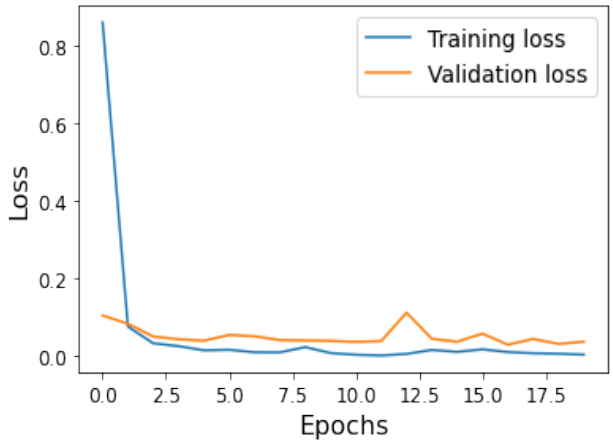

Figure 2. Comparison of training and validation loss for 20 epochs in a CONV-POOL-CONV-POOL-CONV-POOL-FC-FC

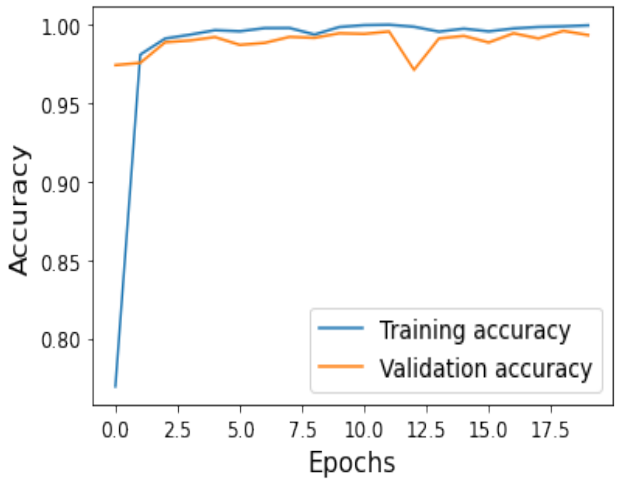

Figure 3. Comparison of training and validation accuracy for 20 epochs in a CONV-POOL-CONV-POOL-CONV-POOL-FC-FC

3.2 REDUCING COMPLEXITY WITH CNN WITH CONV-CONV-POOL-CONV-CONV-POOL-FC-FC ARCHITECTURE

The reduction in the number of parameters from the first model is very great since only **0.5%** of the parameters in our initial model are employed in this model. As expected, the computational burden is greatly reduced and the training time per epoch in Tensorflow goes down to **2 s**. This model also tends to generalize better with a variation of **0.55%** between the final training accuracy (**99.89%**) and validation accuracy (**98.34%**). The test accuracy, however, saw a decrease to **95.29%**.

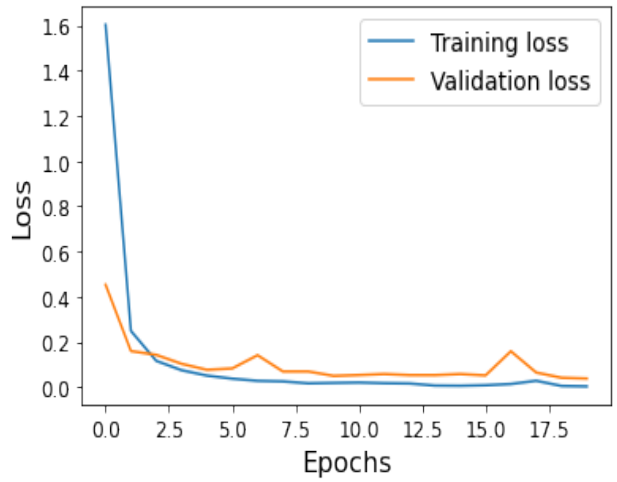

Figure 4. Comparison of training and validation loss for 20 epochs in a CONV-CONV-POOL-CONV-CONV-POOL-FC-FC

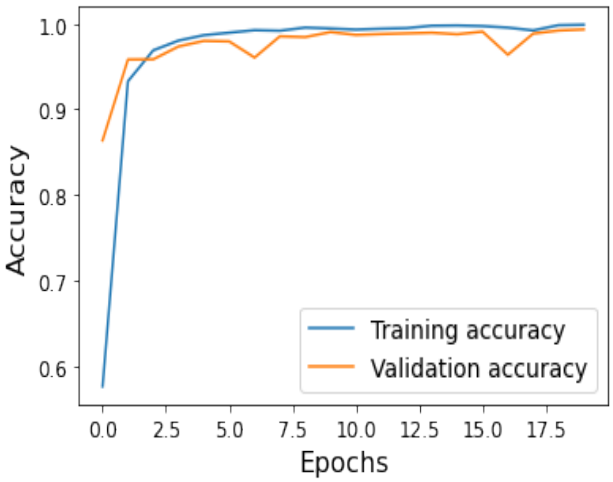

Figure 5. Comparison of training and validation accuracy for 20 epochs in a CONV-CONV-POOL-CONV-CONV-POOL-FC-FC

3.3 IMPROVING ACCURACY WITH INCREASED CAPACITY AND DROPOUTS

Fig. 6 and 7 show the training and validation loss of the modified network as described in section 2.5 with additional fully connected layers and dropout layers. The validation accuracy tends to be higher than the training accuracy with this model. The final training accuracy is **97.44%** with a corresponding validation accuracy of **99.59%**. The test accuracy, however, proves to be closer to the training accuracy at **97.4%**. This also happens to be the best accuracy achieved of the three models. This model also maintained a relatively low number of parameters compared to our first network implementation. Its computational efficiency was also confirmed with a training time of **2 s** per epoch which is no worse than the performance of the model with the least parameters. Due to its combination of efficiency, low complexity and accuracy, this model was used as the final model in the implementation of our image recognition system. The model was then saved and utilized in implementing a web-based solution detailed in the ensuing sections of this report.

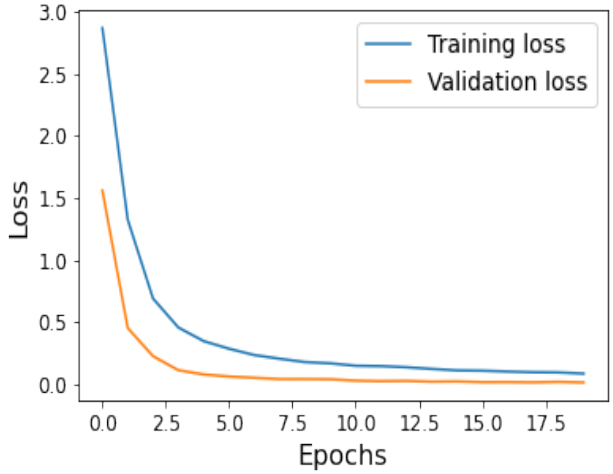

Figure 6. Comparison of training and validation loss for 30 epochs in a CONV-CONV-POOL-DROPOUT-CONV-CONV-POOL-DROPOUT-FC-DROPOUT-FC-DROPOUT-FC CNN with diminishing layer sizes and dropouts

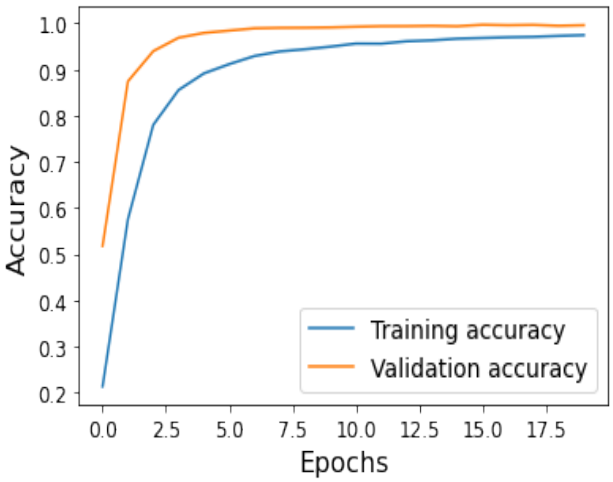

Figure 7. Comparison of training and validation accuracy for 30 epochs in a CONV-CONV-POOL-DROPOUT-CONV-CONV-POOL-DROPOUT-FC-DROPOUT-FC-DROPOUT-FC CNN with diminishing layer sizes and dropouts

4. MODEL TESTING AND IMPLEMENTATION

4.1 Testing the model

In order to see the indicated accuracy at work, the first 5 and last 5 values in the test data set were then plotted. The class of each model is predicted by using the numpy argmax function to find the index of the highest occurring softmax output for each image passed into the model's "predict" method.The actual labels were indicated as the title at the top each image plot and the predicted class indicated at the bottom. The results of this test proved the model was highly accurate as suggested in Fig. 8.

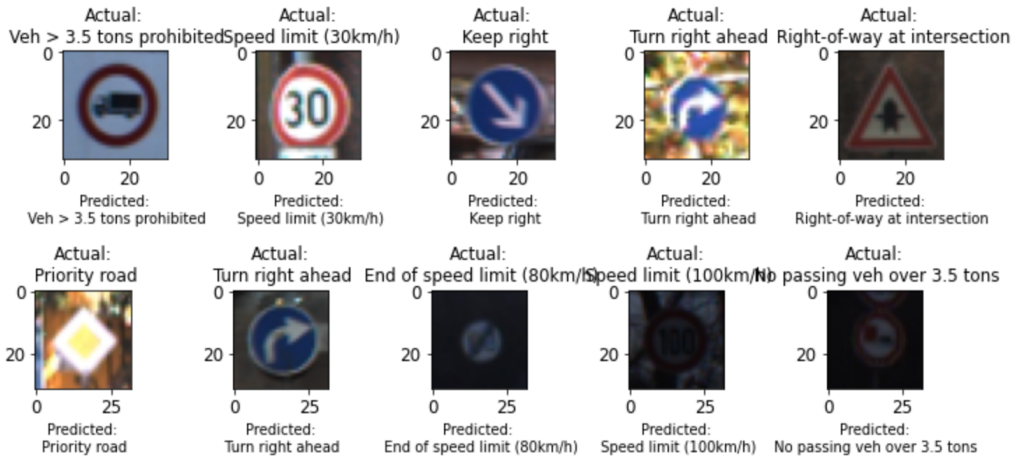

Figure 8. Actual and predicted values indicated for the first and last 5 images in the test data set.

4.2 Web Application Implementation

A simple web application is finally developed using the Streamlit library, which is a library that greatly simplifies the deployment of web applications using python code and comes in very handy in numerous data science visualization applications. This web application was configured to enable a user upload an image and then the image class would be displayed along with the uploaded image. In code, the saved model was loaded for use in predicting the class of this image. The individual classes of the images were also saved and loaded as

a .json file and loaded into the web application. This was done so that the caching feature of Streamlit could be employed in ensuring that the individual classes do not have to be loaded over and over again unless there was an update made to it. The system was set up such that an uploaded image was first resized as done in the data preprocessing phase of the project using the PIL library, and then cast as a normalized numpy array of floats. This data was then passed into the model 'predict' method. The returned value was then passed into the numpy argmax function which returns the index of the most likely class. This is then passed into the dictionary of sign classes to display the name of the predicted class. The code for this implementation can be found in the appendix. This webapp was then deployed using Heroku[5] and can be accessed by visiting `https://morning-anchorage-96843.herokuapp.com` . Some screenshots from the developed GUI can be found in figures 9 and 10.

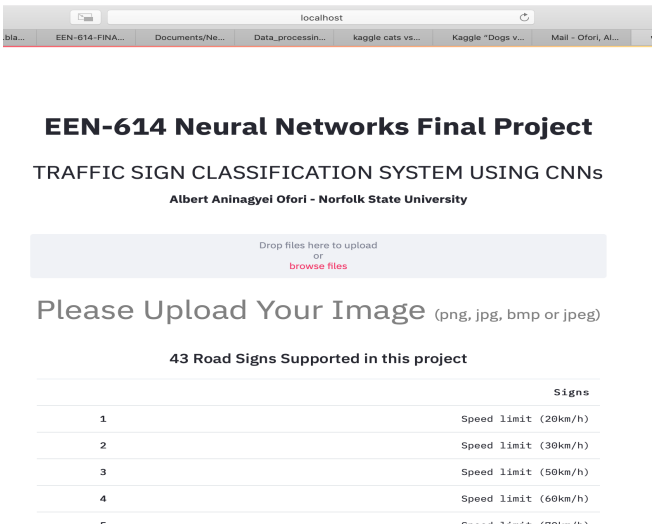

Figure 9. Homepage displayed upon starting web application

Figure 10. Predicting and displaying the class of a random road sign downloaded from the internet

5. CONCLUSION

In this project, the importance of CNNs in computer vision is practically demonstrated using using the German Traffic Sign Recognition Benchmark data set. A model is iteratively developed by making various modification to an initial model architecture and adding some dropout layers. Though all models tend to generalize well, we are able to improve an initial test accuracy of 96.19% and achieve a final accuracy of 97.4% in 20% the initial training time. This model is then implemented in a web application using the Streamlit library, where images can be uploaded and their classes displayed to show how inferences can be made from captured images of signs on the road. The speed with which predictions are made demonstrates its potential for use in autonomous vehicles where various changing road conditions can be inferred in real-time from road signs the vehicle may encounter.

6. APPENDIX

6.1 Code for modelling of network

```
pip install kaggle

 #Import all prerequisite libraries
from PIL import Image
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt import matplotlib
import os
import sklearn.model_selection as skl from zipfile import ZipFile
import google.colab.files
# Specify plot label tick size
matplotlib.rc("xtick", labelsize=12) matplotlib.rc("ytick", labelsize=12)

 #Downloading the dataset from kaggle
google.colab.files.upload()
!mkdir -p ~/.kaggle
!chmod 600 ~/.kaggle/kaggle.json
!cp kaggle.json ~/.kaggle
!kaggle datasets download -d meowmeowmeowmeowmeow/gtsrb-german-traffic-sign
with ZipFile('/content/gtsrb-german-traffic-sign.zip',mode='r') as info:
    info.extractall()

 num_classes=43
test_data=[]
test_labels=[]
test_info=pd.read_csv('Test.csv')
image_data=[]
image_labels=[]
def dim_change(file_path):
     im=Image.open(file_path)
     im=im.resize((32,32))
     data=np.array(im).astype('float32')
     return data/255.0
#Preparing the training data
for image_class in range(num_classes):

    for file_name in os.listdir(f'Train/{image_class}'):
```

```
        image_data.append(dim_change(f'Train/{image_class}/{file_name}'))
        image_labels.append(image_class)
    print(f'Class {image_class} done')

image_data=np.array(image_data)
image_labels=np.array(image_labels)
print("Training Data Done")
print(image_data.shape)
print(image_labels.shape)

#Preparing the test data
print('Starting Test Data')
for id, path in zip(test_info.ClassId, test_info.Path):
    test_data.append(dim_change(path))
    test_labels.append(id)
test_data=np.array(test_data)
test_labels=np.array(test_labels)
print(test_data.shape)
print(test_labels.shape)

#Split data into training and validation sets
x_train, x_val, y_train, y_val = skl. →train_test_split(image_data,image_labels,train_size=0.8,
test_size=0.2,␣ →random_state=42)

#Display sample data
fig=plt.figure(figsize=(10,20))
for i in range(5):
    fig.add_subplot(1,5,i+1)
    plt.imshow(x_train[i])
fig.tight_layout()

 #CNN MODEL WITH A CONV-POOL-CONV-POOL-CONV-POOL-FC-FC ARCHITECTURE
model = keras.models.Sequential(
[keras.layers.Conv2D(filters=128, kernel_size=(3,3),␣
→activation='relu', input_shape=(32,32,3)),
keras.layers.MaxPool2D(pool_size=(2, 2), strides=1),
keras.layers.Conv2D(filters=128, kernel_size=(3, 3),␣ →activation='relu', strides=1),
keras.layers.MaxPool2D(pool_size=(2, 2), strides=1),
keras.layers.Conv2D(filters=128, kernel_size=(3, 3),␣ →activation='relu', strides=1),
keras.layers.MaxPool2D(pool_size=(2, 2), strides=1), keras.layers.Flatten(),
keras.layers.Dense(256, activation='relu'), keras.layers.Dense(43, activation='softmax')]
)

# Compile model using adam optimizer
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',␣ →metrics=['accuracy'])

# Train model
history=model.fit(x_train, y_train, batch_size=128,␣ →epochs=20,
validation_data=(x_val,y_val))
# Display various layers of CNN
model.summary()
# Evaluate model against test data
```

```
test_loss, test_acc = model.evaluate(test_data, test_labels, verbose=2)
print(f'\n Test accuracy: {test_acc}')


# Compare training and validation loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.xlabel('Epochs',fontsize=16)
plt.ylabel('Loss', fontsize=16)
plt.legend(['Training loss', 'Validation loss'], fontsize=14)
plt.show()
# Plot training and validation accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.xlabel('Epochs',fontsize=16)
plt.ylabel('Accuracy', fontsize=16)
plt.legend(['Training accuracy', 'Validation accuracy'], fontsize=14)
plt.show()


 #CNN MODEL WITH CONV-CONV-POOL-CONV-CONV-POOL-FC-FC ARCHITECTURE
model_mod1 = keras.models.Sequential(
[keras.layers.Conv2D(filters=64, kernel_size=(3,3),␣
→activation='relu', input_shape=(32,32,3)),
keras.layers.Conv2D(filters=32, kernel_size=(3, 3),␣
→activation='relu', strides=1),
keras.layers.MaxPool2D(pool_size=(2, 2), strides=2),
keras.layers.Conv2D(filters=32, kernel_size=(3, 3),␣ →activation='relu', strides=1),
keras.layers.Conv2D(filters=16, kernel_size=(3, 3),␣ →activation='relu', strides=1),
keras.layers.MaxPool2D(pool_size=(2, 2), strides=2),
keras.layers.Flatten(),
keras.layers.Dense(128, activation='relu'), keras.layers.Dense(43, activation='softmax')]
)

# Compile model using adam optimizer
model_mod1.compile(optimizer='adam',loss='sparse_categorical_crossentropy',␣ →metrics=['accuracy'])


# Train model
history_mod1=model_mod1.fit(x_train, y_train, batch_size=128,␣ →epochs=20,validation_data=(x_val,y_val))

# Display various layers of CNN
model_mod1.summary()
# Evaluate model against test data
test_loss_mod1, test_acc_mod1 = model_mod1.evaluate(test_data, test_labels,␣ →verbose=2)
print(f'\n Test accuracy: {test_acc_mod1}')
# Compare training and validation loss
plt.plot(history_mod1.history['loss'])
plt.plot(history_mod1.history['val_loss'])
plt.xlabel('Epochs',fontsize=16)
plt.ylabel('Loss', fontsize=16)
plt.legend(['Training loss', 'Validation loss'], fontsize=14)
```

```
plt.show()

# Plot training and validation accuracy
plt.plot(history_mod1.history['accuracy']) plt.plot(history_mod1.history['val_accuracy'])
plt.xlabel('Epochs',fontsize=16)
plt.ylabel('Accuracy', fontsize=16)
plt.legend(['Training accuracy', 'Validation accuracy'], fontsize=14)
plt.show()


 #DEFINE A MODEL WITH␣ →CONV-CONV-POOL-DROPOUT-CONV-CONV-POOL-DROPOUT-FC-DROPOUT-FC
 #-DROPOUT-FC␣ →ARCHITECTURE
model_mod2 = keras.models.Sequential(
[keras.layers.Conv2D(filters=64, kernel_size=(3,3),␣
→activation='relu', input_shape=(32,32,3), strides=1), keras.layers.Conv2D(filters=32, kernel_size=(3,3),
→activation='relu', strides=1),
keras.layers.MaxPool2D(pool_size=(2, 2), strides=2),
keras.layers.Dropout(rate=0.25),
keras.layers.Conv2D(filters=32, kernel_size=(3, 3),␣ →activation='relu', strides=1),
keras.layers.Conv2D(filters=16, kernel_size=(3, 3),␣ →activation='relu', strides=1),
keras.layers.MaxPool2D(pool_size=(2, 2), strides=2), keras.layers.Dropout(rate=0.25),
keras.layers.Flatten(),
keras.layers.Dense(256, activation='relu'), keras.layers.Dropout(rate=0.5),
keras.layers.Dense(128, activation='relu'), keras.layers.Dropout(rate=0.5),
keras.layers.Dense(43, activation='softmax')]
)


# Compile model using adam optimizer
model_mod2.compile(optimizer='adam',loss='sparse_categorical_crossentropy',␣ →metrics=['accuracy'])


# Train model
history_mod2=model_mod2.fit(x_train, y_train, batch_size=128,␣ →epochs=20,validation_data=(x_val,y_val))
# Display various layers of CNN
model_mod2.summary()


# Evaluate model against test data
test_loss_mod2, test_acc_mod2 = model_mod2.evaluate(test_data, test_labels,␣ →verbose=2)
print(f'\n Test accuracy: {test_acc_mod2}')

# Compare training and validation loss
plt.plot(history_mod2.history['loss'])
plt.plot(history_mod2.history['val_loss'])
plt.xlabel('Epochs',fontsize=16)
plt.ylabel('Loss', fontsize=16)
plt.legend(['Training loss', 'Validation loss'], fontsize=14)
plt.show()

# Plot training and validation accuracy
plt.plot(history_mod2.history['accuracy']) plt.plot(history_mod2.history['val_accuracy'])
```

```
plt.xlabel('Epochs',fontsize=16)
plt.ylabel('Accuracy', fontsize=16)
plt.legend(['Training accuracy', 'Validation accuracy'], fontsize=14)
plt.show()


 # Define the 43 classes of the road signs
sign_classes={0: 'Speed limit (20km/h)', 1: 'Speed limit (30km/h)',
2: 'Speed limit (50km/h)',
3: 'Speed limit (60km/h)',
4: 'Speed limit (70km/h)',
5: 'Speed limit (80km/h)',
6: 'End of speed limit (80km/h)', 7: 'Speed limit (100km/h)',
8: 'Speed limit (120km/h)',
9: 'No passing',
10: 'No passing veh over 3.5 tons', 11: 'Right-of-way at intersection', 12: 'Priority road',
13: 'Yield',
14: 'Stop',
15: 'No vehicles',
16: 'Veh > 3.5 tons prohibited', 17: 'No entry',
18: 'General caution',
19: 'Dangerous curve left',
20: 'Dangerous curve right',
21: 'Double curve',
22: 'Bumpy road',
23: 'Slippery road',
 24: 'Road narrows on the right',
25: 'Road work',
26: 'Traffic signals',
27: 'Pedestrians',
28: 'Children crossing',
29: 'Bicycles crossing',
30: 'Beware of ice/snow',
31: 'Wild animals crossing',
32: 'End speed + passing limits',
33: 'Turn right ahead',
34: 'Turn left ahead',
35: 'Ahead only',
36: 'Go straight or right',
37: 'Go straight or left',
38: 'Keep right',
39: 'Keep left',
40: 'Roundabout mandatory',
41: 'End of no passing',
42: 'End no passing veh > 3.5 tons'}

 # Predict the classes for the first 5 images in the test dataset
pred_first=[sign_classes[i] for i in (np.argmax(i) for i in model_mod2. →predict(test_data[:5]))]
pred_last=[sign_classes[i] for i in (np.argmax(i) for i in model_mod2. →predict(test_data[-5:]))]

# Display the first 5 images of the test data and show their actual and␣ →predicted class
fig=plt.figure(figsize=(10,20))
```

```
for i in range(5):
    fig.add_subplot(1,5,i+1)
    plt.imshow(test_data[i])
    plt.title(f"Actual:\n {sign_classes[test_labels[i]]}")
    plt.xlabel(f"Predicted:\n {pred_first[i]}")
fig.tight_layout()

# Display the last 5 images of the test data and show their actual and␣
→predicted class
fig=plt.figure(figsize=(10,25))
for i in range(5):
    fig.add_subplot(1,5,i+1)
    plt.imshow(test_data[i-5])
    plt.title(f"Actual:\n {sign_classes[test_labels[i-5]]}")
    plt.xlabel(f"Predicted:\n {pred_last[i]}")
fig.tight_layout()
```

6.2 Code for web application

```
import numpy as np
import pandas as pd
import streamlit as st
from tensorflow.keras.models import load_model
from PIL import Image
import os
import json

@st.cache
def load_classes():
    with open('sign_classes.json','r') as file:
    class_names=json.load(file)
return class_names


#Load model and class names
model=load_model('final_model.h5')
sign_classes=load_classes()


#Change Image Dimensions
def dim_change(file_path):
    im=Image.open(file_path)
    im=im.resize((32,32))
    data=np.array(im).astype('float32')
    return data/255.0

#Display titles
st.markdown("<h1 style= 'text-align:center'> EEN-614 Neural Networks Final Project
</h1>", unsafe_allow_html=True)
st.markdown("<h2 style = 'text-align: center'>TRAFFIC SIGN CLASSIFICATION SYSTEM USING
CNNs</h2><p style = 'text-align: center'><b>Albert Aninagyei Ofori - Norfolk State
```

```
University</b></p>", unsafe_allow_html=True)
img=st.file_uploader("", type=["jpg", "jpeg", "png", "bmp"])

#Tell user to upload image if no image is uploaded
if img==None:
    st.markdown("<p style='color: grey; text-align: center; font-size: 40px'> Please
    Upload Your Image <span style='font-size:20px'> (png, jpg, bmp or jpeg)</span></p>",
    unsafe_allow_html=True)

#Process and display image with class on top if valid or indicate that image is invalid
else:
    try:
        img_data=np.array([dim_change(img)])
        image_prediction=model.predict(img_data)[0]
        image_class=sign_classes[str(np.argmax(image_prediction))]

     st.markdown(f"<h3 style= 'text-align:center'> This image is a <b>{image_class}</b>
     sign </h3>", unsafe_allow_html=True)
     st.image(img,use_column_width=True)

    except:
     st.markdown("<p style='color: grey; text-align: center; font-size: 30px'> Image
     format error. Please use another image </p>", unsafe_allow_html=True)

#List all supported classes
st.markdown("<h3 style= 'text-align:center'> 43 Road Signs Supported in this project
</h3>", unsafe_allow_html=True)
st.table(pd.Series(data=list(sign_classes.values()),name="Signs", index=range(1,len(sign_classes)+1)))
```

REFERENCES

1. S. Saha, *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*, 2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
2. Mykola, *GTSRB - German Traffic Sign Recognition Benchmark*, 2018. [Online]. Available: https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign
3. *7 Types of Neural Network Activation Functions: How to Choose?* [Online]. Available: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/
4. *Analyzing different types of activation functions in neural networks — which one to prefer?* [Online]. Available: https://towardsdatascience.com/analyzing-different-types-of-activation-functions-in-neural-networks-which-one-to-prefer-e11649256209
5. G. Tanner, *Deploying your Streamlit dashboard with Heroku*, 2019. [Online]. Available: https://gilberttanner.com/blog/deploying-your-streamlit-dashboard-with-heroku