From simulation to reality

Study and demonstration of domain adaptation techniques applied to Reinforcement and Supervised learning algorithms.

Auteur:
Pablo Miralles
Collaborateur:
Vincent Coyette
Tuteur:
David Bertoin

November 19, 2020

Contents

Executive Summary **4**

1 Introduction **6**

2 The Duckietown Project **8**

2.1 Duckiebot . . . 8

2.1.1 Hardware . . . 8

2.1.2 Software . . . 9

2.1.3 Installation and calibration . . . 10

2.2 Duckietown . . . 12

2.3 Gym-Duckietown . . . 12

2.4 Extending the simulation . . . 14

2.5 The Duckietown community . . . 15

3 Artificial Intelligence and Machine Learning **18**

3.1 Artificial Neural Networks . . . 18

3.2 Supervised Learning . . . 20

3.3 Reinforcement Learning . . . 21

3.3.1 Markov Decision Processes (MDPs) . . . 21

3.3.2 Continuous Partially Observable MDPs . . . 23

3.3.3 Returns, Policies, and Value Functions . . . 23

3.3.4 The Bellman Equation . . . 24

3.3.5 The L in RL - Learning . . . 24

3.4 Domain Adaptation . . . 27

3.4.1 State of the Art 28

3.4.2 Tested techniques 30

3.4.3 Domain Adaptation in Duckietown 40

3.5 Imitation learning 41

3.6 Neural Architectures in Duckietown 41

4 Experimental setting **44**

4.1 Measuring performance 44

4.2 Test environments 45

4.3 Algorithms 47

5 Results **49**

5.1 Simulation Results 49

5.2 Reality Results 49

5.3 Exploration of results 51

5.3.1 Are the models too small? 51

5.3.2 Data augmentation study 51

5.3.3 Performance of other DAEL variants 52

5.3.4 Does DAEL work? 52

5.3.5 Adapting DADAPT-IRT 55

5.3.6 Does DWT work? 55

6 Conclusions **58**

6.1 Direct outputs 58

6.2 Research conclusions 58

6.3 Future work 59

6.3.1 Improved Sim2Real 59

6.3.2 Further Domain Adaptation research 60

Executive Summary

This report describes my 5.5 months end of studies internship as an AI Research Intern, the focus of which was the implementation and demonstration of techniques from the Domain Adaptation literature on the problem of autonomous driving. Machine Learning algorithms are famously brittle to shifts in the data distribution, and Domain Adaptation is a sub-field of Machine Learning aiming to palliate this issue. We replicate and implement five Domain Adaptation techniques to the task of autonomous driving in simulated and real environments on a representative model which we assemble, install, and debug: the Duckietown simulator and robot created and distributed by the eponymous foundation. We create a simple containerised solution for robot control cleaner than the original one. We also extend the simulator with a plethora of additional methods to train and benchmark Supervised and Reinforcement Learning methods in default and altered environments

The report is divided into the present summary and 6 chapters. Chapter 1 provides the project's context. Chapter 2 introduces Duckietown: the robot, the circuit it drives in, the simulator where our AI agents are trained, and the community built around this ecosystem. Chapter 3 presents the theoretical foundations of the Machine Learning algorithms used in the project including Domain Adaptation: what it is, why it is needed, what techniques exist, and what algorithms we have used. Chapter 4 presents the conditions under which we carried out our experiments: how results are evaluated, what architecture we have used, and what we tested on. Chapter 5 presents the results of our research and explores the performance of our different techniques under a variety of scenarios. Lastly, Chapter 6 draws conclusions from the project and reflects on potential future work.

Regrettably, we did not manage to obtain good results in passage from simulation to reality: None of our methods managed to consistently complete a whole lap on the real Duckietown. We suspect a few key causes for this result: A gap between simulation and reality that was too wide; an over-reliance of our methods on adaptation of observations which neglects gaps in dynamics; and a lack of generality within the Domain Adaptation literature. We also believe several avenues for attempting to improve performance remain: Bridging the dynamics gap with robust control techniques, using extreme augmentation techniques like others have done with similar problems, and simplifying the problem by renouncing to end-to-end control.

In the face of a lack of results in the simulation to reality task, we switch to an intermediate task of moderate difficulty to benchmark our approaches: Using Domain Adaptation to obtain satisfactory performance across several different simulated circuits of mounting difficulty. Firstly, domain adaptation techniques are remarkably brittle. While they provide significant improvements in the reference benchmarks used for their publication, they fail to provide any improvements in this specific problem, and in fact their underlying mechanisms are utterly broken in our problem. Secondly, data augmentation, a widely used technique to achieve somewhat increased generality renowned for its wide applicability, was only helpful in specific circumstances, and was a hindrance in some others. Nevertheless it was the only method providing any gains.

Overall, it would seem the current status of the Domain Adaptation field does not allow for easy application of developed techniques across domains. One should exercise caution in assuming that good results in a given data-set will apply to another task, rendering the field's findings less useful and of limited validity.

Chapter 1

Introduction

The design of an autonomous driving agent that can traverse any road effectively and safely, at scale, is one of the most promising applications of modern Artificial Intelligence (AI) and Machine Learning (ML). This is a hard problem due to a variety of reasons. Firstly, driving conditions are hugely variable and dependent on location, climate, and time. Driving in Berlin in a heavy winter snow-storm is not the same as driving in Berlin during a sunny summer day, and it is massively different to driving in the Australian outback in autumn. Secondly, consequences of failure are dire, with potential losses of expensive equipment or even loss of life on the line. Lastly, and much due to the same reasons, gathering data is expensive - Large amounts of it are needed from very different places and they all require expensive equipment and labour.

To remedy these issues some have proposed to gather data in simulated environments, where failure is cheap and execution can be accelerated by using more computational resources. However, the leading solutions in autonomous driving, namely Machine Learning algorithms partly or entirely based on Neural Networks, are notoriously brittle to even the smallest shifts between the data distribution seen during training of the agent and the distribution seen in operation. These agents achieve good simulation performance, but that performance degrades dramatically when the agents are deployed in real scenarios. Designing a simulated environment that does not trigger said brittleness is exceedingly difficult, requiring subject matter expertise, extensive calibration and verification to ensure similar behaviour, and systematic re-calibration of models to ensure the characteristics of the real environment do not drift far apart from those of the simulated environment.

Domain Adaptation is a sub-field of Machine Learning that could be applied to mitigate this issue. In Domain Adaptation, we use data from a set of source domains to train an agent that must then be performant in another set of domains, dubbed the target domains, for which data exists but is not labelled. By taking the source domains to be our simulations, and the target domains to be the real environments in which the vehicle will drive, we could apply these techniques and allow our simulation to represent reality less accurately, driving down costs and enabling access to larger data-sets. Domain Adaptation is a field in development, and while a full explanation and mitigation of ML brittleness still eludes the research community, many techniques have been published that combat or even eliminate it in specific cases.

We implement four of these techniques to the task of autonomous driving in simulated and real environments in a representative model of real autonomous driving. Throughout this project we use the Duckietown simulator and robot created and distributed by the eponymous foundation. The simulation provides a comfortable environment in which to train and test our agents while the robot provides a platform to test our passage to reality. The Duckietown robot consists of a camera and two motors actuating two wheels, plus a microprocessor and associated software controlling them. We initially focus on the application of Domain Adaptation to Reinforcement Learning agents and later shift that focus to Supervised Learning.

These activities were carried out in fulfillment of my end of studies internship as an AI Research Intern at the Institut de Recherche Technologique Saint Éxupery in Toulouse, in partnership with the SuReLI research team from ISAE-SUPAERO headed by Emmanuel Rachelson. The internship had a duration of 5 and a half months, was carried out within a team of three, and it is a continuation of my Projet d'Ingénierie et Entreprise (PIE) 074: "Sim2Real". During the PIE we assembled the robot (but could not get it running) and created a basic self-driving agent based on Reinforcement Learning. My colleague Vincent and I worked closely in almost all of the topics covered in the internship, and therefore a full presentation of the topic requires presenting some of his work in addition to mine. To ease the evaluation procedure, I've noted whenever a task was performed by him. Whenever such a remark is absent, the work was performed by me. The third member of our team was our internship supervisor, the PhD. student M. David Bertoin, who had a supervisory role and did not work directly in the project's implementation.

Chapter 2

The Duckietown Project

Duckietown [1] is a project maintaining and distributing a variety of platforms and tools for robotics applied to autonomous driving. The project was conceived in 2016 by MIT and is currently maintained by the non-profit Duckietown Foundation in collaboration with volunteers from Université de Montreal and ETH Zurich. It has a dual purpose. On the one hand, it aims to be an educational tool that makes it easier to get hands-on experience with robotic applications. On the other hand, it also aims to be a common benchmark for algorithms in autonomous driving.

We use three tools from the Duckietown project: the Duckietown, a modular system to create driveable circuits from plastic tiles and tape, the Duckiebot, a minimalist robot with a camera and two wheels acting as a self-driving agent, and the Gym-Duckietown, a Python-based simulation of the Duckiebot and Duckietown ensemble in which self-driving agents can be trained.

Duckietown is far from being a realistic replica of a real driving scene, but the platform contains many of the elements relevant to a driving simulation - A variety of scenarios, static and dynamic obstacles, and some relevant examples of traffic signalling from the real world.

2.1 Duckiebot

2.1.1 Hardware

The robot, or Duckiebot, is the main physical platform for self-driving agents in Duckietown. It is a simple machine meant to be easy to use, hosting two small DC motors each actuating on one of the two wheels, and a camera that acts as the robot's sole sensor. The robot also uses a rear non-actuated omni-directional wheel to act as a support point for

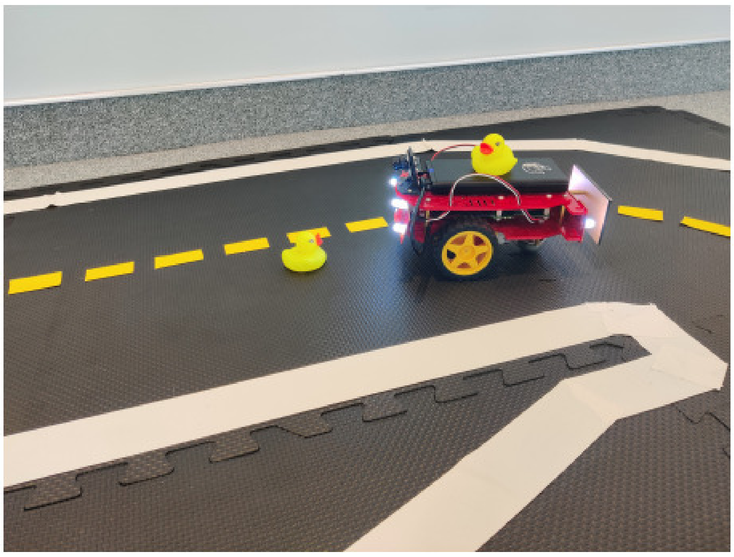


Figure 2.1: Duckiebot used in this project

the robot. Both actuators and the camera are controlled via a Raspberry Pi 3b+ micro-controller, which can host the driving agent or relay its instructions from some external computer. The robot also includes 5 LED's that signal the robot's status. All of these components are held in place via a red plastic chassis, and metal and plastic nuts and bolts. Power is provided by a 20 Ah Li-ion battery enabling several hours of autonomy. Lastly, the Duckiebot also includes a motor driver HAT, a small micro-controller that interfaces the low-level motor command signals (driven by pulse width modulation signals) to the Raspberry Pi. This greatly simplifies writing software that commands the motors.

The components can be bought from the Duckietown Foundation or acquired separately from third parties. In either case, assembly of the elements into a functioning Duckiebot is performed by the user. The assembly process is easy and requires only a screwdriver and no previous assembly experience, and takes about 3h for an inexperienced user. Our assembled duckiebot is shown in figure 2.1. A drone-like flying variant of the Duckiebot exists, but we have not used that variant during this project.

2.1.2 Software

The software running on the robot's micro-controller is structured around 3 major components: the Robot Operating System (ROS), Docker, and the Duckietown-Shell. To give an extensive description of each of these tools is beyond the scope of this report, but I will provide a quick overview.

ROS is an open-source software built to manage distributed applications in robotic environments. It provides several core facilities: Message passing, message recording and playback, remote procedure calls, and distributed configuration management. It also provides additional robot-specific features [2]. In ROS each process constitutes a node. Nodes publish topics, which are sporadic data flows and to which other nodes subscribe. Messages are passed between nodes according to each topic's subscriptions and publications. Nodes also offer services, which are functions that can be called remotely by other nodes. Nodes can be distributed across one or multiple processing systems, which in our implementation are two: The Raspberry Pi and an external computer. The network is coordinated by one node called the ROS Master, which handles naming and registration of nodes and services, and more.

Docker [3] is a very popular software for containerization of applications. A Docker container encapsulates an application and its environment, including its operating system (OS), in isolation from the environment and OS that run Docker. The encapsulation provides more control of the application's environment to its designer, providing reliability. A Docker container is built from a Docker image, which is a file describing the application and the environment it runs on. Docker images are conceived as a layered system, where each image expands the functionality of another image it builds on. At the very core lies the image containing the operating system for the Docker container. This system makes images easier to use. Docker also provides a platform to share and publish images and an application to easily download and manage them.

The Duckietown shell does not run on the robot but instead is meant to ease interaction with it. It is a Python-based tool that provides a shell-like environment with a set of common commands to run on it. The main commands it provides are the flashing and booting of an SD Card with the Duckiebot software stack, the calibration of the robot's wheels and camera, and some testing facilities.

All Duckiebot software runs on top of the HypriotOS operating system, which is *"a minimal Debian-based operating system that is optimized to run Docker"* [4] according to its creators. The core reason for choosing HypriotOS over other OS's is indeed its close integration with Docker, since the latter is a core component of the Duckiebot.

2.1.3 Installation and calibration

Installing the Duckiebot software stack on the Raspberry Pi is meant to be an easy, streamlined process with minimal user interaction. It is not.

Due to the somewhat complex and mostly untested suite of softwares that interrelate with each other, and the minimal user-base to test the software with, bugs are frequent and resources to solve them are scarce. This procedure is made more complex by the frequent updates that the Duckietown foundation pushes on the software, with new bugs appearing due to no fault of the user. Having a Duckiebot where we could load a self-driving agent, a prerequisite for the rest of our project, took 2.5 months of nearly full-time dedication and a switch away from the Duckietown Foundation's solution. Amongst the issues we found were: A defective Raspberry Pi card that froze under unknown circumstances (we later learned that this is a recurrent issue for Duckietown with no known cause), particularly but not limited to Docker use, and which we had to replace; a camera calibration procedure that broke after a faulty update from the Duckietown foundation; a botched python2 to python3 migration (started by the Duckietown Foundation) that left the robot inoperative for 3 weeks; and faulty Docker containers that would fail silently then proceed to perturb each other's functioning through the ROS network.

The difficulty in dealing with the combination of these issues, and the rate at which new ones kept appearing, pushed us to adopt an alternative software solution. This solution was based off the Duckietown Foundation's stack, but written by us and therefore more controllable. The self-built solution was designed to comply with the following requirements:

1. Capture images from the PiCamera
2. Send said images to the controller in under 100ms
3. Transmit actions (motor actuaction signals) from the controller to the motors
4. Present an interface to the user for diagnosis and visualization of the camera observation and the action taken
5. Provide a means for the user to input manual control signals through the keyboard, acting as a controller node
6. Seamlessly switch between the manual and automated modes of control
7. Provide means to create a dataset replicating manual or automated control

To fulfill these requirements my colleague Vincent implemented a simple ROS solution, packaged with Docker, consisting of 5 nodes: One for the Camera, one for each motor, one for the controller, and one for the Writer. The controller is hosted on an external computer and transmits and receives data over a shared WiFi network. The visualization and control tool is written in Python with the PyGame library, and allows the user to control the Duckiebot with the arrow keys and switch between automated or manual driving by pressing the spacebar. Figure 2.2 shows a schematic of the architecture, with ROS nodes highlighted in red while other application components are highlighted in red and other entities are encircled in black. The Writer is not shown in this diagram, its function is to fulfill requirement 7 by storing a number of image-action pairs. I later extended this tool to comply with the same interface that we used within the simulator tool, in order to ease execution and maintenance of the tool.

Figure 2.3 shows a snapshot of the visualization and control tool. The tool shows a live image of the robot's camera for the user, along with a black and white version of it and the 3 last received images. The four black and white images constitute the input to our self-driving agents.

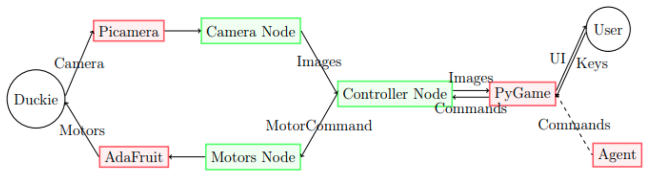


Figure 2.2: Structure of the ROS Solution

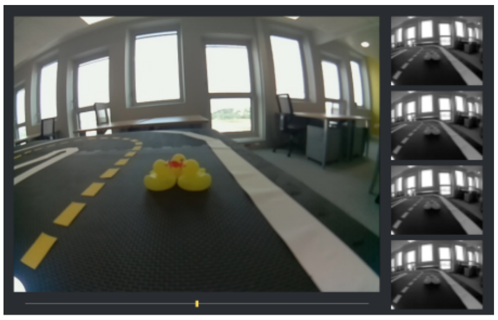


Figure 2.3: Visualization and control tool.

2.2 Duckietown

The town, or Duckietown, is the physical environment in which the robot acts. It consists of a set of black tiles simulating an asphalt surface, upon which we lay colored tape stripes to replicate road lane signalling. The tiles have a corrugated texture to provide additional traction to the Duckiebot's wheels, and their edges follow a puzzle-like jagged shape to ease their assembly into a cohesive town. As with the robot, the Duckietown Foundation sells and distributes internationally these tiles and colored tapes.

Given that the town is made by independent tiles designed by the user, potentially any circuit can be replicated with a sufficient number of tiles. Due to budgetary constraints we acquired 9 such tiles which we arrayed into a 3x3 square building the circuit seen in figure 2.4 [1].

We did not use any traffic signalling, static or dynamic obstacles, intersections, or stop sections for this project. While we possessed the material to do so, mastering this simple sim-to-real task proved to be an already daunting challenge.

2.3 Gym-Duckietown

Gym-Duckietown ('the simulator' from here on after) is a computer simulation of the Duckietown environment, based around timesteps - discrete identically sized increases of time that push the simulation forward. It is based on OpenAI's popular framework for struc-

[1] A certain number of specifications are to be followed in order to build a "Duckietown-compliant" town, some of which constrain the possible arrangements of tiles. One such constraint would have severely limited us: All adjacent road sections must be interconnected. The central, center right, and center up tiles in our design violate that constraint. We found the constraints do not impact the agents unless Reinforcement learning algorithms are trained on circuits violating this constraint. By avoiding to do so we enabled our use of the circuit shown in figure 2.4. Furthermore, some of the maps included by default in Gym-Duckietown also violate the constraint.

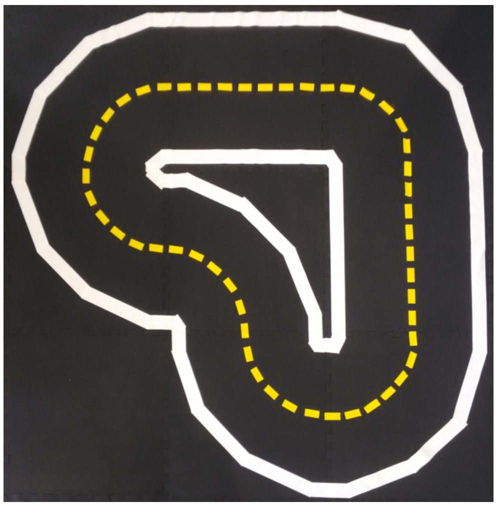


Figure 2.4: Duckietown used in this project

turing reinforcement learning environments OpenAI-Gym, which provides a standardized set of interfaces for a simulated agent to interact with the environment and for users to modify the environment.

The simulator's main function is to provide an environment for the self-driving agents to interact with. It provides a graphical simulation of the environment to feed a virtual camera like the one in the physical duckiebot, and an elementary simulation of the environment's kinematics (it does not, however, model inertia). Additionally it provides a reward signal (that can be readily altered) to train agents which need one such as Reinforcement Learning agents.

To improve versatility, the environment is built from a set of reusable tiles, much like the physical Duckietown. These tiles are split in two main categories, driveable (like a straight patch of road) and non-driveable (like an area of grass). The configuration of these tiles can be modified at will via a map file, a .yaml format file describing the arrangement of tiles in a matrix-like fashion. The actual appearance of the tiles is extracted from a texture file and laid out over this conceptual tile, enabling an easy modification of the environment's appearance. [2]

A set of objects is then created to exist on this environment. The possible objects include other vehicles (represented as Duckiebots), passersby (represented as rubber ducks), traffic signalling such as Stop signs or traffic lights, or relevant decoration such as buildings.

[2]The default tiles did not respect the Duckietown Specification. We originally replaced them with our own, only to find out that the simulator mechanics are incompatible with the Duckietown Specification. We thus reverted back to using the original tiles, and all results reported are evaluated against this set of tiles. The default tiles have changed since the creation and bench-marking of our agents. The tiles we used were the default ones as of the 1st of August, 2020

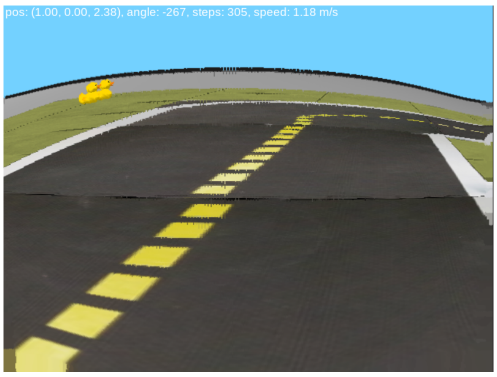

pos: (1.00, 0.00, 2.38), angle: -267, steps: 305, speed: 1.18 m/s

Figure 2.5: Render from an agent's point of view

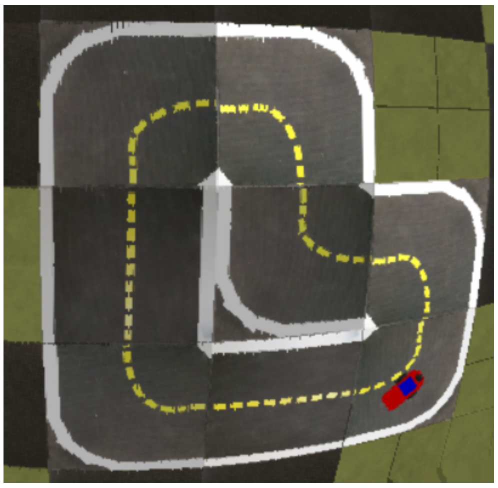


Figure 2.6: Render from a top-down perspective.

One key function of a Gym environment is that of episode management. Upon calculating a new time-step, the environment detects whether a set of episode-ending conditions have been met, and if so, it is reset to a new starting condition. The set of time-steps between simulator re-initialisations constitutes an episode. In Gym-Duckietown an episode ends whenever the agent collides with another object or enters a non-driveable tile[3].

Another function lets the user create a render of the environment from several points of view, most notably from an object's point of view (Fig. 2.5) or from a top-down perspective (Fig. 2.6). This last perspective is particularly useful for diagnosis and debugging.

2.4 Extending the simulation

The Gym-Duckietown environment, like any Gym environment, offers a way to customize its behaviour - environment wrappers. A wrapper is a programming object that extends or wraps another object by overwriting some or all of its functions. In this way we can define a reward wrapper that leaves the environment untouched except for the function calculating the reward obtained by an agent.

To perturb the agent's interactions with the simulation in ways that mimic reality, we resorted to Action and Observation wrappers. Action wrappers take the actions emitted by the agent and modify them in ways that replicate some of the actual perturbations introduced by the real environment. We have included 4 such wrappers:

[3]This is assumed by the creators to coincide with driving outside the white stripes. As mentioned earlier, this is incompatible with the Duckietown specification

1. RandomDelays and RandomFreezes simulate delays in the transmission of messages from the microprocessor and the wheels, and freezes of the microprocessor for several steps due to peaking workloads. Both delays and freezes follow a gaussian with only positive values.
2. PerturbAction simulates errors between the transmitted PWM signal and the applied PWM signal according to a Gaussian law.
3. PerturbWheelDistance simulates inter-wheel distances varying from the nominal values, which causes the agent to turn more or less sharply for a given input.

Observation wrappers instead take the image renderings obtained by the agent and alter them to reflect some of the ways in which reality deviates from the simulator rendering, as seen in Figure 2.7. They are all based on compositions of transformations from the albumentations image transformation library [5], and we have attempted to make them somewhat orthogonal to each other.

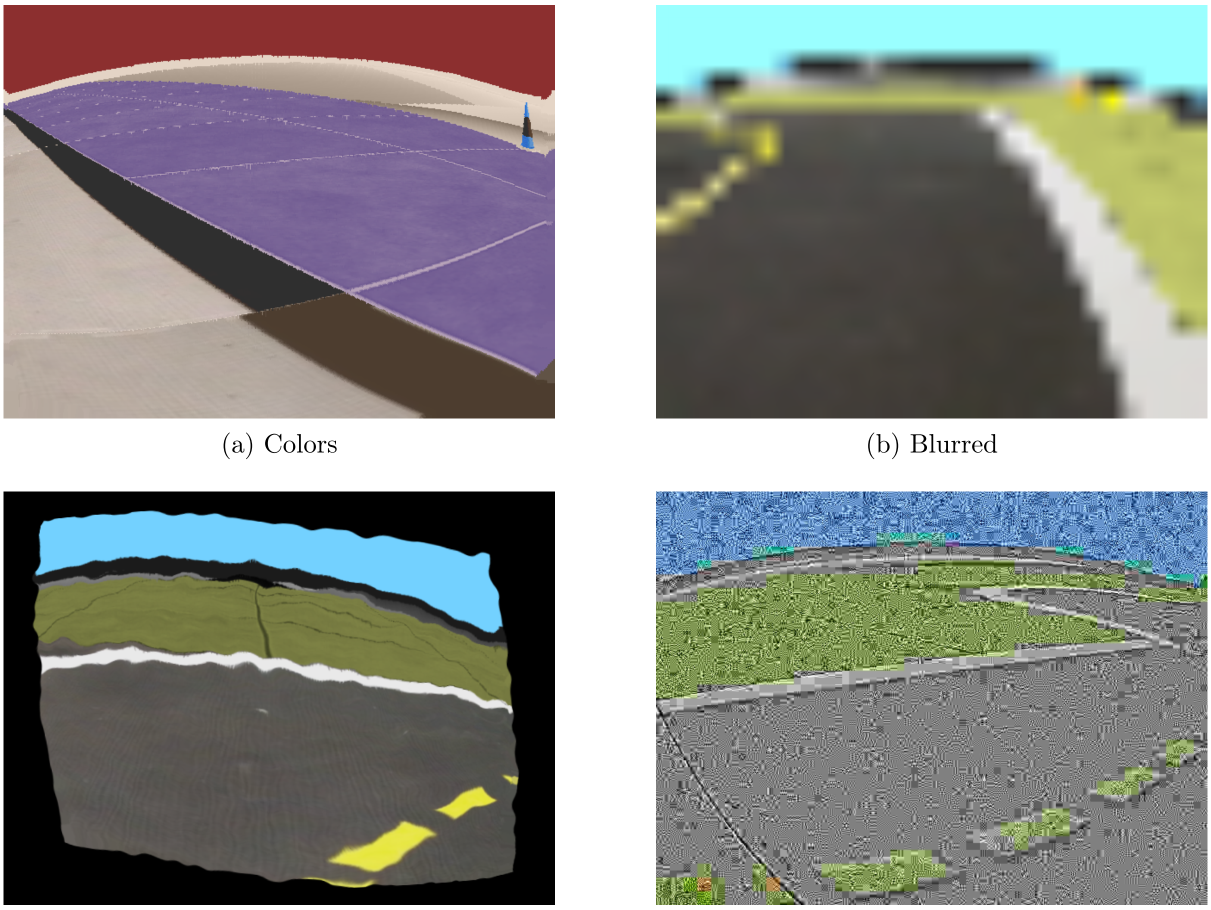
- Colors (Fig 2.7a) focuses on changing the colors of objects and their relationships. It is based on a random change to each of the RGB values of a pixel, an inversion of colors for all pixel values above a random threshold (solarization), and a random change in brightness and contrast
- Blurred (Fig 2.7b) applies blurs and general noise to the image, based on a Blur transformation followed by multiplicative noise and a downscaling and upscaling operation.
- Shapes (Fig 2.7c) alters the shapes of objects in the image while leaving other properties unchanged. It consists of an Elastic Transformation that wobbles shapes followed by an Optical Distortion simulating a convex/concave lens effect.
- Textures (Fig 2.7d): Textures in Duckietown are flat: The sky is the exact same tonality of blue at all points, and the white lane is the exact same white through a tile. This transformation attempts to provide texturization by introducing local variations through an ISO noising operation simulating camera sensor noise, a CLAHE operation, and the effects of an Image compression with large information loss.

2.5 The Duckietown community

On top of maintaining these tools, the Duckietown Foundation and its associated volunteers also provide support for and maintain a community of their users. This community interacts mainly in three ways:

1. A Slack (instant messaging) channel
2. The AI Driving Olympics (AI-DO) sponsored by the Duckietown Foundation
3. A Github project page

The Slack can be readily joined by anyone and provides useful technical support when



(a) Colors (b) Blurred (c) Shapes (d) Textures

Figure 2.7: Transformed simulator

using Duckietown's tools.

The Github page contains every digital asset related to Duckietown, including the documentation and code-base for the robot's software and for Gym-Duckietown. It also constitutes the main way of providing user feedback to the Duckietown Foundation.

AI-DO is a competition for autonomous driving agents within Duckietown environments. It is held semi-annually at several AI conferences, such as Neural Information Processing Systems (NeurIPS) or the International Conference on Robotics and Automation (ICRA) and recently went through its third edition. While it was our original intention to participate in this competition, it was cancelled due to the ongoing global pandemic.

Chapter 3

Artificial Intelligence and Machine Learning

Artificial (Digital) Intelligence (AI) is the field that studies the design of intelligent (artificial) agents [6]. Much of AI revolves around Machine Learning (ML), the study of algorithms that improve with experience [7]. During this internship I worked with two large branches of Machine Learning , Supervised and Reinforcement learning. Although I will provide a brief introduction to each, the reader may refer to authoritative sources on the matter for a more in-depth discussion.

Note that Machine Learning covers many techniques outside the ones listed here, such as neighbourhood techniques, decision trees, or Bayesian modelling. The choice of techniques used reflects purely a choice in focus and not necessarily a belief about their expected performance.

3.1 Artificial Neural Networks

A Neural Network is a statistical model built from the composition of layers of simple additions and multiplications, with non-linear activation functions. The nomenclature derives from the field of neuroscience, as Artificial Neural Networks are loosely inspired on the biological network structures formed by neurons in the brains of humans and other animals.

Each neuron in a neural network is fed data-points x, inputs consisting of various features x_j. For example a data-point may be an image, where each feature is the intensity value of each pixel, or the characteristics of the person being modelled such as their age, height, sex, and so on. Each of this input feature is multiplied with a weighting coefficient or parameter, and the results are summed together then fed to the non-linear activation

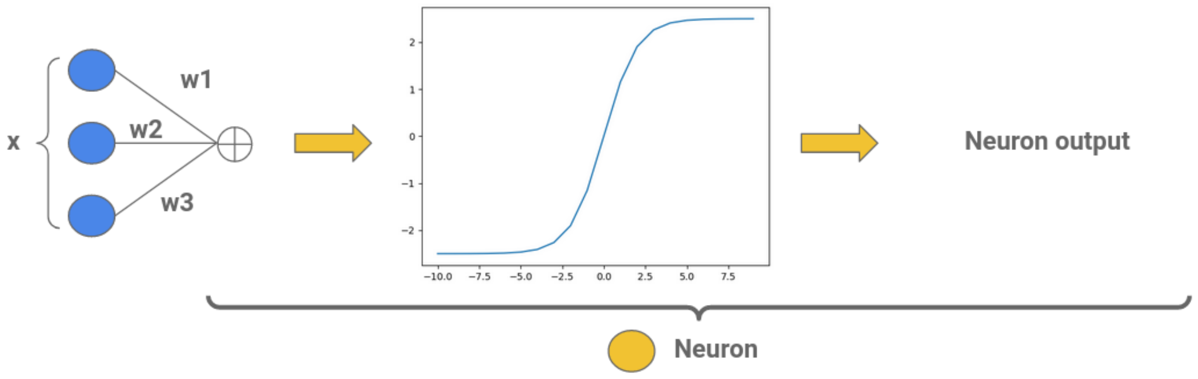


Figure 3.1: Schematic of a neuron

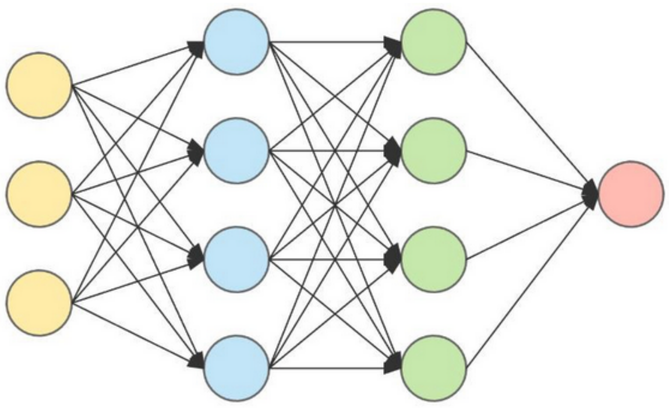


Figure 3.2: Schematic of an Artificial Neural Network composed of fully connected linear layers

function. (Fig 3.1)

The network is formed by many layers of neurons (Fig 3.2). The first layer takes as input the data-point x, and each subsequent layer takes the outputs of the previous layer as inputs. Each unit j in each layer i transforms its inputs $z_{i-1,j'}$ into an output $z_{i,j}$ by applying the operation

$$z_{i,j} = \sigma(\sum_{j'} \theta_{i,j,j'} \cdot z_{i-1,j'})$$

where $\theta_{i,j,j'}$ is a weighting coefficient of unit j in layer i for input j', and $\sigma(\cdot)$ is some non-linear function. If $\sigma(x) = x$, a linear and thus invalid activation function, then each unit in layer is a linear regression model. If $\sigma(\cdot)$ is the logistic function, then each unit in each layer is a logistic regression model of its inputs. Many $\sigma(\cdot)$ exist in the literature, and perhaps the most popular is the leakyReLU function, which we have extensively used in our project.

Much like linear or logistic regression, a neural network attempts to approximate the relationship or function between some input and some output data i.e. they are function approximators. The Universal Approximation theorem states that a finite-width one-layer neural network can approximate any function. Furthermore, the capability of a neural network to approximate *any* function depends exponentially on the amount of units per layer and amount of layers, making them extremely powerful approximators, given the right weights. This type of statistical models have yielded extraordinary results in many hard problems (digit and face recognition, natural language processing, image segmentation, and more), leading to strong academic and industrial interest.

The layer architecture described above is called the linear layer, which is further called fully connected in the case where all previous inputs are used for every feature of that layer. A variant on that architecture is the convolutional layer, which is used throughout this project, and has shown extremely good performance in computer vision problems. An introduction to the convolutional layer is outside of the scope of this report, but I refer the reader to [8] for an excellent introduction to the topic.

3.2 Supervised Learning

In supervised learning, a model is created of one or several datasets $\mathcal{D}_i \in \mathcal{D}$ composed of (input, label) pairs $(x_{i,j}, y_{i,j})$. The model, typically a neural network, receives $x_{i,j}$ as input and outputs a prediction $\hat{y}_{i,j}$. The prediction is then compared to the true label through a loss function $L(y_{i,j}, \hat{y}_{i,j})$ such as the mean squared error loss or the cross-entropy loss. The total loss is then the sum across all training data of the per-data-point loss $L = \sum\limits_{i,j} L_{i,j}$ and it measures the discrepancy between the predictions of the current model and the real data. An optimization algorithm then updates the network's weights to decrease the loss function and thus reduce discrepancy.

Gradient Descent (fig 3.3) is one such algorithm and it has been key to the popularization of Deep Learning, owing to its convergence speed and robust performance compared to other methods. In gradient descent, the weights θ_t describing a model at iteration t are adjusted in proportion to their influence on the loss incurred by the network. Said influence is calculated via the gradient of the loss function with respect to said weights.

$$\theta_{t+1} = \theta_t - \eta_t \nabla_\theta L$$

where η_t is the learning rate, which controls the change speed of the parameters.

Gradient descent leverages the fact that the neural network model is differentiable to gain insight into how altering θ will alter the performance of the model. In practice however, a sum over the full data-set for calculating L is computationally unfeasible given the typical data-set sizes and number of iterations involved. A stochastic variant of Gradient

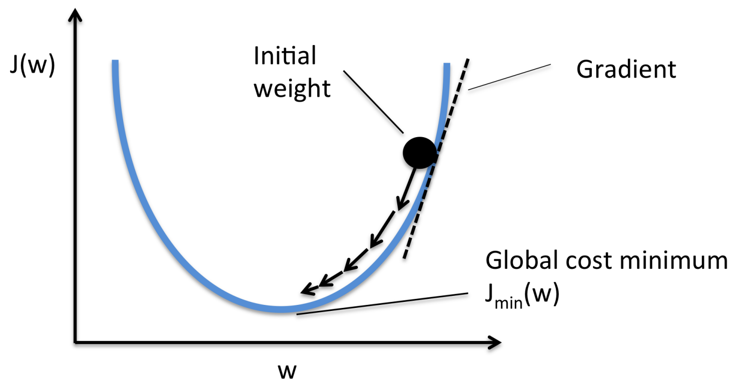


Figure 3.3: Gradient descent for a single parameter

Descent (SGD) is used, where the gradient is estimated only from a few data points. Many variations on SGD exist that improve on its convergence speed and capability to find better optima, such as SGD with momentum, Adam, or RMSProp.

3.3 Reinforcement Learning

3.3.1 Markov Decision Processes (MDPs)

Reinforcement Learning (RL) is a sub-field of Machine learning in which the performance of an agent is improved through interaction with an environment (Fig. 3.4). An agent senses the state of an environment, acts on it, and received a reward based on his interaction with said environment.

In the RL framework, the environment is seen as a Markov Decision Process (MDP), which is a mathematical model for decision making. An MDP is described by a tuple $< \mathcal{S}, \mathcal{A}, T, R >$ containing a set of states s, a set of actions a, a transition function $T(s,a) = p(s'|s,a)$ mapping states and actions to probabilities of reaching a given next state, and a scalar reward function $R(s,a,s') = p(r|s,a,s')$, which may sometimes be reduced to a deterministic reward $r(s,a,s') = E[R(s,a,s')]$ or even further to a form independent of the next state $r(s,a) = \underset{s'\sim T(s,a)}{E}[r(s,a,s')]$ [9]. The state is defined as the ensemble of variables that fully determine the environment, therefore the transition and reward functions of an MDP do not depend on the history of the environment, only on its present state and the actions taken.

An MDP may be described in graphical form such as in Fig 3.5, although this very quickly becomes impractical.

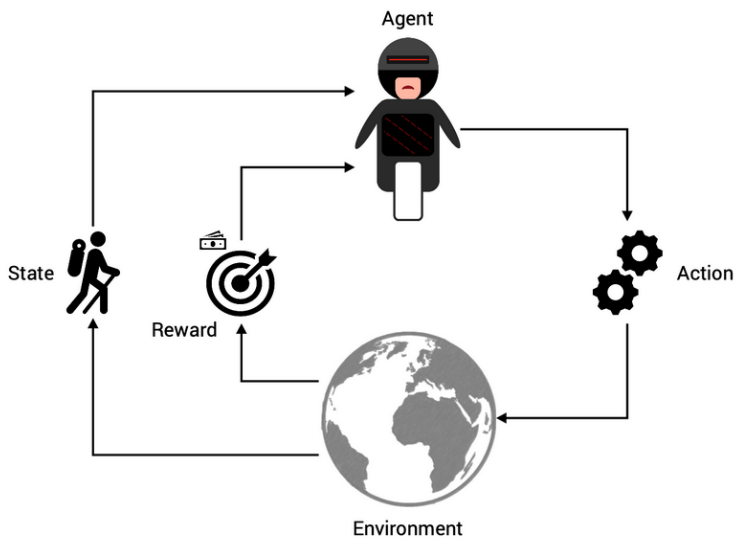


Figure 3.4: Visualisation of the Reinforcement Learning paradigm

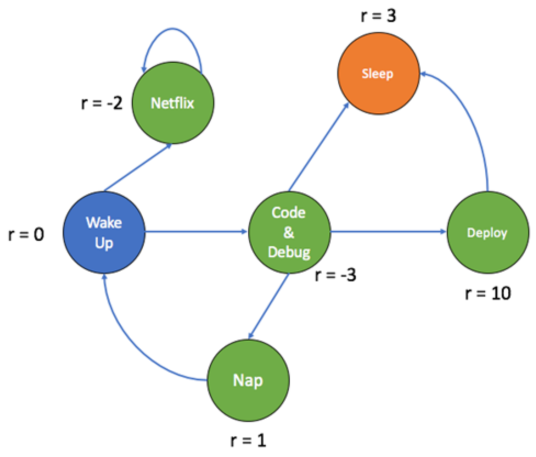


Figure 3.5: MDP of a student trying to finish his coding assignment. In this case, the rewards and transitions are deterministic - A given action in a given state always leads to the same state and results in the same reward

3.3.2 Continuous Partially Observable MDPs

Partially observable MDPs are a sub-type of MDPs in which the agent does not have access to the full state of the environment. Duckietown is one such MDP - The camera used by the agent does not give it information on the state of other vehicles outside its field of view. Therefore, the agent must maintain his own state S_t^a which may or may not accurately represent the environment's state. Many relevant MDP's are only partially observable. Continuous MDPs are MDPs in which at least the state space or the action space are continuous or where transitions occur continuously. Again, many relevant MDPs are continuous, including driving.

3.3.3 Returns, Policies, and Value Functions

We define a trajectory τ as a chain of states $s_0, s_1, ...$ traversed by an agent emitting a chain of actions $a_0, a_1, ...$, and the rewards reaped during that traversal $r_0, r_1, ...$. We define the return of said trajectory as:

$$\mathcal{R}(\tau) = \sum_{t=0}^{\infty} \gamma^t \cdot r_t$$

Which is the sum of rewards reaped by the trajectory, weighted by a discount factor $0 < \gamma < 1$ such that rewards far off into the future are assigned less value. This factor is required to have our infinite sum converge to some value, but it also models the effect of uncertainty by discounting far-off rewards.

A policy $\pi(s_t) = p(a_t|s_t)$ is a mapping from states to actions that determines what an agent will do based on its current state. Policies can be good or bad, earning large or small returns, and they can be expressed as a table from states to actions, or as a function.

The value function $V_\pi(s)$ of some policy is the expected return that a given policy would obtain were we to start following it at the given state and follow it for the rest of time. Thus, we may define

$$V_\pi(s_{t_0}) = \underset{\tau \sim \pi}{E}\left[\mathcal{R}(\tau)\right]$$

.

A related notion is that of the action-value function $Q_\pi(s, a)$ which describes the long-term reward obtained by π if action a is taken at state s and then π is followed for the rest of time.

$$Q_\pi(s, a) = \left[\mathcal{R}(s, a, s') + \gamma \cdot V_\pi(s')\right]_{s' \sim \mathcal{T}(s,a)}$$

The two are related through

$$V_\pi(s_t) = \underset{a_t \sim \pi(s_t)}{E}\left[Q_\pi(s_t, a_t)\right]$$

With V_π we can say which policy is best for a given state and MDP, the optimal policy $\pi^*(s_t)$ maximizing the value function.

$$\pi^*(s_t) = \underset{\pi}{\operatorname{argmax}}\, V_\pi(s_t)$$

3.3.4 The Bellman Equation

The Bellman Equation [10] is at the core of RL. Applied to the value function, it relates the value function of two subsequent states by a recursive relationship - The value of the current state with a given policy is the expected reward obtained by following our policy in this state plus the expected value of the following state, discounted by gamma.

$$V_\pi(s_t) = \underset{\substack{s_{t+1} \sim T(s_t, a_t) \\ a_t \sim \pi(s_t)}}{E}\left[R(s_t, a_t, s_{t+1}) + \gamma V_\pi(s_{t+1})\right] \tag{3.1}$$

It can be applied to the search for an optimal policy by realizing that the optimal policy is the one that maximizes the sum of reward now and value of the next states.

$$V_{\pi^*}(s_t) = \max_{a_t}\left[\underset{s_{t+1} \sim T(s_t, a_t)}{E}\left[R(s_t, a_t, s_{t+1}) + \gamma V_{\pi^*}(s_{t+1})\right]\right] \tag{3.2}$$

Relying on this relationship and a perfect knowledge of the environment's transition function, we can find the optimal policy directly [9]. However this solution is cubic in the number of states, thus infeasible even for moderate size MDPs. Furthermore we may not perfectly know the transition function. We must therefore turn to other ways of finding a good policy.

3.3.5 The L in RL - Learning

Approximate Dynamic Programming

In approximate dynamics programming, we don't have access to the true model of the environment, so we approximate it. Suppose we have an estimator parameterized by ϕ of the action-value function $\hat{Q}(s, a)$ e.g. a neural network. We know that the true action-value function, $Q(s, a)$, must satisfy the Bellman equation. Therefore, we can use the extent to which our estimate $\hat{Q}(s, a)$ does not satisfy Bellman's equation as a measure of the error of our estimate.

$$L_{\hat{Q}_\pi}(s_t, a_t) = \Bigg(\hat{Q}_\pi\big(s_t, a_t)\big) - \underset{s_{t+1} \sim T(s_t, a_t)}{E}\Big[\mathcal{R}(s_t, a_t, s_{t+1}) + \gamma \hat{Q}_\pi\big(s_{t+1}, a_{t+1})\big)\Big]\Bigg)^2 \Bigg|_{\substack{a_{t+1} \sim \pi(s_{t+1}) \\ a_t \sim \pi(s_t)}}$$

We have now transformed policy estimation into a supervised learning problem. By running some optimization algorithm over this loss function, such as Stochastic Gradient Descent (SGD), we arrive to a closer estimate of the true action-value function. However, two complications remain.

1. Our error function demands calculating an expectation over the possible future states and the reward function. We might not know these *a priori*.
2. SGD requires our data to be independent, identically distributed (i.i.d), so that the distribution is representative of the underlying distribution being sampled. That is not the case in an MDP or most real-world interactions, as the states seen by the agent depend on his previous states and actions.

The solution to both issues is to create an experience replay (ER) buffer, a list of state transitions that is then sampled at random. Given a sufficiently large buffer, state transitions are approximately i.i.d., and the expectation can be more accurately approximated. The loss then becomes, for a given sample of state, action taken, reward received, and state transitioned to $< s, a, r, s' >$:

$$L_{\hat{Q}_\pi}(s, a) = \Bigg(\hat{Q}_\pi(s, a) - \Big(r + \gamma \hat{Q}_\pi\big(s', \pi(s')\big)\Big)\Bigg)^2$$

Notice that, due to sampling the ER, we have lost the expectation over possible future states and the expectation over possible rewards:

$$\underset{\substack{s_{t+1} \sim T(s_t, a_t) \\ a_t \sim \pi(s_t)}}{E}\Big[\mathcal{R}(s_t, a_t, s_{t+1})\Big] \rightarrow r$$

$$\underset{s_{t+1} \sim T(s_t, a_t)}{E}\Big[Q_\pi\big(s_{t+1}, \pi(s_{t+1})\big)\Big] \rightarrow Q_\pi\big(s', \pi(s')\big)$$

Suppose now that our policy $\pi(s)$ is described by a function parameterized by θ e.g. another neural network. We can then optimize our policy by modifying θ such that it increases the value of Q, i.e. ascending the gradient of $Q_\pi(s,a)$. We do not have access to the actual Q function, but we can do the same on our estimate if it is a good estimate. We thus define the policy's loss:

$$L_\pi(s) = -\hat{Q}_\pi(s, \pi(s))$$

and then running stochastic gradient descent on this loss. Policy improvement has now be transformed as well into a supervised learning problem. This equates to optimizing the expected return of the policy for the MDP [11]. Methods that employ this trick are dubbed policy-gradient methods.

Deep Deterministic Policy Gradient

We finally have all the pieces needed to build our first RL algorithm for Duckietown, Deep Deterministic Policy Gradients (DDPG) [12].

DDPG contains 5 essential pieces, an actor $\pi_\theta(s)$ and its target $\pi_{\theta_2}(s)$, a critic $Q_{\pi_\theta,\phi}(s)$ and its target $Q_{\pi_\theta,\phi_2}(s)$, and an Experience Replay buffer. The only addition with respect to the previously presented elements is the addition of the targets. Essentially, both the actor and critic updates depend on themselves. This is thought to cause training instabilities through positive feedback loops which initially prevented application of neural networks to these sort of problems.

To overcome this phenomenon, we decouple the right-hand side from the left-hand side in the losses for the actor and critic by introducing a copy of the actor, the actor-target, and of the critic, the critic-target.

$$L_{Q_{\pi_\theta,\phi_2}}(s,a) = \left(Q_{\pi_\theta,\phi}(s,a) - \left(r + \gamma \underset{a\sim\pi_{\theta_2}}{E}\left[Q_{\pi_{\theta_2},\phi_2}(s', \pi_{\theta_2}(s'))\right]\right)\right)^2 \tag{3.3}$$

$$L_{\pi_{\theta_2}}(s) = -Q_{\pi_\theta,\phi}(s, \pi_\theta(s))$$

Thus, the neural networks that are actually trained are the targets, albeit with respect to the output of their non-target counterparts. Every timestep we will perform what's called a soft-update of our actor and critic to follow their respective targets: A weighted average of the non-target weights with the target weights so that they remain in sync:

$$\theta = \tau\theta + (1-\tau)\theta_2$$

$$\phi = \tau\phi + (1-\tau)\phi_2$$

The parameter τ is one of the many user-selected parameters of the method.

The original paper also proposed a few other tricks and improvements over the state of the art. The Ornstein-Uhlenbeck process for exploration tried to improve over standard random exploration by using a noise model that exhibits inertia. This choice was later shown to not be meaningfully useful, and we did not implement it. They also introduced the use batch normalization layers in all of the neural networks they use. These are discussed in section 3.6

Twin-Delayed Deep Deterministic Policy Gradient

TD3 [13] proposes a few tricks to substantially improve the performance of DDPG. These are based on the observation that DDPG suffer from an overestimation bias in its estimate of $Q(s, a)$, which accumulates through the Bellman equation.

Firstly, to mitigate the positive bias, they introduce a second critic, and take the minimum of the two critics as the true Q value estimate. The minimum operation counteracts the observed positive bias. This is where the "twin" in the name comes from.

Secondly, they find that the noise in the critic's estimate propagates through the actor update with cumulative effects, eventually leading to divergence. To reduce the noise in the Q estimate, they iterate more than once on the gradient descent of Q, allowing it to get closer to its true value.

Lastly, they add noise to the action take in the target update (right hand side of equation 3.3). This serves to smooth out the noise in the critic's estimate and prevent it from accumulating further.

3.4 Domain Adaptation

Neural networks suffer from some shortcomings that impede their implementation on areas where the cost of palliating their failings is higher than the benefits gained from their application: Their results are hard to explain and interpret, it can be hard to specify their behaviour and they can be notoriously brittle to shifts in the data distributions they process.

It is that last shortcoming, brittleness, that domain adaptation tries to combat. Domain Adaptation is a sub-field of Transfer Learning research, itself a subfield of Machine Learning. In Transfer Learning, we wonder how the knowledge gained in learning one task can be applied to a second task. Formally, we define a task $\mathcal{T}$ as a mapping from a feature space $\mathcal{X}$ to a label space $\mathcal{Y}$, which we do not know but seek to model. Transfer learning tries to apply a model Ψ to learn a set of source tasks $< \mathcal{T}_s >$ and achieve good predictions in a set of target tasks $< \mathcal{T}_t >$ with fewer examples than would otherwise have been needed, or, in the extreme, no examples whatsoever.

Domain Adaptation further constrains this definition by supposing that we have access to the features of $\mathcal{T}_t$ but not its labels, and that the label space for $\mathcal{T}_s$ and $\mathcal{T}_t$ are the

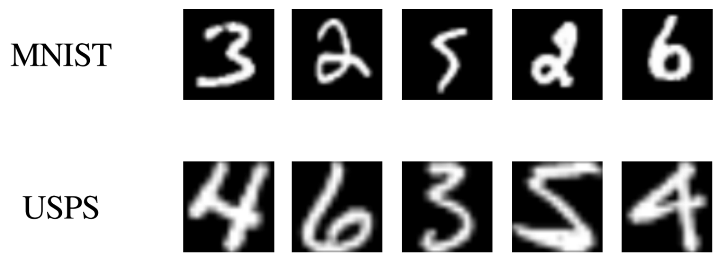


Figure 3.6: MNIST and USPS

same i.e. that we are trying to solve the same task in all cases. In image recognition, we may have access to photos of dogs, but not to the label stating that the photo represents a dog. This is crucial in applications where data is abundant but its labelling would be too expensive too produce, such as cases where data can be massively extracted from the internet.

A standard algorithm trained on the US Postal Service digit recognition data-set to 99% accuracy achieves accuracy as low as 50% when applied to MNIST (Fig 3.6). A Domain Adaptation algorithm implements techniques that prevent this. The hope is that the algorithm will focus on the patterns that are invariant across images, such as the general shape of a nine, and ignore non-relevant patterns such as the color the nine is printed on. Note that while the definition of Domain Adaptation does not constrain its application to neural networks only, in practice Domain Adaptation techniques often focus on this type of algorithms.

3.4.1 State of the Art

What follows is far from a complete survey and is meant to provide the reader with an intuition for the main research avenues in the field. Refer to [14] for an extended listing of DA surveys, techniques, benchmarks, and papers.

The theoretical side of Domain Adaptation has mainly focused on defining what constitutes a domain, defining the difference between domains (typically as some distance of statistical distributions), and establishing bounds on the loss of performance of a model when passing from one domain to another [15][16][17].

There is an array of approaches when creating models that perform properly in the source and target domains. Statistical moment matching ensures that some statistical moments of the data (average, co-variance) are identical at several stages of the network, such as the input or after passing through a Feature Extractor backbone. The objective is to achieve a representation that is more domain-invariant than the original one. Deep Domain Confusion [18] match a pseudo-mean of the features after an initial layer of the neural network. CORAL [19] matches second-order statistics at the input of the learning algorithm, whereas DWT [20] does it in every layer, but does so by groups of features as opposed to globally.

Interestingly CORAL states that *"One might instead attempt to align the distributions by whitening both source and target. However, this will fail since the source and target data are likely to lie on different sub-spaces due to domain shift."*. This is similar to the approach of DWT, which achieves higher performance than CORAL nonetheless. If CORAL's intuition is right, perhaps applying CORAL on a per layer basis could achieve significant performance increases.

AutoDial [21] takes a similar approach, performing a per-feature normalization akin to Batch Normalization but adjusting instead to a mean and a variance constructed from a linear interpolation of the source's and target's means and variances. The actual point in the linear interpolation is a learned parameter, much like the re-colorization in Batch Normalization is.

Another means of achieving a domain invariant representation is through alternative tasks. This is the case of DRCN [22], where a feature space apt for classification of the source domain and reconstruction of the target domain is found then leveraged for classification of the target domain.

DANN [23] instead uses adversarial training, with a part of the network being trained to render the features at a given layer as domain-invariant as possible.

A radically different approach [24] is the usage of generative networks to create labelled target examples. A generative network creates images in one style, potentially by adapting images in another style (Fig 3.7). This allows us to create the target version of a source example, whose label is known as we know the source label. Generative Adversarial Network's are often used for this purpose given their powerful generation capabilities. Russo et al. [25] propose doing the same thing in both directions instead.

Another branch of research uses ensemble methods - methods that use more than one model and try to leverage that fact to achieve increased performance with respect to an equivalent size single model. A common paradigm is that of the mean-teacher [26], a secondary network whose weights are a moving average of the main network. The main network is then enforced to maintain the same predictions as the mean-teacher. Another ensemble approach is DAEL [27], where a group of experts is trained to each specialize in a set of source domains. The unknown domain is then estimated using the most adapted expert, where most adapted is chosen according to some metric (prediction confidence in the case of DAEL).

Both DAEL and the mean-teacher paradigms are also examples of the consensus loss technique, where a network is forced to agree (i.e. reach a consensus) on variations of the data. In DAEL, the non-experts mean prediction on heavily perturbed data is forced

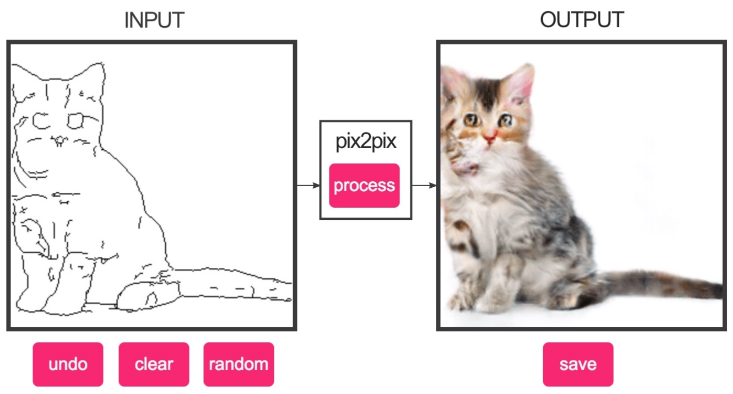


Figure 3.7: Generative models can translate images from one style to another, sometimes with shockingly good performance

to agree with the prediction of the expert on less perturbed data. In the mean-teacher scenario, consensus is forced between the teacher and the network. The aforementioned DWT also implements a consensus loss which is discussed later on.

Another popular loss used in DA to improve learning on the target is Entropy Minimization, where the network is encouraged for predictions with high confidence in the target domain. This leverages our knowledge that in classification tasks the output must be either A or B, and not an in-between, but is only applicable to classification tasks and not regression. Therefore it cannot be readily applied to Duckietown. DWT's MEC loss is an example of such a loss.

3.4.2 Tested techniques

Five domain adaptation techniques have been studied during this project for use within Duckietown: Domain Adversarial training of Neural Networks (DANN) [23], unsupervised Domain adaptation using feature-WhiTening and consensus loss (DWT) [20], Deep Reconstruction Classification Networks (DRCN) [22], Domain Adaptive Ensemble Learning (DAEL) [27], and a fifth method developed at IRT (DADAPT-IRT). The former two were implemented by my colleage Vincent Coyette, whereas the latter three were implemented by me.

DRCN

DRCN [22] is the earliest amongst these methods and builds off a basic idea - how could we use the data of the target domain to build features useful to its classification? It proposes to use a split Auto-encoder and classifier architecture for this purpose.

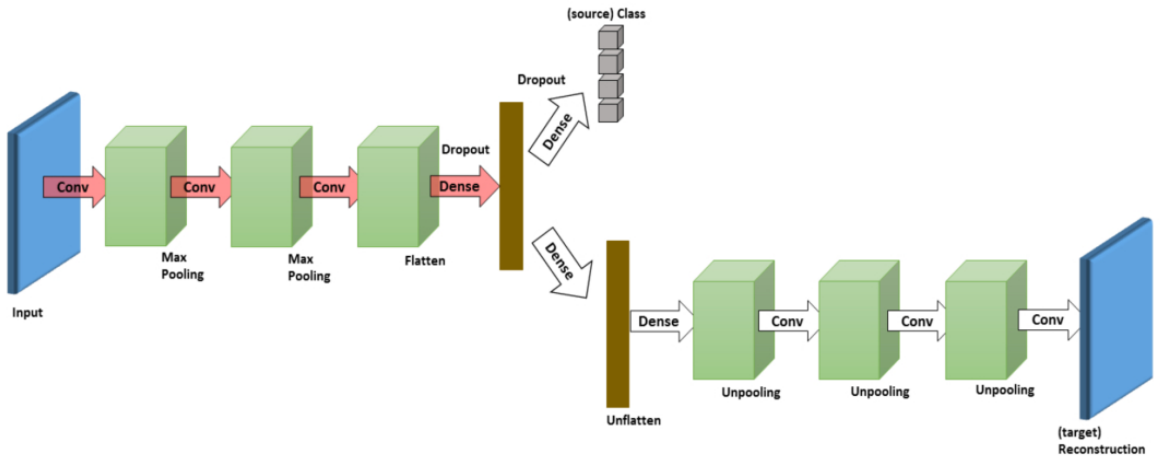


Figure 3.8: DRCN's architecture

An auto-encoder is an architecture with several layers (the encoder) that contain progressively less features, followed by several layers (the decoder) with an increasing number of features up to the original dimension of the input. When the auto-encoder is trained with the Mean Squared Error loss function, it learns to re-create the input images. Futhermore, many single-domain computer vision algorithms rely on an encoder creating a set of features, followed by a fully connected linear layer performing classification over said features. DRCN combines the two, using the same encoder for both. Figure 3.8 illustrates its architecture.

During training, alternate batches of source and target images are passed to the network, which is asked to classify the former with its labelling head and reconstruct the latter with its decoder head. The total loss used to optimize the network is the sum of the losses from these two tasks, weighted by a factor from 0 to 1. The hope is that the latent space created by the encoder, which are by training good for labelling source images and reconstructing target images, will also be good for labelling target images. No theoretical backing is provided as to why that might be the case. After training is finished, images of the target domain are passed through the network, and the labelling head is used to classify them.

Although the original paper touts some impressive results for such a simple approach, we were unable to replicate them, consistently obtaining results as much as 10% lower than the ones in the paper. Initially we thought this to be an error in our implementation, but we later found out another initiative [28] to replicate the paper that had found results similar to ours.

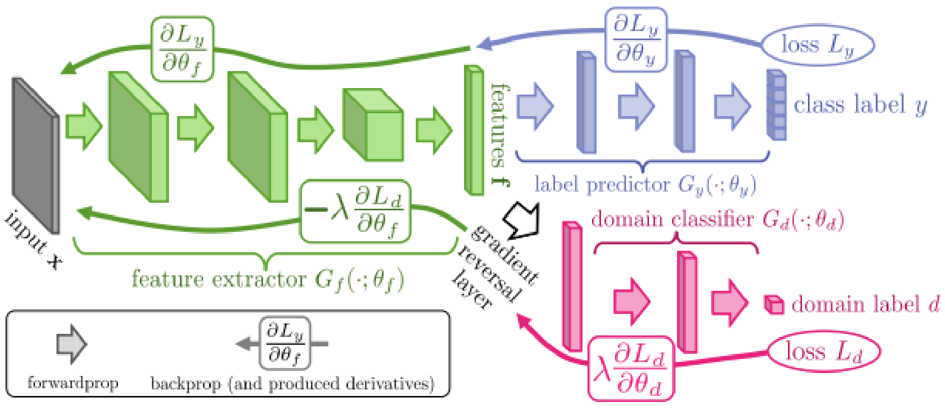

$\frac{\partial L_y}{\partial \theta_f}$
$\frac{\partial L_y}{\partial \theta_y}$
loss L_y
class label y
features f
label predictor $G_y(\cdot;\theta_y)$
$-\lambda\frac{\partial L_d}{\partial \theta_f}$
input x
feature extractor $G_f(\cdot;\theta_f)$
gradient reversal layer
domain classifier $G_d(\cdot;\theta_d)$
domain label d
loss L_d
$\lambda\frac{\partial L_d}{\partial \theta_d}$
forwardprop
backprop (and produced derivatives)

Figure 3.9: DANN's architecture

DANN

DANN [23] builds on earlier works attempting to establish a bound on measures of domain divergence, which quantify the difference between one domain and another assuming one tries to approximate them with a given type of statistical model. Said earlier work had shown that although the real domain divergence was too difficult to quantify, one could empirically estimate it via a learning algorithm trained to discriminate between domains. Said estimate was minimized when a representation of both domains was found in which either domain was indistinguishable from each other - a domain invariant representation.

To achieve that, DANN uses adversarial training. In adversarial training, two parts of the network have conflicting objectives. This way of specifying objectives can be more convenient and/or lead to higher performance than a non-adversarial counterpart.

In DANN, a domain discriminator and a classification head share the same encoder body. The task of the classification head is to classify data, whereas the domain discriminator attempts to detect the domain to which the current data belongs. However, the discriminator is connected to the encoder by a novel "gradient reversal layer", which reverses the sign of the gradient when back-propagating it through the network. This forces the encoder to update itself as to minimize the probability of the discriminator discriminating the domain, rendering the encoder features domain invariant. The encoder and the domain discriminator are adversaries - one tries to discriminate domains whereas the other tries to confuse them. Fig 3.9 illustrates this architecture.

The loss from domain discrimination $\mathcal{L}_d$ is further weighted by a factor λ and added to the loss from classification $\mathcal{L}_c$, resulting in the following loss function:

$$\mathcal{L} = \mathcal{L}_c + \lambda\mathcal{L}_d$$

Despite DANN's sound theoretical foundations, its performance on handwritten digit

Accuracy	MNIST $\rightarrow$ MNIST-M	SVHN $\rightarrow$ MNIST
Source Only	52%	54%
DANN paper	76%	73%
DANN replica	79%	75%

Table 3.1: Performance of DANN across some of Digit-5's datasets

classification like tasks, shown in table 3.1 is lackluster compared to current methodologies. It is far from being immune to the domain shift, and does not manage to make improvements in certain harder data-set combinations such as SVHN to MNIST.

Our attempts to implement it in Duckietown were met with failure. The agent would not learn proper driving and would consistently under-perform with respect to our baselines. After some time attempting to fix the issue, we deemed it too difficult and decided to try alternative methods. For this reason, DANN does not appear in our results discussion.

DWT

DWT [20] is a method proposing two different techniques for domain adaptation: Feature whitening and a consensus loss. Whitening is achieved through a feature whitening layer or DWT layer, that re-normalizes features from the source and target domains to a unit Gaussian centred on the origin (a distribution akin to white noise, hence the name whitening). To do this for all features, the co-variance matrix Σ for each domain must be estimated. Since Σ is ill conditioned for small batches of highly dimensional data, the authors choose instead to apply whitening by groups of features of a given size. This layer is a generalization of the previously introduced Batch Normalization layer, which performs a similar operation, but does not consider correlations between different features when normalizing or the existence of multiple domains.

Secondly they propose a Min-Entropy Consensus loss to be applied to the target domain. This loss mixes two concepts: Entropy minimization losses try to ensure that the network outputs very high predictions, under the hypothesis that high predictions equal more confident predictions. Since a correct label in the target domain can only correspond to one class and not a mix of several, this loss discourages wrong hybrid predictions and encourages sharp, potentially correct ones. Consistency losses are based on the idea that if the network processes two augmented images (i.e. perturbed with distinct random transformations), the correct label for both of them is the same, as they both originate from the same image with one correct label. Therefore, we can encourage correctness by penalizing outputs that correspond to different labels for two images augmented from the same source.

DWT's Min-Entropy Consensus loss implements these two concepts simultaneously by

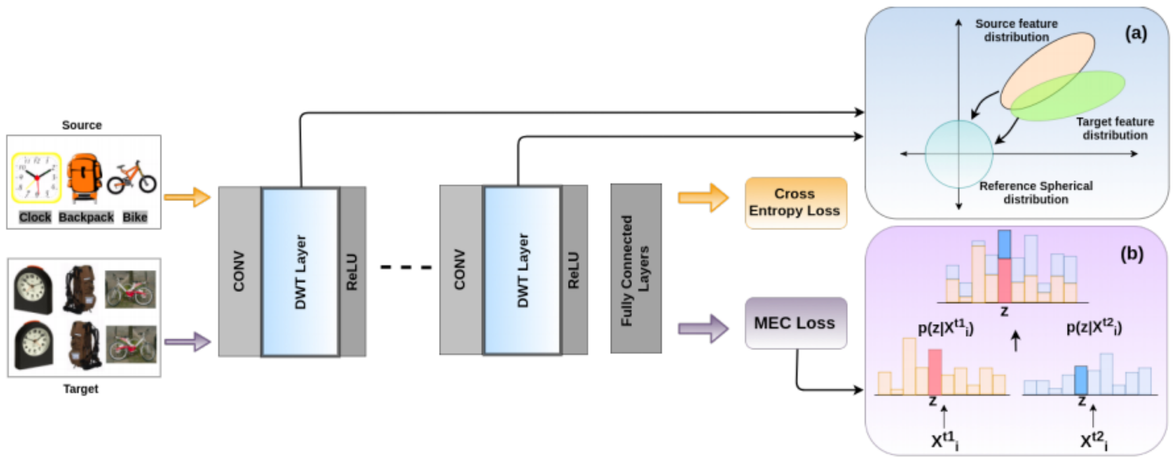


Figure 3.10: Schematic of DWT

Method	MNIST $\rightarrow$ USPS	USPS $\rightarrow$ MNIST	SVHN $\rightarrow$ MNIST	MNIST $\rightarrow$ SVHN
DWT	99.09%	98.79%	97.75%	28.92%
DWT+MEC	99.01%	99.02%	97.80%	30.20%

Table 3.2: Performance of DWT layers with and without the MinEntropy Consensus Loss

feeding the network 2 augmented images from the same original target image. The pseudo-label for that image is then chosen to be the class for whom the sum of both predictions is the highest - i.e. the class for which both predictions are maximally in agreement - as the average of the logarithms of the predictions. Thus the complete expression is:

$$L(x_t^1, x_t^2) = -\frac{1}{2}\max_{y\in\mathcal{Y}}(log(\hat{y}_t^1) + log(\hat{y}_t^2))$$

with $\mathcal{Y}$ the space of possible classes (digits 0 through 9 in Digit-5). Note that the logarithms punish low values (Entropy is minimized) while the sum punishes disagreement (Consensus is encouraged). Fig 3.10 presents a schematic of these two features.

DWT set state of the art results in a subset of Digit-5 when it was first published, achieving high results in MNIST $\leftrightarrow$ USPS and SVHN $\rightarrow$ MNIST. The poor performance in pure MNIST $\rightarrow$ SVHN is widespread across the DA literature and is theorized to be due to the lack of features of MNIST in comparison to SVHN, the latter being colourful, and containing numbers written in a wide range of scales and rotations. DANN, DRCN, and DADAPT-IRT suffer from the same shortcoming whereas DAEL bypasses it by using several source domains simultaneously, including domains with distributions containing the type of information that MNIST lacks in this scenario.

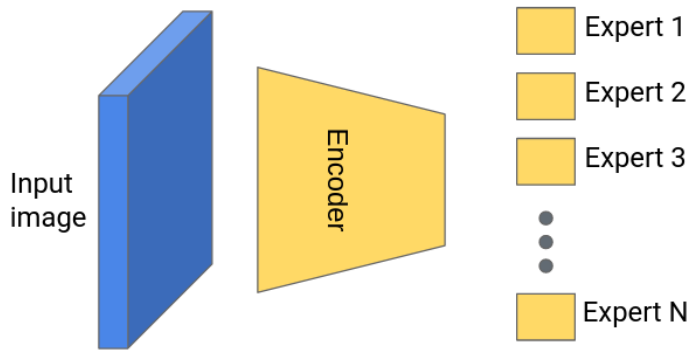


Figure 3.11: DAEL's Architecture

DAEL

DAEL, or Domain Adapative Ensemble Learning is an ensemble method, a set of neural networks that collaborate to be more effective at a given task than another network of equivalent size. Furthermore, unlike the previous methods, DAEL is designed to be trained with multiple source and target domains, in an attempt to leverage information from multiple domains when addressing a new one. This was of particular interest to us, as through the image transformations described in sec 2.4 we had near unlimited capability to generate source domains. On target domains, the network is trained through self-supervised learning, by generating pseudo-labels for the data and treating them as true data.

In DAEL the backbone of the network feeds not one but multiple classification heads, one for each of the source domains, which are dubbed experts. These experts specialize in correctly classifying images from each of the source domains. During each training batch, a mini-batch is drawn from each source domain 1...K and from each target domain.

The expert corresponding to domain $i \in 1...K$ is trained through minimization of a cross-entropy loss between his prediction and the true labels.

$$L_E = \sum_{i=1}^{K} CrossEntropy(y_i, \hat{y}_i)$$

Furthermore, the experts are encouraged to reach a consensus through a consistency loss that penalizes experts from disagreeing on average with the head specializing in the current domain.

$$L_{CR} = \frac{1}{K}\sum_{i=1}^{K}(\hat{y}_i - \frac{1}{K-1}\sum_{j\neq i}^{K}\hat{y}_j)^2$$

Crucially, experts do not see the same input as non-experts. The input to a domain expert is weakly augmented, by applying some minor flips and shifts to the input data in the original implementation. The input to a non-expert is augmented more drastically.

Lastly, whenever facing a target domain during training, a pseudo-label is created from the expert's predictions. A pseudo-label is a response generated by the network and taken to be the correct answer for a data-point for which we do not have a label. The experts, including the generator of the pseudo-label, are then trained to agree, on average, with said pseudo-label through a cross-entropy loss.

The pseudo-expert is chosen by taking the expert that outputs the highest prediction value for any class, so long as it exceeds a threshold of 95%. This method interprets final layer output, which is bounded between 0 and 1, as a probability of correctness of a guess or "degree of confidence". This interpretation is commonplace through the literature, but has little theoretical backing.

$$L_u = CrossEntropy(y', \frac{1}{K}\sum_{i=1}^{K}\hat{y}_i)$$

The final loss of the network is then

$$L = L_E + L_{CR} + \lambda L_u$$

Figure 3.12 schematizes the 3 losses for an image recognition task over 5 source data-sets and one target data-set.

Unlike the other methods presented so far, DAEL uses several domains when training. This would render the comparison of results to other methods unfair, as DAEL would have trained on a much larger amount of data than other methods. The original DAEL paper takes this into account and retrains many of the other methods feeding them data from all source data-sets. Any further differences in results are assumed to be from DAEL's efficient use of mixed source data.

DAEL obtains significant increases with respect to other methods in classification performance, matching and sometimes exceeding the performance of a network trained solely on the target data and its labels (an Oracle). The authors also provide a comparison of results against DANN and other techniques, which they outperform significantly. Table 3.3 showcases an excerpt from DAEL's results

We managed to replicate all of DAEL's results on handwritten digit classification with the exception of its adaptation to MNIST-M, where we fell short by a 5%. We never managed to find the source of this discrepancy.

Regressive DAELs

DAEL as described relies on an interpretation of expert outputs as a degree of confidence, a view that is incompatible with its application to a regression task. In such a task,

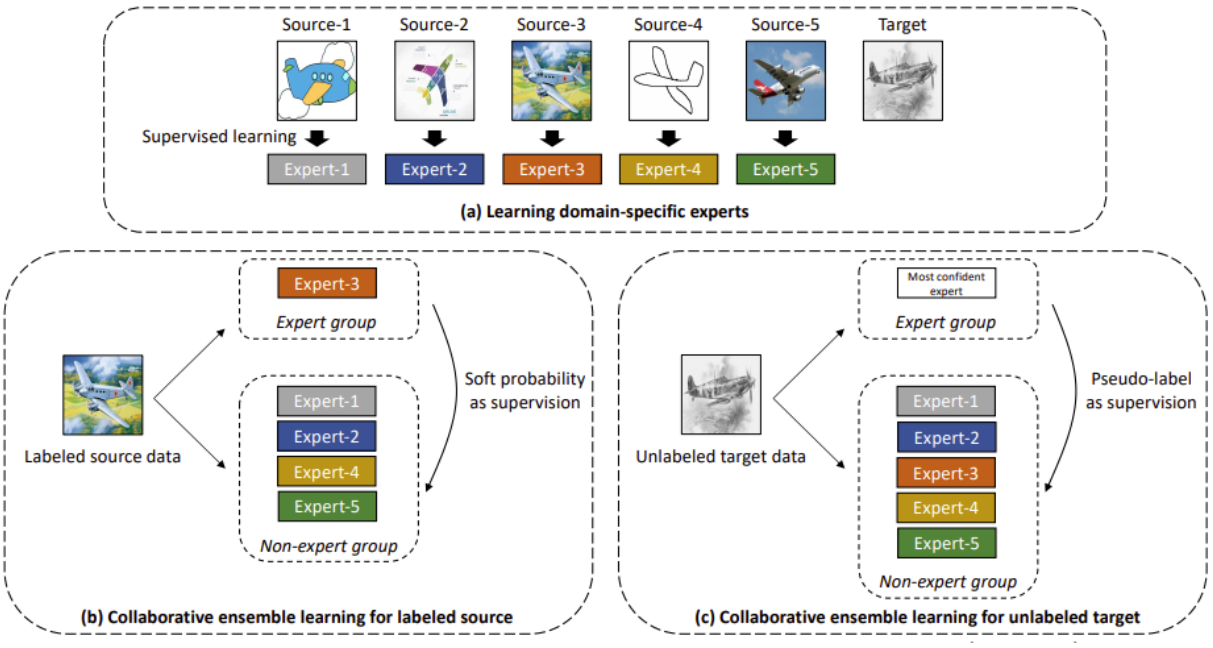


Figure 3.12: DAEL's losses

Method	MNIST-M	MNIST	USPS	SVHN	SYN	Avg
Oracle	95.36%	99.50%	99.18%	92.28%	98.69%	97.00%
DANN	83.44%	98.46%	94.19%	84.08%	92.91%	90.61%
DAEL	93.77%	99.45%	98.69%	92.50%	97.91%	96.46%
DAEL Replica	99.43%	88.63%	99.19%	92.23%	98.02%	95.50%

Table 3.3: Excerpt from DAEL's benchmarks

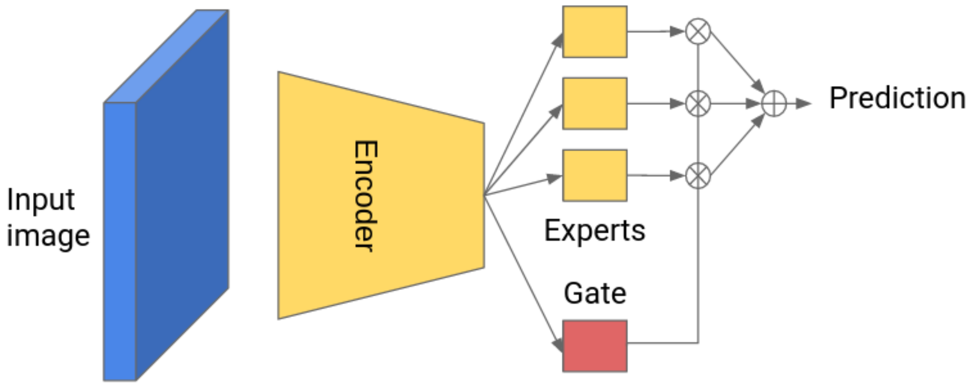


Figure 3.13: Gated DAEL's Architecture

the output of the network is a real value describing the action to be applied. Therefore, there is no number that can be interpreted as a degree of confidence.

We performed a few modifications to adapt DAEL as a regression task, building three variants: Regressive DAEL, Gated DAEL, and Alternative Gated DAEL. All of these substitute all cross-entropy losses from the original DAEL for mean squared error losses, as the former are unsuited to regression tasks.

Regressive DAEL works exactly like DAEL but without an unsupervised loss:

$$L = L_E + L_{CR}$$

Gated DAEL (Fig 3.13) implements the Mixed Mixture Of Experts (MMOE) [29] approach, adapted to a multi-task setting by a research team at Youtube [16], where a domain filter or gate is built from a single linear layer which, from the same latent features used for classification, outputs a vector f with K components, one for each expert, predicting which expert is to be applied in the current setting. We add a filter loss to the total loss, as a cross-entropy between the actual source domain and the source domain prediction by the filter.

$L_f = \text{CrossEntropy(Source Domain, Predicted Domain)}$

When facing unsupervised examples, the filter outputs a prediction that is then interpreted as a linear weighting to be applied on each expert to obtain the true label.

$$y_u = \sum_{i=1}^{K} f_k \cdot y_{u,k}$$

Unlike the original DAEL, there is no notion here of an acceptable level of confidence. The final loss for Gated DAEL thus is:

$$L = L_E + L_{CR} + \lambda L_u + L_f$$

Alternative Gated DAEL is very close to Gated DAEL. Note that the filter in Gated DAEL is performing a classification task and thus its outputs are amenable to being interpreted as confidence in the same way the outputs of the experts in the original DAEL were. When facing unsupervised examples, the filter outputs as before a vector describing the experts to be used. If said prediction is over a given threshold (95%) for any expert, we take that expert and make his prediction into a pseudo-label for the target domain.

$$\begin{aligned} &k = \underset{k}{argmax}(f_k) \\ &y_u = y_{u,k} \\ &L_u = \begin{cases} \left(y_u - \frac{1}{K-1}\sum_{i\neq k}^{K} y_{u,i}\right)^2 & \text{if } max(f_k) \geq threshold \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{3.4}$$

We verified the performance of the DAEL variants on the results of the original paper (Table 3.4) to ensure that any loss in performance with respect to the original implementation was not unacceptably high. While the basic regressive DAEL suffers a very strong performance hit, the other two regressive variants of DAEL show only slightly lower performance. These performance losses were systematic and thus worrying, but nevertheless the modified variants still were largely better than DANN and other methods to which the original DAEL was compared.

Method	MNIST	MNIST_M	USPS	SVHN	SYN	Average
DAEL Replica	99.43%	88.63%	99.19%	92.23%	98.02%	95.50%
Regressive DAEL	91.39%	60.07%	89.68%	63.38%	72.70%	75.44%
Gated DAEL	99.38%	76.70%	96.02%	89.82%	96.97%	91.78%
AltGated DAEL	98.86%	75.76%	97.10%	89.86%	96.09%	91.53%

Table 3.4: DAEL's testing accuracy on Digit-5

5th method

The last method we tested, DADAPT-IRT, was developed for the purpose of domain adaptation at IRT. As the method is currently undergoing anonymous open review, I will not disclose details about it.

DADAPT-IRT, which is also based on using neural networks, combines several of the approaches seen thus far to separate "semantic" information from "style" information. Here, "semantic" refers to all information useful for performing the given task in all domains, like the general shape of a digit in MNIST or SVHN, whereas "style" refers to all

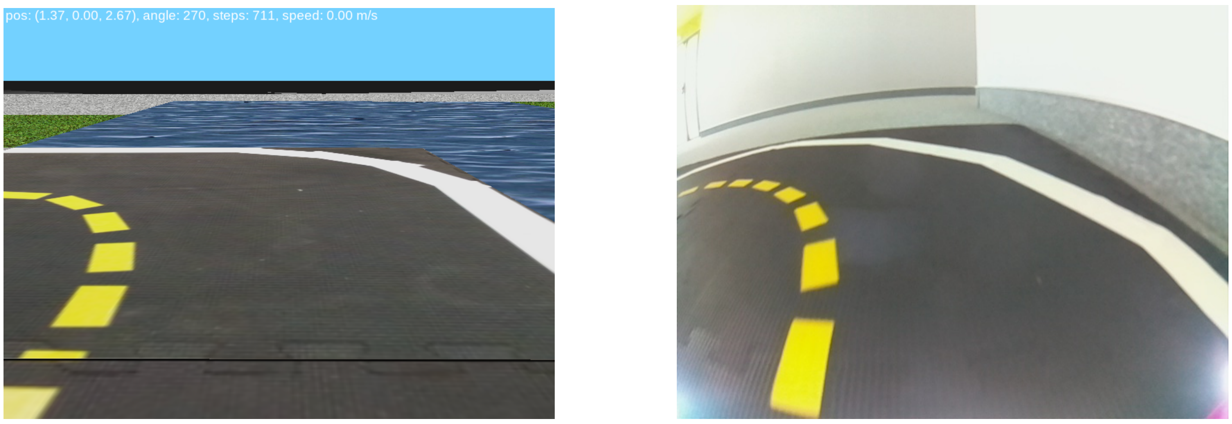

pos: (1.37, 0.00, 2.67), angle: 270, steps: 711, speed: 0.00 m/s

Figure 3.14: Side to side comparison of reality and simulation

information for a task that is not semantic. The information is explicitly separated and then the semantic information is used to perform classification whereas style information is used to reinforce the method's learning. The method achieves significantly strong results in some Digit-5 tasks, slightly under-performing

3.4.3 Domain Adaptation in Duckietown

We applied Domain Adaptation to closing the simulation to reality gap. Much like an algorithm fails to generalize from USPS to MNIST, an algorithm trained in a simulation fails to generalize to the real-world.

Needless to say, Gym-Duckietown looks nothing like the real world. See figures 3.14 for a side to side comparison of the same spot in the simulator and the real world: Textures are richer, the horizon contains objects and dynamic lighting, the camera angle and the geometry of the lane is different, the optical deformations introduced by the camera are different too. Everything about the physics model is different as well: The simulator's dynamic model does not consider inertia, slippage of the wheels, imperfect fits between tiles and associated bumps, delays in communications between the agent and the camera or actuators, variable duration of timesteps, or motor transients. Closing the simulation to reality gap here is an extremely complicated task. Domain adaptation combats discrepancies in input (i.e. images) but not dynamics.

In section 3.5 we presented our generation of a data-set for the simulator. A similar technique is carried out to generate a data-set for reality, except the agent is not an autonomous driver but a human.

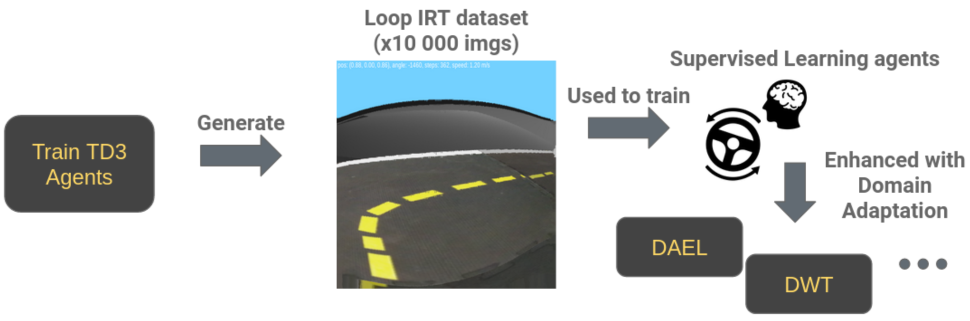

Train TD3 Agents
Generate
Loop IRT dataset (x10 000 imgs)
Used to train
Supervised Learning agents
Enhanced with Domain Adaptation
DAEL
DWT
•••

Figure 3.15: Imitation learning training pipeline

3.5 Imitation learning

Imitation Learning is the modelling of the policy of one agent through supervised learning. Suppose we have a self-driving data-set generated by a human interacting a simulation such as Duckietown. In such a case, one could envision training a supervised learning model on said data-set instead of training a reinforcement learning agent. This has several advantages - Reinforcement learning agents are notoriously unstable and sensitive to small design changes, for one.

We may also use an RL agent to generate such a data-set (Fig 3.15). In our setting, a Python script runs the RL agent in a specified environment for a fixed amount of timesteps (500), constituting an episode, or until the episode ends through failure of the agent (described in section 2.3). The script saves every image-action pair, up to a given amount of images (10,000).

Given that data-set, an Imitation learning model can be trained to follow it, much like it would follow a human. This implies we only need to train a single RL agent instead of one per algorithm and target environment. Furthermore, many domain adaptation techniques from the existing literature are based around supervised image classification tasks. A supervised learning framework deviated less from this type of task than reinforcement learning, easing their implementation in Duckietown.

The agents trained share the same architecture as our original reinforcement learning agents, to ensure they are comparable.

3.6 Neural Architectures in Duckietown

All of our agents use similar or identical architectures to render them comparable to each other. This architecture is derived from the initial architecture used to train our Reinforcement Learning agent, itself derived from one of Duckietown's initial (untrained)

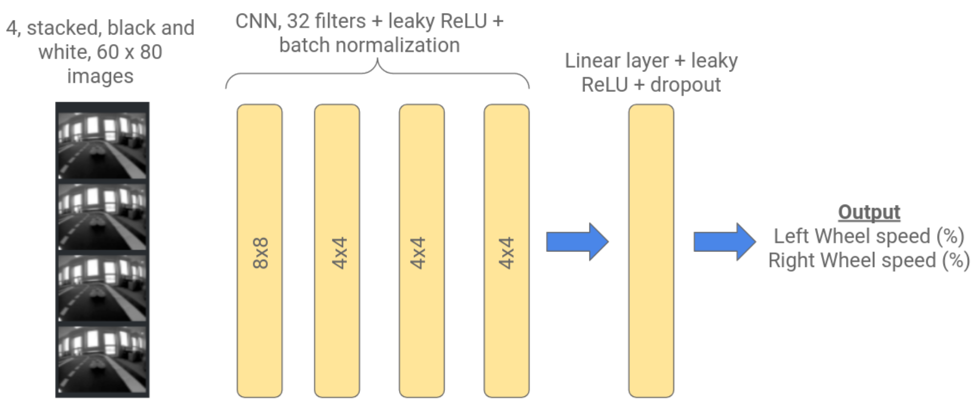

4, stacked, black and white, 60 x 80 images
CNN, 32 filters + leaky ReLU + batch normalization
8x8
4x4
4x4
4x4
Linear layer + leaky ReLU + dropout
Output
Left Wheel speed (%)
Right Wheel speed (%)

Figure 3.16: Basic architecture of RL and Imitation learning agents

architectures. It is illustrated in Fig. 3.16.

To accelerate the training of our agents, we pre-process the data fed to them in a few ways:

- We resize observation from Gym-Duckietown's native 480 pixels height by 640 pixels width to 60 pixels height by 80 pixels width. This preserves the aspect ratio while reducing data dimensionality by 1/64, speeding up computation
- We apply grayscaling as defined in ITU-R 601-2 and implemented by PIL and OpenCV. This reduces data dimensionality from 3 colors to 1, speeding up algorithms.
- We then scale our data from the 0-255 range to the 0-1 range.
- We further normalize the data. Normalization and scaling of data is known to ease training of statistical models through rendering features more easily comparable amongst them, such that features with typically large values do not overwhelm those with lower values.
- We lastly stack 4 images. This was first introduced by the Mnih et al. landmark paper [30], and it provides additional information about the environment's state by allowing agents to study changes in position. Otherwise there would be no straightforward way to deduce the speed of objects with one single image.

We divide each agent into two parts - A backbone and a head. This somewhat arbitrary division is commonplace in the literature and enables modularity and comparison of different agents. We have used mainly one backbone, which we dub cnn_duckieS_baseline. This backbone is composed of 4 convolutional + batch normalization layers followed by a fully connected linear layer. The convolutional layers use 32 filters each, a kernel size of 4x4 for all layers except the first which uses an 8x8 size, and a stride of 2 for all layers except the 4th one, using a stride of one. The linear layer outputs 512 features.

A batch normalization layer implements a similar technique as the one mentioned earlier for data normalization, but across layers. This layer keeps a running tally of the data mean μ and variance σ seen during training, which it uses to normalize input data $\hat{(x)} = \frac{x-\mu}{\sigma}$, preventing large discrepancies in magnitude between different features. It furthermore rescales the data via an affine transformation $x_{rescaled} = \alpha x + \beta$, where α and β are optimized to enhance performance, much like the rest of the network's parameters. This transformation has been shown empirically to improve performance [31]. In our implementation of DWT, we substitute these layers for domain whitening layers.

Two alternative backbones have been tested on occasion, named cnn_duckieS_m3sda and cnn_duckieS_m3sda_bigger. They are both larger (i.e. containing more parameters) variants of the baseline backbone, and substitute the last convolutional layer by one more linear layer. The main purpose of testing these were to assess the increase in performance from network size.

The head is always a single linear fully connected layer mapping the output of the backbone to two magnitudes, which control the Duckiebot's wheel rotational speeds as a % of max speed.

The reader may note that many of the choices made here seem arbitrary. In truth they were all chosen due to a belief or empirical evidence that they were the most highly performing amongst a selection of possibilities, or through arbitrary selection amongst a set of possibilities with unknown performance. Systematic testing of all of them, however desirable, was unfeasible given the time and resources available.

Chapter 4

Experimental setting

In this section I aim to present how we obtained the results presented in chapter 5: How performance was evaluated, which environments it was evaluated on, and what was evaluated.

4.1 Measuring performance

The performance of several self-driving agents is measured through the reward signal. Gym-duckietown includes a default reward signal, but we were unable to train satisfactory agents with it.

Handily, the environment also includes two invisible lines, illustrated in Figure 4.1 that define the center of each lane, corresponding to some "ideal" driving, along with a suite of functions to calculate the state of the agent's relative to that line. We used these to define our reward signal:

$$r_t = \begin{cases} v_t & \text{if } d \leq 0.1 \\ -1 & \text{otherwise} \end{cases} \tag{4.1}$$

where d is the distance between the agent's center and the center-line, and v_t is the magnitude of the speed of the agent in that timestep. While the function does not reward alignment with the center-line, the distance parameter is low enough that such behaviour is instrumentally rewarded i.e. staying aligned makes it easier to reap high rewards so the agent tends to do it. This reward signal allowed us to train agents with satisfactory performance across a range of environment conditions.

The performance of Supervised Learning agents is also measured with respect to that same signal. Although we started out by measuring the performance of these agents against a hold-out test-set of the data-sets they had been trained on (the traditional way to evaluate said methods) we quickly realized that test-set performance was a noisy predictor of

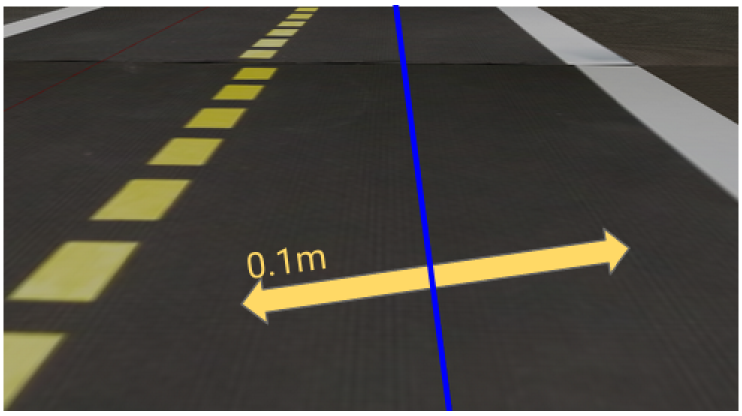


Figure 4.1: The reward for an agent is equal to its speed whenever it is within a set distance of the blue guiding line, which it cannot see. At any other instant, the reward equals -1.

performance in the simulated environment - the two measures were correlated, but only loosely so.

To make the evaluation score independent of the length of the evaluation episodes we chose to normalize it. The final score then is:

$$Score = \frac{\sum_{t=1}^{L_{ep}} r_t}{L_{max} \cdot R_{max}}$$

where L_{ep}, the duration of an episode, is a threshold L_{max} or cut short when the agent goes outside of a driveable tile or crashes into something, and R_{max} is the maximum reward attainable by the agent in any given timestep. We then run the agent's for a number of episodes and draw a confidence interval (CI) on the mean performance based on a t-test.

When driving in the physical Duckietown, we do not have a reward signal to evaluate our agents. Our evaluation is performed in a qualitative manner, with a rating from "No driving" to "Good driving". While this clashes with the quantitative nature of the rest of our study, it is validated by the nature of the performance of our algorithms in reality.

4.2 Test environments

We started out by gauging our performance on one of Duckietown's vanilla maps, "loop_empty" (Fig 4.2). We noticed however that there was a large gap between this circuit and the one we could afford to build with our reduced tile-set, casting doubt on the capability of simulation scores to generalize to reality scores. We therefore used the map editing tools of the environment to create an alternate one which more closely resembled ours, along with 3 variants to add distractions and varying levels of difficulty between them.

We also conserved loop_empty as part of the set,as a representative of circuits with

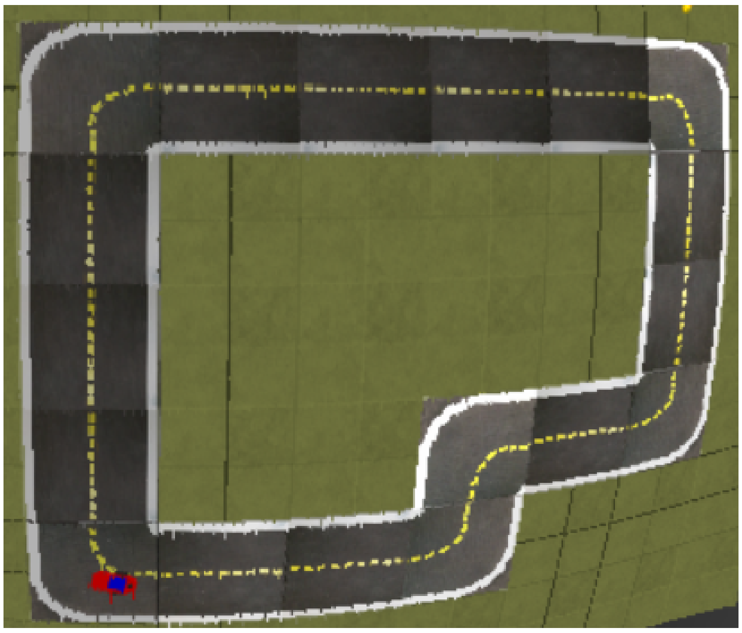


Figure 4.2: Loop empty - One of the vanilla maps within Gym-Duckietown

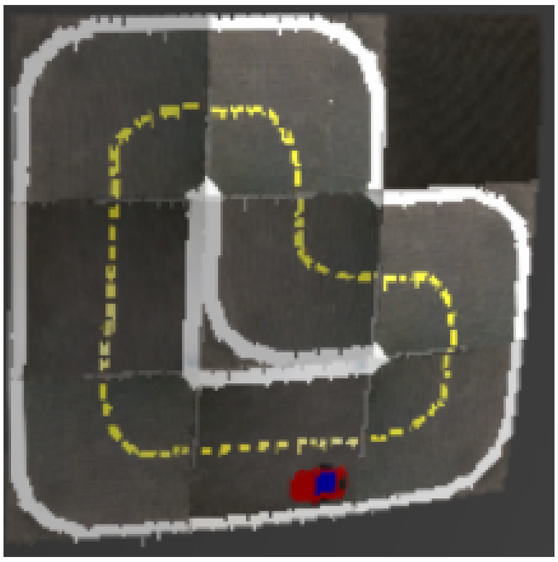


Figure 4.3: Loop_IRT

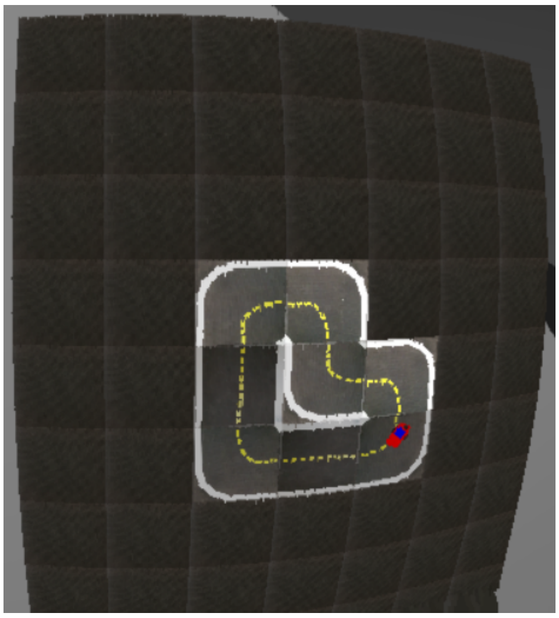


Figure 4.4: Loop_IRT_extended, extended with asphalt

different layouts to loop_IRT. Our final testing environment thus consisted of 5 maps. For each map we estimate the mean evaluation score and provide a confidence interval of 95% confidence around it. We also provide an average for all maps that served as a global evaluation metric.

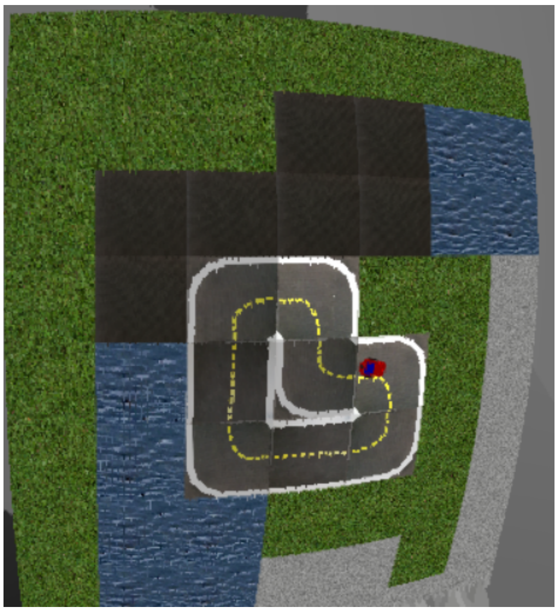


Figure 4.5: Loop_IRT_patchy, with terrain patches

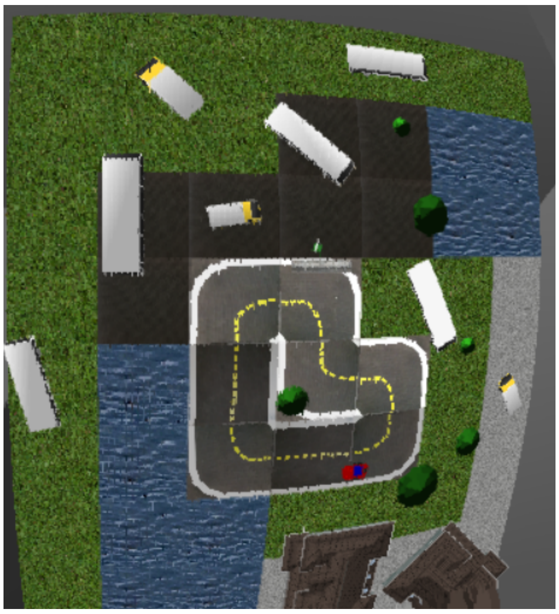


Figure 4.6: Loop_IRT_objects, extended with patches and objects.

4.3 Algorithms

We compare Imitation Learning implementations of the 5 algorithms outlined in section 3.4.2 and their variants. We compare them against three baselines.

The first baseline is the original RL agent used to generate the data-sets. This agent, dubbed TD3 Multimap, trained on all other maps available in the default gym-duckietown, including loop_empty. While we could have used a TD3 agent trained on loop_IRT as an agent, driving performance in this map was never satisfying vis-a-vis the multimap baseline.

The second baseline is trained to follow TD3 Multimap in loop_IRT. representing the adaptation capability of a naive imitation learning implementation.

The last baseline is a domain randomization baseline, trained in settings similar to those of the other baseline but enhanced through alteration of the input images according to the altered domains described in section 2.4. Domain Randomization is the name of a technique used to increase the performance and decrease the brittleness of Neural Networks by augmenting the source domain, and hoping the augmentation includes in some way the target domain. The nature of the specific perturbations depends on the data-set, and specific implementation but in the case of image processing typically includes flipping, cutouts and resizing, changes in brightness and contrast and solarization to name a few. Since domain randomization is conceptually simple and easy to implement with current frameworks, it provides a good baseline against which to compare performance. If an algorithm does not beat domain randomization, that casts serious doubt on the universality and effectiveness of the method.

All models are trained for the equivalent of 120 epochs over 1 dataset i.e. 1,075,200 images. DAEL and the generalist baselines, which train on larger datasets that also contain perturbed simulations, are trained for fewer epochs (24) but the same total amount of images. Additionally DAEL trains for 24 epochs on 3 unlabelled datasets of loop_IRT_extended, loop_IRT_patchy, and loop_IRT_objects. DRCN, DWT, and DADAPT-IRT only take a single target domain as input during training. Therefore, these algorithms were trained multiple times, with loop_IRT as their source and the dataset corresponding to their evaluation map as their target.

Chapter 5

Results

5.1 Simulation Results

The plot of Figure 5.1 gives an overview of our results inside the different simulated environments. For each algorithm, it displays only the variant with the highest average score across maps.

The imitation baseline, which was meant originally as a lower bound on achievable performance, is nearly the highest performing algorithm. Interestingly, the imitation baseline beats the RL algorithm in loop_empty, an environment in which the RL algorithm was trained and imitation baseline is adapting to. Furthermore, it adapts better across the tested environments, albeit marginally. It is beat only by Gated DAEL with a rather slim and inconsistent margin. There seems to be some sort of trade-off between performance within loop_IRT_objects and loop_empty.

DRCN's performance is significantly lower than that of the baseline across all maps, and decreases to essentially 0 in the most complicated map, loop_IRT_objects. This behaviour seems to be replicated by DWT, who furthermore has nil performance on loop_empty, and DADAPT-IRT, whose performance is fairly low or close to 0 on all maps. A worrying trend presents itself - The more an algorithm deviates from the standard supervised learning framework, the worse it seems to fare across all maps and particularly the more sensitive it seems to be to map changes.

5.2 Reality Results

The results in simulation where reinforced by our negative results in Sim2Real passage, shown in Table 5.1. We couldn't test all algorithms, so we focused on those that displayed good across-the-board results

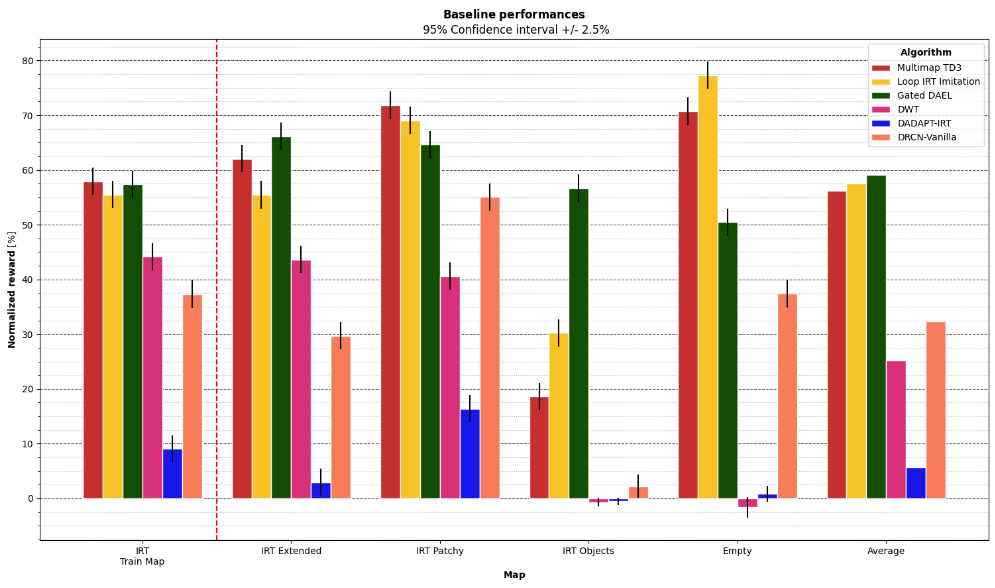


Figure 5.1: Comparison of all algorithms in Duckietown

	Sim. Generalization	Sim. 2 Real.
RL	Good*	None
Imitation	Good	Poor
DRCN	Medium	N/A
DANN	N/A	N/A
DWT	Medium	None
DAEL	Good	Poor
DADAPT-IRT	Poor	N/A

Table 5.1: Qualitative performance of each algorithm in Simulation and Reality. Reinforcement learning shows only medium adaptation capability when using an agent trained on a single map

As we pointed out earlier, the discrepancy between simulation and reality runs along two axes - discrepancies in the dynamics, and discrepancies in the observations. To isolate the former and study their impact, we saved a good trajectory in one of our simulated maps and ran it, from exactly the same starting point, in the real Duckietown. The results were very mixed, while occasionally the robot managed to complete the lap as in simulation, other times it would veer widely off-course. We conclude from this fact that the dynamics differences are too large to be ignored and that Sim-to-Real through domain adaptation exclusively is not possible.

5.3 Exploration of results

In what follows, we will attempt to address several questions regarding the performance of each algorithm with further data

5.3.1 Are the models too small?

The size of our models was fixed early on in our project and was kept constant to retain comparability across algorithms. A natural question is whether that was a good choice. In figure 5.2 we compare the effect of increasing model size across our two highest-performing algorithms: the imitation baseline and Gated DAEL. All other parameters are kept identical.

Unexpectedly, the effect of the size increase interacts strongly with the choice in algorithm. Whereas the imitation baseline significantly gains from the increased size, the opposite is true for DAEL. We conclude from this that the optimal size for each algorithm is unique to that algorithm. Performing a neural architecture search over possible sizes and illustrating the highest performing one was out of the scope for our project, but perhaps could have shed more light on the relative merits of each algorithm.

Disappointingly, testing the highly performing large imitation baseline on the real robot did not yield similarly improved results, with a qualitative performance similar to that of the small gated DAEL.

5.3.2 Data augmentation study

One striking conclusion can be drawn from our results: The only method outperforming the imitation baseline is also the only one using extensive data augmentation. Could that be the key to its success? In figure 5.3 we explore the interaction between data augmentation on the imitation baseline and the original data-set said baseline was trained on. Loop empty baselines were trained on a data-set generated on loop_empty, whereas loop IRT baselines

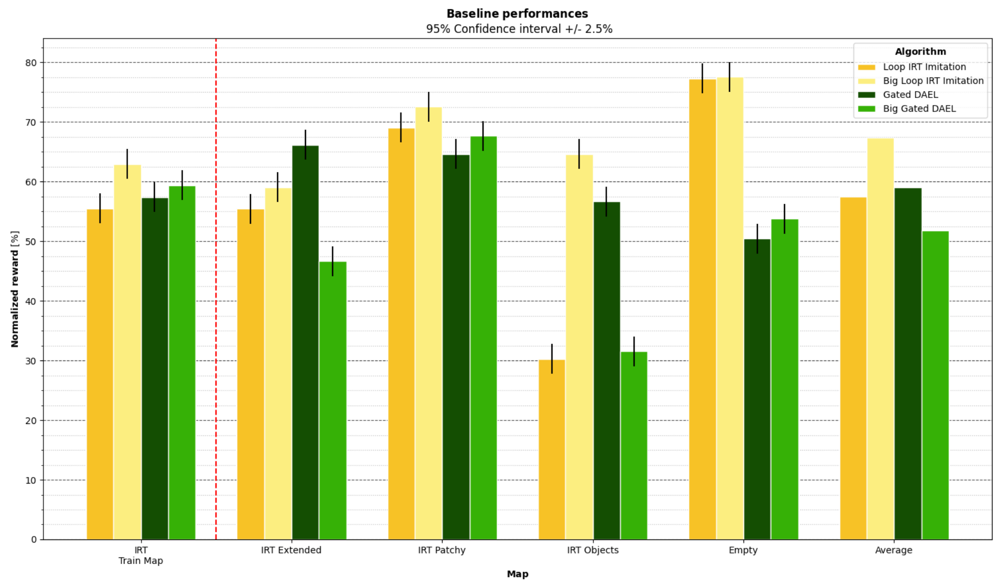


Figure 5.2: Effect of increasing parameters by 1000-fold on the imitation baseline and DAEL

were trained on a data-set generated on loop_IRT. The data augmentation scheme is the one discussed in section 2.4, which is the one used to generate DAEL's experts.

Note that the effect of data augmentation interacts with the source data-set: The agent trained on loop Empty seems to benefit greatly from it whereas the same is not true for the agent trained on Loop IRT

5.3.3 Performance of other DAEL variants

We can see in figure 5.4 that Gated DAEL is equal or superior to the other DAEL Variants across all tested maps. Both alternative versions of DAEL are clearly superior to Regressive DAEL, implying that some gain is being realized by the addition of unsupervised training and the gating head to the original DAEL for regression.

5.3.4 Does DAEL work?

Gated DAEL was the highest performing algorithm in our repertoire. One might assume it's working correctly, but is it really? The main concept behind DAEL is to train experts which each specialize in a given type of domain. To that end, we tried to render the source domains distinct from each other, reflecting different ways in which domains may differ

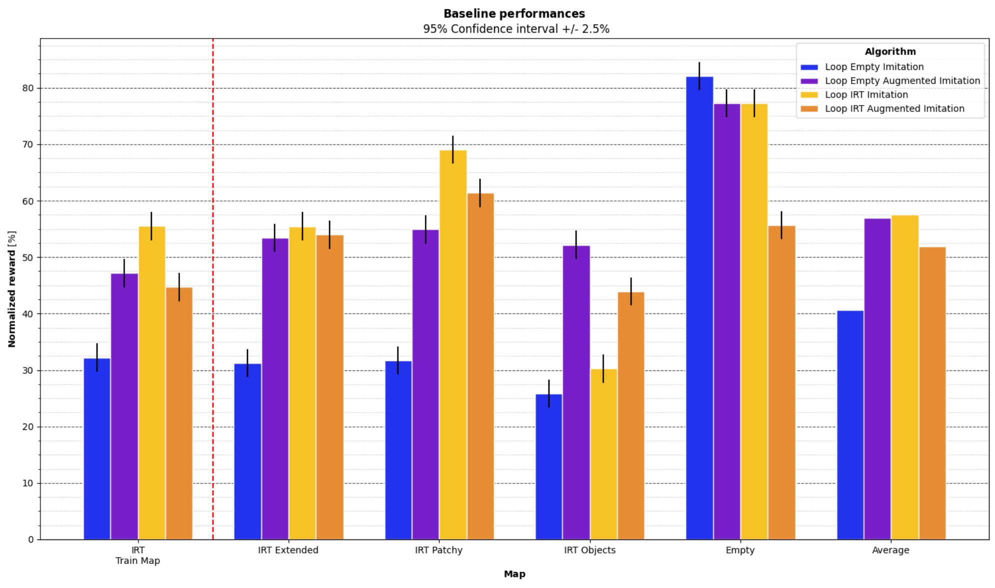


Figure 5.3: Comparing the effect of data augmentation and its interaction with the source data. Not all sources benefit equally from data augmentation

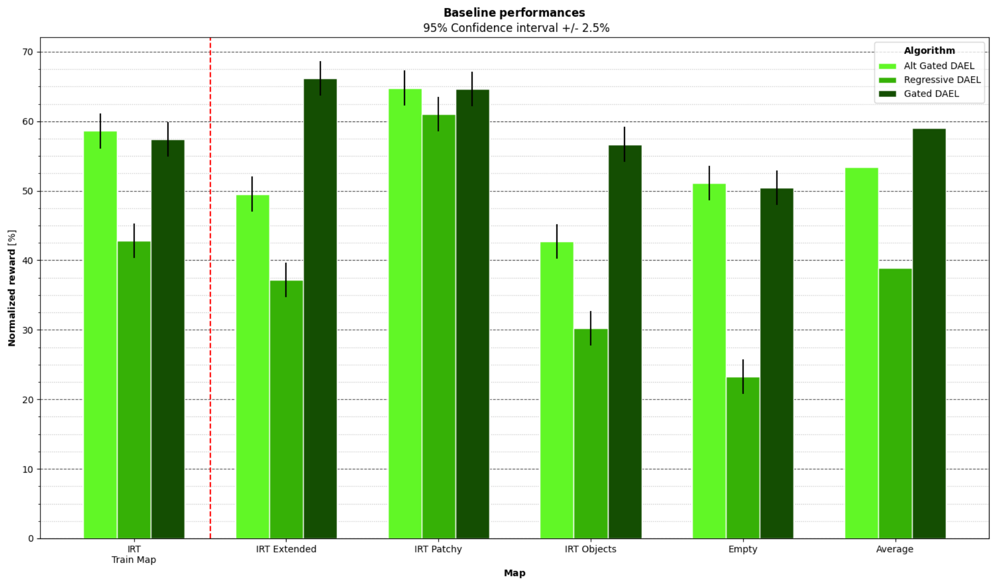


Figure 5.4: Performance of DAEL's variants

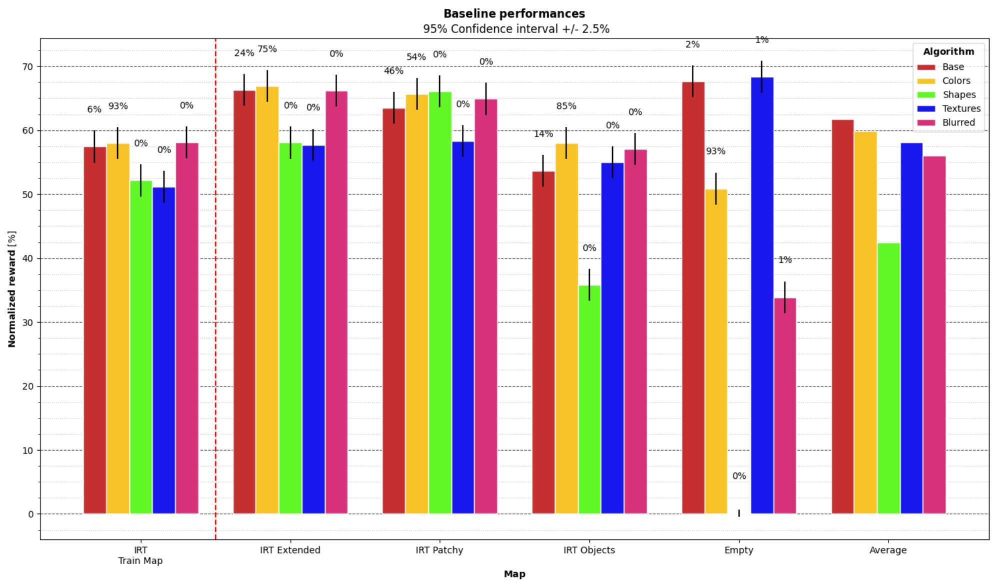


Figure 5.5: Comparing the performance of each Gated DAEL expert. The tag above a column indicates the median weighting of that expert on a given map

from one another such as different textures, different shapes, or different colors.

If DAEL is working correctly, one would expect that the highest performing expert on a given domain be chosen most of the time for that domain. Ideally one would also expect to find that different experts excel at different domains. Neither of those is true in this case, as figure 5.5 clearly shows.

Firstly, notice that the experts have fairly homogeneous performances across all maps. This is to be expected, since we applied heavy consistency regularization throughout their training. The shapes expert is the exception here, a fact for which we found no explanation.

Secondly, both the expert specialized in different textures and on the original data-set have performances significantly higher than that of the expert specialized in different colors in loop_empty. Nonetheless, the median activation for both experts is close to 0, whereas the colors expert is activated most of the time. In fact, the colors expert is the highest weighted expert in all maps, ranging from a close match with the base expert on loop IRT patchy to an overwhelming majority on loop IRT and loop Empty.

Thirdly, colors is chosen on empty despite its weak performance in that domain. It is precisely on loop IRT Patchy, the map where its performance is similar to that of other experts, that it is given a less overwhelming weight in the final action.

Ultimately it would seem that the relative advantage of DAEL vis-a-vis other methods is not the intuitions that led to its design. The factor with the largest weight on DAEL's performance is the performance of the colors expert, and the capability of consistency regularization to improve the actions of other experts upon image discrepancies that are not their domain. In other words, it is a (complicated) data augmentation technique!

5.3.5 Adapting DADAPT-IRT

The method DADAPT-IRT relies on a large amount of hyper-parameters - tuneable knobs at the user's control. We used it with its default parameters, as we lacked the resources for a full hyperparameter search. Nonetheless, we tried two variations on it.

1. DADAPT_IRT + MTAdam tries to automatically adjust the coefficients for different loss terms in DADAPT_IRT to correct unbalances in how the different losses contribute to the overall minimization objective. It does so by implementing a variant of the Adam optimizer during training as proposed by malkiel et al. [32].
2. Alternative DADAPT_IRT Architecture replaces some of the Duckietown baseline architecture by the neural architecture used in the original DADAPT_IRT proposal. Specifically, we eliminate all dropout layers and replace all leakyReLU activations for standard ReLU activations.

Figure 5.6 shows the results of this exploration. The updated optimizer failed, yielding results worse than the original DADAPT_IRT attempt. The alternative architecture had significantly higher results, including one particularly bizarre outlier: The version trained with loop_IRT_ patchy as a target showed considerably higher performance. This result was not a fluke, as we managed to replicate it a second time.

The performance of this particular model was not just larger in its target, but also across the evaluation suite, as shown in figure 5.7. Nevertheless it was consistently inferior to GatedDAEL. Exploring the training logs of this anomalous version of DADAPT-IRT did not yield any findings. Its losses on the source and target data-sets were similar to those of other variants.

5.3.6 Does DWT work?

While exploring the failure of DWT, I evaluated its performance while ignoring its target / source distinction. To recap, the domain whitening layers of DWT save two whitening transformations, one to be used for the source and one to be used for the target. If the intuition behind DWT is correct, then applying the source transformation to the target should yield worse results that correctly applying the transformations. In figure 5.8 we can see that is not at all the case, in fact performance improves greatly when doing just that. We infer from this that DWT's core idea is not working correctly in the Duckietown problem

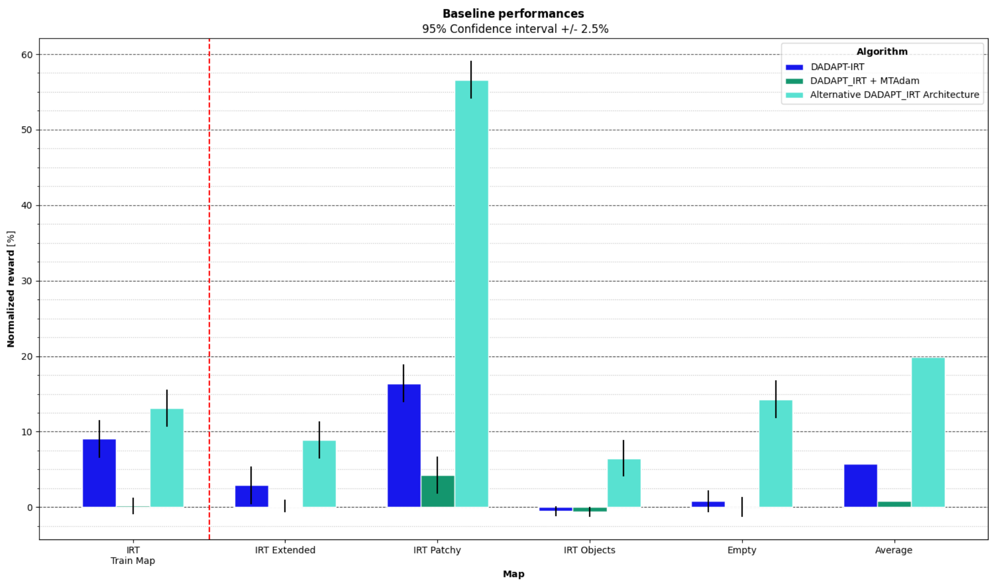


Figure 5.6: Altering DADAPT-IRT

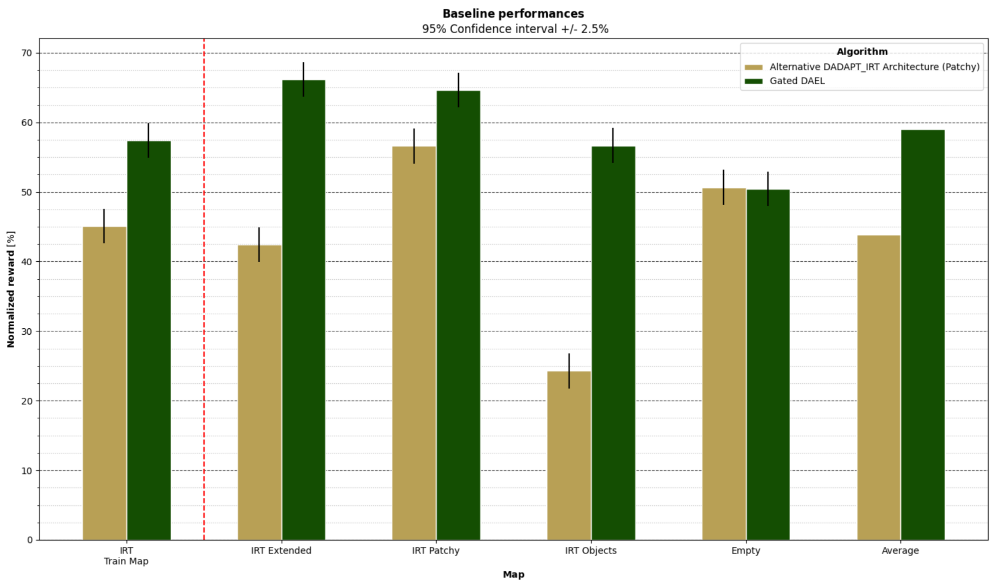


Figure 5.7: Comparing the anomaly in DADAPT-IRT to Gated DAEL. The performance is much higher than other iterations of the same algorithm, yet still lower than the imitation baseline and DAEL.

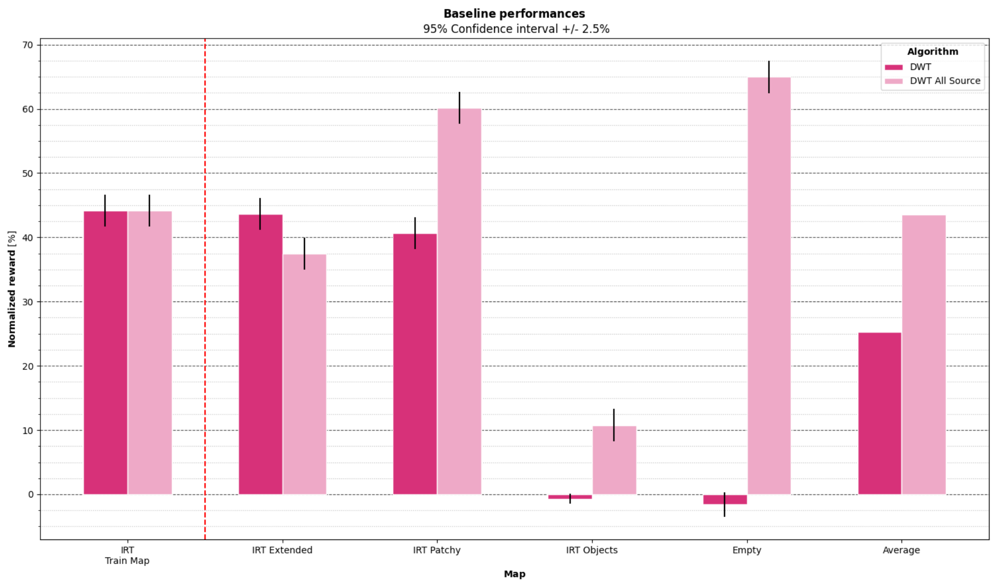


Figure 5.8: Running DWT while using the source transformation for the target yields improved performance. This goes against the intuition behind DWT and would seem to imply it does not apply to Duckietown

Chapter 6

Conclusions

6.1 Direct outputs

Beyond the research aims of the current project, we have developed an extensive infrastructure of software components that will aid future Machine Learning researchers interacting with Duckietown. We:

- Extended the base simulator with easily customisable and configurable observation and action transformations, allowing to quickly test a variety of environments
- Extended the base simulator with four new maps of increasing difficulty to test domain adaptation algorithms on
- Assembled, installed, and debugged a Duckietown platform for use at IRT Saint Exupéry
- Created a simple, containerized, and modular solution for future researchers wanting to interact with the Duckietown platform. This solution is simpler, and so far more reliable, than the original solution implemented by the Duckietown Foundation, and IRT is considering its open-sourcing to close partners or the research community at large

6.2 Research conclusions

We tested five different domain adaptation techniques to a new problem based on autonomous driving, that of domain adaptation and Simulation to Reality transfer within the Duckietown project. The results in environment adaptation are unsatisfying, with published methods under-performing with respect to a bare-bones baseline, and only one slightly surpassing it. The results in simulation to reality transfer were worse, with no satisfying driving being achieved. Our best results were achieved by some applications of data augmentation to our models.

This is despite the success of all 5 techniques in the problems to which they were applied in their original publications, including recognition of hand-written characters and image classification. We draw three main conclusions from this.

Firstly, we concur with Gulrajani et al's [33] findings - most published techniques cannot beat data augmentation. They point out that, in the field of domain generalization, which is closely related to that of domain adaptation, the reference published algorithms could not beat a baseline algorithm trained with adequate data augmentation. A finding not unlike ours.

Secondly, we believe that the published methods may be over specialized for a few data-sets of reference - which are usually a handful per task such as MNIST, USPS and SVHN for character recognition, or ImageNet and CIFAR for image classification. While these are extensive data-sets, they are narrow in terms of tasks, only covering classification in a very narrow sense. We wonder if their application to alternative problems, including regression problems, would not encounter the same issues we have encountered ourselves.

Thirdly, it is clear to us now that Domain Adaptation alone cannot cover the Simulation to Reality gap. The differences in dynamics can be very large, and indeed are in our case. It is clear from our simulation results, where dynamics are identical, that these are not the only factors holding our back. Nonetheless, our experiments in reality also show that the dynamic discrepancies are too large to be ignored.

6.3 Future work

Future work could be carried along two main axes according to two distinct objectives: Improving sim to real performance regardless of methods, or further researching the applications of domain adaptation in an end-to-end autonomous driving solution.

6.3.1 Improved Sim2Real

We believe the difficulty of achieving satisfying results in this task comes, in large part, due to attempting to perform end-to-end driving: Feeding raw pixel values to an ML algorithm and expecting direct wheel controls as an output. This is a considerably hard problem owing to the large dimensionality of the input and the low information content per dimension. Creating an ML model or RL agent to control the car given its position and orientation, for example, is likely to be much easier. Therefore, should improved results be the priority, we recommend to stop looking for an end to end solution and focus on a simpler problem, for example performing lane detection with ML then using classical motion planning and robust control techniques to perform autonomous driving on the extracted map. Furthermore, we would recommend using wider randomization, including dynamics randomization, to achieve results, and potentially combining it with generative models.

CAD2Real, one of the few projects to our knowledge achieving Sim2Real transfer, does both things. The agent here controls a quad-copter, and points at the desired direction attempting to predict collision free courses as opposed to directly actuating the quad-copter's motors. Furthermore, extensive data augmentation is used within its training in simulation, using a large variety of textures to overlay on objects and a wide range of circuits for the agent to fly in.

6.3.2 Further Domain Adaptation research

Should the interest be on further exploring our research, we would recommend doing what we initially intended to do: Apply domain adaptation methods to reinforcement learning. Exploring the interaction between the two could lead to interesting findings and, furthermore, confirm or prove wrong our conclusions from this project. It would be of particular interest to us to also explore how limited real-world roll-outs could be harnessed to adapt to the altered dynamic environment of reality.

Bibliography

[1] *Duckietown – Learning Autonomy.* URL: `https://www.duckietown.org/` (visited on 09/24/2020).

[2] *ROS.org | Powering the world's robots.* en-US. URL: `https://www.ros.org/` (visited on 09/24/2020).

[3] *Empowering App Development for Developers | Docker.* en. URL: `https://www.docker.com/` (visited on 09/24/2020).

[4] HypriotOS. *About Us · Docker Pirates ARMed with explosive stuff.* URL: `https://blog.hypriot.com/about/` (visited on 09/01/2020).

[5] Alexander Buslaev et al. "Albumentations: fast and flexible image augmentations". en. In: *Information* 11.2 (Feb. 2020). arXiv: 1809.06839, p. 125. ISSN: 2078-2489. DOI: `10.3390/info11020125`. URL: `http://arxiv.org/abs/1809.06839` (visited on 09/17/2020).

[6] David Poole and Alan Mackworth. *Artificial Intelligence: Foundations of Computational Agents, 2nd Edition.* 2nd. URL: `https://artint.info/2e/html/ArtInt2e.html` (visited on 09/03/2020).

[7] Tom Mitchell. *Machine Learning.* McGraw Hill, 1997.

[8] Christopher Thomas BSc Hons MIAP. *An introduction to Convolutional Neural Networks.* en. May 2019. URL: `https://towardsdatascience.com/an-introduction-to-convolutional-neural-networks-eb0b60b58fd7` (visited on 09/22/2020).

[9] David Silver. *Teaching.* en-GB. URL: `https://www.davidsilver.uk/teaching/` (visited on 09/07/2020).

[10] Richard Bellman. "The Theory of Dynamic Programming". In: *Bulletin of the American Mathematical Society* 60.6 (), pp. 503–515.

[11] David Silver et al. "Deterministic Policy Gradient Algorithms". en. In: (), p. 9.

[12] Scott Fujimoto, Herke van Hoof, and David Meger. "Addressing Function Approximation Error in Actor-Critic Methods". en. In: *arXiv:1802.09477 [cs, stat]* (Oct. 2018). arXiv: 1802.09477. URL: `http://arxiv.org/abs/1802.09477` (visited on 09/07/2020).

[13] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning". en. In: *arXiv:1509.02971 [cs, stat]* (July 2019). arXiv: 1509.02971. URL: `http://arxiv.org/abs/1509.02971` (visited on 09/07/2020).

[14] Xin Zhao. *zhaoxin94/awesome-domain-adaptation*. original-date: 2018-05-13T02:42:39Z. Sept. 2020. URL: `https://github.com/zhaoxin94/awesome-domain-adaptation` (visited on 09/23/2020).

[15] Werner Zellinger, Bernhard A. Moser, and Susanne Saminger-Platz. "On generalization in moment-based domain adaptation". In: *arXiv:2002.08260 [cs, stat]* (Aug. 2020). arXiv: 2002.08260. URL: `http://arxiv.org/abs/2002.08260` (visited on 09/25/2020).

[16] Dexuan Zhang and Tatsuya Harada. "A General Upper Bound for Unsupervised Domain Adaptation". In: *arXiv:1910.01409 [cs, stat]* (Oct. 2019). arXiv: 1910.01409. URL: `http://arxiv.org/abs/1910.01409` (visited on 09/25/2020).

[17] Trung Le et al. "On Deep Domain Adaptation: Some Theoretical Understandings". In: *arXiv:1811.06199 [cs, stat]* (June 2019). arXiv: 1811.06199. URL: `http://arxiv.org/abs/1811.06199` (visited on 09/25/2020).

[18] Eric Tzeng et al. "Deep Domain Confusion: Maximizing for Domain Invariance". In: *arXiv:1412.3474 [cs]* (Dec. 2014). arXiv: 1412.3474. URL: `http://arxiv.org/abs/1412.3474` (visited on 09/23/2020).

[19] Baochen Sun, Jiashi Feng, and Kate Saenko. "Return of Frustratingly Easy Domain Adaptation". In: *arXiv:1511.05547 [cs]* (Dec. 2015). arXiv: 1511.05547. URL: `http://arxiv.org/abs/1511.05547` (visited on 09/23/2020).

[20] Subhankar Roy et al. "Unsupervised Domain Adaptation Using Feature-Whitening and Consensus Loss". en. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Long Beach, CA, USA: IEEE, June 2019, pp. 9463–9472. ISBN: 978-1-72813-293-8. DOI: `10.1109/CVPR.2019.00970`. URL: `https://ieeexplore.ieee.org/document/8953769/` (visited on 09/18/2020).

[21] Fabio Maria Carlucci et al. "AutoDIAL: Automatic DomaIn Alignment Layers". In: *arXiv:1704.08082 [cs]* (Nov. 2017). arXiv: 1704.08082. URL: `http://arxiv.org/abs/1704.08082` (visited on 09/23/2020).

[22] Muhammad Ghifary et al. "Deep Reconstruction-Classification Networks for Unsupervised Domain Adaptation". In: *arXiv:1607.03516 [cs, stat]* (Aug. 2016). arXiv: 1607.03516. URL: `http://arxiv.org/abs/1607.03516` (visited on 09/18/2020).

[23] Yaroslav Ganin et al. "Domain-Adversarial Training of Neural Networks". In: *arXiv:1505.07818 [cs, stat]* (May 2016). arXiv: 1505.07818. URL: `http://arxiv.org/abs/1505.07818` (visited on 09/18/2020).

[24] Ashish Shrivastava et al. “Learning from Simulated and Unsupervised Images through Adversarial Training”. In: *arXiv:1612.07828 [cs]* (July 2017). arXiv: 1612.07828. URL: `http://arxiv.org/abs/1612.07828` (visited on 09/23/2020).

[25] Paolo Russo et al. “From source to target and back: symmetric bi-directional adaptive GAN”. In: *arXiv:1705.08824 [cs]* (Nov. 2017). arXiv: 1705.08824. URL: `http://arxiv.org/abs/1705.08824` (visited on 09/23/2020).

[26] Antti Tarvainen and Harri Valpola. “Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results”. In: *arXiv:1703.01780 [cs, stat]* (Apr. 2018). arXiv: 1703.01780. URL: `http://arxiv.org/abs/1703.01780` (visited on 09/23/2020).

[27] Kaiyang Zhou et al. “Domain Adaptive Ensemble Learning”. In: *arXiv:2003.07325 [cs]* (Mar. 2020). arXiv: 2003.07325. URL: `http://arxiv.org/abs/2003.07325` (visited on 09/18/2020).

[28] fungtion. *fungtion/DRCN*. original-date: 2018-03-14T15:28:50Z. Aug. 2020. URL: `https://github.com/fungtion/DRCN` (visited on 09/18/2020).

[29] Noam Shazeer et al. “Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer”. en. In: *arXiv:1701.06538 [cs, stat]* (Jan. 2017). arXiv: 1701.06538. URL: `http://arxiv.org/abs/1701.06538` (visited on 09/21/2020).

[30] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. en. In: *arXiv:1312.5602 [cs]* (Dec. 2013). arXiv: 1312.5602. URL: `http://arxiv.org/abs/1312.5602` (visited on 09/17/2020).

[31] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. en. In: *arXiv:1502.03167 [cs]* (Mar. 2015). arXiv: 1502.03167. URL: `http://arxiv.org/abs/1502.03167` (visited on 09/17/2020).

[32] Itzik Malkiel and Lior Wolf. “MTAdam: Automatic Balancing of Multiple Training Loss Terms”. In: *arXiv:2006.14683 [cs, stat]* (June 2020). arXiv: 2006.14683. URL: `http://arxiv.org/abs/2006.14683` (visited on 10/20/2020).

[33] Ishaan Gulrajani and David Lopez-Paz. “In Search of Lost Domain Generalization”. en. In: *arXiv:2007.01434 [cs, stat]* (July 2020). arXiv: 2007.01434. URL: `http://arxiv.org/abs/2007.01434` (visited on 10/20/2020).