Avlonitis Ektor

Kalogeris Spyros

Kapolis George

Traffic Sign Information

Machine Learning & Applications

Table of Contents

Introduction 3

Brief Overview 3

Techniques Employed 3

Dataset 3

Traffic Sign Classification 4

Convolution Neural Network Code Overview 4

Convolution Neural Network 5

Object Detection Methods 6

Brute Force Approach 6

Steps for Brute Force Object Detection in Video 7

Brute Force Code Overview 7

Selective Search Algorithm 8

Steps for Selective Search Object Detection in Image 9

Selective Search Code Overview 10

YOLO 11

Steps used for YOLO 11

YOLO Code Overview 12

Results and Discussion 12

Introduction

Brief Overview

The Traffic Sign Information project focuses on building a deep neural network model for traffic sign classification. The goal is to develop a system that can accurately identify and classify different types of traffic signs in images. The project utilizes a dataset obtained from Kaggle, which contains over 50,000 images of various traffic signs categorized into 43 classes.

The project is divided into four main steps. The first step involves exploring the dataset, understanding its structure, and gaining insights into the distribution of images across different classes. The second step is to build a Convolutional Neural Network (CNN) model. The model architecture is designed to learn and extract relevant features from traffic sign images, enabling accurate classification. Once the model is built, the third step involves training and validating the model using the dataset and in the fourth step, the trained model is tested using a separate test dataset. This step assesses the model's performance on unseen traffic sign images and provides insights into its real-world applicability.

At the end of this report, we need to have demonstrated the following three versions of the system. Firstly, the classifier's functionality in a given input image, which is showcased by applying bounding boxes around a traffic sign and providing the sign's interpretation in the terminal or screen. The second version focuses on real-time classification. A camera facing a traffic sign is used to capture live video input, and the model performs continuous classification in real time. Finally, a video is recorded while driving through city streets, and the system is employed for real-time detection, recognition, and classification of traffic signs along the recorded path.

Techniques Employed

For traffic sign **classification**, **one** convolutional neural network (CNN) architecture was implemented. However, due to several setbacks and for accuracy reasons many different models were trained which are going to be discussed below. For example, in order to get more accurate results using the brute force approach in object detection, a model was trained only on half of the traffic sign classes.

For object **detection**, **three** approaches were implemented: brute force, selective search, and YOLOv8. The brute force approach uses a sliding window technique to scan the image systematically, but it is computationally expensive and prone to false positives. Selective search generates potential object regions based on similarity measures, reducing computational burden. YOLOv8 is a state-of-the-art algorithm that divides the image into a grid and predicts objects in each grid cell and works better than the rest.

Dataset

Our dataset comprises a collection of 43 different traffic signs. Each traffic sign category is represented by 6 to 74 photographs. To augment the dataset and enhance its diversity, the images were resized in total 30 times from the original 30x30 resolution to a larger dimension of 144x144 pixels. This resizing step increases the dataset size and allows for better training of models that require more training examples. By incorporating a variety of traffic sign images at an increased resolution, our dataset aims to provide a comprehensive and representative set of training samples for traffic sign recognition and classification tasks.

Traffic Sign Classification

The transformation applied to the training data and the input image before passing it to the CNN model involves resizing the images and converting them to PyTorch tensors. Initially, we experimented with different resize dimensions, such as 32x32 and 128x128, in an attempt to find the optimal size. However, we found that resizing the images to 64x64 pixels provided the best results in terms of accuracy and performance. We also explored larger dimensions like 256x256 and 512x512, but these sizes were computationally expensive and significantly increased the processing time. Therefore, we decided to prioritize a balance between accuracy and computational efficiency, settling on the 64x64 resize dimension for the training data and input images.

Convolution Neural Network Code Overview

The *TrafficSignsDataset* class is created to preprocess images for training by loading them along with data from a .csv file in a specified directory. The *getitem* method retrieves a specific sample defined by the *idx* parameter, returning the image and its corresponding label. The transformation process involves resizing the input images to 64x64 pixels and converting them to PyTorch tensors, normalizing pixel values from 0 to 1.

Two datasets, one for training and one for testing, are created using the *TrafficSignsDataset* class, with appropriate arguments provided. DataLoaders are then created to load data in batches during training and testing.

The *CNNModel* class defines a convolutional neural network model for image classification. It takes the number of classes as a parameter and defines various layers, such as convolutional, ReLU activation, max-pooling, dropout, and fully connected layers. The *forward* method is implemented to pass the input tensor through the layers sequentially, returning the output logits.

An instance of the model is created, and the loss function is defined as the *cross-entropy loss*, which combines a softmax activation function and the negative log-likelihood loss. The optimizer, *Adam*, is initialized with the model parameters and a learning rate of 0.001.

During the training process, the model is trained on the training dataset for a certain number of epochs. In each epoch, the model parameters are optimized by minimizing the loss

through backpropagation. After training, the model's performance is evaluated on the test set, calculating the validation accuracy by comparing predicted labels with true labels.

Convolution Neural Network

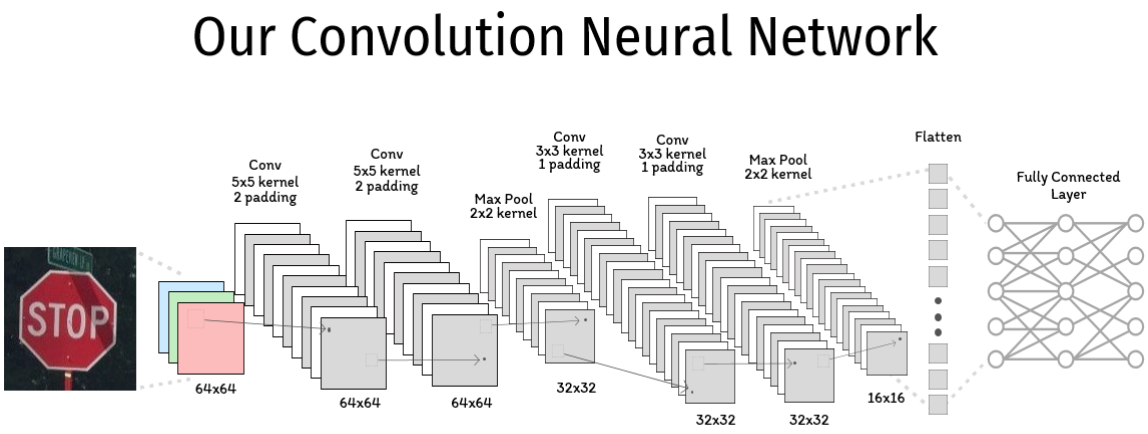

Figure 1. Our Convolution Neural Network

As seen on our model above, the layers of the model include:

4 Convolutional Layers: nn.Conv2d().

- conv1, with 3 input channels, 32 output channels, 5*5 filters (kernel), 1 stride, 2 pixels of zero padding.
- Conv2, with 32 input channels, 32 output channels, 5*5 filters (kernel), 1 stride, 2 pixels of zero padding.
- Conv3, with 32 input channels, 64 output channels, 3*3 filters (kernel), 1 stride, 1 pixel of zero padding.
- Conv4, with 64 input channels, 64 output channels, 3*3 filters (kernel), 1 stride, 1 pixel of zero padding.

ReLu activation layers: nn.ReLU()

MaxPooling layers: nn.MaxPool2d()

- A 2*2 window is used to down-sample the input.

Dropout layers: nn.Dropout()

- 25% probability of each neuron to be set to zero, during training.
- 50% probability of each neuron to be set to zero, during training.

Fully connected layers: nn.Linear()

- fc1, dense of input size 64 channels and dimensions 16*16 which produces an output tensor of 512 size.

- fc2, dense of input 512 neurons and output equals the number of classes.

The choice of using two 5x5 kernels and two 3x3 kernels was decided by the observation that these filter sizes are effective in capturing different levels of spatial information in the images. For each kernel padding was used to preserve the spatial dimensions of the input data, and thus padding equals to 2 and to 1 were used respectively. Also, we used a stride value of 1 in the convolutional layers because a higher stride value would decrease the validation accuracy. The increase in the number of output channels from 32 to 64 in the later layers allows the model to learn more complex and abstract features.

We used a 2x2 window for down-sampling to help reduce computational complexity and prevent overfitting. Also, dropout layers were used to prevent overfitting in the neural network. In this case, two dropout layers with dropout probabilities of 25% and 50% are used after the second and third convolutional layers, respectively.

The fully connected layers are used to transform the features learned by convolutional layers into class probabilities. The first fully connected layer (fc1) produces an output tensor of size 512, acting as a block to reduce the number of parameters before the final classification. The second fully connected layer (fc2) outputs a tensor with dimensions equal to the number of classes representing the probabilities for each class.

Object Detection Methods

Object detection aims to not only classify the objects present in the scene but also determine their precise locations by drawing bounding boxes around them.

Brute Force Approach

This approach is primarily used for traffic sign detection in **images**.

The brute force approach refers to one way of performing object detection using a sliding window. In this method, instead of directly analyzing the entire image, the algorithm scans the image using a small window or sub-image. It starts from one corner of the image and moves the window across the image in a systematic manner, examining each section for the presence of the object of interest, such as the traffic sign in our given example.

By sliding the window across the image, the algorithm exhaustively searches different regions, hoping to find the object in question. In the case of the traffic sign, it checks each section of the image to see if the Convolution Neural Network will recognize the sign. If a match is found, the algorithm considers that region as a potential detection.

However, this brute force sliding window approach has several drawbacks. One major limitation is its inefficiency, as it requires checking the same image multiple times for objects of different sizes. Additionally, the method may generate false positives, depending on the

size and position of the object within the image and it is computationally expensive and may not scale well to larger datasets or more complex objects.

Steps for Brute Force Object Detection in Video

In one of our brute force object detector, we used a fixed window size and stride, ensuring consistent coverage across the frames.

For each frame, we obtained a picture that showed the bounding boxes drawn around the detected objects, along with the corresponding probabilities. To document our findings, we displayed the processed frames one after the other on one laptop screen and recorded it, capturing the bounding boxes and their associated probabilities.

Once we completed the object detection process for the entire video, we proceeded to enhance the video's speed during playback. We achieved this by multiplying the playback speed of the recorded video, effectively making it appear as if the detection was performed in real time.

In order to mitigate the drawbacks associated with the brute force method, we decided to apply this approach in a video simulation of a car driving that we found on the internet. We made this choice primarily because the quality of the simulated video was superior compared to recording the video ourselves using a mobile phone.

Figure 2. Video frame - object detection using brute force.

Brute Force Code Overview

The code performs the brute force object detection on a video, specifically targeting traffic sign detection.

Function for detecting traffic signs – *detect_traffic_signs*: This function implements the sliding window approach for object detection. It takes an image as input and iterates over all possible window positions within the image. For each window, it extracts the region of interest, performs traffic sign classification using a pre-trained CNN model (*classify_traffic_sign* function), and if a traffic sign is detected, it appends the bounding box coordinates and predicted class to the *detected_signs* list.

CNN Model: A CNN model is defined using the CNNModel class, which inherits from nn.Module. This model consists of several convolutional, activation, pooling, and fully connected layers. It takes an image as input and outputs class probabilities for each traffic sign class.

Function for classifying traffic signs – *classify_traffic_sign*: This function takes an image as input, preprocesses it, and passes it through the CNN model. The model which has already been trained on another file as mentioned earlier is loaded and set to evaluation mode. It computes class probabilities using *softmax*, selects the class with the highest probability, and checks if the confidence surpasses a predefined threshold.

Video Processing: The code loads a video file using OpenCV's *VideoCapture* object. It then reads and processes each frame of the video in a loop. Inside the loop, the frame is converted from BGR to RGB, and object detection is performed by calling *detect_traffic_signs* on the RGB frame. Detected traffic signs are classified and labeled on the frame, and the frame is displayed using matplotlib's *imshow* function.

Figure 3. Object detection using the brute force approach

Selective Search Algorithm

Selective Search is a popular algorithm used in object detection and image segmentation tasks. It is an algorithmic approach for generating a set of potential object regions in an image. The goal of Selective Search is to propose a diverse set of regions that potentially contain objects, which can then be used as input to a classification algorithm.

The algorithm works by combining smaller image regions into larger ones based on various similarity measures, such as color similarity, texture similarity, and spatial proximity. It starts by dividing the image into smaller regions and gradually merging them based on similarity until a set of candidate regions is obtained. This approach is employed in both **image**-based tasks and **live video** implementation to generate potential object regions.

Selective Search helps reduce the computational burden of examining all possible image regions like the previously discussed brute force by generating a smaller set of candidate regions that are likely to contain objects, thus improving the efficiency of object detection algorithms.

Steps for Selective Search Object Detection in Image

The approach begins by segmenting the input image into multiple smaller regions based on some criteria. In this case, the Felzenszwalb segmentation algorithm is used, which groups pixels into regions based on their similarity in color, texture, and other low-level features. Each segmented region is treated as an initial region proposal. These proposals represent different regions in the image that may potentially contain objects.

Afterwards, similar neighboring regions are merged iteratively to create larger regions. The merging process is guided by a similarity measure, which is the Intersection over Union (IoU) between bounding boxes and is defined as a function in our code. Overlapping regions with high similarity are merged together, reducing the number of region proposals.

The merged region proposals are filtered based on their size and shape. Small or non-rectangular regions are discarded to remove unwanted proposals. An area threshold is used in order to remove these small regions as they are unlikely to contain a traffic sign. The final set of region proposals represents potential object regions in the image as shown in the figure below.

Figure 4. Final proposed regions for potential object regions.

These proposals are then passed through our pretrained CNN model to classify the objects and recognize if they contain a traffic sign based on a confidence threshold value as seen below.

Figure 5. Traffic sign classification of the region proposals.

In this example, we observe that our approach works really well as it detects and correctly classifies the stop sign. However, it also detects another misclassified region which represents a false positive. The presence of false positives indicates that the object detection system is not perfect and may have limitations in accurately distinguishing between different classes.

Selective Search Code Overview

The code performs selective search object detection on an image, specifically targeting traffic sign detection.

Function for selective search – *selective_search*: This function implements the selective search algorithm. It loads the image preprocesses it, applies Felzenszwalb segmentation to the image and iterates over the segments obtained and generates region proposals by finding rectangular bounding boxes around each segment. Also filters out non-rectangular regions and small regions based on size and merges overlapping region proposals to avoid redundancy and improve computational efficiency.

Function for merging boxes – *merge_boxes*: This function merges overlapping bounding boxes to reduce redundancy in region proposals. It takes a list of boxes and an IoU threshold as inputs. Then it compares the main box with the remaining boxes and filters out the ones that have an IoU (Intersection over Union) value below the threshold.

Function for calculating the IoU – *IoU*: This function calculates the Intersection over Union (IoU) between two bounding boxes.

After generating the region proposals, the code utilizes the pretrained CNN model for traffic sign classification and the function for classifying traffic signs – *classify_traffic_sign* as mentioned in the previous object detector in order to classify each region proposal using the trained CNN model.

YOLO

YOLO (You Only Look Once) can also be used as an object detection algorithm. It has gained significant popularity due to its speed and accuracy. It addresses the task of identifying and localizing objects within images or video frames. YOLO operates by dividing the input image into a grid and making predictions about objects present in each grid cell. For the purposes of this project, we implemented YOLOv8 which is considered the state-of-the-art version of the YOLO. YOLOv8 is utilized for traffic sign detection in both **live video** and **recorded videos**.

Steps used for YOLO

In order to enhance YOLOV8, as being an open source allowing developers to build upon it to improve it, we trained it also in a dataset with traffic signs, to improve its performance.

It is worth mentioning that for the YOLO (You Only Look Once) object detection approach, a different CNN model was utilized. The same CNN model architecture was used as the backbone, but with a distinct training strategy involving augmented data. During the training process, the input images were transformed using a series of data augmentation techniques to enhance the model's ability to generalize and detect objects under various conditions. The transformations applied to the training data included random resized cropping to extract different regions of the image, random horizontal flipping for mirror-image augmentation, random rotation to introduce variations in object orientations, color jittering to adjust brightness, contrast, and saturation levels, and random affine transformations for spatial

shifts. These augmentations enriched the training dataset with diverse examples, allowing the CNN model to learn more robust and discriminative features.

After training, changing the mode to predict, setting the input source, either for camera or from a saved video file, we could use the YOLO detector to predict the bounding box around the traffic sign.

YOLO Code Overview

After importing the necessary libraries, YOLO object is created with the path to the yolo_model.pt, which contains the pre-trained weights and configuration. A capture object is created, which can either be set to open a saved file, or to capture the input from a camera. Width and Height of the capture is set and then, a loop runs, in which:

- ✓ Next frame is read.
- ✓ Converted from BGR to RBG color format.
- ✓ Object detection, in the converted frame, is performed and the detected object, along with the bounding box coordinates, is returned.
- ✓ Labels and bounding boxes are drawn on the frame.
- ✓ Frame is displayed in an OpenCV window.

When the loop is exited, all video capture resources are released and all OpenCV windows are closed.

Results and Discussion

In our study, we trained some convolutional neural network (CNN) models for object detection in traffic signs. One model was trained with a dataset consisting of 20 traffic sign classes, while the second model utilized the whole dataset containing all 43 traffic sign classes. The purpose was to evaluate the performance and generalization ability of the models in different scenarios, while the 20 classes model was used only for the brute force method which is described below.

For the CNN model trained in 20 traffic sign classes, the results were highly promising. The model exhibited consistently high accuracy on the training set, ranging from 88.95% to 99.95%. Moreover, the validation accuracy steadily improved throughout the 10 epochs, reaching a peak accuracy of 97.46%. These results indicated that the model effectively learned to recognize and classify traffic signs, demonstrating strong generalization capabilities.

We have created the training and validation accuracy vs. epochs diagram as well as the training and validation loss vs. epochs diagram during the model training process. The diagrams below show the performance of the model over multiple epochs of training.

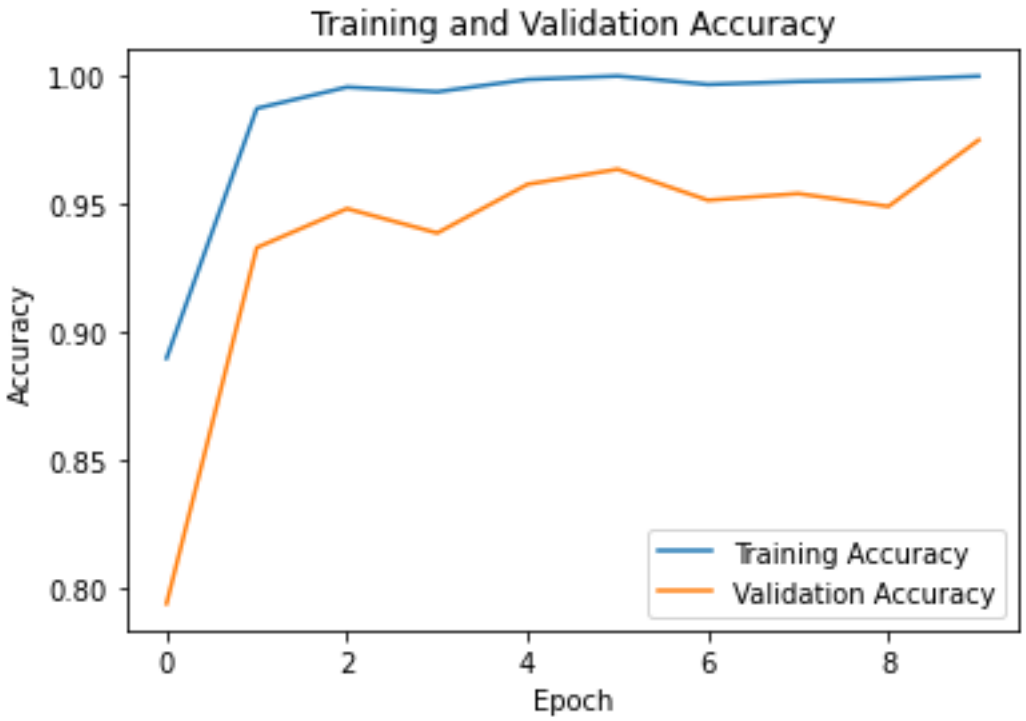

Figure 6. Training and Validation Accuracy after 10 epochs for model with 20 classes.

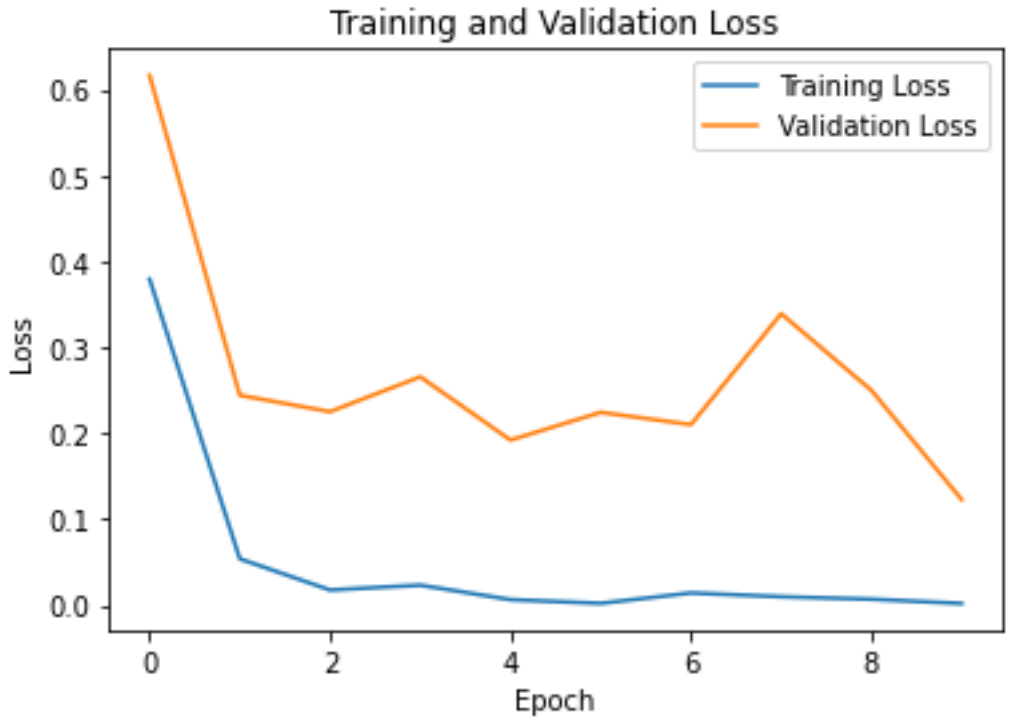

Figure 7. Training and Validation Loss after 10 epochs for model with 20 classes.

The results of the diagrams show that the model is performing effectively, as the validation loss decreases, and the model doesn't overfit the training data. This means that the model can make accurate predictions on new, unseen data, and its performance is not solely based on memorizing the training examples.

Similarly, the CNN model trained on all 43 traffic sign classes achieved excellent results. The training accuracy ranged from 97.51% to 99.72%, with the validation accuracy peaking at 95.19% in the final epoch. Both models demonstrated consistent decrease in validation loss values over the epochs and therefore these outcomes confirmed that the model does not overfit and generalizes well.

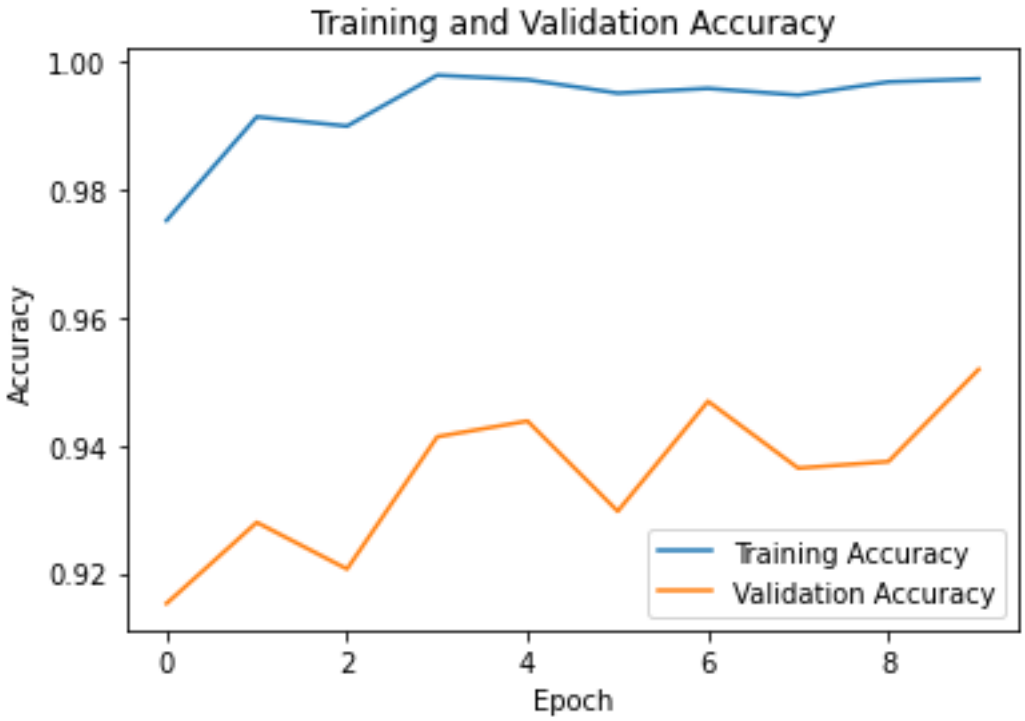

Figure 8. Training and Validation Accuracy after 10 epochs for model with 43 classes.

Figure 9. Training and Validation Loss after 10 epochs for model with 43 classes.

Again, regarding the training and validation loss vs. epochs diagram, we notice that the training loss decreases consistently with each epoch, indicating that the model is learning and fitting well to the training data. Importantly, the validation loss also decreases and follows a similar trend as the training loss. This behavior suggests that this model also generalizes well to unseen data, and there is no significant overfitting.

Regarding the CNN model that was trained on the augmented data model, it exhibits a slightly lower validation accuracy of 76.53% using 30 epochs compared to the other model's 95.19%. However, its performance is well-suited for video processing due to the challenges posed by analyzing dynamic and diverse video content. In video analysis, objects can appear from various angles and undergo transformations over consecutive frames. The YOLO approach, combined with the augmented data model, better handles these diverse conditions, enabling more accurate object detection and localization in real-time.