ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
Τμήμα Ηλεκτρολόγων & Ηλεκτρονικών Μηχανικών
www.eee.uniwa.gr
Θηβών 250, Αθήνα-Αιγάλεω 12244
Τηλ. +30 210 538-1225, Fax. +30 210 538-1226

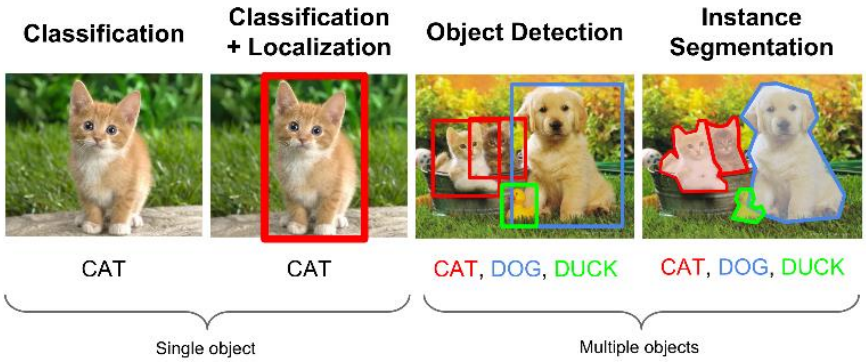
UNIVERSITY of WEST ATTICA
FACULTY OF ENGINEERING
Department of Electrical & Electronics Engineering
www.eee.uniwa.gr
250, Thivon Str., Athens, GR-12244, Greece
Tel:+30 210 538-1225, Fax:+30 210 538-1226

Πρόγραμμα Μεταπτυχιακών Σπουδών
Ηλεκτρικές & Ηλεκτρονικές Επιστήμες μέσω Έρευνας

Master of Science By Research in
Electrical & Electronics Engineering

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Φίλτρα Kalman και εφαρμογή στην ανίχνευση και παρακολούθηση κίνησης



Μεταπτυχιακός Φοιτητής: Γεώργιος Χαρκοφτάκης, ΑΜ MSCRES-0019
Επιβλέπουσα : Μαρία Ραγκούση

ΑΙΓΑΛΕΩ, ΦΕΒΡΟΥΑΡΙΟΣ 2020

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
Τμήμα Ηλεκτρολόγων & Ηλεκτρονικών Μηχανικών
www.eee.uniwa.gr
Θηβών 250, Αθήνα-Αιγάλεω 12244
Τηλ. +30 210 538-1225, Fax. +30 210 538-1226

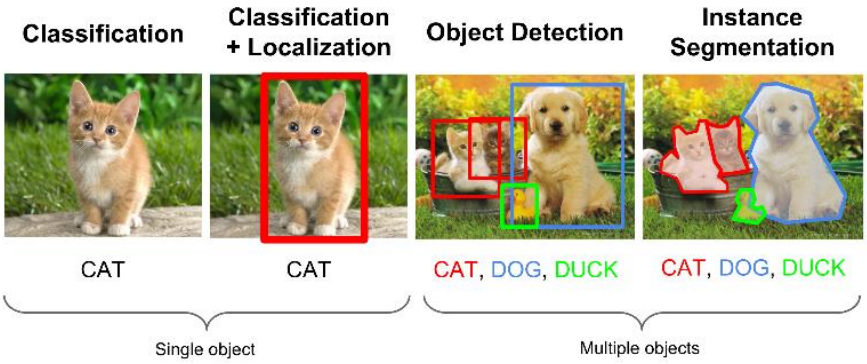
UNIVERSITY of WEST ATTICA
FACULTY OF ENGINEERING
Department of Electrical & Electronics Engineering
www.eee.uniwa.gr
250, Thivon Str., Athens, GR-12244, Greece
Tel:+30 210 538-1225, Fax:+30 210 538-1226

Πρόγραμμα Μεταπτυχιακών Σπουδών
Ηλεκτρικές & Ηλεκτρονικές Επιστήμες μέσω Έρευνας

Master of Science By Research in
Electrical & Electronics Engineering

MSc Thesis

Kalman Filters and their application in motion detection and tracking



Student: Georgios Charkoftakis, Reg. Nr. MSCRES-0019
MSc Thesis Supervisor: Maria Rangoussi

ATHENS-EGALEO, FEBRUARY 2020

ABSTRACT

During the last decade, breakthroughs in the fields of machine vision and reinforcement learning have been made possible thanks to the advancements in machine learning algorithms. 'Machines' now can recognize an object with human accuracy; this implies they could play video games by just having access to the pixels of a screen. In this work a combination of state-of-the-art technologies of deep learning are exploited to detect, track and remotely control a vehicle under realistic conditions. A linear estimator (Kalman filter) is proposed as an application-specific solution to a common problem that is the delay of sensory input – in the present case, the latency introduced by the machine vision system. The thesis presents the proposed system design and practical considerations along with detailed experimental results as a proof of concept. The advantages of the proposed solution are established through comparative evaluation with the results of other 'naive' solutions.

KEYWORDS: *Machine Learning, Machine Vision, Reinforcement learning, Deep Neural Networks, CNN, DQN, YOLOv3, Linear Estimator, Kalman Filter, Motion detection, Motion tracking, Vehicle control*

Η Μεταπτυχιακή Διπλωματική Εργασία έγινε αποδεκτή, εξετάστηκε και βαθμολογήθηκε από την εξής τριμελή εξεταστική επιτροπή:

Επιβλέπουσα	Μέλος	Μέλος
Maria Ragkousi Digitally signed by Maria Ragkousi Date: 2022.01.21 14:16:08 +02'00'	Xenofon-Dionysios Kandris Digitally signed by Xenofon-Dionysios Kandris Date: 2022.01.21 15:01:40 +02'00'	Digitally signed by Nikolaos-Alexandros Tatlas Date: 2022.01.21 18:29:04 +02'00'
Ραγκούση Μαρία	Κανδρής Δ.	Τάτλας Ν.-Α.
Καθηγήτρια	Καθηγητής	Αν. Καθηγητής

ΔΗΛΩΣΗ ΣΥΓΓΡΑΦΕΑ ΜΕΤΑΠΤΥΧΙΑΚΗΣ ΔΙΠΛΩΜΑΤΙΚΗΣ ΕΡΓΑΣΙΑΣ

Ο κάτωθι υπογεγραμμένος Γεώργιος Χαρκοφτάκης του Κων/νου, με αριθμό μητρώου MSCRES-0019 φοιτητής του Προγράμματος Μεταπτυχιακών Σπουδών «Ηλεκτρικές και Ηλεκτρονικές Επιστήμες μέσω Έρευνας» του Τμήματος Ηλεκτρολόγων και Ηλεκτρονικών Μηχανικών της Σχολής Μηχανικών του Πανεπιστημίου Δυτικής Αττικής, δηλώνω ότι:

«Είμαι συγγραφέας αυτής της μεταπτυχιακής διπλωματικής εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της, είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, οι όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε ακριβώς είτε παραφρασμένες, αναφέρονται στο σύνολό τους, με πλήρη αναφορά στους συγγραφείς, τον εκδοτικό οίκο ή το περιοδικό, συμπεριλαμβανομένων και των πηγών που ενδεχομένως χρησιμοποιήθηκαν από το διαδίκτυο. Επίσης, βεβαιώνω ότι αυτή η εργασία έχει συγγραφεί από μένα αποκλειστικά και αποτελεί προϊόν πνευματικής ιδιοκτησίας τόσο δικής μου, όσο και του Ιδρύματος. Παράβαση της ανωτέρω ακαδημαϊκής μου ευθύνης αποτελεί ουσιώδη λόγο για την ανάκληση του πτυχίου μου».

Ο Δηλών

ΓΕΩΡΓΙΟΣ ΧΑΡΚΟΦΤΑΚΗΣ

ΠΕΡΙΛΗΨΗ

Η πρόοδος στον τομέα της μηχανικής μάθησης στην προηγούμενη δεκαετία, έκανε δυνατά σημαντικά επιτεύγματα στους τομείς της μηχανικής όρασης και της ενισχυμένης εκμάθησης. Μέσω καταλλήλων αλγορίθμων, οι 'μηχανές' μπορούν πλέον να αναγνωρίσουν αντικείμενα το ίδιο καλά με ανθρώπους – πράγμα που σημαίνει ότι θα μπορούσαν π.χ. να παίζουν βιντεοπαιχνίδια απλά έχοντας πρόσβαση στα pixels μιας οθόνης. Στην παρούσα εργασία αξιοποιούνται οι πλέον σύγχρονοι αλγόριθμοι βαθιάς μάθησης για τον εντοπισμό, την παρακολούθηση και τον απομακρυσμένο έλεγχο ενός οχήματος σε ρεαλιστικές συνθήκες. Επίσης προτείνεται η χρησιμοποίηση ενός γραμμικού εκτιμητή (φίλτρο Kalman) ως λύση σε ένα συγκεκριμένο πρόβλημα - εκείνο της καθυστέρησης της λήψης δεδομένων από τον αισθητήρα, δηλαδή της καθυστέρησης που εισάγει το σύστημα μηχανικής όρασης. Παρουσιάζονται αναλυτικά οι λεπτομέρειες της σχεδίασης, οι προκλήσεις που αντιμετωπίστηκαν και τα πειραματικά αποτελέσματα. Τέλος τα πλεονεκτήματα της προτεινόμενης λύσης τεκμηριώνονται μέσω συγκριτικής αξιολόγησης με τα αποτελέσματα άλλων «απλοϊκών» λύσεων.

ΛΕΞΕΙΣ – ΚΛΕΙΔΙΑ: *Μηχανική Μάθηση, Μηχανική Όραση, Ενισχυμένη Εκμάθηση, Βαθιά Νευρωνικά Δίκτυα, CNN, DQN, YOLOv3, Γραμμικός Εκτιμητής, Φίλτρο Kalman, Ανίχνευση Κίνησης, Παρακολούθηση Κίνησης, Έλεγχος οχήματος*

ACKNOWLEDGEMENTS

I would like to express my thanks to my wife Eirini for her support and encouragement during this work and to my supervising professor Mrs Maria Rangoussi for her help and advice.

TABLE OF SYMBOLS – ACRONYMS – ABBREVIATIONS

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
DL	Deep Learning
DQN	Deep Q Network
ODS	Object Detection Server
RL	Reinforcement Learning
YOLO	You Only Look Once

TABLE OF CONTENTS

CHAPTER 1:
Introduction and Existing Research

The aim of this thesis is to develop and train algorithms to detect, track and remotely control a vehicle moving on a 2-D plane, under realistic conditions, using a combination of state-of-the-art technologies of deep learning. Furthermore, a linear estimator (Kalman filter) is proposed and introduced as an application-specific solution to a recognized problem of such applications, namely, the delay of sensory input – in the present case, the latency introduced by the machine vision system. The thesis presents the proposed system design and discusses practical considerations. A prototype is constructed and detailed experimental results are presented as a proof of concept. The advantages of the proposed solution are established through comparative evaluation with the results of other 'naive' solutions.

The recent advances in deep learning have brought significant research gains in the fields of object detection within images (Agarwal, Terrail, & Jurie, 2018) and in reinforcement learning (Li, 2018). This was made possible thanks to advances in (micro-)computer architecture, namely, thanks to the massive parallelization of CUDA cores (CUDA, n.d.) inside a desktop workstation in the form of a Graphics Processing Unit (GPU) that allows experimentation with novel architectures of increased performance, accuracy and "super human" speed. Progress in deep learning has relied in the increased efficiency of artificial neural networks (ANN) of the convolutional type. These networks are essentially algorithms that make feasible the running of visual recognition tasks on compact low cost computers (Jetson Nano DK, n.d.). The multi core architecture of those miniature computers that may perform ANN-based inference within milliseconds is equally effective.

ANNs constitute a family of Soft Computing algorithms whose success relies critically on a 'training' or 'learning' phase, where the algorithm's parameters are iteratively adjusted on the basis of a 'rich' training set of examples, each tagged with the correct 'answer' (output value or class). A variety of learning algorithms have been proposed and tested on various use cases. They are essentially optimization methods, where the optimization criterion is a function of the error between actual and correct 'answers' (ANN output values or classes). Reinforcement learning (RL) is a current and promising development along these lines.

The basic idea behind reinforcement learning is to train a software algorithm (an 'agent') by maximizing the accumulated rewards of a policy π. The training of the agent is a tradeoff between exploration and exploitation of the agent's environment. The field of application of RL is vast, ranging from economics to medicine and autonomous vehicle control. It was Mnih et al. (2013) that has first combined the achievements of RL with deep neural networks. An RL agent could compete or even exceed human agents in playing Atari 2600 games. The RL agent would outperform human agents based on knowledge of raw pixel values alone; it knew nothing of the game's inner dynamics.

The same technology combined with Monte Carlo methods gave RL the title of the 'Go' champion (Silver et al., 2016). 'Go' is a traditional Chinese game that is famous for its vagueness and its huge number of combinations. This type of game is clearly suited for human intelligence, so it was a surprise when AlphaGo, a software developed by DeepMind won the world champion by four to one (Sang-Hun, 2016). This was a rare case when state of the art research has been up in the news headlines.

The difficulties of training an RL agent are not restricted only in the algorithm selection and the hyper parameters fine tuning; the large number of 'episodes', in the order of tens of thousands, along which the agent should be able to explore its environment is also a big challenge. Unlike the Atari 2600 games, some environments are not available for

experiment using a real (physical) agent, such as a moving vehicle; a simulated agent should be used instead, for training and evaluation. The restrictions do not come only from the fact that a real agent may be expensive or the environment may be unavailable, as in the case of a spacecraft. It is also possible that each episode may be unfolded ('run') for hours or even days. The real agent may be deployed after a certain level of confidence is reached, following exhaustive performance evaluation with a simulated agent.

Object detection in digital images is known to be one of the most challenging tasks for intelligent algorithms, since it demands the detection of a variety of objects within a mixed background. The objects may vary in size (scale), from just a few pixels (where most of the information is lost) up to the occupation of a large portion of the image. Moreover, the objects may be rotated, or partially obscured by other objects under varying lighting conditions. The successful detection of objects by 'machines' (algorithms) using cameras has drawn significant research attention and effort, due to its numerous applications. Through the use of this technology, today computers may extract semantic information from the surrounding environment; this in turn allows for successful machine interaction. As the highly optimized semiconductors necessary for performing inference tasks are getting cheaper (Nardo, Petrosino, & Santopietro, 2018; Pena, Forembski, Xu, & Moloney, 2017) the wide deployment of this technology is imminent.

The traditional approach to object detection within images has been the extraction of object characteristics in the form of empirically defined features. This approach has had the side benefit of the reduction of the problem space dimension. The compression thus achieved has been more than welcome: in fact, it was a necessity, given the performance limitations of the available hardware.

Since the 1990's, the convolutional neural networks (CNN) have been successfully used for specific tasks like recognition of hand written characters (LeCun et al., 1990). After years of experimentation, CNNs lost interest in favour of SVNs. The work on image

recognition in the first years of 2010's relied on the methods of Histogram of Oriented gradients (HOG) (Dalal & Triggs, 2005) and Scale Invariant Featute Transform (SIFT) (Lowe, 2004); unfortunately, the progress results suggested a stalemate.

Krizhevsky, Sutskever, & Hinton (2012) found that a high number of image characteristics could be processed by an equally complex network with a large number of parameters. They trained a CNN using a training set of 1.2 million images of ImageNet, belonging to 1000 categories. This team ('SuperVision') won the ILSVRC2012 contest and set a new classification (Task 1) record of top-5 and top-1 accuracies at 16.4% and 38.1% respectively. The ILSVRC2012 Task 2 is an object localization task where the SuperVision team achieved an equally important top-5 error of 33.5%. Unfortunately details of the localization method employed are not included in the publication.

Krizhevsky's work shifted again the interest of the researches towards CNNs. This achievement has become feasible thanks to the advent of high performance CUDA GPUs that could massively perform highly optimized 2D convolutions and stochastic gradient descent (SGD) algorithm runs, necessary for network training. The Krizhevsky's CNN is named 'AlexNet' and contains 5 convolutional layers and 3 fully connected layers resulting in 60 million parameters and 650 thousand neurons. The good results of AlexNet in the image classification and object localization tasks mentioned earlier were achieved thanks to introduction in the training algorithms of measures that combat overfitting – a problem that is common in networks of that scale. During training, the AlexNet training algorithm used patches of the original images of 224x224 pixels, plus the horizontal reflections of them. Another anti-overfitting measure employed is the 'dropout' where a proportion of the neurons remain inactive, meaning their output is zero, during training. This forces these neurons to adjust on their own than to rely on the activation of other neurons.

At this point it was clear that network architect choices have a great impact on CNN performance. Drawbacks of AlexNet identified included the vast amount of annotated data

required for supervised learning (network training), the large memory needed, the high computational requirements and the fixed size of the input images (224x224 pixels).

The major interest behind RL and object detection in general is the feasibility of real time control applications. Even if a state of the art object detection algorithm is employed, still there is a significant delay between the issue of the execution command and its actual effects. When training is completed and the network is used for testing, i.e. inference, it is seen that the performance of inference algorithms, even if a hardware accelerator like a GPU is employed, takes tens of milliseconds because of the multilayer nature of the CNN. In an attempt to analyze and further break down the actual delays, when a commercial USB camera is employed, it is estimated that the delay for the image transfer to the computer memory can be hundreds of milliseconds. Low-latency, industrial-grade USB and Ethernet cameras have a prohibitive cost for large scale deployment. Moreover, the performance of these high speed cameras depends on the existence of ideal lighting conditions. In non-ideal conditions, the exposure time increases thus mitigating their advantage. In brief, the problem with the use of machine vision algorithms for real time problems lies at the nature of each application in itself (e.g., poor lighting conditions) and cannot always be overcome by the selection of expensive hardware.

In addition to the challenges mentioned before, the use of RL in real time control is faced with extra difficulties: the deployment of a real application, such as object detection and control, undergoes the constraints of inference execution time, the delay of the command to reach the agent and the effort to build a simulated environment. The delays that accumulate in either case (real or simulated agent) may cause loss of control due to loop instability.

In light of the above discussion, it would be of great value to combine both technologies, object detection and RL, for the real time control of an agent (e.g., a moving vehicle) through the selection and use of the appropriate set of existing methods and tools to

overcome the difficulties of actual deployment. The aim of the present research is to show that efficient remote control and navigation of vehicles is feasible using low cost computing platforms. Furthermore, this can be achieved with robustness using a combination of available technology of reinforcement learning, visual object detection and linear estimators.

CHAPTER 2:
Problem Statement and Specifications

In order to remotely control a vehicle using an RL agent and object detection technology what is required is the flawless integration of all the subsystems involved, i.e., the vehicle, the camera, an object detection process, an RL agent and a communications system. The interconnection and interoperability of these components should be efficient enough to cope with the problems mentioned earlier. The control loop should be stable, the vehicle should reach a designated target and while getting there it should not move off limits. The control should be optimal, the agent should not wait for a stable feedback signal (position) to decide for the control signal due to the object detection latency. The RL agent and the object detection system should be fast enough to decide the new control signal within a fraction of a time step. The communications system should pass the control signal to the vehicle with close to zero delay.

Previous work examines the use of RL methods in the presence of delayed feedback (Walsh, Nouri, Li, & Littman, 2008) or when there is action delay (Firoiu, Ju, & Tenenbaum, 2018). Walsh et al. showed that the augmentation method when a Constant Delay Markov Decision Process (CMDP) is mapped to MDP (Bertsekas, 2017; Katsikopoulos & Engelbrecht, 2003) blows up exponentially the memory / computational requirements and thus renders the problem insolvable. They proposed the Model Based Simulation (MBS) planning method, when the CMDP is finite and in conjunction with Model Parameter Approximation (MPA) when the CMDP is continuous. MPA is a no-delay model-based RL algorithm that complements the MBS.

In a real case scenario the MBS+MPA have some disadvantages:

1. The MPA should be used to extract matrices P and R - something that is not feasible for many real world problems.
2. This setup is oriented towards problems where the time delay T_L is a multiple k of the time step ΔT. When k=0 the CMDP is an MDP without delay. In some real time problems, where k=0, the observation S_t is available at time $t_0+\gamma$ where $\gamma<\Delta T$ and solving the problem by getting S_t at $t_0+\Delta T$ (k=1) is a suboptimal solution.
3. The MBS method does not address the problem of time jitter that is common when observations arrive asynchronously.
4. The MBS does not cope with lost observations, where sometimes the feedback signal is lost.

Due to the lack of a generic theoretic approach, an application specific solution is proposed in the present research that can address the practical problem as described above. A remotely controlled ground vehicle (the robot) is guided by an RL-trained agent to its target. The robot can move with a time step of ΔT=1sec towards three directions: left, right or forward. The positioning system incorporates the machine vision subsystem which detects an object on top of the robot and extracts a feature vector with its planar coordinates. The object detection system runs a YOLOv3 Tiny algorithm (Redmon & Farhadi, 2018) and has position update frequency of 15 positions per second and a lag of $T_L \sim 260$ msec.

The feature vector is fed to a Kalman filter that serves a double purpose.

- First, it compensates for lost samples since the object detection system may fail to detect the robot under varying conditions of lighting and view angle.
- Second, it predicts the current position $S_t + T_L$ ahead of the provided position S_t.

When the robot reaches its target, an acknowledge signal is issued; this is received through the control channel without delay. At the time of its arrival, though, the provided position already lags by T_L. If there is no compensation for this lag, then the algorithm should either wait for T_L (until the RL agent will have yielded a stable position reading) or should treat

the delayed CDMDP as an MDP and ignore the delay. This solution does not cope with the problem of missed position data and is therefore suboptimal, since it has to either cause a delay to the vehicle movement or result in the vehicle missing its target and moving out of the detection field. The object detection subsystem produces position vectors much faster than the time step ΔT. This can be exploited by a linear estimator like the Kalman filter that is suited for tracking moving targets.

CHAPTER 3:
Methods and Tools

Figure 1 shows a block diagram of the experiment setup built for the purposes of this research. A personal computer is hosting the Object Detection Server (ODS) with YOLOv3 Tiny. It runs asynchronously to the vehicle control agent and provides coordinates using socket based inter-process communication. The ODS is trained to provide the position of a remotely controlled vehicle via a camera mounted vertically in a fixed position above the vehicle testing ground.

The vehicle coordinates are fed to a control agent that decides the next move of the vehicle. This setup facilitates the investigation of solutions and tools needed in order to apply the modern methods of object detection and reinforcement learning to a real world problem, as described in the previous sections. The algorithms are selected to realize a state-of-the-art approach with robust solutions. The control agent will use a reliable Deep Q Learning algorithm in conjunction with a two-stage Kalman filtering that will assist in estimating the true position from the time delayed ODS provided position.

In next paragraphs, the subsystems of the experimental prototype are briefly introduced and discussed. Application-specific details of the implementation along with the experimental results follow.

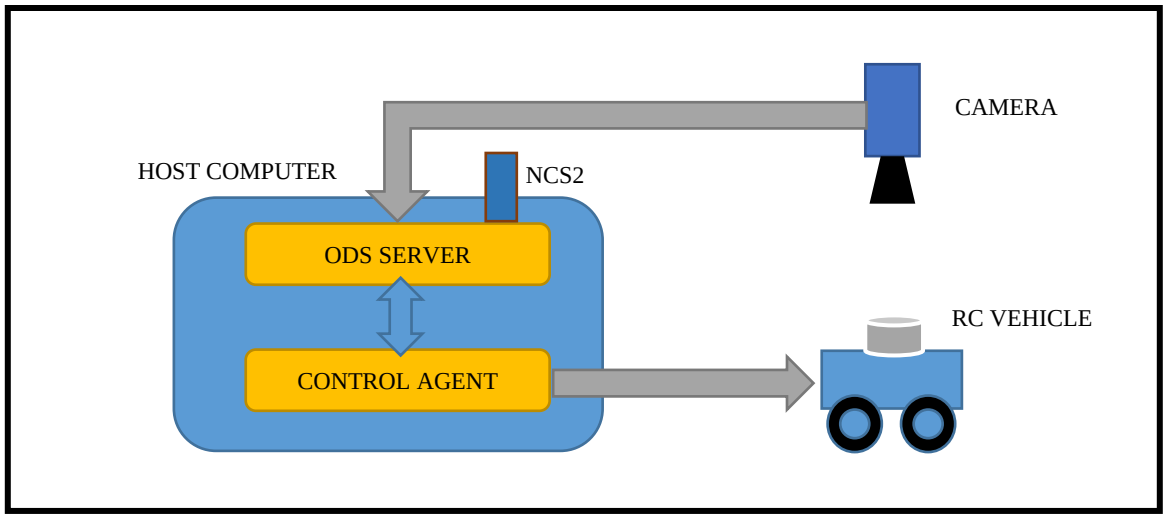


Figure 1: Experiment setup. The Host Computer runs the ODS and the Control Agent software. The ODS is using the Neural Compute Stick 2 (NCS2) as an accelerator for the YOLOv3 Tiny. The Camera is a typical commercial USB Web Camera (Logitech C920 HD Pro).

3.1. Object Detection with the YOLOv3

The early attempts to object detection in images using CNNs relied on ***double stage detectors*** like R-CNN (Girshick, Donahue, Darrell, & Malik, 2013), Fast R-CNN (Girshick, 2015), Faster R-CNN (Ren, He, Girshick, & Sun, 2015) and R-FCN (Dai, Li, He, & Sun, 2016). This double setup with a region proposal network followed by a classification network proved to have superior accuracy at the cost of poor speed; thus it was not suitable for real time video processing. The advent of ***single stage detectors*** with the most prominent works of Redmon (2015) who employed YOLO & derivative algorithm and Liu et al. (2016) who used SSD, have made real time applications feasible. Among the members of this family of algorithms, the ***YOLOv3 Tiny algorithm*** is chosen for this research work.

YOLO (Redmon, Divvala, Girshick, & Farhadi, 2015) along with its incremental improvements known as YOLOv2, YOLO9000 (Redmon & Farhadi, 2016) and YOLOv3

(Redmon & Farhadi, 2018) offer the best real time performance at 22msec (45fps) execution time.

The idea behind YOLO is simple: the same (single) network is used for detection and region proposals. The image under processing is split into a grid of SxS. For each grid cell, B bounded boxes are created along with their confidence and C conditional class probabilities. For the actual YOLO implementation, the parameter values are S=7, B=2 and C=20, meaning that for each of the 7x7 cells, 2 bounding boxes are proposed, with a confidence metric and a vector of size of 20 containing the probability for each class. Each bounding box has 4 coordinates and a confidence number associated with it. The output of the network, therefore, is 7 x 7 x (2 x 5 + 20) = 1470 proposals. The output of the network is then processed to eliminate duplicate boxes or boxes that contain classes with very low probabilities.

The architectural structure of the original YOLO implementation, based on GoogLeNet (Szegedy et al., 2015), has 24 convolutional layers that feed 2 fully connected layers. An alternative Fast YOLO implementation contains only 9 convolutional layers and achieves a processing rate of 155 frames per second, at the cost of reduced accuracy. The mean average precision (mAP) of YOLO is 63.4% and that of Fast YOLO is 52.7% mAP using the PASCAL VOC2007 test set (Redmon et al., 2015). By its design, each grid cell of YOLO may contain only one class. This means that only a limited number of objects may be detected. YOLO is not able to detect small objects that are located close to each other. On the other hand, since the size of bounding boxes is arbitrary and they may have practically any size, YOLO generalizes well on the image content, meaning that there are few background errors.

Tough competition with SSD (Liu et al., 2016) brought a new version of YOLO, YOLOv2 and YOLO9000 (Redmon & Farhadi, 2016). YOLOv2 incorporates several modifications to cope with the main disadvantage of YOLO that is localization errors.

The last fully connected layers were removed and replaced by convolutional layers so that the network could adapt to various image resolutions. Now it could be trained with a mix of high and low resolution images to provide finer detection. The arbitrary bounding boxes were replaced by boxes automatically adjusted during training. Mixed classification and object detection data was used for training as a method to achieve robust detection.

The YOLO CNN required 8.5 billion operations to complete a forward pass. YOLOv2 replaced this architecture with a novel architecture called Darknet-19 that has 19 convolutional layers and 5 max pooling layers. This required 5.6 billion operations for a forward pass resulting in a speed boost while keeping top accuracy performance of 78.6% mAP at 40fps. Using the combined dataset of COCO (Lin et al., 2014) and ImageNet, Redmon proposed YOLO9000 that can detect 9000 object categories with 19.7% mAP.

The next incremental proposal of YOLOv3 (Redmon & Farhadi, 2018) trades speed for improved accuracy. Darknet-19 is replaced by a 53-layer CNN unsurprisingly named Darknet-53. Softmax classification is replaced by independent logistic classifiers. Since classes are not mutual exclusive, overlapping labels may coexist allowing for training on other data sets (Kuznetsova et al., 2018). Using a similar idea from FPN (Lin, Dollar, et al., 2017) boxes are predicted at 3 different scales to improve small object detection. Its accuracy score (AP) is close to the top accuracy performer RetinaNet (Lin, Goyal, Girshick, He, & Dollár, 2017) and SSD (Liu et al., 2016) but being three times faster.

YOLOv3 Tiny is a reduced version suitable for resource-constrained systems. It is based on Darknet19 architecture with input images of 412x412. It includes 13 convolutional layers and can achieve a maximum processing rate of 220fps (Redmon, n.d.).

3.2. Kalman Filter

The Kalman filter (Kalman, 1960) is used to compute accurate estimates of unknown variables based on a series of available measurements. It is intended to be used with systems of linear dynamics that can be described by state space equations and can cope with measurements contaminated by Gaussian noise. The mean and covariance of the system state are predicted and updated through each time step. The Kalman filter algorithm consists of two steps, (i) the prediction step and (ii) the correction or update of the estimates step. Eqn.s 3-1 and 3-2 are the prediction step and eqn.s 3-3, 3-4 and 3-5 are the update step. This distinction is important, since in the case there is no new measurement, the estimated state is predicted by the dynamics of the system as described in eqn 3-1. If a new measurement does become available, then the uncertainty is mitigated since the estimated state produced by the prediction step is combined with the measured state, as in eqn 3-4. The two steps are outlined below.

(i) Prediction Step:

3-1) $X_k^- = AX_{k-1}^+ + BU_{k-1} + W_{k-1}$

Where: X_k^- is the predicted state vector at time step k

A is the state transition matrix

X_{k-1}^+ is the updated state vector from time step *k-1*

B is the control input matrix

U_{k-1} is the control vector at time step *k-1*

W_{k-1} is white Gaussian noise process vector from time step *k-1*

3-2) $P_k = A\,P_{k-1}^+ A^T + Q$

Where: P_k is the error covariance matrix at time step k

P_{k-1}^+ is the updated error covariance matrix from time step *k-1*

Q is the covariance matrix of the noise process vector

(ii) Update Step:

3-3) $K_k = P_k\, H^T\, (H\ P_k\ H^T + R\ )^{-1}$

Where: K_k is the Kalman Gain at time step k

H is the observation matrix

R is the covariance matrix of measurement noise vector

3-4) $X_k^+ = X_k^- + K_k(Y - HX_{k-1})$

Where: X_k^+ is the updated state vector at time step k

Y is the measurement vector

3-5) $P_k^+ = (I - K_k H)P_k$

Where: P_k^+ is the updated error covariance matrix

State and output noise covariance matrices Q and R are considered known. The Kalman filter is an iterative algorithm on k that converges to a state vector after appropriate initialization of the involved matrices for $k=0$.

3.3. Deep Q Learning

Reinforcement Learning (RL) has its origins in the problems of optimal control (Bellman, 1957) and is a development out of the widely used mathematical process known as dynamic programming (DP). Dynamic programming methods can be used only if the dynamics of the system are known. A method that learns the dynamics of the system of interest by itself was not easy to perceive a few years ago. The bases of modern RL were set during the 1980's, when distinct research concepts converged to the seminal work of (Watkins, 1989) that introduced the ***Q-Learning algorithm*** and its convergence properties (Watkins & Dayan, 1992).

RL algorithms are divided in model free and model based ones. In the model-based RL algorithms, the model of the environment is considered known and the algorithm is trying to find the optimal policy that will maximize the 'reward'. Using this setup, a

model can be used to evaluate the results of an action *without taking the action*. In the model-free RL algorithms, the results of an action are not known a-priori and a trial-and-error behavior (exploration) is necessary.

RL is based on the formalization of Markov Decision Processes (MDP). MDP is a decision-making policy that breaks down every system into a set of states, actions and rewards. At each time step t, an 'agent' is positioned within the environment with a state S_t. The agent is trainable. Using this state S_t, the agent should decide its next action A_t. At time step t+1, the agent will end up in state S_{t+1} with probability $P[S_{t+1}| S_t, A_t]$ and will be rewarded with R_{t+1}. This MDP framework can be summarized as a tuple $<S, A, p, r, \gamma>$ where S is a set of states, A is a set of actions, $p= P[s, a, s']$ is a state transition matrix, $r=R[s', a]$ is the reward function and γ is the discount factor. The agent should select actions from within the action set A on the basis of a policy π. The goal is to find the optimal policy π^* that maximizes the accumulated reward G_t as shown in eqn 3-6.

3-6) $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-1} R_T$

A well known example of 'learning a policy' is the training of an agent (a software program) to play a digital strategy game. The RL approach to training the agent is to lead the agent to discover the best policy by having the agent play a large number of game sessions while adjusting its policy.

There are two main approaches to learning a policy. One approach that led to the policy gradient method (Sutton, McAllester, Singh, & Mansour, 2000; Williams, 1992) is to learn the policy directly as a distribution over actions. The other approach is value-based. Here, the policy is derived by following an ε-greedy policy on functions of the states, v(s), (eqn 3-7) or functions of the state s and action a, $q(s,a)$, (eqn 3-8). When the state-action function $q(s,a)$ is defined as in the Bellman optimality equation (eqn 3-9),

then the Q-learning algorithm can be used to obtain the optimal policy π that maximizes q^* using an iterative update (eqn 3-10).

3-7) $v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S = s] \; \forall \; s \in S$

3-8) $q_\pi(s, \alpha) = E_\pi[G_t|S_t = s, A_t = a\,] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S = s] \; \forall \; s, a \in S, A$

3-9) $q^*(s, \alpha) = E_\pi \left[R_t + \max_{a'} q_*(S_{t+1}, a')|S_t = s, A_t = a\right] \forall \; s, a \in S, A$

3-10) $q^*(S_t, A_t) \leftarrow q_*(S_t, A_t) + a\left(R_t + \gamma \max_{a'} q_*(S_{t+1}, a') - q_*(S_t, A_t)\right)$

Eqn 3-10 implies that an RL problem can be solved by keeping a table for every *q(s, a)*. This is practical only for MDPs that have limited number of (state, action) pairs. If the number of pairs is very large or infinite, then a function approximation method should be used that typically can be a linear combination of features or an artificial neural network (ANN). The linear combination of features (Bertsekas, 2012) requires the use of extracted features while the neural network has been proven to work with raw sensory input (Mnih et al., 2013). The use of a function approximation method has also the advantage that an RL system can cope with state-action pairs never seen before.

Consider a Deep Q Network (DQN) that is using an ANN with weights w as a function approximator. The weights are found by the Stochastic Gradient Descent (Ruder, 2017) to minimize the loss function (eqn 3-11).

3-11) $L(w) = E\left[\left(y(\overline{w}) - q(S_t, A_t; w)\right)^2\right]$

Where the target value is *:*

$$3\text{-}12)\quad y(\bar{w}) = \begin{cases} R_t\,, if\ s\ is\ terminal \\ R_t + \gamma \max\limits_{a'} q_{\bar{w}}(S_{t+1}, a'; w)\,, if\ s\ is\ non\ terminal \end{cases}$$

The computer memory is used to store a large number of (S_t, A_t, R_t, S_{t+1}) samples. The SGD optimization is used with a random batch of samples from this memory. This method is called *experience replay* (Lin, 1992) and it is an effective measure to minimize the correlation when training with recent trajectories. From eqn 3-11 and eqn 3-12 it is shown that two distinct ANN weights are used, w that is called the on-line network and $\bar{w}$ that is called the target network. The gradient descent optimization is applied only to the online network that is used as a forward pass during exploitation where the target network is updated periodically from the online network. This duality of target and online networks ensures stability during training. Many improvements have been proposed to the DQN algorithm (Hessel, 2017); however, the basic setup as described here is adequate for the context of the present work.

CHAPTER 4:
Deep Networks Training and System Integration

In this chapter are presented the training of the two neural networks, the object detection algorithm YOLOv3 Tiny and the DQN RL agent.

4.1. The remote controlled vehicle

A three wheel vehicle is remotely controlled by a Bluetooth Low Energy (BLE) communications link as shown in Figure 2. The vehicle has a free running wheel and two wheels driven by DC motors (Figure 3). The supply of the DC motors is provided by an H bridge controlled by IoT Multi Sensor Development Kit (IoT MSDK). The IoT MSDK ("DA14585 IoT Multi Sensor Development Kit," 2018) contains a number of sensors with an extension connector, from which 2 general purpose IO's (GPIO's) are used to control the H bridge. The power source of the motors is a pack of batteries that provide 3.6V and it is not the same power source of IoT MSDK that includes 2 AAA batteries. The IoT MSDK is a BLE peripheral with a GATT server. It contains a control characteristic, for which three custom commands are used for the three allowable movements (forward, left and right) plus a disable/enable all command. All actions last for 1 second, then a completion acknowledgement is sent back to the central device. The BLE central device is a Dialog Semiconductor DA14585 Kit Basic ("DA14585 Development Kit-Basic," 2017), for which a GATT client application is developed that sends commands to IoT MSDK (GATT server) and awaits for acknowledgment. The central device communicates with the central computer using a virtual RS232 communication port over a USB connection.

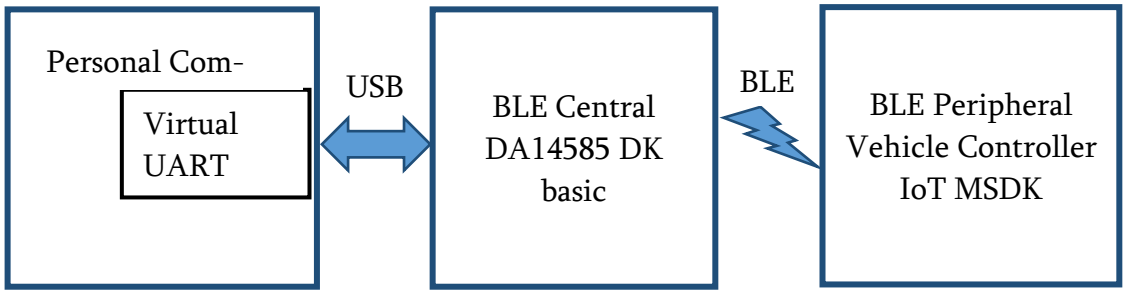


Figure 2: The remote control of the guided vehicle. A Bluetooth Low Energy link is used. The central device (GATT Client) is communicating with the host computer using a virtual UART over USB communications port. The Central device automatically scans and connects with the existing peripheral device (GATT Server).

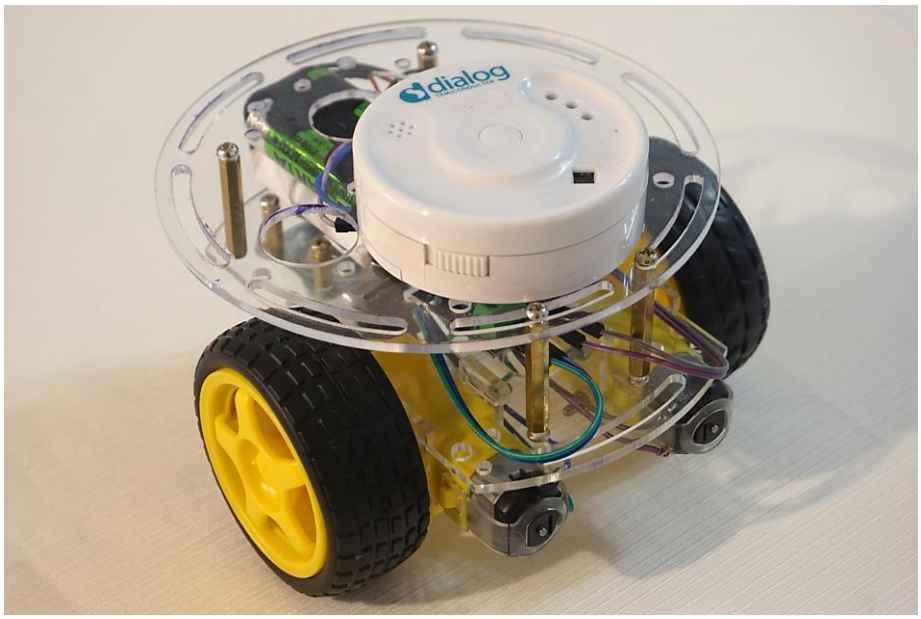


Figure 3: The remotely controlled vehicle. The ODS detects not the robot itself but the IoT MSDK attached on top. The vehicle chassis can thus be altered without having to retrain the ODS. The IoT MSDK Controls the H Bridge of the motors with its GPIO's.

4.2. Training YOLOv3 Tiny

YOLOv3 Tiny is the ODS algorithm of choice. It is suited for real time applications due to its frame rate performance, at least x4 compared to YOLOv3-320 (Redmon & Farhadi, 2018). The training environment of choice is the Darknet open source neural

network software package ("Darknet: Open Source Neural Networks in C," n.d.) that is maintained by Redmon. Darknet provides all the necessary components to train and run any flavor of YOLO, such as training configurations, runtime/inference for the extracted weights and automatic usage of a GPU. To train a CNN like YOLO, a large number of pictures have to be available and marked for every object class to be detected. There is not a strict rule on how many pictures are needed, the more the better. In practice, training with 300 images for a single object class provides acceptable results. The software used for marking images is YOLO Mark (AlexeyAB, n.d.). Marking means drawing a rectangle that encompasses the object of interest. For each picture a new file is created that contains the coordinates of the rectangle. For this procedure the following rules were applied:

- The marking should be tight but precise, meaning that the rectangle should neither encompass unnecessary space nor exclude any part of the object. Precision is important.
- If the object exists more than once in an image, it should be marked in all its instances.
- The object should be pictured, if possible, in all possible angles, lighting conditions and backgrounds that may occur during inference.
- To reduce false positives, not all images should be marked; images with plain background should also be used.

Using a DSLR camera, a set of 359 pictures of the IoT MSDK were taken and marked using the above rules (Figure 4). Each picture had 0 to 4 instances of IoT MSDK. Instead of 80 classes of the standard YOLOv3 configuration, a single class was used - the one of IoT MSDK. The rule of thumb for training YOLOv3 is that 2000 rounds per class are needed. The agent was trained for 5000 rounds (Figure 5) that took 16 hours on a PC equipped with a GPU. Longer training is not desirable since it might lead to overfitting.

After the completion of training, intermediate steps are needed to convert the Darknet CNN weights to the weights format needed by NCS2.

NCS2 (Neural Compute Stick 2, n.d.) is a neural network accelerator that comes in a USB stick form and offers the flexibility to work with different platforms. The software development platform for NCS2 is OpenVINO Toolkit (OpenVINO, n.d.).

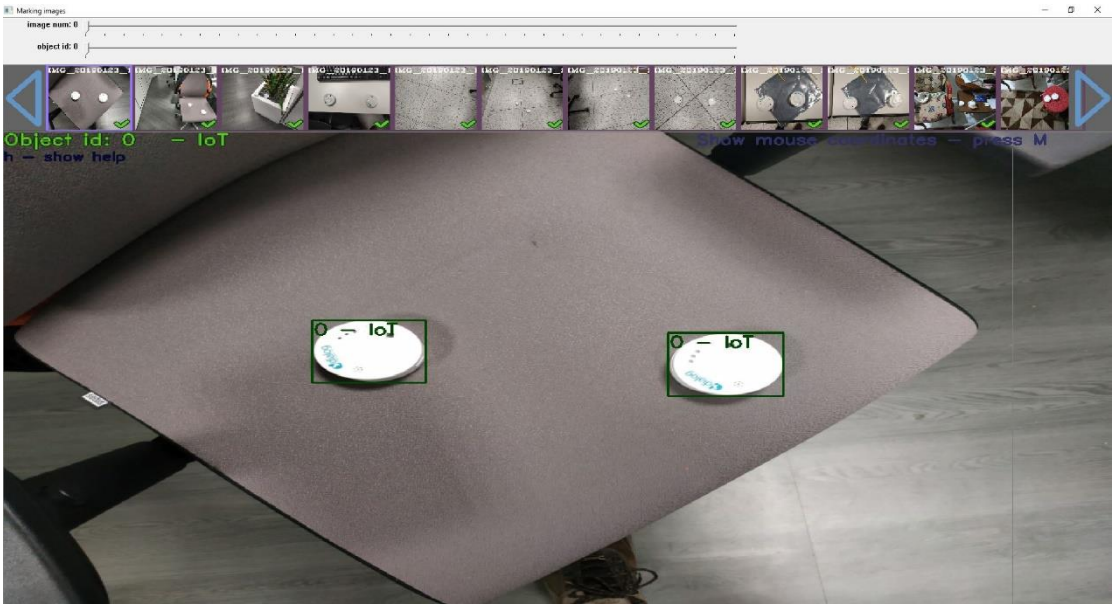


Figure 4: Marking through the Yolo Mark utility is time consuming: at least 300 pictures per category should be marked. Precision is important: the rectangles should accurately enclose the objects.

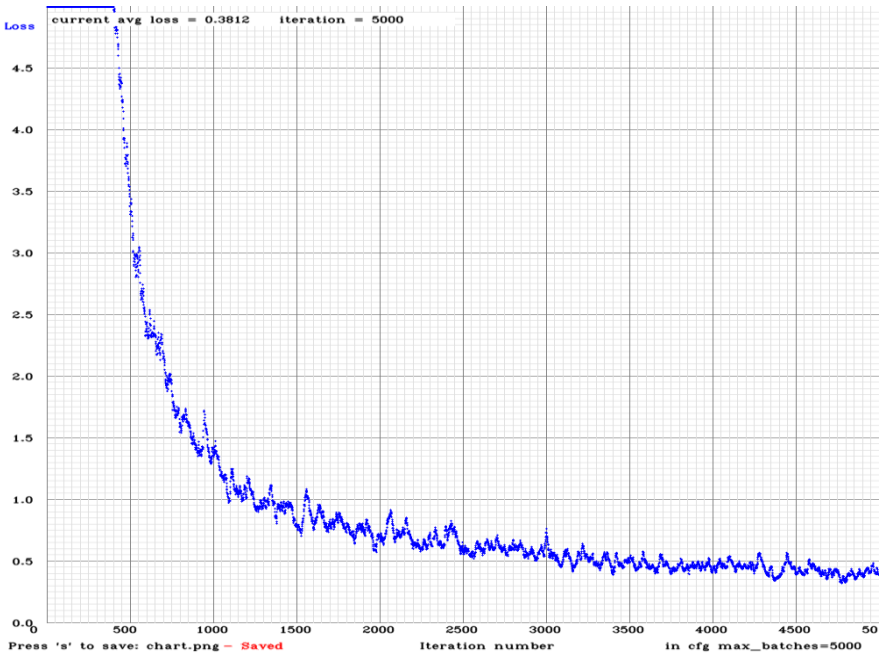


Figure 5: Training YOLOv3 Tiny loss versus iterations. The rule of thumb is that at least 2000 iterations are needed per object category. It was trained for a single category for at least 5000 iterations.

The Darknet weights should be converted to Tensorflow weights (Kapica, n.d.). At first the YOLOv3 was trained using a modified Darknet configuration file for one class of objects (IoT MSDK). In order to be compatible with the conversion utility, it was then retrained using the 20 class configuration that is provided for the VOC dataset. The Tensorflow weights are converted to the IR model used by NCS2 via the IR Model Optimizer utility provided by OpenVINO.

Both YOLOv3 Tiny using Darknet Demo and NCS2 provided demo for YOLOv3 Tiny were benchmarked. A significant speed increase was noted by using the NCS2. The Darknet Demo exhibited maximum performance at ~10fps (Figure 6) versus the ~15fps of NCS2 (Figure 7). The NCS2 is using an FP16 accelerator versus the FP32 GPU exploited by Darknet. Moreover, the NCS2 YOLOv3 Tiny demo is running in optimized asynchronous mode where the images are pipelined in the stages of acquisition, transfer and inference. As shown in Figure 7, the extracted coordinates of the NCS2 port lack

accuracy, this can be explained by the rounding errors of the IR Model Optimizer. This did not have an impact in the present experiments because only the center of the object position is used while the extensions of the rectangle boundaries are symmetric.

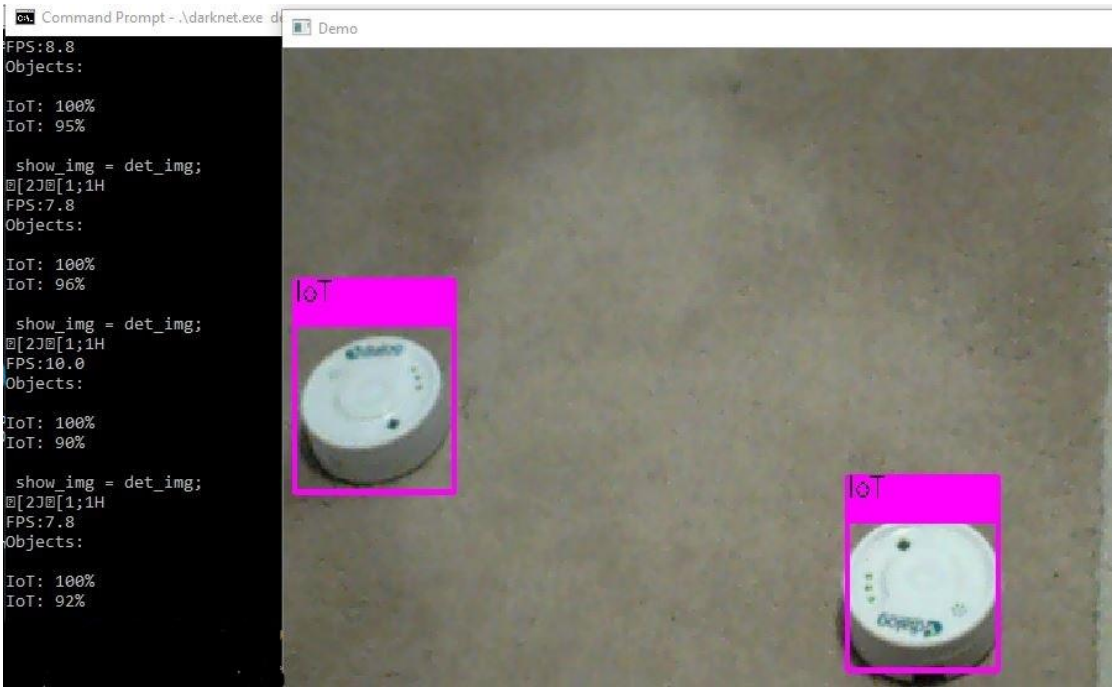


Figure 6: Running YOLOv3 Tiny natively with Darknet using a GPU.

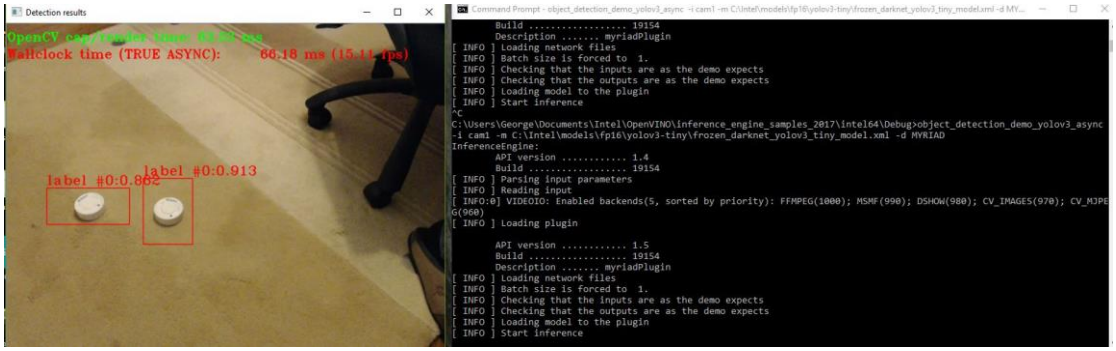


Figure 7: Running YOLOv3 Tiny with NCS2. The weights are converted to NCS2 format using the provided OpenVINO utilities.

4.3. Training the RL agent

The 'environment' considered here is a ground vehicle monitored by an object detection system that provides coordinates (X_t, Y_t) at every time step ΔT. The objective is το train an RL agent that can guide the vehicle from a starting point to its target (destination point) located at (X_{TRG}, Y_{TRG}). The target is fixed at each episode. Each episode ends when the vehicle reaches the target or when it moves out of bounds, meaning that it has left the field of view of the camera. The ODS is developed as described in Section 4.2. The state of the system is defined as vector S_t :

4-1) $S_t = (X_t, Y_t, X_{t-1}, Y_{t-1}, X_{TRG}, Y_{TRG})^T$

Where: X_t, Y_t are the vehicle Cartesian coordinates of the vehicle at time step t

X_{t-1}, Y_{t-1} are the vehicle Cartesian coordinates of the vehicle at time step t-ΔT.

X_{TRG}, Y_{TRG} are the target Cartesian coordinates that do not change over time.

The RL agent is trained using Deep Q Learning to control the vehicle in order to be able to accomplish a 'mission', that is to set off from its origin position and reach the target. The ODS camera is positioned vertically 2m above the level of the vehicle and has a field of view of 1.7mx1.3m with an analysis of 640 x 480 pixels. As shown in Figure 8, the starting point of the vehicle is at coordinates (60, 240) at 0 degrees angle and the target could be any point within the rectangle coordinates (420, 60) and (580, 420).

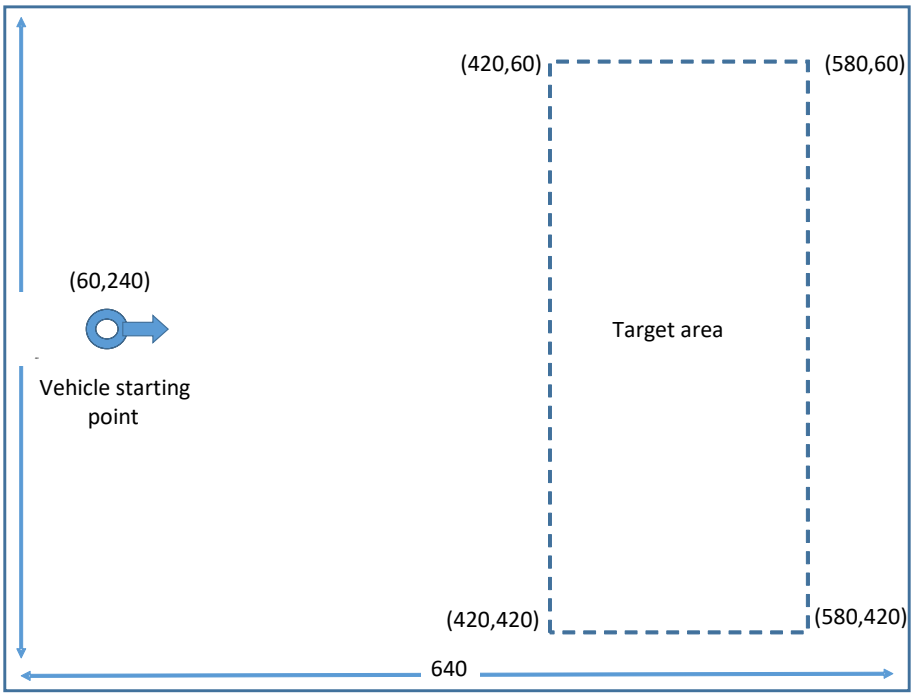


Figure 8: Training environment setup. This is the field of the expected view of the camera, the plane that the RC vehicle moves. The target area is on the right hand side. The plane area is 640x480 pixels, this resolution gives enough detail for the ODS to recognize the IoT MSDK on top of the RC vehicle but also delivers enough frame rate to run in real time.

At each time step *t*, the RL prompts for one of the 3 actions available, namely, 'move forward', 'move left 40º' or 'move right 40º'. All actions last for time ΔT=1 second. In order to train the agent, first an OpenAI Gym (Gym, n.d.) compatible environment is developed that simulates the movement of the vehicle. This compatibility ensures the reuse of this work for future research with other agents. In order to facilitate exploration, convergence and robustness the developed Gym environment followed these rules:

- When the vehicle reaches the target, the starting position becomes the new target. When the vehicle reaches the starting position, a new random target is generated. This 'round-trip' is considered a *mission*. The *episode* ends when the simulated vehicle goes out of bounds. Typically an episode may last for many missions.
- The state vector S_t (eqn 4-1) has 6 additional inputs reserved for future use.
- The magnitude of the speed vector is a uniform random variable between 55 and 65 pixels per second.
- The reward for reaching a target is 100, for moving towards the target -1, -2 otherwise.

To find the optimal architecture and hyperparameters for the ANN a two stage evaluation method is applied. A decision was taken to keep the number of layers constant (Table 1) and experiment with the number of cells and learning rate as shown in Table 2. The rest of the hyperparameters of Table 3 kept constant.

Table 1: There is a single hidden layer and the number of cells in layers 1 and 2 is the same. To enhance the robustness and avoid overfitting a Gaussian noise generator is added prior to activation function. The Gaussian noise generator is active only during training.

Layer Number	**Size** Number of cells	**Activation**	**Notes**
1	*N*, 12 inputs	ReLU	Input Cells, output is disturbed by additive Gaussian noise of standard deviation of 0.01. 6 inputs are used, the other 6 are always zero and added for redundancy in future research.
2	*N*	ReLU	Hidden Cells, output is disturbed by additive Gaussian noise of standard deviation of 0.01
3	3	Linear	Output Cells

Table 2: The hyperparameters that will be tested to find the best candidate.

Hyperparameters	Values
Number of Cells N	N=16, 32, 64, 128
l_r, Learning Rate	0.0005, 0.001, 0.005, 0.01

Table 3: The hyperparameters of the ANN that are kept constant through the training process.

Hyperparameters	Value	Notes
ε	max=1.0, min=0.1, decay=0.99995	Exploration rate
γ	0.98	Discount factor
Batch Size/Epoch	128/1	A random sample of Batch Size is used every K_{TR} time to perform a single fit operation.
Memory Length	100000	Memory Length, used for experience replay (Lin, 1992)
K_{TR}	4	Every K_{TR} number of steps a training operation is performed.
Loss Function	MSE	Mean Squared Error
Optimizer	Adam	Kingma & Lei Ba, 2014

A training agent with Keras/Tensorflow (Keras, n.d.) environment was developed and ran the Q Learning algorithm for at least 120K episodes. The first 8K episodes were running with ε=1 (exploration only) to get enough samples in experience memory. The accumulated reward and snapshot weights were logged and saved every 100 episodes to be used in later evaluation. As an example for N=128 and l_r=0.001 (Figure 9) the reward quickly escalated after episode round 45000 and made a dip around episode round 70000. The training procedure was visualized where, the trajectories of the agent were drawn with different colors depending on the reward acquired, a snapshot of a training procedure with color explanation is shown in Figure 10.

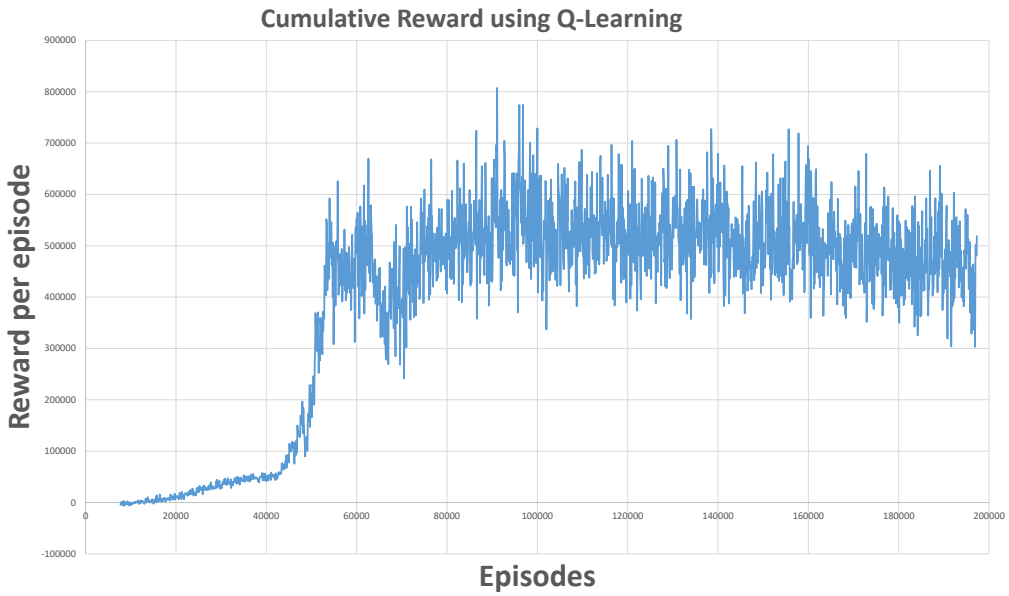


Figure 9: The episode reward versus the number of episodes (N=128, l_r=0.001). The reward is a moving average of the last 100 episodes reward. After certain number of episodes where the agent is trained enough, it completes a number of missions until episode termination, this is why the reward is reaching levels much larger than 100 that is the prize for reaching the target.

Since the stability of the resulted weights was not ensured, a second evaluation stage was added. From the log files, the top ten performers per hyperparameter combination were identified with respect to the logged reward. 160 files containing architectures and weights were passed through a second level of evaluation.

For each file, a set of 3 episodes was ran for 10 times. Each episode in a set started with a different target position from within the set of {(500, 60), (500,240), (500,420)}. When the first mission was accomplished, the new target was generated randomly within the target range limits (Figure 8). During evaluation, thousands of missions were thus included in an episode. The number of missions and reward per episode were logged. A flow of the training procedure is shown in Figure 11 while the final results from the evaluation are shown in Table 4.

Result #1 is a clear winner since it is ranked first with respect to average rounds and average score per episode. Also it has very low standard deviation (0.50% of average score), although result #3 is slightly better with 0.44%. 16 out of 20 results of Table 4 are of $N = 128$ and the rest 4 are of $N = 64$ starting from position #16. With respect to learning rate, the top 6 results are of $l_r = 0.0005$ and $N = 128$, while $l_r = 0.001$ appears 6 times (with $N = 64$ and $N = 128$) and $l_r = 0.005$ one time with $N = 64$.

In Table 4 the first entries of $N = 32$ and $N = 16$ are shown at positions 22 and 51 respectively. From the results it is shown that the hyperparameter combination of $N = 64$ and $l_r = 0.0005$ is a sweet spot that gives the best results according to our simulated evaluation. Also note that with a try of $N = 64$ and $l_r = 0.0001$ the agent failed to train. This may be fixed by readjusting the hyperparameters of Table 3 with lower ε decay but this will lead to lengthier training time.

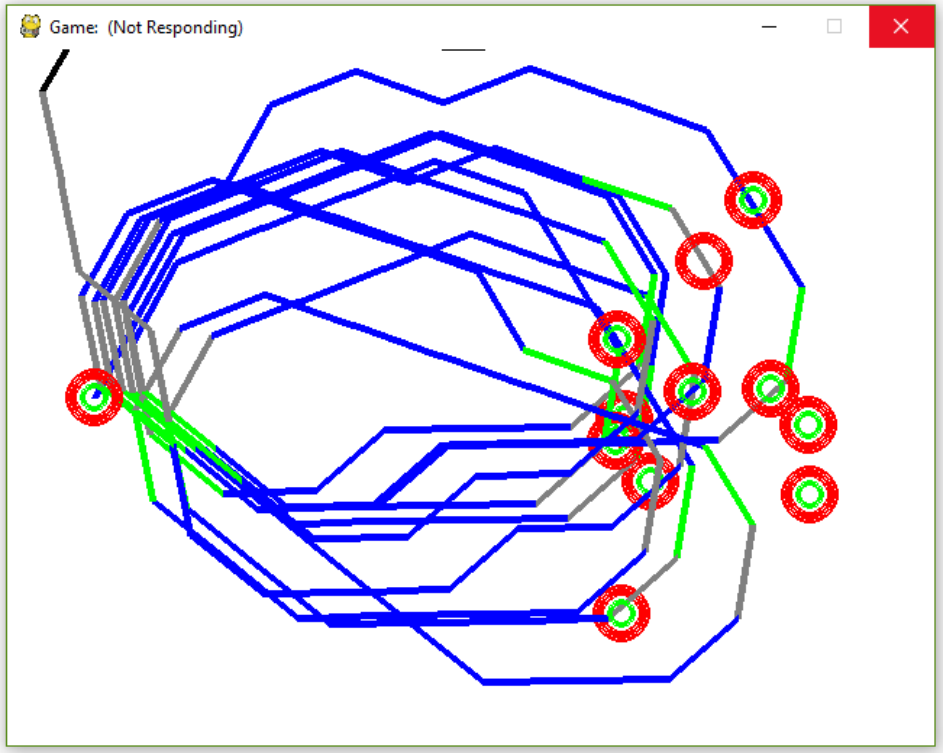


Figure 10: A snapshot from the training procedure. The agent performs multiple missions until it leaves the canvas (above left). The agent starts from center left, is guided to the target and then gets back to its origin. The colors are set to visualize the gains acquired: blue is -1 (distance from the designated target decreases), grey is -2 (distance from the designated target increases) and green is 100 (target reached). Note that the designated target will become the starting position (center left) after it reaches a random target (right).

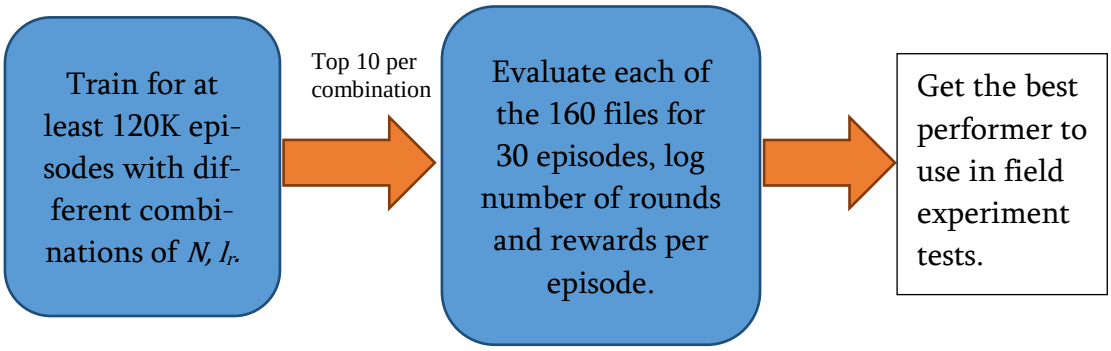


Figure 11: The flow of training and evaluation. For different combinations of N (16, 32, 64, 128) and l_r (0.0005, 0.001, 0.005, 0,01) the agent is trained for at least 120k episodes. The top ten performers per combination are passed through a second level evaluation that includes 30 simulated episodes per saved weight.

Table 4 : The evaluation results ranked with respect to average rounds. The standard deviation is used as an indication for the stability of the result. It is clear that N=128 and l_r=0.005 give the best results while N =16/32 are not good hyperparameters for our application

Rank	Episode#	*N*	l_r	AVERAGE ROUNDS	AVERAGE ROUNDS stdev%	AVERAGE SCORE	AVERAGE SCORE stdev%
1	124600	128	0.0005	3297.9	22.60	215381.97	0.50
2	120300	128	0.0005	3274.4	22.80	214069.00	5.95
3	107900	128	0.0005	3270.0	22.65	213401.23	0.44
4	123000	128	0.0005	3265.3	22.74	213393.97	6.08
5	113100	128	0.0005	3214.7	22.98	209897.37	7.29
6	100800	128	0.0005	3207.9	22.80	209241.33	5.06
7	196400	128	0.001	3159.5	23.91	207032.23	15.90
8	114700	128	0.0005	3162.8	23.68	206530.13	12.89
9	168300	128	0.001	3147.3	24.02	206378.87	17.50
10	122600	128	0.0005	3146.2	23.75	205536.90	14.05
11	161900	128	0.001	3143.1	23.76	205389.70	14.76
12	89000	128	0.0005	3127.0	22.52	200238.77	8.00
13	97800	128	0.0005	3112.8	23.04	202795.37	9.61
14	71600	128	0.0005	3097.1	22.56	201017.67	0.54
15	147900	64	0.0005	3088.7	22.68	195301.40	16.68
16	99400	128	0.001	3039.0	23.58	198752.67	13.95
17	86700	64	0.001	3036.6	22.58	196686.80	0.40
18	77300	64	0.001	3010.9	22.55	194852.03	0.38
19	87300	64	0.005	2989.9	24.80	195341.07	20.80
20	94900	128	0.0005	2967.3	24.87	194112.63	21.86
22	142100	32	0.001	2954.9	23.16	191436.56	9.13
52	184200	16	0.005	2388.2	30.6	148910.26	34.45

4.4. Integration

A number of software modules should be integrated to close the control loop (Figure 12). The following software components need to exchange data:

- The ODS needed to acquire images from the USB camera.
- The control agent needed to acquire the vehicle coordinates from ODS when available.
- The control agent needed to send the commands over the wireless link to the vehicle

The ODS used is a modified C++ demo software that was provided by Intel. This software uses OpenCV (OpenCV, n.d.) and NCS2 drivers to automatically detect and use an attached USB camera and the NCS2 USB stick. The ZeroMQ interprocess messaging library (ZeroMQ, n.d.) was used to implement a client/server architecture. The client was the control agent and the server was the ODS. The client was blocking for a specific time T_{ods} until it could get a pair of coordinates from the ODS. If there was a timeout then the agent concluded that there were no coordinate pairs available, meaning that the presence of the vehicle could not be detected within the viewing angle.
The control agent was the central data fusion and control system.

- It got the ODS coordinates or detected absence of measured coordinates
- It estimated the current position of the vehicle
- It ran the deep RL agent using the estimated position along with the previous position to extract a new command.
- It detected the end-to-end communication with the remote vehicle and sent commands.
- It waited for an acknowledgement from the vehicle's IoT MSDK for every command that was completed at each time step.

The control agent is written in Python and it uses a number of libraries to communicate with ODS and the RC vehicle. The BLE link acts transparently related to the control

agent, the BLE client forwards ASCII commands to the IoT MSDK and waits for an ASCII reply.

The inference engine for the RL is based on Tensorflow/Keras and does not require the existence of a GPU. The inference time is measured to be less than 5msec, even for a low-end CPU. This is because of the small number of layers used. This is an advantage because the deployment can be done even on a very small computer with limited resources, provided that it can execute the ODS. The Kalman filter with KALMANx1 and KALMANxN prediction function has been custom developed for this work, as an application-specific library.

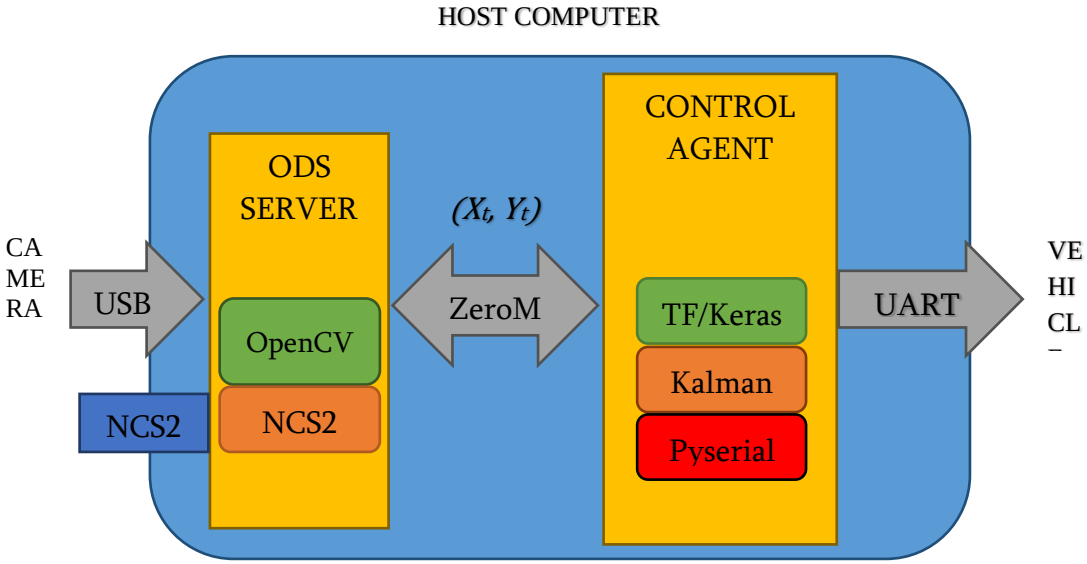


Figure 12: The ODS (C++) and the Control Agent (Python) were running in parallel as separated processes. Interprocess messaging was done by ZeroMQ library using a Linux Host machine. ODS integrated the camera interface and OpenVINO thus providing ~15 object positions per second. The Control Agent was not using hardware accelerator for the Deep RL algorithm. The RC vehicle was transparently controlled via a UART (Section 4.1).

CHAPTER 5:
Experiments

Figure 13: The camera is suspended 2m above the ground, with a 1.7x1.3m field of view.

A number of experiments are conducted using different agents and environment setups. The resulted trajectories are logged and analyzed. The basic experiment setup is shown in Figure 14 where the camera is mounted vertically with its axis perpendicular to the running floor. Detailed explanation for each agent model and environment will be provided in the subsequent sections. The tests start from a starting point (SP1) towards destination points (DP) named P1, P2, P3 (P1, P2, P3|SP1). A planar view of source and destination points are shown in Figure 14.

Table 5: Coordinates of source and destination points.

Source and Destination Points	Values x,y (pixels)
SP1	70, 400
P1	480,100
P2	80,100
P3	580,240
P4	480,380

Although the system is tested using the agent with the training setup of Figure 8 (P1, P3, P4|SP0) the experiments are performed with a slightly different setup.

First it is demonstrated that the trained RL agent is not overfitted and that it can cope with unseen environments. While P1 and P3 lie within the target training range, P2 has never been seen as a target. Second, by selecting SP1 instead of the trained SP0= (60, 240) the agent is allowed to travel a large distance towards P1, P3 and take a long turn towards P2.

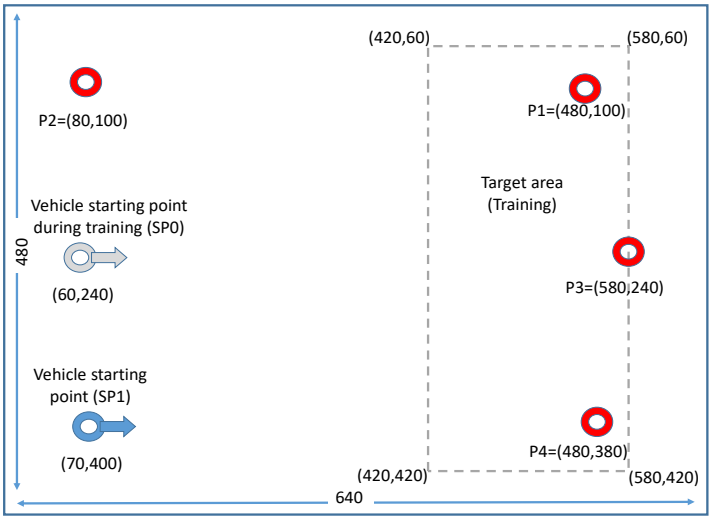


Figure 14: Planar view of source (SP0,SP1) and destination (P1,P2,P3,P4) points

For each test at least 10 trajectories are logged and averaged, compared and analyzed. Placing the vehicle with 1 pixel accuracy has not been possible, so during the

experiments the initial starting point (SP1) and angle of the vehicle lie within an acceptable small range. The environment of the vehicle was mildly stochastic meaning that $P[s, a, s'] < 1$. The distance and turn angle per time step are normally distributed (eqn 5-1 and eqn 5-2). The actual $\Delta T = 1.0$ sec for forward action and $\Delta T = 1.1$ sec for left or right action. The vehicle first rotates for $\Delta T_R = 0.1$ and then moves forward for $\Delta T_S = 1.0$ second (eqn 5-3).

5-1) $s_{\Delta TS} \sim N(S_{\Delta TS}, \sigma^2_{s\Delta TS}) = N(51.9, 109.1)$

Where : $s_{\Delta TS}$ is a sampled distance ran by the vehicle for ΔT_S

$S_{\Delta TS}$ is the mean distance ran by the vehicle for ΔT_S

$\sigma^2_{\Delta TS}$ is the variance of the distance ran by the vehicle for ΔT_S

5-2) $\theta_{\Delta TR} \sim N(\Theta_{\Delta TR}, \sigma^2_{\theta\Delta TR}) = N(29.8 , 17.3)$

Where : $\theta_{\Delta TS}$ is a sampled rotation angle in degrees of the vehicle for ΔT_R

$\Theta_{\Delta TS}$ is the mean rotation angle in degrees of the vehicle for ΔT_R

$\sigma^2_{\Delta TS}$ is variance of the rotation angle of the vehicle for ΔT_R

5-3) $\Delta T = \Delta T_S + \Delta T_R$

Where : ΔT is the duration of a command, that is 1.0 or 1.1 second

ΔT_R is the duration of rotation that is 0.1 second if the command is turn left or right

ΔT_S is the duration of forward movement, always 1.0 second

The actual distance and angle depend on the friction of the wheels, the starting torque of the motors and the momentary drop of the battery voltage. The IoT MSDK has the ability to detect orientation, thus an accurate angle turn can be forced; yet, this would

impose delays for accurate angle adjustment. Thus it has been decided not to use this approach and to favor swift turns and also avoid the requirement for an accurate inertial sensor. The vehicle guidance was based on the accuracy of the machine vision system and the RL agent and not on any on-board sensor.

Every trajectory is logged and the trajectory point P_{dmin} with the closest distance $d_{min,}$ to the target PN is computed. The trajectory distance and time from SP1 to P_{dmin} are calculated as S_{dmin} and T_{dmin} , respectively.

The default delay imposed by the USB camera and the ODS software was measured to be T_L=260msec. Any additional delay was emulated as a FIFO pipeline of the machine vision system. Thus T_L can be increased in multiples of T_{ods}=1/15fps=67msec.

5.1. The Wait Agent

The wait agent shown in Figure 15 is converting the CDMDP to MDP by waiting for T_L at the end of each time step ΔT. Thus if the trajectory is comprised of K time steps, the additional time delay will be $K \cdot \Delta T$. Using the wait agent, a number of trajectories are recorded for the trained setup P1, P3, P4 | SP0. From Figure 16 it can be seen that the agent adopts a hook shaped strategy, whereby it approaches the targets in such a way that it could return back to its starting point.

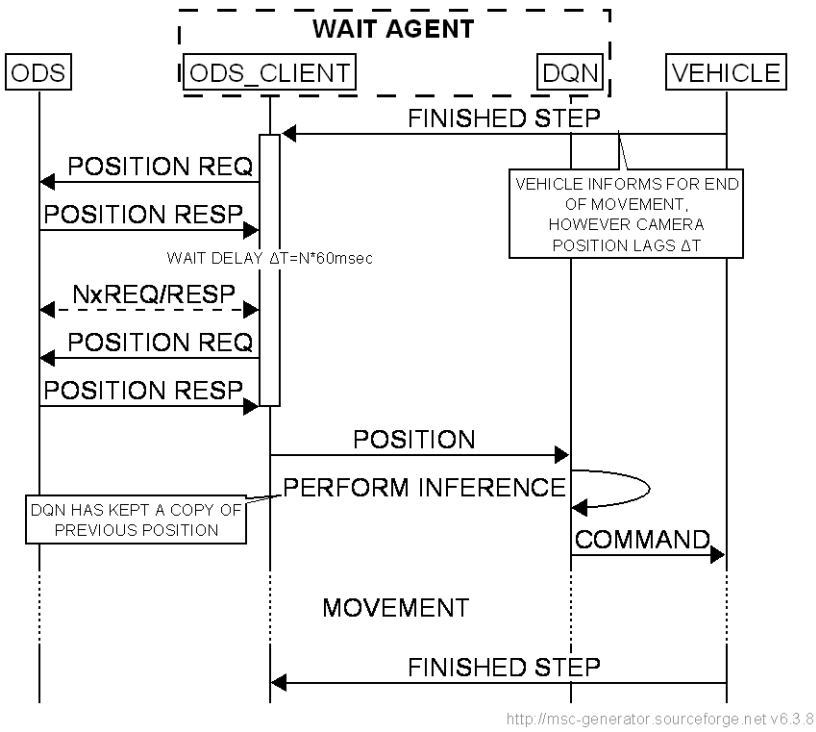


Figure 15: The wait agent is bypassing the problem of delay by waiting by T_L at the end of each time step thus imposing a total delay of $K \cdot \Delta T$.

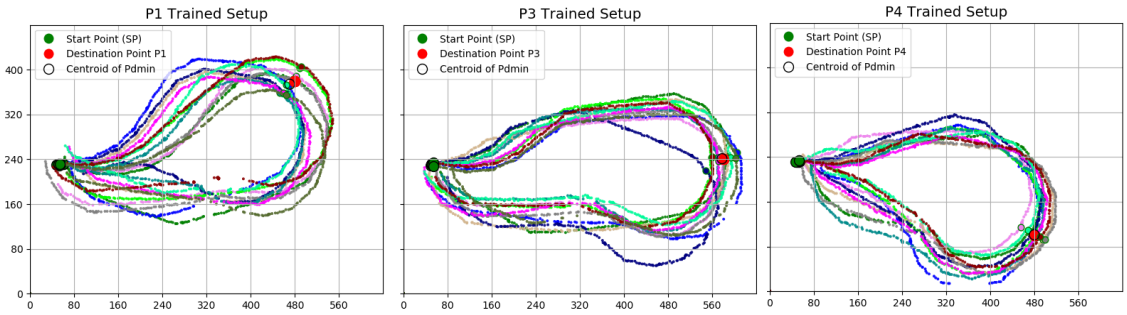


Figure 16: Experiments using the trained setup with P1, P3, P4/SP0, T_L=250msec. The agent adopted a hook like strategy to be able to return to the start point.

The next experiment approach is P1, P3, P4 | SP1 and T_L = 250msec. This approach is one way episodes, from starting point to destination point, to allow for traveling longer distances. Since with this experiment the agent was driven without having been

affected by the delay T_L, under ideal conditions but with suboptimal travel time T_{dmin}, these results are considered as reference results for the following experiments. The centroids PN_{cw}=$P1_{cw}$, $P2_{cw}$ and $P3_{cw}$ for each target were calculated from the P_{dmin}'s based on the wait agent results (Figure 17). Tracking the trajectories using the wait agent is not required, it is used only for analysis of the result.

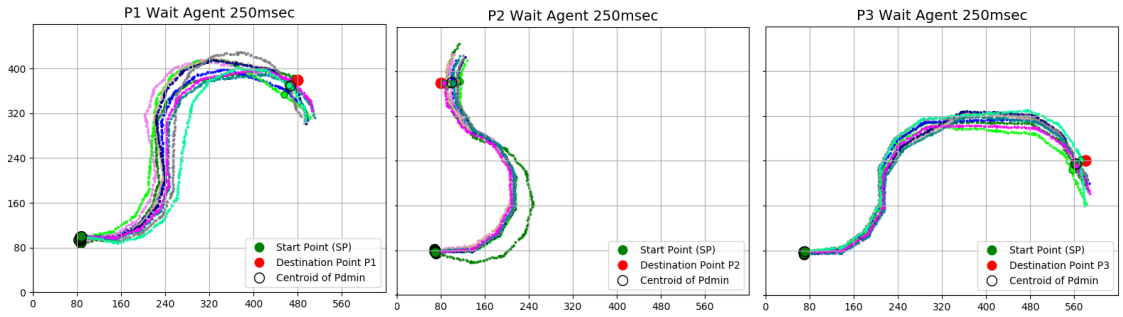


Figure 17: The wait agent trajectories for T_L=250ms using setup P1,P2,P3/SP1. Notice that the wait agent keeps the "hook" like trajectories. This allows to test with a more complex maneuvering than the shortest path that would be a straight line.

Table 6: The wait agent results for T_L=250msec. The centroid coordinates PN_{cw} for each test are calculated. A Hotelling T^2 test is performed that shows the difference between P1,P2 and P3 and the centroids $P1_{cw}$, $P2_{cw}$ and $P3_{cw}$.

Centroid PN_{cw}	**Coordinates of PN_{cw}** X,Y (pixels)	**Target** X,Y (pixels)	**Distance from target d_{min}** (pixels)/stdev	***T^2/F/F-crit***	**Average S_{dmin}** (pixels)/stdev	**Average T_{dmin}** (msec)/setdev
$P1_{cw250}$	467,110	P1 (480,100)	16/8.8	61.6/27.4/ 4.5	650/24	17475/748
$P2_{cw250}$	100,100	P2 (80,100)	20/10.3	47.9/21.0/4.7	471/31	10905/850
$P3_{cw250}$	585,246	P3 (580,240)	16/5.4	148.4/66.0/4.5	706/24	17046/648

5.2. The Naive Agent

The naive agent (Figure 18) treats the CDMDP like an MDP ignoring the T_L. When the vehicle sends a message that the step has been completed, the agent immediately proceeds to its inference step and uses the latest acquired coordinates sample from ODS.

Trajectories are logged for T_L=250, 500, 750 and 1000msec (Figure 19 through Figure 22). Tracking the trajectories in the naive agent is not required, it is used only for analysis of the result.

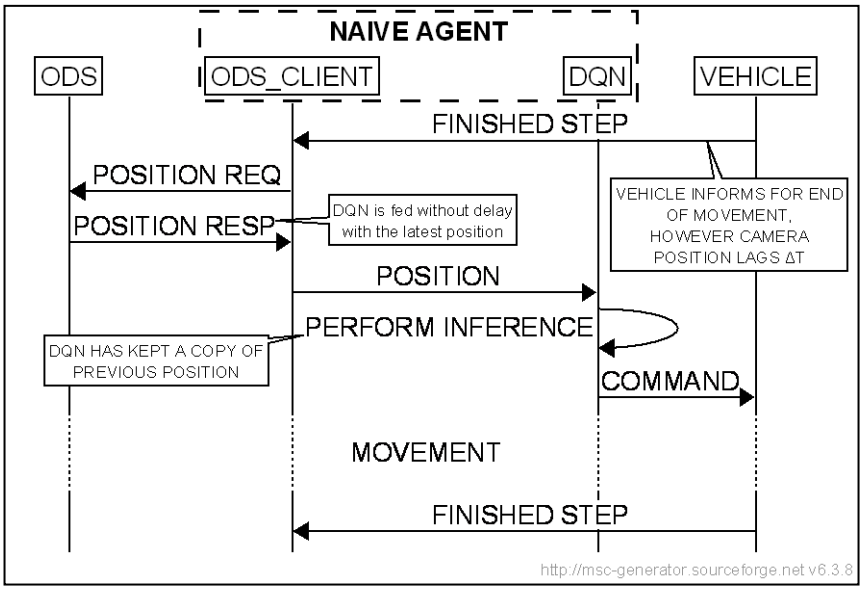


Figure 18: The naive agent treats the CDMDP like an MDP and ignors the delay T_L.

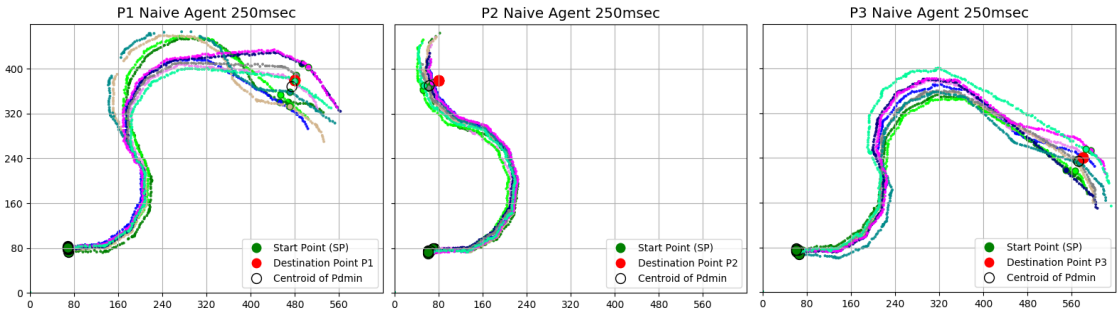


Figure 19: The naive agent with T_L=250msec

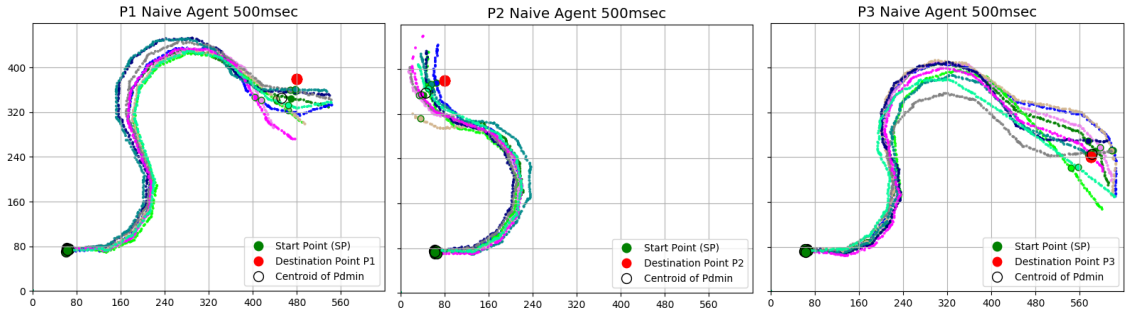


Figure 20: The naive agent with T_L=500msec

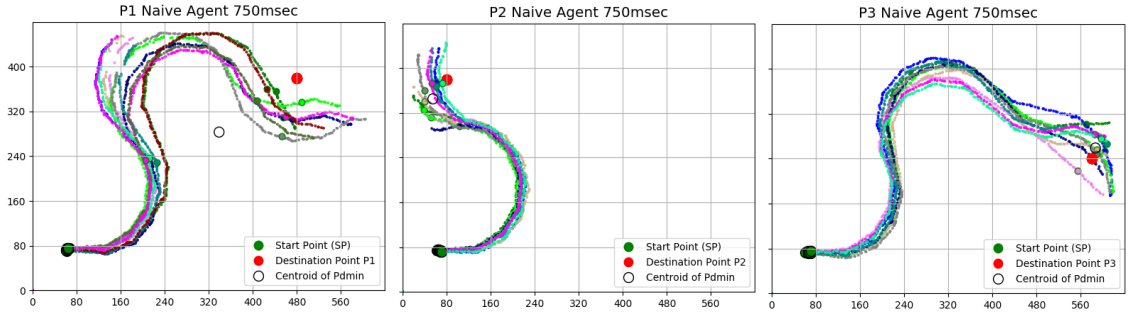


Figure 21: The naive agent with T_L=750msec

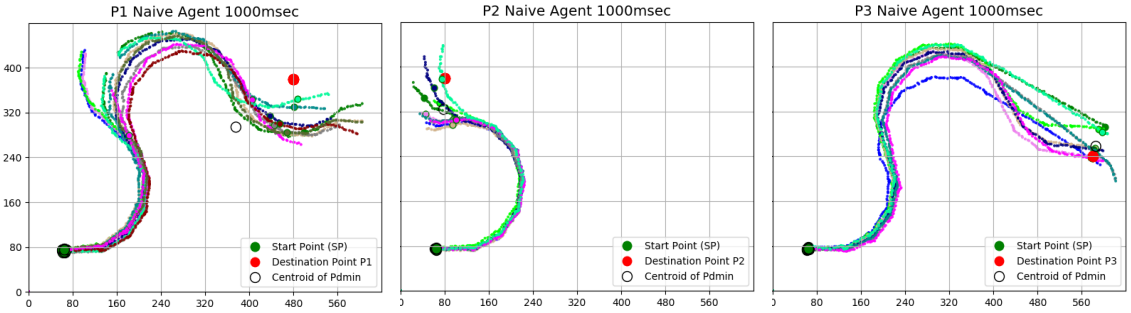


Figure 22: The naive agent with T_L=1000msec. 1/3 of the episodes of P1 ended when the vehicle left the field of view of the camera.

Table 7: Measurements using the naive agent with T_L=250, 500, 750 and 1000 msec

Centroid PN_{cn}	Coordinates of PN_{cn} X,Y (pixels)	Target X,Y (pixels)	Distance from target, d_{min} (pixels)/stdev	$T^2/F/F\text{-}crit$	Average S_{dmin} (pixels)/stdev	Average T_{dmin} (msec)/stdev
$P1_{cn250}$	475,111	P1 (480,100)	12/17.0	1.9/0.8/4.5	735/54	14292/1507
$P1_{cn500}$	453,135		43/21.7	86.3/38.4/4.5	745/43	15248/1027
$P1_{cn750}$	448,153		62/22.7	49.3/20.6/5.8	786/64	16963/1305
$P1_{cn1000}$	457,110		73/21.5	94.7/41.4/4.7	816/68	17829/1907
$P2_{cn250}$	63,109	P2 (80,100)	19/7.7	64.1/28.5/4.5	478/13	9718/386
$P2_{cn500}$	46,122		40/18.5	128.6/57.1/4.5	496/18	10515/686
$P2_{cn750}$	56,134		41/26.1	36.5/16.2/4.5	473/30	10354/608
$P2_{cn1000}$	78,155		55/26.7	60.2/26.8/4.5	455/27	9764/745
$P3_{cn250}$	574,245	P3 (580,240)	8/11.4	2.6/1.14/4.5	758/46	14607/960
$P3_{cn500}$	584,234		7/14.6	1.7/0.7/4.5	830/54	1726/1037
$P3_{cn750}$	587,220		20/17.8	7.5/3.4/4.5	830/37	17467/1081
$P3_{cn1000}$	586,222		19/21.7	7.2/3.2/4.5	868/41	18281/836

Table 7 summarizes the measurements using the naive agent. It is noted that PN_{cn} distance from target (d_{min}) increased when increasing T_L for every destination point. The average S_{dmin} also increased with the exception of $P2_{cn}$ where the agent was always taking a continuous turn. The same applies with T_{dmin} where with the exception of $P2_{cn}$, the time to target was increased from 14 seconds to 18 seconds.

A result that is not described in Table 7 but is shown in Figure 21 and Figure 22 is that almost 1/3 of the trajectories of P2 left the field of view, thus terminating the episodes without reaching the target. It is clear to us that this is the limit where the naive agent can reliably control the vehicle. If $T_L \geq 750$ three problems are seen,

- The distance to target $d_{min,}$ the travel distance to target S_{dmin} and the time to target T_{dmin} increase when T_L increases, thus leading to suboptimal control
- The vehicle can leave the field of view when the maneuvering takes place close to the edge of the canvas (e.g. P1).

- After $T_L \geq 1250$ no successful episode successful episode could be recorded, all trajectories ended up in loops or out of the field of view. The control loop became unstable and the vehicle uncontrollable.

5.3. The Kalman Agent

As a result of the shortcomings of the naive agent our idea is to use a Kalman filter to mitigate the results of the delay T_L. The data flow path is shown in Figure 23, the ODS_CLIENT is continuously feeding the KALMANx1 filter with position measurements. If there is a missing measurement due to inability to detect the vehicle the KALMANx1 works in prediction only mode (eqn 3-1 and eqn 3-2). The KALMANxN filter always predicts xN steps ahead using the estimation of KALMANx1 filter. The N is a multiplication factor of T_{ODS} sampling time, e.g. if T_L =1000msec, N= $(T_L/67)$-1. In practice since the naive agent with T_L =250msec works, to reduce prediction errors the values in Table 8 are used.

Table 8: Number of predictions of KALMANxN filter for every T_L

Delay Time T_L (msec)	KALMANxN Prediction steps N
250	0
500	4
750	8
1000	12
1250	16

The DQN running process is working only with the KALMANxN provided values, thus every input is linearly filtered. Unlike the wait and naive agent, this agent needs to be continuously fed with vehicle position measurements, so when the vehicle sends the command completion signal, the KALMANxN should be able to provide an estimate to the DQN run time.

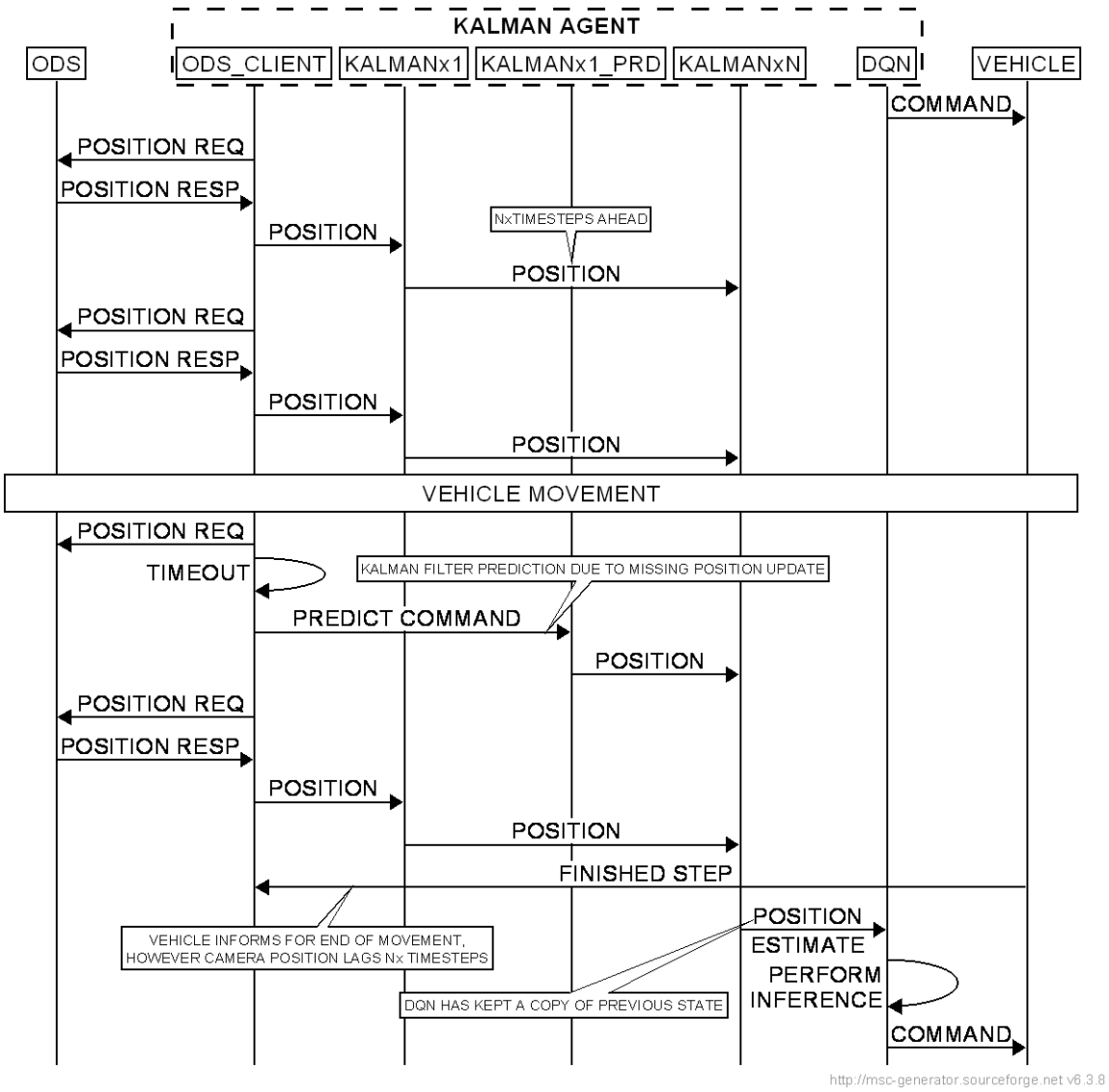


Figure 23: The Kalman agent. The Kalman filter works with x1 update and xN prediction depending on T_L.

The design methodology of the Kalman filter should be explained. At first the filter should estimate position, speed and acceleration from position measurements only. Second the measurement noise covariance matrix R (eqn 3-3) and process noise covariance matrix Q (eqn-3-2) should be found. R could be computed since measurements were readily available. On the other hand an effort was made to find Q by simulating an agent

with logged trajectories and trying to find the closest match of xN prediction with the original trajectory by using a mean squared error metric. This did not worked as planned in practice and finally Q was tuned running a number of experiments with T_L =1000msec using the simulation values as a starting point. A sample from these experiments is shown in Figure 24 while the full set of the Kalman filter matrices are shown from eqn 5-5 through eqn 5-10.

5-5) $$A = \begin{bmatrix} 1 & 0 & T_{ods} & 0 & T_{ods}^2 & 0 \\ 0 & 1 & 0 & T_{ods} & 0 & T_{ods}^2 \\ 0 & 0 & 1 & 0 & T_{ods} & 0 \\ 0 & 0 & 0 & 1 & 0 & T_{ods} \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

where: T_{ods}= 0.067 second

5-6) $$P = \begin{bmatrix} \Delta x^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \Delta y^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \Delta V_x^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \Delta V_y^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \Delta A_x^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \Delta A_x^2 \end{bmatrix}$$

where: P is the initial error covariance matrix.

5-7) $$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

where: H is the observation matrix

5-8) $$R = \begin{bmatrix} \Delta x^2 & 0 \\ 0 & \Delta y^2 \end{bmatrix}$$

where: R is the covariance matrix of measurement noise vector, with initial values $\Delta x^2 = \Delta y^2 = 0.4$

5-9) $$I = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

where: I is the identity matrix used in eqn 3-5

5-10) $$Q = \begin{bmatrix} \sigma x^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma y^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma V_x^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma V_y^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma A_x^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma A_x^2 \end{bmatrix}$$

where: Q is the covariance matrix of process noise vector, with initial values $\sigma x^2 = \sigma y^2 = 10$, $\sigma V_x^2 = \sigma V_y^2 = 1.0$ and $\sigma A_x^2 = \sigma A_x^2 = 1.0$.

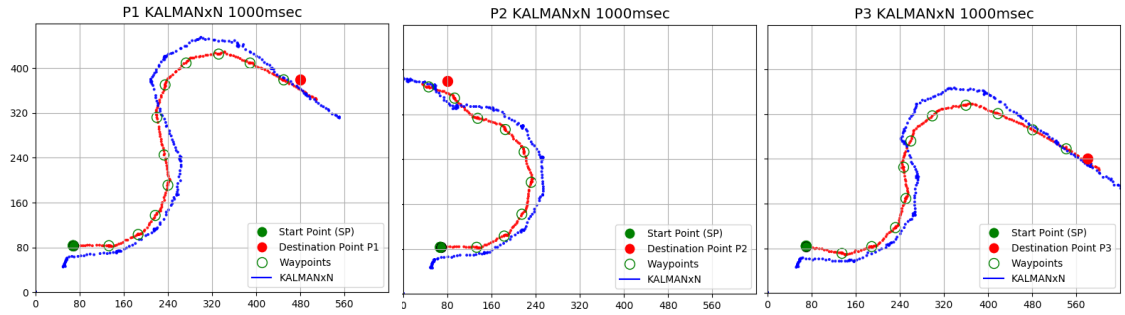


Figure 24: Samples of Kalman agent trajectories for T_L=1000msec. The blue line is the KALMANxN filter prediction. The green empty circles are the waypoints.

Using the Kalman agent a number of experiments were performed from T_L=250msec to T_L=1250msec. All the trajectories were logged and shown from Figure 25 to Figure 29. The experiment with P1| T_L=1250msec resulted in some trajectories leaving the plane, though the control was improved compared to the naive agent. This does not mean that the Kalman agent cannot perform well with T_L=1250msec with an optimized set of parameters. The full results, except T_L=1250msec, are shown in Table 9.

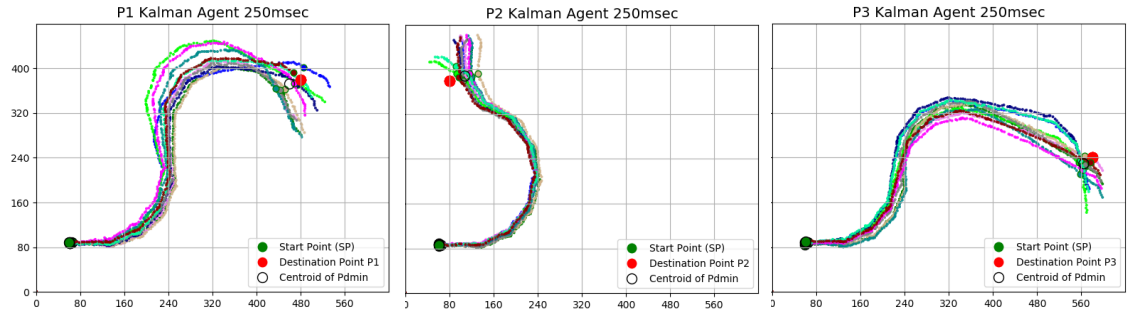


Figure 25: The Kalman agent with T_L =250msec

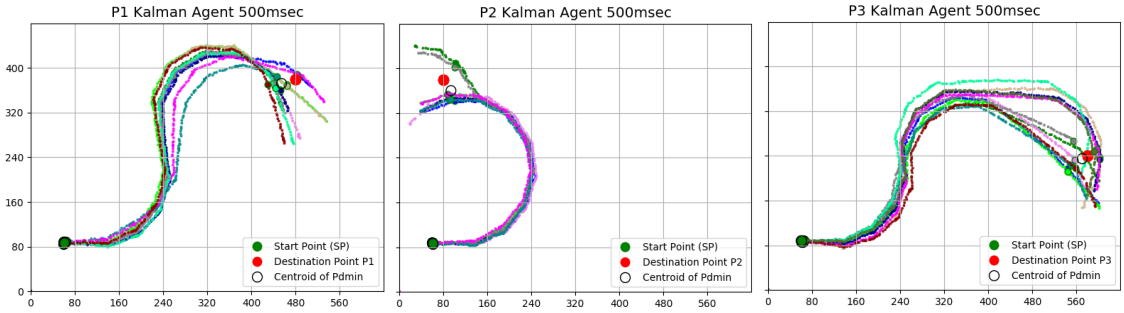


Figure 26: The Kalman agent with T_L =500msec

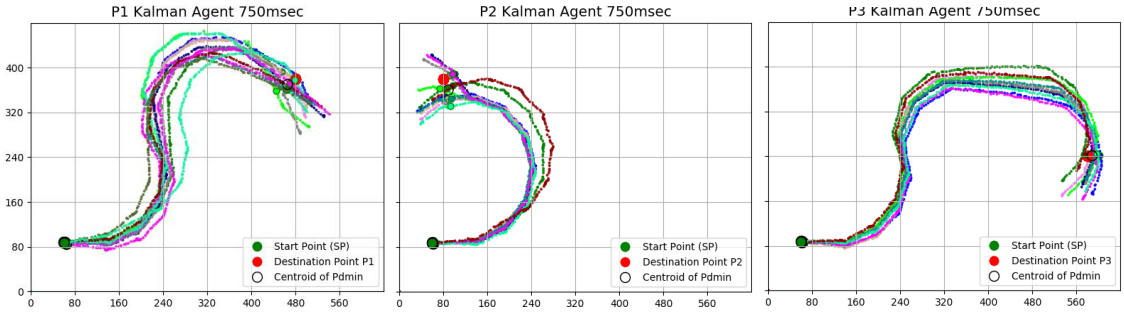


Figure 27: The Kalman agent with T_L =750msec

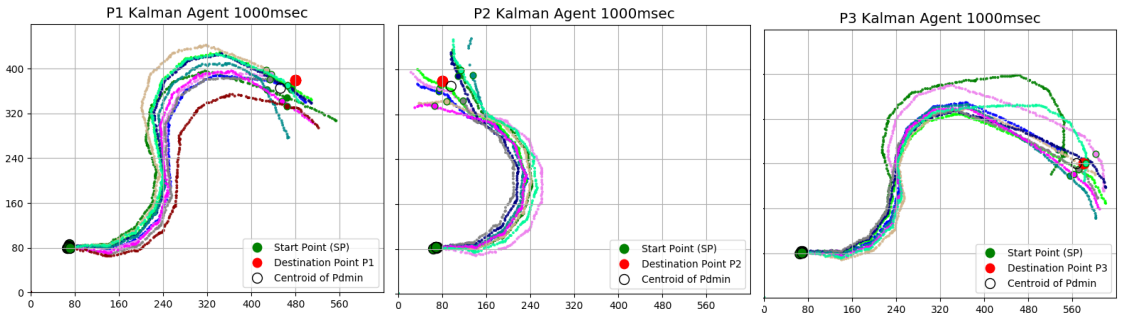


Figure 28: The Kalman agent with T_L =1000msec

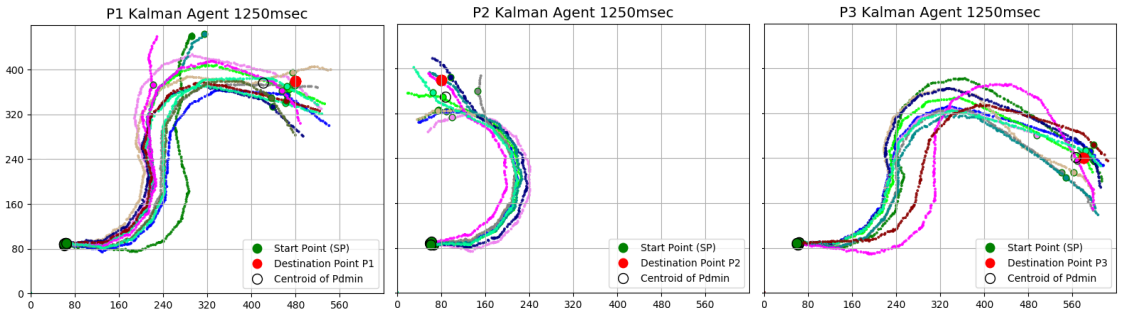


Figure 29: The Kalman agent with T_L =1250msec

Table 9: Measurements with the Kalman agent, TL=250, 500, 750 and 1000 msec

Centroid PN_{ck}	Coordinates of PN_{ck} X,Y (pixels)	Target X,Y (pixels)	Distance from target, d_{min} (pixels)/stdev	$T^2/F/F\text{-}crit$	Average S_{dmin} (pixels)/stdev	Average T_{dmin} (msec)/stdev
$P1_{ck250}$	461,108	P1 (480,100)	20/16.4	15.3/6.9/4.3	656/33	12357/663
$P1_{ck500}$	455,108		26/13.9	21.9/9.8/4.5	666/32	12384/520
$P1_{ck750}$	467,109		16/11.2	27.2/12.6/3.9	698/35	12888/745
$P1_{ck1000}$	454,114		30/16.3	62.3/28.1/4.3	639/33	12779/562
$P2_{ck250}$	108,91	P2 (80,100)	29/10.9	91.9/41.4/4,3	482/11	9449/229
$P2_{ck500}$	94,119		24/5.0	141.9/62.1/4.7	490/11	9725/273
$P2_{ck750}$	90.120		22/12.0	47.1/21.2/4.3	504/28	9860/766
$P2_{ck1000}$	96,110		19/16.0	14.8/6.6/4.5	479/40	10038/840
$P3_{ck250}$	565,520	P3 (580,240)	18/9.3	45.6/20.5/4.3	669/20	13290/981
$P3_{ck500}$	572,245		9/11.8	1.9/0.9/4.1	717/60	13433/1023
$P3_{ck750}$	589,239		9/5.9	12.8/5.8/4.5	813/23	14761/580
$P3_{ck1000}$	567,240		11/16.3	3.1/1.4/4.5	711/49	13968/931

Two slightly modified environment setups using the Kalman agent were evaluated. First the camera was tilted by 45 degrees (Figure 30) and proceeded with P1,P2,P3|SP1, T_L=250msec. Surprisingly, the agent performed well – except for the case of P2 (Figure 31) where the ODS could not always provide a position result. The measurement results are shown in Table 10.

Figure 30: The camera is titled relatively to its axis and moved at the side of the running plane. Right is the real time monitor, the red circle is the target and the green dots are the waypoints.

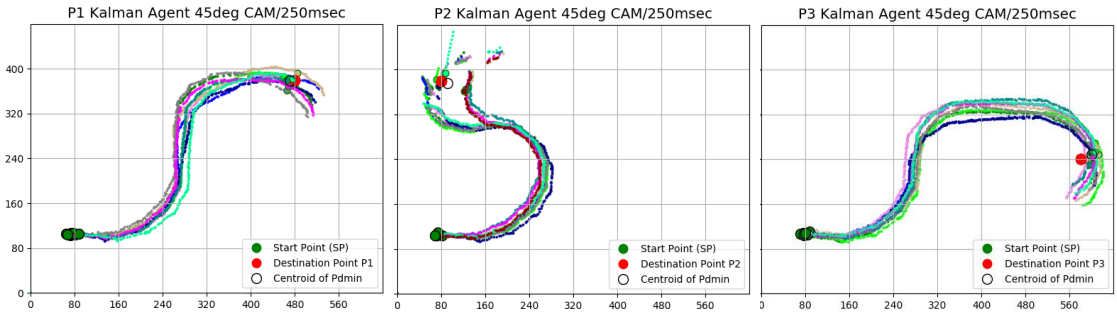


Figure 31: The Kalman agent with T_L =250msec, the camera is tilted 45 degrees.

Table 10 : Measurements using the Kalman agent, T_L =250, the camera is tilted.

Centroid PN_{ck}	Coordinates of PN_{ck} X,Y (pixels)	Target X,Y (pixels)	Distance from target, d_{min} (pixels)/stdev	$T^2/F/F\text{-}crit$	Average S_{dmin} (pixels)/stdev	Average T_{dmin} (msec)/stdev
$P1_{ck250}$	472,102	P1 (480,100)	8.0/7.0	9.9/4.4/4.5	589/11	13299/550
$P2_{ck250}$	93,105	P2 (80,100)	13/16.8	3.0/1.3/4.3	562/64	13552/1588
$P3_{ck250}$	602,232	P3 (580,240)	23/7.8	141.8/63.0/33.9	703/25	15391/801

Moving the camera back to vertical, the next experiment was to randomly reduce the number of available ODS positions by 50%, while the missing 50% was estimated by the KALMANx1 filter prediction. This test was performed with T_L =500 msec, the trajectories are shown in Figure 32 and the measurements in Table 11. A high variance of the trajectory shapes is noticed; since the number of sample available is reduced the estimation of the Kalman filter is less accurate forcing the RL agent into more unseen states.

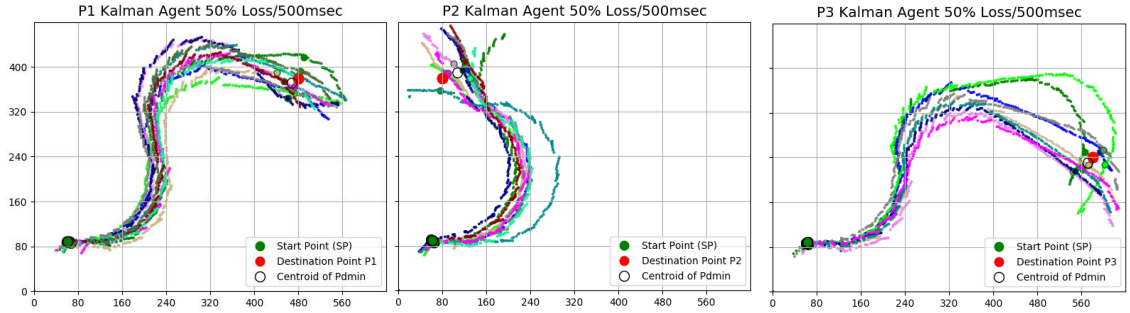


Figure 32: The Kalman agent with T_L=500msec, randomly missing 50% of ODS samples.

Table 11 : Measurements using the Kalman agent with T_L =500msec, there are 50% missing ODS samples.

Centroid PN_{ck}	Coordinates of PN_{ck} X,Y (pixels)	Target X,Y (pixels)	Distance from target, d_{min} (pixels)/stdev	$T^2/F/F\text{-}crit$	Average S_{dmin} (pixels)/stdev	Average T_{dmin} (msec)/stdev
$P1_{ck500}$	469,108	P1 (480,100)	13/13.7	7.6/3.5/4.1	780/162	13205/1101
$P2_{ck500}$	109,91	P2 (80,100)	30/18.6	20.5/9.2/4.3	575/165	9947/938
$P3_{ck500}$	572,249	P3 (580,240)	12/10.0	3.2/1.4/4.7	805/183	13997/1357

5.4. Analysis of the Experiment Results

The results from Table 6 through Table 11 are grouped side by side in Figure 33, Figure 34 and Figure 35. Figure 33 show how the naive agent is affected, when T_L increases the centroids PN_c of the resulted trajectories move away from the target. In every case the variance is high and this is due to the fact that the environment is stochastic and the agent does not take any special measures (e.g. reducing speed or time step) while approaching the target.

- The performance of the Kalman agent is better than the performance of the naive agent for T_L >500msec for every target.
- The Kalman agent for P1 and P2 it is better than the naive agent for T_L = 500msec.
- For T_L = 250msec the agents perform almost the same with exception of the Kalman agent with 45 degrees environment that is slightly worse when targeting P3.
- The Kalman agent with 50% Loss environment and T_L = 500msec is equal or better compared to the naive agent setup. This means that the Kalman agent is suitable not only to counter delays as expressed by T_L but also environments with feedback information that arrives in varying time steps.

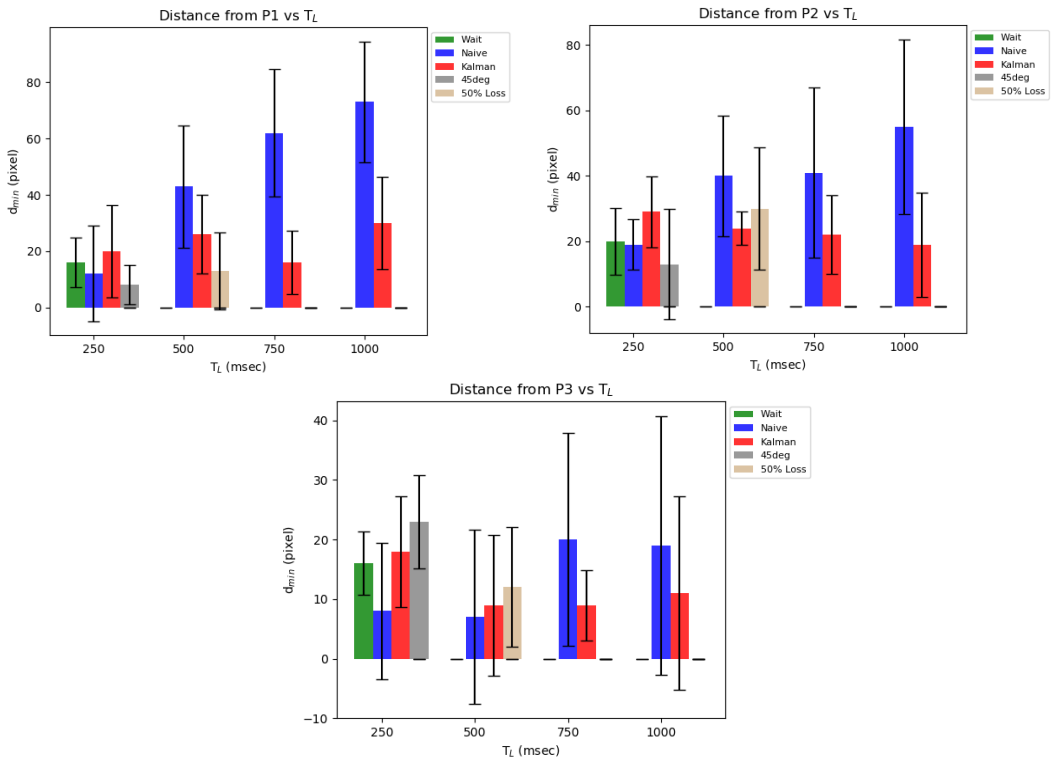


Figure 33: Distance of centroids (PN_c) from their targets for every agent and environment setup.

In Figure 34 it is shown that the average distance ran by the agent is not significantly affected by increasing T_L. The Kalman agent always performs equally well or better than the naive agent.

- Performance on P2 stays almost the same for every T_L, this due to the fact that the path to P2 is a long turn performed almost the same way among agents and environments.
- For the 50% Loss environment, the variance of the distance to P2 is suffering because the KALMANx1 filter works 50% of the time on prediction thus feeding the KALMANxN filter with less accurate information than the no-loss environment.

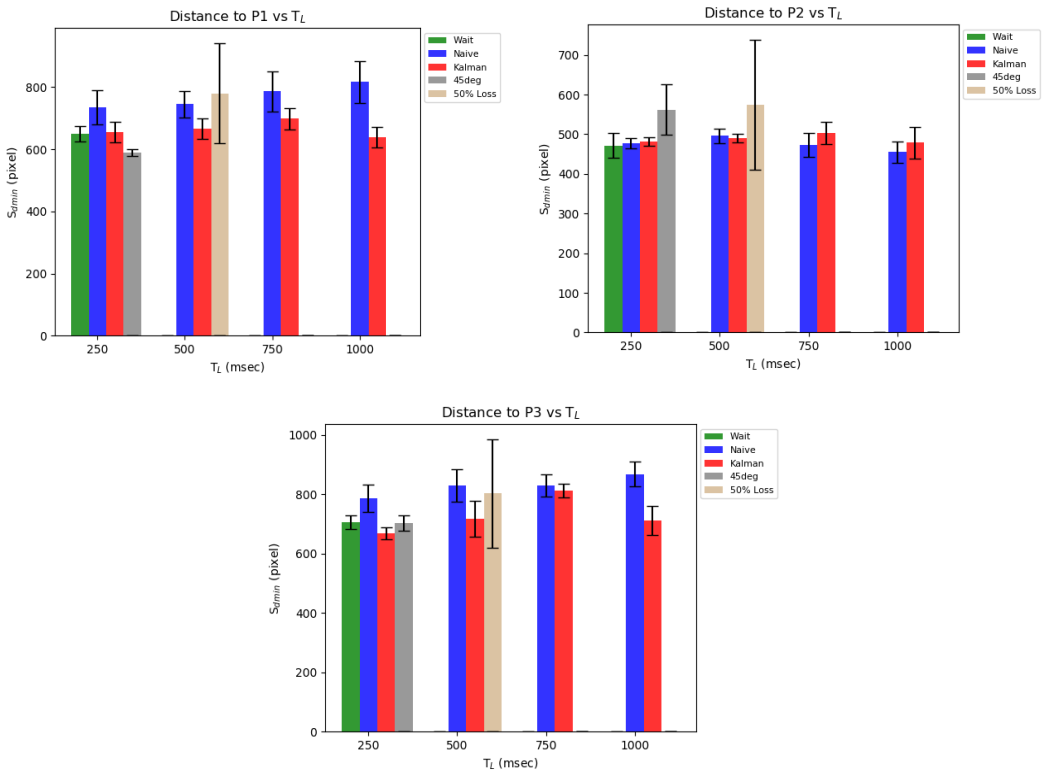


Figure 34: Average distance ran from start point to centroid (PN_c) for every agent and environment setup.

Figure 35 shows the average time the vehicle takes to reach its target (centroid). For P1 and P3, the time to reach the target increases proportionally to T_L.

- The Kalman agent is exceptionally stable for every T_L and much better than the naive agent for P1 and P3.
- For P2 the naive and Kalman agents are a match, since the trajectory to P2 is almost always the same.
- For P2 the Kalman agent for the 45 degrees environment is struggling with ODS detection thus imposing delays.
- As expected, the wait agent has the worst performance for $T_L = 250$msec.

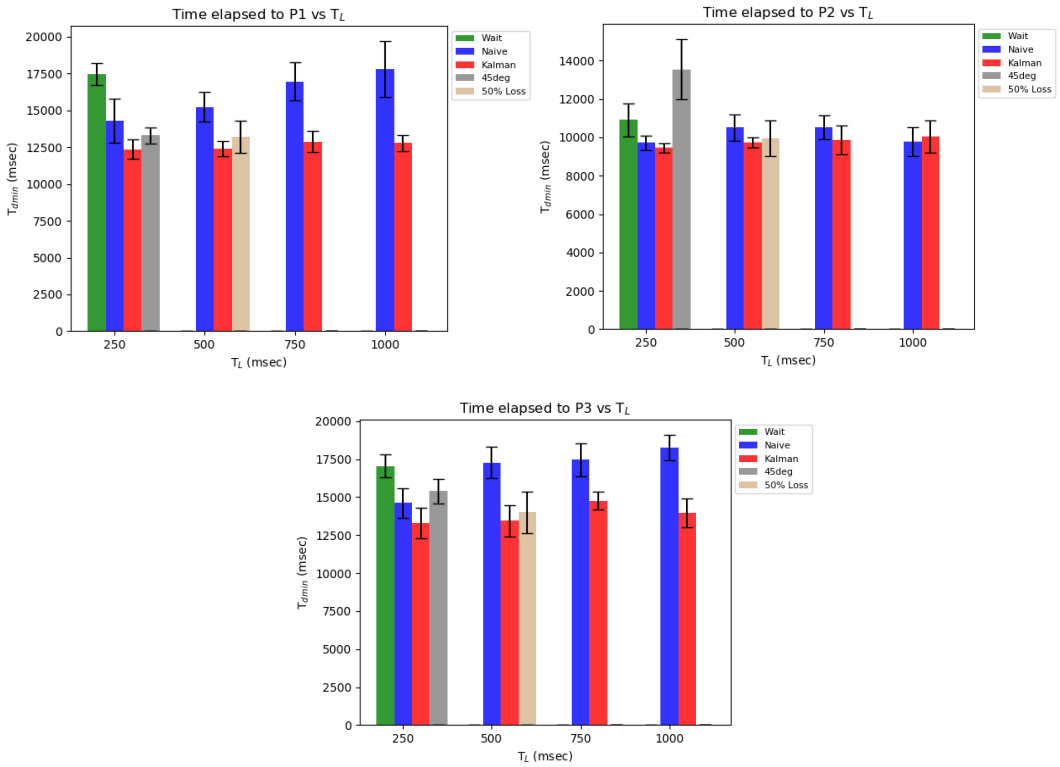


Figure 35: Average time ran from start point to centroid (PN_c) for every agent and environment setup.

CHAPTER 6:
Conclusion

In contrast to the experimental setup of Atari 2600 agents, the raw sensory input of the image is decoupled from the RL agent to make it feasible to deploy a real world application. In the previous chapters have been described the tools, methods and considerations to design the machine vision system and the RL agent to remote control a vehicle under realistic conditions. The experimental setups showed that combining readily available tools and methods, like a linear estimator such as the Kalman filter, one can improve the control under the imposition of a time lag. This is a typical condition in local networks were the control signal can be passed almost instantly while the sensor measurements (like an object detection system) may arrive with a considerable delay. The Kalman filter is applied as an add-on module and it is not integrated with the control agent. Its operation relies only on position measurements thus limiting its dependency from the RL algorithm used.

It is also shown that for this application an RL agent can be designed and that will perform robustly. Our agent performed very well in environments not previously 'seen', like the 45 degrees environment – even when it used starting and destination points other than the ones that it was trained for. Specific measures were taken to assure this performance, e.g., during training Gaussian noise was injected between layers and randomness into the simulated environment. Also, after training, an exhaustive evaluation procedure was performed that has yielded not only the best candidate RL weights for the experiments, but also a guide of what hyperparameters to use in future experiments. The present work included at least 400 logged trajectories that can be used in

future simulations, along with many lines of code to automatically illustrate and analyze the results for better intuition.

The object detection system worked with real time performance and performed well within the assumptions it was trained under. The detection ratio was almost 100% and the frame rate was solid above 15fps. Since the machine vision system was off loaded to an external USB device and the RL agent had 5msec inference time, our setup could run on low cost PCs thus making it feasible for wide deployment. The ODS can be trained to detect targets and obstacles. Preliminary work showed that by using transfer learning and the spared inputs of the state vector S_t the agent can learn to avoid an obstacle that can be engulfed by a rectangle. One can think of many improvements, like using the IoT MSDK on board sensors to perform dead reckoning in shaded areas or using another estimator like an Extended Kalman Filter or an RNN.

References

Agarwal, S., Terrail, J. O. D., & Jurie, F. (2018). Recent Advances in Object Detection in the Age of Deep Convolutional Neural Networks. *ArXiv:1809.03193 [Cs]*.

AlexeyAB. (n.d.). *Yolo_mark*. https://github.com/AlexeyAB/Yolo_mark.

CUDA. (n.d.). https://developer.nvidia.com/cuda-gpus: NVIDIA.

DA14585 Development Kit-Basic. (2017, June 29). [Text].

DA14585 IoT Multi Sensor Development Kit. (2018, June 25). [Text].

Dai, J., Li, Y., He, K., & Sun, J. (2016). R-FCN: Object Detection via Region-based Fully Convolutional Networks. *ArXiv:1605.06409 [Cs]*.

Dalal, N., & Triggs, B. (2005). Histograms of Oriented Gradients for Human Detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)* (Vol. 1, pp. 886–893). San Diego, CA, USA: IEEE.

Darknet: Open Source Neural Networks in C. (n.d.).

Dimitri P. Bertsekas. (2012). *Dynamic Programming and Optimal Control* (Vol. 2).

Dimitri P. Bertsekas. (2017). *Dynamic Programming and Optimal Control* (Vol. 1).

Firoiu, V., Ju, T., & Tenenbaum, J. (2018). At Human Speed: Deep Reinforcement Learning with Action Delay. *ArXiv:1810.07286 [Cs]*.

Girshick, R. (2015). Fast R-CNN. *ArXiv:1504.08083 [Cs]*.

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2013). Rich feature hierarchies for accurate object detection and semantic segmentation. *ArXiv:1311.2524 [Cs]*.

Gym. (n.d.). Available from : https://gym.openai.com/.

Jetson Nano DK. (n.d.). Availlable from : https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

Kalman. (1960). A New Approach to Linear Filtering and Prediction Problems. *Transactions of the ASME–Journal of Basic Engineering,*.

Kapica, P. (n.d.). *convert_weights*. https://github.com/mystic123/tensorflow-yolo-v3.

Katsikopoulos, K. V., & Engelbrecht, S. E. (2003). Markov decision processes with delays and asynchronous cost collection. *IEEE Transactions on Automatic Control*, *48*(4), 568–574.

Keras. (n.d.). Available from : https://keras.io/.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90.

Kuznetsova, A., Rom, H., Alldrin, N., Uijlings, J., Krasin, I., Pont-Tuset, J., … Ferrari, V. (2018). The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale. *ArXiv:1811.00982 [Cs]*.

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., & Jackel, L. D. (1990). Handwritten Digit Recognition with a Back-Propagation Network. In D. S. Touretzky (Ed.), *Advances in Neural Information Processing Systems 2* (pp. 396–404). Morgan-Kaufmann.

Li, Y. (2018). Deep Reinforcement Learning: An Overview. *ArXiv:1701.07274 [Cs]*.

Lin, T.-Y., Dollar, P., Girshick, R., He, K., Hariharan, B., & Belongie, S. (2017). Feature Pyramid Networks for Object Detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 936–944). Honolulu, HI: IEEE.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal Loss for Dense Object Detection. *ArXiv:1708.02002 [Cs]*.

Lin, T.-Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., … Dollár, P. (2014). Microsoft COCO: Common Objects in Context. *ArXiv:1405.0312 [Cs]*.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). SSD: Single Shot MultiBox Detector. *ArXiv:1512.02325 [Cs]*, *9905*, 21–37.

Lowe, D. G. (2004). Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, *60*(2), 91–110.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. *ArXiv:1312.5602 [Cs]*.

Nardo, E. D., Petrosino, A., & Santopietro, V. (2018). Embedded Deep Learning for Face Detection and Emotion Recognition with Intel© Movidius (TM) Neural Compute Stick.

Neural Compute Stick 2. (n.d.). https://software.intel.com/en-us/neural-compute-stick: Intel.

OpenCV. (n.d.). https://opencv.org/.

OpenVINO. (n.d.). https://software.intel.com/en-us/openvino-toolkit: Intel.

Pena, D., Forembski, A., Xu, X., & Moloney, D. (2017). Benchmarking of CNNs for Low-Cost, Low-Power Robotics Applications, 5.

Redmon. (n.d.). *YOLOv3 Tiny*. https://pjreddie.com/darknet/yolo/.

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2015). You Only Look Once: Unified, Real-Time Object Detection. *ArXiv:1506.02640 [Cs]*.

Redmon, J., & Farhadi, A. (2016). YOLO9000: Better, Faster, Stronger. *ArXiv:1612.08242 [Cs]*.

Redmon, J., & Farhadi, A. (2018). YOLOv3: An Incremental Improvement. *ArXiv:1804.02767 [Cs]*.

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 28* (pp. 91–99). Curran Associates, Inc.

Richard Bellman. (1957). Dynamic Programming. *Princeton University Press*.

Ruder, S. (2017). An overview of gradient descent optimization algorithms. *ArXiv:1609.04747 [Cs]*.

Sang-Hun, C. (2016, March 15). Google's Computer Program Beats Lee Se-dol in Go Tournament. *The New York Times*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., … Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.

Sutton, R. S., McAllester, D. A., Singh, S. P., & Mansour, Y. (2000). Policy Gradient Methods for Reinforcement Learning with Function Approximation. In S. A. Solla, T. K. Leen, & K. Müller (Eds.), *Advances in Neural Information Processing Systems 12* (pp. 1057–1063). MIT Press.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., … Rabinovich, A. (2015). Going Deeper With Convolutions (pp. 1–9). Presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.

Walsh, T. J., Nouri, A., Li, L., & Littman, M. L. (2008). Learning and planning in environments with delayed feedback. *Autonomous Agents and Multi-Agent Systems*, *18*(1), 83.

Watkins, C. J. C. H. (1989). Learning From Delayed Rewards.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3), 279–292.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*(3), 229–256.

ZeroMQ. (n.d.). Available from : https://zeromq.org/.