International Institute of Information Technology Bangalore

Machine Learning

AI-825

Mini Project

Team Members:

Achintya Harsha **(IMT2021525)**

Adithya Nagaraja Somasalle **(IMT2021054)**

Akash Perla **(IMT2021530)**

March 22, 2024

Contents

1 Play with CNN:

1.1 Theory

- CIFAR-10 Dataset: It comprises of 60,000 32x32 color images, equally distributed across 10 distinct classes, each containing 6,000 images. These classes include airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. For classification purposes, the 60,000 images are split into training and testing sets, with 50,000 images allocated for training and 10,000 images reserved for testing.
- CNN: CNNs consist of layers of artificial neurons. Each layer's output becomes the input for the next layer. The initial layers identify simple patterns like edges, while deeper layers recognize more complex features like objects or faces. These layers can include convolutional, pooling, and fully connected layers.

1.2 Implementation

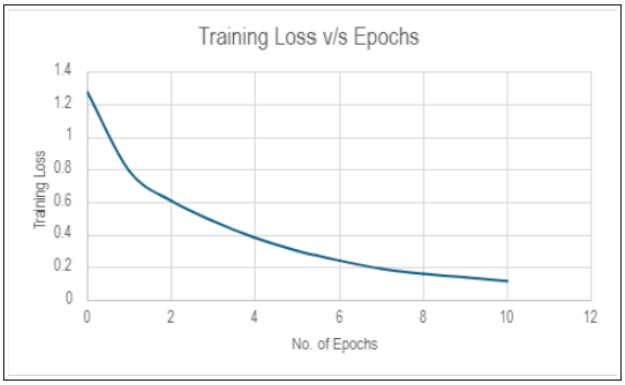
Model:

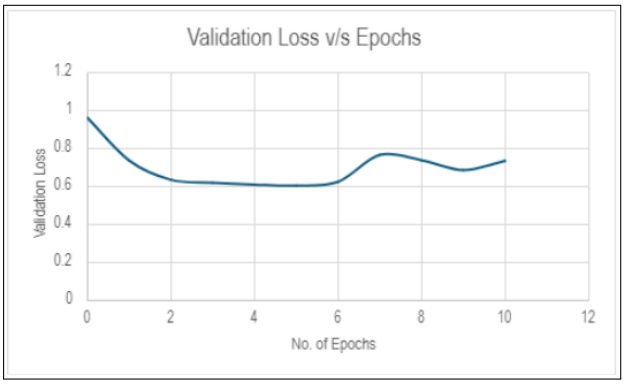
- Convolutional Blocks: These are like building blocks for the model. Each block does some processing to the input data. The first and third blocks have a series of steps: convolutional layer, batch normalization, ReLU activation, another convolutional layer, another ReLU activation, and then max pooling. The second block is similar but includes an additional dropout step.
- Fully Connected Layers: These layers come after the convolutional blocks. They further process the information extracted by the convolutional layers. We have 3 fully connected layers, each followed by ReLU activation and dropout.

1.3 Observations

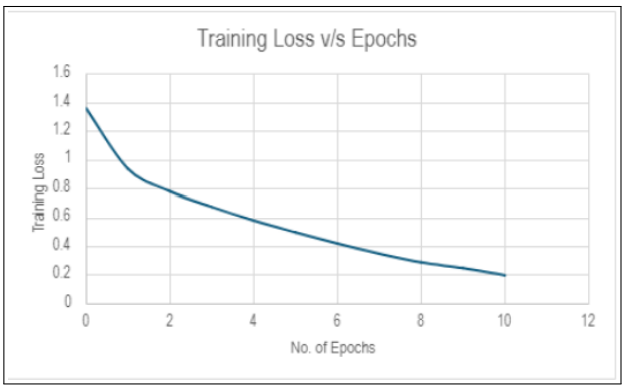
- SGD with momentum

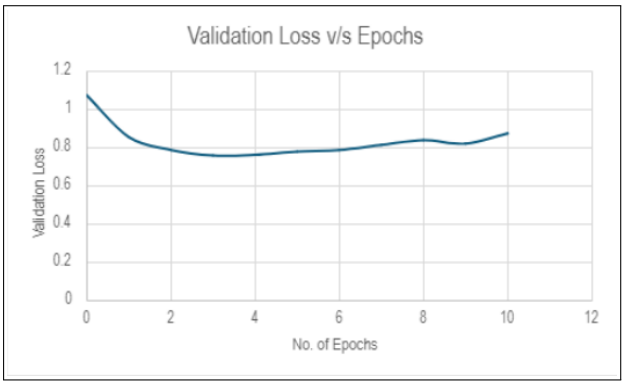
1. Activation-Relu



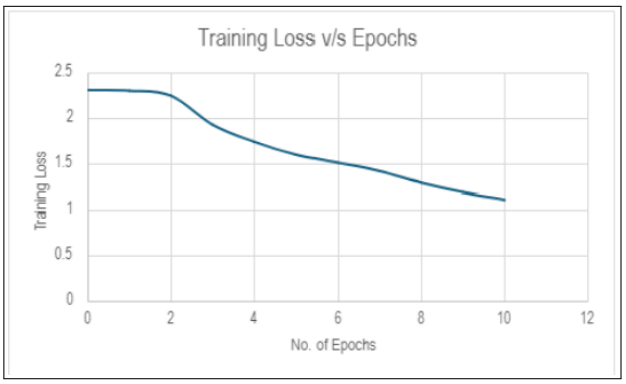


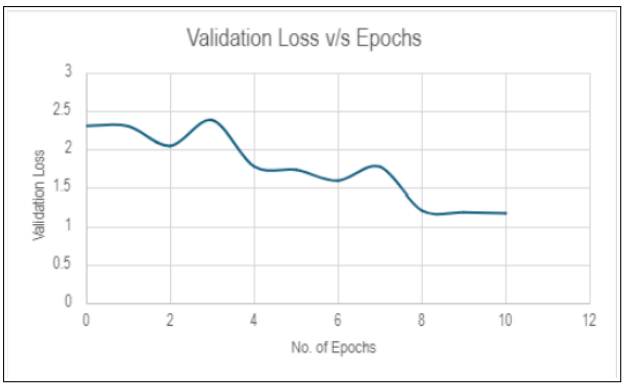
2. Activation-Tanh





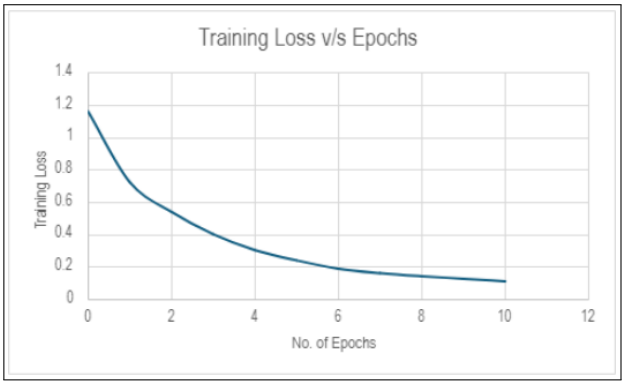
3. Activation-Sigmoid

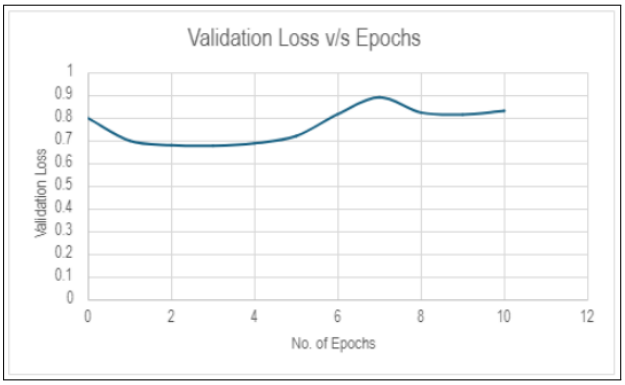




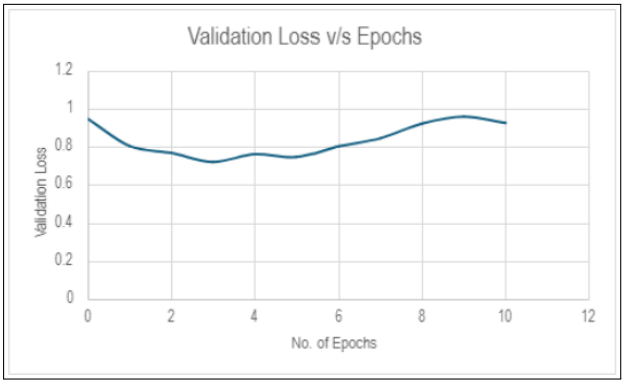
- Adam

1. Activation-Relu

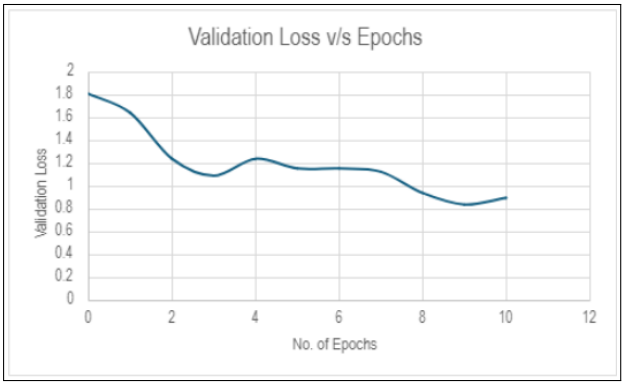




2. Activation-Tanh

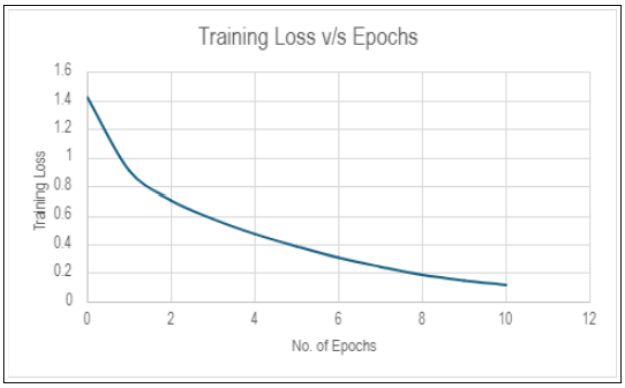


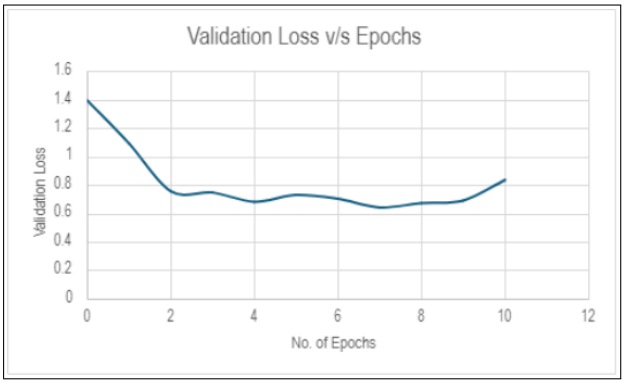
3. Activation-Sigmoid



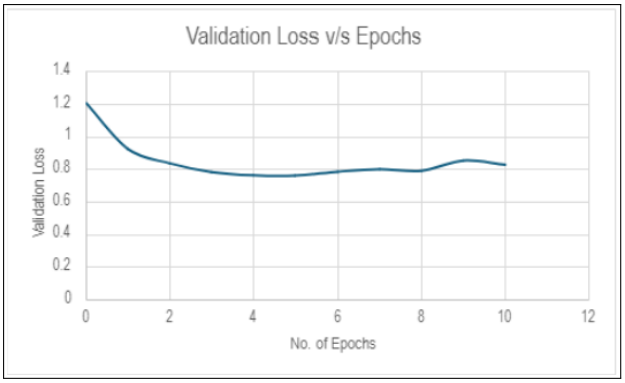
- SGD without momentum

1. Activation-Relu

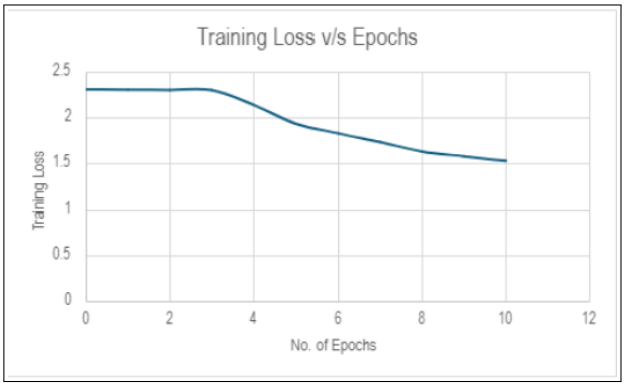


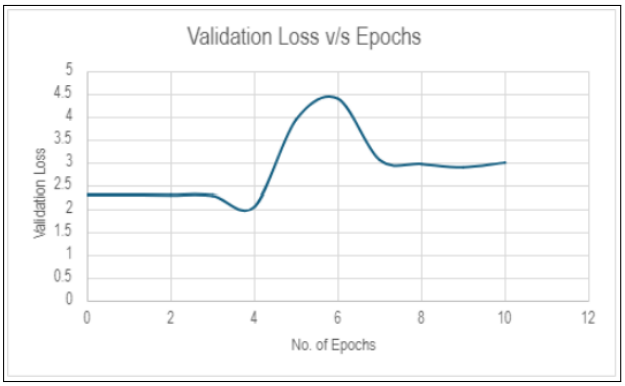


2. Activation-Tanh



3. Activation-Sigmoid





1.4 Results

- SGD with momentum:

Activation function	Accuracy	Training Time
ReLU	82.47%	4min 22s
Tanh	76.70%	4min 15s
Sigmoid	57.44%	4min 3s

- SGD without momentum:

Activation function	Accuracy	Training Time
ReLU	78.88%	6min 50s
Tanh	75.89%	6min 38s
Sigmoid	19.53%	6min 25s

- Adam:

Activation function	Accuracy	Training Time
ReLU	80.47%	5min 41s
Tanh	78.09%	5min 23s
Sigmoid	69.48%	5min 18s

Based on the results obtained, we observe that ReLU performs the best for our CNN followed by Tanh and Sigmoid. ReLU is the best choice of activation function since it gives better accuracy. SGD with momentum (momentum=0.45, lr=0.1) is the best optimiser.

Recommended Architecture: 3 convolutional layer blocks(convolutional layer block mentioned above), 3 fully connected layers and ReLU activation with batch-normalisation, maxpooling and Dropout, i.e. the architecture attached below)

```
Net(
  (conv_layer): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): ReLU(inplace=True)
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (12): Dropout2d(p=0.05, inplace=False)
    (13): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc_layer): Sequential(
    (0): Dropout(p=0.1, inplace=False)
    (1): Linear(in_features=4096, out_features=1024, bias=True)
    (2): ReLU(inplace=True)
    (3): Linear(in_features=1024, out_features=512, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.1, inplace=False)
    (6): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

2 CNN as a feature Extractor:

2.1 Dataset

We have choosen the Multi-Weather classification dataset (link).

It is a collection of images of various weather conditions which can be used for image classification tasks. It contains 4 different categories of weather: Sunny, Cloudy, Rainy and Snowy and it contains 1125 images We also removed rows with null values in columns where the null value percentage was less than 5%. These rows were considered inconsequential as the columns contained sufficient data for training our model.

2.2 Architecture

We have used pre-trained AlexNet in pytorch. AlexNet works great with images of specific mean and standard deviation, so we preprocessed the data by resizing the image into 256x256 and normalized with mean of [[0.485,0.456,0.406] and std of [0.229,0.224,0.225].

Alexnet has an architecture in which the final fully connected layer gives an output of size 1000 vector. So, we took all the images of the dataset seperated as train and test images and ran Alexnet on them to get 1000 size feature vectors. Next, we passed these images through different models such as Logistic Regression(LR), K-Nearest Neighbors(KNN), SVM(Support Vector Machine) and Gaussian Naive Baye(GNB), we will see which models give better score.

We removed the last softmax layer and last linear layer and passed the data to the resultant network which gave us input features of 9216 size and these were passed into various models and trained such as Logistic Regression(LR), K-Nearest Neighbors(KNN), SVM(Support Vector Machine) and Gaussian Naive Baye(GNB).

Following results were obtained for Multi-weather Classification Dataset:

Model	LR	KNN	SVM	GNB
Accuracy	96%	84.44%	96.88%	67.55

Following resuts were obtained for Bike-Vs-Horse Classification dataset:

Model	LR	KNN	SVM	GNB
Accuracy	100%	94.44%	97.22%	66.66

Neural Network on Caltech-101 Dataset : We also ran on Caltech-101 Dataset. Caltech-101 is a well-known object identification dataset that includes photos of things from 101 distinct categories, with an average of 40–800 images per category.

We have split dataset into train, test and validation datasets.

Pre-processing:

1. Resizing of image to 224×224 to fit the AlexNet specifications.
2. Normalizing RGB channels with mean $[0.485, 0.456, 0.406]$ and standard deviation $[0.229, 0.224, 0.225]$.

Data Augmentation:

1. Random rotations of images by 30° angle.

2. Random horizontal and vertical flips.

Data augmentation on train dataset are done to increase the data count for classes having a low amount of images.
We changed the output of AlexNet to 101 classes instead of 1000 classes as out dataset contains 101 classes i.e we built a neural network on top of AlexNet. In this we achieved 77% accuracy on test data.

We then changed the output of AlexNet to 2 classes for Bike vs Horse Dataset and we were able to achieve 100% accuracy on test data for this.
We then changed the output of AlexNet to 4 classes for Multi-class wather classification Dataset and we were able to achieve 90.667% accuracy on test data for this.
We were able to achieve the AlexNet with Neural Network on top by following line:

```
model.classifier[6] = nn.Linear(in_features=4096,out_features=101,bias=True)
```

Final Network

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=101, bias=True)
  )
)
```

3 YOLO V2

YOLOv2, an advancement over YOLOv1, introduces several additional features enhancing object recognition in computer vision applications:

1. Multi-Scale Detection: YOLOv2 conducts detection at multiple scales within the input image, improving accuracy by accommodating objects of varying sizes.

2. Anchor Boxes: YOLOv2 integrates anchor boxes, varying in size and aspect ratio, to improve object localization predictions.

3. Batch Normalization: YOLOv2 incorporates batch normalization to stabilize training and improve overall network accuracy.

4. High-Resolution Classifier: YOLOv2 employs a high-resolution classifier for more precise image representation, thereby enhancing detection accuracy.

5. Darknet-19 Architecture: YOLOv2 adopts the deeper neural network architecture, Darknet-19, comprising 19 layers, which enhances the network's ability to extract features from input images, consequently improving object detection performance.

4 Object Tracking

4.1 Theory

4.1.1 Faster RCNN

Faster R-CNN is an object detection framework comprising two main components: the Region Proposal Network (RPN) and the Region-based Convolutional Neural Network (RCNN). The RPN generates potential object regions by sliding a network over the input image's convolutional feature map, utilizing anchor boxes for different scales and aspect ratios. Non-maximum suppression (NMS) is then employed to refine these proposals.
In parallel, the RCNN extracts fixed-size feature vectors from the proposed regions and performs object classification and bounding box regression, leading to accurate localization and identification of objects. By sharing convolutional layers between the RPN and RCNN, Faster R-CNN facilitates end-to-end training, enhancing both detection accuracy and computational efficiency, making it a cornerstone in object detection research.

4.1.2 YOLO

YOLO (You Only Look Once) is a real-time object detection algorithm introduced in 2015. It employs a single-stage approach, utilizing a convolutional neural network (CNN) to predict bounding boxes and class probabilities of objects in images. By dividing the input image into a grid of cells, YOLO efficiently processes the entire image in one pass, unlike two-stage detectors like R-CNN, resulting in faster and more effective object detection. Initially implemented using the Darknet framework, YOLO revolutionized object detection with its speed and accuracy.

4.1.3 SORT

The SORT(Simple Online Realtime Tracking) algorithm is a real-time effective tracking algorithm. It combines the concepts of Kalman filters and Hungarian algorithms to track objects.
The following steps are involved in the algorithm of SORT:

1. Identify objects in the first frame using a detection algorithm, assigning each a unique ID.Predict object positions in subsequent frames using a constant velocity model and Kalman filter.

2. Match detections with existing tracks based on Intersection over Union (IOU) scores; reject low-scoring associations. Update tracked object states with new detections, adjusting position, velocity, and attributes.

3. Employ efficient algorithms like the Hungarian algorithm for real-time processing. Tolerate occlusions and clutter, maintaining tracks even without detections briefly.

4. Integrate with deep learning detectors for improved accuracy. Fine-tune parameters like IOU threshold for optimal performance.

4.1.4 DeepSORT

DeepSORT is a Computer Vision Tracking Algorithm used to track the objects while assigning each of the tracked object a unique id. DeepSORT is an extension of the SORT. DeepSORT introduces deep learning into SORT algorithm by adding appearance descriptor to reduce the identity switches and hence making the tracking more efficient.

4.2 Implementation

4.2.1 Procedure

1. The video is split into frames using OpenCv. YOLOv5 or Faster RCNN is used to detect the object of the "car" class in the particular frame.
2. SORT or DeepSORT is used to label each detected object uniquely.
3. The total count of cars is updated and the bounding box is drawn around detected objects.
4. The resultant frames are merged into a single final output video.

4.3 Results

All the code and results for this section can be found in this link (link). GitHub Link : (GitHub).

4.3.1 Faster RCNN + SORT

Figure 1: Video 1 RCNN with SORT algorithm

Figure 2: Video 2 RCNN with SORT algorithm

4.3.2 Faster RCNN + DeepSORT

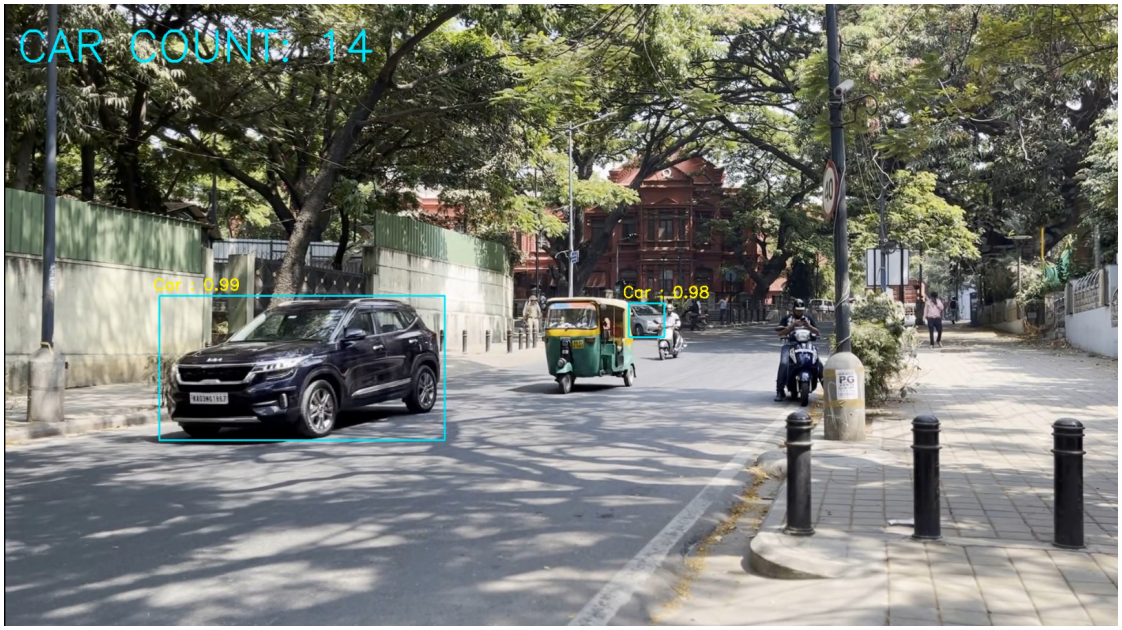


Figure 3: Video 1 RCNN with DeepSORT algorithm

Figure 4: Video 2 RCNN with DeepSORT algorithm

4.3.3 YOLOv5 + SORT

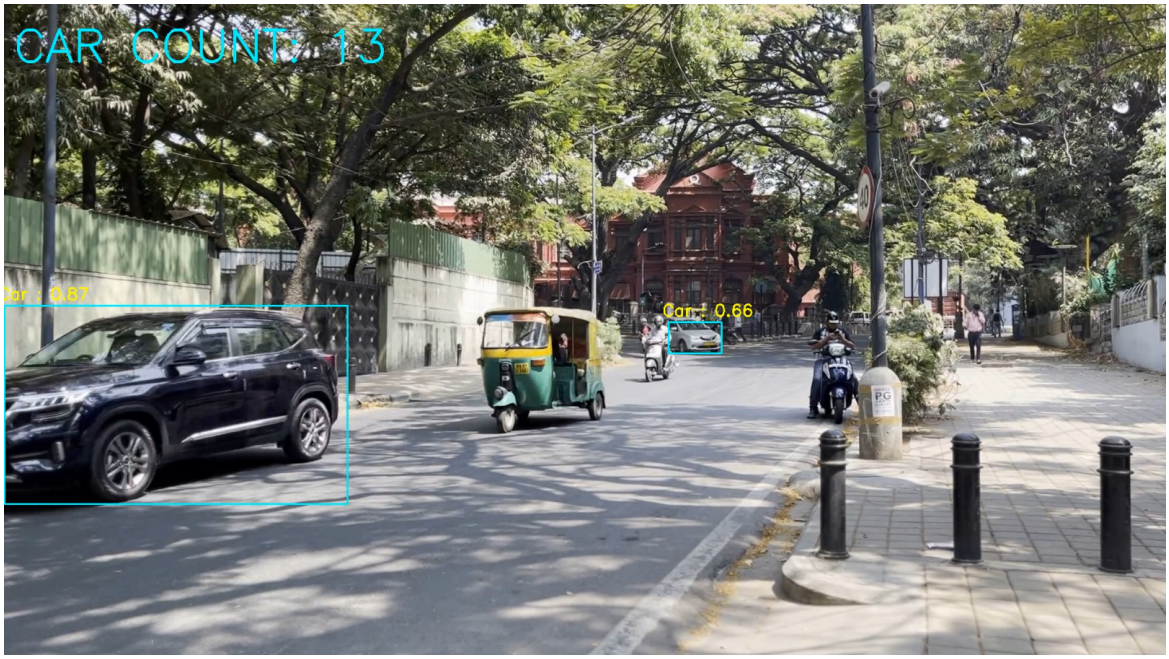


Figure 5: Video 1 RCNN with SORT algorithm

Figure 6: Video 2 YOLO v5 with SORT algorithm

4.3.4 YOLOv5 + DeepSORT

Figure 7: Video 1 YOLO v5 with DeepSORT algorithm

Figure 8: Video 2 YOLO v5 with DeepSORT algorithm

4.3.5 Observations

Ground Truth Count: 17	FasterRCNN	YOLO v5
SORT	19	15
DeepSORT	17	17

Table 1: Count of cars obtained by using various models and tracking algorithms for video 1

Ground Truth Count: 33	FasterRCNN	YOLO v5
SORT	55	41
DeepSORT	31	25

Table 2: Count of cars obtained by using various models and tracking algorithms for video 2

From the video outputs we can observe the folowing:

- YOLOv5 paired with any tracker generally exhibits quicker detection and tracking performance compared to Faster R-CNN paired with any tracker.
- We can observe from video 1 and 2 that since the video 2 is more chaotic and unorganized SORT yields a very bad result as the same car is counted multiple times. But DeepSort significantly improves the count as DeepSort makes sure that each car is uniquely assigned an id and is counted only once.
- It might be beneficial to improve the counting mechanism by introducing a reference line to increment the counter gradually rather than instantly adding to it upon detecting an automobile. This approach could serve as a means of enhancing traffic management.

4.4 Conclusion

4.4.1 FasterRCNN vs YOLO

When comparing the speed and performance of YOLO v5 and Faster RCNN, we discovered that YOLO v5 delivers a higher frame-per-second output, making it suitable for real-time applications such as video stream analysis from camera feeds. However, our evaluation revealed that Faster RCNN outperforms YOLO v5 in detecting various on-screen objects, especially distant automobiles. Despite YOLO v5's faster processing speed, Faster RCNN's superior performance in detecting objects, particularly those at a distance, makes it a more suitable choice for certain applications.

YOLO v5's speed advantage comes from its single-pass architecture, directly predicting bounding boxes and class probabilities for each grid cell. This design facilitates real-time processing by eliminating the need for multiple stages. In contrast, Faster RCNN's two-stage approach involves region proposal and refinement, resulting in more accurate detections, especially for smaller or distant objects, but at the cost of speed. Therefore, the choice between YOLO v5 and Faster RCNN hinges on the trade-off between speed and accuracy, depending on the specific requirements of the application.

4.4.2 SORT vs DeepSORT

SORT (Simple Online and Realtime Tracking) and deepSORT (Deep SORT) are both algorithms utilized for multi-object tracking, but they differ in several key aspects:

- Methodology: SORT employs conventional techniques like the Hungarian algorithm and Kalman filter, while deepSORT integrates deep learning, combining deep appearance descriptors with these traditional methods.
- Feature Representation: SORT uses handcrafted attributes like location and size, while deepSORT employs deep neural networks to extract object appearance features, making it more adaptable to changes in environment.
- Identity Management: deepSORT excels in maintaining object identity over time, even in challenging scenarios like occlusions or similar-looking objects, which can be problematic for SORT.
- Performance: deepSORT typically outperforms SORT in scenarios with appearance fluctuations or occlusions, but SORT is better suited for real-time applications due to its simpler implementation.
- Real-time Considerations: SORT's avoidance of neural network forward passes makes it more suitable for real-time processing on devices with limited GPU capabilities, whereas deepSORT's reliance on neural networks can lead to increased overhead.

In summary, SORT is preferable for real-time tracking on commodity hardware where accuracy may be sacrificed due to online data constraints, while deepSORT is better suited for accuracy-sensitive applications where real-time performance is less critical.