Visual Recognition Mini-Project 1

Deepkumar Patel IMT2021011
Shashank Amarnath Lal IMT2021538
Sheikh Muteeb IMT2021008

March 2024

1 CIFAR-10 Classification

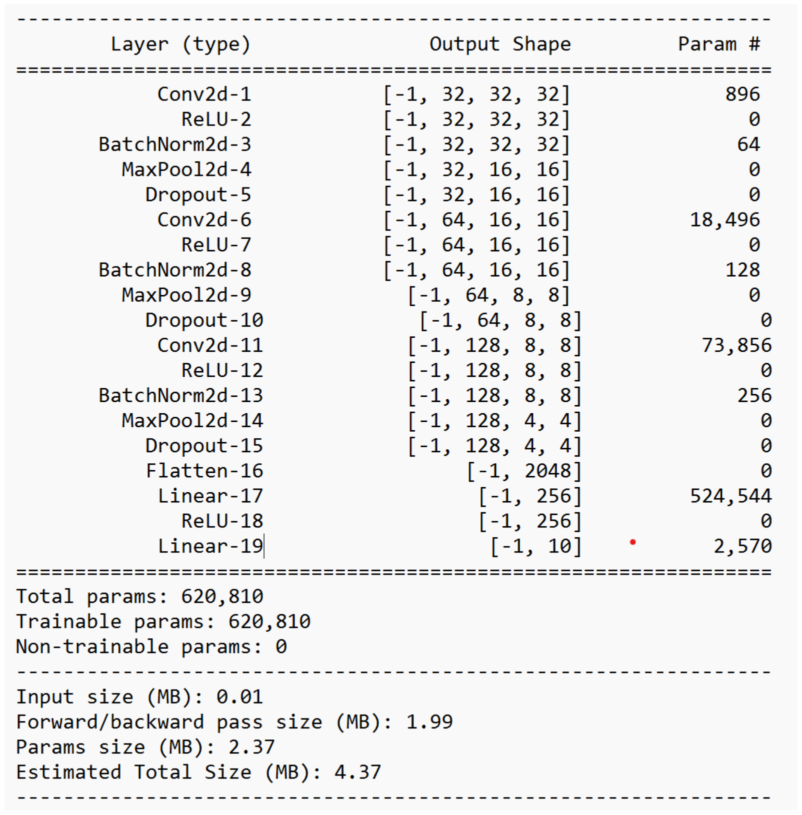
CIFAR-10 is a dataset containing 60,000 32x32 color images distributed across 10 classes, used primarily for image classification tasks in machine learning and computer vision research. It serves as a standard benchmark for evaluating the performance of classification algorithms and models, particularly deep learning approaches such as convolutional neural networks.

1.1 Architecture

We design a convolutional neural network comprising three convolutional layers followed by batch normalization, activation, max-pooling, and dropout layers. This architecture is augmented with two fully connected layers. Employing dropout aids in regularization, enhancing the network's generalization capabilities. All convolutional layers utilize a 3x3 kernel size and "same" padding. The parameter breakdown for each layer is depicted in the accompanying figure.1.

1. `Conv2d` layer with 3 input channels, 32 output channels, 3x3 kernel size, and same padding.
2. Selected activation layer (ReLU, Sigmoid, or Tanh).
3. `BatchNorm2d` layer with 32 channels.
4. `MaxPool2d` layer with 2x2 kernel size.
5. `Dropout` layer with a dropout rate of 0.16.
6. `Conv2d` layer with 32 input channels, 64 output channels, 3x3 kernel size, and same padding.
7. Selected activation layer.
8. `BatchNorm2d` layer with 64 channels.
9. `MaxPool2d` layer with 2x2 kernel size.
10. `Dropout` layer with a dropout rate of 0.16.

11. `Conv2d` layer with 64 input channels, 128 output channels, 3x3 kernel size, and same padding.
12. Selected activation layer.
13. `BatchNorm2d` layer with 128 channels.
14. `MaxPool2d` layer with 2x2 kernel size.
15. `Dropout` layer with a dropout rate of 0.16.
16. `Flatten` layer to flatten the output.
17. `Linear` layer with 2048 input features and 256 output features.
18. Selected activation layer.
19. `Linear` layer with 256 input features and 10 output features (number of classes).



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 32, 32]             896
              ReLU-2           [-1, 32, 32, 32]               0
       BatchNorm2d-3           [-1, 32, 32, 32]              64
         MaxPool2d-4           [-1, 32, 16, 16]               0
           Dropout-5           [-1, 32, 16, 16]               0
            Conv2d-6           [-1, 64, 16, 16]          18,496
              ReLU-7           [-1, 64, 16, 16]               0
       BatchNorm2d-8           [-1, 64, 16, 16]             128
         MaxPool2d-9             [-1, 64, 8, 8]               0
          Dropout-10             [-1, 64, 8, 8]               0
           Conv2d-11            [-1, 128, 8, 8]          73,856
             ReLU-12            [-1, 128, 8, 8]               0
      BatchNorm2d-13            [-1, 128, 8, 8]             256
        MaxPool2d-14            [-1, 128, 4, 4]               0
          Dropout-15            [-1, 128, 4, 4]               0
          Flatten-16                 [-1, 2048]               0
           Linear-17                  [-1, 256]         524,544
             ReLU-18                  [-1, 256]               0
           Linear-19                   [-1, 10]           2,570
================================================================
Total params: 620,810
Trainable params: 620,810
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 1.99
Params size (MB): 2.37
Estimated Total Size (MB): 4.37
----------------------------------------------------------------
```

Figure 1: Model Architecture, thanks to torchsummary

1.2 Training and Results

We develop modular code that incorporates parameters for different activation types, facilitating code modularization. Additionally, we design a versatile function to enable experimentation with various combinations of activations and optimizers, including Adam, RMSprop, and SGD. Throughout our experiments, we use a learning rate of $1.6 * 10^{-3}$ and a batch size of 256 across all combinations. For classification tasks, we utilize the CrossEntropy loss function.

Optimizer/Activation	Adam	RMSprop	SGD
ReLU	0.8405	0.83607	0.6361
Tanh	0.766	0.7573	0.4633
Sigmoid	0.8029	0.7777	0.4179

Table 1: Accuracy for every possible Optimizer, Activation pair.

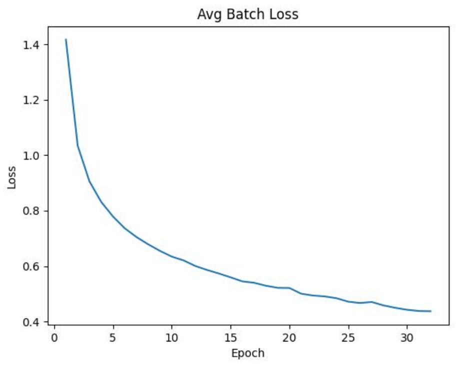
Optimizer/Activation	Adam	RMSProp	SGD
ReLU	29.36346	30.1354	30.3640
Tanh	29.97130	30.1388	29.5171
Sigmoid	30.36905	29.5278	29.6064

Table 2: Time per epoch for every possible Optimizer, Activation pair.

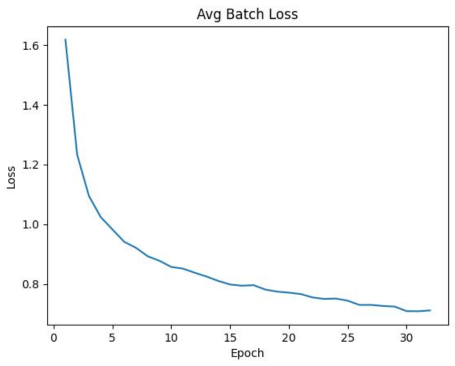
2 CNN as Feature Extractor

2.1 Architecture

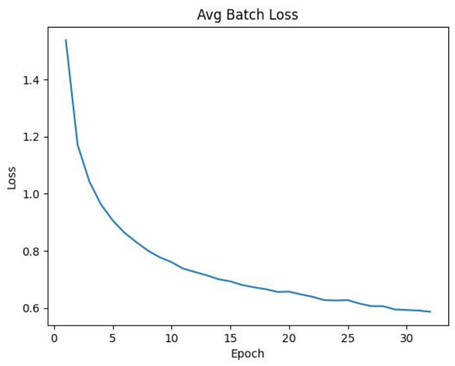
We know that CNNs are excellent feature extractors, as we can use the activations of the penultimate layer as a representation of the input image. We use the following architecture which is quite similar to the above architecture used for cifar classification (figure 5). We agree on using 64-dimensional features for our experiments. We experiment with the bike-horse dataset and the FashionMNIST dataset. FashionMNIST is a dataset containing 70,000 grayscale images of clothing items, divided into 10 categories. It serves as a benchmark for testing machine learning algorithms in image classification tasks, offering a more challenging alternative to the traditional MNIST handwritten digits dataset.



(a) Adam ReLU

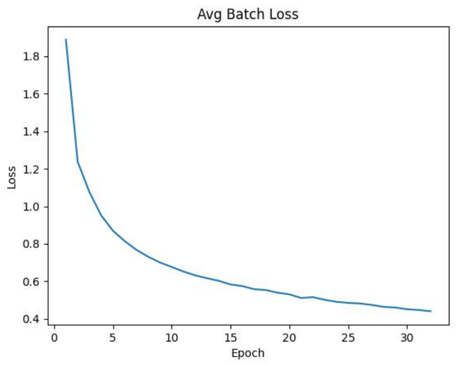


(b) Adam Tanh

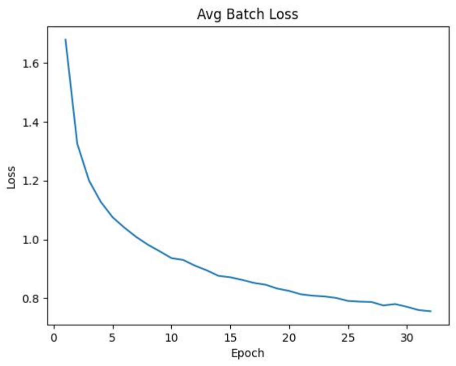


(c) Adam Sigmoid

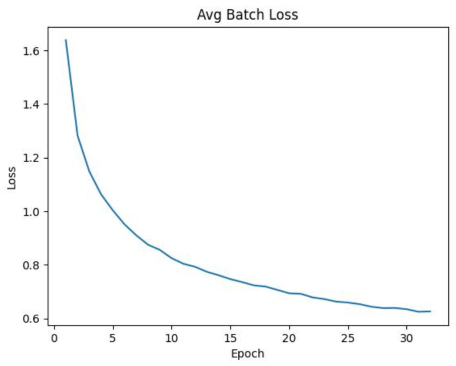
Figure 2: Average Batch Loss values for various activators with Adam optimizer over 32 epochs.



(a) RMSprop ReLU

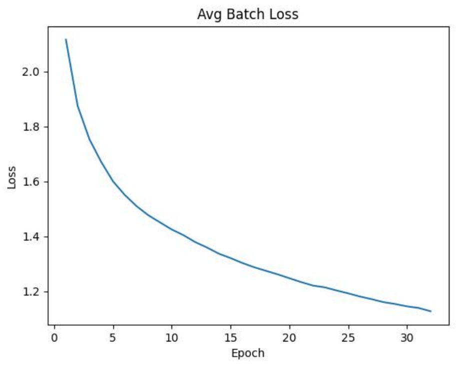


(b) RMSprop Tanh

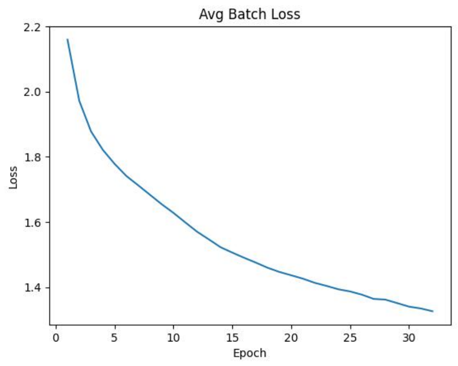


(c) RMSprop Sigmoid

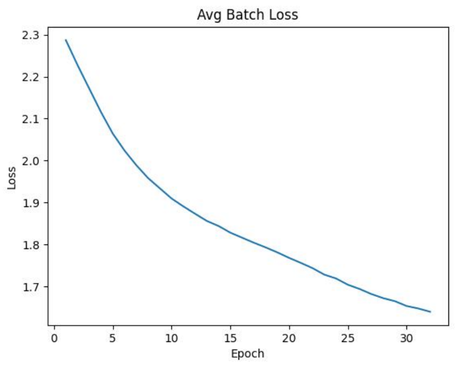
Figure 3: Average Batch Loss values for various activators with RMS optimizer over 32 epochs.



(a) SGD ReLU

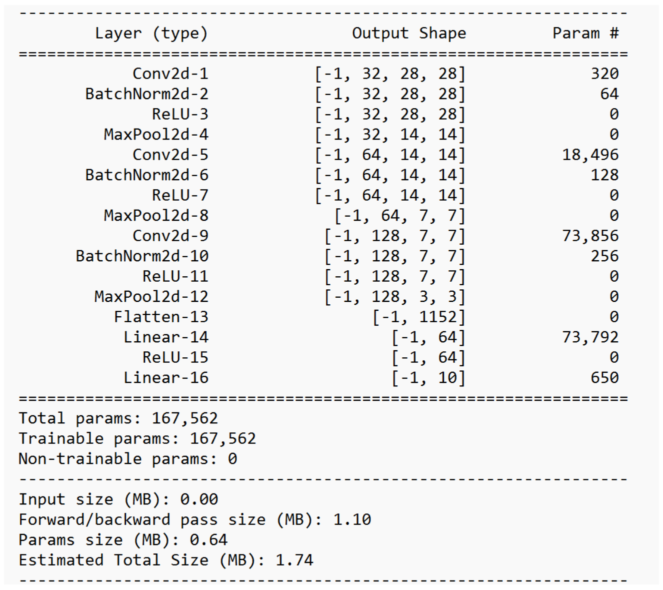


(b) SGD Tanh



(c) SGD Sigmoid

Figure 4: Average Batch Loss values for various activators with SGD optimizer over 32 epochs.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 28, 28]             320
       BatchNorm2d-2           [-1, 32, 28, 28]              64
              ReLU-3           [-1, 32, 28, 28]               0
         MaxPool2d-4           [-1, 32, 14, 14]               0
            Conv2d-5           [-1, 64, 14, 14]          18,496
       BatchNorm2d-6           [-1, 64, 14, 14]             128
              ReLU-7           [-1, 64, 14, 14]               0
         MaxPool2d-8             [-1, 64, 7, 7]               0
            Conv2d-9            [-1, 128, 7, 7]          73,856
      BatchNorm2d-10            [-1, 128, 7, 7]             256
             ReLU-11            [-1, 128, 7, 7]               0
        MaxPool2d-12            [-1, 128, 3, 3]               0
          Flatten-13                 [-1, 1152]               0
           Linear-14                   [-1, 64]          73,792
             ReLU-15                   [-1, 64]               0
           Linear-16                   [-1, 10]             650
================================================================
Total params: 167,562
Trainable params: 167,562
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 1.10
Params size (MB): 0.64
Estimated Total Size (MB): 1.74
----------------------------------------------------------------
```

Figure 5: Architecture for CNN as feature extractor, thanks to torchsummary.

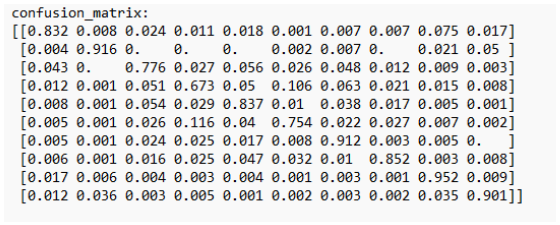
2.2 Training

We use the cross entropy loss with Adam optimizer. The learning rate was set to $3.2 * 10^{-4}$. We use the same architecture for both datasets. We train the model for 32 epochs. One can find the accuracy and the confusion matrix for all the models in the code files. For the bike horse dataset, since the size of the dataset is very small, we picked out six images randomly from both classes for the validation dataset.

This time the network will also return the activations of the penultimate layer, along with the logits, while training we use the logits from the model and ignore the activations. After we have trained the model for a sufficient number of epochs. now we take the training dataset, pass it to the network, and collect the activations. and form a new dataset of shape $(m, 64)$ where m is the number of data points. We decided to go with 64-dimensional representations. Now we use this as input for various models in scikit learn like random forest, logistic regression, and naive bayes. The accuracy results are in the following table 3. the confusion matrices can be found in the code files.

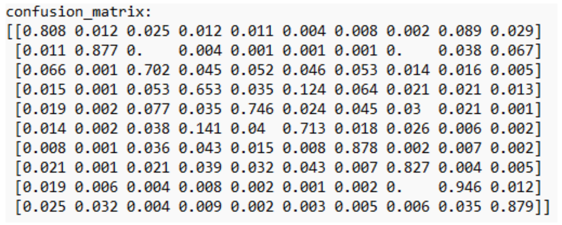
dataset/model	LogisticRegression	RandomForest	GaussianNB	MultinomialNB	OriginalCNN
FashionMNIST	0.9107	0.9132	0.9066	0.9086	0.9071
CarHorseDataset	1.0	1.0	1.0	1.0	1.0

Table 3: Accuracy found using CNN as a feature extractor for various datasets.



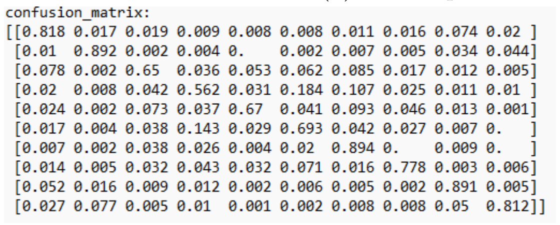
```
confusion_matrix:
[[0.832 0.008 0.024 0.011 0.018 0.001 0.007 0.007 0.075 0.017]
 [0.004 0.916 0.    0.    0.    0.002 0.007 0.    0.021 0.05 ]
 [0.043 0.    0.776 0.027 0.056 0.026 0.048 0.012 0.009 0.003]
 [0.012 0.001 0.051 0.673 0.05  0.106 0.063 0.021 0.015 0.008]
 [0.008 0.001 0.054 0.029 0.837 0.01  0.038 0.017 0.005 0.001]
 [0.005 0.001 0.026 0.116 0.04  0.754 0.022 0.027 0.007 0.002]
 [0.005 0.001 0.024 0.025 0.017 0.008 0.912 0.003 0.005 0.   ]
 [0.006 0.001 0.016 0.025 0.047 0.032 0.01  0.852 0.003 0.008]
 [0.017 0.006 0.004 0.003 0.004 0.001 0.003 0.001 0.952 0.009]
 [0.012 0.036 0.003 0.005 0.001 0.002 0.003 0.002 0.035 0.901]]
```

(a) Adam optimizer with ReLU activation



```
confusion_matrix:
[[0.808 0.012 0.025 0.012 0.011 0.004 0.008 0.002 0.089 0.029]
 [0.011 0.877 0.    0.004 0.001 0.001 0.001 0.    0.038 0.067]
 [0.066 0.001 0.702 0.045 0.052 0.046 0.053 0.014 0.016 0.005]
 [0.015 0.001 0.053 0.653 0.035 0.124 0.064 0.021 0.021 0.013]
 [0.019 0.002 0.077 0.035 0.746 0.024 0.045 0.03  0.021 0.001]
 [0.014 0.002 0.038 0.141 0.04  0.713 0.018 0.026 0.006 0.002]
 [0.008 0.001 0.036 0.043 0.015 0.008 0.878 0.002 0.007 0.002]
 [0.021 0.001 0.021 0.039 0.032 0.043 0.007 0.827 0.004 0.005]
 [0.019 0.006 0.004 0.008 0.002 0.001 0.002 0.    0.946 0.012]
 [0.025 0.032 0.004 0.009 0.002 0.003 0.005 0.006 0.035 0.879]]
```

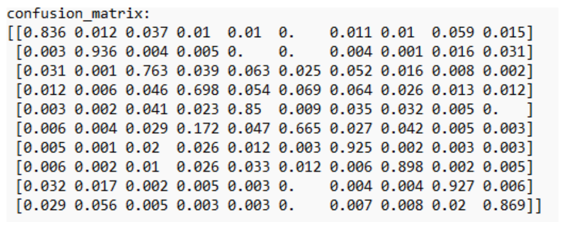
(b) Adam optimizer with Sigmoid activation



```
confusion_matrix:
[[0.818 0.017 0.019 0.009 0.008 0.008 0.011 0.016 0.074 0.02 ]
 [0.01  0.892 0.002 0.004 0.    0.002 0.007 0.005 0.034 0.044]
 [0.078 0.002 0.65  0.036 0.053 0.062 0.085 0.017 0.012 0.005]
 [0.02  0.008 0.042 0.562 0.031 0.184 0.107 0.025 0.011 0.01 ]
 [0.024 0.002 0.073 0.037 0.67  0.041 0.093 0.046 0.013 0.001]
 [0.017 0.004 0.038 0.143 0.029 0.693 0.042 0.027 0.007 0.   ]
 [0.007 0.002 0.038 0.026 0.004 0.02  0.894 0.    0.009 0.   ]
 [0.014 0.005 0.032 0.043 0.032 0.071 0.016 0.778 0.003 0.006]
 [0.052 0.016 0.009 0.012 0.002 0.006 0.005 0.002 0.891 0.005]
 [0.027 0.077 0.005 0.01  0.001 0.002 0.008 0.008 0.05  0.812]]
```

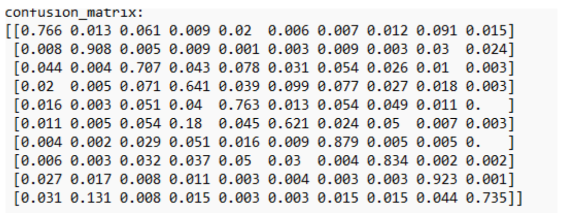
(c) Adam optimizer with Tanh activation

Figure 6: Confusion Matrx for Adam optimizer with different activation fuctions



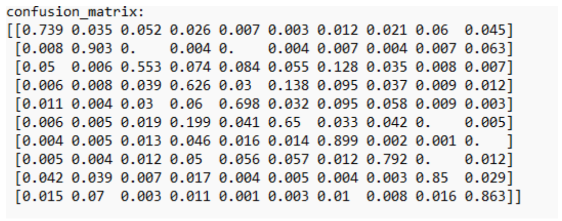
```
confusion_matrix:
[[0.836 0.012 0.037 0.01  0.01  0.    0.011 0.01  0.059 0.015]
 [0.003 0.936 0.004 0.005 0.    0.    0.004 0.001 0.016 0.031]
 [0.031 0.001 0.763 0.039 0.063 0.025 0.052 0.016 0.008 0.002]
 [0.012 0.006 0.046 0.698 0.054 0.069 0.064 0.026 0.013 0.012]
 [0.003 0.002 0.041 0.023 0.85  0.009 0.035 0.032 0.005 0.   ]
 [0.006 0.004 0.029 0.172 0.047 0.665 0.027 0.042 0.005 0.003]
 [0.005 0.001 0.02  0.026 0.012 0.003 0.925 0.002 0.003 0.003]
 [0.006 0.002 0.01  0.026 0.033 0.012 0.006 0.898 0.002 0.005]
 [0.032 0.017 0.002 0.005 0.003 0.    0.004 0.004 0.927 0.006]
 [0.029 0.056 0.005 0.003 0.003 0.    0.007 0.008 0.02  0.869]]
```

(a) RMS optimizer with ReLU activation



```
confusion_matrix:
[[0.766 0.013 0.061 0.009 0.02  0.006 0.007 0.012 0.091 0.015]
 [0.008 0.908 0.005 0.009 0.001 0.003 0.009 0.003 0.03  0.024]
 [0.044 0.004 0.707 0.043 0.078 0.031 0.054 0.026 0.01  0.003]
 [0.02  0.005 0.071 0.641 0.039 0.099 0.077 0.027 0.018 0.003]
 [0.016 0.003 0.051 0.04  0.763 0.013 0.054 0.049 0.011 0.   ]
 [0.011 0.005 0.054 0.18  0.045 0.621 0.024 0.05  0.007 0.003]
 [0.004 0.002 0.029 0.051 0.016 0.009 0.879 0.005 0.005 0.   ]
 [0.006 0.003 0.032 0.037 0.05  0.03  0.004 0.834 0.002 0.002]
 [0.027 0.017 0.008 0.011 0.003 0.004 0.003 0.003 0.923 0.001]
 [0.031 0.131 0.008 0.015 0.003 0.003 0.015 0.015 0.044 0.735]]
```

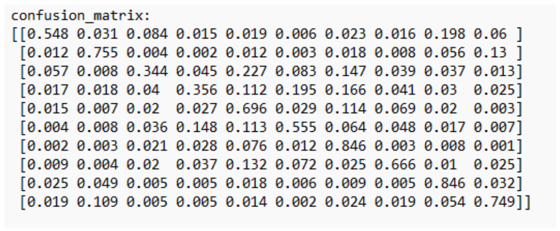
(b) RMS optimizer with Sigmoid activation



```
confusion_matrix:
[[0.739 0.035 0.052 0.026 0.007 0.003 0.012 0.021 0.06  0.045]
 [0.008 0.903 0.    0.004 0.    0.004 0.007 0.004 0.007 0.063]
 [0.05  0.006 0.553 0.074 0.084 0.055 0.128 0.035 0.008 0.007]
 [0.006 0.008 0.039 0.626 0.03  0.138 0.095 0.037 0.009 0.012]
 [0.011 0.004 0.03  0.06  0.698 0.032 0.095 0.058 0.009 0.003]
 [0.006 0.005 0.019 0.199 0.041 0.65  0.033 0.042 0.    0.005]
 [0.004 0.005 0.013 0.046 0.016 0.014 0.899 0.002 0.001 0.   ]
 [0.005 0.004 0.012 0.05  0.056 0.057 0.012 0.792 0.    0.012]
 [0.042 0.039 0.007 0.017 0.004 0.005 0.004 0.003 0.85  0.029]
 [0.015 0.07  0.003 0.011 0.001 0.003 0.01  0.008 0.016 0.863]]
```

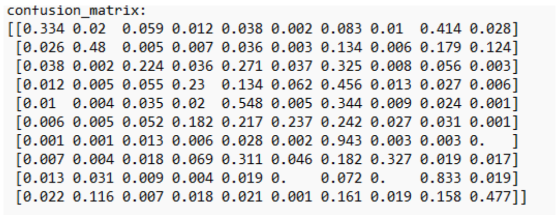
(c) RMS optimizer with Tanh activation

Figure 7: Confusion Matrix for RMS optimizer with different activation functions



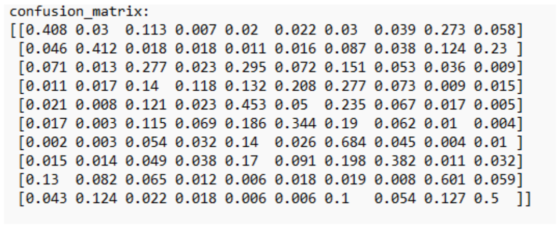
```
confusion_matrix:
[[0.548 0.031 0.084 0.015 0.019 0.006 0.023 0.016 0.198 0.06 ]
 [0.012 0.755 0.004 0.002 0.012 0.003 0.018 0.008 0.056 0.13 ]
 [0.057 0.008 0.344 0.045 0.227 0.083 0.147 0.039 0.037 0.013]
 [0.017 0.018 0.04  0.356 0.112 0.195 0.166 0.041 0.03  0.025]
 [0.015 0.007 0.02  0.027 0.696 0.029 0.114 0.069 0.02  0.003]
 [0.004 0.008 0.036 0.148 0.113 0.555 0.064 0.048 0.017 0.007]
 [0.002 0.003 0.021 0.028 0.076 0.012 0.846 0.003 0.008 0.001]
 [0.009 0.004 0.02  0.037 0.132 0.072 0.025 0.666 0.01  0.025]
 [0.025 0.049 0.005 0.005 0.018 0.006 0.009 0.005 0.846 0.032]
 [0.019 0.109 0.005 0.005 0.014 0.002 0.024 0.019 0.054 0.749]]
```

(a) Confusion Matrix for SGD optimizer with ReLU activation



```
confusion_matrix:
[[0.334 0.02  0.059 0.012 0.038 0.002 0.083 0.01  0.414 0.028]
 [0.026 0.48  0.005 0.007 0.036 0.003 0.134 0.006 0.179 0.124]
 [0.038 0.002 0.224 0.036 0.271 0.037 0.325 0.008 0.056 0.003]
 [0.012 0.005 0.055 0.23  0.134 0.062 0.456 0.013 0.027 0.006]
 [0.01  0.004 0.035 0.02  0.548 0.005 0.344 0.009 0.024 0.001]
 [0.006 0.005 0.052 0.182 0.217 0.237 0.242 0.027 0.031 0.001]
 [0.001 0.001 0.013 0.006 0.028 0.002 0.943 0.003 0.003 0.   ]
 [0.007 0.004 0.018 0.069 0.311 0.046 0.182 0.327 0.019 0.017]
 [0.013 0.031 0.009 0.004 0.019 0.    0.072 0.    0.833 0.019]
 [0.022 0.116 0.007 0.018 0.021 0.001 0.161 0.019 0.158 0.477]]
```

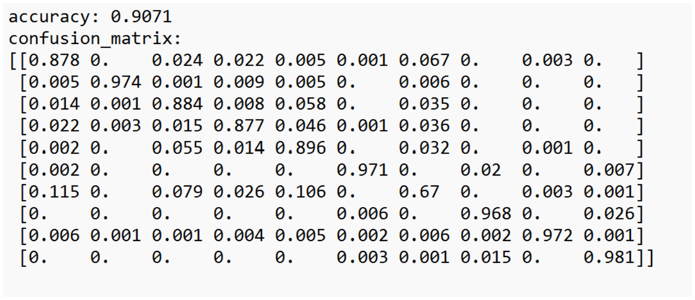
(b) Confusion Matrix for SGD optimizer with Tanh activation



```
confusion_matrix:
[[0.408 0.03  0.113 0.007 0.02  0.022 0.03  0.039 0.273 0.058]
 [0.046 0.412 0.018 0.018 0.011 0.016 0.087 0.038 0.124 0.23 ]
 [0.071 0.013 0.277 0.023 0.295 0.072 0.151 0.053 0.036 0.009]
 [0.011 0.017 0.14  0.118 0.132 0.208 0.277 0.073 0.009 0.015]
 [0.021 0.008 0.121 0.023 0.453 0.05  0.235 0.067 0.017 0.005]
 [0.017 0.003 0.115 0.069 0.186 0.344 0.19  0.062 0.01  0.004]
 [0.002 0.003 0.054 0.032 0.14  0.026 0.684 0.045 0.004 0.01 ]
 [0.015 0.014 0.049 0.038 0.17  0.091 0.198 0.382 0.011 0.032]
 [0.13  0.082 0.065 0.012 0.006 0.018 0.019 0.008 0.601 0.059]
 [0.043 0.124 0.022 0.018 0.006 0.006 0.1   0.054 0.127 0.5  ]]
```

(c) Confusion Matrix for SGD optimizer with Sigmoid activation

Figure 8: Confusion Matrix for SGD optimizer with different activation functions



```
accuracy: 0.9071
confusion_matrix:
[[0.878 0.    0.024 0.022 0.005 0.001 0.067 0.    0.003 0.   ]
 [0.005 0.974 0.001 0.009 0.005 0.    0.006 0.    0.    0.   ]
 [0.014 0.001 0.884 0.008 0.058 0.    0.035 0.    0.    0.   ]
 [0.022 0.003 0.015 0.877 0.046 0.001 0.036 0.    0.    0.   ]
 [0.002 0.    0.055 0.014 0.896 0.    0.032 0.    0.001 0.   ]
 [0.002 0.    0.    0.    0.    0.971 0.    0.02  0.    0.007]
 [0.115 0.    0.079 0.026 0.106 0.    0.67  0.    0.003 0.001]
 [0.    0.    0.    0.    0.    0.006 0.    0.968 0.    0.026]
 [0.006 0.001 0.001 0.004 0.005 0.002 0.006 0.002 0.972 0.001]
 [0.    0.    0.    0.    0.    0.003 0.001 0.015 0.    0.981]]
```

Figure 9: Confusion Matrix for FashionMNIST using the CNN.

3 YOLO V2 v/s YOLO V1

Batch Normalization: YOLOv2 incorporates batch normalization into its architecture, which helps in stabilizing and accelerating the training process by normalizing the activations in each mini-batch. This helps in reducing internal covariate shift and allows the model to learn faster and generalize better.

High-resolution Classifier: YOLOv2 uses a higher resolution classifier compared to YOLOv1. This enables the model to better capture fine-grained details in images, leading to improved detection accuracy, especially for small objects.

Anchor Boxes: YOLO v2 introduces the concept of anchor boxes, which are predetermined bounding boxes with specific aspect ratios. Instead of predicting bounding boxes directly, YOLO v2 predicts offsets to these anchor boxes. This approach helps in handling objects of various sizes and aspect ratios more effectively.

Dimension Clusters for Anchor Boxes: YOLO v2 uses k-means clustering on the training data to automatically determine the optimal anchor box shapes and sizes. This data-driven approach ensures that the anchor boxes are well-suited for the dataset at hand, leading to improved detection performance.

Multi-scale Training: YOLO v2 employs a multi-scale training strategy where it trains on images of different resolutions in a single batch. This allows the model to effectively learn features at different scales, making it more robust to variations in object sizes and improving overall detection accuracy.

These additional features contribute to the improved performance and efficiency of YOLOv2 compared to its predecessor, YOLO v1.

4 Object Tracker: SORT + DeepSORT

4.1 Introduction

In this section, we present our approach to counting cars in traffic junction videos using a combination of object detection and object tracking techniques. The objective is to develop an algorithm that accurately counts the number of cars passing through a traffic junction in real-time. We employed algorithms including YOLOv5 and Faster R-CNN for object detection, and SORT (Simple Online and Realtime Tracking) and DeepSORT (Deep Simple Online and Realtime Tracking) for object tracking.

4.2 Methodology

1. **Object Detection:** We utilized YOLOv5 and Faster R-CNN for object detection in the traffic junction videos. YOLOv5 offers a good balance between speed and accuracy, while Faster R-CNN is known for its robustness in detecting objects with high precision.

2. **Object Tracking:** For object tracking, we experimented with both SORT and DeepSORT algorithms. SORT is a simple yet effective algorithm for online and real-time tracking, while DeepSORT enhances SORT by incorporating deep appearance features for better tracking performance, especially in crowded scenes.

3. **Car Counting:** For SORT we kept track of the id's of cars detected so far and every frame if some new id is detected we appended it to the array and increased count by 1. For DeepSORT we used it's in built counter for storing highest id detected so far, which we use as a car counter.

4.3 Implementation

- We first preprocessed the traffic junction videos to extract individual frames.
- These frames were then fed into the object detection models (YOLOv5 and Faster R-CNN) to identify cars within each frame.
- The detected cars were subsequently passed to the object tracking modules (SORT and DeepSORT) to track their movements across frames.
- Finally, we devised a mechanism to count the number of unique car tracks to determine the total count of cars passing through the junction.

4.4 Experimentation

1. **Experiment 1:** We evaluated the performance of YOLOv5 and Faster R-CNN individually for object detection.

2. **Experiment 2:** We compared the tracking accuracy of SORT and DeepSORT on a dataset of traffic junction videos with varying levels of congestion.

3. **Experiment 3:** Hyperparameter tuning was conducted for SORT and DeepSORT algorithms to enhance tracking performance.

4. **Experiment 4:** To improve the speed of Faster R-CNN implementation, we experimented with skipping frames in the video processing pipeline. Different frame skipping intervals were tested to find the optimal balance between speed and detection accuracy.

5. **Experiment 5:** Additionally, we varied the confidence thresholds for both YOLOv5 and Faster R-CNN to explore their impact on detection accuracy and processing speed. Different confidence thresholds were tested to analyze their effects on false positives and missed detections.

4.5 Results and Observations

```
Parameters used for SORT
 max_age = 2500
 min_hits = 5
Parameters used for DeepSORT
 max_age = 300
 max_iou_distance = 0.3
Confidence Threshold for valid Cars FasterRCNN = 0.9
Confidence Threshold for valid Cars YOLOV5 = 0.7
```

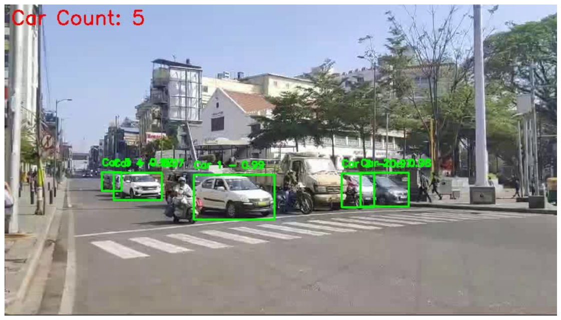
- **FasterRCNN and SORT:**
  - **Video 1** *(sparse)*: Count = 6
  - **Video 2** *(Congested)*: Count = 17
- **FasterRCNN and DeepSORT:**
  - **Video 1** *(sparse)*: Count = 12
  - **Video 2** *(Congested)*: Count = 25
- **YOLOV5 and SORT:**
  - **Video 1** *(sparse)*: Count = 5
  - **Video 2** *(Congested)*: Count = 8
- **YOLOV5 and DeepSORT:**
  - **Video 1** *(sparse)*: Count = 11
  - **Video 2** *(Congested)*: Count = 8

4.6 Conclusion

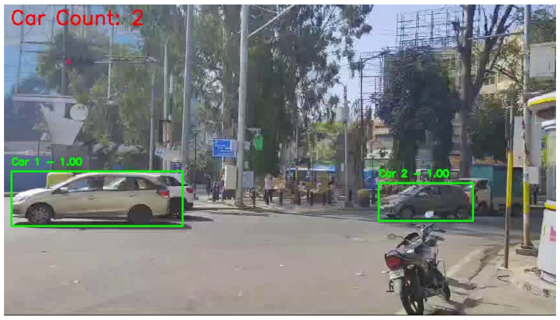
According to our experimentation and analysis we have come to the following conclusions about the performance of different object detection and tracking algorithms for car counting in traffic junction videos.

We observed that YOLOv5 combined with SORT demonstrated excellent overall performance. YOLOv5's fast inference speed coupled with SORT's effective tracking capabilities resulted in accurate car counting in various traffic scenarios. This combination proved particularly effective in dense traffic junctions, where real-time tracking of multiple cars is essential.

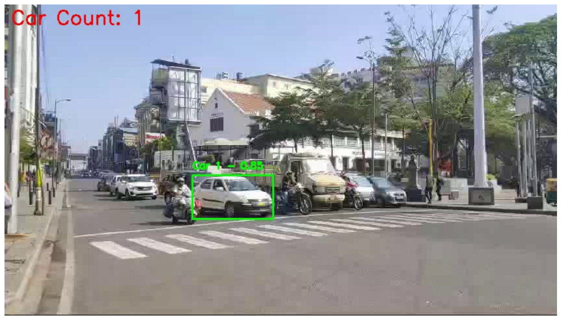
On the other hand, Faster R-CNN exhibited impressive speed in detecting cars, making it suitable for scenarios requiring quick processing. However, it also showed a higher tendency for false detections, especially in noisy environments. This drawback needs to be addressed to ensure reliable car counting results. We found that utilizing Faster R-CNN with a high confidence threshold value improved its performance, mitigating the issue of false detections and enhancing overall accuracy.



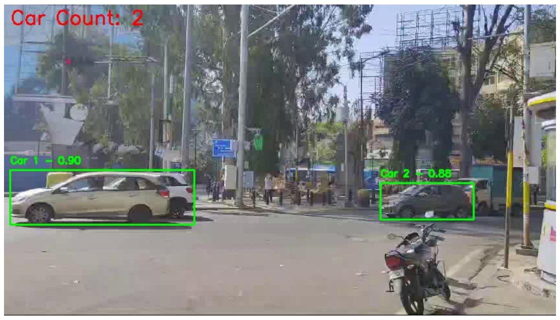
(a) FasterRCNN with SORT



(b) FasterRCNN with DeepSORT



(c) YOLOV5 with SORT



(d) YOLOV5 with DeepSORT

Figure 10: These are the first frames of the output videos, which can be found in the files.

In sparse traffic junction videos, YOLOv5 with DeepSORT did not perform as well as expected. DeepSORT, despite its ability to handle occlusions and complex scenes, may struggle in scenarios with fewer objects to track. This suggests a need for further optimization or alternative approaches for sparse traffic scenarios.

In conclusion, the choice of object detection and tracking algorithms can vary depending on the specific requirements of the application, including speed, accuracy, and environmental conditions.