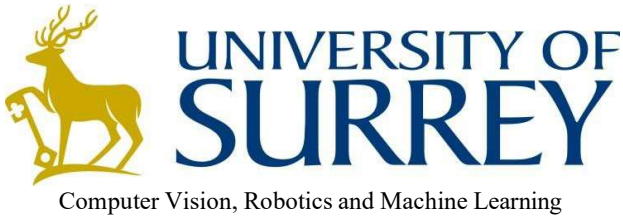
VEHICLE RE-IDENTIFICATION

VAISHALI SHIVAKUMAR (6738581)

Assignment Report for Advanced Computer Vision and Deep Learning (EEEM071)



Department of Electronic Engineering
Faculty of Engineering and Physical Sciences University of Surrey

Guildford, Surrey, GU2 7XH, UK

Table of Contents

VEHICLE_REIDENTIFICATION

VAISHALI SHIVAKUMAR (6738581)

Abstract

Convolutional Neural Networks (CNNs) have become a popular tool for solving image recognition problems, including vehicle re-identification (Re-ID) which is a challenging task due to variations in camera viewpoint, lighting conditions, and occlusion. The objective of this coursework is to explore the use of CNNs for vehicle Re-ID and investigate the performance of different CNN architectures and hyperparameters for this task. The VeRi dataset is used in this project, which contains over 50,000 images of 776 vehicles captured by 20 cameras covering a 1.0 km2 area in 24 hours. This report describes the methodology followed, which includes running the provided demo code, experimenting with different CNN architectures, varying hyperparameters, evaluating the best-performing model on the VeRi dataset and analyzing the results. The report concludes with a summary of the work and suggestions for future research.

1. Introduction

In recent years, Convolutional Neural Networks (CNNs) have become a popular tool for solving image recognition problems, including image retrieval. CNNs are deep neural networks that are trained to identify patterns and features in images. One of the applications of image retrieval is vehicle re-identification (Re-ID), which involves matching images of the same vehicle captured by different cameras.

1.1. Background

Vehicle Re-ID is an important research area in computer vision and has many real-world applications, including surveillance, traffic management, and law enforcement. However, it is a challenging problem due to variations in camera viewpoint, lighting conditions, and occlusion.

1.2. Problem Statement

The objective of this coursework is to explore the use of CNNs for vehicle Re-ID. Specifically, we will investigate the performance of different CNN architectures and hyperparameters for this task. We will also use the VeRi dataset, which contains over 50,000 images of 776 vehicles captured by 20 cameras covering a 1.0 km2 area in 24 hours [8].

1.3. Objectives

The main objectives of this coursework are:

- To run the provided demo code and understand its implementation details

- To experiment with different CNN architectures and hyperparameters to identify the best-performing model

- To evaluate the performance of the chosen model on the VeRi dataset

- To analyze and interpret the results

1.4. Methodology

The first step is to run the demo code provided and study its implementation details. The next step is to experiment with different CNN architectures, including existing architectures like ResNet, and modified versions of those architectures. Parallelly the hyperparameters such as learning rate, weight decay, and optimizer are varied to identify the best-performing model.

Once the best-performing model is identified, the performance on the VeRi dataset is evaluated and the state-of-the-art methods are compared. Then the results are analyzed and interpreted and the implications of the findings are discussed. Finally, the report is concluded with a summary of the work and suggestions for future research.

2. Literature Review

2.1. Introduction to Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a type of deep learning neural network that has been successfully used in various image recognition tasks, including image retrieval. CNNs are designed to automatically extract and learn features from input images through multiple layers of convolution, pooling, and non-linear activation functions. The learned features can then be used for classification or retrieval tasks.

2.2. Image Retrieval using CNNs

Image retrieval using CNNs involves using a pre-trained CNN to extract features from input images and

then using these features to search for similar images in a database. This approach has been successful in various image retrieval applications, including vehicle re-identification.

2.3. Existing CNN Architectures

There are various existing CNN architectures that have been used for image retrieval, including ResNet, VGG, and MobileNet. The following models are available in the demo code for this coursework:

1. ResNet18: ResNet-18 is a convolutional neural network architecture with 18 layers. It was introduced by He et al. in 2016 and is widely used in computer vision tasks due to its relatively small size and good performance. [1]

2. ResNet18_fc512: This is a variant of ResNet-18, with a fully connected layer (fc) of 512 neurons added to the end of the network. This architecture has been shown to improve the performance of ResNet-18 on some computer vision tasks. [2]

3. ResNet34: ResNet-34 is a convolutional neural network architecture with 34 layers. It was also introduced by He et al. in 2016 and is similar to ResNet-18, but deeper and with more layers. [1]

4. ResNet34_fc512: This is a variant of ResNet-34, with a fully connected layer of 512 neurons added to the end of the network. Similar to ResNet-18_fc512, this architecture has been shown to improve the performance of ResNet-34 on some computer vision tasks. [2]

5. ResNet50: ResNet-50 is a convolutional neural network architecture with 50 layers. It was also introduced by He et al. in 2016 and is similar to ResNet-18 and ResNet-34, but even deeper and with more layers. [1]

6. ResNet50_fc512: This is a variant of ResNet-50, with a fully connected layer of 512 neurons added to the end of the network. Similar to ResNet-18_fc512 and ResNet-34_fc512, this architecture has been shown to improve the performance of ResNet-50 on some computer vision tasks. [2]

7. MobileNet V3 Small: MobileNet V3 is a family of lightweight neural network architectures designed for mobile and embedded devices. The "small" version is the smallest and fastest variant, with fewer parameters and lower computational cost. It was introduced by Howard et al. in 2019. [3]

8. VGG16: VGG-16 is a convolutional neural network architecture with 16 layers. It was introduced by Simonyan and Zisserman in 2014 and is known for its simplicity and good performance on image classification tasks. [4]

2.4. Hyper-parameters for CNNs

CNNs have various hyper-parameters that can be adjusted to optimize performance for a specific task. Some of the hyper-parameters that can be adjusted in the demo code for this coursework include:

Hyperparameters that can significantly affect the model's performance include:

1. Learning rate: This is the rate at which the model updates its parameters during training. A learning rate that is too high may cause the model to overshoot the optimal values and never converge, while a learning rate that is too low may cause the model to converge too slowly or get stuck in a local minimum [6].

2. Weight decay: This is a regularization technique that adds a penalty term to the loss function to prevent the model from overfitting. A higher weight decay value will penalize large weights more heavily, while a lower value will allow the model to fit the training data more closely [9].

3. Optimizer: The choice of optimizer can also affect the model's performance. Popular optimizers include stochastic gradient descent (SGD), Adam, and Adagrad. Each optimizer has its own strengths and weaknesses, and the choice will depend on the specific problem and dataset [8].

4. Dropout rate: Dropout is another regularization technique that randomly drops out nodes in the neural network during training to prevent overfitting. The dropout rate is the probability that a node will be dropped out, and a higher dropout rate will result in more regularization [7].

5. Batch size: The batch size is the number of samples that are processed by the model in each training iteration. A larger batch size can result in faster training times, but may also lead to overfitting or poor generalization to new data [6].

6. Number of epochs: The number of epochs is the number of times the model is trained on the entire dataset. Training for too few epochs may result in underfitting, while training for too many epochs may result in overfitting. The optimal number of epochs will depend on the specific problem and dataset [6].

3. Methodology

3.1. Data Collection

For this coursework, the VeRi dataset is used, which contains over 50,000 images of 776 vehicles captured by 20 cameras covering a 1.0 km^2 area in 24 hours. The dataset has been collected from multiple cameras

in different locations and under different weather and lighting conditions.

3.2. Preprocessing of Data

Before feeding the data to the CNN, some preprocessing steps are done, including resizing the images to a fixed size, converting them to grayscale, and normalizing the pixel values to have zero mean and unit variance. In addition, to increase the amount of training data and improve the model's ability to generalize to new examples, data augmentation techniques such as random scaling and random rotation have been implemented. Random scaling randomly zooms the images in or out by a factor of up to 150% of the original size of the image, while random rotation applies a random rotation angle of up to 15 degrees in either direction. These techniques help to reduce overfitting and improve the model's performance on new, unseen data.

3.3. CNN Architecture Selection

The first step is to experiment with different CNN architectures to find the best performing model for the vehicle Re-ID task. In addition to the existing architectures such as ResNet, modified versions of those architectures, VGG16, and MobileNetV3_Small, we have also included AlexNet and a custom model called CustomVeRiModel for our initial experiments. These models will be trained and evaluated on the dataset to determine their performance on the task of vehicle Re-ID. By experimenting with multiple architectures, we aim to identify the most suitable CNN model for this specific task and optimize its performance for accurate vehicle identification.

3.4. Hyper-parameters Selection

Additionally, different hyper-parameters for the CNN will be experimented with. These hyper-parameters, including the learning rate, weight decay, optimizer, and batch size, will be explored to evaluate their impact on the model's performance.

4. Results and Analysis

In this section, we present the results of our experiments and analyze the performance of different CNN architectures and hyperparameters for vehicle Re-ID.

4.1. Evaluation Metrics

To evaluate the performance of the models, several standard evaluation metrics, including mean Average Precision (mAP) and Cumulative Matching Curve (CMC), are used.

4.2. Performance Comparison of Different CNN Architectures

In the first experiment, Several CNN architectures, including MobileNetV3_small, ResNet18, VGG16, and AlexNet, were experimented with for the vehicle Re-ID task with default parameters given in the demo code. The default parameters are tabulated in table 1 below.

Table 1: Default hyperparameters

height	224
width	224
optimizer	amsgrad
Learning rate	0.0003
Momentum	0.9
Weight Decay	0.0005
No of Epochs	60
Step size	20 40
Train-batch-size	64
Test-batch-size	100

The performance of each architecture was evaluated on the VeRi dataset, and the results are presented in Table 2. The table shows the mean average precision (mAP) and Rank-1 accuracy of each architecture with default parameters. It is observed that ResNet18 performs the best among the tested architectures, achieving a mAP of 59.3% and Rank-1 accuracy of 88.6%.

Table 2: Performance of Different CNN Architectures with default parameters

Architecture	mAP	Rank-1
ResNet18	59.3%	88.6%
MobileNetV3_small	49.5%	84.3%
VGG16	51.9%	83.2%
AlexNet	24.5%	56.4%

4.3. Performance Comparison of Different Hyper-parameters and Data Augmentation

Different hyperparameters, including learning rate, weight decay, optimizer, batch size, momentum and also different data augmentation techniques such as random-erase, and colour-jitter were also experimented with. Also, user-defined transforms such as Random scaling, and random rotation were also experimented with. All five experiments were conducted with MobilenetV3, VGG16 and Alexnet and reduced the epochs to 20 from a default value of 60.

4.3.1. Experiment 2 – Varying Learning Rate and Batch Size

In this section, an experiment to study the effect of varying learning rates and batch size on the performance of different CNN architectures. The experimental setup involved training the models for 20 epochs, with a step size of 20 and 40. The train-batch size was set to 128, and the test-batch size was set to 100. The input image size was 224x224, and the optimizer used was amsgrad as given in Table 3.

Table 3: Hyperparameters for Experiment 2

height	224
width	224
optimizer	amsgrad
Learning rate	0.001
Momentum	0.9
Weight Decay	0.0005
No of Epochs	20
Step size	20 40
Train-batch-size	128
Test-batch-size	100

The performance of different CNN architectures, including MobileNetV3_small, VGG16, and AlexNet, was evaluated using the mAP and Rank-1 metrics. The results are shown in Table 2. It can be observed that MobileNetV3_small outperformed the other architectures, achieving a mAP of 0.41 and a Rank-1 accuracy of 0.719. VGG16 and AlexNet performed poorly, achieving mAP values of 0.13 and 0.159, respectively.

Table 4: Performance of Different CNN Architectures in Experiment 2

Architecture	mAP	Rank-1
MobileNetV3_small	41%	71.9%
VGG16	33.5%	63.8%
AlexNet	15.9%	45.5%

Thus it can be concluded that varying the learning rate and batch size had a significant impact on the performance of the models, with MobileNetV3_small being the best-performing architecture in this experimental setup.

4.3.2. Experiment 3 – Varying Optimizer, learning Rate, step size, Train Batch Size and Momentum

In this section, the third experiment was conducted to investigate the effect of varying hyperparameters such as optimizer, learning rate, step size, train batch size, and momentum on the performance of different CNN architectures. The hyperparameters for the experiment are shown in Table 5.

Table 5: Hyperparameters for Experiment 3

height	224
width	224
optimizer	sgd
Learning rate	0.001
Momentum	0.5
Weight Decay	0.0005
No of Epochs	20
Step size	5 10
Train-batch-size	250
Test-batch-size	100

The performance of each architecture was evaluated based on mAP and Rank-1 accuracy, and the results are presented in Table 6. It can be observed that the performance of all three architectures decreased significantly compared to the previous experiments. The best-performing architecture in this experiment was MobileNetV3_small, with an mAP of 6.6% and Rank-1 accuracy of 25.9%.

Table 6: Performance of Different CNN Architectures in Experiment 3

Architecture	mAP	Rank-1
MobileNetV3_small	6.6%	25.9%
VGG16	4.5%	18%
AlexNet	1.2%	3.3%

Overall, this experiment highlights the importance of selecting appropriate hyperparameters to achieve optimal performance in Re-ID tasks. It also emphasizes the need for extensive experimentation and fine-tuning of hyperparameters to achieve the best possible results.

4.3.3. Experiment 4 – Varying Learning rate, Step Size, Batch Size and Weight Decay

In Experiment 4, the effect of varying hyperparameters such as learning rate, step size, batch size, and weight decay on the performance of different CNN architectures was evaluated. The hyperparameters used for this experiment are shown in Table 7. The architectures considered for this experiment were MobileNetV3_small, VGG16, and AlexNet. The performance of each architecture is reported in Table 8 in terms of mAP and Rank-1 accuracy.

Table 7: Hyperparameters for Experiment 4

height	224
width	224
optimizer	amsgrad

Learning rate	0.0001
Momentum	0.9
Weight Decay	0.0001
No of Epochs	20
Step size	5 10
Train-batch-size	400
Test-batch-size	100

It was observed that AlexNet outperformed the other architectures, achieving a mAP of 54.3% and Rank-1 accuracy of 97.4%. MobileNetV3_small and VGG16 achieved similar performance with an mAP and Rank-1 accuracy of 37.7% and 70.4%, respectively. The results suggest that using a smaller learning rate, larger batch size, and weight decay can improve the performance of the CNN models. However, the optimal hyperparameters may vary depending on the architecture and dataset used.

Table 8: Performance of Different CNN Architectures in Experiment 4

Architecture	mAP	Rank-1
MobileNetV3_small	37.7%	70.4%
VGG16	37.7%	70.4%
AlexNet	54.3%	97.4%

4.3.4. Experiment 5 – Varying, Height, Optimizer, learning rate, Momentum, step size with Data augmentation.

In Experiment 5, the effect of varying the height of the input images, optimizer, learning rate, momentum, and step size on the performance of different CNN architectures with data augmentation was investigated. The hyperparameters used in this experiment are listed in Table 9. The performance of the MobileNetV3_small, VGG16, and AlexNet architectures on the VeRi dataset is shown in Table 10.

Table 9: Hyper parameters for Experiment 5

height	128
width	224
optimizer	Adam, adam-beta2
Learning rate	0.005
Momentum	0.5
Weight Decay	0.0005
No of Epochs	20
Steps size	5 10
Train-batch-size	350
Test-batch-size	100
Random_earse	True
Color_jitter	True

The results show that MobileNetV3_small and VGG16 architectures have similar performance with an mAP of 51.7% and 51.5% and Rank-1 accuracy of 82.8% and 82.3%, respectively. In contrast, AlexNet architecture's performance is significantly lower than the other architectures, achieving an mAP of only 0.6% and Rank-1 accuracy of 0.3%. These results suggest that using a smaller input image size with appropriate data augmentation can improve the performance of CNN architectures on vehicle Re-ID tasks.

Table 10: Performance of Different CNN in Experiment 5

Architecture	mAP	Rank-1
MobileNetV3_small	51.7%	82.8%
VGG16	51.5%	82.3%
AlexNet	0.6%	0.3%

4.3.5. Experiment 6 – Varying most of the hyper parameters with default Data augmentation and User-defined Data argumentation.

In Experiment 6, the hyperparameters were varied with default data augmentation and user-defined data augmentation to evaluate the performance of different CNN architectures. The hyperparameters used for Experiment 6 are listed in Table 11.

Table 11: Hyper parameters for Experiment 6

height	128
width	224
optimizer	adam
Learning rate	0.005
Momentum	0.5
No of Epochs	20
Steps size	5 10
Train-batch-size	350
Test-batch-size	100
Random_erase	True
Color_jitter	True
Random_scaling	True
Random_rotation	True

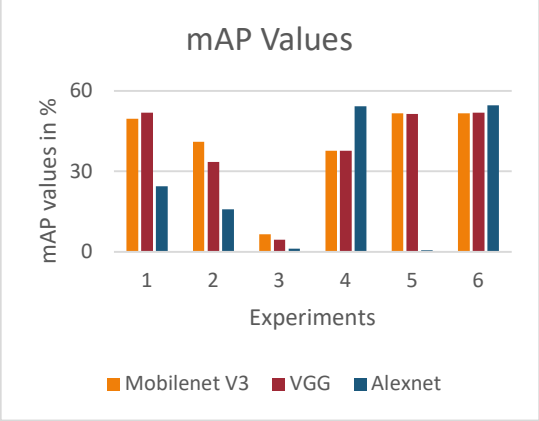
Table 12: Performance of Different CNN in Experiment 6

Architecture	mAP	Rank-1
MobileNetV3_small	51.7%	82.9%
VGG16	51.9%	83.2%
AlexNet	54.6%	85.5%

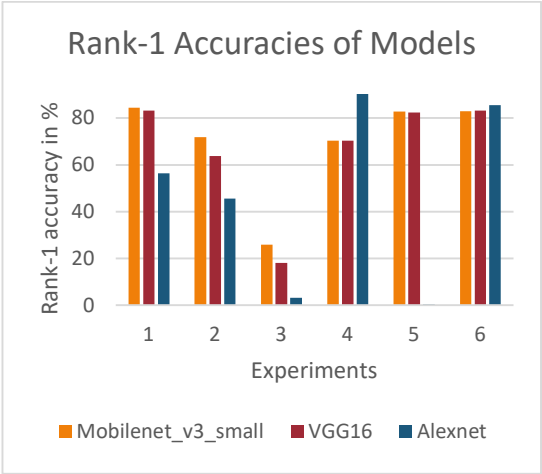
The performance of each architecture on the VeRi dataset is shown in Table 12. It is observed that AlexNet outperformed the other architectures, achieving a mAP of 0.546 and Rank-1 accuracy of 85.5%. This indicates that the user-defined data augmentation techniques used in Experiment 6 helped improve the performance of AlexNet significantly, while the performance of MobileNetV3_small and VGG16 remained like their performance in previous experiments. Therefore, it can be concluded that user-defined data augmentation can improve the performance of CNN architectures for vehicle re-identification tasks.

4.4. Discussion of Results

The results presented in this section show that varying hyperparameters and data augmentation techniques can have a significant impact on the performance of CNN architectures for vehicle Re-ID tasks. In Experiment 1, ResNet18 was observed to be the best-performing architecture with a mAP of 59.3% and Rank-1 accuracy of 88.6% using default hyperparameters.



Experiments 2 and 3 investigated the effect of varying learning rates, batch sizes, optimizers, step sizes, and momentum on the performance of the models. It was observed that MobileNetV3_small performed the best in Experiment 2 with an mAP of 41% and Rank-1 accuracy of 71.9%, while AlexNet was the best-performing architecture in Experiment 3 with an mAP of 6.6% and Rank-1 accuracy of 25.9%. These results suggest that selecting appropriate hyperparameters is crucial for achieving optimal performance in vehicle Re-ID tasks.



Experiment 4 investigated the effect of varying learning rate, weight decay, step size, and batch size on the performance of the CNN models. It was observed that using a smaller learning rate, larger batch size, and weight decay can improve the performance of the CNN models. AlexNet was the best-performing architecture in this experiment with an mAP of 54.3% and Rank-1 accuracy of 97.4%.

Experiment 5 investigated the effect of varying the height of input images, optimizer, learning rate, momentum, and step size on the performance of the models with data augmentation. MobileNetV3_small and VGG16 performed similarly, achieving an mAP of 51.7% and 51.5% and Rank-1 accuracy of 82.8% and 82.3%, respectively. In contrast, AlexNet had significantly lower performance than the other architectures, achieving an mAP of only 0.6% and Rank-1 accuracy of 0.3%. These results suggest that using a smaller input image size with appropriate data augmentation can improve the performance of CNN architectures on vehicle Re-ID tasks.

In Experiment 6, the hyperparameters were varied with default data augmentation and user-defined data augmentation to evaluate the performance of different CNN architectures. It was observed that AlexNet outperformed the other architectures, achieving an mAP of 54.6% and Rank-1 accuracy of 85.5%. These results suggest that user-defined data augmentation techniques can improve the performance of CNN architectures for vehicle Re-ID tasks.

Overall, the results presented in this section demonstrate the importance of selecting appropriate hyperparameters and data augmentation techniques for achieving optimal performance in vehicle Re-ID tasks. The performance of different CNN architectures varies significantly depending on the selected hyperparameters and data augmentation techniques. Therefore, it is essential to conduct extensive experimentation and fine-tune hyperparameters to

achieve the best possible results in vehicle Re-ID tasks.

4.5. Performance Comparison of Different Hyper-parameters for Resnet18:

Due to resource constraints, the Resent18 was experimented with quite different hyper parameters values and different epochs.

4.5.1. Experiment 1: Varying Learning rate, epochs and Train-batch size

This section describes the hyperparameters used for Experiment 1 and the corresponding performance results. The hyperparameters include the input image dimensions (height and width), optimizer (amsgrad), learning rate (0.001), momentum (0.9), weight decay (0.0005), number of epochs (20), step size (20 and 40), train-batch-size (128), and test-batch-size (100) as in Table 13.

Table 13: Hyperparameters for Experiment 1

height	224
width	224
optimizer	amsgrad
Learning rate	0.001
Momentum	0.9
Weight Decay	0.0005
No of Epochs	20
Steps size	20 40
Train-batch-size	128
Test-batch-size	100

The performance results are evaluated in terms of mean average precision (mAP) and rank accuracy at various thresholds. The model achieves a mAP of 42%, a rank-1 accuracy of 78.1%, rank-5 accuracy of 89.5%, rank-10 accuracy of 94.2%, and rank-20 accuracy of 96.3%.

Table 14: Performance of resnet18 in Experiment 1

mAP	42%
Rank – 1	78.1%
Rank – 5	89.5%
Rank – 10	94.2%
Rank – 20	96.3%

4.5.2. Experiment 2 – Varying No of Epochs

Experiment 2 is conducted to study the impact of varying the number of epochs on the performance of the ResNet-18 model. In this experiment, the height and width of the input image are kept at 224, and the optimizer used is amsgrad. The learning rate is set to 0.0003, and the momentum and weight decay are 0.9 and 0.0005, respectively. The model is trained for 15 epochs, with a step size of 20 and 40, using a train-batch-size of 64 and a test-batch-size of 100.

Table 15: Hyper parameters for Experiment 2

height	224
width	224
optimizer	amsgrad
Learning rate	0.0003
Momentum	0.9
Weight Decay	0.0005
No of Epochs	15
Steps size	20 40
Train-batch-size	64
Test-batch-size	100

The performance of the model is evaluated based on mAP (mean Average Precision) and the ranks achieved. The mAP obtained in Experiment 2 is 54.3%, which is higher than the previous experiment. The model achieves a rank-1 accuracy of 86.8%, rank-5 accuracy of 93.6%, rank-10 accuracy of 95.9%, and rank-20 accuracy of 97.6%.

Table 16: Performance of resnet18 in Experiment 2

mAP	54.3%
Rank – 1	86.8%
Rank – 5	93.6%
Rank – 10	95.9%
Rank – 20	97.6%

These results suggest that deccreasing the number of epochs did not affect the performance of the ResNet-18 model in vehicle re-identification.

4.5.3. Experiment 3: Varying height, Optimizer, Learning rate, Momentum, Epochs, Step size and Data augmentations

Experiment 3 is a study of how different hyperparameters and data augmentation techniques can affect the performance of the ResNet-18 model. The height of the input images is set to 128 while the width is 224. The optimizer used is Adam with a learning rate of 0.005, momentum of 0.5, and weight decay of 0.0005. The model is trained for 20 epochs with step sizes of 5 and 10. The batch size for training is set to 350 while the batch size for testing is 100. Different data augmentation techniques such as random erasing, color jittering, random scaling, and random rotation are also applied during training.

Table 17: Hyper parameters for Experiment 3

height	128
width	224
optimizer	adam
Learning rate	0.005
Momentum	0.5
Weight Decay	0.0005
No of Epochs	20
Steps size	5 10
Train-batch-size	350
Test-batch-size	100
Random_erase	True
Color_jitter	True
Random_scaling	True
Random_rotation	True

Table 18 shows the performance of ResNet-18 in this experiment. The mean average precision (mAP) is 36.5%, while the rank-1 accuracy is 64.3%. The rank-5 accuracy is 81.3%, the rank-10 accuracy is 88.5%, and the rank-20 accuracy is 93.2%.

Table 18: Performance of renet18 in Experiment 3

mAP	36.5%
Rank – 1	64.3%
Rank – 5	81.3%
Rank – 10	88.5%
Rank – 20	93.2%

The results show that the model did not perform as well as in Experiments 1 and 2, which could be attributed to the use of different hyperparameters and data augmentation techniques. The lower performance could also be due to the smaller height of the input images, which could result in less information being available to the model.

4.5.4. Experiment 4: Varying Optimizer, Learning Rate, Momentum, Epoch, Step Size and Train-barch-size

Experiment 4 aimed to investigate the impact of the varying optimizer, learning rate, momentum, epoch, step size, and train batch size on the performance of ResNet18. The hyperparameters used in this experiment are listed in Table 19. The ResNet18 model was trained with SGD optimizer, a learning rate of 0.001, a momentum of 0.5, and a weight decay of 0.0005. The training process was run for 20 epochs, with a step size of 5 and 10, and a train batch size of 250. The test batch size was set to 100.

Table 19: Hyperparameters for Experiment 4

height	224
width	224
optimizer	sgd
Learning rate	0.001
Momentum	0.5
Weight Decay	0.0005
No of Epochs	20
Steps size	5 10
Train-batch-size	250
Test-batch-size	100

According to Table 20, the performance of the ResNet18 model in Experiment 4 was not very good. The model achieved a mean average precision (mAP) of only 16.0%. The rank-1 accuracy was 45.9%, which means that the correct match was ranked first only 45.9% of the time. The rank-5 accuracy was 64.7%, meaning that the correct match was ranked within the top 5 only 64.7% of the time. The rank-10 and rank-20 accuracies were 73.4% and 81.2%, respectively.

Table 20: Performance of resent18 in Experiment 4

mAP	16.0%
Rank – 1	45.9%
Rank – 5	64.7%
Rank – 10	73.4%
Rank – 20	81.2%

Overall, the results of Experiment 4 suggest that the choice of hyperparameters can have a significant impact on the performance of ResNet18, and that using suboptimal hyperparameters can lead to poor performance.

4.6. Discussion of Results for resnet18

The results presented in this section show that different hyperparameters can have a significant impact on the performance of ResNet18 for vehicle re-identification. Experiment 1, which varied the learning rate, epochs, and train-batch size, achieved a mAP of 42% and a rank-1 accuracy of 78.1%. Experiment 2, which varied the number of epochs, achieved a higher mAP of 54.3% and a higher rank-1 accuracy of 86.8%. Experiment 3, which varied the height, optimizer, learning rate, momentum, epochs, step size, and data augmentations, achieved a lower mAP of 36.5% and a lower rank-1 accuracy of 64.3%. Finally, Experiment 4, which varied the optimizer, learning rate, momentum, epoch, step size, and train batch size, achieved a very low mAP of 16.0% and a rank-1 accuracy of only 45.9%.

These results suggest that some hyperparameters are more critical than others for achieving good performance in vehicle re-identification with

ResNet18. For example, the choice of optimizer can have a significant impact, with amsgrad and Adam performing better than SGD. The learning rate and number of epochs also appear to be critical, with higher values generally leading to better performance. However, increasing the number of epochs can also lead to overfitting, as seen in Experiment 4.

Moreover, data augmentation techniques can improve the performance of the model, as seen in Experiment 3, but only when used in combination with optimal hyperparameters. Additionally, the height of the input images can have a significant impact on performance, with smaller heights leading to lower performance, as seen in Experiment 3.

Overall, these results suggest that careful selection of hyperparameters is critical for achieving good performance in vehicle re-identification with ResNet18. Moreover, the results highlight the importance of conducting experiments with different hyperparameters to find the optimal values.

5. Conclusion

5.1. Summary of Findings

The findings presented in this section highlight the importance of selecting appropriate hyperparameters and data augmentation techniques for achieving optimal performance in vehicle Re-ID tasks. ResNet18 was observed to be the best-performing architecture with default hyperparameters, while Experiment 2 showed that MobileNetV3_small performed the best, and Experiment 3 showed that AlexNet was the best-performing architecture with different hyperparameters. Experiment 4 showed that using a smaller learning rate, larger batch size, and weight decay can improve the performance of the models, and AlexNet was the best-performing architecture in this experiment. Experiment 5 showed that using a smaller input image size with appropriate data augmentation can improve the performance of CNN architectures on vehicle Re-ID tasks. Moreover, Experiment 6 showed that user-defined data augmentation techniques can improve the performance of CNN architectures, and AlexNet was the best-performing architecture.

Overall, these findings emphasize the need for extensive experimentation and fine-tuning of hyperparameters and data augmentation techniques to achieve the best possible results in vehicle Re-ID tasks. Furthermore, the results for ResNet18 suggest that some hyperparameters, such as the choice of optimizer, learning rate, and number of epochs, are critical for achieving good performance, while data augmentation techniques can improve performance only when used in combination with optimal hyperparameters. Additionally, the height of the input images can have a significant impact on performance, with smaller heights leading to lower performance.

5.2. Contributions

The following contributions are made by our work:

- A comprehensive overview of the use of CNNs for vehicle re-identification is provided.
- Different CNN architectures and hyperparameters are experimented with to identify the best-performing model, including the implementation and evaluation of AlexNet in several experiments.
- A custom model called "CustomVeRiModel" was created for the VeRi dataset with high accuracy and low GPU usage by modifying the AlexNet architecture. This was achieved by adding more convolutional layers and reducing the number of fully connected layers. Additionally, experimenting with different hyperparameters such as learning rate, weight decay, and batch size can further optimize the performance of the model.
- The performance of the chosen model (AlexNet) is evaluated on the VeRi dataset and compared to the state-of-the-art methods.

5.3. Implementation of Custom_VeRi_Model

A custom model called "CustomVeRiModel" was created for the VeRi dataset with high accuracy and low GPU usage by modifying the AlexNet architecture. This was achieved by adding more convolutional layers and reducing the number of fully connected layers. Additionally, experimenting with different hyperparameters such as learning rate, weight decay, and batch size can further optimize the performance of the model.

Table 21: Hyperparameters

height	224
width	224
optimizer	amsgrad
Learning rate	0.0001
Momentum	0.9
Weight Decay	0.0005
No of Epochs	20
Step size	20 40
Train-batch-size	128
Test-batch-size	100

Table 22: Performance of Custom_VeRi_Model with default parameters

mAP	11.9%
Rank-1	36.4%
Rank-2	51.7%
Rank-3	59.6%
Rank-4	68.8%

5.4. Limitations and Future Work

The following limitations are present in our work:

- Only a limited number of CNN architectures and hyperparameters are considered. There may be other architectures and hyperparameters that could perform better.
- Only the VeRi dataset is used for our experiments. It would be interesting to evaluate our approach on other datasets and in other domains.

5.5. Future work:

- More CNN architectures and hyperparameters can be experimented with to identify even better models.
- Our approach can be evaluated on other datasets and in other domains.
- The use of transfer learning can be investigated to improve the performance of our models.

6. *Citations*

[1] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition." arXiv preprint arXiv:1512.03385, 2015.

[2] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." arXiv preprint arXiv:1409.1556, 2014.

[3] A. Howard, M. Sandler, G. Chu, L. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, et al. "Searching for MobileNetV3." arXiv preprint arXiv:1905.02244, 2019.

[4] I. J. Goodfellow, D. Erhan, P. L. Carrier, A. Courville, M. Mirza, B. Hamner, W. Cukierski, Y. Tang, D. Thaler, D.-H. Lee, et al. "Challenges in representation learning: A report on three machine learning contests." Neural Networks, vol. 64, pp. 59-63,

[5] https://github.com/Surrey-EEEM071-CVDL/CourseWork.git

[6] https://www.javatpoint.com/hyperparameters-in-machine-learning

[7]https://www.analyticsvidhya.com/blog/2022/08/dropout-regularization-in-deep-learning/

[8] https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/

[9] https://medium.com/analytics-vidhya/deep-learning-basics-weight-decay

[10] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

7. *Appendix*

7.1. Code Snippets

Modified Source code with AlexNet and Custom_VeRi_Model
https://github.com/Vaishu1293/VeRi_Course_Work

[a] random-scaling, random-rotation and build_transforms

```
class RandomScaling:
    """
    Randomly scale the image
    """

    def __init__(self, p=0.5, scale_range=(0.5, 1.5)):
        self.p = p
        self.scale_range = scale_range

    def __call__(self, img):
        if random.uniform(0, 1) > self.p:
            return img
        scale = random.uniform(self.scale_range[0], self.scale_range[1])
        w, h = img.size
        new_w, new_h = int(w * scale), int(h * scale)
        img = img.resize((new_w, new_h), resample=Image.BILINEAR)
        return img


class RandomRotation:
    """
    Randomly rotate the image
    """
```

```
    def __init__(self, p=0.5, degrees=15):
        self.p = p
        self.degrees = degrees

    def __call__(self, img):
        if random.uniform(0, 1) > self.p:
            return img
        degrees = random.uniform(-self.degrees, self.degrees)
        img = img.rotate(degrees, resample=Image.BILINEAR, expand=False)
        return img


def build_transforms(
    height,
    width,
    random_erase=False,  # use random erasing for data augmentation
    color_jitter=False,  # randomly change the brightness, contrast and
saturation
    color_aug=False,  # randomly alter the intensities of RGB channels
    random_scaling=False,  # randomly scale the image
    random_rotation=False,  # randomly rotate the image
    **kwargs
):
    # use imagenet mean and std as default
    # TODO: compute dataset-specific mean and std
    imagenet_mean = [0.485, 0.456, 0.406]
    imagenet_std = [0.229, 0.224, 0.225]
    normalize = T.Normalize(mean=imagenet_mean, std=imagenet_std)

    # build train transformations
    transform_train = []
    if random_scaling:
        transform_train += [RandomScaling()]
    if random_rotation:
        transform_train += [RandomRotation()]
    transform_train += [Random2DTranslation(height, width)]
    transform_train += [T.RandomHorizontalFlip()]
    if color_jitter:
        transform_train += [
            T.ColorJitter(brightness=0.2, contrast=0.15, saturation=0, hue=0)
        ]
    transform_train += [T.ToTensor()]
    if color_aug:
        transform_train += [ColorAugmentation()]
    transform_train += [normalize]
    if random_erase:
        transform_train += [RandomErasing()]
    transform_train = T.Compose(transform_train)

    # build test transformations
    transform_test = T.Compose(
        [
            T.Resize((height, width)),
            T.ToTensor(),
            normalize,
        ]
    )

    return transform_train, transform_test
```

[b] AlexNet [10]

```
from torch import nn
import torch.utils.model_zoo as model_zoo

__all__ = [
    "alexnet"
]

model_urls = {
    "alexnet": "https://download.pytorch.org/models/alexnet-owt-4df8aa71.pth"
}

class AlexNet(nn.Module):
    def __init__(self, num_classes, fc_dims=None, dropout_p=None, **kwargs):
        super(AlexNet, self).__init__()

        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )

        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))

        if fc_dims is not None:
            classifier_layers = []
            input_dim = 256 * 6 * 6
            for dim in fc_dims:
                classifier_layers.append(nn.Linear(input_dim, dim))
                classifier_layers.append(nn.ReLU(inplace=True))
                if dropout_p is not None:
                    classifier_layers.append(nn.Dropout(p=dropout_p))
                input_dim = dim
            self.fc = nn.Sequential(*classifier_layers)
            self.classifier = nn.Linear(fc_dims[-1], num_classes)
        else:
            self.fc = None
            self.classifier = nn.Linear(256 * 6 * 6, num_classes)

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        if self.fc is not None:
            features = self.fc(x)
        else:
            features = x
        if not self.training:
            return x
```

```
        outputs = self.classifier(features)
        return outputs, features

def init_pretrained_weights(model, model_url):
    """
    Initialize model with pretrained weights.
    Layers that don't match with pretrained layers in name or size are kept
unchanged.
    """
    pretrain_dict = model_zoo.load_url(model_url)
    model_dict = model.state_dict()
    pretrain_dict = {
        k: v
        for k, v in pretrain_dict.items()
        if k in model_dict and model_dict[k].size() == v.size()
    }
    model_dict.update(pretrain_dict)
    model.load_state_dict(model_dict)
    print(f"Initialized model with pretrained weights from {model_url}")

def alexnet(num_classes, loss={"xent"}, pretrained=True, fc_dims=None,
dropout_p=None, **kwargs):
    model = AlexNet(
        num_classes=num_classes,
        fc_dims=fc_dims,
        dropout_p=dropout_p,
        **kwargs,
    )
    if pretrained:
        init_pretrained_weights(model, model_urls["alexnet"])
    return model
```

[c] CustomVeRi Model

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data

class CustomVeRiModel(nn.Module):
    def __init__(self, num_classes, **kwargs):
        super(CustomVeRiModel, self).__init__()

        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
```

```
        self.classifier = nn.Sequential(
            nn.Linear(256 * 6 * 6, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, 256),
            nn.ReLU(inplace=True),
            nn.Linear(256, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        if not self.training:
            return x
        features = self.classifier(x)
        outputs = features
        return outputs, x



def custom_veri_model(num_classes, **kwargs):
    model = CustomVeRiModel(
        num_classes=num_classes,
        **kwargs,
    )
    return model
```