
# MUĞLA SITKI KOÇMAN UNIVERSITY ELECTRICAL & ELECTRONICS ENGINEERING THESIS TEMPLATE

## DEEP LEARNING BASED TRAFFIC SIGN DETECTION

**DENİZ GEÇMİŞ 190702052**

**Supervisor:**

**Doç.Dr.KEMAL UÇAK**

**2024**

## 1.1 ABSTRACT

**Deniz Geçmiş**
**B.Sc. Thesis, Electrical and Electronics Engineering Department**
**Supervisor: Asst. Prof. Dr Kemal Uçak**
**{2024}, {56} pages**

This study was designed to improve driving comfort and identify traffic signs due to the difficulty of driver eye synchronization while driving. First, data was collected for the preparation of this software project, which is the main need of autonomous vehicles, using an offline simulation technique. The articles were reviewed. Many car companies working on autonomous driving are developing this technology.

Keywords: MLP(Multi-layer perceptrons), CNN(Convolutional Neural Networks), Dataset, Train.

## 1.2 Özet

Bu çalışma sürüş konforunu artırmak ve sürüş esnasında sürücünün göz senkornizasyonu zorluğundan dolayı trafik işaretlerini tanımlamak için tasarlanmıştır. Otonom araçların başlıca ihtiyacı olan bu yazılımsal proje hakkında önce çevrimdışı simulasyon tekniğiyle hazırlanması için veriler toplandı. Makaleler incelendi.Bir çok otonom sürüş için çalışma yapan araba firması bu teknolojiyi geliştirmekte.

Anahtar kelimeler:MLP(Çok katmanlı algılayıcı), CNN(evrişimli sinir ağları), Dataset, Antrenman.

# CONTENTS

## FIGURE TABLES

## 1.3 Introduction

Deep learning-based traffic sign detection is a pivotal component within the realm of computer vision and intelligent transportation systems. The primary objective is to harness advanced machine learning techniques, particularly convolutional neural networks (CNNs), to accurately identify, interpret, and respond to the diverse array of traffic signs encountered in real-world scenarios. This technology addresses the challenges posed by the increasing complexity of road networks, offering robustness and adaptability in the face of variations in signage appearance, lighting conditions, and occlusions.

The significance of reliable traffic sign detection lies in its contribution to road safety. By enabling the timely recognition of critical signs such as speed limits, stop signs, and directional indicators, this technology assists both human drivers and autonomous vehicles in making informed decisions. Consequently, the risk of accidents is reduced, and adherence to traffic regulations is promoted. Moreover, the technology plays a crucial role in supporting the development and deployment of autonomous vehicles by providing fundamental capabilities for navigating complex urban environments.

As we move towards creating 'smart cities' where technology is seamlessly integrated into urban infrastructure, deep learning-based traffic sign detection becomes instrumental in optimizing traffic flow, minimizing congestion, and enhancing overall transportation efficiency.

This evolution aligns with the broader goal of fostering smarter and safer transportation ecosystems. In essence, the continuous advancement of deep learning-based traffic sign detection not only contributes to road safety but also shapes the future of transportation by supporting autonomous vehicles and paving the way for intelligent and efficient traffic management systems.

## 2.1 What is the Detection and Recognition?

Detection and recognition are two related concepts often used in the fields of computer vision, image processing, and pattern recognition. While they are distinct, they are frequently employed together in various applications. Here's a brief overview of each:

a)**Detection:**

Detection involves locating and identifying objects or specific features within an image or a set of data. The primary goal is to determine whether a particular object or pattern is present in a given scene. Detection tasks are commonly used in scenarios where the focus is on identifying the existence or location of objects without necessarily identifying their specific class or category.

For example, in object detection, the goal is to locate and draw bounding boxes around objects of interest within an image, indicating their presence and approximate location.

b)**Recognition:**

Recognition goes a step further by identifying and categorizing the objects or patterns that have been detected. Instead of just determining if an object is present, recognition aims to assign a specific label or class to the detected object. This involves understanding the characteristics and features of the object to match it with a predefined set of classes.

In facial recognition, for instance, the system not only detects the presence of a face but also attempts to recognize whose face it is by comparing it to a database of known faces.

In summary, detection is concerned with finding and localizing objects or patterns, while recognition involves identifying and categorizing those objects. Together, detection and recognition form the basis for various applications, including image and video analysis, autonomous vehicles, surveillance systems, and more.

## 2.2 What Does Traffic Sign Detection And Recognition Do?

Traffic sign detection and recognition refer to the process of identifying and interpreting traffic signs within a given scene, typically using computer vision and image processing techniques. This technology is commonly employed in various applications, particularly in the context of intelligent transportation systems, autonomous vehicles, and driver assistance systems.

## 2.3 Key Components

**a)Image Acquisition:**

**Camera or Sensor:** The system starts with an input source, often a camera or other sensors, that captures images or video frames of the road environment.

**b) Preprocessing:**

**Image Enhancement:** Preprocessing techniques may be applied to improve the quality of captured images, such as adjusting brightness, contrast, and sharpness.

**Color Conversion:** Conversion of images to a specific color space may be done to facilitate color-based detection.

**c)Traffic Sign Detection:**

**Feature Extraction:** Relevant features of traffic signs, such as shape, color, and symbols, are extracted from the preprocessed images.

**Object Localization:** Detection algorithms identify the presence and location of potential traffic signs within the image.

**Bounding Box Generation:** Once a traffic sign is detected, a bounding box is often drawn around it to indicate its spatial location.

**d)Postprocessing:**

**Filtering:** Postprocessing steps may include filtering out false positives and refining the detected results to improve accuracy.

**Size and Shape Analysis:** Validating the size and shape of detected objects to ensure they match the expected characteristics of traffic signs.

**e)Traffic Sign Recognition:**

**Classification:** The detected traffic signs are classified into specific categories or types based on their extracted features.

**Symbol Recognition:** If applicable, recognition of specific symbols or text on the traffic sign may be performed.

**f)Decision-Making:**

**Interpretation:** The recognized traffic signs are interpreted to understand their meaning (e.g., speed limit, stop, yield).

**Driver Assistance:** In driver assistance systems, the interpreted information is used to assist the driver, such as providing warnings or recommendations.

**g)Integration:**

**Integration with Navigation System:** In some applications, the system may integrate with a navigation system to provide route-specific information.

**Communication:** Communication modules may be included to transmit relevant information to other vehicles or central control systems.

**h)Feedback and Monitoring:**

**User Interface:** Providing feedback to the driver through a user interface, displaying recognized signs or alerts.

**Monitoring:** Continuous monitoring and evaluation of the system's performance.

## 2.4 Benefits this Project

1. **Enhanced Road Safety:**
    - By promptly identifying and interpreting traffic signs, the system contributes to improved road safety by assisting drivers in adhering to traffic regulations.
2. **Accident Prevention:**
    - The system can help prevent accidents by notifying drivers of important information, such as speed limits, stop signs, and other critical instructions.
3. **Driver Assistance:**
    - It provides valuable assistance to drivers, especially in unfamiliar environments, by offering real-time information about road conditions and regulations.
4. **Reduced Traffic Violations:**
    - Drivers are less likely to violate traffic rules when equipped with a system that actively identifies and communicates relevant traffic sign information.
5. **Adaptive Driving Behavior:**
    - The system enables adaptive driving behavior by providing information on variable speed limits, construction zones, or changing road conditions.
6. **Automation Support:**
    - In the context of autonomous vehicles, traffic sign detection and recognition are crucial for safe navigation, as the system can autonomously respond to road signs and signals.

7. **Efficient Traffic Flow:**
    - Improved adherence to traffic rules contributes to more orderly and efficient traffic flow, reducing congestion and potential traffic-related issues.
8. **Increased Awareness:**
    - Drivers are more aware of their surroundings and potential hazards, leading to a heightened sense of situational awareness.
9. **Customized Driver Alerts:**
    - The system can deliver customized alerts to drivers based on detected traffic signs, enhancing communication and reducing the likelihood of oversights.
10. **Integration with Navigation Systems:**
    - Integration with navigation systems allows for the provision of route-specific information, optimizing the driving experience.
11. **Environmental Benefits:**
    - Smoother traffic flow and reduced incidents may contribute to lower fuel consumption and emissions, offering environmental advantages.
12. **Support for Smart Cities:**
    - Traffic sign detection and recognition systems align with the goals of smart city initiatives by leveraging technology to enhance overall urban mobility and safety.

While these benefits are substantial, it's important to note that the effectiveness of the system depends on its accuracy, reliability, and the integration with other components of the vehicle or transportation infrastructure. Additionally, ethical considerations, data privacy, and cybersecurity are essential aspects to address in the deployment of such systems.

## 3.1 Perceptron

The perceptron is a foundational concept in artificial neural networks and machine learning. It is a simple mathematical model of a biological neuron, designed for binary classification tasks. The perceptron takes multiple binary input signals, assigns weights to each input, calculates a weighted sum, applies an activation function, and produces a binary output. It forms the basis for more complex neural network architectures used in pattern recognition and classification tasks.

### *3.1.1 Basic Components of Perceptron*

Perceptron is a type of artificial neural network, which is a fundamental concept in machine learning. The basic components of a perceptron are:

1. **Input Layer:** The input layer consists of one or more input neurons, which receive input signals from the external world or from other layers of the neural network.

2. **Weights:** Each input neuron is associated with a weight, which represents the strength of the connection between the input neuron and the output neuron.

3. **Bias:** A bias term is added to the input layer to provide the perceptron with additional flexibility in modeling complex patterns in the input data.

4. **Activation Function:** The activation function determines the output of the perceptron based on the weighted sum of the inputs and the bias term. Common activation functions used in perceptrons include the step function, sigmoid function, and ReLU function.

5. **Output:** The output of the perceptron is a single binary value, either 0 or 1, which indicates the class or category to which the input data belongs.

6. **Training Algorithm:** The perceptron is typically trained using a supervised learning algorithm such as the perceptron learning algorithm or backpropagation. During training, the weights and biases of the perceptron are adjusted to minimize the error between the predicted output and the true output for a given set of training examples.

7. Overall, the perceptron is a simple yet powerful algorithm that can be used to perform binary classification tasks and has paved the way for more complex neural networks used in deep learning today.

### 3.1.2 Types of Perceptron:

1. **Single layer:** Single layer perceptron can learn only linearly separable patterns.

2. **Multilayer:** Multilayer perceptrons can learn about two or more layers having a greater processing power.

The Perceptron algorithm learns the weights for the input signals in order to draw a linear decision boundary.

**Note:** Supervised Learning is a type of Machine Learning used to learn models from labeled training data. It enables output prediction for future or unseen data. Let us focus on the Perceptron Learning Rule in the next section.

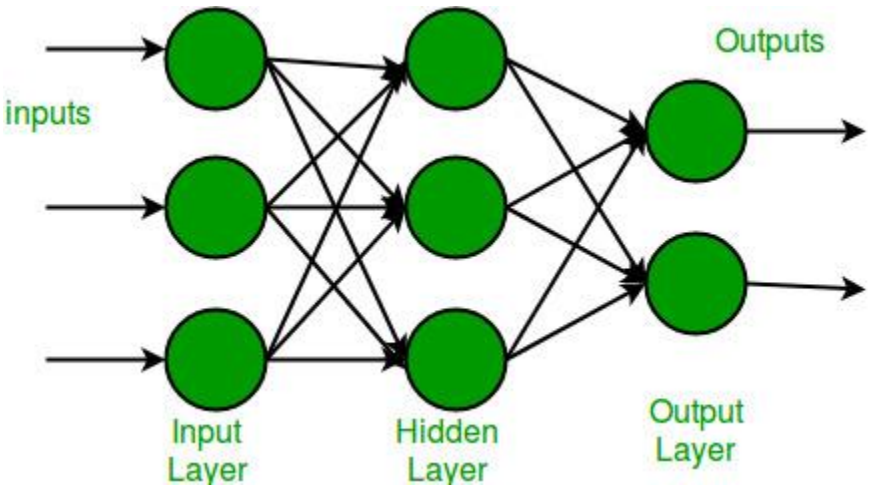



Figure 1 : Structure of perceptron

### *3.1.3 Perceptron in Machine Learning*

The most commonly used term in Artificial Intelligence and Machine Learning (AIML) is Perceptron. It is the beginning step of learning coding and Deep Learning technologies, which consists of input values, scores, thresholds, and weights implementing logic gates. Perceptron is the nurturing step of an Artificial Neural Link. In 19h century, Mr. Frank Rosenblatt invented the Perceptron to perform specific high-level calculations to detect input data capabilities or business intelligence. However, now it is used for various other purposes.

### *3.1.4 Types of Perceptron models*

We have already discussed the types of Perceptron models in the Introduction. Here, we shall give a more profound look at this:

1. **Single Layer Perceptron model:** One of the easiest ANN(Artificial Neural Networks) types consists of a feed-forward network and includes a threshold transfer inside the model. The main objective of the single-layer perceptron model is to analyze the linearly separable objects with binary outcomes. A Single-layer perceptron can learn only linearly separable patterns.

2. **Multi-Layered Perceptron model:** It is mainly similar to a single-layer perceptron model but has more hidden layers.

**Forward Stage:** From the input layer in the on stage, activation functions begin and terminate on the output layer.

**Backward Stage:** In the backward stage, weight and bias values are modified per the model's requirement. The backstage removed the error between the actual output and demands originating backward on the output layer. A multilayer perceptron model has a greater processing power and can process linear and non-linear patterns. Further, it also implements logic gates such as AND, OR, XOR, XNOR, and NOR.

***ADVANTAGES :***

- A multi-layered perceptron model can solve complex non-linear problems.
- It works well with both small and large input data.
- Helps us to obtain quick predictions after the training.
- Helps us obtain the same accuracy ratio with big and small data.

***DISADVANTAGES :***

- In multi-layered perceptron model, computations are time-consuming and complex.
- It is tough to predict how much the dependent variable affects each independent variable.
- The model functioning depends on the quality of training.

### *3.1.5 Characteristics of the Perceptron Model*

**The following are the characteristics of a Perceptron Model:**

1. It is a machine learning algorithm that uses supervised learning of binary classifiers.
2. Perceptron, the weight coefficient is automatically learned.
3. Initially, weights are multiplied with input features, and then the decision is made whether the neuron is fired or not.
4. The activation function applies a step rule to check whether the function is more significant than zero.
5. The linear decision boundary is drawn, enabling the distinction between the two linearly separable classes +1 and -1.
6. If the added sum of all input values is more than the threshold value, it must have an output signal; otherwise, no output willbe shown.

### 3.1.6 Limitation of Perceptron Model

The following are the limitation of a Perceptron model:

1. The output of a perceptron can only be a binary number (0 or 1) due to the hard-edge transfer function.
2. It can only be used to classify the linearly separable sets of input vectors. If the input vectors are non-linear, it is not easy toclassify them correctly.

### *3.1.7 Activation Functions of Perceptron*

The activation function applies a step rule (convert the numerical output into +1 or -1) to check if the output of the weighting function is greater than zero or not.

For example:

If ∑ wixi> 0 => then final output “o” = 1

(issue bank loan) Else, final output “o” = -1

(deny bank loan)

Step function gets triggered above a certain value of the neuron output; else it outputs zero. Sign Function outputs +1 or -1 depending on whether neuron output is greater than zero or not. Sigmoid is the S-curve and outputs a value between 0 and 1.

### *3.1.8 Implementing Basic Logic Gates With Perceptron*

The logic gates that can be implemented with Perceptron are discussed below.

If the two inputs are TRUE (+1), the output of Perceptron is positive,

which amounts to TRUE.

**AND**

This is the desired behavior of an AND gate.

x1= 1 (TRUE), x2= 1 (TRUE)

w0 = -.8, w1 = 0.5, w2 = 0.5

=> o(x1, x2) => -.8 + 0.5*1 + 0.5*1 = 0.2 > 0

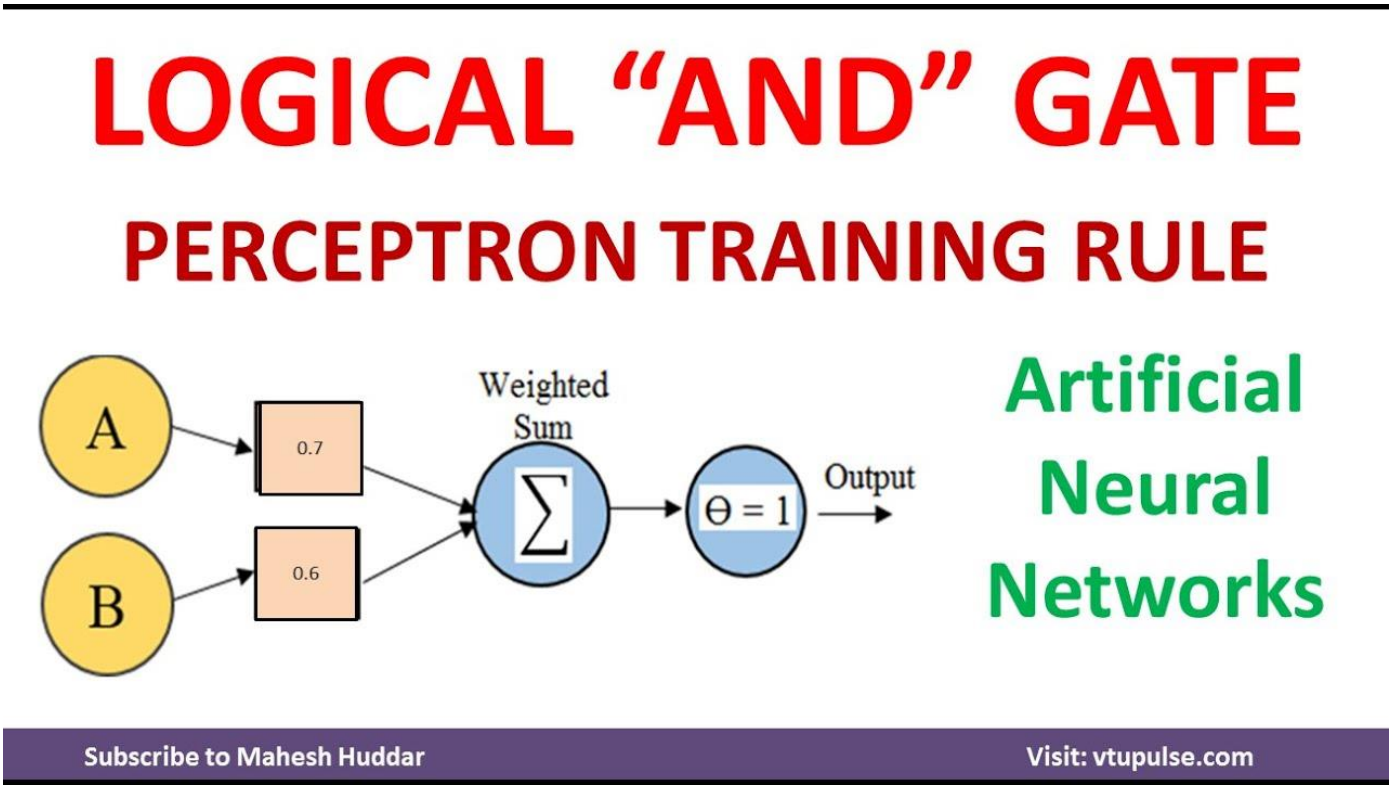
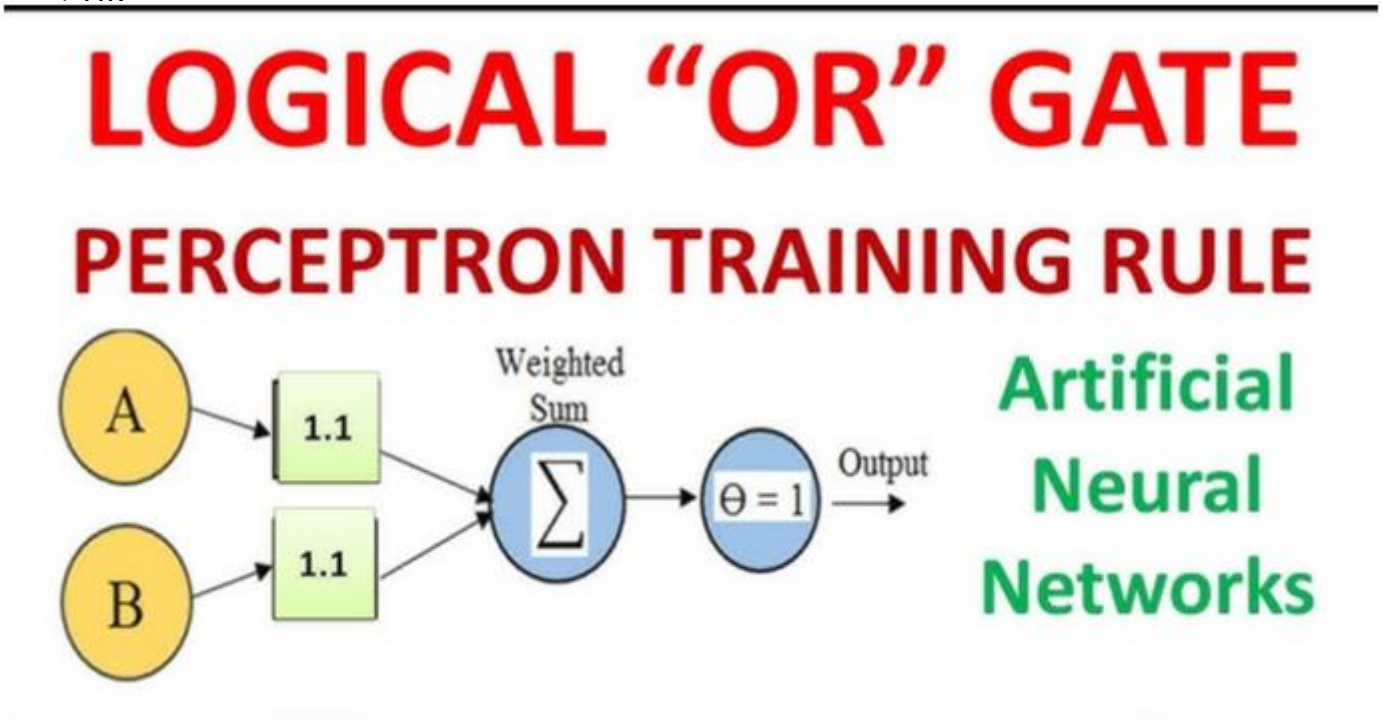



Figure 2 : Logical And Gate example Perceptron.

1 OR




### 3.2 Method of Steepest Descent and its Applications

The method of steepest descent is the simplest of the gradient methods. Imagine that there's a function F(x), which can be definedand differentiable within a given boundary, so the direction it decreases the fastest would be the negative gradient of F(x). To findthe local minimum of F(x), The Method of The Steepest Descent is employed, where it uses a zig-zag like path from an

arbitrary point X0 and gradually slide down the gradient, until it converges to the actual point of minimum.

An extension of the steepest descent method is the so-called nonlinear stationary phase/steepest descent method. Here, instead ofintegrals, one needs to evaluate asymptotically solutions of Riemann–Hilbert factorization problems.

Given a contour C in the complex sphere, a function f defined on that contour and a special point, say infinity, one seeks a function M holomorphic away from the contour C, with prescribed jump across C, and with a given normalization at infinity. If f and hence M are matrices rather than scalars this is a problem that in general does not admit an explicit solution.

An asymptotic evaluation is then possible along the lines of the linear stationary phase/steepest descent method. The idea is to reduce asymptotically the solution of the given Riemann–Hilbert problem to that of a simpler, explicitly solvable, Riemann–Hilbertproblem. Cauchy's theorem is used to justify deformations of the jump contour.

The nonlinear stationary phase was introduced by Deift and Zhou in 1993, based on earlier work of the Russian mathematician Alexander Its. A (properly speaking) nonlinear

steepest descent method was introduced by Kamvissis, K. McLaughlin and P. Millerin 2003, based on previous work of Lax, Levermore, Deift, Venakides and Zhou. As in the linear case, steepest descent contours solve a min-max problem. In the nonlinear case they turn out to be "S-curves" (defined in a different context back in the 80s by Stahl, Gonchar and Rakhmanov).

The nonlinear stationary phase/steepest descent method has applications to the theory of soliton equations and integrable models, random matrices and combinatorics.

Another extension is the Method of Chester–Friedman–Ursell for coalescing saddle points and uniform asymptotic extensions.

### 3.3 LEVENBERG-MARGUARD ALGORITHM

#### What is the Levenberg–Marquardt Algorithm?

1) The standard method for solving least squares problems which lead to non-linear normal equations

depends upon a reduction of the residuals to linear form by first order Taylor approximations taken about an initial or trial solution for the parameters.

2) If the usual least squares procedure, performed with these linear approximations, yields new values for the parameters which are not sufficiently close to the initial values, the neglect of second and higher order terms may invalidate the process, and may actually give rise to a larger value of the sum of the squares of the residuals than that corresponding to the initial solution. This failure of the standard method to improve the initial solution has received some notice in statistical applications of least squares

3) and has been encountered rather frequently in connection with certain engineering applications involving the approximate representation of one function by another. The purpose of this article is to show how the problem may be solved by an extension of the standard method which insures improvement of the initial solution.

4) The process can also be used for solving non-linear simultaneous equations, in which case it may be considered an extension of Newton's method.

The Levenberg–Marquardt (LM) Algorithm is used to solve nonlinear least squares problems. This curve-fitting method is a combination of two other methods: the gradient descent and the Gauss-Newton.

Both the Gradient Descent and Gauss-Newton methods are iterative algorithms, which means they use a series of calculations (based on guesses for x-values) to find a solution. The gradient descent differs in that at each iteration, the solution updates by choosing values that make the function value smaller. More specifically, the sum of the squared errors is reduced by moving toward the direction of steepest descent. At each iteration, the Levenberg–Marquardt Algorithm chooses either the gradient descent or GN and updates the solution.

The iterative update is dependent on the value of an algorithmic parameter, λ— a non-negative dumping factor which smooths out the graph. The update is Gauss-Newton if λ is small (i.e. close to the optimal value) and a gradient descent if λ is large (Gavin, 2007). The Gauss-Newton is more accurate and faster than the gradient descent when close to the minimum error. Therefore, so the algorithm will migrate towards the GN algorithm as soon as possible.

## 4 MultiLayer Perceptrons

Components of an MLP:

**1. Input Layer:**

The Input Layer in a Multilayer Perceptron (MLP), which is a type of artificial neural network (ANN), is the initial layer where the network receives and processes the input data. Here are the key details about the input layer:

**Purpose**:

- **Data Reception**: It receives the raw input data and passes it forward through the network for further processing and feature extraction.

- **Feature Representation**: Each neuron in the input layer represents a specific feature or attribute from the input data.

**Characteristics:**

1. **Neurons Corresponding to Features:**

- The number of neurons in the input layer equals the number of input features in the dataset.

- For example, in an image classification task with RGB images of size 32x32, the input layer might have 32 * 32 * 3 neurons (3 for each RGB channel).

2. **No Computation:**

- Neurons in the input layer do not perform any computation or transformation on the input data.

- They simply act as conduits to transmit the input values to the neurons in the subsequent layers.

3. **Direct Connectivity:**

- Each neuron in the input layer directly connects to every neuron in the next layer (the first hidden layer).

- The connections are associated with weights that are adjusted during the training process to learn patterns and representations in the data.

**Data Representation:**

- Vectorized Format: Input data is usually vectorized or flattened before being fed into the network.

- Numeric Encoding: Categorical or non-numeric data may be encoded into numeric formats suitable for neural networks (one-hot encoding, label encoding, etc.).

**Role in the Network:**

- **Initial Processing**: It serves as the entry point for the data into the neural network architecture.

- **Feature Transmission**: Neurons transmit the input data to the subsequent layers, where computations and learning occur.

**Input Layer Activation:**

- **Identity Activation**: Neurons in the input layer often use an identity activation function (i.e., no activation) because they solely transmit the input values without any modification.

- **Linear Transformation**: The input layer performs a linear transformation of the input data to the subsequent layers.

**Importance:**

- **Feature Representation**: The input layer represents the original features of the data and initiates the flow of information through the network.
- **Network Initialization**: The size and structure of the input layer dictate the dimensions of subsequent layers and the overall network architecture.

**Considerations:**

- **Normalization**: Input data normalization (scaling, standardization) might be performed before feeding **it into the input layer to ensure consistent and efficient learning.**
- **Handling Different Data Types**: Neural networks can handle various types of data (numeric, categorical), but preprocessing steps might vary based on the data characteristics.

The input layer acts as the entry point for data in an MLP, conveying the raw features into the network for subsequent processing, feature extraction, and learning in the hidden and output layers.

**2. Hidden Layers:**

In a Multilayer Perceptron (MLP), the Hidden Layers are the intermediate layers between the input and output layers. These layers perform computations on the input data, learning representations of patterns and features in the data. Here are the key details about the hidden layers:

**Purpose:**

- **Feature Extraction and Representation:** Hidden layers learn increasingly abstract and complex representations of the input data as information passes through them.

**Non-Linear Transformations:** They introduce non-linearities to the network, enabling it to learn and approximate complex relationships in the data.

**Characteristics:**

1. **Neurons and Connectivity:**

- Each hidden layer contains multiple neurons or nodes.
- Neurons in a layer are fully connected to neurons in the previous and subsequent layers.

2. **Computation and Activation:**

- Neurons in hidden layers perform computations on the inputs received from the previous layer.

- Each neuron computes a weighted sum of its inputs, applies an activation function, and passes the result to the next layer.

3. **Learned Representations:**

- As data flows through the hidden layers, neurons learn to extract hierarchical representations of features.
- Early hidden layers capture simpler features, while deeper layers learn more abstract and complex features.

**3. Output Layer:**

The Output Layer in a Multilayer Perceptron (MLP) is the final layer of neurons responsible for producing the network's output. Its configuration and characteristics are tailored to the specific task the MLP is designed for, such as classification, regression, or other prediction tasks. Here are the key details about the output layer:

**Purpose:**

- Final Prediction or Decision: The output layer produces the final result or prediction based on the learned representations from the previous layers.

**Characteristics:**

1. **Neurons in the Output Layer:**

- Number of neurons in this layer depends on the nature of the task:
- For binary classification, a single neuron is often used.
- For multi-class classification, there's a neuron for each class.
- For regression, there's typically a single neuron for each output value.

2. **Activation Function**:

- The choice of activation function in the output layer is task-dependent:
- **Sigmoid**: Commonly used for binary classification, where the output represents probabilities between 0 and 1.
- **Softmax**: Used for multi-class classification, providing probabilities for each class that sum up to 1.
- **Linear/Identity:** Used for regression tasks, providing continuous output without constraints.

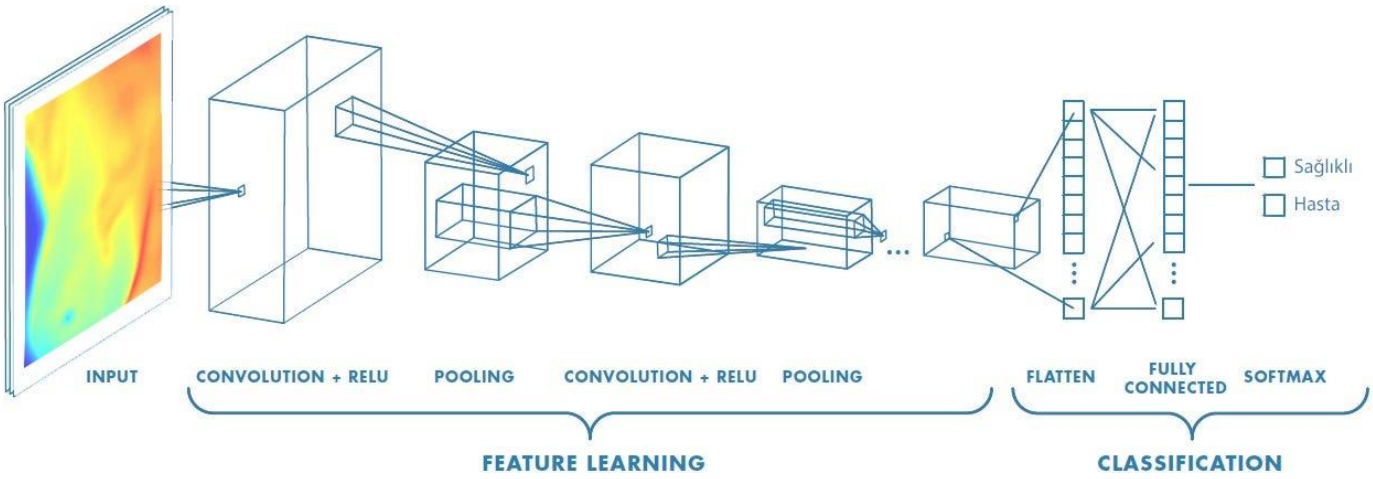



**Figure 3 : Structure of CNN**

**Components within Neurons:**

1. **Calculation and Transformation:**

- Neurons in the output layer perform computations similar to those in the hidden layers.
- Weighted sum calculation followed by the application of an appropriate activation function.

2. **Final Prediction:**

- The activations produced by the output neurons represent the final predictions or outputs of the network.

**Task-specific Configuration:**

- **Classification Tasks:**
- For binary classification, a single neuron using a sigmoid activation is often employed.
- For multi-class classification, the number of neurons matches the number of classes, using softmax activation for probability distribution.

- **Regression Tasks:**
- The number of neurons corresponds to the number of output variables, and linear activation is typically used to produce continuous values.

**Importance:**

- **Decision Making:** The output layer produces the final prediction or decision based on the learned representations from the preceding layers.
- **Task Specificity**: Configured to suit the requirements of the task at hand, ensuring the appropriate format for the output.

**Considerations:**

**Activation Function Selection:** Choosing the correct activation function aligns with the nature of the task and ensures suitable output behavior.

- **Number of Neurons**: Corresponds to the requirements of the task; incorrect neuron count can lead to model misalignment.

The output layer in an MLP is crucial as it generates the final predictions or results based on the network's learned representations. Its configuration, including the number of neurons and choice of activation function, is tailored to suit the specific task, be it classification, regression, or other predictive modeling tasks.

**Training Components:**

a. **Loss Function:**

- **Measures Discrepancy:** Measures the difference between predicted and actual values.
- **Example:** Mean Squared Error (MSE) for regression, Cross-Entropy for classification.

b. **Backpropagation:**

- **Error Propagation**: Gradients are calculated with respect to the loss function and propagated backward through the network.
- **Update Weights**: Weights and biases are adjusted using optimization algorithms (e.g., Gradient Descent) to minimize the loss.

Importance of MLP Components:

- **Feature Learning**: Hidden layers help the network learn representations of complex patterns and relationships within the data.
- **Non-Linearity:** Activation functions introduce non-linearities, enabling the network to approximate complex functions.
- **Model Training:** Backpropagation and optimization algorithms allow the network to adjust its parameters to minimize errors and improve predictions.
- **Task-specific Output:** The final output layer provides predictions based on the learned representations, suited to the task (classification, regression, etc.).

**Limitations and Considerations:**

- **Complexity:** MLPs might struggle with highly complex tasks or datasets compared to more advanced architectures like CNNs or RNNs.

- **Overfitting:** Without proper regularization techniques, MLPs can overfit to the training data, leading to poor generalization on unseen data.

- **Hyperparameter Tuning:** Choosing the right number of layers, neurons per layer, and activation functions requires careful tuning and experimentation.

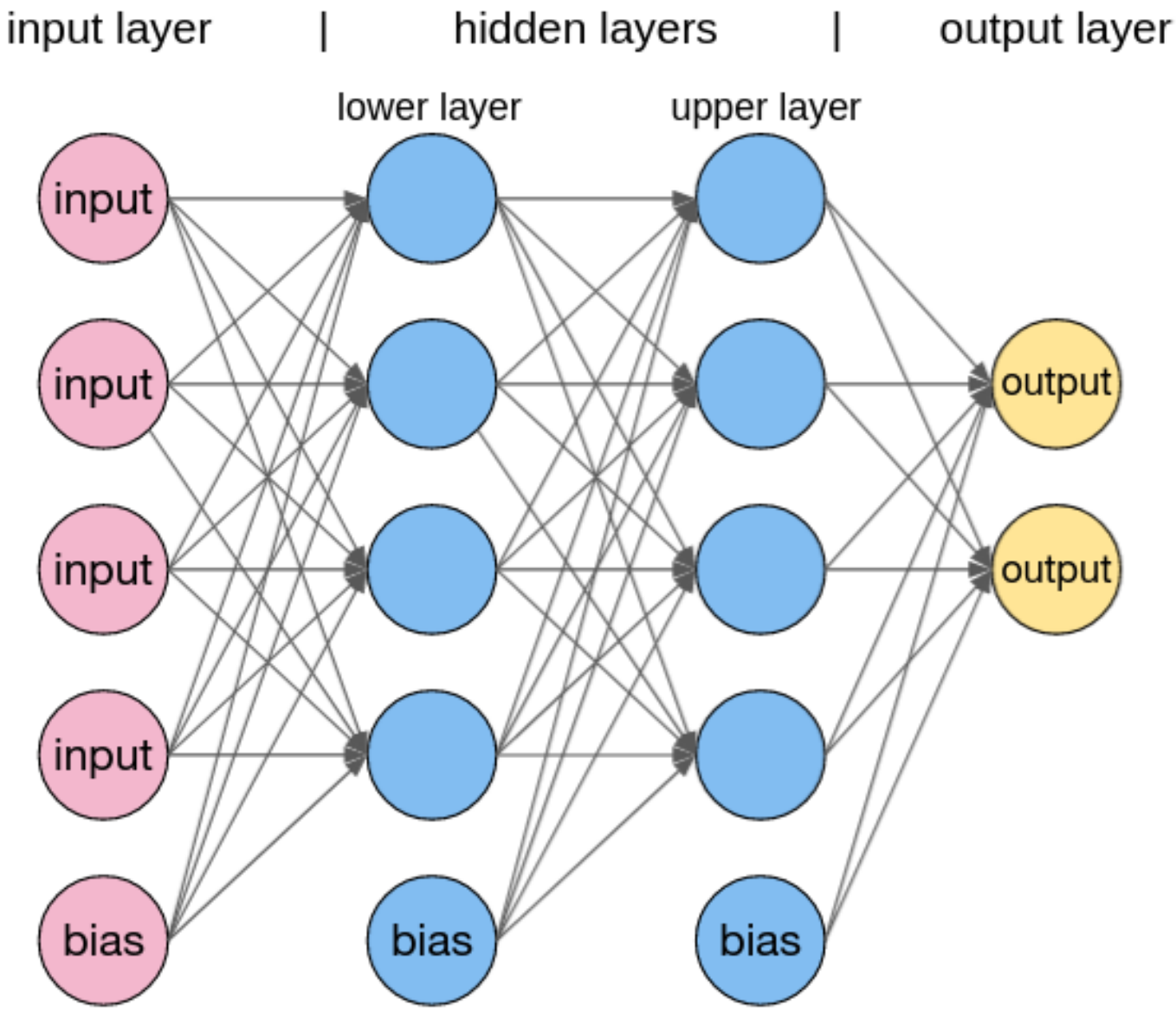



**Figure 4 : MLP**

**5 CNN's**

**-Pattern recognition;** using Multilayer Perceptrons (MLPs) involves using neural networks composed of multiple layers of interconnected neurons to recognize patterns or structures within data. While CNNs are typically more effective for image-related tasks, MLPs are versatile and can be applied to various pattern recognition problems.

**1. Data Preparation ,2. Network Architecture, 3. Forward Propagation, 4. Feature Learning and Representation, 5. Output Prediction, 6. Training and Learning, 7. Pattern Recognition, 8. Model Evaluation, 9. Prediction on New Data.**

**Advantages and Limitations:**

- **Advantages: MLPs are versatile and can handle various types of data and tasks. They're effective for many pattern recognition tasks, including speech recognition, handwriting recognition, and more.**

- **Limitations: They might struggle with handling high-dimensional and spatially structured data like images efficiently compared to CNNs due to the lack of explicitly learned spatial hierarchies.**

**While MLPs are powerful for general pattern recognition tasks, for image-related problems where spatial relationships are crucial, CNNs are often more effective due to their ability to capture spatial patterns through their specialized architecture.**

-CNN stands for Convolutional Neural Network, a type of artificial neural network designed for processing structured grid-like data, such as images. It's widely used in computer vision tasks like image classification, object detection, and image segmentation. CNNs are powerful because they can automatically and adaptively learn spatial hierarchies of features from the input data.

**Components of CNN:**

Convolutional Layer, Activation Function, Pooling (or Subsampling) Layer, Fully Connected (FC) Layer, Flattening.

**CNN workflow:**

Input, Convolutional Layers, Activation and ReLU, Pooling Layers, Flattening, Fully Connected Layers, Output Layer, Loss Calculation and Optimization, Training and Iteration, Model Evaluation, Deployment.

**Role in CNNs:**

Flattening is a critical step in the transition from the feature extraction layers (convolutional and pooling layers) to the classification or decision-making layers (fully connected layers) in CNNs. It enables the network to learn high-level representations of the input data and make predictions based on these learned features.

Flattening helps maintain the representational power of the extracted features while reorganizing them for processing by fully connected layers, allowing the network to learn complex patterns and relationships in the data.

-Pattern recognition via Convolutional Neural Networks (CNNs) involves utilizing specialized neural network architectures designed to efficiently recognize and extract patterns from images or visual data. Here's a detailed explanation of pattern recognition using **CNNs**:

Convolutional Layers, Pooling Layers, Activation Functions, Hierarchical Feature Learning, Fully Connected Layers, Backpropagation and Training, Transfer Learning, Output Layer and Task-Specific Activation, Interpretation and Analysis.

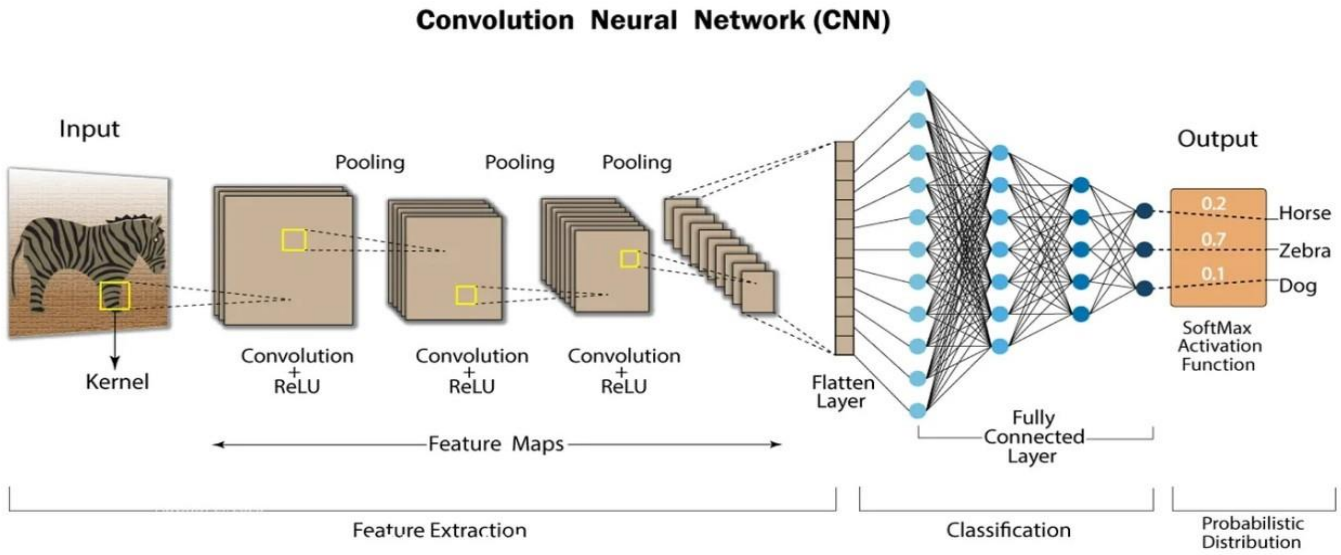



**Figure 5 : CNN's**

# 6 Methods the deep learning based traffic sign detection;

## 6.1 Download dataset of Traffic Signs;

GTSRB (German Traffic Sign Recognition Reference) dataset was used in this project.

The GTSRB (German Traffic Sign Recognition Benchmark) dataset is a collection of images of traffic signs used for the development and evaluation of traffic sign recognition algorithms. It was created for the purpose of benchmarking and comparing the performance of different traffic sign recognition models. The dataset is particularly focused on German traffic signs, but it has been widely used for research and development in the broader field of computer vision and machine learning.

Here are some key details about the GTSRB dataset:

1. **Size and Content:**
    - The dataset consists of more than 50,000 images of traffic signs.
    - There are 43 different classes of traffic signs, each representing a specific type of traffic regulation or information.
2. **Image Characteristics:**
    - Images in the dataset vary in terms of lighting conditions, weather conditions, and viewpoints to simulate real-world scenarios.
    - The images are in color and have varying resolutions.
3. **Dataset Splits:**

- The dataset is typically divided into training and testing sets to facilitate the training and evaluation of machine learning models.
- A common split is to use about 80% of the data for training and 20% for testing.

4. **Annotations:**
    - Each image is labeled with the corresponding class of the traffic sign it represents.
    - The labels are usually represented as numerical values corresponding to the different traffic sign classes.
5. **Usage in Research:**
    - The GTSRB dataset has been widely used in research to develop and evaluate algorithms for traffic sign recognition.
    - Researchers use this dataset to benchmark the performance of their models and compare them to state-of-the-art methods.
6. **Challenges:**
    - Recognizing traffic signs in real-world scenarios is a challenging task due to variations in lighting, weather, and viewpoint.
    - Different traffic signs may have similar visual appearances, making accurate classification a non-trivial problem.

Researchers and practitioners use the GTSRB dataset to test the robustness and accuracy of their computer vision models, especially those designed for real-time traffic sign recognition applications in autonomous vehicles and advanced driver-assistance systems.

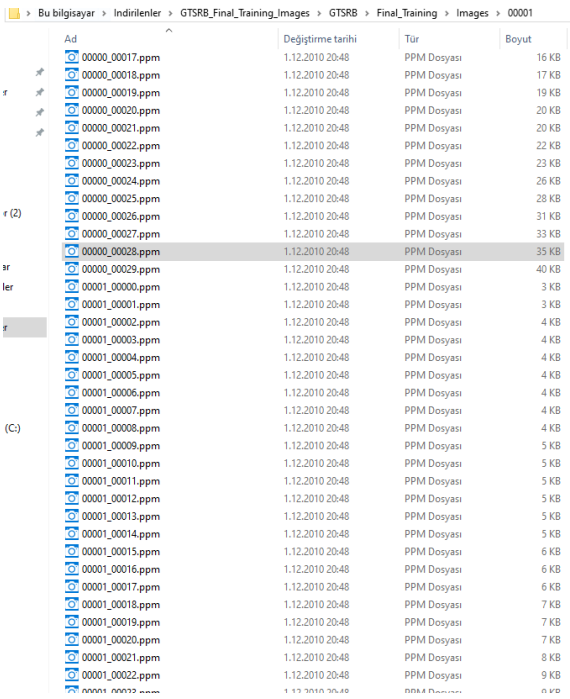



Figure 6: GTSRB Image File

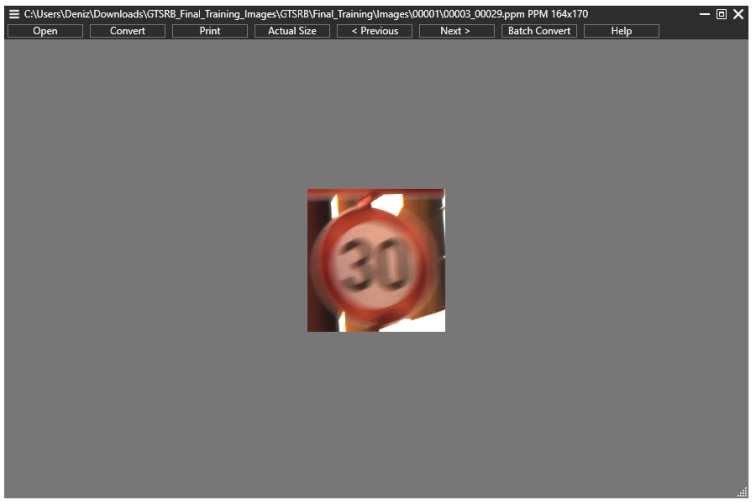



Figure 7: GTSRB Image Example.

**6.2 Convert downloaded dataset to it for Classification;**

| | | |
|---|---|---|
| **0** | 0 | Speed limit (20km/h) |
| **1** | 1 | Speed limit (30km/h) |
| **2** | 2 | Speed limit (50km/h) |
| **3** | 3 | Speed limit (60km/h) |
| **4** | 4 | Speed limit (70km/h) |
| **5** | 5 | Speed limit (80km/h) |
| **6** | 6 | End of speed limit (80km/h) |
| **7** | 7 | Speed limit (100km/h) |
| **8** | 8 | Speed limit (120km/h) |
| **9** | 9 | No passing |
| **10** | 10 | No passing for vehicles over 3.5 metric tons |
| **11** | 11 | Right-of-way at the next intersection |
| **12** | 12 | Priority road |
| **13** | 13 | Yield |
| **14** | 14 | Stop |
| **15** | 15 | No vehicles |
| **16** | 16 | Vehicles over 3.5 metric tons prohibited |
| **17** | 17 | No entry |
| **18** | 18 | General caution |
| **19** | 19 | Dangerous curve to the left |
| **20** | 20 | Dangerous curve to the right |
| **21** | 21 | Double curve |
| **22** | 22 | Bumpy road |
| **23** | 23 | Slippery road |
| **24** | 24 | Road narrows on the right |
| **25** | 25 | Road work |
| **26** | 26 | Traffic signals |

Figure 8 : Classes Images.

**6.3 Construct set of datasets with colour and grayscale images;**

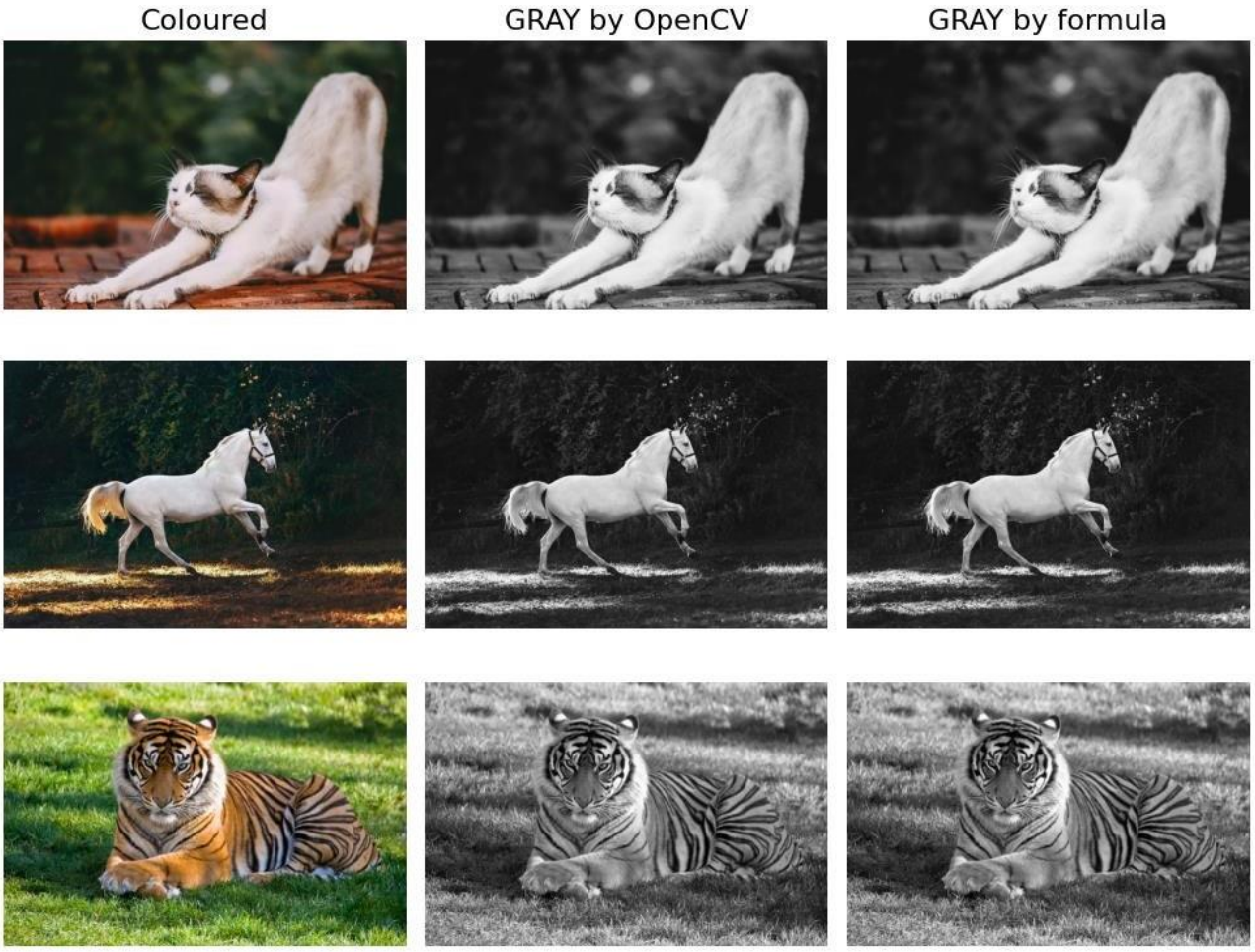



**Figure 9** : Rgb image example.

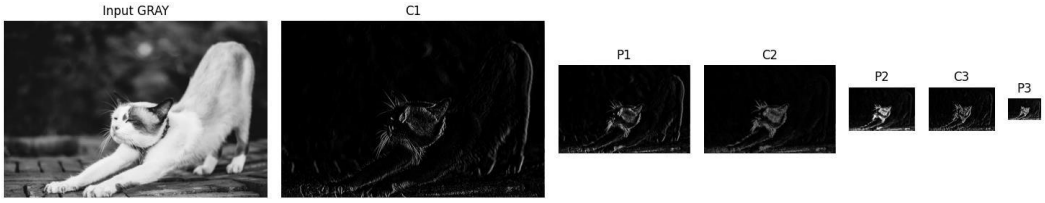



Figure 10: Gray image example.

### 6.4 How many Convolutional-Pooling pairs of layers?

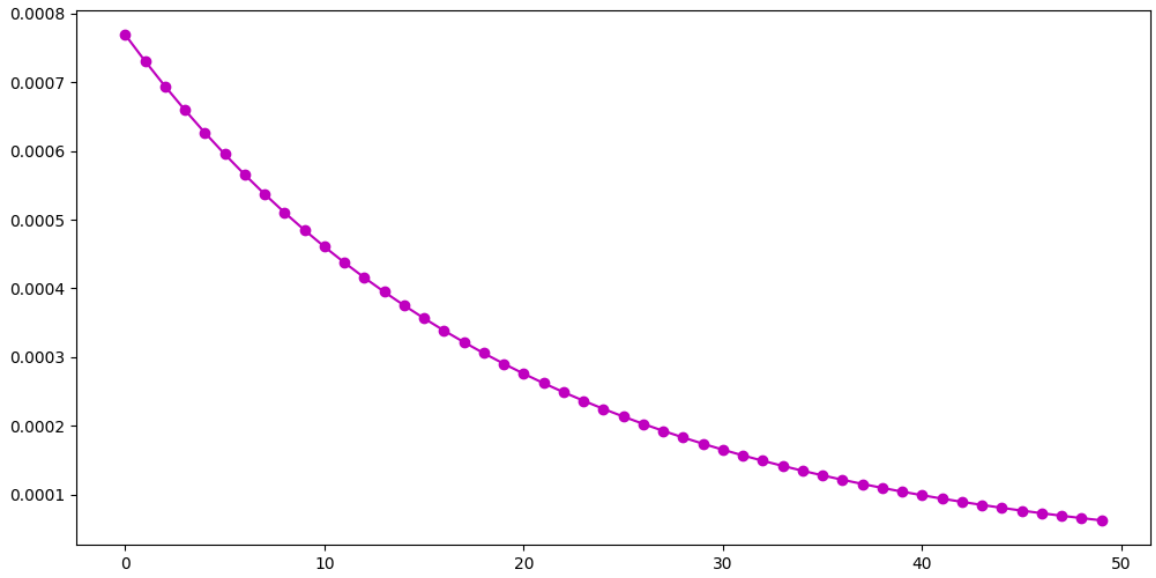



**Figure 11 :** Convolutional Layers graph example

**Python Example** ;

```
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 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(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

### 6.5 How many Feature Maps in Convolutional layers?

The question "How many Feature Maps in Convolutional layers?" is asking about the number of feature maps in each Convolutional layer of a Convolutional Neural Network (CNN).
Each Convolutional layer in a CNN typically contains multiple filters (kernels), and each filter produces a feature map by convolving with the input data. Therefore, the total number of feature maps in a Convolutional layer is equal to the number of filters in that layer.
For example, consider the following Convolutional layer:

**model.add(Conv2D(64, (3, 3), activation='relu', input_shape=(64, 64, 3)))**

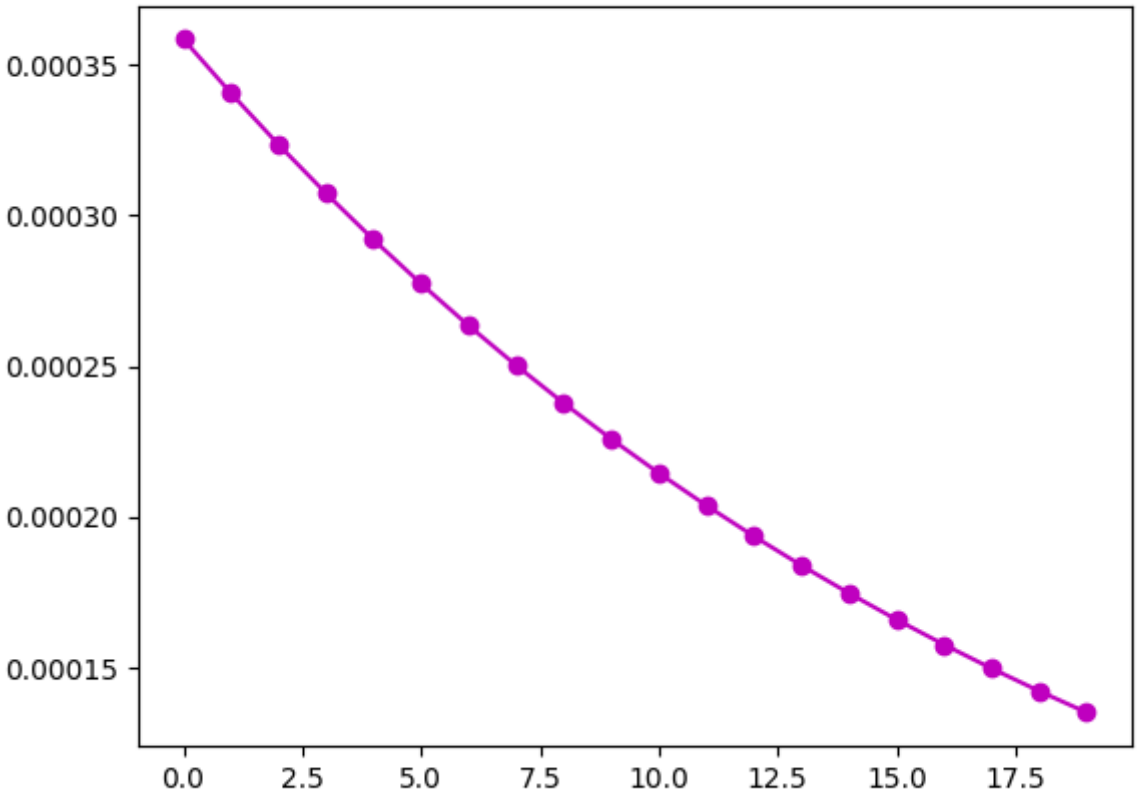



**Figure 12 :** Feature map graph example

**6.6 How many neurons in the fully connected layer?**

The question "How many neurons in the fully connected layer?" is asking about the number of neurons in a fully connected layer. Fully connected layers are typically layers where each neuron is connected to every neuron in the preceding layer.
For example, consider the following fully connected layer:

**model.add(Dense(128, activation='relu'))**

**6.7 How much Dropout?**

The question "How much Dropout?" is asking about the dropout rate used in a neural network, typically in the context of a dropout layer. Dropout is a regularization technique commonly used in neural networks to prevent overfitting.
The dropout rate represents the fraction of neurons (or connections) that are randomly dropped out or ignored during training. It is a value between 0 and 1, where 0 means no neurons are dropped out, and 1 means all neurons are dropped out. A common practice is to use dropout rates between 0.2 and 0.5.
For example, in a Keras model, you might have a dropout layer like this:

**model.add(Dropout(0.5))**

**6.8 Save designed deep CNN models into binary files:**

**Code;**

***

```
# Building and saving models for Traffic Signs dataset
# Input --> {128C5-P2-D30} --> {256C5-P2-D30} --> {512C5-P2-D30} --> {1024C3-P2-D30} --> 2048-D30 --> 43
# Input --> {128C24-P2-D30} --> 256-D30 --> 43



# Building 1st model for RGB datasets
# RGB --> {128C5-P2-D30} --> {256C5-P2-D30} --> {512C5-P2-D30} --> {1024C3-P2-D30} --> 2048-D30 --> 43

# Initializing model to be as linear stack of layers
model = Sequential()

# Adding first convolutional-pooling pair
model.add(Conv2D(128, kernel_size=5, padding='same', activation='relu', input_shape=(48, 48, 3)))
model.add(MaxPool2D())
model.add(Dropout(0.3))

# Adding second convolutional-pooling pair
model.add(Conv2D(256, kernel_size=5, padding='same', activation='relu'))
model.add(MaxPool2D())
model.add(Dropout(0.3))

# Adding third convolutional-pooling pair
model.add(Conv2D(512, kernel_size=5, padding='same', activation='relu'))
model.add(MaxPool2D())
model.add(Dropout(0.3))

# Adding fourth convolutional-pooling pair
model.add(Conv2D(1024, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPool2D())
model.add(Dropout(0.3))

# Adding fully connected layers
model.add(Flatten())
model.add(Dense(2048, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(43, activation='softmax'))

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

```
# Saving 1st model for RGB datasets
# (!) On Windows, it might need to change
# this: + '/' +
# to this: + '\' +
# or to this: + '\\' +
model.save('ts' + '/' + 'model_1_ts_rgb.h5')



# Building 1st model for GRAY datasets
# GRAY --> {128C5-P2-D30} --> {256C5-P2-D30} --> {512C5-P2-D30} --> {1024C3-P2-D30} --> 2048-D30 -->
43

# Initializing model to be as linear stack of layers
model = Sequential()

# Adding first convolutional-pooling pair
model.add(Conv2D(128, kernel_size=5, padding='same', activation='relu', input_shape=(48, 48, 1)))
model.add(MaxPool2D())
model.add(Dropout(0.3))

# Adding second convolutional-pooling pair
model.add(Conv2D(256, kernel_size=5, padding='same', activation='relu'))
model.add(MaxPool2D())
model.add(Dropout(0.3))

# Adding third convolutional-pooling pair
model.add(Conv2D(512, kernel_size=5, padding='same', activation='relu'))
model.add(MaxPool2D())
model.add(Dropout(0.3))

# Adding fourth convolutional-pooling pair
model.add(Conv2D(1024, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPool2D())
model.add(Dropout(0.3))

# Adding fully connected layers
model.add(Flatten())
model.add(Dense(2048, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(43, activation='softmax'))

# Compiling created model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Saving 1st model for GRAY datasets
# (!) On Windows, it might need to change
# this: + '/' +
```

```
# to this: + '\' +
# or to this: + '\\' +
model.save('ts' + '/' + 'model_1_ts_gray.h5')



# Building 2nd model for RGB datasets
# RGB --> {128C24-P2-D30} --> 256-D30 --> 43

# Initializing model to be as linear stack of layers
model = Sequential()

# Adding first convolutional-pooling pair
model.add(Conv2D(128, kernel_size=24, padding='same', activation='relu', input_shape=(48, 48, 3)))
model.add(MaxPool2D())
model.add(Dropout(0.3))

# Adding fully connected layers
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(43, activation='softmax'))

# Compiling created model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Saving 2nd model for RGB datasets
# (!) On Windows, it might need to change
# this: + '/' +
# to this: + '\' +
# or to this: + '\\' +
model.save('ts' + '/' + 'model_2_ts_rgb.h5')



# Building 2nd model for GRAY datasets
# GRAY --> {128C24-P2-D30} --> 256-D30 --> 43

# Initializing model to be as linear stack of layers
model = Sequential()

# Adding first convolutional-pooling pair
model.add(Conv2D(128, kernel_size=24, padding='same', activation='relu', input_shape=(48, 48, 1)))
model.add(MaxPool2D())
model.add(Dropout(0.3))

# Adding fully connected layers
model.add(Flatten())
```

```
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(43, activation='softmax'))

# Compiling created model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Saving 2nd model for GRAY datasets
# (!) On Windows, it might need to change
# this: + '/' +
# to this: + '\' +
# or to this: + '\\' +
model.save('ts' + '/' + 'model_2_ts_gray.h5')



# Check point
print('4 models are saved successfully')
```

***

The provided code is building and saving four different Convolutional Neural Network (CNN) models designed for the traffic signs dataset. These models are created for both RGB and GRAY color spaces. Here's a step-by-step explanation of the code:

1. **Building the 1st RGB Model:**
    - Initializes a model as a linear stack of layers.
    - Adds four convolutional-pooling pairs with decreasing filter sizes and dropout layers.
    - Adds fully connected layers with dropout.
    - Compiles the model with the Adam optimizer and categorical crossentropy loss.
    - Saves the model to a binary file named **model_1_ts_rgb.h5**.
2. **Building the 1st GRAY Model:**
    - Similar architecture to the RGB model, but input dimensions and layer input shapes are adjusted for GRAY color space.
    - Compiles and saves the model to a file named **model_1_ts_gray.h5**.
3. **Building the 2nd RGB Model (Different Architecture):**
    - Initializes a new model.
    - Adds a convolutional-pooling pair and fully connected layers with dropout.
    - Compiles and saves the model to a file named **model_2_ts_rgb.h5**.
4. **Building the 2nd GRAY Model (Different Architecture):**
    - Similar architecture to the 2nd RGB model, with adjustments for GRAY color space.
    - Compiles and saves the model to a file named **model_2_ts_gray.h5**.
5. **Output Checkpoint:**
    - Prints the message '4 models are saved successfully', indicating that all four models have been saved.

In summary, the code designs and saves four different CNN models tailored for the traffic signs dataset, considering both RGB and GRAY color spaces. The models are saved as binary files for later use.

# 7 Train and design deep CNNs models

## 7.1 Overfit designed deep models with prepared datasets:

### Step 1: Opening preprocessed dataset

```
[ ]: # Opening saved Traffic Signs dataset from HDF5 binary file
     # Initiating File object
     # Opening file in reading mode by 'r'
     # (!) On Windows, it might need to change
     # this: + '/' +
     # to this: + '\' +
     # or to this: + '\\' +
     with h5py.File(full_path_to_Section4 + '/' + 'ts' + '/' +
                    'dataset_ts_rgb_255_mean_std.hdf5', 'r') as f:

         # Showing all keys in the HDF5 binary file
         print(list(f.keys()))

         # Extracting saved arrays for training by appropriate keys
         # Saving them into new variables
         x_train = f['x_train']  # HDF5 dataset
         y_train = f['y_train']  # HDF5 dataset
         # Converting them into Numpy arrays
         x_train = np.array(x_train)  # Numpy arrays
         y_train = np.array(y_train)  # Numpy arrays


         # Extracting saved arrays for validation by appropriate keys
         # Saving them into new variables
         x_validation = f['x_validation']  # HDF5 dataset
         y_validation = f['y_validation']  # HDF5 dataset
         # Converting them into Numpy arrays
         x_validation = np.array(x_validation)  # Numpy arrays
         y_validation = np.array(y_validation)  # Numpy arrays


         # Extracting saved arrays for testing by appropriate keys
         # Saving them into new variables
         x_test = f['x_test']  # HDF5 dataset
         y_test = f['y_test']  # HDF5 dataset
         # Converting them into Numpy arrays
         x_test = np.array(x_test)  # Numpy arrays
         y_test = np.array(y_test)  # Numpy arrays
```

**RGB Traffic Signs dataset (255.0 ==> mean ==> std)**

### Step 2: Converting classes vectors to classes matrices

```
]: # Showing class index from the vector
   print('Class index from vector:', y_train[5])
   print()

   # Preparing classes to be passed into the model
   # Transforming them from vectors to binary matrices
   # It is needed to set relationship between classes to be understood by the algor
   # Such format is commonly used in training and predicting
   y_train = to_categorical(y_train, num_classes = 43)
   y_validation = to_categorical(y_validation, num_classes = 43)


   # Showing shapes of converted vectors into matrices
   print(y_train.shape)
   print(y_validation.shape)
   print()


   # Showing class index from the matrix
   print('Class index from matrix:', y_train[5])
```

**Figure 13 :** Converting classes vectors to matrices RGB TRAFFIC SIGNS dataset.

**RGB Traffic Signs dataset (255.0 ==> mean ==> std)**

## Step 3: Loading saved model

In [ ]:
```
# Loading 1st model for RGB datasets
# (!) On Windows, it might need to change
# this: + '/' +
# to this: + '\' +
# or to this: + '\\' +
model = load_model(full_path_to_Section5 + '/' + 'ts' + '/' + 'model_1_ts_rgb.h5

# Check point
print('Model is successfully loaded')
```

In [ ]:
```
# Showing model's summary in table format
print(model.summary())
print()


# Showing dropout rate
print('Dropout rate: ', model.layers[2].rate)

# Showing strides for the 1st layer (convolutional)
print('Strides of the 1st convolutional layer: ', model.layers[0].strides)

# Showing strides for the 2nd layer (max pooling)
print('Strides of the max pooling layer: ', model.layers[1].strides)
print()

# Showing configurations for entire model
# print(model.get_config())

# Showing configurations for specific layers
print('Full configuration details of the 1st layer:\n', model.get_config()['laye
```

**Figure 14 :** Load models RGB TRAFFIC SIGNS dataset.

**RGB Traffic Signs dataset (255.0 ==> mean ==> std)**

## Step 4: Setting up learning rate & epochs

In [ ]:
```
# Defining number of epochs
epochs = 50

# Defining schedule to update learning rate
learning_rate = LearningRateScheduler(lambda x: 1e-3 * 0.95 ** (x + epochs), ver

# Check point
print('Number of epochs and schedule for learning rate are set successfully')
```

**RGB Traffic Signs dataset (255.0 ==> mean ==> std)**

## Step 5: Overfitting loaded CNN model

In [ ]:
```
# If you're using Nvidia GPU and 'cnngpu' environment, there might be an issue l
'''Failed to get convolution algorithm. This is probably because cuDNN failed to
# In this case, close all Jupyter Notebooks, close Terminal Window or Anaconda P
# Open again just this one Jupyter Notebook and run it


# Training model
h = model.fit(x_train[:100], y_train[:100],
                        batch_size=1,
                        epochs=epochs,
                        validation_data=(x_validation[:500], y_validation[:500])
                        callbacks=[learning_rate],
                        verbose=1)
```

**Figure 15 :** Overfitting and setting up learning rate RGB TRAFFIC SIGNS dataset.

```
In [ ]: # Accuracies of the model
print('Training accuracy={0:.5f}, Validation accuracy={1:.5f}'.
                                                    format(max(h.histor
                                                           max(h.histor
```

```
In [ ]: # Magic function that renders the figure in a jupyter notebook
# instead of displaying a figure object
%matplotlib inline


# Setting default size of the plot
plt.rcParams['figure.figsize'] = (12.0, 6.0)


# Plotting accuracies for every model
plt.plot(h.history['accuracy'], '-o')
plt.plot(h.history['val_accuracy'], '-o')


# Showing Legend
plt.legend(['training accuracy', 'validation accuracy'], loc='upper left', fonts


# Giving name to axes
plt.xlabel('Epoch', fontsize=16)
plt.ylabel('Accuracy', fontsize=16)


# Giving name to the plot
plt.title('Overfitting model for Traffic Signs Dataset', fontsize=16)


# Saving plot
plt.savefig('overfitted_model_of_ts_dataset.png', dpi=500)


# Showing the plot
plt.show()
```

**Figure 16 :** Overfitting resume for the RGB TRAFFIC SIGNS dataset.

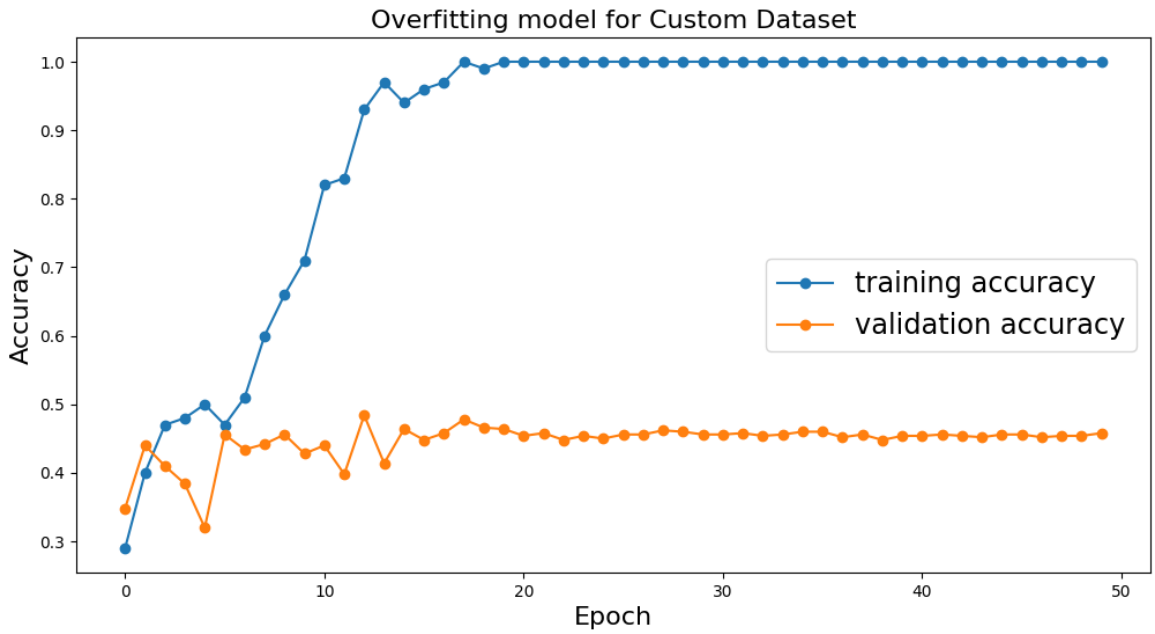



**Figure 17:** Overfitting models graph example.

## 7.2 Train designed deep models with prepared datasets;

## A small code sample for training;

***

```
# If you're using Nvidia GPU and 'cnngpu' environment, there might be an issue like:
'''Failed to get convolution algorithm. This is probably because cuDNN failed to initialize'''
# In this case, close all Jupyter Notebooks, close Terminal Window or Anaconda Prompt
# Open again just this one Jupyter Notebook and run it


# Preparing list with datasets' names
datasets = ['dataset_ts_rgb_255_mean.hdf5',
            'dataset_ts_rgb_255_mean_std.hdf5',
            'dataset_ts_gray_255_mean.hdf5',
            'dataset_ts_gray_255_mean_std.hdf5']


# Defining list to collect results in
h = []


# Training 1st model with all Traffic Signs datasets in a loop
for i in range(4):
    # Opening saved Traffic Signs dataset from HDF5 binary file
    # Initiating File object
    # Opening file in reading mode by 'r'
    # (!) On Windows, it might need to change
    # this: + '/' +
    # to this: + '\' +
    # or to this: + '\\' +
    with h5py.File(full_path_to_Section4 + '/' + 'ts' + '/' + datasets[i], 'r') as f:
        # Extracting saved arrays for training by appropriate keys
        # Saving them into new variables
        x_train = f['x_train']  # HDF5 dataset
        y_train = f['y_train']  # HDF5 dataset
        # Converting them into Numpy arrays
        x_train = np.array(x_train)  # Numpy arrays
        y_train = np.array(y_train)  # Numpy arrays

        # Extracting saved arrays for validation by appropriate keys
        # Saving them into new variables
        x_validation = f['x_validation']  # HDF5 dataset
        y_validation = f['y_validation']  # HDF5 dataset
        # Converting them into Numpy arrays
        x_validation = np.array(x_validation)  # Numpy arrays
        y_validation = np.array(y_validation)  # Numpy arrays


    # Check point
    print('Following dataset is successfully opened:       ', datasets[i])


    # Preparing classes to be passed into the model
    # Transforming them from vectors to binary matrices
    # It is needed to set relationship between classes to be understood by the algorithm
    # Such format is commonly used in training and predicting
    y_train = to_categorical(y_train, num_classes = 43)
    y_validation = to_categorical(y_validation, num_classes = 43)


    # Check point
    print('Binary matrices are successfully created:       ', datasets[i])


    # Preparing filepath to save best weights
    # (!) On Windows, it might need to change
```

```
    # this: + '/' +
    # to this: + '\' +
    # or to this: + '\\' +
    best_weights_filepath = 'ts' + '/' + 'w_1' + datasets[i][7:-5] + '.h5'

    # Formatting options to save all weights for every epoch
    # 'ts' + '/' + 'w_1' + datasets[i][7:-5] + '_{epoch:02d}_{val_accuracy:.4f}' + '.h5'

    # Defining schedule to save best weights
    best_weights = ModelCheckpoint(filepath=best_weights_filepath,
                                   save_weights_only=True,
                                   monitor='val_accuracy',
                                   mode='max',
                                   save_best_only=True,
                                   period=1,
                                   verbose=1)


    # Check point
    print('Schedule to save best weights is created:       ', datasets[i])


    # Checking if RGB dataset is opened
    if i <= 1:
        # Training RGB model with current dataset
        temp = model_rgb[i].fit(x_train, y_train,
                                batch_size=50,
                                epochs=epochs,
                                validation_data=(x_validation, y_validation),
                                callbacks=[learning_rate, best_weights],
                                verbose=1)


        # Adding results of 1st model for current RGB dataset in the list
        h.append(temp)


        # Check points
        print('1st model for RGB is successfully trained on:    ', datasets[i])
        print('Trained weights for RGB are saved successfully:  ', 'w_1' + datasets[i][7:-5] + '.h5')
        print()

    # Checking if GRAY dataset is opened
    elif i >= 2:
        # Training GRAY model with current dataset
        temp = model_gray[i-2].fit(x_train, y_train,
                                   batch_size=50,
                                   epochs=epochs,
                                   validation_data=(x_validation, y_validation),
                                   callbacks=[learning_rate, best_weights],
                                   verbose=1)


        # Adding results of 1st model for current GRAY dataset in the list
        h.append(temp)


        # Check points
        print('1st model for GRAY is successfully trained on:   ', datasets[i])
        print('Trained weights for GRAY are saved successfully: ', 'w_1' + datasets[i][7:-5] + '.h5')
        print()
***
```

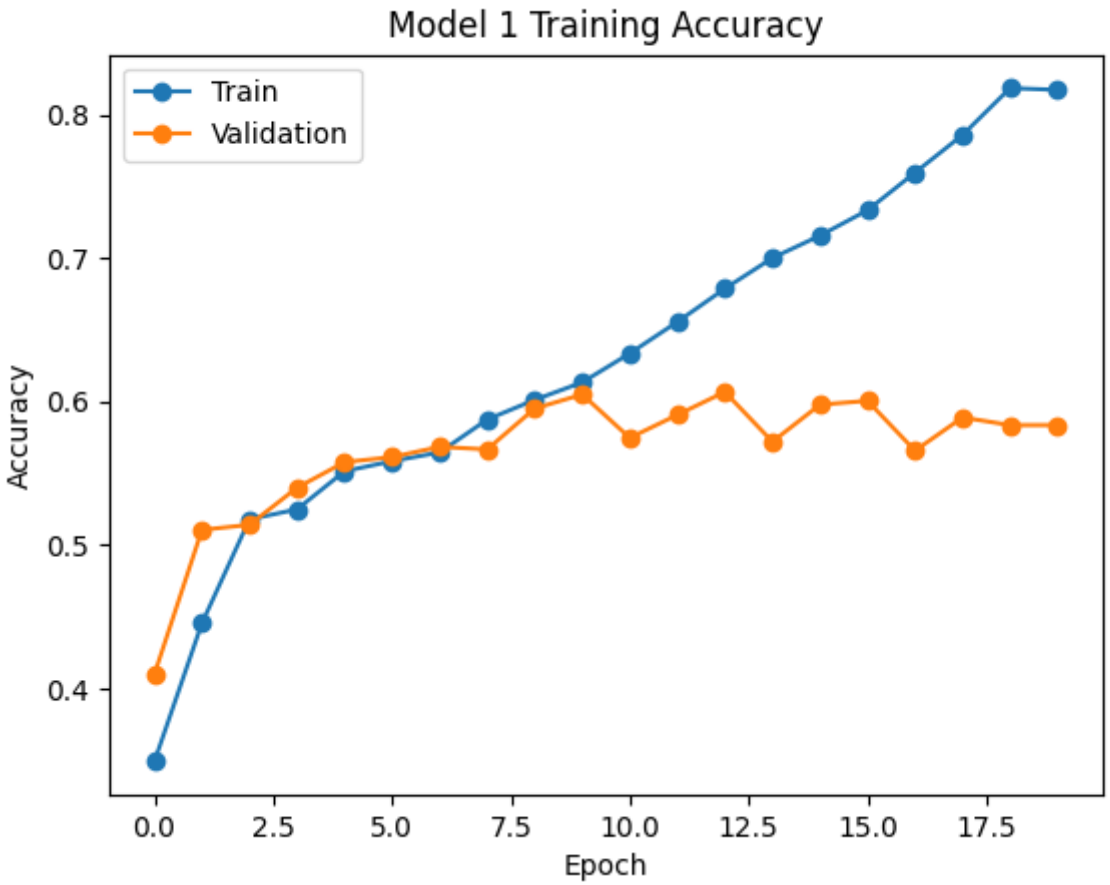



**Figure** 18 : Train graph for the (model 1) h5 file**.**

## 8 Test designed deep CNNs models

**Test trained models;**

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import h5py
import cv2


from keras.models import load_model

from sklearn.metrics import classification_report, confusion_matrix,
ConfusionMatrixDisplay

from timeit import default_timer as timer

# Full or absolute path to 'Section2' with labels for CIFAR-10 dataset
# (!) On Windows, the path should look like following:
# r'C:\Users\your_name\PycharmProjects\CNNCourse\Section2'
# or:
# 'C:\\Users\\your_name\\PycharmProjects\\CNNCourse\\Section2'
full_path_to_Section2 = r'C:\Users\Deniz\Desktop\CNNCOURSE\Section2'
```

```
# Full or absolute path to 'Section3' with labels for Traffic Signs dataset
# (!) On Windows, the path should look like following:
# r'C:\Users\your_name\PycharmProjects\CNNCourse\Section3'
# or:
# 'C:\\Users\\your_name\\PycharmProjects\\CNNCourse\\Section3'
full_path_to_Section3 = r'C:\Users\Deniz\Desktop\CNNCOURSE\Section3'


# Full or absolute path to 'Section4' with preprocessed datasets
# (!) On Windows, the path should look like following:
# r'C:\Users\your_name\PycharmProjects\CNNCourse\Section4'
# or:
# 'C:\\Users\\your_name\\PycharmProjects\\CNNCourse\\Section4'
full_path_to_Section4 = r'C:\Users\Deniz\Desktop\CNNCOURSE\Section4'


# Full or absolute path to 'Section5' with designed models
# (!) On Windows, the path should look like following:
# r'C:\Users\your_name\PycharmProjects\CNNCourse\Section5'
# or:
# 'C:\\Users\\your_name\\PycharmProjects\\CNNCourse\\Section5'
full_path_to_Section6 = r'C:\Users\Deniz\Desktop\CNNCOURSE\Section6'

# Defining lists to collect models in
model_rgb = []
model_gray = []


# Loading 1st model for Traffic Signs dataset
# (!) On Windows, it might need to change
# this: + '/' +
# to this: + '\' +
# or to this: + '\\' +
for i in range(2):
    model_rgb.append(load_model(full_path_to_Section6 + '/' + 'ts' + '/' +
'model_1_ts_rgb.h5'))
    model_gray.append(load_model(full_path_to_Section6 + '/' + 'ts' + '/' +
'model_1_ts_gray.h5'))


# Check point
print('Models are successfully loaded')

# Preparing list with weights' names
weights = ['w_1_ts_rgb_255_mean.h5',
           'w_1_ts_rgb_255_mean_std.h5',
           'w_1_ts_gray_255_mean.h5',
           'w_1_ts_gray_255_mean_std.h5']

# Loading best weights for 1st model
for i in range(4):
    # Checking if it is RGB model
    if i <= 1:
        # loading and assigning best weights
        # (!) On Windows, it might need to change
        # this: + '/' +
        # to this: + '\' +
        # or to this: + '\\' +

model_rgb[i].load_weights(r'C:\Users\Deniz\Desktop\CNNCOURSE\Section6\ts\w_1_ts_rgb_25
5_mean.h5')
```

```
        # Check point
        print('Best weights for 1st RGB model are loaded and assigned  : ',
weights[i])

    # Checking if it is GRAY model
    elif i >= 2:
        # loading and assigning best weights
        # (!) On Windows, it might need to change
        # this: + '/' +
        # to this: + '\' +
        # or to this: + '\\' +
        model_gray[i -
2].load_weights(r'C:\Users\Deniz\Desktop\CNNCOURSE\Section6\ts\w_1_ts_gray_255_mean.h5
')


        # Check point
        print('Best weights for 1st GRAY model are loaded and assigned : ',
weights[i])
# Preparing list with datasets' names
datasets = ['dataset_ts_rgb_255_mean.hdf5',
            'dataset_ts_rgb_255_mean_std.hdf5',
            'dataset_ts_gray_255_mean.hdf5',
            'dataset_ts_gray_255_mean_std.hdf5']

# Defining variable to identify the best model
accuracy_best = 0

# Testing 1st model with all Traffic Signs datasets in a loop
for i in range(4):
    # Opening saved Traffic Signs dataset from HDF5 binary file
    # Initiating File object
    # Opening file in reading mode by 'r'
    # (!) On Windows, it might need to change
    # this: + '/' +
    # to this: + '\' +
    # or to this: + '\\' +
    with h5py.File(full_path_to_Section6 + '/' + 'ts' + '/' + datasets[i], 'r') as f:
        # Extracting saved arrays for testing by appropriate keys
        # Saving them into new variables
        x_test = f['x_test']  # HDF5 dataset
        y_test = f['y_test']  # HDF5 dataset
        # Converting them into Numpy arrays
        x_test = np.array(x_test)  # Numpy arrays
        y_test = np.array(y_test)  # Numpy arrays

    # Check point
    print('Dataset is opened :', datasets[i])

    # Check point
    # Showing shapes of loaded arrays
    if i == 0:
        print('x_test shape      :', x_test.shape)
        print('y_test shape      :', y_test.shape)

    # Checking if RGB dataset is opened
    if i <= 1:
        # Testing RGB model with current dataset
        temp = model_rgb[i].predict(x_test)
```

```
        # Check point
        # Showing prediction shape and scores
        if i == 0:
            print('prediction shape  :', temp.shape)  # (3111, 43)
            print('prediction scores :', temp[0, 0:5])  # 5 score numbers

        # Getting indexes of maximum values along specified axis
        temp = np.argmax(temp, axis=1)

        # Check point
        # Showing prediction shape after convertion
        # Showing predicted and correct indexes of classes
        if i == 0:
            print('prediction shape  :', temp.shape)  # (3111,)
            print('predicted indexes :', temp[0:10])
            print('correct indexes   :', y_test[:10])

        # Calculating accuracy
        # We compare predicted class with correct class for all input images
        # By saying 'temp == y_test' we create Numpy array with True and False values
        # By function 'np.mean' we calculate mean value:
        # all_True / (all_True + all_False)
        accuracy = np.mean(temp == y_test)

        # Check point
        # Showing True and False matrix
        if i == 0:
            print('T and F matrix    :', (temp == y_test)[0:10])

        # Check point
        # Showing calculated accuracy
        print('Testing accuracy  : {0:.5f}'.format(accuracy))
        print()

        # Confusion matrix is a two dimensional matrix that visualizes the
performance,
        # and makes it easy to see confusion between classes,
        # by providing a picture of interrelation

        # Each row represents a number of actual class
        # Each column represents a number of predicted class

        # Computing confusion matrix to evaluate accuracy of classification
        c_m = confusion_matrix(y_test, temp)

        # Showing confusion matrix in the form of Numpy array
        print(c_m)

    # Checking if GRAY dataset is opened
    elif i >= 2:
        # Testing GRAY model with current dataset
        temp = model_gray[i - 2].predict(x_test)

        # Getting indexes of maximum values along specified axis
        temp = np.argmax(temp, axis=1)

        # Calculating accuracy
        # We compare predicted class with correct class for all input images
        # By saying 'temp == y_test' we create Numpy array with True and False values
        # By function 'np.mean' we calculate mean value:
        # all_True / (all_True + all_False)
        accuracy = np.mean(temp == y_test)
```

```
        # Check point
        # Showing calculated accuracy
        print('Testing accuracy  : {0:.5f}'.format(accuracy))
        print()

    # Identifying the best model
    # Saving predicted indexes of the best model
    if accuracy > accuracy_best:
        # Updating value of the best accuracy
        accuracy_best = accuracy

        # Saving predicted indexes of the best model into array
        # Updating array with predicted indexes of the best model
        y_predicted_best = temp

        # Showing the main classification metrics of the best model
        print(classification_report(y_test, y_predicted_best))

        # Confusion matrix is a two dimensional matrix that visualizes the
performance,
        # and makes it easy to see confusion between classes,
        # by providing a picture of interrelation

        # Each row represents a number of actual class
        # Each column represents a number of predicted class

        # Computing confusion matrix to evaluate accuracy of classification
        c_m = confusion_matrix(y_test, y_predicted_best)

        # Showing confusion matrix in form of Numpy array
        print(c_m)

# Preparing labels for Traffic Signs dataset
# Getting Pandas dataFrame with labels
# (!) On Windows, it might need to change
# this: + '/' +
# to this: + '\' +
# or to this: + '\\' +
labels_ts = pd.read_csv(full_path_to_Section3 + '/' + 'classes_names.csv', sep=',')


# Check point
# Showing first 5 elements of the dataFrame
print(labels_ts.head())
print()


# Showing class's name of the 1st element
print(labels_ts.loc[0, 'SignName'])
print()


# Converting into Numpy array
labels_ts = np.array(labels_ts.loc[:, 'SignName']).flatten()


# Check points
# Showing size of Numpy array
# Showing all elements of Numpy array
print('Total number of labels:', labels_ts.size)
print()
```

```
print(labels_ts)

# Magic function that renders the figure in a jupyter notebook
# instead of displaying a figure object



# Setting default size of the plot
# Setting default fontsize used in the plot
plt.rcParams['figure.figsize'] = (14.0, 14.0)
plt.rcParams['font.size'] = 12


# Implementing visualization of confusion matrix
display_c_m = ConfusionMatrixDisplay(c_m)


# Plotting confusion matrix
# Setting colour map to be used
display_c_m.plot(cmap='PuRd')
# Other possible options for colour map are:
# 'OrRd', 'autumn_r', 'Blues', 'cool', 'Greens', 'Greys', 'copper_r'


# Setting fontsize for xticks and yticks
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)


# Setting fontsize for xlabels and ylabels
plt.xlabel('Predicted label', fontsize=18)
plt.ylabel('True label', fontsize=18)


# Giving name to the plot
plt.title('Confusion Matrix: 1st model, Traffic Signs Dataset', fontsize=18)


# Saving plot
plt.savefig('confusion_matrix_model_1_ts_dataset.png', transparent=True, dpi=500)


# Showing the plot
plt.show()

# Opening saved Mean Image for RGB Traffic Signs dataset
# Initiating File object
# Opening file in reading mode by 'r'
# (!) On Windows, it might need to change
# this: + '/' +
# to this: + '\' +
# or to this: + '\\' +
with
h5py.File(r'C:\Users\Deniz\Desktop\CNNCOURSE\Section6\ts\mean_rgb_dataset_ts.hdf5',
'r') as f:

    # Extracting saved array for Mean Image
    # Saving it into new variable
    mean_rgb = f['mean']  # HDF5 dataset
    # Converting it into Numpy array
    mean_rgb = np.array(mean_rgb)  # Numpy arrays
```

```
# Opening saved Standard Deviation for RGB Traffic Signs dataset
# Initiating File object
# Opening file in reading mode by 'r'
# (!) On Windows, it might need to change
# this: + '/' +
# to this: + '\' +
# or to this: + '\\' +
with
h5py.File(r'C:\Users\Deniz\Desktop\CNNCOURSE\Section6\ts\std_rgb_dataset_ts.hdf5',
'r') as f:

    # Extracting saved array for Standard Deviation
    # Saving it into new variable
    std_rgb = f['std']  # HDF5 dataset
    # Converting it into Numpy array
    std_rgb = np.array(std_rgb)  # Numpy arrays


# Opening saved Mean Image for GRAY Traffic Signs dataset
# Initiating File object
# Opening file in reading mode by 'r'
# (!) On Windows, it might need to change
# this: + '/' +
# to this: + '\' +
# or to this: + '\\' +
with
h5py.File(r'C:\Users\Deniz\Desktop\CNNCOURSE\Section6\ts\mean_gray_dataset_ts.hdf5',
'r') as f:
    # Extracting saved array for Mean Image
    # Saving it into new variable
    mean_gray = f['mean']  # HDF5 dataset
    # Converting it into Numpy array
    mean_gray = np.array(mean_gray)  # Numpy arrays


# Opening saved Standard Deviation for GRAY Traffic Signs dataset
# Initiating File object
# Opening file in reading mode by 'r'
# (!) On Windows, it might need to change
# this: + '/' +
# to this: + '\' +
# or to this: + '\\' +
with
h5py.File(r'C:\Users\Deniz\Desktop\CNNCOURSE\Section6\ts\std_gray_dataset_ts.hdf5',
'r') as f:
    # Extracting saved array for Standard Deviation
    # Saving it into new variable
    std_gray = f['std']  # HDF5 dataset
    # Converting it into Numpy array
    std_gray = np.array(std_gray)  # Numpy arrays


# Check points
# Showing shapes of loaded Numpy arrays
print('RGB Mean Image          :', mean_rgb.shape)
print('RGB Standard Deviation  :', std_rgb.shape)
print('GRAY Mean Image         :', mean_gray.shape)
print('GRAY Standard Deviation :', std_gray.shape)

# Magic function that renders the figure in a jupyter notebook
# instead of displaying a figure object
```

```
# Setting default size of the plot
plt.rcParams['figure.figsize'] = (2.5, 2.5)



# Reading image by OpenCV library
# In this way image is opened already as Numpy array
# (!) OpenCV by default reads images in BGR order of channels
# (!) On Windows, it might need to change
# this: + '/' +
# to this: + '\' +
# or to this: + '\\' +
# Dosya adını düzenleyin (örneğin: ts_to_test_1.jpg)
import cv2

# Dosya adını düzenleyin (örneğin: ts_to_test_1.jpg)
image_file_name = '05004.ppm'

# Dosya yolunu düzenleyin
image_path = r'C:\Users\Deniz\Desktop\CNNCOURSE\Section6\Images' + '\\' +
image_file_name

# Görüntüyü oku
image_ts_bgr = cv2.imread(image_path)

# Görüntü başarıyla okunduysa devam edin
if image_ts_bgr is not None:
    # Görüntüyü RGB formatına çevirin
    image_ts_rgb = cv2.cvtColor(image_ts_bgr, cv2.COLOR_BGR2RGB)

    # Görüntüyü yeniden boyutlandırın (örneğin, 48x48 piksel)
    image_ts_rgb = cv2.resize(image_ts_rgb, (48, 48), interpolation=cv2.INTER_CUBIC)

    # ... diğer işlemleri yapabilirsiniz
else:
    print("Görüntü okuma hatası. Dosya yolunu kontrol edin ve dosyanın doğru
olduğundan emin olun.")


# Check point
# Showing loaded and resized image
plt.imshow(image_ts_rgb)
plt.show()



# Implementing normalization by dividing image's pixels on 255.0
image_ts_rgb_255 = image_ts_rgb / 255.0

# Implementing normalization by subtracting Mean Image
image_ts_rgb_255_mean = image_ts_rgb_255 - mean_rgb

# Implementing preprocessing by dividing on Standard Deviation
image_ts_rgb_255_mean_std = image_ts_rgb_255_mean / std_rgb

# Check points
# Showing shape of Numpy array with RGB image
# Showing some pixels' values
print('Shape of RGB image         :', image_ts_rgb.shape)
print('Pixels of RGB image        :', image_ts_rgb[:5, 0, 0])
```

```
print('RGB /255.0                 :', image_ts_rgb_255[:5, 0, 0])
print('RGB /255.0 => mean         :', image_ts_rgb_255_mean[:5, 0, 0])
print('RGB /255.0 => mean => std  :', image_ts_rgb_255_mean_std[:5, 0, 0])
print()



# Converting image to GRAY by OpenCV function
image_ts_gray = cv2.cvtColor(image_ts_rgb, cv2.COLOR_RGB2GRAY)

# Extending dimension from (height, width) to (height, width, one channel)
image_ts_gray = image_ts_gray[:, :, np.newaxis]

# Check point
# Showing converted into GRAY image
plt.imshow(image_ts_gray, cmap=plt.get_cmap('gray'))
plt.show()



# Implementing normalization by dividing image's pixels on 255.0
image_ts_gray_255 = image_ts_gray / 255.0

# Implementing normalization by subtracting Mean Image
image_ts_gray_255_mean = image_ts_gray_255 - mean_gray

# Implementing preprocessing by dividing on Standard Deviation
image_ts_gray_255_mean_std = image_ts_gray_255_mean / std_gray

# Check points
# Showing shape of Numpy array with GRAY image
# Showing some pixels' values
print('Shape of GRAY image        :', image_ts_gray.shape)
print('Pixels of GRAY image       :', image_ts_gray[:5, 0, 0])
print('GRAY /255.0                :', image_ts_gray_255[:5, 0, 0])
print('GRAY /255.0 => mean        :', image_ts_gray_255_mean[:5, 0, 0])
print('GRAY /255.0 => mean => std :', image_ts_gray_255_mean_std[:5, 0, 0])


# Extending dimension from (height, width, channels) to (1, height, width, channels)
image_ts_rgb_255_mean = image_ts_rgb_255_mean[np.newaxis, :, :, :]
image_ts_rgb_255_mean_std = image_ts_rgb_255_mean_std[np.newaxis, :, :, :]

image_ts_gray_255_mean = image_ts_gray_255_mean[np.newaxis, :, :, :]
image_ts_gray_255_mean_std = image_ts_gray_255_mean_std[np.newaxis, :, :, :]

# Check points
# Showing shapes of extended Numpy arrays
print('RGB /255.0 => mean         :', image_ts_rgb_255_mean.shape)
print('RGB /255.0 => mean => std  :', image_ts_rgb_255_mean_std.shape)
print()
print('GRAY /255.0 => mean        :', image_ts_gray_255_mean.shape)
print('GRAY /255.0 => mean => std :', image_ts_gray_255_mean_std.shape)

import matplotlib.pyplot as plt
import numpy as np

def bar_chart(data, bar_title, show_xticks=True):
    """
    Oluşturulan bir çubuk grafiğini gösteren fonksiyon.

    Parameters:
```

```
    - data: numpy.ndarray, çubuk grafiği verisi
    - bar_title: str, çubuk grafiğinin başlığı
    - show_xticks: bool, X eksenindeki etiketleri gösterme veya gösterme
    """
    classes = np.arange(len(data))
    plt.bar(classes, data)
    plt.title(bar_title)
    plt.xlabel('Class Index')  # X ekseni etiketi
    plt.ylabel('Score')  # Y ekseni etiketi
    if not show_xticks:
        plt.xticks([])  # X eksenindeki etiketleri gizle
    plt.show()

# Check point
# Showing information about created function
print(help(bar_chart))

# Magic function that renders the figure in a jupyter notebook
# instead of displaying a figure object


# Setting default size of the plot
plt.rcParams['figure.figsize'] = (12, 7)



# Testing RGB model trained on dataset: dataset_ts_rgb_255_mean.hdf5
# Input image is preprocessed in the same way
# Measuring classification time
start = timer()
scores = model_rgb[0].predict(image_ts_rgb_255_mean)
end = timer()

# Scores are given as 43 numbers of predictions for each class
# Getting only one class with maximum value
prediction = np.argmax(scores)

# Check points
# Showing scores shape and values
# Printing class index, label and time
print()
print('Scores shape        :', scores.shape)
print('Scores values       :', scores[0, 10:15])
print('Scores sum          :', scores[0].sum())
print('Score of prediction : {0:.5f}'.format(scores[0][prediction]))
print('Class index         :', prediction)
print('Label               :', labels_ts[prediction])
print('Time                : {0:.5f}'.format(end - start))

# Plotting bar chart with scores values
bar_chart(scores[0],
          bar_title='1st RGB model, ts_rgb_255_mean',
          show_xticks=False)



# Testing RGB model trained on dataset: dataset_ts_rgb_255_mean_std.hdf5
# Input image is preprocessed in the same way
# Measuring classification time
start = timer()
scores = model_rgb[1].predict(image_ts_rgb_255_mean_std)
end = timer()
```

```
# Scores are given as 43 numbers of predictions for each class
# Getting only one class with maximum value
prediction = np.argmax(scores)

# Check points
# Showing scores shape and values
# Printing class index, label and time
print()
print('Scores shape        :', scores.shape)
print('Scores values       :', scores[0, 10:15])
print('Scores sum          :', scores[0].sum())
print('Score of prediction : {0:.5f}'.format(scores[0][prediction]))
print('Class index         :', prediction)
print('Label               :', labels_ts[prediction])
print('Time                : {0:.5f}'.format(end - start))

# Plotting bar chart with scores values
bar_chart(scores[0],
          bar_title='1st RGB model, ts_rgb_255_mean_std',
          show_xticks=False)



# Testing GRAY model trained on dataset: dataset_ts_gray_255_mean.hdf5
# Input image is preprocessed in the same way
# Measuring classification time
start = timer()
scores = model_gray[0].predict(image_ts_gray_255_mean)
end = timer()

# Scores are given as 43 numbers of predictions for each class
# Getting only one class with maximum value
prediction = np.argmax(scores)

# Check points
# Showing scores shape and values
# Printing class index, label and time
print()
print('Scores shape        :', scores.shape)
print('Scores values       :', scores[0, 10:15])
print('Scores sum          :', scores[0].sum())
print('Score of prediction : {0:.5f}'.format(scores[0][prediction]))
print('Class index         :', prediction)
print('Label               :', labels_ts[prediction])
print('Time                : {0:.5f}'.format(end - start))

# Plotting bar chart with scores values
bar_chart(scores[0],
          bar_title='1st GRAY model, ts_gray_255_mean',
          show_xticks=False)



# Testing GRAY model trained on dataset: dataset_ts_gray_255_mean_std.hdf5
# Input image is preprocessed in the same way
# Measuring classification time
start = timer()
scores = model_gray[1].predict(image_ts_gray_255_mean_std)
end = timer()

# Scores are given as 43 numbers of predictions for each class
```

```
# Getting only one class with maximum value
prediction = np.argmax(scores)

# Check points
# Showing scores shape and values
# Printing class index, label and time
print()
print('Scores shape        :', scores.shape)
print('Scores values       :', scores[0, 10:15])
print('Scores sum          :', scores[0].sum())
print('Score of prediction : {0:.5f}'.format(scores[0][prediction]))
print('Class index         :', prediction)
print('Label               :', labels_ts[prediction])
print('Time                : {0:.5f}'.format(end - start))

# Plotting bar chart with scores values
bar_chart(scores[0],
          bar_title='1st GRAY model, ts_gray_255_mean_std',
          show_xticks=False)
```

**and this codes give the results ;**

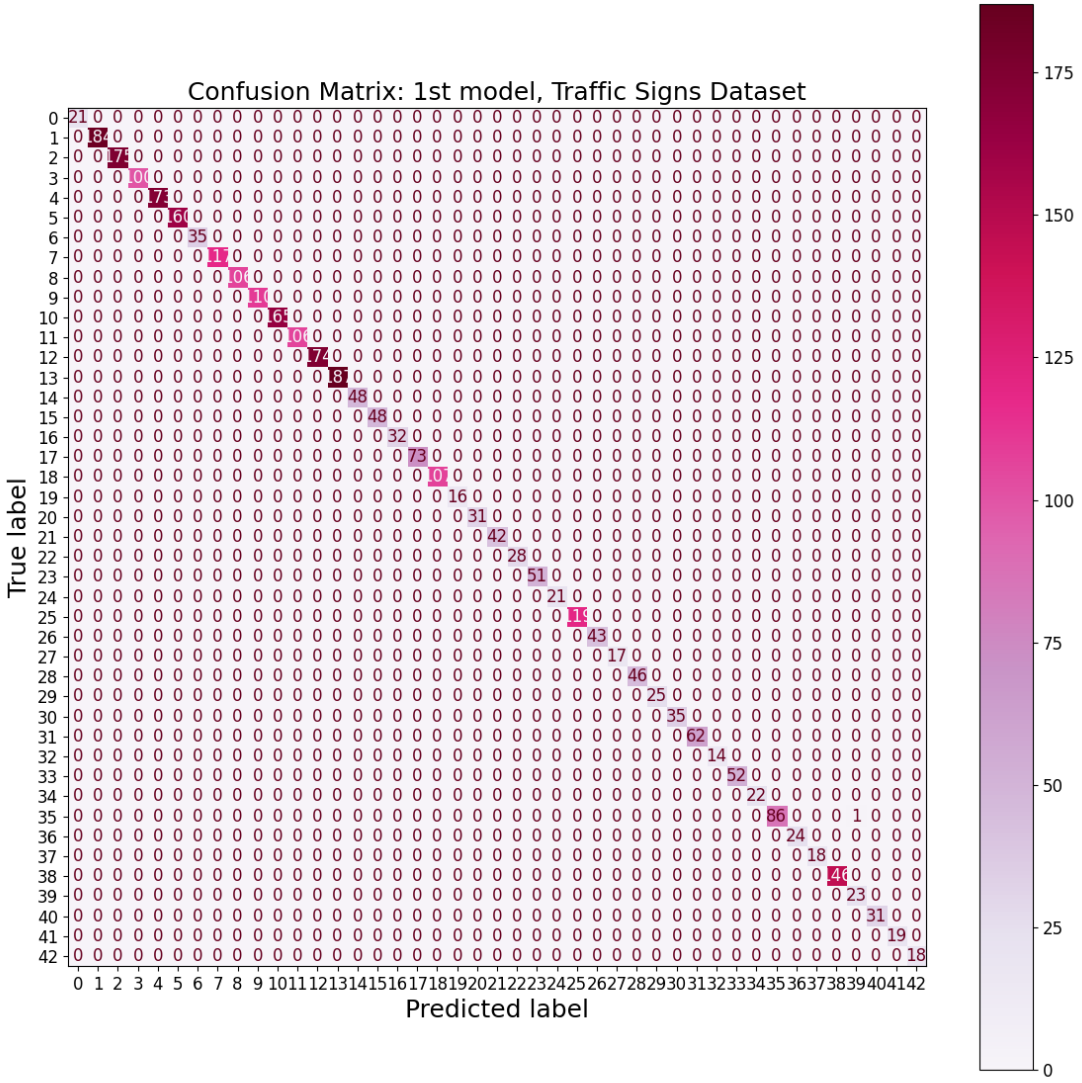



Figure 19: Confusion Matrix with the accuracy values.

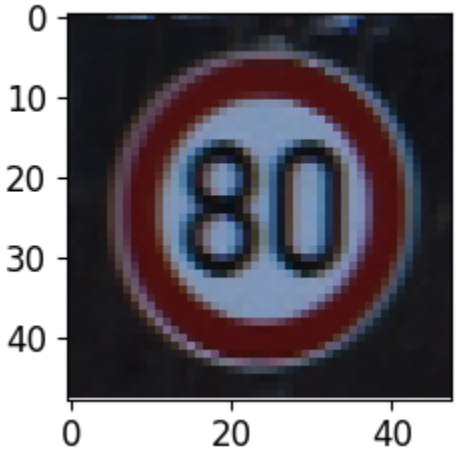



**Figure 20: Colour traffic sign**

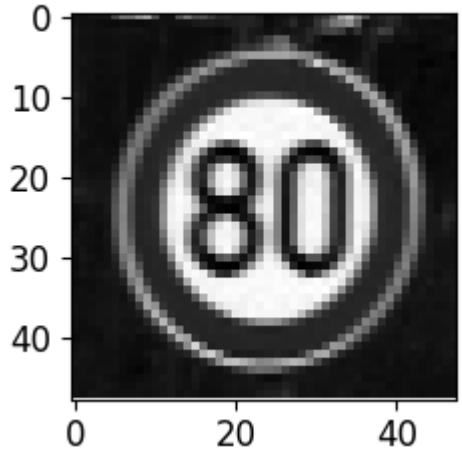



**Figure 21: Gray traffic sign**

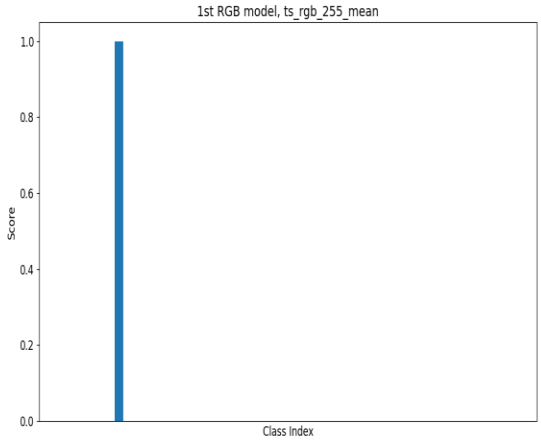



**Figure 22: Classification result for the model_ts_rgb_mean.h5**

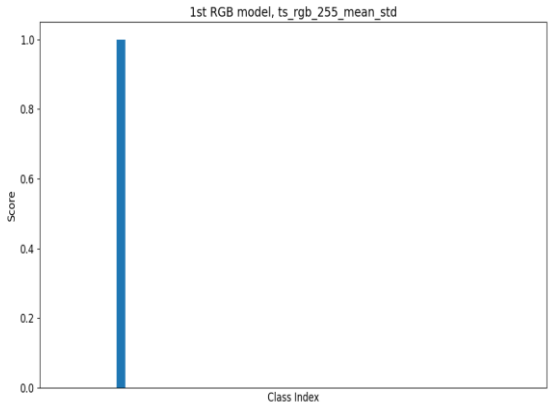



**Figure 23: Classification result for the model_ts_rgb_mean_std.h5**

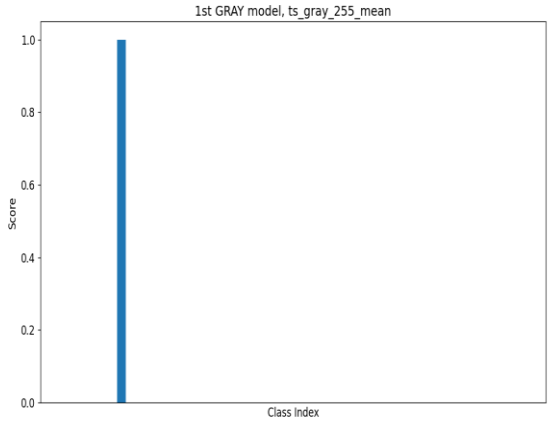



**Figure 24: Classification result for the model_ts_gray_mean.h5**

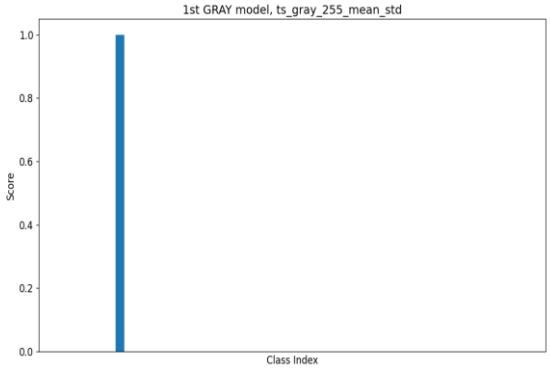



**Figure 25: Classification result for the model_gray_gray_mean_std.h5**

# 9 Results and Discussion

In this research, Convolutional Neural Network algorithm was used to recognize traffic signs. It is anticipated that these determined parameters will contribute to improving the overall performance, especially for autonomous driving vehicles. However, practical applications require consideration of various factors and a real comprehensive evaluation of the results obtained is mandatory.

## 9.1 Conclusion

In this research, CNN algorithm was used to recognize traffic signs. In this research, traffic signs recognized for autonomous driving system; While it provides a high level of comfort to vehicles while driving, it does not change the fact that although the traffic sign recognition accuracy is high, there is a margin of error. Although it is highly reliable, today it needs to be evaluated in autonomous driving tests and the margin of error should be minimized by training. It can be easily preferred because it does not cost much.

## 9.2 References


1)A. Nikonorov, P. Yakimov, M. Petrov, Traffic sign detection on GPU using color shape regular expressions, VISIGRAPP IMTA-4, Paper 8 (2013).

2)R. Belaroussi, P. Foucher, J.P. Tarel, B. Soheilian, P. Charbonnier, N. Paparoditis Road Sign Detection in Images
A Case Study, 20th International Conference on Pattern Recognition (ICPR) (2010), pp. 484-488

3)A. Ruta, F. Porikli, Y. Li, S. Watanabe, H. Kage, K. Sumi, A New Approach for In-Vehicle Camea Traffic Sign Detection and Recognition, IAPR Conference on Machine Vision Applications (MVA), Session 15: Machine Vision for Transportation, 2009.

4)S. Houben, J. Stallkamp, J. Salmen, M. Schlipsing, C. Igel, Detection of Traffic Signs in Real-World Images: The {G}erman {T}raffic
{S}ign {D}etection {B}enchmark, in: Proc. International Joint Conference on Neural Networks, 2013.

5)Z. Zhu, D. Liang, S. Zhang, X. Huang, B. Li, S. Hu, Traffic-Sign Detection and Classification in the Wild. Proceedings of CVPR, 2016, pp. 2110-2118.

6)Y. LeCun, P. Sermanet, Traffic Sign Recognition with Multi-Scale Convolutional Networks, Proceedings of International Joint
Conference on Neural Networks (IJCNN'11), 2011.

7)H Aghdam, E. Heravi, D. Puig
A practical approach for detection and classification of traffic signs using Convolutional Neural Networks.
8)Vlentyn Sichkar , Convolutional Neural Networks for Image Classification.

9) A. Nikonorov, P. Yakimov, M. Petrov, Traffic sign detection on GPU using color shape regular expressions, VISIGRAPP IMTA-4, Paper 8 (2013).

10) A. Ruta, F. Porikli, Y. Li, S. Watanabe, H. Kage, K. Sumi, A New Approach for In-Vehicle Camea Traffic Sign Detection and Recognition, IAPR Conference on Machine Vision Applications (MVA), Session 15: Machine Vision for Transportation, 2009.

11) S. Houben, J. Stallkamp, J. Salmen, M. Schlipsing, C. Igel, Detection of Traffic Signs in Real-World Images: The {G}erman {T}raffic {S}ign {D}etection {B}enchmark, in: Proc. International Joint Conference on Neural Networks, 2013.

12) Z. Zhu, D. Liang, S. Zhang, X. Huang, B. Li, S. Hu, Traffic-Sign Detection and Classification in the Wild. Proceedings of CVPR, 2016, pp. 2110-2118.

13) Y. LeCun, P. Sermanet, Traffic Sign Recognition with Multi-Scale Convolutional Networks, Proceedings of International Joint Conference on Neural Networks (IJCNN'11), 2011.

14) A practical approach for detection and classification of traffic signs using Convolutional Neural Networks

| Thesis Check List | YES(Y) / NO(N) |
|---|---|
| Engineering Design or Graduation Project report has the same template given in the departmentweb page? | Y |
| Report includes whole sections given in the thesis template. (Özet, abstract, introduction etc.)? | Y |
| Introduction part covers state of the art studies in literature? Also, short and clear explanation about the main study of the thesis given at the end of the introduction part? | Y |
| Thesis has at least 10 citation and 6 of them are from state of art articles, conference papers andthesis? | Y |
| Equations, tables and figures are created by the student? | Y |
| Tables, figures etc. are used in the result section to show outcomes of the study. Results of the study is discussed in conclusion section. | Y |

## Personal Information

Name / Surname: Deniz Geçmiş

Nationality : Turkish

Birth Place and Date: Yüreğir / Adana /10.19.2000

Telephone No: +905314672461

E-mail : denizgecmis123@gmail.com