University of Tehran

School of Mechanical Engineering

Artificial Intelligence

<u>**Home Work 5**</u>

Professor:

Dr. Masoud Shariat Panahi

Author:

Masoud Pourghavam

June, 2023

Table of Contents

List of Figures

List of Tables

1. Basic concepts

A. Classification and regression loss functions

a. Classification loss functions

1. **Cross-Entropy Loss (Log Loss):** Cross-entropy loss is commonly used in classification tasks, especially in multi-class classification problems. It measures the dissimilarity between the predicted probabilities and the true class labels. The loss function is calculated by taking the negative logarithm of the predicted probability of the correct class. It encourages the model to assign a higher probability to the correct class and penalizes the model for incorrect predictions. Cross-entropy loss is differentiable and helps in optimizing the model using gradient descent-based algorithms.

2. **Hinge Loss (Support Vector Machine Loss):** Hinge loss is often used in binary classification tasks, particularly in Support Vector Machine (SVM) models. It aims to maximize the margin between the decision boundary and the training samples. The loss function penalizes the model based on the distance between the predicted class scores and the true labels, but only if the predicted score is not already greater than a predefined margin. This loss function is suitable for problems where the focus is on correctly classifying the samples while maximizing the margin between the classes.

b. Regression loss functions

1. **Mean Squared Error (MSE):** MSE is a commonly used loss function in regression tasks. It measures the average squared difference between the predicted and true values. The loss is calculated by taking the mean of the squared differences across all samples. MSE is sensitive to outliers as it squares the errors, making it more suitable for problems where outliers need to be penalized heavily. It is differentiable and widely used due to its simplicity and effectiveness.

2. **Mean Absolute Error (MAE):** MAE is another widely used loss function in regression tasks. It measures the average absolute difference between the predicted and true values. The loss is calculated by taking the mean of the absolute differences across all samples. MAE is less sensitive to outliers compared to MSE since it does not square the errors. It provides a more robust measure of the average prediction error but may not work as well if the problem requires emphasizing larger errors.

B. Batch normalization

Batch Normalization is a technique used in deep neural networks to improve the training process and stability of the model. It addresses the problem of internal covariate shift, which refers to the change in the distribution of layer inputs during training. The Batch Normalization layer aims to normalize the inputs of each layer by adjusting and scaling them.

The function of the Batch Normalization layer can be summarized in the following steps. First, for each mini-batch during training, the Batch Normalization layer normalizes the inputs by subtracting the mini-batch mean and dividing by the mini-batch standard deviation. This step ensures that the inputs have zero mean and unit variance, which helps stabilize the training process. Then, the layer applies a linear transformation to the normalized values, introducing two learnable parameters, often referred to as gamma and beta, which allow the network to adapt and scale the normalized inputs. During training, the Batch Normalization layer maintains running estimates of the mini-batch mean and standard deviation. These estimates are updated using exponential moving averages. During inference, the learned parameters and running statistics are used to normalize the inputs.

The importance of the Batch Normalization layer in deep neural networks can be understood through its benefits. Firstly, it improves training speed by stabilizing the gradients flowing through the network, helping the network converge faster and making it less sensitive to the choice of learning rate. This makes training deep networks more efficient. Secondly, Batch Normalization reduces internal covariate shift, which is the change in the distribution of layer inputs. This reduces the need for careful initialization of weights and the use of small learning rates. It allows the subsequent layers to learn more independently, promoting better gradient flow and network performance. Furthermore, the Batch Normalization layer acts as a regularization technique by introducing slight noise to the inputs due to the normalization process. This noise helps reduce the generalization error of the model and prevents overfitting.

Lastly, Batch Normalization makes deep neural networks more robust to changes in input distributions. By normalizing the inputs to have zero mean and unit variance, it helps the model generalize well to different datasets and makes it more resilient to variations in lighting, contrast, or other factors.

C. Group Normalization and Layer Normalization

a. Group normalization

Group Normalization (GN) is a normalization technique used in deep neural networks, primarily for convolutional neural networks (CNNs). It aims to improve training and performance by reducing the dependence on batch statistics and providing normalization within smaller groups of channels.

In GN, the channels of a convolutional layer are divided into groups. The number of groups is a hyperparameter that can be set based on the network architecture or experimentation. Within each group, the mean and variance of the feature activations are computed independently for each sample in the batch. These statistics are then used to normalize the activations of each channel. After normalization, each channel's activations are scaled and shifted by learned parameters (gamma and beta) to allow the network to learn the optimal representation. By normalizing within smaller groups, GN reduces the dependence on batch statistics. This is particularly useful when dealing with non-i.i.d. data or scenarios with small batch sizes, such as video frames or examples from different domains. GN has been shown to improve the generalization of deep networks, especially in scenarios with limited data or smaller batch sizes. It helps mitigate the effect of batch-to-batch variations and improves the network's ability to learn meaningful representations. Moreover, GN can be computationally efficient compared to Batch Normalization, as it reduces the memory footprint and computational overhead by operating within smaller groups.

In summary, Group Normalization provides an effective alternative to other normalization techniques like Batch Normalization and Layer Normalization. It offers improved performance and generalization in deep neural networks, particularly for scenarios with limited data or small batch sizes. By normalizing within smaller groups, GN reduces the dependence on batch statistics and enhances the network's ability to learn meaningful representations.

b. Layer normalization

Layer normalization is a technique used in deep neural networks to address the issue of internal covariate shift. It aims to normalize the inputs within a layer, stabilizing the learning process and improving overall network performance. While similar to batch normalization, layer normalization operates along the feature dimension rather than the batch dimension.

To understand the function of layer normalization, consider a given layer in the network with a matrix of inputs. Each row of the matrix represents a training example, and each column represents a feature. Layer normalization begins by computing the mean and variance of the inputs along the feature dimension. The mean is obtained by averaging the values across the features for each training example, while the variance is calculated by computing the squared differences from the mean and averaging them. The next step is to normalize the inputs using the computed mean and variance. This is done by subtracting the mean from each element of the input matrix and dividing it by the square root of the variance. These operations result in a normalized matrix, where the inputs for each training example are independent of the scale and distribution of other examples in the layer. After normalization, the normalized inputs are scaled and shifted using learnable parameters called gamma and beta. These parameters allow the network to learn the optimal scaling and shifting factors for the normalized inputs, enabling the preservation of the network's representational capacity. The final output of the layer normalization operation is formed by the scaled and shifted normalized inputs, which are then passed to the next layer in the network. By normalizing the inputs within each layer, layer normalization reduces the dependence of the network on the order of training examples within a batch and mitigates the impact of covariate shift during training. This technique also aids in generalization to new data.

One advantage of layer normalization is its applicability to different types of neural network architectures. It can be applied independently to each layer, making it suitable for recurrent neural networks (RNNs) and other architectures where batch normalization may not be well-suited. Overall, layer normalization plays a vital role in improving the stability and effectiveness of deep networks by normalizing inputs within each layer and reducing the effects of internal covariate shift.

c. Differences with batch normalization

Layer normalization, group normalization, and batch normalization are three different techniques used in deep neural networks to normalize the activations of the network's layers. While all of them aim to improve the training and generalization of the network, they differ in their normalization strategies.

The main differences between layer normalization, group normalization, and batch normalization lie in the dimensions over which normalization is performed. Batch normalization operates across the batch dimension, layer normalization operates across the feature dimension, and group normalization operates across groups of channels. The choice of which normalization technique to use depends on factors such as the network architecture, the availability of batch statistics, and the size of the batch being processed.

2. Classification of traffic signs

One of the important applications of neural networks is to help the automatic driving system (auto pilot) in self-driving cars through the recognition of traffic signs, which enables the system to react appropriately when faced with different signs. In this section, we want to recognize traffic signs by designing two types of neural networks and training them with the help of existing traffic data.

Table 1. Libraries used in section 2.

No.	Library title
1	pandas
2	numpy
3	matplotlib
4	scikit-learn
5	opencv
6	tensorflow
7	PIL
8	os
9	keras
10	random

Where:

1. Pandas: A powerful data manipulation and analysis library for handling structured data in Python.
2. Numpy: A fundamental package for scientific computing that provides efficient numerical operations on multi-dimensional arrays.
3. Matplotlib: A versatile plotting library for creating static, animated, and interactive visualizations in Python.
4. Scikit-learn: A machine learning library that provides a wide range of tools for data preprocessing, model selection, and evaluation.
5. OpenCV: Open Source Computer Vision Library, which offers extensive computer vision functionality for tasks like image processing, object detection, and video analysis.

6. Tensorflow: An open-source deep learning framework that enables the construction and training of artificial neural networks for various machine learning tasks.
7. PIL: The Python Imaging Library, a library for opening, manipulating, and saving many different image file formats.
8. OS: A module that provides a portable way of using operating system-dependent functionalities, such as interacting with the file system and executing system commands.
9. Keras: A high-level neural networks API that is built on top of TensorFlow and simplifies the process of building and training deep learning models.
10. Random: A module that generates pseudo-random numbers for various purposes, including simulations, games, and random sampling.

A. Data preprocessing

First of all, we use the following code that assigns the file path "C:/Users/user/Desktop/German Traffic Sign Recognition Benchmark" to the variable My_path. This can be useful for referencing or manipulating files within that directory in subsequent code.

```
My_path = r'C:/Users/user/Desktop/German Traffic Sign Recognition Benchmark'
```

Then, we use the following code and set 43 as the number of classes. The code processes a dataset that contains images organized into different classes. It iterates over each class, reads the images in that class, resizes them to a fixed size of 30x30 pixels, converts them into NumPy arrays, and appends them to the data list along with their corresponding labels. The data list will eventually contain pairs of images and labels, which can be further used for training or analysis purposes.

```
data = []

labels = []

classes = 43

# Configuration of images
for i in range(classes):
    path = os.path.join(My_path,'train',str(i))
    images = os.listdir(path)

    for a in images:
            image = Image.open(path + '\\'+ a)
            image = image.resize((30,30)) # Resize the images
            image = np.array(image)
            data.append([image,i]) # Append the values together
```

The following code shuffles the data list randomly and then prints the number of elements in the shuffled list.

```
random.shuffle(data)
print(len(data))
```

The output of the code print(len(data)) is 39209, it means that the length of the data list is 39209. This indicates that there are 39,209 elements or samples in the data list. In general, it suggests that we have 39,209 samples or data points in our dataset, which could be used for training, testing, or any other data processing tasks in our program.

The following code initializes empty lists x and y. It then iterates over the data list, extracting the features and labels from each item. The features are appended to the x list, while the labels are appended to the y list. This code separates the features and labels from the data list and stores them in separate lists for further use.

```
x = []
y = []

for features,label in data:
    x.append(features)
    y.append(label)
```

The following code converts the Python lists x and y into NumPy arrays. The NumPy library provides powerful array manipulation capabilities in Python. The code uses the np.array() function from the NumPy module to convert the x and y lists into NumPy arrays.

```
#Converting lists into numpy arrays
x = np.array(x)
y = np.array(y)
```

After this conversion, the variables x and y will be NumPy arrays instead of Python lists. This allows us to perform efficient and convenient operations on the data using NumPy's array-based functions and methods. NumPy arrays provide various mathematical and statistical operations, indexing and slicing capabilities, and compatibility with other scientific computing libraries in Python.

Then, to split the data in the "Train" folder, we use the following code:

```
# Splitting the data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(x, y, test_size =
0.1, random_state=42)

print("Train set shape:", X_train.shape)
print("Validation set shape:", X_val.shape)
```

This code splits the data into training and validation sets using the train_test_split() function. It takes the input features x and corresponding labels y and divides them into two sets: X_train and y_train for training, and X_val and y_val for validation. The split is performed with a validation size of 10% (specified by test_size = 0.1) and a random state of 42 to ensure reproducibility. The code then prints the shapes of the training and validation sets, indicating the number of samples and the dimensionality of the features in each set.

The output shows that the train set shape is (35288, 30, 30, 3) and this means that the training set has 35,288 samples. Each sample has a shape of (30, 30, 3), representing a 30x30 image with 3 color channels (RGB). Validation set shape is (3921, 30, 30, 3) and this indicates that the validation set contains 3,921 samples. Each sample also has a shape of (30, 30, 3), representing a 30x30 image with 3 color channels.

The following code performs pixel normalization on the training and validation images and prints the shapes of the training images and labels. By dividing each pixel value by 255.0, the code scales the pixel values from the original range of 0 to 255 to a new range of 0.0 to 1.0. This normalization step is commonly applied in image processing and machine learning tasks to ensure that the pixel values are within a consistent and manageable range, which can improve the learning process and convergence of the model.

```
X_train = X_train/255.0
X_val = X_val/255.0

print("Shape of train images is:", X_train.shape)
print("Shape of labels is:", y_train.shape)
```

After the normalization, the code prints the shape of the training images, providing insights into the size and structure of the training image dataset. The shape of X_train represents the number of training samples, the height and width of each image, and the number of color channels (if applicable). This information helps to understand the dimensions of the training image data. Similarly, the code also prints the shape of the training labels, which corresponds to the dimensions of the label dataset for the training images. The shape of y_train indicates the number of training samples and the dimensionality of the label data. This output provides information about the size and structure of the training label dataset.

B. Implementation

a. MLP model

i. MLP architecture

For MLP architecture, we use the following code that defines a Multi-Layer Perceptron (MLP) model using the Keras Sequential API. The properties used for MLP model are represented in Table 2. As can be seen through the Table 2, we used a "sparse_categorical_crossentropy" loss function, and a "softmax" activation function for the output layer. The reasons for choosing these functions are given in the following.

Table 2. MLP model properties

Property	Value
Activation function	Relu
Learning rate	0.001
Optimizer	Adam
Clipvalue	1.0
Loss function	sparse_categorical_crossentropy
Output layer activation	Softmax

```
# Define the MLP model
model = Sequential()
model.add(Flatten(input_shape=X_train.shape[1:]))  # Flatten the input
shape excluding the batch dimension

# Add hidden layers
model.add(Dense(256, activation='relu'))  # Example hidden layer with
256 neurons
model.add(Dense(128, activation='relu'))  # Another hidden layer with
128 neurons

# Add output layer
model.add(Dense(classes, activation='softmax'))  # Output layer with
'classes' neurons for classification

# Compile the model with specified learning rate
learning_rate= 0.001
optimizer = keras.optimizers.Adam(learning_rate = learning_rate,
clipvalue = 1.0)
model.compile(optimizer=optimizer,
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Our model consists of a Flatten layer to reshape the input data, followed by multiple dense layers with ReLU activation to introduce non-linearity. The output layer uses softmax activation for classification. The model is compiled with the Adam optimizer, a specified learning rate of 0.001, and 'sparse_categorical_crossentropy' loss for multi-class classification. Overall, this code constructs an MLP model architecture and prepares it for training and evaluation on the given classification task.

Loss function:

When utilizing an MLP neural network architecture for traffic sign classification, the selection of an appropriate loss function is crucial. In this context, the sparse_categorical_crossentropy loss function is often preferred for several reasons. Firstly, traffic sign classification involves categorizing input images into multiple classes, where each image corresponds to a specific traffic sign type. The sparse_categorical_crossentropy loss function is specifically designed for multi-class classification tasks, enabling the model to predict a single class label from a range of mutually exclusive classes. Moreover, traffic sign classification commonly exhibits sparse label distributions due to the large number of possible classes. The sparse_categorical_crossentropy loss function effectively handles this sparsity by internally converting integer labels to a one-hot encoded format. This eliminates the need for explicit one-hot encoding, simplifying the training process and avoiding memory and computational inefficiencies associated with large-scale one-hot encoded representations. Furthermore, the use of the sparse_categorical_crossentropy loss function contributes to computational efficiency within the MLP architecture. By operating with integer labels directly, the loss function conserves computational resources compared to the use of one-hot encoded representations. This reduction in computational complexity and memory usage enhances training and inference efficiency, enabling faster model convergence and improved overall performance. Lastly, the sparse_categorical_crossentropy loss function offers simplicity in implementation within the MLP neural network architecture. Its direct support for integer labels eliminates the need for additional preprocessing steps or explicit handling of one-hot encoded vectors. This streamlines the coding process, reducing the complexity and potential sources of error in the implementation.

Overall, the sparse_categorical_crossentropy loss function is a suitable choice for traffic sign classification within an MLP neural network architecture. Its compatibility with multi-class classification tasks, ability to handle sparse label distributions, computational efficiency, and ease of implementation make it well-suited for achieving accurate and efficient traffic sign classification results.

Activation function of output layer:

Using a softmax activation function in the output layer of a deep MLP neural network architecture for traffic sign classification offers several advantages. The softmax activation function is commonly employed in the output layer of a deep MLP neural network for classification tasks, including traffic sign classification. One key reason is its ability to produce a probability distribution over the multiple classes. In traffic sign classification, each input image belongs to a specific traffic sign type, and the goal is to assign a probability to each class to determine the most likely class for a given image. The softmax activation function ensures that the outputs of the output layer sum up to 1, representing the probabilities of different traffic sign classes. This allows for clear interpretation and decision-making based on the predicted probabilities. Moreover, the softmax function's property of normalization enhances the model's ability to handle multi-class scenarios. It prevents one class from dominating over others by redistributing the probabilities among the classes. This is particularly beneficial in traffic sign classification, where there may be imbalanced class distributions or instances where multiple traffic signs appear in an image. The softmax activation helps the model assign appropriate probabilities to each class, capturing the relative importance of different traffic sign types.

Another advantage of the softmax activation function is its differentiability, which is crucial for effective backpropagation during training. The softmax function smoothly maps the inputs to probabilities, allowing gradient-based optimization algorithms to efficiently adjust the model's weights and biases based on the error signal. This differentiability property facilitates convergence and helps the model learn meaningful representations and decision boundaries for accurate traffic sign classification.

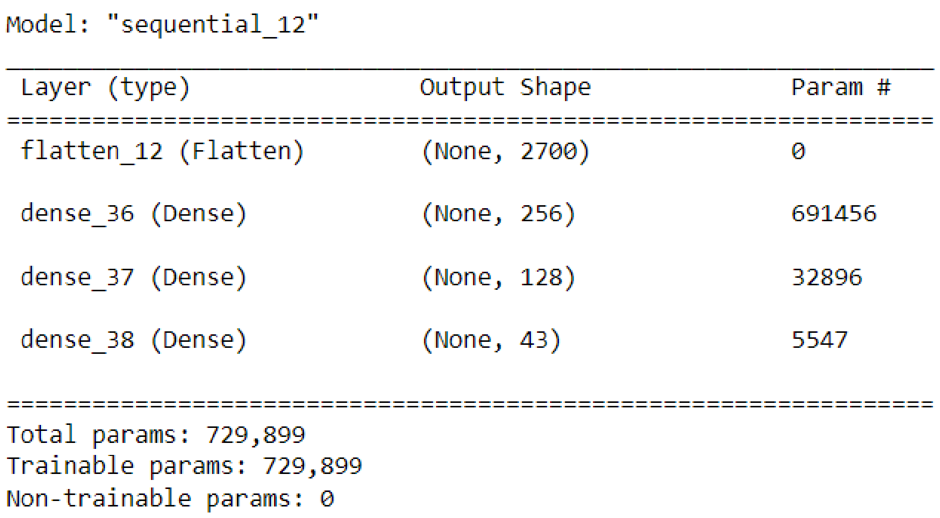
Furthermore, using softmax in the output layer simplifies the implementation and interpretability of the model. It provides a natural and intuitive output format, where the highest probability corresponds to the predicted class. This simplifies the post-processing steps, such as determining the final predicted class based on the probabilities produced by the model.

Overall, incorporating the softmax activation function in the output layer of a deep MLP neural network architecture for traffic sign classification enables the model to generate probability distributions, handle multi-class scenarios, ensure differentiability for effective training, and simplify model interpretation and decision-making processes. We use the following codes to show the model summary which is given in Table 3 and MLP model architecture which is shown in Figure 1.

```
# Print the model summary
model.summary()

# The model visualization
keras.utils.plot_model(model)
```

Table 3. MLP model summary



```
Model: "sequential_12"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_12 (Flatten)        (None, 2700)              0

 dense_36 (Dense)            (None, 256)               691456

 dense_37 (Dense)            (None, 128)               32896

 dense_38 (Dense)            (None, 43)                5547

=================================================================
Total params: 729,899
Trainable params: 729,899
Non-trainable params: 0
```

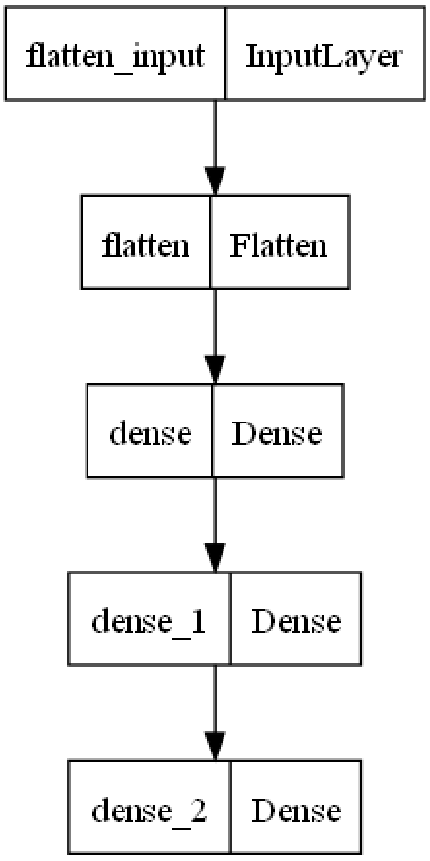


Figure 1. MLP neural network architecture

ii. MLP training

The following provided code trains the defined model using the training data. In this code snippet, the batch size and the number of epochs are specified as batch_size = 32 and epochs = 20, respectively. The batch size determines the number of samples that will be processed at each iteration during training, while the number of epochs defines how many times the entire training dataset will be passed through the model.

```
# Train the model
batch_size = 32
epochs = 20

history = model.fit(X_train, y_train, batch_size=batch_size,
epochs=epochs, validation_data=(X_val, y_val))
```

The model.fit() function is then called, which initiates the training process. It takes the training data (X_train and y_train) as input and includes additional parameters such as the batch size (batch_size), the number of epochs (epochs), and the validation data (X_val and y_val). During training, the model will be optimized using the defined loss function and the specified optimizer. Throughout the training process, the model will iterate over the training data for the specified number of epochs, gradually updating its parameters to minimize the loss function. After each epoch, the model's performance on unseen data will be evaluated using the provided validation data. The training progress and performance metrics, such as loss and accuracy, will be recorded and stored in the history object.

Executing this code will train the model on the training data, enabling it to learn and improve its predictive capabilities. The resulting history object can be used for further analysis and visualization of the training progress and performance metrics, helping to assess the model's training behavior and performance.

Learning rate:

A learning rate of 0.001 is smaller compared to 0.01. When training a neural network, the learning rate determines the step size at which the model's parameters are updated during the optimization process. A higher learning rate such as 0.01 can cause the updates to be too large, leading to overshooting the optimal parameter values. This can result in the model converging slowly or even diverging, ultimately leading to worse accuracy.

It's important to note that the choice of an optimal learning rate depends on the specific dataset and model architecture. While a learning rate of 0.01 resulted in worse accuracy in this case, it doesn't imply that a learning rate of 0.001 is universally optimal.

An inappropriate learning rate can have a significant impact on the learning process of our neural network. Here are the effects it can have:

- **Difficulty finding the global optimum:** The learning rate influences the exploration of the parameter space during training. If the learning rate is too high or too low, the model may have difficulty navigating the parameter space effectively. It may get trapped in local optima or struggle to converge to the global optimum, limiting its ability to capture complex patterns and achieve high accuracy.
- **Overshooting or divergence:** On the other hand, if the learning rate is too large, the updates to the parameters can be excessively large. This can cause the learning process to overshoot the optimal parameter values, leading to oscillations or divergence. The model's accuracy may fluctuate dramatically, and it may fail to converge to a stable solution.
- **Slow convergence or failure to converge:** If the learning rate is too small, the updates to the model's parameters will be too conservative and slow. As a result, the learning process may take a long time to converge, or in some cases, it may fail to converge altogether. The model's accuracy may stagnate, and it may struggle to find an optimal solution within a reasonable number of training iterations.
- **Gradient-related issues:** Inappropriately high learning rates can lead to exploding gradients, where the gradient values become excessively large. This can cause numerical instability, making it challenging to compute accurate updates to the parameters. Conversely, very low learning rates can result in vanishing gradients, where the gradients become extremely small. This can hinder the flow of useful gradient information through the network, impeding effective parameter updates and learning.

- **Unstable or erratic training:** Inappropriate learning rates can lead to unstable or erratic training behavior. The model's performance metrics, such as loss and accuracy, may exhibit inconsistent patterns, with frequent jumps or fluctuations. The model may struggle to learn meaningful representations from the data, resulting in poor generalization and suboptimal performance on unseen examples.

After the training process, we will use the following commands to plot the loss and accuracy of the training and validation data.

```python
# Plot the loss
plt.figure(figsize=(8, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plot the accuracy
plt.figure(figsize=(8, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

The results of above codes are given in Figures 2 and 3.

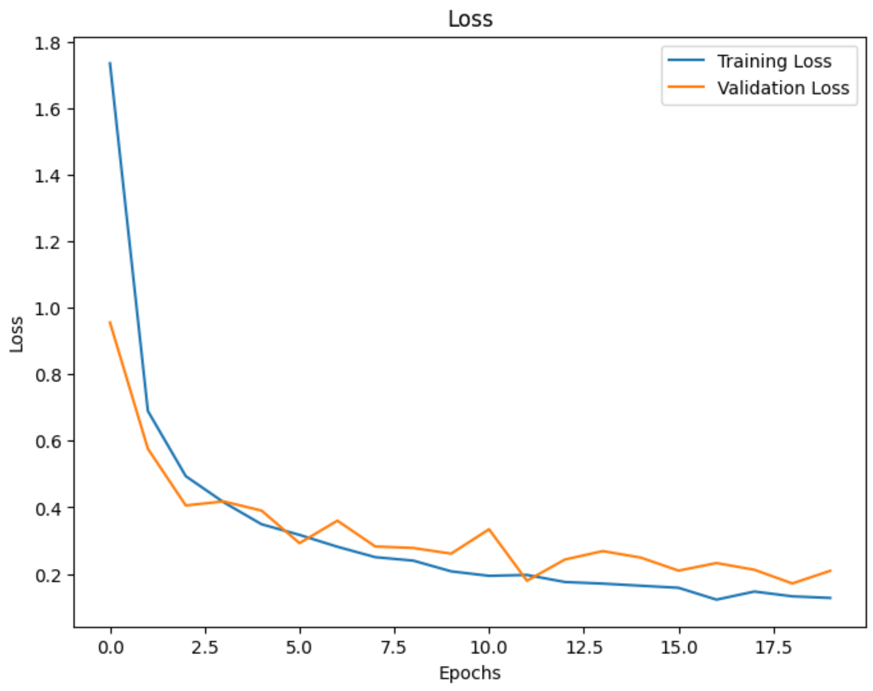


Figure 2. Loss of MLP neural network

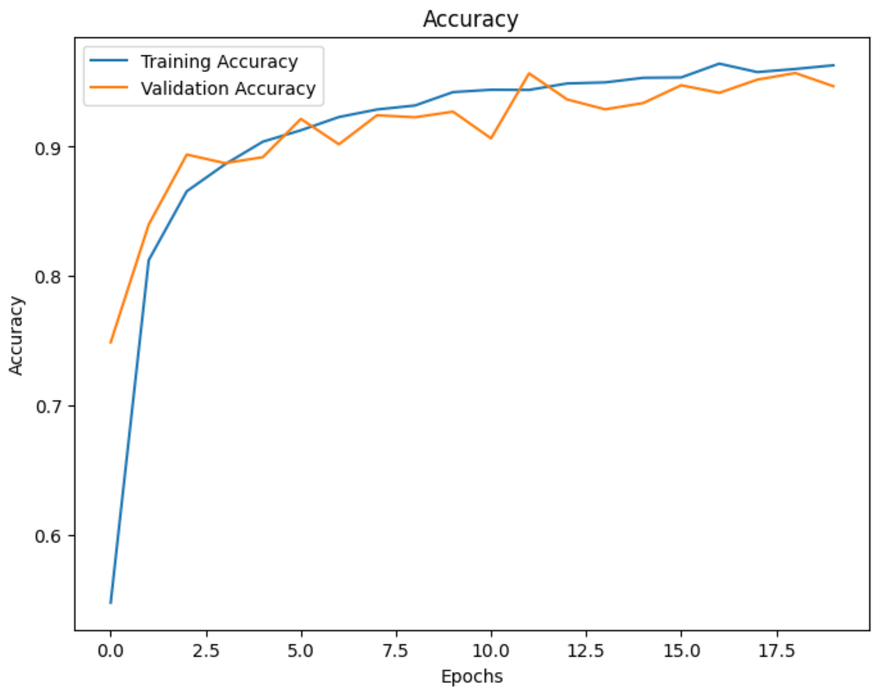


Figure 3. Accuracy of MLP neural network

For evaluating the performance of the model on the testing data, first, we use the following command to read the "Test.csv" file.

```
# Load test dataset
y_test = pd.read_csv('C:/Users/user/Desktop/German Traffic Sign
Recognition Benchmark/Test.csv')
```

The following code is used to prepare the test data for classification in the German Traffic Sign Recognition Benchmark.

```
# Add path prefix to "Path" column
y_test["Path"] = 'C:/Users/user/Desktop/German Traffic Sign Recognition
Benchmark/' + y_test["Path"]

labels = y_test["ClassId"].values
imgs = y_test["Path"].values

data = []

# Load and preprocess images
for img in imgs:
    image = Image.open(img)
    image = image.resize((30, 30))
    data.append(np.array(image) / 255.0)  # Divide each pixel by 255.0

X_test = np.array(data)
```

It begins by adding a path prefix to the "Path" column in the y_test DataFrame. This is achieved by concatenating the specified path prefix 'C:/Users/user/Desktop/German Traffic Sign Recognition Benchmark/' with the values in the "Path" column. This step ensures that the correct file paths are associated with the test images. Next, the code extracts the values from the "ClassId" column in the y_test DataFrame and assigns them to the labels variable. These values represent the class labels associated with the test images. Similarly, the code extracts the values from the "Path" column in the y_test DataFrame and assigns them to the imgs variable, representing the file paths of the test images. To store the preprocessed image data, the code initializes an empty list called data. Then, it iterates over each image path (img) in imgs. For each image, the following steps are performed: the image is opened using the Image.open() function from the Python Imaging Library (PIL). It is then resized to a fixed size of 30x30 pixels using the resize() method, ensuring consistency in the input image dimensions. Finally, the resized image is converted into a NumPy array using np.array(image), and each pixel value is divided by 255.0 to normalize the pixel values between 0.0 and 1.0. The preprocessed image array is then appended to the data list.

After iterating over all the test images, the data list contains the preprocessed image data. The code converts the data list into a NumPy array using np.array(data), resulting in the X_test variable, which represents the input data for the test images in the ML model.

The following codes are used to make predictions on the test and validation data using a trained model and calculate the accuracy and loss metrics.

```
# Make predictions for test data
test_pred = np.argmax(model.predict(X_test), axis=-1)

# Get predictions for validation data
val_pred = np.argmax(model.predict(X_val), axis=-1)

# Calculate accuracy for test
accuracy_test_pred = accuracy_score(labels, test_pred)
print("Accuracy on the test dataset:", accuracy_test_pred)

# Calculate accuracy for validation
accuracy_val_pred = accuracy_score(y_val, val_pred)
print("Accuracy on the validation dataset:", accuracy_val_pred)

# Calculate loss for test
loss_test = model.evaluate(X_test, labels)
print("Loss on the test dataset:", loss_test)

# Calculate loss for validation
loss_val = model.evaluate(X_val, y_val)
print("Loss on the validation dataset:", loss_val)
```

First, test_pred = np.argmax(model.predict(X_test), axis=-1) is executed to obtain the predicted class labels for the test data. The model.predict() function generates the predicted probabilities for each class for each test sample. Then, np.argmax() is applied to obtain the index of the class with the highest probability for each test sample. The resulting test_pred variable contains the predicted class labels for the test data. Similarly, val_pred = np.argmax(model.predict(X_val), axis=-1) is used to obtain the predicted class labels for the validation data in the same manner. The predicted class labels for the validation data are stored in the val_pred variable. To evaluate the accuracy of the test and validation predictions, accuracy_test_pred = accuracy_score(labels, test_pred) calculates the accuracy of the test predictions by comparing the predicted labels (test_pred) with the actual labels (labels). This is done using the accuracy_score() function from scikit-learn, which computes the accuracy metric. The resulting accuracy value is stored in the accuracy_test_pred variable. The accuracy of the test predictions is then printed using print("Accuracy on the test dataset:", accuracy_test_pred). Similarly, the accuracy of the validation predictions is calculated with accuracy_val_pred = accuracy_score(y_val, val_pred), comparing the predicted labels for the validation data (val_pred) with the actual labels (y_val). The accuracy metric is computed and stored in the accuracy_val_pred variable. The accuracy of the validation predictions is printed using print("Accuracy on the validation dataset:", accuracy_val_pred).

To assess the loss (error) of the model on the test and validation data, the following evaluations are performed: loss_test = model.evaluate(X_test, labels) calculates the loss on the test data by evaluating the model's performance using the test data (X_test) and the corresponding true labels (labels). The evaluate() function of the model returns the loss value, which is stored in the loss_test variable. The loss value on the test dataset is printed using print("Loss on the test dataset:", loss_test). Similarly, loss_val = model.evaluate(X_val, y_val) calculates the loss on the validation data in a similar manner. The model's performance is evaluated using the validation data (X_val) and the corresponding true labels (y_val), and the loss value is stored in the loss_val variable. The loss value on the validation dataset is printed using print("Loss on the validation dataset:", loss_val).

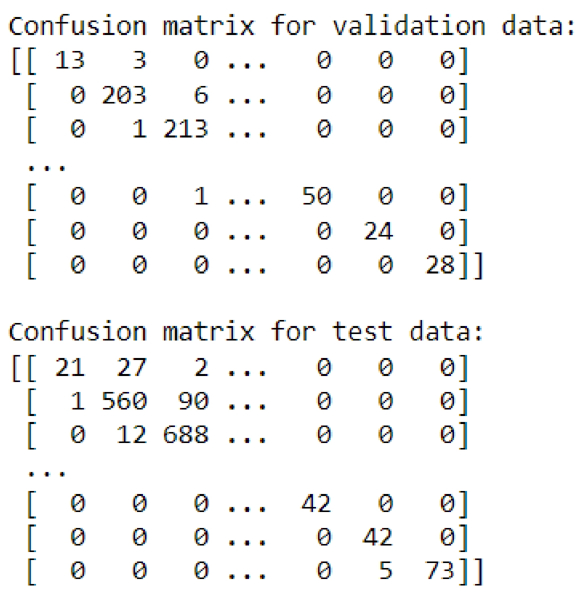
And finally, we will use the following codes to create and print the confusion matrices for validation and test data. The confusion matrices are shown in Figure 4.

```
# Create confusion matrix for validation data
val_cm = confusion_matrix(y_val, val_pred)

# Create confusion matrix for test data
test_cm = confusion_matrix(labels, test_pred)

print("Confusion matrix for validation data:")
print(val_cm)
print()

print("Confusion matrix for test data:")
print(test_cm)
```



```
Confusion matrix for validation data:
[[ 13   3   0 ...   0   0   0]
 [  0 203   6 ...   0   0   0]
 [  0   1 213 ...   0   0   0]
 ...
 [  0   0   1 ...  50   0   0]
 [  0   0   0 ...   0  24   0]
 [  0   0   0 ...   0   0  28]]

Confusion matrix for test data:
[[ 21  27   2 ...   0   0   0]
 [  1 560  90 ...   0   0   0]
 [  0  12 688 ...   0   0   0]
 ...
 [  0   0   0 ...  42   0   0]
 [  0   0   0 ...   0  42   0]
 [  0   0   0 ...   0   5  73]]
```

Figure 4. Confusion matrix of MLP neural network

The accuracy and loss vales for training, testing and validation data for the MLP neural network model are given in Table 4.

Table 4. Results of MLP model training

Dataset	Accuracy	Loss
Train	0.9629	0.1280
Test	0.8345	1.1298
Validation	0.9466	0.2094

The MLP deep neural network model achieved a high accuracy of 96.29% on the training dataset, indicating that it performed well in correctly classifying the traffic signs within the training data. The corresponding loss value on the training dataset was 0.1280, which suggests that the model had a relatively low level of error during training. On the test dataset, the model achieved an accuracy of 83.45%. Although this accuracy is lower than that on the training dataset, it still indicates a reasonable performance in classifying traffic signs within previously unseen data. The loss value on the test dataset was 1.1298, which is higher compared to the loss on the training dataset. This indicates that the model experienced more error when predicting the class labels for the test data.

For the validation dataset, the MLP model achieved an accuracy of 94.66%. This indicates a strong performance on the validation data, which serves as an additional evaluation set to assess the model's generalization ability. The loss value on the validation dataset was 0.2094, which is relatively low, suggesting that the model performed well in minimizing the error on the validation data. Overall, these performance results indicate that the MLP deep neural network model achieved a high level of accuracy on the training and validation datasets. However, it had a slightly lower accuracy on the test dataset, indicating a potential for overfitting or difficulty in generalizing to unseen data. The higher loss value on the test dataset further supports this observation. Therefore, it may be beneficial to further optimize the model or explore other techniques to improve its performance on unseen data.

b. CNN model

i. CNN architecture

The following code is used to build a deep convolutional neural network (CNN) architecture for the purpose of classifying traffic signs. The Sequential() function is used to initialize a sequential model, which represents a linear stack of layers. In this section, we have used the 'sparse_categorical_crossentropy' loss function. Also, we will use the 'softmax' activation function for the output layer. The reason for these choices are given in the following.

```
# Define the model
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(30, 30,
3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(classes, activation='softmax'))

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

# Print the model summary
model.summary()
```

The first layer added to the model is a 2D convolutional layer (Conv2D) with 32 filters of size 3x3. This layer applies the 'relu' activation function to introduce non-linearity. The input shape of the layer is specified as (30, 30, 3), indicating that the input images are expected to have a height and width of 30 pixels and 3 color channels (RGB). A max pooling layer (MaxPooling2D) is added next, which reduces the spatial dimensions of the previous layer's output by taking the maximum value within each pooling region. A pooling window of size 2x2 is used. Another 2D convolutional layer is added with 64 filters of size 3x3, followed by another max pooling layer with the same configuration as before. These layers help the model learn hierarchical features from the input images.

To prepare the data for passing it through fully connected layers, the output from the previous layer is flattened using the Flatten() layer, which converts the output into a one-dimensional vector. Two fully connected layers (Dense) are then added to the model. The first dense layer has 128 neurons and applies the 'relu' activation function. The second dense layer, which serves as the output layer, has a number of neurons determined by the variable classes, representing the number of different traffic sign labels. The 'softmax' activation function is used in the output layer to generate probability distributions over the classes, allowing the model to output the predicted class probabilities. The model is compiled using the compile() function, where the 'adam' optimizer is specified, which is an efficient variant of stochastic gradient descent. The 'sparse_categorical_crossentropy' loss function is chosen as it is suitable for multi-class classification problems with integer-encoded class labels. The model's performance during training will be evaluated based on accuracy.

Finally, the model.summary() function is used to print a summary of the model's architecture. This summary provides information about the types of layers in the model, the output shapes of each layer, and the number of trainable parameters. It gives an overview of the model's structure and the number of parameters that will be learned during training.

Then, we use the following commands:

```
# The model visualization
keras.utils.plot_model(model)
```

The code keras.utils.plot_model(model) is used to generate a visual representation of the deep convolutional neural network (CNN) model. It utilizes the plot_model function from the Keras utilities module. By calling this function and passing the model as an argument, a graphical visualization of the model's architecture is created. The visualization provides a visual representation of the model's layers, their connections, and the flow of data through the network.

Loss function:

The sparse categorical cross-entropy loss function is commonly used for classification tasks with multiple classes, such as the classification of traffic signs in a deep CNN model. It is suitable for this purpose due to several reasons. First, in traffic sign classification, the class labels are often represented as integer values, where each value corresponds to a specific traffic sign category. The sparse categorical cross-entropy loss function is designed to handle such integer-encoded labels efficiently. Furthermore, the traffic sign classification task involves assigning a single label to each input image, indicating the predicted traffic sign category. The sparse categorical cross-entropy loss function is appropriate for this single-label classification scenario, as it penalizes the model based on the discrepancy between the predicted class probabilities and the true class label. It ensures that the model focuses on correctly classifying each input image into a single traffic sign category.

To utilize the sparse categorical cross-entropy loss function effectively, the output layer of the deep CNN model is typically equipped with the softmax activation function. This activation function generates a probability distribution over the classes, indicating the likelihood of each class being the correct classification for a given input image. The sparse categorical cross-entropy loss function expects this probability distribution as input, allowing the model to optimize its parameters based on the discrepancy between the predicted probabilities and the true class labels. In terms of computational efficiency, the sparse categorical cross-entropy loss function offers advantages over other loss functions, such as categorical cross-entropy. It eliminates the need to convert the integer-encoded labels into one-hot encoded vectors, which can be memory-intensive and computationally expensive for datasets with numerous classes. By avoiding this conversion step, the sparse categorical cross-entropy loss function enables more efficient computation during the training process.

By utilizing the sparse categorical cross-entropy loss function in the deep CNN model for traffic sign classification, the model can effectively learn to differentiate and classify the various traffic sign categories based on the input images. The loss function guides the model to optimize its parameters and improve its accuracy by minimizing the discrepancy between the predicted class probabilities and the true class labels. This choice of loss function is essential for achieving accurate and reliable traffic sign classification results.

Activation function of output layer:

The softmax activation function is used in the output layer of deep CNN models for traffic sign classification due to its several advantages. It transforms the model's outputs into a probability distribution over the different classes, allowing for confident predictions. The softmax function introduces non-linearity, enabling better discrimination between traffic sign categories. Its differentiability facilitates efficient backpropagation during training, leading to improved parameter optimization. Additionally, the softmax function provides interpretable predictions as class probabilities, aiding in decision-making. Overall, the softmax activation function enhances the model's ability to accurately classify traffic signs based on input images.

ii. CNN training

The following code is used to train our deep CNN model.

```
# Train the model
history = model.fit(X_train, y_train, batch_size=32, epochs=10,
validation_data=(X_val, y_val))
```

The fit() function is called on the model with the training data (X_train and y_train) as input. The batch_size parameter determines the number of samples processed in each training iteration, and the epochs parameter specifies the number of times the entire training dataset is iterated. The validation_data parameter is used to evaluate the model's performance on the validation set during training. The model is trained by optimizing its parameters to minimize the specified loss function and improve its accuracy. The training process updates the model's weights based on the gradients calculated through backpropagation. The fit() function returns a history object that contains information about the training process, including loss and accuracy values for both the training and validation datasets.

After the training process, we will use the following commands to plot the loss and accuracy of the training and validation data for CNN model.

```
# Plot the loss
plt.figure(figsize=(8, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss for model')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plot the accuracy
plt.figure(figsize=(8, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy for model')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Just as we did for MLP model, for evaluating the performance of the CNN model on the testing data, first, we use the following command to read the "Test.csv" file.

```
# Load test dataset
y_test = pd.read_csv('C:/Users/user/Desktop/German Traffic Sign
Recognition Benchmark/Test.csv')
```

The following code is used to prepare the test data for classification in the German Traffic Sign Recognition Benchmark.

```
# Add path prefix to "Path" column
y_test["Path"] = 'C:/Users/user/Desktop/German Traffic Sign Recognition
Benchmark/' + y_test["Path"]

labels = y_test["ClassId"].values
imgs = y_test["Path"].values

data = []

# Load and preprocess images
for img in imgs:
    image = Image.open(img)
    image = image.resize((30, 30))
    data.append(np.array(image) / 255.0)  # Divide each pixel by 255.0

X_test = np.array(data)
```

It begins by adding a path prefix to the "Path" column in the y_test DataFrame. This is achieved by concatenating the specified path prefix 'C:/Users/user/Desktop/German Traffic Sign Recognition Benchmark/' with the values in the "Path" column. This step ensures that the correct file paths are associated with the test images. Next, the code extracts the values from the "ClassId" column in the y_test DataFrame and assigns them to the labels variable. These values represent the class labels associated with the test images. Similarly, the code extracts the values from the "Path" column in the y_test DataFrame and assigns them to the imgs variable, representing the file paths of the test images. To store the preprocessed image data, the code initializes an empty list called data. Then, it iterates over each image path (img) in imgs. For each image, the following steps are performed: the image is opened using the Image.open() function from the Python Imaging Library (PIL). It is then resized to a fixed size of 30x30 pixels using the resize() method, ensuring consistency in the input image dimensions. Finally, the resized image is converted into a NumPy array using np.array(image), and each pixel value is divided by 255.0 to normalize the pixel values between 0.0 and 1.0. The preprocessed image array is then appended to the data list.

After iterating over all the test images, the data list contains the preprocessed image data. The code converts the data list into a NumPy array using np.array(data), resulting in the X_test variable, which represents the input data for the test images in the ML model.

The following codes are used to make predictions on the test and validation data using a trained model and calculate the accuracy and loss metrics.

```
# Make predictions for test data
test_pred = np.argmax(model.predict(X_test), axis=-1)

# Make predictions for validation data
val_pred = np.argmax(model.predict(X_val), axis=-1)

# Calculate accuracy for test
accuracy_test = accuracy_score(labels, test_pred)
print("Accuracy on the test dataset for model:", accuracy_test)

# Calculate accuracy for validation
accuracy_val = accuracy_score(y_val, val_pred)
print("Accuracy on the validation dataset for model:", accuracy_val)

# Calculate loss for test
loss_test = model.evaluate(X_test, labels)
print("Loss on the test dataset for model:", loss_test)

# Calculate loss for validation
loss_val = model.evaluate(X_val, y_val)
print("Loss on the validation dataset for model:", loss_val)
```

First, test_pred = np.argmax(model.predict(X_test), axis=-1) is executed to obtain the predicted class labels for the test data. The model.predict() function generates the predicted probabilities for each class for each test sample. Then, np.argmax() is applied to obtain the index of the class with the highest probability for each test sample. The resulting test_pred variable contains the predicted class labels for the test data. Similarly, val_pred = np.argmax(model.predict(X_val), axis=-1) is used to obtain the predicted class labels for the validation data in the same manner. The predicted class labels for the validation data are stored in the val_pred variable. To evaluate the accuracy of the test and validation predictions, accuracy_test_pred = accuracy_score(labels, test_pred) calculates the accuracy of the test predictions by comparing the predicted labels (test_pred) with the actual labels (labels). This is done using the accuracy_score() function from scikit-learn, which computes the accuracy metric. The resulting accuracy value is stored in the accuracy_test_pred variable. The accuracy of the test predictions is then printed using print("Accuracy on the test dataset:", accuracy_test_pred). Similarly, the accuracy of the validation predictions is calculated with accuracy_val_pred = accuracy_score(y_val, val_pred), comparing the predicted labels for the validation data (val_pred) with the actual labels (y_val). The accuracy metric is computed and stored in the accuracy_val_pred variable. The accuracy of the validation predictions is printed using print("Accuracy on the validation dataset:", accuracy_val_pred).

To assess the loss (error) of the model on the test and validation data, the following evaluations are performed: loss_test = model.evaluate(X_test, labels) calculates the loss on the test data by evaluating the model's performance using the test data (X_test) and the corresponding true labels (labels). The evaluate() function of the model returns the loss value, which is stored in the loss_test variable. The loss value on the test dataset is printed using print("Loss on the test dataset:", loss_test). Similarly, loss_val = model.evaluate(X_val, y_val) calculates the loss on the validation data in a similar manner. The model's performance is evaluated using the validation data (X_val) and the corresponding true labels (y_val), and the loss value is stored in the loss_val variable. The loss value on the validation dataset is printed using print("Loss on the validation dataset:", loss_val).

And finally, we will use the following codes to create and print the confusion matrices for validation and test data.

```
# Create confusion matrix for validation data
val_cm = confusion_matrix(y_val, val_pred)

# Create confusion matrix for test data
test_cm = confusion_matrix(labels, test_pred)

print("Confusion matrix for validation data for model:")
print(val_cm)
print()

print("Confusion matrix for test data for model:")
print(test_cm)
```

Note: The above codes will be used for all the following cases, so, we will not repeat explaining them again.

1. Max and avg pooling

In this case, we will use two different poolings. First, we will use Maxpooling, then, we will use Avgpooling. The results will be reported after the training.

- Maxpool

To train the CNN model using MaxPooling2D, we will use the following properties given in Table 5 to build the CNN architecture. The activation function used throughout the model is Rectified Linear Unit (ReLU), which introduces non-linearity and allows the model to capture complex patterns in the data. The optimizer selected is Adam, a popular optimization algorithm that adjusts the learning rate dynamically and efficiently updates the model's weights during training. No dropout regularization is applied in this case. The loss function chosen is sparse_categorical_crossentropy, which is suitable for multi-class classification tasks with integer-encoded labels. MaxPooling2D is the pooling type used to downsample the feature maps, helping to reduce spatial dimensions and extract the most salient features. The pool size is set to (2, 2), indicating a 2x2 pooling window. Lastly, the activation function for the output layer is softmax, which generates a probability distribution over the classes, enabling the model to make confident predictions for multi-class classification tasks.

Table 5. Properties CNN model with Maxpooling

Property	Value
Activation function	Relu
Optimizer	Adam
Dropout	--
Loss function	sparse_categorical_crossentropy
Pooling type	MaxPooling2D
Pool size	(2, 2)
Output layer activation	Softmax

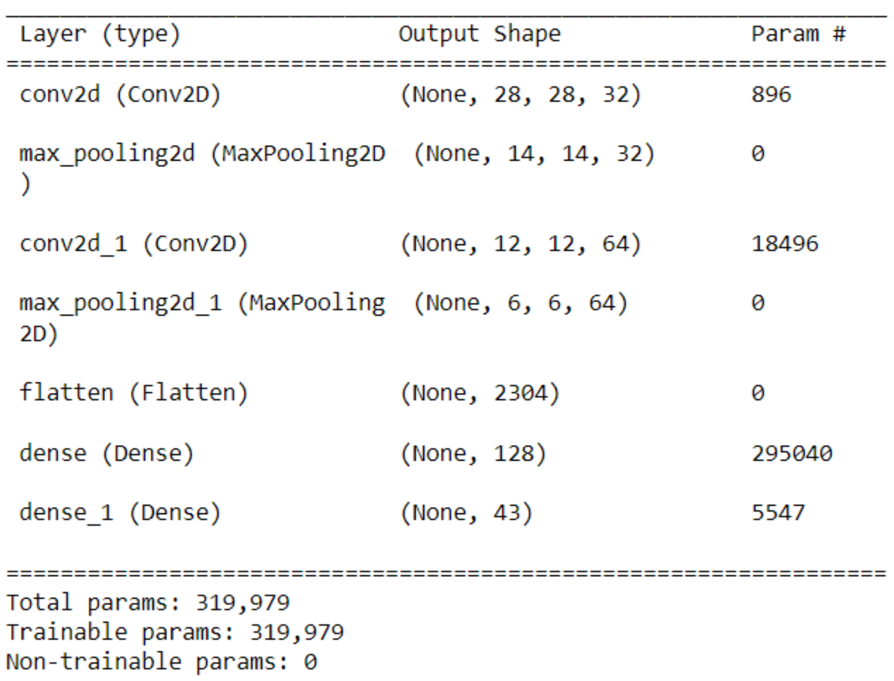
The following code represents the CNN model with the given properties. The summary of the model and its architecture are given in Table 6 and Figure 5, respectively.

```python
# Define the model with MaxPooling2D
model_maxpool = Sequential()
model_maxpool.add(Conv2D(32, (3, 3), activation='relu',
input_shape=(30, 30, 3)))
model_maxpool.add(MaxPooling2D(pool_size=(2, 2)))
model_maxpool.add(Conv2D(64, (3, 3), activation='relu'))
model_maxpool.add(MaxPooling2D(pool_size=(2, 2)))
model_maxpool.add(Flatten())
model_maxpool.add(Dense(128, activation='relu'))
model_maxpool.add(Dense(classes, activation='softmax'))

# Compile the model
model_maxpool.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Print the model summary
model_maxpool.summary()
```

Table 6. CNN model with Maxpooling summary



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 32)        896

 max_pooling2d (MaxPooling2D  (None, 14, 14, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 12, 12, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 flatten (Flatten)           (None, 2304)              0

 dense (Dense)               (None, 128)               295040

 dense_1 (Dense)             (None, 43)                5547

=================================================================
Total params: 319,979
Trainable params: 319,979
Non-trainable params: 0
```

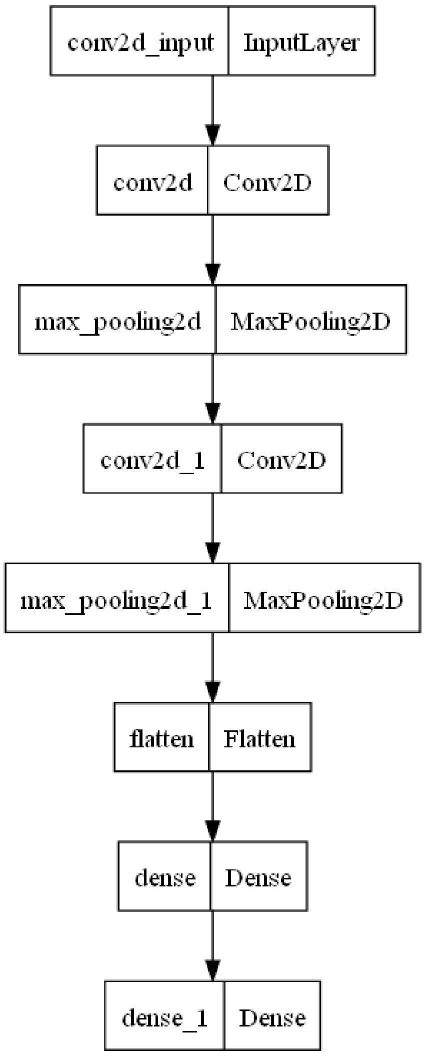

conv2d_input | InputLayer
conv2d | Conv2D
max_pooling2d | MaxPooling2D
conv2d_1 | Conv2D
max_pooling2d_1 | MaxPooling2D
flatten | Flatten
dense | Dense
dense_1 | Dense

Figure 5. Architecture of CNN with Maxpooling

Then, we use the following commands to train the CNN model:

```
# Train the model with maxpool
history_maxpool = model_maxpool.fit(X_train, y_train, batch_size=32,
epochs=10, validation_data=(X_val, y_val))
```

The loss and accuracy plots of training and validation data are shown in Figures 6 and 7. Also, the values of accuracy and loss for training, testing, and validation data are given in Table 7. Furthermore, the confusion matrices for testing and validation data are represented in Figure 8.

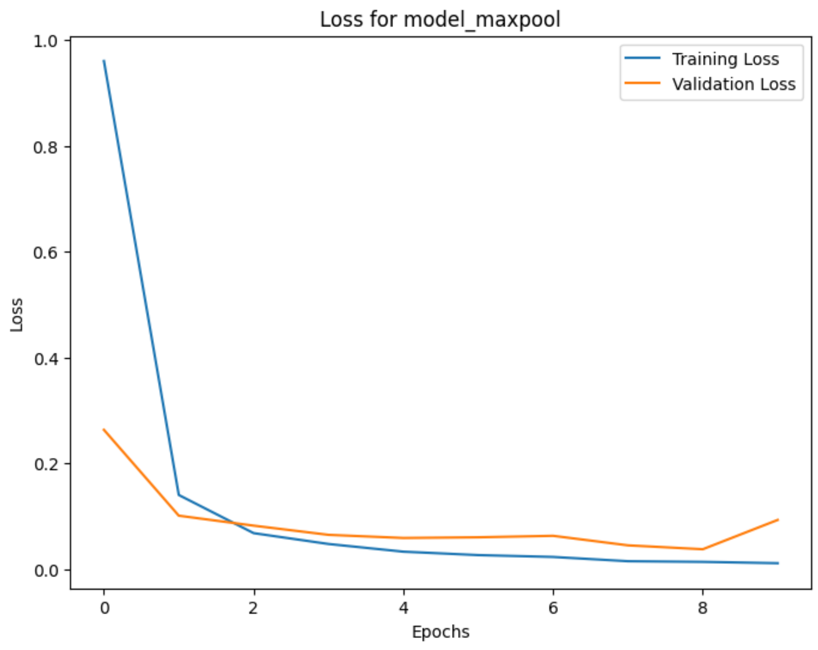


Figure 6. Loss of CNN with Maxpooling

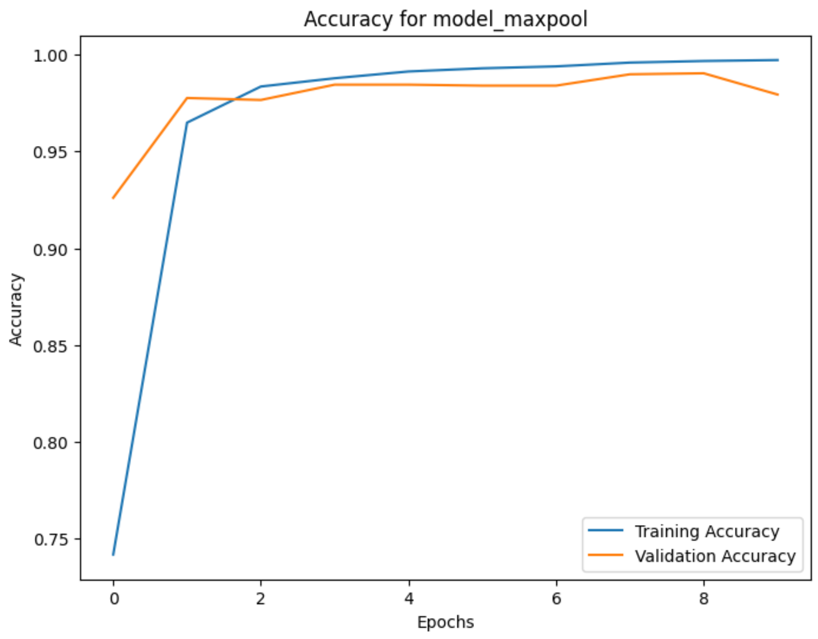
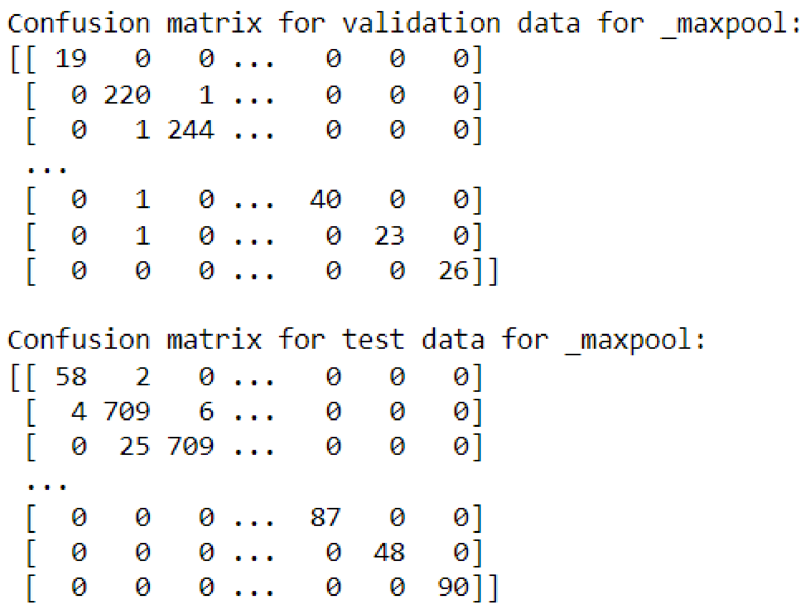


Figure 7. Accuracy of CNN with Maxpooling

Table 7. Results of CNN model training with Maxpooling

Dataset	Accuracy	Loss
Train	0.9971	0.0110
Test	0.9184	0.4431
Validation	0.9793	0.0927

These results indicate that the model has learned to classify the traffic signs with high accuracy on both the training and validation datasets. The model achieved an impressive accuracy of 99.71% on the training dataset, demonstrating its ability to fit the training data very well. However, on the test dataset, the accuracy slightly decreased to 91.84%, indicating some level of overfitting and a drop in performance compared to the training dataset. The validation dataset achieved an accuracy of 97.93%, suggesting that the model generalizes well to unseen data. The loss values on all three datasets are relatively low, indicating that the model's predictions are close to the true labels. Overall, the model shows promising performance.



```
Confusion matrix for validation data for _maxpool:
[[ 19   0   0 ...   0   0   0]
 [  0 220   1 ...   0   0   0]
 [  0   1 244 ...   0   0   0]
 ...
 [  0   1   0 ...  40   0   0]
 [  0   1   0 ...   0  23   0]
 [  0   0   0 ...   0   0  26]]

Confusion matrix for test data for _maxpool:
[[ 58   2   0 ...   0   0   0]
 [  4 709   6 ...   0   0   0]
 [  0  25 709 ...   0   0   0]
 ...
 [  0   0   0 ...  87   0   0]
 [  0   0   0 ...   0  48   0]
 [  0   0   0 ...   0   0  90]]
```

Figure 8. Confusion matrix of CNN with Maxpooling

- Averagepool

To train the CNN model using AvgPooling2D, we will use the following properties given in Table 8 to build the CNN architecture. The activation function used throughout the model is Rectified Linear Unit (ReLU), which introduces non-linearity and allows the model to capture complex patterns in the data. The optimizer selected is Adam, a popular optimization algorithm that adjusts the learning rate dynamically and efficiently updates the model's weights during training. No dropout regularization is applied in this case. The loss function chosen is sparse_categorical_crossentropy, which is suitable for multi-class classification tasks with integer-encoded labels. In this CNN architecture, AveragePooling2D is used as the pooling type. It downsamples the feature maps by taking the average value within each pooling window, helping to reduce spatial dimensions and extract relevant information. The pool size is set to (2, 2), indicating a 2x2 pooling window. Finally, the output layer is activated using the softmax function. This activation function generates a probability distribution over the classes, enabling the model to make confident predictions for multi-class classification tasks.

Table 8. Properties CNN model with Avgpooling

Property	Value
Activation function	Relu
Optimizer	Adam
Dropout	--
Loss function	sparse_categorical_crossentropy
Pooling type	AveragePooling2D
Pool size	(2, 2)
Output layer activation	Softmax

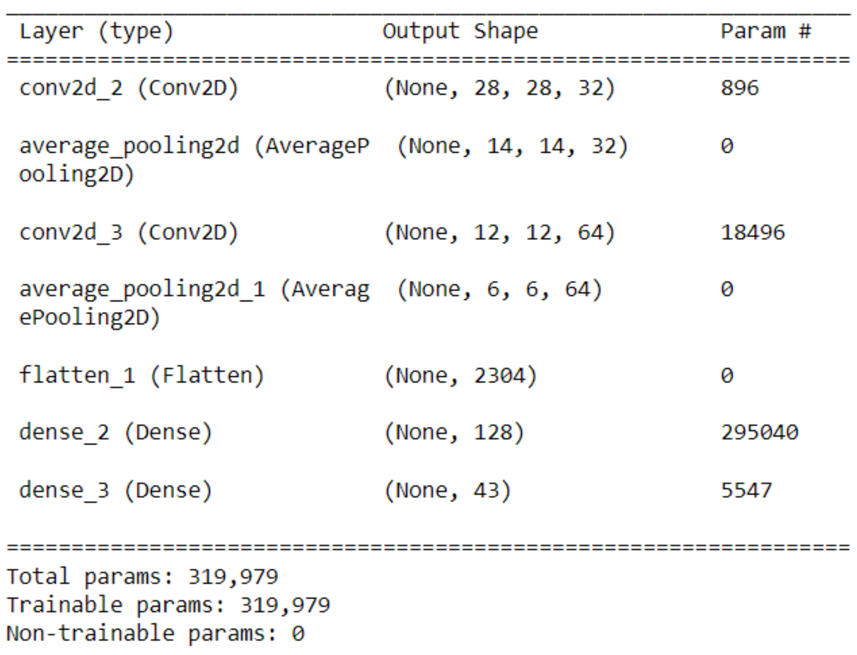
The following code represents the CNN model with the given properties. The summary of the model and its architecture are given in Table 9 and Figure 9, respectively.

```python
# Define the model with AveragePooling2D
model_avgpool = Sequential()
model_avgpool.add(Conv2D(32, (3, 3), activation='relu',
input_shape=(30, 30, 3)))
model_avgpool.add(AveragePooling2D(pool_size=(2, 2)))
model_avgpool.add(Conv2D(64, (3, 3), activation='relu'))
model_avgpool.add(AveragePooling2D(pool_size=(2, 2)))
model_avgpool.add(Flatten())
model_avgpool.add(Dense(128, activation='relu'))
model_avgpool.add(Dense(classes, activation='softmax'))

# Compile the model
model_avgpool.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Print the model summary
model_avgpool.summary()
```

Table 9. CNN model with Avgpooling summary



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 28, 28, 32)        896

 average_pooling2d (AverageP  (None, 14, 14, 32)       0
 ooling2D)

 conv2d_3 (Conv2D)           (None, 12, 12, 64)        18496

 average_pooling2d_1 (Averag  (None, 6, 6, 64)         0
 ePooling2D)

 flatten_1 (Flatten)         (None, 2304)              0

 dense_2 (Dense)             (None, 128)               295040

 dense_3 (Dense)             (None, 43)                5547

=================================================================
Total params: 319,979
Trainable params: 319,979
Non-trainable params: 0
```

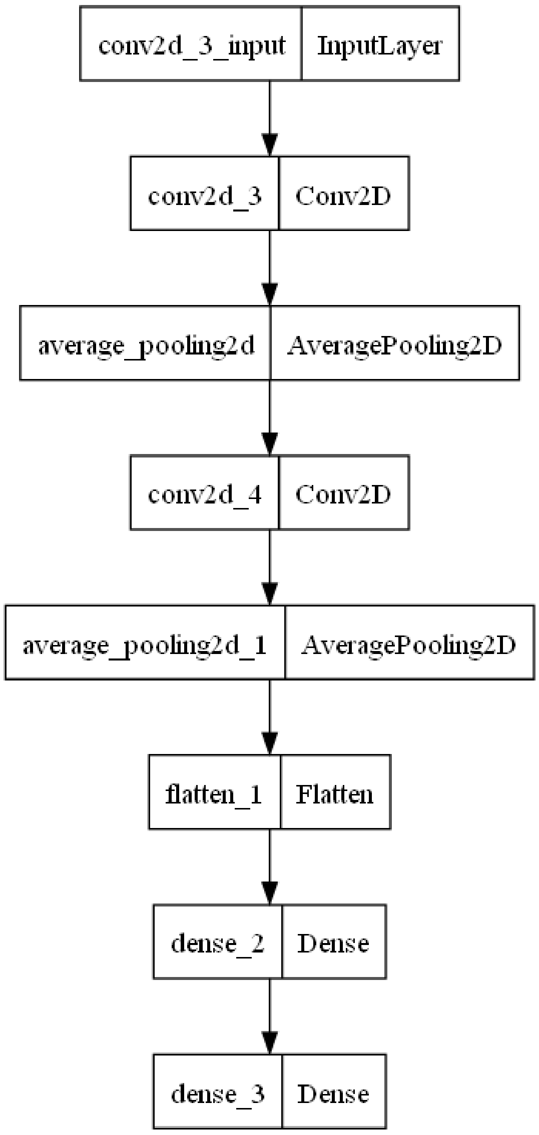

conv2d_3_input | InputLayer
conv2d_3 | Conv2D
average_pooling2d | AveragePooling2D
conv2d_4 | Conv2D
average_pooling2d_1 | AveragePooling2D
flatten_1 | Flatten
dense_2 | Dense
dense_3 | Dense

Figure 9. Architecture of CNN with Avgpooling

Then, we use the following commands to train the CNN model:

```
# Train the model with avgpool
history_avgpool = model_avgpool.fit(X_train, y_train, batch_size=32,
epochs=10, validation_data=(X_val, y_val))
```

The loss and accuracy plots of training and validation data are shown in Figures 10 and 11. Also, the values of accuracy and loss for training, testing, and validation data are given in Table 10. Furthermore, the confusion matrices for testing and validation data are represented in Figure 12.

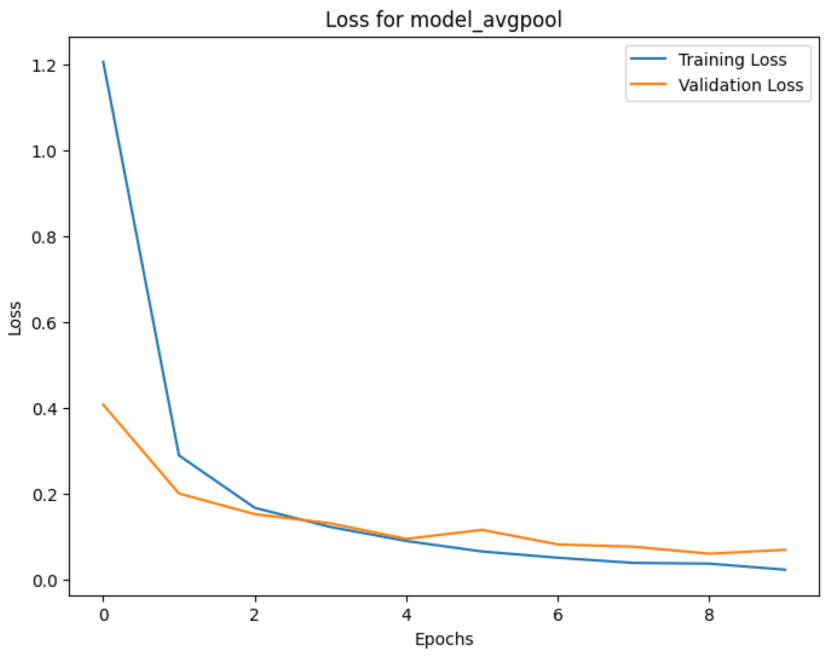


Figure 10. Loss of CNN neural network with Avgpooling

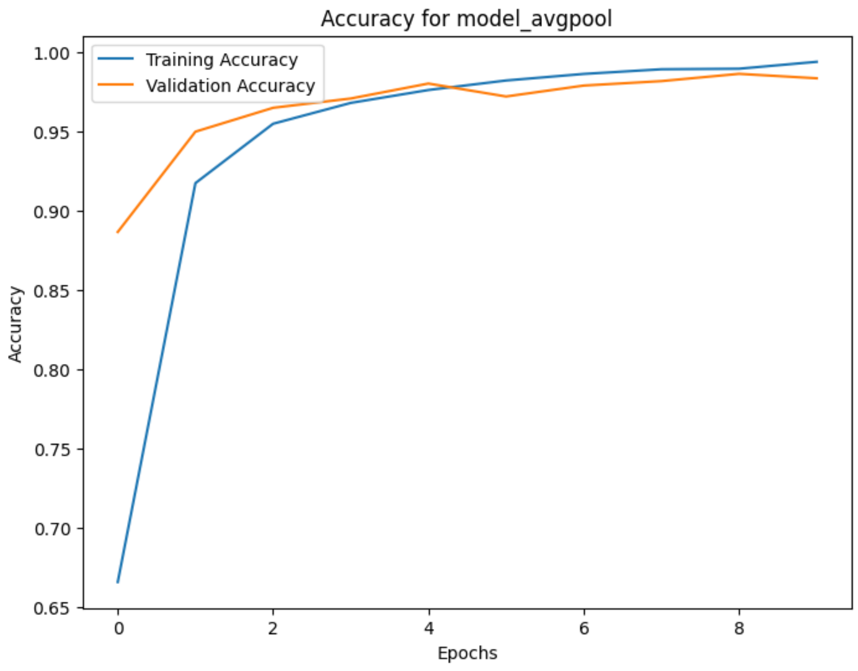


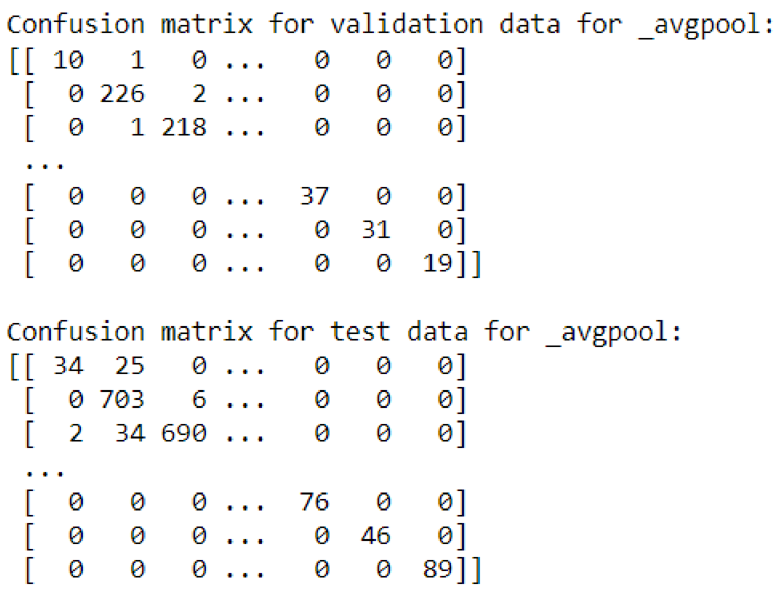
Figure 11. Accuracy of CNN neural network with Avgpooling

Table 10. Results of CNN model training with Avgpooling

Dataset	Accuracy	Loss
Train	0.9938	0.0234
Test	0.9043	0.6057
Validation	0.9834	0.0696

These results provide insights into the model's performance on different datasets. On the training dataset, the model achieved an accuracy of 99.38% and a relatively low loss value of 0.0234, indicating that the model has effectively learned the patterns and features in the training data. However, when evaluated on the test dataset, the model's accuracy decreased to 90.43%, and the loss increased to 0.6057. This suggests that the model may be overfitting the training data to some extent, resulting in reduced performance on unseen test data.

On the other hand, the model performed well on the validation dataset, achieving an accuracy of 98.34% and a low loss of 0.0696. This indicates that the model generalizes well to new, unseen data.



```
Confusion matrix for validation data for _avgpool:
[[ 10   1   0 ...   0   0   0]
 [  0 226   2 ...   0   0   0]
 [  0   1 218 ...   0   0   0]
 ...
 [  0   0   0 ...  37   0   0]
 [  0   0   0 ...   0  31   0]
 [  0   0   0 ...   0   0  19]]

Confusion matrix for test data for _avgpool:
[[ 34  25   0 ...   0   0   0]
 [  0 703   6 ...   0   0   0]
 [  2  34 690 ...   0   0   0]
 ...
 [  0   0   0 ...  76   0   0]
 [  0   0   0 ...   0  46   0]
 [  0   0   0 ...   0   0  89]]
```

Figure 12. Confusion matrix of CNN with Avgpooling

2. With and without dropout

In this section, we will use two different cases. First, we will use a dropout, then, we will train the model without dropout. The results will be reported after the training.

- No dropout

To train the CNN model without dropout, we will use the following properties given in Table 11 to build the CNN architecture. The activation function used throughout the model is Rectified Linear Unit (ReLU), which introduces non-linearity and allows the model to capture complex patterns in the data. The optimizer selected is Adam, a popular optimization algorithm that adjusts the learning rate dynamically and efficiently updates the model's weights during training. In this configuration, no dropout regularization is applied. Dropout is a regularization technique that randomly drops a fraction of neurons during training to prevent overfitting. However, in this case, dropout is not utilized. The loss function chosen is sparse_categorical_crossentropy, which is suitable for multi-class classification tasks with integer-encoded labels. MaxPooling2D is used as the pooling type, which downsamples the feature maps by selecting the maximum value within each pooling window. This helps to reduce spatial dimensions and extract the most salient features. The pool size is set to (2, 2), indicating a 2x2 pooling window. The output layer is activated using the softmax function, which generates a probability distribution over the classes, enabling the model to make confident predictions for multi-class classification tasks.

Table 11. Properties CNN model without dropout

Property	Value
Activation function	Relu
Optimizer	Adam
Dropout	--
Loss function	sparse_categorical_crossentropy
Pooling type	MaxPooling2D
Pool size	(2, 2)
Output layer activation	Softmax

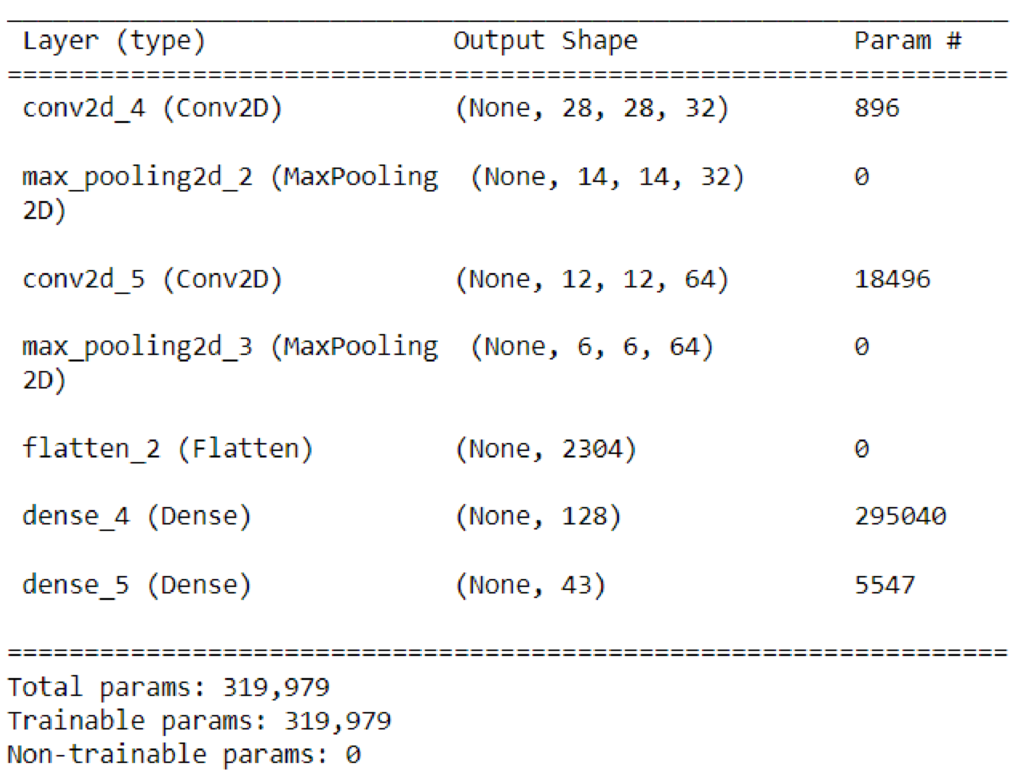
The following code represents the CNN model with the given properties. The summary of the model and its architecture are given in Table 12 and Figure 13, respectively.

```
# Define the model without Dropout
model_no_dropout = Sequential()
model_no_dropout.add(Conv2D(32, (3, 3), activation='relu',
input_shape=(30, 30, 3)))
model_no_dropout.add(MaxPooling2D(pool_size=(2, 2)))
model_no_dropout.add(Conv2D(64, (3, 3), activation='relu'))
model_no_dropout.add(MaxPooling2D(pool_size=(2, 2)))
model_no_dropout.add(Flatten())
model_no_dropout.add(Dense(128, activation='relu'))
model_no_dropout.add(Dense(classes, activation='softmax'))

# Compile the model
model_no_dropout.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Print the model summary
model_no_dropout.summary()
```

Table 12. CNN model with without dropout summary



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 28, 28, 32)        896

 max_pooling2d_2 (MaxPooling  (None, 14, 14, 32)       0
 2D)

 conv2d_5 (Conv2D)           (None, 12, 12, 64)        18496

 max_pooling2d_3 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 flatten_2 (Flatten)         (None, 2304)              0

 dense_4 (Dense)             (None, 128)               295040

 dense_5 (Dense)             (None, 43)                5547

=================================================================
Total params: 319,979
Trainable params: 319,979
Non-trainable params: 0
```

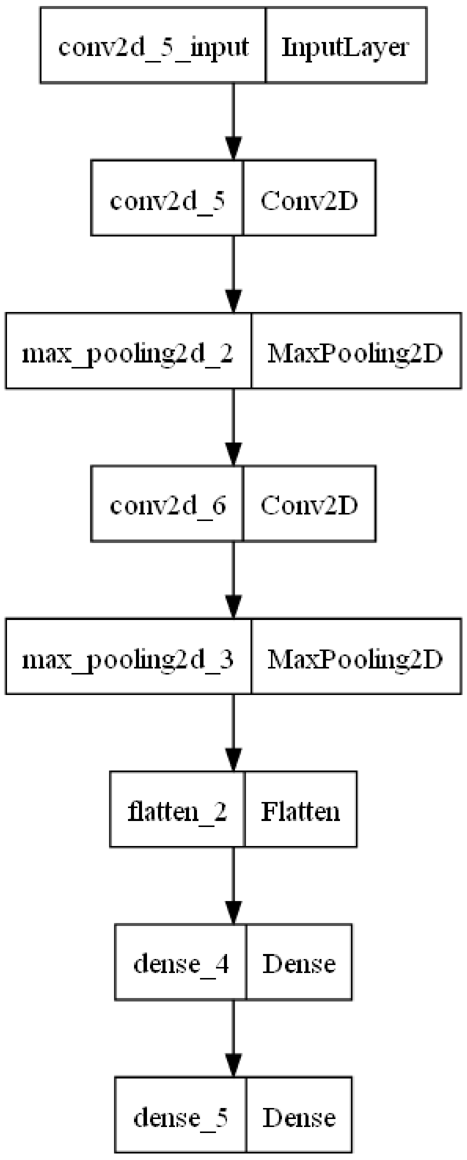

conv2d_5_input | InputLayer
conv2d_5 | Conv2D
max_pooling2d_2 | MaxPooling2D
conv2d_6 | Conv2D
max_pooling2d_3 | MaxPooling2D
flatten_2 | Flatten
dense_4 | Dense
dense_5 | Dense

Figure 13. Architecture of CNN without dropout

Then, we use the following commands to train the CNN model:

```
# Train the model with no_dropout
history_no_dropout = model_no_dropout.fit(X_train, y_train,
batch_size=32, epochs=10, validation_data=(X_val, y_val))
```

The loss and accuracy plots of training and validation data are shown in Figures 14 and 15. Also, the values of accuracy and loss for training, testing, and validation data are given in Table 13. Furthermore, the confusion matrices for testing and validation data are represented in Figure 16.

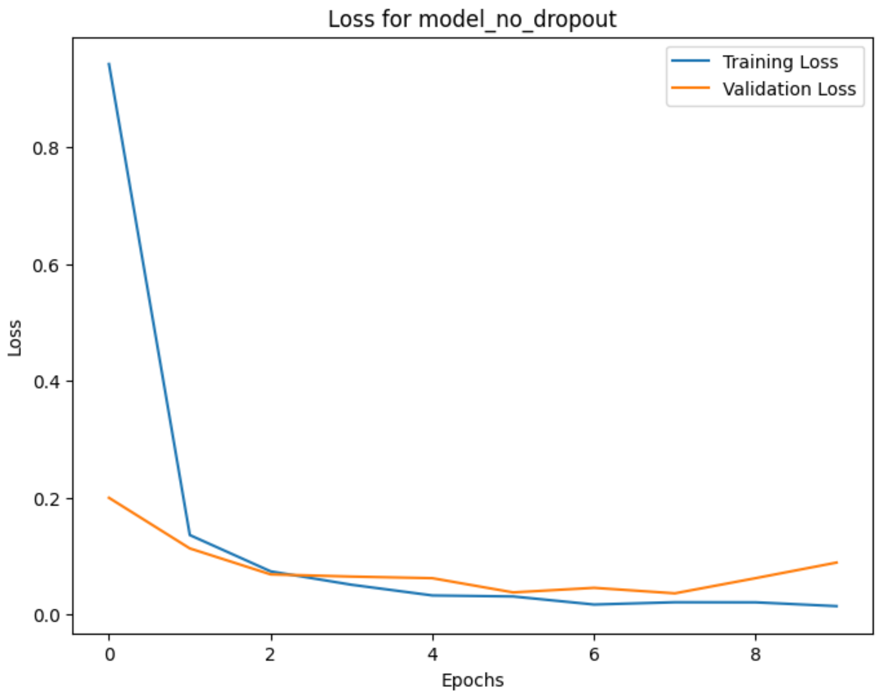


Figure 14. Loss of CNN without dropout

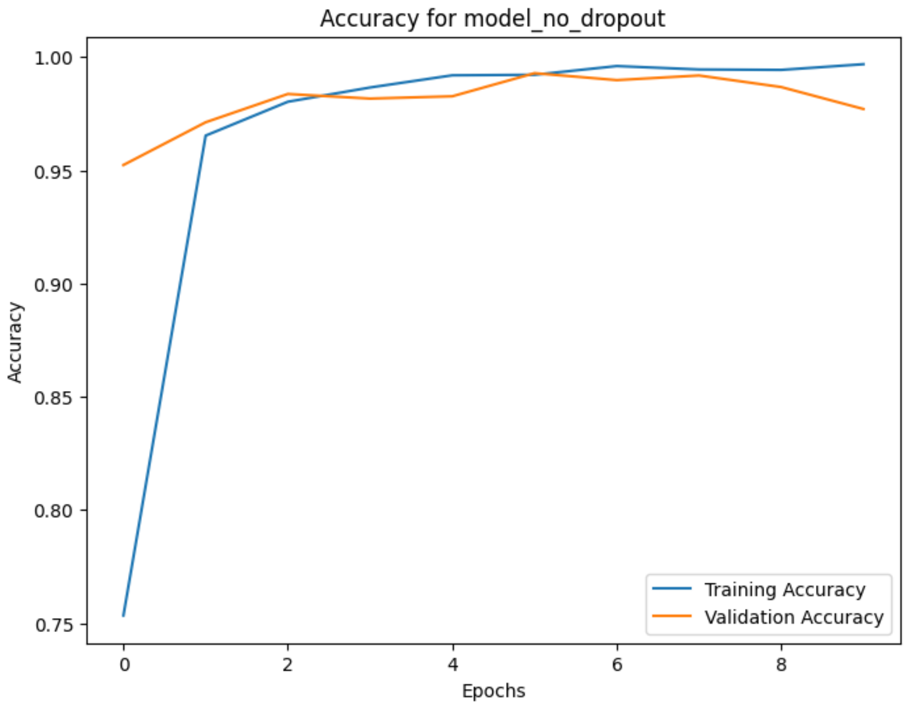


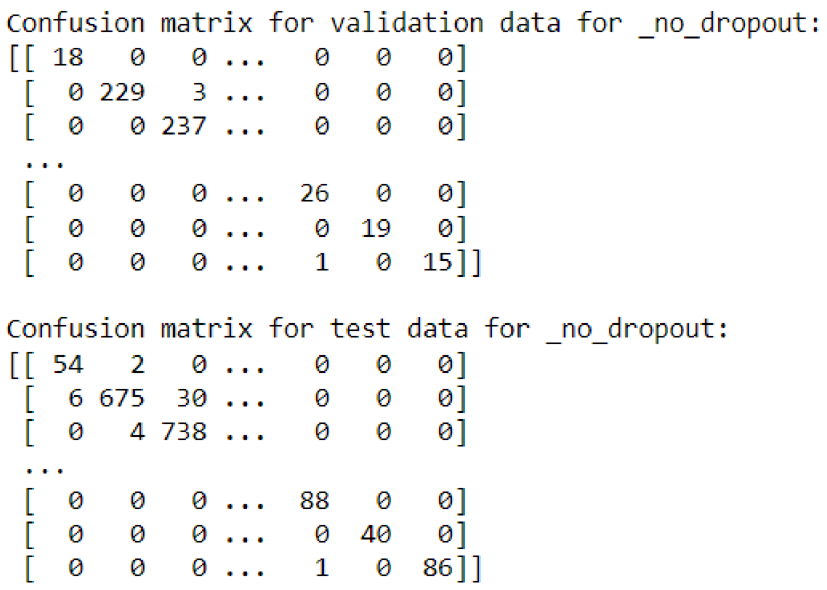
Figure 15. Accuracy of CNN without dropout

Table 13. Results of CNN model training without dropout

Dataset	Accuracy	Loss
Train	0.9968	0.0139
Test	0.9276	0.3845
Validation	0.9770	0.0884

These results provide insights into the model's performance on different datasets. On the training dataset, the model achieved an accuracy of 99.68% and a relatively low loss value of 0.0139, indicating that the model has effectively learned the patterns and features in the training data. When evaluated on the test dataset, the model achieved an accuracy of 92.76% and a loss of 0.3845. This suggests that the model performs well on unseen data but may have some room for improvement in terms of accuracy and loss compared to the training dataset.

On the validation dataset, the model achieved an accuracy of 97.70% and a loss of 0.0884, indicating that the model generalizes well to new, unseen data and performs consistently with good accuracy. Overall, these results demonstrate that the CNN model has learned to classify the traffic sign images with high accuracy and has generalization capability.



```
Confusion matrix for validation data for _no_dropout:
[[ 18   0   0 ...   0   0   0]
 [  0 229   3 ...   0   0   0]
 [  0   0 237 ...   0   0   0]
 ...
 [  0   0   0 ...  26   0   0]
 [  0   0   0 ...   0  19   0]
 [  0   0   0 ...   1   0  15]]

Confusion matrix for test data for _no_dropout:
[[ 54   2   0 ...   0   0   0]
 [  6 675  30 ...   0   0   0]
 [  0   4 738 ...   0   0   0]
 ...
 [  0   0   0 ...  88   0   0]
 [  0   0   0 ...   0  40   0]
 [  0   0   0 ...   1   0  86]]
```

Figure 16. Confusion matrix of CNN without dropout

- With dropout

To train the CNN model with dropout, we will use the following properties given in Table 14 to build the CNN architecture. The activation function used throughout the model is Rectified Linear Unit (ReLU), which introduces non-linearity and allows the model to capture complex patterns in the data. The optimizer selected is Adam, a popular optimization algorithm that adjusts the learning rate dynamically and efficiently updates the model's weights during training. In this configuration, dropout regularization is applied with a rate of 0.25. Dropout randomly sets a fraction of the input units to 0 at each update during training, which helps to prevent overfitting and improve the model's generalization ability. The loss function chosen is sparse_categorical_crossentropy, which is suitable for multi-class classification tasks with integer-encoded labels. MaxPooling2D is used as the pooling type, which downsamples the feature maps by selecting the maximum value within each pooling window. This helps to reduce spatial dimensions and extract the most salient features. The pool size is set to (2, 2), indicating a 2x2 pooling window. As always, the output layer is activated using the softmax function,

Table 14. Properties CNN model with dropout

Property	Value
Activation function	Relu
Optimizer	Adam
Dropout	0.25
Loss function	sparse_categorical_crossentropy
Pooling type	MaxPooling2D
Pool size	(2, 2)
Output layer activation	Softmax

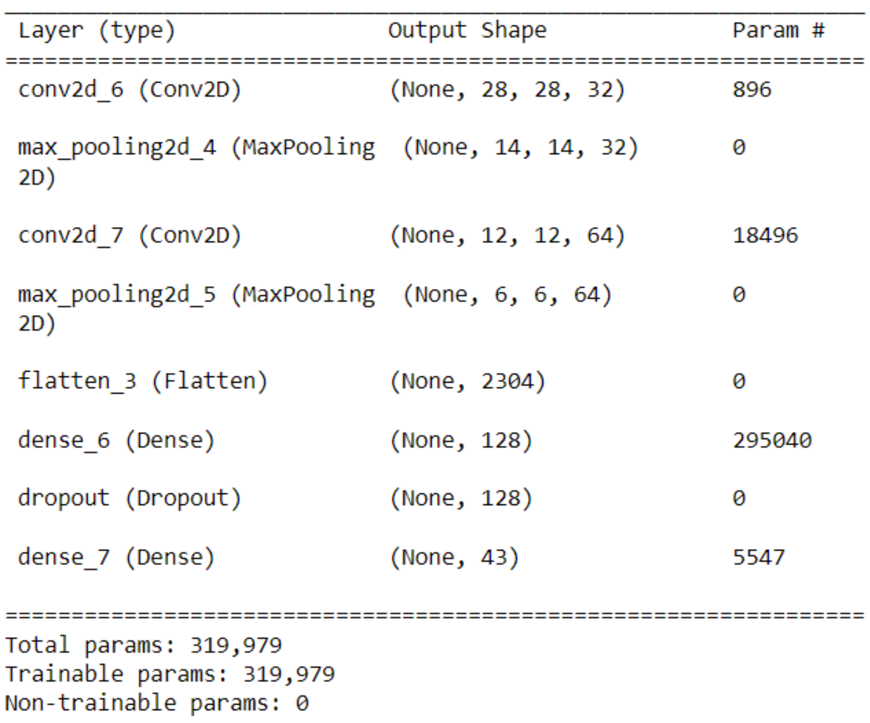
The following code represents the CNN model with the given properties. The summary of the model and its architecture are given in Table 15 and Figure 17, respectively.

```python
# Define the model with Dropout
model_with_dropout = Sequential()
model_with_dropout.add(Conv2D(32, (3, 3), activation='relu',
input_shape=(30, 30, 3)))
model_with_dropout.add(MaxPooling2D(pool_size=(2, 2)))
model_with_dropout.add(Conv2D(64, (3, 3), activation='relu'))
model_with_dropout.add(MaxPooling2D(pool_size=(2, 2)))
model_with_dropout.add(Flatten())
model_with_dropout.add(Dense(128, activation='relu'))
model_with_dropout.add(Dropout(0.25))  # Dropout layer with dropout
rate of 0.25
model_with_dropout.add(Dense(classes, activation='softmax'))

# Compile the model
model_with_dropout.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Print the model summary
model_with_dropout.summary()
```

Table 15. CNN model with dropout summary



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 28, 28, 32)        896

 max_pooling2d_4 (MaxPooling  (None, 14, 14, 32)       0
 2D)

 conv2d_7 (Conv2D)           (None, 12, 12, 64)        18496

 max_pooling2d_5 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 flatten_3 (Flatten)         (None, 2304)              0

 dense_6 (Dense)             (None, 128)               295040

 dropout (Dropout)           (None, 128)               0

 dense_7 (Dense)             (None, 43)                5547

=================================================================
Total params: 319,979
Trainable params: 319,979
Non-trainable params: 0
```

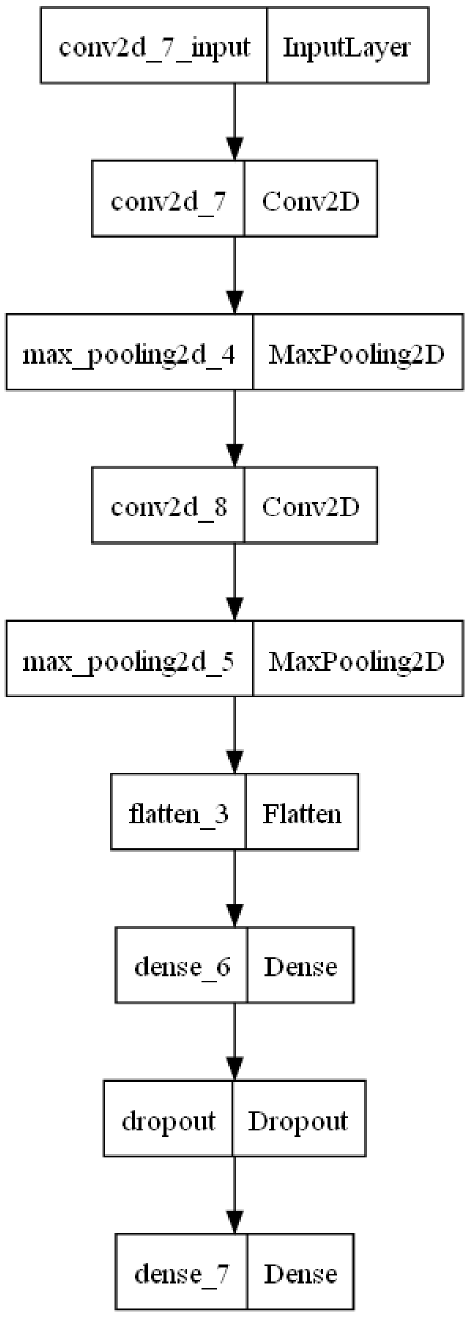


Figure 17. Architecture of CNN with dropout

Then, we use the following commands to train the CNN model:

```
# Train the model with_dropout
history_with_dropout = model_with_dropout.fit(X_train, y_train,
batch_size=32, epochs=10, validation_data=(X_val, y_val))
```

The loss and accuracy plots of training and validation data are shown in Figures 18 and 19. Also, the values of accuracy and loss for training, testing, and validation data are given in Table

16. Furthermore, the confusion matrices for testing and validation data are represented in Figure 20.

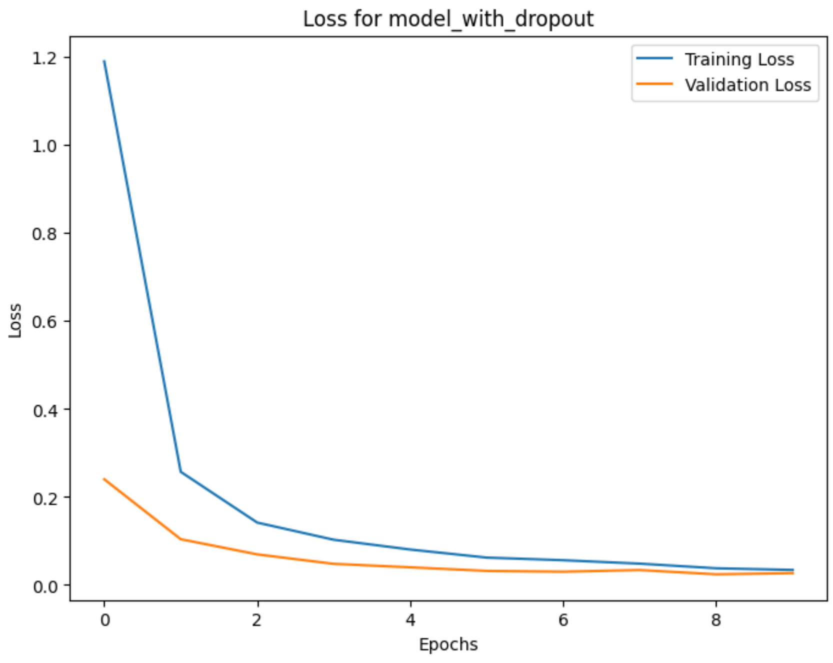


Figure 18. Loss of CNN with dropout

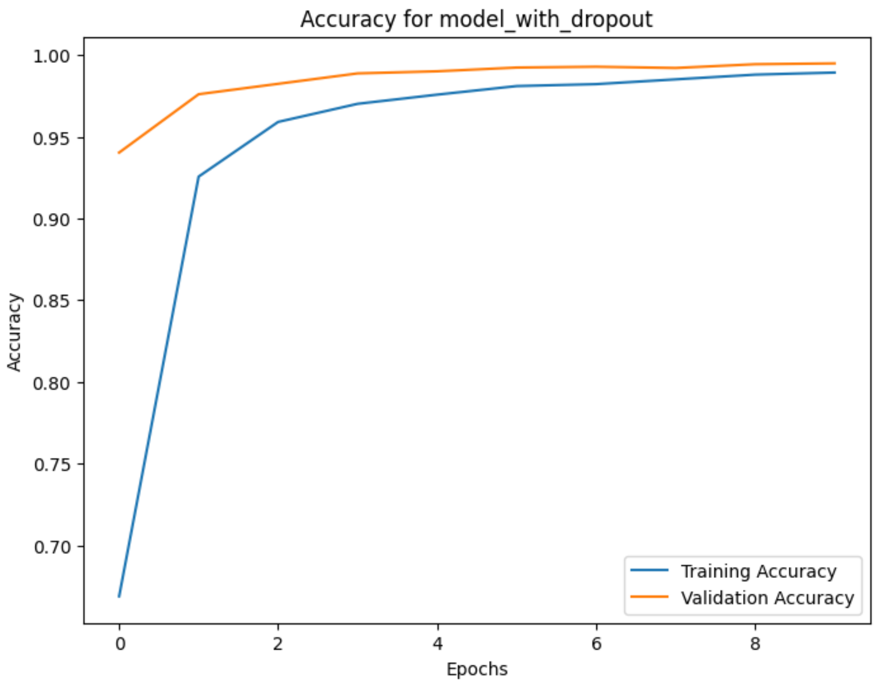


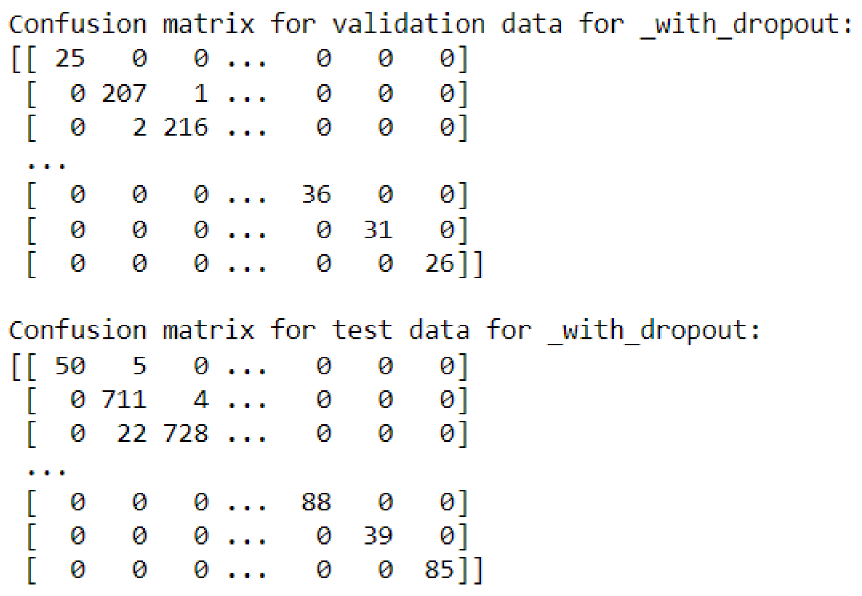
Figure 19. Accuracy of CNN with dropout

Table 16. Results of CNN model training with dropout

Dataset	Accuracy	Loss
Train	0.9893	0.0342
Test	0.9490	0.3076
Validation	0.9948	0.0269

The performance of the CNN deep neural network model was evaluated on different datasets, yielding the following results. These results provide insights into the model's performance on each dataset and its ability to generalize to unseen data. The model achieved a high accuracy of 98.93% on the training dataset, indicating that it has learned the patterns and features present in the training data effectively. The corresponding loss value of 0.0342 suggests that the model has minimized the discrepancies between the predicted and actual labels. When evaluated on the test dataset, the model achieved an accuracy of 94.90%, indicating that it can accurately classify traffic signs in new, unseen data. The loss value of 0.3076 is relatively higher compared to the training dataset, suggesting some level of discrepancy between the predicted and actual labels. However, the model still performs well on the test data, showcasing its ability to generalize and make accurate predictions. The validation dataset results further demonstrate the model's generalization capability. With an accuracy of 99.48% and a low loss of 0.0269, the model performs exceptionally well on this unseen dataset, highlighting its ability to classify traffic signs accurately and consistently.

We can conclude these results indicate that the CNN model exhibits strong performance on all three datasets, with high accuracy and relatively low loss values. It demonstrates its capability to learn and generalize the features of traffic signs effectively, making it a reliable model for traffic sign classification tasks.



```
Confusion matrix for validation data for _with_dropout:
[[ 25   0   0 ...   0   0   0]
 [  0 207   1 ...   0   0   0]
 [  0   2 216 ...   0   0   0]
 ...
 [  0   0   0 ...  36   0   0]
 [  0   0   0 ...   0  31   0]
 [  0   0   0 ...   0   0  26]]

Confusion matrix for test data for _with_dropout:
[[ 50   5   0 ...   0   0   0]
 [  0 711   4 ...   0   0   0]
 [  0  22 728 ...   0   0   0]
 ...
 [  0   0   0 ...  88   0   0]
 [  0   0   0 ...   0  39   0]
 [  0   0   0 ...   0   0  85]]
```

Figure 20. Confusion matrix of CNN with dropout

3. Optimizers

- Adam

To train the CNN model with the Adam optimizer, we will use the following properties given in Table 17 to build the CNN architecture. The activation function used in this model is the Rectified Linear Unit (ReLU), which introduces non-linearity and allows the model to capture complex patterns in the data. The optimizer chosen is Adam, a popular optimization algorithm known for its effectiveness in training deep neural networks. Adam combines the benefits of both AdaGrad and RMSprop algorithms, adapting the learning rate based on the gradient's first and second moments. This adaptive learning rate helps to converge faster and handle different types of data efficiently. In this configuration, no dropout regularization is applied. The loss function used is sparse_categorical_crossentropy, which is appropriate for multi-class classification tasks with integer-encoded labels. MaxPooling2D is employed as the pooling type. The pooling layer uses a pooling window size of (2, 2), indicating a 2x2 window for downsampling. The output layer is activated using the softmax function.

By combining these properties, the CNN model is constructed using the Adam optimizer, providing an effective framework for training and achieving high accuracy in multi-class traffic sign classification tasks.

Table 17. Properties CNN model with Adam

Property	Value
Activation function	Relu
Optimizer	Adam
Dropout	--
Loss function	sparse_categorical_crossentropy
Pooling type	MaxPooling2D
Pool size	(2, 2)
Output layer activation	Softmax

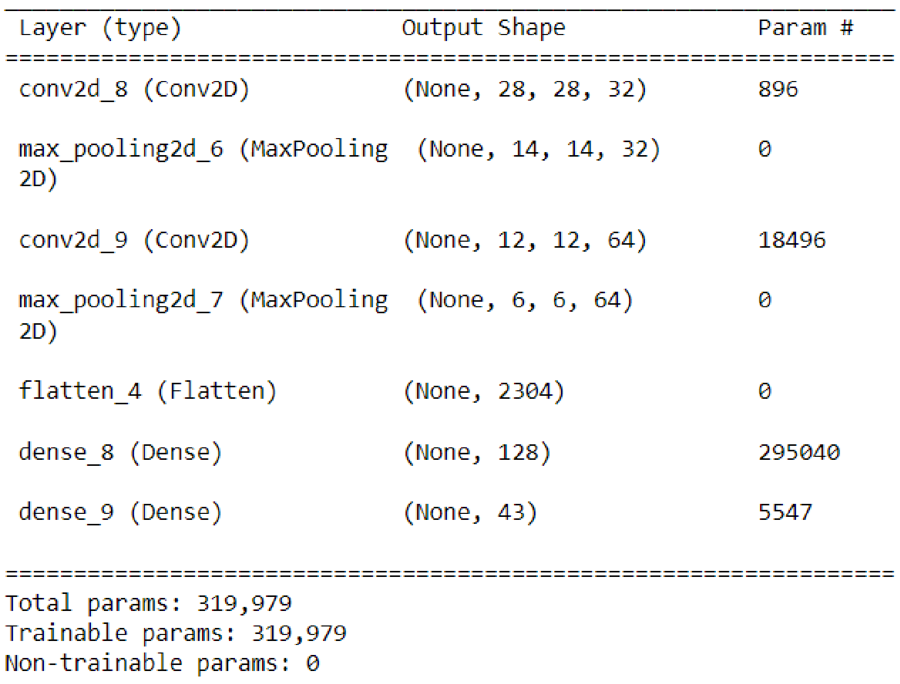
The following code represents the CNN model with the given properties. The summary of the model and its architecture are given in Table 18 and Figure 21, respectively.

```python
# Define the model with Adam optimizer
model_adam = Sequential()
model_adam.add(Conv2D(32, (3, 3), activation='relu', input_shape=(30,
30, 3)))
model_adam.add(MaxPooling2D(pool_size=(2, 2)))
model_adam.add(Conv2D(64, (3, 3), activation='relu'))
model_adam.add(MaxPooling2D(pool_size=(2, 2)))
model_adam.add(Flatten())
model_adam.add(Dense(128, activation='relu'))
model_adam.add(Dense(classes, activation='softmax'))

# Compile the model with Adam optimizer
model_adam.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Print the model summary
model_adam.summary()
```

Table 18. CNN model with Adam summary



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_8 (Conv2D)           (None, 28, 28, 32)        896

 max_pooling2d_6 (MaxPooling  (None, 14, 14, 32)       0
 2D)

 conv2d_9 (Conv2D)           (None, 12, 12, 64)        18496

 max_pooling2d_7 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 flatten_4 (Flatten)         (None, 2304)              0

 dense_8 (Dense)             (None, 128)               295040

 dense_9 (Dense)             (None, 43)                5547

=================================================================
Total params: 319,979
Trainable params: 319,979
Non-trainable params: 0
```

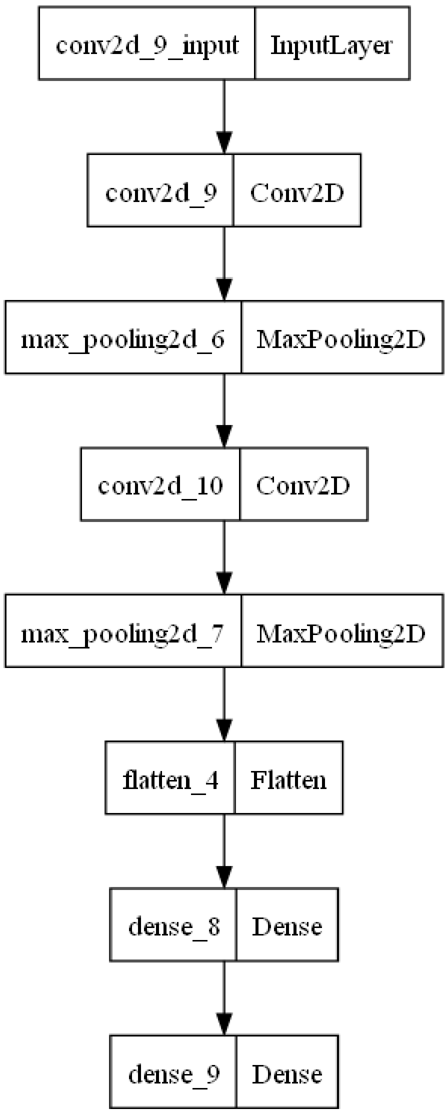


Figure 21. Architecture of CNN with Adam

Then, we use the following commands to train the CNN model:

```
# Train the model with adam
history_adam = model_adam.fit(X_train, y_train, batch_size=32,
epochs=10, validation_data=(X_val, y_val))
```

The loss and accuracy plots of training and validation data are shown in Figures 22 and 23. Also, the values of accuracy and loss for training, testing, and validation data are given in Table 19. Furthermore, the confusion matrices for testing and validation data are represented in Figure 24.

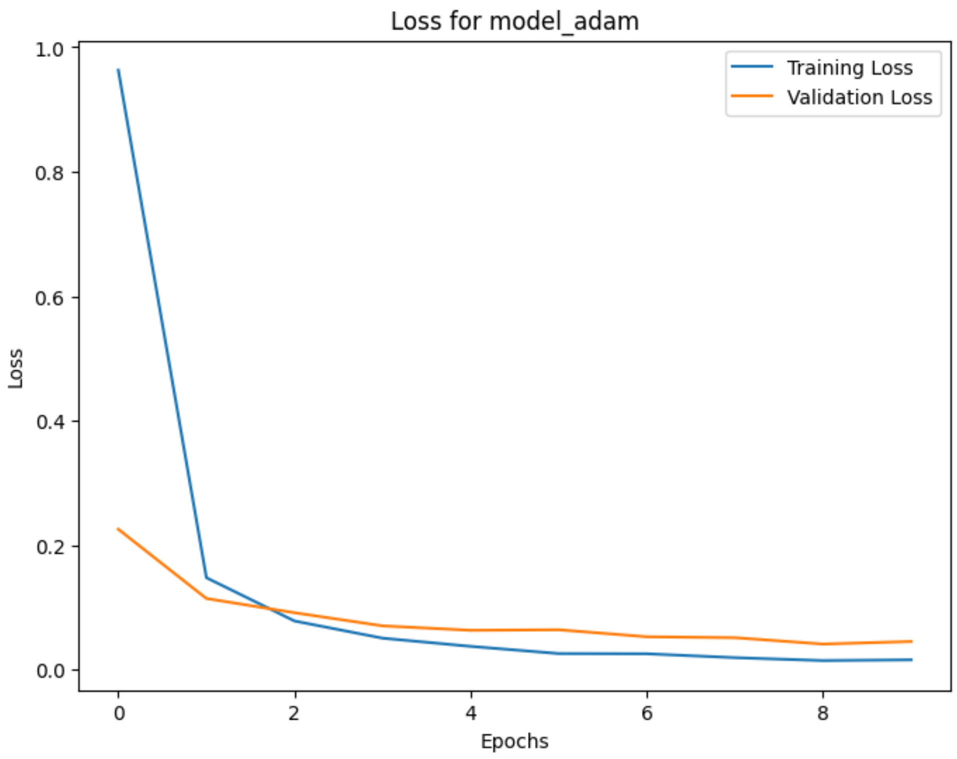


Figure 22. Loss of CNN with Adam

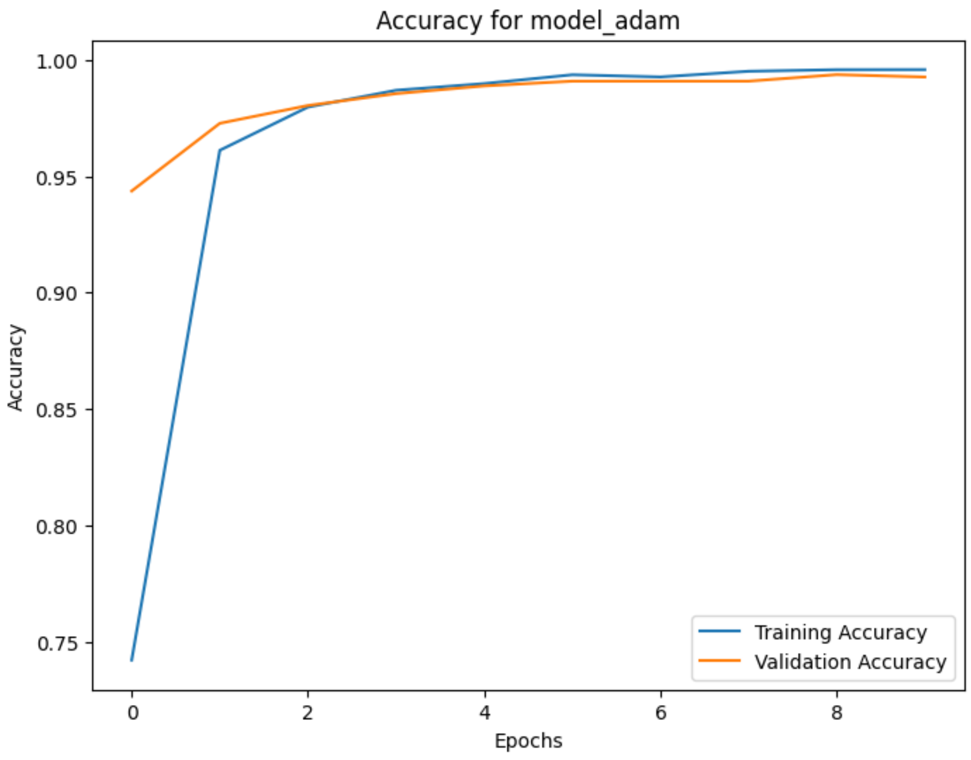


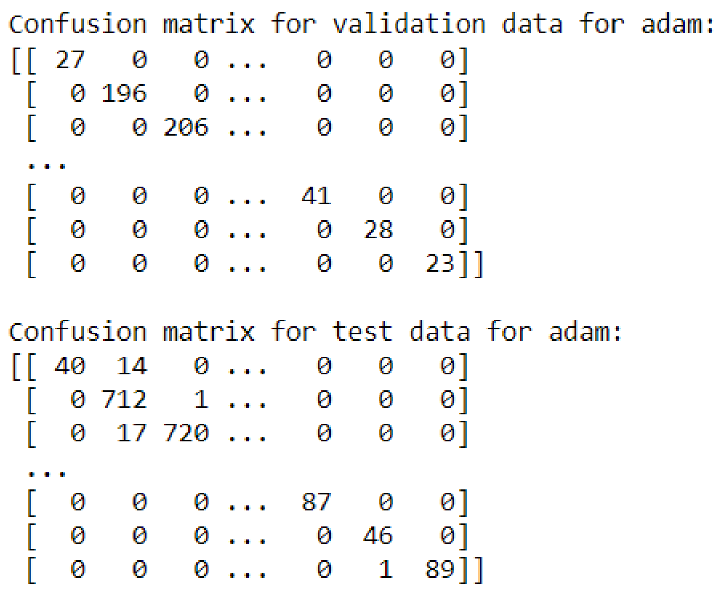
Figure 23. Accuracy of CNN with Adam

Table 19. Results of CNN model training with Adam

Dataset	Accuracy	Loss
Train	0.9958	0.0162
Test	0.9429	0.3227
Validation	0.9926	0.0455

The CNN deep neural network model, trained with the Adam optimizer and the specified properties, demonstrates strong performance in classifying traffic signs. The model achieves an accuracy of 99.58% on the training dataset, indicating its ability to effectively learn and generalize from the training samples. The loss value of 0.0162 further confirms the model's capability to minimize errors during training. When evaluated on the test dataset, the model achieves an accuracy of 94.29%. This result demonstrates the model's ability to generalize well to unseen data and perform accurately on real-world traffic sign images. The corresponding loss value of 0.3227 suggests that the model's predictions are close to the ground truth labels, as indicated by the relatively low loss value. The model also performs well on the validation dataset, with an accuracy of 99.26%. This result indicates the model's ability to generalize to new samples and make accurate predictions. The low loss value of 0.0455 suggests that the model's predictions align closely with the true labels in the validation set.

So, the evaluation results indicate that the CNN model with the specified properties and the Adam optimizer exhibits strong performance in classifying traffic signs. The high accuracies achieved on the training, test, and validation datasets demonstrate the model's ability to generalize and make accurate predictions. The low loss values further confirm the model's effectiveness in minimizing errors during training and inference.



```
Confusion matrix for validation data for adam:
[[ 27   0   0 ...   0   0   0]
 [  0 196   0 ...   0   0   0]
 [  0   0 206 ...   0   0   0]
 ...
 [  0   0   0 ...  41   0   0]
 [  0   0   0 ...   0  28   0]
 [  0   0   0 ...   0   0  23]]

Confusion matrix for test data for adam:
[[ 40  14   0 ...   0   0   0]
 [  0 712   1 ...   0   0   0]
 [  0  17 720 ...   0   0   0]
 ...
 [  0   0   0 ...  87   0   0]
 [  0   0   0 ...   0  46   0]
 [  0   0   0 ...   0   1  89]]
```

Figure 24. Confusion matrix of CNN with Adam

- Gradient descent

To train the CNN model with the Gradient Descent optimizer, the following properties specified in Table 20 are utilized to construct the CNN architecture. To train the CNN model, the architecture includes ReLU activation, MaxPooling2D with a (2, 2) pool size, and a softmax output layer. The loss function is sparse_categorical_crossentropy. No dropout layer is used.

Table 20. Properties CNN model with Gradient Descent

Property	Value
Activation function	Relu
Optimizer	Gradient Descent
Dropout	--
Loss function	sparse_categorical_crossentropy
Pooling type	MaxPooling2D
Pool size	(2, 2)
Output layer activation	Softmax

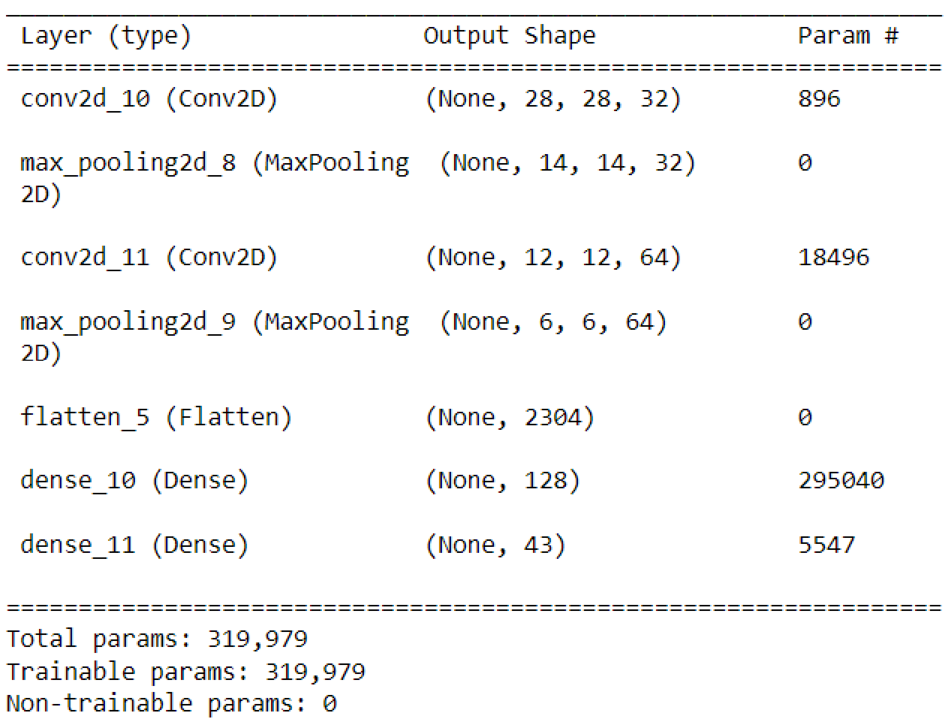
The following code represents the CNN model with the given properties. The summary of the model and its architecture are given in Table 21 and Figure 25, respectively.

```
# Define the model with Gradient Descent optimizer
model_gradient_descent = Sequential()
model_gradient_descent.add(Conv2D(32, (3, 3), activation='relu',
input_shape=(30, 30, 3)))
model_gradient_descent.add(MaxPooling2D(pool_size=(2, 2)))
model_gradient_descent.add(Conv2D(64, (3, 3), activation='relu'))
model_gradient_descent.add(MaxPooling2D(pool_size=(2, 2)))
model_gradient_descent.add(Flatten())
model_gradient_descent.add(Dense(128, activation='relu'))
model_gradient_descent.add(Dense(classes, activation='softmax'))

# Compile the model with Gradient Descent optimizer
model_gradient_descent.compile(optimizer='sgd',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Print the model summary
model_gradient_descent.summary()
```

Table 21. CNN model with Gradient Descent summary



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_10 (Conv2D)          (None, 28, 28, 32)        896

 max_pooling2d_8 (MaxPooling  (None, 14, 14, 32)       0
 2D)

 conv2d_11 (Conv2D)          (None, 12, 12, 64)        18496

 max_pooling2d_9 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 flatten_5 (Flatten)         (None, 2304)              0

 dense_10 (Dense)            (None, 128)               295040

 dense_11 (Dense)            (None, 43)                5547

=================================================================
Total params: 319,979
Trainable params: 319,979
Non-trainable params: 0
```

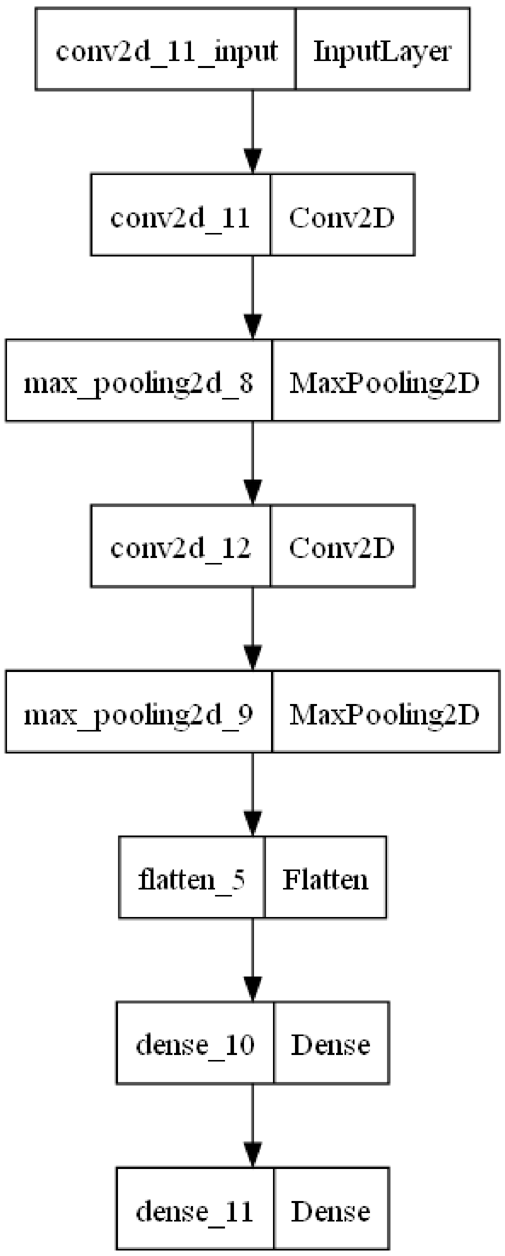


Figure 25. Architecture of CNN with Gradient Descent

Then, we use the following commands to train the CNN model:

```
# Train the model with gradient_descent
history_gradient_descent = model_gradient_descent.fit(X_train, y_train,
batch_size=32, epochs=10, validation_data=(X_val, y_val))
```

The loss and accuracy plots of training and validation data are shown in Figures 26 and 27. Also, the values of accuracy and loss for training, testing, and validation data are given in Table

22. Furthermore, the confusion matrices for testing and validation data are represented in Figure 28.

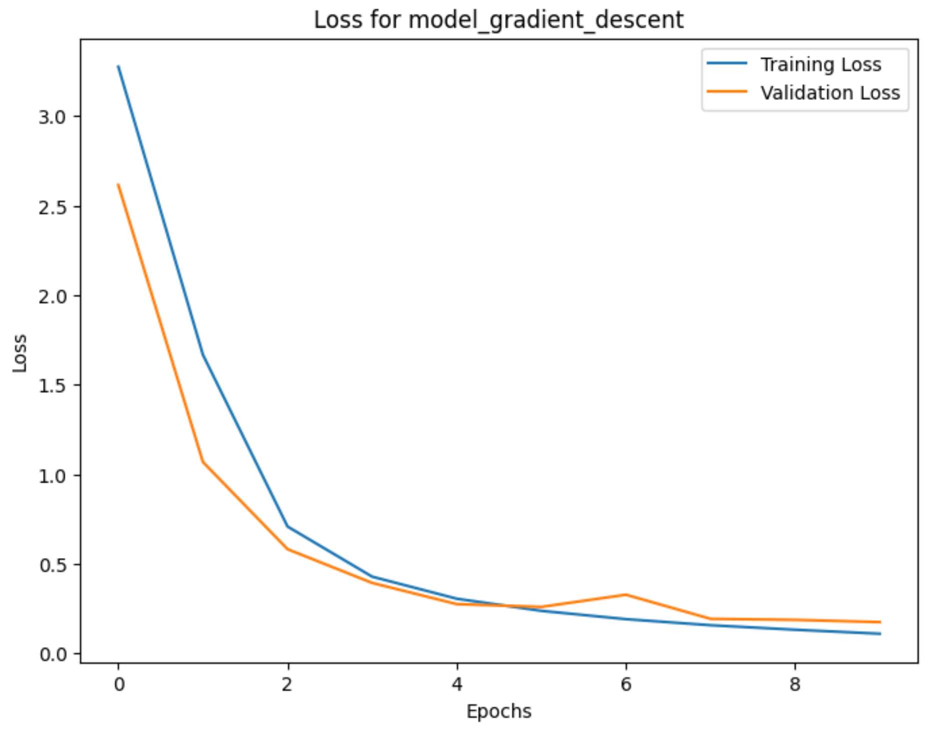


Figure 26. Loss of CNN with Gradient Descent

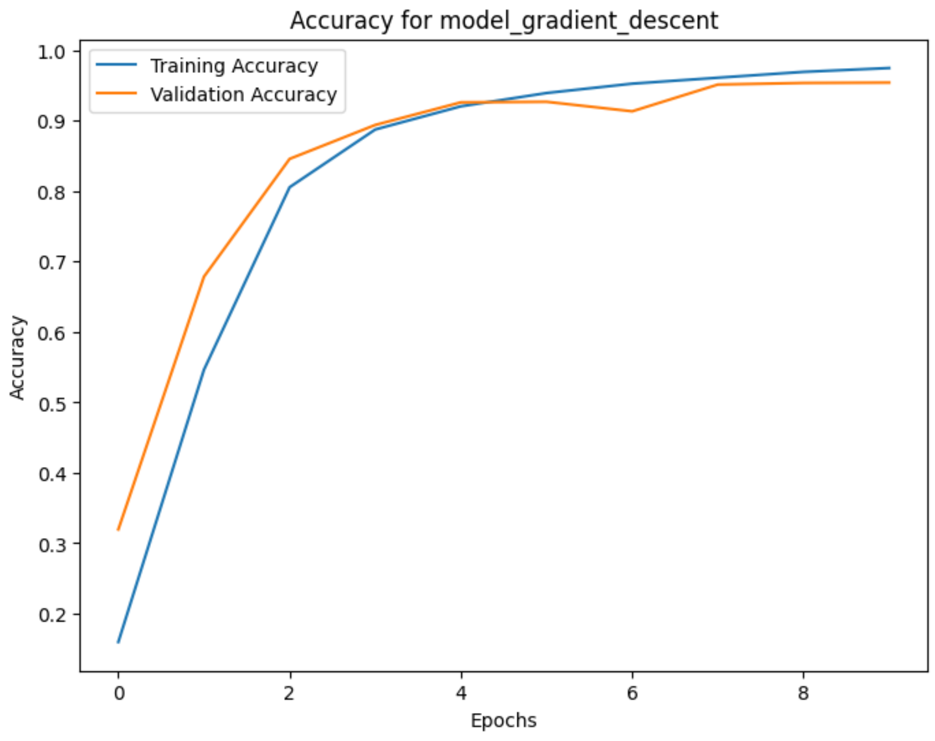
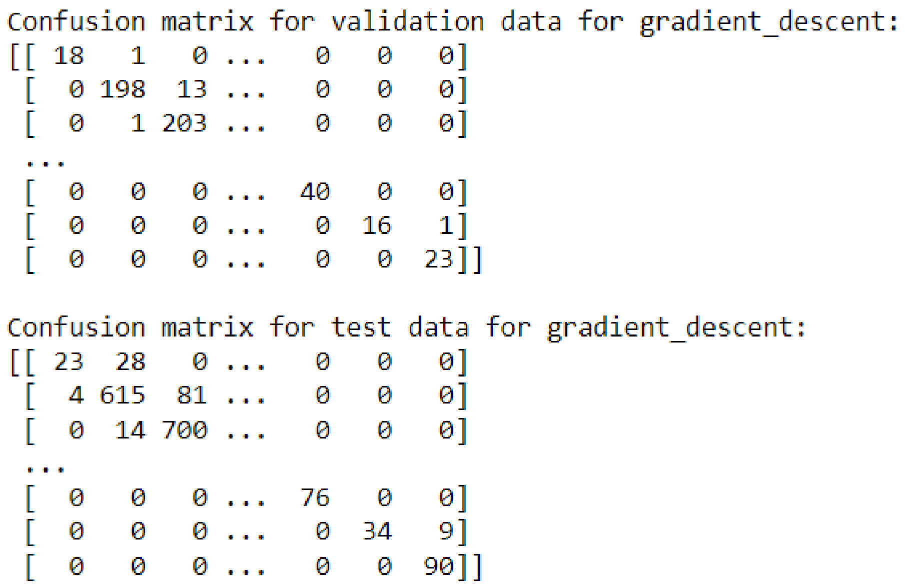


Figure 27. Accuracy of CNN with Gradient Descent

Table 22. Results of CNN model training with Gradient Descent

Dataset	Accuracy	Loss
Train	0.9750	0.1086
Test	0.8570	0.7722
Validation	0.9543	0.1734

The performance evaluation of the CNN deep neural network model resulted in the following outcomes: On the training dataset, the model achieved an accuracy of 0.9750 and a loss of 0.1086. For the test dataset, the accuracy obtained was 0.8570, accompanied by a loss of 0.7722. On the validation dataset, the model attained an accuracy of 0.9543 with a loss of 0.1734. These metrics provide insights into the model's performance in terms of accuracy and loss across different datasets. The performance of the model is acceptable with a high accuracy for train and validation datasets.



```
Confusion matrix for validation data for gradient_descent:
[[ 18   1   0 ...   0   0   0]
 [  0 198  13 ...   0   0   0]
 [  0   1 203 ...   0   0   0]
 ...
 [  0   0   0 ...  40   0   0]
 [  0   0   0 ...   0  16   1]
 [  0   0   0 ...   0   0  23]]

Confusion matrix for test data for gradient_descent:
[[ 23  28   0 ...   0   0   0]
 [  4 615  81 ...   0   0   0]
 [  0  14 700 ...   0   0   0]
 ...
 [  0   0   0 ...  76   0   0]
 [  0   0   0 ...   0  34   9]
 [  0   0   0 ...   0   0  90]]
```

Figure 28. Confusion matrix of CNN with Gradient Descent

4. Activation function

- Sigmoid

To train the CNN model with a Sigmoid activation function, the following properties specified in Table 23 are used to build the CNN architecture. Using the Sigmoid activation function, each neuron in the CNN model will produce an output between 0 and 1, allowing for non-linear transformations of the input data. The Adam optimizer is used to optimize the model's weights during training. The absence of Dropout implies that no regularization technique is applied to prevent overfitting. The loss function, sparse_categorical_crossentropy, is suitable for multi-class classification tasks like traffic sign recognition, where each input can belong to one of several classes. MaxPooling2D is applied to downsample the spatial dimensions of the feature maps, reducing computational complexity and extracting relevant features. The pool size of (2, 2) indicates that a 2x2 window is used for pooling. The output layer activation is set to Softmax, which assigns probabilities to each class, enabling the model to predict the class with the highest probability for classification purposes.

By incorporating these properties, the CNN model is built with the specified Sigmoid activation function, optimizer, loss function, pooling, and output layer settings.

Table 23. Properties CNN model with Sigmoid

Property	Value
Activation function	Sigmoid
Optimizer	Adam
Dropout	--
Loss function	sparse_categorical_crossentropy
Pooling type	MaxPooling2D
Pool size	(2, 2)
Output layer activation	Softmax

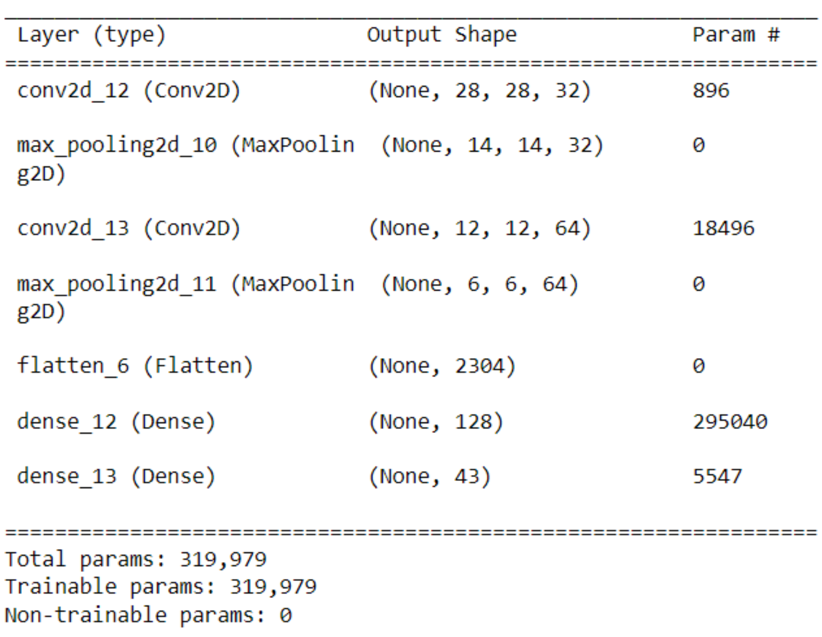
The following code represents the CNN model with the given properties. The summary of the model and its architecture are given in Table 24 and Figure 29, respectively.

```python
# Define the model with Sigmoid activation function
model_sigmoid = Sequential()
model_sigmoid.add(Conv2D(32, (3, 3), activation='sigmoid',
input_shape=(30, 30, 3)))
model_sigmoid.add(MaxPooling2D(pool_size=(2, 2)))
model_sigmoid.add(Conv2D(64, (3, 3), activation='sigmoid'))
model_sigmoid.add(MaxPooling2D(pool_size=(2, 2)))
model_sigmoid.add(Flatten())
model_sigmoid.add(Dense(128, activation='sigmoid'))
model_sigmoid.add(Dense(classes, activation='softmax'))

# Compile the model
model_sigmoid.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Print the model summary
model_sigmoid.summary()
```

Table 24. CNN model with Sigmoid summary



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_12 (Conv2D)          (None, 28, 28, 32)        896

 max_pooling2d_10 (MaxPoolin  (None, 14, 14, 32)       0
 g2D)

 conv2d_13 (Conv2D)          (None, 12, 12, 64)        18496

 max_pooling2d_11 (MaxPoolin  (None, 6, 6, 64)         0
 g2D)

 flatten_6 (Flatten)         (None, 2304)              0

 dense_12 (Dense)            (None, 128)               295040

 dense_13 (Dense)            (None, 43)                5547

=================================================================
Total params: 319,979
Trainable params: 319,979
Non-trainable params: 0
```

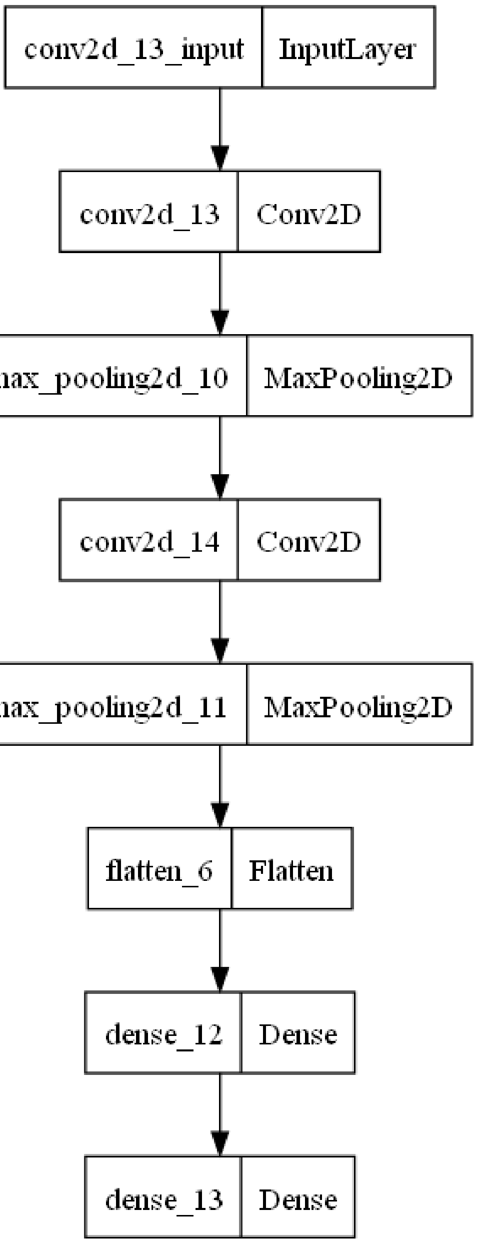

conv2d_13_input | InputLayer
conv2d_13 | Conv2D
max_pooling2d_10 | MaxPooling2D
conv2d_14 | Conv2D
max_pooling2d_11 | MaxPooling2D
flatten_6 | Flatten
dense_12 | Dense
dense_13 | Dense

Figure 29. Architecture of CNN with Sigmoid

Then, we use the following commands to train the CNN model:

```
# Train the model with Sigmoid activation function
history_sigmoid = model_sigmoid.fit(X_train, y_train, batch_size=32,
epochs=10, validation_data=(X_val, y_val))
```

The loss and accuracy plots of training and validation data are shown in Figures 30 and 31. Also, the values of accuracy and loss for training, testing, and validation data are given in Table 25. Furthermore, the confusion matrices for testing and validation data are represented in Figure 32.

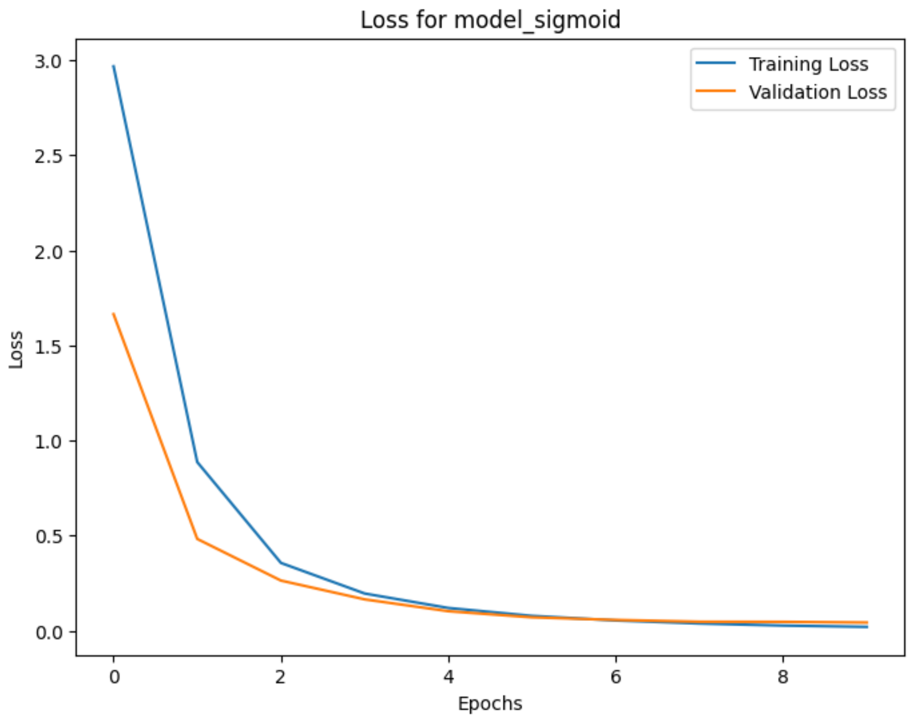


Figure 30. Loss of CNN neural network with Sigmoid

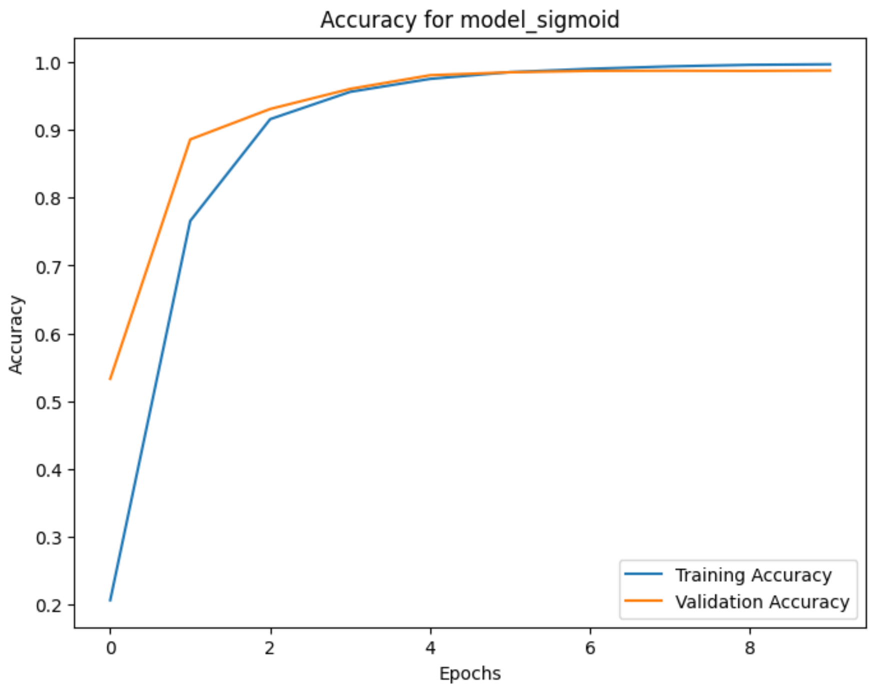


Figure 31. Accuracy of CNN with Sigmoid

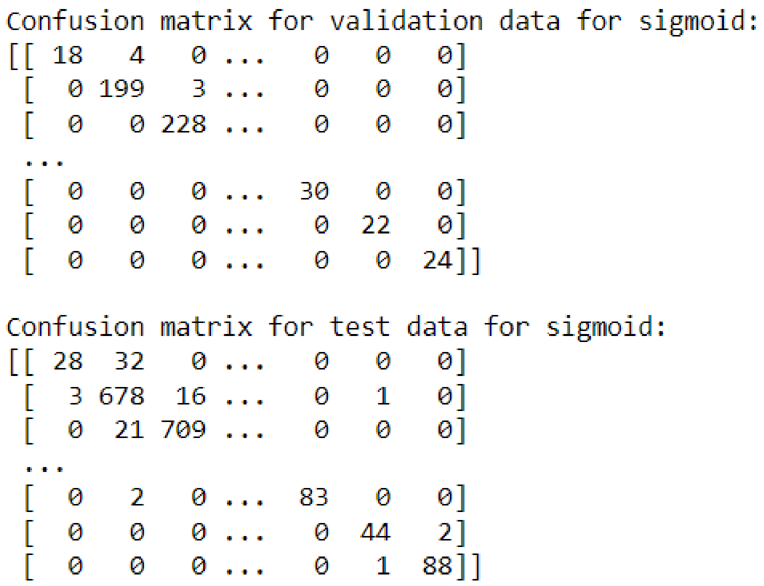
Table 25. Results of CNN model training with Sigmoid

Dataset	Accuracy	Loss
Train	0.9965	0.0198
Test	0.9144	0.3268
Validation	0.9872	0.0426

The CNN deep neural network model trained with the specified properties achieved the following performance results:

- Train dataset: The model achieved an accuracy of 99.65% and a loss of 0.0198. This indicates that the model performed very well on the training data, accurately predicting the classes of the traffic signs with low loss.
- Test dataset: The model achieved an accuracy of 91.44% and a loss of 0.3268. This indicates that the model generalized well to unseen test data, although the accuracy is slightly lower compared to the training accuracy. The loss value suggests that the model's predictions had a moderate level of uncertainty or incorrectness on the test set.
- Validation dataset: The model achieved an accuracy of 98.72% and a loss of 0.0426 on the validation dataset. These results indicate that the model performed well on the validation data, accurately classifying the traffic signs with low loss.

Overall, the model demonstrates good performance, with high accuracy and relatively low loss on both the training and validation datasets. However, there is a slight drop in accuracy on the test set compared to the training and validation sets, suggesting the possibility of some overfitting. Further analysis and evaluation can be done to fine-tune the model and improve its generalization capabilities.



```
Confusion matrix for validation data for sigmoid:
[[ 18   4   0 ...   0   0   0]
 [  0 199   3 ...   0   0   0]
 [  0   0 228 ...   0   0   0]
 ...
 [  0   0   0 ...  30   0   0]
 [  0   0   0 ...   0  22   0]
 [  0   0   0 ...   0   0  24]]

Confusion matrix for test data for sigmoid:
[[ 28  32   0 ...   0   0   0]
 [  3 678  16 ...   0   1   0]
 [  0  21 709 ...   0   0   0]
 ...
 [  0   2   0 ...  83   0   0]
 [  0   0   0 ...   0  44   2]
 [  0   0   0 ...   0   1  88]]
```

Figure 32. Confusion matrix of CNN with Sigmoid

- Relu

As shown in Table 26, the CNN model was built with the Relu activation function, Adam optimizer, and sparse_categorical_crossentropy loss function. MaxPooling2D with a pool size of (2, 2) was used for downsampling. The output layer was activated with softmax.

Table 26. Properties CNN model with Relu

Property	Value
Activation function	Relu
Optimizer	Adam
Dropout	--
Loss function	sparse_categorical_crossentropy
Pooling type	MaxPooling2D
Pool size	(2, 2)
Output layer activation	Softmax

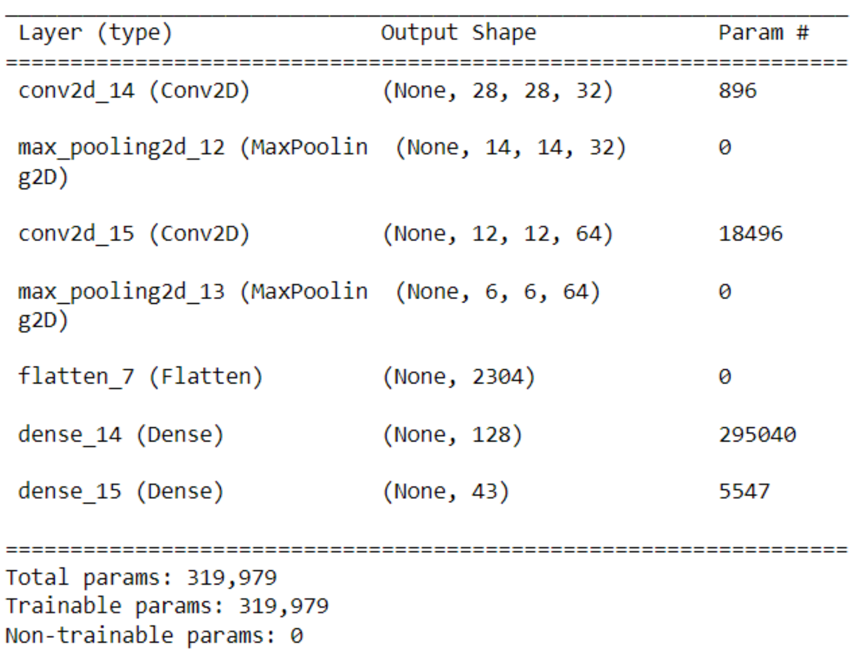
The following code represents the CNN model with the given properties. The summary of the model and its architecture are given in Table 27 and Figure 33, respectively.

```
# Define the model with ReLU activation function
model_relu = Sequential()
model_relu.add(Conv2D(32, (3, 3), activation='relu', input_shape=(30,
30, 3)))
model_relu.add(MaxPooling2D(pool_size=(2, 2)))
model_relu.add(Conv2D(64, (3, 3), activation='relu'))
model_relu.add(MaxPooling2D(pool_size=(2, 2)))
model_relu.add(Flatten())
model_relu.add(Dense(128, activation='relu'))
model_relu.add(Dense(classes, activation='softmax'))

# Compile the model
model_relu.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Print the model summary
model_relu.summary()
```

Table 27. CNN model with Relu summary



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_14 (Conv2D)          (None, 28, 28, 32)        896

 max_pooling2d_12 (MaxPoolin  (None, 14, 14, 32)       0
 g2D)

 conv2d_15 (Conv2D)          (None, 12, 12, 64)        18496

 max_pooling2d_13 (MaxPoolin  (None, 6, 6, 64)         0
 g2D)

 flatten_7 (Flatten)         (None, 2304)              0

 dense_14 (Dense)            (None, 128)               295040

 dense_15 (Dense)            (None, 43)                5547

=================================================================
Total params: 319,979
Trainable params: 319,979
Non-trainable params: 0
```

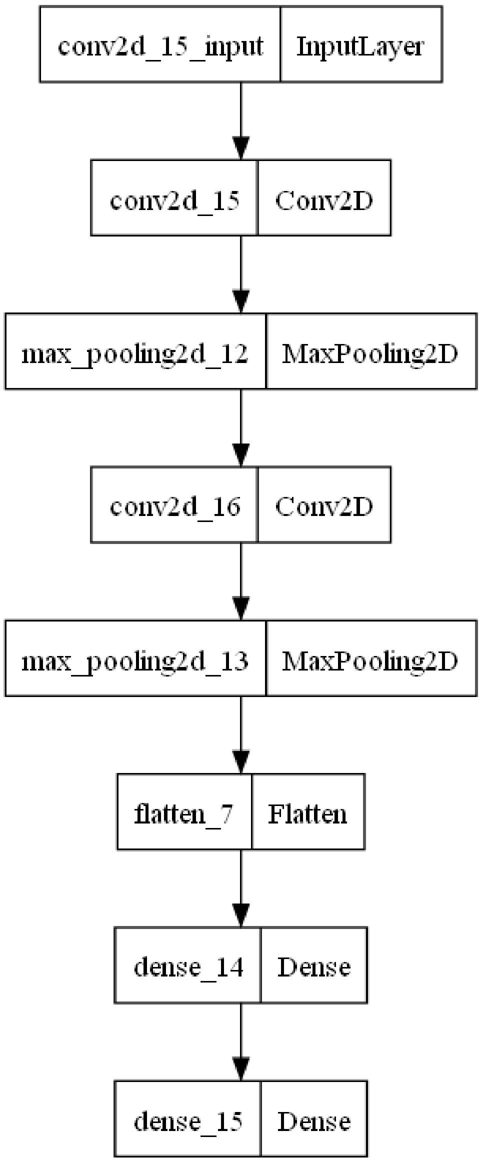

conv2d_15_input | InputLayer
conv2d_15 | Conv2D
max_pooling2d_12 | MaxPooling2D
conv2d_16 | Conv2D
max_pooling2d_13 | MaxPooling2D
flatten_7 | Flatten
dense_14 | Dense
dense_15 | Dense

Figure 33. Architecture of CNN with Relu

Then, we use the following commands to train the CNN model:

```
# Train the model with relu activation function
history_relu = model_relu.fit(X_train, y_train, batch_size=32,
epochs=10, validation_data=(X_val, y_val))
```

The loss and accuracy plots of training and validation data are shown in Figures 34 and 35. Also, the values of accuracy and loss for training, testing, and validation data are given in Table 28. Furthermore, the confusion matrices for testing and validation data are represented in Figure 36.

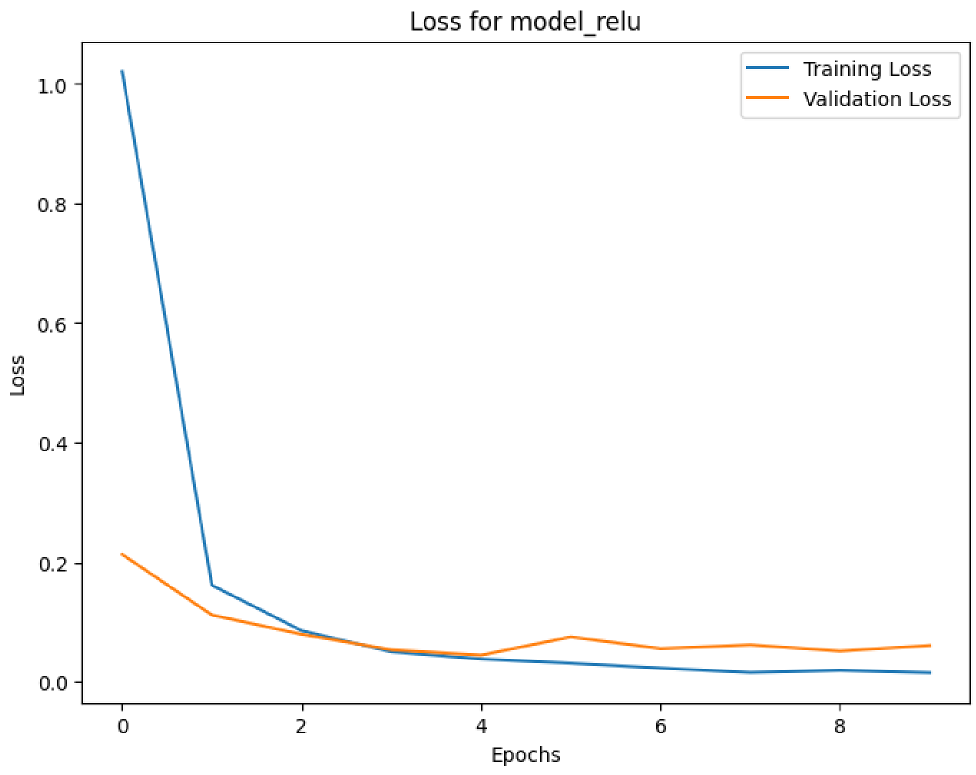


Figure 34. Loss of CNN with Relu

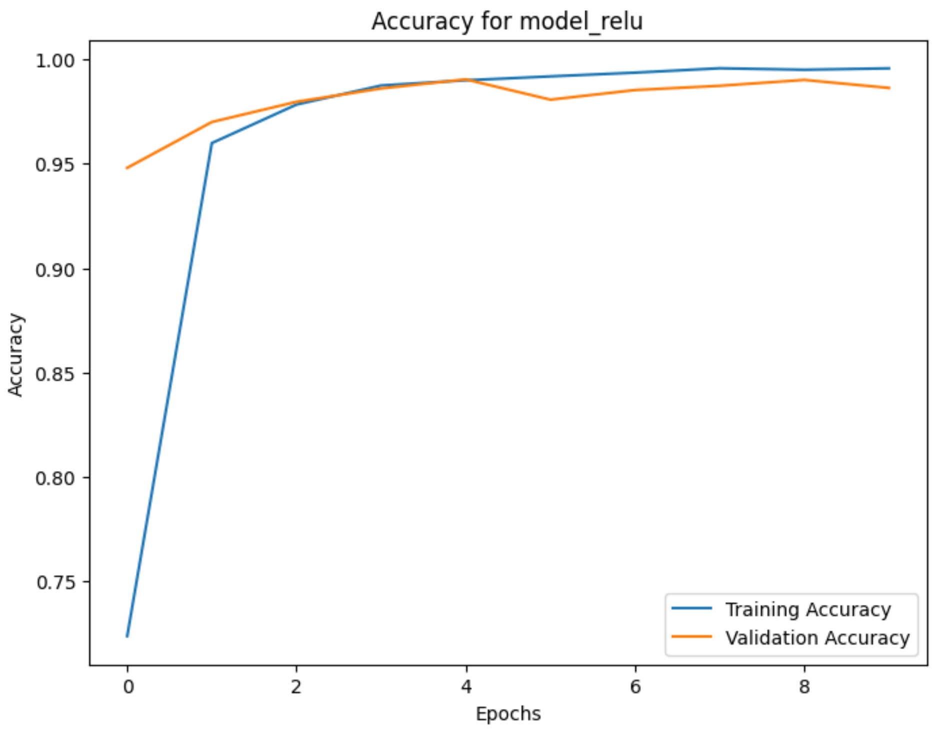
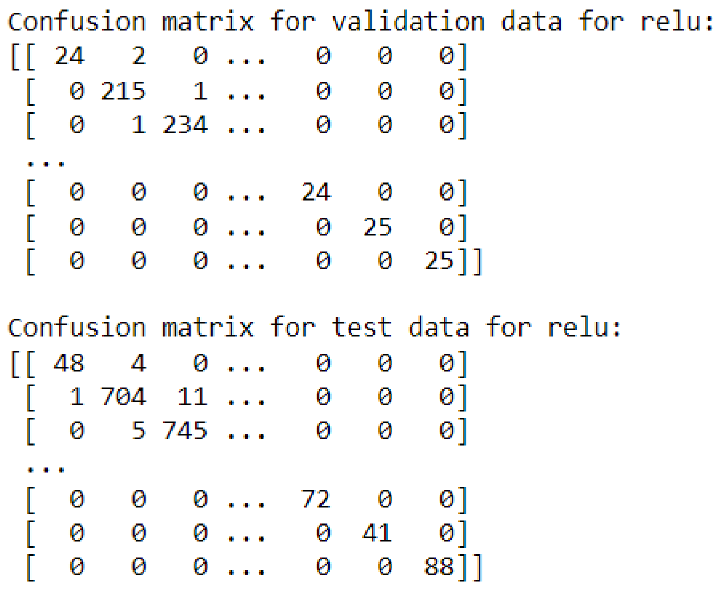


Figure 35. Accuracy of CNN with Relu

Table 28. Results of CNN model training with Relu

Dataset	Accuracy	Loss
Train	0.9956	0.0152
Test	0.9288	0.4529
Validation	0.9862	0.0605

The CNN deep neural network model achieved an accuracy of 0.9956 on the training dataset with a loss of 0.0152. On the test dataset, the model achieved an accuracy of 0.9288 with a loss of 0.4529. The validation dataset showed an accuracy of 0.9862 with a loss of 0.0605. These results indicate that the model performed well, achieving high accuracy and relatively low loss values on both the test and validation datasets.



```
Confusion matrix for validation data for relu:
[[ 24   2   0 ...   0   0   0]
 [  0 215   1 ...   0   0   0]
 [  0   1 234 ...   0   0   0]
 ...
 [  0   0   0 ...  24   0   0]
 [  0   0   0 ...   0  25   0]
 [  0   0   0 ...   0   0  25]]

Confusion matrix for test data for relu:
[[ 48   4   0 ...   0   0   0]
 [  1 704  11 ...   0   0   0]
 [  0   5 745 ...   0   0   0]
 ...
 [  0   0   0 ...  72   0   0]
 [  0   0   0 ...   0  41   0]
 [  0   0   0 ...   0   0  88]]
```

Figure 36. Confusion matrix of CNN with Relu

- Tanh

To train the CNN model with the Tanh activation function, the following properties were used to build the CNN architecture as shown in Table 29. The activation function for the hidden layers was set to Tanh, which provides a smooth non-linear transformation of the input data. The optimizer used was Adam, a popular optimization algorithm that adapts the learning rate dynamically. No dropout regularization was applied, meaning that no neurons were randomly dropped during training to prevent overfitting. The loss function used was sparse_categorical_crossentropy, suitable for multi-class classification tasks. MaxPooling2D was used as the pooling type with a pool size of (2, 2), which reduces the spatial dimensions of the feature maps. The output layer had a softmax activation function to produce class probabilities for classification.

Table 29. Properties CNN model with Tanh

Property	Value
Activation function	Tanh
Optimizer	Adam
Dropout	--
Loss function	sparse_categorical_crossentropy
Pooling type	MaxPooling2D
Pool size	(2, 2)
Output layer activation	Softmax

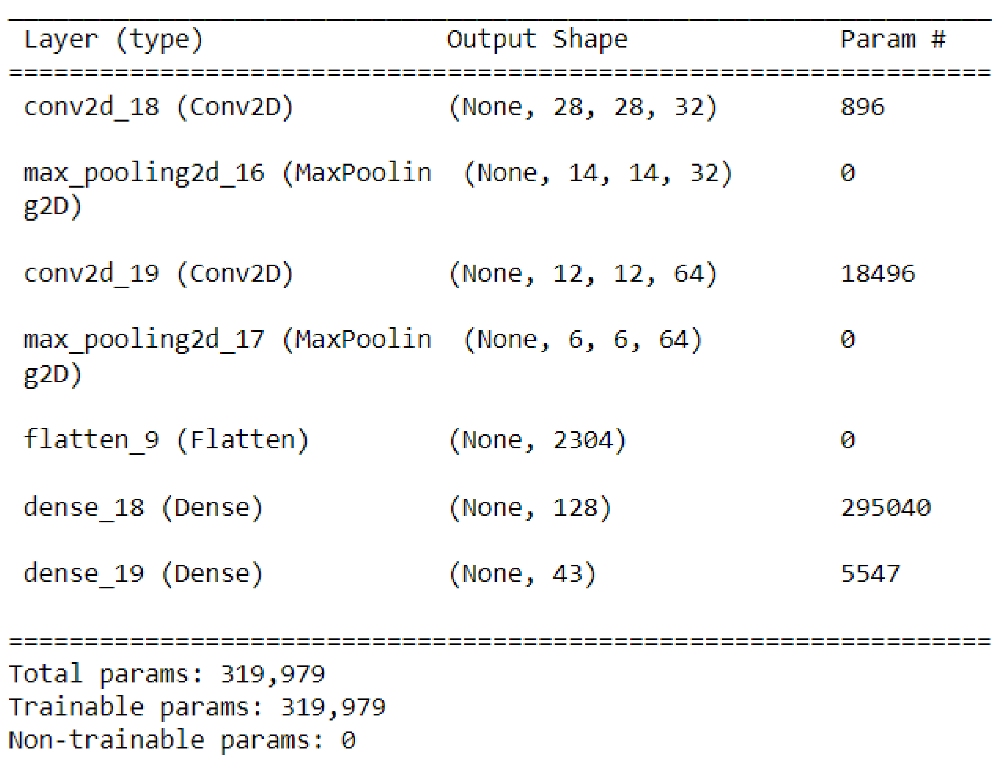
The following code represents the CNN model with the given properties. The summary of the model and its architecture are given in Table 30 and Figure 37, respectively.

```
# Define the model with Tanh activation function
model_tanh = Sequential()
model_tanh.add(Conv2D(32, (3, 3), activation='tanh', input_shape=(30,
30, 3)))
model_tanh.add(MaxPooling2D(pool_size=(2, 2)))
model_tanh.add(Conv2D(64, (3, 3), activation='tanh'))
model_tanh.add(MaxPooling2D(pool_size=(2, 2)))
model_tanh.add(Flatten())
model_tanh.add(Dense(128, activation='tanh'))
model_tanh.add(Dense(classes, activation='softmax'))

# Compile the model
model_tanh.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
# Print the model summary
model_tanh.summary()
```

Table 30. CNN model with Tanh summary



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_18 (Conv2D)          (None, 28, 28, 32)        896

 max_pooling2d_16 (MaxPoolin  (None, 14, 14, 32)       0
 g2D)

 conv2d_19 (Conv2D)          (None, 12, 12, 64)        18496

 max_pooling2d_17 (MaxPoolin  (None, 6, 6, 64)         0
 g2D)

 flatten_9 (Flatten)         (None, 2304)              0

 dense_18 (Dense)            (None, 128)               295040

 dense_19 (Dense)            (None, 43)                5547

=================================================================
Total params: 319,979
Trainable params: 319,979
Non-trainable params: 0
```

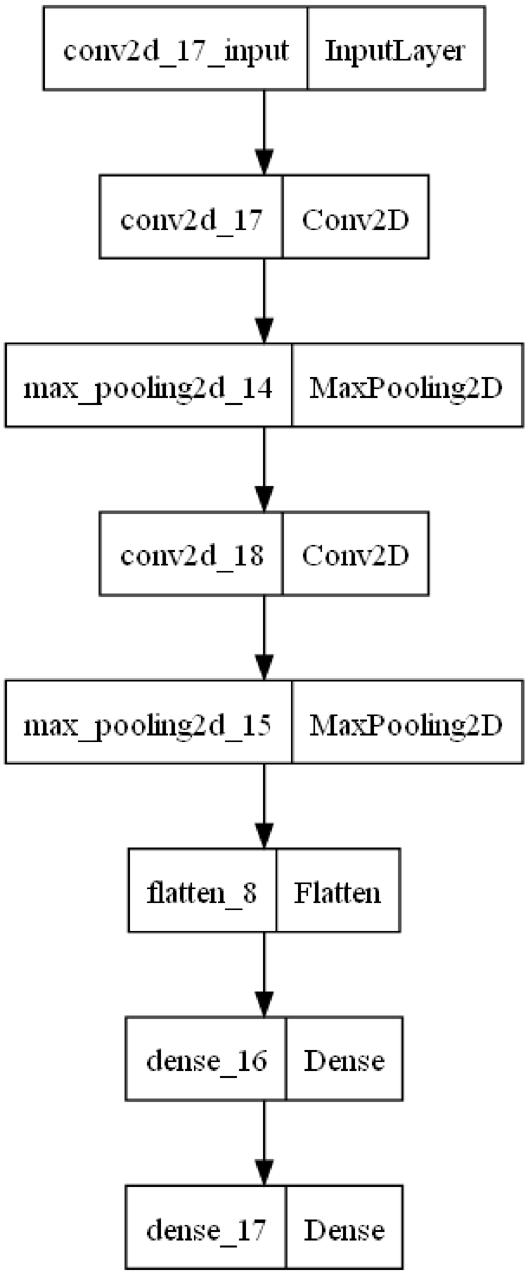

conv2d_17_input | InputLayer
conv2d_17 | Conv2D
max_pooling2d_14 | MaxPooling2D
conv2d_18 | Conv2D
max_pooling2d_15 | MaxPooling2D
flatten_8 | Flatten
dense_16 | Dense
dense_17 | Dense

Figure 37. Architecture of CNN with Tanh

Then, we use the following commands to train the CNN model:

```
# Train the model with Tanh activation function
history_tanh = model_tanh.fit(X_train, y_train, batch_size=32,
epochs=10, validation_data=(X_val, y_val))
```

The loss and accuracy plots of training and validation data are shown in Figures 38 and 39. Also, the values of accuracy and loss for training, testing, and validation data are given in Table

31. Furthermore, the confusion matrices for testing and validation data are represented in Figure 40.

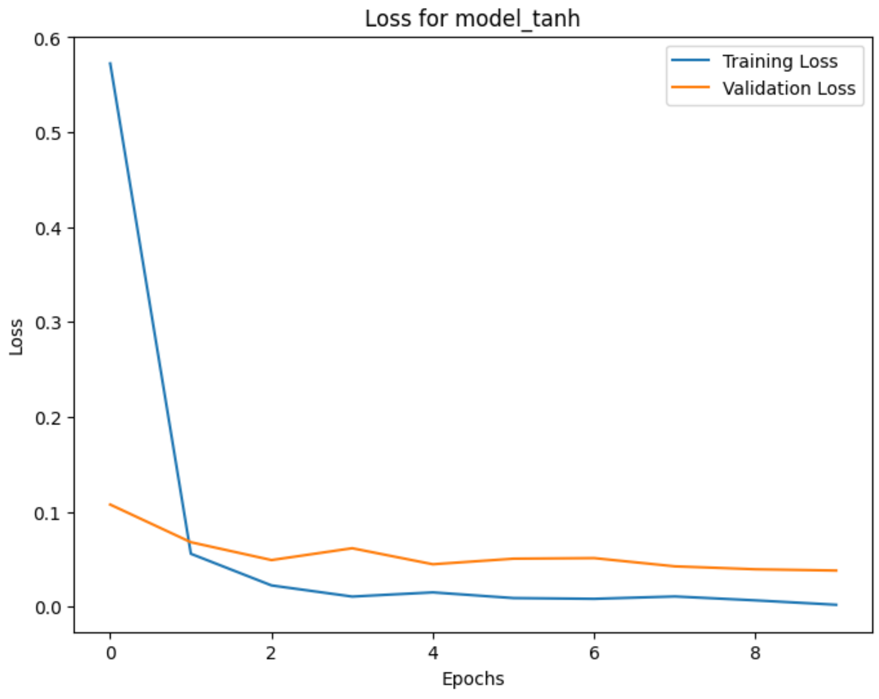


Figure 38. Loss of CNN with Tanh

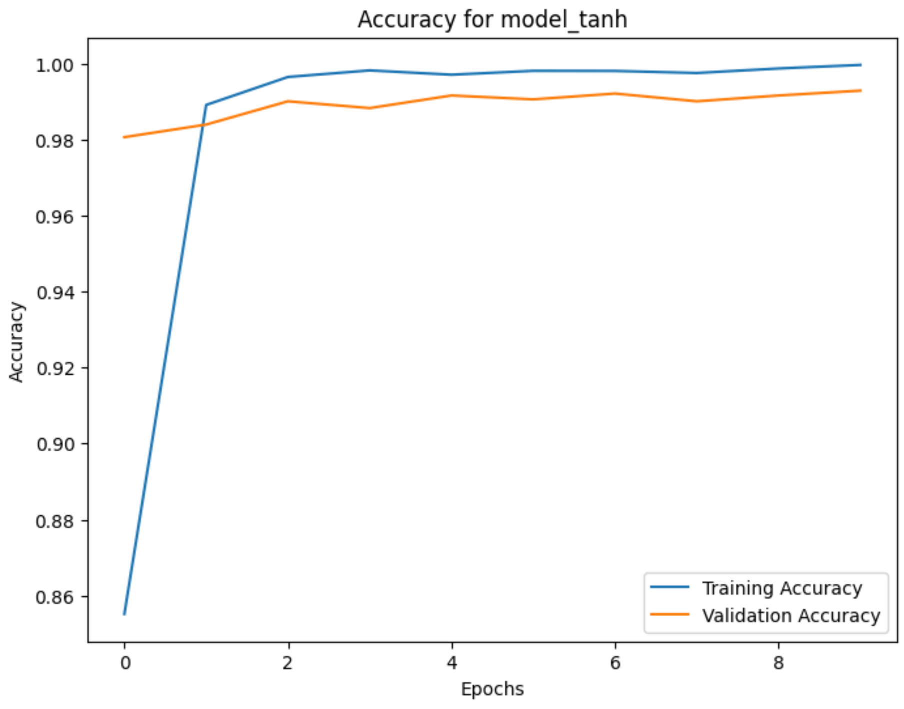
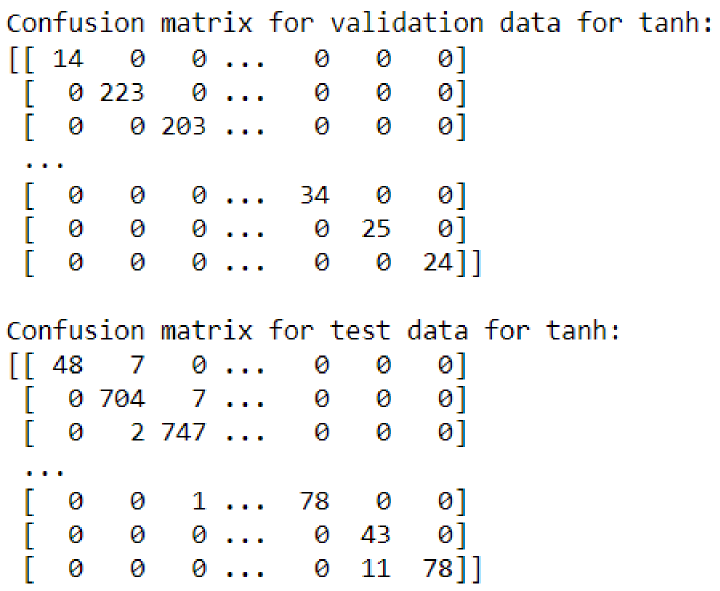


Figure 39. Accuracy of CNN with Tanh

Table 31. Results of CNN model training with Tanh

Dataset	Accuracy	Loss
Train	0.9996	0.0020
Test	0.9490	0.1960
Validation	0.9928	0.0381

The CNN deep neural network model achieved high accuracy on the training dataset, with an accuracy of 0.9996 and a low loss of 0.0020. This indicates that the model was able to effectively learn and generalize patterns from the training data. On the test dataset, the model achieved an accuracy of 0.9490 and a loss of 0.1960, which suggests that the model performed well in classifying unseen data. The validation dataset also showed good performance, with an accuracy of 0.9928 and a loss of 0.0381, indicating that the model was able to generalize well to new data samples. These results indicate that the CNN model with the given configuration was able to effectively learn and classify the traffic sign images with high accuracy and low loss.



```
Confusion matrix for validation data for tanh:
[[ 14   0   0 ...   0   0   0]
 [  0 223   0 ...   0   0   0]
 [  0   0 203 ...   0   0   0]
 ...
 [  0   0   0 ...  34   0   0]
 [  0   0   0 ...   0  25   0]
 [  0   0   0 ...   0   0  24]]

Confusion matrix for test data for tanh:
[[ 48   7   0 ...   0   0   0]
 [  0 704   7 ...   0   0   0]
 [  0   2 747 ...   0   0   0]
 ...
 [  0   0   1 ...  78   0   0]
 [  0   0   0 ...   0  43   0]
 [  0   0   0 ...   0  11  78]]
```

Figure 40. Confusion matrix of CNN with Tanh

iii. Data augmentation

Data augmentation is a technique commonly used in deep learning to increase the size and diversity of a training dataset. It involves applying various transformations or modifications to the existing training data to create new, slightly altered samples. These augmented samples are then used alongside the original data during the training process. The primary goal of data augmentation is to introduce variability into the training data, which helps improve the generalization and robustness of the deep learning model. By exposing the model to a wider range of variations in the data, it becomes better equipped to handle different scenarios, such as varying lighting conditions, different viewpoints, or other potential sources of variability in real-world data.

Data augmentation techniques in deep learning:

- **Geometric transformations:** These include rotations, translations, scaling, and flips. For example, an image can be rotated by a certain angle, shifted horizontally or vertically, resized, or mirrored.
- **Image cropping and padding:** Randomly cropping or padding an image can help the model learn to focus on different regions of interest. It can also simulate different aspect ratios or adjust the image size.
- **Color and contrast adjustments:** Altering the color and contrast of images can make the model more robust to variations in lighting conditions. These adjustments may involve changes to brightness, saturation, hue, or contrast.
- **Gaussian noise:** Adding random Gaussian noise to an image can help the model become more tolerant to noise in the input data.
- **Elastic deformations:** Applying elastic deformations to images simulates small deformations that can occur due to the object's shape or imaging conditions. It helps the model learn to be more invariant to small distortions.
- **Cutout and dropout:** Cutout involves masking out random regions of an image, forcing the model to focus on other features. Dropout is a similar technique but applied to the model's internal layers, randomly dropping out certain units during training.

By applying these transformations to the training data, the augmented dataset effectively becomes larger, allowing the model to see more diverse examples. This increased variation helps prevent overfitting and encourages the model to learn more generalizable features. It's important to note that data augmentation should be applied judiciously, considering the specific characteristics of the dataset and the task at hand. Careful selection and application of augmentation techniques can greatly enhance the performance and generalization of deep learning models.

Importance:

Data augmentation in deep learning is important because it:

- Increases dataset size without additional data collection.
- Improves generalization by exposing the model to diverse variations.
- Helps prevent overfitting and adds regularization to the model.
- Addresses class imbalance by balancing the distribution of samples.
- Enhances the model's robustness to real-world input variations.
- Reduces the need for extensive data collection efforts.

What augmentations:

When training a Convolutional Neural Network (CNN) to classify traffic signs, it's important to consider the nature of the problem and choose augmentations that are suitable. Given that some augmentations may cause the image of a traffic sign to be classified incorrectly, it is crucial to be cautious in the selection of augmentations.

Here are some augmentations that are generally suitable for our problem (traffic sign classification):

- Geometric transformations: These can include random rotations, translations, and scaling within certain limits. However, extreme rotations or distortions that significantly alter the shape or readability of the traffic sign should be avoided, as they may lead to misclassifications.
- Color and contrast adjustments: Modifying the brightness, saturation, or contrast of the images can be useful for making the model more robust to variations in lighting conditions. However, extreme adjustments that distort the colors of the traffic sign should be avoided, as they might affect the accuracy of the classification.

- Image cropping and padding: Randomly cropping or padding the images can provide the model with different perspectives and aspect ratios. However, it's essential to ensure that the traffic sign remains intact and identifiable within the cropped or padded image.
- Gaussian noise: Adding a small amount of Gaussian noise to the images can help the model become more tolerant to noise in the input data. However, excessive noise that obscures the traffic sign's details should be avoided.
- Horizontal flipping: Flipping the images horizontally can provide additional training samples and help the model learn from different viewpoints. Since most traffic signs are symmetrical, horizontal flipping is generally acceptable.

It is important to carefully validate the effects of data augmentation on the classification performance. Augmentations that significantly alter the appearance of the traffic sign, obscure important details, or introduce misleading information should be avoided. Regular evaluation and testing can help identify which augmentations are beneficial for improving the model's performance on traffic sign classification while maintaining classification accuracy.

In this section, we will use the following properties and augmentations given in Table 32.

Table 32. Properties CNN model using Data Augmentation

Property	Value
Activation function	Tanh
Optimizer	Adam
Dropout	--
Loss function	sparse_categorical_crossentropy
Pooling type	MaxPooling2D
Pool size	(2, 2)
Output layer activation	Softmax
Rotation range	10
Width shift range	0.1
Height shift range	0.1
Shear range	0.2
Zoom range	0.2
Horizontal flip	True

The following code demonstrates the utilization of data augmentation in training our deep learning model for image classification. To begin, an instance of the ImageDataGenerator class is created with specific data augmentation parameters. These parameters define the augmentations that will be applied to the training data during the training process. The augmentations include rotation, horizontal and vertical shifting, shearing, zooming, and horizontal flipping.

```
# Create an instance of the ImageDataGenerator with data augmentation
parameters
datagen = ImageDataGenerator(
    rotation_range=10,  # Rotate images randomly by up to 10 degrees
    width_shift_range=0.1,  # Shift images horizontally by up to 10% of
the width
    height_shift_range=0.1,  # Shift images vertically by up to 10% of
the height
    shear_range=0.2,  # Shear images by up to 20%
    zoom_range=0.2,  # Zoom in on images by up to 20%
    horizontal_flip=True  # Flip images horizontally
)

# Fit the data generator on the training data
datagen.fit(X_train)

# Define the model with Tanh activation function
model_tanh_aug = Sequential()
model_tanh_aug.add(Conv2D(32, (3, 3), activation='tanh',
input_shape=(30, 30, 3)))
model_tanh_aug.add(MaxPooling2D(pool_size=(2, 2)))
model_tanh_aug.add(Conv2D(64, (3, 3), activation='tanh'))
model_tanh_aug.add(MaxPooling2D(pool_size=(2, 2)))
model_tanh_aug.add(Flatten())
model_tanh_aug.add(Dense(128, activation='tanh'))
model_tanh_aug.add(Dense(classes, activation='softmax'))

# Compile the model
model_tanh_aug.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])


# Train the model with data augmentation
history_tanh_aug = model_tanh_aug.fit(datagen.flow(X_train, y_train,
batch_size=32),
                              steps_per_epoch=len(X_train) / 32,
epochs=10,
                              validation_data=(X_val, y_val))
```

In the mentioned code, the ImageDataGenerator instance is then fitted on the training data (X_train) using the fit() method. This prepares the generator to apply the specified augmentations to the training data dynamically. By fitting the data generator, it becomes ready to generate augmented data on the fly during training, expanding the effective size of the dataset and introducing variability into the training process. Next, our CNN model (model_tanh_aug) is defined using the Sequential API from Keras. This model consists of several layers, including convolutional layers, max pooling layers, a flatten layer, and dense layers. The activation function used in this model is the hyperbolic tangent (Tanh) function (we chose the model with Tanh activation function because it had the highest accuracy of 0.9996). The model architecture is designed to capture relevant features from the input images and make predictions based on those features.

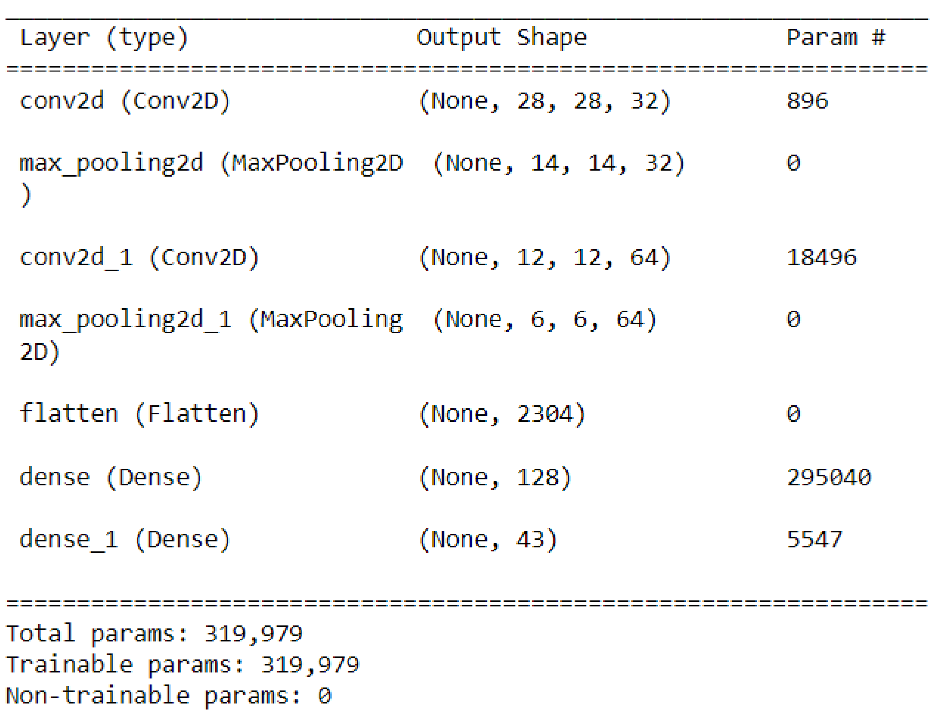
After defining the model, it is compiled with the Adam optimizer. The sparse categorical cross-entropy loss function is specified as the loss metric, suitable for multi-class classification tasks. Additionally, the accuracy metric is defined to evaluate the model's performance during training.

The model is then trained using the fit() method. However, instead of directly passing the raw training data (X_train and y_train), the datagen.flow() function is used. This function generates augmented data batches on the fly by applying the specified augmentations to the training data. The batch size is set to 32, and the number of steps per epoch is determined based on the length of X_train divided by the batch size. This allows for efficient and continuous augmentation during training. During the training process, the model receives the augmented data batches and undergoes the specified augmentations, which enrich the training data with variations and enhance the model's ability to generalize and handle different variations and conditions in real-world scenarios.

Finally, the training history is stored in the history_tanh_aug variable, which captures information such as loss and accuracy values at each epoch. This history can be used for analysis and visualizations to assess the model's performance and monitor its training progress.

The summary of the model and its architecture are given in Table 33 and Figure 41, respectively.

Table 33. CNN model with using Data Augmentation summary



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 32)        896

 max_pooling2d (MaxPooling2D  (None, 14, 14, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 12, 12, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 flatten (Flatten)           (None, 2304)              0

 dense (Dense)               (None, 128)               295040

 dense_1 (Dense)             (None, 43)                5547

=================================================================
Total params: 319,979
Trainable params: 319,979
Non-trainable params: 0
```

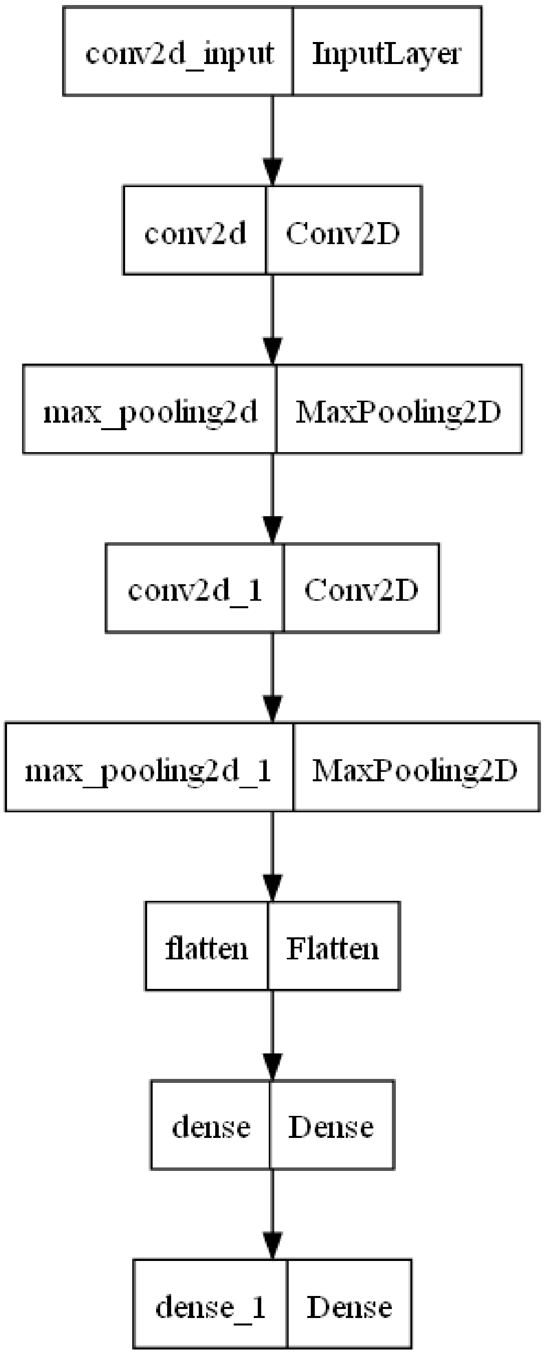


Figure 41. Architecture of CNN using Data Augmentation

The loss and accuracy plots of training and validation data are shown in Figures 42 and 43. Also, the values of accuracy and loss for training, testing, and validation data are given in Table 34. Furthermore, he confusion matrices for testing and validation data are represented in Figure 44.

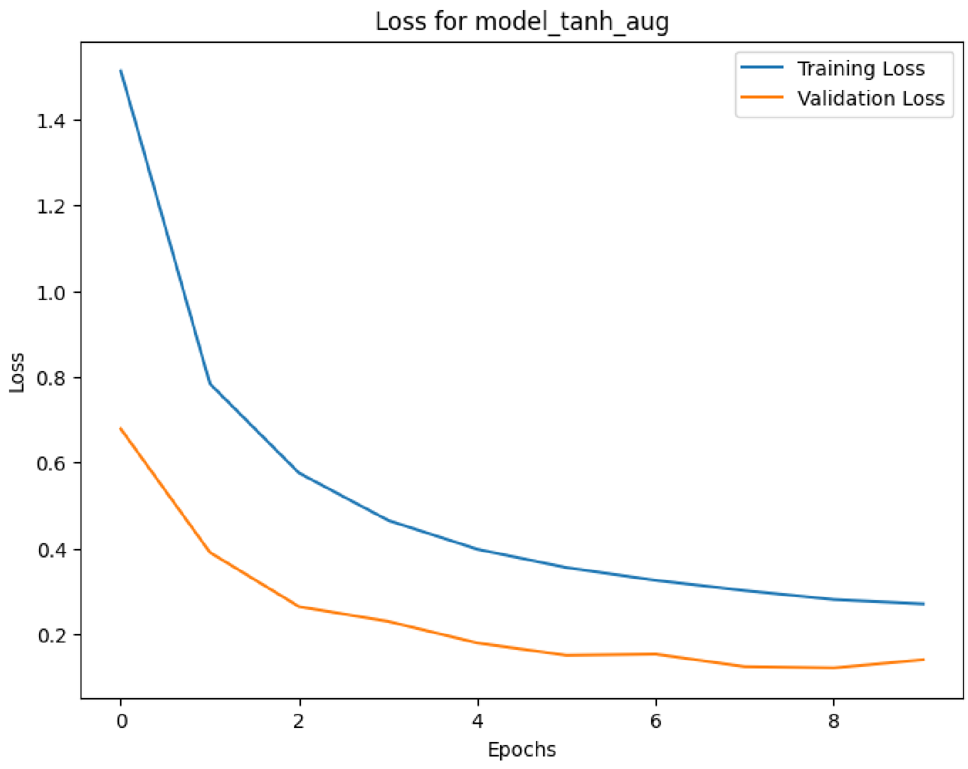


Figure 42. Loss of CNN using Data Augmentation

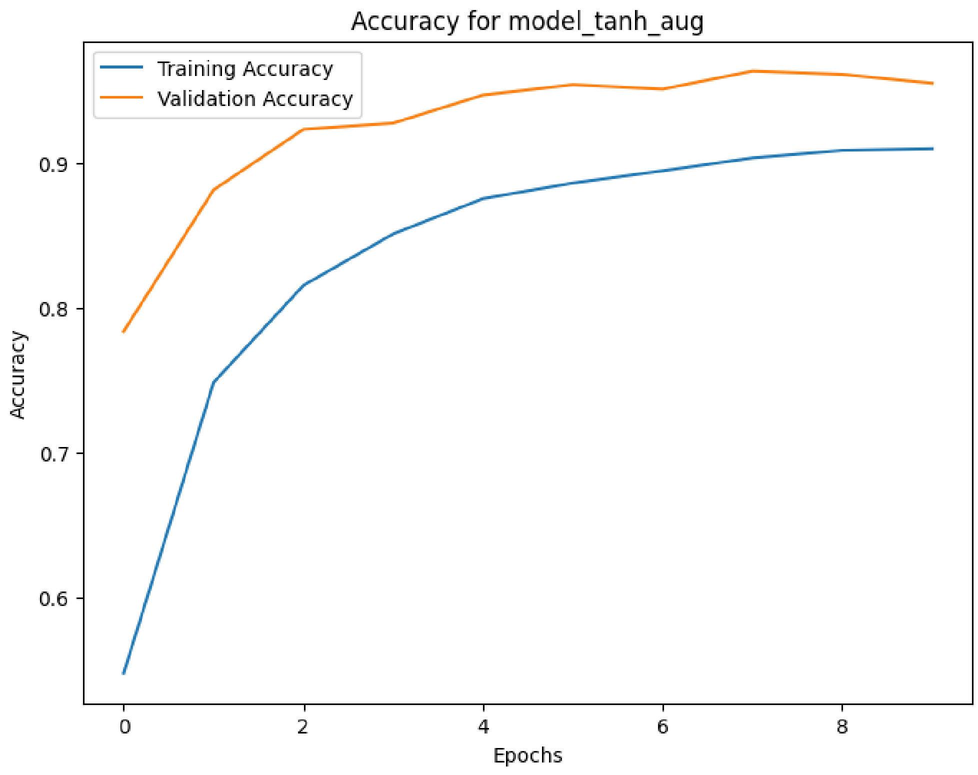


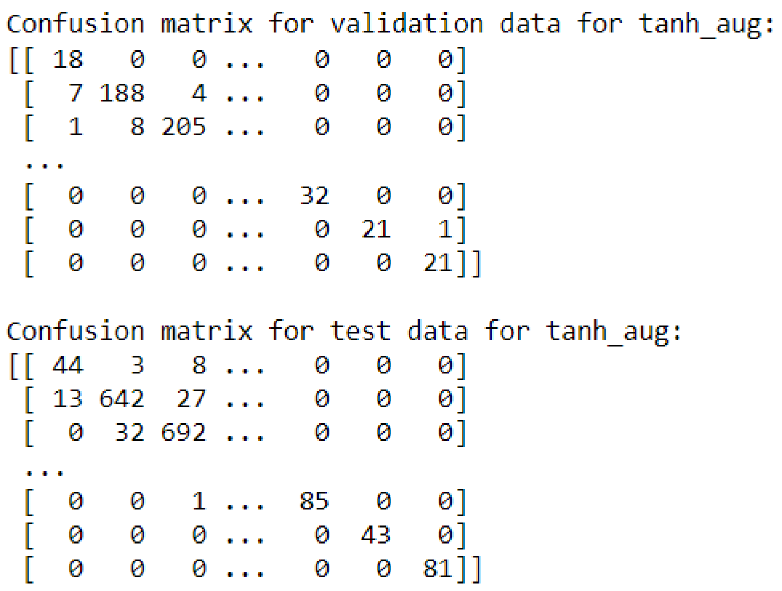
Figure 43. Accuracy of CNN using Data Augmentation

Table 34. Results of CNN model training using Data Augmentation

Dataset	Accuracy	Loss
Train	0.9104	0.2711
Test	0.8771	0.3886
Validation	0.9556	0.1408

The CNN deep neural network model, after applying data augmentation techniques, achieved an accuracy of 0.9104 on the training dataset with a corresponding loss of 0.2711. This indicates that the model was able to learn and classify the augmented training data with a reasonable level of accuracy and a moderate loss. On the test dataset, the model achieved an accuracy of 0.8771 and a loss of 0.3886. These results suggest that the model performed reasonably well in classifying unseen augmented test data, although the accuracy is slightly lower compared to the training dataset. For the validation dataset, the model achieved an accuracy of 0.9556 and a loss of 0.1408. This indicates that the model was able to generalize well to new augmented validation data.

So, the performance of the CNN model after data augmentation is relatively good, with satisfactory accuracy and loss values on both the training and validation datasets. The slightly lower accuracy on the test dataset suggests that there might be some overfitting, but the model still demonstrates a decent ability to classify augmented traffic sign images.



```
Confusion matrix for validation data for tanh_aug:
[[ 18   0   0 ...   0   0   0]
 [  7 188   4 ...   0   0   0]
 [  1   8 205 ...   0   0   0]
 ...
 [  0   0   0 ...  32   0   0]
 [  0   0   0 ...   0  21   1]
 [  0   0   0 ...   0   0  21]]

Confusion matrix for test data for tanh_aug:
[[ 44   3   8 ...   0   0   0]
 [ 13 642  27 ...   0   0   0]
 [  0  32 692 ...   0   0   0]
 ...
 [  0   0   1 ...  85   0   0]
 [  0   0   0 ...   0  43   0]
 [  0   0   0 ...   0   0  81]]
```

Figure 44. Confusion matrix of CNN using Data Augmentation

iv. Comparison with MLP

If we want to compare the performance of the deep MLP model and the deep CNN model, we can say that in terms of accuracy, the deep MLP model achieved an accuracy of 0.9629 on the training dataset, while the deep CNN model achieved higher accuracies ranging from 0.9750 to 0.9996 on the training dataset. This suggests that the deep CNN model generally outperforms the deep MLP model in terms of accuracy. Similarly, when comparing the performance on the test dataset, the deep MLP model achieved an accuracy of 0.8345, while the deep CNN model achieved accuracies ranging from 0.8570 to 0.9490. Once again, the deep CNN model consistently performs better than the deep MLP model. The trend continues when evaluating the models on the validation dataset. The deep MLP model achieved an accuracy of 0.9466, whereas the deep CNN model achieved accuracies ranging from 0.9543 to 0.9948. Regarding the loss values, the deep MLP model had a loss of 0.1280 on the training dataset, while the deep CNN model had lower loss values ranging from 0.0020 to 0.1086. This indicates that the deep CNN model generally provides better generalization and convergence properties.

In conclusion, based on the results, the deep CNN model consistently outperforms the deep MLP model in terms of accuracy and loss on the training, test, and validation datasets. The CNN model demonstrates better classification performance, suggesting its suitability for the task of traffic sign recognition using the German Traffic Sign Recognition Benchmark dataset.

Thanks for your Time

Masoud Pourghavam [illegible]