VR MINIPROJECT REPORT

Kevin Adesara (IMT2021070)

Anant Ojha (IMT2021102)

Varad Bharadiya (IMT2021532)

Note: For section D, all the input and output videos are attached in this link: https://drive.google.com/drive/folders/1kv_UIgEs-T8Ad99al6RRpvnRMPxOo-1u?usp=sharing

Section-A

Image Classification on CIFAR-10 dataset using Convolutional Neural Network (CNN)

Introduction to CNN & CIFAR

CNN model was learnt on CIFAR-10 dataset for image classification using pytorch library of python. Activation functions such as ReLU, tanh and sigmoid were tried with various combinations of convolutional layers and fully connected layers. In the pytorch documentations a good example of training a classifier has been shown.

About CIFAR10 Dataset

The CIFAR-10 dataset consists of 60000 32X32 color images. These 6000 images are from 10 classes with 6000 images in each class. The 10 classes in the dataset are airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. For our classification purpose, these 60000 images are divided into train and test set with 50000 images in the train set and 10000 images in the test set.

About Convolutional Neural Network (CNN)

CNN is a deep learning algorithm that is most applied to analyzing visual imagery. While in traditional methods the image features are hard engineered, CNN provides us the liberty to give input as the image as it is and learn the

features/characteristics of the image on its own. CNN is composed of multiple layers of artificial neurons. The output of one layer of CNN is fed as the input to the next one. The first layer of CNN usually detects basic features such as horizontal or vertical edges. As you move deeper the layer starts detecting the higher-level features such as objects, faces. The layers in CNN can be convolutional layers, pooling layers and fully connected layers.

Approach Explained in Detail:

The model is implemented using pytorch library of python.

Input Layer: The input images are expected to have three channels (RGB) since the in_channels parameter of the CNN constructor is set to 3.

Convolutional Blocks:

Convolutional Layers: Each convolutional block consists of two convolutional layers followed by batch normalization and ReLU activation function.

The convolutional layers use 3x3 kernels with padding to maintain the spatial dimensions of the feature maps.

Batch normalization is applied to normalize the activations of each layer, helping in faster convergence during training.

ReLU activation function introduces non-linearity to the model, allowing it to learn complex patterns in the data.

Pooling Layer: After each pair of convolutional layers, max pooling with a 2x2 kernel and stride of 2 is applied to reduce the spatial dimensions of the feature maps by half, thus down sampling the features.

Fully Connected (FC) Block:

After passing through the convolutional layers, the feature maps are flattened using flatten operation.

Then, the flattened features are passed through a series of fully connected layers with ReLU activation functions and dropout layers in between.

The first fully connected layer reduces the dimensionality to 512, followed by a ReLU activation and a dropout layer with a dropout rate of 0.5.

Another fully connected layer reduces the dimensionality further to 256, followed by ReLU activation and dropout.

Finally, the output layer produces predictions for the classes using a fully connected layer without an activation function applied explicitly. The final activation is applied later during inference using Log SoftMax.

Output Layer: The output layer produces logits for each class. These logits are converted to probabilities using the Log SoftMax activation function in the forward method, after which predictions can be made by selecting the class with the highest probability.

Overall, this CNN architecture consists of multiple convolutional blocks for feature extraction followed by a fully connected block for classification. It is designed to learn hierarchical representations of features in the input images and make predictions based on those features.

COMBINATIONS TRIED:

We initially started with 4 CNN layers and 2 FC layers and using stochastic gradient descent (SGD) with momentum. The following were the observations made for Tanh with SGD and momentum. (Momentum =0.9)

Epoch 1/10, Loss: 1.7178710360661187

Epoch 2/10, Loss: 1.3661400325158064

Epoch 3/10, Loss: 1.1718245865134023

Epoch 4/10, Loss: 1.0442841635335742

Epoch 5/10, Loss: 0.9434976887215129

Epoch 6/10, Loss: 0.834720287367206

Epoch 7/10, Loss: 0.7555759236254656

Epoch 8/10, Loss: 0.684088701177436

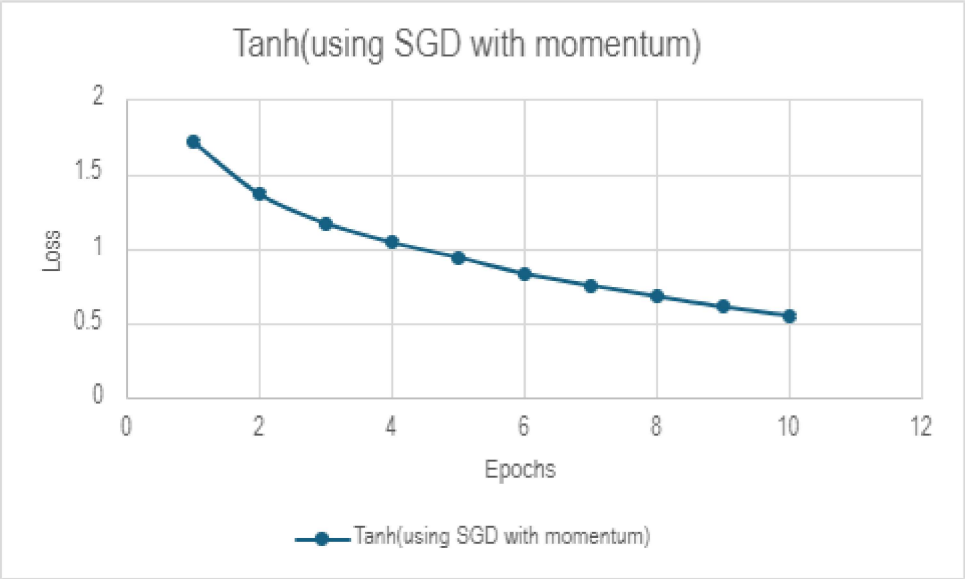
Epoch 9/10, Loss: 0.6128920804128013

Epoch 10/10, Loss: 0.5536008727215135

Accuracy on test set: 77.72%

This activation has the formula – g(x) = 2/(1+ e^(-2x)) – 1

The time taken was 5 hours



For Sigmoid with SGD and momentum:

Epoch 1/10, Loss: 1.5861312919260595

Epoch 2/10, Loss: 1.1503502731128117

Epoch 3/10, Loss: 0.9183450589323288

Epoch 4/10, Loss: 0.7766158564392563

Epoch 5/10, Loss: 0.6582114441163095

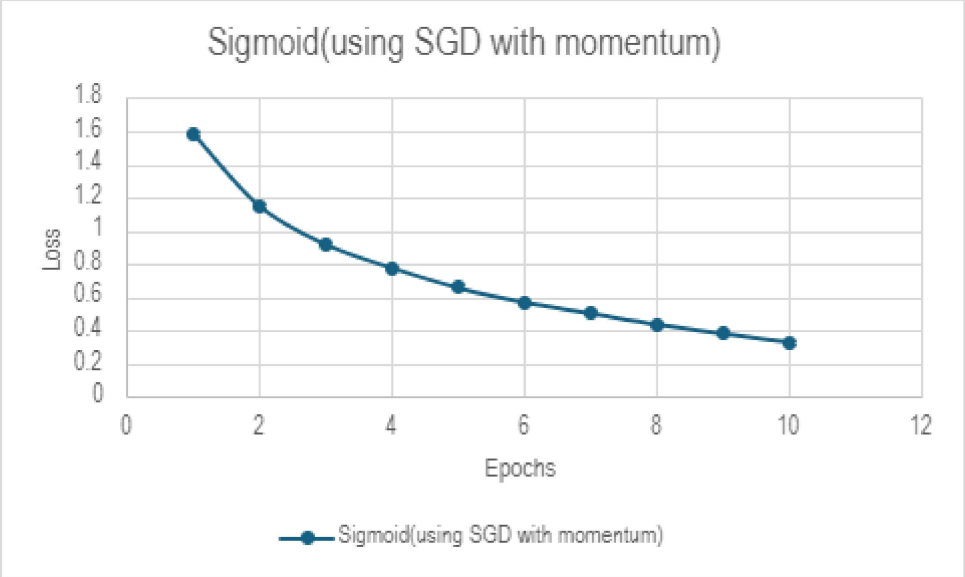
Epoch 6/10, Loss: 0.5690752352061479

Epoch 7/10, Loss: 0.5033227935471498

Epoch 8/10, Loss: 0.43710428074269037

Epoch 9/10, Loss: 0.3813700695019549
Epoch 10/10, Loss: 0.3301420951133494
Accuracy on test set: 76.56%
The time required for training was 5 hours.

This activation has the formula – g(x) = 1/(1 + e^(-x))



For Relu with SGD and momentum:
Epoch 1/10, Loss: 1.6338122188282744

Epoch 2/10, Loss: 1.1588779518671353

Epoch 3/10, Loss: 0.910141977812628

Epoch 4/10, Loss: 0.7591500263613509

Epoch 5/10, Loss: 0.6384064620908569

Epoch 6/10, Loss: 0.5442136704845502

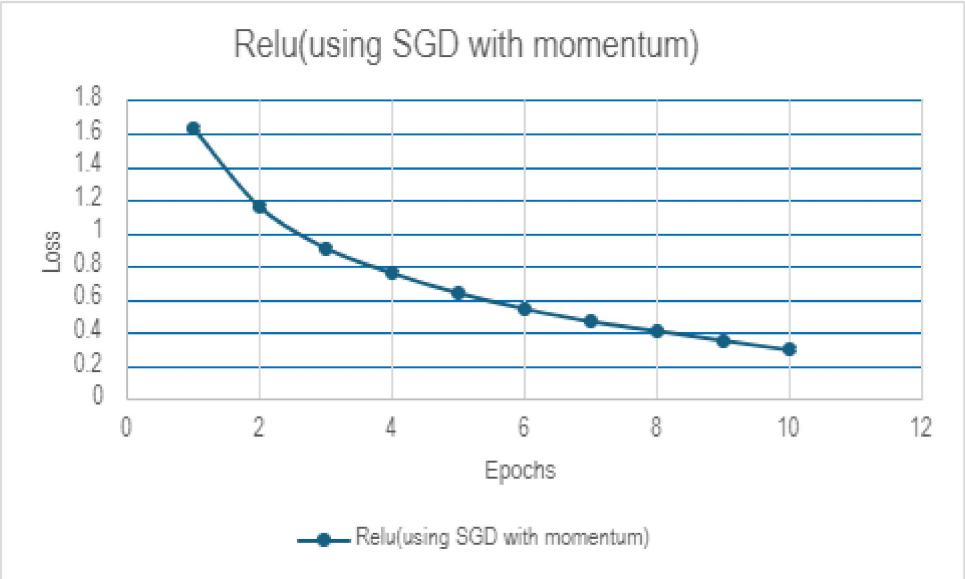
Epoch 7/10, Loss: 0.4667827045292501

Epoch 8/10, Loss: 0.4111713199588039

Epoch 9/10, Loss: 0.35128747323132536

Epoch 10/10, Loss: 0.2963946273888621

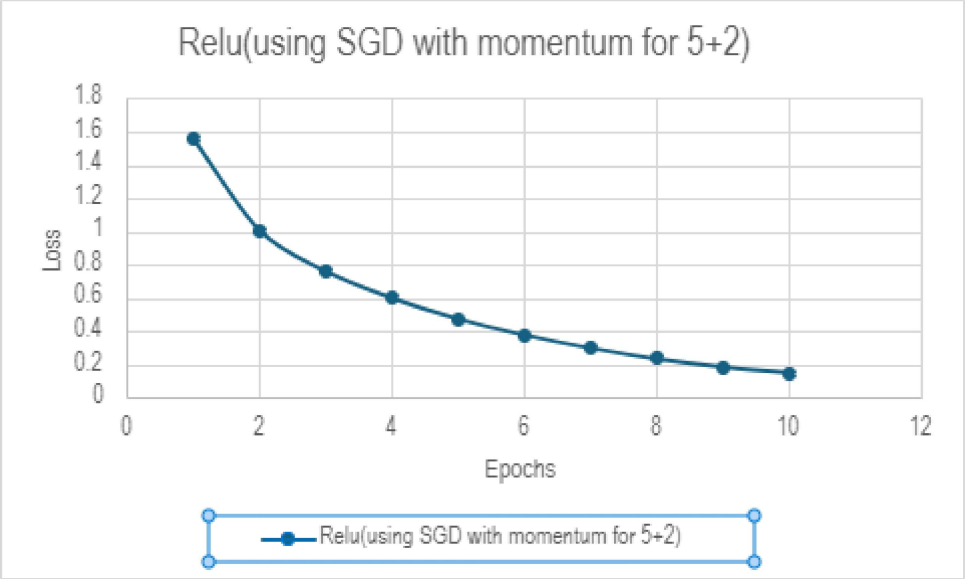
Accuracy on test set: 85.5%
The time required for training in this case was around 4 hours.



Now, we went ahead with increasing complexity of our model with 5 CNN layers and 2 fully connected layers. We did not use tanh and sigmoid as they were eliminated by their performance when compared with ReLU.

Epoch 1/10, Loss: 1.5588648686628512
Epoch 2/10, Loss: 1.0070512165575076
Epoch 3/10, Loss: 0.761334937132533
Epoch 4/10, Loss: 0.6003601265228008
Epoch 5/10, Loss: 0.474278630960323
Epoch 6/10, Loss: 0.3781060623028851
Epoch 7/10, Loss: 0.299476518845924
Epoch 8/10, Loss: 0.23537105634389327
Epoch 9/10, Loss: 0.18608344889119688
Epoch 10/10, Loss: 0.14970063536292147
Accuracy on test set: 78.13%

The time taken was around 6 hours. This can be because the complexity of the model was increased.



Without momentum:
For Tanh

Epoch 1/10, Loss: 2.0486078402575325

Epoch 2/10, Loss: 1.7806983213595418

Epoch 3/10, Loss: 1.586581483521425

Epoch 4/10, Loss: 1.4680592153993104

Epoch 5/10, Loss: 1.3802319844360547

Epoch 6/10, Loss: 1.2950305208525694

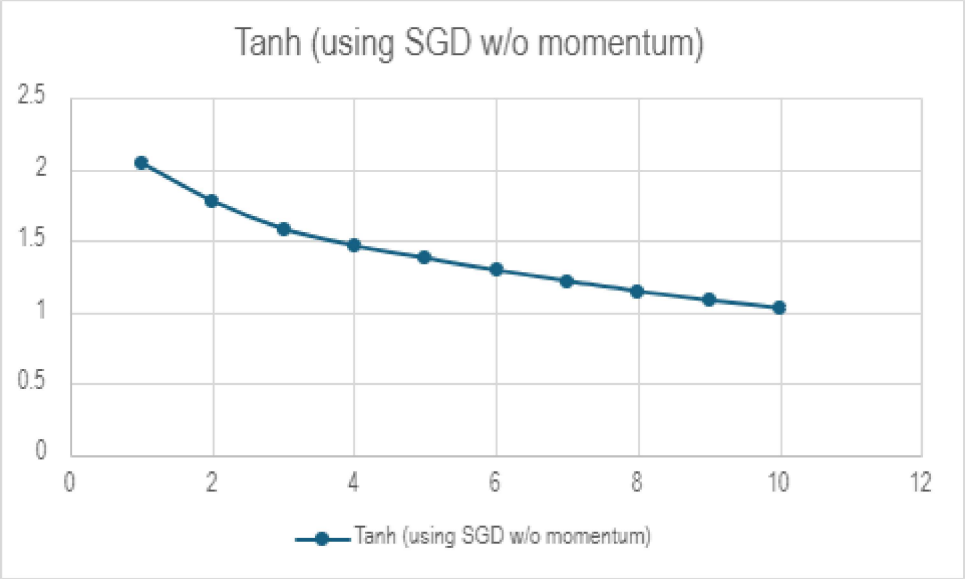
Epoch 7/10, Loss: 1.220568087094885

Epoch 8/10, Loss: 1.1467510557845426

Epoch 9/10, Loss: 1.0860859493312933

Epoch 10/10, Loss: 1.0297946633432833

Accuracy on test set: 57.37%

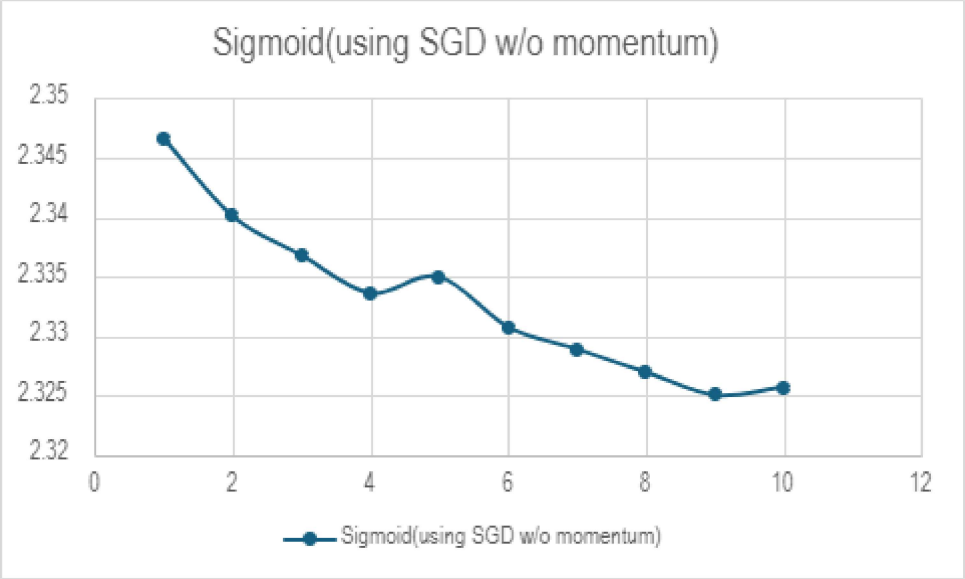


For Sigmoid:

Sigmoid, moment = 0

Epoch 1/10, Loss: 2.346648204357118
Epoch 2/10, Loss: 2.340106597032084
Epoch 3/10, Loss: 2.33679483552723
Epoch 4/10, Loss: 2.3336031391187704
Epoch 5/10, Loss: 2.3349944121392485
Epoch 6/10, Loss: 2.330765844610951
Epoch 7/10, Loss: 2.328891118774024
Epoch 8/10, Loss: 2.3269583279519437
Epoch 9/10, Loss: 2.325028521020699
Epoch 10/10, Loss: 2.325691189607391
Accuracy on test set: 10.0%

In the case of sigmoid we can see that the accuracy has significantly dropped.



Batch Normalization

Batch Normalization is an extra layer in neural networks that perform the function of standardization. When there are many parameters and filters, the gradients are noisy and hence, this normalizes the inputs. There are two parameters called gamma and beta that this learns if affine parameters is set to True in Py torch library. Otherwise, it works with default values. But for me learnable parameters gave good accuracies.

Observations made:

ReLU Outperforming tanh and sigmoid:

ReLU (Rectified Linear Unit) tends to perform better than tanh and sigmoid activations due to its simplicity and effectiveness in combating the vanishing gradient problem. ReLU replaces negative values with zero, which helps in avoiding saturation during training and accelerates convergence. Sigmoid and tanh activations suffer from the vanishing gradient problem, particularly in deep networks, which can slow down training or lead to saturation of gradients.

Stochastic Gradient Descent (SGD) with Momentum:

SGD with momentum incorporates a momentum term that helps accelerate SGD in the relevant direction and dampens oscillations. This often leads to faster convergence and better generalization compared to traditional SGD without momentum, which tends to oscillate and take longer to converge, especially in regions with high curvature.

Batch Normalization:

Batch normalization normalizes the input of each layer to have zero mean and unit variance, which helps stabilize and speed up the training process. It reduces the internal covariate shift, making the network more robust and easier to train. While the improvement in categorization accuracy might be marginal, batch normalization can have a significant impact on the convergence speed and stability of training, especially in deeper networks.

Increased Training Time with More Layers or Parameters:

As the number of layers or parameters in a neural network increase, the computational complexity and memory requirements also increase. This leads to longer training times due to the increased number of calculations required for forward and backward passes through the network. Additionally, deeper networks may suffer from vanishing or exploding gradients, which can further slowdown training. Furthermore, the increased number of parameters means more data needs to be processed during each iteration, resulting in longer training times.

In summary, these observations highlight the importance of understanding different aspects of neural network architectures, activation functions, optimization algorithms, and regularization techniques in achieving better performance and efficiency in deep learning tasks.

Section B

Using CNN as a Feature Extractor (Fruits & Vegetables Dataset)

Dataset Overview:

We worked with a dataset known as the Fruits & Vegetables dataset, which consisted of images categorized into 36 classes of fruits and vegetables. Each image in the dataset was of size 224x224 pixels. The dataset was divided into separate sets for training and testing.

Approach:

1. Preprocessing:

- As a preprocessing step, we utilized the `os.listdir` function from the `os` module to extract the dataset.

- Since we intended to use a pretrained AlexNet model from PyTorch, we performed preprocessing to bring the images to the required format. AlexNet is sensitive to specific mean and standard deviation values, so we normalized the images accordingly.

2. Feature Extraction:

- We leveraged a pretrained AlexNet model to extract features from the images. AlexNet's architecture produces a 1000-dimensional feature vector as output.

- These feature vectors were then passed through a logistic regression classifier to make predictions.

- We experimented with different models to find the one that yielded the best performance.

3. Data Preparation:

- To prepare the data for GPU processing and compatibility with the AlexNet model, we created Dataset objects from PyTorch.

- We utilized DataLoader objects from PyTorch to organize the data into batches, using a batch size of 16 and a collate function to generate batches.

4. Model Fine-Tuning:

- We loaded a pretrained AlexNet model from PyTorch with `pretrained=True`.

- To fine-tune the model for our dataset, we added two additional layers: a linear layer with an output size of 36 (corresponding to the 36 categories of fruits and vegetables) and a softmax layer.

- We froze the early layers of the model (using `require_grad=False`) to avoid full training and applied transfer learning.

- The model was trained on the train set and evaluated on the test set.

5. Model Evaluation:

- After training, we evaluated the model using different traditional machine learning models such as SVM, Logistic Regression, KNN, and XGBoost.

- We achieved varying levels of accuracy with each model:

  - SVM with linear kernel: 31.04%

  - Logistic Regression: 20.09%

  - KNN: 15.19%

  - XGBoost: 67.9%

Insights:

- The low accuracies obtained could be attributed to the large number of categories and relatively few images per category in the dataset (approximately 100 images per category).

- Transfer learning techniques and CNNs as feature extractors were explored, highlighting their potential applications in similar scenarios.

Bike and Horse Dataset:

- In the same notebook, we applied the same approach to a different dataset containing images of bikes and horses.

- Due to the smaller size of this dataset, the model achieved a 100% accuracy rate, indicating that the model was able to learn effectively with fewer images.

Overall, this project provided valuable insights into fine-tuning models for specific datasets, utilizing transfer learning, and leveraging CNNs as feature extractors.

Section-C

Based On the given paper, there are 5 points/areas where YoloV2 is be found to be performing better:

Improved Localization and Recall: YOLOv2 addresses the issue of localization errors and low recall compared to previous methods like Fast R-CNN. By incorporating techniques like anchor boxes and direct location prediction, YOLOv2 achieves better localization accuracy and recall.

Batch Normalization: YOLOv2 includes batch normalization on all convolutional layers, leading to improved convergence and model

regularization. This results in more than a 2% improvement in mean Average Precision (mAP) without the need for additional regularization techniques like dropout.

High-Resolution Classifier: YOLOv2 fine-tunes the classification network at a higher resolution (448x448) for better adaptation to object detection tasks. This approach results in an increase of almost 4% in mAP, indicating improved classification accuracy.

Convolutional With Anchor Boxes: YOLOv2 replaces the fully connected layers with convolutional layers and incorporates anchor boxes for bounding box prediction. This simplifies the problem and makes it easier for the network to learn, resulting in a higher recall rate despite a slight decrease in mAP.

Dimension Clusters: YOLOv2 utilizes k-means clustering on training set bounding boxes to automatically generate anchor box priors, rather than hand-picking them. This approach leads to better representation and easier learning for the model, resulting in improved performance compared to using manually selected anchor boxes.

These points collectively demonstrate how YOLOv2 addresses several shortcomings of YOLOv1 and achieves better performance in terms of accuracy, recall, and ease of learning.

Section-D

Object Tracker: SORT + DeepSORT

Data Collection:

- We recorded 4 videos capturing a busy road.

Networks Utilized:

For car detection, we employed two networks:

1. Faster RCNN
2. YOLOv5

We applied the following object tracking methods for car counting:

1. SORT
2. Deep SORT

Procedure:

- Initially, the video is parsed into frames.
- Car detection is performed in each frame using either Faster RCNN or YOLOv5.

- Detected cars are filtered and tracked, with their IDs stored in a global vector.

- Car counts are maintained using unique IDs.

- Necessary annotations are added to the frames.

- Frames with annotations are saved to file.

Hyperparameters Used:

Four combinations were explored:

1. Faster RCNN + SORT

2. Faster RCNN + DeepSORT

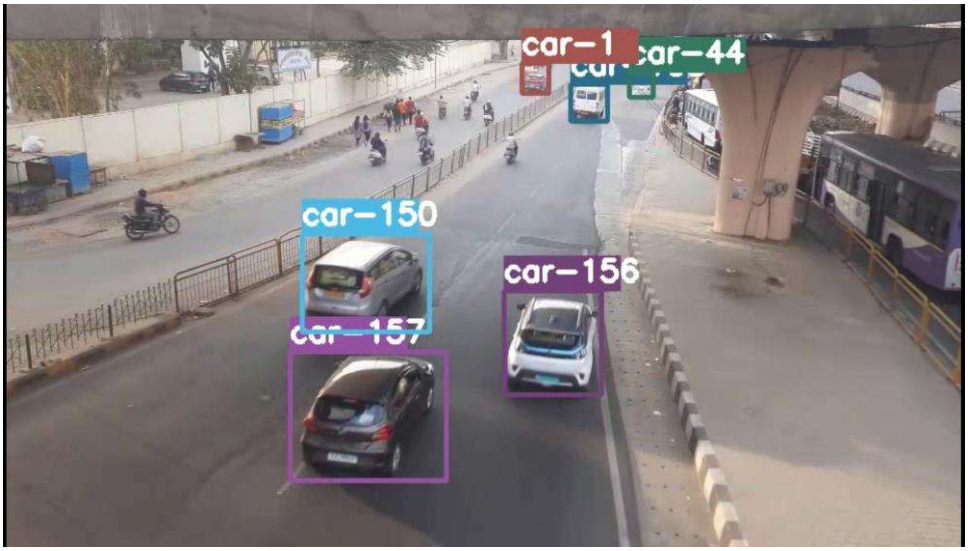
3. YOLOv5 + SORT

4. YOLOv5 + DeepSORT

Results:

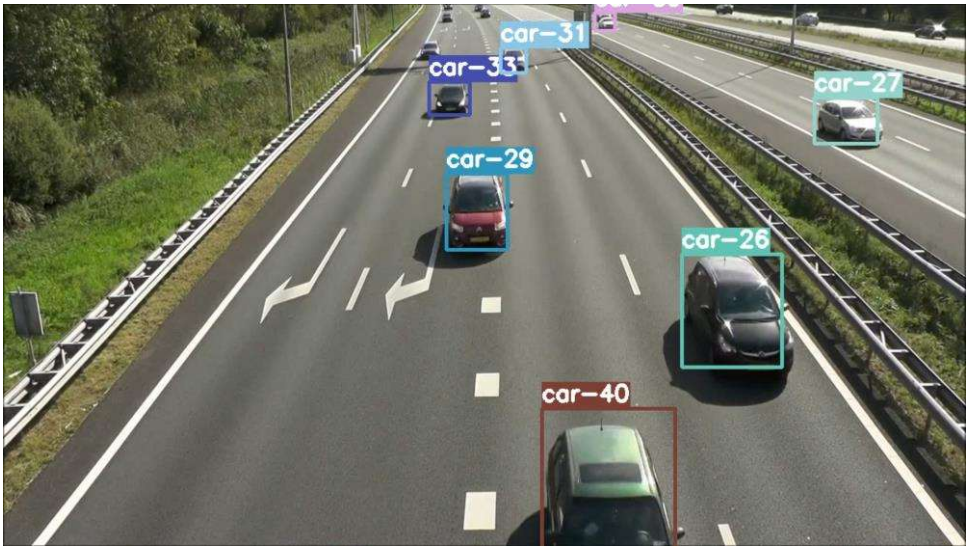
1. YOLOv5 + DeepSORT:

2. YOLOv5 + SORT:



3. Faster RCNN + SORT:

4. Faster RCNN + DeepSORT:



Observations:

1. In some instances, the detectors misclassify "Autos" as cars, likely due to the training data limitations.

2. YOLOv5 paired with any tracker exhibits faster detection and tracking compared to Faster RCNN.

3. Performance is notably poor in the third video, possibly due to blur, occlusion, and unreliable video sources.

4. Implementing a line of reference for counting could enhance accuracy, serving as a traffic control measure.

5. SORT occasionally detects a car multiple times, potentially due to interference or image blurring, while Deep SORT maintains consistent track IDs.

Ground Truths:

As ground truth data wasn't pre-labeled, cars were manually counted. Human error might affect these counts. Results for the defined combinations:

1. Video 1: Visually counted cars: 9

- Faster RCNN + SORT: 11
- Faster RCNN + DeepSORT: 11
- YOLOv5 + DeepSORT: 10
- YOLOv5 + SORT: 10

2. Video 2: Visually counted cars: 6

- Faster RCNN + SORT: 6
- Faster RCNN + DeepSORT: 6
- YOLOv5 + SORT: 6

- YOLOv5 + DeepSORT: 6

3. Video 3: Visually counted cars: 14

- Faster RCNN + SORT: 14
- Faster RCNN + DeepSORT: 14
- YOLOv5 + SORT: 22
- YOLOv5 + DeepSORT: 12

Faster RCNN vs. YOLOv5:

- YOLOv5 offers superior real-time frame rates, making it suitable for applications like camera feeds. However, Faster RCNN performs better in detecting distant objects, including cars.

SORT vs. Deep SORT:

- SORT, a conventional method, uses a Hungarian algorithm and Kalman filter for tracking. Deep SORT, based on deep learning, utilizes a neural network for appearance features, improving identity management. Deep SORT excels in handling appearance variations and occlusions but is slower than SORT. For real-time applications, SORT is preferred due to its simplicity and efficiency.

Conclusion:

- Considering the trade-off between accuracy and real-time processing, SORT is suitable for real-time detection on standard hardware, while

Deep SORT is preferable for applications prioritizing accuracy over frame rates.