Robotics Research Lab
Department of Computer Science
Rheinland-Pfälzische Technische Universität

Master Thesis

Reinforcement Learning based Pretraining for Autonomous Bus Operation

Vedaant Joshi

January 17, 2024

Master Thesis

Reinforcement Learning based Pretraining for Autonomous Bus Operation

Robotics Research Lab
Department of Computer Science
Rheinland-Pfälzische Technische Universität

Vedaant Joshi

Day of issue : 17.07.2023
Day of release : 17.01.2023

First Reviewer : Prof. Dr. Karsten Berns
Second Reviewer : Jakub Pawlak, M. Sc.
Supervisor : Jakub Pawlak, M. Sc.

Hereby I declare that I have self-dependently composed the Master Thesis at hand. The sources and additives used have been marked in the text and are exhaustively given in the bibliography.

January 17, 2024 – Kaiserslautern

(Vedaant Joshi)

Abstract

The object detection algorithm have improved a lot in the span of last two decades, majorly in last decade. The challenges faced by the early detectors that utilized handcrafted features were that their performance was subpar and have lacked the precision to be used in critical tasks as it increased the complexity. In the last decade, this has been improved by the introduction of Convolutional Neural Networks (CNN) in the field of Object Detection. The basic approach of CNN for detecting objects is by learning the features present in the image. The CNN architecture is a deep neural network architecture, where different features of the images are learnt at different hidden layers. In addition to learning the features from the images, this master thesis aims to inculcate the behavioral knowledge into the object detection model and evaluate if the model performs better. In order to achieve this, the behavioral knowledge is acquired using Reinforcement Learning (RL) algorithm. In this project, Deep Reinforcement Learning is used, which comprises of neural networks that learn to take decisions that are favorable in those complex environments. Here the goal of the RL algorithm is to reach a goal state and maximize the rewards. To obtain the behavioral knowledge, the aim is to learn to drive a vehicle autonomously in presence of pedestrian in a simulated environment and then use this trained network as a backbone in an object detection task. The RL agent is trained to drive in this conditions on the basis of rewards and penalties received while trying to navigate through the environment. It learns by trial and error method. Different parameters such as distance from midline, distance to nearest pedestrian, speed, and angle with the road are taken into consideration for designing the reward function. After the training of the RL agent, it learns to navigate through the environment in presence of the pedestrians and this network learns the behavior of the agent as well as the pedestrian around it. The aim is to transfer this knowledge to object detection backbone and train the model for detecting pedestrians in same environment.

Contents

1. Introduction

Intelligent machines have significantly eased the lives of human beings, and each year witnesses rapid advancements in research. Many mundane tasks of human beings have been replaced by machines or intelligent systems, which can complete these tasks with high accuracy. For a machine or system to integrate well within the human environment and potentially replace humans for certain tasks, it is necessary for the system to perceive visual data from the surroundings and process it to make informed decisions. Some of the vision-based tasks that take visual data as input and provide useful information includes object detection, segmentation, optical character recognition, object tracking, and many other tasks. Since 2014, computer vision has advanced considerably due to the introduction of deep learning concepts in the field. However, there are still many challenges faced by this technique in today's age for vision tasks. Autonomous driving is one application that utilizes computer vision to gain knowledge about the surroundings and make decisions accordingly. Different sensors and algorithms are employed to navigate the vehicle without any human guidance.

The application of object detection and image segmentation tasks have improved a lot since introduction of deep learning. It is necessary to see how far computer vision have come. Before the advent of deep neural networks in this field, the earlier algorithms used handcrafted features for object detection tasks. One of the human-face detectors developed in 2001 is known as Viola Jones Detectors [Viola 01]. It was used for detecting human faces in real-time. This technique used the concept of sliding windows and used handcrafted features. Then in 2005, the HOG detector [Dalal 05] was proposed which was famous for pedestrian detection. In year 2008, DPM (deformable part-based model) [Felzenszwalb 10] was introduced which was an extension of HOG detector. However, object detection reached a plateau as the performance of these models, using hand-crafted features, didn't improve after a certain point of time. Until the inception of Convolutional Neural Networks (CNN). CNN proved to be the building blocks for the modern object detection frameworks. In the year 2014, RCNN [Girshick 14] was introduced for object detection. Some of the object detection architectures includes SPPNet [He 14], Fast RCNN [Girshick 15], Faster RCNN [Ren 16], YOLO [Redmon 15] and many more. Aforementioned methodologies are used to perform vision-based task and identify distinct features present in the images. In

object detection, the primary pipeline involves extracting features from the image and forwarding those complex features to a object detection module to detect it. [Borah 20]

1.1 Motivation

In any traditional object detection architecture, the major concept used is to extract the features of the object and train the neural network based on those features. This enables the network to identify those features/objects with high accuracy when unseen images are presented during the test phase. The detection of an object is purely based on the features of that particular object—how it looks—and not on how the object interacts with its environment. The main goal of this thesis is to explore a different approach to object detection. Similar to previous methods, features are extracted from the images using a CNN. However, in addition to that, behavioral information is provided to the network, and an analysis is conducted to verify if the additional behavioral information improves the overall accuracy of the object detection network. The question may arise: why add behavioral knowledge? This additional knowledge might help identify objects that haven't been seen during training. In addition to the features of the object, understanding how it interacts in the environment is also important. This additional information about the behavior of the object might help identify instances of the same object with different features. One drawback in current object detectors is that if a particular variation of the object is not seen during training, it becomes difficult for the model to detect it during inference. So, in addition to feature data, the behavior of the object should also be captured to increase the accuracy of detection.

In this thesis, the behavior of an autonomous vehicle in the presence of pedestrians on the road is considered as the source of additional information for the network to learn from. The first step is to train an autonomous vehicle in a simulation environment (Section 4.1), for which a Reinforcement Learning algorithm (Section 2.4) is employed. This algorithm trains the vehicle to navigate in the environment, encapsulating the behavior of the vehicle in a setting where pedestrians are spawned. The primary goal of the vehicle is to learn to navigate safely in such an environment. The acquired knowledge is then transferred to the object detection network, which has the final task of detecting pedestrians in an urban city environment.

1.2 Outline

The report is structured as follows: In Section 2, we dive into the essential technical background knowledge necessary to understand the thesis objectives. Section 3 offers a brief overview of the prior methods used in for downstream tasks such as object detection and how these methods relate to the work in this thesis. The Section 4 provides an introduction to the building blocks of this projects. All the necessary tools and components used in this project are introduced in this chapter. Further, in the next Section 5 outlines the methodology applied in this thesis. The tools previously introduced are used in this chapter and it elaborates the methods and techniques used for training the Deep Neural network. Section 6 presents a comprehensive examination of the experiments conducted during this research and the resultant findings. Experiments are undertaken under different conditions and the results are compared with other base models. Finally, Section 7 engages

in a detailed discussion of the outcomes derived from analyzing the experimental data, culminating in the conclusion. It also discusses the findings of this research work and lists the future development possibilities in this area on the basis of the knowledge gained from the results and experiments conducted during this research work.

2. Technical Background

In this chapter, we present a comprehensive study of the concepts of Artificial Intelligence (AI), Machine Learning (ML), and Deep Learning (DL). The content covered under this chapter will serve as the essential concepts necessary for understanding the implementation of the Reinforcement Learning algorithm and Object Detection methodologies.

The term "Artificial Intelligence" was coined by John McCarthy [J. McCarthy 55], an American computer scientist. He is considered one of the ground-breakers in the field of AI. Since then, many advancements have occurred in the field of AI. The aim of AI is to exhibit functions such as problem-solving, decision making, language understanding, perception and many more. The primary objective is to learn from data and make informed decisions. Generally, a human supervision is required to check if the decision taken by the system are in accordance, but some AI systems have been designed to work without human supervision. An example of this is found in Autonomous Driving Systems, where the system uses various sensors to gather data from the environment and make decisions with the aim of fulfilling the goal while following all the safety measures. In subsequent subsections, we will look into subsets of AI. The relation between AI, Machine Learning, Neural Networks and Deep learning is shown in Figure 2.1

2.1 Machine Learning

Machine Learning (ML) is a sub-field of Artificial Intelligence with numerous application across different domains, but for every application the underlying concept which ML systems uses is to enhance the performance of the system as more data is given to it overtime. According to Tom Mitchell, the general definition of Machine Learning is articulated as follows: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measure by P, improves with experience E" [Mitchell 97]. There are different categories of Machine Learning, which are divided into groups based on the type of dataset provided during the training. When a dataset consists of both the data point, represented by its selected features, and the corresponding label, which assigns a category to that data point, this method is called as Supervised Learning. Few notable types of Supervised

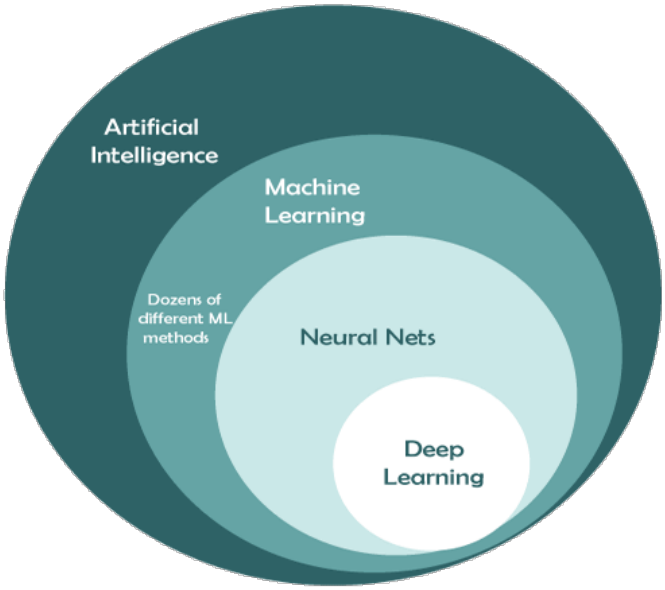


Figure 2.1: Relation between AI, Machine Learning, Neural Networks and Deep Learning. [javatpoint 21]

Learning are Regression, Classification, Naive Bayes, Neural Networks, and Random Forest. In Unsupervised Machine Learning, the data is unlabelled. The algorithm analyzes the data points to gain insight from the hidden features and clusters the similar data points together. K-means clustering, K-Nearest Neighbour, Principal Component Analysis and Singular Value Decomposition fall under the category of Unsupervised Machine Learning algorithms [Naeem 23]. Combining both of the aforementioned methods, third ML method is known as Semi-Supervised Learning, where the dataset contains of small amount of labelled data points and more number of unlabelled data points.This particular method is proved to be useful when it is difficult to obtain enough labelled data to train the algorithm, so both unlabelled and labelled data points are used. Finally, the last category of ML which will be primary focus of this read is Reinforcement Learning(RL). RL is reward-based algorithm that learns to interact with the environment iteratively, and is rewarded positive or negative rewards based on the actions/decisions taken by the systems. Detailed introduction to all the types of ML algorithm are explained in the following subsections.

2.1.1 Supervised Learning

Supervised Machine Learning (SML) algorithms are given labelled data as input. It is called as supervised, as explicitly the category of each data point is available and while training it iteratively, the labels are treated as the ground truth to verify if the ML system has accurately categorized the data point. In accordance of the ground truth and the predicted output, the parameters of the network are updated in such a way that the accuracy of the system increases over time. The assessment of the system is done during the testing phase, by feeding previously unseen data points (never used during training phase). Further, Supervised Learning is divided into two categories. This is based on the type of the task the system is designed to perform [Ahmed 21]. In Figure 2.2, supervised

learning plot shows that the data points have labels (red and green color depicts unique classes), and the task of the SML algorithm is to find the most effective line that separate the classes, allowing it to classify new, unseen data points accurately.

Regression

In Regression based Supervised Learning, the parameters of the model are updated in such a way that it finds a best line fitting all the data points. Fitting the data points means to minimize the distance between the line and each individual data point. Achieving this, results into making accurate predictions for future unseen instances. Here, the labels are continuous values. For instance, predicting Monthly Sales based on input features such as money spend on advertising. This allows the model to be trained using historical data, which then helps in approximating Monthly Sales for the current month and also future upcoming sales

Classification

In contrast to Regression Supervised Learning, Classification-based Supervised Learning involves discrete classes. The model's objective is to assign a single class to each data point, determined by the features of the input data point. During the training phase, the model output and ground truth labels are compared, and the model parameters are adjusted based on the differences between the labels over multiple iterations. One such example is of spam email detection. The emails can be classified into two categories : "spam" or "not spam". The input features given to the model can be certain word occurrences, fishy email address, or presence of unnecessary links.

2.1.2 Unsupervised Learning

The difference between Supervised Learning and Unsupervised Learning is absence of Labels which is visually described in Figure 2.2. Unlike supervised learning, unsupervised learning seeks to identify underlying patterns, relationships, or structures within the data without prior guidance. Some of the objectives for unsupervised learning algorithms include clustering similar data points together, reducing dimensionality, or generating new representation of data. This type of learning is usually used for understanding inherent structure of data without any prior information. One example under Unsupervised learning is Clustering. A dataset follows a pattern which needs to be identified by the algorithm and on that basis, similar points forms a group together called as Cluster. This is further explained in the upcoming Section 2.1.2.1

2.1.2.1 Clustering

In simple terms, clustering involves grouping similar data points together into clusters. As discussed in 2.1.2, data points with similar characteristics form one cluster, while those with different features or properties make up another. When a new data point is introduced to the system, it is assigned to a cluster based on its features. One commonly used clustering algorithm is the K-means algorithm [Jin 10]. This algorithm suggests that, starting with an initial clustering that is not optimal, each point is moved to its nearest new center (mean). The cluster center is then updated by calculating the mean of the members of that cluster. This process is iterated until the mean of each cluster no longer changes.

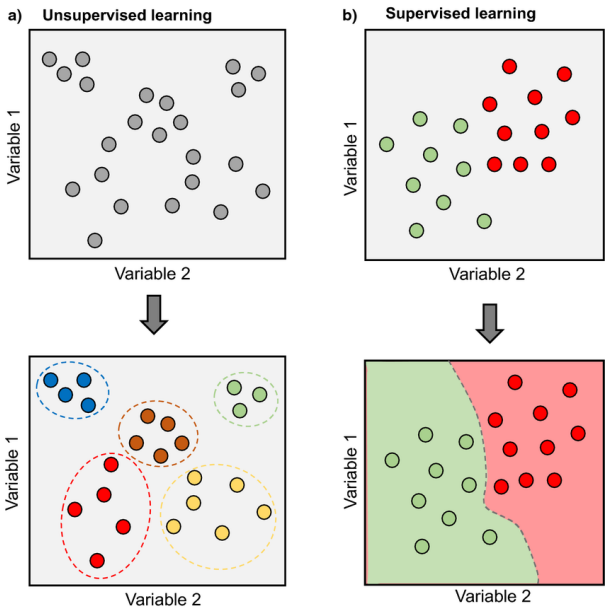


Figure 2.2: Difference between Supervised and Unsupervised Machine learning algorithms [Morimoto 21]

2.1.2.2 Dimensionality Reduction

Dimensionality Reduction (DR) is the process of removing redundant features, noisy and irrelevant data, to improve learning feature accuracy and reduce the training time [Velliangiri 19]. This techniques have been implemented using feature selection and extraction method. Principal Component Analysis (PCA) is one of the DR technique which reduces the learning process time.

2.1.2.3 Association

Association or Association Rule Mining finds interesting associations in large sets of data items [Cios 07]. One of the most used application is of Market-Basket analysis, where it analyzes customer buying patterns by finding relation between items that customers put together. This knowledge then can be used by the seller to make profits over time.

2.1.3 Semi-Supervised Learning

Semi-Supervised is a kind of learning process where the algorithm takes advantage of both labeled dataset and unlabeled dataset. Labeled dataset contains those data points where the ground truth is available and unlabeled dataset lacks explicit labels. During the training process, the labeled data guides the learning process whereas for unlabeled data points the algorithm is required to find out the underlying patterns in that data set. This method is used when it becomes very expensive to acquire labeled dataset. It takes benefits from both concepts (Supervised and Unsupervised). The training process consist of training the base model with labeled data first, like in supervised learning. Thereafter, train the previously trained base model with unlabeled data to get pseudo labels [Y C a 18]. The pseudo labels with prediction confidence higher than a give threshold are added to the pool of labeled dataset. Then the new dataset formed is used to train the model. This can

happen for several iterations and can improve the performance of the model iteratively. This can go other way round as well if the pseudo labels generated are incorrect and then model is trained on the basis of data points with incorrect labels.

2.1.4 Reinforcement Learning

Reinforcement Learning (RL) algorithms differs from the previous approaches. RL serves as a general framework for building systems that do not require human supervision to make decisions in order to perform a task [Buffet 20]. In a Reinforcement Learning system, an agent is defined as an entity that is responsible for interacting with the environment it is present in and learn to make favorable decisions. Hence, the RL agent faces sequential decision-making problem. Based on the current state, the agent has to make a decision of performing an action, which results into changes in the current state and on the basis of that a reward is provided as a feedback. In RL algorithm, the agent needs to learn to take good actions based on observations and rewards received. In-depth discussion on this topic is provided in Section 2.3.

2.2 Deep Learning

Before diving deeper into the major concepts of Deep Learning, this section explores the backbone of Deep Learning called as Neural Networks. Neural Network is a computational model which was inspired by the functioning and structuring of neurons present in human brain. The major components of a Neural Network are nodes, connection between those nodes, input layer, hidden layer(s) and finally an output layer as show in Figure 2.3. The input layer takes training and testing data points. After the initial input layer processes the input it is being forwarded to the hidden layer(s), where each layer extracts features and passes them further to other hidden layers. Every node in a layer is interconnected with nodes in previous and next layer. Before passing the features to the next layer, it is passed through Activation function. Basically, activation function adds non-linearity to the model (further discussed in Section 2.2.3). The interconnected nodes consist of weights (Figure 2.3). Those weights are trainable parameters, which are trained over the period of multiple epochs. These weights are updated in a way that the loss between ground truth and the predicted computations is minimized.

Deep Learning or also called as Deep Neural Networks is the term used for the Neural Networks with multiple layers. Increasing the layers in a Neural Network enables learning of hidden patterns present in the input data distribution. Hence, Deep Learning is able to give accurate results on various different applications. The term itself was proposed in 1986 by Rina Dechter although the history of its appearance is tricky [Fradkov 20]. The very first general working learning algorithm consisting of supervised, deep, feedforward, multilayer perceptron was published by Alexey Ivakhnenko and Lapa in 1967 [Ivakhnenko 67]. In 1989, Yann LeCun showed how to use a big back-propagation method for the first time in image recognition. He combined it with Convoltional Neural Networks to identify "handwritten" digits [LeCun 89]. Significant progress was made after that in the coming years. In 1995, Support Vector Machine was developed by Dana Cortes and Vladimir Vapnik [Cortes 95]. There are several applications of Deep Neural Networks in the field of Computer Vision and Textual-based Application. The main point in history which

can be considered as the turning point in the field of vision was the introduction of AlexNet (which uses Convolutional Neural Networks; further discussed in Section 2.2.4) [Krizhevsky 12a] by Alex K. et al. . It achieved notable success in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012, demonstrating the huge scope of using Deep Neural Networks in the field of image classification. In the field of textual-based data, various contributions have been made such as Transformers [Vaswani 23], Long-Short Term Memory (LSTM) [Hochreiter 97], Gated Recurrent Unit (GRU) [Cho 14] and currently Large Language Models (LLM) [Brown 20]. These contributions have proved to be significant. The upcoming sub-sections dive a little deeper into the main concepts and working of Deep Neural Networks.

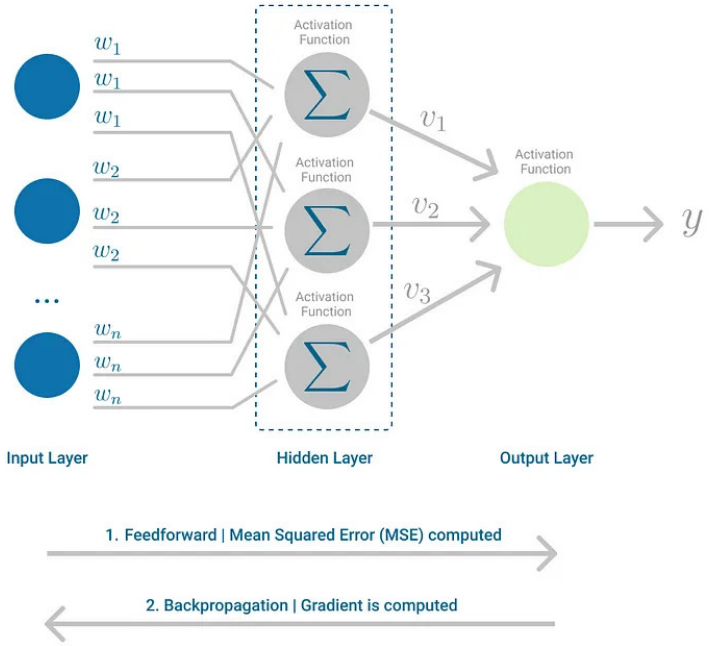


Figure 2.3: This is the basic depiction of a neural network. The input layer consists of multiple neurons. There can be several hidden layers which are stacked together to extract features from the input data. The neurons/nodes from the previous layer are connected with the neurons/nodes in the next hidden layer. The features from the last hidden layer is given as input to the output layer and makes predictions. [Bento 21]

2.2.1 Backpropagation

The most important element of Artificial Neural Network is Backpropagation which was for the very first time described in 1986 by Rumelhart [Rumelhart 86]. The way a neural network works is all dependant upon the concept of Backpropagation. This entire process consists of four steps, that is described in the following sub-sections.

Forward Propagation

As shown in 2.3, the first stage involves Forward Propagation or Feed Forward. Here, the weighted sum is computed between each node in the current layer and every node on the preceding layer, considering the weights associated with each of the connections (w_i).

Furthermore, the output is then passed through an activation function (2.2.3), chosen based on the requirements.

Loss Computation

Similarly, the input is propagated forward through all the computations across the hidden layers. After traversing all the hidden layers, the loss term is calculated, also known as Cost function. Loss term is calculated using the predicted value obtained from the output layer of the Deep Neural Network and the available ground truth (in the case of supervised learning). For example, in a regression problem, the loss term is the average sum of the squares of the difference between the predicted output and the ground truth for each data point. This metric is called Mean Squared Error (MSE). This term indicates how close the predictions are to the true values, thereby reflecting the accuracy of the Neural Network. The choice of the loss function is task dependent.

Backward Pass

The backward pass within the Backpropagation algorithm is the learning phase of the Neural network. After the forward pass and the calculation of loss, the objective of the backward pass is to update the parameters (weights and biases) of the network so that the predicted value over multiple epochs approaches the ground truth value. The amount by which the parameters needs to be updated is based on calculation of gradients and the optimization technique used. Factors such as learning rate, and momentum are responsible for deciding by what amount the parameters are updated. The different optimization algorithms are Gradient Descent (basic algorithm), that is further discussed in sub-section 2.2.2, Stochastic Gradient Descent, Mini-Batch Gradient Descent, Momentum, Nesterov Accelerated Gradient, Adagrad, AdaDelta, and Adam [Doshi 19].

2.2.2 Gradient Descent

Gradient Descent is an algorithm to optimize Neural Networks. Here, optimizing neural networks means to make updates to all the parameters in such a manner that it results in predictions that minimizes the loss function. The main objective of the algorithm is to reduce the Cost function (loss) which in turn increases the accuracy. As shown in Figure 2.4, during "n" iterations over training data the value of weights are updated on the basis of gradient calculated and the learning rate hyperparameter.

Gradients are calculated with respect to each weight present in the neural network. This gradient then decides the direction of update for that particular weight value. In equation 2.1, θ is a parameter with the current value as θ_j and after subtracting the gradient, which is calculated as $\frac{\partial J(\theta_j)}{\partial \theta_j}$, the new updated value for the parameter will be θ_{j+1}. The α in the equation is the learning rate which is described in the later paragraph in detail. The terms loss and cost function are used interchangeably but there is a slight difference. Loss function refers to error in one training example, while a cost function calculates the average error across the data points in a training set.

$$\theta_{j+1} = \theta_j - \alpha \frac{\partial J(\theta_j)}{\partial \theta_j} \tag{2.1}$$

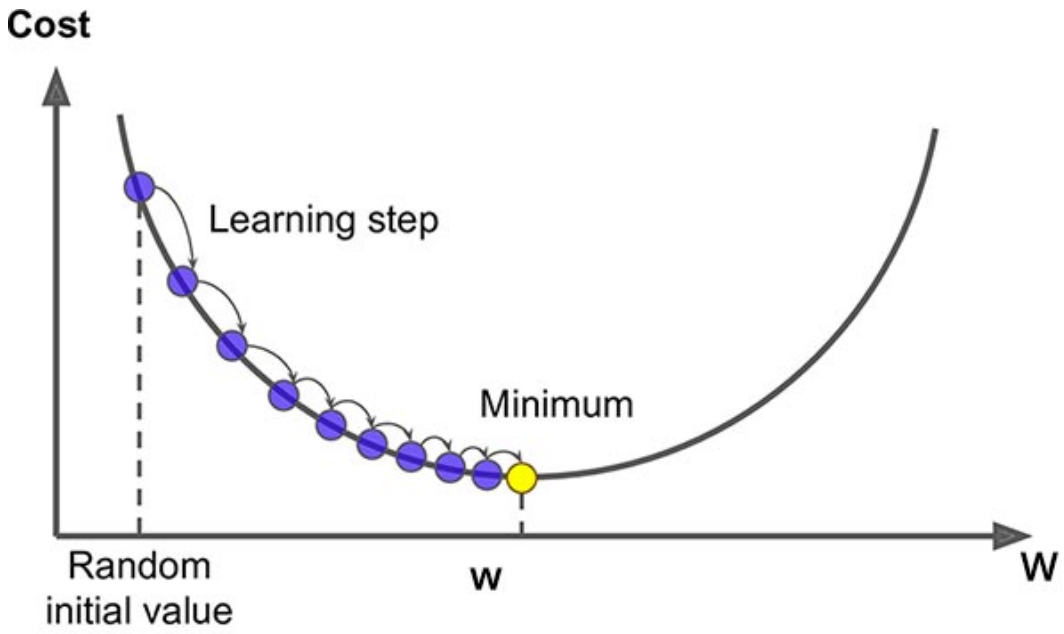


Figure 2.4: The weights are randomly initialized. Updating the value of weight(s) (w) so that after n iterations the cost function (value) achieves it minima. The weights are updated using the Gradient value calculated after each iteration.[Bhatarai 18]

Learning Rate (α)

The gradient described earlier indicates the direction in which the parameter should be adjusted to reach a local or global minima. But the learning rate is the size of the steps that are taken in that direction to reach the minimum. High value of learning rates result in bigger steps but there is a risk of overshooting the minimum. Contrarily, a low learning rate has very small step. This might take forever to reach the minimum and hence affects the overall efficiency [Sears-Collins 19]. The learning rate has been proved to be a crucial hyperparameter for the training phase of the network, and its value needs to be carefully chosen based on previous experience or through trial & error methods. The value of learning rate need not be same through out the entire training phase. Initially, the value can be set higher that allows to take larger steps towards the minima. As it starts to approach the minima, the value can be reduced to ensure that it can reach the minima precisely and not overshoot.

Gradient Descent Optimization Algorithms

There are various different optimization algorithms, some of the most commonly used are Momentum, Nesterov accelerated gradient, Adagrad, Adadelta, RMSprop, Adam, and AdaMax. The choice of optimizer in training neural networks is crucial. If the data is sparse, using adaptive learning-rate methods often yields optimal results without the need for fine-tuning. RMSprop, an extension of Adagrad, addresses issues with diminishing learning rates. It is similar to Adadelta, with the difference that Adadelta uses the RMS of parameter updates in its update rule. Adam, on the other hand, introduces bias-correction and momentum to RMSprop. Studies suggest that Adam slightly outperforms RMSprop towards the end of optimization when gradients become sparser. If fast convergence is a

priority, especially for deep or complex networks, choosing one of the adaptive learning-rate methods is recommended [Ruder 16].

Issues with Gradient Descent

There are few issues with Gradient Descent that needs to be taken into consideration while training the neural network. These issues can be the reason behind not getting the optimal performance of the Neural Network on test data.

Vanishing gradient problem occurs when the learning process slows down or stops due to the value of gradient being too small. When this happens, the weights in the neural network are updated with such a negligible amount that learning effectively stops. Thus the network doesn't reach the minima. This problem is even more noticeable with increase in the number of hidden layers, as the gradient approaches to zero when it reaches to initial layers of the network [Kaul 20].

Exploding gradient is opposite to the vanishing gradient problem. Instead of gradient value diminishing to zero it becomes too large. This large value of gradient leads to the weight value oscillating and diverging from the convergence point and hence never reaching the minima. In figure 2.4, we can see that it is reaching the minima by taking small steps, instead of that it will take large steps and move away from the desired minima [Kaul 20].

Saddle point is an issue which arises when gradient descent runs in multiple dimensions. These are defined as areas on the surface of the loss function, which from one dimension, it looks like it has reached minima, but when the same point observed from other dimension, it looks like a maxima. In Figure 2.5, the behavior depicts that it reaches a point where the slope calculated is zero, causing the algorithm to stop, assuming it has reached the minima. However, the point where the algorithm gets stuck is non-optimal [Subir Varma 18].

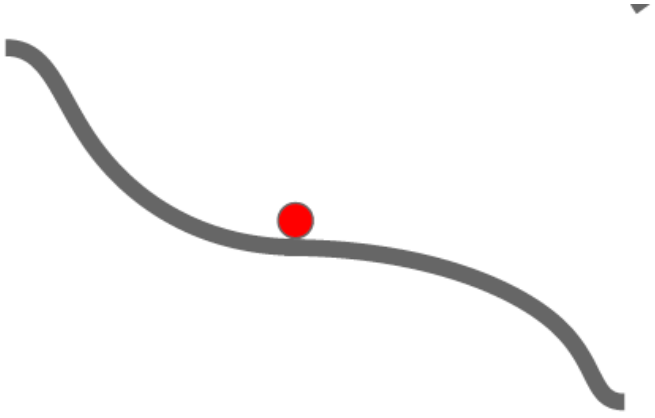


Figure 2.5: [Bhatarai 18]

2.2.3 Activation Function

At each node / neuron of the network a linear operation is being performed. For instance as shown in the equation 2.2, $\boldsymbol{x}$ is the input to the neuron, W are the weights associated with it and the b is the bias term. The output is a result of linear operation. If observed in detail, at each layer the similar functions are being applied over and over again. At the

end, the output is result of linear functions applied over linear functions, which can be computed using a single linear function. Such linear functions are only capable of working with linear vector space.

$$f(x) = W^T \boldsymbol{x} + b \tag{2.2}$$

Hence, Activation Function (AF) become crucial. The AF layers in Neural Networks introduces non-linearity to the neural network. Which tries to capture the complex non-linear patterns present in the input data. AF can learn abstract features through non-linear transformations [Dubey 20]. There are different classes of AFs namely, Logistic Sigmoid, Tanh Based, ReLU based, and ELU based [Dubey 22]. In this paper, the Soft Actor-Critic reinforcement learning algorithm is used, with further details provided in Section 2.3. The activation function used in that particular algorithm is a ReLU based algorithm. We will delve into the specifics of ReLU in the next subsection.

ReLU

ReLU is the abbreviation for Rectified Linear Unit. This function has become the most used AF, as it is easier for the model to train and most of the time it achieves good performance. In simple terms ReLU is define as, it will output the input directly if it is greater than zero, otherwise, it will output zero [Brownlee 20]. This can be observed in equation 2.3. The main advantage of ReLU is that it activates some amount of neurons based on their values; only positive valued neurons are activated. But due to this design there arises an issue, where the neurons having negative values are never activated and hence the weights associated with those neurons are never updated. Those neurons are called as dead neurons and this issues is called as Dying ReLU [Lu 20]. This results into some of the neurons never being activated and hence affects the performance.

$$f(x) = \max(0, \boldsymbol{x}) \tag{2.3}$$

2.2.4 Convolutional Neural Networks

The previous sections were related to the basics of Neural Networks and the fundamental concepts required to understand it. In this section, detailed information about Convolutional Neural Networks (CNNs) is discussed and how CNNs are used in the domain of Reinforcement Learning and Object Detection. The term CNN was coined with the design of LeNet by Yann LeCun [LeCun 98]. The architecture called as LeNet-5 was designed for recognizing handwritten digits in the year 1998. After that, the application of CNN models in the field of image analysis increased drastically after introduction of ImageNet (dataset) in the year 2010. For computer vision tasks such as image classification researchers started using ImageNet to benchmark their models. Significant improvement was achieved after Alex Krizhevsky and Geoffrey Hinton came up with CNN architecture known as the AlexNet [Krizhevsky 12b] which reduced error from 25.8% to 16.4%. In subsequent years, new architecture were developed and tested on ImageNet. Some of the architectures designed were ZFNet [Zeiler 13], VGGNet [Simonyan 14], GoogLeNet [Szegedy 14], and ResNet [He 15].

The basic CNN architecture is divided into four components. Namely, convolutional layers, pooling layers, fully connected layers and activation functions. Each component plays a crucial role for achieving high accuracy models. In Figure 2.6 there is an example. Here, the light blue grid represents the input feature map. In this particular example a simple input is represented, but it can have multiple feature maps stacked one onto another and the dimension will be larger than what is represented in the given example. The shaded area inside the light blue grid is the kernel. This kernel will slide through the input feature map. While sliding through each location of the feature map, the product between each element of the kernel and the elements overlapped in input feature map is computed and the results are summed up to obtain the output of the current location [Dumoulin 18].

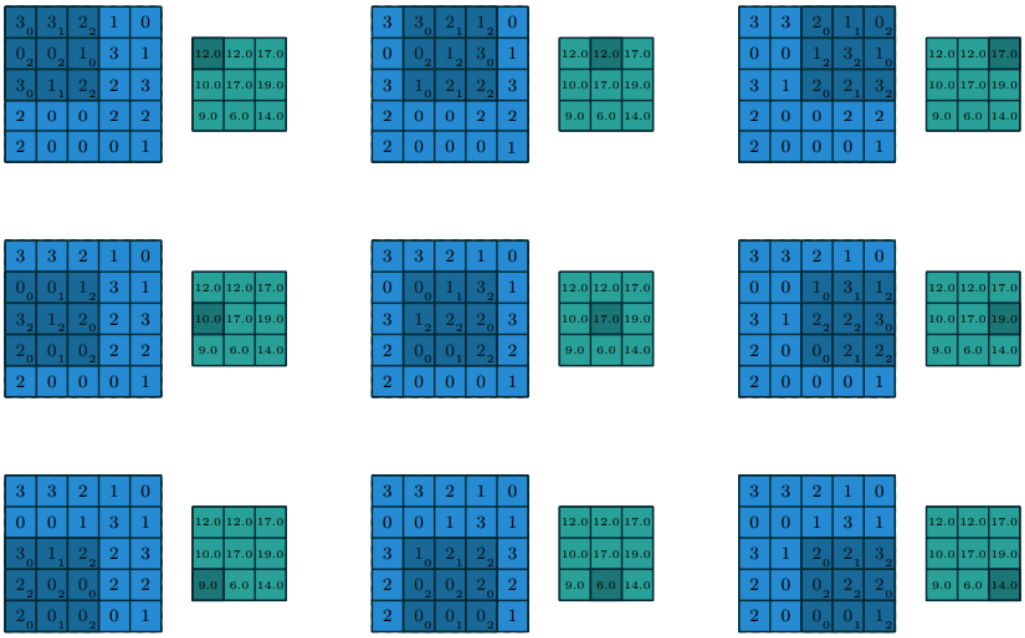


Figure 2.6: An input feature map of size 5×5 (light blue grid), that is convoluted by a 3×3 filter/kernel (dark blue grid). The final output is shown in the green color grid after convolution has taken place on entire feature map. [Dumoulin 18]

Instead of using one single kernel, multiple kernels can be employed and similar computations can be undertaken which results into generation of as many output feature maps as required. The size of the kernel is also dependent upon the dimensions/channels of the input feature maps. On the basis of the size of input feature map the shape of kernel is decided. In figure 2.6, a 2-D convolution is depicted. But this can be extended to N-D convolutions. For instance, an example of 3-D convolution is shown in Figure 2.7, where a cuboid is slid through the entire input feature map to get the output feature map. The kernel slides through the input and the objective of this computation is to extract the local features present in the input feature map and also reduce the dimensionality of the output feature map which will further act as the input to subsequent layers.

Now, to introduce non-linearity, the extracted output feature maps are passed through activation functions, for instance, Sigmoid, ReLU, Tanh, Softmax or Leaky ReLU.

Pooling layers are another essential component of CNNs used to reduce the dimensionality of feature maps. This reduction helps decrease the number of parameters and computation

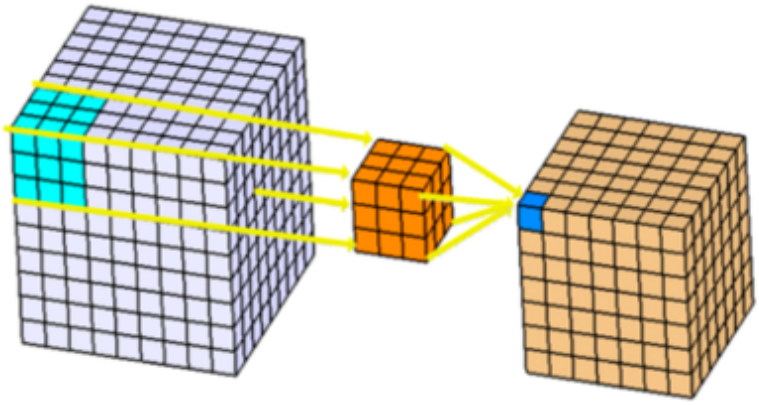


Figure 2.7: 3D Convolutions apply 3D filter on the input feature maps. The filter moves in 3 directions : height, width and depth. The output of the each computation is a 3 dimensional volume space such as cube. [Bansal 19]

power needed to process the data. The pooling layer treats each feature map individually, extracting dominant features from each. Additionally, the pooling layer enables the network to learn different features at various layers, as at each hidden layer the shape of the input feature map is different. Some methods of pooling are : max-pooling and average-pooling. As shown in Figure 2.8, in max pooling, the maximum value is returned from the portion of the image which has been covered by the kernel. Contrary to that, in average pooling the average of all the values from the portion of image covered by the kernel is returned.

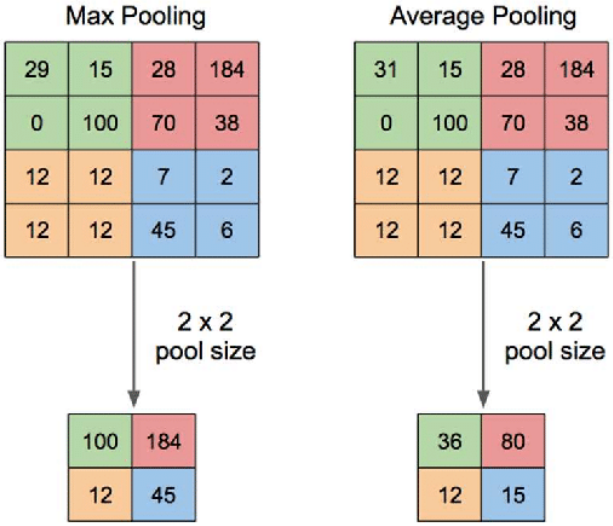


Figure 2.8: The four blocks in similar colors depicts the portion of image covered by the kernel. Output after performing pooing is a 2x2 matrix. Here, we can see that in max pooling the maximum value is chosen from all the values covered by the kernel whereas in average pooling the average of all the values falling under the portion covered by kernel is returned [Yani 19].

Combining the three components mentioned - Convolution Layer, Pooling Layer and Activation function - creates a building block for CNN. Depending upon the architecture design, these blocks are repeated for a specific number of times. The general architecture of the CNN is shown in Figure 2.9. In the figure, the block is made up of CONVOLUTION

+ RELU layer and POOLING layer, which can be repeated as per the requirement. Here, interesting thing to note is that the earlier layers of the network extracts the low level features such as edges, and corners but as we progress in the network, the later layers of the network extract high level features (more abstract) [Molnar 23].

After the $\boldsymbol{n}$ repetitions of CNN blocks, at the end we have Fully-Connected Layers, shown in the Figure 2.9, which is the fourth important component of the CNN network. The Fully-Connected layer learns the non-linear features present in the output of the last convolutional layer. A task head is attached after the Fully-Connected layer, which is task dependent. For instance, if the task is related to classification of vehicle type as shown in the figure, a Softmax Classification technique is used to give probability distribution over all the possible classes in the dataset.

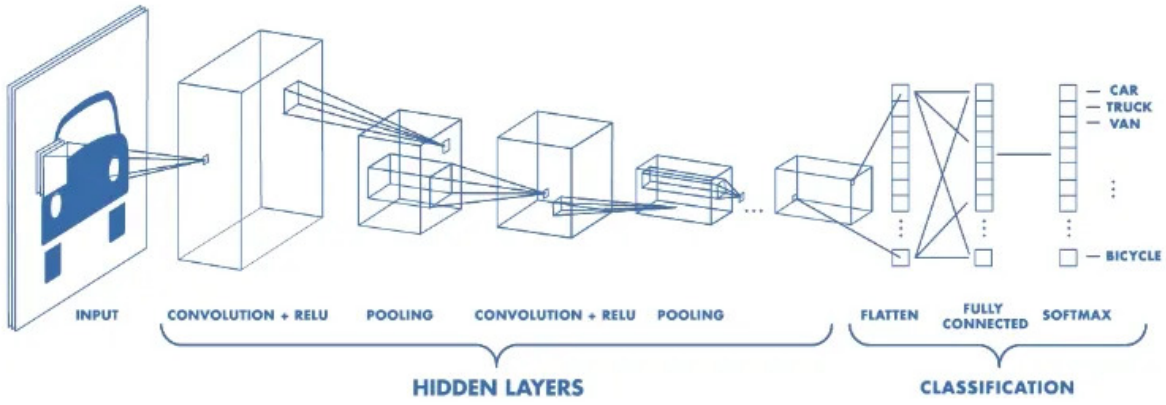


Figure 2.9: The general architecture of CNN. The hidden layers consists of the CNN blocks which are composed of Convolution Layer, Pooling Layer and Activation function. Input image is passed through all the blocks. The output of the last convolutional layer is flattened and given as input to the Fully Connected Layer. The task head is selected on requirement basis. Here, Softmax is used for computing the probabilities for each class. [Mishra 20]

The concepts of neural network and convolutional neural network are considered as the base of the various architectures coming up in section of Object Detection (5.4) and Reinforcement Learning (2.3).

2.2.5 Object Detection

Object detection is a technique in computer vision for classifying as well as locating the object in images or videos. This technique consists of two parts; classification of the object present in the image and locating exactly where the object is present spatially. A classification model is designed just to assign a particular class to the entire image. And in object localization technique the location of an object is approximated using bounding boxes. Both the approaches mentioned above are related to a single object only. But an image can consist more than one objects which needs to be assigned a class and localized. Solution to the aforementioned problem is Object Detection Algorithm. This technique can

detect more than one object present in the image, classify them into their respective classes and locate them spatially using bounding boxes. The difference between the techniques is clearly shown in Figure 2.10. Furthermore, there are different types of Object Detection classified based on network type and the type of data we have. The taxonomy is shown in Figure 2.11. In this paper, our focus will be on types of object detectors based on the Network. It is categorized into two types: Single-stage detectors, and Two-stage detectors. The general architecture of Single-stage detector (One-stage detector) and Two-stage detector is shown in Figure 2.12.

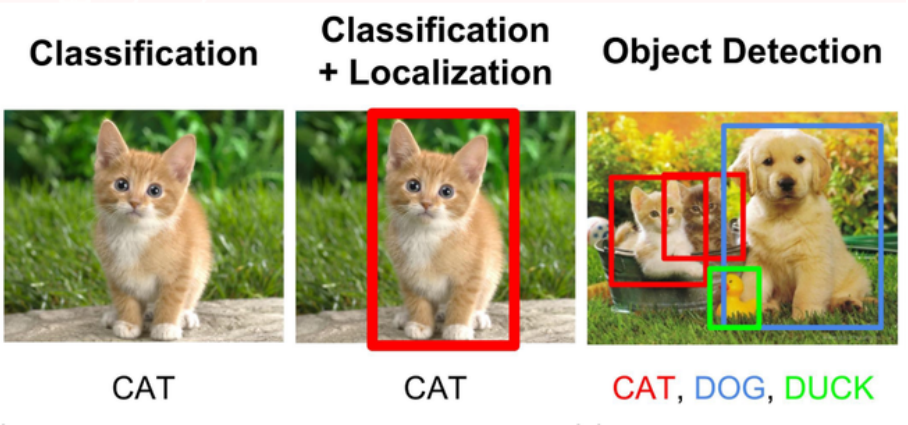


Figure 2.10: Difference between Classification, Localization and Object Detection. Classification and Localization have a single object in the image whereas in third image there are multiple objects, i.e cat, dog and duck. They are classified using color coding and localized with the help of bounding boxes. [GeeksforGeeks 22]

One Stage Detector

One Stage Detector treats object detection as one single regression problem. It uses one CNN network to output the probabilities of the class as well as the coordinates for the bounding boxes. Popular examples are 'You Only Look Once' (YOLO) [Redmon 15] and Single Shot MultiBox Detector (SSD) [Liu 16].

YOLO is considered as a unified model, as a single convolutional network predicts mutliple bounding boxes and also the class probabilities for each of the detection. YOLO trains on full images and the detection performance is performed directly. Due to its simplicity the inference time for YOLO architecture is very fast. Hence, it can be integrated in real life situations where time factor is very crucial. The network takes features of the entire image into consideration to predict the bounding box. This means that it has got a global context for all the objects in the image.

Simple methodology of YOLO is to divide the entire image into a S×S grid. If a center of an object lies into a grid cell, that particular grid cell is responsible for detection of that object. Each grid predicts B (a fixed number) bounding boxes with confidence score associated with each of the box. This score suggests that how confident the model is that the box contains an object and also shows that how accurate are the predicted bounding box coordinates. This architecture is extremely fast and can process 45 frames per second. There is even a Fast YOLO, which is able to process 155 frames per second. Since the inception of YOLO model there has been many versions of YOLO which has improved

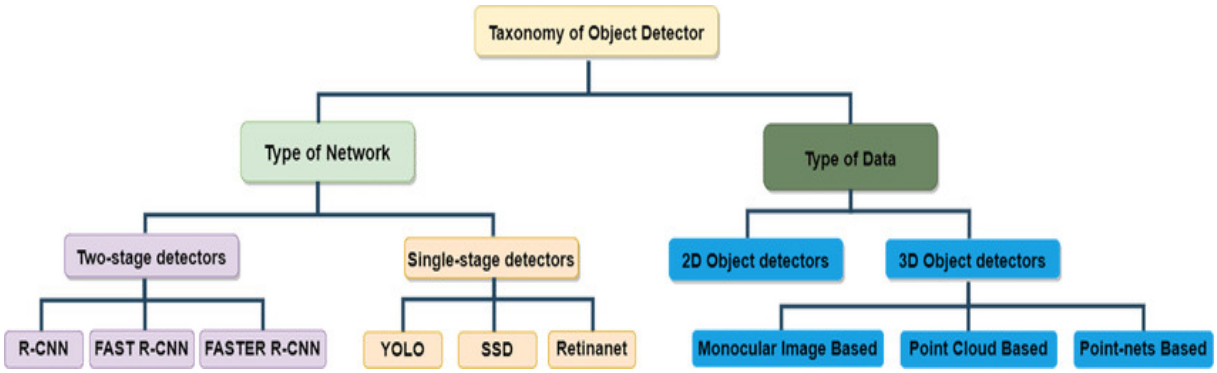


Figure 2.11: The classification of Object detectors are done on the basis of the network type and data type. In this paper we will be focusing more on types of object detector based on different Network architecture. [Balasubramaniam 22]

significantly. At the time of writing, the most recent iteration of YOLO, YOLO8 had been released. There has been architectural changes from the previous versions which has resulted into improved performance. Some of the versions before YOLO8 were YOLO v5, YOLO v6, and YOLO v7. YOLO v8 is build upon the previous versions resulting into further improvement in the performance [Bhalerao 23].

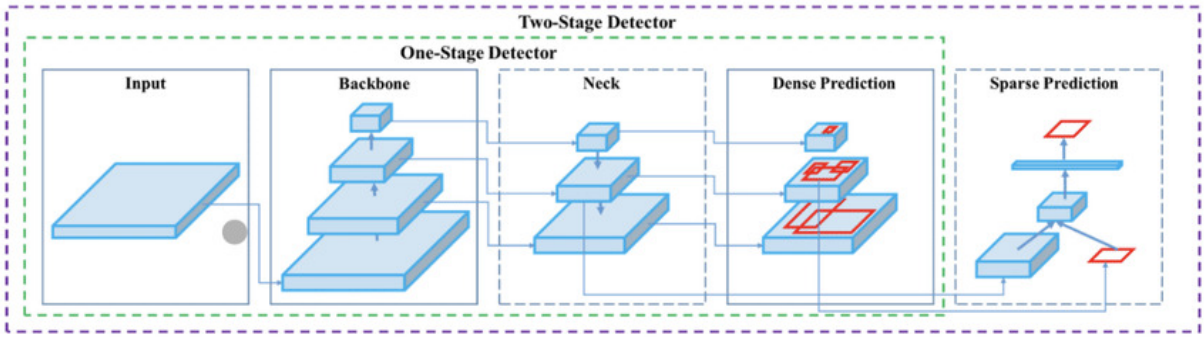


Figure 2.12: The difference between the two-stage and one-stage detectors according to Alexey & Chien-Yao [Daoud 22]

Single Shot Multi-Box Detector (SSD) [Liu 16] is another method under One Stage Detector category. The architecture builds on VGG-16 architecture, but the fully connected layers at the end of the network are discarded. Instead, a set of auxiliary convolutional layers were added, which enables to extract feature maps at each layer. So, at the start of the network we have larger feature maps, which reduces in size at each subsequent level. Similar to YOLO, the bounding boxes are chosen and then those bounding boxes are placed on the feature maps by converting them into grid. There are various bounding boxes with different sizes and aspect ratios. In a nutshell, via several multiscale feature maps, various anchor boxs with different sizes are generated resulting into detection of objects with varying size. The model then predicts class probabilities and regresses bounding box coordinates for each object at multiple scales. For a single object there might be multiple bounding boxes localizing the object, so for that reason at last non-maximum suppression (NMS) is used to compare all the boxes and select the one with maximum confidence.

Two Stage Detector

In contrast to One-stage detector, Two-stage detector takes advantage of two passes through two different Convolutional Networks. The one network is used to extract the features which is also called as the backbone as shown in Figure 2.12 and the second network is for proposing regions of interest (RoI) instead of predefined bounding boxes. The classification and regression only happens on these proposed regions. The very first version where CNN were used with Region proposals was R-CNN: Regions with CNN features [Girshick 14]. The R-CNN network was divided into three modules, each one designed for specific tasks. The first generates the region proposals or region of interests, the second one is a deep convolutional neural network which is used for extracting feature vector for each of the proposed regions. And the last module is a set of class-specific linear SVMs. The region proposal module uses the concept of selective search, which works by dividing the image into multiple regions generally based on the color, texture and other features present in the image. The obtained regions are then merged together to form larger region proposals. Such regions further helps the object detection algorithm to focus on the regions where there is possibility of an object. Particularly in R-CNN, 2000 proposals are made. These 2000 proposals are converted into a format that is compatible with the Convolutional Neural Network in second module. CNN then extract feature from the proposed regions, that is passed onto the class-specific SVMs to compute the confidence scores and four coordinates for the bounding boxes.

R-CNN has slow performance because of performing ConvNet forward pass for each region of interest proposed. The improved version of R-CNN known as Fast-RCNN [Girshick 15], a new method was used known as Spatial Pyramid Pooling Networks (SPPnets) which shared the computation to speed up the process. A convolutional feature map is computed for the entire input image and then classification is performed on the object proposals by using the feature vector extracted from the shared feature map. This method speeds up the R-CNN by 10 to 100x at test time. Training time also reduced by approximately 3x due to speed up in feature extraction of proposals. In the Figure 2.13, it is shown that a new ROI-pooling layer has been added. In this layer, all the proposals of different shape are transformed into square shaped features so that it can be fed to the fully-connected layers further. Similar to other classification networks softmax is used to compute the class prediction and a regressor to compute the coordinates of the bounding box. box.

In the year 2016, even better version of R-CNN was introduced, known as Faster R-CNN [Ren 16]. This paper introduced the concept of Region Proposal Network (RPN). This network shares full-image convolutional features to the detection network. This eventually helps to obtain region proposals nearly cost-free. RPN is a fully convolutional network which predicts the coordinates as well as the objectness scores at each position. The architecture is divided into two parts, one deep fully convolutional network module that proposes the region (RPN) and other module is that of Fast R-CNN detector that uses those regions for further detection.

This paper majorly concentrates on the training of the backbone weights and how it is being used in Faster R-CNN architecture. More about training and usage of the backbone will be discussed in Section 5. This backbone is the common CNN network whose output is used for both Region Proposal Network and the Fast R-CNN detector as shown in the Figure 2.14. Some example for CNN backbone are ResNet (ResNet18, ResNet32,

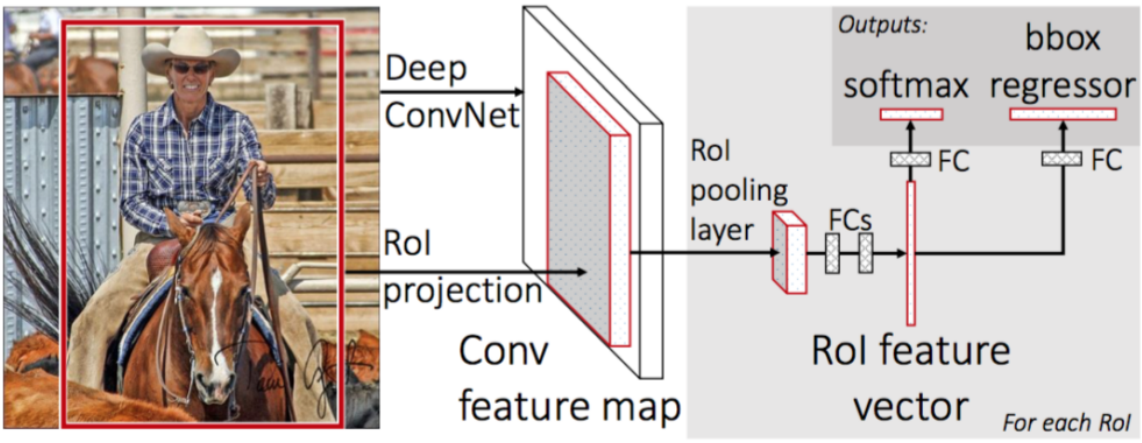


Figure 2.13: Fast R-CNN architecture. The input to the fully convolutional network is an input image and multiple region of interests (RoIs).The red box denotes the RoI. Further it is given as input to the Pooling layer, that makes the dimension of the RoI compatible with the Fully connected layers (FCs). Two outputs of the network: softmax probabilities for classification task and regression offsets for bounding box. [Girshick 15]

Resnet50) and VGG Network. The input image is passed through a CNN backbone to extract feature map. The feature maps computed consists of the visual data from the image. That is further given to the RPN and the Fast R-CNN detector as input. The major aim of the backbone CNN is to capture the important features present in the image. The initial layer of the network captures the low-level features like edges and textures, and the deeper layers of the network captures the high level semantic features. As both the networks, RPN and Fast R-CNN detector takes the same extracted features, it reduces the computing time and memory by significant amount.

2.3 Reinforcement Learning

Reinforcement learning is a type of machine learning algorithm that learns by trial and error methodology. An agent is placed in an environment where it interacts with it and using the feedback from its own actions and experiences learns overtime. The feedback obtained by the system are the positive and negative rewards awarded on the basis of the action taken. The Figure 2.15 shows the working of the reinforcement learning model in generic way. As shown in Figure, the agent interacts with the environment by taking some actions. These set of actions can be viewed as an episode. Each episode consist of different states visited during the entire episode, actions taken to reach those states and the rewards obtained by taking the actions. The main objective of the Reinforcement Learning algorithm is to learn a policy that maximizes the cumulative reward over the course of many episodes. The main advantage of reinforcement learning agents over some supervised learning process is that after going through the iterative process the agent learns to behave in the environment over time, which enables the machine to work accurately in complex and changing environments. There are many important components which make up the entire reinforcement learning algorithm. This chapter gives an introduction to all the components of classical Reinforcement Learning and will serve as the foundation of the Deep Reinforcement Learning approaches in Section 2.4

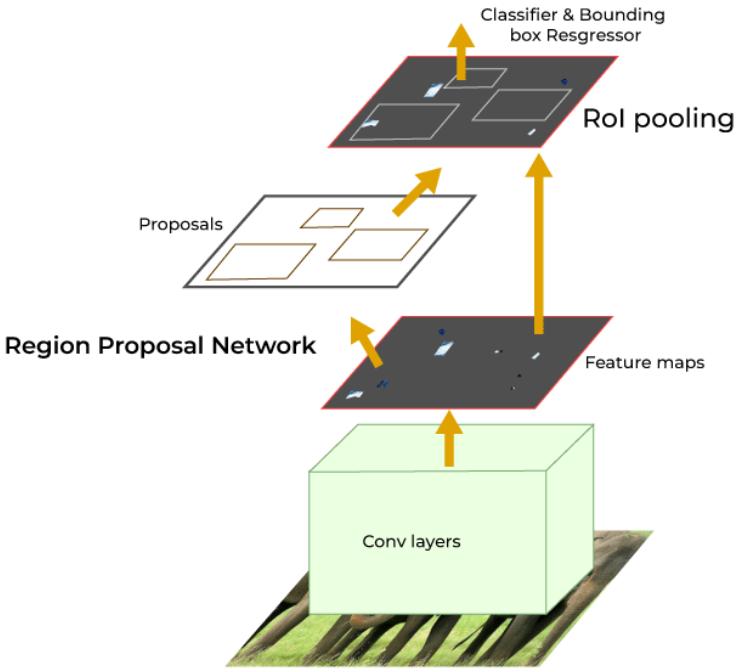


Figure 2.14: Faster R-CNN architecture. It is an unified network for object detection. The first part is a network of Region Proposal and another part is a network that gives Class and bounding box for the detected objects as an output. [Girshick 15]

2.3.1 Markov Decision Process (MDP)

Markov Decision Processes(MDPs) are mathematical frameworks to describe an environment in RL and almost all RL problems can be formulated using MDPs [Bhatt 18]. An MDP consists of a set of finite environment states S, a set of possible actions $A(s)$ in each state, a real valued reward function $R(s)$ and a transition model $P(s', s \mid a)$. The RL problems are closely related to MDP because in MDP the agent or the decision maker takes an action being in a state and in response the environment transitions the system to a new state. On the completion of transitioning to the new state, reward is awarded to the agent. The main aim of any MDP is to find an optimal policy following which maximum cumulative reward can be obtained.

MDPs have proved to be powerful tool for Reinforcement Learning, but there are some drawbacks. MDPs assume that the agent is aware of the entire environment and the transitions between the states. But in reality, it is not always possible to have access to the entire information. Another limitation is that MDP assumes Markov property, which states that the future state only depends on the current state and the action which will be taken in that state, not on the previous actions taken and the states visited. In real scenarios, this assumption might not hold always.

2.3.2 Components of Reinforcement Learning

As shown in Figure 2.15, there are several components such as the agent, state, action, environment, rewards, and episode that makes up the entire RL system. The agent takes

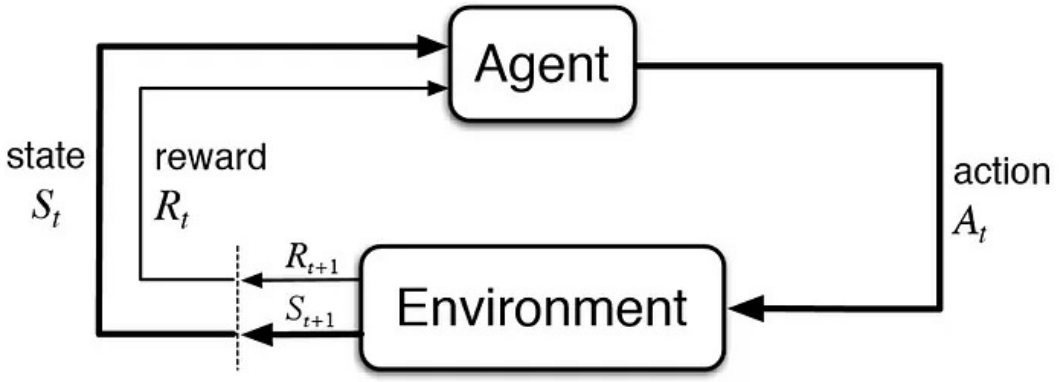


Figure 2.15: The action-reward feedback loop of an generic RL model. [Bhatt 18]

the action A_t at time-step t which is given as input to the Environment. On the basis of the action taken, current states S_t is changed to the next state S_{t+1}. The new state and the reward R_t corresponding to the action taken are given back to the agent as a feedback. This loops for n number of time-steps, and the rewards and experience obtained during the iteration trains the model to take actions that results into maximum cumulative rewards. The following section explains all the components in detail.

2.3.2.1 Action

Each episode consist of time-step, and at each time-step a decision needs to be made by the agent. That decision made is called as the Action taken by the agent. In RL, agent takes the action for example, in Autonomous driving the car is the agent or the ego vehicle and that car takes the action. There are a set of possible actions that agent can take while being in a particular state. Those possible actions can either be discrete actions or continuous actions. Discrete action only have finite number of possible actions, for instance, up, down, left, right. Whereas, in continuous space there can infinitely many actions, such as velocity of an autonomous vehicle. There are two ways to take actions : either the agent takes an action knowing that the reward received after taking that action will be maximum or agent might not know that best action in a particular scenario and has to take random action. The first method is considered as Exploitation and latter method is called as Exploration (which will be discussed in detail in further section). There should be a balance between exploration and exploitation in order to maximize the rewards. In equation 2.4, a_t is the action at time-step t, π is the policy that gives the probability distribution over all the actions possible on the basis of the current state s_t.

$$a_t = \pi(s_t) \tag{2.4}$$

2.3.2.2 Agent

An agent receives an Observation at time t and needs to output an action A_t. As discussed in the previous section, an Agent is an entity or component in RL that makes the decision of what actions needs to be taken in the current state. That action needs to be decided by the agent and that action can be anything. For making the decision, the agent expects

the current state as the input from the environment. The objective of RL is to optimize the agent in a way that actions taken by it results into maximizing the results.

2.3.2.3 State

The state represents the configuration or situation of the environment that an agent can take as information and make a decision at a given time step. In addition to agent perceiving information from environment, state can also be agent's own internal variables that are different from the environment the agent is present in. States can vary depending on the complexity of the environment and the task at hand. Similarly to action space, there is also state space that are all the possible states that the agent can be in. The state can be either discrete or continuous. Grid is an example of a discrete state space, where the number of states are finite such as finite positions in a grid. On other hand there are continuous state spaces, which consists of infinite positions possible for agents to be in. For example, if we have a Simulated Environment of a City, there is a possibility of being anywhere within that range. Furthermore, the state are fully observable or partially observable. In the case of fully observable, the agent is aware of all the possible states in the space. In partially observable case, agent has incomplete knowledge regarding the state spaces. In such cases, the agent needs to be aware of the surrounding and gather information using sensors, like in autonomous driving where entire surrounding is not known to the agent. As it explores the environment new information and new states are obtained. The complexity of the state space directly affects the learning ability of the agent. More complex the state, more difficult and more time taking it might be to learn [Sojitra 23].

2.3.2.4 Environment

The agent is present in a system and also interacts with it is called as an Environment. The environment can either be simulated or in real-world. Typically, training a RL algorithm in real-world scenario proves to be expensive. For instance, the training of autonomous driving car directly in real-world will have lots of expenses. During the training, the agent should be presented with different scenarios to learn from that. This will increase the energy consumption and in addition to that, an incorrect decision by the agent in the real-world can cause damages and incur losses. Therefore, training often occurs in a simulated environment. For instance CARLA is an simulation platform used for various applications related to autonomous driving which is further explained in Section 4. In RL, the environment is a source of all the information that agent requires to learn from. Environment receives the action at time t and gives observation/state as an output at the next time step. The agent is the part of the system which takes the decision, anything beside that is the part of the environment [Silver 21]. For example, in CARLA the sensors like rgb camera, or LIDAR that helps for capturing the surrounding of the car are considered as the part of the environment.

2.3.2.5 Reward

In reinforcement learning, a feedback is required from the system to learn and achieve the desired behavior. The feedback is given in the form of rewards. This can be compared to learning technique used with Humans. A child receives a reward maybe a chocolate

for doing a good job and get a scolding due to improper behavior. In similar way the RL system receives reward which is a scalar value and the value depends upon the quality of the action taken by the agent. The reward signal is the most important of reinforcement learning process, as on the basis of the rewards received all the updates to the policy and value function are made. The reward function must be designed in a unique way for every different applications. That reward function is an incentive mechanism that tells the agent which of the actions taken are right and which of those are wrong. Positive values act as a reward for correct actions and negative values act as punishment for wrong actions. These rewards and punishment encourages the agent to learn to take the correct actions in a given particular state. The aim is to maximize the total rewards, so sometimes there is a requirement to sacrifice current maximum rewards.

Rewards, besides being general, also give feedback along the way, possibly at each step, about how well things are going towards the goal. This in-between signal is crucial when dealing with long or endless experiences – without this feedback, learning wouldn't work [Silver 21].

Designing of a reward function is most crucial part of Reinforcement Learning. the reward function should accurately represent the problem's objective. As rewards are the driving force for the training, poorly designed reward function may lead to converging to local optima which will in turn not allow the agent to reach the goal state. The reward function should be designed in such a way that it not only solves the basic problem but is able to scale with the increase in the complexity of the same problem. The signals given to the agent by the rewards should be clear. It shouldn't be contradicting itself.

2.3.2.6 Episode

An episode in respect to reinforcement learning can be defined as sequence of states, actions and rewards. When an agent starts interacting with the environment, there are sequence of states and actions it goes through. And at each of these points it receives rewards for the actions taken. All these makes up an episode. Every episode is a learning phase for the agent. An episode starts at an initial state, which can be defined on the basis of requirement. The episode last till it reaches the goal state or a termination condition. For example, in autonomous driving situation, if it reaches the final destination the episode will end or it can also end if the car collides with some object in the environment. The sequence mentioned earlier is know as the trajectory of the episode. The agent optimizes its behavior on the basis of the trajectory and tries to maximize the rewards. Shorter episode can lead to faster learning process but there is a possibility of not exploring everything in the environment. On the other hand, longer episode may take more time to learn, but it will offer more exploration and provide more information for learning.

2.3.3 Policy

A policy is an strategy that an agent uses in order to achieve the objective. The policy is a function of the state and the environment it is present in. In other words, a policy maps from states to actions that will define the behavior of the agent. The learning of policy should be done in such a manner that it maximizes the reward over time which is dependent upon the actions taken. There are two types of policy: deterministic and stochastic. In deterministic setting, on reaching a particular state, it will always return

the same action whereas in stochastic method, the policy returns a probability distribution over all the possible actions.

There are different types of policy but we will dive further into the stochastic policies. As show in Figure 2.16, the policy network receives the state information S_2 from previous time-step and it gives distribution over action-space as the output (P(a_1), P(a_2),..., P(a_n)), where n is the total number of action possible. Action a_3 is chosen on the basis of the probability distribution which is then given as input to the environment. The environment gives the next state (S_6) and reward (r_2) as output.

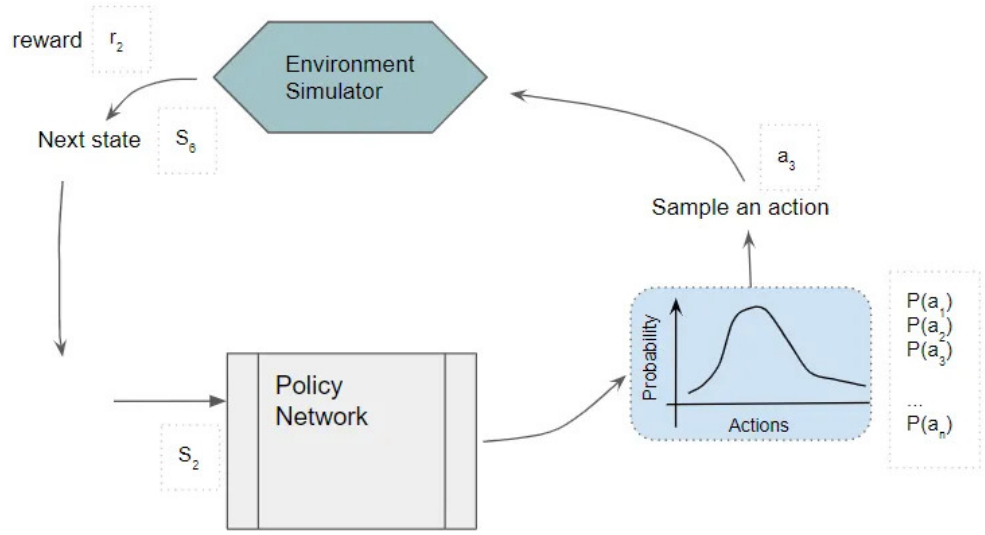


Figure 2.16: The policy network is stochastic and on receiving state information from environment, it outputs a distribution over all the possible actions that is further given as an input to the environment. Application of action in the environment changes the current state to the next state. [Doshi 21]

2.3.4 Value Function

The expected value of an agent in a certain state is defined as a Value function. In other words, value function is a measure of goodness of an agent being in a particular state or taking a particular action. There are two types of value function: state-value function and action-value function

State-value function

It is defined as a function that estimates the cumulative rewards starting from a state s, and following a policy π. This is function of state s as defined in Equation 2.5. [Karunakaran 21]

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{T-1} \gamma^t r_t \mid s_t = s \right] \tag{2.5}$$

Action-value function

This function is defined as the function that estimates the expected reward which starts from the state s, following the policy π, and taking action a. This is function of state s and action a as defined in Equation 2.6. [Karunakaran 21]

$$Q_\pi(s,a) = \mathbb{E}_\pi \left[\sum_{t=0}^{T-1} \gamma^t r_t \mid s_t = s, a_t = a \right] \tag{2.6}$$

In the equations 2.5 and 2.6, γ is the discount factor, which weighs the rewards in the future.

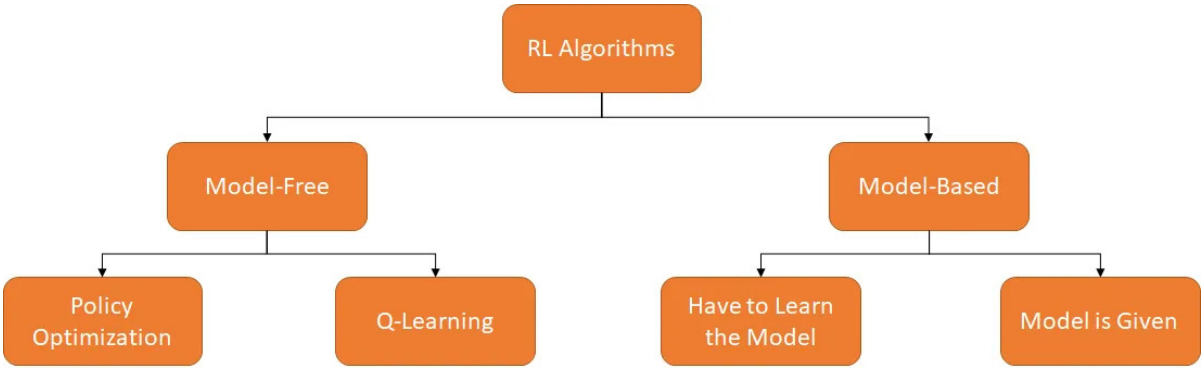


Figure 2.17: Model-Free vs Model-Based Taxonomy [Owen 20]

2.3.5 Model-free RL vs Model-based RL

As shown in the Figure 2.17, the taxonomy of the RL algorithms is mainly divided into Model-free RL and Model-based RL. In context of reinforcement learning, the term 'Model' has a different meaning. It means the collection of obtained information regarding the environment. From Section 2.3.1, we recall that there five elements denoted as : S, A, P, R, γ, S where S and A denotes State-Space and Action-Space respectively. R(S,A) returns rewards and is function of A and S. $P(s_2|s_1, a)$ is a transition function that gives the probability of transitioning to state s_2 when s_1 and a is given. Here, the reward function (R) and the transition function (P) are not known to the agent. The agent needs to go through a lot of trials and errors and learns by observing the environment to leverage the reward feedback [Zhang 20].

There are two ways to solve the above mentioned problem. One way is to predict the state and the rewards. Here the $P(s_2|s_1,a)$ and R(S,A) are unknown, but after collecting some samples (s, a, s´, r) from the environment, and if the samples are enough, the functions can be learned using supervised method. After this, all the elements are known. This approach is known as model-based RL. On the other hand, instead of modeling the environment, directly look for the optimal policy. For instance, the Q-learning algorithm chooses the action based on the Q-values in the table and reaches to an optimal Q-value function. In such kind of algorithms, the approach is different, where focus is not on modeling; instead highest rewards are searched directly. This method is known as model-free RL.

2.3.6 Exploration

In RL, exploration means to try to visit to new states and execute new actions in order to get more information regarding the environment and improve the knowledge base of the agent. The goal of the exploration method is to visit new states and discover new actions in order to maximize the rewards. If exploration is not done, the agent might never get the best actions possible and best state to be in. This might result into attaining local minima due to lack of information. Exploration can lead to reaching the optimal state and finding the optimal policy. One of the most used strategy is epsilon-greedy method.

The downside of exploring more is that there is no guarantee of getting the optimal solution or better solution if explore more. It can risk the current good rewards obtained by taking already known actions or states. But exploration can be the best option if the current actions is not helping to reach to the goal state.

2.3.7 Exploitation

In context of RL, Exploitation means to take the actions that are already known and which gives good rewards. As discussed in previous section, that trying new states or actions might give the agent worst rewards. So, instead of exploring, the cumulative rewards are maximized by taking the best known action in current state. Hence, Exploitation is considered as a greedy approach where agent decides upon the action on the basis of current maximum reward which is totally based on current information.

In RL, exploration-exploitation is a common dilemma which the agent needs to face whenever the agent is trying to learn about the environment. The dilemma is between choosing what the agent already is informed about and getting an immediate maximum reward (exploitation), and choosing something new, which the agent is unsure about and possible learn more and maybe maximize the rewards over a period of time (exploration).

2.4 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is a combination of Deep Neural Networks and Reinforcement Learning. In this domain, RL takes the advantage of neural networks to take decisions for the agent by taking unstructured input from the environment to achieve a particular objective/goal. In today's day and age, Deep RL is being used in various application such as gaming, robotics, computer vision, natural language processing, healthcare and autonomous driving.

Incorporation of Deep Reinforcement Learning in the field of gaming has shown a bigger improvement. AlphaGo [Silver 16] versus Lee Sedol, also known as DeepMind Challenge Match was a match of Go between the world best player Lee Sedol and a computer program trained known as AlphaGo. AlphaGo won 4 matches against him which considered a very big achievement in the field of board games. Advancements in this field are taking place rapidly.

Neural Networks are very good function approximators, that becomes very useful in the field of RL when the action-space and state-space becomes to large and complex. A policy function or a value function can be approximated by using a neural network. Instead of

storing such large transitions between states and their values, neural nets can be trained to map states to values or state-action pairs to Q values.

If the state or the environment is visual, Convolutional Neural Networks can be used to take the visual information as the input. Here the CNN instead of doing a classification task on the input, will provide best action suitable in that state as an output. Such network predicts the action and are called as Action network. Another network that can be used by RL is to predict the value based on the state-action pair. This network is called as a Critic network as it helps to evaluate and optimize the policy.

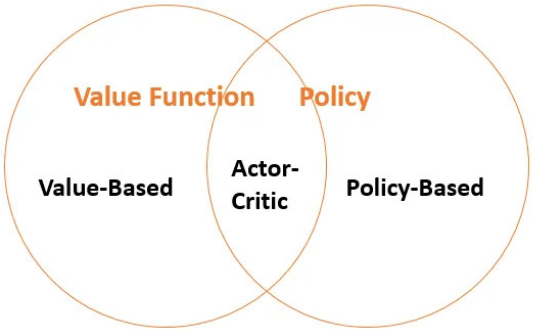


Figure 2.18: Value-based and Policy-based Algorithms [Owen 20]

2.4.1 Actor-Critic Models

As show in Figure 2.18, there are two approaches to RL problems. In Value-based algorithms, the objective is to estimate a value-function, which is the expected rewards for taking an action in a particular State. For instance, Q-Learning and SARSA. Secondly, Policy-based algorithms on the other hand directly learns to map between the states and actions which return the maximum rewards. REINFORCE is one of the major example under this method.

There are some disadvantages for both of aforementioned methods. Value-based RL includes slow-learning rate for large and complex problems, high reliance on initialization step, and the requirement to scale trade-off between exploration and exploitation [Son 22]. The disadvantage of Policy-based methods is that it is inefficient to evaluate policy and these methods have high-variance. REINFORCE is one of the policy-based algorithm. Due to the use of Monte-Carlo sampling to estimate return, there is a significant variance in policy gradient estimation. In REINFORCE, the aim is to increase the probability of actions in a trajectory proportionally to the magnitude of return. But, due to the stochasticity of the policy and the environment, trajectories can give different returns and which can lead to higher variance. To reduce the variance, large number of trajectories can be used, but this significantly reduces the sample efficiency. Hence, it is inefficient [Face 24].

The solution to deal with the drawbacks mentioned here is to use a method that takes the advantage of the strength of both Value-based and Policy-based methods. It is called as the Actor-Critic method, that reduces the variance and also speeds up the training process. There are two networks: Actor and Critic. The responsibility of the actor network

is to predict which action should be taken whereas the critic (as the name suggests) will rate the action and tell how the updates should be made to the networks. One type of actor-critic method is used in the implementation that is explained in detail in Section 2.4.4.5.

2.4.2 On-Policy and Off-Policy

This section will explain the difference between 'Off-Policy' and 'On-Policy' learning. Before getting into detail, it is important to explore two key terms: Behavior Policy, and Target Policy. The behavior policy is the one which the agent takes into consideration while interacting with the environment. In other words, it decides which actions are taken by the agent on the basis of the current state. Whereas, the target policy is the policy that agent wants to improve and trying learn.

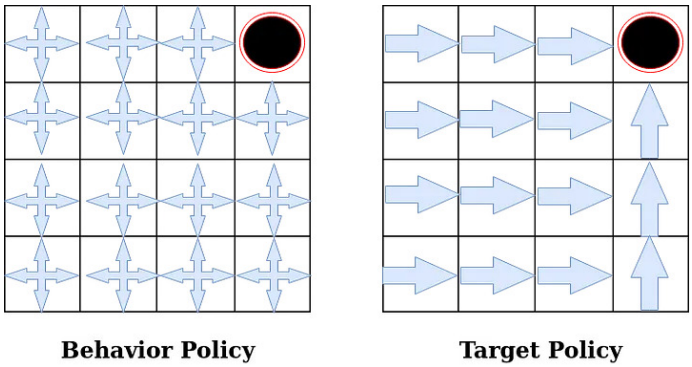


Figure 2.19: Value-based and Policy-based Algorithms [Suran 20]

In On-Policy learning, the policy which is being evaluated and improved by the algorithm is the same that is being used to take actions. Here, the algorithm is evaluating and improving the policy which is being already used by the agent for action selection. So, 'Target Policy' == 'Behavior Policy'. Here the Policy taking the actions and evaluating the expected return is same. For instance, in SARSA, the current policy is used to take actions. And then on the basis of the quality of the action taken, the same policy is being updated. There is constant update in the policy after interacting with the environment from the current state.

In Off-Policy learning, the policy selecting the actions and the policy that is evaluated and improved are both different. So, 'Target Policy' != 'Behavior Policy'. Here, behavior policy is responsible for taking the actions and target policy is being updated on the basis of the actions. Here, the actions taken might not be taken by the current policy. Behavior policy is also updated but after a certain interval of time. So the action is not taken by the updated current policy. There are some benefits of off-policy methods such as: an agent is learning other policy then it can be used for continuing exploration while learning optimal policy, agent can learn from demonstration, and convergence is also faster [Suran 20]. There is a concept called Experience Replay used in off-policy learning methods. The task of this replay buffer is to store previous experiences. The advantage of this buffer is that when the agent is exploring new states or unseen parts of the environment, it not only utilizes those new experiences but also leverages the previous experiences stored in the buffer for learning purposes.

2.4.3 CNN Policy Networks

The policy network used in this work is CNN Policy. Another type of policy available is that of MLP Policy. Here, CNN policy will be discussed in detail. This type of policy is generally used for image-based states. The convolutional layers are used to process the image input. For example, in the case of autonomous driving in simulated environment where each state is an image uses CNN policy. We will see that Stable Baselines3 (Section 4.2.1) provides policy networks for images (CnnPolicies).

2.4.4 Some Deep RL Algorithms

In the upcoming sub-section, we will delve into some of the types of Deep RL algorithms developed and used in real-life scenarios. The main focus of this paper will be on the Soft Actor-Critic model regarding which further implementation details are presented in the Section 5.

2.4.4.1 Deep Q-Networks

In Deep Q-Learning, for each possible action at a particular current state a deep neural network is being used to approximate different Q-values. The neural network is a function approximator which maps states and actions to scalar values. The scalar values approximated represents the value of taking a particular action when being in a particular state. The main objective is to assign greater value to those state-action pair which maximizes the Q-function. Instead of directly updating the Q-value of a state-action pair in the table, in DQN during training phase a loss function is created which compares the prediction from the network with the Q-target and uses gradient descent to update the weights of the network to further make better predictions of Q-values. The Q-target mentioned here is the real target about which there is no idea. It needs to be estimated, so a separate network with fixed parameters is used to estimate the Q-target. And the parameters from our DQN are copied every C-steps.

To deal with the problem of correlated data and non-stationary distributions, experience replay-mechanism was introduced [Mnih 13]. Using which, it could randomly sample previous transitions, and hence stabilize the training process. The advantage of experience replay is that the experiences can be stored and past ones can be used again for training. Learning again from past experiences prevents the network from forgetting what it had learnt earlier. The training of DQN is divided into two steps: firstly, performing the actions and gathering the experience which is stored in the tuples of a replay memory; secondly, small batch from that tuple is randomly selected and then learn from that batch using gradient descent methodology.

2.4.4.2 Advantage Actor Critic (A2C)

Advantage Actor Critic has two variants: Advantage Actor Critic (A2C) and Asynchronous Advantage Actor Critic (A3C). The Q-value is divided into two parts, state value function V(s) and advantage function A(s, a) as show in the Equation 2.7.

$$A_t = Q(s_t, a_t) - V(s_t) = r_t + \gamma * V(s_{t+1}) - V(s_t) \tag{2.7}$$

The characteristics of advantage function is to tell how good an action is compared to other actions when being in a particular state. Whereas, the value function captures how good it is to be in that state. So, instead of critic network learning Q-values, it learns to approximate Advantage function. So instead of only stating which action is good choice, it will also tell how much better it is from other action choices. The main advantage of this function is to reduce the variance of the policy networks. [Karaginannakos 18]

2.4.4.3 Proximal Policy Optimization (PPO)

PPO is an On-policy reinforcement learning algorithm. The main property of Proximal Policy Optimization algorithm is to avoid policy updates that are too large. This will lead to improve agent's training stability. Here, to make sure that the policy has not changed by large difference from the former one, it calculates the ratio between current and former policy. The ratio signifies the difference between current and old policy and this ratio is clipped to a specific range $[1-\epsilon, 1+\epsilon]$. In this manner, the current policy won't go too far from the old policy. This results into small updates to the policy and more stable training [Schulman 17]. The reason behind favoring smaller updates to the policy is that chances of converging to optimal solution using smaller updates is more. Large steps in policy updates can result into overshooting and not getting the optimal policy.

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) A_t, \text{clip} \left(r_t(\theta), 1-\epsilon, 1+\epsilon \right) A_t \right) \right] \tag{2.8}$$

From Equation 2.8, θ is the policy parameter, $\mathbb{E}_t$ denotes expectation over timesteps, r_t is the ratio of probability under the new and old policies, A_t is estimated advantage at time t, ϵ is a hyperparameter.

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \tag{2.9}$$

In equation 2.9, it is the probability of action a at state s in the current policy, divided by the same with respect to previous policy. If $r(\theta) > 1$, the action a in state s is more likely in the current policy than the old policy. If $0 < r(\theta) < 1$, the action is less likely for the current policy than the old policy. This ratio method proves to be an easy method to find the divergence between old and new policy. The expected value mentioned in the equation 2.8 is taken over a batch of t experiences.

2.4.4.4 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient is an Off-Policy algorithm, which means that it has a separate policy for taking actions called as behavior policy and updates the target policy on the basis of experiment obtained using behavior policy (Section 2.4.2). There are four neural networks used in DDPG: Q-network (θ^Q) , deterministic policy network (θ^μ), target Q-network ($\theta^{Q'}$), and target policy network ($\theta^{\mu'}$).

In DDPG, the Q network and policy network are similar to that of Advantage Actor-Critic, but with an important difference. Unlike A2C, in which the network outputs a probability distribution over all the discrete actions, in DDPG, the Actor network directly maps the states to action. The target networks are updated after every C-steps from their original copies. The use of target networks increases the stability of training. [Yoon 19]

2.4.4.5 Soft Actor-Critic (SAC)

Soft Actor-Critic (SAC) is an off-policy actor-critic deep RL algorithm. It is based on maximum entropy reinforcement learning framework [Haarnoja 18]. The objective of this framework is that the actors goal is to maximize the expected reward and simultaneously also maximize entropy. The combination of off-policy algorithm and actor-critic approach it has achieved state-of-the-art performance for difference benchmark tasks. The aforementioned algorithm, DDPG, is also an off-policy DRL algorithm. However, the problem with these method is that it is very sensitive to hyper-parameters and requires lot of fine tuning before it can converge. SAC presents a method that can combat with the convergence problem [V.Kumar 19]. In figure 2.20, general architecture of actor-critic is shown. SAC is an actor-critic algorithm. So, it is a combination of policy-based and value-based approach.

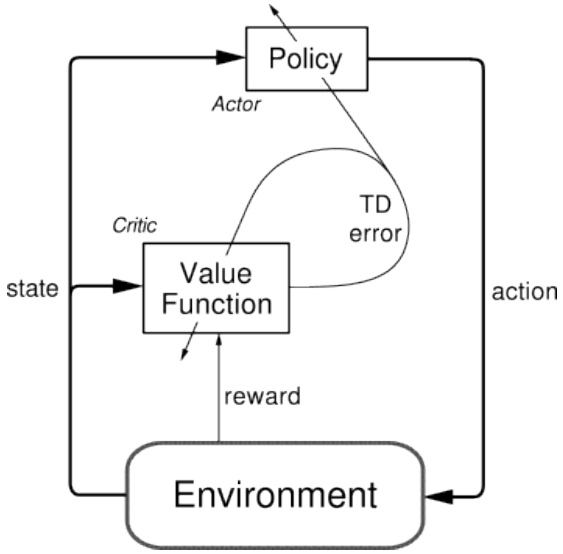


Figure 2.20: General Actor-critic architecture. [Karunakaran]

SAC uses entropy regularization technique. This method encourages exploring by maximizing the entropy of policy. This leads to stability of the entire training process and agent can explore different states. SAC is a good choice for scenarios where the action-space is continuous and state-space is high-dimensional. Here the term Entropy means that how unpredictable a random variable is. If a random variable always takes the same value, the entropy becomes zero. On the other hand, if the random variable can be a any number with equal probability then it attains high entropy and is very unpredictable. In SAC, the aim is to attain high entropy because it will encourage exploration by assigning equal probabilities to actions which has nearly equal Q-values. By this, it will avoid assigning high probability to any one of the actions from the action space. [V.Kumar 19]

There are three networks in SAC: network representing state-value (V), the second network is a policy function, and third one is soft q function. Learning of state-value function is done by minimising the squared difference between the prediction of the value network and expected prediction of Q function with the entropy of the policy [Karunakaran]. Training of Q function is done by minimizing the squared difference between predicted Q value and reward plus the discounted expectation of state-value of next state. And finally, learning of policy depends upon the Q-value to reduce the variance.

$$J(\theta) = \mathbb{E}_{(s_t,a_t)\sim D}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) + \alpha\mathcal{H}(\pi_\theta(\cdot|s_t))\right] \tag{2.10}$$

The above equation 2.10 is the objective function of Maximum Entropy RL, where θ are the parameters of the policy, r(s, a) is the reward function, γ is the discount factor, α is a hyperparameter that controls the trade-off between maximizing the rewards and maximizing the entropy, and $\mathcal{H}(\cdot)$ is the entropy.

3. Related Work

3.1 Autonomous Driving

The potential of Autonomous vehicles being used in transportation is really high. Assistance is already being provided by lane-keeping and adaptive cruise control systems. Autonomous system will be included in driving in the coming years. Most of the analysis suggests that it would be much more safer with inclusion of autonomy to car systems. The reason given is that the car system can simultaneously sense more road users and predict their behavior and hence increase the safety. [Millard-Ball 18]

The interaction between cars and internal communication between them will be lot safer if all the cars present on the roads are autonomous. The major concerns is related to the other entities on the road and their safety. Pedestrians are highly susceptible to injury on the road. Either autonomous vehicle will increase the safety or decrease it due to total reliance on the technology.

3.1.1 Deep Reinforcement Learning approach for decision making

For autonomous vehicle systems, the main question that arises is of making decisions in a space where the states are continuous as well as the actions required to be taken are continuous. There are different decision-making technologies used for autonomous vehicles [Liu 21]. The methods are divided into classical methods which consists of Rule-based methods, Optimization methods, and Probabilistic methods, and learning-based methods which consist of statistic learning-based methods, deep learning-base methods, and reinforcement learning-based methods. The most commonly used method for making decisions in the field of Autonomous driving has been Reinforcement Learning method. All the studies have proven that DRL can be an effective way for developing autonomous vehicles.

3.2 DreamTeacher: Pretraining Image Backbones with Deep Generative Models

In this work of DreamTeacher [Li 23], the main idea is of introducing a new framework which is self-supervised feature representation learning framework and is utilizing generative networks for pre-training for backbone of different downstream tasks. The concept of knowledge distillation is explored in this work, that includes two approaches which are as follows:

- *feature distillation*: distilling learned generative features onto target image backbone instead of pretraining these backbones on large labelled datasets.
- *label distillation*: distilling labels obtained from generative networks with task heads onto logits of target backbones.

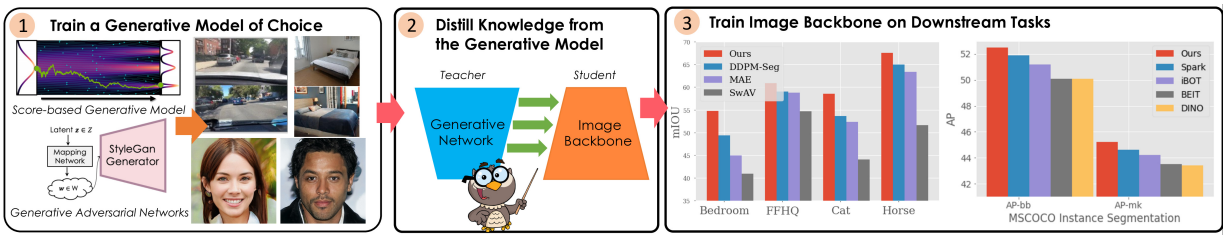


Figure 3.1: A framework for distilling knowledge from pre-trained generative network onto a target image backbone which is then used for downstream task. [Li 23]

The Figure 3.1 describes the general framework where a Generative model is trained in the first step followed by the distillation of the knowledge from generative model onto backbone. And the final step of training the downstream task.

Different generative models are investigated: for GANs, unconditional BigGAN, ICGAN, StyleGAN2, are used and for diffusion-based models, ADM and Stable Diffusion models are used. The conclusion of this paper was that DreamTeacher framework outperformed the existing approaches in both domain of detection and segmentation. The conclusion of the paper is that generative network that uses unlabeled dataset learns semantically important features that can be further distilled successfully in image backbone and improve the downstream tasks.

3.3 Robotic Offline RL from Internet Videos via Value-Function Pre-Training

In this work, it is shown that value learning on video datasets learns representations that are more favorable or suitable to downstream robotic offline RL tasks than other approaches for learning from video data. Here the concept of pre-training on video data with robotic offline RL tasks is utilized. Their main contribution is of V-PTR (Visual Pre-training for Robots), that has the aim to seek good value function initialization for downstream offline RL. [Bhateja 23]

The V-PTR system pre-trains in two phases: first on video data, and then on multi-task robot data. First step is to train an intent-conditioned value function on action-less video data using a value-learning objective [Ghosh 23]. In the next step the representation obtained is refine with multi-task robot data with actions and rewards provided, using Offline RL. The features of the environment should be captured by the two phases of pre-training. Here the robot data is expected to bridge the gap between human video and the robot. After the pretraining is done, the system is fine-tuned for the downstream task.

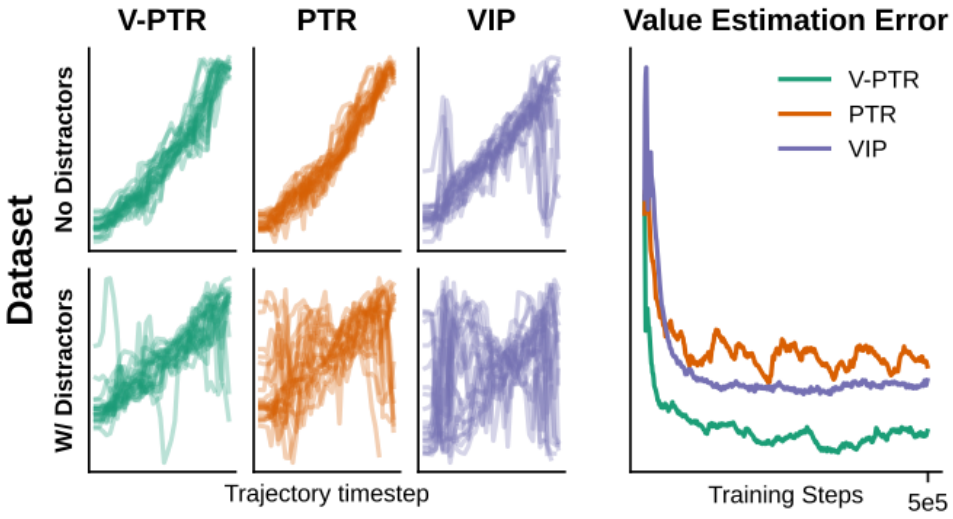


Figure 3.2: Visualizing the learned values V(s) w.r.t. time-step t on rollouts from training data (top), held-out data (middle), and rollouts with distractor objects (bottom) obtained after multi-task pre-training in phase 2. Note that values trained by PTR and VIP tend to be highly non-smooth, especially on held-out rollouts with novel distractors, whereas V-PTR produces smooth value functions. [Ghosh 23]

The conclusion of this paper was that using value function pre-training on large-scale video dataset and robot bridge dataset improves the performance of the policies learned downstreams. V-PTR outperforms prior methods of learning from video data.

The work in our project is relates with this project in the concept of pre-training the Value-Function in order to obtain performance improvement in the downstream tasks. The methodology used in our paper for pretraining the value-function is different from what V-PTR is proposing. But the main idea aligns with their paper. Detailed information regarding the pre-training phase is discussed in Section 5.

4. Tools and Platforms

To develop and train Autonomous Driving system requires different software tools and platforms to create an entire pipeline. Here, CARLA was taken as the simulation environment in which the autonomous agent would navigate. There are already implemented Reinforcement Learning algorithm in Stable Baselines. Training was done using the reliable implementations in stable-baselines3. OpenAI Gym toolkit provided an environment for developing and testing learning agents for deep learning models. It provides a common interface for interacting with reinforcement learning environments.

4.1 Simulator

In case of Autonomous Vehicle systems, the main difficulty faced during research is that of training the models in real-world scenario. Training a vehicle to learn to drive autonomously directly in real-world encounters many problems from all the unseen possibilities. Also, training and testing of the vehicle is very expensive in real-world. Thus, before testing it in real-world it needs to be trained in the driving simulator where the agent goes through many different scenarios that might be possible in the real world. There are many simulators available like Carla [Dosovitskiy 17], AirSim [Shah 17], LGSVL [Rong 20], Webots [Webots], TORCS [Torcs 07], and rFpro. This project uses Carla simulator since it is open-source and free platform, and allows to configure various different experiments on the basis of requirements.

CARLA (Car Learning to Act)

CARLA [Dosovitskiy 17] was released in the year 2017, and it is an open-source simulator for tasks related to autonomous driving. It is an open simulator for urban driving and developed to support training, prototyping and validating already trained autonomous systems. It is built on Unreal Engine 4. The simulator provides lot of control over the simulation environment like weather, time of the day, and traffic. In Figure 4.1, there are different weather settings used for the same location, from clear day, rain, daytime after rain and clear sunset.

Figure 4.1: Different weather conditions in a street in Town02 from third-person view. [Dosovitskiy 17]

CARLA simulates a world and it provides an interface between the environment and an agent which interacts with the world. It has a server-client system. Here, the server runs the simulation and provides an environment for the agent to interact in. An API is used for the interaction between the agent and the environment. This API is implemented in python. Here, the client sends actions or commands to the server (like accelerate, steer, and brake) and in response to a particular action the agents state in environment changes and accordingly receives readings from the sensors. There are lot of sensors that can be used in CARLA environment such as, RGB camera, Radar, LiDAR, segmentation camera, and collision detection sensor. On the basis of the actions, readings from the sensors are received via the API.

4.1.1 Autonomous Driving

CARLA provides the support to develop, train and evaluate autonomous driving systems. There are three approaches that can be used with CARLA in order to train autonomous systems [Dosovitskiy 17]. The first way is of a modular pipeline that is dependent on different subsystems such as visual perception, planning and control. Secondly, train deep network using imitation learning. And at last, training a deep neural network using reinforcement learning methodologies.

As already explained in Section 2.3.2, the different components can be described in terms of CARLA simulation environment. The autonomous vehicle acts as an agent in the simulated world, which gets an observation o_t from the environment and must take an action a_t. The actions are steer, throttle and brake. Here the observation or state is obtained from the sensors attached to the vehicle. The environment is the city in which the agent has been spawned.

4.1.2 Towns

There are eight maps in CARLA ecosystem and each consists of two types of map. Non-layered and layered maps. Layers means the grouped objects in the map such as, NONE,

buildings, decals, foliage, ground, parked vehicles, particles, props, street lights, walls and all. In non-layered maps there are many options. All of the layers are present and cannot be removed from the map. Towns include from Town01 to Town12. Whereas in layered maps, according to use-case the layers can be toggled off or on. One of the layer map is Town01_Opt. In this project we are going to use map Town02 which is a simple small town with residential and some commercial buildings.

4.2 Tools

CARLA provides the environment for the agent, now there is a requirement of tools that will make it possible to interact with the environment as well as train reinforcement learning algorithm. To enable this, two tools or frameworks are used in this project: OpenAI Gym and Stable-Baselines3.

4.2.1 Stable-Baselines3

Stable-Baselines3 is an library with already implemented open-source deep reinforcement learning algorithms. These algorithms are implemented in python. It is an open-source framework with seven common model-free RL algorithms. Main goal of the framework is to keep it user-friendly and reliable RL library. Some of the features of the library are that it provides a simple API using which only few code of lines will enable the interaction with the environment. This makes it easy for the research purpose. It is a well documented framework, with basic and advanced usage with some examples. In addition to that, this framework provides support for parallelization and distributed training which enables faster training and resources used up to its full capacity. In respect of this project, Stable-Baselines3 has following state-of-the-art on-policy and off-policy algorithms: A2C, PPO, DDPG, SAC, TD3, HER and DQN (refer Section 2.4.4) from which SAC algorithm is used to train the autonomous car. These algorithms are widely used in various applications and proved to be highly effective in those scenarios. In Table 4.1, implemented algorithms are listed with the allowed action spaces in that algorithm. [Raffin 21]

Name	Box	Discrete	MultiDiscrete	MultiBinary
ARS	✓	✓	×	×
A2C	✓	✓	✓	✓
DDPG	✓	×	×	×
DQN	×	✓	×	×
HER	✓	✓	×	×
PPO	✓	✓	✓	✓
QR-DQN	×	✓	×	×
RecurrentPPO	✓	✓	✓	✓
SAC	✓	×	×	×
TD3	✓	×	×	×
TQC	✓	×	×	×
TRPO	✓	✓	✓	✓
Maskable PPO	×	✓	✓	✓

Table 4.1: Implemented algorithm in Stable-Baselines3 and the allowed action spaces. [Hill 18]

4.2.2 OpenAI Gym

Gym is an open-source Python library which is used from developing and comparing reinforcement learning algorithms. This provides the communication between the reinforcement learning algorithm and the environments in the form of API. It provides a simple interface for interacting with the environments. There are already available environments which can be trained using RL algorithms. Using the pre-built environment (Figure 4.2), different reinforcement learning algorithms can be compared and evaluated. In addition to that, there is a possibility of creating custom environments. In this project, OpenAI Gym is used to create an interface for communication between the reinforcement learning algorithm and CARLA simulation environment. [Brockman 16]

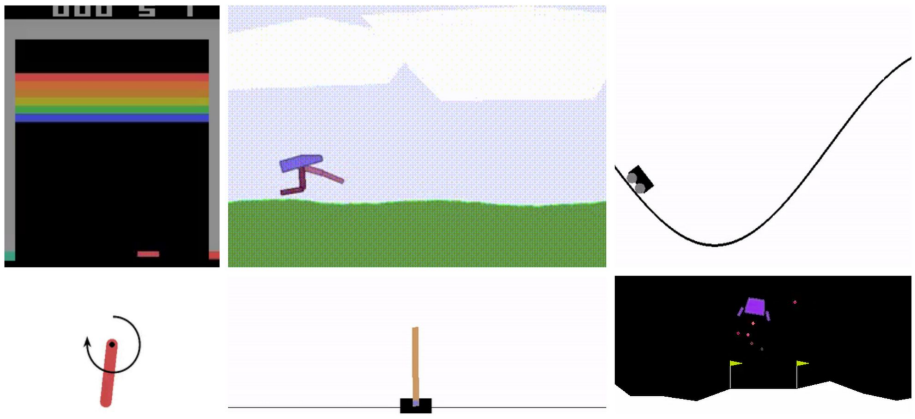


Figure 4.2: The pre-built environment in OpenAI Gym, using which different Reinforcement Learning algorithms can be trained and evaluated. [Kathuria 21]

5. Concept and Implementation

In the previous sections, the fundamental concepts required for understanding and the basic idea behind this project is described in detail. This chapter will provide the main idea and implementation methods used in the project. The explanation of implementation is majorly divided into two parts. The initial stage is related to Deep Reinforcement Learning (DRL) setup and training, which will include the explanation of connections between all the modules required for DRL and how the neural networks are trained. The second phase will be regarding using the pre-trained value-network from DRL is used as the backbone for Object detection task.

5.1 Setup related to DRL

In this work, CARLA 0.9.13 version is used. The CARLA application is launched and loaded with Town02 as shown in Figure 5.1, which consist of simple urban environment with residential as well as commercial buildings. This environment is used for training the DRL algorithm. The ego-vehicle is spawned at a possible random location for cars (roads/highway) as shown in Figure 5.2. The center line of the road serves as the guidance for the ego-vehicle. On the basis of the center line the deviation towards left or right side of the line is calculated which eventually helps the vehicle follow the lane. Training is conducted under three configurations: the first one is without any waypoints and just depends upon the distance from the middle of the road, the second involves spawning different waypoints in each new episode, and under third configuration it always spawn same set of waypoints. The waypoints spawned in the Town02 is shown in the Figure 5.3. Furthermore, pedestrians are spawned at various locations of the map to serve as obstacles for the ego-vehicle.Subsequently, the vehicle trains to navigate around the map while avoiding the pedestrians. More about the training procedure is explained in Section 5.2.5.1. In the coming subsections, information regarding the different components used for DRL and how they help in the training is presented in detail.

5.1.1 Communication between all the components

In Section 4, all the required tools and platforms were introduced. In this subsection we will look into the details of how those components are used for this project. In the Figure

Figure 5.1: Top-view of Town02 which is being used for DRL algorithm training.

Figure 5.2: Car blueprint 'model3' from carla blueprint-library spawned in Town02.

5.5, a simple flowchart is shown describing the connection and communication between all the important components for DRL. The environment provides the observations for the agent. Here the Carla Simulator gives the observations/states in the form of output from the sensors present on the ego-vehicle to the OpenAI Gym API. The OpenAI Gym module takes input from Carla and forwards it to the DRL algorithm from Stable-baselines3

Figure 5.3: Waypoints spawned in the environment are denoted by red markers, which acts as a guiding path.

Figure 5.4: Around 100-200 pedestrians are spawned at different location in Town02.

framework. Using those observations, updates are made to the parameters in neural network of DRL algorithm and Control actions are generated as the prediction. Those generated actions are given back to the agent in the environment through OpenAI Gym and changes in the environment/state are made. The complete sequence involving the

OpenAI Gym receiving the state from simulator/environment, followed by sending back the control to the CARLA simulator, is considered as a singular action, and is called as a Step (Section 5.2.3), which is taken by the agent. Multiple such steps constitute an episode.

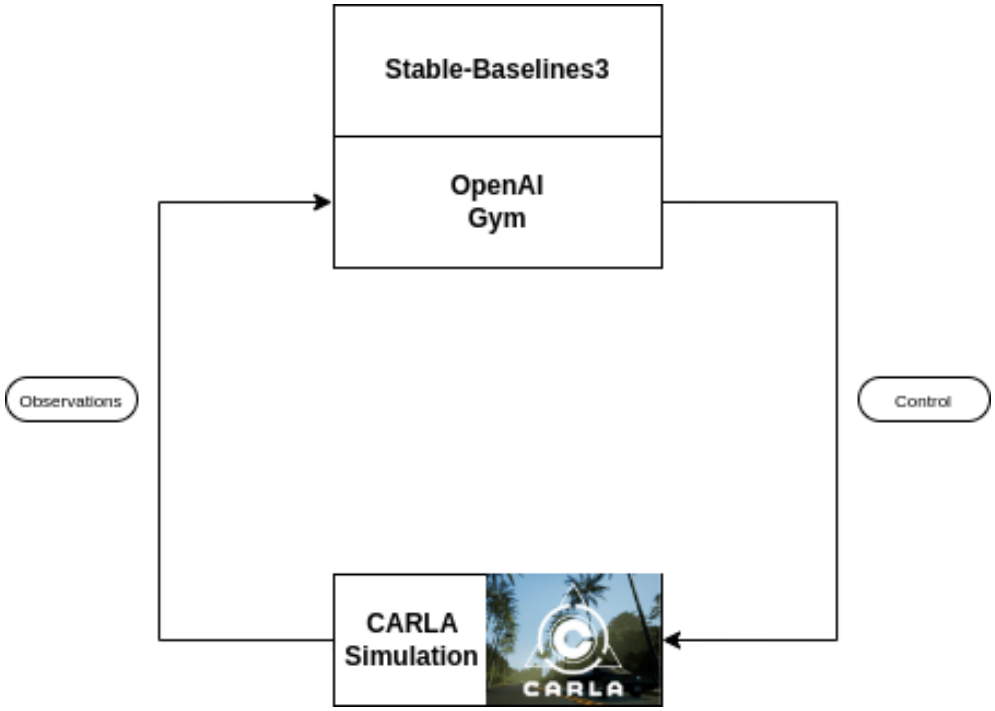


Figure 5.5: The important components for the training of the RL algorithm. CARLA (simulator), OpenAI gym (connecting the RL algorithm and Carla), and Stable-Baselines3 (providing the RL algorithm).

5.1.2 Calculating total distance covered

In order to keep track of progress made by the ego-vehicle, one of the most important parameter that needs to be calculated is the total distance covered by it. The total distance is calculated throughout an entire episode. The distance obtained at the end of the episode is considered as the total distance traversed during that episode. In Carla, there are methods implemented which facilitates to provide with crucial information for calculating distances. First, the vehicle is spawned at some location in the town. The initial location is stored in a variable for further calculations. At every step, the current location of the vehicle is fetched using Carla method "vehicle.get_location()". This method returns the current location of any actor. Algorithm 5.1 describes the distance calculation in n steps. Here n is the number of steps that has been taken in that particular episode.

It is straightforward to calculate the distance travelled due to methods available with Carla Simulator implemented in Python. The main objective is to maximize the distance travelled.

Algorithm 5.1: Total distance calculation

Input: vehicle (actor)
Output: Total distance covered by the ego_vehicle
Initialize: previous_location = vehicle.get_location(), total_distance = 0

```
for step = 1 to n do
    current_location = vehicle.get_location()
    total_distance += current_location - previous_location
    previous_location = current_location
end
```

5.1.3 Getting nearest pedestrian distance

The system needs to know the distance of the vehicle from the nearest pedestrian on the map. This knowledge is required for further processing and training of Reinforcement Learning algorithm. Firstly, as described in the Algorithm 5.2, the list of all the pedestrian actors spawned in the Carla environment is already stored in a "walker_list" and vehicle_location is fetched. The location of all the pedestrian is stored in a new list. That list is then iterated and distance between the vehicle location and individual pedestrian is calculated using x, y, and z component of both the actors. After iterating through each of the pedestrian, the output is the distance between the vehicle and the nearest pedestrian.

Algorithm 5.2: Distance to nearest pedestrian

Input: vehicle, world, walker_list (stores all pedestrians)
Output: dist_to_nearest_pedestrian
Initialize: vehicle_location = vehicle.get_location() and list_of_pedestrian = []

```
for i in walker_list do
    list_of_pedestrian.append(i.get_location())
end
nearest_distance = float('inf')
for i in list_of_pedestrian do
    dx = vehicle_location.x - i.x
    dy = vehicle_location.y - i.y
    dz = vehicle_location.z - i.z distance = math.sqrt(dx2̂ + dy2̂ + dz2̂)
    if distance < nearest_distance then
        nearest_distance = distance
    end
end
```

5.1.4 Calculating the deviation from middle line

To calculate the deviation, the lateral distance from a vehicle to left and right road lines are calculated. The distance is calculated with the help of the vehicles current location in the map, the angle, and distance from the waypoints on the Carla map. It considers the orientation of the vehicle and calculates the distance accordingly. As shown in Algorithm 5.3, the first step is to fetch the location of the vehicle using "vehicle.get_location()". The nearest waypoint from the vehicle is used to get the width of the lane. The bounding box of the vehicle is fetched and then the distance from left and right is calculated using the corners of the bounding box.

The angular difference between orientation of vehicle and that of waypoint corresponding to one of the corners is calculated. The calculation of distance to left and right depends on certain conditions of angle_diff. If the angle_diff meets the condition mentioned in the Algorithm 5.3, it means that the corner it is comparing with is on the left side of the road else the corner is on the right side of the road. The method used here to calculate the distances to left and right is on the basis of the orientation of the bounding box corners of the car with respect to the current road it is at.

Algorithm 5.3: Deviation from middle line [yanlai00 20]

```
Input: vehicle (actor), carla_map
Output: Distance to left from vehicle, distance to right from vehicle, and nearest
        waypoint.
Initialize: dist_to_left=100, dist_to_right=100
current_location = vehicle.get_transform().location
way_point = carla_map.get_waypoint(current_location)
lane_width = waypoint.lane_width
bb = vehicle.bounding_box
corners = bb.get_world_vertices(vehicle.get_transform())
for corner in corners do
    if corner.z < 1 then
        waypt = carla_map.get_waypoint(corner)
        waypt_transform = waypt.transform
        waypoint_vec_x = waypt_transform.location.x - corner.x
        waypoint_vec_y = waypt_transform.location.y - corner.y
        dis_to_waypt = math.sqrt(waypoint_vec_x ** 2 + waypoint_vec_y ** 2)
        waypoint_vec_angle = math.atan2(waypoint_vec_y, waypoint_vec_x) * 180 /
         math.pi
        angle_diff = waypoint_vec_angle - waypt_transform.rotation.yaw
        if (angle_diff > 0 and angle_diff < 180) or (angle_diff > -360 and angle_diff <
         -180) then
            dis_to_left = min(dis_to_left, waypoint.lane_width / 2 - dis_to_waypt);
            dis_to_right = min(dis_to_right, waypoint.lane_width / 2 +
             dis_to_waypt)
        end
        else
            dis_to_left = min(dis_to_left, waypoint.lane_width / 2 + dis_to_waypt);
            dis_to_right = min(dis_to_right, waypoint.lane_width / 2 - dis_to_waypt)
        end
    end
end
```

5.1.5 Setting the controls of the ego-vehicle

The actions predicted by the Stable-baselines3 RL algorithm, as depicted in Figure 5.6, serves as the output actions. These actions undergo processing within the OpenAI Gym module before being forwarded to the Carla environment, where those actions are applied to the ego-vehicle.

The actions prediction consists of two components: throttle and steer. Both of these outputs the predicted values within the range of [-1, 1]. Here, the value for steering signifies

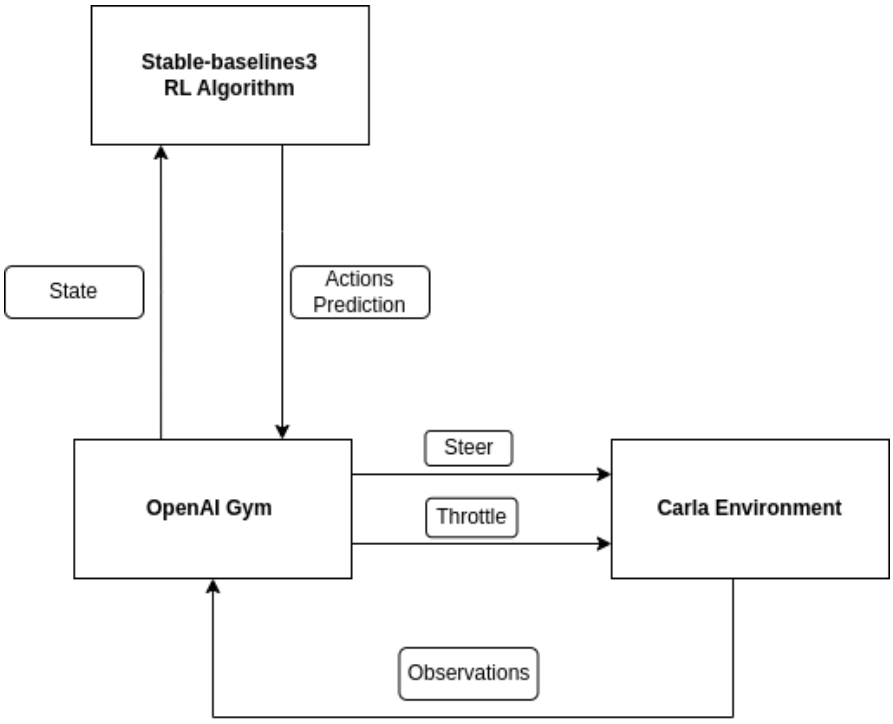


Figure 5.6: The actions are predicted by the algorithm implemented under Stable-baselines3 library. Those actions are applied on the vehicle in the carla server using OpenAI gym api.

the steering angle of the vehicle. As for the throttle, if the predicted value is greater than 0, the throttle is assigned the value directly and brake is set to zero. Conversely, if the predicted value is less than zero, throttle is set to zero, and brake is assigned the magnitude of the predicted value. In Algorithm 5.4, while applying the steer, throttle and brake values to the vehicle, a weighted combination is use. The previous actions are given a weight of 0.9,while the current actions are assigned a weighted of 0.1. This scheme is implemented to achieve smoother steering and acceleration/braking. The method *vehicle.apply_control*() is an already implemented method used for applying controls to the actor.

Algorithm 5.4: Setting the steer and throttle

```
Input: action (predicted output from RL algorithm), carla, and vehicle
Initialize: previous_steer=0.0, previous_throttle=0.0, and previous_brake=0.0
for i in steps do
    throttle = float(action[0]) if action[0] > 0 else 0.0
    brake = float(-action[0]) if action[0] < 0 else 0.0
    steer = action[1]
    steer = max(min(steer, 1.0), -1.0)
    vehicle.apply_control(carla.VehicleControl(steer=previous_steer*0.9 + steer*0.1,
                throttle=previous_throttle*0.9 +throttle*0.1,
               brake=previous_brake*0.9 + 0.1*brake)
          )
    previous_steer = steer
    previous_throttle = throttle
    previous_brake = brake
end
```

5.2 Environment Setup

OpenAI Gym is a toolkit is a platform that provides an environment to train the agent inside it. There are already implemented environments like car racing, cart pole, and lunar lander where different RL algorithms can be trained and evaluated. To train the autonomous vehicle inside Carla Simulation, a CustomEnv (in this project : CarlaEnv) class is created. To create a this custom environment, parameters such as action_space, state_space, and reward functions are defined. More about it in the upcoming subsections. There are some essential methods that are defined under CarlaEnv, such as initialise method, step method, reset method, and reward function(s). In this project, the OpenAI Gym enables the communication with the Carla Simulator. The observations are fetched from the Carla Simulator which is then processed with the help of CarlaEnv class and the methods defined under it. After processing, the observations are forwarded to the RL Algorithm, which utilizes it for training the deep neural network. The predictions from the algorithm are the control actions, that are used as the input for the agent present in the environment. After applying the controls to the agent, it results into the change in the location of the vehicle and therefore change in the state/observation. Here, the actions, rewards, and states are the guiding factors for the training of the RL algorithm.

5.2.1 Initialise method

The ___init___() method is used for initializing all the important variables and spaces that are used for fetching observations, training algorithm and sending actions back. In this project, the very first task in this method is to setup the Carla Environment. This includes, launching of the Carla Simulator, spawning the Town of choice (Town02) and then fetching other relevant data required from simulation environment. Secondly, the observation space and the action space are defined. The action space consists of the values of throttle and steer. Both the actions, throttle and steer can have the values in the range between -1 and 1. The observation space here has a resized image whose values vary between 0 and 255. Other important variables facilitating the training are initialized in this method. Variables such as, car model, list of sensors, steps per episode, maximum distance, blueprint library, and many more. In order to spawn pedestrians in the carla environment, the create_pedestrian flag should be set to 'True'. According to the requirement, 'n' number of pedestrians can be spawned in the Carla Environment using this flag.

5.2.2 Reset method

The main task of this method is to revert all the settings to its initial values in order to start a new episode. This method is evoked after the variable "done" is set to 'True' due to some event. In this project, the parameter done is set to 'True' when it reaches a failure state or terminal state. Here, failure consist of vehicle collision, either with some object on the road or with pedestrians, exceeding the decided speed limit, or not following the lane, and terminal state is considered when it reaches the maximum episode length or destination. In all the cases, the environment requires a reset. Firstly, the car is reset in terms of its spawn position. It spawns at some random position and starts navigating from there. Then there are sensors which needs to be attached to the vehicle. Those are

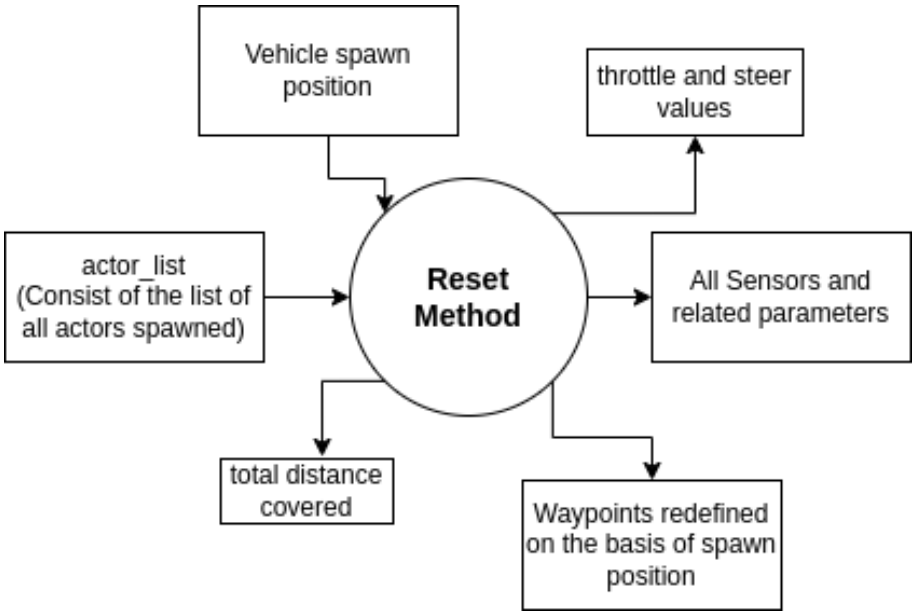


Figure 5.7: Some of the parameters that are set to initial values when reset method is invoked.

defined in this method. The waypoints are defined on the basis of the spawn position of the vehicle. So, those waypoints needs to be generated again whenever new episodes starts. Figure 5.7 shows some of the variables that are set to their original values or assigned a new value. The ___init___() method is called first and at completion of every episode reset() method is invoked.

5.2.3 Step method

An episode can have a particular maximum number of time-steps. And for each time-step, the step method is called. The function of step method is to update the relevant values at every single time-step of the episode and progress the simulation by one time-step. This method at each time-step receives the actions from the RL model as an input and apply it to the vehicle in the Carla Environment. This results into the change in the state/observation of the agent. Hence, this method is crucial for interaction with the RL agent and the environment, allowing the agent to learn by taking actions and then receiving new states in this method. The working is further explained in the upcoming section and shown in Figure 5.8:

Inputs

The inputs are from two sources: RL model and Carla Simulator. From model, actions are received as a parameter and then applied to control the throttle and steering of the vehicle. And then image is received from the Carla Simulator. This image is processed inside this method and at the end returned to the caller.

Applying the controls

The predicted outputs from the model are used to control the car. The actions are sent to the simulator and applied on the agent using the $vehicle.apply\_control(throttle, steer, brake)$ method.

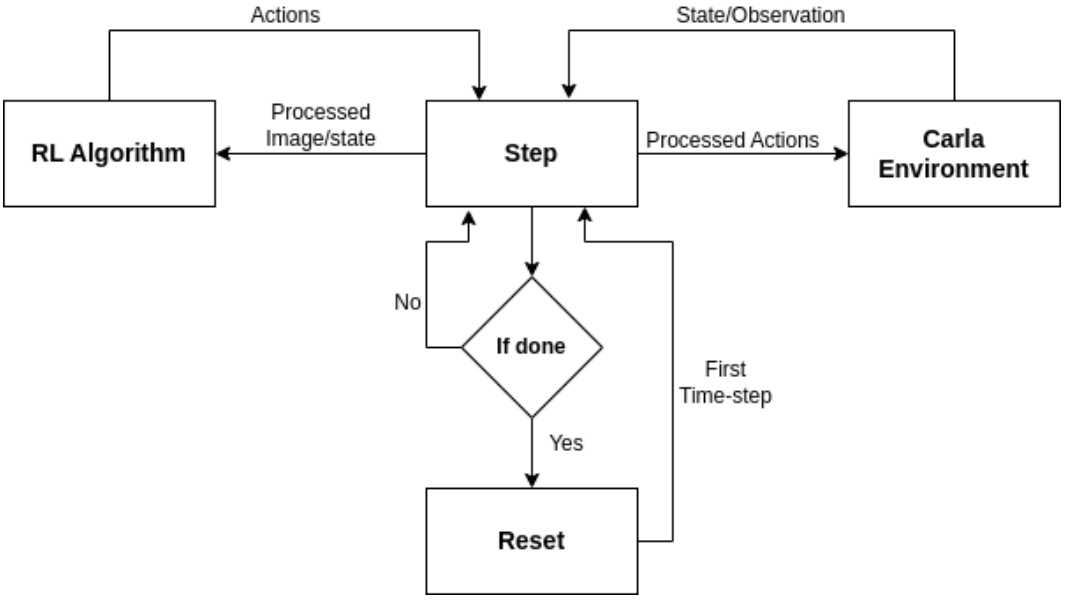


Figure 5.8: In the step method, the actions and observations are obtained as the input and after processing it, those actions and observations are forwarded to Carla Environment and RL algorithm respectively. Actions are then applied to the car and observations are taken as input by the RL algorithm. The reset method is evoked if done is set as True.

Reward as feedback

The RL model obtains feedback in step method during each timestep. In each timestep, on the basis of the current state and the actions taken, calculations are performed to obtain a value that determines whether the action taken in that state was good or bad. That value is used as a feedback for improving the model weights. The calculations related to rewards are further discussed in Section 5.2.4.

Other important parameters

In order to get the rewards, there are several calculations to be done. There are parameters to be defined before those calculations. For this purpose, values such as distance from center line, speed (in kmph), tracking closest waypoint, collision_detection, lane deviation, and total distance covered (in each time-step the distance is accumulated) are some important values that needs to be computed. These values are computed at each time-step and updated for use in the next time-step.

Return (obs, reward, done, info)

At the end of step method, the current observation/state, the reward for the current time-step, the done flag, and other crucial information regarding the environment are returned to the caller. Here, the observation is the image obtained from the simulator that has been processed in step method and then will be given to the RL model as an input. Then, the reward is the value calculated using parameters defined earlier. The value of done flag totally depends upon the action taken in this time-step and it indicates whether the episode has ended or not.

5.2.4 Reward Functions

Reward function acts as a guide for the training of the RL model. It penalizes the for undesirable actions from the vehicle and assigns reward for desirable ones. As discussed in Section 5.2.3, there are several parameters that affects the penalty or reward calculations. There are three different types of reward functions utilized for this project. This section defines the main parameters used for each reward function. The parameters considered by the reward functions are as follows:

- Distance from Pedestrian: The desirable behavior for the car is to drive carefully and maintain a safe distance from the pedestrians present in the environment. Based on the distance from the pedestrian, the penalties are defined.
- Velocity (kmph): In this project, the target speed, minimum speed and the maximum speed for the vehicle is defined. The agent is penalized if the speed is maintained lower than the minimum speed or higher than the maximum speed defined.
- Deviation from Midline: If the distance of the vehicle moves away from the midline by a particular margin then it is considered unfavorable and the penalty is given accordingly.
- Deviation from Waypoint: Waypoints are different from the midline of the road. They are similar but restrictions by waypoints are more tighter than the distance from midline.
- Collision with objects: To detect the collision of the vehicle with objects in the surrounding, there is a collision sensor being used. At all cost the agent should avoid colliding with the objects. So, on collision the agent is penalized heavily.
- Total time Step: There is a fixed maximum number of time-step set for the simulation. If the vehicle reaches this state, it means it has achieved the terminal state. On reaching this state agent is rewarded.
- Slow speed timer: If the vehicle doesn't move from one point for a long time, then it should be penalized. Because the goal is to cover larger distance while avoiding collisions.
- Total Distance Travelled: In each time-step the distance travelled is accumulated. The more distance the vehicle is able to travel, the more reward the agent receives.

Above mentioned parameters are utilized differently in the following reward functions. The usage of the reward functions while training is described in 5.2.5. The following sub sections states the parameter being used in each function.

5.2.4.1 Lane assist Reward

In this function, the parameters utilized are: velocity, deviation from midline, collision, total time step, slow speed timer and total distance travelled.

5.2.4.2 Waypoint Reward

All the parameters used in this function are similar to that of Lane assist reward function, but instead of 'deviation from midline' parameter, deviation from waypoint is being utilized. This parameter is stricter; it penalizes more for small deviations of vehicle from the lane in which the car is spawned.

5.2.4.3 Pedestrian Reward

In this function, in addition to all the parameters from 'Lane assist reward' function, distance from pedestrian is added as an additional parameter. Depending on the distance the penalties are weighted.

5.2.5 RL model Training

The model used in all the experiments is a popular Reinforcement Learning algorithm called as SAC (Soft Actor-Critic). It uses an actor-critic architecture, where the actor-network predicts the actions that needs to be taken by the autonomous vehicle and the critic network assigns values to the actions. The value defines the quality of the actions taken. In this project of training an autonomous car, an algorithm is required can work with continuous action space. The SAC and PPO algorithm can be used for continuous action spaces applications. Here, SAC is an off-policy algorithm whereas PPO is an on-policy algorithm. Off-policy algorithms like SAC are more sample-efficient than on-policy algorithms. In case of off-policy, the earlier experiences of the agent are being stored and can be fetched randomly for training purposes. In SAC paper [Haarnoja 18], it was demonstrated that the combined objective of maximization of rewards and maximizing the entropy has lead to achieve state-of-the-art performance on range of continuous action control benchmark tasks, outperforming prior on-policy as well as off-policy methods.

5.2.5.1 Training Phase

The training is performed using SAC algorithm from Stable-baselines3 (SB3) library. The actions; throttle and steer can have values between -1 and 1. All the reinforcement learning based experiments are trained on one Nvidia GeForce RTX 2080Ti GPU. There are two different deep neural networks involved in the architecture of the SAC RL algorithm. One is the Actor Network, which is responsible for taking the actions, and the second is the Critic Network, responsible for assigning values on the basis of the quality of the actions.

The default network architecture used by SB3 depends on the algorithm and the observation space. For 1D Observation space, a 2 layers fully connected net is used with 256 units for SAC algorithm. The off-policy algorithms implemented in SB3 have separate feature extractors: one for the actor and one for the critic. The first experiment is conducted using the base network available with the SB3 library. Then custom feature extractors are used for other experiments. In custom feature extractor, ResNet18 architecture and ResNet34 architecture is being utilized. As shown in Figure 5.9, there are two parts for the network, first part is the features extractor and second is the fully-connected network. Here, the features extractor network is replaced by either ResNet18 or ResNet34. And then this network is trained further. After completion of the RL training, those feature extractor network of ResNet18 (or ResNet34) are considered as the pretrained network

for the object detection training phase which is described in Section 5.5. In this project, Pytorch [Paszke 19] is choosen as the Deep Learning framework. It is written in Python. Because of its extensive community, Pytorch has gained credibility as a reliable machine learning framework for many research-oriented works. The RL algorithm is trained for 1,500,000 time steps. From section 5.2.4, the reward functions are defined and the training is divided into three phases on the basis of reward function. Each phase is trained for 500,000 time steps with different reward function. The main metric used for tracking the progress is the rewards accumulated, total distance covered, and for how many time steps (ep_len_mean) was the vehicle interacting with the environment. The Figures 5.10, 5.11, and 5.12 are instances of the training progress in terms of rewards accumulated, distance traveled (graph is smoothed out to see the trend), and episode length respectively from the very first experiment with ResNet18 as feature extractor architecture. Along the x-axis is the total time steps elapsed and y-axis is the rewards accumulated, distance traveled and episode length in Figures 5.10, 5.11, and 5.12 respectively. All these measurements are tracked using Tensorboard.

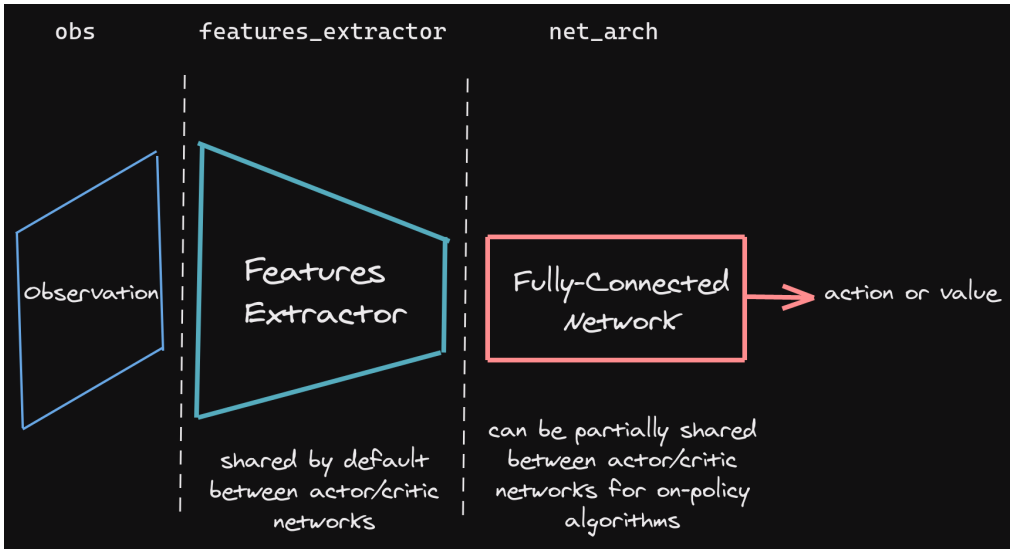


Figure 5.9: The architecture of actor-network and critic-network implemented in Stable-baselines3 library. For actor and critic-networks the output will be action and value respectively. [Stable-baselines3]

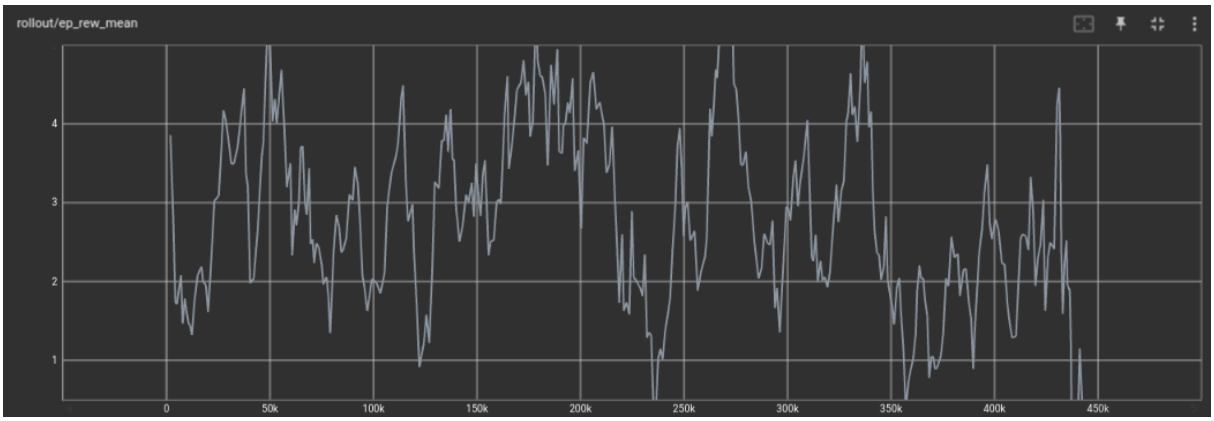


Figure 5.10: Episode reward mean variation during the training phase.

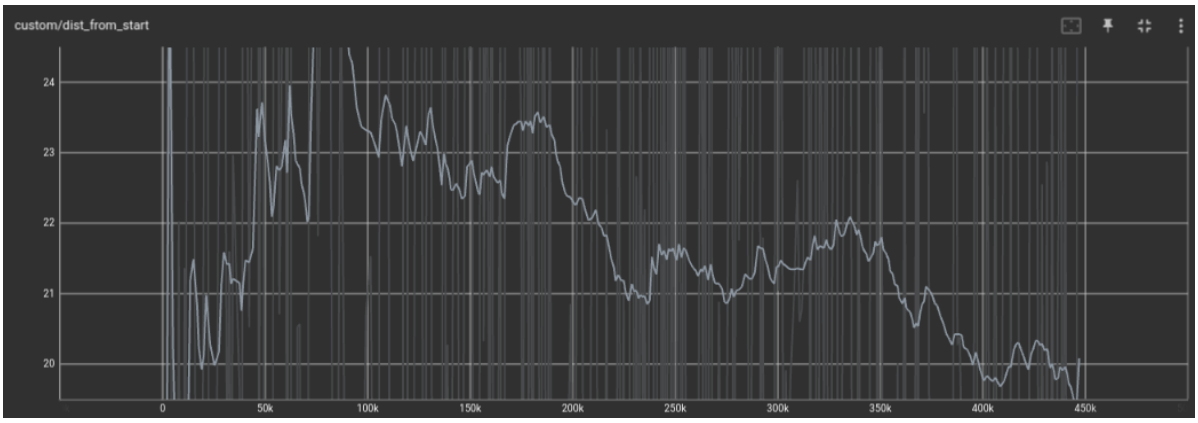


Figure 5.11: Total distance travelled in every episode during training phase.

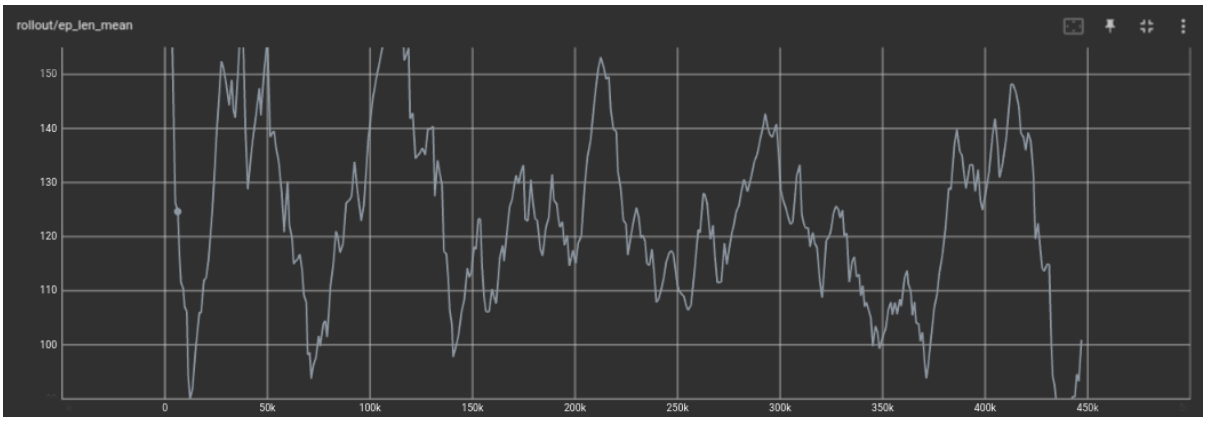


Figure 5.12: Episode length mean variation during the training phase.

5.2.5.2 Rewards

The reward functions and the parameters used in those functions were defined in Section 5.2.4. In this sub-section, the magnitude of the rewards or penalty is defined for each of the parameters affecting the total rewards eventually. The Table 5.1 describes the rewards based on different conditions resulting into end of the episode. There are penalties for car autonomously driving too close to the pedestrians, where the penalties are weighted on the basis of distance to the pedestrian. As shown in Table 5.2, if pedestrians are really close to the ego-vehicle then the penalty is higher.

The Table 5.3 defines the rewards when it exceeds a particular maximum speed limit. Other parameter used for giving rewards to the agent is the total distance covered from start during an episode. The distance is added after multiplying by 2 to the cumulative reward if it surpasses the previous maximum distance travelled by the agent. This particular reward will motivate the agent to take actions in such a way that it maximizes the rewards by traversing more distance. The last parameter affecting the reward is calculated at each time step. Those rewards are assigned only if the current episode hasn't been terminated, the rewards are added on the basis of the distance from the center and the angle from the road. These rewards are defined in Algorithm 5.5.

Reset Condition	Reward
Car collides with some object (collision_history > 0)	-50
if (dist_to_left > max_dist_from_center) or (dist_to_right > max_dist_from_center)	-30
Vehicle not moving for 10 secs (slow_speed_timer > 10 and speed < 1)	-10
Episode runs more than simulation length	20

Table 5.1: Termination condition and corresponding rewards.

Distance from Nearest Pedestrian	Penalty
nearest_pedestrian_distance < 3.0	1.25*penalty
3.0 < nearest_pedestrian_distance < 5.0	0.25*penalty

Table 5.2: Weighted penalty on the basis of the distance from the pedestrians.

Parameters	Reward/Penalty
speed > max_speed	-10
dist_from_start > max_distance_covered	2 * dist_from_start

Table 5.3: Penalty for exceeding speed limit and reward for traversing more than previous maximum distance.

Algorithm 5.5: Assigning rewards on the basis of speed, distance from center and angle between vehicle and road.

```
Input: distance_from_center, max_distance_from_center, angle, done,
       speed, reward, target_speed, min_speed
Output: reward
Initialize: centering_factor = max(1.0 - distance_from_center /
            max_distance_from_center, 0.0)
            angle_factor = max(1.0 - abs(angle / deg2rad(20)), 0.0)
if not done then
    if continuous_action_space then
        if speed < min_speed then
            reward += (speed / min_speed) * centering_factor * angle_factor
        end
        else
            if speed > target_speed then
                reward += (1.0 - (speed-target_speed) / (max_speed-target_speed)) *
                          centering_factor * angle_factor
            end
        end
        else
            reward += 1.0 * centering_factor * angle_factor
        end
    end
    else
        reward += 1.0 * centering_factor * angle_factor
    end
end
```

5.3 Dataset

For RL algorithm, the input or the observations are directly obtained from the Carla Simulator. Conversely, the object detection algorithm is trained and tested on a dataset generated using Carla simulator. The dataset generated using Carla is a synthetic dataset. The annotation for dataset are in COCO format for all the experiments. The dataset consists of one category, i.e. pedestrian, and the label file consists of the bounding boxes for all the pedestrians present in a particular frame, as the goal of this task is to localize the pedestrians in the environment. The dataset is split into training and testing dataset in 80:20 ratio.

5.3.1 Synthetic Dataset

For this work, all the dataset is generated using Carla Simulation environment. The same town, Town02, is used for creating dataset which was used for Reinforcement Learning algorithm. The script used for creating the dataset for pedestrian task is inspired by the work by Mukhlas Adib [Adib 20]. The script uses the advantage of the APIs provided by the Carla Simulator implemented in python. Objects (pedestrians in our case) within the CARLA simulation all have a bounding box and the CARLA Python API provides functions to access the bounding box of each object [Dosovitskiy 17]. The bounding boxes obtained by the function provided by Carla returns 3D bounding boxes. This project requires 2d bounding boxes. The script fetches 3D bounding boxes of the pedestrians in the field-of-view of the camera spawned on ego-vehicle and converts them to 2d Bounding boxes and saves the annotation. The ego-vehicle is made to traverse in the Carla Environment, and the images and the corresponding bounding boxes are saved after every 10 ticks of interval to create the dataset. The Figure 5.13, shows one of the snapshot from Carla simulation environment. Here, Figure 5.13a, is the image without the bounding boxes, which will be used as input during the training of object detection task. On other hand, the Figure 5.13b shows the same image with the bounding boxes generated using the python script. There are two different datasets created for training purposes which are explained in the next section.

(a) Image **(b)** Label

Figure 5.13: Image (a) is the snapshot from the Carla Environment with pedestrians spawned in the environment, and Image (b) is the same image with the ground truth or the bounding boxes generated using the script.

5.3.1.1 Training with all pedestrians blueprints

As shown in Figure 5.14, there are various blueprints provided by Carla that can be used to spawn different types of pedestrians in the simulation. There are 49 different blueprints which consists of Male, Female, Children, Policemen and different versions of each of them. In the first dataset, for both RL algorithm and object detection task, pedestrians are spawned using all the blueprints. So, all the different version of pedestrians are shown to the model when using this dataset.

5.3.1.2 Training with some pedestrians blueprints

Contrary to the earlier dataset, total 49 blueprints are randomly divided into training and testing set following the ratio of 80:20. So, during training only few variations of the pedestrians are shown to the model (both RL and object detection). During the training phase of Object Detection algorithm, only the blueprints from the 80% of total blueprints are used to spawn the pedestrians and rest 20% are used for testing phase of the models. Therefore, unseen variation of pedestrians are shown to the model during testing.

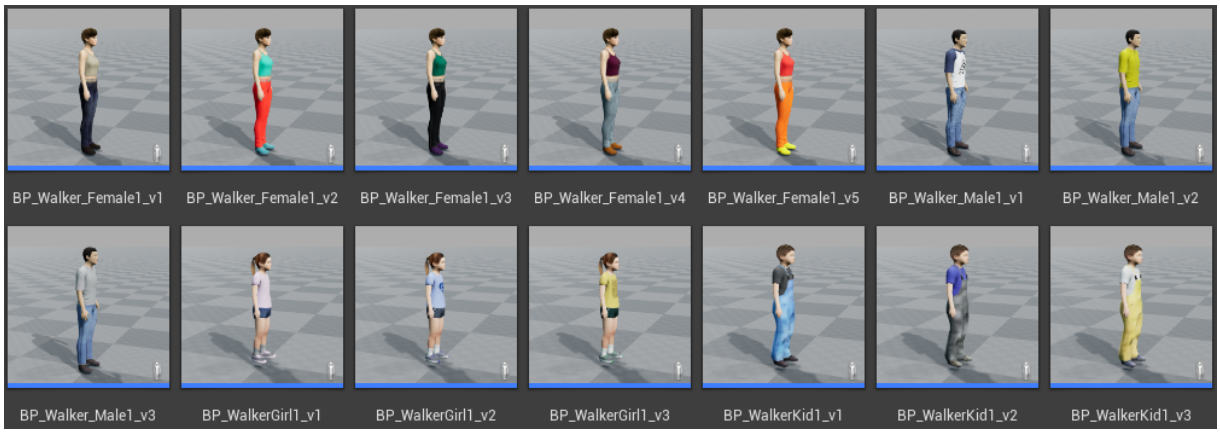


Figure 5.14: Different blueprints of pedestrians available in Carla.[Carla]

5.4 Object Detection

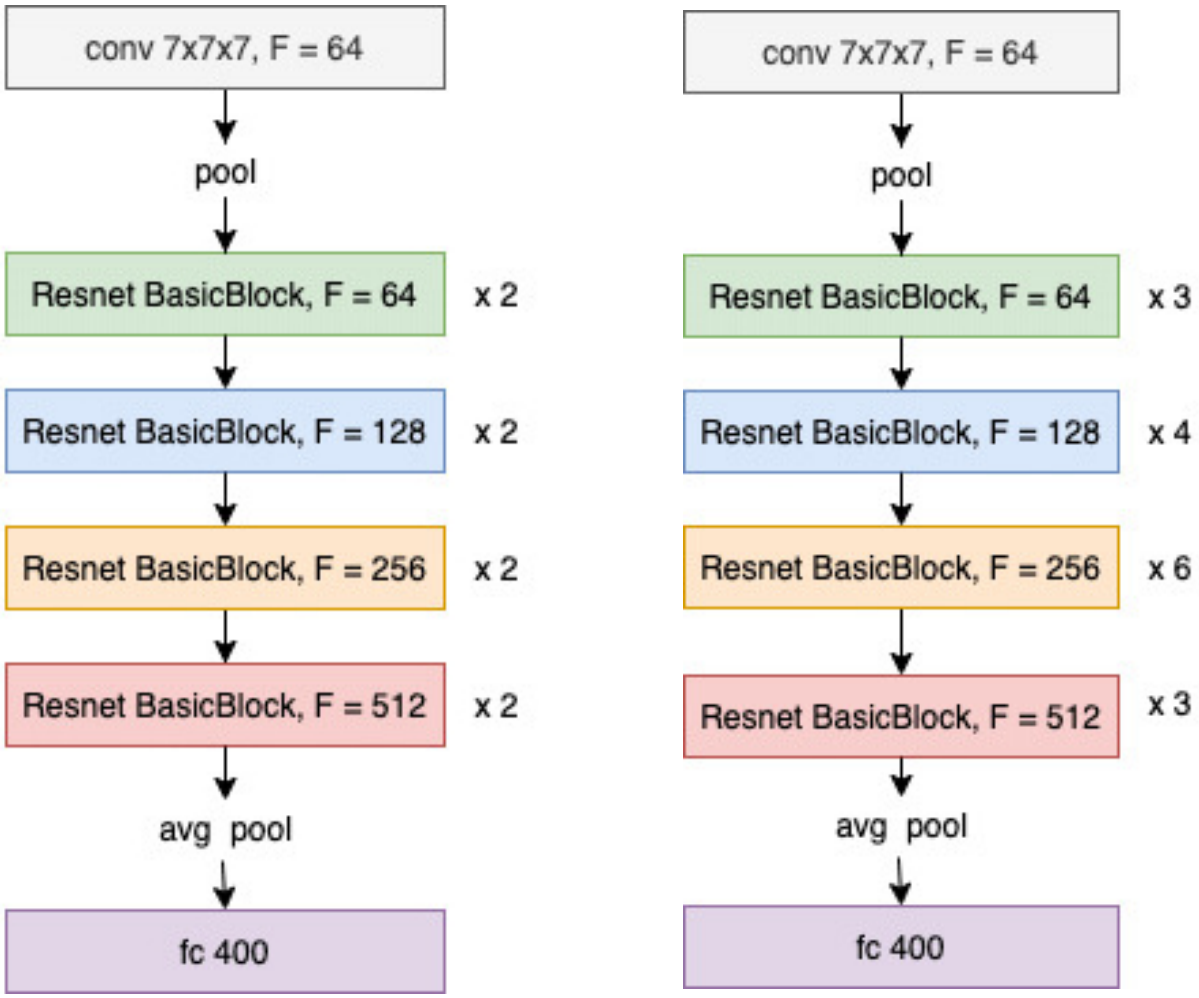
Faster-RCNN is introduced in the Section 5.4. It is a two-stage detector that has been prominently being used in various applications. The Faster RCNN detector added a region proposal network (RPN) which directly proposes regions directly. This majorly contributes in making the network faster compared to previous architectures. Therefore, makes it robust and fast enough to be used real-time applications. Originally, VGG-19 was used as the backbone, but later on ResNet was found to perform better. Hence, it became the base feature extractor for this architecture. In this thesis, ResNet-18 and ResNet-34 are used as the feature extractor for the object detection task.

5.4.1 Backbone

The backbone or feature extractors used in this thesis are ResNet18 and ResNet34. The basic architecture of ResNet18 and ResNet34 is shown in Figure 5.15a and 5.15b respectively. As mentioned in the Training phase of RL (Section 5.2.5.1), these backbones are pretrained during the training of RL algorithm.

5.4.2 Regional Proposal Network (RPN)

RPN is an independent network used for proposing regions of interest instead of selective search algorithms used previously. Along with the proposal, it also predicts the score for the objects. The input for the RPN are the features extracted using the backbone mentioned earlier (ResNet architecture). The RPN architecture uses predefined anchor boxes of various scales and aspect ratios. It uses a sliding window approach, where the sliding window is placed over the convolutional feature map to predict objecteness score and to make adjustments to the bounding boxes. High scoring anchor boxes are selected and then processed through RoI pooling and in final stages to predict class of the object and the final coordinates of the objects. This are done by passing the outputs of the intermediate layers to two different fully connected layers. One is for classification and other for regressing.



(a) ResNet18 **(b)** ResNet34

Figure 5.15: The architecture of ResNet18 and ResNet34 [Jain 21]

5.4.3 RPN Anchors

Using the combination of scales and aspect ratios a bounding box is obtained which is considered as the RPN Anchors. The aspect ratios and scales used in this thesis for

creating anchors are:

$$scales = (32, 64, 128, 256, 512)$$
$$aspect\ ratios = (0.5, 1.0, 2.0)$$

Using the above mentioned values, anchor boxes are generated. For instance, let us say that k anchor boxes are generated. If there are k anchors, the regression gives $4k$ outputs and the classification layer gives $2K$ number of scores for corresponding boxes. If the feature map is size of $W \times H$ then there are $W \times H \times k$ anchor boxes. The above mentioned scales are taken to capture the varied size of persons present in the images.

5.5 Object Detection Training

5.5.1 Training Method and Configuration

All the models are either trained on a system provided by RRLab, RPTU or on a remote cluster at RPTU called as Elwetritsch. The configuration of the system in RRLab is Nvidia GeForce RTX 2080Ti GPU and on the remoter server the GPU used was V100-32GB. The faster R-CNN is taken as the base object detection method. The backbone of choice is the ResNet18 model and ResNet34 model. The base object detector is trained firstly without pretrained weights of ResNet architecture and then for comparison purposes trained with pretrained (Image-Net) ResNet weights. Implementation is done in PyTorch due to its ability to provide easy to use machine learning framework. The other training undertaken are by using the pretrained backbone obtained after Reinforcement Learning training is completed. Those pretrained resnet networks are used as the backbone and further comparison of different object detector model is done in Section 6.

5.5.2 Evaluation Metrics

To evaluate the performance of the object detection models, following metrics are used in this thesis:

5.5.2.1 Precision

It is defined as out of all positive predictions how many predictions are truly positive [Powers 20]. The equation is as follow:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} \tag{5.1}$$

5.5.2.2 Recall

Recall is defined as the portion of truly positive predictions out of all positive labels as per the ground truth [Powers 20]. It is defined as follows:

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \tag{5.2}$$

5.5.2.3 Average Precision (AP)

Average Precision [Zhang 09] is widely used in Object Detection algorithms evaluation. The Equation 5.3 defines the Average precision where, $P(k)$ is the precision at the kth recall level and δrecall(k) is change in recall from k-1 to k. Higher AP values indicate better predictions. In this thesis, AP is computed at different IoU (Section 5.5.2.4) thresholds, and the mean of all these value is the mAP (mean average precision).

$$\text{AP} = \sum_{k} (P(k) \times \Delta\text{recall}(k)) \tag{5.3}$$

5.5.2.4 Intersection over Union (IoU)

The metric IoU [Blaschko 08] evaluates the accuracy of the bounding box predictions. Basically, it calculates the overlap between the predicted bounding boxes and the ground truth bounding boxes. IoU is defined as follows:

$$\text{IoU} = \frac{\text{Area of Intersection}}{\text{Area of Union}} \tag{5.4}$$

Here area of intersection is the total area where the ground truth bounding boxes and predicted bounding boxes overlap and Area of Union is the total area covered considering both boxes.

5.6 Entire Training Procedure

The figure 5.16 shows the entire training procedure from RL training to object detection training.

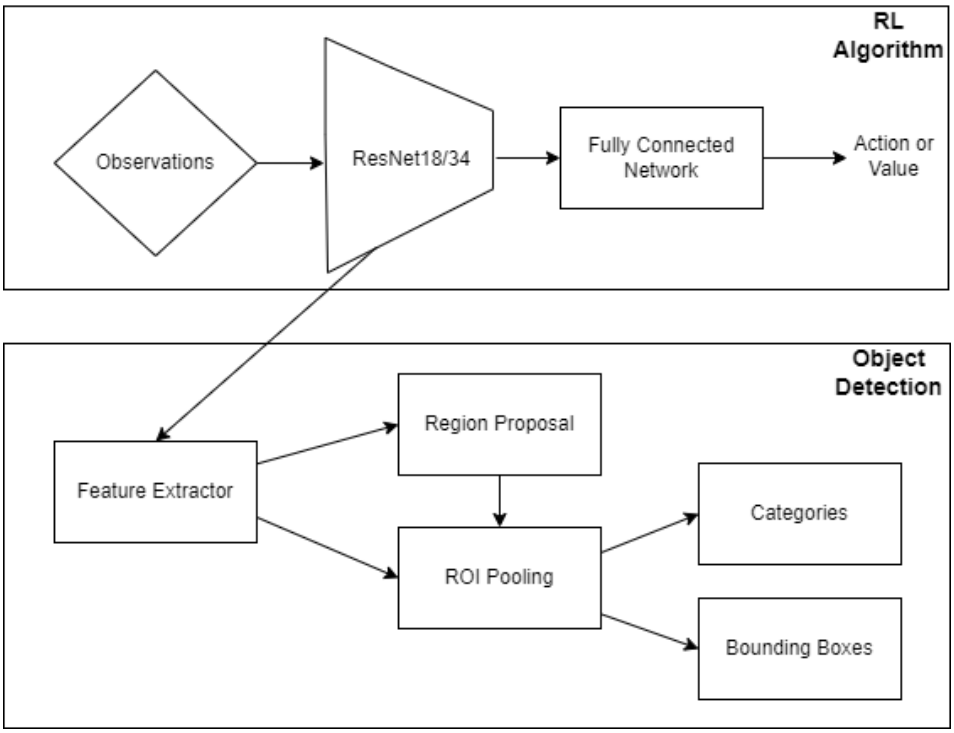


Figure 5.16: Training is divided into two phases. Reinforcement Learning and Object Detection. Value function trained in RL training is used as the backbone for the Object Detection

6. Experiments and Results

This chapter examines the model performance for the autonomous vehicle navigation in Carla Simulation and model performance for pedestrian detection on synthetic dataset. The reinforcement learning algorithm used for training the Autonomous vehicle in Carla simulator is SAC (Soft Actor-Critic). This section is divided into two parts, as follows:

- Section 6.1, results related to RL training are presented.
- Section 6.2, results related to Object Detection on Synthetic datasets are provided.

There is an in-depth comparison between various models trained on the synthetic datasets in upcoming sections.

6.1 Training Reinforcement Learning algorithm

As discussed in Section 5.2.5.1, the architecture of the RL algorithm consists of two parts: a feature extractor and a fully connected layer responsible for predicting value/action. In this project, instead of using the default feature extractor provided by Stable-Baselines3 library in experiments, it has been replaced by ResNet18 or ResNet34.

In the Table 6.1, all the parameters are defined. The parameter Buffer Size is the size of the replay buffer in which some of the past experiences are stored. Train frequency here means the model will get updated after every 64 time-steps. A little action noise is added, which helps to deal with hard exploration problem [Stablebaselines3 24]. Here, γ is the discount factor and tau is the soft update coefficient ("Polyak Update", between 0 and 1). Polyak update is used to slowly update the target value networks and this makes the learning process stable [V.Kumar 19]. The maximum time steps possible in an episode is defined by Episode Length. The rgb sensor is attached to the ego-vehilce and is used to capture the environment surrounding it. The field of view of the sensor needs to be set. Other two sensors used are for detecting collision and detect when the vehicle gets out of the lane. The max_distance_from_center defines the allowed maximum distance for the vehicle to deviate from the center of the lane.

Parameter	Value
time steps	1.5 M
learning rate	0.00001
Batch Size	8
Buffer Size	1000
Seed	7
train frequency	64
action noise	Normal(mean=np.array([0.3, 0]), sigma=np.array([0.5, 0.1]))
gamma (γ)	0.98
tau	0.02
Town	Town02
Episode Length	600
Image width and height	520
sensors	rgb, collision, lane invasion
Vehicle blueprint	model3
throttle space	[-1, 1]
steer space	[-1, 1]
field of view	90
min_speed	10.0
target_speed	20.0
max_speed	25.0
max_dist_from_center	3
low_speed_timer	10 ticks

Table 6.1: Parameters used for training of RL SAC algorithm.

6.1.1 Different Training Approaches

The training approach followed for Reinforcement Learning algorithm in this work is described in Section 5.2.5.1. There are two networks that are part of the SAC RL algorithm, namely, Action Network and Critic Network. Each network, as shown in Figure 5.9, are divided into a Feature Extractor and a Fully-Connected Network components. The basic architecture of both of the components are already implemented by Stable-baselines3 library. In this work, the architecture for feature extractor has been changed and various experiments have been performed. In the upcoming sub-sections, the various different approaches for training the Autonomous Vehicle are explained. In experiments 6.1.1.1, 6.1.1.2, and 6.1.1.3, pedestrians are spawned randomly from all the pedestrian blueprints present in Carla Simulator.

6.1.1.1 Base Feature Extractor

The feature extractor defined by stable-baselines3 library consists of following layers:

- First convolutional layer. It takes 3 channels (RGB) as input and applies 32 filters of size 8×8 and a stride of 4×4.
- Second convolutional layer takes output from the previous layer and applies 64 filters with kernel size 4×4 and stride of 2×2.

- Third convolutional layer takes the previous output from previous and 64 filters of size 3×3 and stride 1×1 which is then followed by ReLU() activation function.
- The output from the convolution layer is flatten using $flatten(start\_dim = 1, end\_dim = -1)$
- Finally, it is followed by a fully connected layer which takes the flattened output and maps it to the lower-dimensional space with 512 features.

In this experiment, the above mentioned default feature extractor is used to train the Rl algorithm. This experiment has ran for 1.5M time-steps and three different reward function as mentioned in Section 5.2.4 are used. The Figure 6.1, 6.2 and 6.3 shows the trend of episode length, episode reward and distance from the start point for the last 100k timesteps of the entire training period.

6.1.1.2 ResNet18 as Feature Extractor

In this experiment, a CustomCNN class is defined which extends from BaseFeaturesExtractor class. In this class, the architecture for the feature extractor of the SAC model is defined. Here, the architecture used for the feature extractor is the ResNet18 architecture (Figure 5.15a). After this, the policy_kwargs dictionary is defined. This dictionary will be used as arguments while initializing the SAC model. This dictionary consist of feature_extractor_class and features_extractor_kwargs. This takes CustomCNN as the input for the feature extractor with $features\_dim$ as input. The reward function is used in the similar way as it was used in earlier function. Combination of three different functions for three different phases of training. In the last phase of the training, pedestrians are spawned in the environment.

The results of the last 500K timestep are shown in Figure 6.4, 6.5, and 6.6. Those Figure depict episode length mean, episode reward mean and distance from start at each timestep respectively. In comparison to Base Feature Extractor, the results with ResNet18 feature extractor has shown improvement. In Base Feature Extractor experiment, the episode reward mean has a trend of decreasing (Figure 6.2), whereas the rewards in ResNet18 (Figure 6.5 has an upward trend with continuous increase the reward as the timestep increases. Similar observation are seen in distance travelled by the vehicle. In Base Feature Extractor, the distance along the timestep (Figure 6.3) is a little stagnant compared to that in ResNet18 experiment (Figure 6.6), which has a small upward trend.

6.1.1.3 ResNet34 as Feature Extractor

Here, the architecture implemented in the CustomCNN class is that of ResNet34 (Figure 5.15b). It will be used as the feature extractor of the SAC model during the training. Similar to earlier experiments, this is also trained with three different reward functions. The results are shown in Figure 6.7, 6.8 and 6.9 for mean episode length, mean episode reward and distance from start through timestep respectively. In the graphs of ResNet34 it is visible that the episode length mean, episode reward mean, and distance from start is increasing through timesteps. This indicates that the vehicle is able to avoid collisions and staying in the line for more number of timesteps and along with it the distance travelled by the vehicle is increasing.

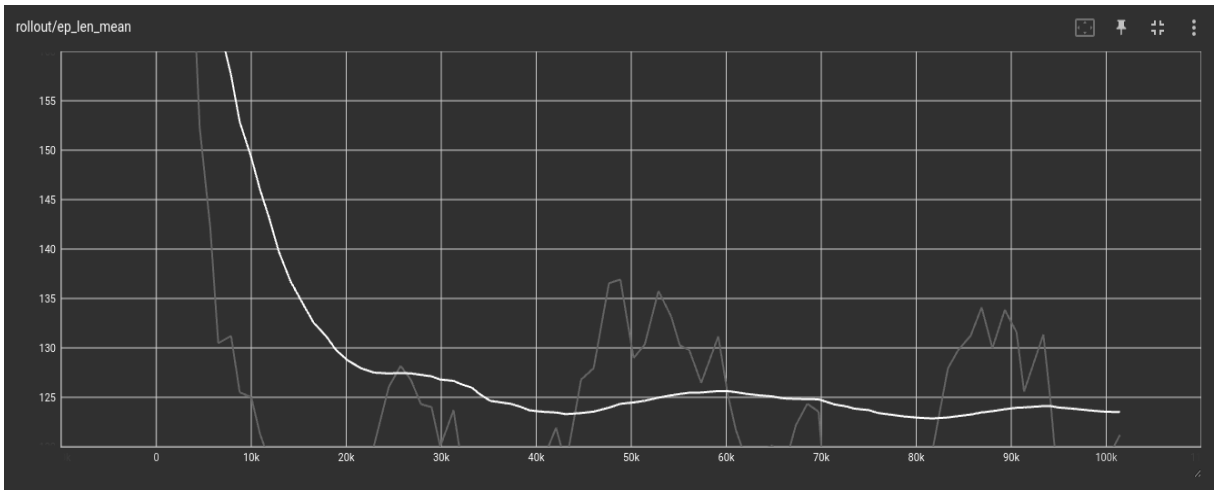


Figure 6.1: Episode Length mean through timesteps (Base Feature Extractor)

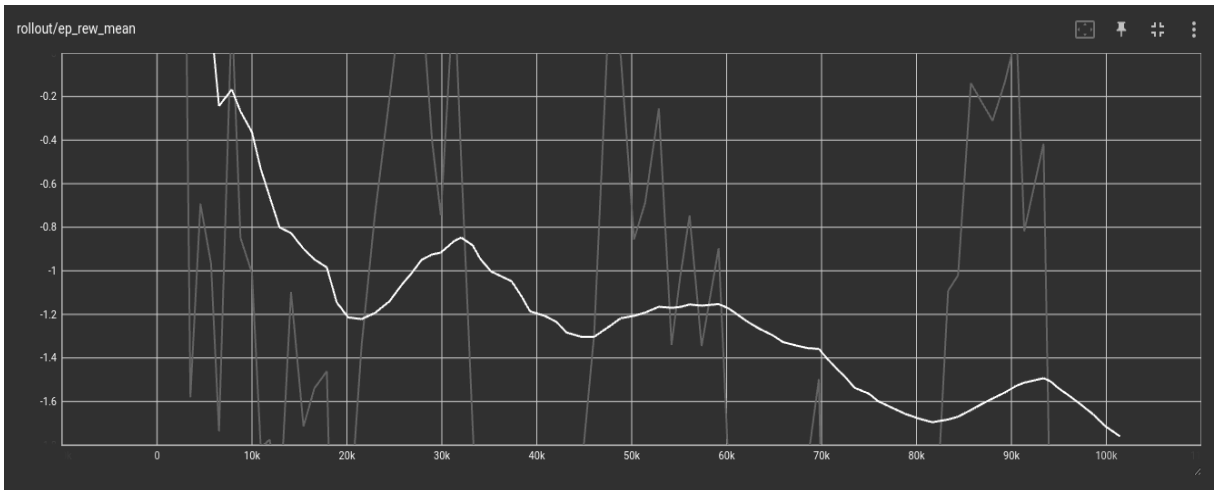


Figure 6.2: Episode Reward mean through timesteps (Base Feature Extractor)

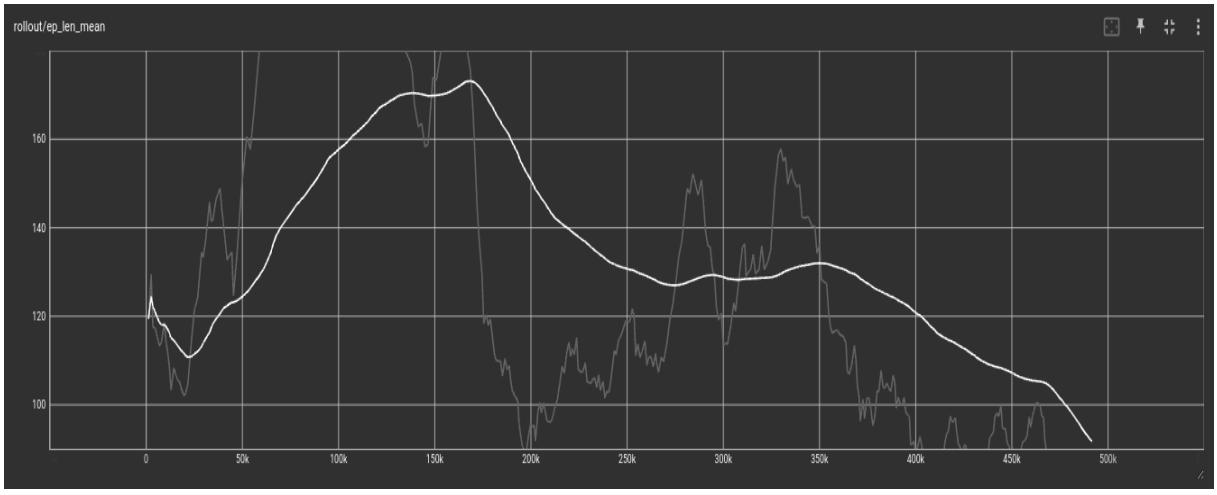


Figure 6.4: Episode Length mean through timesteps (ResNet18 Feature Extractor)

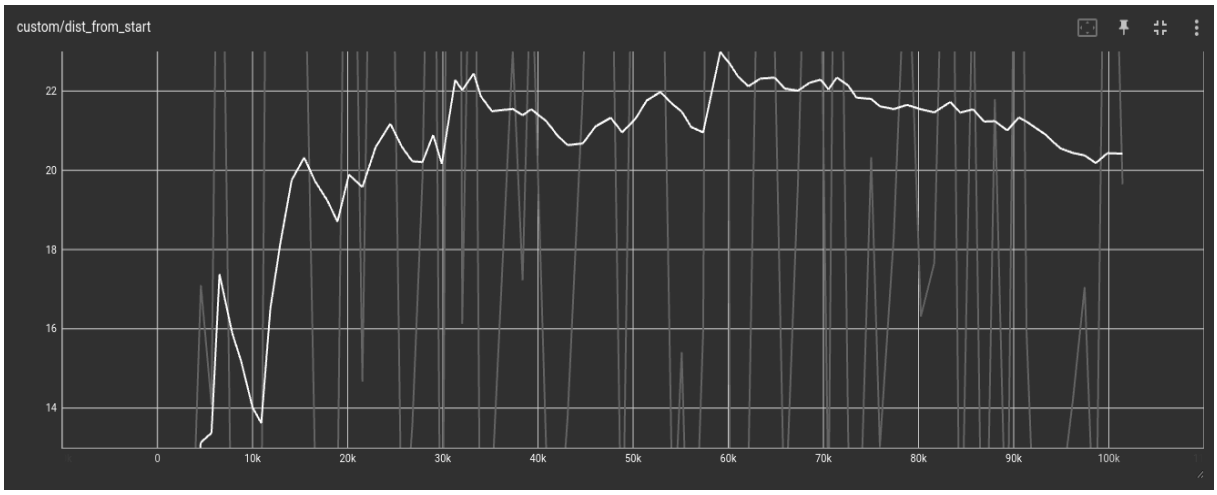


Figure 6.3: Distance travelled by ego-vehicle through timesteps (Base Feature Extractor)

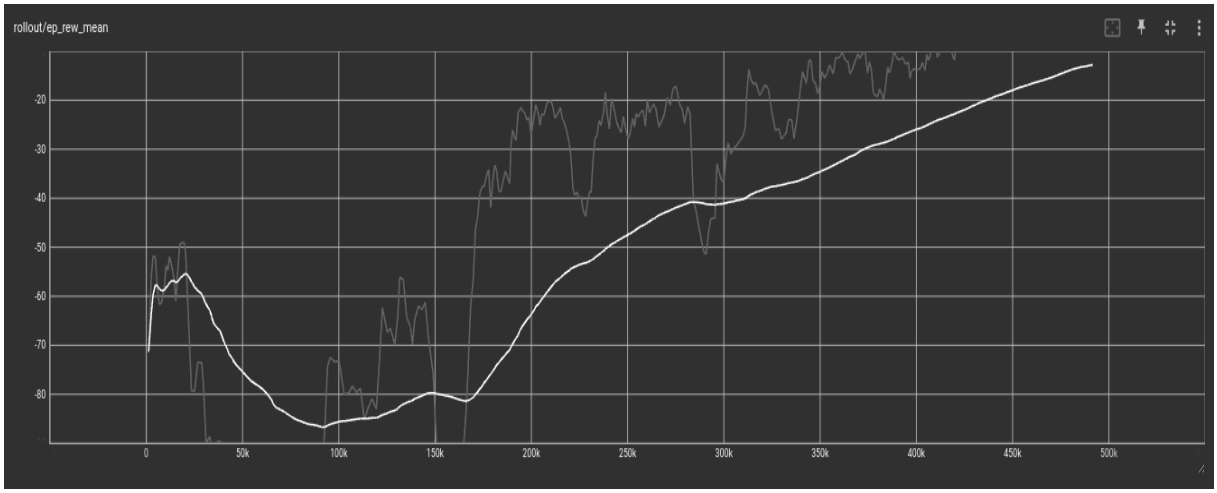


Figure 6.5: Episode Reward mean through timesteps (ResNet18 Feature Extractor)

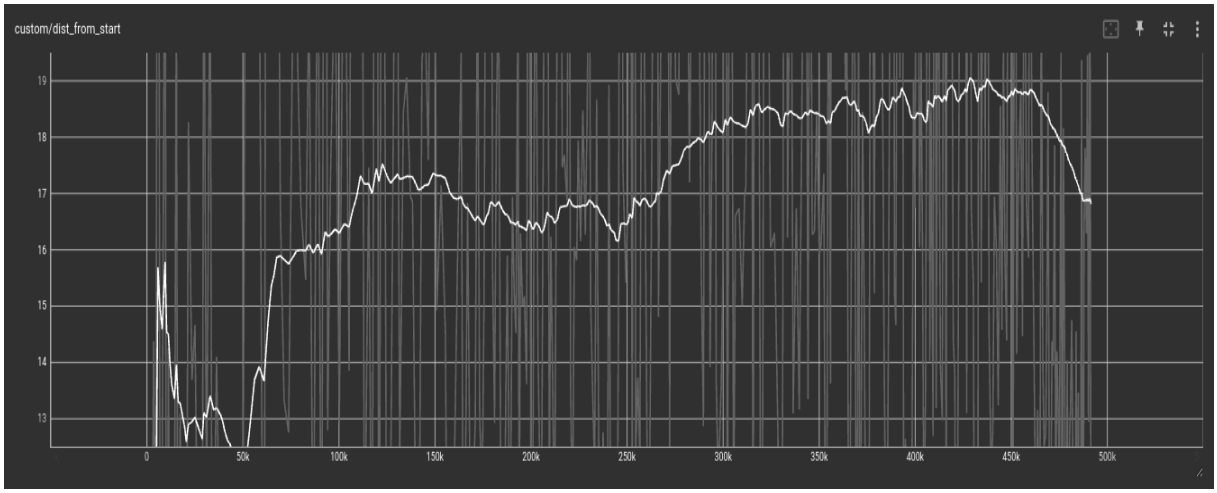


Figure 6.6: Distance travelled by ego-vehicle through timesteps (ResNet18 Feature Extractor)

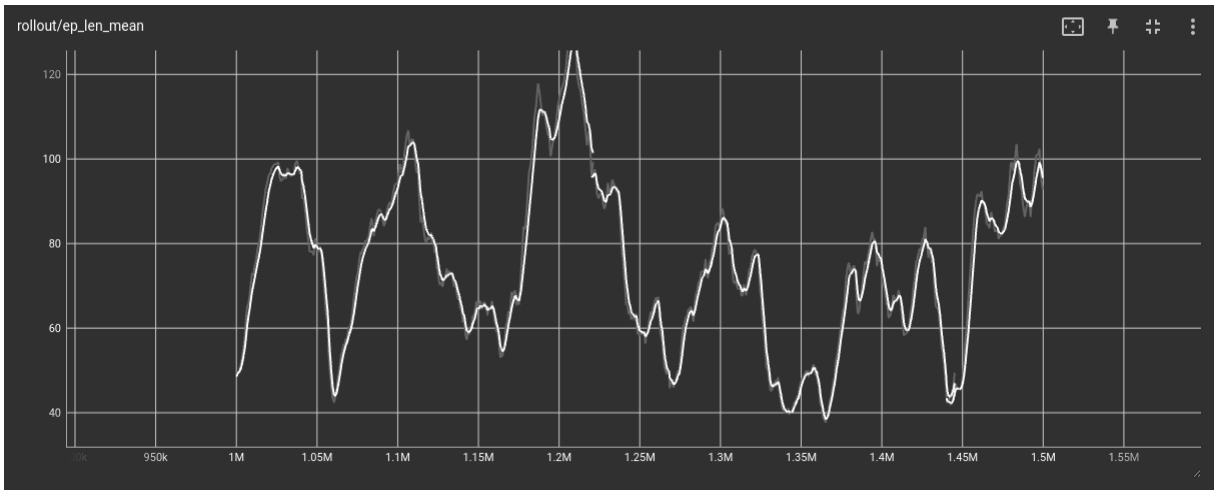


Figure 6.7: Episode Length mean through timesteps (ResNet34 Feature Extractor)

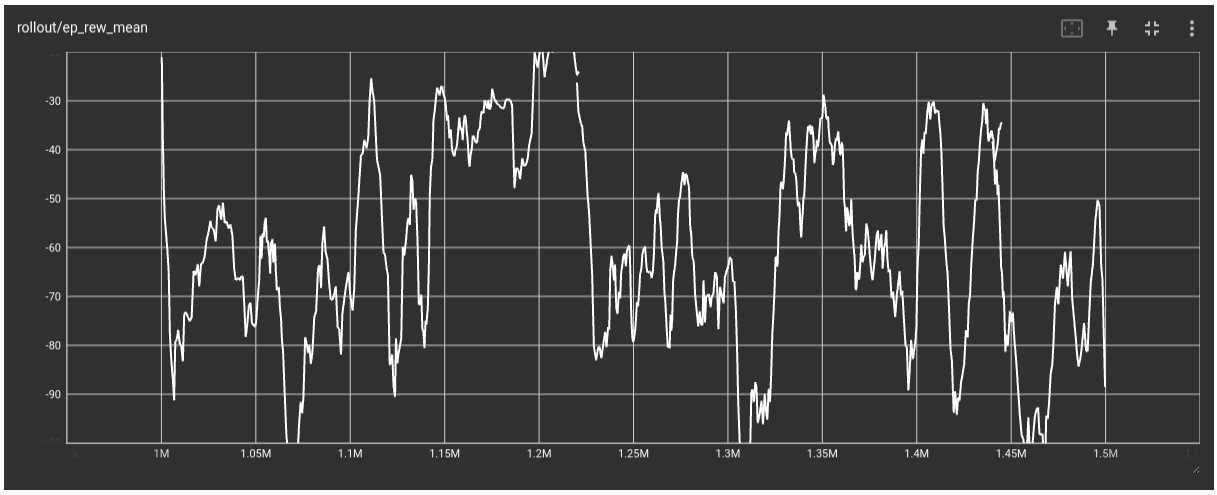


Figure 6.8: Episode Reward mean through timesteps (ResNet34 Feature Extractor)

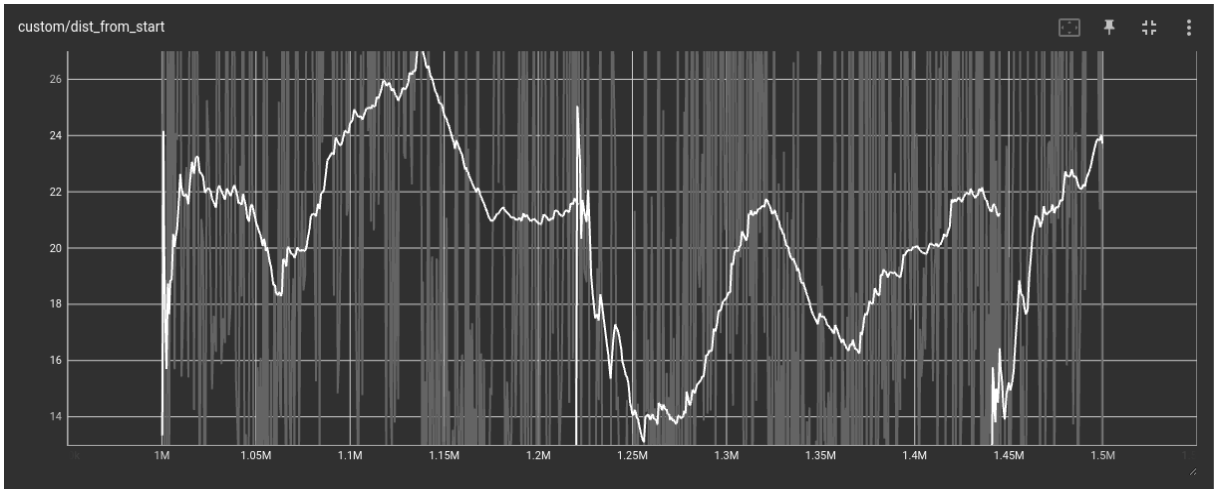


Figure 6.9: Distance travelled by ego-vehicle through timesteps (ResNet34 Feature Extractor)

6.1.1.4 ResNet34 with Dataset with 80:20 pedestrian blueprints

This experiment is similar to training of ResNet34 in Section 6.1.1.3, but the only difference is the pedestrian spawned during the training of RL algorithm. In this experiment, out of all the pedestrian blueprints present in Carla Simulator, only 80% of the blueprints are used to spawn pedestrians in the environment. So, only 80% of the all possible variation of pedestrians are seen by the RL algorithm during training. Similar results are obtained for this experiment which are shown in Figure 6.10, 6.11 and 6.12. The episode length, rewards and the distance covered is following an upward trend. In both ResNet34 experiments (Section 6.1.1.3 & 6.1.1.4), breakpoints in the graphs indicate instances where the training had to be resumed from that specific timestep. For instance, in Figure 6.9, there are two points where the graphs are disconnected (around 1.22M and 1.44M timesteps). When considering the individual sections of the graph, each section shows an upward trend. However, due to the need for training resumption, a continuous upward trend is not visible throughout the figure.

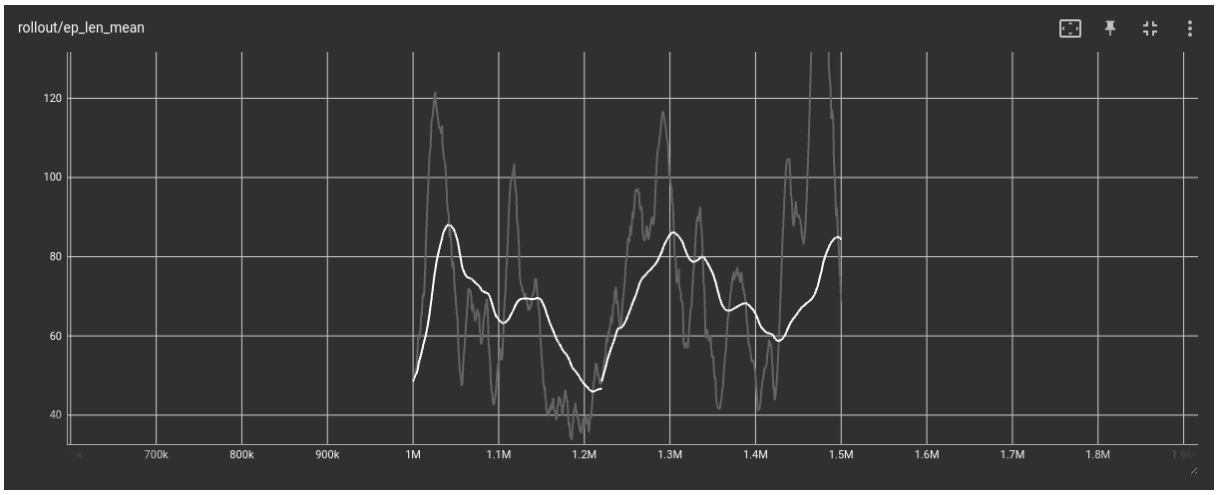


Figure 6.10: Episode Length mean through timesteps (ResNet34 Feature Extractor) with pedestrians spawned from 80% of the total blueprints available

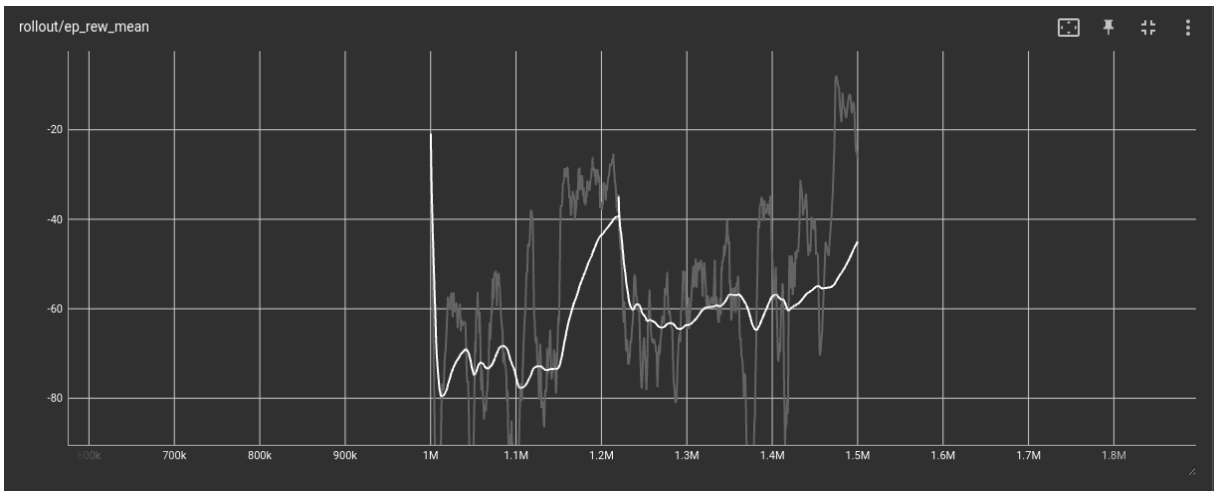


Figure 6.11: Episode Reward mean through timesteps (ResNet34 Feature Extractor) with pedestrians spawned from 80% of the total blueprints available

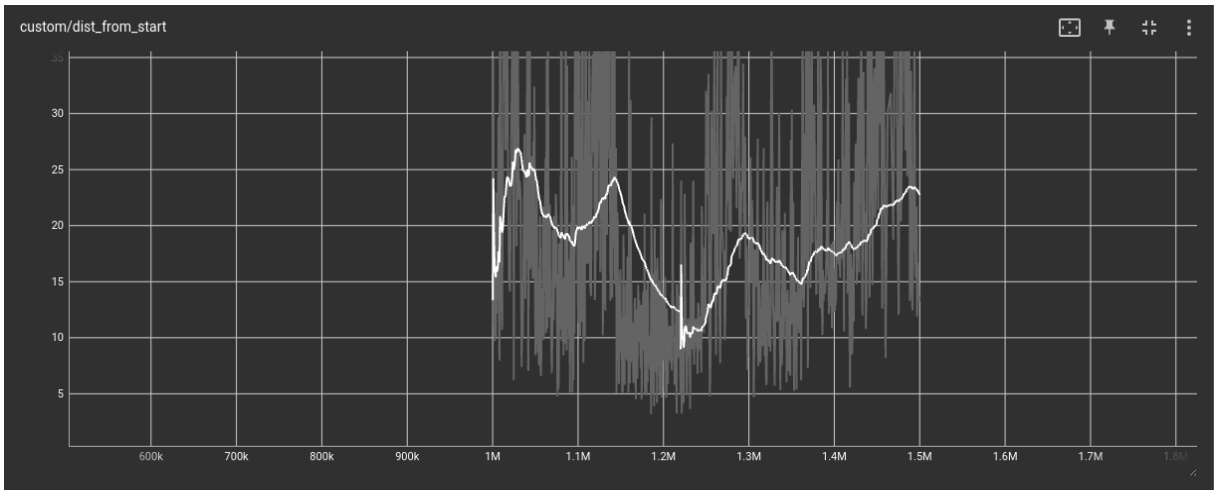


Figure 6.12: Distance travelled by ego-vehicle through timesteps (ResNet34 Feature Extractor) with pedestrians spawned from 80% of the total blueprints available

6.2 Object Detection Model Performance

A Faster R-CNN model utilizing backbones pretrained from RL algorithm training is employed in this study. The object detection model is trained using ResNet18 and ResNet34 as their backbones, which are pretrained in the initial stage. Additionally, to assess the performance of these networks, a Faster R-CNN model is trained from scratch without any pretraining, and another model is trained with ImageNet pretrained weights as a benchmark. The chosen optimizer is the Adam Optimizer, and the learning rate is dynamically adjusted throughout the training process. A learning rate scheduler is implemented to facilitate this adjustment. The parameters for training are listed in Table 6.2.

Parameter	Value
initial learning rate	0.0001
Batch Size	4
Epochs	50
RPN Anchors	(32, 64, 128, 256, 512)
RPN Aspect Ratios	(0.5, 1.0, 2.0)
Image Size	512
Train set	10,745 images
Validation set	2679 images

Table 6.2: Parameters used for training of Object Detection algorithm.

6.2.1 Model evaluation trained with all Pedestrian Blueprints

In this experiment, the model is trained with a dataset created by spawning pedestrians in the environment using all available pedestrian blueprints in the Carla Simulator. The training and test datasets encompass all variations of pedestrians, exposing the model to every pedestrian blueprint during training. Table 6.3 displays the evaluations of different

trained object detection models. This evaluation is done on validation dataset along with training. It's important to note that the model pretrained with ImageNet is included for reference, and a direct comparison is not made between it and the models trained from scratch or with RL-based pretraining. In the case where the backbone is ResNet18, the RL-based pretrained object detection model exhibits a better AP metric, while for the ResNet34 model, the performance of the model trained from scratch and the RL-based pretrained model is equal. As the test dataset was generated using Carla Simulator, the ground truth for the test set are available. The models are evaluated on test set as shown in Table 6.4. The performance of both the models are similar on the test dataset.

The quantitative as well qualitative results of RL-pretrained model and model trained from scratch are very similar. Figure 6.13 has a series of three consecutive images where the pedestrians are detected using the model with both of the models. This is one of the instances where the model with RL-pretraining has performed well. It is able to detect the pedestrian behind the light pole in all the three timestamps. Whereas, the model without any pretraining is unable to detect the pedestrian in first image, in second image it has one pedestrian not detected and one incorrect detection. In third image, both the models are able to detect the pedestrians correctly.

Pretraining	Backbone	AP(0.5:0.95)	AP(0.5)
No, From scratch	ResNet18	0.550	0.824
Yes, RL-based	ResNet18	**0.560**	**0.826**
Yes, ImageNet	ResNet18	0.615	0.867
No, From scratch	ResNet34	**0.573**	**0.836**
Yes, RL-based	ResNet34	0.572	**0.836**
Yes, ImageNet	ResNet34	0.636	0.869

Table 6.3: Quantitative results of Object Detection models (all pedestrian blueprints)

Pretraining	Backbone	AP(0.5:0.95)	AP(0.5)
No, From scratch	ResNet18	0.526	**0.810**
Yes, RL-based	ResNet18	**0.532**	0.805
Yes, ImageNet	ResNet18	0.603	0.855
No, From scratch	ResNet34	**0.552**	0.814
Yes, RL-based	ResNet34	0.550	**0.815**
Yes, ImageNet	ResNet34	0.618	0.858

Table 6.4: Quantitative results of Object Detection models on test dataset (all pedestrian blueprints)

6.2.2 Model evaluation trained with 80% of Pedestrian Blueprints

In this experiment, only 80% of the pedestrian blueprints were used to spawn pedestrians in the environment for training purposes. The model is then evaluated on unseen pedestrian blueprints, with the test set created by spawning pedestrians from the remaining 20% of the blueprints. After conducting experiments on this dataset, the obtained results are presented in Table 6.5. Only ResNet34 based object detection model is trained with this specific dataset.

Pretraining	Backbone	AP(0.5:0.95)	AP(0.5)
No, From scratch	ResNet34	**0.572**	**0.844**
Yes, RL-based	ResNet34	0.567	0.836
Yes, ImageNet	ResNet34	0.637	0.876

Table 6.5: Quantitative results of Object Detection models (80% of pedestrian blueprints used for training)

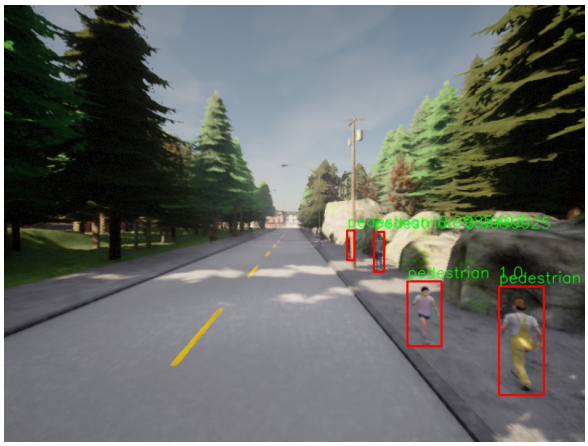
Pretraining	Backbone	AP(0.5:0.95)	AP(0.5)
No, From scratch	ResNet34	**0.529**	0.801
Yes, RL-based	ResNet34	0.524	**0.803**
Yes, ImageNet	ResNet34	0.603	0.844

Table 6.6: Quantitative results of Object Detection models on test dataset (80% of pedestrian blueprints used for training)

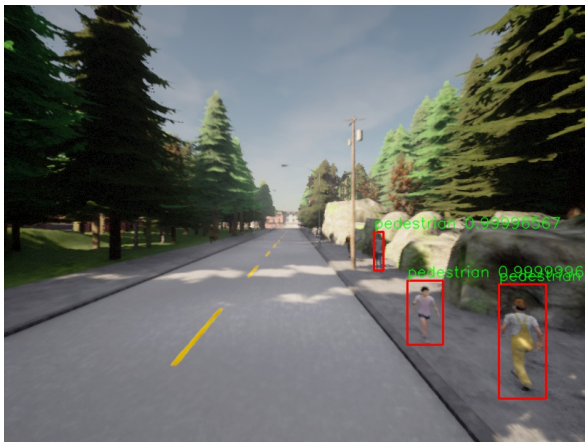
This section presents a summary of how both models perform when evaluated on a dataset where the pedestrians spawned are entirely from blueprints that have not been seen before (during training). During training phase, the validation set is used to evaluate the models. In Table 6.5 it is clear that the model with no pretraining is performing better in comparison to the RL-pretrained model. But, when evaluated on test set, the performance of both the models is found to be similar and in some case RL-pretrained model performs better (shown in Table 6.6).

The qualitative analysis of models on test data is shown in Figure 6.14. In sub figures 6.14a and 6.14b the pedestrian on the left side has gone undetected by the model trained from scratch. Similarly in the Image 2 and Image 3 there are missed detection by model without any pretraining. On further inspection of the performance of the models on test set, it was observed that the model without any pretraining failed many times to detect the pedestrians even when the pedestrian were near the camera and its features were visible. Whereas, the cases where the RL pretrained model failed was when the pedestrians were at a far distance and the features were not that distinct. In Figure 6.15, there are three different frames used to evaluate the model performance. In Image-1 (6.15a & 6.15b), the pedestrian present at far distance wasn't detected by the RL-based pretrained

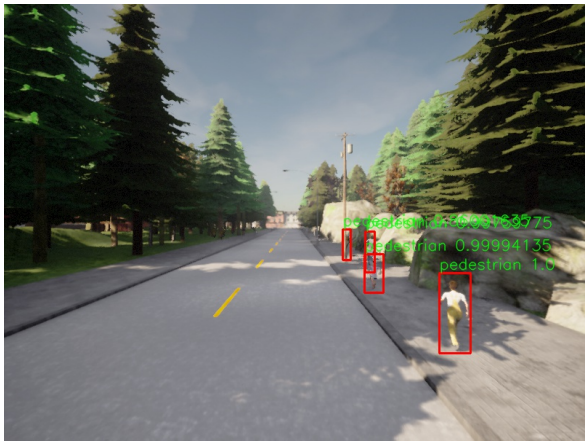
model, whereas the other model could correctly detect both the pedestrians present in the environment. Similar observation is made in Image-3. In Image-2, the pedestrian features which are clearly visible to camera are detected by the RL-based model, whereas the other model fails to detect. But it is able to detect the pedestrian on the other side of the lane.



(a) Timestamp_1, Model pretrained with RL



(b) Timestamp_1, Model without pretraining



(c) Timestamp_2, Model pretrained with RL

(d) Timestamp_2, Model without pretraining

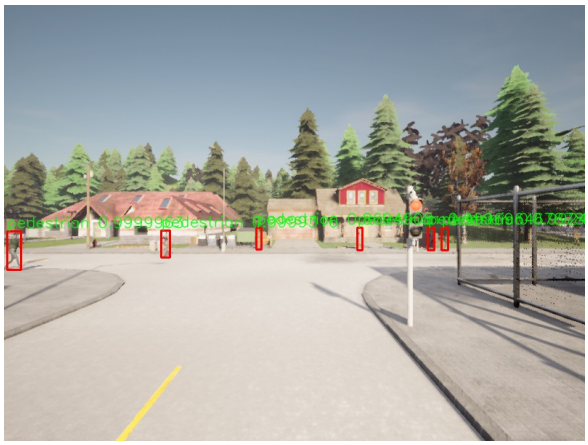
(e) Timestamp_3, Model pretrained with RL

(f) Timestamp_3, Model without pretraining

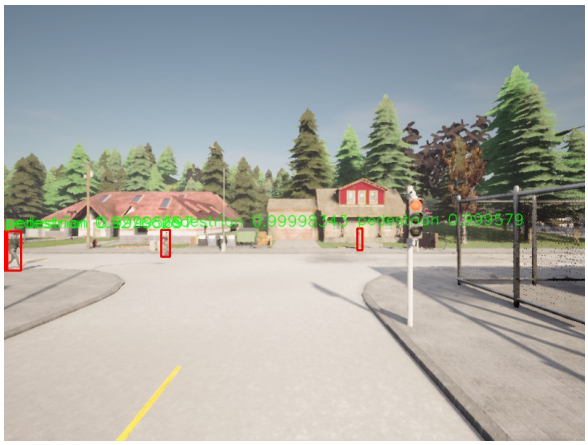
Figure 6.13: These are images from three consecutive timestamps. Left side are all the detection with the model pretrained with RL feature extractor, and right side are all the images trained from scratch. In this series of images, the RL pretrained model has consistent predictions, and on other hand the model without pretraining in first image missed the detection behind the light pole and in second image had missed one detection and one incorrect detection.

(a) Image 1, Model pretrained with RL

(b) Image 1, Model without pretraining



(c) Image 2, Model pretrained with RL



(d) Image 2, Model without pretraining

(e) Image 3, Model pretrained with RL

(f) Image 3, Model without pretraining

Figure 6.14: Three images from test dataset are taken as samples for evaluating models. Left side : model with RL pretraining, Right side: model without pretraining. Cases where the RL based model performed better

(a) Image 1, Model pretrained with RL

(b) Image 1, Model without pretraining

(c) Image 2, Model pretrained with RL

(d) Image 2, Model without pretraining

(e) Image 3, Model pretrained with RL

(f) Image 3, Model without pretraining

Figure 6.15

7. Discussion and Conclusion

In this thesis, we looked into training vehicle to autonomously drive in Carla Simulator environment using deep reinforcement learning approach. SAC algorithm was used in this project. After training RL algorithm along with various different reward functions, it is confirmed that one of the major factpr impacting the training of the algorithm is the design of the reward function. The actions of the ego-vehicle can be affected by the reward functions. The positive and the negative rewards promotes the agent to take actions that are favorable. After training the agent, the agent is able to avoid pedestrian and stop when there are pedestrians crossing the roads. But the behavior of the agent is still inconsistent. Over the course of this project, there is improvement seen in the performance of the agent due to some changes in the reward functions, but there are some areas where further exploration can be done to make improvements.

To improve the behavior of the agent further, redesigning of the reward function and its investigation is required. The features of images are really important for the agent. In this project, image was only taken as the state/observations. Different approach that can be undertaken is to learn agents behavior from semantically segmented images. Other information such as throttle, steer, angle, speed, distance from center can be used as additional states/observations that can be given as input to the Reinforcement Learning algorithm. This will require to use multi-input observations which can be implemented using Stable-baselines3 library.

Finally, in terms of improvement in training the ego-vehicle agent, different learning techniques can be used. The SAC algorithm used in thesis performs well with continuous action space and consist of Experience Buffer. Here, one idea for future work is to use an expert to drive the vehicle in the environment for some episodes and fill up the buffer with those experiences (mostly favorable actions taken and some bad actions). On that basis further training can be undertaken. This might improve the behavior of the agent.

The second phase of the thesis was regarding pedestrian detection in the same environment in which the RL agent was trained. There were major three experiments undertaken, with Base Feature Extractor, ResNet18 as the feature extractor, and ResNet34 as the feature extractor. Using those as the backbone of the architecture, models were trained and are evaluated on two different datasets. The observations are as follows:

- Table 6.3, and 6.4, show the evaluation results of the models trained on first dataset. Here, the model with pretrained feature extractor shows a similar perform to that of model trained from scratch. And in some cases even better. The quantitative metric were similar but the there were some instances in test set where the RL-based model performed better at detecting pedestrian in consecutive series of images, which is evident in Figure 6.13.
- Table 6.5 and 6.6, shows the evaluation results of model trained on second dataset. In this experiment the results are interesting. On the basis of evaluation of the validation set, it is found that the model trained from scratch is performing better than model with pretrained-backbone. But, during inference time, the unseen test set is given as input for evaluation. Here, the performance of both the models are equal and as seen in Figure 6.14, there are many instances where the RL-based model is performing better.

In conclusion, the performance on dataset where the test set comprises of pedestrians spawned using unseen blueprints, the RL-based model performance increases in comparison to what it was during evaluation on validation dataset. More concise experiments can be undertaken to check if there is further improvement after pretraining with RL. This can be done by improving the agent behavior by experimenting with the points mentioned above. In context of object detection task, future work can consist of using more complex backbone for Faster R-CNN architecture or explore other object detection architectures. In addition to exploring the architectures, as the dataset was created using a script, there were some incorrect ground truths present in the dataset. In future, more finely annotated datasets can be used to make improvements. There is a lot of scope on working further on this work. Above mentioned approaches can be considered and experimented for improving the performance of the models.

Bibliography

[Adib 20] M. Adib, "CARLA 2D Bounding Box Annotation Module", 2020. https://mukhlasadib.github.io/CARLA-2DBBox/.

[Ahmed 21] M. . K. . Ahmed, "Supervised Machine Learning: A Survey", pp. 2–3, 2021.

[Balasubramaniam 22] A. Balasubramaniam, S. Pasricha, "Object Detection in Autonomous Vehicles: Status and Open Challenges", 01 2022.

[Bansal 19] S. Bansal, "3D Convolutions: Undersntanding + Use Case", 2019. http://www.kaggle.com/code/shivamb/3d-convolutions-understanding-use-case.

[Bento 21] C. Bento, "Multilayer Perceptron Explained with a Real-Life Example and Python Code: Sentiment Analysis", 2021. https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-

[Bhalerao 23] C. Bhalerao, "YOLO v8! The real state-of-the-art?", 2023-01-17. https://medium.com/mlearning-ai/yolo-v8-the-real-state-of-the-art-eda6c86a1b90#:~:text=On%20January%2010th%2C%202023%2C%20the,architectural%20changes%20with%20better%20results.

[Bhatarai 18] S. Bhatarai, "What is Gradient Descent in Machine Leanring?", 2018. https://saugatbhattarai.com.np/what-is-gradient-descent-in-machine-learning/.

[Bhateja 23] C. Bhateja, D. Guo, D. Ghosh, A. Singh, M. Tomar, Q. Vuong, Y. Chebotar, S. Levine, A. Kumar, "Robotic Offline RL from Internet Videos via Value-Function Pre-Training", 2023.

[Bhatt 18] S. Bhatt, "Reinforcement Learning 101", 2018. https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292.

[Blaschko 08] M. B. Blaschko, C. H. Lampert, "Learning to Localize Objects with Structured Output Regression", in *Computer Vision – ECCV 2008*, D. Forsyth, P. Torr, A. Zisserman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 2–15.

[Borah 20] C. Borah, "Evolution of Object Detection", 2020. https://medium.com/analytics-vidhya/evolution-of-object-detection-582259d2aa9b.

[Brockman 16] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, "OpenAI Gym", 2016.

[Brown 20] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, "Language Models are Few-Shot Learners", 2020.

[Brownlee 20] J. Brownlee, "A Gentle Introduction to the Rectified Linear Unit (ReLU)", 2020-08-20.

[Buffet 20] O. Buffet, O. Pietquin, P. Weng, "Reinforcement Learning", 2020.

[Carla] Carla, "Python API tutorial". `https://carla.readthedocs.io/en/0.9.6/img/pedestrian_types.png`.

[Cho 14] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation", 2014.

[Cios 07] K. J. Cios, R. W. Swiniarski, W. Pedrycz, L. A. Kurgan, *Unsupervised Learning: Association Rules*, Boston, MA: Springer US, 2007, pp. 289–306.

[Cortes 95] C. Cortes, V. Vapnik, "Support-vector networks", *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[Dalal 05] N. Dalal, B. Triggs, "Histograms of oriented gradients for human detection", in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1. 2005, pp. 886–893 vol. 1.

[Daoud 22] E. Daoud, N. Khalil, M. Gaedke, "Implementation of a one Stage Object Detection Solution to Detect Counterfeit Products Marked With a Quality Mark", vol. 17. 08 2022, pp. 37–49.

[Doshi 19] S. Doshi, "Various Optimization Algorithms For Training Neural Networks", 2019. `https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6`.

[Doshi 21] K. Doshi, "Reinforcement Learning Explained Visually (Part 6): Policy Gradients, step by step", 2021.

[Dosovitskiy 17] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, V. Koltun, "CARLA: An Open Urban Driving Simulator", in *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[Dubey 20] S. R. Dubey, S. Chakraborty, S. K. Roy, S. Mukherjee, S. K. Singh, B. B. Chaudhuri, "diffGrad: An Optimization Method for Convolutional Neural Networks", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, no. 11, pp. 4500–4511, 2020.

[Dubey 22] S. R. Dubey, S. K. Singh, B. B. Chaudhuri, "Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark", 2022.

[Dumoulin 18] V. Dumoulin, F. Visin, "A guide to convolution arithmetic for deep learning", 2018.

[Face 24] H. Face, "The advantages and disadvantages of policy-gradient methods - Hugging Face Deep RL Course", 2024. Accessed : 2024-01-10, `https://huggingface.co/learn/deep-rl-course/unit4/advantages-disadvantages`.

[Felzenszwalb 10] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, D. Ramanan, "Object Detection with Discriminatively Trained Part-Based Models", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 9, pp. 1627–1645, 2010.

[Fradkov 20] A. L. Fradkov, "Early History of Machine Learning", vol. 53, pp. 1385–1390, 2020.

[GeeksforGeeks 22] GeeksforGeeks, "Object Detection vs Object Recognition vs Image Segmentation", 2022. `https://www.geeksforgeeks.org/object-detection-vs-object-recognition-vs-image-segmentation/`.

[Ghosh 23] D. Ghosh, C. Bhateja, S. Levine, "Reinforcement Learning from Passive Data via Latent Intentions", 2023.

[Girshick 14] R. Girshick, J. Donahue, T. Darrell, J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation", 2014.

[Girshick 15] R. Girshick, "Fast R-CNN", 2015.

[Haarnoja 18] T. Haarnoja, A. Zhou, P. Abbeel, S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor", 2018.

[He 14] K. He, X. Zhang, S. Ren, J. Sun, *Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition*, Springer International Publishing, 2014, p. 346–361.

[He 15] K. He, X. Zhang, S. Ren, J. Sun, "Deep Residual Learning for Image Recognition", *CoRR*, vol. abs/1512.03385, 2015.

[Hill 18] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, "Stable Baselines", `https://github.com/hill-a/stable-baselines`, 2018.

[Hochreiter 97] S. Hochreiter, J. Schmidhuber, "Long Short-Term Memory", *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, nov 1997.

[Ivakhnenko 67] A. G. Ivakhnenko, V. G. Lapa, *Cybernetics and Forecasting Techniques*, ser. Modern analytic and computational methods in science and mathematics, American Elsevier Publishing Company, 1967.

[J. McCarthy 55] N. R. C. E. S. J McCarthy, M L Minsky, "A Proposal for the Dartmouth Summer Research Project on Artifical Intelligence", 1955.

[Jain 21] P. Jain, S. Goenka, S. Bagchi, B. Banerjee, S. Chaterji, "Federated Action Recognition on Heterogeneous Embedded Devices", 07 2021.

[javatpoint 21] javatpoint, "Deep learning vs. Machine learning vs. Artificial Intelligence - Javatpoint", 2021. https://static.javatpoint.com/tutorial/machine-learning/images/deep-learning-vs-machine-learning-vs-artificial-intelligence1.png.

[Jin 10] X. Jin, J. Han, *K-Means Clustering*, Boston, MA: Springer US, 2010, pp. 563–564.

[Karaginannakos 18] S. Karaginannakos, "The idea behind Actor-Critics and how A2C and A3C improve them", 2018-11-17. https://theaisummer.com/Actor_critics/.

[Karunakaran] D. Karunakaran, ""Soft Actor-Critic Reinforcement Learning algorithm"". https://medium.com/intro-to-artificial-intelligence/soft-actor-critic-reinforcement-learning-algorithm-1934a2c3087f.

[Karunakaran 21] D. Karunakaran, "Relationship between state(V) and action(Q) value function in Reinforcement Learning", 2021. https://medium.com/intro-to-artificial-intelligence/relationship-between-state-v-and-action-q-value-function-in-reinforcement-learning-~:text=Value%20function%20can%20be%20defined,function%20to%20understand%20RL%20better.

[Kathuria 21] A. Kathuria, "Getting Started with OpenAI Gym: The basis building blocks", 2021. https://blog.paperspace.com/content/images/2020/11/openaigym.jpg.

[Kaul 20] S. Kaul, "Gradient Descent Problems and Solutions in Neural Networks", 2020. https://medium.com/analytics-vidhya/gradient-descent-problems-and-solutions-in-deep-learning-8002bbac09d5.

[Krizhevsky 12a] A. Krizhevsky, I. Sutskever, G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. Burges, L. Bottou, K. Q. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012.

[Krizhevsky 12b] A. Krizhevsky, I. Sutskever, G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105.

[LeCun 89] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition", *Neural Computation*, vol. 1, pp. 541–551, 1989.

[LeCun 98] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, "Gradient-based learning applied to document recognition", *Proc. IEEE*, vol. 86, pp. 2278–2324, 1998.

[Li 23] D. Li, H. Ling, A. Kar, D. Acuna, S. W. Kim, K. Kreis, A. Torralba, S. Fidler, "DreamTeacher: Pretraining Image Backbones with Deep Generative Models", 2023.

[Liu 16] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, A. C. Berg, *SSD: Single Shot MultiBox Detector*, Springer International Publishing, 2016, p. 21–37.

[Liu 21] Q. Liu, X. Li, S. Yuan, Z. Li, "Decision-Making Technology for Autonomous Vehicles Learning-Based Methods, Applications and Future Outlook", 2021.

[Lu 20] L. Lu, "Dying ReLU and Initialization: Theory and Numerical Examples", *Communications in Computational Physics*, vol. 28, no. 5, p. 1671–1706, june 2020.

[Millard-Ball 18] A. Millard-Ball, "Pedestrians, Autonomous Vehicles, and Cities", *Journal of Planning Education and Research*, vol. 38, no. 1, pp. 6–12, 2018.

[Mishra 20] M. Mishra, "Convolutional Neural Networks", 2020-08-26. `https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939`.

[Mitchell 97] T. M. Mitchell, "Machine Learning", *McGraw-Hill*, 1997.

[Mnih 13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, "Playing Atari with Deep Reinforcement Learning", 2013.

[Molnar 23] C. Molnar, *10.1 Learned Features | Interpretable Machine Learning - A Guide for making black box Models Explainable*, 2023. `https://christophm.github.io/interpretable-ml-book/`.

[Morimoto 21] J. Morimoto, F. Ponton, "Virtual reality in biology: could we become virtual naturalists?", 05 2021. `https://www.researchgate.net/publication/351953193_Virtual_reality_in_biology_could_we_become_virtual_naturalists`.

[Naeem 23] S. Naeem, A. Ali, S. Anam, M. Ahmed, "An Unsupervised Machine Learning Algorithms: Comprehensive Review", *IJCDS Journal*, vol. 13, pp. 911–921, 04 2023.

[Owen 20] L. Owen, "Bird's-Eye View of Reinforcement Learning Algorithms Taxonomy", 2020. `https://towardsdatascience.com/birds-eye-view-of-reinforcement-learning-algorithms-landscape-2aba7840211c`.

[Paszke 19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library", in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[Powers 20] D. M. W. Powers, "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation", *CoRR*, vol. abs/2010.16061, 2020.

[Raffin 21] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, N. Dormann, "Stable-Baselines3: Reliable Reinforcement Learning Implementations", *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

[Redmon 15] J. Redmon, S. K. Divvala, R. B. Girshick, A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection", *CoRR*, vol. abs/1506.02640, 2015.

[Ren 16] S. Ren, K. He, R. Girshick, J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", 2016.

[Rong 20] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta et al, "LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving", *arXiv preprint arXiv:2005.03778*, 2020.

[Ruder 16] S. Ruder, "An overview of gradient descent optimization algorithms", *CoRR*, vol. abs/1609.04747, 2016.

[Rumelhart 86] D. E. Rumelhart, G. E. Hinton, R. J. Williams, "Learning representations by back-propagating errors", *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[Schulman 17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, "Proximal Policy Optimization Algorithms", 2017.

[Sears-Collins 19] A. Sears-Collins, "How to choose an Optimal Learning Rate For Gradient Descent", 2019. https://automaticaddison.com/how-to-choose-an-optimal-learning-rate-for-gradient-descent/#:~:text=If%20you%20choose%20a%20learning,for%20finding%20the%20optimal%20solution.

[Shah 17] S. Shah, D. Dey, C. Lovett, A. Kapoor, "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles", in *Field and Service Robotics*. 2017.

[Silver 16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis, "Mastering the game of Go with deep neural networks and tree search", *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[Silver 21] D. Silver, S. Singh, D. Precup, R. S. Sutton, "Reward is enough", *Artificial Intelligence*, vol. 299, p. 103535, 2021.

[Simonyan 14] K. Simonyan, A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", *arXiv preprint arXiv:1409.1556*, 2014.

[Sojitra 23] P. Sojitra, "What is Reinforcement Learning?", 2023. https://parthsojitra.medium.com/what-is-reinforcement-learning-2e5944f0bbc#:~:text=Reinforcement%20learning%20also%20has%20some,received%20for%20taking%20different%20actions.

[Son 22] D. B. Son, T. H. Binh, H. K. Vo, B. M. Nguyen, H. T. T. Binh, S. Yu, "Value-based reinforcement learning approaches for task offloading in Delay Constrained Vehicular Edge Computing", *Engineering Applications of Artificial Intelligence*, vol. 113, p. 104898, 2022.

[Stable-baselines3] Stable-baselines3, "Policy Networks". https://stable-baselines3.readthedocs.io/en/master/_images/net_arch.png.

[Stablebaselines3 24] Stablebaselines3, "SAC", 2024. https://stable-baselines3.readthedocs.io/en/master/modules/sac.html.

[Subir Varma 18] S. D. Subir Varma, "Deep Learning", 2018-09-27. https://srdas.github.io/DLBook/GradientDescentTechniques.html#issues-with-gradient-descent.

[Suran 20] A. Suran, "On-Policy v/s Off-Policy Learning", 2020. https://towardsdatascience.com/on-policy-v-s-off-policy-learning-75089916bc2f#:~:text=Off-Policy%20learning%20algorithms%20evaluate,in%20both%20ways)%20%2C%20etc.

[Szegedy 14] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, "Going Deeper with Convolutions", *CoRR*, vol. abs/1409.4842, 2014.

[Torcs 07] Torcs, "TORCS, The Open Racing Car Simulator.", 2007.

[Vaswani 23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, "Attention Is All You Need", 2023.

[Velliangiri 19] S. Velliangiri, S. Alagumuthukrishnan, S. I. Thankumar joseph, "A Review of Dimensionality Reduction Techniques for Efficient Computation", *Procedia Computer Science*, vol. 165, pp. 104–111, 2019. 2nd International Conference on Recent Trends in Advanced Computing ICRTAC -DISRUP - TIV INNOVATION , 2019 November 11-12, 2019.

[Viola 01] P. Viola, M. Jones, "Rapid object detection using a boosted cascade of simple features", in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, vol. 1. 2001, pp. I–I.

[V.Kumar 19] V. V. Kumar, "Soft Actor-Critic Demystified", 2019. https://towardsdatascience.com/soft-actor-critic-demystified-b8427df61665.

[Webots] Webots, "http://www.cyberbotics.com". Open-source Mobile Robot Simulation Software.

[Y C a 18] P. Y C a, V. Pulabaigari, E. B, "Semi-supervised learning: a brief review", *International Journal of Engineering and Technology*, vol. 7, pp. 3–7, 02 2018.

[Yani 19] M. Yani, S. Irawan, C. Setianingsih, "Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail", *Journal of Physics: Conference Series*, vol. 1201, p. 012052, 05 2019.

[yanlai00 20] yanlai00, "RL-Carla", 2020. https://github.com/yanlai00/RL-Carla/tree/main.

[Yoon 19] C. Yoon, "Deep Deterministic Policy Gradients Explained", 2019. https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b.

[Zeiler 13] M. D. Zeiler, R. Fergus, "Visualizing and Understanding Convolutional Networks", *CoRR*, vol. abs/1311.2901, 2013.

[Zhang 09] E. Zhang, Y. Zhang, *Average Precision*, Boston, MA: Springer US, 2009, pp. 192–193.

[Zhang 20] H. Zhang, T. Yu, *Taxonomy of Reinforcement Learning Algorithms*, Singapore: Springer Singapore, 2020, pp. 125–133.

8. Appendix

One additional approach for training Reinforcement Learning agent was taken. In the previous experiments, the autonomous vehicle was spawned randomly anywhere in the Carla map Town02. In this particular experiment, the car was spawned at the same location after every episode and had to learn to cover a single route. While training the previous models, the ResNet34 feature extractor was trained from scratch, whereas for this particular experiment the ResNet34 was loaded with pretrained weights from ImageNet. In Figure 8.1, 8.2, and 8.3 are the results and shows the trend of episode mean length, episode mean reward and distance from start respectively. Here a decrease in rewards is observed over the timesteps.

The object detection trained using this pretrained model is used to get the quantitative results which are shown in comparison with other models in Table 8.1. This model outperforms the earlier models which trained with pretrained backbone and trained from scratch.

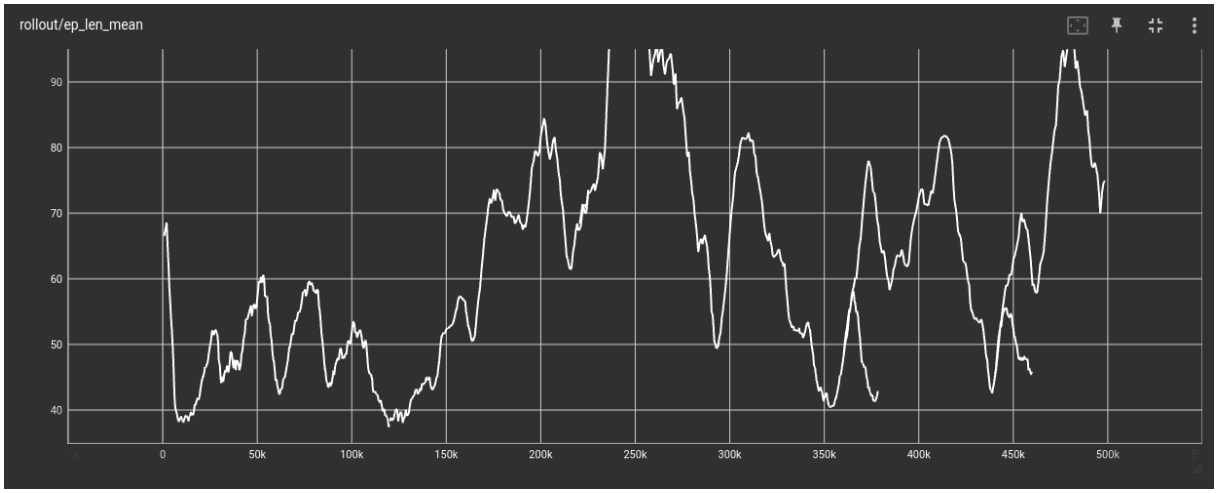


Figure 8.1: Episode Length mean through timesteps (ResNet34 Feature Extractor with one waypoint training approach)

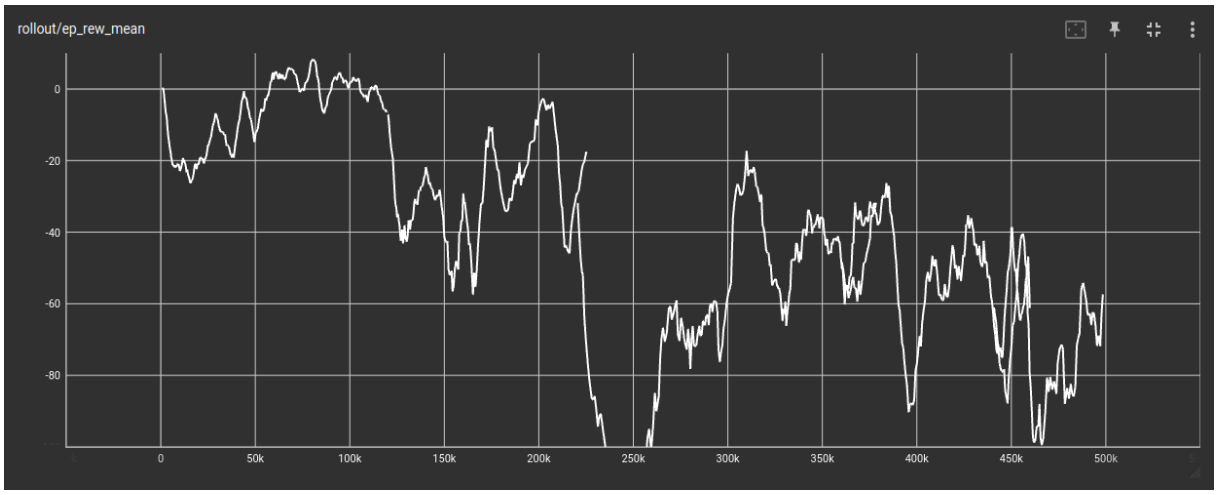


Figure 8.2: Episode Reward mean through timesteps (ResNet34 Feature Extractor with one waypoint training approach)

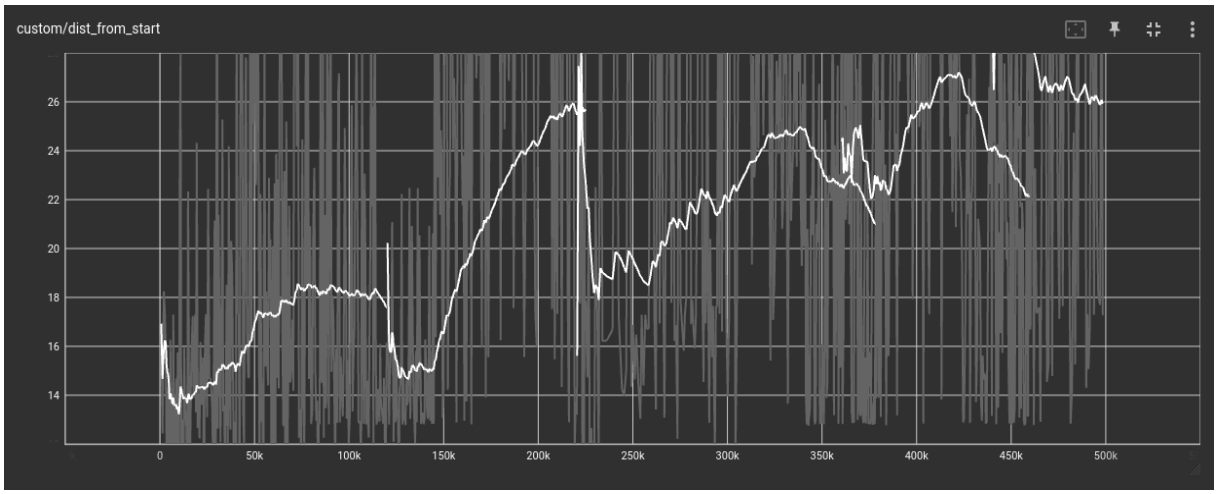
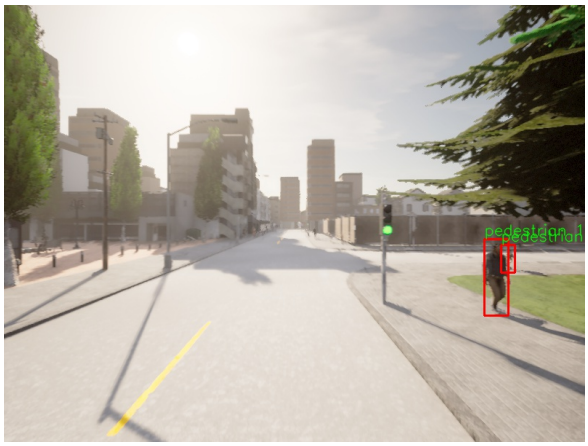


Figure 8.3: Distance travelled by ego-vehicle in each timestep (ResNet34 Feature Extractor with one waypoint training approach)

Pretraining	Backbone	AP(0.5:0.95)	AP(0.5)
No, From scratch	ResNet34	0.573	0.836
Yes, RL-based	ResNet34	0.572	0.836
Yes, RL-based (one waypoint)	ResNet34	**0.581**	**0.845**
Yes, ImageNet	ResNet34	0.636	0.869

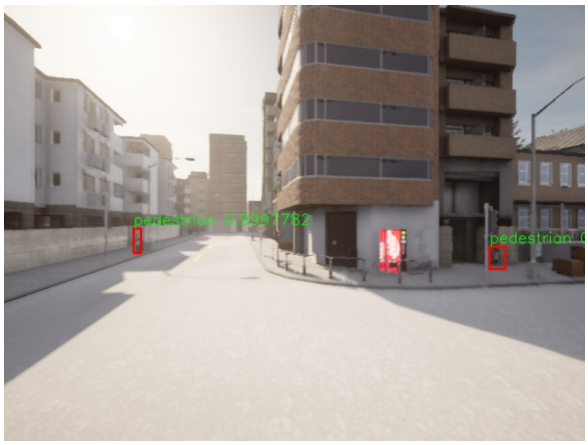
Table 8.1: Quantitative results of Object Detection models (All the available blueprints of pedestrian were used in training)

There are few more qualitative results in Figure 8.4 and 8.5 that obtained using models trained in Section 6.2.2.



(a) Image 1, Model pretrained with RL

(b) Image 1, Model without pretraining



(c) Image 2, Model pretrained with RL

(d) Image 2, Model without pretraining

(e) Image 3, Model pretrained with RL

(f) Image 3, Model without pretraining

Figure 8.4: More qualitative results obtained from ResNet34 models. Left: Predictions with model with pretrained RL, Right: Predictions with model trained from Scratch

(a) Image 1, Model pretrained with RL

(b) Image 1, Model without pretraining

(c) Image 2, Model pretrained with RL

(d) Image 2, Model without pretraining

(e) Image 3, Model pretrained with RL

(f) Image 3, Model without pretraining

Figure 8.5: More qualitative results obtained from ResNet34 models. Left: Predictions with model with pretrained RL, Right: Predictions with model trained from Scratch